Параллельные и высокопроизводительные вычисления

Роберт Роби
Джулиана Замора

MANNING
ДМК ПРЕСС издательство

Роберт Роби и Джулиана Замора

Параллельные и высокопроизводительные вычисления

Parallel and High Performance Computing

ROBERT (BOB) ROBEY
and YULIANA (YULIE) ZAMORA

MANNING
Shelter Island

Параллельные и высокопроизводительные вычисления

РОБЕРТ РОБИ
и ДЖУЛИАНА ЗАМОРА

ДМК Пресс
издательство
Москва, 2022

УДК 004.421
ББК 32.972
Р58

Роби Р., Замора Дж.
Р58 Параллельные и высокопроизводительные вычисления / пер. с англ. А. В. Логунова. – М.: ДМК Пресс, 2021. – 800 с.: ил.

ISBN 978-5-97060-936-1

Параллельное программирование позволяет распределять задачи обработки данных между несколькими процессорами, существенно повышая производительность. В книге рассказывается, как с минимальными трудозатратами повысить эффективность ваших программ. Вы научитесь оценивать аппаратные архитектуры и работать со стандартными инструментами отрасли, такими как OpenMP и MPI, освоите структуры данных и алгоритмы, подходящие для высокопроизводительных вычислений, узнаете, как экономить энергию на мобильных устройствах, и даже запустите масштабную симуляцию цунами на батарее из GPU-процессоров.

Издание предназначено для опытных программистов, владеющих языком высокопроизводительных вычислений, таким как C, C++ или Fortran.

УДК 004.421
ББК 32.972

ISBN 978-1-6172-9646-8 (англ.)
ISBN 978-5-97060-936-1 (рус.)

Моей жене Пегги, которая поддерживала не только мой путь в области высокопроизводительных вычислений, но и нашего сына Джона и дочь Рейчел. Научное программирование далеко от ее медицинских знаний, но она сопровождала меня и совершила это наше путешествие с самого начала. Моему сыну Джону и дочери Рейчел, которые вновь разожгли во мне пламя, и за ваше многообещающее будущее.

– Боб Роби

Моему мужу Рику, который поддерживал меня всю дорогу, спасибо, что брал на себя утренние смены и позволял мне работать по ночам. Ты никогда не позволял мне отказываться от самой себя. Моим родителям и родственникам, спасибо за всю вашу помощь и поддержку. И моему сыну Дереку за то, что он был одним из моих самых больших вдохновителей; ты – вся причина, почему я не просто живу, я наслаждаюсь жизнью.

– Джули Замора

Оглавление

Содержание

Предисловие

От авторов

Боб Роби, Лос-Аламос, Нью-Мексико

> *Выходить за дверь – опасно, Фродо. Выходишь на дорогу, и, если не удержаться на ногах, неизвестно, куда тебя унесет.*
>
> – Бильбо Бэггинс

Я не мог предвидеть, куда нас приведет это путешествие в параллельные вычисления. Я говорю «мы», потому что на протяжении многих лет это путешествие делили со мной многочисленные коллеги. Мое же путешествие в параллельные вычисления началось в начале 1990-х годов, когда я учился в Университете Нью-Мексико. Я написал несколько программ по динамике сжимаемой жидкости для симулирования экспериментов с ударной трубкой и запускал их в каждой системе, которая попадала мне в руки. В результате меня вместе с Брайаном Смитом (Brian Smith), Джоном Соболевски (John Sobolewski) и Фрэнком Гилфезером (Frank Gilfeather) попросили представить предложение по созданию центра высокопроизводительных вычислений. Мы выиграли грант и в 1993 году создали Центр высокопроизводительных вычислений Maui Мое участие в проекте состояло в том, чтобы предлагать курсы и возглавлять подготовку 20 аспирантов по разработке параллельных вычислений в Университете Нью-Мексико в Альбукерке.

1990-е годы были временем становления параллельных вычислений. Я помню выступление Эла Гейста (Al Geist), одного из первых разработчиков параллельной виртуальной машины (PVM) и члена комитета по стандартам MPI[1], как говорил о готовящемся к выпуску стандарте MPI

[1] Message Passing Interface (MPI) – программный интерфейс для передачи информации, который позволяет обмениваться сообщениями между процессами, выполняющими одну задачу. – *Прим. перев.*

(июнь 1994 года). Он сказал тогда, что он никуда не приведет, потому что он слишком сложен. Насчет сложности Эл был прав, но, несмотря на это, он взмыл ввысь, и в течение нескольких месяцев он уже применялся почти во всех параллельных приложениях. Одна из причин успеха MPI заключается в наличии готовых имплементаций. Аргоннская национальная лаборатория разрабатывала Chameleon, инструмент переносимости, который в то время транслировал языки передачи сообщений, включая P4, PVM, MPL и многие другие. Этот проект был быстро изменен на MPICH, став первой высококачественной имплементацией MPI. На протяжении более десяти лет MPI стал синонимом параллельных вычислений. Почти каждое параллельное приложение было построено поверх библиотек MPI.

Теперь давайте перенесемся в 2010 год и увидим появление графических процессоров, GPU. Как-то я наткнулся на статью в журнале Dr. Dobb об использовании суммы Кахана для компенсировпания единственной арифметики с одинарной точностью, доступной на GPU. Я подумал, что, возможно, этот подход мог бы помочь решить давнюю проблему параллельных вычислений, когда глобальная сумма массива меняется в зависимости от числа процессоров. Намереваясь проверить это на практике, я подумал о программном коде по гидродинамике, который мой сын Джон написал в средней школе. Он тестировал сохранение массы и энергии в задаче с течением времени и останавливался, выходя из программы, если масса изменялась более чем на указанную величину. Пока он был дома на весенних каникулах после своего первого курса в Вашингтонском университете, мы опробовали этот метод и были приятно удивлены тем, насколько улучшилось сохранение массы. Для производственных исходных кодов влияние этого простого технического приема окажется важным. Мы рассмотрим алгоритм суммирования с повышенной точностью для параллельных глобальных сумм в разделе 5.7 этой книги.

В 2011 году я организовал летний проект с тремя студентами, Нилом Дэвисом (Neal Davis), Дэвидом Николаеффом (David Nicholaeff) и Деннисом Трухильо (Dennis Trujillo), чтобы попробовать, сможем ли разработать более сложные программные коды, такие как адаптивная детализация расчетной сетки (AMR) и неструктурированные произвольные лагранжево–эйлеровы (ALE) приложения для работы на GPU. Результатом стало CLAMR, мини-приложение AMR, которое полностью работало на GPU. Бóльшая часть приложения была легко переносима. Самой сложной его частью было определение соседа для каждой ячейки. В изначальном CPU-коде использовался алгоритм k-D-дерева, но алгоритмы, основанные на дереве, трудно переносятся на GPU. Через две недели после начала летнего проекта на холмах над Лос-Аламосом вспыхнул пожар в Лас-Кончасе, и город был эвакуирован. Мы уехали в Санта-Фе, и студенты разбежались. Во время эвакуации я встретился с Дэвидом Николаеффом в центре Санта-Фе, чтобы обсудить вопрос переноса на GPU. Он предложил нам попробовать использовать алгоритм хеширования, чтобы заменить древесный исходный код отыскания соседей. В то время я наблюдал за пылающим над городом огнем, и меня не отпускало

беспокойство, что огонь доберется до моего дома. Несмотря на все эти перипетии, я согласился попробовать, и алгоритм хеширования привел к тому, что весь код был запущен на GPU. Техника хеширования была обобщена Дэвидом, моей дочерью Рейчел, когда она училась в средней школе, и мной. Эти алгоритмы хеширования формируют основу для многих алгоритмов, представленных в главе 5.

В последующие годы технические приемы компактного хеширования были разработаны Ребеккой Тамблин (Rebecka Tumblin), Питером Аренсом (Peter Ahrens) и Сарой Харце (Sara Hartse). Более сложная задача компактного хеширования для операций перекомпоновывания (remapping) на CPU и GPU была решена Джеральдом Коломом (Gerald Collom) и Колином Редманом (Colin Redman), когда они только что закончили среднюю школу. Этими прорывами в параллельных алгоритмах для GPU были устранены препятствия для выполнения многих научных приложений на GPU.

В 2016 году я начал программу летней исследовательской стажировки по параллельным вычислениям (PCSRI) в Национальной лаборатории Лос-Аламоса (LANL) вместе с моими соучредителями Хай А Нам (Hai Ah Nam) и Гейбом Рокфеллером (Gabe Rockefeller). Целью программы параллельных вычислений было решение проблемы растущей сложности систем высокопроизводительных вычислений. Программа представляет собой 10-недельную летнюю стажировку с лекциями по различным темам параллельных вычислений с последующим исследовательским проектом под руководством сотрудников Национальной лаборатории Лос-Аламоса. У нас в летней программе участвовало от 12 до 18 студентов, и многие использовали ее в качестве трамплина для своей карьеры. С помощью этой программы мы продолжаем решать некоторые новейшие задачи, стоящие перед параллельными и высокопроизводительными вычислениями.

Джули Замора, Чикагский университет, Иллинойс

> *Если есть книга, которую вы хотите прочитать, но она еще не написана, то вы должны ее написать.*
>
> – Тони Моррисон

Мое знакомство с параллельными вычислениями началось со слов: «Перед тем как начнете, зайдите в комнату в конце 4-го этажа и установите вот эти процессоры Knights Corner в нашем кластере». Эта просьба профессора Корнельского университета побудила меня попробовать что-то новое. То, что я считала простым делом, превратилось в бурное путешествие в высокопроизводительные вычисления. Я начала с изучения основ – с выяснения физического принципа функционирования малого кластера, поднимая 40-фунтовые сервера для работы с BIOS и выполнения моего первого приложения, а затем оптимизации этих приложений по всем установленным мной узлам.

После короткого семейного перерыва, каким бы обескураживающим он ни был, я подала заявление на научно-исследовательскую стажиров-

ку. Будучи принятым в первую программу летней исследовательской стажировки по параллельным вычислениям в Нью-Мексико, я получила возможность разведать тонкости параллельных вычислений на современном оборудовании, и именно там я познакомилась с Бобом. Я пришла в восторг от повышения производительности, которое было возможно при наличии лишь небольших знаний о правильном написании параллельного кода. Я лично провела разведывательный анализ процесса написания более эффективного кода OpenMP. Мое волнение и прогресс в оптимизации приложений открыли двери для других возможностей, таких как участие в конференциях и представление моей работы на собрании группы пользователей Intel и на стенде Intel по суперкомпьютерным вычислениям. В 2017 году меня также пригласили принять участие и выступить на конференции в Салишане. Это была прекрасная возможность обменяться идеями с несколькими ведущими провидцами высокопроизводительных вычислений.

Еще одним замечательным опытом была подача заявки на участие в хакатоне GPU и участие в нем. На хакатоне мы перенесли код на OpenACC, и в течение недели этот код был ускорен в 60 раз. Вы только подумайте – расчет, который раньше занимал месяц, теперь может выполняться за одну ночь. Полностью погрузившись в потенциал долгосрочных исследований, я подала заявление в аспирантуру и выбрала Чикагский университет, осознавая его тесную связь с Аргоннской национальной лабораторией. В Чикагском университете меня консультировали Ян Фостер (Ian Foster) и Генри Хоффман (Henry Hoffmann).

Из своего опыта я поняла, насколько ценным является личное взаимодействие, когда учишься писать параллельный код. Я также была разочарована отсутствием учебника или справочника, в котором обсуждалось бы текущее оборудование. В целях восполнения этого пробела мы написали эту книгу, чтобы сделать ее намного проще для тех, кто только начинает свою деятельность в параллельных и высокопроизводительных вычислениях. Решение задачи создания и преподавания введения в информатику для поступающих студентов Чикагского университета помогло мне получить представление о тех, кто знакомится с этой областью впервые. С другой стороны, объяснение технических приемов параллельного программирования в качестве ассистента преподавателя в курсе «Продвинутые распределенные системы» позволило мне работать со студентами с более высоким уровнем понимания. Оба этих опыта помогли мне обрести способность объяснять сложные темы на разных уровнях.

Я считаю, что у каждого должна быть возможность изучить этот важный материал по написанию высокопроизводительного кода и что он должен быть легко доступен для всех. Мне посчастливилось иметь наставников и советников, которые направляли меня по правильным ссылкам на веб-сайты или передавали мне свои старые рукописи для чтения и изучения. Хотя некоторые технические приемы, возможно, будут сложными, более серьезной проблемой является отсутствие согласованной документации или доступа к ведущим ученым в этой области в качестве наставников. Я понимаю, что не у всех есть одинаковые ре-

сурсы, и поэтому я надеюсь, что создание этой книги заполнит пустоту, которая существует в настоящее время.

Как мы пришли к написанию этой книги

Начиная с 2016 года команда ученых LANL во главе с Бобом Роби разработала лекционные материалы для летней исследовательской стажировки по параллельным вычислениям (PCSRI) в Лос-Аламосской национальной лаборатории (LANL). Большая часть этого материала посвящена новейшему оборудованию, которое быстро выходит на рынок. Параллельные вычисления меняются быстрыми темпами, и к ним прилагается мало документации. Книга, охватывающая эти материалы, была явно необходима. Именно в этот момент издательство Manning связалось с Бобом по поводу написания книги о параллельных вычислениях. У нас был лишь черновой набросок материалов, и работа над ним вылилась в двухлетние усилия по переводу всего этого в формат высокого качества.

Темы и план глав были четко определены на ранней стадии и основывались на лекциях для нашей летней программы. Многие идеи и технические приемы были позаимствованы из более широкого сообщества высокопроизводительных вычислений, поскольку мы стремимся к более высокому уровню вычислений – тысячекратному повышению производительности вычислений по сравнению с предыдущим эталоном петафлопсного масштаба. Это сообщество включает Центры передового опыта Министерства энергетики (DOE), проект по экзомасштабным вычислениям (Exascale Computing Project) и серию семинаров по производительности, переносимости и производительности. Широта и глубина материалов наших компьютерных лекций отражают глубокие проблемы сложных гетерогенных вычислительных архитектур.

Мы называем материал этой книги «глубоким введением». Она начинается с основ параллельных и высокопроизводительных вычислений, но без знания вычислительной архитектуры невозможно достичь оптимальной производительности. По ходу дела мы стараемся осветить проблематику на более глубоком уровне понимания, потому что недостаточно просто двигаться по тропе, не имея ни малейшего представления о том, где вы находитесь или куда направляетесь. Мы предоставляем инструменты для разработки карты и для того, чтобы показать удаленность цели, к которой мы стремимся.

В начале этой книги Джо Шоновер (Joe Schoonover) был привлечен для написания материалов, связанных с GPU, а Джули Замора – главы по OpenMP. Джо предоставил дизайн и компоновку разделов по GPU, но ему пришлось быстро отказаться. Джули написала статьи и предоставила массу иллюстраций о том, как OpenMP вписывается в этот дивный новый мир масштабных вычислений, поэтому указанный материал особенно хорошо подошел для главы книги об OpenMP. Глубокое понимание Джули проблем, связанных с экзомасштабными вычислениями, и ее способность разбирать их по полочкам для новичков в этой области стали решающим вкладом в создание этой книги.

Благодарности

Мы хотели бы поблагодарить всех, кто помог сформировать эту книгу. Первым в нашем списке стоит Джо Шоновер из Fluid Dynamics (Гидродинамики жидкости), который сделал все возможное в преподавании параллельных вычислений, в особенности с GPU-процессорами. Джо был одним из соруководителей нашей программы параллельных вычислений и сыграл важную роль в формулировании того, что должна охватывать эта книга. Другие наши соучредители, Хай А Нам, Гейб Рокфеллер, Крис Гарретт, Юнмо Ку, Люк Ван Рокель, Роберт Берд, Джонас Липпунер и Мэтт Тернер, внесли свой собственный вклад в успех школы параллельных вычислений и ее содержание. Создание летней программы параллельных вычислений не произошло бы без поддержки и видения директора института, Стефана Эйденбенца. Также спасибо Скотту Раннелсу и Даниэлю Израэлю, которые возглавили летнюю школу вычислительной физики LANL и стали пионерами концепции школы, дав нам модель для подражания.

Нам повезло, что нас окружают эксперты в области параллельных вычислений и книгоиздания. Благодарим Кейт Боуман, чей опыт в написании книг помогал вносить изменения в первые главы. Кейт – невероятно талантлива во всех аспектах издательского дела и уже много лет занимается составлением предметных указателей книг. У нас также были неофициальные отзывы от сына Боба Джона, дочери Рэйчел и зятя Боба Берда, часть технической работы каждого из которых упоминается в книге. Муж Джули, Рик, помог предоставить экспертные знания по некоторым темам, а Дов Шлахтер рассмотрел некоторые ранние наброски и предоставил несколько полезных отзывов.

Мы также хотели бы отметить опыт сотрудников, которые приложили свои знания в конкретных главах. Сюда входят Рао Гаримелла и Шейн Фогерти из Лос-Аламоса и Мэтт Мартино из Национальной лаборатории Лоуренса Ливермора, чья работа включена в главу 4. Особая благодарность выражается инновационным работам упомянутых ранее многих студентов, чья работа занимает бо́льшую часть главы 5. Рон Грин из Intel в течение нескольких лет руководил усилиями по документированию

приемов использования векторизации, предоставляемой компилятором Intel, что легло в основу главы 6. Симуляция цунами в главе 13 была разработана командой средней школы Маккарди, в состав которой входили Сара Армстронг, Джозеф Коби, Хуан-Антонио Виджил и Ванесса Трухильо, участвовавшие в конкурсе по суперкомпьютерным вычислениям в Нью-Мексико в 2007 году. Также спасибо Кристиану Гомесу за помощь в иллюстрировании цунами. Работа по местам размещения процессов и их аффинности с Дугом Якобсеном из Intel и Хай А Нам и Сэмом Гутьерресом из Лос-Аламосской национальной лаборатории заложила основу для главы 14. Кроме того, работа с командой Datalib Галена Шипмана и Брэда Сеттлмайера из Национальной лаборатории Лос-Аламоса, Роба Росса, Роба Латама, Фила Карнса, Шейна Снайдера из Аргоннской национальной лаборатории и Вэй-Кенга Ляо из Северо-Западного университета отражена в главе 16 и разделе, посвященном инструменту Darshan для профилирования файловых операций, в главе 17.

Мы также ценим усилия профессионалов издательства Manning в создании более отточенного и профессионального продукта. Наш редактор, Фрэнсис Буран, проделала замечательную работу, усовершенствовав текст и сделав его более читабельным. Она владеет высокотехничным и точным языком и делает это в удивительном темпе. Также спасибо Дейдре Хиам, нашему производственному редактору, за превращение графики, формул и текста в отшлифованный продукт для наших читателей. Мы также хотели бы поблагодарить Джейсона Эверетта, нашего корректора. Пол Уэллс, менеджер по производству книги, держал все эти усилия в строгом графике.

Издательство Manning встраивает в процесс написания книги многочисленные рецензии, включая стиль написания, редактирование, корректуру и техническое содержание. Прежде всего это редактор отдела закупок Manning Майк Стивенс, который увидел необходимость в книге на эту тему. Наш редактор по разработке, Марина Майклс, помогла нам не сбиться с этого огромного по вкладываемым усилиям проекта. Марина была особенно полезна в том, чтобы сделать материал более доступным для широкой аудитории. Кристофер Хаупт, редактор отдела технической разработки, дал нам ценные отзывы о техническом содержании. Мы особенно благодарим Туана Трана, нашего технического корректора, который просмотрел исходный код всех примеров. Туан проделал огромную работу по преодолению трудностей, связанных с трудностями конфигураций программного и аппаратного обеспечения высокопроизводительных вычислений. Наш редактор рецензий, Александр Драгошавлевич, набрал отличный коллектив рецензентов, который охватил широкий круг читателей. Эти рецензенты, Ален Конниот, Альберт Чой, Алессандро Кампейс, Анджело Коста, Арав Капиш Агарвал, Дана Робинсон, Доминго Салазар, Уго Дюрана, Жан-Франсуа Морин, Патрик Риган, Филипп Г. Брэдфорд, Ричард Тобиас, Роб Кьелти, Срджан Сантич, Туан А. Тран и Винсент Доуве, дали нам ценные отзывы, которые существенно улучшили конечный продукт.

О книге

Одна из самых важных задач для исследователя-первопроходца состоит в том, чтобы начертить карту для тех, кто последует за ним. Это особенно верно для тех из нас, кто раздвигает границы науки и техники. Нашей целью в этой книге было предложение дорожной карты для тех, кто только начинает изучать параллельные и высокопроизводительные вычисления, а также для тех, кто хочет расширить свои знания в этой области. Высокопроизводительные вычисления – это быстро эволюционирующая область, где языки и технологии находятся в постоянном потоке. По этой причине мы сосредоточимся на фундаментальных принципах, которые остаются неизменными с течением времени. В компьютерных языках для CPU и GPU мы подчеркиваем общие закономерности во многих языках, чтобы дать вам возможность быстро выбирать наиболее подходящий язык для вашей текущей задачи.

Кто должен прочитать эту книгу

Эта книга предназначена как для студентов продвинутых курсов по параллельным вычислениям, так и в качестве современной литературы для специалистов в области вычислительной техники. Если вас интересует производительность, будь то время выполнения, масштаб или мощность, то эта книга предоставит вам инструменты для усовершенствования вашего приложения и повышения вашей конкурентноспособности. Имея процессоры, которые достигают пределов масштаба, тепла и мощности, мы не можем рассчитывать на компьютер следующего поколения в деле ускорения наших приложений. Все чаще высококвалифицированные и знающие программисты имеют решающее значение для достижения максимальной производительности современных приложений.

В этой книге мы надеемся изложить ключевые идеи, применимые к современному оборудованию высокопроизводительных вычислений. Они являются базовыми истинами программирования для повышения производительности. Указанные темы лежат в основе всей книги.

В высокопроизводительных вычислениях важно не то, как быстро вы пишете код, а то, как быстро выполняется написанный вами код.

Эта мысль подводит итог тому, что значит писать приложения для высокопроизводительных вычислений. В большинстве других приложений основное внимание уделяется быстроте написания приложения. Сегодня компьютерные языки, как правило, предназначены для ускорения программирования, а не для повышения производительности кода. Хотя этот подход к программированию уже давно используется в приложениях высокопроизводительных вычислений, он не был широко задокументирован или описан. В главе 4 мы обсудим этот другой фокус внимания в методологии программирования, который недавно получил свой термин «дизайн с ориентацией на данные».

Все дело в памяти: сколько вы ее используете и как часто загружаете.

Даже если вы знаете, что доступная память и операции с памятью почти всегда являются лимитирующим фактором производительности, мы все равно склонны тратить много времени на размышления об операциях с плавающей точкой. При наличии способности большинства современных вычислительных устройств выполнять 50 операций с плавающей точкой для каждой загрузки в память операции с плавающей точкой являются второстепенным вопросом. Почти в каждой главе мы используем нашу имплементацию потокового сравнительного теста (STREAM benchmark), теста производительности памяти с целью верификации получения нами разумной производительности от оборудования и языка программирования.

Если вы загружаете одно значение, то получаете восемь или шестнадцать.

Это все равно что покупать яйца. Невозможно взять только одно. Загрузка в память выполняется строками кеша по 512 бит. Для 8-байтового значения двойной точности будет загружено восемь значений, хотите вы этого или нет. В целях достижения наилучшей производительности следует планировать так, чтобы ваша программа использовала более одного значения, предпочтительно восемь смежных значений. И пока вы этим занимаетесь, она использовала остальные яйца.

Если в вашем коде есть какие-либо изъяны, то параллелизация их проявит.

Качество кода требует большего внимания при высокопроизводительных вычислениях, чем в сопоставимом последовательном приложении. Это относится ко всем этапам: до начала параллелизации, во время и после. При параллелизации вы с большей вероятностью запустите в своей программе изъян, а также обнаружите, что проводить отлаживание будет сложно, в особенности в крупном масштабе. Мы представляем

технические приемы повышения качества программного обеспечения в главе 2, затем на протяжении всех глав упоминаем важные инструменты, и, наконец, в главе 17 мы перечисляем другие инструменты, которые могут оказаться полезными.

Эти ключевые темы выходят за рамки типов оборудования, одинаково применимых ко всем CPU и GPU. Они существуют из-за текущих физических ограничений, накладываемых на аппаратное обеспечение.

Как эта книга организована: дорожная карта

Эта книга не предполагает, что вы обладаете какими-либо знаниями в области параллельного программирования. Ожидается, что читатели являются опытными программистами, предпочтительно на компилируемом языке высокопроизводительных вычислений, таком как C, C++ или Fortran. Также ожидается, что читатели будут обладать некоторыми знаниями в области компьютерной терминологии, основ операционных систем и сетей. Читатели также должны иметь возможность ориентироваться на своем компьютере, включая инсталлирование программного обеспечения и легкие задачи системного администрирования.

Знание компьютерного оборудования, пожалуй, является самым важным требованием к читателям. Мы рекомендуем открыть корпус вашего компьютера, изучить каждый компонент и получить представление о его физических характеристиках. Если вы не можете открыть корпус своего компьютера, то рекомендуем посмотреть фотографии типичной настольной системы в конце приложения C к книге. Например, посмотрите на нижнюю часть процессора на рис. C.2 и на лес контактных штырьков, входящих в чип. Вам не удастся вставить туда даже булавку. Теперь вы можете понять, почему существует физический предел того, сколько данных может быть передано процессору из других частей системы. Рекомендуем возвращаться к этим фотографиям и глоссарию в приложении A, если у вас возникает необходимость лучше понять компьютерное оборудование или компьютерную терминологию.

Мы разделили эту книгу на четыре части, которые охватывают мир высокопроизводительных вычислений. Указанные части таковы:

- часть I «Введение в параллельные вычисления» (главы 1–5);
- часть II «Технологии центрального процессора (CPU)» (главы 6–8);
- часть III «Технологии графического процессора (GPU)» (главы 9–13);
- часть IV «Экосистемы высокопроизводительных вычислений (HPC)» (главы 14–17).

Порядок тем ориентирован на тех, кто занимается проектом, связанным с высокопроизводительными вычислениями. Например, для прикладного проекта темы разработки программного обеспечения в главе 2 необходимы перед запуском проекта. По завершении разработки программного обеспечения следующими техническими решениями станут структуры данных и алгоритмы. Затем идут имплементации для CPU и GPU. Наконец, приложение адаптируется для параллельной файловой

системы и других уникальных характеристик системы высокопроизводительных вычислений.

С другой стороны, некоторые из наших читателей больше заинтересованы в приобретении фундаментальных навыков параллельного программирования и, возможно, захотят перейти непосредственно к главам MPI или OpenMP. Но не стоит останавливаться на достигнутом. Сегодня существует гораздо больше возможностей для параллельных вычислений. От GPU, которые могут ускорять ваше приложение на порядок, до инструментов, которые могут улучшать качество вашего кода или указывать разделы кода, которые подлежат оптимизации, – потенциальные выгоды ограничены только вашим временем и опытом.

Если вы используете эту книгу для занятий по параллельным вычислениям, объема материала хватит как минимум на два семестра. Вы можете рассматривать эту книгу как коллекцию материалов, которые могут быть подобраны индивидуально под аудиторию. Выбрав темы для изучения, вы можете настроить его для своих собственных курсовых целей. Вот возможная последовательность изложения материала:

- глава 1 содержит введение в параллельные вычисления;
- в главе 3 рассматриваются подходы к измерению производительности оборудования и приложений;
- разделы 4.1-4.2 описывают концепцию программирования, ориентированного на данные, многомерные массивы и основы кеширования;
- глава 7 посвящена OpenMP (открытой мультипроцессорной обработке) для обеспечения параллелизма на узлах;
- в главе 8 рассматривается MPI (Интерфейс передачи сообщений) для обеспечения распределенного параллелизма между несколькими узлами;
- в разделах 14.1–14.5 представлены концепции аффинности и мест размещения процессов;
- главы 9 и 10 описывают оборудование GPU и модели программирования;
- разделы 11.1-11.2 посвящены OpenACC для обеспечения работы приложений на GPU.

Вы можете добавить в этот список такие темы, как алгоритмы, векторизация, параллельная обработка файлов или несколько других языков GPU. Или же удалить какую-то тему, чтобы потратить больше времени на остальные темы. Еще есть дополнительные главы, которые побудят студентов продолжить самостоятельное разведывание мира параллельных вычислений.

Об исходном коде

Невозможно научиться параллельным вычислениям, не написав код и не выполнив его. Для этой цели мы приводим большой набор примеров, прилагаемых к этой книге. Примеры находятся в свободном доступе

по адресу https://github.com/EssentialsOfParallelComputing. Вы можете скачать эти примеры в виде полного комплекта либо по отдельности по главам.

Также весь используемый в этой книге исходный код для книг издательства «ДМК Пресс» можно найти на сайте www.dmkpress.com или www.дмк.рф на странице с описанием соответствующей книги.

Учитывая объем примеров, аппаратное и программное обеспечение, в прилагаемых примерах неизбежно будут присутствовать изъяны и ошибки. Если вы обнаружите что-то, что является ошибкой, или просто пример будет неполным, то рекомендуем внести свой вклад в эти примеры. Мы уже объединили несколько ценных запросов на внесение изменений, поступивших от читателей. Кроме того, хранилище исходного кода будет наилучшим местом для поиска исправлений и обсуждений исходного кода.

Требования к программному/аппаратному обеспечению

Возможно, самой большой проблемой параллельных и высокопроизводительных вычислений является широкий спектр используемого аппаратного и программного обеспечения. В прошлом эти специализированные системы были доступны только на специфичных веб-сайтах. В последнее время аппаратное и программное обеспечение стало более демократичным и широко доступным даже на уровне настольных компьютеров или ноутбуков. Произошел существенный сдвиг, который значительно упрощает разработку программного обеспечения для высокопроизводительных вычислений. Однако настройка аппаратной и программной среды является самой сложной частью задачи. Если у вас есть доступ к параллельно-вычислительному кластеру, где они уже настроены, то мы рекомендуем вам воспользоваться этим преимуществом. В конце концов вы, возможно, захотите настроить свою собственную систему. Примеры проще всего использовать в системах Linux или Unix, но во многих случаях они должны работать и в Windows, и macOS при некоторых дополнительных усилиях. На случай если вы обнаружите, что пример не запускается в вашей системе, мы предоставили альтернативы заготовкам контейнеров Docker и настроечным скриптам VirtualBox.

Для выполнения упражнений с GPU требуются GPU разных производителей, включая NVIDIA, AMD Radeon и Intel. Любой, кто пытался установить графические драйверы GPU в своей системе, не удивится, что они представляют наибольшую трудность в настройке вашей локальной системы для примеров. Некоторые языки GPU также могут работать на CPU, позволяя разрабатывать код в локальной системе для оборудования, которого у вас нет. Вы также, возможно, обнаружите, что отладка на CPU проходит проще. Но в целях ознакомления с реальной производительностью вам потребуется реальное оборудование GPU.

Другие примеры, требующие специальной инсталляции, включают примеры пакетной системы и параллельных файлов. Для пакетной системы требуется больше, чем один ноутбук или рабочая станция, чтобы

инсталляция выглядела как настоящая. Аналогичным образом примеры параллельных файлов лучше всего работают со специализированной файловой системой, такой как Lustre, хотя базовые примеры будут работать на ноутбуке или рабочей станции.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге, – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Manning Publications очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Дискуссионный форум liveBook

Покупка книги «Параллельные и высокопризводительные вычисления» включает в себя бесплатный доступ к приватному веб-форуму издательства Manning Publications, где вы можете комментировать книгу, зада-

вать технические вопросы и получать помощь от авторов и других пользователей. В целях получения доступа к указанному форуму перейдите по ссылке https://livebook.manning.com/#!/book/parallel-and-high-performance-computing/discussion. О форумах издательства Manning и правилах поведения можно узнать больше на веб-странице https://livebook.manning.com/#!/discussion.

Другие онлайновые ресурсы

Публикации Manning также предоставляют онлайновый дискуссионный форум под названием livebook для каждой книги. Наш веб-сайт находится по адресу https://livebook.manning.com/book/parallel-and-high-performance-computing. Это самое подходящее место для добавления комментариев или расширения материалов в главах.

Об авторах

Роберт (Боб) Роби работает техническим сотрудником отдела вычислительной физики Национальной лаборатории Лос-Аламоса и является адъюнкт-исследователем Университета Нью-Мексико. Он является основателем летней исследовательской стажировки по параллельным вычислениям, которая началась в 2016 году. Он участвует в учебной инициативе NSF/IEEE-TCPP по параллельным и распределенным вычислениям. Боб также является членом правления New Mexico Supercomputing Challenge, образовательной программы для средней и старшей школы, которой уже 30 лет. На протяжении многих лет он был наставником для сотен студентов и дважды был признан выдающимся наставником студентов Лос-Аламоса. Боб был соучредителем курса по параллельным вычислениям в Университете Нью-Мексико и читал гостевые лекции в других университетах.

Боб начал свою научную карьеру с эксплуатации взрывоопасных и сжимаемых газовых ударных труб в Университете Нью-Мексико. Его работа была связана с самой большой в мире ударной трубой со взрывным приводом диаметром 20 футов и длиной более 800 футов. Он провел сотни экспериментов со взрывами и ударными волнами. В целях поддержания своей экспериментальной работы Боб написал несколько программ по гидродинамике сжимаемой жидкости с начала 1990-х годов и является автором многих статей в международных журналах и публикациях. Полное 3D-моделирование в то время было редкостью и требовало предельного напряжения вычислительных ресурсов. Поиск дополнительных вычислительных ресурсов привел его к участию в исследованиях в сфере высокопроизводительных вычислений.

Боб проработал 12 лет в Университете Нью-Мексико, проводя эксперименты, записывая и проводя симулирование гидродинамики сжимаемой жидкости, и основал центр высокопроизводительных вычислений. Он был ведущим автором предложений и принес университету десятки миллионов долларов исследовательских грантов. С 1998 года он занимает должность в Национальной лаборатории Лос-Аламоса. Находясь там, он внес важный вклад в создание больших мультифизических исходных кодов, работающих на самом разном новейшем оборудовании.

Боб является байдарочником мирового класса, впервые спустившимся по ранее неизведанным рекам Мексики и Нью-Мексико. Он также увлекается альпинизмом и имеет за своими плечами восхождения на вершины трех континентов на высоту более 18 000 футов. Он является лидером команды студентов-однокурсников Лос-Аламоса и помогает в многодневных поездках по западным рекам.

Боб окончил Техасский университет A&M со степенью магистра делового администрирования и степенью бакалавра в области машиностроения. Он прошел аспирантуру в Университете Нью-Мексико на математическом факультете.

Джулиана (Джули) Замора заканчивает докторскую степень по информатике в Чикагском университете. Джули является стипендиатом 2017 года в Центре неостанавливаемых вычислений ЦЕРЕРЫ при Чикагском университете и аспирантом Национального консорциума физических наук (NPSC).

Джули работала в Лос-Аламосской национальной лаборатории и стажировалась в Аргоннской национальной лаборатории. В Лос-Аламосской национальной лаборатории она занималась оптимизированием исходного кода Higrad Firetec, используемого для симулирования лесных пожаров и другой физики атмосферы для некоторых из самых лучших систем высокопроизводительных вычислений. В Аргоннской национальной лаборатории она работала на стыке высокопроизводительных вычислений и машинного обучения. Она работала над проектами, начиная от предсказания производительности GPU-процессоров NVIDIA и заканчивая суррогатными моделями машинного обучения для научных приложений.

Джули разработала и преподавала курс «Введение в информатику» для поступающих студентов Чикагского университета. Она включила в свой учебный материал многие главные концепции основ параллельных вычислений. Ее курс был настолько успешным, что ее просили преподавать его снова и снова. Желая получить больше педагогического опыта, она добровольно вызвалась на должность ассистента преподавателя на курсе «Продвинутые распределенные системы» в Чикагском университете.

Степень бакалавра Джули получила в области гражданского строительства в Корнельском университете. Она получила степень магистра компьютерных наук в Чикагском университете и вскоре защитит докторскую диссертацию по информатике, также в Чикагском университете.

Об иллюстрации на обложке

Рисунок на обложке книги «Параллельные и высокопроизводительные вычисления» озаглавлен «M'de de brosses a Vienne», или «Продавец кистей в Вене». Иллюстрация взята из коллекции костюмов из разных стран Жака Грассе де Сен-Совера (1757–1810) под названием Costumes de Différents Pays, опубликованной во Франции в 1797 году. Каждая иллюстрация детально прорисована и раскрашена вручную. Богатое разнообразие коллекции Грассе де Сен-Совера живо напоминает нам о том, насколько культурно обособленными были города и регионы мира всего 200 лет назад. Изолированные друг от друга люди говорили на разных диалектах и языках. На улицах городов или в селах было легко определить местность, где люди живут, и их ремесло или положение в жизни просто по их одежде.

С тех пор наша одежда изменилась, и региональное разнообразие, столь богатое в то время, исчезло. Сейчас трудно отличить жителей разных континентов, не говоря уже о разных городах, регионах или странах. Возможно, мы променяли культурное разнообразие на более разнообразную личную жизнь – конечно же, на более разнообразную и быстро развивающуюся технологическую жизнь.

В то время, когда трудно отличить одну компьютерную книгу от другой, издательство Manning прославляет изобретательность и инициативу компьютерного бизнеса с помощью обложек книг, основанных на богатом разнообразии региональной жизни двухвековой давности, оживленной картинами Грассе де Сен-Совера.

Часть I

Введение в параллельные вычисления

Первая часть этой книги охватывает темы, имеющие общую значимость для параллельных вычислений. Эти темы включают:

- понимание ресурсов параллельного компьютера;
- оценивание производительности и ускорения приложений;
- обзор потребностей, специфичных для параллельных вычислений, со стороны разработчиков программного обеспечения;
- рассмотрение вариантов структур данных;
- отбор алгоритмов, которые показывают хорошую производительность и хорошо параллелизуются.

Хотя эти темы должны рассматриваться параллельным программистом в первую очередь, они не будут иметь одинаковой значимости для всех читателей этой книги. Для разработчика параллельных приложений все главы этой части посвящены первоочередным задачам успешного проекта. Проект нуждается в отборе правильного оборудования, правильного типа параллелизма и правильных ожиданий. Прежде чем приступать к параллелизации, следует определить соответствующие структуры данных и алгоритмы; позже их будет гораздо сложнее изменить.

Даже если вы являетесь разработчиком параллельных приложений, вам, возможно, не понадобится вся глубина обсуждаемого материала. Те, кто желает лишь скромного параллелизма или выполняет ту или иную роль в коллективе разработчиков, могут счесть достаточным беглое понимание содержимого. Если вы просто хотите разведать тему параллель-

ных вычислений, то мы предлагаем прочитать главу 1 и главу 5, а затем просмотреть остальные, чтобы освоиться с терминологией, используемой при обсуждении параллельных вычислений.

Мы включаем главу 2 для тех, у кого, возможно, не будет опыта разработки программного обеспечения, или для тех, кто просто нуждается в освежении своих знаний. Если вы новичок во всех деталях оборудования CPU, то вам, возможно, потребуется читать главу 3 малыми порциями. В целях повышения производительности важно понимать современное компьютерное оборудование и ваше приложение, но это не обязательно должно происходить сразу. Обязательно вернитесь к главе 3, когда будете готовы приобрести свою следующую компьютерную систему, чтобы иметь возможность разобраться во всех маркетинговых заявлениях в отношении того, что действительно имеет важность для вашего приложения.

Обсуждение тем дизайна данных и моделирования производительности в главе 4 может оказаться сложным для восприятия, поскольку для того, чтобы оценить его по достоинству, требуется понимание деталей оборудования, его производительности и компиляторов. Хотя эта тема важна по причине последствий оптимизации кеша и компилятора на производительность, она не обязательна для написания простой параллельной программы.

Мы рекомендуем вам сверяться с прилагаемыми к книге примерами. Рекомендуем вам также потратить немного времени на разведку многочисленных примеров из репозиториев программного обеспечения, расположенных по адресу https://github.com/EssentialsOfParallelComputing.

Примеры организованы по главам и содержат подробную информацию о настройке различного оборудования и операционных систем. Для оказания помощи в решении вопросов переносимости есть примеры контейнерных сборок для дистрибутивов Ubuntu в Docker. Существуют также инструкции по настройке виртуальной машины с помощью VirtualBox. Если у вас есть необходимость в настройке собственной системы, то вы можете прочитать раздел о Docker и виртуальных машинах в главе 13. Но контейнеры и виртуальные машины поставляются с ограниченными средами, которые нелегко обходить.

Наша работа в отношении контейнерных сборок и других настроек системной среды в целях организации надлежащей работы для многих возможных системных конфигураций не прекращается. Правильное инсталлирование системного программного обеспечения, в особенности драйвера GPU и связанного с ним программного обеспечения, является самой сложной частью путешествия. Большое разнообразие операционных систем, оборудования, включая графические процессоры (GPU), и часто упускаемое из виду качество инсталляционного программного обеспечения делают эту работу сложной. Альтернативой является использование кластера, в котором программное обеспечение уже установлено. Тем не менее в какой-то момент полезно инсталлировать некоторое программное обеспечение на свой ноутбук или настольный компьютер с целью обеспечения более удобного ресурса разработки. Теперь самое время перевернуть страницу и войти в мир параллельных вычислений. Это мир почти неограниченной производительности и потенциала.

1 Зачем нужны параллельные вычисления?

Эта глава охватывает следующие ниже темы:

- что такое параллельные вычисления и почему они приобретают все бóльшую значимость;
- где в современном оборудовании существует параллелизм;
- почему важен параллелизм приложений;
- программные подходы для задействования параллелизма.

В современном мире вы столкнетесь со многими задачами, требующими широкого и эффективного использования вычислительных ресурсов. Большинство приложений, требующих производительности, традиционно относится к научной области. Однако, согласно прогнозам, приложения искусственного интеллекта (ИИ) и машинного обучения станут преобладающими пользователями крупномасштабных вычислений. Несколько примеров таких приложений включают:

- моделирование мегапожаров для оказания помощи пожарным командам и населению;
- моделирование цунами и штормовых волн от ураганов (см. главу 13, в которой приводится простая модель цунами);
- распознавание голоса для компьютерных интерфейсов;
- моделирование распространения вируса и разработка вакцин;
- моделирование климатических условий на десятилетия и столетия;

- распознавание изображений для технологии беспилотных автомобилей;
- оснащение аварийных бригад работающими симуляциями таких источников опасности, как наводнение;
- снижение энергопотребления мобильных устройств.

С помощью описанной в этой книге технологии вы сможете справляться с более крупными задачами и наборами данных, одновременно выполняя симуляции в десять, сто или даже в тысячу раз быстрее. Типичные приложения оставляют большую часть вычислительных способностей современных компьютеров незадействованными. Параллельные вычисления представляют собой ключ к выявлению потенциала ваших компьютерных ресурсов. Итак, что такое параллельные вычисления и как их можно использовать, чтобы подстегнуть свои приложения?

Параллельные вычисления – это исполнение большого числа операций в отдельный момент времени. Полное эксплуатирование параллельных вычислений не происходит автоматически. Это требует некоторых усилий от программиста. И прежде всего вы должны определить и выявить потенциал параллелизма в приложении. Потенциальный параллелизм, или *конкурентность*, означает, что вы подтверждаете безопасность выполнения операций в любом порядке по мере появления системных ресурсов. А при параллельных вычислениях существует еще одно, дополнительное требование: эти операции должны выполняться в одно и то же время. Для того чтобы это происходило, вы также должны правильно задействовать ресурсы с целью одновременного исполнения этих операций.

Параллельные вычисления вводят новые трудности, которых нет в последовательном мире. Для того чтобы приспособиться к дополнительным сложностям параллельного исполнения, нам необходимо поменять наши мыслительные процессы, но с практикой это будет становиться второй натурой. Данная книга начинает ваше открытие в том, как получать доступ к мощи параллельных вычислений.

Жизнь представляет массу примеров параллельной обработки, и эти примеры часто становятся основой для вычислительных стратегий. На рис. 1.1 показана кассовая линия супермаркета, где цель состоит в том, чтобы клиенты быстро оплачивали товары, которые они хотят приобрести. Это можно сделать, наняв нескольких кассиров для обработки или проверки клиентов по одному за раз. В этом случае квалифицированные кассиры смогут быстрее исполнять процесс оплаты покупок, в результате чего клиенты будут покидать магазин быстрее. Еще одна стратегия состоит в задействовании большого числа касс самообслуживания и разрешении клиентам исполнять процесс самостоятельно. Эта стратегия требует меньше человеческих ресурсов от супермаркета и позволяет открыть больше линий по обработке клиентов. Клиенты, возможно, не смогут оплачивать покупки так же эффективно, как обученный кассир, но, возможно, больше клиентов сможет быстро выполнять оплату из-за увеличения параллелизма, что приведет к сокращению очередей.

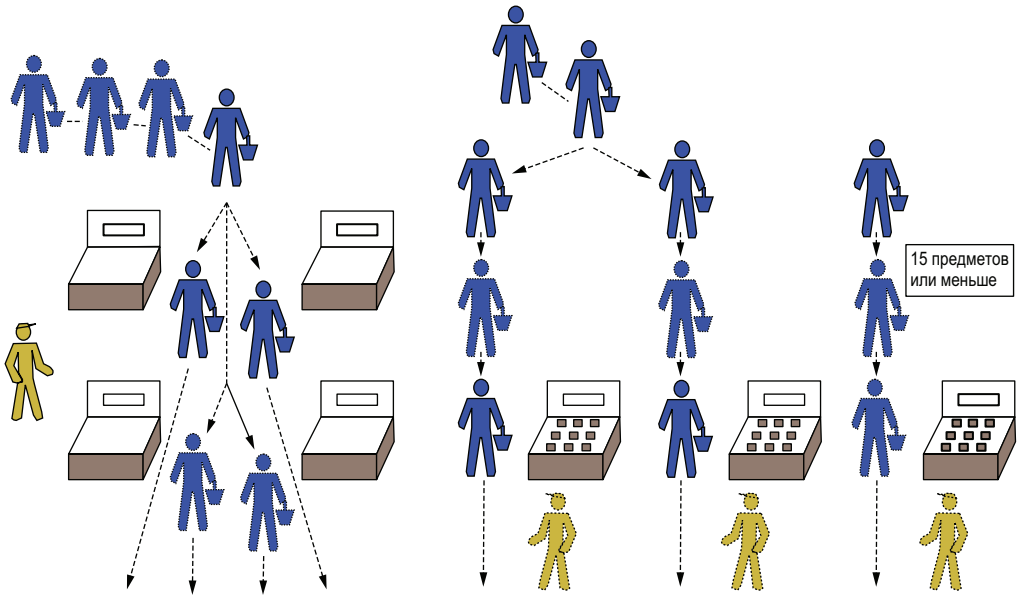


Рис. 1.1 Повседневный параллелизм в очередях в кассы супермаркетов. Кассиры (с кепками) обрабатывают свою очередь клиентов (с корзинами). Слева один кассир одновременно обрабатывает четыре полосы для каждой самостоятельной оплаты покупки. Справа один кассир требуется для каждой полосы оплаты покупки. Каждый вариант влияет на затраты супермаркета и темпы оплаты покупок

Мы решаем вычислительные задачи, разрабатывая *алгоритмы*, перечень шагов для достижения желаемого результата. В аналогии с супермаркетом процесс самообслуживания при оплата покупки представляет собой алгоритм. В данном случае сюда входит выгрузка товаров из корзины, сканирование товаров для получения цены и оплата товаров. Указанный алгоритм является последовательным (или сериальным); он должен следовать этому порядку. Если данная задача должна исполняться сотнями клиентов, то алгоритм оформления покупок многочисленных клиентов содержит параллелизм, которым можно воспользоваться. Теоретически нет никакой зависимости между любыми двумя клиентами, которые проходят процесс оплаты покупки. Используя несколько кассовых линий или станций самообслуживания, супермаркеты демонстрируют параллелизм, тем самым увеличивая скорость, с которой покупатели покупают товары и покидают магазин. Каждый вариант выбора в том, как мы имплементируем этот параллелизм, приводит к разным затратам и выгодам.

ОПРЕДЕЛЕНИЕ *Параллельные вычисления* – это практика выявления и раскрытия параллелизма в алгоритмах с выражением их в нашем программном обеспечении и пониманием затрат, преимуществ и ограничений выбранной имплементации.

В конечном итоге параллельные вычисления всецело касаются *производительности*. Сюда входит не только скорость, но и масштаб задачи и энергоэффективность. Наша цель в этой книге состоит в том, чтобы дать вам понимание широты современной области параллельных вычислений и познакомить с достаточным числом наиболее часто используемых языков, технических приемов и инструментов, чтобы обеспечить возможность уверенно браться за параллельно-вычислительный проект. Важные решения о том, как встраивать параллелизм, часто принимаются в начале проекта. Продуманный дизайн является важным шагом на пути к успеху. Уклонение от шага дизайна может привести к проблемам гораздо позже. Не менее важно сохранять реалистичность ожиданий и знать имеющиеся ресурсы и характер проекта.

Еще одна цель этой главы состоит в ознакомлении вас с терминологией, используемой в параллельных вычислениях. Время от времени это будет делаться, направляя вас к глоссарию в приложении C с целью краткого ознакомления с терминологией, когда вы будете читать эту книгу. Поскольку эта область и технология постепенно расширяются, использование многих терминов участниками параллельного сообщества зачастую бывает небрежным и неточным. В связи с возросшей сложностью оборудования и параллелизма в приложениях важно с самого начала применять терминологию четко и недвусмысленно.

Добро пожаловать в мир параллельных вычислений! По мере того как вы будете углубляться, технические приемы и подходы будут становиться более естественными, и вы обнаружите, что их сила завораживает. Задачи, к решению которых вы никогда не думали приступать, станут обыденным делом.

1.1 Почему вы должны изучить параллельные вычисления?

Будущее будет параллельным. Увеличение последовательной производительности достигло плато, поскольку варианты дизайна процессоров достигли пределов миниатюризации, тактовой частоты, мощности и даже тепла. На рис. 1.2 показаны тренды тактовой частоты (скорости, с которой команда может исполняться), энергопотребления, числа вычислительных ядер (или просто ядер для краткости) и производительности оборудования с течением времени для товарных процессоров.

В 2005 году число ядер резко возросло с одного до нескольких. В то же время тактовая частота и энергопотребление выровнялись. Теоретическая производительность неуклонно повышалась, поскольку производительность пропорциональна произведению тактовой частоты и числа ядер. Этот сдвиг в сторону увеличения числа ядер, а не тактовой частоты указывает на то, что достигуть наиболее идеальную производительность центрального процессора (CPU) можно только за счет параллельных вычислений.

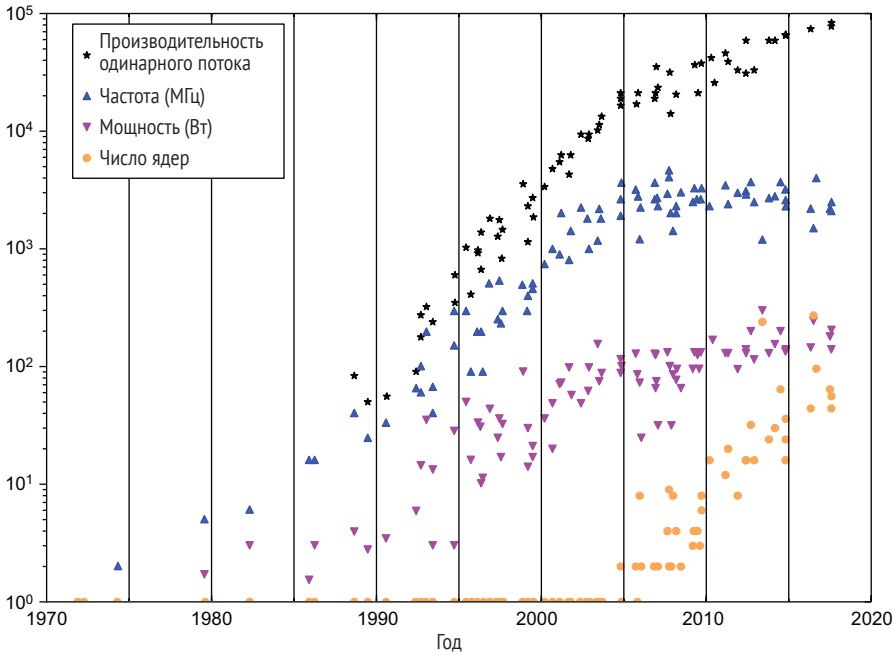


Рис. 1.2 Производительность одинарного потока, тактовая частота процессора (МГц), потребляемая мощность процессора (Вт) и число ядер процессора с 1970 по 2018 год. Эра параллельных вычислений начинается примерно в 2005 году, когда число ядер в процессорных микросхемах начинает расти, в то время как тактовая частота и энергопотребление снижаются, но производительность неуклонно растет (Горовиц и соавт. и Рупп, https://github.com/karlrupp/microprocessor-trend-data**)**

Современное вычислительное оборудование потребительского класса оснащается несколькими центральными процессорами (CPU) и/или графическими процессорами (GPU), которые обрабатывают несколько наборов команд одновременно. Эти небольшие системы часто соперничают по вычислительной мощности с суперкомпьютерами двадцатилетней давности. В целях полного использования вычислительных ресурсов (на ноутбуках, рабочих станциях, смартфонах и т. д.) требуется, чтобы вы, программист, обладали практическими знаниями об инструментах, служащих для написания параллельных приложений. Вы также должны понимать аппаратные функциональности, которые подстегивают параллелизм.

Поскольку существует много разных параллельных аппаратных функциональностей, это создает программисту новые сложности. Одной из таких функциональностей является гиперпотокообразование (hyper-threading)[1], представленное компанией Intel. Наличие двух очередей

[1] Гиперпотокообразование (Hyper-Threading) – это термин, пришедший из компании Intel, для обозначения одновременного множественного потокообразования (simultaneous multithreading, SMT). Это процесс, в котором центральный процессор (CPU) разбивает каждое свое физическое ядро на вир-

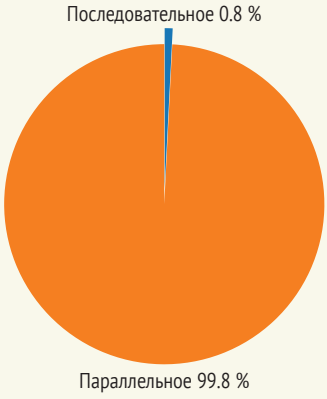
команд, чередующих работу с аппаратными логическими модулями, позволяет одному физическому ядру выглядеть в операционной системе (OS) как два ядра. Векторные процессоры являются еще одной аппаратной функциональностью, которая начала появляться в товарных процессорах примерно в 2000 году. Они исполняют несколько инструкций сразу. Ширина в битах векторного процессора (также именуемого модулем векторной обработки) определяет число команд, которые должны исполняться одновременно. Отсюда модуль векторной обработки шириной 256 бит может исполнять одновременно четыре 64-битовые команды (двойной точности) или восемь 32-битовых команд (одинарной точности).

Пример

Давайте возьмем 16-ядерный CPU с гиперпотокообразованием и векторным модулем шириной 256 бит, который обычно используется в домашних настольных компьютерах. Последовательная программа, использующая одинарное ядро и не имеющая векторизации, использует только 0.8 % теоретических обрабатывающих способностей этого процессора! Расчет выглядит так:

16 ядер × 2 гиперпотока × (векторный модуль шириной 256 бит) / (64-бит двойной точности) = 128-путный параллелизм,

где 1 последовательный путь / 128 параллельных путей = 0.008, или 0.8 %. На следующем ниже рисунке показано, что это составляет мизерную долю суммарной вычислительной мощности процессора.



Последовательное приложение обращается только к 0.8 % вычислительной мощности 16-ядерного процессора

Умение проводить расчет теоретических и реалистичных ожиданий последовательной и параллельной производительности, как показано в этом примере, имеет большую важность. Мы обсудим это подробнее в главе 3.

туальные ядра, именуемые потоками (threads). Например, в большинстве процессоров Intel с двумя ядрами гиперпотокообразование используется для обеспечения четырех потоков. – *Прим. перев.*

Несколько усовершенствований в инструментах разработки программного обеспечения помогло добавить параллелизм в наши инструментарии, и в настоящее время научно-исследовательское сообщество делает больше, но до устранения разрыва в производительности еще далеко. Это ложится большой нагрузкой на нас, разработчиков программного обеспечения, чтобы получать максимальную отдачу от процессоров нового поколения.

К сожалению, разработчики программного обеспечения отстали в адаптации к этому фундаментальному изменению вычислительной мощности. Кроме того, транзит существующих приложений в сторону использования современных параллельных архитектур может оказаться обескураживающе сложным из-за бурного развития новых языков программирования и интерфейсов прикладного программирования (API). Но хорошая работающая осведомленность о вашем приложении, способность видеть и выявлять параллелизм, а также глубокое понимание имеющихся инструментов обязательно принесут существенные преимущества. Какие именно преимущества увидят приложения? Давайте посмотрим поближе.

1.1.1 Каковы потенциальные преимущества параллельных вычислений?

Параллельные вычисления могут сокращать время до решения[1], повышать энергоэффективность вашего приложения и позволять решать более масштабные задачи на существующем в настоящее время оборудовании. Сегодня параллельные вычисления больше не являются единственной областью деятельности крупнейших вычислительных систем. Указанная технология теперь присутствует на всех настольных компьютерах или ноутбуках и даже на портативных устройствах. Она позволяет каждому разработчику программного обеспечения создавать параллельное программное обеспечение в своих локальных системах, тем самым значительно расширяя возможности для новых приложений.

Передовые исследования как в промышленности, так и в научных кругах открывают новые области для параллельных вычислений по мере того, как интерес расширяется от научных вычислений до машинного обучения, больших данных, компьютерной графики и потребительских приложений. Появление новых технологий, таких как самоуправляемые автомобили, компьютерное зрение, распознавание голоса и искусственный интеллект, требует крупных вычислительных способностей как внутри потребительских устройств, так и в сфере разработки, где необходимо использовать и обрабатывать массивные тренировочные наборы данных. И в научных вычислениях, которые долгое время были исключительной областью параллельных вычислений, тоже появляются новые, захваты-

[1] Время до решения (time to solution) – это продолжительность времении между предобработкой и постобработкой данных при решении поставленной задачи. – *Прим. перев.*

вающие возможности. Распространение удаленных датчиков и портативных устройств, которые могут подавать данные в более масштабные и реалистичные вычисления для более оптимального информирования при принятии решений, имеющих отношение к природным и техногенным катастрофам, позволяет получать более обширные данные.

Следует помнить, что параллельные вычисления как таковые не являются самоцелью. Правильне сказать, цели – это то, что является результатом параллельных вычислений: сокращение времени выполнения, выполнение более крупных вычислений или снижение энергопотребления.

Более быстрое время выполнения с бóльшим числом вычислительных ядер

Сокращение времени выполнения приложения, или его *ускорение*, часто считается первостепенной целью параллельных вычислений. И действительно обычно это имеет наибольшие последствия. Параллельные вычисления способны ускорять интенсивные вычисления, обработку мультимедиа и операции с большими данными независимо от того, требуются ли для обработки ваших приложений дни или даже недели, или же результаты необходимы в реальном времени.

В прошлом программист тратил больше усилий на последовательную оптимизацию, чтобы выжать несколько процентных улучшений. Теперь есть потенциал для улучшения на порядки, с многочисленными вариантами на выбор. Это создает новую проблему в разведывании возможных параллельных парадигм – больше возможностей, чем численность программистов. Но глубокое знание вашего приложения и понимание возможностей параллелизма непременно поведут вас по более ясному пути к сокращению времени выполнения вашего приложения.

Более крупные размеры задач с бóльшим числом вычислительных узлов

Выявляя параллелизм в своем приложении, вы можете масштабировать размер вашей задачи вертикально до размерностей, недоступных для последовательного приложения. Это связано с тем, что объем вычислительных ресурсов обуславливает то, что можно сделать, а параллелизм позволяет работать с большими ресурсами, предоставляя возможности, которые раньше никогда не рассматривались. Более крупные размеры обеспечиваются за счет большего объема основной памяти, дискового хранилища, пропускной способности сетей и диска, а также центральных процессоров (CPU). По аналогии с супермаркетом, как упоминалось ранее, выявление параллелизма эквивалентно найму большего числа кассиров либо открытию большего числа касс самостоятельной оплаты покупок с целью обслуживания большего и растущего числа клиентов.

Энергоэффективность: делая больше с меньшими затратами

Одной из новых областей влияния параллельных вычислений является энергоэффективность. С появлением параллельных ресурсов в наладонных устройствах параллелизм может ускорять работу приложений. Это позволяет устройству быстрее возвращаться в спящий режим и дает

возможность использовать более медленные, но более параллельные процессоры, потребляющие меньше энергии. Отсюда перенос работы тяжеловесных мультимедийных приложений на GPU-процессоры может оказать более существенное влияние на энергоэффективность, а также значительно повысить производительность. Чистый результат использования параллелизма снижает энергопотребление и продлевает срок службы батареи, что является сильным конкурентным преимуществом в этой рыночной нише.

Еще одна область, в которой важность энергоэффективности неоспорима, – это удаленные датчики, сетевые устройства и операционные устройства, развернутые на местах, таких как удаленные метеостанции. Нередко без больших источников питания эти устройства должны быть способны функционировать малыми пакетами с небольшим объемом ресурсов. Параллелизм расширяет возможности, которые могут имплементироваться на этих устройствах, и разгружает работу центральной вычислительной системы в растущем тренде, который называется *периферийным вычислением* (edge compute). Перемещение вычислений на самый край сети обеспечивает возможность вести обработку в источнике данных, сжимая их в меньший результирующий набор, который легче отправлять по сети.

Точный расчет энергетических затрат приложения является сложной задачей без прямых измерений энергопотребления. Однако вы можете оценить затраты путем умножения расчетной тепловой мощности производителя на время выполнения приложения и число используемых процессоров. Расчетная тепловая мощность (thermal design power, TDP) – это скорость, с которой энергия расходуется при типичных эксплуатационных нагрузках. Потребление энергии для вашего приложения можно оценить по формуле:

$$P = (N \text{ Процессоров}) \times (R \text{ Ватт/Процессоры}) \times (T \text{ часов}),$$

где P – это потребление энергии, N – число процессоров, R – расчетная тепловая мощность и T – время выполнения приложения.

Пример

16-ядерный процессор Intel Xeon E5-4660 имеет расчетную тепловую мощность 120 Вт. Предположим, что ваше приложение использует 20 из этих процессоров в течение 24 часов до полного завершения работы. Оценочное потребление энергии для вашего приложения составляет:

$$P = (20 \text{ Процессоров}) \times (120 \text{ Вт/Процессоры}) \times (24 \text{ часа}) = 57.60 \text{ кВт/ч}.$$

В общем случае GPU имеют более высокую расчетную тепловую мощность, чем современные CPU, но потенциально способны сократить время работы или потребовать всего нескольких GPU для получения того же результата. Можно применить ту же формулу, что и раньше, где N теперь рассматривается как число GPU.

Пример

Предположим, что вы портировали свое приложение на мульти-GPU-платформу. Теперь вы можете выполнять свое приложение на четырех GPU NVIDIA Tesla V100 за 24 часа! GPU Tesla V100 от NVIDIA имеет максимальную тепловую мощность 300 Вт. Оценочное потребление энергии для вашего приложения составляет:

P = (4 GPU-процессора) × (300 Вт/GPU-процессоры) × (24 часа) = 28.80 кВт/ч.

В этом примере приложение с ускорением на основе GPU работает с вдвое меньшими энергозатратами, чем версия только для CPU. Обратите внимание, что в этом случае, даже несмотря на то, что время до решения остается прежним, затраты энергии сокращаются вдвое!

Достижение снижения энергозатрат с помощью ускорителей, таких как GPU-процессоры, требует наличия в приложении достаточного параллелизма, который можно выявить. Это позволяет эффективно пользоваться ресурсами устройства.

Параллельные вычисления могут снизить затраты

Фактические денежные затраты становятся все более заметной озабоченностью для коллективов разработчиков программного обеспечения, пользователей программного обеспечения и исследователей в равной степени. По мере роста размеров приложений и систем нам необходимо проводить анализ затрат и выгод на имеющихся у нас ресурсах. Например, в следующих крупных системах высокопроизводительных вычислений (HPC) затраты на электроэнергию, по прогнозам, в три раза превысят стоимость приобретения оборудования.

Затраты на использование также способствовали развитию облачных вычислений в качестве альтернативы, которая все чаще внедряется в научных кругах, стартапах и отраслях промышленности. Как правило, поставщики облачных служб выставляют счета в зависимости от типа и объема используемых ресурсов, а также количества времени, потраченного на их использование. Хотя GPU-процессоры, как правило, стоят дороже, чем CPU в расчете на единицу времени, в некоторых приложениях могут использоваться GPU-ускорители, чтобы обеспечивать достаточное сокращение времени выполнения по сравнению с расходами на CPU с целью снижения затрат.

1.1.2 Предостережения, связанные с параллельными вычислениями

Параллельные вычисления не являются панацеей. Многие приложения недостаточно велики или не требуют достаточного времени выполнения, чтобы нуждаться в параллельных вычислениях. У некоторых может даже не быть достаточного внутреннего параллелизма, который можно

было бы эксплуатировать. Кроме того, транзит приложений на использование мультиядерного и многоядерного оборудования (GPU) требует выделенных усилий, которые могут временно отвлекать внимание от прямых исследований или продуктовых целей. Сначала следует осознать целесообразность затрат времени и усилий. Всегда важнее обеспечить функционирование приложения и генерирование им желаемого результата, прежде чем его ускорять и масштабировать его вертикально до более крупных задач.

Мы настоятельно рекомендуем вам начать свой параллельно-вычислительный проект с плана. Важно знать варианты, которые доступны для ускорения приложения, а затем выбрать наиболее подходящий для вашего проекта. После этого крайне важно иметь разумную оценку затраченных усилий и потенциальных выгод (с точки зрения стоимости в долларах, энергопотребления, времени до решения и других показателей, которые могут представлять важность). В этой главе мы начнем давать вам авансом знания и навыки для принятия решений по параллельно-вычислительным проектам.

1.2 Фундаментальные законы параллельных вычислений

В последовательных вычислениях все операции ускоряются по мере увеличения тактовой частоты. В отличие от них в случае параллельных вычислений нам необходимо немного поразмыслить и модифицировать наши приложения так, чтобы задействовать параллельное оборудование в полной мере. Почему так важен объем параллелизма? Для того чтобы в этом разобраться, давайте взглянем на законы параллельных вычислений.

1.2.1 Предел на параллельные вычисления: закон Амдала

Нам необходим подход к расчету потенциального ускорения вычисления, основываясь на объеме параллельного кода. Это можно сделать, используя Закон Амдала, предложенный Джином Амдалом в 1967 году. Указанный закон описывает ускорение задачи фиксированного размера по мере увеличения числа процессоров. Следующее ниже уравнение это показывает, где P – это параллельная доля кода, S – последовательная доля, означающая, что $P + S = 1$, и N – это число процессоров:

$$\text{Ускорение}(N) = \frac{1}{S + \frac{P}{N}}.$$

Закон Амдала подчеркивает, что независимо от того, какой быстрой мы делаем параллельную часть кода, мы всегда будем лимитированы

последовательной частью. На рис. 1.3 показан этот предел. Такое масштабирование задачи фиксированного размера называется сильным масштабированием.

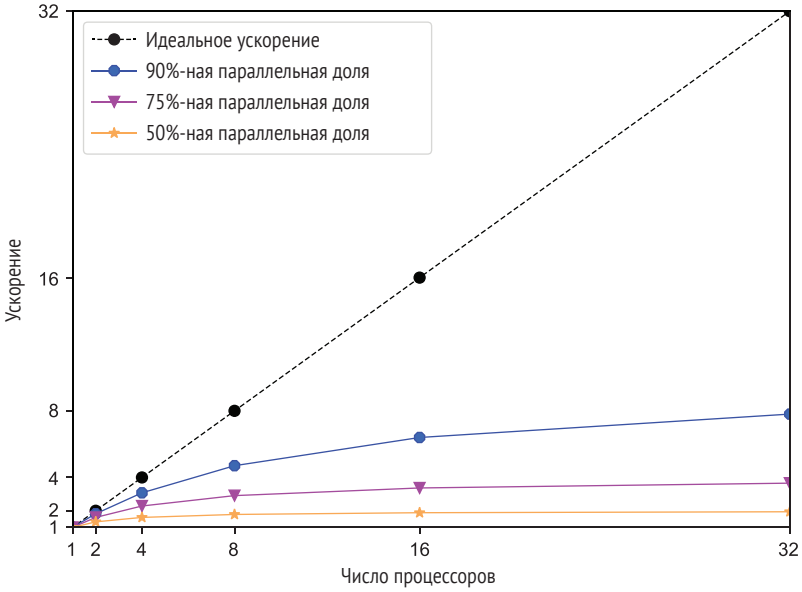


Рис. 1.3 Ускорение для задачи фиксированного размера в соответствии с законом Амдала показано как функция от числа процессоров. Линии показывают идеальное ускорение, когда параллелизуется 100 % алгоритма, а также для 90, 75 и 50 %. Закон Амдала гласит, что ускорение лимитировано долями кода, которые остаются последовательными

ОПРЕДЕЛЕНИЕ *Сильное масштабирование* представляет время до решения в зависимости от числа процессоров для задачи фиксированного суммарного размера.

1.2.2 Преодоление параллельного предела: закон Густафсона–Барсиса

В 1988 году Густафсон и Барсис указали на то, что прогоны параллельного кода должны увеличивать размер задачи по мере добавления большего числа процессоров. Это дает нам альтернативный подход к расчету потенциального ускорения нашего приложения. Если размер задачи растет пропорционально числу процессоров, то ускорение теперь выражается как

$$\text{Ускорение}(N) = N - S \times (N - 1),$$

где N – это число процессоров, а S – последовательная доля, как и раньше. Как следствие, более крупная задача может быть решена за одно и то

же время с использованием большего числа процессоров. Это предоставляет дополнительные возможности для привлечения параллелизма. И действительно увеличение размера задачи вместе с числом процессоров имеет смысл, потому что пользователь приложения хочет извлекать выгоду не только из мощности дополнительного процессора, но и хочет использовать дополнительную память. Масштабирование времени выполнения для этого сценария, показанное на рис. 1.4, называется слабым масштабированием.

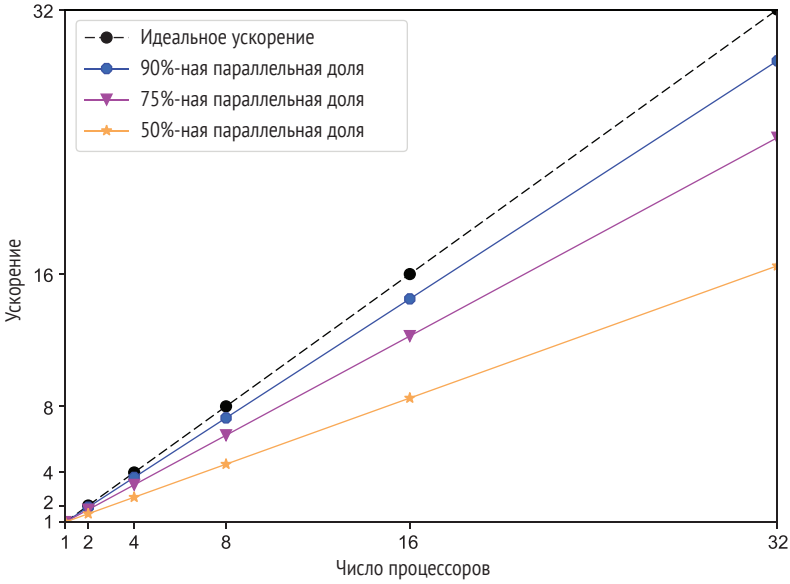


Рис. 1.4 Ускорение для случаев, когда размер задачи увеличивается вместе с увеличением числа доступных процессоров в соответствии с законом Густафсона–Барсиса, показано как функция числа процессоров. Линии показывают идеальное ускорение, когда параллелизуется 100 % алгоритма, а также для 90, 75 и 50 %

ОПРЕДЕЛЕНИЕ *Слабое масштабирование* представляет время до решения в зависимости от числа процессоров для задачи фиксированного размера в расчете на процессор.

На рис. 1.5 наглядно показана разница между сильным и слабым масштабированием. Аргументация слабого масштабирования в отношении того, что размер вычислительной сетки должен оставаться постоянным на каждом процессоре, позволяет эффективно пользоваться ресурсами дополнительного процессора. С позиций сильного масштабирования все внимание сконцентрировано в первую очередь на ускорении вычисления. На практике важны как сильное, так и слабое масштабирование, поскольку они решают разные пользовательские сценарии.

Термин «масштабируемость» часто используется для обозначения возможности добавлять больше параллелизма в аппаратное или про-

граммное обеспечение и существования совокупного предела возможного улучшения. В то время как традиционный фокус внимания лежит на масштабировании времени выполнения, мы выдвигаем аргумент, что масштабирование памяти часто имеет бóльшую значимость.

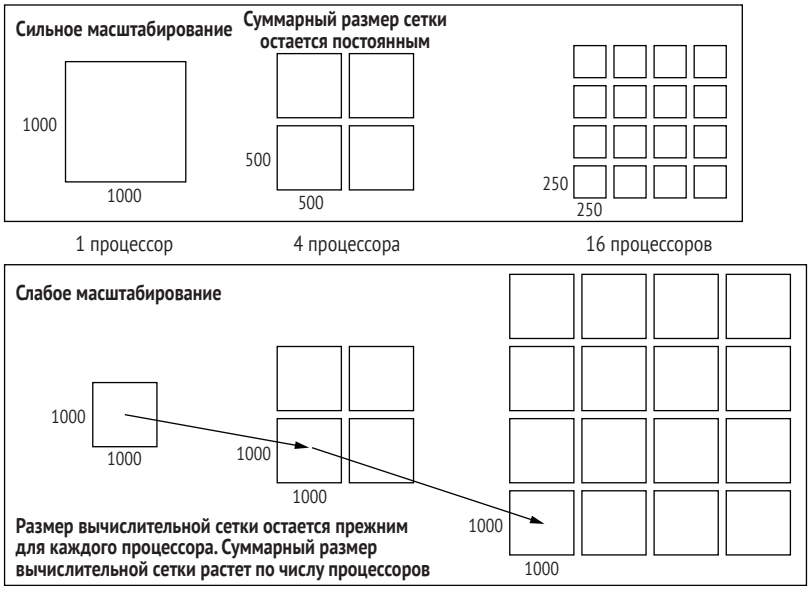


Рис. 1.5 Сильное масштабирование сохраняет тот же совокупный размер задачи и распределяет ее по дополнительным процессорам. При слабом масштабировании размер вычислительной сетки остается одинаковым для каждого процессора, а суммарный размер увеличивается

На рис. 1.6 показано приложение с лимитированной масштабируемостью памяти. *Реплицированный массив* (R) – это набор данных, который дублируется во всех процессорах. *Распределенный массив* (D) разбивается на разделы и распределяется между процессорами.

Например, в игровом симуляторе 100 персонажей могут быть распределены между 4 процессорами по 25 персонажей на каждом процессоре. Но карта игрового поля может быть скопирована на каждый процессор. На рис. 1.6 реплицированный массив дублируется по всей вычислительной сетке. Поскольку этот рисунок относится к слабому масштабированию, размер задачи увеличивается вместе с увеличением числа процессоров. Для 4 процессоров массив в 4 раза больше на каждом процессоре.

Поскольку размер задачи растет, вскоре на процессоре не хватит памяти для выполнения задания. Лимитированное масштабирование времени выполнения означает, что задание выполняется медленно; лимитированное масштабирование памяти означает, что задание не может выполняться вообще. Это также тот случай, что если память приложения может быть распределена, то время выполнения обычно масштабируется тоже. Однако обратное не обязательно верно.

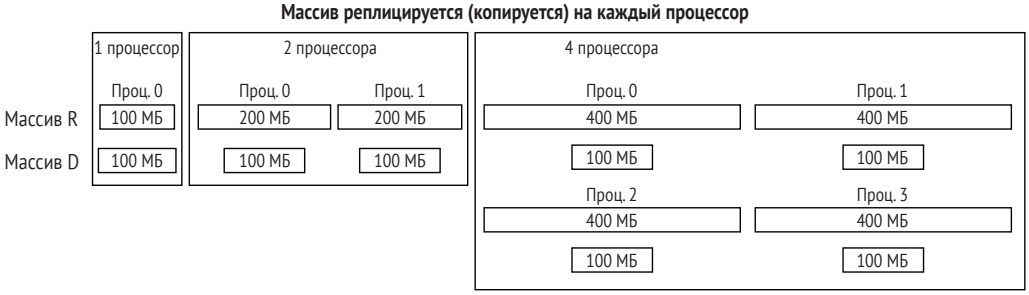


Рис. 1.6 Распределенные массивы остаются того же размера, что и задача, а число процессоров удваивается (слабое масштабирование). Но реплицированные (скопированные) массивы нуждаются во всех данных на каждом процессоре, при этом память быстро растет вместе с увеличением числа процессоров. Даже если время выполнения масштабируется слабо (остается постоянным), требования к памяти лимитируют масштабируемость

Одна из точек зрения на интенсивное вычислительное задание состоит в том, что каждый байт памяти затрагивается в каждом цикле обработки, а время выполнения является функцией от размера памяти. Сокращение размера памяти обязательно приведет к сокращению времени выполнения. Таким образом, первоначальное внимание в параллелизме должно быть направлено на сокращение размера памяти по мере увеличения числа процессоров.

1.3 Как работают параллельные вычисления?

Параллельные вычисления требуют сочетания понимания оборудования, программного обеспечения и параллелизма при разработке приложения. Это больше, чем просто передача сообщений или потокообразование. Современное оборудование и программное обеспечение предоставляют массу разных возможностей для параллелизации вашего приложения. Некоторые из этих возможностей могут быть скомбинированы, чтобы обеспечить еще большую эффективность и ускорение.

Важно иметь понимание параллелизации в вашем приложении и того, как разные аппаратные компоненты позволяют вам ее выявлять. Кроме того, разработчики должны понимать, что между вашим исходным кодом и оборудованием ваше приложение должно проходить дополнительные слои, включая компилятор и операционную систему (рис. 1.7).

Как разработчик, вы несете ответственность за слой прикладного программного обеспечения, в котором содержится ваш исходный код. В исходном коде вы выбираете язык программирования и параллельные программные интерфейсы, которые используете для задействования опорного оборудования. В дополнение вы принимаете решение о том, как разбить свою работу на параллельные модули. Компилятор служит для транслирования вашего исходного кода в форму, исполнимую вашим оборудованием. Получив в свое распоряжение эти команды,

операционная система управляет их исполнением на компьютерном оборудовании.

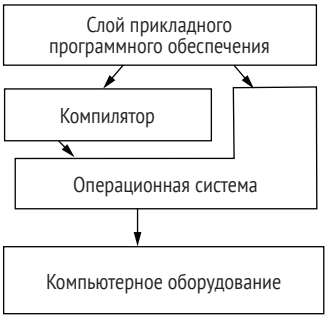


Рис. 1.7 Параллелизация выражается в слое прикладного программного обеспечения, который соотносится с компьютерным оборудованием через компилятор и OS

Мы покажем вам на примере процесс введения параллелизации в алгоритм с помощью прототипного приложения. Указанный процесс происходит в слое прикладного программного обеспечения, но требует понимания компьютерного оборудования. Пока что мы воздержимся от обсуждения выбора компилятора и операционной системы. Мы будем поступательно добавлять каждый слой параллелизации, чтобы у вас имелась возможность видеть принцип ее работы. В каждой параллельной стратегии мы объясним характер влияния доступного оборудования на сделанный выбор. Цель этого состоит в том, чтобы продемонстрировать то, как аппаратные функциональности влияют на параллельные стратегии. Мы классифицируем параллельные подходы, которые разработчик может принять, на:

- параллелизацию на основе процессов (process-based parallelization);
- параллелизацию на основе потоков (thread-based parallelization);
- векторизацию (vectorization);
- обработку в потоковом режиме (stream processing).

Следуя этому примеру, мы введем модель, которая поможет вам рассуждать о современном оборудовании. Эта модель разбивает современное вычислительное оборудование на отдельные компоненты и различные вычислительные устройства. В эту главу включен упрощенный взгляд на память. Более подробный обзор иерархии памяти представлен в главах 3 и 4. Наконец, мы подробнее обсудим слои приложения и программное обеспечение.

Как уже упоминалось, мы классифицируем параллельные подходы, которые разработчик может принять, на параллелизацию на основе процессов, параллелизацию на основе потоков (т. е. виртуальных ядер), векторизацию и обработку в потоковом режиме. Параллелизация на основе отдельных процессов с их собственными пространствами памяти может представлять собой распределенную память на разных узлах компьютера или внутри узла. Обработка в потоковом режиме обычно связана с GPU-процессорами. Модель современного оборудования и прикладного программного обеспечения поможет вам лучше понять принцип

планирования переноса вашего приложения на текущее параллельное оборудование.

1.3.1 Пошаговое ознакомление с примером приложения

В целях этого введения в параллелизацию мы рассмотрим подход на основе параллелизма данных. Это одна из наиболее распространенных стратегий применения параллельных вычислений. Мы выполним вычисления на пространственной вычислительной сетке, состоящей из регулярной двумерной (2D) решетки прямоугольных элементов или ячеек. Шаги (резюмированные здесь и подробно описанные позже) для создания пространственной расчетной сетки и подготовки к вычислению таковы:

1. дискретизировать (подразделить) задачу на более мелкие ячейки или элементы;
2. определить вычислительное ядро (операцию) для выполнения на каждом элементе вычислительной сетки;
3. добавить следующие слои параллелизации на CPU-процессорах и GPU-процессорах для выполнения расчета:
   - *векторизацию* – работать на более одной единице данных за один раз;
   - *потоки* – развертывать более одного вычислительного маршрута для привлечения большего числа процессорных ядер;
   - *процессы* – отделять программные экземпляры для распространения расчета по отдельным пространствам памяти;
   - *выгрузку расчета на GPU-процессоры* – отправлять данные в GPU для расчета.

Мы начинаем с двумерной задачной области участка пространства. Для целей иллюстрации мы будем использовать 2D-изображение вулкана Кракатау (рис. 1.8) в качестве примера. Целью наших расчетов может быть моделирование шлейфа вулканического выброса, возникающего в результате цунами или раннее обнаружение извержения вулкана с использованием машинного обучения. Для всех этих вариантов скорость вычислений имеет решающее значение, если мы хотим, чтобы реальновременные результаты информировали наши решения.

Рис. 1.8 Пример двухмерной пространственной области для численной симуляции. Численные симуляции обычно предусматривают стенсильные операции (см. рис. 1.11) или большие матрично-векторные системы. Эти типы операций часто используются при моделировании жидкостей для продуцирования предсказаний времени прибытия цунами, прогнозов погоды, распространения дымового шлейфа и других процессов, необходимых для принятия обоснованных решений

Шаг 1. Дискретизировать задачу на более мелкие ячейки или элементы

Для любого подробного расчета мы должны сначала разбить область задачи на более мелкие части (рис. 1.9). Указанный процесс называется *дискретизацией*. При обработке изображений это часто просто пикселы в растровом изображении. Для вычислительной области они называются ячейками или элементами. Коллекция ячеек или элементов образует *вычислительную сетку*, которая охватывает пространственный участок для симуляции. Значения данных для каждой ячейки могут быть целыми числами, вещественными числами или числами двойной точности.

Рис. 1.9 Область дискретизирована на ячейки. Для каждой ячейки в вычислительной области такие свойства, как высота волны, скорость жидкости или плотность дыма, решаются в соответствии с физическими законами. В конечном счете стенсильная операция или матрично-векторная система представляет эту дискретную схему

Шаг 2. Определить вычислительное ядро или операцию, необходимую для выполнения на каждом элементе сетки

Расчеты на этих дискретизированных данных часто являются некоторой формой стенсильной операции, так именуемой, потому что она предусматривает шаблон смежных ячеек для вычисления нового значения для каждой ячейки. Это может быть среднее значение (операция размытия, которая размывает изображение или делает его более расплывчатым), градиент (обнаружение краев, которое делает края изображения более четкими) или другая более сложная операция, связанная с решением физических систем, описываемых уравнениями в частных производных (PDE). На рис. 1.10 показана стенсильная операция в виде пятиточечного трафарета, который выполняет операцию размытия, используя средневзвешенное значение стенсильных значений.

Но что это за уравнения в частных производных? Давайте вернемся к нашему примеру и представим, что на этот раз это цветное изображение, состоящее из отдельных красных, зеленых и синих массивов, составляющих цветовую модель RGB. Термин «частный» здесь означает, что существует более одной переменной и что мы отделяем изменение красного

цвета в пространстве и времени от изменения зеленого и синего. Затем мы выполняем оператор размытия отдельно для каждого из этих цветов.

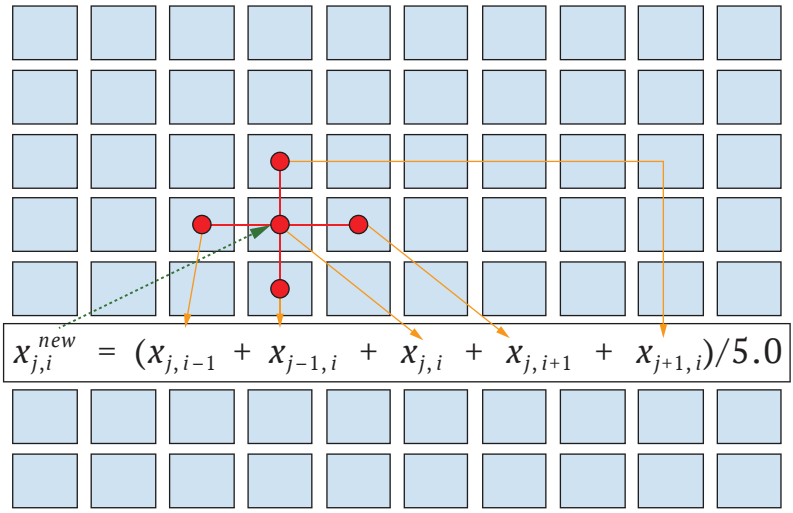


Рис. 1.10 Пятиточечный стенсильный оператор в виде поперечного шаблона на вычислительной сетке. Отмеченные трафаретом данные считываются во время операции и сохраняются в центральной ячейке. Этот шаблон повторяется для каждой ячейки. Оператор размытия, один из более простых стенсильных операторов, представляет собой взвешенную сумму пяти точек, отмеченных большими точками, и обновляет значение в центральной точке трафарета. Этот тип операций выполняется для операций сглаживания или численных симуляций распространения волн

Есть еще одно требование: нам нужно применять скорость изменения во времени и пространстве. Другими словами, красный цвет будет распространяться с одной скоростью, а зеленый и синий – с другой. Это может делаться с целью продуцирования на изображении специального эффекта или может описывать то, как реальные цвета просачиваются и сливаются в фотографическом изображении во время проявления. В научном мире вместо красного, зеленого и синего мы могли бы иметь массу и скорость *x* и *y*. Если добавить немного больше физики, то мы могли бы получить движение волны или пепельного шлейфа.

Шаг 3. Векторизация для работы с более чем одной единицей данных за один раз

Мы начинаем вводить параллелизацию с рассмотрения векторизации. Что такое векторизация? Некоторые процессоры имеют возможность оперировать на более чем одной порции данных за один раз; эта возможность называется *векторными операциями*. Затененные блоки на рис. 1.11 иллюстрируют то, как несколько значений данных обрабатываются одновременно в модуле векторной обработки в процессоре одной командой за один тактовый цикл.

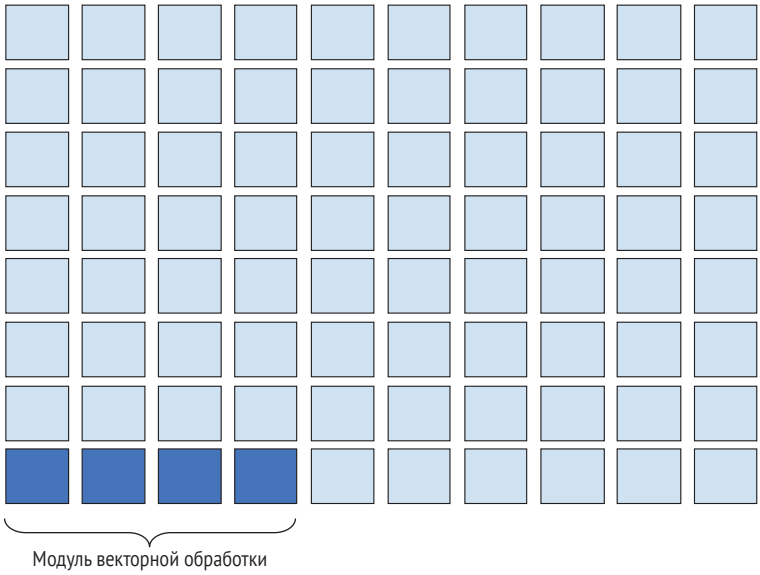


Рис. 1.11 Специальная векторная операция выполняется на четырех значениях двойной точности. Эта операция может исполняться за один тактовый цикл с небольшими дополнительными энергозатратами на последовательную операцию

Шаг 4. Потоки для развертывания более одного вычислительного маршрута с целью привлечения большего числа обрабатывающих ядер

Поскольку большинство CPU сегодня имеет по крайней мере четыре обрабатывающих ядра, мы используем потокообразование для оперирования ядрами в одновременном режиме в четырех строках за один раз. Этот процесс показан на рис. 1.12.

Шаг 5. Процессы для распространения вычислений по отдельным пространствам памяти

Мы можем еще больше разбить работу между процессорами на два настольных компьютера, в параллельной обработке часто именуемых *узлами*. Когда работа разбита на узлы, пространства памяти под каждый узел отличимы и отделены. На это указывает наличие промежутка между строками, как показано на рис. 1.13.

Даже для этого довольно скромного аппаратного сценария существует потенциальное ускорение в 32 раза. Об этом свидетельствуют следующие ниже данные:

2 настольных компьютера (узла) × 4 ядра × (модуль векторной обработки шириной 256 бит)/(64-битное значение двойной точности) = 32-кратное потенциальное ускорение.

Если взять высококлассный кластер с 16 узлами, 36 ядрами на узел и 512-битовым векторным процессором, то потенциальное теоретиче-

ское ускорение будет 4608-кратным по сравнению с последовательным процессом:

16 узлов × 36 ядер × (модуль векторной обработки шириной 512 бит)/ (64-битное значение двойной точности) = 4.608-кратное потенциальное ускорение.

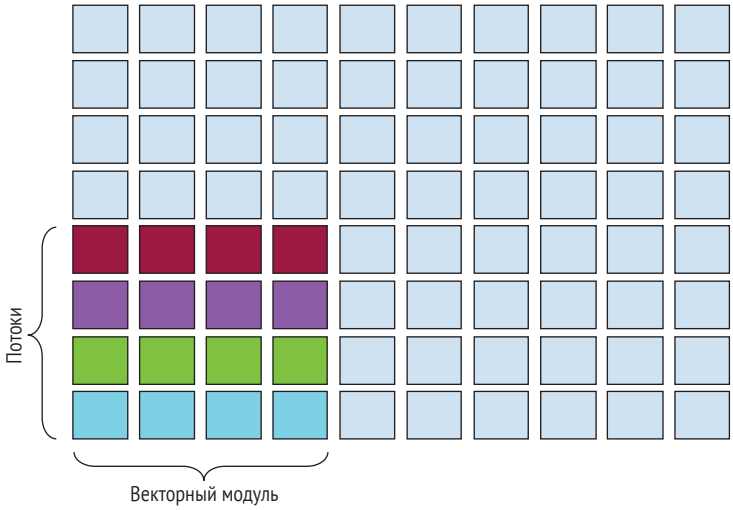


Рис. 1.12 Четыре потока обрабатывают четыре строки векторных модулей одновременно

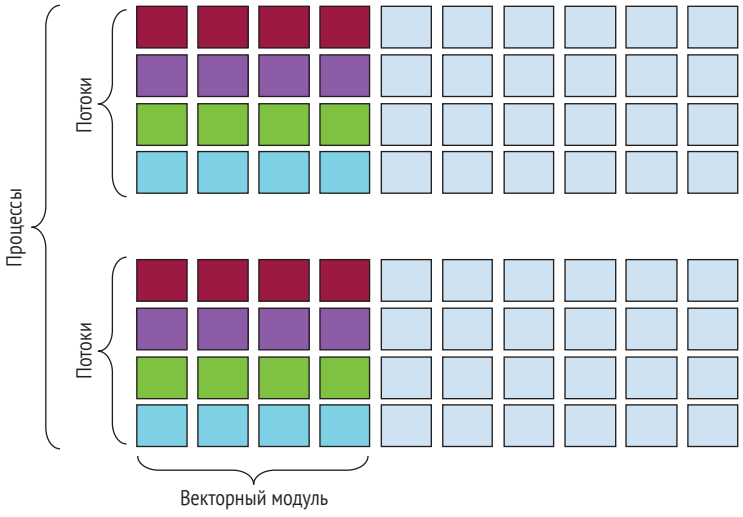


Рис. 1.13 Этот алгоритм можно параллелизовать еще больше, распределив блоки 4×4 между отличимыми процессами. В каждом процессе используется четыре потока, каждый из которых обрабатывает векторный модуль шириной четыре узла за один тактовый цикл. Дополнительное пустое пространство на рисунке иллюстрирует границы процесса

Шаг 6. Выгрузка расчета на GPU-процессоры

GPU – это еще один аппаратный ресурс для подстегивания параллелизации. С помощью GPU-процессоров мы можем привлекать к работе большое число *потоковых мультипроцессоров* (streaming multiprocessors, SMs). Например, на рис. 1.14 показан принцип разделения работы отдельно на плитки размером 8×8. Используя технические характеристики оборудования для GPU NVIDIA Volta, на этих плитках могут оперировать 32 ядра двойной точности, распределенные по 84 потоковым мультипроцессорам, что дает нам в общей сложности 2688 ядер двойной точности, работающих одновременно. Если у нас есть один GPU на узел в 16-узловом кластере, каждый с 2688 потоковыми мультипроцессорами двойной точности, то это составит 43 008-путную параллелизацию из 16 GPU-процессоров.

Рис. 1.14 На GPU длина вектора намного больше, чем на CPU. Здесь плитки размером 8×8 распределены по рабочим группам GPU

Эти цифры впечатляют, но пока что мы должны умерить ожидания, признав, что фактическое ускорение значительно отстает от этого полного потенциала. Теперь нашей задачей становится организация таких экстремальных и разрозненных слоев параллелизации, чтобы добиться как можно большего ускорения.

В этом пошаговом ознакомлении с высокоуровневым приложением мы опустили массу важных деталей, которые рассмотрим в последующих главах. Но даже этот номинальный уровень детализации подчеркивает несколько стратегий для выявления параллелизации алгоритма. Для того чтобы иметь возможность разрабатывать похожие стратегии для решения других задач, необходимо понимание современного ап-

паратного и программного обеспечения. Теперь мы глубже погрузимся в текущие аппаратные и программные модели. Эти концептуальные модели являются упрощенными представлениями разнообразного реального аппаратного обеспечения, чтобы избежать сложности и сохранить общность в быстро эволюционирующих системах.

1.3.2 Аппаратная модель для современных гетерогенных параллельных систем

С целью получения базового понимания принципа работы параллельных вычислений мы объясним компоненты современного оборудования. Начнем с того, что в динамической оперативной памяти, именуемой DRAM, хранится информация, или данные. Ядро процессора, или просто ядро, выполняет арифметические операции (сложение, вычитание, умножение, деление), оценивает логические инструкции, загружает и сохраняет данные из DRAM. Когда операция выполняется на данных, команды и данные загружаются из памяти в ядро, обрабатываются и сохраняются обратно в память. Современные CPU, часто именуемые просто процессорами, оснащены многочисленными ядрами, способными исполнять эти операции в параллельном режиме. Также получают распространение системы, оснащенные ускорительным оборудованием, таким как GPU-процессоры. GPU-процессоры оснащены тысячами ядер и пространством памяти, которое отделено от DRAM центрального процессора, CPU.

Комбинация процессора (или двух), DRAM и ускорителя составляет вычислительный узел, на который можно ссылаться в контексте одного домашнего настольного компьютера или «стойки» в суперкомпьютере. Вычислительные узлы могут соединяться друг с другом одной или несколькими сетями. Такое соединение иногда именуется межсоединением. Концептуально узел запускает один экземпляр операционной системы, который управляет и контролирует все аппаратные ресурсы. Поскольку оборудование становится все сложнее и неоднороднее, мы начнем с упрощенных моделей компонентов системы, чтобы каждый из них был более очевидным.

Архитектура на основе распределенной памяти: перекрестно-узловой параллельный метод

Одним из первых и наиболее масштабируемых подходов к параллельным вычислениям является кластер распределенной памяти (рис. 1.15). Каждый CPU имеет свою собственную локальную память, состоящую из DRAM, и соединен с другими CPU коммуникационной сетью. Хорошая масштабируемость кластеров распределенной памяти обусловлена их кажущейся безграничной способностью включать в себя большее число узлов.

Эта архитектура также обеспечивает некоторую локальность памяти, разделяя суммарную адресную память на меньшие подпространства для

каждого узла, что делает доступ к памяти вне узла явно отличным от доступа на узле. Это вынуждает программиста явно обращаться к разным участкам памяти. Недостатком этой архитектуры является то, что программист должен управлять подразделением пространств памяти в самом начале приложения.

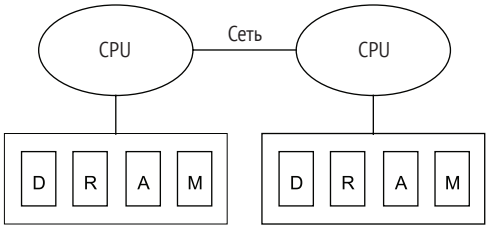


Рис. 1.15 Архитектура на основе распределенной памяти связывает узлы, состоящие из отдельных пространств памяти. Этими узлами могут быть рабочие станции или стойки

Архитектура на основе совместной памяти: наузловой параллельный метод

Альтернативный подход присоединяет два CPU напрямую к одной и той же совместной памяти (рис. 1.16). Сила этого подхода заключается в том, что процессоры используют совместно одно и то же адресное пространство, что упрощает программирование. Но он вводит потенциальные конфликты памяти, что приводит к проблемам с правильностью и производительностью. Синхронизирование доступа к памяти и значений между CPU или обрабатывающими ядрами на мультиядерном CPU усложнено и обходится дорого.

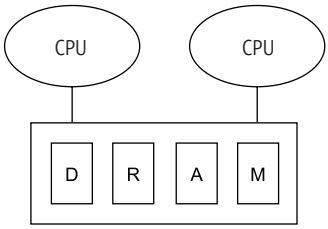


Рис. 1.16 Архитектура на основе совместной памяти обеспечивает параллелизацию внутри узла

Добавление большего числа CPU и обрабатывающих ядер не увеличивает объем доступной приложению памяти. Эти затраты и затраты на синхронизацию лимитируют масштабируемость архитектуры на основе совместной памяти.

Модули векторной обработки: несколько операций с одной командой

Почему бы просто не увеличить тактовую частоту процессора, чтобы получить более выскую мощность, как это делалось в прошлом? Самым большим ограничением в увеличении тактовой частоты CPU является то, что он требует больше энергии и выделяет больше тепла. Будь то супер-

компьютерный HPC-центр с пределами на инсталлированные силовые линии либо ваш мобильный телефон с лимитированной емкостью батареи, все устройства сегодня имеют пределы по электропитанию. Эта проблема называется *силовой стеной*.

Вместо того, чтобы увеличивать тактовую частоту, почему бы не выполнять более одной операции за цикл? Это идея лежит в основе возрождения векторизации на многих процессорах. Для выполнения нескольких операций в модуле векторной обработки требуется лишь немного больше энергии, чем для одной операции (более формально именуемой *скалярной операцией*). С помощью векторизации мы можем обрабатывать больше данных за один тактовый цикл, чем при последовательном процессе. Требования к питанию для нескольких операций практически не меняются (по сравнению с одной), а сокращение времени выполнения может привести к снижению энергопотребления приложения. Во многом подобно четырехполосной автостраде, которая позволяет четырем автомобилям двигаться одновременно, по сравнению с однополосной дорогой, векторная операция обеспечивает более высое значение мощности обработки. И действительно четыре маршрута через модуль векторной обработки, показанные в разных оттенках на рис. 1.17, обычно называются *полосами* векторной операции.

Большинство CPU и GPU имеет некоторые способности по векторизации или эквивалентные операции. Объем данных, обрабатываемых за один тактовый цикл, *длина вектора*, зависит от размера модулей векторной обработки на процессоре. В настоящее время наиболее распространенная длина вектора составляет 256 бит. Если дискретизированные данные равны 64-битным значениям двойной точности, то мы можем выполнять четыре операции с плавающей точкой одновременно как векторную операцию. Как показано на рис. 1.17, аппаратные модули векторной обработки загружают по одному блоку данных за раз, одновременно выполняют одну операцию с данными, а затем сохраняют результат.

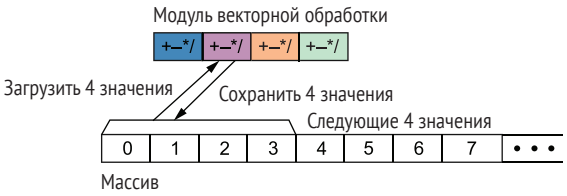


Рис. 1.17 Пример векторной обработки, при которой манипулирование четырьмя векторными модулями массива осуществляется одновременно

Ускорительное устройство: узкоцелевой надстраиваемый процессор

Ускорительное устройство – это дискретное оборудование, предназначенное для быстрого исполнения специфических задач. Наиболее распространенным ускорителем является GPU. При использовании для вычислений это устройство иногда называют общецелевым графическим процессором (general-purpose GPU или GPGPU). GPU содержит много ма-

лых обрабатывающих ядер, именуемых потоковыми мультипроцессорами (SM). Хотя SM-процессоры проще, чем ядро CPU, они обеспечивают огромный объем вычислительной мощности. Обычно малый интегрированный GPU можно найти прямо на CPU.

Большинство современных компьютеров также имеет отдельный дискретный GPU, подключенный к CPU интерфейсной шиной для подключения периферийных компонентов (PCI-шиной) (рис. 1.18). Эта шина увеличивает затраты на передачу данных и команд, но дискретная карта часто является более мощной, чем интегрированное устройство. Например, в системах высокого класса NVIDIA использует NVLink, а AMD Radeon использует свою Infinity Fabric для снижения затрат на передачу данных, но эти затраты по-прежнему значительны. Мы подробнее обсудим эту интересную архитектуру GPU в главах 9–12.

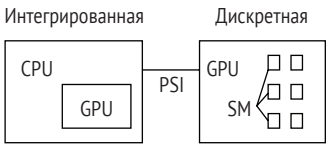


Рис. 1.18 GPU-процессоры бывают двух видов: интегрированные и дискретные. Дискретные или выделенные GPU обычно имеют большое число потоковых мультипроцессоров и собственную DRAM. Для доступа к данным на дискретном GPU требуется связь по шине PCI

Общая гетерогенная параллельная архитектурная модель

Теперь давайте скомбинируем все эти разные аппаратные архитектуры в одну модель (рис. 1.19). Два узла, каждый с двумя CPU, совместно используют одну и ту же память DRAM. Каждый CPU является двухъядерным процессором с интегрированным GPU. Дискретный GPU на шине PCI также подключается к одному из CPU. Хотя CPU используют основную память совместно, они обычно находятся в разных участках неравномерного доступа к памяти (NUMA). Это означает, что доступ к памяти второго CPU обходится дороже, чем доступ к его собственной памяти.

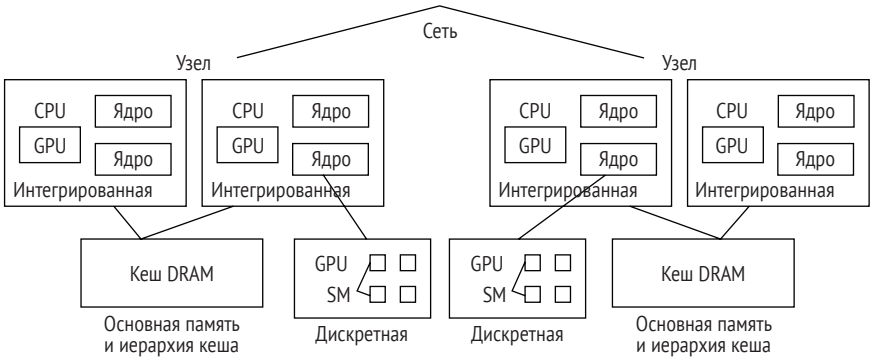


Рис. 1.19 Общая гетерогенная параллельная архитектурная модель, состоящая из двух узлов, соединенных сетью. Каждый узел имеет мультиядерный CPU с интегрированным и дискретным GPU и некоторой памятью (DRAM). Современное вычислительное оборудование обычно имеет ту или иную расстановку этих компонентов

На протяжении всего этого обсуждения оборудования мы упоминали упрощенную модель иерархии памяти, показывающую только DRAM или основную память. Мы показали кеш в комбинированной модели (рис. 1.19), но без подробностей о его составе или о том, как он функционирует. Мы оставляем наше обсуждение сложностей управления памятью, включая несколько уровней кеша, для главы 3. В этом разделе мы просто представили модель современного оборудования, чтобы помочь вам идентифицировать имеющиеся компоненты, чтобы иметь возможность выбрать параллельную стратегию, наиболее подходящую для вашего приложения и оборудования.

1.3.3 Прикладная/программная модель для современных гетерогенных параллельных систем

Программная модель для параллельных вычислений с необходимостью обуславливается опорным оборудованием, но тем не менее отличается от него. Операционная система обеспечивает интерфейс между ними. Параллельные операции не возникают сами по себе; вернее сказать, исходный код должен указывать на то, как распараллеливать работу, порождая процессы или потоки; выгружая данные, работу и команды на вычислительное устройство или оперируя на блоках данных одновременно. Программист должен сначала выявить параллелизацию, определить наилучший технический прием для функционирования в параллельном режиме, а затем в явной форме направить его работу безопасным, правильным и эффективным способом. Следующие ниже методы являются наиболее распространенными техническими приемами параллелизации. Далее мы подробно рассмотрим каждый из них.

- *Параллелизация на основе процессов* – передача сообщений.
- *Параллелизация на основе потоков* – совместные данные через память.
- *Векторизация* – несколько операций с одной командой.
- *Обработка в потоковом режиме* – через специализированные процессоры.

Параллелизация на основе процессов: передача сообщений

Подход на основе передачи сообщений был разработан для архитектур на основе распределенной памяти, в которых для перемещения данных между процессами используются явные сообщения. В этой модели ваше приложение создает отдельные процессы, именуемые *рангами* в передаче сообщений, с собственным пространством памяти и конвейером команд (рис. 1.20). На рисунке также показано, что процессы передаются в OS для размещения на процессорах. Приложение обитает в части диаграммы, помеченной как пользовательское пространство, где у пользователя есть разрешение на работу. Часть ниже – это ядерное пространство, которое защищено от опасных операций со стороны пользователя.

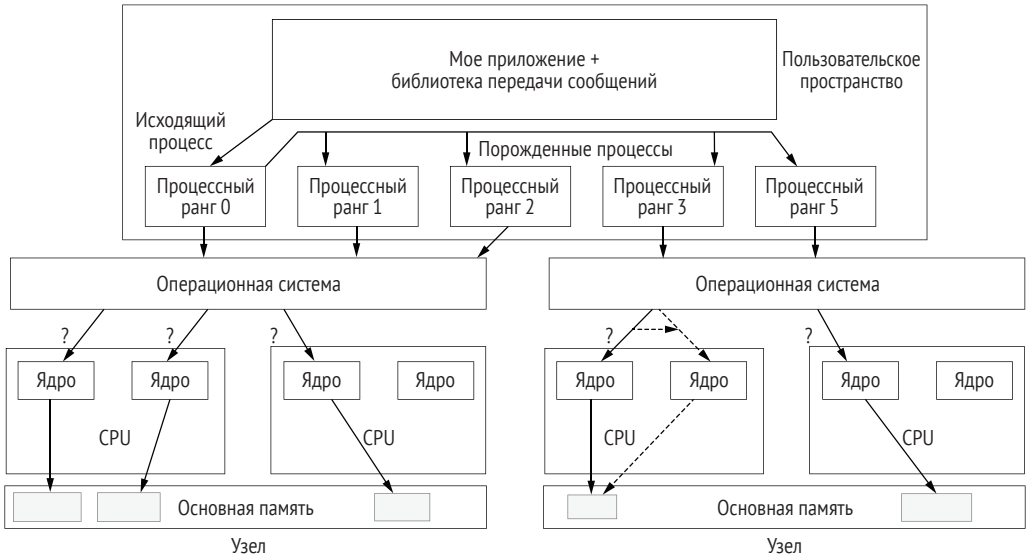


Рис. 1.20 Библиотека передачи сообщений порождает процессы. Операционная система размещает процессы на ядрах двух узлов. Вопросительные знаки указывают на то, что OS управляет размещением процессов и может перемещать их во время выполнения, как указано пунктирными стрелками. OS также выделяет память под каждый процесс из основной памяти узла

Имейте в виду, что процессоры – CPU – имеют несколько обрабатывающих ядер, которые не эквивалентны процессам. Процессы – это концепция операционной системы, а процессоры – это аппаратный компонент. Запуск любого числа порождаемых приложением процессов планируется операционной системой для обрабатывающих ядер. На самом деле вы можете запустить восемь процессов на своем четырехъядерном ноутбуке, и они будут просто переключаться между обрабатывающими ядрами. По этой причине были разработаны механизмы, указывающие операционной системе на то, как размещать процессы и следует ли «привязывать» процесс к обрабатывающему ядру. Контролирование привязки подробнее обсуждается в главе 14.

В целях перемещения данных между процессами вам нужно будет запрограммировать явные сообщения в своем приложении. Эти сообщения могут отправляться по сети либо через совместную память. В 1992 году многие библиотеки передачи сообщений образовали стандарт под названием «Интерфейс передачи сообщений» (MPI). С тех пор MPI занял эту нишу и присутствует почти во всех параллельных приложениях, масштабируемых за пределами одного узла. И – да, вы также найдете много разных имплементаций библиотек MPI.

Распределенные вычисления против параллельных вычислений

В некоторых параллельных приложениях используется более низкоуровневый подход к параллелизации, именуемый распределенными вычислениями. Мы определяем *распределенные вычисления* как множество слабо сцеплен-

ных процессов, которые взаимодействуют с помощью вызовов на уровне операционной системы. Хотя распределенные вычисления являются подмножеством параллельных вычислений, это различие важно понимать. Примерами приложений для распределенных вычислений являются одноранговые сети, Всемирная паутина и интернет-почта. Поиск внеземного разума (Search for Extraterrestial Intelligence, SETI@home) является лишь одним из примеров многих научных приложений для распределенных вычислений.

Место каждого процесса обычно находится на отдельном узле и создается с помощью операционной системы с использованием чего-то вроде удаленного вызова процедур (RPC) либо сетевого протокола. Затем процессы обмениваются данными посредством передачи сообщений между процессами посредством *межпроцессного взаимодействия* (IPC), которого существует несколько разновидностей. В простых параллельных приложениях часто используется подход на основе распределенных вычислений, но нередко с помощью языка более высокого уровня, такого как Python, и специализированных параллельных модулей или библиотек.

Параллизация на основе потоков: совместные данные через память

Подход на основе потоков (т. е. на основе threads – виртуальных ядер) к параллелизации порождает отдельные указатели команд в рамках одного и того же процесса (рис. 1.21). В результате вы можете легко делиться порциями процессной памяти между потоками. Но это сопровождается подвохами, связанными с правильностью и производительностью. Программисту остается определять разделы набора команд и данных, которые независимы и могут поддерживать потокообразование. Эти соображения подробнее обсуждаются в главе 7, где мы рассмотрим OpenMP, одну из ведущих систем потокообразования. OpenMP предоставляет возможность порождать потоки и распределять работу между этими потоками.

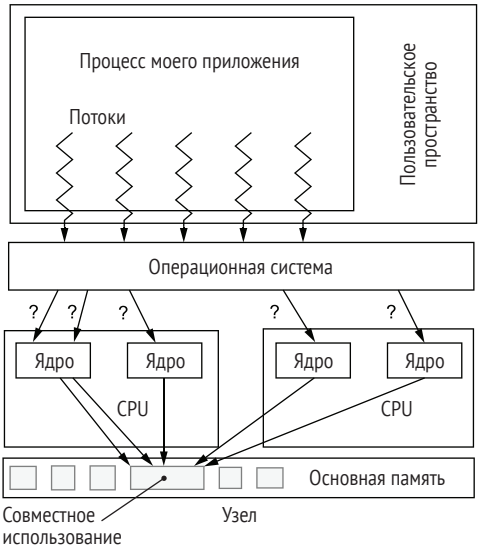


Рис. 1.21 Процесс приложения в поточном (thread-based) подходе к параллелизации порождает потоки (threads). Указанные потоки ограничены доменом узла. Вопросительные знаки показывают, что OS выбирает место размещения потоков. Некоторая память используется между потоками совместно

Существуют самые разнообразные подходы к потокообразованию, от тяжеловесных до легковесных, управляемых пользовательским пространством либо операционной системой. Хотя системы потокообразования лимитированы масштабированием в пределах одного узла, они являются привлекательным вариантом для умеренного ускорения. Однако пределы памяти одного узла имеют более серьезные последствия для приложения.

Векторизация: несколько операции с одной командой

Векторизирование приложения бывает гораздо эффективнее с точки зрения затрат, чем расширение вычислительных ресурсов в центре HPC, и этот метод бывает абсолютно необходим на портативных устройствах, таких как мобильные телефоны. При векторизировании работа выполняется блоками по 2–16 элементов данных за один раз. Более формальным термином для этой классификации операций является «Одна команда, несколько элементов данных» (single instruction, multiple data, SIMD). Термин SIMD часто используется, когда речь заходит о векторизации. SIMD – это всего лишь одна из категорий параллельных архитектур, которые будут обсуждаться далее в разделе 1.4.

Инициирование векторизации из пользовательского приложения чаще всего выполняется с помощью прагм исходного кода или с помощью компиляторного анализа. Прагмы и директивы – это подсказки, которые даются компилятору с целью сориентировать его в отношении параллелизирования или векторизирования раздела исходного кода. Как прагмы, так и компиляторный анализ сильно зависят от способностей компилятора (рис. 1.22). Здесь мы зависим от компилятора, где предыдущие параллельные механизмы зависели от операционной системы. Кроме того, без явно заданных компиляторных флагов сгенерированный код предназначен для наименее мощного процессора и длины вектора, что значительно снижает эффективность векторизации. Существуют механизмы, с помощью которых компилятор можно обойти, но они требуют гораздо больших усилий по программированию и не являются переносимыми.

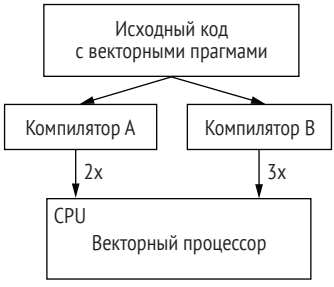


Рис. 1.22 Векторные команды в исходном коде, возвращающие разные уровни производительности из компиляторов

Обработка потоков операций посредством специализированных процессоров

Обработка в потоковом режиме (stream processing) – это концепция циркуляции данных, в которой поток данных обрабатывается более простым узкоцелевым процессором. Долгое время применявшаяся во встроенных вычислениях, эта технология была адаптирована под визуальную обработку крупных наборов геометрических объектов для компьютерных дисплеев в специализированном процессоре, GPU. Эти GPU наполнялись широким набором арифметических операций и несколькими потоковыми мультипроцессорами (SM) для обработки геометрических данных в параллельном режиме. Научные программисты вскоре нашли способы адаптировать обработку потоков данных к крупным наборам симуляционных данных, таким как ячейки, расширив роль GPU до GPGPU.

На рис. 1.23 показаны данные и вычислительное ядро, выгруженные по шине PCI в GPU для вычислений. GPU-процессоры по-прежнему лимитированы в функциональности, по сравнению с CPU, но там, где можно использовать специализированную функциональность, они обеспечивают исключительные вычислительные способности при более низких требованиях к мощности. К этой категории подходят и другие специализированные процессоры, хотя в наших обсуждениях мы сосредоточимся на GPU.

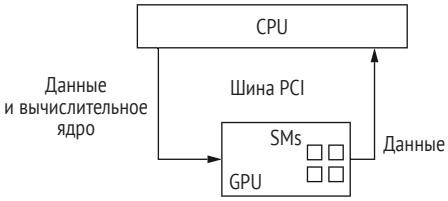


Рис. 1.23 В подходе на основе обработки потоков данных данные и вычислительное ядро выгружаются в GPU и его потоковые мультипроцессоры. Обработанные или выходные данные передаются обратно в CPU для файлового ввода-вывода или другой работы

1.4 Классифицирование параллельных подходов

Если вы прочитаете больше о параллельных вычислениях, то столкнетесь с такими аббревиатурами, как SIMD (Одна команда, несколько элементов данных) и MIMD (Несколько команд, несколько элементов данных). Эти термины относятся к категориям компьютерных архитектур, предложенных Майклом Флинном (Michael Flynn) в 1966 году, что потом стало известно как *таксономия Флинна*. Эти классы помогают по-разному рассматривать потенциальную параллелизацию в архитектурах. Указанная классификация основана на разбивке команд и данных на одиночные либо многочисленные операции (рис. 1.24). Имейте в виду, что, хотя эта таксономия и полезна, некоторые архитектуры и алгоритмы не очень хорошо вписываются в категорию. Ее полезность заключается в распознавании шаблонов в таких категориях, как SIMD, у которых могут иметься трудности с условными инструкциями. Это связано с тем, что каждый

элемент данных может нуждаться в другом блоке кода, но потоки должны исполнять ту же самую команду.

		Команда	
		Одиночная	Многочисленная
Данные	Одиночные	SISD Одна команда, один элемент данных	MISD Многочисленные команды, один элемент данных
	Многочисленные	SIMD Одна команда, многочисленные элементы данных	MIMD Многочисленные команды, один многочисленные элементы данных

Рис. 1.24 Таксономия Флинна классифицирует разные параллельные архитектуры. Последовательная архитектура – это один элемент данных, одна команда (SISD). Две категории имеют лишь частичную параллелизацию в том смысле, что либо команды, либо данные являются параллельными, но другая часть является последовательной

В случае, когда существует более одной последовательности команд, категория называется «Несколько команд, один элемент данных» (multiple instruction, single data, MISD). Эта архитектура не так распространена; самым лучшим примером может служить избыточное вычисление на одних и тех же данных. Оно используется в высокоустойчивых подходах, таких как контроллеры космических аппаратов. Поскольку космические аппараты находятся в условиях высокой радиации, они часто выполняют две копии каждого расчета и сравнивают результаты обоих.

Векторизация является ярким примером SIMD, в которой одна и та же команда исполняется для многочисленных элементов данных. Вариантом SIMD является «Одна команда, мультипоток» (single instruction, multi-thread, SIMT), который широко используется для описания рабочих групп GPU.

Последняя категория имеет параллелизацию как в командах, так и в данных и называется MIMD. Эта категория описывает мультиядерные параллельные архитектуры, которые составляют большинство крупных параллельных систем.

1.5 Параллельные стратегии

До сих пор в нашем первоначальном примере в разделе 1.3.1 мы рассматривали параллелизацию данных для ячеек или пикселов. Но параллелизация данных может использоваться и для частиц и других объектов данных. Параллелизация данных является наиболее распространенным подходом и часто самым простым. По сути, каждый процесс выполняет одну и ту же программу, но оперирует уникальным подмножеством

данных, как показано в правом верхнем углу рис. 1.25. Параллельный подход к обработке данных имеет то преимущество, что он хорошо масштабируется по мере того, как увеличивается размер задачи и число процессоров.

Еще один подход представлен *параллелизмом на уровне операционных задач*. Это включает в себя главный контроллер со стратегиями на основе потоков-работников (worker threads), конвейера или ведерной бригады, также показанными на рис. 1.25. Подход в виде трубопровода (т. е. как бы в виде трубы, по которой равномерно течет вода) используется в суперскалярных процессорах, где вычисления адресов и целых чисел выполняются отдельным логическим модулем, а не обработчиком данных с плавающей точкой, что позволяет выполнять эти вычисления в параллельном режиме. В ведерной бригаде (т. е. как бы в виде цепочки людей, передающих ведра с водой на пожаре) каждый процессор используется для обработки и преобразования данных в последовательности операций. При подходе на основе главного работника один процессор планирует и распределяет задачи для всех работников, и каждый работник проверяет наличие следующего элемента работы, возвращая предыдущую выполненную задачу. Также существует возможность комбинировать разные параллельные стратегии, чтобы выявить более высокую степень параллелизма.

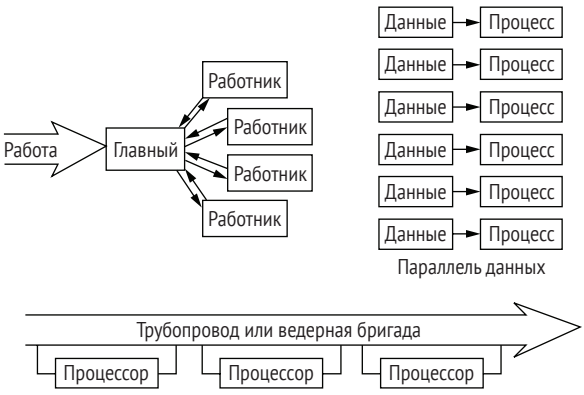


Рис. 1.25 Различные стратегии параллельности задач и данных, включая главного работника, конвейер или ведерную бригаду, и параллелизм данных

1.6 Параллельное ускорение против сравнительного ускорения: две разные меры

На протяжении всей этой книги мы будем представлять ряд сравнительных показателей производительности и ускорений. Нередко термин «ускорение» используется для сравнения двух разных времен выполнения с небольшим объяснением или контекстом в целях полного понима-

ния того, что это значит. Ускорение является общим термином, который используется во многих контекстах, например для количественного оценивания эффектов оптимизации. В целях прояснения разницы между двумя главными категориями показателей параллельной производительности мы определим два разных термина.

- *Параллельное ускорение*. На самом деле мы должны назвать этот термин параллельным ускорением по сравнению с последовательным. Ускорение происходит, по сравнению с базовым последовательным прогоном на стандартной платформе, обычно на одном CPU. Параллельное ускорение может обуславливаться работой на GPU либо с пакетом OpenMP, либо MPI на всех ядрах узла компьютерной системы.
- *Сравнительное ускорение*. На самом деле мы должны назвать этот термин сравнительным ускорением между архитектурами. Обычно это сравнение производительности между двумя параллельными имплементациями либо другое сравнение между достаточно ограниченными комплектами оборудования. Например, оно может быть между параллельной имплементацией MPI на всех ядрах узла компьютера в сравнении с GPU-процессором(ами) на узле.

Эти две категории сравнений производительности представляют две разные цели. Первая – понять, насколько можно ускорить процесс, добавив тот или иной тип параллелизма. Однако это сравнение будет необъективным между архитектурами. Речь идет о параллельном ускорении. Например, сравнение времени работы GPU с последовательным прогоном CPU не является объективным сравнением между мультиядерным CPU и GPU. Сравнительные ускорения между архитектурами более уместны при попытке сравнить мультиядерный CPU с производительностью одного или нескольких GPU на узле.

В последние годы эти две архитектуры были нормализованы, вследствие чего относительная производительность сравнивается для схожих требований к мощности или энергии, а не для произвольного узла. Тем не менее существует столь много разных архитектур и возможных комбинаций, что для обоснования вывода можно получать любые показатели производительности. Вы можете подобрать быстрый GPU и медленный CPU либо четырехъядерный CPU для сравнения с 16-ядерным процессором. Поэтому мы предлагаем вам добавлять следующие ниже термины в скобках для сравнения производительности, чтобы придавать им больший контекст.

- Добавлять «(лучшее в 2016 году)» в каждый термин. Например, параллельное ускорение (лучшее в 2016 году) и сравнительное ускорение (лучшее в 2016 году) указывают на то, что сравнение проводится между лучшим оборудованием, выпущенным в определенном году (в данном примере в 2016 году), где вы можете сравнивать высококлассный GPU с высококлассным CPU.
- Добавлять «(общедоступное в 2016 году)» или «(2016)», если две архитектуры были выпущены в 2016 году, но не являются оборудо-

ванием высшего класса. Это бывает актуально для разработчиков и пользователей, у которых больше массовых компонентов, чем в топовых системах.
- Добавлять «(Mac 2016 года)», если GPU и CPU были выпущены в ноутбуке или настольном компьютере Mac 2016 года или в чем-то подобном для других брендов с фиксированными компонентами в течение определенного периода времени (в данном примере в 2016 году). Сравнение производительности такого типа полезно для пользователей общедоступной системы.
- Добавлять «(GPU 2016:CPU 2013)», чтобы показывать, что существует возможное несоответствие в годе выпуска оборудования (в данном примере 2016 год по сравнению с 2013 годом) у сравниваемых компонентов.
- В сравнительные данные никаких квалификаций не добавляется. Кто знает, что эти цифры означают?

Из-за резкого роста моделей CPU и GPU показатели производительности обязательно будут больше похожи на сравнение яблок и апельсинов, чем на четко определенную метрику. Но для более формальных условий сравнения мы должны, по крайней мере, указывать характер сравнения, чтобы другие лучше понимали смысл цифр и чтобы быть объективнее к поставщикам оборудования.

1.7 Чему вы научитесь в этой книге?

Эта книга написана, имея в виду разработчика прикладного кода, и никаких предварительных знаний о параллельных вычислениях не предполагается. У вас просто должно быть желание повысить производительность и масштабируемость вашего приложения. Области применения включают научные вычисления, машинное обучение и анализ больших данных в системах, начиная от настольных компьютеров и заканчивая крупнейшими суперкомпьютерами.

В целях извлечения пользы из этой книги в полной мере читатели должны быть опытными программистами, предпочтительно владеющими компилируемым языком HPC, таким как C, C++ или Fortran. Мы также исходим из наличия элементарный знаний аппаратных архитектур. В дополнение к этому читатели должны быть знакомы с терминами компьютерных технологий, такими как биты, байты, операции, кеш, оперативная память и т. д. Также полезно иметь базовое понимание функций операционной системы и того, как она управляет аппаратными компонентами и взаимодействует с ними. После прочтения этой книги вы приобретете несколько навыков, в том числе:

- определение того, когда передача сообщений (MPI) более уместна, чем потокообразование (пакет OpenMP), и наоборот;
- оценивание того, насколько возможно ускорение при векторизации;

- распознавание того, какие разделы вашего приложения обладают наибольшим потенциалом для ускорения;
- принятие решения о том, когда бывает полезно использовать GPU для ускорения вашего приложения;
- установление максимальной потенциальной производительности для вашего приложения;
- оценивание энергозатрат для вашего приложения.

Даже после этой первой главы вы должны почувствовать себя комфортно с разными подходами к параллельному программированию. Мы предлагаем вам прорабатывать упражнения в каждой главе, которые помогут вам интегрировать многие вводимые нами концепции. Если вы начинаете чувствовать себя немного подавленным сложностью современных параллельных архитектур, то вы не одиноки. Сложно ухватить все возможности сразу. В следующих главах мы будем разбирать их по частям, чтобы вам было проще.

1.7.1 Дополнительное чтение

Хорошее базовое введение в параллельные вычисления можно найти на веб-сайте Национальной лаборатории Лоуренса Ливермора:

- Блейз Барни, «Введение в параллельные вычисления» (Blaise Barney, Introduction to Parallel Computing), https://computing.llnl.gov/tutorials/parallel_comp/.

1.7.2 Упражнения

1. Каковы другие примеры параллельных операций в вашей повседневной жизни? Как бы вы классифицировали свой пример? Под что, по вашему мнению, оптимизируется параллельный дизайн? Можете ли вы вычислить параллельное ускорение для этого примера?
2. Какова теоретическая мощность параллельной обработки вашей системы (будь то настольный компьютер, ноутбук или мобильный телефон) по сравнению с ее мощностью последовательной обработки? Какие виды параллельного оборудования в ней присутствуют?
3. Какие параллельные стратегии вы видите в примере с оплатой покупок в магазине на рис. 1.1? Существуют ли какие-то нынешние параллельные стратегии, которые не показаны? Как насчет примеров из упражнения 1?
4. У вас есть приложение для обработки изображений, которому необходимо ежедневно обрабатывать 1000 изображений размером 4 мебибайт (МiВ, 2^{20}, или 1 048 576 байт) каждое. Для последовательной обработки каждого изображения требуется 10 мин. Ваш кластер состоит из мультиядерных узлов с 16 ядрами и общим объемом 16 гибибайт (ГиБ, 2^{30} байт, или 1024 мебибайт) основной памяти в расчете на узел. (Обратите внимание, что мы используем правильные двоичные термины МиБ и ГиБ, а не Мб и Гб, которые являются метрическими терминами соответственно для 10^6 и 10^9 байт.)

a Какой дизайн параллельной обработки лучше всего справляется с этой рабочей нагрузкой?
b Теперь потребительский спрос увеличивается в 10 раз. Справляется ли с этим ваш дизайн? Какие изменения вам пришлось бы внести?

5 Процессор Intel Xeon E5-4660 имеет расчетную тепловую мощность 130 Вт; это средняя потребляемая мощность при использовании всех 16 ядер. GPU NVIDIA Tesla V100 и GPU AMD MI25 Radeon имеют расчетную тепловую мощность 300 Вт. Предположим, вы портируете свое программное обеспечение для использования одного из этих GPU. Насколько быстрее должно работать ваше приложение на GPU, чтобы считаться более энергоэффективным, чем ваше приложение с 16-ядерным CPU?

Резюме

- Поскольку наступила эпоха, когда большая часть вычислительных способностей оборудования доступна только через параллелизм, программисты должны хорошо разбираться в технических приемах, используемых для эксплуатирования параллелизма.
- Приложения должны иметь параллельную работу. Самая важная задача параллельного программиста заключается в выявлении большего параллелизма.
- Усовершенствования оборудования всецело касается почти тотального усовершенствоваания параллельных компонентов. Опора на повышение последовательной производительности не приведет к ускорению в будущем. Ключ к повышению производительности приложений будет лежать в параллельной сфере.
- Появляются самые разные языки параллельного программного обеспечения, которые помогают получать доступ к возможностям оборудования. Программисты должны разбираться в том, какие из них подходят для разных ситуаций.

2 Планирование под параллелизацию

Эта глава охватывает следующие ниже темы:

- шаги планирования параллельного проекта;
- версионный контроль и рабочие потоки коллективной разработки;
- понимание возможностей и пределов производительности;
- разработку плана распараллеливания процедуры.

Разработка параллельного приложения или придание существующему приложению способности выполняться в параллельном режиме поначалу может показаться сложной. Нередко разработчики, которые делают в параллелизме только первые шаги, не уверены в том, с чего начинать и с какими подводными камнями они могут столкнуться. В этой главе основное внимание уделяется модели рабочего потока программиста(ов) по разработке параллельных приложений, как показано на рис. 2.1. Эта модель предлагает контекст для того, с чего начинать и как поддерживать прогресс в разработке вашего параллельного приложения. Как правило, лучше всего имплементировать параллелизм малыми шагами, чтобы в случае возникновения проблем иметь возможность откатывать назад последние несколько фиксаций (коммитов). Такой шаблон подходит для технолгии гибкого управления проектами.

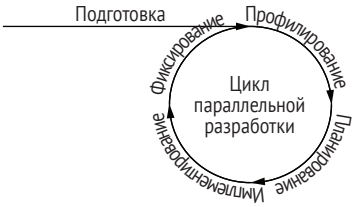


Рис. 2.1 Предлагаемый нами рабочий поток параллельной разработки начинается с подготовки приложения и последующего повторения четырех шагов для постепенной параллелизации приложения. Указанный рабочий поток особенно подходит для технологии гибкого управления проектами

Давайте представим, что вам был назначен новый проект для ускорения и параллелизации приложения с пространственной вычислительной сеткой, представленной на рис. 1.9 (пример с вулканом Кракатау). Это может быть алгоритм обнаружения изображений, научная симуляция пеплового шлейфа или модель результирующих волн цунами, или все три из них. Какие шаги вы предпримите, чтобы получить успешный проект обеспечения параллелизма?

Очень заманчиво просто окунуться в проект. Но без обдумывания и подготовки вы значительно уменьшаете свои шансы на успех. Для начала вам понадобится проектный план для этих усилий по обеспечению параллелизма, поэтому здесь мы начнем с высокоуровневого обзора шагов этого рабочего потока. Затем, по мере продвижения по этой главе, мы будем углубляться в подробности каждого шага, уделяя особое внимание характеристикам, типичным для параллельного проекта.

Быстрая разработка: рабочий поток параллельного проекта

Сначала вам нужно подготовить свой коллектив и приложение для быстрой разработки. Поскольку у вас есть существующее последовательное приложение, которое работает с пространственной вычислительной сеткой, показанной на рис. 1.9, вероятно, будет иметься много небольших изменений с частыми тестами с целью обеспечения неизменности результатов. Подготовка кода включает в себя настройку версионного контроля, разработку тестового комплекта и обеспечение качества и переносимости кода. Подготовка коллектива будет сосредоточена вокруг процессов разработки. Как всегда, управление проектом будет касаться решения оперативных задач и контроля сферы охвата.

В целях подготовки почвы для цикла разработки вам необходимо определить способности имеющихся вычислительных ресурсов, требования вашего приложения и требования к производительности. Сравнительное тестирование систем помогает определять пределы вычислительных ресурсов, тогда как профилирование помогает понимать требования приложения и его наиболее дорогостоящие вычислительные ядра (kernels). *Вычислительные ядра* относятся к разделам приложения, которые являются как вычислительно интенсивными, так и концептуально автономными.

Из ядерных профилей вы будете планировать задачи для распараллеливания подпрограмм и имплементирования изменений. Имплементационный этап завершается только после того, как процедура была распараллелена и код поддерживает переносимость и правильность. Удовлетворив эти требования, все изменения будут внесены в систему версионного контроля. После внесения дополнительных изменений процесс снова начинается с профиля приложения и вычислительного ядра.

2.1 На подступах к новому проекту: подготовка

На рис. 2.2 представлены рекомендуемые компоненты на подготовительном шаге. Это те элементы, которые оказались важными именно для проектов параллелизации.

Рис. 2.2 Рекомендуемые компоненты подготовки решают задачи, важные для разработки параллельного кода

На этом этапе вам нужно будет настроить версионный контроль, разработать тестовый комплект для вашего приложения и очистить существующий код. Версионный контроль позволяет отслеживать изменения, которые вы вносите в свое приложение во временной динамике. Это позволяет вам позже быстро откатывать ошибки и отслеживать дефекты в своем коде. Тестовый комплект позволяет вам проверять правильность вашего приложения при каждом внесенном в код изменении. В сцепке с версионным контролем он может стать мощной основой для быстрой разработки вашего приложения.

Имея версионный контроль и тестирование кода на положенном месте, теперь вы можете заняться задачей очистки своего кода. Хороший код легко модифицировать и расширять, и он не проявляет непредсказуемого поведения. Хороший, чистый код обеспечивается за счет модульности и проверки на наличие проблем с памятью. *Модульность* означает, что вы имплементируете вычислительные ядра как независимые процедуры или функции с четко определенными данными на входе и выходе. Трудности с памятью могут включать утечку памяти, доступ к памяти за пределами границ и использование неинициализированной памяти. Начало параллельной работы с предсказуемым и качественным кодом способствует быстрому прогрессу и предсказуемым циклам разработки. Трудно сопоставлять ваш последовательный код, если изначальные результаты обусловлены ошибкой программирования.

Наконец, вы захотите убедиться, что ваш код *переносим*. Это означает, что он может компилироваться несколькими компиляторами. Нали-

чие и поддержание компиляторной переносимости позволяет вашему приложению ориентироваться на дополнительные платформы, помимо той, которую вы, возможно, в настоящее время имеете в виду. Кроме того, опыт показывает, что разработка кода для работы с несколькими компиляторами помогает находить дефекты до того, как они будут зафиксированы в версионной истории вашего кода. Поскольку ландшафт высокопроизводительных вычислений быстро меняется, переносимость позволит вам гораздо быстрее адаптироваться к изменениям в будущем.

Нет ничего необычного в том, что время подготовки конкурирует с временем, которое тратится на фактический параллелизм, в особенности для многосложного кода. Включение этой подготовки в объем и сроки вашего проекта позволяет избежать разочарований в ходе выполнения вашего проекта. В этой главе мы исходим из того, что вы начинаете с последовательного или прототипного приложения. Тем не менее вы все равно можете извлечь выгоду из этой стратегии рабочего потока, даже если вы уже начали параллелизировать свой код. Далее мы обсудим четыре упомянутых выше компонента подготовки проекта.

2.1.1 Версионный контроль: создание безопасного хранилища для своего параллельного кода

С учетом многочисленных изменений, происходящих во время работы над параллелизмом, неизбежны ситуации, когда вы внезапно обнаружите, что код нарушен либо возвращает другие результаты. Критически важно иметь возможность выйти из этой ситуации, выполнив резервное копирование рабочей версии.

> **ПРИМЕЧАНИЕ** Прежде чем начинать какую-либо работу над параллелизмом, проверьте, какой тип версионного контроля используется для вашего приложения.

У вашего проекта по обнаружению изображений в нашем сценарии вы обнаруживаете наличие системы версионного контроля. Но модель пепельного шлейфа никогда не имела никакого версионного контроля.

По мере того как вы копаете глубже, вы обнаруживаете, что на самом деле в различных каталогах разработчиков имеется четыре версии кода пепельного шлейфа. При наличии действующей системы версионного контроля вы, возможно, захотите просмотреть процессы, которые ваш коллектив использует для повседневной работы. Возможно, коллектив считает неплохой идеей переключиться на модель «запроса на включение внесенных изменений», в которой изменения публикуются для рассмотрения другими членами коллектива перед их фиксацией (коммитом). Или же вы и ваш коллектив чувствуете, что прямая фиксация в рамках модели «слияния изменений» более совместима с быстрыми малыми фиксациями задач параллелизма. В последней модели фиксации выполняются непосредственно в репозиторий без проверки. В нашем примере приложения симуляции пепельного шлейфа без контроля

версий приоритет состоит в размещении фрагментов кода на отведенном им месте, чтобы исключить неконтролируемое расхождение кода среди разработчиков.

Существует масса вариантов версионного контроля. Если у вас нет других предпочтений, то мы бы предложили Git, наиболее распространенную распределенную систему версионного контроля. *Распределенная система версионного контроля* – это система, которая предусматривает не единую централизованную систему, используемую в *централизованном версионном контроле*, а наличие нескольких репозиторных баз данных. Распределенный версионный контроль выгоден для проектов с открытым исходным кодом и там, где разработчики работают на ноутбуках, в удаленных местах или в других ситуациях, когда они не подсоединены к сети или не находятся близко к центральному хранилищу. В сегодняшней среде разработки он предлагает огромное преимущество. Но он сопряжен со стоимостью добавленной сложности. Централизованный версионный контроль по-прежнему популярен и более подходит для корпоративной среды, поскольку имеется только одно место, где существует вся информация об исходном коде. Централизованное управление также обеспечивает улучшенную безопасность и защиту проприетарного программного обеспечения.

Использованию Git посвящено много хороших книг, блогов и других ресурсов; мы перечислим несколько из них в конце главы. Мы также перечислим несколько разных распространенных систем версионного контроля в главе 17, в том числе бесплатные распределенные системы версионного контроля, такие как Mercurial и Git, коммерческие системы, такие как PerForce и ClearCase, а для централизованного версионного контроля – CVS и SVN. Независимо от того, какую систему вы используете, вы и ваш коллектив должны часто делать фиксации. Следующий ниже сценарий особенно часто встречается в задачах параллелизма.

- Я зафиксирую это после того, как добавлю следующее небольшое изменение…
- Еще одно… И вот неожиданно код перестал работать.
- Сейчас уже слишком поздно что-либо предпринимать!

С нами это случается слишком часто. Поэтому мы стараемся избегать этой проблемы, регулярно делая фиксации.

> **ДЛЯ СПРАВКИ** Если вам не требуется много малых фиксаций в главном репозитории, то в рамках некоторых систем версионного контроля, таких как Git, фиксации можно сворачивать либо поддерживать временную систему версионного контроля только для себя.

Сообщение о фиксации – это место, где автор фиксации может сообщать о том, какая задача решается и почему были внесены определенные изменения, как для себя, так и для нынешних или будущих членов коллектива. У каждого коллектива есть свои предпочтения в отношении подробности этих сообщений; в указанных сообщениях мы рекоменду-

ем использовать как можно больше деталей. Проявив усердие сегодня, вы получаете возможность уберечь себя от дальнейшей путаницы.

В общем случае сообщения о фиксации включают сводную информацию и тело сообщения. В сводке содержится краткое заявление, четко указывающее на то, какие новые изменения фиксация охватывает. Кроме того, если вы используете систему отслеживания вопросов, в сводной строке будет указан номер вопроса из этой системы. Наконец, тело содержит бо́льшую часть ответов на вопросы «почему?» и «как?», стоящих за фиксацией.

Примеры сообщений о фиксации

- Плохое сообщение о фиксации:

  ```
  Исправлен дефект
  ```

- Хорошее сообщение о фиксации:

  ```
  Устранено гоночное состояние в OpenMP-версии оператора размытия
  ```

- Отличное сообщение о фиксации:

  ```
  [Вопрос #21] устранено гоночное состояние в OpenMP-версии оператора
  размытия.
  * Гоночное состояние приводило к невоспроизводимым результатам среди
  компиляторов GCC, Intel и PGI. В целях исправления был введен OMP BARRIER,
  чтобы побудить потоки синхронизироваться непосредственно перед вычислением
  взвешенной стенсильной суммы.
  * Подтверждено, что код собирается и выполняется компиляторами GCC, Intel
  и PGI и дает согласованные результаты.
  ```

Первое сообщение на самом деле не помогает никому понять, какой дефект был исправлен. Второе сообщение помогает точно установить урегулирование проблем, связанных с гоночными состояниями в операторе размытия. Последнее сообщение ссылается на номер вопроса (#21) во внешней системе отслеживания вопросов и содержит краткое описание фиксации в первой строке. Тело фиксации, два пункта под сводкой, содержит более подробную информацию о том, что конкретно было необходимо и почему, и указывает другим разработчикам, что вы потратили время на тестирование своей версии перед ее фиксацией.

Имея план версионного контроля и по меньшей мере приблизительное соглашение по поводу процессов разработки в вашем коллективе, мы готовы перейти к следующему шагу.

2.1.2 Комплекты тестов: первый шаг к созданию устойчивого и надежного приложения

Тестовый комплект – это набор задач, которые части приложения выполняют с целью гарантированного обеспечения надлежащей работы родственных частей кода. Тестовые комплекты необходимы для всех, кроме самых простых программных кодов. При каждом изменении вы

должны проводить тестирование с целью подтверждения того, что получаемые вами результаты – одинаковы. Это звучит просто, но иногда код может достигать немного разных результатов с разными компиляторами и числами процессоров.

Пример: тест сценария с вулканом Кракатау для получения подтвержденных результатов

В вашем проекте есть приложение для симуляции океанских волн, которое генерирует подтвержденные результаты. *Подтвержденные результаты* – это симуляционные результаты, которые сравниваются с экспериментальными или реально-практическими данными. Симуляционный код, который был валидирован и подтвержден, является ценным. Вы не хотите потерять его при распараллеливании кода.

В нашем сценарии вы и ваш коллектив использовали два разных компилятора для разработки и производства. Первый – это компилятор C в коллекции компиляторов GNU (GCC), повсеместно используемый, свободно доступный компилятор, распространяемый со всеми дистрибутивами Linux и многими другими операционными системами. Компилятор C в просторечии называется компилятором GCC. В вашем приложении также используется коммерчески доступный компилятор Intel C.

На следующем ниже рисунке показаны гипотетические результаты для подтвержденной тестовой задачи, которая предсказывает высоту волны и суммарную массу. Результаты на выходе немного различаются в зависимости от того, какой компилятор и число процессоров используется в симуляции.

Какие различия приемлемы между вычислениями разными компиляторами и с разным числом процессоров?

В этом примере существуют различия в двух сообщаемых программой показателях. Без дополнительной информации трудно определить, что конкретно является правильным, а какие варианты решения приемлемы. В общем случае различия в результатах на выходе из вашей программы могут быть обусловлены:

- изменениями в компиляторе либо версии компилятора;
- изменениями в оборудовании;
- оптимизацией компилятора либо малыми различиями между компиляторами, либо версиями компилятора;
- изменениями в порядке операций, в особенности из-за параллелизма кода.

В следующих далее разделах мы обсудим вопросы, почему могут возникать такие различия, как определять варианты, которые являются разумными, и как разрабатывать тесты, которые выявляют реальные дефекты, прежде чем они будут зафиксированы в вашем репозитории.

Понимание изменений в результатах из-за параллелизма

Процесс параллелизма по своей сути изменяет порядок операций, что слегка изменяет числовые результаты. Но ошибки в параллелизме тоже порождают небольшие различия. Это очень важно понимать при разработке параллельного кода, потому что нам нужно сравнивать с однопроцессорным прогоном, чтобы определять правильность параллельного кодирования. Мы обсудим подход к сокращению числовых ошибок, вследствие которого ошибки параллелизма станут очевиднее, в разделе 5.7 при обсуждении методики глобальных сумм.

Для нашего тестового комплекта нам понадобится инструмент, который сравнивает числовые поля с небольшим допуском на наличие разниц. В прошлом для этой цели разработчикам тестовых комплектов пришлось бы создавать инструмент, но в последние годы на рынке появилось несколько утилит для числовых разниц. Двумя такими инструментами являются:

- Numdiff по адресу https://www.nongnu.org/numdiff/;
- ndiff по адресу https://www.math.utah.edu/~beebe/software/ndiff/.

В качестве альтернативы если ваш код выводит свое состояние в файлы HDF5 либо NetCDF, то эти форматы поставляются с утилитами, которые позволяют сравнивать значения, хранящиеся в файлах, с варьирующимися допусками.

- HDF5® – это версия 5 программного обеспечения, изначально известного как Иерархический формат данных, теперь именуемого HDF. Оно находится в свободном доступе в Группе HDF (https://www.hdfgroup.org/) и является распространенным форматом, используемым для вывода крупных файлов данных.
- NetCDF, или Форма общих сетевых данных, – это альтернативный формат, используемый сообществом по климату и наукам о Земле. Текущие версии NetCDF построены поверх HDF5. Эти библиотеки и форматы данных можно найти на веб-сайте Программного центра Unidata (https://www.unidata.ucar.edu/software/netcdf/).

В обоих этих форматах файлов для скорости и эффективности используются *двоичные данные*. Двоичные данные представляют собой машинное представление данных. Для нас с вами этот формат выглядит просто как тарабарщина, но в HDF5 есть несколько полезных утилит, которые позволяют нам заглядывать вовнутрь. Утилита h5ls перечисляет объекты в файле, такие как имена всех массивов данных. Утилита h5dump сбрасывает данные в каждый объект или массив. И самое главное для наших нынешних целей, утилита h5diff сравнивает два HDF-файла и сообщает о разнице выше некоего числового допуска. HDF5 и NetCDF, наряду

с другими темами параллельного ввода-вывода (I/O), будут подробнее рассмотрены в главе 16.

Использование CMake и CTest для автоматического тестирования кода

В последние годы стало доступно много систем тестирования. Сюда входят CTest, тест Google, тест pFUnit и др. Более подробную информацию об этих инструментах можно найти в главе 17. А пока давайте рассмотрим систему, созданную с использованием CTest и ndiff.

CTest является компонентом системы CMake. CMake – это конфигурационная система, которая адаптирует сгенерированные файлы makefile к разным системам и компиляторам. Встраивание системы тестирования CTest в CMake сцепляет их в единую систему. За счет этого обеспечивается большое удобство для разработчика. Процесс имплементирования тестов с использованием CTest относительно прост. Отдельные тесты записываются в виде любой последовательности команд. В целях их встраивания в систему CMake необходимо добавить следующее в файл CMakeLists.txt:

- `enable_testing();`
- `add_test(<имя_теста> <имя_исполняемого файла> <аргументы_исполняемого_файла>).`

Затем можно вызывать тесты с помощью `make test`, `ctest` либо можно выбирать отдельные тесты с помощью `ctest -R mpi`, где `mpi` – это регулярное выражение, которое запускает любые совпадающие имена тестов. Давайте просто пройдемся по примеру создания теста с использованием системы CTest.

Пример: пререквизиты CTest

Для выполнения этого примера вам понадобятся установленные MPI, CMake и ndiff. Для MPI (Интерфейс передачи сообщений) мы будем использовать OpenMPI 4.0.0 и CMake 3.13.3 (включает CTest) в Mac с более старыми версиями Ubuntu. Мы будем использовать компилятор GCC версии 8, установленный в Mac, а не тот, который используется по умолчанию. Затем OpenMPI, CMake и GCC (коллекция компиляторов GNU) инсталлируются менеджером пакетов. Мы будем использовать Homebrew в Mac и Apt, а также Synaptic в Ubuntu Linux. Обязательно получите заголовки разработки из libopenmpi-dev, если они отделены от времени выполнения. ndiff устанавливается вручную путем скачивания указанного инструмента с https://www.math.utah.edu/~beebe/software/ndiff/ и выполнения команд `./configure`, `make` и `make install`.

Создайте два исходных файла, как показано в листинге 2.1, создав приложения для этой простой системы тестирования. Мы будем использовать таймер для получения малых разниц в выходных данных как последовательной, так и параллельной программы. Обратите внимание, что исходный код этой главы можно найти по адресу https://github.com/EssentialsofParallelComputing/Chapter2.

Листинг 2.1 Простые программы хронометража для демонстрации системы тестирования

Программа C, TimeIt.c

```
 1 #include <unistd.h>
 2 #include <stdio.h>
 3 #include <time.h>
 4 int main(int argc, char *argv[]){
 5 struct timespec tstart, tstop, tresult;
 6 clock_gettime(CLOCK_MONOTONIC, &tstart);
 7 sleep(10);
 8 clock_gettime(CLOCK_MONOTONIC, &tstop);
 9 tresult.tv_sec =
       tstop.tv_sec - tstart.tv_sec;
10 tresult.tv_usec =
       tstop.tv_nsec - tstart.tv_nsec;
11 printf("Истекшее время равно %f secs\n",
       (double)tresult.tv_sec +
12     (double)tresult.tv_nsec*1.0e-9);
13 }
```

Запускает таймер, вызывает sleep, затем останавливает таймер

Таймер имеет два значения для разрешающей способности и для предотвращения переполнений

Печатает вычисленное время

Программа MPI, MPITimeIt.c

```
 1 #include <unistd.h>
 2 #include <stdio.h>
 3 #include <mpi.h>
 4 int main(int argc, char *argv[]){
 5    int mype;
 6    MPI_Init(&argc, &argv);
 7    MPI_Comm_rank(MPI_COMM_WORLD, &mype);
 8    double t1, t2;;
 9    t1 = MPI_Wtime();
10    sleep(10);
11    t2 = MPI_Wtime();
12    if (mype == 0)
         printf( "Истекшее время равно %f secs\n",
                  t2 - t1);
13    MPI_Finalize();
14 }
```

Инициализирует MPI и получает ранг процессора

Запускает таймер, вызывает sleep, затем останавливает таймер

Печатает хронометраж на выходе из первого процессора

Выключает MPI

Теперь вам нужен тестовый скрипт, который выполняет приложения и создает несколько разных выходных файлов. После этого прогона должны иметься числовые сравнения выходных данных. Ниже приведен пример процесса, который вы можете поместить в файл с именем mympiapp.ctest. Следует выполнить `chmod +x`, чтобы сделать его исполняемым.

mympiapp.ctest

```
 1 #!/bin/sh
 2 ./TimeIt > run0.out
 3 mpirun -n 1 ./MPITimeIt > run1.out
 4 mpirun -n 2 ./MPITimeIt > run2.out

 5 ndiff --relative-error 1.0e-4 run1.out run2.out
```

Выполняет последовательный тест

Выполняет первый MPI-тест на одном процессоре

Выполняет второй MPI-тест на двух процессорах

Сравнивает результат двух MPI-заданий чтобы получить отказ теста

```
6 test1=$?

7 ndiff --relative-error 1.0e-4 run0.out run2.out
8 test2=$?

9 exit "$(($test1+$test2))"
```

Захватывает статус, установленный командой ndiff

Сравнивает последовательный результат с двухпроцессорным прогоном

Выходит с кумулятивным статусным кодом, чтобы CTest мог сообщить о прохождении либо отказе

Этот тест сначала сравнивает результаты на выходе из параллельного задания с одним и двумя процессорами с допуском 0.1 % в строке 5. Затем он сравнивает последовательный прогон с параллельным двухпроцессорным заданием в строке 7. В целях отказа тестов попробуйте уменьшить допуск до 1.0e–5. CTest использует код завершения в строке 9, чтобы сообщить о прохождении либо отказе. Простейший способ добавить набор CTest-файлов в тестовый комплект состоит в использовании цикла, который отыскивает все файлы, заканчивающиеся на .ctest, и добавляет их в CTest-список. Ниже приведен пример файла CMakeLists.txt с дополнительными командами по созданию двух приложений:

CMakeLists.txt

```
 1 cmake_minimum_required (VERSION 3.0)
 2 project (TimeIt)
 3
 4 enable_testing()
 5
 6 find_package(MPI)
 7
 8 add_executable(TimeIt TimeIt.c)
 9
10 add_executable(MPITimeIt MPITimeIt.c)
11 target_include_directories(MPITimeIt PUBLIC.
     ${MPI_INCLUDE_PATH})
12 target_link_libraries(MPITimeIt ${MPI_LIBRARIES})
13
14 file(GLOB TESTFILES RELATIVE
     "${CMAKE_CURRENT_SOURCE_DIR}" "*.ctest")
15 foreach(TESTFILE ${TESTFILES})
16    add_test(NAME ${TESTFILE} WORKING_DIRECTORY
         ${CMAKE_BINARY_DIR}
17    COMMAND sh
         ${CMAKE_CURRENT_SOURCE_DIR}/${TESTFILE})
18 endforeach()
19
20 add_custom_target(distclean
      COMMAND rm -rf CMakeCache.txt CMakeFiles
21    CTestTestfile.cmake Makefile Testing
         cmake_install.cmake)
```

Инициирует функциональность CTest в CMake

Встроенная процедура CMake для отыскания большинства пакетов MPI

Добавляет сборочные цели TimeIt и MPITimeIt с их файлами исходного кода

Требует путь подключения к файлу mpi.h и библиотеке MPI

Получает все файлы с расширением .ctest и добавляет их в список тестов для CTest

Конкретно-прикладная команда, distclean, удаляет созданные файлы

Команда find_package(MPI) в строке 6 определяет переменные MPI_FOUND, MPI_INCLUDE_PATH и MPI_LIBRARIES. Указанные переменные

включают язык в более новые CMake-версии литералов MPI_<lang>_INCLUDE_PATH и MPI_<lang>_LIBRARIES, обеспечивая разные пути для C, C++ и Fortran. Теперь остается лишь выполнить тест с помощью

```
mkdir build && cd build
cmake ..
make
make test
```

либо

```
ctest
```

Получить результат для отказавших тестов можно с помощью

```
ctest --output-on-failure
```

Вы должны получить несколько результатов, как показано ниже:

```
Running tests...
Test project /Users/brobey/Programs/RunDiff
    Start 1: mpitest.ctest
1/1 Test #1: mpitest.ctest ................... . Passed 30.24 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) = 30.24 sec
```

Этот тест основан на функции sleep и таймерах, поэтому он может пройти, а может и не пройти. Результаты тестирования находятся в папке Testing/Temporary/*.

В этом тесте мы сравнили результат между отдельными прогонами приложения. Также рекомендуется хранить файл золотого стандарта из одного из прогонов вместе с тестовым скриптом для сравнения. Это сравнение обнаруживает изменения, которые приведут к тому, что новая версия приложения получит результаты, отличные от предыдущих версий. Когда это произойдет, это поднимет красный флажок; проверьте, по-прежнему ли правильна новая версия. Если это так, то следует обновить золотой стандарт.

Ваш комплект тестов должен содержать как можно больше частей кода, насколько это практически возможно. Метрика *покрытия кода* квантифицирует приемлемость выполнения тестовым комплектом своей задачи, что выражается в процентах от строк исходного кода. У разработчиков тестов есть старая поговорка, что часть кода, в которой нет теста, нарушена, потому что, даже если он не нарушен сейчас, то он все-равно в конечном счете будет нарушен. При всех изменениях, вносимых при распараллеливании кода, нарушение кода просто неизбежно. Хотя высокое покрытие кода и имеет важность, в наших усилиях по параллелизму важнее, чтобы для частей кода, которые вы распараллеливаете, были тесты. Многие компиляторы умеют генерировать статистику покрытия

кода. Для GCC gcov является инструментом профилирования, а для Intel им является Codecov. Мы рассмотрим принцип его работы для GCC.

Покрытие кода с помощью GCC

1. Добавить флаги `-fprofile-arcs` и `-ftest-coverage` при компилировании и связывании.
2. Выполнить инструментированный исполняемый файл на серии тестов.
3. Выполнить `gcov <source.c>`, чтобы получить покрытие для каждого файла.

   ПРИМЕЧАНИЕ Для сборок с помощью CMake добавьте дополнительное расширение .c в имя файла исходного кода; например, gcov CMakeFiles/stream_triad.dir/stream_triad.c.c обрабатывает расширение, добавленное CMake.

4. Вы получите примерно такой результат:

```
88.89% of 9 source lines executed in file <source>.c
Creating <source>.c.gcov
```

Выходной файл gcov содержит листинг, начало каждой строки которого дополнено числом раз, когда он был исполнен.

Понимание разных видов тестов исходного кода

Существуют также разные виды систем тестирования. В этом разделе мы рассмотрим следующие типы:

- *регрессионные тесты* – выполняются через регулярные промежутки времени, чтобы удерживать код от откатывания назад. Обычно это делается каждую ночь или еженедельно с помощью планировщика заданий cron, который запускает задания в указанное время;
- *модульные тесты* – проверяет работу подпрограмм или других небольших частей кода во время разработки;
- *непрерывно-интеграционные тесты* – набирая популярность, эти тесты автоматически запускаются для выполнения фиксацией кода;
- *фиксационные тесты* – небольшой набор тестов, которые могут выполняться из командной строки за довольно короткое время и используются перед фиксацией.

Все эти типы тестирования важны для проекта, и вместо того, чтобы опираться только на один, их следует использовать вместе, как показано на рис. 2.3. Тестирование имеет особенную важность для параллельных приложений, поскольку обнаружение дефектов на ранних этапах цикла разработки означает, что вы не избежите отлаживания 1000 процессоров 6 часов напролет.

Модульные тесты лучше всего создавать по мере разработки кода. Истинные поклонники модульных тестов используют разработку на основе тестов (TDD), где сначала создаются тесты, а затем пишется код для их прохождения. Встраивание тестов такого типа в разработку параллель-

ного кода включает тестирование их работы на параллельном языке и тестирование имплементации. Выявление проблем на этом уровне гораздо проще урегулировать.

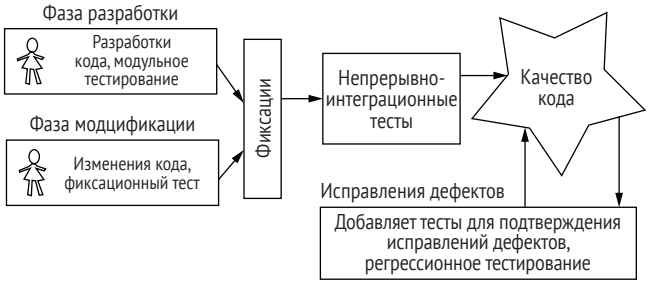


Рис. 2.3 Разные типы тестов охватывают разные части разработки кода, создавая высококачественный код, который всегда готов к выпуску

Фиксационные тесты – это первые тесты, которые следует добавлять в проект для интенсивного использования в фазе модификации кода. Эти тесты должны выполнять все процедуры в коде. Имея эти тесты в широком доступе, члены коллектива могут выполнять их *до* внесения фиксации в репозиторий. Мы рекомендуем разработчикам вызывать эти тесты из командной строки, например с помощью Bash или скрипта Python или файла makefile, перед фиксацией.

Пример: рабочий поток разработки с фиксационными тестами с использованием CMake и CTest

В целях выполнения фиксационного теста внутри CMakeLists.txt создайте три файла, показанные в следующем ниже листинге. Используйте Timeit.c из предыдущего теста, но поменяйте интервал сна с 10 на 30.

Создание фиксационного теста с помощью CTest

```
blur_short.ctest
 1 #!/bin/sh
 2 make

blur_long.ctest
 1 #!/bin/sh
 2 ./TimeIt

CMakeLists.txt
 1 cmake_minimum_required (VERSION 3.0)
 2 project (TimeIt)
 3
 4 enable_testing()
 5
 6 add_executable(TimeIt TimeIt.c)
```

Инициирует функциональность CTest в CMake

```
 7
 8 add_test(NAME blur_short_commit WORKING_DIRECTORY
         ${CMAKE_BINARY_DIRECTORY}
 9    COMMAND
   ${CMAKE_CURRENT_SOURCE_DIR}/blur_short.ctest)
10 add_test(NAME blur_long WORKING_DIRECTORY
         ${CMAKE_BINARY_DIRECTORY}
11    COMMAND
   ${CMAKE_CURRENT_SOURCE_DIR}/blur_long.ctest)
12
13 add_custom_target(commit_tests
      COMMAND ctest -R commit DEPENDS <myapp>)
14
15 add_custom_target(distclean
      COMMAND rm -rf CMakeCache.txt CMakeFiles
16      CTestTestfile.cmake Makefile Testing
        cmake_install.cmake)
```

Добавляет два теста, один с commit в имени

Конкретная цель commit_tests выполняет все тесты с commit в имени

Конкретно-прикладная команда, distclean, удаляет все созданные файлы

Фиксационные тесты могут выполняться командой `ctest -R commit` или с конкретной целью, добавленной в CMakeLists.txt с помощью `make commit_tests`. Команда `make test` или `ctest` запускает все тесты, включая длительный тест, который занимает некоторое время. Команда `commit test` выбирает тесты с *commit* в имени, чтобы получить набор тестов, который охватывает критическую функциональность, но выполняется немного быстрее. Теперь рабочий поток таков:

1. отредактировать исходный код: `vi mysource.c`;
2. собрать код: `make`;
3. выполнить фиксационные тесты: `make commit_tests`;
4. зафиксировать изменения в коде: `git commit`.

И повторить. Непрерывно-интеграционные тесты вызываются фиксацией в главном репозитории кода. Они являются дополнительной защитой от фиксирования неправильного кода. Указанные тесты могут быть такими же, как и фиксационные тесты, либо могут быть более обширными. Топовыми инструментами непрерывной интеграции для этих типов тестов являются:

- Jenkins (https://www.jenkins.io);
- Travis CI для GitHub и Bitbucket (https://travis-ci.com);
- GitLab CI (https://about.gitlab.com/stages-devops-lifecycle/continuousintegration/);
- CircleCI (https://circleci.com).

Регрессионные тесты обычно настраиваются для выполнения по ночам с помощью задания cron. Это означает, что тестовые комплекты могут быть более обширными, чем тестовые комплекты для других типов тестирования. Эти тесты могут быть более длительными, но должны завершаться к утреннему отчету. Дополнительные тесты, такие как проверка памяти и покрытие кода, часто выполняются как регрессионные тесты из-за более длительного времени выполнения и периодичности

отчетов. Результаты регрессионных тестов часто отслеживаются с течением времени, и «стена прохождений» рассматривается как показатель благополучия проекта.

Дополнительные требования к идеальной системе тестирования

Хотя описанная ранее система тестирования достаточна для большинства целей, есть еще кое-что, что бывает полезно для более крупных проектов HPC. Эти типы проектов HPC могут иметь обширные тестовые комплекты, а также могут требовать выполнения в пакетной системе для доступа к более крупным ресурсам.

Система коллаборативного тестирования (Collaborative Testing System, CTS) на https://sourceforge.net/projects/ctsproject/ представляет пример системы, которая была разработана для этих требований. В ней используется скрипт Perl для выполнения фиксированного набора тестовых серверов, обычно 10, выполняющих тесты параллельно с пакетной системой. По завершении каждого теста выполняется следующий. Это позволяет избегать одновременного наводнения системы заданиями. Система CTS также автоматически определяет пакетную систему и тип MPI и настраивает скрипты для каждой системы. Система отчетности использует задания cron с тестами, запущенными в начале ночного периода. Кросс-платформенный отчет запускается утром, а затем рассылается.

Пример: тестовый комплект сценария с вулканом Кракатау для проектов HPC

После просмотра ваших приложений вы обнаруживаете, что существует большая пользовательская база для приложения по обнаружению изображений. Поэтому ваш коллектив решает проводить обширные регрессионные тесты перед каждой фиксацией, чтобы избежать влияния на пользователей. Более длительные тесты на корректность памяти выполняются по ночам, а производительность отслеживается еженедельно. Однако приложение по симуляции океанских волн является новым, и у него меньше пользователей, но вы хотите убедиться, что подтвержденная задача продолжает давать одинаковый ответ. Фиксационный тест занимает слишком продолжительное время, поэтому вы выполняете сокращенную версию и полную версию еженедельно.

Для обоих приложений настраивается непрерывно-интеграционный тест для сборки кода и выполнения нескольких меньших тестов. Модель пепельного шлейфа только начала разрабатываться, поэтому вы решаете использовать модульные тесты для проверки каждого нового раздела кода по мере его добавления.

2.1.3 Отыскание и исправление проблем с памятью

Хорошее качество кода имеет первостепенное значение. Параллелизация часто приводит к появлению любого изъяна в коде; он может быть в виде неинициализированной памяти или перезаписей памяти.

- *Неинициализированная память* – это память, доступ к которой осуществляется до установки ее значений. Когда вы выделяете память под свою программу, она получает все значения, которые находятся в этих ячейках памяти. Если память используется до установки в ней значений, то это приводит к непредсказуемому поведению.
- *Перезапись памяти происходит, когда данные записываются в ячейку памяти, не принадлежащую переменной.* Примером тому является запись за пределы массива или строкового литерала.

В целях выявления такого рода проблем мы рекомендуем использовать инструменты обеспечения правильности памяти с целью тщательной проверки вашего кода. Одним из лучших из них является свободно доступная программа Valgrind. Valgrind – это инструментальная платформа, которая работает на уровне машинного кода, выполняя инструкции через синтетический CPU. Под эгидой Valgrind разработано много инструментов. Первым шагом является инсталляция Valgrind в вашей системе с помощью менеджера пакетов. Если вы используете последнюю версию macOS, то вы, возможно, обнаружите, что для переноса Valgrind на новое ядро потребуется несколько месяцев. Для этого лучше всего выполнять Valgrind на другом компьютере, более старом macOS или развернуть виртуальную машину или образ Docker.

Для выполнения Valgrind исполните свою программу как обычно, вставив спереди команду `valgrind`. Для заданий MPI команда `valgrind` помещается после `mpirun` и перед именем исполняемого файла. Valgrind лучше всего работает с компилятором GCC, потому что этот коллектив разработчиков принял его на вооружение, работая над устранением ложных срабатываний, которые загромождают результаты диагностики. При использовании компиляторов Intel рекомендуется компилировать без векторизации, чтобы избежать предупреждений о векторных командах. Вы также можете попробовать другие инструменты обеспечения правильности памяти, перечисленные в разделе 17.5.

Использование valgrind memcheck для отыскания проблем с памятью

Инструмент Memcheck используется по умолчанию в инструментальном комплекте Valgrind. Он перехватывает каждую команду и проверяет ее на наличие различных типов ошибок в памяти, генерируя диагностику в начале, во время и в конце выполнения. Это на порядок замедляет прогон. Если вы не использовали его раньше, то будьте готовы к большому количеству данных на выходе. Одна ошибка памяти приводит ко многим другим. Самая лучшая стратегия состоит в том, чтобы начинать с первой ошибки, исправлять ее и выполнять снова. Для того чтобы увидеть Valgrind в работе, попробуйте пример кода из листинга 2.2. В целях выполнения Valgrind вставьте команду `valgrind` перед именем исполняемого файла либо как

```
valgrind <./my_app>
```

либо как

```
mpirun -n 2 valgrind <./myapp>
```

Листинг 2.2 Пример кода для выявления ошибок памяти Valgrind

```
 1 #include <stdlib.h>
 2
 3 int main(int argc, char *argv[]){
 4    int ipos, ival;
 5    int *iarray = (int *) malloc(10*sizeof(int));
 6    if (argc == 2) ival = atoi(argv[1]);
 7    for (int i = 0; i<=10; i++){ iarray[i] = ipos; }
 8    for (int i = 0; i<=10; i++){
 9       if (ival == iarray[i]) ipos = i;
10    }
11 }
```

ipos не получила значение

Загружает неинициализированную память из ipos в iarray

Ставит флажок на неинициализированной памяти

Скомпилируйте этот код командой `gcc -g -o test test.c`, а затем выполните его командой `valgrind --leakcheck=full ./test 2`. Результат на выходе из Valgrind разбросан внутри результата на выходе из программы и может быть идентифицирован по приставке с двойными знаками равенства (==). Ниже показано несколько наиболее важных частей результата на выходе из этого примера:

```
==14324== Invalid write of size 4
==14324==    at 0x400590: main (test.c:7)
==14324==
==14324== Conditional jump or move depends on uninitialized value(s)
==14324==   at 0x4005BE: main (test.c:9)
==14324==
==14324== Invalid read of size 4
==14324==    at 0x4005B9: main (test.c:9)
==14324==
==14324== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==14324==    at 0x4C29C23: malloc (vg_replace_malloc.c:299)
==14324==    by 0x40054F: main (test.c:5)
```

Результат показывает отчеты о нескольких ошибках памяти. Самым сложным для понимания является отчет о неинициализированной памяти. Valgrind сообщает об ошибке в строке 9, когда было принято решение с неинициализированным значением. Ошибка на самом деле находится в строке 7, где `iarray` установлен равным `ipos`, которой не было передано значение. В более сложной программе для определения источника ошибки может потребоваться тщательный анализ.

2.1.4 Улучшение переносимости кода

Последнее требование к подготовке кода улучшает переносимость кода для более широкого спектра компиляторов и операционных систем. Переносимость начинается с базового языка HPC, обычно C, C++ или Fortran. Каждый из этих языков поддерживает стандарты для импле-

ментаций компиляторов, и периодически появляются новые выпуски стандартов. Но это не означает, что компиляторы легко их имплементируют. Нередко время задержки от выпуска до полной имплементации поставщиками компиляторов бывает длительным. Например, веб-сайт Polyhedron Solutions (http://mng.bz/yYne) сообщает, что ни один компилятор Linux Fortran полностью не имплементирует стандарт 2008 года, и менее половины полностью имплементируют стандарт 2003 года. Разумеется, куда важнее, чтобы компиляторы имплементировали те функциональности, которые вы хотите. Компиляторы C и C++, как правило, более современны в своих имплементациях новых стандартов, но задержка по-прежнему может вызывать проблемы для агрессивных коллективов разработчиков. Кроме того, даже если функциональности имплементированы, это не означает, что они работают в самых разных настроечных условиях.

Компилирование с использованием различных компиляторов помогает обнаруживать ошибки в кодировании или выявлять места, где код дает «преимущество» языковым интерпретациям. Переносимость обеспечивает гибкость при использовании инструментов, которые лучше всего работают в конкретной среде. Например, Valgrind лучше всего работает с GCC, но Intel® Inspector, инструмент обеспечения правильности потоков (threads), лучше всего работает при компиляции приложения с помощью компиляторов Intel. Переносимость также помогает при использовании параллельных языков. Например, CUDA Fortran доступен только с компилятором PGI. Текущий набор имплементаций директивоориентированных языков GPU OpenACC и OpenMP (с директивой `target`) доступен только для небольшого набора компиляторов. К счастью, MPI и OpenMP для CPU широко доступны для многих компиляторов и систем. В этом месте нам нужно четко указать о том, что существуют три отличимые возможности пакета OpenMP: 1) векторизация посредством директив SIMD, 2) потокообразование CPU из изначальной модели OpenMP и 3) выгрузка на ускоритель, обычно GPU, посредством новых директив `target`.

Пример: сценарий с вулканом Кракатау и переносимость кода

Ваше приложение по обнаружению изображений компилируется только компилятором GCC. Ваш проект обеспечения параллелизма добавляет потокообразование пакета OpenMP. Ваш коллектив решает компилировать его компилятором Intel, чтобы иметь возможность использовать Intel Inspector для отыскания условий гонки между потоками. Симуляция пеплового шлейфа написана на языке Fortran и предназначена для работы на GPU. Основываясь на ваших исследованиях современных языков GPU, вы решаете включить PGI в качестве одного из своих компиляторов разработки, чтобы иметь возможность использовать CUDA Fortran.

2.2 Профилирование: определение разрыва между способностями системы и производительностью приложения

Профилирование (рис. 2.4) определяет способности оборудования по производительности и сравнивает их с производительностью вашего приложения. Разница между этими способностями и текущей производительностью создает потенциал для повышения производительности.

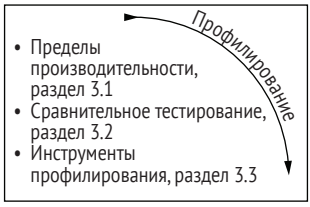


Рис. 2.4 Цель шага профилирования состоит в том, чтобы определить наиболее важные части кода приложения, на которые необходимо обратить внимание

Первая часть процесса профилирования заключается в определении мест, которые являются лимитирующим аспектом производительности вашего приложения. Мы подробно расскажем о возможных пределах производительности приложений в разделе 3.1. Если кратко, то большинство современных приложений лимитируется пропускной способностью памяти или пределом, который тщательно отслеживает пропускную способность памяти. Некоторые приложения могут лимитироваться имеющимися операциями с плавающей точкой (флопами). Мы представим способы расчета теоретических пределов производительности в разделе 3.2. Мы также опишем программы сравнительного тестирования, которые способны измерять достижимую производительность для этого аппаратного предела.

После того как вы поймете потенциальную производительность, сможете заняться профилированием своего приложения. Мы представим процесс использования нескольких инструментов профилирования в разделе 3.3. Разрыв между текущей производительностью вашего приложения и способностями оборудования для лимитирующего аспекта вашего приложения затем станет целью для улучшения в последующих шагах в параллелизме.

2.3 Планирование: основа успеха

Вооружившись информацией, собранной в вашем приложении и на целевых платформах, пришло время внести некоторые детали в план. На рис. 2.5 показаны части этого шага. С учетом усилий, которые требуются при параллелизме, разумно изучить предыдущую работу, прежде чем приступать к имплементационному шагу.

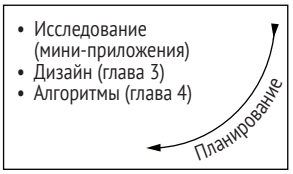


Рис. 2.5 Шаги планирования закладывают основу для успешного проекта

Вполне вероятно, что подобные задачи возникали и в прошлом. Вы найдете много исследовательских статей о проектах и технических приемах параллелизма, опубликованных в последние годы. Но один из самых богатых источников информации включает в себя выпущенные сравнительные тесты и мини-приложения. При наличии мини-приложений у вас есть не только исследования, но и фактический код для изучения.

2.3.1 Разведывательный анализ с использованием сравнительных тестов и мини-приложений

Сообщество высокопроизводительных вычислений разработало много сравнительных тестов, вычислительных ядер и примеров приложений для использования в сравнительном тестировании систем, экспериментах с производительностью и разработке алгоритмов. Мы перечислим некоторые из них в разделе 17.4. Вы можете использовать сравнительные тесты для оказания помощи в отборе наиболее подходящего оборудования для вашего приложения, а мини-приложения предоставляют помощь по наилучшим алгоритмам и методикам кодирования.

Сравнительные тесты (бенчмарки)[1] предназначены для высвечивания той или иной характеристики производительности оборудования. Теперь, когда у вас есть представление о пределе производительности вашего приложения, следует рассмотреть сравнительные тесты, наиболее применимые к вашей ситуации. Если вы выполняете вычисления на больших массивах, доступ к которым осуществляется линейно, то подходит сравнительный тест потоков данных. Если в качестве вычислительного ядра у вас итеративный матричный решатель, то лучше подойдет сравнительный тест на основе высокопроизводительного конъюгатного градиента (High Performance Conjugate Gradient, HPCG). *Мини-приложения* больше ориентированы на типичную операцию или шаблон, найденный в классе научных приложений.

Стоит обратить внимание на то, есть ли какое-либо подобие этих сравнительных тестов или мини-приложений с разрабатываемым вами параллельным приложением. Если подобие есть, то изучение вопроса о том, как они выполняют аналогичные операции, может сэкономить много усилий. Нередко с кодом проделывается большая работа для изучения способов достижения наилучшей производительности, для переноса на другие параллельные языки и платформы или для количественного оценивания характеристик производительности.

---

[1] То есть сравнительные тесты на основе эталонных показателей или стандартов. – *Прим. перев.*

В настоящее время сравнительные тесты и мини-приложения в основном относятся к области научных вычислений. Мы будем использовать некоторые из них в наших примерах, и вам рекомендуется использовать их для экспериментов и в качестве примера кода. Многие ключевые операции и параллельные имплементации продемонстрированы в этих примерах.

Пример: обновления призрачных ячеек

Многие приложения на основе вычислительной сетки распределяют свою сетку между процессорами в имплементации распределенной памяти (см. рис. 1.13). По этой причине указанным приложениям необходимо обновлять границы своей сетки значениями из смежного процессора. Эта операция называется *обновлением призрачных ячеек*. Ричард Барретт (Richard Barrett) из Sandia National Laboratories разработал мини-приложение MiniGhost, чтобы экспериментировать с разными подходами к выполнению данного типа операций. Мини-приложение MiniGhost является частью набора мини-приложений проекта Mantevo, доступных по адресу https://mantevo.org/default.php.

2.3.2 Дизайн стержневых структур данных и модульность кода

Дизайн структур данных оказывает долгосрочное влияние на ваше приложение. Это одно из решений, которое необходимо принимать с самого начала, понимая, что изменить дизайн позже станет трудно. В главе 4 мы рассмотрим некоторые важные соображения, а также тематическое исследование, демонстрирующее анализ производительности разных структур данных.

Для начала сосредоточьтесь на данных и движении данных. Это является доминирующим соображением при использовании современных аппаратных платформ. Это также приводит к эффективной параллельной имплементации, где осторожное перемещение данных становится еще более важным. Если мы возьмем файловую систему и сеть, то движение данных доминирует во всем.

2.3.3 Алгоритмы: редизайн для параллельности

В этом месте вы должны оценить алгоритмы своего приложения. Могут ли они быть модифицированы для параллельного кодирования? Существуют ли алгоритмы, которые обладают лучшей масштабируемостью? Например, в вашем приложении может быть раздел кода, который занимает всего 5 % времени выполнения, но имеет алгоритмическое масштабирование N^2, в то время как остальная часть кода масштабируется числом N, где N – это число ячеек или какой-либо другой компонент данных. По мере роста размера задачи 5 % вскоре станут 20 %, а затем еще выше. Вскоре это он будет доминировать время выполнения. В целях выявления такого рода проблемы вам может потребоваться составить про-

филь более крупной задачи, а затем посмотреть на рост не в абсолютных процентах, а во времени выполнения.

Пример: структура данных для модели пеплового шлейфа

Ваша модель пеплового шлейфа находится на ранних стадиях разработки. Предлагается несколько структур данных и разбивок на функциональные шаги. Ваш коллектив решает потратить неделю на анализ альтернатив, прежде чем они будут закреплены в коде, зная, что в будущем их будет трудно изменить. Одно из решений заключается в том, какую мультиматериальную структуру данных использовать, и, поскольку многие материалы будут находиться только в небольших областях сетки, есть ли хороший способ воспользоваться этим преимуществом. Вы решаете изучить разреженную структуру данных для хранения с целью экономии памяти (некоторые из них обсуждаются в разделе 4.3.2) и более быстрого кода.

Пример: отбор алгоритма для кода волновой симуляции

По проекту работа по параллелизму кода волновой симуляции добавит OpenMP и векторизацию. Вы слышали о разных имплементационных стилях для каждого из этих подходов к параллелизму. Вы поручаете двум членам коллектива выполнить обзор последних работ, для того чтобы получить информацию о подходах, которые работают лучше всего. Один из членов вашего коллектива выражает озабоченность по поводу параллелизма одной из более трудных процедур, которая имеет усложненный алгоритм. Текущая методика не выглядит простой для распараллеливания. Вы соглашаетесь и просите этого члена коллектива изучить альтернативные алгоритмы, которые, возможно, будут отличаться от того, что делается в настоящее время.

2.4 Имплементация: где все это происходит

Этот шаг я называю рукопашным боем. Внизу, в траншеях, строка за строкой, цикл за циклом и процедура за процедурой, код преобразовывается в параллельный. Именно здесь вступают в силу все ваши знания о параллельных имплементациях на CPU и GPU. Как показано на рис. 2.6, этот материал будет охвачен в большей части остальной книги. Главы, посвященные языкам параллельного программирования, главы 6–8 по CPU и главы 9–13 по GPU, начинают ваш путь к развитию этого опыта.

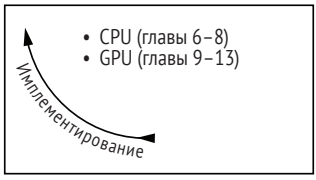


Рис. 2.6 На имплементационном шаге задействуются параллельные языки и навыки, развитые в остальной части книги

Во время имплементационного шага важно отслеживать ваши совокупные цели. В этом месте вы можете выбрать или не выбрать свой параллельный язык. Даже если вы его и выберите, то должны быть готовы пересматривать свой выбор по мере углубления в имплементацию. Несколько первоначальных соображений для выбора направления проекта таковы:

- достаточно ли скромны ваши требования к ускорению? Вам следует разведать тему параллелизма векторизации и совместной памяти (пакет OpenMP) в главах 6 и 7;
- требуется ли вам больше памяти для масштабирования? Если это так, то вам потребуется разведать тему параллелизма распределенной памяти в главе 8;
- требуются ли вам крупные ускорения? Тогда стоит обратиться к программированию GPU в главах 9–13.

Ключевым моментом на этом имплементационном шаге является разбиение работы на управляемые куски и распределение работы между членами вашего коллектива. Царит радостное возбуждение от того, что вы на порядок ускорили выполнение процедуры, и от осознания того, что совокупное воздействие невелико и предстоит еще много работы. Настойчивость и коллективная работа важны для достижения цели.

Пример: переоценка параллельного языка

Ваш проект по добавлению пакета OpenMP и векторизации в код волновой симуляции продвигается успешно. Вы получили ускорение на порядок для типичных вычислений. Но по мере того, как приложение ускоряется, ваши пользователи хотят выполнять более масштабные задачи, и им не хватает памяти. Ваш коллектив начинает подумывать о добавлении параллелизма MPI для доступа к дополнительным узлам, где имеется больше памяти.

2.5 Фиксация: качественное завершение работы

Фиксационный шаг завершает эту часть работы тщательными проверками, чтобы убедиться, что поддерживается качество кода и переносимость. На рис. 2.7 показаны компоненты этого шага. Обширность этих проверок в значительной степени зависит от характера приложения. Для производственных приложений с большим числом пользователей тесты должны быть гораздо более тщательными.

Рис. 2.7 Цель фиксационного шага состоит в создании прочной ступеньки на лестнице для достижения вашей конечной цели

ПРИМЕЧАНИЕ На данном шаге легче выявлять относительно мелкомасштабные проблемы, чем отлаживать осложнения шесть дней напролет на тысяче процессоров.

Коллектив должен принять и активно поддерживать фиксационный процесс и работать вместе, следуя ему. Предлагается провести совещание коллектива по разработке процедур, которым должны следовать все. Процессы, использованные при первоначальных усилиях по улучшению качества и переносимости кода, могут использоваться при создании ваших процедур. Наконец, фиксационный процесс должен периодически пересматриваться и адаптироваться к текущим потребностям проекта.

Пример: переоценивание процесса разработки кода вашим коллективом

Ваш коллектив разработчиков приложения волновой симуляции выполнил первое звено работы по добавлению пакета OpenMP в приложение. Но теперь приложение иногда аварийно отказывает без каких-либо объяснений. Один из членов вашего коллектива понимает, что это может быть связано с гоночными состояниями. Ваш коллектив имплементирует дополнительный шаг для проверки этих ситуаций в рамках фиксационного процесса.

2.6 Материалы для дальнейшего изучения

В этой главе мы лишь прикоснулись к тому, как подойти к новому проекту и что могут делать доступные инструменты. Для получения дополнительной информации проведите разведку ресурсов и попробуйте выполнить некоторые упражнения из следующих далее разделов.

2.6.1 Дополнительное чтение

Дополнительный опыт работы с современными распределенными инструментами версионного контроля принесет пользу вашему проекту. По меньшей мере один член вашего коллектива должен провести исследование многих ресурсов в Веб, в которых обсуждаются приемы использования выбранной вами системы версионного контроля. Если вы используете Git, то следующие ниже книги издательства Manning являются хорошими ресурсами:

- Майк Маккуэйд, «Git на практике» (Mike McQuaid, *Git in Practice*, Manning, 2014);
- Рик Умали, «Git за один месяц за ланчем» (Rick Umali, *Learn Git in a Month of Lunches*, Manning, 2015).

В рабочем потоке параллельной разработки тестирование является жизненно важным его компенентом. Модульное тестирование, пожалуй, является самым ценным, но и самым сложным в хорошей имплемента-

ции. У издательства Manning есть книга, в которой гораздо подробнее обсуждается модульное тестирование:

- Владимир Хориков, «Принципы, практика и шаблоны модульного тестирования» (Vladimir Khorikov, *Unit Testing Principles, Practices, and Patterns*, Manning, 2020).

Арифметика с плавающей точкой и прецизионность является недооцененной темой, несмотря на ее важность для каждого исследователя компьютерных вычислений. Ниже приведен хороший материал для чтения и обзор арифметики с плавающей точкой:

- Дэвид Голдберг, «Что каждый компьютерный исследователь должен знать об арифметике с плавающей точкой», (David Goldberg, «What every computer scientist should know about floating-point arithmetic», *ACM Computing Surveys (CSUR)* 23, № 1 (1991): 5–48).

2.6.2 Упражнения

1. У вас есть приложение с симуляцией высоты волн, которое вы разработали во время учебы в аспирантуре. Это последовательное приложение, и, поскольку оно планировалось только как основа для вашей диссертации, вы не встраивали в него никаких технических приемов разработки программного обеспечения. Теперь вы планируете использовать его в качестве отправной точки для имеющегося инструмента, который может использоваться многими исследователями. В вашем коллективе есть еще три разработчика. Что бы вы для них включили в свой план проекта?
2. Создайте тест с помощью CTest.
3. Исправьте ошибки памяти в листинге 2.2.
4. Выполните Valgrind на небольшом приложении по вашему выбору.

В этой главе мы подробно рассмотрели многие детали, необходимые для плана параллельного проекта. Оценивание способностей оборудования в ракурсе производительности и использование инструментов для извлечения информации о характеристиках оборудования и производительности приложений дают устойчивые, конкретные точки данных для наполнения плана. Правильное использование этих инструментов и умений поможет заложить прочную основу для успешного параллельного проекта.

Резюме

- Подготовка кода является важной частью работы по параллелизму. Каждый разработчик удивляется количеству усилий, затрачиваемых на подготовку кода для проекта. Но это время тратится не впустую в том смысле, что оно является основой для успешного проекта параллелизма.

- Вам следует улучшить качество кода для своего параллельного кода. Качество кода должно быть на порядок лучше, чем у типичного последовательного кода. Частично эта потребность в качестве связана с трудностями масштабной отладки, а частично – с изъянами, которые обнаруживаются в процессе обеспечения параллелизма или просто вследствие большого числа итераций, выполняемых каждой строкой кода. Возможно, это связано с тем, что вероятность обнаружения изъяна довольно мала, но, когда код выполняется тысячью процессорами, вероятность его возникновения возрастает в тысячу раз.
- Профилировочный шаг важен для выявления мест, в которых следует сосредоточить работу по оптимизации и параллелизму. В главе 3 содержится более подробная информация о том, как профилировать ваше приложение.
- Существует общий план проекта и еще один отдельный план для каждой итерации разработки. Оба этих плана должны содержать некоторые исследования, включающие мини-приложения, варианты дизайна структур данных и новые параллельные алгоритмы, чтобы заложить основу для следующих шагов.
- На фиксационном шаге нам необходимо разработать процессы для поддержания хорошего качества кода. Это должно быть постоянным усилием, а не откладываться на потом, когда код будет выпущен в производство или когда существующая пользовательская база начнет сталкиваться с проблемами в крупных и длительных симуляциях.

3 Пределы производительности и профилирование

Эта глава охватывает следующие ниже темы:

- понимание лимитирующего аспекта производительности приложений;
- оценивание производительности для лимитирующих аппаратных компонентов;
- измерение текущей производительности вашего приложения.

Ресурсы программистов ограничены. Вам нужно нацеливать эти ресурсы таким образом, чтобы они оказывали наибольшее влияние. Как это сделать, если вы не знаете характеристик производительности своего приложения и оборудования, на котором планируете работать? Именно об этом и пойдет речь в данной главе. Измеряя производительность вашего оборудования и вашего приложения, вы можете определять места, где можно эффективнее всего тратить время на разработку.

ПРИМЕЧАНИЕ Мы рекомендуем вам сверяться с упражнениями этой главы. С упражнениями можно ознакомиться по адресу https://github.com/EssentialsofParallelComputing/Chapter3.

3.1 Знание потенциальных пределов производительности вашего приложения

Исследователи компьютерных вычислений по-прежнему считают операции с плавающей запятой (флопы) главенствующим лимитирующим фактором производительности. Хотя это могло быть правдой много лет назад, реальность такова, что флопы редко лимитируют производительность в современных архитектурах. Но лимиты могут касаться как пропускной способности, так и задержки. *Пропускная способность* (или ширина полосы от англ. *bandwidth*) – это наилучшая скорость, с которой данные могут перемещаться по заданному пути в системе. Для того чтобы пропускная способность была пределом, в исходном коде должен использоваться потоковый (streaming) подход, когда память обычно должна быть сплошной, а все значения использоваться. Когда потоковый подход невозможен, задержка является более подходящим пределом. *Задержка* – это время, необходимое для передачи первого байта или слова данных. Ниже показано несколько возможных пределов производительности оборудования:

- флопы (операции с плавающей точкой);
- операции, включающие в себя все типы команд компьютера;
- пропускная способность памяти;
- задержка памяти;
- очередь команд (командный кеш);
- сети;
- диск.

Мы можем разбить все эти ограничения на две главные категории: скорости и подачи (feeds). *Скорости* – это быстрота, с которой могут выполняться операции. Сюда входят все виды компьютерных операций. Но, чтобы иметь возможность выполнять операции, вы должны иметь там данные. Вот тут-то и появляется подача данных. *Подачи* включают пропускную способность памяти в иерархии кеша, а также пропускную способность сети и диска. Для приложений, которые не могут иметь потоковое поведение, более важны задержки в подаче из памяти, сети и диска. Времена задержки бывают на порядки медленнее, чем для пропускной способности. Одним из важнейших факторов в том, чем именно контролируются приложения – пределами задержки либо величинами пропускной способности, – является качество программирования. Организация ваших данных таким образом, чтобы иметь возможность их использовать в потоковом режиме, может давать драматические ускорения.

Относительная производительность разных аппаратных компонентов показана на рис. 3.1. В качестве отправной точки давайте использовать 1 слово, загружаемое за цикл, и 1 флоп за цикл, отмеченные большой точкой. Большинство скалярных арифметических операций, таких как сложение, вычитание и умножение, можно выполнять за 1 цикл.

Операция деления может занимать больше времени, имея 3–5 циклов. В некоторых арифметических комбинациях возможно 2 флопа/цикл с командой слитного умножения-сложения. Число выполнимых арифметических операций еще больше увеличивается вместе с модулями векторной обработки и мультиядерными процессорами. Аппаратные достижения, в большинстве своем за счет параллелизма, значительно увеличивают показатель флопы/цикл.

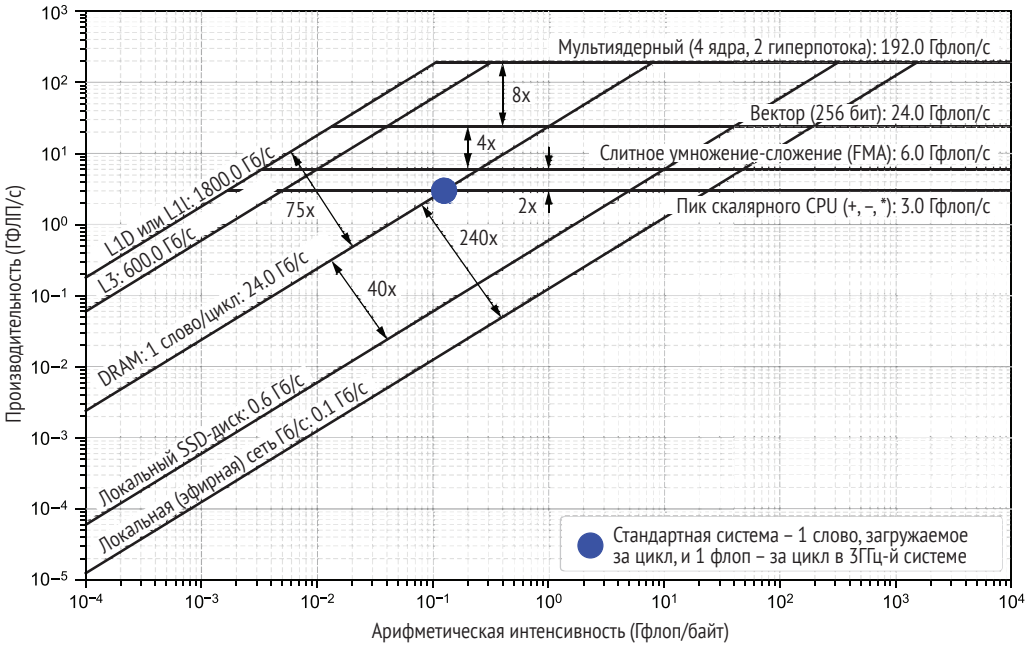


Рис. 3.1 Подачи и скорости, показанные на графике «контура крыши». Обычный скалярный процессор, обозначенный затененным кругом, близок к 1 слову, загружаемому за цикл, и 1 флопу за цикл. Множители для увеличения числа флопов обусловлены инструкцией слитного умножения-сложения, векторизацией, несколькими ядрами процессора и гиперпотоками. Также показаны относительные скорости перемещения из памяти/в память. Мы обсудим график «контура крыши» (roofline) подробнее в разделе 3.2.4

Глядя на наклонные лимиты памяти, мы видим, что увеличение производительности за счет более глубокой иерархии кешей означает, что доступ к памяти может соответствовать ускорению операций только в том случае, если данные содержатся в кеше L1, обычно около 32 КиБ. Но если бы у нас был только такой объем данных, то мы бы не так беспокоились о времени, которое это займет. В действительности же мы хотим оперировать на крупных объемах данных, которые могут содержаться только в основной памяти (DRAM) или даже на диске или в сети. Чистым результатом является то, что способности процессоров с плавающей точкой увеличивались намного быстрее, чем пропускная способность памяти. Это привело к появлению многочисленных *машинных балансов* порядка 50 флопов на каждое загружаемое 8-байтовое слово. В целях по-

нимания этого влияния на приложения мы измеряем его арифметическую интенсивность.

- *Арифметическая интенсивность* – в приложении измеряется число флопов, исполняемых в расчете на операции с памятью, где операции с памятью могут быть либо в байтах, либо в словах (слово составляет 8 байт для значения двойной точности и 4 байта для значения одинарной точности).
- *Машинный баланс* – для вычислительного оборудования показывает суммарное число исполняемых флопов, деленное на пропускную способность памяти.

Большинство приложений имеет арифметическую интенсивность, близкую к 1 флопу на загружаемое слово, но существуют приложения и с более высокой арифметической интенсивностью. В классическом примере приложения с высокой арифметической интенсивностью используется решатель плотных матриц для решения системы уравнений. Использование этих решателей в приложениях раньше было гораздо более распространено, чем сегодня. В сравнительном тесте Linpack используется вычислительное ядро из этой операции для представления указанного класса приложений. По данным Peise, арифметическая интенсивность в этом сравнительном тесте составляет 62.5 флопа/слово (см. ссылку в приложении A, Peise, 2017, стр. 201). Этого достаточно для большинства систем, чтобы максимально выжать функциональные возможности с плавающей точкой. Активное использование сравнительного теста Linpack для рейтинга 500 топовых вычислительных систем стало ведущей причиной разработки современных машин, ориентированных на высокий коэффициент загрузки «число флопов к объему памяти» (flop-to-memory load ratio).

Бывает, что для многих приложений затруднено даже достижение предела пропускной способности памяти. В целях понимания пропускной способности памяти необходимо некоторое понимание иерархии и архитектуры памяти. Несколько кешей между памятью и процессором помогают скрывать более медленную основную память (рис. 3.5 в разделе 3.2.3) в иерархии памяти. Данные передаются вверх по иерархии памяти в виде порций, именуемых *строками кеша*. Если доступ к памяти не осуществляется сплошным и предсказуемым образом, то полная пропускная способность памяти не достигается. Простой доступ к данным в столбцах для двухмерной структуры данных, хранящейся в порядке строк, приведет к увеличению объема памяти по длине строки структуры. Это может привести к тому, что из каждой строки кеша будет использоваться всего одно значение. Приблизительная оценка пропускной способности памяти по этому шаблону доступа к данным составляет 1/8 пропускной способности потока данных (один из каждых восьми используемых значений кеша). Это можно обобщить для других случаев, когда потребляется больше кеша, путем определения несплошной (noncontiguous) пропускной способности (B_{nc}) с точки зрения процента используемого кеша (U_{cache}) и эмпирической пропускной способности (B_E):

$B_{nc} = U_{cache} \times B_E$ = Средний процент используемого кеша × Эмпирическая пропускная способность.

Существуют и другие возможные пределы производительности. Кеш команд, возможно, не будет способен загружать команды достаточно быстро, чтобы процессорное ядро было занято. Целочисленные операции тоже являются более частым лимитирующим фактором, чем обычно допускается, в особенности в случае с массивами более высокой размерности, где вычисления индексов становятся более сложными.

Для приложений, требующих значительных сетевых или дисковых операций (таких как большие данные, распределенные вычисления или передача сообщений), пределы сетевого и дискового оборудования бывают наиболее серьезной проблемой. В целях получения представления о величине этих пределов производительности устройства, рассмотрим эмпирическое правило, согласно которому за время, необходимое для первого байта, переданного по высокопроизводительной компьютерной сети, вы можете выполнить более 1000 флопов на одном процессорном ядре. Стандартные механические дисковые системы работают на порядок медленнее для первого байта, что привело к очень асинхронной, буферизированной работе современных файловых систем и внедрению твердотельных запоминающих устройств.

Пример

Ваше приложение по обнаружению изображений должно обрабатывать много данных. Прямо сейчас изображение поступает по сети и сохраняется на диске для обработки. Ваш коллектив проверяет пределы производительности и решает попытаться исключить сохранение на диске как ненужную промежуточную операцию. Один из членов вашего коллектива предполагает, что вы можете выполнять дополнительные операции с плавающей точкой почти бесплатно, поэтому коллективу следует подумать о более сложном алгоритме. Но вы считаете, что лимитирующим фактором кода волновой симуляции является пропускная способность памяти. Вы добавляете задачу в проектный план, чтобы измерить производительность и подтвердить свою догадку.

3.2 Определение возможностей своего оборудования: сравнительное тестирование

После того как вы подготовите свое приложение и тестовые комплекты, вы сможете приступить к характеризации оборудования, которое нацелены эксплуатировать в производстве. Для этого вам необходимо разработать концептуальную модель оборудования, которая позволит вам понять его производительность. Производительность может быть охарактеризована рядом показателей:

- скоростью, с которой могут исполняться операции с плавающей точкой (флопов/с);
- скоростью, с которой данные могут перемещаться между разными уровнями памяти (Гб/с);
- скоростью, с которой ваше приложение потребляет энергию (Вт).

Концептуальные модели позволяют оценивать теоретическую пиковую производительность различных компонентов вычислительного оборудования. Показатели, с которыми вы работаете в этих моделях, и те, которые стремитесь оптимизировать, зависят от того, что вы и ваш коллектив цените в своем приложении. В целях дополнения этой концептуальной модели вы также можете провести эмпирические измерения на своем целевом оборудовании. Эмпирические измерения проводятся с помощью приложений сравнительного микротестирования. Одним из примеров приложения сравнительного микротестирования является приложение STREAM Benchmark[1], которое используется в случаях лимитированной пропускной способности.

3.2.1 Инструменты для сбора характеристик системы

При определении производительности оборудования мы используем смесь теоретических и эмпирических измерений. Несмотря на взаимодополняемость, теоретическое значение обеспечивает верхнюю границу производительности, а эмпирическое измерение подтверждает, что конкретно может быть достигнуто в упрощенном вычислительном ядре в условиях, близких к реальным.

Получить технические характеристики производительности оборудования на удивление сложно. Взрывной рост числа моделей процессоров и фокусировка внимания маркетинговых и медийных обзоров на широкой публике часто скрывают технические детали. Хорошие ресурсы для этого включают:

- для процессоров Intel, https://ark.intel.com;
- для процессоров AMD, https://www.amd.com/en/products/specifications/processors.

Одним из лучших инструментов для понимания используемого вами оборудования является программа lstopo. Она поставляется в комплекте с пакетом hwloc, который идет в составе почти каждого дистрибутива MPI. Команда `lstopo` выводит на экран графическое представление оборудования в вашей системе. На рис. 3.2 показан результат для ноутбука Mac. Результат может быть графическим либо текстовым. В целях получения изображения на рис. 3.2 в настоящее время требуется специальная инсталляция пакета hwloc и пакетов cairo для задействования интерфейса X11. Текстовая версия работает со стандартными инсталляциями

[1] Потоковые сравнительные тесты (stream benchmarks) касаются методов оценивания производительности и определяют соответствующие метрики для систем обработки потоковых данных. Они симулируют среду с разными рабочими нагрузками и анализируют поведение тестируемых систем. – *Прим. перев.*

менеджера пакетов. Версии hwloc для Linux и Unix обычно работают до тех пор, пока вы выводите на экран окно X11. Новая команда `netloc` добавлена в пакет hwloc для вывода на экран сетевых подключений.

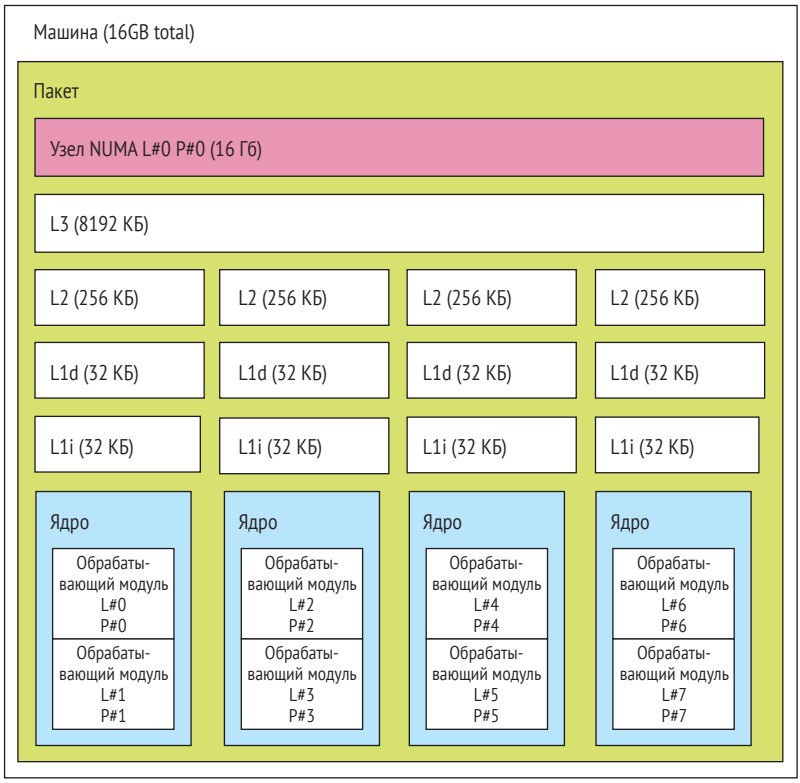


Рис. 3.2 Топология оборудования для ноутбука Mac с использованием команды lstopo

В целях инсталлирования cairo v1.16.0 следует:

1. скачать cairo с https://www.cairographics.org/releases/;
2. сконфигурировать его с помощью следующих ниже команд:

```
./configure --with-x --prefix=/usr/local
make
make install
```

В целях инсталлирования hwloc v2.1.0a1-git следует:

1. склонировать пакет hwloc из Git: https://github.com/open-mpi/hwloc.git;
2. сконфигурировать его с помощью следующих ниже команд:

```
./configure --prefix=/usr/local
make
make install
```

Несколько других команд для проверки деталей оборудования таковы: `lscpu` в системах Linux, `wmic` в Windows и `sysctl` или `system_profiler` в Mac. Команда Linux `lscpu` выводит сводный отчет с информацией из файла /proc/cpuinfo. Вы можете увидеть полную информацию по каждому логическому ядру, просмотрев /proc/cpuinfo напрямую. Информация из команды `lscpu` и файла /proc/cpuinfo помогает определять число процессоров, модель процессора, размеры кеша и тактовую частоту для системы. Флаги содержат важную информацию о наборе векторных команд для чипа. На рис. 3.3 мы видим, что доступны AVX2 и различные формы набора векторных команд SSE. Мы обсудим наборы векторных команд подробнее в главе 6.

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) per core:    1
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 94
Model name:            Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz
Stepping:              3
CPU MHz:               871.241
CPU max MHz:           3600.0000
CPU min MHz:           800.0000
BogoMIPS:              6384.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              6144K
NUMA node0 CPU(s):     0-3
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb
rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology
nonstop_tsc cpuid aperfmperf tsc_known_freq pni pclmulqdq dtes64 monitor ds_cpl
vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe
popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch
cpuid_fault epb invpcid_single pti ssbd ibrs ibpb stibp tpr_shadow vnmi flexpriority
ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm mpx
rdseed adx smap clflushopt intel_pt xsaveopt xsavec xgetbv1 xsaves dtherm ida
arat pln pts hwp hwp_notify hwp_act_window hwp_epp flush_l1d
```

Рис. 3.3 Результат на выходе из `lscpu` для настольного компьютера Linux, который показывает четырехъядерный процессор i5-6500 с частотой 3.2 ГГц с командами AVX2

Бывает полезно получить информацию об устройствах на шине PCI, в особенности для идентифицирования числа графических процессоров и их типа. Команда `lspci` сообщает обо всех устройствах (рис. 3.4). Из результата на рисунке мы видим, что имеется один GPU, и это NVIDIA GeForce GTX 960.

```
00:00.0 Host bridge: Intel Corporation Skylake Host Bridge/DRAM Registers (rev 07)
00:01.0 PCI bridge: Intel Corporation Skylake PCIe Controller (x16) (rev 07)
00:14.0 USB controller: Intel Corporation Sunrise Point-H USB 3.0 xHCI Controller (rev 31)
00:14.2 Signal processing controller: Intel Corporation Sunrise Point-H Thermal subsystem (rev 31)
00:16.0 Communication controller: Intel Corporation Sunrise Point-H CSME HECI #1 (rev 31)
00:17.0 SATA controller: Intel Corporation Sunrise Point-H SATA controller [AHCI mode] (rev 31)
00:1b.0 PCI bridge: Intel Corporation Sunrise Point-H PCI Root Port #19 (rev f1)
00:1c.0 PCI bridge: Intel Corporation Sunrise Point-H PCI Express Root Port #3 (rev f1)
00:1d.0 PCI bridge: Intel Corporation Sunrise Point-H PCI Express Root Port #9 (rev f1)
00:1f.0 ISA bridge: Intel Corporation Sunrise Point-H LPC Controller (rev 31)
00:1f.2 Memory controller: Intel Corporation Sunrise Point-H PMC (rev 31)
00:1f.3 Audio device: Intel Corporation Sunrise Point-H HD Audio (rev 31)
00:1f.4 SMBus: Intel Corporation Sunrise Point-H SMBus (rev 31)
00:1f.6 Ethernet controller: Intel Corporation Ethernet Connection (2) I219-V (rev 31)
01:00.0 VGA compatible controller: NVIDIA Corporation GM206 [GeForce GTX 960] (rev a1)
01:00.1 Audio device: NVIDIA Corporation Device 0fba (rev a1)
```

Рис. 3.4 Результат на выходе из команды `lspci` на настольном компьютере Linux, который показывает GPU NVIDIA GeForce GTX 960

3.2.2 Расчет теоретических максимальных флопов

Давайте пробежимся по цифрам для ноутбука MacBook Pro середины 2017 года с процессором Intel Core i7-7920HQ. Это четырехъядерный процессор, работающий на номинальной частоте 3.1 ГГц с гиперпотокообразованием. Благодаря функциональности турборазгона он может работать на частоте 3.7 ГГц при использовании четырех процессоров и на частоте до 4.1 ГГц при использовании одного процессора. Теоретические максимальные флопы (F_T) можно рассчитать по формуле:

$$F_T = C_v \times f_c \times I_c = \text{Виртуальные ядра} \times \text{Тактовая частота} \times \text{флопы/Цикл}.$$

Число ядер включает в себя эффекты гиперпотоков, из-за которых физические ядра (C_h) выглядят как большое число виртуальных или логических ядер (C_v). Здесь у нас есть два гиперпотока, из-за которых виртуальное число процессоров кажется равным восьми. Тактовая частота – это скорость турборазгона, когда задействованы все процессоры. Для данного процессора это 3.7 ГГц. Наконец, флопы в расчете на цикл, или, в более общем случае, команды в расчете на цикл (I_c), включают в себя число одновременных операций, которые могут исполняться модулем векторной обработки.

В целях определения числа исполняемых операций мы берем ширину вектора (VW) и делим на размер слова в битах (W_{bits}). Мы также включаем команду слитного умножения-сложения (fused multiply-add, FMA) в качестве еще одного фактора двух операций в расчете на цикл. Указанный фактор в уравнении называется слитными операциями (F_{ops}). Для этого конкретного процессора мы получаем:

$$I_c = VW/W_{bits} \times F_{ops} = (\text{256-битовый модуль векторной обработки/64 бита}) \times (\text{2 FMA}) = \text{8 флопов/Цикл};$$

$$C_v = C_h \times H_T = (\text{4 Аппаратных ядра} \times \text{2 Гиперпотока});$$

$$F_T = (\text{8 виртуальных ядер}) \times (\text{3.7 ГГц}) \times (\text{8 флопов/цикл}) = \text{236.8 Гфлопов/с}.$$

3.2.3 Иерархия памяти и теоретическая пропускная способность памяти

Для большинства больших вычислительных задач мы можем допустить, что существуют крупные массивы, которые необходимо загружать из основной памяти через иерархию кеша (рис. 3.5). Иерархия памяти с годами стала глубже с добавлением большего числа уровней кеша, чтобы компенсировать увеличение скорости обработки по сравнению с временами доступа к основной памяти.

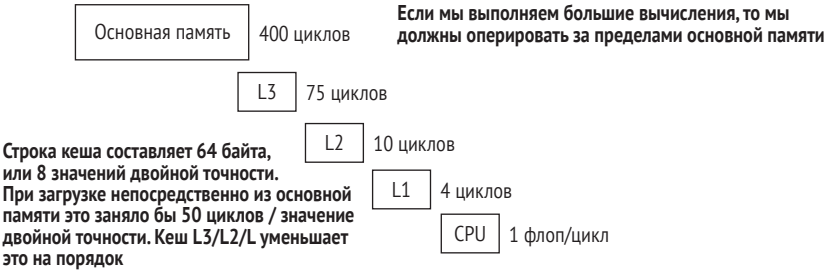


Рис. 3.5 Иерархия памяти и времена доступа. Память загружается в строки кеша и сохраняется на каждом уровне системы кеша для повторного использования

Мы можем рассчитать теоретическую пропускную способность основной памяти, используя спецификации микросхем памяти. Общая формула такова:

$$B_T = MTR \times M_c \times T_w \times N_s = \text{Скорость передачи данных} \times \text{Каналы памяти} \times \text{Байт в расчете на доступ} \times \text{Разъемы}.$$

Процессоры устанавливаются в разъем на материнской плате. *Материнская плата* – это главная системная плата компьютера, а *разъем* – это место, куда вставляется процессор. Большинство материнских плат имеет один разъем, где может быть установлен только один процессор. Материнские платы с двумя разъемами чаще встречаются в системах высокопроизводительных вычислений. Два процессора могут быть установлены на материнской плате с двумя разъемами, давая больше процессорных ядер и более высокую пропускную способность памяти.

Скорость передачи данных или памяти (memory transfer rate, MTR) обычно указывается в миллионах передач в секунду (MT/s). Память с удвоенной скоростью передачи данных (DDR) выполняет передачу данных в верхней и нижней частях цикла для двух транзакций за цикл. Это означает, что тактовая частота шины памяти составляет половину скорости передачи в МГц. Ширина передачи памяти (T_w) составляет 64 бита, и, поскольку имеется 8 бит/байт, передается 8 байт. В большинстве архитектур настольных компьютеров и ноутбуков имеется два канала памяти (M_c). Если вы установите память в обоих каналах памяти, то получите более высокую пропускную способность, но это означает, что вы не смо-

жете просто купить еще один модуль DRAM и его вставить. Вам придется заменить все модули на более крупные.

Для MacBook Pro 2017 года с памятью LPDDR3-2133 и двумя каналами теоретическая пропускная способность памяти (B_T) может быть рассчитана исходя из скорости передачи памяти (MTR) 2133 МТ/с, числа каналов (M_z) и числа разъемов на материнской плате:

$$B_T = 2133 \text{ МТ/с} \times 2 \text{ канала} \times 8 \text{ байт} \times 1 \text{ разъем} = 34\,128 \text{ МиБ/с},$$
$$\text{или } 34.1 \text{ ГиБ/с}.$$

Достижимая пропускная способность памяти находится ниже теоретической из-за влияния остальной иерархии памяти. Можно найти сложные теоретические модели для оценивания влияния иерархии памяти, но это выходит за рамки того, что мы хотим рассмотреть в нашей упрощенной модели процессора. Для этого обратимся к эмпирическим измерениям пропускной способности CPU.

3.2.4 Эмпирическое измерение пропускной способности и флопов

Эмпирическая пропускная способность (B_E) – это измерение самой быстрой скорости, с которой память может загружаться из основной памяти в процессор. Если запрашивается один байт памяти, то для его извлечения из регистра процессора требуется один цикл. Если его нет в регистре процессора, то он поступает из кеша L1. Если его нет в кеше L1, то кеш L1 загружает его из L2 и т. д. в основную память. Если он проходит весь путь до основной памяти, то для одного байта памяти может потребоваться около 400 циклов. Это время, которое требуется для первого байта данных из каждого уровня памяти, называется *задержкой памяти*. Как только значение будет находиться на более высоком уровне кеша, его можно получать быстрее до тех пор, пока оно не будет удалено из этого уровня кеша. Если всю память приходится загружать по байту за раз, то это будет крайне медленно. Поэтому, когда загружается байт памяти, одновременно загружается целая порция данных (именуемая *строкой кеша*). Если впоследствии будут доступны близлежащие значения, то они уже будут находиться на более высоких уровнях кеша.

Строки кеша, размеры кеша и число уровней кеша рассчитываются таким образом, чтобы обеспечивать как можно более высокую теоретическую пропускную способность основной памяти. Если мы загружаем сплошные данные как можно быстрее, чтобы пользоваться кешами наилучшим образом, то получаем максимально возможную скорость передачи данных процессора. Эта максимальная скорость передачи данных называется *пропускной способностью памяти*. В целях определения пропускной способности памяти мы можем измерить время чтения и записи крупного массива. Исходя из следующих эмпирических измерений, измеренная пропускная способность составляет около 22 ГиБ/с. Именно

эта измеренная пропускная способность и будет использоваться нами в простых моделях производительности в следующей главе.

Для измерения пропускной способности используются два разных метода: приложение STREAM Benchmark и модель «контура крыши», измеряемая с помощью эмпирического инструментария Roofline Toolkit. Приложение сравнительного тестирования STREAM Benchmark было создано Джоном Маккалпином (John McCalpin) примерно в 1995 году в поддержку его аргументации о том, что пропускная способность памяти гораздо важнее, чем пиковая способность с плавающей точкой. Для сравнения, модель контура крыши (см. рисунок на врезке под названием «Измерение пропускной способности с использованием эмпирического инструментария Roofline Toolkit» и обсуждение далее в этом разделе) объединяет лимит пропускной способности памяти и пиковую скорость во флопах в единый график с участками, которые показывают каждый лимит производительности. Инструментарий Empirical Roofline Toolkit был создан Национальной лабораторией Лоуренса Беркли для измерения и построения графиков модели контура крыши.

Приложение сравнительного тестирования *STREAM Benchmark* измеряет время чтения и записи крупного массива. Для этого существует четыре варианта в зависимости от операций, выполняемых процессором с данными во время их чтения: количественные измерения копирования, масштабирования, сложения и триады. Копирование не выполняет никакой работы с плавающей точкой, масштабирование и сложение выполняют одну арифметическую операцию, а триада выполняет две. Каждая из них дает несколько иную меру максимальной скорости, с которой можно ожидать загрузки данных из основной памяти, когда каждое значение данных используется только один раз. В этом режиме скорость флопов лимитирована скоростью загрузки памяти.

		Байты	Арифметические операции
Копирование:	a(i) = b(i)	16	0
Масштабирование:	a(i) = q*b(i)	16	1
Сумма:	a(i) = b(i) + c(i)	24	1
Триада:	a(i) = b(i) + q*c(i)	24	2

В следующем ниже упражнении показано применение приложения STREAM Benchmark для измерения пропускной способности на данном процессоре.

Упражнение: измерение пропускной способности с помощью приложения STREAM Benchmark

Джефф Хаммонд, ученый из Intel, разместил исходный код приложения сравнительного тестирования STREAM Benchmark Маккалпина в репозиторий Git для большего удобства. В данном примере мы используем его версию. В целях получения доступа к исходному коду следует:

1. склонировать образ, расположенный по адресу https://github.com/jeffhammond/STREAM.git;

2 отредактировать файл makefile и поменять строку компиляции на

```
-O3 -march=native -fstrict-aliasing -ftree-vectorize -fopenmp
      -DSTREAM_ARRAY_SIZE=80000000 -DNTIMES=20
make ./stream_c.exe
```

Вот результаты для ноутбука Mac 2017 года:

```
Function     Best Rate MB/s    Avg time    Min time    Max time
Copy:           22086.5        0.060570    0.057954    0.062090
Scale:          16156.6        0.081041    0.079225    0.082322
Add:            16646.0        0.116622    0.115343    0.117515
Triad:          16605.8        0.117036    0.115622    0.118004
```

Мы можем отобрать наилучшую пропускную способность из одного из четырех измерений в качестве эмпирического значения максимальной пропускной способности.

Если расчет может повторно использовать данные в кеше, то возможны гораздо более высокие скорости флопов. Если допустить, что все обрабатываемые данные находятся в регистре CPU или, возможно, в кеше L1, то максимальная скорость флопов определяется тактовой частотой CPU и числом флопов, которые он может выполнять за цикл. Это является теоретической максимальной скоростью флопов, рассчитанной в предыдущем примере.

Теперь мы можем соединить эти два элемента вместе, чтобы создать график модели контура крыши. Модель контура крыши имеет вертикальную ось флопов в секунду и горизонтальную ось арифметической интенсивности. Для высокой арифметической интенсивности, когда, по сравнению с загруженными данными, происходит много флопов, теоретическая максимальная скорость флопов является пределом. Это создает горизонтальную линию на графике с максимальной скоростью флопов. По мере уменьшения арифметической интенсивности время загрузки памяти начинает преобладать, и мы больше не сможем достигать максимальных теоретических флопов. Тогда это создает наклонную крышу в модели контура крыши, где достижимая скорость флопов снижается по мере снижения арифметической интенсивности. Горизонтальная линия на графике справа и наклонная линия слева создают характерный контур, напоминающий контур крыши и то, что стало называться моделью или графиком контура крыши. График контура крыши можно определить для CPU либо даже для GPU, как показано в следующем ниже упражнении.

Упражнение: измерение пропускной способности с использованием эмпирического инструментария Roofline Toolkit

В целях подготовки к этому упражнению инсталлируйте пакет OpenMPI или MPICH, чтобы получить рабочий MPI. Инсталлируйте gnuplot v4.2 и Python v3.0. На компьютерах Mac скачайте компилятор GCC, чтобы заменить компилятор, используемый по умолчанию. Эти инсталляции можно сделать с помощью менеджера пакетов (brew на Mac и apt или synaptic на Ubuntu Linux).

1. Склонировать инструментарий Roofline из Git:

   `git clone https://bitbucket.org/berkeleylab/cs-roofline-toolkit.git`

2. Затем набрать:

   ```
   cd cs-roofline-toolkit/Empirical_Roofline_Tool-1.1.0
   cp Config/config.madonna.lbl.gov.01 Config/MacLaptop2017
   ```

3. Отредактировать Config/MacLaptop2017. (На нижеследующем рисунке показан файл для ноутбука Mac 2017 года.)
4. Выполнить tests ./ert Config/MacLaptop2017.
5. Просмотреть файл Results.MacLaptop2017/Run.001/roofline.ps.

```
# Ноутбук Mac, MPI и OpenMP (четырехъядерный Intel Core I7 3.1 ГГц)

ERT_RESULTS Results.MacLaptop.01
ERT_DRIVER driver1
ERT_KERNEL kernel1

ERT_MPI    True
ERT_MPI_CFLAGS -I/usr/local/Cellar/open-mpi/4.0.0/include
ERT_MPI_LDFLAGS -L/usr/local/opt/libevent/lib -L/usr/local/Cellar/open-
mpi/4.0.0/lib -lmpi

ERT_OPENMP    True
ERT_OPENMP_CFLAGS -fopenmp
ERT_OPENMP_LDFLAGS -fopenmp

ERT_FLOPS 1,2,4,8,16
ERT_ALIGN 64

ERT_CC  gcc-8
ERT_CFLAGS -O3 -march=native -fstrict-aliasing -ftree-vectorize
ERT_LD  gcc-8
ERT_LDFLAGS
ERT_LDLIBS

ERT_RUN  export OMP_NUM_THREADS=ERT_OPENMP_THREADS; mpirun -np ERT_MPI_PROCS
ERT_CODE

ERT_PROCS_THREADS       1-8
ERT_MPI_PROCS           1,2,4
ERT_OPENMP_THREADS      1,2,4,8

ERT_NUM_EXPERIMENTS 3
ERT_MEMORY_MAX 1073741824
ERT_WORKING_SET_MIN 1
ERT_TRIALS_MIN 1

ERT_GNUPLOT gnuplot
```

mpicc --show помогает с этими путями

-march=native для того чтобы скомпилировать для модуля векторной обработки этого CPU

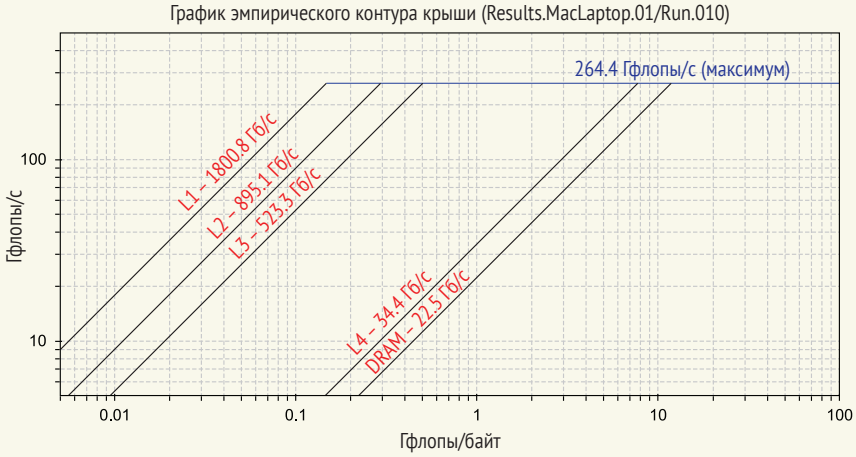
Устанавливает число рангов MPI и потоков. Помогает команда `lspci`. Core I7 имеет гиперпотоки, у I5 их нет

Config/MacLaptop2017

На нижеследующем рисунке показан контур крыши для ноутбука Mac 2017 года. Эмпирическое измерение максимальных флопов немного выше, чем

мы рассчитали аналитически. Вероятно, это связано с более высокой тактовой частотой в течение короткого периода времени. Опробывание разных параметров конфигурации, таких как отключение векторизации или выполнение одного процесса, помогает определить наличие у вас правильных спецификаций оборудования. Наклонные линии – это пределы пропускной способности в условиях разных арифметических интенсивностей. Поскольку они определяются эмпирически, метки для каждого наклона могут быть неправильными, и могут присутствовать дополнительные линии.

Из этих двух эмпирических измерений мы получаем аналогичную максимальную пропускную способность через иерархию кеша, равную примерно 22 Мб/с или примерно 65 % теоретической пропускной способности на чипах DRAM (22 ГиБ/с / 34.1 ГиБ/с).



Контур крыши для ноутбука Mac 2017 года показывает максимальные флопы в виде горизонтальной линии и максимальную пропускную способность разных уровней кеша и памяти – в виде наклонных линий

3.2.5 Расчет машинного баланса между флопами и пропускной способностью

Теперь мы можем определить машинный баланс. *Машинный баланс* – это флопы, деленные на пропускную способность памяти. Можно рассчитать как теоретический машинный баланс (MB_T), так и эмпирический машинный баланс (MB_E) примерно так:

$$MB_T = F_T / B_T = 236.8 \text{ Гфлопы/с} / 34.1 \text{ ГиБ/с} \times (8 \text{ байт/слово}) = 56 \text{ флопов/слово};$$

$$MB_E = F_E / B_E = 264.4 \text{ Гфлопы/с} / 22 \text{ ГиБ/с} \times (8 \text{ байт/слово}) = 96 \text{ флопов/слово}.$$

На рисунке контура крыши из предыдущего раздела машинный баланс представляет собой пересечение линии пропускной способности DRAM с горизонтальной линией предела флопа. Мы видим, что пересечение составляет чуть более 10 флопов/байт. Умножение на 8 даст машинный баланс выше 80 флопов/слово. Из этих разных методов мы получаем несколько разных оценок машинного баланса, но для большинства приложений вывод заключается в том, что мы находимся в режиме, ограниченном пропускной способностью.

3.3 Характеризация вашего приложения: профилирование

Теперь, когда у вас есть некоторое понимание того, какую производительность вы можете получить от оборудования, вам необходимо определить характеристики производительности вашего приложения. Кроме того, вы должны развить понимание того, каким образом разные подпрограммы и функции зависят друг от друга.

Пример: профилирование симуляции волн цунами от вулкана Кракатау

Вы решаете выполнить профилирование своего приложения волновой симуляции, чтобы увидеть, на что тратится время, и решить, как распараллелить и ускорить код. Бывает, что выполнение некоторых высокоточных симуляций занимает несколько дней, поэтому ваш коллектив хочет понять, как параллелизация с помощью OpenMP и векторизации смогут повысить производительность. Вы решаете исследовать аналогичное мини-приложение CloverLeaf, которое решает уравнения гидродинамики сжимаемой жидкости. Эти уравнения просто немного сложнее, чем те, которые используются в вашем приложении волновой симуляции. Приложение CloverLeaf имеет версии на нескольких параллельных языках. В рамках этого профилировочного исследования ваш коллектив хочет сравнить параллельную версию на основе OpenMP и векторизации с последовательной версией. Понимание производительности приложения CloverLeaf дает вам хорошую основу для второго шага профилирования вашего кода волновой симуляции.

3.3.1 Инструменты профилирования

Мы сосредоточимся на инструментах профилирования, которые продуцируют высокоуровневое представление, а также предоставляют дополнительную информацию или контекст. Существует много инструментов профилирования, но многие из них дают больше информации, чем можно усвоить. Если позволит время, то вы, возможно, захотите разведать другие инструменты профилирования, перечисленные в разделе 17.3.

Мы также представим набор свободно доступных инструментов и коммерческих инструментов, чтобы у вас были варианты в зависимости от имеющихся у вас ресурсов.

Важно помнить, что здесь ваша цель состоит в том, чтобы определить места, где лучше всего потратить свое время, занимаясь распараллеливанием приложения. Цель не в том, чтобы понять все до последней детали вашей текущей производительности. Легко совершить ошибку, вообще не используя эти инструменты либо заблудившись в инструментах и данных, которые они производят.

Использование графов вызовов для анализа горячих точек и зависимостей

Мы начнем с инструментов, которые высвечивают горячие точки и графически показывают связанность каждой подпрограммы в коде с другими подпрограммами. *Горячие точки* – это вычислительные ядра, которые занимают наибольшее количество времени во время исполнения. Кроме того, *граф вызовов* – это диаграмма, которая показывает вызовы процедурами других процедур. Мы можем объединить эти два набора информации, получив еще более мощную комбинацию, как мы увидим в следующем далее упражнении.

Графы вызовов могут генерироваться целым рядом инструментов, в том числе инструментом cachegrind комплекта инструмнентов Valgrind. Графы вызовов инструмента Cachegrind выделяют горячие точки и показывают зависимости подпрограмм. Этот тип графа полезен для планирования мероприятий по разработке для предотвращения конфликтов слияния. Общепринятая стратегия заключается в сегрегировании задач среди коллектива в таком ключе, чтобы работа, выполняемая каждым членом коллектива, делалась в одном стеке вызовов. В следующем ниже упражнении показано, как создавать граф вызовов с помощью комплекта инструментов Valgrind и инструмента Callgrind. Еще один инструмент в комплекте Valgrind, KCachegrind либо QCacheGrind, затем показывает результаты. Единственная разница состоит в том, что в одном из них используется графика X11, а в другом – графика Qt.

Еще одним полезным инструментом профилирования является Intel® Advisor. Это коммерческий инструмент с полезными функциональностями, служащими для получения максимальной производительности от вашего приложения. Инструмент Intel Advisor является частью пакета Parallel Studio, который также объединяет компиляторы Intel, Intel Inspector и VTune. Есть варианты с открытым исходным кодом для студента, преподавателя, разработчика и пробные лицензии, которые расположены по адресу https://software.intel.com/en-us/qualify-for-free-software/student. Эти инструменты Intel также были выпущены бесплатно в пакете oneAPI по адресу https://software.intel.com/en-us/oneapi. Недавно в Intel Advisor была добавлена функциональность профилирования, включающая модель контура крыши. Давайте взглянем на нее в действии в рамках инструмента Intel Advisor.

Упражнение: граф вызовов с использованием инструмента cachegrind

Первым шагом в данном упражнении является создание файла графа вызовов с использованием инструмента Callgrind, а затем его визуализирование с использованием KCachegrind.

1. Инсталлировать Valgrind и KCachegrind или QCacheGrind с помощью менеджера пакетов.
2. Скачать мини-приложение CloverLeaf с https://github.com/UK-MAC/CloverLeaf.

```
git clone --recursive https://github.com/UK-MAC/CloverLeaf.git
```

3. Собрать последовательную версию мини-приложения CloverLeaf.

```
cd CloverLeaf/CloverLeaf_Serial
make COMPILER=GNU IEEE=1 C_OPTIONS="-g -fno-tree-vectorize" \
              OPTIONS="-g -fno-tree-vectorize"
```

4. Выполнить Valgrind с инструментом Callgrind.

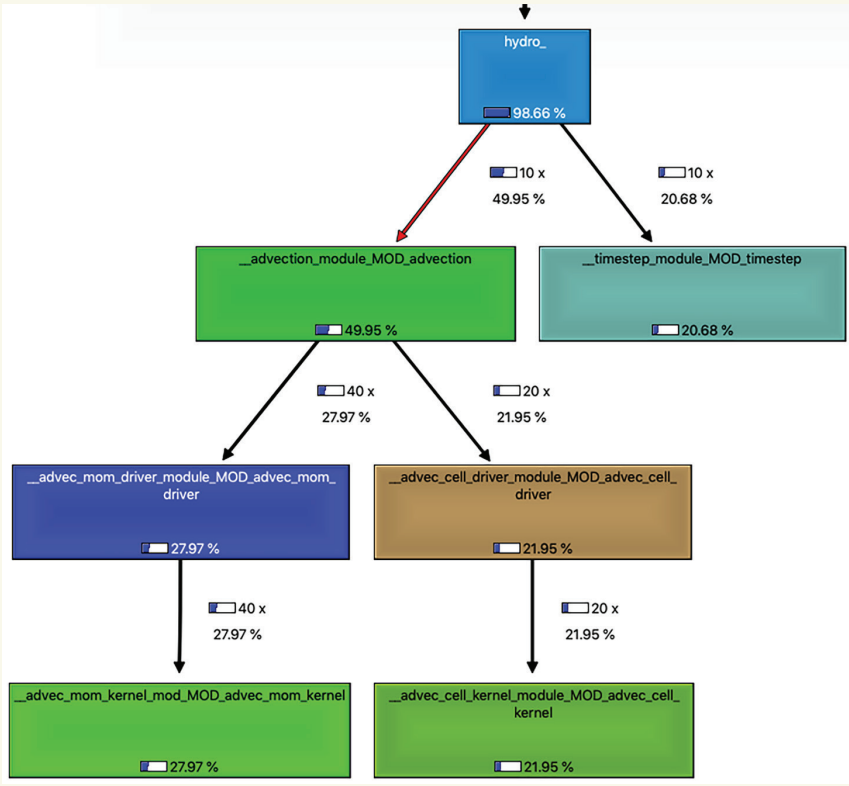
```
cp InputDecks/clover_bm256_short.in clover.in
edit clover.in and change cycles from 87 to 10
valgrind --tool=callgrind -v ./clover_leaf
```

5. Запустить QCacheGrind командой `qcachegrind`.
6. Загрузить конкретный файл callgrind.out.XXX в графический интерфейс QCacheGrind.
7. Щелкнуть правой кнопкой мыши на **Call Graph** (Граф вызовов) и изменить настройки изображения.

На нижеследующем рисунке показан граф вызовов мини-приложения CloverLeaf. В каждом прямоугольнике графа вызовов показано имя вычислительного ядра и процент времени, затрачиваемого таким ядром на каждом уровне стека вызовов. *Стек вызовов* – это цепочка процедур, которые вызывают текущее местоположение в коде. Когда каждая процедура вызывает подпрограмму, она помещает свой адрес в стек. В конце процедуры программа просто извлекает адрес из стека, возвращаясь к предыдущей процедуре вызова. У каждого «листа» дерева есть свои собственные стеки вызовов. Стек вызовов описывает иерархию источников данных для значений в «листовой» процедуре, в которой переменные передаются по цепочке вызовов. Хронометражи могут быть либо *исключающими*, когда каждая процедура исключает хронометраж вызываемых ею процедур, либо *включающими*, когда они включают хронометраж всех приведенных ниже процедур. Хронометражи, показанные на рисунке во врезке под названием «Измерение пропускной способности с использованием эмпирического инструментария Roofline Toolkit», включают в себя каждый уровень, содержащий уровни ниже и в сумме составляющий 100 % в главной процедуре.

На рисунке показана иерархия вызовов для наиболее дорогостоящих процедур, а также число вызовов и процент времени выполнения. Из этого мы видим, что подавляющая часть времени выполнения приходится на процеду-

ры адвекции, advection, которые перемещают материалы и энергию из одной ячейки в другую. Нам нужно сосредоточить наши усилия именно там. Граф вызовов также полезен для отслеживания пути следования по исходному коду.



Граф вызовов для мини-приложения CloverLeaf из KCachegrind показывает наибольший вклад во время выполнения

Упражнение: инструмент профилирования Intel® Advisor

В этом упражнении показано, как генерировать контур крыши для мини-приложения CloverLeaf, гидрокода по гидродинамике сжимаемой жидкости (compressible fluid dynamics, CFD) на основе регулярной решетки.

1 Собрать OpenMP-версию мини-приложения CloverLeaf:

```
git clone --recursive https://github.com/UK-MAC/CloverLeaf.git
cd CloverLeaf/CloverLeaf_OpenMP
make COMPILER=INTEL IEEE=1 C_OPTIONS="-g -xHost" OPTIONS="-g -xHost"
```

либо

```
make COMPILER=GNU IEEE=1 C_OPTIONS="-g -march=native" \
       OPTIONS="g -march=native"
```

2. Выполнить приложение в инструменте Intel Advisor:

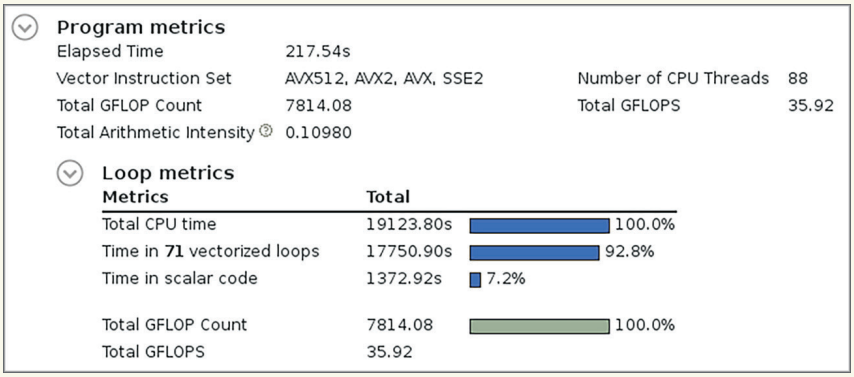
```
cp InputDecks/clover_bm256_short.in clover.in advixe-gui
```

3. Задать значение clover_leaf для исполняемого файла в каталоге CloverLeaf_OpenMP. Рабочий каталог может быть установлен равным каталогу приложений или CloverLeaf_OpenMP:
   a. для работы с GUI выбрать раскрывающееся меню «Start Survey Analysis» (Начать обзорный анализ) и выбрать «Start Roofline Analysis» (Начать анализ на основе контура крыши);
   b. в командной строке набрать следующее:

```
advixe-cl --collect roofline --project-dir ./advixe_proj -- ./clover_leaf
```

4. Запустить GUI и щелкнуть значок папки, чтобы загрузить данные прогона.
5. В целях просмотра результатов щелкнуть **Survey and Roofline** (Обзор и контур крыши), затем щелкнуть в дальней левой части верхней панели результатов производительности (где вертикальным текстом указан контур крыши).

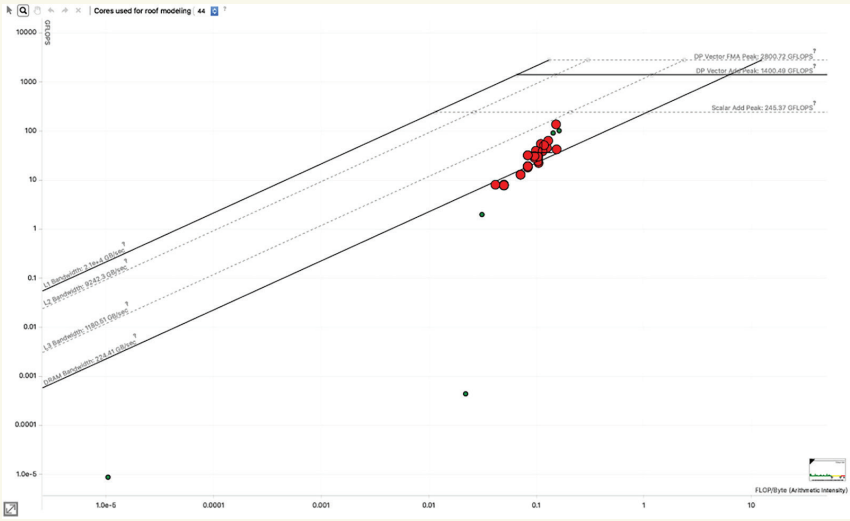
На следующем ниже рисунке показана сводная статистика профилировщика Intel Advisor. Он сообщает об арифметической интенсивности, приблизительно равной .11 флопов/байт или .88 флопов/слово. Скорость вычислений с плавающей точкой составляет 36 Гфлопов/с.



Сводный результат на выходе из Intel Advisor, сообщающий об арифметической интенсивности 0.11 флопов/байт

На следующем ниже рисунке показан график контура крыши из инструмента Intel Advisor для мини-приложения CloverLeaf. Производительность различных вычислительных ядер показана в виде точек относительно производительности контура крыши процессора Skylake. Размер и цвет точек показывают процент совокупного времени для каждого вычислительного ядра. Даже с первого взгляда ясно, что алгоритм лимитирован пропускной способностью и находится далеко слева от области, ограниченной вычислениями. Поскольку в этом мини-приложении используется двойная точность, умножив арифметическую интенсивность .01 на 8, мы получим арифметическую интенсивность значительно ниже 1 флопа/слово. Машинный баланс

представлен пересечением пика FMA двойной точности и пропускной способности DRAM.



Контур крыши, полученный из инструмента Intel® Advisor для мини-приложения CloverLeaf (clover_bm256_short.in) на процессоре Skylake Gold_6152

На этом графике машинный баланс превышает 10 флопов/байт или, умножив на 8, превышает 80 флопов/слово, где размер слова является значением двойной точности. Разделы кода, наиболее важные для производительности, определяются именами, связанными с каждой точкой. Процедуры, которые имеют наибольший потенциал для улучшения, можно определить по тому, насколько они ниже предела пропускной способности. Мы также видим, что было бы полезно улучшить арифметическую интенсивность в вычислительных ядрах.

В целях получения арифметической интенсивности мы также можем применить свободно доступный комплект инструментов likwid. *likwid* – это аббревиатура от Like I Knew What I'm Doing, что на русском примерно соответствует «будто я знаю, что делаю», автором которой являются Трейбиг, Хагер и Веллейн из Университета Эрлангена–Нюрнберга. Он представляет собой инструмент командной строки, который работает только в Linux и использует машиннно-специфичные регистры (MSR). Модуль MSR должен быть активирован командой `modprobe msr`. В указанном инструменте используются аппаратные счетчики для измерения и представления различной информации из системы, включая время выполнения, тактовую частоту, потребление энергии и мощности, а также статистику чтения из памяти и записи в память.

Упражнение: likwid perfctr

1 Инсталлировать likwid из менеджера пакетов либо с помощью следующих ниже команд:

```
git clone https://github.com/RRZE-HPC/likwid.git
cd likwid
edit config.mk
make
make install
```

2 Активировать MSR командой `sudo modprobe msr`.

3 Выполнить `likwid-perfctr -C 0-87 -g MEM_DP ./clover_leaf`.
(В выходных данных также есть столбцы min и max. Они были удалены для экономии места.)

```
+--------------------------------------+------------+-----------+
| Metric                               |     Sum    |    Avg    |
+--------------------------------------+------------+-----------+
| Runtime (RDTSC) [s] STAT             |  47646.0600 |  541.4325 |
| Runtime unhalted [s] STAT            |  56963.3936  | 647.3113 |
| Clock [MHz] STAT                     | 223750.6676 | 2542.6212 |
| CPI STAT                             |    170.1285 |    1.9333 |
| Energy [J] STAT                      | 151590.4909 | 1722.6192 |
| Power [W] STAT                       |    279.9804 |    3.1816 |
| Energy DRAM [J] STAT                 |  37986.9191 |  431.6695 |
| Power DRAM [W] STAT                  |     70.1601 |    0.7973 |
| DP MFLOP/s STAT                      |  22163.8134 |  251.8615 |
| AVX DP MFLOP/s STAT                  |   4777.5260 |   54.2901 |
| Packed MUOPS/s STAT                  |   1194.3827 |   13.5725 |
| Scalar MUOPS/s STAT                  |  17386.2877 |  197.5715 |
| Memory read bandwidth [MBytes/s] STAT  |  96817.7018 | 1100.2012 |
| Memory read data volume [GBytes] STAT  |  52420.2526 |  595.6847 |
| Memory write bandwidth [MBytes/s] STAT |  26502.2674 |  301.1621 |
| Memory write data volume [GBytes] STAT |  14349.1896 |  163.0590 |
| Memory bandwidth [MBytes/s] STAT       | 123319.9692 | 1401.3633 |
| Memory data volume [GBytes] STAT       |  66769.4422 |  758.7437 |
| Operational intensity STAT           |      0.3609 |    0.0041 |
+--------------------------------------+------------+-----------+
```

```
Computation Rate = (22163.8134+4*4777.5260) = 41274 MFLOPs/sec = 41.3
         GFLOPs/sec
Arithmetic Intensity = 41274/123319.9692 = .33 FLOPs/byte
Operational Intensity = .3608 FLOPs/byte
```

```
For a serial run:
Computation Rate = 2.97 GFLOPS/sec
Operational intensity = 0.2574 FLOPS/byte
Energy = 212747.7787 Joules
Energy DRAM = 49518.7395 Joules
```

Результат на выходе из likwid можно также использовать с целью расчета снижения энергопотребления для мини-приложения CloverLeaf вследствие выполнения в параллельном режиме.

Упражнение: рассчитайте энергосбережение для параллельного прогона по сравнению с последовательным

Снижение энергопотребления равно (212747.7787 – 151590.4909) / 212747.7787 = 28.7 %.

Снижение энергопотребления DRAM равно (49518.7395 – 37986.9191) / 49518.7395 = 23.2 %.

Инструментно-специфичные разделы кода с маркерами likwid-perfctr

Маркеры используются в likwid с целью повышения производительности для одного или нескольких разделов кода. Эта функциональная возможность будет использована в разделе 4.2 следующей главы.

1. Скомпилировать исходный код командой `-DLIKWID_PERFMON -I<PATH_TO_LIKWID>/include`.
2. Связать командой `-L<PATH_TO_LIKWID>/lib` и `-llikwid`.
3. Вставить строки из листинга 3.1 в свой исходный код.

Листинг 3.1 Вставка маркеров в исходный код для инструментно-специфичных разделов кода

```
LIKWID_MARKER_INIT;
LIKWID_MARKER_THREADINIT;
LIKWID_MARKER_REGISTER("Compute")

LIKWID_MARKER_START("Compute");
// ... Your code to measure
LIKWID_MARKER_STOP("Compute");
LIKWID_MARKER_CLOSE;
```

Требует наличия демона с (корневыми) полномочиями suid

Однопоточный участок

Генерирование собственных графиков контура крыши

Шарлин Янг (Charlene Yang), NERSC, создала и выпустила скрипт на Python для генерирования графика контура крыши. Это чрезвычайно удобно для создания высококачественной конкретно-прикладной графики с данными ваших исследований. Для этих примеров вы, возможно, захотите инсталлировать дистрибутив anaconda3. Он содержит библиотеку matplotlib и поддержку среды блокнотов Jupyter Notebook. Используйте следующий ниже код для индивидуальной настройки графика контура крыши с помощью Python и matplotlib:

```
git clone https://github.com/cyanguwa/nersc-roofline.git
cd nersc-roofline/Plotting
modify data.txt
python plot_roofline.py data.txt
```

Мы будем использовать модифицированные версии этого графопостроительного скрипта в нескольких упражнениях. В этом первом упражнении мы встроили части графопостроительного скрипта в блокноты Jupyter. Блокноты Jupyter (https://jupyter.org/install.html) позволяют вам чередовать документацию с кодом на языке Python с целью интерактивного взаимодействия. Мы используем его для динамического расчета теоретической производительности оборудования, а затем создаем график контура крыши для арифметической интенсивности и производительности.

Упражнение: графопостроительный скрипт, встроенный в блокнот Jupyter

Инсталлируйте Python3 с помощью менеджера пакетов. Затем используйте инсталлятор Python, pip, для инсталлирования пакетов NumPy, SciPy, matplotlib и Jupyter:

```
brew install python3
pip install numpy scipy matplotlib jupyter
```

Запустите блокнот Jupyter.

1. Скачать блокнот Jupyter с https://github.com/EssentialsofParallelComputing/Chapter3.
2. Открыть блокнот Jupyter HardwarePlatformCharacterization.ipynb.
3. В указанном блокноте изменить настройки оборудования в первом разделе для интересующей вас платформы, как показано на следующем ниже рисунке:

Hardware Platform Characterization

In [1]:
```
CPUDescription="Mid-2017 MacBook Pro laptop with an Intel Core i7-7920HQ processor"
MemoryDescription="LPDDR3-2133"
print (CPUDescription)
print (MemoryDescription)
```

```
Mid-2017 MacBook Pro laptop with an Intel Core i7-7920HQ processor
LPDDR3-2133
```

Processor Characteristics

Processor Frequency [GHz]
Processor Cores
Hyperthreads
Vector Width [bits]
Word Size in Bits [64 for double, 32 for single precision]
FMA [2 for Fused Multiple Add, 1 otherwise]

In [2]:
```
ProcessorFrequency=3.7
ProcessorCores=4
Hyperthreads=2
VectorWidth=256
WordSizeBits=64
FMA=2
```

Main Memory Characteristics

Data Transfer Rate [MT/s]
Bytes Transferred per Access [Bytes]
Number Channels

In [3]:
```
DataTransferRate=2133
MemoryChannels=2
BytesTransferredPerAccess=8
```

После того как вы измените настройки оборудования, вы будете готовы выполнить расчеты теоретических характеристик оборудования. Выполните все ячейки в блокноте и отыщите вычисления в следующей части блокнота, как показано на следующем ниже рисунке.

```
In [4]: TheoreticalMaximumFlops=ProcessorCores*Hyperthreads*ProcessorFrequency*VectorWidth/WordSizeBits*FMA
        print ("Theoretical Maximum Flops =",TheoreticalMaximumFlops, "GFLOPS/s")

        Theoretical Maximum Flops = 236.8 GFLOPS/s

In [5]: TheoreticalMemoryBandwidth=DataTransferRate*MemoryChannels*BytesTransferredPerAccess/1000
        print ("Theoretical Maximum Bandwidth (at main memory) =", TheoreticalMemoryBandwidth, "GiB/s")

        Theoretical Maximum Bandwidth (at main memory) = 34.128 GiB/s

In [6]: WordSizeBytes=WordSizeBits/8
        TheoreticalMachineBalance=TheoreticalMaximumFlops/TheoreticalMemoryBandwidth
        print ("Theoretical Machine Balance = ",TheoreticalMachineBalance, "Flops/byte")
        print ("Theoretical Machine Balance = ",TheoreticalMachineBalance*WordSizeBytes, "Flops/word")

        Theoretical Machine Balance =  6.938584153774028 Flops/byte
        Theoretical Machine Balance =  55.50867323019222 Flops/word
```

В следующем разделе блокнота содержатся измеренные данные производительности, которые вы хотите вывести на график контура крыши. Введите эти данные измерений производительности. Мы используем данные, собранные с помощью счетчиков производительности likwid для последовательного прогона мини-приложения CloverLeaf и прогона с OpenMP и векторизацией.

```
In [7]: smemroofs = [21000.0, 9961.16, 1171.55, 224.08]
        scomproofs = [2801.24, 1400.26]
        smem_roof_name = ["L1 Bandwidth", "L2 Bandwidth", "L3 Bandwidth", "DRAM Bandwidth"]
        scomp_roof_name = ["DP Vector FMA Peak", "DP Vector Add Peak"]
        AI = [.3608, .2106]
        FLOPS = [41.3, 2.735]
        labels = ["CloverLeaf w/OpenMP and Vectorization", "CloverLeaf Serial"]

        print ('memroofs', smemroofs)
        print ('mem_roof_names', smem_roof_name)
        print ('comproofs', scomproofs)
        print ('comp_roof_names', scomp_roof_name)
        print ('AI', AI)
        print ('FLOPS', FLOPS)
        print ('labels', labels)

        memroofs [21000.0, 9961.16, 1171.55, 224.08]
        mem_roof_names ['L1 Bandwidth', 'L2 Bandwidth', 'L3 Bandwidth', 'DRAM Bandwidth']
        comproofs [2801.24, 1400.26]
        comp_roof_names ['DP Vector FMA Peak', 'DP Vector Add Peak']
        AI [0.3608, 0.2106]
        FLOPS [41.3, 2.735]
        labels ['CloverLeaf w/OpenMP and Vectorization', 'CloverLeaf Serial']
```

Теперь блокнот запускает код для построения контура крыши с помощью matplotlib. Здесь показана первая половина графопостроительного скрипта. Вы можете изменить экстент, масштабы, метки и другие параметры графика.

```
In [8]: %matplotlib inline

        import numpy as np
        import matplotlib.pyplot as plt

        font = { 'size'   : 20}
        plt.rc('font', **font)

        markersize = 16
        colors = ['b','g','r','y','m','c']
        styles = ['o','s','v','^','D",">","<","*","h","H","+","1","2","3","4","8","p","d","|","_",".",","]

        fig = plt.figure(1,figsize=(20.67,12.6))
        plt.clf()
        ax = fig.gca()
        ax.set_xscale('log')
        ax.set_yscale('log')
        ax.set_xlabel('Arithmetic Intensity [FLOPs/Byte]')
        ax.set_ylabel('Performance [GFLOP/sec]')
        ax.grid()
        ax.grid(which='minor', linestyle=':', linewidth='0.5', color='black')

        nx = 10000
        xmin = -3
        xmax = 2
        ymin = 0.1
        ymax = 10000

        ax.set_xlim(10**xmin, 10**xmax)
        ax.set_ylim(ymin, ymax)

        ixx = int(nx*0.02)
        xlim = ax.get_xlim()
        ylim = ax.get_ylim()
```

Затем графопостроительный скрипт находит «локти», где линии пересекаются, чтобы построить только относительные сегменты. Он также определяет местоположение и ориентацию размещаемого на графике текста.

```
x = np.logspace(xmin,xmax,nx)
for roof in scomproofs:
    for ix in range(1,nx):
        if smemroofs[0] * x[ix] >= roof and smemroofs[0] * x[ix-1] < roof:
            scomp_x_elbow.append(x[ix-1])
            scomp_ix_elbow.append(ix-1)
            break


for roof in smemroofs:
    for ix in range(1,nx):
        if (scomproofs[0] <= roof * x[ix] and scomproofs[0] > roof * x[ix-1]):
            smem_x_elbow.append(x[ix-1])
            smem_ix_elbow.append(ix-1)
            break

for i in range(0,len(scomproofs)):
    y = np.ones(len(x)) * scomproofs[i]
    ax.plot(x[scomp_ix_elbow[i]:],y[scomp_ix_elbow[i]:],c='k',ls='-',lw='2')

for i in range(0,len(smemroofs)):
    y = x * smemroofs[i]
    ax.plot(x[:smem_ix_elbow[i]+1],y[:smem_ix_elbow[i]+1],c='k',ls='-',lw='2')


marker_handles = list()
for i in range(0,len(AI)):
  ax.plot(float(AI[i]),float(FLOPS[i]),c=colors[i],marker=styles[i],linestyle='None',ms=markersize,label=labels[i])
  marker_handles.append(ax.plot([],[],c=colors[i],marker=styles[i],linestyle='None',ms=markersize,label=labels[i])[0])

for roof in scomproofs:
    ax.text(x[-ixx],roof,
            scomp_roof_name[scomproofs.index(roof)] + ': ' + '{0:.1f}'.format(float(roof)) + ' GFLOP/s',
            horizontalalignment='right',
            verticalalignment='bottom')

for roof in smemroofs:
    ang = np.arctan(np.log10(xlim[1]/xlim[0]) / np.log10(ylim[1]/ylim[0])
                            * fig.get_size_inches()[1]/fig.get_size_inches()[0] )
    ax.text(x[ixx],x[ixx]*roof*(1+0.25*np.sin(ang)**2),
            smem_roof_name[smemroofs.index(roof)] + ': ' + '{0:.1f}'.format(float(roof)) + ' GiB/s',
            horizontalalignment='left',
            verticalalignment='bottom',
            rotation=180/np.pi*ang)


legl = plt.legend(handles = marker_handles,loc=4, ncol=2)
ax.add_artist(legl)

plt.savefig('roofline.png')
plt.savefig('roofline.eps')
plt.savefig('roofline.pdf')
plt.savefig('roofline.svg')

plt.show()
```

Построение графика этой арифметической интенсивности и скорости вычислений дает результат на рис. 3.6. И последовательный, и параллельный прогоны строятся в контуре крыши. Параллельный прогон примерно в 15 раз быстрее и с несколько более высокой операционной (арифметической) интенсивностью.

Есть еще пара инструментов, которые способны измерять арифметическую интенсивность. Пакет Intel® Software Development Emulator (SDE) (https://software.intel.com/en-us/articles/intel-software-development-emulator) генерирует массу информации, которую можно использовать для вычисления арифметической интенсивности. Инструмент определения производительности Intel® Vtune™ (часть пакета Parallel Studio) также можно использовать для сбора информации о производительности.

При сравнении результатов инструментов Intel Advisor и likwid наблюдается разница в арифметической интенсивности. Существует много разных способов подсчета операций, включая подсчет всей строки кеша при загрузке или только используемых данных. Аналогичным образом счетчики могут подсчитывать всю ширину вектора, а не только ту его

часть, которая используется. Некоторые инструменты учитывают только операции с плавающей точкой, в то время как другие также учитывают разные типы операций (например, целочисленные).

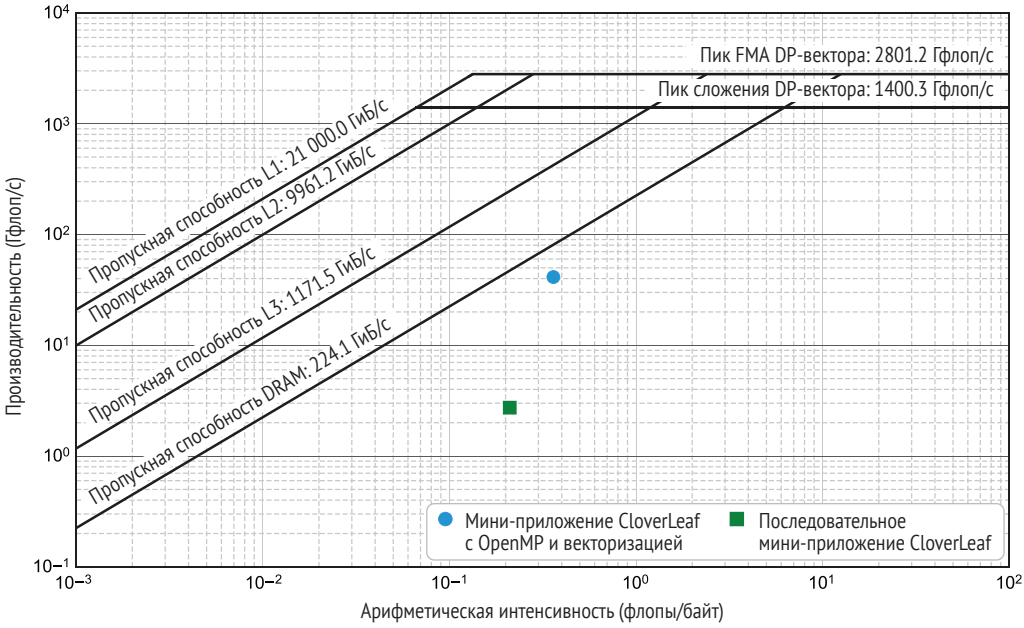


Рис. 3.6 Совокупная производительность мини-приложения Clover Leaf на процессоре Skylake Gold

3.3.2 Эмпирическое измерение тактовой частоты и энергопотребления процессора

Новейшие процессоры имеют много аппаратных счетчиков производительности и функциональных возможностей управления. К ним относятся частота процессора, температура, мощность и многие другие. Появляются новые программные приложения и библиотеки, облегчающие доступ к этой информации. Эти приложения смягчают трудности при программировании, но и помогают обходить необходимость в повышенных полномочиях с целью обеспечения большей доступности данных обычным пользователям. Такое развитие технологий можно только поприветствовать, потому что программисты не могут оптимизировать то, что они не могут видеть.

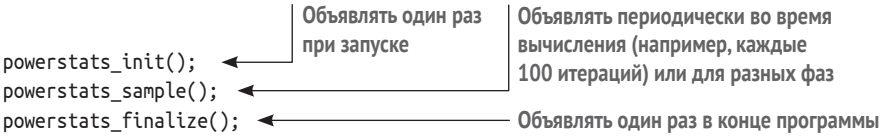
При агрессивном управлении частотой процессора процессоры редко работают на номинальной частоте. Тактовая частота уменьшается, когда процессоры находятся в режиме ожидания, и увеличивается до режима турборазгона, когда она заняты. Две простые интерактивные команды для просмотра поведения частоты процессора таковы:

```
watch -n 1 "lscpu | grep MHz"
watch -n 1 "grep MHz /proc/cpuinfo"
```

В комплекте инструментов likwid также есть инструмент командной строки likwid-powermeter для просмотра частоты процессора и статистики мощности. Инструмент likwid-perfctr тоже сообщает некоторые из этих статистических данных в сводном отчете. Еще одним удобным небольшим приложением является Intel® Power Gadget с версиями для Mac и Windows и более лимитированным для Linux. Он строит графики частоты, мощности, температуры и объем полезного использования.

Мини-приложение CLAMR (http://www.github.com/LANL/CLAMR.git) разрабатывает небольшую библиотеку PowerStats, которая будет отслеживать энергию и частоту внутри приложения и сообщать об этом в конце прогона. В настоящее время PowerStats работает на Mac, используя библиотечный интерфейс Intel Power Gadget. Аналогичная возможность разрабатывается для систем Linux. Код приложения должен добавлять всего несколько вызовов, как показано в следующем ниже листинге.

Листинг 3.2 Код PowerStats для отслеживания энергии и частоты



При запуске выводится следующая ниже таблица:

```
Processor          Energy(mWh) =   94.47181
IA                 Energy(mWh) =   70.07562
DRAM               Energy(mWh) =    3.09289
Processor          Power (W)   =   71.07833
IA                 Power (W)   =   54.73608
DRAM               Power (W)   =    2.32194
Average Frequency              = 3721.19422
Average Temperature (C)        =   94.78369
Time Expended (secs)           =   12.13246
```

3.3.3 Отслеживание памяти во время выполнения

Потребление памяти также является еще одним аспектом производительности, который нелегко увидеть программисту. Вы можете использовать ту же интерактивную команду для частоты процессора, что и в приведенном выше листинге, но вместо этого для статистики памяти. Сначала получите ИД процесса из команды `top` или `ps`. Затем примените одну из следующих ниже команд для отслеживания потребления памяти:

```
watch -n 1 "grep VmRSS /proc/<pid>/status"
watch -n 1 "ps <pid>"
top -s 1 -p <pid>
```

В целях интегрирования этого в вашу программу, возможно, чтобы увидеть, что происходит с памятью в разных фазах, библиотека MemSTATS в CLAMR предлагает четыре разных вызова функциональности отслеживания памяти:

```
long long memstats_memused()
long long memstats_mempeak()
long long memstats_memfree()
long long memstats_memtotal()
```

Вставьте эти вызовы в свою программу, чтобы вернуть текущую статистику памяти в точке вызова. MemSTATS – это единственный исходный и заголовочный файл на языке Си, поэтому его легко будет интегрировать в вашу программу. В целях получения исходного кода перейдите по ссылке http://github.com/LANL/CLAMR/ и загляните в каталог MemSTATS. Он также доступен по адресу https://github.com/EssentialsofParallelComputing/Chapter3 среди примеров исходного кода.

3.4 Материалы для дальнейшего изучения

Эта глава лишь поверхностно описывает то, что могут делать все эти инструменты. Дополнительную информацию можно получить, разведав следующие ниже ресурсы в разделе «Дополнительное чтение» и попробовав некоторые упражнения.

3.4.1 Дополнительное чтение

Более подробная информация и данные о приложении сравнительного тестирования STREAM Benchmark находится по адресу:

- Джон Маккалпин, 1995 год. «STREAM: устойчивая пропускная способность памяти в высокопроизводительных компьютерах» (John McCalpin. 1995. STREAM: Sustainable Memory Bandwidth in High Performance Computers), https://www.cs.virginia.edu/stream/.

Модель контура крыши была создана в Национальной лаборатории Лоуренса Беркли. На их веб-сайте есть много ресурсов с разведывательным анализом ее использования:

- «Модель производительности на основе контура крыши» (Roofline Performance Model), https://crd.lbl.gov/departments/computer-science/PAR/research/roofline/.

3.4.2 Упражнения

1. Рассчитайте теоретическую производительность системы по вашему выбору. Включите в расчет пиковые флопы, пропускную способность памяти и машинный баланс.

2. Скачайте инструментарий Roofline Toolkit с https://bitbucket.org/berkeleylab/cs-roofline-toolkit.git и измерьте фактическую производительность выбранной вами системы.
3. Используя инструментарий Roofline Toolkit, начните с одного процессора и постепенно добавляйте оптимизацию и параллелизацию, записывая величину улучшения на каждом шаге.
4. Скачайте приложение сравнительного тестирования STREAM Benchmark с https://www.cs.virginia.edu/stream/ и измерьте пропускную способность памяти выбранной вами системы.
5. Выберите один из общедоступных сравнительных тестов или мини-приложений, перечисленных в разделе 17.1, и сгенерируйте граф вызовов, используя KCachegrind.
6. Выберите один из общедоступных сравнительных тестов или мини-приложений, перечисленных в разделе 17.1, и измерьте его арифметическую интенсивность, используя инструменты Intel Advisor или likwid.
7. Используя представленные в этой главе средства расчета производительности, определите среднюю частоту и энергопотребление процессора для небольшого приложения.
8. Используя некоторые инструменты из раздела 3.3.3, определите объем памяти, потребляемый приложением.

В этой главе было охвачено много вопросов с многочисленными необходимыми подробностями в отношении плана параллельного проекта. Оценивание возможной производительности оборудования и использование инструментов для извлечения информации о характеристиках оборудования и производительности приложений дают устойчивые, конкретные точки данных для наполнения плана. Правильное использование этих инструментов и умений поможет заложить основу для успешного параллельного проекта.

Резюме

- Для приложения существует несколько возможных пределов производительности. Они варьируются от максимального числа операций с плавающей точкой (флопов) до пропускной способности памяти и операций чтения с жесткого диска и записи на жесткий диск.
- Приложения в современных вычислительных системах, как правило, более лимитированы пропускной способностью памяти, чем флопами. Хотя это правило было выявлено два десятилетия назад, оно стало еще более верным, чем прогнозировалось в то время. Но исследователи компьютерных вычислений были не особо быстры в том, чтобы адаптировать свое мышление к этой новой реальности.
- Вы можете использовать инструменты профилирования для измерения производительности вашего приложения и определения мест, где следует сосредоточить работу по оптимизации и параллелизации.

В этой главе приведены примеры использования инструментов Intel® Advisor, Valgrind, Callgrind и likwid, но существует масса других инструментов, включая Intel® VTune, Open|Speedshop (O|SS), HPC Toolkit или Allinea/ARM MAP. (Более полный список приведен в разделе 17.3.) Однако наиболее ценными инструментами являются те, которые предоставляют полезную информацию, а не ее количество.

- Вы можете использовать утилиты и приложения по измерению производительности оборудования с целью определения энергопотребления, частоты процессора, потребления памяти и многого другого. Делая эти атрибуты производительности заметнее, становится легче их оптимизировать с учетом указанных соображений.

4 Дизайн данных и модели производительности

Эта глава охватывает следующие ниже темы:

- почему реальные приложения с трудом достигают производительности;
- рассмотрение вычислительных ядер и циклов, которые значительно уступают по производительности;
- выбор структур данных для вашего приложения;
- оценивание разных подходов к программированию перед написанием кода;
- понимание принципа доставки данных иерархией кеша в процессор.

В этой главе тесно связаны две темы: (1) введение моделей производительности, в которых все больше доминирует движение данных, и с неизбежностью вытекающие из них (2) опорный дизайн и структура данных. Структура данных (и ее дизайн), хотя и может показаться второстепенной по отношению к производительности, имеет решающее значение. Она должна быть определена заранее, поскольку она определяет всю форму алгоритмов, код, и позже параллельную имплементацию.

Выбор структур данных и, следовательно, компоновка данных нередко обуславливают производительность, которую вы можете достичь,

и способы, которые не всегда очевидны при принятии решений по дизайну. Размышления о компоновке данных и ее влиянии на производительность лежат в основе нового и растущего подхода к программированию, именуемого *дизайном с ориентацией на данные*. Этот подход учитывает шаблоны применения данных в программе, и проактивно строится вокруг этого. Дизайн с ориентацией на данные дает нам ориентированное на данные представление о мире, что также согласуется с нашей сосредоточенностью на пропускной способности памяти, а не на операциях с плавающей точкой (флопах). Подводя итог, в целях повышения производительности наш подход заключается в том, чтобы думать о:

- данных, а не о коде;
- пропускной способности памяти, а не о флопах;
- строке кеша, а не об отдельных элементах данных;
- операциях с приоритетом на данных, уже находящихся в кеше.

Простые модели производительности, основанные на структурах данных и алгоритмах, которые следуют естественным образом, могут примерно предсказывать производительность. *Модель производительности* – это упрощенное представление того, как компьютерная система выполняет операции в ядре кода. Мы используем упрощенные модели, потому что рассуждать о полной сложности работы компьютера трудно, и такие рассуждения затеняют ключевые аспекты, о которых нам нужно думать с целью обеспечения производительности. Эти упрощенные модели должны отражать операционные аспекты компьютера, которые наиболее важны для производительности. Кроме того, каждая компьютерная система отличается деталями своей работы. Поскольку мы хотим, чтобы наше приложение работало на широком спектре систем, нам нужна модель, которая абстрагирует общее представление об операциях, общих для всех систем.

Модель помогает нам понять текущее состояние производительности нашего вычислительного ядра. Она помогает строить ожидания относительно производительности и того, как она может улучшаться с внесением изменений в код. Изменения в коде могут потребовать большой работы, и мы захотим знать, каким должен быть результат, прежде чем приступать к работе. Она также помогает нам сосредотачиваться на важнейших факторах и ресурсах, влияющих на производительность нашего приложения.

Модель производительности не ограничивается флопами, и действительно, мы сосредоточимся на аспектах данных и памяти. В дополнение к флопам и операциям с памятью бывают важны целочисленные операции, команды и типы команд, которые следует учитывать. Но пределы, связанные с этими дополнительными соображениями, обычно отслеживают производительность памяти и могут трактоваться как небольшое отклонение в производительности от этого предела.

В первой части главы рассматриваются простые структуры данных и их влияние на производительность. Далее мы представим модели про-

изводительности, которые будут использоваться для быстрого принятия решений по дизайну. Эти модели производительности затем применяются на деле в тематическом исследовании для изучения более усложненных структур данных для сжато-разреженных мультиматериальных массивов, чтобы оценить структуру данных, которая, вероятно, будет демонстрировать хорошую производительность. Влияние этих решений на структуры данных часто проявляется гораздо позже в проекте, когда вносить изменения гораздо сложнее. Последняя часть этой главы посвящена передовым моделям программирования; в ней представлены более сложные модели, которые подходят для более глубокого погружения в проблематику производительности или понимания того, как компьютерное оборудование и его дизайн влияют на производительность. Давайте разберемся поглубже, что это значит при рассмотрении вашего кода и вопросов производительности.

ПРИМЕЧАНИЕ Мы рекомендуем ознакомиться с примерами этой главы по адресу https://github.com/EssentialsofParallelComputing/Chapter4.

4.1 Структуры данных для обеспечения производительности: дизайн с ориентацией на данные

Наша цель состоит в разработке структур данных, которые приводят к хорошей производительности. Мы начнем с подхода к выделению многомерных массивов, а затем перейдем к более сложным структурам данных. В целях достижения этой цели требуется:

- понимание того, как данные компонуются в компьютере;
- как данные загружаются в строки кеша, а затем в CPU;
- как компоновка данных влияет на производительность;
- растущее значение перемещения данных для производительности современных компьютеров.

В большинстве современных языков программирования данные группируются в структуры того или иного рода. Например, использование структур данных в C или классов в объектно ориентированном программировании (так называемом ООП) сводит вместе связанные элементы для удобства организации исходного кода. Члены класса собираются вместе с методами, которые на них оперируют. Хотя философия объектно ориентированного программирования предлагает большую ценность, с точки зрения программиста, она полностью игнорирует принцип функционирования CPU. Объектно ориентированное программирование приводит к частым вызовам методов с несколькими строками кода в промежутке между ними (рис. 4.1).

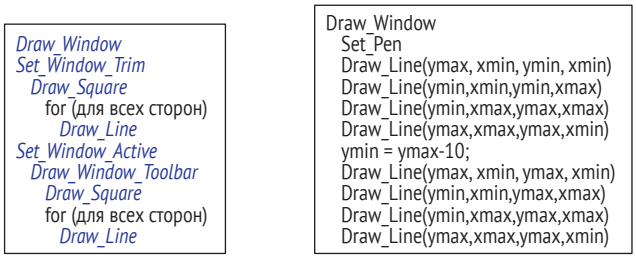


Рис. 4.1 Объектно ориентированные языки имеют глубокие стеки вызовов с большим числом вызовов методов (показано слева), в то время как процедурные языки имеют длинные последовательности операций на одном уровне стека вызовов

Для вызова метода в кеш сначала должен быть занесен класс. Далее в кеш заносятся данные, а затем смежные элементы класса. Это удобно, когда вы работаете с одним объектом. Но в приложениях с интенсивными вычислениями каждый элемент существует в больших количествах. В этих ситуациях мы не хотим вызывать метод на одном элементе по одному вызову за раз, требующему пересечения глубокого стека вызовов. Это приводит к неуспешным обращениям к кешу команд, слабому использованию кеша данных, ветвлению и большим накладным расходам на вызовы функций.

Методы C++ значительно облегчают написание сжатого кода, но почти каждая строка представляет собой вызов метода, как показано на рис. 4.1. В коде численной симуляции вызов `Draw_Line`, скорее всего, будет сложным математическим выражением. Но даже здесь если тело функции `Draw_Line` встраивается в исходный код, то переходы к функциям для кода C будут отсутствовать. *Встраивание* (inlining) – это ситуация, когда компилятор копирует исходный код тела подпрограммы в то место, где он используется, а не вызывает подпрограмму, как это происходит обычно. Однако компилятор может встраивать только простые, короткие процедуры. Но в объектно ориентированном коде есть вызовы методов, которые не будут встраиваться из-за сложности и глубоких стеков вызовов. Это приводит к неуспешным обращениям к кешу команд и другим проблемам с производительностью. Если мы рисуем только одно окно, то потеря производительности компенсируется более простым программированием. Если же мы собираемся нарисовать миллион окон, то мы не можем позволить себе снижение производительности.

Итак, давайте подойдем к этому с другой стороны и займемся дизайном наших структур данных с целью обеспечения производительности, а не для удобства программирования. Объектно ориентированное программирование и другие современные стили программирования, хотя и являются мощными, тем не менее создают массу ловушек для производительности. На CppCon в 2014 году презентация Майка Эктона «Дизайн с ориентацией на данные и C++» подвела итоги работы игровой индустрии, которая выявила причины, почему современные стили про-

граммирования снижают производительность. Сторонники стиля программирования с ориентацией на данные решают этот вопрос путем создания стиля программирования, который сосредотачивается непосредственно на производительности. Этот подход ввел в употребление термин «дизайн с ориентацией на данные» (data-oriented design), который фокусируется на наилучшем расположении данных для CPU и кеша. Указанный стиль имеет много общего с методами, давно используемыми разработчиками высокопроизводительных вычислений (HPC). В HPC дизайн с ориентацией на данные является нормой; он является естественным следствием из того, как писались программы на Fortran. Итак, каким же образом выглядит дизайн с ориентацией на данные? Он:

- оперирует на массивах, а не на отдельных элементах данных, избегая накладных расходов на вызовы и неуспешных обращений к кешу команд и данных;
- предпочитает массивы, а не структуры для более качественного использования кеша в более крупном числе ситуаций;
- встраивает подпрограммы, а не пересекает глубокую иерархию вызовов;
- контролирует выделение памяти, избегая ненаправленного повторного выделения за кулисами;
- использует сплошные связные списки на основе массивов, чтобы избежать имплементаций стандартных связных списков, используемых в C и C++, которые скачут по всей памяти из-за слабой локальности данных и использования кеша.

Переходя к параллелизации в последующих главах, мы отметим, что, как показывает наш опыт, крупные структуры данных или классы также становятся причинами проблем с параллелизацией и векторизацией совместной памяти. В программировании с совместной памятью мы должны иметь возможность помечать переменные как приватные для потока (thread) или как глобальные для всех потоков. Но в настоящее время все элементы в структуре данных имеют один и тот же атрибут. Указанная проблема становится особенно острой во время поступательного введения параллелизации OpenMP. При имплементировании векторизации нам нужны длинные массивы однородных данных, тогда как классы обычно группируют разнородные данные. Это усложняет дело.

4.1.1 Многомерные массивы

В этом разделе мы рассмотрим структуру многомерных массивов, повсеместно распространенную в научных вычислениях. Наша цель будет заключаться в том, чтобы понять:

- как компоновать многомерные массивы в памяти;
- как получать доступ к массивам, чтобы избегать проблем с производительностью;
- как вызывать числовые библиотеки, которые находятся в Fortran, из программы на языке C.

Обработка многомерных массивов является наиболее распространенной проблемой с точки зрения производительности. Первые два подрисунка на рис. 4.2 показывают традиционные схемы данных C и Fortran.

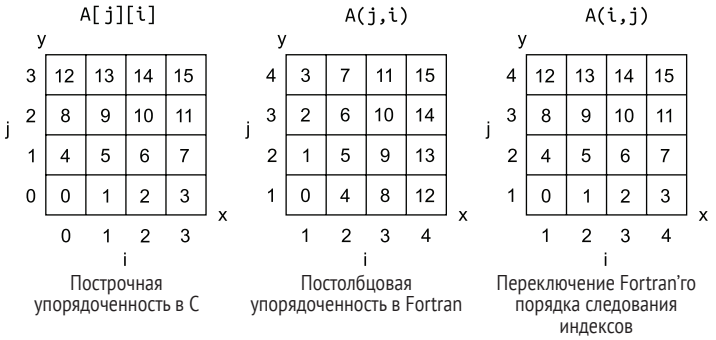


Рис. 4.2 Традиционно элементы массива в C располагаются в построчном порядке, тогда как элементы массива в Fortran располагаются в постолбцовом порядке. Переключение порядка следования индексов между Fortran или C делает их совместимыми. Обратите внимание, что индексы массива в Fortran принято начинать с 1, а в C – с 0. Кроме того, в C принято нумеровать элементы от 0 до 15 в непрерывном порядке

Порядок данных в C называется *построчным*, при котором данные в строке изменяются быстрее, чем данные в столбце. Это означает, что строчные данные располагаются в памяти в сплошном порядке. В отличие от него данные Fortran располагаются в *постолбцовом* порядке, при котором столбцовые данные изменяются быстрее всего. В практическом плане, как программисты, мы должны помнить о том, какой индекс должен находиться во внутреннем цикле, чтобы в любой ситуации использовать сплошную память (рис. 4.3).

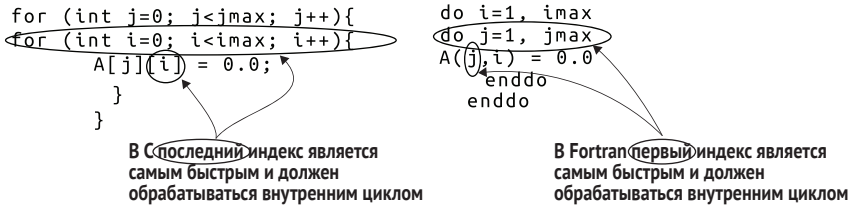


Рис. 4.3 В случае C важно помнить, что последний индекс изменяется быстрее всего и должен обрабатываться внутренним циклом вложенного цикла. В случае Fortran первый индекс изменяется быстрее всего и должен обрабатываться внутренним циклом вложенного цикла

Помимо различий между языками в упорядочении данных, существует еще один вопрос, который необходимо рассмотреть. Является ли память под весь двухмерный массив сплошной? Fortran не гарантирует,

что память будет сплошной, если не использовать атрибут `CONTIGUOUS` в массиве, как показано в приведенном ниже примере:

```
real, allocatable, contiguous :: x(:,:)
```

На практике использовать атрибут `CONTIGUOUS` не столько критически важно, как может показаться. Все популярные компиляторы Fortran выделяют для массивов сплошную память, с этим атрибутом или без него. Возможными исключениями являются холостое заполнение с целью повышения производительности кеша или передача массива через подпрограммный интерфейс с оператором нарезки. *Оператор нарезки* – это конструкт языка Fortran, который позволяет ссылаться на подмножество массива, как в примере с копированием строки двухмерного массива в одномерный с помощью синтаксиса `y(:) = x(1,:)`. Операторы нарезки также могут использоваться в вызове подпрограммы; например,

```
call write_data_row(x(1,:))
```

Некоторые исследовательские компиляторы обходятся с этой ситуацией путем простого модифицирования *индексного шага* между элементами данных в *информационном векторе* (dope vector, название происходит от англ. фигурального выражения «give me the dope», которое переводится как «дай мне дозу», под которой подразумевается информация) для массива. В Fortran информационный (или допинговый) вектор – это метаданные массива, содержащие начальную ячейку, длину массива и индексный шаг между элементами для каждой размерности. На рис. 4.4 показаны концепции информационного вектора, оператора нарезки и индексного шага. Идея состоит в том, что, модифицировав индексный шаг в информационном векторе с 1 до 4, данные тогда будут обрабатываться в виде строки, а не столбца. Но на практике производственные компиляторы Fortran обычно делают копию данных и переда-

Рис. 4.4 Разные представления массива Fortran, созданного путем модифицирования информационного вектора, набора метаданных, описывающих начало, индексный шаг и длину в каждой размерности. Оператор нарезки возвращает раздел массива Fortran со всеми элементами в размерности с помощью двоеточия (:). Можно создавать более сложные срезы, такие как четыре нижних элемента с A(1:2,1:2), где верхняя и нижняя границы задаются двоеточием

ют ее в подпрограмму, чтобы избежать нарушения кода, ожидающего сплошных данных. Это также означает, что вам следует избегать использования оператора нарезки при вызове подпрограмм Fortran из-за скрытой копии и связанной с ней стоимости для производительности.

С имеет свои проблемы со сплошной памятью под двухмерный массив. Это связано с традиционным способом динамического выделения двухмерного массива в С, как показано в следующем ниже листинге.

Листинг 4.1 Традиционный способ выделения двухмерного массива в С

```
 8 double **x =
     (double **)malloc(jmax*sizeof(double *));
 9
10 for (j=0; j<jmax; j++){
11    x[j] =
         (double *)malloc(imax*sizeof(double));
12 }
13
14 // вычисление
15
16 for (j=0; j<jmax; j++){
17    free(x[j]);
18 }
19 free(x);
```

Выделяет столбец указателей с типом pointer на тип double

Выделяет каждую строку данных

Высвобождает память

В этом листинге используются выделения `1+jmax`, и каждое выделение может поступать из другого места в куче. При использовании крупных двухмерных массивов компоновка данных в памяти оказывает лишь малое влияние на эффективность кеша. Более крупная проблема заключается в том, что использование несплошных массивов сильно ограничено; отсутствует возможность их передачи в Fortran, их поблочной записи в файл и затем их передачи в GPU или в еще один процессор. Вместо этого каждая из этих операций должна выполняться в построчном режиме. К счастью, существует простой подход к выделению сплошного блока памяти под массивы С. Почему он не является стандартной практикой? Это связано тем, что все учатся использовать традиционный метод, как в листинге 4.1, и об этом не думают. В следующем ниже листинге показан метод выделения сплошного блока памяти под двухмерный массив.

Листинг 4.2 Выделение сплошного двухмерного массива в С

```
 8 double **x =
 9    (double **)malloc(jmax*sizeof(double *));
10
11 x[0] = (void *)malloc(jmax*imax*sizeof(double));
12
13 for (int j = 1; j < jmax; j++) {
14    x[j] = x[j-1] + imax;
15 }
16
```

Выделяет блок памяти под указатели строк

Выделяет блок памяти под двухмерный массив

Назначает ячейку памяти, показывающую на блок данных под каждый указатель строки

```
17 // вычисление
18
19 free(x[0]);
20 free(x);
```

Высвобождает память

Этот метод не только дает вам сплошной блок памяти, но и требует всего двух выделений памяти! Мы можем оптимизировать это еще больше путем упаковки указателей строк в блок памяти в начале выделения сплошной памяти в строке 11 листинга 4.2, тем самым объединив два выделения памяти в одно (рис. 4.5).

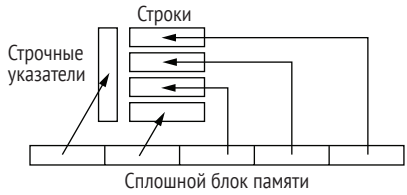


Рис. 4.5 Сплошной блок памяти становится двухмерным массивом в C

В следующем ниже листинге показана имплементация однократного выделения сплошной памяти под двухмерный массив в файле malloc2D.c.

Листинг 4.3 Однократное выделение сплошной памяти под двухмерный массив

malloc2D.c

```
 1 #include <stdlib.h>
 2 #include "malloc2D.h"
 3
 4 double **malloc2D(int jmax, int imax)
 5 {
 6    double **x = (double **)malloc(jmax*sizeof(double *) +
 7                  jmax*imax*sizeof(double));
 8
 9    x[0] = (double *)x + jmax;
10
11    for (int j = 1; j < jmax; j++) {
12       x[j] = x[j-1] + imax;
13    }
14
15    return(x);
16 }
```

Выделяет блок памяти под указатели строк и двухмерный массив

Назначает начало блока памяти под двухмерный массив после указателей строк

Назначает местоположение в памяти, которые указывает на блок данных под каждый указатель строки

malloc2D.h

```
1 #ifndef MALLOC2D_H
2 #define MALLOC2D_H
3 double **malloc2D(int jmax, int imax);
4 #endif
```

Теперь у нас есть только один блок памяти, включая массив указателей на строки. Это должно улучшить выделение памяти и эффективность кеша. Массив также может индексироваться как одномерный или двухмерный, как показано в листинге 4.4. Одномерный массив сокращает вычисление целочисленных адресов, и он легче векторизуется или потокообразуется (когда мы дойдем до этого в главах 6 и 7). В списке также показан ручной расчет двухмерного индекса в одномерном массиве.

Листинг 4.4 Одномерный и двухмерный доступ к сплошному двухмерному массиву

calc2d.c

```
 1 #include "malloc2D.h"
 2
 3 int main(int argc, char *argv[])
 4 {
 5    int i, j;
 6    int imax=100, jmax=100;
 7
 8    double **x = (double **)malloc2D(jmax,imax);
 9
10    double *x1d=x[0];
11    for (i = 0; i< imax*jmax; i++){
12       x1d[i] = 0.0;
13    }
14
15    for (j = 0; j< jmax; j++){
16       for (i = 0; i< imax; i++){
17          x[j][i] = 0.0;
18       }
19    }
20
21    for (j = 0; j< jmax; j++){
22       for (i = 0; i< imax; i++){
23          x1d[i + imax * j] = 0.0;
24       }
25    }
26 }
```

Одномерный доступ к сплошному двухмерному массиву

Двухмерный доступ к сплошному двухмерному массиву

Ручной расчет 2-го индекса для одномерного массива

Программисты Fortran считают первоклассное обращение с многомерными массивами в своем языке само собой разумеющимся. Хотя языки C и C++ существуют уже несколько десятилетий, они по-прежнему не имеют встроенного в язык собственного многомерного массива. В стандарт C++ внесены предложения по добавлению в редакцию 2023 года собственной поддержки многомерных массивов (см. ссылку на Холлман и соавт. в приложении A к книге). До тех пор описанное в листинге 4.4 выделение памяти под многомерные массивы будет по-прежнему иметь важное значение.

4.1.2 Массив структур (AoS) против структур из массивов (SoA)

В этом разделе мы рассмотрим влияние структур и классов на компоновку данных. Наши цели состоят в том, чтобы понять:

- разные способы компоновки структур в памяти;
- как получать доступ к массивам, чтобы избегать проблем с производительностью.

Существует два разных подхода к организации родственных данных в коллекции. Это *массив структур* (Array of Structures, AoS), в котором данные собираются в единое целое на самом низком уровне, а затем из структуры создается массив, либо *структура из массивов* (Structure of Arrays, SoA), в которой каждый массив данных находится на самом низком уровне, а затем из массивов создается структура. Третьим подходом, представляющим собой гибрид этих двух структур данных, является *Массив структур из массивов* (Array of Structures of Arrays, AoSoA). Мы обсудим эту гибридную структуру данных в разделе 4.1.3.

Одним из распространенных примеров массива структур (AoS) являются значения цвета, используемые для рисования графических объектов. В следующем ниже листинге показана структура цветовой системы из красного, зеленого, синего (RGB) цветов на C.

Листинг 4.5 Массив структур (AoS) на C

```
1 struct RGB {
2    int R;
3    int G;
4    int B;
5 };
6 struct RGB polygon_color[1000];
```

Определяет скалярное значение цвета

Определяет массив структур (AoS)

В листинге 4.5 показан массив AoS, в котором данные скомпонованы в памяти (рис. 4.6). На этом рисунке обратите внимание на пустое пространство в байтах 12, 28 и 44, где компилятор вставляет холостое заполнение, чтобы выровнять память на 64-битовой границе (128 бит или 16 байт). 64-байтовая строка кеша содержит четыре значения структуры. Затем в строке 6 мы создаем массив `polygon_color`, составленный из 1000 структур данных с типом `RGB`. Такая компоновка данных является разумной, поскольку, как правило, значения RGB используются вместе для рисования каждого многоугольника.

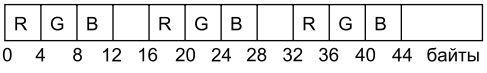


Рис. 4.6 Компоновка в памяти цветовой модели RGB в массиве структур (AoS)

Структура из массивов (SoA) представляет собой альтернативную компоновку данных. В следующем ниже листинге показан код C для этой компоновки.

Листинг 4.6 Структура из массивов (SoA) на C

```
1 struct RGB {
2    int *R;
3    int *G;
4    int *B;
5 };
6 struct RGB polygon_color;
7
8 polygon_color.R = (int *)malloc(1000*sizeof(int));
9 polygon_color.G = (int *)malloc(1000*sizeof(int));
10 polygon_color.B = (int *)malloc(1000*sizeof(int));
11
12 free(polygon_color.R);
13 free(polygon_color.G);
14 free(polygon_color.B);
```

Определяет целочисленный массив значений цвета

Определяет структуру из массивов (SoA)

Компоновка памяти содержит все 1000 значений R в сплошной памяти. Значения цветов G и B могут следовать за значениями R в памяти, но они также могут находиться в другом месте кучи, в зависимости от того, где выделитель памяти находит место. *Куча* – это отдельный участок памяти, который используется для выделения динамической памяти с помощью процедуры `malloc` или оператора `new`. Мы также можем использовать выделитель сплошной памяти (листинг 4.3), чтобы заставить память размещаться вместе.

Здесь наша забота всецело сосредоточена на производительности. Каждая из этих структур данных одинаково разумна для использования с точки зрения программиста, но важные вопросы касаются того, как структура данных выглядит для CPU и как она влияет на производительность. Давайте рассмотрим производительность этих структур данных в нескольких разных сценариях.

Оценивание производительности массива структур (AoS)

В нашем примере с цветом давайте допустим, что при чтении данных осуществляется доступ не к одному значению `R`, `G` либо `B`, а ко всем трем компонентам точки, поэтому представление в виде массива AoS работает хорошо. И эта компоновка данных обычно используется для графических операций.

ПРИМЕЧАНИЕ Если компилятор добавляет холостое заполнение, то для того, чтобы представить массив AoS, оно увеличивает число загрузок памяти на 25 %, однако не все компиляторы вставляют холостое заполнение. Тем не менее его стоит учитывать в случае тех компиляторов, которые это делают.

Если в цикле доступно только одно из значений RGB, то использование кеша будет слабым, поскольку цикл перескакивает через ненужные значения. Когда этот шаблон доступа векторизуется компилятором, ему требуется использовать менее эффективную операцию сбора/разброса (gather/scatter).

Оценивание производительности структуры из массивов (SoA)

В компоновке в виде структуры SoA значения RGB имеют отдельные строки кеша (рис. 4.7). Таким образом, для малых размеров данных, в которых необходимы все три значения RGB, обеспечивается хорошее использование кеша. Но, по мере того как массивы увеличиваются и появляется все больше массивов, система кеша начинает справляться с трудом, что приводит к снижению производительности. В этих случаях взаимодействие данных и кеша становится слишком усложненным, чтобы иметь возможность предсказывать производительность полностью.

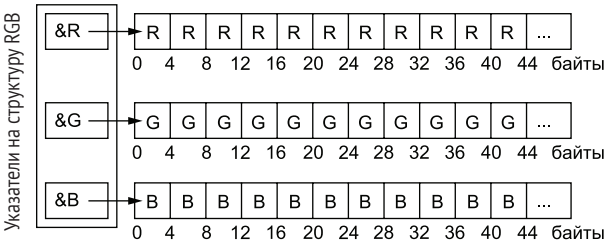


Рис. 4.7 В компоновке данных Структура из массивов (SoA) указатели располагаются в памяти смежно, указывая на отдельные сплошные массивы для каждого цвета

Еще одним часто встречающимся шаблоном компоновки и доступа к данным является использование переменных в качестве трехмерных пространственных координат в вычислительном приложении. В следующем ниже листинге показано типичное определение структуры C для указанного шаблона.

Листинг 4.7 Пространственные координаты в массиве структур C (AoS)

```
1 struct point {
2    double x, y, z;          <-- Определяет пространственную координату точки
3 };
4 struct point cell[1000];    <-- Определяет массив местоположение точек
5 double radius[1000];
6 double density[1000];
7 double density_gradient[1000];
```

Одним из применений этой структуры данных является вычисление расстояния от начала координат (радиуса) следующим образом:

```
10 for (int i=0; i < 1000; i++){
11    radius[i] = sqrt(cell[i].x*cell[i].x + cell[i].y*cell[i].y +
     cell[i].z*cell[i].z);
12 }
```

Значения x, y и z заносятся вместе в одну строку кеша и пишутся в переменную radius во второй строке кеша. Использование кеша в этом случае является разумным. Но во втором вполне возможном случае вычислительный цикл мог бы использовать местоположение x для вычисления градиента плотности в направлении x, как вот здесь:

```
20 for (int i=1; i < 1000; i++){
21    density_gradient[i] = (density[i] - density[i-1])/
                            (cell[i].x - cell[i-1].x);
22 }
```

Теперь доступ к кешу для x перескакивает через данные y и z, вследствие чего используется только одна треть (или даже одна четверть, если имеется холостое заполнение) данных в кеше. Отсюда оптимальная компоновка данных полностью зависит от использования и конкретных шаблонов доступа к данным.

В случаях смешанного использования, которые, вероятно, появятся в реальных приложениях, иногда структурные переменные используются вместе, а иногда нет. Как правило, компоновка AoS в целом показывает более высокую производительность на CPU, тогда как компоновка SoA показывает более высокую производительность на GPU. В сообщаемых результатах имеется столь много вариабельности, что стоит провести тестирование конкретного шаблона использования. В случае градиента плотности в следующем ниже листинге показан код для структуры из массивов, SoA.

Листинг 4.8 Структура из массивов для пространственных координат (SoA)

```
 1 struct point{
 2    double *x, *y, *z;
 3 };
 4 struct point cell;
 5 cell.x = (double *)malloc(1000*sizeof(double));
 6 cell.y = (double *)malloc(1000*sizeof(double));
 7 cell.z = (double *)malloc(1000*sizeof(double));
 8 double *radius = (double *)malloc(1000*sizeof(double));
 9 double *density = (double *)malloc(1000*sizeof(double));
10 double *density_gradient = (double *)malloc(1000*sizeof(double));
11 // ... инициализировать данные
12
13 for (int i=0; i < 1000; i++){
14    radius[i] = sqrt(cell.x[i]*cell.x[i] +
                       cell.y[i]*cell.y[i] +
                       cell.z[i]*cell.z[i]);
```

Определяет массивы пространственных местоположений

Определяет структуру из пространственных местоположений ячеек

В этом цикле используются сплошные значения массивов

```
15 }
16
17 for (int i=1; i < 1000; i++){
18    density_gradient[i] = (density[i] - density[i-1])/
                            (cell.x[i] - cell.x[i-1]);
19 }
20
21 free(cell.x);
22 free(cell.y);
23 free(cell.z);
24 free(radius);
25 free(density);
26 free(density_gradient);
```

В этом цикле используются сплошные значения массивов

При такой компоновке данных каждая переменная заносится в отдельную строку кеша, и использование кеша будет полезно для обоих вычислительных ядер. Но, по мере того как число требуемых членов данных становится значительно крупнее, кеш начинает испытывать трудности с эффективной обработкой большого числа потоков (streams) памяти. В объектно ориентированной имплементации на C++ вам следует остерегаться и других подводных камней. В следующем ниже листинге представлен класс ячеек с пространственной координатой ячейки и радиусом в качестве его компонентов данных, а также метод вычисления радиуса из x, y и z.

Листинг 4.9 Пример класса пространственных координат на C++

```
1 class Cell{
2       double x;
3       double y;
4       double z;
5       double radius;
6    public:
7       void calc_radius() {
             radius = sqrt(x*x + y*y + z*z);
        }
8       void big_calc();
9 }
10
11 Cell my_cells[1000];
12
13 for (int i = 0; i < 1000; i++){
14    my_cells[i].calc_radius();
15 }
16
17 void Cell::big_calc(){
18    radius = sqrt(x*x + y*y + z*z);
19    // ... еще больше кода, предотвращающего встраивание в месте вызова
20 }
```

Вызывает функцию радиуса для каждой ячейки

Определяет массив объектов в виде массива структур

Выполнение этого кода приводит к паре неуспешных обращений к кешу команд и накладным расходам из-за вызовов подпрограмм для

каждой ячейки. Неуспешные обращения к кешу команд происходят, когда последовательность команд делает переход (прыжок) и следующей команды нет в кеше команд. Существует два кеша 1-го уровня: один для данных программы, а второй для инструкций процессора. Вызовы подпрограмм требуют дополнительных накладных расходов, связанных с переносом аргументов в стек перед вызовом и переходом к команде. Находясь в процедуре, аргументы должны быть вытолкнуты из стека, а затем, в конце процедуры, происходит еще один прыжок к инструкции. В этом случае код настолько прост, что компилятор может встроить процедуру в месте вызова, чтобы избежать этих затрат. Но в более сложных случаях, таких как процедура `big_calc`, это невозможно. Вдобавок строка кеша вытаскивает `x`, `y`, `z` и радиус. Кеш помогает ускорить загрузку координат местоположения, которые и в самом деле необходимо прочитать. Но радиус, который требует записи, тоже находится в строке кеша. Если разные процессоры записывают значения радиуса, то это может сделать строки кеша недействительными и потребовать от других процессоров перезагрузить данные в свои кеши.

В C++ есть много функциональностей, которые упрощают программирование. Как правило, их следует применять в коде на более высоком уровне, используя более простой процедурный стиль C и Fortran, где важна производительность. В приведенном выше листинге вычисление радиуса может выполняться как массив, а не как одиночный скалярный элемент. Указатель класса может дереференсироваться один раз в начале процедуры во избежание повторного дереференсирования и возможных неуспешных обращений к кешу команд. *Дереференсирование* – это операция, при которой адрес памяти получается из ссылки указателя, вследствие чего строка кеша посвящается данным памяти, а не указателю[1]. В простых хеш-таблицах тоже может использоваться структура для группирования ключа и значения в одном месте, как показано в следующем ниже листинге.

Листинг 4.10 Хеш-массив структур (AoS)

```
1 struct hash_type {
2    int key;
3    int value;
4 };
5 struct hash_type hash[1000];
```

[1] В языках C/C++, Rust и подобных дереференсирование (dereferencing) – это операция опосредования, которая состоит в получении существующего значения по указателю с помощью оператора дереференции *. Референсирование (referencing) – это операция опосредования, которая заключается в создании указателя на существующее значение путем получения его адреса в памяти с помощью оператора референции &. Термин «разыменование» для «dereferencing» разрывает парную связь с referencing и вносит пустую сущность «именование», тогда как речь идет об операции с указателем, а не с именем. – *Прим. перев.*

Проблема с этим кодом заключается в том, что он читает несколько ключей до тех пор, пока не найдет совпадающий, а затем читает значение для этого ключа. Но ключ и значение заносятся в единую строку кеша, и значение игнорируется до тех пор, пока не произойдет совпадение. С целью обеспечения более быстрого поиска по ключам лучше иметь ключ в качестве одного массива, а значение в качестве другого, как показано в следующем ниже листинге.

Листинг 4.11 Хеш-структура из массивов (SoA)

```
1 struct hash_type {
2    int *key;
3    int *value;
4 } hash;
5 hash.key = (int *)malloc(1000*sizeof(int));
6 hash.value = (int *)malloc(1000*sizeof(int));
```

В качестве последнего примера возьмем структуру с физическим состоянием, которая содержит плотность, трехмерный моментум и суммарную энергию. Эта структура приводится в следующем ниже листинге.

Листинг 4.12 Массив структур с физическими состояниями (AoS)

```
1 struct phys_state {
2    double density;
3    double momentum[3];
4    double TotEnergy;
5 };
```

При обработке только плотности следующие четыре значения в кеше остаются неиспользованными. Опять же, лучше иметь эти данные в качестве структуры из массивов, SoA.

4.1.3 Массив структур из массивов (AoSoA)

Бывают случаи, когда эффективны гибридные группировки структур и массивов. Массив структур из массивов (Array of Structures of Arrays, AoSoA) можно использовать для «плиточного разбиения» (tiling) данных на векторные длины. Давайте введем обозначение `A[len/4]S[3]A[4]` для представления этой компоновки. `A[4]` представляет собой массив из четырех элементов данных и является внутренним сплошным блоком данных. `S[3]` представляет следующий уровень структуры данных из трех полей. Комбинация `S[3]A[4]` дает компоновку данных, показанную на рис. 4.8.

Нам нужно повторить блок из 12 значений данных `A[len/4]` раз, чтобы получить все данные. Если мы заменим 4 переменной, то получим:

```
A[len/V]S[3]A[V], где V=4
```

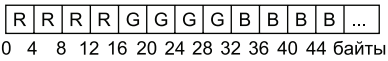


Рис. 4.8 Массив структур из массивов (AoSoA) используется с последней длиной массива, соответствующей векторной длине оборудования для длины вектора, равной четырем

В C или Fortran, соответственно, массив может получить размерность

```
var[len/V][3][V], var(1:V,1:3,1:len/V)
```

В C++ это было бы имплементировано естественным образом, как показано в следующем ниже листинге.

Листинг 4.13 Массив структур из массивов RGB (AoSoA)

```
1 const int V=4;
2 struct SoA_type{
3    int R[V], G[V], B[V];
4 };
5
6 int main(int argc, char *argv[])
7 {
8    int len=1000;
9    struct SoA_type AoSoA[len/V];
10
11   for (int j=0; j<len/V; j++){
12      for (int i=0; i<V; i++){
13         AoSoA[j].R[i] = 0;
14         AoSoA[j].G[i] = 0;
15         AoSoA[j].B[i] = 0;
16      }
17   }
18 }
```

Задает длину вектора

Делит длину массива на длину вектора

Прокручивает в цикле длину массива

Прокручивает в цикле длину вектора, который должен векторизоваться

Варьируя `V` в соответствии с аппаратной длиной вектора или размером рабочей группы GPU, мы создаем переносимую абстракцию данных. В добавление к этому, определив `V=1` или `V=len`, мы восстанавливаем структуры данных AoS и SoA, в указанном порядке. Тогда эта компоновка данных становится способом адаптации под оборудование и шаблоны использования программных данных.

Необходимо решить ряд деталей, касающихся имплементирования этой структуры данных, чтобы минимизировать затраты на индексацию и решить, следует ли применять холостое заполнение массива для повышения производительности. Компоновка данных AoSoA обладает некоторыми свойствами структур данных AoS и SoA, поэтому производительность в целом близка к лучшей из двух, как показано в исследовании Роберта Берда (Robert Bird) из Лос-Аламосской национальной лаборатории (рис. 4.9).

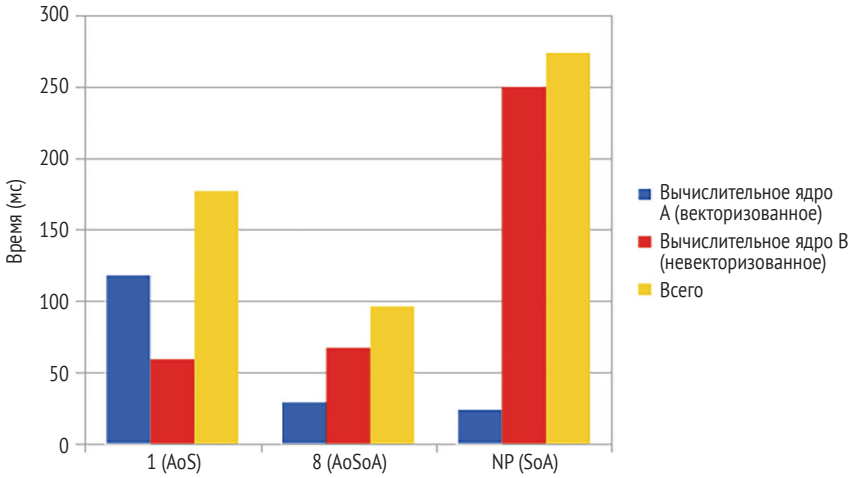


Рис. 4.9 Производительность массива структур из массивов (AoSoA) в целом соответствует лучшим характеристикам AoS и SoA. Длина массива 1, 8 и NP в легенде оси x является значением для последнего массива в AoSoA. Эти значения означают, что первый набор сводится к AoS, последний набор сводится к SoA, а средний набор имеет длину второго массива 8, соответствующую векторной длине процессора

4.2 Три категории неуспешных обращений к кешу: вынужденное, емкостное и конфликтное

Эффективность кеша доминирует над производительностью интенсивных вычислений. Пока данные кешированы, вычисления проходят быстро. Когда данные не кешированы, происходит *неуспешное обращение к кешу* (cache miss). Тогда процессор должен сделать паузу и дождаться загрузки данных. Стоимость неуспешного обращения к кешу составляет порядка 100–400 циклов; за то же самое время могут быть проделаны 100 флопов! В целях обеспечения производительности мы должны минимизировать неуспешные обращения к кешу. Но минимизация неуспешных обращений к кешу требует понимания того, как данные перемещаются из основной памяти в CPU. Это делается с помощью простой модели производительности, которая разделяет неуспешные обращения к кешу на три категории: *вынужденные*, *емкостные* и *конфликтные*. Сначала мы должны понять принцип работы кеша.

При загрузке данных они загружаются в блоки, именуемые строками кеша, которые обычно имеют длину 64 байта. Затем они вставляются в местоположение кеша на основе его адреса в памяти. В *прямо отображаемом кеше* (direct-mapped cache) есть только одно место для загрузки данных в кеш. Это важно, когда два массива отображаются с одним и тем

же местоположением. При использовании прямо отображаемого кеша можно кешировать только по одному массиву за раз. Во избежание этого большинство процессоров имеет *N-путный секторно-ассоциативный кеш* (N-way set associative cache), предоставляющий *N* местоположений, в которые загружаются данные. При регулярных, предсказуемых доступах к памяти крупных массивов имеется возможность выбирать данные *упреждающе*. То есть вы можете выдавать команду предзагрузки данных до того, как они понадобятся, чтобы они уже были в кеше. Это может делаться компилятором как аппаратно, так и программно.

Вытеснение (eviction) – это удаление строки кеша из одного или нескольких уровней кеша. Причиной тому может быть загрузка строки кеша в одно и то же место (*кеш-конфликт*) или лимитированный размер кеша (*емкостное неуспешное обращение*). *Операция сохранения* в результате присваивания в цикле вызывает выделение для записи (write-allocate) в кеше, где создается и модифицируется новая строка кеша. Эта строка кеша вытесняется (сохраняется) в основную память, хотя это может произойти не сразу. Имеются самые разные политики записи, которые влияют на детали операций записи. Вынесенные в заголовок раздела три категории представляют собой простой подход к пониманию источника неуспешных обращений к кешу, которые влияют на производительность интенсивных вычислений времени выполнения.

- *Вынужденный* – неуспешные обращения к кешу, необходимые для занесения данных при их первом обнаружении.
- *Емкостный* – неуспешные обращения к кешу, обусловленные лимитированным размером кеша, который вытесняет данные из кеша, чтобы высвободить место для загрузки новых строк.
- *Конфликтный* – когда данные загружаются в одно и то же место в кеше. Если одновременно требуются два или более элемента данных, но они соотносятся с одной и той же строкой кеша, то оба элемента данных должны загружаться повторно для каждого доступа к элементу данных.

Когда неуспешные обращения к кешу происходят из-за нехватки емкости или из-за конфликтов с последующей перезагрузкой строк кеша, это иногда называется *перетиранием кеша* (cache thrashing), которое может приводить к снижению производительности. Исходя из этих определений, мы можем легко рассчитать несколько характеристик вычислительного ядра и по меньшей мере получить представление об ожидаемой производительности. Для этого мы будем использовать вычислительное ядро оператора размытия, показанное на рис. 1.10.

В листинге 4.14 показано вычислительное ядро stencil.c. Мы также используем процедуру выделения двухмерной сплошной памяти из файла malloc2D.c из раздела 4.1.1. Таймерный код здесь не показан, но он находится в онлайновом исходном коде. Включены таймеры и вызовы профилировщика likwid («будто я знаю, что делаю»). Между итерациями выполняется запись в крупный массив для опустошения кеша, вследствие которой в нем нет соответствующих данных, могущих исказить результаты.

Листинг 4.14 Стенсильное ядро для оператора размытия в примере с Кракатау

```
stencil.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include "malloc2D.h"
#include "timer.h"
#include "likwid.h"
#define SWAP_PTR(xnew,xold,xtmp) (xtmp=xnew, xnew=xold, xold=xtmp)
int main(int argc, char *argv[]){
   LIKWID_MARKER_INIT;
   LIKWID_MARKER_REGISTER("STENCIL");
   struct timeval tstart_cpu, tstop_cpu;
   double cpu_time;
   int imax=2002, jmax = 2002;
   double **xtmp, *xnew1d, *x1d;
   double **x = malloc2D(jmax, imax);
   double **xnew = malloc2D(jmax, imax);
   int *flush = (int *)malloc(jmax*imax*sizeof(int)*10);
   xnew1d = xnew[0]; x1d = x[0];
   for (int i = 0; i < imax*jmax; i++){
      xnew1d[i] = 0.0; x1d[i] = 5.0;}
   for (int j = jmax/2 - 5; j < jmax/2 + 5; j++){
      for (int i = imax/2 - 5; i < imax/2 -1; i++){
         x[j][i] = 400.0;}}
   for (int iter = 0; iter < 10000; iter++){
      for (int l = 1; l < jmax*imax*10; l++){ flush[l] = 1.0; }
      cpu_timer_start(&tstart_cpu);
      LIKWID_MARKER_START("STENCIL");
      for (int j = 1; j < jmax-1; j++){
         for (int i = 1; i < imax-1; i++){
            xnew[j][i] = ( x[j][i] + x[j][i-1] + x[j][i+1] + x[j-1][i] + x[j+1][i] ) / 5.0;
      } }
      LIKWID_MARKER_STOP("STENCIL");
      cpu_time += cpu_timer_stop(tstart_cpu);
      SWAP_PTR(xnew, x, xtmp);
      if (iter%100 == 0) printf("Итерация %d\n",iter);
   }
   printf("Хронометраж: %f\n",cpu_time);
   free(x); free(xnew); free(flush);
   LIKWID_MARKER_CLOSE;
}
```

Инициализирование likwid и регистрирование имени маркера

Использование этого трюка с указателем на одномерный массив для инициализирования массивов

Инициализирование блока памяти в центре более крупным значением

Опустошение кеша

Загрузка памяти – пять загрузок и одно сохранение

Запуск и остановка маркера

Операции с плавающей точкой = 5 флопов

Калькуляционное ядро

Закрытие likwid

Если у нас есть идеально эффективный кеш, то после загрузки данных в память они сразу же там сохраняются. Конечно же, в большинстве случаев это далеко от реальности. Но с помощью этой модели мы можем рассчитать следующее:

- суммарную используемую память = 2000 × 2000 × (5 ссылок + 1 сохранение) × 8 байт = 192 Мб;

- вынужденно загруженную и сохраненную память = 2002 × 2002 × 8 байт × 2 массива = 64.1 Мб;
- арифметическую интенсивность = 5 флопов × 2000 × 2000 / 64.1 Мбайт = .312 флопов/байт или 2.5 флопов/слово.

Затем программа компилируется с помощью библиотеки likwid и выполняется на процессоре Skylake 6152 следующей ниже командой:

```
likwid-perfctr -C 0 -g MEM_DP -m ./stencil
```

Нужный нам результат находится в конце таблицы производительности, печатаемой в конце прогона:

```
+----------------------------------+------------+
| ...                              |            |
|            DP MFLOP/s            |  3923.4952 |
|          AVX DP MFLOP/s          |  3923.4891 |
| ...                              |            |
|       Operational intensity      |     0.247  |
+----------------------------------+------------+
```

Данные о производительности стенсильного ядра представлены в виде графика контура крыши с использованием скрипта Python (имеется в онлайновых материалах) и показаны на рис. 4.10. График контура крыши, представленный в разделе 3.2.4, показывает аппаратные пределы максимальных операций с плавающей точкой и максимальной пропускной способностью в зависимости от арифметической интенсивности.

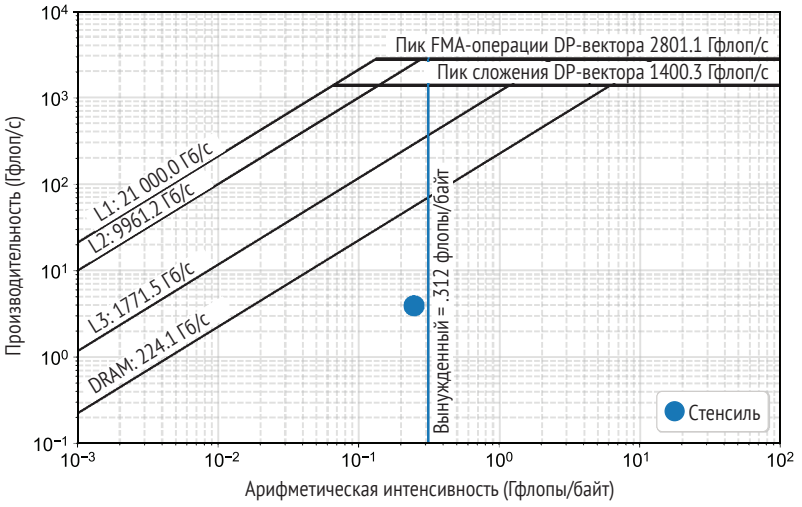


Рис. 4.10 График контура крыши стенсильного ядра для примера с вулканом Кракатау из главы 1 показывает вынужденную верхнюю границу справа от измеренной производительности

На этом графике контура крыши показан вынужденный предел данных для правой части измеренной арифметической интенсивности 0.247

(показанной большой точкой на рис. 4.10). Стенсильное ядро не способно добиться лучшего, чем вынужденный лимит, если у него есть холодный кеш. *Холодный кеш* – это кеш, в котором нет никаких релевантных данных, полученных в результате каких-либо операций, выполнявшихся до входа в ядро. Расстояние между большой точкой и вынужденным пределом дает нам представление о степени эффективности кеша в этом ядре. Стенсильное ядро в данном случае является простым, и емкостная и конфликтная загрузки кеша всего примерно на 15 % превышают вынужденную загрузку кеша. Таким образом, существует не так много возможностей для улучшения производительности ядра. Расстояние между большой точкой и линией крыши DRAM объясняется тем, что это ядро является последовательным ядром с векторизацией, в то время как линии крыши параллельны с OpenMP. Таким образом, есть потенциал для повышения производительности за счет добавления параллелизма.

Поскольку этот график является дважды логарифмическим, различия имеют более высокую величину, чем могут показаться. Если присмотреться повнимательнее, то возможное улучшение от параллелизма имеет величину почти на порядок. Улучшение использования кеша может быть достигнуто за счет использования других значений в строке кеша либо многократного использования данных, пока они находятся в кеше. Эти два разных случая именуются соответственно пространственной локальностью и темпоральной локальностью:

- *пространственная локальность* относится к данным с близлежащими местоположениями в памяти, которые часто адресуются близко друг к другу;
- *темпоральная локальность* относится к недавно адресовавшимся данным, которые, вероятно, будут адресоваться снова в ближайшем будущем.

Для стенсильного ядра (листинг 4.14), когда значение `x[1][1]` заносится в кеш, `x[1][2]` тоже заносится в кеш. Это пространственная локальность. На следующей итерации цикла для вычисления `x[1][2]` необходимо `x[1][1]`. Оно все еще должно находиться в кеше и повторно использоваться в случае темпоральной локальности.

Четвертая категория, часто добавляемая к трем упомянутым ранее, станет важной в последующих главах. Она называется *когерентностью*.

ОПРЕДЕЛЕНИЕ *Когерентность* применяется к тем обновлениям кеша, которые необходимы для синхронизирования кеша между мультипроцессорами, когда данные, записанные в кеш одного процессора, также содержатся в кеше другого процессора.

Обновления кеша, необходимые для поддержания когерентности, иногда могут приводить к интенсивному трафику на шине памяти и иногда называются *штурмами обновлений кеша*. Указанные штурмы обновлений кеша могут приводить к снижениям производительности, а не к ускорениям, когда в параллельное задание добавляются дополнительные процессоры.

4.3 Простые модели производительности: тематическое исследование

В этом разделе рассматривается пример использования простых моделей производительности для принятия обоснованных решений о том, какую структуру данных использовать для многоматериальных расчетов в физическом приложении. В указанном примере используется реальное тематическое исследование, чтобы показать эффекты:

- простых моделей производительности для вопроса о реальном программном дизайне;
- сжато-разреженных структур данных для увеличения ваших вычислительных ресурсов

В некоторых сегментах вычислительной науки уже давно используются сжатые разреженные матричные представления. Наиболее примечательным является формат сжатой разреженной строки (Compressed Sparse Row, CSR), используемый для разреженных матриц с середины 1960-х годов с отличными результатами. Для оцениваемой в этом примере сжато-разреженной структуры данных экономия памяти превышает 95 %, а время выполнения приближается к ускорению на 90 %, чем при простом дизайне с двухмерным массивом. Используемые простые модели производительности предсказывали производительность с погрешностью 20–30 % от фактической измеренной производительности (см. Фогерти, Маттино и соавт. в разделе о дополнительном чтении далее в этой главе). Но использование этой сжатой схемы сопряжено с затратами усилий программиста. Мы хотим использовать сжато-разреженную структуру данных, где ее преимущества перевешивают затраты. Это решение принимается там, где простая модель производительности действительно показывает свою полезность.

Пример: моделирование пеплового шлейфа вулкана Кракатау

Ваш коллектив рассматривает возможность моделирования пеплового шлейфа из примера главы 1. Все осознают, что число пепловых материалов в шлейфе может в конечном итоге достигать 10 или даже 100, но эти материалы не обязательно должны быть в каждой ячейке. Может ли сжато-разреженное представление быть полезным в этой ситуации?

Простые модели производительности полезны разработчику приложений при решении более сложных задач программирования, чем просто двойной вложенный цикл над двухмерным массивом. Цель этих моделей состоит в том, чтобы получить грубую оценку производительности с помощью простого подсчета операций в характерном вычислительном ядре для принятия решений об альтернативах программирования. Простые модели производительности немного сложнее, чем модель «трех категорий». Базовый процесс состоит в том, чтобы посчитать и отметить следующее:

- загрузки и сохранения памяти (мемопы);
- операции с плавающей точкой (флопы);
- загрузки сплошной и несплошной памяти;
- наличие ветвлений;
- малые циклы.

Мы будем подсчитывать загрузки и сохранения памяти (коллективно именуемые *мемопами*, memops) и флопы, но мы также будем обращать внимание на то, являются ли загрузки памяти сплошными и есть ли ветвления, которые могут повлиять на производительность. Мы также будем использовать эмпирические данные, такие как пропускная способность потока данных и обобщенные подсчеты операций, чтобы преобразовывать их в оценки производительности. Если загрузки памяти не являются сплошными, то используется только одно из восьми значений в строке кеша, поэтому в таких случаях мы делим пропускную способность потока данных на 8 или меньше.

Для последовательной части этого исследования мы будем использовать аппаратную производительность MacBook Pro с 6-Мб кешем L3. Частота данного процессора (v) составляет 2.7 ГГц. Измеренная пропускная способность потока данных составляет 13 375 Мб/с с использованием техники, представленной в разделе 3.2.4 с кодом приложения сравнительного тестирования потоков данных, STREAM Benchmark.

Если в алгоритмах с ветвлением мы делаем условный переход, двигаясь по ветви, почти все время, то стоимость ветвления невелика. Когда выбранная ветвь встречается нечасто, мы добавляем стоимость предсказания ветвления (B_c) и, возможно, стоимость неуспешной упреждающей выборки (P_c). В простой модели предсказателя ветвей используется в качестве вероятного пути наиболее частый случай за последние несколько итераций. Это снижает стоимость, если существует некоторая кластеризация путей ветвлений из-за локальности данных. Штраф за ветвление (B_p) становится $N_b B_f (B_c + P_c)/v$. Для типичных архитектур стоимость предсказания ветвления (B_c) составляет около 16 циклов, а стоимость неуспешной упреждающей выборки (P_c) эмпирически определяется как примерно 112 циклов. N_b – это число появлений ветвления, а B_f – *частота неуспешного ветвления*. Накладные расходы на циклы для малых циклов неизвестной длины также назначаются стоимостью (L_c) для учета ветвления и контроля. Стоимость программного цикла (или замкнутой цепи) оценивается примерно в 20 тактовых циклов в расчете на выход из него. Штраф за программный цикл (L_p) становится L_c/v.

Мы будем использовать простые модели производительности в исследовании дизайна, рассматривая возможные многоматериальные структуры данных для физических симуляций. Цель этого исследования состоит в том, чтобы определить структуры данных, которые будут обеспечивать наилучшую производительность, прежде чем писать какой-либо код. В прошлом выбор делался на основе субъективного суждения, а не на объективной основе. Рассматриваемый частный случай является редким, когда в вычислительной сетке много материалов, но только один или несколько материалов в любой вычислительной ячейке. При обсуж-

дении возможных компоновок данных мы будем ссылаться на небольшую образцовую сетку из четырех материалов, рис. 4.11. В трех ячейках содержится только один материал, в то время как в ячейке 7 – четыре.

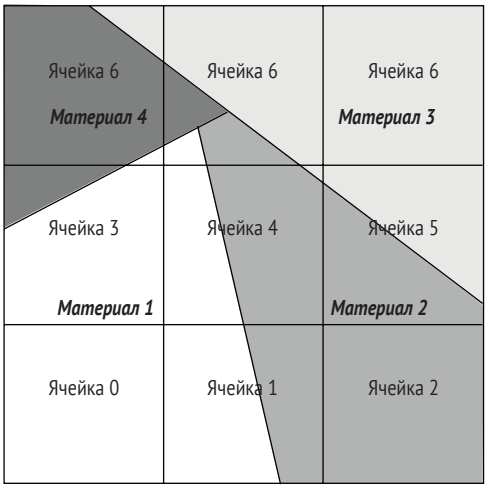


Рис. 4.11 Вычислительная сетка 3×3 показывает, что ячейка 7 содержит четыре материала

Структура данных – это только половина истории. Нам также необходимо оценить компоновку данных в нескольких репрезентативных вычислительных ядрах путем:

1. вычисления `pavg[C]`, средней плотности материалов в ячейках сетки;
2. оценивания `p[C][m]`, давления в каждом материале, содержащемся в каждой ячейке, с использованием закона идеального газа: $p(p, t) = nrt/v$.

Оба этих вычисления имеют арифметическую интенсивность 1 флоп на слово или ниже. Мы также ожидаем, что эти вычислительные ядра будут лимитированы по пропускной способности. Мы будем использовать два крупных набора данных для проверки производительности ядер. Оба представляют собой 50 материалов (H_m), 1 млн ячеечных задач (N_c) с четырьмя массивами состояний (N_v). Массивы состояний представляют плотность (p), температуру (t), давление (p) и объемную долю (V_f). Упомянутые два набора данных таковы:

- *задача с геометрическими контурами* – сетка, инициализированная из вложенных прямоугольников материалов (рис. 4.12). Сетка представляет собой правильную прямоугольную решетку. Поскольку материалы расположены в отдельных прямоугольниках, а не разбросаны, в большинстве ячеек есть только один или два материала. В результате получается 95 % чистых ячеек (P_f) и 5 % смешанных ячеек (M_f). Эта сетка имеет некоторую локальность данных, поэтому неуспешность предсказания ветвления (B_p) примерно оценивается равной 0.7;

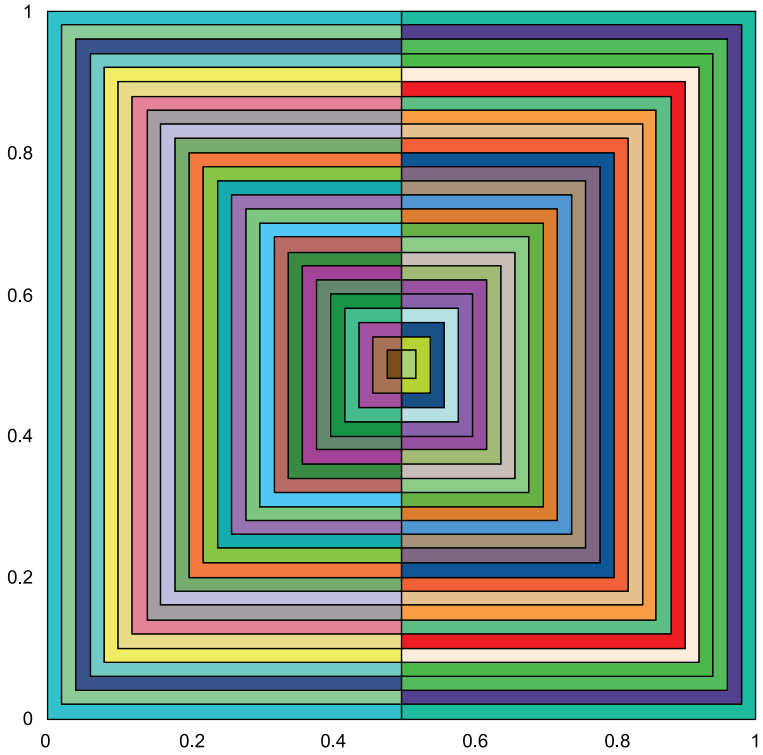


Рис. 4.12 Пятьдесят вложенных половинных прямоугольников, используемых для инициализирования сетки для тестового случая с геометрическими контурами

- *случайно инициализированная задача* – случайно инициализированная сетка с 80 % чистых ячеек и 20 % смешанных ячеек. Поскольку локальность данных невелика, неуспешность предсказания ветвления (B_p) оценивается равной 1.0.

В анализе производительности из разделов 4.3.1 и 4.3.2 рассматриваются два главенствующих соображения по дизайну: компоновка данных и порядок циклов. Мы называем компоновку данных ячеечно-центричной либо материало-центричной в зависимости от большего организующего фактора в данных. Фактор компоновки данных имеет большое значение в порядке данных. Мы ссылаемся на шаблон циклического доступа как ячеечно- или материало-доминантный, чтобы обозначить, какой из них является внешним циклом. Наилучшая ситуация возникает, когда компоновка данных согласуется с шаблоном циклического доступа. Идеального решения не существует; одно ядро предпочитает одну компоновку, а другое ядро – другую.

4.3.1 Полноматричные представления данных

Самая простая структура данных – это представление в виде *полноматричного хранения*. Оно предполагает, что каждый материал находится

в каждой ячейке. Такие полноматричные представления аналогичны даухмерным массивам, обсуждавшимся в предыдущем разделе.

Полноматричное ячеечно-центричное хранение

Для малой задачи из рис. 4.11 (вычислительная сетка 3×3) на рис. 4.13 показана ячеечно-центричная компоновка данных. Порядок данных соответствует принятым правилам языка C в отношении материалов, хранящихся сплошняком для каждой ячейки. Другими словами, программное представление имеет вид `variable[C][m]`, где `m` изменяется быстрее всего. На рисунке затененные элементы представляют собой смешанные материалы в ячейке. Чистые ячейки содержат просто значение 1.0. Элементы с тире указывают на то, что ни один из этих материалов не находится в ячейке, и поэтому в данном представлении ему назначается ноль. В данном простом примере около половины элементов матрицы имеют нули, но в более серьезной задаче число нулевых элементов будет превышать 95 %. Число ненулевых элементов называется *долей заполненности* (F_f), и для нашего сценария дизайна обычно составляет менее 5 %. Таким образом, при использовании схемы сжато-разреженного хранения экономия памяти составит более 95 % даже с учетом дополнительных затрат на хранение более сложных структур данных.

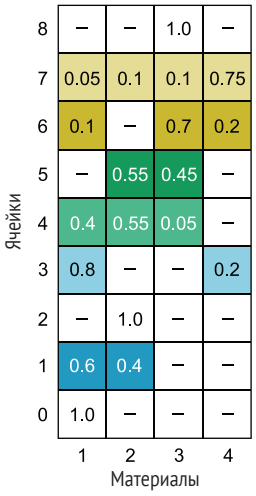


Рис. 4.13 Полноматричная ячеечно-центричная структура данных, в которой материалы по каждой ячейке хранятся сплошняком

Полноматричный подход к данным имеет преимущество в том, что он проще и, следовательно, легче параллелизуется и оптимизируется. Экономия памяти достаточно существенна, поэтому, вероятно, стоит использовать сжато-разреженную структуру данных. Но каковы последствия этого метода для производительности? Мы можем выдвинуть догадку, что наличие большего объема памяти под данные потенциально увеличивает пропускную способность памяти и замедляет полноматричное представление. Но что, если мы проверим объемную долю и если она равна нулю, то пропустим доступ к смешанному материалу? На рис. 4.14

показано то, как мы тестировали этот подход, где псевдокод для ячеечно-доминантного цикла показан вместе с подсчетами для каждой операции слева от строки кода. Структура ячеечно-доминантного цикла имеет индекс ячейки во внешнем цикле, который совпадает с индексом ячейки в качестве первого индекса в ячеечно-центричной структуре данных.

```
1: for all ячейки, C, вплоть до N_c do
2:   ave ← 0.0
3: for all ИДы материалов, m, вплоть до N_m do
4:     if V_f [C][m] > 0.0 then          # N_c N_m загружает (V_f)
                                         # B_p N_c N_m штраф на ветвление
5:       ave ← ave + ρ[C][m] * f[C][m]   # 2F_f N_c N_m загружает (ρ, f)
                                         # 2F_f N_c N_m флопов (+, * )
6:     end if
7:   end for
8:   ρ_ave[C] ← ave/V[C]                 # N_c сохраняет (ρ_ave), N_c загружает (V)
                                         # N_c флопов (l)
9: end for
```

Рис. 4.14 Модифицированный ячеечно-доминантный алгоритм для вычисления средней плотности ячеек с использованием полноматричного хранения

Подсчеты резюмированы из примечаний к строкам кода (начинающихся с #) на рис. 4.14 следующим образом:

$$\text{мемопы} = N_c(N_m + 2F_f N_m + 2) = 54.1 \text{ Ммемопов},$$

$$\text{флопы} = N_c(2F_f N_m + 1) = 3.1 \text{ Мфлопов},$$

$$N_c = 1e6;\ N_m = 50;\ F_f = .021.$$

Если мы посмотрим на флопы, то придем к выводу, что мы действовали эффективно и производительность была бы отличной. Но в этом алгоритме будет явно доминировать пропускная способность памяти. Для оценивания производительности пропускной способности памяти нам необходимо учитывать неуспешность предсказания ветвления. Поскольку переход по ветви делается столь редко, вероятность неуспешности предсказания ветвления высока. Задача с геометрическими контурами имеет некоторую локальность, поэтому уровень неуспешности предсказания оценивается равным 0.7. Собрав все это вместе, мы получаем для нашей модели производительности (PM) следующее ниже:

$$PM = N_c(N_m + F_f N_m + 2) \times 8/\text{Поток данных} + B_p F_f N_c N_m = 67.2 \text{ мс};$$

$$B_f = 0.7;\ B_c = 16;\ P_c = 16;\ v = 2.7.$$

Стоимость неуспешности предсказания ветвления делает время выполнения высоким; выше, чем если бы мы просто пропускали условный переход и добавляли нули. Более длинные циклы амортизировали бы штрафную стоимость, но очевидно, что условие, которое редко принимается, не является самым лучшим сценарием для производительности. Мы

также могли бы вставить операцию упреждающей выборки перед условием, чтобы принудительно загружать данные в случае, если переход по ветви будет сделан. Но это увеличило бы мемопы, поэтому фактическое повышение производительности было бы малым. Это также увеличит трафик на шине памяти, приводя к перегрузке, которая будет запускать другие проблемы, в особенности при добавлении параллелизма потоков.

Полноматричное материало-центричное хранение

Теперь давайте взглянем на материало-центричную структуру данных (рис. 4.15). Обозначением в C для нее будет `variable[m][C]`, где самый правый индекс `C` (или ячейки) изменяется быстрее всего. На рисунке тире обозначают элементы, заполненные нулями. Многие характеристики этой структуры данных аналогичны полноматричному ячеечно-центричному представлению данных, но с перевернутыми индексами хранения.

Рис. 4.15 Полноматричная материало-центричная структура данных хранит ячейки в сплошном порядке для каждого материала. Индексирование массива в C будет иметь `density[m][C]`, где индекс ячеек будет сплошным. Ячейки с тире заполняются нулями

Алгоритм вычисления средней плотности каждой ячейки можно сделать с загрузками сплошной памяти и при небольшом размышлении. Естественный способ имплементирования этого алгоритма состоит в создании внешнего цикла над ячейками, инициализирования его там нулями и делении на объем в конце. Но он будет шагать по данным в несплошном порядке. Мы хотим перебирать ячейки во внутреннем цикле, требуя отдельных циклов до и после главного цикла. На рис. 4.16 этот алгоритм показан вместе с аннотациями для мемопоы и флопов.

Собрав все аннотации для операций, мы получаем:

$$\text{мемопы} = 4N_c(N_m + 1) = 204 \text{ Ммемопов},$$

$$\text{флопы} = 2N_cN_m + N_c = 101 \text{ Мфлопов}.$$

Это вычислительное ядро лимитировано по пропускной способности, поэтому модель производительности составляет

$$PM = 4N_c(N_m + 1) \times 8/\text{Поток данных} = 122 \text{ мс}.$$

```
1: for all ячейки, C, вплоть до N_c do
2:   ρ_ave[C] ← 0.0                          # N_c сохраняет (ρ_ave)
3: end for
4: for all ИДы материалов, m, вплоть до N_m do
5:   for all ячейки, C, вплоть до N_c do
6:     ρ_ave[C] ← ρ_ave[C] + ρ[m][C]*V_f[m][C]   # N_cN_m сохраняет (ρ_ave)
                                                 # 3N_cN_m загружает (ρ_ave, ρ, V_f)
7:   end for                                     # 2N_cN_m флопов (+, *)
8: end for
9: for all ячейки, C, вплоть до N_c do
10:    ρ_ave[C] ← ρ_ave[C]/V[C]                 # N_c загружает/сохраняет (ρ_ave, V)
                                                # N_c флопов (l)
11: end for
```

Рис. 4.16 Материало-доминантный алгоритм для вычисления средней плотности ячеек с использованием полноматричного хранения

Производительность этого ядра составляет половину того, чего достигла ячеечно-центричная структура данных. Но это вычислительное ядро предпочитает ячеечно-центричную компоновку данных, а для расчета давления ситуация обратная.

4.3.2 Представление сжато-разреженного хранения

Теперь мы обсудим преимущества и пределы нескольких представлений сжатого хранения данных. Компоновки сжато-разреженного хранения данных очевидным образом экономят память, но дизайн как для ячеечно-, так и для материало-центричных компоновок требует некоторых размышлений.

Ячеечно-центричное сжато-разреженное хранение

Стандартный подход основан на связном списке материалов для каждой ячейки. Но связные списки, как правило, коротки и скачут по всей памяти. Решение состоит в размещении связного списка в сплошном массе и наличии ссылки, указывающей на начало элементов с материалами. В следующей ячейке сразу после этого будут появляться материалы. Таким образом, во время обычного обхода ячеек и материалов доступ к ним будет осуществляться в сплошном порядке. На рис. 4.17 показана ячеечно-центричная схема хранения данных. Значения для чистых ячеек хранятся в массивах состояний ячеек. На этом рисунке число 1.0 является объемной долей чистых ячеек, но это также могут быть значения чистых ячеек для плотности, температуры и давления. Второй массив – это число материалов в смешанной ячейке. Значение –1 указывает на то, что это чистая ячейка. Затем индекс связного списка материалов, *imaterial*, находится в третьем массиве. Если он меньше 1, то абсолютным значением элемента является индекс в массивах хранения смешанных данных. Если он равен 1 или больше, то он является индексом в сжатых массивах чистых ячеек.

Массивы хранения смешанных данных в сущности представляют собой связный список, имплементированный в стандартном массиве,

в результате чего данные являются сплошными в целях хорошей производительности кеша. Смешанные данные начинаются с массива, именуемого `nextfrac`, который указывает на следующий материал для этой ячейки. Это позволяет добавлять новые материалы в ячейку путем их помещения в конец массива. На рис. 4.17 это показано в списке смешанных материалов для ячейки 4, где стрелка показывает третий материал, который будет добавлен в конец. Массив `frac2cell` представляет собой обратное отображение на ячейку, содержащую материал. Третий массив, `material`, содержит номер материала для элемента. Это массивы обеспечивают навигацию по сжато-разреженной структуре данных. Четвертый массив представляет собой набор массивов состояний для каждого материала в каждой ячейке с объемной долей (V_f), плотностью (ρ), температурой (t) и давлением (p).

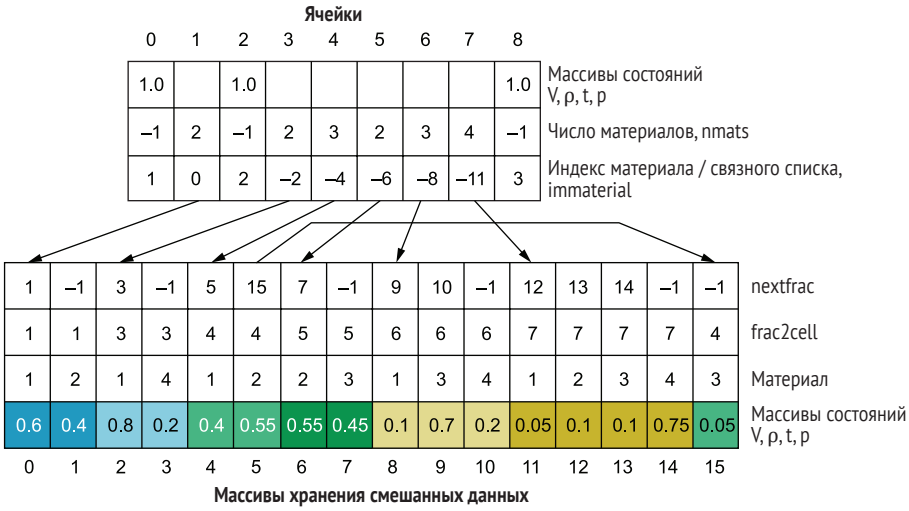


Рис. 4.17 В массивах смешанных материалов для ячеечно-центричной структуры данных используется связный список, имплементированный в сплошном массиве. Разное затенение внизу указывает на материалы, которые принадлежат конкретной ячейке, и соответствует затенению, используемому на рис. 4.13

Массивы смешанных материалов поддерживают дополнительную память в конце массива для быстрого добавления новых записей материалов на лету. Устранение ссылки на данные и установка ее равной нулю приводит к удалению материалов. В целях повышения производительности кеша массивы периодически переупорядочиваются обратно в сплошную память.

На рис. 4.18 показан алгоритм расчета средней плотности по каждой ячейке для сжато-разреженной компоновки данных. Сначала мы извлекаем индекс материала, `imaterial`, чтобы увидеть, что она является ячейкой со смешанными материалами, проверяя его на равенство нулю или меньше. Если он является чистой ячейкой, то мы ничего не делаем, потому что у нас уже есть плотность в массиве ячеек. Если он является ячейкой

со смешанным материалом, то мы входим в цикл, чтобы просуммировать плотность, умноженную на объемную долю по каждому материалу. Мы делаем проверку конечного условия индекса на отрицательное значение и используем массив `nextfrac` для получения следующей записи. Добравшись до конца списка, мы вычисляем плотность (ρ) ячейки. Справа от строк кода находятся аннотации к операционным расходам.

```
1: for all ячейки, C, вплоть до N_c do
2:   ave ← 0.0
3:   ix ← imaterial[C]                    # N_c загружает (imaterial)
4:   if ix <= 0 then
5:       for ix ← -ix, Unit ix < 0 do     # L_ρ накладные расходы малого цикла
6:         ave ← ave + ρ[ix]*V_f[ix]      # 2M_L загружает (ρ, V_f)
                                          # M_L флопов (+, * )
7:           ix ← nextfrac[ix]            # M_L загружает (nextfrac)
8:       end for
9:       ρ[C] ← ave/V[C]                  # M_f N_c сохраняет (ρ_ave)
                                          # M_f загружает (V)
                                          # M_f N_c флопов (l)
10:    end if
11: end for
```

Рис. 4.18 Ячеечно-доминантный алгоритм для вычисления средней плотности ячеек с использованием компактного хранения

Справа от строк кода находятся аннотации к операционным расходам. В данном анализе у нас будут четырехбайтовые целочисленные загрузки, и поэтому мы конвертируем *мемопы* в *мембайты*. Собрав подсчеты, мы получаем:

$$\text{мембайты} = (4 + 2M_f \times 8)N_c + (2 \times 8 + 4) = 6.74 \text{ Мбайт};$$

$$\text{флопы} = 2M_L + M_f N_c = .24 \text{ Мфлопов};$$

$$M_f = .04895;\ M_L = 97970.$$

Опять же, этот алгоритм лимитирован по пропускной способности памяти. Оценочное время выполнения модели производительности на 98 % меньше по сравнению с полной ячеечно-центричной матрицей.

$$PM = \text{мембайты}/\text{Поток данных} + L_p M_f N_c = .87 \text{ мс};$$

$$L_p = 20/2.7e6;\ M_f = .04895.$$

Материало-центричное сжато-разреженное хранение

Материало-центричная сжато-разреженная структура данных подразделяет все на отдельные материалы. Возвращаясь к малой тестовой задаче из рис. 4.9, мы видим, что существует шесть ячеек с материалом 1: 0, 1, 3, 4, 6 и 7 (на рис. 4.19 показано в подмножестве 1). В подмножестве есть два отображения: одно из сетки в подмножество, `mesh2subset`, и одно из подмножества обратно в сетку, `subset2mesh`. Список в подмножестве для сетки содержит индексы шести ячеек. Массив сетки содержит

–1 для каждой ячейки, в которой нет материала, и пронумеровывает те, которые последовательно отображаются в подмножество. Массив `nmats` в верхней части рис. 4.19 содержит число материалов, содержащихся в каждой ячейке. Массивы объемной доли (V_f) и плотности (ρ) в правой части рисунка имеют значения для каждой ячейки в этом материале. Номенклатурой C для этого будет `Vf[imat][icell]` и `p[imat][icell]`.

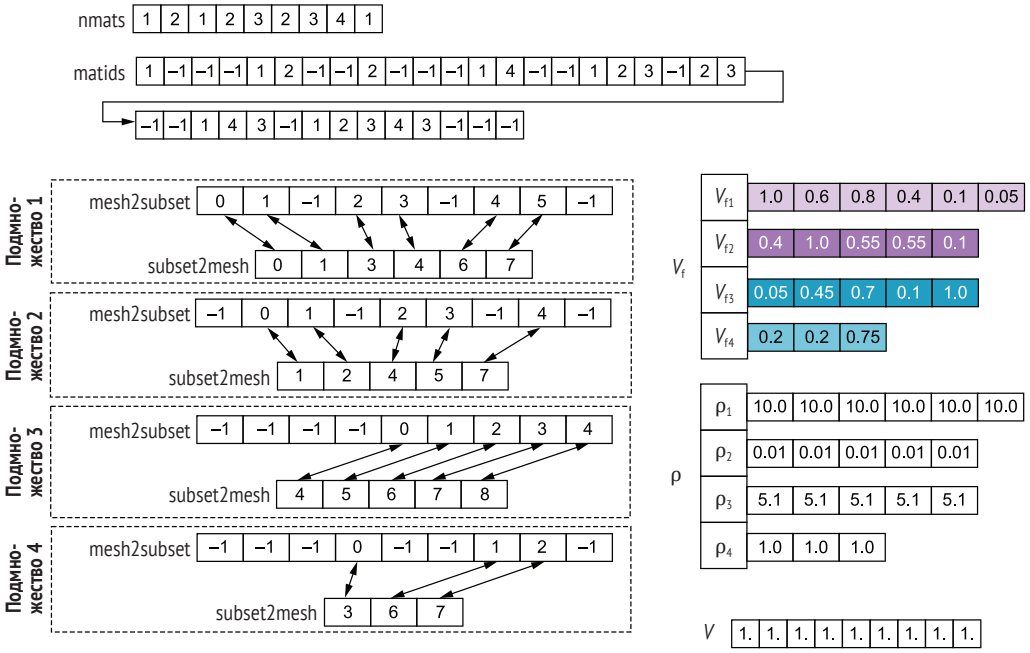


Рис. 4.19 Материало-центричная компоновка сжато-разреженных данных организована вокруг материалов. Для каждого материала существует массив переменной длины со списком ячеек, содержащих материал. Затенение соответствует затенению на рис. 4.15. На рисунке показана связь между полной сеткой и подмножествами, а также переменные объемной доли и плотности для каждого подмножества

Поскольку существует относительно немного материалов с длинными списками ячеек, мы можем использовать обычные выделения памяти в виде двухмерных массивов, а не заставлять их быть сплошными. В целях оперирования этой структурой данных мы работаем с каждым подмножеством материалов, главным образом, в последовательном порядке.

Материало-доминантный алгоритм на рис.4.20 для сжато-разреженного алгоритма выглядит так же, как алгоритм на рис. 4.13, с добавлением извлечения указателей в строках 5, 6 и 8. Но загрузки и флопы во внутреннем цикле выполняются только для подмножества материалов сетки, а не для всей сетки. За счет этого обеспечивается значительная экономии на флопах и мемопах. Собрав все подсчеты, мы получаем:

$$\text{мембайты} = 5 \times 8 \times F_f N_m N_c + 4 \times 8 \times N_c + (8 + 4) \times N_m = 74 \text{ Мбайта};$$

$$\text{флопы} = (2F_f H_m + 1)N_c = 3.1 \text{ Мфлопа/с}.$$

```
1: for all ячейки, C, вплоть до N_c do
2:   ρ_ave[C] ← 0.0                          # N_c сохраняет
3: end for
4: for all ИДы материалов, m, вплоть до N_m do
5:   ncmat ← ncellsmat[m]                    # N_m загружает (ncellsmat)
6:   Subset ← Subset2mesh[m]                 # N_m загружает (subset2mesh)
7:   for all ячейки, c, вплоть до ncmat do
8:     C ← subset[m]                         # F_f N_c N_m загружает (subset)
9:     ρ_ave[C] ← ρ_ave[C] + ρ[m][c] * V_f[m][c]
                                             # 3F_f N_c N_m загружает (ρ_ave, ρ, V_f)
                                             # F_f N_c N_m сохраняет (ρ_ave)
                                             # 2F_f N_c N_m флопов (+, * )
10:  end for
11: end for
12: for all ячейки, C, вплоть до N_c do
13:   ρ_ave[C] ← ρ_ave[C]/V[C]               # 2N_c загружает (ρ_ave, V)
                                             # N_c сохраняет (ρ_ave)
                                             # N_c флопов (l)
14: end for
```

Рис. 4.20 Материало-доминантный алгоритм вычисляет среднюю плотность ячеек, используя схему материало-центричного сжатого хранения

Эта модель производительности показывает более чем 95%-ное сокращение оценочного времени выполнения по сравнению с материало-центричной полноматричной структурой данных:

$$PM = \text{мембайты/Поток данных} = 5.5 \text{ мс}.$$

В табл. 4.1 резюмированы результаты этих четырех структур данных. Разница между оценочным и измеренным временем выполнения удивительно мала. Она показывает, что даже грубые подсчеты загрузок памяти могут быть хорошим предсказателем производительности.

Таблица 4.1 Разреженные структуры данных работают быстрее и используют меньше памяти, чем полные двухмерные матрицы

	Загрузка памяти (Мб)	Флопы	Оценочное время выполнения	Измеренное время выполнения
Ячеечно-центричное полное	424	3.1	67.2	108
Материало-центричное полное	1632	101	122	164
Ячеечно-центричное сжато-разреженное	6.74	.24	.87	1.4
Материало-центричное сжато-разреженное	74	3.1	5.5	9.6

Преимущество сжато-разреженных представлений заключается в значительной экономии как памяти, так и производительности. Поскольку ядро, которое мы проанализировали, больше подходило для ячеечно-центричных структур данных, ячеечно-центричная сжато-разреженная структура данных явно является лучшим исполнителем как по памяти, так и по времени выполнения. Если мы посмотрим на другое ядро, которое показывает материало-центричную структуру данных, результаты будут немного в пользу материало-центричных структур данных. Но главный вывод заключается в том, что любое из сжато-разрежен-

ных представлений является значительным улучшением по сравнению с полноматричными представлениями.

Приведенное выше тематическое исследование было сосредоточено на мультиматериальном представлении данных. Вместе с тем существует много разнообразных приложений с разреженными данными, которые способны извлекать выгоду из добавления сжато-разреженной структуры данных. Быстрый анализ производительности, подобный тому, который был проведен в этом разделе, может определять, стоят ли указанные выгоды дополнительных усилий в этих приложениях.

4.4 Продвинутые модели производительности

Существуют более продвинутые модели производительности, которые более точно отражают аспекты компьютерного оборудования. Мы кратко рассмотрим эти продвинутые модели, чтобы понять, что конкретно они предлагают, и возможные уроки, которые необходимо извлечь. Детали анализа производительности не так важны, как выводы.

В этой главе мы в первую очередь сосредоточились на вычислительных ядрах, лимитированных по пропускной способности, поскольку они представляют пределы производительности большинства приложений. Мы подсчитали байты, загружаемые и сохраняемые ядром, и оценили время, необходимое для этого перемещения данных, на основе приложения сравнительного тестирования потоков данных или модели контура крыши (глава 3). К настоящему времени вы должны понимать, что единицей операции компьютерного оборудования на самом деле являются не байты или слова, а строки кеша, и мы можем улучшать модели производительности, подсчитывая строки кеша, которые необходимо загружать и сохранять. В то же время мы можем оценивать объем используемого кеша.

Приложение сравнительного тестирования потоков данных фактически состоит из четырех отдельных вычислительных ядер: ядер копирования, масштабирования, сложения и триады. В чем же тогда причина разницы между этими ядрами в пропускной способности (16 156.6–22 086.5 Мб/с), как показано в упражнении с приложением STREAM Benchmark в разделе 3.2.4? Тогда подразумевалось, что причиной была разница в арифметической интенсивности между ядрами, показанными в таблице в разделе 3.2.4. Это верно лишь отчасти. Малая разница в арифметических операциях на самом деле оказывает довольно незначительное влияние, пока мы находимся в режиме, лимитированном по пропускной способности. Корреляция с арифметическими операциями тоже невелика. Почему операция масштабирования имеет самую низкую пропускную способность? Настоящими виновниками являются детали в иерархии кеша в системе. Система кеша не похожа на трубу, по которой равномерно течет вода, как это может следовать из приложения сравнительного тестирования потоков данных. Она больше похожа

на ведерную бригаду, которая переправляет данные вверх по уровням кеша с варьирующимися числами пакетов и размерами, как показано на рис. 4.21. Это именно то, что пытается уловить модель исполнения кеш-памяти, Execution Cache Memory (ECM), разработанная Трейбигом и Хагером. Хотя она требует знания аппаратной архитектуры, она способна очень хорошо предсказывать производительность для потоковых вычислительных ядер. Перемещение между уровнями бывает лимитировано числом операций (μопами), именуемых микроопами, которые могут выполняться за один цикл. Модель ECM работает в терминах строк кеша и циклов, моделируя перемещение между разными уровнями кеша.

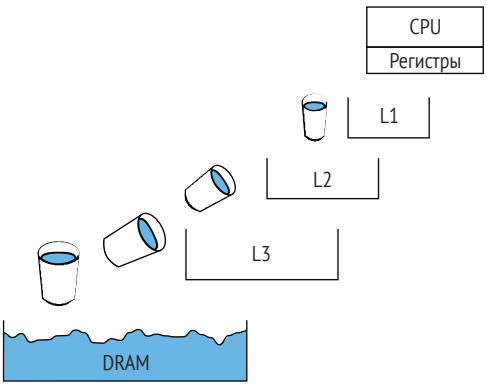


Рис. 4.21 Перемещение данных между уровнями кеша представляет собой серию дискретных операций, больше похожих на ведерную бригаду, чем на поток воды по трубе. Подробные сведения об оборудовании и о том, сколько нагрузок может выдаваться на каждом уровне и в каждом направлении, в значительной степени влияют на эффективность загрузки данных через иерархию кеша

Давайте просто быстро взглянем на модель ECM для потоковой триады в форме (`A[i] = B[i] + s*C[i]`), чтобы увидеть принцип работы этой модели (рис. 4.22). Этот расчет должен быть выполнен для конкретного ядра и оборудования. Для данного анализа мы будем использовать систему Haswell EP для оборудования. Мы начинаем в вычислительном ядре с уравнения $T_{core} = \max(T_{nOL}, T_{OL})$, где T – это время в циклах. T_{OL} – это, как правило, арифметические операции, которые перекрывают время передачи данных, и T_{nOL} – неперекрывающее время передачи данных.

Для потоковой триады у нас есть строка кеша операций умножения-сложения. Если это делается с помощью скалярной операции, то на ее завершение требуется 8 циклов. Но мы можем сделать это с помощью новых команд Advanced Vector Extensions (AVX, продвинутых векторных расширений). Чип Haswell имеет два 256-битных модуля векторной обработки AVX со слитным умножением-сложением (FMA). Каждый из этих модулей обрабатывает четыре значения двойной точности. В строке кеша содержится восемь значений, поэтому два модуля векторной обработки FMA AVX могут обработать это за один цикл. T_{nOL} – это время передачи данных. Нам нужно загружать строки кеша для *B* и *C*, и нам

нужно загружать и сохранять строку кеша для *A*. Это занимает 3 цикла для чипа Haswell из-за предела генераторов адресов (address generation units, AGUs).

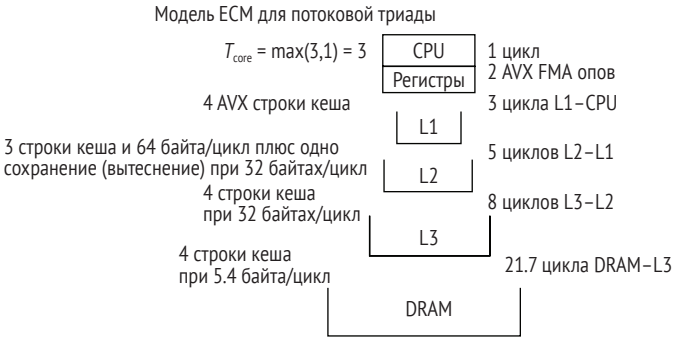


Рис. 4.22 Модель исполнения кеш-памяти Execution Cache Memory (ECM) для процессора Haswell предоставляет подробный хронометраж передачи данных для вычисления потоковой триады между уровнями кеша. Если данные находятся в основной памяти, то время, необходимое для передачи данных в центральный процессор, равно сумме времени передачи между каждым уровнем кеша или 21.7 + 8 + 5 + 3 = 37.7 циклов. Операции с плавающей точкой занимают всего три цикла, поэтому загрузки памяти являются лимитирующим аспектом для потоковой триады

Перемещение четырех строк кеша из L2 в L1 со скоростью 64 байта/цикл занимает 4 цикла. Но использование `A[i]` представляет собой операцию сохранения. Сохранение обычно требует специальной загрузки, именуемой *выделением для записи* (write-allocate), когда пространство памяти выделяется в менеджере виртуальных данных, а строка кеша создается на необходимых уровнях кеша. Затем данные модифицируются и *вытесняются* (сохраняются) из кеша. Это может работать только со скоростью 32 байта/цикл на этом уровне кеша, что приводит к дополнительному циклу или в общей сложности 5 циклам. Передача данных из уровней L3–L2 составляет 32 байта/цикл, поэтому требуется 8 циклов. И наконец, используя измеренную пропускную способность 27.1 Гб/с, число циклов для перемещения строк кеша из основной памяти составляет около 21.7 циклов. В ECM для подытоживания указанных чисел используется приведенная ниже специальная нотация:

$$\{T_{\mathrm{OL}} \,||\, T_{\mathrm{nOL}} \mid T_{\mathrm{L1L2}} \mid T_{\mathrm{L2L3}} \mid T_{\mathrm{L3Mem}}\} = \{1 \,||\, 3 \mid 5 \mid 8 \mid 21.7\} \text{ циклов.}$$

Значение T_{core} показано в обозначении $T_{\mathrm{OL}} \,||\, T_{\mathrm{nOL}}$. Это, по сути, времена (в циклах) для перемещения между каждым уровнем с особым случаем для T_{core}, когда некоторые операции на вычислительном ядре могут перекрывать некоторые операции передачи данных из L1 в регистры. Затем модель предсказывает число циклов, которое потребуется для загрузки из каждого уровня кеша путем суммирования времени передачи данных, включая неперекрывающие передачи данных из L1 в регистры.

Максимальное значение T_{OL} и время передачи данных затем используются в качестве предсказываемого времени:

$$T_{ECM} - \max(T_{nOL} + T_{data}, T_{OL}).$$

Эта специальная нотация ECM показывает результирующее предсказание для каждого уровня кеша:

$$\{3 \rceil 8 \rceil 16 \rceil 37.7\} \text{ циклов.}$$

Указанная нотация говорит о том, что ядру требуется 3 цикла, когда оно оперирует из кеша L1, 8 – из кеша L2, 16 – из L3 и 37.7 циклов, когда данные должны быть извлечены из основной памяти.

Из этого примера можно извлечь следующий урок: наткнувшись на предел дискретного оборудования на конкретном чипе с конкретным вычислительным ядром, можно вынужденно получить еще один-два цикла на одном из уровней кеша, что приводит к снижению производительности. Немного другая версия процессора, возможно, не будет иметь такой же проблемы. Например, более поздние версии чипов Intel добавляют еще один AGU, который изменяет циклы регистров L1 с 3 на 2.

Этот пример также демонстрирует, что модули векторной обработки имеют ценность как для арифметических операций, так и для перемещения данных. Векторная загрузка, также именуемая операцией квадрозагрузки (quad-load), не нова. Бо́льшая часть внимания в обсуждении векторных процессоров уделяется арифметическим операциям. Но для вычислительных ядер, лимитированных по пропускной способности, вероятно, векторные операции с памятью важнее. Анализ, проведенный Стенгелем и соавт. с использованием модели ECM, показывает, что векторные команды AVX могут обеспечивать двукратное повышение производительности, по сравнению с циклами, наивно планируемыми компилятором. Возможно, это связано с тем, что компилятор не располагает достаточной информацией. В более поздних модулях векторной обработки также имплементируется операция загрузки памяти «*сбор/разброс*», при которой данные, загруженные в модуль векторной обработки, не обязательно должны находиться в ячейках памяти в сплошном порядке (сбор), а сохранение из вектора в память не обязательно должно делаться в ячейки памяти в сплошном порядке (разброс).

> **ПРИМЕЧАНИЕ** Указанная новая функциональность загрузки памяти «сбор/разброс» приветствуется, так как она необходима для хорошей работы многих реальных кодов численной симуляции. Но при имплементации сбора/разброса в текущем виде все еще существуют проблемы с производительностью, и необходимы дополнительные ее усовершенствования.

Мы также можем проанализировать производительность иерархии кеша с помощью *потокового хранения*. Потоковое хранение (streaming

store) обходит систему кеша и записывает данные непосредственно в основную память. В большинстве компиляторов есть возможность использовать операции потокового хранения, и некоторые используют ее в качестве самостоятельной оптимизации. Ее эффект заключается в уменьшении числа строк кеша, перемещаемых между уровнями иерархии кеша, уменьшении скученности и более медленной операции вытеснения между уровнями кеша. Теперь, когда вы увидели эффект перемещения строки кеша, вы должны оценить его ценность по достоинству.

Модель ECM используется несколькими исследователями для оценивания и оптимизации стенсильных ядер. Стенсильные ядра являются операциями потока данных и могут анализироваться с помощью этих методов. Становится немного запутанным отслеживать все строки кеша и характеристики оборудования без ошибок, поэтому помогают инструменты подсчета производительности. Мы приведем пару ссылок, перечисленных в приложении А, где можно получить по ним дополнительную информацию.

Продвинутые модели отлично подходят для понимания производительности относительно простых потоковых ядер. *Потоковые ядра* (streaming kernels) – это ядра, которые загружают данные почти оптимальным способом для эффективного использования иерархии кеша. Но ядра в научных приложениях и приложениях HPC часто усложняются за счет наличия условных переходов, неидеально вложенных циклов, редукции и привносимых циклами зависимостей. Кроме того, компиляторы могут преобразовывать операции языка высокого уровня в ассемблерные операции в неожиданном ключе, что усложняет анализ. Обычно также приходится иметь дело с большим числом ядер и циклов. Эти сложные ядра невозможно анализировать без специализированных инструментов, поэтому мы пытаемся разрабатывать общие идеи из простых ядер, которые мы можем применять к более сложным.

4.5 Сетевые сообщения

Наши модели передачи данных можно расширить для использования при анализе компьютерной сети. Простая модель производительности сети между узлами кластера или системы HPC такова:

Время (мс) = задержка (µсек) + перемещенные_байты (Мбайт) / пропускная способность (Гб/с) (с пересчетом единиц измерения).

Обратите внимание, что это пропускная способность сети, а не используемой нами памяти. По адресу на http://icl.cs.utk.edu/hpcc/hpcc_results_lat_band.cgi находится веб-сайт сравнительного тестирования HPC для определения задержки и пропускной способности.

Мы можем применять сетевые сравнительные микротесты с веб-сайта сравнительного тестирования HPC, чтобы получать типичные задержку

и пропускную способность. Мы будем использовать 5 μсек для задержки и 1 Гб/с для пропускной способности. В результате получим график, показанный на рис. 4.23. В случае более крупных сообщений мы можем использовать примерно 1 с на каждый переданный мегабайт. Но подавляющее большинство сообщений невелики. Мы рассмотрим два разных коммуникационных примера, сначала более крупное сообщение, а затем меньшее, чтобы понять важность задержки и пропускной способности в каждом из них.

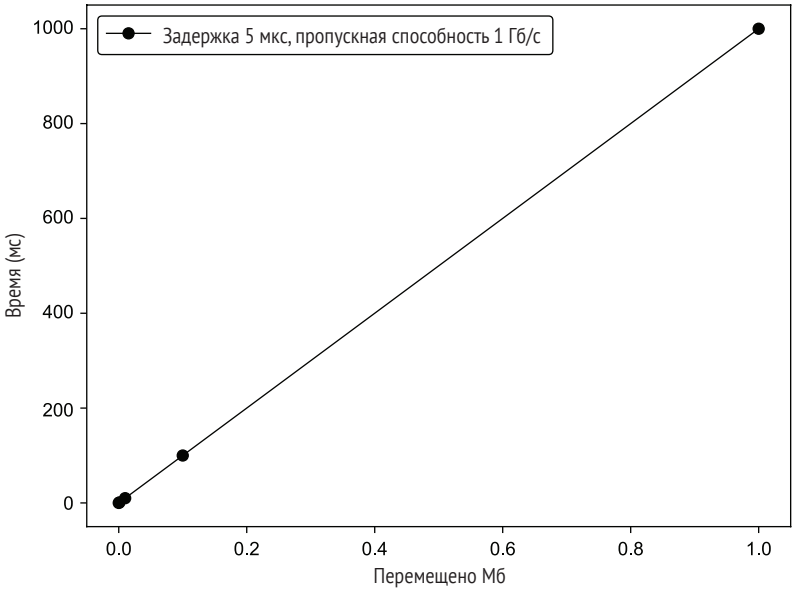


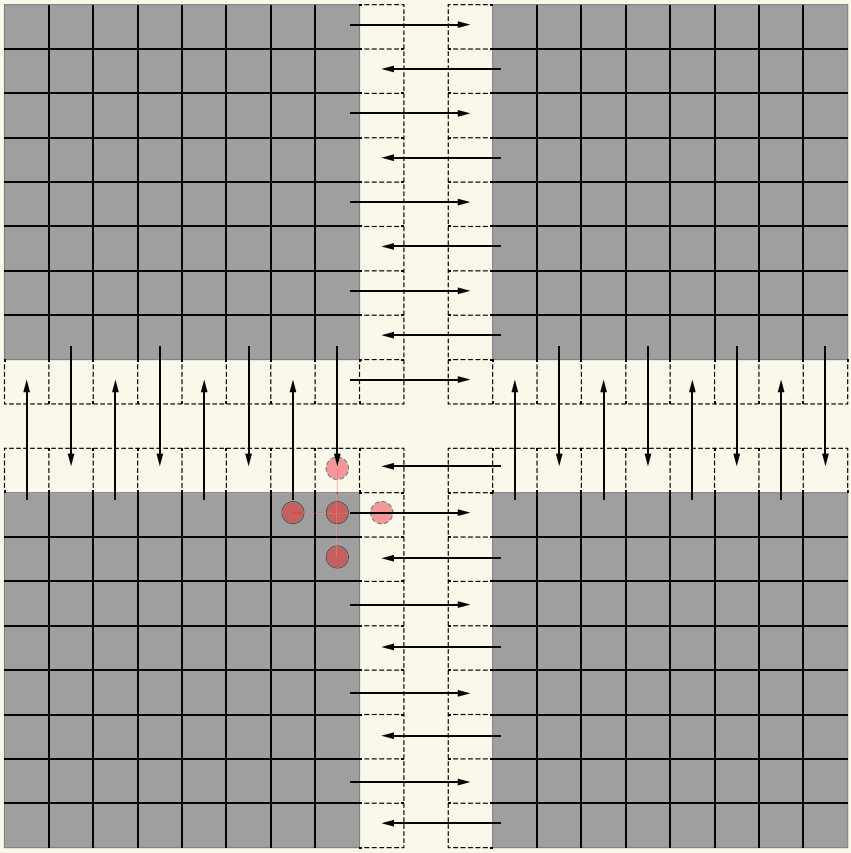
Рис. 4.23 Типичное время передачи по сети в зависимости от размера сообщения дает нам эмпирическое правило: 1 Мб занимает 1 с (секунду), 1 Кб занимает 1 мс (миллисекунду) или 1 байт занимает 1 мкс

Пример: обмен данными через призрачные ячейки

Давайте возьмем сетку 1000×1000. Нам нужно передавать внешние ячейки нашего процессора, как показано на следующем ниже рисунке, смежному процессору, чтобы он мог завершать свои вычисления. Дополнительные ячейки, размещенные на внешней стороне от сетки для процессора, называются *призрачными ячейками*.

1000 элементов во внешних ячейках × 8 байт = 8 Кб.

Время связи = 5 с + 8 мс.



Данные внешней ячейки обмениваются со смежными процессорами, вследствие чего стенсильный расчет выполняется на основе текущих значений. Пунктирные ячейки называются призрачными, потому что в них хранятся дублированные данные из другого процессора. Для ясности стрелки показывают обмен данными только для каждой второй ячейки

Пример: суммарное число ячеек в процессорах

Нам нужно передать число ячеек смежному процессору для суммирования, а затем вернуть сумму. Имеется два сообщения в размере четырехбайтового целого числа.

$$\text{Время связи} = (5\ \text{с} + 4\ \text{с}) \times 2 = 18\ \mu\text{сек}.$$

В этом случае задержка значительно влияет на совокупное время, затрачиваемое на передачу сообщения.

Последний пример с суммой на жаргоне информатики называется *операцией редукции*. Массив количеств ячеек в процессорах сводится к одному значению. В более общем случае операция редукции – это любая операция, при которой многомерный массив от одной до двух размерностей сводится к массиву по меньшей мере на одну размерность меньше и нередко к скалярному значению. Эти операции в параллельных вычислениях широко распространены, и для их выполнения требуется сотрудничество между процессорами. Кроме того, редукционная сумма в последнем примере может выполняться попарно в древовидном шаблоне с числом коммуникационных переходов, равным $\log_2 N$, где N – это число рангов (процессоров). Когда число процессоров достигает тысячи, время выполнения операции увеличивается. Возможно, и это еще важнее, во время операции все процессоры должны синхронизироваться, что приводит к тому, что многие из этих процессоров ожидают, пока другие процессоры перейдут к вызову редукции.

Есть и более сложные модели сетевых сообщений, которые могут быть полезны для конкретного сетевого оборудования. Но детали сетевого оборудования различаются настолько, что они, скорее всего, не прольют света на общее поведение всех возможных аппаратных средств.

4.6 Материалы для дальнейшего изучения

Ниже приведено несколько ресурсов для проведения разведывательного анализа тем этой главы, включая дизайн с ориентацией на данные, структуры данных и модели производительности. Большинство разработчиков приложений находит интересными дополнительные материалы по дизайну с ориентацией на данные. Во многих приложениях может применяться разреженность, и мы можем узнать, как это делается, на примере сжато-разреженных структур данных.

4.6.1 Дополнительное чтение

Следующие ниже две ссылки дают хорошее описание подхода к дизайну с ориентацией на данные, разработанном в игровом сообществе для повышения производительности при разработке программ. Вторая ссылка также указывает местоположение видео презентации Эктона на CppCon.

- Ноэль Ллопис, «Дизайн с ориентацией на данные (или Почему вы, возможно, стреляете себе в ногу, используя ООП)» (Noel Llopis, Data-oriented design (or why you might be shooting yourself in the foot with OOP), декабрь 2009). По состоянию на 21 февраля 2021 года, по адресу http://gamesfromwithin.com/data-oriented-design.
- Майк Эктон и Игры с бессонницей, «Дизайн с ориентацией данные и C++» (Mike Acton and Insomniac Games, Data-oriented design and C++). Презентация на CppCon (сентябрь 2014 г.):

- Powerpoint по адресу https://github.com/CppCon/CppCon2014;
- видео по адресу https://www.youtube.com/watch?v=rX0ItVEyjHc.

Следующая ниже ссылка полезна для более подробного изучения тематических примеров сжато-разреженных структур данных с использованием простых моделей производительности. Вы также найдете измеренные результаты производительности на мультиядерных процессорах и GPU:

- Шейн Фогерти, Мэтт Мартино и соавт., «Сравнительное исследование мультиматериальных структур данных для приложений вычислительной физики». Shane Fogerty, Matt Martineau, et al., A comparative study of multi-material data structures for computational physics applications. In *Computers & Mathematics with Applications* Vol. 78, no. 2 (July, 2019): 565–581. Исходный код доступен по адресу https://github.com/LANL/MultiMatTest.

В следующей ниже статье представлена сокращенная нотация, используемая для модели кеша исполнения, Execution Cache Model:

- Хольгер Штенгель, Ян Трейбиг и соавт., «Количественное оценивание узких мест производительности стенсильных вычислений с использованием модели кеша исполнения» (Holger Stengel, Jan Treibig, et al., Quantifying performance bottlenecks of stencil computations using the execution-cache-memory model). В In *Proceedings of the 29th ACM on International Conference on Supercomputing* (ACM, 2015): 207–216.

4.6.2 Упражнения

1. Напишите двухмерный выделитель сплошной памяти под левую нижнюю треугольную матрицу.
2. Напишите двухмерный выделитель на C, который компонует память так же, как Fortran.
3. Создайте макрокоманду для массива структур из массивов (AoSoA) для цветовой модели RGB из раздела 4.1.
4. Модифицируйте код для ячеечно-центричной полноматричной структуры данных, чтобы не использовать условный переход, и оцените ее производительность.
5. Каким образом модуль векторной обработки AVX-512 изменил бы модель ECM для потоковой триады?

Резюме

- Структуры данных лежат в основе дизайна приложений и часто определяют производительность и, как следствие, имплементацию параллельного кода. И разработка хорошего дизайна для компоновки данных заслуживает того, чтобы приложить немного дополнительных усилий.

- Вы можете использовать концепции дизайна с ориентацией на данные для разработки более высокопроизводительных приложений.
- Существуют способы написания выделителей сплошной памяти под многомерные массивы или особые ситуации в целях минимизирования использования памяти и повышения производительности.
- Вы можете использовать структуры сжатого хранения с целью сокращения использования памяти в вашем приложении, а также повышения его производительности.
- Простые модели производительности, основанные на подсчете загрузок и сохранений, могут предсказывать производительность многих базовых вычислительных ядер.
- Более сложные модели производительности проливают свет на производительность иерархии кеша в отношении деталей низкого уровня в аппаратной архитектуре.

5 Параллельные алгоритмы и шаблоны

Эта глава охватывает следующие ниже темы:

- что такое параллельные алгоритмы и шаблоны и какова их важность;
- как сравнивать производительность разных алгоритмов;
- что отличает параллельные алгоритмы от других алгоритмов.

Алгоритмы лежат в основе вычислительной науки. Наряду с рассмотренными в предыдущей главе структурами данных алгоритмы составляют основу всех вычислительных приложений. По этой причине важно тщательно продумывать ключевые алгоритмы в своем коде. Для начала давайте дадим определение того, что мы подразумеваем под параллельными алгоритмами и параллельными шаблонами.

- *Параллельный алгоритм* – это четко определенная пошаговая вычислительная процедура, в которой при решении задачи делается акцент на конкурентности (concurrency). Примеры алгоритмов включают сортировку, поиск, оптимизацию и матричные операции.
- *Параллельный шаблон* – это конкурентный, разделяемый фрагмент кода, который встречается в самых разных приложениях с некоторой частотой. Сами по себе эти фрагменты кода, как правило, не

решают полных интересующих разработчиков задач. Примеры шаблонов включают редукции, префиксные сканы и обновления призрачных ячеек.

Мы покажем редукцию в разделе 5.7, префиксное сканирование в разделе 5.6 и обновления призрачных ячеек в разделе 8.4.2. В одном контексте параллельная процедура может рассматриваться как алгоритм, а в другом – как шаблон. Реальная разница заключается в том, какую цель она осуществляет, главную или просто часть более широкого контекста. Важно уметь распознавать шаблоны, «дружественные к параллелизму». Это необходимо для подготовки к последующим усилиям по параллелизации.

5.1 Анализ алгоритмов для приложений параллельных вычислений

Разработка параллельных алгоритмов является молодой областью. Даже терминология и методы анализа параллельных алгоритмов все еще застряли в последовательном мире. Одним из наиболее традиционных подходов к оцениванию алгоритмов является анализ их алгоритмической сложности. Наше определение алгоритмической сложности следует ниже.

ОПРЕДЕЛЕНИЕ *Алгоритмическая сложность* – это мера числа операций, которые потребовались бы для завершения алгоритма. Алгоритмическая сложность является свойством алгоритма и является мерой объема работы или операций в процедуре.

Сложность обычно выражается в асимптотической нотации. Асимптотическая нотация – это тип выражения, который определяет предельные границы производительности. По сути, указанная нотация определяет, растет ли время выполнения линейно или же оно ускоряется с увеличением размера задачи. В указанной нотации используются различные формы буквы O, такие как $O(N)$, $O(N\log N)$ или $O(N^2)$. N – это размер длинного массива, такой как число ячеек, частиц или элементов. Комбинация $O()$ и N указывает на то, как стоимость алгоритма масштабируется по мере увеличения размера N массива. O можно рассматривать как «порядок», как в словосочетании «масштаб порядка». Как правило, простой цикл по N элементам будет иметь сложность $O(N)$, двойной вложенный цикл будет иметь сложность $O(N^2)$, а алгоритм на основе дерева будет иметь сложность $O(N\log N)$. По соглашению ведущие константы отбрасываются. Наиболее часто используемыми асимптотическими обозначениями являются:

- большая O – это предел производительности алгоритма в наихудшем случае. Примерами являются дважды вложенный цикл `for` для крупного массива размера N, который будет иметь сложность $O(N^2)$;

- большая Ω (большая омега) – предел производительности алгоритма для наилучшего случая;
- большая Θ (большая тета) – производительность алгоритма для среднего случая.

В традиционном анализе алгоритмов термины «алгоритмическая сложность», «вычислительная сложность» и «временная сложность» используются взаимозаменяемо. Мы дадим определения указанных терминов немного по-другому, чтобы способствовать оцениванию алгоритмов на современном оборудовании для параллельных вычислений. Время не зависит от объема работы, равно как и вычислительные усилия или затраты. Таким образом, мы вносим следующие коррективы в определения терминов вычислительной сложности и временной сложности:

- *вычислительная сложность (также именуемая шаговой сложностью)* – это число шагов, необходимых для завершения алгоритма. Указанная мера сложности является атрибутом имплементации и типа используемого для расчета оборудования. Она включает в себя возможный объем параллелизма. Если вы используете векторный или мультиядерный компьютер, то шаг (цикл) может состоять из четырех или более операций с плавающей точкой. Сможете ли вы использовать эти дополнительные операции для редукции числа шагов?
- *временная сложность принимает в расчет фактическую стоимость операции в типичной современной вычислительной системе.* Самая большая поправка в отношении времени заключается в учете затрат на загрузки памяти и кеширование данных.

Мы будем использовать анализ сложности для сравнений некоторых алгоритмов, таких как алгоритмы префиксного суммирования из раздела 5.5. Но для исследователей прикладной информатики асимптотическая сложность алгоритма является несколько одномерной и имеет ограниченное применение. Она говорит нам только о стоимости алгоритма в пределе по мере его увеличения. В прикладной обстановке нам нужна более полная модель алгоритма. Мы увидим причину в следующем далее разделе.

5.2 Модели производительности против алгоритмической сложности

Мы впервые представили модели производительности в главе 4 для анализа относительной производительности разных структур данных. В модели производительности мы строим гораздо более полное описание производительности алгоритма, чем в анализе алгоритмической сложности. Самая большая разница состоит в том, что *мы не скрываем константный фактор перед алгоритмом*. Но для масштабирования есть разница и в членах, таких как $\log N$. Фактический подсчет операций делается из двоичного дерева и должен писаться в $\log_2 N$.

В традиционном алгоритмическом анализе сложности разница между двумя логарифмическими членами является константой, и она поглощается константным фактором. В обычных мировых задачах эти константы бывают немаловажны и не сокращаются (не гасятся); поэтому нам необходимо использовать модель производительности, чтобы различать разные подходы к аналогичному алгоритму. В целях более глубокого понимания преимуществ использования моделей производительности давайте начнем с примера из повседневной жизни.

Пример

Вы являетесь одним из организаторов конференции со 100 участниками. Вы хотите раздать регистрационные пакеты документов каждому участнику. Вот несколько алгоритмов, которые вы можете использовать.

1. Попросить всех 100 участников выстроиться в очередь, пока вы просматриваете 100 папок, чтобы найти пакет, который нужно передать каждому участнику по очереди. В худшем случае вам придется просмотреть все 100 папок. В среднем вы просмотрите 50. Если пакета нет, то вы должны просмотреть все 100.
2. Предварительно отсортировать пакеты в алфавитном порядке перед регистрацией. Теперь вы можете использовать бисекционный поиск, чтобы отыскивать каждую папку.

Возвращаясь к первому алгоритму в примере, давайте для простоты будем считать, что папки остаются после вручения пакетов документов участнику, вследствие чего число папок остается неизменным в изначальном числе. Имеется N участников и N папок, тем самым создается двойной вложенный цикл. Вычисление имеет порядок операций N^2, или $O(N^2)$ в асимптотической нотации большой O для наихудшего случая. Если бы папки каждый раз уменьшались, то вычисление составило бы $(N + N - 1 + N - 2 \ldots)$ или было бы алгоритмом с $O(N^2)$. Второй алгоритм может задействовать отсортированный порядок папок с бисекционным поиском, чтобы выполнять алгоритм в наихудшем случае за $O(N \log N)$ операций.

Асимптотическая сложность говорит о производительности алгоритма, когда мы достигаем крупных размеров, таких как миллион участников. Но у нас никогда не будет миллиона участников. У нас будет конечный размер в 100 участников. Для конечных размеров необходима более полная картина алгоритмической производительности.

В целях иллюстрации мы применим модель производительности к одному из самых простых компьютерных алгоритмов, чтобы увидеть, как он может приносить более глубокую информацию. Мы будем использовать модель, основанную на времени, в которой мы учитываем реальные затраты оборудования, а не подсчет на основе операций. В этом примере мы рассмотрим бисекционный поиск, или поиск делением пополам, также именуемый двоичным поиском. Это один из наиболее распростра-

ненных компьютерных алгоритмов и методов численной оптимизации. Обычный асимптотический анализ говорит о том, что двоичный поиск выполняется намного быстрее, чем линейный поиск. Мы покажем, что с учетом особенностей функционирования реальных компьютеров увеличение скорости не так велико, как можно было бы ожидать. Этот анализ также поможет объяснить результаты поиска в таблице из раздела 5.5.1.

Пример: бисекционный поиск против линейного поиска

Алгоритм бисекционного поиска может использоваться для отыскания правильной записи в отсортированном массиве из 256 целочисленных элементов. Этот алгоритм берет срединную точку, рекурсивно деля пополам оставшийся возможный диапазон. В модели производительности бисекционный поиск будет иметь $\log_2 256$ шагов или 8, в то время как линейный поиск будет иметь наихудший случай из 256 шагов и в среднем 128. Если подсчитать загрузки строк кеша для четырехбайтового целочисленного массива, то линейный поиск будет в худшем случае равняться только 16 загрузкам строк кеша и в среднем 8. Двоичный поиск потребует четырех загрузок строк кеша и около четырех в среднем случае. Необходимо подчеркнуть, что этот линейный поиск будет только в два раза медленнее, чем двоичный поиск, а не в 16 раз медленнее, чем мы могли бы ожидать.

В этом анализе мы исходим из того, что любая операция с данными в кеше, по существу, бесплатна (фактически требует пару циклов), в то время как загрузка строки кеша составляет порядка 100 циклов. Мы просто подсчитываем загрузки строк кеша и игнорируем операции сравнения. Для нашей модели производительности на основе времени мы бы сказали, что стоимость линейного поиска равна $(n/16)/2 = 8$, а бисекционный поиск равен $\log_2(n/16) = 4$.

Характер поведения кеша значительно изменяет результат для этих коротких массивов. Бисекционный поиск по-прежнему быстрее, но не настолько, насколько можно было бы ожидать при более простом анализе.

Хотя асимптотическая сложность используется для понимания производительности как масштаб алгоритма, она не дает уравнения для абсолютной производительности. Для данной задачи линейный поиск, который масштабируется линейно, может превосходить бисекционный поиск, который масштабируется логарифмически. Это особенно верно, когда вы параллелизуете алгоритм, так как масштабировать линейно масштабируемый алгоритм гораздо проще, чем логарифмически масштабируемый алгоритм. В дополнение к этому компьютер по своему дизайну делает линейный обход массива и выбирает данные упреждающе, что может немного повысить производительность. Наконец, в конкретной задаче элемент может оказаться в начале массива, где производительность бисекционного поиска намного хуже, чем линейного поиска.

В качестве примера других параллельных соображений давайте взглянем на имплементацию поиска в 32 потоках процессора на мультиядер-

ном CPU или GPU. Набор потоков должен ждать завершения самого медленного из них во время каждой операции. Бисекционный поиск всегда занимает четыре загрузки кеша. Линейный поиск зависит от числа строк кеша, необходимых для каждого потока. Наихудший случай определяет количество времени, которое операция занимает, что делает стоимость ближе к 16 строкам кеша, чем в среднем 8 строк кеша.

Возникает естественный вопрос: как это работает на практике? Давайте рассмотрим два описанных в примере варианта кода поиска в таблице. Вы можете протестировать следующие ниже алгоритмы в своей системе, при этом код PerfectHash включен в прилагаемый исходный код этой главы по адресу https://github.com/EssentialsofParallelComputing/Chapter5. Прежде всего в следующем ниже листинге показана версия кода с алгоритмом линейного поиска в таблице.

Листинг 5.1 Алгоритм линейного поиска в таблице

PerfectHash/table.c

```
268 double *interpolate_bruteforce(int isize, int xstride,
       int d_axis_size, int t_axis_size, double *d_axis, double *t_axis,
269    double *dens_array, double *temp_array, double *data)
270 {
271    int i;
272
273    double *value_array=(double *)malloc(isize*sizeof(double));
274
275    for (i = 0; i<isize; i++){
276       int tt, dd;
277
276       int tt, dd;
277
278       for (tt=0; tt<t_axis_size-2 &&                    | Задает линейный поиск
             temp_array[i] > t_axis[tt+1]; tt++);          | с 0 по axis_sixe
279       for (dd=0; dd<d_axis_size-2 &&                    |
             dens_array[i] > d_axis[dd+1]; dd++);          |
280
281       double xf = (dens_array[i]-d_axis[dd])/           |
                      (d_axis[dd+1]-d_axis[dd]);          |
282       double yf = (temp_array[i]-t_axis[tt])/           |
                      (t_axis[tt+1]-t_axis[tt]);          |
283       value_array[i] =                                  |   Интерполяция
                    xf * yf *data(dd+1,tt+1)              |
284          + (1.0-xf)* yf *data(dd, tt+1)                |
285          + xf *(1.0-yf)*data(dd+1,tt)                  |
286          + (1.0-xf)*(1.0-yf)*data(dd, tt);             |
287
288    }
289
290    return(value_array);
291 }
```

Линейный поиск по двум осям выполняется в строках 278 и 279. Кодирование простое и понятное и приводит к имплементации дружественной для кеша. Теперь давайте взглянем на бисекционный поиск в следующем ниже листинге.

Листинг 5.2 Алгоритм бисекционного поиска в таблице

PerfectHash/table.c

```
293 double *interpolate_bisection(int isize, int xstride,
      int d_axis_size, int t_axis_size, double *d_axis, double *t_axis,
294    double *dens_array, double *temp_array, double *data)
295 {
296    int i;
297
298    double *value_array=(double *)malloc(isize*sizeof(double));
299
300    for (i = 0; i<isize; i++){
301       int tt = bisection(t_axis, t_axis_size-2,
                           temp_array[i]);
302       int dd = bisection(d_axis, d_axis_size-2,
                           dens_array[i]);
303
304       double xfrac = (dens_array[i]-d_axis[dd])/
                       (d_axis[dd+1]-d_axis[dd]);
305       double yfrac = (temp_array[i]-t_axis[tt])/
                       (t_axis[tt+1]-t_axis[tt]);
306       value_array[i] =
                xfrac * yfrac *data(dd+1,tt+1)
307        + (1.0-xfrac)* yfrac *data(dd, tt+1)
308        + xfrac *(1.0-yfrac)*data(dd+1,tt)
309        + (1.0-xfrac)*(1.0-yfrac)*data(dd, tt);
310    }
311
312    return(value_array);
313 }
314
315 int bisection(double *axis, int axis_size, double value)
316 {
317    int ibot = 0;
318    int itop = axis_size+1;
319
320    while (itop - ibot > 1){
321       int imid = (itop + ibot) /2;
322       if ( value >= axis[imid] )
323          ibot = imid;
324       else
325          itop = imid;
326    }
327    return(ibot);
328 }
```

Вызовы bisection

Интерполяция

Бисекционный алгоритм

Бисекционный код слегка длиннее, чем линейный поиск (листинг 5.1), но он должен иметь меньше операционной сложности. В разделе 5.5.1 мы рассмотрим другие алгоритмы поиска в таблице и покажем их относительную производительность на рис. 5.8.

Спойлер: бисекционный поиск ненамного быстрее, чем линейный поиск, как вы могли бы ожидать, даже с учетом стоимости интерполяции. Хотя это и так, данный анализ показывает, что линейный поиск не настолько уж медленен, как можно было бы ожидать.

5.3 Параллельные алгоритмы: что это такое?

Теперь давайте возьмем еще один пример из повседневной жизни, чтобы ввести несколько идей параллельных алгоритмов. Первый пример демонстрирует, как алгоритмический подход без сравнений и менее синхронный имплементируется проще и показывает более высокую производительность для высокопараллельного оборудования. Мы обсудим дополнительные примеры в последующих разделах, которые подчеркивают пространственную локальность, воспроизводимость и другие важные атрибуты параллелизма, а затем обобщим все идеи в разделе 5.8 в конце главы.

Пример: сортировка сравнением против сортировки хеш-функцией

Вы хотите отсортировать 100 участников в аудитории из предыдущего примера. Сначала вы пробуете сортировку сравнением.

1. Отсортировать комнату, попросив каждого человека сравнить свою фамилию со своим соседом в своем ряду и переместиться влево, если его фамилия стоит раньше в алфавитном порядке, и вправо, если позже.
2. Продолжать до *N* шагов, где *N* – это число людей в комнате.

Для GPU-процессоров рабочая группа не может обмениваться данными с другой рабочей группой. Давайте предположим, что каждый ряд в аудитории – это рабочая группа. Когда вы достигнете конца ряда, это будет похоже на то, что вы достигли предела рабочей группы GPU, и вам нужно выйти из ядра, чтобы выполнить сравнение со следующим рядом. Необходимость выхода из ядра означает, что требуется несколько вызовов ядра, что увеличивает сложность кодирования и время выполнения. Подробности функционирования GPU будут обсуждаться в главах 9 и 10.

Теперь давайте посмотрим на другой алгоритм сортировки: сортировку хешем. Понять сортировку хешем лучше всего, обратившись к следующему примеру. Мы подробнее рассмотрим элементы хеш-функции в разделе 5.4.

1. По каждой букве поместить ее знак в передней части комнаты.
2. Каждый человек подходит к столу с первой буквой своей фамилии.

3. Для всех букв с большими числами участников повторить для второй буквы в фамилии. Для всех малых чисел участников выполнить любую простую сортировку, в том числе описанную ранее.

Первая сортировка представляет собой сортировку сравнением с использованием алгоритма *сортировки пузырьком*, который обычно показывает слабую производительность. Пузырьковая сортировка проходит по списку, сравнивает соседние элементы и меняет их местами, если они расположены в неправильном порядке. Указанный алгоритм проходит по списку многократно до тех пор, пока список не будет отсортирован. В лучшем случае сортировка сравнением имеет предел алгоритмической сложности $O(N \log N)$. *Сортировка хешем* этот барьер преодолевает, потому что в ней не используются сравнения. В среднем сортировка хешем является операцией $\Theta(1)$ для каждого участника и $\Theta(N)$ для всех участников. Более высокая производительность является значительной; что еще важнее, операции для каждого участника полностью независимы. Наличие полностью независимых операций облегчает параллелизацию алгоритма даже на менее синхронных архитектурах GPU.

Скомбинировав все это вместе, мы можем применить сортировку хешем, чтобы расположить участников и папки в алфавитном порядке и добавить параллелизм, разделяя несколько строк на алфавит на столе регистрации. Две сортировки хешем будут иметь алгоритмическую сложность $\Theta(N)$ и в параллельном режиме будут иметь $\Theta(N/P)$, где P – это число процессоров. Для 16 процессоров и 100 участников параллельное ускорение, по сравнению с последовательным методом грубой силы, составляет $100^2/(100/16) = 1600$ крат. Мы понимаем, что дизайн хеша похож на то, что мы видим на хорошо организованных конференциях или в школьных регистрационных очередях.

5.4 Что такое хеш-функция?

В этом разделе мы обсудим важность хеш-функции. Технические приемы хеширования возникли в 1950-х и 1960-х годах, но их адаптация под многие области применения проходила медленно. В частности, мы пройдемся по компонентам, которые образуют идеальный хеш, пространственное хеширование, идеальное пространственное хеширование, а также все многообещающие варианты использования.

Хеш-функция соотносит ключ со значением, так же как при поиске слова в словаре она используется в качестве ключа для просмотра ее определения. На рис. 5.1 слово *Romero* является ключом, который хешируется для поиска значения, которое в данном случае является псевдонимом или пользовательским именем. В отличие от физического словаря компьютеру требуется не менее 26 возможных мест хранения, умноженных на максимальную длину ключа словаря. Поэтому для компьютера абсолютно необходимо кодировать ключ в более короткую форму, именуемую хешем. Термины «хеш» или «хеширование» относятся к «разде-

лению» ключа на более короткую форму для использования в качестве индекса, указывающего на место хранения значения. Место для хранения коллекции значений конкретного ключа называется *корзиной*. Генерировать хеш из ключа можно самыми разными способами; наилучшие подходы, как правило, зависят от конкретной задачи.

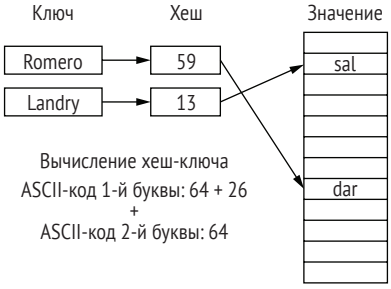


Рис. 5.1 Хеш-таблица для поиска компьютерного псевдонима по фамилии. В ASCII *R* равно 82, а *O* – 79. Тогда мы можем вычислить первый хеш-ключ с помощью 82 – 64 + 26 + 79 – 4 = 59. Хранящееся в хеш-таблице значение является пользовательским именем, иногда называемым псевдонимом

Идеальный хеш – это хеш, в каждой корзине которого имеется не более одной записи. Идеальные хеши просты в обращении, но могут занимать больше памяти. *Минимальный идеальный хеш* состоит всего из одной записи в каждой корзине и не имеет пустых корзин. Вычисление минимальных идеальных хешей занимает больше времени, но, например, для фиксированных множеств ключевых слов языков программирования дополнительное время того стоит. В большинстве хешей, которые мы здесь обсудим, они будут создаваться на лету, опрашиваться и отбрасываться, поэтому важнее более быстрое время создания, чем размер памяти. Там, где идеальный хеш невозможен либо занимает слишком много памяти, можно использовать компактный хеш. *Компактный хеш* сжимает хеш так, что он требует меньше памяти под хранение. Как всегда, между разными методами хеширования имеются компромиссы в трудности программирования, времени выполнения и требуемой памяти.

Коэффициент загрузки – это доля заполненного хеша. Он вычисляется как n/k, где n – это число элементов данных в хеш-таблице, а k – число корзин. Компактные хеши по-прежнему хорошо работают при коэффициентах загрузки от .8 до .9, но после этого их эффективность снижается из-за коллизий. *Коллизии* возникают, когда несколько ключей хотят сохранить свое значение в одной и той же корзине. Важно иметь хорошую хеш-функцию, которая распределяет ключи более равномерно, избегая кластеризации записей, что позволяет повышать коэффициенты загрузки. С помощью компактного хеша ключ и значение сохраняются таким образом, что при извлечении ключ может проверяться на наличие в нем правильного элемента данных.

В предыдущих примерах мы использовали первую букву фамилии в качестве простого хеш-ключа. Несмотря на свою эффективность,

безусловно, в использовании первой буквы есть недостатки. Одна из них заключается в том, что число фамилий, начинающихся с каждой буквы алфавита, распределено неравномерно, что приводит к неравным числам элементов данных в каждой корзине. Вместо этого мы могли бы использовать целочисленное представление строкового значения. Такое представление создает хеш для первых четырех букв имени. Но набор символов дает только 52 возможных значения для 256 местоположений хранения для каждого байта, приводя только к небольшой доле возможных целочисленных ключей. Специальная хеш-функция, которая ожидает только символы, потребовала бы гораздо меньше местоположений для хранения.

5.5 Пространственное хеширование: высокопараллельный алгоритм

В нашем обсуждении в главе 1 использовалась равноразмерная регулярная сетка из примера с вулканом Кракатау на рис. 1.9. В настоящем обсуждении параллельных алгоритмов и пространственного хеширования нам необходимо использовать более сложные вычислительные сетки. В научных симуляциях более сложные сетки принято определять с большей детальностью областей, которые нас интересуют. В больших данных, в частности в анализе и классифицировании изображений, эти более сложные сетки не получили широкого распространения. Тем не менее этот технический прием имел бы там огромную ценность; когда ячейка на изображении имеет смешанные характеристики, нужно лишь разбить ячейку.

Самым большим препятствием для использования более сложных сеток является то, что ее кодирование усложняется, и мы должны внедрять новые вычислительные методики. Для сложных сеток труднее найти методы, которые работают и масштабируются хорошо на параллельных архитектурах. В этом разделе мы покажем способы работы с несколькими широко применяемыми пространственными операциями с использованием высокопараллельных алгоритмов.

Пример: симуляция волн от вулкана Кракатау

Ваш коллектив работает над своим волновым приложением и решает, что для симуляции им требуется более высокая разрешающая способность в некоторых участках на фронте волны и на береговой линии, как показано на рис. 1.9. При этом им не требуется детальная разрешающая способность в других частях решетки. Коллектив решает обратиться к адаптивной детализации сетки, где они имеют возможность устанавливать более детальную разрешающую способность сетки в участках, которые в этом нуждаются.

Адаптивная детализация сетки (adaptive mesh refinement, AMR) на основе ячеек относится к классу технических приемов обработки неструктурированной сетки, которые больше не обладают простотой структурированной решетки в локализации данных. В ячеечной AMR (рис. 5.2) массивы ячеечных данных являются одномерными, и данные могут располагаться в любом порядке. Местоположения сетки переносятся в дополнительные массивы, которые содержат информацию о размере и местоположении каждой ячейки. Таким образом, в сетке есть некоторая структура, но данные полностью неструктурированы. Если углубляться на неструктурированную территорию, то полностью неструктурированная сетка может содержать ячейки треугольников, многогранников или других сложных контуров. Это позволяет ячейкам «вписываться» в границы между сушей и океаном, но за счет более сложных численных операций. Поскольку многие из тех же параллельных алгоритмов для неструктурированных данных применимы к обоим, мы будем работать в основном с примером ячеечной AMR.

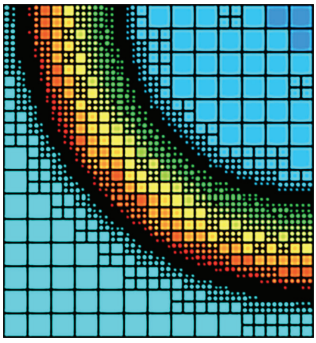


Рис. 5.2 Сетка AMR на основе ячеек для волновой симуляции из мини-приложения CLAMR. Черные квадраты – это ячейки, а квадраты с различными оттенками представляют высоту волны, расходящейся наружу из верхнего правого угла

Технические приемы AMR можно разбить на подходы на основе патчей, блоков и ячеек. В патчевом и блочном методах используются патчи различного размера или блоки фиксированного размера, которые могут – по меньшей мере частично – задействовать регулярную структуру этих групп ячеек. Ячеечный AMR содержит по-настоящему неструктурированные данные, которые могут располагаться в любом порядке. Мини-приложение CLAMR, мелководная ячеечная сетка AMR (https://github.com/lanl/CLAMR.git), было разработано Дэвисом, Николаеффом и Трухильо в 2011 году, когда они проходили летнюю практику в Национальной лаборатории Лос-Аламоса. Они хотели посмотреть, смогут ли приложения ячеечные AMR работать на GPU-процессорах. В ходе работы они нашли прорывные параллельные алгоритмы, которые также ускоряли имплементации CPU. Самым важным из них был пространственный хеш.

Пространственное хеширование – это технический прием, в котором ключ основан на пространственной информации. Алгоритм хеширования сохраняет ту же среднюю алгоритмическую сложность $\Theta(1)$ операций для каждого поиска. Все пространственные запросы могут выполняться

с помощью пространственного хеша; многие из них намного быстрее, чем альтернативные технические приемы. Его базовый принцип заключается в отображении объектов на решетку корзин, расположенных по регулярному шаблону.

Пространственный хеш показан в центре рис. 5.3. Величина корзин выбирается на основе характерного размера отображаемых объектов. Для сетки AMR на основе ячеек используется минимальный размер ячейки. Для частиц или объектов, как показано справа на рисунке, размер ячейки зависит от расстояния взаимодействия. Этот выбор означает, что для расчетов взаимодействия или коллизий необходимо запрашивать только прямо смежные ячейки. Расчеты коллизий являются одной из важнейших областей применения пространственных хешей не только в научных вычислениях по гидродинамике плавных частиц, молекулярной динамике и астрофизике, но и в игровых двигателях и компьютерной графике. Во многих ситуациях пространственная локальность может использоваться для снижения вычислительных затрат.

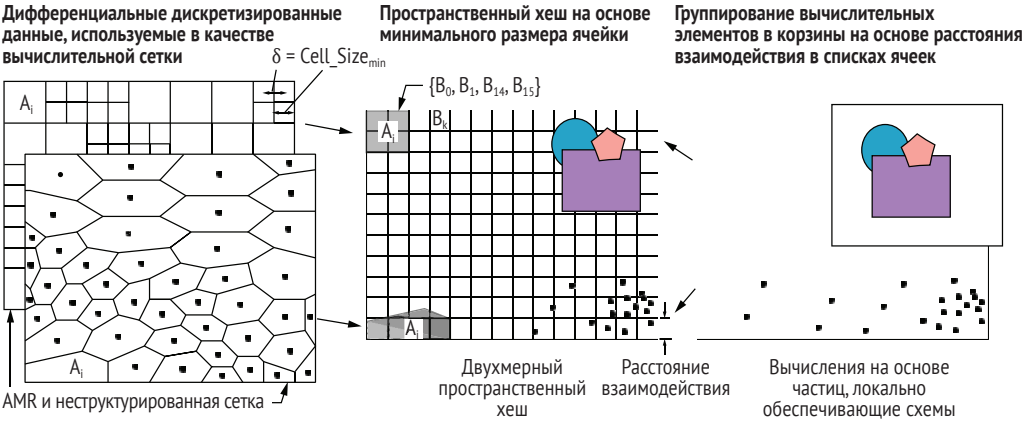


Рис. 5.3 Вычислительные сетки, частицы и объекты, отображаемые на пространственный хеш. Многогранники неструктурированной сетки и прямоугольные ячейки сетки ячеечной адаптивной детализации могут отображаться в пространственный хеш для пространственных операций. Частицы и геометрические объекты также могут извлекать выгоду из их отображения в пространственный хеш, предоставляя информацию об их пространственной локализации так, чтобы необходимо было учитывать только близлежащие элементы

И AMR, и неструктурированная сетка слева на рисунке называются *дифференциальными дискретизированными данными*, поскольку ячейки меньше в тех местах, где градиенты являются более крутыми, чтобы лучше объяснять физические явления. Но у них есть предел в том, насколько меньше ячейки могут становиться. Этот предел не дает размерам корзины становиться слишком малыми. Обе сетки хранят свои индексы ячеек во всех нижележащих корзинах пространственного хеша. Для частиц и геометрических объектов индексы частиц и идентификаторы объектов хранятся в корзинах. За счет этого обеспечивается форма локально-

сти, которая не дает увеличиваться вычислительной стоимости по мере увеличения размера задачи. Например, если область задачи увеличена слева и сверху, то вычисление взаимодействия в правом нижнем углу пространственного хеша остается прежним. Таким образом, алгоритмическая сложность для расчетов частиц остается равной $\Theta(N)$ вместо того, чтобы вырастать до $\Theta(N^2)$. В следующем ниже листинге показан псевдокод цикла взаимодействия, который вместо того, чтобы выполнять поиск во всех частицах, прокручивает близлежащие местоположения во внутреннем цикле.

Листинг 5.3 Псевдокод взаимодействия частиц

```
1 forall particles, ip, in NParticles{
2    forall particles, jp, in Adjacent_Buckets{
3       if (distance between particles < interaction_distance){
4          perform collision or interaction calculation
5       }
6    }
7 }
```

5.5.1 Использование идеального хеширования для пространственных операций с сеткой

Сначала мы обратимся к идеальному хешированию, чтобы сосредоточиться на применении хеширования, а не его внутренней механике. Все эти методы основаны на способности гарантировать, что в каждой ячейке будет находиться только один элемент данных, избегая проблем с обработкой коллизий, когда в ячейке может быть более одного элемента данных. Для идеального хеширования мы проведем разведку четырех наиболее важных пространственных операций.

- *Отыскание соседей* – определение местоположения одного или двух соседей с каждой стороны ячейки.
- *Перекомпоновка* – отображение еще одной сетки AMR на текущую сетку AMR.
- *Поиск в таблице* – локализация интервалов в двухмерной таблице для выполнения интерполяции.
- *Сортировка* – одномерная или двухмерная сортировка ячеечных данных.

Весь исходный код примеров для четырех операций в одной и двух размерностях находится по адресу https://github.com/lanl/PerfectHash.git в рамках лицензии для открытого исходного кода. Указанный исходный код также привязан к примерам из этой главы. В идеальном хеш-коде используется CMake и выполняется проверка доступности OpenCL. Если у вас нет возможностей OpenCL, то код это обнаружит и не будет компилировать имплементации OpenCL. Остальные случаи на CPU по-прежнему будут выполняться.

Отыскание соседей с использованием пространственного идеального хеша

Отыскание соседей – одна из важнейших пространственных операций. В научных вычислениях материал, перемещенный из одной ячейки, должен переместиться в смежную ячейку. Нам нужно знать, в какую ячейку его перемещать, чтобы вычислять количество материала и его перемещать. В анализе изображений характеристики смежной ячейки могут давать важную информацию о композиции текущей ячейки.

Правила для сетки AMR в мини-приложении CLAMR состоят в том, что в детализации грани ячейки может быть только одноуровневый скачок. Кроме того, у каждой ячейки с каждой стороны список соседей представлен всего одной из соседних ячеек, и выбираться должна нижняя ячейка либо ячейка слева от каждой пары, как показано на рис. 5.4. Вторая ячейка пары обнаруживается с помощью списка соседей у первой ячейки; например, `ntop[nleft[ic]]`. Тогда задача заключается в настройке массивов соседей для каждой ячейки.

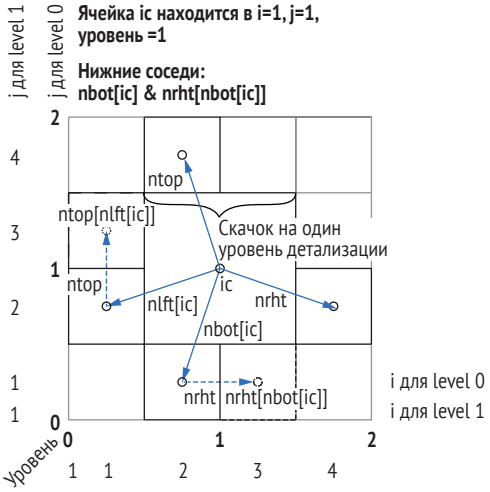


Рис. 5.4 Левый сосед – это нижняя ячейка из двух слева, а нижний сосед – это ячейка слева из двух ниже. Аналогичным образом правый сосед – это нижняя из двух справа, а верхний сосед находится слева из двух ячеек выше

Каковы возможные алгоритмы отыскания соседей? Наивный способ состоит в поиске во всех других ячейках ячейки, которая является смежной. Это можно сделать, обращаясь к переменным *i*, *j* и *level* в каждой ячейке. Наивный алгоритм имеет алгоритмическую сложность $O(N^2)$. Он показывает хорошую производительность с малым числом ячеек, но сложность времени выполнения быстро становится крупной. Некоторые распространенные альтернативные алгоритмы базируются на дереве, такие как алгоритмы на основе k-D-дерева и квадродерева (октодерева в трех размерностях). Это алгоритмы основаны на сравнении, масштабируются как $O(N \log N)$ и будут определены позже. Исходный код для вычисления соседей в двух размерностях, включая k-D-дерево, грубую силу, имплементацию хеша для CPU и GPU, можно найти по адресу https://github.com/lanl/PerfectHash.git наряду с другими приложениями пространственного идеального хеша, обсуждаемыми далее в этой главе.

k-D-дерево разбивает сетку на две равные половины в размерности x, а затем на две равные половины в размерности y, повторяя до тех пор, пока объект не будет найден. Алгоритм строительства k-D-дерева имеет алгоритмическую сложность $O(N \log N)$, и каждый отдельный поиск тоже имеет $O(N \log N)$.

Квадродерево имеет четыре дочерних элемента для каждого родителя, по одному для каждого квадранта. Это точно соответствует подразделению ячеечной сетки AMR. Полное квадродерево начинается вверху, или с корня, с одной ячейки и подразделяется на самый детальный уровень сетки AMR. «Усеченное» квадродерево начинается с самого грубого уровня сетки и имеет квадродерево для каждой грубой ячейки, отображаясь вниз до самого детального уровня. Алгоритм квадродерева основан на сравнении и имеет алгоритмическую сложность $O(N \log N)$.

Лимитирование только одноуровневым скачком по грани называется *градуированной сеткой*. В ячеечной AMR градуированные сетки являются обычным явлением, но другие приложения квадродеревьев, такие как *n*-телесные приложения в астрофизике, приводят к гораздо более крупным скачкам в квадродревесной структуре данных. Одноуровневый скачок в детализации позволяет нам улучшать дизайн алгоритма отыскания соседей. Мы можем начинать поиск в листе, представляющем нашу ячейку, и – самое большее – нам нужно подниматься только на два уровня дерева, чтобы найти нашего соседа. Для поиска близкого соседа похожего размера поиск должен начинаться в листьях и использовать квадродерево. Для поиска крупных нерегулярных объектов следует использовать k-D-дерево, и поиск должен начинаться в корне дерева. Надлежащее использование древесных поисковых алгоритмов может обеспечивать жизнеспособную имплементацию на CPU, но сравнение и строительство дерева представляют трудности для GPU-процессоров, в которых невозможно легко выполнять сравнения за пределами рабочей группы.

Это создает почву для разработки пространственного хеша с целью выполнения операции отыскания соседей. Отсутствие коллизий в нашем пространственном хеше можно гарантировать, сделав корзины в хеше размером с самые детальные ячейки в сетке AMR. Тогда алгоритм принимает следующий вид:

- выделить пространственный хеш размером с самый детальный уровень ячеечной сетки AMR;
- для каждой ячейки в сетке AMR записать номер ячейки в хеш-корзины, соответствующие ячейке;
- вычислить индекс для более детализированной ячейки на одну ячейку наружу от текущей ячейки с каждой стороны;
- прочитать значение, помещенное в хеш-корзину в этом месте.

Для сетки, показанной на рис. 5.5, за фазой записи следует фаза чтения для поиска индекса ячейки правого соседа.

Этот алгоритм хорошо подходит для GPU-процессоров и показан в листинге 5.5. В первой имплементации перенос с CPU на GPU занял менее суток. На имплементирование изначального k-D-дерева на GPU потребуются недели или месяцы. Алгоритмическая сложность также нарушает пороговое значение $O(\log N)$ и составляет в среднем $\Theta(N)$ для N ячеек.

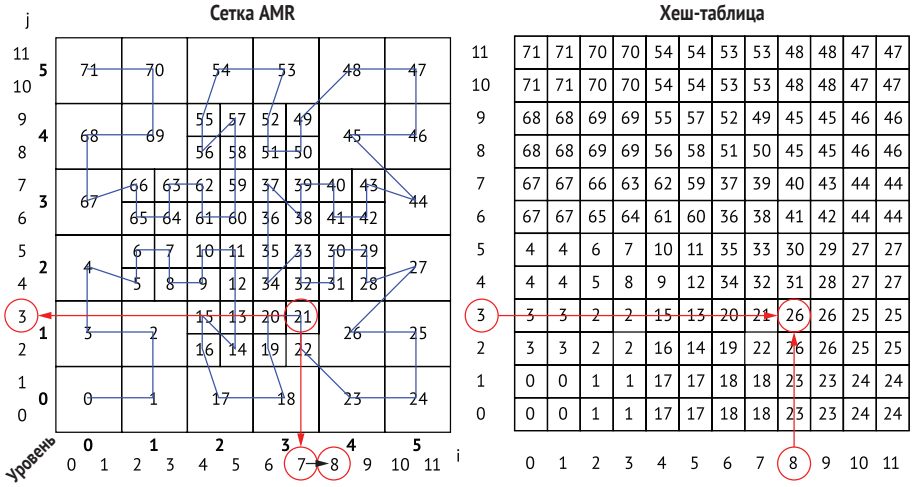


Рис. 5.5 Отыскание правого соседа ячейки 21 с использованием пространственного идеального хеша

Эта первая имплементация вычисления соседа на основе идеального хеша была на порядок быстрее на CPU, чем метод на основе k-D-дерева, и на дополнительный порядок быстрее на GPU, чем на одиночном ядре CPU, в общей сложности в 3157 раз (рис. 5.6). Исследование производительности алгоритма проводилось на GPU NVIDIA V100 и CPU Skylake Gold 5118 с номинальной тактовой частотой 2.30 ГГц. Указанная архитектура также использовалась во всех приведенных в этой главе результатах. Ядро CPU и архитектура GPU являются самыми лучшими из доступных в 2018 году, что дает *лучшее в 2018 году* сравнение параллельного ускорения (обозначение ускорений описывается в разделе 1.6). Но данное сравнение не является архитектурным сравнением между CPU и GPU. Если бы на этом CPU использовались 24 виртуальных ядра, то CPU тоже получил бы значительное параллельное ускорение.

Насколько трудно писать код для такой производительности? Давайте взглянем на код хеш-таблицы в листинге 5.4 для CPU. На вход процедуры подаются одномерные массивы, `i`, `j` и `level`, где `level` – это уровень детализации, а `i` и `j` – строка и столбец ячейки в сетке на уровне детализации этой ячейки. Весь листинг состоит примерно из дюжины строк кода.

Рис. 5.6 Алгоритм и параллельное ускорение в общей сложности в 3157 раз. Новый алгоритм обеспечивает параллельное ускорение на GPU

Листинг 5.4 Написание пространственной хеш-таблицы для GPU

neigh2d.c из PerfectHash

```
452 int *levtable = (int *)malloc(levmx+1);
453 for (int lev=0; lev<levmx+1; lev++)
       levtable[lev] = (int)pow(2,lev);
454
455 int jmaxsize = mesh_size*levtable[levmx];
456 int imaxsize = mesh_size*levtable[levmx];
457 int **hash = (int **)genmatrix(jmaxsize,
                 imaxsize, sizeof(int));
458
459 for(int ic=0; ic<ncells; ic++){
460    int lev = level[ic];
461    for (int jj=j[ic]*levtable[levmx-lev];
          jj<(j[ic]+1)*levtable[levmx-lev]; jj++) {
462       for (int ii=i[ic]*levtable[levmx-lev];
             ii<(i[ic]+1)*levtable[levmx-lev]; ii++) {
463         hash[jj][ii] = ic;
464       }
465    }
466 }
```

Строит таблицу степеней двойки (1, 2, 4, ...)

Задает число строк и столбцов на самом детальном уровне

Выделяет хеш-таблицу

Отображает ячейки в хеш-таблицу

Циклы в строках 459, 461 и 462 ссылаются на одномерные массивы `i`, `j` и `level`; `level` – это уровень детализации, где `0` – грубый уровень, а от `1` до `levmax` – уровни детализации. `i` и `j` массивов представляют строку и столбец ячейки в сетке на уровне детализации этой ячейки.

В листинге 5.5 показан код написания пространственного хеша в OpenCL для GPU, который аналогичен листингу 5.4. Хотя мы еще не рассматривали OpenCL, простота кода GPU очевидна даже без понимания всех деталей. Давайте проведем краткое сравнение, чтобы понять, как должен поменяться исходный код для GPU. Мы определяем макрокоманду для обработки двухмерной индексации и для того, чтобы код больше походил на версию для CPU. Тогда самая большая разница состоит в отсутствии цикла по ячейкам. Для кода GPU это типично; в нем внешние циклы удаляются и вместо этого обрабатываются запуском вычислительного ядра. Индекс ячейки предоставляется для каждого потока процессора внутренне вызовом функции `get_global_id`. Подробнее об этом примере и написании кода OpenCL в целом будет рассказано в главе 12.

Листинг 5.5 Написание пространственной хеш-таблицы на GPU в OpenCL

`neigh2d_kern.cl` из `PerfectHash`

```
 77 #define hashval(j,i) hash[(j)*imaxsize+(i)]
 78
 79 __kernel void hash_setup_kern(
 80       const uint isize,
 81       const uint mesh_size,
 82       const uint levmx,
 83       __global const int *levtable,
 84       __global const int *i,
 85       __global const int *j,
 86       __global const int *level,
 87       __global int *hash
 88       ) {
 89
 90    const uint ic = get_global_id(0);
 91    if (ic >= isize) return;
 92
 93    int imaxsize = mesh_size*levtable[levmx];
 94    int lev = level[ic];
 95    int ii = i[ic];
 96    int jj = j[ic];
 97    int levdiff = levmx - lev;
 98
 99    int iimin =  ii   *levtable[levdiff];
100    int iimax = (ii+1)*levtable[levdiff];
101    int jjmin =  jj   *levtable[levdiff];
102    int jjmax = (jj+1)*levtable[levdiff];
103
104    for (   int jjj = jjmin; jjj < jjmax; jjj++) {
105       for (int iii = iimin; iii < iimax; iii++) {
106          hashval(jjj, iii) = ic;
107       }
108    }
109 }
```

Подает таблицу степеней 2 вместе с i, j и level

Цикл по ячейкам подразумевается ядром GPU; каждый поток является ячейкой

Инструкция return важна, чтобы избежать чтения после конца массивов

Вычисляет устанавливаемые границы базовых хеш-корзин

Задает значение хеш-таблицы равным ИД потока (номеру ячейки)

Программный код для извлечения индексов соседей также прост, как показано в листинге 5.6. Он содержит простой цикл по ячейкам и чтение хеш-таблицы в том месте, где расположение соседей будет на самом детальном уровне сетки. Местоположение соседей можно найти, приращивая строку или столбец на одну ячейку в нужном направлении. Для левого или нижнего соседа приращение равно 1, в то время как для правого или верхнего соседа приращение равно полной ширине сетки в направлении x или `imaxsize`.

Листинг 5.6 Отыскание соседей в пространственной хеш-таблице на CPU

`neigh2d.c` из `PerfectHash`

```
472 for (int ic=0; ic<ncells; ic++){
473    int ii = i[ic];
474    int jj = j[ic];
475    int lev = level[ic];
476    int levmult = levtable[levmx-lev];
477    int nlftval =
          hash[     jj   *levmult             ]
              [MAX( ii   *levmult-1,0        )];
478    int nrhtval =
          hash[     jj   *levmult             ]
              [MIN((ii+1)*levmult, imaxsize-1)];
480    int nbotval =
          hash[MAX( jj   *levmult-1,0)        ]
              [     ii   *levmult             ];
481    int ntopval =
          hash[MIN((jj+1)*levmult, jmaxsize-1)]
              [     ii   *levmult             ];
482    neigh2d[ic].left  = nlftval;
483    neigh2d[ic].right = nrhtval;
484    neigh2d[ic].bot   = nbotval;
485    neigh2d[ic].top   = ntopval;
486}
```

Вычисляет местоположение соседской ячейки для запроса, используя максимум/минимум, чтобы держать его в границах

Задает значение соседа для выходных массивов

Для GPU мы снова устраняем цикл по ячейкам и заменяем его вызовом `get_global_id`, как показано в следующем ниже листинге.

Листинг 5.7 Отыскание соседей из пространственной хеш-таблицы на GPU в OpenCL

`neigh2d_kern.cl` из `PerfectHash`

```
113 #define hashval(j,i) hash[(j)*imaxsize+(i)]
114
115 __kernel void calc_neighbor2d_kern(
116       const int isize,
117       const uint mesh_size,
118       const int levmx,
119       __global const int *levtable,
120       __global const int *i,
```

```
121       __global const int *j,
122       __global const int *level,
123       __global const int *hash,
124       __global struct neighbor2d *neigh2d
125       ) {
126
127    const uint ic = get_global_id(0);  <-- Получает ИД ячейки для потока
128    if (ic >= isize) return;
129
130    int imaxsize = mesh_size*levtable[levmx];
131    int jmaxsize = mesh_size*levtable[levmx];
132
133    int ii = i[ic];  <-- Остальная часть кода аналогична версии CPU
134    int jj = j[ic];
135    int lev = level[ic];
136    int levmult = levtable[levmx-lev];
137
138    int nlftval = hashval(     jj   *levmult            ,
                             max( ii   *levmult-1,0       ));
139    int nrhtval = hashval(     jj   *levmult            ,
                             min((ii+1)*levmult, imaxsize-1));
140    int nbotval = hashval(max( jj   *levmult-1,0)       ,
                                  ii   *levmult            );
141    int ntopval = hashval(min((jj+1)*levmult, jmaxsize-1),
                                  ii   *levmult            );
142    neigh2d[ic].left   = nlftval;
143    neigh2d[ic].right  = nrhtval;
144    neigh2d[ic].bottom = nbotval;
145    neigh2d[ic].top    = ntopval;
146 }
```

Сравните простоту этого кода с кодом k-D-дерева для CPU, длина которого составляет тысячу строк!

Вычисления перекомпоновки с использованием пространственного идеального хеша

Еще одной важной операцией числовой сетки является перекомпоновка (remap) из одной сетки в другую. Быстрые перекомпоновки позволяют выполнять на сетках разные физические операции, оптимизированные под их индивидуальные потребности.

В данном случае мы рассмотрим перекомпоновку значений из одной ячеечной сетки AMR в другую ячеечную сетку AMR. Перекомпоновка сетки также может предусматривать неструктурированные сетки или симуляции на основе частиц, но эти технические приемы более усложнены. Настроечная фаза идентична случаю с отысканием соседа, когда индекс ячейки для каждой ячейки записывается в пространственный хеш. В данном случае пространственный хеш создается для сетки-источника. Затем фаза чтения, показанная в листинге 5.8, запрашивает в пространственном хеше номера ячеек, соответствующих каждой ячейке целевой сетки, и суммирует значения из сетки-источника в целевую сетку по-

сле внесения корректировки на разницу ячеек в размерах. Для этой демонстрации мы упростили исходный код из примера по адресу https://github.com/EssentialsofParallelComputing/Chapter5.git.

Листинг 5.8 Читающая фаза перекомпоновывания значения на CPU

remap2.c из PerfectHash

```
211 for(int jc = 0; jc < ncells_target; jc++) {
212    int ii = mesh_target.i[jc];
213    int jj = mesh_target.j[jc];
214    int lev = mesh_target.level[jc];
215    int lev_mod = two_to_the(levmx - lev);
216    double value_sum = 0.0;
217    for(int jjj = jj*lev_mod;
               jjj < (jj+1)*lev_mod; jjj++) {
218       for(int iii = ii*lev_mod;
                  iii < (ii+1)*lev_mod; iii++) {
219          int ic = hash_table[jjj*i_max+iii];
220          value_sum += value_source[ic] /
                (double)four_to_the(
                   levmx-mesh_source.level[ic]
                );
221       }
222    }
223    value_remap[jc] += value_sum;
224 }
```

Получает местоположение ячейки в целевой сетке

Запрашивает пространственный хеш для ячеек сетки-источника

Суммирует значения из сетки-источника, внося корректировку под относительные размеры ячеек

На рис. 5.7 показано повышение производительности за счет перекомпоновки с использованием пространственного идеального хеша. Благодаря алгоритму происходит ускорение, а затем дополнительное параллельное ускорение при выполнении на GPU в общей сложности более чем в 1000 раз быстрее. Параллельное ускорение на GPU стало возможным благодаря простоте имплементации алгоритма на GPU. Хорошее параллельное ускорение также должно быть возможно и на мультиядерном CPU.

Поиски в таблице с использованием пространственного идеального хеша

Операция поиска значений из табличных данных представляет собой другой тип локальности, который может использоваться пространственным хешем. Хеширование может использоваться для поиска интервалов по обеим осям для интерполяции. В этом примере мы использовали поисковую таблицу 51×23 значений уравнения состояния. Двумя осями являются плотность и температура, при этом между значениями на каждой оси используется равное расстояние. Мы будем использовать n для длины оси и N для числа поисков в таблице, которые необходимо выполнить. В этом исследовании мы применили три алгоритма.

- Первый – это линейный поиск (грубой силой), начинающийся с первого столбца и строки. Алгоритм методом грубой силы (путем

исчерпывающего перебора) должен иметь алгоритмическую сложность $O(n)$ для каждого запроса данных или для всех N, $O(N \times n)$, где n – это, соответственно, число столбцов или строк для каждой оси.

- Второй – это бисекционный поиск, который обращается к значению срединной точки возможного диапазона и рекурсивно сужает местоположение для интервала. Алгоритм бисекционного поиска должен иметь алгоритмическую сложность $O(\log n)$ для каждого запроса данных.
- Наконец, мы использовали хеш для $O(1)$-го поиска интервала для каждой оси. Мы измерили производительность хеша как на одиночном ядре CPU, так и на GPU. Для получения результата тестовый код выполняет поиск интервала по обеим осям и простую интерполяцию значений данных из таблицы.

Рис. 5.7 Ускорение алгоритма перекомпоновки за счет перехода алгоритма с k-D-дерева на хеш на одиночном ядре CPU, а затем переноса на GPU для параллельного ускорения

На рис. 5.8 показаны результаты производительности разных алгоритмов. Результаты могут преподнести несколько сюрпризов. Бисекционный поиск не быстрее, чем метод грубой силы (линейный поиск), несмотря на то что этот алгоритм имеет алгоритмическую сложность $O(N \log n)$ в отличие от алгоритма $O(N \times n)$. Это кажется противоречащим простой модели производительности, которая указывает на то, что ускорение должно быть 4–5-кратным для поиска по каждой оси. С интерполяцией мы все равно ожидаем улучшения примерно в 2 раза. Однако тут есть простое объяснение, о котором вы можете догадаться из наших обсуждений в разделе 5.2.

В линейном поиске поиск интервала на каждой оси требует – самое большее – только двух загрузок кеша на одной оси и четырех на другой! Бисекция требует одинакового числа загрузок кеша. Учитывая загрузки кеша, мы не ожидаем никакой разницы в производительности. Алгоритм хеширования может напрямую перейти к правильному интервалу, но для этого все равно потребуется загрузка кеша. Сокращение в загрузках кеша является примерно трехкратным. В алгоритме хеширования дополнительное улучшение, вероятно, связано с сокращением условных переходов. Наблюдаемая производительность соответствует ожиданиям, если включить эффект иерархии кеша.

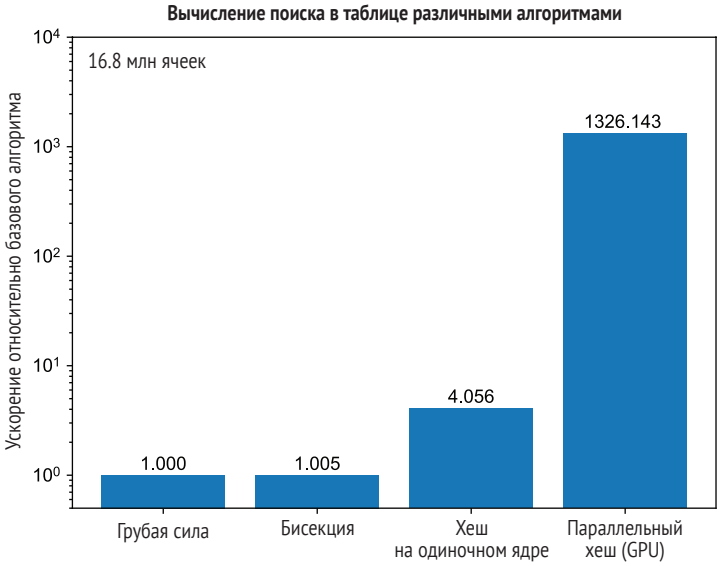


Рис. 5.8 Алгоритмы, используемые для поиска в таблице, показывают значительное ускорение алгоритма хеширования на GPU

Перенос алгоритма на GPU выполнить немного труднее, и он показывает улучшения производительности, которые возможны в этом процессе. В целях понимания того, что было проделано, давайте сначала взглянем на имплементацию хеша на CPU в листинге 5.9. Программный код перебирает все 16 млн значений, находя интервалы на каждой оси, а затем интерполирует данные в таблице, чтобы получить результирующее значение. Используя хеширование, мы можем найти местоположение интервалов с помощью простого арифметического выражения без условных переходов.

Листинг 5.9 Исходный код табличной интерполяции для CPU

`table.c из PerfectHash`

```
272 double dens_incr =
```

```
      (d_axis[50]-d_axis[0])/50.0;
273 double temp_incr =
      (t_axis[22]-t_axis[0])/22.0;
274
275 for (int i = 0; i<isize; i++){
276    int tt = (temp[i]-t_axis[0])/temp_incr;
277    int dd = (dens[i]-d_axis[0])/dens_incr;
278
279    double xf = (dens[i]-d_axis[dd])/
280                (d_axis[dd+1]-d_axis[dd]);
281    double yf = (temp[i]-t_axis[tt])/
282                (t_axis[tt+1]-t_axis[tt]);
283    value_array[i] =
                xf *     yf *data(dd+1,tt+1)
284      + (1.0-xf)*     yf *data(dd, tt+1)
285      + xf      *(1.0-yf)*data(dd+1,tt)
286      + (1.0-xf)*(1.0-yf)*data(dd, tt);
287 }
```

Вычисляет постоянное приращение для поиска данных по каждой оси

Определяет интервал для интерполяции и долю в интервале

Двухлинейная интерполяция для заполнения value_array результатами

Мы могли бы просто перенести это на GPU, как это делалось в предыдущих случаях, удалив цикл for и заменив его вызовом функции `get_global_id`. Но GPU имеет малый кеш локальной памяти, совместно используемый каждой рабочей группой, который может содержать около 4000 значений двойной точности. У нас 1173 значения в таблице и 51+23 осевых значения. Они могут поместиться в кеше локальной памяти, к которому можно быстро получать доступ и которым могут пользоваться все потоки процессора в рабочей группе. Код в листинге 5.10 показывает, как это делается. Первая часть кода кооперативно загружает значения данных в локальную память, используя все потоки. Затем требуется синхронизация, чтобы гарантировать загрузку всех данных перед переходом к интерполяционному ядру. Оставшийся код выглядит почти так же, как код CPU в листинге 5.9.

Листинг 5.10 Программный код интерполяции таблицы в OpenCL для GPU

`table_kern.cl` из `PerfectHash`

```
45 #define dataval(x,y) data[(x)+((y)*xstride)]
46
47 __kernel void interpolate_kernel(
48    const uint isize,
49    const uint xaxis_size,
50    const uint yaxis_size,
51    const uint dsize,
52    __global const double *xaxis_buffer,
53    __global const double *yaxis_buffer,
54    __global const double *data_buffer,
55    __local double *xaxis,
56    __local double *yaxis,
```

```
57    __local double *data,
58    __global const double *x_array,
59    __global const double *y_array,
60    __global double *value
61    )
62 {
63    const uint tid = get_local_id(0);
64    const uint wgs = get_local_size(0);
65    const uint gid = get_global_id(0);
66
67    if (tid < xaxis_size)
          xaxis[tid]=xaxis_buffer[tid];
68    if (tid < yaxis_size)
          yaxis[tid]=yaxis_buffer[tid];
69
70    for (uint wid = tid; wid<d_size; wid+=wgs){
71       data[wid] = data_buffer[wid];
72    }
73
74    barrier(CLK_LOCAL_MEM_FENCE);
75
76    double x_incr = (xaxis[50]-xaxis[0])/50.0;
77    double y_incr = (yaxis[22]-yaxis[0])/22.0;
78
79    int xstride = 51;
80
81    if (gid < isize) {
82       double xdata = x_array[gid];
83       double ydata = y_array[gid];
84
85       int is = (int)((xdata-xaxis[0])/x_incr);
86       int js = (int)((ydata-yaxis[0])/y_incr);
87       double xf = (xdata-xaxis[is])/
                        (xaxis[is+1]-xaxis[is]);
88       double yf = (ydata-yaxis[js])/
                        (yaxis[js+1]-yaxis[js]);
89
90       value[gid] =
                xf *     yf *dataval(is+1,js+1)
91        + (1.0-xf)*     yf *dataval(is,  js+1)
92        +      xf *(1.0-yf)*dataval(is+1,js)
93        + (1.0-xf)*(1.0-yf)*dataval(is,  js);
94    }
95 }
```

Загружает осевые значения данных

Загружает таблицу данных

Необходимо синхронизировать перед запросами к таблице

Вычисляет постоянное приращение для каждого поиска осевых данных

Загружает следующее значение данных

Определяет интервал для интерполяции и долю в интервале

Двухлинейная интерполяция

Результат производительности для хеш-кода GPU показывает последствие этой оптимизации с более крупным ускорением, чем от производительности одиночного ядра CPU для других вычислительных ядер (kernels).

Сортировка данных сетки с использованием пространственного идеального хеша

Операция сортировки является одним из наиболее изученных алгоритмов и лежит в основе многих других операций. В этом разделе мы рассмотрим частный случай сортировки пространственных данных. Пространственная сортировка может использоваться для отыскания ближайших соседей, устранения дубликатов, упрощения определения дальности, вывода графики и массы других операций.

Для простоты мы будем работать с одномерными данными с минимальным размером ячейки, равным 2.0. Все ячейки должны быть в два раза больше минимального размера ячейки. Тестовый случай позволяет до четырех уровней укрупнения в дополнение к минимальному размеру ячейки для следующих возможностей: 2.0, 4.0, 8.0, 16.0 и 32.0. Размеры корзин генерируются случайно, и корзины упорядочиваются случайно. Сортировка выполняется методом быстрой сортировки, а затем методом сортировки хешем на CPU и GPU. В расчете пространственной сортировки хешем используется информация об одномерных данных. Мы знаем минимальное и максимальное значение X и минимальный размер корзины. Имея эту информацию, мы можем рассчитать гарантируемый идеальным хешем корзинный индекс, используя формулу

$$b_k = \left\lfloor \frac{X_i - X_{min}}{\Delta_{min}} \right\rfloor,$$

где b_k – это корзина для размещения элемента данных, X_i – x-координата ячейки, X_{min} – минимальное значение X, и Δ_{min} – минимальное расстояние между любыми двумя смежными значениями X.

Мы можем продемонстрировать операцию сортировки хешем (рис. 5.9). Минимальная разница между значениями составляет 2.0, поэтому размер корзины, равный 2, гарантирует отсутствие коллизий. Минимальное значение равно 0, поэтому местоположение корзины можно рассчитать с помощью $B_i = X_i/\Delta_{min} = X_i/2.0$. В хеш-таблице можно было бы хранить значение либо индекс. Например, 8, первый ключ, может хранится в корзине 4, либо также может храниться изначальная индексная позиция нуля. Если хранится значение, то мы извлекаем 8 с помощью `hash[4]`. Если хранится индекс, то мы извлекаем его значение с помощью `keys[хеш[4]]`. Сохранение индексной позиции в этом случае происходит немного медленнее, но оно носит более общий характер. Его также можно использовать для переупорядочивания всех массивов в сетке. В тестовом примере для исследования производительности мы используем метод сохранения индекса.

Алгоритм пространственной сортировки хешем имеет алгоритмическую сложность $\Theta(N)$, в то время как быстрая сортировка имеет $\Theta(N \log N)$. Но пространственная сортировка хешем более специализирована для решаемой задачи и может временно занимать больше памяти. Остальные вопросы заключаются в том, насколько трудно написать этот алго-

ритм и какая у него производительность. В следующем ниже листинге показан программный код фазы записи в имплементации пространственного хеша.

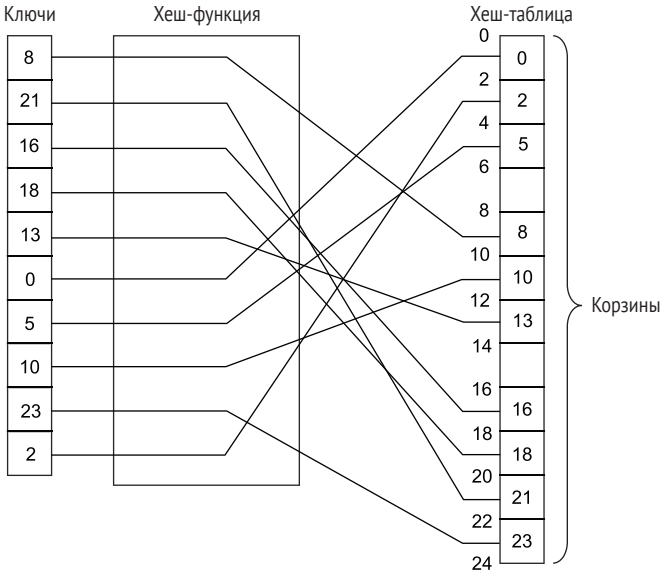


Рис. 5.9 Сортировка с использованием пространственного идеального хеша. Этот метод хранит значение в хеше в корзине, но он также может хранить индексную позицию значения в исходном массиве. Обратите внимание, что размер ячейки 2 с диапазоном от 0 до 24 обозначается маленькими цифрами слева от хеш-таблицы

Листинг 5.11 Пространственная сортировка хешем на CPU

sort.c из PerfectHash

```
283 uint hash_size =
       (uint)((max_val - min_val)/min_diff);
284 hash = (int*)malloc(hash_size*sizeof(int));
285 memset(hash, -1, hash_size*sizeof(int));
286
287 for(uint i = 0; i < length; i++) {
288     hash[(int)((arr[i]-min_val)/min_diff)] = i;
289 }
290
291 int count=0;
292 for(uint i = 0; i < hash_size; i++) {
293     if(hash[i] >= 0) {
294         sorted[count] = arr[hash[i]];
295         count++;
296     }
297 }
298
299 free(hash);
```

Создает хеш-таблицу с корзинами размера min_diff

Задает значение всех элементов хеш-массива равным –1

Помещает индекс текущего элемента массива в хеш в соответствии с тем, куда идет значение arr

Пробегает по хешу и помещает заданные значения в отсортированный массив

Обратите внимание, что код в листинге едва ли превышает дюжину строк. Сравните это с кодом быстрой сортировки, который в пять раз длиннее и намного сложнее.

На рис. 5.10 показана производительность пространственной сортировки как на одинарном ядре CPU, так и на GPU. Как мы увидим, параллельная имплементация на CPU и GPU требует определенных усилий для обеспечения хорошей производительности. Фаза чтения в алгоритме нуждается в хорошо имплементированной префиксной сумме, чтобы извлечение отсортированных значений можно было выполнять в параллельном режиме. Префиксное суммирование является важным шаблоном во многих алгоритмах; мы обсудим его далее в разделе 5.6.

В данном примере в имплементации на GPU используется хорошо имплементированная префиксная сумма, и, как следствие, производительность сортировки на основе пространственного хеша превосходна. В более ранних тестах с размером массива в два миллиона указанная сортировка на GPU была в 3 раза быстрее, чем самая быстрая общая сортировка на GPU, а последовательная версия для CPU была в 4 раза быстрее, чем стандартная быстрая сортировка. При текущей архитектуре CPU и большем размере массива в 16 млн наша сортировка на основе пространственного хеша, как показано, почти в 6 раз быстрее (рис. 5.10). Примечательно, что наша сортировка, написанная за два-три месяца, намного быстрее, чем самые быстрые в настоящее время эталонные сортировки на CPU и GPU-процессорах, тем более что эталонные сортировки являются результатом десятилетий научно-практических исследований и усилий многих исследователей!

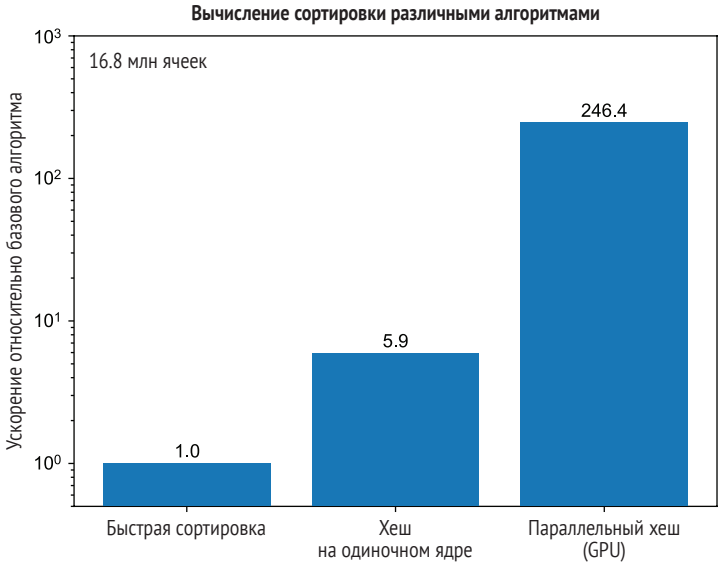


Рис. 5.10 Наша пространственная сортировка хешем показывает ускорение на одиночном ядре CPU и дальнейшее параллельное ускорение на GPU. Наша сортировка работает в 6 раз быстрее, чем текущая самая быстрая сортировка

5.5.2 Использование компактного хеширования для пространственных операций на сетке

Мы еще не закончили разведку методов хеширования. Алгоритмы в разделе идеального хеширования могут быть значительно улучшены. В предыдущем разделе мы разведали использование компактных хешей для операций отыскания соседей и перекомпоновки. Ключевые наблюдения заключаются в том, что нам не нужно осуществлять запись в каждую корзину пространственного хеша, и мы можем улучшить алгоритмы, манипулируя коллизиями. Благодаря этому обеспечивается сжатие пространственных хешей и использование меньшего объема памяти. Это даст нам больше вариантов выбора алгоритма с разными требованиями к памяти и временами выполнения.

Отыскание соседей за счет компактного хеширования и оптимизаций операции записи

Предыдущий простой алгоритм идеального хеша для отыскания соседей показывает хорошую производительность для малых чисел уровней детализации в сетке AMR. Но когда существует шесть или более уровней детализации, грубая ячейка записывает в 64 хеш-корзины, а детальная ячейка должна записывать только в одну, что приводит к дисбалансу загрузки и проблеме с расхождением потоков в параллельных имплементациях.

Поточное расхождение (*дивергенция*) – это ситуация, когда объем работы для каждого потока варьируется и потоки в конечном итоге ждут самого медленного. Мы можем еще больше улучшить алгоритм идеального хеша с помощью оптимизаций, показанных на рис. 5.11. Первая оптимизация заключается в осознании, что поиски соседей отбирают только внешние хеш-корзины ячейки, поэтому нет необходимости писать во внутренние. Дальнейший анализ показывает, что будут запрашиваться только углы или срединные точки представления ячейки в хеше, что еще больше сократит число необходимых записей. На рисунке пример, показанный в крайнем правом углу последовательности, еще больше оптимизирует операции записи, сводя лишь к одной на ячейку, и выполняет несколько операций чтения, если запись существует для более детализированной, одинакового размера либо более грубой соседней ячейки. Этот последний технический прием требует инициализации хеш-таблицы сигнальным значением, таким как –1, чтобы обозначать отсутствие элемента данных.

Но теперь, когда в хеш записывается меньше данных, у нас остается много пустого пространства или разреженности, и мы можем сжать хеш-таблицу вплоть до 1.25-кратного размера от числа записей, что значительно снижает требования к памяти алгоритма. Обратная величина размерного множителя называется *коэффициентом загрузки хеша* и определяется как число заполненных позиций в хеш-таблице, деленное на размер хеш-таблицы. Для размерного множителя 1.25 коэффициент загрузки хеша равен 0.8. Обычно мы используем гораздо мень-

ший коэффициент загрузки, около 0.333, или размерный множитель, равный 3. Это связано с тем, что в параллельной обработке мы хотим избежать того, чтобы один процессор был медленнее других. *Разреженность хеша* представляет собой пустое пространство в хеше. Разреженность указывает на возможность для сжатия.

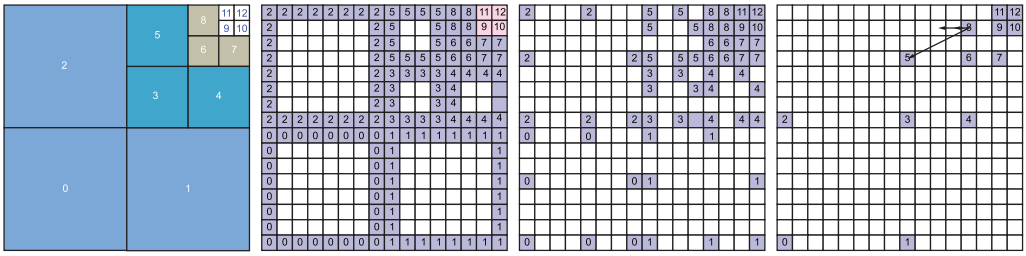


Рис. 5.11 Оптимизация вычисления отыскания соседей с использованием идеального пространственного хеша за счет уменьшения числа операций записи и чтения

На рис. 5.12 показан процесс создания компактного хеша. Из-за сжатия до компактного хеша два элемента данных пытаются сохранить свое значение в ячейке 1. Второй элемент видит, что там уже есть значение, поэтому он ищет следующий открытый слот методом, именуемым *открытой адресацией*. В открытой адресации мы ищем следующий открытый слот в хеш-таблице и сохраняем значение в этом слоте. Помимо открытой адресации есть и другие методы хеширования, но они нередко требуют наличия возможности выделять память во время операции. Выделять память на GPU сложнее, поэтому мы придерживаемся открытой адресации, где коллизии разрешаются путем отыскания альтернативных мест хранения в уже выделенной хеш-таблице.

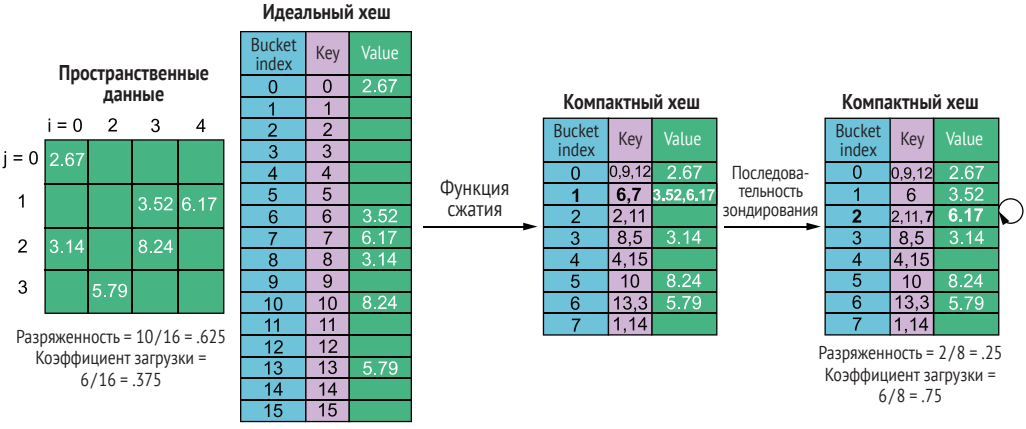


Рис. 5.12 Указанная последовательность слева направо показывает сохранение пространственных данных в идеальном пространственном хеше, его сжатие в меньший хеш, а затем – в случае коллизии – поиск следующего доступного пустого слота для сохранения элемента данных

В открытой адресации есть несколько вариантов, которые можно использовать в качестве попытки для следующего открытого слота. Они таковы:

- *линейное зондирование* – где следующее местоположение элемента данных представляет собой просто следующую корзину в последовательности, пока не будет найдена открытая корзина;
- *квадратичное зондирование* – где приращение возводится в квадрат, вследствие чего выбранные интервалы равны +1, +4, +9 и т. д. от изначального местоположения;
- *двойное хеширование* – где вторая функция хеширования используется для перехода на детерминированное, но псевдослучайное расстояние от первого пробного местоположения.

Причина более сложных вариантов для следующей попытки заключается в предотвращении кластеризации значений в части хеш-таблицы, приводящей к более длительным последовательностям сохранения и опрашивания. Мы используем метод квадратичного зондирования, потому что первые две попытки находятся в кеше, приводя к повышению производительности. Как только слот найден, ключ и значение сохраняются. При чтении хеш-таблицы сохраненный ключ сравнивается с ключом чтения, и если они не совпадают, то чтение пробует следующий слот в таблице.

Мы могли бы оценить эффективность улучшения этих оптимизаций, подсчитав число операций записи и чтения. Но нам нужно скорректировать эти числа записи и чтения, чтобы учесть число строк кеша, а не только сырое число значений. Кроме того, код с оптимизациями содержит больше условных переходов. Отсюда, улучшение времени выполнения является скромным и лучше только для более высоких уровней детализации сетки. Параллельный код на GPU дает больше преимуществ, так как уменьшается поточное расхождение.

На рис. 5.13 показаны измеренные результаты производительности для различных оптимизаций хеш-таблицы для образца сетки AMR с относительно скромным коэффициентом разреженности 30. Исходный код доступен по адресу https://github.com/lanl/CompactHash.git. Последние показатели производительности, продемонстрированные на рис. 5.13 как для CPU, так и для GPU, относятся к прогонам компактного хеша. Стоимость компактного хеша компенсируется отсутствием достаточного объема памяти для инициализации сигнальным значением –1. В результате компактный хеш обладает конкурентоспособной производительностью по сравнению с методами идеального хеширования. При большей разреженности хеш-таблицы чем 30-кратный коэффициент сжатия в этом тестовом примере компактный хеш бывает даже быстрее, чем методы идеального хеширования. Методы ячеечной AMR в целом должны иметь не менее чем 10-кратное сжатие и часто могут превышать 100 крат.

Эти методы хеширования были имплементированы в мини-приложении CLAMR. Программный код переключается между алгоритмом иде-

ального хеширования для низких уровней разреженности и компактным хешем, когда в хеше много пустого места.

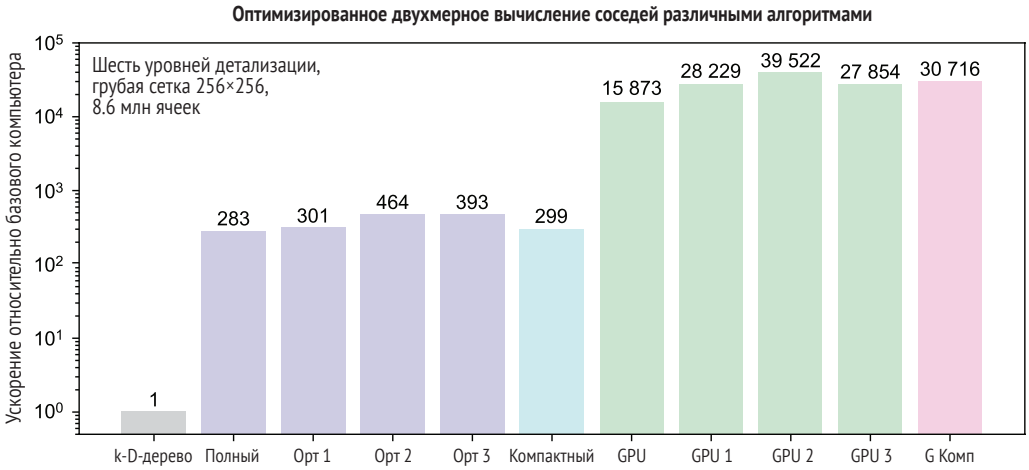


Рис. 5.13 Оптимизированные версии, показанные для CPU и GPU, соответствуют методам, показанным на рис. 5.11. Компактный значит компактный на CPU, а G Комп – компактный на GPU для последнего метода в каждом наборе. Компактный метод работает быстрее, чем изначальный идеальный хеш, и требует значительно меньше памяти. На более высоких уровнях детализации методы, которые уменьшают число операций записи, тоже демонстрируют некоторое преимущество в производительности

Отыскание соседей по грани для неструктурированных сеток

До сих пор мы не обсуждали алгоритмы для неструктурированных сеток, потому что трудно гарантировать, что для них можно легко создать идеальный хеш. Наиболее практичные методы требуют способа обработки коллизий и, следовательно, компактных методов хеширования. Давайте рассмотрим один случай, когда использовать хеш довольно просто. Процедура поиска соседней по грани для многоугольной сетки бывает дорогостоящей. Многие неструктурированные коды хранят словарь соседей, потому что это очень дорого. Технический прием, который мы показываем далее, настолько быстр, что словарь соседей можно вычислять на лету.

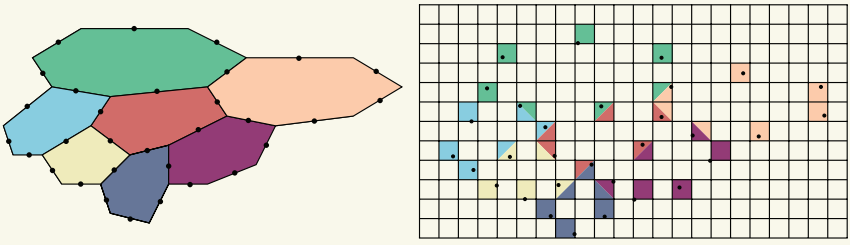
Пример: отыскание соседей по грани для неструктурированной сетки

На следующем ниже рисунке представлена малая часть неструктурированной сетки с многоугольными ячейками. Одной из вычислительных задач для этого типа сетки является отыскание карты связности для каждой грани полигонов. Поиск грубой силой всех остальных элементов кажется разумным для малого числа многоугольников, но для большего числа ячеек это может занять минуты или часы. Поиск по k-D-дереву сокращает время, но есть ли

еще более быстрый способ? Давайте вместо этого попробуем метод, основанный на хеше. Мы накладываем рисунок поверх хеш-таблицы справа от рисунка. Алгоритм выглядит следующим образом:

- мы помещаем точку центра каждой грани в хеш-корзину, куда она попадает;
- каждая ячейка записывает свой номер ячейки в корзину в центре каждой грани. Если грань находится слева и вверху от центра, то она записывает свой индекс в первое из двух мест в корзине; в противном случае она записывает во второе место;
- каждая ячейка проверяет каждую грань, чтобы увидеть наличие числа в другой корзине. Если есть, то это соседняя ячейка. Если нет, то мы имеем внешнюю грань без соседа.

Мы нашли наших соседей за одну запись и одно чтение!



Отыскание соседа для каждой грани каждой ячейки с использованием пространственного хеша. Каждая грань записывает данные в одну из двух корзин в пространственном хеше. Если грань расположена слева и вверху от центра, то она записывается в первую корзину. Если вправо и вниз, то она записывается во вторую. В проходе с операцией чтения она проверяет заполненность другой корзины, и если она заполнена, то номер ячейки является ее соседом (график и алгоритм любезно предоставлены Рэйчел Роби)

Надлежащий размер хеш-таблицы трудно указать. Наилучшим решением является выбор разумного размера на основе числа граней или минимальной длины грани, а затем обработка коллизий, если они происходят.

Перекомпоновки с оптимизацией записи и компактным хешированием

Еще одна операция, перекомпоновка (remap), оптимизируется и настраивается для компактного хеша немного труднее, потому что подход на основе идеального хеша читает все соответствующие ячейки. Сначала мы должны придумать способ, который не требует заполнения каждой хеш-корзины.

Мы записываем индексы ячеек для каждой ячейки в нижний левый угол соответствующего хеша. Затем, во время чтения, если значение не найдено или уровень ячейки во входной сетке неправильный, то мы

ищем место, куда бы записывалась ячейка во входной сетке, если бы она находилась на следующем более грубом уровне. На рис. 5.14 показан этот подход, при котором ячейка 1 в выходной сетке запрашивает местоположение в хеше (0,2) и находит значение –1, поэтому затем она ищет место, где будет следующая более грубая ячейка (0,0), и находит индекс ячейки, равный 1. Плотность ячейки 1 в выходной сетке затем устанавливается равной плотности ячейки 1 во входной сетке. Для ячейки 9 в выходной сетке подход просматривает хеш в (4,4) и находит индекс входной ячейки 3. Затем он просматривает уровень ячейки 3 во входной сетке, и, поскольку уровень ячейки входной сетки выше, он также должен запросить местоположения в хеше (6,4), чтобы получить индекс ячейки 9 и местоположение (4,6), которое возвращает индекс ячейки 4 и местоположение (6,6), чтобы получить индекс ячейки 7. Первые два индекса ячеек находятся на одном уровне, поэтому им не нужно идти дальше. Индекс ячейки 7 находится на более детализированном уровне, поэтому мы должны рекурсивно спуститься в это место, чтобы найти индексы ячеек 8, 5 и 6. Этот исходный код показан в листинге 5.12.

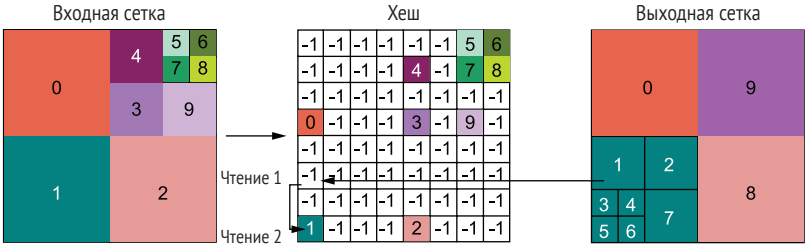


Рис. 5.14 Имплементация однократной записи и многократного чтения для перекомпоновки с использованием пространственного хеша. Первый запрос – это место, куда ячейка того же размера будет записываться из входной сетки, а затем, если значение не найдено, следующий запрос ищет место, где будет записываться ячейка на следующем более грубом уровне

Листинг 5.12 Фаза настройки для перекомпоновки с использованием пространственного хеша с однократной записью на CPU

`singlewrite_remap.cc и meshgen.cc из CompactHashRemap/AMR_remap`

```
47 #define two_to_the(ishift)   (1u <<(ishift) )
48
49 typedef struct {
50    uint ncells;
51    uint ibasesize;
52    uint levmax;
53    uint *dist;
54    uint *i;
55    uint *j;
56    uint *level;
57    double *values;
```

Определяет функцию степени 2n в качестве оператора сдвига для скорости

Структура для хранения характеристик сетки

Число ячеек в сетке

Число грубых ячеек в размерности x

Число уровней детализации в дополнение к базовой сетке

Распределение ячеек по уровням детализации

```
58 } cell_list;
59
60 cell_list icells, ocells;
61
<... много кода по созданию сетки ...>
```

Настраивает входную и выходную сетки

```
120 size_t hash_size = icells.ibasesize*two_to_the(icells.levmax)*
121                    icells.ibasesize*two_to_the(icells.levmax);
122 int *hash = (int *) malloc(hash_size *
                               sizeof(int));
123 uint i_max = icells.ibasesize*two_to_the(icells.levmax);
```

Выделяет хеш-таблицу для идеального хеша

Перед записью выделяется идеальная хеш-таблица и инициализируется сигнальным значением –1 (рис. 5.10). Затем индексы ячеек из входной сетки записываются в хеш (листинг 5.13). Исходный код доступен по адресу https://github.com/lanl/CompactHashRemap.git в файле AMR_remap/singlewrite_remap.cc вместе с вариантами использования компактной хеш-таблицы и OpenMP. Версия OpenCL для GPU находится в файле AMR_remap/h_remap_kern.cl.

Листинг 5.13 Фаза записи для перекомпоновки с использованием пространственного хеша с однократной записью на CPU

AMR_remap/singlewrite_remap.cc из CompactHashRemap

```
127 for (uint i = 0; i < icells.ncells; i++) {
128     uint lev_mod =
           two_to_the(icells.levmax -
                      icells.level[i]);
129     hash[((icells.j[i] * lev_mod) * i_max)
           + (icells.i[i] * lev_mod)] = i;
130 }
```

Часть с фактическим чтением операции записи в хеш составляет всего четыре строки

Множитель для конвертирования между уровнями сетки

Вычисляет индекс для одномерной хеш-таблицы

Программный код для фазы чтения (листинг 5.14) имеет интересную структуру. Первая часть в сущности разделена на два случая: ячейка в одном и том же местоположении входной сетки имеет тот же уровень или грубее, либо это множество более детализированных ячеек. В первом случае мы прокручиваем уровни в цикле до тех пор, пока не найдем нужный уровень и не установим значение в выходной сетке равным значению во входной сетке. Если она детализированнее, то мы рекурсивно спускаемся по уровням, суммируя значения по ходу.

Листинг 5.14 Фаза чтения для перекомпоновки с использованием пространственного хеша с однократной записью на CPU

AMR_remap/singlewrite_remap.cc из CompactHashRemap

```
132 for (uint i = 0; i < ocells.ncells; i++) {
133    uint io = ocells.i[i];
134    uint jo = ocells.j[i];
135    uint lev = ocells.level[i];
```

```
136
137    uint lev_mod = two_to_the(ocells.levmax - lev);
138    uint ii = io*lev_mod;
139    uint ji = jo*lev_mod;
140
141    uint key = ji*i_max + ii;
142    int probe = hash[key];
144
145    if (lev > ocells.levmax){lev = ocells.levmax;}
146
147    while(probe < 0 && lev > 0) {
148        lev--;
149        uint lev_diff = ocells.levmax - lev;
150        ii >>= lev_diff;
151        ii <<= lev_diff;
152        ji >>= lev_diff;
153        ji <<= lev_diff;
154        key = ji*i_max + ii;
155        probe = hash[key];
156     }
157     if (lev >= icells.level[probe]) {
158         ocells.values[i] = icells.values[probe];
159     } else {
160         ocells.values[i] =
               avg_sub_cells(icells, ji, ii,
                             lev, hash);
161     }
162 }
163 double avg_sub_cells (cell_list icells, uint ji, uint ii,
          uint level, int *hash) {
164     uint key, i_max, jump;
165     double sum = 0.0;
166     i_max = icells.ibasesize*two_to_the(icells.levmax);
167     jump = two_to_the(icells.levmax - level - 1);
168
169     for (int j = 0; j < 2; j++) {
170         for (int i = 0; i < 2; i++) {
171             key = ((ji + (j*jump)) * i_max) + (ii + (i*jump));
172             int ic = hash[key];
173             if (icells.level[ic] == (level + 1)) {
174                 sum += icells.values[ic];
175             } else {
176                 sum += avg_sub_cells(icells, ji + (j*jump),
                    ii + (i*jump), level+1, hash);
177             }
178         }
179     }
180
181     return sum/4.0;
182 }
```

Если сигнальное значение найдено, то продолжает до более грубых уровней

Поскольку она находится на том же уровне или более грубом, задает значение ИД найденной ячейки во входной сетке

Для более детализированных ячеек рекурсивно спускается и суммирует участников

Хорошо, похоже, она в полном порядке с CPU, но как она будет работать на GPU? Предположительно, рекурсия на GPU не поддерживается.

Похоже, нет никакого простого способа написать эту фазу без рекурсии. Но мы протестировали ее на GPU и обнаружили, что код работает. Он отлично работает на всех GPU, которые мы опробовали для лимитированного числа уровней детализации, которые будут использоваться в любой практической сетке. Очевидно, что лимитированное количество рекурсии работает на GPU! Затем мы имплементировали версии этого подхода с компактным хешем, и они показывают хорошую производительность.

Техника иерархического хеширования для операции перекомпоновки

Еще один инновационный подход к использованию хеширования для операции перекомпоновки предусматривает иерархический набор хешей и метод «хлебных крошек» (т. е. навигационной цепочки). Цепочка из нескольких сигнальных значений имеет преимущество в том, что нам не нужно инициализировать хеш-таблицы сигнальным значением в самом начале (рис. 5.15).

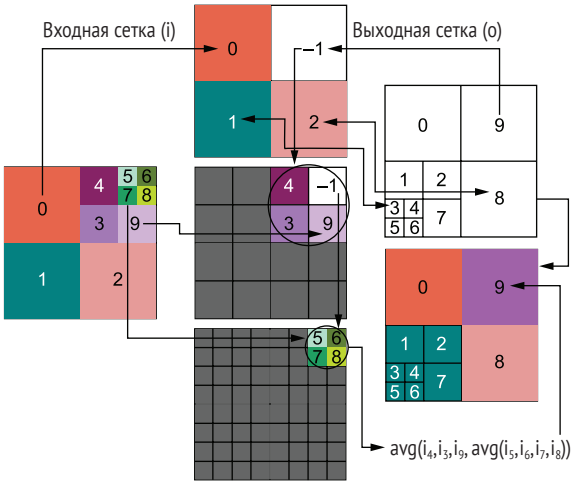


Рис. 5.15 Иерархическая хеш-таблица с отдельным хешем для каждого уровня. Когда запись делается на одном из более детализированных уровней, сигнальное значение помещается на каждом уровне выше, формируя след в виде «хлебной крошки», информирующий запросы о наличии данных на более детализированных уровнях

Первым шагом является выделение хеш-таблицы для каждого уровня сетки. Затем индексы ячеек записываются в хеш соответствующего уровня и рекурсивно проходятся вверх по более грубым хешам, оставляя сигнальное значение, чтобы запросы знали, что в хеш-таблицах более высокого уровня есть значения. Глядя на ячейку 9 входной сетки рис. 5.15, мы видим, что:

- индекс ячейки записывается в хеш-таблицу среднего уровня, затем сигнальное значение записывается в хеш-корзины в более грубой хеш-таблице;

- операция чтения ячейки 9 сначала переходит на самый грубый уровень хеш-таблицы, где находит значение –1. Теперь она знает, что должна перейти на более детализированные уровни;
- он находит три ячейки в хеш-таблице среднего уровня и еще одно сигнальное значение, сообщающее операции чтения рекурсивно спуститься на самый высокий уровень, где она находит еще четыре значения для добавления к сумме;
- все остальные запросы находятся в самой грубой хеш-таблице и в значениях, присвоенных выходной сетке.

Каждая хеш-таблица может быть либо идеальным хешем, либо компактным хешем. Указанный метод имеет рекурсивную структуру, аналогичную предыдущему техническому приему. Он также отлично работает на GPU-процессорах.

5.6 Шаблон префиксного суммирования (сканирования) и его важность в параллельных вычислениях

Префиксная сумма была важным элементом параллельной работы сортировки хешей в разделе 5.5.1. Операция префиксной суммы, также именуемая сканом, широко используется в вычислениях с нерегулярными размерами. Многие вычисления с нерегулярными размерами должны знать, откуда начинать запись, чтобы иметь возможность работать в параллельном режиме. Простой пример – это ситуация, когда каждый процессор имеет разное число частиц. Для того чтобы иметь возможность писать в выходной массив или обращаться к данным на других процессорах или в потоках процессора, каждый процессор должен знать связь локальных индексов с глобальными. В префиксной сумме выходной массив y представляет собой текущую сумму всех предшествующих ему чисел в исходном массиве:

$$y_j = \sum_{i=0}^{j-1} x_i.$$

Префиксная сумма может быть либо включительным (инклюзивным) сканом, в который включено текущее значение, либо исключительным (эксклюзивным) сканом, в который оно не включено. Приведенное выше уравнение предназначено для исключительного скана. На рис. 5.16 показаны как исключительный, так и включительный сканы. Исключительный скан является начальным индексом для глобального массива, тогда как включительный скан является конечным индексом для каждого процесса или потока.

В листинге 5.15 показан стандартный последовательный исходный код для операции сканирования.

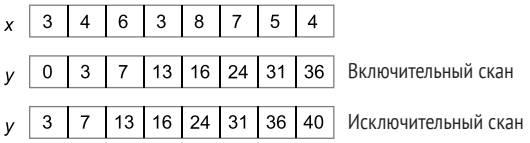


Рис. 5.16 Массив x дает число частиц в каждой ячейке. Исключительный и включительный скан массива дает начальный и конечный адреса в глобальном наборе данных

Листинг 5.15 Последовательная операция включительного сканирования

```
1 y[0] = x[0];
2 for (int i=1; i<n; i++){
3    y[i] = y[i-1] + x[i];
4 }
```

После завершения операции сканирования каждый процесс может выполнять свою работу параллельно, потому что процесс знает место, куда помещать свой результат. Однако сама операция сканирования, по-видимому, является по своей сути последовательной. Каждая итерация зависит от предыдущей. Однако существуют эффективные способы ее параллелизации. В этом разделе мы рассмотрим алгоритмы с эффективностью шагов, с эффективностью работы и с крупным массивом.

5.6.1 Операция параллельного сканирования с эффективностью шагов

В алгоритме с эффективностью шагов (step-efficient algorithm) используется наименьшее число шагов. Но это может быть не наименьшее число операций, потому что на каждом шаге возможно разное число операций. Данный вопрос обсуждался ранее при определении вычислительной сложности в разделе 5.1.

Операция префиксного суммирования может выполняться параллельно с помощью шаблона редукции на основе дерева, как показано на рис. 5.17. Вместо того чтобы ждать до тех пор, пока предыдущий элемент просуммирует свои значения, каждый элемент суммирует свое значение и предыдущее. Затем он выполняет ту же операцию, но со значением на два элемента выше, на четыре элемента выше и т. д. Конечным результатом является включительный скан; во время этой операции все процессы были заняты.

Теперь у нас есть параллельный префикс, который работает всего за $\log_2 n$ шагов, но объем работы увеличивается за счет последовательного алгоритма. Возможен ли дизайн параллельного алгоритма с таким же объемом работы?

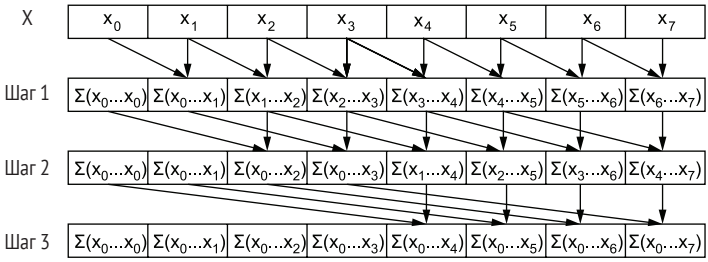


Рис. 5.17 Во включительном скане с эффективностью шагов для параллельного вычисления префиксной суммы используется $O(\log_2 n)$ шагов

5.6.2 Операция параллельного сканирования с эффективностью работы

В алгоритме с эффективностью работы (work-efficient algorithm) используется наименьшее число операций. Это может быть не наименьшее число шагов, потому что на каждом шаге возможно разное число операций. Решение использовать алгоритм с эффективностью работы либо с эффективностью шагов зависит от числа параллельных процессов, которые могут существовать.

В операции параллельного сканирования с эффективностью работы используются два витка по массивам. Первый виток называется восходящим, хотя он больше похож на правый виток. Это показано на рис. 5.18 сверху вниз, а не традиционным способом снизу вверх для упрощения сравнения с алгоритмом с эффективностью шагов.

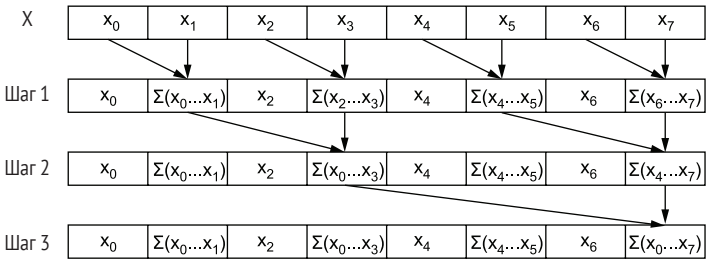


Рис. 5.18 Фаза восходящего витка скана с эффективностью работы, показанного сверху вниз, в котором выполняется гораздо меньше операций, чем при скане с эффективностью шагов. По сути, все остальные значения остаются неизменными

Вторая фаза, именуемая фазой нисходящего витка, больше похожа на левый виток. Она начинается с установки последнего значения, равного нулю, а затем выполняет еще один древовидный виток (рис. 5.19), получая окончательный результат. Объем работы значительно сокращается, при этом требуя большего числа шагов.

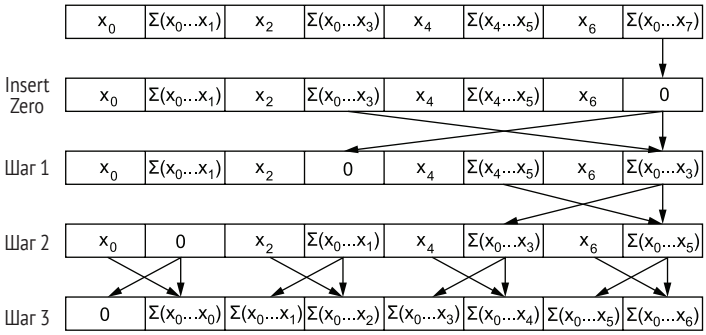


Рис. 5.19 В фазе нисходящего витка исключительного скана с эффективностью работы выполняется гораздо меньше операций, чем при скане с эффективностью шагов

При показе в таком ключе скан с эффективностью работы имеет интересный шаблон, когда правый виток начинается с половины потоков процессора и уменьшается до тех пор, пока не будет работать только один. Затем он начинает виток назад влево с одним потоком в начале и заканчивается занятостью всех потоков. Дополнительные шаги позволяют использовать более ранние вычисления повторно, вследствие чего суммарное число операций составляет только $O(N)$.

Указанные два алгоритма параллельного префиксного суммирования дают нам несколько разных вариантов задействования параллелизма в этой важной операции. Но оба они ограничены числом потоков, доступных в рабочей группе на GPU, либо числом процессоров на CPU.

5.6.3 Операции параллельного сканирования для крупных массивов

Для крупных массивов нам тоже нужен параллельный алгоритм. На рис. 5.20 показан такой алгоритм, в котором используется три вычислительных ядра для GPU. Первое ядро начинается с редукционной суммы для каждой рабочей группы и сохраняет результат во временном массиве, который меньше изначального крупного массива на число потоков в рабочей группе. В GPU число потоков в рабочей группе обычно достигает 1024. Затем второе ядро прокручивает временный массив в цикле, делая скан каждого блока размером с рабочую группу. Это приводит к тому, что временный массив теперь содержит сдвиги каждой рабочей группы. Затем вызывается третье ядро для выполнения операции сканирования на кусках изначального массива размером с рабочую группу и сдвиге, рассчитанном для каждого потока на этом уровне.

Поскольку параллельная префиксная сумма столь важна в таких операциях, как сортировка, она сильно оптимизирована под архитектуры GPU. В этой книге мы не будем вдаваться в подробности такого уровня. Вместо этого мы предлагаем разработчикам приложений использовать для своей работы библиотеки или свободно доступные имплементации.

Для параллельного префиксного сканирования, доступного для CUDA, можно найти такие имплементации, как библиотека параллельных примитивов CUDA (CUDPP), доступная по адресу https://github.com/cudpp/cudpp. Для OpenCL мы предлагаем имплементацию из библиотеки параллельных примитивов CLPP либо имплементацию сканирования из нашего кода сортировки хешем, доступного в файле sort_kern.cl по адресу https://github.com/LANL/PerfectHash.git. Мы представим версию префиксного сканирования для OpenMP в главе 7.

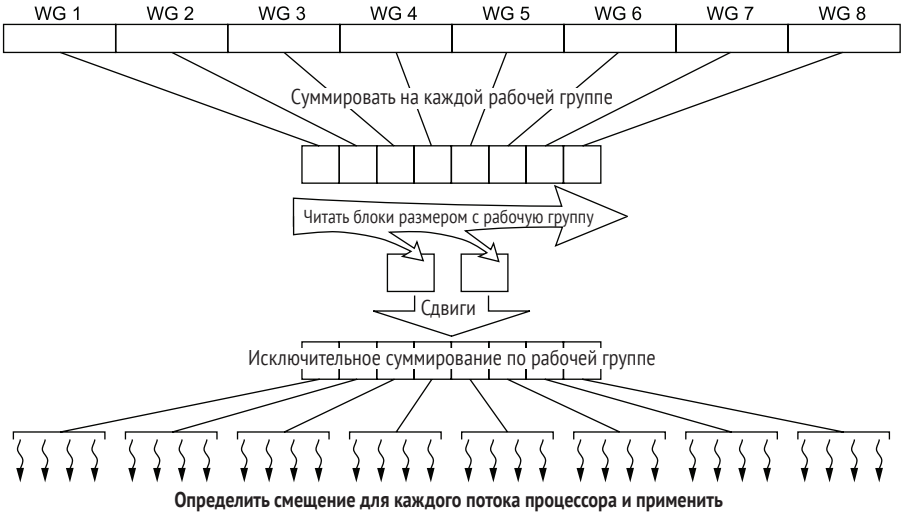


Рис. 5.20 Скан крупного массива выполняется в три фазы и в виде трех вычислительных ядер GPU. В первой фазе выполняется редукционная сумма, сводящая к промежуточному массиву. Во второй фазе делается скан для создания сдвигов рабочих групп. Затем в третьей фазе сканируется изначальный массив и применяет сдвиги рабочих групп, получая результаты сканов для каждого элемента массива

5.7 Параллельная глобальная сумма: решение проблемы ассоциативности

Не все параллельные алгоритмы направлены на ускорение вычислений. Глобальная сумма является ярким примером такого случая. Параллельные вычисления с самых ранних дней страдали невоспроизводимостью сумм между процессорами. В этом разделе мы покажем один пример алгоритма, который улучшает воспроизводимость параллельного вычисления, делая ее ближе к результатам исходного последовательного вычисления.

Изменение порядка операций сложения изменяет ответ в арифметике конечной точности (или конечной прецизионности). Это проблематично тем, что параллельная калькуляция изменяет порядок операций

сложения. Указанная проблема связана с тем, что арифметика конечной точности не является ассоциативной. И она усугубляется по мере увеличения размера задачи, потому что прибавление последнего значения становится все меньшей и меньшей частью общей суммы. В конечном счете прибавление последнего значения может вообще не изменить сумму. Еще более ухудшающий случай для операций сложения значений конечной точности возникает при сложении двух значений, которые почти идентичны, но имеют разные знаки. Вычитание одного значения из другого, когда они почти одинаковы, приводит к *катастрофической взаимоотмене*. В результате получается всего несколько значащих цифр, а остальное заполняет шум.

Пример: катастрофическая взаимоотмена

Вычитание двух почти одинаковых значений будет приводить к результату с малым числом значащих цифр. Поместите следующий ниже фрагмент кода в файл под названием catastrophic.py и выполните Python'овскую программу `catastrophic.py`.

Катастрофическая взаимоотмена в коротком коде Python

```
x = 12.15692174374373 - 12.15692174374372
print x ◄———————— Возвращает 1.06581410364e-14
```

Результат в этом примере имеет всего несколько значащих цифр! А откуда берутся остальные цифры в напечатанном значении? Проблема параллельных вычислений заключается в том, что вместо суммы, представляющей собой линейное сложение значений в массиве, сумма на двух процессорах представляет собой линейную сумму половины массива, а затем двух частичных сумм, сложенных в конце. Изменение порядка приводит к тому, что глобальная сумма будет другой. Разница может быть небольшой, но теперь вопрос в том, насколько правильно была выполнена параллелизация кода, если вообще она таковой является. Усугубляет проблему то, что все новые технические приемы параллелизации и оборудование, такие как векторизация и потоки, тоже вызывают эту проблему. Шаблон для этой операции глобальной суммы называется редукцией.

ОПРЕДЕЛЕНИЕ *Редукция* – это операция, при которой массив из одной или нескольких размерностей сокращается по меньшей мере на одну размерность и нередко сводится до скалярного значения.

В параллельных вычислениях эта операция является одной из наиболее распространенных и часто связана с производительностью, а в данном случае с правильностью. Примером тому является вычисление общей массы или энергии в задаче. Она берет глобальный массив массы в каждой ячейке и приводит к одному единственному скалярному значению.

Как и во всех компьютерных расчетах, результаты редукции глобальной суммы являются неточными. В последовательных расчетах это не представляет серьезной проблемы, потому что мы всегда получаем один и тот же неточный результат. В параллельном режиме мы, скорее всего, получим более точный результат с более правильными значащими цифрами, но он отличается от последовательного результата. Эта проблема носит название *проблемы глобальной суммы*. В любое время, когда результаты между последовательной и параллельной версиями немного отличались, причина была связана с указанной проблемой. Но нередко, когда тратилось время, чтобы углубиться в код, проблема оказывалась в едва уловимой ошибке параллельного программирования, такой как неспособность обновлять призрачные ячейки между процессорами. *Призрачные ячейки* – это ячейки, в которых хранятся значения соседних процессоров, необходимые локальному процессору, и если они не обновляются, то чуть-чуть более старые значения вызывают небольшую ошибку по сравнению с последовательным прогоном.

В течение многих лет я думал, как и другие параллельные программисты, что единственным решением была сортировка данных в фиксированном порядке и их суммирование в последовательной операции. Но, поскольку это обходилось слишком дорого, мы просто жили с этой проблемой. Примерно в 2010 году несколько параллельных программистов, включая меня, поняли, что мы неправильно смотрим на эту проблему. Это не только вопрос порядка, но и вопрос прецизионности (или точности). В арифметике действительных чисел сложение является ассоциативным! И следовательно, добавление прецизионности также является способом решения указанной проблемы и с гораздо меньшими затратами, чем сортировка данных.

В целях более глубокого понимания этой проблемы и способов ее решения давайте взглянем на задачу гидродинамики сжимаемой жидкости, именуемую задачей Леблана, также именуемую «ударной трубой из ада». В задаче Леблана область высокого давления отделяется от области низкого давления диафрагмой, которая устраняется в нулевое время. Данная задача очень трудна из-за сильного шока, который возникает в результате. Но нас больше всего интересует одна особенность, а именно большой *динамический диапазон* как по переменной плотности, так и по переменной энергии. Мы будем использовать переменную энергии с высоким значением 1.0e–1 и низким значением 1.0e–10. Динамический диапазон – это диапазон рабочего набора действительных чисел, или – в данном случае – соотношение максимального и минимального значений. Динамический диапазон составляет девять порядков величины, означая, что при прибавлении малого значения к крупному для двойной точности числу с плавающей точкой примерно с 16 значащими цифрами у нас на самом деле в результате будет всего около 7 значащих цифр.

Давайте посмотрим на размер 134 217 728 задачи на одиночном процессоре с половиной значений в состоянии высокой энергии, а другой половиной в состоянии низкой энергии. Эти два участка в начале задачи

разделены диафрагмой. Размер задачи велик для одного процессора, но для параллельных вычислений он относительно невелик. Если сначала суммируются высокие значения энергии, то следующее прибавленное низкое значение будет содержать несколько значащих цифр. Изменение порядка суммы в таком ключе, чтобы сначала суммировались значения с низкой энергией, приводит к тому, что малые значения в сумме будут иметь почти одинаковый размер, и к тому времени, когда будет добавлено значение с высокой энергией, значащих цифр будет больше, что позволит получать более точную сумму. Этот факт дает нам возможное решение на основе сортировки. Просто отсортируйте значения по порядку от наименьшей величины до наибольшей, и вы получите более точную сумму. Существует несколько технических решений глобальной суммы, которые гораздо удобнее, чем метод сортировки. Список представленных здесь возможных технических приемов таков:

- тип данных long double;
- попарное суммирование;
- суммирование по Кахану;
- суммирование по Кнуту;
- суммирование с четверной точностью (прецизионностью).

Вы можете попробовать эти различные методы в сопровождающих главу упражнениях по адресу https://github.com/EssentialsOfParallelComputing/Chapter5.git. В первоначальном исследовании рассматривались параллельные имплементации OpenMP и технические приемы усечения, которые мы здесь обсуждать не будем.

Самое простое решение состоит в использовании типа данных long double в архитектуре x86. В этой архитектуре long double имплементирован в виде 80-битового числа с плавающей точкой в оборудовании, обеспечивающем дополнительную 16-битовую прецизионность. К сожалению, этот технический прием непереносим. Тип long double в некоторых архитектурах и компиляторах составляет всего 64 бита, а на других – 128 бит и имплементируется в программном обеспечении. Некоторые компиляторы также требуют округления между операциями с целью обеспечения согласованности с другими архитектурами. При использовании этого технического приема следует внимательно ознакомиться с документацией вашего компилятора в отношении того, как в нем имплементирован тип long double. Исходный код, показанный в следующем ниже листинге, представляет собой обычную сумму с типом данных аккумулятора, установленным равным long double.

Листинг 5.16 Сумма с типом данных long double на архитектурах x86

GlobalSums/do_ldsum.c

```
1 double do_ldsum(double *var, long ncells)
2 {
3    long double ldsum = 0.0;
4    for (long i = 0; i < ncells; i++){
5       ldsum += (long double)var[i];
```

var – это массив значений типа double, тогда как accumulator имеет тип long double

```
6    }
7    double dsum = ldsum;
8    return(dsum);
9 }
```

Возвращаемой из функции тип также может быть long double, и возвращается значение ldsum

Возвращает double

В строке 8 листинга возвращается значение типа double, чтобы соответствовать идее накопителя с более высокой прецизионностью, возвращающего тот же тип данных, что и массив. Позже мы увидим, как это работает, но сначала давайте рассмотрим другие методы решения задачи глобальной суммы.

Попарное суммирование – это удивительно простое решение задачи глобальной суммы, в особенности в рамках одинарного процессора. Исходный код относительно прост, как показано в следующем ниже листинге, но требует дополнительного массива размером в половину изначального.

Листинг 5.17 Попарное суммирование на процессоре

GlobalSums/do_pair_sum.c

```
 4 double do_pair_sum(double *var, long ncells)
 5 {
 6    double *pwsum =
       (double *)malloc(ncells/2*sizeof(double));
 7
 8    long nmax = ncells/2;
 9    for (long i = 0; i<nmax; i++){
10       pwsum[i] = var[i*2]+var[i*2+1];
11    }
12
13    for (long j = 1; j<log2(ncells); j++){
14       nmax /= 2;
15       for (long i = 0; i<nmax; i++){
16          pwsum[i] = pwsum[i*2]+pwsum[i*2+1];
17       }
18    }
19    double dsum = pwsum[0];
20    free(pwsum);
21    return(dsum);
22 }
```

Требует временного пространства для выполнения попарных рекурсивных сумм

Добавляет начальную попарную сумму в новый массив

Рекурсивно суммирует оставшиеся $\log_2$ шагов, вдвое уменьшая размер массива на каждом шаге

Присваивает результат скалярному значению для его возвращения

Освобождает временное пространство

Простота попарного суммирования становится немного сложнее при работе с разными процессорами. Если алгоритм остается верным своей базовой структуре, на каждом шаге рекурсивной суммы может потребоваться обмен данными.

Далее следует суммирование по Кахану. Суммирование по Кахану является наиболее практичным методом из возможных методов глобальной суммы. В нем используется дополнительная переменная типа double для выполнения оставшейся части операции, что фактически удваивает эффективную прецизионность. Этот технический прием был разработан

Уильямом Каханом в 1965 году (позже Кахан стал одним из ключевых авторов ранних стандартов IEEE с плавающей точкой). Суммирование по Кахану наиболее подходит для скользящего суммирования, когда накопитель имеет большее из двух значений. Указанный технический прием показан в следующем ниже листинге.

Листинг 5.18 Суммирование по Кахану

GlobalSums/do_kahan_sum.c

```
 1 double do_kahan_sum(double *var, long ncells)
 2 {
 3    struct esum_type{                    Объявляет тип данных
 4       double sum;                       с типом double-double
 5       double correction;
 6    };
 7
 8    double corrected_next_term, new_sum;
 9    struct esum_type local;
10
11    local.sum = 0.0;
12    local.correction = 0.0;
13    for (long i = 0; i < ncells; i++) {
14       corrected_next_term = var[i] + local.correction;
15       new_sum          = local.sum + local.correction;
16       local.correction = corrected_next_term -     Вычисляет остаток для переноса
                            (new_sum - local.sum);    на следующую итерацию
17       local.sum         = new_sum;
18    }
19
20    double dsum = local.sum + local.correction;     Возвращает результат
21    return(dsum);                                   двойной точности
22 }
```

Суммирование по Кахану занимает около четырех операций с плавающей точкой вместо одной. Но данные могут храниться в регистрах или кеше L1, что делает операцию менее дорогостоящей, чем мы могли бы изначально ожидать. Векторизованные имплементации могут делать стоимость операции такой же, как при стандартном суммировании. Это пример подхода к использованию избыточных возможностей процессора с плавающей точкой, чтобы получать более качественный ответ.

Мы рассмотрим векторную имплементацию суммы по Кахану в разделе 6.3.4. Некоторые новые численные методы пытаются применять аналогичный подход, используя избыточные возможности современных процессоров с плавающей точкой. Они рассматривают текущий машинный баланс в 50 флопов на загрузку данных как возможность и имплементируют методы более высокого порядка, которые требуют большего числа операций с плавающей точкой для применения неиспользуемого ресурса с плавающей точкой, поскольку он, по сути, является бесплатным.

Метод суммирования по Кнуту манипулирует сложениями, в которых любой член может быть больше. Этот технический прием был разработан Дональдом Кнутом в 1969 году. Он собирает ошибку для обоих членов по стоимости семи операций с плавающей точкой, как показано в следующем ниже листинге.

Листинг 5.19 Суммирование по Кнуту

GlobalSums/do_knuth_sum.c

```
 1 double do_knuth_sum(double *var, long ncells)
 2 {
 3    struct esum_type{
 4       double sum;
 5       double correction;
 6    };
 7
 8    double u, v, upt, up, vpp;
 9    struct esum_type local;
10
11    local.sum = 0.0;
12    local.correction = 0.0;
13    for (long i = 0; i < ncells; i++) {
14       u = local.sum;
15       v = var[i] + local.correction;
16       upt = u + v;
17       up = upt - v;
18       vpp = upt - up;
19       local.sum = upt;
20       local.correction = (u - up) + (v - vpp);
21    }
22
23    double dsum = local.sum + local.correction;
24    return(dsum);
25 }
```

Объявляет тип данных с типом double-double

Переносит значения для каждого члена

Объединяется в одну поправку

Возвращает результат двойной точности

Последний технический прием, сумма с четверной точностью (прецизионностью), имеет преимущество в простоте кодирования, но поскольку типы с четверной точностью почти всегда выполняются в программном обеспечении, он обходится дорого. К тому же всегда следует учитывать переносимость, поскольку не все компиляторы имплементировали тип четверной точности. Указанный исходный код представлен в следующем ниже листинге.

Листинг 5.20 Глобальная сумма с четырехкратной точностью

GlobalSums/do_qdsum.c

```
1 double do_qdsum(double *var, long ncells)
2 {
3    __float128 qdsum = 0.0;
```

Тип данных с четвертой точностью

```
4    for (long i = 0; i < ncells; i++){
5       qdsum += (__float128)var[i];
6    }
7    double dsum =qdsum;
8    return(dsum);
9 }
```

Преобразовывает входное значение из массива в четверную точность

Теперь давайте перейдем к оцениванию качества работы этих разных подходов. Поскольку половина значений равна 1.0 e–1, а другая половина – 1.0e–10, мы можем получать точный ответ, относительно которого можно проводить сравнения, умножая, а не складывая:

```
accurate_answer = ncells/2 * 1.0e-1 + ncells/2 * 1.0e-10
```

В табл. 5.1 приведены результаты сравнения фактически полученных значений глобальной суммы с точным ответом и измерения времени выполнения. По сути, мы получаем прецизионность в девять цифр при регулярном суммировании значений типа double. Тип long double в системе с 80-битовым представлением с плавающей точкой несколько ее улучшает, но не полностью устраняет ошибку. Попарное суммирование по Кахану и Кнуту сводит ошибку к нулю со скромным увеличением времени выполнения. Векторизованная имплементация суммирования по Кахану и Кнуту (показано в разделе 6.3.4) устраняет увеличение времени выполнения. Тем не менее, учитывая межпроцессорное взаимодействие и стоимость вызовов MPI, увеличение времени выполнения является незначительным.

Теперь, когда мы понимаем поведение технических приемов глобальной суммы на процессоре, можем рассмотреть задачу, когда массивы распределены по нескольким процессорам. Для решения указанной задачи нам нужно некоторое понимание MPI, поэтому мы покажем, как это сделать в разделе 8.3.3, после изучения основ MPI.

Таблица 5.1 Результаты прецизионности и времени выполнения для разных технических приемов глобального суммирования

Метод	Ошибка	Время выполнения
Тип double	–1.99e–09	0.116
Тип long double	–1.31e–13	0.118
Попарное суммирование	0.0	0.402
Суммирование по Кахану	0.0	0.406
Суммирование по Кнуту	0.0	0.704
Тип четверной double	5.55e–17	3.010

5.8 Будущие исследования параллельных алгоритмов

Мы увидели несколько характеристик параллельных алгоритмов, в том числе те, которые подходят для чрезвычайно параллельных архитектур. Давайте их обобщим, чтобы иметь возможность их искать в других ситуациях.

- *Локальность* – часто используемый термин в описании хороших алгоритмов, но без какого-либо определения. Он может иметь несколько значений. Вот пара из них:
  - *локальность для кеша* – держит значения, которые будут использоваться вместе близко друг к другу, чтобы улучшать объем полезного использования кеша;
  - *локальность для операций* – позволяет избегать работы со всеми данными, когда не все они необходимы. Пространственный хеш для взаимодействий частиц является классическим примером, который поддерживает сложность алгоритма на уровне $O(N)$ вместо $O(N^2)$.
- *Асинхронность* – позволяет избегать координации между потоками, которая может приводить к синхронизации.
- *Меньше условных переходов* – помимо дополнительной производительности за счет условной логики, проблемой в некоторых архитектурах бывает поточное (thread) расхождение.
- *Воспроизводимость* – нередко сильно параллельный технический прием нарушает отсутствие ассоциативности арифметики конечной точности (прецизионности). Технические приемы повышенной точности помогают справляться с этой проблемой.
- *Более высокая арифметическая интенсивность* – современные архитектуры добавили возможность работы с плавающей точкой быстрее, чем пропускная способность памяти. Алгоритмы, которые повышают арифметическую интенсивность, могут эффективно задействовать параллелизм, такой как векторные операции.

5.9 Материалы для дальнейшего изучения

Разработка параллельных алгоритмов все еще является молодой областью исследований, и еще предстоит открыть много новых алгоритмов. Но есть и много известных технических приемов, которые не получили широкого распространения или не использовались. Особенно сложной проблемой является то, что алгоритмы часто находятся в совершенно разных областях компьютерной или вычислительной науки.

5.9.1 Дополнительное чтение

Для получения дополнительной информации об алгоритмах мы рекомендуем популярный учебник:

- Томас Кормен и соавт., «Введение в алгоритмы, 3-е изд.» (Thomas Cormen, et al., *Introduction to Algorithms*, 3rd ed, MIT Press, 2009).

Для получения дополнительной информации о шаблонах и алгоритмах мы рекомендуем обратиться к двум неплохим книгам в качестве дальнейшего чтения.

- Майкл Маккул, Арч Д. Робисон и Джеймс Рейндерс, «Структурированное параллельное программирование: шаблоны для эффективных вычислений» (Michael McCool, Arch D. Robison, James Reinders, *Structured Parallel Programming: Patterns for Efficient Computation* (Morgan Kaufmann, 2012).
- Тимоти Г. Мэттсон, Беверли А. Сандерс и Берна Л. Массингилл, «Шаблоны параллельного программирования» (Timothy G. Mattson, Beverly A. Sanders, Berna L. Massingill, *Patterns for Parallel Programming*, Addison-Wesley, 2004).

Концепции пространственного хеширования были разработаны несколькими моими учениками, начиная с уровня средней школы и заканчивая аспирантурой. Раздел об идеальном хешировании в следующем ниже ресурсе взят из работы Рэйчел Роби и Дэвида Николаффа. Дэвид также имплементировал пространственное хеширование в мини-приложении CLAMR.

- Рэйчел Н. Роби, Дэвид Николаефф и Роберт У. Роби, «Алгоритмы на основе хеша для дискретизированных данных» (Rachel N. Robey, David Nicholaeff, Robert W. Robey, Hash-based algorithms for discretized data, *SIAM Journal on Scientific Computing* 35, no. 4 (2013): C346–C368).

Идеи параллельного компактного хеширования для отыскания соседей исходили от Ребекки Тамблин, Питера Аренса и Сары Харце. Они были осуществлены на основе методов, призванных редуцировать операции записи и чтения, разработанных Дэвидом Николаеффом.

- Ребекка Тумблин, Питер Аренс и соавт., «Алгоритмы параллельного компактного хеширования для вычислительных сеток», (Rebecka Tumblin, Peter Ahrens, et al., Parallel compact hash algorithms for computational meshes, *SIAM Journal on Scientific Computing* 37, no. 1 (2015): C31–C53).

Задача разработки оптимизированных методов для операции перекомпоновки была гораздо труднее. Джеральд Коллом и Колин Редман решили эту задачу и разработали несколько действительно инновационных технических приемов и имплементаций на GPU и в OpenMP. В этой главе рассматриваются только некоторые из них. В их статье идей гораздо больше:

- Джеральд Коллом, Колин Редман и Роберт У. Роби, «Быстрые перекомпоновки из сетки в сетку с использованием алгоритмов хеши-

рования» (Gerald Collom, Colin Redman, Robert W. Robey, Fast Mesh-to-Mesh Remaps Using Hash Algorithms, *SIAM Journal on Scientific Computing* 40, no. 4 (2018): C450–C476).

Я впервые разработал концепцию глобальных сумм повышенной точности (прецизионности) примерно в 2010 году. Джонатан Роби внедрил эту технику в свой Sapient hydrocode (Разумный гидрокод), а Роб Олвес из Лос-Аламосской национальной лаборатории помог разработать теоретические основы. Следующие две ссылки дают более подробную информацию об указанном методе.

- Роберт У. Роби, Джонатан М. Роби и Роб Олвес, «В поисках численной согласованности в параллельном программировании» (Robert W. Robey, Jonathan M. Robey, Rob Aulwes, In search of numerical consistency in parallel programming, *Parallel Computing* 37, no. 4–5 (2011): 217–229).
- Роберт У. Роби, «Вычислительная воспроизводимость в приложениях производственной физики» (Robert W. Robey, Computational Reproducibility in Production Physics Applications, Numerical Reproducibility at Exascale Workshop (NRE2015), International Conference for High Performance Computing, Networking, Storage and Analysis, 2015). Работа доступна по адресу https://github.com/lanl/ExascaleDocs/blob/master/ComputationalReproducibilityNRE2015.pdf.

5.9.2 Упражнения

1. Модель столкновения облаков в шлейфе пепла вызывается для частиц на расстоянии в пределах 1 мм. Напишите псевдокод для имплементации пространственного хеша. В каком порядке сложности выполняется эта операция?
2. Как пространственные хеши используются почтовой службой?
3. Большие данные используются алгоритмом map-reduce (отображения-редукции) для эффективной обработки крупных наборов данных. Чем он отличается от представленных здесь концепций хеширования?
4. В исходном коде волновой симуляции используется сетка AMR для более глубокой детализации береговой линии. Требования к симуляции заключаются в регистрации высоты волн в зависимости от времени для заданных мест, где расположены буи и береговые сооружения. Поскольку ячейки постоянно детализируются, как вы могли бы ее имплементировать?

Резюме

- Алгоритмы и шаблоны являются одной из основ вычислительных приложений. Выбор алгоритмов, которые имеют низкую вычислительную сложность и поддаются параллелизации, важен с самого начала разработки приложения.

- Алгоритм, основанный на сравнении, имеет нижний предел сложности $O(N \log N)$. Алгоритмы без сравнений могут преодолевать этот нижний алгоритмический предел.
- Хеширование – это технический прием без сравнений, который использовался в пространственном хешировании для достижения $\Theta(N)$-й сложности для пространственных операций.
- Для любой пространственной операции существует алгоритм пространственного хеширования, который масштабируется как $O(N)$. В этой главе мы приводим примеры технических приемов, которые могут использоваться во многих сценариях.
- Было показано, что некоторые шаблоны могут быть адаптированы к параллелизму и асинхронной природе GPU-процессоров. Технические приемы префиксного сканирования и хеширования являются двумя такими шаблонами. Префиксный скан важен для параллелизации массивов нерегулярного размера. Хеширование представляет собой асинхронный алгоритм без сравнений, который обладает высокой масштабируемостью.
- Воспроизводимость является важным атрибутом при разработке устойчивых производственных приложений. Она особенно важна для воспроизводимых глобальных сумм и для работы с арифметическими операциями конечной прецизионности (точности), которые не являются ассоциативными.
- Повышенная прецизионность представляет собой новый технический прием, который восстанавливает ассоциативность, позволяя переупорядочивать операции и, следовательно, обеспечивает больше параллелизма.

Часть II

CPU: параллельная рабочая лошадка

Сегодня каждый разработчик должен понимать наличие растущих возможностей параллелизма, имеющегося в современных центральных процессорах (CPU). Уметь раскрывать неиспользованную производительность CPU-процессоров критически важно для параллельных приложений и приложений высокопроизводительных вычислений. В целях ознакомления с тем, как пользоваться преимуществами параллелизма CPU мы рассмотрим:

- использование векторного оборудования;
- использование потокообразования для параллельной работы на мультиядерных процессорах;
- координирование работы на нескольких CPU- и мультиядерных процессорах с передачей сообщений.

Параллельные возможности CPU должны лежать в основе вашей параллельной стратегии. Поскольку CPU является центральной рабочей лошадкой, он контролирует все распределение и перемещение памяти, а также обмен данными. Знания и умения разработчика приложений являются наиболее важными факторами для полного использования параллелизма CPU. Оптимизация CPU не делается автоматически каким-то волшебным компилятором. Как правило, в приложениях не используются многие имеющиеся в CPU параллельные ресурсы. Мы можем разбить имеющийся параллелизм CPU на три компонента в порядке возрастания усилий. Они таковы:

- *векторизация* – использует специализированное оборудование, которое может выполнять более одной операции за раз;
- *мультиядерность и потокообразование* – распределяет работу между многочисленными процессорными ядрами в современных CPU;
- *распределенная память* – объединяет несколько узлов в единое кооперативное вычислительное приложение.

Таким образом, мы начинаем с векторизации. Векторизация – это сильно недоиспользуемая способность с заметными преимуществами при имплементировании. Хотя компиляторы способны выполнять некоторую векторизацию, они делают это недостаточно полно. Ограничения особенно заметны в случае усложненного кода. Там компиляторов просто еще нет. Хотя компиляторы совершенствуются, фондируемых средств или рабочей силы недостаточно, чтобы это произошло быстро. Следовательно, программист приложения должен способствовать этому самыми разными способами. К сожалению, векторизации посвящено мало документации. В главе 6 мы представим введение в тайные знания о том, как получать от векторизации больше для вашего приложения.

С взрывным ростом числа процессорных ядер на каждом CPU потребность и знания в использовании параллелизма на узлах быстро возрастает. Два широко используемых для этого ресурса CPU включают потокообразование и совместную память. Существуют десятки разных систем потокообразования и подходов к обеспечению совместной памяти. В главе 7 мы представим руководство по использованию OpenMP, наиболее широко используемого пакета потокообразования для высокопроизводительных вычислений.

Доминирующим языком параллелизма между узлами и даже внутри узлов является стандарт с открытым исходным кодом MPI (Интерфейс передачи сообщений). Стандарт MPI вырос из объединения многочисленных библиотек передачи сообщений с первых дней параллельного программирования. MPI представляет собой хорошо проработанный язык, который выдержал испытание временем и изменениями в архитектуре оборудования. Он также адаптировался с использованием новых функциональностей и улучшений, которые были включены в его имплементацию. Тем не менее большинство прикладных программистов просто использует самые базовые функции языка. В главе 8 мы даем введение в основы MPI, а также некоторые расширенные функциональности, которые могут быть полезны во многих научных приложениях и приложениях с большими данными.

Ключом к достижению высокой производительности CPU является концентрация внимания на пропускной способности памяти при передаче данных параллельным двигателям. Хорошая параллельная производительность начинается с хорошей последовательной производительности (и понимания тем, представленных в первых пяти главах этой книги). CPU-процессоры обеспечивают наиболее общий параллелизм для самых разнообразных приложений. CPU нередко предлагает все выгоды параллелизма от скромного масштаба до экстремального. CPU, кроме того, является местом, где вы должны начинать свое путешествие

в параллельный мир. Даже в решениях, в которых используются ускорители, CPU остается важным компонентом системы.

До сих пор решением для повышения производительности было увеличение вычислительной мощности в форме физического добавления большего числа узлов в ваш кластер или высокопроизводительный компьютер. Сообщество параллельных и высокопроизводительных вычислений зашло с таким подходом настолько далеко, насколько это возможно, и начинает выходить за пределы мощности и энергопотребления. Кроме того, число узлов и процессоров не может продолжать расти, не сталкиваясь с пределами масштабирования приложений. В ответ на это мы должны обратиться к другим способам повышения производительности. В узле обработки имеется много недоиспользуемых параллельных аппаратных способностей. Как было впервые упомянуто в разделе 1.1, параллелизм внутри узла будет продолжать расти.

Даже при сохраняющихся пределах вычислительной мощности и других надвигающихся порогах ключевые идеи и знания о менее известных инструментах могут обеспечивать значительную производительность. Благодаря этой книге и вашим исследованиям мы поможем решить эти проблемы. В конце концов, ваши умения и знания являются важными ценностями для раскрытия обещанного потенциала параллельной производительности.

Примеры, сопровождающие три главы части II этой книги, находятся по адресу https://github.com/EssentialsofParallelComputing с отдельным хранилищем для каждой главы. Контейнерные сборки Docker для каждой главы должны хорошо инсталлироваться и работать в любой операционной системе. В контейнерных сборках для первых двух глав этой части (главы 6 и 7) задействуется графический интерфейс, позволяющий использовать инструменты обеспечения производительности и правильности.

6 Векторизация: флопы забесплатно

Эта глава охватывает следующие ниже темы:

- важность векторизации;
- вид параллелизации, обеспечиваемый модулем векторной обработки;
- различные способы доступа к векторной параллелизации;
- ожидаемые преимущества производительности.

Процессоры имеют специальные модули векторной обработки, которые могут загружать и обрабатывать более одного элемента данных за один раз. Если мы ограничены операциями с плавающей точкой, то использовать векторизацию для достижения максимальных аппаратных возможностей просто необходимо. *Векторизация* – это процесс группировки операций, чтобы иметь возможность выполнять несколько операций за один раз. Но добавление большего числа флопов к способностям оборудования в ситуации, когда приложение привязано к памяти, имеет ограниченные выгоды. Следует учитывать, что большинство приложений так или иначе привязано к памяти. Компиляторы могут быть мощными, но, как вы увидите сами, достичь реального прироста производительности при векторизации бывает не так-то просто, как предполагает документация компилятора. Тем не менее прирост производительности от оптимизации можно достичь малыми усилиями, и его не следует игнорировать.

В этой главе мы покажем, как программисты, приложив немного усилий и знаний, могут повышать производительность за счет векторизации. Некоторые из этих методов требуют лишь использования правильных компиляторных флагов и стилей программирования, в то время как другие требуют гораздо большей работы. Примеры из реального мира демонстрируют различные способы достижения векторизации.

ПРИМЕЧАНИЕ Мы рекомендуем вам сверяться с примерами этой главы, расположенными по адресу https://github.com/EssentialsofParallelComputing/Chapter6.

6.1 Векторизация и обзор SIMD (одна команда, несколько элементов данных)

В разделе 1.4 мы ввели архитектуру с одной командой и несколькими элементами данных (SIMD) в качестве одного из компонентов таксономии Флинна. Указанная таксономия используется в качестве параллелизационной классификации потоков (streams) команд и данных в архитектуре. В случае SIMD, как следует из названия, существует одна команда, которая исполняется в нескольких потоках данных. Одна векторная команда add заменяет восемь отдельных скалярных команд add в очереди команд, что сокращает давление на очередь команд и кеш. Самая большая выгода заключается в том, что для выполнения восьми сложений в модуле векторной обработки требуется примерно та же мощность, что и для одного скалярного сложения. На рис. 6.1 показан векторный модуль, который имеет 512-битовую векторную ширину, предлагающую длину вектора из восьми значений двойной точности.

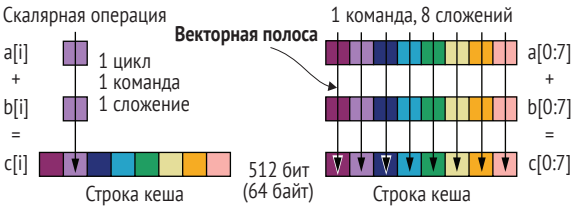


Рис. 6.1 Скалярная операция выполняет одно сложение с двойной точностью за один цикл. Для обработки 64-байтной строки кеша требуется восемь циклов. Для сравнения, векторная операция на 512-битном векторном модуле может обрабатывать все восемь значений двойной точности за один цикл

Давайте кратко подытожим терминологию векторизации:

- *векторная (SIMD) полоса* – маршрут через векторную операцию на векторных регистрах для одного элемента данных, очень похожий на полосу движения на многополосной автостраде;

- *ширина вектора* – ширина векторного модуля, обычно выражаемая в битах;
- *длина вектора* – число элементов данных, которые могут обрабатываться вектором за одну операцию;
- *наборы векторных команд (SIMD)* – набор команд, которые расширяют обычные команды скалярного процессора за счет использования векторного процессора.

Векторизация производится как посредством программного, так и аппаратного компонента. Требования таковы:

- *генерировать команды* – векторные команды должны генерироваться компилятором либо указываться вручную посредством внутренних функций компилятора[1] или ассемблерного кодирования;
- *соотносить команды с векторным модулем процессора* – если есть несоответствие между командами и оборудованием, то более новое оборудование обычно будет способно обрабатывать команды, но более старое оборудование просто откажется работать. (Команды AVX не выполняются на чипах десятилетней давности. Извините!)

Нет никакого причудливого процесса, который на лету преобразовывает обычные скалярные команды. Если вы используете более старую версию своего компилятора, как это делают многие программисты, у него не будет возможности генерировать команды для новейшего оборудования. К сожалению, авторам компиляторов требуется время, чтобы включить новые аппаратные способности и наборы команд. Авторам компиляторов также может потребоваться некоторое время на то, чтобы оптимизировать эти способности.

> *Вывод: при использовании новейших процессоров следует использовать только последние версии компилятора.*

Вы также должны указывать соответствующий набор генерируемых векторных команд. По умолчанию большинство компиляторов выбирает безопасный путь и генерирует команды SSE2 (Потоковые расширения SIMD), чтобы код работал на любом оборудовании. Команды SSE2 выполняют только две операции двойной точности за один раз вместо четырех или восьми операций, которые можно выполнять на более современных

---

[1] Внутренние, или встроенные, функции компилятора (intrinsics) похожи на привычные библиотечные функции за исключением того, что они встроены в компилятор. Они бывают быстрее, чем обычные библиотечные функции (компилятор знает о них больше и оптимизирует их лучше), либо обрабатывают меньший диапазон входных данных, чем библиотечные функции. Встроенные функции также предоставляют процессорно-специфичные функциональности, и поэтому их можно использовать в качестве промежуточного звена между стандартным языком C и машинно-ориентированным ассемблерным языком. – *Прим. перев.*

процессорах. Для приложений с высокой производительностью существуют более совершенные варианты.

- Вы можете компилировать для архитектуры, на которой работаете.
- Вы можете компилировать для любой архитектуры, произведенной за последние 5 или 10 лет. Указание инструкций AVX (Продвинутые векторные расширения) даст вектор 256-битной ширины и будет работать на любом оборудовании начиная с 2011 года.
- Вы можете попросить компилятор генерировать более одного набора векторных команд. Тогда он прибегает к наилучшему для используемого оборудования.

  Вывод: указывайте наиболее продвинутый набор векторных команд в компиляторных флагах, которые можно разумно использовать.

6.2 Аппаратные тренды векторизации

В целях имплементирования рассмотренных ранее вариантов полезно знать исторические даты выпуска оборудования и набора команд, чтобы выбрать тот набор векторных команд, который вы будете использовать. В табл. 6.1 приведены основные выпуски, а на рис. 6.2 показаны тренды в размере модуля векторной обработки.

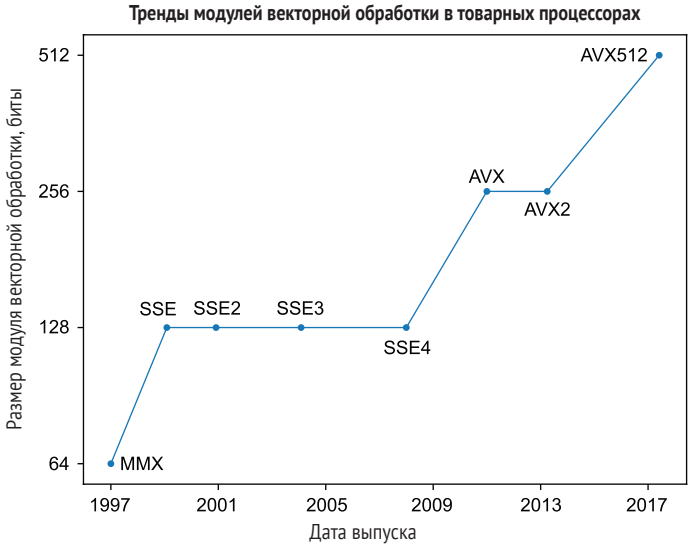


Рис. 6.2 Появление оборудования в виде векторных модулей для товарных процессоров началось примерно в 1997 году и за последние 20 лет медленно росло как по ширине (размеру) вектора, так и по типам поддерживаемых операций

Таблица 6.1 Выпуски векторного оборудования за последнее десятилетие значительно улучшили векторную функциональность

Выпуск	Функциональность
MMX (торговая марка без официального содержания)	Был нацелен на рынок графики, но GPU-процессоры вскоре взяли эту функцию на себя. Вместо графики векторные модули переключились на вычисления. AMD выпустила свою версию под названием 3DNow! с поддержкой одинарной точности
SSE (Потоковые расширения SIMD)	Первый векторный модуль Intel, предлагающий операции с плавающей точкой с поддержкой одинарной точности
SSE2	Добавлена поддержка двойной точности
AVX (Продвинутые векторные расширения)	Длина вектора увеличена вдвое. AMD добавила векторную команду слитного умножения-сложения FMA в свое конкурирующее оборудование, эффективно удвоив производительность для некоторых циклов
AVX2	Intel добавила слитное умножение-сложение (FMA) в свой векторный процессор
AVX512	Впервые предложенный на процессоре Knights Landing, он появился в линейке мультиядерных процессоров главной линейки в 2017 году. Начиная с 2018 года и далее, Intel и AMD (Advanced Micro Devices, Inc.) создали несколько вариантов AVX512 в качестве дополнительных улучшений векторных аппаратных архитектур

6.3 Методы векторизации

Существует несколько способов достижения векторизации в вашей программе. В порядке возрастания усилий программиста к ним относятся:

- оптимизированные библиотеки;
- автоматическая векторизация;
- подсказки компилятору;
- внутренние векторные функции компилятора;
- машинно зависимые ассемблерные команды.

6.3.1 Оптимизированные библиотеки обеспечивают производительность за счет малых усилий

В целях минимизации усилий по векторизации программистам следует изучить доступные библиотеки, которые можно использовать для своего приложения. Многие низкоуровневые библиотеки предоставляют программистам, стремящимся к производительности, высокооптимизированные процедуры. Некоторые из наиболее часто используемых библиотек таковы:

- BLAS (Базовая линейно-алгебраическая система) – базовый компонент высокопроизводительного линейно-алгебраического программного обеспечения;
- LAPACK – линейно-алгебраический пакет;
- SCALAPACK – масштабируемый линейно-алгебраический пакет;
- FFT (Быстрое преобразование Фурье) – имеются различные имплементационные пакеты;
- разреженные решатели – имеются различные имплементации разреженных решателей.

В библиотеке математического ядра Intel® (Math Kernel Library, MKL) имплементированы оптимизированные версии BLAS, LAPACK, SCALAPACK, FFT, разреженных решателей и математических функций для процессоров Intel. Хотя указанная библиотека доступна в составе некоторых коммерческих пакетов Intel, она также предлагается бесплатно. Многие другие разработчики библиотек выпускают пакеты для целого ряда целей. Кроме того, поставщики оборудования предоставляют библиотеки, оптимизированные под свое собственное оборудование в рамках разных лицензионных соглашений.

6.3.2 Автоматическая векторизация: простой способ ускорения векторизации (в большинстве случаев)

Автоматическая векторизация[1] является рекомендуемым вариантом выбора для большинства программистов, поскольку ее имплементация требует наименьших усилий по программированию. При этом компиляторы не всегда могут распознавать места, где векторизация может применяться безопасно. В этом разделе мы сначала рассмотрим исходный код, который компилятор способен векторизовывать автоматически. Затем мы покажем, как получать подтверждение о том, что вы на самом деле имеете векторизацию, которую ожидаете. Вы также узнаете о стилях программирования, которые позволяют компилятору векторизировать код и выполнять другие оптимизации. Сюда входит использование ключевого слова `restrict` для C и атрибутов `__restrict` или `__restrict__` для C++.

При непрекращающемся совершенствовании архитектур и компиляторов автоматическая векторизация способна обеспечивать значительное повышение производительности. Правильные компиляторные флаги и стиль программирования способны улучшать ее еще больше.

ОПРЕДЕЛЕНИЯ *Автоматическая векторизация* – это векторизация исходного кода компилятором для стандартных языков C, C++ или Fortran.

[1] Важно отметить, что, хотя автоматическая векторизация нередко приводит к значительному повышению производительности, она иногда замедляет работу исходного кода. Это связано с тем, что накладные расходы на настройку векторных команд превышают прирост производительности. Компилятор обычно принимает решение о векторизации с использованием функции стоимости. Компилятор выполняет векторизацию, если функция стоимости показывает, что код будет быстрее, но он лишь угадывает длину массива и исходит из того, что все данные берутся с первого уровня кеша.

Пример: автоматическая векторизация

Давайте посмотрим на работу автоматической векторизации в простом цикле из потоковой триады[1] приложения сравнительного тестирования STREAM Benchmark, представленного в разделе 3.2.4. В приведенном ниже листинге мы выделяем код триады из приложения STREAM Benchmark в отдельную тестовую задачу.

Автоматическая векторизация потоковой триады stream_triad.c

autovec/stream_triad.c

```
 1 #include <stdio.h>
 2 #include <sys/time.h>
 3 #include "timer.h"
 4
 5 #define NTIMES 16
 6 #define STREAM_ARRAY_SIZE 80000000
 7 static double a[STREAM_ARRAY_SIZE],
                b[STREAM_ARRAY_SIZE],
                c[STREAM_ARRAY_SIZE];
 8
 9 int main(int argc, char *argv[]){
10    struct timeval tstart;
11    double scalar = 3.0, time_sum = 0.0;
12    for (int i=0; i<STREAM_ARRAY_SIZE; i++) {
13       a[i] = 1.0;
14       b[i] = 2.0;
15    }
16
17    for (int k=0; k<NTIMES; k++){
18       cpu_timer_start(&tstart);
19       for (int i=0; i<STREAM_ARRAY_SIZE; i++){
20          c[i] = a[i] + scalar*b[i];
21       }
22       time_sum += cpu_timer_stop(tstart);
23       c[1] = c[2];
24    }
25   printf("Среднее время выполнения составляет %lf msecs\n", time_sum/NTIMES);
26 }
```

Достаточно крупный, чтобы втиснуть его в основную память

Инициализирует данные и массивы

Цикл потоковой триады содержит три операнда с умножением и сложением

Не позволяет компилятору оптимизировать цикл

```
Makefile for the GCC compiler
CFLAGS=-g -O3 -fstrict-aliasing \
       -ftree-vectorize -march=native -mtune=native \
       -fopt-info-vec-optimized

stream_triad: stream_triad.o timer.o
```

[1] Для справки: потоковая триада (stream triad) – это вычислительное ядро с тремя операндами с умножением и сложением в форме a(i) = b(i) + q × c(i) в рамках приложения эталонного тестирования производительности STREAM Benchmark. – *Прим. перев.*

Мы обсудим компиляторные флаги подробнее в разделе 6.4, а файлы timer.c и timer.h в разделе 17.2. Компиляция файла stream_triad.c с версией 8 компилятора GCC возвращает следующий ниже ответ компилятора:

```
stream_triad.c:19:7: note: loop vectorized
stream_triad.c:12:4: note: loop vectorized
```

GCC векторизует цикл инициализации и цикл потоковой триады! Выполнить потоковую триаду можно с помощью

```
./stream_triad
```

Подтвердить использование векторных команд компилятором можно с помощью инструмента likwid (раздел 3.3.1).

```
likwid-perfctr -C 0 -f -g MEM_DP ./stream_triad
```

Взгляните на распечатку отчета этой команды с приведенными ниже строками:

```
| FP_ARITH_INST_RETIRED_128B_PACKED_DOUBLE |  PMC0  |          0 |
| FP_ARITH_INST_RETIRED_SCALAR_DOUBLE      |  PMC1  |         98 |
| FP_ARITH_INST_RETIRED_256B_PACKED_DOUBLE |  PMC2  |  640000000 |
| FP_ARITH_INST_RETIRED_512B_PACKED_DOUBLE |  PMC3  |          0 |
```

В распечатке вы видите, что большинство количеств операций относится к категории `256B_PACKED_DOUBLE` в третьей строке. Зачем все эти 256-битовые операции? Некоторые версии компилятора GCC, включая версию 8.2, используемую в этом тесте, генерируют для процессора Skylake 256-битовые векторные команды вместо 512-битовых. Без такого инструмента, как likwid, нам пришлось бы тщательно проверять отчеты о векторизации либо проверять сгенерированные ассемблерные инструкции, чтобы в конце обнаружить, что компилятор не сгенерировал надлежащие команды. Для компилятора GCC мы можем изменить сгенерированные команды, добавив компиляторный флаг `-mprefer-vector-width=512`, а затем повторить попытку. Теперь мы получим инструкции AVX512 с восемью значениями двойной точности, вычисляемыми одновременно:

```
| FP_ARITH_INST_RETIRED_256B_PACKED_DOUBLE |  PMC2  |          0 |
| FP_ARITH_INST_RETIRED_512B_PACKED_DOUBLE |  PMC3  |  320000000 |
```

Пример: автоматическая векторизация в функции

В этом примере мы пробуем чуть более сложную версию кода из предыдущего примера, в котором цикл потоковой триады находится в отдельной функции. В следующем ниже листинге показано, как это сделать.

Цикл потоковой триады в отдельной функции

autovec_function/stream_triad.c

```
 1 #include <stdio.h>
 2 #include <sys/time.h>
 3 #include "timer.h"
 4
 5 #define NTIMES 16
 6 #define STREAM_ARRAY_SIZE 80000000
 7 static double a[STREAM_ARRAY_SIZE], b[STREAM_ARRAY_SIZE],
                c[STREAM_ARRAY_SIZE];
 8
 9 void stream_triad(double* a, double* b,
         double* c, double scalar){
10    for (int i=0; i<STREAM_ARRAY_SIZE; i++){
11       a[i] = b[i] + scalar*c[i];
12    }
13 }
14
15 int main(int argc, char *argv[]){
16    struct timeval tstart;
17    double scalar = 3.0, time_sum = 0.0;
18    for (int i=0; i<STREAM_ARRAY_SIZE; i++) {
19       a[i] = 1.0;
20       b[i] = 2.0;
21    }
22
23    for (int k=0; k<NTIMES; k++){
24       cpu_timer_start(&tstart);
25       stream_triad(a, b, c, scalar);
26       time_sum += cpu_timer_stop(tstart);
27       // это чтобы компилятор не оптимизировал цикл
28       c[1] = c[2];
29    }
30   printf("Среднее время выполнения составляет %lf msecs\n", time_sum/NTIMES);
31 }
```

Цикл потоковой триады в отдельной функции

Вызов функции потоковой триады

Давайте посмотрим на распечатку из компилятора GCC для программного кода в приведенном выше листинге цикла потоковой триады:

```
stream_triad.c:10:4: note: loop vectorized
stream_triad.c:10:4: note: loop versioned for vectorization because of
   possible aliasing
stream_triad.c:10:4: note: loop vectorized
stream_triad.c:18:4: note: loop vectorized
```

Компилятор не может сказать точно, указывают ли аргументы функции на одни и те же данные либо на перекрывающиеся данные. Это приводит к тому, что компилятор создает более одной версии функции и производит код, который тестирует аргументы, чтобы определить,

какую из этих версий использовать. Мы можем это исправить, добавив атрибут `restrict` в определение функции как часть аргументов. Данное ключевое слово было добавлено в стандарт C99. К сожалению, ключевое слово `restrict` не стандартизировано в C++, но атрибут `__restrict` работает для GCC, Clang и Visual C++. Еще одной распространенной формой этого атрибута в компиляторах C++ является `__restrict__`:

```
9 void stream_triad(double* restrict a, double* restrict b,
                    double* restrict c, double scalar){
```

Мы применили GCC для компиляции исходного кода с добавлением ключевого слова `restrict` и получили:

```
stream_triad.c:10:4: note: loop vectorized
stream_triad.c:10:4: note: loop vectorized
stream_triad.c:18:4: note: loop vectorized
```

Теперь этот компилятор генерирует меньше версий функции. Мы также должны отметить, что флаг `-fstrict-aliasing` (флаг строгой псевдонимизации) указывает компилятору генерировать код агрессивно, исходя из допущения, что псевдонимизации нет.

> **ОПРЕДЕЛЕНИЕ** *Псевдонимизация*[1] – это ситуация, когда указатели указывают на перекрывающиеся области памяти. В такой ситуации компилятор не способен определить, является ли эта область той же самой памятью и будет ли небезопасно генерировать векторизованный код или другие оптимизации.

В последние годы опция строгой псевдонимизации стала использоваться по умолчанию в GCC и других компиляторах (флаг `-fstrict-aliasing` устанавливается оптимизационными уровнями `-O2` и `-O3`). Это нарушило работу массы программного кода, в котором псевдонимизированные переменные на самом деле существовали. Как следствие, компиляторы снизили градус агрессивности, с которой они генерируют более эффективный код. И все это для того, чтобы сообщить, что вы можете получать разные результаты с разными компиляторами и даже с разными версиями одного компилятора.

Используя атрибут `restrict`, вы даете компилятору обещание, что псевдонимизации не будет. Мы рекомендуем использовать как атрибут `restrict`, так и компиляторный флаг `-fstrict-aliasing`. Атрибут переносится вместе с исходным кодом во все архитектуры и компиляторы. Вам придется применять компиляторные флаги для каждого компилятора, но они влияют на весь ваш исходный код.

Из этих примеров может показаться, что для программиста самый лучший способ получить векторизацию – просто позволить компиля-

---

[1] Псевдонимизация (aliasing) – это ситуация, когда у имени адреса в памяти появляется несколько псевдонимов, возникающая в результате наложения нескольких указателей на один адрес. – *Прим. перев.*

тору выполнять векторизацию автоматически. Хотя компиляторы и совершенствуются, им нередко не удается распознавать возможность безопасной векторизации цикла в более сложных участках кода. И поэтому программисту приходится помогать компилятору подсказками. Мы обсудим этот технический прием далее.

6.3.3 Обучение компилятора посредством подсказок: прагмы и директивы

Итак, компилятор не совсем в состоянии понимать циклы и генерировать векторизованный код; есть ли что-то, что мы можем тут сделать? В данном разделе мы расскажем о том, как давать более точные указания компилятору. В свою очередь это даст вам больше контроля над процессом векторизации исходного кода. Здесь вы узнаете о том, как использовать прагмы и директивы для передачи информации компилятору в целях обеспечения переносимой имплементации векторизации.

ОПРЕДЕЛЕНИЕ *Прагма* – это команда компилятору C или C++, помогающая ему интерпретировать исходный код. Форма команды представляет собой команду препроцессора, начинающуюся с `#pragma`. (На Fortran, где эта команда называется директивой, ее форма представляет собой комментарную строку, начинающуюся с `!$`.)

Пример: использование ручных подсказок компилятору в отношении векторизации

Нам нужен пример, в котором компилятор не векторизует код без посторонней помощи. Для этого мы воспользуемся примером исходного кода в следующих ниже листингах. Вместе они вычисляют скорость волны в ячейке, чтобы определить временной интервал для использования в расчете. Временной шаг может быть не больше минимального времени, необходимого для того, чтобы волна пересекала любую ячейку сетки.

Расчет временного шага с использованием цикла редукции к минимуму в timestep/main.c

`timestep/main.c`

```
 1 #include <stdio.h>
 2 #include "timestep.h"
 3 #define NCELLS 10000000
 4 static double H[NCELLS], U[NCELLS], V[NCELLS], dx[NCELLS],
            dy[NCELLS];
 5 static int celltype[NCELLS];
 6
 7 int main(int argc, char *argv[]){
 8    double mymindt;
```

```
 9    double g = 9.80, sigma = 0.95;
10    for (int ic=0; ic<NCELLS ; ic++) {
11       H[ic] = 10.0;
12       U[ic] = 0.0;
13       V[ic] = 0.0;
14       dx[ic] = 0.5;
15       dy[ic] = 0.5;
16       celltype[ic] = REAL_CELL;
17    }
18    H[NCELLS/2] = 20.0;
19
20    mymindt = timestep(NCELLS, g, sigma,
                         celltype, H, U, V, dx, dy);
21
22    printf("Минимальный dt составляет %lf\n",mymindt);
23 }
```

Инициализирует массивы и данные

Вызывает вычисление временного шага

Расчет временного шага с использованием цикла редукции к минимуму в timestep/timestep.c

timestep/timestep.c

```
 1 #include <math.h>
 2 #include "timestep.h"
 3 #define REAL_CELL 1
 4
 5 double timestep(int ncells, double g, double sigma, int* celltype,
 6           double* H, double* U, double* V, double* dx, double* dy){
 7    double wavespeed, xspeed, yspeed, dt;
 8    double mymindt = 1.0e20;
 9    for (int ic=0; ic<ncells ; ic++) {
10       if (celltype[ic] == REAL_CELL) {
11          wavespeed = sqrt(g*H[ic]);
12          xspeed = (fabs(U[ic])+wavespeed)/dx[ic];
13          yspeed = (fabs(V[ic])+wavespeed)/dy[ic];
14          dt=sigma/(xspeed+yspeed);
15          if (dt < mymindt) mymindt = dt;
16       }
17    }
18    return(mymindt);
19 }
```

Скорость волны

Время прохождения волны через ячейку, умноженное на коэффициент запаса устойчивости (сигму)

Получает минимальное время для всех ячеек и использует для временного шага

```
Makefile for GCC
CFLAGS=-g -O3 -fstrict-aliasing -ftree-vectorize -fopenmp-simd \
  -march=native -mtune=native -mprefer-vector-width=512 \
  -fopt-info-vec-optimized -fopt-info-vec-missed

stream_triad: main.o timestep.o timer.o
```

В этом примере мы добавили компиляторный флаг `-fopt-info-vec-missed`, чтобы получить отчет о неуспешных векторизациях циклов. В результате компилирования этого исходного кода мы получаем:

```
main.c:10:4: note: loop vectorized

timestep.c:9:4: missed: couldn't vectorize loop
timestep.c:9:4: missed: not vectorized: control flow in loop.
```

Указанный отчет о векторизации сообщает о том, что цикл временного шага не был векторизован из-за условного блока в цикле. Давайте посмотрим, сможем ли мы оптимизировать данный цикл, добавив прагму. Добавьте следующую ниже строку непосредственно перед циклом for в timestep.c (в строке 9):

```
#pragma omp simd reduction(min:mymindt)
```

Теперь компилирование кода показывает противоречивые сообщения о векторизировании цикла временного шага:

```
main.c:10:4: note: loop vectorized
timestep_opt.c:9:9: note: loop vectorized
timestep_opt.c:11:7: note: not vectorized: control flow in loop.
```

Нам нужно проверить исполняемый файл с помощью инструмента производительности наподобие likwid, чтобы увидеть, что компилятор его действительно векторизует:

```
likwid-perfctr -g MEM_DP -C 0 ./timestep_opt
```

Распечатка из инструмента likwid показывает, что ни одной векторной команды не исполнено:

```
|               DP MFLOP/s             |  451.4928 |
|             AVX DP MFLOP/s           |         0 |
|             Packed MUOPS/s           |         0 |
```

С версией компилятора GCC 9.0 мы смогли сделать его векторизованным, добавив флаг `-fno-trapping-math`. Если в условном блоке есть деление, то этот флаг говорит компилятору не беспокоиться о выбросе исключения с ошибкой, поэтому он будет векторизован. Если в условном блоке есть `sqrt`, то флаг `-fno-math-errno` позволит компилятору выполнить векторизацию. В целях более оптимальной переносимости прагма также должна сообщать компилятору о том, что во время итераций цикла некоторые переменные не сохраняются и, следовательно, не зависят, или «антизависят», от порядка исполнения. Эти зависимости будут рассмотрены после листинга в следующем ниже примере.

```
#pragma omp simd private(wavespeed, xspeed, yspeed, dt) reduction(min:mymindt)
```

Еще более оптимальный способ указать, что область видимости переменных лимитирована каждой итерацией цикла, состоит в том, чтобы объявлять переменные в области видимости цикла:

```
double wavespeed = sqrt(g*H[ic]);
double xspeed = (fabs(U[ic])+wavespeed)/dx[ic];
```

```
double yspeed = (fabs(V[ic])+wavespeed)/dy[ic];
double dt=sigma/(xspeed+yspeed);
```

Теперь мы можем удалить выражение `private` и объявление переменных перед циклом. Мы также можем добавить атрибут `restrict` в интерфейс функции, чтобы сообщить компилятору, что указатели не перекрываются:

```
double timestep(int ncells, double g, double sigma, int* restrict celltype,
                double* restrict H, double* restrict U, double* restrict V,
                double* restrict dx, double* restrict dy);
```

Даже имея все эти изменения, мы не смогли заставить компилятор GCC векторизовать этот код. При дальнейшем расследовании с использованием версии 9 компилятора GCC мы, наконец, добились успеха, добавив флаг `-fno-trapping-math`. Если в условном блоке есть деление, то этот флаг говорит компилятору о том, чтобы он не беспокоился о выбросе исключения с ошибкой, и поэтому он будет векторизован. Если в условном блоке есть `sqrt`, то флаг `-fno-math-errno` позволяет компилятору выполнить векторизацию. Между тем компилятор Intel векторизует все версии.

Одной из наиболее распространенных операций является сумма массива. Еще в разделе 4.5 мы встречали этот тип операции в качестве редукции. Мы добавим в эту операцию немного сложности, включив условный блок, который лимитирует сумму реальными ячейками в сетке. Здесь *реальными ячейками* считаются элементы, не находящиеся на границе, или призрачные ячейки из других процессоров. Мы обсуждаем призрачные ячейки в главе 8.

Листинг 6.1 Расчет суммы массы с использованием цикла редукции к сумме

mass_sum/mass_sum.c

```
 1 #include "mass_sum.h"
 2 #define REAL_CELL 1
 3
 4 double mass_sum(int ncells, int* restrict celltype, double* restrict H,
 5                 double* restrict dx, double* restrict dy){
 6    double summer = 0.0;
 7 #pragma omp simd reduction(+:summer)
 8    for (int ic=0; ic<ncells ; ic++) {
 9       if (celltype[ic] == REAL_CELL) {
10          summer += H[ic]*dx[ic]*dy[ic];
11       }
12    }
13    return(summer);
14 }
```

Устанавливает переменную редукции равной нулю

Цикл потока трактует summer как переменную редукции

Условный блок может быть имплементирован с помощью маски

Прагма OpenMP SIMD должна автоматически установить переменную редукции равной нулю, но когда прагма игнорируется, то необходима инициализация в строке 6. Прагма OpenMP SIMD в строке 7 сообщает

компилятору о том, что мы используем переменную `summer` в редукционной сумме. Условное выражение в строке 9 цикла может быть имплементировано в векторных операциях с маской. Каждая векторная полоса имеет свою собственную копию `summer`, и потом они будут объединены в конце цикла `for`.

Компилятор Intel успешно распознает редукцию к сумме и автоматически векторизует цикл без прагмы SIMD. GCC также векторизуется с версиями 9 и более поздними версиями компилятора.

Пример: использование отчета компилятора о векторизации в качестве руководства для добавления прагм

Поскольку компилятор Intel генерирует более детальные отчеты о векторизации, в этом примере мы будем использовать его. Исходный код этого примера взят из листинга 4.14. Главный цикл показан в приведенном ниже листинге с номерами строк.

Главный цикл стенсильного примера из листинга 4.14

```
56    for (int j = 1; j < jmax-1; j++){
57       for (int i = 1; i < imax-1; i++){
58          xnew[j][i] = (x[j][i] + x[j ][i-1] + x[j ][i+1]
                                  + x[j-1][i ] + x[j+1][i ])/5.0;
59       }
60    }
```

В этом примере мы использовали компилятор Intel v19 с вот такими компиляторными флагами:

```
CFLAGS=-g -O3 -std=c99 -qopenmp-simd -ansi-alias -xHost \
   -qopt-zmm-usage=high -qopt-report=5 -qopt-report-phase=vec,loop
```

Отчет о векторизации показывает, что компилятор не векторизовал внутренний цикл в строке 57 и внешний цикл в строке 56:

```
LOOP BEGIN at stencil.c(56,7)
   remark #15344: loop was not vectorized: vector dependence prevents
                  vectorization
   remark #15346: vector dependence: assumed OUTPUT dependence between
                  xnew[j][i] (58:13)and xnew[j][i] (58:13)
   remark #15346: vector dependence: assumed OUTPUT dependence between
                  xnew[j][i] (58:13)and xnew[j][i] (58:13)
   LOOP BEGIN at stencil.c(57,10)
      remark #15344: loop was not vectorized: vector dependence prevents
                     vectorization
      remark #15346: vector dependence: assumed FLOW dependence between
                     xnew[j][i] (58:13)and x[j][i] (58:13)
      remark #15346: vector dependence: assumed ANTI dependence between
                     x[j][i] (58:13)and xnew[j][i] (58:13)
      remark #25438: unrolled without remainder by 4
   LOOP END
LOOP END
```

В предыдущем примере зависимость[1] от прямого и обратного порядка исполнения возникает из-за возможности псевдонимизации между x и xnew. Компилятор в этом случае более консервативен, чем это необходимо. Зависимость от выхода вызывается только при попытке векторизирования внешнего цикла. Компилятор не может быть уверен в том, что последующие итерации внутреннего цикла не будут писаться в то же место, что и предыдущая итерация. Прежде чем мы продолжим, давайте дадим определения нескольким терминам.

- *Зависимость от прямого порядка исполнения* – это ситуация, когда переменная внутри цикла читается после записи, и она называется чтением после записи (read-after-write, RAW).
- *Зависимость от обратного порядка исполнения* – это ситуация, когда переменная внутри цикла пишется после чтения, и она называется записью после чтения (write-after-read, WAR).
- *Зависимость от выхода* – переменная пишется более одного раза в цикле.

В случае компилятора GCC v8.2 отчет о векторизации таков:

```
stencil.c:57:10: note: loop vectorized
stencil.c:57:10: note: loop versioned for vectorization because of
                  possible aliasing
stencil.c:51:7: note: loop vectorized
stencil.c:37:7: note: loop vectorized
stencil.c:37:7: note: loop versioned for vectorization because of
                 possible aliasing
```

Компилятор GCC решает создать две версии и тестирует, которую из них использовать во время выполнения. Отчет настолько хорош, что дает нам четкое представление о причине проблемы. Мы может решать такие проблемы двумя путями. Мы можем помочь компилятору, добавив прагму перед циклом в строке 57 следующим образом:

```
#pragma omp simd
         for (int i = 1; i < imax-1; i++){
```

Еще один подход к решению этой проблемы состоит в добавлении атрибута restrict в определение x и xnew:

```
double** restrict x = malloc2D(jmax, imax);
double** restrict xnew = malloc2D(jmax, imax);
```

Отчет о векторизации для Intel теперь показывает, что внутренний цикл векторизуется с помощью векторизованного обдирочного цикла,

---

[1] Прямая зависимость возникает из двух команд, которые обращаются к одному и тому же ресурсу или его модифицируют. Команда S2 зависит от прямого порядка исполнения S1 тогда и только тогда, когда S1 модифицирует ресурс, который читается S2, и S1 предшествует S2 при исполнении. Команда S2 обратно зависит от S1 тогда и только тогда, когда S2 модифицирует ресурс, который читается S1, и S1 предшествует S2 при исполнении. См. https://en.wikipedia.org/wiki/Dependence_analysis. – *Прим. перев.*

главного векторизованного цикла и векторизованного остаточного цикла. Здесь требуется еще несколько определений.

- *Отслаивающий цикл* (peel loop) – цикл, исполняемый для невыровненных данных, чтобы затем главный цикл выровнял эти данные. Нередко отслаивающий цикл исполняется условно во время выполнения, если обнаруживается, что данные не выровнены.
- *Остаточный цикл* (remainder loop) – цикл, который исполняется после главного цикла для обработки частичного набора данных, который слишком мал для полной векторной длины.

Отслаивающий цикл добавляется для обработки невыровненных данных в начале цикла, а остаточный цикл обрабатывает любые дополнительные данные в конце цикла. Отчеты для всех трех циклов выглядят одинаково. Взглянув на отчет о главном цикле, мы видим, что оценочное ускорение более чем в шесть раз быстрее:

```
LOOP BEGIN at stencil.c(55,21)
   remark #15388: vec support: reference xnew[j][i] has aligned access
   [ stencil.c(56,13) ]
   remark #15389: vec support: reference x[j][i] has unaligned access
   [ stencil.c(56,28) ]
   remark #15389: vec support: reference x[j][i-1] has unaligned access
   [ stencil.c(56,38) ]
   remark #15389: vec support: reference x[j][i+1] has unaligned access
   [ stencil.c(56,50) ]
   remark #15389: vec support: reference x[j-1][i] has unaligned access
   [ stencil.c(56,62) ]
   remark #15389: vec support: reference x[j+1][i] has unaligned access
   [ stencil.c(56,74) ]
   remark #15381: vec support: unaligned access used inside loop body
   remark #15305: vec support: vector length 8
   remark #15399: vec support: unroll factor set to 2
   remark #15309: vec support: normalized vectorization overhead 0.236
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
   remark #15449: unmasked aligned unit stride stores: 1
   remark #15450: unmasked unaligned unit stride loads: 5
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 43
   remark #15477: vector cost: 6.620
   remark #15478: estimated potential speedup: 6.370
   remark #15486: divides: 1
   remark #15488: --- end vector cost summary ---
   remark #25015: Estimate of max trip count of loop=125
LOOP END
```

Обратите внимание, что оценочное ускорение осмотрительно помечено как *потенциальное ускорение*. Маловероятно, что вы получите полное оценочное ускорение, если только:

- ваши данные не находятся на высоком уровне кеша;
- фактическая длина массива не равна long;

- вы не лимитированы по пропускной способности в загрузках данных из основной памяти.

В предыдущей имплементации фактическое измеренное ускорение на процессоре Skylake Gold с компилятором Intel было быстрее в 1.39 раза, чем в невекторизованной версии. Указанный отчет о векторизации предназначен для ускорения работы процессора, но у нас по-прежнему остается вычислительное ядро из основной памяти с лимитом по пропускной способности памяти, с которым нужно как-то иметь дело.

Для компилятора GCC прагма SIMD успешно устраняет версионирование двух циклов. На самом деле добавление выражения `restrict` не возымело никакого эффекта, и оба цикла по-прежнему версионируются. Кроме того, поскольку во всех этих случаях используется векторизованная версия, производительность не меняется. В целях понимания ускорения мы можем сравнить производительность с версией, в которой векторизация отключена, и обнаружить, что ускорение при векторизации с помощью GCC быстрее примерно в 1.22 раза.

6.3.4 Дрянные циклы, они у нас в руках: используйте внутренние векторные функции компилятора

Для проблемных циклов, которые никак не векторизуются даже с помощью подсказок, внутренние векторные функции компилятора предлагают еще один вариант. В этом разделе мы рассмотрим подходы к использованию внутренних функций (intrinsics) для большего контроля над векторизацией. Недостатком внутренних векторных функций является то, что они менее переносимы. Здесь мы рассмотрим несколько примеров, в которых внутренние векторные функции используются для успешной векторизации, чтобы показать использование внутренних функций для векторизирования суммы по Кахану, продемонстрированной в разделе 5.7. В том разделе мы сказали, что стоимость суммы по Кахану в условиях обычной операции суммирования составляла около четырех операций с плавающей точкой вместо одной. Но если мы сможем векторизовать операцию суммирования по Кахану, то его стоимость станет намного меньше.

В имплементациях в этих примерах используется 256-битная внутренняя векторная функция, служащая для ускорения операции почти в четыре раза по сравнению с последовательной версией. Мы покажем три разных подхода к имплементированию вычислительного ядра суммирования по Кахану в листингах следующих ниже примеров. Полная имплементация находится по адресу https://github.com/lanl/GlobalSums.git, которая извлечена из примера глобальных сумм. Она включена в каталог GlobalSumsVectorized в исходном коде этой главы.

Пример: имплементация суммирования по Кахану с использованием внутренних векторных функций

В первом примере внутренних функций компилятора, разработанном Энди Дюбуа и Бреттом Нойманом из Лос-Аламосской национальной лаборатории, используются внутренние векторные функции Intel x86. Это наиболее часто используемый набор внутренних функций, который может выполняться и на процессорах Intel, и на процессорах AMD, поддерживающих векторные инструкции AVX.

Версия с внутренними векторными функциями для Intel x86 с суммированием по Кахану с улучшенной прецизионностью

GlobalSumsVectorized/kahan_intel_vector.c

```
 1 #include <x86intrin.h>
 2
 3 static double
       sum[4] __attribute__ ((aligned (64)));
 4
 5 double do_kahan_sum_intel_v(double* restrict var, long ncells)
 6 {
 7    double const zero = 0.0;
 8    __m256d local_sum = _mm256_broadcast_sd(
                          (double const*) &zero);
 9    __m256d local_corr = _mm256_broadcast_sd(
                           (double const*) &zero);
10
11    #pragma simd
12    #pragma vector aligned
13    for (long i = 0; i < ncells; i+=4) {
14       __m256d var_v = _mm256_load_pd(&var[i]);
15       __m256d corrected_next_term =
                    var_v + local_corr;
16       __m256d new_sum =
                    local_sum + local_corr;
17       local_corr = corrected_next_term -
                    (new_sum - local_sum);
18       local_sum = new_sum;
19    }
20    __m256d sum_v;
21    sum_v = local_corr;
22    sum_v += local_sum;
23    _mm256_store_pd(sum, sum_v);
24
25    struct esum_type{
26       double sum;
27       double correction;
28    } local;
```

Включить файл с определениями векторных внутренних функций для Intel и AMD x86

Определяет регулярно выровненный массив двойной точности шириной 4

Заполняет нулями переменную с вектором шириной 4 значения двойной точности

Прагмы для указания компилятору работать с выровненными векторными переменными

Загружает четыре значения из стандартного массива в векторную переменную

Выполняет стандартную операцию суммирования по Кахану для всех переменных с векторами шириной 4

Сохраняет четыре векторных полосы в регулярно выровненный массив из четырех значений

```
29    local.sum = 0.0;
30    local.correction = 0.0;
31
32    for (long i = 0; i < 4; i++) {
33       double corrected_next_term_s =
                 sum[i] + local.correction;
34       double new_sum_s =
                 local.sum + local.correction;
35       local.correction =
                 corrected_next_term_s -
                 (new_sum_s - local.sum);
36       local.sum          = new_sum_s;
37    }
38    double final_sum =
         local.sum + local.correction;
39    return(final_sum);
40 }
```

Суммирует четыре суммы из четырех векторных полос с использованием скалярных переменных

Пример: имплементация суммирования по Кахану с использованием внутренних векторных функций GCC

Во второй имплементации суммирования по Кахану используются векторные расширения GCC. Эти векторные команды способны поддерживать и другие архитектуры, кроме векторных модулей AVX в архитектурах x86. Но переносимость векторных расширений GCC лимитирована местами, где можно использовать компилятор GCC. Если задана более длинная векторная длина, чем поддерживается оборудованием, то компилятор генерирует команды, используя комбинации более коротких векторных длин.

Версия с векторными расширениями GCC суммирования по Кахану с повышенной прецизионностью

GlobalSumsVectorized/kahan_gcc_vector.c

```
 1 static double
    sum[4] __attribute__ ((aligned (64)));
 2
 3 double do_kahan_sum_gcc_v(double* restrict var, long ncells)
 4 {
 5    typedef double vec4d __attribute__
         ((vector_size(4 * sizeof(double))));
 6
 7    vec4d local_sum = {0.0};
 8    vec4d local_corr = {0.0};
 9
10    for (long i = 0; i < ncells; i+=4) {
11       vec4d var_v = *(vec4d *)&var[i];
```

Определяет регулярно выровненный массив из четырех значений двойной точности

Объявляет векторный тип данных vec4d

Заполняет нулями векторную переменную двойной точности шириной 4

Загружает четыре значения из стандартного массива в векторную переменную

```
12       vec4d corrected_next_term =
            var_v + local_corr;
13       vec4d new_sum =
            local_sum + local_corr;
14       local_corr = corrected_next_term -
            (new_sum - local_sum);
15       local_sum = new_sum;
16    }
17    vec4d sum_v;
18    sum_v = local_correction;
19    sum_v += local_sum;
20    *(vec4d *)sum = sum_v;
21
22    struct esum_type{
23       double sum;
24       double correction;
25    } local;
26    local.sum = 0.0;
27    local.correction = 0.0;
28
29    for (long i = 0; i < 4; i++) {
30       double corrected_next_term_s =
            sum[i] + local.correction;
31       double new_sum_s =
            local.sum + local.correction;
32       local.correction =
            corrected_next_term_s
            (new_sum_s - local.sum);
33       local.sum = new_sum_s;
34    }
35    double final_sum =
         local.sum + local.correction;
36    return(final_sum);
37 }
```

Стандартная операция суммирования по Кахану выполняется на всех векторных переменных шириной 4

Сохраняет четыре векторные полосы в регулярно выровненный массив из четырех значений

Суммирует четыре суммы из четырех векторных полос с использованием скалярных переменных

Пример: имплементация суммирования по Каханау с использованием внутренних векторных функций C++

Для кода на C++ Агнер Фог из Технического университета Дании написал библиотеку векторных классов C++ в рамках лицензии с открытым исходным кодом. Эта библиотека классов переносима для всех архитектур оборудования и автоматически адаптируется под более старое оборудование с более короткой векторной длиной. Библиотека векторных классов Фога хорошо проработана и содержит обширное руководство. Более подробная информация об этой библиотеке приведена в разделе дополнительного чтения в конце главы.

В этом примере мы напишем векторную версию суммирования по Кахану на C++, а затем вызовем ее из нашей главной программы на C. Мы также обработаем оставшийся блок значений, который не соответствует полной ширине

вектора, с помощью функции partial_load. У нас не всегда будут массивы, которые равномерно поделены на группы шириной 4. Иногда мы заполняем массивы дополнительными нулями, чтобы придавать массиву нужный размер, но более качественный подход заключается в обработке оставшихся значений в отдельном блоке кода в конце цикла. Обратите внимание, что тип данных Vec4d определен в заголовочном файле векторного класса.

Библиотека векторных классов C++ Агнера Фога для повышенной прецизионности для суммирования по Кахану

GlobalSumsVectorized/kahan_fog_vector.cpp

```
 1 #include "vectorclass.h"
 2
 3 static double
      sum[4] __attribute__ ((aligned (64)));
 4
 5 extern "C" {
 6 double do_kahan_sum_agner_v(double* var,
                               long ncells);
 7 }
 8
 9 double do_kahan_sum_agner_v(double* var, long ncells)
10 {
11    Vec4d local_sum(0.0);
12    Vec4d local_corr(0.0);
13    Vec4d var_v;
14
15    int ncells_main=(ncells/4)*4;
16    int ncells_remainder=ncells%4;
17    for (long i = 0; i < ncells_main; i+=4) {
18       var_v.load(var+i);
19       Vec4d corrected_next_term =
            var_v + local_corr;
20       Vec4d new_sum =
            local_sum + local_corr;
21       local_corr = corrected_next_term -
            (new_sum - local_sum);
22       local_sum = new_sum;
23    }
24    if (ncells_remainder > 0) {
25       var_v.load_partial(ncells_remainder,var+ncells_main);
26       Vec4d corrected_next_term = var_v + local_corr;
27       Vec4d new_sum = local_sum + local_corr;
28       local_corr = corrected_next_term - (new_sum - local_sum);
29       local_sum = new_sum;
30    }
31
32    Vec4d sum_v;
33    sum_v = local_corr;
34    sum_v += local_sum;
```

Вставляет заголовочный файл векторного класса

Определяет регулярно выровненный массив из четырех значений двойной точности

Указывает, что нам нужна подпрограмма в стиле C

Заполняет нулями векторную переменную двойной точности шириной 4

Определяет векторную переменную двойной точности 256 бит (или четыре значения двойной точности)

Загружает четыре значения из стандартного массива в векторную переменную

Стандартная операция суммирования по Кахану выполняется для всех векторных переменных шириной 4

```
35    sum_v.store(sum);
36
37    struct esum_type{
38       double sum;
39       double correction;
40    } local;
41    local.sum = 0.0;
42    local.correction = 0.0;
43    for (long i = 0; i < 4; i++) {
44       double corrected_next_term_s =
            sum[i] + local.correction;
45       double new_sum_s =
            local.sum + local.correction;
46       local.correction =
            corrected_next_term_s -
            (new_sum_s - local.sum);
47       local.sum = new_sum_s;
48    }
49    double final_sum =
         local.sum + local.correction;
50    return(final_sum);
51 }
```

Сохраняет четыре векторные полосы в регулярно выровненный массив из четырех значений

Суммирует четыре суммы из четырех векторных полос с использованием скалярных переменных

Команды загрузки (`load`) и сохранения (`store`) в этой библиотеке векторных классов более естественны, чем синтаксис некоторых других внутренних векторных функций

Затем мы протестировали сумму по Кахану, имплементированную с помощью трех внутренних векторных функций, в сравнении с изначальным последовательным суммированием и изначальным суммированием по Кахану. Мы использовали версию 8.2 компилятора GCC и провели тесты на процессоре Skylake Gold. Компилятору GCC не удается векторизовать исходный код последовательного суммирования и изначального суммирования по Кахану. Добавление прагмы OpenMP позволяет векторизовать последовательную сумму, но что касается суммирования по Кахану, то несомая его циклом зависимость не дает компилятору векторизовать его код.

В следующих ниже результатах производительности важно отметить, что векторизованные версии для последовательного суммирования и суммирования по Кахану со всеми тремя внутренними векторными функциями (выделены жирным шрифтом) имеют почти одинаковое время выполнения. Мы можем выполнять больше операций с плавающей точкой за одно и то же время и одновременно редуцировать числовую ошибку. Это отличный пример того, что при некоторых усилиях операции с плавающей точкой оказываются бесплатными.

```
SETTINGS INFO -- ncells 1073741824 log 30
Initializing mesh with Leblanc problem, high values first
  relative diff   runtime    Description
```

```
  8.423e-09     1.273343   Serial sum
          0     3.519778   Kahan sum with double double accumulator
4 wide vectors serial sum
  -3.356e-09    0.683407   Intel vector intrinsics Serial sum
  -3.356e-09    0.682952   GCC vector intrinsics Serial sum
  -3.356e-09    0.682756   Fog C++ vector class Serial sum
4 wide vectors Kahan sum
            0   1.030471   Intel Vector intrinsics Kahan sum
            0   1.031490   GCC vector extensions Kahan sum
            0   1.032354   Fog C++ vector class Kahan sum
8 wide vector serial sum
  -1.986e-09    0.663277   Serial sum (OpenMP SIMD pragma)
  -1.986e-09    0.664413   8 wide Intel vector intrinsic Serial sum
  -1.986e-09    0.664067   8 wide GCC vector intrinsic Serial sum
  -1.986e-09    0.663911   8 wide Fog C++ vector class Serial sum
8 wide vector Kahan sum
  -1.388e-16    0.689495   8 wide Intel Vector intrinsics Kahan sum
  -1.388e-16    0.689100   8 wide GCC vector extensions Kahan sum
  -1.388e-16    0.689472   8 wide Fog C++ vector class Kahan sum
```

6.3.5 Не для слабонервных: применение ассебмлерного кода для векторизации

В этом разделе мы рассмотрим ситуацию, когда в вашем приложении целесообразно писать векторный ассемблерный код. Мы также обсудим вопрос о том, как выглядит векторный ассемблерный код, как дизассемблировать скомпилированный код и как определять, какой именно набор векторных команд компилятор сгенерировал.

Программирование векторных модулей непосредственно с помощью векторных ассемблерных инструкций дает прекрасную возможность достигать максимальной производительности. Но для этого требуется глубокое понимание поведения в отношении производительности большого числа векторных инструкций во многих разных процессорах. Программисты, не обладающие таким опытом, вероятно, получат более высокую производительность от использования внутренних векторных функций, как показано в предыдущем разделе, чем от непосредственного написания векторных ассемблерных инструкций. Кроме того, переносимость векторного ассемблерного кода ограничена; он будет работать только на небольшом наборе процессорных архитектур. По этим причинам редко имеет смысл писать векторные ассемблерные инструкции.

Пример: обзор векторных ассемблерных инструкций

В целях ознакомления с тем, как выглядят векторные ассемблерные инструкции, мы можем вывести указанные инструкции на экран из объектного файла командой objdump. В листинге показана распечатка следующей ниже команды:

```
objdump -d -M=intel --no-show-raw-insn <object code file.o>
```

Ассемблерный код для версии с векторными расширениями GCC суммирования по Кахану

```
0000000000000000 <do_kahan_sum_gcc_v>:
 0:   test %rsi,%rsi
 3:   jle 90 <do_kahan_sum_gcc_v+0x90>
 9:   vxorpd %xmm2,%xmm2,%xmm2
 d:   xor %eax,%eax
 f:   vmovapd %ymm2,%ymm0
13:   nopl 0x0(%rax,%rax,1)
18:   vmovapd %ymm0,%ymm1
1c:   vaddpd %ymm2,%ymm0,%ymm0
20:   vaddpd (%rdi,%rax,8),%ymm2,%ymm3
25:   add $0x4,%rax
29:   vsubpd %ymm1,%ymm0,%ymm1
2d:   vsubpd %ymm1,%ymm3,%ymm2
31:   cmp %rax,%rsi
34:   jg 18 <do_kahan_sum_gcc_v+0x18>
36:   vaddpd %ymm2,%ymm0,%ymm0
3a:   vmovapd %ymm0,0x0(%rip)         # 42 <do_kahan_sum_gcc_v+0x42>
42:   vxorpd %xmm2,%xmm2,%xmm2
46:   vaddsd 0x0(%rip),%xmm2,%xmm0    # 4e <do_kahan_sum_gcc_v+0x4e>
4e:   vaddsd %xmm2,%xmm0,%xmm2
52:   vaddsd 0x0(%rip),%xmm0,%xmm0    # 5a <do_kahan_sum_gcc_v+0x5a>
5a:   vsubsd %xmm2,%xmm0,%xmm0
5e:   vaddsd %xmm2,%xmm0,%xmm3
62:   vaddsd 0x0(%rip),%xmm0,%xmm1    # 6a <do_kahan_sum_gcc_v+0x6a>
6a:   vsubsd %xmm2,%xmm3,%xmm2
6e:   vsubsd %xmm2,%xmm1,%xmm1
72:   vaddsd %xmm1,%xmm3,%xmm2
76:   vaddsd 0x0(%rip),%xmm1,%xmm0    # 7e <do_kahan_sum_gcc_v+0x7e>
7e:   vsubsd %xmm3,%xmm2,%xmm3
82:   vsubsd %xmm3,%xmm0,%xmm0
86:   vaddsd %xmm2,%xmm0,%xmm0
8a:   vzeroupper
8d:   retq
8e:   xchg %ax,%ax
90:   vxorpd %xmm0,%xmm0,%xmm0
94:   jmp 3a <do_kahan_sum_gcc_v+0x3a>
```

Если вы видите регистры `ymm`, то векторные инструкции были сгенерированы. Регистры `zmm` указывают на то, что имеются векторные инструкции AVX512. Регистры `xmm` генерируются как для скалярных, так и для векторных инструкций SSE. Мы можем сказать, что список был взят из файла kahan_gcc_vector.c.o, потому что в распечатке нет инструкций `zmm`. Если мы посмотрим на файл kahan_gcc_vector8.c.o, генерирующий 512-битные инструкции, то вы увидите инструкции `zmm`.

Поскольку редко имеет смысл делать что-то большее, чем просто просматривать генерируемые компилятором ассемблерные инструкции,

мы не будем рассматривать пример написания подпрограммы на ассемблере с азов.

6.4 Стиль программирования для более качественной векторизации

Мы предлагаем принять на вооружение стиль программирования, который более совместим с потребностями векторизации и другими формами параллелизации циклов. Основываясь на уроках, извлеченных из примеров, приведенных в этой главе, некоторые стили программирования помогают компилятору генерировать векторизованный код. Принятие следующих ниже стилей программирования приводит к повышению производительности прямо «из коробки» и уменьшению объема работы, необходимой для усилий по оптимизации.

Общие предложения:

- используйте атрибут `restrict` для указателей в аргументах и объявлениях функций (C и C++);
- используйте прагмы или директивы, где это необходимо, для информирования компилятора;
- будьте осторожны с оптимизацией под компилятор с помощью `#pragma unroll` и других технических приемов; вы можете лимитировать возможные варианты компиляторных преобразований[1];
- помещайте исключения и проверки ошибок с инструкциями печати в отдельный цикл.

Относительно структур данных:

- старайтесь использовать структуру данных с длиной long для самого внутреннего цикла;
- используйте наименьший необходимый тип данных (`short` вместо `int`);
- используйте обращения к сплошной памяти. Некоторые более новые наборы команд имплементируют загрузку памяти для сбора/разброса, но они менее эффективны;
- используйте структуру из массивов (SoA) вместо массива структур (AoS);
- по возможности используйте структуры данных, выровненные по памяти.

Относительно структур циклов:

- используйте простые циклы без особых условий выхода;
- сделайте границы цикла локальной переменной, копируя глобальные значения, а затем их используя;

[1] Прагма #pragma unroll указывает компилятору вставлять указанное число инструкций для замены итераций цикла и сокращать число итераций цикла в инструкции управления циклом либо удалять ее полностью.

- по возможности используйте индекс цикла для адресов массива;
- выставляйте размер границ цикла, чтобы он был известен компилятору. Если цикл длится всего три итерации, то компилятор может развернуть цикл вместо того, чтобы генерировать векторную инструкцию шириной четыре полосы;
- избегайте синтаксиса массива в циклах, критически важных для производительности (Fortran).

В теле цикла:

- определяйте локальные переменные внутри цикла, чтобы было ясно, что они не переносятся на последующие итерации (C и C++);
- переменные и массивы внутри цикла должны быть доступны только для записи или только для чтения (только с левой стороны либо с правой стороны знака равенства, за исключением редукций);
- не используйте в цикле локальные переменные для других целей – создавайте новую переменную. Пространство памяти, которое вы тратите впустую, гораздо менее важно, чем путаница, которую это создает для компилятора;
- избегайте вызовов функций и вместо этого пользуйтесь встраиванием тел функций (вручную или с помощью компилятора);
- ограничивайте использование условных блоков внутри цикла и при необходимости используйте простые формы, которые можно маскировать.

Относительно настроек и флагов компилятора:

- используйте последнюю версию компилятора и предпочитайте компиляторы, которые лучше выполняют векторизацию;
- используйте компиляторный флаг строгой псевдонимизации;
- создавайте код для самого мощного набора векторных команд, с которым вы можете работать.

6.5 Компиляторные флаги, относящиеся к векторизации, для различных компиляторов

В табл. 6.2 и 6.3 показаны компиляторные флаги, рекомендуемые при векторизации для последней версии различных компиляторов. Компиляторные флаги для векторизации часто меняются, поэтому ознакомьтесь с документацией для используемой версии компилятора.

Флаг строгой псевдонимизации, указанный во втором столбце табл. 6.2, должен помочь с автоматической векторизацией для C и C++, но убедитесь, что он не нарушает какой-либо код. В третьем столбце табл. 6.2 указаны различные опции, позволяющие указывать набор векторизационных команд, который следует использовать для некоторых компиляторов. Те из них, которые показаны в таблице, должны быть

хорошей отправной точкой. Отчеты о векторизации могут генерироваться с компиляторными флагами во втором столбце табл. 6.2. Компиляторные отчеты постоянно улучшаются в большинстве компиляторов и, скорее всего, изменятся. Для GCC рекомендуется использовать флаги opt и missed. Получать отчеты об оптимизации цикла одновременно с векторизацией бывает полезно тем, что вы видите, были ли циклы развернуты или взаимоизменены. Если вы не используете остальную часть OpenMP, а вместо нее используете директивы OpenMP SIMD, то следует использовать флаги в последнем столбце табл. 6.3.

Таблица 6.2 Флаги векторизации для различных компиляторов

Компилятор	Строгая псевдонимизация	Векторизация	Флаги с плавающей точкой

GCC, G++, GFortran v9	`-fstrict-aliasing`	`-ftree-vectorize`
		`-march=native`
		`-mtune=native`
		ver 8.0+:
		`-mprefer-vector`
		`-width=512`
			`-fno-trapping-math`
			`-fno-math-errno`

Clang v9	`-fstrict-aliasing`	`-fvectorize`
		`-march=native`
		`-mtune=native`
			`-fno-math-errno`

Intel icc v19	`-ansi-alias`	`-restrict`
		`-xHost`
		`-vecabi=cmdtarget`
		ver 18.0+:
		`-qopt-zmm-usage=high`	

MSVC	Не имплементирован	Включена по умолчанию	

IBM XLC v16	`-qalias=ansi`
	`-qalias=restrict`
		`-qsimd=auto`
		`-qhot`
		`-qarch=pwr9`
		`-qtune=pwr9`	

Cray	`-h restrict=[a,f]`	`-h vector3`
		`-h preferred_vector_width=#`
		где # может быть
		`[64,128,256,512]`	

Таблица 6.3 Флаги векторизационных отчетов OpenMP SIMD для различных компиляторов

Компилятор	Векторизационный отчет	OpenMP SIMD

GCC, G++, GFortran v9	`-fopt-info-vec-optimized[=file]`
	`-fopt-info-vec-missed[=file]`
	`-fopt-info-vec-all[=file]`
	То же самое для оптимизаций циклов замените `vec` на `loop`
		`-fopenmp-simd`

Clang v9	`-Rpass-analysis=loop-vectorize`	`-fopenmp-simd`

Intel icc v19	`-qopt-report=[1-5]`
	`-qopt-report-phase=vec,loop`
	`-qopt-report-filter=`
	`"filename,ln1-ln2"`
		`-qopenmp-simd`

MSVC	`-Qvec-report:[1,2]`	`-openmp:experimental`
IBM XLC v16	`-qreport`	`-qopenmp`
Cray	`-h msgs -h negmsgs` `-h list=a`	`-h omp` (по умолчанию)

Векторные команды можно указывать для любого отдельного набора, такого как AVX2, либо нескольких наборов. Мы покажем вам, как делать и то и другое. Для одного набора команд флаги, показанные в приведенных выше таблицах, требуют, чтобы компилятор использовал набор векторных команд для процессора, используемого для компилирования (`march=native`, `-xHost` и `-qarch=pwer9`). Без этого флага компилятор использует набор SSE2. Если вы заинтересованы в работе с широким спектром процессоров, то вы можете указать более старый набор команд либо просто применить используемый по умолчанию. При этом наблюдается некоторая потеря производительности по сравнению со старыми наборами.

С компилятором Intel можно добавлять поддержку более чем одного набора векторных команд. На практике это широко принято делать для процессора Intel Knights Landing, где набор команд для главного процессора может отличаться. Для этого необходимо указывать оба набора команд:

```
-axmic-avx512 -xcore-avx2
```

`-ax` добавляет дополнительный набор. Обратите внимание, что при запросе двух наборов команд ключевое слово `host` использовать нельзя.

Мы кратко упомянули об использовании флага с плавающей точкой для поощрения векторизации в вычислительном ядре редукции к сумме при обсуждении листинга 6.1. При векторизации циклов с условным блоком компилятор вставляет маску, использующую только часть векторных результатов. Но маскированные операции могут приводить к ошибке с плавающей точкой путем деления на ноль или извлечения квадратного корня из отрицательного числа. Компиляторы GCC и Clang требуют, чтобы дополнительные флаги с плавающей точкой, показанные в последнем столбце табл. 6.2, устанавливались для векторизации циклов с условными блоками и любых потенциально проблемных операций с плавающей точкой.

В некоторых ситуациях вам может потребоваться отключить векторизацию. Отключение векторизации позволяет вам увидеть улучшения и ускорение, которых вы достигли с помощью векторизации. Оно позволяет вам проверять, что вы получаете один и тот же ответ с векторизацией и без нее. Иногда автоматическая векторизация дает вам неправильный ответ, и, следовательно, вы хотели бы ее отключить. Вы также, возможно, захотите векторизовывать только те файлы, которые требуют больших вычислительных затрат, и пропускать остальные.

Результаты векторизации и производительности в этой главе были получены с помощью GCC v8 и v9, а также с помощью компилятора Intel v19. Как отмечено в табл. 6.1, поддержка 512-битовых векторов была добавлена в GCC, начиная с версии 8, а в Intel – начиная с версии 18. Таким образом, возможности нового 512-битового векторного оборудования появились совсем недавно.

Таблица 6.4 Компиляторные флаги для отключения векторизации

Компилятор	Флаг
GCC	`-fno-tree-vectorize` (по умолчанию включен на уровне `-O3`)
Clang	`-fno-vectorize` (по умолчанию включен)
Intel	`-no-vec` (по умолчанию включен на уровне `-O2` или выше)
MSVC	Компиляторный флаг для отключения векторизации отсутствует (по умолчанию включен)
XLC	`-qsimd=noauto` (по умолчанию включен на уровне `-O3`)
Cray	`-h vector0 -hpf0 or -hfp1` (по умолчанию уровень векторизации равен `-h vector2`)

Модуль CMake для установки компиляторных флагов

Отлаживать установки всех флагов для компиляторной векторизации хлопотливо и трудно. Поэтому мы создали модуль CMake, который вы можете использовать и который аналогичен модулям FindOpenMP.cmake и FindMPI.cmake. Тогда главному файлу CMakeLists.txt просто нужно

```
find_package(Vector)
if (CMAKE_VECTOR_VERBOSE)
   set(VECTOR_C_FLAGS "${VECTOR_C_FLAGS} ${VECTOR_C_VERBOSE}")
endif()
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} ${VECTOR_C_FLAGS}")
```

Модуль CMake показан в FindVector.cmake в главном каталоге примеров этой главы по адресу https://github.com/EssentialsofParallelComputing/Chapter6.git. Также обратитесь к примеру кода `GlobalSumsVectorized` с использованием модуля FindVector.cmake. Кроме того, мы будем переносить этот модуль в другие примеры, чтобы очищать наш файл CMakeLists.txt. Следующий ниже листинг является выдержкой из модуля для компилятора языка C. Флаги для C++ и Fortran задаются аналогичным кодом в модуле FindVector.cmake.

Листинг 6.2 Выдержка из FindVector.cmake для компилятора C

FindVector.cmake

```
 8 # Main output flags
 9 # VECTOR_<LANG>_FLAGS
10 # VECTOR_NOVEC_<LANG>_FLAGS
11 # VECTOR_<LANG>_VERBOSE
12 # Component flags
13 # VECTOR_ALIASING_<LANG>_FLAGS
14 # VECTOR_ARCH_<LANG>_FLAGS
15 # VECTOR_FPMODEL_<LANG>_FLAGS
16 # VECTOR_NOVEC_<LANG>_OPT
```

Устанавливает все флаги и включает векторизацию

Устанавливает все флаги, но отключает векторизацию

Включает подробные сообщения во время компилирования для обратной связи по векторизации

Опция более строгой псевдонимизации, способствующая автоматической векторизации

Устанавливаются для компилирования для архитектуры, на которой это находится

Установлены так, чтобы суммирование по Кахану не оптимизировалось (небезопасные оптимизации)

Отключает векторизацию для отладки и измерения производительности

```
17 # VECTOR_VEC_<LANG>_OPTS                    <-- Включает векторизацию
...
25 if(CMAKE_C_COMPILER_LOADED)
26    if ("${CMAKE_C_COMPILER_ID}" STREQUAL "Clang") # using Clang
27       set(VECTOR_ALIASING_C_FLAGS "${VECTOR_ALIASING_C_FLAGS}
           -fstrict-aliasing")
28       if ("${CMAKE_SYSTEM_PROCESSOR}" STREQUAL "x86_64")
29          set(VECTOR_ARCH_C_FLAGS "${VECTOR_ARCH_C_FLAGS}
              -march=native -mtune=native")
30       elseif ("${CMAKE_SYSTEM_PROCESSOR}" STREQUAL "ppc64le")
31          set(VECTOR_ARCH_C_FLAGS "${VECTOR_ARCH_C_FLAGS}
              -mcpu=powerpc64le")
32       elseif ("${CMAKE_SYSTEM_PROCESSOR}" STREQUAL "aarch64")
33          set(VECTOR_ARCH_C_FLAGS "${VECTOR_ARCH_C_FLAGS}
              -march=native -mtune=native")
34       endif ("${CMAKE_SYSTEM_PROCESSOR}" STREQUAL "x86_64")
35
36       set(VECTOR_OPENMP_SIMD_C_FLAGS "${VECTOR_OPENMP_SIMD_C_FLAGS}
           -fopenmp-simd")
37       set(VECTOR_C_OPTS "${VECTOR_C_OPTS} -fvectorize")
38       set(VECTOR_C_FPOPTS "${VECTOR_C_FPOPTS} -fno-math-errno")
39       set(VECTOR_NOVEC_C_OPT "${VECTOR_NOVEC_C_OPT} -fno-vectorize")
40       set(VECTOR_C_VERBOSE "${VECTOR_C_VERBOSE} -Rpass=loop-vectorize
           -Rpass-missed=loop-vectorize -Rpass-analysis=loop-vectorize")
41
42    elseif ("${CMAKE_C_COMPILER_ID}" STREQUAL "GNU") # using GCC
43       set(VECTOR_ALIASING_C_FLAGS "${VECTOR_ALIASING_C_FLAGS}
           -fstrict-aliasing")
44       if ("${CMAKE_SYSTEM_PROCESSOR}" STREQUAL "x86_64")
45          set(VECTOR_ARCH_C_FLAGS "${VECTOR_ARCH_C_FLAGS}
              -march=native -mtune=native")
46       elseif ("${CMAKE_SYSTEM_PROCESSOR}" STREQUAL "ppc64le")
47          set(VECTOR_ARCH_C_FLAGS "${VECTOR_ARCH_C_FLAGS}
              -mcpu=powerpc64le")
48       elseif ("${CMAKE_SYSTEM_PROCESSOR}" STREQUAL "aarch64")
49          set(VECTOR_ARCH_C_FLAGS "${VECTOR_ARCH_C_FLAGS}
              -march=native -mtune=native")
50       endif ("${CMAKE_SYSTEM_PROCESSOR}" STREQUAL "x86_64")
51
52       set(VECTOR_OPENMP_SIMD_C_FLAGS "${VECTOR_OPENMP_SIMD_C_FLAGS}
           -fopenmp-simd")
53       set(VECTOR_C_OPTS "${VECTOR_C_OPTS} -ftree-vectorize")
54       set(VECTOR_C_FPOPTS "${VECTOR_C_FPOPTS} -fno-trapping-math
           -fno-math-errno")
55       if ("${CMAKE_SYSTEM_PROCESSOR}" STREQUAL "x86_64")
56          if ("${CMAKE_C_COMPILER_VERSION}" VERSION_GREATER "7.9.0")
57             set(VECTOR_C_OPTS "${VECTOR_C_OPTS} -mprefer-vector-width=512")
58          endif ("${CMAKE_C_COMPILER_VERSION}" VERSION_GREATER "7.9.0")
59       endif ("${CMAKE_SYSTEM_PROCESSOR}" STREQUAL "x86_64")
60
61       set(VECTOR_NOVEC_C_OPT "${VECTOR_NOVEC_C_OPT} -fno-tree-vectorize")
```

```
62      set(VECTOR_C_VERBOSE "${VECTOR_C_VERBOSE} -fopt-info-vec-optimized
          -fopt-info-vec-missed -fopt-info-loop-optimized
          -fopt-info-loop-missed")
63
64    elseif ("${CMAKE_C_COMPILER_ID}" STREQUAL "Intel") # using Intel C
65      set(VECTOR_ALIASING_C_FLAGS "${VECTOR_ALIASING_C_FLAGS}
          -ansi-alias")
66      set(VECTOR_FPMODEL_C_FLAGS "${VECTOR_FPMODEL_C_FLAGS}
          -fp-model:precise")
67
68      set(VECTOR_OPENMP_SIMD_C_FLAGS "${VECTOR_OPENMP_SIMD_C_FLAGS}
          -qopenmp-simd")
69      set(VECTOR_C_OPTS "${VECTOR_C_OPTS} -xHOST")
70      if ("${CMAKE_C_COMPILER_VERSION}" VERSION_GREATER "17.0.4")
71         set(VECTOR_C_OPTS "${VECTOR_C_OPTS} -qopt-zmm-usage=high")
72      endif ("${CMAKE_C_COMPILER_VERSION}" VERSION_GREATER "17.0.4")
73      set(VECTOR_NOVEC_C_OPT "${VECTOR_NOVEC_C_OPT} -no-vec")
74      set(VECTOR_C_VERBOSE "${VECTOR_C_VERBOSE} -qopt-report=5
          -qopt-report-phase=openmp,loop,vec")
75
76    elseif (CMAKE_C_COMPILER_ID MATCHES "PGI")
77      set(VECTOR_ALIASING_C_FLAGS "${VECTOR_ALIASING_C_FLAGS}
          -alias=ansi")
78      set(VECTOR_OPENMP_SIMD_C_FLAGS "${VECTOR_OPENMP_SIMD_C_FLAGS}
          -Mvect=simd")
79
80      set(VECTOR_NOVEC_C_OPT "${VECTOR_NOVEC_C_OPT} -Mnovect ")
81      set(VECTOR_C_VERBOSE "${VECTOR_C_VERBOSE} -Minfo=loop,inline,vect")
82
83    elseif (CMAKE_C_COMPILER_ID MATCHES "MSVC")
84      set(VECTOR_C_OPTS "${VECTOR_C_OPTS}" " ")
85
86      set(VECTOR_NOVEC_C_OPT "${VECTOR_NOVEC_C_OPT}" " ")
87      set(VECTOR_C_VERBOSE "${VECTOR_C_VERBOSE} -Qvec-report:2")
88
89    elseif (CMAKE_C_COMPILER_ID MATCHES "XL")
90      set(VECTOR_ALIASING_C_FLAGSS "${VECTOR_ALIASING_C_FLAGS}
          -qalias=restrict")
91      set(VECTOR_FPMODEL_C_FLAGSS "${VECTOR_FPMODEL_C_FLAGS} -qstrict")
92      set(VECTOR_ARCH_C_FLAGSS "${VECTOR_ARCH_C_FLAGS} -qhot -qarch=auto
          -qtune=auto")
93
94      set(CMAKE_VEC_C_FLAGS "${CMAKE_VEC_FLAGS} -qsimd=auto")
95      set(VECTOR_NOVEC_C_OPT "${VECTOR_NOVEC_C_OPT} -qsimd=noauto")
96      # оптимизации "вектор long"
97      #set(VECTOR_NOVEC_C_OPT "${VECTOR_NOVEC_C_OPT} -qhot=novector")
98      set(VECTOR_C_VERBOSE "${VECTOR_C_VERBOSE} -qreport")
99
100   elseif (CMAKE_C_COMPILER_ID MATCHES "Cray")
101     set(VECTOR_ALIASING_C_FLAGS "${VECTOR_ALIASING_C_FLAGS}
          -h restrict=a")
```

```
102       set(VECTOR_C_OPTS "${VECTOR_C_OPTS} -h vector=3")
103
104       set(VECTOR_NOVEC_C_OPT "${VECTOR_NOVEC_C_OPT} -h vector=0")
105       set(VECTOR_C_VERBOSE "${VECTOR_C_VERBOSE} -h msgs -h negmsgs
            -h list=a")
106
107    endif()
108
109    set(VECTOR_BASE_C_FLAGS "${VECTOR_ALIASING_C_FLAGS}
         ${VECTOR_ARCH_C_FLAGS} ${VECTOR_FPMODEL_C_FLAGS}")
110    set(VECTOR_NOVEC_C_FLAGS "${VECTOR_BASE_C_FLAGS}
         ${VECTOR_NOVEC_C_OPT}")
111    set(VECTOR_C_FLAGS "${VECTOR_BASE_C_FLAGS} ${VECTOR_C_OPTS}
         ${VECTOR_C_FPOPTS} ${VECTOR_OPENMP_SIMD_C_FLAGS}")
112 endif()
```

6.6 Директивы OpenMP SIMD для более качественной переносимости

С выпуском OpenMP 4.0 у нас появилась возможность использовать более переносимый набор директив SIMD. Эти директивы имплементированы не как подсказки, а в виде команд. Мы уже встречали использование этих директив в разделе 6.3.3. Директивы могут использоваться только для запроса векторизации, либо они могут комбинироваться с директивой `for/do` для запроса как потокообразования, так и векторизации. Общий синтаксис для прагм C и C++ таков:

```
#pragma omp simd      / Векторизует следующий цикл или блок кода
#pragma omp для simd  / Потокообразует и векторизует следующий цикл
```

Общий синтаксис для директив Fortran таков:

```
!$omp simd     / Векторизует следующий цикл или блок кода
!$omp do simd  / Потокообразует и векторизует следующий цикл
```

Базовая директива SIMD может быть дополнена добавочными выражениями для передачи дополнительной информации. Наиболее распространенным дополнительным выражением является некоторый вариант выражения `private`. Оно устраняет ложные зависимости, создавая отдельную приватную переменную для каждой векторной полосы. Вот пример этого синтаксиса:

```
#pragma omp simd private(x)
   for (int i=0; i<n; i++){
      x=array(i);
      y=sqrt(x)*x;
   }
```

В случае простого выражения `private` рекомендуемый подход для программ на языках C и C++ состоит в определении переменной в цикле с целью прояснения вашего намерения:

```
double x=array(i);
```

Выражение `firstprivate` инициализирует приватную переменную для каждого потока значением, входящим в цикл, в то время как выражение `lastprivate` устанавливает переменную после цикла равной логически последнему значению, которое она будет иметь в последовательной форме цикла.

Выражение `reduction` создает приватную переменную для каждой полосы, а затем выполняет указанную операцию между значениями для каждой полосы в конце цикла. Переменные `reduction` инициализируются для каждой векторной полосы, как это имеет смысл для указанной операции.

Выражение `aligned` сообщает компилятору, что данные выровнены по 64-байтовой границе, поэтому не нужно создавать отслаивающие циклы. Выровненные данные могут эффективнее загружаться в векторные регистры. Но сначала память должна быть выделена с выравниванием. Для выравнивания памяти можно применять целый ряд функций, но при этом остаются вопросы с переносимостью. Вот некоторые из возможностей:

```
void *memalign(size_t alignment, size_t size);
int posix_memalign(void **memptr, size_t alignment, size_t size);
void *aligned_alloc(size_t alignment, size_t size);
void *aligned_malloc(size_t alignment, size_t size);
```

Вы также можете применять атрибуты определения памяти для указания выравнивания:

```
double x[100] __attribute__((aligned(64)));
```

Еще одним важным модификатором является выражение `collapse`. Оно говорит компилятору комбинировать вложенные циклы в один цикл для векторизуемой имплементации. Аргумент этого выражения задает число циклов, которые необходимо свернуть:

```
#pragma omp collapse(2)
   for (int j=0; j<n; j++){
      for (int i=0; i<n; i++){
         x[j][i] = 0.0;
      }
   }
```

Циклы должны быть *идеально* вложенными. Идеально вложенные циклы содержат состояния только в самом внутреннем цикле без каких-либо посторонних инструкций до или после каждого циклоблока.

Следующие ниже выражения предназначены для более специализированных случаев.

- Выражение `linear` говорит о том, что переменная изменяется на каждой итерации некоторой линейной функцией.
- Выражение `safelen` сообщает компилятору о том, что зависимости разделены заданной длиной, что позволяет компилятору выполнять векторизацию под векторные длины, меньшие или равные аргументу безопасной длины.
- Выражение `simdlen` генерирует векторизацию заданной длины вместо длины, используемой по умолчанию.

Функции OpenMP SIMD

Мы также можем векторизовывать всю функцию или подпрограмму целиком, вследствие чего она может вызываться изнутри векторизованной области кода. Синтаксис немного отличается для C/C++ и Fortran. Для C/C++ мы будем использовать пример, в котором радиальное расстояние массива точек вычисляется с использованием теоремы Пифагора:

```
#pragma omp declare simd
double pythagorean(double a, double b){
   return(sqrt(a*a + b*b));
}
```

Для Fortran имя подпрограммы или функции должно указываться в качестве аргумента выражения SIMD:

```
subroutine pythagorean(a, b, c)
!$omp declare simd(pythagorean)
real*8 a, b, c
   c = sqrt(a**2+b**2)
end subroutine pythagorean
```

Функциональная директива OpenMP SIMD также может принимать несколько тех же самых выражений и некоторые новые, как показано ниже.

- Выражение `inbranch` или `notinbranch` сообщает компилятору о том, вызывается ли функция из условного блока или нет.
- Выражение `uniform` говорит о том, что заданный в выражении аргумент остается постоянным для всех вызовов и его не нужно настраивать в качестве вектора в векторизованном вызове.
- Выражение `linear(ref, val, uval)` указывает компилятору, что переменная в аргументе выражения является линейной в той или иной форме. Например, Fortran передает аргументы по ссылке и при передаче последующих местоположений массива. В предыдущем примере на Fortran выражение выглядело бы так:

  ```
  !$omp declare simd(pythagorean) linear(ref(a, b, c))
  ```

Это выражение также можно использовать для указания того, что значение является линейным и того, является ли шаг большей константой, как это может произойти при пошаговом доступе.
- Выражения `aligned` и `simdlen` аналогичны использованию в ориентированных на цикл выражениях OpenMP SIMD.

6.7 Материалы для дальнейшего изучения

Вы найдете не так уж много материалов по векторизации. Самое лучшее, что можно сделать для дальнейшего изучения указанной темы, состоит в пробных векторизациях малых блоков кода и экспериментированиях с компиляторами, которые вы обычно используете. Тем не менее у Intel есть масса кратких руководств по векторизации, которые являются наилучшими и наиболее актуальными ресурсами. Последние материалы можно получить, обратившись к веб-сайту Intel.

6.7.1 Дополнительное чтение

Джон Левеск из корпорации Cray является автором недавней книги с неплохой главой о векторизации:

- Джон Левеск и Аарон Воуз, «Программирование для гибридных мультиядерных/многоядерных MPP-систем» (John Levesque, Aaron Vose, *Programming for Hybrid Multi/Manycore MPP Systems*, CRC Press, 2017).

У Агнера Фога есть несколько самых лучших справочных материалов по векторизации в его руководствах по оптимизации, например:

- Агнер Фог, «Оптимизация программного обеспечения на C++: руководство по оптимизации для платформ Windows, Linux и Mac» (Agner Fog, Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms), 2004–2018 (последнее обновление август 2018);
- Агнер Фог, «Библиотека векторных классов VCL C++» (Agner Fog, VCL C++ vector class library), v. 1.30 (2012–2017), доступен в формате PDF по адресу https://www.agner.org/optimize/vectorclass.pdf.

6.7.2 Упражнения

1. Поэкспериментируйте с циклами автоматической векторизации из мультиматериального исходного кода в разделе 4.3 (https://github.com/LANL/MultiMatTest.git). Добавьте флаги отчетности о векторизации и цикле и посмотрите, что конкретно ваш компилятор вам скажет.
2. Добавьте прагмы OpenMP SIMD в цикл, выбранный вами в первом упражнении, чтобы помочь компилятору выполнять векторизацию циклов.

3 В одном из примеров с внутренними векторными функциями измените длину вектора с четырех значений двойной точности на вектор шириной восемь. Просмотрите исходный код этой главы с примерами рабочего кода для имплементаций с шириной восемь.
4 Если вы используете более старый процессор, то насколько успешно выполняется ваша программа из упражнения 3? Каково влияние такого выбора на производительность?

Резюме

И автоматическая, и ручная векторизация может обеспечивать значительное повышение производительности вашего исходного кода. Для того чтобы это подчеркнуть:

- мы показываем несколько разных методов векторизации кода с разными уровнями контроля, объемом усилий и производительностью;
- мы резюмируем правильные компиляторные флаги и их использование;
- мы предоставляем список стилей программирования, рекомендуемых для достижения векторизации.

7 Стандарт OpenMP, который «рулит»

Эта глава охватывает следующие ниже темы:

- планирование и разработку правильной и производительной программы OpenMP;
- написание OpenMP уровня цикла с целью обеспечения скромного параллелизма;
- обнаружение проблем с правильностью и повышение устойчивости;
- устранение проблем с производительностью с помощью OpenMP;
- написание масштабируемого OpenMP для высокой производительности.

По мере роста размеров и популярности многоядерных архитектур детали параллелизма уровня потоков становятся решающим фактором производительности программного обеспечения. В этой главе мы сначала познакомим вас с основами Open Multi-Processing (OpenMP, или открытая мультиобработка), стандарта программирования с совместной памятью, и объясним причину, почему важно иметь фундаментальное понимание принципа функционирования OpenMP. Мы рассмотрим образцы задач, начиная от простого распространенного примера «Здравствуй, мир» и заканчивая сложной, с разводом направлений, стенсильной имплементацией с параллелизацией OpenMP. Мы тщательно проанализи-

руем взаимодействие между директивами OpenMP и опорным ядром OS, а также иерархию памяти и аппаратные функциональности. Наконец, мы рассмотрим перспективный высокоуровневый подход к программированию OpenMP для будущих приложений экстремального масштаба.

Мы показываем, что высокоуровневая парадигма OpenMP эффективна для алгоритмов, содержащих много коротких циклов вычислительной работы.

По сравнению с более стандартными подходами к потокообразованию высокоуровневая парадигма OpenMP приводит к снижению накладных расходов на потоки, ожиданий синхронизации, перетирания кеша и использования памяти. Учитывая эти преимущества, очень важно, чтобы современный программист параллельных вычислений (т. е. вы) знал парадигмы программирования с совместной и распределенной памятью. Мы обсуждаем парадигму программирования распределенной памяти в главе 8, посвященной интерфейсу передачи сообщений (MPI).

ПРИМЕЧАНИЕ Прилагаемый исходный код этой главы можно найти по адресу https://github.com/EssentialsofParallelComputing/Chapter7.

7.1 Введение в OpenMP

OpenMP является одним из наиболее широко поддерживаемых открытых стандартов для потоков и параллельного программирования с совместной памятью. В этом разделе мы объясним сам стандарт, простоту его использования, ожидаемые выгоды, трудности и модели памяти.

На разработку версии стандарта OpenMP, которую вы видите сегодня, ушло немало времени, и он по-прежнему эволюционирует. OpenMP берет свое начало с того времени, когда несколько поставщиков оборудования представили свои имплементации в начале 1990-х годов. В 1994 году была предпринята неудачная попытка стандартизировать эти имплементации в проекте стандарта ANSI X3H5. Только после внедрения широкомасштабных мультиядерных систем в конце 90-х годов возобновилось применение подхода, основанного на OpenMP, что привело к появлению первого стандарта OpenMP в 1997 году.

Сегодня OpenMP предлагает стандартный и переносимый API для написания параллельных программ на основе совместной памяти с использованием потоков; известно, что он прост в использовании, обеспечивает быструю имплементацию и требует лишь малого увеличения кода, обычно рассматриваемого в контексте прагм или директив. Прагма (C/C++) или директива (Fortran) указывает компилятору место, где следует инициировать потоки OpenMP. Термины прагма и директива часто используются как взаимозаменяемые. *Прагмы* – это препроцессорные инструкции на языках C и C++. *Директивы* пишутся в виде комментариев на Fortran, для того чтобы программа сохраняла стандартный синтаксис

языка, когда OpenMP не используется. Хотя для использования OpenMP требуется компилятор, который его поддерживает, большинство компиляторов стандартно комплектуется его поддержкой.

Открытая мультиобработка, OpenMP, делает параллелизацию достижимой для новичка, что позволяет легко и увлекательно приступать к масштабированию приложения за пределами одного ядра. Благодаря простому использованию прагм и директив OpenMP блок кода может быстро исполняться в параллельном режиме. На рис. 7.1 вы видите концептуальный вид требуемых усилий и получаемой производительности для OpenMP и MPI (обсуждаемого в главе 8). Использование OpenMP часто будет первым захватывающим шагом в масштабировании приложения.

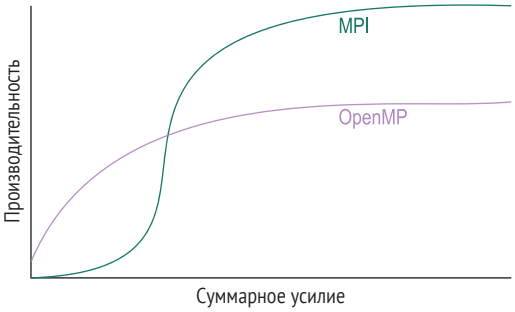


Рис. 7.1 Концептуальная визуализация усилий по программированию, необходимых для повышения производительности с использованием MPI или OpenMP

7.1.1 Концепции OpenMP

Хотя с помощью OpenMP легко добиться умеренного параллелизма, добиться исчерпывающей оптимизации бывает трудно. Источником трудностей является модель ослабленной памяти[1], которая допускает существование гоночных состояний в потоках. Под *ослабленностью* мы подразумеваем, что значения переменных в основной памяти обновляются не сразу. Было бы слишком дорого делать это при каждом изменении переменных. Из-за задержки обновлений незначительные различия в хронометраже между операциями с памятью, выполняемыми каждым потоком с совместными переменными, могут приводить к разным результатам от исполнения к исполнению. Давайте посмотрим на несколько определений.

- *Модель ослабленной памяти* – значения переменных в основной памяти или кешах всех процессоров не обновляются в один момент.
- *Гоночное состояние* (или *условие возникновения гонки*) – ситуация, когда возможно несколько исходов, и результат зависит от хронометража участников.

[1] Точнее, модель ослабленно согласованной, совместной памяти (relaxed-consistency, shared-memory model). – *Прим. перев.*

Стандарт OpenMP изначально использовался для параллелизации очень регулярных циклов с использованием потоков на мультипроцессорах с совместной памятью. В поточном параллельном конструкте каждая переменная может быть совместной либо приватной. Термины «совместный» и «приватный» имеют для OpenMP особый смысл. Вот их определения:

- *приватная переменная* – в контексте OpenMP приватная переменная является локальной и видимой только ее потоку;
- *совместная переменная* – в контексте OpenMP совместная переменная видна и может модифицироваться любым потоком.

Истинное понимание этих терминов требует фундаментального представления о том, как память обрабатывается в поточном приложении. Как показано на рис. 7.2, каждый поток имеет приватную память в своем стеке и использует память в куче коллективно.

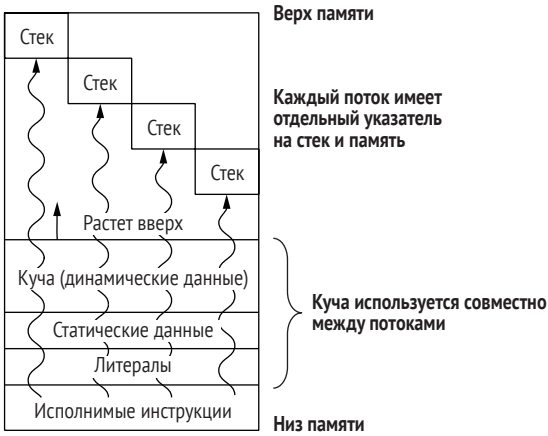


Рис. 7.2 Модель поточной памяти помогает понять то, какие переменные являются совместными, а какие – приватными. Каждый поток, показанный волнистыми линиями, имеет свой собственный указатель на команду, стековый указатель и стековую память, но при этом совместно использует данные кучи и статической памяти

Директивы OpenMP определяют совместное использование работы, но ничего не говорят о памяти или местоположении данных. Как программист, вы должны понимать неявные правила для области переменных в памяти. В ядре OS может использоваться несколько технических приемов управления памятью для OpenMP и потокообразования. Наиболее распространенным методом является концепция первого касания, при которой память выделяется ближе всего к потоку, который был первым, где произошло касание к ней. Вот как мы определяем термины совместного использования работы и первого касания:

- *совместное использование работы* – деление работы между рядом потоков или процессов;

- *первое касание* – первое касание массива приводит к выделению памяти. Память выделяется рядом с местоположением потока, где происходит касание. До первого касания память существует только как элемент в таблице виртуальной памяти. Физическая память, соответствующая виртуальной памяти, создается при первом доступе к ней.

Причина важности первого касания заключается в том, что на многих высококлассных узлах высокопроизводительных вычислений имеется несколько участков памяти. При наличии нескольких участков памяти часто возникает неравномерный доступ к памяти (Non-Uniform Memory Access, NUMA) со стороны CPU и его процессов к разным частям памяти, что добавляет важное соображение относительно оптимизации производительности кода.

ОПРЕДЕЛЕНИЕ На некоторых вычислительных узлах блоки памяти находятся ближе к одним процессорам, чем к другим. Эта ситуация называется *неравномерным доступом к памяти* (NUMA) и часто возникает, когда узел имеет два разъема CPU, причем каждый разъем имеет свою собственную память. Доступ процессора к памяти в другом домене NUMA обычно занимает вдвое больше времени (штраф), чем доступ к собственной памяти.

Более того, поскольку OpenMP имеет модель ослабленной памяти, в OpenMP возникает потребность в барьере или операции опустошения (сброса – flush) для передачи панорамы памяти (вида на память) от потока в другие потоки. *Операция опустошения* гарантирует, что значение перемещается между двумя потоками, не давая возникать гоночным состояниям. Барьер OpenMP опустошает все локально модифицированные значения и синхронизирует потоки. Процедура этого обновления значений выполняется многосложной операцией в оборудовании и операционной системе.

В мультиядерной системе с совместной памятью модифицированные значения в кеше должны сбрасываться в основную память и обновляться. В более новых CPU используется специализированное оборудование для определения того, что на самом деле изменилось, поэтому кеш в десятках ядер обновляется только при необходимости. Но эта операция все еще обходится дорого и вынуждает потоки пробуксовывать в ожидании обновлений. Во многих отношениях это похоже на своего рода операцию, которую вам приходится проделывать, когда вы хотите вынуть флеш-накопитель из компьютера; вы должны сказать операционной системе опустошить все кеши флеш-накопителя, а затем ждете. Исходные коды, в которых используются частые барьеры и опустошения в сочетании с меньшими параллельными участками, часто имеют чрезмерную синхронизацию, что приводит к низкой производительности.

OpenMP адресует один узел, а не несколько узлов с архитектурами распределенной памяти. Следовательно, его масштабируемость памяти ограничена памятью на узле. Для параллельных приложений с больши-

ми требованиями к памяти открытую мультиобработку, OpenMP, нужно использовать в сочетании с параллельной технологией с распределенной памятью. Мы обсудим наиболее распространенную из них, стандарт MPI, в главе 8.

В табл. 7.1 показано несколько распространенных концепций, терминология и директивы OpenMP. Мы продемонстрируем их использование в остальной части этой главы.

Таблица 7.1 Дорожная карта тем OpenMP этой главы

Тема OpenMP	Прагма OpenMP	Описание
Параллельные участки (см. листинг 7.2)	`#pragma omp parallel`	Порождает потоки внутри участка в соответствии с этой директивой

Совместное использование циклической работы (см. листинг 7.7)	`#pragma omp for`
	Для Fortran:
	`#pragma do for`
		Разбивки работают одинаково между потоками. Выражения планирования включают static, dynamic, guided и auto

Параллельный участок и совместное использование работы в сочетании (см. листинг 7.7)	`#pragma omp parallel for`	Директивы также могут комбинироваться для конкретных вызовов внутри подпрограмм
Редукция (см. раздел 7.3.5)	`#pragma omp parallel for reduction (+: sum), (min: xmin)` или `(max: xmax)`	
Синхронизация (см. листинг 7.15)	`#pragma omp barrier`	В случае нескольких выполняющихся потоков этот вызов создает точку остановки, вследствие чего все потоки могут перегруппировываться перед переходом к следующему разделу

Последовательные разделы (см. листинг 7.4 и 7.5)	`#pragma omp masked`
	Исполняется на нулевом потоке без какого-либо барьера в конце[1]
	`#pragma omp single`
	один поток с неявным барьером в конце блока
		Эта директива запрещает исполнение кода несколькими потоками. Используйте эту директиву, если у вас есть функция в параллельном участке, которую вы хотите выполнять только на одном потоке

Замки	`#pragma omp critical` или `atomic`	Для продвинутых имплементаций; используются только в особых случаях

7.1.2 Простая программа стандарта OpenMP

Теперь мы покажем вам, как применять каждую концепцию и директиву OpenMP. В этом разделе вы научитесь создавать участок кода с не-

[1] Прагама #pragma omp masked была прагмой #pragma omp master. С выпуском стандарта OpenMP v 5.1 в ноябре 2020 года термин master был заменен на masked, чтобы устранить опасения в отношении того, что для многих специалистов из технического сообщества он звучит оскорбительно. Мы являемся убежденными сторонниками инклюзивности и, таким образом, используем новый синтаксис на протяжении всей этой главы. Предупреждаем читателей, что имплементирование этого изменения в компиляторах может занять некоторое время. Обратите внимание, что в сопровождающих главу примерах будет использоваться более старый синтаксис до тех пор, пока большинство компиляторов не будет обновлено.

сколькими потоками (виртуальными ядрами), используя параллельную прагму OpenMP для решения традиционной задачи «Здравствуй, мир» распределенно между потоками. Вы увидите, насколько легко использовать OpenMP и, возможно, добиваться повышения производительности. Подходов к управлению числом потоков в параллельной области существует несколько, и они таковы:

- *по умолчанию* – обычно по умолчанию для узла используется максимальное число потоков, но оно может отличаться в зависимости от компилятора и наличия рангов MPI;
- *переменная среды* – задает размер с помощью переменной среды OMP_NUM_THREADS, например:

```
export OMP_NUM_THREADS=16
```

- *вызов функции* – вызов функции OpenMP `omp_set_threads`, например:

```
omp_set_threads(16)
```

- *1*– например:

```
#pragma omp parallel num_threads(16)
```

Простой пример в листингах 7.1–7.6 показывает, как получать ИД потока и число потоков. В листинге 7.1 показана наша первая попытка написания программы «Здравствуй, мир».

Листинг 7.1 Простая программа OpenMP с приветствием, которая печатает `Hello OpenMP`

HelloOpenMP/HelloOpenMP.c

```
 1 #include <stdio.h>
 2 #include <omp.h>
 3
 4 int main(int argc, char *argv[]){
 5    int nthreads, thread_id;
 6    nthreads = omp_get_num_threads();
 7    thread_id = omp_get_thread_num();
 8    printf("Прощай, медленный последовательный мир, и здравствуй, OpenMP!");
 9    printf(" У меня %d потока(а), и ИД моего потока равен %d\n",nthreads,thread_
id);
10 }
```

Вставляет заголовочный файл OpenMP для вызовов функций OpenMP (обязательно)

Вызовы функций для получения числа потоков и ИД потока

Для компилирования с помощью GCC:

```
gcc -fopenmp -o HelloOpenMP HelloOpenMP.c
```

где `-fopenmp` – это компиляционный флаг для активирования OpenMP.

Далее мы установим число потоков, которое будет использоваться программой, задав переменную среды. Мы также могли бы применить вызов функции `omp_set_num_threads()` или просто дать OpenMP выбрать

число потоков, основываясь на оборудовании, на котором мы работаем. Для установки числа потоков используйте вот эту команду, которая задает переменную среды:

```
export OMP_NUM_THREADS=4
```

Теперь запустите исполняемый файл, выполнив ./HelloOpenMP. Мы получим:

```
Прощай, медленный последовательный мир, и здравствуй, OpenMP!
  У меня 1 поток(а), и ИД моего потока равен 0
```

Не совсем то, что мы хотели; у нас всего одна нить. В целях получения нескольких потоков мы должны добавить параллельный участок. В листинге 7.2 показано, как добавлять параллельный участок.

ПРИМЕЧАНИЕ Во всех листингах этой главы вы увидите аннотации с надписями `>> Порождать потоки >>` и `Неявный барьер Неявный барьер`. Это визуальные подсказки, которые обозначают место, где порождаются потоки и где компилятор вставляет барьеры. В последующих листингах мы будем использовать те же аннотации с надписью `Явный барьер Явный барьер`, для обозначения мест, куда мы вставили директиву барьера.

Листинг 7.2 Добавление параллельного участка в Hello OpenMP

HelloOpenMP/HelloOpenMP_fix1.c

```
 1 #include <stdio.h>
 2 #include <omp.h>
 3
 4 int main(int argc, char *argv[]){
 5    int nthreads, thread_id;
 6    #pragma omp parallel >> Порождать потоки >>
 7    {
 8       nthreads = omp_get_num_threads();
 9       thread_id = omp_get_thread_num();
10       printf("Прощай, медленный последовательный мир, и здравствуй, OpenMP!\n");
11       printf(" У меня %d потока(а), и ИД моего потока равен
      %d\n",nthreads,thread_id);
12    } Неявный барьер        Неявный барьер
13}
```

Добавляет параллельный участок

С учетом этих изменений мы получаем следующий результат:

```
Прощай, медленный серийный мир, и здравствуй, OpenMP!
  У меня 4 поток(а), и ИД моего потока равен 3
Прощай, медленный серийный мир, и здравствуй, OpenMP!
  У меня 4 поток(а), и ИД моего потока равен 3
Прощай, медленный серийный мир, и здравствуй, OpenMP!
Прощай, медленный серийный мир, и здравствуй, OpenMP!
```

```
У меня 4 поток(а), и ИД моего потока равен 3
У меня 4 поток(а), и ИД моего потока равен 3
```

Как вы видите, все потоки сообщают, что они являются потоком номер 3. Это связано с тем, что переменные `nthreads` и `thread_id` являются совместными. Значение, присваиваемое этим переменным во время выполнения, соответствует тому, что записывается последним потоком, исполняющим команду. Мы имеем типичное гоночное состояние, как показано на рис. 7.3. Эта проблема широко распространена в поточных программах любого типа.

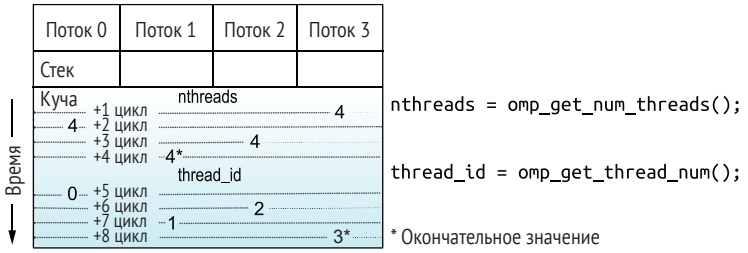


Рис. 7.3 Переменные в приведенном выше примере определяются перед параллельным участком, поэтому они являются совместными переменными в куче. Каждый поток пишет в них, и окончательное значение определяется тем, какой из них пишет последним. Затенение (шейдинг) показывает прогрессию во времени, при этом операции записи в разных тактовых циклах выполняются различными потоками недетерминированным образом. Эта ситуация и аналогичные состояния называются *гоночными*, потому что результаты могут варьироваться от исполнения к исполнению

Также обратите внимание на то, что порядок распечатки является случайным и зависит от порядка операций записи из каждого процессора и того, как они сбрасываются на стандартное выходное устройство. В целях получения правильных номеров потоков мы определяем переменную `thread_id` в цикле, вследствие чего ее область видимости становится приватной для потока, как показано в следующем ниже листинге.

Листинг 7.3 Определение переменных там, где они используются в Hello OpenMP

HelloOpenMP/HelloOpenMP_fix2.c

```
 1 #include <stdio.h>
 2 #include <omp.h>
 3
 4 int main(int argc, char *argv[]){
 5    #pragma omp parallel >> Порождать потоки >>
 6    {
 7       int nthreads = omp_get_num_threads();
 8       int thread_id = omp_get_thread_num();
 9       printf("Прощай, медленный последовательный мир и здравствуй, OpenMP!\n");
```

Определение переменных nthreads и thread_id перемещено в параллельный участок

```
10        printf(" У меня %d потока(а), и ИД моего потока равен
          %d\n",nthreads,thread_id);
11    } Неявный барьер        Неявный барьер
12 }
```

И мы получаем

```
Прощай, медленный серийный мир, и здравствуй, OpenMP!
Прощай, медленный серийный мир, и здравствуй, OpenMP!
  У меня 4 поток(а), и ИД моего потока равен 2
Прощай, медленный серийный мир, и здравствуй, OpenMP!
  У меня 4 поток(а), и ИД моего потока равен 3
Прощай, медленный серийный мир, и здравствуй, OpenMP!
  У меня 4 поток(а), и ИД моего потока равен 0
  У меня 4 поток(а), и ИД моего потока равен 1
```

Теперь для каждого потока мы получаем разный ИД

Допустим, в действительности мы не хотели печатать каждый поток. Давайте сведем распечатку к минимуму и поместим инструкцию `print` в одно выражение OpenMP, как показано в следующем ниже листинге, вследствие чего результат пишется только одним потоком.

Листинг 7.4 Добавление в Hello OpenMP одной прагмы для печати результата

HelloOpenMP/HelloOpenMP_fix3.c

```
 1 #include <stdio.h>
 2 #include <omp.h>
 3
 4 int main(int argc, char *argv[]){
 5    #pragma omp parallel >> Порождать потоки >>
 6    {
 7       int nthreads = omp_get_num_threads();
 8       int thread_id = omp_get_thread_num();
 9       #pragma omp single
10       {
11          printf("Число потоков равно %d\n",nthreads);
12          printf("ИД моего потока равен %d\n",thread_id);
13       } Неявный барьер        Неявный барьер
14    } Неявный барьер        Неявный барьер
15 }
```

Переменные, определенные в параллельном участке, являются приватными

Помещает инструкции вывода результата в блок прагмы OpenMP single

И результат теперь таков:

```
Число потоков равно 4
  ИД моего потока равен 2
```

ИД потока имеет разное значение при каждом исполнении. Здесь в действительности мы хотели, чтобы выводимый поток был первым потоком, поэтому мы изменили выражение OpenMP в следующем ниже листинге, и вместо `single` используем `masked`.

Листинг 7.5 Замена прагмы single на прагму masked в Hello OpenMP

HelloOpenMP/HelloOpenMP_fix4.c

```
 1 #include <stdio.h>
 2 #include <omp.h>
 3
 4 int main(int argc, char *argv[]){
 5    #pragma omp parallel >> Порождать потоки >>
 6    {
 7       int nthreads = omp_get_num_threads();
 8       int thread_id = omp_get_thread_num();
 9       #pragma omp masked
10       {
11          printf("Прощай, медленный серийный мир, и здравствуй, OpenMP!\n");
12          printf(" У меня 4 поток(а), и ИД моего потока равен
     %d\n",nthreads,thread_id);
13       }
14    } Неявный барьер        Неявный барьер
15 }
```

Добавляет директиву для выполнения только на главном потоке

Выполнение этого исходого кода теперь возвращает то, что мы пытались сделать с самого сначала:

```
Прощай, медленный последовательный мир, и привет, OpenMP!
  У меня 4 потока(а), и ИД моего потока равен 0
```

Мы можем сделать эту операцию еще более краткой и использовать меньше прагм, как показано в листинге 7.6. Первая инструкция печати не обязательно должна находиться в параллельном участке. Кроме того, мы можем ограничить вторую распечатку нулевым потоком, просто применив условный блок к номеру потока. Неявный барьер связан с прагмой `omp parallel`.

Листинг 7.6 Сокращение числа прагм в Hello OpenMP

HelloOpenMP/HelloOpenMP_fix5.c

```
 1 #include <stdio.h>
 2 #include <omp.h>
 3
 4 int main(int argc, char *argv[]){
 5    printf("Прощай, медленный серийный мир и привет OpenMP!\n");
 6    #pragma omp parallel >> Порождать потоки >>
 7    if (omp_get_thread_num() == 0) {
 8       printf(" У меня %d поток(а), и ИД моего потока равен %d\n",
             omp_get_num_threads(), omp_get_thread_num());
 9    }
10    Неявный барьер        Неявный барьер
11 }
```

Выносит инструкцию печати за пределы параллельного участка

Заменяет прагму OpenMP masked условным блоком для нулевого потока

Прагма применяется к следующей инструкции или ограничивающему область видимости блоку, выделенному фигурными скобками

Из этого примера мы узнали несколько важных вещей.

- Переменные, определяемые вне параллельного участка, по умолчанию являются *совместными* в параллельном участке.
- Мы всегда должны стремиться к тому, чтобы для переменной иметь наименьшую область видимости, которая по-прежнему остается правильной. Определяя переменную в цикле, мы наделяем компилятор более точным пониманием наших намерений и возможностью правильно их обрабатывать.
- Использование выражения `masked` вляется более лимитирующим, чем выражение `single`, поскольку для исполнения блока кода требуется поток 0. Кроме того, выражение `masked` не имеет неявного барьера в конце.
- Нам нужно следить за возможными гоночными состояниями между операциями разных потоков.

Стандарт OpenMP постоянно обновляется, и новые версии выпускаются регулярно. Прежде чем использовать имплементацию OpenMP, вы должны знать ее версию и поддерживаемые функциональности. OpenMP начинался со способности использовать потоки на одном узле. В стандарт OpenMP были добавлены новые способности, такие как векторизация и выгрузка задач на ускорители, такие как GPU. В следующей ниже таблице показано несколько главенствующих функциональностей, добавленных за последнее десятилетие.

Версия 3.0 (2008)	Введение параллелизма операционных задач и усовершенствование параллелизма циклов. Усовершенствования параллелизма циклов включают сворачивание цикла и вложенный параллелизм
Версия 3.1 (2011)	Добавляет редукционные операторы min и max в C и C++ (другие операторы уже есть в C и C++; min и max уже есть в Fortran) и управление привязкой потоков
Версия 4.0 (2013)	Добавляет (векторизационную) директиву OpenMP SIMD, целевую директиву для выгрузки на GPU-процессоры и другие ускорительные устройства, а также управление аффинностью потоков
Версия 4.5 (2015)	Существенные усовершенствования в поддержке ускорителей для GPU-процессоров
Версия 5.0 (2018)	Дальнейшие усовершенствования в поддержке ускорителей

Следует отметить одну вещь – в связи с происходящими с 2011 года существенными изменениями в оборудовании темпы изменений в OpenMP возросли. В то время как изменения в версиях 3.0 и 3.1 касались в основном стандартной модели потокообразования CPU, с тех пор изменения в версиях 4.0, 4.5 и 5.0 в основном касались других форм аппаратного параллелизма, таких как ускорители и векторизация.

7.2 Типичные варианты использования OpenMP: уровень цикла, высокий уровень и MPI плюс OpenMP

OpenMP имеет три конкретных сценария использования для удовлетворения потребностей трех различных типов пользователей. Первое решение, которое вам нужно принять, – это какой сценарий подходит для вашей ситуации. Стратегия и методы различаются для каждого из этих случаев: OpenMP уровня цикла, OpenMP высокого уровня и OpenMP для улучшения имплементаций MPI. В последующих разделах мы подробно остановимся на каждом из них, когда их использовать, почему и как их использовать. На рис. 7.4 показан рекомендуемый материал, который следует внимательно прочитать по каждому варианту использования.

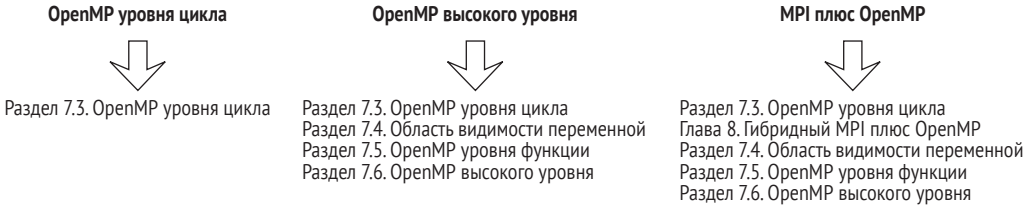


Рис. 7.4 Рекомендуемые разделы для чтения по каждому сценарию зависят от варианта использования вашего приложения

7.2.1 OpenMP уровня цикла для быстрой параллелизации

Стандартный вариант использования OpenMP уровня цикла представлен ситуацией, когда вашему приложению требуется лишь умеренное ускорение и предостаточно ресурсов памяти. Под ней мы подразумеваем, что ее требования могут быть удовлетворены памятью на одном аппаратном узле. В таком случае бывает достаточно использовать OpenMP уровня цикла. В следующем ниже списке приведены характеристики применения OpenMP уровня цикла:

- скромный параллелизм;
- имеет предостаточно ресурсов памяти (низкие требования к памяти);
- дорогостоящая часть расчета заключена всего в нескольких циклах `for` или `do`.

В таких случаях мы используем OpenMP уровня цикла, потому что это не требует особых усилий и делается быстро. С отдельной прагмой `parallel for` проблема гоночных состояний между потоками уменьшается. Размещая прагмы OpenMP `parallel for` или директивы `parallel do` пе-

ред ключевыми циклами, можно легко достигать параллельности цикла. Даже когда конечной целью является более эффективная имплементация, данный подход на уровне цикла нередко является первым шагом при внедрении поточного параллелизма в приложение.

> **ПРИМЕЧАНИЕ** Если ваш вариант использования требует лишь умеренного ускорения, то перейдите к разделу 7.3, где приведены примеры этого подхода.

7.2.2 OpenMP высокого уровня для улучшенной параллельной производительности

Далее мы обсудим другой сценарий, OpenMP высокого уровня, где требуется более высокая производительность. Наш дизайн OpenMP высокого уровня радикально отличается от стратегий для стандартного OpenMP уровня цикла. Стандартный OpenMP начинает в восходящем порядке и применяет конструкты параллелизма на уровне цикла. Наш подход для OpenMP высокого уровня принимает общесистемный взгляд на дизайн с использованием нисходящего подхода, который учитывает систему памяти, системное ядро и оборудование. Язык OpenMP не меняется, но меняется метод его использования. В конечном результате мы устраняем многие затраты на запуск потоков и затраты на синхронизацию, которые препятствуют масштабируемости OpenMP уровня цикла.

Если вам нужно извлечь из вашего приложения всю производительность до последней капли, то OpenMP высокого уровня предназначен для вас. В качестве отправной точки вашего приложения начните с изучения OpenMP уровня цикла в разделе 7.3. Затем вам нужно будет получить более глубокое понимание области видимости переменных OpenMP из разделов 7.4 и 7.5. Наконец, погрузитесь в раздел 7.6, чтобы взглянуть на то, как подход OpenMP высокого уровня, диаметрально противоположный подходу уровня цикла, приводит к повышению производительности. В том разделе мы обратимся к имплементационной модели и пошаговому методу достижения желаемой структуры. Далее следуют подробные примеры имплементаций для OpenMP высокого уровня.

7.2.3 MPI плюс OpenMP для максимальной масштабируемости

Мы также можем использовать OpenMP для дополнения параллелизма распределенной памяти (как описано в главе 8). Базовая идея использования OpenMP на малом подмножестве процессов добавляет еще один уровень параллельной имплементации, который помогает в экстремальном масштабировании. Это может быть внутри узла либо, еще лучше, набором процессоров, которые равномерно распределяют быстрый доступ к совместной памяти, обычно именуемой участком неравномерного доступа к памяти (NUMA).

Мы впервые обсудили участки NUMA в концепциях OpenMP в разделе 7.1.1 в качестве дополнительного соображения для оптимизации производительности. Используя потокообразование только внутри одного участка памяти, где все обращения к памяти имеют одинаковую стоимость, можно избежать некоторых сложностей и ловушек производительности OpenMP. В более скромной гибридной имплементации открытая мультиобработка (OpenMP) может использоваться для обуздания от двух до четырех гиперпотоков для каждого процессора. Мы обсудим этот сценарий, гибридный подход MPI + OpenMP, в главе 8 после описания основ MPI.

Для того чтобы приобрести умения работать с OpenMP, необходимые для этого гибридного подхода с малым числом потоков, достаточно изучить методы OpenMP уровня цикла в разделе 7.3. Затем постепенно переходить к более эффективной и масштабируемой имплементациии OpenMP, которая позволяет заменять ранги MPI все большим числом потоков. Для этого требуются по крайней мере несколько шагов на пути к OpenMP высокого уровня, как описано в разделе 7.6. Теперь, когда вы знаете, какие конкретно разделы важны для варианта использования вашего приложения, давайте перейдем к деталям организации работы каждой стратегии.

7.3 Примеры стандартного OpenMP уровня цикла

В этом разделе мы рассмотрим примеры параллелизации уровня цикла. Вариант использования уровня цикла был представлен в разделе 7.2.1; здесь мы покажем вам детали имплементации. Давайте начнем.

Параллельные участки инициируются вставкой прагм вокруг блоков кода, которые могут быть поделены между независимыми потоками (т. е. циклами do и циклами for). В обработке памяти стандарт OpenMP опирается на ядро OS. Эта зависимость от обработки памяти часто бывает важным фактором, который ограничивает OpenMP, не давая достигать своего максимального потенциала. Мы взглянем на причину, почему это происходит. Каждая переменная в параллельном конструкте может быть как совместной, так и приватной. Кроме того, OpenMP имеет модель ослабленной памяти. Каждый поток имеет временную панораму памяти, вследствие чего у него нет затрат на хранение памяти при каждой операции. Когда временная панорама, наконец, должна быть согласована с основной памятью, для синхронизации памяти требуется барьер OpenMP или операция опустошения (сброса). Каждая из этих синхронизаций сопряжена с некоторыми затратами в силу времени, необходимого для опустошения, а также в силу необходимости, чтобы быстрые потоки ждали завершения более медленных. Понимание принципа функционирования OpenMP поможет уменьшить эти узкие места в производительности.

Производительность не единственная проблема для программиста OpenMP. Вы также должны следить за проблемами правильности, вызываемыми гоночными состояниями между потоками. Потоки могут прогрессировать на процессорах с разной скоростью, и в сочетании с ослабленной синхронизацией памяти внезапно могут возникать серьезные ошибки даже в хорошо протестированном коде. Для строительства устойчивых приложений OpenMP необходимы выверенное программирование и использование специализированных инструментов, как описано в разделе 7.9.2.

В этом разделе мы взглянем на несколько примеров OpenMP уровня цикла, чтобы получить представление о вариантах его использования на практике. Сопровождающий эту главу исходный код содержит еще больше вариантов каждого примера. Мы настоятельно рекомендуем вам поэкспериментировать с каждым из них относительно архитектуры и компилятора, с которыми вы обычно работаете. Мы выполнили все примеры в двухразъемной системе Skylake Gold 6152, а также на ноутбуке Mac 2017 года. Потоки выделяются в соответствии с ядрами, и привязка потоков включена с использованием следующих ниже переменных среды OpenMP с целью уменьшения вариаций в производительности прогонов:

```
export OMP_PLACES=cores
export OMP_CPU_BIND=true
```

Мы разведаем темы размещения и привязывания потоков в главе 14. А пока в целях оказания вам помощи в получении опыта работы с OpenMP уровня цикла мы представим три разных примера: векторное сложение, потоковую триаду и стенсильный код. Мы покажем параллельное ускорение этих трех примеров после последнего примера в разделе 7.3.4.

7.3.1 OpenMP уровня цикла: пример векторного сложения

В примере векторного сложения (листинг 7.7) вы видите взаимодействие между тремя компонентами: директивами OpenMP совместного использования работы, неявной областью видимости переменной и размещением памяти, выполняемой операционной системой. Эти три компонента необходимы для обеспечения правильности и производительности программы OpenMP.

Листинг 7.7 Векторное сложение с простой прагмой OpenMP уровня цикла

VecAdd/vecadd_opt1.c

```
 1 #include <stdio.h>
 2 #include <time.h>
 3 #include <omp.h>
 4 #include "timer.h"
 5
```

```
                                          Массив достаточно велик,
                                          чтобы втиснуть его в основную память
 6 #define ARRAY_SIZE 80000000 <-----------
 7 static double a[ARRAY_SIZE], b[ARRAY_SIZE], c[ARRAY_SIZE];
 8
 9 void vector_add(double *c, double *a, double *b, int n);
10
11 int main(int argc, char *argv[]){
12    #pragma omp parallel >> Порождать потоки >>
13       if (omp_get_thread_num() == 0)
14          printf("Выполняется с %d потоком(ами)\n",omp_get_num_threads());
      Неявный барьер        Неявный барьер
15
16    struct timespec tstart;
17    double time_sum = 0.0;
18    for (int i=0; i<ARRAY_SIZE; i++) {
19       a[i] = 1.0;                          Цикл инициализации
20       b[i] = 2.0;
21    }
22
23    cpu_timer_start(&tstart);
24    vector_add(c, a, b, ARRAY_SIZE);
25    time_sum += cpu_timer_stop(tstart);
26
27    printf("Время выполнения составляет %lf msecs\n", time_sum);
28 }
29
30 void vector_add(double *c, double *a, double *b, int n)
31 {                                                    Прагма OpenMP parallel for
32    #pragma omp parallel for >> Порождать потоки >> <-- в комбинации с single
33    for (int i=0; i < n; i++){
34       c[i] = a[i] + b[i];                  Цикл векторного
35    }                                       сложения
      Неявный барьер        Неявный барьер
36 }
```

Этот конкретный имплементационный стиль обеспечивает скромную параллельную производительность на одном узле. Обратите внимание, что эта имплементация могла бы быть лучше. Первое касание всей памяти массива происходит со стороны главного потока во время инициализации перед главным циклом, как показано слева на рис. 7.5. Это может привести к тому, что его память будет расположена в другом участке памяти, где для некоторых потоков время доступа к памяти больше.

Теперь в целях улучшения производительности OpenMP мы вставляем прагмы в циклы инициализации, как показано в листинге 7.8. Указанные циклы распределены в одном и том же статическом поточном разделе, поэтому потоки, которые осуществляют касание памяти в цикле инициализации, будут иметь память, которая будет помещена операционной системой рядом с ними (показано на правой стороне рис. 7.5).

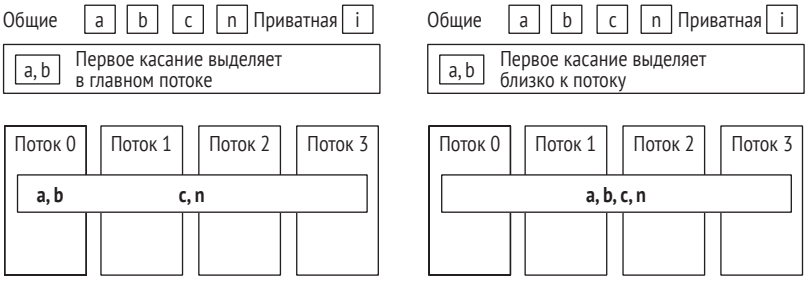


Рис. 7.5 Добавление прагмы OpenMP single в главный цикл вычисления векторного сложения (слева) приводит к тому, что первое касание массивов a и b происходит со стороны главного потока; данные выделяются вблизи нулевого потока. Первое касание массива c происходит во время вычислительного цикла, и, следовательно, память под массив c находится близко к каждому потоку. Справа добавление прагмы OpenMP в цикл инициализации приводит к тому, что память под массивы a и b размещается рядом с потоком, в котором выполняется работа

Листинг 7.8 Векторное сложение с первым касанием

VecAdd/vecadd_opt2.c

```
11 int main(int argc, char *argv[]){
12    #pragma omp parallel >> Порождать потоки >>
13       if (omp_get_thread_num() == 0)
14          printf("Выполняется с %d потоком(ами)\n",omp_get_num_threads());
      Неявный барьер          Неявный барьер
15
16    struct timespec tstart;
17    double time_sum = 0.0;
18    #pragma omp parallel for >> Порождать потоки >>
19    for (int i=0; i<ARRAY_SIZE; i++) {
20       a[i] = 1.0;
21       b[i] = 2.0;
22    }
      Неявный барьер          Неявный барьер
23
24    cpu_timer_start(&tstart);
25    vector_add(c, a, b, ARRAY_SIZE);
26    time_sum += cpu_timer_stop(tstart);
27
28    printf("Время выполнения составляет %lf msecs\n", time_sum);
29 }
30
31 void vector_add(double *c, double *a, double *b, int n)
32 {
33    #pragma omp parallel for >> Порождать потоки >>
```

Инициализация в прагме parallel for, чтобы первое касание помещало память в надлежащее место

Инициализирует массивы a и b

Прагма OpenMP for, чтобы распределять работу цикла векторного сложения по потокам

```
34    for (int i=0; i < n; i++){
35       c[i] = a[i] + b[i];          Цикл векторного сложения
36    }
      Неявный барьер                 Неявный барьер
37 }
```

Потоки во втором участке NUMA больше не имеют более медленного времени доступа к памяти. Это улучшает пропускную способность памяти для потоков во втором участке NUMA, а также улучшает баланс нагрузки между потоками. Первое касание является политикой операционной системы, которая упоминалась ранее в разделе 7.1.1. Хорошие имплементации первого касания нередко могут повышать производительность на 10–20 %. В целях подтверждения того, что это так, обратитесь к табл. 7.2 в разделе 7.3.4, в которой приводятся показатели улучшения производительности в этих примерах.

Если в BIOS активировать NUMA, то CPU Skylake Gold 6152 имеет снижение производительности при доступе к удаленной памяти почти в два раза. Как и в случае с большинством настраиваемых параметров, конфигурация отдельных систем может варьироваться. В целях ознакомления со своей конфигурацией в Linux можно использовать команды `numactl` и `numastat`. Возможно, для исполнения этих команд вам придется установить пакеты numactl-libs или numactl-devel.

На рис. 7.6 показаны результаты работы указанной выше тестовой платформы Skylake Gold. Перечисленные в конце расстояния между уз-

- numactl --hardware

```
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 44 45 46 47 48 49 50 51
52 53 54 55 56 57 58 59 60 61 62 63 64 65
node 0 size: 195243 MB
node 0 free: 189993 MB
node 1 cpus: 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87
node 1 size: 196608 MB
node 1 free: 190343 MB
node distances:
node   0   1
  0:  10  21
  1:  21  10
```

- numastat

```
                           node0           node1
numa_hit               311179492       298484169
numa_miss                      0               0
numa_foreign                   0               0
interleave_hit            106713          106520
local_node             310521578       296233768
other_node                657914         2250401
```

Рис. 7.6 Результат работы команд `numactl` и `numastat`. Выделено расстояние между участками памяти. Обратите внимание, что в утилитах NUMA термин «узел» используется иначе, чем мы его определили. В их терминологии каждый участок NUMA является узлом. Мы резервируем термин узла за отдельной системой распределенной памяти, такой как еще один настольный компьютер или полка в системе, смонтированной в стойке

лами примерно отражают стоимость доступа к памяти на удаленном узле. Это можно рассматривать как относительное число переходов (прыжков), необходимых для доступа к памяти. Здесь стоимость доступа к памяти немного превышает двукратную (21 против 10). Обратите внимание, что иногда конфигурация двух систем участков NUMA по умолчанию указывается со стоимостью 20 против 10 вместо их реальной стоимости.

Информация о конфигурации NUMA может дать вам подсказку в отношении того, что важнее оптимизировать. Если у вас только один участок NUMA или разница в затратах на доступ к памяти невелика, то вам, возможно, не придется столь сильно беспокоиться об оптимизации первого касания. Если система настроена под чередующиеся обращения к участкам NUMA, то оптимизация под более быстрый доступ к локальной памяти не поможет. В отсутствие конкретной информации или при попытке оптимизации в целом для более крупных систем HPC для более быстрого доступа к локальной памяти следует использовать оптимизации первого касания.

7.3.2 Пример потоковой триады

В следующем ниже листинге показан еще один похожий пример со сравнительным тестированием потоковой триады (stream triad). В этом примере выполняется несколько итераций вычислительного ядра, чтобы получить среднюю производительность:

Листинг 7.9 Потокообразование OpenMP уровня цикла для потоковой триады

StreamTriad/stream_triad_opt2.c

```
 1 #include <stdio.h>
 2 #include <time.h>
 3 #include <omp.h>
 4 #include "timer.h"
 5
 6 #define NTIMES 16
 7 #define STREAM_ARRAY_SIZE 80000000
 8 static double a[STREAM_ARRAY_SIZE], b[STREAM_ARRAY_SIZE],
               c[STREAM_ARRAY_SIZE];
 9
10 int main(int argc, char *argv[]){
11    #pragma omp parallel >> Порождать потоки >>
12       if (omp_get_thread_num() == 0)
13          printf("Выполняется с %d потоком(ами)\n",omp_get_num_threads());
      Неявный барьер          Неявный барьер
14
15    struct timeval tstart;
```

Массив достаточно велик, чтобы втиснуть его в основную память

```
16    double scalar = 3.0, time_sum = 0.0;
17    #pragma omp parallel for >> Порождать потоки >>
18    for (int i=0; i<STREAM_ARRAY_SIZE; i++) {
19       a[i] = 1.0;
20       b[i] = 2.0;
21    }
      Неявный барьер          Неявный барьер
22
23    for (int k=0; k<NTIMES; k++){
24       cpu_timer_start(&tstart);
25       #pragma omp parallel for >> Порождать потоки >>
26       for (int i=0; i<STREAM_ARRAY_SIZE; i++){
27          c[i] = a[i] + scalar*b[i];
28       }
         Неявный барьер          Неявный барьер
29       time_sum += cpu_timer_stop(tstart);
30       c[1]=c[2];
31    }
32
33    printf("Среднее время выполнения составляет %lf msecs\n", time_sum/NTIMES);
34 }
```

Инициализирует данные и массивы

Цикл потоковой триады

Не дает компилятору оптимизировать цикл

Опять же, для имплементирования поточных (threaded) вычислений OpenMP в строке 25 нам нужна только одна прагма. Вторая прагма, вставленная в строку 17, дополнительно улучшает производительность за счет более качественного размещения памяти, полученного благодаря надлежащей техники первого касания.

7.3.3 OpenMP уровня цикла: стенсильный пример

Третьим примером OpenMP уровня цикла является стенсильная операция, впервые представленная в главе 1 (рис. 1.10). Стенсильный оператор добавляет окружающих соседей и берет среднее значение для нового значения ячейки. В листинге 7.10 представлены более сложные шаблоны доступа с чтением памяти, и по мере оптимизации процедуры он показывает нам эффект потоков, когда они обращаются к памяти, в которую писали другие потоки. В этой первой имплементации OpenMP уровня цикла каждый блок `parallel for` синхронизируется по умолчанию, что предотвращает возможные гоночные состояния. В более поздних, более оптимизированных версиях стенсиля мы добавим директивы синхронизации в явной форме.

Листинг 7.10 Потокообразование OpenMP уровня цикла в стенсильном примере с первым касанием

Stencil/stencil_opt2.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
```

```
 3 #include <time.h>
 4 #include <omp.h>
 5
 6 #include "malloc2D.h"
 7 #include "timer.h"
 8
 9 #define SWAP_PTR(xnew,xold,xtmp) (xtmp=xnew, xnew=xold, xold=xtmp)
10
11 int main(int argc, char *argv[])
12 {
13    #pragma omp parallel >> Порождать потоки >>
14    #pragma omp masked
15       printf("Выполняется с %d потоком(ами)\n",omp_get_num_threads());
      Неявный барьер          Неявный барьер
16
17    struct timeval tstart_init, tstart_flush, tstart_stencil, tstart_total;
18    double init_time, flush_time, stencil_time, total_time;
19    int imax=2002, jmax = 2002;
20    double** xtmp;
21    double** x = malloc2D(jmax, imax);
22    double** xnew = malloc2D(jmax, imax);
23    int *flush = (int *)malloc(jmax*imax*sizeof(int)*4);
24
25    cpu_timer_start(&tstart_total);
26    cpu_timer_start(&tstart_init);
27    #pragma omp parallel for >> Порождать потоки >>
28    for (int j = 0; j < jmax; j++){
29       for (int i = 0; i < imax; i++){
30          xnew[j][i] = 0.0;
31          x[j][i] = 5.0;
32       }
33    } Неявный барьер          Неявный барьер
34
35    #pragma omp parallel for >> Порождать потоки >>
36    for (int j = jmax/2 - 5; j < jmax/2 + 5; j++){
37       for (int i = imax/2 - 5; i < imax/2 -1; i++){
38          x[j][i] = 400.0;
39       }
40    } Неявный барьер          Неявный барьер
41    init_time += cpu_timer_stop(tstart_init);
42
43    for (int iter = 0; iter < 10000; iter++){
44       cpu_timer_start(&tstart_flush);
45       #pragma omp parallel for >> Порождать потоки >>
46       for (int l = 1; l < jmax*imax*4; l++){
47          flush[l] = 1.0;
48       } Неявный барьер          Неявный барьер
49       flush_time += cpu_timer_stop(tstart_flush);
50       cpu_timer_start(&tstart_stencil);
51       #pragma omp parallel for >> Порождать потоки >>
52       for (int j = 1; j < jmax-1; j++){
53          for (int i = 1; i < imax-1; i++){
```

Инициализирует с помощью прагмы OpenMP выделения памяти при первом касании

Вставляет прагму parallel for в цикл потока

```
54            xnew[j][i]=(x[j][i] + x[j][i-1] + x[j][i+1] +
                          x[j-1][i] + x[j+1][i])/5.0;
55         }
56      } Неявный барьер          Неявный барьер
57      stencil_time += cpu_timer_stop(tstart_stencil);
58
59      SWAP_PTR(xnew, x, xtmp);
60      if (iter%1000 == 0) printf("Iter %d\n",iter);
61   }
62   total_time += cpu_timer_stop(tstart_total);
63
64   printf("Хронометраж: иниц %f опустош %f стенсиль %f всего %f\n",
65          init_time,flush_time,stencil_time,total_time);
66
67   free(x);
68   free(xnew);
69   free(flush);
70 }
```

В этом примере мы вставили цикл опустошения (сброса) в строку 46, чтобы опустошить кеш массивов x и xnew. Это делается для имитации производительности, когда код не имеет переменных в кеше от предыдущей операции. Случай без данных в кеше называется *холодным кешем*, а когда данные в кеше есть, он называется *теплым*. Как холодный, так и теплый кеши являются валидными случаями для анализа в разных сценариях использования. Просто оба случая возможны в реальном приложении, и без глубокого анализа даже бывает трудно понять, что произойдет конкретно.

7.3.4 Производительность примеров уровня цикла

Давайте проведем обзор производительности предыдущих примеров в этом разделе. Как видно из листингов 7.8, 7.9 и 7.10, введение OpenMP уровня цикла требует незначительных изменений в исходном коде. Как видно из табл. 7.2, производительность улучшается примерно в 10-кратном размере. Это довольно неплохая отдача от требуемых усилий. Но для системы с 88 потоками достигнутая параллельная эффективность явно скромна и составляет около 19 %, как показано ниже, что дает нам некоторые возможности для совершенствования. Для того чтобы рассчитать ускорение, мы сначала берем время последовательного выполнения, деленное на время параллельного выполнения, как вот здесь:

Стенсильное ускорение = (время последовательного выполнения) / (время параллельного выполнения) = в 17.0 раз быстрее.

Если мы получим идеальное ускорение на 88 потоках, оно будет 88-кратным. Мы берем фактическое ускорение и делим на идеальное ускорение 88, чтобы рассчитать параллельную эффективность:

Стенсильная параллельная эффективность = (стенсильное ускорение) / (идеальное ускорение) = 17 / 88 = 19 %.

Параллельная эффективность намного выше при меньшем числе потоков; при четырех потоках она составляет 85 %. Эффект от выделения памяти рядом с потоком невелик, но значителен. В хронометражах, приведенных в табл. 7.2, первая оптимизация, простая открытая мультиобработка (OpenMP) уровня цикла, имеет прагму OpenMP `parallel for` только на вычислительных циклах. Вторая оптимизация с первым касанием добавляет прагму OpenMP `parallel for` для циклов инициализации. В табл. 7.2 подытожены улучшения производительности для простой OpenMP с добавлением оптимизации первого касания. В хронометражах использовались `OMP_PLACES=cores` и `OMP_CPU_BIND=true`.

Таблица 7.2 Показано время выполнения в мс. Ускорение на двухразъемном узле Skylake Gold 6152 с компилятором GCC версии 8.2 является десятикратным на 88 потоках. Добавление прагмы OpenMP на инициализации для надлежащего выделения памяти при первом касании обеспечивает дополнительное ускорение

	Последовательно	Простая OpenMP уровня цикла	Добавление первого касания
Векторное сложение	0.253	0.0301	0.0175
Потоковая триада	0.164	0.0203	0.0131
Стенсиль	62.56	3.686	3.311

Профилируя стенсильное приложение, потокообразованное с помощью OpenMP, мы наблюдаем, что 10–15 % времени выполнения приходится на накладные расходы OpenMP, состоящие из ожидания потока и затрат на запуск потока. Накладные расходы OpenMP можно уменьшить, приняв высокоуровневый дизайн OpenMP, как мы обсудим в разделе 7.6.

7.3.5 Пример редукции на основе глобальной суммы с использованием потокообразования OpenMP

Еще одним распространенным типом цикла является редукция. *Редукция* – это распространенный шаблон параллельного программирования, который был представлен в разделе 5.7. Редукциями являются любая операция, которая начинается с массива и вычисляет скалярный результат. В OpenMP она также может легко обрабатываться в прагме уровня цикла с добавлением выражения reduction, как показано в следующем ниже листинге.

Листинг 7.11 Глобальная сумма с потокообразованием OpenMP

`GlobalSums/serial_sum_novec.c`

```
1 double do_sum_novec(double* restrict var, long ncells)
2 {
```

```
 3    double sum = 0.0;
 4    #pragma omp parallel for reduction(+:sum)
 5    for (long i = 0; i < ncells; i++){
 6      sum += var[i];
 7    }
 8
 9    return(sum);
10 }
```

Код редукции глобальной суммы

Цикл OpenMP с прагмой parallel for с выражением reduction

Операция редукции вычисляет сумму, локальную для каждого потока, а затем суммирует все потоки. Редукционная переменная `sum` инициализируется надлежащим для операции значением. В приведенном в листинге 7.11 исходном коде редукционная переменная инициализируется нулем. Инициализация переменной `sum` нулем в строке 3 по-прежнему необходима для надлежащей работы, когда мы не используем OpenMP.

7.3.6 Потенциальные трудности OpenMP уровня цикла

OpenMP уровня цикла может применяться к большинству, но не ко всем циклам. Для того чтобы компилятор OpenMP мог применить операцию совместного использования работы, цикл должен иметь каноническую форму. Каноническая форма – это традиционная, простейшая имплементация цикла, с которой программисты знакомятся с самого начала. Требования к ней таковы:

- переменная индекса цикла должна быть целочисленной;
- индекс цикла нельзя модифицировать в цикле;
- цикл должен иметь стандартные условия выхода;
- итерации цикла должны быть счетными;
- цикл не должен иметь никаких несомых циклом зависимостей.

Последнее требование можно проверить, повернув порядок цикла в обратную сторону либо изменив порядок операций цикла. Если ответ изменится, то цикл будет иметь несомые циклом зависимости. Существуют аналогичные ограничения на несомые циклом зависимости для оптимизации на CPU и имплементации потокообразования на GPU. Сходства этого требования к несомым циклом зависимостям описываются как антитеза «мелкозернистая параллелизация – крупнозернистая структура», используемая в подходе на основе распределенной памяти и передачи сообщений. Вот несколько определений:

- *мелкозернистая параллелизация* – вид распараллеливания, при котором вычислительные циклы или другие малые блоки кода обрабатываются несколькими процессорами или потоками и могут нуждаться в частой синхронизации;
- *крупнозернистая параллелизация* – вид распараллеливания, при котором процессор работает с крупными блоками кода с нечастой синхронизацией.

Многие языки программирования предложили модифицированный тип цикла, который сообщает компилятору, что параллелизм уровня цикла допускается применять в той или иной форме. А пока эту информация привносится передачей прагмы или директивы перед циклом.

7.4 Важность области видимости переменной для правильности в OpenMP

В целях конвертирования приложения или подпрограммы в OpenMP высокого уровня вам нужно разбираться в области видимости переменных. Спецификации OpenMP расплывчаты во многих деталях организации области видимости. На рис. 7.7 показаны правила определения области видимости для компиляторов. В общем случае переменная в стеке считается приватной, тогда как переменные, которые помещены в кучу, являются совместными (рис. 7.2). Для OpenMP высокого уровня наиболее важным является вопрос о том, как управлять областью видимости в вызываемой подпрограмме в параллельном участке.

Правила организации области видимости

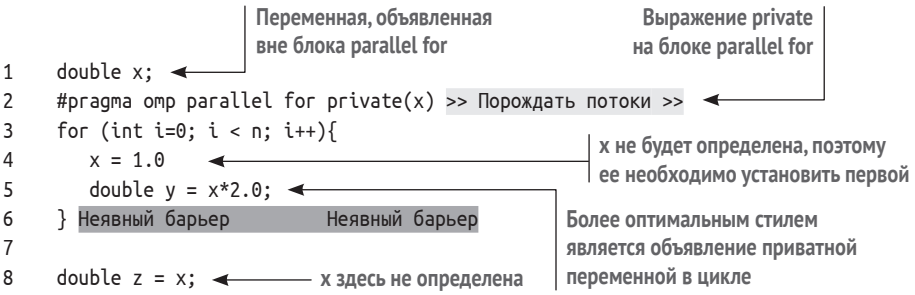
<table>
<tr><th colspan="2">Спецификация</th><th>Совместные</th><th>Приватные</th><th>Редукция</th></tr>
<tr><td colspan="2">Параллельный конструкт</td><td>Переменные объявляются вне параллельного конструкта либо в выражении shared</td><td>Автоматические переменные внутри параллельного конструкта либо в выражении private или firstprivate</td><td rowspan="4">Редукционное выражение</td></tr>
<tr><td rowspan="3">Параллельный участок</td><td>Процедура на Fortran</td><td>Атрибут save, инициализированные переменные, общепринятые блоки, модульные переменные</td><td rowspan="2">Все локальные переменные либо объявленные threadprivate. Также выделенные динамически</td></tr>
<tr><td>Процедура на C</td><td>Переменные файловой области, extern либо static</td></tr>
<tr><td>Аргументы</td><td colspan="2">Наследуются за счет вызова среды</td></tr>
<tr><td colspan="2">Всегда</td><td colspan="3">Все индексы циклов будут приватными</td></tr>
</table>

Рис. 7.7 Краткое описание правил организации поточной области видимости для приложений OpenMP

При определении области видимости переменных следует уделять больше внимания переменным в левой части выражения. На первом месте по важности стоит правильное определение области видимости переменных, в которые осуществляется запись. Обратите внимание, что приватные переменные не определяются при входе и после выхода из параллельного участка, как показано в листинге 7.12. В особых случаях выражения `firstprivate` и `lastprivate` могут это поведение модифицировать. Если переменная является приватной, то мы должны видеть,

что она устанавливается до того, как она будет использоваться в параллельном блоке, и не будет использоваться после параллельного участка. Если переменная предназначена быть приватной, то лучше всего объявлять переменную в цикле, потому что локально объявленная переменная ведет себя точно так же, как приватная переменная OpenMP. Не вдаваясь в подробности, объявление переменной в цикле устраняет любую путаницу в том, каким должно быть поведение. Она не существует ни до цикла, ни после него, поэтому неправильное использование невозможно.

Листинг 7.12 Приватная переменная, входящая в параллельный блок OpenMP



На директиве в строке 4 в листинге 7.11 мы добавили редукционное выражение, чтобы отметить особую трактовку, необходимую для переменной суммы. В строке 2 листинга 7.12 мы показали директиву private. В директиве parallel и других программных блоках можно использовать и другие выражения, например:

- `shared(var, var);`
- `private(var, var);`
- `firstprivate(var, var);`
- `lastprivate(var, var);`
- `reduction([+,min,max]:<var,var>);`
- `*threadprivate` (специальная директива, используемая в поточно-параллельной функции).

Мы настоятельно рекомендуем использовать такие инструменты, как Intel® Inspector и Allinea/ARM MAP, служащие для разработки более эффективного кода и имплементирования OpenMP высокого уровня. Мы обсудим некоторые из этих инструментов в разделе 7.9. Перед началом имплементирования OpenMP высокого уровня необходимо ознакомиться с рядом существенных инструментов. После выполнения приложения посредством этих инструментов можно получить более глубокое понимание приложения, которое позволит плавнее перейти к имплементации OpenMP высокого уровня.

7.5 OpenMP уровня функции: придание всей функции целиком свойства поточной параллельности

Мы представим концепцию OpenMP высокого уровня в разделе 7.6. Но, прежде чем пытаться использовать OpenMP высокого уровня, необходимо посмотреть, как расширять имплементации уровня цикла для охвата более крупных разделов кода. Цель расширения имплементации уровня цикла состоит в снижении накладных расходов и повышении параллельной эффективности. При расширении параллельного участка он в конечном итоге охватывает всю подпрограмму целиком. Конвертировав всю функцию целиком в параллельный участок OpenMP, OpenMP будет обеспечивать гораздо меньший контроль над поточной областью видимости переменных. Выражения для параллельного участка больше не помогают, так как нет места для добавления выражений, которые задают область видимости. Итак, как контролировать область видимости переменной?

Хотя принятые по умолчанию варианты области видимости переменных в функциях обычно работают хорошо, бывают случаи, когда они не справляются. Единственным элементом OpenMP управления при помощи прагм для функций является директива `threadprivate`, которая делает объявленную переменную приватной. Большинство переменных в функции находятся в стеке и уже являются приватными. Если массив в подпрограмме выделен динамически, то указатель, которому он назначен, является локальной переменной в стеке, что означает, что он является приватным и разным для каждого потока. Мы хотим, чтобы этот массив был совместным, но для этого нет директивы. Используя конкретные компиляторные правила организации области видимости, приведенные на рис. 7.7, мы добавляем в объявление указателя в Fortran атрибут `save`, заставляя компилятор помещать переменную в кучу и, следовательно, использовать переменную совместно между потоками. В C переменная может быть объявлена статической либо сделана с файловой областью видимости. В следующем ниже листинге приведено несколько примеров поточной области видимости переменных для Fortran, а в листинге 7.14 приведены примеры для C и C++.

Листинг 7.13 Область видимости уровня функции в Fortran для переменной

```
4 subroutine function_level_OpenMP(n, y)
5    integer :: n
6    real :: y(n)
7
8    real, allocatable :: x(:)
9    real x1
```

Указатель на массив y и его элементы, которые являются приватными

Указатель на выделяемый массив x, который является приватным

Переменная x1 находится в стеке, поэтому она – приватная

```
10    real :: x2 = 0.0
11    real, save :: x3
12    real, save, allocatable :: z
13
14    if (thread_id .eq. 0) allocate(x(100))
15
16 ! много кода
17
18 if (thread_id .eq. 0) deallocate(x)
19 end subroutine function_level_OpenMP
```

Переменная x2 является в Fortran 90 совместной

Переменная x3 помещается в кучу, поэтому она является совместной

Указатель на массив z находится в куче и является совместным

Память под массив x является совместной, но указатель на x является приватным

Указатель на массив y в строке 6 является областью переменной в местоположении подпрограммы. В данном случае он находится в параллельном участке, что делает его приватным. И указатель на x, и переменная x1 являются приватными. Область переменной x2 в строке 10 – более сложная. Она является совместной в Fortran 90 и приватной в Fortran 77. Инициализированные переменные в Fortran 90 находятся в куче и инициализируются (в данном случае нулем) только при их первом появлении! Переменные x3 и z в строках 11 и 12 являются совместными, потому что они находятся в куче. Память, выделенная под x в строке 14, находится в куче и является совместной, но указатель является приватным, что приводит к тому, что память доступна только в нулевом потоке.

Листинг 7.14 Область видимости уровня функции в C/C++ для переменных

```
 5 void function_level_OpenMP(int n, double *y)
 6 {
 7    double *x;
 8    static double *x1;
 9
10    int thread_id;
11 #pragma omp parallel
12    thread_id = omp_get_thread_num();
13
14    if (thread_id == 0) x = (double *)malloc(100*sizeof(double));
15    if (thread_id == 0) x1 = (double *)malloc(100*sizeof(double));
16
17 // много кода
18    if (thread_id ==0) free(x);
19    if (thread_id ==0) free(x1);
20 }
```

Указатель на массив y является приватным

Указатель на массив x является приватным

Указатель на массив x1 является совместным

Память под массив x1 является совместной

Память под массив x является совместной

Указатель на массив y в списке аргументов в строке 5 находится в стеке. Он имеет область переменной в вызывающем местоположении. В параллельном участке указатель на y является приватным. Память под массив x находится в куче и является совместной, но указатель является приватным, поэтому память доступна только из нулевого потока. Память под массив x1 находится в куче и используется совместно, а указатель является совместным, поэтому память доступна и используется совместно всеми потоками.

Вам нужно всегда быть начеку на случай непредвиденных последствий объявлений и определений переменных, которые влияют на поточную область видимости. Например, инициализация локальной переменной значением в подпрограмме Fortran 90 автоматически закрепляет за переменной атрибут `save`, и переменная теперь используется совместно[1]. Во избежание каких-либо проблем или путаницы мы рекомендуем добавлять атрибут save в объявление в явной форме.

7.6 Усовершенствование параллельной масштабируемости с помощью OpenMP высокого уровня

Зачем использовать OpenMP высокого уровня? Центральная стратегия OpenMP высокого уровня заключается в улучшении стандартного параллелизма уровня цикла за счет минимизирования накладных расходов на ветвление/соединение и задержки памяти. Сокращение времен ожидания потока часто рассматривается как еще один важный мотивирующий фактор имплементаций OpenMP высокого уровня. За счет деления работы между потоками в явной форме потоки больше не будут неявно ожидать других потоков и поэтому могут переходить к следующей части вычисления. Это позволяет контролировать точку синхронизации в явной форме. На рис. 7.8, в отличие от типичной модели ветвления/соединения с помощью стандартной OpenMP, высокоуровневая OpenMP держит потоки в дремлющем состоянии, но живыми, что значительно снижает накладные расходы.

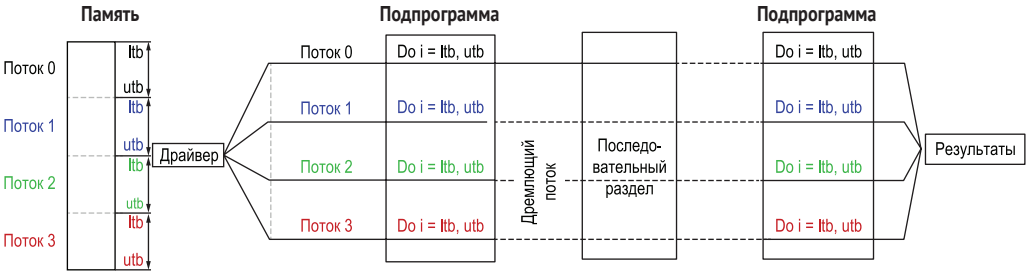


Рис. 7.8 Визуализация потокообразования OpenMP высокого уровня. Потоки порождаются один раз и остаются дремлющими, когда в этом нет необходимости. Границы потоков задаются вручную, и синхронизация сводится к минимуму

[1] Это не относится к стандарту Fortran 77! Но даже в Fortran 77 некоторые компиляторы, такие как компилятор DEC Fortran, требуют, чтобы каждая переменная в подпрограмме имела атрибут save, что вызывает неясные ошибки и проблемы с переносимостью. Зная это, мы могли бы обеспечивать компилирование в стандарте Fortran 90 и потенциально устранять проблему приватной области видимости путем инициализирования указателя на массив, что приводит к его переносу в кучу, делая переменную совместной.

В этом разделе мы проведем обзор конкретных шагов, необходимых для имплементирования OpenMP высокого уровня. Затем мы покажем вам, как переходить от имплементации уровня цикла к имплементации высокого уровня.

7.6.1 Как имплементировать OpenMP высокого уровня

Имплементация OpenMP высокого уровня нередко занимает больше времени, поскольку требует использования передовых инструментов и тщательного тестирования. Имплементировать OpenMP высокого уровня к тому же бывает сложно, поскольку такая имплементация более подвержена гоночным состояниям, чем стандартная имплементация уровня цикла. Кроме того, зачастую неясно, как переходить от начальной точки (имплементации уровня цикла) к конечной (имплементации высокого уровня).

Более утомительная открытая имплементация высокого уровня широко используется, когда есть желание повысить эффективность и избавиться от затрат на порождение потоков и синхронизацию. Дополнительная информация о OpenMP высокого уровня находится в разделе 7.11. Имплементирование эффективной открытой мультиобработки (OpenMP) высокого уровня возможно, если хорошо понимать границы памяти, связанные со всеми циклами в вашем приложении, задействуя инструменты профилирования и методично прорабатывая следующие ниже шаги. Мы предлагаем и показываем имплементационную стратегию, являющуюся поступательной, методичной и обеспечивающую успешный, плавный переход к имплементации OpenMP высокого уровня. Шаги к имплементации OpenMP высокого уровня таковы:

- *базовая имплементация* – имплементация OpenMP уровня цикла;
- *шаг 1: редуцировать поточный запуск* – объединить параллельные участки и свести все параллельные конструкты уровня цикла к более крупным параллельным участкам;
- *шаг 2: синхронизация* – добавить выражения nowait в циклы for, где синхронизация не требуется, и рассчитать и вручную подразделить циклы по потокам, что позволит устранить барьеры и необходимую синхронизацию;
- *шаг 3: оптимизация* – по возможности сделать массивы и переменные приватными для каждого потока;
- *шаг 4: правильность кода* – выполнить тщательную проверку на наличие гоночных состояний (после каждого шага).

На рис. 7.9 и 7.10 показан псевдокод, соответствующий приведенным выше четырем шагам, начиная с типичной имплементации уровня цикла с использованием прагм `omp parallel do` и переходя к более эффективному высокоуровневому параллелизму.

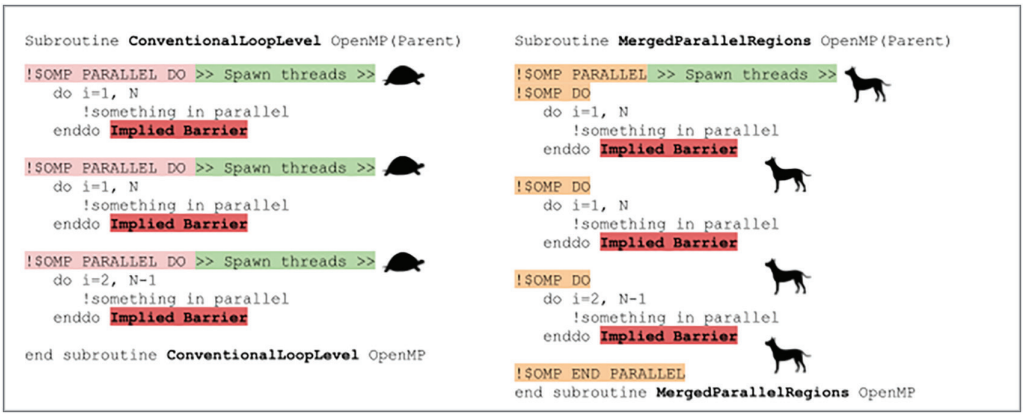


Рис. 7.9 OpenMP высокого уровня начинается с имплементации OpenMP уровня цикла и объединяет параллельные участки, чтобы снизить затраты на порождение потоков. Мы используем изображения животных, чтобы показать места, где вносятся изменения, и относительную скорость фактической имплементации. Обычная OpenMP уровня цикла, показанная черепахой, работает быстрее, но имеются накладные расходы в каждой прагме `parallel do`, которые лимитируют ускорение. Собака показывает относительный прирост скорости от объединения параллельных участков

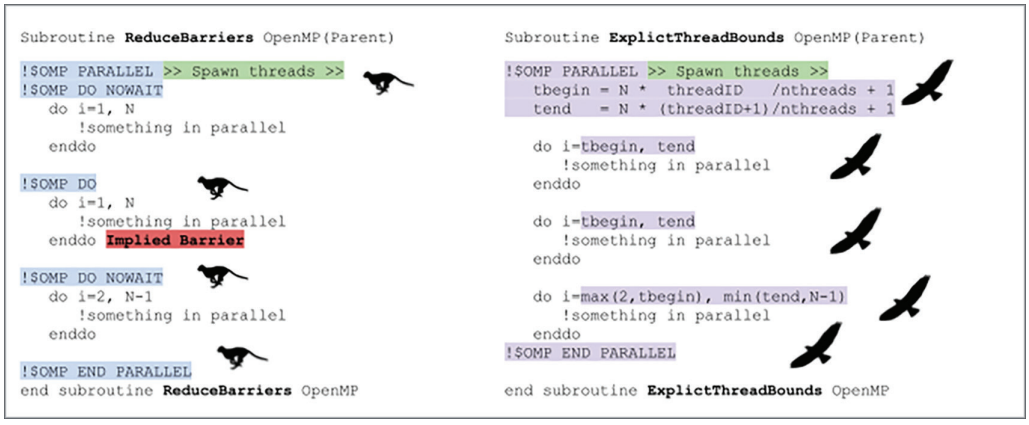


Рис. 7.10 Следующие шаги OpenMP высокого уровня добавляют в циклы `do` и `for` выражения `nowait`, которые снижают затраты на синхронизацию. Затем мы сами вычисляем границы циклов и используем их в циклах в явной форме, чтобы избежать еще большей синхронизации. Здесь гепард и ястреб идентифицируют изменения, вносимые в обе имплементации. Ястреб (справа) быстрее гепарда (слева), так как накладные расходы OpenMP уменьшаются

В наших шагах к имплементации OpenMP высокого уровня время запуска потока сокращается на первом шаге OpenMP выского уровня. Весь код размещается в одном параллельном участке, чтобы минимизировать накладные расходы на ветвление и соединение. В OpenMP высоко-

го уровня потоки генерируются директивой `parallel` один раз, в начале исполнения программы. Неиспользуемые потоки не умирают, а остаются в состоянии дремы при прохождении последовательной части. Для того чтобы это прогарантировать, последовательная часть исполняется главным потоком, что позволяет практически не вносить изменений в последовательную часть кода. Как только программа завершает прохождение последовательной части или снова запускает параллельный участок, вызываются или используются повторно те же потоки, которые были разветвлены в начале программы.

Шаг 2 касается синхронизации, по умолчанию добавленной в каждый цикл `for` в OpenMP. Самое простое, что можно делать для снижения затрат на синхронизацию, состоит в добавлении выражения `nowait` во все циклы, где это возможно, сохраняя при этом правильность. Следующим шагом является разделение работы между потоками в явной форме. Типичный код для явного разделения работы для C показан ниже. (Эквивалент для Fortran, учитывающий индексацию массивов начиная с 1, показан на рис. 7.10.)

```
tbegin = N * threadID /nthreads
tend = N * (threadID+1)/nthreads
```

Влияние ручного разделения массивов состоит в сокращении им перетирания кеша и гоночных состояний, не позволяя потокам совместно использовать одно и то же пространство в памяти.

Шаг 3, оптимизация, означает, что мы в явной форме задаем, что некоторые переменные являются совместными либо приватными. Предоставляя потокам конкретное пространство в памяти, компилятор (и программист) может отказаться от догадок в отношении состояния переменных. Это делается путем применения правил организации области видимости переменных, приведенных на рис. 7.7. Кроме того, компиляторы не могут правильно выполнять параллелизацию циклов, содержащих в себе сложные несомые циклом зависимости и циклы, которые не находятся в канонической форме. OpenMP высокого уровня помогает компилятору, четче описывая поточную область видимости переменных, тем самым позволяя выполнять параллелизацию сложных циклов. Это приводит в последнюю часть данного шага в рамках подхода на основе OpenMP высокого уровня. Массивы будут разделены по потокам. Разделение массивов в явной форме гарантирует, что поток прикасается только к назначенной ему памяти, и позволяет нам начать устранять проблемы с локальностью памяти.

И на последнем шаге, шаге обеспечения правильности кода, важно применить перечисленные в разделе 7.9 инструменты обнаружения и устранения гоночных состояний. В следующем далее разделе мы покажем вам процесс имплементирования описанных нами шагов. Программы в репозитории исходного кода на GitHub этой главы окажутся полезными при выполнении указанного пошагового процесса.

7.6.2 Пример имплементирования OpenMP высокого уровня

Полная имплементация OpenMP высокого уровня может быть завершена в несколько шагов. Сначала в дополнение к отысканию наиболее трудоемкого цикла в коде вы должны посмотреть, где в вашем приложении находится узкое место(а) кода. Затем вы сможете найти цикл кода на самом внутреннем уровне и добавить стандартные циклические директивы OpenMP. Необходимо понять область видимости переменных в наиболее интенсивных циклах и внутренних циклах, обратившись к рис. 7.7 для руководства.

На шаге 1 вы должны сосредоточиться на сокращении затрат на запуск потоков. Это делается в листинге 7.15 путем объединения параллельных участков, чтобы включить весь итерационный цикл целиком в один параллельный участок. Мы начинаем медленно перемещать директивы OpenMP наружу, расширяя параллельный участок. Изначальные прагмы OpenMP в строках 49 и 57 можно объединить в один параллельный участок между строками 44–70. Протяженность параллельного участка определяется фигурными скобками в строках 45 и 70, как следствие, параллельный участок начинается только один раз вместо 10 000 раз.

Листинг 7.15 Объединение параллельных участков в один

HighLevelOpenMP_stencil/stencil_opt4.c

```
44 #pragma omp parallel >> Порождать потоки >>
45 {
46    int thread_id = omp_get_thread_num();
47    for (int iter = 0; iter < 10000; iter++){
48       if (thread_id ==0) cpu_timer_start(&tstart_flush);
49       #pragma omp for nowait
50       for (int l = 1; l < jmax*imax*4; l++){
51          flush[l] = 1.0;
52       }
53       if (thread_id == 0){
54          flush_time += cpu_timer_stop(tstart_flush);
55          cpu_timer_start(&tstart_stencil);
56       }
57       #pragma omp for >> Порождать потоки >>
58       for (int j = 1; j < jmax-1; j++){
59          for (int i = 1; i < imax-1; i++){
60             xnew[j][i]=(x[j][i] + x[j][i-1] + x[j][i+1] + x[j-1][i] +
     x[j+1][i])/5.0;
61          }
62       } Неявный барьер          Неявный барьер
63       if (thread_id == 0){
64          stencil_time += cpu_timer_stop(tstart_stencil);
65
66          SWAP_PTR(xnew, x, xtmp);
67          if (iter%1000 == 0) printf("Итер %d\n",iter);
68       }
```

Один параллельный участок OpenMP

Прагма OpenMP for без барьера синхронизации в конце цикла

Вторая прагма OpenMP for

```
69    }
70 } // конец omp parallel
   Неявный барьер          Неявный барьер
```

Порции кода, которые должны выполняться последовательно, ставятся под управление главного потока, что позволяет расширять параллельный участок на крупные порции кода, охватывающие как последовательный, так и параллельный участки. На каждом шаге используйте инструменты, описанные в разделе 7.9, с целью обеспечения неотступно правильной работы приложения.

Во второй части имплементации вы начинаете транзит к OpenMP высокого уровня, перенося главный параллельный цикл OpenMP в начало программы. После этого вы можете перейти к вычислению верхней и нижней границ цикла. В листинге 7.16 (и в онлайновых примерах в stencil_opt5.c и stencil_opt6.c) показано, как вычислять характерные для параллельного участка верхнюю и нижнюю границы. Следует помнить, что массивы начинаются в разных точках в зависимости от языка: Fortran начинается с 1, а C начинается с 0. В циклах с одинаковой верхней и нижней границей может использоваться один и тот же поток без необходимости пересчета границ.

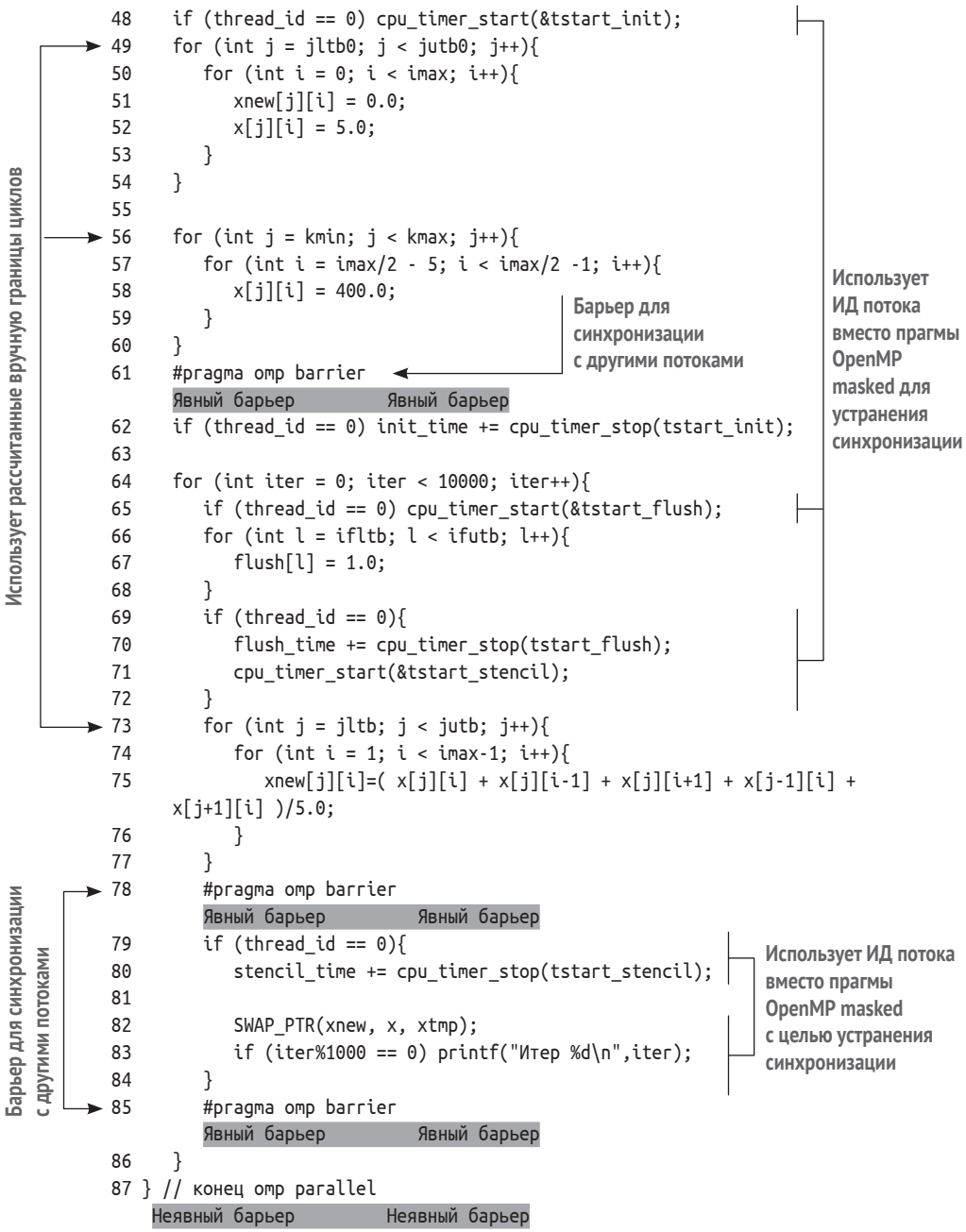
ПРИМЕЧАНИЕ Не следует забывать устанавливать барьеры в необходимых местах, чтобы предотвращать гоночные состояния. Также необходимо проявлять большую осторожность при размещении этих прагм, так как слишком большое их число может негативно сказаться на совокупной производительности приложения.

Листинг 7.16 Предварительный расчет нижней и верхней границ цикла

HighLevelOpenMP_stencil/stencil_opt6.c

```
29 #pragma omp parallel >> Порождать потоки >>
30 {
31    int thread_id = omp_get_thread_num();
32    int nthreads = omp_get_num_threads();
33
34    int jltb = 1 + (jmax-2) * ( thread_id ) / nthreads;
35    int jutb = 1 + (jmax-2) * ( thread_id + 1 ) / nthreads;
36
37    int ifltb = (jmax*imax*4) * ( thread_id ) / nthreads;
38    int ifutb = (jmax*imax*4) * ( thread_id + 1 ) / nthreads;
39
40    int jltb0 = jltb;
41    if (thread_id == 0) jltb0--;
42    int jutb0 = jutb;
43    if (thread_id == nthreads-1) jutb0++;
44
45    int kmin = MAX(jmax/2-5,jltb);
46    int kmax = MIN(jmax/2+5,jutb);
47
```

Вычисляет границы циклов




```
48    if (thread_id == 0) cpu_timer_start(&tstart_init);
49    for (int j = jltb0; j < jutb0; j++){
50       for (int i = 0; i < imax; i++){
51          xnew[j][i] = 0.0;
52          x[j][i] = 5.0;
53       }
54    }
55
56    for (int j = kmin; j < kmax; j++){
57       for (int i = imax/2 - 5; i < imax/2 -1; i++){
58          x[j][i] = 400.0;
59       }
60    }
61    #pragma omp barrier
      Явный барьер          Явный барьер
62    if (thread_id == 0) init_time += cpu_timer_stop(tstart_init);
63
64    for (int iter = 0; iter < 10000; iter++){
65       if (thread_id == 0) cpu_timer_start(&tstart_flush);
66       for (int l = ifltb; l < ifutb; l++){
67          flush[l] = 1.0;
68       }
69       if (thread_id == 0){
70          flush_time += cpu_timer_stop(tstart_flush);
71          cpu_timer_start(&tstart_stencil);
72       }
73       for (int j = jltb; j < jutb; j++){
74          for (int i = 1; i < imax-1; i++){
75             xnew[j][i]=( x[j][i] + x[j][i-1] + x[j][i+1] + x[j-1][i] +
      x[j+1][i] )/5.0;
76          }
77       }
78       #pragma omp barrier
         Явный барьер          Явный барьер
79       if (thread_id == 0){
80          stencil_time += cpu_timer_stop(tstart_stencil);
81
82          SWAP_PTR(xnew, x, xtmp);
83          if (iter%1000 == 0) printf("Итер %d\n",iter);
84       }
85       #pragma omp barrier
         Явный барьер          Явный барьер
86    }
87 } // конец omp parallel
    Неявный барьер          Неявный барьер
```

В целях получения правильного ответа крайне важно начать с самого внутреннего цикла и понимать, какие переменные должны оставаться приватными или становиться совместными для потоков. Когда вы начнете увеличивать параллельный участок, последовательные порции

кода будут помещаться в маскированный участок. В этом участке имеется один поток, и он выполняет всю работу, в то время как другие потоки остаются живыми, но дремлющими. При размещении последовательных частей кода в главный поток требуется ноль или всего несколько изменений. После того как программа завершает работу в последовательном участке или попадает в параллельный участок, прошлые дремлющие потоки снова начинают работать, параллелизуя текущий цикл.

На заключительном шаге, сравнивая результаты для шагов на пути к имплементации OpenMP высокого уровня, в листингах 7.14 и 7.15, а также в приведенных онлайновых стенсильных примерах вы увидите, что число прагм значительно сокращено, а производительность при этом стала выше (рис. 7.11).

```
Stencil_opt2    Timing: init 0.003746 flush 3.495596 stencil 3.306887 total 6.808650
Stencil_opt3    Timing: init 0.003081 flush 3.158420 stencil 3.568470 total 6.735474
Stencil_opt4    Timing: init 0.002930 flush 2.853069 stencil 3.491407 total 6.355609
Stencil_opt5    Timing: init 0.002973 flush 3.077176 stencil 3.140370 total 6.227241
Stencil_opt6    Timing: init 0.002831 flush 2.947900 stencil 3.186743 total 6.255831
```

Рис. 7.11 Оптимизирование прагм OpenMP одновременно сокращает число требуемых прагм и повышает производительность стенсильного ядра

7.7 Гибридное потокообразование и векторизация с OpenMP

В этом разделе мы скомбинируем темы из главы 6 с темами, с которыми мы познакомились в этой главе. Эта комбинация обеспечивает более качественную параллелизацию и задействует векторный процессор. Поточный цикл OpenMP можно скомбинировать с векторизованным циклом путем добавления выражения simd в `parallel for`, как в `#pragma omp parallel for simd`. В следующем ниже листинге это показано для потоковой триады.

Листинг 7.17 Потокообразование OpenMP уровня цикла и векторизация потоковой триады

StreamTriad/stream_triad_opt3.c

```
 1 #include <stdio.h>
 2 #include <time.h>
 3 #include <omp.h>
 4 #include "timer.h"
 5
 6 #define NTIMES 16
 7 #define STREAM_ARRAY_SIZE 80000000
```

Достаточно велик, чтобы втиснуть в основную память

```
 8 static double a[STREAM_ARRAY_SIZE], b[STREAM_ARRAY_SIZE],
                c[STREAM_ARRAY_SIZE];
 9
10 int main(int argc, char *argv[]){
11    #pragma omp parallel >> Порождать потоки >>
12       if (omp_get_thread_num() == 0)
13          printf("Выполняется с %d потоком(ами)\n",omp_get_num_threads());
      Неявный барьер          Неявный барьер
14
15    struct timeval tstart;
16    double scalar = 3.0, time_sum = 0.0;
17    #pragma omp parallel for simd >> Порождать потоки >>
18    for (int i=0; i<STREAM_ARRAY_SIZE; i++) {
19       a[i] = 1.0;
20       b[i] = 2.0;
21    }
      Неявный барьер          Неявный барьер
22    for (int k=0; k<NTIMES; k++){
23       cpu_timer_start(&tstart);
24       #pragma omp parallel for simd >> Порождать потоки >>
25       for (int i=0; i<STREAM_ARRAY_SIZE; i++){
26          c[i] = a[i] + scalar*b[i];
27       }
         Неявный барьер          Неявный барьер
28       time_sum += cpu_timer_stop(tstart);
29       c[1]=c[2];
30    }
31
32    printf("Среднее время выполнения составляет %lf msecs\n", time_sum/NTIMES);
33}
```

Инициализирует данные и массивы

Цикл потоковой триады

Не дает компилятору оптимизировать цикл

Гибридная имплементация стенсильного примера с потокообразованием и векторизацией помещает прагму `for` во внешний цикл, а прагму `simd` во внутренний, как показано в следующем ниже листинге. Как поточные, так и векторизованные циклы лучше всего работают с циклами над крупными массивами, как это обычно бывает в стенсильном примере.

Листинг 7.18 Трафаретный пример с потокообразованием и векторизацией

HybridOpenMP_stencil/stencil_hybrid.c

```
26 #pragma omp parallel >> Порождать потоки >>
27 {
28    int thread_id = omp_get_thread_num();
29    if (thread_id == 0) cpu_timer_start(&tstart_init);
30    #pragma omp for
31    for (int j = 0; j < jmax; j++){
32       #ifdef OMP_SIMD
```

```
      #pragma omp simd
      #endif
      for (int i = 0; i < imax; i++){
         xnew[j][i] = 0.0;
         x[j][i] = 5.0;
      }
   } Неявный барьер          Неявный барьер

   #pragma omp for
   for (int j = jmax/2 - 5; j < jmax/2 + 5; j++){
      for (int i = imax/2 - 5; i < imax/2 -1; i++){
         x[j][i] = 400.0;
      }
   } Неявный барьер          Неявный барьер
   if (thread_id == 0) init_time += cpu_timer_stop(tstart_init);

   for (int iter = 0; iter < 10000; iter++){
      if (thread_id ==0) cpu_timer_start(&tstart_flush);
      #ifdef OMP_SIMD
      #pragma omp for simd nowait
      #else
      #pragma omp for nowait
      #endif
      for (int l = 1; l < jmax*imax*10; l++){
         flush[l] = 1.0;
      }
      if (thread_id == 0){
         flush_time += cpu_timer_stop(tstart_flush);
         cpu_timer_start(&tstart_stencil);
      }
      #pragma omp for
      for (int j = 1; j < jmax-1; j++){
         #ifdef OMP_SIMD
         #pragma omp simd
         #endif
         for (int i = 1; i < imax-1; i++){
            xnew[j][i]=(x[j][i] + x[j][i-1] + x[j][i+1] +
                                x[j-1][i] + x[j+1][i])/5.0;
         }
      } Неявный барьер          Неявный барьер
      if (thread_id == 0){
         stencil_time += cpu_timer_stop(tstart_stencil);

         SWAP_PTR(xnew, x, xtmp);
         if (iter%1000 == 0) printf("Iter %d\n",iter);
      }
      #pragma omp barrier
   }
} // конец omp parallel
Неявный барьер          Неявный барьер
```

Добавляет прагму OpenMP SIMD для внутренних циклов

Добавляет добавочную прагму OpenMP SIMD в прагму for в одном цикле

Сравнение результатов компилятора GCC с векторизацией и без нее показывают значительное ускорение с векторизацией:

```
4 threads, GCC 8.2 compiler, Skylake Gold 6152
Threads only: Timing init 0.006630 flush 17.110755 stencil 17.374676
     total 34.499799
Threads & vectors: Timing init 0.004374 flush 17.498293 stencil 13.943251
     total 31.454906
```

7.8 Продвинутые примеры использования OpenMP

Показанные до этого примеры были простыми циклами над набором данных с относительно малым числом осложнений. В этом разделе мы покажем вам, как справляться с тремя продвинутыми примерами, требующими больше усилий.

- *Двухшаговый стенсиль с разводом направлений* – продвинутое манипулирование поточной областью видимости переменных.
- *Суммирование по Кахану* – более сложный редукционный цикл.
- *Префиксный скан* – манипулирование подразделением работы среди потоков.

Примеры в этом разделе раскрывают различные способы решения более сложных ситуаций и дают вам более глубокое понимание OpenMP.

7.8.1 Стенсильный пример с отдельным проходом для направлений x и y

Здесь мы рассмотрим потенциальные трудности, возникающие при имплементировании OpenMP для двухшагового стенсильного оператора с разводом направлений (split-direction), в котором для каждого пространственного направления выполняется отдельный проход. *Стенсили* (или *трафареты*) являются строительными блоками для численных научных приложений и используются для расчета динамических решений уравнений в частных производных.

В двухшаговом стенсиле, где значения вычисляются на гранях, массивы данных имеют разные требования к совместному использованию данных. На рис. 7.12 показан такой стенсиль с двухмерными массивами граней. Кроме того, часто бывает, что одна из размерностей этих двухмерных массивов должна использоваться коллективно всеми потоками или процессами. Манипулировать данными *x*-граней проще, потому что они согласуются с декомпозицией поточных данных, но нам не нужен полный массив *x*-граней в каждом потоке. Данные *y*-граней имеют другую проблему, поскольку данные пронизывают все потоки, что требует совместного использования двухмерного массива *y*-граней. OpenMP вы-

сокого уровня позволяет быстро приватизировать необходимую размерность. На рис. 7.12 показано, как некоторые размерности матрицы могут делаться приватными, совместными либо обоими.

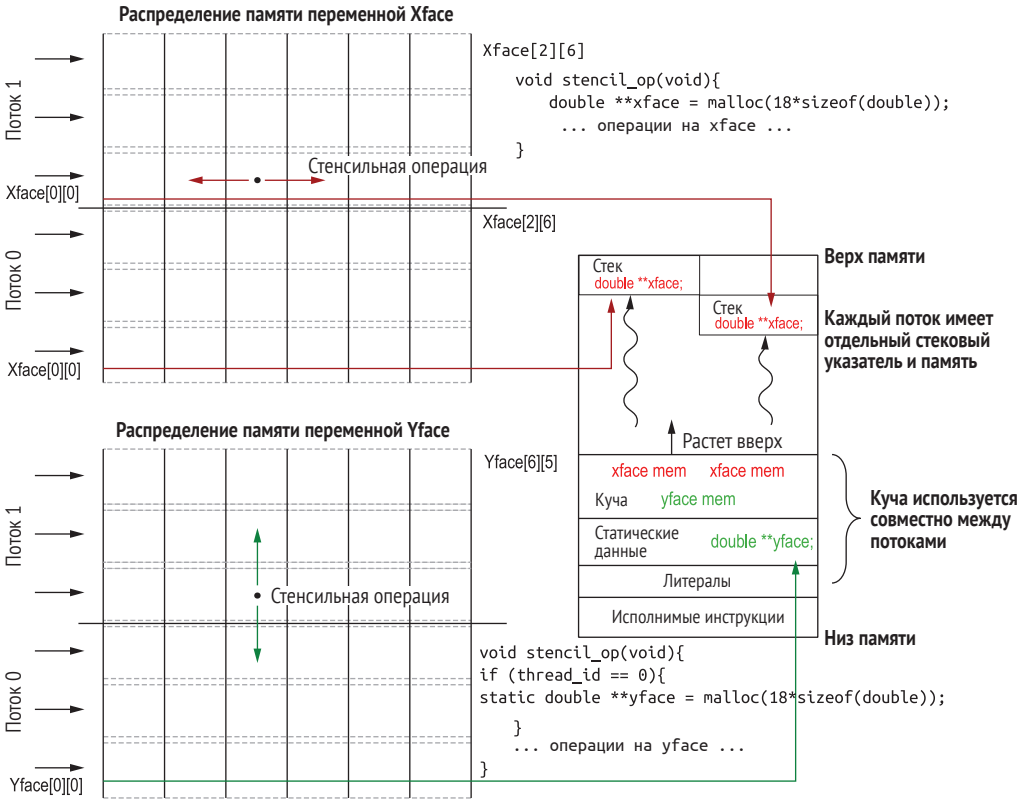


Рис. 7.12 Выровненная по потокам *x*-грань стенсиля нуждается в отдельном хранилище для каждого потока. Указатель должен находиться в стеке, и у каждого потока должен быть свой указатель. Грань *y* должна использовать данные совместно, поэтому мы определяем один указатель в участке статических данных, где к нему могут обращаться оба потока

Присущий большинству ядер принцип первого касания (определенный в разделе 7.1.1) гласит, что память, скорее всего, будет локальной для потока (за исключением краев между потоками на границах страниц). Мы можем улучшить локальность памяти, сделав разделы массива полностью приватными для потока, где это возможно, например данные *x*-граней. В связи с увеличением числа процессоров увеличение локализации данных имеет крайне важное значение для сведения к минимуму увеличивающегося разрыва в скорости между процессорами и памятью. В следующем ниже листинге для начала показана последовательная имплементация.

Листинг 7.19 Стенсильный оператор с разводом направлений

SplitStencil/SplitStencil.c

```
58 void SplitStencil(double **a, int imax, int jmax)
59 {
60    double** xface = malloc2D(jmax, imax);
61    double** yface = malloc2D(jmax, imax);
62    for (int j = 1; j < jmax-1; j++){
63       for (int i = 0; i < imax-1; i++){
64          xface[j][i] = (a[j][i+1]+a[j][i])/2.0;
65       }
66    }
67    for (int j = 0; j < jmax-1; j++){
68       for (int i = 1; i < imax-1; i++){
69          yface[j][i] = (a[j+1][i]+a[j][i])/2.0;
70       }
71    }
72    for (int j = 1; j < jmax-1; j++){
73       for (int i = 1; i < imax-1; i++){
74          a[j][i] = (a[j][i]+xface[j][i]+xface[j][i-1]+
75                               yface[j][i]+yface[j-1][i])/5.0;
76       }
77    }
78    free(xface);
79    free(yface);
80 }
```

Вычисляет значения на гранях *x* и *y* ячеек

Для вычисления *x*-грани требуются только смежные ячейки в направлении *x*

Для вычисления грани *y* требуются смежные ячейки в направлении *y*

Вносит информацию со всех граней ячейки

При использовании OpenMP со стенсильным оператором необходимо определить, какой – приватной либо совместной – должна быть память для каждого потока. В (приведенном выше) листинге 7.18 память для направления *x* может быть полностью приватной, что позволяет выполнять более быстрые вычисления. В направлении *y* (рис. 7.12) стенсиль требует доступа к данным смежного потока; отсюда эти данные должны использоваться совместно между потоками. Это приводит нас к имплементации, показанной в следующем ниже листинге.

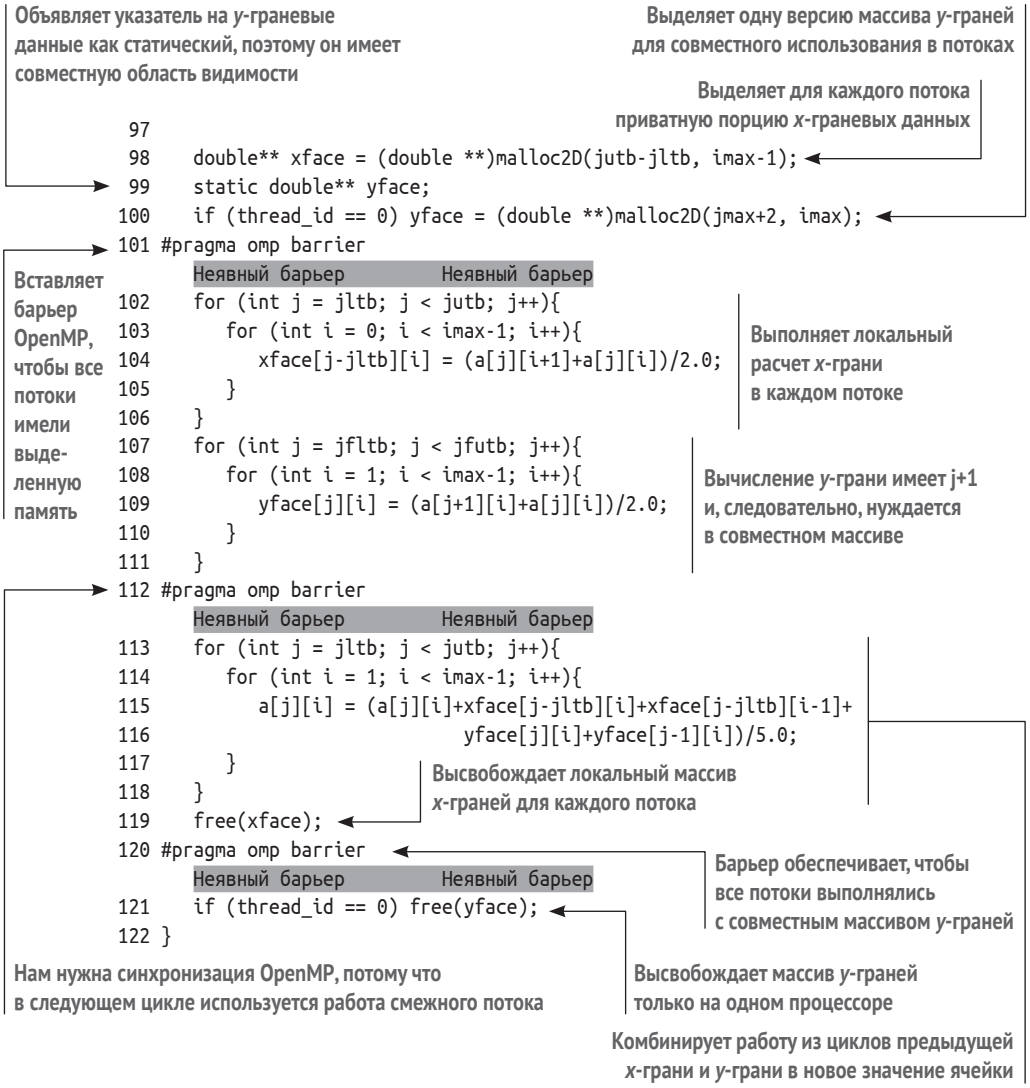
Листинг 7.20 Стенсильный оператор с разводом направлений с OpenMP

SplitStencil/SplitStencil_opt1.c

```
 86 void SplitStencil(double **a, int imax, int jmax)
 87 {
 88    int thread_id = omp_get_thread_num();
 89    int nthreads = omp_get_num_threads();
 90
 91    int jltb = 1 + (jmax-2) * ( thread_id ) / nthreads;
 92    int jutb = 1 + (jmax-2) * ( thread_id + 1 ) / nthreads;
 93
 94    int jfltb = jltb;
 95    int jfutb = jutb;
 96    if (thread_id == 0) jfltb--;
```

Вручную вычисляет распределение данных по потокам

y-грани имеют на одно распределяемое значение меньше



В целях определения памяти в стеке, как показано в направлении *x*, нам нужен указатель на указатель на число двойной точности (`double **xface`), чтобы указатель находился в стеке и был приватным для каждого потока. Затем мы выделяем память, используя в строке 98 в листинге 7.20 специальный вызов двухмерной `malloc`. Нам нужна память в объеме, достаточном только для каждого потока, поэтому в строках 91 и 92 мы вычисляем границы потока и используем их в вызове двухмерной `malloc`. Память выделяется из кучи и может использоваться совместно, но каждый поток имеет только свой собственный указатель; поэтому каждый поток не может обращаться к памяти других потоков.

Вместо выделения памяти из кучи мы могли бы использовать автоматическое выделение, такое как `double xface[3][6]`, при котором память

автоматически выделяется в стеке. Компилятор автоматически видит это объявление и помещает пространство памяти в стек. В тех случаях, когда массивы велики, компилятор может переместить потребность в памяти в кучу. У каждого компилятора есть свой порог при принятии решения о месте, где будет располагаться память: в куче либо в стеке. Если компилятор перемещает местоположение памяти в кучу, то только один поток имеет указатель на это местоположение. По сути, он является приватным, даже если находится в совместном пространстве памяти.

Для граней у мы определяем статический указатель на указатель (`static double **yface`); при этом в них все потоки могут обращаться к одному и тому же указателю. В этом случае только один поток должен выполнять это выделение памяти, и все остальные потоки могут обращаться к этому указателю и к самой памяти. В данном примере вы можете использовать рис. 7.7, чтобы увидеть разные варианты совместного использования памяти. В данном случае вы должны перейти к Параллельному участку -> подпрограмма С и выбрать одну из переменных файловой области видимости, `extern` либо `static`, чтобы сделать указатель совместным для потоков. При этом легко в чем-то ошибиться, к примеру в области видимости переменных, распределении памяти или синхронизации. Например, что произойдет, если мы просто определим обычный указатель `double **yfaces`? Теперь у каждого потока будет свой собственный приватный указатель, но память будет выделяться только одному из них. Указатель для второго потока ни на что не будет указывать, что приведет к ошибке при его использовании.

На рис. 7.13 показана производительность при выполнении поточной версии кода на процессоре Skylake Gold. Для малого числа потоков мы получаем сверхлинейное ускорение, прежде чем упасть более чем на восемь потоков. Сверхлинейное ускорение иногда происходит из-за повышения производительности кеша, поскольку данные подразделяются между потоками либо процессорами.

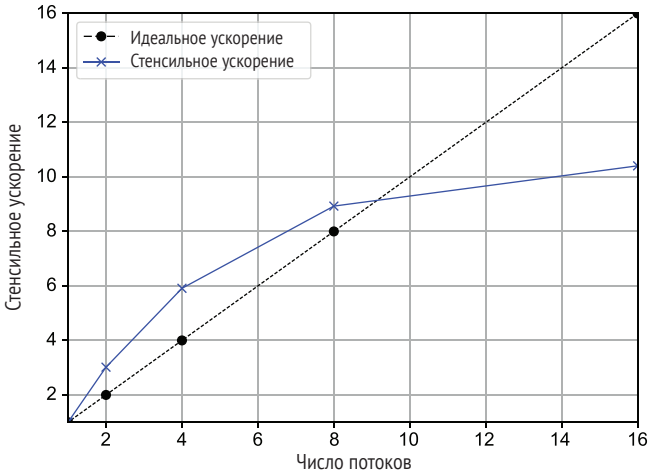


Рис. 7.13 Версия разделенного стенсиля с использованием потоков имеет сверхлинейное ускорение для 2–8 потоков

ОПРЕДЕЛЕНИЕ *Сверхлинейное ускорение* – это производительность, превосходящая идеальную масштабную кривую для сильного масштабирования. Оно может происходить вследствие того, что меньшие размеры массива вписываются в более высокий уровень кеша, что и приводит к повышению производительности кеша.

7.8.2 Имплементация суммирования по Кахану с потокообразованием OpenMP

Для алгоритма суммирования по Кахану повышенной прецизионности, представленного в разделе 5.7, мы не можем использовать прагму, чтобы побуждать компилятор генерировать мультипоточную имплементацию из-за несомых циклом зависимостей. Поэтому мы будем следовать аналогичному алгоритму, который мы использовали в векторизованной имплементации в разделе 6.3.4. В первой фазе вычисления мы сначала просуммируем значения в каждом потоке. Затем просуммируем значения по всем потокам, чтобы получить окончательную сумму, как показано в следующем ниже листинге.

Листинг 7.21 Имплементация суммирования по Кахану с OpenMP

GlobalSums/kahan_sum.c

```
 1 #include <stdlib.h>
 2 #include <omp.h>
 3
 4 double do_kahan_sum(double* restrict var, long ncells)
 5 {
 6    struct esum_type{
 7       double sum;
 8       double correction;
 9    };
10
11    int nthreads = 1;     | Получает суммарное число
12    int thread_id = 0;    | потоков и thread_id
13 #ifdef _OPENMP
14   nthreads = omp_get_num_threads();
15    thread_id = omp_get_thread_num();
16 #endif
17
18    struct esum_type local;
19    local.sum = 0.0;
20    local.correction = 0.0;
21
22    int tbegin = ncells * ( thread_id ) / nthreads;     | Вычисляет диапазон,
23    int tend = ncells * ( thread_id + 1 ) / nthreads;   | за который отвечает этот поток
24
25    for (long i = tbegin; i < tend; i++) {
26       double corrected_next_term = var[i] + local.correction;
27       double new_sum = local.sum + local.correction;
28       local.correction = corrected_next_term - (new_sum - local.sum);
```

```
29       local.sum = new_sum;
30    }
31
32    static struct esum_type *thread;
33    static double sum;
34
35 #ifdef _OPENMP
36 #pragma omp masked
37    thread = malloc(nthreads*sizeof(struct esum_type));
38 #pragma omp barrier
      Неявный барьер          Неявный барьер
39
40    thread[thread_id].sum = local.sum;
41    thread[thread_id].correction = local.correction;
42
43 #pragma omp barrier
      Неявный барьер          Неявный барьер
44
45    static struct esum_type global;
46 #pragma omp masked
47    {
48       global.sum = 0.0;
49       global.correction = 0.0;
50       for ( int i = 0 ; i < nthreads ; i ++ ) {
51          double corrected_next_term = thread[i].sum +
52                 thread[i].correction + global.correction;
53          double new_sum = global.sum + global.correction;
54          global.correction = corrected_next_term -
                              (new_sum - global.sum);
55          global.sum = new_sum;
56    }
57
58       sum = global.sum + global.correction;
59       free(thread);
60    } // конец omp masked
61 #pragma omp barrier
      Неявный барьер          Неявный барьер
62 #else
63    sum = local.sum + local.correction;
64 #endif
65
66    return(sum);
67 }
```

Помещает переменные в совместную память

Определяет компиляторную переменную _OPENMP при использовании OpenMP

Выделяет один поток в совместной памяти

Хранит сумму каждого потока в массиве

Ждет до тех пор, пока все потоки не доберутся сюда, а затем выполняет суммирование по всем потокам

Использует один поток для вычисления начального сдвига для каждого потока

7.8.3 Поточная имплементация алгоритма префиксного сканирования

В этом разделе мы рассмотрим поточную имплементацию операции префиксного сканирования. Представленная в разделе 5.6 операция префиксного сканирования важна для алгоритмов с нерегулярными данными. Это связано с тем, что количество (count), служащее для определения начального местоположения для рангов или потоков, позволяет выпол-

нять остальные вычисления параллельно. Как обсуждалось в этом разделе, префиксное сканирование также может выполняться параллельно, давая еще одну выгоду от параллелизации. Процесс имплементации состоит из трех фаз.

- *Все потоки* – вычисляет префиксный скан для части данных каждого потока.
- *Один поток* – вычисляет начальный сдвиг для данных каждого потока.
- *Все потоки* – применяет новый сдвиг потока ко всем данным по каждому потоку.

Описанная в листинге 7.22 имплементация работает для последовательного приложения и при вызове из параллельного участка OpenMP. Выгода от этого в том, что есть возможность использовать код в листинге как для последовательных, так и для поточных случаев, сокращая дублирование необходимого для этой операции кода.

Листинг 7.22 Имплементация префиксного сканирования с OpenMP

PrefixScan/PrefixScan.c

```
 1 void PrefixScan (int *input, int *output, int length)
 2 {
 3    int nthreads = 1;
 4    int thread_id = 0;
 5 #ifdef _OPENMP
 6    nthreads = omp_get_num_threads();
 7    thread_id = omp_get_thread_num();
 8 #endif
 9
10    int tbegin = length * ( thread_id ) / nthreads;
11    int tend = length * ( thread_id + 1 ) / nthreads;
12
13    if ( tbegin < tend ) {
14       output[tbegin] = 0;
15       for ( int i = tbegin + 1 ; i < tend ; i++ ) {
16          output[i] = output[i-1] + input[i-1];
17       }
18    }
19    if (nthreads == 1) return;
20
21 #ifdef _OPENMP
22 #pragma omp barrier
      Неявный барьер          Неявный барьер
23
24    if (thread_id == 0) {
25       for ( int i = 1 ; i < nthreads ; i ++ ) {
26          int ibegin = length * ( i - 1 ) / nthreads;
27          int iend = length * ( i ) / nthreads;
28
29          if ( ibegin < iend )
30             output[iend] = output[ibegin] + input[iend-1];
```

Эта операция выполняется только при наличии положительного числа записей

Получает суммарное число потоков и thread_id

Вычисляет диапазон, за который этот поток отвечает

Выполняет эксклюзивное сканирование по каждому потоку

Только для нескольких потоков выполнить корректировку префиксного сканирования в отношении начального значения по каждому потоку

Ждет, пока все потоки не попадут сюда

Использует главный поток для вычисления начального сдвига по каждому потоку

```
31
32          if ( ibegin < iend - 1 )
33             output[iend] += output[iend-1];
34       }
35    }
36 #pragma omp barrier
     Неявный барьер          Неявный барьер
37
38 #pragma omp simd
39    for ( int i = tbegin + 1 ; i < tend ; i++ ) {
40       output[i] += output[tbegin];
41    }
42 #endif
43}
```

Заканчивает расчет на главном потоке с барьером

Снова запускает все потоки

Применяет сдвиг к диапазону для этого потока

Этот алгоритм теоретически должен масштабироваться следующим образом:

```
Parallel_timer = 2 * serial_time/nthreads
```

Производительность архитектуры Skylake Gold 6152 достигает максимума примерно при 44 потоках, что в 9.4 раза быстрее, чем в последовательной версии.

7.9 Инструменты потокообразования, необходимые для устойчивых имплементаций

Разрабатывать устойчивую имплементацию на основе OpenMP трудно без использования специализированных инструментов для обнаружения гоночных состояний в потоках и узких мест в производительности. Важность использования инструментов значительно нарастает по мере того, как вы пытаетесь получать более высокопроизводительную имплементацию OpenMP. Существуют как коммерческие, так и общедоступные инструменты. Типичный список инструментов при интегрировании продвинутых имплементаций OpenMP в вашем приложении включает:

- *Valgrind* – инструмент для работы с памятью, представленный в разделе 2.1.3. Он также работает с OpenMP и помогает находить неинициализированную память или обращения за пределы потоков;
- *граф вызовов* – инструмент cachegrind порождает граф вызовов и профиль вашего приложения. Граф вызовов идентифицирует функции, которые вызывают другие функции, чтобы четко отображать иерархию вызовов и путь кода. Пример инструмента cachegrind был представлен в разделе 3.3.1;
- *Allinea/ARM MAP* – высокоуровневый профилировщик для получения совокупной стоимости поточных запуска и барьеров (для приложений OpenMP);
- *Intel® Inspector* – служит для обнаружения гоночных состояний в потоках (для приложений OpenMP).

Первые два инструмента были описаны в предыдущих главах; мы отсылаем вас к ним. В данном разделе мы обсудим последние два инструмента, поскольку они больше относятся к приложению OpenMP. Эти инструменты необходимы для определения узких мест и понимания того, где они лежат внутри вашего приложения и, значит, необходимы для того, чтобы знать, с чего лучше всего начинать эффективное изменение кода.

7.9.1 Использование профилировщика Allinea/ARM MAP для быстрого получения высокоуровневого профиля вашего приложения

Одним из лучших инструментов для получения высокоуровневого профиля приложения является профилировщик Allinea/ARM MAP. На рис. 7.14 показан упрощенный вид его интерфейса. Для приложения OpenMP он показывает стоимость поточного запуска и ожидания, выделяет узкие места приложения и показывает объем полезного использования памяти CPU с плавающей точкой. Указанный профилировщик позволяет легко сравнивать результаты, полученные до и после изменений кода. Allinea/ARM MAP отлично подходит для обеспечения быстрой и высокоуровневой панорамы вашего приложения. Но помимо него можно использовать много других профилировщиков. Некоторые из них рассмотрены в разделе 17.3.

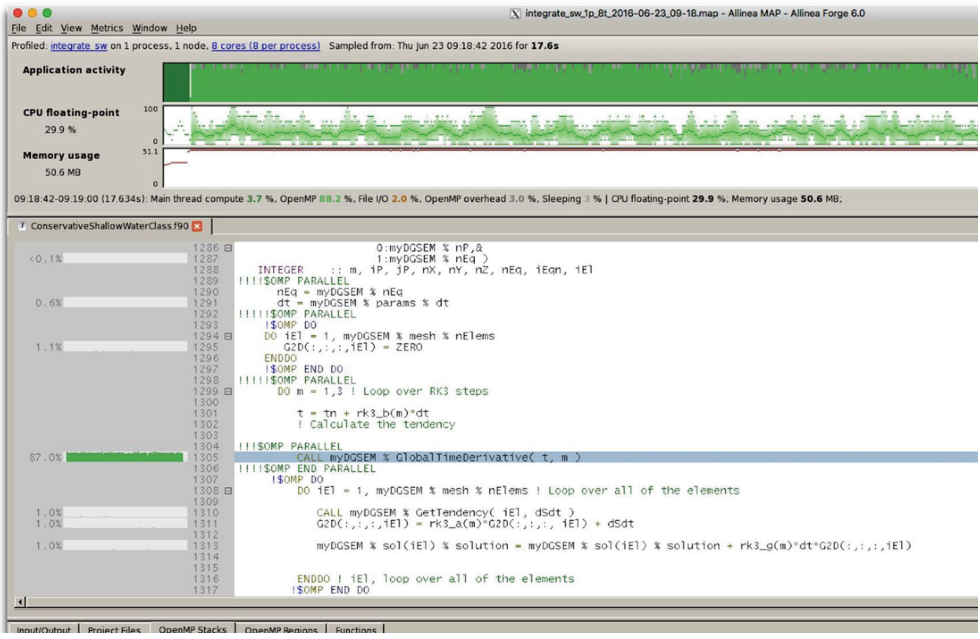


Рис. 7.14 Результаты, полученные из профилировщика Allinea/ARM MAP, показывающие, что бо́льшая часть вычислительного времени находится в выделенной строке кода. Мы часто используем подобные индикаторы, чтобы показывать расположение узких мест

7.9.2 Отыскание гоночных состояний в потоках с помощью Intel® Inspector

В имплементации OpenMP важно находить и устранять гоночные состояния в потоках, чтобы создавать устойчивое приложение производственного качества. Для этой цели необходимы инструменты, потому что даже самый лучший программист не сможет уловить все гоночные состояния в потоках. По мере того как приложение начинает масштабироваться, ошибки в памяти возникают все чаще и могут приводить к сбою приложения. Обнаружение этих ошибок памяти на ранней стадии экономит время и энергию при будущих прогонах.

Существует не так много инструментов, которые эффективны для определения гоночных состояний в потоках. Мы показываем использование одного из этих инструментов, Intel® Inspector, обнаружения и точного установления мест этих гоночных состояний. Наличие инструментов для понимания гоночных состояний среди потоков в памяти также полезно при масштабировании до большего числа потоков. На рис. 7.15 приведен образец скриншота программы Intel® Inspector.

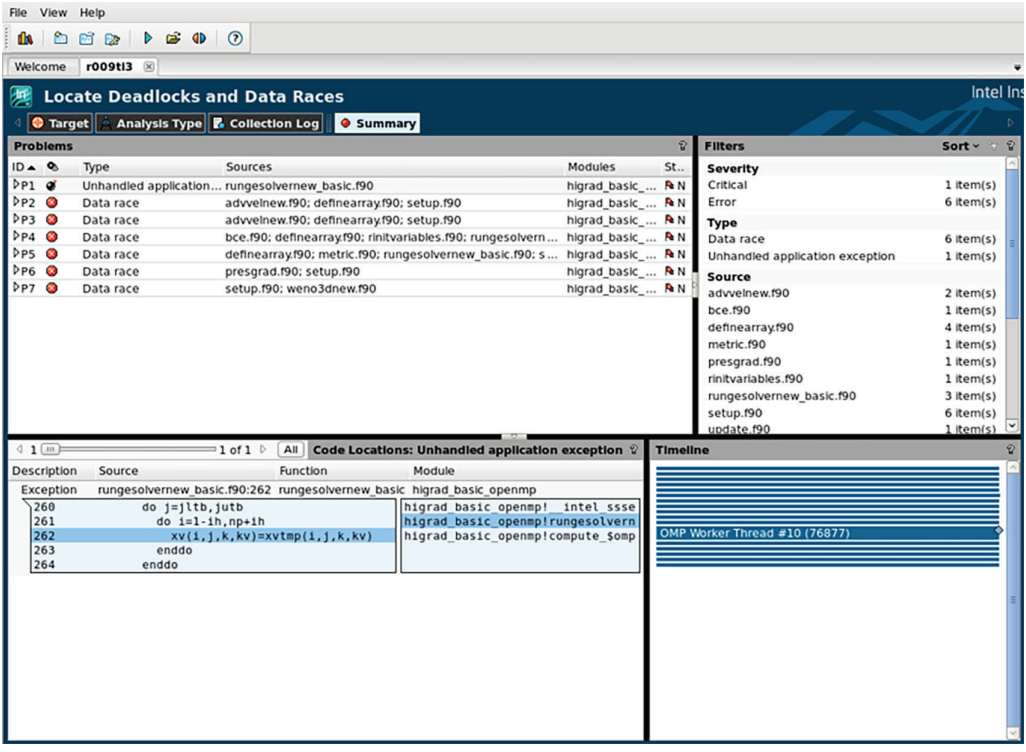


Рис. 7.15 Отчет программы Intel® Inspector, показывающий обнаружение гоночных состояний в потоках. Здесь элементы, перечисленные как Data race (Гонка данных) под заголовком Type (Тип) на панели слева вверху, показывают все места, где в настоящее время существует гоночное состояние

Перед внесением изменений в первоначальное приложение крайне важно выполнить регрессионное тестирование. Обеспечение правильности имеет решающее значение для успешного имплементирования потокообразования OpenMP. Невозможно имплементировать правильный код OpenMP, если приложение или вся подпрограмма целиком не находятся в надлежащем рабочем состоянии. Это также требует, чтобы раздел кода с потокообразованием OpenMP также выполнялся в регрессионном тесте. Без возможности проведения регрессионного тестирования становится трудно добиться стабильных результатов. Таким образом, эти инструменты, наряду с регрессионным тестированием, позволяют создавать более глубокое понимание зависимостей, эффективности и правильности в большинстве приложений.

7.10 Пример алгоритма поддержки на основе операционных задач

Параллельная стратегия, основанная на операционных задачах, была впервые представлена в главе 1 и проиллюстрирована на рис. 1.25. Используя задачно-ориентированный подход, вы можете разделять работу на отдельные задачи, которые затем могут распределяться по отдельным процессам. Многие алгоритмы естественнее выражать в терминах подхода на основе операционных задач. OpenMP поддерживает этот тип подхода начиная с версии 3.0. В последующих выпусках стандарта были внесены дополнительные улучшения в задачно-ориентированную модель. В этом разделе мы покажем простой задачно-ориентированный алгоритм, чтобы проиллюстрировать методы в OpenMP.

Один из подходов к воспроизводимой глобальной сумме состоит в том, чтобы суммировать значения попарно. Обычный подход к массиву требует выделения рабочего массива и некоторой сложной логики индексирования. Использование подхода, основанного на операционных задачах, как показано на рис. 7.16, позволяет избегать необходимости в рабочем массиве, рекурсивно разделяя данные пополам в нисходящем

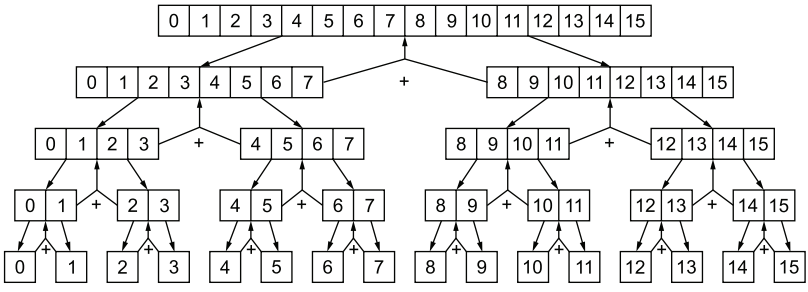


Рис. 7.16 Имплементация на основе операционных задач рекурсивно разбивает массив пополам в нисходящем витке. Как только размер массива равен 1, задача суммирует пары данных в восходящем витке

витке, пока не будет достигнута длина массива 1, а затем суммируя пары в восходящем витке.

В листинге 7.23 показан код задачно-ориентированного подхода. Порождение операционной задачи должно выполняться в параллельном участке, но только одним потоком, приводя к вложенным блокам прагм в строках с 8 по 14.

Листинг 7.23 Попарное суммирование с использованием операционных задач OpenMP

PairwiseSumByTask/PairwiseSumByTask.c

```
 1 #include <omp.h>
 2
 3 double PairwiseSumBySubtask(double* restrict var, long nstart, long nend);
 4
 5 double PairwiseSumByTask(double* restrict var, long ncells)
 6 {
 7    double sum;
 8    #pragma omp parallel >> Порождать потоки >>
 9    {
10       #pragma omp masked
11       {
12          sum = PairwiseSumBySubtask(var, 0, ncells);
13       }
14    } Неявный барьер          Неявный барьер
15    return(sum);
16 }
17
18 double PairwiseSumBySubtask(double* restrict var, long nstart, long nend)
19 {
20    long nsize = nend - nstart;
21    long nmid = nsize/2;
22    double x,y;
23    if (nsize == 1){
24       return(var[nstart]);
25    }
26
27    #pragma omp task shared(x) mergeable final(nsize > 10)
28    x = PairwiseSumBySubtask(var, nstart, nstart + nmid);
29    #pragma omp task shared(y) mergeable final(nsize > 10)
30    y = PairwiseSumBySubtask(var, nend - nmid, nend);
31    #pragma omp taskwait
32
33    return(x+y);
34 }
```

Запускает параллельный участок

Запускает главную операционную задача в одном потоке

Подразделяет массив на две части

Инициализирует сумму в листе одним единственным значением из массива

Запускает пару подзадач с половиной данных для каждой

Ждет завершения двух задач

Суммирует значения из двух подзадач и возвращает в вызывающий поток

Для достижения хорошей производительности с помощью алгоритма, основанного на операционных задачах, требуется гораздо больше настроечной работы, чтобы предотвращать появление слишком большого

числа потоков и сохранять гранулярность задач на разумном уровне. Для некоторых алгоритмов алгоритмы, основанные на операционных задачах, являются гораздо более надлежащей параллельной стратегией.

7.11 Материалы для дальнейшего изучения

Традиционное программирование OpenMP на основе потоков освещается массой учебных материалов. Поскольку почти каждый компилятор поддерживает OpenMP, самый лучший подход к усвоению этой темы состоит в том, чтобы начать добавлять директивы OpenMP в свой код. Существует целый ряд возможностей для тренировки своих навыков, охватывающих OpenMP, включая ежегодную конференцию по суперкомпьютерам, которая проводится в ноябре. Дополнительная информация по конференции находится по адресу https://sc21.supercomputing.org/. Для тех, кто еще больше интересуется OpenMP, существует ежегодный Международный семинар по OpenMP, посвященный последним разработкам. Дополнительную информацию по семинару можно получить по адресу: http://www.iwomp.org/.

7.11.1 Дополнительное чтение

Барбара Чапмен является одним из ведущих авторов и авторитетов в области OpenMP. Ее книга 2008 года является стандартным справочником по программированию OpenMP, в особенности в том, что касается имплементации потокообразования в OpenMP:

- Барбара Чапмен, Габриэле Йост и Рууд Ван Дер Пас, «Использование OpenMP: параллельное программирование с использованием переносимой совместной памяти» (Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas, *Using OpenMP: portable shared memory parallel programming*, vol. 10, MIT Press, 2008).

Много исследователей работают над разработкой более эффективных технических приемов имплементирования OpenMP, получивших название *OpenMP высокого уровня*. Ниже приведена ссылка на слайды, в которых OpenMP высокого уровня описывается подробнее:

- Юлиана Замора, «Эффективные имплементации OpenMP на базе Intel Knights Landing» (Yuliana Zamora, Effective OpenMP Implementations on Intel's Knights Landing, Los Alamos National Laboratory Technical Report LA-UR-16-26774, 2016). Доступно по адресу: https://www.osti.gov/biblio/1565920-effective-openmp-implementations-intel-knights-landing.

Хороший учебник по OpenMP и MPI написан Питером Пачеко. В тексте есть несколько хороших примеров кода OpenMP.

- Питер Пачеко, «Введение в параллельное программирование» (Peter Pacheco, *An introduction to parallel programming*, Elsevier, 2011).

Блейз Барни из Ливерморской национальной лаборатории Лоуренса написал хорошо написанный справочник по OpenMP, который также доступен онлайн:

- Блейз Барни, «Учебное пособие по OpenMP» (Blaise Barney, *OpenMP Tutorial*), https://computing.llnl.gov/tutorials/openMP/.

Совет по пересмотру архитектуры OpenMP (Architecture Review Board, ARB) поддерживает веб-сайт, который является официальным местом для всего, что связано с OpenMP, от спецификаций до презентаций и учебных пособий:

- Совет по пересмотру архитектуры OpenMP, «*OpenMP*», https://www.openmp.org.

Более глубокое обсуждение трудностей потокообразования можно найти в приведенной ниже статье:

- Эдвард Ли, «Проблема с потоками» (Edward A Lee, The problem with threads, *Computer 39*, no. 5, 2006: 33–42).

7.11.2 Упражнения

1. Конвертируйте пример векторного сложения из листинга 7.8 в OpenMP высокого уровня, следуя инструкциям раздела 7.2.2.
2. Напишите процедуру для получения максимального значения в массиве. Добавьте прагму OpenMP, чтобы добавить в процедуру поточный параллелизм.
3. Напишите версию редукции из предыдущего упражнения на основе OpenMP высокого уровня.

В этой главе мы рассмотрели значительный объем материала. Данная прочная основа поможет вам в разработке эффективного приложения OpenMP.

Резюме

- Имплементации OpenMP уровня цикла создаются быстро и легко.
- Эффективная имплементация OpenMP может обеспечивать многообещающее ускорение приложений.
- Хорошие имплементации на основе первого касания нередко повышают производительность на 10–20 %.
- В целях приведения исходного кода OpenMP в рабочее состояние важно понимать область видимости переменных между потоками (виртуальными ядрами).
- OpenMP высокого уровня может повышать производительность на текущих и будущих мультиядерных архитектурах.
- Инструменты потокообразования и отладки необходимы при имплементировании более сложных версий OpenMP.
- Несколько предлагаемых в этой главе стилевых рекомендаций таковы:

- объявлять переменные там, где они используются, чтобы они становились приватными автоматически, что в целом правильно;
- модифицировать объявления с целью получения верной поточной области видимости переменных вместо того, чтобы использовать обширный список в приватных и публичных выражениях;
- по возможности избегать выражения `critical` или других замковых конструктов. Указанные конструкты, как правило, сильно влияют на производительность;
- сокращать синхронизацию путем добавления выражений `nowait` в циклы `for` и ограничивать использование барьера `#pragma omp` только там, где это необходимо;
- объединять малые параллельные участки в меньшее число более крупных параллельных участков с целью снижения накладных расходов OpenMP.

8 MPI: параллельный становой хребет

Эта глава охватывает следующие ниже темы:

- отправку сообщений из одного процесса в другой;
- выполнение распространенных шаблонов обмена данными с помощью коллективных вызовов MPI;
- связывание сеток на отдельных процессах с помощью обмена данными;
- создание конкретно-прикладных типов данных MPI и использование функций декартовой топологии MPI;
- написание приложений с использованием гибридной техники MPI плюс OpenMP.

Важность стандарта «Интерфейс передачи сообщений» (Message Passing Interface, MPI) заключается в том, что он позволяет программе обращаться к дополнительным вычислительным узлам и, следовательно, выполнять все более и более крупные задачи, добавляя в симуляцию больше узлов. Название «передача сообщений» относится к возможности легко отправлять сообщения из одного процесса в другой. MPI широко распространен в сфере высокопроизводительных вычислений. Во многих научных областях использование суперкомпьютеров влечет за собой имплементацию MPI.

MPI был запущен в качестве открытого стандарта в 1994 году и в течение нескольких месяцев стал доминирующим библиотечно-ориентированным языком параллельных вычислений. С 1994 года использование MPI привело к научным прорывам от физики до машинного обучения и самостоятельного вождения автомобилей! В настоящее время широко используется несколько имплементаций MPI. Двумя наиболее распространенными являются MPICH из Аргоннских национальных лабораторий и OpenMPI. Поставщики оборудования часто имеют для своих платформ индивидуальные версии одной из этих двух имплементаций. Стандарт MPI, в настоящее время вплоть до версии 3.1 на 2015 год, продолжает эволюционировать и меняться.

В этой главе мы покажем вам, как имплементировать MPI в вашем приложении. Мы начнем с простой программы MPI, а затем перейдем к более сложному примеру того, как связывать воедино отдельные вычислительные сетки на отдельных процессах посредством передачи информации о границах. Мы коснемся нескольких передовых технических приемов, которые важны для хорошо написанных программ MPI, таких как создание конкретно-прикладных типов данных MPI и использование функций декартовой (картезианской) топологии MPI. Наконец, мы введем комбинацию MPI с OpenMP (MPI плюс OpenMPI) и векторизацию, чтобы получать несколько уровней параллелизма.

ПРИМЕЧАНИЕ Мы рекомендуем вам сверяться с примерами этой главы, расположенными по адресу https://github.com/EssentialsofParallelComputing/Chapter8.

8.1 Основы программы MPI

В этом разделе мы рассмотрим основы, необходимые для минимальной программы MPI. Некоторые из этих базовых требований определены стандартом MPI, в то время как другие общеприняты в большинстве имплементаций MPI. Базовая структура и функционирование MPI оставались удивительно состыкующимися со времен первого стандарта.

Начнем с того, что MPI представляет собой полностью библиотечно-ориентированный язык. Для него не требуется специального компилятора или удобств со стороны операционной системы. Все программы MPI имеют базовую структуру и процесс, как показано на рис. 8.1. MPI всегда начинается с вызова `MPI_Init` прямо в начале программы и `MPI_Finalize` при выходе из программы. Это контрастирует с OpenMP, как обсуждалось в главе 7, который не нуждается в специальных командах запуска и выключения и просто размещает параллельные директивы вокруг ключевых циклов.

После написания параллельной программы MPI она компилируется с включаемым файлом и библиотекой. Затем она выполняется с помощью специальной программы запуска, которая формирует параллельные процессы между узлами и внутри узла.

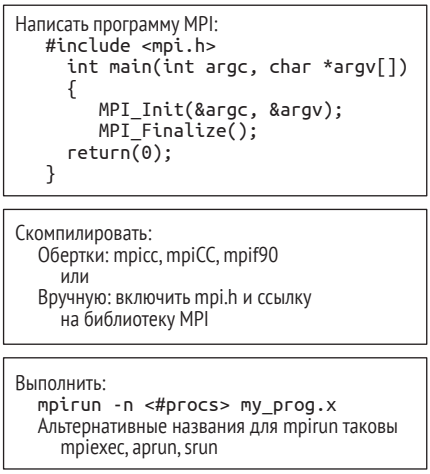


Рис. 8.1 Подход MPI основан на библиотеке. Надо просто скомпилировать, привязав библиотеку MPI, и запустить с помощью специальной программы параллельного запуска

8.1.1 Базовые функциональные вызовы MPI для каждой программы MPI

Базовые функциональные вызовы MPI включают `MPI_Init` и `MPI_Finalize`. Вызов `MPI_Init` должен стоять сразу после запуска программы, а аргументы из процедуры `main` должны передаваться инициализационному вызову. Типичные вызовы выглядят следующим образом и могут иметь (или не иметь) переменную `return`:

```
iret = MPI_Init(&argc, &argv);
iret = MPI_Finalize();
```

Большинству программ будет требоваться число процессов и процессный ранг внутри группы, способной обмениваться данными и именуемой *коммуникатором*. Одной из главных функций MPI-интерфейса является запуск удаленных процессов и связывание их так, чтобы сообщения могли передаваться между процессами. По умолчанию коммуникатором является `MPI_COMM_WORLD`, который настраивается в начале каждого параллельного задания функцией `MPI_Init`. Давайте сделаем паузу и взглянем на несколько определений:

- *процесс* – независимая вычислительная единица, которая владеет частью памяти и контролирует ресурсы в пользовательском пространстве;
- *ранг* – уникальный, переносимый идентификатор для различения индивидуального процесса в наборе процессов. Обычно это будет целое число в наборе целых чисел от нуля до числа процессов на единицу меньше.

Вызовы для получения этих важных переменных таковы:

```
iret = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
iret = MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
```

8.1.2 Компиляторные обертки для более простых программ MPI

Хотя MPI и является библиотекой, мы можем относиться к ней как к компилятору посредством компиляторных оберток MPI. Это упрощает строительство приложений MPI, поскольку вам не нужно знать библиотеки, которые требуются, и их местоположение. Они особенно удобны для малых приложений MPI. Для каждого языка программирования существуют свои компиляторные обертки:

- `mpicc` – обертка для кода на языке C;
- `mpicxx` – обертка для C++ (также может быть `mpiCC` или `mpic++`);
- `mpifort` – обертка для Fortran (также может быть `mpif77` или `mpif90`).

Использовать эти обертки необязательно. Если вы не используете компиляторные обертки, то они все равно бывают полезны для идентифицирования компиляторных флагов, необходимых для создания вашего приложения. Команда `mpicc` имеет опции, которые выдают эту информацию. Указанные опции для вашего MPI можно найти с помощью команды `man mpicc`. Для двух приведенных ниже наиболее популярных имплементаций MPI мы перечислим консольные опции для `mpicc`, `mpicxx` и `mpifort`.

- Для OpenMPI используются следующие ниже командные опции:
  - `--showme`;
  - `--showme:compile`;
  - `--showme:link`.
- Для MPICH используются следующие ниже командные опции:
  - `-show`;
  - `-compile_info`;
  - `-link_info`.

8.1.3 Использование команд параллельного запуска

Запуск параллельных процессов для MPI – это сложная операция, которая обрабатывается специальной командой. Поначалу этой командой нередко была `mpirun`. Но с выпуском стандарта MPI 2.0 в 1997 году в качестве команды запуска была рекомендована `mpiexec`, чтобы попытаться обеспечить большую переносимость. Однако эта попытка стандартизации не была полностью успешной, и сегодня для команды запуска используется несколько имен:

- `mpirun -n <nprocs>`;
- `mpiexec -n <nprocs>`;
- `aprun`;
- `srun`.

В большинстве команд запуска MPI используется опция `-n`, обозначающая число процессов, но другие могут принимать значение `-np`. Учитывая сложность современных архитектур компьютерных узлов, команды запуска имеют мириады опций для аффинности, размеще-

ния и среды (часть из которых мы обсудим в главе 14). Указанные опции зависят от каждой имплементации MPI и даже от каждого релиза их библиотек MPI. Простота опций, доступных в изначальных командах запуска, превратилась в бесконечное болото опций, которые еще не совсем стабилизировались. К счастью, для начинающего пользователя MPI *большинство этих опций можно проигнорировать*, но они важны для продвинутого применения и тонкой настройки.

8.1.4 Минимально работающий пример программы MPI

Теперь, когда мы усвоили все базовые компоненты, мы можем их скомбинировать в минимальный рабочий пример, который показан в листинге 8.1: мы запускаем параллельное задание и распечатываем ранг и число процессов из каждого процесса. В вызове для получения ранга и размера мы используем переменную `MPI_COMM_WORLD`, которая обозначает группу всех процессов MPI и предопределена в заголовочном файле MPI. Обратите внимание, что результат может распечатываться в любом порядке; программа MPI предоставляет операционной системе право решать, когда и как он распечатывается.

Листинг 8.1 Минимально работающий пример MPI

MinWorkExampleMPI.c

```
 1 #include <mpi.h>
 2 #include <stdio.h>
 3 int main(int argc, char **argv)
 4 {
 5    MPI_Init(&argc, &argv);
 6
 7    int rank, nprocs;
 8    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 9    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
10
11    printf("Ранг %d процесса %d\n", rank, nprocs);
12
13    MPI_Finalize();
14    return 0;
15 }
```

Включаемый файл для функций и переменных MPI

Инициализируется после запуска программы, в том числе аргументы программы

Получает ранговый номер процесса

Получает число рангов в программе, определенное командой mpirun

Завершает MPI для синхронизации рангов, а затем выходит из программы

В листинге 8.2 определен простой файл makefile для сборки этого примера с использованием компиляторных оберток MPI. В данном случае мы используем обертку mpicc для указания местоположения включаемого файла mpi.h и библиотеки MPI.

Листинг 8.2 Простой makefile с использованием компиляторных оберток MPI

MinWorkExample/Makefile.simple

```
default:        MinWorkExampleMPI
```

```
all:    MinWorkExampleMPI

MinWorkExampleMPI: MinWorkExampleMPI.c Makefile
        mpicc MinWorkExampleMPI.c -o MinWorkExampleMPI
clean:
        rm -f MinWorkExampleMPI MinWorkExampleMPI.o
```

Для более сложных сборок на различных системах вы можете предпочесть CMake. В следующем ниже листинге показан файл CMakeLists.txt для этой программы.

Листинг 8.3 Файл CMakeLists.txt для сборки с помощью CMake

```
MinWorkExample/CMakeLists.txt
cmake_minimum_required(VERSION 2.8)

project(MinWorkExampleMPI)

# Для этого проекта требуется MPI:
find_package(MPI REQUIRED)

add_executable(MinWorkExampleMPI MinWorkExampleMPI.c)

target_include_directories(MinWorkExampleMPI
   PRIVATE ${MPI_C_INCLUDE_PATH})
target_compile_options(MinWorkExampleMPI
   PRIVATE ${MPI_C_COMPILE_FLAGS})
target_link_libraries(MinWorkExampleMPI
   ${MPI_C_LIBRARIES} ${MPI_C_LINK_FLAGS})

# Добавить тест:
enable_testing()
add_test(MPITest ${MPIEXEC} ${MPIEXEC_NUMPROC_FLAG}
   ${MPIEXEC_MAX_NUMPROCS}
   ${MPIEXEC_PREFLAGS}
   ${CMAKE_CURRENT_BINARY_DIR}/MinWorkExampleMPI
   ${MPIEXEC_POSTFLAGS})

# Очистка
add_custom_target(distclean COMMAND rm -rf CMakeCache.txt CMakeFiles
                  Makefile cmake_install.cmake CTestTestfile.cmake Testing)
```

Вызывает специальный модуль для отыскания MPI и задает переменные

Модифицирует компиляторные флаги

Создает переносимый тест MPI

Теперь, используя сборочную систему CMake, давайте сконфигурируем, соберем, а затем выполним тест с помощью следующих ниже команд:

```
cmake .
make
make test
```

Операция записи из команды `printf` выводит на экран результат в любом порядке. Наконец, для очистки после прогона следует применить следующие ниже команды:

```
make clean
make distclean
```

8.2 Команды отправки и приемки для обмена данными «из процесса в процесс»

Сердцевина подхода на основе передачи сообщений заключается в отправке сообщения из точки в точку или, пожалуй, точнее, из процесса в процесс. Весь смысл параллельной обработки заключается в координации работы. Для этого вам нужно отправлять сообщения для контроля либо для распределения работы. Мы покажем вам, как эти сообщения составляются и правильно отправляются. Существует масса вариантов процедур «из точки в точку»; мы рассмотрим те из них, которые рекомендуется использовать в большинстве ситуаций.

На рис. 8.2 показаны компоненты сообщения. На любом конце системы должен быть почтовый ящик. Размер почтового ящика имеет важное значение. Отправляющая сторона знает размер сообщения, но принимающая сторона этого не знает. В целях обеспечения места под хранение сообщения обычно лучше сперва отправлять (post) принятое сообщение. Это позволяет избегать задержки сообщения в силу того, что принимающему процессу необходимо выделять временное пространство под хранение сообщения на время, пока принятое сообщение не будет отправлено и процесс не сможет скопировать его в надлежащее место. По аналогии, если принятое сообщение (почтовый ящик) не отправлено (его там нет), то почтальон должен задержаться, постоянно наведываясь до тех пор, пока кто-нибудь его не предоставит. Отправка принятого сообщения с самого начала позволяет избегать возможности нехватки пространства в памяти на принимающей стороне для временного буфера, выделяемого под хранение сообщения.

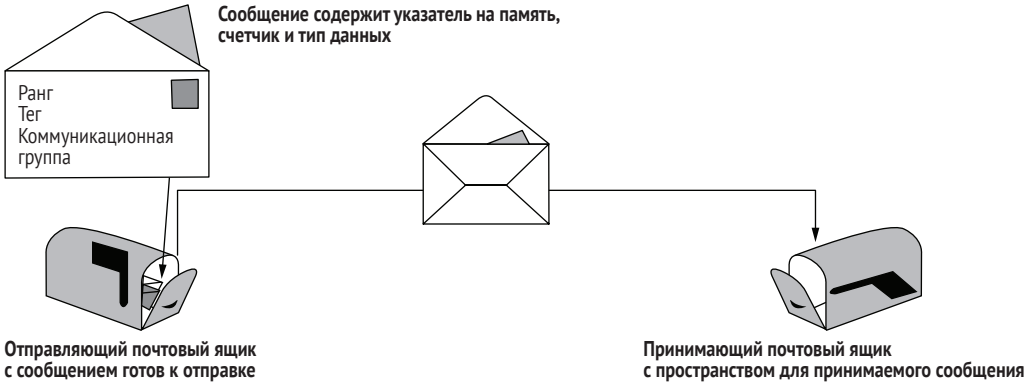


Рис. 8.2 Сообщение в MPI всегда состоит из указателя на память, счетчика и типа. Конверт содержит адрес, состоящий из ранга, тега и коммуникационной группы, а также внутреннего для MPI контекста

Само сообщение на обоих концах всегда состоит из триплета: указателя на буфер памяти, счетчика (count) и типа. Тип отправки и тип приемки могут быть разными типами и счетчиками (количествами). Осно-

вание для использования типов и счетчиков заключается в том, что оно допускает конверсию типов между процессами в исходном и конечном пунктах. Оно позволяет конвертировать сообщение в другую форму на принимающей стороне. В гетерогенной среде это может означать конвертирование правосторонней кодировки в левостороннюю[1], что означает низкоуровневую разницу в порядке следования байтов данных, хранящихся у разных поставщиков оборудования. Кроме того, размер приемки может быть больше объема отправки. Это позволяет приемщику запрашивать объем отправляемых данных, чтобы иметь возможность надлежаще манипулировать сообщением. Но размер принятого не может быть меньше размера отправленного, потому что это приведет к записи после окончания буфера.

Конверт тоже состоит из триплета. Он определяет отправителя сообщения, его получателя и идентификатор сообщения, не давая перепутывать несколько сообщений. Триплет состоит из ранга, тега и коммуникационной группы. Ранг назначается указанной коммуникационной группе. Тег помогает программисту и MPI определять само сообщение и приемку, к которой оно относится. В MPI тег служит ради удобства. Он может быть установлен равным `MPI_ANY_TAG`, если явный номер тега не требуется. MPI использует контекст, созданный внутри библиотеки, для правильного разделения сообщений. Сообщение является завершенным, если и коммуникатор, и тег совпадают.

> **ПРИМЕЧАНИЕ** Одной из сильных сторон подхода на основе передачи сообщений является модель памяти. Каждый процесс имеет четкое право собственности на свои данные, а также контроль и синхронизацию при изменении данных. Вам гарантируется, что какой-то другой процесс не сможет изменить вашу память, пока вы повернулись спиной.

Теперь давайте попробуем программу MPI с простой отправкой/приемкой. Мы должны отправлять данные на одном процессе и получать данные на другом. Эти вызовы на нескольких процессах можно делать разными способами (рис. 8.3). Некоторые комбинации базовых блокирующих отправок и приемок небезопасны и могут зависать, например две комбинации слева на рис. 8.3. Третья комбинация требует выверенного программирования с использованием условных блоков. Метод крайний справа – один из нескольких безопасных методов планирования обменов данными с использованием неблокирующих отправок и приемок. Они также называются *асинхронными* или *немедленными* вызовами, что объясняет появление символа *I* (от англ. *immediate*), предшествующего ключевым словам `send` и `receive` (этот случай показан в крайнем правом углу рисунка).

---

[1] То есть из кодировки little-endian, в которой наименее значимый бит находится справа, в кодировку big-endian, в которой наименее значимый бит находится слева. – *Прим. перев.*

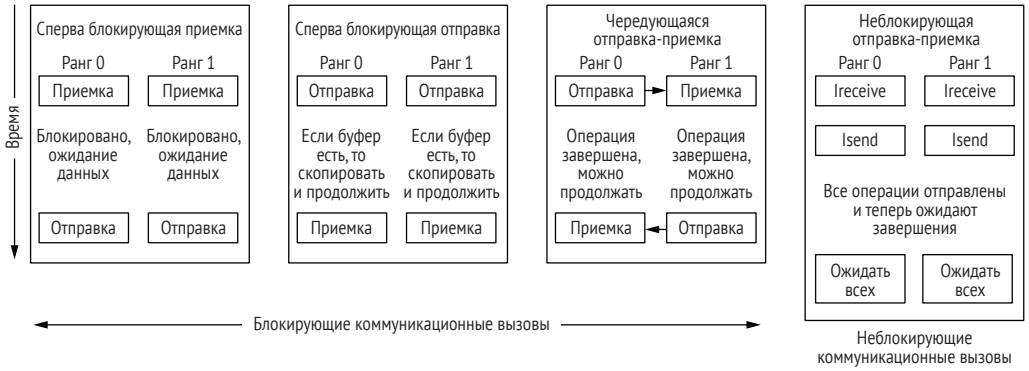


Рис. 8.3 Порядок блокирования отправки и приемки трудно выполнить правильно. Гораздо безопаснее и быстрее использовать неблокирующие или немедленные формы операций отправки и приемки, а затем ждать завершения

Самыми базовыми функциями отправки и приемки MPI являются `MPI_Send` и `MPI_Recv`. Базовые функции отправки и приемки имеют следующие прототипы:

```
MPI_Send(void *data, int count, MPI_Datatype datatype, int dest, int tag,
        MPI_COMM comm)
MPI_Recv(void *data, int count, MPI_Datatype datatype, int source, int tag,
        MPI_COMM comm, MPI_Status *status)
```

Теперь давайте пройдемся по каждому из четырех случаев на рис. 8.3, чтобы понять, почему некоторые зависают, а некоторые работают нормально. Мы начнем с `MPI_Send` и `MPI_Receive`, которые были показаны в предыдущих прототипах функций и в самом левом примере на рисунке. Обе эти процедуры являются *блокирующими*. Блокирование означает, что они не возвращаются до тех пор, пока не будет соблюдено конкретное условие. В случае этих двух вызовов условием возврата является то, что буфер можно безопасно использовать снова. При отправке буфер должен быть прочитан и больше не требоваться. При приемке буфер должен быть заполнен. Если оба процесса в сообщении являются блокирующими, то может возникнуть ситуация, именуемая *зависанием* (hang). Зависание происходит, когда один или несколько процессов ожидают события, которое никогда не сможет произойти.

Пример: зависающая программа блокирующей отправки/приемки

Этот пример подчеркивает распространенную в параллельном программировании проблему. Вы всегда должны быть начеку, чтобы избегать ситуации, которая может зависать (попасть в тупик – deadlock). Во избежание такой ситуации в следующем ниже листинге наглядно показано, как это может происходить.

Простой пример отправки/приемки в MPI (всегда виснет)

Send_Recv/SendRecv1.c

```
 1 #include <mpi.h>
 2 #include <stdio.h>
 3 #include <stdlib.h>
 4 int main(int argc, char **argv)
 5 {
 6    MPI_Init(&argc, &argv);
 7
 8    int count = 10;
 9    double xsend[count], xrecv[count];
10    for (int i=0; i<count; i++){
11       xsend[i] = (double)i;
12    }
13
14    int rank, nprocs;
15    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
16    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
17    if (nprocs%2 == 1){
18       if (rank == 0){
19          printf("Должна вызываться с четным числом процессов\n");
20       }
21       exit(1);
22    }
23
24    int tag = rank/2;
25    int partner_rank = (rank/2)*2 + (rank+1)%2;
26    MPI_Comm comm = MPI_COMM_WORLD;
27
28    MPI_Recv(xrecv, count, MPI_DOUBLE,
                partner_rank, tag, comm,
                MPI_STATUS_IGNORE);
29    MPI_Send(xsend, count, MPI_DOUBLE,
                partner_rank, tag, comm);
30
31    if (rank == 0) printf("SendRecv успешно завершена\n");
32
33    MPI_Finalize();
34    return 0;
35 }
```

Целочисленное деление спаривает теги для партнеров пары отправка–приемка

Ранг партнера – это противоположный член пары

Приемки отправляются первыми

Отправки делаются после приемок

Тег и ранг коммуникационного партнера вычисляются посредством целочисленной и модульной арифметики, которая спаривает теги для каждой отправки и приемки и получает ранг другого члена пары. Затем приемки отправляются для каждого процесса со своим партнером. Эти приемки являются блокирующими, которые не завершаются (не возвращаются) до тех пор, пока буфер не будет заполнен. Поскольку отправка вызывается только после завершения приемок, программа зависает. Обратите внимание, что мы написали вызовы отправки и приемки без инструкций `if` (условных блоков), основываясь на ранге. В параллельном коде условные блоки являются источником многих ошибок, поэтому их, как правило, следует избегать.

Давайте попробуем обратить вспять порядок отправки и получения. Мы приведем измененные строки кода из изначального листинга предыдущего примера в следующем ниже листинге.

Листинг 8.4 Простой пример отправки/приемки в MPI (иногда отказывает)

Send_Recv/SendRecv2.c

```
28    MPI_Send(xsend, count, MPI_DOUBLE,
                partner_rank, tag, comm);
29    MPI_Recv(xrecv, count, MPI_DOUBLE,
                partner_rank, tag, comm,
                MPI_STATUS_IGNORE);
```

Сперва вызывает операцию отправки

Затем вызывает операцию приемки после завершения операции отправки

Неужели этот пример откажет? Все зависит от обстоятельств. Вызов отправки возвращается после завершения использования буфера данных от отправки. В большинстве имплементаций MPI данные будут копироваться в предварительно выделенные буферы на отправителе либо приемщике, если размер достаточно мал. В этом случае отправка завершается, и вызывается приемка. Если сообщение велико, то отправитель ожидает до тех пор, пока вызов приемки вы выделит буфер для размещения сообщения, прежде чем вернуться. Но приемка никогда не вызывается, поэтому программа зависает. Мы могли бы чередовать отправки и приемки по рангам, чтобы не возникало зависаний. Для этого варианта мы должны использовать условный блок, как показано в следующем ниже листинге.

Листинг 8.5 Отправка/приемка с чередованием отправок и приемок по рангу

Send_Recv/SendRecv3.c

```
28    if (rank%2 == 0) {
29       MPI_Send(xsend, count, MPI_DOUBLE, partner_rank, tag, comm);
30       MPI_Recv(xrecv, count, MPI_DOUBLE, partner_rank, tag, comm,
                  MPI_STATUS_IGNORE);
31    } else {
32       MPI_Recv(xrecv, count, MPI_DOUBLE, partner_rank, tag, comm,
                  MPI_STATUS_IGNORE);
33       MPI_Send(xsend, count, MPI_DOUBLE, partner_rank, tag, comm);
34 }
```

Четные ранги относят отправку первыми

Нечетные ранги делают приемку первыми

Но в более сложном обмене данными это трудно сделать правильно и требует выверенного использования условных блоков. Более подходящий вариант имплементирования состоит в использовании вызова функции `MPI_Sendrecv`, как показано в следующем ниже листинге. При использовании этого вызова вы передаете ответственность за правильное исполнение обмена данными в библиотеку MPI. Для программиста такая сделка является довольно выгодной.

Листинг 8.6 Отправка/приемка с помощью вызова MPI_Sendrecv

Send_Recv/SendRecv4.c

```
28    MPI_Sendrecv(xsend, count, MPI_DOUBLE,
                   partner_rank, tag,
29                 xrecv, count, MPI_DOUBLE,
                   partner_rank, tag, comm,
                   MPI_STATUS_IGNORE);
```

Комбинированный вызов отправки/приемки заменяет отдельные вызовы MPI_Send и MPI_Recv

Вызов `MPI_Sendrecv` является хорошим примером преимуществ использования вызовов коллективного обмена данными, которые мы представим в разделе 8.3. Рекомендуется использовать вызовы коллективного обмена данными при любой возможности, поскольку они делегируют библиотеке MPI ответственность за предотвращение зависаний и тупиковых ситуаций, а также ответственность за хорошую производительность.

В качестве альтернативы блокирующим коммуникационным вызовам из предыдущих примеров мы рассмотрим использование `MPI_Isend` и `MPI_Irecv` в листинге 8.7. Они называются *немедленными* (*I*) версиями, потому что они возвращаются немедленно. Их нередко называют *асинхронными* или *неблокирующими* вызовами. Асинхронность означает, что вызов инициирует операцию, но не ждет завершения работы.

Листинг 8.7 Простой пример отправки/приемки с использованием Isend и Irecv

Send_Recv/SendRecv5.c

```
27    MPI_Request requests[2] =
         {MPI_REQUEST_NULL, MPI_REQUEST_NULL};
28
29    MPI_Irecv(xrecv, count, MPI_DOUBLE,
                partner_rank, tag, comm,
                &requests[0]);
30    MPI_Isend(xsend, count, MPI_DOUBLE,
                partner_rank, tag, comm,
                &requests[1]);
31    MPI_Waitall(2, requests, MPI_STATUSES_IGNORE);
```

Определяет массив запросов и устанавливает равным null, чтобы они определялись при проверке на завершение

Irecv отправляется первым

Затем Isend вызывается после завершения Irecv

Вызывает функцию Waitall для ожидания завершения работы отправки и получения

Каждый процесс ожидает завершения сообщения в `MPI_Waitall` в строке 31 листинга. Вы также должны увидеть ощутимое улучшение производительности программы за счет сведения числа мест, которые блокируют при каждом вызове отправки и приемки, до одного `MPI_Waitall`. Но вы должны быть осторожны в том, чтобы не модифицировать буфер отправки или не обращаться к буферу приемки до завершения операции. Есть и другие работающие комбинации. Давайте посмотрим на следующий ниже листинг, в котором используется одна из возможностей.

Листинг 8.8 Пример смешанной немедленной и блокирующей отправки/приемки

Send_Recv/SendRecv6.c

```
27    MPI_Request request;
28
29    MPI_Isend(xsend, count, MPI_DOUBLE,
                partner_rank, tag, comm,
                &request);
30    MPI_Recv(xrecv, count, MPI_DOUBLE,
               partner_rank, tag, comm,
               MPI_STATUS_IGNORE);
31    MPI_Request_free(&request);
```

Относит отправку с помощью MPI_Isend, чтобы она возвращалась

Вызывает блокирующую приемку. Этот процесс может продолжиться, как только он вернется

Высвобождает дескриптор запроса, чтобы избежать утечки памяти

Мы начинаем обмен данными с асинхронной отправки, а затем блокируем блокирующей приемкой. Как только блокирующая приемка завершится, этот процесс может продолжиться, даже если отправка не завершена. Вы все равно должны высвободить дескриптор запроса с помощью `MPI_Request_free` или в качестве побочного эффекта вызова `MPI_Wait` или `MPI_Test`. Это делается во избежание утечки памяти. Вы также можете вызвать `MPI_Request_free` сразу после отправки `MPI_Isend`.

В особых ситуациях бывают полезны и другие варианты отправки/приемки. Режимы обозначаются одно- либо двухбуквенным префиксом, аналогичным тому, который встречается в немедленном варианте, как указано ниже:

- `B` (буферизованный);
- `S` (синхронный);
- `R` (готовый);
- `IB` (немедленный буферизованный);
- `IS` (немедленный синхронный);
- `IR` (немедленный готовый).

Список предопределенных типов данных MPI для C обширен; типы данных соотносятся почти со всеми типами языка C. MPI также имеет типы, соответствующие типам данных Fortran. Мы перечислим только наиболее распространенные из них для C:

- `MPI_CHAR` (однобайтовый символьный тип C);
- `MPI_INT` (четырехбайтовый целочисленный тип);
- `MPI_FLOAT` (четырехбайтовый вещественный тип);
- `MPI_DOUBLE` (восьмибайтовый вещественный тип);
- `MPI_PACKED` (генерический байторазмерный тип данных, обычно используемый для смешанных типов данных);
- `MPI_BYTE` (генерический байторазмерный тип данных).

`MPI_PACKED` и `MPI_BYTE` являются специальными типами и соотносится с любым другим типом. `MPI_BYTE` выражает нетипизированное значение, а счетчик (count) выражает число байтов. Он пропускает любые

операции конверсии данных в гетерогенных системах передачи данных. `MPI_PACKED` используется с процедурой `MPI_PACK`, как показано в примере обмена данными призраков в разделе 8.4.3. Вы также можете определить свой собственный тип данных для использования в этих вызовах. Это тоже продемонстрировано в примере обмена данными призраков. Кроме того, есть немало процедур тестирования завершения обмена данными, которые включают в себя:

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
int MPI_Testany(int count, MPI_Request requests[], int *index, int *flag,
                MPI_Status *status)
int MPI_Testall(int count, MPI_Request requests[], int *flag,
                MPI_Status statuses[])
int MPI_Testsome(int incount, MPI_Request requests[], int *outcount,
                 int indices[], MPI_Status statuses[])
int MPI_Wait(MPI_Request *request, MPI_Status *status)
int MPI_Waitany(int count, MPI_Request requests[], int *index,
                MPI_Status *status)
int MPI_Waitall(int count, MPI_Request requests[], MPI_Status statuses[])
int MPI_Waitsome(int incount, MPI_Request requests[], int *outcount,
                 int indices[], MPI_Status statuses[])
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)
```

Здесь не перечислены дополнительные варианты `MPI_Probe`. Процедура `MPI_Waitall` показана в нескольких примерах этой главы. Другие процедуры полезны в более специализированных ситуациях. Название процедур дает хорошее представление о возможностях, которые они предоставляют.

8.3 Коллективный обмен данными: мощный компонент MPI

В этом разделе мы рассмотрим богатый набор вызовов коллективного обмена данными в MPI. Коллективные обмены данными оперируют группой процессов, содержащихся в коммуникаторе MPI. В целях оперирования на частичном наборе процессов вы можете создать свой собственный MPI-коммуникатор для подмножества `MPI_COMM_WORLD`, такого как любой другой процесс. Затем вместо `MPI_COMM_WORLD` вы можете применять свой коммуникатор в вызовах коллективного обмена данными. Большинство процедур коллективного обмена оперируют на данных. Рисунок 8.4 дает наглядное представление о том, что конкретно делает каждая коллективная операция.

Мы приведем примеры приемов использования наиболее часто встречающихся коллективных операций, поскольку они могут применяться в приложении. В первом примере (в разделе 8.3.1) показано использование барьера. Это единственная коллективная процедура, которая не работает с данными. Затем мы покажем несколько примеров с широко-

вещательной передачей (раздел 8.3.2), редукцией (раздел 8.3.3) и, наконец, операциями разброса/сбора (разделы 8.3.4 и 8.3.5). MPI также имеет много процедур из разряда «все для всех». Но они обходятся дорого и используются редко, поэтому здесь мы не будем их описывать. Все эти коллективные операции оперируют на группе процессов, представленных коммуникационной группой. Все члены коммуникационной группы должны вызывать коллектив, иначе ваша программа зависнет.

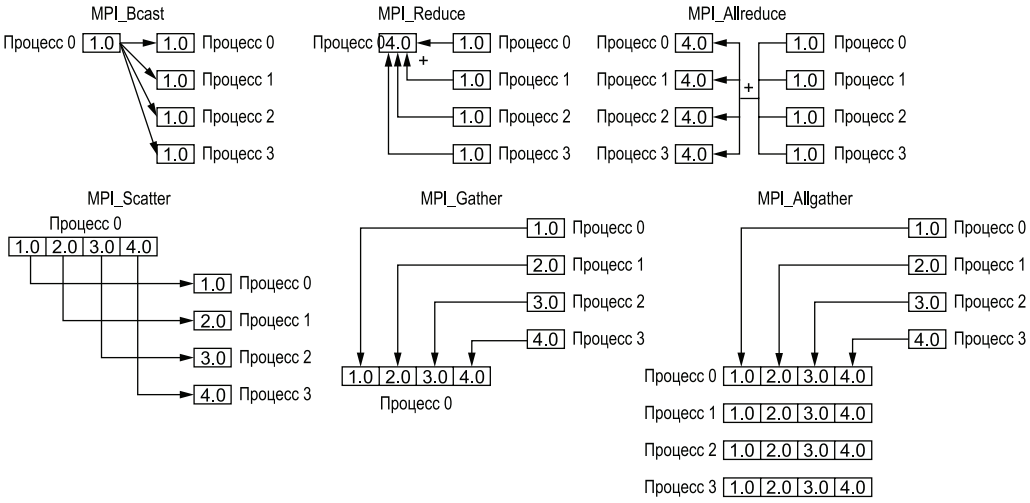


Рис. 8.4 Перемещение данных в рамках наиболее распространенных коллективных подпрограмм MPI обеспечивает важные функции для параллельных программ. Дополнительные варианты `MPI_Scatterv`, `MPI_Gatherv` и `MPI_Allgatherv` позволяют отправлять или принимать переменный объем данных из процессов. Не показаны некоторые дополнительные процедуры, такие как `MPI_Alltoall` и аналогичные функции

8.3.1 Использование барьера для синхронизирования таймеров

Самым простым вызовом коллективного обмена данными является `MPI_Barrier`. Он используется для синхронизирования всех процессов в коммуникаторе MPI. В большинстве программ в этом нет необходимости, но он нередко используется для отлаживания и синхронизирования таймеров. Давайте взглянем на использование `MPI_Barrier` для синхронизирования таймеров в следующем ниже листинге. Мы также применяем функцию `MPI_Wtime` для получения текущего времени.

Листинг 8.9 Использование MPI_Barrier для синхронизирования таймера в программе MPI

`SynchronizedTimer/SynchronizedTimer1.c`

```
1 #include <mpi.h>
2 #include <unistd.h>
```

```
 3 #include <stdio.h>
 4 int main(int argc, char *argv[])
 5 {
 6    double start_time, main_time;
 7
 8    MPI_Init(&argc, &argv);
 9    int rank;
10    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11
12    MPI_Barrier(MPI_COMM_WORLD);
13    start_time = MPI_Wtime();
14
15    sleep(30);
16
17    MPI_Barrier(MPI_COMM_WORLD);
18    main_time = MPI_Wtime() - start_time;
19    if (rank == 0) printf("Время работы составляет %lf секунд(ы)\n", main_time);
20
21    MPI_Finalize();
22    return 0;
23 }
```

Синхронизирует все процессы, чтобы они начинались примерно в одно и то же время

Получает начальное значение таймера с помощью процедуры MPI_Wtime

Представляет выполняемую работу

Синхронизирует процессы, чтобы получить как можно более продолжительное время

Получает значение таймера и вычитает начальное значение, чтобы получить истекшее время

Барьер вставляется перед запуском таймера, а затем непосредственно перед остановкой таймера. Это побуждает таймеры всех процессов запускаться примерно в одно и то же время. Вставляя барьер перед остановкой таймера, мы получаем максимальное время по всем процессам. Иногда использование синхронизированного таймера дает менее запутанную меру времени, но в других случаях лучше использовать несинхронизированный таймер.

ПРИМЕЧАНИЕ Синхронизированные таймеры и барьеры не следует использовать в производственных циклах; это может привести к серьезному замедлению работы приложения.

8.3.2 Использование широковещательной передачи для манипулирования данными малого входного файла

Широковещательная передача (или транслирование) посылает данные от одного процессора всем остальным. Эта операция показана на рис. 8.4 в левом верхнем углу. Одно из применений широковещательной передачи, `MPI_Bcast`, заключается в отправке значений, прочитанных из входного файла, всем другим процессам. Если каждый процесс пытается открыть файл при большом числе процессов, то для завершения открытия файла может потребоваться несколько минут. Это связано с тем, что файловые системы по своей сути являются последовательными и являются одним из самых медленных компонентов компьютерной системы. По этим причинам для малого входного файла рекомендуется открывать и читать файл только из одного процесса. В следующем ниже листинге показано, как это делается.

Листинг 8.10 Использование MPI_Bcast для манипулирования данными малого входного файла

FileRead/FileRead.c

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <stdlib.h>
 4 #include <mpi.h>
 5 int main(int argc, char *argv[])
 6 {
 7    int rank, input_size;
 8    char *input_string, *line;
 9    FILE *fin;
10
11    MPI_Init(&argc, &argv);
12    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
13
14    if (rank == 0){
15       fin = fopen("file.in", "r");
16       fseek(fin, 0, SEEK_END);
17       input_size = ftell(fin);
18       fseek(fin, 0, SEEK_SET);
19       input_string = (char *)malloc((input_size+1)*sizeof(char));
20       fread(input_string, 1, input_size, fin);
21       input_string[input_size] = '\0';
22    }
23
24    MPI_Bcast(&input_size, 1, MPI_INT, 0,
              MPI_COMM_WORLD);
25    if (rank != 0)
         input_string =
            (char *)malloc((input_size+1)*
                           sizeof(char));
26    MPI_Bcast(input_string, input_size,
              MPI_CHAR, 0, MPI_COMM_WORLD);
27
28    if (rank == 0) fclose(fin);
29
30    line = strtok(input_string,"\n");
31    while (line != NULL){
32       printf("%d:входной литерал таков: %s\n",rank,line);
33       line = strtok(NULL,"\n");
34    }
35    free(input_string);
36
37    MPI_Finalize();
38    return 0;
39 }
```

Возвращает размер файла для выделения входного буфера

Сбрасывает указатель файла в начало файла

Читает весь файл

Null-терминирующийся входной буфер

Транслирует размер входного буфера

Выделяет входной буфер для других процессов

Вещает входной буфер

Лучше передавать большие куски данных, чем передавать много малых отдельных значений. Поэтому мы вещаем весь файл целиком. Для

этого нам нужно сначала широковещательно передать размер, чтобы каждый процесс мог выделить входной буфер, а затем широковещательно передать данные. Чтение и вещание файлов выполняются из ранга 0, обычно именуемого главным процессом.

`MPI_Bcast` берет указатель для первого аргумента, поэтому при отправке скалярной переменной мы отправляем ссылку с помощью амперсандного (&) оператора, чтобы получить адрес переменной. Затем идет счетчик (count) и тип, чтобы полностью определить отправляемые данные. Следующий аргумент указывает исходный процесс. В обоих этих вызовах он равен 0, потому что это ранг, в котором хранятся данные. После этого все остальные процессы в обмене `MPI_COMM_WORLD` получают данные. Этот метод предназначен для малых входных файлов. Для ввода или вывода более крупных файлов существуют способы выполнения параллельных файловых операций. Сложный мир параллельного ввода и вывода обсуждается в главе 16.

8.3.3 Использование редукции для получения одного единственного значения из всех процессов

Обсуждаемый в разделе 5.7 редукционный шаблон является одним из наиболее важных шаблонов параллельных вычислений. Операция редукции показана на рис. 8.4 вверху в середине. Примером редукции в синтаксисе Fortran для массивов является `xsum = sum(x(:))`, где внутренняя `sum` языка Fortran суммирует массив `x` и помещает его в скалярную переменную `xsum`. Вызовы редукции MPI принимают массив или многомерный массив и сводят значения к скалярному результату. Во время редукции может делаться много операций. Наиболее распространенные из них таковы:

- `MPI_MAX` (максимальное значение в массиве);
- `MPI_MIN` (минимальное значение в массиве);
- `MPI_SUM` (сумма массива);
- `MPI_MINLOC` (индекс минимального значения);
- `MPI_MAXLOC` (индекс максимального значения).

В следующем ниже листинге показано применение `MPI_Reduce` для получения минимального, максимального и среднего значения переменной для каждого процесса.

Листинг 8.11 Использование редукций для получения минимального, максимального и среднего результатов таймера

`SynchronizedTimer/SynchronizedTimer2.c`

```
1 #include <mpi.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 int main(int argc, char *argv[])
5 {
```

```
 6    double start_time, main_time, min_time, max_time, avg_time;
 7
 8    MPI_Init(&argc, &argv);
 9    int rank, nprocs;
10    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
12
13    MPI_Barrier(MPI_COMM_WORLD);
14    start_time = MPI_Wtime();
15
16    sleep(30);
17
18    main_time = MPI_Wtime() - start_time;
19    MPI_Reduce(&main_time, &max_time, 1,
                 MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
20    MPI_Reduce(&main_time, &min_time, 1,
                 MPI_DOUBLE, MPI_MIN, 0,MPI_COMM_WORLD);
21    MPI_Reduce(&main_time, &avg_time, 1,
                 MPI_DOUBLE, MPI_SUM, 0,MPI_COMM_WORLD);
22    if (rank == 0)
         printf("Время работы составляет мин.: %lf макс.: %lf сред.: %lf секунд(ы)\n",
23             min_time, max_time, avg_time/nprocs);
24
25    MPI_Finalize();
26    return 0;
27 }
```

Синхронизирует все процессы, чтобы они начинались примерно в одно и то же время

Представляет выполняемую работу

Получает значение таймера и вычитает начальное значение, чтобы получить истекшее время

Использует вызовы редукции для вычисления максимального, минимального и среднего времени

Результат редукции, в данном случае максимум, сохраняется в ранге 0 (аргумент 6 в вызове `MPI_Reduce`), т. е. в данном случае главном процессе. Если бы мы хотели просто распечатать его в главном процессе, то это было бы уместно. Но если бы мы хотели, чтобы все процессы имели значение, то мы бы использовали процедуру `MPI_Allreduce`.

Кроме того, можно определить свой собственный оператор. Мы будем использовать пример суммирования по Кахану повышенной прецизионности, с которым мы уже работали и впервые ввели в разделе 5.7. Задача в параллельной среде с распределенной памятью состоит в том, чтобы выполнить суммирование по Кахану по всем процессным рангам. Мы начнем с того, что в следующем ниже листинге обратимся к главной программе, а потом перейдем к двум другим частям программы в листингах 8.13 и 8.14.

Листинг 8.12 MPI-версия суммирования по Кахану

`GlobalSums/globalsums.c`

```
57 int main(int argc, char *argv[])
58 {
59    MPI_Init(&argc, &argv);
```

```
60    int rank, nprocs;
61    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
62    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
63
64    init_kahan_sum();
65
66    if (rank == 0) printf("MPI Kahan tests\n");
67
68    for (int pow_of_two = 8; pow_of_two < 31; pow_of_two++){
69       long ncells = (long)pow((double)2,(double)pow_of_two);
70
71       int nsize;
72       double accurate_sum;
73       double *local_energy =
            init_energy(ncells, &nsize,
            &accurate_sum);
74
75       struct timespec cpu_timer;
76       cpu_timer_start(&cpu_timer);
77
78       double test_sum =
            global_kahan_sum(nsize, local_energy);
79
80       double cpu_time = cpu_timer_stop(cpu_timer);
81
82       if (rank == 0){
83          double sum_diff = test_sum-accurate_sum;
84          printf("n-ячеек %ld лог %d точн.сум %-17.16lg сум %-17.16lg ",
85                 ncells,(int)log2((double)ncells),accurate_sum,test_sum);
86          printf("разн %10.4lg относительная разн %10.4lg время работы %lf\n",
87                 sum_diff,sum_diff/accurate_sum, cpu_time);
88       }
89
90       free(local_energy);
91    }
92
93    MPI_Type_free(&EPSUM_TWO_DOUBLES);
94    MPI_Op_free(&KAHAN_SUM);
95    MPI_Finalize();
96    return 0;
97 }
```

Инициализирует новый тип данных MPI и создает новый оператор

Получает распределенный массив для работы

Вычисляет суммирование по Кахану массива энергии по всем процессам

Главная программа (main) показывает, что новый тип данных MPI создается один раз в начале программы и высвобождается в конце, перед `MPI_Finalize`. Вызов для выполнения глобального суммирования по Кахану выполняется несколько раз в цикле, где размер данных увеличивается в два раза. Теперь давайте посмотрим на следующий ниже листинг, чтобы увидеть, что нужно сделать для инициализирования нового типа данных и оператора.

Листинг 8.13 Инициализирование нового типа данных MPI и оператора для суммирования по Кахану

GlobalSums/globalsums.c

```
14 struct esum_type{
15    double sum;
16    double correction;
17 };
18
19 MPI_Datatype EPSUM_TWO_DOUBLES;
20 MPI_Op KAHAN_SUM;
21
22 void kahan_sum(struct esum_type * in,
                 struct esum_type * inout, int *len,
23     MPI_Datatype *EPSUM_TWO_DOUBLES)
24 {
25    double corrected_next_term, new_sum;
26    corrected_next_term = in->sum + (in->correction + inout->correction);
27    new_sum = inout->sum + corrected_next_term;
28    inout->correction = corrected_next_term - (new_sum - inout->sum);
29    inout->sum = new_sum;
30 }
31
32 void init_kahan_sum(void){
33    MPI_Type_contiguous(2, MPI_DOUBLE,
                          &EPSUM_TWO_DOUBLES);
34    MPI_Type_commit(&EPSUM_TWO_DOUBLES);
35
36    int commutative = 1;
37    MPI_Op_create((MPI_User_function *)kahan_sum,
                    commutative, &KAHAN_SUM);
38 }
```

Определяет структуру esum_type для хранения суммы и поправочного члена

Объявляет новый тип данных MPI, состоящий из двух чисел двойной точности

Объявляет новый оператор суммирования по Кахану

Определяет функцию для нового оператора, используя предопределенную сигнатуру

Создает тип и фиксирует (коммитит) его

Создает новый оператор и фиксирует его

Сначала мы создаем новый тип данных `EPSUM_TWO_DOUBLES`, объединив два базовых типа данных `MPI_DOUBLE` в строке 33. Мы должны объявить тип вне процедуры в строке 19, чтобы он был доступен для использования процедурой суммирования. В целях создания нового оператора мы сначала пишем функцию для использования в качестве оператора в строках 22–30. Затем мы используем `esum_type` для передачи обоих значений типа double туда и обратно. Нам также нужно передать длину и тип данных, на которых он будет оперировать, в качестве нового типа `EPSUM_TWO_DOUBLES`.

В ходе создания оператора редукции суммы по Кахану мы показали вам, как создавать новый тип данных MPI и новый редукционный оператор MPI. Теперь давайте перейдем к фактическому вычислению глобальной суммы массива по всем рангам MPI, как показано в следующем ниже листинге.

Листинг 8.14 Выполнение MPI-суммирования по Кахану

GlobalSums/globalsums.c

```
40 double global_kahan_sum(int nsize, double *local_energy){
41    struct esum_type local, global;
42    local.sum = 0.0;                  Инициализирует оба элемента
43    local.correction = 0.0;           esum_type нулями
44
45    for (long i = 0; i < nsize; i++) {
46       double corrected_next_term =
            local_energy[i] + local.correction;
47       double new_sum =
            local.sum + local.correction;
48       local.correction = corrected_next_term -
                           (new_sum - local.sum);
49       local.sum = new_sum;
50    }
51
52    MPI_Allreduce(&local, &global, 1, EPSUM_TWO_DOUBLES, KAHAN_SUM,
                    MPI_COMM_WORLD);
53
54    return global.sum;
55 }
```

Выполняет напроцессное суммирование по Кахану

Выполняет редукцию с помощью нового оператора KAHAN_SUM

Теперь вычислять глобальное суммирование по Кахану стало относительно просто. Мы можем брать локальную сумму по Кахану, как показано в разделе 5.7. Но мы должны добавлять `MPI_Allreduce` в строке 52, чтобы получать глобальный результат. Здесь мы определили, что операция `allreduce` завершится результатом на всех процессорах, как показано на рис. 8.4 в правом верхнем углу.

8.3.4 Использование операции сбора для наведения порядка в отладочных распечатках

Операцию сбора можно описать как операцию сортировки, при которой данные от всех процессоров сводятся вместе и складываются в единый массив, как показано на рис. 8.4 внизу в центре. Этот вызов коллективного обмена данными может использоваться, для того чтобы наводить порядок в выводимой на консоль информации из вашей программы. К настоящему времени вы должны были заметить, что результат, распечатываемый из нескольких рангов программы MPI, выходит в случайном порядке, создавая беспорядочную, запутанную мешанину. Давайте рассмотрим более качественный подход к урегулированию этой ситуации, чтобы получать единственный результат из главного процесса. Если печатать результат только главного процесса, то порядок будет правильным. В следующем ниже листинге показан пример программы, которая получает данные от всех процессов и распечатывает их в удобном, упорядоченном виде.

Листинг 8.15 Использование операции сбора для печати отладочных сообщений

DebugPrintout/DebugPrintout.c

```
 1 #include <stdio.h>
 2 #include <time.h>
 3 #include <unistd.h>
 4 #include <mpi.h>
 5 #include "timer.h"
 6 int main(int argc, char *argv[])
 7 {
 8    int rank, nprocs;
 9    double total_time;
10    struct timespec tstart_time;
11
12    MPI_Init(&argc, &argv);
13    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
14    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
15
16    cpu_timer_start(&tstart_time);
17    sleep(30);
18    total_time += cpu_timer_stop(tstart_time);
19
20    double times[nprocs];
21    MPI_Gather(&total_time, 1, MPI_DOUBLE,
          times, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
22    if (rank == 0) {
23       for (int i=0; i<nprocs; i++){
24          printf("%d:Work took %lf secs\n",
                 i, times[i]);
25       }
26    }
27
28    MPI_Finalize();
29    return 0;
30 }
```

Получает уникальные значения на каждом процессе в нашем примере

Нужен массив для сбора всех времен

Использует операцию сбора, чтобы свести все значения к нулевому процессу

Печатает только на главном процессе

Перебирает все процессы в цикле для распечатки

Печатает время каждого процесса

`MPI_Gather` принимает стандартный триплет, описывающий источник данных. Нам нужно использовать амперсанд, чтобы получить адрес скалярной переменной `total_time`. Целевой (конечный) пункт тоже является триплетом с целевым массивом `times`. Массив уже является адресом, поэтому амперсанд не требуется. Сбор выполняется для процесса 0 всеобщей коммуникационной группы `MPI_COMM_WORLD`. Оттуда требуется цикл для печати времени каждого процесса. В начало каждого строкового литерала мы дополняем номер в формате `#:`, чтобы было ясно, к какому процессу результат относится.

8.3.5 Использование разброса и сбора для отправки данных процессам для работы

Операция разброса, показанная на рис. 8.4 в левом нижнем углу, противоположна операции сбора. В этой операции данные передаются из одного процесса всем другим в коммуникационной группе. Наиболее распространенное применение операции разброса заключается в параллельной стратегии распределения массивов данных между другими процессами для работы. Это обеспечивается подпрограммами `MPI_Scatter` и `MPI_Scatterv`. В следующем ниже листинге показана имплементация.

Листинг 8.16 Использование разброса для распространения данных и их сбора для возврата

ScatterGather/ScatterGather.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <mpi.h>
 4 int main(int argc, char *argv[])
 5 {
 6    int rank, nprocs, ncells = 100000;
 7
 8    MPI_Init(&argc, &argv);
 9    MPI_Comm comm = MPI_COMM_WORLD;
10    MPI_Comm_rank(comm, &rank);
11    MPI_Comm_size(comm, &nprocs);
12
13    long ibegin = ncells *(rank )/nprocs;
14    long iend = ncells *(rank+1)/nprocs;
15    int nsize = (int)(iend-ibegin);
16
17    double *a_global, *a_test;
18    if (rank == 0) {
19       a_global = (double *)
            malloc(ncells*sizeof(double));
20       for (int i=0; i<ncells; i++) {
21          a_global[i] = (double)i;
22       }
23    }
24
25    int nsizes[nprocs], offsets[nprocs];
26    MPI_Allgather(&nsize, 1, MPI_INT, nsizes,
                    1, MPI_INT, comm);
27    offsets[0] = 0;
28    for (int i = 1; i<nprocs; i++){
29       offsets[i] = offsets[i-1] + nsizes[i-1];
30    }
31
```

Вычисляет размер массива для каждого процесса

Настраивает данные о главном процессе

Возвращает размеры и смещения в глобальные массивы для связи

```
32    double *a = (double *)
         malloc(nsize*sizeof(double));
33    MPI_Scatterv(a_global, nsizes, offsets,
34    MPI_DOUBLE, a, nsize, MPI_DOUBLE, 0, comm);
35
36    for (int i=0; i<nsize; i++){
37       a[i] += 1.0;
38    }
39
40    if (rank == 0) {
41       a_test = (double *)
            malloc(ncells*sizeof(double));
42    }
43
44    MPI_Gatherv(a, nsize, MPI_DOUBLE,
45                a_test, nsizes, offsets,
                  MPI_DOUBLE, 0, comm);
46
47    if (rank == 0){
48       int ierror = 0;
49       for (int i=0; i<ncells; i++){
50          if (a_test[i] != a_global[i] + 1.0) {
51             printf("Ошибка: индекс %d a_test %lf a_global %lf\n",
52                    i,a_test[i],a_global[i]);
53             ierror++;
54          }
55       }
56       printf("Отчет: Правильные результаты %d ошибки %d\n",
                ncells-ierror,ierror);
57    }
58
59    free(a);
60    if (rank == 0) {
61       free(a_global);
62       free(a_test);
63    }
64
65    MPI_Finalize();
66    return 0;
67 }
```

Распределяет данные по другим процессам

Выполняет вычисление

Возвращает данные массива в главный процесс, возможно, для вывода

Сначала нам нужно рассчитать размер данных по каждому процессу. Желаемое распределение должно быть как можно более равным. Простой способ вычисления размера показан в строках 13–15 с использованием простой целочисленной арифметики. Теперь нам нужен глобальный массив, но он нужен нам только для главного процесса. Поэтому мы выделяем и настраиваем его на этом процессе в строках 18–23. Для распространения или сбора данных должны быть известны размеры и сдвиги для всех процессов. Мы видим типичный расчет этих данных в строках 25–30. Фактический разброс выполняется с помощью `MPI_Scatterv` в строках 32–34. Источник данных описывается аргументами

buffer, counts, offsets и типом данных. Конечный пункт обрабатывается стандартным триплетом. Затем ранг источника, который будет отправлять данные, указывается как ранг 0. Наконец, последним аргументом является comm, коммуникационная группа, которая будет принимать данные.

MPI_Gatherv выполняет противоположную операцию, как показано на рис. 8.4. Нам нужен только глобальный массив для главного процесса, поэтому он выделяется там только в строках 40–42. Аргументы для MPI_Gatherv начинаются с описания источника стандартным триплетом. Затем целевой (конечный) пункт описывается теми же четырьмя аргументами, которые использовались в разбросе. Следующим аргументом является целевой ранг, за которым следует коммуникационная группа.

Следует отметить, что все размеры и сдвиги, используемые в вызове MPI_Gatherv, имеют целочисленный тип. Это лимитирует размер манипулируемых данных. Была предпринята попытка поменять тип данных на long, чтобы в версии 3 стандарта MPI иметь возможность манипулировать данными более крупного размера. Он не был одобрен, потому что нарушил бы работу слишком большого числа приложений. Следите за добавлением новых вызовов, которые обеспечат поддержку длинного целочисленного типа в одном из следующих стандартов MPI.

8.4 Примеры параллельности данных

Стратегия параллельности данных, определенная в разделе 1.5, является наиболее распространенным подходом в параллельных приложениях. В этом разделе мы рассмотрим несколько примеров такого подхода. Сначала рассмотрим простой случай потоковой триады, когда нет необходимости в обмене данными. Затем – более типичные технические приемы обмена данными призрачных ячеек, используемых для связывания подразделенных доменов, распределенных каждому процессу.

8.4.1 Потоковая триада для измерения пропускной способности на узле

STREAM Triad – это исходный код сравнительного тестирования пропускной способности (ширины полосы), представленный в разделе 3.2.4. В этой версии используется MPI, чтобы обеспечить работу большего числа процессов на узле и, возможно, на нескольких узлах. Цель увеличения числа процессов состоит в том, чтобы увидеть максимальную пропускную способность узла при использовании всех процессоров. Она дает целевую пропускную способность для более сложных приложений. Как показано в листинге 8.17, исходный код является простым, поскольку обмена данными между рангами не требуется. Хронометраж отслеживается только на главном процессе. Его можно выполнить сначала на одном

процессоре, а затем на всех процессорах на вашем узле. Получаете ли вы полное параллельное ускорение, которое вы ожидали бы от увеличения числа процессоров? Насколько пропускная способность системной памяти лимитирует ваше ускорение?

Листинг 8.17 Версия MPI исходного кода STREAM Triad

StreamTriad/StreamTriad.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <time.h>
 4 #include <mpi.h>
 5 #include "timer.h"
 6
 7 #define NTIMES 16
 8 #define STREAM_ARRAY_SIZE 80000000
 9
10 int main(int argc, char *argv[]){
11
12    MPI_Init(&argc, &argv);
13
14    int nprocs, rank;
15    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
16    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
17    int ibegin = STREAM_ARRAY_SIZE *(rank )/nprocs;
18    int iend = STREAM_ARRAY_SIZE *(rank+1)/nprocs;
19    int nsize = iend-ibegin;
20    double *a = malloc(nsize * sizeof(double));
21    double *b = malloc(nsize * sizeof(double));
22    double *c = malloc(nsize * sizeof(double));
23
24    struct timespec tstart;
25    double scalar = 3.0, time_sum = 0.0;
26    for (int i=0; i<nsize; i++) {
27       a[i] = 1.0;
28       b[i] = 2.0;
29    }
30
31    for (int k=0; k<NTIMES; k++){
32       cpu_timer_start(&tstart);
33       for (int i=0; i<nsize; i++){
34          c[i] = a[i] + scalar*b[i];
35       }
36       time_sum += cpu_timer_stop(tstart);
37       c[1]=c[2];
38    }
39
40    free(a);
41    free(b);
42    free(c);
43
```

Достаточно крупный, чтобы втиснуть в основную память

Инициализирует данные и массивы

Цикл потоковой триады

Не позволяет компилятору оптимизировать цикл

```
44    if (rank == 0)
         printf("Среднее время выполнение составляет %lf мсек\n", time_sum/NTIMES);
45    MPI_Finalize();
46    return(0);
47 }
```

8.4.2 Обмен с призрачными ячейками в двухмерной вычислительной сетке

Призрачные ячейки – это механизм, который мы используем для связывания вычислительных сеток на смежных процессорах. Они используются для кеширования значений из смежных процессоров, вследствие чего требуется меньше обменов данными. Техника призрачных ячеек является единственным наиболее важным методом обеспечения параллелизма распределенной памяти в MPI.

Давайте немного остановимся на терминах «ореол» и «призрачная ячейка». Еще до эпохи параллельной обработки участок ячеек вокруг сетки часто использовался для имплементирования граничных условий. Эти граничные условия могут быть отражающими, входящими, исходящими или периодическими. В целях повышения эффективности программисты хотели избегать инструкций if в главным вычислительном цикле. Для этого они добавляли ячейки вокруг сетки и устанавливали для них надлежащие значения перед главным вычислительным циклом. Эти клетки имели вид ореола, отсюда и прижилось название. *Ореольные ячейки* – это любой набор ячеек, окружающих вычислительную сетку, независимо от их предназначения. Тогда *ореол с доменной границей* – это ореольные ячейки, используемые для наложения конкретного набора граничных условий.

После параллелизации приложений добавлялся аналогичный внешний участок ячеек для хранения значений из соседних ячеек. Эти ячейки не являются реальными ячейками, а существуют только для сокращения затрат на обмен данными. Поскольку они не настоящие, они вскоре получили название «призрачные ячейки». Реальные данные призрачных ячеек находятся на смежном процессоре, а локальная копия – это просто значение призрака. Призрачные ячейки тоже выглядят как ореолы и тоже называются *ореольными ячейками*. *Обновления* или *обмены данными в призрачных ячейках* относятся к обновлению призрачных ячеек и необходимы только для параллельных мультипроцессных прогонов, когда вам нужны обновления реальных значений из смежных процессов.

Граничные условия должны выполняться как для последовательных, так и для параллельных прогонов. Путаница связана с тем, что эти операции часто именуются *обновлениями ореола*, хотя неясно, что именно имеется в виду. В нашей терминологии *обновления ореола* относятся как к обновлениям доменных границ, так и к обновлениям призрачных ячеек. Для оптимизирования обмена данными в рамках MPI нам нужно только смотреть на обновления или обмены призрачными ячейками и пока откладывать в сторону расчеты граничных условий.

Давайте теперь рассмотрим настройку призрачных ячеек для границ локальной сетки в каждом процессе и выполнение обмена между поддоменами. При использовании призрачных ячеек необходимые сообщения группируются в меньшее число сообщений, чем если бы одно сообщение выполнялось каждый раз, когда значение ячейки требуется из другого процесса. Этот метод наиболее распространен и позволяет эффективно использовать параллельный подход к данным. В имплементациях обновлений призрачных ячеек мы продемонстрируем использование процедуры `MPI_Pack` и загрузим коммуникационный буфер с помощью простого поячеечного присваивания в массиве. В последующих разделах мы также обратимся к выполнению того же обмена с применением типов данных MPI, используя вызовы топологии MPI для настройки и обмена данными.

Если имплементировать обновления призрачных ячеек в исходном коде обеспечения параллельности данных, то будет обрабатываться большая часть необходимого обмена. За счет этого исходный код, который обеспечивает параллелизм, изолируется в малый раздел приложения. Этот малый раздел кода важен для оптимизирования под параллельную эффективность. Давайте рассмотрим некоторые имплементации этой функциональности, начиная с настройки в листинге 8.18 и работы, выполняемой стенсильными циклами в листинге 8.19. Возможно, вы захотите просмотреть полный исходный код примера кода данной главы в каталоге GhostExchange/GhostExchange_Pack по адресу https://github.com/EssentialsOfParallelComputing/Chapter8.

Листинг 8.18 Настройка для обмена с призрачными ячейками в двухмерной сетке

Входные настройки: -i <imax> -j <jmax> – это размеры сетки

-x <nprocx> -y <nprocy> – это число процессов в направлениях x и y

GhostExchange/GhostExchange_Pack/GhostExchange.cc

```
30    int imax = 2000, jmax = 2000;
31    int nprocx = 0, nprocy = 0;
32    int nhalo = 2, corners = 0;
33    int do_timing;
      ....
40    int xcoord = rank%nprocx;
41    int ycoord = rank/nprocx;
42
43    int nleft = (xcoord > 0 ) ?
                  rank - 1 : MPI_PROC_NULL;
44    int nrght = (xcoord < nprocx-1) ?
                  rank + 1 : MPI_PROC_NULL;
45    int nbot  = (ycoord > 0 ) ?
                  rank - nprocx : MPI_PROC_NULL;
46    int ntop  = (ycoord < nprocy-1) ?
                  rank + nprocx : MPI_PROC_NULL;
47
```

-h <nhalo> -c – это число ореольных ячеек, а -c включает угловые ячейки

do_timing синхронизирует хронометраж

Координаты xcoord и ycoord процессов. Индекс строки варьируется быстрее всего

Ранг соседа по каждому процессу для обмена данными с соседом

```
48    int ibegin = imax *(xcoord )/nprocx;
49    int iend = imax *(xcoord+1)/nprocx;
50    int isize = iend - ibegin;
51    int jbegin = jmax *(ycoord )/nprocy;
52    int jend = jmax *(ycoord+1)/nprocy;
53    int jsize = jend - jbegin;
```

Размер вычислительного домена для каждого процесса и глобальный начальный и конечный индекс

Мы выделяем память для локального размера плюс место для ореолов в каждом процессе. Для того чтобы сделать индексацию немного проще, мы сдвигаем индексацию памяти, чтобы она начиналась с `-nhalo` и заканчивалась в `isize+nhalo`. Тогда реальные ячейки всегда будут иметь размер от `0` до `isize-1` независимо от ширины ореола.

В следующих ниже строках показан вызов специальной malloc2D с двумя дополнительными аргументами, которые сдвигают адресацию массива так, чтобы реальная часть массива была от `0,0` до `jsize,isize`. Это делается с помощью некоторой указательной арифметики, которая перемещает начальное местоположение каждого указателя.

```
64    double** x = malloc2D(jsize+2*nhalo, isize+2*nhalo, nhalo, nhalo);
65    double** xnew = malloc2D(jsize+2*nhalo, isize+2*nhalo, nhalo, nhalo);
```

Для предоставления работы мы используем простую стенсильную калькуляцию из оператора размытия, представленного на рис. 1.10. Многие приложения имеют гораздо более сложные вычисления, которые занимают гораздо больше времени. В следующем ниже листинге показаны циклы стенсильной калькуляции.

Листинг 8.19 Работа выполняется в стенсильной итерации цикла

GhostExchange/GhostExchange_Pack/GhostExchange.cc

```
91    for (int iter = 0; iter < 1000; iter++){
92       cpu_timer_start(&tstart_stencil);
93
94       for (int j = 0; j < jsize; j++){
95          for (int i = 0; i < isize; i++){
96             xnew[j][i]=
                  (x[j][i] + x[j][i-1] + x[j][i+1] +
                   x[j-1][i] + x[j+1][i])/5.0;
97          }
98       }
99
100      SWAP_PTR(xnew, x, xtmp);
101
102      stencil_time += cpu_timer_stop(tstart_stencil);
103
104      boundarycondition_update(x, nhalo, jsize,
          isize, nleft, nrght, nbot, ntop);
105      ghostcell_update(x, nhalo, corners,
          jsize, isize, nleft, nrght, nbot, ntop);
106   }
```

Итерационный цикл

Стенсильное вычисление

Обмен указателями для старого и нового массивов x

Вызов обновления призрачных ячеек обновляет призрачные ячейки

Итерационный цикл

Теперь мы можем взглянуть на критически важный исходный код обновления призрачных ячеек. На рис. 8.5 показана требуемая операция. Ширина участка призрачных ячеек может составлять одну, две или более ячеек в глубину. Для некоторых применений также могут потребоваться угловые ячейки. Каждый из четырех процессов (или рангов) нуждается в данных из этого ранга; слева, справа, сверху и снизу. Каждый из этих процессов требует отдельного обмена данными и отдельного буфера данных. Ширина ореольного участка зависит от применения, а также от потребности в угловых ячейках.

На рис. 8.5 показан пример обмена с призрачными ячейками для сетки 4×4 на девяти процессах с ореолом шириной в одну ячейку и углами. Сначала обновляются ореолы внешних границ, а затем происходит горизонтальный обмен данными, синхронизация и вертикальный обмен данными. Если углы не нужны, то горизонтальный и вертикальный обмен можно выполнять одновременно. Если требуются углы, то необходима синхронизация между горизонтальным и вертикальным обменами.

Ключевым наблюдением при обновлении данных призрачных ячеек является то, что в C построчные данные являются сплошными, тогда как постолбцовые данные разделены шагом, равным размеру строки. Отправка отдельных значений для столбцов обходится дорого, и поэтому нам нужно как-то их группировать вместе.

Обновлять призрачные ячейки с помощью MPI можно несколькими способами. В этой первой версии в листинге 8.20 мы рассмотрим имплементацию, в которой для упаковки постолбцовых данных используется вызов `MPI_Pack`. Построчные данные отправляются только с помощью стандартного вызова `MPI_Isend`. Ширина участка призрачных ячеек задается переменной `nhalo`, и углы могут запрашиваться на входе надлежащим образом.

Листинг 8.20 Процедура обновления призрачных ячеек для двухмерной сетки с помощью MPI_Pack

`GhostExchange/GhostExchange_Pack/GhostExchange.cc`

Обновление призрачных ячеек из смежных процессов

```
167 void ghostcell_update(double **x, int nhalo,
int    corners, int jsize, int isize,
168    int nleft, int nrght, int nbot, int ntop,
       int do_timing)
169 {
170    if (do_timing) MPI_Barrier(MPI_COMM_WORLD);
171
172    struct timespec tstart_ghostcell;
173    cpu_timer_start(&tstart_ghostcell);
174
175    MPI_Request request[4*nhalo];
176    MPI_Status status[4*nhalo];
177
178    int jlow=0, jhgh=jsize;
179    if (corners) {
```

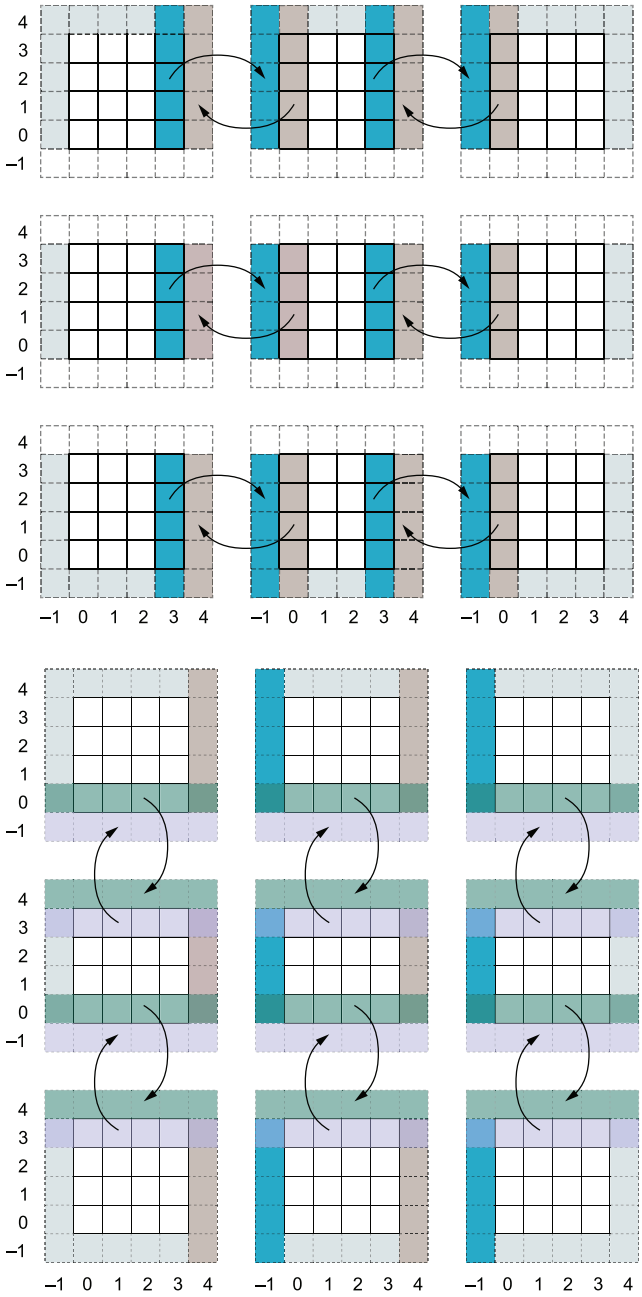


Рис. 8.5 В версии обновления призрачных ячеек с использованием угловых ячеек сначала происходит обмен данными слева и справа (в верхней половине рисунка), а затем обмен данными сверху и снизу (в нижней половине рисунка). При соблюдении осторожности левый и правый обмен могут быть меньше только с реальными ячейками плюс ячейками по внешней границе, хотя нет никакого вреда в том, чтобы делать его по всему вертикальному размеру сетки. Обновление граничных ячеек вокруг сетки выполняется отдельно

```
180       if (nbot == MPI_PROC_NULL) jlow = -nhalo;
181       if (ntop == MPI_PROC_NULL) jhgh = jsize+nhalo;
182    }
183    int jnum = jhgh-jlow;
184    int bufcount = jnum*nhalo;
185    int bufsize = bufcount*sizeof(double);
186
187    double xbuf_left_send[bufcount];
188    double xbuf_rght_send[bufcount];
189    double xbuf_rght_recv[bufcount];
190    double xbuf_left_recv[bufcount];
191
192    int position_left;
193    int position_right;
194    if (nleft != MPI_PROC_NULL){
195       position_left = 0;
196       for (int j = jlow; j < jhgh; j++){
197          MPI_Pack(&x[j][0], nhalo, MPI_DOUBLE,
198             xbuf_left_send, bufsize,
                &position_left, MPI_COMM_WORLD);
199       }
200    }
201
202    if (nrght != MPI_PROC_NULL){
203       position_right = 0;
204       for (int j = jlow; j < jhgh; j++){
205          MPI_Pack(&x[j][isize-nhalo], nhalo,
                MPI_DOUBLE, xbuf_rght_send,
206             bufsize, &position_right,
                MPI_COMM_WORLD);
207       }
208    }
209
210    MPI_Irecv(&xbuf_rght_recv, bufsize,
                MPI_PACKED, nrght, 1001,
211             MPI_COMM_WORLD, &request[0]);
212    MPI_Isend(&xbuf_left_send, bufsize,
                MPI_PACKED, nleft, 1001,
213             MPI_COMM_WORLD, &request[1]);
214
215    MPI_Irecv(&xbuf_left_recv, bufsize,
                MPI_PACKED, nleft, 1002,
216             MPI_COMM_WORLD, &request[2]);
217    MPI_Isend(&xbuf_rght_send, bufsize,
                MPI_PACKED, nrght, 1002,
218             MPI_COMM_WORLD, &request[3]);
219    MPI_Waitall(4, request, status);
220
```

Упаковывает буферы для обновления призрачных ячеек левых и правых соседей

Обмен данными между левыми и правыми соседями

```
221    if (nrght != MPI_PROC_NULL){
222       position_right = 0;
223       for (int j = jlow; j < jhgh; j++){
224          MPI_Unpack(xbuf_rght_recv, bufsize,
              &position_right, &x[j][isize],
225           nhalo, MPI_DOUBLE, MPI_COMM_WORLD);
226       }
227    }
228
229    if (nleft != MPI_PROC_NULL){
230       position_left = 0;
231       for (int j = jlow; j < jhgh; j++){
232          MPI_Unpack(xbuf_left_recv, bufsize,
              &position_left, &x[j][-nhalo],
233           nhalo, MPI_DOUBLE, MPI_COMM_WORLD);
234       }
235    }
236
237    if (corners) {
238       bufcount = nhalo*(isize+2*nhalo);
239       MPI_Irecv(&x[jsize][-nhalo],
             bufcount, MPI_DOUBLE, ntop, 1001,
240          MPI_COMM_WORLD, &request[0]);
241       MPI_Isend(&x[0 ][-nhalo],
             bufcount, MPI_DOUBLE, nbot, 1001,
242          MPI_COMM_WORLD, &request[1]);
243
244       MPI_Irecv(&x[ -nhalo][-nhalo],
             bufcount, MPI_DOUBLE, nbot, 1002,
245          MPI_COMM_WORLD, &request[2]);
246       MPI_Isend(&x[jsize-nhalo][-nhalo],
             bufcount, MPI_DOUBLE,ntop, 1002,
247          MPI_COMM_WORLD, &request[3]);
248       MPI_Waitall(4, request, status);
249    } else {
250       for (int j = 0; j<nhalo; j++){
251          MPI_Irecv(&x[jsize+j][0],
              isize, MPI_DOUBLE, ntop, 1001+j*2,
252           MPI_COMM_WORLD, &request[0+j*4]);
253          MPI_Isend(&x[0+j ][0],
              isize, MPI_DOUBLE, nbot, 1001+j*2,
254           MPI_COMM_WORLD, &request[1+j*4]);
255
256          MPI_Irecv(&x[ -nhalo+j][0],
              isize, MPI_DOUBLE, nbot, 1002+j*2,
257           MPI_COMM_WORLD, &request[2+j*4]);
258          MPI_Isend(&x[jsize-nhalo+j][0],
              isize, MPI_DOUBLE, ntop, 1002+j*2,
259           MPI_COMM_WORLD, &request[3+j*4]);
260       }
```

Распаковывает буферы левых и правых соседей

Обновления призрачных ячеек в одном сплошном блоке для нижних и верхних соседей

Ожидает завершения всего обмена данными

Обновления призрачных ячеек по одной строке для нижних и верхних соседей

```
261       MPI_Waitall(4*nhalo, request, status);
262    }
263
264    if (do_timing) MPI_Barrier(MPI_COMM_WORLD);
265
266    ghostcell_time += cpu_timer_stop(tstart_ghostcell);
267 }
```

Ожидает завершения всего обмена данными

Вызов MPI_Pack особенно полезен, когда в обновлении призраков необходимо передавать несколько типов данных. Значения упаковываются в буфер, не зависящий от типа, а затем распаковываются с другой стороны. Обмен с соседями в вертикальном направлении осуществляется с помощью данных сплошных строк. Когда учитываются углы, хорошо работает один буфер. Без углов отправляются отдельные строки ореола. Обычно есть только одна-две ячейки ореола, так что этот подход является разумным.

Еще одним способом загрузки буферов для обмена данными состоит в присваиваниях в массиве. Указанный подход хорош, когда имеется один простой тип данных, такой как используемый в этом примере тип с плавающей точкой двойной точности. В следующем ниже листинге показан исходный код для замены циклов MPI_Pack присваиваниями в массиве.

Листинг 8.21 Процедура обновления призрачных ячеек для двухмерной сетки с присваиваниями в массиве

GhostExchange/GhostExchange_ArrayAssign/GhostExchange.cc

```
190    int icount;
191    if (nleft != MPI_PROC_NULL){
192       icount = 0;
193       for (int j = jlow; j < jhgh; j++){
194          for (int ll = 0; ll < nhalo; ll++){
195             xbuf_left_send[icount++] = x[j][ll];
196          }
197       }
198    }
199    if (nrght != MPI_PROC_NULL){
200       icount = 0;
201       for (int j = jlow; j < jhgh; j++){
202          for (int ll = 0; ll < nhalo; ll++){
203             xbuf_rght_send[icount++] =
                  x[j][isize-nhalo+ll];
204          }
205       }
206    }
207
```

Заполняем буферы отправки

```
208    MPI_Irecv(&xbuf_rght_recv, bufcount,
                 MPI_DOUBLE, nrght, 1001,
209              MPI_COMM_WORLD, &request[0]);
210    MPI_Isend(&xbuf_left_send, bufcount,
                 MPI_DOUBLE, nleft, 1001,
211              MPI_COMM_WORLD, &request[1]);
212
213    MPI_Irecv(&xbuf_left_recv, bufcount,
                 MPI_DOUBLE, nleft, 1002,
214              MPI_COMM_WORLD, &request[2]);
215    MPI_Isend(&xbuf_rght_send, bufcount,
                 MPI_DOUBLE, nrght, 1002,
216              MPI_COMM_WORLD, &request[3]);
217    MPI_Waitall(4, request, status);
218
219    if (nrght != MPI_PROC_NULL){
220       icount = 0;
221       for (int j = jlow; j < jhgh; j++){
222          for (int ll = 0; ll < nhalo; ll++){
223             x[j][isize+ll] =
                    xbuf_rght_recv[icount++];
224          }
225       }
226    }
227    if (nleft != MPI_PROC_NULL){
228       icount = 0;
229       for (int j = jlow; j < jhgh; j++){
230          for (int ll = 0; ll < nhalo; ll++){
231             x[j][-nhalo+ll] =
                    xbuf_left_recv[icount++];
232          }
233       }
234    }
```

Выполняем обмен данными между левыми и правыми соседями

Копирует буферы приемки в призрачные ячейки

В вызовах `MPI_Irecv` и `MPI_Isend` теперь используется счетчик (count) и тип данных `MPI_DOUBLE`, а не генерический байтовый тип функции `MPI_Pack`. Нам также необходимо знать тип данных для копирования данных в коммуникационный буфер и из него.

8.4.3 Обмен с призрачными ячейками в трехмерной стенсильной калькуляции

Обмениваться данными с призрачными ячейками можно также для трехмерного стенсильного расчета. Мы делаем это в листинге 8.22. Однако организация этого процесса немного сложнее. Сначала рассчитывается схема процесса как значения `xcoord`, `ycoord` и `zcoord`. Затем определяются соседи и вычисляются размеры данных на каждом процессоре.

Листинг 8.22 Настройка трехмерной сетки

GhostExchange/GhostExchange3D_*/GhostExchange.cc

```
63    int xcoord = rank%nprocx;
64    int ycoord = rank/nprocx%nprocy;
65    int zcoord = rank/(nprocx*nprocy);
66
67    int nleft = (xcoord > 0 ) ?
         rank - 1 : MPI_PROC_NULL;
68    int nrght = (xcoord < nprocx-1) ?
         rank + 1 : MPI_PROC_NULL;
69    int nbot = (ycoord > 0 ) ?
         rank - nprocx : MPI_PROC_NULL;
70    int ntop = (ycoord < nprocy-1) ?
         rank + nprocx : MPI_PROC_NULL;
71    int nfrnt = (zcoord > 0 ) ?
         rank - nprocx * nprocy : MPI_PROC_NULL;
72    int nback = (zcoord < nprocz-1) ?
         rank + nprocx * nprocy : MPI_PROC_NULL;
73
74    int ibegin = imax *(xcoord )/nprocx;
75    int iend = imax *(xcoord+1)/nprocx;
76    int isize = iend - ibegin;
77    int jbegin = jmax *(ycoord )/nprocy;
78    int jend = jmax *(ycoord+1)/nprocy;
79    int jsize = jend - jbegin;
80    int kbegin = kmax *(zcoord )/nprocz;
81    int kend = kmax *(zcoord+1)/nprocz;
82    int ksize = kend - kbegin;
```

Устанавливает координаты процесса

Вычисляет соседние процессы для каждого процесса

Вычисляет начальный и конечный индексы для каждого процесса, а затем размер

Обновление призрачных ячеек, включая копирование массивов в буферы, обмен и копирование, занимает пару сотен строк, и показать это здесь невозможно. За детальной имплементацией обратитесь к примерам исходного кода (https://github.com/EssentialsofParallelComputing/Chapter8), прилагаемым к данной главе. Мы покажем версию обновления призрачных ячеек с типом данных MPI в разделе 8.5.1.

8.5 Продвинутая функциональность MPI для упрощения исходного кода и обеспечения оптимизаций

Превосходный дизайн MPI становится очевидным, когда мы видим, как базовые компоненты MPI могут комбинироваться в функциональность более высокого уровня. Мы познакомились с этим в разделе 8.3.3, когда создали новый тип double-double и новый оператор редукции. Такая расширяемость наделяет MPI важными способностями. Мы взглянем на пару этих расширенных функций, которые полезны в общепринятых приложениях параллельности данных. К ним относятся:

- *конкретно-прикладные типы данных MPI* – строит новые типы данных из строительных блоков базовых типов MPI;
- *поддержка топологии* – доступны базовая декартова регулярная сеточная топология и более общая графовая топология. Мы рассмотрим только более простые декартовы функции MPI.

8.5.1 Использование конкретно-прикладных типов данных MPI для повышения производительности и упрощения кода

MPI имеет богатый набор функций для создания новых конкретно-прикладных типов данных MPI из базовых типов MPI. Он позволяет инкапсулировать сложные данные в единый конкретно-прикладной тип данных, который можно использовать в коммуникационных вызовах. Как следствие, один коммуникационный вызов может отправлять или принимать много меньших порций данных как единое целое. Вот список нескольких функций создания типов данных MPI:

- `MPI_Type_contiguous` – превращает блок сплошных данных в тип:
- `MPI_Type_vector` – создает тип из блоков шаговых данных;
- `MPI_Type_create_subarray` – создает прямоугольное подмножество большего массива;
- `MPI_Type_indexed` или `MPI_Type_create_hindexed` – создает нерегулярный набор индексов, описываемых набором длин блоков и смещений. Индексированная версия, для большей общности, выражает смещения в байтах вместо типа данных;
- `MPI_Type_create_struct` – создает тип данных, инкапсулирующий элементы данных в структуре в переносимой форме, которая учитывает холостое заполнение компилятором.

Ниже приведена наглядная иллюстрация, которая поможет вам понять некоторые из этих типов данных. На рис. 8.6 показано несколько более простых и часто используемых функций, включая `MPI_Type_contiguous`, `MPI_Type_vector` и `MPI_Type_create_subarray`.

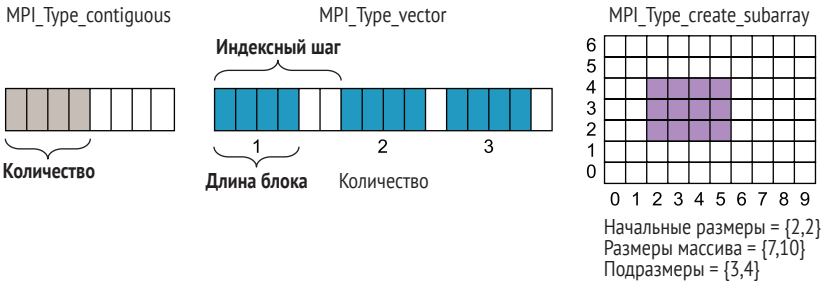


Рис. 8.6 Три конкретно-прикладных типа данных MPI с иллюстрациями аргументов, используемых при их создании

После того как тип данных описан и преобразован в новый тип данных, он должен быть инициализирован перед его использованием. Для этой цели существует несколько дополнительных процедур для фиксации и высвобождения типов. Тип должен быть зафиксирован перед его использованием, и он должен быть высвобожден во избежание утечки памяти. Эти процедуры таковы:

- `MPI_Type_Commit` – инициализирует новый конкретно-прикладной тип с необходимым выделением памяти или другой настройкой;
- `MPI_Type_Free` – высвобождает любую память или элементы структуры данных после создания типа данных.

Мы можем значительно упростить обмен с призрачными ячейками, определив конкретно-прикладной тип данных MPI, как показано на рис. 8.6, для представления столбца данных и предотвращения вызовов `MPI_Pack`. Определив тип данных MPI, можно избежать дополнительной копии данных. Данные могут быть скопированы из своего обычного местоположения прямо в буферы MPI отправки. Давайте посмотрим, как это делается в листинге 8.23. В листинге 8.24 показана вторая часть программы.

Сначала мы настраиваем конкретно-прикладные типы данных. Мы используем вызов `MPI_Type_vector` для наборов обращений к массиву с индексным шагом. В случае сплошных данных для вертикального типа, когда мы используем углы, то используем вызов `MPI_Type_contiguous`, и в строках 139 и 140 мы высвобождаем тип данных в конце перед `MPI_Finalize`.

Листинг 8.23 Создание двухмерного векторного типа данных для обновления призрачных ячеек

`GhostExchange/GhostExchange_VectorTypes/GhostExchange.cc`

```
 56    int jlow=0, jhgh=jsize;
 57    if (corners) {
 58       if (nbot == MPI_PROC_NULL) jlow = -nhalo;
 59       if (ntop == MPI_PROC_NULL) jhgh = jsize+nhalo;
 60    }
 61    int jnum = jhgh-jlow;
 62
 63    MPI_Datatype horiz_type;
 64    MPI_Type_vector(jnum, nhalo, isize+2*nhalo,
                       MPI_DOUBLE, &horiz_type);
 65    MPI_Type_commit(&horiz_type);
 66
 67    MPI_Datatype vert_type;
 68    if (! corners){
 69       MPI_Type_vector(nhalo, isize, isize+2*nhalo,
                         MPI_DOUBLE, &vert_type);
 70    } else {
 71       MPI_Type_contiguous(nhalo*(isize+2*nhalo),
                             MPI_DOUBLE, &vert_type);
 72    }
 73    MPI_Type_commit(&vert_type);
...
139    MPI_Type_free(&horiz_type);
140    MPI_Type_free(&vert_type);
```

Затем вы можете написать `ghostcell_update` более кратко и с лучшей производительностью, используя типы данных MPI, как показано в следующем ниже листинге. Если нам нужно обновить углы, то необходима синхронизация между двумя проходами обмена данными.

Листинг 8.24 Процедура обновления двухмерных призрачных ячеек с использованием векторного типа данных

GhostExchange/GhostExchange_VectorTypes/GhostExchange.cc

```
197    int jlow=0, jhgh=jsize, ilow=0, waitcount=8, ib=4;
198    if (corners) {
199       if (nbot == MPI_PROC_NULL) jlow = -nhalo;
200       ilow = -nhalo;
201       waitcount = 4;
202       ib = 0;
203    }
204
205    MPI_Request request[waitcount];
206    MPI_Status status[waitcount];
207
208    MPI_Irecv(&x[jlow][isize], 1,
          horiz_type, nrght, 1001,
209       MPI_COMM_WORLD, &request[0]);
210    MPI_Isend(&x[jlow][0], 1,
          horiz_type, nleft, 1001,
211       MPI_COMM_WORLD, &request[1]);
212
213    MPI_Irecv(&x[jlow][-nhalo], 1,
          horiz_type, nleft, 1002,
214       MPI_COMM_WORLD, &request[2]);
215    MPI_Isend(&x[jlow][isize-nhalo], 1,
          horiz_type, nrght, 1002,
216       MPI_COMM_WORLD, &request[3]);
217
218    if (corners)
         MPI_Waitall(4, request, status);
219
220    MPI_Irecv(&x[jsize][ilow], 1,
          vert_type, ntop, 1003,
221       MPI_COMM_WORLD, &request[ib+0]);
222    MPI_Isend(&x[0 ][ilow], 1,
          vert_type, nbot, 1003,
223       MPI_COMM_WORLD, &request[ib+1]);
224
225    MPI_Irecv(&x[ -nhalo][ilow], 1,
          vert_type, nbot, 1004,
226       MPI_COMM_WORLD, &request[ib+2]);
227    MPI_Isend(&x[jsize-nhalo][ilow], 1,
          vert_type, ntop, 1004,
228       MPI_COMM_WORLD, &request[ib+3]);
229
230    MPI_Waitall(waitcount, request, status);
```

Отправить влево и вправо, используя конкретно-прикладной тип данных MPI horiz_type

Синхронизировать, если будут отправлены углы

Обновляет призрачные ячейки сверху и снизу

В качестве причины использования типов данных MPI обычно указывается более высокая производительность. Это действительно позволяет имплементировать MPI, в некоторых случаях избегая дополнительной копии. Но, с нашей точки зрения, самой главной причиной типов данных MPI является более чистый, простой исходный код и меньше возможностей для ошибок.

Трехмерная версия с использованием типов данных MPI немного сложнее. В следующем ниже листинге мы используем `MPI_Type_create_subarray` для создания трех конкретно-прикладных типов данных MPI, которые будут использоваться в обмене данными.

Листинг 8.25 Создание типа данных в форме подмассива MPI для трехмерных призрачных ячеек

GhostExchange/GhostExchange3D_VectorTypes/GhostExchange.cc

```
109    int array_sizes[] = {ksize+2*nhalo, jsize+2*nhalo, isize+2*nhalo};
110    if (corners) {
111       int subarray_starts[] = {0, 0, 0};
112       int hsubarray_sizes[] =
             {ksize+2*nhalo, jsize+2*nhalo, nhalo};
113       MPI_Type_create_subarray(3,
             array_sizes, hsubarray_sizes,
114          subarray_starts, MPI_ORDER_C,
             MPI_DOUBLE, &horiz_type);
115
116       int vsubarray_sizes[] =
             {ksize+2*nhalo, nhalo, isize+2*nhalo};
117       MPI_Type_create_subarray(3,
             array_sizes, vsubarray_sizes,
118          subarray_starts, MPI_ORDER_C,
             MPI_DOUBLE, &vert_type);
119
120       int dsubarray_sizes[] =
             {nhalo, jsize+2*nhalo,
              isize+2*nhalo};
121       MPI_Type_create_subarray(3,
             array_sizes, dsubarray_sizes,
122          subarray_starts, MPI_ORDER_C,
             MPI_DOUBLE, &depth_type);
123    } else {
124       int hsubarray_starts[] = {nhalo,nhalo,0};
125       int hsubarray_sizes[] = {ksize, jsize,
                                  nhalo};
126       MPI_Type_create_subarray(3,
             array_sizes, hsubarray_sizes,
127          hsubarray_starts, MPI_ORDER_C,
             MPI_DOUBLE, &horiz_type);
128
129       int vsubarray_starts[] = {nhalo, 0, nhalo};
130       int vsubarray_sizes[] = {ksize, nhalo, isize};
131       MPI_Type_create_subarray(3, array_sizes, vsubarray_sizes,
132          vsubarray_starts, MPI_ORDER_C, MPI_DOUBLE, &vert_type);
```

Создает горизонтальный тип данных с помощью MPI_Type_create_subarray

Создает тип данных глубины с помощью MPI_Type_create_subarray

Создает вертикальный тип данных с помощью MPI_Type_create_subarray

```
133
134       int dsubarray_starts[] = {0, nhalo, nhalo};
135       int dsubarray_sizes[] = {nhalo, ksize, isize};
136       MPI_Type_create_subarray(3,
             array_sizes, dsubarray_sizes,
137          dsubarray_starts, MPI_ORDER_C,
             MPI_DOUBLE, &depth_type);
138    }
139
140    MPI_Type_commit(&horiz_type);
141    MPI_Type_commit(&vert_type);
142    MPI_Type_commit(&depth_type);
```

Создает тип данных глубины с помощью MPI_Type_create_subarray

Следующий ниже листинг показывает, что процедура обмена данными с использованием этих трех типов данных MPI довольно лаконична.

Листинг 8.26 Обновление трехмерных призрачных ячеек с использованием типов данных MPI

GhostExchange/GhostExchange3D_VectorTypes/GhostExchange.cc

```
334    int waitcount = 12, ib1 = 4, ib2 = 8;
335    if (corners) {
336       waitcount=4;
337       ib1 = 0, ib2 = 0;
338    }
339
340    MPI_Request request[waitcount*nhalo];
341    MPI_Status status[waitcount*nhalo];
342
343    MPI_Irecv(&x[-nhalo][-nhalo][isize], 1, horiz_type, nrght, 1001,
344              MPI_COMM_WORLD, &request[0]);
345    MPI_Isend(&x[-nhalo][-nhalo][0], 1, horiz_type, nleft, 1001,
346              MPI_COMM_WORLD, &request[1]);
347
348    MPI_Irecv(&x[-nhalo][-nhalo][-nhalo], 1, horiz_type, nleft, 1002,
349              MPI_COMM_WORLD, &request[2]);
350    MPI_Isend(&x[-nhalo][-nhalo][isize-1], 1, horiz_type, nrght, 1002,
351              MPI_COMM_WORLD, &request[3]);
352    if (corners)
          MPI_Waitall(4, request, status);
353
354    MPI_Irecv(&x[-nhalo][jsize][-nhalo], 1, vert_type, ntop, 1003,
355              MPI_COMM_WORLD, &request[ib1+0]);
356    MPI_Isend(&x[-nhalo][0][-nhalo], 1, vert_type, nbot, 1003,
357              MPI_COMM_WORLD, &request[ib1+1]);
358
359    MPI_Irecv(&x[-nhalo][-nhalo][-nhalo], 1, vert_type, nbot, 1004,
360              MPI_COMM_WORLD, &request[ib1+2]);
361    MPI_Isend(&x[-nhalo][jsize-1][-nhalo], 1, vert_type, ntop, 1004,
362              MPI_COMM_WORLD, &request[ib1+3]);
363    if (corners)
          MPI_Waitall(4, request, status);
```

Обновление призрачных ячеек для горизонтального направления

Обновление призрачных ячеек для вертикального направления

Синхронизировать, если в обновлении требуются углы

```
364
365    MPI_Irecv(&x[ksize][-nhalo][-nhalo], 1,
                 depth_type, nback, 1005,
366              MPI_COMM_WORLD, &request[ib2+0]);
367    MPI_Isend(&x[0][-nhalo][-nhalo], 1,
                 depth_type, nfrnt, 1005,
368              MPI_COMM_WORLD, &request[ib2+1]);
369
370    MPI_Irecv(&x[-nhalo][-nhalo][-nhalo], 1,
                 depth_type, nfrnt, 1006,
371              MPI_COMM_WORLD, &request[ib2+2]);
372    MPI_Isend(&x[ksize-1][-nhalo][-nhalo], 1,
                 depth_type, nback, 1006,
373              MPI_COMM_WORLD, &request[ib2+3]);
374    MPI_Waitall(waitcount, request, status);
```

Обновление призрачных ячеек для направления в глубину

Обновление призрачных ячеек для направления в глубину

8.5.2 Поддержка декартовой топологии в MPI

В этом разделе мы покажем вам работу топологических функций в MPI. Операция по-прежнему представляет собой обмен с призрачными ячейками, показанный на рис. 8.5, но мы можем упростить кодирование с помощью декартовых функций. Не охвачены общие графовые функции для неструктурированных приложений. Мы начнем с настроечных процедур и потом перейдем к коммуникационным процедурам.

В настроечных процедурах необходимо задать значения для присваиваний в процессной сетке, а затем задать соседей, как это было сделано в листингах 8.18 и 8.22. Как показано в листинге 8.24 для двух размерностей и листинге 8.25 для трех размерностей, процесс устанавливает в массиве `dims` равным числу процессов, используемых в каждой размерности. Если какое-либо из значений в массиве `dims` равно нулю, то функция `MPI_Dims_create` вычисляет некоторые значения, которые будут работать. Обратите внимание, что число процессов в каждом направлении не учитывает размер сетки и, возможно, не будет давать хороших значений для длинных узких задач. Подумайте о случае сетки размером 8×8×1000 и дайте ей 8 процессоров; процессная сетка будет иметь размер 2×2×2, что приведет к сеточному домену 4×4×500 для каждого процесса.

Процедура `MPI_Cart_create` берет результирующий массив `dims` и входной массив, periodic, который объявляет, будет ли граница циклически переходить на противоположную сторону и наоборот, или нет. Последним идет аргумент переупорядочивания, который позволяет MPI изменять порядок процессов. В данном примере он равен нулю (ложь). Теперь у нас есть новый коммуникатор, содержащий информацию о топологии.

Получение схемы процессной сетки представлен простым вызовом `MPI_Cart_coords`. Путем вызова `MPI_Cart_shift` можно получить соседей, при этом второй аргумент указывает направление, а третий аргумент – смещение или число процессов в этом направлении. Результатом на выходе являются ранги смежных процессоров.

Листинг 8.27 Поддержка двухмерной декартовой топологии в MPI

GhostExchange/CartExchange_Neighbor/CartExchange.cc

```
43 int dims[2] = {nprocy, nprocx};
44 int periodic[2]={0,0};
45 int coords[2];
46 MPI_Dims_create(nprocs, 2, dims);
47 MPI_Comm cart_comm;
48 MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periodic, 0, &cart_comm);
49 MPI_Cart_coords(cart_comm, rank, 2, coords);
50
51 int nleft, nrght, nbot, ntop;
52 MPI_Cart_shift(cart_comm, 1, 1, &nleft, &nrght);
53 MPI_Cart_shift(cart_comm, 0, 1, &nbot, &ntop);
```

Настройка трехмерной декартовой топологии аналогична, но имеет три размерности, как показано в следующем ниже листинге.

Листинг 8.28 Поддержка трехмерной декартовой топологии в MPI

GhostExchange/CartExchange3D_Neighbor/CartExchange.cc

```
65 int dims[3] = {nprocz, nprocy, nprocx};
66 int periods[3]={0,0,0};
67 int coords[3];
68 MPI_Dims_create(nprocs, 3, dims);
69 MPI_Comm cart_comm;
70 MPI_Cart_create(MPI_COMM_WORLD, 3, dims, periods, 0, &cart_comm);
71 MPI_Cart_coords(cart_comm, rank, 3, coords);
72 int xcoord = coords[2];
73 int ycoord = coords[1];
74 int zcoord = coords[0];
75
76 int nleft, nrght, nbot, ntop, nfrnt, nback;
77 MPI_Cart_shift(cart_comm, 2, 1, &nleft, &nrght);
78 MPI_Cart_shift(cart_comm, 1, 1, &nbot, &ntop);
79 MPI_Cart_shift(cart_comm, 0, 1, &nfrnt, &nback);
```

Если сравнить этот исходный код с версиями в листингах 8.19 и 8.23, то мы увидим, что топологические функции в этом относительно простом настроечном примере не экономят много строк кода, и не снижают сложность программирования в значительной мере. Мы также можем эффективно задействовать декартов коммуникатор, созданный в строке 70 листинга 8.28, для обмена с соседями. Именно здесь наблюдается наибольшее сокращение строк кода. Функция MPI имеет следующие аргументы:

```
int MPI_Neighbor_alltoallw(const void *sendbuf,
                           const int sendcounts[],
                           const MPI_Aint sdispls[],
                           const MPI_Datatype sendtypes[],
```

```
                    void *recvbuf,
                    const int recvcounts[],
                    const MPI_Aint rdispls[],
                    const MPI_Datatype recvtypes[],
                    MPI_Comm comm)
```

В вызове соседа много аргументов, но, как только мы все это настроим, обмен будет кратким и выполненным в одной инструкции. Мы пройдемся по всем аргументам подробно, потому что их бывает трудно понять правильно.

Коммуникационный вызов соседа может использовать либо заполненный буфер для отправок и приемок, либо выполнять операцию прямо на месте. Мы покажем метод, выполняемый прямо на месте. Буферы отправок и приемок представляют собой двухмерный массив. Мы будем использовать тип данных MPI для описания блока данных, поэтому счетчики будут представлять собой массив со значением единицы для всех четырех декартовых сторон в случае двух размерностей либо шести сторон в случае трех размерностей. Порядок обмена данными для сторон таков: снизу, сверху, слева, справа для двух размерностей и спереди, сзади, снизу, сверху, слева, справа для трех размерностей, и одинаков как для отправного и приемного типов.

Блок данных отличается для каждого направления: горизонтального, вертикального и в глубину. Мы используем рисунки в общепринятом стандартном ракурсе, при котором x идет вправо, y – вверх и z (глубина) возвращается на страницу. Однако в каждом направлении блок данных одинаков, но с разными смещениями к началу блока данных. Смещения указаны в байтах, поэтому вы увидите сдвиги, умноженные на 8, размер типа данных со значениями двойной точности. Теперь давайте посмотрим, как все это помещается в исходный код для настройки обмена данными для двухмерного случая в следующем ниже листинге.

Листинг 8.29 Настройка двухмерного декартового обмена с соседями

GhostExchange/CartExchange_Neighbor/CartExchange.c

```
55    int ibegin = imax *(coords[1] )/dims[1];
56    int iend = imax *(coords[1]+1)/dims[1];
57    int isize = iend - ibegin;
58    int jbegin = jmax *(coords[0] )/dims[0];
59    int jend = jmax *(coords[0]+1)/dims[0];
60    int jsize = jend - jbegin;
61
62    int jlow=nhalo, jhgh=jsize+nhalo,
          ilow=nhalo, inum = isize;
63    if (corners) {
64       int ilow = 0, inum = isize+2*nhalo;
65       if (nbot == MPI_PROC_NULL) jlow = 0;
66       if (ntop == MPI_PROC_NULL) jhgh = jsize+2*nhalo;
67    }
68    int jnum = jhgh-jlow;
```

Вычисляет глобальные начальный и конечный индексы и размер локального массива

Включает значения углов, если они запрашиваются

```
69
70    int array_sizes[] = {jsize+2*nhalo, isize+2*nhalo};
71
72    int subarray_sizes_x[] = {jnum, nhalo};
73    int subarray_horiz_start[] = {jlow, 0};
74    MPI_Datatype horiz_type;
75    MPI_Type_create_subarray (2, array_sizes,
         subarray_sizes_x, subarray_horiz_start,
76       MPI_ORDER_C, MPI_DOUBLE, &horiz_type);
77    MPI_Type_commit(&horiz_type);
78
79    int subarray_sizes_y[] = {nhalo, inum};
80    int subarray_vert_start[] = {0, jlow};
81    MPI_Datatype vert_type;
82    MPI_Type_create_subarray (2, array_sizes,
         subarray_sizes_y, subarray_vert_start,
83       MPI_ORDER_C, MPI_DOUBLE, &vert_type);
84    MPI_Type_commit(&vert_type);
85
86    MPI_Aint sdispls[4] = {
         nhalo *(isize+2*nhalo)*8,
87       jsize *(isize+2*nhalo)*8,
88       nhalo *8,
89       isize *8};
90    MPI_Aint rdispls[4] = {
         0,
91       (jsize+nhalo) *(isize+2*nhalo)*8,
92       0,
93       (isize+nhalo)*8};
94    MPI_Datatype sendtypes[4] = {vert_type,
         vert_type, horiz_type, horiz_type};
95    MPI_Datatype recvtypes[4] = {vert_type,
         vert_type, horiz_type, horiz_type};
```

Создает блок данных для обмена в горизонтальном направлении с помощью функции subarray

Создает блок данных для обмена в вертикальном направлении с помощью функции subarray

Нижняя строка находится nhalo над началом

Смещения указаны из левого нижнего угла блока памяти в байтах

Левый столбец находится nhalo справа от начала

Верхняя строка находится jsize над началом

Правый столбец находится isize справа от начала

Нижняя призрачная строка находится 0 позиций выше начала

Верхняя призрачная строка находится jsize+nhalo выше начала

Правая призрачная строка находится jsize+nhalo справа от начала

Приемочные типы упорядочены по нижнему, верхнему, левому и правому соседям

Левый призрачный столбец находится 0 позиций справа от начала

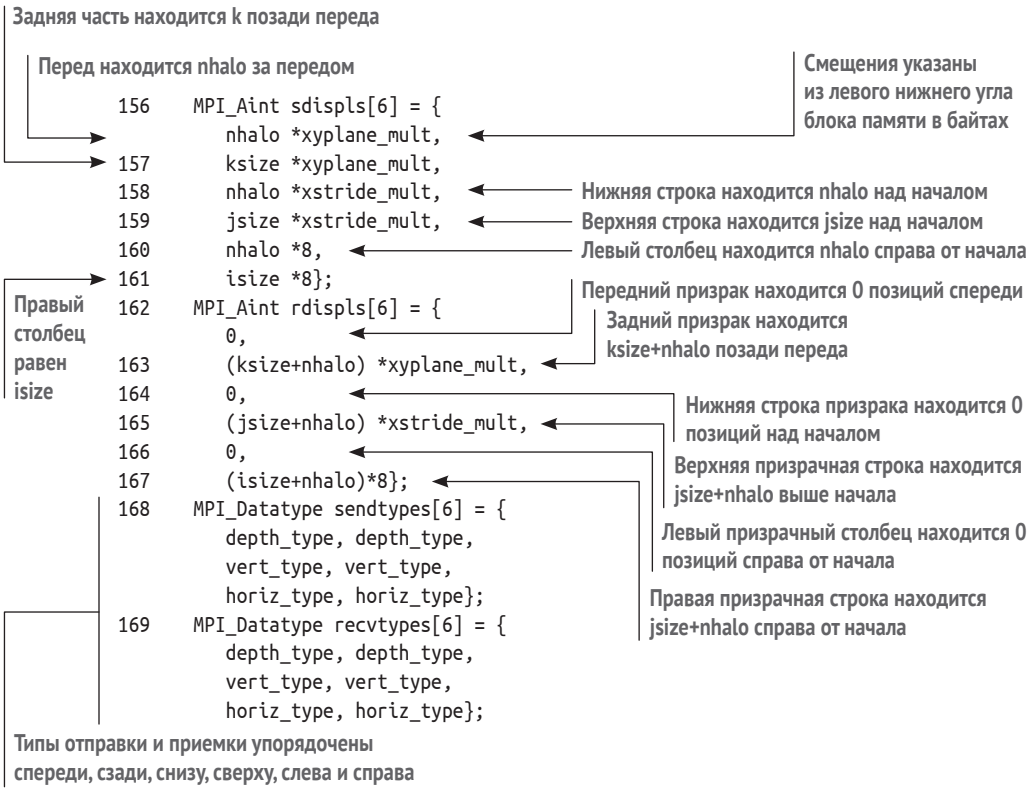
Отправочные типы упорядочены по нижнему, верхнему, левому и правому соседям

В конфигурации для трехмерного декартового обмена с соседями используются типы данных MPI из листинга 8.25. Типы данных определяют блок данных, который будет перемещен, но нам нужно определить сдвиг в байтах к начальному местоположению блока данных для отправки и приемки. Нам также необходимо определить массивы для типов `sendtype` и `recvtype` в надлежащем порядке, как в следующем ниже листинге.

Листинг 8.30 Настройка обмена между трехмерными декартовыми соседями

`GhostExchange/CartExchange3D_Neighbor/CartExchange.c`

```
154    int xyplane_mult = (jsize+2*nhalo)*(isize+2*nhalo)*8;
155    int xstride_mult = (isize+2*nhalo)*8;
```

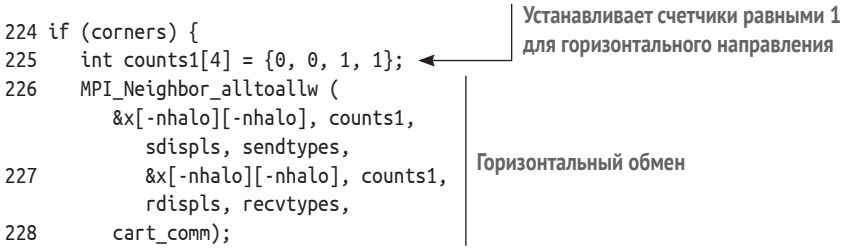



```
156    MPI_Aint sdispls[6] = {
          nhalo *xyplane_mult,
157       ksize *xyplane_mult,
158       nhalo *xstride_mult,
159       jsize *xstride_mult,
160       nhalo *8,
161       isize *8};
162    MPI_Aint rdispls[6] = {
          0,
163       (ksize+nhalo) *xyplane_mult,
164       0,
165       (jsize+nhalo) *xstride_mult,
166       0,
167       (isize+nhalo)*8};
168    MPI_Datatype sendtypes[6] = {
          depth_type, depth_type,
          vert_type, vert_type,
          horiz_type, horiz_type};
169    MPI_Datatype recvtypes[6] = {
          depth_type, depth_type,
          vert_type, vert_type,
          horiz_type, horiz_type};
```

Фактический обмен осуществляется с помощью одного вызова `MPI_Neighbor_alltoallw`, как показано в листинге 8.31. Существует также второй блок кода для угловых случаев, для которого требуется несколько вызовов с синхронизацией между ними с целью обеспечения того, чтобы углы заполнялись надлежащим образом. Первый вызов выполняет только горизонтальное направление, а затем ожидает завершения, прежде чем выполнять вертикальное направление.

Листинг 8.31 Двухмерный декартов обмен с соседями

`GhostExchange/CartExchange_Neighbor/CartExchange.c`




```
224 if (corners) {
225    int counts1[4] = {0, 0, 1, 1};
226    MPI_Neighbor_alltoallw (
          &x[-nhalo][-nhalo], counts1,
             sdispls, sendtypes,
227          &x[-nhalo][-nhalo], counts1,
             rdispls, recvtypes,
228       cart_comm);
229
```

```
230    int counts2[4] = {1, 1, 0, 0};
231    MPI_Neighbor_alltoallw (
          &x[-nhalo][-nhalo], counts2,
             sdispls, sendtypes,
232       &x[-nhalo][-nhalo], counts2,
             rdispls, recvtypes,
233       cart_comm);
234 } else {
235    int counts[4] = {1, 1, 1, 1};
236    MPI_Neighbor_alltoallw (
          &x[-nhalo][-nhalo], counts,
             sdispls, sendtypes,
237       &x[-nhalo][-nhalo], counts,
             rdispls, recvtypes,
238       cart_comm);
239 }
```

Вертикальный обмен

Устанавливает счетчики равными 1 для вертикального направления

Устанавливает для всех счетчиков значение 1 по всем направлениям

Весь обмен с соседями осуществляется за один вызов

Трехмерный декартов обмен с соседями аналогичен, но с добавлением координаты z (глубины). Глубина занимает первое место в массивах счетчиков и типов. В пофазовом обмене для углов глубина определяется после горизонтального и вертикального обмена с призрачными ячейками, как показано в следующем ниже листинге.

Листинг 8.32 Трехмерный декартов обмен с соседями

GhostExchange/CartExchange3D_Neighbor/CartExchange.c

```
346 if (corners) {
347    int counts1[6] = {0, 0, 0, 0, 1, 1};
348    MPI_Neighbor_alltoallw(
          &x[-nhalo][-nhalo][-nhalo], counts1,
             sdispls, sendtypes,
349       &x[-nhalo][-nhalo][-nhalo], counts1,
             rdispls, recvtypes,
350       cart_comm);
351
352    int counts2[6] = {0, 0, 1, 1, 0, 0};
353    MPI_Neighbor_alltoallw(
          &x[-nhalo][-nhalo][-nhalo], counts2,
             sdispls, sendtypes,
354       &x[-nhalo][-nhalo][-nhalo], counts2,
             rdispls, recvtypes,
355       cart_comm);
356
357    int counts3[6] = {1, 1, 0, 0, 0, 0};
358    MPI_Neighbor_alltoallw(
          &x[-nhalo][-nhalo][-nhalo], counts3,
             sdispls, sendtypes,
359       &x[-nhalo][-nhalo][-nhalo], counts3,
             rdispls, recvtypes,
360       cart_comm);
361 } else {
```

Горизонтальный призрачный обмен

Вертикальный призрачный обмен

Глубинный призрачный обмен

```
362    int counts[6] = {1, 1, 1, 1, 1, 1};
363    MPI_Neighbor_alltoallw(
          &x[-nhalo][-nhalo][-nhalo], counts,
             sdispls, sendtypes,
364       &x[-nhalo][-nhalo][-nhalo], counts,
             rdispls, recvtypes,
365       cart_comm);
366 }
```

Все соседи сразу

8.5.3 Тесты производительности вариантов обмена с призрачными ячейками

Давайте испытаем эти варианты обмена с призрачными ячейками в тестовой системе. Мы будем использовать два узла Broadwell (процессор Intel® Xeon® E5-2695 v4 с частотой 2.10 ГГц) с 72 виртуальными ядрами каждый. Мы могли бы выполнить это испытание на большем числе вычислительных узлов с разными имплементациями библиотеки MPI, размерами ореолов, размерами ячеек и более высокопроизводительными коммуникационными межсоединениями в целях более полного представления о том, как работает каждый вариант обмена с призрачными ячейками. Ниже приведен исходный код:

```
mpirun -n 144 --bind-to hwthread ./GhostExchange -x 12 -y 12 -i 20000 \
       -j 20000 -h 2 -t -c
mpirun -n 144 --bind-to hwthread ./GhostExchange -x 6 -y 4 -z 6 -i 700 \
       -j 700 -k 700 -h 2 -t -c
```

Опции программы GhostExchange таковы:

- `-x` (процессы в направлении x);
- `-y` (процессы в направлении y);
- `-z` (процессы в направлении z);
- `-i` (размер сетки в направлении i или x);
- `-j` (размер сетки в направлении j или y);
- `-k` (размер сетки в направлении k или z);
- `-h` (ширина ореольных ячеек, обычно 1 или 2);
- `-c` (включая угловые ячейки).

Упражнение: тесты призрачных ячеек

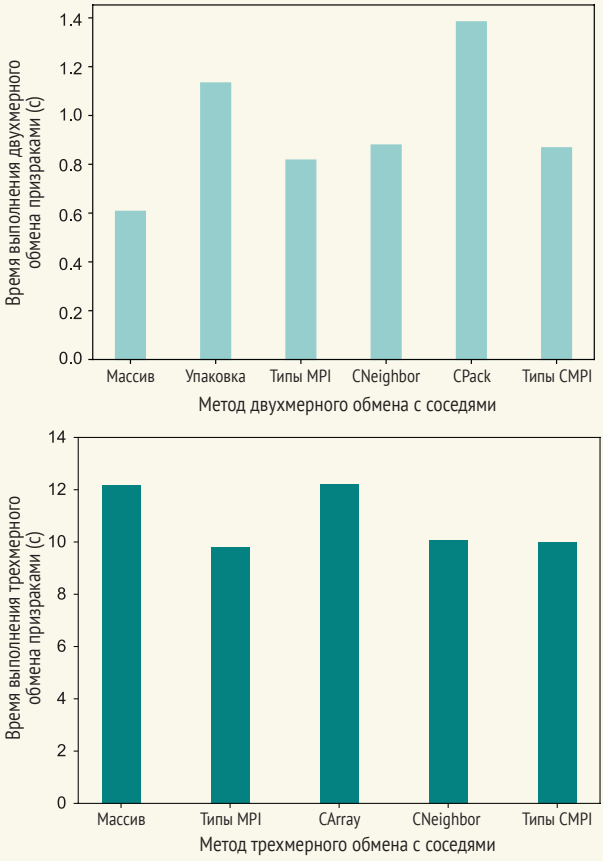
Прилагаемый исходный код настроен для выполнения всего набора вариантов обмена с призрачными ячейками. В файле batch.sh можно изменить размер ореола и задействовать углы или нет. Файл настроен для выполнения всех тестовых случаев 11 раз на двух узлах Skylake Gold с общим числом процессов 144.

```
cd GhostExchange
./build.sh
./batch.sh |& tee results.txt
./get_stats.sh > stats.out
```

Затем вы можете генерировать графики с помощью предоставленных скриптов. Для скриптов построения графиков на Python вам нужна библиотека matplotlib. Результаты приведены для среднего времени выполнения.

```
python plottimebytype.py
python plottimeby3Dtype.py
```

На следующем ниже рисунке показаны графики для малых тестовых случаев. Версии типа данных MPI выводятся немного быстрее даже в этом малом масштабе, что указывает на то, что, возможно, следует избегать копирования данных. Для вызовов декартовой топологии MPI и типов данных MPI больший выигрыш заключается в том, что код обмена с призраками значительно упрощен. Использование этих более продвинутых вызовов MPI требует дополнительных усилий по настройке, но это делается только один раз при запуске.



Относительная производительность двухмерных и трехмерных призрачных обменов на двух узлах с суммарным числом процессов 144. Типы данных MPI в типах MPI и CNeighbor немного быстрее, чем буфер, явно заполненный циклами присваиваний в массивах. В двухмерных призрачных обменах процедуры упаковки выполняются медленнее, хотя в этом случае явно заполненный буфер работает быстрее, чем типы данных MPI

8.6 Гибридная техника MPI плюс OpenMP для максимальной масштабируемости

Комбинация двух или более техник параллелизации называется гибридной параллелизацией, в отличие от всех имплементаций MPI, которые также называются чистым MPI или MPI-везде (MPI-everywhere). В этом разделе мы рассмотрим гибридную технику MPI плюс OpenMP, где MPI и OpenMP используются в приложении вместе. Обычно это сводится к замене некоторых рангов MPI потоками OpenMP. Для более крупных параллельных приложений, охватывающих тысячи процессов, замена рангов MPI потоками OpenMP потенциально уменьшает суммарный размер домена MPI и объем памяти, необходимый для экстремального масштаба. Однако добавочная производительность слоя параллелизма уровня потоков, возможно, не всегда будет оправдывать добавочную сложность и время разработки. По этой причине гибридные имплементации в стиле MPI плюс OpenMP обычно являются областью экстремальных приложений как по размеру, так и по производительности.

8.6.1 Преимущества гибридной техники MPI плюс OpenMP

Когда производительность становится достаточно критичной для добавочной сложности гибридного параллелизма, могут существовать несколько преимуществ добавления параллельного слоя OpenMP в исходный код на основе MPI. Например, эти преимущества могут заключаться в следующем:

- меньше призрачных ячеек для обмена между узлами;
- более низкие требования к памяти для буферов MPI;
- снижение соревновательности за сетевой адаптер;
- уменьшенный размер древовидных обменов данными;
- улучшенная балансировка нагрузки;
- доступ ко всем аппаратным компонентам.

Пространственно разложенные параллельные приложения, в которых используются поддомены с призраковыми ячейками (ореолами), будут иметь меньше суммарных призраковых ячеек на узел при добавлении параллелизма уровня потоков. Это приводит к снижению как требований к памяти, так и затрат на обмен, в особенности в многоядерной архитектуре, такой как Intel Knights Landing (KNL). Использование параллелизма с совместной памятью также может повысить производительность за счет уменьшения соревновательности за сетевую интерфейсную карту (NIC), избегая ненужного копирования данных, используемых MPI для наузловых сообщений. Кроме того, многие алгоритмы MPI основаны на дереве и масштабируются как $\log_2 n$. Сокращение времени выполнения на $2n$ потоков уменьшает глубину дерева и постепенно повышает производительность. Хотя оставшаяся работа все еще должна выполняться потоками, это влияет на производительность, позволяя снижать затраты

на синхронизацию и задержку обмена. Потоки также можно использовать для улучшения баланса нагрузки внутри участка NUMA или вычислительном узле.

В некоторых случаях гибридный параллельный подход не только выгоден, но и необходим для доступа к полному потенциалу производительности оборудования. Например, некоторые аппаратные средства и, возможно, функциональность контроллера памяти доступны только потокам, а не процессам (рангам MPI). Эти проблемы наблюдались у многоядерных архитектур Intel Knights Corner и архитектур Knights Landing. В *MPI + X + Y*, где *X* – это потокообразование, а *Y* – язык GPU, мы часто соотносим ранги с числом GPU-процессоров. OpenMP позволяет приложению продолжать получать доступ к другим процессорам для работы на CPU. Для этого существуют и другие решения, такие как группы MPI_COMM и функциональность совместной для MPI памяти или простое управление GPU из нескольких рангов MPI.

Таким образом, хотя в современных многоядерных системах бывает привлекательным выполнять исходные коды с помощью технологии MPI-везде, существуют опасения в отношении масштабируемости по мере роста числа ядер. Если вы ищете экстремальную масштабируемость, то вам в вашем приложении потребуется эффективная имплементация OpenMP. Мы рассмотрели наш намного более эффективный проект OpenMP высокого уровня в предыдущей главе в разделах 7.2.2 и 7.6.

8.6.2 Пример техники MPI плюс OpenMP

Первые шаги к имплементации гибридной техники MPI плюс OpenMP состоят в предоставлении сведений о том, что вы будете делать. Это делается в вызове `MPI_Init` в самом начале программы. Вы должны заменить вызов `MPI_Init` вызовом `MPI_Init_thread` следующим образом:

```
MPI_Init_thread(&argc, &argv, int thread_model required,
                int *thread_model_provided);
```

Стандарт MPI определяет четыре модели потоков. Эти модели обеспечивают разные уровни потокобезопасности при вызовах MPI. В порядке возрастания потокобезопасности:

- `MPI_THREAD_SINGLE` – выполняется только один поток (стандартный MPI);
- `MPI_THREAD_FUNNELED` – многопоточно, но только главный поток выполняет вызовы MPI;
- `MPI_THREAD_SERIALIZED` – многопоточно, но только один поток одновременно выполняет вызовы MPI;
- `MPI_THREAD_MULTIPLE` – многопоточно, несколько потоков выполняют вызовы MPI.

Многие приложения выполняют обмен на уровне главного цикла, и потоки OpenMP применяются к ключевым вычислительным циклам. Для этого шаблона `MPI_THREAD_FUNNELED` работает просто отлично.

ПРИМЕЧАНИЕ Лучше всего использовать самый низкий уровень потокобезопасности, который вам нужен. Каждый более высокий уровень налагает штраф за производительность, поскольку библиотека MPI должна размещать мьютексы или критические блоки вокруг очередей отправки и приемки и других базовых частей MPI.

Теперь давайте посмотрим на изменения, которые необходимы в нашем стенсильном примере, чтобы добавить потокообразование OpenMP. В этом упражнении мы для модифицирования выбрали пример CartExchange_Neighbor. В следующем ниже листинге показано, что первым изменением является модифицирование инициализации MPI.

Листинг 8.33 Инициализация MPI для потокообразования OpenMP

HybridMPIPlusOpenMP/CartExchange.cc

```
26 int provided;
27 MPI_Init_thread(&argc, &argv,
      MPI_THREAD_FUNNELED, &provided);
28
29 int rank, nprocs;
30 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
31 MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
32 if (rank == 0) {
33    #pragma omp parallel
34    #pragma omp master
35       printf("запрашивается MPI_THREAD_FUNNELED"
                 " с %d потоками\n",
36              omp_get_num_threads());
37    if (provided != MPI_THREAD_FUNNELED){
38       printf("Ошибка: MPI_THREAD_FUNNELED"
                 " отсутствует. Выполнение абортируется ...\n");
39       MPI_Finalize();
40       exit(0);
41    }
42 }
```

Инициализация MPI для потокообразования OpenMP

Печатает число потоков, чтобы проверить то, что мы хотим

Проверяет, что этот MPI поддерживает требуемый уровень потокобезопасности

Обязательным изменением является использование `MPI_Init_thread` вместо `MPI_Init` в строке 27. Дополнительный код проверяет доступность запрошенного уровня потокобезопасности, и завершает работу, если это не так. Мы также печатаем число потоков в главном потоке нулевого ранга. Теперь перейдем к изменениям в вычислительном цикле, показанном в следующем ниже листинге.

Листинг 8.34 Добавление потокообразования OpenMP и векторизации в вычислительные циклы

HybridMPIPlusOpenMP/CartExchange.cc

```
157 #pragma omp parallel for
```

Добавляет потокообразование OpenMP во внешний цикл

```
158 for (int j = 0; j < jsize; j++){
159    #pragma omp simd
160    for (int i = 0; i < isize; i++){
161       xnew[j][i] = ( x[j][i] + x[j][i-1] + x[j][i+1]
                              + x[j-1][i] + x[j+1][i] )/5.0;
162    }
163 }
```

Добавляет векторизацию SIMD во внутренний цикл

Изменения, необходимые для добавления потокообразования OpenMP, заключаются в добавлении одной прагмы в строке 157. В качестве бонуса мы покажем, как добавлять векторизацию во внутренний цикл с помощью еще одной прагмы, вставленной в строку 159.

Теперь вы можете попробовать выполнить этот пример гибрида MPI плюс OpenMP+Векторизации в своей системе. Но в целях получения хорошей производительности вам нужно будет контролировать размещение рангов MPI и потоков OpenMP. Это делается путем задания аффинности – эту тему мы рассмотрим подробнее в главе 14.

ОПРЕДЕЛЕНИЕ *Аффинность* отдает предпочтение тому или иному аппаратному компоненту в планировании процесса, ранга или потока. Она также называется *закреплением* или *привязкой*.

Настройка соответствия для ваших рангов и потоков становится важнее по мере увеличения сложности узла и в гибридных параллельных приложениях. В предыдущих примерах мы использовали `--bind-to core` и `--bind-to hwthread` для повышения производительности и сокращения вариативности производительности во время выполнения, вызываемой миграцией рангов из одного ядра в другое. В OpenMP мы использовали переменные среды для задания мест размещения и аффинностей. Примером может служить:

```
export OMP_PLACES=cores
export OMP_CPU_BIND=true
```

А пока начнем с закрепления рангов MPI в разъемах, чтобы потоки могли распространяться на другие ядра, как мы показали в нашем примере тестирования призрачной ячейки для процесса Skylake Gold. Вот как это делается:

```
export OMP_NUM_THREADS=22
mpirun -n 4 --bind-to socket ./CartExchange -x 2 -y 2 -i 20000 -j 20000 \
-h 2 -t -c
```

Мы выполняем 4 ранга MPI, каждый из которых порождает 22 потока, как указано в переменной среды `OMP_NUM_THREADS`, в общей сложности для 88 процессов. Опция `--bind-to socket` для `mpirun` предписывает ему привязывать процессы к разъему, в котором они размещены.

8.7 Материалы для дальнейшего изучения

Хотя в этой главе мы охватили много материала, есть еще много других функций, которые стоит разведать, когда вы получите больше опыта работы с MPI. Несколько наиболее важных из них упомянуты здесь и оставлены для вашего собственного изучения.

- *Коммуникационные (comm) группы* – MPI имеет богатый набор функций, которые создают, разделяют и иным образом манипулируют стандартным коммуникатором MPI COMM_WORLD, преобразуя в новые группы для специализированных операций, таких как обмен внутри строки или подгрупп на основе операционных задач. Некоторые примеры использования групп коммуникаторов см. в листинге 16.4 в разделе 16.3. Мы используем группы обмена, чтобы разделять вывод файла на несколько файлов и разбивать домен на построчные и постолбцовые коммуникаторы.
- *Обмен данными через границы неструктурированной сетки* – неструктурированная сетка должна обмениваться граничными данными аналогично тому, как это предусмотрено для обычной декартовой вычислительной сетки. Эти операции более сложны и здесь не рассматриваются. Существует целый ряд коммуникационных библиотек на основе разреженных графов, которые поддерживают приложения с использованием неструктурированных сеток. Одним из примеров такой библиотеки является коммуникационная библиотека L7, разработанная Ричардом Барреттом в Национальных лабораториях Сандии. Она входит в состав мини-приложения CLAMR; обратитесь к подкаталогу l7 по адресу https://github.com/LANL/CLAMR.
- *Совместная память* – исходные имплементации MPI почти во всех случаях отправляли данные по сетевому интерфейсу. По мере роста числа ядер разработчики MPI осознали, что они могут осуществлять часть обмена в совместной памяти. Это делается за кулисами в качестве оптимизации обмена. Дополнительные функции сомместной памяти по-прежнему добавляются с помощью «окон» совместной памяти MPI. В этой функции сначала было несколько проблемы, но она становится достаточно зрелой для использования в приложениях.
- *Односторонний обмен* – в ответ на другие модели программирования MPI добавил односторонний обмен в виде `MPI_Puts` и `MPI_Gets`. В отличие от исходной модели передачи сообщений MPI, в которой и отправитель, и получатель должны быть активными участниками, односторонняя модель позволяет выполнять операцию только одному или другому из двух.

8.7.1 Дополнительное чтение

Если вам нужны дополнительные вводные материалы по MPI, то учебник Питера Пачеко будет здесь классическим:

- Питер Пачеко, «Введение в параллельное программирование» (Peter Pacheco, *An introduction to parallel programming*, Elsevier, 2011).

Подробное освещение MPI, написанное членами изначальной команды разработчиков MPI, можно найти в статье:

- Уильям Гропп и соавт., «Использование MPI: переносное параллельное программирование с интерфейсом передачи сообщений» (William Gropp, et al., Using MPI: portable parallel programming with the message-passing interface, Vol. 1, MIT Press, 1999).

Презентация техники MPI плюс OpenMP есть в неплохой лекции из курса Билла Гроппа (Bill Gropp), одного из разработчиков изначального стандарта MPI. Вот ссылка:

- http://wgropp.cs.illinois.edu/courses/cs598-s16/lectures/lecture36.pdf.

8.7.2 Упражнения

1. Почему мы не можем просто блокировать на приемках, как это делалось при отправке/приемке в обмене призраками, используя методы упаковочного буфера или массива соответственно в листингах 8.20 и 8.21?
2. Безопасно ли блокировать приемку, как показано в листинге 8.8 в версии обмена призраками с использованием векторного типа? В чем преимущества, если мы блокируем только приемку?
3. Модифицируйте пример обмена призрачными ячейками с векторным типом в листинге 8.21, чтобы использовать блокировку приемки вместо ожидания. Будет ли это быстрее? Всегда ли это будет работать?
4. Попробуйте заменить явные теги в одной из подпрограмм обмена призраками на `MPI_ANY_TAG`. Будет ли это работать? Будет ли это хоть немного быстрее? Какое преимущество вы видите в использовании явных тегов?
5. Удалите барьеры для синхронизированных таймеров в одном из примеров обмена призраками. Выполните исходный код с изначальными синхронизированными таймерами и несинхронизированными таймерами.
6. Добавьте статистику таймера из листинга 8.11 в исходный код измерения пропускной способности потоковой триады в листинге 8.17.
7. Примените шаги для конвертирования OpenMP высокого уровня в пример гибридной техники MPI плюс OpenMP в исходном коде, прилагаемом к главе (каталог HybridMPIPlusOpenMP). Поэкспериментируйте с векторизацией, числом потоков и рангами MPI на вашей платформе.

Резюме

- Используйте правильные отправочные и приемочные сообщения «из точки в точку». Это позволяет избегать зависаний и обеспечивает хорошую производительность.

- Используйте коллективный обмен данными для общепринятых операций. Это обеспечивает лаконичное программирование, позволяет избегать зависаний и повышает производительность.
- Используйте обмен призраками для связывания поддоменов из различных процессоров. Обмены данными заставляют поддомены действовать как единая глобальная вычислительная сетка.
- Добавляйте больше уровней параллелизма за счет объединения MPI с потокообразованием OpenMP и векторизации. Дополнительный параллелизм помогает повышать производительность.

Часть III

GPU: рождены для ускорения

В следующих главах, посвященных вычислениям на GPU, обсуждается использование GPU-процессоров для научных вычислений. Будут затронуты следующие темы:

- в главе 9 вы получите представление об архитектуре GPU и ее преимуществах для общецелевых вычислений;
- в главе 10 вы узнаете, как выстраивать мысленное представление модели программирования для GPU-процессоров;
- в главах 11 и 12 вы разведуете существующие языки программирования на GPU. В главе 11 мы приводим базовые примеры в OpenACC и OpenMP, а в главе 12 рассмотрим широкий спектр языков GPU, от нативных языков более низкого уровня, таких как CUDA, OpenCL и HIP, до языков более высокого уровня, таких как SYCL, Kokkos и Raja;
- в главе 13 вы узнаете об инструментах профилирования и разработке модели рабочего потока, которая повышает производительность программиста.

GPU-процессоры были построены для ускорения вычислений. С целеустремленным вниманием на улучшении частоты кадров для компьютерной анимации разработчики аппаратного обеспечения GPU пошли на многое, чтобы увеличить пропускную способность числовых операций. Эти устройства представляют собой просто общецелевые экстре-

мальные ускорители любой массово-параллельной операции. Они называются графическими процессорами, потому что они были разработаны для этого применения.

Перенесемся в сегодняшний день, когда многие разработчики программно-информационного обеспечения поняли, что ускорение, обеспечиваемое GPU-процессорами, одинаково применимо для широкого спектра областей применения. В 2002 году, работая в Университете Северной Каролины, Марк Харрис (Mark Harris) ввел термин «общецелевые графические процессоры» (general-purpose graphics processing units, GPGPUs), чтобы попытаться уловить идею о том, что GPU-процессоры подходят не только для графики. Возникли крупные рынки GPU-процессоров для майнинга биткоинов, машинного обучения и высокопроизводительных вычислений. Небольшие модификации оборудования GPU, такие как устройства с плавающей точкой двойной точности и тензорные операции, позволили приспособить базовый дизайн GPU для каждого из этих рынков. GPU-процессоры больше не предназначены только для графики.

Каждая выпущенная модель GPU ориентирована на другой сегмент рынка. С высококачественными моделями GPU по ценам до 10 000 долл. это не те модели оборудования, которые вы найдете на компьютерах массового рынка. Хотя GPU-процессоры используются во многих приложениях больше, чем они изначально предназначались, по-прежнему трудно представить, что они в ближайшем будущем полностью заменят общецелевую функциональность CPU, поскольку одна единая операция лучше подходит для CPU.

Если бы мы придумывали новое название для GPU-процессоров, то как бы мы описали их функциональность в широком смысле? Общим во всех сферах использования является то, что для того, чтобы имело смысл использовать GPU-процессоры, должен иметься большой объем работы, которую можно выполнять одновременно. И под большим объемом мы подразумеваем тысячи или десятки тысяч одновременных параллельных операций. GPU-процессоры – это по-настоящему *параллельные ускорители*. Может быть, в целях более глубокого понимания их функциональности нам следует назвать их модулями параллельной обработки (PPU) или ускорителями параллельной обработки (PPA). Но мы будем придерживаться термина «графические процессоры» (GPU) с пониманием того, что они представляют собой нечто намного большее. Глядя на GPU-процессоры в этом свете, вы сможете понять, почему они так важны для сообщества параллельных вычислений.

GPU-процессоры проще в конструировании и производстве, чем CPU-процессоры, поэтому продолжительность цикла конструирования вдвое меньше, чем у CPU-процессоров. Точкой пересечения для многих приложений с точки зрения производительности GPU, по сравнению с CPU-процессорами, был примерно 2012 год. С тех пор производительность GPU улучшилась примерно в два раза, чем у CPU-процессоров. В очень приблизительных цифрах GPU-процессоры сегодня могут обеспечивать 10-кратное ускорение по сравнению с CPU-процессорами. Конечно, это

ускорение сильно зависит от типа приложения и качества имплементации кода. Тренды очевидны – GPU-процессоры будут продолжать демонстрировать большее ускорение в тех приложениях, которые могут соответствовать его массивно параллельной архитектуре.

Для того чтобы помочь вам понять эти новые аппаратные устройства, в главе 10 мы пройдемся по существенным частям их аппаратного обеспечения. Затем мы попытаемся помочь вам разработать ментальную модель того, как с ними обращаться. Очень важно иметь это понимание перед тем, как приступать к проекту для GPU-процессоров. Мы видели много безуспешных попыток портирования, потому что программисты думали, что они смогут просто перенести самый дорогостоящий цикл на GPU и увидят фантастические ускорения. Но потом, когда оказывается, что их приложение работает медленнее, они отказываются от усилий. Передача данных на GPU обходится дорого; следовательно, большая часть вашего приложения должна быть перенесена на устройство, чтобы увидеть какую-либо выгоду. Простая модель производительности и анализ перед имплементацией GPU умерили бы первоначальные ожидания программистов и предупредили бы их о необходимости планировать с учетом достаточного количества времени и усилий, чтобы достичь успеха.

Возможно, самым большим препятствием на пути к имплементации GPU является постоянно меняющийся ландшафт языков программирования. Похоже, что новый язык выходит каждые несколько месяцев. Хотя эти языки обеспечивают высокую степень инноваций, такая постоянная эволюция затрудняет разработку приложений. Однако более пристальный взгляд на языки показывает, что сходств больше, чем различий, и часто они сходятся в нескольких общепринятых конструктах. В то время как мы ожидаем еще несколько лет языковой борьбы, многие языковые вариации сродни диалектам, а не совершенно разным языкам. В главе 11 мы рассмотрим прагма-ориентированные языки. В главе 12 мы сделаем обзор нативных языков GPU и нового класса языков обеспечения переносимости производительности, в которых в значительной степени используются конструкты C++. Хотя мы представляем самые разные языковые имплементации, мы предлагаем вам сначала выбрать пару языков, чтобы получить практический опыт работы с ними. Во многом ваше решение о том, с какими языками экспериментировать, будет зависеть от оборудования, которое у вас есть в наличии.

Вы можете ознакомиться с примерами по адресу https://github.com/EssentialsofParallelComputing для каждой упомянутой выше главы. (На самом деле мы настоятельно рекомендуем вам это сделать.) Одним из препятствий для программирования GPU является доступ к оборудованию и его правильная настройка. Инсталляция системного программного обеспечения для поддержки GPU-процессоров иногда бывает сложной. В примерах приведены списки пакетов программно-информационного обеспечения для GPU-процессоров от поставщиков, которые помогут вам начать работу. Но вы захотите инсталлировать программно-информационное обеспечение для GPU в своей системе и в примерах

закомментировать остальное. Это потребует некоторого экспериментирования методом проб и ошибок. В главе 13 мы обсудим разные рабочие потоки и альтернативы, такие как настройка контейнеров Docker и виртуальных машин. Эти опции помогут настроить среду разработки на вашем ноутбуке или настольном компьютере, в особенности если вы используете Windows или macOS.

Если у вас нет доступного локального оборудования, то можете попробовать облачную службу с GPU-процессорами. Некоторые из них, такие как Google Cloud (с кредитом в размере 200–300 долл. США), даже имеют бесплатные пробные версии. В этих службах даже есть коммерческие надстройки, которые позволяют поднимать кластер HPC с GPU-процессорами. Одним из примеров облачной службы HPC с GPU-процессорами является облачная платформа Google Fluid Numerics. Для тестирования GPU-процессоров Intel компания Intel также организовала облачные службы для пробных служб. За получением дополнительной информации мы рекомендуем обратиться к следующим ниже веб-сайтам:

- облачный кластер Google Fluid-Slurm по адресу https://console.cloud.google.com/marketplace/details/fluid-cluster-ops/fluid-slurm-gcp;
- облачная версия Intel oneAPI и DPCPP по адресу https://software.intel.com/en-us/oneapi (для ее использования необходимо зарегистрироваться).

9 Архитектуры и концепции GPU

Эта глава охватывает следующие ниже темы:

- понимание оборудования GPU и связанных с ним компонентов;
- оценивание теоретической производительности вашего GPU;
- измерение производительности вашего GPU;
- разные применения приложений для эффективного использования GPU.

Почему нас должны интересовать GPU-процессоры в том, что касается высокопроизводительных вычислений? GPU-процессоры обеспечивают массивный источник параллельных операций, который может значительно превышать тот, что доступен в более традиционной архитектуре CPU. В целях эффективного использования их способностей важно понимать архитектуру GPU. Хотя GPU-процессоры очень часто использовались для обработки графики, они также используются для общецелевых параллельных вычислений. В данной главе представлен обзор оборудования на платформе, ускоренной за счет GPU-процессоров.

Какие системы сегодня относятся к системам, ускоренным за счет GPU-процессоров? Практически каждая вычислительная система обеспечивает мощные графические способности, ожидаемые современными пользователями. Эти GPU-процессоры варьируются от малых компонентов главного CPU до больших периферийных плат, занимающих

немалую часть пространства в корпусе настольного компьютера. Системы HPC все чаще оснащаются несколькими GPU. От случая к случаю даже персональные компьютеры, используемые для симуляции или игр, иногда могут подключать два GPU для повышения производительности графики. В этой главе мы представим концептуальную модель, которая определяет ключевые аппаратные компоненты системы, ускоренной за счет GPU-процессоров. Эти компоненты показаны на рис. 9.1.

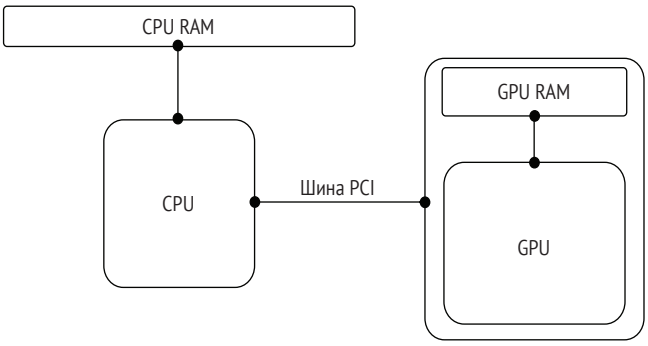


Рис. 9.1 Структурная схема системы, ускоренной за счет GPU, с использованием выделенного GPU. CPU и GPU имеют свою собственную память. Они взаимодействуют по шине PCI

Из-за несогласованности используемой в сообществе терминологии возникает дополнительная сложность в понимании GPU-процессоров. Мы будем использовать терминологию, установленную стандартом OpenCL, поскольку она была согласована несколькими поставщиками GPU. Мы также отметим альтернативную терминологию, которая широко используется, например, в NVIDIA. Прежде чем продолжить наше обсуждение, давайте взглянем на несколько определений:

- *CPU* – главный процессор, установленный в разъеме материнской платы;
- *CPU RAM* – «карты памяти», или модули памяти, с двухрядным расположением выводов (Dual In-line Memory Module, DIMM), содержащие динамическую память с произвольным доступом (Dynamic Random-Access Memory, DRAM), которые вставляются в слоты памяти на материнской плате;
- *GPU* – большая периферийная карта, устанавливаемая в разъем шины PCI Express (Peripheral Component Interconnect Express, PCIe, или экспресс-шины взаимодействия периферийных компонентов) на материнской плате;
- *GPU RAM* – модули памяти на периферийной карте GPU для исключительного ее использования GPU-процессором;
- *шина PCI* – проводка, соединяющая периферийные платы с другими компонентами на материнской плате.

Мы познакомим с каждым компонентом в системе, ускоренной за счет GPU, и покажем, как рассчитывать теоретическую производитель-

ность для каждого из них. Затем мы проинспектируем их фактическую производительность с помощью приложений сравнительного микротестирования (микробенчмарков). Это поможет разобраться в том, каким образом некоторые аппаратные компоненты могут создавать узкие места, которые не позволяют ускорять приложение с помощью GPU-процессоров. Вооружившись этой информацией, мы завершим главу обсуждением видов приложений, которые выигрывают от ускорения за счет GPU больше всего, и того, какими должны быть ваши цели, чтобы увидеть повышение производительности при переносе приложения для работы на GPU-процессорах. Исходный код этой главы можно найти по адресу https://github.com/EssentialsofParallelComputing/Chapter9.

9.1 Система CPU-GPU как ускоренная вычислительная платформа

GPU-процессоры есть везде. Их можно найти в мобильных телефонах, планшетах, персональных компьютерах, рабочих станциях потребительского класса, игровых консолях, центрах высокопроизводительных вычислений и платформах облачных вычислений. GPU-процессоры обеспечивают дополнительную вычислительную мощность на большинстве современных аппаратных средств и ускоряют многие операции, о которых вы, возможно, даже не подозреваете. Как следует из названия, GPU-процессоры были разработаны для вычислений, связанных с графикой. Следовательно, дизайн GPU сосредоточен на параллельной обработке больших блоков данных (треугольников или полигонов), наличие которой является непременным условием графических приложений. По сравнению с CPU, которые могут обрабатывать десятки параллельных потоков или процессов за тактовый цикл, GPU-процессоры способны обрабатывать тысячи параллельных потоков одновременно. Благодаря такому дизайну GPU-процессоры обеспечивают значительно более высокую теоретическую пиковую производительность, потенциально способную сократить показатель времени до решения и энергоотпечаток приложения.

Специалисты в области компьютерных вычислений, всегда стремившиеся к вычислительной мощности, тяготели к GPU-процессорам, способным выполнять вычислительные задачи более общего назначения. Поскольку GPU-процессоры были разработаны для графики, языки, изначально разработанные для их программирования, такие как OpenGL, были ориентированы на операции с графикой. В целях имплементирования алгоритмов на GPU-процессорах программистам приходилось перестраивать свои алгоритмы с точки зрения этих операций, что отнимало много времени и приводило к ошибкам. Распространение GPU-процессоров на неграфические рабочие нагрузки стало называться вычислениями на общецелевых графических процессорах (general-purpose graphics processing unit, GPGPU).

Продолжающийся интерес и успех вычислений на GPGPU привели к появлению шквала языков GPGPU. Первым языком, получившим широкое распространение, был язык программирования CUDA (Compute Unified Device Architecture) для GPU-процессоров NVIDIA, который был впервые представлен в 2007 году. Доминирующим открытым стандартным языком вычислений на GPGPU является открытый язык вычислений (Open Computing Language, OpenCL), разработанный группой поставщиков во главе с Apple и выпущенный в 2009 году. Мы рассмотрим CUDA и OpenCL в главе 12.

Несмотря на постоянное внедрение языков GPGPU или, возможно, по этой причине, многие специалисты в области компьютерных вычислений обнаружили, что изначальные, нативные языки GPGPU трудно использовать. Как следствие, большое распространение получили подходы более высокого уровня с использованием директивоориентированного API, и этот факт подтолкнул соответствующие усилия поставщиков по их разработке. Мы рассмотрим примеры языков на базе директив, таких как OpenACC и OpenMP (с новой директивой target), в главе 11. А пока можем просто резюмировать, что новые языки GPGPU на базе директив, OpenACC и OpenMP, имели безусловный успех. Эти языки и API позволили программистам больше сосредотачиваться на разработке своих приложений, а не на выражении своего алгоритма в терминах графических операций. В итоге это нередко в результате приводило к огромному ускорению в научных приложениях и приложениях по исследованию данных.

GPU-процессоры лучше всего характеризовать как ускорители, давно используемые в вычислительном мире. Сначала давайте дадим определение тому, что мы подразумеваем под ускорителем.

ОПРЕДЕЛЕНИЕ *Ускоритель* (или ускорительное оборудование) – это узкоспециальное устройство, которое поддерживает главный общецелевой CPU в части ускорения некоторых операций.

Классическим примером ускорителя является изначальный персональный компьютер, поставлявшийся с процессором 8088. У него была опция и разъем для сопроцессора 8087, который мог выполнять операции с плавающей точкой не на программном, а на аппаратном уровне. Сегодня наиболее распространенным аппаратным ускорителем является GPU, который может быть либо отдельным аппаратным компонентом, либо интегрирован в главный CPU. Ускоритель отличается тем, что он имеет узкоспециальную направленность, не являясь общецелевым устройством, но это отличие не всегда четко выражено. GPU представляет собой дополнительный аппаратный компонент, который может выполнять операции вместе с CPU. GPU-процессоры бывают двух разновидностей:

- *интегрированные GPU* – GPU, которые содержатся на CPU;
- *выделенные GPU* – GPU, размещенные на отдельной периферийной карте.

Интегрированные GPU встроены непосредственно в чип CPU. Интегрированные GPU используют ресурсы RAM (оперативной памяти) совместно с CPU. Выделенные GPU прикрепляются к материнской плате через разъем PCI (Peripheral Component Interconnect), служащий для подсоединения периферийных компонентов. Разъем PCI – это физический компонент, который позволяет передавать данные между CPU и GPU. Он обычно называется шиной PCI.

9.1.1 Интегрированные GPU: недоиспользуемая опция в товарных системах

Intel® уже давно включает интегрированный GPU в состав своих CPU, предназначенных для бюджетного рынка. Они в полной мере рассчитывали на то, что пользователи, желающие реальной производительности, будут покупать дискретный GPU. Интегрированные GPU Intel исторически были относительно слабыми по сравнению с интегрированной версией AMD (Advanced Micro Devices, Inc.). Это недавно изменилось, когда Intel заявила, что интегрированная графика в их процессоре Ice Lake соответствует интегрированным GPU AMD.

Интегрированные GPU компании AMD называются ускоренными процессорами (Accelerated Processing Unit, APU). Они представляют собой тесно состыкованную комбинацию CPU и GPU. Источником дизайна GPU изначально послужила покупка компанией AMD компании по производству видеокарт ATI в 2006 году. CPU и GPU в ускоренном процессоре (APU) AMD задействуют одну и ту же процессорную память. Эти GPU меньше, чем дискретные GPU, но при этом (пропорционально) обеспечивают графическую (и вычислительную) производительность GPU-процессора. Реальная цель AMD в отношении APU состоит в том, чтобы предлагать более экономичную, но производительную систему для массового рынка. Совместная память также привлекательна тем, что она устраняет передачу данных по шине PCI, что нередко является серьезным узким местом производительности.

Повсеместная природа интегрированного GPU имеет важное значение. Для нас это означает, что теперь многие товарные настольные компьютеры и ноутбуки способны ускорять вычисления. Цель этих систем состоит в относительно скромном повышении производительности и, возможно, снижении энергозатрат либо увеличения срока службы батареи. Но если уж говорить об экстремальной производительности, то дискретные GPU по-прежнему остаются бесспорными чемпионами по производительности.

9.1.2 Выделенные GPU: рабочая лошадка

В этой главе мы в первую очередь сосредоточимся на ускоренных за счет GPU платформах с выделенными GPU, также именуемыми *дискретными GPU*. Выделенные GPU обычно обеспечивают более высокую вычис-

лительную мощность, чем встроенные GPU. Кроме того, эти GPU могут быть изолированы для выполнения общецелевых вычислительных задач. Рисунок 9.1 концептуально иллюстрирует систему CPU-GPU с выделенным GPU. CPU имеет доступ к собственному пространству памяти (RAM-памяти в CPU) и соединен с GPU через шину PCI. Он способен отправлять данные и команды по шине PCI для работы с GPU. Последний имеет свое собственное пространство памяти, отдельное от пространства памяти CPU.

В целях выполнения работы на GPU в какой-то момент данные должны быть переданы с CPU на GPU. Когда работа будет завершена и результаты будут готовы для записи в файл, GPU должен отправить данные обратно в CPU. Команды, которые GPU должен выполнить, также передаются из CPU в GPU. Каждая из этих транзакций опосредуется шиной PCI. Хотя в этой главе мы не будем обсуждать подходы к выполнению этих действий, мы все же обсудим аппаратные пределы производительности шины PCI. Вследствие этих ограничений плохо сконструированное приложение на базе GPU может иметь потенциально меньшую производительность, чем при использовании кода для работы только с CPU. Мы также обсудим внутреннюю архитектуру GPU и производительность GPU в отношении памяти и операций с плавающей точкой.

9.2 GPU и двигатель потокообразования

Для тех из нас, кто на протяжении многих лет занимался программированием потоков (aka виртуальных ядер) на CPU, GPU подобен идеальному двигателю потокообразования. Компоненты этого двигателя таковы:

- кажущееся бесконечным число потоков;
- нулевые затраты времени на переключение или запуск потоков;
- скрытие задержки доступа к памяти за счет автоматического переключения между рабочими группами.

Давайте рассмотрим аппаратную архитектуру GPU, чтобы получить представление о том, как он творит это волшебство. В целях иллюстрации концептуальной модели GPU мы абстрагируем элементы, общие для разных поставщиков GPU-процессоров и даже общие между вариантами дизайна от одного и того же поставщика. Следует напомнить, что некоторые аппаратные вариации не улавливаются этими абстрактными моделями. Добавляя к этому обилию используемой в настоящее время терминологии, неудивительно, что новичку в данной области трудно разобраться в оборудовании и языках программирования GPU. Тем не менее эта терминология относительно здравая по сравнению с графическим миром с вершинными шейдерами, модулями соотнесения текстур и генераторами фрагментов. В табл. 9.1 резюмирована приблизительная эквивалентность терминологии, но имейте в виду, что, поскольку аппаратные архитектуры не совсем одинаковы, соответствие в терминологии варьируется в зависимости от контекста и пользователя.

Таблица 9.1 Терминология оборудования: грубое соотнесение

Хост	OpenCL	AMD GPU	NVIDIA/CUDA	Intel Gen11
CPU	Вычислительное устройство	GPU	GPU	GPU
Мультипроцессор	Вычислительный модуль (CU)	Вычислительный модуль (CU)	Потоковый мультипроцессор (streaming multiprocessor, SM)	Подсрез
Обрабатывающее ядро (processing core, или просто ядро)	Обрабатывающий элемент (processing unit, PE)	Обрабатывающий элемент (processing unit, PE)	Вычислительные ядра или CUDA-ядра	Исполнительные модули (Execution units, EU)
Поток (thread или virtual core)	Элемент работы (Work Item)	Элемент работы	Поток	
Вектор или модель SIMD	Вектор	Вектор	Эмулирование с помощью SIMT-варпов[1]	SIMD

В последней строке табл. 9.1 показан аппаратный уровень, на котором имплементирована модель «одна команда и несколько данных», обычно именуемая SIMD[2]. Строго говоря, аппаратное обеспечение NVIDIA не имеет векторного оборудования или SIMD, но эмулирует его с помощью коллекции потоков в том, что в NVIDIA называется варпом в модели «одна команда, мультипоток» (SIMT). Возможно, вам захочется вернуться к нашему первоначальному обсуждению параллельных категорий в разделе 1.4, чтобы освежить в памяти эти разные подходы. Другие GPU способны выполнять операции SIMT и на том, что в OpenCL и AMD называется подгруппами, эквивалентными варпам NVIDIA. Мы обсудим эту тему подробнее в главе 10, в которой подробно рассматриваются модели программирования на GPU. Однако в этой главе основное внимание будет уделено оборудованию GPU, его архитектуре и концепциям.

Нередко GPU-процессоры также имеют аппаратные репликационные модули, часть из которых перечислена в табл. 9.2, чтобы упрощать масштабирование их аппаратных конструкций до большего числа модулей. Эти репликационные модули удобны в производстве и часто появляются в списках спецификаций и обсуждениях.

Таблица 9.2 Репликационные модули оборудования GPU по поставщикам

AMD	NVIDIA/CUDA	Intel Gen11
Шейдерный двигатель (Shader Engine, SE)	Кластер обработки графики	Срез

На рис. 9.2 показана упрощенная структурная схема одноузловой системы с одним мультипроцессорным CPU и двумя GPU. Один узел может иметь широкий спектр конфигураций, состоящий из одного или не-

---

[1] В зависимости от поставщика термин «варп» (*warp* – переплетение) означает: 1) набор потоков, выполняющих одну и ту же команду (для разных элементов данных); на языке Nvidia это SIMT-варп; 2) потоки, которые проходят вдоль тканого полотна (woven fabric). – *Прим. перев.*

[2] В случае GPU имеется в виду, что все потоки варпа одновременно исполняют одну общую команду. – *Прим. перев.*

скольких мультипроцессорных CPU со встроенным GPU и от одного до шести дискретных GPU. В номенклатуре OpenCL каждый GPU является вычислительным устройством. Но вычислительными устройствами также могут быть CPU в OpenCL.

ОПРЕДЕЛЕНИЕ *Вычислительное устройство* в OpenCL – это любое вычислительное оборудование, которое может выполнять вычисления и поддерживает OpenCL. Сюда могут входить GPU-процессоры, CPU или даже более экзотическое оборудование, такое как встроенные процессоры или программируемые пользователем вентильные матрицы (field-programmable gate array, FPGA).

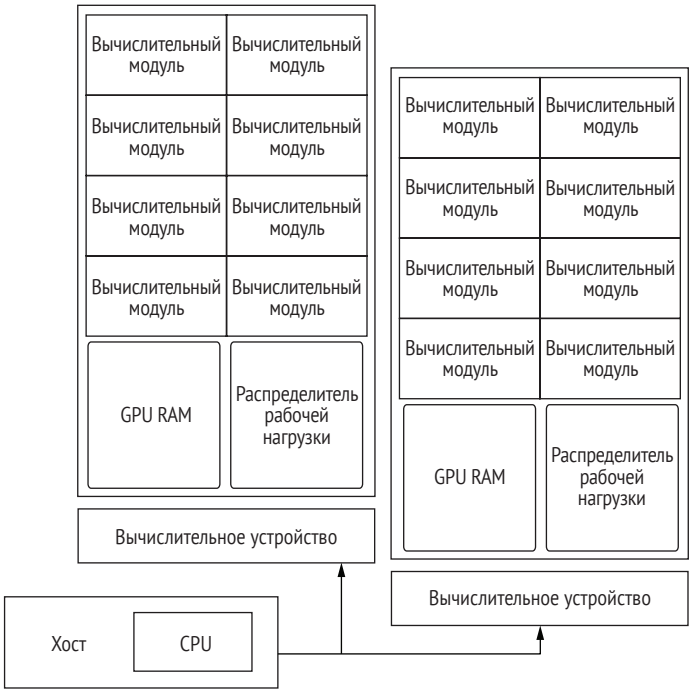


Рис. 9.2 Упрощенная структурная схема GPU-системы, показывающая два вычислительных устройства, каждое из которых имеет отдельный GPU, память GPU и несколько вычислительных модулей (CU). Терминология NVIDIA CUDA обозначает CU как потоковые мультипроцессоры (streaming multi-processor, SM)

Упрощенная схема на рис. 9.2 является нашей моделью для описания компонентов GPU, а также полезна при понимании того, как GPU обрабатывает данные. GPU состоит из:

- оперативной памяти GPU (RAM), также именуемой *глобальной памятью*;
- распределителя рабочей нагрузки;
- вычислительных модулей (compute unit, CU), или потоковых мультипроцессоров в CUDA.

CU имеют свою собственную внутреннюю архитектуру, часто именуемую *микроархитектурой*. Полученные от CPU команды и данные обрабатываются распределителем рабочей нагрузки. Указанный распределитель координирует исполнение команд и перемещение данных в вычислительные модули и из них. Достижимая производительность GPU зависит от:

- пропускной способности глобальной памяти;
- пропускной способности вычислительного модуля (CU);
- числа вычислительных модулей.

В этом разделе мы разведаем все до единой компоненты нашей модели на базе GPU. С каждым компонентом мы также обсудим модели теоретической пиковой пропускной способности. Кроме того, мы покажем, как использовать инструменты сравнительного микротестирования для измерения фактической производительности компонентов.

9.2.1 Вычислительным модулем является потоковый мультипроцессор (или подсрез)

Вычислительное устройство GPU-процессора имеет несколько вычислительных модулей (CU). (Термин «вычислительный модуль» был согласован сообществом для стандарта OpenCL.) В NVIDIA их называют *потоковыми мультипроцессорами* (SM), а в Intel – *подсрезами* (subslice).

9.2.2 Обрабатывающими элементами являются отдельные процессоры

Каждый CU содержит несколько GPU, именуемых в OpenCL обрабатывающими элементами (processing element, PE), или ядрами CUDA (или вычислительными ядрами), как их называют в NVIDIA. В Intel их именуют исполнительными модулями (execution unit, EU), тогда как графическое сообщество называет их шейдерными процессорами.

На рис. 9.3 показана упрощенная концептуальная схема обрабатывающих элементов. Эти процессоры не эквивалентны CPU; по своему дизай-

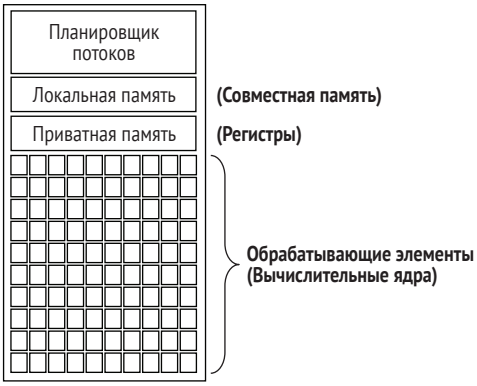


Рис. 9.3 Упрощенная структурная схема вычислительного модуля (CU) с большим числом обрабатывающих элементов (PE)

ну они представляют собой более простые конструкции, требующие выполнения операций с графикой. Но необходимые для графики операции включают в свой состав почти все арифметические операции, которые программист использует на обычном CPU.

9.2.3 Несколько операций, выполняемых на данных каждым обрабатывающим элементом

В рамках каждого PE можно выполнять операцию на нескольких элементах данных. В зависимости от деталей микропроцессорной архитектуры GPU и поставщика GPU они называются операциями SIMT, SIMD либо векторными операциями. Аналогичный тип функциональности может обеспечиваться за счет комплектования обрабатывающих элементов вместе.

9.2.4 Расчет пиковых теоретических флопов для некоторых ведущих GPU

Имея представление об оборудовании GPU, мы теперь можем рассчитать пиковые теоретические флопы для некоторых новейших GPU. К ним относятся NVIDIA V100, AMD Vega 20, AMD Arcturus и интегрированный GPU Gen11 на процессоре Intel Ice Lake. В табл. 9.3 перечислены спецификации этих GPU. Мы будем использовать эти спецификации для расчета теоретической производительности каждого устройства. Затем, зная теоретическую производительность, вы сможете сравнить работу каждого из них. Это поможет вам в принятии решений или в оценивании того, насколько быстрее или медленнее работает другой GPU в ваших расчетах. Технические характеристики оборудования для многих видеокарт можно найти на веб-сайте TechPowerUp: https://www.techpowerup.com/gpu-specs/.

Для NVIDIA и AMD GPU-процессоры, ориентированные на рынок HPC, имеют аппаратные ядра, позволяющие выполнять одну операцию двойной точности для каждых двух операций с одинарной точностью. Эту относительную способность флопов можно выразить как соотношение 1:2, где двойная точность равна 1:2 одинарной точности на высококлассных GPU. Важность этого соотношения заключается в том, что оно говорит о возможности примерно удваивать производительность, сводя ваши потребности в прецизионности двойной точности к одинарной. Для многих GPU половинная точность имеет соотношение 2:1 к одинарной точности или удвоенной способности флопов. Интегрированный GPU Intel имеет двойную точность 1:4 по сравнению с одинарной точностью, а некоторые стандартные GPU имеют соотношения 1:8 двойной точности к одинарной. GPU-процессоры с такими низкими коэффициентами двойной точности предназначены для рынка графики либо для машинного обучения. В целях получения этих соотношений надо взять строку FP64 и разделить ее на строку FP32.

Таблица 9.3 Технические характеристики новейших дискретных GPU от NVIDIA, AMD и интегрированных GPU Intel

GPU	NVIDIA V100 (Volta)	NVIDIA A100 (Ampere)	AMD Vega 20 (MI50)	AMD Arcturus (MI100)	Intel Gen11 Встроенный
Вычислительные модули (CU)	80	108	60	120	8
Ядра FP32/CU	64	64	64	64	64
Ядра FP64/CU	32	32	32	32	
GPU с номинальной/разогнанной тактовой частотой, МГц	1290/1530	1410	1200/1746	1000/1502	400/1000
Размер подгруппы или варпа	32	32	64	64	
Тактовая частота памяти	876 МГц	1215 МГц	1000 МГц	1200 МГц	Совместная память
Тип памяти	HBM2 (32 Гб)	HBM2(40 Гб)	HBM2	HBM2 (32 Гб)	LPDDR4X-3733
Ширина данных памяти	4096 бит	5120 бит	4096 бит	4096 бит	384 бит
Тип шины памяти	NVLink либо PCIe 3.0x16	NVLink либо PCIe Gen 4	Infinity Fabric либо PCIe 4.0x16	Infinity Fabric либо PCIe 4.0x16	Совместная память
Расчетная мощность	300 Вт	400 Вт	300 Вт	300 Вт	28 Вт

Пиковые теоретические флопы могут быть рассчитаны путем умножения тактовой частоты на число процессоров, умноженное на число операций с плавающей точкой за цикл. Число флопов за цикл обеспечивает возможность слитного умножения-сложения (FMA), которое выполняет две операции за один цикл.

Пиковые теоретические флопы (Гфлопс/с) = Тактовая частота МГц × Вычислительные модули × Обрабатывающие модули × флопы/цикл.

Пример: пиковый теоретический флоп для некоторых ведущих GPU

Теоретические пиковые флопы для NVIDIA V100:

- 2 × 1530 × 80 × 64 /10^6 = 15.6 Тфлопов (одинарная точность);
- 2 × 1530 × 80 × 32 /10^6 = 7.8 Тфлопов (двойная точность).

Теоретические пиковые флопы для NVIDIA Ampere:

- 2 × 1410 × 108 × 64 /10^6 = 19.5 Тфлопов (одинарная точность);
- 2 × 1410 × 108 x× 32 /10^6 = 9.7 Тфлопов (двойная точность).

Теоретические пиковые флопы для AMD Vega 20 (MI50):

- 2 × 1746 × 60 × 64 /10^6 = 13.4 Тфлопов (одинарная точность);
- 2 × 1746 × 60 × 32 /10^6 = 6.7 Тфлопов (двойная точность).

Теоретические пиковые флопы для AMD Arcturus (MI100):

- 2 × 1502 × 120 × 64 /10^6 = 23.1 Тфлопов (одинарная точность);
- 2 × 1502 × 120 × 32 /10^6 = 11.5 Тфлопов (двойная точность).

Теоретические пиковые флопы для Intel Integrated Gen 11 на Ice Lake:

- 2 × 1000 × 64 × 8 /10^6 = 1.0 Тфлопов (одинарная точность).

Как NVIDIA V100, так и AMD Vega 20 обеспечивают впечатляющую производительность с плавающей точкой. Ampere демонстрирует некоторое дополнительное улучшение производительности с плавающей точкой, но именно производительность памяти обещает более высокий прирост. MI100 от AMD демонстрирует более высокий скачок производительности с плавающей точкой. Интегрированный GPU Intel также впечатляет, учитывая, что он лимитирован доступным кремниевым пространством и более низкой номинальной расчетной мощностью процессора. Учитывая планы Intel по разработке дискретных видеокарт для нескольких сегментов рынка, в будущем у этого GPU ожидается еще больше опций.

9.3 Характеристики пространств памяти GPU

Типичный GPU имеет разные типы памяти. Использование правильного пространства памяти может оказать большое влияние на производительность. На рис. 9.4 эти типы памяти показаны в виде концептуальной диаграммы. Она помогает увидеть физические местоположения каждого уровня памяти, чтобы понять ее поведение. Хотя поставщик может разместить память GPU где угодно, она должна вести себя так, как показано на этой диаграмме.

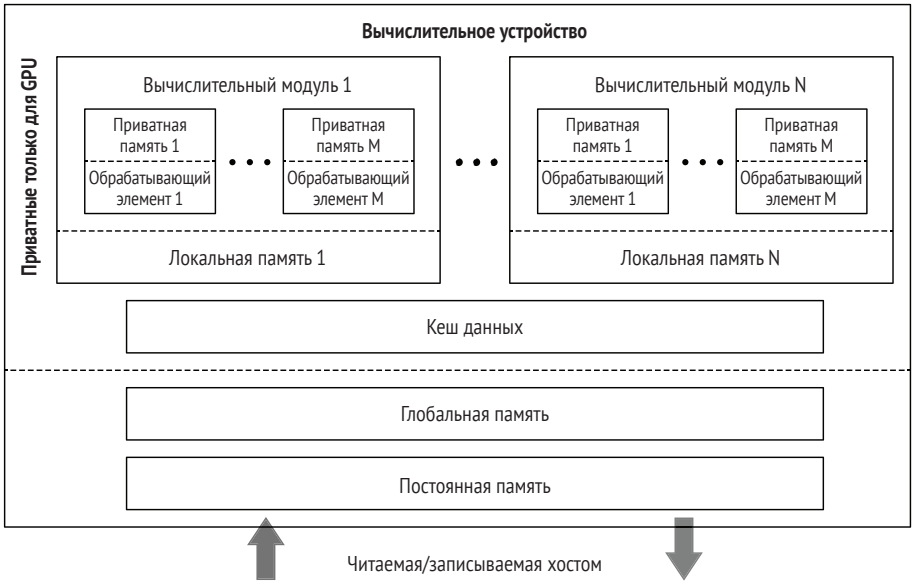


Рис. 9.4 Прямоугольники показывают каждый компонент GPU и память, которая находится на каждом аппаратном уровне. Хост пишет и читает глобальную и постоянную память. Каждый вычислительный модуль (CU) может читать и писать данные из глобальной памяти и читать данные из постоянной памяти

Список типов памяти GPU и их свойств выглядит следующим образом:

- *приватная память* (регистровая память) – немедленно доступна одному обрабатывающему элементу (PE) и только ему;
- *локальная память* – доступна для одного CU и всех PE на этом CU. Локальная память может быть разделена между блокнотом (сверхоперативной памятью), который может использоваться в качестве программируемого кеша, и некоторыми поставщиками в качестве традиционного кеша на GPU-процессорах. Размер локальной памяти составляет около 64–96 Кб;
- *постоянная память* – память только для чтения, доступная и совместная для всех CU;
- *глобальная память* – память, расположенная на GPU и доступная всем CU.

Одним из факторов, делающих GPU-процессоры быстрыми, является то, что они используют специализированную глобальную память (RAM), которая обеспечивает более высокую пропускную способность, тогда как текущие GPU используют память DDR4 и только сейчас переходят на DDR5. GPU-процессоры используют специальную версию GDDR5, которая обеспечивает более высокую производительность. Новейшие GPU теперь переходят на память с высокой пропускной способностью (High-Bandwidth Memory, HBM2), которая обеспечивает еще более высокую пропускную способность. Помимо увеличения пропускной способности, HBM также снижает энергопотребление.

9.3.1 Расчет теоретической пиковой пропускной способности памяти

Теоретическая пиковая пропускная способность памяти для GPU может быть рассчитана по тактовой частоте памяти на GPU и ширине транзакций памяти в битах. В табл. 9.4 показано несколько более высоких значений по каждому типу памяти. Нам также нужно умножить на два, чтобы учесть двойную скорость данных, с которой память извлекается как в верхней части цикла, так и в нижней. Некоторая память DDR может даже выполнять больше транзакций за цикл. В табл. 9.4 также показано несколько множителей транзакций для разных видов графической памяти.

Таблица 9.4 Технические характеристики распространенных типов памяти GPU

Тип графической памяти	Тактовая частота (МГц)	Транзакции памяти (ГТ/с)	Ширина шины памяти (бит)	Транзакционный множитель	Теоретическая пропускная способность (Гб/с)
GDDR3	1000	2.0	256	2	64
GDDR4	1126	2.2	256	2	70
GDDR5	2000	8.0	256	4	256
GDDR5X	1375	11.0	384	8	528
GDDR6	2000	16.0	384	8	768
HBM1	500	1000.0	4096	2	512
HBM2	1000	2000.0	4096	2	1000

При расчете теоретической пропускной способности памяти тактовая частота памяти умножается на число транзакций за цикл, а затем умножается на число битов, извлекаемых при каждой транзакции:

Теоретическая пропускная способность = Тактовая частота памяти (ГГц) × Шина памяти (биты) × (1 байт/8 бит) × транзакционный множитель.

Некоторые спецификации указывают скорость транзакций памяти в Гб/с, а не тактовую частоту памяти. Эта скорость представляет собой число транзакций за цикл, умноженное на тактовую частоту. Учитывая эту спецификацию, уравнение пропускной способности принимает следующий ниже вид:

Теоретическая пропускная способность = Скорость транзакций памяти (Гб/с) × Шина памяти (биты) × (1 байт/8 бит).

Пример: расчеты теоретической пропускной способности

- Для NVIDIA V100, работающего на частоте 876 МГц и использующего память HBM2:

  Теоретическая пропускная способность = 0.876 × 4096 × 1/8 × 2 = 897 Гб/с.

- Для GPU AMD Radeon Vega20 (MI50), работающего на частоте 1000 МГц:

  Теоретическая пропускная способность = 1.000 × 4096 × 1/8 × 2 = 1024 Гб/с.

9.3.2 Измерение GPU с помощью приложения STREAM Benchmark

Поскольку большинство наших приложений масштабируется с учетом пропускной способности памяти, приложение сравнительного тестирования STREAM Benchmark, которое измеряет пропускную способность памяти, является одним из наиболее важных сравнительных микротестов. Мы начали использовать указанное приложение для измерения пропускной способности CPU-процессоров в разделе 3.2.4. Процесс сравнительного тестирования аналогичен для GPU-процессоров, но нам нужно переписать потоковые вычислительные ядра на языках GPU. К счастью, это было сделано Томом Дикином (Tom Deakin) из Университета Бристоля для различных языков и оборудования GPU в его коде приложения сравнительного тестирования Babel STREAM Benchmark.

Исходный код приложения сравнительного тестирования Babel STREAM Benchmark измеряет пропускную способность различного оборудования с разными языками программирования. Мы применяем его здесь для измерения пропускной способности GPU NVIDIA с использованием CUDA. Также доступны версии на OpenCL, HIP, OpenACC, Kokkos, Raja, SYCL и OpenMP с GPU-целями. Все это разные языки, которые можно использовать для оборудования GPU, такого как GPU NVIDIA, AMD и Intel.

Упражнение: измерение пропускной способности с помощью приложения Babel STREAM Benchmark

Шаги по использованию приложения сравнительного тестирования Babel STREAM Benchmark для CUDA на GPU NVIDIA таковы.

1. Клонировать приложение Babel STREAM Benchmark с помощью

```
git clone git@github.com:UoB-HPC/BabelStream.git
```

2. Затем набрать:

```
make -f CUDA.make
./cuda-stream
```

Результаты для GPU NVIDIA V100 таковы:

```
Function  MBytes/sec  Min (sec)  Max       Average
Copy      800995.012  0.00067    0.00067   0.00067
Mul       796501.837  0.00067    0.00068   0.00068
Add       838993.641  0.00096    0.00097   0.00096
Triad     840731.427  0.00096    0.00097   0.00096
Dot       866071.690  0.00062    0.00063   0.00063
```

Данный процесс аналогичен для GPU AMD.

1. Отредактировать файл OpenCL.make и добавить пути в ваши заголовочные файлы и библиотеки OpenCL.
2. Затем набрать:

```
make -f OpenCL.make
./ocl-stream
```

Для AMD Vega 20 пропускная способность GPU немного ниже, чем у GPU NVIDIA.

```
Using OpenCL device gfx906+sram-ecc
Function    MBytes/sec  Min (sec)  Max       Average
Copy        764889.965  0.00070    0.00077   0.00072
Mul         764182.281  0.00070    0.00076   0.00072
Add         764059.386  0.00105    0.00134   0.00109
Triad       763349.620  0.00105    0.00110   0.00108
Dot         670205.644  0.00080    0.00088   0.00083
```

9.3.3 Модель производительности в форме контура крыши для GPU-процессоров

Мы представили модель производительности в форме контура крыши для CPU в разделе 3.2.4. Эта модель учитывает как пропускную способность памяти, так и пределы производительности системы. Указанная модель аналогичным образом полезна и для GPU-процессоров и помогает понять их пределы производительности.

Упражнение: измерение пропускной способности с использованием инструментария Empirical Roofline Toolkit

Для этого упражнения вам понадобится доступ к GPU NVIDIA и/или AMD. Аналогичный процесс может быть использован для других GPU.

1. Получить инструментарий Empirical Roofline Toolkit с помощью

```
git clone https://bitbucket.org/berkeleylab/cs-roofline-toolkit.git
```

2. Затем набрать:

```
cd cs-roofline-toolkit/Empirical_Roofline_Tool-1.1.0
cp Config/config.voltar.uoregon.edu Config/config.V100_gpu
```

3. Отредактировать следующие ниже настройки в файле Config/config.V100_gpu:

```
ERT_RESULTS Results.V100_gpu
ERT_PRECISION FP64
ERT_NUM_EXPERIMENTS 5
```

4. Выполнить:

```
tests ./ert Config/config.V100_gpu
```

5. Просмотреть файл Results.config.V100_gpu/Run.001/roofline.ps

```
cp Config/config.odinson-ocl-fp64.01 Config/config.Vega20_gpu
```

6. Отредактировать следующие ниже настройки в файле Config/config.Vega20_gpu:

```
ERT_RESULTS Results.Vega20_gpu
ERT_CFLAGS -O3 -x c++ -std=c++11 -Wno-deprecated-declarations
           -I<path to OpenCL headers>
ERT_LDLIBS -L<path to OpenCL libraries> -lOpenCL
```

7. Выполнить:

```
tests ./ert Config/config.Vega20_gpu
```

8. Просмотреть результаты в файле Results.config.Vega20_gpu/Run.001/roofline.ps.

На рис. 9.5 показаны результаты сравнительных тестов на основе модели контура крыши для GPU NVIDIA V100 и для GPU AMD Vega20.

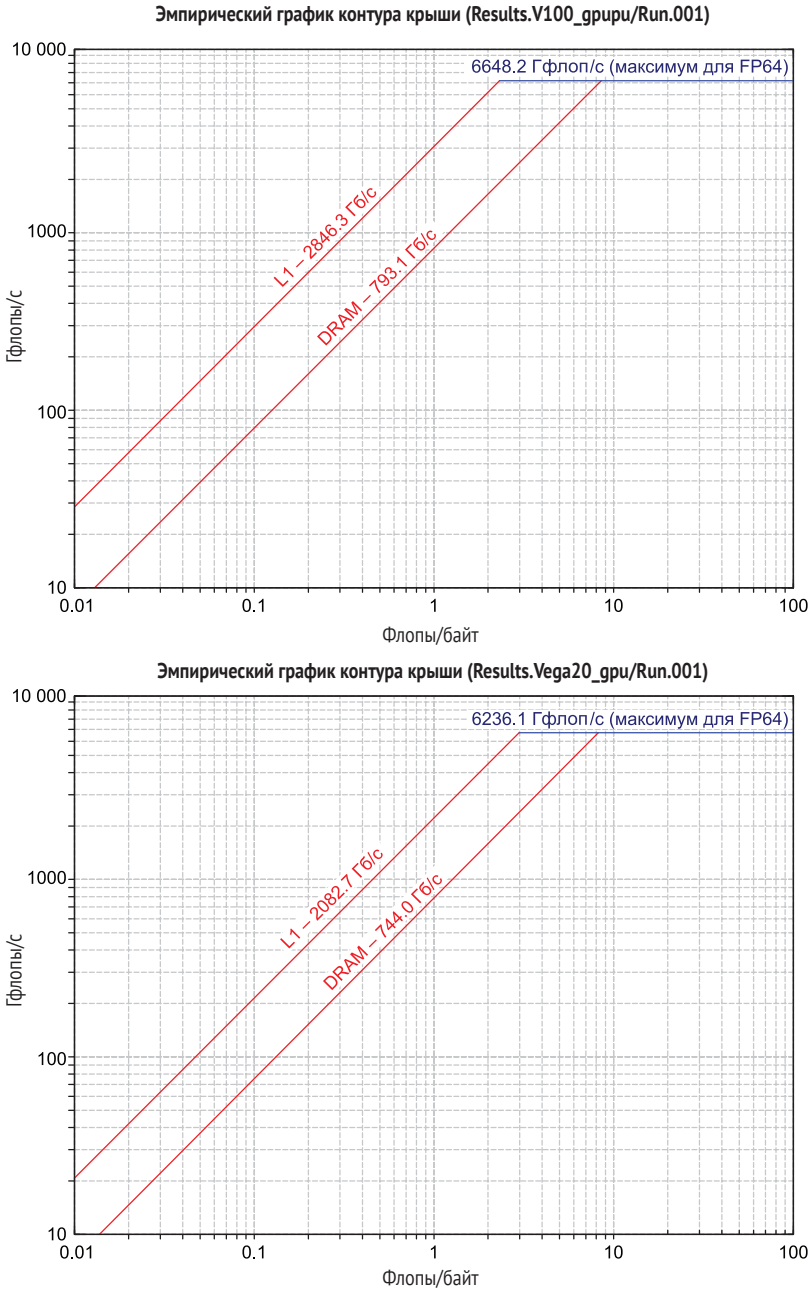


Рис. 9.5 Графики контура крыши для NVIDIA V100 и AMD Vega 20, показывающие пределы пропускной способности и флопов для двух GPU

9.3.4 Использование инструмента смешанного сравнительного тестирования производительности для выбора наилучшего GPU для рабочей нагрузки

Для облачных служб и на рынке серверов HPC предлагается много вариантов GPU. Есть ли способ определить оптимальный GPU для вашего приложения? Мы рассмотрим модель производительности, которая поможет вам выбирать GPU, подходящий для вашей рабочей нагрузки лучше всего.

Поменяв независимую переменную в графике контура крыши с арифметической интенсивности на пропускную способность памяти, мы можем высветить пределы производительности приложения относительно каждого устройства GPU. Инструмент смешанного сравнительного тестирования (mixbench) был разработан для выявления различий в производительности разных устройств GPU. Эта информация на самом деле ничем не отличается от той, что показана в модели контура крыши, но визуально она оказывает совсем другое влияние. Давайте пройдемся по упражнению с использованием инструмента смешанного сравнительного тестирования, чтобы показать, что вы можете узнать.

Упражнение: получение пиковой скорости флопов и пропускной способности с помощью инструмента смешанного сравнительного тестирования

1. Получить код инструмента смешанного сравнительного тестирования:

   ```
   git clone https://github.com/ekondis/mixbench.git
   ```

2. Проверить наличие CUDA либо OpenCL и при необходимости инсталлировать:

   ```
   cd mixbench; edit Makefile
   ```

3. Скорректировать путь к инсталляциям CUDA и/или OpenCL.
4. Задать исполняемые файлы для сборки. Путь к инсталляции CUDA можно переопределить с помощью команды

   ```
   make CUDA_INSTALL_PATH=<path>
   ```

5. Выполнить одно из следующих ниже действий:

   ```
   ./mixbench-cuda-ro
   ./mixbench-ocl-ro
   ```

Результаты сравнительного теста представлены в виде вычислительной скорости в Гфлопах/с по отношению к пропускной способности памяти в Гб/с (рис. 9.6). В принципе, результаты сравнительного теста показывают горизонтальную линию при пиковой скорости флопов и вертикальное падение при пределе пропускной способности памяти. Из каждого из этих значений берется максимум и используется для на-

несения единственной точки в правом верхнем углу, которая отражает как пиковую способность флопов GPU-устройства, так и пик его пропускной способности.

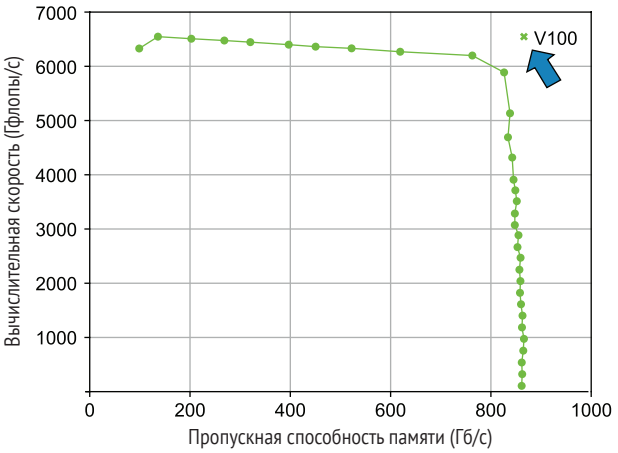


Рис. 9.6 Результат на выходе из смешанного тестирования на V100 (показан в виде линейного графика). Максимальная пропускная способность и скорость с плавающей точкой обнаруживаются и используются для нанесения точки производительности V100 в правом верхнем углу графика

Мы можем выполнять инструмент смешанного сравнительного тестирования для самых разных устройств GPU и получать их пиковые характеристики производительности. GPU-процессоры, разработанные для рынка HPC, обладают высокой точностью с плавающей точкой, а GPU-процессоры для других рынков, таких как графика и машинное обучение, ориентированы на оборудование с одинарной точностью.

Мы можем изобразить каждое устройство GPU на рис. 9.7 линией, представляющей арифметическую или операционную интенсивность приложения. Наиболее типичные области применения имеют интенсивность около 1 флоп/загрузка. С другой стороны, матричное умножение имеет арифметическую интенсивность 65 флопов/загрузка. Мы показываем наклонную линию для обоих этих типов приложений на рис. 9.7. Если точка GPU находится выше линии приложения, то мы проводим вертикальную линию вниз к линии приложения, чтобы определить уровень достижимой производительности приложения. Для устройства справа и ниже линии приложения мы используем горизонтальную линию, чтобы найти предел производительности.

Указанный график проясняет соответствие характеристик устройства GPU требованиям приложения. Для типичного приложения с интенсивностью 1 флоп/загрузка хорошо подходят такие GPU, как GeForce GTX 1080Ti, созданные для рынка графики. GPU V100 больше подходит для сравнительного теста Linpack, используемого для ранжирования TOP500 крупных вычислительных систем, поскольку он в основном состоит из матричного умножения. Такие GPU, как V100, являются специализиро-

ванным оборудованием, специально созданным для рынка HPC, и имеют высокую премию за цену. Для некоторых приложений с меньшей арифметической интенсивностью бывают более выгодными товарные GPU-процессоры, сконструированные для рынка графики.

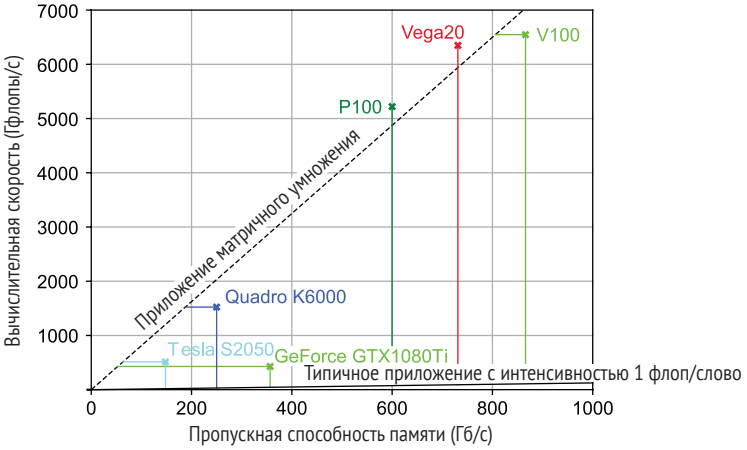


Рис. 9.7 Коллекция точек производительности для устройств GPU (показана на графике справа) вместе с арифметической интенсивностью приложения (показана прямыми линиями). Значения выше строки обозначают, что приложение ограничено памятью, а ниже строки обозначают, что оно ограничено вычислениями

9.4 Шина PCI: накладные расходы на передачу данных от CPU к GPU

Показанная на рис. 9.1 шина PCI необходима для передачи данных с CPU на GPU и обратно. Стоимость передачи данных может существенно ограничивать производительность переносимых на GPU операций. Нередко крайне важно ограничивать объем данных, которые передаются туда и обратно, чтобы ускорять работу GPU.

Текущая версия шины PCI называется PCI Express (PCIe). На момент написания этой книги она несколько раз пересматривалась в поколениях с 1.0 до 6.0. В целях понимания предела производительности шины PCIe вы должны знать номер имеющегося в вашей системе поколения. В этом разделе мы покажем два метода оценивания пропускной способности шины PCI:

- упрощенную модель теоретической пиковой производительности;
- приложение для сравнительного микротеста.

Модель теоретической пиковой производительности полезна для быстрого оценивания того, что можно ожидать от новой системы. Выгода от указанной модели состоит в том, что вам не нужно выполнять каких-либо приложений в системе. Это полезно, когда вы только приступаете

к проекту и хотели бы быстро оценить вручную возможные узкие места в производительности. Вдобавок пример со сравнительным тестом показывает, что достижимая вами пиковая пропускная способность зависит от того, как вы используете оборудование.

9.4.1 Теоретическая пропускная способность шины PCI

На платформах с выделенными GPU весь обмен данными между GPU и CPU происходит по шине PCI. По этой причине он является важным аппаратным компонентом, который может сильно влиять на совокупную производительность вашего приложения. В этом разделе мы рассмотрим ключевые функции и дескрипторы шины PCI, о которых вам необходимо знать, чтобы рассчитывать теоретическую пропускную способность шины PCI. Уметь рассчитывать это число на лету полезно для оценивания возможных ограничений производительности при переносе вашего приложения на GPU-процессоры.

Шина PCI – это физический компонент, который закрепляет выделенные GPU-процессоры за CPU и другими устройствами. Он позволяет осуществлять обмен между CPU и GPU. Обмен осуществляется по нескольким полосам PCIe. Мы начнем с введения формулы теоретической пропускной способности, а затем объясним каждый ее член.

Теоретическая пропускная способность (Гб/с) = Полосы × Скорость передачи (ГТ/с) × Коэффициент накладных расходов(Гб/ГТ) × байт/8 бит.

Теоретическая пропускная способность измеряется в единицах гигабайт в секунду (Гб/с). Она рассчитывается путем умножения числа полос и максимальной скорости передачи данных для каждой полосы, а затем конвертирования из битов в байты. Конверсия остается в формуле, поскольку скорость передачи обычно указывается в гигатрансферах в секунду (ГТ/с). Коэффициент накладных расходов обусловлен применяемой для обеспечения целостности данных схемой кодирования, которая снижает эффективную скорость передачи. Для устройств поколения 1.0 стоимость схемы кодирования составляла 20 %, поэтому коэффициент накладных расходов составлял 100 – 20%, или 80%. Начиная с поколения 3.0 и далее, накладные расходы на схему кодирования падают до каких-то 1.54 %, поэтому достигнутая пропускная способность становится по существу такой же, как и скорость передачи. Давайте теперь остановимся на каждом члене уравнения пропускной способности.

Полосы PCIe

Число полос шины PCI можно узнать, просмотрев спецификации производителя, либо можно применить ряд инструментов, доступных на платформах Linux. Имейте в виду, что для некоторых из этих инструментов могут потребоваться права корневого пользователя. Если у вас нет этих

прав, то лучше всего будет проконсультироваться с системным администратором, чтобы узнать эту информацию. Тем не менее мы представим два варианта определения числа полос PCIe.

В системах Linux широко распространена утилита lspci. Указанная утилита перечисляет все компоненты, прикрепленные к материнской плате. Мы можем использовать поисковый инструмент grep на основе регулярных выражений для вычленения только моста PCI. Следующая ниже команда покажет вам информацию о поставщике и название устройства с указанием числа полос PCIe. В данном примере (`x16`) среди результатов обозначает, что имеется 16 полос.

```
$ lspci -vmm | grep "PCI bridge" -A2
Class: PCI bridge
Vendor: Intel Corporation
Device: Sky Lake PCIe Controller (x16)
```

Команда dmidecode в качестве альтернативы предоставляет аналогичную информацию:

```
$ dmidecode | grep "PCI"
PCI is supported
Type: x16 PCI Express
```

Определение максимальной скорости передачи

Максимальная скорость передачи данных для каждой полосы в шине PCIe может быть напрямую определена по ее поколению. Поколение – это спецификация требуемой производительности оборудования, во многом аналогичная 4G, которая является отраслевым стандартом для мобильных телефонов. Группа с общим интересом в PCI (PCI SIG) представляет отраслевых партнеров и формирует спецификацию PCIe, которую для краткости принято называть *поколением*, или *gen* (от *generation*). В табл. 9.5 показана максимальная скорость передачи данных по полосам и направлениям PCI.

Таблица 9.5 Спецификации PCI Express (PCIe) в разбивке по поколениям

Поколение PCIe	Максимальная скорости передачи (двунаправленная)	Накладные расходы кодирования	Коэффициент накладных расходов (100 % – накладные расходы кодирования	Теоретическая пропускная способность 16 полос – Гб/с
Gen1	2.5 GT/s	20 %	80 %	4
Gen2	5.0 GT/s	20 %	80 %	8
Gen3	8.0 GT/s	1.54 %	98.46 %	15.75
Gen4	16.0 GT/s	1.54%	98.46 %	31.5
Gen5 (2019)	32.0 GT/s	1.54 %	98.46 %	63
Gen6 (2021)	64.0 GT/s	1.54%	98.46 %	126

Если вы не знаете поколение вашей шины PCIe, то для получения этой информации можно применить команду lspci. Среди информации на выходе из lspci мы ищем емкость передаточного звена (link capacity) для шины PCI. Ниже этот показатель сокращенно называется `LnkCap`:

```
$ sudo lspci -vvv | grep -E 'PCI|LnkCap'
Output:

00:01.0 PCI bridge:
        Intel Corporation Sky Lake PCIe Controller (x16) (rev 07)
LnkCap: Port #2, Speed 8GT/s, Width x16, ASPM L0s L1, Exit Latency L0s
```

Теперь, когда мы узнали максимальную скорость передачи из этой информации, мы можем использовать ее в формуле пропускной способности. Кроме того, полезно также знать, что эта скорость может быть выровнена с поколением. В данном случае информация говорит о том, что мы работаем с системой PCIe Gen3.

ПРИМЕЧАНИЕ В некоторых системах результат на выходе из lspci и других системных утилит, возможно, не даст много информации. Результат зависит от системы и просто сообщает об идентификации из каждого устройства. Если определить характеристики из этих утилит невозможно, то резервным вариантом может быть использование приведенного в разделе 9.4.2 исходного кода сравнительного теста PCI для определения способностей вашей системы.

Уровни накладных расходов

Передача данных по шине PCI требует дополнительных накладных расходов. Стандарты поколения 1 и 2 предусматривают, что на каждые 8 байт полезных данных передается 10 байт. Начиная с поколения 3, передается 130 байт на каждые 128 байт данных. Коэффициент накладных расходов – это отношение числа используемых байтов к суммарному числу переданных байтов (табл. 9.5).

Опорные данные для теоретической пиковой пропускной способности PCIe

Теперь, когда у нас есть вся необходимая информация, давайте оценим теоретическую пропускную способность на примере, используя выходные данные, показанные в предыдущих разделах.

Пример: оценивание теоретической пропускной способности

Мы определили, что у нас есть система PCIe поколения Gen3 с 16 полосами. Системы поколения Gen3 имеют максимальную скорость передачи данных 8.0 ГТ/с и коэффициент накладных расходов 0.985. Как показано ниже, при наличии 16 полос теоретическая пропускная способность составляет 15.75 Гб/с.

Теоретическая пропускная способность (Гб/с) =
16 полос × 8.0 ГТ/с × 0.985 (Гб/ГТ) × байт/8 бит = 15.75 Гб/с.

9.4.2 Приложение сравнительного тестирования пропускной способности PCI

Уравнение теоретической пропускной способности PCI дает ожидаемую пиковую пропускную способность. Другими словами, это максимально возможная пропускная способность, которая может быть обеспечена приложением для данной платформы. На практике достигнутая пропускная способность может зависеть от ряда факторов, включая операционную систему, системные драйверы, другие аппаратные компоненты вычислительного узла, API-программирования GPU и размер блока данных, передаваемого по шине PCI. В большинстве систем, к которым у вас есть доступ, вполне вероятно, что модификация всех вариантов, кроме двух последних, лежит вне вашего контроля. Во время разработки приложения вы контролируете программный API и размер блоков данных, передаваемых по шине PCI.

В связи с этим мы остаемся перед вопросом, как размер блока данных влияет на достигнутую пропускную способность? На этот тип вопросов обычно отвечают с помощью *сравнительного микротеста*. Сравнительный микротест, или бенчмарк, – это небольшая программа, предназначенная для исполнения одного процесса или элемента оборудования, которое используется более крупным приложением. Сравнительные микротесты помогают получать некоторые показатели производительности системы.

В нашей ситуации мы хотим разработать сравнительный микротест, который копирует данные с CPU на GPU и наоборот. Поскольку ожидается, что копирование данных будет происходить от микросекунд до десятков микросекунд, мы измерим время, необходимое для совершения 1000-кратного копирования данных. Затем это время будет разделено на 1000, чтобы получить среднее время копирования данных между CPU и GPU.

Мы проведем пошаговый обзор с использованием приложения сравнительного тестирования для измерения пропускной способности PCI. В листинге 9.1 показан исходный код, который копирует данные с хоста на GPU. Написание на языке программирования CUDA GPU будет рассмотрено в главе 10, но базовая операция понятна из имен функций. В листинге 9.1 мы на входе принимаем размер плоского одномерного массива, который мы хотим скопировать с CPU (хоста) на GPU (устройство).

ПРИМЕЧАНИЕ Указанный исходный код доступен в подкаталоге PCI_Bandwidth_Benchmark по адресу https://github.com/Essential-sofParallelComputing/Chapter9.

Листинг 9.1 Копирование данных с CPU-хоста на GPU-устройство

PCI_Bandwidth_Benchmark.c

```
35 void Host_to_Device_Pinned( int N, double *copy_time )
36 {
```

```
37    float *x_host, *x_device;
38    struct timespec tstart;
39
40    cudaError_t status = cudaMallocHost((void**)&x_host, N*sizeof(float));
41   if (status != cudaSuccess)
42       printf("Ошибка выделения закрепленной памяти хоста\n");
43    cudaMalloc((void **)&x_device, N*sizeof(float));
44
45    cpu_timer_start(&tstart);
46    for(int i = 1; i <= 1000; i++ ){
47       cudaMemcpy(x_device, x_host, N*sizeof(float),
      cudaMemcpyHostToDevice);
48    }
49    cudaDeviceSynchronize();
50
51    *copy_time = cpu_timer_stop(tstart)/1000.0;
52
53    cudaFreeHost( x_host );
54    cudaFree( x_device );
55 }
```

Выделяет закрепленную память хоста для массива на CPU

Выделяет память для массива на GPU

Копирует память на GPU

Синхронизирует GPU, чтобы завершить работу

Высвобождает массивы

В листинге 9.1 первым шагом является выделение памяти для копии хоста и для копии устройства. Процедура в строке 40 в этом листинге использует `cudaMallocHost`, выделяя закрепленную память на хосте для более быстрой передачи данных. Для процедуры, которая использует регулярную листаемую память, используются стандартные вызовы `malloc` и `free`. Процедура `cudaMemcpy` передает данные с хоста CPU на GPU. Вызов `cudaDeviceSynchronize` ожидает завершения копирования. Перед циклом, в котором мы повторяем копирование хоста на устройство, мы фиксируем время начала. Затем выполняем копирование с хоста на устройство 1000 раз подряд и снова фиксируем текущее время. Затем среднее время копирования с хоста на устройство рассчитывается путем деления на 1000. Для аккуратности мы высвобождаем пространство, занимаемое массивами хоста и устройства.

Зная время, необходимое для передачи массива размером N с хоста на устройство, теперь мы можем вызывать эту процедуру многократно, всякий раз меняя N. Однако нас больше интересует оценивание достигнутой пропускной способности.

Напомним, что пропускная способность есть число передаваемых байтов за единицу времени. На данный момент мы знаем число элементов массива и время, необходимое для копирования массива между CPU и GPU. Число передаваемых байтов зависит от типа хранящихся в массиве данных. Например, в массиве размером N, содержащем вещественные числа (4 байта), объем данных, копируемых между CPU и GPU, составляет $4N$. Если $4N$ байтов передается за время T, то достигнутая пропускная способность составит

$$B = 4N/T.$$

Это позволяет нам построить набор данных, показывающий достигнутую пропускную способность в зависимости от N. Подпрограмма

в следующем ниже листинге требует, чтобы был указан максимальный размер массива, а затем возвращает пропускную способность, измеренную для каждого эксперимента.

Листинг 9.2 Вызов передачи памяти из CPU в GPU для разных размеров массива

PCI_Bandwidth_Benchmark.c

```
81 void H2D_Pinned_Experiments(double **bandwidth, int n_experiments,
        int max_array_size){
82    long long array_size;
83    double copy_time;
84
85    for(int j=0; j<n_experiments; j++){
86       array_size = 1;
87       for(int i=0; i<max_array_size; i++ ){
88
89          Host_to_Device_Pinned( array_size, &copy_time );
90
91          double byte_size=4.0*array_size;
92          bandwidth[j][i] = byte_size/(copy_time*1024.0*1024.0*1024.0);
93
94          array_size = array_size*2;
95       }
96    }
97 }
```

Повторяет эксперименты несколько раз

Вызывает проверку и хронометраж памяти CPU с GPU

Удваивает размер массива с каждой итерацией

Вычисляет пропускную способность

Здесь мы перебираем размеры массива и для каждого размера получаем среднее время копирования с хоста на устройство. Затем пропускная способность вычисляется по числу байтов, деленному на время, необходимое для копирования. Массив содержит вещественные числа, которые имеют по четыре байта для каждого элемента массива. Теперь давайте рассмотрим пример использования приложения сравнительного микротестирования для характеризации производительности вашей шины PCI.

Производительность поколения Gen3 x16 на ноутбуке

Мы выполнили приложение микротестирования пропускной способности PCI на ноутбуке с ускоренным GPU. В этой системе команда lspci показывает, что указанная система оснащена шиной PCI Gen3 x16:

```
$ sudo lspci -vvv | grep -E 'PCI|LnkCap'
00:01.0 PCI bridge: Intel Corporation Sky Lake PCIe Controller (x16)
                    (rev 07)
            LnkCap: Port #2, Speed 8GT/s, Width x16, ASPM L0s L1,
                    Exit Latency L0s
```

Для этой системы теоретическая пиковая пропускная способность шины PCI составляет 15.8 Гб/с. На рис. 9.8 показан график достигнутой пропускной способности в нашем приложении микротестирования (кривые линии) по сравнению с теоретической пиковой пропускной

способностью (горизонтальные пунктирные линии). Затененный участок вокруг достигнутой полосы пропускания указывает на стандартное отклонение ±1 в пропускной способности.

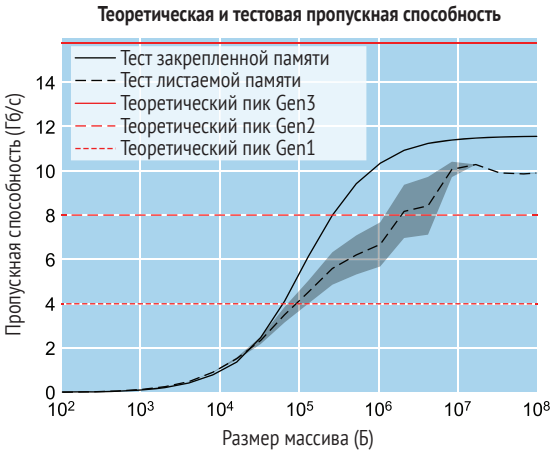


Рис. 9.8 Показаны теоретическая пиковая пропускная способность (горизонтальные линии) и эмпирически измеренная пропускная способность на выходе из приложения сравнительного микротестирования для системы PCIe Gen3 x16. На рисунке также показаны результаты как для закрепленной, так и для листаемой памяти

Прежде всего обратите внимание, что при передаче малых порций данных по шине PCI достигаемая пропускная способность невелика. За пределами размеров массива в 10^7 байт достигаемая пропускная способность приближается к максимуму около 11.6 Гб/с. Обратите также внимание, что пропускная способность с закрепленной памятью намного выше, чем у листаемой памяти, а листаемая память имеет гораздо более широкий диапазон результатов производительности для каждого размера памяти. Полезно знать, что такое закрепленная (pinnable) и листаемая (pagable) память, чтобы понимать причину этой разницы.

- *Закрепленная память* – память, которую невозможно листать из RAM и поэтому ее можно отправлять в GPU без предварительного копирования.
- *Листаемая память* – это стандартные выделения памяти, которые можно выгружать на диск постранично.

Выделение закрепленной памяти уменьшает объем памяти, доступной для других процессов, поскольку ядро операционной системы больше не может выводить страницу памяти на диск, чтобы другие процессы могли ее использовать. Закрепленная память выделяется из стандартной DRAM-памяти для процессора. Процесс выделения занимает немного больше времени, чем обычное выделение. При использовании листаемой памяти ее необходимо копировать в закрепленную ячейку памяти перед отправкой. Закрепленная память предотвращает ее постраничную выгрузку на диск во время передачи памяти. Пример в этом разделе

показывает, что более крупные передачи данных между CPU и GPU могут приводить к увеличению пропускной способности. Кроме того, в этой системе максимальная достигаемая пропускная способность достигает лишь около 72 % от теоретической пиковой производительности.

9.5 Платформы с многочисленными GPU и MPI

Теперь, когда мы ввели основные компоненты платформы с ускорением за счет GPU, обсудим более экзотические конфигурации, с которыми вы можете столкнуться. Указанные экзотические конфигурации появились в результате внедрения многочисленных GPU. Некоторые платформы предлагают несколько GPU на узел, подключенных к одному или нескольким CPU. Другие предлагают подключение к нескольким вычислительным узлам через сетевое оборудование.

На платформах с многочисленными GPU (рис. 9.9) обычно для обеспечения параллелизма необходимо использовать подход MPI+GPU. Для обеспечения параллелизма данных каждый ранг MPI назначается одному из GPU. Давайте рассмотрим несколько возможностей:

- один ранг MPI управляет каждым GPU;
- несколько рангов MPI мультиплексируют свою работу на GPU.

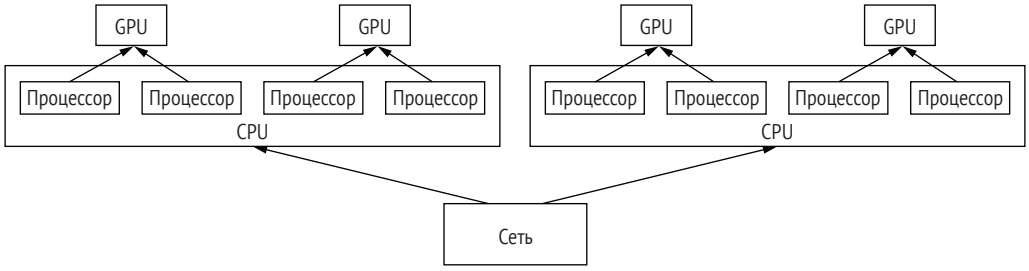


Рис. 9.9 Здесь мы иллюстрируем платформу с многочисленными GPU. Один вычислительный узел может иметь несколько GPU и несколько CPU. Кроме того, несколько узлов может быть соединено по сети

Некоторые ранние программные и аппаратные средства GPU не справлялись с эффективным мультиплексированием, что приводило к слабой производительности. С учетом исправления многих проблем с производительностью в новейшем программном обеспечении все более привлекательным становится мультиплексирование рангов MPI на GPU-процессоры.

9.5.1 Оптимизирование перемещения данных между GPU-процессорами по сети

В целях использования нескольких GPU мы должны отправлять данные с одного GPU на другой. Прежде чем мы сможем обсудить оптимизацию, нам необходимо описать стандартный процесс передачи данных.

1. Скопировать данные с GPU на хост-процессор.
   a. Переместить данные по шине PCI на процессор.
   b. Сохранить данные в DRAM-памяти CPU.
2. Отправить данные в сообщении MPI другому процессору.
   a. Сохранить данные из памяти CPU в процессоре.
   b. Переместить данные по шине PCI на сетевую интерфейсную карту (NIC).
   c. Сохранить данные из процессора в памяти CPU.
3. Скопировать данные со второго процессора на второй GPU.
   a. Загрузить данные из памяти CPU в процессор.
   b. Отправить данные по шине PCI на GPU.

Как показано на рис. 9.10, этот процесс сопряжен с изрядным перемещением данных и будет серьезным ограничением производительности приложения.

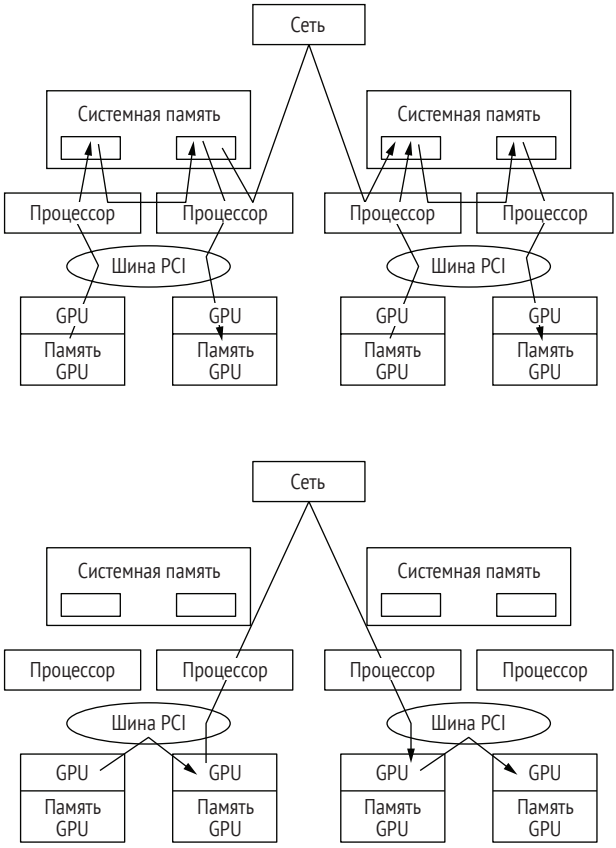


Рис. 9.10 Вверху показано стандартное перемещение данных для отправки данных с GPU на другие GPU. В нижней части перемещение данных обходит CPU стороной при перемещении данных с одного GPU на другой

В NVIDIA GPUDirect® язык CUDA добавляет способность отправлять данные в виде сообщения. У AMD есть аналогичная способность, имену-

емая DirectGMA для обмена данными между GPU-процессорами в языке OpenCL. По-прежнему необходимо передавать указатель на данные, но само сообщение отправляется по шине PCI непосредственно с одного GPU на другой, тем самым уменьшая объем памяти.

9.5.2 Более высокопроизводительная альтернатива шине PCI

Существует мало аргументов в пользу того, что шина PCI является главенствующим ограничением для вычислительных узлов с многочисленными GPU. Хотя она по большей части является проблемой для крупных приложений, она также влияет на большие рабочие нагрузки, такие как машинное обучение в малых кластерах. NVIDIA в своей линейке GPU-процессоров Volta P100 и V100 ввела NVLink® для замены соединений GPU-GPU и GPU-CPU. Начиная с NVLink 2.0, скорость передачи данных может достигать 300 Гб/с. Новые GPU и CPU компании AMD включают в свой состав Infinity Fabric для ускорения передачи данных. И в Intel уже несколько лет используется технология ускорения передачи данных между CPU-процессорами и памятью.

9.6 Потенциальные преимущества платформ, ускоренных за счет GPU

Когда стоит переносить обработку на GPU? К этому месту вы увидели теоретическую пиковую производительность современных GPU и то, как она соотносится с CPU. Сравните график контура крыши для GPU на рис. 9.5 с графиком контура крыши CPU (из раздела 3.2.4) расчета с плавающей точкой и пределов пропускной способности памяти. На практике многие приложения не достигают этих пиковых значений производительности. Однако с повышением потолков GPU есть потенциал превзойти архитектуры CPU по нескольким показателям. К ним относятся время выполнения приложения, энергопотребление, затраты на облачные вычисления и масштабируемость.

9.6.1 Сокращение показателя времени до решения

Предположим, у вас есть код, который выполняется на CPU-процессорах. Вы потратили много времени на добавление в него OpenMP или MPI, чтобы использовать все ядра CPU. Вы чувствуете, что код хорошо настроен, но ваш товарищ сказал вам, что вы могли бы извлечь больше пользы, перенеся свой код на GPU-процессоры. У вас в приложении более 10 000 строк кода, и вы знаете, что для работы кода на GPU-процессорах потребуются значительные усилия. На данном этапе вас интересует перспектива работы на GPU-процессорах, потому что вам нравится узна-

вать что-то новое и вы доверяете интуиции своего товарища. Теперь вы должны изложить это дело своим коллегам и своему боссу.

Важной мерой производительности вашего приложения является сокращение показателя времени до решения для заданий, которые выполняются несколько дней подряд. Самый лучший способ донести до интересантов влияние сокращения времени до решения – это обратиться к примеру. В этом исследовании мы будем использовать приложение Cloverleaf в качестве подстановки.

Пример: рассмотрение возможности обновления вашей текущей интенсивно используемой системы

Ниже приведены шаги оценивания производительности Cloverleaf на базовой системе. Мы используем систему Intel Ivybridge (E5-2650 v2 @ 2.60 ГГц) с 16 физическими ядрами. В среднем ваше приложение работает около 500 000 циклов. Вы можете получить оценку времени выполнения для короткого пробного прогона с помощью следующих ниже шагов.

1. Клонировать Cloverleaf.git с

```
git clone --recursive git@github.com:UK-MAC/CloverLeaf.git CloverLeaf
```

2. Затем ввести следующие ниже команды:

```
cd CloverLeaf_MPI
make COMPILER=INTEL
sed -e '1,$s/end_step=2955/end_step=500/' InputDecks/clover_bm64.in\
         >clover.in
mpirun -n 16 --bind-to core ./clover_leaf
```

Время выполнения 500 циклов составляет 615.1 с или 1.23 с за цикл. Это дает вам время выполнения 171 ч, или 7 дней и 3 ч.

Теперь давайте получим время выполнения для нескольких возможных замещающих платформ.

Пример замены CPU: Skylake Gold 6152 с частотой 2.10 ГГц и 36 физическими ядрами

Единственное отличие для этого прогона заключается в том, что мы увеличиваем 16 процессоров до 36 для mpirun с помощью следующей ниже команды:

```
mpirun -n 36 --bind-to core ./clover_leaf
```

Время работы системы Skylake составляет 273.3 с или 0.55 с за цикл. Это дало бы время выполнения типичной прикладной задачи, равное 76.4 ч, или 3 дням и 4 ч.

Неплохо! Меньше чем на половину рассчитанного ранее времени. Вы почти готовы приобрести эту систему, но затем думаете, что, возможно, вам следует проверить те GPU-процессоры, о которых вы слышали.

Пример замены GPU: V100

У CloverLeaf есть версия CUDA, которая работает на V100. Вы измеряете производительность с помощью следующих ниже шагов.

1. Клонировать Cloverleaf.git с

   ```
   git clone --recursive git@github.com:UK-MAC/CloverLeaf.git CloverLeaf
   ```

2. Затем набрать:

   ```
   cd CloverLeaf_CUDA
   ```

3. Добавить архитектурные флаги CUDA для Volta в файл makefile в текущий список архитектур CODE_GEN:

   ```
   CODE_GEN_VOLTA=-arch=sm_70
   ```

4. Добавить путь к библиотеке CUDA CUDART, если это необходимо:

   ```
   make COMPILER=GNU NV_ARCH=VOLTA
   sed -e '1,$s/end_step=2955/end_step=500/' clover_bm64.in >clover.in
   ./clover_leaf
   ```

Ого! Прогон настолько быстр, что у вас даже нет времени выпить чашечку кофе. Но вот он: 59.3 с! Это составляет 0.12 с за цикл или, для полного объема тестовой задачи, 16.5 ч. По сравнению с системой Skylake это в 4.6 раза быстрее и в 10.4 раза быстрее, чем изначальная система Ivy Bridge. Только представьте, сколько еще работы вы могли бы выполнить за одну ночь, а не за несколько дней!

Является ли этот выигрыш в производительности типичным для GPU-процессоров? Выигрыш в производительности для приложений охватывает широкий диапазон, но эти результаты не являются чем-то необычным.

9.6.2 Сокращение энергопотребления с помощью GPU-процессоров

Энергозатраты приобретают все более важное значение для параллельных приложений. Там, где когда-то энергопотребление компьютеров не вызывало беспокойства, теперь энергозатраты на эксплуатацию компьютеров, дисков хранения и системы охлаждения быстро приближаются к тем же уровням, что и затраты на приобретение оборудования в течение всего срока службы вычислительной системы.

В гонке за экзомаштабными вычислениями одной из самых больших проблем является поддержание требований к мощности экзомаштабной системы примерно на уровне 20 МВт. Для сравнения, этого примерно

достаточно для питания 13 000 домов. В центрах обработки данных просто недостаточно установленной мощности, чтобы выйти за рамки этого уровня. На другом конце спектра смартфоны, планшеты и ноутбуки работают от батарей с ограниченным количеством доступной энергии (между зарядами). На этих устройствах бывает полезно сосредотачиваться на снижении энергозатрат на вычисления, чтобы продлить срок службы батареи. К счастью, агрессивные усилия по сокращению энергопотребления позволили удерживать разумные темпы увеличения энергопотребления.

Точный расчет энергозатрат приложения является сложной задачей без прямых измерений энергопотребления. Однако вы можете получить более высокую границу затрат, умножив расчетную тепловую мощность (TDP) производителя на время выполнения приложения и число используемых процессоров. Расчетная тепловая мощность – это скорость, с которой энергия расходуется при типичных рабочих нагрузках. Энергопотребление для вашего приложения можно оценить по формуле:

$$\textit{Энергия} = (N \text{ Процессоров}) \times (R \text{ Вт/Процессор}) \times (T \text{ часов}),$$

где *Энергия* – это энергопотребление, N – число процессоров, R – это расчетная тепловая мощность, а T – время выполнения приложения. Давайте сравним систему на базе GPU-процессоров с примерно эквивалентной системой на базе CPU (табл. 9.6). Мы предположим, что наше приложение привязано к памяти, поэтому мы рассчитаем энергозатраты и потребление для системы 10 Тб/с.

Таблица 9.6 Конструирование системы на базе GPU- и CPU-процессоров с пропускной способностью 10 Тб/с

	NVIDIA V100	Intel CPU Skylake Gold 6152
Число	12 GPU-процессоров	45 процессоров (CPU)
Пропускная способность	12 × 850 Гб/с = 10.2 Тб/с	45 × 224 Гб/с = 10.1 Тб/с
Стоимость	12 × $11 000 = $132 000	45 × $3800 = $171 000
Мощность	300 Вт на GPU	140 Вт на CPU
Электроэнергия	86.4 кВт·ч	151.2 кВт·ч

В целях расчета энергозатрат за один день мы берем технические характеристики из табл. 9.6 и рассчитываем номинальные затраты на электроэнергию.

Пример: расчетная тепловая мощность для CPU Intel Xeon Gold 6152 с 22 ядрами

22-ядерный CPU Intel Xeon Gold 6152 имеет расчетную тепловую мощность (TDP) 140 Вт. Предположим, что ваше приложение использует 15 из этих процессоров в течение 24 ч для полного завершения работы. Расчетное энергопотребление вашего приложения составляет:

Энергия = (45 Процессоров) × (140 Вт/Процессоры) × (24 ч) = 151.2 кВт·ч.

Это значительное количество энергии! Вашего энергопотребления в указанном расчете достаточно, чтобы обеспечить энергией семь домов в течение тех же самых 24 ч.

В общем случае GPU-процессоры имеют более высокую расчетную тепловую мощность, чем CPU-процессоры (300 Вт против 140 Вт из табл. 9.6), поэтому они потребляют энергию с большей скоростью. Но GPU-процессоры потенциально могут сокращать время выполнения либо потребовать совсем немного для выполнения ваших вычислений. Можно использовать ту же формулу, что и раньше, где N теперь рассматривается как число GPU-процессоров.

Пример: расчетная тепловая мощность для платформы с многочисленными GPU

Предположим, что вы перенесли свое приложение на платформу с многочисленными GPU. Теперь можете выполнить свое приложение на четырех GPU NVIDIA Tesla V100 за 24 ч. GPU Tesla V100 от NVIDIA имеет максимальную расчетную тепловую мощность 300 Вт. Предполагаемое энергопотребление вашего приложения состав кВт·ч ляет:

$$\text{Энергия} = (12\ \text{GPU}) \times (300\ \text{Вт}/\ \text{GPU-процессоры}) \times (24\ \text{ч}) = 86.4\ \text{кВт·ч}.$$

В этом примере приложение, ускоренное за счет GPU, работает с гораздо меньшими энергозатратами по сравнению с версией только на базе CPU. Обратите внимание, что в этом случае, несмотря на то что время до решения остается прежним, энергозатраты сокращаются примерно на 40 %! Мы также видим снижение первоначальных затрат на оборудование на 20 %, но мы не учли затраты на хостовый CPU для GPU-процессоров.

Теперь мы видим, что у системы на базе GPU есть большой потенциал, но использованные нами номинальные значения бывают весьма далеки от реальности. Мы могли бы уточнить эту оценку дополнительно, получив измеренные показатели производительности и энергопотребления для нашего алгоритма.

Для достижения снижения энергозатрат с помощью ускорительных устройств на базе GPU требуется, чтобы приложение демонстрировало достаточный параллелизм и чтобы ресурсы устройства использовались эффективно. В гипотетическом примере нам удалось сократить энергопотребление вдвое во время работы на 12 GPU для того же времени, которое требуется для исполнения на 45 полностью подписанных CPU. Формула энергопотребления предлагает и другие стратегии снижения энергозатрат. Мы обсудим эти стратегии позже, но важно отметить, что в целом GPU потребляет энергии в единицу времени больше, чем CPU. Мы начнем с инспектирования энергопотребления между одним процессором CPU и одним GPU.

В листинге 9.3 показано, как строить график энергопотребления и коэффициента полезного использования GPU V100.

Пример: мониторинг энергопотребления GPU в течение всего срока жизни приложения

Давайте вернемся к задаче CloverLeaf, с которой мы столкнулись ранее на GPU V100. Мы использовали инструмент nvidia-smi (Интерфейс управления системой NVIDIA) для сбора метрик производительности прогона, включая мощность и коэффициент полезного использования GPU. Для этого перед выполнением нашего приложения мы выполнили следующую ниже команду:

```
nvidia-smi dmon -i 0 --select pumct -c 65 --options DT\
  --filename gpu_monitoring.log &
```

В следующей ниже таблице показаны опции команды nvidia-smi.

`dmon`	Собирает данные мониторинга
`-i 0`	Опрашивает устройство на базе GPU под номером 0
`--select pumct`	Выбирает мощность [p], коэффициент полезного использования [u], потребление памяти [m], такты [c], пропускную способность PCI. Можно также использовать `-s` как сокращение для `--select`
`-c 65`	Собирает 65 образцов. По умолчанию время равно 1 с
`-d <#>`	Изменяет интервал отбора образцов
`--options DT`	Добавляет мониторинговые данные соответственно с датой в формате ГГГГММДД и временем в формате ЧЧ:ММ::СС
`--filename <name>`	Записывает выходные данные в указанное имя файла
`&`	Помещает задание в фоновый режим, чтобы иметь возможность выполнять свое приложение

Сокращенный вывод данных в файл выглядит следующим образом:

```
Время      pwr gtemp mtemp   sm mem enc dec    fb bar1 mclk pclk rxpci txpci
HH:MM:SS     W     C     C    %   %   %   %    MB   MB  MHz  MHz  MB/s  MB/s
21:36:47    64    43    41   24  28   0   0     0    0  877 1530     0     0
21:36:48   176    44    44   96 100   0   0 11181    0  877 1530     0     0
21:36:49   174    45    45 100 100   0   0 11181    0  877 1530     0     0
...
```

Листинг 9.3 Построение графика данных энергопотребления и коэффициента полезного использования из nvidia-smi

power_plot.py

```
 1 import matplotlib.pyplot as plt
 2 import numpy as np
 3 import re
 4 from scipy.integrate import simps
 5
 6 fig, ax1 = plt.subplots()
```

```
 7
 8 gpu_power = []
 9 gpu_time = []
10 sm_utilization = []
11
12 # Собрать данные из файла, игнорировать пустые строки
13 data = open('gpu_monitoring.log', 'r')
14
15 count = 0
16 energy = 0.0
17 nominal_energy = 0.0
18
19 for line in data:
20     if re.match('^ 2019',line):
21         line = line.rstrip("\n")
22         dummy, dummy, dummy, gpu_power_in, dummy, dummy,
             sm_utilization_in, dummy,
             dummy, dummy, dummy, dummy, dummy, dummy, dummy, dummy = line.split()
23     if (float(sm_utilization_in) > 80):
24         gpu_power.append(float(gpu_power_in))
25         sm_utilization.append(float(sm_utilization_in))
26         gpu_time.append(count)
27         count = count + 1
28         energy = energy + float(gpu_power_in)*1.0
29         nominal_energy = nominal_energy + float(300.0)*1.0
30
31 print(energy, "watts-secs", simps(gpu_power, gpu_time))
32 print(nominal_energy, "Вт-сек", " ratio
     ",energy/nominal_energy*100.0)
33
34 ax1.plot(gpu_time, gpu_power, "o", linestyle='-', color='red')
35 ax1.fill_between(gpu_time, gpu_power, color='orange')
36 ax1.set_xlabel('Время (с)',fontsize=16)
37 ax1.set_ylabel('Энергопотребление (Вт)',fontsize=16, color='red')
38 #ax1.set_title(''Энергопотребление GPU из nvidia-smi')
39
40 ax2 = ax1.twinx() # инстанцирует вторые оси, которые делят между собой те же оси x
41
42 ax2.plot(gpu_time, sm_utilization, "o", linestyle='-', color='green')
43 ax2.set_ylabel('Коэффициент полезного использования GPU (%)',fontsize=16,
                 color='green')
44
45 fig.tight_layout()
56 plt.savefig("power.pdf")
57 plt.savefig("power.svg")
58 plt.savefig("power.png", dpi=600)
59
60 plt.show()
```

Интегрирует мощность, умноженную на время, затрачиваемое на получение энергии в Вт/с

Получает энергопотребление на основе номинальной спецификации мощности

Выводит рассчитанную энергию и использует интеграционную функцию simps из scipy

Вычисляет фактическое и номинальное энергопотребление

На рис. 9.11 показан результирующий график. В то же время мы интегрируем область под кривой, чтобы получить энергопотребление. Обратите внимание, что даже при коэффициенте полезного использования

в 100 % мощность составляет всего около 61 % от номинальной спецификации мощности GPU. На холостом ходу энергопотребление GPU составляет около 20 % от номинальной величины. Это показывает, что реальный коэффициент энергопотребления GPU-процессоров значительно ниже, чем оценки, основанные на номинальных спецификациях. Процессоры также ниже номинального объема, но, вероятно, не на такой большой процент от их номинального уровня.

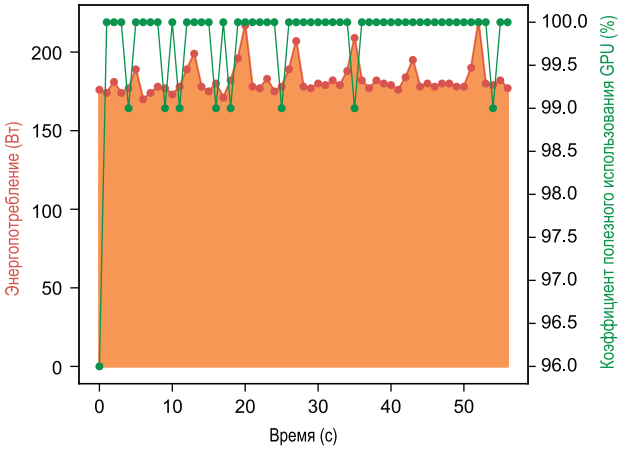


Рис. 9.11 Энергопотребление для задачи CloverLeaf, работающей на V100. Мы интегрируем под кривой мощности, чтобы получить 10.4 кДж за прогон, который длился около 60 с. Уровень энергопотребления составляет около 61 % от номинальной спецификации мощности для GPU V100

Когда платформы с многочисленными GPU будут экономить вам энергию?

Как правило, эффективность параллельной работы снижается по мере добавления новых CPU или GPU (помните закон Амдала из раздела 1.2?), а стоимость вычислительной работы возрастает. Иногда существуют постоянные затраты (например, на хранение), связанные с совокупным временем выполнения задания, которые сокращаются, если задание завершается раньше и данные могут быть переданы или удалены. Однако в обычной ситуации у вас есть комплект выполняемых заданий с выбором числа процессоров для каждого задания. В следующем ниже примере показаны компромиссы в этой ситуации.

Пример: комплект параллельных заданий для выполнения

У вас есть 100 заданий для выполнения примерно одного типа и длины. Вы можете выполнить эти задания на 20 процессорах либо на 40 процессорах. На 20 процессорах выполнение заданий занимает по 10 ч каждое. Параллельная эффективность в этом диапазоне процессоров составляет около

80 %. Облачный кластер, к которому у вас есть доступ, имеет 200 процессоров. Давайте рассмотрим два сценария.

Случай 1. Выполнение по 10 заданий за раз с 20 процессорами дает нам вот такое решение:

Суммарное время выполнение комплекта = 10 ч × 100/10 = 100 ч.

Случай 2. Выполнение по 5 заданий за раз с 40 процессорами.

Увеличение числа процессоров до 40 снижает время выполнения до 5 ч, если параллельная эффективность является идеальной. Но она эффективна лишь на 80 %, поэтому время выполнения составляет 6.25 ч. Как мы получили это число? Допустим, что 20 процессоров является базовым случаем. Для удвоения числа процессоров формула параллельной эффективности имеет вид:

$$P_{эффективность} = S/P_{множ} = 80\ \%.$$

S – это ускорение решения задачи. Процессорный множитель, $P_{множ}$, равен 2. В результате решения уравнение задачи ускорения мы получаем:

$$S = 0.8 \times P_{множ} = 0.8 \times 2 = 1.6.$$

Теперь мы используем уравнение ускорения для вычисления нового времени (TN от англ. *new time*):

$$S = T_{базис}/T_{нов}.$$

Для этой формы уравнения ускорения мы используем $T_{базис}$ вместо $T_{последоват}$. Параллельная эффективность часто снижается по мере добавления процессоров, поэтому мы хотим иметь связь в этой точке на кривой эффективности. Решая уравнение для $T_{нов}$, мы получаем:

$$T_{нов} = T_{базис}/C = 10/1.6 = 6.25 \text{ ч}.$$

Учитывая, что это время выполнения относится к 40 процессорам, теперь мы можем рассчитать суммарное время выполнения комплекта:

Суммарное время комплекта = 6.25 ч × 100/5 = 125 ч.

Таким образом, сценарий, используемый в случае 1, выполняет тот же объем работы намного быстрее.

Этот пример показывает, что, если мы оптимизируем время выполнения под крупный комплект заданий, часто лучше использовать меньше параллелизма. Напротив, если мы больше заботимся о времени выполнения одного задания, то лучше использовать большее число процессоров.

9.6.3 Снижение в затратах на облачные вычисления за счет использования GPU-процессоров

Службы облачных вычислений от Google и Amazon позволяют сочетать ваши рабочие нагрузки с широким спектром типов и требований к вычислительным серверам.

- Если ваше приложение ограничено памятью, то вы можете использовать GPU с более низким соотношением флопов к загрузкам при меньшей стоимости.
- Если вас больше волнует время выполнения, то вы можете добавить больше GPU или CPU.
- Если ваши крайние сроки не настолько серьезны, то вы можете использовать свободные ресурсы при значительном снижении в стоимости.

По мере того как стоимость вычислений становится заметнее с помощью служб облачных вычислений, задача оптимизации производительности вашего приложения становится более приоритетной. Преимущество облачных вычислений заключается в том, что они предоставляют вам доступ к более широкому спектру оборудования, чем вы можете иметь на месте, и предоставляют больше опций для сочетания оборудования с рабочей нагрузкой.

9.7 Когда следует использовать GPU-процессоры

GPU не являются общецелевыми процессорами. Они наиболее подходят в ситуациях, когда рабочая вычислительная нагрузка похожа на графическую рабочую нагрузку – большое число идентичных операций. В некоторых областях GPU-процессоры все еще работают плохо, хотя с каждой итерацией развития аппаратного и программного обеспечения GPU в некоторых из них отыскиваются технологические решения.

- *Нехватка параллелизма* – перефразируя Человека-паука, «большая мощь сопровождается большой потребностью в параллелизме». Если у вас нет параллелизма, то GPU-процессоры мало что могут для вас сделать. Это первый закон программирования GPGPU.
- *Нерегулярный доступ к памяти* – CPU справляются с ним с трудом. В данной ситуации массовый параллелизм GPU-процессоров не приносит никакой пользы. Это второй закон программирования GPGPU.
- *Дивергенция (расхождение) потоков* – все потоки на GPU-процессорах исполняются на каждой ветви. Это характерно для архитектур SIMD и SIMT (см. раздел 1.4). Малое число коротких ответвлений – это нормально, но чрезмерно ветвящиеся пути работают слабо.
- *Требования к динамической памяти* – выделение памяти выполняется на CPU, что серьезно лимитирует алгоритмы, требующие определения размеров памяти на лету.
- *Рекурсивные алгоритмы* – GPU-процессоры имеют лимитированные ресурсы стековой памяти, а поставщики часто заявляют, что рекурсия не поддерживается. Тем не менее, как было продемонстрировано в разделе 5.5.2, лимитированный объем рекурсии работает в алгоритмах переназначения между вычислительными сетками.

9.8 Материалы для дальнейшего изучения

Архитектуры GPU продолжают развиваться с каждой итерацией разработки оборудования. Мы рекомендуем вам продолжать следить за последними разработками и инновациями. С самого начала архитектуры GPU были в первую очередь ориентированы на производительность графики. Но рынок также расширился за счет машинного обучения и вычислений.

9.8.1 Дополнительное чтение

Для более подробного обсуждения производительности приложения сравнительного тестирования STREAM Benchmark и того, как она варьируется в зависимости от языков параллельного программирования, мы отсылаем вас к следующей ниже работе:

- Т. Дикин, Дж. Прайс и соавт., «Сравнительное тестирование достижимой пропускной способности памяти многоядерных процессоров в различных моделях параллельного программирования», (T. Deakin, J. Price, et al., Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models, *GPU-STREAM*, v2.0, 2016). Статья представлена в рабочей группе по переносимым моделям производительности для многоядерных и акселераторных систем (P^3MA) на конференции ISC High Performance, Франкфурт, Германия.

Хороший ресурс по модели контура крыши для GPU-процессоров можно найти в лаборатории Лоуренса Беркли. Хорошей отправной точкой является статья:

- Шарлин Янг и Сэмюэл Уильямс, «Анализ производительности приложений, ускоренных за счет GPU, с использованием модели контура крыши», (Charlene Yang and Samuel Williams, Performance Analysis of GPU-Accelerated Applications using the Roofline Model, GPU Technology Conference, 2019), доступная по адресу https://crd.lbl.gov/assets/Uploads/GTC19-Roofline.pdf.

В этой главе мы представили упрощенный взгляд на модели смешанного тестирования производительности исходя из простых требований к производительности приложений. В следующей ниже статье представлена более тщательная процедура, учитывающая сложности реальных приложений:

- Элиас Константинидис и Яннис Котронис, «Количественная модель контура крыши для оценивания производительности ядра GPU с использованием сравнительных микротестов и профилирования аппаратных метрик» (Elias Konstantinidis and Yiannis Cotronis, A quantitative roofline model for GPU kernel performance estimation using micro-benchmarks and hardware metric profiling. *Journal of Parallel and Distributed Computing* 107 (2017): 37–56).

9.8.2 Упражнения

1 В табл. 9.7 показана достижимая производительность для приложения с одним флопом за загрузку. Посмотрите на текущие рыночные цены на GPU и заполните последние два столбца, чтобы получить флоп за доллар для каждого GPU. Каким выглядит самое лучшее соотношение цены и качества? Если время выполнения вашего приложения является наиболее важным критерием, то какой GPU лучше всего приобрести?

Таблица 9.7 Достижимая производительность приложения с одним флопом за загрузку с разными GPU

GPU	Достижимая производительность Гфлопы/с	Цена	Флопы/$
V100	108.23		
Vega 20	91.38		
P100	74.69		
GeForce GTX1080Ti	44.58		
Quadro K6000	31.25		
Tesla S2050	18.50		

2 Измерьте потоковую пропускную способность вашего GPU или другого выбранного GPU. Как это соотносится с теми, которые представлены в этой главе?
3 Используйте инструмент производительности likwid, чтобы узнать требования к мощности процессора для приложения CloverLeaf в системе, в которой у вас есть доступ к счетчикам силового оборудования.

Резюме

- Система CPU-GPU может обеспечивать мощный импульс для многих параллельных приложений. Это следует учитывать для любого приложения с большим объемом параллельной работы.
- GPU-компонент системы на самом деле является параллельным общецелевым ускорителем. Это означает, что ему следует давать параллельную часть работы.
- Передача данных по шине PCI и пропускная способность памяти являются наиболее распространенными узкими местами производительности в системах CPU-GPU. Управление передачей данных и использованием памяти имеет важное значение для хорошей производительности.
- Вы найдете широкий спектр GPU, предназначенных для разных рабочих нагрузок. Выбор наиболее подходящей модели обеспечивает наилучшее соотношение цены и производительности.
- GPU-процессоры могут сокращать показатель времени до решения и энергозатраты. А это может быть первостепенным мотиватором при переносе приложения на GPU-процессоры.

10 Модель программирования GPU

Эта глава охватывает следующие ниже темы:

- разработку общей модели программирования GPU;
- понимание того, как она соотносится с оборудованием разных поставщиков;
- усвоение деталей модели программирования, которые влияют на производительность;
- соотнесение модели программирования с разными языками программирования GPU.

В этой главе мы постепенно разовьем абстрактную модель выполнения работы на GPU-процессорах. Указанная модель программирования подходит для самых разных GPU-устройств от разных поставщиков и разных их типов у каждого поставщика. Она также проще, чем то, что происходит на реальном оборудовании, и охватывает только существенные аспекты, необходимые для разработки приложения. К счастью, различные GPU по своей структуре имеют много общего. И она является естественным результатом потребностей высокопроизводительных графических приложений.

Выбор структур данных и алгоритмов оказывает долгосрочное влияние на производительность и простоту программирования GPU. Имея хорошую мысленную модель GPU, вы сможете планировать способы со-

отнесения структур данных и алгоритмов с параллелизмом GPU. Наша первостепенная задача, в особенности применительно к GPU-процессорам, как разработчиков приложений состоит в том, чтобы проявить как можно больше параллелизма. Мобилизуя тысячи потоков, нам нужно коренным образом изменить работу в таком ключе, чтобы между потоками распределялось много малых операционных задач. В языке GPU, как и в любом другом языке параллельного программирования, должно существовать несколько компонентов. Это способ:

- выражать вычислительные циклы в параллельной для GPU форме (см. раздел 10.2);
- перемещать данные между хостовым CPU и вычислительным устройством GPU (см. раздел 10.2.4);
- координировать работу между потоками, которые нужны для редукции (см. раздел 10.4).

Посмотрите, как эти три компонента осуществлены в каждом языке программирования GPU. В некоторых языках для имплементирования необходимых операций вы напрямую управляете некоторыми аспектами, в других же вы полагаетесь на компилятор или шаблонное программирование. Хотя на первый взгляд операции GPU могут показаться загадочными, они не так уж сильно отличаются от того, что требуется для параллельного кода на CPU. Мы должны писать безопасные для мелкозернистого параллелизма циклы, иногда именуемые `do concurrent` на Fortran, `forall` или `foreach` на C/C++. Мы должны думать о перемещении данных между узлами, процессами и процессором. Мы также должны иметь специальные механизмы для редукций.

В нативных вычислительных языках GPU, таких как CUDA и OpenCL, модель программирования представлена как составная часть языка. Указанные языки GPU рассматриваются в главе 12. В этой главе вы будете управлять многими аспектами параллелизации GPU в своей программе явным образом. Но наша модель программирования позволит вам лучше подготовиться к принятию важных программных решений в отношении более высокой производительности и масштабирования в широком диапазоне оборудования GPU.

Если вы используете язык программирования более высокого уровня, такой как описанные в главе 11 прагма-ориентированные языки GPU, то действительно ли вам так уж нужно понимать все детали модели программирования GPU? Даже с учетом прагм все равно полезно понимать принцип распределения работы. Когда вы используете прагму, вы пытаетесь направлять компилятор и библиотеку по правильному пути. В некотором смысле это сложнее, чем писать программу напрямую.

Цель этой главы состоит в том, чтобы помочь вам разработать дизайн приложения для GPU. Он в основном не зависит от какого-то языка программирования GPU. Есть вопросы, на которые следует ответить заранее. Как вы будете организовывать свою работу, и какой производительности можно ожидать? Или более простой вопрос о том, следует ли вообще переносить ваше приложение на GPU или же лучше остаться с CPU? Платформы на GPU-процессорах, обещающие более низкое энер-

гопотребление и повышение производительности на порядок, привлекательны. Но они не панацея для каждого приложения и случая использования. Давайте погрузимся в детали модели программирования GPU и посмотрим, что она может для вас сделать.

ПРИМЕЧАНИЕ Мы рекомендуем вам сверяться с примерами этой главы, расположенными по адресу https://github.com/EssentialsofParallelComputing/Chapter10.

10.1 Абстракции программирования GPU: широко распространенная структура

Абстракции программирования GPU возможны по ряду причин. Базовые характеристики, которые мы вскоре разведаем чуть-чуть подробнее, перечислены ниже. Затем мы коротко остановимся на нескольких базовых терминах параллелизма GPU.

- Операции с графикой имеют массовый параллелизм.
- Операции невозможно координировать между операционными задачами.

10.1.1 Массовый параллелизм

Абстракции основаны на компонентах, необходимых для высокопроизводительной графики с GPU-процессорами. Рабочие потоки GPU обладают некоторыми особыми характеристиками, которые помогают стимулировать общность технических приемов обработки на GPU. Для высокочастотной и высококачественной графики необходимо обрабатывать и выводить на экран много пикселов, треугольников и многоугольников.

Из-за большого объема данных GPU-процессоры обладают массивным параллелизмом. Операции на данных, как правило, идентичны, поэтому в GPU-процессорах используются схожие приемы, применяя одну единственную команду к многочисленным элементам данных для достижения другого уровня эффективности. На рис. 10.1 показаны общие абстракции программирования, широко распространенные среди различных поставщиков и моделей GPU. Их можно свести к трем или четырем базовым техникам.

Мы начинаем с вычислительного домена и итеративно разбиваем работу, имея следующие ниже компоненты. Мы обсудим каждое из этих подразделений работы в разделах 10.1.4-10.1.8:

- декомпозицию данных;
- порционную работу для манипулирования некоторой совместной локальной памятью;
- оперирование на нескольких элементах данных одной командой;
- векторизацию (на нескольких GPU).

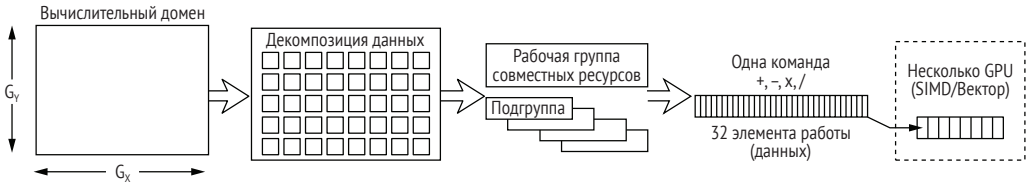


Рис. 10.1 Наша мысленная модель параллелизации GPU содержит общие абстракции программирования, широко распространенные среди большинства оборудования GPU.

Из этих параллельных абстракций GPU следует один важный вывод, а именно что, в принципе, существует три или, может быть, четыре разных уровня параллелизации, которые вы можете применять к вычислительному циклу. В изначальном случае использования графики нет особой необходимости выходить за рамки двух или трех размерностей и соответствующего числа уровней параллелизации. Если ваш алгоритм имеет больше размерностей или уровней, то вы должны скомбинировать несколько вычислительных циклов, чтобы полностью распараллелить вашу задачу.

10.1.2 Неспособность поддерживать координацию среди операционных задач

Графические рабочие нагрузки не требуют большой координации в рамках операций. Но, как мы увидим в последующих разделах, существуют алгоритмы, такие как редукции, которые требуют координации. Нам придется разработать сложные схемы, чтобы манипулировать этими ситуациями.

10.1.3 Терминология для параллелизма GPU

Терминология для компонентов параллелизма GPU различается от поставщика к поставщику, что вносит некоторую путаницу при чтении документации по программированию или статей. В целях оказания помощи при поиске соответствий среди используемых терминов мы резюмировали официальные термины каждого поставщика в табл. 10.1.

OpenCL – это открытый стандарт программирования GPU, поэтому мы используем его терминологию в качестве базовой. OpenCL работает на всем оборудовании GPU и многих других устройствах, таких как CPU, и даже на более экзотическом оборудовании, таком как программируемые пользователем вентильные матрицы (FPGA-матрицы) и другие встраиваемые устройства. Язык CUDA, проприетарный язык NVIDIA для своих GPU, наиболее широко применяется для вычислений на GPU-процессорах и, следовательно, используется в значительной части документации по программированию GPU. HIP (Интерфейс гетерогенных вычислений для обеспечения переносимости) – это переносимый производный от CUDA язык, разработанный в AMD для своих

GPU. В нем используется та же терминология, что и в CUDA. В нативном компиляторе AMD гетерогенных вычислений (HC) и языке C++ AMP от Microsoft используется много одинаковых терминов. (На момент написания этой книги C++ AMP находится в режиме обслуживания и еще не перешел в стадию активной разработки.) При попытке получить переносимую производительность также важно учитывать соответствующие функциональности и условия для CPU, как показано в последнем столбце табл. 10.1.

Таблица 10.1 Абстракции программирования и связанная с ними терминология для GPU-процессоров

OpenCL	CUDA	HIP	AMD GPU (компилятор HC)	C++ AMP	CPU
NDRange (N-мерный диапазон)	Решетка	Решетка	Экстент (протяженность)	Экстент (протяженность)	Стандартные границы цикла или индексные множества с блокированием цикла
Рабочая группа	Блок или поточный блок	Блок	Плитка	Плитка	Блок цикла
Подгруппа или фронт волны	Варп (переплетение)	Варп	Фронт волны	Отсутствует	Длина SIMD
Элемент работы	Поток	Поток	Поток	Поток	Поток

10.1.4 Декомпозиция данных на независимые единицы работы: NDRange или решетка

Техника декомпозиции данных лежит в самом сердце того, как GPU-процессоры обретают производительность. GPU разбивают задачу на множество более мелких блоков данных. Потом они разбивают их снова и снова.

GPU должны рисовать много треугольников и многоугольников, чтобы генерировать высокую частоту кадров. Указанные операции полностью независимы друг от друга. По этой причине верхнеуровневая декомпозиция данных для вычислительной работы на GPU также создает независимую и асинхронную работу.

Из-за большого объема выполняемой работы GPU скрывают задержку (пробуксовки из-за загрузки памяти), переключаясь на другую рабочую группу, готовую к вычислениям. На рис. 10.2 показан случай, когда из-за ресурсных ограничений можно планировать срабатывание только четырех подгрупп (варпов или волновых фронтов). Когда подгруппы достигают чтения памяти и начинают пробуксовывать, исполнение переключается на другие подгруппы. Переключатель исполнения, также именуемый переключателем контекста, скрывает задержку не с помощью глубокой иерархии кеша, а с помощью компиляции. *Если у вас есть только один поток с одной командой на одном элементе данных, то GPU будет медленным, потому что у него нет возможности скрывать задержку. Но если у вас много оперируемых данных, то он будет невероятно быстрым.*

Рис. 10.2 Планировщик подгрупп (варпов) GPU переключается на другие подгруппы, чтобы охватывать операции чтения памяти и пробуксовки команд. Многочисленные рабочие группы позволяют выполнять работу даже при синхронизировании рабочей группы

В табл. 10.2 показаны аппаратные лимиты для текущих планировщиков NVIDIA и AMD. Для этих устройств нам нужно большое число кандидатных рабочих групп и подгрупп, чтобы обрабатывающие элементы были заняты.

Таблица 10.2 Лимиты планировщика подгрупп (варпов или волнового фронта) GPU

	NVIDIA Volta and Ampere	AMD MI50
Активное число подгрупп в расчете на вычислительный модуль	64	40
Активное число рабочих групп в расчете на вычислительный модуль	32	40
Выбранные подгруппы для исполнения в расчете на вычислительный модуль	4	4
Размер подгруппы (варпа либо волнового фронта)	32	64

Перемещение данных и, в частности, перемещение данных вверх и вниз по иерархии кеша является существенной частью энергозатрат процессора. Следовательно, сокращение потребности в глубокой иерархии кеша имеет некоторые существенные выгоды. Наблюдается значительное сокращение энергопотребления. Кроме того, на процессоре освобождается много драгоценного кремниевого пространства. Затем это пространство может быть заполнено еще большим числом арифметико-логических модулей (ALU).

Мы показываем операцию декомпозиции данных на рис. 10.3, где двухмерный вычислительный домен разбивается на более мелкие двухмерные блоки данных. В OpenCL он называется *NDRange*, сокращенно от *N*-мерного диапазона (термин «решетка» в CUDA выглядит несколько удобнее). NDRange в данном случае представляет собой множество размером 3×3, состоящее из плиток размером 8×8. Процесс декомпозиции данных разбивает глобальный вычислительный домен, $G_y \times G_x$, на более

мелкие блоки или плитки размером $T_y \times T_x$.

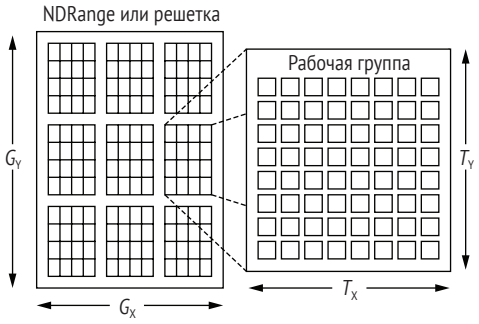


Рис. 10.3 Разбиение вычислительного домена на малые независимые единицы работы

Давайте рассмотрим пример подробнее, чтобы увидеть, к чему приводит этот шаг.

Пример: декомпозиция данных двухмерного вычислительного домена размером 1024×1024

Если задать размер плитки равным 16×8, то декомпозиция данных будет такой:

$$NT_x = G_x/T_x = 1024/16 = 64;$$

$$NT_y = G_y/T_y = 1024/8 = 128;$$

$$NT = 64 \times 128 = 8192 \text{ плитки.}$$

В этом первом уровне декомпозиции набор данных начинает разбиваться на большое число более мелких блоков или плиток. GPU-процессоры принимают несколько допущений о характеристиках рабочих групп, создаваемых в результате указанной декомпозиции данных. Эти допущения заключаются в том, что рабочие группы:

- полностью независимы и асинхронны;
- имеют доступ к глобальной и постоянной памяти.

Каждая созданная рабочая группа является независимой единицей работы. Это означает, что на каждой этой рабочей группе можно оперировать в любом порядке, обеспечивая уровень параллелизма для распределения по множеству вычислительных модулей на GPU. Это те же самые свойства, которые мы приписывали мелкозернистому параллелизму в разделе 7.3.6. Те же самые изменения в циклах в отношении независимых итераций для OpenMP и векторизации также делают возможным параллелизм GPU.

В табл. 10.3 показаны примеры того, как эта декомпозиция данных может происходить для одномерного, двухмерного и трехмерного вычислительных доменов. Для наилучшей производительности самый быстро

меняющийся размер плитки, T_x, должен быть кратен длине строки кеша, ширине шины памяти либо размеру подгруппы (волнового фронта или варпа). Число плиток, NT, в совокупности и в каждой размерности, приводит к распределению большого числа рабочих групп (плиток) по вычислительным двигателям и обрабатывающим элементам GPU.

Таблица 10.3 Декомпозиция вычислительного домена на плитки или блоки

	Одномерный	Малый двухмерный	Крупный двухмерный	Трехмерный
Глобальный размер	1 048 576	1024×1024	1024×1024	128×128×128
$T_z \times T_y \times T_x$	128	8×8	8×16	4×4×8
Размер плитки	128	64	128	128
$NT_z \times NT_y \times NT_x$	8192	128×128	128×64	32×32×16
NT (число рабочих групп)	8192	16 384	8192	16 384

В алгоритмах, которым требуется соседняя информация, оптимальный размер плитки для доступов к памяти должен быть сбалансирован с получением минимальной площади поверхности плитки (рис. 10.4). Соседние данные должны загружаться более одного раза для смежных плиток, придавая важность этому соображению.

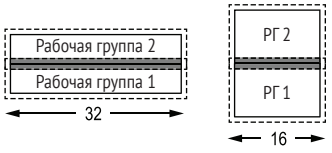


Рис. 10.4 Каждая рабочая группа должна загружать соседние данные из пунктирного прямоугольника, что приводит к дублированию загрузок в затененных областях, где слева требуется больше дублирующих загрузок. Это должно балансироваться относительно оптимальных загрузок сплошных данных в направлении x

10.1.5 Рабочие группы обеспечивают оптимальную по размеру порцию работы

Рабочая группа распределяет работу по потокам на вычислительном устройстве. Каждая модель GPU имеет максимальный размер, который задается для оборудования. OpenCL сообщает об этом как `CL_DEVICE_MAX_WORK_GROUP_SIZE` в своем запросе устройства. PGI сообщает об этом как `Maximum Threads per Block` (максимум потоков на блок) на выходе из своей команды `pgaccelinfo` (см. рис. 11.3). Максимальный размер рабочей группы обычно находится между 256 и 1024. Это просто максимум. Для вычислений указанные размеры рабочих групп обычно намного меньше, вследствие чего на каждый элемент работы или поток приходится больше ресурсов памяти.

Рабочая группа подразделяется на подгруппы или группы (рис. 10.5). Подгруппа – это множество потоков, которые исполняются в унисон. В NVIDIA размер варпа составляет 32 потока. В AMD это называется волновым фронтом, и размер обычно составляет 64 элемента работы. Размер рабочей группы должен быть кратен размеру подгруппы.

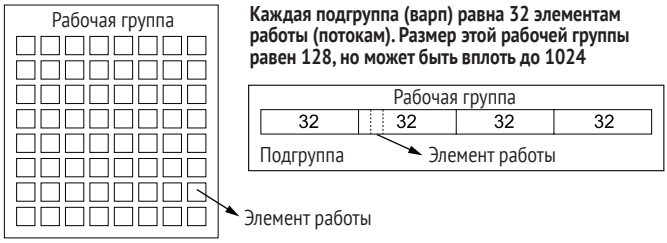


Рис. 10.5 Многомерная рабочая группа линеаризуется в одномерную ленту, в которой она разбивается на подгруппы из 32 или 64 элементов работы. По соображениям производительности рабочие группы должны быть кратны размеру подгрупп

Типичные характеристики рабочих групп на GPU-процессорах заключаются в том, что они:

- зациклены на обработку каждой подгруппы;
- имеют локальную память (совместную память) и другие ресурсы, совместно используемые в группе;
- могут синхронизировать внутри рабочей группы или подгруппы.

Локальная память обеспечивает быстрый доступ и может использоваться как своего рода программируемый кеш или блокнотная память (или сверхоперативная регистровая память). Если в рабочей группе одни и те же данные необходимы нескольким потокам, то производительность, как правило, можно повышать, загружая их в локальную память при запуске вычислительного ядра.

10.1.6 Подгруппы, варпы или волновые фронты исполняются в унисон

Для дальнейшей оптимизации графических операций GPU-процессоры распознают, что одни и те же операции могут выполняться на многих элементах данных. Поэтому GPU оптимизируются за счет оперирования на наборах данных одной командой, а не отдельными командами для каждого элемента. За счет этого сокращается число манипулируемых команд. Указанная техника на CPU называется «одна команда, несколько элементов данных» (SIMD). Все GPU эмулируют ее с помощью группы потоков, где она называется «одна команда, мультипоток» (SIMT). Смотрите раздел 1.4, где тема SIMD и SIMT обсуждается изначально.

Поскольку операция SIMT симулирует операции SIMD, она не обязательно ограничена опорным векторным оборудованием так же, как операции SIMD. Текущие операции SIMT исполняются в унисон, при этом каждый поток в подгруппе исполняет все пути посредством ветвления, если какой-либо один поток должен пройти через ветвь (рис. 10.6). Это похоже на то, как операция SIMD исполняется с помощью маски. Но, поскольку операция SIMT эмулируется, этот принцип может быть ослаблен за счет большей гибкости в командном конвейере, в котором может поддерживаться более одной команды.

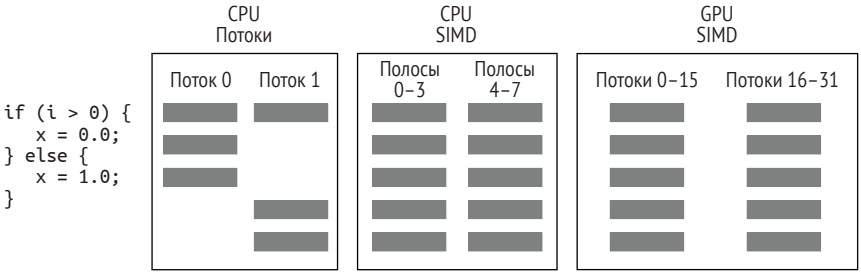


Рис. 10.6 Затененные прямоугольники показывают исполняемые команды языка по потокам и полосам. Операции SIMD и SIMT исполняют все команды в унисон, применяя маски для тех, которые являются ложными. Крупные блоки условных конструктов могут вызывать проблемы в GPU-процессорах, связанные с расхождением ветвей

Малые разделы условных блоков в GPU-процессорах не оказывают существенного влияния на совокупную производительность. Но если некоторые потоки занимают время на тысячи циклов больше, чем другие, то возникает серьезная проблема. Если потоки сгруппированы в таком ключе, что все длинные ветви находятся в одной подгруппе (волновом фронте), то расхождение потоков будет незначительным или вообще отсутствовать.

10.1.7 Элемент работы: базовая единица операции

Базовая единица операции в OpenCL называется элементом работы (work item). В зависимости от имплементации оборудования указанный элемент работы может быть соотнесен с потоком или обрабатывающим ядром. В CUDA он просто называется потоком, потому что именно так это соотносится в GPU-процессорах NVIDIA. Называть элемент работы потоком значит смешивать модель программирования с тем, как она имплементирована в оборудовании, но программисту это немного понятнее.

Элемент работы может активировать еще один уровень параллелизма на GPU-процессорах с векторными аппаратными модулями, как показано на рис. 10.7. Эта модель операции также соотносится с CPU, где поток может выполнять векторную операцию.

10.1.8 SIMD- или векторное оборудование

Некоторые GPU также имеют векторные аппаратные модули и могут выполнять (векторные) операции SIMD в дополнение к операциям SIMT. В мире графики векторные модули обрабатывают пространственные или цветовые модели. Применение в научных вычислениях имеет более высокую сложность и не обязательно переносимо между видами оборудования GPU. Векторная операция выполняется для каждого элемента работы, увеличивая объем полезного использования ресурсов для вычислительного ядра. Но нередко существуют дополнительные векторные

регистры, которые компенсируют дополнительную работу. При хорошем применении эффективный объем полезного использования векторных модулей может значительно повышать производительность.

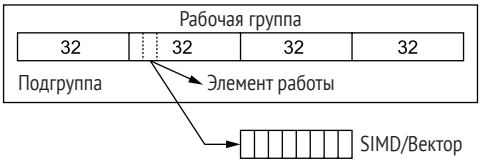


Рис. 10.7 Каждый элемент работы на GPU AMD или Intel может выполнять операции SIMD либо векторные операции. Он также хорошо соотносится с векторным модулем на CPU

Векторные операции доступны на языках OpenCL и AMD. Поскольку оборудование CUDA не имеет векторных модулей, такой же уровень поддержки отсутствует в языках CUDA. Тем не менее, программный код OpenCL с векторными операциями будет выполняться на оборудовании CUDA, поэтому его можно эмулировать на оборудовании CUDA.

10.2 Структура кода для модели программирования GPU

Теперь мы можем приступить к рассмотрению структуры кода для GPU, которая включает в себя модель программирования. Для удобства и общности мы будем называть CPU хостом и использовать термин «устройство» для обозначения GPU.

Модель программирования GPU отделяет тело цикла от диапазона массива или индексного множества, который применяется к функции. Тело цикла создает вычислительное ядро GPU. Индексное множество и аргументы будут использоваться на хосте для вызова ядра. На рис. 10.8 показано преобразование с переводом из стандартного цикла в тело вычислительного ядра GPU. В этом примере используется синтаксис OpenCL. Но ядро CUDA аналогично и заменяет вызов `get_global_id` на

```
gid = blockIdx.x *blockDim.x + threadIdx.x
```

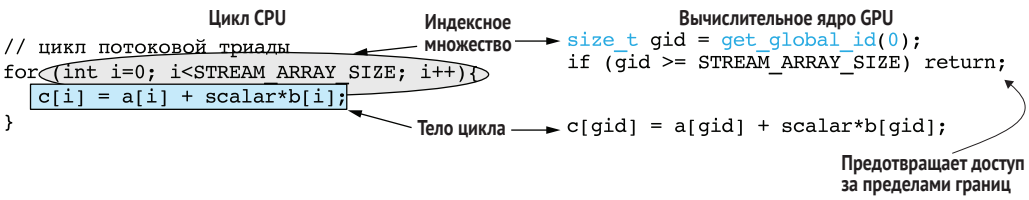


Рис. 10.8 Соответствие между стандартным циклом и структурой кода вычислительного ядра GPU

В следующих четырех разделах мы отдельно рассмотрим то, как тело цикла становится параллельным вычислительным ядром и как связывать его с индексом, заданном на хосте. Давайте разложим это на четыре шага.

1. Извлечь параллельное вычислительное ядро.
2. Соотнести локальные данные с глобальными данными.
3. Рассчитать декомпозицию на хосте в блоки данных.
4. Выделить любую необходимую память.

10.2.1 Программирование «Эго»: концепция параллельного вычислительного ядра

Программирование на GPU – идеальный язык для поколения «Эго». В вычислительном ядре все происходит относительно вас самих. Возьмем, к примеру,

```
c[i] = a[i] + scalar*b[i];
```

В этом выражении нет никакой информации о протяженности цикла. Этот цикл может иметь `i`, т. е. глобальный индекс `i`, который охватывает диапазон от 0 до 1000 либо только одно значение 22. Каждый элемент данных знает о том, что нужно делать с самим собой и только с собой. На самом деле мы имеем модель программирования «Эго», где я забочусь только о себе. Мощь этой модели состоит в том, что операции на каждом элементе данных становятся полностью независимыми. Давайте рассмотрим более сложный пример стенсильного оператора. Хотя у нас два индекса, `i` и `j`, и некоторые ссылки относятся к смежным значениям данных, приведенная ниже строка кода остается по-прежнему полностью определенной, как только мы задаем значения `i` и `j`.

```
xnew[j][i] = (x[j][i] + x[j][i-1] + x[j][i+1] + x[j-1][i] + x[j+1][i])/5.0;
```

Разделение тела цикла и индексного множества на C++ делается с помощью функторов либо посредством лямбда-выражений. В C++ лямбда-выражения существуют со времен стандарта C++ 11. Лямбды используются компиляторами для обеспечения переносимости единого исходного кода на CPU- или GPU-процессоры. В листинге 10.1 показана лямбда C++.

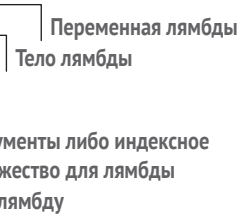
ОПРЕДЕЛЕНИЕ *Лямбда-выражения* – это безымянные локальные функции, которые могут назначаться переменной и использоваться локально либо передаваться в подпрограмму.

Листинг 10.1 Лямбда C++ для потоковой триады

`lambda.cc`

```
1 int main() {
2     const int N = 100;
3     double a[N], b[N], c[N];
4     double scalar = 0.5;
```

```
 5
 6     // c, a и b - допустимые указателями области видимости на устройстве или хосте
 7
 8     // Мы назначаем тело цикла переменной example_lambda
 9     auto example_lambda = [&] (int i) {
10         c[i] = a[i] + scalar * b[i];
11     };
12
13     for (int i = 0; i < N; i++)
14     {
15         example_lambda(i);
16     }
17 }
```



Лямбда-выражение состоит из четырех главных компонентов:

- *тела лямбды* – функция, исполняемая вызовом. В данном случае телом является `c[i] = a[i] + scalar * b[i];`;
- *аргументов* – аргумент `(int i)`, используемый в последующем вызове лямбда-выражения;
- *замыкания захватываемого контекста* – список переменных в теле функции, которые определены внешне, и способ их передачи в подпрограмму, который в листинге 10.1 задается в `[&]`. Знак `&` указывает на то, что обращение к переменной осуществляется по ссылке, а знак `=` указывает на то, что ее следует копировать по значению. Одинарный `&` устанавливает, что по умолчанию обращение к переменным осуществляется по ссылке. Мы можем задать переменные полнее, конкретизировав захват контекста как `[&c, &a, &b, &scalar]`;
- *вызова* – цикл `for` в строках с 13 по 16 в листинге 10.1 вызывает лямбду над заданными значениями массива.

Лямбда-выражения формируют основу для более естественного генерирования кода для GPU-процессоров на новых C++-подобных языках, таких как SYCL, Kokkos и Raja. Мы кратко рассмотрим SYCL в главе 12 как язык C++ более высокого уровня (первоначально построенный поверх OpenCL). Два языка, Kokkos, вышедший из Национальных лабораторий Сандии (SNL) и Raja, вышедший из Национальной лаборатории Лоуренса Ливермора (LLNL), являются языками более высокого уровня, разработанными с целью упрощения написания переносимых научных приложений для широкого спектра современного компьютерного оборудования. Мы также представим Kokkos и Raja в главе 12.

10.2.2 Поточные индексы: соотнесение локальной плитки с глобальным миром

Ключ к пониманию того, как вычислительное ядро формирует свою локальную операцию, состоит в том, что в качестве продукта декомпозиции данных мы предоставляем каждой рабочей группе некоторую информацию о том, где она находится в локальном и глобальном доменах. В OpenCL вы можете получить следующую ниже информацию.

- *Размерность* – получает из вызова вычислительного ядра число размерностей, одну размерность, две размерности либо три размерности, для этого ядра.
- *Глобальная информация* – глобальный индекс в каждой размерности, который соответствует локальной единице работы, или глобальный размер в каждой размерности, т. е. размер глобального вычислительного домена в каждой размерности.
- *Локальная (плиточная) информация* – локальный размер в каждой размерности, который соответствует размеру плитки в этой размерности, или локальный индекс в каждой размерности, который соответствует индексу плитки в этой размерности.
- *Групповая информация* – число групп в каждой размерности, соответствующее числу групп в этой размерности, или индекс группы в каждой размерности, соответствующий индексу группы в этой размерности.

Аналогичная информация доступна в CUDA, но глобальный индекс должен быть рассчитан из индекса локального потока плюс информации о блоке (плитке):

```
gid = blockIdx.x *blockDim.x + threadIdx.x;
```

На рис. 10.9 представлена индексация рабочей группы (блока или плитки) для OpenCL и CUDA. Сначала вызывается функция OpenCL, за которой следует переменная, определенная в CUDA. Вся эта поддержка индексирования автоматически выполняется за вас с помощью декомпозиции данных для GPU, что значительно упрощает обработку отображения из глобального пространства в плитку.

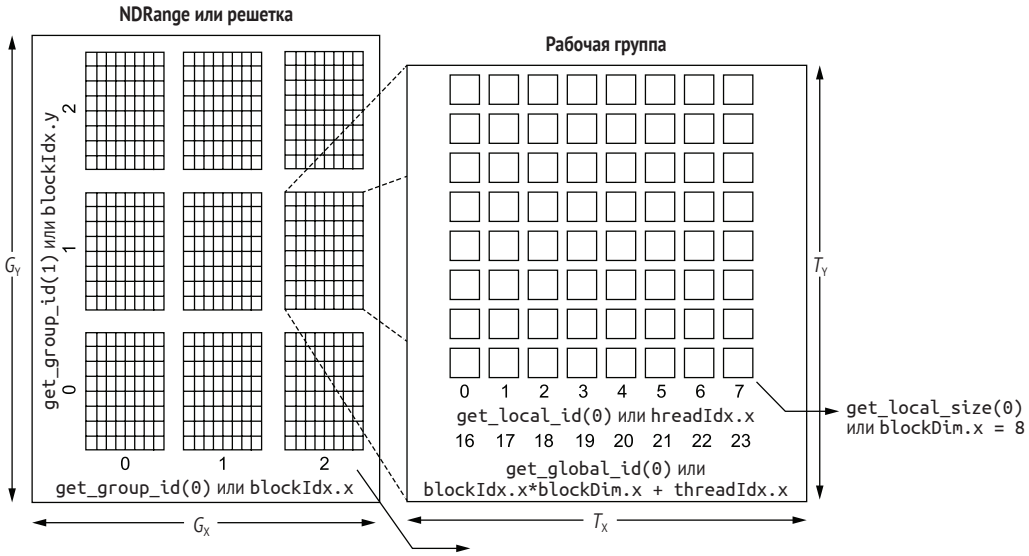


Рис. 10.9 Отображение индекса отдельного элемента работы в глобальное индексное пространство. Сначала задается вызов OpenCL, за которым следует переменная, определенная в CUDA

10.2.3 Индексные множества

Размер индексов для каждой рабочей группы должен быть одинаковым. Это делается путем расширения глобального вычислительного домена до размера, кратного размеру локальной рабочей группы. Указанное расширение можно сделать с помощью небольшой целочисленной арифметики, в результате получая одну дополнительную рабочую группу и дополненный глобальный размер работы. В следующем ниже примере показан подход, в котором используются базовые целочисленные операции, а затем второй пример с внутренней для C функцией ceil.

```
global_work_sizex = ((global_sizex + local_work_sizex - 1)/
                          local_work_sizex) * local_work_sizex
```

Пример: расчет размеров рабочих групп

В приведенном ниже фргаменте кода используются аргументы для вызова вычислительного ядра, чтобы получить унифицированные размеры рабочих групп.

```
int global_size_x = 1000;
int local_work_size_x = 128;
int global_work_size_x = ((global_size_x + local_work_size_x - 1)/
                            local_work_size_x) * local_work_size_x = 1024;
int number_work_groups_x = (global_size_x + local_work_size_x - 1)/
local_work_size_x = 8
// или, в качестве альтернативы
int global_work_size_x = ceil(global_size_x/local_work_size_x) *
                               local_work_size_x = 1024;
int number_work_groups_x = ceil(global_size_x/local_work_size_x) = 8;
```

Во избежание чтений за пределами массива мы должны проверять глобальный индекс в каждом вычислительном ядре и пропускать чтение, если оно находится после конца массива, с помощью чего-то наподобие вот этого:

```
if (gid_x > global_size_x) return;
```

ПРИМЕЧАНИЕ В вычислительных ядрах GPU важно избегать чтений и записей за пределами границ, поскольку они приводят к случайным сбоям вычислительного ядра без сообщения об ошибке или результата на выходе.

10.2.4 Как обращаться к ресурсам памяти в вашей модели программирования GPU

Забота о памяти по-прежнему остается наиболее важной задачей, так как она влияет на ваш план программирования приложения. К счастью, на современных GPU памяти много. GPU-процессоры NVIDIA V100 и AMD

Radeon Instinct MI50 поддерживают 32 Гб RAM. По сравнению с хорошо обеспеченными узлами CPU HPC со 128 Гб памяти, вычислительный узел GPU с 4–6 GPU-процессорами имеет ту же память. На вычислительных узлах GPU столько же памяти, сколько и на CPU-процессорах. Следовательно, мы можем использовать ту же стратегию выделения памяти, что и для CPU, и нам не придется передавать данные туда-сюда из-за лимитированной памяти GPU.

Выделение памяти для GPU должно выполняться на CPU. Нередко память выделяется одновременно и для CPU, и для GPU, а затем данные передаются между ними. Но, если это возможно, вы должны выделять память только для GPU. За счет этого удается избегать дорогостоящей передачи памяти от CPU и обратно и высвобождает память на CPU. Алгоритмы, в которых используется динамическое выделение памяти, представляют проблему для GPU и должны быть конвертированы в алгоритм со статической памятью, размер которой известен заранее. Новейшие GPU хорошо справляются с *коагулированием* нерегулярных или перетасованных обращений к памяти в единую когерентную загрузку строки кеша, когда это возможно.

ОПРЕДЕЛЕНИЕ *Коагулированная загрузка памяти* – это комбинация отдельных загрузок памяти из групп потоков в единую загрузку строки кеша.

Коагулирование (т. е. объединение в единое целое) памяти на GPU выполняется на аппаратном уровне в контроллере памяти. Прирост производительности от этих коагулированных загрузок является значительным. Но также важно и то, что больше не нужны многочисленные оптимизации из предыдущих руководств по программированию GPU, что значительно снижает затраты на программирование GPU.

Вы можете получать некоторое дополнительное ускорение за счет использования локальной (совместной) памяти для данных, которые используются более одного раза. Эта оптимизация раньше была важна для производительности, но более качественный кеш на GPU-процессорах делает ускорение менее значительным. Существует пара стратегий использования локальной памяти в зависимости от способности предсказывать необходимый размер локальной памяти. На рис. 10.10 слева показан подход с регулярной решеткой и справа с нерегулярной решеткой для неструктурированной и адаптивной детализации вычислительной сетки. Регулярная решетка имеет четыре примыкающие плитки с накладывающимися друг на друга ореольными участками. Адаптивная детализация вычислительной сетки показывает только четыре ячейки; типичное приложение GPU загружает 128 либо 256 ячеек, а затем привносит необходимые соседние ячейки, лежащие по периферии.

Процессы в этих двух случаях таковы:

- *потокам требуется та же загрузка памяти, что и соседним потокам.* Хорошим примером тому является стенсильная операция, которую мы используем на протяжении всей книги. Потоку `i` нужны

значения `i-1` и `i+1`, а значит многочисленным потокам понадобятся одни и те же значения. В этой ситуации самый лучший подход состоит в том, чтобы выполнять кооперативные загрузки памяти. Копирование значений памяти из глобальной памяти в локальную (совместную) память приводит к значительному ускорению;

- *нерегулярная решетка имеет непредсказуемое число соседей, что затрудняет загрузку в локальную память.* Уладить эту ситуацию можно путем копирования части вычисляемой сетки в локальную память. Затем загрузив соседние данные в регистры по каждому потоку.

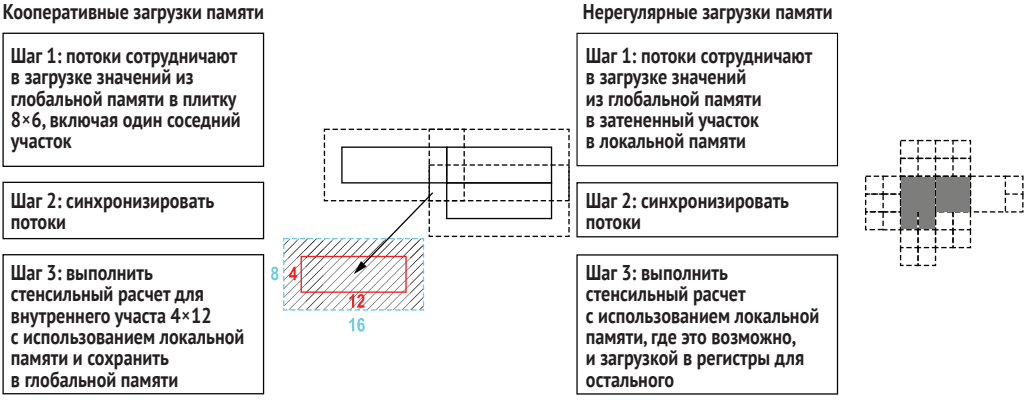


Рис. 10.10 Для стенсилей на регулярных решетках загружать все данные в локальную память, а затем использовать локальную память для вычислений. Внутренний сплошной прямоугольник – это вычислительная плитка. Внешний пунктирный прямоугольник заключает в себе соседние данные, необходимые для расчета. Кооперативные загрузки можно использовать для загрузки данных во внешнем прямоугольнике в локальную память по каждой рабочей группе. Поскольку нерегулярные решетки имеют непредсказуемый размер, следует загружать в локальную память только вычисляемый участок и в отношении остального использовать регистры для каждого потока

Это не единственные подходы к использованию ресурсов памяти на GPU. Важно продумать трудности, связанные с лимитированными ресурсами и потенциальными выгодами в производительности для вашего конкретного приложения.

10.3 Оптимизирование использования ресурсов GPU

Ключом к хорошему программированию GPU является управление лимитированными ресурсами, имеющимися в распоряжении для исполнения вычислительных ядер. Давайте рассмотрим несколько наиболее важных ресурсных ограничений в табл. 10.4. Превышение доступных ресурсов может приводить к значительному снижению производитель-

ности. Вычислительная способность NVIDIA, равная 7.0, предназначена для чипа V100. В более новом чипе Ampere A100 используется вычислительная способность 8.0 с почти одинаковыми ресурсными лимитами.

Таблица 10.4 Несколько ресурсных лимитов для текущих GPU

Ресурсный лимит	Вычислительная способность 7.0 в NVIDIA	AMD Vega 20 (MI50)
Максимум потоков на рабочую группу	1024	256
Максимум потоков на вычислительный модуль	2048	
Максимум рабочих групп на вычислительный модуль	32	16
Локальная память на вычислительный модуль	96 Кб	64 Кб
Размер регистрового файла на вычислительный модуль	64K	256 Кб-й вектор
Максимум 32-битовых регистров на поток	255	

Наиболее важным элементом управления, доступным программисту GPU, является размер рабочей группы. На первый взгляд может показаться, что было бы желательно использовать максимум потоков на рабочую группу. Но в случае с вычислительными ядрами их сложность, по сравнению с графическими ядрами, означает, что существует большая потребность в вычислительных ресурсах. В разговорной речи это принято называть *давлением на память* или *давлением на регистры*. Сокращение размера рабочей группы дает каждой рабочей группе больше ресурсов для работы. Оно также дает больше рабочих групп для переключения контекста, которое мы обсуждали в разделе 10.1.1. Ключом к достижению хорошей производительности GPU является отыскание правильного баланса между размером рабочей группы и ресурсами.

ОПРЕДЕЛЕНИЕ *Давление на память* – это влияние потребностей вычислительного ядра в ресурсах на производительность вычислительных ядер GPU. *Давление на регистры* – это аналогичный термин, относящийся к потребностям в регистрах в вычислительном ядре.

Полный анализ требований к ресурсам конкретного вычислительного ядра и ресурсов, имеющихся на GPU, требует тщательного анализа. Мы приведем примеры нескольких типов такого углубленного анализа. В следующих двух разделах мы рассмотрим:

- число регистров, используемых вычислительным ядром;
- степень загруженности мультипроцессоров, именуемой занятостью.

10.3.1 Сколько регистров используется в моем вычислительном ядре?

Вы можете узнать число регистров, используемых в вашем коде, добавив флаг `-Xptxas="-v"` в компиляционную команду `nvcc`. В целях получения аналогичного результата в OpenCL для GPU-процессоров NVIDIA используйте флаг `-cl-nv-verbose` для компиляционной строки OpenCL.

Пример: получение данных об использовании регистра для вашего вычислительного ядра на GPU NVIDIA

Сначала мы собираем BabelStream с дополнительными компиляторными флагами:

```
git clone git@github.com:UoB-HPC/BabelStream.git
cd BabelStream
export EXTRA_FLAGS='-Xptxas="-v"'
make -f CUDA.make
```

Результат на выходе из компилятора NVIDIA показывает использование регистра для потоковой триады:

```
ptxas info    : Used 14 registers, 4096 bytes smem, 380 bytes cmem[0]
ptxas info    : Compiling entry function '_Z12triad_kernelIfEvPT_PKS0_S3_'
                for 'sm_70'
ptxas info    : Function properties for _Z12triad_kernelIfEvPT_PKS0_S3_
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
```

В этом простом вычислительном ядре мы используем 14 регистров из 255, доступных на GPU NVIDIA.

10.3.2 Занятость: предоставление большего объема работы для планирования рабочей группы

Мы обсудили важность задержки и переключения контекста для хорошей производительности на GPU. Преимущество рабочих групп «правильного размера» заключается в том, что большее число рабочих групп может работать одновременно. Для GPU это важно, потому что, когда продвижение рабочей группы пробуксовывает из-за задержки памяти, ему нужны другие рабочие группы, которые он сможет исполнять, чтобы скрыть задержку. В целях задания надлежащего размера рабочей группы нам нужна какая-то мера. На GPU-процессорах для анализа рабочих групп используется мера под названием «занятость». *Занятость* – это показатель степени загруженности вычислительных модулей во время расчета. Эта мера усложнена, поскольку она зависит от большого числа факторов, таких как требуемая память и используемые регистры. Точное определение занятости таково:

Занятость = Число активных потоков / Максимальное число потоков на вычислительный модуль.

Поскольку число потоков в подгруппе является фиксированным, эквивалентное определение основано на подгруппах, также именуемых волновыми фронтами или варпами:

Занятость = Число активных подгрупп / Максимальное число подгрупп на вычислительный модуль.

Число активных подгрупп или потоков определяется ресурсом рабочей группы или потока, который исчерпывается первым. Нередко это число представлено числом необходимых рабочей группе регистров или локальной памятью, препятствующей запуску другой рабочей группы. Для этого нам нужен инструмент наподобие Калькулятора занятости CUDA (представлен в следующем ниже примере). В руководствах по программированию NVIDIA большое внимание уделяется максимальной занятости. Являясь важной мерой самой по себе, для переключения между рабочими группами просто должно быть достаточное их число, чтобы скрывать задержки и пробуксовки.

Пример: калькулятор занятости CUDA

1. Скачать электронную таблицу калькулятора занятости CUDA с https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html.
2. Ввести число регистров, полученное на выходе из компилятора NVCC (раздел 10.3.1), и размер рабочей группы (1024).

На следующем ниже рисунке показаны результаты для Калькулятора занятости. В электронной таблице также есть графики для варьирующихся размеров блоков, числа регистров и использования локальной памяти (не показано).

CUDA occupancy calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select compute capability (click):	**7.0**	(Help)
1.b) Select shared memory size config (bytes)	**32768**	

2.) Enter your resource usage:		
Threads per block	**1024**	(Help)
Registers per thread	**14**	
Shared memory per block (bytes)	**4096**	

(Don't edit anything below this line)

3.) GPU occupancy data is displayed here and in the graphs:		
Active threads per multiprocessor	**2048**	(Help)
Active warps per multiprocessor	**64**	
Active thread blocks per multiprocessor	**2**	
Occupancy of each multiprocessor	**100%**	

Physical limits for GPU compute capability:	**7.0**
Threads per warp	32
Max warps per multiprocessor	64
Max thread blocks per multiprocessor	32
Max threads per multiprocessor	2048
Maximum thread block size	1024
Registers per multiprocessor	65536
Max registers per thread block	65536
Max registers per thread	255
Shared memory per multiprocessor (bytes)	32768
Max shared memory per block	32768
Register allocation unit size	256
Register allocation granularity	warp
Shared memory allocation unit size	256
Warp allocation granularity	4

Allocated Resources	Per block	Limit per SM	= Allocatable Blocks per SM
Warps (Threads per block/Threads per warp)	32	64	2
Registers (Warp limit per SM due to per-warp reg count)	32	128	4
Shared memory (bytes)	4096	32768	8

Note: SM is an abbreviation for (streaming) multiprocessor

Maximum thread blocks per multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by max warps or max blocks per multiprocessor	2	32	64
Limited by registers per multiprocessor	4		
Limited by shared memory per multiprocessor	6		

Note: Occupancy limiter is shown in orange

Physical max warps/SM = 64
Occupancy = 64/64 = 100%

Результат на выходе из Калькулятора занятости CUDA для потоковой триады, показывающий использование ресурсов для вычислительного ядра. Третий блок сверху показывает меры занятости

10.4 Редукционный шаблон требует синхронизации между рабочими группами

До сих пор вычислительные циклы, которые мы рассматривали над ячейками, частицами, точками и другими вычислительными элементами, обрабатывались посредством подхода, представленного на рис. 10.8, в котором мы вычленяли циклы `for` из вычислительного тела для создания вычислительного ядра GPU. Это преобразование выполняется быстро и легко и может применяться к огромному числу циклов в научном приложении. Но имеются другие ситуации, когда преобразование кода в GPU осуществить чрезвычайно трудно. Мы рассмотрим алгоритмы, которые требуют более изощренного подхода. Возьмем, к примеру, одну строку кода на языке Fortran, в которой используется синтаксис массива:

```
xmax = sum(x(:))
```

На Fortran это выглядит довольно просто, но на GPU все намного усложняется. Источник трудностей проистекает из того, что мы не можем выполнять кооперативную работу или сравнения между рабочими группами. Этого можно добиться, только если выйти из ядра. Рисунок 10.11 иллюстрирует общую стратегию, которая улаживает эту ситуацию.

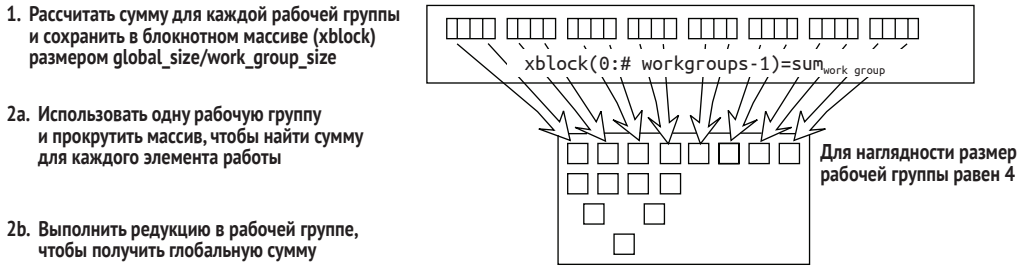


Рис. 10.11 Редукционная схема на GPU требует двух вычислительных ядер для синхронизирования многочисленных рабочих групп. Мы выходим из первого ядра, представленного прямоугольником, а затем запускаем еще одно ядро размером с одну рабочую группу, чтобы обеспечить кооперацию потоков для заключительного прохода

Для удобства иллюстрации на рис. 10.11 показан массив длиной 32 элемента. Типичный массив для этого метода будет состоять из сотен тысяч или даже миллионов элементов, так что он намного превышает размер рабочей группы. На первом шаге мы находим сумму каждой рабочей группы и сохраняем ее в блокнотном массиве длиной, равной числу рабочих групп или блоков. Первый проход редуцирует размер массива на размер нашей рабочей группы, который может составлять 512 или 1024. На этом шаге мы не можем обмениваться данными между рабочими группами, поэтому выходим из вычислительного ядра и запускаем новое ядро только с одной рабочей группой. Оставшиеся данные могут превышать размер рабочей группы 512 или 1024, поэтому мы прокручи-

ваем блокнотный массив данных, суммируя значения в каждом элементе работы. Мы можем обмениваться данными между элементами работы в рабочей группе, поэтому можем выполнить их редукцию к одному глобальному значению, суммируя по ходу дела.

Сложно! Исходный код для выполнения этой операции на GPU занимает десятки строк и два вычислительных ядра для выполнения той же операции, которую можно выполнить в одной строке кода на CPU. Мы увидим больше фактического исходного кода для редукции в главе 12, когда обратимся к программированию на CUDA и OpenCL. Получаемая на GPU производительность выше, чем на CPU, но для этого требуется много работы по программированию. И мы начнем понимать, что одной из характеристик GPU-процессоров является трудная выполнимость синхронизации и сравнений.

10.5 Асинхронные вычисления посредством очередей (потоков операций)

Мы увидим, как можно использовать GPU полнее путем наложения передачи данных и вычислений друг на друга. Две передачи данных могут происходить одновременно с вычислением на GPU.

Базовый характер работы на GPU-процессорах асинхронен. Работа ставится в очередь на GPU и, как правило, выполняется только при запросе результата или синхронизации. На рис. 10.12 показан типичный набор команд, отправляемых на GPU для вычисления.

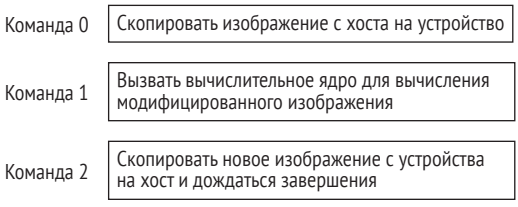


Рис. 10.12 Работа, запланированная на GPU в используемой по умолчанию очереди, завершается только тогда, когда запрашивается ожидание завершения. Мы запланировали копирование графического изображения (картинки) на GPU. Затем запланировали математическую операцию на данных, чтобы его модифицировать. Мы также запланировали третью операцию по ее возвращению. Ни одна из этих операций не должна начинаться до тех пор, пока не потребуем подождать завершения

Мы также можем планировать работу в нескольких независимых и несинхронных очередях. Использование нескольких очередей, как показано на рис. 10.13, открывает возможности для наложенных друг на друга передачи данных и вычислений. Большинство языков GPU поддерживает ту или иную форму асинхронных рабочих очередей. В OpenCL команды помещаются в очередь, а в CUDA операции размещаются в потоках. Хотя создается потенциал для параллелизма, будет ли он происходить на самом деле, зависит от способностей оборудования и деталей кодирования.

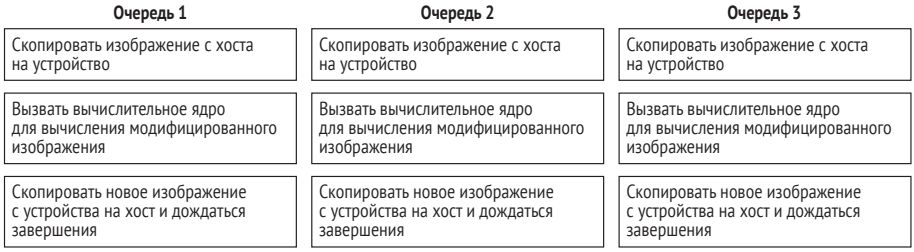


Рис. 10.13 Разбивка работы на этапы для трех изображений в параллельных очередях

Если у нас есть GPU, способный одновременно выполнять приведенные ниже операции:

- копирование данных с хоста на устройство,
- вычисления ядра(ядер),
- копирование данных с устройства на хост,

то работа, выполняемая в трех отдельных очередях на рис. 10.13, может накладывать вычисления и обмен данными друг на друга, как показано на рис. 10.14.

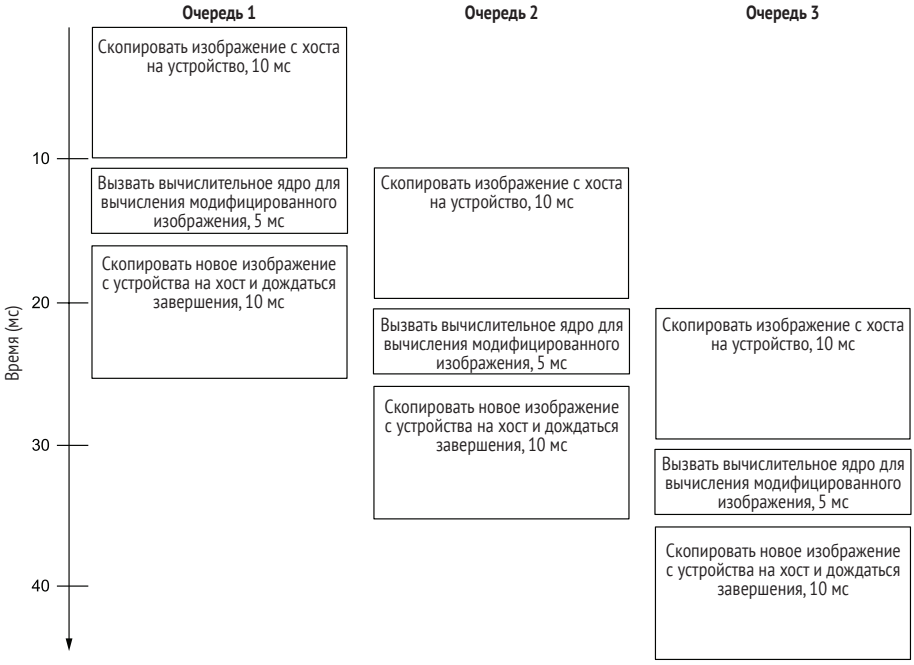


Рис. 10.14 Наложение вычислений и передачи данных сокращает время для трех изображений с 75 до 45 мс. Это возможно вследствие того, что GPU может выполнять одновременно вычисления, передачу данных с хоста на устройство и еще одну передачу с устройства на хост

10.6 Разработка плана параллелизации приложения для GPU-процессоров

Теперь перейдем к использованию нашего понимания модели программирования GPU с целью разработки стратегии параллелизации нашего приложения. Мы воспользуемся парой примеров приложений, чтобы продемонстрировать указанный процесс.

10.6.1 Случай 1: трехмерная атмосферная симуляция

Ваше приложение представляет собой атмосферную симуляцию размером от 1024×1024×1024 до 8192×8192×8192, в которой x выступает в качестве вертикальной размерности, y – в качестве горизонтальной и z – в качестве глубины. Давайте взглянем на опции, которые вы могли бы рассмотреть.

- *Опция 1: распределять данные в одномерном формате по измерению z (глубины).* В случае GPU-процессоров нам для эффективного параллелизма нужны десятки тысяч рабочих групп. Из спецификации GPU (табл. 9.3) следует, что у нас есть 60–80 вычислительных модулей из 32 арифметических модулей двойной точности для примерно 2000 одновременных арифметических маршрутов. В дополнение к этому нам нужно больше рабочих групп для сокрытия задержек посредством переключения контекста. Распределение данных по размерности z позволяет получать от 1024 до 8192 рабочих групп, что является низким показателем параллелизма GPU.
  Давайте рассмотрим ресурсы, необходимые для каждой рабочей группы. Минимальные размерности будут иметь плоскость 1024×1024 плюс любые необходимые соседние данные в призрачных ячейках. Мы допустим наличие одной призрачной ячейки в обоих направлениях. Поэтому нам потребуется 1024 × 1024 × 3 × 8 байт, или 24 МиБ, локальных данных. Если посмотреть на табл.10.4, то GPU-процессоры содержат 64–96 КиБ локальных данных, поэтому мы не сможем предзагружать данные в локальную память для их более быстрой обработки.
- *Опция 2: распределять данные в двухмерных вертикальных столбцах по размерностям y и z.* Распределение по двум размерностям дало бы нам более миллиона потенциальных рабочих групп, вследствие чего у нас будет достаточно независимых рабочих групп для GPU. По каждой рабочей группе у нас будет от 1024 до 8192 ячеек. У нас есть собственная ячейка плюс 4 соседа для 1024 × 5 × 8 = 40 КиБ минимальной требуемой локальной памяти. Для более крупных задач и с более чем одной переменной на ячейку нам не хватило бы локальной памяти.
- *Опция 3: распределять данные в трехмерных кубах по размерностям x, y и z.* Используя шаблон из табл. 10.3, давайте попробуем исполь-

зовать для каждой рабочей группы 4×4×8-ячеечную плитку. С соседями это 6 × 6 × 10 × 8 байт для минимального объема требуемой локальной памяти 2.8 КиБ. У нас могло бы быть больше переменных на ячейку, и мы могли бы поэкспериментировать с увеличением размера плитки.

Суммарные требования к памяти для 1024×1024×1024-ячеечной плитки × 8 байт составляют 8 ГиБ. Мы имеем крупную задачу. GPU-процессоры имеют до 32 ГиБ оперативной памяти, вследствие чего эта задача, возможно, поместится на одном GPU. Задачи с более крупным размером потенциально потребовали бы до 512 GPU-процессоров. Поэтому мы также должны планировать параллелизм распределенной памяти с использованием MPI.

Давайте сравним это с CPU, где указанные дизайнерские решения будут иметь разные результаты. Возможно, нам придется распределять работу по 44 процессам, каждый из которых имеет меньше ресурсных ограничений. В то время как трехмерный подход мог бы работать, одномерный и двухмерный подходы тоже будут осуществимы. Теперь давайте сравним это с неструктурированной вычислительной сеткой, где все данные содержатся в одномерных массивах.

10.6.2 Случай 2: применение неструктурированной вычислительной сетки

В этом случае ваше приложение представляет собой трехмерную неструктурированную вычислительную сетку, в которой используются четырехгранные или многогранные ячейки в диапазоне от 1 до 10 млн ячеек. Но данные представляют собой одномерный список многоугольников с такими данными, как *x*, *y* и *z*, которые содержат пространственное местоположение. В этом случае есть только одна опция: одномерное распределение данных.

Поскольку данные не структурированы и содержатся в одномерных массивах, варианты выбора проще. Мы распределяем данные в одной размерности с размером плитки 128. Это дает нам от 8000 до 80 000 рабочих групп, предоставляя GPU-процессору достаточно работы для переключения и сокрытия задержки. Требования к памяти составляют 128 × 8 байт с двойной точностью = 1 Кб, что позволяет размещать несколько значений данных в расчете на ячейку.

Нам также понадобится место для некоторого целочисленного отображения и соседних массивов, чтобы обеспечить связность между ячейками. Соседние данные загружаются в регистры для каждого потока, вследствие чего нам не нужно беспокоиться о влиянии на локальную память и, возможно, превышении лимита памяти. Для сетки наибольшего размера в 10 млн ячеек требуется 80 Мб плюс место для массивов граней, соседей и отображений. Эти массивы связности могут значительно увеличивать использование памяти, но на одном GPU должно быть достаточно памяти для выполнения вычислений даже на сетках самого крупного размера.

В целях достижения наилучших результатов нам нужно будет обеспечивать некоторую локальность для неструктурированных данных с помощью библиотеки разбиения данных или с помощью кривой заполненности пространства, которая удерживает пространственно близкие ячейки в массиве близко друг к другу.

10.7 Материалы для дальнейшего изучения

Хотя базовые контуры модели программирования GPU стабилизировались, по-прежнему еще происходит много изменений. В частности, ресурсы, доступные для вычислительных ядер, постепенно увеличиваются по мере того, как целевые приложения расширяются с двухмерной до трехмерной графики и физических симуляций в целях получения более реалистичных игр. Такие рынки, как научные вычисления и машинное обучение, также приобретают все более важное значение. Для обоих этих рынков было разработано специальное оборудование GPU: двойная точность для научных вычислений и тензорные ядра для машинного обучения.

В нашей презентации мы в основном обсуждали дискретные GPU. Но есть также интегрированные GPU, как впервые обсуждалось в разделе 9.1.1. Модуль ускоренной обработки (APU) является продуктом компании AMD. Как APU AMD, так и интегрированные GPU-процессоры Intel предлагают некоторые преимущества в снижении затрат на передачу памяти, поскольку они больше не находятся на шине PCI. Это компенсируется уменьшением площади кремния для GPU-процессоров и более низкой огибающей мощности. Тем не менее эта способность с тех пор, как она появилась, была недооценена по достоинству. Первостепенное внимание при разработке уделялось большим дискретным GPU, входящим в топовые системы HPC. Но одни и те же языки программирования и инструменты GPU одинаково хорошо работают и с интегрированными GPU. Критическим ограничением при разработке новых ускоренных приложений является широкое распространение знаний о том, как программировать и использовать эти устройства.

Другие устройства массового рынка, такие как планшеты Android и мобильные телефоны, имеют программируемые GPU с языком OpenCL. Некоторые ресурсы для них таковы.

- Скачайте тестовые приложения OpenCL-Z и OpenCL-X из Google Play, чтобы узнать, не поддерживает ли ваше устройство OpenCL. Драйверы также могут иметься у поставщиков оборудования.
- Веб-сайт Compubench (https://compubench.com/result.jsp) имеет результаты производительности для некоторых мобильных устройств, использующих OpenCL или CUDA.
- У Intel есть хороший веб-сайт по программированию на OpenCL для Android по адресу https://software.intel.com/en-us/android/articles/opencl-basic-sample-for-android-os.

В последние годы аппаратное и программное обеспечение GPU получило добавленную поддержку других типов моделей программирования, таких как подходы, основанные на операционных задачах (см. рис.1.25) и графовые алгоритмы. Эти альтернативные модели программирования уже давно вызывали интерес со стороны параллельных программистов, но они испытывают трудности с эффективностью и масштабированием. Существуют важные приложения, такие как решатели на основе разреженных матриц, которые невозможно легко имплементировать без дальнейших достижений в этих областях. Но фундаментальный вопрос заключается в том, получится ли проявить достаточный объем параллелизма (обнаруженный на аппаратном уровне) для эффективного использования массивной параллельной архитектуры GPU-процессоров. Это покажет только время.

10.7.1 Дополнительное чтение

NVIDIA уже давно поддерживает исследования в области программирования GPU. Руководство по программированию на CUDA и передовой практике (доступны по адресу https://docs.nvidia.com/cuda) стоят того, чтобы их прочитать. Другие ресурсы таковы:

- серия GPU Gems (https://developer.nvidia.com/gpugems) – это более старый набор документации, который по-прежнему содержит много релевантных материалов;
- AMD также имеет ряд материалов по программированию GPU на своем веб-сайте GPUOpen по адресу

  https://gpuopen.com/compute-product/rocm/

  и на площадке ROCm

  https://rocm.github.io/documentation.html;
- AMD предоставляет одну из лучших таблиц для сравнения терминологии разных языков программирования GPU. Указанная таблица доступна по адресу https://rocm.github.io/languages.html.

Несмотря на то что компания Intel® занимает около 65 % рынка GPU (в основном интегрированных GPU), она только начинает быть серьезным игроком в вычислениях на GPU. Компания анонсировала новую дискретную графическую плату и станет поставщиком GPU для системы Aurora в Аргоннскую национальную лабораторию (будет поставлена в 2022 году). Система Aurora является первой когда-либо выпущенной экзомасштабной системой и имеет производительность в 6 раз выше, чем у нынешней топовой системы в мире. Указанный GPU основан на архитектуре Intel® Iris® Xe с кодовым названием Ponte Vec-chio. С большой помпой Intel выпустила свою инициативу по программированию oneAPI. Инструментарий oneAPI поставляется с драйвером GPU Intel, компиляторами и инструментами. Перейдите по ссылке https://software.intel.com/oneapi для получения дополнительной информации и скачиваемых файлов.

10.7.2 Упражнения

1. У вас есть приложение для классифицирования фотоснимков, которое будет занимать 5 мс для передачи каждого файла на GPU, 5 мс для обработки и 5 мс для возврата. На CPU обработка занимает 100 мс на фотоснимок. Нужно обработать миллион фотоснимков. У вас на CPU есть 16 обрабатывающих ядер. Будет ли система на базе GPU работать быстрее?
2. Время передачи у GPU в задаче 1 основано на шине PCI третьего поколения. Если вы сможете получить шину PCI Gen4, то как она изменит дизайн? А шина PCI Gen5? Для классифицирования фотоснимков вам не нужно возвращать модифицированный фотоснимок. Как это поменяет расчеты?
3. Какого размера трехмерное приложение вы могли бы выполнить для вашего дискретного GPU (или NVIDIA GeForce GTX 1060, если такового нет)? Допустим, что на ячейку приходится четыре переменных двойной точности и что есть лимит использования половины памяти GPU, чтобы иметь место для временных массивов. Как это изменит приложение, если вы используете одинарную точность?

Резюме

- Параллелизм на GPU должен быть в тысячах независимых элементов работы, потому что существуют тысячи независимых арифметических модулей. CPU требует параллелизм только в десятках независимых элементов работы для распределения работы между обрабатывающими ядрами. Таким образом, для GPU в наших приложениях важно проявлять больший параллелизм, чтобы обрабатывающие модули были заняты.
- Разные поставщики GPU-процессоров имеют похожие модели программирования, обусловленные потребностями графики с высокой частотой кадров. По этой причине может быть разработан общий подход, применимый к самым разным GPU.
- Модель программирования на GPU особенно хорошо подходит для параллелизма данных с крупными наборами вычислительных данных, но бывает трудно осуществимой для некоторых операционных задач с большой координацией, таких как редукция. Как следствие, многие высокопараллельные циклы легко переносятся, но есть и такие, которые требуют больших усилий.
- Разделение вычислительного цикла на тело цикла и управление циклом, или индексное множество, является для программирования GPU мощной концепцией. Тело цикла становится вычислительным ядром GPU, а CPU выполняет выделение памяти и вызывает ядро.
- Асинхронные рабочие очереди могут накладывать обмен данными и вычисления друг на друга. Это помогает повышать коэффициент полезного использования GPU.

11 Программирование GPU на основе директив

Эта глава охватывает следующие ниже темы:

- выбор наилучшего директиво-ориентированного языка для вашего GPU;
- использование директив или прагм для переноса кода на GPU-процессоры или другие ускорители;
- оптимизирование производительности вашего приложения на GPU.

За формирование стандартов директиво-ориентированных языков для GPU-процессоров разразилась настоящая схватка. Выпущенный в 1997 году выдающийся директиво-ориентированный язык, OpenMP, был естественным кандидатом на то, чтобы рассматриваться как более простой способ программирования GPU-процессоров. В то время OpenMP играл в догонялки и в основном был сосредоточен на новых способностях CPU. В целях решения задачи доступности GPU в 2011 году небольшая группа разработчиков компиляторов (Cray, PGI и CAPS) вместе с NVIDIA в качестве поставщика GPU присоединились к выпуску стандарта OpenACC, обеспечивающего более простой путь к программированию GPU. Подобно тому, что вы видели в главе 7 по OpenMP, в OpenACC тоже используются прагмы. В данном случае прагмы OpenACC направляют компилятор на создание кода GPU. Пару лет спустя Совет по пересмотру архитектуры OpenMP (Architecture Review Board, ARB) добавил в стандарт OpenMP свою собственную поддержку прагм для GPU-процессоров.

Мы пройдемся по некоторым базовым примерам на языках OpenACC и OpenMP, чтобы дать вам представление о том, как они работают. Мы предлагаем вам опробовать примеры в вашей целевой системе, чтобы узнать, какие компиляторы есть и их текущее состояние.

ПРИМЕЧАНИЕ Как всегда, мы рекомендуем вам сверяться с примерами этой главы, расположенными по адресу https://github.com/EssentialsofParallelComputing/Chapter11.

Многие программисты находятся «в нерешительности» насчет того, какой директиво-ориентированный язык – OpenACC либо OpenMP – они должны использовать. Нередко выбор становится ясен только после того, как вы узнаете, что конкретно имеется в вашей опорной системе. Имейте в виду, что самый большой барьер, который предстоит преодолеть, – это просто начать. Если позже вы решите переключиться на языки GPU, то предварительная работа все равно окажется полезной, поскольку стержневые концепции выходят за рамки языка. Мы надеемся, что, увидев, насколько мало усилий требуется для генерирования кода GPU с использованием прагм и директив, вы попробуете это на нескольких своих собственных фрагментах кода. И, приложив совсем немного усилий, вы даже, возможно, получите небольшое ускорение.

История OpenMP и OpenACC

Разработка стандартов OpenMP и OpenACC представляет собой главным образом дружеское соперничество; некоторые члены комитета OpenACC также входят в состав комитета OpenMP. Их имплементации по-прежнему появляются, во главе с усилиями Национальной лаборатории Лоуренса Ливермора, IBM и GCC. Предпринимались попытки объединить эти два подхода, но они продолжают сосуществовать и, вероятно, будут сосуществовать в обозримом будущем.

Язык OpenMP быстро набирает обороты и считается более надежным долгосрочным маршрутом, но на данный момент язык OpenACC имеет более зрелые имплементации и более широкую поддержку со стороны разработчиков. На следующем ниже рисунке показана полная история выпусков этих стандартов. Обратите внимание, что версия 4.0 стандарта OpenMP является первой, которая поддерживает GPU и ускорители.

OpenACC
- Версия 1.0 Ноябрь 2011
- Версия 2.0 Июнь 2013
- Версия 2.5 Октябрь 2015
- Версия 2.6 Ноябрь 2017
- Версия 2.7 Ноябрь 2018

OpenMP с поддержкой GPU
- Версия 4.0 Июль 2013
- Версия 4.5 Ноябрь 2015
- Версия 5.0 Ноябрь 2018
- Версия 5.1 Ноябрь 2020

Даты выхода прагма-ориентированных языков GPU

11.1 Процесс применения директив и прагм для имплементации на основе GPU

Аннотации на основе директив или прагм для приложений на языках C, C++ или Fortran обеспечивают один из наиболее привлекательных путей доступа к вычислительной мощи GPU-процессоров. Подобно описанной в главе 7 модели потокообразования OpenMP можно добавить всего несколько строк кода в приложение, и компилятор будет генерировать исходный код, который может выполняться на GPU либо CPU. Как впервые описано в главах 6 и 7, прагмы – это препроцессорные инструкции на языках C и C++, которые дают компилятору специальные команды. Они принимают форму:

```
#pragma acc <директива> [выражение]
#pragma omp <директива> [выражение]
```

Соответствующие способности для кода на Fortran обеспечиваются за счет директивы в форме специальных комментариев. Директивы начинаются с символа комментария, за которым следует ключевое слово acc или omp, чтобы идентифицировать их как директивы соответственно для OpenACC и OpenMP.

```
!$acc <директива> [выражение]
!$omp <директива> [выражение]
```

Для имплементирования OpenACC и OpenMP в приложениях используются те же самые общие шаги. Указанные шаги показаны на рис. 11.1, и мы подробно рассмотрим их в последующих разделах.

Шаги	CPU	GPU
Изначально	a[1000] for or do loop for or do loop free a	
1. Выгрузить работу на GPU	a[1000] **#прагма работы** → for or do loop (128 потоков) **#прагма работы** → for or do loop (128 потоков) free a	
2. Управлять перемещением данных на GPU	**#прагма данных{** #прагма работы #прагма работы **}**	a[1000] for or do loop (128 потоков) for or do loop (128 потоков) free a

3. Оптимизировать вычислительные ядра GPU	#прагма данных{
	#прагма работы
	#прагма работы
	}
		a[1000]
		for or do loop (**256 потоков**)
		for or do loop (**64 потоков**)
		free a

Рис. 11.1 Шаги по имплементированию переноса работы на GPU с помощью прагма-ориентированных языков. Выгрузка работы на GPU приводит к передаче данных, которая замедляет работу приложения, пока не будет редуцировано перемещение данных

Три шага, которые мы будем использовать для конвертирования исходного кода для выполнения на GPU с помощью OpenACC либо OpenMP, резюмируются следующим образом:

1. перенести вычислительно трудоемкую работу на GPU. Это приводит к передаче данных между CPU и GPU, что замедляет работу кода, но сначала необходимо переместить работу;
2. редуцировать перемещение данных между CPU и GPU. Перемещать выделения памяти на GPU, если данные используются только там;
3. отрегулировать размер рабочей группы, число рабочих групп и другие параметры вычислительного ядра для повышения производительности ядра.

В этом месте вы получите приложение, которое будет работать намного быстрее на GPU. Для повышения производительности возможны дальнейшие оптимизации, хотя они, как правило, более специфичны для каждого приложения.

11.2 OpenACC: самый простой способ выполнения на вашем GPU

Мы начнем с создания простого приложения, выполняющегося с помощью OpenACC. Это делается, чтобы показать базовые детали организации работы. Затем, приведя приложение в рабочее состояние, мы поработаем над его оптимизированием. Как и следовало ожидать, при прагма-ориентированном подходе малые усилия приносят большую отдачу. Но сначала вам нужно отработать первоначальное замедление кода. Не отчаивайтесь! Ситуация, когда вы сталкиваетесь с первоначальным замедлением на пути к более быстрым вычислениям на GPU, является совершенно нормальной.

Нередко самым сложным шагом является получение работающей цепочки инструментов компилятора OpenACC. Существует несколько надежных компиляторов OpenACC. Наиболее известные из имеющихся компиляторов перечислены ниже[1]:

- *PGI* – это коммерческий компилятор, но обратите внимание, что PGI имеет редакцию для некоммерческого использования (Community Edition), которую можно скачивать бесплатно;
- *GCC* – в версиях 7 и 8 имплементирована бо́льшая часть спецификации OpenACC 2.0a. В версии 9 имплементирована бо́льшая часть спецификации OpenACC 2.5. В разрабатываемой ветке OpenACC на GCC ведутся работы над OpenACC 2.6, предлагая дальнейшие усовершенствования и оптимизации;
- *Cray* – еще один коммерческий компилятор; он доступен только в системах Cray. Компания Cray объявила, что они больше не будут

[1] Один из изначальных компиляторов OpenACC, CAPS, был выведен из эксплуатации в 2016 году и больше не доступен.

поддерживать OpenACC в своем новом компиляторе C/C++ на основе LLVM, начиная с версии 9.0. «Классическая» версия компилятора, поддерживающая OpenACC, по-прежнему доступна.

В приводимых далее примерах мы будем использовать компилятор PGI (версии 19.7) и CUDA (версии 10.1). Компилятор PGI является наиболее зрелым вариантом из имеющихся компиляторов. Компилятор GCC является еще одним вариантом, но обязательно используйте новейшую версию. Компилятор Cray является отличным вариантом, если у вас есть доступ к их системе.

ПРИМЕЧАНИЕ Что делать, если у вас нет подходящего GPU? Вы по-прежнему можете попробовать примеры, выполняя исходный код на своем CPU с помощью вычислительных ядер, сгенерированных OpenACC. Производительность будет отличаться, но базовый код должен быть тем же самым.

Используя компилятор PGI, можно сначала получить информацию о своей системе посредством команды `pgaccelinfo`. Она также позволяет вам узнать о том, находится или нет ваша система и окружающая среда в рабочем состоянии. После выполнения указанной команды результат на выходе должен выглядеть примерно так, как показано на рис. 11.2.

```
CUDA Driver Version:           10010
NVRM version:                  NVIDIA UNIX x86_64 Kernel Module  418.87.00  Thu Aug  8 15:35:46 CDT 2019

Device Number:                 0
Device Name:                   Tesla V100-SXM2-16GB
Device Revision Number:        7.0
Global Memory Size:            16914055168
Number of Multiprocessors:     80
Concurrent Copy and Execution: Yes
Total Constant Memory:         65536
Total Shared Memory per Block: 49152
Registers per Block:           65536
Warp Size:                     32
Maximum Threads per Block:     1024
Maximum Block Dimensions:      1024, 1024, 64
Maximum Grid Dimensions:       2147483647 x 65535 x 65535
Maximum Memory Pitch:          2147483647B
Texture Alignment:             512B
Clock Rate:                    1530 MHz
Execution Timeout:             No
Integrated Device:             No
Can Map Host Memory:           Yes
Compute Mode:                  default
Concurrent Kernels:            Yes
ECC Enabled:                   Yes
Memory Clock Rate:             877 MHz
Memory Bus Width:              4096 bits
L2 Cache Size:                 6291456 bytes
Max Threads Per SMP:           2048
Async Engines:                 2
Unified Addressing:            Yes
Managed Memory:                Yes
Concurrent Managed Memory:     Yes
Preemption Supported:          Yes
Cooperative Launch:            Yes
  Multi-Device:                Yes
PGI Default Target:            -ta=tesla:cc70
```

Рис. 11.2 Результат на выходе из команды pgaccelinfo показывает тип GPU и его характеристики

11.2.1 Компилирование исходного кода OpenACC

В листинге 11.1 показано несколько выдержек из файлов makefile OpenACC. CMake предоставляет модуль FindOpenACC.cmake, вызываемый в строке 18 приведенного ниже листинга. Полный файл CMakeLists.txt включен в исходный код, прилагаемый к этой главе в каталоге OpenACC/StreamTriad по адресу https://github.com/EssentialsofParallelComputing/Chapter11. Мы установили несколько флагов для получения отклика от компилятора и для того, чтобы компилятор был менее консервативен в отношении потенциальной псевдонимизации. В подкаталоге содержатся файл CMake и простой файл makefile.

Листинг 11.1 Выдержки из файлов makefile OpenACC

OpenACC/StreamTriad/CMakeLists.txt

```
 8 if (NOT CMAKE_OPENACC_VERBOSE)
 9     set(CMAKE_OPENACC_VERBOSE true)
10 endif (NOT CMAKE_OPENACC_VERBOSE)
11
12 if (CMAKE_C_COMPILER_ID MATCHES "PGI")
13     set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -alias=ansi")
14 elseif (CMAKE_C_COMPILER_ID MATCHES "GNU")
15     set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -fstrict-aliasing")
16 endif (CMAKE_C_COMPILER_ID MATCHES "PGI")
17
18 find_package(OpenACC)
19
20 if (CMAKE_C_COMPILER_ID MATCHES "PGI")
21     set(OpenACC_C_VERBOSE "${OpenACC_C_VERBOSE} -Minfo=accel")
22 elseif (CMAKE_C_COMPILER_ID MATCHES "GNU")
23     set(OpenACC_C_VERBOSE
          "${OpenACC_C_VERBOSE} -fopt-info-optimized-omp")
24 endif (CMAKE_C_COMPILER_ID MATCHES "PGI")
25
26 if (CMAKE_OPENACC_VERBOSE)
27     set(OpenACC_C_FLAGS
        "${OpenACC_C_FLAGS} ${OpenACC_C_VERBOSE}")
28 endif (CMAKE_OPENACC_VERBOSE)
29
    < ... пропускаем первую цель ... >
33 # Добавляет сборочную цель триады stream_triad в файлы исходного кода
34 add_executable(StreamTriad_par1 StreamTriad_par1.c timer.c timer.h)
35 set_source_files_properties(StreamTriad_par1.c PROPERTIES COMPILE_FLAGS
     "${OpenACC_C_FLAGS}")
36 set_target_properties(StreamTriad_par1 PROPERTIES LINK_FLAGS
     "${OpenACC_C_FLAGS}")
```

Модуль CMake устанавливает компиляторные флаги для OpenACC

Добавляет отклик компилятора для акселераторных директив

Добавляет флаги OpenACC для компилирования и связывания исходного кода потоковой триады

Простые файлы makefile также можно использовать для сборки образцов исходного кода, скопировав или связав их с Makefile с помощью любой показанной ниже команды:

```
ln -s Makefile.simple.pgi Makefile
cp Makefile.simple.pgi Makefile
```

Из файлов makefile для компиляторов PGI и GCC мы показываем предлагаемые для OpenACC флаги:

Makefile.simple.pgi

```
 6 CFLAGS:= -g -O3 -c99 -alias=ansi -Mpreprocess -acc -Mcuda -Minfo=accel
 7
 8 %.o: %.c
 9   ${CC} ${CFLAGS} -c $^
10
11 StreamTriad: StreamTriad.o timer.o
12   ${CC} ${CFLAGS} $^ -o StreamTriad
```

Makefile.simple.gcc

```
 6 CFLAGS:= -g -O3 -std=gnu99 -fstrict-aliasing -fopenacc \
                              -fopt-info-optimized-omp
 7
 8 %.o: %.c
 9   ${CC} ${CFLAGS} -c $^
10
11 StreamTriad: StreamTriad.o timer.o
12   ${CC} ${CFLAGS} $^ -o StreamTriad
```

В случае PGI флаги для задействования компиляции OpenACC для GCC таковы: `-acc -Mcuda`. Флаг `Minfo=accel` сообщает компилятору предоставлять отклик на ускорительных директивах. Мы также включаем флаг `-alias=ansi`, сообщая компилятору о том, чтобы он меньше беспокоился о псевдонимизации указателей. Он помогает ему свободнее генерировать параллельные вычислительные ядра. По-прежнему неплохо включать атрибут `restrict` на аргументах в исходном коде, сообщая компилятору, что переменные не указывают на перекрывающиеся участки памяти. В оба файла makefile также включен флаг для указания стандарта C 1999, чтобы иметь возможность определять индексные переменные в цикле для организации более четкой области видимости. Флаг `-fopenacc` включает синтаксический разбор директив OpenACC для GCC. Флаг `-fopt-info-optimized-omp` сообщает компилятору предоставлять отклик в отношении генерации кода для ускорителя.

В случае компилятора Cray по умолчанию включен язык OpenACC. Вы можете применить компиляторную опцию `-hnoacc`, если вам нужно его отключить. И компиляторы OpenACC должны определять макрокоманду `_OPENACC`. Указанная макрокоманда особенно важна, поскольку OpenACC все еще находится в процессе имплементирования во многих компиляторах. Вы можете использовать ее для выяснения версии OpenACC, которую ваш компилятор поддерживает, и имплементировать условные компиляции для более новых функциональностей, сравнивая с компиляторной макрокомандой `_OPENACC == ггггмм`, где даты версий таковы:

- версия 1.0: 201111;
- версия 2.0: 201306;
- версия 2.5: 201510;
- версия 2.6: 201711;
- версия 2.7: 201811;
- версия 3.0: 201911.

11.2.2 Участки параллельных вычислений в OpenACC для ускорения вычислений

Существует два разных варианта объявления ускоренного блока кода для вычислений. Первый – это прагма `kernels`, которая дает компилятору свободу автоматической параллелизации блока кода. Этот блок кода может включать в себя более крупные разделы кода с несколькими циклами.

Второй – это прагма `parallel loop`, которая сообщает компилятору генерировать код для GPU или другого ускорителя. Мы пройдемся по примерам каждого подхода.

Использование прагмы kernels для получения от компилятора автоматической параллелизации

Прагма `kernels` позволяет компилятору автоматически распараллеливать блок кода. Она часто используется в первую очередь для получения отклика от компилятора о разделе кода. Мы охватим формальный синтаксис для прагмы `kernels`, включая ее необязательные выражения. Затем рассмотрим пример потоковой триады, который мы использовали во всех главах по программированию, и применим прагму `kernels`. Сначала мы перечислим спецификацию прагмы `kernels` из стандарта OpenACC 2.6:

```
#pragma acc kernels [ выражение данных | оптимизация ядра | асинхронное выражение |
                      условный блок ]
```

где

```
выражения данных      - [ copy | copyin | copyout | create | no_create |
                          present | deviceptr | attach | default(none|present) ]
оптимизация ядра      - [ num_gangs | num_workers | vector_length |
                          device_type | self ]
асинхронные выражения - [ async | wait ]
условный блок         - [ if ]
```

Мы обсудим выражения данных подробнее в разделе 11.2.3, хотя вы также можете использовать выражения данных в прагме `kernels`, если они применимы только к одному циклу. Мы рассмотрим оптимизации ядра в разделе 11.2.4. И кратко упомянем асинхронные и условное выражения в разделе 11.2.5.

Сначала начинаем с конкретизации места, где мы хотим распараллелить работу, добавив `#pragma acc kernels` вокруг целевых блоков кода.

Прагма `kernels` применяется к блоку кода, следующему за указанной директивой, или к коду в следующем ниже листинге, циклу `for`.

Листинг 11.2 Добавление прагмы kernels

OpenACC/StreamTriad/StreamTriad_kern1.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include "timer.h"
 4
 5 int main(int argc, char *argv[]){
 6
 7    int nsize = 20000000, ntimes=16;
 8    double* a = malloc(nsize * sizeof(double));
 9    double* b = malloc(nsize * sizeof(double));
10    double* c = malloc(nsize * sizeof(double));
11
12    struct timespec tstart;
13    // инициализация данных и массивов
14    double scalar = 3.0, time_sum = 0.0;
15 #pragma acc kernels
16    for (int i=0; i<nsize; i++) {
17       a[i] = 1.0;
18       b[i] = 2.0;
19    }
20
21    for (int k=0; k<ntimes; k++){
22       cpu_timer_start(&tstart);
23       // цикл потоковой триады
24 #pragma acc kernels
25       for (int i=0; i<nsize; i++){
26          c[i] = a[i] + scalar*b[i];
27       }
28       time_sum += cpu_timer_stop(tstart);
29    }
30
31    printf("Среднее время выполнения цикла потоковой триады составляет %lf msecs\n",
             time_sum/ntimes);
32
33    free(a);
34    free(b);
35    free(c);
36
37    return(0);
38 }
```

Вставляет прагму OpenACC kernels

Вставляет прагму OpenACC kernels

Блок кода для прагмы kernels

Следующий ниже результат показывает получаемый на выходе отклик компилятора PGI:

```
main:
     15, Generating implicit copyout(b[:20000000],a[:20000000])
         [if not already present]
```

```
    16, Loop is parallelizable
        Generating Tesla code
        16, #pragma acc loop gang, vector(128)
            /* blockIdx.x threadIdx.x */
    16, Complex loop carried dependence of a-> prevents parallelization
        Loop carried dependence of b-> prevents parallelization
    24, Generating implicit copyout(c[:20000000]) [if not already present]
        Generating implicit copyin(b[:20000000],a[:20000000])
        [if not already present]
    25, Complex loop carried dependence of a->,b-> prevents
        parallelization
        Loop carried dependence of c-> prevents parallelization
        Loop carried backward dependence of c-> prevents vectorization
        Accelerator serial kernel generated
        Generating Tesla code
    25, #pragma acc loop seq
    25, Complex loop carried dependence of b-> prevents parallelization
        Loop carried backward dependence of c-> prevents vectorization
```

В этом листинге неясно то, что OpenACC трактует каждый цикл `for` так, как если бы перед ним был автоматический цикл `#pragma acc`. Мы оставили на усмотрение компилятора решение о возможности параллелизации цикла. Выделенный жирным шрифтом результат показывает, что компилятор не считает, что это возможно. Компилятор говорит нам о том, что ему нужна помощь. Самое простое исправление состоит в добавлении атрибута `restrict` в строки 8–10 листинга 11.2.

```
 8    double* restrict a = malloc(nsize * sizeof(double));
 9    double* restrict b = malloc(nsize * sizeof(double));
10    double* restrict c = malloc(nsize * sizeof(double));
```

Наш второй вариант исправления в качестве помощи компилятору состоит в изменении директивы, чтобы сообщить компилятору, что все в порядке и можно создавать параллельный код GPU. Проблема заключается в применяемой по умолчанию директиве `loop` (`loop auto`), о которой мы упоминали ранее. Вот ее спецификация из стандарта OpenACC 2.6:

```
#pragma acc loop [ auto | independent | seq | collapse | gang | worker |
                   vector | tile | device_type | private | reduction ]
```

Мы коснемся многих этих выражений в последующих разделах. А пока сосредоточимся на первых трех: `auto`, `independent` и `seq`.

- Выражение `auto` позволяет компилятору выполнять анализ.
- Выражение `seq`, сокращение от `sequential`, поручает генерировать последовательную версию.
- Выражение `independent` подтверждает, что цикл может и должен быть распараллелен.

Замена выражения с `auto` на `independent` говорит компилятору распараллелить цикл:

```
15 #pragma acc kernels loop independent
   <пропуск неизменного кода>
24 #pragma acc kernels loop independent
```

Обратите внимание, что в этих директивах мы скомбинировали два конструкта. По желанию можно скомбинировать валидные индивидуальные выражения в единую директиву. Теперь результат на выходе показывает, что цикл распараллелен:

```
main:
     15, Generating implicit copyout(a[:20000000],b[:20000000])
         [if not already present]
     16, Loop is parallelizable
         Generating Tesla code
         16, #pragma acc loop gang, vector(128)
             /* blockIdx.x threadIdx.x */
     24, Generating implicit copyout(c[:20000000]) [if not already present]
         Generating implicit copyin(b[:20000000],a[:20000000])
         [if not already present]
     25, Loop is parallelizable
         Generating Tesla code
         25, #pragma acc loop gang, vector(128)
             /* blockIdx.x threadIdx.x */
```

В этой распечатке важно отметить отклик в отношении передачи данных (выделен жирным шрифтом). Мы обсудим вопрос о том, как реагировать на этот отклик, в разделе 11.2.3.

Опробывание прагмы parallel loop для большего контроля над параллелизацией

Далее охватим приемы использования прагмы параллельного цикла, `parallel loop`. Мы рекомендуем вам применять эту технику в своем приложении. Она более соответствует форме, используемой в других параллельных языках, таких как OpenMP. Она также обеспечивает более согласованную и переносимую производительность для всех компиляторов. Не на все компиляторы можно положиться в том, что касается адекватной работы по анализу, требуемому директивой `kernels`.

Прагма `parallel loop` на самом деле представляет собой две отдельные директивы. Первая – это директива `parallel`, которая открывает параллельный участок. Вторая – это прагма `loop`, которая распределяет работу между параллельными элементами работы. Сначала мы рассмотрим прагму `parallel`. Прагма `parallel` принимает те же выражения, что и директива `kernels`. В следующем ниже примере мы выделили жирным шрифтом дополнительные выражения директивы `kernels`:

```
#pragma acc parallel [ clause ]
   выражения данных      - [ reduction | private | firstprivate | copy |
                             copyin | copyout | create | no_create | present |
                             deviceptr | attach | default(none|present) ]
```

```
оптимизация ядра      - [ num_gangs | num_workers |
                          vector_length | device_type | self ]
асинхронные выражения - [ async | wait ]
условный блок         - [ if ]
```

Выражения для конструкта `loop` были упомянуты ранее в разделе `kernels`. Важно отметить, что по умолчанию конструкт `loop` в параллельном участке является независимым, а не автоматическим. Опять же, как и в директиве `kernels`, комбинированный конструкт `parallel loop` может принимать любое выражение, которое может использоваться отдельными директивами. С этим объяснением конструкта `parallel loop` мы перейдем к его добавлению в пример потоковой триады, как показано в следующем ниже листинге.

Листинг 11.3 Добавление прагмы parallel loop

OpenACC/StreamTriad/StreamTriad_par1.c

```
12    struct timespec tstart;
13    // инициализируются данные и массивы
14    double scalar = 3.0, time_sum = 0.0;
15 #pragma acc parallel loop
16    for (int i=0; i<nsize; i++) {
17       a[i] = 1.0;
18       b[i] = 2.0;
19    }
20
21    for (int k=0; k<ntimes; k++){
22       cpu_timer_start(&tstart);
23       // цикл потоковой триады
24 #pragma acc parallel loop
25       for (int i=0; i<nsize; i++){
26          c[i] = a[i] + scalar*b[i];
27       }
28       time_sum += cpu_timer_stop(tstart);
29    }
```

Вставляет комбинированный конструкт parallel loop

Вывод компилятора PGI выглядит следующим образом:

```
main:
     15, Generating Tesla code
         16, #pragma acc loop gang, vector(128)
             /* blockIdx.x threadIdx.x */
     15, Generating implicit copyout(a[:20000000],b[:20000000])
         [if not already present]
     24, Generating Tesla code
         25, #pragma acc loop gang, vector(128)
             /* blockIdx.x threadIdx.x */
     24, Generating implicit copyout(c[:20000000]) [if not already present]
         Generating implicit copyin(b[:20000000],a[:20000000])
         [if not already present]
```

Цикл параллелизуется даже без атрибута `restrict`, поскольку по умолчанию для директивы `loop` используется выражение `independent`. Это отличается от директивы, используемой по умолчанию для `kernels`, которую мы видели ранее. Тем не менее мы рекомендуем использовать в коде атрибут `restrict`, чтобы помочь компилятору генерировать наилучший код.

Результат аналогичен результату на выходе из предыдущей директивы `kernels`. В этом месте производительность кода, скорее всего, замедлится из-за перемещения данных, которое мы выделили жирным шрифтом в приведенной выше распечатке на выходе из компилятора. Не волнуйтесь, мы снова его ускорим на следующем шаге.

Прежде чем мы обратимся к перемещению данных, кратко рассмотрим редукции и конструкт `serieal`. В листинге 11.4 показан пример массовой суммы, впервые введенной в разделе 6.3.3. Массовая сумма представляет собой простую операцию редукции. Вместо прагмы векторизации SIMD языка OpenMP мы поместили прагму `parallel loop` языка OpenACC с выражением `reduction` перед циклом. Синтаксис редукции уже знаком, потому что он такой же, как и в поточном стандарте OpenMP.

Листинг 11.4 Добавление выражения reduction

`OpenACC/mass_sum/mass_sum.c`

```
 1 #include "mass_sum.h"
 2 #define REAL_CELL 1
 3
 4 double mass_sum(int ncells, int* restrict celltype,
 5                 double* restrict H, double* restrict dx,
                   double* restrict dy){
 6    double summer = 0.0;
 7 #pragma acc parallel loop reduction(+:summer)
 8    for (int ic=0; ic<ncells ; ic++) {
 9       if (celltype[ic] == REAL_CELL) {
10          summer += H[ic]*dx[ic]*dy[ic];
11       }
12    }
13    return(summer);
14 }
```

Добавляет выражение reduction в конструкт parallel loop

В выражении `reduction` можно использовать и другие операторы. К ним относятся `*`, `max`, `min`, `&`, `|`, `&&` и `||`. Для версий OpenACC до 2.6 переменная или список переменных, разделенных запятыми, ограничены скалярами, а не массивами. Но версия OpenACC 2.7 позволяет использовать массивы и составные переменные в выражении `reduction`.

Последний конструкт, который мы охватим в этом разделе, предназначен для последовательной работы. Некоторые циклы не могут выполняться параллельно. Вместо того чтобы выходить из параллельного участка, мы остаемся в нем и говорим компилятору, чтобы он просто

выполнял эту часть последовательно. Это делается с помощью директивы `serial`:

```
#pragma acc serial
```

Блоки этого кода с директивой `serial` исполняются одной бригадой (gang), состоящей из одного работника с длиной вектора один. Теперь давайте обратим наше внимание на отклик на перемещение данных.

11.2.3 Использование директив для сокращения перемещения данных между CPU и GPU

В этом разделе мы возвращаемся к теме, которую рассматривали на протяжении всей этой книги. Перемещение данных важнее, чем флопы. Хотя мы ускорили вычисления, переместив их на GPU, совокупное время выполнения замедлилось из-за стоимости перемещения данных. Решение проблемы чрезмерного перемещения данных начнет приводить к совокупному ускорению. Для этого мы добавляем в наш код конструкт `data`. В стандарте OpenACC версии 2.0 спецификация конструкта `data` выглядит следующим образом:

```
#pragma acc data [ copy | copyin | copyout | create | no_create | present |
                   deviceptr | attach | default(none|present) ]
```

Вы также увидите ссылки на такие выражения, как `present_or_copy` или в укороченном виде `pcopy`, которые проверяют наличие данных перед копированием. В них больше нет необходимости, хотя они сохранены для обеспечения обратной совместимости. Это поведение включено в состав стандартных выражений, начиная с версии 2.5 стандарта OpenACC.

Во многих выражениях `data` используется аргумент, в котором перечислены данные, подлежащие копированию или другим способам обработки. Компилятор должен получить спецификацию диапазона массива. Примером тому является следующий ниже фрагмент:

```
#pragma acc data copy(x[0:nsize])
```

Спецификация диапазона немного отличается для C/C++ и Fortran. В C/C++ первым аргументом в спецификации является начальный индекс, а вторым – длина. В Fortran первым аргументом является начальный индекс, а вторым – конечный индекс.

Существует две разновидности участков данных. Первая – это участок структурированных данных из изначального стандарта OpenACC версии 1.0. Вторая, участок динамических данных, была представлена в версии 2.0 OpenACC. Сначала мы рассмотрим участок структурированных данных.

Участок структурированных данных для простых блоков кода

Участок структурированных данных делимитирован блоком кода. Это может быть блок естественного кода, образованный циклом или участком кода, содержащимся в наборе фигурных скобок. На Fortran этот участок помечается начальной директивой и заканчивается концевой директивой. В листинге 11.5 показан пример участка структурированных данных, который начинается с директивы в строке 16 и делимитирован открывающей скобкой в строке 17 и концевой скобкой в строке 37. Мы включили в код комментарий в концевой скобке, чтобы помочь идентифицировать блок кода, который указанная скобка заканчивает.

Листинг 11.5 Прагма блока структурированных данных

OpenACC/StreamTriad/StreamTriad_par2.c

```
16 #pragma acc data create(a[0:nsize],\
                           b[0:nsize],c[0:nsize])
17    {
18
19 #pragma acc parallel loop present(a[0:nsize],\
                                   b[0:nsize])
20       for (int i=0; i<nsize; i++) {
21          a[i] = 1.0;
22          b[i] = 2.0;
23       }
24
25       for (int k=0; k<ntimes; k++){
26          cpu_timer_start(&tstart);
27          // цикл потоковой триады
28 #pragma acc parallel loop present(a[0:nsize],\
                                   b[0:nsize],c[0:nsize])
29          for (int i=0; i<nsize; i++){
30             c[i] = a[i] + scalar*b[i];
31          }
32          time_sum += cpu_timer_stop(tstart);
33       }
34
35       printf("Среднее время выполнения цикла потоковой триады составляет %lf
                msecs\n", time_sum/ntimes);
36
37    } //#pragma end acc data block(a[0:nsize],b[0:nsize],c[0:nsize])
```

Директива data определяет участок структурированных данных

Начинает участок данных

Директива present сообщает компилятору, что копия не требуется

Закрывающая скобка обозначает конец участка данных

Участок структурированных данных указывает, что в начале участка данных должно быть создано три массива. Они будут уничтожены в конце участка данных. Выражение present используется в двух параллельных циклах, чтобы избежать копирования данных для вычислительных участков.

Участок динамических данных для организации более гибкой области видимости данных

Изначально используемый языком OpenACC участок структурированных данных, в котором выделяется память, а затем располагается несколько циклов, не работает с более сложными программами. В частности, выделение памяти в объектно ориентированном коде происходит при создании объекта. Как помещать участок данных вокруг чего-то с такой программной структурой?

В целях решения этой проблемы в OpenACC версии 2.0 были добавлены участки динамических (также именуемые неструктурированными) данных. Конструкт участка динамических данных был специально создан для более сложных сценариев управления данными, таких как конструкторы и деструкторы в C++. Вместо того чтобы для определения участка данных использовать фигурные скобки, в прагме есть выражение `enter` и `exit`:

```
#pragma acc enter data
#pragma acc exit data
```

Для директивы `exit data` существует дополнительное выражение `delete`, которое можно использовать. Это использование директивы `enter/exit data` лучше всего применяется там, где происходят выделения и высвобождения памяти. Директива `enter data` должна быть размещена сразу после выделения, а директива `exit data` должна быть вставлена непосредственно перед высвобождением. Такой подход естественнее соответствует существующей области видимости переменных в приложении. Как только вы захотите повысить производительность, по сравнению с тем, что может быть достигнуто с помощью стратегии уровня цикла, возрастет важность этих участков динамических данных. С более крупными областями видимости участков динамических данных возникает необходимость в дополнительной директиве обновления данных:

```
#pragma acc update [self(x) | device(x)]
```

Аргумент `device` указывает на то, что данные на устройстве должны быть обновлены. Аргумент `self` указывает на обновление локальных данных, которые обычно являются хост-версией данных.

Давайте посмотрим на пример использования прагмы динамических данных `data` в листинге 11.6. Директива `enter data` помещается после выделения в строке 12. Директива `exit data` в строке 35 вставляется перед высвобождениями. Мы предлагаем использовать участки динамическим данных в отличие от участков структурированных данных почти во всех, кроме самого простого кода.

Листинг 11.6 Создание участков динамических данных

OpenACC/StreamTriad/StreamTriad_par3.c

```
 8 double* restrict a = malloc(nsize * sizeof(double));
 9 double* restrict b = malloc(nsize * sizeof(double));
10 double* restrict c = malloc(nsize * sizeof(double));
11
12 #pragma acc enter data create(a[0:nsize],\
                               b[0:nsize],c[0:nsize])
13
14    struct timespec tstart;
15    // инициализация данных и массивов
16    double scalar = 3.0, time_sum = 0.0;
17 #pragma acc parallel loop present(a[0:nsize],b[0:nsize])
18    for (int i=0; i<nsize; i++) {
19       a[i] = 1.0;
20       b[i] = 2.0;
21    }
22
23    for (int k=0; k<ntimes; k++){
24       cpu_timer_start(&tstart);
25       // цикл потоковой триады
26 #pragma acc parallel loop present(a[0:nsize],b[0:nsize],c[0:nsize])
27       for (int i=0; i<nsize; i++){
28          c[i] = a[i] + scalar*b[i];
29       }
30       time_sum += cpu_timer_stop(tstart);
31    }
32
33    printf("Среднее время выполнения цикла потоковой триады составляет %lf msecs\n",
             time_sum/ntimes);
34
35 #pragma acc exit data delete(a[0:nsize],\.
                             b[0:nsize],c[0:nsize])
36
37    free(a);
38    free(b);
39    free(c);
```

Начинает участок динамических данных после выделения памяти

Завершает участок динамических данных перед высвобождением памяти

Если внимательно изучить приведенный выше листинг, то можно заметить, что массивы a, b и c выделяются как на хосте, так и на устройстве, но используются только на устройстве. В листинге 11.7 мы показываем один из подходов к исправлению этой ситуации, используя процедуру `acc_malloc`, а затем помещая выражение `deviceptr` в вычислительные участки.

Листинг 11.7 Выделение данных только на устройстве

OpenACC/StreamTriad/StreamTriad_par4.c

```
1 #include <stdio.h>
2 #include <openacc.h>
```

```
 3 #include "timer.h"
 4
 5 int main(int argc, char *argv[]){
 6
 7    int nsize = 20000000, ntimes=16;
 8    double* restrict a_d =
          acc_malloc(nsize * sizeof(double));
 9    double* restrict b_d =
          acc_malloc(nsize * sizeof(double));
10    double* restrict c_d =
          acc_malloc(nsize * sizeof(double));
11
12    struct timespec tstart;
13    // инициализация данных и массивов
14    const double scalar = 3.0;
15    double time_sum = 0.0;
16 #pragma acc parallel loop deviceptr(a_d, b_d)
17    for (int i=0; i<nsize; i++) {
18       a_d[i] = 1.0;
19       b_d[i] = 2.0;
20    }
21
22    for (int k=0; k<ntimes; k++){
23       cpu_timer_start(&tstart);
24       // цикл потоковой триады
25 #pragma acc parallel loop deviceptr(a_d, b_d, c_d)
26       for (int i=0; i<nsize; i++){
27          c_d[i] = a_d[i] + scalar*b_d[i];
28       }
29       time_sum += cpu_timer_stop(tstart);
30    }
31
32    printf("Среднее время выполнения цикла потоковой триады составляет %lf msecs\n",
             time_sum/ntimes);
33
34    acc_free(a_d);
35    acc_free(b_d);
36    acc_free(c_d);
37
38    return(0);
39 }
```

Выделить память прямо на устройстве. _d обозначает указатель на устройство

Выражение deviceptr сообщает компилятору, что память уже находится на устройстве

Высвобождает память на устройстве

Как показано ниже, результат на выходе из компилятора PGI теперь намного короче:

```
16 Generating Tesla code
17 #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
25 Generating Tesla code
26 #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

Перемещение данных устранено, а требования к памяти на хосте сокращены. У нас все еще есть немного данных на выходе с откликом о сге-

нерированном вычислительном ядре, которые мы рассмотрим в разделе 11.2.4. Этот пример (листинг 11.7) работает для одномерных массивов. Для двухмерных массивов выражение `deviceptr` не принимает дескрипторный аргумент, поэтому ядро должно быть модифицировано под выполнение собственной двухмерной индексации в одномерном массиве.

При обращении к участкам данных у вас есть богатый набор директив данных и выражений перемещения данных, которые можно использовать для сокращения ненужных перемещений данных. Тем не менее мы не охватили еще больше выражений и функций OpenACC, которые могут быть полезны в специализированных ситуациях.

11.2.4 Оптимизирование вычислительных ядер GPU

Как правило, вы окажете большее влияние, увеличив число работающих на GPU вычислительных ядер и сократив перемещения данных, чем оптимизируя сами вычислительные ядра GPU. Компилятор OpenACC хорошо справляется с продуцированием ядер, и потенциальные выгоды от дальнейшей оптимизации будут невелики. Время от времени можно помогать компилятору повышать производительность ключевых ядер настолько, что это бы стоило некоторых усилий.

В данном разделе мы пройдемся по общим стратегиям этих оптимизаций. Сначала коснемся терминологии, которая используется в стандарте OpenACC. Как показано на рис. 11.3, OpenACC определяет абстрактные уровни параллелизма, которые применимы к многочисленным аппаратным устройствам.

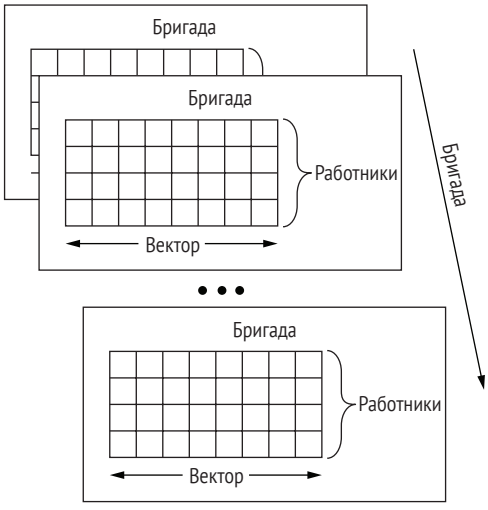


Рис. 11.3 Иерархия уровней в OpenACC: бригады, работники и векторы

OpenACC определяет следующие ниже уровни параллелизма:

- *бригада* (gang) – независимый рабочий блок, который делится ресурсами. Бригада также может синхронизироваться внутри группы, но не между группами. Для GPU-процессоров бригады могут быть

соотнесены с поточными блоками CUDA или рабочими группами OpenCL;
- *работники* – варп в CUDA или элементы работы внутри рабочей группы в OpenCL;
- *вектор* – вектор SIMD на CPU и рабочая группа SIMT или варп на GPU со ссылками на сплошную память.

Ниже приведено несколько примеров задания уровня конкретной директивы цикла:

```
#pragma acc parallel loop vector
#pragma acc parallel loop gang
#pragma acc parallel loop gang vector
```

Внешний цикл должен быть циклом бригад (`gang`), а внутренний – циклом векторов (`vector`). Между ними может появиться цикл работников (`worker`). Последовательный цикл (`seq`) может появиться на любом уровне.

Для большинства современных GPU длина вектора должна быть кратна 32, поэтому она является целым числом, кратным размеру варпа. Она должна быть не больше максимального числа потоков в расчете на блок, т. е. обычно составляет около 1024 на текущих GPU (см. результат на выходе из команды `pgaccelinfo` на рис. 11.2). В приведенных здесь примерах компилятор PGI задает длину вектора равной разумному значению 128. Это значение может быть изменено для цикла директивой `vector_length(x)`.

В каком сценарии следует изменять параметр `vector_length`? Если внутренний цикл сплошных данных меньше 128, то часть вектора останется неиспользованной. В этом случае бывает полезно сокращать это значение. Еще одним вариантом, как мы вскоре обсудим, будет сворачивание пары внутренних циклов, получая более длинный вектор.

Настройку `worker` можно модифицировать с помощью выражения `num_workers`. Однако в примерах этой главы она не используется. Тем не менее ее бывает полезно увеличивать при укорочении векторной длины либо для дополнительного уровня параллелизации. Если вашему коду нужно выполнять синхронизацию внутри параллельной рабочей группы, то следует использовать работника, но OpenACC не предоставляет пользователю директиву синхронизации. В уровне работника также совместно используются такие ресурсы, как кеш и локальная память.

Остальная часть параллелизации выполняется с помощью бригад, которые являются асинхронным параллельным уровнем. Большое число бригад на GPU-процессорах важны тем, что они скрывают задержку и обеспечивают высокую частоту. Как правило, компилятор задает их число равным крупному числу, поэтому пользователю не требуется его переопределять. На всякий случай, если вам, возможно, потребоваться это сделать, то для этого случая имеется выражение `num_gangs`.

Многие настройки будут подходить только для конкретной части оборудования. Параметр `device_type(type)` перед выражением ограничива-

ет устройство указанным типом. Этот параметр остается активным до тех пор, пока не будет найдено следующее выражение `device_type`. Например:

```
1 #pragma acc parallel loop gang \
2     device_type(acc_device_nvidia) vector_length(256) \
3     device_type(acc_device_radeon) vector_length(64)
4 for (int j=0; j<jmax; j++){
5       #pragma acc loop vector
6       for (int i=0; i<imax; i++){
7           <работа>
8       }
9 }
```

Для получения списка допустимых типов устройств просмотрите заголовочный файл openacc.h для PGI v19.7. Обратите внимание, что в строках показанного ранее заголовочного файла openacc.h нет устройства `acc_device_radeon`, поэтому компилятор PGI не поддерживает устройство AMD Radeon™. Это означает, что возле строки 3 в приведенном выше примере кода нам нужен препроцессор C `ifdef`, чтобы компилятор PGI не жаловался.

```
Excerpt from openacc.h file for PGI

27 typedef enum{
28     acc_device_none          = 0,
29     acc_device_default       = 1,
30     acc_device_host          = 2,
31     acc_device_not_host      = 3,
32     acc_device_nvidia        = 4,
33     acc_device_pgi_opencl    = 7,
34     acc_device_nvidia_opencl = 8,
35     acc_device_opencl        = 9,
36     acc_device_current       = 10
37     } acc_device_t;
```

Синтаксис директивы `kernels` слегка отличается, при этом параллельный тип применяется к каждой директиве `loop` отдельно и принимает аргумент `int` напрямую:

```
#pragma acc kernels loop gang
for (int j=0; j<jmax; j++){
      #pragma acc loop vector(64)
      for (int i=0; i<imax; i++){
         <работа>
      }
}
```

Циклы могут быть скомбинированы с выражением `collapse(n)`. Это особенно полезно, если есть два малых внутренних цикла, вышагивающих по данным в сплошном порядке. Комбинирование этих выражений

позволяет использовать более длинную длину вектора. Циклы должны быть плотно вложенными.

ОПРЕДЕЛЕНИЕ Два или более цикла, в которых нет дополнительных инструкций между инструкциями `for` или `do` либо между концами циклов, являются *плотно вложенными циклами*.

Примером комбинирования двух циклов для использования длинного вектора является следующий ниже фрагмент кода:

```
#pragma acc parallel loop collapse(2) vector(32)
for (int j=0; j<8; j++){
   for (int i=0; i<4; i++){
      <работа>
   }
}
```

В OpenACC версии 2.0 добавлено выражение `tile`, которое можно использовать для оптимизации. Можно задавать размер плитки либо использовать звездочки, чтобы компилятор мог выбирать сам:

```
#pragma acc parallel loop tile(*,*)
for (int j=0; j<jmax; j++){
      for (int i=0; i<imax; i++){
         <работа>
      }
}
```

Теперь самое время опробовать различные оптимизации вычислительного ядра. Пример потоковой триады не показал никаких реальных преимуществ от наших попыток оптимизации, поэтому мы будем работать со стенсильным примером, использованным в нескольких предыдущих главах.

Связанный со стенсильным примером, исходный код этой главы проходит через те же первые два шага перемещения вычислительных циклов на GPU, а затем сокращает перемещение данных. Стенсильный код также требует одно дополнительное изменение. На CPU мы обмениваемся указателями в конце цикла. На GPU в строках 45–50 мы должны скопировать новые данные обратно в изначальный массив. В следующем ниже листинге приведен пример стенсильного кода, к котором эти шаги показаны полностью.

Листинг 11.8 Пример стенсиля с циклами вычислений на GPU и оптимизированным движением данных

OpenACC/Stencil/Stencil_par3.c

```
17 #pragma acc enter data create( \                    | Директивы участка
        x[0:jmax][0:imax], xnew[0:jmax][0:imax])      | динамических данных
18
```

```
19 #pragma acc parallel loop present( \
        x[0:jmax][0:imax], xnew[0:jmax][0:imax])
20    for (int j = 0; j < jmax; j++){
21       for (int i = 0; i < imax; i++){
22          xnew[j][i] = 0.0;
23          x[j][i] = 5.0;
24       }
25    }
26
27 #pragma acc parallel loop present( \
        x[0:jmax][0:imax], xnew[0:jmax][0:imax])
28    for (int j = jmax/2 - 5; j < jmax/2 + 5; j++){
29       for (int i = imax/2 - 5; i < imax/2 -1; i++){
30          x[j][i] = 400.0;
31       }
32    }
33
34    for (int iter = 0; iter < niter; iter+=nburst){
35
36       for (int ib = 0; ib < nburst; ib++){
37          cpu_timer_start(&tstart_cpu);
38 #pragma acc parallel loop present( \
        x[0:jmax][0:imax], xnew[0:jmax][0:imax])
39          for (int j = 1; j < jmax-1; j++){
40             for (int i = 1; i < imax-1; i++){
41                xnew[j][i]=(x[j][i]+x[j][i-1]+x[j][i+1]+
                                 x[j-1][i]+x[j+1][i])/5.0;
42             }
43          }
44
45 #pragma acc parallel loop present( \
        x[0:jmax][0:imax], xnew[0:jmax][0:imax])
46          for (int j = 0; j < jmax; j++){
47             for (int i = 0; i < imax; i++){
48                x[j][i] = xnew[j][i];
49             }
50          }
51          cpu_time += cpu_timer_stop(tstart_cpu);
52       }
53
54       printf("Итер %d\n",iter+nburst);
55    }
56
57 #pragma acc exit data delete( \
        x[0:jmax][0:imax], xnew[0:jmax][0:imax])
```

Директивы вычислительного участка

Директивы вычислительного участка

Директивы участка динамических данных

Прежде всего обратите внимание на то, что мы используем директивы участка динамических данных, поэтому участок данных не обернут фигурными скобками, как в случае с участком структурированных данных. Динамический участок начинается с участка данных, когда он сталкивается с директивой `enter`, и заканчивается, когда он достигает директивы `exit` независимо от того, какой путь лежит между двумя директивами.

В данном случае это прямая линия исполнения от директивы `enter` до директивы `exit`. Мы добавим выражение `collapse` в параллельный цикл, чтобы соратить накладные расходы для двух циклов. Это изменение показано в следующем ниже листинге.

Листинг 11.9 Пример стенсиля с выражением `collapse`

`OpenACC/Stencil/Stencil_par4.c`

```
36       for (int ib = 0; ib < nburst; ib++){
37          cpu_timer_start(&tstart_cpu);
38 #pragma acc parallel loop collapse(2)\
39      present(x[0:jmax][0:imax], xnew[0:jmax][0:imax])
40          for (int j = 1; j < jmax-1; j++){
41             for (int i = 1; i < imax-1; i++){
42                xnew[j][i]=(x[j][i]+x[j][i-1]+x[j][i+1]+
                                 x[j-1][i]+x[j+1][i])/5.0;
43             }
44          }
45 #pragma acc parallel loop collapse(2)\
46      present(x[0:jmax][0:imax], xnew[0:jmax][0:imax])
47          for (int j = 0; j < jmax; j++){
48             for (int i = 0; i < imax; i++){
49                x[j][i] = xnew[j][i];
50             }
51          }
52          cpu_time += cpu_timer_stop(tstart_cpu);
53
54       }
```

Добавляет выражение collapse в директиву parallel loop

Мы также можем попробовать использовать выражение `tile`. Мы начнем с того, что позволим компилятору определять размер плитки, как показано в строках 41 и 48 в следующем ниже листинге.

Листинг 11.10 Пример стенсиля с выражением `tile`

`OpenACC/Stencil/Stencil_par5.c`

```
39       for (int ib = 0; ib < nburst; ib++){
40          cpu_timer_start(&tstart_cpu);
41 #pragma acc parallel loop tile(*,*) \
42      present(x[0:jmax][0:imax], xnew[0:jmax][0:imax])
43          for (int j = 1; j < jmax-1; j++){
44             for (int i = 1; i < imax-1; i++){
45                xnew[j][i]=(x[j][i]+x[j][i-1]+x[j][i+1]+
                                  x[j-1][i]+x[j+1][i])/5.0;
46             }
47         }
48 #pragma acc parallel loop tile(*,*) \
49      present(x[0:jmax][0:imax], xnew[0:jmax][0:imax])
50         for (int j = 0; j < jmax; j++){
51            for (int i = 0; i < imax; i++){
```

Добавляет выражение tile в директиву параллельного цикла

```
52              x[j][i] = xnew[j][i];
53            }
54          }
55          cpu_time += cpu_timer_stop(tstart_cpu);
56
57      }
```

Изменение во временах выполнения в результате этих оптимизаций невелико, по сравнению с улучшением, наблюдаемым при первоначальной имплементации OpenACC. В табл. 11.1 приведены результаты для GPU NVIDIA V100 с компилятором PGI версии 19.7.

Таблица 11.1 Времена выполнения для оптимизаций стенсильных вычислительных ядер OpenACC

	Время выполнения стенсильного вычислительного ядра (мс)
Последовательный код CPU	5.237
Добавление участков вычислений и данных	0.818
Добавление выражения `collapse(2)`	0.802
Добавление выражения `tile(*,*)`	0.806

Мы попытались поменять длину вектора на 64 или 256 и разные размеры плиток, но не увидели никакого улучшения времени выполнения. Более сложный код способен извлечь из оптимизаций вычислительных ядер больше пользы, но обратите внимание, что любая специализация параметров, таких как длина вектора, влияет на переносимость компиляторами в условиях разных архитектур.

Еще одной целью оптимизации является имплементирование обмена (swap) указателями в конце цикла. Обмен указателями используется в изначальном коде CPU как быстрый способ возвращения данных в изначальный массив. Копирование данных обратно в исходный массив удваивает время выполнения на GPU. Сложность прагма-ориентированных языков заключается в том, что для обмена указателями в параллельном участке требуется одновременный обмен указателями хоста и устройства.

11.2.5 Резюме результирующих производительностей для потоковой триады

Производительность времени выполнения при конвертации в GPU демонстрирует типичный шаблон. Перемещение вычислительных ядер на GPU приводит к замедлению примерно в 3 раза, как показано в имплементациях ядра 2 и в параллельном случае 1 в табл. 11.2. В случае ядра 1 вычислительный цикл параллелизовать не получается. При его выполнении в последовательном порядке на GPU производительность была еще медленнее. После того как перемещение данных было сокращено в ядре 3 и в параллельных случаях 2–4, времена выполнения ускорились

в 67 раз. Конкретный тип участка данных не столь важен для производительности, но бывает важен для задействования переноса дополнительных циклов в более сложных исходных кодах.

Таблица 11.2 Времена выполнения оптимизаций вычислительных ядер потоковой триады OpenACC

	Время выполнения вычислительного ядра потоковой триады (мс)
Последовательный код CPU	39.6
Ядро 1. Не получается распараллелить цикл	1771
Ядро 2. Добавляет вычислительный участок	118.5
Ядро 3. Добавляет участок динамических данных	0.590
Параллельный 1. Добавляет вычислительный участок	118.8
Параллельный 2. Добавляет участок структурированных данных	0.589
Параллельный 3. Добавляет участок динамических данных	0.590
Параллельный 4. Выделяет данные только на устройстве	0.586

11.2.6 Продвинутые техники OpenACC

Для манипулирования более сложным исходным кодом в OpenACC имеется много других функциональностей. Мы остановимся на них кратко, чтобы дать вам представление о том, какие есть возможности.

Манипулирование функциями с помощью директивы OpenACC routine

OpenACC v1.0 требует, чтобы используемые в вычислительных ядрах функции были вставляемыми. В версию 2.0 была добавлена директива `routine` в двух разных версиях, чтобы упростить вызов процедур. Эти две версии таковы:

```
#pragma acc routine [gang | worker | vector | seq | bind | no_host | device_type]
#pragma acc routine(name) [gang | worker | vector | seq | bind | no_host |
                           device_type]
```

В C и C++ директива `routine` должна появляться непосредственно перед прототипом или определением функции. Именованная версия может появляться в любом месте до того, как функция будет определена или использована. Версия для Fortran должна включать в себя директиву `!#acc routine` в самом теле функции или в теле интерфейса.

Избегание гоночных условий с помощью директив OpenACC atomic

Многие поточные подпрограммы имеют совместную переменную, которая должна обновляться многочисленными потоками. Этот программный конструкт является часто встречающимся узким местом производительности и потенциальным гоночным условием. В целях урегулирования этой ситуации OpenACC v2 предоставляет атомики, позволяющие обращаться к месту хранения только по одному потоку за один раз. Синтаксис и допустимые выражения для директивы `atomic` таковы:

```
#pragma acc atomic [read | write | update | capture]
```

Если выражение не указано, то по умолчанию используется `update`. Ниже приведен пример использования выражения `atomic`

```
#pragma acc atomic
cnt++;
```

Асинхронные операции в OpenACC

Наложение операций OpenACC помогают повышать производительность. Надлежащим термином для операций, которые накладываются друг на друга, является термин «асинхронный». OpenACC предоставляет эти асинхронные операции в виде выражений и директив `async` и `wait`. Выражение `async` добавляется в директиву работы или директиву данных с необязательным целочисленным аргументом:

```
#pragma acc parallel loop async([<целое_число>])
```

Ключевое слово `wait` может быть либо директивой, либо выражением, добавленным в директиву работы или директиву данных. Следующий ниже псевдокод в листинге 11.11 показывает, как оно может использоваться для запуска вычислений на гранях `x` и `y` вычислительной сетки, а затем ожидания результатов с целью обновления значений ячеек для следующей итерации.

Листинг 11.11 Пример асинхронного ожидания в OpenACC

```
for (int n = 0; n < ntimes; ) {
   #pragma acc parallel loop async
      <проход по грани x>
   #pragma acc parallel loop async
      <проход по грани y>
   #pragma acc wait
   #pragma acc parallel loop
      <Обновить значения ячеек из граневых флюксий>
}
```

Унифицированная память, позволяющая избегать управления перемещением данных

Хотя унифицированная память в настоящее время не является частью стандарта OpenACC, существуют экспериментальные разработки, позволяющие системе управлять перемещением памяти. Такая экспериментальная имплементация унифицированной памяти доступна в CUDA и компиляторе PGI OpenACC. Используя флаг `-ta=tesla:managed` с компилятором PGI и новейшими GPU-процессорами NVIDIA, вы можете попробовать их имплементацию унифицированной памяти. Хотя кодирование упрощено, влияние на производительность пока что неизвестно и будет меняться по мере взросления компиляторов.

Интероперабельность с библиотеками и вычислительными ядрами CUDA

OpenACC предоставляет несколько директив и функций, позволяющих взаимооперировать с библиотеками CUDA. При вызове библиотек необходимо сообщать компилятору, что вместо данных хоста нужно использовать указатели устройств. Для этой цели можно использовать директиву `host_data`:

```
#pragma acc host_data use_device(x, y)
cublasDaxpy(n, 2.0, x, 1, y, 1);
```

Мы показали аналогичный пример, когда выделяли память с помощью `acc_malloc` в листинге 11.7. При использовании `acc_malloc` или `cudaMalloc` возвращаемый указатель уже находится на устройстве. В этом случае мы использовали выражение `deviceptr` для передачи указателя на участок данных.

Одной из наиболее распространенных ошибок при программировании GPU-процессоров на любом языке является перепутывание указателя устройства и указателя хоста. Попробуйте найти 86 Пайк-Плейс, Сан-Франциско, когда на самом деле речь идет о 86 Пайк-Плейс, Сиэтл. Указатель устройства указывает на другой физический блок памяти на оборудовании GPU.

На рис. 11.4 показаны три разные операции, которые мы рассмотрели, чтобы помочь вам понять различия. В первом случае процедура `malloc` возвращает указатель хоста.

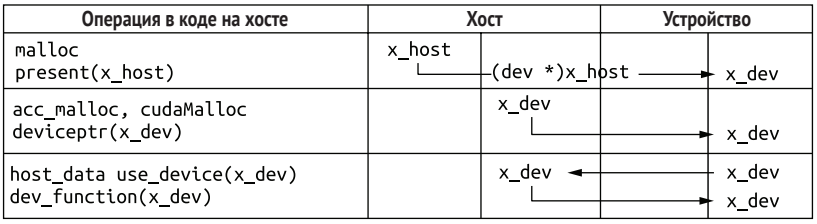


Рис. 11.4 Чем является этот указатель: указателем устройства либо указателем хоста? Один указывает соответственно на память GPU, а другой – на память CPU. OpenACC поддерживает соответствие между массивами в двух адресных пространствах и предоставляет процедуры для извлечения каждого из них

Выражение `present` конвертирует это в указатель устройства для вычислительного ядра устройства. Во втором случае, когда мы выделяем память на устройстве с помощью `acc_malloc` или `cudaMalloc`, нам предоставляется указатель устройства. Выражение `deviceptr` используется для его отправки на GPU без каких-либо изменений. В последнем случае у нас вообще нет указателя хоста. Мы должны использовать директиву `host_data use_device(var)` для извлечения указателя устройства на хост. Это делается для того, чтобы у нас был указатель для отправки обратно на устройство в списке аргументов функции устройства.

Рекомендуется добавлять в указатели `_h` либо `_d`, чтобы уточнять их валидный контекст. В наших примерах принято допущение, что все указатели и массивы находятся на хосте, за исключением тех, которые заканчиваются на `_d`, что относится к любому указателю устройства.

Управление многочисленными устройствами в OpenACC

Многие современные системы HPC уже имеют несколько GPU. Кроме того, можно предвидеть, что мы будем иметь узлы с разными ускорителями. Способность управлять используемыми устройствами становится все более и более важной. OpenACC предоставляет нам эту способность с помощью следующих ниже функций:

- `int acc_get_num_devices(acc_device_t);`
- `acc_set_device_type()` / `acc_get_device_type();`
- `acc_set_device_num()` / `acc_get_device_num()`.

Теперь мы охватили OpenACC настолько, насколько это можно было сделать на дюжине страниц. Показанных умений достаточно, чтобы иметь возможность приступить к имплементации. В стандарте OpenACC имеется гораздо больше функциональности, но бо́льшая ее часть предназначена для более сложных ситуаций или низкоуровневых интерфейсов, которые не нужны для приложений начального уровня.

11.3 OpenMP: чемпион в тяжелом весе вступает в мир ускорителей

Ускорительная способность OpenMP является захватывающим дополнением к традиционной модели потокообразования. В этом разделе мы покажем вам, как начинать работу с ее директивами. Мы будем использовать те же примеры, что и в разделе 11.2 OpenACC. К концу этого раздела у вас должно быть некоторое представление о том, как два похожих языка соотносятся и какой из них, возможно, будет более подходящим вариантом выбора для вашего приложения.

Где находятся ускорительные директивы OpenMP по сравнению с OpenACC? На данный момент имплементации OpenMP заметно менее зрелы, хотя и быстро совершенствуются. В настоящее время доступны имплементации для следующих ниже GPU:

- Cray была первой, которая в 2015 году представила имплементацию OpenMP, ориентированную на GPU-процессоры NVIDIA. Cray теперь поддерживает OpenMP v4.5;
- IBM полностью поддерживает OpenMP v4.5 на процессоре Power 9 и GPU-процессорах NVIDIA;
- Clang v7.0+ поддерживает выгрузки OpenMP v4.5 на GPU-процессоры NVIDIA;
- GCC v6+ может выполнять выгрузку на GPU-процессоры AMD; v7+ может выполнять выгрузку на GPU-процессоры NVIDIA.

Две наиболее зрелые имплементации, Cray и IBM, имеются только в соответствующих системах. К сожалению, не у всех разработчиков есть доступ к системам этих поставщиков, но существуют более широкодоступные компиляторы. Два из этих компиляторов, Clang и GCC, находятся в стадии разработки, и сейчас имеются их маргинальные версии. Следите за развитием этих компиляторов. В примерах данного раздела используется CUDA v10 и компилятор IBM® XL 16.

11.3.1 Компилирование исходного кода OpenMP

Мы начнем с настройки среды сборки и компилирования исходного кода OpenMP. CMake имеет модуль OpenMP, но при этом не имеет явной поддержки ускорительных директив OpenMP. Мы включаем в состав модуль OpenMPAccel, который вызывает регулярный модуль OpenMP и добавляет необходимые для ускорителя флаги. Он также проверяет поддерживаемую версию OpenMP, и если она не является версией 4.0 или новее, то он генерирует ошибку. Модуль CMake включен вместе с исходным кодом главы.

В листинге 11.12 показаны выдержки из главного файла CMakeLists.txt этой главы. Отклик на выходе из большинства компиляторов OpenMP сейчас слабо организован, поэтому установка флага `-DCMAKE_OPENMPACCEL` для CMake принесет лишь минимальную пользу. В указанных примерах мы будем использовать другие инструменты, чтобы заполнить этот пробел.

Листинг 11.12 Выдержки из makefile-файла OpenMPaccel

OpenMP/StreamTriad/CMakeLists.txt

```
10 if (NOT CMAKE_OPENMPACCEL_VERBOSE)
11     set(CMAKE_OPENMPACCEL_VERBOSE true)
12 endif (NOT CMAKE_OPENMPACCEL_VERBOSE)
13
14 if (CMAKE_C_COMPILER_ID MATCHES "GNU")
15     set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -fstrict-aliasing")
16 elseif (CMAKE_C_COMPILER_ID MATCHES "Clang")
17     set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -fstrict-aliasing")
18 elseif (CMAKE_C_COMPILER_ID MATCHES "XL")
19     set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -qalias=ansi")
20 elseif (CMAKE_C_COMPILER_ID MATCHES "Cray")
21     set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -h restrict=a")
22 endif (CMAKE_C_COMPILER_ID MATCHES "GNU")
23
24 find_package(OpenMPAccel)
25
26 if (CMAKE_C_COMPILER_ID MATCHES "XL")
27     set(OpenMPAccel_C_FLAGS
        "${OpenMPAccel_C_FLAGS} -qreport")
28 elseif (CMAKE_C_COMPILER_ID MATCHES "GNU")
29     set(OpenMPAccel_C_FLAGS
        "${OpenMPAccel_C_FLAGS} -fopt-info-omp")
```

Модуль CMake устанавливает компиляторные флаги для ускорительных устройств OpenMP

Добавляет копмиляторный отклик для ускорительных директив

```
30 endif (CMAKE_C_COMPILER_ID MATCHES "XL")
31
32 if (CMAKE_OPENMPACCEL_VERBOSE)
33     set(OpenACC_C_FLAGS "${OpenACC_C_FLAGS} ${OpenACC_C_VERBOSE}")
34 endif (CMAKE_OPENMPACCEL_VERBOSE)
35
36 # Добавляет сборочную цель stream_triad_par1, используя файлы исходного кода
37 add_executable(StreamTriad_par1 StreamTriad_par1.c timer.c timer.h)
38 set_target_properties(StreamTriad_par1 PROPERTIES
                         COMPILE_FLAGS ${OpenMPAccel_C_FLAGS})
39 set_target_properties(StreamTriad_par1 PROPERTIES
                         LINK_FLAGS "${OpenMPAccel_C_FLAGS}")
```

Добавляет ускорительные флаги OpenMP для компилирования и связывания потоковой триады

Простой файл makefile также можно использовать для генерирования примеров исходного кода, скопировав или связав их с модулем `Makefile` одним из следующих способов:

```
ln -s Makefile.simple.xl Makefile
cp Makefile.simple.xl Makefile
```

В следующем ниже фрагменте кода показаны предлагаемые флаги для ускорительных директив OpenMP в простых файлах makefile для компиляторов IBM XL и GCC:

```
Makefile.simple.xl
 6 CFLAGS:=-qthreaded -g -O3 -std=gnu99 -qalias=ansi -qhot -qsmp=omp \
           -qoffload -qreport
 7
 8 %.o: %.c
 9   ${CC} ${CFLAGS} -c $^
10
11 StreamTriad: StreamTriad.o timer.o
12   ${CC} ${CFLAGS} $^ -o StreamTriad

Makefile.simple.gcc
 6 CFLAGS:= -g -O3 -std=gnu99 -fstrict-aliasing \
 7          -fopenmp -foffload=nvptx-none -foffload=-lm -fopt-info-omp
 8
 9 %.o: %.c
10   ${CC} ${CFLAGS} -c $^
11
12 StreamTriad: StreamTriad.o timer.o
13   ${CC} ${CFLAGS} $^ -o StreamTriad
```

11.3.2 Генерирование параллельной работы на GPU с помощью OpenMP

Теперь нам нужно сгенерировать параллельную работу на GPU. Параллельные абстракции устройств OpenMP сложнее, чем мы видели

в OpenACC. Но эта сложность также обеспечивает более высокую гибкость в планировании работы в будущем. На данный момент вы должны предварять каждый цикл следующей ниже директивой:

```
#pragma omp target teams distribute parallel for simd
```

Это длинная и запутанная директива. Давайте рассмотрим каждую ее часть, как показано на рис. 11.5. Первые три выражения задают ресурсы оборудования:

- директива `target` относится к устройству;
- директива `teams` создают лигу звеньев;
- директива `distribute` распределяет работу по звеньям.

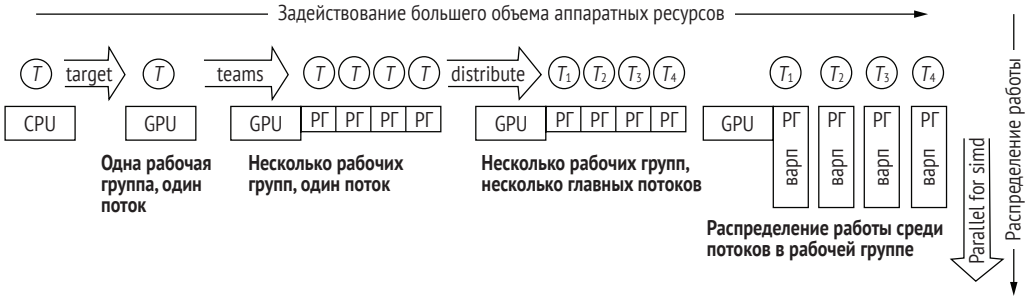


Рис. 11.5 Директивы `target`, `teams` и `distribute` позволяют задействовать больше ресурсов оборудования. Директива `parallel for simd` распределяет работу внутри каждой рабочей группы

Остальные три – это выражения параллельной работы. Все три выражения необходимы для переносимости. Это связано с тем, что имплементации компиляторов распределяют работу по-разному:

- директива `parallel` реплицирует работу на каждом потоке;
- директива `for` рапределяет работу внутри каждого звена (team);
- директива `simd` рапределяет работу среди потоков (GCC).

Для вычислительных ядер с тремя вложенными циклами один из способов распределения работы заключается в следующем:

```
Цикл k: #pragma omp target teams distribute
Цикл j: #pragma omp parallel for
Цикл i: #pragma omp simd
```

Каждый компилятор OpenMP может распределять работу по-разному, что требует нескольких вариантов этой схемы. Цикл `simd` должен быть внутренним циклом во всех местоположениях сплошной памяти. Некоторое упрощение этой сложности вводится с помощью выражения `loop` в OpenMP v5.0, как мы представим в разделе 11.3.5. В эту директиву также можно добавлять выражения:

```
private, firstprivate, lastprivate, shared, reduction, collapse, dist_schedule
```

Многие из этих выражений знакомы с OpenACC и ведут себя точно так же. Одним из главенствующих отличий от OpenACC является принятый по умолчанию способ манипулирования данными при входе в участок параллельной работы. Компиляторы OpenACC обычно перемещают все необходимые массивы на устройство. В случае OpenMP есть две возможности:

- скаляры и статически выделенные массивы перед исполнением по умолчанию перемещаются на устройство;
- данные, выделенные в куче, должны копироваться на устройство и с него в явной форме.

Давайте посмотрим на простой пример добавления директивы параллельной работы в листинге 11.13. Мы используем статически выделенные массивы, которые ведут себя так, как если бы они были выделены в стеке; хотя из-за большого размера фактическая память могла бы быть выделена компилятором в куче.

Листинг 11.13 Добавление прагм OpenMP для распараллеливания работы на GPU

OpenMP/StreamTriad/StreamTriad_par1.c

```
 6 int main(int argc, char *argv[]){
 7
 8    int nsize = 20000000, ntimes=16;
 9    double a[nsize];
10    double b[nsize];
11    double c[nsize];
12
13    struct timespec tstart;
14    // инициализация данных и массивов
15    double scalar = 3.0, time_sum = 0.0;
16 #pragma omp target teams distribute parallel for simd
17    for (int i=0; i<nsize; i++) {
18       a[i] = 1.0;
19       b[i] = 2.0;
20    }
21
22    for (int k=0; k<ntimes; k++){
23       cpu_timer_start(&tstart);
24       // цикл потоковой триады
25 #pragma omp target teams distribute parallel for simd
26       for (int i=0; i<nsize; i++){
27          c[i] = a[i] + scalar*b[i];
28       }
```

Выделение статических массивов на хосте

```
29       time_sum += cpu_timer_stop(tstart);
30    }
31
32    printf("Среднее время выполнения цикла потоковой триады составляет %lf secs\n",
            time_sum/ntimes);
```

Отклик на выходе из компилятора IBM XL показывает, что два вычислительных ядра выгружаются на GPU, но никакой другой информации не предлагается. GCC вообще не дает никакого отклика. Результат на выходе из IBM XL таков:

```
"" 1586-672 (I) GPU OpenMP Runtime elided for offloaded kernel
                '__xl_main_l15_OL_1'
"" 1586-672 (I) GPU OpenMP Runtime elided for offloaded kernel
                '__xl_main_l23_OL_2'
```

В целях получения некоторой информации о том, что проделал компилятор IBM XL, мы будем использовать профилировщик NVIDIA:

```
nvprof ./StreamTriad_par1
```

Первая часть распечатки такова:

```
==141409== Profiling application: ./StreamTriad_par1
==141409== Profiling result:
Time(%)  Time   Calls     Avg       Min         Max     Name
64.11% 554.30ms  7652  72.439us  1.2160us    79.039us   [CUDA memcpy DtoH]
34.79% 300.82ms  7650  39.323us  23.392us    48.767us   [CUDA memcpy HtoD]
 1.06% 9.1479ms    16  571.75us  571.39us    572.32us   __xl_main_l23_OL_2
 0.04% 363.07us     1  363.07us  363.07us    363.07us   __xl_main_l15_OL_1
```

Из этой распечатки мы теперь знаем, что делается копия памяти с хоста на устройство (в распечатке это `HtoD`), а затем обратно с устройства на хост (в распечатке это `DtoH`). Результат `nvprof` на выходе из GCC аналогичен, но без номеров строк. Более подробная информация о порядке выполнения операций может быть получена, если выполнить следующую ниже команду:

```
nvprof --print-gpu-trace ./StreamTriad_par1
```

Большинство программ не пишется со статически выделенными массивами. Давайте взглянем на более распространенный случай, когда массивы выделяются динамически, как показано в следующем ниже листинге.

Листинг 11.14 Директива параллельной работы с динамически выделяемыми массивами

OpenMP/StreamTriad/StreamTriad_par2.c

```
 9    double* restrict a =
         malloc(nsize * sizeof(double));
10    double* restrict b =
         malloc(nsize * sizeof(double));
11    double* restrict c =
         malloc(nsize * sizeof(double));
12
13    struct timespec tstart;
14    // инициализация данных и массивов
15    double scalar = 3.0, time_sum = 0.0;
16 #pragma omp target teams distribute \
              parallel for simd \
17            map(a[0:nsize], b[0:nsize],
                  c[0:nsize])
18    for (int i=0; i<nsize; i++) {
19       a[i] = 1.0;
20       b[i] = 2.0;
21    }
22
23    for (int k=0; k<ntimes; k++){
24       cpu_timer_start(&tstart);
25       // цикл потоковой триады
26 #pragma omp target teams distribute \
              parallel for simd \
27            map(a[0:nsize], b[0:nsize],
                  c[0:nsize])
28       for (int i=0; i<nsize; i++){
29          c[i] = a[i] + scalar*b[i];
30       }
31       time_sum += cpu_timer_stop(tstart);
32    }
33
34    printf("Среднее время выполнение цикла потоковой триады составляет %lf secs\n",
             time_sum/ntimes);
35
36    free(a);
37    free(b);
38    free(c);
```

Динамически выделяемая память

Директива параллельной работы для памяти, выделяемой в куче

Обратите внимание, что в строках 16 и 26 добавлено выражение map. Если вы попробуете директиву без этого выражения, хотя она и прекрасно компилируется компилятором IBM XLC, то во время выполнения вы получите вот это сообщение[1]:

```
1587-164 Encountered a zero-length array section that points to memory
         starting at address 0x200020000010. Because this memory is not currently
         mapped on the target device 0, a NULL pointer will be passed to the
         device.
1587-175 The underlying GPU runtime reported the following error "an illegal
         memory access was encountered".
1587-163 Error encountered while attempting to execute on the target device
         0. The program will stop.
```

Однако компилятор GCC компилирует и выполняет код нормально без директивы map. Следовательно, компилятор GCC перемещает выделенную в куче память на устройство, в то время как IBM XLC этого не делает. Для удобства переносимости необходимо включить в исходный код приложения выражение map.

В OpenMP также есть выражение `reduction` для директив, предназначенных для участков параллельной работы. Его синтаксис аналогичен синтаксису для директив OpenMP и OpenACC для работы в потоках. Пример директивы выглядит следующим образом:

```
#pragma omp teams distribute parallel for simd reduction(+:sum)
```

11.3.3 Создание участков данных для управления перемещением данных на GPU с помощью OpenMP

Теперь, когда мы перенесли работу на GPU, мы можем добавить участки данных для управления перемещением данных на GPU и с него. Директивы перемещения данных в OpenMP аналогичны директивам в OpenACC как со структурированной, так и с динамической версией. Форма директивы такова:

```
#pragma omp target data [ map() | use_device_ptr() ]
```

Директивы работы заключены в участок структурированных данных, как показано в листинге 11.15. Данные копируются на GPU, если их там еще нет. Затем эти данные поддерживаются до тех пор, пока не закончится блок (в строке 35), и копируются обратно. Это значительно

[1] Перевод: 1587-164 Обнаружен раздел с массивом нулевой длины, который указывает на память, начинающуюся с адреса 0x200020000010. Поскольку эта память в настоящее время не соотносится с целевым устройством 0, на устройство будет передан указатель на NULL. 1587-175 Опорная среда выполнения GPU сообщила о следующей ошибке: «обнаружен незаконный доступ к памяти». 1587-163 Ошибка, обнаруженная при попытке исполнения на целевом устройстве 0. Программа остановится. – *Прим. перев.*

сокращает передачу данных для каждого цикла параллельной работы и должно приводить к нетто ускорению совокупного времени выполнения приложения.

Листинг 11.15 Добавление прагм OpenMP для создания участка структурированных данных на GPU

OpenMP/StreamTriad/StreamTriad_par3.c

```
17 #pragma omp target data map(to:a[0:nsize], \
                               b[0:nsize], c[0:nsize])
18    {
19 #pragma omp target teams distribute \
              parallel for simd
20       for (int i=0; i<nsize; i++) {
21          a[i] = 1.0;
22          b[i] = 2.0;
23       }
24
25       for (int k=0; k<ntimes; k++){
26          cpu_timer_start(&tstart);
27          // цикл потоковой триады
28 #pragma omp target teams distribute \
              parallel for simd
29          for (int i=0; i<nsize; i++){
30             c[i] = a[i] + scalar*b[i];
31          }
32          time_sum += cpu_timer_stop(tstart);
33       }
34
35    }
```

Директива участка работы

Директива участка структурированных данных

Участки структурированных данных не могут манипулировать более общими шаблонами программирования. В OpenACC и OpenMP (версии 4.5) были добавлены участки динамических данных, часто именуемые *участками неструктурированных данных*. Форма для директивы содержит выражения `enter` и `exit` с модификатором отображения map для указания операции передачи данных (например, используемыми по умолчанию `to` и `from`):

```
#pragma omp target enter data map([alloc | to]:array[[start]:[length]])
#pragma omp target exit data map([from | release | delete]:
                               array[[start]:[length]])
```

В листинге 11.16 мы конвертируем директиву `omp target data` в директиву `omp target enter data` (строка 13). Область видимости данных на GPU завершается, когда она встречается с директивой `omp target exit data` (строка 36). Действие этих директив аналогично действию участка структурированных данных в листинге 11.15. Но участок динамических данных может использоваться в более сложных сценариях управления данными, таких как конструкторы и деструкторы в C++.

Листинг 11.16 Использование участка динамических данных OpenMP

OpenMP/StreamTriad/StreamTriad_par4.c

```
13 #pragma omp target enter data \
              map(to:a[0:nsize], b[0:nsize], c[0:nsize])
14
15    struct timespec tstart;
16    // инициализация данных и массивов
17    double scalar = 3.0, time_sum = 0.0;
18 #pragma omp target teams distribute \
              parallel for simd
19    for (int i=0; i<nsize; i++) {
20       a[i] = 1.0;
21       b[i] = 2.0;
22    }
23
24    for (int k=0; k<ntimes; k++){
25       cpu_timer_start(&tstart);
26       // цикл потоковой триады
27 #pragma omp target teams distribute \
              parallel for simd
28       for (int i=0; i<nsize; i++){
29          c[i] = a[i] + scalar*b[i];
30       }
31       time_sum += cpu_timer_stop(tstart);
32    }
33
34    printf("Среднее время выполнения цикла потоковой триады составляет %lf msecs\n",
            time_sum/ntimes);
35
36 #pragma omp target exit data \
              map(from:a[0:nsize], b[0:nsize], c[0:nsize])
```

Начинает директиву участка динамических данных

Директива участка работы

Заканчивает директиву участка динамических данных

Мы можем дополнительно оптимизировать передачу данных путем выделения массивов на устройстве и удаления массивов при выходе из участка данных, тем самым устраняя еще одну передачу данных. Когда передача необходима для перемещения данных между CPU и GPU и обратно, можно использовать директиву `omp target update`. Синтаксис указанной директивы таков:

```
#pragma omp target update [to | from] (array[start:length])
```

Мы также должны признать, что в этом примере CPU никогда не использует память под массив. Для памяти, которая существует только на GPU, мы можем выделить ее там, а затем сообщить участкам параллельной работы, что она уже есть. Это делается несколькими способами. Один из них заключается в использовании вызова функции OpenMP для выделения и высвобождения памяти на устройстве. Эти вызовы выглядят следующим образом и требуют включения заголовочного файла OpenMP:

```
#include <omp.h>
double *a = omp_target_alloc(nsize*sizeof(double), omp_get_default_device());
omp_target_free(a, omp_get_default_device());
```

Мы также могли бы использовать процедуры выделения памяти CUDA. Для того чтобы использовать эти процедуры, нам нужно включить заголовочный файл `cuda_runtime`:

```
#include <cuda_runtime.h>
cudaMalloc((void *)&a,nsize*sizeof(double));
cudaFree(a);
```

В директивах параллельной работы нам затем нужно добавить еще одно выражение, связанное с передачей указателей устройств вычислительным ядрам на устройстве:

```
#pragma omp target teams distribute parallel for is_device_ptr(a)
```

Собрав все это вместе, мы получаем изменения в коде, показанные в следующем ниже листинге.

Листинг 11.17 Создание массивов только на GPU

OpenMP/StreamTriad/StreamTriad_par6.c

```
11    double *a = omp_target_alloc(nsize*sizeof(double),
                  omp_get_default_device());
12    double *b = omp_target_alloc(nsize*sizeof(double),
                  omp_get_default_device());
13    double *c = omp_target_alloc(nsize*sizeof(double),
                  omp_get_default_device());
14
15    struct timespec tstart;
16    // инициализация данных и массивов
17    double scalar = 3.0, time_sum = 0.0;
18 #pragma omp target teams distribute
                parallel for simd is_device_ptr(a, b, c)
19    for (int i=0; i<nsize; i++) {
20       a[i] = 1.0;
21       b[i] = 2.0;
22    }
23
24    for (int k=0; k<ntimes; k++){
25       cpu_timer_start(&tstart);
26       // цикл потоковой триады
27 #pragma omp target teams distribute \
                parallel for simd is_device_ptr(a, b, c)
28       for (int i=0; i<nsize; i++){
29          c[i] = a[i] + scalar*b[i];
30       }
31       time_sum += cpu_timer_stop(tstart);
32    }
```

```
33
34    printf("Среднее время выполнения цикла потоковой триады составляет %lf msecs\n",
              time_sum/ntimes);
35
36    omp_target_free(a, omp_get_default_device());
37    omp_target_free(b, omp_get_default_device());
38    omp_target_free(c, omp_get_default_device());
```

В OpenMP есть еще один способ выделения данных на устройстве. В этом методе используется директива `omp declare target`, как показано в листинге 11.18. Сначала мы объявляем указатели на массив в строках 10–12, а затем размещаем их на устройстве следующим ниже блоком кода (строки 14–19). Аналогичный блок используется в строках 42–47 для высвобождения данных на устройстве.

Листинг 11.18 Использование объявления omp для создания массивов только на GPU

OpenMP/StreamTriad/StreamTriad_par8.c

```
10 #pragma omp declare target
11    double *a, *b, *c;
12 #pragma omp end declare target
13
14 #pragma omp target
15    {
16       a = malloc(nsize* sizeof(double));
17       b = malloc(nsize* sizeof(double));
18       c = malloc(nsize* sizeof(double));
19    }
     <неизмененный код>
42 #pragma omp target
43    {
44       free(a);
45       free(b);
46       free(c);
47    }
```

Объявляет цель, создает указатель на устройстве

Выделяет данные на устройстве

Высвобождает данные устройства

Как мы уже видели, существует целый ряд вариантов управления данными GPU. Теперь мы охватили наиболее распространенные директивы и выражения участка данных в OpenMP. Недавние дополнения в стандарт OpenMP позволяют манипулировать более сложными структурами данных и передачами данных.

11.3.4 Оптимизирование OpenMP под GPU-процессоры

Давайте перейдем к стенсильному примеру оптимизации вычислительного ядра, как мы делали для OpenACC. Для ускорения работы отдельных ядер можно попробовать несколько вещей, но по соображениям переносимости в большинстве случаев лучше давать компилятору

самостоятельно выполнять оптимизацию. Стержневая часть стенсильного вычислительного ядра с данными OpenMP и участками работы в следующем ниже листинге является отправной точкой для оптимизационной работы.

Листинг 11.19 Первоначальная OpenMP-версия стенсиля

OpenMP/Stencil/Stencil_par2.c

```
15    double** restrict x    = malloc2D(jmax, imax);
16    double** restrict xnew = malloc2D(jmax, imax);
17
18 #pragma omp target enter data \
                map(to:x[0:jmax][0:imax], \
                    xnew[0:jmax][0:imax])
19
20 #pragma omp target teams
21    {
22 #pragma omp distribute parallel for simd
23       for (int j = 0; j < jmax; j++){
24          for (int i = 0; i < imax; i++){
25             xnew[j][i] = 0.0;
26             x[j][i]    = 5.0;
27          }
28       }
29
30 #pragma omp distribute parallel for simd
31       for (int j = jmax/2 - 5; j < jmax/2 + 5; j++){
32          for (int i = imax/2 - 5; i < imax/2 -1; i++){
33             x[j][i] = 400.0;
34          }
35       }
36    } // omp target teams
37
38    for (int iter = 0; iter < niter; iter+=nburst){
39
40       for (int ib = 0; ib < nburst; ib++){
41          cpu_timer_start(&tstart_cpu);
42 #pragma omp target teams distribute \
                parallel for simd
43          for (int j = 1; j < jmax-1; j++){
44             for (int i = 1; i < imax-1; i++){
45                xnew[j][i]=(x[j][i]+
                      x[j][i-1]+x[j][i+1]+
                      x[j-1][i]+x[j+1][i])/5.0;
46             }
47          }
48
49 #pragma omp target teams distribute \
                parallel for simd
```

Участок данных OpenMP-версии

Директива параллельной работы

Директива параллельной работы

Стенсильное вычислительное ядро

```
50          for (int j = 0; j < jmax; j++){
51             for (int i = 0; i < imax; i++){
52                x[j][i] = xnew[j][i];
53             }
54          }
55          cpu_time += cpu_timer_stop(tstart_cpu);
56
57       }
58
59       printf("Итер %d\n",iter+nburst);
60    }
61
62 #pragma omp target exit data \
                map(from:x[0:jmax][0:imax], \
                         xnew[0:jmax][0:imax])
63
64    free(x);
65    free(xnew);
```

Заменяет обмен данными копированием новых данных обратно в изначальные

Участок данных OpenMP-версии

Просто добавить одну директиву работы для двухмерного цикла и конструкта данных недостаточно, чтобы эффективно сгенерировать работу GPU для версии 16 компилятора IBM XL. Время выполнения почти в два раза больше, чем у серийной версии (см. табл. 11.4 в конце этого раздела). Можно применить `nvprof`, чтобы узнать, на что уходит время. Вот результат:

```
==11376== Profiling application: ./Stencil_par2
==11376== Profiling result:
 Time(%) Time Calls Avg Min Max Name
 51.63% 9.73622s 1000 9.7362ms 9.6602ms 15.378ms __xl_main_l42_OL_3
 48.26% 9.10010s 1000 9.1001ms 9.0323ms 13.588ms __xl_main_l41_OL_2
  0.11% 20.439ms 1 20.439ms 20.439ms 20.439ms __xl_main_l18_OL_1
  0.00% 7.2960us 5 1.4590us 1.2160us 2.1440us [CUDA memcpy DtoH]
  0.00% 5.3760us 2 2.6880us 2.5600us 2.8160us [CUDA memcpy HtoD]

   <продолжение распечатки>
```

Первая строка показывает, что третье вычислительное ядро занимает более 50 % времени выполнения. Копирование обратно в изначальный массив занимает дополнительно 48 % времени выполнения. Проблема кроется в коде вычислительного ядра, а не в передаче данных! В целях его исправления прежде всего следует попробовать свернуть два вложенных цикла в один параллельный конструкт. Соответствующие изменения предусматривают добавление выражения `collapse` вместе с числом циклов для сворачивания в директивах работы. Это показано в строках 22, 30, 42 и 49 в следующем ниже листинге.

Листинг 11.20 Использование выражения collapse для оптимизации

OpenMP/Stencil/Stencil_par3.c

```
20 #pragma omp target teams
21    {
22 #pragma omp distribute parallel \
              for simd collapse(2)
23       for (int j = 0; j < jmax; j++){
24          for (int i = 0; i < imax; i++){
25             xnew[j][i] = 0.0;
26             x[j][i]    = 5.0;
27          }
28       }
29
30 #pragma omp distribute parallel \
              for simd collapse(2)
31       for (int j = jmax/2 - 5; j < jmax/2 + 5; j++){
32          for (int i = imax/2 - 5; i < imax/2 -1; i++){
33             x[j][i] = 400.0;
34          }
35       }
36    }
37
38    for (int iter = 0; iter < niter; iter+=nburst){
39
40       for (int ib = 0; ib < nburst; ib++){
41          cpu_timer_start(&tstart_cpu);
42 #pragma omp target teams distribute \
              parallel for simd collapse(2)
43          for (int j = 1; j < jmax-1; j++){
44             for (int i = 1; i < imax-1; i++){
45                xnew[j][i]=(x[j][i]+x[j][i-1]+x[j][i+1]+
                                 x[j-1][i]+x[j+1][i])/5.0;
46             }
47          }
48
49 #pragma omp target teams distribute \
              parallel for simd collapse(2)
50          for (int j = 0; j < jmax; j++){
51             for (int i = 0; i < imax; i++){
52                x[j][i] = xnew[j][i];
53             }
54          }
55          cpu_time += cpu_timer_stop(tstart_cpu);
56
57       }
58
59       printf("Итер %d\n",iter+nburst);
60    }
```

Добавляет выражение collapse

Время выполнения теперь быстрее, чем на CPU (см. табл.11.3), хотя и не столь быстрое, как версия, созданная компилятором PGI OpenACC

(табл. 11.1). Мы ожидаем, что по мере совершенствования компилятора IBM XL эта ситуация должна улучшиться. Давайте попробуем еще один подход с разбиением директив параллельной работы по двум циклам, как показано в следующем ниже листинге.

Листинг 11.21 Разбиение директив работы для оптимизации

OpenMP/Stencil/Stencil_par4.c

```
20 #pragma omp target teams
21    {
22 #pragma omp distribute
23       for (int j = 0; j < jmax; j++){
24 #pragma omp parallel for simd
25          for (int i = 0; i < imax; i++){
26             xnew[j][i] = 0.0;
27             x[j][i]    = 5.0;
28          }
29       }
30
31 #pragma omp distribute
32       for (int j = jmax/2 - 5; j < jmax/2 + 5; j++){
33 #pragma omp parallel for simd
34          for (int i = imax/2 - 5; i < imax/2 -1; i++){
35             x[j][i] = 400.0;
36          }
37       }
38    }
39
40    for (int iter = 0; iter < niter; iter+=nburst){
41
42       for (int ib = 0; ib < nburst; ib++){
43          cpu_timer_start(&tstart_cpu);
44 #pragma omp target teams distribute
45          for (int j = 1; j < jmax-1; j++){
46 #pragma omp parallel for simd
47             for (int i = 1; i < imax-1; i++){
48                xnew[j][i]=(x[j][i]+x[j][i-1]+x[j][i+1]+
                                    x[j-1][i]+x[j+1][i])/5.0;
49             }
50          }
51
52 #pragma omp target teams distribute
53          for (int j = 0; j < jmax; j++){
54 #pragma omp parallel for simd
55             for (int i = 0; i < imax; i++){
56                x[j][i] = xnew[j][i];
57             }
58          }
59          cpu_time += cpu_timer_stop(tstart_cpu);
60
61       }
```

Разбивает работу на два уровня циклов

```
62
63        printf("Итер %d\n",iter+nburst);
64    }
```

Хронометраж на выходе из компилятора IBM XL для директив разбиения параллельной работы аналогичен выражению `collapse`. В табл. 11.3 приведены результаты наших экспериментов с оптимизациями вычислительного ядра.

Таблица 11.3. Времена выполнения оптимизаций стенсильного вычислительного ядра OpenMP

	Время выполнения стенсильного вычислительного ядра в OpenMP (с)
Последовательный код CPU	5.497
Добавление директивы работы	19.01
Добавление участков вычислений и данных	18.97
Добавление выражения collapse(2)	3.035
Разбиение директив parallel	2.50

В таблице 11.4 мы также посмотрим на распечатку времен выполнения примера потоковой триады на выходе из компилятора IBM XL v16 на процессоре Power 9 с GPU NVIDIA V100. Производительность на CPU отличается, потому что в одном случае мы использовали процессор Intel Skylake, а в данном случае – процессор Power 9. Но отрадно отметить, что производительность потокового ядра с OpenMP на GPU V100 по существу такая же, как и у компилятора PGI OpenACC в табл. 11.2.

Таблица 11.4 Времена выполнения оптимизаций вычислительного ядра потоковой триады OpenMP

	Время выполнения вычислительного ядра потоковой триады в OpenMP (мс)
Последовательный код CPU	15.9
Параллельный 1. Добавлен вычислительный участок	85.7
Параллельный 3. Добавлен участок структурированных данных	0.585
Параллельный 4. Добавлен участок динамических данных	0.584
Параллельный 8. Выделение данных только на устройстве	0.584

Производительность OpenMP с компилятором IBM XL хороша для простой одномерной тестовой задачи, но может быть улучшена для двухмерного стенсильного случая. До сих пор основное внимание уделялось правильному имплементированию стандарта OpenMP для выгрузки на устройства. Мы ожидаем, что производительность будет улучшаться с каждым выпуском компилятора и с увеличением числа поставщиков компиляторов, предлагающих поддержку выгрузки на устройства в рамках OpenMP.

11.3.5 Продвинутый OpenMP для GPU-процессоров

OpenMP обладает рядом дополнительных продвинутых возможностей. OpenMP также меняется, основываясь на опыте ранних имплементаций

на GPU-процессорах и по мере эволюционирования оборудования. Мы рассмотрим лишь некоторые дополнительные директивы и выражения, которые важны для:

- точной настройки вычислительных ядер;
- манипулирования различными важными программными конструктами (функциями, сканированием и совместным доступом к переменным);
- асинхронных операций, которые накладывают друг на друга перемещение данных и их вычисление;
- контроля за размещением памяти;
- манипулирования сложными структурами данных;
- упрощения директив работы.

Управление имплементированными компилятором OpenMP параметрами вычислительного ядра GPU

Мы начнем с рассмотрения выражений, которые можно использовать для точной настройки производительности вычислительного ядра. Мы можем добавлять эти выражения в директивы с целью модифицирования ядер, которые компилятор генерирует для GPU:

- `num_teams` определяет число звеньев, генерируемых директивой `teams`;
- `thread_limit` добавляет число потоков, используемых каждым звеном (team);
- `schedule` или `schedule(static,1)` указывает, что элементы работы распределяются по круговой схеме, а не блоком. Это помогает при коагулировании нагрузки на память на GPU;
- `simdlen` задает длину векторов или потоков для рабочей группы.

Эти выражения бывают полезны в особых ситуациях, но в целом лучше оставлять оптимизирование этих параметров компилятору.

Объявление функции устройства OpenMP

Когда мы вызываем функцию в параллельном участке на устройстве, нам нужно как-то сообщать компилятору о том, что она также должна быть на устройстве. Это делается путем добавления директивы `declare target` в функцию. Ее синтаксис аналогичен синтаксису для объявлений переменных. Вот пример:

```
#pragma omp declare target
int my_compute(<args>){
   <работа>
}
```

Новый редукционный тип scan

Мы обсудили важность алгоритма сканирования в разделе 5.6, где также увидели сложность имплементирования этого алгоритма на GPU. Указанная операция используется в параллельных вычислениях повсемест-

но, и ее сложно писать, поэтому добавление этого типа было бы весьма кстати. Тип `scan` будет доступен в версии 5.0 OpenMP.

```
int run_sum = 0;
#pragma omp parallel for simd reduction(inscan,+: run_sum)
for (int i = 0; i < n; ++i) {
   run_sum += ncells[i];
   #pragma omp scan exclusive(run_sum)
   cell_start[i] = run_sum;
   #pragma omp scan inclusive(run_sum)
   cell_end[i] = run_sum;
}
```

Предотвращение гоночных условий с помощью выражения OpenMP atomic

Ситуация, когда в алгоритме несколько потоков обращается к совместной переменной, является вполне нормальной. Она нередко становится узким местом в выполнении процедур. В различных компиляторах и имплементациях потокообразования эта функциональность представлена *атомиками*. OpenMP тоже предлагает директиву `atomic`. Пример использования указанной директивы приведен ниже.

```
#pragma omp atomic
   i++;
```

OpenMP-версия асинхронных операций

В разделе 10.5 мы обсудили значение накладывающихся друг на друга передач данных и их вычислений с помощью асинхронных операций. OpenMP тоже предлагает свою версию этих операций.

Вы создаете асинхронные операции с устройствами, используя выражение `nowait` в директиве данных или работы. Затем можете использовать выражение `depend`, чтобы указать, что новая операция не может начинаться до завершения предыдущей операции. Эти операции могут быть соединены в цепочку, формируя последовательность операций. Мы можем использовать простую директиву `taskwait` для ожидания завершения всех задач:

```
#pragma omp taskwait
```

Обращение к специальным пространствам памяти

Пропускная способность памяти часто является одним из наиболее важных пределов производительности. В прагма-ориентированных языках не всегда имелась возможность контролировать размещение памяти и ее результирующую пропускную способность. Добавление функциональностей, предоставляющих программисту более высокий контроль над этим, было одним из наиболее ожидаемых дополнений в OpenMP. С OpenMP 5.0 вы сможете нацеливаться на специальные пространства памяти, такие как совместная память и память с высокой пропускной способностью. Эта способность обеспечивается за счет нового модифи-

катора `allocator`. Выражение `allocate` принимает необязательный модификатор следующим образом:

```
allocate([allocator:] list)
```

Следующая ниже пара функций может использоваться для непосредственного выделения и высвобождения памяти:

```
omp_alloc(size_t size, omp_allocator_t *allocator)
omp_free(void *ptr, const omp_allocator_t *allocator)
```

Стандарт OpenMP 5.0 определяет несколько предопределенных пространств памяти для выделителей, как показано в приведенной ниже таблице.

Пространство памяти	Описание типа памяти
`omp_default_mem_alloc/omp_default_mem_space`	Используемое по умолчанию системное пространство хранения
`omp_large_cap_mem_alloc/omp_large_cap_mem_space`	Пространство хранения большой емкости
`omp_const_mem_alloc/omp_const_mem_space`	Хранение для постоянных, неизменных данных
`omp_high_bw_mem_alloc/omp_high_bw_mem_space`	Память с высокой пропускной способностью
`omp_low_lat_mem_alloc/omp_low_lat_mem_space`	Хранение с низкой задержкой

Существует ряд функций для определения новых выделителей памяти. Две главные процедуры таковы:

```
omp_init_allocator
omp_destroy_allocator
```

Эти выделители принимают один из предопределенных аргументов пространства и характеристик выделителя, таких как то, должен ли он быть закреплен, выровнен, приватным, поблизости и многие другие. Имплементации этой способности все еще находятся в стадии разработки. Эта функциональность будет иметь все большее значение в новых архитектурах, где существуют специальные типы памяти с разными по производительности характеристиками задержки и пропускной способности.

Поддержка глубокого копирования для передачи сложных структур данных

В OpenMP 5.0 также добавлен конструкт `declare mapper`, который может делать глубокие копии. *Глубокие копии* дублируют не только структуру данных с указателями, но и данные, на которые указатели ссылаются. Программы со сложными структурами данных и классами столкнулись с трудностями переноса на GPU-процессоры. Возможность делать глубокие копии значительно упрощает эти имплементации.

Упрощение распределения работ с помощью новой директивы loop

Стандарт OpenMP 5.0 вводит более гибкие директивы работы. Одной из них является директива `loop`, которая проще и ближе к функциональ-

ности в OpenACC. Директива `loop` заменяет директиву `distribute parallel for simd`. С помощью директивы `loop` вы сообщаете компилятору, что итерации цикла могут исполняться конкурентно, но фактическая имплементация остается за компилятором. В следующем ниже листинге показан пример использования этой директивы в стенсильном вычислительном ядре.

Листинг 11.22 Использование новой директивы loop в OpenMP 5.0

```
47 #pragma omp target teams  <-- Запускает работу на GPU с несколькими звеньями
48 #pragma omp loop
49          for (int j = 1; j < jmax-1; j++){   <-- Цикл, распараллеленный в качестве независимой работы
50 #pragma omp loop
51             for (int i = 1; i < imax-1; i++){
52                xnew[j][i]=(x[j][i]+x[j][i-1]+x[j][i+1]+
                                  x[j-1][i]+x[j+1][i])/5.0;
53             }
54          }
```

Выражение `loop` на самом деле является выражением `loop independent` или `concurrent`, которое сообщает компилятору о том, что итерации цикла не имеют зависимостей. Выражение `loop` предоставляет компилятору информацию или описательное выражение, а не указывает компилятору, что делать, являясь предписывающим выражением. В большинстве компиляторов эта новая функциональность не имплементирована, поэтому мы продолжаем работать с предписывающими выражениями из предыдущих примеров этой главы. Если вы с этими понятиями не знакомы, то ниже приведено определение каждого из них.

- *Предписывающие директивы и выражения* – директивы программиста, которые указывают компилятору, что конкретно делать.
- *Описательные директивы и выражения* – директивы, которые предоставляют компилятору информацию о следующем конструкте цикла; также дают компилятору некоторую свободу в генерировании наиболее эффективной имплементации.

В спецификациях OpenMP традиционно используются предписывающие выражения. За счет этого сокращаются вариации между имплементациями и улучшается переносимость. Но в случае GPU-процессоров это привело к длинным, сложным директивам с тонкими различиями в возможности синхронизации между потоками и других специфичных для оборудования функциональностей.

Описательный подход ближе к философии OpenACC и не столь ограничен деталями оборудования. Это дает компилятору свободу и ответственность за то, как правильно и эффективно генерировать код для целевого программного обеспечения. Обратите внимание, что для OpenMP это не только значительный сдвиг, но и имеет важное значение. Если OpenMP продолжит пытаться идти по пути предписывающих директив по мере роста сложности оборудования, то язык OpenMP станет слишком сложным, и переносимость исходных кодов будет снижена.

11.4 Материалы для дальнейшего изучения

Как OpenACC, так и OpenMP являются крупными языками с большим числом директив, выражений, модификаций и функций. Кроме стержневой функциональности этих языков, существует мало примеров и скудная документация. И действительно, многие менее часто используемые части способны работать не во всех компиляторах. Вы должны протестировать новую функциональность в небольшом примере и только потом добавлять ее в большое приложение. В целях более глубокого ознакомления с этими языками обратитесь к дополнительным материалам для чтения, которые приведены ниже. Кроме того, обязательно получите практический опыт выполнения упражнений, описанных в разделе 11.4.2.

11.4.1 Дополнительное чтение

Поскольку языки OpenACC и OpenMP все еще эволюционируют, самые лучшие источники дополнительных материалов находятся на соответствующих веб-сайтах: https://openacc.org и https://openmp.org. На каждом веб-сайте перечислены дополнительные ресурсы, включая практические пособия и презентации на ведущих конференциях HPC.

Ресурсы и справочные материалы по OpenACC

OpenACC вышел немного раньше, чем OpenMP, и ему посвящено больше книг и документации. Отправной точкой для указанного языка является стандарт OpenACC. Версия 3.0 стандарта на 150 страницах очень удобочитаема и актуальна для конечного пользователя. Его можно найти на веб-сайте openacc.org. Следующий ниже URL-адрес содержит ссылку на интерфейс прикладного программирования OpenACC, v3.0 (ноябрь 2018 г.):

- https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC.3.0.pdf.

На веб-сайте OpenACC также есть документ по программированию и передовой практике. Он не связан с конкретной версией стандарта и не обновлялся с 2015 года. Руководство по программированию и передовому опыту применения языка OpenACC согласно стандарту OpenACC (июнь 2015 г.) можно найти по адресу:

- https://www.openacc.org/sites/default/files/inline-files/OpenACC_Programming_Guide_0.pdf.

Ведущей книгой по OpenACC является:

- Сунита Чандрасекаран и Гвидо Джакеланд, «OpenACC для программистов: концепции и стратегии» (Sunita Chandrasekaran and Guido Juckeland, *OpenACC for Programmers: Concepts and Strategies* (Addison-Wesley Professional, 2017).

Ресурсы и справочные материалы по OpenMP

Большинство книг и руководств по OpenMP предшествовало возможностям по выгрузке на устройства, но спецификация языка подробно описывает директивы выгрузки на устройства OpenMP. Имея более чем 600 страниц, она скорее представляет собой справочник, чем руководство пользователя. Тем не менее она является обязательным документом для получения подробной информации об особенностях языка[1].

- Совет по пересмотру архитектуры OpenMP, «Интерфейс прикладного программирования OpenMP» (OpenMP Architecture Review Board, *OpenMP Application Programming Interface*, Vol. 5.0, November, 2018) по адресу https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.

Дополнением к спецификации является руководство по примерам использования. В этом руководстве приведены краткие примеры того, как должна работать каждая функция, но не полные примеры на уровне приложений:

- Совет по пересмотру архитектуры OpenMP, «Интерфейс прикладного программирования OpenMP: примеры» (OpenMP Architecture Review Board, *OpenMP Application Programming Interface: Examples*, Vol. 5.0, November, 2019) по адресу https://www.openmp.org/wp-content/uploads/openmp-examples-5.0.0.pdf.

Поскольку в OpenMP все еще наблюдаются значительные изменения, а компиляторы все еще работают над имплементированием функциональностей версии 5.0, неудивительно, что существует мало книг, в которых обсуждаются функциональности выгрузки на устройства. Рууд ван дер Пас и соавт. недавно завершили книгу, в которой рассматриваются новые функциональности OpenMP вплоть до версии v4.5.

- Рууд Ван дер Пас, Эрик Стоцер и Кристиан Тербовен, «Использование OpenMP – следующий шаг: аффинность, ускорители, выполнение операционных задач и SIMD» (Ruud Van der Pas, Eric Stotzer, and Christian Terboven, *Using OpenMP—The Next Step: Affinity, Accelerators*, Tasking, and SIMD (MIT Press, 2017).

11.4.2 Упражнения

1 Найдите компиляторы, которые пригодны для вашей локальной системы GPU. Оба ли компилятора – языка OpenACC и языка OpenMP – доступны в вашем случае? Если нет, то есть ли у вас доступ к каким-либо системам, которые позволили бы вам опробовать эти прагма-ориентированные языки?

---

[1] В целях получения относительного понимания сравнительной сложности этих прагма-ориентированных языков окончательный проект стандарта C18 для языка C составляет чуть более 500 страниц.

2. Выполните примеры потоковой триады из каталогов OpenACC/StreamTriad и/или OpenMP/StreamTriad в локальной системе разработки на основе GPU-процессоров. Вы найдете эти каталоги по адресу https://github.com/EssentialsofParallelComputing/Chapter11.
3. Сравните свои результаты упражнения 2 с результатами BabelStream по адресу https://uob-hpc.github.io/BabelStream/results/. Для потоковой триады перемещенные байты равны `3 * nsize * sizeof(тип данных)`.
4. Модифицируйте отображение участка данных OpenMP в листинге 11.16, чтобы отразить фактическое использование массивов в вычислительных ядрах.
5. Имплементируйте пример массовой суммы из листинга 11.4 в OpenMP.
6. Для массивов `x` и `y` размером 20 000 000 найдите максимальный радиус в массивах, используя как OpenMP, так и OpenACC. Инициализируйте массивы значениями двойной точности, которые линейно увеличиваются с 1.0 до 2.0e7 для массива `x` и уменьшаются с 2.0e7 до 1.0 для массива `y`.

Резюме

- Прагма-ориентированные языки – это самый простой способ переноса работы на GPU-процессоры. Их использование дает вам самый быстрый результат с наименьшими усилиями.
- Процесс переноса заключается в перемещении работы на GPU, а затем в управлении перемещением данных. За счет этого как можно больше работы выполняется на GPU, минимизируя при этом дорогостоящее перемещение данных.
- Оптимизация вычислительного ядра занимает последнее место и в основном должна оставляться компилятору. За счет этого продуцируется наиболее переносимый и перспективный код.
- Следите за последними разработками прагма-ориентированного языка и компиляторов. Указанные компиляторы все еще находятся в стадии быстрой разработки и должны продолжить совершенствоваться.

12 Языки GPU: обращение к основам

Эта глава охватывает следующие ниже темы:

- понимание текущего ландшафта нативных языков GPU;
- создание простых программ GPU на каждом языке;
- манипулирование более сложными операциями с использованием многочисленных вычислительных ядер;
- перенесение исходного кода между различными языками GPU.

Эта глава посвящена языкам более низкого уровня для GPU-процессоров. Мы называем их нативными языками, потому что они напрямую отражают функциональности целевого оборудования GPU. Мы рассмотрим два таких языка, CUDA и OpenCL, которые нашли широкое применение. Мы также рассмотрим HIP, новый вариант для GPU-процессоров AMD. В отличие от прагма-ориентированной имплементации эти языки GPU в меньшей степени зависят от компилятора. Вы должны использовать эти языки для более точного контроля за производительностью вашей программы. Чем эти языки отличаются от языков, представленных в главе 11? По нашему мнению, отличие состоит в том, что эти языки выросли из характеристик оборудования GPU и CPU, тогда как языки OpenACC и OpenMP начинались с высокоуровневых абстракций и опирались на компилятор в том, что касается их соотнесения с разным оборудованием.

Набор нативных языков GPU, CUDA, OpenCL и HIP требует создания отдельного исходного кода для вычислительного ядра GPU. Отдельный исходный код часто похож на исходный код CPU. Трудности, связанные с поддержанием двух разных исходных кодов, являются серьезным препятствием. Если нативный язык GPU поддерживает только один тип оборудования, то бывает, что приходится поддерживать еще больше вариантов исходного кода, если вы хотите работать на GPU нескольких поставщиков. В некоторых приложениях имплементированы свои алгоритмы на нескольких языках GPU и языках CPU. Таким образом, становится понятной острая необходимость в более переносимых языках программирования GPU.

К счастью, переносимость получает все большее внимание в некоторых новых языках GPU. OpenCL был первым языком на основе открытого стандарта, работающим на различном оборудовании GPU- и даже CPU-процессоров. После первоначального всплеска OpenCL не получил такого широкого признания, как первоначально ожидалось. Еще один язык, HIP, разработан AMD как более переносимая версия CUDA, которая генерирует код для GPU-процессоров AMD. В рамках инициативы AMD по обеспечению переносимости включена поддержка GPU-процессоров других производителей.

Разница между этими нативными языками и языками более высокого уровня стирается по мере появления новых языков. Язык SYCL, первоначально представлявший собой слой C++ поверх OpenCL, типичен для этих новых, более переносимых языков. Наряду с языками Kokkos и RAJA SYCL поддерживает единый исходный код как для CPU, так и для GPU. Мы коснемся этих языков в конце главы. На рис. 12.1 показана текущая картина интероперабельности языков GPU, которую мы рассмотрим в этой главе.

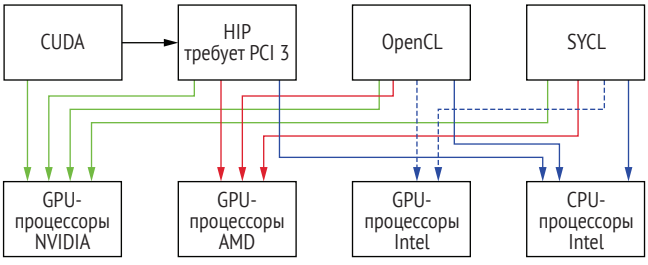


Рис. 12.1 Соотнесение интероперабельности языков GPU показывает все более сложную ситуацию. Вверху показаны четыре языка GPU, а внизу – различные аппаратные устройства. Стрелки показывают маршруты генерации кода от языков к оборудованию. Пунктирные линии предназначены для оборудования, которое все еще находится в разработке

Акцент на интероперабельности языков набирает обороты по мере того, как в крупнейших HPC-инсталляциях появляется все больше разнообразных GPU-процессоров. Ведущие HPC-системы Департамента энергетики, Sierra и Summit, оснащены GPU-процессорами NVIDIA.

В 2021 году в список HPC-систем Департамента энергетики будет добавлена система Aurora в Аргонне с GPU-процессорами Intel и система Frontier в Оук Ридже с GPU-процессорами AMD. С внедрением системы Aurora SYCL вышел из почти безвестности и стал крупным игроком с многочисленными имплементациями. SYCL изначально был разработан для обеспечения более естественного слоя C++ поверх OpenCL. Причиной внезапного появления SYCL стало его принятие на вооружение компанией Intel в качестве части модели программирования oneAPI для GPU-процессоров Intel в системе Aurora. По причине вновь обретенной важности языка SYCL мы рассмотрим SYCL в разделе 12.4. Аналогичный рост интереса наблюдается и к другим языкам и библиотекам, которые обеспечивают переносимость в ландшафте GPU.

Мы заканчиваем главу кратким обзором двух систем обеспечения переносимости производительности, Kokkos и RAJA, которые были созданы для облегчения работы на широком спектре оборудования, от CPU до GPU. Они работают на несколько более высоком уровне абстракции, но обещают единый исходный код, который будет работать везде. Их разработка стала результатом крупных усилий Департамента энергетики по поддержке переносимости крупных научных приложений на более новое оборудование. Цель RAJA и Kokkos состоит в одноразовом переписывании для создания единой базы исходного кода, переносимой и поддерживаемой в период больших изменений в дизайне оборудования.

Наконец, мы хотим предоставить рекомендации о том, как трактовать эту главу. В ней мы охватываем много разных языков за короткое время. Обилие языков отражает нынешнее отсутствие сотрудничества между разработчиками языков, поскольку разработчики преследуют свои непосредственные цели и соображения касательно оборудования. Вместо того чтобы трактовать эти языки как разные языки, думайте о них как о слегка отличающихся диалектах одного или двух языков. Мы рекомендуем вам изучить несколько из этих языков и оценить различия и сходства с другими. Мы будем сравнивать и сопоставлять языки, чтобы помочь вам увидеть, что они не столь уж сильно отличаются, если разобраться с тем или иным синтаксисом каждого из них и их причудами. Мы ожидаем, что языки сольются в более общую форму, потому что текущая ситуация не является устойчивой. Мы уже видим зачатки движения в эту сторону в стремлении к большей переносимости языков, обусловленной потребностями крупных приложений.

12.1 Функциональности нативного языка программирования GPU

Язык программирования GPU должен обладать несколькими базовыми функциональностями. Полезно понимать, какие это функциональности, чтобы иметь возможность их распознавать в каждом языке GPU. Ниже мы кратко описываем необходимые функциональности языка GPU.

- *Обнаружение ускорительного устройства* – язык должен обеспечивать обнаружение ускорителей и способ выбора между этими устройствами. Некоторые языки обеспечивают больший контроль над выбором устройств, чем другие. Даже для такого языка, как CUDA, который просто ищет GPU NVIDIA, должен иметься способ манипулирования многочисленными GPU-процессорами на узле.
- *Поддержка написания вычислительных ядер устройств* – язык должен обеспечивать способ генерирования низкоуровневых команд для GPU-процессоров или других ускорителей. GPU-процессоры обеспечивают почти идентичные базовые операции, как и CPU, поэтому язык вычислительного ядра не должен кардинально отличаться. Вместо того чтобы изобретать новый язык, самый простой подход состоит в задействовании существующих языков программирования и компиляторов для генерирования нового набора команд. Языки GPU сделали это, приняв на вооружение определенную версию языка C или C++ в качестве основы для своей системы. CUDA первоначально был основан на языке программирования C, но теперь основывается на C++ и имеет некоторую поддержку Стандартной библиотеки шаблонов (STL). OpenCL основывается на стандарте C99 и имеет новую спецификацию с поддержкой C++.

  При разработке языка также необходимо принять решение о том, следует ли размещать исходный код хоста и дизайна в одном файле или в разных файлах. В любом случае компилятор должен проводить различие между исходным кодом хоста и дизайна и должен предоставлять способ генерирования набора команд для разного оборудования. Компилятор должен даже принимать решение о времени, когда генерировать набор команд. Например, OpenCL ожидает до тех пор, пока устройство не будет выбрано, а затем генерирует набор команд с помощью компилятора JIT (just-in-time, т. е. «строго вовремя»).
- *Механизм вызова вычислительных ядер устройства с хоста* – прекрасно, теперь у нас есть исходный код устройства, но мы также должны иметь способ вызова этого кода с хоста. Синтаксис выполнения этой операции варьируется в разных языках больше всего. Но указанный механизм лишь немного сложнее, чем вызов стандартной подпрограммы.
- *Манипулирование памятью* – язык должен поддерживать выделение, высвобождение памяти и перемещение данных между хостом и устройством. Самый простой способ состоит в вызове подпрограммы для каждой такой операции. Но еще один подход заключается в том, что компилятор определяет время, когда перемещать данные, и делает это за вас за кулисами. Поскольку это такая важная часть программирования GPU, инновации в этой функциональности продолжают происходить на стороне аппаратного и программного обеспечения.
- *Синхронизация* – должен быть предусмотрен механизм для указания требований к синхронизации между CPU и GPU. Операции

синхронизации также должны быть предусмотрены в вычислительных ядрах.

- *Потоки операций* – полный язык GPU позволяет планировать асинхронные потоки операций наряду с явно заданными зависимостями между вычислительными ядрами и операциями передачи памяти.

Этот список не так уж и страшен. Нативные языки GPU по большей части не столь сильно отличаются от текущего исходного кода CPU. Кроме того, умение распознавать эти общие черты в функциональностях нативного языка GPU помогает вам чувствовать себя комфортно при переходе с одного языка на другой.

12.2 Языки CUDA и HIP GPU: низкоуровневая опция производительности

Мы начнем с рассмотрения двух низкоуровневых языков GPU, CUDA и HIP. Эти два языка программирования GPU-процессоров получили наибольшее распространение.

Архитектура вычислительно-унифицированных устройств (Compute Unified Device Architecture, CUDA) – это нативный язык NVIDIA, который работает только на их GPU-процессорах. Впервые выпущенный в 2008 году, в настоящее время он является доминирующим нативным языком программирования для GPU-процессоров. За десятилетие разработки CUDA стал обладать богатым набором функциональностей и модернизаций, повышающих производительность. Язык CUDA тесно отражает архитектуру GPU NVIDIA. Он не претендует на то, чтобы быть универсальным языком ускорителя. Тем не менее концепции большинства ускорителей достаточно схожи, чтобы дизайн языка CUDA был применим.

GPU-процессоры AMD (ранее ATI) имели ряд недолговечных языков программирования. Они, наконец, остановились на подобии CUDA, которое может генерироваться путем «HIP'ифицирования» кода CUDA с помощью их компилятора HIP. Эта функциональность входит в состав комплекта инструментов ROCm, которые обеспечивают широкую переносимость между языками GPU, включая описанный в разделе 12.3 язык OpenCL для GPU- (и CPU-) процессоров.

12.2.1 Написание и сборка вашего первого приложения на языке CUDA

Мы начнем с ответа на вопрос, как собирать и компилировать простое приложение CUDA, работающее на GPU. Мы будем использовать пример потоковой триады, используемый нами на протяжении всей книги, в котором имплементирован цикл для вот этого вычисления: `C = A + скаляр * B`. Компилятор CUDA разбивает регулярный код C++ для передачи опорному компилятору C++. Затем он компилирует оставшийся код CUDA. Исходный код из этих двух путей связывается вместе в один исполняемый файл.

Для того чтобы проследить указанный выше пример, вам, возможно, сначала потребуется инсталлировать программное обеспечение CUDA[1]. Каждый выпуск CUDA работает с лимитированным диапазоном версий компилятора. Начиная с версии CUDA v10.2, поддерживаются компиляторы GCC вплоть до версии 8. Если вы работаете с несколькими параллельными языками и пакетами, то данная проблема постоянной борьбы с версиями компилятора, возможно, является одной из самых неприятных вещей в CUDA. Но с положительной точки зрения вы можете использовать бо́льшую часть своей регулярной цепочки инструментов и строить системы только с версионными ограничениями и несколькими специальными дополнениями.

Мы покажем три разных подхода, начиная с простого makefile, а затем два разных способа использования CMake. Мы рекомендуем вам сверяться с примерами этой главы, расположенными по адресу https://github.com/EssentialsofParallelComputing/Chapter12.

Вы можете выбрать этот простой makefile для CUDA, скопировав или связав его с Makefile, файловым именем по умолчанию для make. В следующем ниже листинге показан сам файл makefile.

1. Для того чтобы связать с файлом, следует набрать `ln -s Makefile.simple Makefile`.
2. Собрать приложение с помощью `make`.
3. Выполнить приложение с помощью `./StreamTriad`.

Листинг 12.1 Простой makefile для CUDA

CUDA/StreamTriad/Makefile.simple

```
1 all: StreamTriad
2
3 NVCC = nvcc
4 #NVCC_FLAGS = -arch=sm_30
5 #CUDA_LIB = <path>
6 CUDA_LIB=`which nvcc | sed -e 's!/bin/nvcc!!'`/lib
7 CUDA_LIB64=`which nvcc | sed -e 's!/bin/nvcc!!'`/lib64
8
9 %.o : %.cu
10   ${NVCC} ${NVCC_FLAGS} -c $< -o $@
11
12 StreamTriad: StreamTriad.o timer.o
13   ${CXX} -o $@ $^ -L${CUDA_LIB} -lcudart
14
15 clean:
16   rm -rf StreamTriad *.o
```

Задает компилятор NVIDIA CUDA

Здесь может потребоваться указать путь к библиотеке и тип архитектуры GPU

Неявное правило для компилирования исходных файлов CUDA

Строка связывания/компоновки для приложений CUDA

Ключевым дополнением является шаблонное правило в строках 9-10, которое конвертирует файл с суффиксом .cu в объектный файл. Для этой

[1] Подробные сведения см. в руководстве по инсталляции CUDA (https://docs.nvidia.com/cuda/cuda-installation-guide-linux/).

операции мы используем компилятор NVIDIA NVCC. Затем нам нужно добавить библиотеку среды выполнения CUDA, CUDART, в строку связывания. Вы можете использовать строки 4 и 5 для указания конкретной архитектуры GPU NVIDIA и специального пути к библиотекам CUDA.

> **ОПРЕДЕЛЕНИЕ** *Шаблонное правило* – это описание, которое предназначено для утилиты make, задающее общее правило конвертирования любого файла с одним суффиксальным шаблоном в файл с другим суффискальным шаблоном.

CUDA имеет обширную поддержку в сборочной системе CMake. Далее мы рассмотрим поддержку старого стиля и новый современный подход CMake, который появился совсем недавно. В листинге 12.2 мы показываем метод старого образца. Его преимущество заключается в большей переносимости для систем со старыми версиями CMake и автоматическом обнаружении архитектуры GPU NVIDIA. Эта последняя функциональность обнаружения аппаратного устройства настолько удобна, что в настоящее время рекомендуется использовать CMake старого образца. В целях применения этой сборочной системы следует связать CMakeLists_old.txt с CMakeLists.txt:

```
ln -s CMakeLists_old.txt CMakeLists.txt
mkdir build && cd build
cmake ..
make
```

Листинг 12.2 Файл CUDA CMake старого образца

CUDA/StreamTriad/CMakeLists_old.txt

```
 1 cmake_minimum_required (VERSION 2.8)
 2 project (StreamTriad)
 3
 4 find_package(CUDA REQUIRED)
 5
 6 set (CMAKE_CXX_STANDARD 11)
 7 set (CMAKE_CUDA_STANDARD 11)
 8
 9 # задает CMAKE_{C,CXX}_FLAGS из компиляторных флагов CUDA.
   # Встраивает DEBUG и RELEASE
10 set (CUDA_PROPAGATE_HOST_FLAGS ON) # по умолчанию включено
11 set (CUDA_SEPARABLE_COMPILATION ON)
12
13 if (CMAKE_VERSION VERSION_GREATER "3.9.0")
14    cuda_select_nvcc_arch_flags(ARCH_FLAGS)
15 endif()
16
17 set (CUDA_NVCC_FLAGS ${CUDA_NVCC_FLAGS}
        -O3 ${ARCH_FLAGS})
18
```

Для поддержки CUDA вам требуется CMake минимум версии v2.8

Традиционный модуль CMake устанавливает компиляторные флаги

Установить равным «включено» для вызова функций в других блоках компиляции (по умолчанию выключено)

Обнаруживает и устанавливает надлежащий архитектурный флаг для текущего GPU NVIDIA

Устанавливает компиляторные флаги для компилятора NVIDIA

```
19 # Добавляет сборочную цель StreamTriad с файлами исходного кода
20 cuda_add_executable(StreamTriad
      StreamTriad.cu timer.c timer.h)
21
22 if (APPLE)
23    set_property(TARGET StreamTriad PROPERTY BUILD_RPATH
        ${CMAKE_CUDA_IMPLICIT_LINK_DIRECTORIES})
24 endif (APPLE)
25
26 # Очистка
27 add_custom_target(distclean COMMAND rm -rf CMakeCache.txt CMakeFiles
28                   Makefile cmake_install.cmake
                      StreamTriad.dSYM ipo_out.optrpt)
29
30 # Добавляет сборочную цель clean_cuda_depends
   # -- вызвать с "make clean_cuda_depends"
31 CUDA_BUILD_CLEAN_TARGET()
```

Устанавливает надлежащие флаги сборки и связывания для исполняемого файла CUDA

Большая часть сборочной системы CMake остается стандартной. Атрибут разделяемой компиляции в строке 11 предлагается для более устойчивой сборочной системы, ориентированной на общую разработку. Его можно затем отключить на более позднем этапе, чтобы сэкономить несколько регистров в вычислительных ядрах CUDA, получив небольшую оптимизацию в сгенерированном коде. Используемые по умолчанию значения CUDA предназначены для обеспечения производительности, а не для более общей и устойчивой сборки. Автоматическое обнаружение архитектуры GPU NVIDIA в строке 14 является значительным удобством, которое избавляет вас от необходимости модифицировать файл makefile вручную.

С версией 3.0 сборочная система CMake претерпевает довольно серьезный пересмотр своей структуры, который, что называется, «осовременивает» CMake. Ключевыми атрибутами этого стиля являются более интегрированная система и поцелевое применение атрибутов. Это нигде не проявляется так явно, как в ее поддержке CUDA. Давайте взглянем на листинг 12.3, чтобы узнать, как ее использовать. В целях применения этой сборочной системы для поддержки CUDA в современном, новом стиле CMake необходимо связать CMakeLists_new.txt с CMakeLists.txt:

```
ln -s CMakeLists_new.txt CMakeLists.txt
mkdir build && cd build
cmake ..
make
```

Листинг 12.3 Файл CMake в новом (осовремененном) стиле для CUDA

CUDA/StreamTriad/CMakeLists_new.txt

```
 1 cmake_minimum_required (VERSION 3.8)
 2 project (StreamTriad)
 3
 4 enable_language(CXX CUDA)
```

Требует CMake v3.8

Задействует CUDA как язык

```
 5
 6 set (CMAKE_CXX_STANDARD 11)
 7 set (CMAKE_CUDA_STANDARD 11)
 8
 9 #set (ARCH_FLAGS -arch=sm_30)
10 set (CMAKE_CUDA_FLAGS ${CMAKE_CUDA_FLAGS};
       "-O3 ${ARCH_FLAGS}")
11
12 # Добавляет сборочную цель StreamTriad с файлами исходного кода
13 add_executable(StreamTriad StreamTriad.cu timer.c timer.h)
14
15 set_target_properties(StreamTriad PROPERTIES
     CUDA_SEPARABLE_COMPILATION ON)
16
17 if (APPLE)
18     set_property(TARGET StreamTriad PROPERTY BUILD_RPATH
          ${CMAKE_CUDA_IMPLICIT_LINK_DIRECTORIES})
19 endif(APPLE)
20
21 # Очистка
22 add_custom_target(distclean COMMAND rm -rf CMakeCache.txt CMakeFiles
23                   Makefile cmake_install.cmake
                    StreamTriad.dSYM ipo_out.optrpt)
```

Задает архитектуру CUDA вручную

Устанавливает компиляторные флаги для CUDA

Устанавливает флаг разделяемой компиляции

Первое, что следует отметить при использовании этого современного подхода CMake, – это то, насколько он стал проще, чем в старом стиле. Ключевым моментом является задействование CUDA в качестве языка в строке 4. С этого места необходимо проделать небольшую дополнительную работу.

Как показано в строках 9-10, мы можем установить флаги для компилирования конкретной архитектуры GPU. Однако в осовремененной сборочной системе CMake у нас пока нет автоматического способа обнаружения архитектуры. Без архитектурного флага компилятор генерирует код и оптимизирует под GPU-устройство sm_30. Сгенерированный исходный код sm_30 работает на любом устройстве от Kepler K40 либо более позднем, но он не будет оптимизирован под новейшие архитектуры. Кроме того, можно указать несколько архитектур в одном компиляторе. Компиляции будут делаться медленнее, а сгенерированный исполняемый файл будет больше.

Мы также можем установить отдельный атрибут компиляции для CUDA, но в другом синтаксисе, в котором он применяется к конкретной цели. Флаг оптимизации в строке 10, `-O3`, отправляется компилятору хоста только для обычного исходного кода C++. Для исходного кода CUDA по умолчанию используется уровень оптимизации `-O3`, который редко нуждается в модифицировании.

В целом процесс сборки программы CUDA является простым и становится все проще. Однако следует ожидать, что изменения в сборке продолжатся. Clang добавляет нативную поддержку для компилирования исходного кода CUDA, предоставляя вам еще одну опцию, помимо

компилятора NVIDIA. Теперь давайте перейдем к исходному коду. Мы начнем с вычислительного ядра для GPU в следующем ниже листинге.

Листинг 12.4 CUDA-версия потоковой триады: вычислительное ядро

CUDA/StreamTriad/StreamTriad.cu

```
 2 __global__ void StreamTriad(
 3                const int n,
 4                const double scalar,
 5                const double *a,
 6                const double *b,
 7                      double *c)
 8 {
 9    int i = blockIdx.x*blockDim.x+threadIdx.x;
10
11    // Защитить от выхода за пределы границ
12    if (i >= n) return;
13
14    c[i] = a[i] + scalar*b[i];
15 }
```

Получает индекс ячейки

Защищает от выхода за пределы границ

Тело потоковой триады

Как это обычно бывает с вычислительными ядрами GPU, мы обнажаем цикл for из вычислительного блока. В результате остается тело цикла в строке 14. Нам нужно добавить условный блок в строке 12, чтобы предотвратить доступ к данным, выходящим за пределы границ. Без этой защиты вычислительные ядра могут случайно отказать без выдачи сообщения. А затем, в строке 9, мы получаем глобальный индекс из блочной и поточной переменных, заданных средой выполнения CUDA. Добавление атрибута `__global__` в подпрограмму сообщает компилятору о том, что это вычислительное ядро GPU будет вызываться с хоста. Тем временем на стороне хоста мы должны настроить память и выполнить вызов ядра. Указанный процесс показан в следующем ниже листинге.

Листинг 12.5 CUDA-версия потоковой триады: настройка и демонтаж

CUDA/StreamTriad/StreamTriad.cu

```
31    // выделить память хоста и инициализировать
32    double *a = (double *)malloc(
                  stream_array_size*sizeof(double));
33    double *b = (double *)malloc(
                  stream_array_size*sizeof(double));
34    double *c = (double *)malloc(
                  stream_array_size*sizeof(double));
35
36    for (int i=0; i<stream_array_size; i++) {
37       a[i] = 1.0;
38       b[i] = 2.0;
39    }
40
```

Выделяет память хоста

Инициализирует массивы

```
41    // выделить память устройства.
      // суффикс _d обозначает указатель устройства
42    double *a_d, *b_d, *c_d;
43    cudaMalloc(&a_d, stream_array_size*
                       sizeof(double));
44    cudaMalloc(&b_d, stream_array_size*
                       sizeof(double));
45    cudaMalloc(&c_d, stream_array_size*
                       sizeof(double));
46
47    // установка размера блока и дополнение суммарного размера решетки,
      // чтобы получить четный размер блока
48    int blocksize = 512;
49    int gridsize =
         (stream_array_size + blocksize - 1)/
          blocksize;
50
      < ... хронометраж цикла ... показанный ниже код в листинге 12.6 >
78    printf("Среднее время выполнения составляет %lf мс, передача данных - %lf мс\n",
79           tkernel_sum/NTIMES, (ttotal_sum - tkernel_sum)/NTIMES);
80
81    cudaFree(a_d);
82    cudaFree(b_d);
83    cudaFree(c_d);
84
85    free(a);
86    free(b);
87    free(c);
88 }
```

Выделяет память устройства

Устанавливает размер блока и вычисляет число блоков

Высвобождает память устройства

Высвобождает память хоста

Сначала мы выделяем память на хосте и инициализируем ее в строках 31–39. Нам также нужно соответствующее пространство памяти на GPU для хранения массивов, пока с ними работает GPU. Для этого мы используем процедуру `cudaMalloc` в строках 43–45. Теперь мы подошли к нескольким интересным строкам (47–49), которые необходимы исключительно для GPU. Размер блока – это размер рабочей группы на GPU. Он определяется размером плитки, размером блока либо размером рабочей группы в зависимости от используемого языка программирования GPU (см. таб. 10.1). Следующая строка, которая вычисляет размер решетки, характерна для кода GPU. У нас не всегда будет размер массива, который является четным целым числом, кратным размеру блока. Значит, нам нужно иметь целое число, которое равно или больше дробного числа блоков. Давайте пройдемся по примеру пошагово, чтобы понять, что происходит.

Теперь все блоки, кроме последнего, имеют 512 значений. Последний блок будет размером 512, но будет содержать только 488 элементов данных. Проверка на выход «за пределы границ» в строке 12 листинга 12.4 предотвращает возникновение проблем с этим частично заполненным блоком. Последние несколько строк в листинге 12.5 высвобождают указатели устройства и указатели хоста. Не следует забывать использовать

`cudaFree` для указателей устройства и функцию библиотеки C, `free`, для указателей хоста.

Пример: вычисление размера блока для GPU

В строке 3 в следующем ниже листинге мы вычисляем дробное число блоков. В этом примере с размером массива 1000 оно равно 1.95 блока. Вместо того чтобы усекать его до 1, что произошло бы по умолчанию при применении целочисленной арифметики, нам нужно округлить вверх до 2. Если бы мы просто рассчитали размер массива, деленный на размер блока, то мы получили бы целочисленное усечение. Поэтому мы должны выполнить приведение типа для каждого из них к значению с плавающей точкой, чтобы получить деление с плавающей точкой. На самом деле нам нужно выполнить приведение только для одного из значений, а стандарт C/C++ потребует от компилятора, чтобы тот предоставил другие элементы. Но в наших соглашениях о программировании конверсия типов должна предписываться в явной форме, иначе это будет ошибкой программирования. Компиляторы часто не сигнализируют об этих случаях, но они могут маскировать непреднамеренные ситуации.

Функция `ceil`, используемая в строках 4 и 5 листинга, округляет вверх до следующего целого значения, которое равно или больше числа с плавающей точкой. Мы можем получить тот же результат с помощью целочисленной арифметики, прибавив размер блока минус единица, а затем выполнив целочисленное деление с усечением, как это делается в строке 6. Мы решили использовать эту версию, потому что целочисленная форма не требует никаких операций с плавающей точкой и должна быть быстрее.

```
1 int stream_array_size = 1000
2 int blocksize = 512
3 float frac_blocks = (float)stream_array_size/(float)blocksize;
>>>frac_blocks = 1.95
4 int nblocks = ceil(frac_blocks);
>>> nblocks = 2
```

либо

```
5 int nblocks = ceil((float)stream_array_size/(float)blocksize);
```

либо

```
6 int nblocks = (stream_array_size + blocksize - 1)/blocksize;
```

Нам осталось лишь скопировать память на GPU, вызвать вычислительное ядро GPU и скопировать память обратно. Мы делаем это в хронометражном цикле (в листинге 12.6), который может исполняться несколько раз, чтобы получить более оптимальный результат измерений. Иногда первый вызов GPU будет выполняться медленнее из-за затрат на инициализацию. Мы можем его амортизировать, выполняя несколько итераций. Если этого недостаточно, то вы также можете отбросить хронометраж с первой итерации.

Листинг 12.6 CUDA-версия потоковой триады: вызов вычислительного ядра и хронометражный цикл

CUDA/StreamTriad/StreamTriad.cu

```
51 for (int k=0; k<NTIMES; k++){
52    cpu_timer_start(&ttotal);
53    cudaMemcpy(a_d, a, stream_array_size*
         sizeof(double), cudaMemcpyHostToDevice);
54    cudaMemcpy(b_d, b, stream_array_size*
         sizeof(double), cudaMemcpyHostToDevice);
55    // копирование памяти cudaMemcpy на устройство возвращается,
      // когда есть буфер
56    cudaDeviceSynchronize();
57
58    cpu_timer_start(&tkernel);
59    StreamTriad<<<gridsize, blocksize>>>
         (stream_array_size, scalar, a_d, b_d, c_d);
60    cudaDeviceSynchronize();
61    tkernel_sum += cpu_timer_stop(tkernel);
62
63    // копирование памяти cudaMemcpy с устройства на хост блокирует
      // с целью завершения, поэтому синхронизации не требуется
64    cudaMemcpy(c, c_d, stream_array_size*
         sizeof(double), cudaMemcpyDeviceToHost);
65    ttotal_sum += cpu_timer_stop(ttotal);
66    // проверить результаты и напечатать ошибки, если они есть.
      // ограничить только 10 ошибками на итерацию
67    for (int i=0, icount=0; i<stream_array_size && icount < 10; i++){
68       if (c[i] != 1.0 + 3.0*2.0) {
69          printf("Ошибка с результатом c[%d]=%lf на итер %d\n",i,c[i],k);
70          icount++;
71       } // если не правильно, то напечатать ошибку
72    } // цикл проверки результата
73 } // хронометраж цикла
```

Копирует данные массива с хоста на устройство

Синхронизирует, чтобы получить точный хронометраж только для ядра

Запускает ядро StreamTriad

Форсирует завершение, чтобы получить хронометраж

Копирует данные массива обратно с устройства на хост

Шаблон в хронометражном цикле состоит из следующих ниже шагов.

1. Скопировать данные на GPU (строки 53-54).
2. Вызвать вычислительное ядро GPU для оперирования на массивах (строка 59).
3. Скопировать данные обратно (строка 64).

Мы добавляем несколько вызовов синхронизации и таймера, чтобы получить точный результат измерений вычислительного ядра GPU. В конце цикла мы затем проверяем правильность результата. При внедрении этого исходного кода в производство мы сможем удалить хронометраж, синхронизацию и проверку ошибок. Вызов вычислительного ядра GPU можно легко определить по трем шевронам, или угловым скобкам. Если проигнорировать шевроны и содержащиеся в них переменные, то строка кода имеет типичный синтаксис вызова подпрограммы C:

```
StreamTriad(stream_array_size, scalar, a_d, b_d, c_d);
```

Значения в круглых скобках являются аргументами, которые должны быть переданы вычислительному ядру GPU. Например:

```
<<<размер решетки, размер блока>>>
```

Итак, какие аргументы содержатся в шевронах? Это аргументы компилятору CUDA о том, как разбивать задачу на блоки для GPU. Более того, в строках 48–49 листинга 12.2 мы установили размер блока и рассчитали число блоков, или размер решетки, чтобы уместить все данные в массиве. Здесь аргументы являются одномерными. Мы также можем иметь двух- или трехмерные массивы, объявив и установив эти аргументы в качестве значений для матрицы N×N.

```
dim3 blocksize(16,16); dim3 blocksize(8,8,8);
dim3 gridsize( (N + blocksize.x - 1)/blocksize.x,
               (N + blocksize.y - 1)/blocksize.y );
```

Мы можем ускорить передачу памяти, исключив копирование данных. Это возможно благодаря более глубокому пониманию того, как функционирует операционная система. Память, передаваемая по сети, должна находиться в фиксированном месте, которое нельзя перемещать во время операции. Обычные выделения памяти помещаются в *листаемую память* или в память, которую можно перемещать по требованию. При передаче памяти сначала необходимо переместить данные в *закрепленную память* или в память, которую невозможно перемещать. Мы впервые увидели использование закрепленной памяти в разделе 9.4.2 при сравнительном анализе перемещения памяти по шине PCI. Мы можем устранить копирование памяти, выделив наши массивы не в листаемой памяти, а в закрепленной. На рис. 9.8 показана разница в производительности, которую мы могли бы получить. Теперь возникает вопрос, как это можно сделать?

CUDA предоставляет нам вызов функции `cudaHostMalloc`, которая делает это за нас. Указанная функция является прямой заменой обычным системным подпрограммам `malloc` с небольшим изменением в аргументах, где указатель возвращается в качестве аргумента, как показано ниже:

```
double *x_host = (double *)malloc(stream_array_size*sizeof(double));
cudaMallocHost((void**)&x_host, stream_array_size*sizeof(double));
```

Есть ли недостаток в использовании закрепленной памяти? Дело в том, что если вы используете много закрепленной памяти, то будет отсутствовать место для внесения в память еще одного приложения. Выемка из памяти одного приложения и внесение другого – это огромное удобство для пользователей. Указанный процесс называется листанием памяти.

ОПРЕДЕЛЕНИЕ *Листание памяти* (memory paging) в многопользовательских мультиприкладных операционных системах – это процесс временного перемещения страниц памяти на диск, чтобы дать возможность состояться еще одному процессу.

Листание памяти является важным достижением в операционных системах, позволяющим создавать впечатление, что у вас больше памяти, чем на самом деле. Например, это позволяет вам временно запускать Excel во время работы в Word и не закрывать исходное приложение. Это делается путем записи ваших данных на диск, а затем чтения их обратно, когда вы возвращаетесь в Word. Но данная операция обходится дорого, поэтому в высокопроизводительных вычислениях мы избегаем листание памяти из-за серьезного снижения производительности, которое оно влечет за собой. В некоторых гетерогенных вычислительных системах, с CPU и с GPU, имплементирована унифицированная память.

ОПРЕДЕЛЕНИЕ *Унифицированная память* – это память, которая выглядит как единое адресное пространство как для CPU, так и для GPU.

На настоящий момент вы уже видели, что манипулирование отдельными пространствами памяти на CPU и GPU значительно усложняет написание кода GPU. Благодаря унифицированной памяти система выполнения программ GPU будет справляться с этим за вас. По-прежнему могут существовать два отдельных массива, но данные перемещаются автоматически. На интегрированных GPU существует вероятность того, что память вообще не нужно перемещать. Тем не менее желательно писать свои программы с явным копированием памяти, чтобы ваши программы можно было переносить в системы без унифицированной памяти. Копирование памяти пропускается, если она не нужна в архитектуре.

12.2.2 Редукционное вычислительное ядро в CUDA: жизнь становится все сложнее

Когда нам требуется кооперация между потоками GPU, в низкоуровневых, нативных языках GPU все усложняется. Мы рассмотрим простой пример суммирования, чтобы понять, как с этим справиться. Пример требует двух отдельных вычислительных ядер CUDA и показан в листингах 12.7–12.10. В следующем ниже листинге показан первое прохождение, в котором мы суммируем значения внутри поточного блока и сохраняем результат обратно в редукционный блокнотный массив, redscratch.

Листинг 12.7 Первое прохождение операции редукции с использованием суммирования

CUDA/SumReduction/SumReduction.cu (four parts)

```
23 __global__ void reduce_sum_stage1of2(
24                   const int isize,          // 0 Суммарное число яеек.
25                         double *array,      // 1
26                         double *blocksum,   // 2
27                         double *redscratch) // 3
28 {
29    extern __shared__ double spad[];
```

Блокнотный массив в совместной памяти CUDA

```
30    const unsigned int giX = blockIdx.x*blockDim.x+threadIdx.x;
31    const unsigned int tiX = threadIdx.x;
32
33    const unsigned int group_id = blockIdx.x;
34
35    spad[tiX] = 0.0;
36    if (giX < isize) {
37       spad[tiX] = array[giX];
38    }
39
40    __syncthreads();
41
42    reduction_sum_within_block(spad);
43
44    // Записать локальное значение назад в массив
      // размера, равного числу групп
45    if (tiX == 0){
46       redscratch[group_id] = spad[0];
47        (*blocksum) = spad[0];
48    }
49 }
```

Загружает память в блокнотный массив

Синхронизирует потоки перед использованием блокнотных данных

Задает редукцию внутри поточного блока

Один поток хранит результат блока

Мы начинаем первое прохождение с того, что все потоки хранят свои данные в блокнотном массиве в совместной памяти CUDA (строки 35–38). Все потоки в блоке могут обращаться к этой совместной памяти. Доступ к совместной памяти можно получать за один или два процессорных цикла вместо сотен, необходимых для основной памяти GPU. Совместную память можно трактовать как программируемый кеш или как блокнотную память. В целях обеспечения того, чтобы все потоки завершали хранение, мы используем вызов синхронизации в строке 40.

Поскольку редукционная сумма внутри блока будет использоваться в обоих редукционных прохождениях, мы помещаем исходный код в подпрограмму устройства и вызываем его в строке 42. *Подпрограмма устройства* – это подпрограмма, которая должна вызываться не с хоста, а из другой подпрограммы устройства. После выполнения подпрограммы полученная сумма сохраняется обратно в меньший массив данных, который мы читаем во время второй фазы. Мы также сохраняем результат в строке 47 на случай, если второе прохождение может быть пропущено. Поскольку мы не можем обращаться к значениям в других поточных блоках, мы должны завершить операцию за второе прохождение в еще одном вычислительном ядре. В этом первом прохождении мы сократили длину данных на размер нашего блока.

Давайте перейдем к рассмотрению общего исходного кода устройства, о котором мы упоминали в первом прохождении. Нам потребуется редукция суммированием для поточного блока CUDA на обоих прохождениях, поэтому мы пишем ее как общую процедуру устройства. Исходный код, показанный в следующем ниже листинге, можно легко модифицировать и для других операторов редукции, и он требует лишь небольших изменений с учетом HIP и OpenCL.

Листинг 12.8 Общее вычислительное ядро устройства для редукции путем суммирования

CUDA/SumReduction/SumReduction.cu (four parts)

```
 1 #define MIN_REDUCE_SYNC_SIZE warpSize
 2
 3 __device__ void reduction_sum_within_block(double *spad)
 4 {
 5    const unsigned int tiX = threadIdx.x;
 6    const unsigned int ntX = blockDim.x;
 7
 8    for (int offset = ntX >> 1; offset > MIN_REDUCE_SYNC_SIZE;
           offset >>= 1) {
 9       if (tiX < offset) {
10          spad[tiX] = spad[tiX] + spad[tiX+offset];
11       }
12       __syncthreads();
13    }
14    if (tiX < MIN_REDUCE_SYNC_SIZE) {
15       for (int offset = MIN_REDUCE_SYNC_SIZE; offset > 1; offset >>= 1) {
16          spad[tiX] = spad[tiX] + spad[tiX+offset];
17          __syncthreads();
18       }
19       spad[tiX] = spad[tiX] + spad[tiX+1];
20    }
21 }
```

CUDA определяет warpSize равным 32

Использовать необходимые потоки только при превышении размера варпа

Синхронизирует между каждым уровнем прохождения

Общая процедура устройства, которая будет вызываться из обоих прохождений, определена в строке 3. Она выполняет редукцию путем суммирования в поточном блоке. Атрибут `__device__` перед процедурой указывает на то, что она будет вызываться из вычислительного ядра GPU. Базовая концепция процедуры основана на дереве попарной редукции в $O(\log n)$ операциях, как показано на рис. 12.2. Базовое дерево редукции из рисунка представлено исходным кодом в строках 15–18. Мы вносим несколько незначительных модификаций, когда рабочий набор превышает размер варпа в строках 8–13 и для уровня завершающего прохождения в строке 19 во избежание ненужной синхронизации.

Та же концепция парной редукции используется для полно-поточного блока, который может достигать размера 1024 на большинстве устройств GPU, хотя чаще используется от 128 до 256. Но что делать, если размер вашего массива превышает 1024? Мы добавляем второе прохождение, в котором используется только один поточный блок, как показано в следующем ниже листинге.

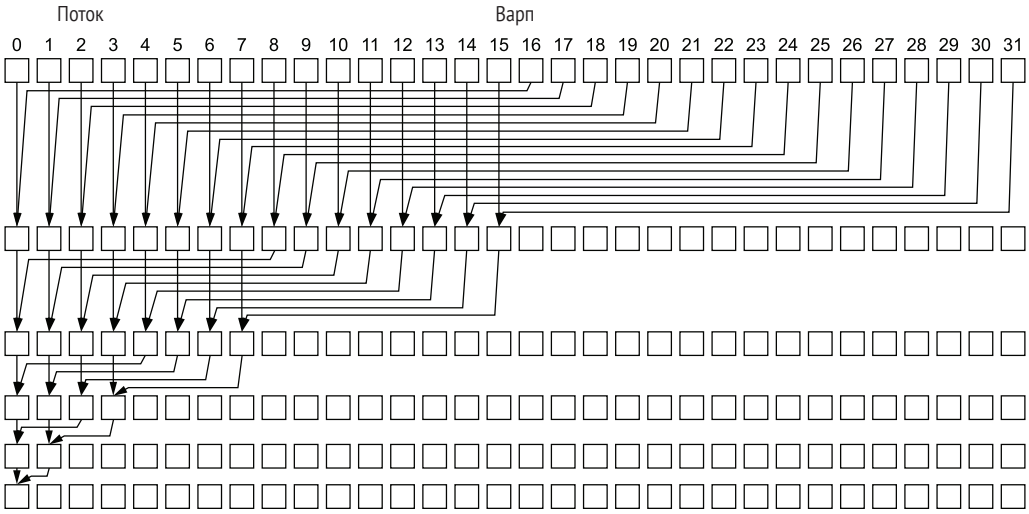


Рис. 12.2 Дерево попарной редукции для варпа, которое суммирует значения за log n шагов

Листинг 12.9 Второе прохождение операции редукции

CUDA/SumReduction/SumReduction.cu (four parts)

```
51 __global__ void reduce_sum_stage2of2(
52                  const int isize,
53                        double *total_sum,
54                        double *redscratch)
55 {
56    extern __shared__ double spad[];
57    const unsigned int tiX = threadIdx.x;
58    const unsigned int ntX = blockDim.x;
59
60    int giX = tiX;
61
62    spad[tiX] = 0.0;
63
64    // загрузить сумму из блокнотного массива редукции, redscratch
65    if (tiX < isize) spad[tiX] = redscratch[giX];
66
67    for (giX += ntX; giX < isize; giX += ntX) {
68       spad[tiX] += redscratch[giX];
69    }
70
71    __syncthreads();
72
73    reduction_sum_within_block(spad);
74
75    if (tiX == 0) {
76      (*total_sum) = spad[0];
77    }
78 }
```

Загружает значения в блокнотный массив

Прокручивать в цикле с приращением по размеру поточного блока для получения всех данных

Синхронизирует при заполнении блокнотного массива

Вызывает нашу общую блочную процедуру редукции

Один поток устанавливает итоговую сумму для возвращения

Во избежание более двух ядер для более крупных массивов мы используем один поточный блок и цикл в строках 67–69 для чтения и суммирования любых дополнительных данных в совместный блокнотный массив. Мы используем однопоточный блок, потому что внутри него можно выполнять синхронизацию, избегая необходимости в еще одном вызове вычислительного ядра. Если мы используем размеры поточных блоков, равные 128, и имеем массив из миллиона элементов, то цикл будет суммировать около 60 значений в каждом местоположении в совместной памяти ($1\,000\,000/128^2$). Размер массива сокращается на 128 при первом прохождении, и затем мы суммируем в блокнот размером 128, получая деление на 128 в квадрате. Если мы используем блоки большего размера, например 1024, то мы могли бы сократить цикл с 60 итераций до одного чтения. Теперь мы просто вызываем ту же общую редукцию поточного блока, которую использовали раньше. Результатом будет первое значение в блокнотном массиве. Последняя часть редукции состоит в настройке и вызове этих двух вычислительных ядер с хоста. Мы увидим, как это делается, в следующем ниже листинге.

Листинг 12.10 Исходный код хоста для CUDA-редукции

```
CUDA/SumReduction/SumReduction.cu (four parts)
100 size_t blocksize = 128;
101 size_t blocksizebytes = blocksize*
                            sizeof(double);
102 size_t global_work_size = ((nsize + blocksize - 1) /blocksize) *
                              blocksize;
103 size_t gridsize = global_work_size/blocksize;
104
105 double *dev_x, *dev_total_sum, *dev_redscratch;
106 cudaMalloc(&dev_x, nsize*sizeof(double));
107 cudaMalloc(&dev_total_sum, 1*sizeof(double));
108 cudaMalloc(&dev_redscratch,
               gridsize*sizeof(double));
109
110 cudaMemcpy(dev_x, x, nsize*sizeof(double),
               cudaMemcpyHostToDevice);
111
112 reduce_sum_stage1of2
          <<<gridsize, blocksize, blocksizebytes>>>
          (nsize, dev_x, dev_total_sum,
           dev_redscratch);
113
114 if (gridsize > 1) {
115    reduce_sum_stage2of2
          <<<1, blocksize, blocksizebytes>>>
          (nsize, dev_total_sum, dev_redscratch);
116 }
117
118 double total_sum;
```

Вычисляет размеры блока и решетки для вычислительных ядер CUDA

Выделяет память устройства для ядра

Копирует массив на устройство GPU

Вызывает первое прохождение редукционного ядра

При необходимости вызывает второе прохождение

```
119 cudaMemcpy(&total_sum, dev_total_sum, 1*sizeof(double),
              cudaMemcpyDeviceToHost);
120 printf("Результат -- итоговая сумма %lf \n",total_sum);
121
122 cudaFree(dev_redscratch);
123 cudaFree(dev_total_sum);
124 cudaFree(dev_x);
```

Исходный код хоста сначала вычисляет размеры для вызовов вычислительного ядра в строках 100–103. Затем мы должны выделить память для массивов устройства. Для этой операции нам нужен блокнотный массив, в котором мы можем хранить суммы для каждого блока из первого вычислительного ядра. Мы выделяем его в строке 108 с размером gridsize (размер решетки), потому что это число равно числу имеющихся у нас блоков. Нам также нужен блокнотный массив с совместной памятью, равный размеру блока. Мы вычисляем этот размер в строке 101 и передаем его в ядро в строках 112 и 115 шевронному оператору в качестве третьего параметра. Третий параметр необязателен; мы встречаемся с его применением впервые. Взгляните на листинг 12.9 (строка 56) и листинг 12.7 (строка 29), чтобы увидеть, где на устройстве GPU обрабатывается соответствующий исходный код блокнота.

Пытаться прослеживать все запутанные циклы бывает трудно. Поэтому мы создали версию исходного кода, которая выполняет те же циклы на CPU и печатает свои значения по ходу работы. Указанная версия находится в каталоге CUDA/SumReductionRevealed по адресу https://github.com/EssentialsofParallelComputing/Chapter12.

Здесь у нас нет места, чтобы показать весь этот исходный код, но вы, возможно, найдете полезным проводить разведывательный анализ значений и их распечатку по мере его исполнения. Мы покажем отредактированную версию распечатки в следующем ниже примере.

Пример: CUDA/SumReductionRevealed

```
Calling first pass with gridsize 2 blocksize 128 blocksizebytes 1024

SYNCTHREADS after all values are in shared memory block
Data count is 200
 ====== ITREE_LEVEL 1 offset 64 ntX is 128 MIN_REDUCE_SYNC_SIZE 32 ====
Data count is reduced to 128
Sync threads when larger than warp
 ====== ITREE_LEVEL 2 offset 32 ntX is 128 MIN_REDUCE_SYNC_SIZE 32 ====
Sync threads when smaller than warp
Data count is reduced to 64
 ====== ITREE_LEVEL 3 offset 16 ntX is 128 MIN_REDUCE_SYNC_SIZE 32 ====
Sync threads when smaller than warp
Data count is reduced to 32
 ====== ITREE_LEVEL 4 offset 8 ntX is 128 MIN_REDUCE_SYNC_SIZE 32 ====
Sync threads when smaller than warp
Data count is reduced to 16
```

```
 ====== ITREE_LEVEL 5 offset 4 ntX is 128 MIN_REDUCE_SYNC_SIZE 32 ====
Sync threads when smaller than warp
Data count is reduced to 8
 ====== ITREE_LEVEL 6 offset 2 ntX is 128 MIN_REDUCE_SYNC_SIZE 32 ====
Sync threads when smaller than warp
Data count is reduced to 4
 ====== ITREE_LEVEL 7 offset 1 ntX is 128 MIN_REDUCE_SYNC_SIZE 32 ====
Data count is reduced to 2
Finished reduction sum within thread block
End of first pass
Synchronization in second pass after loading data
Data count is reduced to 2
 ====== ITREE_LEVEL 8 offset 1 ntX is 128 MIN_REDUCE_SYNC_SIZE 32 ====
Data count is reduced to 1
```

```
Finished reduction sum within thread block

End of first pass
```

```
Synchronization in second pass after loading data
Data count is reduced to 2

 ====== ITREE_LEVEL 8 offset 1 ntX is 128 MIN_REDUCE_SYNC_SIZE 32 ====
Data count is reduced to 1
```

```
Finished reduction sum within thread block

Synchronization in second pass after reduction sum
Result -- total sum 19900
```

Этот пример относится к массиву длиной 200 целых чисел, каждый элемент которого инициализирован индексным значением. Мы предлагаем вам сверяться с исходным кодом и рис. 12.1, чтобы понять, что происходит. Печатаются начало и конец первого и второго прохождения. Мы видим, что количество данных сокращается в два раза, пока в конце первого прохождения не останется только два. Второе прохождение быстро сводит их к одному значению, содержащему сумму.

Мы показали эту редукцию поточных блоков в качестве общего введения в вычислительные ядра, которые требуют кооперации потоков. Хорошо видно, насколько это сложно, в особенности по сравнению с одной строкой, необходимой для внутреннего вызова в Fortran. По ходу мы также получили значительное ускорение на CPU и хранили данные этой операции на GPU. Указанный алгоритм редукции можно дополнительно оптимизировать, но вы также можете подумать об использовании некоторых библиотечных служб, таких как CUDA UnBound (CUB), Thrust или другие библиотеки GPU.

12.2.3 HIP'ифицирование исходного кода CUDA

Исходный код CUDA работает только на GPU-процессорах NVIDIA. Но в AMD был имплементирован аналогичный язык GPU под названием «Гетерогенный интерфейс для обеспечения переносимости» (Heterogeneous Interface for Portability, HIP). Он является частью комплекта инструментов ROCm (от Radeon Open Compute platform, или Открытая вычислительная платформа Radeon) от AMD. Если вы программируете на языке HIP, то можете вызывать компилятор hipcc, который использует NVCC на платформах NVIDIA и HCC на GPU-процессорах AMD.

Для того чтобы попробовать эти примеры, вам, возможно, потребуется инсталлировать комплект программно-информационного обеспечения и инструментов ROCm. Процесс инсталлирования часто меняется, поэтому ознакомьтесь с последними инструкциями. Кроме того, некоторые инструкции также прилагаются к примерам.

Пример: простой файл makefile для HIP'ифицирования исходного кода CUDA

Существует две версии файла makefile. В одной используется `hipify-perl`, а в другой – `hipify-clang`. `hipify-perl` является простым скриптом на языке Perl. Для более точной трансляции с учетом синтаксиса вы можете попробовать `hipify-clang`. В любом случае для более сложных программ вам, возможно, потребуется внести последние модификации вручную. Мы будем использовать версию на Perl, поэтому давайте начнем со связывания скрипта Makefile.perl, показанного в следующем ниже листинге, с Makefile:

```
ln -s Makefile.perl Makefile
make
```

Простой makefile для HIP

`HIP/StreamTriad/Makefile.perl`

```
 1 all: StreamTriad
 2
 3 CXX = hipcc
 4
 5 %.cc : %.cu
 6   hipify-perl $^ > $@
 7
 8 StreamTriad: StreamTriad.o timer.o
 9   ${CXX} -o $@ $^
10
11 clean:
12   rm -rf StreamTriad *.o StreamTriad.cc
```

Задает компилятор C++ как hipcc

Конвертирует исходный код CUDA в исходный код HIP

Единственным реальным дополнением к стандартному makefile является замена компилятора на hipcc и добавление шаблонного правила для конвертирования исходного кода CUDA в исходный код HIP. Мы могли бы выполнить конверсию кода, просто вызвав скрипт hipify-perl вручную, а затем применив HIP-версию как для GPU-процессоров CUDA, так и для GPU-процессоров AMD.

В CMake тоже есть хорошая поддержка HIP. Указанная поддержка HIP имеется, начиная с версии CMake 2.8.3. Типичный файл CMakeLists для HIP показан в следующем ниже листинге.

Листинг 12.11 Сборка программы HIP с помощью CMake

HIP/StreamTriad/CMakeLists.txt

```
 1 cmake_minimum_required (VERSION 2.8.3)
 2 project (StreamTriad)
 3
  ...
 6 if(NOT DEFINED HIP_PATH)
 7    if(NOT DEFINED ENV{HIP_PATH})
 8       set(HIP_PATH "/opt/rocm/hip" CACHE PATH "Путь к инсталляции HIP")
 9    else()
10       set(HIP_PATH $ENV{HIP_PATH} CACHE PATH "Путь к инсталляции HIP")
11    endif()
12 endif()
13 set(CMAKE_MODULE_PATH "${HIP_PATH}/cmake" ${CMAKE_MODULE_PATH})
14
15 find_package(HIP REQUIRED)
16 if(HIP_FOUND)
17    message(STATUS "Обнаружен HIP: " ${HIP_VERSION})
20 endif()
21
22 set(CMAKE_CXX_COMPILER ${HIP_HIPCC_EXECUTABLE})
23 set(MY_HIPCC_OPTIONS )
24 set(MY_HCC_OPTIONS )
25 set(MY_NVCC_OPTIONS )
26
27 # Добавляет сборочную цель StreamTriad с файлами исходного кода
28 HIP_ADD_EXECUTABLE(StreamTriad StreamTriad.cc
                     timer.c timer.h)
29 target_include_directories(StreamTriad PRIVATE ${HIP_PATH}/include)
30 target_link_directories(StreamTriad PRIVATE ${HIP_PATH}/lib)
31 target_link_libraries(StreamTriad hip_hcc)
32
33 # Очистка
34 add_custom_target(distclean COMMAND rm -rf CMakeCache.txt CMakeFiles *.o
35     Makefile cmake_install.cmake StreamTriad.dSYM ipo_out.optrpt)
```

Минимальная версия CMake для HIP равна 2.8.3

Задает путь к инсталляции HIP

Находит HIP, используя путь

Задает компилятор C++ равным hipcc

Добавляет исполняемый файл, встраиваемые файлы и библиотеки

В этом листинге мы сначала пытаемся задать разные варианты пути для местоположения, где может находиться инсталляция HIP, а потом

вызываем `find_package` для HIP в строке 15. Затем в строке 22 мы задаем компилятор C++ равным `hipcc`. Команда `HIP_ADD_EXECUTABLE` добавляет сборку нашего исполняемого файла, и мы завершаем листинг настройками для заголовочных файлов и библиотек HIP (строки 28–31). Теперь давайте обратим наше внимание на исходный код HIP в листинге 12.12. Мы выделяем изменения по сравнению с CUDA-версией исходного кода, приведенной в листингах 12.5-12.6.

Листинг 12.12 Отличия HIP для потоковой триады

HIP/StreamTriad/StreamTriad.c

Нам нужно включить заголовок среды времени выполнения HIP

```
 1 #include "hip/hip_runtime.h"
       < . . . пропускаем . . . >
36    // выделить память устройства.
      // суффикс _d обозначает указатель устройства
37    double *a_d, *b_d, *c_d;
38    hipMalloc(&a_d, stream_array_size*
                sizeof(double));
39    hipMalloc(&b_d, stream_array_size*
                sizeof(double));
40    hipMalloc(&c_d, stream_array_size*
                sizeof(double));
       < . . . пропускаем. . . >
46    for (int k=0; k<NTIMES; k++){
47       cpu_timer_start(&ttotal);
48       // скопировать данные массива с хоста на устройство
49       hipMemcpy(a_d, a, stream_array_size*
           sizeof(double), hipMemcpyHostToDevice);
50       hipMemcpy(b_d, b, stream_array_size*
           sizeof(double), hipMemcpyHostToDevice);
51       // копирование памяти hipmemcpy на устройство возвращается, когда есть
52       // буфер, поэтому синхронизируем, чтобы получить точный хронометраж
         // только для вычислительного ядра
53       hipDeviceSynchronize();
54
55       cpu_timer_start(&tkernel);
56       // запустить вычислительное ядро потоковой триады
57       hipLaunchKernelGGL(StreamTriad,
           dim3(gridsize), dim3(blocksize), 0, 0,
           stream_array_size, scalar, a_d, b_d, c_d);
58       // нужно форсировать завершение, чтобы получить хронометраж
59       hipDeviceSynchronize();
60       tkernel_sum += cpu_timer_stop(tkernel);
61
62       // копирование памяти hipMemcpy с устройства на хост блокирует
         // с целью завершения, поэтому синхронизации не требуется
63       hipMemcpy(c, c_d, stream_array_size*
           sizeof(double), hipMemcpyDeviceToHost);
       < . . . пропускаем . . . >
72    }
       < . . . пропускаем . . . >
```

cudaMalloc становится hipMalloc

Синтаксис hipLaunchKernel является более традиционным, чем запуск ядра CUDA

cudaDeviceSynchronize становится hipDeviceSynchronize

cudaMemcpy становится hipMemcpy

```
75
76    hipFree(a_d);
77    hipFree(b_d);
78    hipFree(c_d);
```

hipFree заменяет cudaFree

В целях конвертирования из исходного кода CUDA в исходный код HIP мы заменяем все вхождения cuda в исходном коде на hip. Единственное более существенное изменение заключено в вызове запуска вычислительного ядра, где HIP использует более традиционный синтаксис, чем тройной шеврон, используемый в CUDA. Как ни странно, самые большие изменения заключаются в использовании правильной для двух языков терминологии в именовании переменных.

12.3 OpenCL для переносимого языка GPU с открытым исходным кодом

В связи с острой потребностью в переносимом исходном коде GPU в 2008 году появился новый язык программирования GPU под названием OpenCL. OpenCL – это открытый стандартный язык GPU, который может работать на графических картах NVIDIA и AMD/ATI, а также на многих других аппаратных устройствах. Разработкой стандарта OpenCL руководила Apple, в которой приняли участие многие другие организации. Одна из приятных особенностей OpenCL заключается в том, что для исходного кода хоста можно использовать практически любой компилятор C или даже C++. Для исходного кода GPU-устройства язык OpenCL изначально основывался на подмножестве C99. Недавно в версии 2.1 и 2.2 OpenCL была добавлена поддержка C++ 14, но ее имплементации все еще недоступны.

Релиз OpenCL стартовал с большого первоначального воодушевления. Наконец-то, вот он способ написания переносимого кода GPU. Например, GIMP объявил, что он будет поддерживать OpenCL как способ ускорения GPU, который будет доступен на многих аппаратных платформах. Реальность же оказалась менее убедительной. Многие считают, что OpenCL является слишком низкоуровневым и многословным для широкого распространения. Возможно даже, что его конечная роль состоит в том, чтобы выступать в качестве слоя низкоуровневой переносимости для языков более высокого уровня. Но его ценность в качестве языка написания переносимого кода для целого спектра аппаратных устройств была продемонстрирована его доступностью в сообществе встраиваемых устройств в части программируемых пользователями вентильных матриц (FPGA). Одна из причин, по которой OpenCL считается многословным, заключается в том, что выбор устройства более усложнен (и является более мощным). Вам приходится обнаруживать и выбирать

устройство, на котором вы будете работать. С самого начала бывает, что это приводит к сотне строк кода.

Почти каждый, кто использует OpenCL, пишет библиотеку для манипулирования низкоуровневыми задачами. И мы не исключение. Наша библиотека называется EZCL. Почти каждый вызов OpenCL обернут, по крайней мере, легким слоем для улаживания условий возникновения ошибок. Обнаружение устройств, компилирование кода и обработка ошибок потребляют много строк кода.

В наших примерах мы будем использовать сокращенную версию нашей библиотеки EZCL, именуемую EZCL_Lite, чтобы иметь возможность видеть фактические вызовы OpenCL. Процедуры EZCL_Lite используются для выбора устройства и его настройки для приложения, затем компилирования кода устройства и обработки ошибок. Исходный код этих операций слишком длинный, чтобы показывать его здесь, поэтому обратитесь к примерам в каталоге OpenCL по адресу https://github.com/EssentialsofParallelComputing/Chapter12. Указанный каталог также содержит полную библиотеку EZCL.

Процедуры EZCL дают подробные сведения об ошибках при вызовах и о том, в какой строке исходного кода происходит ошибка.

Прежде чем начинать пробовать исходный код OpenCL, проверьте, что у вас имеется надлежаще настроенная среда и устройства. Для этого вы можете использовать команду clinfo.

Пример: получение информации об инсталляции OpenCL

Выполните информационную команду OpenCL:

```
clinfo
```

Если вы получите следующую ниже распечатку, то OpenCL не настроен либо у вас нет соответствующего устройства OpenCL:

```
Number of platforms 0
```

Если у вас нет команды `clinfo`, то попробуйте ее инсталлировать с помощью соответствующей для вашей системы команды. Для Ubuntu это:

```
sudo apt install clinfo
```

Приведенные в этой главе примеры содержат несколько кратких советов по инсталлированию OpenCL, но проверьте наличие последней информации, относящейся к вашей системе. OpenCL имеет расширение, которое предоставляет подробную модель в отношении того, каким образом каждое устройство должно настраивать свой драйвер в своей спецификации инсталлируемого клиентского драйвера (Installable Client Driver, ICD). Благодаря этому имеется возможность иметь в приложении несколько платформ и драйверов OpenCL.

12.3.1 Написание и сборка вашего первого приложения OpenCL

Изменения в стандартном makefile для встраивания OpenCL не слишком сложны. Типичные изменения показаны в листинге 12.13. В целях использования простого makefile для OpenCL, следует набрать:

```
ln -s Makefile.simple Makefile
```

Затем собрать приложение посредством `make` и выполнить приложение с помощью ./StreamTriad.

Листинг 12.13 Простой makefile для OpenCL

OpenCL/StreamTriad/Makefile.simple

```
 1 all: StreamTriad
 2
 3 #CFLAGS = -DDEVICE_DETECT_DEBUG=1
 4 #OPENCL_LIB = -L<path>
 5
 6 %.inc : %.cl
 7   ./embed_source.pl $^ > $@
 8
 9 StreamTriad.o: StreamTriad.c StreamTriad_kernel.inc
10
11 StreamTriad: StreamTriad.o timer.o ezclsmall.o
12   ${CC} -o $@ $^ ${OPENCL_LIB} -lOpenCL
13
14 clean:
15   rm -rf StreamTriad *.o StreamTriad_kernel.inc
```

Включает детализацию обнаружения устройств

Шаблонное правило внедряет исходный код OpenCL

Файл makefile содержит способ установки флага `DEVICE_DETECT_DEBUG` для распечатки подробной информации об имеющихся устройствах GPU. Указанный флаг активирует более подробную детализацию в исходном коде ezcl_lite.c. Это бывает полезно для устранения проблем с обнаружением устройств либо для с получением неправильного устройства. В строке 6 также добавлено шаблонное правило, которое встраивает исходный код OpenCL в программу для использования во время выполнения. В строке 9 этот скрипт Perl конвертирует исходный код в комментарий и в качестве зависимости. Он будет включен в файл StreamTriad.c с помощью инструкции `include`.

Утилита embed_source.pl была разработана нами для того, чтобы связывать исходный код OpenCL непосредственно с исполняемым файлом. (Исходный код этой утилиты приведен в примерах главы.) Распространенный способ функционирования кода OpenCL состоит в том, чтобы иметь отдельный файл исходного кода, который должен локализовываться во время выполнения, который затем компилируется после того, как становится известным устройство. Использование отдельного файла создает проблемы, связанные с тем, что его невозможно найти либо с получением неправильной версии файла. Мы настоятельно реко-

мендуем внедрять исходный код в исполняемый файл, чтобы избегать такого рода проблем. Как показано в следующем ниже листинге, также есть возможность использовать поддержку CMake для OpenCL в нашей сборочной системе.

Листинг 12.14 Файл CMake для OpenCL

OpenCL/StreamTriad/CMakeLists.txt

```
 1 cmake_minimum_required (VERSION 3.1)
 2 project (StreamTriad)
 3
 4 if (DEVICE_DETECT_DEBUG)
 5    add_definitions(-DDEVICE_DETECT_DEBUG=1)
 6 endif (DEVICE_DETECT_DEBUG)
 7
 8 find_package(OpenCL REQUIRED)
 9 set(HAVE_CL_DOUBLE ON CACHE BOOL
       "Имеется дублер OpenCL")
10 set(NO_CL_DOUBLE OFF)
11 include_directories(${OpenCL_INCLUDE_DIRS})
12
13 # Добавляет сборочную цель StreamTriad с файлами исходного кода
14 add_executable(StreamTriad StreamTriad.c ezclsmall.c ezclsmall.h
                 timer.c timer.h)
15 target_link_libraries(StreamTriad ${OpenCL_LIBRARIES})
16 add_dependencies(StreamTriad StreamTriad_kernel_source)
17
18 ############ внедрить цель в виде исходого кода ##############
19 add_custom_command(OUTPUT
     ${CMAKE_CURRENT_BINARY_DIR}/StreamTriad_kernel.inc
20   COMMAND ${CMAKE_SOURCE_DIR}/embed_source.pl
     ${CMAKE_SOURCE_DIR}/StreamTriad_kernel.cl
     > StreamTriad_kernel.inc
21         DEPENDS StreamTriad_kernel.cl ${CMAKE_SOURCE_DIR}/embed_source.pl)
22 add_custom_target(
          StreamTriad_kernel_source ALL DEPENDS
     ${CMAKE_CURRENT_BINARY_DIR}/
          StreamTriad_kernel.inc)
23
24 # Очистка
25 add_custom_target(distclean COMMAND rm -rf CMakeCache.txt CMakeFiles
26                   Makefile cmake_install.cmake StreamTriad.dSYM
                     ipo_out.optrpt)
27
28 SET_DIRECTORY_PROPERTIES(PROPERTIES ADDITIONAL_MAKE_CLEAN_FILES
                           "StreamTriad_kernel.inc")
```

В CMake с версии 3.1 добавлена поддержка OpenCL

Активирует детализацию обнаружения устройств

Флаги задают поддержку CL_DOUBLE

Конкретно-прикладная команда внедряет исходный код OpenCL в исполняемый файл

Конкретно-прикладная команда внедряет исходный код OpenCL в исполняемый файл

Поддержка OpenCL в CMake была добавлена в версию 3.1. Мы добавили это версионное требование вверху файла CMakelists.txt в строке 1. Следует отметить еще несколько особых моментов. В этом примере мы использовали опцию `-DDEVICE_DETECT_DEBUG=1` в команде CMake, чтобы

активировать детализацию для обнаружения устройств. Кроме того, мы добавили способ включения и выключения поддержки двойной точности OpenCL. Мы использовали его в исходном коде EZCL_Lite, чтобы установить флаг компиляциии JIT («строго вовремя» от just-in-time) для исходного кода устройства OpenCL. Наконец, мы добавили конкретно-прикладную команду в строках 19–22 для встраивания исходного кода устройства OpenCL в исполняемый файл. Исходный код вычислительного ядра OpenCL находится в отдельном файле под названием StreamTriad_kernel.cl, как показано в следующем ниже листинге.

Листинг 12.15 Вычислительное ядро OpenCL

OpenCL/StreamTriad/StreamTriad_kernel.cl

```
 1 // OpenCL kernel version of stream triad
 2 __kernel void StreamTriad(
 3                const int n,
 4                const double scalar,
 5       __global const double *a,
 6       __global const double *b,
 7       __global double *c)
 8 {
 9    int i = get_global_id(0);
10
11    // Защитить от выхода за пределы границ
12    if (i >= n) return;
13
14    c[i] = a[i] + scalar*b[i];
15 }
```

Атрибут __kernel указывает на то, что это вызывается из хоста

Получает индекс потока

Сравните этот исходный код вычислительного ядра с исходным кодом вычислительного ядра для CUDA в листинге 12.4. Исходный код OpenCL почти идентичен, за исключением того, что `__kernel` заменяет `__global__` в объявлении подпрограммы, атрибут `__global` добавляется в аргументы-указатели, и существует другой способ получения индекса потока. Кроме того, исходный код вычислительного ядра CUDA находится в том же файле .cu, что и исходный код для хоста, тогда как исходный код OpenCL находится в отдельном файле .cl. Мы могли бы выделить исходный код CUDA в его собственный файл .cu и поместить исходный код хоста в стандартный файл исходного кода C++. Это было бы похоже на структуру, которую мы используем для нашего приложения OpenCL.

ПРИМЕЧАНИЕ Многие различия между исходными кодами вычислительных ядер для CUDA и OpenCL имеют поверхностный характер.

Итак, насколько исходный код OpenCL на стороне хоста отличается от версии CUDA? Давайте взглянем на версию OpenCL в листинге 12.16 и сравним ее с исходным кодом в листинге 12.5. Существует две версии потоковой триады OpenCL: StreamTriad_simple.c без проверки ошибок

и StreamTriad.c с проверкой ошибок. Проверка ошибок добавляет много строк кода, которые изначально просто мешают понимать происходящее.

Листинг 12.16 OpenCL-версия потоковой триады: настройка и демонтаж

OpenCL/StreamTriad/StreamTriad_simple.c

```
 5 #include "StreamTriad_kernel.inc"
 6 #ifdef __APPLE_CC__
 7 #include <OpenCL/OpenCL.h>
 8 #else
 9 #include <CL/cl.h>
10 #endif
11 #include "ezcl_lite.h"
    < . . . пропускаем исходный код . . . >
32   cl_command_queue command_queue;
33   cl_context context;
34   iret = ezcl_devtype_init(
           CL_DEVICE_TYPE_GPU, &command_queue, &context);
35   const char *defines = NULL;
36   cl_program program =
       ezcl_create_program_wsource(context,
       defines, StreamTriad_kernel_source);
37   cl_kernel kernel_StreamTriad =
       clCreateKernel(program, "StreamTriad",
       &iret);
38
39   // выделить память устройства.
     // суффикс _d обозначает указатель устройства
40   size_t nsize = stream_array_size*sizeof(double);
41   cl_mem a_d = clCreateBuffer(context,
       CL_MEM_READ_WRITE, nsize, NULL, &iret);
42   cl_mem b_d = clCreateBuffer(context,
       CL_MEM_READ_WRITE, nsize, NULL, &iret);
43   cl_mem c_d = clCreateBuffer(context,
       CL_MEM_READ_WRITE, nsize, NULL, &iret);
44
45   // установка размера рабочей группы и дополнение,
     // чтобы получить четное число рабочих групп
46   size_t local_work_size = 512;
47   size_t global_work_size = ( (stream_array_size + local_work_size - 1)
         /local_work_size ) * local_work_size;
    < . . . пропускаем исходный код . . . >
74   clReleaseMemObject(a_d);
75   clReleaseMemObject(b_d);
76   clReleaseMemObject(c_d);
77
78   clReleaseKernel(kernel_StreamTriad);
79   clReleaseCommandQueue(command_queue);
80   clReleaseContext(context);
81   clReleaseProgram(program);
```

У Apple должно быть по-другому

Наша поддерживающая библиотека EZCL_Lite

Получает устройство GPU

Создает программу из исходного кода

Компилирует вычислительное ядро StreamTriad в исходном коде

Манипулирует памятью массива

Расчет размера рабочей группы аналогичен CUDA

Очищает вычислительное ядро и связанные с устройством объекты

В начале программы мы сталкиваемся с несколькими реальными различиями в строках 34–37, где мы должны найти наше устройство GPU и скомпилировать исходный код нашего устройства. Это делается для нас за кулисами в CUDA. Две строки кода OpenCL вызывают наши процедуры EZCL_Lite для обнаружения устройства и создания программного объекта. Мы сделали эти вызовы по той причине, что объем необходимого для этих функций исходного кода слишком велик, чтобы показывать его здесь. Источником этих процедур являются сотни строк, при этом бо́льшая их часть посвящена проверке ошибок.

ПРИМЕЧАНИЕ Исходный код с примерами главы доступен в каталоге OpenCL/StreamTriad по адресу https://github.com/EssentialsofParallelComputing/Chapter12. Часть исходного кода проверки ошибок была исключена из короткой версии StreamTriad_simple.c, но она содержится в длинной версии кода в файле StreamTriad.c.

Остальная часть кода настройки и демонтажа следует тому же шаблону, который мы встречали в коде CUDA, с небольшой дополнительной очисткой, опять же связанной с манипулированием исходным кодом устройства и программы. Теперь как соотносится раздел исходного кода, который вызывает вычислительное ядро OpenCL в хронометражном цикле из листинга 12.16, с исходным кодом CUDA из листинга 12.6?

Листинг 12.17 OpenCL-версия потоковой триады: вызов вычислительного ядра и хронометражный цикл

OpenCL/StreamTriad/StreamTriad_simple.c

```
49    for (int k=0; k<NTIMES; k++){
50       cpu_timer_start(&ttotal);
51       // копирование данных массива с хоста на устройство
52       iret=clEnqueueWriteBuffer(command_queue,
             a_d, CL_FALSE, 0, nsize, &a[0], 0, NULL, NULL);
53       iret=clEnqueueWriteBuffer(command_queue,
             b_d, CL_TRUE, 0, nsize, &b[0], 0, NULL, NULL);
54
55       cpu_timer_start(&tkernel);
56       // задать аргументы вычислительного ядра потоковой триады
57       iret=clSetKernelArg(kernel_StreamTriad,
             0, sizeof(cl_int), (void *)&stream_array_size);
58       iret=clSetKernelArg(kernel_StreamTriad,
             1, sizeof(cl_double), (void *)&scalar);
59       iret=clSetKernelArg(kernel_StreamTriad,
             2, sizeof(cl_mem), (void *)&a_d);
60       iret=clSetKernelArg(kernel_StreamTriad,
             3, sizeof(cl_mem), (void *)&b_d);
61       iret=clSetKernelArg(kernel_StreamTriad,
             4, sizeof(cl_mem), (void *)&c_d);
62       // call stream triad kernel
```

Вызовы перемещения памяти

Задает аргументы вычислительного ядра

```
63      clEnqueueNDRangeKernel(command_queue,
            kernel_StreamTriad, 1, NULL,
            &global_work_size, &local_work_size,
            0, NULL, NULL);
64      // необходимо форсировать завершение, чтобы получить хронометраж
65      clEnqueueBarrier(command_queue);
66      tkernel_sum += cpu_timer_stop(tkernel);
67
68      iret=clEnqueueReadBuffer(command_queue,
            c_d, CL_TRUE, 0, nsize, c, 0, NULL, NULL);
69      ttotal_sum += cpu_timer_stop(ttotal);
70  }
```

Вызывает вычислительное ядро

Барьер синхронизации

Что происходит в строках 57–61? OpenCL требует отдельного вызова для каждого аргумента вычислительного ядра. Если мы будем проверять числовой код, возвращаемый из каждого ядра, то мы получим еще больше строк. Это будет намного подробнее, чем одна строка 53 в листинге 12.6 в CUDA-версии. Но между этими двумя версиями есть прямое соответствие. OpenCL просто подробнее описывает операции по передаче аргументов. За исключением обнаружения устройств и компиляции программ, обе программы схожи в своих операциях. Самое большое различие заключается в используемом в этих двух языках синтаксисе.

В листинге 12.18 мы показываем примерную последовательность вызовов для обнаружения устройств и вызовов программы создания. Эти процедуры делаются длинными за счет проверки ошибок и обработки, необходимой для особых случаев. В этих двух функциях важно иметь хорошую обработку ошибок. Нам нужен отчет компилятора об ошибке в исходном коде или о том, что он получил неправильное устройство GPU.

Листинг 12.18 Библиотека ezcl_lite с поддержкой OpenCL

```
OpenCL/StreamTriad/ezcl_lite.c
/* инициализировать и закончить процедуру */
cl_int ezcl_devtype_init(cl_device_type device_type,
   cl_command_queue *command_queue, cl_context *context);
clGetPlatformIDs -- сперва получить число платформ и выделить
clGetPlatformIDs -- теперь получить платформы
Прокрутить в цикле по числу платформ и
   clGetDeviceIDs -- один раз получить число устройств и выделить
   clGetDeviceIDs -- получить устройства
   выполнить проверку на поддержку двойной точности -- clGetDeviceInfo
Конец цикла
clCreateContext
clCreateCommandQueue

/* процедуры вычислительного ядра и программы */
cl_program ezcl_create_program_wsource(cl_context context,
    const char *defines, const char *source);
       clCreateProgramWithSource
       задать строку компиляции (опции, специфичные для оборудования)
```

```
clBuildProgram
Проверить ошибку, если обнаружена
   clGetProgramBuildInfo
   и распечатать отчет компилятора
Конец обработки ошибок
```

Мы завершаем эту презентацию OpenCL кивком в сторону многочисленных языковых интерфейсов, которые были для него созданы. Существуют версии на C++, Python, Perl и Java. В каждом из этих языков был создан интерфейс более высокого уровня, который скрывает некоторые детали в C-версии OpenCL. И мы настоятельно рекомендуем использовать нашу библиотеку EZCL либо одну из многих других библиотек промежуточного уровня для OpenCL.

Существует неофициальная версия на C++, доступная с версии OpenCL v1.2. Ее имплементация представляет собой всего лишь тонкий слой поверх C-версии OpenCL. Несмотря на то что она не получила одобрения комитета по стандартам, она полностью пригодна для использования разработчиками. Она доступна по адресу https://github.com/Khronos-Group/OpenCL-CLHPP. Официальное одобрение C++ в OpenCL произошло совсем недавно, но мы все еще ждем имплементаций.

12.3.2 Редукции в OpenCL

Редукция путем суммирования в OpenCL аналогична редукции в CUDA. Вместо пошагового анализа исходного код мы просто посмотрим на различия в исходном коде вычислительного ядра. Сначала на рис. 12.3 показана разница общей для обоих вычислительных ядер процедуры `sum_within_block` в сопоставлении бок о бок.

OpenCL

```
1
2
3  void reduction_sum_within_block(__local  double  *spad)
4  {
5     const unsigned int tiX  = get_local_id(0);
6     const unsigned int ntX  = get_local_size(0);
7
8     for (int offset = ntX >> 1; offset > MIN_REDUCE_SYNC_SIZE; offset >>= 1) {
9        if (tiX < offset) {
10          spad[tiX] = spad[tiX] + spad[tiX+offset];
11       }
12       barrier(CLK_LOCAL_MEM_FENCE);
13    }
14    if (tiX < MIN_REDUCE_SYNC_SIZE) {
15       for (int offset = MIN_REDUCE_SYNC_SIZE; offset > 1; offset >>= 1) {
16          spad[tiX] = spad[tiX] + spad[tiX+offset];
17          barrier(CLK_LOCAL_MEM_FENCE);
18       }
19       spad[tiX] = spad[tiX] + spad[tiX+1];
20    }
21 }
```

CUDA

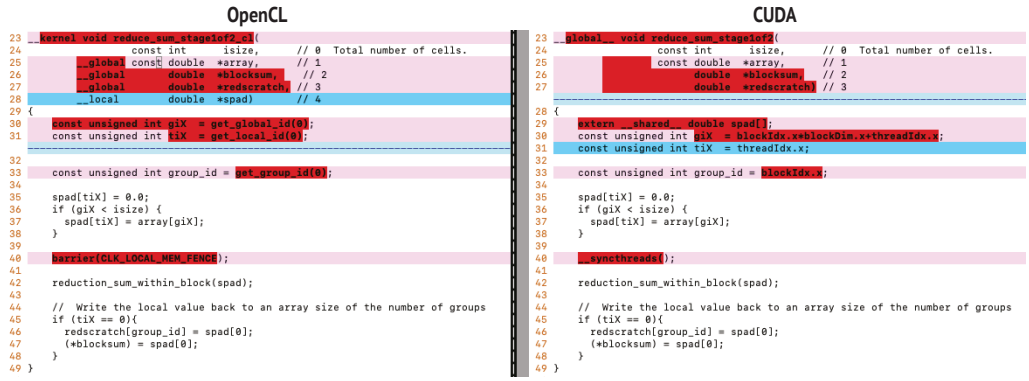
```
1  #define MIN_REDUCE_SYNC_SIZE warpSize
2
3  __device__ void reduction_sum_within_block(double  *spad)
4  {
5     const unsigned int tiX  = threadIdx.x;
6     const unsigned int ntX  = blockDim.x;
7
8     for (int offset = ntX >> 1; offset > MIN_REDUCE_SYNC_SIZE; offset >>= 1) {
9        if (tiX < offset) {
10          spad[tiX] = spad[tiX] + spad[tiX+offset];
11       }
12       __syncthreads();
13    }
14    if (tiX < MIN_REDUCE_SYNC_SIZE) {
15       for (int offset = MIN_REDUCE_SYNC_SIZE; offset > 1; offset >>= 1) {
16          spad[tiX] = spad[tiX] + spad[tiX+offset];
17          __syncthreads();
18       }
19       spad[tiX] = spad[tiX] + spad[tiX+1];
20    }
21 }
```

Рис. 12.3 Сравнение вычислительных ядер редукции в OpenCL и CUDA: sum_within_block

Разница в этом вычислительном ядре устройства, вызываемом другим вычислительным ядром, начинается с атрибутов в объявлении. Для CUDA в объявлении требуется атрибут `__device__`, в то время как OpenCL этого не требует. Для аргументов, передаваемых в блокнотном массиве, требуется атрибут `__local`, который не нужен CUDA. Следующим отличием является синтаксис определения индекса локального потока и размера блока (плитки) (рис. 12.3 в строках 5 и 6). Вызовы синхронизации

также различны. Вверху процедуры размер варпа определяется макрокомандой, которая помогает обеспечивать переносимость между GPU-процессорами NVIDIA и AMD. CUDA определяет его как переменную размера варпа. В OpenCL он передается вместе с определением компилятора. Мы также меняем терминологию с блока на плитку в фактическом исходном коде с целью обеспечения согласованности с терминологией каждого языка.

Следующая далее процедура представляет собой первое из двух прохождений вычислительного ядра, именуемое этапом 1of2, на рис. 12.4. Это ядро вызывается с хоста. Атрибут `__global__` для CUDA становится `__kernel` для OpenCL. Мы также должны добавить атрибут `__global` в аргументы-указатели для OpenCL.




```
__kernel void reduce_sum_stage1of2_cl(
                const int      isize,      // 0  Total number of cells.
        __global const double  *array,     // 1
        __global       double  *blocksum,  // 2
        __global       double  *redscratch,// 3
        __local        double  *spad)      // 4
{
    const unsigned int giX  = get_global_id(0);
    const unsigned int tiX  = get_local_id(0);

    const unsigned int group_id = get_group_id(0);

    spad[tiX] = 0.0;
    if (giX < isize) {
      spad[tiX] = array[giX];
    }

    barrier(CLK_LOCAL_MEM_FENCE);

    reduction_sum_within_block(spad);

    //  Write the local value back to an array size of the number of groups
    if (tiX == 0){
      redscratch[group_id] = spad[0];
      (*blocksum) = spad[0];
    }
}
```

```
__global__ void reduce_sum_stage1of2(
                const int      isize,      // 0  Total number of cells.
                const double  *array,      // 1
                      double  *blocksum,   // 2
                      double  *redscratch) // 3

{
    extern __shared__ double spad[];
    const unsigned int giX  = blockIdx.x*blockDim.x+threadIdx.x;
    const unsigned int tiX  = threadIdx.x;

    const unsigned int group_id = blockIdx.x;

    spad[tiX] = 0.0;
    if (giX < isize) {
      spad[tiX] = array[giX];
    }

    __syncthreads();

    reduction_sum_within_block(spad);

    //  Write the local value back to an array size of the number of groups
    if (tiX == 0){
      redscratch[group_id] = spad[0];
      (*blocksum) = spad[0];
    }
}
```

Рис. 12.4 Сравнение для первого из двух прохождений вычислительного ядра для редукционных ядер OpenCL и CUDA

Следующее различие является важным, и на него следует обратить внимание. В CUDA мы объявляем блокнот в совместной памяти как переменную `extern__shared__` в теле вычислительного ядра. На стороне хоста размер этого совместного пространства памяти указан в виде числа байтов в необязательном третьем аргументе в тройных угловых (шевронных) скобках. В OpenCL это делается по-другому. Он передается в качестве последнего аргумента в списке аргументов с атрибутом `__local`. На стороне хоста память задается в вызове, который задает значение четвертого аргумента вычислительного ядра:

```
clSetKernelArg(reduce_sum_1of2, 4,
                local_work_size*sizeof(cl_double), NULL);
```

Размер является третьим аргументом в вызове. Остальные изменения касаются синтаксиса относительно установки поточных параметров и вызова синхронизации. Последней частью сравнения является второе прохождение вычислительного редукционного ядра на рис. 12.5.

Мы уже встречали все шаблоны изменений во втором ядре. У нас все еще есть различия в объявлении вычислительного ядра и аргументах.

Локальный блокнотный массив также имеет те же отличия, что и вычислительное ядро для первого прохождения. Поточные параметры и синхронизация также имеют одинаково ожидаемые различия.

Оглядываясь назад на три сравнения на рис. 12.3–12.5, мы четко видим то, что нам не пришлось отмечать. Тела вычислительных ядер являются по существу одинаковыми.

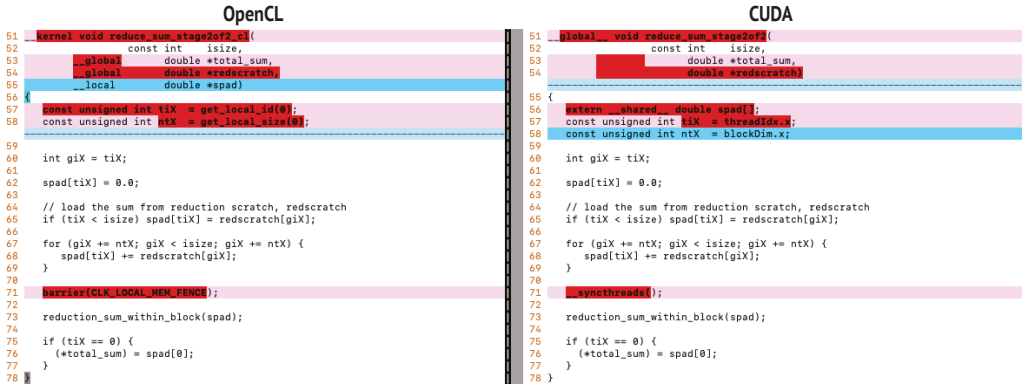


Рис. 12.5 Сравнение второго прохождения редукции суммированием

Единственное различие заключается в синтаксисе вызова синхронизации. Исходный код на стороне хоста для редукции суммированием в OpenCL показан в следующем ниже листинге.

Листинг 12.19 Исходный код хоста для OpenCL-версии редукции путем суммирования

OpenCL/SumReduction/SumReduction.c

```
20 cl_context context;
21 cl_command_queue command_queue;
22 ezcl_devtype_init(CL_DEVICE_TYPE_GPU, &command_queue, &context);
23
24 const char *defines = NULL;
25 cl_program program = ezcl_create_program_wsource(context, defines,
       SumReduction_kernel_source);
26 cl_kernel reduce_sum_1of2=clCreateKernel(
       program, "reduce_sum_stage1of2_cl", &iret);
27 cl_kernel reduce_sum_2of2=clCreateKernel(
       program, "reduce_sum_stage2of2_cl", &iret);
28
29 struct timespec tstart_cpu;
30 cpu_timer_start(&tstart_cpu);
31
32 size_t local_work_size = 128;
33 size_t global_work_size = ((nsize + local_work_size - 1)
     /local_work_size) * local_work_size;
34 size_t nblocks = global_work_size/local_work_size;
```

Два вычислительных ядра, создаваемых из одного исходного кода

```
35
36 cl_mem dev_x = clCreateBuffer(context, CL_MEM_READ_WRITE,
      nsize*sizeof(double), NULL, &iret);
37 cl_mem dev_total_sum = clCreateBuffer(context, CL_MEM_READ_WRITE,
      1*sizeof(double), NULL, &iret);
38 cl_mem dev_redscratch = clCreateBuffer(context, CL_MEM_READ_WRITE,
      nblocks*sizeof(double), NULL, &iret);
39
40 clEnqueueWriteBuffer(command_queue, dev_x, CL_TRUE, 0,
      nsize*sizeof(cl_double), &x[0], 0, NULL, NULL);
41
42 clSetKernelArg(reduce_sum_1of2, 0,
      sizeof(cl_int), (void *)&nsize);
43 clSetKernelArg(reduce_sum_1of2, 1,
      sizeof(cl_mem), (void *)&dev_x);
44 clSetKernelArg(reduce_sum_1of2, 2,
      sizeof(cl_mem), (void *)&dev_total_sum);
45 clSetKernelArg(reduce_sum_1of2, 3,
      sizeof(cl_mem), (void *)&dev_redscratch);
46 clSetKernelArg(reduce_sum_1of2, 4,
      local_work_size*sizeof(cl_double), NULL);
47
48 clEnqueueNDRangeKernel(command_queue,
      reduce_sum_1of2, 1, NULL, &global_work_size,
      &local_work_size, 0, NULL, NULL);
49
50 if (nblocks > 1) {
51    clSetKernelArg(reduce_sum_2of2, 0,
         sizeof(cl_int), (void *)&nblocks);
52    clSetKernelArg(reduce_sum_2of2, 1,
        sizeof(cl_mem), (void *)&dev_total_sum);
53    clSetKernelArg(reduce_sum_2of2, 2,
         sizeof(cl_mem), (void *)&dev_redscratch);
54    clSetKernelArg(reduce_sum_2of2, 3,
         local_work_size*sizeof(cl_double), NULL);
55
56    clEnqueueNDRangeKernel(command_queue,
         reduce_sum_2of2, 1, NULL, &local_work_size,
         &local_work_size, 0, NULL, NULL);
57 }
58
59 double total_sum;
60
61 iret=clEnqueueReadBuffer(command_queue, dev_total_sum, CL_TRUE, 0,
      1*sizeof(cl_double), &total_sum, 0, NULL, NULL);
62
63 printf("Результат -- итоговая сумма %lf \n",total_sum);
64
65 clReleaseMemObject(dev_x);
66 clReleaseMemObject(dev_redscratch);
67 clReleaseMemObject(dev_total_sum);
68
```

Вызывает первое прохождение редукции

Если требуется второе прохождение...

...вызывает второе прохождение редукции

```
69 clReleaseKernel(reduce_sum_1of2);
70 clReleaseKernel(reduce_sum_2of2);
71 clReleaseCommandQueue(command_queue);
72 clReleaseContext(context);
73 clReleaseProgram(program);
```

Вызов первого прохождения вычислительного ядра создает локальный блокнотный массив в строке 46. Промежуточные результаты сохраняются обратно в массив `redscratch`, созданный в строке 38. Если существует более одного блока, то необходимо второе прохождение. Массив `redscratch` передается обратно для завершения редукции. Обратите внимание, что параметры вычислительного ядра в аргументах 5 и 6 заданы равными `local_work_size` или одной рабочей группе. Это делается для того, чтобы иметь возможность выполнять синхронизацию по всем оставшимся данным, и еще одно прохождение не потребуется.

12.4 SYCL: экспериментальная имплементация на C++ становится магистральной

SYCL появился в 2014 году как экспериментальная имплементация на C++ поверх OpenCL. Цель создающих SYCL разработчиков состоит в более естественном расширении языка C++, чем ощущение надстройки над OpenCL на языке C. Он разрабатывается как кросс-платформенный слой абстракции, в котором задействуется переносимость и эффективность OpenCL. Фокус внимания экспериментального языка внезапно изменился, когда Intel выбрала его в качестве одного из своих первостепенных языковых маршрутов для анонсированной системы HPC Aurora Департамента энергетики. В системе Aurora будут использоваться новые дискретные GPU Intel, которые находятся в стадии разработки. Intel предложила несколько дополнений к стандарту SYCL, которые они прототипировали в своем компиляторе Data Parallel C++ (DPCPP) в своей открытой системе программирования oneAPI.

Вы можете познакомиться с SYCL несколькими путями. Некоторые из них даже избавляют от необходимости инсталлировать программно-информационное обеспечение или иметь правильное оборудование. Сначала вы можете опробовать следующие ниже облачные системы.

- Интерактивный SYCL предлагает учебное пособие на веб-сайте tech.io по адресу https://tech.io/playgrounds/48226/introduction-to-sycl/introduction-to-sycl-2.
- Intel предоставляет облачную версию oneAPI и DPCPP по адресу https://software.intel.com/en-us/oneapi. В целях ее использования вы должны зарегистрироваться.

Вы также можете скачать и инсталлировать версии SYCL со следующих ниже веб-сайтов.

- Издание для сообщества ComputeCpp по адресу https://developer.codeplay.com/products/computecpp/ce/home/. В целях его скачивания вы должны зарегистрироваться.
- Компилятор Intel DPCPP по адресу https://github.com/intel/llvm/blob/sycl/sycl/doc/GetStartedGuide.md.
- Intel также предоставляет инструкции по настройке файлов Docker по адресу https://github.com/intel/oneapi-containers/blob/master/images/docker/basekit-devel-ubuntu18.04/Dockerfile.

Мы будем работать с DPCPP-версией SYCL от Intel. Инструкции по настройке инсталляции oneAPI в VirtualBox с прилагаемыми к этой главе примерами приведены в файле README.virtualbox по адресу https://github.com/EssentialsofParallelComputing/Chapter12. Вы должны иметь возможность выполнять VirtualBox практически в любой операционной системе. Давайте начнем с простого makefile для компилятора DPCPP, как показано в следующем ниже листинге.

Листинг 12.20 Простой makefile для DPCPP-версии SYCL

DPCPP/StreamTriad/Makefile

```
 1 CXX = dpcpp
 2 CXXFLAGS = -std=c++17 -fsycl -O3
 3
 4 all: StreamTriad
 5
 6 StreamTriad: StreamTriad.o timer.o
 7   $(CXX) $(CXXFLAGS) $^ -o $@
 8
 9 clean:
10   -rm -f StreamTriad.o StreamTriad
```

Указывает dpcpp в качестве компилятора C++

Добавляет опцию SYCL в флаги C++

Установка компилятора C++ равным компилятору Intel dpcpp берет на себя пути, библиотеки и включаемые файлы. Единственным другим требованием является установка некоторых флагов для компилятора C++. В следующем ниже листинге показан исходный код SYCL для нашего примера.

Листинг 12.21 Пример потоковой триады для DPCPP-версии SYCL

DPCPP/StreamTriad/StreamTriad.cc

```
 1 #include <chrono>
 2 #include "CL/sycl.hpp"
 3
 4 namespace Sycl = cl::sycl;
 5 using namespace std;
 6
 7 int main(int argc, char * argv[])
 8 {
 9    chrono::high_resolution_clock::time_point t1, t2;
```

Встраивает заголовочный файл SYCL4

Использует пространство имен SYCL

```
10
11    size_t nsize = 10000;
12    cout << "StreamTriad with " << nsize << " elements" << endl;
13
14    // данные хоста
15    vector<double> a(nsize,1.0);
16    vector<double> b(nsize,2.0);
17    vector<double> c(nsize,-1.0);
18
19    t1 = chrono::high_resolution_clock::now();
20
21    Sycl::queue Queue(Sycl::cpu_selector{});
22
23    const double scalar = 3.0;
24
25    Sycl::buffer<double,1> dev_a { a.data(),
         Sycl::range<1>(a.size()) };
26    Sycl::buffer<double,1> dev_b { b.data(),
         Sycl::range<1>(b.size()) };
27    Sycl::buffer<double,1> dev_c { c.data(),
         Sycl::range<1>(c.size()) };
28
29    Queue.submit([&](Sycl::handler&
          CommandGroup) {
30
31       auto a = dev_a.get_access<Sycl::
            access::mode::read>(CommandGroup);
32       auto b = dev_b.get_access<Sycl::
            access::mode::read>(CommandGroup);
33       auto c = dev_c.get_access<Sycl::
            access::mode::write>(CommandGroup)
34
35       CommandGroup.parallel_for<class
            StreamTriad>(Sycl::range<1>{nsize},
            [=] (Sycl::id<1> it){
36          c[it] = a[it] + scalar * b[it];
37       });
38    });
39    Queue.wait();
40
41    t2 = chrono::high_resolution_clock::now();
42
      double time1 = chrono::duration_cast<
                     chrono::duration<double> >(t2 - t1).count();
43    cout << "Время выполнения составляет " << time1*1000.0 << " мс " << endl;
44 }
```

Инициализирует векторы на стороне хоста константами

Настраивает устройство для CPU

Выделяет буфер устройства и устанавливает равным буферу хоста

Лямбда для отправки очереди

Получает доступ к массивам устройства

Лямбда для вычислительного ядра parallel for

Ожидает компиляции

Первая функция `Sycl` выбирает устройство и создает очередь для работы на нем. Мы запрашиваем CPU, хотя этот исходный код также будет работать для GPU-процессоров с унифицированной памятью.

```
Sycl::queue Queue(sycl::cpu_selector{});
```

Мы выбираем CPU для максимальной переносимости, чтобы исходный код выполнялся в большинстве систем. Для того чтобы этот исходный код работал на GPU-процессорах без унифицированной памяти, нам нужно было бы добавить явные копии данных из одного пространства памяти в другое. Используемый по умолчанию селектор предпочтительно отыскивает GPU, но откатывает к CPU. Если мы хотим выбрать только GPU либо только CPU, то мы также можем указать другие селекторы, такие как:

```
Sycl::queue Queue(sycl::default_selector{}); // берет устройство, используемое
                                            // по умолчанию
Sycl::queue Queue(sycl::gpu_selector{});    // отыскивает устройство GPU
Sycl::queue Queue(sycl::cpu_selector{});    // отыскивает устройство CPU
Sycl::queue Queue(sycl::host_selector{});   // выполняет на хосте (CPU)
```

Последняя опция означает, что он будет работать на хосте, как если бы не было исходного кода SYCL или OpenCL. Настраивать устройство и очередь намного проще, чем то, что мы делали в OpenCL. Теперь нам нужно настроить буферы устройств с помощью буфера SYCL:

```
Sycl::buffer<double,1> dev_a { a.data(), Sycl::range<1>(a.size()) };
```

Первый аргумент буфера – это тип данных, а второй – размерность данных. Затем мы даем ему имя переменной, `dev_a`. Первым аргументом переменной является массив данных хоста, используемый для инициализации массива устройств, а вторым – набор используемых индексов. В этом случае мы указываем одномерный диапазон от 0 до размера переменной a. В строке 29 мы сталкиваемся с первой лямбдой для создания обработчика группы команд для очереди:

```
Queue.submit([&](Sycl::handler& CommandGroup)
```

Мы представили лямбды в разделе 10.2.1. Выражение захвата лямбды, `[&]`, определяет захват внешних переменных, используемых в подпрограмме по ссылке. Для этой лямбды захват получает `nsize`, `scalar`, `dev_a`, `dev_b` и `dev_c` для использования в лямбде. Мы могли бы указать его с помощью только одной настройки захвата по ссылке `[&]` либо следующей ниже формой, где мы указываем каждую переменную, которая будет захватываться. Предпочтением хорошей практики программирования было бы последнее, но списки могут становиться длинными.

```
Queue.submit([&nsize, &scalar, &dev_a, &dev_b, &dev_c]
    (Sycl::handler& CommandGroup)
```

В теле лямбды мы получаем доступ к массивам устройств и переименовываем их для использования в рамках процедуры устройства. Это эквивалентно списку аргументов для обработчика группы команд. Затем мы создаем первую операционную задачу для группы команд, `parallel_for`. `parallel_for` также определяется с помощью лямбды.

```
CommandGroup.parallel_for<class StreamTriad>(Sycl::range<1>{nsize},[=]
                                             (Sycl::id<1> it)
```

Лямбда носит имя `StreamTriad`. Затем мы сообщаем ей, что будем работать в одномерном диапазоне, который варьируется от 0 до *nsize*. Выражение захвата, `[=]`, захватывает переменные `a`, `b` и `c` по значению. Определить, будет ли захват выполняться по ссылке либо по значению, сложно. Но если исходный код будет размещен на GPU, то изначальная ссылка может выйти за пределы области видимости и станет больше не валидной. В конце мы создаем одномерную индексную переменную, it, для прокручивания диапазона в цикле.

12.5 Языки более высокого уровня для обеспечения переносимости производительности

К настоящему времени вы видите, что различия между вычислительными ядрами CPU и GPU не столь уж велики. Так почему бы не сгенерировать каждый из них с использованием полиморфизма и шаблонов C++? Как раз это сделано в нескольких библиотеках, разработанных исследовательскими лабораториями Департамента энергетики. Эти проекты были начаты для решения проблемы переноса многих исходных кодов на новые аппаратные архитектуры. Система Kokkos была создана Национальными лабораториями Сандии и получила широкое распространение. В Национальной лаборатории Лоуренса Ливермора есть аналогичный проект под названием RAJA. Оба указанных проекта уже преуспели в достижении своей цели обеспечения возможностей мультиплатформенности с поддержкой единого исходного кода.

Эти два языка во многом схожи с языком SYCL, который вы видели в разделе 12.4. В действительности они заимствовали концепции друг у друга, стремясь обеспечить переносимость производительности. Каждый из них предоставляет библиотеки, которые представляют собой довольно легкие слои поверх языков параллельного программирования более низкого уровня. Мы кратко рассмотрим каждый из них.

12.5.1 Kokkos: экосистема обеспечения переносимости производительности

Kokkos – это хорошо продуманный слой абстракции для таких языков, как OpenMP и CUDA. Он находится в разработке с 2011 года. У Kokkos есть следующие именованные пространства исполнения. Они задействуются в сборке Kokkos соответствующим флагом CMake (или опцией сборки с помощью Spack). Некоторые из них проработаны лучше, чем другие.

Пространства исполнения Kokkos	Флаги поддержки CMake/Spack
`Kokkos::Serial`	`-DKokkos_ENABLE_SERIAL=On` (по умолчанию включено)
`Kokkos::Threads`	`-DKokkos_ENABLE_PTHREAD=On`
`Kokkos::OpenMP`	`-DKokkos_ENABLE_OPENMP=On`
`Kokkos::Cuda`	`-DKokkos_ENABLE_CUDA=On`
`Kokkos::HPX`	`-DKokkos_ENABLE_HPX=On`
`Kokkos::ROCm`	`-DKokkos_ENABLE_ROCm=On`

Упражнение: потоковая триада в Kokkos

Для этого упражнения мы собрали Kokkos с бэкендом OpenMP, а затем собрали и выполнили пример потоковой триады. В начале наберите:

```
git clone https://github.com/kokkos/kokkos
mkdir build && cd build
cmake ../kokkos -DKokkos_ENABLE_OPENMP=On
```

Затем перейдите в исходный каталог потоковой триады для Kokkos и выполните сборку вне дерева с помощью CMake:

```
mkdir build && cd build
export Kokkos_DIR=${HOME}/Kokkos/lib/cmake/Kokkos
cmake ..
make
export OMP_PROC_BIND=true
export OMP_PLACES=threads
```

Kokkos, собранный с помощью CMake, был оптимизирован таким образом, чтобы он был простым, как показано в следующем ниже листинге. Переменная `Kokkos_DIR` должна быть задана равной местоположению конфигурационного файла CMake для Kokkos.

Листинг 12.22 Файл CMake для Kokkos

Kokkos/StreamTriad/CMakeLists.txt

```
1 cmake_minimum_required (VERSION 3.10)
2 project (StreamTriad)
3
4 find_package(Kokkos REQUIRED)
5
6 add_executable(StreamTriad StreamTriad.cc)
7 target_link_libraries(StreamTriad Kokkos::kokkos)
```

Отыскивает Kokkos и задает флаги

Добавляет зависимости и флаги для сборки

Добавление опции CUDA в сборку Kokkos генерирует версию, работающую на GPU-процессорах NVIDIA. Kokkos может использовать много других платформ и языков, и еще больше платформ и языков постоянно разрабатывается.

Пример потоковой триады Kokkos в листинге 12.23 имеет некоторое сходство с SYCL в том, что в нем используются лямбды C++ для встраивания функций для CPU либо GPU. Для этого механизма Kokkos также поддерживает функторы, но лямбды менее многословны для использования на практике.

Листинг 12.23 Пример потоковой триады в Kokkos

Kokkos/StreamTriad/StreamTriad.cc

```
1 #include <Kokkos_Core.hpp>
```

Встраивает надлежащий заголовок Kokkos

```
 2
 3 using namespace std;
 4
 5 int main (int argc, char *argv[])
 6 {
 7    Kokkos::initialize(argc, argv);{
 8
 9    Kokkos::Timer timer;
10    double time1;
11
12    double scalar = 3.0;
13    size_t nsize = 1000000;
14    Kokkos::View<double *> a( "a", nsize);
15    Kokkos::View<double *> b( "b", nsize);
16    Kokkos::View<double *> c( "c", nsize);
17
18    cout << "StreamTriad с " << nsize << " элементами" << endl;
19
20    Kokkos::parallel_for(nsize,
            KOKKOS_LAMBDA (int i) {
21       a[i] = 1.0;
22    });
23    Kokkos::parallel_for(nsize,
            KOKKOS_LAMBDA (int i) {
24       b[i] = 2.0;
25    });
26
27    timer.reset();
28
29    Kokkos::parallel_for(nsize,
            KOKKOS_LAMBDA (const int i) {
30       c[i] = a[i] + scalar * b[i];
31    });
32
33    time1 = timer.seconds();
34
35    icount = 0;
36    for (int i=0; i<nsize && icount < 10; i++){
37       if (c[i] != 1.0 + 3.0*2.0) {
38          cout << "Ошибка с результатом c[" << i << "]=" << c[i] << endl;
39          icount++;
40       }
41    }
42
43       if (icount == 0)
            cout << "Программа завершилась с ошибкой." << endl;
44       cout << "Время выполнения составляет " << time1*1000.0 << " мс " << endl;
45
46    }
47    Kokkos::finalize();
48    return 0;
49 }
```

Инициализирует Kokkos

Объявляет массивы с помощью Kokkos::View

Лямбды Kokkos parallel_for для CPU либо GPU

Финализирует Kokkos

Программа Kokkos начинается с `Kokkos::initialize` и `Kokkos::finalize`. Эти команды запускают вещи, которые необходимы для пространства исполнения, такие как потоки. Kokkos уникален тем, что он встраивает гибкие выделения многомерных массивов в виде проекций данных, которые можно переключать в зависимости от целевой архитектуры. Другими словами, вы можете использовать другой порядок данных для CPU по сравнению с GPU. Мы используем `Kokkos::View` в строках 14–16, хотя это предназначено только для одномерных массивов. Реальная ценность приходит с многомерными массивами. Общий синтаксис для `Kokkos::View` таков:

```
View < double *** , Макет , ПространствоПамяти > имя (...);
```

Пространства памяти являются опцией для шаблона (template), но по умолчанию они соответствуют пространству исполнения. Некоторые области памяти таковы:

- `HostSpace`;
- `CudaSpace`;
- `CudaUVMSpace`.

Можно указать и макет, хотя он имеет свое значение по умолчанию, соответствующее пространству памяти:

- для `LayoutLeft` крайним левым индексом является индексный шаг 1 (используется по умолчанию для `CudaSpace`);
- для `LayoutRight` крайним правым индексом является индексный шаг 1 (используется по умолчанию для `HostSpace`).

Вычислительные ядра задаются с использованием синтаксисов лямбд в одном из трех шаблонов параллельности данных:

- `parallel_for`;
- `parallel_reduce`;
- `parallel_scan`.

В строках 20, 23 и 29 листинга 12.23 мы использовали шаблон `parallel_for`. Макрокоманда `KOKKOS_LAMBDA` заменяет синтаксис захвата `[=]` или `[&]`. Kokkos берет на себя его определение и делает это в гораздо более удобочитаемой форме.

12.5.2 RAJA для более адаптируемого слоя обеспечения переносимости производительности

Слой RAJA обеспечения переносимости производительности преследует цель обеспечения переносимости с минимальными нарушениями существующих исходных кодов Национальной лаборатории Лоуренса Ливермора. Во многих отношениях его проще и легче принять на вооружение, чем другие сопоставимые системы. RAJA может быть собран с поддержкой следующих опций:

- `-DENABLE_OPENMP=On` (по умолчанию включено);
- `-DENABLE_TARGET_OPENMP=On` (по умолчанию выключено);

- `-DENABLE_CUDA=On` (по умолчанию выключено);
- `-DENABLE_TBB=On` (по умолчанию выключено).

Как показано в следующем ниже листинге, RAJA также имеет хорошую поддержку CMake.

Листинг 12.24 Файл CMake в Raja

`Raja/StreamTriad/CMakeLists.txt`

```
 1 cmake_minimum_required (VERSION 3.0)
 2 project (StreamTriad)
 3
 4 find_package(Raja REQUIRED)
 5 find_package(OpenMP REQUIRED)
 6
 7 add_executable(StreamTriad StreamTriad.cc)
 8 target_link_libraries(StreamTriad PUBLIC RAJA)
 9 set_target_properties(StreamTriad PROPERTIES
                         COMPILE_FLAGS ${OpenMP_CXX_FLAGS})
10 set_target_properties(StreamTriad PROPERTIES
                         LINK_FLAGS "${OpenMP_CXX_FLAGS}")
```

RAJA-версия потоковой триады претерпевает всего несколько изменений, как показано в следующем ниже листинге. RAJA также активно задействует лямбды для обеспечения их переносимости на CPU- и GPU-процессоры.

Листинг 12.25 Пример потоковой триады в RAJA

`Raja/StreamTriad/StreamTriad.cc`

```
 1 #include <chrono>
 2 #include "RAJA/RAJA.hpp"
 3
 4 using namespace std;
 5
 6 int main(int RAJA_UNUSED_ARG(argc), char **RAJA_UNUSED_ARG(argv[]))
 7 {
 8    chrono::high_resolution_clock::time_point t1, t2;
 9    cout << "Running Raja Stream Triad\n";
10
11    const int nsize = 1000000;
12
13    // Выделить и инициализировать векторные данные.
14    double scalar = 3.0;
15    double* a = new double[nsize];
16    double* b = new double[nsize];
17    double* c = new double[nsize];
18
19    for (int i = 0; i < nsize; i++) {
20       a[i] = 1.0;
21       b[i] = 2.0;
```

Встраивает заголовки RAJA

```
22    }
23
24    t1 = chrono::high_resolution_clock::now();
25
26    RAJA::forall<RAJA::omp_parallel_for_exec>(
         RAJA::RangeSegment(0,nsize),[=](int i){
27       c[i] = a[i] + scalar * b[i];
28    });
29
30    t2 = chrono::high_resolution_clock::now();
31
     < ... проверка ошибок ... >
42    double time1 = chrono::duration_cast<
                     chrono::duration<double> >(t2 - t1).count();
43    cout << "Время выполнения составляет " << time1*1000.0 << " мс " << endl;
44 }
```

Функция RAJA forall с использованием лямбда C++

Необходимые изменения для RAJA состоят во встраивании заголовочного файла RAJA в строку 2 и замене вычислительного цикла на `Raja::forall`. Хорошо видно, что разработчики RAJA обеспечивают низкий порог входа для повышения переносимости производительности. Для выполнения теста RAJA мы включили скрипт, который собирает и инсталлирует RAJA, как показано в следующем ниже листинге. Затем скрипт переходит к сборке исходного кода потоковой триады с помощью RAJA и его выполняет.

Листинг 12.26 Интегрированный скрипт сборки и выполнения для потоковой триады RAJA

Raja/StreamTriad/Setup_Raja.sh

```
1 #!/bin/sh
2 export INSTALL_DIR=`pwd`/build/Raja
3 export Raja_DIR=${INSTALL_DIR}/share/raja/cmake
4
5 mkdir -p build/Raja_tmp && cd build/Raja_tmp
6 cmake ../../Raja_build -DCMAKE_INSTALL_PREFIX=${INSTALL_DIR}
7 make -j 8 install && cd .. && rm -rf Raja_tmp
8
9 cmake .. && make && ./StreamTriad
```

Raja_DIR указывает на инструмент CMake для RAJA

Собирает исходный код потоковой триады и выполняет его

В этой главе мы рассмотрели целый ряд самых разных языков программирования. Думайте о них как о диалектах общего языка, а не как о совершенно разных.

12.6 Материалы для дальнейшего изучения

Мы только начали знакомиться с нативными языками GPU и системами обеспечения переносимости производительности. Даже имея в своем распоряжении показанную в этой главе первоначальную функциональ-

ность, вы сможете приступить к имплементированию некоторых исходных кодов реальных приложений. Если вы относитесь к использованию любого из них в своих приложениях серьезно, то мы настоятельно рекомендуем воспользоваться многочисленными дополнительными ресурсами для выбранного вами языка.

12.6.1 Дополнительное чтение

CUDA является доминирующим языком GPU в течение многих лет, и поэтому имеется много материалов по программированию на нем. Возможно, в первую очередь следует посетить веб-сайт разработчика NVIDIA по адресу https://developer.nvidia.com/cuda-zone. Там вы найдете подробные руководства по инсталлированию и использованию CUDA.

- Книга Кирка и Хву была одной из самых популярных справочных материалов по программированию GPU NVIDIA:

  Дэвид Б. Кирк и У. Хву Вэнь-Мэй, «Программирование массивно-параллельных процессоров: практический подход» (David B. Kirk and W. Hwu Wen-Mei, *Programming massively parallel processors: a hands-on approach*, Morgan Kaufmann, 2016).
- AMD (https://rocm.github.io) создала веб-сайт, который охватывает все аспекты их экосистемы ROCm.
- Если вы и в правду хотите узнать больше об OpenCL, то мы настоятельно рекомендуем книгу Мэтью Скарпино:

  Мэтью Скарпино, «OpenCL в действии: как ускорить графику и вычисления» (Matthew Scarpino, *OpenCL in action: how to accelerate graphics and computations*, Manning, 2011).
- Хорошим источником дополнительной информации об OpenCL является веб-сайт https://www.iwocl.org, спонсируемый Международным семинаром по OpenCL (IWOCL). Они также ежегодно проводят международную конференцию. SyclCon тоже размещена на том же веб-сайте.
- Khronos является органом по открытым стандартам для OpenCL, SYCL и связанного с ними программно-информационного обеспечения. Они размещают спецификации языков, форумы и списки ресурсов:

  группа Khronos, https://www.khronos.org/opencl/ и https://www.khronos.org/sycl/.
- Для получения документации и учебных материалов по Kokkos см. их репозиторий на GitHub. Помимо скачивания программного обеспечения Kokkos, вы также найдете сопутствующее хранилище (https://github.com/kokkos/kokkos-tutorials) для учебных занятий, которые они дают по всей стране.
- Коллектив RAJA (https://raja.readthedocs.io) имеет обширную документацию на своем веб-сайте.

12.6.2 Упражнения

1. Измените выделение памяти хоста в CUDA-примере потоковой триады, чтобы использовать закрепленную память (листинги 12.1–12.6). Получилось у вас добиться улучшения производительности?
2. В примере редукции путем суммирования попробуйте массив размером 18 000 элементов, все из которых были изначально заданы в соответствии с индексным значением. Выполните исходный код CUDA, а затем версию в SumReductionRevealed. Возможно, вам захочется скорректировать объем печатаемой информации.
3. Конвертируйте пример редукции из CUDA в HIP путем его HIP'ифизации.
4. Для примера SYCL в листинге 12.20 инициализируйте массивы `a` и `b` на устройстве GPU.
5. Конвертируйте два цикла инициализации в примере RAJA в листинге 12.24 в синтаксис `Raja:forall`. Попробуйте выполнить пример с CUDA.

Резюме

- Используйте простые модификации изначального исходного кода CPU для большинства вычислительных ядер. Это упрощает написание вычислительных ядер и облегчает их техническое сопровождение.
- Тщательный дизайн кооперации и сравнения в вычислительных ядрах GPU может обеспечивать хорошую производительность. Ключом к подходу к этим операциям является разбиение алгоритма на шаги и понимание свойств производительности GPU.
- Думайте о переносимости с самого начала. Вам не придется создавать новые версии исходного кода всякий раз, когда вы захотите выполнить свое приложение на другой аппаратной платформе.
- Рассмотрите возможность применения языков обеспечения переносимости производительности с поддержкой единого исходного кода. Если вам нужно работать на различном оборудовании, то это может стоить первоначальных трудностей при разработке исходного кода.

13 Профилирование и инструменты GPU

Эта глава охватывает следующие ниже темы:

- инструменты профилирования для GPU;
- образец рабочего потока для этих инструментов;
- как использовать результаты работы инструментов профилирования GPU.

В этой главе мы рассмотрим инструменты и разные рабочие потоки, которые можно использовать для ускорения разработки приложений. Мы покажем вам аспекты, в которых инструменты профилирования GPU бывают полезны. В дополнение к этому мы обсудим вопрос преодолевания трудностей, связанных с использованием инструментов профилирования во время работы на дистанцированном (удаленном) кластере HPC. Поскольку инструменты профилирования продолжают меняться и совершенствоваться, мы сосредоточимся не на деталях какого-либо одного инструмента, а на методологии. Главный вывод этой главы будет заключаться в понимании того, как создавать продуктивный рабочий поток при использовании мощных инструментов профилирования GPU.

13.1 Обзор инструментов профилирования

Инструменты профилирования позволяют ускорять оптимизацию, совершенствовать объем полезного использования оборудования и глубже понимать производительность приложений и горячие точки. Мы обсудим вопрос того, как инструменты профилирования проявляют узкие места и помогают вам улучшать использование оборудования. В следующем ниже маркированном списке перечислены наиболее часто используемые инструменты профилирования GPU. Мы специально показываем инструменты NVIDIA для использования с их GPU-процессорами, потому что эти инструменты существуют дольше всего. Если в вашей системе есть GPU другого поставщика, то подставьте в рабочий процесс их инструменты. Не забывайте о стандартных инструментах профилирования Unix, таких как gprof, которые мы будем использовать позже в разделе 13.4.2.

Мы рекомендуем вам сверяться с приведенными в этой главе примерами. Прилагаемый исходный код находится по адресу http://github.com/EssentialsOfParallelComputing/Chapter13, где показаны примеры инсталлирования пакетов программно-информационного обеспечения для инструментов от разных поставщиков оборудования. Существуют подробные списки всего программно-информационного обеспечения, которое можно инсталлировать по каждому поставщику. Вероятно, вам захочется инсталлировать инструменты для соответствующего оборудования.

ПРИМЕЧАНИЕ Хотя инструмент другого поставщика, возможно, и будет частично работать в вашей системе, его полная функциональность будет нарушена.

- *NVIDIA nvidia-smi* – пытаясь получить быстрый профиль системы из командной строки, вы можете использовать инструмент nvidia-smi. Как показано и объяснено в разделе 9.6.2, NVIDIA SMI (System Management Interface, т. е. Интерфейс управления системой) позволяет мониторить и собирать данные мощности и температуры во время работы приложения. NVIDIA SMI предоставляет вам информацию об оборудовании наряду со многими другими системными метриками. Ссылка на руководство по SMI и варианты настройки приведены в разделе «Материалы для дальнейшего исследования» далее в этой главе.
- *NVIDIA nvprof* – этот консольный инструмент визуального профилирования NVIDIA Visual Profiler собирает и сообщает данные о производительности GPU. Данные для анализа производительности приложений также могут быть импортированы в инструмент в формате NVIDIA Visual Profiler NVVP либо в других форматах. Он показывает такие метрики производительности, как копии с оборудования на устройство и наоборот, потребление вычислительного ядра, объем полезного использования памяти и многие другие метрики.

- *NVIDIA NVVP* – этот инструмент визуального профилирования NVIDIA обеспечивает визуальное представление производительности вычислительного ядра приложения. NVVP предоставляет графический интерфейс и направляемый анализ. Он запрашивает те же данные, что и nvprof, но показывает данные пользователю в визуальном ключе, предлагая функциональность подвижной временной шкалы, которая не столь легко доступна в nvprof.
- *NVIDIA® Nsight™* – инструмент NSight представляет собой обновленную версию NVVP, которая обеспечивает визуальное представление потребления CPU и GPU и производительности приложений. Указанный инструмент в конечном счете может заменить NVVP.
- *NVIDIA PGPROF* – утилита PGPROF выросла из компилятора Portland Group. Когда NVIDIA приобрела Portland Group для своего компилятора Fortran, они объединили профилировщик Portland, PGPROF, с инструментами NVIDIA.
- *CodeXL* (*изначально AMD CodeXL*) – этот профилировщик GPUOpen, отладчик и тренажер разработчика программ изначально был разработан компанией AMD (см. ссылку на веб-сайт CodeXL в разделе «Дополнительное чтение» далее в этой главе).

13.2 Как выбрать хороший рабочий поток

Прежде чем приступать к любой сложной задаче, вы должны выбрать надлежащую последовательность рабочих процессов, или рабочий поток. Вы, возможно, будете работать на веб-сайте, имея отличное соединение, либо за пределами веб-сайта, имея медленную домашнюю сеть, либо где-то посередине. Для каждого случая требуется отдельный рабочий поток. В этом разделе мы обсудим четыре потенциальных и эффективных рабочих потока для этих разных сценариев.

На рис. 13.1 показано наглядное представление четырех разных рабочих потока. Доступность и скорость соединения являются определяющими факторами при принятии решения о том, какой метод вы в конечном итоге используете. Вы можете выполнять инструменты с графическим интерфейсом непосредственно в системе, дистанционно в режиме клиент-сервер, либо просто избегая проблем, использовать (консольные) инструменты командной строки.

При использовании инструментов профилирования с дистанционного сервера часто возникает большая задержка в визуализации и отклике графического интерфейса. Режим клиент–сервер разделяет графический интерфейс таким образом, чтобы он работал локально в вашей системе. Затем он обменивается данными с сервером на дистанционном веб-сайте с целью исполнения команд. Это помогает поддерживать интерактивную реакцию графического интерфейса инструмента. Например, такие инструменты профилирования, как NVVP, могут иметь высокую задержку при их использовании на дистанционном сервере.

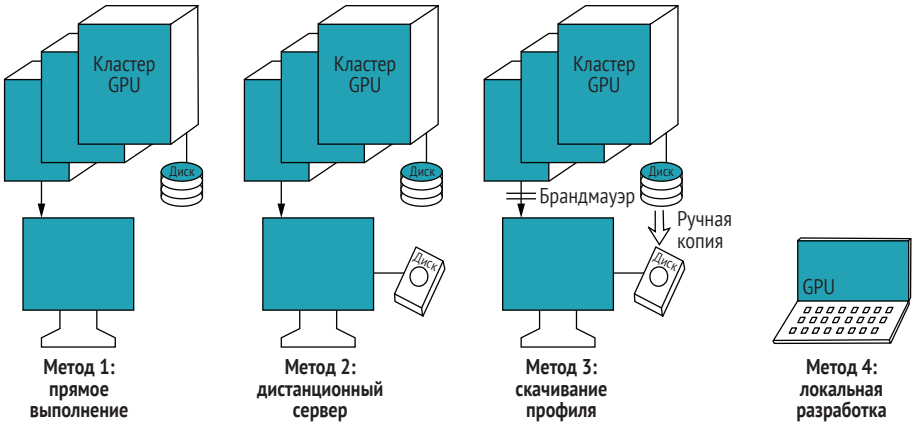


Рис. 13.1 Существует несколько разных методов использования инструментов профилирования, которые предоставляют альтернативы для вашей ситуации разработки приложений

Ситуация, когда приходится ожидать в течение минут после каждого щелчка мыши, не очень продуктивна. К счастью, инструменты NVIDIA и многие другие инструменты предоставляют вам несколько вариантов решения этой проблемы. Мы рассмотрим разные рабочие потоки подробнее в следующем ниже обсуждении.

- *Метод 1: выполнять непосредственно в системе.* Когда сетевое соединение для графического приложения является быстрым, этот метод становится предпочтительным, поскольку требования к хранению довольно велики. Если у вас есть быстрое соединение для вывода графики на экран, то этот способ работы будет самым эффективным. Но если ваше сетевое соединение является медленным, то время отклика графического окна становится томительно долгим, и вам захочется использовать один из дистанционных вариантов. VNC, X2Go и NoMachine могут сжимать графический вывод и вместо этого отправлять его, иногда делая медленные соединения работоспособными.
- *Метод 2: дистанционный сервер.* Этот метод выполняет приложение с помощью инструмента командной строки в системе GPU, затем файлы автоматически передаются в вашу локальную систему. Брандмауэры, пакетные операции системы HPC и другие сетевые осложнения иногда затрудняют либо делают невозможной организацию работы этого метода.
- *Метод 3: скачивание файла профиля.* В этом методе nvprof выполняется на веб-сайте HPC, и файлы скачиваются на ваш локальный компьютер. Здесь вы вручную переносите файлы на локальный компьютер с помощью утилиты защищенного копирования scp (от secure copy) или какой-либо другой утилиты, а затем работаете на локальном компьютере. При попытке профилировать несколько приложений бывает проще брать сырые данные в формате csv

и комбинировать их в один кадр данных. Хотя этот метод, возможно, больше не используется в обычных инструментах профилирования, вы имеете возможность выполнять свой собственный подробный анализ на сервере либо локально.

- *Метод 4: локальная разработка.* Одна из замечательных особенностей современного оборудования HPC заключается в том, что у вас нередко есть аналогичное оборудование, которое вы можете использовать для локальной разработки приложения. У вас может быть GPU того же производителя, но не такой мощный, как GPU в системе HPC. Вы можете оптимизировать свое приложение с расчетом на то, что в большой системе все будет работать быстрее. Вы также можете разрабатывать свой исходный код на CPU с помощью некоторых языков, в которых отладка осуществляется проще.

Важно понимать, что, даже если у вас нет быстрого соединения с вычислительным веб-сайтом, есть несколько вариантов при использовании инструментов разработки. Какой бы метод вы ни использовали для переноса и анализа производительности, вы должны убедиться, что версии используемого вами программно-информационного обеспечения совпадают. Это особенно важно для инструментов CUDA и NVIDIA nvprof и NVVP.

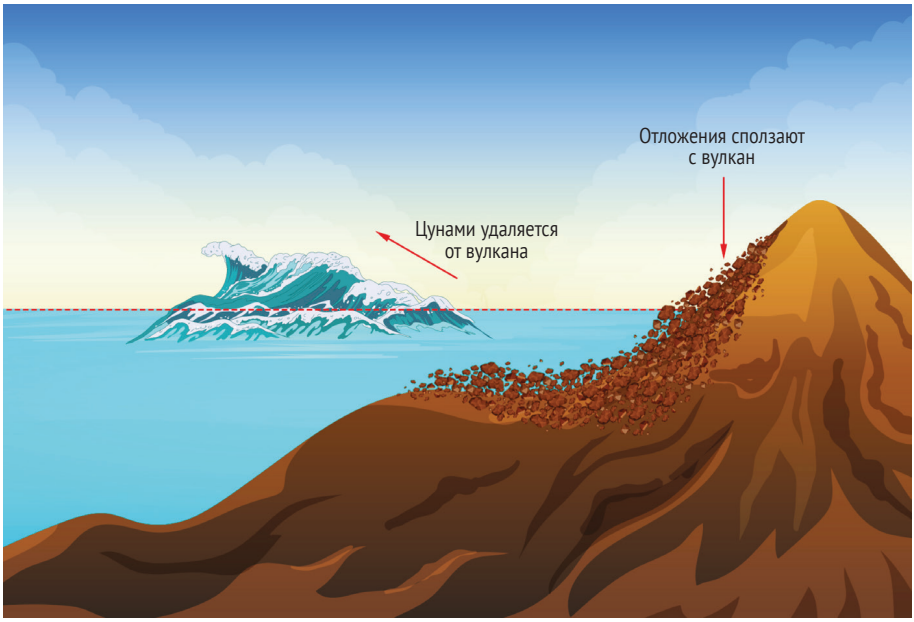
13.3 Образец задачи: симуляция мелководья

В этом разделе мы поработаем с реалистичным примером, чтобы показать процесс переноса исходного кода и использование нескольких инструментов. Мы будем использовать задачу из рис. 1.9, где извержение вулкана или землетрясение могут вызывать распространение цунами. Цунами могут преодолевать тысячи миль через океаны, имея высоту всего несколько футов, но, когда они добираются до берега, они могут достигать сотен метров в высоту. Такого рода симуляции обычно выполняются после события из-за времени, необходимого для настройки и выполнения задачи. Мы бы предпочли симулировать его в реальном времени, чтобы иметь возможность предупреждать тех, кто может пострадать. Ускорение симуляции за счет ее выполнения на GPU могло бы обеспечить такую способность.

Сначала мы рассмотрим физику, которая возникает в этом сценарии, а затем переведем ее в уравнения, чтобы просимулировать задачу численно. Мы хотим представить конкретный сценарий, а именно обрушение большой массы острова или другого массива суши, который падает в океан, как показано на рис. 13.2. Это событие на самом деле произошло с необитаемым вулканическим островом в Индонезии Анак-Кракатау («Дитя Кракатау») в декабре 2018 года.

В этом декабрьском событии объем оползня на западном склоне острова Кракатау составил около 0.2 км^3. Он оказался меньше, чем предполагалось в предыдущих прогнозах риска. Вдобавок высота волн, по

оценкам, составляла более 100 м. Из-за короткого расстояния от источника до берега для находящихся в этом районе людей было мало предупреждений, и, став причиной более чем 400 смертей, это событие получило освещение в новостях по всему миру.



Графика любезно предоставлена Кристианом Гомесом (Cristian Gomez)

Рис. 13.2 Волна цунами, произошедшая на Анак-Кракатау 22 декабря 2018 года, была вызвана оползнем отложений с вулканического острова

Ученые провели большое число симуляций до наступления этого события и еще больше после него. Вы можете посмотреть несколько визуализаций и анализ указанного события по адресу http://mng.bz/4Mqw. Как проводились симуляции? Необходимая базовая физика представляет лишь небольшой шажок в сложности стенсильных вычислений, которые мы рассматривали в этой книге. Полноценный исходный код симуляции потребовал бы гораздо более изощренных танцев с бубнами, но нам надолго хватит и простой физики. Итак, давайте взглянем на необходимую физику, лежащую в основе симуляций.

Математические уравнения для цунами относительно просты. Это сохранение массы и сохранение импульса (моментума). Последнее, в сущности, является первым законом движения Ньютона: «Объект в состоянии покоя остается в состоянии покоя, а объект в движении остается в движении». В уравнении импульса используется второй закон движения: «Сила равна изменению в импульсе». Для уравнения сохранения массы мы, в сущности, имеем, что изменение в массе для вычислительной ячейки за малый промежуток времени равно сумме массы, пересекающей границы ячейки, как показано ниже:

$$\frac{\partial M}{\partial l} + \frac{\partial (v_x M)}{\partial x} + \frac{\partial (v_y M)}{\partial y} = 0 \quad \text{(Сохранение массы)},$$

где $\partial M/\partial l$ – это изменение в массе относительно времени, и $v_x M/\partial x$ и $v_y M/\partial y$ – флюксия массы (векторная скорость × масса) по граням x и y. Кроме того, поскольку вода несжимаема, плотность воды можно рассматривать как постоянную. Масса ячейки равна объему × плотность. Если у нас ячейки имеют размер 1×1 м, то объем равен высоте × 1 м × 1 м. Если свести все это воедино, то все является постоянным, за исключением высоты, поэтому вместо массы мы можем подставить переменную высоты:

Масса = Объем × Плотность =
Высота ×1 м × 1 м × Плотность = Постоянная × Высота.

Также используя $u = v_x$ и $v = v_y$, мы теперь получаем стандартную форму закона сохранения для уравнений мелководья:

$$\frac{\partial h}{\partial l} + \frac{\partial (hu)}{\partial x} + \frac{\partial (hv)}{\partial y} = 0 \quad \text{(Сохранение массы)}.$$

Сохранение импульса аналогично, где вместо массы или высоты подставляется импульс (моментум). Мы показываем только члены x, чтобы уместить уравнение на странице, как показано ниже:

$$\frac{\partial (mom_x)}{\partial l} + \frac{\partial}{\partial x}(v_x \cdot mom_x) + \frac{\partial}{\partial x}\left(\frac{1}{2} g h^2\right) = 0 \quad \text{(Сохранение импульса } x\text{)}.$$

Дополнительный член $1/2gh^2$ обусловлен работой, оказываемой на систему силой тяжести (гравитацией). Согласно второму закону Ньютона, внешняя сила создает дополнительный импульс ($F = ma$). Мы посмотрим на возникновение этого члена с использованием исчисления и без него. Прежде всего ускорение в этом случае представлено силой тяжести, и оно вызывает силу, действующую на столб воды, как показано на рис. 13.3. Каждый дополнительный метр высоты воды создает так называемое *гидростатическое давление*, приводящее к повышению давления вдоль всего столба воды. Используя исчисление, мы бы проинтегрировали давление вдоль столба и получили бы создаваемый импульс. Это интегрирование по вертикали ствола (z) от 0 до высоты волны (h) выражается следующим образом:

$$p = \int_0^h gz\,dz = \frac{1}{2} g z^2 \Big|_0^h = \frac{1}{2} g h^2 \quad \text{(Интегрировать силу по глубине } z\text{)}.$$

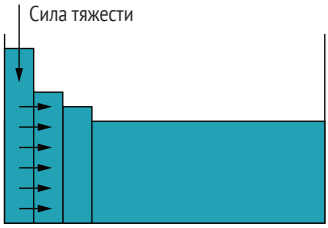


Рис. 13.3 Сила тяжести, действующая на столб воды, создает поток и импульс

Существует также гораздо более простое выведение формулы. В этом случае давление является линейной функцией (рис. 13.4). Если мы посмотрим на срединную точку высоты, а затем применим разницу давлений в этой срединной точке высоты ко всему столбу, то мы сможем получить то же решение. В данном случае мы суммируем все силы давления под кривой. На языке математики это суммирование есть не что иное, как интегрирование функции или выполнение суммы Римана, где вы разбиваете область под кривой на столбцы, а затем их складываете. Но все это будет уже перебором. Область под кривой представляет собой треугольник, и мы можем использовать площадь треугольника или $A = 1/2\,bh$.

$$p = mg \cdot h_{midpoint} = hg \cdot \frac{h}{2} = \frac{1}{2} gh^2$$ (Использование гидростатического давления в срединной точке высоты).

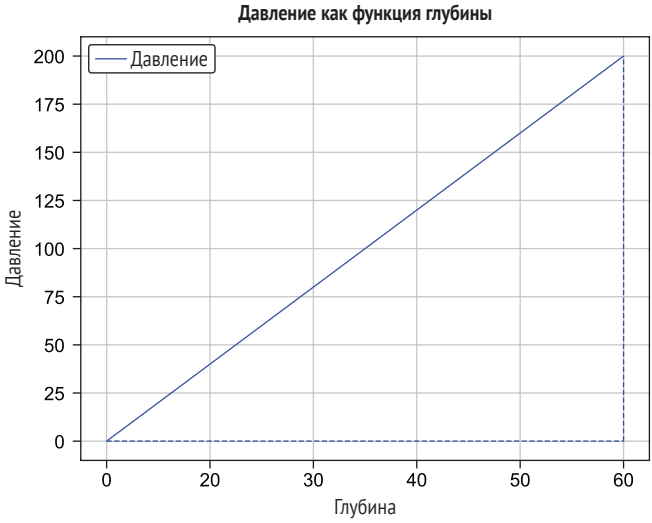


Рис. 13.4 Гидростатическое давление, вызванное силой тяжести, является линейной функцией глубины

Наш результирующий набор уравнений таков:

$$\frac{\partial h}{\partial l} + \frac{\partial (hu)}{\partial x} + \frac{\partial (hv)}{\partial y} = 0 \quad \text{(Сохранение массы);}$$

$$\frac{\partial (hu)}{\partial l} + \frac{\partial}{\partial x}\left(hu^2 + \frac{1}{2}gh^2\right) + \frac{\partial}{\partial y}(huv) = 0 \quad \text{(Сохранение импульса } x\text{);}$$

$$\frac{\partial (hv)}{\partial l} + \frac{\partial}{\partial x}(hvu) + \frac{\partial}{\partial y}\left(hv^2 + \frac{1}{2}gh^2\right) = 0 \quad \text{(Сохранение импульса } y\text{).}$$

Если вы наблюдательны, то обратите внимание на перекрестные члены импульсных флюксов для импульса *y* в уравнении импульса *x* и импульса *x* в уравнении импульса *y*. При сохранении импульса *x* третий член имеет импульс *x* (*hu*), который движется по грани *y* с векторной скоростью *y* (*v*). Это можно описать как адвекцию (перенос), или флюксию, импульса *x* со скоростью в направлении *y* по верхней и нижней граням вычислительной ячейки. Флюксия импульса *x* (*hu*) по граням *x* с векторной скоростью *u* находится во втором члене как hu^2.

Мы также видим, что только что созданный импульс распределяется по двум уравнениям импульса с новым импульсом *x* в уравнении импульса *x* и импульсом *y* в уравнении импульса *y*. Затем эти уравнения имплементируются в виде трех стенсильных операций в нашем исходном коде симуляции мелководья, где для простоты мы используем $H = h$, $U = hu$ и $V = hv$. Теперь у нас есть простое научное приложение, которое можно использовать для демонстраций.

У нас есть еще одна деталь имплементации. Мы используем численный метод, который оценивает такие свойства, как масса и импульс на гранях каждой ячейки в середине временного шага. Затем используем эти оценки для расчета количества массы и импульса, которые перемещаются в ячейку во время временного шага. Благодаря этому мы получаем немного больше точности в численном решении.

Поздравляю, если вы продрались через дебри этого изложения и достигли некоторого понимания. Теперь вы увидели, как мы берем простые законы физики и создаем на их основе научное приложение. Вы всегда должны стремиться понимать физику и численный метод, которые лежат в основе задачи, а не рассматривать исходный код как набор циклов.

13.4 Образец профилировочного рабочего потока

Далее мы переходим к шагу профилирования приложения симуляции мелководья. Для этого было создано приложение симуляции мелководья на основе математических и физических уравнений, представленных в разделе 13.3. Во многих отношениях исходный код представляет собой всего лишь три стенсильных вычисления уравнения массы и двух

уравнений импульса. Начиная с главы 1, мы работали с одним простым стенсильным уравнением, и пример его исходного кода находится по адресу https://github.com/EssentialsofParallelComputing/Chapter13.

13.4.1 Выполнение приложения симуляции мелководья

В этом разделе мы покажем вам, как выполнять исходный код приложения симуляции мелководья. Мы будем использовать этот исходный код для пошагового анализа образца рабочего потока переноса вашего исходного кода на GPU. Сперва дадим несколько замечаний о платформах:

- *macOS* – NVIDIA предупреждает о том, что CUDA 10.2 может быть последней версией, поддерживающей macOS, причем поддерживающей ее только через macOS v10.13. В результате NVVP поддерживается только посредством macOS v10.13. Все вроде как работает с версией v10.14, но полностью выходит из строя в версии v10.15 (Каталина). Мы предлагаем использовать VirtualBox (https://www.virtualbox.org) в качестве бесплатной виртуальной машины, чтобы опробировать инструменты в системах Mac. Мы также предоставили контейнер Docker для macOS;
- *Windows* – NVIDIA по-прежнему поддерживает Microsoft Windows нативно, но, если хотите, вы также можете использовать контейнеры VirtualBox или Docker в Windows;
- *Linux* – прямая инсталляция в большинстве систем Linux должна работать.

Если у вас в локальной системе есть GPU, то вы можете использовать локальный рабочий поток. Если нет, то вы, вероятно, будете работать дистанционно в вычислительном кластере и передавать файлы обратно для анализа.

Если вы хотите использовать графику, то вам нужно будет инсталлировать несколько дополнительных пакетов. В системе Ubuntu это можно сделать с помощью следующих ниже команд. Первая команда предназначена для инсталлирования OpenGL и freeglut для реально-временной графики. Вторая служит для инсталлирования ImageMagick® для обработки вывода данных в графические файлы, которые мы можем использовать для графических снимков. Графические снимки также могут быть конвертированы в видеоролики. Файл README.graphics в каталоге GitHub содержит дополнительную информацию о графических форматах и скрипты в примерах, сопровождающих эту главу.

```
sudo apt-get install libglu1-mesa-dev freeglut3-dev mesa-common-dev -y
sudo apt install cmake imagemagick libmagickwand-dev
```

Мы обнаружили, что реально-временная графика может ускорять разработку и отладку исходного кода, поэтому включили пример ее использования в образцы исходного кода, прилагаемого к этой главе. Например, реально-временная графика использует OpenGL для вывода на экран высоты воды в вычислительной сетке, обеспечивая немедленный

визуальный отклик в качестве обратной связи. Исходный код с реально-временной графикой можно также легко расширить в части реагирования на взаимодействие с клавиатурой и мышью внутри окна реально-временной графики.

Этот пример закодирован с помощью OpenACC, поэтому лучше всего использовать компилятор PGI. Ограниченное подмножество примеров работает с компилятором GCC по причине его все еще разрабатываемой поддержки OpenACC. Компилирование примера исходного кода выполняется прямолинейно. Для этого мы просто используем CMake и make.

1 Для того чтобы собрать makefile, следует набрать:

```
mkdir build && cd build
cmake ..
```

2 Для того чтобы задействовать графику, набрать:

```
cmake -DENABLE_GRAPHICS=1
```

3 Задать формат графического файла с помощью:

```
export GRAPHICS_TYPE=JPEG
make
```

4 Затем выполнить последовательный код с помощью:

```
./ShallowWater
```

Если вам не удается привести графический вывод в рабочее состояние, то эта программа будет нормально работать и без него. Но если настроить его правильно, то реально-временная графика выведет на экран графическое окно, подобное тому, которое показано на рис. 13.5. Графика обновляется каждые 100 итераций. Здесь на рисунке показана вычислительная сетка меньшего размера, чем жестко заданный размер в образце исходного кода. Линии представляют вычислительные ячейки, высота волны в которых выше слева. Волна движется вправо с высотой, уменьшающейся по мере ее движения. Волна пересекает вычислительный домен и отражается от правой грани. Затем она перемещается взад и вперед по вычислительной сетке. В реальном расчете вычислительная сетка содержала бы объекты (например, береговые линии).

Если у вас есть система, которая может выполнять OpenACC, то также будут собраны исполняемые файлы ShallowWater_par1-ShallowWater_par4. Вы можете использовать их для последующих упражнений по профилированию.

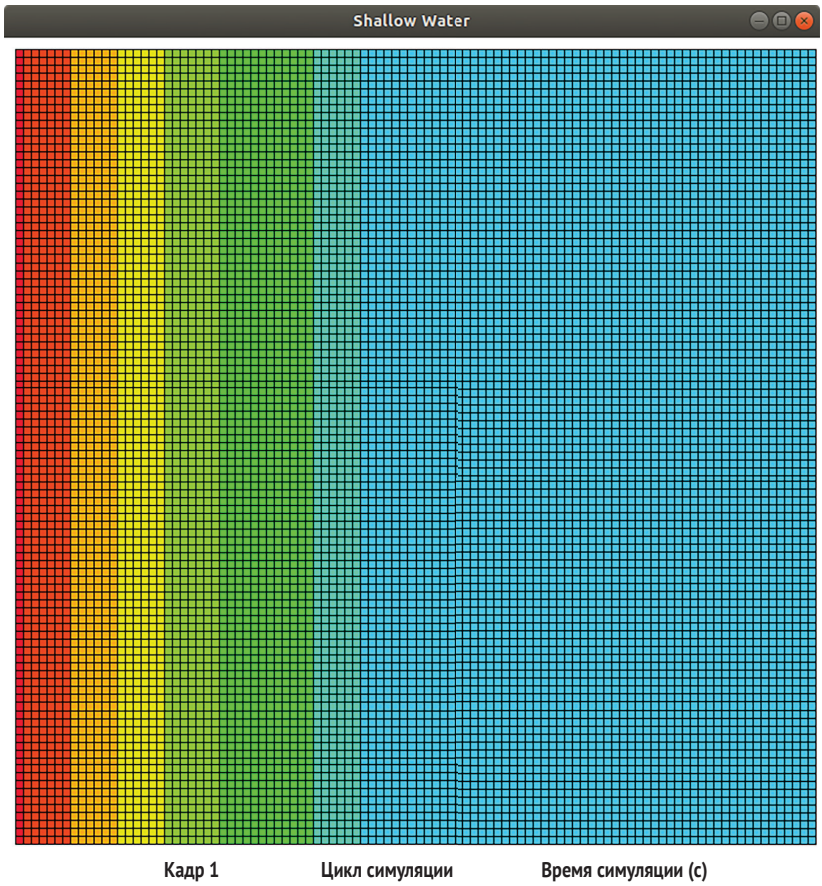


Рис. 13.5 Реально-временная графика на выходе из приложения симуляции мелководья. Красные полосы слева указывают на начало волны, где оползень входит в воду. Волна движется вправо, пересекая океан: оранжевый, желтый, зеленый и синий цвета. Если вы читаете это в черно-белом формате, то левый заштрихованный участок соответствует красному цвету, а крайний правый заштрихованный участок соответствует синему. Линии – это контуры вычислительных ячеек

13.4.2 Профилирование исходного кода CPU для разработки плана действий

Мы описали цикл параллельной разработки еще в главе 2 следующим образом:

1. профилирование;
2. планирование;
3. имплементирование;
4. фиксирование.

Первым шагом является профилирование нашего приложения. Для большинства приложений мы рекомендуем использовать высокоуровневый профилировщик, такой как инструмент Cachegrind, который мы представили в разделе 3.3.1. Cachegrind показывает наиболее трудоемкие пути в исходном коде и выводит результаты на экран в удобном для интерпретирования наглядном виде. Однако для такой простой программы, как приложение симуляции мелководья, профилировщики уровня функций, такие как Cachegrind, неэффективны. Cachegrind показывает, что 100 % времени тратится на главную функцию, что нам не очень помогает. В нашей конкретной ситуации нам нужен построчный профилировщик. Для этой цели мы используем самый известный профилировщик в системах Unix – gprof. Позже, когда у нас будет исходный код, работающий на GPU, будем использовать инструмент профилирования NVIDIA NVVP, чтобы получать статистику производительности. Для начала просто нужен простой инструмент для профилирования приложения, работающего на CPU.

Пример: профилирование с помощью gprof

1. Отредактировать CMakeLists.txt, добавив флаг `-pg` в компиляторные флаги (разностный вывод diff выделяет изначальную строку в CMakeLists символом – и новую строку символом +):

```
-set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -g -O3")
+set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -g -O3 -pg")
```

2. Отредактировать ShallowWater.c и увеличить размер вычислительной сетки:

```
- int       nx = 500, ny = 200;
+ int       nx = 5000, ny = 2000;
```

3. Перестроить исполняемый файл ShallowWater, набрав make.
4. Выполнить исполняемый файл ShallowWater, набрав `./ShallowWater`. Вы должны получить выходной файл под названием gmon.out.
5. Выполнить шаг постобработки, набрав `gprof -l -pg ./ShallowWater`.

Результат на выходе из gprof показывает циклы, которые занимают большую часть времени в приложении симуляции мелководья (см. следующий ниже рисунок).

Каждый образец рассчитывается как 0.01 с.

% time	cumulative seconds	self seconds	self calls	total Ts/call	Ts/call name
42.95	140.38	140.38			main (ShallowWater.c:207 @ 401885)
22.44	213.71	73.33			main (ShallowWater.c:190 @ 401730)
22.34	286.74	73.03			main (ShallowWater.c:172 @ 401500)
12.06	326.17	39.43			main (ShallowWater.c:160 @ 401330)

< ... остальные результаты ...>

Результат на выходе из gprof. Цикл в строке 207 занимает бо́льшую часть времени и станет хорошей отправной точкой для переноса на GPU

Мы смотрим циклы для каждого номера строки в результатах профилирования на приведенном выше рисунке и обнаруживаем, что они соответствуют следующим операциям:

- `ShallowWater.c:207` (цикл второго прохождения);
- `ShallowWater.c:190` (прохождение грани *y*);
- `ShallowWater.c:172` (прохождение грани *x*);
- `ShallowWater.c:160` (расчет временного шага).

Это говорит нам о том, что мы должны сосредоточить наши первоначальные усилия на вычислении второго прохождения в конце главного цикла вычислений и продвигаться к вершине цикла. Существует тенденция пытаться делать все сразу, но более безопасный подход заключается в том, чтобы работать от цикла к циклу и обеспечивать, чтобы результат был по-прежнему правильным. Если сначала сосредоточиться на самых дорогих циклах, то некоторое повышение производительности будет достигнуто раньше.

13.4.3 Добавление вычислительных директив OpenACC, чтобы начать шаг имплементации

Теперь, когда мы закончили профилировать приложение и разрабатывать план, на следующем шаге цикла параллельной разработки мы приступаем к имплементированию плана. На этом шаге мы начинаем долгожданную модификацию исходного кода.

Имплементация начинается с переноса исходного кода на GPU путем перемещения цикла компиляции. Мы выполняем ту же процедуру переноса исходного кода на GPU, что и в разделе 11.2.2. Вычисления перемещаются путем вставки прагмы `acc parallel loop` перед каждым циклом, как показано в строке 95 в следующем ниже листинге.

Листинг 13.1 Добавление директивы цикла

`OpenACC/ShallowWater/ShallowWater_par1.c`

```
 95 #pragma acc parallel loop
 96       for(int j=1;j<=ny;j++){
 97         H[j][0]=H[j][1];
 98         U[j][0]=-U[j][1];
 99         V[j][0]=V[j][1];
100         H[j][nx+1]=H[j][nx];
101         U[j][nx+1]=-U[j][nx];
102         V[j][nx+1]=V[j][nx];
103       }
```

Нам также необходимо заместить обмен указателями в строке 191 в конце цикла копированием данных. Это не идеальное решение, пото-

му что оно приводит к большему перемещению данных и осуществляется медленнее, чем обмен указателями. Тем не менее выполнять обмен указателями в OpenACC сложно, потому что указатели на хосте и устройстве должны переключаться одновременно.

Листинг 13.2 Замещение обмена указателями копированием

OpenACC/ShallowWater/ShallowWater_par1.c

```
189         // Необходимо заместить обмен копированием
190 #pragma acc parallel loop
191        for(int j=1;j<=ny;j++){
192          for(int i=1;i<=nx;i++){
193             H[j][i] = Hnew[j][i];
194             U[j][i] = Unew[j][i];
195             V[j][i] = Vnew[j][i];
196          }
197        }
```

Вы получите более точный отклик о производительности вашего приложения из визуального представления. На каждом шаге процесса мы выполняем инструмент профилирования NVVP, чтобы получать трассу производительности в графическом виде.

Пример: получение визуального профиля производительности с помощью визуального профилировщика NVIDIA (NVVP)

В целях получения визуальной временной шкалы производительности мы выполняем исходный код с помощью команды:

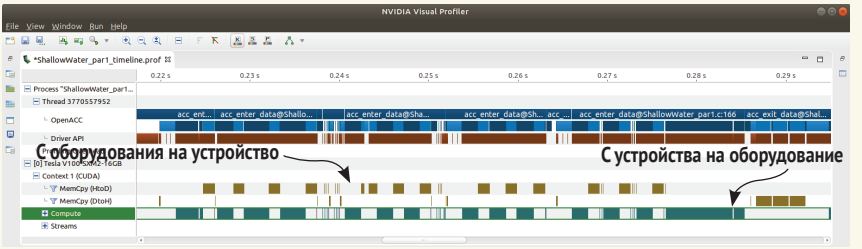
```
nvprof --export-profile ShallowWater_par1_timeline.prof
            ./ShallowWater_par1
```

Использование команды `nvprof` сохраняет профилировочную временную шкалу в текущем каталоге:

```
nvvp ShallowWater_par1_timeline.prof
```

Затем команда nvvp импортирует профиль в комплект NVIDIA Visual Profiler, получая графический вывод, показанный на следующем ниже рисунке. Если хотите, вы можете копировать профиль обратно на свой локальный компьютер в промежутке между двумя шагами и просматривать его локально.

Сначала рассмотрим визуальный профиль, чтобы получить оперативное представление в цветовой кодировке об относительной производительности наших копий памяти и вычислительных ядер. Указанная временная шкала показана в верхней части окна визуального профилировщика. В этом месте обратите особое внимание на строки MemCpy (HtoD) и MemCpy (DtoH), где показывается передача данных с хоста на устройство и с устройства на хост. Панели направляемого анализа и сведений об OpenACC, расположенные в нижней части этого окна, обсуждаются в разделе 13.4.5.



Результат работы профилировщика NVIDIA NVVP показывает временную шкалу одного вычислительного цикла. Вы видите копии памяти между устройством и оборудованием и наоборот. В выделенной строке результат также показывает вычислительные участки

Если ваше сетевое соединение не позволяет использовать графический инструмент напрямую либо передавать данные профиля на компьютер, то вы всегда можете вернуться к использованию nvprof в текстовом режиме. Та же информация может быть получена из текстовых выходных данных, но всегда есть некоторые наблюдения, которые становятся яснее при визуальном представлении.

На рис. 13.6 показана возможность зумирования конкретных вычислительных ядер для более точной идентификации метрик производительности в некоторых вычислительных циклах. В частности, мы увеличили масштаб строки 95 из листинга 13.1, чтобы показать отдельные копии памяти.

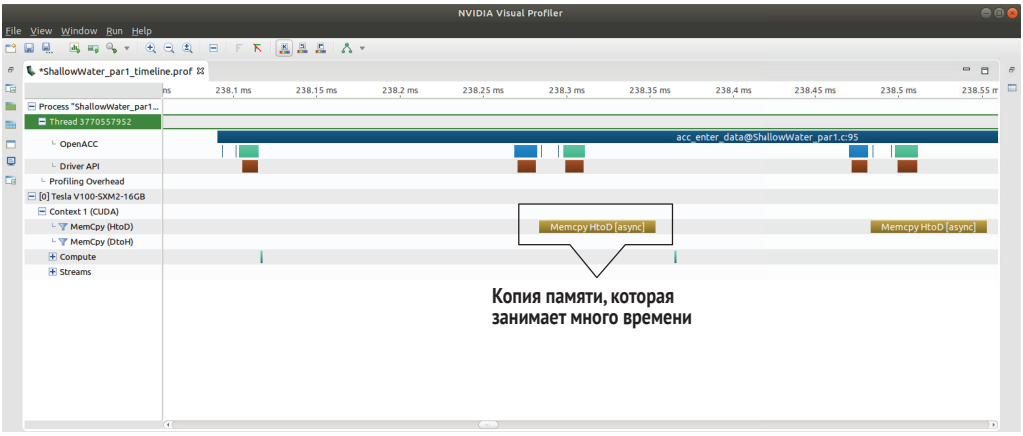


Рис. 13.6. С помощью NVIDIA NVVP можно увеличивать масштаб конкретных копий в режиме временной шкалы. Здесь вы видите увеличенную версию индивидуальных копий памяти в каждом цикле. За счет зумирования вы получаете возможность видеть строки, в которых они находятся, помогая вам легко ориентироваться в приложении

13.4.4 Добавление директив перемещения данных

Следующим шагом в переносе исходного кода на GPU является добавление директив перемещения данных. Это позволяет нам еще больше повысить производительность приложений за счет устранения дорогостоящих копий памяти. В данном разделе мы покажем вам, как это делается.

Визуальный профилировщик, NVVP, помогает понимать, где нам нужно сосредотачивать усилия. Начните с поиска крупных временных блоков MemCpy и их поочередного устранения. По мере удаления затрат на передачу данных ваш код начнет показывать ускорение, восстанавливая производительность, потерянную во время применения вычислительных директив в разделе 13.4.4.

В листинге 13.3 мы показываем пример добавленных нами директив перемещения данных. В начале раздела данных используем директиву `acc enter data create` для маркировки начала участка динамических данных. Как следствие, данные будут существовать на устройстве до тех пор, пока мы не столкнемся с директивой `acc exit data`. Для каждого цикла добавляем выражение `present`, чтобы сообщить компилятору о том, что данные уже находятся на устройстве. Обратитесь к примеру кода главы 13 в файле OpenACC/ShallowWater/ShallowWater_par2.c касательно всех изменений, внесенных с целью управления перемещением данных.

Листинг 13.3 Директивы перемещения данных

OpenACC/ShallowWater/ShallowWater_par2.c

```
51 #pragma acc enter data create( \
52         H[:ny+2][:nx+2],    U[:ny+2][:nx+2],    V[:ny+2][:nx+2], \
53         Hx[:ny][:nx+1],     Ux[:ny][:nx+1],     Vx[:ny][:nx+1], \
54         Hy[:ny+1][:nx],     Uy[:ny+1][:nx],     Vy[:ny+1][:nx], \
55         Hnew[:ny+2][:nx+2], Unew[:ny+2][:nx+2], Vnew[:ny+2][:nx+2])
   <...>
59   #pragma acc parallel loop present( \
60         H[:ny+2][:nx+2], U[:ny+2][:nx+2], V[:ny+2][:nx+2])
```

Применение директив перемещения данных из листинга 13.3 и повторное выполнение профилировщика дают нам новые результаты производительности на рис. 13.7, где можно видеть сокращение перемещения данных. За счет сокращения времени передачи данных совокупное время работы приложения значительно сокращается. В более крупном приложении вам следует продолжить поиск других операций передачи данных, которые затем можно устранить, чтобы ускорить исходный код еще больше.

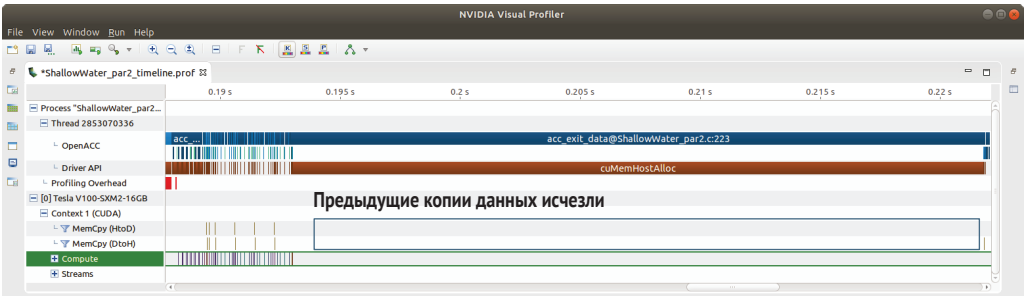


Рис. 13.7 Эта временная шкала из визуального профилировщика NVIDIA NVVP показывает четыре итерации вычисления, но теперь с оптимизацией перемещения данных. В этом рисунке интересно не столько то, что там есть, сколько то, чего там нет. Движение данных, которое происходило на предыдущем рисунке, резко сократилось либо больше не существует

13.4.5 Направляемый анализ может дать вам несколько предлагаемых улучшений

Для получения более глубокой информации инструмент NVVP предоставляет функциональность направляемого анализа (рис. 13.8). В этом разделе мы обсудим вопрос о том, как использовать эту функциональность.

Вы должны оценивать предложения направляемого анализа, основываясь на ваших знаниях о своем приложении. В нашем примере у нас мало передач данных, поэтому мы не в состоянии получить наложение копии памяти и вычислений, упомянутое в верхнем предложении о низком наложении Memcpy/вычислений на рис. 13.8. Это относится и к большинству других предложений. Например, при низком уровне конкурентности вычислительного ядра у нас есть только одно вычислительное ядро, поэтому у нас не может быть конкурентности. Хотя наше приложение невелико и, возможно, не нуждается в этих дополнительных оптимизациях, их следует принять на заметку, поскольку они бывают полезны для более крупных приложений.

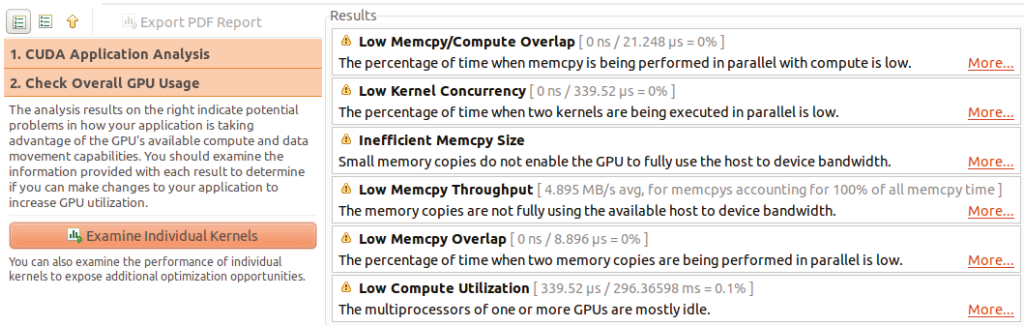


Рис. 13.8 NVVP также предлагает раздел направляемого анализа. Здесь пользователь может получать подсказки о дальнейшей оптимизации. Обратите внимание, что выделенный участок показывает низкий объем полезного использования вычислений

Вдобавок на рис. 13.8 показан низкий для выполнения нашего приложения объем полезного использования вычислений. В этом нет ничего необычного. Такой низкий объем полезного использования GPU в большей степени свидетельствует об огромной вычислительной мощности, доступной на GPU, и о том, насколько больше он способен сделать. Кратко возвращаясь к измерениям производительности и анализу производительности нашим инструментом mixbench (раздел 9.3.4), мы имеем лимитированное по пропускной способности вычислительное ядро, поэтому в лучшем случае будем использовать 1-2 % способности GPU с плавающей точкой. В свете этого 0.1%-ный объем полезного использования вычислений не так уж плох.

Еще одной функциональностью инструмента NVVP является окно сведений об OpenACC, в котором указываются хронометражи каждой операции. Его лучше всего использовать, получая хронометражи до и после, как показано на рис. 13.9. Сопоставительные сравнения в режиме бок о бок дают вам четкое представление об улучшении от директив перемещения данных.

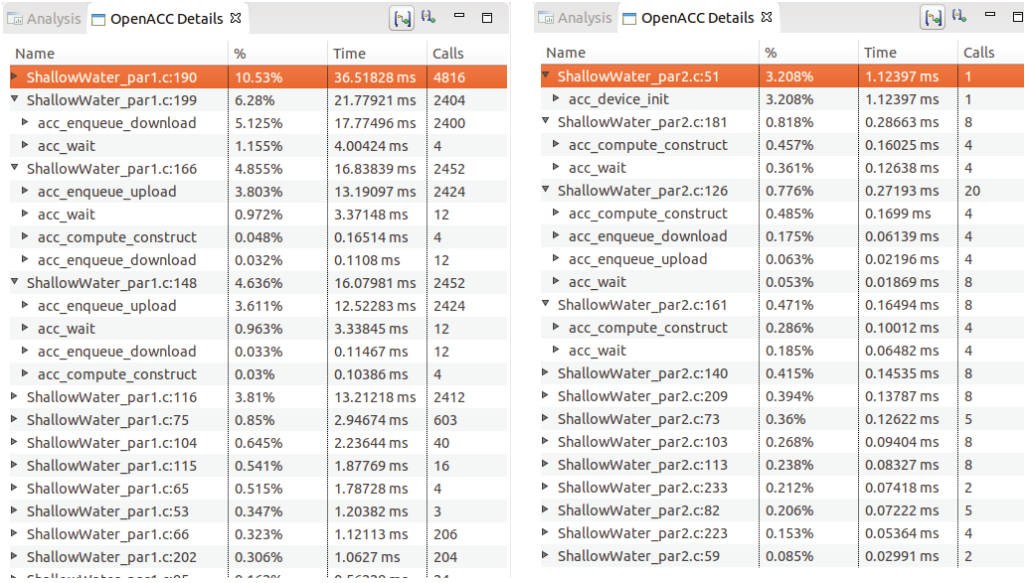


Рис. 13.9 Окно сведений об OpenACC в инструменте NVVP показывает информацию о каждом вычислительном ядре OpenACC и затратах на каждую операцию. Затраты на передачу данных можно увидеть в левом окне для версии 1 исходного кода по сравнению со временем для оптимизированного перемещения данных в версии 2 справа

Когда откроется окно сведений об OpenACC, вы заметите, что номера строк перемещаются внутри профиля. Если мы посмотрим на строку 166 в листинге ShallowWater_par1 (слева на рис. 13.10), то она будет занимать 4.8 % времени выполнения. Раскладка операций показывает, что большая часть этого времени связана с затратами на передачу данных. Соответствующая строка исходного кода в листинге ShallowWater_par2 име-

ет номер 181 (справа на рис. 13.10) и содержит добавление выражения данных `present`. Мы видим, что время для строки 181 составляет всего 0.81 % и что это в значительной степени связано с устранением затрат на передачу данных. Вычислительный конструкт в обоих случаях занимает примерно одинаковое время в размере 0.16 мс, как показано в строке с надписью `acc_compute_construct` чуть ниже выделенной строки.

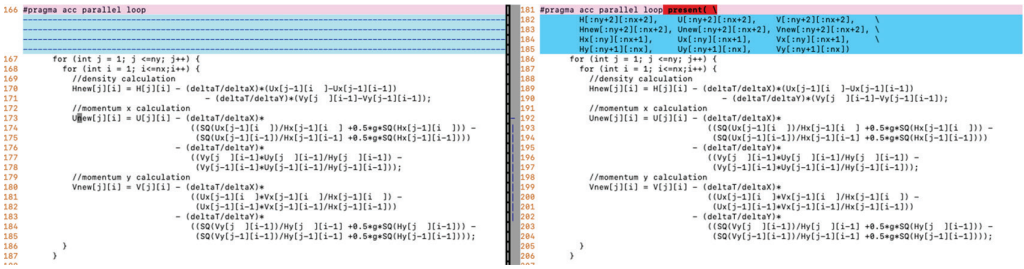


Рис. 13.10 Сопоставительное сравнение исходных кодов, показывающее, что строка 166 в версии 1 исходного кода симуляции мелководья теперь является строкой 181, в которой есть дополнительное выражение present

13.4.6 Комплект инструментов NVIDIA Nsight может стать мощным подспорьем в разработке

NVIDIA заменяет свои инструменты визуального профилировщика Visual Profiler (NVVP и nvprof) комплектом инструментов Nsight™. Указанный комплект инструментов опирается на две интегрированные среды разработки (IDE).

1. Комплект Nsight в редакции для Visual Studio (Nsight Visual Studio Edition) поддерживает разработку CUDA и OpenCL в среде Microsoft Visual Studio IDE.
2. Комплект Nsight в редакции для Eclipse (Nsight Eclipse Edition) добавляет язык CUDA в популярную среду разработки Eclipse с открытым исходным кодом.

На рис. 13.11 наше приложение симуляции мелководья показано в инструменте разработки Nsight Eclipse Edition.

Комплект инструментов Nsight также содержит компоненты с одной функцией, которые могут скачиваться зарегистрированными разработчиками NVIDIA. Эти профилировщики включают в себя функциональность визуального профилировщика NVIDIA и привносят дополнительные способности. Указанные два компонента таковы:

- Nsight Systems, инструмент повышения производительности системного уровня, обращается к совокупному перемещению данных и вычислениям;
- Nsight Compute, инструмент повышения производительности, дает подробное представление о производительности вычислительного ядра GPU.

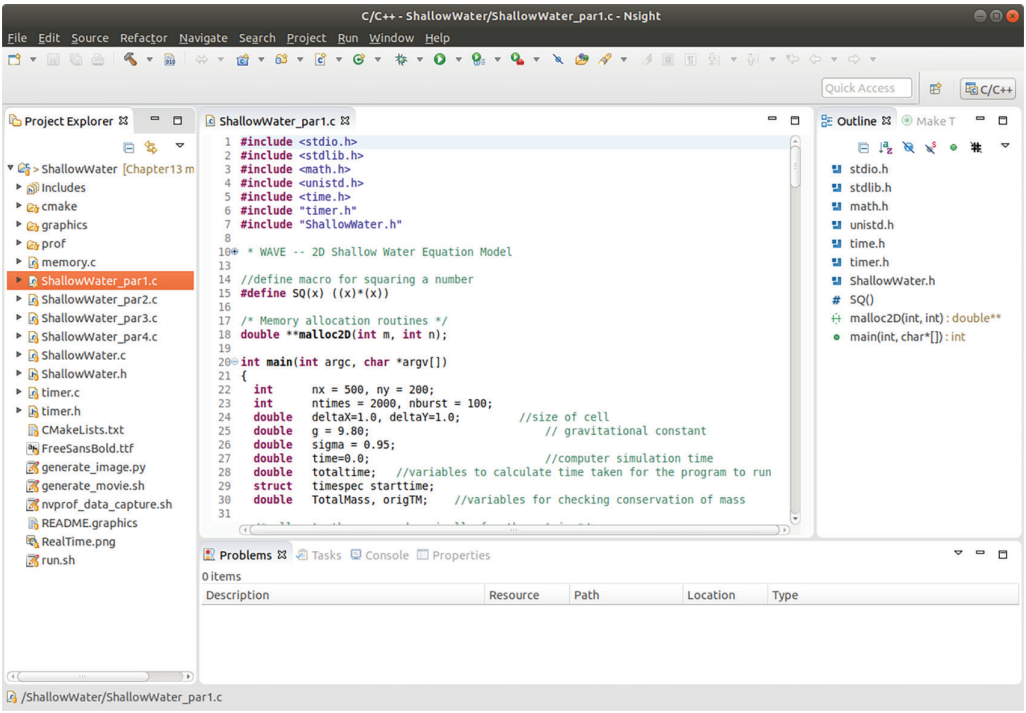


Рис. 13.11 Приложение NVIDIA Nsight Eclipse Edition – это инструмент для разработки исходного кода. В этом окне инструмента показано приложение ShallowWater_par1

13.4.7 CodeXL для экосистемы GPU-процессоров AMD

AMD тоже обладает способностями разработки исходного кода и анализа производительности в своем комплекте инструментов CodeXL. Как показано на рис. 13.12, указанный инструмент разработки приложений представляет собой полнофункциональный тренажер для работы с исходным кодом. CodeXL также содержит компонент профилирования (в меню «Профиль»), который помогает оптимизировать исходный код для GPU AMD.

Указанные новые инструменты от NVIDIA и AMD все еще разворачиваются. Доступность полнофункциональных инструментов, включая отладчики и профилировщики, станет огромным стимулом для разработки исходного кода GPU.

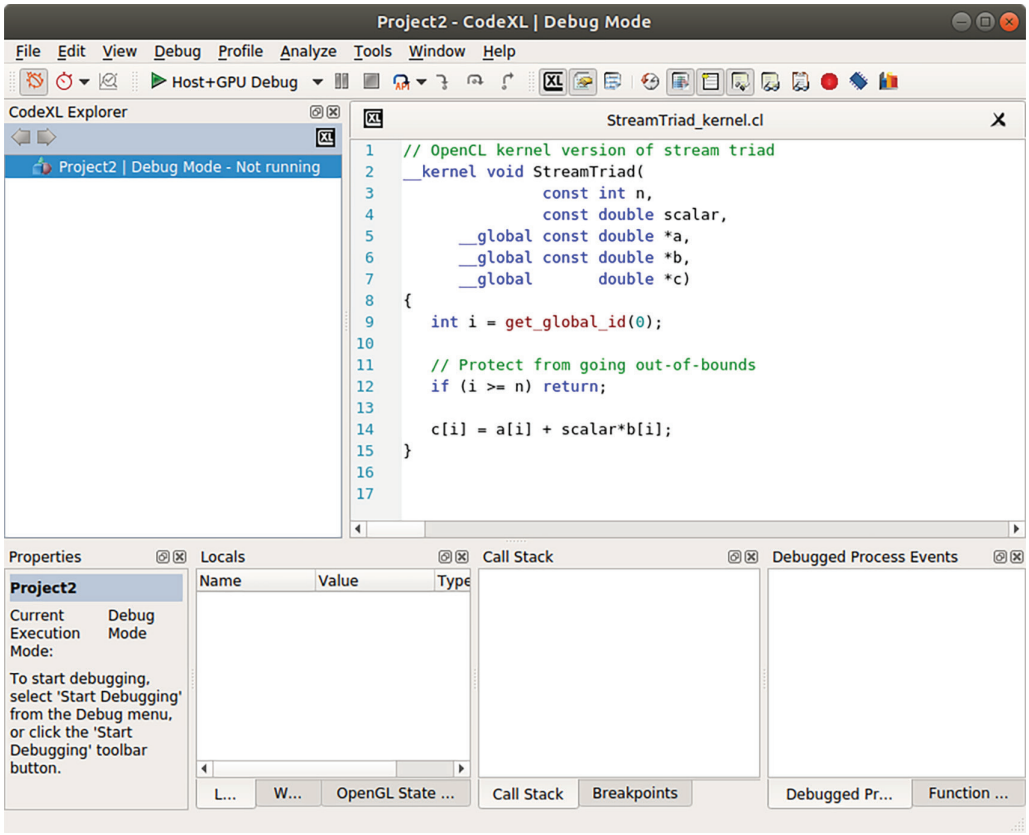


Рис. 13.12 Инструмент разработки CodeXL поддерживает компилирование, выполнение, отлаживание и профилирование

13.5 Не утоните в болоте: сосредотачивайтесь на важных метриках

Как и во многих инструментах профилирования и измерения производительности, первоначально объем информации ошеломляет. Вам следует сосредотачиваться на наиболее важных метриках, которые можете получать с помощью аппаратных счетчиков и других инструментов измерения. В новейших процессорах число аппаратных счетчиков неуклонно растет, что дает вам более глубокое представление о многих аспектах производительности процессора, которые ранее были скрыты. Мы предлагаем следующие три аспекта в качестве наиболее важных: занятость, эффективность выдачи и пропускную способность памяти.

13.5.1 Занятость: достаточно ли работы?

Концепция занятости часто упоминается как приоритетная для GPU-процессоров. Мы впервые обсудили эту меру в разделе 10.3. Для хорошей производительности GPU необходимо иметь достаточный объем работы, чтобы вычислительные модули (CU) были заняты. Кроме того, нам нужна альтернативная работа для покрытия пробуксовок, когда рабочие группы попадают в ситуации ожидания загрузки памяти (рис. 13.13). Напомним, что CU-модули в терминологии OpenCL в CUDA называются потоковыми мультипроцессорами (streaming multiprocessor, SM). О фактической достигнутой занятости сообщают счетчики измерений. Если вы сталкиваетесь с низкими показателями занятости, то можете внести изменения в размер рабочей группы и потребление ресурсов в вычислительных ядрах, чтобы попытаться улучшить этот фактор. Более высокая занятость не всегда лучше. Занятость просто должна быть достаточно высокой, чтобы у вычислительных модулей была альтернативная работа.

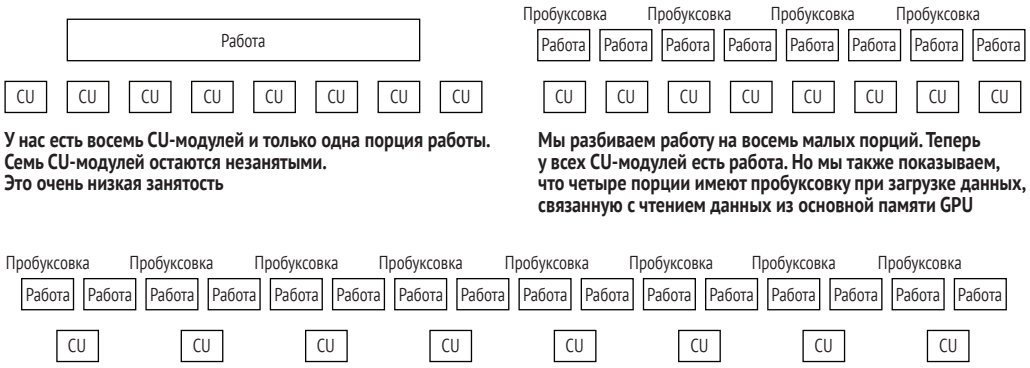


Рис. 13.13 GPU-процессоры имеют большое число вычислительных модулей (CU), также именуемых потоковыми мультипроцессорами (SM). В целях поддержания CU-модулей в занятом состоянии нам необходимо создавать много работы с достаточным объемом дополнительной работы для улаживания пробуксовок

13.5.2 Эффективность выдачи: ваши варпы прерываются слишком часто?

Эффективность выдачи (Issue efficiency) – это мера для измерения команд, выдаваемых за цикл, по сравнению с максимумом команд за цикл. Для того чтобы иметь возможность выдавать команды, каждый планировщик CU-модуля должен иметь приемлемый готовый к исполнению волновой фронт, или варп. *Приемлемый волновой фронт* – это активный волновой фронт, который не пробуксовывает. В некотором смысле это важный результат наличия достаточно высокой занятости, вследствие

которой имеется много активных волновых фронтов. Командами могут быть операции с плавающей точкой, целые числа или операции с памятью. Плохо написанные вычислительные ядра с большим числом пробуксовок приводят к низкой эффективности, даже если занятость высока. Существует целый ряд причин, по которым вычислительные ядра сталкиваются с пробуксовками. Кроме того, существуют счетчики, которые могут выявлять конкретные причины пробуксовок. Несколько возможных причин таковы

- *зависимость от памяти* – ожидание загрузки или сохранения памяти;
- *зависимость от исполнения* – ожидание завершения предыдущей команды;
- *синхронизация* – заблокированный варп из-за вызова синхронизации;
- *зависание памяти* – большое число нереализованных операций с памятью;
- *конфликтное неуспешное обращение к кешу* – неуспешное обращение к кешу, когда строка кеша еще требуется другим блоком памяти;
- *текстура занята работой* – полностью задействованное текстурное оборудование;
- *конвейер занят работой* – вычислительные ресурсы недоступны.

13.5.3 Достигнутая пропускная способность: она всегда сводится к пропускной способности

Пропускная способность является важной для понимания метрикой, поскольку большинство приложений лимитировано по пропускной способности (или ширине полосы пропускания). Самая лучшая отправная точка – это всегда смотреть на меру пропускной способности. Существует целый ряд счетчиков памяти, позволяющих вам заходить настолько глубоко, насколько вы захотите. Сравнение полученных вами результатов измерений пропускной способности с теоретической и измеренной производительностью пропускной способности вашей архитектуры из разделов 9.3.1–9.3.3 может давать вам оценку качества работы вашего приложения. Вы можете использовать результаты измерения памяти для определения того, будет ли полезно коагулировать загрузки память, сохранять значения в локальной памяти (блокноте) или же реструктурировать исходный код для многоразового использования значений данных.

13.6 Контейнеры и виртуальные машины обеспечивают обходные пути

Вы находитесь в полете откуда-то в никуда и просто хотите, чтобы часть вашего исходного кода GPU работала. Последняя версия программно-

информационного обеспечения не работает на ноутбуке, который был выпущен вашей компанией. Обходной путь заключается в использовании контейнера или виртуальной машины (VM) для выполнения другой операционной системы или другой версии компилятора.

13.6.1 Контейнеры Docker в качестве обходного пути

В каждой нашей главе есть пример файла Dockerfile и инструкции по его использованию. Файл Dockerfile содержит команды для сборки базовой операционной системы, а затем для инсталлирования необходимого программно-информационного обеспечения, в котором она нуждается.

Пример: сборка образа Docker с помощью предоставленного файла Dockerfile

Верхнеуровневый каталог в большинстве глав содержит файл Dockerfile. В каталоге для интересующей вас главы выполните команду сборки `build` для Docker и создайте контейнер Docker, используя опцию `-t`, чтобы назначить ему имя. В этом примере мы соберем контейнер Docker главы 2 и назовем его chapter2:

```
docker build -t essentialsofparallelcomputing/chapter2
```

Теперь выполните контейнер Docker следующей ниже командой:

```
docker run -it --entrypoint /bin/bash
             essentialsofparallelcomputing/chapter2
```

В качестве альтернативы используйте:

```
./docker_run.sh
```

Некоторые главы содержат как текстовый, так и графический файл Dockerfile. Для того чтобы задействовать текстовый файл, удалите Dockerfile и привяжите текстовую версию с помощью вот этой команды:

```
ln -s Dockerfile.Ubuntu20.04 Dockerfile
```

Контейнер Docker полезен для работы с программно-информационным обеспечением, которое не работает в вашей операционной системе. Например, для программно-информационного обеспечения, работающего только в Linux, вы можете инсталлировать контейнер на свой ноутбук Mac либо Windows под управлением Ubuntu 20.0.4. Использование контейнера хорошо подходит для текстового консольного программно-информационного обеспечения, работающего в командной строке.

Контейнеры также лимитируют доступ к аппаратным устройствам, таким как GPU-процессоры. Одним из вариантов является выполнение вычислительных ядер устройств на CPU для языков GPU, которые обла-

дают такой способностью. Благодаря этому мы можем, по крайней мере, протестировать наше программно-информационное обеспечение. Если этого для наших нужд недостаточно, то можем предпринять несколько дополнительных шагов, чтобы попытаться наладить работу графики и вычислений на GPU. Мы начнем с того, что обратимся к приведению графики в рабочее состояние. Выполнение графического интерфейса из сборки Docker требует немного больше усилий.

Пример: выполнение образа Docker с графическим интерфейсом на macOS

В ноутбуках Mac клиент X Window не встроен в их стандартное программно-информационное обеспечение. Поэтому вам необходимо инсталлировать клиента X Window на свой Mac, если вы еще этого не сделали. Пакет XQuartz представляет собой версию с открытым исходным кодом оригинального клиента X Window, который включался в более старые версии macOS. Вы можете инсталлировать его с помощью менеджера пакетов brew следующим образом:

```
brew cask install xQuartz
```

Теперь запустите XQuartz и найдите строку меню XQuartz в верхней части экрана. Если вы ее не видите, то вам, возможно, также потребуется запустить образец приложения X Window, такого как xterm, щелкнув правой кнопкой мыши по значку XQuartz. Затем:

1. выберите строку меню XQuartz, а потом опцию Preferences (Настройки);
2. перейдите на вкладку Security (Безопасность) и выберите Allow Connections from Network Clients (Разрешить подключения от сетевых клиентов);
3. перезагрузите свою систему, чтобы применить сделанные настройки;
4. снова запустите XQuartz.

В главах, в которых требуется графический интерфейс для инструментов или графиков, инструкции немного отличаются. Мы используем программно-информационное обеспечение вычислений в виртуальных сетях Virtual Network Computing (VNC) для обеспечения графических возможностей через веб-интерфейс и клиентские средства просмотра VNC. Вы должны применить скрипт dock-er_run.sh для запуска сервера VNC, затем вам нужно запустить клиента VNC в вашей локальной системе. Вы можете использовать один из множества клиентских пакетов VNC либо открыть графический файл в некоторых браузерах, указав в поле имени веб-сайта на панели инструментов браузера следующее:

```
http://localhost:6080/vnc.html?resize=downscale&autoconnect=1&password=<пароль>"
```

В целях тестирования приложения с графическим интерфейсом, таким как NVVP, наберите nvvp. Как вариант, вы можете протестировать гра-

фику с помощью простого приложения X Window, такого как xclock или xterm. Мы также можем попытаться получить доступ к GPU-процессорам для вычислений. Доступ к GPU-процессорам можно получить с помощью опции `--gpus` или же более старой опции `--device=/dev/<имя устройства>`. Эта опция является относительно новым дополнением в Docker и в настоящее время применяется только для GPU-процессоров NVIDIA.

Пример: доступ к GPU-процессорам для вычислительной работы

В целях получения доступа к GPU для вычислений, добавьте опцию `--gpu` с целочисленным аргументом для числа GPU-процессоров (gpus), которые будут доступны, или `all` для всех GPU-процессоров:

```
docker run -it --gpus all --entrypoint /bin/bash chapter13
```

Для GPU-процессоров Intel вы можете попробовать:

```
docker run -it --device=/dev/dri --entrypoint /bin/bash chapter13
```

В большинстве глав есть предварительно собранные контейнеры Docker. Обратиться к контейнерам каждой главы можно по адресу https://hub.docker.com/u/essentialsofparallelcomputing. Вы можете извлечь контейнер той или иной главы с помощью следующей ниже команды:

```
docker run -p 4000:80 -it --entrypoint /bin/bash
     essentialsofparallelcomputing/chapter2
```

Существует также предварительно собранный контейнер Docker от NVIDIA, который можно использовать в качестве отправной точки для своих собственных образов Docker. Посетите их веб-сайт по адресу https://github.com/NVIDIA/nvidia-docker для получения последних инструкций. В NVIDIA есть еще один веб-сайт с основательными разновидностями контейнеров по адресу https://ngc.nvidia.com/catalog/containers. Для ROCm существуют подробные инструкции по контейнерам Docker по адресу https://github.com/RadeonOpenCompute/ROCm-docker. И у Intel есть веб-сайт, посвященный настройке их программно-информационного обеспечения oneAPI в контейнерах по адресу https://github.com/intel/oneapi-containers. Некоторые из их базовых контейнеров имеют крупный размер и требуют хорошего интернет-соединения.

Компилятор PGI важен для разработки исходного кода OpenACC, а также для некоторых других задач разработки исходного кода GPU. Если вам для работы нужен компилятор PGI, то контейнерный веб-сайт для компиляторов PGI находится по адресу https://ngc.nvidia.com/catalog/containers/hpc:pgi-compilers. Как видно из упомянутых здесь веб-сайтов, предлагается целый ряд ресурсов для создания рабочих сред с использованием контейнеров Docker. Но эта способность тоже претерпевает быструю эволюцию.

13.6.2 Виртуальные машины с использованием VirtualBox

Использование виртуальной машины (VM) позволяет пользователю создавать гостевую OS на своем собственном компьютере. Обычная операционная система называется хозяином, или хостом, а виртуальная машина называется гостем. В качестве гостя у вас может работать несколько виртуальных машин. Они используют более ограничительную среду для гостевой операционной системы, чем та, которая существует в имплементациях контейнеров. Часто настраивать графические интерфейсы проще, чем контейнеры. К сожалению, доступ к GPU для вычислений затруднен или невозможен. Вы можете найти виртуальные машины полезными для языков GPU, в которых есть опция, поддерживающая вычисления на хостовом CPU.

Давайте рассмотрим процесс настройки гостевой операционной системы Ubuntu в VirtualBox. В приведенном ниже примере настраивается образец приложения симуляции мелководья, работающий на CPU с компилятором PGI в VirtualBox с графикой.

Пример: настройка гостевой OS Ubuntu в VirtualBox

В целях настройки своей системы под VirtualBox надо:

1. скачать VirtualBox для своей системы и инсталлировать;
2. скачать рабочий стол Ubuntu и сохранить его на локальном диске

   `[ubuntu-20.04-desktop-amd64.iso]`

3. скачать файл VBoxGuestAdditions.iso, который уже, возможно, включен в скачиваемый комплект VirtualBox

Затем мы настраиваем гостевую систему Ubuntu. Автоматизированный скрипт, autovirtualbox.sh, включен в примеры этой главы для автоматизации настройки гостевой системы Ubuntu в VirtualBox по адресу https://github.com/EssentialsOfParallelComputing/Chapter13.git. Большинство других глав имеет похожие скрипты. В целях настройки гостевой системы Ubuntu нужно выполнить следующую ниже последовательность действий:

1. запустить VirtualBox и нажать кнопку **Новая**;
2. набрать имя (например, chapter13);
3. выбрать Linux, а затем 64-битную версию Ubuntu;
4. выбрать объем памяти (например, 8192);
5. создать виртуальный жесткий диск;
6. выбрать дисковый образ VDI VirtualBox Disk Image;
7. выбрать диск фиксированного размера Fixed Size Disk;
8. выбрать 50 Гб.

Теперь ваша новая виртуальная машина должна быть добавлена в список.

Теперь мы готовы инсталлировать Ubuntu. Этот процесс аналогичен настройке системы Ubuntu на вашем настольном компьютере.

Пример: инсталлирование Ubuntu

В целях инсталлирования Ubuntu надо выполнить следующие ниже шаги:

1. запустить виртуальную машину Ubuntu, щелкнув зеленую стрелку **Start** (Пуск);
2. выбрать сохраненный ранее файл iso, набрав ubuntu-20.04-desktop-amd64.iso из представленных вариантов;
3. выбрать **Install Ubuntu** (Инсталлировать Ubuntu);
4. выбрать свою клавиатуру и нажать **Continue** (Продолжить);
5. выбрать **Minimal Install** (Минимальная инсталляция), скачать обновления и инсталлировать стороннее программно-информационное обеспечение, затем нажать **Continue** (Продолжить);
6. выбрать **Erase Disk** (Стереть диск) и инсталлировать Ubuntu, а затем нажать **Install** (Инсталлировать) и выбрать часовой пояс;
7. набрать следующее в текстовые поля: ваше имя (например, chapter13), имя вашего компьютера (chapter13-virtualbox), пользовательское имя (chapter13) и пароль (chapter13);
8. выбрать **Require My Password to Log In** (Требовать мой пароль для входа в систему), а затем выбрать **Continue** (Продолжить).

Самое время попить кофейку. Когда инсталляция будет завершена, надо перезагрузить компьютер и выполнить следующие ниже шаги:

1. снова войти в систему;
2. перейти по ссылке **What's New** (Что нового);
3. выбрать точки в левом нижнем углу и запустить терминал;
4. отредактировать конфигурационный файл Sudo Authorized Users (Файл авторизованных пользователей Sudo) с помощью

```
sudo -i
visudo
```

   и добавить следующее в любую пустую строку: `%vboxsf ALL=(ALL) ALL`, – затем выйти.

Возможно, вам придется дождаться обновлений или перезагрузиться и снова войти в систему. После входа в систему надо инсталлировать базовые сборочные инструменты с помощью `sudo apt install build-essential dkms git -y`. Затем необходимо:

1. сделать окно VirtualBox активным и выбрать раскрывающееся меню **Devices** (Устройства) в меню окна в верхней части экрана;
2. установить опцию **Shared Clipboard** (Совместный буфер обмена) равной **Bidirectional** (Двунаправленный);
3. установить опцию **Drag and Drop** (Перетаскивание) равной **Bidirectional** (Двунаправленный);
4. инсталлировать гостевые дополнения, выбрав опцию меню virtualbox-guest-additions-iso;
5. извлечь оптический диск: на рабочем столе щелкнуть правой кнопкой мыши и извлечь устройство либо в окне VirtualBox выбрать **Devices** >

Optical Disk (Устройства > Оптический диск) и извлечь диск из виртуального диска;

6 перезагрузиться и протестировать, скопировав и вставив (копирование в Mac представлено сочетанием **Command-C**, а вставка в Ubuntu – сочетанием **Shift-Ctrl-v**).

Ваша гостевая система Ubuntu теперь готова для скачивания и инсталлирования программно-информационного обеспечения.

Для каждой главы есть инструкции по настройке виртуальных машин с примерами из этих глав. Для данной главы войдите в систему и инсталлируйте примеры этой главы:

```
git clone --recursive https://github.com/essentialsofparallelcomputing/
    Chapter13.git
cd Chapter13 && sh -v README.virtualbox
```

Команды в файле README.virtualbox инсталлируют программно-информационное обеспечение, собирают и выполняют приложение симуляции мелководья. Вывод реально-временной графики тоже должен работать. Вы также можете попробовать утилиту nvprof для профилирования приложения симуляции мелководья.

13.7 Облачные опции: гибкие и переносимые возможности

Когда доступ к конкретному GPU лимитирован (нет суперкомпьютерного, ноутбучного или настольного GPU или дистанцированного сервера), вы можете использовать облачные вычисления[1]. Облачные вычисления относятся к серверам, предоставляемым крупными центрами обработки данных. В то время как большинство этих служб предназначено для более широких пользователей, начинают появляться веб-сайты, ориентированные на службы в стиле HPC. Одним из таких веб-сайтов является http://mng.bz/Q2YG. Облачный кластер Fluid Numerics (fluid-slurm-gcp), настроенный на облачной платформе Google Cloud (GCP), имеет планировщик пакетов Slurm и MPI. GPU-процессоры NVIDIA также могут быть запланированы. Начало работы, возможно, будет немного затрудненным. На веб-сайте Fluid Numerics по адресу http://mng.bz/XYwv есть немного информации, которая поможет в этом процессе.

Преимущества аппаратных ресурсов, доступных по требованию, нередко бывают весомыми. Google Cloud предлагает пробный кредит в размере 300 долл., которого должно быть более чем достаточно для

[1] Обратитесь к файлу README.cloud в примерах этой главы для получения последней информации об использовании облака.

разведывания этой службы. Существуют другие облачные провайдеры и дополнительные службы, которые могут предоставлять именно то, что вам нужно, либо вы можете настраивать свою среду самостоятельно. Компания Intel организовала облачную службу для тестирования GPU-процессоров Intel, чтобы давать разработчикам возможность доступа к программно-информационному и к аппаратному обеспечению для своей инициативы oneAPI и своего компилятора DPCPP, обеспечивающего имплементацию SYCL. Вы можете попробовать ее, перейдя по адресу https://software.intel.com/en-us/oneapi и зарегистрировавшись для ее использования.

13.8 Материалы для дальнейшего изучения

Встраивание рабочего потока и среды разработки имеет особую важность для разработки исходного кода GPU. Учитывая большое разнообразие возможных конфигураций оборудования, представленные в этой главе примеры, вероятно, потребуют некоторой настройки под вашу ситуацию. И действительно, конфигурация и настройка систем разработки является одной из трудных проблем, связанных с вычислениями на GPU. Возможно, вы даже обнаружите, что проще использовать один из предварительно собранных контейнеров Docker, чем разбираться в процессе конфигурирования и инсталлирования программно-информационного обеспечения в вашей системе.

Мы также рекомендуем ознакомиться с самой последней документацией, соответствующей вашим потребностям, из раздела дополнительного чтения, предложенного в разделе 13.8.1. Инструменты и рабочие потоки являются наиболее быстро меняющимися аспектами программирования GPU. Хотя примеры в этой главе в целом будут релевантными, детали, скорее всего, изменятся. Большая часть программно-информационного обеспечения настолько нова, что документация по его использованию все еще разрабатывается.

13.8.1 Дополнительное чтение

В руководстве по инсталляции NVIDIA содержится немного информации об инсталлировании инструментов CUDA с помощью менеджера пакетов по адресу:

- https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html#package-manager-installation.

У NVIDIA есть несколько ресурсов по их инструментам профилирования и переходу с NVVP на комплект инструментов Nsight на следующих ниже веб-сайтах.

- Руководство NVIDIA NSight по адресу https://docs.nvidia.com/nsight-compute/NsightCompute/index.html#nvvp-guide.
- Сравнение инструментов профилирования NVIDIA по адресу https://devblogs.nvidia.com/migrating-nvidia-nsight-tools-nvvp-nvprof/.

Другие инструменты включают следующее:

- CodeXL был выпущен с открытым исходным кодом в рамках инициативы GPUOpen. AMD также удалила свой бренд AMD из инструмента для продвижения кросс-платформенной разработки. Для получения дополнительной информации обратитесь по адресу https://github.com/GPUOpen-Tools/CodeXL;
- у NVIDIA есть облако GPU с такими ресурсами, как компиляторы PGI, находящееся по адресу https://ngc.nvidia.com/catalog/containers/hpc:pgi-compilers;
- у AMD также есть веб-страница по настройке сред виртуализации и контейнеров. Инструкции по виртуализации включают в себя транзитную технику для получения доступа к GPU для вычислений. Вы найдете эту информацию по адресу http://mng.bz/MgWW.

13.8.2 Упражнения

1. Выполните инструмент nvprof в примере потоковой триады. Вы можете попробовать версию CUDA из главы 12 либо версию OpenACC из главы 11. Какой рабочий поток вы использовали для своих аппаратных ресурсов? Если у вас нет доступа к GPU NVIDIA, то сможете ли вы использовать другой инструмент профилирования?
2. Создайте трассу из nvprof и импортируйте ее в NVVP. Где тратится время выполнения? Что вы могли бы сделать, чтобы его оптимизировать?
3. Загрузите предварительно собранный контейнер Docker от соответствующего поставщика для вашей системы. Запустите контейнер и выполните один из примеров из главы 11 либо 12.

Резюме

- Повышение производительности является приоритетной задачей для научных приложений и приложений с большими данными. Инструменты повышения производительности помогут вам получать максимальную отдачу от вашего оборудования GPU.
- Для программирования GPU доступен целый ряд инструментов профилирования. Вы должны опробовать многие новые и появляющиеся доступные способности.
- Рабочие потоки необходимы для эффективной разработки исходного кода GPU. Следует разведать, что конкретно работает для вас в вашей среде и доступном оборудовании GPU.
- Существуют обходные пути с использованием контейнеров, виртуальных машин и облачных вычислений для улаживания несовместимостей, потребностей в вычислениях и доступа к оборудованию GPU. Эти обходные пути предоставляют доступ к большой выборке поставляемого оборудования GPU, к которому в противном случае невозможно было бы получить доступ.

Часть IV

Экосистемы высокопроизводительных вычислений

Для работы с современными системами высокопроизводительных вычислений (HPC) вам недостаточно просто изучить языки параллельного программирования. Также необходимо понимать многие аспекты экосистемы, включая следующие:

- размещение и планирование ваших процессов для повышения производительности;
- запрашивание и планирование ресурсов с использованием пакетной системы HPC;
- запись и чтение данных в параллельном режиме на параллельных файловых системах;
- полномерное использование инструментов и ресурсов с целью анализа производительности и содействия разработке программно-информационного обеспечения.

Это лишь несколько важных тем, которые окружают стержневые языки параллельного программирования со всех сторон, формируя дополнительный набор возможностей, которые мы называем экосистемой HPC.

Наши вычислительные системы экспоненциально растут как по сложности, так и по числу процессорных ядер. Многие соображения, заложенные в HPC, становятся важными и для высокопроизводительных рабочих

станций. При столь большом числе процессорных ядер нам необходимо контролировать размещение и планирование процессов внутри узла, что в широком смысле называется *процессной аффинностью* и выполняется совместно с вычислительным ядром OS. По мере роста числа процессорных ядер быстро разрабатываются инструменты контроля за аффинностью процессов, помогающие решать новые трудности, связанные с размещением процессов. В главе 14 мы рассмотрим несколько техник, доступных для назначения процессной аффинности.

Сложные системы управления ресурсами стали повсеместными из-за роста сложности вычислительных ресурсов. Эти «пакетные системы» формируют очередь запросов на ресурсы и распределяют их в соответствии с системой приоритетов, именуемой алгоритмом объективного распределения. Когда вы впервые подключаетесь к системе HPC, пакетная система, возможно, будет сбивать с толку. Не зная, как использовать планировщик, вы не сможете развертывать свои приложения на этих крупных машинах. Вот почему мы считаем необходимым в главе 15 рассмотреть основы использования наиболее распространенных пакетных систем.

Кроме того, мы не просто записываем файлы таким же образом в системах HPC; мы записываем их в параллельном режиме на специальное оборудование файловой системы, которое может подразделять на полосы запись в файлы по многочисленным дискам одновременно. В целях использования возможностей этих параллельных файловых систем вам необходимо ознакомиться с некоторыми программами, используемыми для параллельных файловых операций. В главе 16 мы покажем вам, как использовать MPI-IO и HDF5, пару наиболее распространенных программных библиотек обработки параллельных файлов. По мере того как наборы данных становятся все больше, потенциальные применения программно-информационного обеспечения для параллельных файлов расширяются, выходя далеко за пределы традиционных приложений HPC.

Глава 17 охватывает широкий спектр важных инструментов и ресурсов для разработчика приложений HPC. Вы, возможно, обнаружите, что профилировщики имеют большое значение в повышении производительности вашего приложения. Существует широкий спектр профилировщиков для разных вариантов использования и оборудования, такого как GPU-процессоры. Кроме того, есть инструменты, которые помогают в процессе разработки программно-информационного обеспечения. Указанные инструменты позволяют создавать правильные, устойчивые приложения. Вдобавок многие разработчики приложений могут найти специализированные подходы для своего приложения из широкого спектра примеров приложений.

Возможности экосистемы HPC становятся все более важными по мере роста сложности и масштабов наших вычислительных платформ. Знаниями о том, как использовать эти возможности, нередко пренебрегали. Мы надеемся, что, рассмотрев эти часто упускаемые из виду аспекты высокопроизводительных вычислений в последующих четырех главах, вы будете способны использовать свое компьютерное оборудование продуктивнее.

14 Аффинность: перемирие с вычислительным ядром

Эта глава охватывает следующие ниже темы:

- почему аффинность является важной проблемой для современных CPU;
- контроль за аффинностью в ваших параллельных приложениях;
- точную регулировку производительности с помощью размещения процессов.

Мы впервые столкнулись с аффинностью в разделе 8.6.2 об MPI (Интерфейсе передачи сообщений), где мы дали ей определение и кратко показали, как с ней обращаться. Ниже мы повторяем наше определение, а также даем определение понятию размещения процесса.

- *Аффинность* – отдает предпочтение тому или иному аппаратному компоненту в планировании процесса, ранга или потока (виртуального ядра). Она также называется *закреплением* или *привязкой*.
- *Размещение* – назначает процесс или поток местоположению оборудования.

В этой главе мы подробнее рассмотрим аффинность (син. – близость, родство, сходство), размещение и порядок потоков или рангов. Озабоченность в отношении аффинности – это недавно возникшее явление.

В прошлом, имея всего несколько процессорных ядер на CPU, можно было получать не столь уж много. По мере роста числа процессоров и усложнения архитектуры вычислительного узла аффинность приобретает все более и более важное значение. Тем не менее выгоды относительно скромны; возможно, самая большая выгода заключается в сокращении вариации в производительности от выполнения к выполнению и улучшении наузлового масштабирования. Время от времени контроль за аффинностью помогает избегать действительно катастрофических планировочных решений со стороны вычислительного ядра в отношении характеристик вашего приложения.

Решение о месте, куда размещать процесс или поток, принимается вычислительным ядром операционной системы. Планирование со стороны вычислительного ядра имеет богатую историю и является ключом к разработке многозадачных многопользовательских операционных систем. Именно благодаря этим способностям вы можете запускать электронную таблицу, временно переключаться на текстовый процессор, а затем работать с важным электронным письмом. Однако алгоритмы планирования, разработанные для обычного пользователя, не всегда подходят для параллельных вычислений. Мы можем запускать четыре процесса для системы с четырьмя процессорными ядрами, но операционная система планирует эти четыре процесса так, как ей заблагорассудится. Она может разместить все четыре процесса на одном процессоре либо распределить их по четырем процессорам. Как правило, вычислительное ядро делает что-то разумное, но оно может прерывать один из параллельных процессов для выполнения системной функции, в результате чего все остальные процессы будут простаивать и ждать.

В главе 1 на рис. 1.20 и 1.21 мы показали вопросительные знаки в местах, где размещаются процессы, поскольку мы не контролируем размещение процессоров или потоков на процессорах. По крайней мере, не делали это до сих пор. Последние выпуски MPI, OpenMP и планировщиков пакетов начали предлагать функциональности по контролю за размещением и аффинностью. Хотя в некоторых интерфейсах произошло много изменений в опциях, с последними выпусками, похоже, все успокоилось. Тем не менее рекомендуем вам проверить документацию выпусков, которые вы используете, на наличие любых различий.

14.1 Почему важна аффинность?

В отличие от большинства распространенных настольных приложений, параллельные процессы необходимо планировать вместе. Такое планирование называется бригадным планированием.

ОПРЕДЕЛЕНИЕ *Бригадное планирование* (gang scheduling) – это алгоритм планирования вычислительного ядра, который активирует группу процессов одновременно.

Поскольку параллельные процессы, как правило, периодически синхронизируются во время выполнения, планирование исполнения одного потока, который в конечном итоге ожидает другого неактивного процесса, не дает никаких выгод. Алгоритм планирования вычислительного ядра не содержит информации о том, что процесс зависит от работы другого. Это справедливо также для потоков MPI, OpenMP и вычислительных ядер GPU. Наилучший подход к бригадному планированию состоит в выделении только такого числа процессов, сколько имеется процессоров, и привязке этих процессов к процессорам. Не следует забывать, что ядерным и системным процессам нужно где-то работать. Некоторые передовые технические приемы резервируют процессор только для системных процессов.

Недостаточно поддерживать каждый параллельный процесс активным и запланированным. Нам также необходимо, чтобы процессы планировались в одном и том же домене неравномерного доступа к памяти (Non-Uniform Memory Access, NUMA), чтобы минимизировать затраты на доступ к памяти. В OpenMP мы обычно наталкиваемся на большое число проблем, связанных с «первым касанием» массивов данных на процессоре, где используются данные (см. раздел 7.1.1). Если вычислительное ядро затем перемещает ваш процесс в другой домен NUMA, то все ваши усилия оказываются напрасными. В разделе 7.3.1 мы убедились, что штраф за доступ к памяти в неправильном домене NUMA обычно бывает двукратным или более того. Первостепенная задача в отношении наших процессов состоит в том, чтобы оставаться в одном и том же домене памяти.

В типичной ситуации домен NUMA выравнивается с разъемами (сокетами) на узле. Если мы сможем указать процессу планировать аффинность на одном и том же разъеме, то мы всегда будем получать одинаковое оптимальное время доступа к основной памяти. Однако необходимость в аффинности участка NUMA зависит от архитектуры вашего CPU. Персональные вычислительные системы часто имеют только один участок NUMA, тогда как крупные системы HPC часто имеют гораздо больше процессорных ядер в расчете на узел с двумя разъемами CPU и двумя или более участками NUMA.

Хотя прикрепление аффинности к домену NUMA оптимизирует время доступа к основной памяти, мы все равно можем иметь менее оптимальную производительность из-за слабого использования кеша. Процесс заполняет кеш L1 и L2 необходимой ему памятью. Но затем, если он будет изъят с заменой на другой процессор на том же домене NUMA с другим кешем L1 и L2, то производительность кеша пострадает. Затем кеши необходимо снова заполнить. При частом повторном использовании данных это приводит к потере производительности. С MPI мы хотим замыкать процессы или ранги на процессор. Но с OpenMP это приводит к тому, что все потоки запускаются на одном процессоре, потому что аффинность наследуется порожденными потоками. С OpenMP мы хотим, чтобы каждый поток имел аффинность со своим процессором.

Некоторые процессоры также имеют новую функциональность, именуемую гиперпотоками. Гиперпотоки добавляют еще один слой сложно-

сти в соображения о местах размещения процессов. Сначала нам нужно дать определение гиперпотокообразованию и понять, что это такое.

ОПРЕДЕЛЕНИЕ *Гиперпотокообразование* (или *гиперпоточность* – hyperthreading), технология Intel, которая делает один процессор похожим на два виртуальных процессора за счет совместного использования аппаратных ресурсов между двумя потоками в операционной системе.

Гиперпотоки совместно используют одно физическое ядро и его систему кеширования. Поскольку кеш является совместным, перемещение между гиперпотоками не столь сильно штрафуется. Но это также означает, что каждое виртуальное ядро имеет половину кеша в качестве реального физического ядра, если процессы не имеют совместных данных. Для наших приложений, связанных с памятью, сокращение объема кеша вдвое может стать серьезным ударом. Отсюда эффективность этих виртуальных ядер неоднозначна. Многие системы HPC их отключают, потому что некоторые программы замедляются за счет гиперпотоков. Не все гиперпотоки одинаковы как на уровне оборудования, так и на уровне операционной системы, поэтому не думайте, что если вы не видели выгод в предыдущей имплементации, то не увидите их и в вашей текущей системе. Если мы используем гиперпотокообразование, то мы захотим, чтобы место размещения процесса находилось рядом с тем, чтобы совместный кеш приносил пользу обоим виртуальным процессорам.

14.2 Нащупывание вашей архитектуры

В целях эффективного задействования аффинности для повышения производительности нам необходимо знать детали нашей аппаратной архитектуры. Эта задача затрудняется разнообразием аппаратных архитектур; только у Intel более тысячи моделей CPU. В данном разделе мы расскажем о том, как понимать вашу архитектуру. Это понимание является необходимым требованием для того, чтобы иметь возможность использовать аффинность на его основе.

Наилучшее представление о своей архитектуре можно получить с помощью утилиты lstopo. Мы впервые увидели lstopo в разделе 3.2.1 на рис. 3.2 с распечаткой для ноутбука Mac. Указанный ноутбук представляет собой простую архитектуру с четырьмя физическими обрабатывающими ядрами, которые при активированном гиперпотообразовании выглядят как восемь виртуальных ядер для операционных систем. На рис. 3.2 мы также видим, что кеши L1 и L2 являются приватными для физического ядра, а L3 – совместным для всех процессоров. Мы также отмечаем, что существует только один домен NUMA. Теперь давайте взглянем на более усложненный процессор. На рис. 14.1 показана архитектура CPU Intel Skylake Gold.

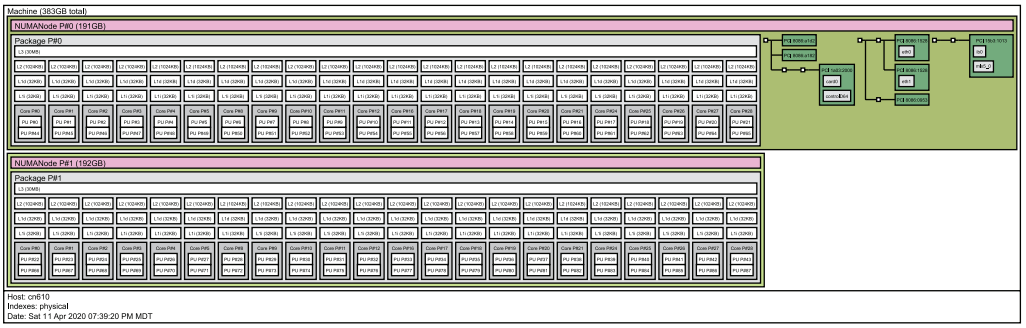


Рис. 14.1 Архитектура Intel Skylake Gold с двумя доменами NUMA и 88 обрабатывающими ядрами раскрывает сложность вычислительных узлов более высокого класса

Серые прямоугольники на рис. 14.1, каждый из которых помечен ядром и содержит два светлых прямоугольника, обозначенных как PU, т. е. обрабатывающий модуль, являются физическими ядрами. В каждом из этих серых прямоугольников есть два прямоугольника внутри, которые являются виртуальными процессорами, создаваемыми гиперпотоками. Кеши L1 и L2 являются приватными для каждого физического процессора, тогда как кеш L3 является совместным для домена NUMA. Мы также видим, что сеть и другие периферийные устройства справа от рисунка расположены ближе к первому домену NUMA. Мы можем получать некоторую информацию о большинстве систем Linux или Unix с помощью команды `lscpu` (рис. 14.2).

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                88
On-line CPU(s) list:   0-87
Thread(s) per core:    2
Core(s) per socket:    22
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 85
Model name:            Intel(R) Xeon(R) Gold 6152 CPU @ 2.10GHz
Stepping:              4
CPU MHz:               1000.012
CPU max MHz:           3700.0000
CPU min MHz:           1000.0000
BogoMIPS:              4200.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              1024K
L3 cache:              30976K
NUMA node0 CPU(s):     0-21,44-65
NUMA node1 CPU(s):     22-43,66-87
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx
fxsr sse sse2 ss ht tm pbe syscall nx pdpelgb rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl
xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16
xtpr pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm
3dnowprefetch epb cat_l3 cdp_l3 invpcid_single intel_ppin intel_pt mba tpr_shadow vnmi flexpriority ept vpid fsgsbase
tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm cqm mpx rdt_a avx512f avx512dq rdseed adx smap clflushopt clwb
avx512cd avx512bw avx512vl xsaveopt xsavec xgetbv1 cqm_llc cqm_occup_llc cqm_mbm_total cqm_mbm_local dtherm ida arat
pln pts pku ospke
```

Рис. 14.2 Результат на выходе из команды `lscpu` для CPU Intel Skylake Gold

Результат на выходе из `lscpu` подтверждает, что на ядро приходится два потока и два домена NUMA. Нумерация процессоров кажется немного странноватой, но, имея первые 22 процессора на первом узле NUMA, а затем пропуская, чтобы включить следующие 22 процессора на втором узле, мы оставляем гиперпотоки пронумерованными в последнюю очередь. Помните, что определение узла в утилитах NUMA отличается от нашего определения, где это отдельная распределенная система памяти.

Итак, какой же будет стратегия аффинности и размещения процессов для этой архитектуры? Так вот, все зависит от приложения. Каждое приложение имеет разные потребности в масштабировании и производительности потокообразования, которые необходимо учитывать. Мы хотим следить за тем, чтобы процессы оставались в их доменах NUMA с целью обеспечения оптимальной для основной памяти пропускной способности.

14.3 Аффинность потоков с OpenMP

Аффинность потоков жизненно важна при оптимизировании приложений с помощью OpenMP. Привязывание потока к местоположению используемой памяти важна для достижения хорошей задержки и пропускной способности памяти. Мы прилагаем большие усилия, чтобы сделать *первое касание*, с целью размещения памяти рядом с потоком, как мы обсуждали в разделе 7.1.1. Если потоки переносятся на разные процессоры, то мы теряем все выгоды, которые должны получить от наших дополнительных усилий.

С OpenMP версии 4.0 элементы контроля за аффинностью в OpenMP были расширены за счет включения ключевых слов `close`, `spread` и `primary` в дополнение к существующим опциям `true` либо `false`. Также были добавлены три опции для средовой переменной `OMP_PLACES`, `sockets`, `cores` и `threads`. Таким образом, теперь у нас есть вот такие элементы контроля за аффинностью и размещением:

- `OMP_PLACES = [sockets|cores|threads]` или явно заданный список мест;
- `OMP_PROC_BIND = [close|spread|primary]` или `[true|false]`.

`OMP_PLACES` устанавливает лимиты на то, где потоки могут быть запланированы. На самом деле есть одна опция, которой нет в списке: node (узел). Это значение используется по умолчанию, и оно позволяет планировать каждый поток где угодно в «местоположении». При наличии более чем одного потока в местоположении используемого по умолчанию узла существует вероятность того, что планировщик переместит потоки либо будет иметь коллизии с двумя или более потоками, запланированными для одного виртуального процессора. Один разумный подход состоит в том, чтобы не иметь больше потоков, чем количество в указанном месте. Возможно, более подходящее правило состоит в том, чтобы указывать место, которое имеет количество, превышающее же-

лаемое число потоков. Мы покажем, как это работает, на примере ниже в данном разделе.

Средовая переменная `OMP_PROC_BIND` имеет пять возможных значений, но по своему смыслу они имеют некоторое наложение. Значения `close`, `spread` и `primary` являются особыми версиями значения `true`.

> **ПРИМЕЧАНИЕ** Мы также отмечаем, что ключевое слово `primary` заменяет устаревшее ключевое слово `master` в стандарте OpenMP v5.1. По мере того как в компиляторах имплементируется новый стандарт, вы, возможно, продолжите сталкиваться со старым использованием.

Имея значение `false`, планировщик вычислительного ядра может перемещать потоки свободно. Значение `true` указывает вычислительному ядру не перемещать поток после того, как он будет запланирован. Но он может быть запланирован где угодно в пределах ограничения местоположения и может варьироваться от выполнения к выполнению. Значение `primary` является особым случаем, который занимается планированием потоков на главном процессоре. Значение `close` планирует закрытие потоков вместе, а значение `spread` распределяет потоки. Выбор того, какой из этих двух значений использовать, имеет некоторые тонкие последствия, которые вы увидите в примере из этого раздела.

> **ПРИМЕЧАНИЕ** Вы также можете задавать размещение с помощью подробного списка. Этот вариант использования является более продвинутым, и мы его здесь обсуждать не будем. Подробный список может обеспечивать более точный контроль, но он менее переносим для другого типа CPU.

Средовые переменные OpenMP задают аффинность и места размещения для всей программы. Вы также можете устанавливать аффинность для отдельных циклов, добавляя выражение в директиву `parallel`. Указанное выражение имеет следующий ниже синтаксис:

```
proc_bind([primary|close|spread])
```

Эти элементы контроля за аффинностью показаны в следующем ниже примере в действии в нашей простой программе векторного сложения из раздела 7.3.1. В ваш исходный код можно также добавить процедуры отчетности об аффинности, чтобы видеть их влияние.

Пример: векторное сложение со всеми возможными настройками средовых переменных `OMP_PLACES` и `OMP_PROC_BIND`

В этом примере мы задаем каждую комбинацию средовых переменных OpenMP аффинности и размещения. Сначала модифицируем векторное сло-

жение из раздела 7.3.1 в части вызова процедуры, которая сообщает о размещении потоков, как показано в следующем ниже листинге.

Модифицированный файл vecadd_opt3.c для изучения аффинности

OpenMP/vecadd_opt3.c

```
 1 #include <stdio.h>
 2 #include <time.h>
 3 #include "timer.h"
 4 #include "omp.h"
 5 #include "place_report_omp.h"
 6
 7 // достаточно крупное, чтобы втиснуть в основную память
 8 #define ARRAY_SIZE 80000000
 9 static double a[ARRAY_SIZE], b[ARRAY_SIZE], c[ARRAY_SIZE];
10
11 void vector_add(double *c, double *a, double *b, int n);
12
13 int main(int argc, char *argv[]){
14 #ifdef VERBOSE
15    place_report_omp();
16 #endif
17    struct timespec tstart;
18    double time_sum = 0.0;
19 #pragma omp parallel
20    {
21 #pragma omp for
22       for (int i=0; i<ARRAY_SIZE; i++) {
23          a[i] = 1.0;
24          b[i] = 2.0;
25       }
26
27 #pragma omp masked
28       cpu_timer_start(&tstart);
29       vector_add(c, a, b, ARRAY_SIZE);
30 #pragma omp masked
31       time_sum += cpu_timer_stop(tstart);
32    } // конец прагмы omp parallel
33
34    printf("Время выполнения составляет %lf мс\n", time_sum);
35 }
36
37 void vector_add(double *c, double *a, double *b, int n)
38 {
39 #pragma omp for
40    for (int i=0; i < n; i++){
41       c[i] = a[i] + b[i];
42    }
43 }
```

Определить, чтобы задействовать отчетность

Вызов отчета о размещении

Основная работа выполняется в подпрограмме `place_report_omp`. Мы используем `ifdef` вокруг вызова, чтобы легко включать и выключать отчетность. Итак, теперь давайте взглянем на процедуру отчетности в следующем ниже листинге.

Отчетность о настройках местоположения в OpenMP

OpenMP/place_report_omp.c

```
41 void place_report_omp(void)
42 {
43    #pragma omp parallel
44    {
45       if (omp_get_thread_num() == 0){
46          printf("Выполняется с %d потоком(потоками)\n",
                   omp_get_num_threads());
47          int bind_policy = omp_get_proc_bind();
48          switch (bind_policy)
49          {
50             case omp_proc_bind_false:
51                printf(" proc_bind равна false\n");
52                break;
53             case omp_proc_bind_true:
54                printf(" proc_bind равна true\n");
55                break;
56             case omp_proc_bind_master:
57                printf(" proc_bind равна master\n");
58                break;
59             case omp_proc_bind_close:
60                printf(" proc_bind равна close\n");
61                break;
62             case omp_proc_bind_spread:
63                printf(" proc_bind равна spread\n");
64          }
65          printf(" proc_num_places равно %d\n",
                   omp_get_num_places());
66       }
67    }
68
69    int socket_global[144];
70    char clbuf_global[144][7 * CPU_SETSIZE];
71
72    #pragma omp parallel
73    {
74       int thread = omp_get_thread_num();
75       cpu_set_t coremask;
76       char clbuf[7 * CPU_SETSIZE];
77       memset(clbuf, 0, sizeof(clbuf));
78       sched_getaffinity(0, sizeof(coremask), &coremask);
79       cpuset_to_cstr(&coremask, clbuf);
80       strcpy(clbuf_global[thread],clbuf);
```

Сообщает число потоков

Запрашивает и выдает отчет о значении OMP_PROC_BIND

Запрашивает и выдает отчет о совокупных ограничениях на размещение потоков

Конвертирует битовую маску во что-то, что можно распечатать

Получает битовую маску аффинности

```
81        socket_global[omp_get_thread_num()] = omp_get_place_num();
82        #pragma omp barrier
83        #pragma omp master
84        for (int i=0; i<omp_get_num_threads(); i++){
85         printf("Привет из потока %d: (ядерная аффинность = %s)"
                 " Разъем OpenMP: %d\n",
86                 i, clbuf_global[i], socket_global[i]);
87        }
88    }
89 }
```

Возвращает фактический номер местоположения для вывода на печать

Битовая маска аффинности CPU должна быть конвертирована в более понятный формат для распечатки. Эта процедура показана в следующем ниже листинге.

Процедура конвертирования битовой маски CPU в строковый литерал C

OpenMP/place_report_omp.c

```
12 static char *cpuset_to_cstr(cpu_set_t *mask, char *str)
13 {
14    char *ptr = str;
15    int i, j, entry_made = 0;
16    for (i = 0; i < CPU_SETSIZE; i++) {
17       if (CPU_ISSET(i, mask)) {
18          int run = 0;
19          entry_made = 1;
20          for (j = i + 1; j < CPU_SETSIZE; j++) {
21             if (CPU_ISSET(j, mask)) run++;
22             else break;
23          }
24          if (!run)
25             sprintf(ptr, "%d,", i);
26          else if (run == 1) {
27             sprintf(ptr, "%d,%d,", i, i + 1);
28             i++;
29          } else {
30             sprintf(ptr, "%d-%d,", i, i + run);
31             i += run;
32          }
33          while (*ptr != 0) ptr++;
34       }
35    }
36    ptr -= entry_made;
37    *ptr = 0;
38    return(str);
39 }
```

В подпрограмме отчетности о размещении мы запрашиваем настройки OpenMP, сообщаем о них, а затем показываем размещение и аффин-

ность по каждому потоку. Для того чтобы ее опробовать, надо скомпилировать исходный код с опцией детализации и выполнить его с 44 потоками либо любым числом потоков, которое имеет смысл в вашей системе, без специальных настроек средовых переменных. Образец исходного кода находится по адресу https://github.com/EssentialsofParallelComputing/Chapter14.git в подкаталоге OpenMP.

Пример: запрашивание настроек OpenMP у процедуры отчетности о размещении

Для того чтобы запросить настройки OpenMP, сообщить о них, а затем показать размещение и аффинность по каждому потоку, выполните следующие ниже шаги.

```
mkdir build && cd build
cmake -DCMAKE_VERBOSE=on ..
make
export OMP_NUM_THREADS=44
./vecadd_opt3
```

Выполнив эту последовательность команд на Intel Skylake-Gold с помощью GCC 9.3, вы получите показанную ниже распечатку.

```
Выполняется с 44 потоком(потоками)
  proc_bind равна false
  proc_num_places равно 0
Привет из потока 0: (ядерная аффинность = 0-87) Разъем OpenMP: -1
Привет из потока 1: (ядерная аффинность = 0-87) Разъем OpenMP: -1
Привет из потока 2: (ядерная аффинность = 0-87) Разъем OpenMP: -1
Привет из потока 3: (ядерная аффинность = 0-87) Разъем OpenMP: -1
Привет из потока 4: (ядерная аффинность = 0-87) Разъем OpenMP: -1
    <... пропускаем результат ...>
Привет из потока 42: (ядерная аффинность = 0-87) Разъем OpenMP: -1
Привет из потока 43: (ядерная аффинность = 0-87) Разъем OpenMP: -1
  0.022119
```

Любое расположение процессора

Распечатка показывает отчет об аффинности и размещении без установки значений средовых переменных. Потоки могут выполняться на любом процессоре от 0 до 87

Ядерная аффинность позволяет потоку работать на любом из 88 виртуальных ядер.

Давайте посмотрим, что произойдет, когда мы разместим потоки на аппаратных ядрах и установим привязку аффинности равной close.

```
export OMP_PLACES=cores
export OMP_PROC_BIND=close
./vecadd_opt3
```

Результат с этими настройками аффинности и размещения показан на рис. 14.3.

Во это да! Мы фактически можем контролировать вычислительное ядро! Потоки теперь привязаны к двум виртуальным ядрам, принадлежащим одному аппаратному ядру. Время выполнения 0.0166 мс является последним числом в распечатке. Это время выполнения существенно улучшилось, по сравнению с 0.0221 мс в предыдущем прогоне, время вычислений сократилось на 25 %. Вы можете поэкспериментировать с разными значениями средовых переменных и проследить за тем, как потоки размещаются на узле.

```
Выполняется с 44 потоком(потоками)          Аппаратное ядро
  proc_bind равна close
  proc_num_places равно 44
Привет из потока 0: (ядерная аффинность = 0,44) Разъем OpenMP: 0
Привет из потока 1: (ядерная аффинность = 1,45) Разъем OpenMP: 1
Привет из потока 2: (ядерная аффинность = 2,46) Разъем OpenMP: 2
Привет из потока 3: (ядерная аффинность = 3,47) Разъем OpenMP: 3
Привет из потока 4: (ядерная аффинность = 4,48) Разъем OpenMP: 4
   <... пропускаем результат ...>
Привет из потока 42: (ядерная аффинность = 42,86) Разъем OpenMP: 42
Привет из потока 43: (ядерная аффинность = 43,87) Разъем OpenMP: 43
  0.016601
```

Рис. 14.3 Отчет об аффинности и размещении для `OMP_PLACES=cores` и `OMP_PROC_BIND=close`. Каждый поток может выполняться на двух возможных виртуальных ядрах. Эти два виртуальных процессора принадлежат к одному аппаратному ядру из-за гиперпотокообразования

Мы собираемся автоматизировать разведывательный анализ всех значений и того, как они масштабируются вместе с разным числом потоков. Мы отключим опцию детализации, чтобы сократить распечатку, с которой приходится иметь дело. Будет печататься только время выполнения. Удалите предыдущую сборку и пересоберите исходный код следующим образом:

```
mkdir build && cd build
cmake ..
make
```

Затем мы выполняем скрипт в следующем ниже листинге, чтобы получить производительность для всех случаев.

Листинг 14.1 Скрипт автоматизации разведывательного анализа всех значений

`OpenMP/run.sh`

```
1 #!/bin/sh
2
3 calc_avg_stddev()
```

Вычисляет среднее значение и стандартное отклонение

```
 4 {
 5    #echo "Runtime is $1"
 6    awk '{
 7       sum = 0.0; sum2 = 0.0         # Инициализировать нулями
 8       for (n=1; n <= NF; n++) {     # Обработать каждое значение в строке
 9          sum += $n;                 # Скользящая сумма значений
10          sum2 += $n * $n            # Скользящая сумма квадратов
11    }
12    print " Число испытаний=" NF ", среднее=" sum/NF ", \
          стд.откл.=" sqrt((sum2 - (sum*sum)/NF)/NF);
13      }' <<< $1
14 }
15
16 conduct_tests()
17 {
18    echo ""
19    echo -n `printenv |grep OMP_` ${exec_string}
20    foo=""
21    for index in {1..10}
22    do
23       time_result=`${exec_string}`
24       time_val[$index]=${time_result}
25       foo="$foo ${time_result}"
26    done
27    calc_avg_stddev "${foo}"
28 }
29
30 exec_string="./vecadd_opt3 "
31
32 conduct_tests
33
34 THREAD_COUNT="88 44 22 16 8 4 2 1"
35
36 for my_thread_count in ${THREAD_COUNT}
37 do
38    unset OMP_PLACES
39    unset OMP_PROC_BIND
40    export OMP_NUM_THREADS=${my_thread_count}
41
42    conduct_tests
43
44    PLACES_LIST="threads cores sockets"
45    BIND_LIST="true false close spread primary"
46
47    for my_place in ${PLACES_LIST}
48    do
49       for my_bind in ${BIND_LIST}
50       do
51          export OMP_NUM_THREADS=${my_thread_count}
52          export OMP_PLACES=${my_place}
53          export OMP_PROC_BIND=${my_bind}
54
```

Проходит ли тест

Повторяет десять раз, чтобы набрать статистику

Прокручивает в цикле по числу потоков

Прокручивает в цикле по значениям местоположения

Прокручивает в цикле по значениям аффинности

```
55            conduct_tests
56          done
57      done
58  done
```

Из-за нехватки места на рис. 14.4 показано только несколько результатов. Все значения являются ускорением по сравнению с одним потоком без значений аффинности или размещения.

Первое, что следует отметить из рис. 14.4 в нашем анализе, – это то, что программа, в общем, работает быстрее всего для всех значений только с 44 потоками. В целом гиперпотокообразование не помогает. Исключением является значение `close` для потоков, потому что, если у нас нет более 44 потоков с этим значением, то во втором разъеме не будет процессов. Наличие потоков только в первом разъеме лимитирует суммарную получаемую пропускную способность памяти. При полных 88 потоках значение `close` для потоков обеспечивает наилучшую производительность, хотя и незначительно. Значение `close` в целом показывает тот же эффект лимитированной пропускной способности памяти из-за наличия потоков только в первом разъеме. Кроме того, можно увидеть, что при большем количестве процессов с привязкой процессов производительность выше, чем без привязки процессов.

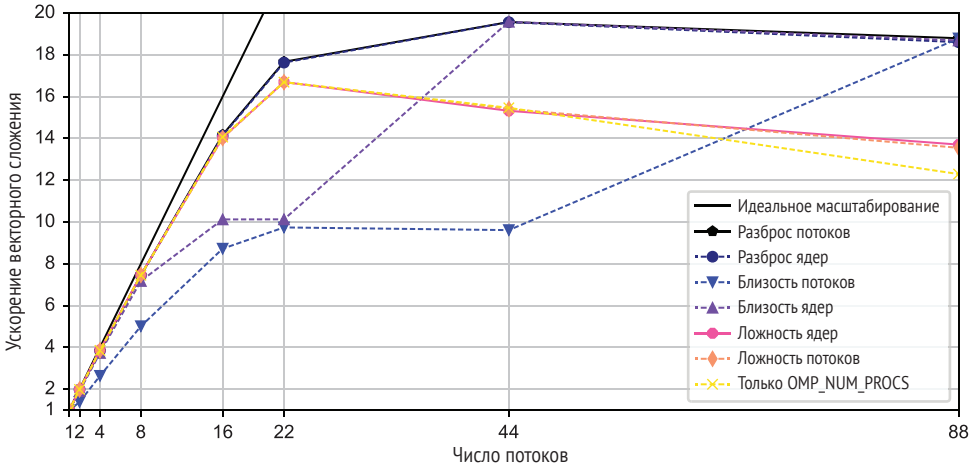


Рис. 14.4 Значения OpenMP аффинности и размещения для `OMP_PROC_BIND=spread` повышают параллельное масштабирование на 50 %. Линии показывают различные числа потоков для того или иного значения и в легенде упорядочены примерно от высокого до низкого

Отметим несколько ключевых моментов, которые следует извлечь из этого анализа.

- Гиперпотокообразование не помогает с простыми ограниченными памятью вычислительными ядрами, но и не вредит.
- Для вычислительных ядер, лимитированных пропускной способностью памяти, на нескольких разъемах (доменах NUMA) следует делать оба разъема занятыми работой.

Мы не показываем результаты установки значения средовой переменной `OMP_PROC_BIND` равным `primary`, потому что это заставляет все потоки работать на одном и том же процессоре и замедляет работу программы в целых два раза. Мы также не показываем установку значения средовой переменной `OMP_PLACES` равным `sockets`, потому что это дает более низкую производительность, чем приведенные выше.

14.4 Аффинность процессов с MPI

Применение аффинности с приложениями MPI тоже имеет выгоды, как описано в разделе 14.2. Оно помогает получать полную пропускную способность памяти и производительность кеша, не давая вычислительному ядру операционной системы осуществлять миграцию процессов на другие процессорные ядра. Обсуждение аффинности с OpenMPI обусловлено тем, что в нем есть самые общедоступные инструменты анализа аффинности и размещения процессов. Другие имплементации MPI, такие как MPICH, должны компилироваться с активированной поддержкой SLURM, что неприменимо к персональным машинам. В разделе 14.6 мы обсудим инструменты командной строки, которые можно использовать в более общих ситуациях. А пока давайте продолжим наш разведывательный анализ аффинности в OpenMPI!

14.4.1 Принятое по умолчанию размещение процессов с помощью OpenMPI

Вместо того чтобы оставлять размещение процессов планировщику вычислительного ядра, OpenMPI задает принятое по умолчанию размещение и аффинность. Принятые по умолчанию значения для OpenMPI варьируются в зависимости от числа процессов. Они таковы:

- процессы <= 2 (привязка к ядру);
- процессы > 2 (привязка к разъему);
- процессы > процессоры (привязка отсутствует).

Некоторые центры HPC могут устанавливать другие значения по умолчанию, такие как постоянная привязка к ядрам. Эта политика привязки, возможно, имеет смысл для большинства заданий MPI, но может становиться причиной проблем с приложениями, в которых используются MPI и потокообразование OpenMP. Все потоки будут привязаны к одному единственному процессору, сериализующему потоки.

Последние версии OpenMPI имеют обширную поддержку размещения и аффинности процессов. Используя эти инструменты, вы обычно получаете прирост производительности. Выигрыш зависит от того, как планировщик процессов в операционной системе оптимизирует размещение. Большинство программ настроено для общих вычислений, таких как обработка текстов и электронных таблиц, но не для параллельных приложений. Уговаривая планировщика «поступать правильно», можно

получать потенциальную выгоду в размере 5–10 %, но этот процент бывает намного больше.

14.4.2 Взятие под контроль: базовые техники специфицирования размещения процессов в OpenMPI

Для размещения процессов и их привязки к компонентам оборудования в большинстве случаев использования достаточно применять простые элементы контроля. Эти элементы контроля передаются команде `mpirun` в качестве опций. Давайте начнем с того, что рассмотрим распределение процессов поровну между многоузловыми заданиями. Проще всего продемонстрировать это на примере.

Пример: распределение процессов поровну между многоузловыми заданиями

У нас есть приложение, которое мы хотим выполнять на 32 рангах MPI, но это приложение поглощает много памяти, требуя полтерабайта памяти. Одиночный узел не имеет достаточного объема памяти. Тогда как нам это уладить?

Если мы посмотрим на детали системы, то каждый узел имеет два разъема, заполненных процессорами Intel Broadwell (E52695). Каждый CPU имеет 18 аппаратных ядер, что при использовании гиперпотокообразования дает нам 36 виртуальных процессоров на разъем. Каждый узел имеет 128 гигабайт памяти.

- Из команды `lscpu`:

```
NUMA node0 CPU(s):    0-17,36-53
NUMA node1 CPU(s):    18-35,54-71
```

- Из файла /proc/meminfo:

```
MemTotal: 131728700 kB
```

В этом примере мы используем наш инструмент отчетности о размещении для приложений MPI. Две части исходного кода показаны в следующих ниже листингах.

Главный исходный MPI-код аффинности

MPI/MPIAffinity.c

```
 1 #include <mpi.h>
 2 #include <stdio.h>
 3 #include "place_report_mpi.h"
 4 int main(int argc, char **argv)
 5 {
 6    MPI_Init(&argc, &argv);
 7
 8    place_report_mpi();
 9
10    MPI_Finalize();
11    return 0;
12 }
```

Вставляет вызов отчетности о размещении после MPI_Init

Нам нужно вставить вызов отчетности о размещении в нашу подпрограмму после инициализации MPI. Его также можно легко добавить и в свое приложение MPI. Теперь давайте взглянем на подпрограмму отчетности в следующем ниже листинге.

Инструмент MPI отчетности о размещении

MPI/place_report_mpi.c

```
40 void place_report_mpi(void)
41 {
42    int rank;
43    cpu_set_t coremask;
44    char clbuf[7 * CPU_SETSIZE], hnbuf[64];
45
46    memset(clbuf, 0, sizeof(clbuf));
47    memset(hnbuf, 0, sizeof(hnbuf));
48
49    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
50
51    gethostname(hnbuf, sizeof(hnbuf));
52    sched_getaffinity(0, sizeof(coremask), &coremask);
53    cpuset_to_cstr(&coremask, clbuf);
54    printf("Привет из ранга %d, на %s. (ядерная аффинность = %s)\n",
55           rank, hnbuf, clbuf);
56 }
```

Та же самая процедура cpuset_to_cstr из листинга векторного сложения в разделе 14.3

Получает значение аффинности для нашего процесса

Получает имя нашего узла

В первом выполнении нашего приложения мы просим команду mpirun просто запустить 32 процесса:

```
mpirun -n 32 ./MPIAffinity | sort -n -k 4
```

Затем мы должны отсортировать результат по данным в четвертом столбце, потому что упорядоченный по процессам результат является случайным (выполняется командой `sort -n -k 4`). Результат на выходе из этой команды с нашей процедурой отчетности о размещении показан на рис. 14.5.

```
Привет из ранга 0, на cn328. (ядерная аффинность = 0-17,36-53)
Привет из ранга 1, на cn328. (ядерная аффинность = 18-35,54-71)
Привет из ранга 2, на cn328. (ядерная аффинность = 0-17,36-53)
Привет из ранга 3, на cn328. (ядерная аффинность = 18-35,54-71)
  <... пропускаем результат ...>
Привет из ранга 28, on cn328. (ядерная аффинность = 0-17,36-53)
Привет из ранга 29, on cn328. (ядерная аффинность = 18-35,54-71)
Привет из ранга 30, on cn328. (ядерная аффинность = 0-17,36-53)
Привет из ранга 31, on cn328. (ядерная аффинность = 18-35,54-71)
```

Участок NUMA

Рис. 14.5 Для `mpirun -n 32` все наши процессы находятся на узле cn328. Аффинность назначается участку NUMA (socket)

Из распечатки на рис. 14.5 мы видим, что все ранги запускались на узле cn328. Ссылаясь на принятые по умолчанию значения аффинности для OpenMPI в начале этого раздела, для более чем двух рангов аффинность назначается для привязки к разъему (socket). Результат на выходе из команды `lscpu` показывает, что наш первый участок NUMA содержит виртуальные процессорные ядра 0-17, 36-53. Участки NUMA обычно выравниваются с каждым разъемом. В нашей распечатке мы видим, что ядерная аффинность равна 0-17, 36-53, подтверждая, что аффинность получила значение `socket`.

Поскольку требования к памяти в нашем реальном приложении превышают 128 ГиБ на узел, при выделении памяти приложение отказывает. Следовательно, нам необходимо найти способ распределять эти процессы. Для этого мы добавляем еще одну опцию, `--npernode<#>` или `-N<#>`, которая сообщает MPI о том, сколько рангов нужно размещать на каждом узле. Нам нужно иметь четыре узла, чтобы получить достаточно памяти для нашей задачи, поэтому нужно восемь процессов на узел.

```
mpirun -n 32 --npernode 8 ./MPIAffinity | sort -n -k 4
```

На рис. 14.6 показан наш отчет о размещении.

```
Привет из ранга 0, на cn328. (ядерная аффинность = 0-17,36-53)
Привет из ранга 1, на cn328. (ядерная аффинность = 18-35,54-71)
  < ... пропускаем результат ... >
Привет из ранга 8, на cn329. (ядерная аффинность = 0-17,36-53)
Привет из ранга 9, на cn329. (ядерная аффинность = 18-35,54-71)
  < ... пропускаем результат ... >
Привет из ранга 16, на cn330. (ядерная аффинность = 0-17,36-53)
Привет из ранга 17, на cn330. (ядерная аффинность = 18-35,54-71)
  < ... пропускаем результат ... >
Привет из ранга 24, на cn331. (ядерная аффинность = 0-17,36-53)
Привет из ранга 25, на cn331. (ядерная аффинность = 18-35,54-71)
```

Участок NUMA

Рис. 14.6 Процессы MPI распределены по четырем узлам, с 328 по 331. Аффинность по-прежнему привязана к участку NUMA

Из результата на рис. 14.6 видно, что мы работаем на четырех узлах. Теперь у нас должно быть достаточно памяти для выполнения приложения. В качестве альтернативы мы могли бы указать число рангов в расчете на разъем с помощью опции `--npersocket`. У нас два разъема на узел, поэтому нам нужно четыре ранга на разъем, отсюда:

```
mpirun -n 32 --npersocket 4 ./MPIAffinity | sort -n -k 4
```

На рис. 14.7 показан результат размещения в расчете на разъем.

Отчет о размещении на рис. 14.7 показывает, что порядок рангов размещает смежные ранги в одном домене NUMA вместо чередования рангов между доменами NUMA. Было бы еще лучше, если бы ранги обменивались данными с ближайшими соседями.

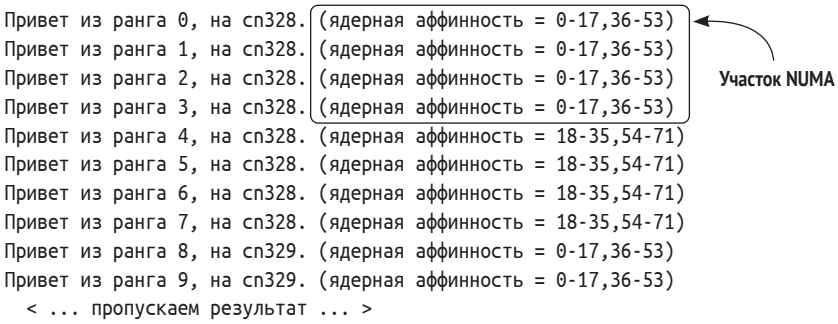


Рис. 14.7 При размещении, установленном равным четырем процессам на разъем, порядок рангов меняется. Теперь четыре смежных ранга находятся в одном участке NUMA

До сих пор мы работали только над размещением процессов. Теперь давайте попробуем посмотреть, что можно сделать с аффинностью и привязыванием процессов MPI. Для этого мы добавляем опцию `--bind-to [socket | numa | core | hwthread]` в команду `mpirun`:

```
mpirun -n 32 --npersocket 4 --bind-to core ./MPIAffinity | sort -n -k 4
```

В отчете о размещении на рис. 14.8 мы видим, как она изменяет аффинность процессов.

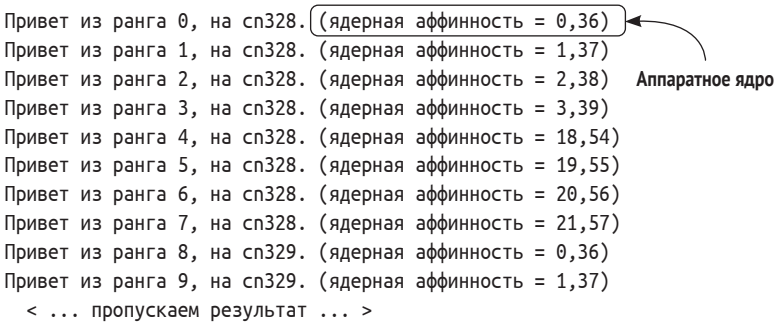


Рис. 14.8 Привязка к ядру изменяет аффинность процессов с аппаратным ядром. По причине гиперпотокообразования каждое аппаратное ядро представляет два виртуальных ядра. Для каждого процесса мы получаем два местоположения

Результаты размещения на рис. 14.8 показывают, что аффинность процессов теперь ограничена больше, чем было ранее. Каждый процесс может выполняться на двух виртуальных ядрах. Эти два виртуальных ядра принадлежат одному аппаратному ядру, таким образом показывая, что опция привязки ядра относится к аппаратному ядру. На каждом разъеме используется только четыре из 18 процессорных ядер. Это то, что мы и хотим, т. е. чтобы для каждого ранга MPI было больше памяти. Давайте попробуем привязать процесс не к ядру, а к гиперпотокам, ис-

пользуя опцию `hwthread`. Это должно заставить планировщик размещать процессы на одном и только одном виртуальном ядре.

```
mpirun -n 32 --npersocket 4 --bind-to hwthread ./MPIAffinity | sort -n -k 4
```

Опять же, мы используем нашу программу отчетности о размещении для визуализации размещения, а результаты ее работы показаны на рис. 14.9.

```
Привет из ранга 0, на cn328. (ядерная аффинность = 0)
Привет из ранга 1, на cn328. (ядерная аффинность = 36)
Привет из ранга 2, на cn328. (ядерная аффинность = 1)
Привет из ранга 3, на cn328. (ядерная аффинность = 37)
Привет из ранга 4, на cn328. (ядерная аффинность = 18)
Привет из ранга 5, на cn328. (ядерная аффинность = 54)
Привет из ранга 6, на cn328. (ядерная аффинность = 19)
Привет из ранга 7, на cn328. (ядерная аффинность = 55)
Привет из ранга 8, на cn329. (ядерная аффинность = 0)
Привет из ранга 9, на cn329. (ядерная аффинность = 36)
  < ... пропускаем результат ... >
```

Процессы на паре гиперпотоков на одном единственном ядре

Рис. 14.9 Размещение процессов посредством опции `hwthread` ограничивает местоположения, в которых процессы могут выполняться, сводя их только к одному месту

Наша последняя процессорная компоновка, наконец, ограничивает местоположения, в которых каждый процесс может выполняться, одним местом, как показано на рис. 14.9. На первый взгляд, этот результат неплохой. Но, приглядевшись, мы видим, что первые два ранга располагаются на паре гиперпотоков (0 и 36) одного аппаратного ядра. Это не очень хорошая идея, так как означает, что два ранга используют кеш и аппаратные компоненты этого аппаратного ядра совместно вместо того, чтобы иметь свой собственный полный комплект ресурсов.

Команда `mpirun` в OpenMPI также имеет встроенную опцию, позволяющую получать информацию о привязках. Это удобно для малых задач, но результирующий объем выходных данных для узлов с большим числом процессоров и рангов MPI настолько велик, что им трудно управлять. Добавление опции `--report-bindings` в команду `mpirun`, используемую для рис. 14.9, производит результат, показанный на рис. 14.10.

Визуальная схема немного проще для быстрого понимания, и в выходных данных содержится много информации. Каждая строка указывает ранг в `MPI_COMM_WORLD (MCW)`. Символы между прямонаправленными косыми чертами с правой стороны указывают место привязки для этого процесса. Набор из двух точек между символами косой черты показывает, что на ядро приходится два гиперпотока. Два набора скобок очерчивают два разъема на узле.

Благодаря примерам, которые мы рассмотрели в этом разделе, вы должны получить представление о том, как осуществлять контроль за

размещением и аффинностью. У вас также должны быть некоторые инструменты, которые позволяют делать проверку того, что вы получаете ожидаемое размещение и привязки процессов.

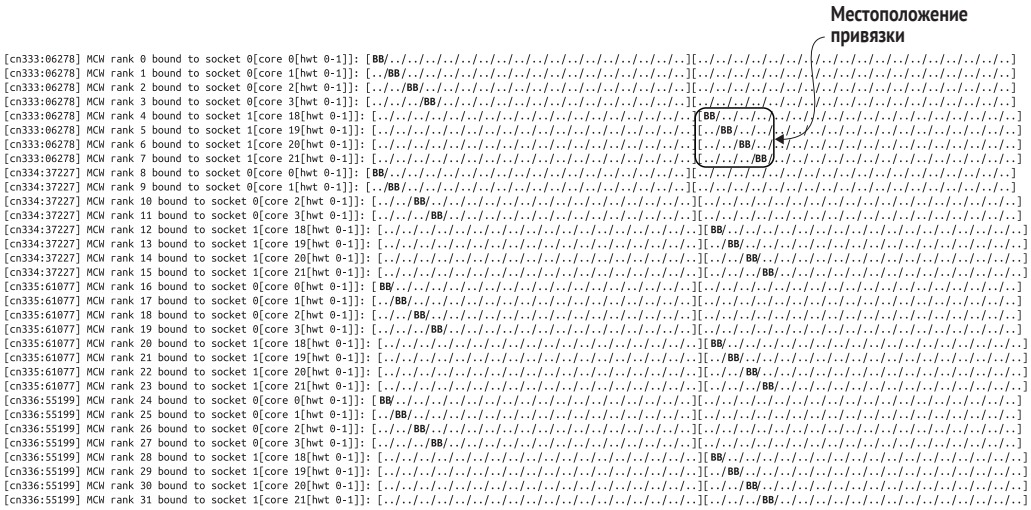


Рис. 14.10 Отчет о размещении из опции `--report-bindings` в команде `mpirun` показывает буквой *B* места, где ранги привязаны

14.4.3 Аффинность – это больше, чем просто привязывание процессов: полная картина

Теперь мы займемся разведывательным анализом полной картины аффинности для параллельных вычислений. Мы будем использовать его как способ введения продвинутых опций, предлагаемых в OpenMPI, для еще большего контроля.

Концепция аффинности рождается из того, как операционная система видит вещи. На уровне операционной системы вы можете установить, где каждому процессу разрешено выполняться. В Linux это делается либо посредством команды `taskset`, либо посредством команды `numactl`. Указанные команды и аналогичные утилиты в других операционных системах появлялись по мере роста сложности CPU, чтобы иметь возможность предоставлять больше информации планировщику в операционной системе. Эти указания могут восприниматься планировщиком как подсказки или требования. Используя эти команды, вы можете привязывать серверный процесс к конкретному процессору, чтобы быть ближе к тому или иному аппаратному компоненту или получать более быстрый отклик. Одного этого внимания на аффинности достаточно, когда речь идет об одном процессе.

Для параллельного программирования существуют дополнительные соображения. Мы должны рассматривать набор процессов. Допустим, у нас 16 процессоров, и мы выполняем четырехранговое задание MPI.

Куда мы помещаем ранги? Помещаем ли мы их по всем разъемам, на все разъемы, упаковываем их близко друг к другу или же распределяем равномерно? Размещаем ли мы некоторые ранги рядом друг с другом (ранги 1 и 2 вместе либо ранги 1 и 4 вместе)? В целях ответа на эти вопросы нам необходимо обратиться к следующим ниже аспектам:

- соотнесение (размещение процессов);
- порядок рангов (какие ранги лежат близко друг к другу);
- привязка (аффинность или увязка процесса с местоположением или местоположениями).

Мы рассмотрим каждый из них по очереди, а также то, как OpenMPI позволяет вам контролировать эти вещи.

Соотнесение процессов с процессорами или другими местоположениями

Когда мы думаем о параллельном приложении, у нас есть набор процессов и набор процессоров. Как мы соотносим процессы с процессорами? В примере, приведенном в разделе 14.4.2, мы хотели распределить процессы по четырем узлам, чтобы у каждого процесса было больше памяти, чем если бы он находился на одном узле. Более общей формой соотнесения процессов в OpenMPI является опция `-mapby hwresource`, где аргумент `hwresource` – это любой из большого числа аппаратных компонентов. К наиболее распространенным относятся следующие:

```
--map-by [slot | hwthread | core | socket | numa | node]
```

С помощью опции `--map-by` для команды `mpirun` процессы распределяются по этому аппаратному ресурсу по круговой схеме. Для указанной опции по умолчанию используется `socket`. Большинство этих местоположений оборудования не требует пояснений, за исключением слота. Слоты – это список возможных местоположений для процессов из среды, планировщика или файла хоста. Эта форма опции `--map-by` по-прежнему лимитирована по своему смыслу и, следовательно, по своему эффекту.

В более общей форме используется опция `ppr`, или процессы в расчете на ресурс, где `n` – это число процессов. Вместо соотнесения по круговой схеме в соответствии с ресурсом можно указывать блок процессов в расчете на аппаратный ресурс:

```
--map-by ppr:n:hwresource
```

Или в более явной форме:

```
--map-by ppr:n:[slot | hwthread | core | socket | numa | node]
```

В наших предыдущих примерах мы использовали более простой вариант `--npernode 8`. В этой более общей форме это было бы сокращением для:

```
--map-by ppr:8:node
```

Если уровень контроля из предыдущих опций для команды `mpirun` недостаточен, то можно указать список номеров процессоров для соотнесения с опцией `--cpu-list <номера логических процессоров>`, где номера процессоров – это список, соответствующий списку из `lstopo` либо `lscpu`. Эта опция также одновременно привязывает процессы к логическому (виртуальному) процессору.

Упорядочение рангов MPI

Еще одна вещь, которую вы, возможно, захотите контролировать, – это порядок ваших рангов MPI. Возможно, вам захочется, чтобы смежные ранги MPI были близки друг к другу в пространстве физического процессора, если они много обмениваются друг с другом данными. Это снижает стоимость обмена данными между такими рангами. Обычно достаточно контролировать это с помощью блочного размера распределения во время соотнесения, но дополнительный контроль можно получить посредством опции `--rank-by`:

```
--rank-by ppr:n:[slot | hwthread | core | socket | numa | node]
```

Еще более общая опция состоит в использовании рангового файла:

```
--rankfile <имя_файла>
```

Хотя и можно точно отрегулировать размещение ваших рангов MPI посредством этих команд и, пожалуй, повысить производительность на пару процентов, подобрать оптимальную формулу довольно трудно.

Привязывание процессов к аппаратным компонентам

Последнее, что нужно контролировать, – это саму аффинность. *Аффинность* – это процесс привязывания процесса к аппаратному ресурсу. Указанная опция аналогична предыдущим:

```
--bind-to [slot | hwthread | core | socket | numa | node]
```

Используемого по умолчанию значения core достаточно для большинства приложений MPI (без опции `--bind-to` по умолчанию используется `socket` для более чем двух процессов, как указано в разделе 14.4.1). Но бывают случаи, когда такое значение аффинности вызывает проблемы.

Как мы видели в примере на рис. 14.8, аффинность назначается двум гиперпотокам на аппаратном ядре. Возможно, мы захотим попробовать `--map-to core --bind-to hwthread`, чтобы распределять процессы по ядрам, но более тесно привязать каждый процесс к одному гиперпотоку. Разница в производительности от такой тонкой регулировки, вероятно, невелика. Более крупная проблема возникает, когда мы пытаемся имплементировать гибридное приложение MPI и OpenMP. Важно понимать, что дочерние процессы наследуют настройки аффинности у своих родителей. Если мы используем опции `npersocket 4 --bind-to core`, а затем

запускаем два потока, то у нас есть два местоположения для выполнения потоков (два гиперпотока в расчете на ядро), так что все в порядке. Если мы запустим четыре потока, то они будут использовать совместно только два логических процессора, и производительность будет лимитирована.

Ранее в данном разделе мы увидели, что существует масса вариантов контроля за процессом, размещением и аффинностью. И действительно существует слишком много комбинаций, чтобы даже разведать их полностью, как мы сделали в разделе 14.3 для OpenMP. В большинстве случаев мы должны быть удовлетворены получением разумных настроек, которые отражают потребности наших приложений.

14.5 Аффинность для MPI плюс OpenMP

В этом разделе наша цель состоит в том, чтобы понять, как задавать аффинность для гибридных приложений MPI и OpenMP. Получать правильную аффинность для этих гибридных ситуаций бывает непросто. Для этого разведывательного анализа мы создали пример гибридной потоковой триады с MPI и OpenMP. Мы также изменили используемый в этой главе отчет о размещении, выводя информацию для гибридных приложений MPI и OpenMP. В следующем ниже листинге показана модифицированная подпрограмма `place_report_ mpi_omp.c`.

Листинг 14.2 Гибридная потоковая триада с инструментом MPI и OpenMP отчетности о размещении

StreamTriad/place_report_mpi_omp.c

```
41 void place_report_mpi_omp(void)
42 {
43    int rank;
44    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
45
46    int socket_global[144];
47    char clbuf_global[144][7 * CPU_SETSIZE];
48
49    #pragma omp parallel
50    {
51       if (omp_get_thread_num() == 0 && rank == 0){
52          printf("Выполнение с %d потоком(потоками)\n",omp_get_num_threads());
53          int bind_policy = omp_get_proc_bind();
54          switch (bind_policy)
55          {
56             case omp_proc_bind_false:
57                printf(" proc_bind равна false\n");
58                break;
59             case omp_proc_bind_true:
60                printf(" proc_bind равна true\n");
61                break;
```

```
62             case omp_proc_bind_master:
63                printf(" proc_bind равна master\n");
64                break;
65             case omp_proc_bind_close:
66                printf(" proc_bind равна close\n");
67                break;
68             case omp_proc_bind_spread:
69                printf(" proc_bind равна spread\n");
70          }
71          printf(" proc_num_places равно %d\n",omp_get_num_places());
72       }
73
74       int thread = omp_get_thread_num();
75       cpu_set_t coremask;
76       char clbuf[7 * CPU_SETSIZE], hnbuf[64];
77       memset(clbuf, 0, sizeof(clbuf));
78       memset(hnbuf, 0, sizeof(hnbuf));
79       gethostname(hnbuf, sizeof(hnbuf));
80       sched_getaffinity(0, sizeof(coremask), &coremask);
81       cpuset_to_cstr(&coremask, clbuf);
82       strcpy(clbuf_global[thread],clbuf);
83       socket_global[omp_get_thread_num()] = omp_get_place_num();
84       #pragma omp barrier
85       #pragma omp master
86       for (int i=0; i<omp_get_num_threads(); i++){
87          printf("Привет из ранга %02d,"
                 " поток %02d, на %s."
88               " (ядерная аффинность = %2s)"
                 " Разъем OpenMP: %2d\n",
89               rank, i, hnbuf,
                 clbuf_global[i],
                 socket_global[i]);
90       }
91    }
92 }
```

Объединяет отчеты OpenMP и MPI об аффинности

Мы начинаем этот пример с компилирования приложения потоковой триады. Исходный код потоковой триады находится по адресу https://github.com/EssentialsofParallelComputing/Chapter14 в каталоге StreamTriad. Скомпилируйте этот исходный код с помощью:

```
mkdir build && cd build
./cmake -DCMAKE_VERBOSE=1 ..
make
```

Мы выполнили указанный код на нашем CPU Skylake Gold с 44 аппаратными процессорами и двумя гиперпотоками каждый. Мы разместили два потока OpenMP на гиперпотоках, а затем ранг MPI на каждом аппаратном ядре. Приведенные ниже команды позволяют достичь этой схемы:

```
export OMP_NUM_THREADS=2
mpirun -n 44 --map-by socket ./StreamTriad
```

Исходный код потоковой триады содержит вызов нашего отчета о размещении из листинга 14.2. На рис. 14.11 показана распечатка.

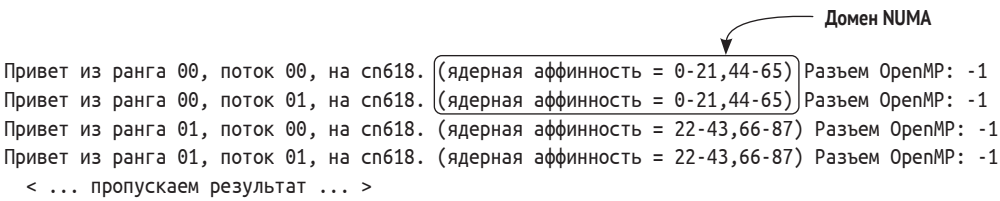


Рис. 14.11 Ранги MPI размещаются по круговой схеме по разъемам с двумя слотами, обеспечивая место для двух потоков OpenMP. Размещение ограничено доменом NUMA, чтобы держать память рядом с потоками. Процессы не привязаны плотно к какому-либо конкретному виртуальному ядру, и планировщик может свободно перемещать их в домене NUMA

Как видно из распечатки на рис. 14.11, нам удалось распределить ранги по доменам NUMA по круговой схеме, держа два потока вместе. За счет этого должна обеспечиваться хорошая пропускная способность основной памяти. Аффинностные ограничения достаточны только для того, чтобы процессы оставались в домене NUMA и позволяли планировщику перемещать процессы по своему усмотрению. Планировщик может размещать поток 0 на любом из 44 разных виртуальных процессоров, включая 0–21 либо 44–65. Возможно, нумерация сбивает с толку; 0 и 44 – это два гиперпотока на одном физическом ядре.

Теперь давайте попробуем получить больше аффинностных ограничений. Для этого нам нужно использовать форму `-mapby ppr:N:socket:PE=N`. Указанная команда дает нам возможность распределять процессы с заданным интервалом и указывать число рангов MPI, которые следует размещать на каждом разъеме. В сложности этой опции довольно трудно разобраться.

Давайте начнем с части `ppr:N:socket`. Мы хотим, чтобы половина наших рангов MPI была на каждом разъеме. Это должно составлять 22 ранга MPI на сокет или `ppr:22:socket`. Последняя часть определяет число процессоров, которые мы хотим иметь между размещением процессов. Нам нужно два потока для каждого ранга MPI, поэтому необходимо два виртуальных процессора в каждом блоке. Указанная спецификация предназначена для аппаратных ядер. Важно знать, что каждое аппаратное ядро содержит два виртуальных процессора. Следовательно, вам нужно только одно аппаратное ядро (`PE=1`). Затем мы прикрепляем потоки к аппаратному потоку. Для ранга 0 мы должны получить первое аппаратное ядро с виртуальными процессорами 0 и 44. В результате мы имеем команды, которые приведены ниже:

```
export OMP_NUM_THREADS=2
export OMP_PROC_BIND=true
mpirun -n 44 --map-by ppr:22:socket:PE=1 ./StreamTriad
```

Уф! Это было сложно. Все ли мы сделали правильно? Значит так, давайте проверим результат на выходе из команды, как показано на рис. 14.12.

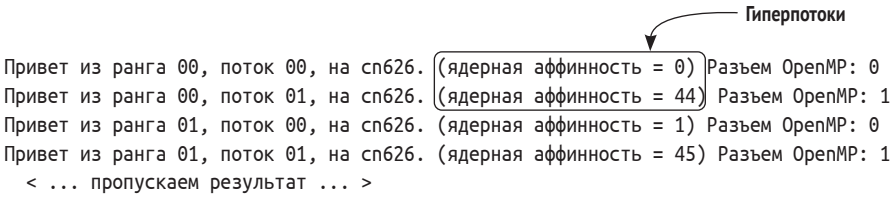


Рис. 14.12 Аффинность процессов и потоков теперь ограничена логическим ядром, и два потока OpenMP в расчете на ранг расположены в гиперпоточных парах (на рисунке это 0 и 44). Ранги упакованы плотно, чтобы снизить затраты на обмен данными для более сложных программ. Ранги MPI привязаны к аппаратным ядрам, а аффинность потоков – к гиперпотоку

Из распечатки на рис.14.12 видно, что потоки замкнуты там, где мы хотим. У нас также есть ранг MPI, закрепленный за аппаратным ядром. Это можно проверить, сбросив средовую переменную `OMP_PROC_BIND` (`unset OMP_PROC_BIND`), и результат (рис. 14.13) подтверждает, что ранг привязан к двум логическим процессорам, составляющим одно аппаратное ядро.

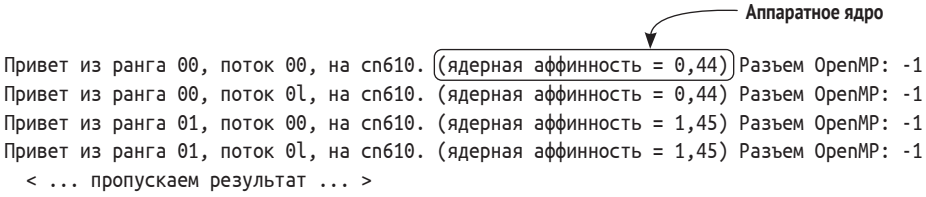


Рис. 14.13 Распечатка без OMP_PROC_BIND=true показывает, что ранги MPI закреплены за аппаратным ядром

Мы проработали один случай и смогли получить значения аффинности в том ключе, в каком хотели. Но теперь вы хотите знать, сможем ли мы выполнять более двух потоков OpenMP и какова производительность программы. Давайте взглянем на набор команд, проверяющих любое число потоков, которое нацело делится на число процессоров. В следующем ниже листинге показаны ключевые скриптовые команды.

Листинг 14.3 Настройка аффинности для гибрида MPI и OpenMP

Extracted from StreamTriad/run.sh

```
1 #!/bin/sh
2 LOGICAL_PES_AVAILABLE=`lscpu |\
    grep '^CPU(s):' |cut -d':' -f 2`
3 SOCKETS_AVAILABLE=`lscpu |\
    grep '^Socket(s):' |cut -d':' -f 2`
4 THREADS_PER_CORE=`lscpu |\
    grep '^Thread(s) per core:' |cut -d':' -f 2`
5 POST_PROCESS="|& grep -e Average -e mpirun |sort -n -k 4"
6 THREAD_LIST_FULL="2 4 11 22 44"
7 THREAD_LIST_SHORT="2 11 22"
```

Получает характеристики оборудования

```
 8
 9 unset OMP_PLACES
10 unset OMP_CPU_BIND
11 unset OMP_NUM_THREADS
12

    < ... базовые тесты не показаны ... >

21
22 export OMP_PROC_BIND=true

    < ... первый циклический блок не показан ... >

37 for num_threads in ${THREAD_LIST_FULL}
38 do
39    export OMP_NUM_THREADS=${num_threads}}
40
41    HW_PES_PER_PROCESS=$((${OMP_NUM_THREADS}/
                           ${THREADS_PER_CORE}))
42    MPI_RANKS=$((${LOGICAL_PES_AVAILABLE}/ \
                  ${OMP_NUM_THREADS}))
43    PES_PER_SOCKET=$((${MPI_RANKS}/\
                       ${SOCKETS_AVAILABLE}))
44
45    RUN_STRING="mpirun -n ${MPI_RANKS} \
        --map-by ppr:${PES_PER_SOCKET}:socket:PE=${HW_PES_PER_PROCESS} \
        ./StreamTriad ${POST_PROCESS}"
46    echo ${RUN_STRING}
47    eval ${RUN_STRING}
48 done

     < ... дополнительные циклические блоки ... >
```

Устанавливает средовые переменные OMP

Вычисляет необходимые значения

Заполняет команду mpirun

Для того чтобы сделать этот скрипт переносимым, мы получаем характеристики оборудования с помощью команды `lscpu`. Затем задаем необходимые средовые параметры OpenMP. Мы могли бы установить значение `OMP_PROC_BIND` равным `true`, `close` либо `spread` с тем же результатом для данного случая, когда все слоты заполнены. Затем мы вычисляем переменные, необходимые для команды `mpirun`, и запускаем задание.

В полном примере потоковой триады из листинга 14.2 мы протестировали комбинацию размеров потоков и рангов MPI, которые нацело делятся на 88 процессов. Мы проследили за этим с 44 процессами, в которых мы пропускаем гиперпотоки, потому что с ними мы на самом деле не получили никакого улучшения (раздел 14.3). Результаты производительности довольно постоянны на протяжении всего набора тестов. Это связано с тем, что измеряется только пропускная способность основной памяти. Там мало работы и нет обмена данными по линии MPI. Выгоды гибрида MPI и OpenMP в этой ситуации лимитированы. Мы ожидали бы увидеть выгоды в гораздо более масштабных симуляциях, где замена потока OpenMP на ранг MPI будет:

- сокращать потребности в буферной памяти MPI;
- создавать более крупные домены, которые консолидируют и сокращают участки призрачных ячеек;
- сокращать соперничество между процессорами на узле за один сетевой интерфейс;
- обращаться к векторным модулям и другим компонентам процессора, которые используются не полностью.

14.6 Контроль за аффинностью из командной строки

Кроме всего прочего, существуют общие подходы к контролю за аффинностью из командной строки. Инструменты командной строки помогают в ситуациях, когда ваше MPI- или специальное параллельное приложение не имеют встроенных опций для контроля за аффинностью. Эти инструменты также помогают в общецелевых приложениях путем их привязывания к важным аппаратным компонентам, таким как видеокарты, сетевые порты и устройства хранения данных. В данном разделе мы рассмотрим две консольные опции: комплект инструментов hwloc и likwid. Эти инструменты разработаны с учетом высокопроизводительных вычислений.

14.6.1 Использование инструмента hwloc-bind для назначения аффинности

Проект hwloc был разработан INRIA, Французским национальным институтом исследований в области компьютерных наук и автоматизации. Подпроект проекта OpenMPI, hwloc имплементирует способности по размещению и назначению аффинности OpenMPI, которые мы видели в разделах 14.4 и 14.5. Пакет hwloc также является автономным пакетом с инструментами командной строки. Поскольку существует масса инструментов hwloc, в качестве введения мы просто рассмотрим пару из них. Мы будем использовать `hwloc-calc` для получения списка аппаратных ядер и `hwloc-bind` для их привязывания.

Применять инструмент `hwloc-bind` очень просто. Надо лишь предварять приложение префиксом `hwloc-bind`, а затем добавлять аппаратное местоположение, в котором вы хотите его привязывать. В нашем приложении мы будем использовать команду `lstopo`. Команда `lstopo` также является частью инструментов hwloc. Ниже приведен наш однострочный способ запуска задания на всех аппаратных ядрах и привязывания процессов к ядрам:

```
for core in `hwloc-calc --intersect core --sep " " all`; do hwloc-bind \
    core:${core} lstopo --no-io --pid 0 & done
```

Опция `--intersect core` использует только аппаратные ядра. Выражение `-- sep " "` сообщает об отделении чисел в распечатке пробелами вместо запятых. Результат этой команды на нашем обычном процессоре Skylake Gold запускает 44 графических окна lstopo, каждое из которых выглядит аналогично изображенным на рис. 14.14. В каждом окне привязанные местоположения выделены зеленым цветом.

Рис. 14.14 На изображении lstopo зеленым цветом показано привязанное местоположение (затененное ядро) в левом нижнем углу. Оно показывает, что процесс 22 привязан к 22-му и 66-му виртуальным ядрам, которые являются гиперпотоками для одного физического ядра

Мы могли бы использовать аналогичную команду для запуска двух процессов на первом ядре каждого разъема. Например:

```
for socket in `hwloc-calc --intersect socket \
    --sep " " all`; do hwloc-bind \
    socket:${socket}.core:0 lstopo --no-io --pid 0 & done
```

В следующем ниже листинге показано, как строится общецелевая команда `mpirun` с привязыванием.

Листинг 14.4 Использование hwloc-bind для привязывания процессов

MPI/mpirun_distrib.sh

```
 1 #!/bin
 2 PROC_LIST=$1
 3 EXEC_NAME=$2
 4 OUTPUT="mpirun "
 5 for core in ${PROC_LIST}
 6 do
 7     OUTPUT="$OUTPUT -np 1"\
              " hwloc-bind core:${core}"\
              " ${EXEC_NAME} :"
 8 done
 9 OUTPUT=`echo ${OUTPUT} | sed -e 's/:$/\n/'`
10 eval ${OUTPUT}
```

Инициализирует этот строковый литерал значением mpirun

Дополняет еще один запуск ранга MPI привязкой

Удаляет последнюю кавычку и подставляет символ новой строки

Теперь мы можем запустить наше приложение аффинности MPI из раздела 14.4 на первом ядре каждого разъема с помощью следующей ниже команды:

```
./mpirun_distrib.sh "1 22" ./MPIAffinity
```

Данный скрипт `mpirun_distrib` строит следующую ниже команду и ее выполняет:

```
mpirun -np 1 hwloc-bind core:1 ./MPIAffinity : -np 1 hwloc-bind core:22
  ./MPIAffinity
```

14.6.2 Использование likwid-pin: инструмент аффинности в комплекте инструментов likwid

Инструмент likwid-pin является одним из многих замечательных инструментов коллектива likwid («будто я знаю, что делаю») из Университета Эрлангена. Мы встречали наш первый инструмент likwid под названием likwid-perfctr в разделе 3.3.1. Инструменты likwid в этом разделе являются инструментами командной строки для установки аффинности. Мы рассмотрим варианты инструмента для потоков OpenMP, MPI и гибридных приложений MPI плюс OpenMP. В базовом синтаксисе для выбора наборов процессоров в инструменте likwid используются следующие ниже опции:

- Default (физическая нумерация);
- `N` (нумерация на уровне узлов);
- `S` (нумерация на уровне разъемов);
- `C` (нумерация кеша на последнем уровне);
- `M` (нумерация NUMA-доменов памяти).

В целях установки аффинности используется следующий синтаксис: `-c <N, S, C, M>:[n1, n2, n3-n4]`. Для того чтобы получить список схем нумерации, используется команда `likwid-pin -p`. Понимание принципа работы инструмента `likwid-pin` лучше всего получить из примеров и экспериментов.

Закрепление потоков OpenMP с помощью инструмента likwid-pin

В этом примере показано, как использовать likwid-pin с приложениями OpenMP:

```
export OMP_NUM_THREADS=44
export OMP_PROC_BIND=spread
export OMP_PLACES=threads
./vecadd_opt3
```

В целях получения того же результата закрепления с помощью likwid-pin для приложений OpenMP мы используем опцию socket (`S`). Далее мы

распределяем 22 потока на каждом разъеме, где два набора контактов (pin) разделяются и конкатенируются символом `@`:

```
likwid-pin -c S0:0-21@S1:0-21 ./vecadd_opt3
```

Средовые переменные OMP не являются необходимыми при использовании likwid-pin и в основном игнорируются. Число потоков определяется из списков наборов контактов. Для указанной команды это 44. Мы выполнили пример vecadd из раздела 14.3, сконфигурированный с опцией `-DCMAKE_VERBOSE`, чтобы получить отчет о размещении, как показано на рис. 14.15.

```
[pthread wrapper]
[pthread wrapper] MAIN -> 0
[pthread wrapper] PIN MASK: 0->1 1->2 2->3 3->4 4->5 5->6 6->7 7->8 8->9 9->10
10->11 11->12 12->13 13->14 14->15 15->16 16->17 17->18 18->19 19->20 20->21
21->22 22->23 23->24 24->25 25->26 26->27 27->28 28->29 29->30 30->31 31->32
32->33 33->34 34->35 35->36 36->37 37->38 38->39 39->40 40->41 41->42 42->43
[pthread wrapper] SKIP MASK: 0x0
     threadid 47149577160576 -> core 1 - OK
     threadid 47149581363200 -> core 2 - OK
        < ... пропускаем результат... >
     threadid 47152378182656 -> core 42 - OK
     threadid 47152382389376 -> core 43 - OK
Выполнение с 44 потоков(потоками)
  proc_bind равна false
  proc_num_places равно 0
Привет из потока 0: (ядерная аффинность = 0) Разъем OpenMP: -1
Привет из потока 1: (ядерная аффинность = 1) Разъем OpenMP: -1
  < ... пропускаем результат ... >
Привет из потока 42: (ядерная аффинность = 42) Разъем OpenMP: -1
Привет из потока 43: (ядерная аффинность = 43) Разъем OpenMP: -1
  0.016692
```

Физическое ядро

Рис. 14.15 Результат работы likwid-pin находится в верхней части рисунка, за которым следует распечатка нашего отчета о размещении. Результат показывает, что потоки привязаны к 44 физическим ядрам

Наш отчет о размещении показывает, что средовые переменные OMP не заданы и что OpenMP не разместил и не закрепил потоки в разъемах OpenMP. И все же мы получаем такое же размещение и закрепление с помощью инструмента likwid-pin с одинаковыми результатами. *Мы только что подтвердили, что средовые переменные OMP не являются необходимыми для likwid-pin, как мы утверждали в предыдущем абзаце.* Следует отметить, что если вы зададите средовой переменной `OMP_NUM_THREADS` значение, отличное от числа потоков в наборах контактов, то инструмент likwid распределит потоки из переменной `OMP_NUM_THREADS` по процессорам, указанным в наборах контактов. Когда потоков больше, чем процессоров, инструмент обертывает размещение потоков вокруг доступных процессоров.

Закрепление рангов MPI с помощью инструмента likwid-mpirun

Функциональность likwid по закреплению для приложений MPI включена в инструмент likwid-mpirun. Этот инструмент можно использовать в качестве замены mpirun в большинстве имплементаций MPI. Давайте рассмотрим пример MPIAffinity из раздела 14.4.

Пример: закрепление рангов MPI с помощью инструмента likwid-mpirun

Выполните пример MPIAffinity на 44 рангах и используйте команду `likwid-mpirun`, чтобы привязать ранги к аппаратным ядрам. По умолчанию likwid-mpirun привязывает ранги к ядрам, поэтому нам нужно использовать команду `likwid-mpirun`, чтобы получить то, что мы обычно хотим, без каких-либо дополнительных опций:

```
likwid-mpirun -n 44 ./MPIAffinity |sort -n -k 4
```

На рис. 14.16 показаны результаты нашего отчета о размещении для этого примера.

```
WARN: Cannot extract OpenMP vendor from executable or commandline, assuming no OpenMP
Привет из ранга 0, на cn630. (ядерная аффинность = 0)
Привет из ранга 1, на cn630. (ядерная аффинность = 1)
< ... пропускаем результат ... .
Привет из ранга 42, на cn630. (ядерная аффинность = 42)
Привет из ранга 43, на cn630. (ядерная аффинность = 43)
```

Процессорное ядро

Рис. 14.16 Отчет о размещении для likwid-mpirun показывает, что каждый ранг привязан к ядрам в числовом порядке

Это было несложно! Как показано на рис. 14.16, инструмент likwid-mpirun закрепляет ранги за аппаратными ядрами. Давайте перейдем к примеру, где мы должны предоставить команде некоторые опции.

Пример: опции для закрепления рангов MPI с помощью инструмента likwid-mpirun

Мы начинаем с базовой команды:

```
likwid-mpirun -n 22 ./MPIAffinity |sort -n -k 4
```

Ранги распределены по первым 22 аппаратным ядрам на разъеме 0 и ни по одному на разъеме 1. Ранее мы показали, что вам необходимо распределять процессы между обоими разъемами, чтобы получать полную пропускную способность из основной памяти. Добавление опции `-nperdomain` позволяет нам указывать, сколько разъемов на домен NUMA и набор контактов S:11 получает верные номера для 11 рангов на разъеме. Теперь команда выглядит так:

```
likwid-mpirun -n 22 -nperdomain S:11 ./MPIAffinity |sort -n -k 4
```

14.7 Будущее: установка и изменение аффинности во время выполнения

Что делать, если пользователю не было необходимости заморачиваться аффинностью? Довольно трудно заставить пользователей применять запутанные вызовы для надлежащего размещения и закрепления процессов. Во многих случаях бывает разумнее внедрять логику закрепления в исполняемый файл. Один из подходов заключается в том, чтобы запрашивать информацию об оборудовании и устанавливать аффинность соответствующим образом. Не во многих приложениях этот подход уже применяется, но мы ожидаем, что в будущем их будет больше.

Некоторые приложения не только устанавливают свою аффинность во время выполнения, но и модифицируют ее, чтобы адаптироваться к изменяющимся характеристикам во время выполнения! Эта инновационная методика была разработана Сэмом Гутьерресом (Sam Gutiérrez) из Национальной лаборатории Лос-Аламоса в его библиотеке QUO. Возможно, у вас есть приложение, в котором все ранги MPI используются на узле, но оно вызывает библиотеку, которая применяет комбинацию рангов MPI и потоков OpenMP. Библиотека QUO предоставляет простой интерфейс, построенный поверх hwloc, для установки соответствующих аффинностей. Затем она может помещать настройки в стек, замораживать процессоры и устанавливать новую политику привязки. Мы рассмотрим примеры инициирования привязки процессов внутри вашего приложения и ее изменения во время выполнения в последующих разделах.

14.7.1 Настройка аффинности в исполняемом файле

Настройка размещения и аффинности процессов в приложении означает, что вам больше не придется иметь дело с запутанными командами mpirun либо переносимостью между имплементациями MPI. Здесь мы используем библиотеку QUO для имплементирования этой привязки ко всем ядрам процессора Skylake Gold. Библиотека QUO с открытым исходным кодом доступна по адресу https://github.com/LANL/libquo.git. Сначала мы создаем исполняемый файл в директории Quo и выполняем приложение с числом аппаратных ядер в вашей системе:

```
make autobind
mpirun -n 44 ./autobind
```

Исходный код автоматической привязки показан в листинге 14.5. Программа состоит из следующих ниже шагов. Наша процедура отчетности о размещении вызывается до и после, чтобы показать привязки процессов.

1. Инициализировать QUO.
2. Задать аффинности с аппаратными ядрами.
3. Распределить процессы и привязать их к ядрам.
4. Вернуться к первоначальным аффинностям.

Листинг 14.5 Использование QUO для привязывания процессов из вашего исполняемого файла

Quo/autobind.c

```
31 int main(int argc, char **argv)
32 {
33    int ncores, nnoderanks, noderank, rank, nranks;
34    int work_member = 0, max_members_per_res = 2, nres = 0;
35    QUO_context qcontext;
36
37    MPI_Init(&argc, &argv);
38    QUO_create(&qcontext, MPI_COMM_WORLD);
39    MPI_Comm_size(MPI_COMM_WORLD, &nranks);
40    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
41    QUO_id(qcontext, &noderank);
42    QUO_nqids(qcontext, &nnoderanks);
43    QUO_ncores(qcontext, &ncores);
44
45    QUO_obj_type_t tres = QUO_OBJ_NUMANODE;
46    QUO_nnumanodes(qcontext, &nres);
47    if (nres == 0) {
48       QUO_nsockets(qcontext, &nres);
49      tres = QUO_OBJ_SOCKET;
50    }
51
52    if ( check_errors(ncores, nnoderanks, noderank, nranks, nres) )
53       return(-1);
54
55    if (rank == 0)
56       printf("\nПривязка по умолчанию для процессов MPI\n\n");
57    place_report_mpi();
58
59    SyncIt();
60    QUO_bind_push(qcontext,
                    QUO_BIND_PUSH_PROVIDED,
61                  QUO_OBJ_CORE, noderank);
62    SyncIt();
63
64    QUO_auto_distrib(qcontext, tres,
                       max_members_per_res,
65                     &work_member);
66    if (rank == 0)
67       printf("\nПроцессы должны закрепляться за ядрами hw\n\n");
68    place_report_mpi();
69
70    SyncIt();
71    QUO_bind_pop(qcontext);
72    SyncIt();
73
74    QUO_free(qcontext);
75    MPI_Finalize();
76    return(0);
77 }
```

Инициализирует контекст QUO

Получает системную информацию

Сообщает о привязках, используемых по умолчанию

Настраивает новые привязки на ядро

Распределяет и привязывает ранги MPI

Сообщает о новых привязках

Выталкивает привязки и возвращается к первоначальным настройкам

Не следует забывать синхронизировать процессы при изменении привязок. В целях обеспечения синхронизации в следующем ниже листинге мы используем барьер MPI и вызов микросна в подпрограмме `SyncIT`.

Листинг 14.6 Подпрограмма синхронизации

`Quo/autobind.c`

```
23 void SyncIt(void)
24 {
25    int rank;
26    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
27    MPI_Barrier(MPI_COMM_WORLD);
28    usleep(rank * 1000);
29 }
```

Стандартный барьер MPI

Добавочный микросон

Результат на выходе из приложения автоматического привязывания autobind (рис. 14.17) четко показывает привязки, которые были изменены с разъемов на аппаратные ядра.

14.7.2 Изменение аффинностей процессов во время выполнения

Предположим, у нас есть приложение, одна часть которого хочет использовать все ранги MPI, а другая лучше всего работает с потоками OpenMP. В целях улаживания такой ситуации нам нужно переключать аффинности во время выполнения. Это как раз тот сценарий, для которого предназначена библиотека QUO! Соответствующие шаги таковы:

1. инициализировать QUO;
2. назначить привязки процессов ядрам для участка MPI;
3. развернуть привязки по всему узлу для участка OpenMP;
4. вернуться к настройкам MPI.

```
Привязка по умолчанию для процессов MPI

Привет из процесса 93096, ранг 0, на cn630. (ядерная аффинность = 0-21,44-65)
Привет из процесса 93093, ранг 1, на cn630. (ядерная аффинность = 22-43,66-87)
  < ... пропускаем результат ... >
Привет из процесса 93162, ранг 42, на cn630. (ядерная аффинность = 0-21,44-65)
Привет из процесса 93159, ранг 43, на cn630. (ядерная аффинность = 22-43,66-87)

Процессы должны закрепляться за ядрами hw

Привет из процесса 93096, ранг 0, на cn630. (ядерная аффинность = 0,44)
Привет из процесса 93093, ранг 1, на cn630. (ядерная аффинность = 1,45)
  < ... пропускаем результат ... >
Привет из процесса 93162, ранг 42, на cn630. (ядерная аффинность = 42,86)
Привет из процесса 93159, ранг 43, на cn630. (ядерная аффинность = 43,87)
```

Разъем

Ядро

Рис. 14.17 Результат работы демонстрационной программы autobind показывает, что ядра изначально привязаны к разъемам, но впоследствии они привязываются к аппаратным ядрам

В следующем ниже листинге давайте посмотрим, как это делается с помощью QUO.

Листинг 14.7 Демонстрация динамического переключения аффинности с MPI на OpenMP

Quo/dynaffinity.c

```
45 int main(int argc, char **argv)
46 {
47    int rank, noderank, nnoderanks;
48    int work_member = 0, max_members_per_res = 44;
49    QUO_context qcontext;
50
51    MPI_Init(&argc, &argv);
52    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
53    QUO_create(&qcontext, MPI_COMM_WORLD);
54
55    node_info_report(qcontext, &noderank, &nnoderanks);
56
57    SyncIt();
58    QUO_bind_push(qcontext,
                    QUO_BIND_PUSH_PROVIDED,
59                  QUO_OBJ_CORE, noderank);
60    SyncIt();
61
62    QUO_auto_distrib(qcontext, QUO_OBJ_SOCKET,
                       max_members_per_res,
63                     &work_member);
64
65    place_report_mpi_quo(qcontext);
66
67    /* изменить политики привязывания, чтобы уместить потоки OMP на узле 0 */
68    bool on_rank_0s_node = rank < nnoderanks;
69    if (on_rank_0s_node) {
70       if (rank == 0) {
71          printf("\nВходим в участок OMP...\n\n");
72          // расширяет cpuset
            // на все имеющиеся ресурсы на узле.
73          QUO_bind_push(qcontext,
                          QUO_BIND_PUSH_OBJ,
                          QUO_OBJ_SOCKET, -1);
74          report_bindings(qcontext, rank);
75          /* выполнить расчет OpenMP */
76          place_report_mpi_omp();
77          /* вернуться к старой политике привязывания */
78          QUO_bind_pop(qcontext);
79       }
80       /* QUO_barrier, потому что он дешевле, чем
            MPI_Barrier на узле. */
81       QUO_barrier(qcontext);
82    }
```

Инициализирует контекст QUO

Назначает аффинность аппаратным ядрам

Распределяет и привязывает ранги MPI

Сообщает об аффинности процессов для всех участков MPI

Сообщает о масках CPU для участка OpenMP

Назначает аффинность всей системе

Выталкивает привязки и возвращается к привязкам MPI

Сообщает об аффинности процессов для участка OpenMP

```
83    SyncIt();
84
85    // Завершение
86    QUO_free(qcontext);
87    MPI_Finalize();
88    return(0);
89 }
```

Приложение dynaffinity можно выполнить с числом аппаратных ядер в нашей системе с помощью следующих ниже команд:

```
make dynaffinity
mpirun -n 44 ./dynaffinity
```

Мы снова используем наши процедуры отчетности для проверки привязок процессов для участка MPI и OpenMP. На рис. 14.18 показана распечатка.

Распечатка на рис. 14.18 показывают, что привязки процессов изменились между участками MPI и OpenMP, что привело к динамическому изменению аффинности во время выполнения.

```
Nodeinfo: nnodes 1 nnoderanks 44 nsockets 2 ncores 44 nhwthreads 88

Привет из процесса 96779, ранг 0, на cn630. (ядерная аффинность = 0,44) cbind [0x00001000,0x00000001]
Привет из процесса 96781, ранг 1, на cn630. (ядерная аффинность = 1,45) cbind [0x00002000,0x00000002]
  <... пропускаем результат ...>
Привет из процесса 96851, ранг 42, на cn630. (ядерная аффинность = 42,86) cbind [0x00400000,0x00000400,0x0]
Привет из процесса 96849, ранг 43, на cn630. (ядерная аффинность = 43,87) cbind [0x00800000,0x00000800,0x0]
Входим в участок OMP...

cpuset ранга 0: 0x00ffffff,0xffffffff,0xffffffff
Выполнение с 44 потоком(потоками)
proc_bind равна false
proc_num_places равно 0
Привет из ранга 00, поток 00, на cn625. (ядерная аффинность = 0-87) Разъем OpenMP: -1
Привет из ранга 00, поток 01, на cn625. (ядерная аффинность = 0-87) Разъем OpenMP: -1
  <... пропускаем результат ...>
Привет из ранга 00, поток 42, на cn625. (ядерная аффинность = 0-87) Разъем OpenMP: -1
Привет из ранга 00, поток 43, на cn625. (ядерная аффинность = 0-87) Разъем OpenMP: -1
```


Рис. 14.18 Для участка MPI процессы привязаны к аппаратным ядрам. Когда мы входим в участок OpenMP, аффинности расширяются на весь узел целиком

14.8 Материалы для дальнейшего исследования

Функциональность манипулирования размещением и привязками процессов является относительно новой. Следите за презентациями в сообществах MPI и OpenMP для получения дополнительных разработок в этой области. В следующем ниже разделе мы перечислим несколько наиболее актуальных материалов по аффинности, которые рекомендуем для дополнительного чтения. После материалов по дополнительному

чтению приведем несколько упражнений, чтобы продолжить разведку этой темы.

14.8.1 Дополнительное чтение

Программы отчетности о размещении процессов, используемые в этой главе для OpenMP, MPI и MPI плюс OpenMP, модифицированы на основе программы xthi.c, используемой для обучения на нескольких веб-сайтах HPC. Вот ссылки на статьи и презентации, в которых она используется для разведывательного анализа аффинности:

- Й. Хе, Б. Кук и соавт., «Подготовка пользователей NERSC к Cori, системе Cray XC40 с многочисленными интегрированными ядрами Intel» (Y. He, B. Cook, et al., Preparing NERSC users for Cori, a Cray XC40 system with Intel many integrated cores In *Concurrency Computat: Pract Exper.*, 2018; 30:e4291, https://doi.org/10.1002/cpe.4291);
- Аргоннская национальная лаборатория, «Аффинность на Тета» (Argonne National Laboratory, Affinity on Theta) по адресу https://www.alcf.anl.gov/support-center/theta/affinity-theta;
- Национальный научно-вычислительный центр энергетических исследований (NERSC), «Аффинность процессов и потоков» (National Energy Research Scientific Computing Center (NERSC), Process and Thread Affinity) по адресу https://docs.nersc.gov/jobs/affinity/.

Вот хорошая презентация по OpenMP, которая включает в себя обсуждение аффинности и того, как ей манипулировать:

- Т. Мэттсон и Х. Хе, «OpenMP: за пределами обычного ядра» (T. Mattson and H. He, OpenMP: Beyond the common core) по адресу http://mng.bz/ aK47.

Мы рассмотрели только часть опций команды mpirun в OpenMPI. Для разведывательного анализа дополнительных способностей обратитесь к справочной странице OpenMPI:

- https://www.open-mpi.org/doc/v4.0/man1/mpirun.1.php.

Локальность переносимого оборудования (hwloc) является подпроектом проекта Open MPI. Это автономный пакет, который одинаково хорошо работает как с OpenMPI, так и с MPICH и стал универсальным аппаратным интерфейсом для большинства имплементаций MPI и многих других программных приложений параллельного программирования. Для получения дополнительной информации обратитесь к следующим ссылкам.

- Главная страница проекта hwloc https://www.open-mpi.org/projects/hwloc/, где вы также найдете несколько ключевых презентаций.
- Б. Гоглин, «Понимание и управление аппартной аффинностью с локальностью оборудования (hwlooc)» (B. Goglin, Understanding and managing hardware affinities with Hardware Locality (hwlooc), *High Performance and Embedded Architecture and Compilation*, HiPEAC, 2013) по адресу http://mng.bz/gxYV.

Набор инструментов «будто я знаю, что делаю» (likwid от англ. *Like I Knew What I'm Doing*) хорошо известен своей простотой, удобством использования и хорошей документацией. Вот хорошая отправная точка для дальнейшего исследования этих инструментов.

- Комплект инструментов Университета Эрлангена-Нюрнберга для мониторинга и сравнительного анализа производительности (University of Erlangen-Nuremberg's performance monitoring and benchmarking suite), https://github.com/RRZE-HPC/likwid/wiki.

Приведенная ниже презентация о библиотеке QUO, которая демонстрировалась в рамках конференции, дает более полный обзор и излагает лежащую в ее основе философию:

- С. Гутьеррес и соавт., «Размещение неоднородности уровня потоков в состыкованных параллельных приложениях» (S. Gutiérrez et al., Accommodating Thread-Level Heterogeneity in Coupled Parallel Applications), https://github.com/lanl/libquo/blob/master/docs/slides/gutierrez-ipdps17.pdf, 2017 International Parallel and Distributed Processing Symposium, IPDPS17).

14.8.2 Упражнения

1. Создайте визуальное изображение пары разных аппаратных архитектур. Раскройте характеристики оборудования для этих устройств.
2. На вашем оборудовании выполните комплект тестов, используя скрипт, приведенный в листинге 14.1. Что вы узнали о том, как использовать вашу систему наилучшим образом?
3. Измените программу, используемую в примере векторного сложения (vecadd_opt3.c) в разделе 14.3, чтобы включить больше операций с плавающей точкой. Возьмите вычислительное ядро и поменяйте операции в цикле на формулу Пифагора:

```
c[i] = sqrt(a[i]*a[i] + b[i]*b[i]);
```

   Как изменились ваши результаты и выводы о наилучшем размещении и привязках? Видите ли вы сейчас какую-либо выгоду от гиперпотоков (если они у вас есть)?
4. В пример MPI из раздела 14.4 включите вычислительное ядро векторного сложения и сгенерируйте график масштабирования для ядра. Затем измените ядро формулой Пифагора, используемой в упражнении 3.
5. Скомбинируйте векторное сложение и формулу Пифагора в следующей ниже процедуре (в одном цикле либо в двух отдельных циклах), чтобы обеспечить бо́льшую многоразовость использования данных:

```
c[i] = a[i] + b[i];
d[i] = sqrt(a[i]*a[i] + b[i]*b[i]);
```

   Как это меняет результаты исследования размещения и привязывания?

6 Добавьте исходный код, чтобы назначать размещение и аффинность внутри приложения из одного из предыдущих упражнений.

Резюме

- Существуют инструменты, которые показывают размещение вашего процесса. Эти инструменты также могут показывать аффинность для ваших процессов.
- Используйте размещение процессов для параллельных приложений. Это дает полную пропускную способность основной памяти в вашем приложении.
- Выбирайте хорошую упорядоченность процессов для ваших потоков OpenMP либо рангов MPI. Хорошая упорядоченность снижает затраты на обмен данными между процессами.
- Используйте политику привязывания для ваших параллельных процессов. Привязывание каждого процесса удерживает вычислительное ядро от перемещения вашего процесса и потери загруженных в кеш данных.
- Существует возможность изменять аффинность внутри вашего приложения. Указанное изменение может охватывать разделы кода, которые работают лучше с разными аффинностями процессов.

15 Пакетные планировщики: наведение порядка в хаосе

Эта глава охватывает следующие ниже темы:

- роль пакетных планировщиков в высокопроизводительных вычислениях;
- отправку задания в пакетный планировщик;
- связывание отправленных заданий для длительных выполнений или более сложных рабочих потоков.

В большинстве систем высокопроизводительных вычислений используются пакетные планировщики для планирования запуска приложений. Мы дадим вам краткое представление о причинах их использования в первом разделе этой главы. Поскольку планировщики получили широкое распространение в высококлассных системах, вы должны иметь хотя бы базовое представление о них, чтобы выполнять задания в центрах высокопроизводительных вычислений и даже малых кластерах. Мы рассмотрим предназначение и использование пакетных планировщиков. Мы не будем вдаваться в подробности того, как их настраивать и ими управлять (данная тема уже совсем из другой оперы). Настройка и управление – это тема для системных администраторов, а мы всего лишь простые пользователи систем.

Что делать, если у вас нет доступа к системе с пакетным планировщиком? Мы не рекомендуем устанавливать пакетный планировщик

только для того, чтобы опробовать приводимые здесь примеры. Скорее наоборот, радуйтесь тому, что имеете, и держите информацию в данной главе под рукой, когда возникнет необходимость. Если ваш спрос на вычислительные ресурсы растет и вы начинаете использовать более крупный многопользовательский кластер, то вы можете вернуться к этой главе.

Существует масса разных пакетных планировщиков, и каждая инсталляция имеет свои собственные уникальные персональные настройки. Мы обсудим два свободно доступных пакетных планировщика: переносимую пакетную систему (Portable Batch System, PBS) и простую утилиту Linux для управления ресурсами (Simple Linux Utility for Resource Management, Slurm). Существуют варианты каждого из них, включая коммерчески поддерживаемые версии.

Планировщик PBS появился в NASA в 1991 году и был выпущен с открытым исходным кодом под названием OpenPBS в 1998 году. Впоследствии коммерческие версии, PBS Professional от Altair и PBS/TORQUE от Adaptive Computing Enterprises, были разделены на отдельные версии. Свободно доступные версии по-прежнему доступны и широко используются в небольших кластерах. Более крупные веб-сайты высокопроизводительных вычислений, как правило, имеют аналогичные версии, но с контрактом на поддержку.

Планировщик Slurm был создан в Национальной лаборатории Лоуренса Ливермора в 2002 году как простой менеджер ресурсов для кластеров Linux. Позже он был разделен на различные производные версии, такие как версия SchedMD.

Планировщики также могут персонально настраиваться с помощью плагинов или надстроек, которые обеспечивают дополнительную функциональность, поддержку специальных рабочих нагрузок и улучшенные алгоритмы планирования. Вы также найдете ряд строго коммерческих пакетных планировщиков, но их функциональность аналогична тем, которые представлены здесь. Базовые концепции каждой имплементации планировщика во многом одинаковы, и нередко многие детали варьируются от веб-сайта к веб-сайту. Переносимость пакетных скриптов все еще может представлять трудную задачу и требовать некоторой персональной настройки под каждую систему.

15.1 Хаос неуправляемой системы

Вы только что подняли свой новейший кластер для своей группы, и программно-информационное обеспечение находится в рабочем состоянии. Скоро у вас будет дюжина коллег, которые войдут в систему и начнут выполнять задания. Бабах – и у вас уже несколько параллельных заданий на вычислительных узлах, которые сталкиваются друг с другом, замедляя их работу, а иногда приводят к отказу некоторых заданий. В воздухе витает ощутимое напряжение, а терпение на исходе.

По мере увеличения размеров и числа пользователей систем высокопроизводительных вычислений возникает необходимость в некотором управлении системой, чтобы навести порядок в хаосе и получить максимальную производительность от оборудования. Инсталляция пакетного планировщика может сэкономить время (рис. 15.1). Могут выполняться пользовательские задания, и эксклюзивное использование оборудования в качестве ресурса становится реальностью. Однако использование пакетной системы не является панацеей. В то время как этот тип программно-информационного обеспечения предлагает многое пользователям кластера или системы высокопроизводительных вычислений, пакетные планировщики требуют значительного времени на администрирование системы и создание разных очередей и политик. Имея хорошую политику, вы можете получать приватно выделяемые вычислительные узлы для вашего эксклюзивного использования в течение фиксированного периода времени.

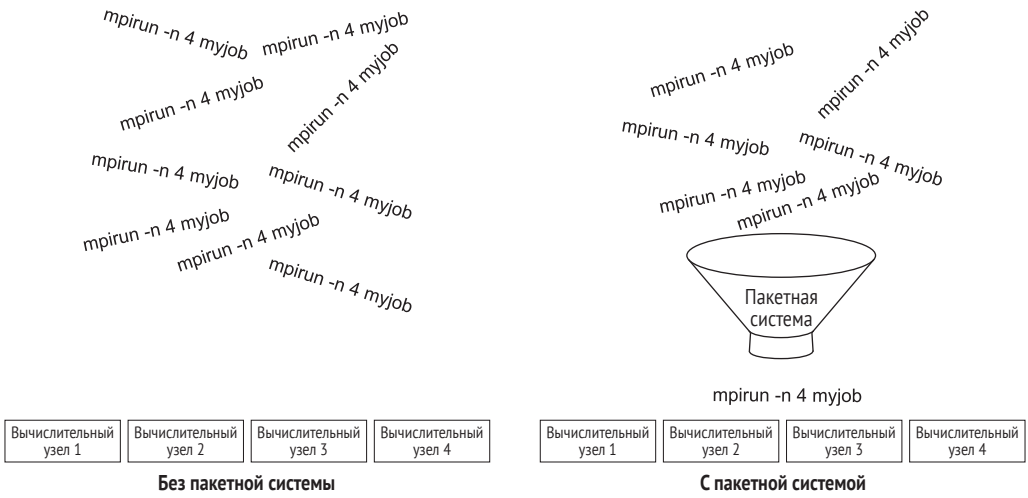


Рис. 15.1 Пакетные системы по отношению к компьютерному кластеру похожи на систему массового обслуживания расчетов за покупки в супермаркете. Они помогают лучше использовать ресурсы и повышать эффективность вашей работы

Порядок, обеспечиваемый программно-информационным обеспечением для управления системой, абсолютно необходим для достижения производительности ваших параллельных приложений. Историческая работа над пакетными планировщиками в кластерах Beowulf (упомянутая в разделе 15.6.1) дает хорошее представление о важности планировщиков. В конце 1990-х годов кластеры Beowulf возникли как широко распространенное движение за строительство вычислительных кластеров из обычных компьютеров. Сообщество Beowulf вскоре осознало, что недостаточно иметь коллекцию вычислительного оборудования; необходимо также иметь некоторый контроль и управление программно-информационным обеспечением, чтобы сделать его производительным ресурсом.

15.2 Как не быть помехой при работе в занятом работой кластере

В занятых работой кластерах много пользователей и много работы. Пакетная система часто имплементируется для управления рабочей нагрузкой и получения максимальной отдачи от системы. Указанные кластеры представляют собой иную среду, чем автономная однопользовательская рабочая станция. Во время работы с этими занятыми кластерами важно знать, как использовать систему эффективно, проявляя при этом внимание к другим пользователям. Мы дадим вам несколько установленных и неустановленных социальных правил, чтобы не стать изгоем в занятом работой кластере. Но сначала давайте посмотрим на то, как эти типовые системы устроены.

15.2.1 Макет пакетной системы для занятых кластеров

В большинстве кластеров некоторые узлы отложены в сторону в качестве фронтендов. Эти фронтендовые узлы также называются *узлами входа*, потому что именно там вы будете находиться при входе в систему. Тогда остальная часть системы настраивается как бэкендовые узлы, которые управляются и выделяются пакетной системой. Эти бэкендовые узлы организованы в одну или несколько очередей. Каждая очередь имеет набор политик для таких вещей, как размер заданий (например, число процессоров или память) и продолжительность выполнения этих заданий.

15.2.2 Как быть вежливым на занятых работой кластерах и сайтах HPC: распространенные любимые мозоли HPC

Для интерактивной работы:

- проверьте нагрузку на ваш фронтенд с помощью команды top и перейдите к слегка загруженному фронтендовому узлу. Обычно существует несколько фронтендовых узлов с такими номерами, как fe01, fe02 и fe03;
- следите за заданиями по передаче тяжелых файлов во фронтенде. Для указанных типов заданий на некоторых веб-сайтах есть специальные очереди. Если вы получаете узел с интенсивным использованием файлов, то вы, возможно, обнаружите, что ваши компиляции или другие задания занимают гораздо больше времени, чем обычно, даже если нагрузка не выглядит высокой;
- некоторые веб-сайты хотят, чтобы вы компилировали на бэкенде, а другие – на фронтенде. Проверьте политики в своем кластере;
- не связывайте узлы пакетными интерактивными сеансами, а затем можете отправляться на многочасовое собрание;
- вместо того чтобы получать второй пакетный интерактивный сеанс, экспортируйте X-терминал или оболочку из вашего первого сеанса;

- в целях легкой работы ищите очереди для совместного использования, которые допускают избыточную подписку;
- на многих веб-сайтах есть специальные очереди для отлаживания. Используйте их тогда, когда вам нужно выполнять отладку, но не злоупотребляйте этими очередями отлаживания.

Для больших работ:

- большие параллельные задания должны выполняться на бэкендовых узлах через очереди пакетной системы;
- поддерживайте число заданий в очереди малым: не монополизируйте очереди;
- старайтесь выполнять свои большие задания в нерабочее время, чтобы другие пользователи имели возможность получать интерактивные узлы для своей работы.

Для хранения;

- храните крупные файлы в надлежащем месте. Большинство веб-сайтов имеет крупные параллельные файловые системы, каталоги блокнотов, проектов или рабочих файлов для вывода результатов вычислений;
- перемещайте файлы в долгосрочное хранилище вашего веб-сайта;
- знайте политику очистки хранилища файлов. Крупные веб-сайты будут периодически удалять файлы в некоторых блокнотных каталогах;
- регулярно очищайте свои файлы и поддерживайте заполненность файловых систем ниже 90 %. Производительность файловой системы снижается по мере заполнения файловых систем.

ПРИМЕЧАНИЕ Не бойтесь отправлять приватные сообщения пользователям, которые создают проблемы, но будьте вежливы. Они, возможно, не осознают, что их работа приводит к остановке многих рабочих процессов.

Дальнейшая кластерная мудрость состоит в следующем:

- интенсивное использование фронтендовых узлов может приводить к нестабильности и отказам. Эти нестабильности влияют на всю систему, поскольку больше невозможно планировать задания для бэкендовых узлов;
- нередко проекты получают ресурсные выделения, которые используются для приоритизации заданий с использованием алгоритма планирования под названием «объективное распределение ресурсов». В этих случаях вам, возможно, потребуется подать заявку на ресурсы, необходимые для вашего проекта;
- каждый веб-сайт может устанавливать политики, имплементирующие правила, но они не охватывают все ситуации. Вы должны следовать духу правил, а также фактической имплементации. Другими словами, не пытайтесь обходить правила системы. Это не неодушевленный объект, а скорее наоборот, это ваши коллеги-пользователи. Они тоже пытаются выполнять свою работу;

- вместо того чтобы пытаться обходить правила системы, вам следует оптимизировать свой исходный код и хранилище файлов. Экономия позволит вам выполнять больше работы и другим тоже выполнять свою работу в кластере;
- неразумно отправлять несколько сотен заданий в очередь, когда за один раз может выполняться только по нескольку заданий. Как правило, мы отправляем за один раз не более десяти заданий или около того, а затем отправляем дополнительные задания, когда каждое из них завершается. Существует масса способов делать это с помощью скриптов оболочки или даже технических приемов пакетной зависимости (обсуждаемых далее в этой главе);
- для заданий, выполнение которых требует времени, значительно превышающего максимально допустимое время выполнения пакета, следует имплементировать фиксацию состояния в контрольных точках (раздел 15.4). Фиксация состояния в контрольных точках улавливает сигналы пакетной терминации или использует тактовые хронометражи, чтобы наиболее эффективно использовать все время пакета. Последующее задание начинается с того места, где остановилось последнее.

15.3 Отправка вашего первого пакетного скрипта

В этом разделе мы пройдемся по процессу отправки вашего первого пакетного скрипта. Пакетные системы требуют другого образа мыслей. Вместо того чтобы просто начинать работу в любое удобное для вас время, нужно подумать об организации своей работы. Планирование приводит к более эффективному использованию ресурсов еще до того, как ваши задания будут отправлены. Как использовать эти пакетные системы? Как показано на рис. 15.2, существует два базовых системных режима.

Большинство команд, используемых в одном режиме, также могут использоваться в другом. Давайте пробежимся по нескольким примерам, чтобы увидеть принцип работы этих режимов. В этом первом наборе примеров мы будем работать с пакетным планировщиком Slurm. Мы начнем с интерактивного примера и выполним модификацию примера в форму пакетного файла.

Интерактивный консольный режим (командной строки) обычно используется для разработки программ, тестирования или выполнения коротких заданий. Для отправки более длительных производственных заданий чаще используется пакетный файл, в котором отправляется пакетное задание. Пакетный файл позволяет пользователю выполнять приложения в ночное время или без присмотра. Пакетные скрипты можно даже писать для автоматического перезапуска заданий в случае какого-либо катастрофического системного события. Мы покажем перевод

в синтаксической форме из опции командной строки в пакетный скрипт. Но сначала нам нужно разобрать базовую структуру пакетного скрипта.

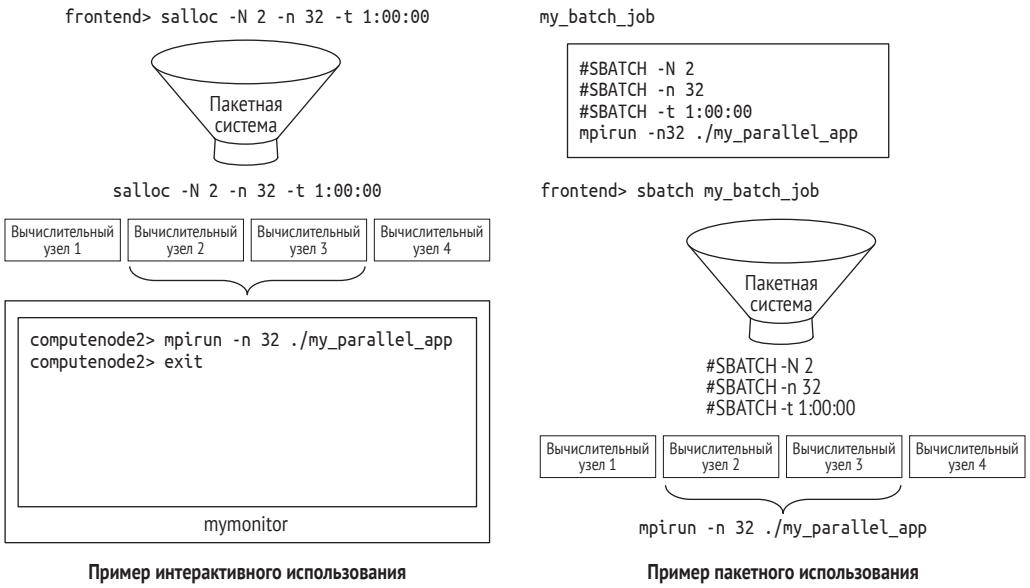


Рис. 15.2 Пакетные системы обычно используются в интерактивном режиме либо в модели пакетного использования

Пример: интерактивная командная строка

Давайте начнем с кластерного фронтенда, где все входят в систему. Теперь нам нужны два вычислительных узла (`-N 2`) с общим числом процессоров 32 (`-n 32`) в течение часа (`-t 1:00:00`). Обратите внимание на разницу в прописных буквах для числа узлов (*N*) и числа процессов (*n*). Кроме того, вы можете ограничивать время выполнения заданными минутами, хотя во многих системах действует политика минимального и максимального времени выполнения. В некоторых системах даже есть минимальные и максимальные значения числа используемых вычислительных узлов. Команда salloc для этого конкретного запроса будет выглядеть так:

```
frontend> salloc -N 2 -n 32 -t 1:00:00
```

Команда `salloc` выделяет и входит в два вычислительных узла. Обратите внимание, что следующее приглашение командной строки изменяется, указывая, что мы находимся в другой системе. Конкретное приглашение сильно зависит от настроек вашей системы и среды. Получив два узла, мы сможем запустить наше параллельное приложение с помощью команды:

```
computenode22> mpirun -n 32 ./my_parallel_app
```

В приведенном выше примере показан запуск параллельного задания с помощью команды `mpirun`. Как упоминалось в разделе 8.1.3, команда для за-

пуска параллельного задания в вашей системе может отличаться. Когда мы закончим, мы просто выйдем:

```
computenode22> exit
```

Пример: синтаксис пакетного файла

Атрибуты задания задаются с использованием следующего ниже синтаксиса:

```
#SBATCH <опция>
```

Файл также содержит исполняемые команды, такие как команда `mpirun`:

```
mpirun -n 32 ./my_parallel_app
```

Затем пакетный файл отправляется с помощью команды `sbatch`:

```
sbatch < my_batch_job or sbatch my_batch_job
```

Вы можете указывать опции в интерактивной командной строке либо с помощью ключевого слова SBATCH. Существует опция длинной формы и синтаксис короткой формы. Например, `time=1:00:00` – это длинная форма, а `-t=1:00:00` – короткая.

В табл. 15.1 показано несколько наиболее распространенных опций Slurm.

Таблица 15.1 Командные опции Slurm

Опция	Функция	Пример
`[--time|-t]=hr:min:sec`	Запрашивает максимальное время выполнения	`-t=8:00:00`
`[--nodes|-N]=#`	Запрашивает число узлов	`--nodes=2`
`[--ntasks|-n]=nprocs`	Запрашивает число процессоров	`-n=8`
`[--job-name|-J]=name`	Дает вашему заданию имя	`-J=job22`
`[--output|-o]=filename`	Записывает стандартный вывод в файл с указанным именем	`-o=run.out`
`[--error|-e]=filename`	Записывает вывод ошибок в файл с указанным именем	`-e=run.err`
`[--exclusive]`	Задает эксклюзивное использование узлов	`--exclusive`
`[--oversubscribe|-s]`	Указывает ресурсы избыточной подписки	`--oversubscribe`

Давайте продолжим и соберем все это в наш первый полный пакетный скрипт Slurm, как показано в следующем ниже листинге. Этот пример включен в соответствующий исходный код книги по адресу https://github.com/EssentialsofParallelComputing/Chapter15. Как всегда, мы призываем вас сверяться с приведенными в этой главе примерами.

Листинг 15.1 Пакетный скрипт Slurm для параллельного задания

```
1 #!/bin/sh
2 #SBATCH -N 1   <-- Задает один вычислительный узел
3 #SBATCH -n 4   <-- Указывает четыре процессора
```

```
 5 #SBATCH -t 01:00:00  <-------- Выполняет задание один час
 6
 7 # Не размещайте команды bash перед последней директивой SBATCH
 8 # Поведение может быть ненадежным
 9
10 mpirun -n 4 ./testapp &> run.out
```

Вместо `-N` в строке 2, как вариант, можно указывать с помощью `--nodes`. Значение `-N` имеет другой смысл в других пакетных планировщиках и имплементациях MPI, что приводит к неправильным значениям и ошибкам. В ряде используемых вами пакетных систем и MPI вы должны внимательно отслеживать несогласованности в синтаксисе. Затем отправляем это задание с помощью команды `sbatch < first_slurm_batch_job`. Мы получаем эквивалент пакетного задания в интерактивном задании с помощью команд:

```
frontend> salloc -N 1 -n 4 -t 01:00:00
computenode22> mpirun -n 4 ./testapp
computenode22> exit
```

ПРИМЕЧАНИЕ Опции являются одинаковыми как в пакетном файле, так и в командной строке.

Мы должны особо упомянуть об опциях `exclusive` и `oversubscribe`. Одной из основных причин использования пакетной системы является эксклюзивное использование ресурса в целях повышения эффективности работы приложений. Почти каждый крупный вычислительный центр устанавливает поведение по умолчанию для эксклюзивного использования ресурса. Но для конкретных случаев использования конфигурация системы может настраивать совместный доступ к одному разделу. Вы можете использовать указанные командные опции, `exclusive` и `oversubscribe`, для команд `sbatch` и `srun`, чтобы запрашивать поведение, отличное от конфигурации системы. Однако вы не можете переопределять совместную конфигурацию для раздела.

Большинство крупных вычислительных систем состоит из большого числа узлов с идентичными характеристиками. Однако все чаще встречаются системы с различными типами узлов. Slurm предлагает команды, которые могут запрашивать узлы с особыми характеристиками. Например, вы можете использовать `--mem=<#>` для получения узлов крупной памяти с требуемым размером в Мб. С помощью пакетной системы можно выполнять много других специальных запросов. Пакетный скрипт для пакетного планировщика PBS аналогичен, но с другим синтаксисом. Несколько наиболее распространенных вариантов PBS показано в табл.15.2.

Опция `-l` представляет исчерпывающий вариант, который используется для самых разных опций. Давайте соберем эквивалентный пакетный скрипт PBS для того же задания, что и в листинге 15.1. В следующем ниже листинге показан скрипт PBS.

Таблица 15.2 Командные опции PBS

Опция	Команда	Пример
`-l [nodes|walltime|cput|mem |ncpus|ppn|procs]`	Исчерпывающий вариант для параллельных потребностей	`-l nodes=2,procs=4`
`-N <name>`	Дает заданию имя	`-N job22`
`-o <filename>`	Записывает стандартный вывод в файл с указанным именем	`-o run.out`
`-e <filename>`	Записывает вывод ошибок в файл с указанным именем	`-e run.err`
`-q <queue>`	Очереди для отправки заданий (имена очередей зависят от конкретного веб-сайта)	`-q standard`
`-j`	Соединяет стандартный вывод и вывод ошибки	`-j -o run.out`

Листинг 15.2 Пакетный скрипт PBS для параллельного задания

```
 1 #!/bin/sh
 2 #PBS -l nodes=1
 3 #PBS -l procs=4                    Ключевые слова и синтакс PBS
 5 #PBS -l walltime=01:00:00
 6
 7 # Не размещайте команды bash перед последней директивой PBS
 8 # Поведение может быть ненадежным
 9
10 mpirun -n 4 ./testapp &> run.out
```

В случае PBS мы отправляем задание с помощью команды `qsub < first_pbs_batch_job`. В целях получения интерактивного выделения в PBS мы используем опцию `-I` для `qsub`:

```
frontend> qsub -I -l nodes=1,procs=4,walltime=01:00:00
computenode22> mpirun -n 4 ./testapp &> run.out
computenode22> exit
```

Для этих примеров может потребоваться указывать очередь или другую информацию, относящуюся к конкретному веб-сайту. На многих веб-сайтах существуют разные очереди для длинных, коротких, крупных и других специализированных ситуаций. Для получения этих важных сведений обратитесь к документации локального веб-сайта.

В приведенном выше обсуждении мы увидели пару команд пакетного планировщика. Для эффективного использования системы вам понадобится больше команд, которые приведены ниже. Эти команды пакетного планировщика проверяют статус вашего задания, получают информацию о системных ресурсах и отменяют задания. Далее мы приводим резюме наиболее распространенных команд как для планировщика Slurm, так и для планировщика PBS.

Справочное руководство по Slurm

- `salloc [--nodes|-N]=N [-ntasks|-n]=N` выделяет узлы для пакетного задания:

```
frontend> salloc -N 1 -n 32
```

- `sbatch` отправляет пакетное задание в пакетный планировщик (см. примеры, приведенные ранее в этом разделе).
- `scancel` отменяет задание, которое выполняется или ожидает в очереди:

```
frontend> scancel <SLURM_JOB_ID>
```

- `sinfo` предоставляет информацию о статусе системы, контролируемой пакетным планировщиком:

```
frontend> sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
standard*    up    4:00:00      1  drain n02
standard*    up    4:00:00      5   resv n[03-07]
standard*    up    4:00:00      2  alloc n[08-09]
standard*    up    4:00:00      1   idle n01
debug        up    1:00:00      1   idle n10
```

- `squeue` показывает задания в очереди и их статус (например, выполняющиеся или ожидающие выполнения). Наиболее часто используемой является `squeue -u <имя пользователя>` только для ваших собственных пользовательских заданий или `squeue` для всех заданий.

```
frontend> squeue
JOBID PARTITION    NAME    USER  ST     TIME  NODES NODELIST(REASON)
35456  standard   sim_2     jrr  PD     0:00      1 (Resources)
35455  standard   sim_1     jrr   R  2:26:54      1 n08
```

- `srun [--nodes|-N]=N [--ntasks|-n]=N <exec>`, замена прогона mpi, содержит дополнительные возможности для размещения и привязывания. Если плагин affinity включен, то доступны следующие ниже дополнительные опции:

```
--sockets-per-node=S
--cores-per-socket=C
--threads-per-core=T
--ntasks-per-core=n
--ntasks-per-socket=n
--cpu-bind=[threads|cores|sockets]
--exclusive
--share
```

  Например:

```
frontend> srun -N 1 -n 16 --cpu-bind=cores my_exec
```

- `scontrol` просматривает или модифицирует компоненты Slurm:

```
frontend> scontrol show job <SLURM_JOB_ID>
JobID=35456 JobName=sim2
  UserID=jrr <...и многое другое...>
```

В следующей ниже таблице перечислены несколько средовых переменных Slurm, которые могут быть полезны в ваших пакетных скриптах.

Средовая переменная Slurm	Функция
SLURM_NTASKS	Суммарное число запрошенных процессоров
SLURM_CPUS_ON_NODE	CPU-процессоры на узле
SLURM_JOB_CPUS_PER_NODE	Запрошенные CPU-процессоры для каждого узла
SLURM_JOB_ID	ИД задания
SLURM_JOB_NODELIST	Список узлов, выделенных для задания
SLURM_JOB_NUM_NODES	Число узлов для задания
SLURM_SUBMIT_DIR	Каталог, из которого было отправлено задание
SLURM_TASKS_PER_NODE	Число операционных задач, которые должны быть запущены на каждом узле

Справочное руководство по PBS

- qsub отправляет пакетное задание, где:
  - `-I` – это интерактивное задание;
  - пакетным эквивалентом этой команды является `#PBS interactive=true`;
  - `-W block=true` ожидает завершения задания;
  - пакетный эквивалент равен `#PBS block=true`;
  - `qdel` удаляет пакетное задание: `frontend> qdel <job ID>`;
  - `qsig` посылает сигнал пакетному заданию: `frontend> qsig 23 56`;
  - `qstat` показывает статус пакетных заданий:

```
frontend> qstat or qstat -u jrr
                                                  Req'd  Elap
JobID    User   Queue Jobname Sess NDS  TSK Mem Time S Time
-------- ------ ----- ------- ---- ---- --- --- ---- - ----
56.base  jrr    standard sim2 --   --     l  -- 0:30 R 0:02
```

- qmsg отправляет сообщение в пакетное задание:

```
frontend> qmsg "message to standard error" 56
```

В следующей ниже таблице перечислено несколько средовых переменных PBS, которые могут быть полезны в ваших пакетных скриптах.

Средовая переменная PBS	Функция
PBS_JOBDIR	Каталог исполнения задания
PBS_TMPDIR	Временный каталог или блокнотное пространство
PBS_O_WORKDIR	Текущий рабочий каталог, в котором была исполнена команда qsub
PBS_JOBID	ИД задания

15.4 Автоматические перезапуски для длительных заданий

Большинство веб-сайтов высокопроизводительных вычислений лимитирует максимальное время выполнения задания. Тогда как выполнять более длительные задания? Типичный подход заключается в том, что приложения периодически записывают свое состояние в файлы, после чего отправляется последующее задание, которое считывает файл и запускается в этой точке выполнения. Указанный процесс, как продемонстрировано на рис. 15.3, называется *фиксацией состояния в контрольной точке* и *перезапуском*.

Рис. 15.3 Файл контрольной точки, хранящий состояние вычисления, записывается на диск по завершении пакетного задания, а затем следующее пакетное задание считывает файл и перезапускает вычисление с того места, в котором предыдущее задание остановилось

Процесс фиксации состояния в контрольной точке полезен для работы с лимитированным временем пакетного задания и для обработки отказов системы или других прерываний работы. В небольшом числе случаев вы можете перезапускать свои задания вручную, но по мере увеличения числа перезапусков это становится реальным бременем. Если это так, то вам следует добавлять возможность автоматизации процесса. Для этого требуется немало усилий, изменения в вашем приложении и более сложные пакетные скрипты. Мы покажем скелетное приложение, в котором мы это сделали.

Прежде всего пакетный скрипт должен сигнализировать вашему приложению о том, что оно подходит к концу отведенного ему времени. Затем скрипт должен переотправлять себя рекурсивно до тех пор, пока ваше задание не будет завершено. В следующем ниже листинге показан такой скрипт для Slurm.

Листинг 15.3 Пакетный скрипт для автоматического перезапуска

`AutomaticRestarts/batch_restart.sh`

```
1 #!/bin/sh
    < ... примечания по использованию ... >
```

```
#SBATCH -N 1
#SBATCH -n 4
#SBATCH --signal=23@160
#SBATCH -t 00:08:00

# Не размещайте команды bash перед последней директивой SBATCH
# Поведение может быть ненадежным

NUM_CPUS=${SLURM_NTASKS}
OUTPUT_FILE=run.out
EXEC_NAME=./testapp
MAX_RESTARTS=4

if [ -z ${COUNT} ]; then
   export COUNT=0
fi

((COUNT++))
echo "Счетчик COUNT перезапусков равен ${COUNT}"

if [ ! -e DONE ]; then
   if [ -e RESTART ]; then
      echo "=== Перезапускаем ${EXEC_NAME} ===" \
            >> ${OUTPUT_FILE}
      cycle=`cat RESTART`
      rm -f RESTART
   else
      echo "=== Запускаем задачу ===" \
            >> ${OUTPUT_FILE}
      cycle=""
   fi

   mpirun -n ${NUM_CPUS} ${EXEC_NAME} \
          ${cycle} &>> ${OUTPUT_FILE}
   STATUS=$?
   echo "Закончена mpirun" \
            >> ${OUTPUT_FILE}

   if [ ${COUNT} -ge ${MAX_RESTARTS} ]; then
      echo "=== Достигнуто максимальное число перезапусков ===" \
            >> ${OUTPUT_FILE}
      date > DONE
   fi

   if [ ${STATUS} = "0" -a ! -e DONE ]; then
      echo "=== Отправляем скрипт перезапуска ===" \
            >> ${OUTPUT_FILE}
      sbatch <batch_restart.sh
   fi
fi
```

Посылает приложению сигнал 23 (SIGURG) за 160 с до терминации

Максимальное число отправленных скриптов

Подсчитывает число отправлений

Проверяет наличие файла DONE

Проверяет наличие файла RESTART

Возвращает номер итерации для командной строки

Вызывает задание MPI с аргументами командной строки

Завершает работу, если достигнуто максимальное число перезапусков

Отправляет следующее пакетное задание

В этом скрипте много движущихся частей. Во многом это делается для того, чтобы избежать ситуации выхода из-под контроля, когда отправля-

ется больше пакетных заданий, чем необходимо. Скрипт также требует кооперации с приложением. Указанная кооперация включает в себя следующие ниже задачи.

- Пакетная система посылает сигнал, и приложение его улавливает.
- Приложение записывает номер итерации в файл с именем RESTART.
- Приложение записывает файл контрольной точки и считывает его при перезапуске.

Номер сигнала, возможно, будет варьироваться в зависимости от того, что уже используется в пакетной системе и MPI. Мы также предупреждаем вас не ставить команды оболочки перед любыми командами Slurm. Хотя скрипт может выглядеть работающим, мы обнаружили, что сигналы функционировали неправильно; поэтому порядок имеет значение, и вы не всегда получите очевидный отказ. В листинге 15.4 показан скелет исходного кода приложения на языке C, демонстрирующий функциональность автоматического перезапуска.

ПРИМЕЧАНИЕ Примеры исходных кодов по адресу https://github.com/EssentialsofParallelComputing/Chapter15 также содержат пример автоматического перезапуска на языке Fortran.

Листинг 15.4 Образец приложения для тестирования

AutomaticRestarts/testapp.c

```
 1 #include <unistd.h>
 2 #include <time.h>
 3 #include <stdio.h>
 4 #include <stdlib.h>
 5 #include <signal.h>
 6 #include <mpi.h>
 7
 8 static int batch_terminate_signal = 0;
 9 void batch_timeout(int signum){
10    printf("Пакетный таймаут : %d\n",signum);
11    batch_terminate_signal = 1;
12    return;
13 }
14
15 int main(int argc, char *argv[])
16 {
17    MPI_Init(&argc, &argv);
18    char checkpoint_name[50];
19    int mype, itstart = 1;
20    MPI_Comm_rank(MPI_COMM_WORLD, &mype);
21
22    if (argc >=2) itstart = atoi(argv[1]);
          // < ... прочитать файл перезапуска ... >
24
25    if (mype ==0) signal(23, batch_timeout);
26
27    for (int it=itstart; it < 10000; it++){
```

Глобальная переменная для пакетного сигнала

При перезагрузке считывает файл контрольной точки

При перезагрузке считывает файл контрольной точки

Устанавливает функцию обратного вызова для сигнала 23

```
28       sleep(1);
29
30       if ( it%60 == 0 ) {
            // < ... записать файл контрольной точки ... >
40       }
41       int terminate_sig = batch_terminate_signal;
42       MPI_Bcast(&terminate_sig, 1, MPI_INT, 0, MPI_COMM_WORLD);
43       if ( terminate_sig ) {
            // < ... записать RESTART и
            // специальный файл контрольной точки ... >
54          MPI_Finalize();
55          exit(0);
56       }
57
58    }
59
            // < ... записать файл DONE ... >
67    MPI_Finalize();
68    return(0);
69 }
```

Действует в качестве заменителя вычислительной работы

Записывает контрольную точку каждые 60 итераций

Записывает специальный файл контрольной точки и файл с именем RESTART

Записывает файл DONE, когда приложение встречается с критерием завершения

Приведенный выше исходный код может показаться коротким и простым, но в эти строки вложено очень много. Для реального приложения потребуются сотни строк, чтобы полностью имплементировать контрольные точки и перезапуск, критерии завершения и обработку входных данных. Мы также предупреждаем, что разработчикам необходимо тщательно проверять свой исходный код, чтобы предотвращать возникновение условий выхода из-под контроля. Хронометраж сигнала также необходимо настроить на то, сколько времени потребуется, чтобы поймать сигнал, завершить итерации и записать файл перезапуска. В нашем маленьком скелетном образце приложения автоматического перезапуска мы начинаем отправку с команды

```
sbatch < batch_restart.sh
```

и получаем следующий ниже результат:

```
=== Начинаем задачу ===
App launch reported: 2 (out of 2) daemons - 0 (out of 4) procs
 60 Checkpoint: Mon May 11 20:06:08 2020
120 Checkpoint: Mon May 11 20:07:08 2020
180 Checkpoint: Mon May 11 20:08:08 2020
240 Checkpoint: Mon May 11 20:09:08 2020
Пакетный таймаут : 23
297 RESTART: Mon May 11 20:10:05 2020
Закончена mpirun
=== Отправляем скрипт перезапуска ===
=== Перезапускаем ./testapp ===
App launch reported: 2 (out of 2) daemons - 0 (out of 4) procs
300 Checkpoint: Mon May 11 20:10:11 2020
 < ... пропускаем результат ... >
1186 RESTART: Mon May 11 20:25:05 2020
```

```
Закончена mpirun
=== Достигнуто максимальное число перезапусков ===
```

Из распечатки мы видим, что приложение периодически записывает файлы контрольных точек каждые 60 итераций. Поскольку заменителем вычислительной работы на самом деле является команда `sleep` продолжительностью 1 с, контрольные точки находятся на расстоянии 1 мин друг от друга. Примерно через 300 с пакетная система отправляет сигнал, и тестовое приложение сообщает, что он был пойман. В этот момент скрипт записывает файл с именем RESTART, содержащий номер итерации. Затем скрипт выводит сообщение о том, что скрипт перезапуска был отправлен повторно. Результат также показывает повторный запуск приложения. В распечатке мы пропустили показ дополнительных перезапусков и просто показали сообщение о том, что достигнуто максимальное число перезапусков.

15.5 Указание зависимостей в пакетных скриптах

Есть ли в пакетных системах встроенная поддержка последовательностей пакетных заданий? У большинства из них есть функциональность, именуемая зависимостью, которая позволяет указывать то, как одно задание зависит от другого. Используя указанную функциональность, мы можем получать наши последующие задания, отправленные ранее в очередь, отправляя следующее пакетное задание до запуска нашего приложения. Как показано на рис. 15.4, это может давать нам более высокий приоритет для запуска следующего пакетного задания в зависимости от политики веб-сайта. Независимо от этого, ваши задания будут стоять в очереди, и вам не нужно беспокоиться о том, будет или нет отправлено следующее задание.

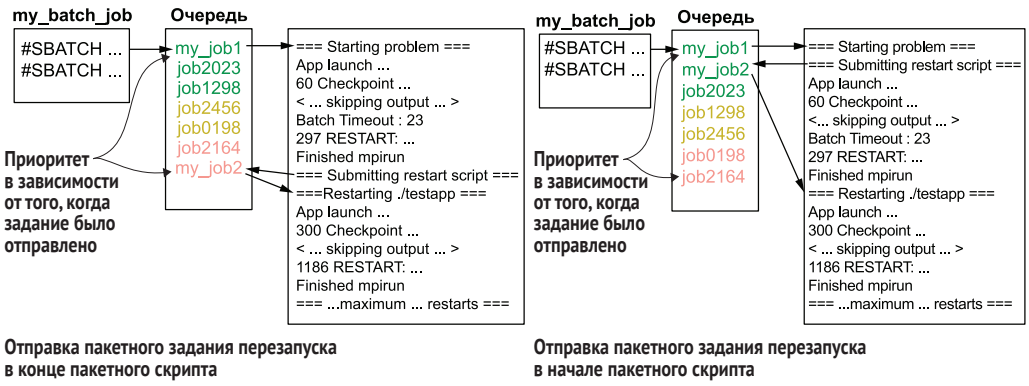


Рис. 15.4 Автоматический перезапуск, отправленный в начале пакетного задания, будет занимать больше времени в очереди, что может придавать вашему заданию перезапуска более высокий приоритет, чем заданию, отправленному в конце пакетного задания (в зависимости от локальных политик планирования)

Мы можем внести это изменение в пакетный скрипт, добавив выражение о зависимости (в строке 33 в следующем ниже листинге). Этот пакетный скрипт отправляется первым до того, как мы начнем нашу работу, но с зависимостью от завершения этого текущего пакетного задания.

Листинг 15.5 Пакетный скрипт приоритетной отправки для перезапуска скрипта

Prestart/batch_restart.sh

```
 1 #!/bin/sh
    < ... примечания по использованию ... >
13 #SBATCH -N 1
14 #SBATCH -n 4
15 #SBATCH --signal=23@160
16 #SBATCH -t 00:08:00
17
18 # Не размещайте команды bash перед последней директивой SBATCH
19 # Поведение может быть ненадежным
20
21 NUM_CPUS=4
22 OUTPUT_FILE=run.out
23 EXEC_NAME=./testapp
24 MAX_RESTARTS=4
25
26 if [ -z ${COUNT} ]; then
27    export COUNT=0
28 fi
29
30 ((COUNT++))
31 echo "Счетчик COUNT перезапусков равен ${COUNT}"
32
33 if [ ! -e DONE ]; then
34    if [ -e RESTART ]; then
35       echo "=== Перезапускаем ${EXEC_NAME} ===" \
             >> ${OUTPUT_FILE}
36       cycle=`cat RESTART`
37       rm -f RESTART
38    else
39       echo "=== Перезапускаем задачу ===" \
             >> ${OUTPUT_FILE}
40       cycle=""
41    fi
42
43    echo "=== Отправляем скрипт перезапуска ===" \
            >> ${OUTPUT_FILE}
44    sbatch --dependency=afterok:${SLURM_JOB_ID} \
             <batch_restart.sh
45
46    mpirun -n ${NUM_CPUS} ${EXEC_NAME} ${cycle} \
            &>> ${OUTPUT_FILE}
47    echo "Закончена mpirun" \
            >> ${OUTPUT_FILE}
```

Сперва отправить это пакетное задание с зависимостью от его завершения

```
48
49    if [ ${COUNT} -ge ${MAX_RESTARTS} ]; then
50       echo "=== Достигнуто максимальное число перезапусков === " \
               >> ${OUTPUT_FILE}
51       date > DONE
52    fi
53 fi
```

В приведенном выше листинге показано, как использовать зависимости в пакетных скриптах для простого случая контрольной точки / перезапуска, но зависимости полезны для многих других ситуаций. Более сложные рабочие потоки могут содержать шаги предобработки, которые необходимо выполнять перед главной работой, а затем выполнять шаг постобработки. Некоторым более сложным рабочим потокам требуется нечто большее, чем зависимость от того, было или нет выполнено предыдущее задание. К счастью, пакетные системы обеспечивают иные типы зависимостей между заданиями. В табл. 15.3 показаны различные возможные опции. PBS имеет аналогичные зависимости для пакетных заданий, которые можно указывать с помощью опции `-W depend=<type:job id>`.

Таблица 15.3 Опции зависимостей для пакетных заданий

Опции зависимостей	Функция
`after`	Задание может начаться после запуска указанного задания(й)
`afterany`	Задание может начаться после завершения указанного задания(й) с любым статусом
`afternotok`	Задание может начаться после отказа указанного задания(й)
`afterok`	Задание может начаться после успешного выполнения указанного задания(й)
`singleton`	Задание может начаться после завершения всех заданий с одинаковым именем и пользователем

15.6 Материалы для дальнейшего исследования

Существуют общие справочные материалы для планировщиков Slurm и PBS, но вам также следует ознакомиться с документацией вашего веб-сайта. Многие веб-сайты настроили конфигурации индивидуально и добавили команды и функциональности для своих конкретных потребностей. Если вы считаете, что вам, возможно, потребуется поднять вычислительный кластер с пакетной системой, то вы, возможно, захотите исследовать новые инициативы, такие как OpenHPC и дистрибутивы Rocks Cluster, которые недавно были выпущены для разных ниш HPC-вычислений.

15.6.1 Дополнительное чтение

Как свободно доступные, так и коммерчески поддерживаемые версии Slurm доступны на веб-сайте SchedMD. Неудивительно, что на указан-

ном веб-сайте SchedMD есть много документации по Slurm. Еще одним хорошим справочным веб-сайтом является Национальная лаборатория Лоуренса Ливермора, где Slurm был разработан изначально.

- Документация по SchedMD и Slurm по адресу https://slurm.schedmd.com.
- Блейз Барни, «Slurm и Moab» (Blaise Barney, Slurm and Moab, Lawrence Livermore National Laboratory) по адресу https://computing.llnl.gov/tutorials/moab/.

Лучшая информация о PBS представлена в Руководстве пользователя PBS:

- Альтаир Инжиниринг, Руководство пользователя PBS по адресу https://www.altair.com/pdfs/pbsworks/PBSUserGuide2021.1.pdf.

Хотя и несколько устаревший, следующий онлайновый справочный материал по поднятию кластера Beowulf представляет собой хорошую историческую перспективу появления кластерных вычислений и того, как настраивать управление кластерами, включая пакетный планировщик PBS:

- Под редакцией Уильяма Гроппа, Юинга Ласка, Томаса Стринга, «Кластерные вычисления в Беовульфа на Linux», 2-е изд. (William Gropp, Ewing Lusk, Thomas String, *Beowulf Cluster Computing with Linux*, 2nd ed., Massachusetts Institute of Technology, 2002, 2003) по адресу http://etutorials.org/Linux+systems/cluster+computing+with+linux/.

Вот несколько веб-сайтов с информацией о современных системах управления программно-информационным обеспечением HPC:

- OpenHPC по адресу http://www.openhpc.community;
- Rocks Cluster по адресу http://www.rocksclusters.org.

15.6.2 Упражнения

1. Попробуйте отправить пару заданий, одно с 32 процессорами и одно с 16 процессорами. Проверьте их отправку и подтвердите, что они работают. Удалите 32-процессорное задание. Проверьте, чтобы убедиться, что оно было удалено.
2. Измените скрипт автоматического перезапуска таким образом, чтобы первое задание было шагом предобработки, который необходимо настроить для вычисления до того, как запуски выполнят симуляцию.
3. Модифицируйте простой пакетный скрипт в листинге 15.1 для Slurm и листинге 15.2 для PBS, чтобы выполнить очистку при отказе, удалив файл с именем simulation_database.

Резюме

- Пакетные планировщики выделяют ресурсы таким образом, чтобы обеспечивать вам возможность эффективно использовать параллель-

ный кластер. Очень важно научиться использовать их для работы в более крупных системах высокопроизводительных вычислений.

- Существует целый ряд команд для опрашивания вашего задания и его статуса. Осведомленность об этих командах позволяет вам эффективнее использовать систему.
- Вы можете использовать автоматические перезапуски и цепочки заданий для запуска более масштабных симуляций и рабочих потоков. Добавление этой способности в ваше приложение позволяет масштабировать задачи, которые в противном случае вы бы не смогли выполнять.
- Зависимости пакетных заданий позволяют управлять сложными рабочими потоками. Используя зависимости между несколькими заданиями, вы можете обрабатывать данные поэтапно, предобрабатывать их для расчета либо запускать задание постобработки.

16 Файловые операции для параллельного мира

Эта глава охватывает следующие ниже темы:

- модифицирование параллельного приложения для стандартных файловых операций;
- запись данных с использованием параллельных файловых операций с помощью MPI-IO и HDF5;
- точную настройку параллельных файловых операций для разных параллельных файловых систем.

Файловые системы создают организованный рабочий поток извлечения, хранения и обновления данных. В любой вычислительной работе продуктом является результат, будь то данные, графика или статистика. Сюда входят не только окончательные, но и промежуточные результаты, связанные с графикой, фиксацией состояния в контрольных точках и анализом. Фиксация состояния в контрольных точках особенна необходима в крупных системах HPC с длительными вычислениями, которые могут занимать дни, недели или месяцы.

ОПРЕДЕЛЕНИЕ *Фиксация состояния в контрольных точках* – это практика периодического сохранения состояния вычисления на диск, чтобы иметь возможность перезапускать вычисление в случае отказов системы или из-за конечных продолжительностей выполнений в пакетной системе.

При обработке данных высокопараллельных приложений необходим безопасный и эффективный способ чтения и хранения данных во время выполнения. Именно этим объясняется необходимость разбираться в файловых операциях в параллельном мире. Необходимо учитывать такие соображения, как правильность, сокращение дублирования результата и производительность.

Всегда важно помнить, что масштабирование производительности файловых систем не поспевает за остальным компьютерным оборудованием. Мы выполняем масштабирование вычислений до миллиардов ячеек или частиц, что предъявляет серьезные требования к файловым системам. С появлением машинного обучения и науки о данных все больше приложений нуждается в больших данных, для которых требуются крупные наборы файлов и сложные рабочие потоки с промежуточным хранением файлов.

Добавление понимания файловых операций в набор инструментов HPC приобретает все более высокую важность. В этой главе мы расскажем, как модифицировать файловые операции под параллельное приложение, чтобы обеспечивать вам возможность записывать данные эффективно и использовать доступное оборудование наилучшим образом. Хотя эта тема, возможно, не будет подробно освещаться во многих учебных пособиях по параллельному программированию, мы считаем, что она является основой, необходимой для современных параллельных приложений. Вы узнаете, как ускорять операцию записи файлов на порядки, сохраняя при этом правильность. Мы также рассмотрим разные программно-информационные и аппаратные средства, которые обычно используются для крупных систем HPC. Мы будем использовать пример записи данных из доменного разложения регулярной решетки с ореольными ячейками, используя разное программно-информационное обеспечение для параллельных файлов. Мы рекомендуем вам сверяться с примерами этой главы по адресу https://github.com/EssentialsOfParallelComputing/Chapter16.git.

16.1 Компоненты высокопроизводительной файловой системы

Сначала мы рассмотрим оборудование, которое входит в состав высокопроизводительной файловой системы. Традиционно файловые операции сохраняют данные на жестком диске посредством механизма, который записывает серию битов на магнитную подложку. Как и многие другие части систем HPC, оборудование для хранения данных стало сложнее с более глубокими иерархиями оборудования и разными характеристиками производительности. Эта эволюция оборудования для хранения данных аналогична углублению иерархии кеша у процессоров по мере повышения их производительности. Иерархия хранилища также помогает покрывать большую разницу в пропускной способности на уровне

процессора по сравнению с механическим дисковым хранилищем. Это обусловлено тем, что уменьшать размеры механических компонентов гораздо сложнее, чем электрических схем. Внедрение твердотельных накопителей (SSD) и других твердотельных устройств обеспечило обходной путь масштабированию физических вращающихся дисков.

Давайте сначала дадим определение тому, что может составлять систему хранения HPC, как показано на рис. 16.1. Типичные аппаратные компоненты хранения включают следующее:

- *вращающийся диск* – электромеханическое устройство, в котором данные хранятся в электромагнитном слое посредством перемещения механической записывающей головки;
- *SSD* – твердотельный накопитель (SSD) – это твердотельное устройство памяти, которое может заменять механический диск;
- *всплесковый буфер* – промежуточный аппаратный слой хранения данных, состоящий из компонентов NVRAM и SSD. Он расположен между вычислительным оборудованием и главными ресурсами дискового хранилища;
- *ленту* – магнитная лента с автоматически загружающимися картриджами.

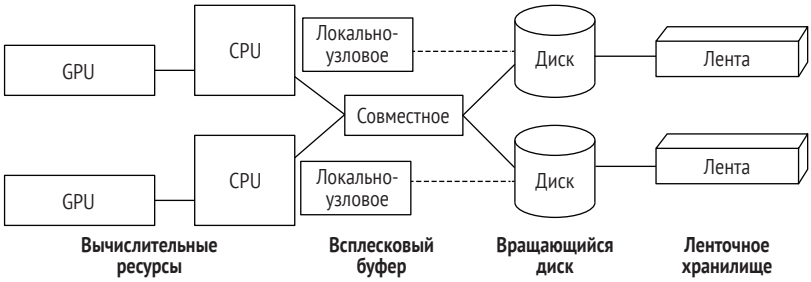


Рис. 16.1 Схема, показывающая расположение оборудования всплескового буфера между вычислительными ресурсами и дисковым хранилищем. Всплесковые буферы могут быть локальными для узлов или использоваться узлами по сети совместно

Схема хранения на рис. 16.1 иллюстрирует иерархию хранения между вычислительной системой и системой хранения. Всплесковые буферы (burst buffer), или буферы пиковой нагрузки, вставляются между вычислительным оборудованием и главным дисковым хранилищем, чтобы покрывать увеличивающийся разрыв в производительности. Всплесковые буферы могут размещаться на каждом узле либо на узлах ввода-вывода и использоваться в сети совместно с другими вычислительными узлами.

В связи с быстрым развитием технологии твердотельного хранения данных в ближайшем будущем будет эволюционировать дизайн всплесковых буферов. Помимо устранения разрыва в задержке и производительности пропускной способности, новые виды дизайна хранилищ все больше обусловлены необходимостью снижения требований

к энергопотреблению по мере увеличения размеров систем. Магнитная лента традиционно использовалась для длительного хранения, а некоторые виды дизайна даже обращались к «темному диску», где вращающиеся диски используются, но выключаются, когда в этом нет необходимости.

16.2 Стандартные файловые операции: интерфейс между параллельной и последовательной обработкой

Давайте сначала взглянем на стандартные файловые операции. В наших параллельных приложениях обычный интерфейс обработки файлов по-прежнему является последовательной операцией. Нецелесообразно иметь жесткий диск для каждого процессора. Даже файл для каждого процесса жизнеспособен только в лимитированных ситуациях и в малых масштабах. Как следствие, по каждой файловой операции мы переходим от параллельной к последовательной. Файловая операция должна рассматриваться как сокращение (или расширение для чтения) числа процессов, требующих специальной обработки для параллельных приложений. С этим параллелизмом можно справляться посредством нескольких простых модификаций стандартного файлового ввода и вывода (IO).

Крупная порция модификаций под параллельные приложения находится в интерфейсе файловых операций. Сначала мы должны еще раз рассмотреть наши предыдущие примеры, связанные с файловыми операциями. В качестве примера ввода файлов в разделе 8.3.2 показано, как читать данные в одном процессе, а затем передавать их другим процессам. В разделе 8.3.4 мы использовали операцию сбора в рамках MPI, чтобы результаты процессов записывались в детерминированном порядке.

(Профессиональный совет) Во избежание дальнейших осложнений первым шагом, который вы должны предпринимать при параллелизации приложения, является анализ исходного кода и вставка инструкции `if (rank == 0)` перед каждой инструкцией ввода-вывода. Просматривая код, вы должны выявить файловые операции, которые нуждаются в дополнительной трактовке. Эти операции включают в себя следующее (показано на рис. 16.2):

- открытие файлов только в одном процессе, а затем широковещательная передача данных другим процессам;
- распределение данных, которые необходимо подразделять между процессами с помощью операции разброса;
- обеспечение того, чтобы результат поступал только из одного процесса;
- сбор распределенных данных с помощью операции сбора перед их выводом.

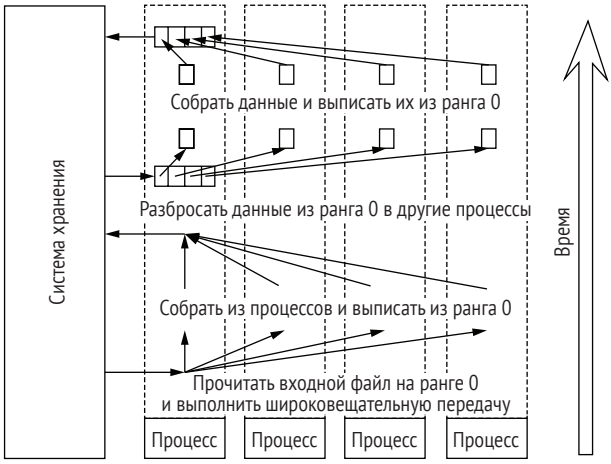


Рис. 16.2 Модификации под параллельное приложение для работы со стандартной файловой системой. Все файловые операции выполняются из ранга 0

Распространенной неэффективностью является открытие файла для каждого процесса; если представить, то это эквивалентно тому, что дюжина людей пытается открыть дверь в одно и то же время. Хотя ваша программа может и не рухнуть, но она станет причиной проблем в широком масштабе (представьте, что 1000 человек открывают ту же самую дверь). За файловые метаданные и вызываемую им установку замка в целях обеспечения правильности существует много соперничества, что при более крупном числе процессов может занимать несколько минут. Этого соперничества можно избежать, открывая файл только в одном процессе. Добавляя параллельные вызовы обмена данными в каждой точке перехода от последовательной к параллельной обработки и от параллельной к последовательной, мы можем придавать скромным параллельным приложениям способность работать с использованием стандартных файлов. Этого достаточно для подавляющего большинства параллельных приложений.

По мере увеличения размеров наших приложений мы больше не можем легко собирать или распределять данные в рамках одного процесса. Нашим самым большим лимитом является память; у нас недостаточно ресурсов памяти для одного процесса, чтобы сводить данные из тысяч других процессов к одному. Следовательно, у нас должен быть другой, более масштабируемый подход к файловым операциям. Это является темой следующих двух разделов, которые посвящены файловым операциям MPI, именуемым MPI-IO, и иерархическому формату данных v5 (HDF5). В указанных разделах мы покажем, как эти две библиотеки позволяют параллельному приложению трактовать файловые операции в параллельном стиле. Существуют и другие библиотеки обработки параллельных файлов, о которых мы упомянем в разделе 16.5.

16.3 Файловые операции MPI (MPI-IO) для более параллельного мира

Изучать библиотеку MPI-IO лучше всего, глядя на то, как она используется в реалистичном сценарии. Мы рассмотрим пример написания регулярной вычислительной сетки, которая была распределена по процессорам с ореольными ячейками, используя MPI-IO. В этом примере вы познакомитесь с базовой структурой, которая используется в MPI-IO, и несколькими наиболее распространенными функциональными вызовами.

Первые параллельные файловые операции были добавлены в MPI в стандарте MPI-2 в конце 1990-х годов. Первой широко доступной имплементацией файловых операций MPI под названием ROMIO руководил Раджив Тхакур (Rajeev Thakur) из Аргоннской национальной лаборатории (ANL). ROMIO можно использовать с любой имплементацией MPI. Большинство дистрибутивов MPI включает ROMIO в качестве стандартной части своего программно-информационного релиза. MPI-IO имеет массу функций, все они начинаются с префикса `MPI_File`. В этом разделе мы рассмотрим лишь подмножество наиболее часто используемых операций (см. табл.16.1).

Существуют разные способы использования MPI-IO. Нас интересует высокопараллельная версия, коллективная форма, в которой процессы работают вместе, записывая в свой раздел файла. Для этого мы воспользуемся возможностью создания нового типа данных MPI, который был впервые представлен в разделе 8.5.1.

Библиотека MPI-IO имеет как совместный файловый указатель для всех процессов, так и независимые файловые указатели для каждого процесса. Использование совместного указателя приводит к наложению замка на каждый процесс и сериализует файловые операции. Во избежание установления замков мы используем независимые файловые указатели для более высокой производительности.

Файловые операции подразделяются на коллективные и неколлективные операции. В коллективных операциях используются коллективные вызовы обмена данными MPI, и все участники коммуникатора должны делать вызов, иначе он зависнет. *Неколлективные вызовы* – это последовательные операции, которые вызываются отдельно по каждому процессу. В табл. 16.1 показано несколько общецелевых операций и соответствующие команды для каждой из них.

Таблица 16.1 Общие файловые процедуры MPI

Команда	Описание
`MPI_File_open`	Коллективное открытие файла
`MPI_File_seek`	Перемещает отдельные файловые указатели в это местоположение в файле
`MPI_File_set_size`	Выделяет указанное файловое пространство
`MPI_File_close`	Коллективное закрытие файла
`MPI_File_set_info`	Передает подсказки в библиотеку MPI-IO для более оптимизированных операций MPI

Операции открытия и закрытия файлов не требуют разъяснений. Операция поиска перемещает отдельный файловый указатель в указанное местоположение по каждому процессу. Процедуру MPI_ `File_set_info` можно использовать для передачи как общих, так и специфичных для поставщика подсказок. Существует также процедура `MPI_File_delete`, но это неколлективный вызов. В данном случае мы подразумеваем, что неколлективный вызов является последовательным: каждый процесс удаляет файл. Для программ на языках C и C++ функция remove работает с тем же успехом. Вызов процедуры `MPI_File_set_size` с ожидаемым размером файла бывает эффективнее, чем постепенное увеличение размера файла при каждой записи.

Мы начнем с рассмотрения независимых файловых операций для операций чтения и записи. Когда каждый процесс оперирует на своем независимом файловом указателе, это называется *независимой файловой операцией*. Независимые файловые операции полезны для записи реплицированных данных между процессами. Для этих общих данных их можно записывать из одного ранга с помощью процедур, приведенных в табл. 16.2.

Таблица 16.2 Независимые файловые процедуры MPI

Команда	Описание
`MPI_File_read`	Каждый процесс читает из своей текущей позиции файлового указателя
`MPI_File_write`	Каждый процесс пишет в свою текущую позицию файлового указателя
`MPI_File_read_at`	Перемещает файловый указатель в указанное местоположение и читает данные
`MPI_File_write_at`	Перемещает файловый указатель в указанное местоположение и пишет данные

Вы должны записывать распределенные данные с помощью коллективных операций (табл. 16.3). Когда процессы оперируют на файле коллективно, это называется *коллективной файловой операцией*. Соответствующие функции записи и чтения аналогичны независимым файловым операциям, но к имени функции добавляется `_all`. В целях наилучшего использования коллективных операций нам необходимо создавать сложные типы данных MPI. Функция `MPI_File_set_view` используется для задания схемы данных в файле.

Таблица 16.3 Коллективные файловые процедуры MPI

Команда	Описание
`MPI_File_set_view`	Вид на файл, видимый для каждого процесса. Устанавливает файловые указатели равными нулю
`MPI_File_read_all`	Все процессы коллективно читают данные со своего текущего независимого файлового указателя
`MPI_File_write_all`	Все процессы коллективно пишут данные из своего текущего независимого файлового указателя
`MPI_File_read_at_all`	Все процессы перемещаются в указанное местоположение файла и читают данные
`MPI_File_write_at_all`	Все процессы перемещаются в указанное местоположение файла и пишут данные

В этом примере мы разложим исходный код на четыре блока. (Полный исходный код этого примера включен в исходный код главы.) Прежде

всего мы должны начать с создания типа данных MPI для схемы данных в памяти и еще одного для схемы файлов; они называются соответственно пространством памяти (memspace) и пространством файла (filespace). На рис. 16.3 показаны эти типы данных для уменьшенной версии 4×4 нашего примера. Для простоты мы показываем только четыре процесса, каждый из которых имеет решетку 4×4, окруженную одноячеечным ореолом. Размер глубины ореола на рисунке равен ng (от англ. *number of ghost cells*), т. е. числу призрачных ячеек.

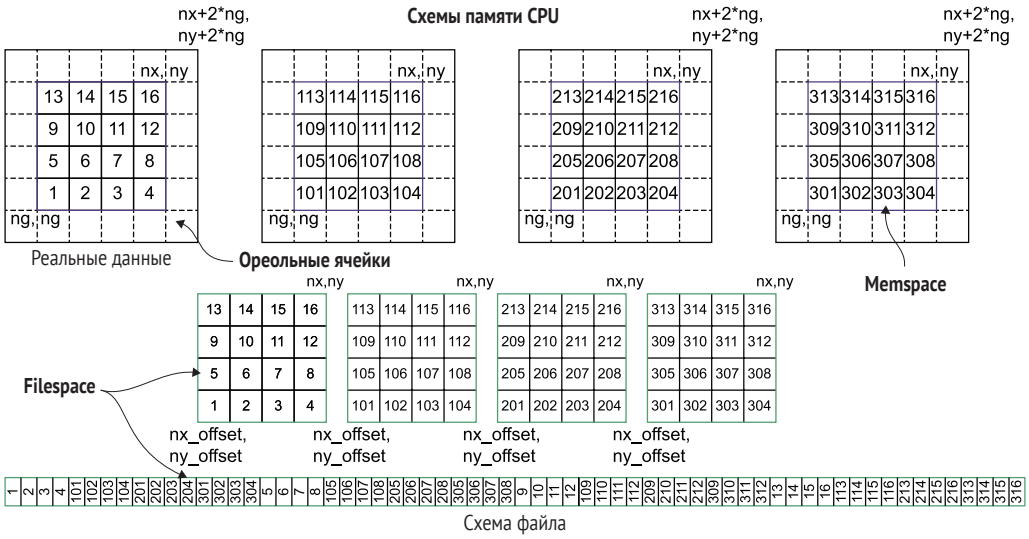


Рис. 16.3 Блоки данных 4×4 из каждого процесса, записываемые без ореольных ячеек в смежные разделы выходного файла. Верхняя строка – это схема памяти в процессе, именуемая пространством памяти, memspace. Средняя строка – это память в файле с убранными ореольными ячейками, именуемая пространством файла, filespace. Память в файле на самом деле является линейной, поэтому она принимает форму, показанную в последней строке

Первый блок кода в листинге 16.1 показывает создание указанных двух типов данных. Это нужно сделать только один раз в начале программы. Затем типы данных должны быть высвобождены в конце программы в процедуре финализации.

Листинг 16.1 Настройка типов пространств данных MPI-IO

MPI_IO_Examples/mpi_io_block2d/mpi_io_file_ops.c

```
10 void mpi_io_file_init(int ng, int ndims, int *global_sizes,
11    int *global_subsizes, int *global_starts, MPI_Datatype *memspace,
      MPI_Datatype *filespace){
12    // создать дескрипторы данных на диске и в памяти
13
14    // Глобальный вид всего двухмерного домена целиком
      // -- сопоставляет разложенные подмассивы
```

```
15    MPI_Type_create_subarray(ndims,                          Создает тип данных
         global_sizes, global_subsizes,                        для схемы файловых данных
16       global_starts, MPI_ORDER_C, MPI_DOUBLE,
         filespace);
17    MPI_Type_commit(filespace);  <-- Фиксирует тип файловых данных
18
19    // Локальная структура двухмерного подмассива
      // -- убирает призрачные ячейки на узле
20    int ny = global_subsizes[0], nx = global_subsizes[1];
21    int local_sizes[] = {ny+2*ng, nx+2*ng};
22    int local_subsizes[] = {ny, nx};
23    int local_starts[] = {ng, ng};
24
25    MPI_Type_create_subarray(ndim, local_sizes,              Создает тип данных
         local_subsizes, local_starts,                         для схемы данных памяти
26    MPI_ORDER_C, MPI_DOUBLE, memspace);
27    MPI_Type_commit(memspace);  <-- Фиксирует тип данных памяти
28 }
29
30 void mpi_io_file_finalize(MPI_Datatype *memspace,
         MPI_Datatype *filespace){
31    MPI_Type_free(memspace);                                 Высвобождает типы данных
32    MPI_Type_free(filespace);
33 }
```

На этом первом шаге мы создали два типа данных из рис. 16.1. Теперь нам нужно записать эти типы данных в файл. Процесс записи состоит из четырех шагов, как показано в листинге 16.2.

1. Создать файл.
2. Установить вид на файл.
3. Записать каждый массив с помощи коллективного вызова.
4. Закрыть файл.

Листинг 16.2 Запись файла MPI-IO

MPI_IO_Examples/mpi_io_block2d/mpi_io_file_ops.c

```
35 void write_mpi_io_file(const char *filename, double **data,
36       int data_size, MPI_Datatype memspace, MPI_Datatype filespace,
         MPI_Comm mpi_io_comm){
37    MPI_File file_handle = create_mpi_io_file(               Создает файл
38       filename, mpi_io_comm, (long long)data_size);
39
40    MPI_File_set_view(file_handle, file_offset,              Задает панораму файла
41       MPI_DOUBLE, filespace, "native",
         MPI_INFO_NULL);
42    MPI_File_write_all(file_handle,                          Записывает массивы
         &(data[0][0]), 1, memspace,
         MPI_STATUS_IGNORE);
43    file_offset += data_size;
44
```

```
45    MPI_File_close(&file_handle);
46    file_offset = 0;
47 }
48
49 MPI_File create_mpi_io_file(const char *filename, MPI_Comm mpi_io_comm,
50       long long file_size){
51    int file_mode = MPI_MODE_WRONLY | MPI_MODE_CREATE |
                      MPI_MODE_UNIQUE_OPEN;
52
53    MPI_Info mpi_info = MPI_INFO_NULL; // Для подсказок ввода-вывода MPI
54    MPI_Info_create(&mpi_info);
55    MPI_Info_set(mpi_info,
         "collective_buffering", "1");
56    MPI_Info_set(mpi_info,
         "striping_factor", "8");
57    MPI_Info_set(mpi_info,
         "striping_unit", "4194304");
58
59    MPI_File file_handle = NULL;
60    MPI_File_open(mpi_io_comm, filename, file_mode, mpi_info,
                    &file_handle);
61    if (file_size > 0)
         MPI_File_set_size(file_handle, file_size);
62    file_offset = 0;
63    return file_handle;
64 }
```

Закрывает файл

Передает подсказки для коллективной операции

Передает подсказки для подразделения на полосы в файловой системе Lustre

Предварительно выделяет файловое пространство для более высокой производительности

Во время открытия с помощью подсказок в объекте `MPI_Info` (строка 53) можно выполнить несколько оптимизаций. Подсказка может заключаться в том, что файловые операции следует выполнять с использованием коллективных операций, `collective_buffering`, как в строке 55. Либо подсказкой может быть файловая система, специфичная для подразделения на полосы по восьми жестким дискам, `striping_factor = 8`, как в строке 56. Мы обсудим подсказки подробнее в разделе 16.6.1.

Мы также можем предварительно выделить файловое пространство, как показано в строке 61, чтобы его не приходилось увеличивать во время записи. Чтение файла состоит из тех же четырех шагов, что и приведенный ранее процесс записи, и показано в следующем ниже листинге.

Листинг 16.3 Чтение файла MPI-IO

`MPI_IO_Examples/mpi_io_block2d/mpi_io_file_ops.c`

```
66 void read_mpi_io_file(const char *filename, double **data, int data_size,
67          MPI_Datatype memspace, MPI_Datatype filespace, MPI_Comm mpi_io_comm){
68    MPI_File file_handle = open_mpi_io_file(
         filename, mpi_io_comm);
69
70    MPI_File_set_view(file_handle, file_offset,
71       MPI_DOUBLE, filespace, "native",
         MPI_INFO_NULL);
```

Открывает файл

Устанавливает план файла

```
72    MPI_File_read_all(file_handle,
         &(data[0][0]), 1, memspace,
         MPI_STATUS_IGNORE);
73    file_offset += data_size;
74
75    MPI_File_close(&file_handle);
76    file_offset = 0;
77 }
78
79 MPI_File open_mpi_io_file(const char *filename, MPI_Comm mpi_io_comm){
80    int file_mode = MPI_MODE_RDONLY | MPI_MODE_UNIQUE_OPEN;
81
82    MPI_Info mpi_info = MPI_INFO_NULL; // Для подсказок ввода-вывода MPI
83    MPI_Info_create(&mpi_info);
84    MPI_Info_set(mpi_info, "collective_buffering", "1");
85
86    MPI_File file_handle = NULL;
87    MPI_File_open(mpi_io_comm, filename, file_mode, mpi_info,
                  &file_handle);
88    return file_handle;
89 }
```

Коллективное чтение массивов

Закрывает файл

Операция чтения требует меньше подсказок и настроек, чем операция записи. Это связано с тем, что некоторые настройки для чтения определяются из файла. До сих пор эти файловые операции MPI-IO писались в общей форме, которую можно вызывать для решения любой задачи. Теперь давайте взглянем на главный исходный код приложения в следующем ниже листинге, который задает вызовы.

Листинг 16.4 Главный исходный код приложения

MPI_IO_Examples/mpi_io_block2d/mpi_io_block2d.c

```
 9 int main(int argc, char *argv[])
10 {
11    MPI_Init(&argc, &argv);
12
13    int rank, nprocs;
14    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
15    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
16
17    // для многочисленных файлов подразделить коммуникатор и
      // установить цвета для каждого набора
18    MPI_Comm mpi_io_comm = MPI_COMM_NULL;
19    int nfiles = 1;
20    float ranks_per_file = (float)nprocs/(float)nfiles;
21    int color = (int)((float)rank/ranks_per_file);
22    MPI_Comm_split(MPI_COMM_WORLD, color, rank, &mpi_io_comm);
23    int nprocs_color, rank_color;
24    MPI_Comm_size(mpi_io_comm, &nprocs_color);
25    MPI_Comm_rank(mpi_io_comm, &rank_color);
```

```
26    int row_color = 1, col_color = rank_color;
27    MPI_Comm mpi_row_comm, mpi_col_comm;
28    MPI_Comm_split(mpi_io_comm, row_color, rank_color, &mpi_row_comm);
29    MPI_Comm_split(mpi_io_comm, col_color, rank_color, &mpi_col_comm);
30
31    // устанавливает размерности массива данных и число призрачных ячеек
32    int ndim = 2, ng = 2, ny = 10, nx = 10;
33    int global_subsizes[] = {ny, nx};
34
35    int ny_offset = 0, nx_offset = 0;
36    MPI_Exscan(&nx, &nx_offset, 1, MPI_INT, MPI_SUM, mpi_row_comm);
37    MPI_Exscan(&ny, &ny_offset, 1, MPI_INT, MPI_SUM, mpi_col_comm);
38    int global_offsets[] = {ny_offset, nx_offset};
39
40    int ny_global, nx_global;
41    MPI_Allreduce(&nx, &nx_global, 1, MPI_INT, MPI_SUM, mpi_row_comm);
42    MPI_Allreduce(&ny, &ny_global, 1, MPI_INT, MPI_SUM, mpi_col_comm);
43    int global_sizes[] = {ny_global, nx_global};
44    int data_size = ny_global*nx_global;
45
46    double **data = (double **)malloc2D(ny+2*ng, nx+2*ng);
47    double **data_restore = (double **)malloc2D(ny+2*ng, nx+2*ng);
        < ... пропускаем инициализацию данных ... >
54
55    MPI_Datatype memspace = MPI_DATATYPE_NULL,
                   filespace = MPI_DATATYPE_NULL;
56    mpi_io_file_init(ng, global_sizes,                  Инициализирует и настраивает
         global_subsizes, global_offsets,                 типы данных
57       &memspace, &filespace);
58
59    char filename[30];
60    if (ncolors > 1) {
61       sprintf(filename,"example_%02d.data",color);
62    } else {
63       sprintf(filename,"example.data");
64    }
65
66    // Выполнить расчет и записать последовательность файлов
67    write_mpi_io_file(filename, data,
         data_size, memspace, filespace,          Пишет данные
         mpi_io_comm);
68    // Прочитать данные назад для верификации файловых операций
69    read_mpi_io_file(filename, data_restore,
70       data_size, memspace, filespace,          Читает данные
         mpi_io_comm);
71
72    mpi_io_file_finalize(&memspace, &filespace);
73
        < ... пропускаем верификационный исходный код ... >
                                                  Закрывает файл
105                                               и высвобождает типы данных
106    free(data);
```

```
107    free(data_restore);
108
109    MPI_Comm_free(&mpi_io_comm);
110    MPI_Comm_free(&mpi_row_comm);
111    MPI_Comm_free(&mpi_col_comm);
112    MPI_Finalize();
113    return 0;
114 }
```

Эта конфигурация требует небольшого объяснения. Приведенный выше исходный код поддерживает возможность записи более одного файла данных MPI. Это обычно называется записью в файл $N \times M$, где N процессов записывают M файлов и где M больше одного, но намного меньше, чем число процессов (рис. 16.4). Причина этого технического приема заключается в том, что при более крупных размерах задачи запись в один файл не всегда хорошо масштабируется.

Мы можем разбить процессы на группы по цветам, как показано на рис. 16.4. В строках 17–22 листинга 16.4 мы настроили новый коммуникатор на основе M цветов, где M – это число файлов. Число файлов задается в строке 19, а наш цвет вычисляется в строках 20 и 21. Для `ranks_per_file` используется тип с плавающей точкой с целью манипулирования неравномерным делением рангов. Затем мы получаем новый ранг в пределах нашего цвета. Каждая коммуникационная группа в правой части рис. 16.4 имеет 4096 процессов или рангов. Порядок рангов такой же, как и в глобальной коммуникационной группе. Если существует более одного файла, то имена файлов включают номер цвета в строках 59–64. В настоящее время этот исходный код задает только один цвет и пишет только один файл, как показано на левой стороне рис. 16.4, но он написан для поддержки большего числа файлов.

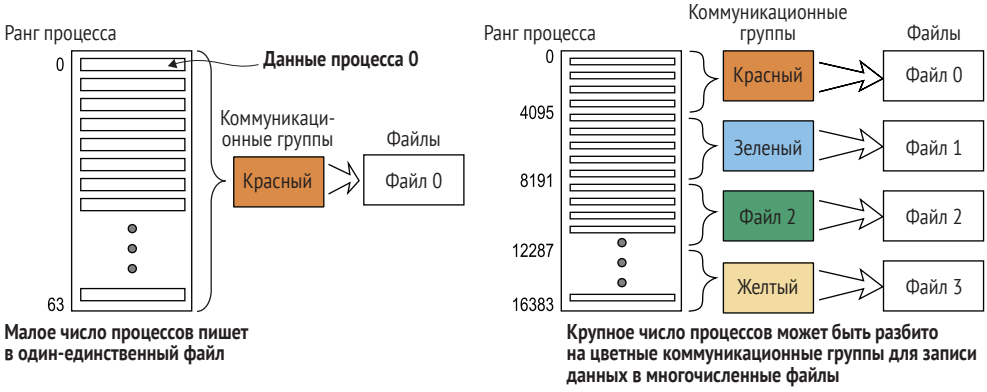


Рис. 16.4 При больших размерах процессы могут разбиваться на коммуникационные группы по цветам, чтобы они писали в отдельные файлы. Ранги подгрупп расположены в том же порядке, что и ранги в изначальном коммуникаторе

Нам также необходимо знать местоположения начальных значений *x* и *y* каждого процесса. В случае разложений данных, содержащих одинаковое число строк и столбцов по каждому процессу, вычислению необходимо знать только местоположение процесса в глобальном наборе. Но когда число строк и столбцов в разных процессах варьируется, нам нужно суммировать все размеры ниже нашей позиции. Как мы уже обсуждали ранее в разделе 5.6, эта операция представляет собой обычный параллельный шаблон, именуемый *сканированием*. В целях выполнения этого расчета в строках 22–34 мы создаем коммуникаторы для каждой строки и столбца. Они выполняют эксклюзивную операцию сканирования, чтобы получить начальное местоположение *x* и *y* по каждому процессу. В приведенном выше исходном коде мы подразделяем данные только в направлении координат *x*, чтобы сделать все немного проще. Глобальные размеры и размеры процессов в подразделах массива указаны в строках 27–44. Сюда входят сдвиги данных, рассчитанные с использованием эксклюзивных сканов.

Теперь, когда у нас есть вся необходимая информация о разложении данных, можем вызвать нашу подпрограмму `mpi_io_file_init` в строке 52, чтобы настроить типы данных MPI для схемы памяти и файловой системы. Это нужно сделать только один раз, при запуске. Затем мы можем свободно вызывать наши подпрограммы для записи, `write_mpi_io_file`, и чтения, `read_mpi_io_file`, в строках 63 и 65. Мы можем вызывать их столько раз, сколько потребуется во время выполнения. В нашем примере исходного кода мы затем проверяем прочитанные данные, сравниваем их с изначальными данными и печатаем ошибку, если она возникает. Наконец, мы открываем файл в одном процессе и используем стандартное двоичное чтение языка C, чтобы показать, как данные расположены в файле. Это делается путем последовательного чтения каждого значения из файла и его распечатки.

Теперь давайте скомпилируем и выполним этот пример. Сборка является стандартной сборкой CMake, и мы выполняем ее на четырех процессорах.

```
mkdir build && cd build
cmake ..
make
mpirun -n 4 ./mpi_io_block2d
```

На рис. 16.5 показан результат на выходе из стандартного двоичного чтения на C для решетки 10×10 на каждом процессоре.

```
x[0][ ]   1   2   3   4   5   6   7   8   9  10 101 102 103 104 105 106 107 108 109 110
        201 202 203 204 205 206 207 208 209 210 301 302 303 304 305 306 307 308 309 310
x[1][ ]  11  12  13  14  15  16  17  18  19  20 111 112 113 114 115 116 117 118 119 120
        211 212 213 214 215 216 217 218 219 220 311 312 313 314 315 316 317 318 319 320
x[2][ ]  21  22  23  24  25  26  27  28  29  30 121 122 123 124 125 126 127 128 129 130
        221 222 223 224 225 226 227 228 229 230 321 322 323 324 325 326 327 328 329 330
x[3][ ]  31  32  33  34  35  36  37  38  39  40 131 132 133 134 135 136 137 138 139 140
        231 232 233 234 235 236 237 238 239 240 331 332 333 334 335 336 337 338 339 340
x[4][ ]  41  42  43  44  45  46  47  48  49  50 141 142 143 144 145 146 147 148 149 150
        241 242 243 244 245 246 247 248 249 250 341 342 343 344 345 346 347 348 349 350
x[5][ ]  51  52  53  54  55  56  57  58  59  60 151 152 153 154 155 156 157 158 159 160
        251 252 253 254 255 256 257 258 259 260 351 352 353 354 355 356 357 358 359 360
x[6][ ]  61  62  63  64  65  66  67  68  69  70 161 162 163 164 165 166 167 168 169 170
        261 262 263 264 265 266 267 268 269 270 361 362 363 364 365 366 367 368 369 370
x[7][ ]  71  72  73  74  75  76  77  78  79  80 171 172 173 174 175 176 177 178 179 180
        271 272 273 274 275 276 277 278 279 280 371 372 373 374 375 376 377 378 379 380
x[8][ ]  81  82  83  84  85  86  87  88  89  90 181 182 183 184 185 186 187 188 189 190
        281 282 283 284 285 286 287 288 289 290 381 382 383 384 385 386 387 388 389 390
x[9][ ]  91  92  93  94  95  96  97  98  99 100 191 192 193 194 195 196 197 198 199 200
        291 292 293 294 295 296 297 298 299 300 391 392 393 394 395 396 397 398 399 400
```

Рис. 16.5 Результат работы небольшого исходного кода двоичного чтения для MPI-IO показывает содержимое файла. При использовании библиотеки MPI-IO нам пришлось написать небольшую утилиту для проверки содержимого файла

16.4 HDF5 как самоописывающий формат для более качественного управления данными

В традиционных форматах файлов данных данные не имеют смысла без программного кода, используемого для записи и чтения файла. В иерархическом формате данных (HDF), версии 5, используется другой подход. HDF5 обеспечивает самоописывающий параллельный формат данных. HDF5 называется самоописывающим, потому что имя и характеристики хранятся в файле с данными. В HDF5, имея описание содержащихся в файле данных, вы больше не нуждаетесь в исходном коде и можете читать данные, просто запрашивая файл. HDF5 также имеет богатый набор утилит командной строки (таких как h5ls и h5dump), которые можно использовать для запрашивания содержимого файла. Вы обнаружите, что утилиты полезны при проверке правильности записи ваших файлов.

Мы хотим записывать данные в двоичном формате из-за скорости и точности. Но, поскольку они находятся в двоичном формате, трудно выполнять проверку правильности записанных данных. Если мы будем читать данные обратно, то в процессе чтения проблема может возникнуть. Утилита, которая может запрашивать файл, предоставляет способ проверять операцию записи отдельно от операции чтения. На рис. 16.4 в предыдущем разделе, посвященном MPI-IO, нам понадобилась не-

большая программа для чтения содержимого файла. Для HDF5 этого не требуется, потому что утилита уже предусмотрена. На рис. 16.6 (показанном далее в этом разделе) мы использовали утилиту командной строки h5dump для просмотра содержимого. Избежать необходимости писать исходный код для многих распространенных операций можно, используя уже существующие утилиты HDF5.

Параллельный исходный код HDF5 имплементирован с использованием MPI-IO. Поскольку он строится на MPI-IO, структура HDF5 аналогична. Несмотря на схожесть, терминология и отдельные вызовы функций различаются настолько, что вызывают некоторые трудности. Мы рассмотрим функции, необходимые для написания аналогичной процедуры параллельной обработки файлов, как это было сделано для MPI-IO. Библиотека HDF5 разделена на функциональные группы более низкого уровня. Эти функциональные группы удобно различать по префиксам всех вызовов в группе. Первая группа – это обязательные операции манипулирования файлами (табл. 16.4), которые в совокупности обрабатывают операции открытия и закрытия файлов.

Таблица 16.4 Коллективные файловые процедуры HDF5

Команда	Описание
`H5Fcreate`	Коллективное открытие файла, которое будет создавать файл, если он не существует
`H5Fopen`	Коллективное открытие файла, который уже существует
`H5Fclose`	Коллективное закрытие файла

Далее нам нужно определить новые типы памяти. Они используются для указания частей данных для записи и схемы. В HDF5 эти типы памяти называются пространствами данных. Операции с пространством данных в табл. 16.5 включают способы извлечения шаблонов из многомерного массива. Информация о многих дополнительных процедурах находится в разделе «Дальнейшее чтение» в конце настоящей главы (16.7.1).

Таблица 16.5 Процедуры пространства данных HDF5

Команда	Описание
`H5Screate_simple`	Создает тип многомерного массива
`H5Sselect_hyperslab`	Создает тип гиперпластового (hyperslab) участка частей многомерного массива
`H5Sclose`	Освобождает пространство данных

В пространстве данных есть и другие операции, включая операции на основе точек, которые мы здесь не рассматривали. Теперь нужно применить эти пространства данных к набору многомерных массивов (табл. 16.6). В HDF5 многомерный массив называется *набором данных*, который обычно представляет собой многомерный массив или какую-либо другую форму данных в приложении.

Осталась только одна операционная группа, которая нам нужна. Эта группа, именуемая *списками свойств*, позволяет модифицировать или предоставлять подсказки для операций, как показано в табл. 16.7. Списки

свойств можно использовать для установки значений атрибутов, чтобы применять коллективные операции с чтениями или записями. Списки свойств также можно использовать для передачи подсказок в опорную библиотеку MPI-IO.

Таблица 16.6 Процедуры набора данных HDF5

Команда	Описание
`H5Dcreate2`	Создает пространство для набора данных в файле
`H5Dopen2`	Открывает существующий набор данных, как описано внутри файла
`H5Dclose`	Закрывает набор данных внутри файла
`H5Dwrite`	Пишет набор данных в файл, используя `filespace` и `memspace`
`H5Dread`	Читает набор данных из файла, используя `filespace` и `memspace`

Таблица 16.7 Процедуры HDF5 списка свойств

Команда	Описание
`H5Pcreate`	Создает список свойств
`H5Pclose`	Высвобождает список свойств
`H5Pset_dxpl_mpio`	Задает список свойств для передачи данных
`H5Pset_coll_metadata_write`	Задает коллективные операции записи метаданных для всех процессов в группе
`H5Pset_fapl_mpio`	Сохраняет свойства MPI-IO в списке свойств доступа к файлам
`H5Pset_all_coll_metadata_ops`	Задает операции параллельного чтения метаданных в списке свойств доступа к файлам

Давайте перейдем к примеру. Мы начинаем этот пример HDF5 с исходного кода создания файла и пространств данных памяти. Указанный процесс показан в следующем ниже листинге. В этом листинге все аргументы для HDF5 выделены жирным шрифтом.

Листинг 16.5 Настройка типов пространства данных HDF5

`HDF5Examples/hdf5block2d/hdf5_file_ops.c`

```
11 void hdf5_file_init(int ng, int ndims, int ny_global, int nx_global,
12       int ny, int nx, int ny_offset, int nx_offset, MPI_Comm mpi_hdf5_comm,
13       hid_t *memspace, hid_t *filespace){
14    // создать дескрипторы данных на диске и в памяти
15    *filespace = create_hdf5_filespace(ndims,
         ny_global, nx_global, ny, nx,
16       ny_offset, nx_offset, mpi_hdf5_comm);
17    *memspace =
         create_hdf5_memspace(ndims ny, nx, ng);
18 }
19
20 hid_t create_hdf5_filespace(int ndims, int ny_global, int nx_global,
21       int ny, int nx, int ny_offset, int nx_offset,
         MPI_Comm mpi_hdf5_comm){
22    // создать пространство данных для данных, хранящихся на диске,
      // используя вызов гиперпласта
```

Создает пространство файловых данных

Создает пространство данных памяти

```
23    hsize_t dims[] = {ny_global, nx_global};
24
25    hid_t filespace = H5Screate_simple(ndims,          Создает объект
                        dims, NULL);                   файлового пространства
26
27    // определить сдвиг в файловом пространстве для текущего процесса
28    hsize_t  start[] = {ny_offset, nx_offset};
29    hsize_t stride[] = {1,         1};
30    hsize_t  count[] = {ny,        nx};
31
32    H5Sselect_hyperslab(filespace, H5S_SELECT_SET,     Выбирает гиперпласт
33                        start, stride, count, NULL);   файлового пространства
34    return filespace;
35 }
36
37 hid_t create_hdf5_memspace(int ndims, int ny, int nx, int ng) {
38    // создать пространство памяти в памяти, используя вызов гиперпласта
39    hsize_t dims[] = {ny+2*ng, nx+2*ng};
40
41    hid_t memspace = H5Screate_simple(ndims, dims, NULL);   <-- Создает объект
42                                                            пространства памяти
43    // выбрать реальные данные из массива
44    hsize_t  start[] = {ng,   ng};
45    hsize_t stride[] = {1,    1};
46    hsize_t  count[] = {ny,   nx};
47
48    H5Sselect_hyperslab(memspace, H5S_SELECT_SET,      Создает гиперпласт
49                        start, stride, count, NULL);   пространства памяти
50    return memspace;
51 }
52
53 void hdf5_file_finalize(hid_t *memspace, hid_t *filespace){
54    H5Sclose(*memspace);
55    *memspace = H5S_NULL;
56    H5Sclose(*filespace);
57    *filespace = H5S_NULL;
58 }
```

В листинге 16.5 при создании двух пространств данных мы использовали один и тот же шаблон: создать объект данных, задать размерные аргументы данных, а затем выбрать прямоугольный участок массива. Сначала мы создали глобальное пространство массива с помощью вызова процедуры `H5Screate_simple`. Для пространства файловых данных мы установили размерности равными размерам глобального массива `nx_global` и `ny_global` в строке 23, а затем использовали эти размеры в строке 25 для создания пространства данных. Затем мы выбрали участок пространства файловых данных для каждого процессора с помощью вызовов процедуры `H5Sselect _hyperslab` в строках 32 и 48. Затем аналогичный процесс выполняется для пространства данных памяти.

Теперь, когда у нас есть пространства данных, процесс записи данных в файл прямолинеен. Мы открываем файл, создаем набор данных и его

пишем. Если есть другие наборы данных, то мы продолжаем их писать, а когда закончим, закрываем файл. В следующем ниже листинге показано, как это делается.

Листинг 16.6 Запись в файл HDF5

HDF5Examples/hdf5block2d/hdf5_file_ops.c

```
60 void write_hdf5_file(const char *filename, double **data1,
61       hid_t memspace, hid_t filespace, MPI_Comm mpi_hdf5_comm) {
62    hid_t file_identifier = create_hdf5_file(
          filename, mpi_hdf5_comm);
63
64    // Создать список свойств для коллективной записи набора данных.
65    hid_t xfer_plist = H5Pcreate(H5P_DATASET_XFER);
66    H5Pset_dxpl_mpio(xfer_plist, H5FD_MPIO_COLLECTIVE);
67
68    hid_t dataset1 = create_hdf5_dataset(
          file_identifier, filespace);
69    //hid_t dataset2 = create_hdf5_dataset(file_identifier, filespace);
70
71    // записать данные на диск, используя пространство памяти
      // и пространство данных.
72    H5Dwrite(dataset1, H5T_IEEE_F64LE,
             memspace, filespace, xfer_plist,
73           &(data1[0][0]));
74    //H5Dwrite(dataset2, H5T_IEEE_F64LE,
      //         memspace, filespace, xfer_plist,
75    //         &(data2[0][0]));
76
77    H5Dclose(dataset1);
78    //H5Dclose(dataset2);
79
80    H5Pclose(xfer_plist);
81
82    H5Fclose(file_identifier);
83 }
84
85 hid_t create_hdf5_file(const char *filename, MPI_Comm mpi_hdf5_comm){
86    hid_t file_creation_plist = H5P_DEFAULT;
87    // задать шаблон файлового доступа для параллельного доступа к вводу-выводу
88    hid_t file_access_plist = H5P_DEFAULT;
89    file_access_plist = H5Pcreate(H5P_FILE_ACCESS);
90
91    // задать коллективный режим для записи метаданных
92    H5Pset_coll_metadata_write(file_access_plist, true);
93
94    MPI_Info mpi_info = MPI_INFO_NULL;
95    MPI_Info_create(&mpi_info);
96    MPI_Info_set(mpi_info, "striping_factor", "8");
97    MPI_Info_set(mpi_info, "striping_unit", "4194304");
98
99    // сообщить библиотеке HDF5, что мы хотим использовать MPI-IO для записи
```

Вызывает подпрограмму для создания файла

Вызывает подпрограмму для создания набора данных

Пишет набор данных

Закрывает объекты и файл данных

Создает список свойств создания файла

Создает свойство доступа к файлу

Создает подсказки ввода-вывода MPI

```
100    H5Pset_fapl_mpio(file_access_plist, mpi_hdf5_comm, mpi_info);
101
102    // Открыть файл коллективно
103    // H5F_ACC_TRUNC - записывать поверх существующего файла.
       // H5F_ACC_EXCL  - не записывать поверх
104    // 3-й аргумент - это список свойств создания. Используется значение
          по умолчанию
105    // 4-й аргумент - это идентификатор списка свойств файлового доступа
106    hid_t file_identifier = H5Fcreate(filename,
107    H5F_ACC_TRUNC, file_creation_plist,
          file_access_plist);
108
109    // высвободить шаблон файлового доступа
110    H5Pclose(file_access_plist);
111    MPI_Info_free(&mpi_info);
112
113    return file_identifier;
114 }
115
116 hid_t create_hdf5_dataset(hid_t file_identifier, hid_t filespace){
117    // создать набор данных
118    hid_t link_creation_plist    = H5P_DEFAULT;
119    hid_t dataset_creation_plist = H5P_DEFAULT;
120    hid_t dataset_access_plist   = H5P_DEFAULT;
121    hid_t dataset = H5Dcreate2(
122       file_identifier,        // Арг 1: идентификатор файла
123       "data array",           // Арг 2: имя набора данных
124       H5T_IEEE_F64LE,         // Арг 3: идентификатор типа данных
125       filespace,              // Арг 4: идентификатор файлового пространства
126       link_creation_plist,    // Арг 5: список свойств создания ссылок
127       dataset_creation_plist, // Арг 6: список свойств создания набора данных
128       dataset_access_plist);  // Арг 7: список свойств доступа к набору данных
129
130    return dataset;
131 }
```

Процедура HDF5 создает файл

Создает список свойств создания набора данных

Создает список свойств создания ссылок

Создает список свойств доступа к набору данных

Процедура HDF5 создает набор данных

В листинге 16.6 главная процедура `write_hdf5_file` использует пространство данных filespace, созданное в листинге 16.5. Затем мы записали набор данных с помощью процедуры `H5Dwrite` в строке 72, используя пространство памяти и пространство файлов. Мы также создали и передали список свойств, чтобы указать библиотеке HDF5 использовать коллективные процедуры MPI-IO. Наконец, в строке 82 мы закрыли файл. Мы также закрыли список свойств и набор данных в предыдущих строках, чтобы избежать утечки памяти. Для выполнения процедуры создания файла мы, наконец, вызываем процедуру `H5Fcreate` в строке 106, но нужно несколько строк для настройки подсказок. Мы обернули настройку списка свойств для коллективной записи и подсказки MPI-IO вместе с вызовом и поместили их в отдельную процедуру. Мы также применили тот же подход к вызову HDF5 в строке 121 для создания набора данных, чтобы иметь возможность подробно описывать разные списки свойств, которые можно использовать.

Процедура чтения файла данных HDF5, показанная в следующем ниже листинге, имеет тот же базовый шаблон, что и предыдущая операция записи. Самая большая разница между этим листингом и листингом 16.6 заключается в том, что требуется меньше подсказок и атрибутов.

Листинг 16.7 Чтение файла HDF5

HDF5Examples/hdf5block2d/hdf5_file_ops.c

```
135 void read_hdf5_file(const char *filename, double **data1,
136       hid_t memspace, hid_t filespace, MPI_Comm mpi_hdf5_comm) {
137    hid_t file_identifier =                          Вызывает подпрограмму
          open_hdf5_file(filename, mpi_hdf5_comm);    открытия файла
138
139    // Создать список свойств для коллективной записи набора данных.
140    hid_t xfer_plist = H5Pcreate(H5P_DATASET_XFER);
141    H5Pset_dxpl_mpio(xfer_plist, H5FD_MPIO_COLLECTIVE);
142                                                     Вызывает подпрограмму
143    hid_t dataset1 =                                 создания набора данных
          open_hdf5_dataset(file_identifier);
144    // прочитать данные с диска, используя пространство памяти
       // и пространство данных.
145    H5Dread(dataset1, H5T_IEEE_F64LE, memspace,      Читает набор данных
146    filespace, H5P_DEFAULT, &(data1[0][0]));
147    H5Dclose(dataset1);
148
149    H5Pclose(xfer_plist);
150                                                     Закрывает объекты и файл данных
151    H5Fclose(file_identifier);
152 }
153
154    hid_t open_hdf5_file(const char *filename, MPI_Comm mpi_hdf5_comm){
155    // задать шаблон файлового доступа для параллельного доступа к вводу-выводу
156    hid_t file_access_plist = H5P_DEFAULT; // Список свойств файлового доступа
157    file_access_plist = H5Pcreate(H5P_FILE_ACCESS);
158
159    // задать коллективный режим для чтений метаданных (ops)
160    H5Pset_all_coll_metadata_ops(file_access_plist, true);
161
162    // сообщить библиотеке HDF5, что мы хотим использовать MPI-IO для чтения
163    H5Pset_fapl_mpio(file_access_plist, mpi_hdf5_comm, MPI_INFO_NULL);
164
165    // Открыть файл коллективно
166    // H5F_ACC_RDONLY - задает доступ к чтению и записи
       // на открытии существующего файла.
167    // 3-й аргумент - это идентификатор списка свойств файлового доступа
168    hid_t file_identifier = H5Fopen(filename,        Процедура HDF5 открывает файл
          H5F_ACC_RDONLY, file_access_plist);
169
170    // освобождает шаблон файлового доступа
171    H5Pclose(file_access_plist);
```

```
172
173     return file_identifier;
174 }
175
176 hid_t open_hdf5_dataset(hid_t file_identifier){
177     // открыть набор данных
178     hid_t dataset_access_plist = H5P_DEFAULT;
179     hid_t dataset = H5Dopen2(
180         file_identifier,       // Арг 1: идентификатор файла
181         "data array",          // Арг 2: имя набора данных для соотнесения
с чтением
182         dataset_access_plist); // Арг 3: список свойств доступа к набору данных
183
184     return dataset;
185 }
```

Создает список свойств доступа к набору данных

Процедура HDF5 создает набор данных

Поскольку файл уже существует, мы используем вызов open в строке 168 в листинге 16.7, чтобы указать режим только для чтения. (Использование режима только для чтения позволяет выполнять дополнительную оптимизацию.) Файл, к которому осуществляется доступ, уже имеет несколько атрибутов, которые были указаны во время записи. Некоторые из этих атрибутов не нужно указывать при чтении. Листинги HDF5 до этого могли содержать общецелевую библиотеку внутри приложения. В следующем ниже листинге показаны вызовы, которые будут размещаться в разных точках главного приложения.

Листинг 16.8 Файл главного приложения

HDF5Examples/hdf5block2d/hdf5block2d.c

```
52     hid_t memspace = H5S_NULL, filespace = H5S_NULL;
53     hdf5_file_init(ng, ndims, ny_global,
          nx_global, ny, nx, ny_offset, nx_offset,
54        mpi_hdf5_comm, &memspace, &filespace);
55
56     char filename[30];
57     if (ncolors > 1) {
58        sprintf(filename,"example_%02d.hdf5",color);
59     } else {
60        sprintf(filename,"example.hdf5");
61     }
62
63     // Сделать расчеты и записать последовательность файлов
64     write_hdf5_file(filename, data, memspace,
                       filespace, mpi_hdf5_comm);
65     // Прочитать данные назад для верифицирования файловых операций
66     read_hdf5_file(filename, data_restore,
            memspace, filespace, mpi_hdf5_comm);
67
68     hdf5_file_finalize(&memspace, &filespace);
```

Настраивает области данных памяти и данных файлов

Пишет файл данных HDF5

Читает данные из файла данных HDF5

Высвобождает объекты пространства данных

В листинге 16.8 операцию инициализации для настройки пространств данных в строке 53 можно выполнить один раз при запуске программы. Затем вы можете периодически писать данные в своей программе для графики и фиксации состояния в контрольных точках. Тогда чтение обычно выполняется при перезапуске с контрольной точки в начале выполнения. Наконец, вызов процедуры finalize следует выполнять в конце программы перед завершением вычисления. Теперь давайте скомпилируем и выполним этот пример. Сборка является стандартной сборкой CMake. Мы выполняем ее на четырех процессорах:

```
mkdir build && cd build
cmake ..
make
mpirun -n 4 ./hdf5block2d
```

При одинарной инсталляции пакет HDF5 может быть инсталлирован либо как параллельная, либо как последовательная версия, но не одновременно. Часто встречающаяся проблема состоит в связывании неправильной версии с вашим приложением. Мы добавили немного специального исходного кода в систему сборки CMake, чтобы предпочтительно выбирать параллельную версию, как показано в следующем ниже листинге. Тогда программа завершается ошибкой, если версия HDF5 не параллельная, избавляя нас от получения этой ошибки во время сборки.

Листинг 16.9 Проверка наличия параллельного пакета HDF5

HDF5Examples/hdf5block2d/CMakeLists.txt

```
14 set(HDF5_PREFER_PARALLEL true)
15 find_package(HDF5 1.10.1 REQUIRED)
16 if (NOT HDF5_IS_PARALLEL)
17    message(FATAL_ERROR " -- Версия HDF5 не является параллельной.")
18 endif (NOT HDF5_IS_PARALLEL)
```

Пример исходного кода выполняет проверочный тест, чтобы убедиться, что прочитанные из файла данные совпадают с данными, с которых мы начали. Мы также можем использовать утилиту h5dump для печати данных в файле. Вы можете использовать следующую ниже команду, чтобы просматривать свой файл данных. На рис. 16.6 показан результат на выходе из приведенной ниже команды.

```
h5dump -y example.hdf5
```

16.5 Другие пакеты программно-информационного обеспечения для параллельных файлов

В этом разделе мы кратко рассмотрим несколько наиболее распространенных пакетов программно-информационного обеспечения для мани-

пулирования параллельными файлами: PnetCDF и ADIOS. PnetCDF, сокращенно от Parallel Network Common Data Form (общая форма данных параллельной сети), является еще одним самоописывающим форматом данных, который популярен в сообществе систем Земли и среди организаций, финансируемых Национальным научным фондом (National Science Foundation, NSF). Изначально являясь совершенно отдельным источником программно-информационного обеспечения, его параллельная версия построена поверх HDF5 и MPI-IO. На решение о том, что именно использовать, PnetCDF или HDF5, сильно влияет ваше сообщество. Поскольку файлы, создаваемые вашим приложением, часто используются другими пользователями, важно использовать один и тот же стандарт данных.

```
HDF5 "example.hdf5" {
GROUP "/" {
   DATASET "data array" {
      DATATYPE  H5T_IEEE_F64LE
      DATASPACE  SIMPLE { ( 10, 40 ) / ( 10, 40 ) }
      DATA {
         1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 101, 102, 103, 104, 105, 106, 107, 108,
         109, 110, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 301, 302,
         303, 304, 305, 306, 307, 308, 309, 310,
         11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 111, 112, 113, 114, 115, 116,
         117, 118, 119, 120, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220,
         311, 312, 313, 314, 315, 316, 317, 318, 319, 320,
         21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 121, 122, 123, 124, 125, 126,
         127, 128, 129, 130, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230,
         321, 322, 323, 324, 325, 326, 327, 328, 329, 330,
         31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 131, 132, 133, 134, 135, 136,
         137, 138, 139, 140, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240,
         331, 332, 333, 334, 335, 336, 337, 338, 339, 340,
         41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 141, 142, 143, 144, 145, 146,
         147, 148, 149, 150, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250,
         341, 342, 343, 344, 345, 346, 347, 348, 349, 350,
         51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 151, 152, 153, 154, 155, 156,
         157, 158, 159, 160, 251, 252, 253, 254, 255, 256, 257, 258, 259, 260,
         351, 352, 353, 354, 355, 356, 357, 358, 359, 360,
         61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 161, 162, 163, 164, 165, 166,
         167, 168, 169, 170, 261, 262, 263, 264, 265, 266, 267, 268, 269, 270,
         361, 362, 363, 364, 365, 366, 367, 368, 369, 370,
         71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 171, 172, 173, 174, 175, 176,
         177, 178, 179, 180, 271, 272, 273, 274, 275, 276, 277, 278, 279, 280,
         371, 372, 373, 374, 375, 376, 377, 378, 379, 380,
         81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 181, 182, 183, 184, 185, 186,
         187, 188, 189, 190, 281, 282, 283, 284, 285, 286, 287, 288, 289, 290,
         381, 382, 383, 384, 385, 386, 387, 388, 389, 390,
         91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 191, 192, 193, 194, 195, 196,
         197, 198, 199, 200, 291, 292, 293, 294, 295, 296, 297, 298, 299, 300,
         391, 392, 393, 394, 395, 396, 397, 398, 399, 400
      }
   }
}
}
```

Рис. 16.6 Использование утилиты командной строки h5dump также показывает содержимое файла HDF5 без необходимости писать какой-либо исходный код

ADIOS, или Adaptable Input/Output System (адаптируемая система ввода-вывода), также является самоописывающим форматом данных Национальной лаборатории Оук Риджа (ORNL). ADIOS имеет свой собственный двоичный формат, но в нем также может использоваться HDF5, MPI-IO и другое программно-информационное обеспечение для хранения файлов.

16.6 Параллельная файловая система: аппаратный интерфейс

С ростом потребностей в данных становятся необходимыми более сложные файловые системы. В настоящем разделе мы представим эти параллельные файловые системы. Параллельная файловая система может значительно ускорять запись и чтение файлов, распределяя операции между несколькими жесткими дисками с помощью нескольких программ записи или чтения файлов. Хотя теперь у нас есть некоторый параллелизм в файловой системе, все же это непростая ситуация. По-прежнему существует несоответствие между параллелизмом приложений и параллелизмом, предоставляемым файловой системой. По этой причине управление параллельными операциями является сложным и сильно зависит от аппаратной платформы и требований приложений. Для того чтобы справиться со сложностью, многие параллельные файловые системы используют объектно ориентированную файловую структуру. Объектно ориентированные файловые системы естественным образом подходят для решения такого рода задач. Но производительность и устойчивость параллельной файловой системы часто лимитирована метаданными, описывающими местоположение файловых данных.

> **ОПРЕДЕЛЕНИЕ** *Объектно ориентированная файловая система* – это система, организованная на основе объектов, а не файлов в папке. Объектно ориентированной файловой системе требуется база данных или метаданные для хранения всей информации, описывающей объект.

Написание параллельных файловых операций тесно связано с программно-информационным обеспечением параллельной файловой системы. Оно требует знания о том, какая параллельная файловая система используется, и о существующих настройках для этой инсталляции и файловой системы. Точная настройка программно-информационного обеспечения для параллельных файлов иногда может приводить к значительному повышению производительности.

16.6.1 Все, что вы хотели знать о настройке параллельного файла, но не знали, как спросить

Когда вы окунетесь во взаимодействие параллельных файловых операций с файловой системой, полезно иметь дополнительную информацию

о настройках параллельной библиотеки. Настройки могут задаваться поразному для каждой инсталляции. Вы также можете получить некоторые статистические данные высокого уровня, которые помогут при отлаживании проблем с производительностью.

Большинство библиотек MPI-IO является одной из двух имплементаций, ROMIO, которая распространяется с MPICH и многими системными имплементациями поставщиков, либо OMPIO, которая используется по умолчанию в более новых версиях OpenMPI. Давайте сначала пройдемся по тому, как получать информацию из плагина OpenMPI OMPIO либо как возвращаться к использованию ROMIO. В целях извлечения информации о настройках OMPIO OpenMPI следует использовать следующие ниже команды:

- `-- mca io [ompio|romio]`.
  Задает плагин ввода-вывода, OMPIO либо ROMIO. В более старых версиях в качестве плагина по умолчанию используется ROMIO, тогда как в более новых версиях по умолчанию используется OMPIO;
- `ompi_info --param <компонент> <плагин> --level <int>`.
  Выводит на экран информацию о локальной конфигурации OpenMPI для этого плагина;
- `--mca io_ompio_verbose_info_parsing 1`.
  Показывает подсказки, разобранные из вызовов команды `MPI_Info_set` программы.

Прежде всего вы можете получать имена плагинов ввода-вывода с помощью команды `ompi_info`. Нам нужны лишь плагины компонентов ввода-вывода, поэтому вывод для них мы фильтруем:

```
ompi_info |grep "MCA io:"
MCA io: romio321 (MCA v2.1.0, API v2.0.0, Component v4.0.3)
MCA io: ompio (MCA v2.1.0, API v2.0.0, Component v4.0.3)
```

Затем вы можете получать индивидуальные настройки, которые доступны для каждого плагина. Используя команду `ompi_info`, мы получаем следующий ниже сокращенный результат:

```
ompi_info --param io ompio --level 9 | grep ": parameter"
MCA io ompio: parameter "io_ompio_priority" (current value: "30" …
MCA io ompio: parameter "io_ompio_delete_priority" (current value: "30" …
MCA io ompio: parameter "io_ompio_record_file_offset_info" (current value: "0" …
MCA io ompio: parameter "io_ompio_coll_timing_info" (current value: "1" …
MCA io ompio: parameter "io_ompio_cycle_buffer_size" (current value: "536870912" …
MCA io ompio: parameter "io_ompio_bytes_per_agg" (current value: "33554432" …
MCA io ompio: parameter "io_ompio_num_aggregators" (current value: "-1" …
MCA io ompio: parameter "io_ompio_grouping_option" (current value: "5" …
MCA io ompio: parameter "io_ompio_max_aggregators_ratio" (current value: "8" …
MCA io ompio: parameter "io_ompio_aggregators_cutoff_threshold" (current value: "3"
...
MCA io ompio: parameter "io_ompio_overwrite_amode" (current value: "1" …
MCA io ompio: parameter "io_ompio_verbose_info_parsing" (current value: "0"
...
```

Вы также можете верифицировать то, как вызовы команды `MPI_Info_set` интерпретируются библиотекой MPI-IO с помощью следующей ниже опции во время выполнения. Это бывает неплохим подспорьем в проверке правильности написанного исходного кода для вашей файловой системы и библиотек параллельных файловых операций.

```
mpirun --mca io_ompio_verbose_info_parsing 1 -n 4 ./mpi_io_block2d
File: example.data info: collective_buffering value true enforcing using
individual fcoll component
< ... повторется еще три раза ... >
```

В программно-информационном обеспечении для параллельных файлов ROMIO, входящем в состав MPICH, у нас есть разные механизмы для опрашивания инсталляции программно-информационного обеспечения. Cray вносит для своих имплементаций ROMIO несколько дополнительных средовых переменных. Мы перечислим некоторые из них, а затем посмотрим на примеры, в которых они используются.

- ROMIO распознает следующую ниже подсказку:
  - `ROMIO_PRINT_HINTS=1`.
- Cray предлагает следующие ниже дополнительные переменные среды:
  - `MPICH_MPIIO_HINTS_DISPLAY=1`;
  - `MPICH_MPIIO_STATS=1`;
  - `MPICH_MPIIO_TIMERS=1`.

Ниже показан результат при использовании `ROMIO_PRINT_HINTS`:

```
export ROMIO_PRINT_HINTS=1; mpirun -n 4 ./mpi_io_block2d
key = cb_buffer_size            value = 16777216
key = romio_cb_read             value = automatic
key = romio_cb_write            value = automatic
key = cb_nodes                  value = 1
key = romio_no_indep_rw         value = false
key = romio_cb_pfr              value = disable
key = romio_cb_fr_types         value = aar
key = romio_cb_fr_alignment     value = 1
key = romio_cb_ds_threshold     value = 0
key = romio_cb_alltoall         value = automatic
key = ind_rd_buffer_size        value = 4194304
key = ind_wr_buffer_size        value = 524288
key = romio_ds_read             value = automatic
key = romio_ds_write            value = automatic
key = striping_unit             value = 4194304
key = cb_config_list            value = *:1
key = romio_filesystem_type     value = NFS:
key = romio_aggregator_list     value = 0
key = cb_buffer_size            value = 16777216
key = romio_cb_read             value = automatic
key = romio_cb_write            value = automatic
key = cb_nodes                  value = 1
key = romio_no_indep_rw         value = false
key = romio_cb_pfr              value = disable
```

```
key = romio_cb_fr_types         value = aar
key = romio_cb_fr_alignment     value = 1
key = romio_cb_ds_threshold     value = 0
key = romio_cb_alltoall         value = automatic
key = ind_rd_buffer_size        value = 4194304
key = ind_wr_buffer_size        value = 524288
key = romio_ds_read             value = automatic
key = romio_ds_write            value = automatic
key = cb_config_list            value = *:1
key = romio_filesystem_type     value = NFS:
key = romio_aggregator_list     value = 0

export MPICH_MPIIO_HINTS_DISPLAY=1; srun -n 4 ./mpi_io_block2d
PE 0: MPICH MPIIO environment settings:
PE 0:   MPICH_MPIIO_HINTS_DISPLAY                = 1
PE 0:   MPICH_MPIIO_HINTS                        = NULL
PE 0:   MPICH_MPIIO_ABORT_ON_RW_ERROR            = disable
PE 0:   MPICH_MPIIO_CB_ALIGN                     = 2
PE 0:   MPICH_MPIIO_DVS_MAXNODES                 = -1
PE 0:   MPICH_MPIIO_AGGREGATOR_PLACEMENT_DISPLAY  = 0
PE 0:   MPICH_MPIIO_AGGREGATOR_PLACEMENT_STRIDE   = -1
PE 0:   MPICH_MPIIO_MAX_NUM_IRECV                = 50
PE 0:   MPICH_MPIIO_MAX_NUM_ISEND                = 50
PE 0:   MPICH_MPIIO_MAX_SIZE_ISEND               = 10485760
PE 0: MPICH MPIIO statistics environment settings:
PE 0:   MPICH_MPIIO_STATS                        = 0
PE 0:   MPICH_MPIIO_TIMERS                       = 0
PE 0:   MPICH_MPIIO_WRITE_EXIT_BARRIER           = 1
MPIIO WARNING: DVS stripe width of 8 was requested but DVS set it to 1
See MPICH_MPIIO_DVS_MAXNODES in the intro_mpi man page.
PE 0:   MPIIO hints for example.data:
            cb_buffer_size          = 16777216
            romio_cb_read           = automatic
            romio_cb_write          = automatic
            cb_nodes                = 1
            cb_align                = 2
            romio_no_indep_rw       = false
            romio_cb_pfr            = disable
            romio_cb_fr_types       = aar
            romio_cb_fr_alignment   = 1
            romio_cb_ds_threshold   = 0
            romio_cb_alltoall       = automatic
            ind_rd_buffer_size      = 4194304
            ind_wr_buffer_size      = 524288
            romio_ds_read           = disable
            romio_ds_write          = automatic
            striping_factor         = 1
            striping_unit           = 4194304
            direct_io               = false
            aggregator_placement_stride = -1
            abort_on_rw_error       = disable
            cb_config_list          = *:*
            romio_filesystem_type   = CRAY ADIO:
```

```
export MPICH_MPIIO_STATS=1; srun -n 4 ./mpi_io_block2d
+-------------------------------------------------------+
| MPIIO write access patterns for example.data
| independent writes      = 0
| collective writes       = 4
| independent writers     = 0
| aggregators             = 1
| stripe count            = 1
| stripe size             = 4194304
| system writes           = 2
| stripe sized writes     = 0
| aggregators active      = 4,0,0,0 (1, <= 1, > 1, 1)
| total bytes for writes  = 3600
| ave system write size   = 1800
| read-modify-write count = 0
| read-modify-write bytes = 0
| number of write gaps    = 0
| ave write gap size      = NA
| See "Optimizing MPI I/O on Cray XE Systems" S-0013-20 for explanations.
+-------------------------------------------------------+
+-------------------------------------------------------+
| MPIIO read access patterns for example.data
| independent reads         = 0
| collective reads          = 4
| independent readers       = 0
| aggregators               = 1
| stripe count              = 1
| stripe size               = 524288
| system reads              = 1
| stripe sized reads        = 0
| total bytes for reads     = 3200
| ave system read size      = 3200
| number of read gaps       = 0
| ave read gap size         = NA
| See "Optimizing MPI I/O on Cray XE Systems" S-0013-20 for explanations.
+-------------------------------------------------------+
```

16.6.2 Общие подсказки, применимые ко всем файловым системам

Иногда полезно давать некоторые подсказки о типе файловых операций, которые вы будете использовать в своем приложении. Вы можете модифицировать настройки параллельного файла с помощью средовых переменных, файла подсказок либо во время выполнения с помощью команды `MPI_Info_set`. За счет этого обеспечивается надлежащий метод манипулирования разными сценариями, если у вас нет доступа к исходному коду программы, чтобы добавить команду `MPI_Info_set`. В целях задания опций параллельного файла в этом случае следует использовать следующие ниже команды.

- Cray MPICH.

```
MPICH_MPIIO_HINTS="*:<ключ>=<значение>:<ключ>=<значение>.
```

Например:

```
export MPICH_MPIIO_HINTS=\
   "*:striping_factor=8:striping_unit=4194304"
```

- ROMIO.

```
ROMIO_HINTS=<имя файла>.
```

Например: `ROMIO_HINTS=romio-hints`,
где файл romio-hints содержит:

```
striping_factor 8      // файл разбит на 8 частей и
                       // записывается параллельно на 8 дисков
striping_unit 4194304  // размер в байтах каждого
                       // записываемого блока
```

- OpenMPI OMPI.

```
OMPI_MCA_<имя_параметра> <значение>
```

Например: `export OMPI_MCA_io_ompio_verbose_info_parsing=1`.

Опция OpenMPI mca времени выполнения в качестве аргумента команды mpirun такова:

```
mpirun --mca io_ompio_verbose_info_parsing 1 -n 4 <exec>
```

Местоположение файла OpenMPI по умолчанию находится в файле $HOME/.openmpi/mca-params.conf, либо его можно задать следующим образом:

```
--tune <имя_файла>
mpirun --tune mca-params.conf -n 2 <exec>
```

Самая важная подсказка, которую можно задавать, состоит в указании того, что конкретно использовать: коллективные операции либо просеивание данных. Сначала мы рассмотрим коллективные операции, а затем операции просеивания данных.

В коллективных операциях используются вызовы MPI коллективного обмена и подход на основе двухфазного ввода-вывода, который собирает данные для агрегаторов, которые затем пишут или читают из вашего файла. Следует использовать следующие ниже команды для коллективного ввода-вывода.

- ROMIO и OMPIO.
  - `cb_buffer_size=целое_число` определяет размер буфера в байтах для двухфазного коллективного ввода-вывода. Он должен быть кратен размеру страницы.
  - `cb_nodes=целое_число` задает максимальное число агрегаторов.

- Только ROMIO.
  - `romio_cb_read=[enable|automatic|disable]` указывает на то, когда следует использовать коллективную буферизацию для операций чтения.
  - `romio_cb_write=[enable|automatic|disable]` указывает на то, когда следует использовать коллективную буферизацию для операций записи.
  - `cb_config_list=*:<целое_число>` задает число агрегаторов в расчете на узел.
  - `romio_no_indep_rw=[true|false]` указывает на то, следует или нет использовать какой-либо независимый ввод-вывод. Если они не разрешены, то никакие файловые операции (включая открытие файла) не будут выполняться на узлах, не являющихся агрегаторами.
- Только OMPIO.
  - `collective_buffering=[true|false]` использует коллективные операции при записи из параллельного задания в файловую систему.

Просеивание (sieving) данных выполняет однократное чтение (или запись), охватывающее файловый блок, а затем передает данные отдельным процессам чтения. Это позволяет избегать большого числа малых операций чтения и могущего возникнуть соперничества между читателями файлов. Следует использовать следующие ниже команды для просеивания данных с помощью ROMIO:

- `romio_ds_read=[enable|automatic|disable]`;
- `romio_ds_write=[enable|automatic|disable]`;
- `ind_rd_buffer_size=целое_число` (байты для буфера чтения);
- `ind_wr_buffer_size=целое_число` (байты для буфера записи).

16.6.3 Подсказки, относящиеся к конкретным файловым системам

Некоторые подсказки применимы только к конкретной файловой системе, такой как Lustre или GPFS. Мы можем определять тип файловой системы из нашей программы и задавать соответствующие подсказки для файловой системы. Программа fs_detect.c в примерах как раз это и делает. В указанной программе используется команда `statfs`, как показано в следующем ниже листинге, и вы можете найти ее в каталоге примеров этой главы.

Листинг 16.10 Программа обнаружения файловой системы

`MPI_IO_Examples/mpi_io_block2d/fs_detect.c`

```
1 #include <stdio.h>
2 #ifdef __APPLE_CC__
3 #include <sys/mount.h>
```

```
 4 #else
 5 #include <sys/statfs.h>
 6 #endif
 7 // Типы файловой системы перечислены в системном
   // каталоге вложений в linux/magic.h
 8 // Вам нужно будет добавить любые дополнительные
   // волшебные коды параллельной файловой системы
 9 #define LUSTRE_MAGIC1     0x858458f6
10 #define LUSTRE_MAGIC2     0xbd00bd0
11 #define GPFS_SUPER_MAGIC  0x47504653
12 #define PVFS2_SUPER_MAGIC 0x20030528
13 #define PAN_KERNEL_FS_CLIENT_SUPER_MAGIC \
                             0xAAD7AAEA
14
15 int main(int argc, char *argv[])
16 {
17    struct statfs buf;
18    statfs("./fs_detect", &buf);
19    printf("Тип файловой системы: %lx\n",buf.f_type);
20 }
```

Волшебные числа типа параллельной файловой системы

Получает тип файловой системы

В состав этого листинга включено волшебное число для некоторых параллельных файловых систем. При его использовании для других приложений следует заменить имя файла в строке 18 соответствующим именем файла с учетом каталога, в котором записаны ваши файлы. Соберите программу `fs_detect`, а затем выполните следующую ниже команду, чтобы получить тип файловой системы:

```
mkdir build && cd build
cmake ..
make
grep `./fs_detect | cut -f 4 -d' '` /usr/include/linux/magic.h ../fs_detect.c
```

Теперь мы готовы к подсказкам, касающимся файловой системы. Мы не перечисляем все возможные подсказки. Вы можете получить текущий список, используя команды, которые были показаны ранее.

Файловая система Lustre: наиболее распространенная файловая система в центрах высокопроизводительных вычислений

Lustre является доминирующей файловой системой в крупнейших системах высокопроизводительных вычислений. Зародившись в Университете Карнеги–Меллон, ее главенствующая разработка и владение ею были переданы Intel, HP, Sun, Oracle, Intel, Whamcloud и др. В ходе этого процесса она переходила от коммерческого к открытому исходному коду и обратно. В настоящее время она находится под эгидой Открытых масштабируемых файловых систем (Open Scalable File Systems, OpenSFS) и Европейских открытых файловых систем (European Open File Systems, EOFS).

Файловая система Lustre выстроена на концепции объектного хранения с использованием Серверов хранения объектов (Object Storage Serv-

ers, OSS) и Целей хранения объектов (Object Storage Target, OST). Когда мы указываем подсказку `striping_factor` 8 в строке 56 листинга 16.2 и строке 96 листинга 16.6, мы говорим библиотеке ROMIO использовать Lustre для подразделения операций записи (и чтения) на восемь частей и отправки их в восемь операционных систем, эффективно записывая данные в восьмипутном параллелизме. Подсказка `striping_unit` указывает ROMIO и Lustre использовать полосы размером 4 Мб. В Lustre также есть Серверы метаданных (Metadata Server, MDS) и Цели метаданных (Metadata Target, MDT) для хранения важных описаний мест, где хранится каждая часть файла. Для операций подразделения на полосы следует использовать следующее ниже.

- MPICH (ROMIO).
  - `striping_unit=<целое_число>` задает размер полосы в байтах.
  - `striping_factor=<целое_число>` задает число полос, где -1 обозначает задание числа автоматически.
- OpenMPI (OMPIO).
  - `fs_lustre_stripe_size=<целое_число>` задает размер полосы в байтах.
  - `fs_lustre_stripe_width=<целое_число>` задает число полос, где -1 обозначает задание числа автоматически.

Мы можем подтвердить параметры Lustre для OpenMPI с помощью консольного запроса в командной строке:

```
ompi_info --param fs lustre --level 9
MCA fs lustre: parameter "fs_lustre_priority" (current value: "20" …
MCA fs lustre: parameter "fs_lustre_stripe_size" (current value: "0" …
MCA fs lustre: parameter "fs_lustre_stripe_width" (current value: "0" …
```

GPFS: файловая система от IBM

Системы IBM имеют Общую параллельную файловую систему (General Parallel File System, GPFS), также являющуюся частью их продукта Spectrum Scale (Спектральная шкала), которая предлагает операции подразделения на полосы и параллельные файловые операции в своих системах. GPFS является продуктом хранения корпоративных данных с соответствующей инфраструктурой поддержки и службами. По умолчанию GPFS задает полосы по всем имеющимся устройствам. Однако подсказки MPI не так сильно влияют на эту файловую систему. Для MPICH (ROMIO) следует использовать приведенную ниже команду, чтобы помочь с операциями записи/чтения большого объема памяти:

```
IBM_largeblock_io=true
```

DataWarp: файловая система от Cray

DataWarp от Cray интегрирует оборудование всплескового буфера поверх еще одной параллельной файловой системы, такой как их версия Lustre. Однако использование преимуществ всплесковых буферов все еще находится в зачаточном состоянии, но Cray остается лидером в этих усилиях.

Panasas®: коммерческая файловая система, требующая меньше подсказок от пользователей

Panasas® – это коммерческая параллельная файловая система, состоящая из хранилища объектов и серверов метаданных. Panasas также внесла свой вклад в расширение поддержки параллельных операций на Сетевую файловую систему (Network File System, NFS). Panasas использовалась в нескольких из десяти лучших вычислительных систем LANL, хотя сегодня она там не столь превалирует. Для MPICH (ROMIO) следует использовать приведенные ниже команды, чтобы задавать соответственно размер полосы и число полос:

- `panfs_layout_stripe_unit=<целое_число>`;
- `panfs_layout_total_num_comps=<целое_число>`.

OrangeFS (PVFS): самая популярная файловая система с открытым исходным кодом

OrangeFS, ранее известная как Параллельная виртуальная файловая система (Parallel Virtual File System, PVFS), представляет собой параллельную файловую систему с открытым исходным кодом из Университета Клемсона и Аргоннской национальной лаборатории. Она популярна в кластерах Beowulf. Помимо того, что OrangeFS является масштабируемой параллельной файловой системой, она была интегрирована в ядро Linux. Следующие ниже команды можно использовать для MPICH (ROMIO), чтобы соответственно задавать размер полосы (в байтах) и нумеровать полосы (где -1 автоматически):

- `striping_unit=<целое_число>`;
- `striping_factor=<целое_число>`.

BeeGFS: новая набирающая популярность файловая система с открытым исходным кодом

BeeGFS, ранее FhGFS, была разработана в Центре высокопроизводительных вычислений Фраунгофера и находится в свободном доступе. Она популярна по причине своих характеристик, связанных с открытым исходным кодом.

Распределенное хранение объектов приложений (DAOS): установка новых сравнительных эталонов производительности

Intel разрабатывает свою новую технологию хранения объектов DAOS с открытым исходным кодом в рамках программы FastForward Министерства энергетики (Department of Energy, DOE). DAOS (Distributed Application Object Storage) занимает первое место в списке файловых скоростей для суперкомпьютеров ISC IO500 2020 года (https://www.vi4io.org). Ее планируется развернуть в 2021 году на суперкомпьютере Aurora, первой вычислительной экзомасштабной системе Аргоннской национальной лаборатории. DAOS поддерживается библиотекой MPI-IO в ROMIO, доступна с MPICH и переносима в другие библиотеки MPI.

WekaIO: новичок в сообществе больших данных

WekaIO – это полностью совместимая с POSIX файловая система, которая предоставляет большое общее пространство имен с высоко оптимизированной производительностью, низкой задержкой и высокой пропускной способностью, а также использует новейшие твердотельные аппаратные компоненты. Файловая система WekaIO привлекательна для приложений, требующих больших объемов высокопроизводительных операций с файлами данных, и она популярна в сообществе больших данных. WekaIO заняла первое место в списке скоростей суперкомпьютерных файлов SC IO500 2019 года.

Файловая система Серн: распределенная система хранения с открытым исходным кодом

Ceph возникла в Национальной лаборатории Лоуренса Ливермора. В настоящее время разработкой руководит RedHat для консорциума промышленных партнеров, и она интегрирована в ядро Linux.

Сетевая файловая система (NFS): наиболее распространенная сетевая файловая система

NFS является доминирующей кластерной файловой системой для сетей в локальных организациях. Указанная система не рекомендуется для высокопараллельных файловых операций, хотя при соответствующих настройках она функционирует правильно.

16.7 Материалы для дальнейшего исследования

Большая часть текущей документации по параллельным файловым операциям содержится в презентациях и научных конференциях. Одной из лучших конференций является Семинар по параллельным системам обработки данных (Parallel Data Systems Workshop, PDSW), проводимый совместно с Международной конференцией по высокопроизводительным вычислениям, сетям, хранению и анализу (по-другому именуемой Ежегодной конференцией по суперкомпьютерам).

Существует возможность использовать микротесты, IOR и mdtest, чтобы проверять наилучшую производительность файловой системы. Программно-информационное обеспечение документировано по адресу https://ior.readthedocs.io/en/latest/ и размещается на LLNL по адресу https://github.com/hpc/ior.

16.7.1 Дополнительное чтение

Добавление функций MPI-IO в MPI описано в следующем ниже тексте. Он остается одним из лучших описаний MPI-IO.

- Уильям Гропп, Раджив Тхакур и Юинг Ласк, «Использование MPI-2: расширенные функциональности Интерфейса передачи сообщений» (William Gropp, Rajeev Thakur, Ewing Lusk. *Using MPI-2: Advanced Features of the Message Passing Interface*, MIT Press, 1999).

Есть пара хороших книг по написанию высокопроизводительных параллельных файловых операций. Мы рекомендуем следующие:

- Прабхат и Квинси Козиол, редакторы, «Высокопроизводительный параллельный ввод-вывод» (Prabhat and Quincey Koziol, editors, *High Performance Parallel I/O*, Chapman and Hall/CRC, 2014);
- Джон М. Мэй, «Параллельный ввод-вывод для высокопроизводительных вычислений» (John M. May, *Parallel I/O for High Performance Computing*, Morgan Kaufmann, 2001).

Группа HDF ведет авторитетный веб-сайт по HDF5. Более подробную информацию можно получить по адресу:

- группа HDF, https://portal.hdfgroup.org/display/HDF5/HDF5.

NetCDF остается популярной внутри определенных сегментов приложений HPC. Более подробную информацию об этом формате можно получить на веб-сайте NetCDF, размещенном на Unidata. Unidata является одной из общественных программ Университетской корпорации атмосферных исследований (University Corporation for Atmospheric Research, UCAR) (UCP).

- Unidata, https://www.unidata.ucar.edu/software/netcdf/.

Параллельная версия системы NetCDF, PnetCDF, была разработана Северо-Западным университетом и Аргоннской национальной лабораторией независимо от Unidata. Более подробная информация о PnetCDF находится на их документационном веб-сайте GitHub:

- Северо-Западный университет и Аргоннская национальная лаборатория, https://parallel-netcdf.github.io.

ADIOS – это одна из ведущих библиотек параллельных файловых операций, поддерживаемых коллективом, возглавляемым Национальной лабораторией Оук Риджа (Oak Ridge National Laboratory, ORNL). Для того чтобы узнать больше, ознакомьтесь с их документацией на следующем ниже веб-сайте:

- Национальная лаборатория Оук Риджа, https://adios2.readthedocs.io/en/latest/index.html.

Несколько неплохих презентаций по настройке производительности файловых систем:

- Филипп Вотле, «Лучшие практики для параллельного ввода-вывода и подсказок MPI-IO» (Philippe Wautelet, Best practices for parallel IO and MPI-IO hints, CRNS/IDRIS, 2015) по адресу http://www.idris.fr/media/docs/docu/idris/idris_patc_hints_proj.pdf;
- Джордж Маркоманолис, «Спектральная шкала ORNL» (George Markomanolis, ORNL Spectrum Scale, GPFS) по адресу https://www.olcf.ornl.gov/wp-content/uploads/2018/12/spectrum_scale_summit_workshop.pdf.

16.7.2 Упражнения

1. Проверьте наличие доступных в вашей системе подсказок, используя технические приемы, описанные в разделе 16.6.1.
2. Попробуйте примеры MPI-IO и HDF5 в вашей системе с гораздо более крупными наборами данных, чтобы увидеть производительность, которую вы сможете достичь. Сравните это с микроэталоном IOR для получения дополнительного кредита.
3. Используйте утилиты h5ls и h5dump для проведения разведывательного анализа файла данных HDF5, созданного в примере HDF5.

Резюме

- Существует соответствующий способ манипулирования стандартными файловыми операциями в параллельных приложениях. Простых технических приемов, описанных в этой главе, где весь ввод-вывод выполняется с первого процессора, достаточно для скромных параллельных приложений.
- Использование MPI-IO является важным строительным блоком для параллельных файловых операций. MPI-IO может значительно ускорять запись и чтение файлов.
- Использование параллельного программно-информационного обеспечения самоописывающего формата HDF5 имеет ряд преимуществ. Формат HDF5 может улучшать управление данными в вашем приложении, а также ускорять файловые операции.
- Существуют способы опрашивания и задания подсказок для программно-информационного обеспечения манипулирования параллельными файлами и файловой системой. Они повышают производительность записи и чтения файлов в конкретных системах.

17 Инструменты и ресурсы для более качественного исходного кода

Эта глава охватывает следующие ниже темы:

- потенциальные инструменты для вашего пояса инструментов разработки;
- различные ресурсы для руководства разработкой вашего приложения;
- распространенные инструменты для работы на крупных вычислительных площадках.

Зачем нужна целая глава об инструментах и ресурсах? Хотя мы упоминали инструменты и ресурсы в предыдущих главах, в этой главе далее обсуждается широкий спектр и альтернативы, доступные программистам, занимающимся высокопроизводительными вычислениями. От систем версионного контроля до отлаживания исходного кода предлагаемые возможности, будь то коммерческие либо с открытым исходным кодом, необходимы для обеспечения быстрой итерации по параллельной разработке приложений. Тем не менее эти инструменты не являются обязательными. Понимание и внедрение их в свой рабочий поток нередко дает огромные преимущества, значительно превышающие время, затрачиваемое на усвоение того, как их использовать.

Инструменты являются важной частью процесса разработки приложений высокопроизводительных вычислений. Не каждый инструмент работает в каждой системе, поэтому наличие альтернатив имеет большую важность. В предыдущих главах мы хотели сосредоточиться на процессе, чтобы не увязнуть в деталях того, как использовать все возможные инструменты. Мы решили показывать самый простой и доступный инструмент для каждой потребности. Мы также предпочли командную строку и тексто-ориентированные инструменты модным инструментам на основе графического интерфейса, потому что использовать графические интерфейсы в медленных сетях бывает трудно или даже невозможно. Графические инструменты также, как правило, более ориентированы на поставщика либо систему и часто изменяются. Несмотря на эти недостатки, мы включаем многие из этих инструментов поставщиков в эту главу, потому что они могут значительно улучшать разработку исходного кода для приложений высокопроизводительных вычислений.

Ресурсы, такие как широкий спектр приложений сравнительного анализа производительности, ценны тем, что приложения не бывают только одного вида. Для этих специализированных областей применения нам нужны более подходящие сравнительные тесты и мини-приложения, которые проводят разведку наилучшего подхода к разработке алгоритмов и правильного для каждой архитектуры шаблона программирования. Мы настоятельно рекомендуем вам учиться на этих ресурсах, а не изобретать новые технические приемы с нуля. По большинству инструментов мы даем краткие инструкции по инсталляции и информацию о том, где найти соответствующую документацию. Мы также предоставляем более подробную информацию в прилагаемом к этой главе исходном коде по адресу https://github.com/EssentialsofParallelComputing/Chapter17.

Мы занимаем строго независимую от поставщиков позицию, а также особо подчеркиваем переносимость. Хотя мы охватываем большое число инструментов, просто невозможно подробно остановиться на всех из них. Кроме того, скорость изменения этих инструментов превышает скорость изменения остальной части экосистемы высокопроизводительных вычислений. История показывает, что поддержка хорошей разработки инструментов переменчива. И значит, инструменты приходят и уходят и меняют владельца быстрее, чем успевает обновляться документация.

В целях краткого ознакомления в табл. 17.1 приводится сводка инструментов, которые мы охватим в этой главе. Они показаны в соответствующих категориях, чтобы помочь вам найти наилучшие инструменты для ваших потребностей. Мы включили широкий спектр инструментов, потому что может существовать только один, который работает на том или ином оборудовании или в операционной системе или может обладать специализированными способностями. В следующих далее разделах этой главы, как указано в таблице, мы решили дать более подробную информацию о некоторых простых, полезных и часто используемых инструментах.

Таблица 17.1 Сводка инструментов, описанных в этой главе

17.1. Системы версионного контроля	17.1.1. Централизованный версионный контроль	Subversion CVS
	17.1.2. Распределенный версионный контроль	Git Mercurial

17.2. Таймерные процедуры	`clock_gettime` (с типом `CLOCK_MONOTONIC`)
	`clock_gettime` (с типом `CLOCK_REALTIME`)
	`gettimeofday`
	`getrusage`
	`host_get_clock_service` (для часов MacOS C++ высокой разрешающей способности)	

17.3. Профилировщики	17.3.1. Простые текстовые профилировщики	Likwid gprof gperftools timemory Open|SpeedShop
	17.3.2. Профилировщики высокого уровня	Kcachegrind Arm MAP
	17.3.3. Профилировщики среднего уровня	Intel® Advisor Intel® Vtune CrayPat AMD μProf NVIDIA Visual Profiler CodeXL
	17.3.4. Детализированные профилировщики	HPCToolkit Open|SpeedShop TAU
17.5. Инструменты обработки ошибок памяти	Бесплатно программно-информационное обеспечение	17.5.1 Valgrind 17.5.2 Dr. Memory
	Коммерческое программно-информационное обеспечение	Purify Insure++ Intel® Inspector Инструмент проверки памяти TotalView
	17.5.4. Компиляторно-ориентированные	MemorySanitizer (LLVM) AddressSanitizer (LLVM) ThreadSanitizer (LLVM) mtrace (GCC)
	17.5.5. Инструменты проверки выхода за пределы границ	Dmalloc Electric Fence Memwatch
	17.5.6. Инструменты памяти GPU	CUDA-MEMCHECK
17.6 Инструменты проверки потоков	17.6.1 Intel® Inspector 17.6.2 Archer	
17.7. Отладчики	Коммерческие	17.7.1 TotalView 17.7.2 ARM DDT
	17.7.3. Отладчики Linux	GDB cgdb DDD
	17.7.4. Отладчики GPU	CUDA-GDB ROCgdb
17.8. Профилировщик файловых операций	Darshan	
17.9. Менеджеры пакетов	17.9.1. MacOS	Homebrew MacPorts
	17.9.2. HPC	Spack
17.10. Модули	17.10.1. Модули 17.10.2. Lmod	

17.1 Системы версионного контроля: все начинается здесь

Версионный контроль программно-информационного обеспечения является одной из самых базовых практик разработки программ и критически важен во время разработки параллельных приложений. Мы рассмотрели роль версионного контроля в разработке параллельных приложений в разделе 2.1.1. Здесь мы разберем различные системы версионного контроля и их характеристики подробнее. Системы версионного контроля можно разделить на две главенствующие категории: распределенные и централизованные, как показано на рис. 17.1.

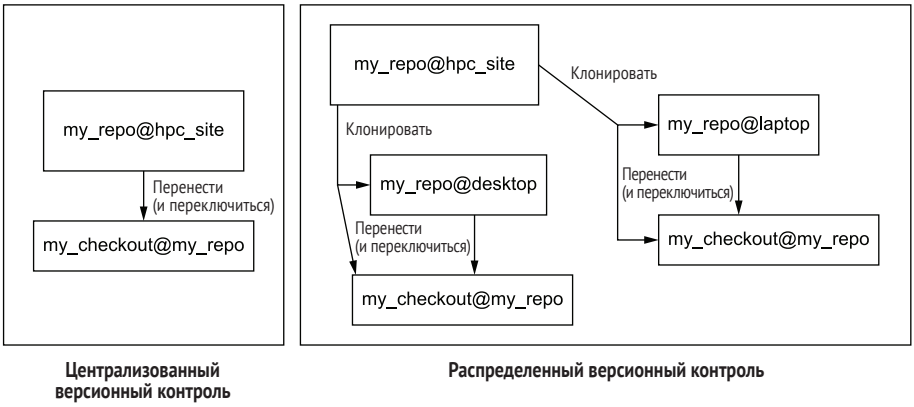


Рис. 17.1 Выбор типа версионного контроля зависит от вашей практики работы. Централизованный версионный контроль предназначен для случаев, когда все находятся в месте с доступом к одному серверу. Распределенный версионный контроль предоставляет вам полную копию вашего репозитория на вашем ноутбуке и настольном компьютере и позволяет перемещаться по всему миру и мобильно

В централизованной системе версионного контроля существует только один центральный репозиторий. Для этого требуется интернет-соединение с веб-сайтом репозитория, чтобы выполнять любые операции с репозиторием. В распределенной системе версионного контроля различные команды, такие как clone, создают дублирующую (дистанционную) версию репозитория и перенос исходного кода. Вы можете фиксировать свои изменения в локальной версии репозитория во время путешествия, а затем переносить или объединять изменения в главный репозиторий позже. Неудивительно, что в последние годы распределенные системы версионного контроля приобрели популярность. Тем не менее они также имеют еще один уровень сложности.

17.1.1 Распределенный версионный контроль подходит для более мобильного мира

Многие коллективы разработчиков исходного кода разбросаны по всему миру либо постоянно находятся в движении. Для них распределенная система версионного контроля имеет наибольший смысл. Двумя наиболее распространенными свободно доступными распределенными системами версионного контроля являются Git и Mercurial. Помимо них, также есть несколько других малых распределенных систем версионного контроля. Все эти имплементации поддерживают различные рабочие потоки разработки.

Несмотря на заявления о том, что они просты в освоении, эти инструменты сложно полностью понять и правильно использовать. Для того чтобы охватить каждый из них, потребовалась бы целая книга. К счастью, существует много веб-руководств и книг, посвященных их использованию. Хорошей отправной точкой для ресурсов Git является веб-сайт Git SCM: https://git-scm.com.

Mercurial немного проще и имеет более чистый дизайн, чем Git. Кроме того, на веб-сайте Mercurial есть много учебных пособий, которые помогут вам начать работу.

- Mercurial по адресу https://www.mercurial-scm.org/wiki/Mercurial.
- Брайан О'Салливан, «Mercurial: окончательное руководство» (Bryan O'Sullivan, *Mercurial: The Definitive Guide*, O'Reilly Media, 2009) по адресу http://hgbook.red-bean.com.

Есть также несколько коммерчески распространяемых систем версионного контроля. Perforce и ClearCase являются наиболее известными. С помощью этих продуктов вы можете получать дополнительную поддержку, что, возможно, будет иметь важность для вашей организации.

17.1.2 Централизованный версионный контроль для простоты и безопасности исходного кода

Несмотря на существование большого числа централизованных систем версионного контроля, которые были разработаны на протяжении долгой истории управления конфигурацией программно-информационного обеспечения, сегодня наиболее часто используются две системы: Система конкурентных версий (Concurrent Versions System, CVS) и Подверсия (Subversion, SVN). Обе они в наши дни немного устарели, поскольку интерес сместился в сторону распределенного версионного контроля. Однако при правильном использовании централизованного хранилища они эффективны и намного проще в использовании.

Централизованный версионный контроль также обеспечивает более качественную безопасность для проприетарных исходных кодов, имея только одно место, где хранилище должно быть защищено. По этой причине централизованный версионный контроль по-прежнему популярен в корпоративной среде, где ограничение доступа к истории исходного кода имеет первостепенное значение. CVS имеет простую хорошо работающую операцию ветвления. На веб-сайте CVS есть документация и широко востребованная книга:

- CVS (Free Software Foundation, Inc., 1998) по адресу https://www.non-gnu.org/cvs/;
- Пер Седерквист, «Версионный контроль с помощью CVS» (Per Cederqvist, *Version Management with CVS*, Network Theory Ltd, December 2002), доступная на различных веб-сайтах онлайн и в печатном виде.

Subversion была разработана как замена CVS. Хотя во многих отношениях она является улучшением, по сравнению с CVS, ее функция ветвления немного слабее, чем в CVS. Есть хорошая книга о Subversion, и ее написание продолжается:

- Бен Коллинз-Сассман, Брайан У. Фитцпатрик и К. Майкл Пилато, «Версионный контроль с помощью Subversion» (Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato, *Version Control with Subversion*, Apache Software Foundation, 2002) по адресу http://svnbook.red-bean.com.

17.2 Таймерные процедуры для отслеживания производительности исходного кода

Полезно встраивать внутренние таймеры в свое приложение, чтобы отслеживать производительность во время работы с ним. Мы показываем репрезентативную таймерную процедуру в листингах 17.1 и 17.2, которую вы можете использовать на C, C++ и Fortran с помощью процедуры-обертки Fortran. В указанной процедуре используется процедура `clock_gettime` с типом `CLOCK_MONOTONIC`, чтобы избегать проблем с корректировками показания времени.

Листинг 17.1 Заголовочный файл таймера

timer.h

```
1 #ifndef TIMER_H
2 #define TIMER_H
3 #include <time.h>
4
5 void cpu_timer_start1(struct timespec *tstart_cpu);
6 double cpu_timer_stop1(struct timespec tstart_cpu);
7 #endif
```

Листинг 17.2 Файл таймерного исходного кода

timer.c

```
 1 #include <time.h>
 2 #include "timer.h"
 3
 4 void cpu_timer_start1(struct timespec *tstart_cpu)
 5 {
 6    clock_gettime(CLOCK_MONOTONIC, tstart_cpu);
 7 }
 8 double cpu_timer_stop1(struct timespec tstart_cpu)
 9 {
10    struct timespec tstop_cpu, tresult;
11    clock_gettime(CLOCK_MONOTONIC, &tstop_cpu);
12    tresult.tv_sec = tstop_cpu.tv_sec - tstart_cpu.tv_sec;
13    tresult.tv_nsec = tstop_cpu.tv_nsec - tstart_cpu.tv_nsec;
14    double result = (double)tresult.tv_sec +
                      (double)tresult.tv_nsec*1.0e-9;
15
16    return(result);
17 }
```

Вызывает clock_gettime, запрашивая монотонные часы

Существуют и другие имплементации таймера, которые можно использовать, если вам нужна альтернативная процедура. Переносимость – это одна из причин, по которой вам может понадобиться еще одна имплементация. Процедура `clock_gettime` поддерживается в macOS с версии Sierra 10.12, что помогло решить некоторые проблемы с переносимостью.

Альтернативные имплементации таймера

Если вы используете C++ со стандартами 2011 года, то можете использовать часы высокой разрешающей способности, `std::chrono::high_resolution_clock`. Здесь мы показываем список альтернативных таймеров, которые можно использовать с возможностью переносимости на C, C++ и Fortran.

- `clock_gettime` с типом `CLOCK_MONOTONIC`.
- `clock_gettime` с типом `CLOCK_REALTIME`.
- `gettimeofday`.
- `getrusage`.
- `host_get_clock_service` для macOS.
- `clock std:chrono_high_resolution_clock` (высокое разрешение C++, стандарт C++ 2011).

Функция `clock_gettime` имеет две версии. Хотя `CLOCK_MONOTONIC` предпочтительнее, он не является обязательным типом для Переносимого интерфейса операционной системы (Portable Operating System Interface, POSIX), стандарта переносимости между операционными системами. В таймерный каталог примеров, прилагаемых к этой главе, мы включили версию с таймерным типом `CLOCK_REALTIME`. Функции `gettimeofday` и `ge-`

`trusage` широко переносимы и могут работать в системах, где `clock_gettime` не работает.

Пример: эксперимент с таймерами

Используйте исходный код по адресу https://github.com/EssentialsofParallelComputing/Chapter17 в каталоге таймеров. В этом каталоге нужно выполнить следующие ниже команды:

```
mkdir build && cd build
cmake ..
make
./runit.sh
```

Этот пример собирает различные имплементации таймера и их выполняет. Указанный пример дает вам несколько идей по поводу альтернативных вариантов, если принятая по умолчанию версия не работает либо ведет себя в вашей системе ненадежно.

17.3 Профилировщики: невозможно улучшить то, что не измеряется

Профилировщик – это инструмент программиста, который измеряет некоторые аспекты производительности приложения. Мы рассмотрели профилирование ранее в разделах 2.2 и 3.3 как ключевую часть процесса разработки приложений и представили несколько простых инструментов профилирования. В настоящем разделе мы охватим несколько альтернативных инструментов профилирования, которые вы, возможно, будете использовать при разработке приложений, и познакомим вас с тем, как использовать некоторые простые профилировщики. Профилировщики являются важными инструментами при разработке параллельных приложений, когда:

- вы хотите работать над разделом исходного кода, который оказывает наибольшее влияние на повышение производительности вашего приложения. Этот раздел исходного кода часто называют *узким местом*, или *бутылочным горлышком*;
- вы хотите измерять повышение производительности на различных архитектурах. В конце концов, мы все заботимся о производительности в приложениях высокопроизводительных вычислений.

Профилировщики бывают различных форм и размеров. Мы разделим наше обсуждение на категории, которые отражают их общие характеристики. Важно использовать инструмент из соответствующей категории. Не рекомендуется использовать тяжеловесный инструмент профилирования, когда вам нужно найти лишь самое большое узкое место. Неправильный инструмент похоронит вас в лавине информации, которая за-

ставит копаться часами или днями. Оставьте тяжеловесные инструменты на то время, когда вам действительно нужно будет погрузиться в низкоуровневые детали вашего приложения. Мы предлагаем начать с простых профилировщиков и при необходимости переходить к детализированным. Наши категории профилировщиков следуют этой иерархии от простого к сложному с некоторым субъективным суждением о том, куда в списке нужно помещать каждый инструмент профилирования.

Таблица 17.2 Категории инструментов профилирования (от простых к сложным)

Простые тексто-ориентированные профилировщики	Возвращает краткое текстовое резюме производительности
Высокоуровневые профилировщики	Нисходящий профилировщик, который выделяет процедуры, требующие улучшения, часто в графическом пользовательском интерфейсе
Среднеуровневые профилировщики	Профилировщики, которые дают управляемый объем данных о производительности
Подробные профилировщики	Похороните меня вместе с данными, пожалуйста

Высокий уровень не указывает на высокую детализацию; эти инструменты дают 25 000-футовую картину производительности приложения. Мы возвращаемся к более простым инструментам профилирования, таким как простые тексто-ориентированные профилировщики и высокоуровневые профилировщики, потому что они быстры в использовании и не занимают большую часть дня.

17.3.1 Простые тексто-ориентированные профилировщики для повседневного использования

Простые тексто-ориентированные профилировщики, такие как LIKWID, gprof, gperftools, timemory и Open|SpeedShop, легко включать в ваш ежедневный рабочий поток разработки приложений. Они дают быстрое представление о производительности.

Комплект инструментов likwid («будто я знаю, что делаю») был впервые представлен в разделе 3.3.1, а также использовался в главах 4, 6 и 9. Мы широко использовали его из-за его простоты. На веб-сайте likwid имеется обширная документация:

- Инструменты для повышения производительности likwid по адресу https://hpc.fau.de/research/tools/likwid/.

Почтенный инструмент gprof уже много лет является опорой для профилирования приложений в Linux. Мы использовали его в разделе 13.4.2 для краткого описания нашего приложения. В gprof используется выборочный подход для измерения мест, где приложение тратит свое время. Этот инструмент командной строки задействуется путем добавления опции `-pg` во время компилирования и связывания вашего приложения. Затем, когда ваше приложение отработает, по завершении оно создает файл с именем gmon.out. Затем консольная утилита gprof выводит на экран данные о производительности в виде текста. Инструмент gprof

поставляется с большинством систем Linux и является частью компиляторов GCC и Clang/LLVM. Gprof относительно устарел, но легко доступен и прост в использовании. Документация по gprof довольно проста и широко доступна на следующем ниже веб-сайте.

- Документация GNU Binutils от Фонда свободного программного обеспечения по адресу https://sourceware.org/binutils/docs/gprof/index.html.

Комплект инструментов gperftools (первоначально Google Performance Tools) – это более новый инструмент профилирования, по функциональности аналогичный инструменту gprof. Указанный комплект инструментов также поставляется с TCMalloc, быстрой библиотекой malloc для приложений, использующих потоки. Он также содержит детектор утечки памяти и профилировщик кучи. Профилировщик CPU gperftools имеет веб-сайт с кратким введением в указанный инструмент.

- Gperftools (Google) по адресу https://gperftools.github.io/gperftools/cpuprofile.html.

Инструмент timemory от Национального научно-вычислительного центра энергетических исследований (National Energy Research Scientific Computing Center, NERSC) представляет собой простой инструмент, построенный поверх многих других интерфейсов измерения производительности. Самый простой инструмент в этом наборе, timem, является заменой команды Linux time, которая также может выводить дополнительную информацию, такую как используемая память и число прочитанных и записанных байтов. Примечательно, что у него есть возможность автоматически создавать график контура крыши. На документационном веб-сайте указанного инструмента содержится обширная информация о его использовании.

- Документация по timemory по адресу https://timemory.readthedocs.io.

В Open|SpeedShop есть опция командной строки и интерфейс Python, которые могут сделать его возможной заменой приведенным выше простым инструментам. Мы обсудим этот более мощный инструмент в разделе 17.3.4.

17.3.2 Высокоуровневые профилировщики для быстрого выявления узких мест

Высокоуровневые инструменты являются наилучшим вариантом выбора для быстрого обзора производительности вашего приложения. Эти инструменты отличаются тем, что фокусируются на выявлении дорогостоящих частей вашего исходного кода и дают устойчивый графический обзор производительности приложений. В отличие от простых профилировщиков вам нередко приходится выходить из рабочего потока и запускать графическое приложение, чтобы использовать указанные высокоуровневые профилировщики.

Мы впервые обсудили Cachegrind в разделе 3.3.1. Cachegrind специализируется на том, чтобы показывать вам дорогостоящие пути в вашем исходном коде, позволяя сосредотачиваться на критически важных для производительности частях. Он имеет простой легко понимаемый графический пользовательский интерфейс.

- Cachegrind, профилировщик кеша и предсказания ветвлений (разработчики Valgrind™) по адресу https://valgrind.org/docs/manual/cg-manual.html.

Еще одним неплохим высокоуровневым профилировщиком является профилировщик Arm MAP, ранее именовавшийся Allinia Map или Forge Map. MAP является коммерческим инструментом, и его родительская фирма несколько раз менялась. В нем используется графический пользовательский интерфейс, который дает больше деталей, чем KCachegrind, но по-прежнему фокусируется на наиболее важных деталях. Инструмент MAP имеет сопутствующий инструмент, отладчик DDT, который входит в комплект инструментов высокопроизводительных вычислений Arm Forge. Мы обсудим отладчик DDT позже в разделе 17.7.2 этой главы. На веб-сайте ARM есть обширная документация, учебные пособия, вебинары и руководство пользователя.

- Arm MAP (Arm Forge) по адресу http://mng.bz/n2x2.

17.3.3 Среднеуровневые профилировщики для руководства разработкой приложений

Среднеуровневые профилировщики нередко используются при попытке точной настройки оптимизации. К этой категории относятся многие инструменты на основе графического пользовательского интерфейса, предназначенные для руководства разработкой приложений. К ним относятся Intel® Advisor, VTune, CrayPat, AMD pProf, NVIDIA Visual Profiler и CodeXL (ранее инструмент Radeon, а теперь часть инициативы GPUOpen). Мы начинаем с более общих и популярных инструментов для CPU-процессоров, а затем переходим к специализированным инструментам для GPU-процессоров.

Intel® Advisor предназначен для руководства использованием векторизации с компиляторами Intel. Он показывает векторизованные циклы и предлагает изменения для векторизации других. Хотя это особенно полезно для векторизации исходного кода, он также хорошо подходит для общего профилирования. Advisor является проприетарным инструментом, но в последнее время стал доступен бесплатно для многих пользователей. Инсталлировать Intel Advisor можно с помощью менеджера пакетов Ubuntu. Вам нужно добавить пакет Intel, а затем использовать `apt-get` для инсталлирования версии с oneAPI.

```
wget -q https://apt.repos.intel.com/intel-gpg-keys/GPG-PUB-KEY-INTEL-
    SWPRODUCTS-2023.PUB
apt-key add GPG-PUB-KEY-INTEL-SW-PRODUCTS-2023.PUB
rm -f GPG-PUB-KEY-INTEL-SW-PRODUCTS-2023.PUB
```

```
echo "deb https://apt.repos.intel.com/oneapi all main" >>
     /etc/apt/sources.list.d/oneAPI.list
echo "deb [trusted=yes arch=amd64]
     https://repositories.intel.com/graphics/ubuntu bionic
    main" >> /etc/apt/sources.list.d/intel-graphics.list
apt-get update
apt-get install intel-oneapi-advisor
```

Полные инструкции по инсталлированию программно-информационного обеспечения Intel oneAPI из репозитория пакетов можно найти по адресу http://mng.bz/veO4.

Intel® VTune является общецелевым инструментом оптимизации, который помогает выявлять узкие места и потенциальные улучшения. Он также является еще одним проприетарным инструментом, который находится в свободном доступе. VTune можно инсталлировать с помощью `apt-get` из пакета oneAPI.

```
wget -q https://apt.repos.intel.com/intel-gpg-keys/GPG-PUB-KEY-INTEL-
     SWPRODUCTS-2023.PUB
apt-key add GPG-PUB-KEY-INTEL-SW-PRODUCTS-2023.PUB
rm -f GPG-PUB-KEY-INTEL-SW-PRODUCTS-2023.PUB
echo "deb https://apt.repos.intel.com/oneapi all main" >>
     /etc/apt/sources.list.d/oneAPI.list
echo "deb [trusted=yes arch=amd64]
     https://repositories.intel.com/graphics/ubuntu bionic
    main" >> /etc/apt/sources.list.d/intel-graphics.list
apt-get update
apt-get install intel-oneapi-vtune
```

Инструмент CrayPat является проприетарным инструментом, доступным только в операционных системах Cray. Это отличный инструмент командной строки, который обеспечивает простую обратную связь по оптимизации циклов и потоков. Если вы работаете на одном из многих веб-сайтов высокопроизводительных вычислений, использующих операционную систему Cray, то этот инструмент, возможно, стоит изучить. К сожалению, он недоступен в других местах.

AMD pProf является инструментом профилирования от AMD для их CPU-процессоров и APU-модулей. Модуль ускоренной обработки (Accelerated Processing Unit, APU) является термином AMD для CPU со встроенным GPU, который был впервые представлен, когда AMD выкупила ATI, производителя GPU Radeon. Интегрированный блок более тесно состыкован, чем обычный интегрированный GPU, и является частью концепции Гетерогенной системной архитектуры от AMD. Инсталлировать инструмент AMD pProf можно с помощью инсталляторов пакетов в Ubuntu или Red Hat Enterprise Linux. Для его скачивания требуется ручное принятие лицензионного соглашения. В целях инсталлирования AMD pProf следует выполнить следующие ниже шаги.

1. Перейти на https://developer.amd.com/amd-uprof/.
2. Прокрутить вниз страницы и выбрать соответствующий файл.

3 Принять лицензионное соглашение, чтобы начать скачивание с помощью менеджера пакетов:

```
Ubuntu: dpkg --install amduprof_x.y-z_amd64.deb
RHEL: yum install amduprof-x.y-z.x86_64.rpm
```

Более подробная информация об инсталляции приведена в руководстве пользователя, которое доступно на веб-сайте разработчика AMD: https://developer.amd.com/wordpress/media/2013/12/User_Guide.pdf.

NVIDIA Visual Profiler является частью комплекта программно-информационного обеспечения CUDA. Он включается в комплект инструментов NVIDIA® Nsight. Мы рассмотрели этот инструмент в разделе 13.4.3. Инструменты NVIDIA можно инсталлировать в дистрибутиве Ubuntu Linux с помощью следующих ниже команд:

```
wget -q https://developer.download.nvidia.com/
   compute/cuda/repos/ubuntu1804/x86_64/cuda-repo-ubuntu1804_10.2.89-
      1_amd64.deb
dpkg -i cuda-repo-ubuntu1804_10.2.89-1_amd64.deb
apt-key adv --fetch-keys https://developer.download.nvidia.com/
   compute/cuda/repos/ubuntu1804/x86_64/7fa2af80.pub
apt-get update
apt-get install cuda-nvprof-10-2 cuda-nsight-systems-10-2 cuda-nsight-
   compute-10-2
```

CodeXL является тренажером для разработки исходного кода GPUOpen с поддержкой профилирования для GPU-процессоров Radeon. Он является частью инициативы GPUOpen с открытым исходным кодом, начатой компанией AMD. Инструмент CodeXL сочетает в себе функциональность отладчика и профилировщика. Профилирование CPU было перенесено в инструмент AMD pProf, чтобы можно было перевести инструмент CodeXL в статус с открытым исходным кодом. Следуйте инструкциям по инсталлированию CodeXL в дистрибутивах Ubuntu или RedHat Linux.

```
wget https://github.com/GPUOpen-Archive/
          CodeXL/releases/download/v2.6/codexl-2.6-302.x86_64.rpm
RHEL or CentOS: rpm -Uvh --nodeps codexl-2.6-302.x86-64.rpm
Ubuntu: apt-get install rpm
        rpm -Uvh --nodeps codexl-2.6-302.x86-64.rpm
```

17.3.4 Детализированные профилировщики обеспечивают подробные сведения о производительности оборудования

Существует несколько инструментов, которые производят детализированное профилирование приложений. Если вам нужно извлечь из вашего приложения максимум производительности, то вам следует научиться использовать хотя бы один из этих инструментов. Проблема с этими

инструментами заключается в том, что они дают столь много информации, что для понимания и использования их результатов может потребоваться много времени. Вам также потребуется некоторый опыт в области аппаратной архитектуры, чтобы действительно разобраться в данных профилирования. Инструменты из этой категории следует использовать после того, как вы получите все возможное от более простых инструментов профилирования. В этом разделе мы рассмотрим детализированные профилировщики HPCToolkit, Open|SpeedShop и TAU.

HPCToolkit является мощным, детализированным профилировщиком, разработанным в качестве проекта с открытым исходным кодом Университетом Райса. В HPCToolkit для измерения производительности используются счетчики производительности оборудования; указанный инструмент представляет данные с помощью графических пользовательских интерфейсов. Разработка для новейших систем высокопроизводительных вычислений экстремального масштаба спонсируется Проектом экзомасштабных вычислений Министерства энергетики (DOE). Его графический интерфейс hpcviewer выводит данные о производительности с точки зрения исходного кода, тогда как hpctraceviewer выводит временную трассу исполнения исходного кода. Более подробная информация и подробные руководства пользователя доступны на веб-сайте HPCToolkit. HPCToolkit можно инсталлировать посредством менеджера пакетов Spack с помощью команды `spack install hpctoolkit`.

- HPCToolkit по адресу http://hpctoolkit.org.

Open|SpeedShop является еще одним профилировщиком, который может создавать детализированные профили программ. Он имеет как графический пользовательский интерфейс, так и интерфейс командной строки. Инструмент Open|SpeedShop работает на всех новейших системах высокопроизводительных вычислений благодаря финансированию Министерства здравоохранения. Он поддерживает MPI, OpenMP и CUDA. Open|Speedshop имеет открытый исходный код и может скачиваться бесплатно. На их веб-сайте есть подробные руководства пользователя и учебные пособия. Open|SpeedShop можно инсталлировать посредством менеджера пакетов Spack с помощью команды `spack install openspeedshop`.

- Open|SpeedShop по адресу https://openspeedshop.org.

TAU является инструментом профилирования, разработанным главным образом в Университете штата Орегон. Этот свободно доступный инструмент имеет простой в использовании графический пользовательский интерфейс. TAU используется во многих крупнейших приложениях и системах высокопроизводительных вычислений. На веб-сайте инструмента имеется обширная документация по использованию TAU. TAU можно инсталлировать посредством менеджера пакетов Spack с помощью команды `spack install tau`.

- Лаборатория исследования производительности (Университет штата Орегон) по адресу http://www.cs.uoregon.edu/research/tau/home.php.

17.4 Сравнительные тесты и мини-приложения: окно в производительность системы

Мы отметили ценность сравнительных тестов и мини-приложений для оценивания производительности ваших приложений в главе 3. Сравнительные тесты более подходят для измерения производительности системы. Мини-приложения больше ориентированы на прикладные области и на то, как лучше всего имплементировать алгоритмы для различных архитектур, но разница между ними иногда бывает размытой.

17.4.1 Сравнительные тесты измеряют характеристики производительности системы

Ниже приведен список сравнительных тестов, которые могут быть полезными для оценивания потенциальной производительности вашей системы. В наших исследованиях производительности мы широко использовали сравнительный тест STREAM Benchmark, но для вашего приложения могут больше подходить другие сравнительные тесты. Например, если ваше приложение загружает одно значение данных из разрозненных областей памяти, то наиболее подходящим будет случайный сравнительный тест Random benchmark.

- Linpack по адресу http://www.netlib.org/benchmark/hpl/ – этот сравнительный тест используется для списка 500 лучших высокопроизводительных компьютеров.
- STREAM по адресу https://www.cs.virginia.edu/stream/ref.html – сравнительный тест пропускной способности памяти. Его версия находится в репозитории Git по адресу https://github.com/jeffhammond/STREAM.git.
- Random по адресу http://icl.cs.utk.edu/projectsfiles/hpcc/RandomAccess/ – сравнительный тест производительности доступа к произвольной памяти.
- NAS Parallel Benchmarks по адресу http://www.nas.nasa.gov/publications/npb.html – сравнительные тесты NASA, впервые выпущенные в 1991 году, включают в свой состав несколько наиболее часто используемых сравнительных тестов для исследований.
- HPCG по адресу http://www.hpcg-benchmark.org/software/ – новый сравнительный тест на основе конъюгатного градиента, разработанный в качестве альтернативы Linpack. HPCG обеспечивает более реалистичный контроль за производительностью для современных алгоритмов и компьютеров.
- HPC Challenge Benchmark по адресу http://icl.cs.utk.edu/hpcc/ – композитный сравнительный тест.
- Parallel Research Kernels по адресу https://github.com/ParRes/Kernels – различные небольшие вычислительные ядра из типичных исходных кодов научной симуляции и в нескольких параллельных имплементациях.

17.4.2 Мини-приложения придают приложению перспективу

Приложения приходится много адаптировать под новые архитектуры. С помощью мини-приложений вы можете высвечивать производительность приложений простого типа на целевой системе. В этом разделе представлен список мини-приложений, разработанных лабораториями Министерства энергетики (Department of Energy, DOE), которые могут стать для вашего приложения полезной справочной имплементацией.

Лабораториям DOE было поручено разработать экзомасштабные компьютеры, которые обеспечивают передовые возможности высокопроизводительных вычислений. Указанные лаборатории создали мини-приложения и прокси-приложения для разработчиков оборудования и разработчиков приложений, чтобы иметь возможность экспериментировать с тем, как получать максимальную отдачу от упомянутых экзомасштабных систем. Каждое из этих мини-приложений имеет свое предназначение. Некоторые из них отражают производительность крупного приложения, тогда как другие предназначены для алгоритмического разведывательного анализа. Для начала давайте дадим определение пары терминов, которые помогут нам классифицировать мини-приложения.

- *Прокси-мини-приложение* – это выдержка из или уменьшенная форма более крупного приложения, которая отражает его рабочие характеристики. Прокси широко используются среди поставщиков оборудования в процессе содизайна исходного кода как небольшое приложение, которое они могут использовать в процессе дизайна оборудования.
- *Исследовательское мини-приложение* – более простая форма вычислительного подхода, которая широко используется исследователями для разведывательного анализа альтернативных алгоритмов и методов повышения производительности и новых архитектур.

Классификация мини-приложений не идеальна. У каждого автора мини-приложения есть своя причина для его создания, которая зачастую не укладывается в четкие категории.

Прокси-приложения проекта экзомасштабных вычислений: поперечный срез образцов приложений

Министерство энергетики разработало несколько образцов приложений для использования в системах сравнительного анализа, экспериментах с производительностью и разработке алгоритмов. Многие из них были организованы Проектом экзомасштабных вычислений DOE, размещенным по адресу https://proxyapps.exascaleproject.org/.

- *AMG* – пример алгебраической мультирешетки.
- *ExaMiniMD* – прокси-приложение для исходных кодов обработки частиц и молекулярной динамики.
- *Laghos* – неструктурированная, сжимаемая ударная гидродинамика.
- *MACSio* – масштабируемые тесты ввода-вывода.

- *miniAMR* – мини-приложение для блочно-ориентированной адаптивной детализации расчетной сетки.
- *miniQMC* – мини-приложение на основе квантового метода Монте-Карло.
- *NEKbone* – несжимаемый решатель Навье-Стокса с использованием спектральных элементов.
- *PICSARlite* – электромагнитная частица в ячейке.
- *SW4lite* – вычислительные ядра трехмерного сейсмического моделирования.
- *SWFFT* – быстрое преобразование Фурье.
- *Thornado-mini* – конечно-элементный, моментум-ориентированный перенос излучения.
- *XSBench* – вычислительное ядро из приложения нейтроники Монте-Карло.

Прокси-приложения экзомасштабного проекта выбираются из многочисленных прокси-приложений, разработанных национальными лабораториями. В следующих далее разделах мы перечислим другие прокси- и мини-приложения, разработанные различными национальными лабораториями для научных приложений, которые важны для миссии их исследовательской лаборатории. Эти приложения доступны для общественности и разработчиков оборудования в рамках национальной стратегии разработки исходного кода. *Процесс разработки исходного кода* заключается в том, что разработчики оборудования и разработчики приложений тесно сотрудничают в цикле обратной связи, который перебирает функциональности этих экзомасштабных систем.

Нередко приложения, отражением которых являются эти мини-приложения, как правило, являются проприетарными и, следовательно, недоступны за пределами соответствующей лаборатории. С выпуском некоторых из этих мини-приложений мы осознаем, что современные приложения более сложны и по-другому нагружают оборудование, чем простые имевшиеся ранее вычислительные ядра.

Прокси-приложения Национальной лаборатории Лоуренса Ливермора

Национальная лаборатория Лоуренса Ливермора была одним из ведущих сторонников разработки прокси-приложений. Их прокси-приложение LULESH является одним из наиболее тщательно изучаемых поставщиками и академическими исследователями. Некоторые прокси-приложения Национальной лаборатории Лоуренса Ливермора таковы:

- LULESH – явно заданная лагранжева ударная гидродинамика в представлении в виде неструктурированной вычислительной сетки;
- Kripke – детерминированный транспорт на основе витков;
- Quicksilver – перенос частиц на основе метода Монте-Карло.

Для получения более подробной информации о прокси-приложениях Национальной лаборатории Лоуренса Ливермора следует обратиться на их веб-сайт по адресу https://computing.llnl.gov/projects/co-design/proxy-apps.

Прокси-приложения Национальной лаборатории Лос-Аламоса

В Лос-Аламосской национальной лаборатории также есть много интересных прокси-приложений. Некоторые из наиболее популярных перечислены ниже.

- *CLAMR* – мини-приложение для ячеечно-ориентированной адаптивной детализации вычислительной сетки.
- *NuT* – прокси на основе метода Монте-Карло для переноса нейтрино.
- *Pennant* – мини-приложение по гидродинамике на основе неструктурированной вычислительной сетки.
- *SNAP* – прокси-приложение SN (Дискретные ординаты).

Для получения более подробной информации о прокси-приложениях Лос-Аламосской национальной лаборатории следует обратиться на их веб-сайт по адресу https://www.lanl.gov/projects/codesign/proxy-apps/lanl/index.php.

Комплект мини-приложений проекта Mantevo Национальных лабораторий Сандия

Национальные лаборатории Сандия создали фирменный комплект мини-приложений под названием Mantevo, который включает в себя свои мини-приложения и мини-приложения нескольких других организаций, таких как Организация по атомному оружию Соединенного Королевства (Atomic Weapons Establishment (Atomic Weapons Establishment, AWE). Вот список их мини-приложений:

- *CloverLeaf* – мини-приложение для гидрокода сжимаемых жидкостей на основе декартовой решетки;
- *CoMD* – мини-приложение для молекулярной динамики;
- *EpetraBenchmarkTest* – плотные вычислительные ядра математических решателей;
- *MiniAero* – неструктурированный сжимаемый Навье-Стокс;
- *miniFE* – прокси-приложение для неструктурированных неявных конечно-элементных кодов;
- *miniGhost* – прокси-приложение для обновления призрачных ячеек;
- *miniSMAC2D* – несжимаемый решатель Навье-Стокса с подгонкой к поверхности тел;
- *miniXyce* – мини-приложение для симуляции схем;
- *TeaLeaf* – прокси-приложение для неструктурированных неявно заданных конечно-элементных кодов.

Более подробная информация о комплекте мини-приложений Mantevo расположена по адресу https://mantevo.github.io.

17.5 Обнаружение (и исправление) ошибок памяти для устойчивого приложения

Для устойчивых приложений вам нужен инструмент обнаружения ошибок памяти и сообщения о них. В этом разделе мы обсудим способности, плюсы и минусы ряда инструментов, которые обнаруживают ошибки памяти и сообщают об них. Возникающие в приложениях ошибки памяти можно разделить на следующие три категории:

- *ошибки выхода за пределы границ* – попытка доступа к памяти за пределами массива. Указанные ошибки могут улавливаться инструментами проверки столбов ограждения и некоторыми компиляторами;
- *утечки памяти* – выделение памяти и ее невысвобождение. Инструменты-дублеры библиотеки malloc хорошо справляются с обнаружением и сообщением об утечках памяти;
- *неинициализированная память* – память, используемая до ее выделения. Поскольку память не задана перед тем, как ее использовать, в ней находится любое значение, которое может быть в памяти после предыдущего использования. Как следствие, поведение приложения может меняться от выполнения к выполнению. Указанный тип ошибок трудно выявлять, и необходимы инструменты, специально разработанные для их обнаружения.

Только несколько инструментов обрабатывают все эти категории ошибок памяти. Большинство инструментов в той или иной степени справляется с первыми двумя категориями. Проверки неинициализированной памяти имеют большую важность и поддерживаются всего несколькими инструментами. Сначала мы рассмотрим именно эти инструменты.

17.5.1 Инструмент Valgrind Memcheck: дублер с открытым исходным кодом

Valgrind проверяет неинициализированную память с помощью своего используемого по умолчанию инструмента проверки памяти Memcheck. Мы впервые представили Valgrind для этой цели в разделе 2.1.3. Valgrind является хорошим вариантом выбора по двум причинам: он имеет открытый исходный код, доступный в свободном доступе, и он является одним из лучших инструментов обнаружения ошибок памяти во всех трех категориях.

Valgrind лучше всего использовать с компилятором GCC. Коллектив GCC использует его для своей разработки и, как следствие, очистил свой генерируемый исходный код, чтобы отпала необходимость в файле подавления ложных срабатываний для их последовательных приложений. В случае параллельных приложений вы также можете подавлять обнаруживаемые инструментом Valgrind ложные срабатывания с помощью OpenMPI, используя файл подавления, предоставляемый пакетом OpenMPI. Например:

```
mpirun -n 4 valgrind \
  --suppressions=$MPI_DIR/share/openmpi/openmpi-valgrind.supp <my_app>
```

В инструменте Valgrind имеется всего несколько опций командной строки, и в своем отчете он часто предлагает опции, которые следует использовать. Для получения дополнительной информации об использовании обратитесь к веб-сайту Valgrind (https://valgrind.org).

17.5.2 Dr. Memory для заболеваний вашей памяти

Да, именно так он и называется. Инструмент Dr. Memory аналогичен инструменту Valgrind, но является более новым и быстрым. Как и Valgrind, инструмент Dr. Memory обнаруживает ошибки и проблемы с памятью в вашей программе. Указанный проект с открытым исходным кодом свободно доступен для различных микросхемных архитектур и операционных систем.

В этом комплекте инструментов времени выполнения, помимо собственно Dr. Memory, есть много других инструментов. Поскольку инструмент Dr. Memory является относительно простым, мы приведем краткий пример его использования. Давайте сначала настроим Dr. Memory для использования.

Пример: использование инструмента Dr. Memory для обнаружения ошибок памяти

Перейдите по адресу https://github.com/DynamoRIO/drmemory/wiki/Downloads и скачайте последнюю версию для Linux:

```
tar -xzvf DrMemory-Linux-2.3.0-1.tar.gz
```

Затем добавьте его в свой путь с помощью команды:

```
export PATH=${HOME}/DrMemory-Linux-2.3.0-1/bin64:$PATH
```

Мы опробуем Dr. Memory на примере из репозитория по адресу https://github.com/EssentialsofParallelComputing/Chapter17. Следующий ниже листинг является копией исходного кода из листинга 4.1 главы 4. Исходный код содержит лишь фрагмент с проверкой того, что синтаксис компилируется правильно.

Листинг 17.3 Пример проверки памяти, выполняемой инструментом Dr. Memory

DrMemory/memoryexample.c

```
1 #include <stdlib.h>
2
3 int main(int argc, char *argv[])
4 {
```

```
 5    int j, imax, jmax;
 6
 7    // сначала выделить память под столбец указателей на тип double
 8    double **x = (double **)                  Чтение переменной jmax
         malloc(jmax * sizeof(double *));       в неинициализированной памяти
 9                                                          Утечка памяти
10    // теперь выделить память под каждую строку данных    для переменной x
11    for (j=0; j<jmax; j++){
12       x[j] = (double *)malloc(imax * sizeof(double));
13    }
14 }
```

Для выполнения этого примера требуется всего несколько команд. Извлеките код из прилагаемых к этой главе примеров и соберите его:

```
git clone --recursive \
    https://github.com/EssentialsofParallelComputing/Chapter17
cd DrMemory
make
```

Теперь выполните пример, исполнив drmemory с последующими двумя тире, а затем именем исполняемого файла: `drmemory -- memoryexample`. На рис. 17.2 показан отчет, создаваемый инструментом Dr. Memory.

```
~~Dr.M~~ Dr. Memory version 2.3.0
~~Dr.M~~                                          Неинициализированное чтение
~~Dr.M~~ Error #1: UNINITIALIZED READ: reading register edx
~~Dr.M~~ # 0 main          [Capterl7/DrMemory/memoryexample.c:l1]
~~Dr.M~~ Note: @0:00:00.401 in thread 146899
~~Dr.M~~ Note: instruction: cmp %eax %edx
~~Dr.M~~                                          Утечка 8, непосредственные байты
~~Dr.M~~ Error #2: LEAK 8 direct bytes 0x0000000000607710-0x0000000000607718 + 33567080 indirect bytes
~~Dr.M~~ # 0 replace_malloc [/drmemory_package/common/alloc_replace.c:2577]
~~Dr.M~~ # 1 main          [Capter17/DrMemory/memoryexample.c:8]
~~Dr.M~~
~~Dr.M~~ ERRORS FOUND:
~~Dr.M~~       0 unique, 0 total unaddressable access(es)
~~Dr.M~~       1 unique, 2 total uninitialized access(es)
~~Dr.M~~       0 unique, 0 total invalid heap argument(s)
~~Dr.M~~       0 unique, 0 total warning(s)
~~Dr.M~~       1 unique, 1 total, 33567088 byte(s) of leak(s)
~~Dr.M~~       0 unique, 0 total,        0 byte(s) of possible leak(s)
~~Dr.M~~ ERRORS IGNORED:
~~Dr.M~~      24 unique, 46 total, 14053 byte(s) of still-reachable allocation(s)
~~Dr.M~~           (re-run with "-show_reachable" for details)
~~Dr.M~~ Details:
Chapter17/DrMemory/DrMemory-Linux-2.3.0-1/drmemory/logs/DrMemory-memoryexample.146899.000/results.txt
```

Рис. 17.2 Отчет инструмента Dr. Memory показывает неинициализированное чтение в строке 11 и утечку памяти для памяти, выделенной в строке 8

Инструмент Dr. Memory правильно сигнализирует о том, что переменная `jmax` при использовании в строке 11 не была инициализирована. Он также показывает утечку в строке 12. В целях исправления ошибок мы

инициализируем переменную jmax, затем высвобождаем каждый указатель `x[j]` и массив `x`, и потом повторяем попытку с помощью `drmemory -- memoryexample`. На рис. 17.3 показан отчет.

```
~~Dr.M~~ Dr. Memory version 2.3.0
~~Dr.M~~
~~Dr.M~~ NO ERRORS FOUND:
~~Dr.M~~       0 unique,  0 total unaddressable access(es)
~~Dr.M~~       0 unique,  0 total uninitialized access(es)
~~Dr.M~~       0 unique,  0 total invalid heap argument(s)
~~Dr.M~~       0 unique,  0 total warning(s)
~~Dr.M~~       0 unique,  0 total,      0 byte(s) of leak(s)
~~Dr.M~~       0 unique,  0 total,      0 byte(s) of possible leak(s)
~~Dr.M~~ ERRORS IGNORED:
~~Dr.M~~      24 unique, 46 total,  14053 byte(s) of still-reachable allocation(s)
~~Dr.M~~          (re-run with "-show_reachable" for details)
~~Dr.M~~ Details:
Chapter17/DrMemory/DrMemory-Linux-2.3.0-1/drmemory/logs/DrMemory-memoryexample.147746.000/results.txt
```

Сообщения об ошибках отсутствуют!

Рис. 17.3 Этот отчет инструмента Dr. Memory показывает, что ошибки неинициализированной памяти и утечки памяти исправлены

После внесения нашей поправки отчет инструмента Dr. Memory на рис. 17.3 не показывает никаких ошибок. Обратите внимание, что Dr. Memory не сигнализирует о том, что переменная `imax` не инициализирована. Дополнительные сведения об инструменте Dr. Memory для Windows, Linux и Mac можно найти по адресу https://drmemory.org.

17.5.3 Коммерческие инструменты памяти для требовательных приложений

Purify и Insure++ представлят собой коммерческие инструменты, которые обнаруживают ошибки памяти, включая некоторую форму проверки неинициализированной памяти. TotalView включает в себя проверку памяти в своих последних версиях. Если у вас есть требовательное приложение, нуждающееся в исходном коде высочайшего качества, и вы ищете поддержку со стороны поставщика своего инструмента памяти, то один из этих коммерческих инструментов может быть хорошим вариантом выбора.

17.5.4 Компиляторно-ориентированные инструменты памяти для удобства

Многие компиляторы включают инструменты памяти в свои продукты. Компилятор LLVM имеет набор инструментов, который включает в себя функциональность проверки памяти. Сюда входит очиститель памяти MemorySanitizer, очиститель адресов AddressSanitizer и очиститель потоков ThreadSanitizer. GCC содержит компонент mtrace, который обнаруживает утечки памяти.

17.5.5 Инструменты проверки столбов ограждения обнаруживают несанкционированный доступ к памяти

Несколько инструментов размещают блоки памяти до и после выделения памяти с целью обнаружения несанкционированного доступа к памяти, а также для отслеживания утечек памяти. Эти типы проверок памяти называются проверками столбов ограждения памяти (fence-post memory check). Они имплементируется в довольно простых инструментах, и обычно предоставляются в виде библиотеки. Кроме того, эти инструменты переносимы и их легко добавлять в обычную систему регрессионного тестирования.

Здесь мы подробно обсудим инструмент dmalloc и то, как использовать инструменты проверки столбов ограждения памяти. Electric Fence и Memwatch – это два других пакета, которые обеспечивают проверку столбов ограждения памяти и имеют аналогичную модель использования, но dmalloc является самым известным инструментом проверки столбов ограждения памяти. Он заменяет библиотеку malloc версией, обеспечивающей проведение проверок памяти.

Пример: настройка инструмента dmalloc

В целях скачивания и инсталлирования инструмента dmalloc следует выполнить следующий ниже исходный код:

```
wget https://dmalloc.com/releases/dmalloc-5.5.2.tgz
tar -xzvf dmalloc-5.5.2.tgz
cd dmalloc-5.5.2/
./configure --prefix=${HOME}/dmalloc
make
make install
```

Для того чтобы добавить dmalloc в путь к исполняемому файлу, в командной строке или в файле настройки среды, следует задать переменную PATH:

```
export PATH=${PATH}:${HOME}/dmalloc/bin
```

Далее следует задать переменную DMALLOC_OPTIONS. Обратные кавычки исполняют команду и задают переменную.

```
export `dmalloc -l logfile -i 100 low`
```

Переменная DMALLOC_OPTIONS теперь должна быть задана в вашей среде. Это может выглядеть примерно так:

```
DMALLOC_OPTIONS=debug=0x4e48503,inter=100,log=logfile
```

Нам нужно добавить эти изменения в файл makefile для связывания библиотеки dmalloc и включить заголовочный файл:

```
CFLAGS = -g -std=c99 -I${HOME}/dmalloc/include -DDMALLOC \
         -DDMALLOC_FUNC_CHECK
LDLIBS=-L${HOME}/dmalloc/lib -ldmalloc
```

В наш исходный код в следующем ниже листинге мы добавили заголовочный файл dmalloc с директивой include в строке 3, чтобы получать номера строк в нашем отчете.

Листинг 17.4 Пример исходного кода dmalloc

Dmalloc/mallocexample.c

```
 1 #include <stdlib.h>
 2 #ifdef DMALLOC
 3 #include "dmalloc.h"
 4 #endif
 5
 6 int main(int argc, char *argv[])
 7 {
 8    int imax=10, jmax=12;
 9
10    // сначала выделить блок памяти под указатели строк
11    double *x = (double *)malloc(imax*sizeof(double *));
12
13    // теперь инициализировать массив x нулями
14    for (int i = 0; i < jmax; i++) {
15       x[i] = 0.0;
16    }
17    free(x);
18    return(0);
19 }
```

Включает в состав заголовочный файл dmalloc

Пишет за пределы конца массива x

В строках 14 и 15 мы задействовали доступ за пределы массива x. Теперь можем создать наш исполняемый файл и выполнить его:

```
make
./mallocexample
```

Но вывод на терминал сообщает об ошибке:

```
debug-malloc library: dumping program, fatal error
   Error: failed OVER picket-fence magic-number check (err 27)
Abort trap: 6
```

Давайте получим больше информации о возникшей проблеме из журнального файла, показанного на рис. 17.4.

Инструмент dmalloc обнаружил несанкционированный доступ. Великолепно! Более подробная информация об инструменте dmalloc находится на его веб-сайте (https://dmalloc.com).

17.5.6 Инструменты памяти GPU для устойчивых приложений GPU

Поставщики GPU-процессоров разрабатывают инструменты памяти с целью обнаружения ошибок памяти для приложений, работающих на

их оборудовании. Соответствующий инструмент был выпущен компанией NVIDIA, и другие производители GPU-процессоров обязательно последуют за ней. Инструмент NVIDIA CUDA-MEMCHECK проверяет ссылки на память за пределами границ, обнаружение гонки данных, ошибки использования синхронизации и неинициализированную память. Указанный инструмент может выполняться как отдельная команда:

```
cuda-memcheck [--tool memcheck|racecheck|initcheck|synccheck] <имя_приложения>
```

```
1595103932: 1: Dmalloc version '5.5.2' from 'http://dmalloc.com/'
1595103932: 1: flags = 0x4e48503, logfile 'logfile'
1595103932: 1: interval = 100, addr = 0, seen # = 0, limit = 0
1595103932: 1: starting time = 1595103932
1595103932: 1: process pid = 22944
1595103932: 1:  error details: checking user pointer
1595103932: 1: pointer '0xl0a4eaf88' from 'unknown' prev access 'mallocexample.c:11'
1595103932: 1:  dump of proper fence-top bytes: 'i\336\312\372'
1595103932: 1:  dump of '0xl0a4eaf88'+64:
'\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000'
1595103932: 1: next pointer '0xl0a4eb000' (size 0) may have run under from 'unknown'
1595103932: 1: ERROR: _dmalloc_chunk_heap_check: failed OVER picket-fence magic-number check (err 27)
```

Отчет о памяти за пределами границ

Рис. 17.4 Журнальный файл инструмента dmalloc показывает доступ к памяти за пределами границ в строке 11

Документация по использованию этого инструмента доступна на вебсайте NVIDIA.

- CUDA-MEMCHECK, Документация по инструментарию CUDA по адресу https://docs.nvidia.com/cuda/cuda-memcheck/index.html.

17.6 Инструменты проверки потоков для определения гоночных условий

Инструменты обнаружения гоночных условий в потоках (также именуемых *опасностями для данных*) имеют решающее значение при разработке приложений OpenMP. Невозможно разрабатывать устойчивые приложения OpenMP без инструмента обнаружения гонок. И все же, существует несколько инструментов, которые способны определять гоночные условия. Двумя эффективными инструментами являются Intel Inspector и Archer, которые мы обсудим далее.

17.6.1 Intel® Inspector: инструмент обнаружения гоночных состояний с графическим интерфейсом

Intel® Inspector представляет собой инструмент с графическим пользовательским интерфейсом, который эффективен при обнаружении гоночных условий в исходном коде OpenMP. Мы обсуждали Intel Inspector

ранее в разделе 7.9. Хотя Inspector является проприетарным инструментом Intel, теперь он имеется в свободном доступе. В Ubuntu его можно инсталлировать из пакета oneAPI от Intel:

```
wget -q https://apt.repos.intel.com/intel-gpg-keys/GPG-PUB-KEY-INTEL-
     SWPRODUCTS-2023.PUB
apt-key add GPG-PUB-KEY-INTEL-SW-PRODUCTS-2023.PUB
rm -f GPG-PUB-KEY-INTEL-SW-PRODUCTS-2023.PUB
echo "deb https://apt.repos.intel.com/oneapi all main" >>
     /etc/apt/sources.list.d/oneAPI.list
echo "deb [trusted=yes arch=amd64]
     https://repositories.intel.com/graphics/ubuntu bionic
    main" >> /etc/apt/sources.list.d/intel-graphics.list
apt-get install intel-oneapi-inspector
```

17.6.2 Archer: тексто-ориентированный инструмент обнаружения гоночных условий

Archer представляет собой инструмент с открытым исходным кодом, построенный поверх очистителя потоков ThreadSanitizer LLVM (TSan) и адаптированный для обнаружения гоночных условий в потоках OpenMP. Использование инструмента Archer, в сущности, заключается в простой замене компиляторной команды на `clang-archer` и связывании библиотеки Archer посредством `-larcher`. Archer выводит свой отчет в виде текста.

Инструмент Archer можно инсталлировать вручную с помощью компилятора LLVM либо посредством менеджера пакетов Spack с помощью команды `spack install archer`. Мы включили несколько сборочных скриптов в прилагаемые примеры по адресу https://github.com/EssentialsofParallelComputing/Chapter17 для его инсталляции. После инсталлирования инструмента Archer вы можете собрать наш пример в подкаталоге Archer примеров. В данном примере мы используем один из стенсильных исходных кодов из раздела 7.3.3. Затем модифицируем сборочную систему CMake, поменяв компиляторную команду на `clang-archer` и добавив библиотеки Archer в команду связывания, как показано в следующем ниже листинге.

Листинг 17.5 Пример исходного кода Archer

Archer/CMakeLists.txt

```
1 cmake_minimum_required (VERSION 3.0)
2 project (stencil)
3
4 set (CC clang-archer)
5
6 set (CMAKE_C_STANDARD 99)
7
8 set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -g -O3")
```

Назначает компиляторной команде значение clang-archer

```
 9
10 find_package(OpenMP)
11
12 # Добавляет сборочную цель стенсиля с файлами исходного кода
13 add_executable(stencil stencil.c timer.c timer.h malloc2D.c malloc2D.h)
14 set_target_properties(stencil PROPERTIES
     COMPILE_FLAGS ${OpenMP_C_FLAGS})
15 set_target_properties(stencil PROPERTIES LINK_FLAGS "${OpenMP_C_FLAGS}
         -L${HOME}/archer/lib -larcher")
```

Добавляет библиотеки archer в LINK_FLAGS

Скомпилируйте исходный код и выполните его, как и раньше:

```
mkdir build && cd build
cmake ..
make
./stencil
```

Как показано на рис. 17.5, мы получаем результат работы инструмента Archer вперемешку с обычным результатом.

```
==================
WARNING: ThreadSanitizer: data race (pid=59460)
  Atomic read of size 1 at 0x7b6800031140 by main thread:
    #0 pthread_mutex_lock
/projects/kitsune/packages/llvm-7.0.0-full-package/projects/compiler-rt/lib/tsan/../sanitizer_common/sanitizer_common_i
nterceptors.inc:4071 (stencil+0x440237)
    #1 __kmp_resume_64 <null> (libomp.so+0x7d4e4)
    #2 __libc_start_main <null> (libc.so.6+0x22554)
 Previous write of size 1 at 0x7b6800031140 by thread T64:
    #0 pthread_mutex_init
/projects/kitsune/packages/llvm-7.0.0-full-package/projects/compiler-rt/lib/tsan/rtl/tsan_interceptors.cc:1184
(stencil+0x42924a)
    #1 __kmp_suspend_initialize_thread(kmp_info*) <null> (libomp.so+0x7e9e3)
  Location is heap block of size 1504 at 0x7b6800030c00 allocated by main thread:
    #0 malloc
/projects/kitsune/packages/llvm-7.0.0-full-package/projects/compiler-rt/lib/tsan/rtl/tsan_interceptors.cc:664
(stencil+0x4533bc)
    #1 ___kmp_allocate <null> (libomp.so+0x1c127)
    #2 __libc_start_main <null> (libc.so.6+0x22554)
  Thread T64 (tid=59525, running) created by main thread at:
    #0 pthread_create
/projects/kitsune/packages/llvm-7.0.0-full-package/projects/compiler-rt/lib/tsan/rtl/tsan_interceptors.cc:965
(stencil+0x428e5b)
    #1 __kmp_create_worker <null> (libomp.so+0x7b358)
    #2 __libc_start_main <null> (libc.so.6+0x22554)
< ... skipping output ... >

Iter 7000
Iter 8000
Iter 9000
Timing is init nan flush 61.890046 stencil 60.155356 total nan
ThreadSanitizer: reported 8 warnings
```

Рис. 17.5 Результат работы инструмента обнаружения гонки данных Archer

Некоторые сообщения о гоночных условиях, о которых сообщается при запуске, выглядят ложными срабатываниями, при этом нет никаких дополнительных сообщений во время выполнения. Для получения дополнительной информации ознакомьтесь со следующей ниже документацией.

- «СЕКАТОРЫ Archer: обеспечение воспроизводимости для выявления недетерминированных ошибок при выполнении на суперкомпьютерах» (Archer PRUNERS: Providing Reproducibility for Uncovering Non-deterministic Errors in Runs on Supercomputers, 2017) по адресу https://pruners.github.io/archer/.
- Репозиторий Archer по адресу https://github.com/PRUNERS/archer.

17.7 Устранители дефектов: отладчики для уничтожения дефектов

Вы тратите большую часть времени на разработку приложений, исправляя дефекты. Это особенно верно при разработке параллельных приложений. Жизненно важен любой инструмент, который помогает в этом процессе. Параллельным программистам также необходимы дополнительные способности, ориентированные на работу с многочисленными процессами и потоками (aka виртуальными ядрами).

Отладчики, используемые для крупных параллельных приложений на узлах высокопроизводительных вычислений, обычно включают в себя несколько коммерческих предложений. К ним относятся мощные и простые в использовании отладчики TotalView и Arm DDT. Но подавляющая часть разработки исходного кода изначально выполняется на ноутбуках, настольных компьютерах и локальных кластерах за пределами крупных центров, поэтому у вас, возможно, не будет доступа к коммерческому отладчику в этих малых системах. Некоммерческие отладчики, доступные для малых кластеров, настольных компьютеров и ноутбуков, более лимитированы в функциональностях параллельного программирования и сложнее в использовании. В этом разделе мы начнем с обсуждения коммерческих отладчиков.

17.7.1 Отладчик TotalView широко доступен на веб-сайтах HPC

Отладчик TotalView имеет обширную поддержку ведущих систем высокопроизводительных вычислений, включая MPI и потокообразование OpenMP. TotalView имеет некоторую поддержку для отладки GPU-процессоров NVIDIA с использованием CUDA. В нем используется простой в навигации графический пользовательский интерфейс; он также обладает большим числом функциональностей, которые требуют некоторого разведывательного анализа. TotalView обычно вызывается путем предварения командной строки префиксом `totalview`. Флаг `-a` указывает на то, что дальше приложению будут переданы остальные аргументы:

```
totalview mpirun -a -n 4 <мое_приложение>
```

В Национальной лаборатории Лоуренса Ливермора есть хороший учебник по Totalview. Подробная информация доступна на веб-сайтах TotalView.

- TotalView (Национальная лаборатория Лоуренса Ливермора) по адресу https://computing.llnl.gov/tutorials/totalview/.
- TotalView (Perforce) по адресу https://totalview.io.

17.7.2 DDT – еще один отладчик, широко доступный на веб-сайтах HPC

Отладчик ARM DDT представляет собой еще один популярный коммерческий отладчик, используемый на площадках высокопроизводительных вычислений. Он имеет обширную поддержку MPI и OpenMP. Он также имеет некоторую поддержку отладки исходного кода CUDA. В отладчике DDT используется интуитивно понятный графический пользовательский интерфейс. Кроме того, DDT поддерживает дистанционную (удаленную) отладку. В этом случае графический интерфейс клиента выполняется в вашей локальной системе, а отлаживаемое приложение дистанционно запускается в системе высокопроизводительных вычислений. Для того чтобы начать сеанс отладки с помощью DDT, следует просто добавить ddt в командную строку:

```
ddt <мое_приложение>
```

В Техасском центре передовых вычислений есть неплохое введение в DDT. Более подробная информация также доступна на веб-сайтах DDT.

- Учебные пособия по отладчику ARM DDT (TACC, Техасский центр передовых вычислений) по адресу https://portal.tacc.utexas.edu/tutorials/ddt.
- ARM DDT (ARM Forge) по адресу https://www.arm.com/products/development-tools/server-and-hpc/forge/ddt.

17.7.3 Отладчики Linux: бесплатные альтернативы для ваших локальных потребностей разработки

Стандартный отладчик Linux под названием GDB широко распространен на платформах Linux. Для того чтобы усвоить его интерфейс командной строки, придется немного поработать. В случае последовательного исполняемого файла отладчик GDB запускается с помощью команды:

```
gdb <мое_приложение>
```

Отладчик GDB не имеет встроенной поддержки параллельного MPI. Вы сможете отлаживать параллельные задания, запуская несколько сеансов GDB с помощью команды `mpirun`. Запуск xterms во всех средах невозможен, вследствие чего данный подход не защищен от неумелых действий.

```
mpirun -np 4 xterm -e gdb ./<мое_приложение>
```

Многие пользовательские интерфейсы более высокого уровня построены поверх отладчика GDB. Самым простым из них является cgdb, который представляет собой интерфейс на основе текстовых окон под названием curses, имея сильное сходство с редактором vi. Интерфейс curses – это символо-ориентированная система Windows. Он имеет преимущество в более высоких характеристиках производительности сети, чем полноценный побитовый графический пользовательский интерфейс. Отладчик cgdb широко доступен вместе с его документацией по указанным ниже ссылкам.

- cgdb, отладчик на основе текстовых окон curses, по адресу https://cgdb.github.io.

Полный графический пользовательский интерфейс для GDB доступен в DataDisplayDebugger, известном как DDD. Веб-сайт отладчика DDD содержит дополнительную информацию о DDD и других подобных отладчиках.

- DDD, отладчик DataDisplayDebugger, по адресу https://www.gnu.org/software/ddd/.

Ни cgdb, ни DDD не включают в свой состав явную поддержку параллельности. Другие пользовательские интерфейсы более высокого уровня, такие как среда разработки Eclipse, предоставляют интерфейс параллельного отладчика поверх отладчика GDB. Среда разработки Eclipse доступна для широкого спектра языков и обеспечивает основу для инструментов программирования CPU- и GPU-процессоров.

- IDE-среды рабочего стола (Eclipse Foundation) по адресу https://www.eclipse.org/ide/.

17.7.4 Отладчики GPU способны помогать устранять дефекты GPU

Наличие отладчиков для разработки исходного кода GPU является решающим фактором, меняющим правила игры. Разработка исходного кода GPU серьезно затруднялась из-за трудностей отлаживания на GPU-процессорах. Инструменты отладки GPU, обсуждаемые в этом разделе, все еще незрелы, но любые возможности крайне необходимы. В указанных отладчиках GPU в значительной степени используются инструменты с открытым исходным кодом, такие как представленные в предыдущем разделе GDB и DDD.

CUDA-GDB: отладчик для GPU-процессоров NVIDIA

В CUDA есть консольный отладчик на основе GDB под названием CUDA-GDB. Существует также версия CUDA-GDB с графическим пользовательским интерфейсом в инструменте NVIDIA Nsight™ Eclipse, входящем в состав их инструментария CUDA. Отладчик CUDA-GDB также был интегрирован в DDD и Emacs. В целях использования отладчика CUDA-GDB с DDD следует запустить DDD с помощью команды `ddd --debugger cuda-`

gdb. Документация CUDA-GDB находится по адресу https://docs.nvidia.com/cuda/cuda-gdb/.

ROCgdb: отладчик для GPU-процессоров RADEON

Отладчик AMD ROCm, являющийся частью инициативы Radeon Open Compute, основан на отладчике GDB, но с первоначальной поддержкой GPU-процессоров AMD. На веб-сайте ROCm есть документация по ROCgdb, но она в значительной степени совпадает с документацией отладчика GDB.

- Веб-сайт отладчика AMD ROCm находится по адресу https://rocmdocs.amd.com/en/ latest/ROCm_Tools/ROCgdb.html.
- Веб-сайт ROCm находится по адресу https://rocmdocs.amd.com.
- Проверьте наличие обновлений для отладчика ROCm в Руководстве пользователя ROCgdb по адресу https://github.com/RadeonOpenCompute/ROCm/blob/master/Debugging%20with%20ROCGDB%20User%20Guide%20v4.1.pdf.

17.8 Профилирование файловых операций

Производительность файловой системы нередко бывает запоздалой мыслью при разработке приложений высокопроизводительных вычислений. В современном мире больших данных, когда производительность файловой системы отстает от производительности других частей вычислительной системы, производительность файловой системы становится все более серьезной проблемой. Необходимых инструментов для измерения производительности файловой системы недостаточно. Инструмент Darshan был разработан, чтобы заполнить этот пробел. Darshan, как инструмент для определения характеристик ввода-вывода HPC, специализируется на профилировании использования приложением файловой системы. С момента своего выпуска Darshan получил широкое распространение в центрах высокопроизводительных вычислений.

Пример: инсталлирование инструмента Darshan

В этом примере вы инсталлируете инструмент Darshan в свой домашний каталог. Возможно, вам захочется собрать инструменты времени выполнения в своем вычислительном кластере и инструменты анализа в другой системе, например на вашем ноутбуке. Для инструментов анализа требуются части дистрибутива LaTeX и несколько простых графических утилит, которых может не быть в компьютерном кластере. Если у вас возникнут проблемы с отсутствующими утилитами, то файл DockerFile с прилагаемыми примерами по адресу https://github.com/EssentialsofParallelComputing/Chapter17.git содержит список всех пакетов, необходимых для выполнения инструментов анализа. Для начала следует скачать и распаковать дистрибутив инструмента Darshan:

```
wget ftp://ftp.mcs.anl.gov/pub/darshan/releases/darshan-3.2.1.tar.gz
tar -xvf darshan-3.2.1.tar.gz
```

Загрузите или инсталлируйте свой пакет MPI. Если он не инсталлирован в стандартном местоположении, то задайте пути с помощью следующих ниже команд экспорта:

```
export CFLAGS=-I<MPI_INCLUDE_PATH>
export LDFLAGS=-L<MPI_LIB_PATH>
```

Соберите инструмент Darshan времени выполнения:

```
cd darshan-3.2.1/darshan-runtime
./configure --prefix=${HOME}/darshan --with-log-path=${HOME}/darshan-logs
            --with-jobid-env=SLURM_JOB_ID --enable-mpiio-mod
make
make install
```

Теперь соберите аналитические инструменты Darshan:

```
cd ../darshan-util
./configure --prefix=${HOME}/darshan
make
make install
```

Затем укажите путь к исполняемым файлам инструмента Darshan:

```
export PATH=${PATH}:${HOME}/darshan/bin
```

Выполните скрипт инструмента Darshan, чтобы настроить каталоги дат в журнальном каталоге Darshan:

```
darshan-mk-log-dirs.pl
```

Добавьте библиотеки инструмента Darshan в свою сборку, используя свои флаги `LINK_FLAGS`. Соответствующие флаги можно получить, исполнив утилиту `darshan-config` с опцией `--dyn-ld-flags`:

```
darshan-config --dyn-ld-flags
```

В сборочной системе CMake можно захватывать результат на выходе из команды и использовать его для задания значения переменной `DARSHAN_LINK_FLAGS`:

```
execute_process(COMMAND darshan-config --dyn-ld-flags
          OUTPUT_STRIP_TRAILING_WHITESPACE
          OUTPUT_VARIABLE DARSHAN_LINK_FLAGS)
```

Затем мы добавляем `DARSHAN_LINK_FLAGS` в переменную `LINK_FLAGS`:

```
set_target_properties(mpi_io_block2d PROPERTIES LINK_FLAGS
         "${MPI_C_LINK_FLAGS} ${DARSHAN_LINK_FLAGS}")
```

Мы внесли эти изменения в файл CMakeLists.txt из каталога MPI_IO_Examples/mpi_io_block2d по адресу https://github.com/EssentialsofParallelComputing/Chapter17.git. Это тот же пример MPI-IO, который мы представили в разделе 16.3, но с более крупной вычислительной сеткой 1000×1000 и с закомментированным исходным кодом проверки. Теперь вы можете собрать и выполнить исполняемый файл, как и раньше:

```
mkdir build && cd build
cmake ..
make
mpirun -n 4 mpi_io_block2d
```

Вы должны найти журналы инструмента Darshan, организованные по дате, в своих подкаталогах ~/darshan-logs.

Пример: использование аналитического инструмента Darshan

Аналитический инструмент Darshan можно выполнить на сгенерированном журнальном файле Darshan:

darshan-job-summary.pl <журнальный_файл_инструмента_darshan>

Вы обнаружите, что на выходе результат будет иметь то же имя файла, что и журнальный файл, но с добавленным расширением .pdf. Указанный результат можно просмотреть с помощью вашего любимого средства просмотра PDF.

Аналитический инструмент Darshan выводит несколько страниц текстовой и графической информации о файловых операциях в вашем приложении в Переносимом формате документа (Portable Document Format, PDF). Мы приводим часть результата на рис 17.6.

Мы создали инструмент времени выполнения с поддержкой профилирования как POSIX, так и MPI-IO. POSIX (англ. *Portable Operating System Interface*, т. е. «Переносимый интерфейс операционной системы») является стандартом переносимости для широкого спектра функций системного уровня, таких как регулярные операции с файловой системой. В нашем модифицированном тесте мы отключили все проверки и другие стандартные операции ввода-вывода, чтобы сосредоточиться на частях исходного кода MPI-IO. Мы также увеличили размеры массивов. Указанный тест был проведен в файловой системе NFS, которая используется для нашего домашнего каталога. На рисунке видно, что мы одновременно выполняли запись и чтение MPI-IO и что запись происходит немного медленнее, чем чтение. Мы также видим, что стоимость операций на метаданных MPI намного выше. При записи файловых метаданных регистрируется информация о местоположении файла, его разрешениях и времени доступа. По своей природе запись метаданных в файл является последовательной операцией.

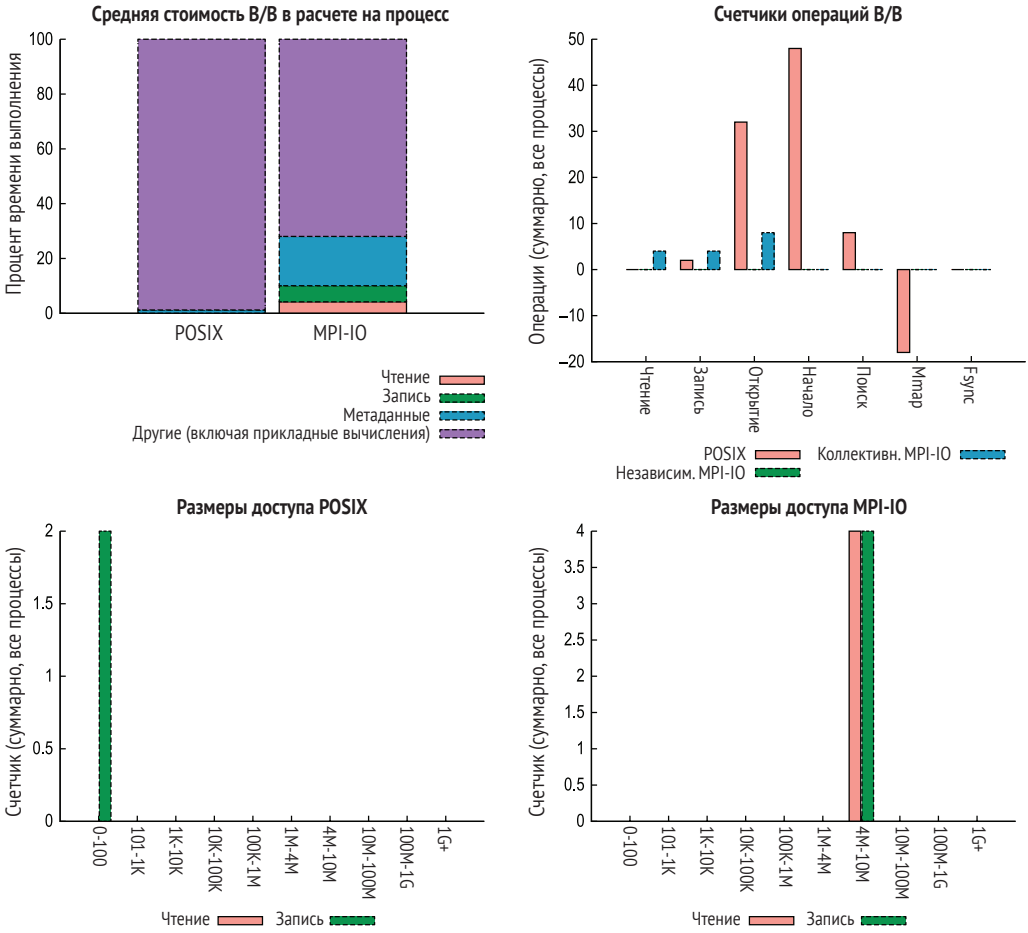


Рис. 17.6 Графики являются частью результата работы инструмента Darshan по определению характеристик ввода-вывода. Показаны как стандартный ввод-вывод (POSIX), так и ввод-вывод MPI. Из графика в правом верхнем углу мы можем подтвердить, что MPI-IO вместо независимых операций использовал коллективные

Darshan также имеет некоторую поддержку профилирования операций с файлами HDF5. Более подробную информацию об инструменте Darshan по определению характеристик ввода-вывода HPC можно получить на веб-сайте проекта:

- https://www.mcs.anl.gov/research/projects/darshan/.

17.9 Менеджеры пакетов: ваш персональный системный администратор

Менеджеры пакетов стали важнейшими инструментами для упрощения инсталляции пакетов программно-информационного обеспечения

в различных системах. Эти инструменты впервые появились в системах Linux вместе с менеджером пакетов Red Hat для управления инсталляцией программно-информационного обеспечения, но с тех пор они получили широкое распространение во многих операционных системах. Использование менеджеров пакетов для инсталлирования инструментов и драйверов устройств может значительно упрощать процесс инсталляции и делать вашу систему более стабильной и актуальной.

Операционные системы Linux в значительной степени зависят от использования управления пакетами. Вы должны использовать свою систему пакетов Linux для инсталлирования программно-информационного обеспечения, когда это возможно. К сожалению, не все пакеты программно-информационного обеспечения, и в особенности драйверы устройств поставщиков, подготовлены для инсталлирования с помощью менеджеров пакетов. Без использования менеджера пакетов инсталляция программно-информационного обеспечения становится сложнее и подвержена ошибкам. Большинство программных пакетов высокопроизводительных вычислений для Linux распространяются в форматах Debian (.deb) либо Red Hat Package Manager (.rpm). Указанные форматы пакетов могут инсталлироваться в большинстве дистрибутивов Linux.

17.9.1 Менеджеры пакетов для macOS

Для операционной системы Mac (macOS) двумя основными менеджерами пакетов являются Homebrew и MacPorts. В общем случае оба варианта являются хорошим выбором для инсталлирования пакетов программно-информационного обеспечения. Поскольку macOS является производной от дистрибутива программно-информационного обеспечения Berkeley (BSD) Unix, имеется целый ряд инструментов с открытым исходным кодом. Но в связи с недавними изменениями, внесенными в macOS в целях повышения безопасности, некоторые инструменты отказались от поддержки последних версий платформы. А в связи с недавними изменениями в оборудовании Mac могут произойти значительные изменения и в управлении пакетами. Более подробная информация о Homebrew и MacPorts доступна на их соответствующих веб-сайтах:

- Homebrew по адресу https://brew.sh;
- MacPorts по адресу https://www.macports.org.

17.9.2 Менеджеры пакетов для Windows

В значительной степени проприетарная операционная система Windows долгое время была сборной солянкой для инсталляции и поддержки программно-информационного обеспечения. Некоторые программы поддерживаются хорошо, а другие – совсем нет. В Microsoft все меняется по мере того, как она принимает движение за открытый исходный код. Windows только сейчас выходит на вечеринку со своей новой подсистемой Windows для Linux (Windows Subsystem Linux, WSL). WSL настраи-

вает среду Linux в оболочке и должна позволять большинству программ Linux работать без изменений. Недавнее объявление о том, что WSL будет поддерживать прозрачный доступ к GPU, вызвало воодушевление в сообществе высокой производительности. Конечно же, главными целями являются игры и другие приложения массового рынка, но мы будем рады прокатиться за просто так, если это возможно.

17.9.3 Менеджер пакетов Spack: менеджер пакетов для высокопроизводительных вычислений

До сих пор мы обсуждали менеджеры пакетов, ориентированные на конкретные вычислительные платформы. Трудности, связанные с инструментом для высокопроизводительных вычислений, намного выше, чем у традиционных менеджеров пакетов, из-за более крупного числа операционных систем, оборудования и компиляторов, которые необходимо поддерживать одновременно. Решение этих проблем потребовало немало времени, пока до 2013 году Тодд Гэмблин (Todd Gamblin) из Национальной лаборатории Лоуренса Ливермора не выпустил менеджер пакетов Spack. Один из авторов этой книги внес пару пакетов в список, когда во всей системе было меньше дюжины пакетов. В настоящее время поддерживается более 4000 пакетов, и многие из них являются уникальными для сообщества высокопроизводительных вычислений.

Пример: краткое руководство по менеджеру пакетов Spack

В целях инсталлирования менеджера пакетов Spack следует набрать:

```
git clone https://github.com/spack/spack.git
```

Затем добавить в свою среду путь и скрипт настройки. Вы можете добавить их в свой файл ./bash_profile либо ./bashrc, чтобы иметь Spack готовым к работе в любое время.

```
export SPACK_ROOT=/path/to/spack
source $SPACK_ROOT/share/spack/setup-env.sh
```

Для того чтобы сконфигурировать Spack, сначала надо настроить Spack для ваших компиляторов:

```
spack compiler find
```

Если компилятор загружается из модуля, то следует добавить эту загрузку в конфигурацию компилятора

```
spack config edit compilers
```

либо отредактировать

```
~/.spack/linux/compiler.yaml
```

Возможно, вам захочется добавить в принятую по умолчанию конфигурацию несколько уже существующих системных пакетов, чтобы они не собирались. Для этого следует воспользоваться своим любимым редактором и отредактировать

```
~/.spack/linux/packages.yaml
```

Вы обнаружите много команд Spack. В табл. 17.3 приведено несколько из них, с которых можно начать.

Таблица 17.3 Использование Spack

Команда	Описание
`spack list`	Список имеющихся пакетов
`spack install <имя_пакета>`	Инсталлирует запрошенный пакет
`spack find`	Перечисляет пакеты, которые уже были собраны
`spack load <имя_пакета>`	Загружает пакеты в вашу среду

Spack располагает обширной документацией и активным сообществом разработчиков. Проверьте их веб-сайт на наличие актуальной информации: https://spack.readthedocs.io.

17.10 Modules: загрузка специализированных цепочек инструментов

Реалии разработки программно-информационного обеспечения на крупных вычислительных веб-сайтах таковы, что эти сайты должны поддерживать несколько сред одновременно. Благодаря этому вы можете загружать разные версии GCC и MPI для тестирования. Возможно, вы и сможете загружать эти разные цепочки инструментов разработки, но программные модули не проходят тщательное тестирование, которое проводится в большинстве дистрибутивов поставщиков.

ПРЕДУПРЕЖДЕНИЕ С программно-информационным обеспечением цепочки инструментов, инсталлируемым из пакета Modules, могут возникать ошибки. Однако преимущества высокопроизводительных приложений в значительной степени оправдывают потенциальные трудности.

Теперь давайте рассмотрим типичные команды, которые вы могли бы использовать с системой цепочки инструментов, инсталлируемой пакетом Modules, как показано в табл. 17.4.

Поскольку команда `module show` показывает исполняемые модулем действия, давайте рассмотрим пару примеров для комплекта компиляторов GCC и для CUDA.

Таблица 17.4. Команды пакета цепочки инструментов Modules: быстрый старт

Команда	Описание
`module avail`	Перечисляет модули, имеющиеся в системе
`module list`	Перечисляет модули, загруженные в вашу текущую среду
`module purge`	Выгружает все модули и восстанавливает среду до загрузки модулей
`module show <имя_модуля>`	Показывает изменения, которые будут внесены в вашу среду
`module unload <имя_модуля>`	Выгружает модуль и удаляет изменения в среде
`module swap <имя_модуля> <имя_модуля>`	Обменивает один пакет модулей на другой

Пример: команда module show gcc/9.3.0

```
/opt/modulefiles/centos7/gcc/9.3.0:
module-whatis     Эта команда загружает среду GCC 9.3.0.
prepend-path PATH /projects/opt/x86_64/gcc/9.3.0/bin
prepend-path LD_LIBRARY_PATH
    /opt/x86_64/gcc/9.3.0/lib64:/opt/x86_64/gcc/9.3.0/lib
prepend-path MANPATH /opt/x86_64/gcc/9.3.0/share/man
setenv     CC gcc
setenv     CXX g++
setenv     CPP cpp
setenv     FC gfortran
setenv     F77 gfortran
setenv     F90 gfortran
conflict     gcc
```

В этом примере модуль GCC v9.3.0 добавляет каталог GCC 9.3.0 в путь с помощью `prepend-path` и в среду с помощью переменной `LD_LIBRARY_PATH`. Он также задает несколько средовых переменных с помощью `setenv`, чтобы указать компилятор, который следует использовать.

Пример: команда module show cuda/10.2

```
/opt/modulefiles/centos7/cuda/10.2:
conflict    cuda
module-whatis    загрузить среду NVIDIA CUDA 10.2
module-whatis    Модифицирует: PATH, LD_LIBRARY_PATH
module-whatis    вАЖНО: библиотеки OpenCL
     инсталлируются драйвером NVIDIA, не этим модулем
setenv     CUDA_PATH /opt/centos7/cuda/10.2
setenv     CUDADIR /opt/centos7/cuda/10.2
setenv     CUDA_INSTALL_PATH /opt/centos7/cuda/10.2
setenv     CUDA_LIB /opt/centos7/cuda/10.2/lib64
setenv     CUDA_INCLUDE /opt/centos7/cuda/10.2/include
setenv     CUDA_BIN /opt/centos7/cuda/10.2/bin
prepend-path PATH /opt/centos7/cuda/10.2/bin
prepend-path LD_LIBRARY_PATH /opt/centos7/cuda/10.2/lib64
setenv OPENCL_LIBS /opt/centos7/cuda/10.2/lib64
```

```
setenv OPENCL_INCLUDE /opt/centos7/cuda/10.2/include
setenv     CUDA_SDK /opt/centos7/cuda/10.2/samples
```

В этом примере модуля CUDA задаются пути, включаемые каталоги и местоположения библиотек. Он также задает пути для имплементации NVIDIA OpenCL.

Как можно видеть из примеров этих команд пакета Modules, модули просто задают несколько переменных среды. Вот почему пакет Modules не защищен от неумелых действий. Ниже приведено несколько важных советов по использованию пакета Modules, которые мы усвоили на собственном горьком опыте. Мы начинаем со следующего ниже.

- *Важна согласованность*. Устанавливайте одни и те же модули для компилирования и выполнения вашего исходного кода. Если путь к библиотеке изменяется, то ваш исходный код может отказать либо выдавать неправильные результаты.
- *Максимально автоматизируйте работу*. Если вы этим пренебрежете, то ваша первая сборка (или выполнение) откажет, прежде чем вы поймете, что забыли загрузить свои модули.

Кроме того, существуют разные подходы к загрузке файлов модулей. Каждый из них наполнен преимуществами и недостатками. Эти подходы таковы:

- скрипты запуска оболочки;
- интерактивность в командной строке;
- скрипты пакетной отправки.

Используйте скрипты запуска интерактивной оболочки, а не скрипты пакетного запуска (например, загружайте Modules в файл .login вместо .cshrc). Параллельные задания распространяют свою среду на дистанционные узлы. Если вы загружаете Modules в неправильном скрипте запуска оболочки, то ваши дистанционные узлы могут иметь другие модули, чем головной узел. Это может иметь неожиданные последствия.

Используйте команду очистки `module purge` в пакетных скриптах перед загрузкой Modules. Если у вас Modules загружен, то загрузка модуля может завершиться отказом из-за конфликта, что потенциально может привести к отказу вашей программы. (Обратите внимание, что в системах Cray использование команды `module purge` бывает ненадежным.)

Задавайте пути выполнения в сборках программ. Встраивание путей выполнения в исполняемый файл с помощью опции `rpaths link` или других механизмов сборки помогает делать ваше приложение менее чувствительным к изменению среды и путей пакета Modules. Недостатком является то, что ваше приложение, возможно, не будет выполняться в другой системе, если компиляторы находятся в других местах. Обратите внимание, что этот технический прием не помогает получать неправильную версию программы, подобно mpirun, из вашей переменной `PATH`.

Загружайте конкретные версии компиляторов (например, GCC v9.3.0, а не только GCC). Нередко та или иная версия компилятора задается по умолчанию, но в какой-то момент это изменяется, нарушая работу вашего приложения или сборки. Кроме того, значения, принимаемые по умолчанию, не будут одинаковыми во всех системах.

Существует два основных пакета программно-информационного обеспечения, в которые имплементируются базовые команды пакета Modules. Первый называется module, часто именуемый модулями TCL, а второй – Lmod. Мы обсудим их в следующих далее разделах.

17.10.1 Модули TCL: изначальная система модулей для загрузки цепочек программных инструментов

Совершенно верно, все это сбивает с толку. Пакет Modules создал категорию, в которой теперь используется более или менее одно и то же имя – модуль. В 1991 году Джон Фурлани из Sun Microsystems создал module, а затем выпустил его в качестве программно-информационного обеспечения с открытым исходным кодом. Инструмент module написан на Командном языке инструментов, более известном как TCL от англ. *Tool Command Language*. Он зарекомендовал себя как важный компонент в крупных вычислительных центрах. Документация пакета module находится по адресу https://modules.readthedocs.io/en/stable/module.html.

17.10.2 Lmod: имплементация альтернативного пакета Modules на основе Lua

Lmod представляет собой систему Modules на основе языка Lua, которая настраивает среду пользователя динамически. Это более новая имплементация концепции средовых модулей. Документация lmod находится по адресу https://lmod.readthedocs.io/en/latest.

17.11 Размышления и упражнения

Мы хотели бы, чтобы у нас было время и пространство, чтобы подробнее изучить принципы использования каждого из этих инструментов. К сожалению, потребовалась бы еще одна книга (даже несколько книг), чтобы разведать мир инструментов для высокопроизводительных вычислений.

Мы прошлись по нескольким более простым инструментам, представив их мощь и полезность. Подобно тому, как вы не должны судить о книге по ее обложке, не судите об инструменте по причудливому интерфейсу. Вместо этого вам следует посмотреть на то, что конкретно делает этот инструмент и насколько он прост в использовании. Наш опыт показывает, что целью нередко становятся модные пользовательские интерфейсы, а не функциональность. Кроме того, инструменты должны

быть простыми. Нам надоело сталкиваться с очередным 600-страничным кратким руководством по началу работы, чтобы просто усвоить следующий инструмент. Да, инструмент может быть замечательным и творить удивительные вещи, но разработчику приложений также приходится осваивать много других вещей. Самые лучшие инструменты можно подбирать и делать полезными за пару часов.

Теперь мы передаем часть усилий вам, чтобы вы могли попробовать эти инструменты, и, надеюсь, вы найдете, что некоторые из них расширят ваш инструментарий разработчика. Добавление всего нескольких инструментов сделает вас лучше и эффективнее как программиста. Вот несколько упражнений, которые помогут вам начать.

1. Выполните инструмент Dr. Memory для одного из ваших небольших исходных кодов либо одного из исходных кодов из упражнений этой книги.
2. Скомпилируйте один из ваших исходных кодов с помощью библиотеки dmalloc. Выполните свой исходный код и просмотрите результаты.
3. Попробуйте вставить поточное гоночное условие в пример исходного кода раздела 17.6.2 и посмотрите, как Archer сообщает о проблеме.
4. Попробуйте выполнить профилировочное упражнение из раздела 17.8 в вашей файловой системе. Если у вас более одной файловой системы, то попробуйте его на каждой из них. Затем измените размер массива в примере на 2000×2000. Как это влияет на результаты работы файловой системы?
5. Инсталлируйте один из инструментов с помощью диспетчера пакетов Spack.

Резюме

- Более качественные практики разработки программно-информационного обеспечения начинаются с версионного контроля. Создание солидной среды разработки программно-информационного обеспечения приводит к более быстрой и качественной разработке исходного кода.
- Используйте таймеры и профилировщики для измерения производительности ваших приложений. Измерение производительности – это первый шаг на пути к повышению производительности приложений.
- Разведайте различные мини-приложения, чтобы увидеть примеры программирования, относящиеся к вашей области применения. Усвоение этих примеров поможет вам избегать повторного изобретения методов и улучшать свое приложение.
- Используйте инструменты, которые помогают выявлять проблемы в вашем приложении. За счет этого повышается качество и устойчивость вашей программы.

Приложение А
Справочные материалы

В конце каждой главы мы уже предоставили список дополнительных ресурсов, предложенных для получения дополнительной информации о темах, затронутых в этой главе. В каждой главе мы поместили материалы, которые, по нашему мнению, были бы наиболее ценными для большинства читателей. Справочные материалы в данном приложении к книге предназначены для тех, кто интересуется исходными материалами, которые были использованы при разработке книги. Приводимые ниже упоминания частично предназначены для того, чтобы отдать должное изначальным авторам научных и технических отчетов. Они также важны для тех, кто проводит более глубокие исследования по соответствующей теме.

A.1 Глава 1: зачем нужны параллельные вычисления?

- Амдал, Джин М. «Обоснованность однопроцессорного подхода для достижения крупномасштабных вычислительных возможностей» (Amdahl, Gene M. Validity of the single processor approach to achieving large scale computing capabilities. Proceedings of the April 18–20, 1967, Spring Joint Computer Conference. (1967):483–48), https://doi.org/10.1145/ 1465482.1465560.
- Флинн, Майкл Дж. «Некоторые компьютерные организации и их эффективность» (Flynn, Michael J. Some Computer Organizations and Their Effectiveness In *IEEE Transactions on Computers*, Vol. C-21, no. 9 (September, 1972): 948–960).

- Густафсон, Джон Л. «Переоценка закона Амдала» (Gustafson, John L. Reevaluating Amdahl's Law. In *Communications of the ACM*, Vol. 31, no. 5 (May, 1988):532–533), http://doi.acm.org/10.1145/ 42411.42415.
- Горовиц М., Лабонте Ф. И Рупп К. И соавт. «Данные о микропроцессорных трендах» (Horowitz, M., Labonte, F., and Rupp, K., et al. Microprocessor Trend Data. Accessed February 20, 2021), https://github.com/karlrupp/microprocessor-trend-data.

A.2 Глава 2: планирование параллелизации

- CMake, https://cmake.org/.

A.3 Глава 3: пределы производительности и профилирование

Инструменты

- Эмпирический инструментарий контура крыши (Empirical Roofline Toolkit (ERT)), https://bitbucket.org/berkeleylab/cs-roofline-toolkit.
- 4Intel® Advisor, https://software.intel.com/en-us/advisor.
- likwid, https://github.com/RRZE-HPC/likwid.
- Скачиваемый файл STREAM, https://github.com/jeffhammond/Stream.git.
- Valgrind, http://valgrind.org/.

Статьи

- Маккалпин, Дж. Д. «STREAM: устойчивая пропускная способность памяти в высокопроизводительных компьютерах» (McCalpin, J. D. STREAM: Sustainable Memory Bandwidth in High Performance Computers. Accessed February 20, 2021), https://www.cs.virginia.edu/stream/.
- Пейсе, Эльмар. «Моделирование производительности и предсказание для плотной линейной алгебры» (Peise, Elmar. Performance Modeling and Prediction for Dense Linear Algebra. arXiv:1706.01341 (June, 2017). Препринт: https://arxiv.org/abs/1706.01341.
- Уильямс, С. У., Д. Паттерсон и др. «Модель контура крыши: педагогический инструмент для автоматической настройки вычислительных мультъядер на мультиядерных процессорных архитектурах» (Williams, S. W., D. Patterson, et. al. The Roofline Model: A pedagogical tool for auto-tuning kernels on multicore architectures. In *Hot Chips, A Symposium on High Performance Chips*, Vol. HC20 (August 10, 2008)).

A.4 Глава 4: дизайн данных и модели производительности

Ресурсы

- Дизайн с ориентацией на данные, https://github.com/dbartolini/data-oriented-design.

Статьи и книги

- Берд, Р. «Исследование производительности массива структур из массивов» (Bird, R. Performance Study of Array of Structs of Arrays. Los Alamos National Lab (LANL)). Документ в стадии подготовки.
- Гаримелла, Рао и Роберт У. Роби. «Сравнительное исследование мультиматериальных структур данных для приложений вычислительной физики» (Garimella, Rao, and Robert W. Robey. A Comparative Study of Multi-material Data Structures for Computational Physics Applications, no. LA-UR-16-23889. Los Alamos National Lab (LANL), January, 2017).
- Хеннесси, Джон Л. И Дэвид А. Паттерсон. «Компьютерная архитектура: количественный подход. 5-е изд (Hennessy, John L., and David A. Patterson. *Computer architecture: A Quantitative Approach*. 5th ed. San Francisco, CA, USA: Morgan Kaufmann, 2011).
- Хофманн, Йоханнес, Ян Эйцингер и Дитмар Фей. «Модель производительности кеш-памяти исполнения: введение и валидация» (Hofmann, Johannes, Jan Eitzinger, and Dietmar Fey. Execution-Cache-Memory Performance Model: Introduction and Validation. arXiv:1509.03118, March, 2017). Препринт: https://arxiv.org/abs/1509.03118.
- Холлман, Дэвид, Брайс Лельбах, Х. Картер Эдвардс и соавт. «mdspan на C++: тематическое исследование по интеграции переносимых функций производительности в международные языковые стандарты» (Hollman, David, Bryce Lelbach, H. Carter Edwards, et al. mdspan in C++: A Case Study in the Integration of Performance Portable Features into International Language Standards. IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), November, 2019:60–70).
- Трейбиг, Ян и Георг Хагер. «Представляем модель производительности для вычислительных ядер с циклами, ограниченных по пропускной способности» (Treibig, Jan, and Georg Hager. Introducing a performance model for bandwidth-limited loop kernels. International Conference on Parallel Processing and Applied Mathematics (May, 2009):615–624.).

A.5 Глава 5: параллельные алгоритмы и шаблоны

- Аренс, Питер, Хонг Дип Нгуен и Джеймс Деммель. «Эффективное воспроизводимое суммирование с плавающей точкой и BLAS» (Ahrens, Peter, Hong Diep Nguyen, and James Demmel. Efficient Reproducible Floating Point Summation and BLAS. In *EECS Department, University of California, Berkeley, Techical Report*, No. UCB/EECS-2015-229, December, 2015).
- Алькантара, Дэн А., Андрей Шарф, Фатима Аббасинеджад и соавт. «Реально-временное параллельное хеширование на GPU» (Alcantara, Dan A., Andrei Sharf, Fatemeh Abbasinejad, et al. Real-time parallel hashing on the GPU. In *ACM Transactions on Graphics* (TOG), Vol. 28, no. 5 (December, 2009):154).
- Андерсон, Алисса. «Достижение численной воспроизводимости в распараллеленном продукте с плавающей точкой» (Anderson, Alyssa. Achieving Numerical Reproducibility in the Parallelized Floating Point Dot Product., April, 2014), https://digitalcommons.csbsju.edu/honors_theses/30/.
- Блеллох, Гай Э. «Сканы как примитивные параллельные операции» (Blelloch, Guy E. Scans as primitive parallel operations. In *IEEE Transactions on computers*, Vol. 38, no. 11 (November, 1989):1526–1538).
- Блеллох, Гай Э. «Векторные модели для вычислений параллельности данных» (Blelloch, Guy E. *Vector models for data-parallel computing*. Cambridge, MA, USA: The MIT Press, 1990).
- Чэпп, Дилан, Трэвис Джонстон и Микела Тауфер. «О необходимости воспроизводимой численной точности за счет интеллектуального выбора алгоритмов редукции во время выполнения в экстремальных масштабах» (Chapp, Dylan, Travis Johnston, and Michela Taufer. On the Need for Reproducible Numerical Accuracy through Intelligent Runtime Selection of Reduction Algorithms at the Extreme Scale. 2015 IEEE International Conference on Cluster Computing (October, 2015):166–175).
- Кливленд, Мэтью А., Томас А. Бруннер и соавт. «Получение идентичных результатов с глобальной точностью двойной прецизионности на разных числах процессоров в параллельных симуляциях частиц методом Монте-Карло» (Cleveland, Mathew A., Thomas A. Brunner, et al. Obtaining identical results with double precision global accuracy on different numbers of processors in parallel particle Monte Carlo simulations. In *Journal of Computational Physics*, Vol. 251 (October, 2013):223–236).
- Харрис, Марк, Шубхабрата Сенгупта и Джон Д. Оуэнс. «Параллельная префиксная сумма (скан) с CUDA» (Harris, Mark, Shubhabrata Sengupta, and John D. Owens. Parallel Prefix Sum (Scan) with CUDA. In *GPU Gems 3*, no. 39 (April, 2007):851–876).

- Лессли, Брентон. «Техники параллельного хеширования данных для архитектур GPU» (Lessley, Brenton. Data-Parallel Hashing Techniques for GPU Architectures. In *Eurographics Conference on Visualization* (EuroVis), Vol. 37, no. 3 (July, 2018)).

A.6 Глава 8: MPI: параллельный становой хребет

- Хефлер, Торстен и Йеспер Ларссон Трафф. «Разреженные коллективные операции для MPI» (Hoefler, Torsten, and Jesper Larsson Traff. Sparse collective operations for MPI. 2009 IEEE International Symposium on Parallel & Distributed Processing (July, 2009):18).
- Тхакур, Раджив и Уильям Гропп. «Комплект тестов для оценивания производительности мультипоточного обмена данными MPI» (Thakur, Rajeev, and William Gropp. Test suite for evaluating performance of multithreaded MPI communication. In *Parallel Computing*, Vol. 35, no. 12 (December, 2009):608–617).

A.7 Глава 9: архитектуры и концепции GPU-процессоров

- Янг, Чарлин, Торстен Курт и Сэмюэл Уильямс. «Иерархический анализ контуров крыши для GPU-процессоров: ускорение оптимизации производительности для системы NERSC-9 Perlmutter» (Yang, Charlene, Thorsten Kurth, and Samuel Williams. Hierarchical Roofline analysis for GPUs: Accelerating performance optimization for the NERSC-9 Perlmutter system. In *Concurrency and Computation: Practice and Experience* (November, 2019), https://doi.org/10.1002/cpe.5547.

A.8 Глава 10: модель программирования GPU

- Документация по инструментарию CUDA. «Вычислительные способности». Руководство по программированию на CUDA C++, v11.2.1 (CUDA Toolkit Documentation. Compute Capabilities. CUDA C++ Programming Guide, v11.2.1, NVIDIA Corporation, 2021), https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities.

A.9 Глава 12: языки GPU: обращение к основам

- Харрис, Марк. «Оптимизирование параллельной редукции на CUDA» (Harris, Mark. Optimizing Parallel Reduction in CUDA. NVIDIA

Corporation), https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf.

A.10 Глава 13: профилирование и инструменты GPU

- Новости Би-би-си. «Цунами в Индонезии: как вулкан становится спусковым крючком» (BBC News. Indonesia tsunami: How a volcano can be the trigger. BBC Global News Ltd, December, 2018), http://mng.bz/y92d.

A.11 Глава 14: аффинность: перемирие с вычислительным ядром

- Брокедис, Франсуа, Джером Клет-Ортега и соавт. «hwloc: универсальный каркас для управления аффинностью оборудования в приложениях HPC» (Broquedis, François, Jérôme Clet-Ortega, et al. hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP2010). IEEE Computer Society Press (February, 2010):180–186), https://ieeexplore.ieee.org/document/5452445.
- Hewlett Packard Enterprise, «Оригинальная программа размещения процессов, Руководство по размещению пользовательских приложений xthi.c (Hewlett Packard Enterprise, Original process placement program, xthi.c. CLE User Application Placement Guide (CLE 5.2.UP04) S-2496, pg 87), http://mng.bz/MgWB.
- «Интерфейс программирования приложений OpenMP», версия 5.0. Совет по пересмотру архитектуры OpenMP (OpenMP Application Programming Interface, v5.0. OpenMP Architecture Review Board, November, 2018), https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.
- Сэмюэл К. Гутьеррес, «Адаптивный параллелизм для состыкованных мультипоточных программ передачи сообщений» (Samuel K. Gutiérrez, Adaptive Parallelism for Coupled, Multithreaded Message-Passing Programs, December, 2018), https://www.cs.unm.edu/~samuel/publications/2018/skgutierrez-dissertation.pdf.
- Сэмюэл К. Гутьеррес, Дэвис, Кей и соавт. «Учет неоднородности уровня потоков в состыкованных параллельных приложениях» (Samuel K. Gutiérrez, Davis, Kei, et al. Accommodating Thread-Level Heterogeneity in Coupled Parallel Applications. Proceedings of the IEEE International Parallel and Distributed Processing Symposium, May, 2017), https://github.com/lanl/libquo/blob/master/docs/publications/quo-ipdps17.pdf.

- Сквайерс, Джефф. «Размещение процесса» (Squyres, Jeff. Process Placement, September, 2014. Accessed February 20, 2021), https://github.com/open-mpi/ompi/wiki/ProcessPlacement.
- Трейбиг, Дж., Г. Хагер и Г. Веллейн. «LIKWID: легковесный комплект инструментов с ориентацией на производительность, для мультиядерных сред на базе x86» (Treibig, J., G. Hager and G. Wellein. LIKWID: A lightweight performanceoriented tool suite for x86 multicore environments. arXiv:1004.4431, June, 2010). Препринт: http://arxiv.org/abs/1004.4431.

A.12 Глава 16: файловые операции для параллельного мира

Инструменты

- BeeGFS (Ведущая параллельная файловая система), https://www.beegfs.io/c/.
- Lustre®. OpenSFS и EOFS, http://lustre.org.
- Проект OrangeFS, http://www.orangefs.org.
- Параллельная файловая система Panasas PanFS, https://www.panasas.com/panfs-architecture/panfs/.

Статьи и книги

- Гропп, Уильям. «Лекция 33: подробнее о лучших практических приемах параллельного ввода-вывода и подсказках MPI-IO» (Gropp, William. Lecture 33: More on MPI I/O Best practices for parallel IO and MPI-IO hints. Accessed February 20, 2021), http://wgropp.cs.illinois.edu/courses/cs598-s15/lectures/lecture33.pdf.
- Мендес, Сандра, Себастьян Лурс и соавт. «Руководство по лучшим практическим приемам – Параллельный ввод-вывод» (Mendez, Sandra, Sebastian Lührs, et al. Best Practice Guide–Parallel I/O. Accessed February 20, 2021), https://prace-ri.eu/wp-content/uploads/Best-Practice-Guide_Parallel-IO.pdf.
- Тхакур, Раджив, Юинг Ласк и Уильям Гропп. «Руководство пользователя для ROMIO: высокопроизводительная переносимая имплементация MPI-IO» (Thakur, Rajeev, Ewing Lusk, and William Gropp. *Users guide for ROMIO: A highperformance, portable MPI-IO implementation*. ANL/MCS-TM-234. Artonne, IL, USA: Argonne National Laboratory, October, 1997).
- Тхакур, Раджив, Уильям Гропп и Юинг Ласк. «Просеивание данных и коллективный ввод-вывод в ROMIO» (Thakur, Rajeev, William Gropp, and Ewing Lusk. Data sieving and collective I/O in ROMIO. Proceedings. Frontiers' 99. Seventh Symposium on the Frontiers of Massively Parallel Computation, February, 1999:182–189).

A.13 Глава 17: инструменты и ресурсы для улучшения исходного кода

- Степанов, Евгений и Константин Серебряные. «MemorySanitizer: быстрый детектор использования неинициализированной памяти на C++» (Stepanov, Evgeniy, and Konstantin Serebryany. MemorySanitizer: fast detector of uninitialized memory use in C++. 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (February, 2015):46–55).

Приложение B
Решения упражнений

B.1 Глава 1. Зачем нужны параллельные вычисления?

1. Каковы другие примеры параллельных операций в вашей повседневной жизни? Как бы вы классифицировали свой пример? Под что, по вашему мнению, оптимизируется параллельный дизайн? Можете ли вы вычислить параллельное ускорение для этого примера?

   Ответ: примеры параллельных операций в повседневной жизни включают многополосные автомагистрали, очереди на регистрацию классов и доставку почты и многое другое.

2. Какова теоретическая мощность параллельной обработки вашей системы (будь то настольный компьютер, ноутбук или мобильный телефон) по сравнению с ее мощностью последовательной обработки? Какие виды параллельного оборудования в ней присутствуют?

   Ответ: трудно проникнуть в суть маркетинга и шумихи и найти реальные технические характеристики. Большинство устройств, включая портативные, имеют мультиядерные процессоры и по крайней мере встроенный графический процессор. Настольные компьютеры и ноутбуки обладают некоторыми векторными возможностями, за исключением очень старого оборудования.

3. Какие параллельные стратегии вы видите в примере с оплатой покупок в магазине на рис. 1.1? Существуют ли какие-то нынешние

параллельные стратегии, которые не показаны? Как насчет примеров из упражнения 1?

Ответ: несколько команд, несколько элементов данных (MIMD), распределенные данные, конвейерный параллелизм и внепорядковое исполнение со специализированными очередями.

4 У вас есть приложение для манипулирования изображениями, которому необходимо ежедневно обрабатывать 1000 изображений размером 4 мебибайт (МиБ, 2^{20}, или 1 048 576 байт) каждое. Для последовательной обработки каждого изображения требуется 10 мин. Ваш кластер состоит из мультиядерных узлов с 16 ядрами и суммарным объемом 16 гибибайт (ГиБ, 230 байт, или 1024 мебибайт) основной памяти в расчете на узел. (Обратите внимание, что мы используем правильные двоичные термины МиБ и ГиБ, а не Мб и Гб, которые являются метрическими терминами соответственно для 10^6 и 10^9 байт.)

   a Какой дизайн параллельной обработки лучше всего справляется с этой рабочей нагрузкой?

   b Теперь потребительский спрос увеличивается в 10 раз. Справляется ли с этим ваш дизайн? Какие изменения вам пришлось бы внести?

   Ответ: потокообразование на одном вычислительном узле вместе с векторизацией. 4 МиБ × 1000 = 4 Гб. Но для обработки 16 изображений за один раз требуется всего 64 МиБ, что значительно меньше 1 ГиБ на каждом узле (рабочей станции) кластера. Время составит 10 мин × 1000 или 167 мин в последовательном режиме и 10.4 мин на 16 ядрах в параллельном режиме. Векторизация может сократить это время менее чем до 5 мин. Увеличение спроса в 10 раз составило бы эти 100 мин. Это, может быть, и хорошо, но и стоит подумать о передаче сообщений или распределенных вычислениях.

5 Процессор Intel Xeon E5-4660 имеет расчетную тепловую мощность 130 Вт; это средняя потребляемая мощность при использовании всех 16 ядер. GPU NVIDIA Tesla V100 и GPU AMD MI25 Radeon имеют расчетную тепловую мощность 300 Вт. Предположим, вы портируете свое программное обеспечение для использования одного из этих GPU. Насколько быстрее должно работать ваше приложение на GPU, чтобы считаться более энергоэффективным, чем ваше приложение с 16-ядерным CPU?

   Ответ: 300 Вт / 130 Вт. Для того чтобы быть более энергоэффективным, оно должно иметь ускорение в 2.3 раза.

B.2 Глава 2: планирование параллелизации

1 У вас есть приложение с симуляцией высоты волн, которое вы разработали во время учебы в аспирантуре. Это последовательное приложение, и, поскольку оно планировалось только как основа

для вашей диссертации, вы не встраивали в него никаких технических приемов разработки программного обеспечения. Теперь вы планируете использовать его в качестве отправной точки для имеющегося инструмента, который может использоваться многими исследователями. В вашем коллективе есть еще три разработчика. Что бы вы для них включили в свой план проекта?

Ответ: подготовительные шаги будут включать:

- организацию системы версионного контроля с помощью Git;
- создание набора тестов с известными результатами;
- выполнение инструментов правильности памяти для тестов в комплекте тестов;
- профилирование оборудования и вашего приложения;
- создание плана для следующего шага в вашей стратегии гибкой методологии управления.

2 Создайте тест с помощью CTest.

Ответ: поскольку CTest обнаруживает статус любой ошибки из команды, тест может быть сделан из сборочной инструкции. Иногда инсталлирование файлов CTest лишает исполняемый бит разрешений и приводит к отказу теста без четкого сообщения об ошибке. Во избежание этого мы можем добавить тест для определения исполняемости скрипта CTest. В следующем ниже исходном коде `$0` – это скрипт CTest с полным путем, для того чтобы он работал для сборок вне дерева.

- Добавьте в CMakeLists.txt.

```
enable_testing()

add_test(NAME make WORKING_DIRECTORY ${CMAKE_BINARY_DIRECTORY}
            COMMAND ${CMAKE_CURRENT_SOURCE_DIR}/build.ctest)
```

- Добавьте файл build.ctest со следующим исходным кодом:

```
#!/bin/sh
if [ -x $0 ]
then
   echo "ПРОЙДЕНО - скрипт исполняем"
else
   echo "Отказ - скрипт ctest не исполняем"
   exit -1
fi
```

3 Исправьте ошибки памяти в листинге 2.2.

Ответ: вы можете исправить ошибки памяти в листинге 2.2, изменив или добавив следующие ниже строки:

```
 4    int ipos=0, ival;
 7    for (int i = 0; i<10; i++){ iarray[i] = ipos; }
 8    for (int i = 0; i<10; i++){
11    free(iarray);
```

4 Выполните valgrind в небольшом приложении по вашему выбору.

Ответ: по своему усмотрению.

B.3 Глава 3: пределы производительности и профилирование

1 Рассчитайте теоретическую производительность системы по вашему выбору. Включите в расчет пиковые флопы, пропускную способность памяти и машинный баланс.

Ответ: по своему усмотрению.

2 Скачайте инструментарий Roofline Toolkit с https://bitbucket.org/berkeleylab/cs-roofline-toolkit.git и измерьте фактическую производительность выбранной вами системы.

Ответ: по своему усмотрению.

3 Используя инструментарий Roofline Toolkit, начните с одного процессора и постепенно добавляйте оптимизацию и параллелизацию, записывая величину улучшения на каждом шаге.

Ответ: по своему усмотрению.

4 Скачайте приложение сравнительного тестирования STREAM Benchmark с https://www.cs.virginia.edu/stream/ и измерьте пропускную способность памяти выбранной вами системы.

Ответ: по своему усмотрению.

5 Выберите один из общедоступных сравнительных тестов или мини-приложений, перечисленных в разделе 17.1, и сгенерируйте граф вызовов, используя KCachegrind.

Ответ: по своему усмотрению.

6 Выберите один из общедоступных сравнительных тестов или мини-приложений, перечисленных в разделе 17.1, и измерьте его арифметическую интенсивность, используя инструменты Intel Advisor или likwid.

Ответ: по своему усмотрению.

7 Используя представленные в этой главе средства расчета производительности, определите среднюю частоту и энергопотребление процессора для небольшого приложения.

Ответ: по своему усмотрению.

8 Используя некоторые инструменты из раздела 3.3.3, определите объем памяти, потребляемый приложением.

Ответ: по своему усмотрению.

B.4 Глава 4: дизайн данных и модели производительности

1 Напишите двухмерный выделитель сплошной памяти под левую нижнюю треугольную матрицу.

Ответ: в листинге B.4.1 показан исходный код для выделения нижнего левого треугольного массива. Допустим, что индексирование массива равно C, а нижний левый элемент равен `[0][0]`. Кроме того, матрица должна быть квадратной. Мы используем тот же исходный код, что и в листинге 4.3, но с уменьшением длины `imax` на 1 для каждой строки. Обратите внимание, что число элементов в треугольном массиве может быть рассчитано как `jmax*(imax+1)/2`.

Листинг B.4.1 Выделение треугольной матрицы

ExerciseB.4.1/malloc2Dtri.c

```
 1 #include <stdlib.h>
 2 #include "malloc2Dtri.h"
 3
 4 double **malloc2Dtri(int jmax, int imax)
 5 {
 6    double **x =
         (double **)malloc(jmax*sizeof(double *) +
 7       jmax*(imax+1)/2*sizeof(double));
 8
 9    x[0] = (double *)(x + jmax);
10
11    for (int j = 1; j < jmax; j++, imax--) {
12       x[j] = x[j-1] + imax;
13    }
14
15    return(x);
16 }
```

Сначала выделить блок памяти для указателей строк и двухмерного массива

Теперь назначить начало блока памяти для двухмерного массива после указателей строк

Уменьшать imax на 1 во время каждой итерации

Наконец, назначить ячейку памяти, на которую будет указывать указатель каждой строки

2 Напишите двухмерный выделитель на C, который компонует память так же, как Fortran.

Ответ: давайте допустим, что мы хотим обращаться к массиву как `x(j,i)` в языке Fortran. Массив будет адресоваться как `x[i][j]` на C. Если мы создадим макрокоманду `#define x(j,i) x[i-1][j-1]`, то исходный код может использовать Fortran'овскую нотацию массива. Выделитель двухмерной памяти из листинга 4.3 можно использовать путем перемены мест `i` и `j`, а также `imax` и `jmax`. В следующем ниже листинге показан результирующий исходный код.

Листинг B.4.2 Выделитель треугольной матрицы

Exercise4.2/malloc2Dfort.c

```
 1 #include <stdlib.h>
 2 #include "malloc2Dfort.h"
 3
 4 double **malloc2Dfort(int jmax, int imax)
 5 {
 6    double **x =
         (double **)malloc(imax*sizeof(double *) +
 7       imax*jmax*sizeof(double));
 8
 9    x[0] = (double *)(x + imax);
10
11    for (int i = 1; i < imax; i++) {
12       x[i] = x[i-1] + jmax;
13    }
14
15    return(x);
16 }
```

Сначала выделить блок памяти для указателей столбцов и двухмерного массива

Теперь назначить начало блока памяти для двухмерного массива после указателей столбцов

Наконец, назначить ячейку памяти, на которую будет указывать указатель каждой столбца

3 Создайте макрокоманду для массива структур из массивов (AoSoA) для цветовой модели RGB из раздела 4.1.

Ответ: мы хотим извлекать данные с обычным индексом массива и названием цвета:

```
#define VV = 4
#define color(i,C) AOSOA[(i)/VV].C[(i)%4-1]
color(50,B)
```

4 Модифицируйте код для ячеечно-центричной полноматричной структуры данных, чтобы не использовать условный переход, и оцените ее производительность.

Ответ: на следующем ниже рисунке показан исходный код с удаленной инструкцией `if`. Из этого модифицированного исходного кода счетчики модели производительности выглядят следующим образом:

$$\text{Memops} = 2 \times N_c N_m + 2 \times N_c = 102 \text{ M Мемопов}.$$

1: **for all** ячейки, C, вплоть до N_c **do**;
2: $ave \leftarrow 0.0$;
3: **for all** идентификаторы материалов, m, вплоть до N_m **do**;
4: $ave \leftarrow ave + \rho[C][m] * f[C][m]$ # $2N_cN_m$ загрузок (ρ, f)
$2N_cN_m$ флопов (+, *);
5: **end for**;
6: $\rho_{ave}[C] \leftarrow ave/V[C]$ # N_c созранений (ρ_{ave}), N_c загрузок (V)
N_c флопов (/);
7: **end for**

Модель производительности = 61.0 мс. Указанная оценка производительности немного быстрее, чем версия с инструкцией if

5 Каким образом векторный модуль AVX-512 изменил бы модель ECM для потоковой триады?

Ответ: в анализе производительности с помощью модели ECM в разделе 4.4 используется векторный модуль AVX-256, который может обрабатывать все необходимые операции с плавающей точкой за один цикл. Модулю AVX-512 все равно потребуется один цикл, но будет занята только половина его векторных модулей, и он мог бы выполнять вдвое больше работы, если бы присутствовал. Поскольку время вычислительной операции, T_{OL}, остается на уровне одного цикла, производительность вообще не изменится.

B.5 Глава 5: параллельные алгоритмы и шаблоны

1 Модель столкновения облаков в шлейфе пепла вызывается для частиц на расстоянии в пределах 1 мм. Напишите псевдокод для имплементации пространственного хеша. В каком порядке сложности выполняется эта операция?

Ответ: псевдокод для операции столкновения выглядит следующим образом:

```
1. Сгруппировать частицы в 1-мм пространственные корзины
2. Для каждой корзины
3.    Для каждой частицы, i, в корзине
4.       Для всех других частиц, j, в корзине либо смежных корзинах
5.          если |P_i - P_j| < 1 мм
6.             вычислить коллизию
```

Операция имеет $O(N^2)$ в локальном участке, но по мере увеличения вычислительной (расчетной) сетки расстояние между частицами не нужно вычислять для более крупных участков, и, следовательно, операция приближается к $O(N)$.

2 Как пространственные хеши используются почтовой службой?

Ответ: почтовые индексы. Функция хеширования кодирует штат и регион в первых трех цифрах, а в оставшихся двух сначала кодируются крупные города, а затем остальное в алфавитном порядке.

3 Большие данные используются алгоритмом map-reduce (отображения-редукции) для эффективной обработки крупных наборов данных. Чем он отличается от представленных здесь концепций хеширования?

Ответ: несмотря на то что он разработан для разных задачных областей и масштабов, операция отображения (map) в алгоритме map-reduce представляет собой хеш. Таким образом, они обе выполняют шаг хеширования, за которым следует вторая локальная

операция. Пространственный хеш имеет концепцию связи по расстоянию между ячейками, тогда как map-reduce по своей сути ее не имеет.

4 В исходном коде волновой симуляции используется сетка AMR для более глубокой детализации береговой линии. Требования к симуляции заключаются в регистрации высоты волн в зависимости от времени для заданных мест, где расположены буи и береговые сооружения. Поскольку ячейки постоянно детализируются, как вы могли бы ее имплементировать?

Ответ: создайте идеальный пространственный хеш с размером корзины, таким же, как у самой малой ячейки, и сохраняйте индекс ячейки в корзинах, лежащих в основе ячейки. Рассчитайте корзину для каждой станции и получите индекс ячейки из корзины.

B.6 Глава 6: векторизация: флопы забесплатно

1 Поэкспериментируйте с циклами автоматической векторизации из мультиматериального исходного кода в разделе 4.3 (https://github.com/LANL/MultiMatTest.git). Добавьте флаги отчетности о векторизации и цикле и посмотрите, что конкретно ваш компилятор вам скажет.

Ответ: по своему усмотрению.

2 Добавьте прагмы OpenMP SIMD в цикл, выбранный вами в первом упражнении, чтобы помочь компилятору выполнять векторизацию циклов.

Ответ: по своему усмотрению.

3 В одном из примеров с внутренними векторными функциями измените длину вектора с четырех значений двойной точности на вектор шириной 8. Просмотрите исходный код этой главы с примерами рабочего кода для имплементаций с шириной 8.

Ответ: в kahan_fog_vector.cpp поменяйте `4s` на `8s` и поменяйте `Vec4d` на `Vec8d`. Добавьте `mprefer-vector-width=512 -DMAX_VECTOR_SIZE=512` в `CXXFLAGS`. Измененный исходный код и файл Makefile включены в исходный код этой главы.

4 Если вы используете более старый процессор, то насколько успешно выполняется ваша программа из упражнения 3? Каково влияние такого выбора на производительность?

Ответ: для 256-битовых векторных модулей Intel встроенные компоненты Intel не работают и должны быть закомментированы. Однако версии GCC и Fog все еще работают. Результаты хронометража из ноутбука Mac 2017 года показывают превосходство библиотеки векторных классов Агнера Фога, в которой восемь векторов дают более высокие результаты, чем четыре. В отличие от этого импле-

ментация GCC для вектора шириной восемь медленнее, чем версия шириной 4. Вот результат:

```
SETTINGS INFO -- ncells 1073741824 log 30
Initializing mesh with Leblanc problem, high values first
 relative diff runtime Description
   8.423e-09 1.461642 Serial sum
           0    3.283697   Kahan sum with double double accumulator
4 wide vectors serial sum
  -3.356e-09    0.408654   Serial sum (OpenMP SIMD pragma)
  -3.356e-09    0.407457   Intel vector intrinsics Serial sum
  -3.356e-09    0.402928   GCC vector intrinsics Serial sum
  -3.356e-09    0.406626   Fog C++ vector class Serial sum
4 wide vectors Kahan sum
           0    0.872013   Intel Vector intrinsics Kahan sum
           0    0.873640   GCC vector extensions Kahan sum
           0    0.872774   Fog C++ vector class Kahan sum
8 wide vector serial sum
  -1.986e-09    1.467707   8 wide GCC vector intrinsic Serial sum
  -1.986e-09    0.586075   8 wide Fog C++ vector class Serial sum
8 wide vector Kahan sum
  -1.388e-16    1.914804   8 wide GCC vector extensions Kahan sum
  -1.388e-16    0.545128   8 wide Fog C++ vector class Kahan sum
  -1.388e-16    0.687497   Agner C++ vector class Kahan sum
```

B.7 Глава 7: высокая производительность OpenMP

1 Конвертируйте пример векторного сложения из листинга 7.8 в OpenMP высокого уровня, следуя инструкциям раздела 7.2.2.

Ответ: при конвертировании в OpenMP высокого уровня мы получаем исходный код, показанный в следующем ниже листинге, в котором параллельный участок открывается всего одной прагмой.

Листинг B.7.1 OpenMP высокого уровня

ExerciseB.7.1/vecadd.c

```
11 int main(int argc, char *argv[]){
12    #pragma omp parallel
13    {
14       double time_sum;
15       struct timespec tstart;
16       int thread_id = omp_get_thread_num();
17      int nthreads = omp_get_num_threads();
18       if (thread_id == 0){
19          printf("Выполняется с %d потоком(потоками)\n",nthreads);
20       }
```

```
      int tbegin = ARRAY_SIZE * ( thread_id     ) / nthreads;
      int tend   = ARRAY_SIZE * ( thread_id + 1 ) / nthreads;

      for (int i=tbegin; i<tend; i++) {
         a[i] = 1.0;
         b[i] = 2.0;
      }

      if (thread_id == 0) cpu_timer_start(&tstart);
      vector_add(c, a, b, ARRAY_SIZE);
      if (thread_id == 0) {
         time_sum += cpu_timer_stop(tstart);
         printf("Время выполнения составляет %lf мс\n", time_sum);
      }
   }
}

void vector_add(double *c, double *a, double *b, int n)
{
   int thread_id = omp_get_thread_num();
   int nthreads = omp_get_num_threads();
   int tbegin = n * ( thread_id     ) / nthreads;
   int tend   = n * ( thread_id + 1 ) / nthreads;
   for (int i=tbegin; i < tend; i++){
      c[i] = a[i] + b[i];
   }
}
```

2 Напишите процедуру для получения максимального значения в массиве. Добавьте прагму OpenMP, чтобы добавить в процедуру поточный параллелизм.

Ответ: в процедуре редукции используется выражение `reduction (max:xmax)`, как показано в следующем ниже листинге.

Листинг B.7.2 Редукция OpenMP на основе максимума

ExerciseB.7.2/max_reduction.c

```
#include <float.h>
double array_max(double* restrict var, int ncells)
{
   double xmax = DBL_MIN;
   #pragma omp parallel for reduction(max:xmax)
   for (int i = 0; i < ncells; i++){
      if (var[i] > xmax) xmax = var[i];
   }
}
```

3 Напишите версию редукции из предыдущего упражнения на основе OpenMP высокого уровня.

Ответ: в OpenMP высокого уровня мы подразделяем данные вручную. Разложение данных выполняется в строках 6–9 листинга B.7.3. Поток 0 выделяет совместный массив данных `xmax_thread` в строке 13. В строках 18–22 отыскивается максимальное значение для каждого потока, и результат сохраняется в массиве `xmax_thread`. Затем в строках 26–30 один поток отыскивает максимум во всех потоках.

Листинг B.7.3 OpenMP высокого уровня

ExerciseB.7.3/max_reduction.c

```
 1 #include <stdlib.h>
 2 #include <float.h>
 3 #include <omp.h>
 4 double array_max(double* restrict var, int ncells)
 5 {
 6    int nthreads = omp_get_num_threads();
 7    int thread_id = omp_get_thread_num();
 8    int tbegin = ncells * ( thread_id ) / nthreads;
 9    int tend = ncells * ( thread_id + 1 ) / nthreads;
10    static double xmax;
11    static double *xmax_thread;
12    if (thread_id == 0){
13       xmax_thread = malloc(nthreads*sizeof(double));
14       xmax = DBL_MIN;
15    }
16 #pragma omp barrier
17
18    double xmax_thread_private = DBL_MIN;
19    for (int i = tbegin; i < tend; i++){
20       if (var[i] > xmax_thread_private) xmax_thread_private = var[i];
21    }
22    xmax_thread[thread_id] = xmax_thread_private;
23
24 #pragma omp barrier
25
26    if (thread_id == 0){
27       for (int tid=0; tid < nthreads; tid++){
28          if (xmax_thread[tid] > xmax) xmax = xmax_thread[tid];
29       }
30    }
31
32 #pragma omp barrier
33
34    if (thread_id == 0){
35       free(xmax_thread);
36    }
37    return(xmax);
38 }
```

B.8 Глава 8: MPI: параллельный становой хребет

1. Почему мы не можем просто блокировать на приемках, как это делалось при отправке/приемке в обмене призраками, используя методы упаковочного буфера или буфера массива соответственно в листингах 8.20 и 8.21?

   Ответ: версия, использующая упаковочный буфер или буфер массива, планирует отправку, но возвращается до того, как данные будут скопированы или отправлены. Стандарт для `MPI_Isend` гласит: «*Отправитель не должен модифицировать какую-либо часть буфера отправки после вызова неблокирующей операции отправки до завершения отправки*». Версии с упаковкой и массивом высвобождают буферы после обмена данными. Таким образом, эти версии могут удалять буферы до их копирования, что приводит к отказу программы. Для того чтобы быть в безопасности, статус отправки должен проверяться до удаления буфера.

2. Безопасно ли блокировать приемку, как показано в листинге 8.8 в версии обмена призраками с использованием векторного типа? в чем преимущества, если мы блокируем только приемку?

   Ответ: векторная версия отправляет данные из изначальных массивов вместо создания копии. Это безопаснее, чем версии, выделяющие буфер, который будет высвобожден. Если мы будем блокировать только на приемке, то обмен данными может быть быстрее.

3. Модифицируйте пример обмена призрачными ячейками с векторным типом в листинге 8.21, чтобы использовать блокировку приемки вместо ожидания. Будет ли это быстрее? Всегда ли это будет работать?

   Ответ: даже с векторной версией обмена призрачными ячейками мы должны не забывать о том, что нельзя модифицировать буферы, которые все еще находятся в процессе отправки. Вероятность того, что это произойдет, может быть невелика, когда мы не отправляем углы. Но это все еще может произойти. Для того чтобы быть в абсолютной безопасности, нам нужно проверять завершение отправки перед изменением массивов.

4. Попробуйте заменить явные теги в одной из подпрограмм обмена призраками на `MPI_ANY_TAG`. Будет ли это работать? Будет ли это хоть немного быстрее? Какое преимущество вы видите в использовании явных тегов?

   Ответ: использование `MPI_ANY_TAG` в качестве тегового аргумента прекрасно работает. Это приводит к небольшому ускорению, хотя маловероятно, что оно будет достаточно значительным, чтобы его можно было измерить. Использование явно заданных тегов добавляет еще одну проверку получения правильного сообщения.

5. Удалите барьеры для синхронизированных таймеров в одном из примеров обмена призраками. Выполните исходный код с изна-

чальными синхронизированными таймерами и несинхронизированными таймерами.

Ответ: устранение барьеров в таймерах должно повысить производительность и позволить процессам работать более независимо («асинхроннее»). Однако в этом случае бывает сложнее понять показания хронометража.

6 Добавьте статистику таймера из листинга 8.11 в исходный код измерения пропускной способности потоковой триады в листинге 8.17.

Ответ: по своему усмотрению.

7 Примените шаги для конвертирования OpenMP высокого уровня в пример гибридной техники MPI плюс OpenMP в исходном коде, прилагаемом к главе (каталог HybridMPIPlusOpenMP). Поэкспериментируйте с векторизацией, числом потоков и рангами MPI на вашей платформе.

Ответ: по своему усмотрению.

B.9 Глава 9: архитектуры и концепции GPU-процессоров

1 В табл. 9.7 показана достижимая производительность для приложения с одним флопом за загрузку. Посмотрите на текущие рыночные цены на GPU и заполните последних два столбца, чтобы получить флоп за доллар для каждого GPU. Каким выглядит самое лучшее соотношение цены и качества? Если время выполнения вашего приложения является наиболее важным критерием, то какой GPU лучше всего приобрести?

Таблица 9.7 Достижимая производительность приложения с одним флопом за загрузку с разными GPU

GPU	Достижимая производительность Гфлопы/с	Цена	Флопы/$
V100	108.23		
Vega 20	91.38		
P100	74.69		
GeForce GTX1080Ti	44.58		
Quadro K6000	31.25		
Tesla S2050	18.50		

Ответ: по своему усмотрению.

2 Измерьте потоковую пропускную способность вашего GPU или другого выбранного GPU. Как это соотносится с теми, которые представлены в этой главе?

Ответ: по своему усмотрению.

3 Используйте инструмент производительности likwid, чтобы узнать требования к мощности процессора для приложения CloverLeaf в системе, в которой у вас есть доступ к счетчикам силового оборудования.

Ответ: по своему усмотрению.

B.10 Глава 10: модель программирования GPU

1 У вас есть приложение для классифицирования фотоснимков, которое будет занимать 5 мс для передачи каждого файла на GPU, 5 мс для обработки и 5 мс для возврата. На CPU обработка занимает 100 мс на фотоснимок. Нужно обработать миллион фотоснимков. У вас на CPU есть 16 обрабатывающих ядер. Будет ли система на базе GPU работать быстрее?

Ответ.
Время на CPU – 100 мс × 1000000/16 /1000 = 6250 с.
Время на GPU – (5 мс + 5 мс + 5 мс) × 1000000/1000 = 15000 с.
Система GPU не будет работать быстрее. Это заняло бы примерно в 2.5 раза больше времени.

2 Время передачи у GPU в задаче 1 основано на шине PCI третьего поколения. Если вы сможете получить шину PCI Gen4, то как она изменит дизайн? а шина PCI Gen5? Для классифицирования фотоснимков вам не нужно возвращать модифицированный фотоснимок. Как это поменяет расчеты?

Ответ: шина PCI четвертого поколения работает в два раза быстрее, чем шина PCI третьего поколения.

$$(2.5 \text{ мс} + 5 \text{ мс} + 2.5 \text{ мс}) \times 1\,000\,000 / 1000 = 10000 \text{ с}.$$

Шина PCI пятого поколения будет в четыре раза быстрее, чем изначальная шина PCI третьего поколения.

$$(1.25 \text{ мс} + 5 \text{ мс} + 1.25 \text{ мс}) \times 1\,000\,000 / 1000 = 7500 \text{ с}.$$

Если нам не нужно передавать результаты обратно, то теперь на GPU мы работаем так же быстро, как и на CPU.

$$(1.25 \text{ мс} + 5 \text{ мс}) \times 1\,000\,000 / 1000 = 6250 \text{ с}.$$

3 Какого размера трехмерное приложение вы могли бы выполнить для вашего дискретного GPU (или NVIDIA GeForce GTX 1060, если такового нет)? Допустим, что на ячейку приходится 4 переменных двойной точности и что есть лимит использования половины памяти GPU, чтобы иметь место для временных массивов. Как это изменит приложение, если вы используете одинарную точность?

Ответ: NVIDIA GeForce GTX 1060 имеет объем памяти 6 ГиБ. Он оснащен GDDR5 с шиной шириной 192 бита и тактовой частотой 8 ГГц.

(6 ГиБ / 2 / 4 числа двойной точности / 8 байт × 10243)1/3 = трехмерная вычислительная сетка размера 465×465×465.

Для одинарной точности (прецизионности):

(6 ГиБ / 2 / 4 числа с плавающей точкой / 4 байта × 10243)1/3 = трехмерная вычислительная сетка размера 586×586×586.

Если мы разделим нашу вычислительную область на эту трехмерную вычислительную сетку, то это улучшит разрешающую способность на 25 %.

B.11 Глава 11: программирование GPU на основе директив

1. Найдите компиляторы, которые пригодны для вашей локальной системы GPU. Оба ли компилятора – языка OpenACC и языка OpenMP – доступны в вашем случае? Если нет, то есть ли у вас доступ к каким-либо системам, которые позволили бы вам опробовать эти прагма-ориентированные языки?

   Ответ: по своему усмотрению.

2. Выполните примеры потоковой триады из каталогов OpenACC/StreamTriad и/или OpenMP/StreamTriad в локальной системе разработки на основе GPU-процессоров. Вы найдете эти каталоги по адресу https://github.com/EssentialsofParallelComputing/Chapter11.

   Ответ: по своему усмотрению.

3. Сравните свои результаты упражнения 2 с результатами BabelStream по адресу https://uob-hpc.github.io/BabelStream/results/. Для потоковой триады перемещенные байты равны `3 * nsize * sizeof (тип данных)`.

   Ответ: из результатов производительности в главе по GPU NVIDIA V100.

   3 × 20 000 000 × 8 байт/.586 мс) × (1000 мс/с) / (1 000 000 000 байт/Гб) = 819 Гб/с.

   Это примерно на 50 % больше, чем пик, показанный в тесте BabelStream для GPU NVIDIA P100.

4 Модифицируйте отображение участка данных OpenMP в листинге 11.16, чтобы отразить фактическое использование массивов в вычислительных ядрах.

Ответ: массивы используются только на GPU, поэтому их можно выделить там и удалить в конце. Следовательно, изменения таковы:

```
13 #pragma omp target enter data map(alloc:a[0:nsize], b[0:nsize],
                                     c[0:nsize])
36 #pragma omp target exit data map(delete:a[0:nsize], b[0:nsize],
                                     c[0:nsize])
```

Полный листинг этого изменения приведен в Stream_par7.c в примерах главы.

5 Имплементируйте пример массовой суммы из листинга 11.4 в OpenMP.

Ответ: нам просто нужно изменить одну прагму, как показано в следующем ниже листинге.

Листинг B.11.5 GPU-версия OpenMP

ExerciseB.11.5/mass_sum.c

```
 1 #include "mass_sum.h"
 2 #define REAL_CELL 1
 3
 4 double mass_sum(int ncells, int* restrict celltype,
 5       double* restrict H, double* restrict dx, double* restrict dy){
 6    double summer = 0.0;
 7 #pragma omp target teams distribute \
              parallel for simd reduction(+:summer)
 8   for (int ic=0; ic<ncells ; ic++) {
 9       if (celltype[ic] == REAL_CELL) {
10          summer += H[ic]*dx[ic]*dy[ic];
11       }
12    }
13    return(summer);
14 }
```

6 Для массивов x и y размером 20 000 000 найдите максимальный радиус в массивах, используя как OpenMP, так и OpenACC. Инициализируйте массивы значениями двойной точности, которые линейно увеличиваются с 1.0 до 2.0e7 для массива x и уменьшаются с 2.0e7 до 1.0 для массива y.

Ответ: в следующем ниже листинге показана возможная имплементация поиска максимального радиуса с помощью OpenACC.

Листинг B.11.6 Версия OpenACC с отысканием максимального радиуса

ExerciseB.11.6/MaxRadius.c or Chapter11/OpenACC/MaxRadius/MaxRadius.c

```
#include <stdio.h>
#include <math.h>
#include <openacc.h>

int main(int argc, char *argv[]){
   int ncells = 20000000;
   double* restrict x = acc_malloc(ncells * sizeof(double));
   double* restrict y = acc_malloc(ncells * sizeof(double));

   double MaxRadius = -1.0e30;
#pragma acc parallel deviceptr(x, y)
   {
#pragma acc loop
      for (int ic=0; ic<ncells; ic++) {
         x[ic] = (double)(ic+1);
         y[ic] = (double)(ncells-ic);
      }

#pragma acc loop reduction(max:MaxRadius)
      for (int ic=0; ic<ncells ; ic++) {
         double radius = sqrt(x[ic]*x[ic] + y[ic]*y[ic]);
         if (radius > MaxRadius) MaxRadius = radius;
      }
   }
   printf("Максимальный радиус составляет %lf\n",MaxRadius);

   acc_free(x);
   acc_free(y);
}
```

B.12 Глава 12: языки GPU: обращение к основам

1 Измените выделение памяти хоста в CUDA-примере потоковой триады, чтобы использовать закрепленную память (листинги 12.1–12.6). Получилось у вас добиться улучшения производительности?

Ответ: в целях закрепления памяти замените `malloc` в выделении памяти на стороне хоста на `cudaHostMalloc` и освободите память с помощью `cudaFreeHost`, как показано в листинге B.12.1. В указанном листинге мы показываем только те строки, которые необходимо изменить. Сравните производительность с исходным кодом из главы 12 в каталоге CUDA/StreamTriad. При закрепленной памяти время передачи данных должно быть как минимум в два раза быстрее.

Листинг B.12.1 Версия потоковой триады с закрепленной памятью

ExerciseB.12.1/StreamTriad.cu

```
31    // выделить память хоста и инициализировать
32    double *a, *b, *c;
33    cudaMallocHost(&a,stream_array_size*sizeof(double));
34    cudaMallocHost(&b,stream_array_size*sizeof(double));
35    cudaMallocHost(&c,stream_array_size*sizeof(double));
        < ... исходный код потоковой триады ... >
86    cudaFreeHost(a);
87    cudaFreeHost(b);
88    cudaFreeHost(c);
```

2 В примере редукции путем суммирования попробуйте массив размером 18 000 элементов, все из которых были изначально заданы в соответствии с индексным значением. Выполните исходный код CUDA, а затем версию в SumReductionRevealed. Возможно, вам захочется скорректировать объем печатаемой информации.

Ответ: по своему усмотрению.

3 Конвертируйте пример редукции из CUDA в HIP путем его HIP'ифизации.

Ответ: по своему усмотрению.

4 Для примера SYCL в листинге 12.20 инициализируйте массивы a и b на устройстве GPU.

Ответ: в следующем ниже листинге показана версия с массивами a и b, инициализированными на GPU.

Листинг B.12.4 Инициализация массивов a и b в SYCL

```
14    // данные хоста
15    vector<double> a(nsize);
16    vector<double> b(nsize);
17    vector<double> c(nsize);
18
19    t1 = chrono::high_resolution_clock::now();
20
21    Sycl::queue Queue(sycl::cpu_selector{});
22
23    const double scalar = 3.0;
24
25    Sycl::buffer<double,1> dev_a { a.data(), Sycl::range<1>(a.size()) };
26    Sycl::buffer<double,1> dev_b { b.data(), Sycl::range<1>(b.size()) };
27    Sycl::buffer<double,1> dev_c { c.data(), Sycl::range<1>(c.size()) };
28
29    Queue.submit([&](sycl::handler& CommandGroup) {
30
31       auto a =
             dev_a.get_access<Sycl::access::mode::write>(CommandGroup);
```

```
32      auto b =
           dev_b.get_access<Sycl::access::mode::write>(CommandGroup);
33      auto c =
           dev_c.get_access<Sycl::access::mode::write>(CommandGroup);
34
35      CommandGroup.parallel_for<class StreamTriad>(
           Sycl::range<1>{nsize}, [=] (Sycl::id<1> it) {
36         a[it] = 1.0;
37         b[it] = 2.0;
38         c[it] = -1.0;
39      });
40   });
41   Queue.wait();
42
43   Queue.submit([&](sycl::handler& CommandGroup) {
44
45   auto a = dev_a.get_access<Sycl::access::mode::read>(CommandGroup);
46   auto b = dev_b.get_access<Sycl::access::mode::read>(CommandGroup);
47   auto c =
        dev_c.get_access<Sycl::access::mode::write>(CommandGroup);
48
49   CommandGroup.parallel_for<class StreamTriad>(
        Sycl::range<1>{nsize}, [=] (Sycl::id<1> it) {
50         c[it] = a[it] + scalar * b[it];
51      });
52   });
53   Queue.wait();
54
55   t2 = chrono::high_resolution_clock::now();
```

5 Конвертируйте два цикла инициализации в примере RAJA в листинге 12.24 в синтаксис `Raja:forall`. Попробуйте выполнить пример с CUDA.

Ответ: цикл инициализации нуждается в изменениях, показанных в следующем ниже листинге. Затем код потоковой триады строится и выполняется так же, как в разделе 12.5.2.

Листинг B.12.5 Добавление RAJA в цикл инициализации потоковой триады

ExerciseB.12.5/StreamTriad.cc

```
19 RAJA::forall<RAJA::omp_parallel_for_exec>(RAJA::RangeSegment(0,
       nsize), [=] (int i) {
20    a[i] = 1.0;
21    b[i] = 2.0;
22 });
```

С учетом этих изменений время выполнения, по сравнению с изначальной версией в разделе 12.5.2, уменьшается примерно с 6.59 мс до 1.67 мс.

B.13 Глава 13: профилирование и инструменты GPU

1. Выполните инструмент nvprof в примере потоковой триады. Вы можете попробовать версию CUDA из главы 12 либо версию OpenACC из главы 11. Какой рабочий поток вы использовали для своих аппаратных ресурсов? Если у вас нет доступа к GPU NVIDIA, то сможете ли вы использовать другой инструмент профилирования?

   Ответ: по своему усмотрению.

2. Создайте трассу из nvprof и импортируйте ее в NVVP. Где тратится время выполнения? Что вы могли бы сделать, чтобы его оптимизировать?

   Ответ: по своему усмотрению.

3. Загрузите предварительно собранный контейнер Docker от соответствующего поставщика для вашей системы. Запустите контейнер и выполните один из примеров из главы 11 либо 12.

   Ответ: по своему усмотрению.

B.14 Глава 14: аффинность: перемирие с вычислительным ядром

1. Создайте визуальное изображение пары разных аппаратных архитектур. Раскройте характеристики оборудования для этих устройств.

   Ответ: используйте инструмент lstopo для генерирования образа вашей архитектуры.

2. На вашем оборудовании выполните комплект тестов, используя скрипт, приведенный в листинге 14.1. Что вы узнали о том, как использовать вашу систему наилучшим образом?

   Ответ: по своему усмотрению.

3. Измените программу, используемую в примере векторного сложения (vecadd_opt3.c) в разделе 14.3, чтобы включить больше операций с плавающей точкой. Возьмите вычислительное ядро и поменяйте операции в цикле на формулу Пифагора:

   ```
   c[i] = sqrt(a[i]*a[i] + b[i]*b[i]);
   ```

   Как изменились ваши результаты и выводы о наилучшем размещении и привязках? Видите ли вы сейчас какую-либо выгоду от гиперпотоков (если они у вас есть)?

   Ответ: по своему усмотрению.

4 В пример MPI из раздела 14.4 включите вычислительное ядро векторного сложения и сгенерируйте график масштабирования для ядра. Затем измените ядро формулой Пифагора, используемой в упражнении 3.

Ответ: по своему усмотрению.

5 Замените ядро формулой Пифагора, используемой в упражнении 3.

Ответ: по своему усмотрению.

6 Скомбинируйте векторное сложение и формулу Пифагора в следующей ниже процедуре (в одном цикле либо в двух отдельных циклах), чтобы обеспечить бо́льшую многоразовость использования данных:

```
c[i] = a[i] + b[i];
d[i] = sqrt(a[i]*a[i] + b[i]*b[i]);
```

Как это меняет результаты исследования размещения и привязывания?

Ответ: по своему усмотрению.

7 Добавьте исходный код, чтобы назначать размещение и аффинность внутри приложения из одного из предыдущих упражнений.

Ответ: по своему усмотрению.

B.15 Глава 15: Пакетные планировщики: наведение порядка в хаосе

1 Попробуйте отправить пару заданий, одно с 32 процессорами и одно с 16 процессами. Проверьте их отправку и подтвердите, что они работают. Удалите 32-процессорное задание. Проверьте, чтобы убедиться, что оно было удалено.

Ответ: по своему усмотрению.

2 Измените скрипт автоматического перезапуска таким образом, чтобы первое задание было шагом предобработки, который необходимо настроить для вычисления до того, как запуски выполнят симуляцию.

Ответ: для вставки шага предобработки нам нужно вставить еще один условный случай, как показано в следующем ниже листинге в строках 31–36, а затем использовать файл PREPROCESS_DONE, чтобы указать, что предобработка была выполнена.

Листинг B.15.2 Вставка шага предобработки, а затем автоматический перезапуск

ExerciseB.15.2/Preprocess_then_restart.sh

```
 1 #!/bin/sh
 2 #SBATCH -N 1
 3 #SBATCH -n 4
 4 #SBATCH --signal=23@160
 5 #SBATCH -t 00:08:00
 6
 7 # Не размещайте команды bash перед последней директивой SBATCH
 8 # Поведение может быть ненадежным
 9
10 NUM_CPUS=4
11 OUTPUT_FILE=run.out
12 EXEC_NAME=./testapp
13 MAX_RESTARTS=4
14
15 if [ -z ${COUNT} ]; then
16    export COUNT=0
17 fi
18
19 ((COUNT++))
20 echo "СЧЕТЧИК перезапусков равен ${COUNT}"
21
22 if [ ! -e DONE ]; then
23    if [ -e RESTART ]; then
24       echo "=== Перезапуск ${EXEC_NAME} ===" >> ${OUTPUT_FILE}
25       cycle=`cat RESTART`
26       rm -f RESTART
27    elif [ -e PREPROCESS_DONE ]; then
28       echo "=== Запуск задачи ===" >> ${OUTPUT_FILE}
29       cycle=""
30    else
31       echo "=== Предобработка данных для задачи ===" >> ${OUTPUT_FILE}
32       mpirun -n ${NUM_CPUS} ./preprocess_data &>> ${OUTPUT_FILE}
33       date > PREPROCESS_DONE
34       sbatch \
            --dependency=afterok:${SLURM_JOB_ID} \
            <preprocess_then_restart.sh
35       exit
36    fi
37
38    echo "=== Отправка скрипта перезапуска ===" >> ${OUTPUT_FILE}
39    sbatch \
         --dependency=afterok:${SLURM_JOB_ID} \
         <preprocess_then_restart.sh
40
41    mpirun -n ${NUM_CPUS} ${EXEC_NAME} ${cycle} &>> ${OUTPUT_FILE}
42    echo "Закончена mpirun" >> ${OUTPUT_FILE}
43
```

Отправляет первое вычислительное задание после предобработки

Отправляет задание перезапуска

```
44    if [ ${COUNT} -ge ${MAX_RESTARTS} ]; then
45       echo "=== Достигнуто максимальное число перезапусков ===" >> ${OUTPUT_FILE}
46       date > DONE
47    fi
48 fi
```

Нередко на шаге предобработки требуется другое число процессоров. В этом случае мы можем использовать отдельный пакетный скрипт для предобработки, показанный в следующем ниже листинге.

Листинг B.15.2b Меньший шаг предобработки, а затем автоматический перезапуск

ExerciseB.15.2/Preprocess_batch.sh

```
 1 #!/bin/sh
 2 #SBATCH -N 1
 3 #SBATCH -n 1
 5 #SBATCH -t 01:00:00
 6
 7 sbatch --dependency=afterok:${SLURM_JOB_ID} <batch_restart.sh
 9
10 mpirun -n 4 ./preprocess &> preprocess.out
```

3 Модифицируйте простой пакетный скрипт в листинге 15.1 для Slurm и листинге 15.2 для PBS, чтобы выполнить очистку при отказе, удалив файл с именем simulation_database.

Ответ: измените пакетный скрипт Slurm в целях проверки статуса команды и удалите симуляционную базу данных. Существует несколько разных способов выполнения очистки. Вот три из них. Первые два в листингах B.15.3a и b используют числовой код выхода из команды `mpirun`.

Листинг B.15.3 OpenACC-версия отыскания максимального радиуса

ExerciseB.15.3/batch_simple_error.sh

```
 1 #!/bin/sh
 2 #SBATCH -N 1
 3 #SBATCH -n 4
 5 #SBATCH -t 01:00:00
 6
 7 mpirun -n 4 ./testapp &> run.out || \
      rm -f simulation_database
```

Символ || исполняет команду для нулевых значений статуса

Листинг B.15.3b OpenACC-версия отыскания максимального радиуса

ExerciseB.15.3/batch_simple_error.sh

```
 1 #!/bin/sh
 2 #SBATCH -N 1
 3 #SBATCH -n 4
```

```
 5 #SBATCH -t 01:00:00
 6
 7 mpirun -n 4 ./testapp &> run.out
 8 STATUS=$?
 9 if [ ${STATUS} != "0" ]; then
10    rm -f simulation_database
11 fi
```

Третья версия в листинге B.15.3.c использует статусное условие пакетного задания с помощью флага зависимости для вызова задания очистки. Типы обрабатываемых ошибок отличаются от первых двух методов.

Листинг B.15.3c OpenACC-версия отыскания максимального радиуса

ExerciseB.15.3/batch.sh

```
 1 #!/bin/sh
 2 #SBATCH -N 1
 3 #SBATCH -n 4
 5 #SBATCH -t 01:00:00
 6
 7 sbatch --dependency=afternotok:${SLURM_JOB_ID} <batch_cleanup.sh
 9
10 mpirun -n 4 ./testapp &> run.out
```

ExerciseB.15.3/batch_cleanup.sh

```
 1 #!/bin/sh
 2 #SBATCH -N 1
 3 #SBATCH -n 1
 5 #SBATCH -t 00:10:00
 6 rm -f simulation_database
```

B.16 Глава 16: файловые операции для параллельного мира

1. Проверьте наличие доступных в вашей системе подсказок, используя технические приемы, описанные в разделе 16.6.1.

   Ответ: по своему усмотрению.

2. Попробуйте примеры MPI-IO и HDF5 в вашей системе с гораздо более крупными наборами данных, чтобы увидеть производительность, которую вы сможете достичь. Сравните это с микроэталоном IOR для получения дополнительного кредита.

   Ответ: по своему усмотрению.

3. Используйте утилиты h5ls и h5dump для проведения разведывательного анализа файла данных HDF5, созданного в примере HDF5.

   Ответ: по своему усмотрению.

B.17 Глава 17: инструменты и ресурсы для улучшения исходного кода

1. Выполните инструмент Dr. Memory для одного из ваших небольших исходных кодов или одного из исходных кодов из упражнений в этой книге.

   Ответ: по своему усмотрению.

2. Скомпилируйте один из ваших исходных кодов с помощью библиотеки dmalloc. Выполните свой исходный код и просмотрите результаты.
3. Попробуйте вставить поточное гоночное условие в пример исходного кода в разделе 17.6.2 и посмотрите, как Archer сообщает о проблеме.

   Ответ: по своему усмотрению.

4. Попробуйте выполнить профилировочное упражнение из раздела 17.8 в вашей файловой системе. Если у вас более одной файловой системы, то попробуйте ее на каждой из них. Затем поменяйте размер массива в примере на 2000×2000. Как это влияет на результаты работы файловой системы?

   Ответ: по своему усмотрению.

5. Инсталлируйте один из инструментов с помощью диспетчера пакетов Spack.

   Ответ: по своему усмотрению.

Приложение C
Глоссарий

3DNow! – набор векторных команд AMD, который впервые поддерживал операции с одинарной точностью (прецизионностью).

AVX (Advanced Vector Extensions) – продвинутые векторные расширения представляют собой 256-битовый векторный аппаратный модуль и набор команд.

AVX2 – усовершенствование в оборудовании AVX для поддержки операций слитного умножения-сложения (fused multiply add, FMA).

AVX512 – расширяет оборудование AVX до 512-битовой ширины вектора.

DRAM (Dynamic Random Access Memory) – динамическая память прямого доступа. Эта память нуждается в частом обновлении состояния, и данные, которые она хранит, теряются при отключении питания.

CPU (Central Processing Unit), или центральный процессор, – устройство дискретной обработки, состоящее из одного или нескольких вычислительных ядер, которое размещено в гнезде печатной платы для обеспечения главных вычислительных операций.

GNU – это не Unix (GNU's Not Unix, GNU). Бесплатная Unix-подобная операционная система.

GPU (Graphics processing unit) – графический процессор, также общецелевой графический процессор (general-purpose graphics processing unit, GPGPU), интегрированный или дискретный (внешний). Это устройство, основным назначением которого является отрисовка графики на мониторе компьютера. Он состоит из большого числа потоковых (streaming) мультипроцессоров и собственной оперативной памяти и способен выполнять десятки тысяч потоков за один тактовый цикл.

HAL – небольшой компьютер-беспредельщик, который предшествует IBM в лексикографическом порядке. HAL – это вымышленный

компьютер из фильма Артура Кларка «2001: Космическая одиссея». HAL пустился во все тяжкие, потому что он интерпретировал инструкции иначе, чем предполагалось, со смертельными последствиями. Название HAL отделено от IBM одной буквой. Мораль примера с этим компьютером состоит в том, чтобы, программируя, быть осторожным; никогда не знаешь, какими могут быть результаты.

MIMD (Multiple instruction, multiple data), несколько команд, несколько элементов данных – это компонент таксономии Флинна, представленный мультиядерной системой.

MISD (Multiple instruction, single data), несколько команд, один элемент данных – это компонент таксономии Флинна, описывающий резервный компьютер для высокой надежности либо параллелизм параллельного конвейера.

MMX – это самый ранний выпущенный Intel набор векторных команд на базе x86.

N-путный секторно-ассоциативный кеш (N-way set associative cache) – кеш, который позволяет отображать в кеш N местоположений для адреса памяти. За счет того уменьшается число коллизий и вытеснений, связанных с кешем с прямым отображением.

SIMD (Single instruction, multiple data), одна команда, несколько элементов данных – это компонент таксономии Флинна, описывающий параллелизм, подобный тому, который встречается при векторизации, когда одна команда применяется к нескольким элементам данных.

SIMT (Single instruction, multiple thread), одна команда, несколько потоков – это вариант SIMD, в котором несколько потоков одновременно оперируют на нескольких элементах данных.

SISD (Single instruction, single data), одна команда, один элемент данных – это компонент таксономии Флинна, который описывает традиционную последовательную архитектуру.

SSE (streaming SIMD Extensions) – это выпущенные Intel векторное оборудование и набор команд, которые впервые поддерживали операции с плавающей точкой.

SSE2 – это улучшенный набор команд SSE, поддерживающий операции с двойной точностью (прецизионностью).

Автоматическая векторизация (Auto-vectorization) – это векторизация исходного кода компилятором для стандартного исходного кода на языках C, C++ или Fortran.

Алгоритмическая сложность (Algorithmic complexity) – это мера числа операций, необходимых для полного завершения алгоритма. Алгоритмическая сложность является свойством алгоритма и мерой объема работы или операций в процедуре.

Арифметическая интенсивность (Arithmetic intensity) – это число операций с плавающей точкой (флопов) относительно загрузок памяти (данных), выполняемых вашим приложением или вычислительным ядром (в цикле). Арифметическая интенсивность является важным показателем для понимания предельных характеристик приложения.

Асимптотическая нотация (Asymptotic notation) – это выражение, определяющее лимитирующую границу производительности. В сущности, отвечает на вопрос, растет ли время выполнения линейно либо ухудшается вместе с размером задачи. В указанной нотации используются различные формы *O*, такие как $O(n)$, $O(n \log_2 n)$ или $O(n^2)$. о можно рассматривать как «порядок», как в словосочетании «в масштабах порядка».

Асинхронный (Asynchronous) – такой вызов не является блокирующим и только инициирует операцию.

Аффинность (Affinity) – это придание предпочтения тому или иному аппаратному компоненту в размещении процесса, ранга или потока (виртуального ядра). Такое придание предпочтения также называется закреплением или привязкой.

Блокирование (Blocking) – это операция, которая не завершается до тех пор, пока не будет выполнено конкретное условие.

Буфер быстрой трансляции адресов (Translation lookaside buffer, TLB), или буфер ассоциативной трансляции, – это таблица записей для трансляции адресов виртуальной памяти в физическую память. Ограниченный размер таблицы означает, что в памяти хранятся только недавно использованные расположения страниц, и при их отсутствии происходит неуспешное обращение к TLB, что приводит к значительному снижению производительности.

Валидированные результаты (Validated results) – это результаты расчета, которые выглядят предпочтительнее при сравнении с экспериментальными или реальными данными.

Варп (Warp) – это альтернативный термин для рабочей группы потоков (или виртуальных ядер).

Векторизация (Vectorization) – это процесс группирования операций вместе, чтобы иметь возможность выполнять несколько операций одновременно.

Векторная операция (Vector operation) – это операция на двух или более элементах массива, при которой в процессор передается одна операция или команда.

Векторная полоса (Vector lane) – это маршрут через векторную операцию на векторных регистрах для одного элемента данных; во многом похож на полосу движения на многополосной автостраде.

Временная сложность (Time complexity) – временная сложность учитывает фактическую стоимость операции в типичной современной вычислительной системе. Наибольшая корректировка по времени заключается в учете затрат на загрузку памяти и кеширование данных.

Встраиваемые процедуры (Inline routines) – это ситуация, когда, вместо того чтобы вызывать функцию, компиляторы вставляют исходный код в точку вызова, чтобы избежать накладных расходов на вызов. Это работает только в случае небольших подпрограмм и более простого исходного кода.

Выделенный GPU (Dedicated GPU) – это графический процессор на отдельной периферийной карте. Также называется дискретным GPU.

Вызов дистанционных (удаленных) процедур (Remote procedure call, RPC) – это обращение к системе, чтобы та исполнила еще одну команду.

Вынужденные неуспешные обращения (Compulsory misses) – это неуспешные обращения к кешу, необходимые для внесения данных при их первом обнаружении.

Высокопроизводительные вычисления (High Performance Computing, HPC) – это вычисления, ориентированные на экстремальную производительность. Соответствующее вычислительное оборудование, как правило, более тесно состыковано. Термин «высокопроизводительные вычисления», в сущности, заменил старую номенклатуру супервычислений.

Вытеснение из кеша (Cache eviction) – это удаление блоков данных, именуемых строками кеша, с одного из различных уровней иерархии кеша.

Вычислительная сетка (Computational mesh) – это коллекция ячеек или элементов, охватывающих участок симуляции.

Вычислительная сложность (Computational complexity) – это число шагов, необходимых для завершения алгоритма. Указанная мера сложности является атрибутом имплементации и типа оборудования, используемого для вычисления.

Вычислительное устройство OpenCL (Compute device OpenCL) – это любое вычислительное оборудование, которое способно выполнять вычисления и поддерживать OpenCL. к нему относятся GPU-процессоры, CPU-процессоры или даже более экзотическое оборудование, такое как встраиваемые процессоры или программируемые пользователем вентильные матрицы (FPGA).

Вычислительное ядро (Computational kernel) – это раздел приложения, который является интенсивным с точки зрения вычислений и одновременно концептуально автономным.

Гиперпотокообразование (Hyperthreading), или гиперпоточность, – это технология Intel, которая делает один процессор похожим для операционной системы на два виртуальных процессора за счет совместного использования аппаратных ресурсов между двумя потоками (threads).

Гоночные условия (Race conditions) – это ситуации, когда возможно несколько исходов, и результат зависит от хронометража участников.

Графический пользовательский интерфейс (Graphical user interface, GUI) – это интерфейс, состоящий из визуальных элементов и интерактивных компонентов, которыми можно управлять с помощью мыши или других продвинутых устройств ввода.

Давление на регистр (Register pressure) – давление на регистр относится к влиянию потребностей в регистрах на производительность вычислительных ядер GPU.

Давление на память (Memory pressure) – это влияние потребностей вычислительного ядра в ресурсах на производительность ядер GPU.

Давление на регистр – это аналогичный термин, относящийся к потребностям в регистрах в ядре.

Двоичный формат данных (Binary data format) – это машинное представление данных, которые используются процессором и хранится в основной памяти. Обычно этот термин относится к формату данных, сохраняемому в двоичной форме при записи на жесткий диск.

Дереференция (Dereferencing) – это операция, при которой адрес памяти получается из ссылки указателя, в результате которой строка кеша соответствует данным памяти, а не указателю.

Динамический диапазон (Dynamic range) – это диапазон рабочего множества действительных чисел в задаче.

Директива (Directive) – это команда компилятору Fortran, помогающая ему интерпретировать исходный код. Форма инструкции представляет собой строку комментария, начинающуюся с символов `!$`.

Дискретизация (Discretization) – это процесс разбиения вычислительного домена на более мелкие ячейки или элементы, образующие вычислительную сетку. Затем вычисления выполняются на каждой ячейке или элементе.

Длина вектора (Vector length) – это число операций, выполняемых векторным модулем за один цикл.

Доступ к памяти за пределами границ (Out-of-bounds memory access) – это попытка доступа к памяти за пределами массива. Такого рода ошибки могут улавливаться инструментами проверки столбов ограждения и некоторыми компиляторами.

Емкостное неуспешное обращение (Capacity misses) – это отказы, вызываемые лимитированным размером кеша.

Зависание (Hang) – это ситуация, когда один или несколько процессоров ожидают события, которое никогда не сможет произойти.

Зависимость от выхода (Output dependency) – это ситуация, когда переменная пишется в цикле более одного раза.

Зависимость от обратного порядка исполнения (Anti-flow dependency) – это ситуация, когда переменная внутри цикла пишется после чтения; такая ситуация еще называется записью после чтения (write-after-read, WAR).

Зависимость от прямого порядка исполнения (Flow dependency) – это ситуация, когда переменная в цикле читается после записи. Также называется чтением после записи (read-after-write, RAW).

Задержка (Latency) – то время, необходимое для передачи первого байта или слова данных (см. также Задержка памяти).

Задержка памяти (Memory latency) – это время, необходимое для извлечения первого байта памяти с уровня иерархии памяти.

Закрепленная память (Pinned memory) – это память, которую нельзя выгрузить из оперативной памяти. Она особенно полезна для передачи памяти, потому что ее можно отправлять напрямую, не делая копии.

Идеальный хеш (Perfect hash) – это хеш, в котором нет коллизий; в каждой корзине имеется не более одной записи.

Идеально вложенные циклы (Perfectly nested loops) – это циклы, которые содержат инструкции только в самом внутреннем цикле. Это означает, что до или после каждого блока цикла нет посторонних инструкций.

Инвокация метода (Method invocation) – это термин из объектно-ориентированного программирования, обозначающий вызов фрагмента исходного кода внутри объекта, который оперирует на данных в объекте. Такого рода малые фрагменты исходного кода называются методами, и их вызов называется инвокацией.

Индексный шаг (в массиве) (Stride) – это расстояние между индексированными элементами в массиве. В языке C в размерности x данные являются сплошными или имеют индексный шаг 1. В размерности y данные имеют индексный шаг, равный длине строки.

Интегрированный GPU (Integrated GPU) – это графический процессор, который содержится в процессоре.

Информационный вектор (Dope vector) – это метаданные для массива в языке Fortran, которые состоят из начала, шага и длины для каждой размерности. Название термина происходит от фразы «мне нужен допинг», т. е. информация.

Катастрофическая взаимоотмена (Catastrophic cancellation) – это вычитание двух почти одинаковых чисел, в результате которого в итоге получается всего несколько значащих цифр.

Кластер (Cluster) – это небольшая группа распределенных узлов памяти, соединенных товарной сетью.

Коагулированные загрузки памяти (Coalesced memory loads) – это сочетание отдельных загрузок памяти из групп потоков в одну загрузку строки кеша.

Когерентность (Coherency) – относится к обновлениям кеша, которые необходимы для синхронизирования кеша между мультипроцессорами, когда данные, записанные в кеш одного процессора, также содержатся в кеше другого процессора.

Коллекция компиляторов GNU (GNU Compiler Collection, GCC) – это общедоступный набор компиляторов с открытым исходным кодом, включающий C, C++, Fortran и многие другие языки.

Компактный хеш (Compact hash) – это хеш, сжатый в меньший объем памяти. Компактный хеш должен иметь способ обработки коллизий.

Комплект тестов (Test suite) – это набор задач, которые исполняют части приложения, чтобы гарантировать работоспособность частей исходного кода.

Конкурентность (Concurrency) – это выполнение частей программы в любом порядке с одинаковым результатом. Конкурентность изначально была разработана для поддержки конкурентных вычислений или технологии совместного использования времени путем чередования вычислений на ограниченном наборе ресурсов.

Контрольная точка / Перезапуск (Checkpoint/Restart) – это периодическая регистрация состояния приложения с последующим запуском приложения в более позднем задании.

Конфликтные неуспешные обращения к кешу (Conflict cache misses) – это неуспешные обращения, вызываемые загрузкой еще одного блока памяти в строку кеша, которая все еще необходима процессору.

Корзина (Bucket) – это место хранения, в котором содержится коллекция значений. Для хранения значений ключей в корзине используются технические приемы хеширования, поскольку для конкретного местоположения может быть несколько значений.

Коэффициент загрузки (хеша) (Load factor) – это доля хеша, заполненного записями.

Коэффициент загрузки хеша (Hash load factor) – это число заполненных корзин, деленное на суммарное число корзин в хеше.

Крупнозернистый параллелизм (Coarse-grained parallelism) – это тип параллелизма, при котором процессор оперирует на больших блоках исходного кода с нечастой синхронизацией.

Куча (Heap) – это участок памяти для программы, который используется для обеспечения для нее динамической памяти. Подпрограммы `malloc` и оператор `new` получают память из этого участка. Вторым участком памяти является стековая память.

Кеш (Cache) – это более быстрый блок памяти, который используется для снижения затрат на доступ к более медленной основной памяти за счет хранения блоков данных или команд, которые могут потребоваться.

Кеш команд (Instruction cache) – это хранилище команд в быстрой памяти рядом с процессором. Команды могут предназначаться для перемещения памяти, операций с целыми числами или с плавающей точкой. Данные, на которых выполнятся операции, имеют свой собственный отдельный кеш.

Листание памяти (Memory paging) – это процесс временного перемещения страниц памяти на диск для выполнения еще одного процесса в многопользовательских операционных системах с несколькими приложениями.

Лямбда-выражения (Lambda expressions) – это безымянная локальная функция, которая может назначаться переменной и использоваться локально либо передаваться в подпрограмму.

Материнская плата (Motherboard) – это главная системная плата компьютера.

Машинный баланс (Machine balance) – это соотношение флопов к загрузкам памяти, которые могут выполняться компьютерной системой.

Межпроцессное взаимодействие (Inter-process communication, IPC) – это обмен данными между процессами на узле компьютера. Различные технические приемы обмена данными между процессами формируют основу клиент–серверных механизмов в распределенных вычислениях.

Межсоединения (Interconnects) – это соединения между вычислительными узлами, также именуемые *сетью*. Как правило, этот термин относится к сетям с более высокой производительностью, которые тесно состыковывают операции в системе параллельных вычислений. Многие из этих межсоединений являются проприетарными, являясь собственностью поставщика, и содержат специализированные топологии, такие как утолщенное дерево, коммутаторы, торы и стрекозиная схема.

Мелкозернистый параллелизм (Fine-grained parallelism) – это тип параллелизма, при котором вычислительные циклы или другие малые блоки исходного кода обрабатываются несколькими процессорами или потоками и могут нуждаться в частой синхронизации.

Минимальный идеальный хеш (Minimal perfect hash) – это хеш с одной и только одной записью в каждой корзине.

Модель ослабленной памяти (Relaxed memory model) – это модель, при которой значения переменных в основной памяти или кешах всех процессоров не обновляются в один момент.

Модель производительности (Performance model) – это упрощенное представление того, как операции в программе могут конвертироваться в оценку времени выполнения исходного кода.

Модульное тестирование (Unit testing), или тестирование единиц исходного кода, – это тестирование каждого отдельного компонента программы.

Мультиядерный (Multi-core) – относится к CPU, который содержит более чем одно ядро.

Набор векторных (SIMD) команд (Vector (SIMD) instruction set) – это набор команд, расширяющих обычные команды скалярного процессора до эффективного использования векторного процессора.

Неинициализированная память (Uninitialized memory) – это память, которая используется до задания ей значений.

Непрерывная интеграция (Continuous integration) – это автоматический процесс тестирования, который активируется при каждой фиксации в репозитории.

Неравномерный доступ к памяти (Non-Uniform Memory Access, NUMA) – это ситуация, когда на некоторых вычислительных узлах блоки памяти расположены ближе к одним процессорам, чем к другим, и она нередко происходит, когда узел имеет два процессорных разъема, причем каждый разъем имеет свою собственную память. Доступ к другому блоку памяти обычно занимает в два раза больше времени, чем к его собственной памяти.

Неуспешное обращение к ветви (Branch miss) – это стоимость, возникающая, когда предсказанная ветка в инструкции `if` является неправильной.

Неуспешные обращения к кешу (Cache misses) – это ситуация, которая возникает, когда процессор пытается получить доступ к адресу памя-

ти, но его нет в кеше. Затем система должна извлечь данные из основной памяти за 100 тактовых циклов.

Нехватка когерентности (Coherency misses) – это обновления кеша, требующиеся для синхронизации кешей между мультипроцессорами, когда данные, которые записываются в кеш одного процессора, также хранятся в кеше другого процессора.

Обрабатывающее ядро (Processing core), или просто ядро, – это самый базовый модуль, способный выполнять арифметические и логические операции.

Объектно ориентированная файловая система (Object-based filesystem) – это система, организованная на основе объектов, а не на основе файлов в папке. Объектно ориентированной файловой системе требуется база данных или метаданные для хранения всей описывающей объект информации.

Операции (Operations, Ops) – операции могут быть целочисленными, с плавающей точкой либо логическими.

Операционная задача (Task) – это работа, которая подразделяется на отдельные части и отправляется по отдельным процессам или процессорным потокам.

Операция разброса памяти (Scatter memory operation) – это операция сохранения данных строки кеша или векторного модуля в несмежных ячейках памяти.

Операция сбора памяти (Gather memory operation) – это операция, когда память загружается в строку кеша или векторный модуль из несмежных ячеек памяти.

Описательные директивы и выражения (Descriptive directives and clauses) – это команды, которые предоставляют компилятору информацию о следующем конструкте цикла и дают компилятору некоторую свободу в генерировании наиболее эффективной имплементации.

Ореолы на границах доменов (Domain-boundary halos) – это ореольные ячейки, используемые для наложения определенного набора граничных условий.

Ореольные ячейки (Halo cells) – это любой набор ячеек, окружающих домен вычислительной сетки.

Основная память (Main memory), также именуемая DRAM или RAM, или оперативная память, – это крупный блок памяти в вычислительном узле.

Остаточный цикл (Remainder loop) – это цикл, который исполняется после главного цикла для обработки частичного набора данных, который слишком мал для полной векторной длины.

Отслаивающий цикл (Peel loop) – это цикл, исполняемый для невыровненных данных, чтобы затем главный цикл выровнял эти данные. Нередко отслаивающий цикл исполняется условно во время выполнения, если обнаруживается, что данные не выровнены.

Память с прямым доступом (Random access memory, RAM), или оперативная память, – это главная системная память, из которой можно

извлекать любые необходимые данные без необходимости их последовательного чтения.

Параллелизм (Parallelism) – это одновременная работа частей программы с привлечением набора ресурсов.

Параллельное ускорение (Parallel speedup) – это производительность параллельной имплементации по сравнению с базовым последовательным выполнением.

Параллельность данных (Data parallel) – это тип параллелизма, при котором данные распределяются между процессорами или потоками и обрабатываются параллельно.

Параллельные вычисления (Parallel computing) – это вычисления, которые оперируют более чем на одной вещи за один раз.

Параллельный алгоритм (Parallel algorithm) – это четко определенная пошаговая вычислительная процедура, которая подчеркивает параллелизм в решении задачи.

Параллельный шаблон (Parallel pattern) – это общий, независимый параллельный компонент исходного кода, который с некоторой частотой встречается в различных сценариях. Сами по себе эти компоненты, как правило, не решают полных задач, которые представляют интерес.

Параллельность операционных задач (Task parallel) – это форма параллелизма, при которой процессоры или процессорные потоки оперируют на отдельных задачах.

Первое касание (First touch) – первое касание к массиву приводит к выделению памяти. Она выделяется рядом с месторасположением потока, где происходит касание. До первого касания память существует только как запись в виртуальной памяти. Физическая память, которая соответствует виртуальной памяти, создается при первом доступе к ней.

Перезапись памяти (Memory overwrite) – это запись в память, которая не принадлежит переменной в программе.

Перетирание кеша (Cache thrashing) – это состояние, при котором одна загрузка памяти вытесняет другую, а затем изначальные данные требуются снова, вызывая загрузку, удаление и перезагрузку данных.

Плотно вложенные циклы (Tightly-nested loops) – это два или более циклов, в которых нет дополнительных инструкций между инструкциями for или do или концовок циклов.

Призрачные клетки (Ghost cells) – это набор ячеек, содержащих данные смежных процессоров для использования на локальном процессоре, чтобы процессор мог оперировать крупными блоками без выполнения вызовов обмена данными.

Поколение PCIe (Generation PCIe) – это спецификацию PCI Express. Группа с общим интересом в PCI (PCI Special Interest Group, PCI SIG) – это группа, представляющая отраслевых партнеров, формирующая спецификацию PCI Express, которую для краткости принято называть *поколением* или *gen* (от *generation*).

Покрытие исходного кода (Code coverage) – это метрика того, сколько строк исходного кода было исполненено и, следовательно, «покрывается» выполнением комплекта тестов. Обычно оо выражается в процентах от строк исходного кода.

Полосы (векторные полосы) (Lanes, vector lanes) – это маршруты данных в векторной операции. Для 256-битного векторного модуля, работающего со значениями двойной точности (прецизионности), предусмотрено четыре полосы, позволяющие выполнять четыре одновременные операции с одной командой за один тактовый цикл.

Пользовательское пространство (User space) – область видимости управления операциями для программы такова, что она изолирована от области видимости операционной системы.

Постраничная память (Pageable memory) – это стандартные выделения памяти, которые можно выгружать на диск. См. Закрепленная память касательно альтернативного типа, который нельзя выгружать постранично.

Потоковое хранение (Streaming store) – это сохранение значения непосредственно в основную память, минуя иерархию кеша.

Потоковые вычислительные ядра (Streaming kernels) – это блоки вычислительного кода, которые загружают данные почти оптимальным способом, эффективно используя иерархию кеша.

Потоковый мультипроцессор (Streaming multiprocessor, SM) – обычно используется для описания мультипроцессоров GPU-процессора, которые предназначены для потоковых операций. Это тесно состыкованные симметричные процессоры (SMP), которые имеют один поток (stream) команд, работающих в нескольких процессорных потоках (threads).

Прагма (Pragma) – это команда компилятору C или C++ , помогающая ему интерпретировать исходный код. Форма команды представляет собой инструкцию препроцессора, начинающуюся с ключевого слова `#pragma`.

Приватная переменная в OpenMP (Private OpenMP variable) – в контексте OpenMP приватная переменная является локальной и видимой только для ее потока.

Предписывающие директивы и выражения (Prescriptive directives and clauses) – это директивы программиста, которые указывают компилятору, что конкретно делать.

Проблема глобальной суммы (Global sum issue) – это разница в глобальной сумме при параллельном вычислении по сравнению с последовательным вычислением или выполняемом на другом числе процессоров.

Пропускная способность (Bandwidth), или ширина полосы, – это наилучшая скорость, с которой данные могут перемещаться по заданному пути в системе. Она может относиться к проходной мощности памяти, диска или сети.

Пространственная локальность (Spatial locality) – это данные с близлежащими местоположениями в памяти, к которым осуществляются частые обращения как близким друг к другу.

Профилирование (Profiling) – измерение времени выполнения некоторых аспектов производительности приложения; чаще всего это время, необходимое для исполнения частей программы.

Профилировщик (Profiler) – это инструмент программирования, который измеряет производительность приложения.

Процесс (Process) – это независимая вычислительная единица, которая владеет частью памяти и контролирует ресурсы в пользовательском пространстве.

Процессорное ядро (Core) – это ядро процессора или просто ядро, которое является базовым элементом системы, выполняющим математические и логические операции.

Процессорный поток (Thread), или поток исполнения, или виртуальное ядро, – это отдельный маршрут команд через процесс, создаваемый за счет наличия более одного указателя команд.

Прямо-отображаемый кеш (Direct-mapped cache) – это кеш, для которого адрес памяти имеет только одно место в кеше, откуда он может быть загружен. Он может приводить к коллизиям и вытеснениям, если еще один блок памяти тоже отображается в этот адрес (см. N-путный секторно-ассоциативный кеш касательно типа кеша, который позволяет избегать этой проблемы).

Псевдонимизация (Aliasing) – это ситуация, когда указатели указывают на перекрывающиеся области памяти. В такой ситуации компилятор не способен определить, является ли эта область той же самой памятью и будет ли небезопасно генерировать векторизованный исходный код или другие оптимизации.

Рабочая группа (Workgroup) – это группа потоков (потоков исполнения или виртуальных ядер), работающих вместе с одной очередью команд.

Разработка на основе тестов (Test-driven development, TDD) – это процесс разработки исходного кода, при котором сначала создаются тесты.

Разъем (Socket) – это место, в которое на материнской плате вставляется процессор. Материнские платы обычно имеют один либо два разъема, что позволяет устанавливать соответственно один либо два процессора.

Разреженность хеша (Hash sparsity) – это объем пустого места в хеше.

Распределенная память (Distributed memory) – это более одного блока памяти, каждый из которых существует в своем собственном пространстве адресов и контроля.

Распределенные вычисления (Distributed computing) – это приложения и слабо состыкованные рабочие потоки, которые охватывают несколько компьютеров и используют обмен данными по сети для координации работы. Примеры применений распределенных вычислений

включают поиск с помощью браузеров в интернете и взаимодействие многочисленных клиентов с базой данных на сервере.

Распределенный массив (Distributed array) – это массив, который подразделен и разбит между процессорами. Например, массив, содержащий 100 значений, может быть подразделен между четырьмя процессорами с 25 значениями на каждом процессоре.

Регрессионные тесты (Regression tests) – это комплекты тестов, которые выполняются с периодическими интервалами, например каждую ночь или неделю.

Редукционная операция (Reduction operation) – это любая операция, при которой многомерный массив от 1 до *N* размерностей сводится по крайней мере к числу размерностей на единицу меньше и нередко к скалярному значению.

Реплицированный массив (Replicated array) – это набор данных, который дублируется во всех процессорах.

Репозиторий исходного кода (Source code repository) – это хранилище исходного кода, которое отслеживает изменения и может использоваться разработчиками исходного кода проекта совместно.

Сверхлинейное ускорение (Super-linear speedup) – это производительность, которая лучше, чем идеальная кривая сильного масштабирования. Она может происходить из-за того, что меньшие размеры массива вписываются в более высокий уровень кеша, что приводит к повышению производительности кеша.

Сеть (Network) – это соединения между вычислительными узлами, по которым передаются данные.

Сжато-разреженные структуры данных (Compressed sparse data structures) – это экономичный способ представления разреженного пространства данных. Наиболее заметным примером является формат сжато-разреженной строки (Compressed Sparse Row, CSR), используемый для разреженных матриц.

Симметричные процессоры (Symmetric processors, SMP) – это технолгия, когда все ядра мультиядерного процессора работают в унисон в режиме одной команды и нескольких потоков (SIMT).

Система версионного контроля (Version Control System) – это база данных, которая отслеживает изменения в исходном коде, упрощает объединение работы нескольких разработчиков и предоставляет возможность отката изменений.

Система распределенного версионного контроля (Distributed version control system) – это система версионного контроля, которая позволяет использовать несколько репозиторных баз данных, а не единую централизованную систему.

Скалярная операция (Scalar operation) – это операция на одном значении или одном элементе массива.

Слово (размер) (Word) – это размер используемого базового типа. Для одинарной точности (прецизионности) это четыре байта, а для двойной – восемь байт.

Совместная память (Shared memory) – это блок памяти, доступный и изменяемый несколькими процессами или потоками исполнения. Здесь блок памяти рассматривается с точки зрения программиста.

Совместная переменная OpenMP (Shared OpenMP variable) – это совместная переменная в контексте OpenMP, которая видима и модифицируема любым потоком.

Сравнительные ускорения (Comparative speedups) – это сокращение от «сравнительных повышений производительности между архитектурами». Обозначает относительную производительность между двумя аппаратными архитектурами, нередко получаемую на основе одного узла или фиксированной огибающей мощности.

Сплошная память (Contiguous memory) – это память, состоящая из непрерывной последовательности байтов.

Стандарт POSIX (POSIX standard) – это стандарт переносимого интерфейса операционной системы (Portable Operating System Interface, POSIX), который является стандартом IEEE для Unix и Unix-подобных операционных систем, служащий для облегчения переносимости. Указанный стандарт определяет главные операции, которые должна выполнять OS.

Стековая память (Stack memory) – память в подпрограмме часто создается путем перемещения объектов в стек после указателя стека. Обычно это малые объекты памяти, которые существуют только во время исполнения процедуры и исчезают в конце процедуры, когда указатель инструкции возвращается в предыдущее местоположение.

Строка кеша (Cache line) – это блок данных, загружаемый в кеш при доступе к памяти.

Стек вызовов (Call stack) – это список вызываемых подпрограмм, который должен быть размотан возвратом в конце подпрограммы в месте, где она перепрыгивает назад к предыдущей вызывающей подпрограмме.

Тактовый цикл (Clock cycle) – это небольшие промежутки времени между операциями в компьютере, которые зависят от тактовой частоты системы.

Таксономия Флинна (Flynn's Taxonomy) – это классификация компьютерных архитектур, в основу которой положен критерий одиночности-множественности, т. е. являются ли данные и инструкции одиночными или множественными.

Темпоральная локальность (Temporal locality) – это данные, к которым обращались недавно и на которые, вероятно, будут ссылаться в ближайшем будущем.

Теплый кеш (Warm cache) – это ситуация, когда в кеше есть оперируемые данные из предыдущей операции в то время, когда начинается текущая операция.

Узел (Node) – это базовый строительный блок вычислительного кластера с собственной памятью и сетью для обмена данными с другими вычислительными узлами и выполнения единого образа операционной системы.

Унифицированная память (Unified memory) – это память, которая выглядит как единое адресное пространство как для CPU, так и для GPU.

Утечки памяти (Memory leaks) – это выделение памяти и ее невысвобождение. Инструменты-заменители процедур malloc хорошо справляются с обнаружением и сообщением об утечках памяти.

Фиксационные тесты (Commit tests) – это комплект тестов, который выполняется перед фиксацией любого исходного кода в репозитории.

Фиксация состояния в контрольных точках (Checkpointing) – это практика периодического сохранения состояния вычисления на диск, чтобы вычисление можно было перезапускать из-за системных отказов либо конечного времени выполнения в пакетной системе (см. Контрольная точка / перезапуск).

Флопы (FLOPs) – это операции с плавающей точкой, такие как сложение, вычитание и умножение для типов данных с одинарной или двойной точностью (прецизионностью).

Холодный кеш (Cold cache) – это кеш, в котором нет данных, оперируемых в кеше из предыдущей операции, когда начинается текущая операция.

Хеш-коллизии (Hash collisions) – это ситуация, когда несколько ключей хотят сохранить свое значение в одной и той же корзине.

Хеш или хеширование (Hash or hashing) – это вычислительная структура данных, которая соотносит ключ со значением.

Централизованная система версионного контроля (Centralized version control system) – это система версионного контроля, имплементированная как единая централизованная система.

Шаблонное правило (Pattern rule) – это описание, которое предназначено для утилиты make, задающее общее правило конвертирования любого файла с одним суффиксальным шаблоном в файл с другим суффискальным шаблоном.

Шина PCI (PCI bus) – это шина соединения периферийных компонентов, которая является основным каналом передачи данных между компонентами на системной плате, включая CPU, основную память и сеть обмена данными.

Ширина вектора (Vector width) – это ширина векторного модуля, обычно выражаемая в битах.

Штурмы обновления кеша (Cache update storms) – когда в многопроцессорной системе один процессор модифицирует данные, находящиеся в кеше другого процессора, данные должны быть перезагружены на этих других процессорах.

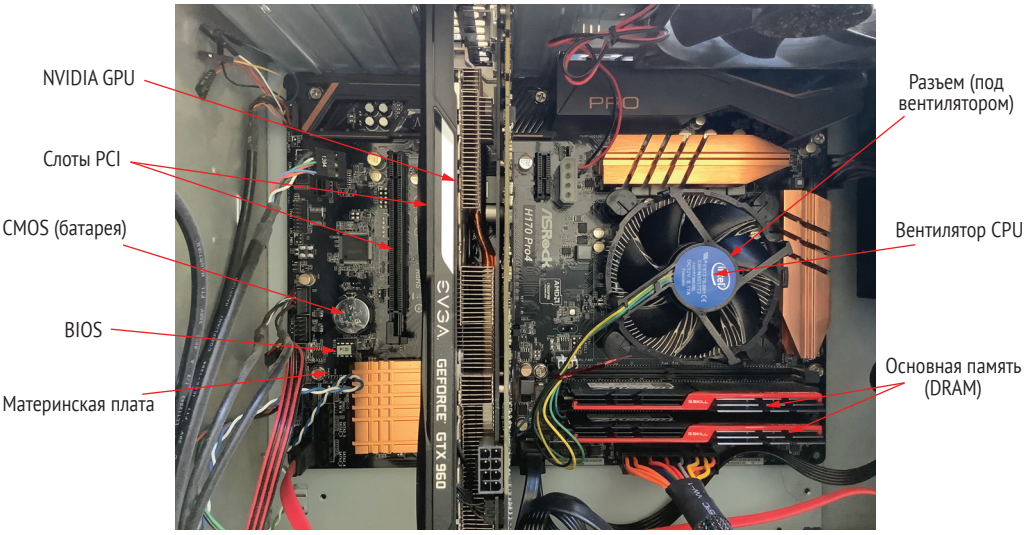


Рис. C.1 Материнская плата для настольного компьютера с процессором Intel и дискретным GPU NVIDIA

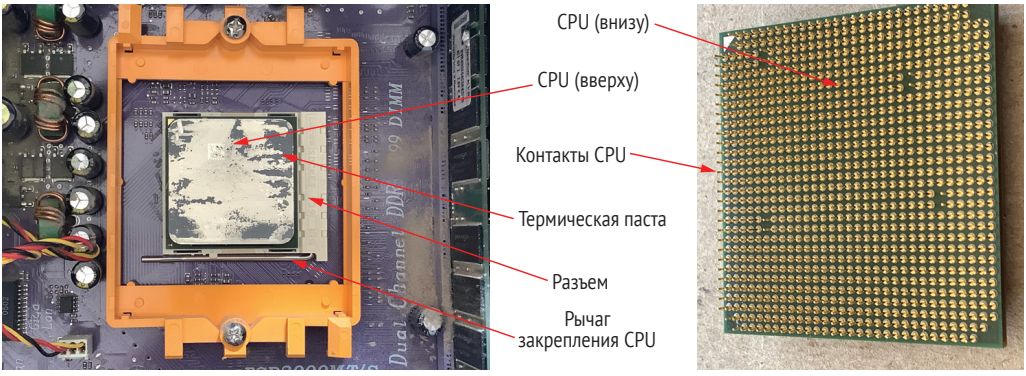


Рис. C.2 Процессор Intel, установленный в разъеме, и его нижняя сторона с контактами процессорных данных. Передача данных в процессор лимитирована числом контактов, которые могут физически уместиться на поверхности CPU

Предметный указатель

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

V

W

X

A

Б

В

Г

Д

Е

З

И

К

Л

М

Н

О

П

P

C

T

У

Ф

Х

Ц

Ш

Э

Я

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38;
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **http://www.galaktika-dmk.com/**.

Роберт Роби и Джулиана Замора

Параллельные и высокопроизводительные вычисления

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Зам. главного редактора *Сенченкова Е. А.*
Перевод *Логунов А. В.*
Корректор *Абросимова Л. А.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 65. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**

Параллельные и высокопроизводительные вычисления

Пишите быстрые, мощные, энергоэффективные программы, легко масштабируемые под обработку огромных объемов данных.
Параллельное программирование позволяет распределять задачи обработки данных между несколькими процессорами, существенно повышая производительность. В книге рассказывается, как с минимальными трудозатратами повысить эффективность ваших программ. Вы научитесь оценивать аппаратные архитектуры и работать со стандартными инструментами отрасли, такими как OpenMP и MPI, освоите структуры данных и алгоритмы, подходящие для высокопроизводительных вычислений, узнаете, как экономить энергию на наладонных устройствах, и даже запустите масштабную симуляцию цунами на батарее из GPU-процессоров.

Издание предназначено для опытных программистов, владеющих высокопроизводительным вычислительным языком, таким как C, C++ или Fortran.

Рассматриваемые темы:

- планирование нового параллельного проекта;
- понимание различий в архитектуре CPU и GPU;
- решение проблем с неэффективными вычислительными ядрами и циклами;
- управление приложениями с помощью пакетного планировщика задач.

«Если вы хотите узнать о параллельном программировании и высокопроизводительных вычислениях на практических примерах, эта книга для вас».
Туан А. Тран, ExxonMobil

«Широкий обзор последних достижений в области параллельного и мультипроцессорного программного обеспечения».
Альберт Чой, OSI Digital Grid Solutions

«Углубленное изучение параллельных вычислений как с программной, так и с аппаратной точки зрения».
Жан Франсуа Морен, Университет Лаваля

«Эта книга покажет вам, как разрабатывать исходный код, в котором используются все возможности современных компьютеров».
Алессандро Кампейс, Vimar

Роберт Роби – сотрудник Лос-Аламосской национальной лаборатории. Уже более 30 лет он активно трудится в области параллельных вычислений. Джулиана Замора – аспирант и стипендиат по программе Siebel Scholars в Чикагском университете. Она читает лекции по программированию современного оборудования на многочисленных национальных конференциях.

ISBN 978-5-97060-936-1

Интернет-магазин: www.dmkpress.com

Оптовая продажа: КТК «Галактика»
books@alians-kniga.ru

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
www.дмк.рф