Серия
Суперкомпьютерное
Образование

Координационный совет
Системы научно-образовательных центров суперкомпьютерных технологий

Председатель Координационного совета

В. А. Садовничий,
ректор МГУ имени М. В. Ломоносова,
академик

Заместители председателя совета

Е. И. Моисеев,
декан факультета вычислительной математики и кибернетики
МГУ имени М. В. Ломоносова,
академик

А. В. Тихонравов,
директор Научно-исследовательского вычислительного центра
МГУ имени М. В. Ломоносова,
профессор

Члены совета

В. Н. Васильев, ректор Санкт-Петербургского национального исследовательского государственного университета информационных технологий, механики и оптики, чл.-корр. РАН, профессор; **М. А. Боровская**, ректор Южного федерального университета, профессор; **Н. Н. Кудрявцев**, ректор Московского физико-технического института, чл.-корр. РАН, профессор; **Г. В. Майер**, президент национального исследовательского Томского государственного университета, профессор; **С. В. Иванец**, ректор Дальневосточного федерального университета, профессор; **Е. В. Чупрунов**, ректор национального исследовательского Нижегородского государственного университета, профессор; **А. Л. Шестаков**, ректор национального исследовательского Южно-Уральского государственного университета, профессор; **В. Н. Чубариков**, декан механико-математического факультета МГУ имени М. В. Ломоносова, профессор; **М. И. Панасюк**, директор Научно-исследовательского института ядерной физики МГУ имени М. В. Ломоносова, профессор; **Вл. В. Воеводин**, заместитель директора Научно-исследовательского вычислительного центра МГУ имени М. В. Ломоносова, исполнительный директор НОЦ «СКТ-Центр», член-корреспондент РАН.

Московский государственный университет
имени М. В. Ломоносова

ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ НА GPU

Архитектура и программная модель CUDA

2-е издание

Допущено УМО по классическому университетскому образованию
в качестве учебного пособия для студентов высших учебных заведений,
обучающихся по направлениям ВПО
010400 «Прикладная математика и информатика»
и 010300 «Фундаментальная информатика
и информационные технологии»

Издательство Московского университета
2015

УДК 007 (075)
ББК 32.973.2
П18

Авторы:

А. В. Боресков, А. А. Харламов, Н. Д. Марковский, Д. Н. Микушин,
Е. В. Мортиков, А. А. Мыльцев, Н. А. Сахарных, В. А. Фролов

П18 **Параллельные** вычисления на GPU. Архитектура и программная модель CUDA: Учеб. пособие / А. В. Боресков и др. Предисл.: В. А. Садовничий. – 2-е издание. – М.: Издательство Московского университета, 2015. – 336 с., илл. – (Серия «Суперкомпьютерное образование»)

ISBN 978-5-19-011058-6

Данная книга представляет собой подробное практическое руководство по разработке приложений с использованием технологии NVIDIA CUDA версии 4. В первой части последовательно излагаются основы программной модели CUDA применительно к языкам C и Fortran, сведения о типах памяти GPU и методы эффективного использования разделяемой памяти на примере некоторых вычислительных алгоритмов. Во второй части дан обзор прикладных математических библиотек и языковых надстроек на основе CUDA. Специальные разделы книги посвящены элементам профессиональной разработки – средствам анализа, отладки и диагностики. Рассмотрены методы управления несколькими GPU на рабочих станциях и распределенных кластерных системах. Заключительная часть содержит несколько статей о применении CUDA в задачах математического моделирования гидродинамических процессов и компьютерной графике.

Книга предназначена для разработчиков и исследователей, применяющих параллельные вычисления.

Ключевые слова: CUDA, GPU, кластеры, отладка, уравнения Навье – Стокса, трассировка лучей, multi-GPU.

УДК 007 (075)
ББК 32.973.2

ISBN 978-5-19-011058-6

Уважаемый читатель!

Вы держите в руках одну из книг серии «Суперкомпьютерное образование», выпущенную в рамках реализации проекта комиссии Президента РФ по модернизации и технологическому развитию экономики России «Создание системы подготовки высококвалифицированных кадров в области суперкомпьютерных технологий и специализированного программного обеспечения». Инициатором издания выступил Суперкомпьютерный консорциум университетов России.

Серия включает более 20 учебников и учебных пособий, подготовленных ведущими отечественными специалистами в области суперкомпьютерных технологий. В книгах представлен ценный опыт преподавания суперкомпьютерных технологий в таких авторитетных вузах России, как МГУ, ННГУ, ТГУ, ЮУрГУ, СПбГУ ИТМО и многих других. При подготовке изданий были учтены рекомендации, сформулированные в Своде знаний и умений в области суперкомпьютерных технологий, подготовленном группой экспертов Суперкомпьютерного консорциума, а также международный опыт.

Современный уровень развития вычислительной техники и методов математического моделирования дает уникальную возможность для перевода промышленного производства и научных исследований на качественно новый этап. Эффективность такого перехода напрямую зависит от наличия достаточного числа высококвалифицированных специалистов. Данная серия книг предназначена для широкого круга студентов, аспирантов и специалистов, желающих изучить и практически использовать параллельные компьютерные системы для решения трудоемких вычислительных задач.

Издание серии «Суперкомпьютерное образование» наглядно демонстрирует тот вклад, который внесли участники Суперкомпьютерного консорциума университетов России в создание национальной системы

подготовки высококвалифицированных кадров в области суперкомпьютерных технологий, а также их четкое понимание ответственности за подготовку высококвалифицированных специалистов и формирование прочного научного фундамента, столь необходимого для эффективного использования суперкомпьютерных технологий на практике.

Ректор Московского университета,
Президент Суперкомпьютерного консорциума университетов России,
академик РАН *В.А. Садовничий*

Оглавление

Предисловие

Для меня большая честь и удовольствие писать предисловие к этой новой интересной книге. В настоящее время вычисления на GPU становятся очень актуальны, и эта книга появилась как нельзя кстати. GPU-вычисления в общем и вычисления для CUDA в частности привлекают все больший интерес во всем мире, в том числе среди Российских ученых и студентов. Авторы этой книги постарались качественно изложить как базовые аспекты программирования для CUDA, так и более сложные вопросы, связанные с иерархией памяти GPU, методами оптимизации приложений и использованием математических библиотек.

Первые две главы книги посвящены основам внутреннего устройства GPU и программной модели CUDA, в главе 3 подробно рассмотрена иерархия памяти GPU. В соответствии с высоким приоритетом научных и инженерных приложений, в главе 4 те же основные сведения изложены применительно к программам на языке Fortran. В главе 5 представлены некоторые типовые алгоритмы параллельной обработки данных для CUDA, в которых шаг за шагом применяются различные методы оптимизации. В главах 6, 9, 10 и 11 затрагиваются более сложные вопросы, включая архитектуру GPU, общие приемы оптимизации, программирование multi-GPU-систем, профилирование и анализ GPU-программ. Главы 7 и 8 содержат обзор прикладных библиотек и технологий, работающих на основе CUDA. В главах 12 и 13 приведены примеры несложных приложений с открытым исходным кодом, которые могут служить примерами для новых проектов.

Многие академические центры и отраслевые институты в России уже используют GPU- и CUDA-вычисления, и Россия стремительно становится одним из лидеров в этой технологии. Внедрение GPU-вычислений даст толчок новым научным открытиям, а также ускорит существующие приложения. Самая трудная задача заключается в обучении новых специалистов по вычислениям и прикладным областям, а также содействии сегодняшним профессионалам в совершенствовании навыков использования GPU-вычислений. Эта книга принесет пользу студентам и экспертам любого уровня. Она быстро станет классическим пособием в сфере GPU-вычислений.

Доктор Дэвид Б. Кирк, NVIDIA

Введение

Развитие архитектуры вычислительных систем – это история постоянного поиска баланса свойств, оптимального для множества целевых приложений. Пока не был исчерпан ресурс основных факторов роста, массовое производство и экономическая выгода сдерживали сколько-нибудь значительную *специализацию* основных вычислительных архитектур. Однако каждое новое инженерное решение в своем развитии со временем обнаруживало соответствующий противовес: частота и тепловыделение, многоядерность и когерентность кэшей, общая память и неоднородный доступ, конвейерность и ветвления и т.д. В условиях недостатка новых идей фактором роста в настоящее время становятся специализированные вычислители. Наибольший успех графических ускорителей (GPU) в этом качестве связан с их устойчивым положением в основной сфере применения.

Устройство архитектуры GPU можно кратко охарактеризовать как «макроархитектуру вычислительного кластера, реализованную в микромасштабе». GPU состоит из однородных вычислительных элементов с общей памятью. Каждый вычислительный элемент способен исполнять тысячи потоков, переключение между которыми не имеет накладных расходов. Потоки могут быть сгруппированы в блоки, имеющие общий кэш и быструю разделяемую память, явно контролируемую пользователем. Данная реализация в сочетании с расширениями для процедурных языков программирования носит название Compute Unified Device Architecture (CUDA).

Цель этой книги – дать достаточно полное практическое руководство по эффективному использованию CUDA 4.x на вычислительных системах различной сложности и в контексте других технологий. Полный исходный код примеров, рассмотренных в книге, можно найти на сайте [1].

Глава 1

От графических процессоров к GPGPU

1.1. Производительность и параллелизм

Одной из важнейших характеристик любого вычислительного устройства является производительность (performance). Для математических расчетов она обычно измеряется в количестве операций над вещественными данными в секунду (floating-point operations per second, FLOPS). В зависимости от того, учитывается ли только скорость расчета или также влияние других факторов, таких как скорость обмена данными, различают максимальную теоретически возможную *пиковую* и *реальную* производительность.

Производительность сильно зависит от тактовых частот центрального процессора (CPU) и памяти. Процессоры с высокой частотой и большим количеством интегрированной памяти могли бы обладать превосходной производительностью, но не могут быть массовыми из-за слишком высокой цены. По этой причине основной объем памяти находится в отделенных от процессора внешних модулях, и частота их работы в несколько раз ниже частоты процессора. Таким образом, *реальная* производительность системы при обработке больших массивов данных в значительной мере характеризуется именно скоростью работы внешней памяти и поэтому может быть значительно ниже *пиковой*.

Процессоры архитектуры x86 с момента своего появления в 1978 году увеличили свою тактовую частоту с 4,77 МГц до 3 ГГц, т.е. более чем в 600 раз, однако в последние несколько лет рост частоты более не наблюдается. Это связано как с ограничениями технологии производства микросхем, так и с тем, что энергопотребление (а значит и выделение тепла) пропорционально четвертой степени частоты. Таким образом, увеличение тактовой частоты всего в 2 раза приводит к увеличению тепловыделения в 16 раз! До сих пор с этим удавалось справляется за счет уменьшения размеров отдельных элементов микросхем. Дальнейшая миниатюризация связана со значительными трудностями, поэтому в настоящее время рост производительности идет в основном за счет увеличения числа параллельно работающих ядер, т.е. за счет *параллелизма*.

Максимальное ускорение, которое можно получить, распределив выполнение программы на N параллельно работающих элементов (процессоров, ядер), определяется законом Амдала (Amdahl Law):

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

В этой формуле P – это доля работы, которая может быть распараллелена. Видно, что при увеличении числа процессоров N ускорение стремится к $\frac{1}{1-P}$. Таким образом, если возможно распараллелить $\frac{3}{4}$ всех вычислений в программе, то при сколь-угодно большом числе доступных процессоров ускорение никогда не превысит 4 раз! Закон Амдала показывает, что возможная выгода от использования параллельных вычислений во многом предопределена свойствами применяемых в программе методов или алгоритмов.

1.2. Эволюция GPU

Термин Graphics Processing Unit (GPU) был впервые использован корпорацией NVIDIA для обозначения того факта, что графический ускоритель, первоначально используемый только для ускорения трехмерной графики, стал мощным программируемым устройством (процессором), пригодным для решения значительно более широкого класса вычислительных задач (General Purpose computations on GPU – GPGPU).

Современные GPU представляют собой массивно-паралельные вычислительные устройства с производительностью порядка Терафлопса и большим объемом собственной памяти (DRAM).

История GPU начиналась более чем скромно: первые графические ускорители Voodoo компании 3DFx лишь выполняли растеризацию (перевод треугольников в массивы пикселов) с поддержкой буфера глубины, наложение текстур и альфа-блендинг. При этом вся обработка вершин проводилась центральным процессором, и ускоритель получал на вход уже отображенные на экран (т.е. спроектированные) вершины. Однако именно эти очень простые задачи Voodoo умел решать намного быстрее, чем универсальный центральный процессор, что привело к широкому распространению графических ускорителей трехмерной графики.

Традиционные задачи рендеринга имеют значительный *ресурс параллелизма*: все вершины и фрагменты, полученные при растеризации треугольников, можно обрабатывать независимо друг от друга. Данное свойство типовых задач графических ускорителей определило их архитектурные принципы.

Ускорители трехмерной графики быстро эволюционировали, при этом помимо увеличения производительности, также росла и их функциональность. Так, графические ускорители следующего поколения (например, Riva TNT), уже могли обрабатывать вершины без участия CPU и одновременно накладывать несколько текстур. Важно отметить постепенное расширение возможностей *программируемой* обработки отдельных элементов с целью реализации ряда сложных эффектов, таких как попиксельное освещение. В GeForce 256 были впервые добавлены блоки register combiners, каждый из которых позволял выполнять простые вычислительные операции, например, скалярное произведение. Построение сложного эффекта сводилось к настройке и соединению входов и выходов множества таких блоков.

Следующим шагом стала поддержка в GeForce 2 вершинных программ на специальном ассемблере. Такие программы выполнялись параллельно для каждой вершины в 32-битной вещественной арифметике. Впоследствии подобная функциональность стала доступна уже на уровне отдельных фрагментов в серии GeForce FX. Благодаря тому, что графический ускоритель содержал как вершинные, так и фрагментные процессоры, соответствующие программы для вершин и фрагментов также выполнялись параллельно, что приводило к еще большему быстродействию. В целом графические ускорители на тот момент представляли собой мощные *SIMD-процессоры*.

Термином SIMD (Single Instruction Multiple Data) называют метод обработки, при котором одна и та же операция применяется одновременно ко множеству независимых данных. SIMD-процессор получает на вход поток однородных данных и параллельно обрабатывает их, порождая выходной поток (рис. 1.1). Программный модуль, описывающий подобное преобразование, называют *вычислительным ядром* (kernel). Отдельные ядра могут быть соединены между собой, образуя сложные составные схемы.

С введением поддержки текстур 16- и 32-битных вещественных элементов появился простой и универсальный метод обмена большими массивами данных меж-

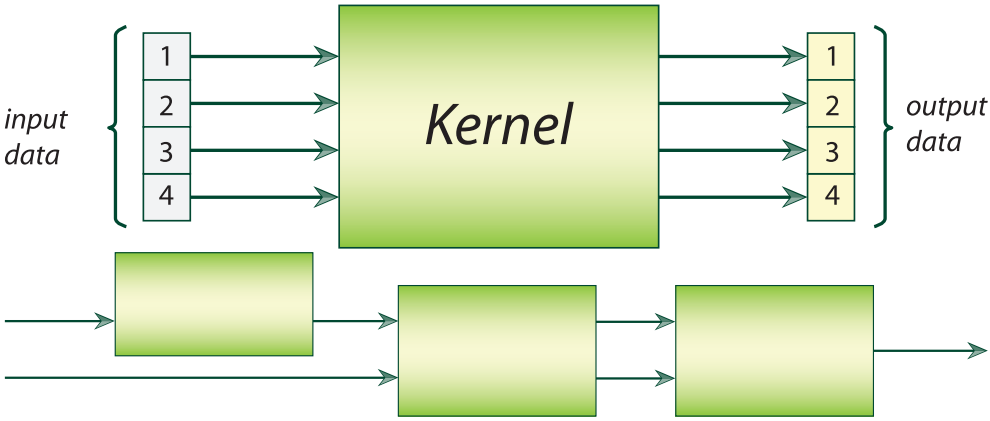


Рис. 1.1. Работа SIMD архитектуры

ду GPU и памятью основной системы. Высокоуровневые языки, такие как Cg, GLSL и HLSL, значительно упростили процесс написания программ для GPU. Ниже приводится пример программы (шейдера) на языке GLSL. Функциями OpenGL или Direct3D входные данные загружались в текстуры. Затем на графическом процессоре через операцию рендеринга (обычно – прямоугольника) запускалась программа обработки этих данных. Результат получался также в виде текстур, которые оставалось выгрузить обратно в системную память. Таким образом, программа состояла из двух частей, написанных на разных языках: C/C++ – для подготовки и передачи данных и язык шейдеров – для вычислений на GPU.

```
varying vec3 lt;
varying vec3 ht;

uniform sampler2D tangentMap;
uniform sampler2D decalMap;
uniform sampler2D anisoTable;

void main (void)
{
    const vec4  specColor = vec4 ( 0, 0, 1, 0 );
          vec3  tang      = normalize ( 2.0 * texture2D ( tangentMap,
                                        gl_TexCoord [0].xy ).xyz – 1.0);
    float       dot1      = dot  ( normalize ( lt ), tang );
    float       dot2      = dot  ( normalize ( ht ), tang );
          vec2  arg       = vec2 ( dot1, dot2 );
```

```
        vec2  ds        = texture2D ( anisoTable, arg*arg ).rg;
        vec4  color     = texture2D ( decalMap, gl_TexCoord [0].xy );

    gl_FragColor        = color * ds.x + specColor * ds.y;
    gl_FragColor.a      = 1.0;
}
```

Тем не менее, API, ориентированные на работу с графикой, имеют ряд ограничений, затрудняющих реализацию вычислительных алгоритмов. Так, отсутствует возможность какого-либо взаимодействия между параллельно обрабатываемыми пикселами, что для вычислительных задач является желательным. В частности, это приводит к отсутствию поддержки операции scatter, применяемой, например, при построении гистограмм: очередной элемент входных данных может приводить к изменению заранее неизвестного элемента (или элементов) гистограммы.

В последние годы на смену графическим API в GPGPU пришли программные системы, предназначенные именно для вычислений – CUDA, DirectCompute, OpenCL. Примечательно, что теперь они оказывают сильное обратное влияние и в сфере своих прародителей – графических приложений. Так, визуальные эффекты во многих современных играх основаны на численном решении дифференциальных уравнений в реальном времени с помощью GPU.

1.3. Сравнение архитектуры CPU и GPU

В чем же основные отличия архитектур GPU и CPU? Во-первых, CPU имеет лишь небольшое число ядер, работающих на высокой тактовой частоте независимо друг от друга. GPU же напротив работает на низкой тактовой частоте и имеет сотни сильно упрощенных вычислительных элементов (например, отсутствует предсказатель ветвлений и суперскалярное исполнение команд). Во-вторых, значительная доля площади кристалла CPU занята кэшем, в то время как практически весь GPU состоит из арифметико-логических блоков (рис. 1.2). В архитектуре GPU кэш имеет меньшее значение, поскольку используется принципиально иная стратегия покрытия латентности памяти. За счет этих отличий производительность каждого нового поколения GPU быстро растет как в пиковом значении (рис. 1.3), так и на реальных приложениях, например, Linpack (рис. 1.4).

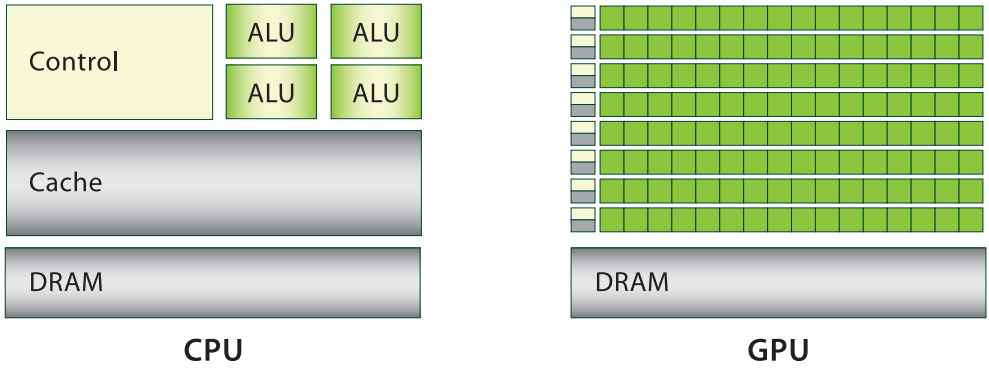


Рис. 1.2. GPU отводит большее колличество транзисторов под операции над данными

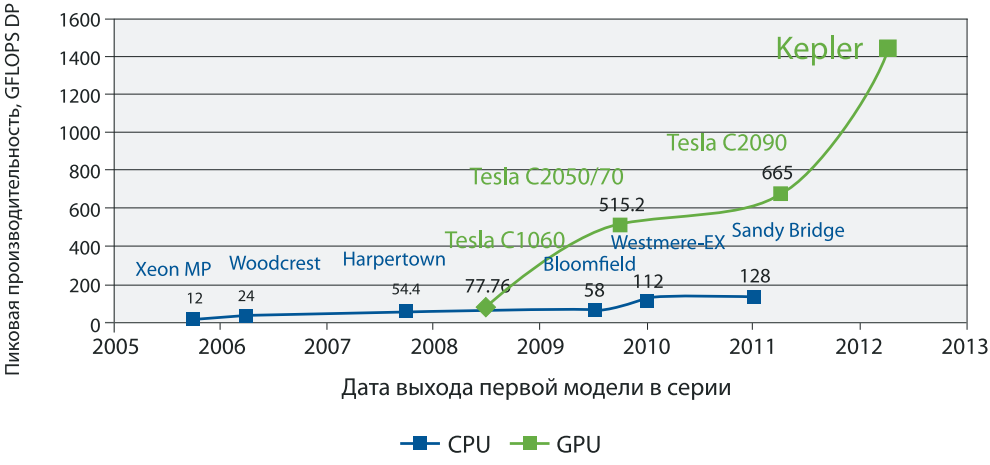


Рис. 1.3. Динамика роста пиковой производительности вычислений с двойной точностью CPU (Intel Xeon) и GPU (NVIDIA Tesla)

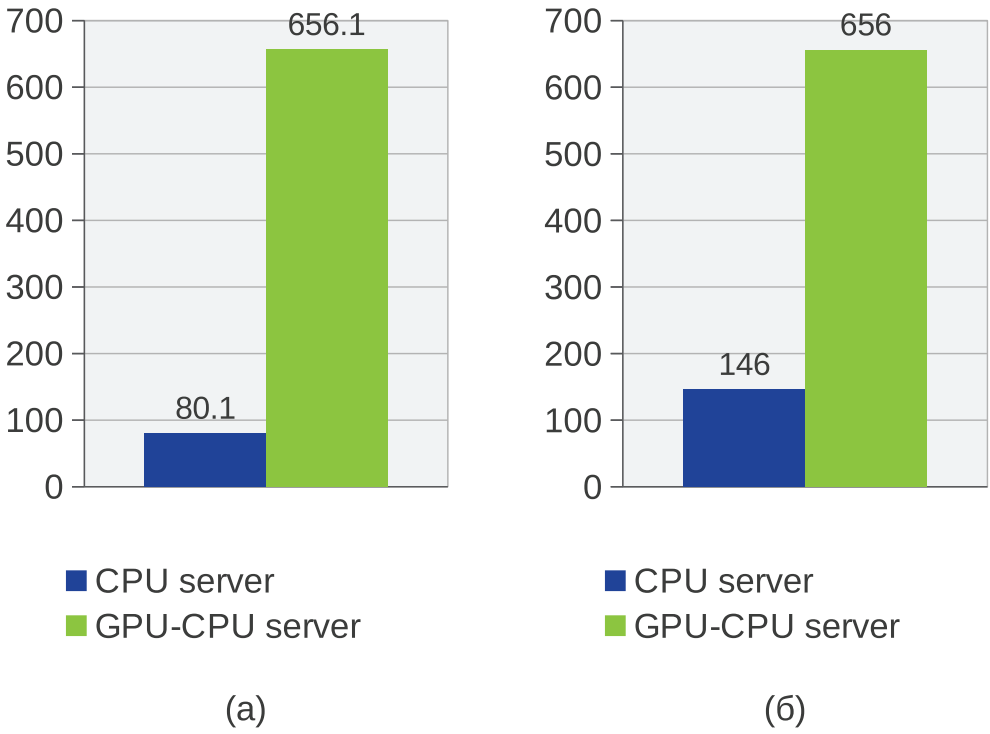


Рис. 1.4. Абсолютная производительность, GFLOPS (а) и производительность GFLOPS на Ватт (б) на тесте Linpack конфигураций 2x Intel Xeon X5550 2.66 GHz (CPU server) и 2x Tesla C2050 + 2x Intel Xeon X5550 (CPU-GPU server)

GPU наиболее эффективны при решении задач, обладающих *параллелизмом по данным*, число арифметических операций в которых велико по сравнению с операциями над памятью. Например, в 3D-рендеринге параллелизм по данным выражается в распределении по потокам обработки отдельных вершин. Аналогично, обработка изображений, кодирование и декодирование видео и распознавание образов легко делятся на подзадачи над блоками изображений и пикселов. Более того, множество задач, не связанных с графикой, также обладают параллелизмом по данным: обработка сигналов, физика, финансовый анализ, вычислительная биология и т.д.

Глава 2

Программная модель CUDA

Compute Unified Device Architecture (CUDA) – это программная модель, включающая описание вычислительного параллелизма и иерархичной структуры памяти непосредственно в язык программирования. С точки зрения программного обеспечения, реализация CUDA представляет собой кроссплатформенную систему компиляции и исполнения программ, части которых работают на CPU и GPU. CUDA предназначена для разработки GPGPU-приложений без привязки к графическим API и поддерживается всеми GPU NVIDIA, начиная с серии GeForce 8.

2.1. Основные принципы

Концепция CUDA отводит GPU роль массивно-параллельного *сопроцессора*. В литературе о CUDA основная система, к которой подключен GPU, для краткости называется термином *хост* (host), аналогично сам GPU по отношению к хосту часто называется просто *устройством* (device). CUDA-программа задействует как CPU, так и GPU, на CPU выполняется последовательная часть кода и подготовительные стадии для GPU-вычислений. Параллельные участки кода могут быть перенесены на GPU, где будут одновременно выполняться большим множеством *нитей* (threads). Важно отметить ряд принципиальных различий между обычными потоками CPU и нитями GPU:

- Нить GPU чрезвычайно легковесна, ее контекст минимален, регистры распределены заранее;
- Для эффективного использования ресурсов GPU программе необходимо задействовать тысячи отдельных нитей, в то время как на многоядерном CPU максимальная эффективность обычно достигается при числе потоков, равном или в несколько раз большем количества ядер.

В целом работа нитей на GPU соответствует принципу SIMD, однако есть существенное различие. Только нити в пределах одной группы (для GPU архитектуры Fermi – 32 нити), называемой *варпом* (warp) выполняются *физически одновре-*

менно. Нити различных варпов могут находиться на разных стадиях выполнения программы. Такой метод обработки данных обозначается термином SIMT (Single Instruction – Multiple Threads). Управление работой варпов производится на аппаратном уровне.

По ряду возможностей новых версий CUDA прослеживается тенденция к постепенному превращению GPU в самодостаточное устройство, полностью замещающее обычный CPU за счет реализации некоторых системных вызовов (в терминологии GPU системными вызовами являются, например, malloc и free, реализованные в CUDA 3.2) и добавления облегченного энергоэффективного CPU-ядра в сам GPU (архитектура Maxwell).

Важным преимуществом CUDA является использование для программирования GPU языков высокого уровня. В настоящее время существуют компиляторы C++ и Fortran. Эти языки расширяются небольшим множеством новых конструкций: атрибуты функций и переменных, встроенные переменные и типы данных, оператор запуска ядра.

2.2. Нити и блоки

Рассмотрим пример CUDA-программы, использующей GPU для поэлементного сложения двух одномерных массивов:

Листинг 2.1. sum_kernel.cu

```
// Ядро, выполняется параллельно на большом числе нитей.
__global__ void sum_kernel ( float * a, float * b, float * c )
{
    // Глобальный индекс нити.
    int idx = threadIdx.x + blockIdx.x * blockDim.x;

    // Выполнить обработку соответствующих данной нити данных.
    c [idx] = a [idx] + b [idx];
}

#include <stdio.h>

int sum_host( float * a, float * b, float * c, int n )
{
    int nb = n * sizeof ( float );
    float *aDev = NULL, *bDev = NULL, *cDev = NULL;
```

```
// Выделить память на GPU.
cudaError_t cuerr = cudaMalloc ( (void**)&aDev, nb );
if (cuerr != cudaSuccess)
{
    fprintf(stderr, "Cannot allocate GPU memory for aDev: %s\n",
        cudaGetErrorString(cuerr));
    return 1;
}
cuerr = cudaMalloc ( (void**)&bDev, nb );
if (cuerr != cudaSuccess)
{
    fprintf(stderr, "Cannot allocate GPU memory for bDev: %s\n",
        cudaGetErrorString(cuerr));
    return 1;
}
cuerr = cudaMalloc ( (void**)&cDev, nb );
if (cuerr != cudaSuccess)
{
    fprintf(stderr, "Cannot allocate GPU memory for cDev: %s\n",
        cudaGetErrorString(cuerr));
    return 1;
}

// Задать конфигурацию запуска n нитей.
dim3 threads = dim3(BLOCK_SIZE, 1);
dim3 blocks  = dim3(n / BLOCK_SIZE, 1);

// Скопировать входные данные из памяти CPU в память GPU.
cuerr = cudaMemcpy ( aDev, a, nb, cudaMemcpyHostToDevice );
if (cuerr != cudaSuccess)
{
    fprintf(stderr, "Cannot copy data from a to aDev: %s\n",
        cudaGetErrorString(cuerr));
    return 1;
}
cuerr = cudaMemcpy ( bDev, b, nb, cudaMemcpyHostToDevice );
if (cuerr != cudaSuccess)
{
    fprintf(stderr, "Cannot copy data from b to bDev: %s\n",
        cudaGetErrorString(cuerr));
    return 1;
}

// Вызвать ядро с заданной конфигурацией для обработки данных.
sum_kernel<<<blocks, threads>>> (aDev, bDev, cDev);
```

```
    cuerr = cudaGetLastError();
    if (cuerr != cudaSuccess)
    {
        fprintf(stderr, "Cannot launch CUDA kernel: %s\n",
            cudaGetErrorString(cuerr));
        return 1;
    }

    // Ожидать завершения работы ядра.
    cuerr = cudaDeviceSynchronize();
    if (cuerr != cudaSuccess)
    {
        fprintf(stderr, "Cannot synchronize CUDA kernel: %s\n",
            cudaGetErrorString(cuerr));
        return 1;
    }

    // Скопировать результаты в память CPU.
    cuerr = cudaMemcpy ( c, cDev, nb, cudaMemcpyDeviceToHost );
    if (cuerr != cudaSuccess)
    {
        fprintf(stderr, "Cannot copy data from cdev to c: %s\n",
            cudaGetErrorString(cuerr));
        return 1;
    }

    // Освободить выделенную память GPU.
    cudaFree ( aDev );
    cudaFree ( bDev );
    cudaFree ( cDev );

    return 0;
}
```

Листинг 2.2. sum_kernel.CUF

```
! Ядро, выполняется параллельно на большом числе нитей.
attributes(global) subroutine sum_kernel (a, b, c )

    implicit none

    real, device, dimension(*) :: a, b, c
    integer :: idx

    ! Глобальный индекс нити.
    idx = threadIdx%x + (blockIdx%x - 1) * blockDim%x
```

```
    ! Выполнить обработку соответствующих данной нити данных.
    c (idx) = a (idx) + b (idx)
end

function sum_host (a, b, c, n )

    use cudafor
    implicit none

    real, dimension(n), intent(in) :: a, b
    real, dimension(n), intent(out) :: c
    integer, intent(in) :: n

    real, device, allocatable, dimension(:) :: aDev, bDev, cDev
    integer :: sum_host, istat

    type(dim3) :: blocks, threads

    sum_host = 1
    istat = 0

    ! Выделить память на GPU.
    allocate(aDev(n), stat = istat)
    if (istat .ne. 0) then
        write(*, *) 'Cannot allocate GPU memory for aDev: ', &
            cudaGetErrorString(istat)
        return
    endif
    allocate(bDev(n), stat = istat)
    if (istat .ne. 0) then
        write(*, *) 'Cannot allocate GPU memory for bDev: ', &
            cudaGetErrorString(istat)
        return
    endif
    allocate(cDev(n), stat = istat);
    if (istat .ne. 0) then
        write(*, *) 'Cannot allocate GPU memory for cDev: ', &
            cudaGetErrorString(istat)
        return
    endif

    ! Задать конфигурацию запуска n нитей.
    threads = dim3(BLOCK_SIZE, 1, 1);
    blocks  = dim3(n / BLOCK_SIZE, 1, 1);
```

```
    ! Скопировать входные данные из памяти CPU в память GPU.
    aDev = a
    bDev = b

    ! Вызвать ядро с заданной конфигурацией для обработки данных.
    call sum_kernel<<<blocks, threads>>> (aDev, bDev, cDev);
    istat = cudaGetLastError();
    if (istat .ne. cudaSuccess) then
        write(*, *) 'Cannot launch CUDA kernel: ', &
            cudaGetErrorString(istat)
        return
    endif
    ! Ожидать завершения работы ядра.
    istat = cudaThreadSynchronize();
    if (istat .ne. cudaSuccess) then
        write(*, *) 'Cannot synchronize CUDA kernel: ', &
            cudaGetErrorString(istat)
        return
    endif

    ! Скопировать результаты в память CPU.
    c = cDev

    ! Освободить выделенную память GPU.
    deallocate(aDev)
    deallocate(bDev)
    deallocate(cDev)

    sum_host = 0
    return
end
```

Функция (или процедура в случае Fortran) *sum_kernel* является *ядром* (атрибут *__global__* или *global*) и будет выполнять на GPU по одной независимой нити для каждого набора элементов *a[i]*, *b[i]* и *c[i]*. Нить GPU имеет координаты во вложенных трехмерных декартовых равномерных сетках «индексы блоков» и «индексы потоков внутри каждого блока», двумерный случай показан на рис. 2.1. В контексте каждой нити значения координат и размерностей доступны через встроенные переменные *threadIdx*, *blockIdx* и *blockDim*, *gridDim* соответственно. Если проводить аналогию с Message Passing Interface (MPI), то значения этих переменных по смыслу аналогичны результатам функций *MPI_Comm_rank* и *MPI_Comm_size*.

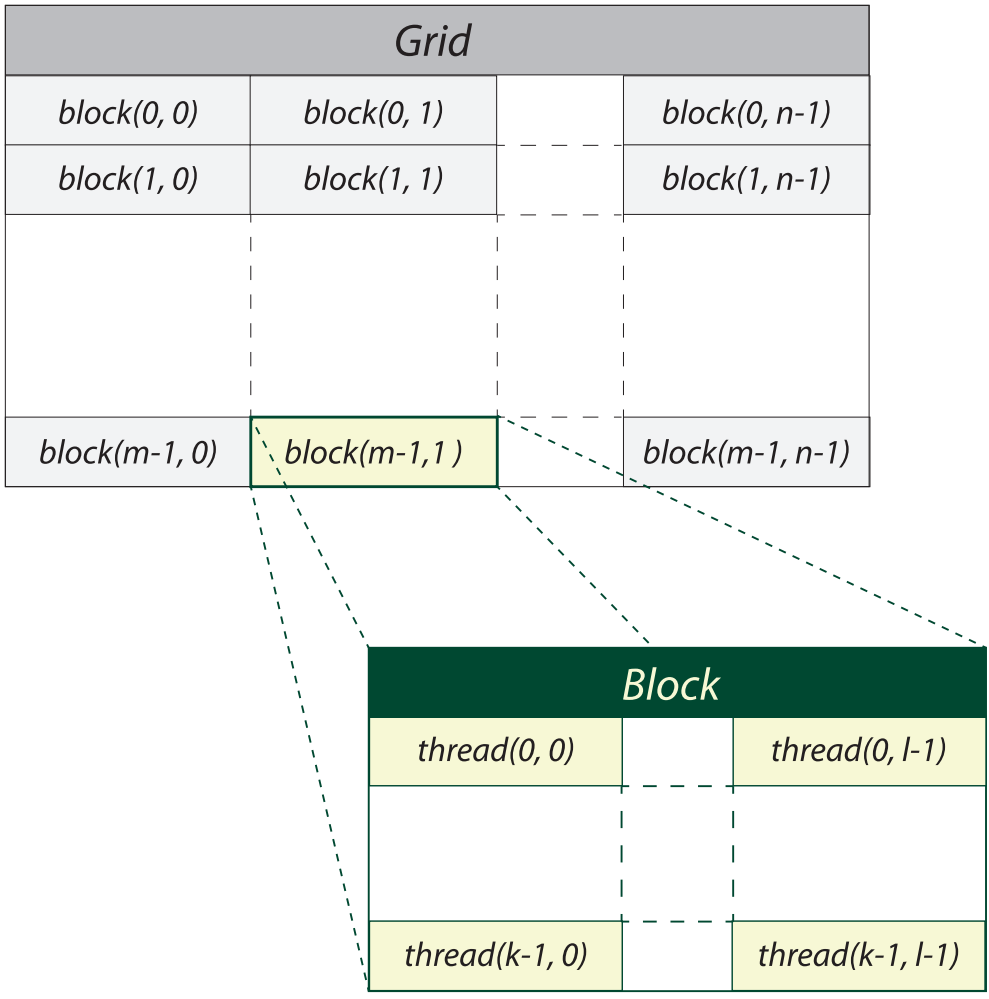


Рис. 2.1. Иерархия нитей в CUDA

Код ядра *sum_kernel* начинается с определения глобального индекса массива *idx*, зависящего от координат нити. В общем случае соответствие нитей и частей задачи может быть любым, например, одна нить может обрабатывать не один элемент массива, а определенный диапазон. В течение времени работы ядра декартовы сетки нитей и блоков зафиксированы для отображения на аппаратный уровень вычислительных элементов GPU. Каждая нить производит сложение элементов массивов *a* и *b* и помещает результат в элемент массива *c*. После этого нить завершает работу.

При использовании глобальной памяти, расположенной физически на самом GPU, управлять ею можно с хоста. В функции *sum_host* память, выделяемая на GPU, заполняется копией данных из памяти хоста, затем производится запуск ядра *sum_kernel*, синхронизация и копирование результатов обратно в память хоста. В конце производится высвобождение ранее выделенной глобальной памяти GPU. Такая последовательность действий характерна для любого CUDA-приложения.

Общим приемом программирования для CUDA является группировка множества нитей в блоки. На это есть две причины. Во-первых, далеко не для каждого параллельного алгоритма существует эффективная реализация на полностью независимых нитях: результат одной нити может зависеть от результата некоторых других, исходные данные нити могут частично совпадать с данными соседних. Во-вторых, размер одной выборки данных из глобальной памяти намного больше размера вещественного или целочисленного типа, т.е. одна выборка может покрыть запросы группы из нескольких нитей, работающих с подряд идущими в памяти элементами. В результате группировки нитей исходная задача распадается на независимые друг от друга подзадачи (блоки нитей) с возможностью взаимодействия нитей в рамках одного блока и объединения запросов в память в рамках одного варпа (рис. 2.2). Разбиение нитей на варпы также происходит отдельно для каждого блока. Объединение в блоки является удачным компромиссом между необходимостью обеспечить взаимодействие нитей между собой и возможностью сделать соответствующую аппаратную логику эффективной и дешевой.

На время выполнения ядра каждый блок получает в распоряжение часть быстрой разделяемой памяти, которую могут совместно использовать все нити блока. Поскольку не все нити блока выполняются физически одновременно, то для их вза-

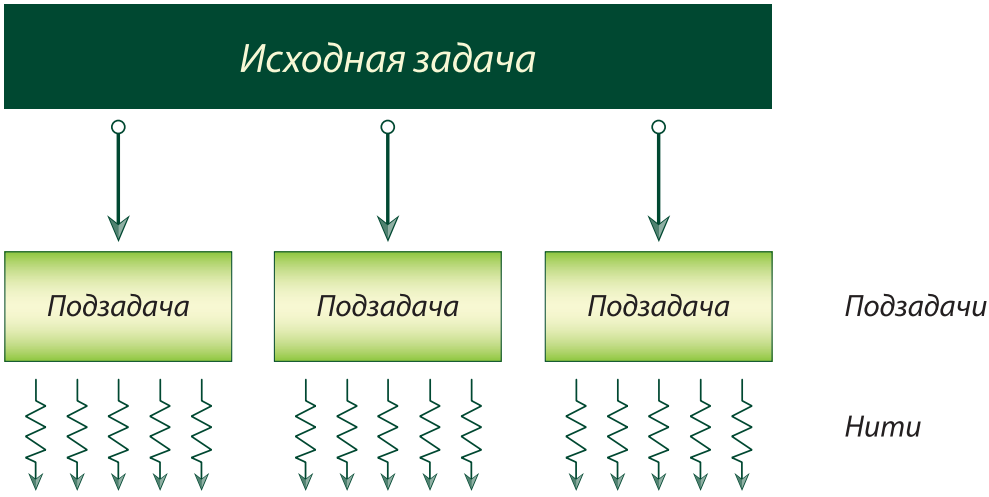


Рис. 2.2. Разбиение исходной задачи на набор независимо решаемых подзадач

имного согласования необходим механизм синхронизации. Для этой цели в CUDA предусмотрен вызов __syncthreads(), который блокирует дальнейшее исполнение кода ядра до тех пор, пока все нити блока не войдут в эту функцию (рис. 2.3).

2.3. Расширения языка

Расширения языка для CUDA можно объединить в следующие группы:

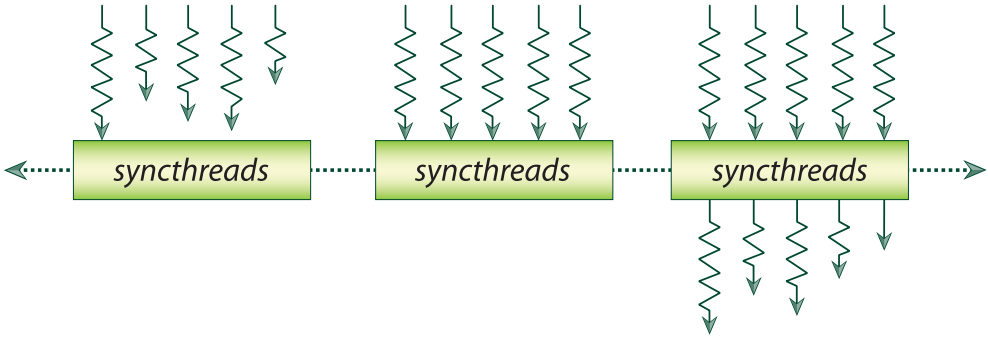


Рис. 2.3. Барьерная синхронизация

- Атрибуты функций – показывают, где будет выполняться функция и откуда она может быть вызвана;
- Атрибуты переменных – задают тип используемой памяти;
- Оператор запуска ядра – определяет иерархию нитей, очередь команд (CUDA Stream) и размер разделяемой памяти;
- Встроенные переменные – содержат контекстную информацию относительно текущей нити;
- Дополнительные типы данных – определяют несколько новых векторных типов.

В настоящее время существуют два компилятора с поддержкой CUDA: реализация CUDA для языка C++ компании NVIDIA, основанная на open-source компиляторе Open64 [2], [3] и реализация CUDA для Fortran компании Portland Group Inc. (PGI) с закрытой лицензией. Стандартное расширение имен исходных файлов – .cu или .cuf (.CUF – с проходом через препроцессор) соответственно. В случае если CPU-программа также написана на C++ или Fortran, части кода для CPU и GPU могут объединяться в общие модули компиляции.

2.3.1. Атрибуты функций и переменных

В CUDA используются следующие атрибуты функций:

Таблица 2.1. Атрибуты функций и переменных в CUDA

Атрибут C	**Атрибут Fortran**	**Функция выполняется на**	**Функция может вызываться из**
__device__	device	device (GPU)	device (GPU)
__global__	global	device (GPU)	host (CPU)
__host__	host	host (CPU)	host (CPU)

Атрибуты __host__ (host) и __device__ (device) могут быть использованы вместе (это значит, что соответствующая функция может выполняться как на GPU, так и

на CPU – соответствующий код для обеих платформ будет автоматически сгенерирован компилятором). Атрибуты __global__ и __host__ не могут быть использованы вместе. Атрибут __global__ обозначает ядро, и соответствующая функция CUDA C должна возвращать значение типа void, CUDA Fortran – быть процедурой. На функции, выполняемые на GPU (__device__ и __global__), накладываются следующие ограничения:

- не поддерживаются static-переменные внутри функции;
- не поддерживается переменное число входных аргументов.

Для задания способа размещения переменных в памяти GPU используются следующие атрибуты: __device__, __constant__ и __shared__. На их использование также накладывается ряд ограничений:

- атрибуты не могут быть применены к полям структуры (struct или union);
- соответствующие переменные могут использоваться только в пределах одного файла, их нельзя объявлять как extern;
- запись в переменные типа __constant__ может осуществляться только CPU при помощи специальных функций;
- __shared__ переменные не могут инициализироваться при объявлении.

Пока существующие компиляторы для CUDA не реализуют поддержку модульной сборки GPU-кода, каждая __global__-функция должна находиться в одном исходном файле вместе со всеми __device__-функциями и переменными, которая она использует.

2.3.2. Встроенные типы

В язык добавлены 1/2/3/4-мерные векторные типы на основе *char*, *unsigned char*, *short*, *unsigned short*, *int*, *unsigned int*, *long*, *unsigned long*, *longlong*, *float* и *double*. Имя векторного типа формируется из базового имени и числа элементов, например, *float4*.

Компоненты векторных типов имеют имена x, y, z и w. Для создания значений-векторов заданного типа служит конструкция вида make_<typeName>:

```
int2   a = make_int2   ( 1, 7 );       // Создает вектор (1, 7).
float3 u = make_float3 ( 1, 2, 3.4f ); // Создает вектор (1.0f, 2.0f, 3.4f ).
```

В отличие от шейдерных языков GLSL, Cg и HLSL, для этих типов не поддерживаются векторные покомпонентные операции, т.е. нельзя просто сложить два вектора при помощи оператора «+», это необходимо делать отдельно для каждой компоненты.

Также добавлен тип dim3, используемый для задания размерностей сеток нитей и блоков. Этот тип основан на типе uint3, и обладает нормальным конструктором, по умолчанию инициализирующим компоненты единицами.

```
dim3 blocks ( 16, 16 ); // Эквивалентно blocks ( 16, 16, 1 ).
dim3 grid   ( 256 );    // Эквивалентно grid ( 256, 1, 1 ).
```

2.3.3. Встроенные переменные

В язык добавлены следующие специальные переменные:

- gridDim – размер сетки (имеет тип dim3);
- blockDim – размер блока (имеет тип dim3);
- blockIdx – индекс текущего блока в сетке (имеет тип uint3);
- threadIdx – индекс текущей нити в блоке (имеет тип uint3);
- warpSize – размер варпа (имеет тип int).

2.3.4. Оператор вызова GPU-ядра

Для запуска ядра на GPU используется следующая конструкция:

```
kernel_name <<<Dg,Db,Ns,S>>> ( args );
```

Здесь *kernel_name* – это имя или адрес соответствующей __global__-функции. Параметр *Dg* типа *dim3* задает размерности сетки блоков (число блоков в сетке блоков), параметр *Db* типа *dim3* задает размерности блока нитей (число нитей в

блоке). Необязательный параметр *Ns* типа *size_t* задает дополнительный объем разделяемой памяти в байтах (по умолчанию – 0), которая должна быть динамически выделена каждому блоку (в дополнение к статически выделенной). Параметр *S* типа *cudaStream_t* ставит вызов ядра в определенную очередь команд (CUDA Stream), по умолчанию – 0. Вызов функции-ядра также может иметь произвольное фиксированное число параметров, суммарный размер которых не превышает 4 Кбайт. Следующий пример запускает ядро *my_kernel* параллельно на *n* нитях, используя одномерный массив из двумерных блоков нитей 16×16, и передает на вход ядру два параметра – *a* и *n*. При этом каждому блоку дополнительно выделяется 512 байт разделяемой памяти и запуск ядра производится в очереди команд *my_stream*.

```
my_kernel<<<dim3(n / 256), dim3(16, 16), 512, my_stream>>> ( a, n );
```

2.3.5. Встроенные функции

Для CUDA реализованы математические функции, совместимые с ISO C. Также имеются соответствующие аналоги, вычисляющие результат с пониженной точностью, например, *__sinf* для *sinf*.

2.4. CUDA runtime API

Помимо компилятора, реализация CUDA должна предоставлять удобную систему взаимодействия с хост-системой и другими API. Для этой цели служит CUDA Runtime API – библиотека функций, обеспечивающих:

- управление GPU;
- работу с контекстом;
- работу с памятью;
- работу с модулями;
- управление выполнением кода;
- работу с текстурами;

- взаимодействие с OpenGL и Direct3D.

Теоретически GPU мог бы самостоятельно контролировать большую часть данного функционала, однако текущий дизайн библиотек таков, что в CUDA-ядре присутствует минимальное количество возможностей, прямо не связанных с вычислениями, а все функции CUDA Runtime API исполняются на CPU.

CUDA Runtime API делится на два уровня: CUDA Driver API и CUDA API. Вызовы обоих уровней доступны CUDA-приложению и в целом аналогичны, за исключением ряда особенностей. В частности, driver API является более низкоуровневым и требует явной инициализации устройства, тогда как в CUDA API неявная инициализация происходит при первом вызове любой функции библиотеки. Напрямую взаимодействовать с GPU может только драйвер устройства, на нем базируются CUDA Driver API, CUDA Runtime API и прикладные библиотеки (рис. 2.4). Во всех примерах этой книги используется CUDA API.

2.4.1. Асинхронное исполнение

Многие функции CUDA являются *асинхронными*, т.е. управление в вызывающую функцию возвращается до завершения требуемой операции. К числу асинхронных операций относятся:

- запуск ядра;
- функции копирования и инициализации памяти, имена которых оканчиваются на Async;
- функции копирования памяти device ↔ device внутри устройства и между устройствами;

С асинхронностью связаны объекты CUDA streams (потоки исполнения), позволяющие группировать последовательности операций, которые необходимо выполнять в строго определенном порядке. При этом порядок выполнения операций между разными CUDA streams не является строго определенным и может изменяться. По умолчанию все асинхронные операции выполняются в нулевом CUDA

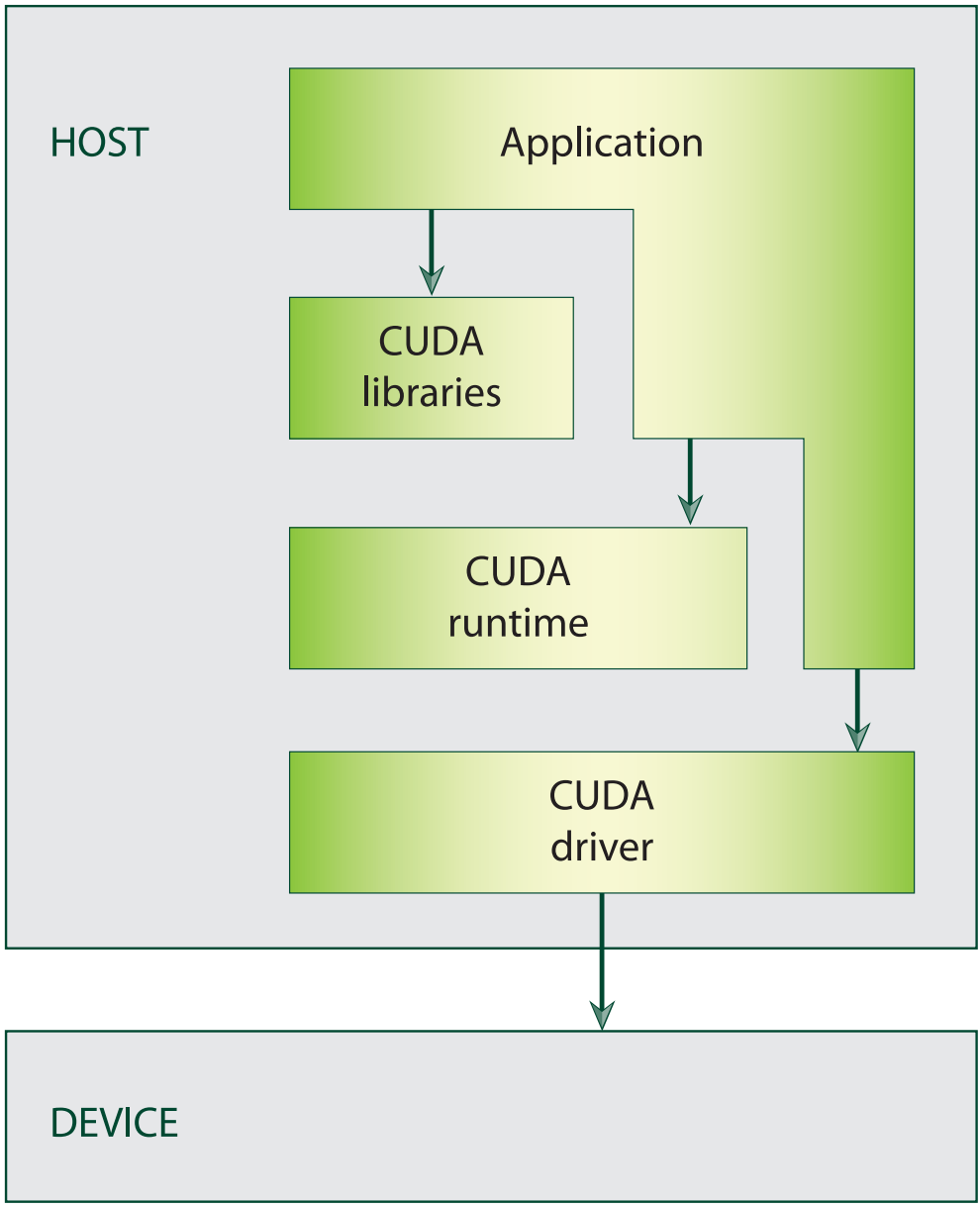


Рис. 2.4. Программный стек CUDA

stream, т.е. последовательны между собой и асинхронны по отношению к остальным вызовам программы. Часто бывает необходимо в определенный момент дождаться завершения CUDA-ядра, т.е. выполнить *синхронизацию*. Для явной синхронизации всех CUDA streams текущего устройства с текущим потоком (thread) CPU используется функция *cudaDeviceSynchronize* (до CUDA версии 4.0 – функция *cudaThreadSynchronize*). Во многих случаях синхронизация происходит неявно, например, при копировании результатов с GPU по адресу, ранее использованному при запуске ядра, основной поток будет заблокирован на копировании до завершения работы ядра. Также неявная синхронизация происходит при использовании CUDA Events.

```
cudaError_t cudaDeviceSynchronize();

cudaError_t cudaStreamSynchronize(cudaStream stream);
```

2.4.2. Обработка ошибок в CUDA

Каждая функция CUDA runtime API (кроме запуска ядра) возвращает значение типа *cudaError_t*. При успешном выполнении функции возвращается значение *cudaSuccess*, в противном случае возвращается код ошибки. Получить текстовое описание ошибки позволяет функция cudaGetErrorString:

```
char* cudaGetErrorString ( cudaError_t code );
```

Также можно получить код последней ошибки:

```
cudaError_t cudaGetLastError();

cudaError_t cudaPeekAtLastError();
```

Эти две функции отличаются тем, что в случае устранимой ошибки вызов *cudaGetLastError* приводит к сбросу хранимого значения, т.е. при повторном вызове всегда возвращается *cudaSuccess*. Если же *cudaGetLastError* постоянно возвращает один и тот же код ошибки, то устройство находится в некорректном состоянии и может быть из него выведено повторной инициализацией:

```
cudaError_t cudaDeviceReset();
```

2.4.3. Доступ к свойствам установленных GPU

Возможности и ресурсы GPU различных моделей могут существенно отличаться. Например, некоторые GPU не поддерживают конкурентное исполнение нескольких ядер или имеют только двумерную сетку блоков. Если ставится цель обеспечить в приложении поддержку различных GPU, то свойства доступного GPU потребуется анализировать программно, аналогично тому, как для CPU проверяется наличие расширенных наборов инструкций. Для обозначения возможностей GPU CUDA использует понятие compute capability, выражаемое парой целых чисел – *major.minor*. Первое число обозначает глобальную архитектурную версию, второе – небольшие изменения. Так, GPU GeForce 8800 Ultra/GTX/GTS имеют compute capability 1.0, а современные GPU архитектуры Fermi – 2.x.

Получить детальную информацию обо всех установленных GPU позволяет вызов *cudaGetDeviceProperties*. Параметры возвращаются в передаваемую по указателю структуру *cudaDeviceProp*. Ниже приведен полный код программы. Также в наборе примеров CUDA SDK имеется готовая программа *deviceQuery*.

```
struct cudaDeviceProp
{
    char name[256];                 // Название устройства.
    size_t totalGlobalMem;          // Полный объем глобальной памяти в байтах.
    size_t sharedMemPerBlock;       // Объем разделяемой памяти в блоке в байтах.
    int regsPerBlock;               // Количество 32-битных регистров в блоке.
    int warpSize;                   // Размер варпа.
    size_t memPitch;                // Максимальный pitch в байтах, допустимый
                                    // функциями копирования памяти, выделенной
                                    // cudaMallocPitch
    int maxThreadsPerBlock;         // Максимальное число активных нитей в блоке.
    int maxThreadsDim      [3];     // Максимальный размер блока по каждому
                                    // измерению.
    int maxGridSize        [3];     // Максимальный размер сетки по каждому
                                    // измерению.
    int maxTexture1D;               // Максимальный размер 1D текстуры
    int maxTexture2D       [2];     // Максимальный размер 2D текстуры
    int maxTexture3D       [3];     // Максимальный размер 3D текстуры
    int maxTexture1DLayered[2];     // Максимальный размер массива 1D текстур -
                                    // (dim, layers)
    int maxTexture2DLayered[3];     // Максимальный размер массива 2D текстур -
                                    // (dim1, dim2, layers)
    size_t totalConstMem;           // Объем константной памяти в байтах.
    int major;                      // Compute capability, старший номер.
```

```
    int minor;                    // Compute capability, младший номер.
    int clockRate;                // Частота в килогерцах.
    size_t surfaceAlignment;      // Выравнивание памяти для поверхностей.
    size_t textureAlignment;      // Выравнивание памяти для текстур.
    int deviceOverlap;            // Можно ли осуществлять копирование
                                  // параллельно с вычислениями.
    int asyncEngineCount;         // Колличество операций копирования,
                                  // выполняемых параллельно
    int concurrentKernels;        // 1, если есть возможность одновременного
                                  // выполнения ядер
    int kernelExecTimeoutEnabled; // 1, если введено ограничение на время
                                  // выполнения ядер
    int multiProcessorCount;      // Количество мультипроцессоров в GPU.
    int kernelExecTimeoutEnables; // 1, если есть ограничение на время
                                  // выполнения ядра
    int integrated;               // 1, если GPU встроено в материнскую плату
    int canMapHostMemory;         // 1, если можно отображать память CPU в
                                  // память CUDA - для использования
                                  // функциями cudaHostAlloc,
                                  // cudaHostGetDevicePointer
    int computeMode;              // Режим в котором находится GPU.
                                  // Возможные значения:
                                  // cudaComputeModeDefault,
                                  // cudaComputeModeExclusive - только одна
                                  // нить может вызывать cudaSetDevice для
                                  // данного GPU,
                                  // cudaComputeModeProhibited - ни одна
                                  // нить не может вызывать cudaSetDevice для
                                  // данного GPU,
                                  // cudaComputeModeExclusiveProcess - нити
                                  // единственного процесса могут вызывать
                                  // cudaSetDevice для данного GPU.
    int ECCEnabled;               // 1, если устройстве установлена поддержка
                                  // ECC
    int pciBusID;                 // ID PCI шины устройства
    int pciDeviceID;              // PCI ID устройства
    int tccDriver;                // 1, если подключено устройство Tesla и
                                  // используется TCC драйвер
    int unifiedAddressing;        // Есть ли поддержка режима UVA
}
```

Листинг 2.3. label=info.cu

```
#include <stdio.h>

int main ( int argc, char * argv [] )
```

```
{
    int            deviceCount;
    cudaDeviceProp dp;

    cudaGetDeviceCount ( &deviceCount );

    printf ( "Found %d devices\n", deviceCount );

    for ( int device = 0; device < deviceCount; device++ )
    {
        cudaGetDeviceProperties ( &dp, device );

        printf ( "Device %d\n", device );
        printf ( "Compute capability     : %d.%d\n", dp.major, dp.minor );
        printf ( "Name                   : %s\n", dp.name );
        printf ( "Total Global Memory    : %d\n", dp.totalGlobalMem );
        printf ( "Shared memory per block: %d\n", dp.sharedMemPerBlock );
        printf ( "Registers per block    : %d\n", dp.regsPerBlock );
        printf ( "Warp size              : %d\n", dp.warpSize );
        printf ( "Max threads per block  : %d\n", dp.maxThreadsPerBlock );
        printf ( "Total constant memory  : %d\n", dp.totalConstMem );
        printf ( "Clock Rate             : %d\n", dp.clockRate );
        printf ( "Texture Alignment      : %u\n", dp.textureAlignment );
        printf ( "Device Overlap         : %d\n", dp.deviceOverlap );
        printf ( "Multiprocessor Count   : %d\n", dp.multiProcessorCount );
        printf ( "Max Threads Dim        : %d %d %d\n", dp.maxThreadsDim [0],
                                 dp.maxThreadsDim [1],
                                 dp.maxThreadsDim [2] );
        printf ( "Max Grid Size          : %d %d %d\n", dp.maxGridSize [0],
                                 dp.maxGridSize [1],
                                 dp.maxGridSize [2] );
    }

    return 0;
}
```

2.5. Атомарные операции

Атомарные операции вводятся для обеспечения корректного доступа к разделяемому ресурсу в параллельной программе. В случае CUDA разделяемый ресурс – это переменная, доступная множеству параллельных нитей. При атомарном изменении значения переменной параллельные запросы будут обрабатываться так, чтобы запись никогда не происходила одновременно с чтением или еще одной попыт-

кой записи, т.е. была непрерываема или атомарна. GPU с compute capability 1.1 и выше поддерживают атомарные операции над глобальной памятью. GPU с compute capability 1.2 и выше поддерживают атомарные операции над 64-битными значениями и значениями в разделяемой памяти. Поддержка операций над 64-битными целыми числами в разделяемой памяти доступна в устройствах начиная с compute capability 2.x. Все атомарные операции, за исключением *atomicExch* и *atomicAdd*, работают только с целыми числами. Операция *atomicAdd* может работать с 32-битными числами с плавающей точкой только в устройствах с compute capability 2.x и выше.

2.5.1. Атомарные арифметические операции

Наиболее часто используются атомарные арифметические операции *atomicAdd* и *atomicSub*, позволяющие увеличить или уменьшить значение переменной на заданную величину. В качестве результата возвращается исходное значение. Версия *atomicAdd* для 32-битных вещественных чисел поддерживается GPU с compute capability 2.x и выше.

```
              int atomicAdd ( int                * addr,
                              int                  value );
unsigned      int atomicAdd ( unsigned int       * addr,
                              unsigned int         value );
unsigned long long atomicAdd ( unsigned long long * addr,
                               unsigned long long   value );
            float atomicAdd ( float              * addr,
                              float                value );
unsigned      int atomicAdd ( unsigned int       * addr,
                              unsigned int         value );
              int atomicSub ( int                * addr,
                              int                  value );
unsigned      int atomicSub ( unsigned int       * addr,
                              unsigned int         value );
```

Операция *atomicExch* производит атомарный обмен значениями: новое значение записывается по указанному адресу, а предыдущее возвращается как результат. Обмен происходит как одна транзакция, т.е. ни одна нить не может «вклиниться» между его этапами.

```
              int atomicExch ( int                * addr,
                               int                  value );
```

```
unsigned       int atomicExch ( unsigned int       * addr,
                                unsigned int         value );
unsigned long long atomicExch ( unsigned long long * addr,
                                unsigned long long   value );
             float atomicExch ( float              * addr,
                                float                value );
```

Следующие две операции сравнивают значение по адресу с переданным значением, записывают минимум/максимум из этих двух значений по заданному адресу и возвращают предыдущее значение, находившееся по адресу. Все эти шаги выполняются атомарно, как одна транзакция.

```
               int atomicMin ( int                * addr,
                               int                  value );
unsigned       int atomicMin ( unsigned int       * addr,
                               unsigned int         value );
               int atomicMax ( int                * addr,
                               int                  value );
unsigned       int atomicMax ( unsigned int       * addr,
                               unsigned int         value );
```

Операция *atomicInc* читает слово по заданному адресу и сравнивает его с переданным значением. Если прочитанное слово больше, то по адресу записывается ноль, иначе значение по адресу увеличивается на единицу. Возвращается старое значение.

```
unsigned int atomicInc ( unsigned int * addr, unsinged int value );
```

Операция *atomicDec* читает слово по переданному адресу. Если прочитанное значение равно нулю или больше переданного значения, то записывает по адресу переданное значение, иначе уменьшает значение по адресу на единицу. Возвращается старое значение.

```
unsigned       int atomicDec ( unsigned int       * addr,
                               unsigned int         value );
```

Функция *atomicCAS* (CAS – Compare And Swap) читает старое 32- или 64-битное значение по переданному адресу и сравнивает его с параметром *compare*. В случае совпадения по переданному адресу записывается значение параметра *value*, иначе значение по адресу не изменяется. Во всех случаях возвращается старое прочитанное значение.

```
              int atomicCAS ( int                * addr,
                              int                  compare,
                              int                  value );
unsigned      int atomicCAS ( unsigned int       * addr,
                              unsigned int         compare,
                              unsigned int         value );
unsigned long long atomicCAS ( unsigned long long * addr,
                              unsigned long long   compare,
                              unsigned long long   value );
```

2.5.2. Атомарные побитовые операции

Побитовые атомарные операции читают слово по заданному адресу, применяют к нему побитовую операцию с заданным параметром и записывают результат обратно. Во всех случаях результатом функций является исходное значение, находившееся по заданному адресу до начала операции.

```
         int atomicAnd (          int * addr,          int value );
unsigned int atomicAnd ( unsigned int * addr, unsigned int value );
         int atomicOr  (          int * addr,          int value );
unsigned int atomicOr  ( unsigned int * addr, unsigned int value );
         int atomicXor (          int * addr,          int value );
unsigned int atomicXor ( unsigned int * addr, unsigned int value );
```

2.5.3. Проверка статуса нитей варпа

Начиная с compute capability 1.2 поддерживаются две атомарные операции, выполняющие сравнение переданного значения с нулем для всех нитей. Результат показывает, получено ли ненулевое значение для всех нитей или хотя бы для одной нити варпа.

```
int __all ( int predicate );
int __any ( int predicate );
```

Для устройств с compute capability 2.x введена операция, возвращающая целое число, чей N-ый бит выставлен в 1 для каждой N-ой нити, получившей ненулевое значение.

```
unsigned int __ballot ( int predicate );
```

2.5.4. Доступность и производительность атомарных операций

Атомарные операции, как и любой другой метод синхронизации, всегда являются узким местом параллельной программы. Степень поддержки атомарных операций различными поколениями GPU неодинакова. В таблице 2.2 приведены наборы доступных атомарных операций в глобальной памяти в зависимости от compute capability.

Таблица 2.2. Атомарные операции в глобальной памяти

	1.0	**1.1**	**1.2**	**1.3**	**2.x**	**3.0**
Операции с целыми 32-битными словами	–	+	+	+	+	+
atomicExch() с 32-битными значениями с плавающей запятой	–	+	+	+	+	+
Операции с целыми 64-битными словами	–	–	+	+	+	+
atomicAdd() с 32-битными значениями с плавающей запятой	–	–	–	–	+	+

Атомарные операции также доступны и в разделяемой памяти. В таблице 2.3 показаны возможности использования атомарных операций в разделяемой памяти для GPU с различной compute capability.

Скорость выполнения атомарных операций напрямую влияет на производительность ядра. Поэтому при проектировании каждой новой архитектуры GPU ставится задача повысить их эффективность. В таблице 2.4 показаны различия в скорости работы атомарных операций чипов GF100 (архитектура Fermi) и GK104 (архитектура Kepler).

Таблица 2.3. Атомарные операции в разделяемой памяти

	1.0	**1.1**	**1.2**	**1.3**	**2.x**	**3.0**
Операции с целыми 32-битными словами	–	–	+	+	+	+
atomicExch() с 32-битными значениями с плавающей запятой	–	–	+	+	+	+
Операции с целыми 64-битными словами	–	–	–	–	+	+
atomicAdd() с 32-битными значениями с плавающей запятой	–	–	–	–	+	+
__all(), *__any()*	–	–	+	+	+	+
__ballot	–	–	–	–	+	+

Таблица 2.4. Производительность атомарных операций

	GF100	**GK104**	**Соотношение**	**Соотношение с учетом частоты ядер**
Атомарные операции для разделяемого адреса	1/9	1	9.0x	11.7x
Атомарные операции для независимого адреса	24	64	2.7x	3.5x

Глава 3

Иерархия памяти

GPU и CPU существенно отличаются организацией систем памяти и методами работы с ней. В CPU большую долю площади схемы обычно занимают кэши различных уровней. Основная же часть GPU занята вычислительными блоками. Кроме того, CPU обычно не предоставляет прямой доступ к управлению кэшами, ограничиваясь в основном лишь инструкцией *prefetch*. В GPU также присутствует не контролируемый явно кэш, однако существует еще несколько видов памяти, которые всегда должны управляться программно (таблица 3.1). Одни виды памяти расположены непосредственно в каждом из потоковых мультипроцессоров, другие – размещены в DRAM. В умелом использовании различных видов памяти состоит одновременно и сложность программирования для CUDA, и значительный потенциал эффективности.

Таблица 3.1. Типы памяти в CUDA

Тип памяти	**Расположение**	**Кэш**	**Доступ**	**Видимость**	**Время жизни**
Регистры	Мультипроц.	Нет	R/W	Нить	Нить
Локальная	DRAM GPU	Нет	R/W	Нить	Нить
Разделяемая	Мультипроц.	Нет	R/W	Блок	Блок
Глобальная	DRAM GPU	Нет	R/W		
Константная	DRAM GPU	Есть	R/O		
Текстурная	DRAM GPU	Есть	R/O		
Общее адресное пространство (UVA)	DRAM хоста	Нет	R/W	Везде	Динамическое

Наиболее простым видом памяти является *регистровый файл* (или просто *регистры*). Каждый потоковый мультипроцессор в GPU с compute capability 1.0, 1.1, 1.2, 1.3 и 2.x, содержит 8192, 16384 или 32768 32-битных регистров соответственно. Регистры распределяются между нитями блока на этапе компиляции, что влия-

ет на максимальное количество блоков, которые может выполнять один мультипроцессор. Каждая нить получает в монопольное использование для чтения и записи некоторое количество регистров на все время исполнения ядра. Доступ к регистрам других нитей запрещен. Поскольку регистры расположены непосредственно в потоковом мультипроцессоре, они обладают минимальной латентностью.

Если регистров не хватает, то для размещения локальных данных нити дополнительно используется *локальная память*. Так как она размещена в DRAM, то латентность доступа к ней велика: 400–800 тактов. Также в локальную память всегда попадают союзы (unions), а также большие или нетривиально индексируемые автоматические массивы.

Важным типом памяти в CUDA является *разделяемая память* (shared memory). Эта память расположена непосредственно в потоковом мультипроцессоре, но в отличие от регистров, выделяется не на уровне нитей, а на уровне блоков. Каждый блок получает в свое распоряжение одно и то же количество разделяемой памяти. Всего каждый мультипроцессор устройств с compute capability 2.x содержит 48 Кбайт разделяемой памяти. От размеров разделяемой памяти, требуемой каждому блоку, зависит максимальное количество блоков, которое может быть запущено на одном мультипроцессоре. Разделяемая память обладает очень небольшой латентностью, сравнимой со временем доступа к регистрам и может быть использована всеми нитями блока для чтения и записи.

Глобальная память – это обычная память DRAM, расположенная на плате GPU или разделяемая с памятью мобильного устройства. Начиная с архитектуры Fermi глобальная память кэшируется. Влияние присутствия кэша тем не менее не стоит переоценивать: в отличие от CPU, его размер в пересчете на одну нить очень мал. Глобальная память может выделяться как с CPU функциями CUDA API, так и нитями CUDA-ядра с помощью вызова *malloc*. Все нити ядра могут читать и писать в глобальную память. Как и локальная память, глобальная обладает высокой латентностью. Минимизация доступа к глобальной памяти – это основной метод получения высокоэффективных CUDA-приложений.

Константная и *текстурная память* также расположены в DRAM, но обладают независимым кэшем, обеспечивающим высокую скорость доступа. Оба типа памяти доступны всем нитям GPU, но только на чтение. Запись в них может произ-

водить CPU при помощи функций CUDA API. Общий объем константной памяти ограничен 64 Кбайтами. Текстурная память также предоставляет специальные возможности по работе с текстурами.

Следует обратить внимание на то, что любой CUDA-вызов, связанный с выделением и освобождением памяти на GPU, всегда является синхронным и поэтому будет выполнен только после завершения всех активных асинхронных вызовов.

3.1. Константная память

Константная память – кэширумая область DRAM размером 64 Кбайт, доступная с GPU только для чтения и для чтения и записи с хоста при помощи следующих функций:

```
cudaError_t cudaMemcpyToSymbol(
    const char * symbol, const void * src,
    size_t count, size_t offset, enum cudaMemcpyKind kind);

cudaError_t cudaMemcpyFromSymbol(
    void * dst, const char * symbol,
    size_t count, size_t offset, enum cudaMemcpyKind kind);

cudaError_t cudaMemcpyToSymbolAsync(
    const char * symbol, const void * src,
    size_t count, size_t offset, enum cudaMemcpyKind kind,
    cudaStream_t stream);

cudaError_t cudaMemcpyFromSymbolAsync(
    void * dst, const char * symbol,
    size_t count, size_t offset, enum cudaMemcpyKind kind,
    cudaStream_t stream);
```

Параметр *kind* задает направление операции копирования и может принимать одно из следующих значений: *cudaMemcpyHostToHost*, *cudaMemcpyHostToDevice*, *cudaMemcpyDeviceToHost* и *cudaMemcpyDeviceToDevice*. Параметр *stream* позволяет организовать несколько асинхронных потоков команд. По умолчанию используется синхронный режим, соответствующий нулевому потоку.

Константная память может быть выделена только путем статического объявления в коде программы с добавлением атрибута *__constant__*. В следующем фрагменте кода массив в константной памяти заполняется данными из DRAM хоста:

```
__constant__ float contsData [256]; // константная память GPU
float              hostData  [256]; // данные в памяти CPU
...
// Скопировать данные из памяти CPU в константную память GPU
cudaMemcpyToSymbol(constData, hostData,
    sizeof(data), 0, cudaMemcpyHostToDevice);
```

Поскольку константная память кэшируется, то она подходит для размещения небольшого объема часто используемых неизменяемых данных, которые должны быть доступны всем нитям. Дополнительно в устройствах с compute capability 2.x доступно аналогичное константной памяти кэширование произвольного участка глобальной памяти (инструкция LDU или LoaD Uniform). Данный режим будет автоматически активирован при запросе только для чтения по адресу памяти, не зависящему от индекса нити.

```
__global__ void kernel( float *g_dst, const float *g_src )
{
    g_dst = g_src[0];            // не зависит от индекса нити -> uniform load
    g_dst = g_src[blockIdx.x];   // не зависит от индекса нити -> uniform load
    g_dst = g_src[threadIdx.x];  // зависит от индекса -> non-uniform
}
```

3.2. Глобальная память

Основную часть DRAM GPU занимает глобальная память. При корректной работе ядер динамически выделенная глобальная память сохраняет целостность данных на протяжении всего времени жизни приложения, что, в частности, позволяет использовать ее как основное хранилище для передачи данных между несколькими ядрами.

Глобальная память может быть выделена как статически, так и динамически. Динамическое выделение и освобождение глобальной памяти в коде хоста производится при помощи следующих вызовов:

```
cudaError_t cudaMalloc(void ** devPtr, size_t size);

cudaError_t cudaFree(void * devPtr);

cudaError_t cudaMallocPitch(void ** devPtr, size_t * pitch,
    size_t width, size_t height);
```

С появлением device-функций *malloc* и *free* в CUDA 3.2, на GPU с compute capability 2.0 и выше динамическое выделение и освобождение глобальной памяти возможно не только в хост-коде но и в коде CUDA-ядра. Однако при этом динамическое выделение происходит лишь относительно нитей ядра, тогда как общий пул памяти под эти аллокации выделяется заранее. Размер пула может быть установлен вызовом функции *cudaLimitMallocHeapSize* или по умолчанию равен 8 Мб. В следующем примере два CUDA-ядра выделяют и освобождают глобальную память GPU:

```
#include <stdio.h>
#include <stdlib.h>

__global__ void mass_malloc(void** ptrs, size_t size)
{
  ptrs[threadIdx.x] = malloc(size);
}

__global__ void mass_free(void** ptrs)
{
  free(ptrs[threadIdx.x]);
}

int main(int argc, char* argv[])
{
  if (argc != 3)
  {
    printf("Usage: %s <nthreads> <size>\n", argv[0]);
    return 0;
  }

  int nthreads = atoi(argv[1]);
  size_t size = atoi(argv[2]);

  // Установить размер пула. Это действие должно быть
  // выполнено до запуска каких-либо ядер.
  cudaDeviceSetLimit(cudaLimitMallocHeapSize, 16 * 1024 * 1024);

  void** ptrs; cudaMallocHost(&ptrs, sizeof(void*) * nthreads);
  memset(ptrs, 0, sizeof(void*) * nthreads);
  mass_malloc<<<1, nthreads>>>(ptrs, size);
  cudaDeviceSynchronize();
  for (int i = 0; i < nthreads; i++)
    printf("Thread %d got pointer: %p\n", i, ptrs[i]);
```

```
  mass_free<<<1, nthreads>>>(ptrs);
  cudaDeviceSynchronize();
  cudaFreeHost(ptrs);

  return 0;
}
```

```
$ ./device_malloc_test 8 24
Thread 0 got pointer: 0x2013ffb20
Thread 1 got pointer: 0x2013ffb70
Thread 2 got pointer: 0x2013ffbc0
Thread 3 got pointer: 0x2013ffc10
Thread 4 got pointer: 0x2013ffc60
Thread 5 got pointer: 0x2013ffcb0
Thread 6 got pointer: 0x2013ffd00
Thread 7 got pointer: 0x2013ffd50
```

Глобальная переменная может быть объявлена статически при помощи атрибута *__device__*. В этом случае память будет выделена при инициализации модуля с CUDA-ядрами и данными (в зависимости от способа загрузки, в момент старта приложения или при вызове *cuModuleLoad*). В следующем примере утилита *nm* показывает, что в объектном представлении *cubin* (двоичный файл CUDA-модуля), глобальные данные размещаются таким же образом как на CPU: “B” – неинициализированная переменная, “D” – инициализированная переменная, “R” – только для чтения. Отличие же состоит в том, что в cubin статические переменные не становятся приватными (соответствующие литеры в нижнем регистре) и не декорируются. Правила видимости не имеют смысла, поскольку в CUDA отсутствует линковка, и все CUDA-объекты должны быть самодостаточными, без внешних зависимостей.

```
__device__ float val1;
__device__ float val2 = 1.0f;
__device__ const float val3 = 2.0f;
__constant__ float val_pi = 3.14;
__device__ static float val4;

__global__ void kernel(float* val5)
{
    if (!threadIdx.x && !blockIdx.x)
    {
        val4 = sin(val1 + val2 * val3 + val_pi);
        *val5 = val4;
```

```
    }
}

$ nvcc –keep –c test.cu
$ nm test.sm_10.cubin | grep val
0000000000000004 B val1
0000000000000004 D val2
0000000000000000 D val3
0000000000000000 B val4
0000000000000018 R val_pi


float val1;
float val2 = 1.0f;
const float val3 = 2.0f;
float val_pi = 3.14;
static float val4;

#include <cmath>

void kernel(float* val5)
{
    val4 = sin(val1 + val2 * val3 + val_pi);
    *val5 = val4;
}

$ g++ –c test.c
$ nm test.o | grep val
0000000000000000 r _ZL4val3
0000000000000004 b _ZL4val4
0000000000000000 B val1
0000000000000000 D val2
0000000000000004 D val_pi
```

Доступ к многомерным массивам в глобальной памяти может быть более эффективен при наличии выравнивания строк. Выравнивание обеспечивается добавлением фиктивных элементов в конец каждой строки и соответствующих сдвигов при индексации. Функция *cudaMallocPitch* выделяет память с выравниванием строк и возвращает сдвиг через параметр *pitch*. Например, если выделена память для матрицы с элементами типа *T*, то для получения адреса элемента, расположенного в строке *row* и столбце *col*, используется следующая формула:

```
T * item = (T *) ((char *) baseAddress + row * pitch) + col;
```

Хотя функция *cudaMalloc* возвращает обычный указатель, его значение имеет смысл только для адресного пространства GPU. Для заполнения памяти GPU данными хоста и наоборот, необходимо использовать специальные функции копирования:

```
cudaError_t cudaMemcpy (
    void * dst, const void * src, size_t size,
    enum cudaMemcpyKind kind );

cudaError_t cudaMemcpyAsync (
    void * dst, const void * src, size_t size,
    enum cudaMemcpyKind kind, cudaStream_t stream );

cudaError_t cudaMemcpy2D (
    void * dst, size_t dpitch, const void * src, size_t spitch,
    size_t width, size_t height, enum cudaMemcpyKind kind );

cudaError_t cudaMemcpy2DAsync (
    void * dst, size_t dpitch, const void * src, size_t spitch,
    size_t width, size_t height, enum cudaMemcpyKind kind,
    cudaStream_t stream );
```

Копирование данных в глобальной памяти внутри одного GPU обычно производится на порядок быстрее, чем внутри памяти хоста, например, для GPU Tesla C2050 характерная скорость – 144 Гбайт/сек. Копирование данных между памятью хоста и памятью GPU значительно медленнее. Наиболее распространенный в данный момент стандарт интерфейса PCI Express 2.0 способен обеспечить пропускную способность до 8 Гбайт/сек. С учетом потерь на кодирование, латентности и задержки, на практике, как правило, удается добиться не более 4 Гбайт/сек при копировании между хостом и одним GPU и не более 6 Гбайт/сек – при копировании между хостом и несколькими GPU одновременно.

Наилучшая скорость копирования данных между хостом и GPU может быть достигнута при использовании *pinned-памяти*. Pinned-память – это память *хоста*, которая может быть либо выделена функциями *cudaHostAlloc* или *cudaMallocHost*, заменяющими стандартный вызовы *malloc* или *new*, либо получена переводом обычной памяти в категорию pinned функцией *cudaHostRegister* (см. Unified Virtual Address Space).

```
cudaError_t cudaHostAlloc (void** pHost, size_t size, unsigned int flags);
cudaError_t cudaMallocHost(void** devPtr, size_t size);
```

```
cudaError_t cudaFreeHost(void* devPtr);

cudaError_t cudaHostRegister(void* ptr, size_t size, unsigned int flags);
cudaError_t cudaHostUnregister(void* ptr);
```

При асинхронном копировании памяти между CPU и GPU с помощью функции *cudaMemcpyAsync* в CUDA версии 4.0 должна использоваться только pinned-память. Начиная с CUDA версии 4.0 может использоваться обычная (не pinned) память, но в этом случае вызов *cudaMemcpyAsync* будет синхронным. Выделение большого количества pinned-памяти может отрицательно сказаться на быстродействии всей системы.

В CUDA используется *прямая адресация* глобальной памяти GPU, т.е., в отличие от OpenCL, для хранения адресов подходят обычные указатели, и для них корректна адресная арифметика. Если память выделяется с помощью *cudaMalloc*, то выделенный диапазон имеет смысл только в контексте GPU-ядра. Память, выделенная на одном GPU некорректна по отношению к другому GPU (см. контексты GPU). Если память выделяется *cudaHostAlloc(..., , cudaHostAllocMapped)*, то выделенный на CPU диапазон памяти становится доступен как для CPU, так и для всех GPU (см. Unified Virtual Address Space).

3.2.1. Кэширование

Для более эффективного использования глобальной памяти в GPU архитектуры Fermi (compute capability 2.0 и 2.1) реализованы кэши первого и второго уровня (Рис. 3.1).

Кэш первого уровня (L1) находится на каждом мультипроцессоре, тогда как кэш второго уровня (L2) – общий и имеет размер 768 Кбайт. Кэш L1 и разделяемая память расположены на одном физическом носителе, объем которого может быть разделен между ними одним из двух способов: 48 Кбайт разделяемой памяти и 16 Кбайт L1-кэша или 16 Кбайт разделяемой памяти и 48 Кбайт L1-кэша. Переключение производится функцией *cudaFuncSetCacheConfig*:

```
// Device code
__global__ void MyKernel()
{
    ...
}
```

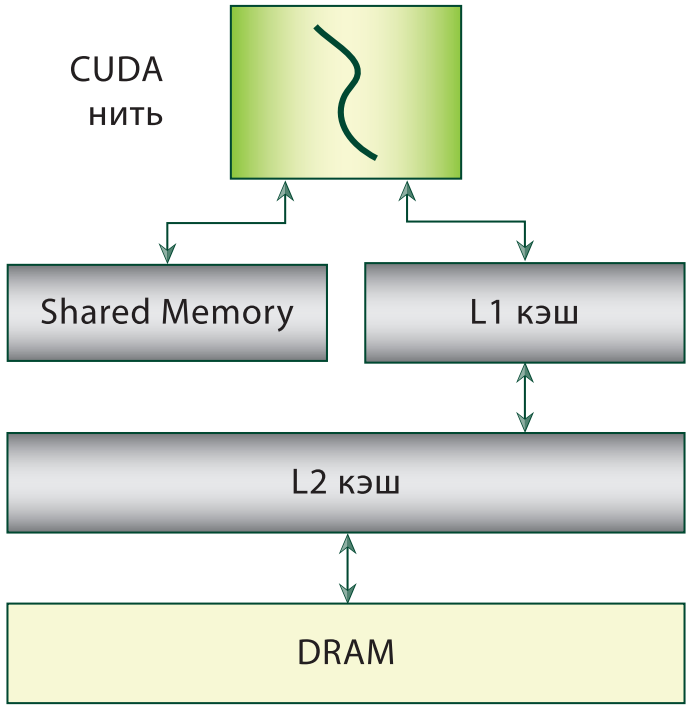


Рис. 3.1. Fermi – подсистема памяти

```
// Host code
// cudaFuncCachePreferShared: 48 KB разделяемой памяти
// cudaFuncCachePreferL1: 16 KB разделяемой памяти
// cudaFuncCachePreferNone: без предпочтения, использовать текущий контекст
cudaFuncSetCacheConfig(MyKernel, cudaFuncCachePreferShared);
```

По умолчанию используется конфигурация «без предпочтения». В этом режиме будет использоваться конфигурация текущего контекста, который может быть задан с помощью *cudaDeviceSetCacheConfig*. Если текущий контекст также не имеет предпочтения, то будет использована конфигурация предыдущего запуска любого ядра, если нет конфликта по требованиям к кэшу и разделяемой памяти. По умолчанию изначально используется конфигурация в пользу большего объема разделяемой памяти.

Длина кэш-линии составляет 128 байт и соответствует выровненному по 128 байтам сегменту глобальной памяти. Все транзакции с памятью, проходящие через L1- и L2-кэш, имеют размер 128 байт. Использование L1-кэша можно отключить на этапе компиляции с помощью опций *-Xptxas -dlcm=cg*. При отключенном L1-кэше размер транзакции уменьшается до 32 байт. Этот режим может быть более эффективен в случае, если необходимые данные расположены в коротких разрывных участках памяти.

Если размер слова для каждой нити равен 4 байтам, то запросы в память всех 32 нитей варпа объединяются в один. Если размер слова для каждой нити превышает 4 байта, обращение варпа к памяти делится на независимые запросы по 128 байт: два запроса при размере слова 8 байт и четыре при размере слова 16 байт. Каждому запросу, в свою очередь, соответствуют собственные линии в L1- и L2-кэше.

3.2.2. Пример: транспонирование матрицы

Пусть необходимо транспонировать квадратную матрицу $A : N \times N$ (здесь и далее мы будем считать, что N кратно 16). Поскольку матрица – это двумерный массив, то будет удобно использовать двумерную сетку и двухмерные блоки. Размер блока выберем равным 16×16, что позволит запустить до 3 блоков на одном мультипроцессоре архитектуры 1.x и до 6 – на архитектуре Fermi. Тогда для транспонирования матрицы можно использовать следующее ядро:

```
__global__ void transpose ( float * inData, float * outData, int n )
{
    unsigned int xIndex   = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int yIndex   = blockDim.y * blockIdx.y + threadIdx.y;
    unsigned int inIndex  = xIndex + n * yIndex;
    unsigned int outIndex = yIndex + n * xIndex;

    outData [outIndex] = inData [inIndex];
}
```

3.2.3. Пример: перемножение двух матриц

Пусть необходимо перемножить две квадратные матрицы $A, B : N \times N$. Как и в предыдущем примере, будем использовать двумерные блоки 16×16 и двумерную сетку. Ниже приведено ядро, в точности реализующее общую формулу перемножения матриц:

```
__global__ void matmul1 (float* a, float* b, int n, float* c)
{
    // Индексы блока
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Индексы нити внутри блока
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Переменная для накопления результата
    float sum = 0.0f;

    // Смещение для a [i][0]
    int ia = n * BLOCK_SIZE * by + n * ty;

    // Смещение для b [0][j]
    int ib  = BLOCK_SIZE * bx + tx;

    // Перемножить строку и столбец
    for (int k = 0; k < n; k++)
        sum += a [ia + k] * b [ib + k * n];

    // Смещение для записываемого элемента
    int ic = n * BLOCK_SIZE * by + BLOCK_SIZE * bx;

    // Сохранить результат в глобальной памяти
```

```
    c[ic + n * ty + tx] = sum;
}
#define kernel matmul1
#include "main.h"
```

В данном случае при вычислении одного элемента произведения двух матриц на $2N-1$ арифметических операций приходится $2N$ чтений из глобальной памяти. При таком соотношении производительность GPU-ядра ограничена скоростью работы глобальной памяти (в англоязычной литературе – *memory bound*). Значительно большую эффективность может иметь блочный алгоритм перемножения матриц с применением разделяемой памяти (см. раздел о разделяемой памяти).

3.2.4. Оптимизация работы с глобальной памятью

Глобальная память обладает высокой латентностью, поэтому для достижения высокой эффективности приложений доступ к ней необходимо оптимизировать.

Выравнивание. При чтении и записи значений глобальной памяти на низком уровне используются выровненные 32-, 64- и 128-битные слова. По этой причине все функции выделения глобальной памяти CUDA API всегда возвращают адреса, выровненные по 256-байтам. Если при кратном выравниванию размере элементов, базовый адрес массива по какой-либо причине оказался невыровненным, то чтение каждого элемента вместо одного $[0..3]$ (рис. 3.2, верхняя строка) потребует двух обращений – $[0..3]$ и $[4..7]$, полностью покрывающих невыровненный запрос (рис. 3.2, нижняя строка).

Аналогичная ситуация возникает при использовании элементов массива, размер которых не кратен выравниванию:

```
struct vec3
{
    float x, y, z;
};
```

В этом случае при выровненном базовом адресе и длине элемента в 12 байт, выровненым будет только каждый четвертый элемент, а все остальные потребуют по два обращения. Проблема может быть решена добавлением фиктивного элемента или директивы выравнивания по 16 байтам:

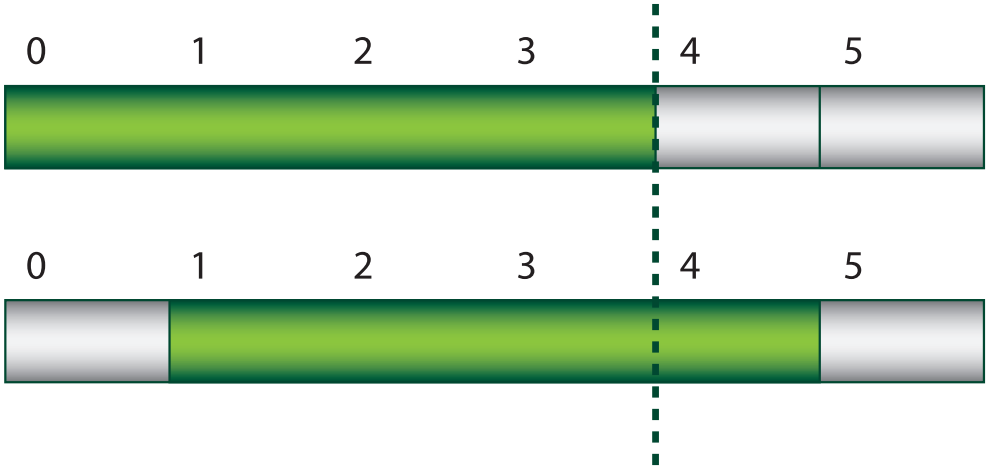


Рис. 3.2. Пример выровненного (сверху) и невыровненного (внизу) 4-байтового блока

```
struct __attribute__((aligned(16))) vec3
{
    float x, y, z;
};
```

Теперь все элементы массива будут располагаться по адресам, кратным 16, что обеспечит чтение одного элемента за раз. Таким образом, эффективность доступа может быть повышена ценой увеличения расхода памяти.

Объединение запросов. Оптимизации, позволяющие объединять запросы всех нитей полуварпа (compute capability 1.x) или всего варпа в одно обращение к непрерывному блоку глобальной памяти (coalescing), могут на порядок ускорить работу GPU-приложения.

Для того, чтобы GPU с compute capability 1.0 или 1.1 произвел объединение запросов нитей половины варпа, необходимо выполнение следующих условий:

- Все нити должны обращаться к 32-битным словам, давая в результате один 64-байтовый блок, или к 64-битным словам, давая один 128-байтовый блок;
- Полученный блок должен быть выровнен по своему размеру, т.е. адрес 64-байтового блока кратен 64, а адрес 128-байтовго блока кратен 128;

- Все 16 слов, к которым обращаются нити, должны находиться внутри этого блока;

- Нити должны обращаться к словам последовательно: k-ая нить должна обращаться к k-му слову (при этом допускается, что отдельные нити пропустят обращение к соответствующим словам).

Если нити полуварпа не удовлетворяют какому-либо из данных условий, то каждое обращение к памяти происходит как отдельная транзакция. На рис. 3.3 приведены типичные шаблоны обращения к памяти, приводящие к объединению запросов в одну транзакцию: слева выполнены все условия, а справа для части нитей пропущено обращение к соответствующим словам, что позволяет добавить фиктивные обращения и свести к случаю слева. На рис. 3.4 слева для нитей 4 и 5 нарушен порядок обращения к словам, а справа нарушено условие выравнивания: несмотря на то, что слова образуют непрерывный блок из 64 байт, начало этого блока (по адресу 132) не кратно его размеру.

На GPU с compute capability 1.2 и 1.3 объединение запросов в один будет происходить, если слова, к которым обращаются нити, лежат в одном сегменте размером 32 байта (если все нити обращаются к 8-битным словам), 64 байта (если все нити обращаются к 16-битным словам) и 128 байт (если все нити обращаются к 32-битным или 64-битным словам). Объединенный сегмент (блок) должен быть выровнен по 32, 64 или 128 байтам соответственно. В случае compute capability 1.2 и 1.3 порядок, в котором нити обращаются к словам, не играет никакой роли и ситуация на рис. 3.4 слева приведет к объединению всех запросов в одну транзакцию, а для случая справа произойдет объединение запросов в две транзакции.

В устройствах архитектуры Fermi объединение запросов в память происходит для всех 32 нитей (рис. 3.5). В верхней части рисунка приведен пример использования L1-кэша (размер транзакции – 128 байт). В данном случае нарушение выравнивания может и не привести к дополнительной транзакции при успешном попадании остатка в L1-кэш. На нижней части рисунка приведен пример с отключенным L1-кэшем (размер транзакции – 32 байта). Такой режим доступа выгоднее использовать в случае обращения нитей варпа в разрозненные участки глобальной памяти или, наоборот, при обращении всех нитей варпа к одному и тому же элементу.

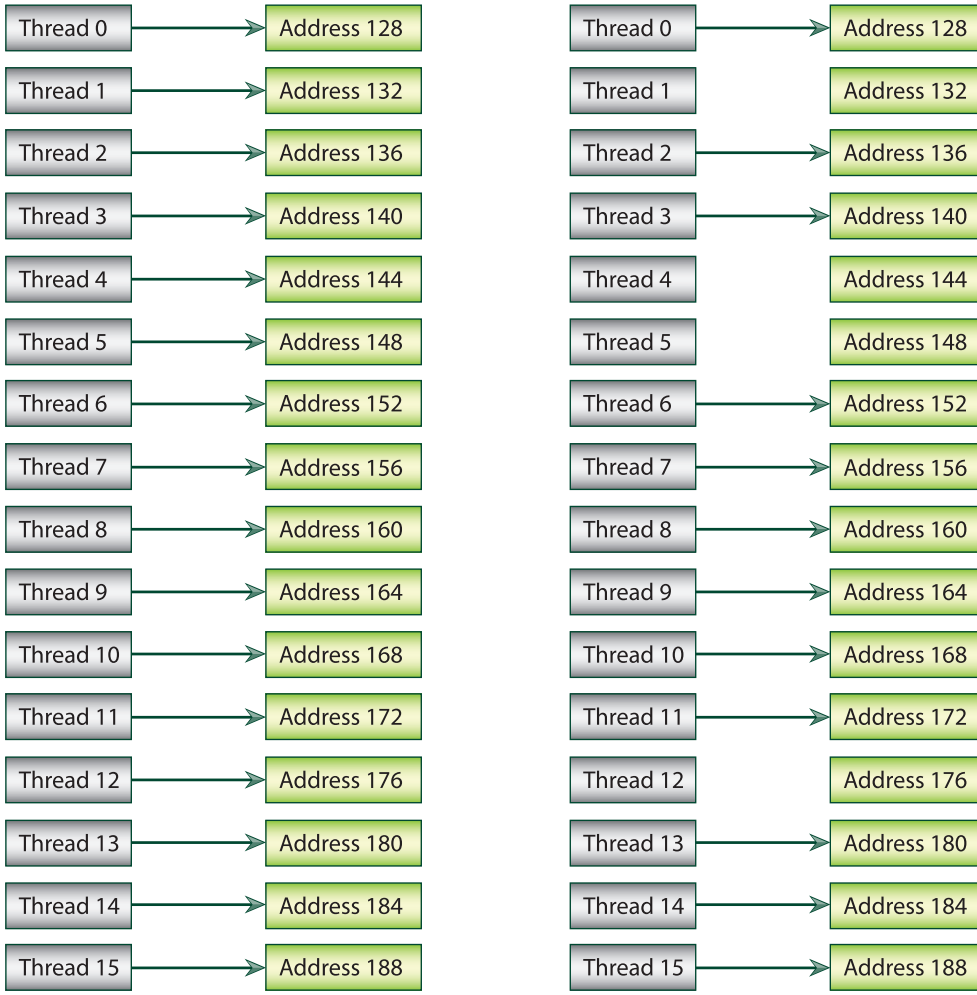


Рис. 3.3. Паттерны обращения к памяти, дающие объединение для GPU с compute capability 1.0 и 1.1

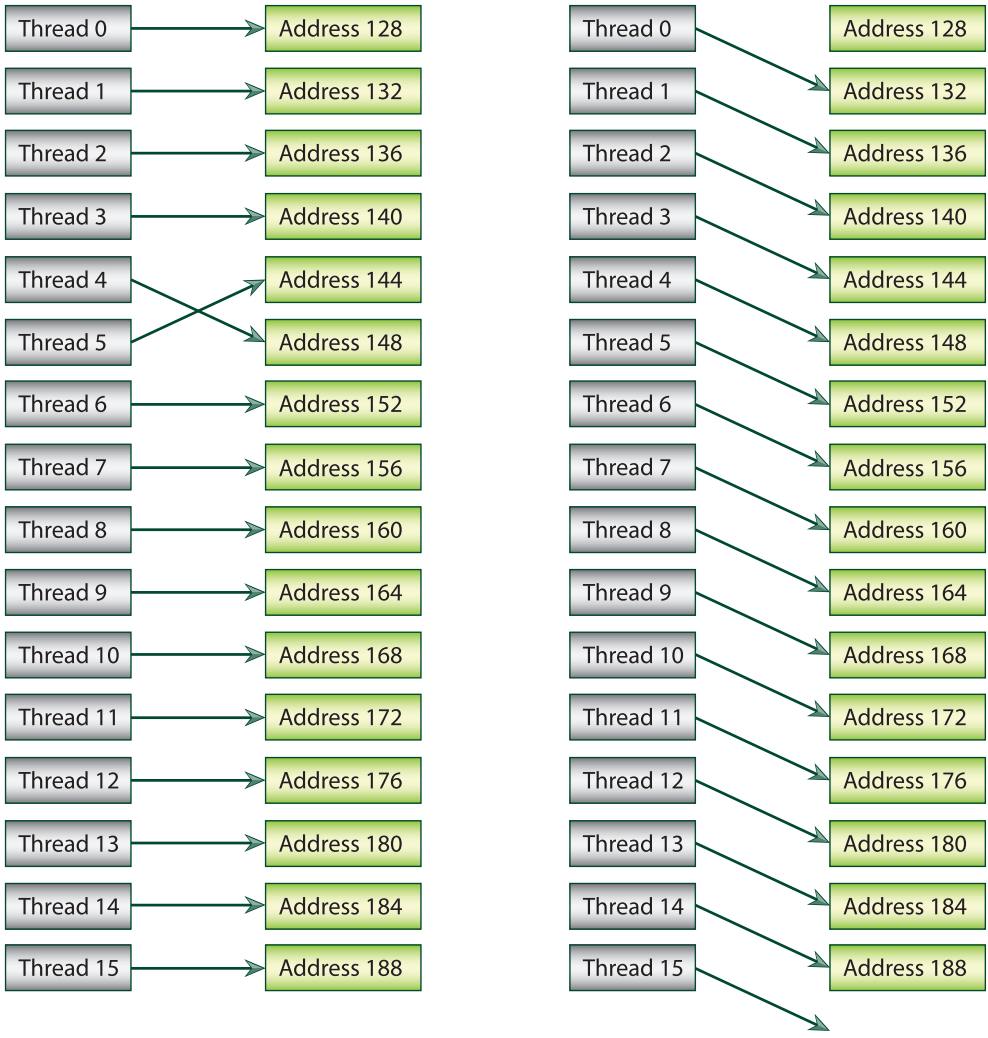


Рис. 3.4. Паттерны обращения к памяти, не дающие объединение для GPU с compute capability 1.0 и 1.1

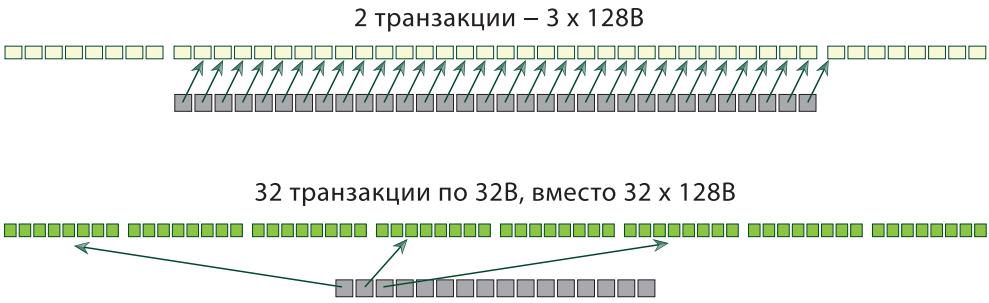


Рис. 3.5. Паттерны обращения к памяти на архитектуре Fermi

Объединения запросов может работать более эффективно со структурами массивов, чем с массивами структур. Например, при использовании структуры A объединение происходить не будет, и для доступа к каждому элементу массива $array$ потребуется отдельная транзакция:

```
struct A __attribute__((aligned(16)))
{
    float a;
    float b;
    uint  c;
};

A array [1024];

. . .

A a = array [threadIdx.x];
```

Напротив, если использовать массивы, в которых компоненты структуры выровнены и лежат друг за другом, то запросы всех нитей варпа или полуварпа будут объединены в 3 транзакции (вместо 32 транзакций ранее):

```
float a [1024];
float b [1024];
uint  c [1024];
. . .
float fa = a [threadIdx.x];
float fb = b [threadIdx.x];
uint  uc = c [threadIdx.x];
```

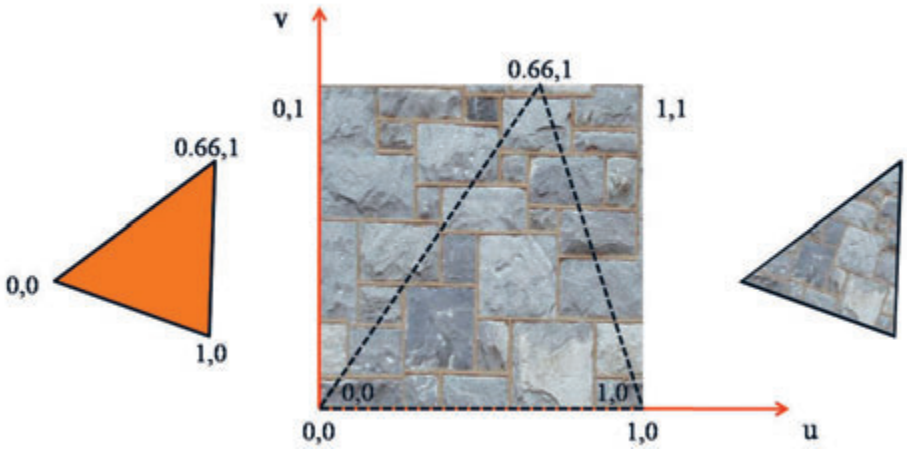


Рис. 3.6. Текстура

3.3. Текстурная память

Текстурная память и аппаратные схемы интерполяции объединены на GPU в *текстурные блоки*, которые используются в графических задачах для заполнения треугольников двумерными изображениями – *текстурами* (рис. 3.6).

В зависимости от архитектуры, каждому мультипроцессору может быть доступно различное количество текстурных блоков (см. секцию 6.1 и рис. 6.3). В текстурном блоке аппаратно реализованы следующие функции:

- фильтрация текстурных координат;
- билинейная или точечная интерполяция;
- разумное возвращаемое значение в случае, когда значения текстурных координат выходят за допустимые границы;
- обращение по нормализованным или целочисленным координатам;
- возвращение нормализованных значений;
- кэширование данных.

В общем случае текстура представляет собой простой и удобный интерфейс для работы с одномерными, двумерными и трехмерными массивами в режиме «только для чтения». Текстуры могут быть полезны на GPU с compute capability 1.x, если обеспечить объединение запросов в память не представляется возможным. С появлением L1- и L2-кэшей в устройствах архитектуры Fermi роль текстурной памяти снижается, поскольку она обладает меньшей скоростью, чем L1-кэш.

Текстурная память выделяется с помощью функции *cudaMallocArray*:

```
cudaError_t cudaMallocArray(struct cudaArray **arrayPtr,
                            const struct cudaChannelFormatDesc *desc,
                            size_t width, size_t height);

cudaError_t cudaFreeArray(struct cudaArray *array);
```

Однако в отличие от методов работы с глобальной памятью, *arrayPtr* является непрозрачным контейнером, управление которым осуществляет драйвер. Доступ к контейнеру из CUDA-ядра возможен только через специальные *текстурные ссылки* (texture reference), которые в графических API назывались *сэмплерами* (sampler). Задача такой абстракции – отделить данные (*cudaArray*) и способ их хранения от интерфейса доступа к ним (texture reference). Для чтения *cudaArray* из CUDA-ядра необходимо сначала ассоциировать его с текстурной ссылкой:

```
cudaError_t
cudaBindTextureToArray (const struct textureReference *texref,
                        const struct cudaArray *array,
                        const struct cudaChannelFormatDesc *desc);

cudaError_t
cudaBindTextureToArray(const struct texture<T, dim, readMode> & tex,
                       const struct cudaArray * array);
```

Первый вид вызова является низкоуровневым и требует задания переменной *textureReference* и дескриптора канала вручную:

```
textureReference texref;
texref.addressMode[0] = cudaAddressModeWrap;
texref.addressMode[1] = cudaAddressModeWrap;
texref.addressMode[2] = cudaAddressModeWrap;
texref.channelDesc = cudaCreateChannelDesc<uchar4>();
texref.filterMode = cudaFilterModeLinear;
texref.normalized = cudaReadModeElementType;
```

```
cudaChannelFormatDesc desc = cudaCreateChannelDesc<uchar4>();
```

Второй тип вызова использует шаблоны для задания текстурных ссылок и наследует дескриптор канала от переданного *cudaArray*:

```
texture<uchar4, 2, cudaReadModeElementType> texName;
```

Чтение текстуры из ядра производится функциями *tex1D()*, *tex2D()*, *tex3D()*, которые принимают текстурную ссылку и одну, две или три координаты. В случае, если выбран ненормализованный режим обращения к текстуре, стоит учитывать, что для точного попадания в центр пиксела, необходимо добавлять смещение, равное половине пиксела (т.е. равное $0.5f$ или $0.5f$ / ширину, $0.5f$ / высоту при нормализованных координатах):

```
uchar4 a = tex2D(texName, texcoord.x + 0.5f, texcoord.y + 0.5f);
```

Текстурные ссылки можно получить, зная имя текстурного шаблона, с помощью следующей функции:

```
const textureReference* pTexRef = NULL;
cudaGetTextureReference(&pTexRef, "texName");
```

Кроме cudaArray, привязать к текстурной ссылке можно и обычную линейную память.

```
cudaError_t
cudaBindTexture (size_t * offset,
                 const struct texture<T, dim, readMode> & tex,
                 const void * dev_ptr,
                 size_t size);

cudaError_t
cudaBindTexture2D(size_t * offset,
                  const struct texture<T, dim, readMode> & tex,
                  const void * dev_ptr,
                  const struct cudaChannelFormatDesc *desc,
                  size_t width, size_t height,
                  size_t pitch_in_bytes);
```

Основным полезным свойством текстуры является возможность кэширования данных в двумерном измерении.

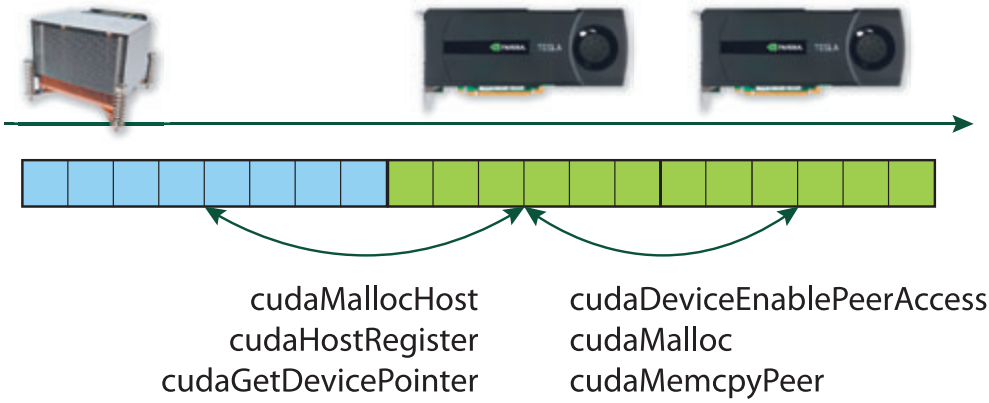


Рис. 3.7. Общее виртуальное адресное пространство (UVA)

3.4. Общее виртуальное адресное пространство (UVA)

Начиная с CUDA 4.0, диапазоны адресов глобальной памяти всех GPU и памяти хоста, выделенной с помощью *cudaHostAlloc*, могут не пересекаться между собой и образовывать *общее виртуальное адресное пространство* (Unified virtual address space, UVA), рис. 3.7. На этой технологии основаны две новые возможности:

- использование в GPU-ядрах pinned-памяти, без необходимости явно копировать ее вызовом *cudaMemcpy*;
- копирование данных между двумя GPU *напрямую* (peer-to-peer), без копирования в хост-буфер.

Загрузка данных pinned-памяти из GPU-ядра, в отличие от обычного «активного» копирования, производится без участия CPU, что позволяет более эффективно использовать вычислительные ресурсы CPU параллельно с GPU.

С работой UVA связаны следующие вызовы:

- *cudaSetDeviceFlags(cudaDeviceMapHost)* – включение режима поддержки использования pinned-памяти хоста из GPU-ядер;
- *cudaHostAlloc(..., cudaHostAllocMapped)* – выделение pinned-памяти на хосте, которая напрямую доступна всем GPU, если ранее был сделан вызов

cudaSetDeviceFlags(cudaDeviceMapHost);

- *cudaHostRegister* – преобразование памяти, выделенной обычным образом (например, *malloc*, *new* или *allocate*) в pinned-память;
- *cudaHostGetDevicePointer* – получение UVA-совместимого адреса памяти хоста, в случае если эта память была не выделена функцией *cudaHostAlloc(..., cudaHostAllocMapped)*, а превращена в pinned из обычной с помощью вызова *cudaHostRegister*;
- *cudaDeviceCanAccessPeer* – проверка, включен ли режим поддержки копирования peer-to-peer;
- *cudaDeviceEnablePeerAccess* – включение режима поддержки использования на GPU данных из памяти другого GPU;
- *cudaMemcpyPeer* – копирование данных напрямую между двумя GPU.

3.4.1. Пример: использование pinned-памяти хоста в GPU-ядре

В следующем примере показаны различные ситуации использования pinned-памяти из GPU. На устройствах с compute capability 1.1 или 1.2 никакой адрес pinned-памяти не может быть использован непосредственно из-за малой ширины диапазона адресов. Тем не менее, pinned-память хоста будет доступна, если с помощью вызова *cudaHostGetDevicePointer* преобразовать исходный адрес в совместимый с GPU. Для устройств с compute capability 2.0 и выше в преобразовании нет необходимости, если pinned-память выделена с помощью *cudaHostAlloc*. А если память выделена обычным образом и преобразована в pinned-память функцией *cudaHostRegister*, то *cudaHostGetDevicePointer* нужен в любом случае:

```
...

// Включить режим поддержки использования pinned-памяти
// хоста из GPU-ядер
status = cudaSetDeviceFlags(cudaDeviceMapHost);

// Выделить pinned-память
float* din;
status = cudaHostAlloc((void**)&din, size, cudaHostAllocMapped);
```

```
// Выделить обычную память и перевести ее в pinned.
float* dout = (float*)malloc(size);
status = cudaHostRegister(dout, size, cudaHostRegisterMapped);

// Для устройств с compute capability < 2.0
// преобразовывать адрес pinned-памяти в отображаемый.
float* mdin = din;
struct cudaDeviceProp props;
status = cudaGetDeviceProperties(&props, 0);
int use_mapping = props.major < 2;
if (use_mapping)
{
    status = cudaHostGetDevicePointer((void**)&mdin, din, 0);
    if (status != cudaSuccess)
    {
        fprintf(stderr, "Cannot map input buffer to device memory: %s\n",
            cudaGetErrorString(status));
        return 1;
    }
}

// Адрес памяти, переведенной в pinned с помощью
// cudaHostRegister, всегда преобразовывается.
float* mdout = NULL;
status = cudaHostGetDevicePointer((void**)&mdout, dout, 0);

// Чтение и запись в din и dout с хоста.
// Host-side reading and writing to din and dout
double dinvrmax = 1.0 / RAND_MAX;
for (int i = 0; i < n * n; i++)
    din[i] = rand() * dinvrmax;
memset(dout, 0, size);

// Чтение и запись в din и dout в GPU-ядре!
// Device-side reading and writing to din and dout!
pattern2d_gpu(1, n, 1, 1, n, 1, mdin, mdout);

...
```

Память на хосте, выделенная с помощью *cudaHostAlloc* или *cudaMallocHost*, всегда выровнена, тогда как другие методы аллокации могут возвращать произвольные адреса. В CUDA 4.0 адрес памяти и размер буфера в параметрах вызова *cudaHostRegister* должны быть выровнены по границе страницы памяти (4 Кбай-

та). В CUDA 4.1 это ограничение снято. Диапазоны адресов, для которых выполнен вызов *cudaHostRegister* не должны иметь пересечений.

3.4.2. Пример: обмен данными напрямую между GPU

На рабочей станции с несколькими GPU (compute capability 2.0 и выше), подключенными к одному хосту, поддерживается прямое копирование данных с GPU на GPU по PCI-E шине, без использования памяти хоста. Для этого необходимо активировать режим “peer access” на соответствующих устройствах с помощью вызовов *cudaDeviceCanAccessPeer* и *cudaDeviceEnablePeerAccess*:

```
float *devicePtr, *peerDevicePtr;
...
int canAccessPeer;
int deviceIdx = 0;
int peerDeviceIdx = 1;

// задать текущее устройство
cudaSetDevice(deviceIdx);

// проверить, возможен ли P2P доступ из устройства 0 к устройству 1
cudaDeviceCanAccessPeer(&canAccessPeer, deviceIdx, peerDeviceIdx);
if (canAccessPeer)
{
    // скопировать 1024 элемента типа float из области памяти devicePtr на
        устройстве 0
    // в область памяти peerDevicePtr устройства 1
    cudaMemcpy(peerDevicePtr, devicePtr,
        sizeof(float) * 1024, cudaMemcpyDefault);
}

if (canAccessPeer)
{
    // Включить peer-to-peer доступ.
    cudaDeviceEnablePeerAccess(peerDeviceIdx, 0);

    // Ядро запускается на устройстве 0,
    // но использует память на устройтсве 1 через PCI-E
    kernel<<<…>>>(peerDevicePtr);
}
```

На ОС Windows общее адресное пространство отключено при выборе драйвера WDDM и включено с Tesla Compute Cluster Driver for Windows (TCC). Переключе-

ние режима производит утилита *nvidia-smi* с флагом *-dm* или *–driver-model=*. При отключенном UVA peer-to-peer копирование возможно только с помощью вызова *cudaMemcpyPeer*, в котором явно указываются индексы устройств.

Наличие общего адресного пространства делает избыточным параметр *kind* вызова *cudaMemcpy*. Поэтому он может быть задан *cudaMemcpyDefault* для любых операций peer-to-peer доступа. При отсутствии поддержки peer-to-peer, выключенном peer access или физическом отсутствии прямой связи между GPU на системах с несколькими PCI-E шинами вызов *cudaMemcpy* (или *cudaMemcpyPeer*) произведет копирование данных через оперативную память управляющего устройства.

3.5. Разделяемая память

Разделяемая память размещена непосредственно в каждом мультипроцессоре и доступна для чтения и записи всем нитям блока. Ее наличие отличает CUDA от традиционных графических API. Разделяемая память может существенно улучшить производительность GPU-приложения в случае, если ее удается использовать как буфер, заменяющий обращения к глобальной памяти. Всего каждому мультипроцессору устройства с compute capability 2.x может быть доступно 16 или 48 Кбайт разделяемой памяти в зависимости от размера L1-кэша, который может быть настроен программно. Объем разделяемой памяти делится поровну между всеми блоками нитей, запущенными на мультипроцессоре. Разделяемая память также используется для передачи значений аргументов ядра.

Размер разделяемой памяти может быть задан в CUDA-ядре при определении массивов с атрибутом *__shared__* или в параметрах запуска ядра. В последнем случае размеры используемых *__shared__*-массивов могут не указываться. Если таких массивов несколько, то все они будут указывать на начало выделенной блоку дополнительной разделяемой памяти, т.е. если они должны следовать друг за другом, то потребуется явно указывать смещение:

```
__global__ void kernel1(float* a)
{
    // Явно задано выделить 256*4 байт на блок.
    __shared__ float buf [256];

    // Запись значения из глобальной памяти в разделяемую.
```

```
    buf [threadIdx.x] = a [blockIdx.x * blockDim.x + threadIdx.x];

    ...
}

__global__ void kernel2(float* a)
{
    // Размер явно не указан.
    __shared__ float buf [];

    // Запись значения из глобальной памяти в разделяемую.
    buf [threadIdx.x] = a [blockIdx.x * blockDim.x + threadIdx.x];

    ...
}

// Запустить ядро и задать выделяемый (под buf)
// объем разделяемой памяти в байтах.
kernel2<<<dim3(n / 256), dim3(256), k * sizeof(float)>>> ( a );

__global__ void kernel3(float* a, int k)
{
    // Размер явно не указан.
    __shared__ float buf1 [];

    // Размер явно не указан, считаем, что он передан как k.
    __shared__ float buf2 [];

    buf1 [threadIdx.x]     = a [blockIdx.x * blockDim.x + threadIdx.x];
    buf2 [k + threadIdx.x] = a [blockIdx.x * blockDim.x + threadIdx.x + k];

    ...
}
```

В CUDA-программе нельзя гарантировать, что операция, только что завершенная в текущей нити, уже выполнена и в нитях других варпов. По этой причине любая коллективная операция с разделяемой памятью, после которой нить будет использовать значения, измененные другими нитями, должна завершаться *барьерной синхронизацией – __syncthreads()*. Многие приложения используют следующую последовательность действий:

- загрузить необходимые данные в shared-память из глобальной памяти;
- __syncthreads ();

- выполнить вычисления над загруженными данными;
- __syncthreads ();
- записать результат в глобальную память.

3.5.1. Пример: перемножение матриц

Рассмотрим блочный вариант перемножения матриц с использованием разделяемой памяти. Пусть каждый блок GPU вычисляет одну подматрицу C' размером 16×16 (будем считать, что размер матриц N кратен 16). Как показано на рис. 3.8, для вычисления подматрицы C' произведения $A \cdot B$ приходится постоянно обращаться к двух полосам-подматрицам $N \times 16$ исходных матриц A' и B'. Идеальным вариантом было бы разместить копии этих полос в разделяемой памяти, однако с разделяемой памятью размером 48 Кбайт это невозможно: в случае $N = 1024$ одна полоса будет занимать в памяти $1024 \cdot 16 \cdot 4 = 64$ Кбайта. Однако, если каждую из полос разбить на квадратные подматрицы 16×16, то результирующая матрица C' будет суммой попарных произведений подматриц из этих двух полос (рис. 3.9). С таким разбиением вычисление подматрицы C' можно выполнить за $N/16$ шагов, на каждом из которых в разделяемую память загружается одна 16×16 подматрица A и одна подматрица B, что потребует $16 \cdot 16 \cdot 4 \cdot 2 = 2$ Кбайта разделяемой памяти на блок. Каждая нить блока загружает по одному элементу из каждой подматрицы, т.е. на один шаг требуется лишь два обращения в глобальную память.

Поскольку каждая нить загружает только по одному элементу подматрицы, и затем все элементы используются всеми нитями блока, необходимо добавить синхронизацию, которая бы гарантировала полную загрузку всех элементов подматриц (а не только 32 элементов, загруженных данным варпом). Аналогично, синхронизацию необходимо добавить и после вычислений, до загрузки очередной пары, чтобы элементы текущих подматриц гарантированно не использовались какой-либо нитью. Кроме того, в этой версии обращения к глобальной памяти всех нитей блока будут объединены в одну транзакцию (coalesced).

```
__global__ void matmul2(float* a, float* b, int n, float* c)
{
    int bx = blockIdx.x;
    int by = blockIdx.y;
```

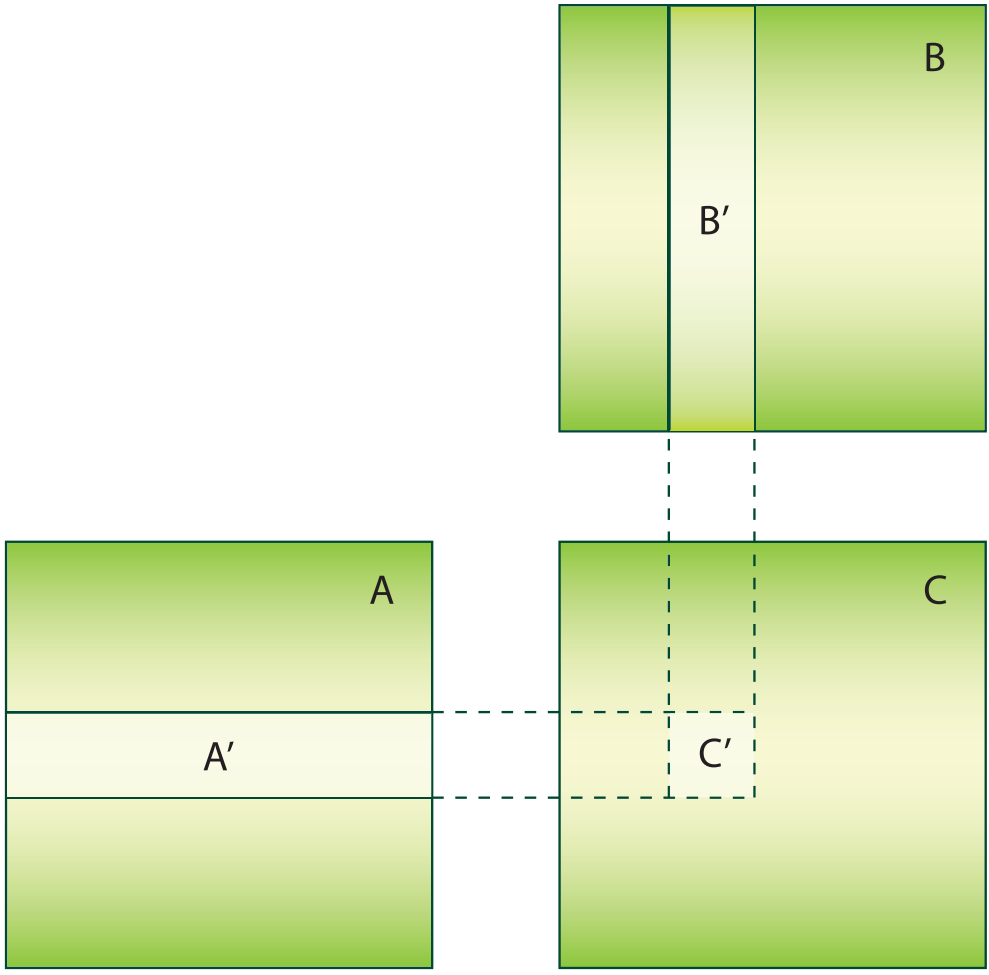


Рис. 3.8. Для вычисления элементов C’ нужны только элементы из A’ и B’

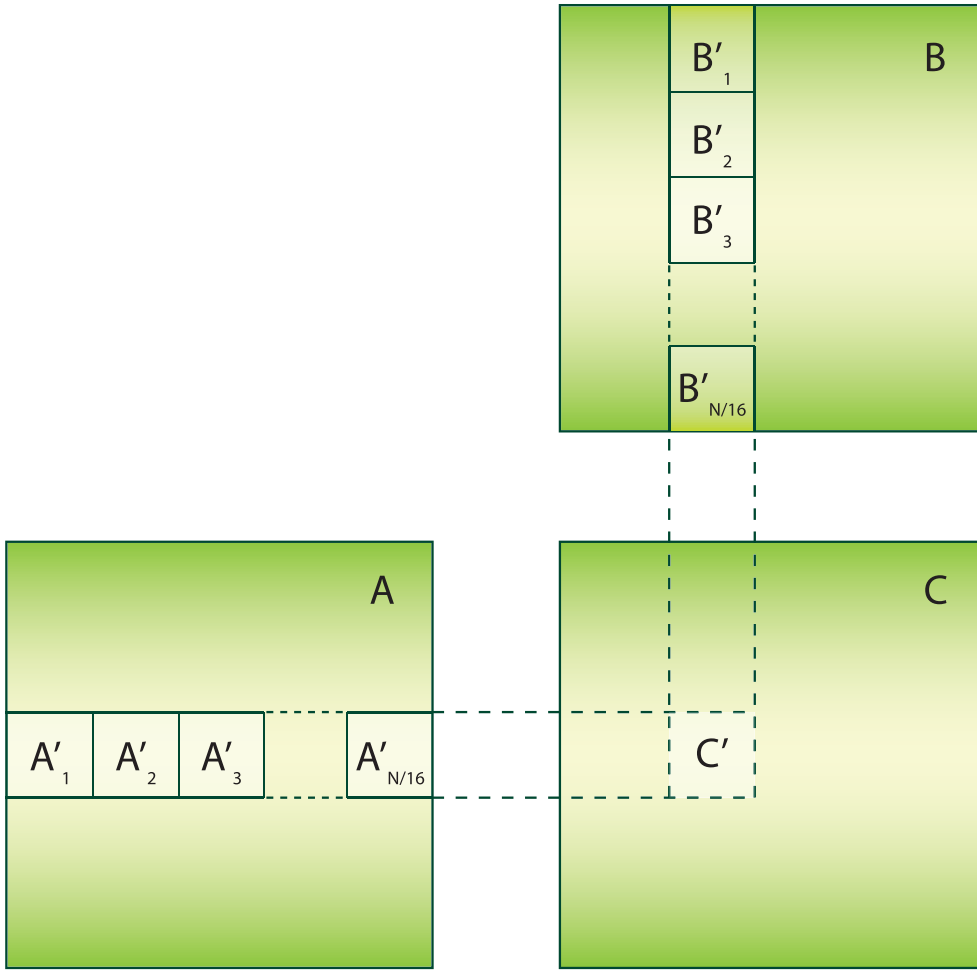


Рис. 3.9. Разложение требуемой подматрицы в сумму произведений матриц 16×16

```
int tx = threadIdx.x;
int ty = threadIdx.y;

// Индекс начала первой подматрицы A обрабатываемой блоком.
int aBegin = n * BLOCK_SIZE * by;
int aEnd = aBegin + n - 1;

// Шаг перебора подматриц A.
int aStep = BLOCK_SIZE;

// Индекс первой подматрицы B обрабатываемой блоком.
int bBegin = BLOCK_SIZE * bx;

// Шаг перебора подматриц B
int bStep = BLOCK_SIZE * n;

// Вычисляемый элемент C'.
float sum = 0.0f;

// Цикл по всем подматрицам
for (int ia = aBegin, ib = bBegin; ia <= aEnd; ia += aStep, ib += bStep)
{
    // Очередная подматрица A в разделяемой памяти.
    __shared__ float as [BLOCK_SIZE][BLOCK_SIZE];

    // Очередная подматрица B в разделяемой памяти.
    __shared__ float bs [BLOCK_SIZE][BLOCK_SIZE];

    // Загрузить по одному элементу из A и B в разделяемую память.
    as [ty][tx] = a [ia + n * ty + tx];
    bs [ty][tx] = b [ib + n * ty + tx];

    // Дождаться когда обе подматрицы будут полностью загружены.
    __syncthreads();

    // Вычислить элемент произведения загруженных подматриц.
    for (int k = 0; k < BLOCK_SIZE; k++)
        sum += as [ty][k] * bs [k][tx];

    // Дождаться пока все остальные нити блока закончат вычислять
    // свои элементы.
    __syncthreads();
}

// Записать результат.
int ic = n * BLOCK_SIZE * by + BLOCK_SIZE * bx;
```

```
    c [ic + n * ty + tx] = sum;
}
#define kernel matmul2
#include "main.h"
```

В отличие от версии, использующей только глобальную память, в данном варианте вместо $2N$ чтений из глобальной памяти достаточно сделать $2N/16$. Количество арифметических операций не изменилось и осталось равным $2N - 1$. Это привело к соответствующему изменению числа обращений в глобальную память (таблица 3.2) и увеличению производительности в 3,5 раза (таблица 3.3). Для сравнения, перемножение матриц также было выполнено с помощью функции библиотеки CUBLAS, дополнительно использующей аналогичный алгоритм блочного перемножения на уровне регистров [4].

Таблица 3.2. Результаты профилирования вариантов перемножения матриц 2048×2048 на Tesla C2070

Метод	**Количество чтений из глобальной памяти на варп в одном SM**	**Количество записей в глобальную память на варп в одном SM**
«в лоб», без использования разделяемой памяти	38535168	9408
С использованием разделяемой памяти	1196032	9344

3.5.2. Эффективный доступ к разделяемой памяти

Для повышения пропускной способности разделяемая память разбита на 16 банков в устройствах с compute capability 1.x и на 32 банка – в более современных архитектурах[1]. В каждый момент времени банк может выполнять одно чтение или запись 32-битного слова. Подряд идущие 32-битные слова попадают в различные

[1] Без ограничения общности все иллюстрации в данном разделе даны для 16 банков.

Таблица 3.3. Быстродействие вариантов перемножения матриц $C = AB, 2048 \times 2048$ на Tesla C2070

Метод	Время, мс
Без использования разделяемой памяти	324,63
С использованием разделяемой памяти	93,26
С использованием CUBLAS	30,84

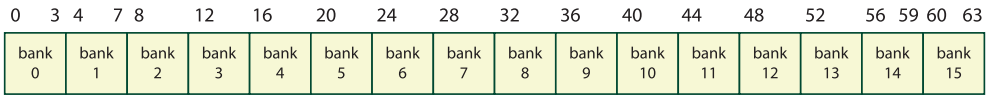


Рис. 3.10. Разбиение 64 байт разделяемой памяти на банки

подряд идущие банки. Если все 32 нити варпа обращаются к 32 32-битным словам, находящимся в разных банках, то данные будут получены без дополнительных задержек. Подобная организация разделяемой памяти имеет цель оптимизировать типичный для приложений характер запросов: чтение различными нитями варпа подряд идущих 32-битных значений (рис. 3.10). Обращения к одному банку могут быть выполнены только последовательно. Одновременный запрос данных из одного банка несколькими нитями называется *конфликтом банков* и характеризуется *порядком конфликта* – максимальным числом обращений в один банк. Если имеет место конфликт второго порядка хотя бы для одного банка, то скорость доступа к разделяемой памяти снижается вдвое. Особым случаем является обращение всех 32 нитей варпа к одному и тому же элементу одного банка: тогда включается режим broadcast-запроса, и конфликта не возникает.

На рис. 3.11 приведены примеры бесконфликтного доступа к разделяемой памяти. В частности, конфликта не возникает при непоследовательном соответствии нитей и банков.

На рис. 3.12 приведены два примера доступа к разделяемой памяти с конфликтами по банкам памяти. В случае, показанном слева, возникает 8 конфликтов вто-

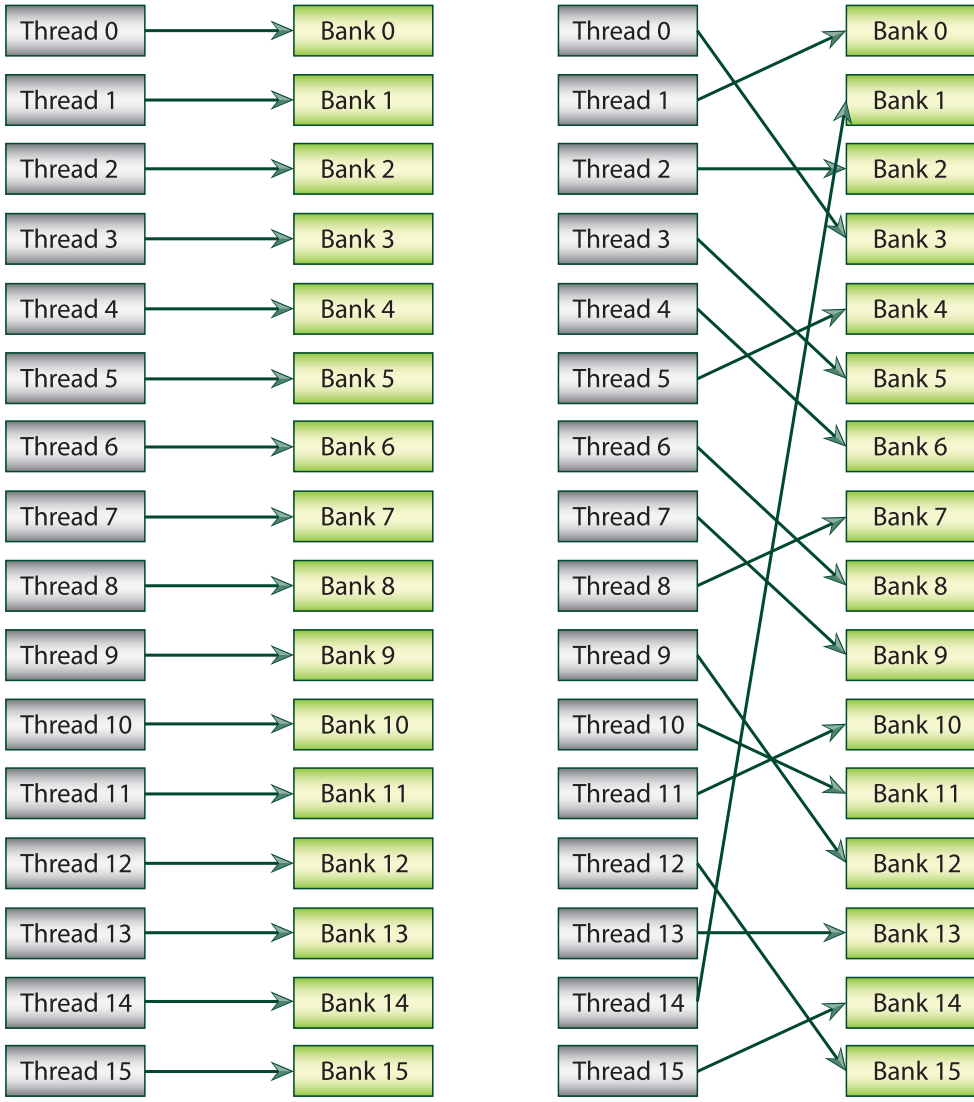


Рис. 3.11. Примеры доступа к разделяемой памяти, в которых не возникает конфликт банков

рого порядка, и скорость работы с разделяемой памятью снижается вдвое, а справа – конфликты 4, 5 и 6-го порядков, приводящие к 6-кратному замедлению.

Рассмотрим также некоторые примеры кода CUDA-ядер. Обычно адреса, по которым производится доступ в разделяемую память, линейно зависят от номера нити. В первом случае конфликтов не будет, однако ситуация меняется при использовании элементов размером менее 32 бит. В следующем фрагменте кода элементы *buf[0]*, *buf[1]*, *buf[2]* и *buf[3]* лежат в одном и том же банке памяти, что приведет к конфликту четвертого порядка. Аналогично в третьем случае будет получен конфликт 2-го порядка.

```
// Нет конфликтов.
__shared__ float buf [128];
float v = buf [baseIndex + threadIdx.x];

// Конфликт 4-го порядка.
__shared__ char buf [128];
char v = buf [baseIndex + threadIdx.x];

// Конфликт 2-го порядка.
__shared__ short buf [128];
short v = buf [baseIndex + threadIdx.x];
```

3.5.3. Пример: умножение матрицы на транспонированную

Рассмотрим перемножение матрицы на транспонированную: $C = AA^T$. В данном случае имеется только одна входная матрица, однако в разделяемой памяти по-прежнему необходимо иметь две подматрицы 16×16, соответствующие A и A^T:

```
__global__ void matmult1(float* a, int n, float* c)
{
    int bx = blockIdx.x;
    int by = blockIdx.y;

    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Индекс первой подматрицы A, обрабатываемой блоком.
    int aBegin = n * BLOCK_SIZE * by;
    int aEnd = aBegin + n - 1;
```

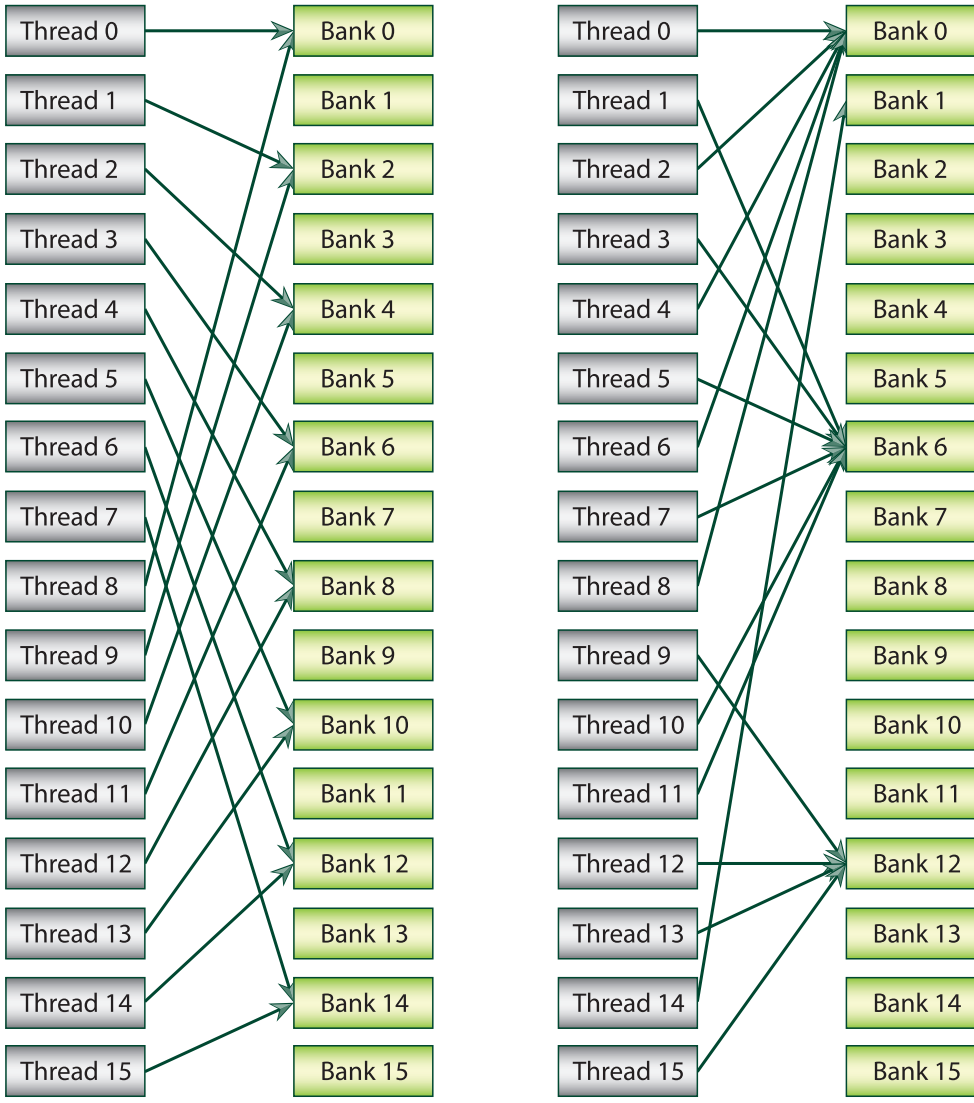


Рис. 3.12. Примеры доступа к разделяемой памяти, в которых возникает конфликт банков

```
    // Индекс первой подматрицы B, обрабатываемой блоком.
    int atBegin = n * BLOCK_SIZE * bx;

    // Вычисляемый элемент C.
    float sum = 0.0f;

    // Цикл по 16*16 подматрицам
    for (int ia = aBegin, iat = atBegin; ia <= aEnd;
        ia += BLOCK_SIZE, iat += BLOCK_SIZE)
    {
        __shared__ float as  [BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float ats [BLOCK_SIZE][BLOCK_SIZE];

        // Загрузить подматрицы в разделяемую память.
        as  [ty][tx] = a [ia  + n * ty + tx];
        ats [ty][tx] = a [iat + n * ty + tx];

        // Синхронизация, чтобы убедиться, что обе подматрицы загружены.
        __syncthreads();

        // Находим нужный элемент произведения подматриц
        for (int k = 0; k < BLOCK_SIZE; k++)
            sum += as [ty][k] * ats [tx][k];

        // Синхронизация, чтобы убедиться, что
        // текущие подматрицы не нужны ни одной нити блока.
        __syncthreads();
    }

    // Записать найденный элемент произведение матриц
    // в глобальную память.
    int ic = n * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    c [ic + n * ty + tx] = sum;
}
#define kernel matmult1
#include "main.h"
```

В отличие от общего случая, доступ к транспонированной матрице в разделяемой памяти (A^T) будет идти не по строкам, а по столбцам, т.е. нити одного варпа будут обращаться к элементам столбца этой матрицы. Поскольку матрица имеет размер 16×16, каждый ее столбец полностью располагается в одном банке, что приводит к банк-конфликту 16-го порядка. Избавиться от конфликтов можно, добавив в матрицу A^T один фиктивный столбец (рис. 3.13). Тогда 16 элементов столбца

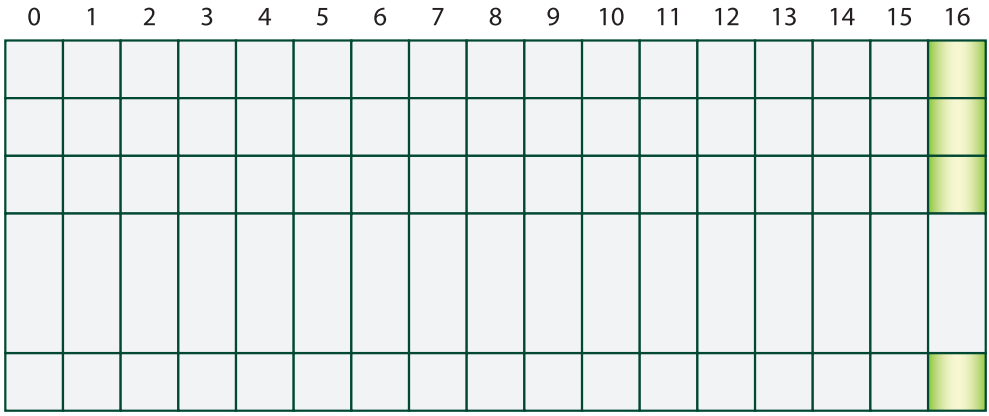


Рис. 3.13. Добавление к матрице 16 × 16 одного столбца для избавления от конфликта банков

будут распределены по всем 16 банкам памяти.

Вставка фиктивных элементов является распространенным приемом выравнивания доступа к памяти и устранения банк-конфликтов:

```
__global__ void matmult2(float* a, int n, float* c)
{
    int bx = blockIdx.x;
    int by = blockIdx.y;

    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Индекс первой подматрицы A, обрабатываемой блоком.
    int aBegin = n * BLOCK_SIZE * by;
    int aEnd = aBegin + n - 1;

    // Индекс первой подматрицы B, обрабатываемой блоком.
    int atBegin = n * BLOCK_SIZE * bx;

    // Вычисляемый элемент C.
    float sum = 0.0f;

    // Цикл по 16*16 подматрицам
    for (int ia = aBegin, iat = atBegin; ia <= aEnd;
        ia += BLOCK_SIZE, iat += BLOCK_SIZE)
    {
        __shared__ float as  [BLOCK_SIZE][BLOCK_SIZE];
```

```
        __shared__ float ats [BLOCK_SIZE][BLOCK_SIZE + 1]; // +1!

        // Загрузить подматрицы в разделяемую память.
        as  [ty][tx] = a [ia  + n * ty + tx];
        ats [ty][tx] = a [iat + n * ty + tx];

        // Синхронизация, чтобы убедиться, что обе подматрицы загружены.
        __syncthreads();

        // Находим нужный элемент произведения подматриц
        for (int k = 0; k < BLOCK_SIZE; k++)
            sum += as [ty][k] * ats [tx][k];

        // Синхронизация, чтобы убедиться, что
        // текущие подматрицы не нужны ни одной нити блока.
        __syncthreads();
    }

    // Записать найденный элемент произведение матриц
    // в глобальную память.
    int ic = n * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    c [ic + n * ty + tx] = sum;
}
#define kernel matmult2
#include "main.h"
```

Таблица 3.4. Быстродействие вариантов перемножения матриц $C = AA^T, 2048 \times 2048$ на Tesla C2070

Метод	Время, мс
Без выравнивания в разделяемой памяти	596,60
С выравниванием в разделяемой памяти	92,50
С использованием CUBLAS	30,77

Глава 4

Взаимодействие CUDA и Fortran

Fortran традиционно имеет широкое применение в инженерных приложениях, следовательно, было бы весьма логично обеспечить его совместимость с CUDA. Глубина такой совместимости может быть разной. Если необходимо лишь запускать ядра на CUDA-C из Fortran, то для этого не требуется никаких дополнительных средств, достаточно лишь написать интерфейсы к используемым функциям CUDA API с помощью модуля ISO_C_BINDING – части стандарта Fortran:

```
function sum_host (a, b, c, n)
    ! Использовать стандартный модуль iso_c_binding,
    ! содержащий типы и функции для написания Си-совместимых программ
    use iso_c_binding
    implicit none

    ! Определить тип dim3, эквивалентный одноименному типу CUDA C
    type, bind(C) :: dim3
        integer(c_int) :: x, y, z
    end type dim3

    ! Интерфейс-блок с описаниями используемых в sum_host
    ! внешних функций runtime-библиотеки CUDA
    interface

        ! Описание прототипа функции cudaMalloc с явным указанием
        ! внешнего имени
        function cudaMalloc(ptr, length) &
            bind(C, name = "cudaMalloc")
            use iso_c_binding
            implicit none

            ! Аргумент ptr - типа "указатель" - void*,
            ! с учетом передачи значения по адресу - void**
            type(c_ptr) :: ptr
            ! Аргумент length - типа size_t, атрибут value -
            ! передается по значению (а не по адресу)
            integer(c_size_t), value :: length
            ! Тип возвращаемого значения функции - int
            integer(c_int) :: cudaMalloc
        end function
```

```
function cudaMemcpy(dst, src, length, dir) &
    bind(C, name = "cudaMemcpy")
    use iso_c_binding
    implicit none

    ! Аргументы dst и src - типа "указатель" - void*,
    ! с учетом передачи по значению (value) - void*
    type(c_ptr), value :: dst, src
    integer(c_size_t), value :: length
    integer(c_int), value :: dir
    integer(c_int) :: cudaMemcpy
end function

function cudaFree(ptr) &
    bind(C, name = "cudaFree")
    use iso_c_binding
    implicit none

    type(c_ptr), value :: ptr
    integer(c_int) :: cudaFree
end function

function cudaGetErrorString(istat) &
    bind(C, name = "cudaGetErrorString")
    use iso_c_binding
    implicit none

    ! Функция cudaGetErrorString возвращает const char*,
    ! но в данном интерфейсе - void*
    type(c_ptr) :: cudaGetErrorString
    integer(c_int), value :: istat
end function

function cudaGetLastError() &
    bind(C, name = "cudaGetLastError")
    use iso_c_binding
    implicit none

    integer(c_int) :: cudaGetLastError
end function

function cudaThreadSynchronize() &
    bind(C, name = "cudaThreadSynchronize")
    use iso_c_binding
    implicit none
```

```
            integer(c_int) :: cudaThreadSynchronize
        end function

        function cudaConfigureCall(gridDim, blockDim, sharedMem, stream) &
            bind(C, name = "cudaConfigureCall")
            use iso_c_binding
            ! Импорт определения типа в интерфейс
            import :: dim3
            implicit none

            type(dim3), value :: gridDim, blockDim
            integer(c_size_t), value :: sharedMem
            type(c_ptr), value :: stream
            integer(c_int) :: cudaConfigureCall
        end function

        function cudaSetupArgument(arg, length, offset) &
            bind(C, name = "cudaSetupArgument")
            use iso_c_binding
            implicit none

            type(c_ptr) :: arg
            integer(c_size_t), value :: length
            integer(c_size_t), value :: offset
            integer(c_int) :: cudaSetupArgument
        end function

        function cudaLaunch(handle) &
            bind(C, name = "cudaLaunch")
            use iso_c_binding
            implicit none

            character(c_char) :: handle
            integer(c_int) :: cudaLaunch
        end function

    end interface

    ! Описание аргументов sum_kernel, target -
    ! для работы функции c_loc
    real, target, dimension(n), intent(in) :: a, b
    real, target, dimension(n), intent(out) :: c
    integer, intent(in) :: n

    type(c_ptr) :: aDev, bDev, cDev
    integer :: sum_host
```

```
integer(c_int) :: istat

! Переменные для обработки строки описания ошибки
type(c_ptr) :: msg_ptr
integer, parameter :: msg_len = 1024
character(len=msg_len), pointer :: msg

! Константы CUDA
integer(c_int), parameter :: cudaMemcpyHostToDevice = 1
integer(c_int), parameter :: cudaMemcpyDeviceToHost = 2

integer(c_size_t), parameter :: zero = 0
type(dim3) :: blocks, threads

sum_host = 1
istat = 0

! Выделить память на GPU.
istat = cudaMalloc(aDev, n * sizeof(a(1)))
if (istat .ne. 0) then
    ! Получить указатель void* на начало строки ошибки
    msg_ptr = cudaGetErrorString(istat)
    ! Привести указатель void* к указателю на строку
    call c_f_pointer(msg_ptr, msg, [msg_len])
    write(*, *) 'Cannot allocate GPU memory for aDev: ', &
        ! Взять часть строки от начала до первого вхождения
        ! символа '0' - маркера конца строки в Си
        msg(1:index(msg, c_null_char))
    return
endif
istat = cudaMalloc(bDev, n * sizeof(b(1)))
if (istat .ne. 0) then
    msg_ptr = cudaGetErrorString(istat)
    call c_f_pointer(msg_ptr, msg, [msg_len])
    write(*, *) 'Cannot allocate GPU memory for bDev: ', &
        msg(1:index(msg, c_null_char))
    return
endif
istat = cudaMalloc(cDev, n * sizeof(c(1)))
if (istat .ne. 0) then
    msg_ptr = cudaGetErrorString(istat)
    call c_f_pointer(msg_ptr, msg, [msg_len])
    write(*, *) 'Cannot allocate GPU memory for cDev: ', &
        msg(1:index(msg, c_null_char))
    return
endif
```

```
! Задать конфигурацию запуска n нитей.
threads = dim3(BLOCK_SIZE, 1, 1)
blocks  = dim3(n / BLOCK_SIZE, 1, 1)

! Скопировать входные данные из памяти CPU в память GPU.
istat = cudaMemcpy(aDev, c_loc(a), n * sizeof(a(1)), &
    cudaMemcpyHostToDevice)
if (istat .ne. 0) then
    msg_ptr = cudaGetErrorString(istat)
    call c_f_pointer(msg_ptr, msg, [msg_len])
    write(*, *) 'Cannot copy aDev data from host to device: ', &
        msg(1:index(msg, c_null_char))
    return
endif
istat = cudaMemcpy(bDev, c_loc(b), n * sizeof(b(1)), &
    cudaMemcpyHostToDevice)
if (istat .ne. 0) then
    msg_ptr = cudaGetErrorString(istat)
    call c_f_pointer(msg_ptr, msg, [msg_len])
    write(*, *) 'Cannot copy bDev data from host to device: ', &
        msg(1:index(msg, c_null_char))
    return
endif

! Задать конфигурация ядра.
istat = cudaConfigureCall(blocks, threads, zero, c_null_ptr)
if (istat .ne. 0) then
    msg_ptr = cudaGetErrorString(istat)
    call c_f_pointer(msg_ptr, msg, [msg_len])
    write(*, *) 'Cannot configure CUDA call: ', &
        msg(1:index(msg, c_null_char))
    return
endif

! Задать значения аргументов ядра
istat = cudaSetupArgument(aDev, sizeof(aDev), zero)
if (istat .ne. 0) then
    msg_ptr = cudaGetErrorString(istat)
    call c_f_pointer(msg_ptr, msg, [msg_len])
    write(*, *) 'Cannot set CUDA kernel argument: ', &
        msg(1:index(msg, c_null_char))
    return
endif
istat = cudaSetupArgument(bDev, sizeof(bDev), sizeof(aDev))
if (istat .ne. 0) then
```

```
    msg_ptr = cudaGetErrorString(istat)
    call c_f_pointer(msg_ptr, msg, [msg_len])
    write(*, *) 'Cannot set CUDA kernel argument: ', &
        msg(1:index(msg, c_null_char))
    return
endif
istat = cudaSetupArgument(cDev, sizeof(cDev), &
    sizeof(aDev) + sizeof(bDev))
if (istat .ne. 0) then
    msg_ptr = cudaGetErrorString(istat)
    call c_f_pointer(msg_ptr, msg, [msg_len])
    write(*, *) 'Cannot set CUDA kernel argument: ', &
        msg(1:index(msg, c_null_char))
    return
endif

! Вызвать ядро с ранее заданной конфигурацией и аргументами
! Ядро идентифицируется Си-строкой с именем (в simple_kernel.cu
! ядро также должно иметь атрибут extern "C", чтобы имя не
! декорировалось).
istat = cudaLaunch('sum_kernel' // c_null_char)
if (istat .ne. 0) then
    msg_ptr = cudaGetErrorString(istat)
    call c_f_pointer(msg_ptr, msg, [msg_len])
    write(*, *) 'Cannot launch CUDA kernel: ', &
        msg(1:index(msg, c_null_char))
    return
endif
istat = cudaGetLastError()
if (istat .ne. 0) then
    msg_ptr = cudaGetErrorString(istat)
    call c_f_pointer(msg_ptr, msg, [msg_len])
    write(*, *) 'Cannot execute CUDA kernel: ', &
        msg(1:index(msg, c_null_char))
    return
endif

! Ожидать завершения работы ядра.
istat = cudaThreadSynchronize()
if (istat .ne. 0) then
    msg_ptr = cudaGetErrorString(istat)
    call c_f_pointer(msg_ptr, msg, [msg_len])
    write(*, *) 'Cannot synchronize CUDA kernel: ', &
        msg(1:index(msg, c_null_char))
    return
endif
```

```
    ! Скопировать результаты в память CPU.
    istat = cudaMemcpy(c_loc(c), cDev, n * sizeof(c(1)), &
        cudaMemcpyDeviceToHost)
    if (istat .ne. 0) then
        msg_ptr = cudaGetErrorString(istat)
        call c_f_pointer(msg_ptr, msg, [msg_len])
        write(*, *) 'Cannot copy cDev data from device to host: ', &
            msg(1:index(msg, c_null_char))
        return
    endif

    ! Освободить выделенную память GPU.
    istat = cudaFree(aDev)
    istat = cudaFree(bDev)
    istat = cudaFree(cDev)

    sum_host = 0
    return
end
```

Функция *sum_host* может быть использована совместно с ядром *sum_kernel* на CUDA C, приведенным в Главе 2. Дополнительно в заголовок *sum_kernel* необходимо добавить *extern "C"*:

```
extern "C" __global__ void sum_kernel ( float * a, float * b, float * c )
```

В данном примере функции CUDA API доступны обычной программе на языке Fortran посредством интерфейсов и C-совместимых типов. Программа может быть собрана любым компилятором с поддержкой *ISO_C_BINDING* стандарта Fortran 2003 и соответствующих встроенных функций. Функция *sum_host* содержит блок интерфейсов с описанием каждой используемой функции: имя, типы аргументов и возвращаемого значения, а также метод передачи аргументов (по адресу или по значению). Дополнительно определены константы направлений копирования и тип *dim3*. Оператор запуска ядра является специфическим расширением CUDA C и не имеет эквивалента в языке Fortran. Его можно заменить командами *CUDA Driver API – cudaConfigureCall*, *cudaSetupArgument*, *cudaLaunch*, производящими определение конфигурации вычислительной сети, загрузку значений аргументов в стек и запуск ядра с заданным именем или адресом соответственно.

4.1. Введение в CUDA Fortran

Для более тесной интеграции CUDA и Fortran, включающей поддержку Fortran в GPU-ядрах, расширения, эквивалентные CUDA C, необходимо добавить уже в сам язык. Более того, Fortran не идентичен C, имея, к примеру, более широкие возможности работы с многомерными массивами. Таким образом, для наибольшего удобства использования, CUDA-расширения должны органично встраиваться в существующие элементы языка. В этом разделе будет рассмотрена реализация PGI CUDA Fortran 2011 (далее, CUDA Fortran).

CUDA Fortran реализует стандартное разделение функциональности между хостом и GPU. На хосте производится выделение памяти на GPU и pinned-памяти основной системы, инициируется обмен данными и запуск CUDA-ядер. Код для GPU – это также код на языке Fortran с процедурами-ядрами и device-подпрограммами, помеченными соответствующими атрибутами.

4.1.1. Элементы host-части программы

Следуя специфике Fortran, для выделения и высвобождения памяти переменным и массивам в CUDA Fortran используются стандартные функции *allocate* и *deallocate*. По этой причине необходимый признак принадлежности к GPU – атрибут “device” – должен быть добавлен в объявления переменных:

```
real, device, allocatable :: a(:)
real, attributes(device) :: a
...
real, device, allocatable :: a(:,:), b
allocate( a(1:n,1:m), b )
...
deallocate( a, b )
```

В коде хост-программы память может быть выделена под *allocatable* или автоматические переменные, которые могут быть затем переданы в виде параметров другим хост-подпрограммам или процедурам-ядрам. На хосте, благодаря наличию виртуальной памяти, эти вызовы крайне редко завершаются ошибкой, поэтому ее возникновение обычно не проверяется. При использовании CUDA Fortran это может быть плохой практикой, поскольку в случае device-переменных память на GPU может кончиться.

Важным элементом языка Fortran являются модули – расширенный аналог пространства имен C++ (namespace). CUDA Fortran позволяет объединять в одном модуле несколько CUDA-ядер и их общие данные. Также переменная или массив может иметь атрибут “constant”, обозначающий размещение в константной памяти GPU:

```
module mm
real, device, allocatable :: a(:)
real, device :: x, y(10)
real, constant :: c1, c2(10)
integer, device :: n
contains
attributes(global) subroutine s( b )
...
```

Для копирования данных между CPU и GPU в CUDA Fortran через модуль cudafor доступны функции, аналогичные CUDA API:

```
use cudafor
real, allocatable, device :: a(:)
real :: b(10), b2(2), c(10)
...
istat = cudaMalloc( a, 10 )
istat = cudaMemcpy( a, b, 10 )
istat = cudaMemcpy( a(2), b2, 2)
istat = cudaMemcpy( c, a, 10 )
istat = cudaFree( a )
```

Существует второй, возможно, более удобный способ копировать данные – непосредственно с помощью оператора присваивания:

```
real, device, allocatable :: a(:,:), b
allocate( a(1:n,1:m), b )
a(1:n,1:m) = x(1:n,1:m)
! copies to device
b = 99.0
...
x(1:n,1:m) = a(1:n,1:m)
! copies from device
y = b
deallocate( a, b )
```

Как и стандартный Fortran, CUDA Fortran позволяет указывать диапазоны индексов для копирования или присваивания значения только определенным частям

массивов. Это возможно благодаря тому, что в Fortran следом за указателем на данные в памяти хранится структура, описывающая размерности allocatable-массива. Однако, в случае диапазонов передаваемый участок памяти перестает быть непрерывным, а значит требует множества отдельных операций копирования, что может понизить скорость пересылки данных.

Запуск ядра в CUDA Fortran производится точно так же, как в CUDA C с добавлением ключевого слова *call*: имя процедуры-ядра, конфигурация вычислительной сети в тройных угловых скобках и список аргументов:

```
call gpu_vecadd_kernel <<< (N+31)/32, 32 >>> ( A, B, C, N )
type(dim3) :: g, b
g = dim3((N+31)/32, 1, 1)
b = dim3( 32, 1, 1 )
call gpu_vecadd_kernel <<< g, b >>> ( A, B, C, N)
```

Размеры сетки и блока могут быть целыми числами или значениями типа dim3. Процедура-ядро доступна для запуска при выполнении одного из следующих трех условий:

- Для процедуры-ядра в вызывающей подпрограмме определен явный интерфейс (*interface*)
- Процедура-ядро определена в используемом модуле (*use*)
- Процедура-ядро определена в том же модуле, что и вызывающая функция

4.1.2. Программирование GPU-ядер

Поскольку для GPU-ядер в CUDA запрещен возврат значений через *return* (только void-функции), в CUDA Fortran CUDA-ядрами могут быть только процедуры. Признаком процедуры для GPU является наличие в заголовке атрибута “global”:

```
attributes(global) subroutine kernel ( A, B, C, N)
```

В процедуре-ядре могут быть определены переменные (скаляры и массивы фиксированной длины), локальные по отношению к каждой нити, и массивы в памяти, разделяемой нитями одного блока:

```
real, shared :: sm(16,16)
```

В ядре поддерживается использование основных встроенных типов данных, а также созданных из них производных типов (структур):

```
integer(1,2,4,8), logical(1,2,4,8), real(4,8),
complex(4,8), character(len=1)
```

В CUDA Fortran доступен стандартный набор встроенных переменных CUDA: *blockidx*, *threadidx*, *griddim*, *blockdim*, *warpsize*. Поддержка языка в коде процедур-ядер ограничена следующим множеством конструкций:

- оператор присваивания;
- do, if, goto, case;
- call (подпрограмм с атрибутом “device”);
- intrinsics;
- where, forall.

4.1.3. Правила передачи аргументов

При взаимодействии Fortran и C необходимо согласовывать правила передачи аргументов. В Fortran по умолчанию **все** аргументы подпрограмм передаются по адресу, тогда как в C скалярные величины передаются как значения. Если в программе на языке Fortran необходимо передать аргумент по значению, определение соответствующей переменной вызываемой функции должно содержать атрибут *value*:

```
integer(c_int), value :: dir
```

CUDA Fortran реализует стандартный метод передачи параметров, однако он не очень хорошо согласуется с разделением адресных пространств, характерным для CUDA-программ: аргумент, переданный при запуске CUDA-ядра, не может быть корректно прочитан со стороны GPU, поскольку его адрес соответствует памяти хоста! Таким образом, скалярные переменные должны либо передаваться как значения с помощью атрибута *value*, либо копироваться в память GPU, подобно массивам.

4.1.4. Правила видимости

Процедура-ядро с атрибутом “global” в модуле:

- Может напрямую обращаться к данным в памяти GPU, объявленным в этом же модуле
- Может вызывать функции и процедуры с атрибутом “device”, определенные в этом же модуле

Процедура или функция с атрибутом “device” в модуле:

- Может напрямую обращаться к данным в памяти GPU, объявленным в этом же модуле;
- Может вызывать функции и процедуры с атрибутом “device”, определенные в этом же модуле;
- Неявно опередедена как приватная.

Процедура-ядро с атрибутом “global” вне модуля:

- Не может обращаться к данным в памяти GPU, за исключением аргументов.

Процедура или функция, выполняемая на хосте:

- Может вызывать процедуры-ядра, определенные как в модулях, так и вне модулей;
- Может обращаться к любым данным в модулях;
- Может вызывать ядра на CUDA C при наличии интерфейса.

4.1.5. CUDA Fortran и CUDA C

CUDA Fortran обеспечивает простую форму взаимодействия с CUDA C. Для использования ядер на CUDA C из Fortran-программы достаточно определить интерфейс процедуры-ядра:

```
interface
    attributes(global) subroutine saxpy(a,x,y,n) bind(C)
        real, device :: x(*), y(*)
        real, value :: a
        integer, value :: n
    end subroutine
end interface
...
call saxpy<<<grid,block>>>( aa, xx, yy, nn )
```

Для использования ядер на CUDA Fortran из C-программы достаточно задать прототип ядра с добавлением стандартного в именовании Fortran знака подчеркивания:

```
extern __global__ void saxpy_(float a, float* x, float* y, int n);
...
saxpy_( a, x, y, n );
```

В таблице 4.1 приведено краткое сравнение основных возможностей CUDA Fortran и CUDA C. CUDA Fortran не поддерживает текстурную память, низкоуровневый CUDA Driver API и графические API. Функции CUDA API доступны как в традиционном виде, так и через встроенные операторы языка. Индексация элементов массивов и встроенных переменных ведется с единицы. Для удобства использования прикладных библиотек в CUDA Fortran предопределены соответствующие модули.

4.1.6. Компиляция

Для сборки объектов из исходных файлов с ядрами на CUDA Fortran, а также для линковки полученных объектов в исполняемый образ необходимо использовать компилятор pgfortran с опцией -Mcuda:

```
pgfortran –Mcuda[=[emu|cc10|cc11|cc12|cc13|cc20]] a.cuf
```

Стандратные суффиксы файлов CUDA Fortran: .cuf – free form, .CUF – free form и препроцессор, опция pgfortran -Mfixed – fixed form.

4.1.7. Компактная форма записи

В CUDA Fortran доступен еще один способ определения вычислительного ядра, уже по сути являющийся не CUDA-ядром, а директивой подобной OpenMP:

```
!$cuf kernel do(2) <<< (*,*), (32,4) >>>
do j = 1, m
    do i = 1, n
        a(i,j) = b(i,j) + c(i,j)
    end do
end do
```

В данном примере двумерный цикл отображается на двумерную сетку блоков 32×4, размерность сетки потоков (*, *) вычисляется во время исполнения делением размерностей циклов на размерности сетки блоков. Директивный метод программирования GPU получил дальнейшее развитие в рамках технологий PGI Accelerator и OpenACC.

Таблица 4.1. Сравнение основных возможностей CUDA Fortran и CUDA C

CUDA C	**CUDA Fortran**
Текстуры, текстурная память	Текстурная память недоступна
Поддержка CUDA Runtime API	Поддержка CUDA Runtime API
Поддержка Driver API	Driver API недоступен
cudaMalloc, cudaFree	allocate, deallocate
cudaMemcpy	Оператор присваивания
OpenGL, Direct3D	OpenGL и Direct3D недоступны
Индексация элементов массивов и блоков/потоков – с нуля	Индексация элементов массивов и блоков/потоков – с единицы
Специализированные библиотеки	Модули с коллекциями функций, например *use cublas*

Глава 5

Некоторые алгоритмы обработки массивов

В силу закона Амдала, эффективность параллельных вычислений определяется свойствами используемых алгоритмов. Для некоторых задач, таких как перемножение матриц или моделирование взаимодействия материальных тел, параллельная реализация достаточно тривиальна. Но существует и множество на первый взгляд последовательных задач, для которых тем не менее возможно построить нетривиальные параллельные эквиваленты. В этой главе рассмотрен ряд таких задач из области операций над массивами.

5.1. Параллельная редукция

Пусть задан массив $a_0, a_1, ..., a_{n-1}$ и некоторая бинарная ассоциативная операция. Без ограничения общности в качестве операции далее будет рассматриваться сложение. Тогда *редукцией* массива $a_0, a_1, ..., a_{n-1}$ относительно сложения называется:

$$A = (((a_0 + a_1) + a_2) + \ldots + a_{n-1})$$

Редукция тривиально реализуема следующим последовательным кодом:

```
int sum = 0;
for (int i = 0; i < n; i++)
    sum += a [i];
```

Распараллелим редукцию методом «разделяй и властвуй». Разделим весь исходный массив на части, и каждой из них поставим в соответствие один блок сетки блоков. Поскольку в CUDA максимальная размерность блока невелика (обычно 65535, см. свойства GPU), большие одномерные массивы может быть необходимо представить как дву- или трехмерные, чтобы увеличить диапазон индексации. Если в каждом блоке суммировать только соответствующую часть элементов массива, то исходная задача преобразуется в набор независимых подзадач по нахождению частичных сумм. Для распараллеливания каждой подзадачи по отдельным нитям внутри блока аналогичным образом продолжим делить подмассив сначала на пары элементов, затем на пары из их частичных сумм и так далее вплоть до одной

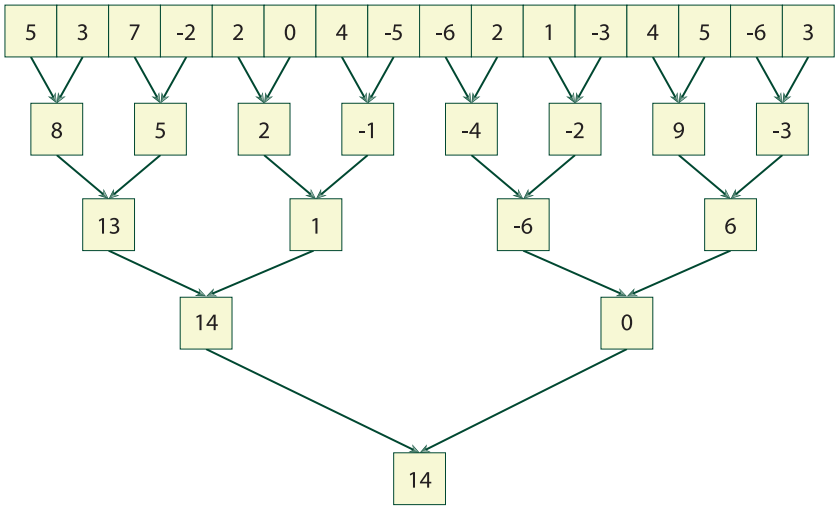


Рис. 5.1. Параллельное суммирование элементов массива

пары, суммирующей две половины подмассива. Таким образом, на каждом шаге суммирования число элементов будет уменьшаться вдвое, например, в случае 512 элементов в блоке потребуется $\log_2 512 = 9$ шагов. Описанный процесс для массива из 16 элементов показан на рис. 5.1.

Если операция над элементами проста, то основным фактором, ограничивающим производительность в данной задаче будет доступ к памяти. Повысить эффективность может предварительное копирование соответствующих элементов блока в разделяемую память. В этом случае данные из медленной глобальной памяти будут загружены единственный раз, и при нахождении частичных сумм будет использоваться только быстрая разделяемая память. На рис. 5.2 показано соответствие нитей, элементов массива и шагов суммирования для данного метода, ниже приведен вариант реализации CUDA-ядра и заголовочный файл тестового приложения.

```
__global__ void reduce1 (int* inData, int* outData)
{
    // Суммируемые данные в разделяемой памяти.
    __shared__ int data [BLOCK_SIZE];
    int tid = threadIdx.x;
    int i   = blockIdx.x * blockDim.x + threadIdx.x;

    // Загрузить данные в разделяемую память.
```

```
    data [tid] = inData [i];

    // Заблокировать нити блока до окончания загрузки данных.
    __syncthreads ();

    // Выполнять попарное суммирование.
    for ( int s = 1; s < blockDim.x; s *= 2 )
    {
        // Проверить, участвует ли нить на данном шаге.
        if ( tid % (2 * s) == 0 )
            data [tid] += data [tid + s];

        __syncthreads ();
    }

    // Первая нить записывает итоговую сумму.
    if ( tid == 0 )
        outData [blockIdx.x] = data [0];
}
#define kernel reduce1
#include "main1.h"

#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>

int main ( int argc, char *  argv [] )
{
    if (argc != 3)
    {
        printf("Usage: %s <power> <multiply>\n", argv[0]);
        printf("where length will be %d^power * multiply,\n", BLOCK_SIZE);
        printf("and power > 1, multiple <= %d\n", BLOCK_SIZE);
        return 0;
    }
    int len = atoi(argv[2]);
    if ((len < 1) || (len > BLOCK_SIZE))
    {
        fprintf(stderr, "Invalid length multiplier: %d\n", len);
        return 1;
    }
    int power = atoi(argv[1]);
    if (power < 1)
    {
        fprintf(stderr, "Invalid power: %d\n", power);
        return 1;
```

```
}
for (int i = 0; i < power; i++)
    len *= BLOCK_SIZE;

printf("Length = %d\n", len);
int lenb = len * sizeof(int);
int* a = (int*)malloc(lenb);
int* b = (int*)malloc(lenb);

// Генерация входных данных и расчет контрольной суммы.
int sum = 0;
int sum = 0;
for (int i = 0; i < len; i++)
{
    a[i] = rand() % 2;
    sum += a[i];
}

int* adev[2] = { NULL, NULL };
cudaEvent_t start, stop;
float gpuTime = 0.0f;
cudaError_t cuerr = cudaMalloc ((void**)&adev[0], lenb);
if (cuerr != cudaSuccess)
{
    fprintf(stderr, "Cannot allocate device memory: %s\n",
        cudaGetErrorString(cuerr));
    return 1;
}
cuerr = cudaMalloc ((void**)&adev[1], lenb);
if (cuerr != cudaSuccess)
{
    fprintf(stderr, "Cannot allocate device memory: %s\n",
        cudaGetErrorString(cuerr));
    return 1;
}

// Создание событий для контроля времени.
cuerr = cudaEventCreate (&start);
if (cuerr != cudaSuccess)
{
    fprintf(stderr, "Cannot create CUDA event: %s\n",
        cudaGetErrorString(cuerr));
    return 1;
}
cuerr = cudaEventCreate (&stop);
if (cuerr != cudaSuccess)
```

```
    {
        fprintf(stderr, "Cannot create CUDA event: %s\n",
            cudaGetErrorString(cuerr));
        return 1;
    }
    cuerr = cudaMemcpy (adev[0], a, lenb, cudaMemcpyHostToDevice);
    if (cuerr != cudaSuccess)
    {
        fprintf(stderr, "Cannot copy data from host to device: %s\n",
            cudaGetErrorString(cuerr));
        return 1;
    }
    cuerr = cudaEventRecord (start, 0);
    if (cuerr != cudaSuccess)
    {
        fprintf(stderr, "Cannot record CUDA event: %s\n",
            cudaGetErrorString(cuerr));
        return 1;
    }

    int i = 0, n = len;
    for ( ; n >= BLOCK_SIZE; n /= BLOCK_SIZE, i ^= 1)
    {
#ifdef _DEBUG
        printf ("n= %d i = %d\n", n, i);
#endif
        dim3 threads (BLOCK_SIZE, 1, 1);
        dim3 blocks (n / threads.x, 1, 1);

        // Запуск ядра на GPU.
        kernel<<<blocks, threads>>> (adev[i], adev [i ^ 1]);
        cuerr = cudaGetLastError();
        if (cuerr != cudaSuccess)
        {
            fprintf(stderr, "Cannot launch kernel: %s\n",
                cudaGetErrorString(cuerr));
            return 1;
        }
        cuerr = cudaDeviceSynchronize();
        if (cuerr != cudaSuccess)
        {
            fprintf(stderr, "Cannot synchronize kernel: %s\n",
                cudaGetErrorString(cuerr));
            return 1;
        }
    }
```

```
    cuerr = cudaEventRecord (stop, 0);
    if (cuerr != cudaSuccess)
    {
        fprintf(stderr, "Cannot record CUDA event: %s\n",
            cudaGetErrorString(cuerr));
        return 1;
    }
    cuerr = cudaMemcpy (b, adev[i], sizeof(int) * n, cudaMemcpyDeviceToHost);
    if (cuerr != cudaSuccess)
    {
        fprintf(stderr, "Cannot copy data from device to host: %s\n",
            cudaGetErrorString(cuerr));
        return 1;
    }
    cuerr = cudaEventSynchronize (stop);
    if (cuerr != cudaSuccess)
    {
        fprintf(stderr, "Cannot synchronize CUDA event: %s\n",
            cudaGetErrorString(cuerr));
    }
    cuerr = cudaEventElapsedTime (&gpuTime, start, stop);
#ifdef _DEBUG
    printf ( "n= %d i = %d\n", n, i );
#endif
    for (int i = 1; i < n; i++)
        b[0] += b[i];

    // Печать результатов и времен исполнения.
    printf ("time spent executing on GPU: %.2f milliseconds\n", gpuTime);
    printf ("CPU sum = %d, CUDA sum = %d\n", sum, b[0]);

    // Высвобождение ресурсов.
    cudaEventDestroy (start);
    cudaEventDestroy (stop);
    cudaFree (adev[0]);
    cudaFree (adev[1]);
    free(a);
    free(b);

    return 0;
}
```

Недостатком первого варианта является большое количество итераций цикла по *s*, в которых нити одного варпа переходят в различные ветви условного операто-

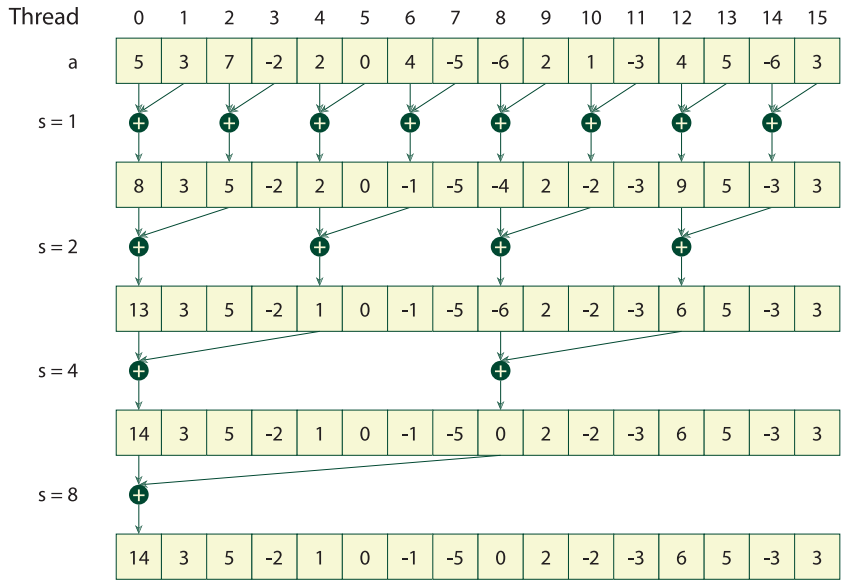


Рис. 5.2. Распределение данных и операций по нитям для первого варианта редукции

ра. В этом случае все нити варпа выполняют все ветви. Вызванные этим эффектом избыточные вычисления можно исключить, перераспределив данные и операции, как это показано на рис. 5.3.

Ниже приведен код соответствующего CUDA-ядра.

```
__global__ void reduce2 (int* inData, int* outData)
{
    __shared__ int data [BLOCK_SIZE];
    int tid = threadIdx.x;
    int i   = blockIdx.x * blockDim.x + threadIdx.x;

    // Загрузить данные в разделяемую память.
    data [tid] = inData [i];

    __syncthreads ();

    for (int s = 1; s < blockDim.x; s <<= 1)
    {
        // Проверить участвует ли нить в суммировании.
        int index = 2 * s * tid;
        if (index < blockDim.x)
            data[index] += data[index + s];
```

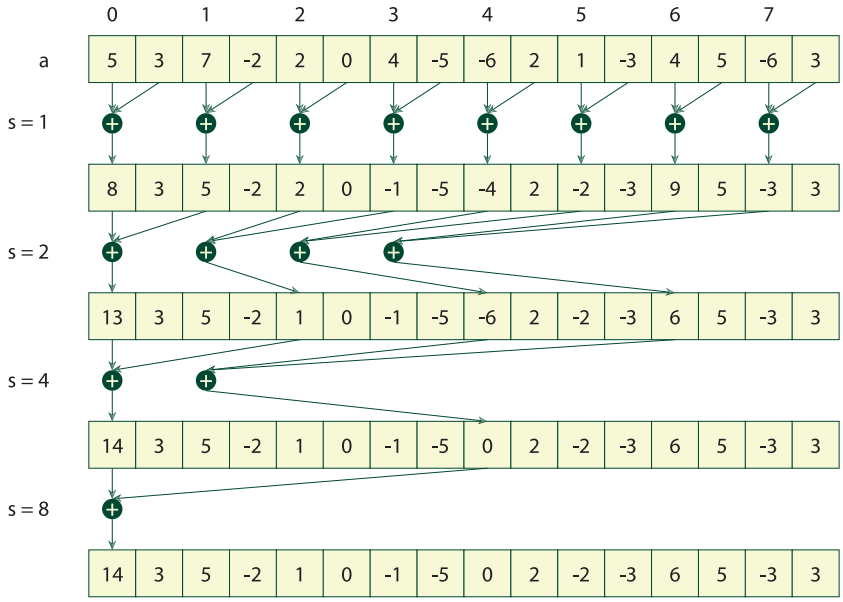


Рис. 5.3. Распределение данных и операций по нитям для второго варианта редукции

```
        __syncthreads ();
    }

    // Первая нить записывает итоговую сумму.
    if ( tid == 0 )
        outData [blockIdx.x] = data [0];
}
#define kernel reduce2
#include "main1.h"
```

Во втором варианте ядра на каждом шаге цикла по *s* будет не более одного варпа с ветвлением, причем это ветвление будет иметь место только для нескольких последних итераций цикла, а в остальных случаях – приходиться на границу между варпами. Второй вариант также имеет недостаток: при *s=2* все обращения к разделяемой памяти соответствуют банкам с четными номерами, что приводит к конфликту банков 2-го порядка. Аналогично, при *s=4* все обращения соответствуют банкам с номерами 0, 4, 8 и 12, что приводит к конфликту 4-го порядка и т.д. Таким образом, данная реализация будет постоянно приводить к конфликтам

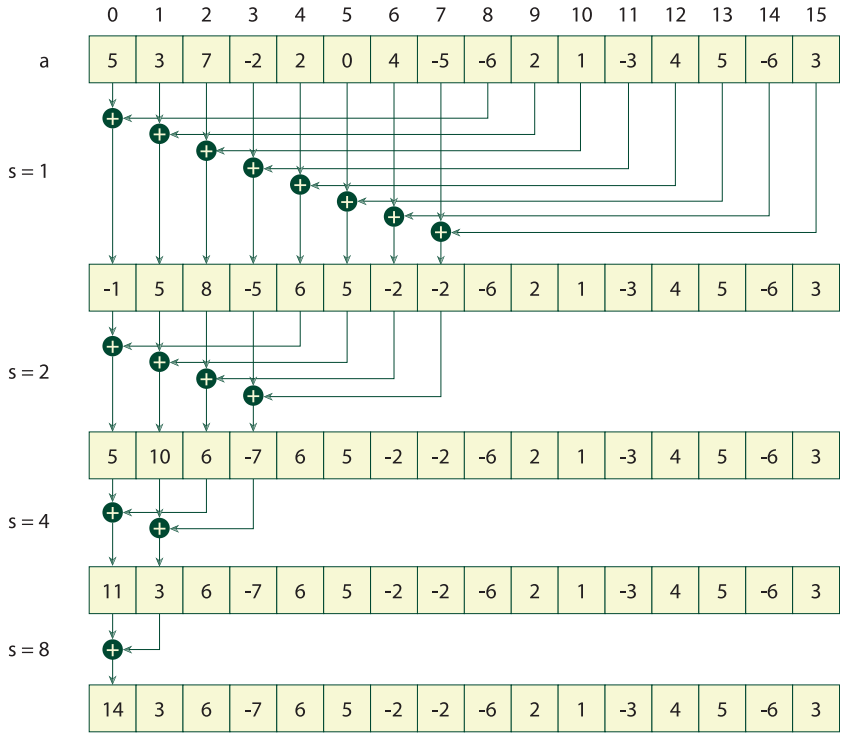


Рис. 5.4. Распределение данных и операций по нитям для третьего варианта редукции

по банкам высокого порядка. Избавиться от конфликтов можно просто изменив порядок выбора пар: вместо выбора пары соседних элементов с удвоением расстояния на каждой следующей операции, следует начать с пар элементов, находящихся на расстоянии *BLOCK_SIZE/2*, и на каждом шаге уменьшать расстояние вдвое (рис. 5.4). Ниже приведен код соответствующего CUDA-ядра.

```
__global__ void reduce3 (int* inData, int* outData)
{
    __shared__ int data [BLOCK_SIZE];
    int tid = threadIdx.x;
    int i   = blockIdx.x * blockDim.x + threadIdx.x;

    // Загрузить данные в разделяемую память. data [tid] = inData [i];

    __syncthreads ();

    for (int s = blockDim.x / 2; s > 0; s >>= 1)
```

```
    {
        if (tid < s)
            data [tid] += data [tid + s];

        __syncthreads ();
    }

    // Первая нить записывает итоговую сумму. if ( tid == 0 )
        outData [blockIdx.x] = data [0];
}
#define kernel reduce3
#include "main1.h"
```

Конфликты по банкам исключены, тем не менее на первом шаге цикла будет загружена только половина нитей блока. Нити можно загрузить полностью, если уменьшить число блоков вдвое, и в каждом блоке обрабатывать вдвое больше элементов. Чтобы избежать увеличения требуемого объема разделяемой памяти, суммирование первых пар можно проводить одновременно с записью в разделяемую память. Ниже приводятся ядро и изменения в заголовочном файле тестового приложения.

```
__global__ void reduce4 (int* inData, int* outData)
{
    __shared__ int data [BLOCK_SIZE];
    int tid = threadIdx.x;
    int i   = 2 * blockIdx.x * blockDim.x + threadIdx.x;

    // Записать сумму первых двух элементов в разделяемую память.
    data [tid] = inData [i] + inData [i + blockDim.x];

    __syncthreads ();

    for (int s = blockDim.x / 2; s > 0; s >>= 1)
    {
        if ( tid < s )
            data [tid] += data [tid + s];

        __syncthreads ();
    }

    // Первая нить записывает итоговую сумму.
    if ( tid == 0 )
        outData [blockIdx.x] = data [0];
}
```

```
#define kernel reduce4
#include "main2.h"

    // Аналогично main1.h
    ...
    int i = 0, n = len;
    for ( ; n >= BLOCK_SIZE; n /= (2 * BLOCK_SIZE), i ^= 1)
    {
#ifdef _DEBUG
        printf ("n= %d i = %d\n", n, i);
#endif
        // Set kernel launch configuration
        dim3 block (BLOCK_SIZE, 1, 1);
        dim3 grid  (n / (2 * block.x), 1, 1);

        // Запуск ядра на GPU.
        kernel<<<grid, block>>> (adev[i], adev [i ^ 1]);
        cuerr = cudaGetLastError();
        if (cuerr != cudaSuccess)
        {
            fprintf(stderr, "Cannot launch kernel: %s\n",
                cudaGetErrorString(cuerr));
            return 1;
        }
        cuerr = cudaDeviceSynchronize();
        if (cuerr != cudaSuccess)
        {
            fprintf(stderr, "Cannot synchronize kernel: %s\n",
                cudaGetErrorString(cuerr));
            return 1;
        }
    }
    // Аналогично main1.h
    ...
```

Так как в одном варпе 32 нити, то при $s \leq 32$ в каждом блоке останется всего по одному варпу, и все нити будут синхронными по отношению друг к другу. Поэтому в данной части цикла по s можно исключить проверки и синхронизацию. До определенного предела эффективность также повышает *раскрутка цикла* – запись нескольких итераций подряд. Однако в отсутствие синхронизации компилятор по умолчанию предполагает, что значение элемента массива между двумя последовательными чтениями не изменяется. Поскольку в данном случае это не так, то необходимо специальное указание: ключевое слово *volatile*.

```
__global__ void reduce5 (int* inData, int* outData)
{
    volatile __shared__ int data [BLOCK_SIZE];
    int tid = threadIdx.x;
    int i   = 2 * blockIdx.x * blockDim.x + threadIdx.x;

    // Записать сумму первых двух элементов в разделяемую память.
    data [tid] = inData [i] + inData [i + blockDim.x];

    __syncthreads ();

    for (int s = blockDim.x / 2; s > 32; s >>= 1)
    {
        if ( tid < s )
            data [tid] += data [tid + s];

        __syncthreads ();
    }

    // Раскрутить последние итерации.
    {
        data [tid] += data [tid + 32];
        data [tid] += data [tid + 16];
        data [tid] += data [tid +  8];
        data [tid] += data [tid +  4];
        data [tid] += data [tid +  2];
        data [tid] += data [tid +  1];
    }

    // Первая нить записывает итоговую сумму.
    if ( tid == 0 )
        outData [blockIdx.x] = data [0];
}
#define kernel reduce5
#include "main2.h"
```

Рассмотрим теперь, как можно полностью посчитать сумму элементов большого массива. Исходный массив разбивается на блоки по 512 элементов, и каждый блок суммируется с использованием ядра reduce5. В результате его работы будет получен массив сумм элементов каждого блока, размер которого в 512 раз меньше. Если размер нового массива мал, то суммирование можно произвести и на CPU, иначе следует снова применить редуцирующее CUDA-ядро. За небольшое число таких шагов может быть получена искомая сумма. Ниже приводится код функции.

```
int reduce(int * data, int n)
{
    int * sums      = NULL;
    int   numBlocks = n / 512;
    int   res       = 0;

    // Выделить память под массив сумм блоков.
    cudaMalloc ( (void **) &sums, numBlocks * sizeof ( int ) );

    // Провести поблочную редукцию, записав
    // суммы для каждого блока в массив sums.
    reduce5<<< dim3 ( numBlocks ), dim3 ( BLOCK_SIZE ) >>> ( data, sums );

    // Теперь редуцируем массив сумм для блоков.
    if ( numBlocks > BLOCK_SIZE )
        res = reduce ( sums, numBlocks );
    else
    {
        // Если значений мало, то просуммируем явно.
        int * sumsHost = new int [numBlocks];

        cudaMemcpy ( sumsHost, sums, numBlocks * sizeof ( int ),
            cudaMemcpyDeviceToHost );

        for ( int i = 0; i < numBlocks; i++ )
            res += sumsHost [i];

        delete [] sumsHost;
    }
    cudaFree ( sums );
    return res;
}
```

В таблице 5.1 приводится время, затраченное на вычисление поблочных сумм при помощи каждого из приведенных выше вариантов без учета времени копирования (Tesla C2070), а также с помощью CPU (Xeon E5620).

Легко видеть, что временные затраты на редуцирование массива составляют $O(\log 2N)$.

5.2. Префиксная сумма (scan)

Пусть задан массив $a_0, a_1, \ldots, a_{n-1}$ и некоторая бинарная ассоциативная операция. Без ограничения общности в качестве операции далее будет рассматривать-

Таблица 5.1. Время суммирования массива 8 млн целых чисел

Используемый вариант	**Время, мс**
reduce1	5,28
reduce2	2,52
reduce3	1,88
reduce4	0,99
reduce5	0,65
reduce serial	6,92
reduce OpenMP	2,34

ся сложение. Тогда *префиксной суммой* исходного массива будет называться следующий массив:

$$\{0, a_0, (a_0 + a_1), ((a_0 + a_1) + a_2), \ldots, a_0 + a_1 + a_2 \ldots + a_{n-2}\}$$

Первый элемент массива должен быть *нейтральным* по отношению к используемой бинарной операции (т.е. 0 в случае сложения, 1 – в случае умножения).

Префиксная сумма тривиально реализуема следующим последовательным кодом:

```
sum [0] = 0;
for (int i = 1; i < n; i++)
    sum [i] = sum [i-1] + a [i-1];
```

5.2.1. Реализация с помощью CUDA

Параллельная реализация префиксной суммы состоит из двух этапов по $\log 2n$ шагов: построение дерева сумм (аналогично редукции) и построение результирующего массива по дереву сумм (рис 5.5). Ниже приведен код первого этапа:

```
#define BLOCK_SIZE   256

__global__ void scan1 ( float * inData, float * outData, int n )
{
    __shared__ float temp [2*BLOCK_SIZE];

    int tid    = threadIdx.x;
```

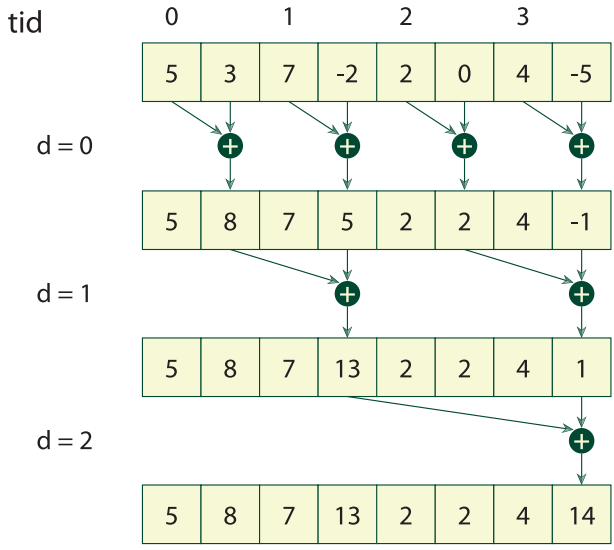


Рис. 5.5. Построение дерева сумм

```
int offset = 1;

// Загрузить данные.
temp [tid]            = inData [tid];
temp [tid+BLOCK_SIZE] = inData [tid + BLOCK_SIZE];

for ( int d = n >> 1; d > 0; d >>= 1 ) {
    __syncthreads ();

    if ( tid < d ) {
        int ai = offset * (2 * tid + 1) - 1;
        int bi = offset * (2 * tid + 2) - 1;

        temp [bi] += temp [ai];
    }

    offset <<= 1;
}
```

Следующая часть ядра по дереву частичных сумм строит результирующий массив. Этот процесс начинается с обнуления последнего элемента (содержащего сумму всех элементов), после чего идет обработка пар, начиная с пары элементов, удаленных на половину длины блока и заканчивая парами соседних элементов. На

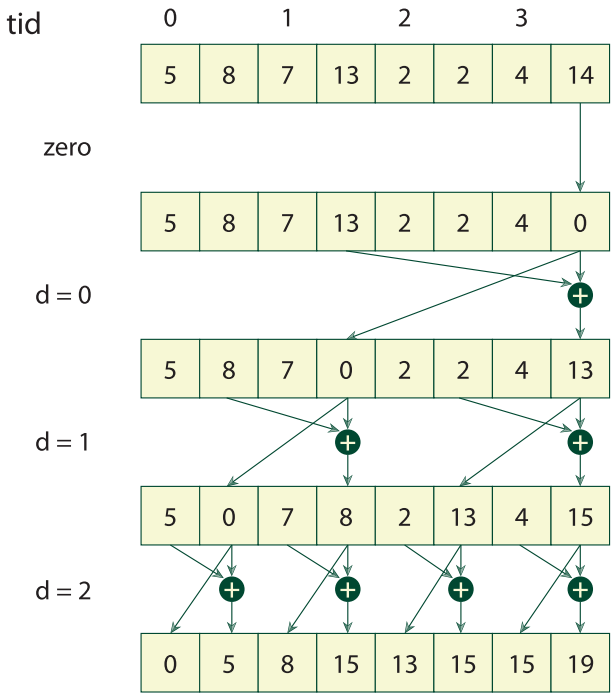


Рис. 5.6. Построение префиксных сумм по дереву сумм

каждом шаге один из элементов пары копируется на место второго, а на место первого записывается сумма исходных элементов (рис. 5.6). Ниже приведен код второго этапа:

```
// Очистить последний элемент.
if ( tid == 0 )
    temp [n-1] = 0;

for ( int d = 1; d < n; d <<= 1 )
{
    offset >>= 1;
    __syncthreads ();

    // Выполнить копирование и сложение.
    if ( tid < d )
    {
        int   ai = offset * (2 * tid + 1) - 1;
        int   bi = offset * (2 * tid + 2) - 1;
        float t  = temp [ai];
```

```
            temp [ai]  = temp [bi];
            temp [bi] += t;
        }
    }

    __syncthreads ();

    // Записать результат.
    outData [2 * tid]     = temp [2 * tid];
    outData [2 * tid + 1] = temp [2 * tid + 1];
}
#define kernel scan1
#include "main.h"
```

Эффективность данной реализации снижают постоянные конфликты банков вплоть до 16 порядка. Для их исключения в массив разделяемой памяти достаточно добавить по одному дополнительному элементу через каждые 16. Вычисление смещений производит макрос *CONFLICT_FREE_OFFS*:

```
// Размер блока.
#define BLOCK_SIZE 256

// Логарифм числа банков 16 по основанию 2.
#define LOG_NUM_BANKS 4

// Поправка для доступа к массиву в разделяемой памяти.
#define CONFLICT_FREE_OFFS(i) ((i) >> LOG_NUM_BANKS)

__global__ void scan2 ( float * inData, float * outData, int n )
{
    __shared__ float temp [2 * BLOCK_SIZE +
        CONFLICT_FREE_OFFS(2 * BLOCK_SIZE)];

    int tid = threadIdx.x;
    int offset = 1;
    int ai = tid;
    int b = tid + (n / 2);
    int offsA = CONFLICT_FREE_OFFS(ai);
    int offsB = CONFLICT_FREE_OFFS(bi);

    // Загрузить данные в разделяемую память.
    temp [ai + offsA] = inData [ai];
    temp [bi + offsB] = inData [bi];
```

```
// Вычисление дерева сумм.
for ( int d = n >> 1; d > 0; d >>= 1 )
{
    __syncthreads ();

    if ( tid < d )
    {
        int ai = offset * (2 * tid + 1) - 1;
        int bi = offset * (2 * tid + 2) - 1;

        ai        += CONFLICT_FREE_OFFS(ai);
        bi        += CONFLICT_FREE_OFFS(bi);
        temp [bi] += temp [ai];
    }

    offset <<= 1;
}

// Очистить последний элемент.
if ( tid == 0 )
    temp [n - 1 + CONFLICT_FREE_OFFS(n-1)] = 0;

for ( int d = 1; d < n; d <<= 1 )
{
    offset >>= 1;
    __syncthreads ();

    if ( tid < d )
    {
        int ai = offset * (2 * tid + 1) - 1;
        int bi = offset * (2 * tid + 2) - 1;
        float t;

        ai        += CONFLICT_FREE_OFFS(ai);
        bi        += CONFLICT_FREE_OFFS(bi);
        t          = temp [ai];
        temp [ai]  = temp [bi];
        temp [bi] += t;
    }
}

__syncthreads ();

// Сохранить результат.
outData [ai] = temp [ai + offsA];
outData [bi] = temp [bi + offsB];
```

```
}
#define kernel scan2
#include "main.h"
```

Рассмотренные CUDA-ядра будут работать только для небольших массивов, которые можно целиком поместить в разделяемую память. В общем случае исходный массив можно также разделить на части по 512 элементов и искать префиксную сумму для каждой части отдельно. Однако для получения окончательного результата необходимо выполнить дополнительный этап. Рассмотрим элементы на стыке первых двух блоков:

$$
\begin{aligned}
out_{510} &= a_0 + \ldots + a_{510}, \\
out_{511} &= a_0 + \ldots + a_{511}, \\
out_{512} &= 0, \\
out_{513} &= out_{512}, \\
out_{514} &= out_{512} + out_{513}.
\end{aligned}
$$

Очевидно, что к элементам, соответствующим второму блоку, нужно добавить сумму всех элементов первого блока. Точно так же к элементам третьего блока нужно добавить сумму элементов первых двух блоков (т.е. первых 1024 элементов входного массива) и т.д. Другими словами, если через $s_0, s_1, \ldots, s_{n/512-1}$ обозначить массив сумм элементов для каждого блока (обнуляемый элемент), то величины, на которые следует откорректировать каждый блок, будут являться результатом применения префиксной суммы к этому массиву. На основе данного свойства можно построить рекурсивный алгоритм для общего случая:

```
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>

// Ядро дополнительного этапа коррекции массива.
__global__ void scanDistribute ( float * data, float * sums )
{
    data [threadIdx.x + blockIdx.x * blockDim.x + blockDim.x] +=
        sums[blockIdx.x + 1];
}

// Функция scan для массива произвольной длины.
```

```
int scan(float * inData, float * outData, int n)
{
    int       numBlocks = n / (2 * BLOCK_SIZE);

    // Суммы элементов для каждого блока.
    float   * sums1      = NULL;

    // Результат scan'а этих сумм.
    float   * sums2     = NULL;

    if (numBlocks < 1)
        numBlocks = 1;

    // Выделяем память под массивы.
    cudaError_t cuerr = cudaMalloc (
        (void**)&sums1,  numBlocks * sizeof ( float ) );
    cuerr = cudaMalloc ( (void**)&sums2, numBlocks * sizeof ( float ) );

    // Осуществляем поблочный scan, одна нить на два элемента.
    dim3 threads ( BLOCK_SIZE, 1, 1 );
    dim3 blocks  ( numBlocks, 1, 1 );
    kernel<<<blocks, threads>>> ( inData, outData, sums1, 2 * BLOCK_SIZE );
    cuerr = cudaGetLastError();
    cuerr = cudaDeviceSynchronize();

    // Теперь выполняем scan для сумм.
    if ((n >= 2 * BLOCK_SIZE) && (numBlocks > 1))
    {
        if (scan(sums1, sums2, numBlocks))
            return 1;

        // Теперь корректируем результат.
        threads = dim3(2 * BLOCK_SIZE, 1, 1);
        blocks  = dim3(numBlocks - 1, 1, 1);
        scanDistribute<<<blocks,threads>>>(outData, sums2);
        cuerr = cudaGetLastError();
        cuerr = cudaDeviceSynchronize();
    }

    cudaFree(sums1);
    cudaFree(sums2);

    return 0;
}
```

5.2.2. Реализация с помощью CUDPP

Вместо того, чтобы искать префиксную сумму вручную, можно воспользоваться готовой реализацией из CUDA Performance Primitives (CUDPP) – библиотеки алгоритмов работы с массивами. В частности, в CUDPP входит и эффективная реализация нахождения префиксной суммы.

Библиотека CUDPP может быть собрана из исходного кода, доступного на сайте проекта:

```
wget http://cudpp.googlecode.com/files/cudpp_src_2.0.zip
unzip cudpp_src_2.0.zip
cd cudpp_src_2.0
mkdir build
```

Для компиляции потребуется система *cmake* и CUDA Toolkit, ниже приведены команды сборки для Linux:

```
cd build/
cmake .. -DBUILD_SHARED_LIBS=ON -DBUILD_APPLICATIONS=ON -DCUDA_VERBOSE_BUILD=↩
    ON -DCMAKE_BUILD_TYPE=Release
make -j6
```

CUDPP оперирует описателями (handles), позволяющими один раз задать параметры требуемой операции и затем многократно проводить ее, указывая только ранее созданный *handle*. Такой программный дизайн обычно выбирается в случаях, когда инициализация затратна, например, содержит внутренний бенчмаркинг оборудования для выбора наилучших параметров. При создании описателя передается информация о количестве обрабатываемых элементов (*numElements*), количестве строк (*numRows*, для двухмерных операций) и выравнивании строк (*rowPitch*).

```
CUDPPResult cudppPlan(
    CUDPPHandle * plan, CUDPPConfiguration config, size_t numElements,
    size_t numRows, size_t rowPitch);

CUDPPResult cudppDestroyPlan(CUDPPHandle plan);
```

Тип алгоритма, операции, входных данных и другие опции передаются полями структуры *CUDPPConfiguration*. В случае префиксной суммы – *algorithm* = *CUDPP_SCAN*, *op* = *CUDPP_ADD*, *datatype* = *CUDPP_FLOAT*. Через поле *options* могут быть также заданы тип преобразования (*CUDPP_OPTION_FORWARD*, *CUDPP_OPION_BACKWARD*) и флаг включения первого элемента для алгоритма

scan (*CUDPP_OPTION_EXCLUSIVE*, *CUDPP_OPTION_INCLUSIVE*). Префиксная сумма в CUDPP находится следующим вызовом:

```
CUDPPResult cudppScan(
    CUDPPHandle plan, void * outData, const void * inData,
    size_t numElements );
```

Следующая main-функция сравнивает результаты и производительность scan-ядер и реализации CUDPP:

```
#include <cudpp.h>
int main (int argc, char *  argv [])
{
    if (argc < 2)
    {
        printf("Usage: %s <len>\nlen must be power of 2\n", argv[0]);
        return 1;
    }
    int n = atoi(argv[1]);
    if (n % 2)
    {
        printf("Invalid len: must be power of 2\n");
        return 1;
    }
    int nb = n * sizeof ( float );

    // Выделить память на CPU.
    float* a = (float*)malloc(nb);
    float* b [2] = { NULL, NULL };
    b[0] = (float*)malloc(nb);
    b[1] = (float*)malloc(nb);

    // Заполнить входной массив случайным значениями.
    double dinvrandmax = 1.0 / RAND_MAX;
    for (int i = 0; i < n; i++ )
        a [i] = 2.0 * (rand() * dinvrandmax - 0.5);

    // Выделить память на GPU.
    float       * adev [2] = { NULL, NULL };
    cudaEvent_t   start, stop;
    float         gpuTime = 0.0f;

    cudaError_t cuerr = cudaMalloc ( (void**)&adev [0], nb );
    cuerr = cudaMalloc ( (void**)&adev [1], nb );

    // Создать события для замера времени.
```

```
cuerr = cudaEventCreate ( &start );
cuerr = cudaEventCreate ( &stop );

//
// Тест версии, написанной вручную.
//
cuerr = cudaMemcpy      ( adev [0], a, nb, cudaMemcpyHostToDevice );
cuerr = cudaEventRecord ( start, 0 );

// Сначала идет входной массив, затем выходной.
scan ( adev [0], adev [1], n );

cuerr = cudaEventRecord ( stop, 0 );
cuerr = cudaMemcpy      ( b[0], adev [1], nb, cudaMemcpyDeviceToHost );

cudaEventElapsedTime ( &gpuTime, start, stop );

// Вывести затраченное время.
printf ( "time spent scanning by hand: %.2f milliseconds\n",
    gpuTime );

//
// Тест версии на CUDPP.
//
cuerr = cudaMemcpy      ( adev [0], a, nb, cudaMemcpyHostToDevice );
cudaEventRecord ( start, 0 );

// Создать описатель библиотеки.
CUDPPHandle         cudppLib;
cudppCreate(&cudppLib);

// Создать план CUDPP.
CUDPPHandle         plan;
CUDPPConfiguration  config;
config.algorithm  = CUDPP_SCAN;
config.op         = CUDPP_ADD;
config.datatype   = CUDPP_FLOAT;
config.options    = CUDPP_OPTION_FORWARD | CUDPP_OPTION_EXCLUSIVE;
CUDPPResult pperr = cudppPlan ( cudppLib, &plan, config, n, 1, 0 );

// Найти префиксные суммы.
// Сначала идет выходной массив, затем входной.
pperr = cudppScan ( plan, adev [1], adev [0], n );

cuerr = cudaDeviceSynchronize();
```

```
    // Скопировать результат в память CPU.
    cuerr = cudaEventRecord ( stop, 0 );
    cuerr = cudaMemcpy      ( b[1], adev [1], nb, cudaMemcpyDeviceToHost );

    cudaEventElapsedTime ( &gpuTime, start, stop );
    printf ( "time spent, scanning using CUDPP: %.2f milliseconds\n",
        gpuTime );

    // Сравнить результаты scan и CUDPP scan.
    double maxdiff = 0;
    int imaxdiff = 0;
    for (int i = 0; i < n; i++)
    {
        double diff = b[0][i] / b[1][i];
        if (diff != diff) diff = 0; else diff = 1.0 - diff;
        if (diff > maxdiff)
        {
            maxdiff = diff;
            imaxdiff = i;
        }
    }
    if (n <= 1024)
    {
        for (int i = 0; i < n; i++)
            printf("%d\t%f\t%f\t%f\n", i, a[i], b[0][i], b[1][i]);
    }
    printf("results max diff = %f @ i = %d\n", maxdiff, imaxdiff);

    // Освобождаем выделенные ресурсы.
    cudaEventDestroy ( start );
    cudaEventDestroy ( stop  );
    cudaFree         ( adev [0] );
    cudaFree         ( adev [1] );

    free(a);
    free(b[0]); free(b[1]);

    return 0;
}
```

В таблице 5.2 приводится время, затраченное на вычисление префиксной суммы обоими вариантами алгоритма и с помощью готовой реализации из библиотеки CUDPP (Tesla C2070).

Таблица 5.2. Время вычисления префиксной суммы для массива 16 млн вещественных чисел

Используемый вариант	**Время, мс**
scan1	11,60
scan2	9,49
cudpp	2,87

Глава 6

Архитектура GPU

В данной главе рассматриваются сущствующие архитектуры GPU и общие приемы оптимизации программ.

6.1. Архитектура GPU

GPU архитектур G80, Tesla, Fermi и Kepler построены как масштабируемый массив потоковых мультипроцессоров (Streaming Multiprocessor, SM – для архитектур до Fermi включительно, SMX – для Kepler). Когда на GPU запускается CUDA-ядро, то блоки его сетки выполняются на доступных мультипроцессорах. При этом на каждый мультипроцессор целиком помещается до 8 блоков. По мере того, как отдельные блоки завершают свое выполнение, их место занимают новые блоки. Это позволяет даже на небольшом числе потоковых мультипроцессоров запустить на выполнение сетку с очень большим числом блоков. При этом планировщик блоков «скрыт» от программиста – ядро просто запускается на выполнение, все остальное делает устройство.

На рис. 6.1 и рис. 6.2 представлены архитектуры мультипроцессора GPU Tesla C870 и Tesla C1060. Каждый потоковый мультипроцессор содержит восемь *скалярных ядер* (SP, Scalar Processor), каждая нить выполняется на одном из этих ядер. Кроме скалярных ядер, потоковый мультипроцессор также содержит два блока для вычисления специальных функций (SFU, Special Function Unit), блок управления командами (Instruction Unit) и собственную память. В потоковых мультипроцессорах GPU Tesla C1060 дополнительно реализован специальный блок для обработки 64-битных чисел с плавающей точкой (Double Precision Unit).

Непосредственно в мультипроцессоре находится память следующих типов:

- набор 32-битных регистров (register file);
- разделяемая память, доступная всем скалярным ядрам;
- кэш константной памяти, доступный на чтение всем скалярным ядрам;
- кэш текстурной памяти, доступный на чтение всем скалярным ядрам.

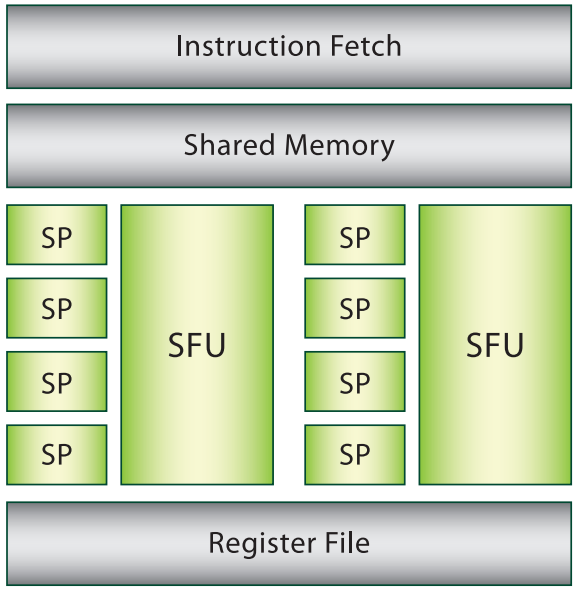


Рис. 6.1. Устройство потокового мультипроцессора для GPU Tesla C870/GeForce 8800

Для доступа к текстурной памяти предназначен специальный текстурный блок (TEX), совместно используемый сразу несколькими мультипроцессорами. Текстурный блок вместе со своей группой мультипроцессоров образует так называемый Texture Processing Cluster (TPC) (рис. 6.3).

На рис. 6.4 представлен мультипроцессор GPU Tesla M2090 архитектуры Fermi, который значительно увеличился в размерах по сравнению с предыдущими архитектурами. GPU Tesla M2090 содержит 16 таких мультипроцессоров (рис. 6.5). Каждый мультипроцессор имеет 32 скалярных ядра, четыре блока для вычисления специальных функций (SFU) и два текстурных блока (TEX). Кроме типов памяти, перечисленных для устройств с compute capability 1.x, в мультипроцессор архитектуры Fermi встроен кэш первого уровня, совмещенный с разделяемой памятью (см. секцию 3.2.1). Кэш-память второго уровня размером 768 Кбайт (L2-cache) является общей для всех мультипроцессоров.

Группу из четырех мультипроцессоров архитектуры Fermi принято называть GPC (Graphics Processing Cluster) вместо аббревиатуры TPC, использовавшейся в

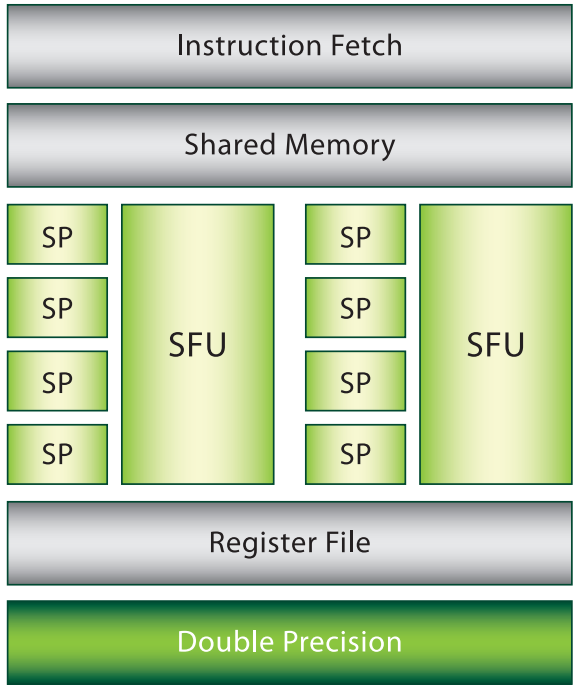


Рис. 6.2. Устройство потокового мультипроцессора для GPU Tesla C1060/GeForce GTX 260

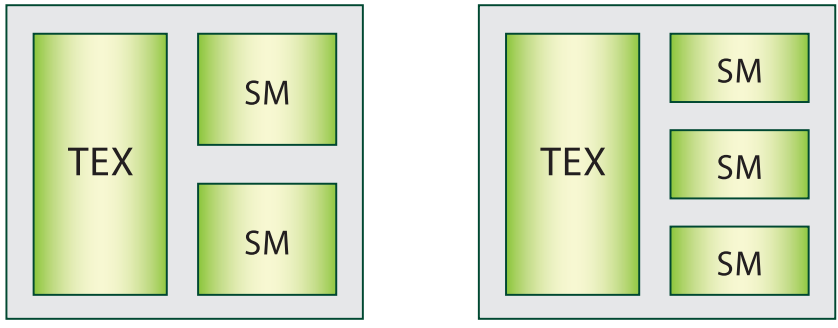


Рис. 6.3. Структура TPC для архитектур Tesla C860 (слева) и Tesla C1060 (справа)

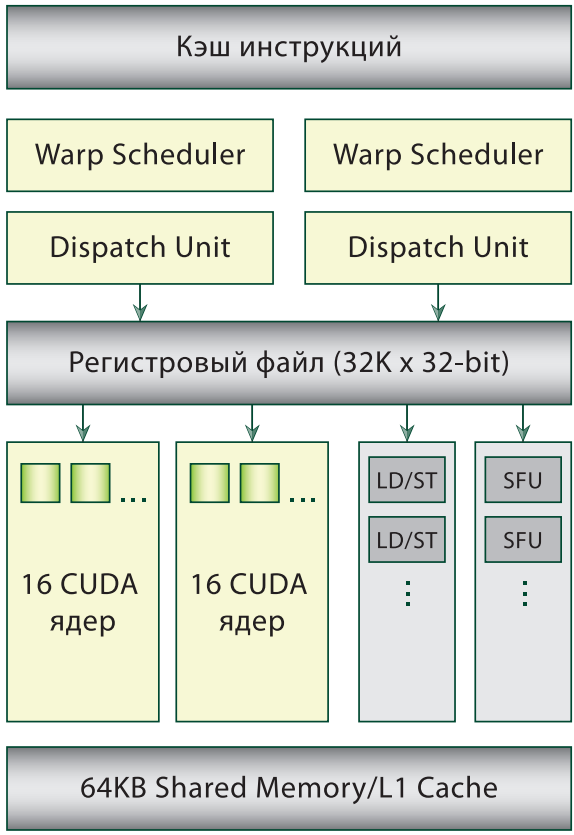


Рис. 6.4. Устройство потокового мультипроцессора для GPU Tesla M2090/GeForce GTX 580

предыдущих архитектурах. Например, GPU Tesla M2090 содержит 16 мультипроцессоров или 4 GPC (рис. 6.5).

На рис. 6.6 представлен мультипроцессор SMX архитектуры Kepler. GPU GTX 680 содержит 8 таких мультипроцессоров (рис. 6.6). Каждый мультипроцессор имеет 192 скалярных ядра, 32 блока для вычисления специальных функций (SFU) и 16 текстурных блоков (TEX). Кэш-память второго уровня имеет размер 512 Кбайт.

В архитектуре Kepler были упразднены несколько компонентов, использовавшихся для определения времени выполнения инструкций. Вместо этого латент-

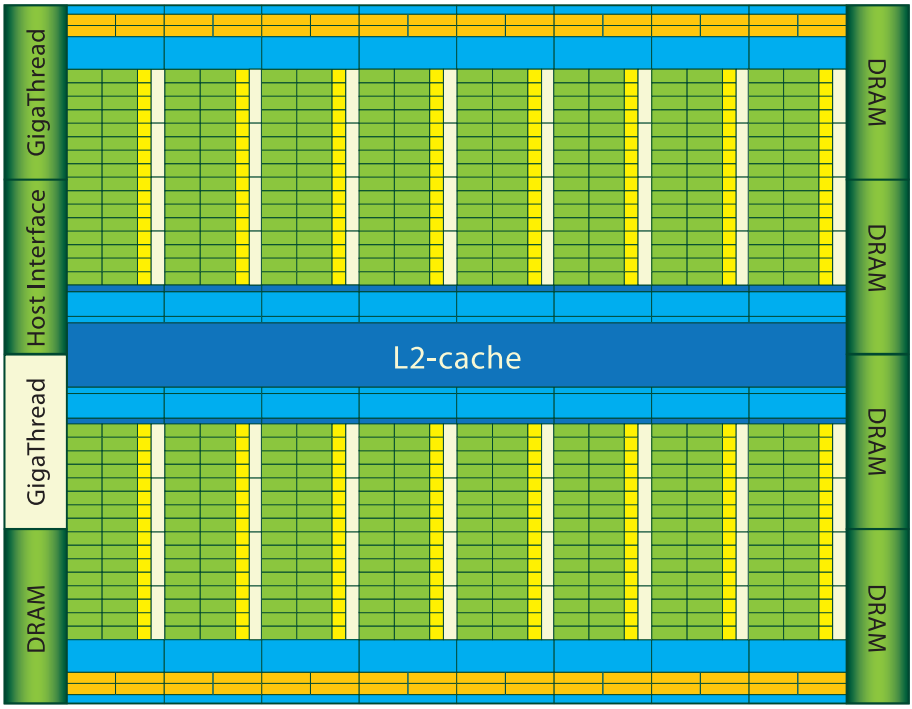


Рис. 6.5. Архитектура Fermi (Tesla M2090)

ности расчитываются на этапе компиляции и записываются в ассемблерный код (рис. 6.7). Скалярные ядра в Kepler работают на обычной частоте, в то время как в предыдущих архитетурах использовались ядра с удвоенной частотой. Это позволило снизить энергопотребление ядер в 2 раза при той же производительности.

При выполнении блока на потоковом мультипроцессоре все его нити перенумеровываются (порядок всегда фиксированный) и разбиваются на группы по 32 нити, каждая такая группа называется *варп* (warp). Нити в составе варпа выполняют физически параллельно одну и ту же команду (нити разных варпов могут выполнять разные команды). В таблице 6.1 указано, сколько требуется тактов одному варпу на выполнение одной арифметической операции на одном мультипроцессоре для устройств с compute capability 1.x и 2.0.

Если нити одного варпа должны идти по разным веткам кода (например, из-за условного оператора), то выполнятся все проходимые ветки. Даже если только

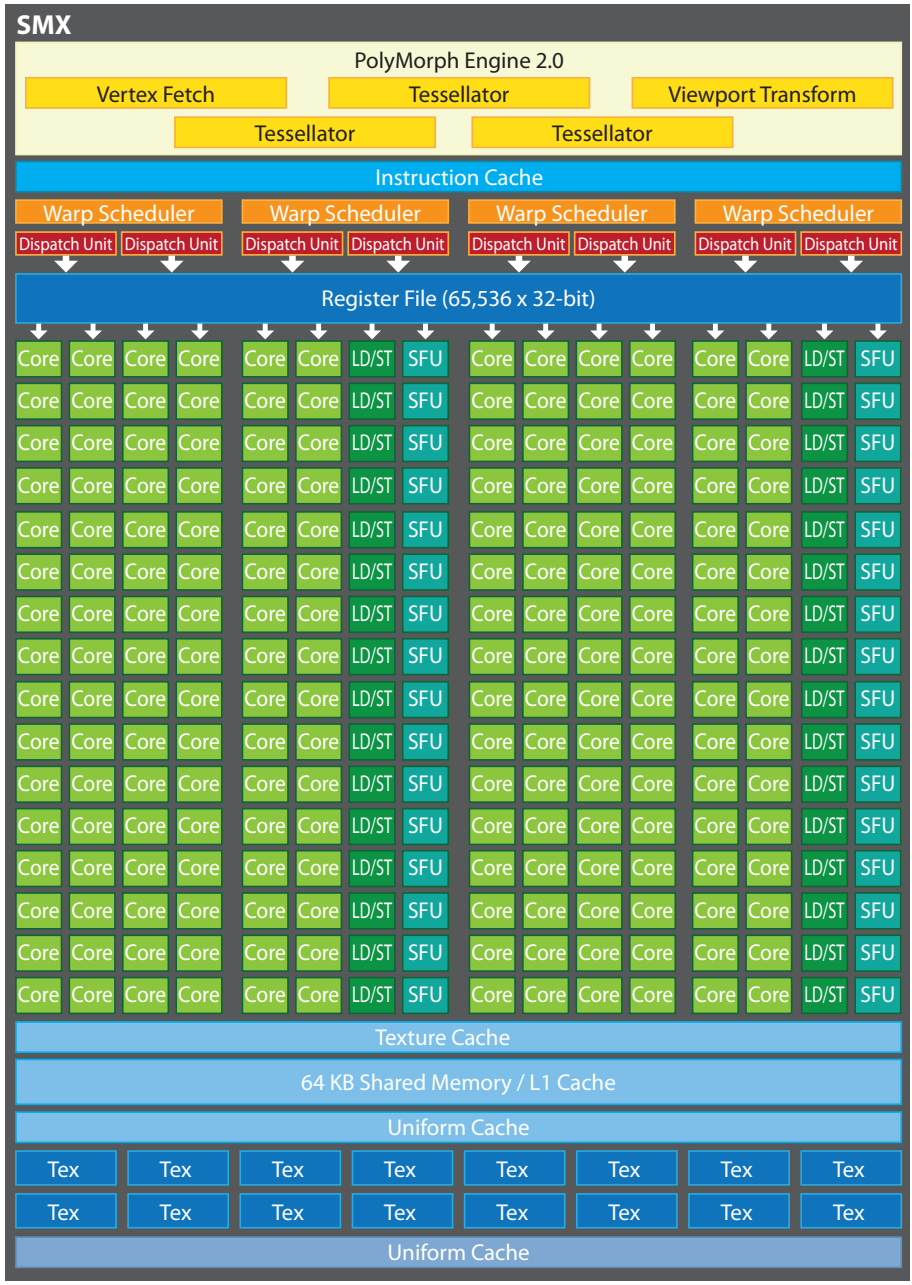


Рис. 6.6. Архитектура Kepler (GTX 680)

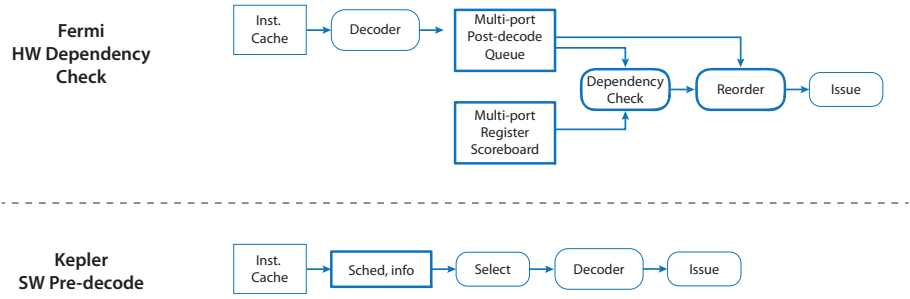


Рис. 6.7. Пример планирования работы SMX

Таблица 6.1. Количество тактов на мультипроцессор на варп

Операция	**Tesla C870/C1060**	**Tesla C2050**
32-битная вещественная add, multiply, multiply-add	4	1
64-битная вещественная add, multiply, multiply-add	32	2
32-битная целочисленная add, logical operation	4	1
32-битная целочисленная shift, compare	4	2
32-битная целочисленная multiply, multiply-add	4 (24-битная)	2
32-битная вещественная reciprocal, reciprocal square root	16	8

одна из 32 нитей пойдет по другой ветке кода, то все 32 нити должны будут выполнить обе ветки. Этот эффект называется *дивергентным ветвлением* (divergent branching) и ведет к снижению быстродействия. При этом нити, принадлежащие другим варпам, не оказывают никакого влияния на данный варп. Рассмотрим, как будет выполняться на GPU следующий фрагмент кода:

```
if ( threadIdx.x == 0 )
    x = cos ( y );
```

```
else
    x = sin ( y * 2.0f );
```

Сразу видно, что для первой нити варпа будет выполнена одна ветка, а для всех остальных нитей – другая, т.е. имеет место ветвление внутри варпа. В результате все нити данного варпа выполнят обе ветви ($x = cos(y)$ и $x = sin(y * 2.0f)$), что приведет к снижению быстродействия ядра. Чем больше ветвление внутри варпа, тем медленнее он выполняется из-за необходимости пройти все встречающиеся ветви кода. Знание размера варпа и того, каким образом происходит распределение нитей по варпам, позволяет в ряде случаев снизить ветвление.

Потоковый мультипроцессор для каждого варпа отслеживает готовность данных и на каждой очередной команде выбирает готовый к выполнению варп и выполняет для него одну команду. Для него определяется, какие данные необходимы для следующей команды, после чего выбирается очередной готовый к выполнению варп, для него выполняется одна команда и т.д.

Наличие большого количества выполняемых потоковым мультипроцессором варпов позволяет эффективно покрывать латентность доступа к памяти – пока одни варпы ожидают готовности данных, другие выполняются. Поскольку, например, для устройств compute capability 1.x размер варпа в четыре раза больше числа скалярных процессоров, то на выполнение одной команды всеми нитями варпа нужно четыре такта. Для того чтобы полностью покрыть латентность в 200 тактов, достаточно всего 50 варпов – команда обращения каждого варпа к глобальной памяти займет 4 такта (без учета латентности). Таким образом, когда 50-й варп выполнит свое обращение, данные для первого варпа уже будут готовы. Если обращение к глобальной памяти происходит лишь в каждой четвертой команде, то для полного покрытия латентности в 200 тактов достаточно всего 13 варпов.

Таким образом, потоковый мультипроцессор фактически осуществляет «бесплатное» управление выполнением варпов.

6.2. Общие методы оптимизации CUDA-программ

Оптимизация производительности, как правило, сводятся к следующим шагам:

1. Максимальное использование параллелизма задачи.

2. Оптимизация доступа в память.

3. Оптимизация математики.

Для достижения наилучшей производительности, следует попытаться разложить задачу на такие подзадачи, чтобы полностью использовать ресурс параллелизма по данным. В тех участках алгоритма, где параллелизм нарушается (например, потокам необходимо синхронизироваться для разделения данных между собой) возможны два случая:

1. Если потоки принадлежат одному блоку, то они могут использовать разделяемую память для эффективного обмена данными и *__syncthreads()* для синхронизации внутри одного ядра.

2. Если потоки принадлежат разным блокам, то эффективнее использовать 2 разных ядра с промежуточной записью в глобальную память.

При условии, что алгоритм достаточно распараллелен, можно приступить к оптимизации работы с памятью:

1. Оптимизация начинается с минимизации обмена данными между хостом и GPU. В этом могут помочь следующие приемы:

   (а) Помимо функций вида *cudaMemcpy*, возвращающих управление только по завершении копирования, существуют механизмы асинхронного копирования данных *cudaMemcpyAsync*. Асинхронное копирование работает с pinned-памятью (память, которую запрещено выгружать из ОЗУ). Для управления асинхронными вызовами можно применить *cudaEvents*.

   (б) Устройства с compute capability 1.1 и выше имеют отдельный DMA-контроллер, который позволяет копировать данные по PCI-E во время исполнения ядра.

   (в) CUDA stream позволяет задавать различные очереди задач. Например, пусть существует два независимых ядра:

   ```
   __global__ void ka(float *pDst, float *pSrc);
   ```

```
__global__ void kb(float *pDst, float *pSrc);
```

и их фактические аргументы:

```
float A[]; float B[];       // входные значения на CPU
float *pD_a; float *pD_b; // память на GPU для результата
float *pS_a; float *pS_b; // входные значения на GPU
```

и требуется выполнить две независимые цепочки операций:

$$(pS\_a) \rightarrow ka(pD\_a, pS\_a) \rightarrow (pD\_aA),$$
$$(BpS\_b) \rightarrow kb(pD\_b, pS\_b) \rightarrow (pD\_bB)$$

Эти цепочки необходимо поместить в разные очереди задач, так как это позволит производить копирование данных для ядра *B* параллельно с выполнением ядра *A* и одновременно копирование данных для *A* обратно на CPU будет происходить вместе с выполнением ядра *B*.

(г) Зачастую бывает эффективнее выполнить работу на GPU (даже если при этом прирост минимален, или ядро медленнее аналога на CPU), чем передавать данные по PCI-E и обратно.

2. Оптимизацию обращений в память из ядра необходимо начать с проверки выполнения правил коалесинга (были рассмотрены в гл. 3). Выполнение или невыполнение условий коалесинга может изменять производительность на порядок. Рассмотрим типичные ошибки:

   (а) При работе с двумерными и трехмерными типами данных массив оказывается не выровненным по ширине. Представим, что дана сетка *pDst* размера $1000 \times Height$ (значение *Height* не принципиально), каждый элемент которой есть вещественное число. В ядре CUDA вычисляется индекс для записи в pDst по формуле:

   ```
   tidx = threadIdx.x + blockIdx.x * blockDim.x;
   tidy = threadIdx.y + blockIdx.y * blockDim.y;
   pDst[tidx + tidy * 1000] = 0.0f.
   ```

   Такой простой код вызовет невыполнение условий коалесинга уже на второй строке (в данном случае нарушается правило о выравнивании

адреса по *16 * sizeof(*pDst)*), и производительность сразу упадет на порядок на архитектуре карт серии 8x и 9x.

(б) Использование невыровненных типов данных часто приводит к аналогичным проблемам. Например, если сравнить производительность двух ядер:

```
__global__
void kf3( float3 * pDst ){ pDst[threadIdx.x] = f3(); }

__global__
void kf4( float4 * pDst ){ pDst[threadIdx.x] = f4(); },
```

то окажется, что второе работает на порядок быстрее. Это связано с тем, что правила коалесинга работают с 32-битными, 64-битными и 128-битными структурами данных.

(в) Существуют задачи, в которых необходимо читать элементы по ширине, а записывать – по высоте. Такой задачей, например, является задача транспонирования матрицы. Прямолинейная запись по высоте сразу нарушает все правила коалесинга и вызывает падение производительности на любой архитектуре.

3. Задержка при доступе варпа к глобальной памяти занимает сотни тактов, поэтому необходимо, чтобы на мультипроцессоре исполнялось достаточное количество независимых от этой операции варпов для ее покрытия. Другими словами, во время ожидания одного варпа по инструкции доступа к памяти GPU должен иметь возможность исполнять другие независящие от этой инструкции. При этом нужно иметь в виду, что задержки, связанные с арифметикой, на порядок меньше, и одних этих операций зачастую может быть недостаточно. Таким образом, общим методом получения наилучшей пропускной способности памяти является покрытие задержек путем максимизации количества одновременно исполняемых на мультипроцессоре варпов.

4. При написании ядра важно следить за тем, использует ли компилятор локальную память. Использование локальной памяти происходит в следующих случаях:

(а) Если используется большое количество регистров. Компилятор ограничивает использование регистров (например, числом 32). Если компилятор не может вместить все вычисления в заданное количество регистров, то он выделяет массив глобальной памяти, в который при необходимости сбрасывает данные из регистров или восстанавливает их по мере надобности.

(б) Если в ядре объявить автоматический массив – переменную вида

```
float a[2];
```

и в дальнейшем обращаться к ней по индексу, который приходит из внешних данных (например, индекс можно прочитать из глобальной памяти или получить в результате вычислений, которые компилятор не сможет провести на этапе компиляции), то компилятор вынужден разместить данную переменную в локальной памяти, так как регистры не являются индексируемым видом памяти. В такой ситуации лучше использовать разделяемую память (если она не используется активно) или по возможности переосмыслить алгоритм.

(в) В CUDA нет способа обозначить на какую именно память ссылается передаваемый в __device__-функцию указатель, поэтому компилятор вынужден пытаться получить заключение о типе памяти на основании анализа контекста. В некоторых случаях передача указателя может привести к использованию локальной памяти.

(г) Некоторые математические функции могут приводить к использованию локальной памяти.

В устройствах архитектуры Fermi появилась кэш-память первого уровня, которая также используется для ускорения доступа к локальной памяти. В тех случаях, когда использование локальной памяти предотвратить нельзя, увеличение L1-кэша за счет разделяемой памяти (см. секцию 3.2.1) может улучшить производительность.

5. Разделяемая память требует правильного использования банков.

(а) Просто достигнуть максимальной производительности, если каждый поток обращается в свой банк или все потоки обращаются к одному элементу.

(б) Стоит избегать хаотичного обращения в разделяемую память.

(в) Не следует хранить в разделяемой памяти структуры, поскольку это приведет к банк-конфликтам. Например, в массиве структур float4 последовательные поля x, y, z, w займут 4 банка, что потенциально дает банк-конфликт четвертого порядка. От конфликтов можно избавиться, если добавить смещения:

```
__shared__ float a[512*4];
#define V_X  a[threadIdx.x+512*0]
#define V_Y  a[threadIdx.x+512*1]
#define V_Z  a[threadIdx.x+512*2]
#define V_W  a[threadIdx.x+512*3]
```

Однако, всегда стоит оценивать насколько это увеличит объем математики, ведь теперь на каждое обращение будет выполняться до 4-х MAD-инструкций.

6. При использовании текстурной памяти важно соблюдать локальность при обращении к ней. Это повышает эффективность текстурного кэша.

(а) Использование текстурной памяти в случае, если перекрытия по данным между соседними варпами нет, вряд ли позволит получить более высокую производительность, чем с использованием глобальной памяти. Это оправдано только в том случае, если выполнить условия коалесинга невозможно.

(б) Кроме того, не стоит забывать, что текстурный блок имеет свой конвейер (по отображению адресов, нормализации значений, фильтрации и проч.), а значит обладает большей латентностью по сравнению с прямым обращением в глобальную память.

(в) Использование текстуры и разделяемой памяти может как увеличить производительность, так и уменьшить. Для оценки возможного эффек-

та необходимо учесть роль различных факторов. Например, если попробовать ускорить таким образом алгоритм Non-Local Means, то:

- Загрузка из текстуры в разделяемую память возможна только при использовании *uchar4* элементов, если на *float4* не хватит памяти. Загрузка ничего не стоит по сравнению со временем вычислений.
- При чтении *uchar4* нормализацию значений придется проводить самостоятельно, что добавляет существенный объем инструкций.

7. Работая с константной памятью, важно помнить, что ее максимальная производительность достигается при обращении всеми потоками варпа по одному адресу.

Наконец, если выбрана самая эффективная параллельная реализация алгоритма и налажена оптимальная работа с памятью, то можно приступить к оптимизации арифметики. Необходимо иметь в виду следующее:

1. Разные операции имеют разную пропускную способность, соответственно, это нужно учитывать при анализе приложения. Например, операции *ADD* и *FMAD* – «быстрые» (на Fermi за 1 такт для всего варпа), тогда как операции *LOG* и *RCP* – в несколько раз медленнее. Также есть специальные встроенные функции, которые имеют более высокую пропускную способность, но меньшую точность, например, *__sinf* (точность на 2–3 бита меньше). Существуют также дополнительные опции компилятора, как, например, *-ftz=true*, *-prec-div=false*, *-prec-sqrt=false*. Поэтому можно выбирать между более точным и более быстрым вариантом, в зависимости от поставленных целей;

2. Пропускная способность GPU измеряется в IPC (число инструкций за один такт). Этот показатель доступен в CUDA profiler. Чем ближе он к пиковому значению, тем более эффективно работает устройство. Таким образом, можно определять имеет ли смысл оптимизировать инструкции. Стоит учитывать, что разные инструкции имеют разную пропускную способность и пик IPC нужно рассчитывать в зависимости от набора инструкций в приложении;

3. Одна из основных проблем, связанных с инструкциями, – это *сериализация*. В нормальном режиме каждая инструкция вызывается для всего варпа один

раз. Сериализация возникает, когда инструкция вызывается несколько раз, для каждой нити варпа – тем самым снижается производительность. Сериализация бывает 2 типов: разные пути исполнения и банк-конфликты.

- Любая условная операция (if, switch, do, for, while) может заметно повлиять на общую производительность, так как нити внутри варпа имеют дивергентные пути исполнения. Степень влияния можно оценить довольно просто. Для этого необходимо заменить все условные выражения так, чтобы они выполнялись или не выполнялись для всего варпа. Уменьшить число дивергентных варпов можно сгруппировав нити так, чтобы они шли по одному пути исполнения. При профилировании, счетчик “divergent branch” увеличивается на единицу при каждом ветвлении внутри варпа, и его следует сравнивать со счетчиком “branch”, показывающим общее количество ветвлений.
- Конфликты банков были рассмотрены в секциях 3.5.2 и 3.5.3. Простой способ проверить, насколько сильно они влияют на общую производительность – заменить все индексы при обращении к разделяемой памяти на threadIdx.x. Таким образом, гарантируется, что обращения будет без конфликтов банков, при этом, если основная логика программы не поменяется, можно оценить потенциальное ускорение. Для индикации конфликтов банков в устройствах архитектуры Fermi при профилировании используется счетчик “l1 shared bank conflict”. Если его величина сравнима со значениями счетчиков “shared load”, “shared store” и “instructions issued”, то можно ожидать влияния конфликтов банков на производительность. В секции 3.5.3 также был рассмотрен пример устранения конфликтов с помощью добавления к массиву в разделяемой памяти фиктивных данных.

Глава 7

Прикладные математические библиотеки

В основе методов решения большинства современных задач, требующих высокопроизводительных вычислений (HPC), лежит одна или несколько технологий из следующего списка:

1. Линейная алгебра для плотных матриц;
2. Линейная алгебра для разреженных матриц;
3. Операции над регулярными сетками;
4. Операции над нерегулярными сетками;
5. Спектральные методы;
6. Взаимодействие частиц;
7. Метод Монте-Карло.

Для того, чтобы GPU играли существенную роль в мире HPC, необходимы не только вычислительный потенциал и перспективная программная модель, но и готовая технологическая экзосистема. По этой причине были разработаны и включены в состав CUDA Toolkit следующие математические библиотеки:

- CUBLAS – линейная алгебра для плотных матриц (1);
- CUSPARSE – линейная алгебра для разреженных матриц (2);
- CUFFT – преобразования Фурье (5);
- CURAND – генераторы псевдо- и квазислучайных чисел (7).

В соответствии с лицензией, данные библиотеки могут быть включены в поставку сторонних приложений. Доступ к исходному коду могут получить участники программы зарегистрированных разработчиков.

Также существует множество качественных свободно распространяемых библиотек сторонних разработчиков, например:

- MAGMA – линейная алгебра для плотных матриц, открытый эквивалент библиотеки CUBLAS, расширенный дополнительными методами;
- CUSP – линейная алгебра и итерационные методы решения СЛАУ для разреженных матриц;
- OpenCurrent – динамика жидких сред.

7.1. CUBLAS

CUBLAS реализует программный интерфейс BLAS (Basic Linear Algebra Subprograms – основные операции линейной алгебры над векторами и матрицами) на CUDA для одного GPU.

В соответствии с классическим BLAS на языке Fortran, в CUBLAS многомерные массивы располагаются в памяти *по столбцам* (column-major order), индексирование ведется от 1. Функции BLAS разделены на три уровня (таблица 7.1).

Таблица 7.1. Уровни функций BLAS

Уровень	**Вычислительная сложность**	**Примеры функций**
1 (векторные операции)	$O(N)$	AXPY: y := alpha x + y, DOT: dot := (x, y)
2 (матрица-вектор)	$O(N^2)$	GEMV – умножение матрицы общего вида на вектор
3 (матрица-матрица)	$O(N^3)$	GEMM – умножение двух матриц общего вида

Имя любой функции CUBLAS образуется по правилу *cublas<T><func>*, где *T* – литера, определяющая тип данных (*S* – вещественный, одинарная точность, *D* – вещественный, двойная точность, *C* – комплексный, одинарная точность, *Z* – комплексный, двойная точность), *func* – 3–4 литеры классического имени BLAS-функции, например, *cublasDgemm*. Как и вызовы CUDA-ядер, функции CUBLAS являются *асинхронными*.

Начиная с версии CUDA 4.0, в библиотеке CUBLAS появился новый API “v2” (старый API по-прежнему поддерживается). В новом API каждый вызов CUBLAS-функции использует *дескриптор* (handle), связанный с текущим контекстом устройства и потоком исполнения (CUDA Stream). Такой дизайн призван упростить разработку приложений, использующих несколько потоков исполнения или несколько GPU. Все результаты вычислений возвращаются через указатели в аргументах. В новом API функции библиотеки возвращают статус ошибки типа *cublasStatus_t*, тогда как в старом текущий статус ошибки доступен через общую вспомогательную функцию *cublasGetError()*.

Типовая схема использования CUBLAS в приложении имеет вид:

1. Инициализировать дескриптор CUBLAS функцией *cublasCreate*;
2. Выделить необходимую память на GPU для матриц и векторов, загрузить входные данные;
3. Вызвать необходимую последовательность функций CUBLAS;
4. Выгрузить результаты вычислений из памяти GPU в память основной системы, освободить память;
5. Освободить дескриптор CUBLAS функцией *cublasDestroy*.

7.1.1. Пример: *matmul*

Для проверки результата и сравнения производительности в примерах *matmul* главы 3 используется функция *cublasSgemm* библиотеки CUBLAS:

```
#include <cublas_v2.h>
#include <stdio.h>
//...
// Создать дескриптор CUBLAS.
cublasHandle_t handle;
cublasStatus_t cberr = cublasCreate_v2(&handle);
if (cberr != CUBLAS_STATUS_SUCCESS)
{
    fprintf(stderr, "Cannot create cublas handle: %d\n", cberr);
    return 1;
}
```

```
// Выполнить умножение матриц cdev := adev * bdev на GPU.
float alpha = 1.0, beta = 0.0;
cberr = cublasSgemm_v2(
    handle, CUBLAS_OP_T, CUBLAS_OP_T, n, n, n,
    &alpha, adev, n, bdev, n, &beta, cdev, n);
if (cberr != CUBLAS_STATUS_SUCCESS)
{
    fprintf(stderr, "Error launching cublasSgemm_v2: %d\n", cberr);
    return 1;
}

// Ожидать завершения операции.
cuerr = cudaDeviceSynchronize();
if (cuerr != cudaSuccess)
{
    fprintf(stderr, "Cannot synchronize kernel: %s\n",
        cudaGetErrorString(cuerr));
    return 1;
}

// Удалить дексриптор CUBLAS.
cberr = cublasDestroy_v2(handle);
if (cberr != CUBLAS_STATUS_SUCCESS)
{
    fprintf(stderr, "Cannot destroy cublas handle: %d\n", cberr);
    return 1;
}
```

7.1.2. Пример: степенной метод

В вычислительной математике часто возникает необходимость оценить спектр (собственные значения) линейного оператора. Это связанно с тем, что многие априорные оценки ошибок алгоритмов опираются на оценки спектров. Более того, нередко подобные оценки в алгоритме приходится выполнять постоянно. Например, такая необходимость возникает при решении нелинейных гиперболических задач, где шаг по времени обратно пропорционален величине максимального по модулю собственного значения матрицы системы, которая меняется на каждом временном слое.

Рассмотрим матрицу A. Как известно, вектор a называется собственным вектором матрицы A, если выполнено равенство $Aa = \lambda a$. При этом λ называется

собственным числом [5]. Если матрица A имеет размеры $N \times N$, то у нее есть N собственных чисел (возможно, комплексных). При этом данные собственные числа не обязательно различны.

В общем случае, задача нахождения максимального по модулю собственного числа не очень проста. Поэтому мы введем ряд предположений, которые позволят использовать компактный алгоритм.

Предположим, что матрица A имеет N собственных числел и полную систему из собственных векторов. Это означает, что любой вектор можно разложить в линейную комбинацию собственных векторов:

$$x = C_1 a_1 + C_2 a_2 + ... + C_N a_N = \sum_{i=0}^{N} C_i a_i, \tag{7.1}$$

где $Aa_i = \lambda_i a_i, i = 1..N$ - собственные вектора. Также предположим, что максимальное по модулю собственное число единственно: $|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq ... \geq |\lambda_N|$. Данные предположения позволяют построить более простой и компактный алгоритм, чем для матрицы общего вида.

Умножим уравнение 7.1 на матрицу A слева:

$$Ax = \sum_{i=0}^{N} C_i A a_i = \sum_{i=0}^{N} C_i \lambda_i a_i. \tag{7.2}$$

Умножая k раз, получаем:

$$A^k x = \sum_{i=0}^{N} C_i A^k a_i = \sum_{i=0}^{N} C_i \lambda_i^k a_i, \tag{7.3}$$

где $A^k = \underbrace{A \cdot A \cdot ... \cdot A}_{k \text{ раз}}$. Преобразовав 7.3, получаем:

$$\begin{aligned} A^k x = \lambda_1^k \left(C_1 a_1 + C_2 \left(\frac{\lambda_2}{\lambda_1} \right)^k a_2 + \ldots + C_N \left(\frac{\lambda_N}{\lambda_1} \right)^k a_N \right) = \\ = \lambda_1^k \sum_{i=0}^{N} C_i \left(\frac{\lambda_i}{\lambda_1} \right)^k a_i. \end{aligned} \tag{7.4}$$

Поскольку λ_1 – самое большое по модулю число, то все члены суммы в 7.4, кроме первого, стремятся к нулю при $k \to \infty$. Следовательно, по сравнению с первым членом суммы остальные уменьшаются при росте k, и вектор $A^k x$ близок к собственному вектору $C_1 \lambda_1^k a_1$.

Таким образом, для нахождения максимального по модулю собственного значения может быть использован следующий алгоритм:

1. выбрать вектор начального приближения x_0;
2. построить следующее приближение по формулам: $y_{n+1} = Ax_n$; $x_{n+1} = \alpha_{n+1} y_{n+1}$, где α_{n+1} выбирается из условия $|x_{n+1}| = 1$;
3. продолжать процесс, пока не будет выполнено заданное условие сходимости, например:
$$\|x_{n+1} - x_n\| < \varepsilon;$$
4. последнее приближение $x_{n+1} = x$ является собственным вектором, а собственное значение вычисляется по формуле:
$$x^T Ax \approx \lambda_1 x^T x = \lambda_1,$$
поскольку $\|x\| = 1$.

Данный алгоритм называется *степенным методом* нахождения максимального собственного значения [6]. Можно показать, что степенной метод применим и при отсутствии предположения о полноте системы собственных векторов. Однако, для корректной работы алгоритма в любом случае требуется наличие единственного максимального по модую собственного числа. Согласно теореме Фробениуса – Перрона [7], для этого достаточно, чтобы матрица имела только положительные элементы: $A : a_{ij} > 0$.

Степенной метод легко реализовать с помощью прикладных библиотек CUDA: CUBLAS – для операций с матрицами и векторами и CURAND – для генерации случайной положительной матрицы и начального приближения.

```
#include <cuda.h>
#include <curand.h>
```

```
#include <cublas_v2.h>
#include <stdio.h>
#define CUDA_CALL(x) \
    do { cudaError_t err = x; if (( err ) != cudaSuccess ) { \
        printf ("Error \"%s\" at %s :%d \n" , cudaGetErrorString(err), \
        __FILE__ , __LINE__ ) ; return -1; \
    }} while (0);

#define CURAND_CALL(x) do { if (( x ) != CURAND_STATUS_SUCCESS ) { \
    printf ("Error at %s :%d \n" , __FILE__ , __LINE__ ) ; \
    return -1; }} while (0);

#define CUBLAS_CALL(x) do { if (( x ) != CUBLAS_STATUS_SUCCESS ) { \
    printf ("Error at %s :%d \n" , __FILE__ , __LINE__ ) ; \
    return -1; }} while (0);

// Функция вывода матрицы на экран
// ddata - указатель на матрицу, расположенную в памяти GPU
int PrintDenseMatrix(char *name, float *ddata, int n, int m)
{
    float *data;
    data = (float*) malloc(n*m*sizeof(float));
    CUDA_CALL(cudaMemcpy(data, ddata, n*m*sizeof(float),
        cudaMemcpyDeviceToHost));
    printf("Dense matrix %s:", name);
    int ln, lm;
    ln = (n > 10) ? 10 : n;
    lm = (m > 10) ? 10 : m;
    for (int i=0; i<ln; i++)
    {
        for (int j=0; j<lm; j++)
            printf("%.4f ", data[j*n + i]);
        if (m != lm)
            printf("... ");
        printf("\n");
    }
    if (ln != n)
        printf("...\n");
    free(data);
    return 0;
}
```

В памяти GPU необходимо выделить массивы для хранения матрицы и двух векторов. Матрица и один из векторов заполняются случайными числами, с помощью библиотеки CURAND:

```
// Функция генерации матрицы и начального условия
int Init (float **dA, float **dx, float **dy, int n)
{
   printf("Data generation: ");
   // Выделение памяти для A, x_0 и вектора y (необходим в алгоритме)
   CUDA_CALL(cudaMalloc((void**) dA, n*n*sizeof(float)));
   CUDA_CALL(cudaMalloc((void**) dx, n*sizeof(float)));
   CUDA_CALL(cudaMalloc((void**) dy, n*sizeof(float)));

   // Генерация матрицы и вектора
   curandGenerator_t gen;
   CURAND_CALL(curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT));
   CURAND_CALL(
   curandSetPseudoRandomGeneratorSeed(gen, (unsigned long long)time(NULL)));

   CURAND_CALL(curandGenerateUniform(gen, dA[0], n*n));
   CURAND_CALL(curandGenerateUniform(gen, dx[0], n));
   printf("Done\n");
   return 0;
}
```

Основная часть кода содержит цикл итераций степенного метода: вычисление нормы вектора, умножение вектора на число, сложение векторов (с коэффициентами), а также умножение матрицы на вектор. Время работы алгоритма измеряется с использованием *cudaEvent*:

```
// Функция, выполняющая итерации алгоритма
int CublasIterations (cublasHandle_t cublasHandle,
                      float *A, float *x, float *y, int n,
                      float eps, int DEBUG)
{
    // Используем cudaEvent для замеров времени
    cudaEvent_t start, stop;
    CUDA_CALL(cudaEventCreate(&start));
    CUDA_CALL(cudaEventCreate(&stop));
    float a;
    float norm;
    int loop = 0;
    // Приводим начальное приближени к виду norm2(x) = 1
    CUBLAS_CALL(cublasSnrm2(cublasHandle, n, x, 1, &norm));
    a = 1.0f / norm;
    CUBLAS_CALL(cublasSscal(cublasHandle, n, &a, x, 1));
    float alpha = 1.0f;
    float beta = 0.0f;
    float diff = 100.0f;
```

```
    CUDA_CALL(cudaEventRecord(start, 0));
    while ((diff > eps) && (loop < 1000))
    {
        loop++;
        CUBLAS_CALL(cublasSgemv(cublasHandle, CUBLAS_OP_N,
            n, n, &alpha, A, n, x, 1, &beta, y, 1));
        CUBLAS_CALL(cublasSnrm2(cublasHandle, n, y, 1, &norm));
        a = 1.0f / norm;
        CUBLAS_CALL(cublasSscal(cublasHandle, n, &a, y, 1));

        // На данном этапе в y находится (n+1) итерация процесса,
        // а в x - n-ая
        // Вычислим норму разности приближений
        // x[i] = x[i] - y[i]:
        a = -1.0f;
        CUBLAS_CALL(cublasSaxpy(cublasHandle, n, &a, y, 1, x, 1));
        CUBLAS_CALL(cublasSnrm2(cublasHandle, n, x, 1, &diff));
        if (DEBUG)
        {
            printf("%d-th iteration. Difference norm %f\n", loop, diff);
        }
        // Копируем (n+1)-ую итерацию в x
        CUBLAS_CALL(cublasScopy(cublasHandle, n, y, 1, x, 1));
    }
    CUDA_CALL(cudaEventRecord(stop, 0));
    CUDA_CALL(cudaEventSynchronize(stop));

    if (loop < 1000)
    {
        printf("Process converged in %d iterations.\n", loop);
        printf("Residuals norm is %f\n", diff);
    }
    else
    {
        printf("Process didn't converge in 1000 iterations.\n");
        printf("Residuals norm is %f\n", diff);
    }

    // Вычисление затраченного времени на итерации
    float elapsedTime;
    CUDA_CALL(cudaEventElapsedTime(&elapsedTime, start, stop));
    CUDA_CALL(cudaEventDestroy(start));
    CUDA_CALL(cudaEventDestroy(stop));
    printf("Elapsed time: %.3f ms\n", elapsedTime);
    return 0;
}
```

Несмотря на то, что алгоритм должен сходиться, в программе установлено дополнительное ограничение на число итераций для защиты от зацикливания, в случае если ошибка округления превысит ε.

В последней части кода определена функция, вычисляющая собственное число, соответствующее найденному собственному вектору x по формуле $\lambda \approx \frac{x^T Ax}{x^t x}$:

```
// Функция, вычисляющая по приближенному значению собственного векотора
// собственное значение
int EigValEstimate(cublasHandle_t cublasHandle,
                   float *A, float *x, float *y, int n, int DEBUG)
{
    // y = Ax
    float alpha = 1.0f;
    float beta = 0.0f;
    CUBLAS_CALL(cublasSgemv(cublasHandle, CUBLAS_OP_N, n, n,
        &alpha, A, n, x, 1, &beta, y, 1));
    float num, denom;
    CUBLAS_CALL(cublasSdot(cublasHandle, n, x, 1, y, 1, &num));
    CUBLAS_CALL(cublasSdot(cublasHandle, n, x, 1, x, 1, &denom));
    printf("Approximate eigenvalue is %f", num/denom);
    PrintDenseMatrix("eigenvector", x, n, 1);
    return 0;
}

int main (int argc, char** argv)
{
    float *dA, *dx, *dy;

    // Обработка входных параметров
    if (argc != 4)
    {
        printf("Error. Eig usage: ./eig n, eps, DEBUG\n");
        return -1;
    }
    int n = atoi(argv[1]);
    float eps = atof(argv[2]);
    int DEBUG = atoi(argv[3]);
    int res = 0;

    // Вызов функции инициализации
    res = Init(&dA, &dx, &dy, n);
    if (res < 0)
        return res;

    // Вывод матрицы
```

```
    res = PrintDenseMatrix("A", dA, n, n);
    if (res < 0)
        return res;

    // Вывод вектора
    res = PrintDenseMatrix("x_0", dx, n, 1);
    if (res < 0)
        return res;
    printf("\n");

    // Создание контекста CUBLAS
    cublasHandle_t cublasHandle;
    CUBLAS_CALL(cublasCreate(&cublasHandle));

    // Вызов функции, реализующей алгоритм
    res = CublasIterations(cublasHandle, dA, dx, dy, n, eps, DEBUG);
    if (res < 0)
        return res;

    // Оценка собственного значения
    res = EigValEstimate(cublasHandle, dA, dx, dy, n, DEBUG);
    if (res < 0)
        return res;

    // Освобождение дескриптора
    CUBLAS_CALL(cublasDestroy(cublasHandle));

    return 0;
}
```

7.2. CUSPARSE

CUSPARSE реализует основные операции линейной алгебры для разреженных векторов и матриц (sparse matrices and vectors). Функции библиотеки имеют интерфейс для C и C++, поддерживается индексация элементов как с нуля, так и с единицы.

Разреженными называются матрицы или векторы с преимущественно[1] нулевыми элементами. Принцип, в соответствии с которым хранится разреженный вектор или матрица, прост: хранить только ненулевые значения и информацию об их

[1] Есть разные понимания «преимущественно». Например: «имеет малый процент ненулевых элементов», или «для матрицы $N \times N$ количество ненулевых элементов есть $O(N)$».

положении в матрице. Разреженный вектор представляется парой массивов. В первом массиве находятся все ненулевые значения из соответствующего плотного массива. Второй целочисленный массив содержит индексы этих элементов. Существует множество форматов для хранения матриц; среди них поддерживаются только:

1. Плотный формат (Dense format);
2. Координатный формат (Coordinate format, COO);
3. Строчный разреженный формат (Compressed Sparse Row Format, CSR);
4. Столбцовый разреженный формат (Compressed Sparse Column Format, CSC).

Все функции библиотеки реализованы для вещественных и комплексных типов одинарной и двойной точности и делятся на 4 группы:

1. Операции над разреженными векторами и плотными векторами: сложение векторов, скалярное произведение, поворот Гивенса, сборка (gather) и разборка (scatter) элементов вектора.
2. Операции над разреженными матрицами и плотными векторами:
   - *csrmv*: $y = \alpha op(A) \cdot x + \beta \cdot y$, где $op(A) = A$, либо $op(A) = A^T$, либо $op(A) = A^H$. A – матрица размером $m \times n$ в формате упакованных разреженных строк (CSR). α и β – скаляры. x и y – вектора в плотном формате.
   - *csrsv_analysis* – выполняет анализ при решении разреженной треугольной системы: $op(A) \cdot y = \alpha \cdot x$, где $op(A) = A$, либо $op(A) = A^T$, либо $op(A) = A^H$. A – матрица размером $m \times n$ в формате упакованных разреженных строк (CSR).
   - *csrsv_solve* – выполняет решение разреженной треугольной системы, описанной в предыдущем примере.
3. Операции над разреженными векторами и множествами плотных векторов:

$$C = \alpha \cdot op(A) \cdot B + \beta \cdot C,$$

где $op(A) = A$, либо $op(A) = A^T$, либо $op(A) = A^H$. α и β – скаляры. B и C – матрицы, хранящиеся в памяти *по столбцам* (column-major order). A – матрица размером $m \times k$ в формате упакованных разреженных строк (CSR).

4. Функции для преобразования одного формата матриц в другой.

Имя любой функции CUSPARSE образуется по правилу *cusparse<T><func>*, где *T* – литера, определяющая тип данных (*S* – вещественный, одинарная точность, *D* – вещественный, двойная точность, *C* – комплексный, одинарная точность, *Z* – комплексный, двойная точность), *func* – литеры имени функции; например *cusparseScsrmv*.

Типовая схема использования CUSPARSE в приложении выглядит следующим образом:

1. Выделить память на GPU. Инициализировать память данными для вектора или матрицы в одном из поддерживаемых форматов.
2. Инициализировать библиотеку.
3. Выполнить преобразования над данными.
4. Освободить память на GPU и само преобразование.

Функции CUSPARSE можно вызывать из программ на языке Fortran.

7.2.1. Пример: решение треугольной линейной системы уравнений

Треугольная матрица – квадратная матрица, в которой все элементы ниже или выше главной диагонали равны нулю. *Треугольная линейная система уравнений* – линейная система уравнений $Ax = b$, где A – треугольная матрица.

Ключевые шаги программы:

1. Создание разреженной матрицы в формате COO:

```
/* |1.0            |
   |2.0 3.0        |
   |4.0 5.0 6.0    |
   |7.0 8.0 9.0 1.0| */
```

```
int n=4, nnz=10;
cooRowIndexHostPtr = (int *)   malloc(
    nnz*sizeof(cooRowIndexHostPtr[0]));
cooColIndexHostPtr = (int *)   malloc(
    nnz*sizeof(cooColIndexHostPtr[0]));
cooValHostPtr      = (double *)malloc(
    nnz*sizeof(cooValHostPtr[0]));

cooRowIndexHostPtr[0]=0; cooColIndexHostPtr[0]=0;
cooValHostPtr[0]=1.0;

cooRowIndexHostPtr[1]=1; cooColIndexHostPtr[1]=0;
cooValHostPtr[1]=2.0;
cooRowIndexHostPtr[2]=1; cooColIndexHostPtr[2]=1;
cooValHostPtr[2]=3.0;

cooRowIndexHostPtr[3]=2; cooColIndexHostPtr[3]=0;
cooValHostPtr[3]=4.0;
cooRowIndexHostPtr[4]=2; cooColIndexHostPtr[4]=1;
cooValHostPtr[4]=5.0;
cooRowIndexHostPtr[5]=2; cooColIndexHostPtr[5]=2;
cooValHostPtr[5]=6.0;

cooRowIndexHostPtr[6]=3; cooColIndexHostPtr[6]=0;
cooValHostPtr[6]=7.0;
cooRowIndexHostPtr[7]=3; cooColIndexHostPtr[7]=1;
cooValHostPtr[7]=8.0;
cooRowIndexHostPtr[8]=3; cooColIndexHostPtr[8]=2;
cooValHostPtr[8]=9.0;
cooRowIndexHostPtr[9]=3; cooColIndexHostPtr[9]=3;
cooValHostPtr[9]=1.0;
```

2. Создание плотного вектора:

```
/* y   = [10.0 20.0 30.0 40.0 | 0.0 0.0 0.0 0.0] */
yHostPtr    = (double *)malloc(2*n       *sizeof(yHostPtr[0]));

yHostPtr[0] = 10.0;
yHostPtr[1] = 20.0;
yHostPtr[2] = 30.0;
yHostPtr[3] = 40.0;
yHostPtr[4] = 0.0;
yHostPtr[5] = 0.0;
yHostPtr[6] = 0.0;
yHostPtr[7] = 0.0;
```

3. Копирование вектора и матрицы в память на GPU:

```
cudaStat1 = cudaMalloc((void**)&cooRowIndex,
                       nnz*sizeof(cooRowIndex[0]));
cudaStat2 = cudaMalloc((void**)&cooColIndex,
                       nnz*sizeof(cooColIndex[0]));
cudaStat3 = cudaMalloc((void**)&cooVal,
                       nnz*sizeof(cooVal[0]));
cudaStat4 = cudaMalloc((void**)&y,
                       2*n*sizeof(y[0]));

cudaStat1 = cudaMemcpy(cooRowIndex, cooRowIndexHostPtr,
                       (size_t)(nnz*sizeof(cooRowIndex[0])),
                       cudaMemcpyHostToDevice);
cudaStat2 = cudaMemcpy(cooColIndex, cooColIndexHostPtr,
                       (size_t)(nnz*sizeof(cooColIndex[0])),
                       cudaMemcpyHostToDevice);
cudaStat3 = cudaMemcpy(cooVal,      cooValHostPtr,
                       (size_t)(nnz*sizeof(cooVal[0])),
                       cudaMemcpyHostToDevice);
cudaStat4 = cudaMemcpy(y,           yHostPtr,
                       (size_t)(2*n*sizeof(y[0])),
                       cudaMemcpyHostToDevice);
```

4. Инициализация CUSPARSE:

```
status = cusparseCreate(&handle);
```

5. Создание дескриптора матрицы (где указывается необходимое свойство *CUSPARSE_MATRIX_TYPE_TRIANGULAR*):

```
status = cusparseCreateMatDescr(&descra);
cusparseSetMatType(descra,CUSPARSE_MATRIX_TYPE_TRIANGULAR);
cusparseSetMatIndexBase(descra,CUSPARSE_INDEX_BASE_ZERO);
```

6. Преобразование матрицы из формата COO в CSR:

```
cudaStat1 = cudaMalloc((void**)&csrRowPtr,
                       (n+1)*sizeof(csrRowPtr[0]));
status= cusparseXcoo2csr(handle, cooRowIndex, nnz, n,
                         csrRowPtr, CUSPARSE_INDEX_BASE_ZERO);
```

7. Анализ и решение треугольной системы уравнений:

```
cusparseSolveAnalysisInfo_t info;
status = cusparseCreateSolveAnalysisInfo(&info);
status = cusparseDcsrsv_analysis(
    handle, CUSPARSE_OPERATION_NON_TRANSPOSE, n,
    descra, cooVal, csrRowPtr, cooColIndex, info);
status = cusparseDcsrsv_solve(
    handle, CUSPARSE_OPERATION_NON_TRANSPOSE, n, 7.0,
    descra, cooVal, csrRowPtr, cooColIndex, info, &y[0], &y[n]);
status = cudaMemcpy(yHostPtr, y, (size_t)(2*n*sizeof(y[0])),
                    cudaMemcpyDeviceToHost);
cusparseDestroySolveAnalysisInfo(info);
```

8. Проверка результатов:

```
if (FLOATS_EQ(yHostPtr[0], 10.0) &&
    FLOATS_EQ(yHostPtr[1], 20.0) &&
    FLOATS_EQ(yHostPtr[2], 30.0) &&
    FLOATS_EQ(yHostPtr[3], 40.0) &&
    FLOATS_EQ(yHostPtr[4], 70.0) &&
    FLOATS_EQ(yHostPtr[5], 0.0) &&
    FLOATS_EQ(yHostPtr[6], -11.6667) &&
    FLOATS_EQ(yHostPtr[7], -105.0))
{
    CLEANUP("example test PASSED");
    return EXIT_SUCCESS;
} else {
    CLEANUP("example test FAILED");
    return EXIT_FAILURE;
}
```

В качестве упражнения читатель может переписать код для проверки результатов: умножить исходную разреженную матрицу A на плотный вектор решения x и убедиться, что результат не отличается от исходного плотного вектора b в правой части.

7.3. CUFFT

Библиотека CUFFT реализует прямое и обратное быстрое дискретное преобразование Фурье (ДПФ):

$$F(x) = \sum_{n=0}^{N-1} f(n) e^{-j2\pi\left(x\frac{n}{N}\right)}$$

$$f(n) = \frac{1}{N}\sum_{n=0}^{N-1} F(x)e^{j2\pi\left(x\frac{n}{N}\right)}$$

Простейшая реализация ДПФ в виде произведения матрицы на вектор будет иметь алгоритмическую сложность $O(N^2)$. Для уменьшения сложности используется алгоритм Кули-Тьюки (Cooley-Tukey). (В популярной open-source библиотеке FFTW сложность используемых алгоритмов – $O(NlogN)$). Параллельная реализация преобразований основана на принципе «разделяй и властвуй» (divide and conquer)[8]. CUFFT оптимизирован для преобразований, размерности которых выражаются как $2^a \cdot 3^b \cdot 5^c \cdot 7^d$. Для преобразований других размеров используется алгоритм Блуштейна (Bluestein), реализованный на основе алгоритма Кули-Тьюки [9].

Интерфейс и формат данных CUFFT во многом схож с FFTW, но отличается в некоторых деталях. Для исключения отличий в CUFFT предусмотрен режим совместимости (FFTW Compatibility Mode). Ниже перечислены основные свойства реализации CUFFT:

- одно-, дву- и трехмерные вещественные и комплексные преобразования;
- одинарная и двойная точность;
- одномерное преобразование до 128 млн элементов одинарной точности и до 64 млн элементов двойной точности (точное ограничение на конкретном GPU определяется размером доступной глобальной памяти);
- поддержка CUDA Streams (Streamed CUFFT Transforms);
- in-place и out-of-place преобразования для вещественных и комплексных данных;
- преобразования с двойной точностью можно выполнить только на GPU с поддержкой двойной точности (GT200 и более поздние версии);
- ненормализованный вывод: $IFFT(FFT(A)) = len(A) * A$ (результат последовательного применения прямого и обратного преобразования – исходный вектор, умноженный на длину).

Для того, чтобы использовать библиотеку, необходимо включить заголовочный файл “cufft.h”. Библиотека состоит из объявлений, типов данных и функции обработки данных этих типов.

Основные типы данных библиотеки CUFFT:

1. *cufftHandle* – план CUFFT (аналог *fftw_plan*), используется для адаптивного выбора наилучшего алгоритма и многократного использования известной оптимальной настройки;
2. *cufftResult* – результат вызова функции;
3. *cufftType* – тип преобразования (поддерживаются комплексные и вещественные преобразования).

Основные функции библиотеки CUFFT:

1. *cufftPlan** – функции создания планов, принимают на вход дескриптор плана, размерности и тип преобразования; *cufftDestroy* – освобождает план;
2. *cufftExec** – функции для выполнения преобразования, принимают на вход дескриптор плана, входные и выходные данные, возвращают *cufftResult*;
3. *cufftSetStream* связывает CUDA stream с CUFFT-планом. На вход принимает план и stream, а возвращает *cufftResult*.

FFTW реализует «расширенный интерфейс» (advanced interface), который дает возможность преобразовать несколько массивов одновременно. Его аналогом в CUFFT является функция *cufftPlanMany*.

Типовая схема использования CUFFT в приложении выглядит следующим образом:

1. Выделить память на GPU;
2. Создать и настроить преобразование (размер, тип, …) и план;
3. Выполнить преобразование необходимое число раз, используя план и данные;

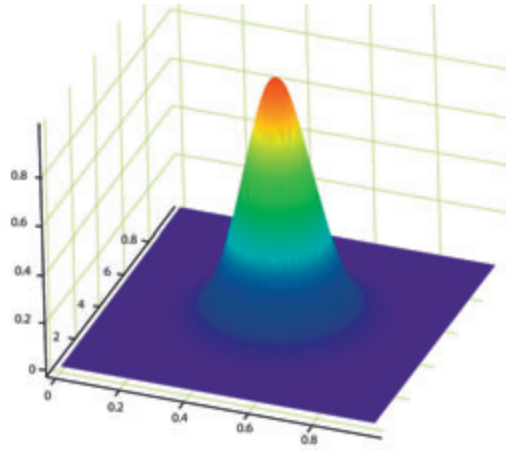


Рис. 7.1. Точное решение уравнения Пуассона

4. Освободить память на GPU и преобразование.

CUFFT может быть также использован в программах на языке Fortran.

7.3.1. Пример: решение уравнение Пуассона

Уравнение Пуассона описывает многие явления в физике. Оно встречается в задачах магнитостатики, электростатики, гидро- и газодинамики, описывает стационарное поле температуры. Поставим задачу с уравнением Пуассона следующего вида:

$$\begin{cases} \Delta u(p) = \rho(p), p \in \Omega = (x, y), 0 \leq x, y \leq 1, \\ \rho(x, y) = \frac{s(x,y)-2\sigma^2}{\sigma^4} \exp^{-\frac{s(x,y)}{2\sigma^2}} . \end{cases} \tag{7.5}$$

Пусть на границе области Ω заданы периодические граничные условия, тогда уравнение имеет точное решение (рис. 7.1):

$$u_0(x, y) = \exp^{\frac{s(x,y)}{2\sigma^2}} . \tag{7.6}$$

Решение уравнение Пуассона можно получить с помощью преобразования Фурье. Пусть N – число разбиений отрезка $[0..1]$, $h = \frac{1}{N}$ – шаг сетки одинаковый по обоим осям X и Y. Тогда

$$\overline{\rho} = \frac{1}{N^2} \sum_{j=0}^{N-1} \sum_{k=0}^{N-1} \rho(x_k, y_j) W^{nk+mj}, \tag{7.7}$$

где $W = \exp^{i\frac{2\pi}{N}}$. Можно убедиться, что при незначительных допущениях:

$$\overline{u}(n,m) = \overline{\rho}(n,m) h^2 (W^{-n} + W^n + W^{-m} + W^m - 4)^{-1}), \tag{7.8}$$

$$u(x_k, y_j) = \sum_{j=0}^{N-1} \sum_{k=0}^{N-1} \overline{u}(n,m) W^{nk+mj}. \tag{7.9}$$

В данном примере прямое (7.7) и обратное преобразование (7.9) Фурье будет выполнено с помощью библиотек FFTW и CUFFT, а вычисления в Фурье-пространстве (7.8) – с помощью отдельного CUDA-ядра.

Сначала решим задачу с помощью FFTW. Читатель может сравнить эту реализацию с аналогичной для CUFFT и убедиться в схожести интерфейсов двух библиотек.

```
double s(double x, double y)
{
    return (x - 0.5) * (x - 0.5) + (y - 0.5) * (y - 0.5);
}

double rho(double x, double y)
{
    double ss = s(x, y);
    return (ss - 2 * sigma2) * exp(-ss / (2 * sigma2)) / sigma4;
}

double wave_num2(int i, int j, int n)
{
    if (!i && !j) return 1.0;

    double wn1 = i < n / 2 ? i : i - n;
    double wn2 = j < n / 2 ? j : j - n;

    return -4 * M_PI * M_PI * (wn1 * wn1 + wn2 * wn2);
}

int fft_cpu(int n, double* u)
{
```

```
    const double h = 1.0 / n;

    fftw_complex* v = (fftw_complex*)fftw_malloc(
        sizeof(fftw_complex) * (n / 2 + 1) * n);

    fftw_plan forward = fftw_plan_dft_r2c_2d(n, n, u, v, FFTW_ESTIMATE);
    fftw_plan inverse = fftw_plan_dft_c2r_2d(n, n, v, u, FFTW_ESTIMATE);

    fftw_execute(forward);

    for (int j = 0; j < n; j++)
        for (int i = 0; i < (n / 2 + 1); i++)
            v[j * (n / 2 + 1) + i][0] /= wave_num2(i, j, n);

    fftw_execute(inverse);

    double shift = u[0];
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            u[j * n + i] -= shift;
            u[j * n + i] /= n * n;
        }
    }

    fftw_destroy_plan(forward);
    fftw_destroy_plan(inverse);

    fftw_free(v);

    return 0;
}
```

Массив u в памяти CPU содержит значения функции ρ в точках сетки. Память для массивов u и v эффективнее выделить с помощью выравнивающего аллокатора fftw_malloc – это делает возможным использование инструкций SIMD. Число элементов в результате прямого преобразования из вещественного пространства вдвое меньше, чем в исходном массиве, поскольку вторая половина будет содержать комплексно-сопряженные значения. Для прямого и обратного преобразования создаются в начале и освобождаются в конце соответствующие планы. Важно, что планы настроены для преобразования из вещественных чисел в комплексные, без лишних промежуточных преобразований. Между вызовом прямого и обратного преобразования Фурье, согласно (7.8), производятся вычисления в спектральном

пространстве. Поскольку решение уравнение Пуассона определено с точностью до константы, то для сравнения результата работы FFTW и CUFFT удобно эту константу в конце привести к 0.

Теперь рассмотрим реализацию с помощью CUFFT:

```
__global__ void solve_transformed(int n,
    const cufftDoubleComplex* rhs, cufftDoubleComplex* u)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    int j = blockDim.y * blockIdx.y + threadIdx.y;

    int m = n / 2 + 1;
    if (i < m && j < n)
    {
        cufftDoubleComplex t = rhs[j * m + i];

        cufftDoubleReal w = cufftDoubleReal(M_PI) * (i < n/2 ? i : i - n);
        cufftDoubleReal v = cufftDoubleReal(M_PI) * (j < n/2 ? j : j - n);
        cufftDoubleReal s = (!i && !j) ? 1 : -4 * (w * w + v * v);

        t.x /= s;
        t.y /= s;

        u[j * m + i] = t;
    }
}

__global__ void scale_and_shift(int n, cufftDoubleReal* u, double shift)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    int j = blockDim.y * blockIdx.y + threadIdx.y;

    if (i < n && j < n)
        u[j * n + i] = (u[j * n + i] - shift) / (n * n);
}

int fft_gpu(int n, double* u)
{
    dim3 blk (32, 2);
    dim3 rgrd((n + blk.x - 1) / blk.x, (n + blk.y - 1) / blk.y);
    dim3 cgrd((n/2 + blk.x) / blk.x, (n + blk.y - 1) / blk.y);
    double shift;

    cufftDoubleComplex* v = NULL;
    cudaError_t cuerr = cudaMalloc((void**)&v, n * (n / 2 + 1) *
```

```
        sizeof(cufftDoubleComplex));
    if (cuerr != cudaSuccess)
    {
        fprintf(stderr, "Cannot create gpu memory buffer for v: %s\n",
            cudaGetErrorString(cuerr));
        return 1;
    }

    set_rhs<<<rgrd, blk>>>(n, u);

    cufftHandle forward, inverse;
    cufftPlan2d(&forward, n, n, CUFFT_D2Z);
    cufftPlan2d(&inverse, n, n, CUFFT_Z2D);
    cufftExecD2Z(forward, u, v);

    solve_transformed<<<cgrd, blk>>>(n, v, v);
    cufftExecZ2D(inverse, v, u);
    cudaMemcpy(&shift, u, sizeof(double), cudaMemcpyDeviceToHost);
    scale_and_shift<<<rgrd, blk>>>(n, u, shift);
    cudaFree(v);

    cufftDestroy(forward);
    cufftDestroy(inverse);

    return 0;
}
```

Здесь массив u расположен в памяти GPU. Так же, как и для FFTW, создаются и освобождаются планы. Важное отличие состоит в том, что вычисления в спектральном пространстве (7.8) необходимо выполнить над данными в памяти GPU – для этого запускается отдельное CUDA-ядро. Еще одним CUDA-ядром выполняется приведение константы к 0.

Далее рассмотрим вариант решения, демонстрирующий взаимную совместимость CUFFT и FFTW:

```
int fft_gpu_compatibility(int n, double* u)
{
    const double h = 1.0 / n;

    dim3 blk (32, 2);
    dim3 rgrd((n + blk.x - 1) / blk.x, (n + blk.y - 1) / blk.y);
    dim3 cgrd((n/2 + blk.x) / blk.x, (n + blk.y - 1) / blk.y);

    fftw_complex* v = (fftw_complex*)fftw_malloc(
```

```
        sizeof(fftw_complex) * (n / 2 + 1) * n);
    fftw_plan forward = fftw_plan_dft_r2c_2d(n, n, u, v, FFTW_ESTIMATE);

    fftw_execute(forward);

    for (int j = 0; j < n; j++)
        for (int i = 0; i < (n / 2 + 1); i++)
            v[j * (n / 2 + 1) + i][0] /= wave_num2(i, j, n);

    double* u_gpu_dev = NULL;
    cufftDoubleComplex* v_gpu_dev = NULL;
    cufftHandle inverse;
    cufftPlan2d(&inverse, n, n, CUFFT_Z2D);
    cufftSetCompatibilityMode(inverse, CUFFT_COMPATIBILITY_FFTW_ALL);
    size_t size_of_u = sizeof(double) * n * n;
    size_t size_of_v = sizeof(cufftDoubleComplex) * (n / 2 + 1) * n;
    cudaMalloc((void**)&u_gpu_dev, size_of_u);
    cudaMalloc((void**)&v_gpu_dev, size_of_v);
    cudaMemcpy(v_gpu_dev, v, size_of_v, cudaMemcpyHostToDevice);
    cufftExecZ2D(inverse, v_gpu_dev, u_gpu_dev);

    double shift;
    cudaMemcpy(&shift, u_gpu_dev, sizeof(double), cudaMemcpyDeviceToHost);
    scale_and_shift<<<rgrd, blk>>>(n, u_gpu_dev, shift);

    cudaMemcpy(u, u_gpu_dev, size_of_u, cudaMemcpyDeviceToHost);

    cudaFree(u_gpu_dev);
    cudaFree(v_gpu_dev);

    cufftDestroy(inverse);

    fftw_free(v);

    fftw_destroy_plan(forward);

    return 0;
}
```

Массив u располагается в памяти CPU. Для него с помощью FFTW производится прямое преобразование Фурье и (7.8). Затем создается план CUFFT со свойством $CUFFT\_COMPATIBILITY\_FFTW\_ALL$. Далее результат прямого преобразования и (7.8) копируется из массива v в память GPU, обратное преобразование выполняется с помощью CUFFT, и возвращается в массив u.

В качестве упражнения читатель может реализовать прямое преобразования с помощью CUFFT, а обратное – функциями FFTW.

7.4. CURAND

CURAND – это библиотека генераторов (псевдо)случайных чисел. На GPU поддержание приемлемого статистического качества и периода представляет некоторую проблему, поскольку генераторы, работающие в отдельных нитях, должны быть достаточно независимы. Поэтому для GPU необходимы специальные генераторы.

Последовательность неслучайных чисел называется *псевдослучайной*, если она удовлетворяет большинству свойств случайной последовательности, но генерируется детерминированным алгоритмом. Последовательность неслучайных чисел называется *квазислучайной*, если она генерируется детерминированным алгоритмом и равномерно заполняет n-мерное пространство; таким образом, ее можно использовать в методах Монте-Карло вместо случайной (при этом метод может работать лучше, чем со случайной последовательностью).

С помощью CURAND можно генерировать последовательности псевдо- и квазислучайных чисел:

- *CURAND_RNG_PSEUDO_XORWOW* – генератор псевдослучайных чисел на основе алгоритма XORWOW[10]. Генератор выдает последовательности из $2^{31} - 1$ целых чисел, или $2^{64} - 1$ пар целых чисел, или $2^{96} - 1$ троек целых чисел.

- 4 типа генераторов на основе алгоритма Соболя[11] для генерации квазислучайных чисел (каждый тип генерирует последовательности с измерениями до 20000):

  - $CURAND\_RNG\_QUASI\_SOBOL32$
  - $CURAND\_RNG\_QUASI\_SCRAMBLED\_SOBOL32$
  - $CURAND\_RNG\_QUASI\_SOBOL64$
  - $CURAND\_RNG\_QUASI\_SCRAMBLED\_SOBOL64$

Библиотека CURAND имеет два интерфейса: для хоста (“curand.h”) и для GPU-ядер (“curand_kernel.h”). В первом случае вызовы функций CURAND должны являться частью кода хост-программы, и случайные числа могут генерироваться как на хосте, так и на GPU. Если генератор работает на GPU, то все необходимые вычисления производятся на GPU, результат будет помещен в глобальную память GPU и доступен для использования в CUDA-ядрах или для копирования в память хоста. Если генератор работает на хосте, то все вычисления производятся на хосте, и результат помещается в память хоста.

Типовая схема использования хост-интерфейса CURAND выглядит следующим образом:

1. Создать генератор с помощью *curandCreateGenerator()*.
2. Задать необходимые свойства генератора, например, начальное состояние: *curandSetPseudoRandomGeneratorSeed()*.
3. Выделить память на GPU с помощью *cudaMalloc*.
4. Запустить генерацию случайных чисел с необходимым распределением:
   - равномерное: *curandGenerateUniform()*;
   - нормальное: *curandGenerateNormal()*;
   - log-нормальное: *curandGenerateLogNormal()*.
5. Использовать результаты работы генератора в приложении.
6. При необходимости сгенерировать дополнительные данные новыми вызовами генератора.
7. *curandDestroyGenerator()* – освободить генератор.

Вызовы функций CURAND возвращают статус ошибки типа *curandStatus_t*.

Второй интерфейс позволяет генерировать случайные числа непосредственно в CUDA-ядрах, в месте их использования. Ниже приведена одна из возможных схем:

1. Выделить место в глобальной памяти GPU для массива состояний генераторов каждой нити *curandState*.

2. Создать и запустить ядро, инициализирующее *curandState* для каждой нити.

3. Создать и запустить ядро, использующее случайные числа, генерируемые функцией *curand(curandState)*.

4. Освободить память.

Инициализацию и использование генератора может быть более эффективно разместить в разных ядрах, поскольку инициализация требует больше регистров и локальной памяти.

7.4.1. Пример: генерация показаний распределенных датчиков

Пусть имеется некоторое приложение, обрабатывающее показания распределенных на местности датчиков, например, решающее задачу из пункта 8.1.7 Если необходимо проверить работу приложения на различных входных данных, поступающих от нескольких миллионов датчиков, то целесообразно использовать генератор из GPU API библиотеки CURAND:

```
const uint number_to_print = 15;
const uint num_days = 1024, num_sites = 16;
const uint thread_dim = 256, block_dim = num_days * num_sites / thread_dim;
__global__ void setup_random_states(curandState *state, int n)
{
    int id = threadIdx.x + blockIdx.x * blockDim.x;
    if (id < n)
        curand_init(1234, id, 0, &state[id]);
}
__global__ void generate_numbers(
    curandState *state, int* numbers, int numbers_size, uint range_max) {
    int id = threadIdx.x + blockIdx.x * blockDim.x;

    if (id >= numbers_size) return;

    curandState localState = state[id];
    numbers[id] = curand(&localState) % (2 * range_max) - range_max;
    state[id] = localState;
}
struct measurement_to_triple
{
    int m_num_days, m_num_sites;
    measurement_to_triple(int num_days, int num_sites) :
```

```
        m_num_days(num_days), m_num_sites(num_sites) { }
    __host__ __device__
    thrust::tuple<int, int, int> operator()(thrust::tuple<int, int> x)
    {
        int index, measurement;
        thrust::tie(index, measurement) = x;
        int day = index / m_num_sites, site = index % m_num_sites;
        return thrust::make_tuple(day, site, measurement);
    }
};
struct measurement_is_negative
{
    __host__ __device__
    bool operator()(thrust::tuple<int, int, int> x)
    {
        return thrust::get<2>(x) < 0; // i.e. measurement < 0
    }
};
struct is_positive
{
    __host__ __device__
    bool operator()(int v) { return v > 0; }
};

void collect_data_curand(
    const uint num_days, const uint num_sites,
    device_vector<int>& day, device_vector<int>& site,
    device_vector<int>& measurement)
{
    int total_size = num_days * num_sites;
    device_vector<int> measurement_n(total_size),
        day_n(total_size), site_n(total_size);

    curandState *devStates;
    cudaMalloc((void**)&devStates, total_size * sizeof(curandState));
    setup_random_states<<<block_dim, thread_dim>>>(devStates, total_size);
    int *measurement_n_ptr = thrust::raw_pointer_cast(&measurement_n[0]);
    generate_numbers<<<block_dim, thread_dim>>>(
        devStates, measurement_n_ptr, measurement_n.size(), 50);

    thrust::counting_iterator<int> cnt_iter(0);
    thrust::transform(
        thrust::make_zip_iterator(
            thrust::make_tuple(cnt_iter, measurement_n.begin())),
        thrust::make_zip_iterator(
            thrust::make_tuple(cnt_iter + measurement_n.size(),
```

```
            measurement_n.end())),
        thrust::make_zip_iterator(
            thrust::make_tuple(day_n.begin(), site_n.begin(),
            measurement_n.begin())),
        measurement_to_triple(num_days, num_sites));

    int size_n = thrust::count_if(measurement_n.begin(),
        measurement_n.end(), is_positive());
    day.resize(size_n); site.resize(size_n); measurement.resize(size_n);
    thrust::remove_copy_if(
        thrust::make_zip_iterator(
            thrust::make_tuple(day_n.begin(), site_n.begin(),
            measurement_n.begin())),
        thrust::make_zip_iterator(
            thrust::make_tuple(day_n.end(), site_n.end(),
            measurement_n.end())),
        thrust::make_zip_iterator(
            thrust::make_tuple(day.begin(), site.begin(),
            measurement.begin())),
        measurement_is_negative());

    cudaFree(devStates);
}
```

Промежуточные данные хранятся в векторах *measurement_n*, *day_n* и *site_n* в памяти GPU. Работа начинается с инициализации состояния генераторов *devStates* в ядре *setup_random_states*. С помощью ядра *generate_numbers* генерируется последовательность измерений. Значения не отсортированы и принимают значения на отрезке $[-50, 50]$. Для передачи указателя на память вектора *measurement_n* используется *thrust::raw_pointer_cast*. С помощью функтора *measurement_to_triple* из каждого значения датчика образуется тройка день – участок – измерение. Для большего значения индекса соответствующая пара день – участок оказывается больше. Из получившейся последовательности трансформацией *thrust::remove_copy_if* удаляются все тройки, значения измерений которых отрицательны. Оставшиеся тройки копируются в три выходных последовательности.

В качестве упражнения читатель может реализовать генерацию данных, используя хост-API библиотеки CURAND.

Глава 8

Технологии для разработки на основе CUDA

8.1. Thrust

Thrust [12] – это библиотека параллельных алгоритмов обработки данных на GPU, представленных в виде векторов, с интерфейсом аналогичным C++ Standard Template Library (STL). Thrust удобен возможностью быстро реализовывать необходимые вычислительные алгоритмы в более простой и читаемой форме, чем явное программирование на CUDA. Библиотека уже содержит реализации некоторых базовых алгоритмов (scan, sort и reduce) и позволяет комбинировать их для составления более сложных схем обработки. С другой стороны, Thrust имеет более высокий уровень абстракции, чем CUDA, и не позволяет разработчику непосредственно управлять GPU на низком уровне, например, разделяемой памятью или синхронизацией нитей. Таким образом, Thrust может лучше всего подходить для GPU-приложений, в которых наиболее важную роль играет скорость разработки и надежность.

Thrust входит в состав CUDA Toolkit 4.0, но может быть также установлен отдельно. Сама библиотека состоит только из заголовочных файлов, и необходимые части кода Thrust будут скомпилированы вместе с целевым приложением.

8.1.1. Простейшее преобразование на примере сложения векторов

Пусть необходимо сложить два вектора и поместить результат в третий вектор. На C++ это тривиально реализуется следующим кодом:

```
for (int i = 0; i < N; ++i)
    Z[i] = X[i] + Y[i];
```

Реализация с помощью Thrust не намного сложнее:

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <iostream>

int main(void)
{
```

```
    thrust::device_vector<float> X(3);
    thrust::device_vector<float> Y(3);
    thrust::device_vector<float> Z(3);

    X[0] = 10; X[1] = 20; X[2] = 30;
    Y[0] = 15; Y[1] = 35; Y[2] = 10;

    thrust::transform(X.begin(), X.end(),
                      Y.begin(),
                      Z.begin(),
                      thrust::plus<float>());

    for (size_t i = 0; i < Z.size(); i++)
        std::cout << "Z[" << i << "] = " << Z[i] << "\n";

    return 0;
}
```

Реализация с помощью Thrust начинается с подключения необходимых заголовочных файлов. Далее в функции *main* объявляются входные векторы *X* и *Y* и выходной вектор *Z*, в который будет помещен результат сложения. Контейнер *device_vector* аналогичен *std::vector*, но сами данные размещаются в глобальной памяти GPU (при объявлении будет вызван *cudaMalloc*). Затем элементы входных векторов инициализируются с помощью оператора присваивания. Каждое присваивание копирует значение элемента в память GPU вызовом *cudaMemcpy*. Функция *thrust::transform* применяет к соответствующим элементам двух входных последовательностей бинарную операцию *thrust::plus*, результат которой помещается в выходную последовательность. Последовательности задаются *итераторами* начал векторов – *X.begin()*, *Y.begin()*, *Z.begin()* – и конца вектора *X* – *X.end()*. Задавать *Y.end()* и *Z.end()* нет необходимости, поскольку длина всех последовательностей должна быть одинаковой, и пары *X.begin()*, *X.end()* достаточно для ее определения. Конструкция *thrust::plus<float>* – это библиотечный функтор (о функторах речь пойдет далее) операции сложения двух элементов, параметризованный типом *float*, который применяется к каждой паре элементов *X* и *Y*. Аналогично оператору присваивания, оператор «[ ]» реализует копирование данных из памяти GPU в память хоста, что позволяет легко производить поэлементный доступ. Память GPU, занимаемая векторами, будет автоматически освобождена деструктором вектора при выходе за пределы области видимости, содержащей определения.

Сборка приложения, использующего Thrust ничем не отличается от сборки любого CUDA-приложения:

```
\$ nvcc —O2 ex01_vector_addition.cu —o ex01_vector_addition
\$ ./ex01_vector_addition
Z[0] = 25
Z[1] = 55
Z[2] = 40
```

Входной файл должен иметь расширение cu. Файл с расширением cpp, содержащий расширения языка C++, будет сразу передан CPU-компилятору, который скорее всего вернет ошибки компиляции.

8.1.2. Функторы на примере операции SAXPY

Реализуем с помощью Thrust операцию сложения двух векторов с коэффициентом, аналогичную функции SAXPY интерфейса BLAS (см. пункт 7.1). На C++ это тривиально реализуется следующим кодом:

```
for (int i = 0; i < N; ++i)
    Z[i] = a * X[i] + Y[i];
```

Этот пример интересен тем, что в Thrust нет соответствующей встроенной операции суммирования векторов, аналогичной *thrust::plus*, а значит ее необходимо реализовать вручную в виде *функтора* – оператора вызова «()», инкапсулированного в структуру или класс:

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <iostream>

struct saxpy
{
    float a;

    saxpy(float a) : a(a) {}

    __host__ __device__
    float operator()(float x, float y)
    {
        return a * x + y;
    }
```

```
};

int main(void)
{
    thrust::device_vector<float> X(3), Y(3), Z(3);

    X[0] = 10; X[1] = 20; X[2] = 30;
    Y[0] = 15; Y[1] = 35; Y[2] = 10;

    float a = 2.0f;

    thrust::transform(X.begin(), X.end(),
                      Y.begin(),
                      Z.begin(),
                      saxpy(a));

    for (size_t i = 0; i < Z.size(); i++)
        std::cout << "Z[" << i << "] = " << Z[i] << "\n";

    return 0;
}
```

Функтор является частью класса *saxpy*, который определяет дополнительные параметры состояния функтора через поле *a* и конструктор. Оператор вызова «()» принимает два вещественных числа и возвращает их линейную комбинацию. Функция *main* отличается от предыдущего примера созданием экземпляра функтора *saxpy(a)* в качестве параметра вызова *thrust::transform*, где *a = 2,0*. Можно заметить, что при отсутствии умножения на скаляр *a* функтор становится эквивалентен *thrust::plus*. Поскольку оператор «()» объявлен с квалификаторами *__host__* и *__device__*, то по отношению к GPU он будет являться не ядром (*__global__*), а device-функцией, вызываемой из окружения, которое формирует библиотека Thrust.

Операцию SAXPY из предыдущего примера можно выразить в более компактной форме, используя *маркеры подстановки* (placeholders)[1]:

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <iostream>
```

[1] Функторы и маркеры подстановки по сути очень близки к *лямбда-выражениям*. Лямбда-выражения являются частью стандарта *C++0x*, однако в Thrust они не реализованы, поскольку их пока не поддерживает компилятор nvcc.

```
using namespace thrust::placeholders;

int main(void)
{
    thrust::device_vector<float> X(3), Y(3), Z(3);

    X[0] = 10; X[1] = 20; X[2] = 30;
    Y[0] = 15; Y[1] = 35; Y[2] = 10;

    float a = 2.0f;

    thrust::transform(X.begin(), X.end(),
                      Y.begin(),
                      Z.begin(),
                      a * _1 + _2);

    for (size_t i = 0; i < Z.size(); i++)
        std::cout << "Z[" << i << "] = " << Z[i] << "\n";

    return 0;
}
```

Маркеры подстановки доступны в Thrust 1.5, который войдет в состав CUDA Toolkit 4.1. С их помощью функторы создаются неявно, что упрощает реализацию простых арифметических операций над векторами.

8.1.3. Трансформации общего вида, zip_iterator

В предыдущих примерах рассматривались только бинарные трансформации. Кроме них Thrust поддерживает унарные и тернарные трансформации, а также трансформации общего вида (таблица 8.1).

Таблица 8.1. Виды трансформаций

Унарная	$X[i] = f(A[i])$
Бинарная	$X[i] = f(A[i], B[i])$
Тернарная	$X[i] = f(A[i], B[i], C[i])$
Общего вида	$X[i] = f(A[i], B[i], C[i], \ldots)$

Трансформацию можно обобщить для различного числа последовательностей с

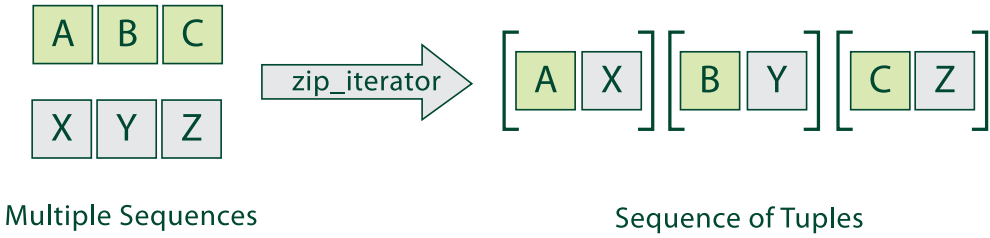


Рис. 8.1. Zip_iterator

помощью *кортежей* (tuples). Конструкция $zip\_iterator$ объединяет n входных последовательностей в n-местные кортежи (рис. 8.1). За счет этого thrust::transform для нескольких входных последовательностей превращается в унарное преобразование над множеством кортежей. Максимальная размерность преобразования ограничена реализацией шаблонов языка C++, и верхняя граница равна 10. Ниже приведен пример тернарной трансформации:

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <thrust/iterator/zip_iterator.h>
#include <iostream>

struct linear_combo
{
    __host__ __device__
    float operator()(thrust::tuple<float,float,float> t)
    {
        float x, y, z;

        thrust::tie(x,y,z) = t;

        return 2.0f * x + 3.0f * y + 4.0f * z;
    }
};

int main(void)
{
    thrust::device_vector<float> X(3), Y(3), Z(3);
    thrust::device_vector<float> U(3);

    X[0] = 10; X[1] = 20; X[2] = 30;
    Y[0] = 15; Y[1] = 35; Y[2] = 10;
```

```
    Z[0] = 20; Z[1] = 30; Z[2] = 25;

    thrust::transform(
        thrust::make_zip_iterator(thrust::make_tuple(X.begin(),
            Y.begin(), Z.begin())),
        thrust::make_zip_iterator(thrust::make_tuple(X.end(),
            Y.end(),   Z.end())),
        U.begin(),
        linear_combo());

    for (size_t i = 0; i < Z.size(); i++)
        std::cout << "U[" << i << "] = " << U[i] << "\n";

    return 0;
}
```

В этом примере имеется три входных последовательности X, Y и Z, и одна выходная последовательность U. В отличие от предыдущего примера, X, Y и Z передаются не в качестве отдельных аргументов, а в «контейнерах» zip_iterator, каждый из которых возвращает кортеж. Кроме этого, в операторе вызова функтора «()» вещественные переменные теперь также объединены в кортеж. Отдельные вещественные значения из кортежа можно получить вызовом функции thrust::get или thrust::tie. Функтор подобной формы также может иметь параметры состояния, например, коэффициенты линейного преобразования.

8.1.4. Редукция

Помимо алгоритма трансформации, важным является алгоритм редукции (см. пункт 5.1). Рассмотрим пример суммирования элементов последовательности – редукции с бинарной операцией сложения:

```
#include <thrust/device_vector.h>
#include <thrust/reduce.h>
#include <thrust/functional.h>
#include <iostream>

int main(void)
{
    thrust::device_vector<float> X(3);

    X[0] = 10; X[1] = 30; X[2] = 20;
```

```
    float result = thrust::reduce(X.begin(), X.end());

    std::cout << "sum is " << result << "\n";

    return 0;
}
```

В данном примере вызывается функция $thrust :: reduce$. По умолчанию индекс начального элемента равен нулю, а операция – сложение. Тип данных определяется автоматически.

Аналогичным образом может быть реализована редукция с операцией поиска максимума:

```
#include <thrust/device_vector.h>
#include <thrust/reduce.h>
#include <thrust/functional.h>
#include <iostream>

int main(void)
{
  thrust::device_vector<float> X(3);

  X[0] = 10; X[1] = 30; X[2] = 20;

  float init = 0.0f;

  float result = thrust::reduce(X.begin(), X.end(),
                                init,
                                thrust::maximum<float>());

  std::cout << "maximum is " << result << "\n";

  return 0;
}
```

Для редукции с поиском глобального максимума передается начальный элемент $init$ и экземпляр функтора thrust::maximum<float>, который возвращает максимум из двух элементов. Данная простая реализация может корректно работать только для неотрицательных элементов.

Вслед за поиском максимума, рассмотрим поиск индекса максимума:

```
#include <thrust/device_vector.h>
#include <thrust/reduce.h>
#include <thrust/functional.h>
```

```
#include <thrust/iterator/zip_iterator.h>
#include <iostream>

typedef thrust::tuple<int,int> Tuple;

struct max_index
{
    __host__ __device__
    Tuple operator()(Tuple a, Tuple b)
    {
        if (thrust::get<0>(a) > thrust::get<0>(b))
            return a;
        else
            return b;
    }
};

int main(void)
{
    thrust::device_vector<int> X(3), Y(3);

    X[0] = 10; X[1] = 30; X[2] = 20;  // values
    Y[0] =  0; Y[1] =  1; Y[2] =  2;  // indices

    Tuple init(X[0],Y[0]);

    Tuple result = thrust::reduce
        (thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y.begin())),
         thrust::make_zip_iterator(thrust::make_tuple(X.end(),   Y.end())),
         init,
         max_index());

    int value, index;  thrust::tie(value,index) = result;

    std::cout << "maximum value is " << value <<
        " at index " << index << "\n";

    return 0;
}
```

Для того, чтобы вместе со значением хранить и индекс максимума, формируется вспомогательная последовательность индексов $[0..2]$. Из этих двух последовательностей $zip\_iterator$ получает пары (значение, индекс). Функтор max_index производит сравнение кортежей по первым элементам пар. Данный метод может

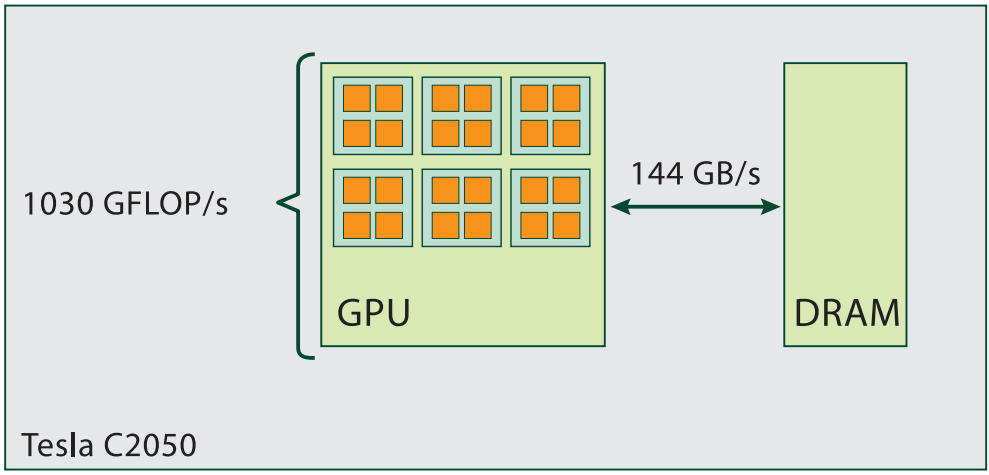


Рис. 8.2. Архитектура Tesla C2050

быть использован и в более сложных случаях преобразований над данными вида (ключ, значение).

8.1.5. Производительность

Обратной стороной удобства и простоты Thrust является невозможность использования элементов программирования CUDA-ядер (выбор вычислительной сетки, синхронизация нитей, разделяемая память и т.д.), поэтому производительность вычислений во многом зависит от выбора оптимальных параметров внутри самого Thrust. Но прежде, чем предпринимать в этой связи действия, важно правильно оценить факторы эффективности в задачах, решаемых с помощью Thrust.

Двойственность в ограничении производительности скоростью памяти и скоростью вычислений выражает характеристика *плотности вычислений* (computational intensity). Например, Tesla C2050 имеет пиковую производительность 1030 GFLOPS и пиковую скорость обмена данными с глобальной памятью 144 Gb/сек (рис. 8.2). В таком случае оптимальная плотность вычислений для данной модели GPU – 1 запрос в память на 7 вычислительных операций, 7:1. В зависимости от того, как соотносятся оптимальная плотность вычислений устройства и задачи, следует применять те или иные оптимизации.

Таблица 8.2. Плотность вычислений в некоторых типовых задачах

Задача	**FLOP/байт (без учета индексной арифметики)**
Сложение векторов	1 : 12
SAXPY	2 : 12
Тернарная трансформация	5 : 20
Суммирование элементов	1 : 4
Индекс максимального элемента	1 : 12

На рис. 8.3 представлена абстрактная шкала плотности вычислений для некоторых типичных приложений. Чем левее расположена отметка, тем больше производительность в задаче зависит от скорости обмена данными (memory bound), и, соответственно, чем правее – тем больше от скорости вычислений (compute bound). Задача *saxpy* находится слева, поскольку имеет плотность 1:1 – умножение и сложение двух операндов. Чем правее операция, тем больше отношение «количество вычислений / количество обращений к памяти». Большее значение достигается за счет того, что единожды загруженные из памяти значения используются в нескольких арифметических операциях. В таблице 8.2 приведены теоретические значения вычислительной плотности для некоторых классов задач, решаемых с помощью Thrust. Видно, что по сравнению с пиковой вычислительной плотностью GPU (таблица 8.3), эти значения малы. Следовательно, при разработке все усилия должны быть приложены к оптимизации скорости передачи данных. В следующих примерах показано, как этого достичь.

Таблица 8.3. Пиковая плотность вычислений для различных моделей GPU

Устройство	**FLOP/байт**
GeForce GTX 280	$\sim 7.0 : 1$
GeForce GTX 480	$\sim 7.6 : 1$
Tesla C870	$\sim 6.7 : 1$
Tesla C1060	$\sim 9.1 : 1$
Tesla C2050	$\sim 7.1 : 1$

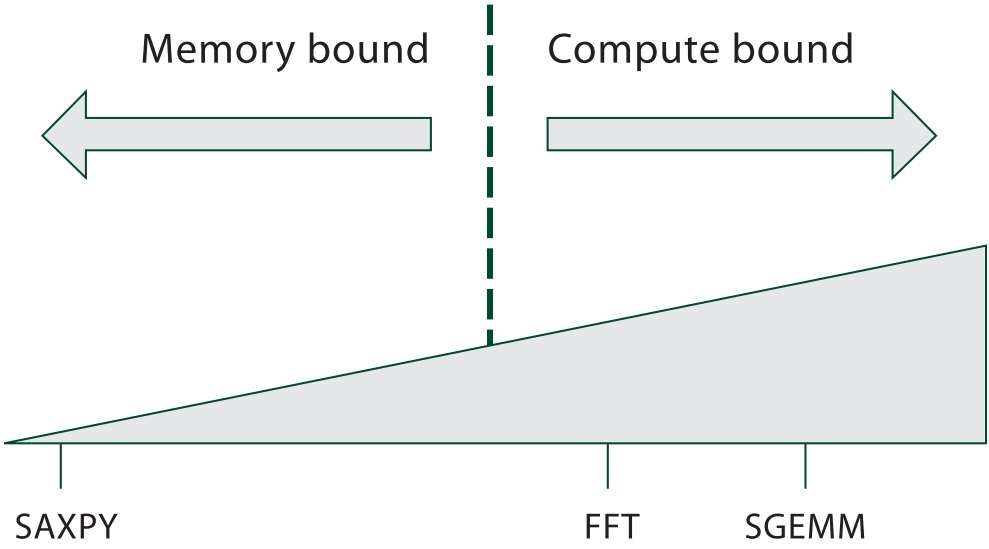


Рис. 8.3. Шкала плотности вычислений

Оптимизация поиска максимума. Рассмотрим оптимизированный вариант поиска индекса максимального элемента:

```
#include <thrust/device_vector.h>
#include <thrust/reduce.h>
#include <thrust/functional.h>
#include <thrust/iterator/zip_iterator.h>
#include <iostream>

typedef thrust::tuple<int,int> Tuple;

struct max_index
{
    __host__ __device__
    Tuple operator()(Tuple a, Tuple b)
    {
        if (thrust::get<0>(a) > thrust::get<0>(b))
            return a;
        else
            return b;
    }
};

int main(void)
```

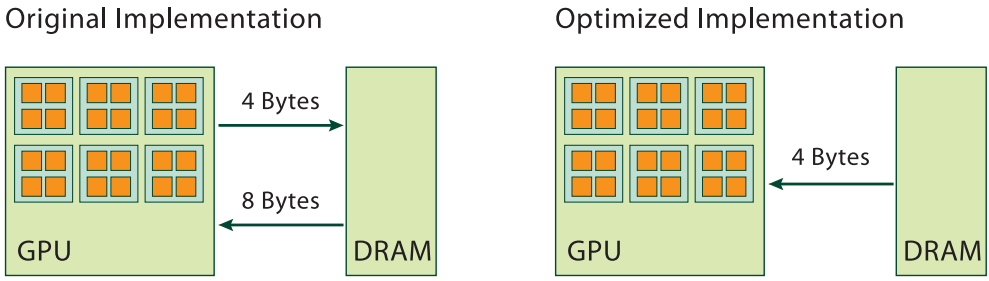


Рис. 8.4. Передача данных в оптимизированном поиске индекса максимума

```
{
    thrust::device_vector<int>     X(3);
    thrust::counting_iterator<int> Y(0);

    X[0] = 10; X[1] = 30; X[2] = 20;

    Tuple init(X[0],Y[0]);

    Tuple result = thrust::reduce
        (thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y)),
         thrust::make_zip_iterator(thrust::make_tuple(X.end(),
                                   Y + X.size())),
         init,
         max_index());

    int value, index;  thrust::tie(value,index) = result;

    std::cout << "maximum value is " << value <<
        " at index " << index << "\n";

    return 0;
}
```

Отличие от предыдущей версии состоит в использовании *counting_iterator*. Вместо явного задания массива индексов, *counting_iterator* при необходимости генерирует соответствующие значения во время исполнения программы.

Таким образом, оптимизирована передача данных: в случае наличия вектора Y, необходимо было загрузить 8 байт и выгрузить обратно 4 байта, а с использованием *counting_iterator* – только загрузить 4 байта (рис. 8.4). В данной “memory bound”-задаче такая оптимизация ускорит выполнение ядра примерно в 3 раза.

Слияние операций. Другой метод оптимизации – *слияние* (fusion) операций для повторного использования загруженных из памяти значений. Слияние можно пояснить на примере двух независимых циклов:

```
for (int i = 0; i < N; ++i)
    U[i] = F(X[i],Y[i],Z[i]);
for (int i = 0; i < N; ++i)
    V[i] = G(X[i],Y[i],Z[i]);
```

Тогда результатом слияния будет объединение операций в одном цикле:

```
for (int i = 0; i < N; ++i)
{
    U[i] = F(X[i],Y[i],Z[i]);
    V[i] = G(X[i],Y[i],Z[i]);
}
```

В результате, входные аргументы будут прочитаны из памяти один раз, а затем использованы снова.

В следующем примере показано, как слияние может быть использовано в трансформациях Thrust:

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <thrust/iterator/zip_iterator.h>
#include <iostream>

typedef thrust::tuple<float,float>       Tuple2;
typedef thrust::tuple<float,float,float> Tuple3;

struct linear_combo
{
    __host__ __device__
    Tuple2 operator()(Tuple3 t)
    {
        float x, y, z; thrust::tie(x,y,z) = t;

        float u = 2.0f * x + 3.0f * y + 4.0f * z;
        float v = 1.0f * x + 2.0f * y + 3.0f * z;

        return Tuple2(u,v);
    }
};
```

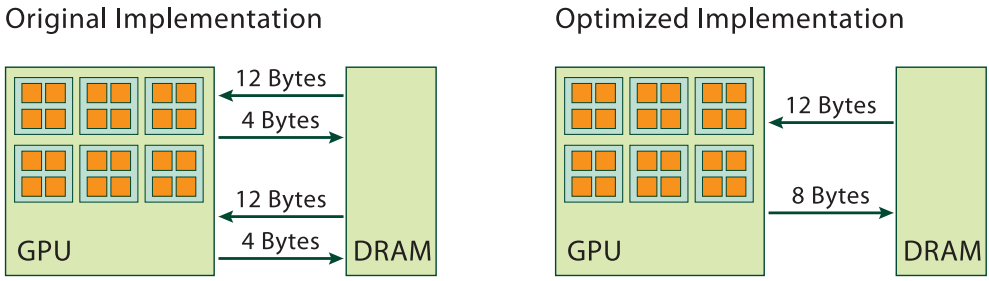


Рис. 8.5. Передача данных при слиянии для трансформаций

```
int main(void)
{
    thrust::device_vector<float> X(3), Y(3), Z(3);
    thrust::device_vector<float> U(3), V(3);

    X[0] = 10; X[1] = 20; X[2] = 30;
    Y[0] = 15; Y[1] = 35; Y[2] = 10;
    Z[0] = 20; Z[1] = 30; Z[2] = 25;

    thrust::transform(
        thrust::make_zip_iterator(thrust::make_tuple(
            X.begin(), Y.begin(), Z.begin())),
        thrust::make_zip_iterator(thrust::make_tuple(
            X.end(),   Y.end(),   Z.end())),
        thrust::make_zip_iterator(thrust::make_tuple(
            U.begin(), V.begin())),
        linear_combo());

    for (size_t i = 0; i < Z.size(); i++)
        std::cout << "U[" << i << "] = " << U[i] <<
            "  V[" << i << "] = " << V[i] << "\n";

    return 0;
}
```

С помощью *zip_iterator* в кортежи объединены как входные последовательности *X*, *Y* и *Z*, так и выходные – *U* и *V*. Таким образом, при расчете *U* и *V* дважды используются загруженные один раз элементы *X*, *Y* и *Z*, что сокращает трафик операций с памятью с 32 байт до 20 (рис. 8.5).

Слияние для редукции. Теперь рассмотрим слияние в сочетании с редукцией. Обычный CPU-код редукции на C++

```
for (int i = 0; i < N; ++i)
    Y[i] = F(X[i]);
for (int i = 0; i < N; ++i)
    sum += Y[i];
```

в результате слияния примет вид:

```
for (int i = 0; i < N; ++i)
    sum += F(X[i]);
```

Аналогичным образом слияние может быть использовано при редукции суммы квадратов на Thrust:

```
#include <thrust/device_vector.h>
#include <thrust/transform_reduce.h>
#include <thrust/functional.h>
#include <iostream>

using namespace thrust::placeholders;

int main(void)
{
    thrust::device_vector<float> X(3);

    X[0] = 10; X[1] = 30; X[2] = 20;

    float result = thrust::transform_reduce
        (X.begin(), X.end(),
         _1 * _1,
         0.0f,
         thrust::plus<float>());

    std::cout << "sum of squares is " << result << "\n";

    return 0;
}
```

Вызов *thrust::transform_reduce* помимо редукции также выполняет и трансформацию. Для создания функтора используются маркеры подстановки. В данном случае слияние ускоряет работу примерно в 3 раза (рис. 8.6).

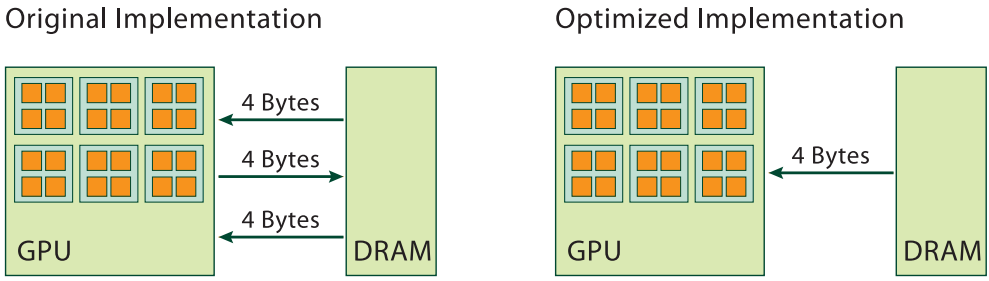


Рис. 8.6. Передача данных при слиянии для редукции

8.1.6. Взаимодействие Thrust и CUDA C

В Thrust предусмотрены механизмы обеспечения совместимости форматов данных с CUDA C в обоих направлениях:

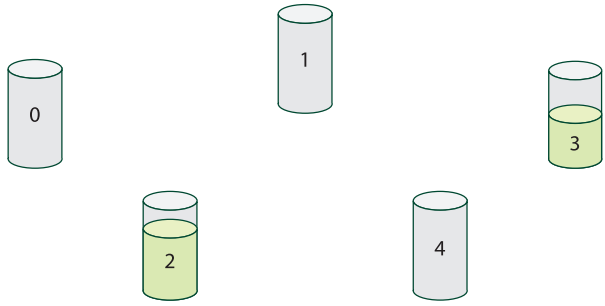
1. Преобразование итератора в указатель для обработки на CUDA C;
2. Преобразование (обертка) указателя в итератор для обработки с использованием Thrust.

В первом случае вектор в памяти GPU можно преобразовать с помощью функции *thrust::raw_pointer_cast* и, например, передать в CUDA-ядро:

```
// Создать вектор на устройстве
thrust::device_vector<int> d_vec(4);
// Получить указатель на память вектора
int* ptr = thrust::raw_pointer_cast(&d_vec[0]);
// Использовать указатель при вызове ядра на CUDA C
my_kernel<<< N / 256, 256 >>>(N, ptr);
// Использовать указатель в функции из CUDA API
cudaMemcpyAsync(ptr, ... );
```

Во втором случае указатель на память GPU (например, выделенную с помощью cudaMalloc), можно преобразовать функцией *thrust::device_ptr* для последующего использования в Thrust:

```
// Указатель на память устройства
int * raw_ptr;
cudaMalloc((void **) &raw_ptr, N * sizeof(int));
// Преобразовать указатель с помощью device_ptr
thrust::device_ptr<int> dev_ptr(raw_ptr);
```



day	[0	0	1	2	5	5	6	6	7	8	...	]
site	[2	3	0	1	1	2	0	1	2	1	...	]
measurement	[9	5	6	3	3	8	2	6	5	10	...	]

Notes
1) Time series sorted by day
2) Measurements of zero are exluded from the time series

Рис. 8.7. Задача расчета общего количества осадков

```
// Использовать device_ptr в алгоритмах Thrust
thrust::fill(dev_ptr, dev_ptr + N, (int) 0);
// Обратиться к памяти устройства с помощью device_ptr
dev_ptr[0] = 1;
// Освободить память
cudaFree(raw_ptr);
```

Прямое использование указателя на массив в Thrust приведет к тому, что преобразование будет выполнено на CPU.

8.1.7. Пример: расчет общего количества осадков

Пусть дано множество распределенных на местности датчиков осадков, производящих ежедневные измерения (рис. 8.7). Все показания отсортированы по возрастанию по дням и затем по номеру участка. Если в определенный день ни один датчик не зафиксировал осадков, то день не включается в список измерений. Требуется по трем входным последовательностям (номер дня, номер участка, количество осадков) получить последовательность с общим количеством осадков на каждом участке.

Следующая функция генерирует тестовые входные данные на хосте с помощью Thrust:

```
template<typename Vector>
void collect_data(
    int num_days, int num_sites, Vector& day,
    Vector& site, Vector& measurement)
{
    thrust::default_random_engine rng;
    for (int i = 0; i < num_days; i++)
    {
        for (int j = 0; j < num_sites; j++)
        {
            thrust::uniform_int_distribution<int> dist(0, 10);
            int sum = -35;
            for (int k = 0; k < 6; k++) sum += dist(rng);

            if (sum > 0)
            {
                day.push_back(i);
                site.push_back(j);
                measurement.push_back(sum);
            }
        }
    }
}
```

Для каждого дня запускается цикл по номеру участка. Затем для каждой пары случайно генерируется значение датчика. Если оно больше нуля, то тройка добавляется во входную последовательность, иначе выполняется следующая итерация цикла. При такой генерации входные последовательности отсортированы по номеру дня, а затем по номеру участка.

Пример генерации входных данных *day*, *site* и *measurement* с помощью библиотеки CURAND приведен в пункте 7.4.1.

Для решения самой задачи достаточно всего двух операций: *sort_by_key* и *reduce_by_key*:

```
template<typename Vector>
void total_rainfall_at_each_site(
    int num_days, int num_sites,
    Vector& day, Vector& site, Vector& measurement)
{
    std::cout << "Initial data:" << std::endl;
```

```
    thrust::copy(day.begin(), day.end(),
        std::ostream_iterator<int>(std::cout, " "));
    std::cout << std::endl;
    thrust::copy(site.begin(), site.end(),
        std::ostream_iterator<int>(std::cout, " "));
    std::cout << std::endl;
    thrust::copy(measurement.begin(), measurement.end(),
        std::ostream_iterator<int>(std::cout, " "));
    std::cout << std::endl;
    Vector site_total(num_sites), measurement_total(num_sites);
    thrust::sort_by_key(site.begin(), site.end(),
        measurement.begin());
    thrust::reduce_by_key(site.begin(), site.end(),
        measurement.begin(), site_total.begin(),
        measurement_total.begin());

    std::cout << "Total rainfall at each site: " << std::endl;
    thrust::copy(site_total.begin(), site_total.end(),
        std::ostream_iterator<int>(std::cout, " "));
    std::cout << std::endl;
    thrust::copy(measurement_total.begin(), measurement_total.end(),
        std::ostream_iterator<int>(std::cout, " "));
    std::cout << std::endl;
}
```

Первой функцией пары *(site, measurement)* сортируются по ключу *site*. Второй функцией с помощью редукции по ключу находятся суммы значений *measurement* с одинаковым *site*.

При написании этого раздела использованы материалы доклада [13].

8.1.8. Переключение целевой платформы Thrust (backend)

Thrust реализует единый интерфейс пользователя для трех типов целевых платформ: GPU/CUDA, CPU/OpenMP и CPU/TBB (начиная с Thrust версии 1.6.0). Другими словами, одна и та же программа, использующая Thrust, может быть исполнена как на GPU, так и на CPU. Рассмотрим эту возможность на примере сортировки массива:

```
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/sort.h>
#include <thrust/copy.h>
#include <cstdio>
```

```
#include <cstdlib>

#include "timing.h"

void usage(const char* filename)
{
    printf("Sort the random key-value array of the given length by key.\n");
    printf("Usage: %s <n>\n", filename);
}

using namespace thrust;

int main(int argc, char* argv[])
{
    const int printable_n = 128;

    if (argc != 2)
    {
        usage(argv[0]);
        return 0;
    }

    int n = atoi(argv[1]);
    if (n <= 0)
    {
        usage(argv[0]);
        return 0;
    }

    // Generate the random keys and values on the host.
    host_vector<int> h_keys(n);
    generate(h_keys.begin(), h_keys.end(), rand);
    host_vector<int> h_vals(n);
    generate(h_vals.begin(), h_vals.end(), rand);

    // Print out the input data if n is small.
    if (n <= printable_n)
    {
        printf("Input data:\n");
        for (int i = 0; i < n; i++)
            printf("(%d, %d)\n", h_keys[i], h_vals[i]);
        printf("\n");
    }

    // Transfer data to the device.
    util_time_t load_start, load_finish;
```

```
    util_get_time(&load_start);
#ifndef CPU
        device_vector<int> d_keys = h_keys;
        device_vector<int> d_vals = h_vals;
#endif
    util_get_time(&load_finish);

    util_time_t sort_start, sort_finish;
    util_get_time(&sort_start);
#ifdef CPU
        sort_by_key(h_keys.begin(), h_keys.end(), h_vals.begin());
#else
        sort_by_key(d_keys.begin(), d_keys.end(), d_vals.begin());
#endif
    util_get_time(&sort_finish);

    // Transfer data back to host.
    util_time_t save_start, save_finish;
    util_get_time(&save_start);
#ifndef CPU
        copy(d_keys.begin(), d_keys.end(), h_keys.begin());
        copy(d_vals.begin(), d_vals.end(), h_vals.begin());
#endif
    util_get_time(&save_finish);

    printf("Load time = %f sec\n",
        util_get_time_diff(&load_start, &load_finish));
    printf("Sort time = %f sec\n",
        util_get_time_diff(&sort_start, &sort_finish));
    printf("Save time = %f sec\n",
        util_get_time_diff(&save_start, &save_finish));

    // Print out the output data if n is small.
    if (n <= printable_n)
    {
        printf("Output data:\n");
        for (int i = 0; i < n; i++)
            printf("(%d, %d)\n", h_keys[i], h_vals[i]);
        printf("\n");
    }

    return 0;
}
```

По умолчанию программа будет скомпилирована для GPU/CUDA:

```
$ nvcc –O3 –c sort_by_key.cu –o sort_by_key.gpu.o
```

Для переключения на вариант CPU/OpenMP или CPU/TBB (Thread Building Blocks) достаточно установить соответствующее значение символа препроцессора *THRUST_DEVICE_BACKEND*:

```
$ nvcc -O3 -Xcompiler -fopenmp -o sort_by_key sort_by_key.cu -DCPU -↩
    DTHRUST_DEVICE_BACKEND=THRUST_DEVICE_BACKEND_OMP

$ nvcc -O2 -o sort_by_key sort_by_key.cu -DCPU -DTHRUST_DEVICE_BACKEND=↩
    THRUST_DEVICE_BACKEND_TBB -ltbb
```

С OpenMP или TBB не обязательно использовать компилятор *nvcc*, сборка может быть произведена, например, с помощью *g++*:

```
$ g++ -O2 -o sort_by_key sort_by_key.cpp -fopenmp -DTHRUST_DEVICE_BACKEND=↩
    THRUST_DEVICE_BACKEND_OMP -lgomp -I<path-to-thrust-headers>
```

Аналогично можно переключать реализацию алгоритмов, работающих с host-данными:

- *THRUST_HOST_BACKEND_CPP* (STL, по умолчанию),
- *THRUST_HOST_BACKEND_OMP*,
- *THRUST_HOST_BACKEND_TBB*.

Следующие тесты показывают время работы отдельных этапов приложения. Для компьютера с Intel Core2 Quad Q6600, 2.40GHz и NVIDIA Tesla C2050:

```
$ ./sort_by_key.cpu $((1024*1024*128))
Load time = 0.000000 sec
Sort time = 35.166115 sec
Save time = 0.000000 sec
$ ./sort_by_key.gpu $((1024*1024*128))
Load time = 0.662939 sec
Sort time = 0.412526 sec
Save time = 0.297485 sec
```

Для компьютера с Intel Xeon E5620, 2.40GHz и NVIDIA Tesla S1060:

```
$ ./sort_by_key.cpu $((1024*1024*128))
Load time = 0.000000 sec
Sort time = 7.626176 sec
Save time = 0.000000 sec
$ ./sort_by_key.gpu $((1024*1024*128))
Load time = 0.464901 sec
```

```
Sort time = 0.917674 sec
Save time = 0.176966 sec
```

Видно, что время CPU-реализации сортировки велико, но затраты на загрузку и выгрузку данных отсутствуют. В случае GPU-реализации время сортировки значительно меньше, но при этом некоторое время расходуется на пересылку массивов между памятью хоста и GPU.

8.1.9. Вызов Thrust из Fortran

Thrust может быть задействован в программах на Фортран посредством оберточных функций, написанных на языке C, и интерфейсов, использующих элементы *ISO_C_BINDING*. Рассмотрим данный метод на примере программы сортировки массива по ключу. Для сравнения производительности режим работы Thrust переключается между исполнением на GPU и CPU/OpenMP. Ниже приведена реализация сортировки с применением Thrust, в которой C++-код используется внутри C-совместимых функций (*extern "C"*):

```
// Filename: creduce.cu
#include <cstdio>
#include <curand_kernel.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/sort.h>
#include <thrust/copy.h>

using namespace thrust;

template <typename T>
struct gpu_rand : public thrust::unary_function<unsigned int, T>
{
    int scale;

    gpu_rand(int _scale = 1) : scale(_scale) { }

    __device__ T operator()(unsigned int thread_id)
    {
        unsigned int seed = thread_id;
        curandState s;

        // Seed a random number generator.
        curand_init(seed, 0, 0, &s);
```

```
        return (T)(curand_uniform(&s) * scale);
    }
};

template <typename T>
struct cpu_rand : public thrust::unary_function<unsigned int, T>
{
        int scale;

        cpu_rand(int _scale = 1) : scale(_scale) { }

        T operator()(unsigned int thread_id)
        {
        return (T)(((double)rand() / RAND_MAX) * scale);
        }
};

extern "C" void thrust_generate_floats(float* values, int n)
{
#ifdef CPU
    thrust::transform(counting_iterator<int>(0),
        counting_iterator<int>(n), values, cpu_rand<float>());
#else
    device_ptr<float> d_values(values);
    thrust::transform(counting_iterator<int>(0),
        counting_iterator<int>(n),d_values, gpu_rand<float>());
#endif
}

extern "C" void thrust_generate_integers(int* values, int n)
{
#ifdef CPU
    thrust::transform(counting_iterator<int>(0),
        counting_iterator<int>(n), values, cpu_rand<int>(n));
#else
    device_ptr<int> d_values(values);
    thrust::transform(counting_iterator<int>(0), counting_iterator<int>(n),
        d_values, gpu_rand<int>(n));
#endif
}

extern "C" void thrust_sort_by_key(int* keys, float* values, int n)
{
#ifdef CPU
    sort_by_key(keys, keys + n, values);
```

```
#else
    device_ptr<int> d_keys(keys);
    device_ptr<float> d_values(values);
    sort_by_key(d_keys, d_keys + n, d_values);
#endif
}
```

Для CPU/OpenMP файл компилируется командой

```
$ nvcc –O3 –DCPU –DTHRUST_DEVICE_BACKEND=THRUST_DEVICE_BACKEND_OMP –c thrust.↩
    cu –o thrust.o
```

Для GPU файл компилируется командой

```
$ nvcc –O3 –c thrust.cu –o thrust.o
```

В результате получен объектный файл *thrust.o*, С-функции из которого далее могут быть использованы в Фортран-программе, если для них определен соответствующий интерфейс:

```
! thrust.CUF

module thrust

interface

  subroutine thrust_generate_floats(values, n) bind(C)
  use iso_c_binding
  implicit none

  integer(c_int), value :: n
#ifdef CPU
  real, dimension(n) :: values
#else
  real, dimension(n), device :: values
#endif

  end subroutine thrust_generate_floats


  subroutine thrust_generate_integers(values, n) bind(C)
  use iso_c_binding
  implicit none

  integer(c_int), value :: n
#ifdef CPU
  integer, dimension(n) :: values
```

```
#else
  integer, dimension(n), device :: values
#endif

  end subroutine thrust_generate_integers


  subroutine thrust_sort_by_key(keys, values, n) bind(C)
  use iso_c_binding
  implicit none

  integer(c_int), value :: n
#ifdef CPU
  integer, dimension(n) :: keys
  real, dimension(n) :: values
#else
  integer, dimension(n), device :: keys
  real, dimension(n), device :: values
#endif

  end subroutine thrust_sort_by_key

end interface

end module thrust
```

Данный модуль использует расширения PGI CUDA Fortran и поэтому может быть собран только соответствующим компилятором. Однако, используя сведения из главы 4 читатель без труда сможет адаптировать данный код для работы с любым стандартным компилятором Фортран.

Для CPU/OpenMP файл компилируется командой

```
$ pgfortran –DCPU –c thrust.CUF –o thrust.mod.o
```

Для GPU файл компилируется командой

```
$ pgfortran –c thrust.CUF –o thrust.mod.o
```

Клиентом данного модуля может являться произвольный код на языке Фортран. В данном случае генерируются случайные массивы данных:

```
! sort_by_key.CUF

subroutine sort_by_key(n)

use cudafor
```

```
use thrust
implicit none

  integer, intent(in) :: n

  integer, parameter :: printable_n = 16
  integer :: i
  real, volatile :: start, finish

  integer, allocatable, dimension(:) :: h_keys
  real, allocatable, dimension(:) :: h_values

#ifndef CPU
  integer, allocatable, dimension(:), device :: d_keys
  real, allocatable, dimension(:), device :: d_values
#endif

  allocate(h_keys(n))
  allocate(h_values(n))

#ifndef CPU
  allocate(d_keys(n))
  allocate(d_values(n))
#endif

#ifdef CPU
  call thrust_generate_integers(h_keys, n)
  call thrust_generate_floats(h_values, n)
#else
  call thrust_generate_integers(d_keys, n)
  call thrust_generate_floats(d_values, n)
#endif

  if (n <= printable_n) then
#ifndef CPU
    h_keys = d_keys
    h_values = d_values
#endif
    print *, 'Input data: (keys, values)'
    do i = 1, n
      print *, h_keys(i), h_values(i)
    enddo
  endif

  call cpu_time(start)
```

```
#ifdef CPU
  call thrust_sort_by_key(h_keys, h_values, n)
#else
  call thrust_sort_by_key(d_keys, d_values, n)
#endif

  call cpu_time(finish)

  print *, 'time = ', finish - start

  if (n <= printable_n) then
#ifndef CPU
    h_keys = d_keys
    h_values = d_values
#endif
    print *, 'Output data: (keys, values)'
    do i = 1, n
      print *, h_keys(i), h_values(i)
    enddo
  endif

  deallocate(h_keys)
  deallocate(h_values)
#ifndef CPU
  deallocate(d_keys)
  deallocate(d_values)
#endif

end subroutine sort_by_key
```

Для CPU/OpenMP файл компилируется командой

```
$ pgfortran -DCPU -fast -c sort_by_key.CUF -o sort_by_key.o
```

Для GPU файл компилируется командой

```
$ pgfortran -fast -c sort_by_key.CUF -o sort_by_key.o
```

Чтобы упростить обработку входных аргументов, функция *main* реализована на языке C, и из нее вызывается Фортран-процедура *sort_by_key*:

```
// main.cu

#include <stdio.h>
#include <stdlib.h>

// В соответствии со стандартом, имена функций и процедур
```

```
// на языке Фортран завешаются знаком подчеркивания.
extern "C" void sort_by_key_(int* n);

void usage(const char* filename)
{
    printf("Sort the random key-value array of the given length by key.\n");
    printf("Usage: %s <n>\n", filename);
}

int main(int argc, char* argv[])
{
    if (argc != 2)
    {
        usage(argv[0]);
        return 0;
    }

    int n = atoi(argv[1]);
    if (n <= 0)
    {
        usage(argv[0]);
        return 0;
    }

#ifndef CPU
    int ngpus;
    cudaGetDeviceCount(&ngpus);
    printf("ngpus = %d\n", ngpus);
#endif
    printf("n = %d\n", n);

    sort_by_key_(&n);

    return 0;
}
```

Для CPU/OpenMP файл компилируется командой

```
$ nvcc –DCPU –c main.cu –o main.o
```

Для GPU файл компилируется командой

```
$ nvcc –c main.cu –o main.o
```

Следующие вызовы выполняют линковку исполняемой программы из объектных файлов (для CPU/OpenMP и для GPU):

```
$ pgfortran –Mnomain –Mcuda sort_by_key.o main.o thrust.mod.o thrust.o –o ↩
    sort_by_key
```

Запустим программу для проверки корректности сортировки массива малого размера:

```
$ ./sort_by_key.cpu 8
n = 8
 Input data: (keys, values)
            6    0.2777747
            3    0.5539700
            6    0.4773971
            6    0.6288709
            7    0.3647845
            1    0.5134009
            2    0.9522297
            6    0.9161951
 time =     1.4066696E–05
 Output data: (keys, values)
            1    0.5134009
            2    0.9522297
            3    0.5539700
            6    0.2777747
            6    0.4773971
            6    0.6288709
            6    0.9161951
            7    0.3647845

$ ./sort_by_key.gpu 8
ngpus = 3
n = 8
 Input data: (keys, values)
            5    0.7402194
            5    0.6747900
            2    0.3043255
            7    0.8955101
            1    0.1878665
            2    0.3233705
            6    0.7777888
            2    0.3012942
 time =     4.7612190E–04
 Output data: (keys, values)
            1    0.1878665
            2    0.3043255
            2    0.3233705
            2    0.3012942
```

```
5    0.7402194
5    0.6747900
6    0.7777888
7    0.8955101
```

Выгода от использования GPU видна при сортировки больших массивов, например из 256 млн пар:

```
$ ./sort_by_key.cpu $((1024 * 1024 * 256))
n = 268435456
 time =     11.67672

$ ./sort_by_key.gpu $((1024 * 1024 * 256))
ngpus = 3
n = 268435456
 time =     1.105390
```

8.2. PyCUDA

8.2.1. Введение

PyCUDA [14], [15] – это библиотека для работы с CUDA из Python, распространяемая под свободной лицензией MIT.

Python – это язык программирования с динамической типизацией и поддержкой объектно-ориентированной, структурной, функциональной, императивной и аспектно-ориентированной парадигм программирования. Python прост в изучении и использовании, для его освоения достаточно материала [16]. Python является популярным средством разработки приложений для web, ввиду таких свойств, как, например, читаемость кода, интерактивный режим разработки (без компиляции), динамическая типизация. Однако множество сфер применения значительно шире. Python используется в графике (PyOgre, PyOpenGL), распознавании изображений (Python для OpenCV, Python для OpenVIDIA), робототехнике (в Robotics Operating System), численных методах (numpy) и других областях. Наличие Python-интерфейсов у множества прикладных библиотек делает этот язык отличным средством организации взаимодействия компонентов составных программ.

Есть готовые библиотеки для научных приложений: для построения графиков на плоскости и в пространстве, для численных вычислений, MPI и другие.

PyCUDA можно установить как дополнительный пакет к существующей инсталляции Python 2.6, 2.7 или 3.0 [17]. PyCUDA доступен на всех операционных системах, где доступна разработка для CUDA: Linux, Windows и OS X.

Программы на PyCUDA можно отлаживать с помощью CUDA-GDB:

```
cuda–gdb ––args python –m pycuda.debug demo.py
```

Математические библиотеки из состава CUDA Toolkit также имеют готовые интерфейсы в окружении Python – *scikits.cuda*.

8.2.2. Простой пример работы с PyCUDA

Ниже приведен простейший пример работы с PyCUDA, в котором с помощью CUDA-ядра удваиваются все элементы заданной матрицы:

```
import pycuda.driver as cuda
import pycuda.autoinit
from pycuda.compiler import SourceModule

import numpy
a = numpy.random.randn(4,4)

a = a.astype(numpy.float32)

a_gpu = cuda.mem_alloc(a.size * a.dtype.itemsize)

cuda.memcpy_htod(a_gpu, a)

mod = SourceModule("""
    __global__ void doublify(float *a)
    {
        int idx = threadIdx.x + threadIdx.y*4;
        a[idx] *= 2;
    }
    """)

func = mod.get_function("doublify")
func(a_gpu, block=(4,4,1))

a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled, a_gpu)
print "original array:"
print a
print "doubled with kernel:"
```

```
print a_doubled
```

Программа начинается с подключения всех необходимых для работы модулей. Далее с помощью библиотеки numpy создается случайная матрица 4×4. PyCUDA использует ту же программную модель, что и CUDA, вследствие чего в примере происходит явное выделение памяти на GPU и копирование в нее данных. Затем создается объект типа Module, который ассоциирован с модулем CUBIN. В качестве аргумента SourceModule в тройных кавычках в виде символьной строки передается программный код ядра на CUDA C. Уже во время исполнения этот код передается компилятору nvcc, который и создает CUBIN (если его еще нет в кэше). Функции из модуля можно получить по символьному имени и затем вызвать. Обработанные на GPU данные копируются обратно на хост и выводятся в консоль.

Запустить эту программу можно либо непосредственно вводом кода в интерпретатор Python, либо, поместив в отдельный файл, командой:

```
python example1.py
```

Первый пример показывает, как PyCUDA может быть использован для программирования хост-части CUDA-программы, в то время как вид CUDA-ядер не претерпел никаких изменений. Таким образом, в отличие от обычной компилируемой программы, программа на PyCUDA состоит из двух частей: GPU-часть попрежнему компилируется (рис. 8.8), а для «склейки» низкоуровневых высокопроизводительных блоков применяется *интерпретируемый* скриптовый язык (рис. 8.9). Вместе эти две части средствами Python объединены в гибридную программу (рис. 8.10), что может упростить разработку, не ухудшая при этом производительность ядер. За счет дополнительной интеграции CUDA API, все возникающие ошибки преобразуются в исключения Python.

8.2.3. Модуль gpuarray и взаимодействие с NumPy

NumPy – это модуль Python для работы с многомерными массивами. Основным типом данных в нем является многомерный массив – *narray*. Его функционально совместимым аналогом в PyCUDA является *gpuarray*. Хранение и обработка *gpuarray* происходит на GPU. Передача данных *gpuarray* производится функциями:

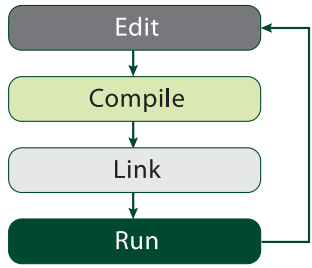


Рис. 8.8. Процесс создания компилируемой программы

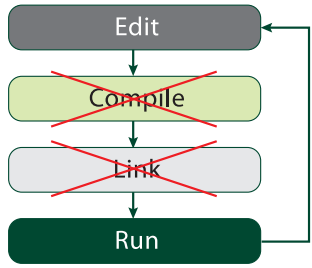


Рис. 8.9. Процесс создания интерпретируемой программы

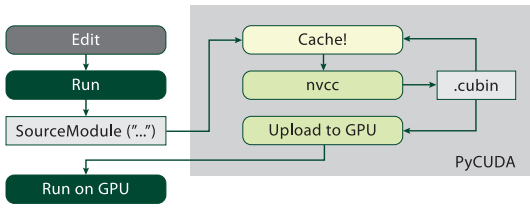


Рис. 8.10. Процесс разработки на PyCUDA

- gpuarray.to_gpu(numpy_array) – для записи в gpuarray;
- numpy_array = gpuarray.get() – для чтения из gpuarray.

gpuarray обладает следующими свойствами:

- доступно множество операций, среди которых: +, –, *, /, fill, sin, exp, rand;
- поддержка операций над разными типами (например, в результате сложения многомерных массивов с типами int32 и float32 получится многомерный массив типа float64);
- возможность использования низкоуровневого представления *gpuarray* в качестве обычного массива в CUDA-ядрах;
- поддержка вызова print gpuarray при отладке приложения.

С помощью *gpuarray* первый пример может быть реализован проще:

```
import pycuda.gpuarray as gpuarray
import pycuda.autoinit
import numpy

a_gpu = gpuarray.to_gpu(numpy.random.randn(4,4).astype(numpy.float32))
a_doubled = (2*a_gpu).get()

print "original array:"
print a_gpu
print "doubled with gpuarray:"
print a_doubled
```

В этом примере случайная матрица 4×4 создается в памяти GPU как *gpuarray*, умножается на скаляр в одну строчку и выводится в консоль.

Еще одной важной возможностью *gpuarray* является простой механизм реализации составных операций. Например, операция поэлементного преобразования transform реализуется так:

```
import pycuda.gpuarray as gpuarray
import pycuda.autoinit
import numpy
from pycuda.curandom import rand as curand
```

```
a_gpu = curand((50,))
b_gpu = curand((50,))

from pycuda.elementwise import ElementwiseKernel
lin_comb = ElementwiseKernel(
        "float a, float *x, float b, float *y, float *z",
        "z[i] = a*x[i] + b*y[i]",
        "linear_combination")

c_gpu = gpuarray.empty_like(a_gpu)
lin_comb(5, a_gpu, 6, b_gpu, c_gpu)

import numpy.linalg as la
assert la.norm((c_gpu - (5*a_gpu+6*b_gpu)).get()) < 1e-5
```

В данном примере создается линейная комбинация, которая применяется к двум массивам. Вместо вывода на экран используется проверка допустимой нормы различия результатов. Весь необходимый для работы код PyCUDA генерирует во время исполнения программы.

Аналогичным образом реализуется преобразование transform-reduce (map-reduce) для поиска скалярного произведения двух векторов:

```
import pycuda.gpuarray as gpuarray
import pycuda.autoinit
import numpy
from pycuda.curandom import rand as curand
from pycuda.reduction import ReductionKernel

dot = ReductionKernel(
    dtype_out=numpy.float32,
    neutral = "0",
    reduce_expr = "a + b",
    map_expr = "x[i] * y[i]",
    arguments = "const float *x, const float *y")

x = curand((1000*1000), dtype=numpy.float32)
y = curand((1000*1000), dtype=numpy.float32)

x_dot_y = dot(x, y).get()
x_dot_y_cpu = numpy.dot(x.get(), y.get ())
```

Здесь создается ядро преобразования *dot* с выражениями для трансформации (*map_expr*) и редукции (*reduce_expr*), а также нейтральным элементом (*neutral*), для операции сложения равным нулю.

Глава 9

Анализ работы приложений на GPU

9.1. Профилирование

9.1.1. CUDA events

При разработке CUDA-приложений важное значение имеет возможность точно замерять время выполнения различных операций на GPU. Для этой цели в CUDA API введен объект *событие* (CUDA event) типа *cudaEvent_t*. Соответствующие функции позволяют создавать и уничтожать события, обозначать позиции начала и конца анализируемого фрагмента кода, проверять возникновение определенного события или ожидать его наступления, а также получать время в миллисекундах, прошедшее между двумя событиями. Ниже приведен пример кода, замеряющий время выполнения ядра на GPU:

```
// Объявление переменных-событий начала и окончания
// выполнения ядра.
cudaEvent_t start, stop;
float gpuTime = 0.0f;

// Инициализация переменных-событий.
cudaEventCreate ( &start );
cudaEventCreate ( &stop );

// Привязка события start к данной позиции в коде
// программы (начало выполнения ядра).
cudaEventRecord ( start, 0 );

// Запуск GPU-ядра.
myKernel<<<blocks, threads>>> ( adev, bdev, N, cdev );

// Привязка события stop к данной позиции в коде
// программы (окончание выполнения ядра).
cudaEventRecord ( stop, 0 );

// Ожидание окончания выполнения ядра,
// синхронизация по событию stop.
cudaEventSynchronize ( stop );

// Получение времени, прошедшего между событиями start и stop.
```

```
cudaEventElapsedTime ( &gpuTime, start, stop );

printf("time spent executing on the GPU: %.2f millseconds\n", gpuTime );

// Уничтожение событий.
cudaEventDestroy ( start );
cudaEventDestroy ( stop  );
```

9.1.2. CUDA profiler

Профилирование – это процесс сбора сведений о параметрах работы приложения для последующего анализа, например, с целью повышения эффективности. Возможности профилирования так или иначе связаны с наличием аппаратных счетчиков событий. На GPU, в зависимости от compute capability, реализованы различные наборы счетчиков (см. Приложение Б). Одновременно может быть использовано не более 4 аппаратных счетчиков.

CUDA runtime содержит встроенный профилировщик, активация которого может быть произведена заданием переменной окружения *COMPUTE_PROFILE=1* перед запуском целевого приложения. Результаты профилирования будут по умолчанию записаны в файл *cuda_profile_%d.log*, где *%d* – индекс GPU, или в файл, имя которого содержит переменная окружения *COMPUTE_PROFILE_LOG*. Набор активных счетчиков может быть передан через файл, заданный переменной окружения *COMPUTE_PROFILE_CONFIG*. Строки этого файла могут содержать имена счетчиков или комментарии после символа «#». В дополнение к аппаратным счетчикам, профилировщик предоставляет параметры, приведенные в таблице 9.1. Например, настроив профилировщик на получение количества запросов на чтение и запись из глобальной памяти,

```
$ export COMPUTE_PROFILE=1
$ export COMPUTE_PROFILE_CONFIG=cuda_profile.cfg
$ cat cuda_profile.cfg
gld_request
gst_request
```

для теста перемножения матриц *matmul2* можно получить следующие результаты:

```
$ matmul/matmul2 2048
n = 2048
time spent executing kernel: 93.73 millseconds
```

Таблица 9.1. Дополнительные параметры профилировки, доступные в CUDA profiler

Имя	**Описание**
timestamp	Метка времени моментов запуска ядер и обменов данными, в микросекундах
gpustarttimestamp	Метка времени старта работы ядра на GPU
gpuendtimestamp	Метка времени окончания работы ядра на GPU
gridsize	Размерности сетки блоков по X и Y
gridsize3d	Размерности сетки блоков по X, Y и Z
threadblocksize	Размерности сетки нитей в блоке по X, Y и Z
dynsmemperblock	Количество динамически выделяемой разделяемой памяти на блок, в байтах
stasmemperblock	Количество статически выделяемой разделяемой памяти на блок, в байтах
regperthread	Количество регистров на нить
memtransferdir	Направление передачи данных: 0 – от хоста устройству, 1 – от устройства хосту
memtransfersize	Размер буфера данных при копировании, в байтах
memtransferhostmem	Тип памяти при копировании: обычная или pinned
streamid	Номер очереди команд (CUDA stream) для запуска ядра
cacheconfigrequested	Конфигурация кэша, запрошенная для ядра: 0 – безразличная, 1 – больше разделяемой памяти, 2 – больше кэша, 3 – равные размеры кэша и разделяемой памяти
cacheconfigexecuted	Конфигурация кэша, использованная для ядра (см. cacheconfigrequested)

```
time spent executing cublasSgemm_v2: 32.80 millseconds
$ cat cuda_profile_0.log
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 Tesla C2070
# CUDA_CONTEXT 1
# TIMESTAMPFACTOR fffff6d9efc41ee8
method,gputime,cputime,occupancy,gld_request,gst_request
method=[ memcpyHtoD ] gputime=[ 6074.208 ] cputime=[ 6260.000 ]
```

```
method=[ memcpyHtoD ] gputime=[ 5630.656 ] cputime=[ 5712.000 ]
method=[ _Z7matmul2PfS_iS_ ] gputime=[ 93172.797 ] cputime=[ 93207.000 ] ↩
    occupancy=[ 0.667 ]
gld_request=[ 1200128 ] gst_request=[ 9376 ]
method=[ memcpyDtoH ] gputime=[ 8205.632 ] cputime=[ 9982.000 ]
method=[ memcpyHtoD ] gputime=[ 1.280 ] cputime=[ 36.000 ]
method=[ _Z24fermiSgemm_v2_kernel_valILb1ELb1ELb1EEvPfPKfS2_iiiiiiffii ] ↩
    gputime=[ 26447.359 ]
cputime=[ 26492.004 ] occupancy=[ 0.333 ] gld_request=[ 0 ] gst_request=[ ↩
    8928 ]
method=[ _Z22gemm_kernel1x1_tex_valIfLb1ELb0ELb0ELb1ELb1EEvPT_iiiiiiiiS0_S0_ ↩
    ]
gputime=[ 1391.616 ] cputime=[ 1425.000 ] occupancy=[ 1.000 ] gld_request=[ 0↩
     ] gst_request=[ 144 ]
method=[ _Z22gemm_kernel1x1_tex_valIfLb1ELb0ELb0ELb1ELb1EEvPT_iiiiiiiiS0_S0_ ↩
    ]
gputime=[ 1391.680 ] cputime=[ 1426.000 ] occupancy=[ 1.000 ] gld_request=[ 0↩
     ] gst_request=[ 144 ]
method=[ ↩
    _Z18gemm_kernel1x1_valIfLb1ELb0ELb0ELb1ELb1EEvPT_PKS0_S3_iiiiiiS0_S0_ ]
gputime=[ 176.608 ] cputime=[ 204.000 ] occupancy=[ 0.667 ] gld_request=[ ↩
    2048 ] gst_request=[ 8 ]
method=[ memcpyDtoH ] gputime=[ 8170.560 ] cputime=[ 9908.000 ]
```

9.2. Отладка

Сложная архитектура целевой системы, такой как GPU, требует адекватных средств отладки. Отладчик (debugger) позволяет остановить программу, подобно «паузе» при воспроизведении видеозаписи, предоставляя доступ к текущему логическому состоянию и данным «кадра».

9.2.1. Принципы и терминология

Отладчик представляет собой командную среду запуска приложения с дополнительными функциями. В консольном режиме среда отладчика также является консолью и позволяет целевому приложению работать обычным образом, без каких-либо ограничений. При возникновении ошибки или заданного пользователем события программа останавливается, продолжая быть резидентной в памяти. В зависимости от условий, работа программы может быть продолжена по команде или запущена заново в фатальном случае, например, необработанной ошибки сег-

ментации (segmentation fault). В положении останова отладчик представляет состояние приложение как *трассу* – путь в графе вызовов, начиная с самого внешнего уровня, обычно $main$ – точки входа, и заканчивая самой вложенной из вызывавших друг друга функций, в которой произошло событие. Состояние программы в контексте одной из функций называется *фреймом*. Другими словами, трасса – это совокупность фреймов. Среда отладчика позволяет просматривать значения переменных и вычислять несложные выражения, назначать условные и безусловные точки останова, перемещаться по стеку фреймов, отлаживать несколько потоков или анализировать *дамп* (core dump) – информацию о состоянии приложения, сохраненную в файл.

В ОС Linux приложения и динамические библиотеки используют формат ELF (Executable and Linkable Format). Исполняемый образ в формате ELF состоит из секций элементов, сгруппированных по назначению, например, секция $.text$ содержит исполняемый код, $.data$ и $.bss$ – данные. Также существует специальная секция $.debug$ для хранения *отладочной информации*. Секция $.debug$ может содержать таблицы соответствия декорированных имен реальным именам (например, для функций на C++), соответствие адресов строкам исходного кода и любую другую дополнительную информацию. Конкретное содержимое отладочной секции зависит от компилятора, а возможность ее использования – от отладчика. Наличие отладочной информации практически не влияет на производительность кода, однако оптмизации снижают качество отладки: соответствие строк кода адресам может быть неточным, значения некоторых промежуточных переменных могут быть недоступны и т.д.

9.2.2. GDB

Отладчик GDB работает в командной строке. Пользоваться такой системой проще, чем графическими редакторами, благодаря поддержке сокращенных до одной-двух букв команд и автозаполнения имен функций и переменных по клавише “Tab”. Отладочная информация добавляется в программу на стадии компиляции при наличии флага “-g”:

```
$ cd multigpu/pthread/pthread_cuda/
$ make
gcc —g —std=c99 —I/opt/cuda/include —c pthread_cuda.c —o pthread_cuda.o
```

```
nvcc –g –c pattern2d.cu –o pattern2d.o
gcc pthread_cuda.o pattern2d.o –o pthread_cuda –lpthread –L/opt/cuda/lib64 –↩
    lcudart –lm
```

После компиляции программа может быть запущена в отладчике GDB, если указать путь к ней в качестве аргумента:

```
[marcusmae@T61p pthread_cuda]$ gdb ./pthread_cuda
GNU gdb (GDB) Fedora (7.2–51.fc14)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64–redhat–linux–gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/marcusmae/Samples/cuda–training/samples/multigpu/↩
    pthread/pthread_cuda/
pthread_cuda...done.
(gdb)
```

Второй вариант – подключить отладчик к ранее запущенной программе, указав в качестве второго аргумента идентификатор процесса:

```
[marcusmae@T61p pthread_cuda]$ gdb pthread_cuda 3346
Reading symbols from /home/marcusmae/Samples/cuda–training/samples/multigpu/↩
    pthread/pthread_cuda/
pthread_cuda...done.
Attaching to program: /home/marcusmae/Samples/cuda–training/samples/multigpu/↩
    pthread/pthread_cuda/
pthread_cuda, process 3346
(gdb)
```

После запуска отладчика начать или продолжить работу приложения можно, соответственно, с помощью команд "r" ("run") и "c" ("continue"). До начала работы можно также задать точку останова (breakpoint) командой "b":

```
(gdb) b pattern2d_cpu
Breakpoint 1 at 0x401675: file pattern2d.cu, line 34.
(gdb) r
Starting program: /home/marcusmae/Samples/cuda–training/samples/multigpu/↩
    pthread/pthread_cuda/
pthread_cuda
[Thread debugging using libthread_db enabled]
1 CUDA device(s) found
```

```
[New Thread 0x7ffff6f44700 (LWP 3349)]

Breakpoint 1, pattern2d_cpu (bx=1, nx=128, ex=1, by=1, ny=128, ey=1,
    in=0x7ffff6f65010, out=0x7ffff6f55010, id=1) at pattern2d.cu:34
34      size_t size = nx * ny * sizeof(float);
(gdb)
```

Здесь breakpoint был установлен на первую строчку функции $pattern2d\_cpu$. Аналогичным образом breakpoint задается по номеру строки в текущем файле или по сочетанию «имя_файла:номер_строки»:

```
(gdb) b 36
Breakpoint 2 at 0x40169b: file pattern2d.cu, line 36.
(gdb) b pattern2d.cu:37
Note: breakpoint 2 also set at pc 0x40169b.
Breakpoint 3 at 0x40169b: file pattern2d.cu, line 37.
(gdb) c
Continuing.

Breakpoint 2, pattern2d_cpu (bx=1, nx=128, ex=1, by=1, ny=128, ey=1,
    in=0x7ffff6f65010, out=0x7ffff6f55010, id=1) at pattern2d.cu:37
37      for (int j = by; j < ny - ey; j++)
(gdb)
```

Рассмотренный выше breakpoint является *безусловным*, т.е. срабатывающим всякий раз в момент исполнения заданной строчки. Для отбора нужных ситуаций может быть дополнительно указано условие срабатывания breakpoint:

```
(gdb) condition 1 nx==128
```

Существует еще один важный элемент отладки – *watchpoint*. Он позволяет прервать программу в момент изменения содержимого памяти по заданному адресу. Этот элемент особенно полезен при отладке многопоточных приложений и выходов за границы диапазонов массивов.

```
(gdb) b 38
Breakpoint 4 at 0x4016a6: file pattern2d.cu, line 38.
(gdb) c
Continuing.

Breakpoint 4, pattern2d_cpu (bx=1, nx=128, ex=1, by=1, ny=128, ey=1,
    in=0x7ffff6f55010, out=0x7ffff6f65010, id=1) at pattern2d.cu:38
38          for (int i = bx; i < nx - ex; i++)
(gdb) p &j
$1 = (int *) 0x7fffffffdf3c
```

```
(gdb) wa *(int*)0x7fffffffdf3c
Hardware watchpoint 5: *(int*)0x7fffffffdf3c
(gdb) c
Continuing.
Hardware watchpoint 5: *(int*)0x7fffffffdf3c

Old value = 1
New value = 2
0x00000000004017a2 in pattern2d_cpu (bx=1, nx=128, ex=1, by=1, ny=128, ey=1,
    in=0x7ffff6f55010, out=0x7ffff6f65010, id=1) at pattern2d.cu:37
37      for (int j = by; j < ny - ey; j++)
```

С помощью команды “info th” (“info threads”) можно отобразить все действующие в приложении потоки. Текущий по отношению к отладчику поток помечен символом «*». Переходить из одного потока в другой позволяет команда “t” (“thread”):

```
(gdb) info th
  2 Thread 0x7ffff6f44700 (LWP 3349)  0x0000003befed8997 in ioctl ()
   from /lib64/libc.so.6
* 1 Thread 0x7ffff786c740 (LWP 3346)  0x00000000004017a2 in pattern2d_cpu (
    bx=1, nx=128, ex=1, by=1, ny=128, ey=1, in=0x7ffff6f55010,
    out=0x7ffff6f65010, id=1) at pattern2d.cu:37
(gdb) t 2
[Switching to thread 2 (Thread 0x7ffff6f44700 (LWP 3349))]#0  0↩
    x0000003befed8997 in ioctl ()
from /lib64/libc.so.6
(gdb)
```

Все потоки и работают, и останавливаются при достижении точки останова в любом потоке одновременно. Поэтому, если переключиться в другой поток после останова, то его текущая команда может находиться за пределами кода основной программы. Ниже приведен вывод трассы – “bt” (“backtrace”), который показывает, что поток в данный момент работает с драйвером устройства (вызов ioctl):

```
(gdb) bt
#0  0x0000003befed8997 in ioctl () from /lib64/libc.so.6
#1  0x00007ffff710b923 in ?? () from /usr/lib64/libcuda.so
...
#10 0x00007ffff70e7b41 in ?? () from /usr/lib64/libcuda.so
#11 0x00007ffff7daa82e in ?? () from /opt/cuda/lib64/libcudart.so.4
#12 0x00007ffff7daad7b in ?? () from /opt/cuda/lib64/libcudart.so.4
#13 0x00007ffff7dab764 in ?? () from /opt/cuda/lib64/libcudart.so.4
#14 0x00007ffff7da09b6 in ?? () from /opt/cuda/lib64/libcudart.so.4
```

```
#15 0x00007ffff7d98659 in ?? () from /opt/cuda/lib64/libcudart.so.4
#16 0x00007ffff7dbdf88 in cudaMalloc () from /opt/cuda/lib64/libcudart.so.4
#17 0x0000000000400dea in thread_func (arg=0x611cf0) at pthread_cuda.c:68
#18 0x0000003bf0206ccb in start_thread () from /lib64/libpthread.so.0
#19 0x0000003befee0c2d in clone () from /lib64/libc.so.6
(gdb)
```

Переключение текущего фрейма по индексу или командами “up” и “do” (следующий и предыдущий), переводит область видимости с одной функции в другую. Поскольку программа остановлена в текущем состоянии, переход между фреймами обратим.

```
(gdb) f 17
#17 0x0000000000400dea in thread_func (arg=0x611cf0) at pthread_cuda.c:68
68      cuda_status = cudaMalloc((void**)&config->in_dev, size);
(gdb) list
63      }
64
65      size_t size = config->nx * config->ny * sizeof(float);
66
67      // Create device arrays for input and output data.
68      cuda_status = cudaMalloc((void**)&config->in_dev, size);
69      if (cuda_status != cudaSuccess)
70      {
71          fprintf(stderr, "Cannot allocate CUDA input buffer on device %d, ↩
    status = %d\n",
72              idevice, cuda\_status);
(gdb)
```

В контексте каждой функции существуют свои переменные, отдельные значения которых можно вывести командой “p имя” (“print имя”), или в случае массива – диапазоном, например, “p name[3]@32” выведет 32 элемента, начиная с name[3]. Также можно вывести поля структуры, переданной по указателю, добавив «*», например, “p *name”.

Любой отладчик должен обеспечивать возможность навигации по исходному коду. В случае gdb, первый вариант – использовать команду “list” (листинг). Эта команда позволяет вывести по умолчанию 10 строк кода, окружающих текущую позицию останова, или другое число строк, если в команде задан целый параметр. Повторное нажатие “Enter” выведет следующие несколько строк.

```
(gdb) list
32      float* in, float* out, int id)
```

```
33  {
34      size_t size = nx * ny * sizeof(float);
35      memset(out, 0, size);
36
37      for (int j = by; j < ny - ey; j++)
38          for (int i = bx; i < nx - ex; i++)
39              OUT(i,j) = sqrtf(fabs(IN(i,j) + IN(i-1,j) + IN(i+1,j) -
40                  2.0f * IN(i,j-1) + 3.0f * IN(i,j+1)));
41      return 0;
(gdb)
```

Второй вариант – использовать терминальный редактор (TUI) с прокруткой строк клавишами «вверх» и «вниз», доступный по сочетанию клавиш Ctrl-X+A:

```
      pattern2d.cu
    33      {
B+  34              size_t size = nx * ny * sizeof(float);
    35              memset(out, 0, size);
B+  36
B+> 37              for (int j = by; j < ny - ey; j++)
    38                      for (int i = bx; i < nx - ex; i++)
    39                              OUT(i,j) = sqrtf(fabs(IN(i,j) + IN(i-
    40                                      2.0f * IN(i,j-1) + 3.0f * IN(
    41              return 0;
    42      }
    43
    44      // GPU device kernel.
    45      __global__ void pattern2d_gpu_kernel(
i,j+1)
multi-thre Thread 0x7ffff In: pattern2d_cpu             Line: 37   PC: 0x
(gdb)
```

Использование простых терминалов часто лучше подходит для встроенных систем или удаленных серверов, позволяя выполнить полноценную отладку минимальными средствами. Тем не менее, в Linux отладчики также подключаются и к визуальным средам (рис. 9.1), например, DDD или Emacs. Кроме того, поддерживается режим отладки «клиент-сервер», при котором удаленное устройство может, к примеру, запускать серверную часть в терминальном режиме, а локальная машина разработчика – визуальный клиент.

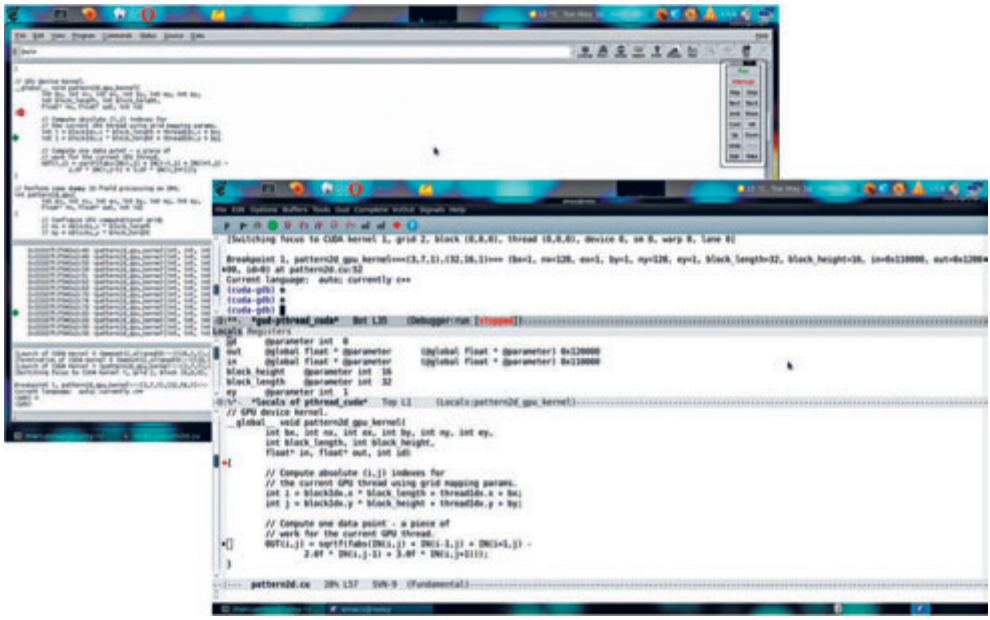


Рис. 9.1. Использование CUDA-GDB в визуальных средах DDD и Emacs

9.2.3. CUDA-GDB

CUDA-GDB – это вариант GDB с поддержкой отладки в cuda-ядрах. CUDA-GDB так же, как GDB, распространяется по лицензии GPL и доступен в исходных кодах [3]. Для работы CUDA-GDB в системе достаточно иметь хотя бы один свободный GPU. Если единственный GPU системы используется для работы X Window System, то этот GPU занят, и для отладки его использовать нельзя. Однако, если графическую подсистему выключить, то обычный ноутбук с одним GPU становится вполне пригоден для ведения отладки. Поскольку CUDA-GDB в общем случае предназначен для отладки multi-gpu приложений, одновременно может быть запущено не более одного сеанса CUDA-GDB.

В случае CUDA-GDB включение отладочной информации в хост-программу по-прежнему производится при наличии флага “-g” (строчная буква). Отладочная информация для GPU-ядер добавляется отдельно по флагу “-G” (заглавная буква). Как и в случае GDB, возможности отладки GPU-ядер сильно ограничены при наличии оптимизаций.

```
$ make
nvcc —g —G —c pattern2d.cu
gcc —g —std=c99 —c —I/usr/include/ImageMagick draw_diffs.c
nvcc —g pattern2d.o draw_diffs.o —o pattern2d —lMagickCore
```

Точки останова в GPU-ядрах работают точно так же, как в GDB:

```
(cuda—gdb) b pattern2d_gpu_kernel
Breakpoint 1 at 0x4032ba: file pattern2d.cu, line 38.
(cuda—gdb) r
Starting program: /home/dmikushin/Programming/cuda—training/samples/multigpu/↩
    pthread/pthread_cuda/
pthread_cuda
[Thread debugging using libthread_db enabled]
[New process 26320]
[New Thread 139849021003584 (LWP 26320)]
8 CUDA device(s) found
[New Thread 139849009342208 (LWP 26324)]
[New Thread 139849000949504 (LWP 26325)]
[New Thread 139848992556800 (LWP 26326)]
[New Thread 139848984164096 (LWP 26327)]
[New Thread 139848975771392 (LWP 26328)]
[New Thread 139848899884800 (LWP 26329)]
[New Thread 139848891492096 (LWP 26330)]
[New Thread 139848883099392 (LWP 26331)]
Device 6 initialized
Device 0 initialized
Device 7 initialized
```

Дополнительно выводится информация о запуске ядер на соответствующих GPU:

```
Device 5 initialized
Device 3 initialized
Device 1 initialized
[Launch of CUDA Kernel 0 (pattern2d_gpu_kernel) on Device 6]
Device 4 initialized
Device 2 initialized
[Launch of CUDA Kernel 1 (pattern2d_gpu_kernel) on Device 1]
[Termination of CUDA Kernel 0 (pattern2d_gpu_kernel) on Device 6]
[Launch of CUDA Kernel 2 (pattern2d_gpu_kernel) on Device 2]
[Launch of CUDA Kernel 3 (pattern2d_gpu_kernel) on Device 7]
[Launch of CUDA Kernel 4 (pattern2d_gpu_kernel) on Device 4]
[Termination of CUDA Kernel 3 (pattern2d_gpu_kernel) on Device 7]
[Launch of CUDA Kernel 5 (pattern2d_gpu_kernel) on Device 0]
[Launch of CUDA Kernel 6 (pattern2d_gpu_kernel) on Device 7]
[Switching to CUDA Kernel 5 (<<<(0,0),(0,0,0)>>>)]
```

```
Breakpoint 1, pattern2d_gpu_kernel<<<(3,7),(32,16,1)>>> (bx=1, nx=128,
    ex=1, by=1, ny=128, ey=1, block_length=32, block_height=16,
    in=0x100000, out=0x110000, id=0) at pattern2d.cu:52
52      int i = blockIdx.x * block_length + threadIdx.x + bx;
```

Помимо переменных фрейма, в ядрах всегда доступны значения встроенных переменных CUDA, например $blockIdx$:

```
(cuda-gdb) b 57
Breakpoint 2 at 0x7fdbac085e98: file pattern2d.cu, line 57.
(cuda-gdb) c
Continuing.
[Termination of CUDA Kernel 8 (pattern2d_gpu_kernel) on Device 4]

Breakpoint 2, pattern2d_gpu_kernel<<<(3,7),(32,16,1)>>> (bx=1, nx=128,
    ex=1, by=1, ny=128, ey=1, block_length=32, block_height=16,
    in=0x100000, out=0x110000, id=0) at pattern2d.cu:57
57      OUT(i,j) = sqrtf(fabs(IN(i,j) + IN(i-1,j) + IN(i+1,j) -
(cuda-gdb) p blockIdx
$1 = {x = 0, y = 0}
```

Подобно тому, как для CPU можно переключаться между потоками (threads), CUDA-GDB поддерживает переключение текущего блока и нити GPU-ядра. В случае безусловной точки останова срабатывание происходит для произвольной нити. Из текущей нити можно переключиться в любую другую нить:

```
(cuda-gdb) cuda block 2,1 thread 15,4,0
[Switching to CUDA Kernel 1 (device 0, sm 15, warp 4, lane 15, grid 2, block ↩
    (2,1), thread (15,4,0))]
57      OUT(i,j) = sqrtf(fabs(IN(i,j) + IN(i-1,j) + IN(i+1,j) -
(cuda-gdb) p i
$2 = 80
(cuda-gdb) p j
$3 = 21
```

Рассмотренный обзор GDB и CUDA-GDB имел цель дать представление о ключевых возможностях отладки хост- и GPU-кода. Изложенные принципы и терминология будут справедливы для любой системы отладки.

9.3. Диагностика

9.3.1. CUDA-MEMCHECK

Полезным средством диагностики ошибок является CUDA-MEMCHECK. Подключаясь к приложению, CUDA-MEMCHECK сообщает подробную информацию об ошибках доступа к памяти в GPU-ядрах. В следующем примере GPU-ядро пытается использовать адрес памяти, находящийся за пределами выделенного массива:

```
$ cuda—memcheck ./pthread_cuda
========= CUDA—MEMCHECK
1 CUDA device(s) found
Device 0 initialized
Cannot execute CUDA kernel #2 by device 0, status = 4
Cannot execute pattern 2d on device 0, status = 4
========= Invalid __global__ read of size 4
=========     at 0x00000170 in pattern2d_gpu_kernel
=========     by thread (0,0,0) in block (1,0,0)
=========     Address 0x00100ef4 is out of bounds
=========
========= ERROR SUMMARY: 1 error
```

Утилита CUDA-MEMCHECK доступна и как отдельное приложение, и как подключаемая компонента отладчика CUDA-GDB:

```
(cuda—gdb) set cuda memcheck on
```

Глава 10

Использование нескольких GPU

В этой главе рассматриваются некоторые методы организации работы приложений, использующих множество GPU. Современная типовая рабочая станция или процессорный узел кластерной системы обычно имеет структуру, показанную на рис. 10.1: два и более сокетов с многоядерными CPU, соединенных мостами, неоднородная оперативная память и одна или более шин периферийных устройств PCI-Express. Каждая шина содержит фиксированное количество линий обмена данными, не более 40 (для CPU Intel) или 48 (для CPU AMD). Хотя максимальное число *физически* подключаемых к шине GPU ограничено наличием слотов, реальная скорость обмена данными между GPU и хостом будет зависеть именно от числа доступных линий. Например, серверное решение NVIDIA Tesla S2050 с 4 GPU в корпусе 1U подключается к двум слотам PCI-Express. В таком случае 4 устройствам в сумме доступно вдвое меньше линий, чем при раздельном подключении, что может иметь эффект на пропускную способность при одновременном интенсивном обмене данными.

В случае кластерной системы к локальному расслоению памяти добавляются различные случаи взаимодействия между узлами и их GPU, например, пересылка данных из памяти хоста в глобальную память GPU другого хоста. В свою очередь, межузловая обменная сеть большого кластера сама может быть неоднородной иерархией малых сетей. Гибкий учет возникающих дисбалансов соединений на уровне приложения может быть довольно сложен.

В каждой новой версии CUDA API постоянно пополняется новыми функциями, тем не менее, они предназначены только для управления локальными GPU. CUDA работает в своей нише и поэтому не может контролировать работу распределенного кластера. В этом случае CUDA необходимо сочетать с другими программными моделями.

Основные единицы исполнения в операционных системах – это процессы и потоки. Процессы специфичны наличием контекста в системе и собственным адресным пространством. Обращаться напрямую к памяти другого процесса запрещено. Между GPU и процессами можно установить некоторое соответствие (рис. 10.2).

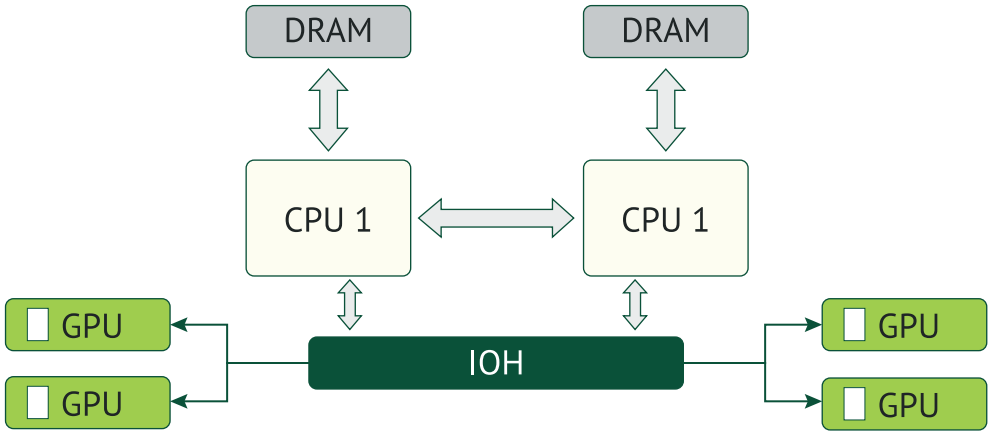


Рис. 10.1. Узел целевой кластерной системы для multi-GPU-приложений

Потоки же могут быть реализованы как на уровне системы, так и на микроуровне, и могут совместно использовать память родительского процесса, что упрощает обмен данными. Как и в случае процессов, каждый GPU системы можно связать с соответствующим рабочим потоком (рис. 10.3).

10.1. Контекст устройства

Контекст устройства – это ключевой термин при управлении несколькими GPU. Контекст представляет собой контейнер с управляющей информацией, характеризующей состояние конкретного GPU. Большинство вызовов CUDA API требует существования контекста. Изначально контекста не существует, но он будет создан неявно при первом вызове функции CUDA API. Таким образом, можно писать CUDA-программы для одного GPU и никогда не узнать о существовании контекстов. Но в случае использования нескольких GPU из одного потока переключение контекстов вручную является единственным способом организовать работу.

Фактически процессы и потоки могут управлять GPU только опосредованно, контролируя контексты. Их можно создавать, удалять и выбирать текущий активный контекст из нескольких существующих. В каждый момент времени в каждом потоке хоста может быть не более одного текущего контекста.

Управление контекстами может быть организовано средствами CUDA API (вы-

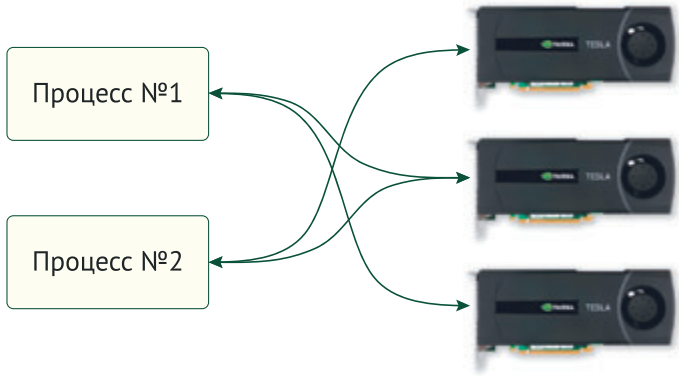


Рис. 10.2. Управление несколькими GPU на уровне отдельных процессов

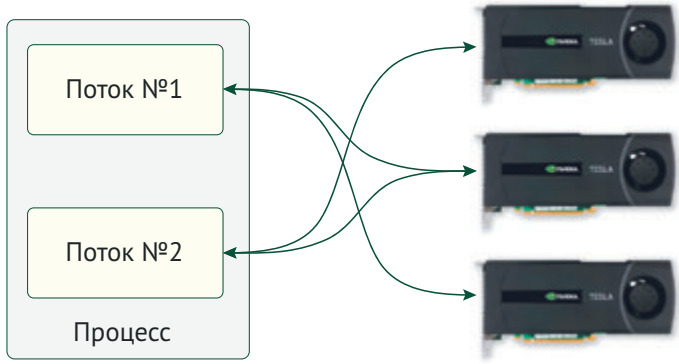


Рис. 10.3. Управление несколькими GPU потоками одного процесса

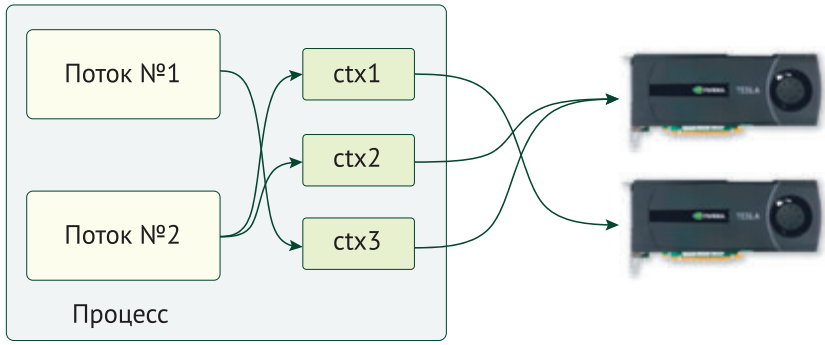


Рис. 10.4. Взаимодействие программы и GPU посредством CUDA-контекстов

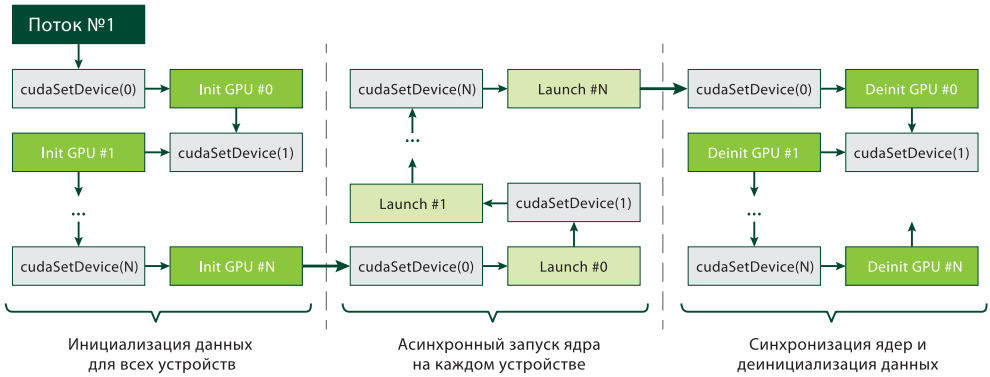


Рис. 10.5. Схема работы примера № 1

зов cudaSetDevice) или CUDA Driver API (вызовы с префиксом cuCtx).

Первый пример multi-GPU имеет важную цель – продемонстрировать возможность использования нескольких GPU из **одного** хост-потока. За счет механизма переключения контекстов устройств и асинхронных вызовов последовательная хост-программа может содержать части, распараллеленные между несколькими GPU. Управление работой GPU можно разделить на 3 этапа (рис. 10.5).

Первый этап – это цикл по всем устройствам, содержащий код инициализации – копирование входных данных и проч. Если время вычислений велико, то этот цикл может быть последовательным:

```
// Цикл инициализации устройств.
for (int idevice = 0; idevice < ndevices; idevice++)
{
    config_t* config = configs + idevice;
    // Установить текущее устройство.
    cuda_status = cudaSetDevice(idevice);

    // Создать массивы для входных и выходных данных.
    cudaError_t cuda_status = cudaMalloc((void**)&config->in_dev, size);
    cuda_status = cudaMalloc((void**)&config->out_dev, size);

    // Скопировать входные данные в память GPU.
    cuda_status = cudaMemcpy(config->in_dev, input, size,
        cudaMemcpyHostToDevice);

    printf("Device %d initialized\n", idevice);
}
```

Второй этап – ключевой: запуск ядер. Точно таким же образом, последовательно переключая в каждой итерации цикла фокус на соответствующиее устройство, производится запуск ядра. Поскольку он является асинхронным вызовом, переключение в следующий контекст и запуск другого ядра будет произведен до завершения данного ядра. За счет этого механизма и достигается параллельность: несмотря на то, что ядра запущены последовательно, интервал между запусками достаточно мал, чтобы времена исполнения ядер наложились друг на друга, если только они не слишком простые:

```
// Запуск ядер. На каждом устройстве запускается
// по одному ядру.
for (int idevice = 0; idevice < ndevices; idevice++)
{
    config_t* config = configs + idevice;

    // Установить текущее устройство.
    cuda_status = cudaSetDevice(idevice);

    get_time(&config->start);

    // Запустить вычислительное ядро (асинхронно).
    int status = pattern2d_gpu(1, nx, 1, 1, ny, 1,
        config->in_dev, config->out_dev, idevice);
}
```

После того, как все ядра запущены, хост-программа может последовательно ожидать завершения работы каждого ядра и проводить необходимую деинициализацию аналогично первому этапу:

```
// Ожидать завершения всех ядер.
for (int idevice = 0; idevice < ndevices; idevice++)
{
    config_t* config = configs + idevice;

    // Установить текущее устройство.
    cuda_status = cudaSetDevice(idevice);

    // Синхронизация - ожидание завершения ядра.
    cuda_status = cudaThreadSynchronize();

    get_time(&finish);

    printf("GPU %d time = %f sec\n", idevice,
```

```
        get_time_diff(&config->start, &finish));
}
```

Степень параллельности исполнения может быть проверена сравнением времен работы отдельных ядер со временем, прошедшим с момента запуска первого ядра до синхронизации последнего:

```
printf("Total time of %d GPUs = %f\n", ndevices,
    get_time_diff(&configs[0].start, &finish));
```

В таблице 10.1 приведены времена исполнения тестовой задачи на сервере с 8 GPU. В идеальном случае общее время работы должно быть примерно равно сумме времени самого медленного ядра и времени запуска всех ядер. Полученные результаты хорошо согласуются с этой оценкой.

Таблица 10.1. Времена параллельного исполнения multi-GPU задачи отдельно для каждого из 8 GPU и общее время работы (Xeon E5620 + 2×S1060)

Счетчик	**Значение, сек**
GPU 0	0,165219
GPU 1	0,165125
GPU 2	0,165125
GPU 3	0,165073
GPU 4	0,165142
GPU 5	0,165274
GPU 6	0,165227
GPU 7	0,165180
Общее время	**0,165677**

Таким образом, для управления несколькими GPU в рамках одного хоста нет необходимости создавать параллельную хост-программу. Параллельность требуется лишь в случае распределенной кластерной системы.

10.2. Fork

Рассмотрим возможные варианты реализации multi-GPU-программы с несколькими рабочими хост-процессами. Простейший способ распараллеливания – рабо-

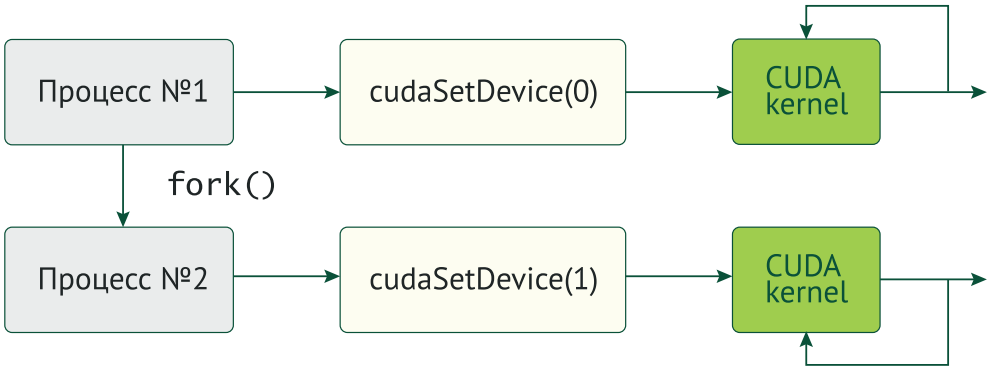


Рис. 10.6. Схема работы примера № 2

тать в каждом процессе только с одним GPU.

При явном создании процессов необходимое их число может быть порождено основным процессом. Далее каждый процесс может работать со своим GPU как независимая последовательная программа (рис. 10.6)

Для этой цели в UNIX-совместимых операционных системах существует стандартный системный вызов fork():

```
// Породить еще один рабочий процесс с помощью вызова fork().
// Стандарт: "Memory mappings created in the parent shall be
// retained in the child process."
pid_t fork_status = fork();

// From this point two processes are running the same code, if no errors.
if (fork_status == -1)
{
    fprintf(stderr, "Cannot fork process, errno = %d\n", errno);
    return errno;
}

// By fork return value we can determine the process role:
// master or child (worker).
int master = fork_status ? 1 : 0, worker = !master;

// Get the process ID.
int pid = (int)getpid();
```

Важно отметить, что хотя вызов *fork* производит полное клонирование памяти и ресурсов порождающего процесса, если контекст CUDA-устройства был создан

до вызова *fork*, то он не будет скопирован в дочерний процесс. Данное исключение было специально реализовано в CUDA, поскольку одновременное использование одного и того же контекста устройства в нескольких процессах может нарушить корректность внутренних данных.

10.3. MPI

В более общем случае параллельной программы для распределенной системы порождение процессов на вычислительных узлах производится неявно с помощью управляющих средств. Самым доступным и популярным средством является Message Passing Interface (MPI). Его реализация обычно состоит из библиотеки и системных демонов – фоновых программ, которые функционируют на каждом вычислительном узле. Единый код исполняется множеством параллельных процессов. Демоны MPI контролируют запуск, состояние и контроль ошибок в процессах, работающих в вычислительной сети. Основной функционал MPI – это организация обмена данными и синхронизация процессов при помощи отправки и приема сообщений.

Основные команды MPI: запуск (*mpirun*, *mpiexec*), инициализация/деинициализация (*MPI_Init*, *MPI_Finalize*), посылка сообщений (*MPI_Send*, *MPI_Recv* и др.) и синхронизация (*MPI_Barrier* и др.).

Для MPI, как и для предыдущего примера с вызовом *fork*, справедливо замечание о невозможности клонирования контекса CUDA-устройства, если он был создан до вызова *MPI_Init*.

В соответствии со стандартом MPI, буфер данных для приема или передачи сообщений (например, в *MPI_Send* и *MPI_Recv*) всегда должен находиться в памяти хоста. Начиная с CUDA 4.0, новейшие реализации OpenMPI и MVAPICH2 могут корректно работать и в том случае, когда в качестве буфера указан адрес в глобальной памяти GPU, позволяя исключить промежуточное копирование в память хоста.

Пусть необходимо организовать работу некоторого итерационного алгоритма на нескольких GPU с помощью MPI. После каждой итерации данные из памяти GPU должны быть переданы GPU соседнего процесса в кольцевой топологии.

Программа начинается с инициализации MPI, получения числа используемых

процессов и индекса данного процесса в глобальном коммуникаторе. Количество реально используемых процессов ограничим числом доступных GPU. В этом случае каждому процессу можно поставить в соответствие один GPU:

```
// Инициализация MPI. Начиная с этого места,
// соответствующее множество процессов будет
// выполняться параллельно.
int mpi_status = MPI_Init(&argc, &argv);
int mpi_error_msg_length;
char mpi_error_msg[MPI_MAX_ERROR_STRING];
if (mpi_status != MPI_SUCCESS)
{
    ...
}

// Получить размер глобального коммуникатора MPI
// - общее количество рабочих процессов.
int nprocesses;
mpi_status = MPI_Comm_size(MPI_COMM_WORLD, &nprocesses);

// Получить ранг (индекс) данного MPI-процесса в
// глобальном коммуникаторе.
int iprocess;
mpi_status = MPI_Comm_rank(MPI_COMM_WORLD, &iprocess);

int ndevices = 0;
cudaError_t cuda_status = cudaGetDeviceCount(&ndevices);

...

int n = atoi(argv[1]);
int npasses = atoi(argv[2]);
size_t size = n * n * sizeof(float);

// Каждому MPI-процессу поставить в соответствие один GPU.
cuda_status = cudaSetDevice(iprocess);
```

Для определенности пусть данные каждого процесса пересылаются следующему и принимаются от предыдущего. Поддержка device-адресов в командах обмена сообщениями позволяет указывать в качестве аргументов *MPI_Send* и *MPI_Recv* непосредственно адреса массивов, выделенных в памяти GPU. Если поддержка device-адресов в используемой реализации MPI отсутствует, то подобное обращение приведет к ошибке сегментации.

```
float *din1, *din2;
cuda_status = cudaMalloc((void**)&din1, size);
...
cuda_status = cudaMalloc((void**)&din2, size);
...
MPI_Request request;
int inext = (iprocess + 1) \% nprocesses;
int iprev = iprocess - 1; iprev += (iprev < 0) ? nprocesses : 0;

// Использовать GPU-память в командах пересылки MPI.
// Работает в модифицированной версии OpenMPI.
mpi_status = MPI_Isend(din1, n * n, MPI_FLOAT, inext,
    0, MPI_COMM_WORLD, &request);
mpi_status = MPI_Recv(din2, n * n, MPI_FLOAT, iprev,
    0, MPI_COMM_WORLD, NULL);
mpi_status = MPI_Wait(&request, MPI_STATUS_IGNORE);
```

Ниже приведены команды сборки реализации OpenMPI из исходного кода. Поддержка GPU-адресов включается флагом "–with-cuda". Работать эта функциональность будет только на GPU архитектуры Fermi, поскольку используются указатели на device-функции.

```
$ svn co http://svn.open-mpi.org/svn/ompi/trunk ompi-trunk
$ cd ompi-trunk/
$ ./autogen.pl
$ mkdir build
$ cd build
$ ../configure --prefix=/home/dmikushin/opt/openmpi_gcc-trunk --with-cuda
$ make install
```

Тем не менее, в версиях OpenMPI и MVAPICH2 с поддержкой данной возможности, вышедших вместе с CUDA 4.0, данные GPU все равно пересылаются в память хоста. Пересылка напрямую между GPU (peer-to-peer) была бы возможна, если бы Общее Виртуальное Адресное Пространство (Unified Virtual Address Space – UVA) было общим для всех рабочих процессов. Реализация такого расширения запланирована в следующих версиях CUDA.

10.4. POSIX-потоки

Несколькими GPU одной системы можно управлять с помощью множества потоков. В отличие от процессов, при использовании потоков упрощается обмен дан-

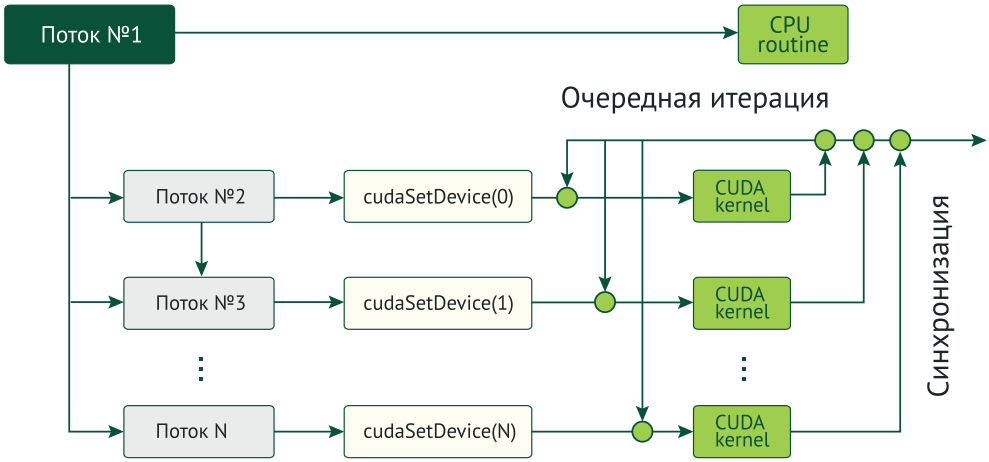


Рис. 10.7. Схема работы примера № 4

ными, доступно Общее Виртуальное Адресное Пространство (Unified Virtual Address Space – UVA), но нет возможности использовать распределенную систему из множества отдельных хостов. Многие POSIX-совместимые системы реализуют библиотеку функций для явного управления исполнением потоков – *pthread*.

С ее помощью поток создается вызовом *pthread_create*, синхронизируется и завершается вызовом *pthread_join*. Также предусмотрены различные функции синхронизации.

Пусть необходимо организовать работу некоторого итерационного алгоритма на нескольких GPU с помощью *pthread*. После каждой итерации данные из памяти GPU должны быть переданы GPU соседнего потока в кольцевой топологии посредством вызова *cudaMemcpyPeer*.

Схема реализации данного примера может быть такой. Начальный поток приложения порождает необходимое количество рабочих потоков, соответствующее числу доступных GPU. Каждому потоку с помощью вызова *cudaSetDevice* назначается GPU. Затем в цикле по итерациям каждый поток запускает CUDA-ядро, и все потоки блокируются на глобальном барьере до тех пор, пока не завершаются все ядра для данной итерации (рис. 10.7).

Основная функция примера начинается с определения числа доступных GPU, затем инициализируется примитив синхронизации «барьер». Далее с помощью вы-

зова *pthread_create* создается необходимое число потоков. Он принимает следующие параметры: идентификатор потока, набор атрибутов, указатель на функцию, вызовом которой поток начнет свою работу, и ссылка на параметр для передачи в функцию потока. Если в поток требуется передать несколько параметров, то необходимо объединить их в один, например, поместив в структуру. После того, как потоки начали работать, в следующем цикле основной поток ожидает их завершения с помощью вызова *pthread_join*.

```
#include <pthread.h>

// Тип для контейнера параметров потока.
struct thread_params_t
{
    int ithread, nthreads;
    int n, npasses;
    size_t size;

    float *din1, *din2, *dout;

    pthread_t handle;
    pthread_barrier_t* barrier;

    struct thread_params_t* prev;
};

int main(int argc, char* argv[])
{
    int nthreads = atoi(argv[1]);
    int n = atoi(argv[2]);
    int npasses = atoi(argv[3]);
    size_t size = n * n * sizeof(float);

    if ((nthreads <= 0) || (n <= 0) || (npasses <= 0)) return 0;

    int ndevices = 0;
    cudaError_t status = cudaGetDeviceCount(&ndevices);
    if (ndevices < nthreads)
    {
        fprintf(stderr, "There are less devices than working threads ↩
            requested\n");
        return 1;
    }

    // Создание контейнеров параметров потоков.
```

```
struct thread_params_t* thread_params =
    (struct thread_params_t*)malloc(
        sizeof(struct thread_params_t) * nthreads);

// Инициализация барьера для синхронизации.
pthread_barrier_t barrier;
if (pthread_barrier_init(&barrier, NULL, nthreads))
{
    fprintf(stderr, "Cannot initialize pthread barrier\n");
    return 1;
}

// Запуск потоков.
for (int ithread = 0; ithread < nthreads; ithread++)
{
    struct thread_params_t* t = thread_params + ithread;
    t->ithread = ithread;
    t->nthreads = nthreads;
    t->n = n; t->npasses = npasses; t->size = size;
    t->prev = thread_params + ithread - 1;
    t->prev += ithread ? 0 : nthreads;
    t->barrier = &barrier;

    if (pthread_create(&thread_params[ithread].handle, NULL,
        thread_func, &thread_params[ithread]))
    {
        fprintf(stderr, "Cannot create pthread thread\n");
        return 1;
    }
}

// Ожидать завершения работы потоков.
for (int ithread = 0; ithread < nthreads; ithread++)
{
    if (pthread_join(thread_params[ithread].handle, NULL))
    {
        fprintf(stderr, "Cannot join pthread thread\n");
        return 1;
    }
}

free(thread_params);

if (pthread_barrier_destroy(&barrier))
{
    fprintf(stderr, "Cannot destroy pthread barrier\n");
```

```
        return 1;
    }

    return 0;
}
```

Рассмотрим теперь функцию потока. Как и в предыдущих случаях, каждый поток может быть ассоциирован с одним GPU с помощью *cudaSetDevice*.

После инициализации входных данных каждый поток запускает CUDA-ядро в цикле и затем обменивается данными с GPU соседнего потока посредством вызова *cudaMemcpyPeer*. Поскольку он асинхронен по отношению к хосту, за ним следует синхронизация. В зависимости от задачи и от того, какой массив подлежит пересылке, может быть необходима дополнительная синхронизация между соответствующими потоками. В данном случае вызов *pthread_barrier_wait* устанавливает общий барьер для всех потоков в конце каждой итерации.

```
void* thread_func(void* arg)
{
    // Распаковать контейнер параметров.
    struct thread_params_t* current =
        (struct thread_params_t*)arg;

    // Установить соответствие между GPU и соответствующим потоком.
    cudaError_t status = cudaSetDevice(current->ithread);

    // Подключить peer-to-peer доступ к памяти GPU соседнего потока.
    cudaDeviceEnablePeerAccess(current->prev->ithread, 0);

    // Создать два входных массива и один выходной массив в памяти GPU.
    status = cudaMalloc((void**)&current->din1, current->size);
    status = cudaMalloc((void**)&current->din2, current->size);
    status = cudaMalloc((void**)&current->dout, current->size);

    // Создать входной и выходной массивы на хосте,
    // инициализировать входной массив случайными данными.
    float* hin = (float*)malloc(current->size);
    float* hout = (float*)malloc(current->size);
    double dinvrmax = 1.0 / RAND_MAX;
    for (int i = 0; i < current->n * current->n; i++)
    {
        for (int j = 0; j < current->ithread + 1; j++)
            hin[i] += rand() * dinvrmax;
        hin[i] /= current->ithread + 1;
```

```
}

// Скопировать входные данные в память GPU.
status = cudaMemcpy(current->din1, hin, current->size, ↩
    cudaMemcpyHostToDevice);

// Perform the specified number of processing passes.
for (int ipass = 0; ipass < current->npasses; ipass++)
{
    // Заполнить выходной буфер нулями.
    status = cudaMemset(current->dout, 0, current->size);

    // Запустить GPU-ядро.
    pattern2d_gpu(1, current->n, 1, 1, current->n, 1,
        current->din1, current->dout);

    // Ожидать завершения GPU-ядра.
    status = cudaDeviceSynchronize();

    // Скопировать результат в память хоста и расчитать
    // контрольное среднее значение.
    status = cudaMemcpy(hout, current->dout, current->size, ↩
        cudaMemcpyDeviceToHost);
    float avg = 0.0;
    for (int i = 0; i < current->n * current->n; i++)
        avg += hout[i];
    avg /= current->n * current->n;

    printf("Receiving thread %d resulting field with average = %f by ↩
        thread %d\n",
        current->prev->ithread, avg, current->ithread);

    // Скопировать входные данные предыдущего потока в данный поток
    // с помощью peer-to-peer доступа.
    status = cudaMemcpyPeer(current->din2, current->ithread,
        current->prev->din1, current->prev->ithread, current->prev->size)↩
            ;

    status = cudaDeviceSynchronize();

    // Синхронизировать все потоки.
    pthread_barrier_wait(current->barrier);

    // Переобозначить массивы для следующей итерации.
    float* swap = current->din1;
    current->din1 = current->din2;
```

```
        current->din2 = swap;
    }

    status = cudaFree(current->din1);
    status = cudaFree(current->din2);
    status = cudaFree(current->dout);
    free(hin);
    free(hout);
    pthread_exit(NULL);
    return NULL;
}
```

10.5. Boost.Thread

Boost – популярная открытая коллекция кроссплатформенных библиотек различного назначения для языка C++. В частности, Boost.Thread реализует интерфейс управления потоками, похожий на *pthread*, но за счет кроссплатформенности доступный как для Linux, так и для Windows.

Пусть необходимо организовать работу некоторого итерационного алгоритма на нескольких GPU и CPU с помощью Boost.Thread.

Для этого сформируем класс, который будет хранить данные и управлять исполнением одного потока:

```
class ThreadRunner
{
    static int status;
    static boost::mutex m;

    void SetLastError(int status);

    int idevice;
    int step, finish;
    int nx, ny;

    float *in_dev, *out_dev;

    CUcontext ctx;

    boost::barrier* b1, b2;
    boost::thread t;

    // Основная функция рабочего потока.
```

```
    void thread_func();

public:

    ThreadRunner(int idevice, int nx, int ny, boost::barrier* b);

    static int GetLastError();

    // Загрузка входных данных в память GPU.
    void Load(float* input);

    // Выгрузка результатов в память хоста.
    void Unload(float* output);

    void Pass();

    ~ThreadRunner();
};
```

В основной функции примера создается множество экземпляров управляющего класса, по одному для каждого GPU. Синхронизация рабочих и основного потока производится после каждой итерации с помощью барьера – *boost::barrier*. При инициализации барьера указывается целое число – количество вызовов метода *wait*, при котором барьер разблокируется. В остальных случаях барьер заблокирован.

```
int main(int argc, char* argv[])
{
    int ndevices = 0;
    cudaError_t cuda_status = cudaGetDeviceCount(&ndevices);
    printf("%d CUDA device(s) found\n", ndevices);
    if (!ndevices) return 0;

    // Сгенерировать входные данные.
    size_t np = nx * ny;
    float* data = new float[np * 2];
    float invdrandmax = 1.0 / RAND_MAX;
    for (size_t i = 0; i < np; i++)
        data[i] = rand() * invdrandmax;

    // Создать барьер, разблокирующийся при (ndevices + 1)
    // вызове метода wait().
    boost::barrier b(ndevices + 1);
```

```
// Создать экземпляры управляющего класса и загрузить
// в каждый одинаковые входные данные.
ThreadRunner** runners = new ThreadRunner*[ndevices + 1];
for (int i = 0; i < ndevices; i++)
{
    runners[i] = new ThreadRunner(i, nx, ny, &b);
    runners[i]->Load(data);
}

// Выполнить заданное число итераций.
float* input = data;
float* output = data + np;
for (int i = 0; i < nticks; i++)
{
    // Отправить сигналы для разблокировки барьера и
    // проведения одной итерации в каждом потоке.
    for (int i = 0; i < ndevices; i++)
        runners[i]->Pass();

    int status = ThreadRunner::GetLastError();
    if (status) return status;

    // Параллельно с GPU-потоками расчитать контрольный
    // результат с помощью эквивалентного CPU-ядра.
    pattern2d_cpu(1, nx, 1, 1, ny, 1, input, output, ndevices);
    float* swap = output;
    output = input;
    input = swap;

    // Отправить сигнал для разблокировки барьера и
    // перехода к очередной итерации.
    b.wait();
}

// Compare each GPU result to CPU result.
float* control = input;
float* result = output;
for (int idevice = 0; idevice < ndevices; idevice++)
{
    runners[idevice]->Unload(result);

    // Вычислить максимальное различие результатов.
    int maxi = 0, maxj = 0;
    float maxdiff = fabs(control[0] - result[0]);
    for (int j = 0; j < ny; j++)
    {
```

```
            for (int i = 0; i < nx; i++)
            {
                float diff = fabs(
                    control[i + j * nx] - result[i + j * nx]);
                if (diff > maxdiff)
                {
                    maxdiff = diff;
                    maxi = i; maxj = j;
                }
            }
        }
        printf("Device %d result abs max diff = %f @ (%d,%d)\n",
            idevice, maxdiff, maxi, maxj);

        delete runners[idevice];
    }

    delete[] data;
    delete[] runners;

    return 0;
}
```

При инициализации экземпляра класса вызовом *boost::bind* создается рабочий поток, в котором запускается функция *ThreadRunner::thread_func*, а параметром потока является сам экземпляр класса – *this*. Конструктор также привязывает поток к определенному устройству и создает в нем CUDA-контекст. Для разнообразия в данном примере управление контекстами производится с помощью вызовов CUDA Driver API: *cuDeviceGet* – получить идентификатор устройства, *cuCtxCreate* – создать контекст и сделать его текущим, *cuCtxPopCurrent* – выгрузить текущий контекст в заданную переменную, *cuCtxPushCurrent* – сделать текущим заданный контекст.

```
ThreadRunner::ThreadRunner(int idevice, int nx, int ny, boost::barrier* b) :
    b2(2), finish(0)
{
    t = boost::bind(&ThreadRunner::thread_func, this);

    this->idevice = idevice;
    this->step = 0;
    this->nx = nx; this->ny = ny;
    this->b1 = b;
```

```
    // Получить GPU с заданным индексом.
    CUdevice dev;
    CUresult cu_status = cuDeviceGet(&dev, idevice);

    // Создать для GPU CUDA-контекст и сделать его текущим.
    cu_status = cuCtxCreate(&ctx, 0, dev);

    // Создать входные и выходные массивы в памяти GPU.
    size_t size = nx * ny * sizeof(float);
    cudaError_t cuda_status = cudaMalloc((void**)&in_dev, size);
    cuda_status = cudaMalloc((void**)&out_dev, size);

    // Выгрузить текущий контекст устройства.
    cu_status = cuCtxPopCurrent(&ctx);

    printf("Device %d initialized\n", idevice);
}
```

Функция потока полностью состоит из одного цикла с выходом по условию останова. В момент запуска она сразу же блокируется на барьере *b2* – втором барьере, синхронизирующем попарно каждый рабочий поток с основным потоком. При вызове метода *Pass* этот барьер разблокируется, поток проходит одну итерацию и блокируется на другом общем барьере *b1*, задающим границу итерации.

```
// The function executed by each thread assigned with CUDA device.
void ThreadRunner::thread_func()
{
    // Thread iterations loop.
    while (1)
    {
        this->b2.wait();

        // Destructor signals "finish" to end thread loop.
        if (finish) break;

        int idevice = this->idevice;

        // Set focus on the specified CUDA context.
        // Previously we created one context for each thread.
        CUresult cu_status = cuCtxPushCurrent(ctx);
        if (cu_status != CUDA_SUCCESS)
        {
            fprintf(stderr, "Cannot push current context for device %d, ↩
                status = %d\n",
                idevice, cu_status);
```

```
            ThreadRunner::SetLastError((int)cu_status);
        }

        int status = pattern2d_gpu(1, nx, 1, 1, ny, 1,
            in_dev, out_dev, idevice);
        if (status)
        {
            fprintf(stderr, "Cannot execute pattern 2d on device %d, status =↩
                 %d\n",
                idevice, status);
            ThreadRunner::SetLastError(status);
        }
        step++;

        // Pop the previously pushed CUDA context out of this thread.
        cu_status = cuCtxPopCurrent(&ctx);
        if (cu_status != CUDA_SUCCESS)
        {
            fprintf(stderr, "Cannot pop current context for device %d, status↩
                 = %d\n",
                idevice, cu_status);
            ThreadRunner::SetLastError((int)cu_status);
        }

        // Swap device input and output buffers.
        float* swap = in_dev;
        in_dev = out_dev;
        out_dev = swap;

        printf("Device %d completed step %d\n", idevice, step);

        this->b1->wait();
    }
}
```

Таким образом, с помощью потоков и двух барьеров методами *Boost.Thread* организована параллельная работа CUDA-ядер на нескольких GPU и одновременный расчет эквивалентного CPU-ядра на хост-системе.

10.6. OpenMP

Одна из самых популярных технологий организации многопоточной обработки – это OpenMP. OpenMP расширяет целевой язык набором *директив* – аннотаций

к исходному коду, содержащих дополнительные параметры. В отличие от *pthread* и *Boost.Thread*, в OpenMP создание и завершение потоков происходит неявно, и распределение нагрузки по рабочим потокам может быть выполнено автоматически.

Директивы OpenMP встраиваются во многие языки программирования, например, в C или Fortran. Реализация OpenMP обычно состоит из двух частей: поддержки директив самим компилятором (который должен их упростить до вызовов внутренних функций OpenMP API) и библиотеки, реализующей OpenMP на низком уровне, например, *libgomp*.

Пусть необходимо организовать работу некоторого итерационного алгоритма на нескольких GPU и CPU с помощью OpenMP. Для этой цели в следующем примере посредством директив *#pragma omp section* определены отдельные параллельные секции для одного CPU-потока и всех GPU-потоков.

```
// Для каждого GPU создать отдельный поток
// и вызываеть из него thread_func.
#pragma omp sections
{
    // Секция для GPU-потоков.
    #pragma omp section
    {
        ...
    }

    // Секция для CPU-потоков.
    #pragma omp section
    {
        ...
    }
}
```

В GPU-секции проводится инициализация и запуск функции потока из предыдущего примера, использовавшего *pthread*.

```
// Секция для GPU-потоков.
#pragma omp section
{
    #pragma omp parallel for
    for (int i = 0; i < ndevices; i++)
    {
        config_t* config = configs + i;
```

```
        config->idevice = i;
        config->step = 0;
        config->nx = nx; config->ny = ny;
        config->inout_cpu = inout + np * i;
        config->status = thread_func(config);
    }
}
```

В CPU-секции запускается проверочная CPU-функция для контроля результата.

```
// Секция для CPU-потоков.
#pragma omp section
{
    // Запуск CPU-аналога вычислительной функции
    // для контроля правильности результатов.
    control = inout + ndevices * np;
    float* input = inout + (ndevices + 1) * np;
    for (int i = 0; i < nticks; i++)
    {
        pattern2d_cpu(1, configs->nx, 1, 1, configs->ny, 1,
            input, control, ndevices);
        float* swap = control;
        control = input;
        input = swap;
    }
    float* swap = control;
    control = input;
    input = swap;
}
```

Таким образом, CUDA может взаимодействовать с другими системами параллельных вычислений, и примеры их использования показанные в данной главе, могут быть использованы для решения реальных задач.

Глава 11

CUDA Streams

Максимальная эффективность вычислительной системы может быть достигнута только в том случае, когда все ее функциональные элементы непрерывно выполняют полезную работу. В частности, этот принцип лежит в основе методологии запуска тысячей легковесных нитей на GPU: если одна нить ожидает операцию с памятью, то планировщик стремится переключить исполнение в другую нить, в которой данные для вычислительной операции уже готовы. За счет этого подсистема памяти всегда будет наилучшим образом обеспечена работой во время вычислений, и наоборот. CUDA streams (потоки или очереди команд) имеют аналогичное предназначение, но в отличие от планировщика GPU, управляются на уровне пользовательской программы с помощью функций CUDA API и применимы только к операциям запуска ядер и обмена данными между хостом и GPU.

CUDA stream не имеет отношения к системным потокам исполнения и является абстрактной конструкцией. Его функция состоит в возможности создания *очередей из команд* запуска ядер и копирования данных между хостом и GPU. Каждый CUDA stream определяет некоторый идентификатор, который может быть передан в качестве аргумента при запуске определенных команд. В таком случае эти команды будут работать в одном CUDA stream, *последовательно* по отношению друг к другу и *независимо* (т.е. возможно, что и параллельно) по отношению к другим очередям команд.

Если программа запускает несколько ядер без явного указания CUDA stream, то по умолчанию все они будут работать в CUDA stream 0, т.е. всегда последовательно. Также всегда последовательно происходит выполнение операций копирования без суффикса Async или с суффиксом Async, но с одинаковым номером CUDA stream. Асинхронная передача данных может работать только с pinned-памятью, выделенной с помощью *cudaMallocHost* или *cudaHostRegister*.

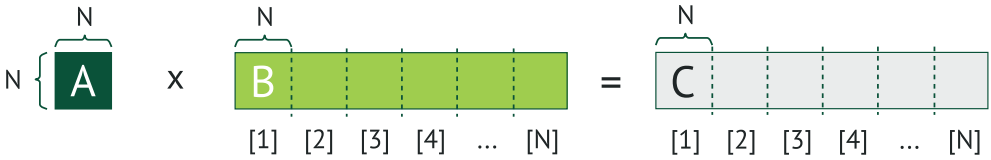


Рис. 11.1. Схема блочного перемножения матриц для применения CUDA streams

11.1. Пример: перемножение матриц

Рассмотрим следующий пример: матрица A умножается на матрицу B с помещением результата в матрицу C. Пусть размерности матрицы A малы, а число колонок матрицы B достаточно велико, для того, чтобы ее копирование занимало продолжительное время. Решим эту задачу двумя способами: копируя данные и умножая матрицы обычным образом или производя блочное копирование и умножение для отдельных подматриц B в разных очередях команд.

В случае использования CUDA stream для каждой подматрицы единым потоком исполнения необходимо связать всю последовательность команд: загрузка данных, запуск ядра и выгрузка результатов. Различные очереди команд могут работать независимо друг от друга: пока один CUDA stream выполняет загрузку данных, другой может уже иметь свою часть данных загруженной и заниматься вычислениями. В идеальном случае за счет подобного наложения во времени вычислений и копирования данных может быть повышена эффективность программы.

Сравним обычное перемножение матриц и блочный алгоритм с применением CUDA streams. В первом случае измеряется время работы последовательных шагов «загрузка – расчет – выгрузка»:

```
cudaEvent_t start;
cudaEventCreate(&start);
cudaEventRecord(start, 0);

// Скопировать матрицы из памяти CPU в память GPU
status = cublasSetVector(n2, sizeof(real), h_A, 1, d_A, 1);
assert(status == CUBLAS_STATUS_SUCCESS);
status = cublasSetVector(n2, sizeof(real), h_B, 1, d_B, 1);
assert(status == CUBLAS_STATUS_SUCCESS);
status = cublasSetVector(n2, sizeof(real), h_C, 1, d_C, 1);
assert(status == CUBLAS_STATUS_SUCCESS);
```

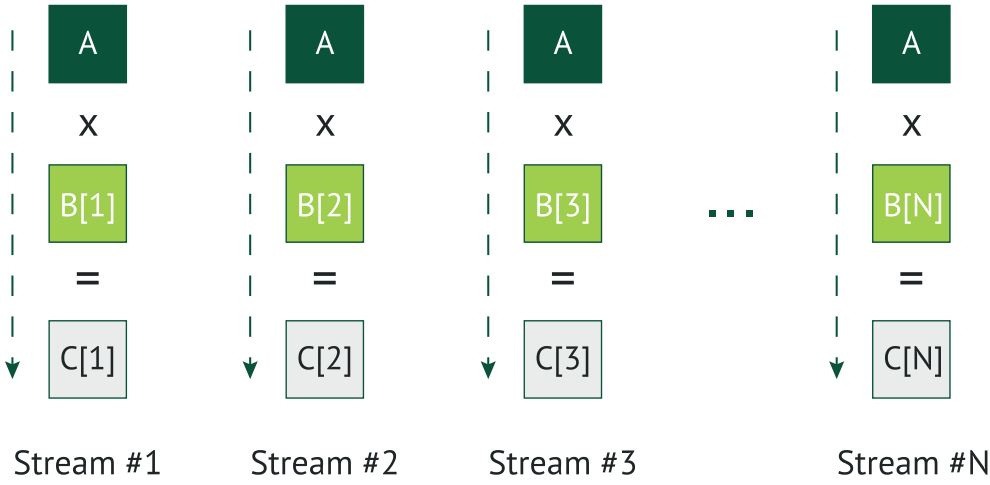


Рис. 11.2. Объединение операций над связанными блоками в CUDA streams

```
// Умножить матрицы с использованием CUBLAS
cublas_gemm(transa, transb, n, n, n,
        alpha, d_A, n, d_B, n, beta, d_C, n);
status = cublasGetError();
assert(status == CUBLAS_STATUS_SUCCESS);

// Скопировать обратно результат
status = cublasGetVector(n2, sizeof(real), d_C, 1, h_C, 1);
assert(status == CUBLAS_STATUS_SUCCESS);
cudaEvent_t stop;
cudaEventCreate(&stop);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
```

Во втором случае выбирается размер подматрицы, и вычисления предваряют создание CUDA streams по числу подматриц (на месте многоточий следует контроль ошибок):

```
for (int i = 0; i < nstreams; i++)
{
    cudaerr = cudaStreamCreate(&stream[i]);
    ...
}
```

Расчетная часть сохраняет структуру “загрузка – расчет – выгрузка”, появляются лишь циклы по обработке подматриц в соответствующих CUDA streams.

Кроме того, все операции заменены на асинхронные версии:

```
cudaEventRecord(event_start[0], 0);

for (int istream = 0; istream < nstreams; istream++)
{
    // Вычисление отступа с добавлением остатка.
    int szpart = n / nstreams;
    size_t shift = n * szpart * istream;
    if (istream == nstreams - 1)
        szpart += n % nstreams;

    // Загрузка подматриц в память GPU по отступам
    // в соответствующих CUDA streams.
    status = cublasSetVectorAsync(n * szpart, sizeof(real),
        h_B + shift, 1, d_B[istream], 1, stream[istream]);
    ...
    status = cublasSetVectorAsync(n * szpart, sizeof(real),
        h_C + shift, 1, d_C[istream], 1, stream[istream]);
    ...
}

for (int istream = 0; istream < nstreams; istream++)
{
    int szpart = n / nstreams;
    if (istream == nstreams - 1)
        szpart += n % nstreams;

    // Установка текущего CUDA stream для последующих операций CUBLAS.
    status = cublasSetKernelStream(stream[istream]);
    ...

    // Перемножение подматриц в соответствующем CUDA stream.
    cublas_gemm(transa, transb, n, szpart, n,
        alpha, d_A, n, d_B[istream], n, beta, d_C[istream], n);
    status = cublasGetError();
    ...
}

for (int istream = 0; istream < nstreams; istream++)
{
    int szpart = n / nstreams;
    size_t shift = n * szpart * istream;
    if (istream == nstreams - 1)
        szpart += n % nstreams;
```

```
    // Выгрузка подматриц из памятьи GPU по отступам
    // в соответствующих CUDA streams.
    status = cublasGetVectorAsync(n * szpart, sizeof(real),
        d_C[istream], 1, h_C + shift, 1, stream[istream]);
    ...
}

cudaEventRecord(event_end[0], 0);
cudaEventSynchronize(event_end[0]);
```

В конце следует удаление CUDA streams:

```
for (int istream = 0; istream < nstreams; istream++)
{
    cudaerr = cudaStreamDestroy(stream[istream]);
    ...
}
```

Ниже приведены результаты тестирования последовательной версии (верхняя строчка) и версии с использованием CUDA streams (нижняя строчка) перемножения матриц с двойной точностью на GPU Tesla C1060:

```
$ ./gemm_streamed 4 1024 1025 1 N N 1.0 0.0 16
n       time            gflops      test    enorm       rnorm
1024    0.013909 sec    154.390014  PASSED  0.018676    262177.187500
1024    0.011231 sec    191.204784  PASSED  0.018693    262323.812500

$ ./gemm_streamed 4 4096 4097 1 N N 1.0 0.0 16
n       time            gflops      test    enorm       rnorm
4096    0.431783 sec    318.305308  PASSED  0.293618    4194287.250000
4096    0.396524 sec    346.609298  PASSED  0.293707    4194416.500000
```

Показано, что CUDA streams позволяют в той или иной степени увеличить эффективность программ. Однако прежде чем использовать CUDA streams, имеет смысл сделать простую оценку: насколько соизмеримы времена работы асинхронных процессов (в данном случае – копирования и вычислений)? Если ядро достаточно вычислительно-емкое, чтобы работать значительно дольше, чем пересылка соответствующих данных, то эффект от применения CUDA streams скорее всего будет минимальным. Напротив, если времена вычислений и пересылки данных примерно одинаковы, то в идеальном случае можно получить двукратный прирост производительности.

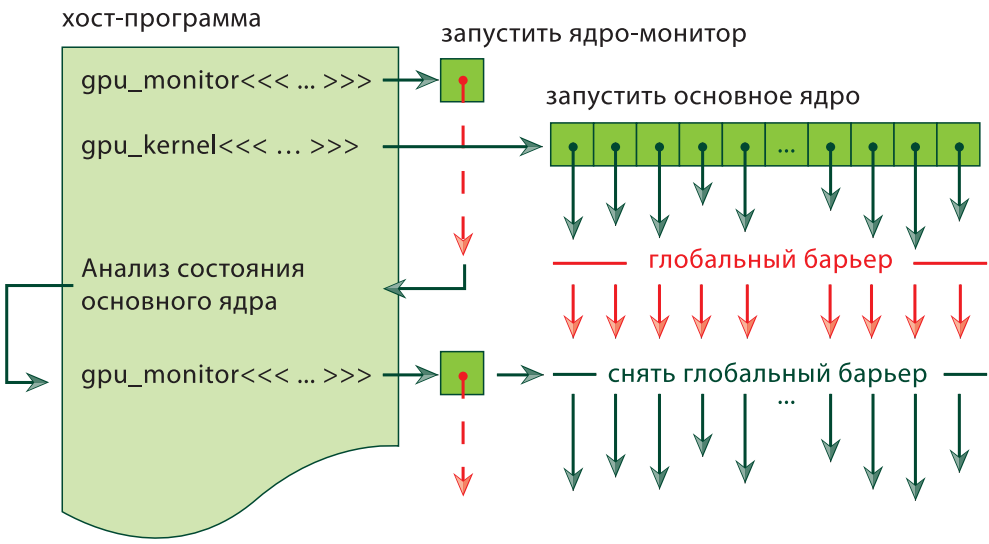


Рис. 11.3. Схема системы активной синхронизации

Эффективность CUDA streams может быть ниже на версиях ОС Windows с поддержкой WDDM (Windows Display Driver Model).

11.2. Пример: взаимодействие между CUDA-ядром и хостом

Обычно запущенное CUDA-ядро пассивно работает вплоть до своего завершения, но возникают ситуации, когда необходимо передать некоторое сообщение хосту и отреагировать на ответное сообщение, *не завершая работу ядра*. Очевидно, обмен сообщениями может быть организован посредством глобальной памяти GPU или pinned-памяти хоста. Однако для гарантированной передачи корректного значения при этом необходимо еще обеспечить *атомарный* режим операций чтения и записи, доступность которого является определяющим фактором при выборе метода синхронизации. Следующий пример показывает, каким образом может быть организовано подобное взаимодействие с помощью дополнительного ядра и двух CUDA streams.

Выполнение алгоритма как обычно начинается с инициализации данных в хост-программе. Помимо данных приложения для CUDA-ядра, в памяти GPU создаются переменные *lock* и *finish*. Переменная *lock* хранит текущее состояние исполнения

ядра: 0 – «работает», 1 – «ожидает сигнала на продолжение работы». Переменная *finish* служит маркером окончания работы ядра. Прежде чем запускать основное ядро приложения, на том же GPU начинает работу простое однопоточное ядро – монитор состояния:

```
// Ядро-монитор - для мониторинга исполнения целевого ядра.
__global__ void gpu_monitor(int* lock)
{
    // Разблокировать gpu_kernel, связанное с lock,
    // которое ожидает пока lock не станет 0.
    atomicCAS(lock, 1, 0);

    // Ожидать, пока gpu_kernel заблокирует себя.
    // Как только это произошло, lock становится 1,
    // и по этому условию данное ядро-монитор завешается,
    // сигнализируя хосту о наступлении события в
    // ядре gpu_kernel.
    while (!atomicCAS(lock, 1, 1)) continue;
}
```

Ядро-монитор переводит *lock* в состояние «разблокировано» и с помощью атомарной операции ожидает новой блокировки со стороны основного ядра. Вслед за монитором запускается основное ядро и начинает вычисления. Активный режим взаимодействия предполагает, что ядро выбирает момент синхронизации произвольным образом. В данном случае отправка параметров состояния производится на каждом шаге некоторого итерационного процесса и после завершения работы:

```
// "Целевое" ядро - ядро, выполняющее полезную работу.
__global__ void gpu_kernel(float* data, size_t size, int npasses,
    int* lock, int* finish, int* pmaxidx, float* pmaxval)
{
    *finish = 0;

    for (int ipass = 0; ipass < npasses; ipass++)
    {
        // Выполнить шаг вычислений.
        for (int i = size - 1; i >= 0; i--)
            data[i] = data[i - 1];
        data[0] = data[size - 1];

        // Приготовить некоторые данные для отправки на хост.
        int maxidx = 0;
        float maxval = data[0];
        for (int i = 1; i < size; i++)
```

```
            if (data[i] >= maxval)
            {
                maxval = data[i];
                maxidx = i;
            }
        *pmaxidx = maxidx;
        *pmaxval = maxval;

        // Поток продолжает работать, если lock = 0 и блокируется
        // в случае lock = 1.

        // Заблокировать поток: установить lock = 1, если он был 0.
        atomicCAS(lock, 0, 1);

        // Ожидать снятия блокировки: пока lock не станет 0.
        while (atomicCAS(lock, 0, 0)) continue;
    }

    // Заблокировать поток.
    atomicCAS(lock, 0, 1);

    *finish = 1;
}
```

После выполнения необходимых вычислений основное ядро готовит параметры состояния (например, нормы ошибки или скорость сходимости) и присваивает *lock* значение 1. Перемена значения *lock* служит сигналом разблокирования ядра-монитора, которое просто завершает свою работу. В свою очередь, установленное на стороне хоста ожидание завершения ядра-монитора сигнализирует о возможности начать читать данные GPU с хоста, гарантированно после окончания их записи со стороны GPU, т.е. атомарно. Обработав данные, хост-программа снова запускает ядро-монитор, которое сразу же снимает блокировку основного ядра, и далее работа продолжается циклически по вышеописанному алгоритму.

```
struct params_t
{
    cudaStream_t stream;
    float* data;
    int* maxidx;
    float* maxval;
    int* lock;
    int* finish;
}
```

```
cpu, gpu;

// Хост-программа
int main(int argc, char* argv[])
{
    if (argc != 3)
    {
        printf("%s <size> <npasses>\n", argv[0]);
        return 0;
    }

    int count = 0;
    cudaError_t custat = cudaGetDeviceCount(&count);
    if (custat != cudaSuccess)
    {
        fprintf(stderr, "Cannot get CUDA device count: %s\n",
            cudaGetErrorString(custat));
        return 1;
    }
    if (!count)
    {
        fprintf(stderr, "No CUDA devices found\n");
        return 1;
    }

    size_t size = atoi(argv[1]);
    int npasses = atoi(argv[2]);

    cpu.data = (float*)malloc(sizeof(float) * size);
    double dinvrandmax = (double)1.0 / RAND_MAX;
    for (int i = 0; i < size; i++)
        cpu.data[i] = rand() * dinvrandmax;

    gpu.data = NULL;
    custat = cudaMalloc((void**)&gpu.data, sizeof(float) * size);
    if (custat != cudaSuccess)
    {
        fprintf(stderr, "Cannot create GPU data buffer: %s\n",
            cudaGetErrorString(custat));
        return 1;
    }
    custat = cudaMemcpy(gpu.data, cpu.data, sizeof(float) * size,
        cudaMemcpyHostToDevice);
    if (custat != cudaSuccess)
    {
        fprintf(stderr, "Cannot fill GPU data buffer: %s\n",
```

```
            cudaGetErrorString(custat));
        return 1;
    }
    free(cpu.data);

    custat = cudaMalloc((void**)&gpu.maxidx, sizeof(int));
    if (custat != cudaSuccess)
    {
        fprintf(stderr, "Cannot create GPU maxidx buffer: %s\n",
            cudaGetErrorString(custat));
        return 1;
    }

    custat = cudaMalloc((void**)&gpu.maxval, sizeof(float));
    if (custat != cudaSuccess)
    {
        fprintf(stderr, "Cannot create GPU maxval buffer: %s\n",
            cudaGetErrorString(custat));
        return 1;
    }

    custat = cudaMallocHost((void**)&gpu.finish, sizeof(int));
    if (custat != cudaSuccess)
    {
        fprintf(stderr, "Cannot create GPU finish buffer: %s\n",
            cudaGetErrorString(custat));
        return 1;
    }

    // Инициализировать lock - переменную состояния блокировки.
    // Начальное состояние - "заблокирован". Ядро-монитор
    // gpu_monitor должно быть запущен *раньше*, чем gpu_kernel
    // и разблокировать lock.
    custat = cudaMalloc((void**)&gpu.lock, sizeof(int));
    if (custat != cudaSuccess)
    {
        fprintf(stderr, "Cannot create GPU lock buffer: %s\n",
            cudaGetErrorString(custat));
        return 1;
    }
    int one = 1;
    custat = cudaMemcpy(gpu.lock, &one, sizeof(int),
        cudaMemcpyHostToDevice);
    if (custat != cudaSuccess)
    {
        fprintf(stderr, "Cannot initialize GPU lock buffer: %s\n",
```

```
        cudaGetErrorString(custat));
    return 1;
}

// Создать CUDA streams, в которых будут одновременно исполняться
// целевое ядро и ядро-монитор.
custat = cudaStreamCreate(&gpu.stream);
if (custat != cudaSuccess)
{
    fprintf(stderr, "Cannot create monitoring stream: %s\n",
        cudaGetErrorString(custat));
    return 1;
}
custat = cudaStreamCreate(&cpu.stream);
if (custat != cudaSuccess)
{
    fprintf(stderr, "Cannot create monitoring stream: %s\n",
        cudaGetErrorString(custat));
    return 1;
}

// Запустить ядро-монитор.
gpu_monitor<<<1, 1, 1, gpu.stream>>>(gpu.lock);
custat = cudaGetLastError();
if (custat != cudaSuccess)
{
    fprintf(stderr, "Cannot launch monitoring GPU kernel: %s\n",
        cudaGetErrorString(custat));
    return 1;
}

// Запустить целевое ядро.
gpu_kernel<<<1, 1, 1, cpu.stream>>>(
    gpu.data, size, npasses, gpu.lock,
    gpu.finish, gpu.maxidx, gpu.maxval);
custat = cudaGetLastError();
if (custat != cudaSuccess)
{
    fprintf(stderr, "Cannot launch target GPU kernel: %s\n",
        cudaGetErrorString(custat));
    return 1;
}

while (1)
{
    // Ожидать завершения ядра-монитора.
```

```
custat = cudaStreamSynchronize(gpu.stream);
if (custat != cudaSuccess)
{
    fprintf(stderr, "Cannot synchronize GPU monitor kernel: %s\n",
        cudaGetErrorString(custat));
    return 1;
}

// Получить из памяти GPU параметры, характеризующие
// текущее состояние целевого ядра.
int maxidx = 0;
custat = cudaMemcpyAsync(&maxidx, gpu.maxidx, sizeof(int),
    cudaMemcpyDeviceToHost, gpu.stream);
if (custat != cudaSuccess)
{
    fprintf(stderr, "Cannot get GPU maxidx value: %s\n",
        cudaGetErrorString(custat));
    return 1;
}
float maxval = 0.0;
custat = cudaMemcpyAsync(&maxval, gpu.maxval, sizeof(float),
    cudaMemcpyDeviceToHost, gpu.stream);
if (custat != cudaSuccess)
{
    fprintf(stderr, "Cannot get GPU maxval value: %s\n",
        cudaGetErrorString(custat));
    return 1;
}
custat = cudaStreamSynchronize(gpu.stream);
if (custat != cudaSuccess)
{
    fprintf(stderr, "Cannot synchronize GPU monitor kernel: %s\n",
        cudaGetErrorString(custat));
    return 1;
}
printf("max value = %f @ index = %d\n", maxval, maxidx);

// Проверить, завершило ли целевое ядро работу.
if (*gpu.finish == 1) break;

// Снова запустить ядро-монитор.
gpu_monitor<<<1, 1, 1, gpu.stream>>>(gpu.lock);
custat = cudaGetLastError();
if (custat != cudaSuccess)
{
    fprintf(stderr, "Cannot launch monitoring GPU kernel: %s\n",
```

```
                cudaGetErrorString(custat));
            return 1;
        }
    }

    // Ожидать завершения целевого ядра.
    custat = cudaStreamSynchronize(cpu.stream);
    if (custat != cudaSuccess)
    {
        fprintf(stderr, "Cannot synchronize GPU monitor kernel: %s\n",
            cudaGetErrorString(custat));
        return 1;
    }

    custat = cudaFree(gpu.data);
    custat = cudaFree(gpu.maxidx);
    custat = cudaFree(gpu.maxval);
    custat = cudaFreeHost(gpu.finish);
    custat = cudaFree(gpu.lock);
    custat = cudaStreamDestroy(gpu.stream);
    custat = cudaStreamDestroy(cpu.stream);
    return 0;
}
```

Для обеспечения одновременной работы основного ядра и ядра-монитора использованы CUDA streams. Также требуется поддержка целевым устройством конкурентного исполнения ядер (доступна, начиная с архитектуры Fermi).

В данном примере основное CUDA-ядро для простоты исполняет лишь одну нить, хотя на практике их может быть очень много. Однако здесь CUDA-ядро фактически реализует *глобальный барьер*, что в сочетании с моделью SIMT ограничивает их возможное число максимальным количеством нитей, которые данное устройство может исполнять *физически одновременно* (т.н. *режим persistent threads*).

11.3. Пример: использование нескольких устройств и асинхронное копирование

Для распределенного вычисления значений в точках сетки по локальному шаблону (например, явные разностные схемы решения ОДУ) часто применяют *декомпозицию* расчетной области задачи на отдельные части (рис. 11.4). Поскольку каж-

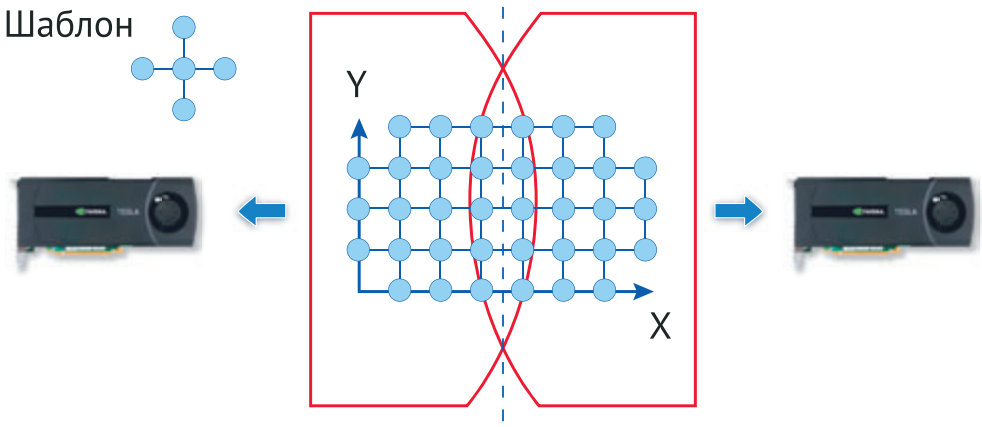


Рис. 11.4. Декомпозиция расчетной сетки для узлов с распределенной памятью

дое новое значение в узле зависит от некоторой локальной окрестности, при простом разделении сетки узлы вблизи границ внутренних частей не имеют необходимых соседних значений. Устранить этот эффект можно, разделив область с наложением внутренних границ, т.е. добавив соответствующие полосы узлов соседних частей.

Данных для новых «фиктивных» граничных узлов (в английской литературе halo – «границы без внутренности») по-прежнему будет не хватать, но теперь они только дублируются для обеспечения вычислений в соседних узлах. Распределенные вычисления с подобным разделением сетки требуют синхронизации «фиктивных» границ перед каждым новым пересчетом. Синхронизацию такого рода между несколькими GPU можно еализовать с помощью функции прямого асинхронного копирования *cudaMemcpyPeerAsync* (или *cudaMemcpyAsync* при использовании UVA), как это показано для случая двух частей в следующем примере:

```
// Массивы указателей с граничными полосами на каждом устройстве:
float **dev_halo_right; // правая синхронизируемая область
float **dev_halo_left; // левая синхронизируемая область
float **dev_data_right; // правая граничная область
float **dev_data_left; // левая граничная область
float **dev_data; // область, которую не требуется синхронизировать

cudaStream_t *devStream; // одна очередь команд на каждое устройство
```

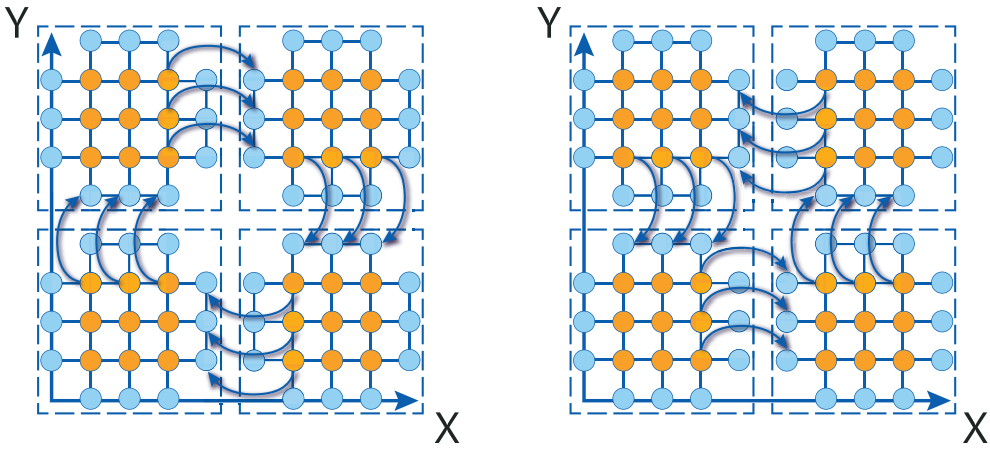


Рис. 11.5. Обмен внутренними границами в распределенной вычислительной сети

```
// Выделение и инициализация массивов и потоков на каждом устройстве.
...

int haloSize = 1024;

// Синхронизация правой области с левой на устройстве i+1.
for( int i = 0; i < numDev - 1; i++)
    cudaMemcpyPeerAsync(dev_halo_left[i+1], i+1, dev_data_right[i], i,
        sizeof(float) * haloSize, devStream[i]);

// Синхронизация левой области с правой на устройстве i-1.
for( int i = 1; i < numDev; i++)
    cudaMemcpyPeerAsync(dev_halo_right[i-1], i-1, dev_data_left[i], i,
        sizeof(float) * haloSize, devStream[i]);

// Ожидание окончания копирования:
for( int i = 0; i < numDev; i++)
{
    cudaSetDevice(i);
    cudaStreamSynchronize(devStream[i]);
}

// Очередной полный пересчет узлов в отдельных частях сетки.
for( int i = 0; i < numDev; i++)
{
    // Переключение индекса текущего GPU.
    cudaSetDevice(i);
```

```
    kernel<<<grid, block, 0, devStream[i]>>>(dev_data[i], dev_data_left[i],
                dev_data_right[i], dev_halo_left[i], dev_halo_right[i]);
}

...
```

В качестве одного из своих аргументов *cudaMemcpyPeerAsync* использует имя потока источника и не требует предварительного вызова *cudaSetDevice*. Для включения поддержки прямого копирования необходимо предварительно вызвать функцию *cudaDeviceEnablePeerAccess* (см. секцию 3.4). В отличие от *cudaMemcpyPeer*, вызов *cudaMemcpyPeerAsync* является асинхронным по отношению к текущему устройству и позволяет реализовать одновременное копирование на нескольких устройствах сразу. В таблице 11.1 приведен сравнительный анализ времени, затраченного на синхронизацию при решении уравнений Навье – Стокса (12.1.2) на 8 устройствах Tesla M2050, расположенных на одном хосте.

Таблица 11.1. Влияние асинхронного копирования на среднее время синхронизации границ между 8 устройствами Tesla M2050

Метод	**среднее время синхронизации, мс**	**ускорение**
cudaMemcpyPeer	8,25	
cudaMemcpyPeerAsync	1,77	4,7x

Глава 12

Решение уравнений Навье – Стокса на GPU

Течения многих жидкостей и газов в естественной среде (движение воздуха в земной атмосфере, воды в реках и морях, газа в атмосферах Солнца и звезд и в межзвездных туманностях и т.п.) и инженерных сооружениях (в трубах, каналах, струях, пограничных слоях около движущихся в жидкости или газе твердых тел, следах за такими телами и т.п.) при определенных условиях являются *тубрулентными*. Для прямого численного моделирования турбулентных режимов течений используется система уравнений Навье – Стокса соответствующего вида. Полная форма уравнений Навье – Стокса в трехмерной декартовой системе координат для идеального газа с постоянной плотностью имеет вид:

$$\nabla \cdot \vec{\mathbf{u}} = 0, \tag{12.1}$$

$$\rho \left[\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} \right] = -\nabla T + \frac{1}{Re} \nabla^2 \mathbf{u}, \tag{12.2}$$

$$\rho \left[\frac{\partial T}{\partial t} + \mathbf{u} \cdot \nabla T \right] = \frac{1}{Re \cdot Pr} \Delta T + \frac{\gamma - 1}{\gamma \cdot Re} \Phi, \tag{12.3}$$

$$p = \rho R T, \tag{12.4}$$

где $\mathbf{u} = (u, v, w)$ – компоненты вектора скорости, p – давление, ρ – плотность. Здесь уравнение неразрывности 12.1 выражает постоянство объема, 12.2 – закон сохранения энергии, 12.3 – уравнение теплопроводности, и замыкает систему уравнение состояния 12.4. Число Рейнольдса $Re = \frac{UL}{\nu}$ характеризует свойства течения: $\nu = \frac{\mu}{\rho}$ – кинематическая вязкость, μ – вязкость, а L и U – средняя скорость и размер области в конкретной задаче. В уравнениях также присутствуют число Прандтля Pr, газовая постоянная R и удельная теплоемкость γ. Φ – это диссипативная функция, характеризующая работу вязких сил:

$$\Phi = \Phi_x + \Phi_y + \Phi_z$$

$$\Phi_x = 2(\frac{\partial u}{\partial x})^2 + (\frac{\partial v}{\partial x})^2 + (\frac{\partial w}{\partial x})^2 + \frac{\partial u}{\partial y}\frac{\partial v}{\partial x} + \frac{\partial u}{\partial z}\frac{\partial w}{\partial x},$$
$$\Phi_y = (\frac{\partial u}{\partial y})^2 + 2(\frac{\partial v}{\partial y})^2 + (\frac{\partial w}{\partial y})^2 + \frac{\partial v}{\partial x}\frac{\partial u}{\partial y} + \frac{\partial v}{\partial z}\frac{\partial w}{\partial y}, \qquad (12.5)$$
$$\Phi_z = (\frac{\partial u}{\partial z})^2 + (\frac{\partial v}{\partial z})^2 + 2(\frac{\partial w}{\partial z})^2 + \frac{\partial w}{\partial x}\frac{\partial u}{\partial z} + \frac{\partial w}{\partial y}\frac{\partial v}{\partial z}.$$

Подробный вывод уравнений, их описание и применение можно найти, например, в [18].

Если численное решение системы (12.1)–(12.4) требует явного описания всего диапазона пространственных и временных масштабов в случае турбулентных течений, то необходимое число узлов сетки пропорционально $Re^{9/4}$. Таким образом, практическое применение численного моделирования течений на основе системы уравнений Навье – Стокса сильно ограничено узким классом задач. Существуют модели LES (large eddy simulation), основанные на воспроизведении наиболее значимых вихрей и параметризации оставшейся части спектра [19], что позволяет снизить требования к разрешению сетки и вычислительным ресурсам. Тем не менее, такой подход сохраняет основные вычислительные особенности решения, которые и рассматриваются далее.

12.1. Метод покоординатного расщепления и соответствующий разностный метод первого порядка

Метод покоординатного расщепления применяется для решения параболических или эллиптических уравнений в частных производных. Идея метода заключается в расщеплении уравнений на несколько более простых – по уравнению вдоль каждой из осей координат. Эта операция проводится таким образом, что производные вдоль соответствующего направления берутся неявно, а остальные координаты считаются постоянными.

Например, уравнение для x-компоненты 12.2

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z} = -\frac{\partial T}{\partial x} + \frac{1}{Re}(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}) \qquad (12.6)$$

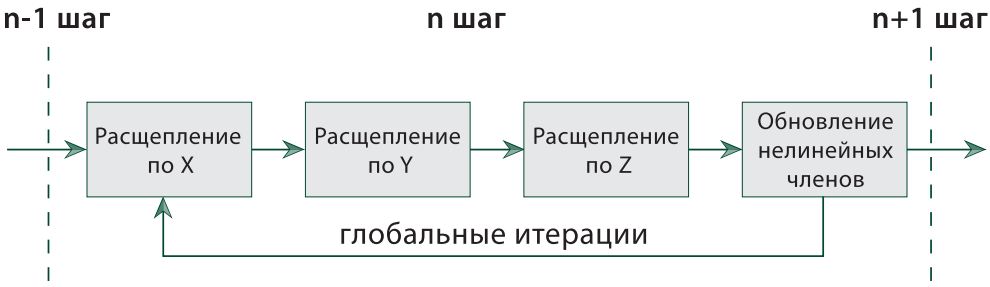


Рис. 12.1. Целый шаг метода покоординатного расщепления

расщепляется на 3 уравнения (12.7)–(12.9). Применение к ним конечно-разностной схемы первого порядка приводит к набору независимых трехдиагональных систем.

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = -\frac{\partial T}{\partial x} + \frac{1}{Re}\frac{\partial^2 u}{\partial x^2}, \tag{12.7}$$

$$\frac{\partial u}{\partial t} + v\frac{\partial u}{\partial y} = \frac{1}{Re}\frac{\partial^2 u}{\partial y^2}, \tag{12.8}$$

$$\frac{\partial u}{\partial t} + w\frac{\partial u}{\partial z} = \frac{1}{Re}\frac{\partial^2 u}{\partial z^2}. \tag{12.9}$$

Остальные уравнения могут быть преобразованы аналогичным образом. В общем случае метод расщепления может быть применен к задаче любой размерности.

Поскольку уравнения Навье – Стокса являются нелинейными, необходимо также проводить итерации для обновления нелинейных коэффициентов. В данной реализации проводятся итерации двух типов: глобальные и локальные. Глобальные итерации применяются для целого временного шага для всей системы, локальные – для каждого дробного шага, соответствующего одному из направлений. Общая схема алгоритма приведена на рис. 12.1. На каждом целом временном шаге система расщепляется по трем направлениям, и соответствующие системы уравнений последовательно решаются в глобальных итерациях. Число глобальных итераций выбирается так, чтобы численная ошибка, рассчитываемая как невязка в уравнении неразрывности, после каждого шага не превышала установленную величину.

На каждом дробном шаге расщепления соответствующие уравнения решаются при использовании конечно-разностной схемы. На рис. 12.2 показана модель дан-

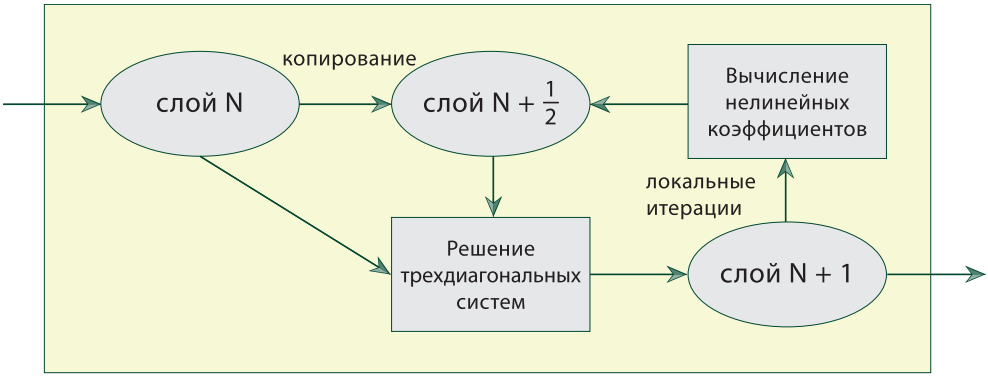


Рис. 12.2. Дробный шаг расщепления

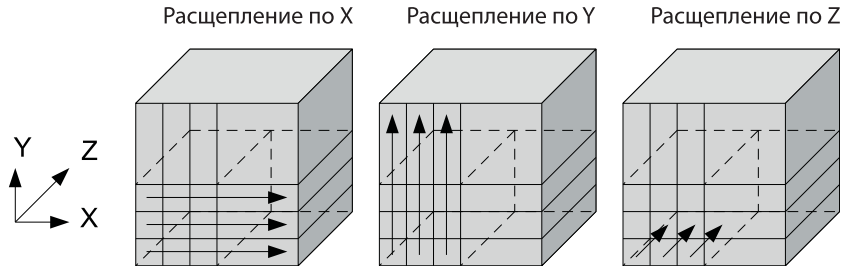


Рис. 12.3. Прогонки вдоль каждого направления расщепления

ных и расчетов для отдельного шага расщепления. Основное вычислительное ядро – это решатель наборов трехдиагональных систем. Полученные решения систем используются для обновления нелинейных коэффициентов. После этого проводится несколько локальных итераций для уменьшения численной ошибки.

Подобная схема была использована для решения двумерных уравнений Навье – Стокса в работе [20]. В дальнейшем на основе этой идеи был реализован численный метод для трехмерного случая и динамическая система визуализации [21].

12.1.1. Реализация метода прогонок на одном GPU

В CUDA-реализации прогонки каждая нить решает ровно одну трехдиагональную систему. За счет массивной параллельности нитей на GPU и большого числа систем удается эффективно загрузить устройство. На рис. 12.3 показано распреде-

ление трехмерного массива на дробном шаге расщепления по отдельным нитям. Каждая нить обрабатывает отдельный столбец массива в определенном направлении. Код соответствующего ядра аналогичен CPU-версии:

```
__device__ void solve_tridiagonal(
    FTYPE *a, FTYPE *b, FTYPE *c, FTYPE *d, FTYPE *x,
    int num, int id, int num_seg, int max_n )
{
    get(c,num-1) = 0.0;
    get(c,0) = get(c,0) / get(b,0);
    get(d,0) = get(d,0) / get(b,0);

    // прямой ход прогонки
    for (int i = 1; i < num; i++)
    {
        get(c,i) = get(c,i) / (get(b,i) - get(a,i) * get(c,i-1));
        get(d,i) = (get(d,i) - get(d,i-1) * get(a,i)) / (get(b,i) -
            get(a,i) * get(c,i-1));
    }
    get(x,num-1) = get(d,num-1);

    // обратный ход прогонки
    for (int i = num-2; i >= 0; i--)
        get(x,i) = get(d,i) - get(c,i) * get(x,i+1);
}
```

Часто для достижения максимальной эффективности на GPU приходится менять структуру данных, а иногда и сам метод.

Анализ показал, что подобные прогонки вдоль оси Z работают медленнее, чем по X и Y. Это объясняется тем, что прогонки вдоль Z приводят к чтению **каждой нитью** последовательных значений в памяти. В результате запросы нитей варпа к памяти не объединяются (uncoalesced, см. секцию 3.2.4). Напротив, в прогонках вдоль X и Y, **соседние нити** читают последовательные элементы в памяти, и все запросы объединяются. Оптимизировать прогонки по Z можно одним из двух способов: либо менять метод, либо менять шаблон доступа к данным. В данном случае реализован второй способ: входной массив данных транспонируется, а затем вместо Z прогонки запускается Y-прогонка. Эффективное транспонирование возможно за счет использования разделяемой памяти. В таблице 12.1 показано, насколько быстрее работает улучшенный вариант с переупорядочиванием данных в одинарной и двойной точности.

Таблица 12.1. Влияние транспонирования на производительность (среднее время работы ядра в миллисекундах), NVIDIA Tesla C2050

Точность	**Одинарная**			**Двойная**		
ядро	оригинал	с трансп.	ускор-е	оригинал	с трансп.	ускор-е
X dir	523	528	**1,0x**	717	709	**1,0x**
Y dir	525	531	**1,0x**	694	686	**1,0x**
Z dir	1681	533	**2,4x**	1901	693	**2,7x**
трансп.		164			190	
всего	2729	1756	**1,6x**	3312	2278	**1,5x**

Реализация метода расщепления тестировалась на двух задачах. В первой задаче моделировалось течение внутри прямоугольного канала со стенками параллельными осям координат (box_pipe). Такой вид границ позволяет равномерно загрузить вычислительное устройство. Во второй задаче рассматривалась геометрия акватории Белого моря (white_sea). В этом случае можно оценить, насколько хорошо метод работает на реальных задачах со сложными границами и неравномерной загрузкой устройства. Тестирование производительности проводилось на одном GPU (Tesla C1060 или Tesla C2050), а также на одном многоядерном CPU (Intel Core i7, 4 ядра, 2.8GHz). На CPU метод был распараллен с помощью директив OpenMP и использовал все доступные ядра. Ниже приведены таблицы результатов по каждому направлению расщепления и для всего метода в целом.

12.1.2. Реализации метода прогонок на нескольких GPU

Схема работы метода прогонок на нескольких GPU, подключенных к одному хосту, может состоять из следующих трех стадий:

- разделение данных между GPU;
- распараллеливание кода;
- синхронизация результатов между GPU.

Таблица 12.2. Результаты тестов производительности на модели box_pipe (систем в секунду)

Направление	CPU (4 ядра)	Tesla C1060	Tesla C2050	Tesla C2050 vs CPU
X	1,55	5,61	17,73	**11,4x**
Y	2,17	5,36	18,11	**8,3x**
Z	2,34	5,64	18,12	**7,8x**
все	1,96	5,30	16,09	**8,2x**

Таблица 12.3. Результаты тестов производительности на модели white_sea (систем в секунду)

Направление	CPU (4 ядра)	Tesla C1060	Tesla C2050	Tesla C2050 vs CPU
X	3,65	15,40	37,95	**10,4x**
Y	5,12	16,89	44,91	**8,8x**
Z	2,25	5,65	13,38	**5,9x**
все	3,82	11,88	27,40	**7,2x**

Скорость обмена данными между GPU существенно ниже скорости глобальной памяти, поэтому необходимо минимизировать количество пересылаемой информации. Разделение данных между GPU лучше всего провести вдоль направления X, поскольку все трехмерные массивы хранятся по строкам в виде одномерного вектора $(i, j, k) \rightarrow [dimY \cdot dimZ \cdot i + dimZ \cdot j + k]$, и сетка естественным образом разбивается на блоки размера, кратного $dimY \cdot dimZ$. В этом случае распараллеливание прогонок вдоль Y и Z тривиально и сводится к одновременному запуску ядер на нескольких GPU, работающих над соответствующими частями сетки:

```
for (int i = 0; i < nGPU; i++)
{
    cudaSetDevice(i); // Переключение устройства
    kernel<<<…>>>(devArray[i], ...); // Расчет своей области сетки
}
```

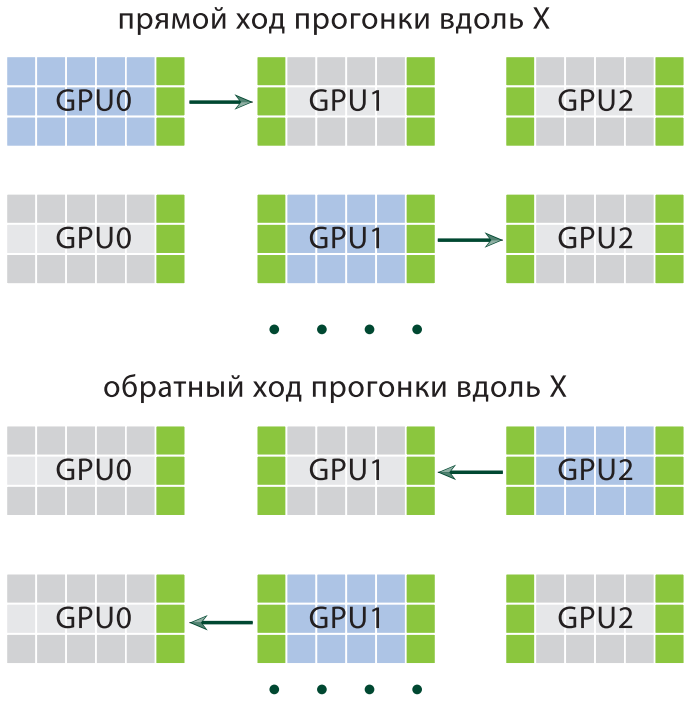
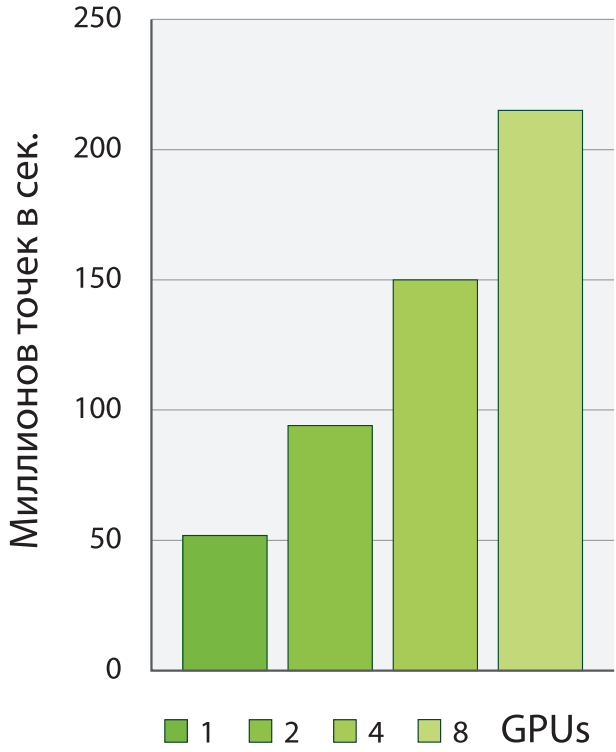


Рис. 12.4. В каждый момент времени работает только один GPU над своей частью прогонки по оси X

Прогонка вдоль направления X разделяется между всеми GPU и выполняется последовательно, по цепочке: каждый GPU сначала ожидает данные, затем вычисляет часть результатов и отправляет их следующему GPU, используя функцию *cudaMemcpyPeer* (рис. 12.4).

Таким образом, с ростом числа GPU, прироста производительности вдоль оси X не будет, и, согласно закону Амдала, это послужит основным лимитирующим фактором производительности при дальнейшем масштабировании. Тем не менее, другие части шага по X – вычисление производных и диссипативной функции – можно эффективно распределить между GPU, за счет чего достигается хороший скейлинг на нескольких GPU, подключенных к одному хосту (рис. 12.5).

В силу того, что вычисление производных у границ требует доступа к граничным значениям частей сетки соседних GPU, необходима их эффективная



1 GPU	2 GPUs	4 GPUs	8 GPUs
51,9	94,1	150,0	215,0

Рис. 12.5. Производительность системы на базе Tesla M2050, измеряемая в количестве точек обработанного объема в секунду для 1–8 GPUs

синхронизация на каждом шаге метода. Такую синхронизацию можно выполнять в асинхронном режиме, см. пункт 11.3.

Полный исходный код модели, использующей данную реализацию метода, можно найти на сайте проекта [22].

12.2. Метод погруженной границы

Пренебрегая эффектами внешних сил и температуры, систему (12.1)–(12.4) можно переписать в следующем виде:

$$\nabla \cdot \vec{\mathbf{u}} = 0, \tag{12.10}$$

$$\rho \left[\frac{\partial \vec{\mathbf{u}}}{\partial t} + \vec{\mathbf{u}} \cdot \nabla \vec{\mathbf{u}} \right] = -\nabla p + \mu \nabla^2 \vec{\mathbf{u}}. \tag{12.11}$$

Будем считать, что область Ω представляет собой параллелепипед со сторонами $L = (L_x, L_y, L_z)$. Определим на области Ω равномерную прямоугольную сетку Ω_h размерности $N = (N_x, N_y, N_z)$ с шагом сетки $h = (h_x, h_y, h_z) = (\frac{L_x}{N_x}, \frac{L_y}{N_y}, \frac{L_z}{N_z})$. Значения скалярных величин (давления p) задаются в центре ячеек, а значения векторных (скорости $\vec{\mathbf{u}}$) – на гранях.

Численное решение системы (12.10)–(12.11) будем искать методом конечных разностей на прямоугольной сетке Ω_h. Для интегрирования уравнений по времени воспользуемся методом дробных шагов. На первом шаге из уравнения движения вычисляется промежуточное поле скорости $\tilde{\mathbf{u}} = (\tilde{u}, \tilde{v}, \tilde{w})$. Предполагая, что известен вектор скорости на n-ом и $(n-1)$-ом шаге по времени и давление на n-ом шаге, вычисление промежуточного поля скорости можно условно записать:

$$\frac{\tilde{\mathbf{u}} - \vec{\mathbf{u}}^{\,n}}{\Delta t} = -[\,\vec{\mathbf{u}} \cdot \nabla \vec{\mathbf{u}}\,]^{n+\frac{1}{2}} - \frac{1}{\rho}\nabla p^n + \frac{\mu}{\rho}[\nabla^2 \vec{\mathbf{u}}\,]^{n+\frac{1}{2}}, \tag{12.12}$$

где Δt – шаг по времени, а для аппроксимации слагаемых на $(n+\frac{1}{2})$-ом шаге используется явная схема Адамса – Бэшфорта второго порядка: $[\cdot]^{n+\frac{1}{2}} = \frac{3}{2}[\cdot]^n - \frac{1}{2}[\cdot]^{n-1}$. Поскольку промежуточное поле скорости не удовлетворяет уравнению неразрывности (12.1), то неизвестные $\vec{\mathbf{u}}^{\,n+1}, p^{n+1}$ будем искать в виде:

$$\vec{\mathbf{u}}^{\,n+1} = \tilde{\mathbf{u}} - \Delta t \nabla \phi^{n+1}, \tag{12.13}$$

$$p^{n+1} = p^n + \rho \phi^{n+1}, \tag{12.14}$$

где поправка к давлению ϕ^{n+1} находится как решение уравнения Пуассона с однородными краевыми условиями Неймана на границе области:

$$\Delta \phi^{n+1} = \frac{\nabla \cdot \tilde{\mathbf{u}}}{\Delta t} \tag{12.15}$$

Конкретный вид конечно-разностных приближений рассматривается в работе [23], где также подробно описываются сеточные приближения нелинейного слагаемого в уравнении (12.12) и их свойства как для второго, так и четвертого порядка точности. Будем считать, что для дискретизации производных по пространству используется схема центральных разностей второго порядка. Время вычисления промежуточного поля скорости (12.12) для явной схемы и операций (12.13), (12.14) несущественно по сравнению со временем, необходимым для решения эллиптического уравнения (12.15).

Считая, что для дискретизации используется схема центральных разностей второго порядка, запишем сеточное уравнение Пуассона, опуская индекс шага по времени:

$$\frac{\phi_{i+1,j,k} - 2\phi_{i,j,k} + \phi_{i-1,j,k}}{h_x^2} + \frac{\phi_{i,j+1,k} - 2\phi_{i,j,k} + \phi_{i,j-1,k}}{h_y^2} + \\ + \frac{\phi_{i,j,k+1} - 2\phi_{i,j,k} + \phi_{i,j,k-1}}{h_z^2} = R_{i,j,k}, \tag{12.16}$$

где $\phi_{i,j,k}$ – определяет значения в центре ячеек, $R_{i,j,k}$ – соответствует сеточной аппроксимации правой части (12.15). Система уравнений (12.16) в матричной записи:

$$Ax = b, \tag{12.17}$$

где матрица A размерности $N_x \times N_y \times N_z$ является симметричной, семидиагональной (пятидиагональной для двумерной задачи).

Одним из способов, позволяющих для выбранной дискретизации уравнений на прямоугольных сетках, рассматривать течения в более сложных областях с криволинейными или нестационарными границами, является *метод погруженной границы*. Основная идея метода состоит в модификации дифференциальной системы уравнений или численной схемы для аппроксимации криволинейной геометрии, сохраняя возможность дискретизации уравнений на простой сетке. Достоинства данного подхода очевидны для параллельных реализаций и моделирования течений в областях с подвижными границами, где в этом случае нет необходимости в построении криволинейных сеток в сложных областях. Опишем общую схему ме-

тода погруженной границы, предложенного в работе [24] (подробный обзор различных методик приведен в статье [25]).

Пусть область Ω содержит набор замкнутых криволинейных границ Γ^k,$k = 1...N_b$ на которых задано краевое условие Дирихле:

$$\vec{\mathbf{u}}|_{\Gamma^k} = \vec{\mathbf{U}}^k(t), \tag{12.18}$$

Для выполнения условий (12.18) в правую часть уравнений движения (12.11) добавляется функция **F**. Нахождение промежуточного поля скорости $\tilde{\mathbf{u}}$ (12.12) разделяется на несколько шагов:

$$\frac{\hat{\mathbf{u}} - \vec{\mathbf{u}}^n}{\Delta t} = -[\,\vec{\mathbf{u}} \cdot \nabla\,\vec{\mathbf{u}}\,]^{n+\frac{1}{2}} - \frac{1}{\rho}\nabla p^n + \frac{\mu}{\rho}[\nabla^2\,\vec{\mathbf{u}}\,]^{n+\frac{1}{2}}, \tag{12.19}$$

$$\frac{\tilde{\mathbf{u}} - \hat{\mathbf{u}}}{\Delta t} = \mathbf{F} = L^* A_b^{-1}\left(\frac{\vec{\mathbf{U}}^k - L\hat{\mathbf{u}}}{\Delta t}\right), A_b = LL^*, \tag{12.20}$$

где L, L^* – специальные матричные операторы интерполяции поля значений, заданного на сетке, в точки границы и интерполяции в узлы сетки по значениям в точках дискретной границы соответственно. При их построении используются сеточные аппроксимации дельта-функций, что делает возможным формулировку метода без учета пространственной связи узлов сетки и точек криволинейной границы, а только с учетом расстояния между ними. Таким образом, вычисление добавочной функции **F** сводится к решению системы с симметричной матрицей A_b, для решения которой можно воспользоваться методом сопряженных градиентов, и матрично-векторным операциям. В случае, если рассматриваются нестационарные границы, то на каждом шаге по времени требуется пересчет матрицы A_b и операторов L, L^*.

12.2.1. Реализация для кластера с множеством GPU

Рассмотрим реализацию описанной выше численной задачи на параллельной архитектуре, а именно на кластере, состоящем из графических процессоров. Такая система предполагает наличие двух уровней параллельной программы: распределение данных между отдельными графическими процессорами и внутрен-

ний параллелизм отдельных устройств. Библиотека MPI используется для организации обменов данными между устройствами, а для организации вычислений на графических процессорах применяется технология CUDA. Вычисления проводились на суперкомпьютере «ГрафИТ!/GraphIT!», где установлены GPU NVIDIA Tesla M2050.

Поскольку область Ω и соответствующая прямоугольная сетка Ω_h имеют простую структуру, то способы декомпозиции области по MPI процессам очевидны. Для реализации сеточных операторов требуется определение дополнительных фиктивных узлов по границе сетки, отвечающих областям соседних процессов. Значения в таких узлах можно получить за счет проведения обменов. В методе дробных шагов такие обмены требуются для вектора промежуточной скорости при вычислении правой части эллиптического уравнения (12.15), для скорости и давления после каждого шага по времени. В методе сопряженных градиентов дополнительные расходы возникают на каждой итерации, здесь необходимы обмены данными при вычислении произведения Ap, а также в процессе решения системы $M\tilde{r} = r$ с недиагональной матрицей M. Взаимодействие процессов MPI предполагается при вычислении скалярных произведений (12.21), (12.26) и проверки сходимости метода.

Каждый MPI-процесс связан с отдельным графическим процессором, в каждом из которых данные распределяются между блоками нитей, сгруппированных в вычислительную сетку. Конечно-разностные приближения реализуются в виде ядер, где каждая нить вычисляет значение сеточного оператора в отдельных точках или наборе точек сетки. Важной особенностью архитектуры графических процессоров является существенная разнородность типов памяти устройства. Необходимо как можно более эффективно использовать быструю память (регистры, разделяемую память между блоками) и минимизировать операции чтения/записи для глобальной памяти. Общую схему вычислений в ядрах можно представить следующим образом: копирование значений из глобальной памяти в быструю память, проведение вычислений, копирование результата в глобальную память.

12.2.2. Оптимизация метода сопряженных градиентов

Поскольку матрица системы (12.17) является симметричной и положительно определенной, для нахождения решения можно воспользоваться предобусловленным методом сопряженных градиентов [26], [27]:

выбрать x^0, *положить* $r^0 = b - Ax^0$

for $k = 0, 1, ...$

$$\alpha_k = -(\tilde{r}^k, r^k)/(p^k, Ap^k) \tag{12.21}$$

$$x^{k+1} = x^k - \alpha_k p^k \tag{12.22}$$

$$r^{k+1} = r^k + \alpha_k A p^k \tag{12.23}$$

проверить сходимость итераций (12.24)

решить систему $M\tilde{r}^{k+1} = r^{k+1}$ (12.25)

$$\beta_k = (\tilde{r}^{k+1}, r^{k+1})/(\tilde{r}^k, r^k) \tag{12.26}$$

$$p^{k+1} = \tilde{r}^{k+1} + \beta_k p^k \tag{12.27}$$

Здесь x^k – приближения решения, r^k – невязка, $\tilde{r}$,p – дополнительные векторы, α, β – скалярные величины, M – матрица предобусловливателя, которая должна сохранять свойства симметричности и положительной определенности.

Отметим, что вычисление матрично-векторного произведения Ap в (12.21) и (12.23) не требует явного задания и хранения матрицы для конечно-разностных аппроксимаций вида (12.16).

В качестве условия сходимости итераций можно использовать $||r^{k+1}||_2 < \epsilon$ (или сравнение относительной погрешности по начальному приближению), где ϵ – известное значение. Сократить вычисления можно за счет дополнительной проверки $(\tilde{r}^{k+1}, r^{k+1}) < \epsilon$. Если данное условие выполнено, то желательна последующая проверка нормы невязки: $||r^{k+1}||_2 < \epsilon$. Такой подход теоретически может приводить к увеличению числа итераций.

Одной из проблем при реализации итерационных методов типа сопряженных градиентов на графических процессорах, как правило, является большое число точек синхронизации, сокращающих вычисления в быстрой памяти. Часть операций

в методе сопряженных градиентов можно разбить на группы, которые можно выполнять совместно: вычисление скалярных произведений (12.21); векторные операции (12.22), (12.23) и проверка сходимости; решение системы (12.25) вместе со скалярными произведениями (12.26); векторная операция (12.27).

Построение матрицы M определяется с одной стороны требованием быстрого решения системы $M\tilde{r} = r$ на рассматриваемой вычислительной архитектуре, а с другой стороны сходимость метода можно улучшить в том случае, если матрица M является хорошим приближением матрицы A системы (12.17).

Решение системы $M\tilde{r} = r$ можно свести к выполнению m шагов дополнительного итерационного процесса [28]:

$$P\tilde{r}^{i+1} = Q\tilde{r}^{i} + r, i = 0, 1, ..., m-1, \tilde{r}^{0} = 0, \tag{12.28}$$

где $A = P - Q$, тогда M определяется соотношением $M = P(I + H + \cdots + H^{m-1})^{-1}$, $H = P^{-1}Q$. В качестве вспомогательного итерационного процесса (12.28) можно рассматривать классические итерационные методы:

– Якоби:

$$P = D, Q = L + L^{T} \tag{12.29}$$

– Симметричный метод Гаусса – Зейделя:

$$P = (D - L)D^{-1}(D - L^{T}), Q = LD^{-1}L^{T}, \tag{12.30}$$

где D – диагональная, а L – строго нижнедиагональная часть матрицы A. Метод (12.30) можно записать как последовательное применение итераций метода Гаусса – Зейделя, который в данном случае неприменим, поскольку не сохраняет симметричность матрицы M, с чередованием направления:

$$(D - L)x^{k+\frac{1}{2}} = L^{T}x^{k} + b, \tag{12.31}$$

$$(D - L^{T})x^{k+1} = Lx^{k+\frac{1}{2}} + b. \tag{12.32}$$

Итерационный процесс (12.31), (12.32) для реализации на параллельных архитектурах вместо решения систем с треугольными матрицами можно свести к

матрично-векторным умножениям за счет переупорядочивания уравнений системы (12.17). Классическим переупорядочиванием для сеточного уравнения Пуассона (12.16) является красно-черное (шахматное) переупорядочивание. Точки сетки разделяются на два подмножества – «красное» и «черное» так, что красные точки связаны только с черными относительно шаблона (12.16), и наоборот. Тогда матрицу системы A можно записать в блочном виде:

$$\begin{bmatrix} D_R & -C \\ -C^T & D_B \end{bmatrix} \tag{12.33}$$

Тогда первый шаг (12.31) сводится к соотношениям:

$$\begin{bmatrix} D_R & 0 \\ -C^T & D_B \end{bmatrix} \begin{bmatrix} u_R^{k+1} \\ u_B^{k+1} \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} - \begin{bmatrix} 0 & C \\ 0 & 0 \end{bmatrix} \begin{bmatrix} u_R^{k} \\ u_B^{k} \end{bmatrix} \tag{12.34}$$

$$u_R^{k+1} = D_R^{-1}(b_1 - Cu_B^k), \tag{12.35}$$

$$u_B^{k+1} = D_B^{-1}(b_2 - C^T u_R^{k+1}). \tag{12.36}$$

И аналогичным соотношениям для обратного шага (12.32). Часть вычислений можно сократить за счет удаления повторяющихся вычислений прямых и обратных пересчетов одного цвета. Для схем высокого порядка можно получить аналогичный результат за счет определения дополнительных цветов [28].

Заметим, что методы (12.29), (12.30) допускают введение параметра релаксации ω для ускорения сходимости. Симметричный метод Гаусса – Зейделя является частным случаем метода SSOR с параметром релаксации $\omega = 1$, что соответствует оптимальному значению для предобусловливателя в методе сопряженных градиентов, записанному в красно-черной форме [29].

На рис. 12.6 приведен график времени счета при увеличении числа графических процессоров для предобусловливателя Якоби (12.29) при различных числах m. Численно решалась задача о течение вокруг неподвижной сферы на сетке, состоящей из 12×10^6 узлов. Для аппроксимации краевых условий на поверхности сферы использовался метод погруженной границы. При $m = 1$ необходим единственный обмен при вычислении произведения Ap, а умножение диагональной матрицы на вектор $D^{-1}r$ можно объединить с векторной операцией (12.22) и

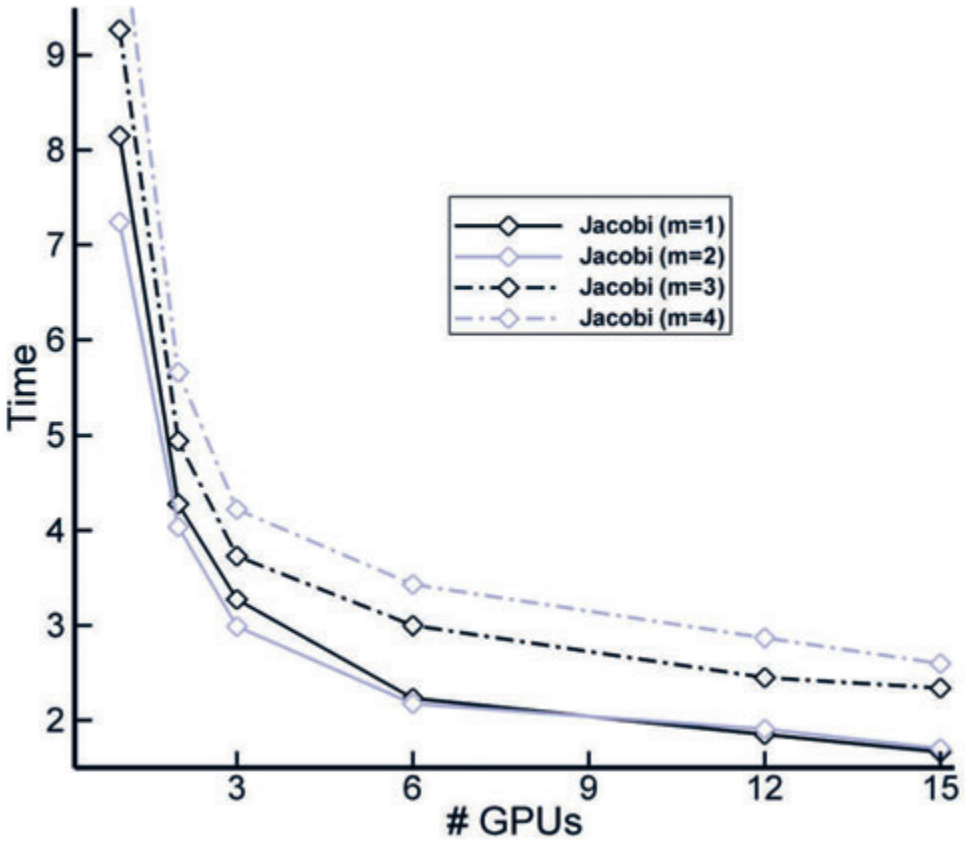


Рис. 12.6. Применение предобусловливателя (12.29) при $m = 1, 2, 3, 4$. По вертикальной оси приведено время счета в секундах, по горизонтальной – число GPU

вычислением скалярного произведения (12.26). Тем не менее, такое приближение не приводит к значимому снижению числа итераций метода сопряженных градиентов, а значит и уменьшению числа обменов и объема вычислений. Для $m = 1$ наблюдается существенное уменьшение числа итераций метода сопряженных градиентов, однако увеличение числа обменов внутри цикла снижает общую эффективность. Выполнение дополнительных шагов далее приводит лишь к увеличению затрат на обмены и не дает значительного выигрыша относительно сходимости метода. Основной проблемой при реализации итераций метода Якоби является значительная доля обменов.

При использовании симметричного метода Гаусса – Зейделя наблюдается схожая ситуация, однако при эффективной реализации операций (12.31), (12.32) удается получить небольшое преимущество относительно метода Якоби. С учетом особенностей доступа памяти на графических процессорах и организации обменов желательно отдельно хранить значения в красных и черных точках, что также требует дополнительных операций копирования и достаточно оправдано при увеличении числа шагов вспомогательного итерационного процесса.

Интересным способом построения предобусловливателей для параллельных архитектур являются методы аппроксимации обратной матрицы: $M^{-1} \approx A^{-1}$ [30]. В этом случае шаг решения системы $M\tilde{r} = r$ (12.25) сводится к умножению матрицы на вектор: $\tilde{r} = M^{-1}r$, который можно эффективно реализовать, если матрица M^{-1} сохраняет разряженную структуру. В работе [31] предложена следующая аппроксимация матрицы A^{-1} для уравнения Пуассона:

$$M^{-1} = (I - LD^{-1})(I - D^{-1}L^T) \tag{12.37}$$

Поскольку выполнение обменов между MPI процессами, как правило, связано с матрично-векторными умножениями, то данное обстоятельство можно использовать для одновременного выполнения обменов и вычислений произведения во внутренних точках сетки, шаблон которых не использует внешние точки. Для предобусловливателя (12.37) такой подход может быть реализован совместно с использованием модификации сопряженных градиентов [32], позволяющей разбить умножение $M^{-1}r$ на два шага. При этом увеличивается число вычислений, которое можно реализовать одновременно с проведением обменов. Преимущество такого подхода для предобусловливателя (12.37) приводится на рис. 12.7 для описанной выше задачи. Отметим, что общее время счета в обоих случаях меньше, чем для метода Якоби, рассмотренного выше.

Аналогичную схему одновременного проведения обменов и вычислений можно использовать и при предобусловливании системы с помощью итерационных методов (12.29)–(12.30).

Существуют другие подходы к предобусловливанию. Например, построение матрицы M из неполной факторизации матрицы $A = HH^T + K$, $M = HH^T$. Основная идея неполной факторизации состоит в возможности сохранения раз-

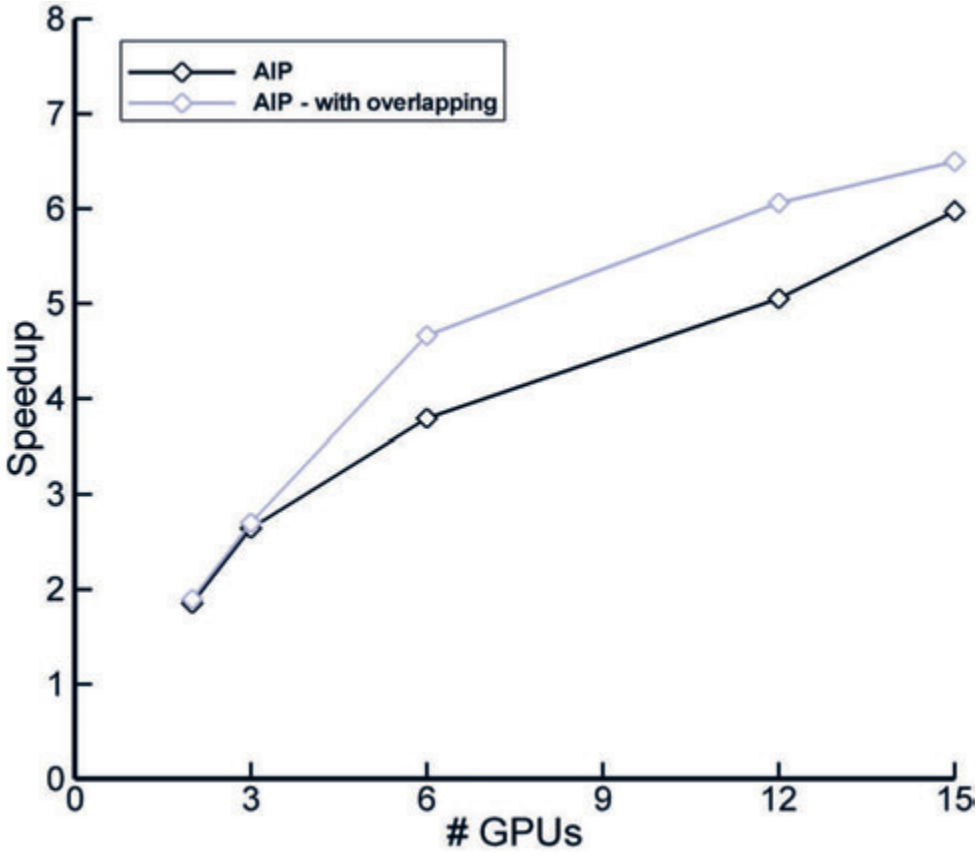


Рис. 12.7. Реализация предобусловливателя (12.37). По вертикальной оси приведено ускорение по сравнению со временем расчета на одном GPU, по горизонтальной – число GPU

ряженной структуры для нижнетреугольной матрицы H. Тогда решение предобусловленной системы сводится к решению двух систем: $Hq = r$, $H^T\tilde{r} = q$. К таким методам можно отнести и симметричный метод Гаусса – Зейделя 12.30. В общем случае существенные трудности вызывает параллельная реализация решения системы $M\tilde{r} = r$, а также построения неполной факторизации, когда на каждом шаге интегрирования уравнений (12.12)–(12.15) требуется пересчет матрицы A.

Одним из наиболее эффективных методов численного решения уравнения Пуассона являются многосеточные методы [33]. В таких методах численно решается

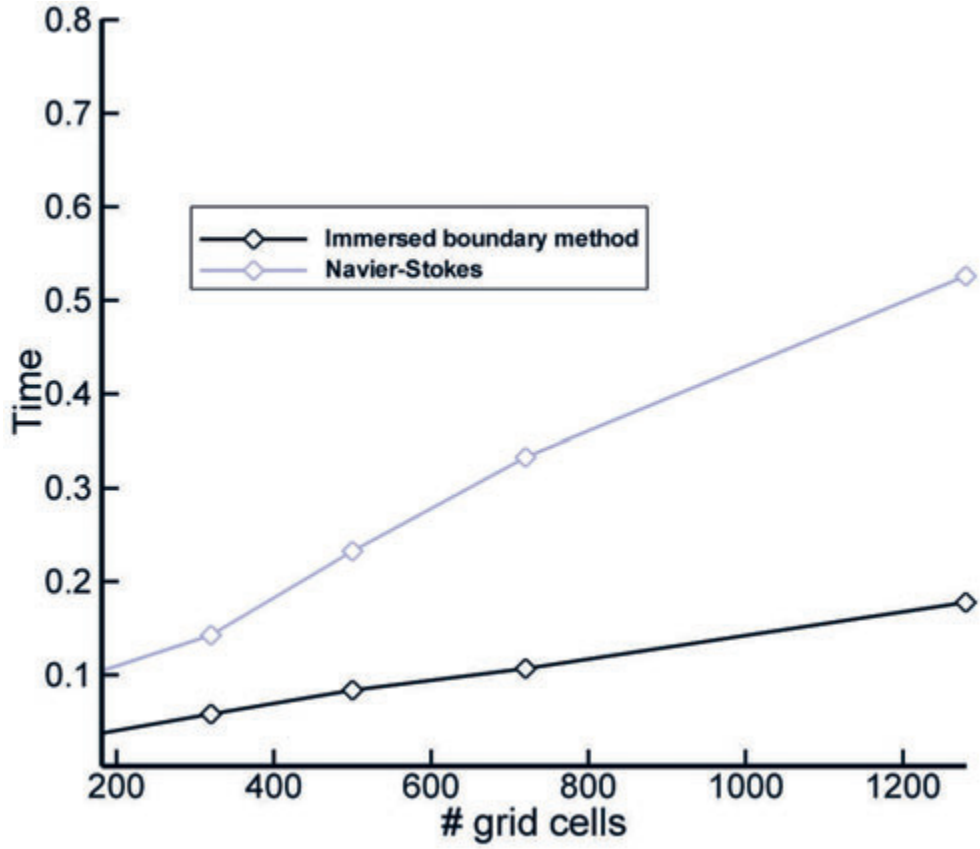


Рис. 12.8. Время расчета одного полного шага и операций метода погруженной границы для течения вокруг кругового цилиндра, совершающего вынужденные колебания. По вертикальной оси приведено время счета в секундах, по горизонтальной – число точек сетки (в тысячах)

последовательность систем вида (12.17) на все более грубых сетках. Для решения систем можно использовать итерационные методы (12.29), (12.30). Одной из проблем при реализации многосеточных методов является невозможность использования всех ресурсов графических процессоров при вычислениях на грубых сетках, а также большое число обменов. Применимость подобных методов зависит от простоты построения последовательности сеток и аппроксимаций вида (12.16), что затруднительно для ряда задач и дискретизаций уравнений на сложных сетках.

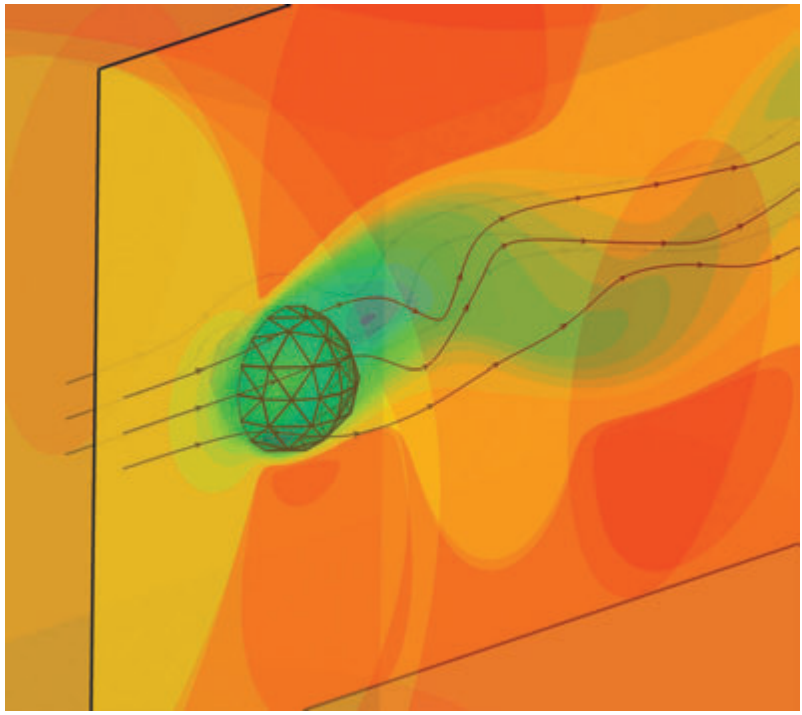


Рис. 12.9. Течение вокруг сферы. Профиль продольной компоненты скорости

Для такой задачи общее время вычислений для операций метода погруженной границы мало по сравнению со временем решения уравнения Пуассона. Но при увеличении числа погруженных областей и решении задач с нестационарными границами, вычислением добавочной функции (12.20) и пересчетом матрицы A_b уже нельзя пренебречь. На рисунке 12.9 приведено сравнение времен вычислений для метода погруженной границы и для интегрирования уравнений на последовательности сеток. В качестве численной задачи рассматривается течение вокруг кругового цилиндра, совершающего вынужденные колебания. Для решения системы с матрицей предобусловливателя применяется многосеточный метод. Очевидно, что быстродействие операций метода погруженной границы для таких задач еще более существенно при распределении данных между графическими процессорами.

Полный исходный код модели, использующей данную реализацию метода, можно найти на сайте проекта [34].

Глава 13

Методы трассировки лучей на GPU

Трассировка лучей – это метод, позволяющий восстановить непрерывное изображение из дискретного пиксельного представления (матрицы дисплея) с помощью математической модели интенсивности освещения. Каждой точке плоскости экрана соответствует точка x, освещенность в которой может быть рассчитана, например, при помощи интеграла следующего вида [35]:

$$I(\phi_r, \Theta_r) = \int_{\phi_i} \int_{\Theta_i} L(\phi_i, \Theta_i) R(\phi_i, \Theta_i, \phi_r, \Theta_r) d\phi_i d\Theta_i \qquad (13.1)$$

$L(\phi_i, \Theta_i)$ – это функция, описывающая общее освещение, падающее в точку x под всеми возможными углами в пределах полусферы. $R(\phi_i, \Theta_i, \phi_r, \Theta_r)$ – двунаправленная функция распределения отражения (bidirectional reflectance distribution function, BRDF). Она полностью описывает характер взаимодействия света с конкретной поверхностью, связывая интенсивность и угол падающего света с интенсивностью и углом отраженного или пропущенного прозрачной поверхностью света (рис. 13.1).

Формула (13.1) показывает, что интенсивность освещения в точке $I(\phi_r, \Theta_r)$ является функцией интенсивности света, отраженного поверхностью под разными углами. В результате интегрирования по полусфере приходящего (падающего) в точку освещения L, с учетом свойств поверхности R, получается новая функция распределения освещения в пространстве, обусловленная отражающими свойствами поверхности (материала) в точке x. Другими словами, освещенность в точке поверхности складывается из света, падающего на поверхность со всех направлений и отраженного по некоторому закону, который определяет BRDF. Формула (13.1) применима только для сравнительно простых материалов. Так, в случае стекла необходимо дополнительно учитывать свет, приходящий из-под поверхности и проводить интегрирование по полной сфере. В случае кожи требуется более сложная модель, учитывающая подповерхностное рассеивание.

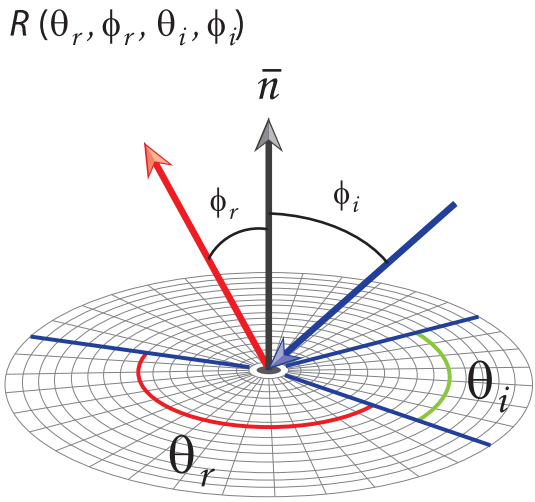


Рис. 13.1. BRDF – функция, зависящая от параметров входного и выходного лучей

13.1. Обратная трассировка лучей

Пусть из виртуальной точки обзора («глаза») через каждый пиксел изображения пропускается луч, и находится точка его пересечения с поверхностью сцены (для простоты, объемные эффекты типа тумана не рассматриваются). Лучи, выпущенные из глаза называются *первичными*. Допустим, первичный луч пересекает некий объект 1 в точке H1 (рис. 13.2).

Далее необходимо определить для каждого источника освещения, видна ли из него эта точка. Если все источники света точечные, то чтобы определить, освещается ли точка $H1$, достаточно испустить из нее в каждый источник *теневой луч*. Наличие пересечений теневого луча с другими объектами будет означать, что точка $H1$ находится в тени этого источника, и освещать ее не нужно. Освещение рассчитывается отдельно для каждого видимого источника по некоторой локальной модели (Фонг, Кук-Торранс и.т.д.), и затем вклады всех источников суммируются. Далее, если материал объекта имеет отражающие свойства, то из точки $H1$ испускается отраженный луч, и для него вся процедура трассировки рекурсивно повторяется. Аналогичные действия должны быть выполнены, если материал имеет преломляющие свойства. Если материалы в сцене имеют только отражающие свойства, как

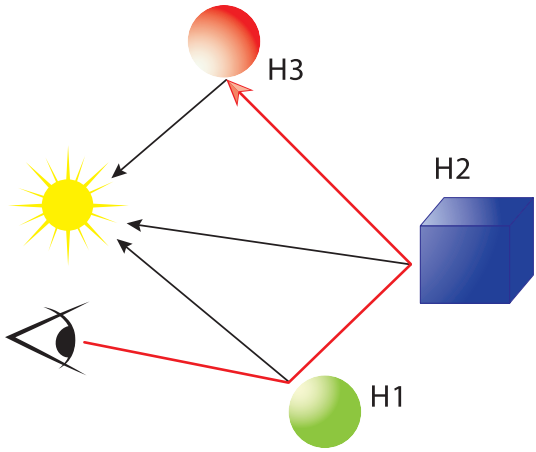


Рис. 13.2. Алгоритм обратной трассировки лучей

на рис. 13.2, то преломленный луч никогда не генерируется, что позволяет избежать рекурсии, заменив ее циклом. Такую модель можно без существенных затруднений реализовать на GPU.

```
float3 RayTrace(const Ray& ray)
{
    float3 color(0,0,0);
    Hit hit = RaySceneIntersection(ray);
    if (!hit.exist)
        return color;

    float3 hit_point = ray.pos + ray.dir*hit.t;

    for (int i=0;i<NLights;i++)
        if (Visible(hit_point, lights[i]))
            color += Shade(hit, lights[i]);

    if (hit.material.reflection > 0)
    {
        Ray reflRay = reflect(ray, hit);
        color += RayTrace(reflRay);
    }

    if (hit.material.refraction > 0)
    {
        Ray refrRay = refract(ray, hit);
        color += RayTrace(refrRay);
```

```
    }

    return color;
}
```

В данном фрагменте программы луч представлен двумя векторными переменными (*float3*): *pos* – точка испускания луча, *dir* – нормализованное направление луча. Цвет является вектором значений трех своих компонентов: синий, красный и зеленый. В самом начале функции *RayTrace* рассчитывается точка пересечения луча со сценой, представленной некоторым списком объектов. Информация о точке пересечения сохраняется в переменной *hit*, а расстояние до точки пересечения – в переменной *hit.t*. Если луч не пересекает сцену, то возвращается фоновый цвет, например, черный. При наличии пересечения, точка *hit_point* вычисляется при помощи уравнения прямой. После вычисления точки пересечения в мировых координатах можно приступать к расчету теней. Пусть координаты источников света заданы массивом *lights*. Тогда необходимо проверить видимость каждого источника из точки *hit_point*. Если источник виден, то его освещение аккумулируется в переменной *color*. Для объектов, сделанных из материала с отражающими или преломляющими свойствами, производится рекурсивная трассировка лучей с аккумулированием в *color* полученного цвета, умноженного на соответствующий коэффициент отражения или преломления. Коэффициенты *reflection* и *refraction* могут быть как монохромными, так и цветными.

В случае наличия перекрытий одного объекта другим прозрачным объектом используются *цветные* теневые лучи. В зависимости от длины пути теневого луча в толще прозрачного объекта, тень может приобрести какой-либо оттенок (если объект им обладает). Разумеется, тени, рассчитанные таким образом, корректны, только если прозрачный объект, отбрасывающий тень, имеет очень близкий к единице коэффициент преломления (коэффициент преломления воздуха считается равным 1). Если это не так, то под прозрачным объектом образуется сложная картина, называемая *каустиком*. Каустики рассчитываются отдельно с помощью метода фотонных карт [36]. Типичный пример каустика – «солнечный зайчик» от стакана воды, когда через него просвечивает солнце.

Описанный метод обратной трассировки лучей не позволяет получить фотореалистичную картину, поскольку корректно рассчитывает лишь четкие отражения,

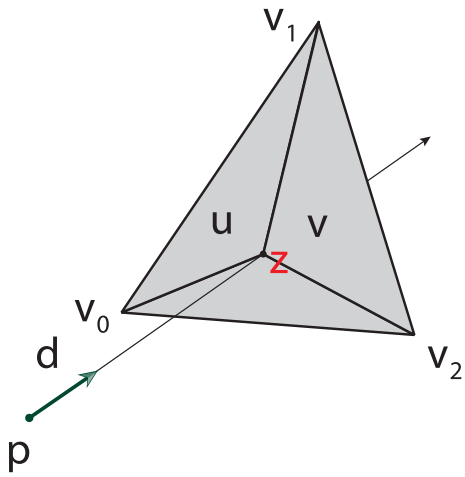


Рис. 13.3. Пересечение луча и треугольника

преломления и устраняет ступенчатость при большом числе лучей на пиксел (что медленно). Чтобы получить весь спектр видимых эффектов, необходимо использовать более сложные алгоритмы, которые, тем не менее, имеют в своей основе трассировку лучей.

13.2. Поиск пересечений

В компьютерной графике сцена состоит из набора *примитивов* – объектов, пересечение луча с которыми можно рассчитать по простым аналитическим формулам. В качестве примитивов могут выступать любые геометрические фигуры (сфера, цилиндр, конус, тор), параметрические кривые (*сплайновые поверхности* или *поверхности Безье*), но самым распространенным примитивом, используемым для представления части поверхности, является треугольник. Поэтому именно случай расчета пересечения луча и треугольника рассматривается далее (рис. 13.3).

Любую точку плоскости треугольника можно выразить с использованием *барицентрических координат*, связанных с координатами вершин треугольника. Причем, если точка находится внутри треугольника, сумма трех барицентрических координат будет равна единице. Барицентрические координаты соответствуют отношениям площадей маленьких треугольников к площади большого треугольника:

$$u = \frac{S(v0, z, v1)}{S(v0, v1, v2)}, \tag{13.2}$$

$$v = \frac{S(v1, z, v2)}{S(v0, v1, v2)}. \tag{13.3}$$

Пусть точка z – искомая точка пересечения луча и треугольника. Тогда, используя барицентрические координаты, получим первое уравнение:

$$z(u, v) = (1 - u - v) * v1 + u * v2 + v * v0.$$

С другой стороны, точка z лежит на луче:

$$z(t) = p + t * d.$$

Приравняв правые части этих двух уравнений 1 и 2, получаем третье уравнение (в векторном виде):

$$p + t * d = (1 - u - v) * v1 + u * v2 + v * v0.$$

В результате несложных преобразований могут быть получены искомые величины: две барицентрические координаты (u, v) и расстояние до точки пересечения – t:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{dot(P, E1)} * \begin{bmatrix} dot(Q, E2) \\ dot(P, T) \\ dot(Q, d) \end{bmatrix},$$

где $E1 = v1 - v0, E2 = v2 - v0, T = p - v0, P = cross(d, E2), Q = cross(T, E1)$.

Зная барицентрические координаты, несложно вычислить все остальные параметры в точке z: положение точки, нормаль, текстурные координаты. Для этого нужно умножить каждый параметр в вершине треугольника на соответствующую ему барицентрическую координату и сложить результат со всех трех вершин.

Еще один знаменитый алгоритм поиска пересечения луча и треугольника, который может улучшить скорость трассировки лучей на GPU, носит название *Woop's Unit Test*. Основная идея данного алгоритма заключается в том, чтобы посчитать

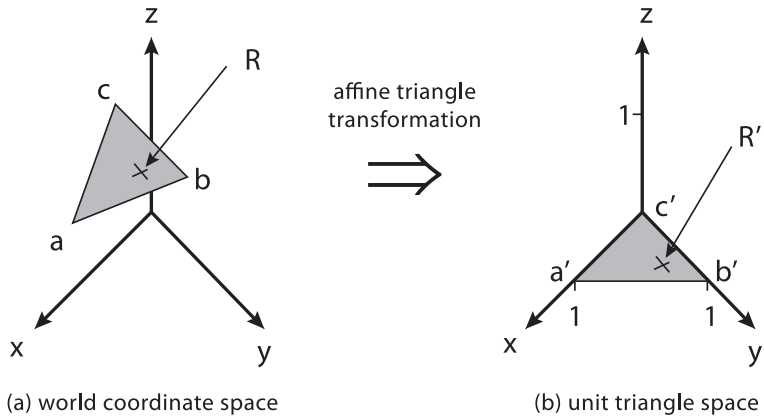


Рис. 13.4. Афинное преобразование треугольника в пространство, где он имеет единичное представление

матрицу преобразования треугольника в некий единичный треугольник (отсюда происходит название метода) с вершинами $(1,0,0);(0,1,0);(0,0,0)$. Во время подсчета пересечения луч преобразуется этой матрицей в пространство, где треугольник имеет единичное представление. После преобразования вычислить пересечение намного проще, так как нужно считать пересечение с заранее известным треугольником $(1,0,0);(0,1,0);(0,0,0)$ (рис. 13.4).

Пусть задан треугольник с вершинами A, B, C. Рассмотрим преобразование $T_{\triangle}^{-1}$:

$$T_{\triangle}^{-1}\begin{pmatrix}1\\0\\0\end{pmatrix}=A,\qquad T_{\triangle}^{-1}\begin{pmatrix}0\\1\\0\end{pmatrix}=B,$$

$$T_{\triangle}^{-1}\begin{pmatrix}0\\0\\0\end{pmatrix}=C,\qquad T_{\triangle}^{-1}\begin{pmatrix}0\\0\\1\end{pmatrix}=N,$$

Для того, чтобы найти искомое преобразование, необходимо дополнить матрицу $T_{\triangle}^{-1}$ до 4×4 еще одной строкой – $(0,0,0,1)$ – и найти к ней обратную матрицу. Это будет соответствовать обратному $T_{\triangle}^{-1}$-преобразованию. Далее остается лишь

умножить луч на полученную матрицу и посчитать пересечение с единичным треугольником:

```
float3 o = mul3x4(m, ray_origin);
float3 d = mul3x3(m, ray_direction);
float t = -o.z/d.z;
float u = o.x + t*d.x;
float v = o.y + t*d.y;
```

Функция *mul3x4* выполняет умножение подматрицы 3×3 на трехмерный вектор и добавляет к результату последний столбец (3 его компоненты). Функция *mul3x3* умножает подматрицу 3×3 на трехмерный вектор.

13.3. Ускорение поиска пересечений

Самой ресурсоемкой частью трассировки лучей обычно является именно поиск пересечений. Проблема заключается в том, что примитивов в сцене обычно достаточно много (порядка миллиона). Даже при использовании ускоряющих структур в большинстве случаев все равно приходится проверять достаточно много объектов для одного луча. И конечно, о том, чтобы искать пересечения методом грубой силы, проверяя последовательно все примитивы, не может быть и речи.

13.3.1. Регулярная сетка

Самый очевидный алгоритм решения – разбить трехмерное пространство равномерной сеткой (рис. 13.5) и рассматривать только те кубы (воксели), через которые прошел луч.

В трехмерном *алгоритме Брезенхема* используется целочисленное приближение луча. Оно хорошо подходит для быстрого приблизительного рисования линий, но совершенно не годится для аккуратного обхода вокселей, поскольку существует вероятность того, что луч попадет не в те воксели. Чем больше вокселей в сетке, тем это приближение точнее, и вероятность промаха меньше, но тем медленнее работает поиск (рис. 13.6).

Другой алгоритм – 3DDA или *алгоритм Фуджимото* (Fujimoto) обходит сетку более аккуратно за счет использования арифметики с плавающей точкой. Он состоит из двух частей: инициализации и основной части. Инициализирующая часть

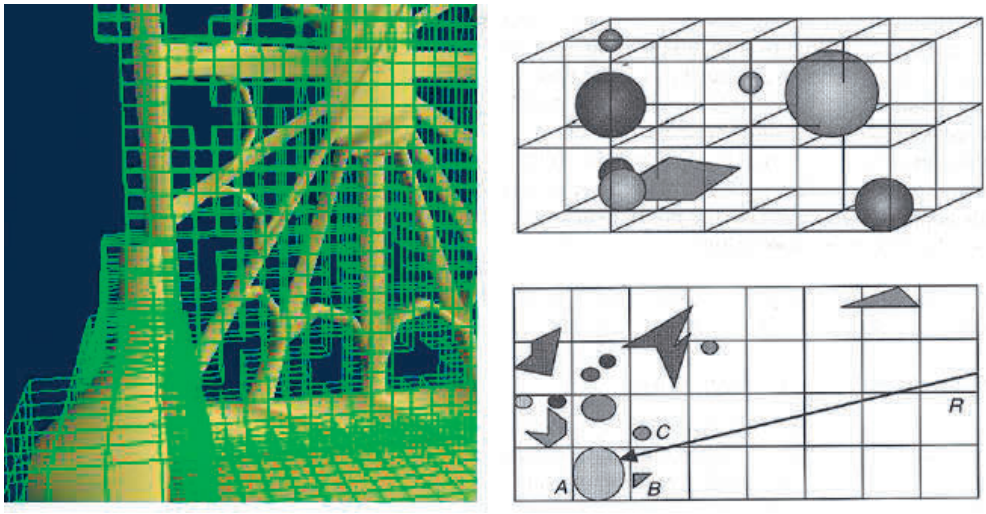


Рис. 13.5. Регулярная сетка

довольно сложна в вычислительном плане, однако выполняется она единственный раз. Ее цель – посчитать некоторые величины, которые будут использоваться в основной части.

```
float t_min;

if(!IntersectRay2fBox2f(ray, box, t_min))
    return;

if(t_min < 0)
    t_min = 0;

const float startX = ray.pos.x + t_min*ray.dir.x;
const float startY = ray.pos.y + t_min*ray.dir.y;

int x = (int)((startX - box.vmin.x)/(box.vmax.x - box.vmin.x)*CELL_NUMBER);
int y = (int)((startY - box.vmin.y)/(box.vmax.y - box.vmin.y)*CELL_NUMBER);

if(x == CELL_NUMBER) x--;
if(y == CELL_NUMBER) y--;

float2 boxSize = box.vmax - box.vmin;

float tVoxelX, tVoxelY;
int stepX, stepY;
```

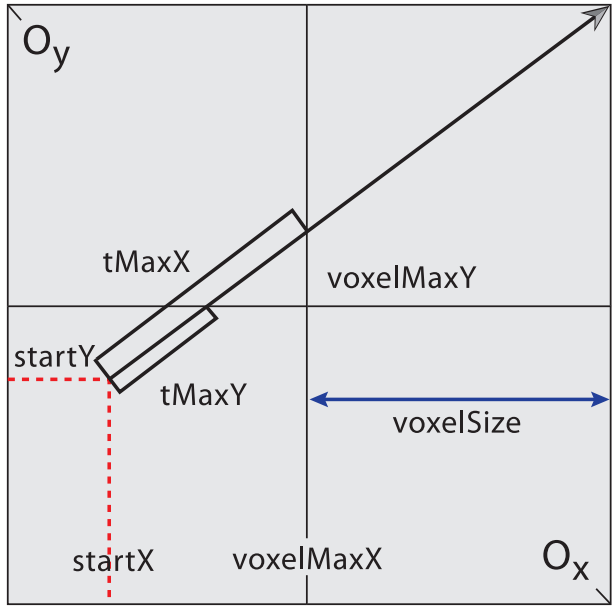


Рис. 13.6. Иллюстрация алгоритма прослеживания луча в регулярной сетке

```
if(ray.dir.x > 0)
{
    tVoxelX  = (float)(x+1)/(float)CELL_NUMBER;
    stepX = 1;
}
else
{
    tVoxelX  = (float)x/(float)CELL_NUMBER;
    stepX = -1;
}

if(ray.dir.y > 0)
{
    tVoxelY = (float)(y+1)/(float)CELL_NUMBER;
    stepY = 1;
}
else
{
    tVoxelY = (float)y/(float)CELL_NUMBER;
    stepY = -1;
}
```

```
float voxelMaxX = box.vmin.x + tVoxelX*boxSize.x;
float voxelMaxY = box.vmin.y + tVoxelY*boxSize.y;

float tMaxX = t_min + (voxelMaxX – startX)/ray.dir.x;
float tMaxY = t_min + (voxelMaxY – startY)/ray.dir.y;

const float voxelSize = (box.vmax.x – box.vmin.x)/CELL_NUMBER;
const float tDeltaX = voxelSize/fabs(ray.dir.x);
const float tDeltaY = voxelSize/fabs(ray.dir.y);
```

Основная часть очень проста. Для начала рассмотрим двумерный случай:

```
while(x < N && x >= 0 &&
      y < N && y >= 0)
{
    NextVoxel(x,y);
    if (tMaxX < tMaxY)
    {
        tMaxX += tDeltaX
        x += stepX
    }
    else
    {
        tMaxY += tDeltaY
        y += stepY
    }
}
```

В 3D алгоритм выглядит не сложнее, чем 2DDA:

```
while(x < N && x >= 0 &&
      y < N && y >= 0 &&
      z < N && z >= 0)
{
    NextVoxel(x,y,z);

    if (tMaxX <= tMaxY && tMaxX <= tMaxZ)
    {
        tMaxX += tDeltaX
        x += stepX
    }
    else if(tMaxY <= tMaxX && tMaxY <= tMaxZ)
    {
        tMaxY += tDeltaY
        y += stepY
    }
```

```
    else
    {
        tMaxZ += tDeltaZ
        z += stepZ
    }
}
```

Несмотря на то, что алгоритм выглядит очень просто и может быть эффективно реализован на GPU, он имеет ряд недостатков, характерных как для CPU- так и для GPU-реализации:

1. Проблема «чайника на стадионе»: регулярная сетка не подходит для реальных сцен, содержащих разномасштабные примитивы. Попытка сделать сетку многоуровневой (иерархической) приводит к тому, что тяжелая инициализирующая часть сводит на нет выигрыш от простоты алгоритма.

2. Требует много памяти. Памяти на GPU не так много.

3. Проблема повторных пересечений. Так как один и тот же треугольник может попадать во множество узлов сетки, при обходе сетки луч может несколько раз посчитать пересечение с одним и тем же треугольником. При реализации на CPU эта проблема решается достаточно просто – для каждого треугольника записывается индекс луча, с которым последний раз считалось пересечение. При подсчете пересечений сначала проверяется, не равен ли индекс текущего луча индексу луча, сохраненного для треугольника. Если равен, то пересечение не считается. Но для GPU такой подход неприемлем, так как использование атомарных операций (во избежание конкурирующих записей индекса луча для одного и того же треугольника несколькими нитями) может существенно замедлить работу.

Таким образом, регулярная сетка не вполне подходит для реализации трассировки лучей на GPU. Поэтому более разумно использовать иерархические представления, такие как BVH и kd-деревья.

13.3.2. Kd-дерево

Рассмотрим структуру бинарного пространственного разбиения, называемого *kd-дерево*. Эта структура представляет собой бинарное дерево ограничивающих

параллелепипедов («боксов»), вложенных друг в друга. Каждый параллелепипед в kd-дереве разбивается плоскостью, перпендикулярной одной из осей координат на два дочерних параллелепипеда. Вся сцена целиком содержится внутри корневого параллелепипеда, но продолжая рекурсивное разбиение параллелепипедов, можно прийти к тому, что в каждом листовом параллелепипеде будет содержаться лишь небольшое число примитивов. Таким образом, kd-дерево позволяет использовать бинарный поиск для нахождения примитива, пересекаемого лучом.

Построение kd-дерева. Алгоритм построения kd-дерева можно представить следующим образом:

1. Построить «глобальный» бокс, ограничивающий все примитивы сцены, который будет соответствовать корневому узлу дерева.
2. Если примитивов в узле мало или достигнут предел глубины дерева, завершить построение.
3. Выбрать плоскость разбиения, которая делит данный узел на два дочерних – правый и левый.
4. Добавить примитивы, пересекающиеся с боксом левого узла в левый узел, правого узла – в правый.
5. Для каждого из узлов рекурсивно выполнить данный алгоритм начиная с шага 2.

Наиболее важным в алгоритме построения kd-дерева является шаг 3, поскольку от него напрямую зависит эффективность ускоряющей структуры. Существует несколько способов выбора плоскости разбиения, их подробное описание можно найти, например, в [37].

Поиск в kd-дереве. Классический алгоритм бинарного поиска в kd-деревьях (kd-tree traversal в англоязычной литературе), применяющийся в большинстве CPU-реализаций, состоит в следующем. На первом шаге алгоритма необходимо посчитать пересечение луча с ограничивающим сцену корневым параллелепипедом и

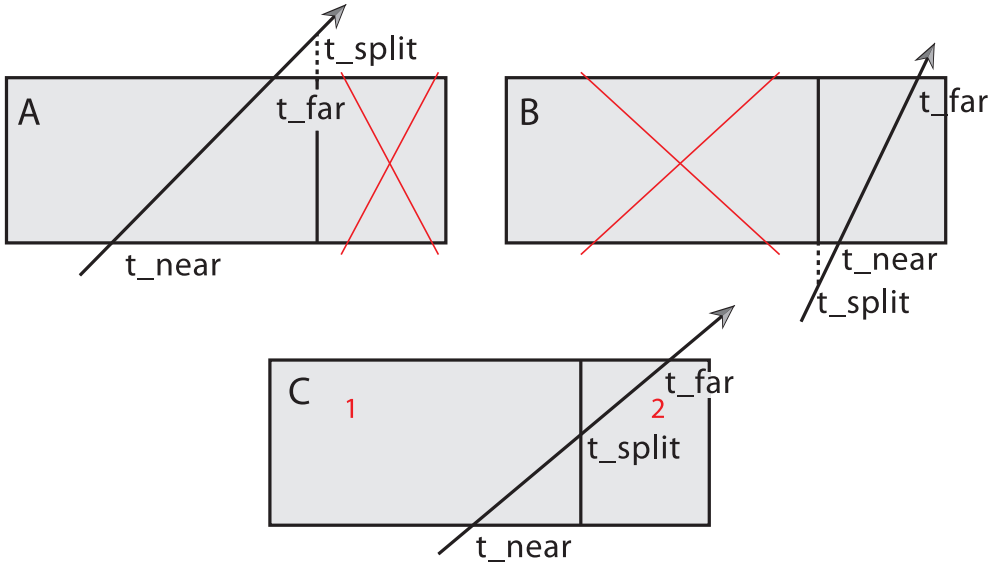


Рис. 13.7. 3 варианта событий при поиске в kd-дереве

запомнить информацию о пересечении в виде двух координат (в пространстве луча) – *t_near* и *t_far*, обозначающих пересечение с ближней и дальней плоскостями соответственно. На каждом следующем шаге необходима информация только о текущем узле (его адрес) и координатах двух плоскостей. При этом нет необходимости вычислять пересечение луча и дочернего параллелепипеда, достаточно лишь узнать пересечение с разбивающей параллелепипед плоскостью (обозначим соответствующую координату как *t_split*). Каждый нелистовой узел kd-дерева имеет два дочерних узла (рис. 13.7).

```
bool KdTreeTraverse(Ray ray)
{
    (hit, t_near, t_far) = RayBoxIntersection(ray, scene_box);

    if ((!hit) or (t_near > t_far))
        return false;

    if (t_near < 0)
        t_near = 0;

    while (true)
    {
```

```
        while (!node.Leaf())
        {
            axis = node.GetAxis(); // axis = 0..2
            t_split = (node.GetSplitPos() – ray.pos[axis])/ray.dir[axis];

            if (t_split < t_near)
                node = node.Near();
            else if (t_split > t_far)
                node = node.Far();
            else
            {
                stack.push(node.Far(), t_far)
                t_far = t_split;
            }
        }

        t_hit = IntersectAllPrimitivesInLeaf(ray,node)
        if (t_hit <= t_far)
            return true;

        if (stack.empty())
            return false;

        t_near = t_far;
        (node, t_far) = stack.pop();
    }
}
```

13.4. Советы по оптимизации

В этом разделе приводится ряд общих советов по разработке кода для GPU на примере методов трассировки лучей.

Вычисления вместо данных. На GPU выгодны избыточные вычисления, если они позволяют выполнить меньше запросов в память.

Принцип KISS (Keep It Simple, Stupid). При разработке кода для GPU важно предвидеть, во что транслируется код. При этом нет необходимости в глубоком знании ассемблера, достаточно понимать, какой по характеру код на низком уровне будет получен:

- Типы циклов: полностью динамические, развернутые или частично развернутые. Циклы лучше разворачивать явно, не надеясь на то, что это сделает компилятор.

- Наличие зависимостей между обращениями в глобальную и разделяемую память.

- Атомарные операции и точки синхронизации

Пониманию генерируемого кода не способствует чрезмерное использование шаблонов C++ и глубоких каскадов простых функций.

Нехватка регистров. Одна из проблем трассировки лучей на GPU состоит в том, что для достижения оптимальной эффективности алгоритма не хватает регистров. Для того чтобы занятость (occupancy) мультипроцессоров была высокой, каждая нить исполнения должна занимать как можно меньше регистров.

Хранение 3 вершин треугольника по 3 float-координаты потребует 9 регистров. Луч – 6 регистров. С учетом множества промежуточных переменных и вспомогательных счетчиков (идентификатор потока, счетчик цикла по треугольникам и проч.), получается, что даже в теории одно только пересечение луча и треугольника занимает более 20 регистров. Такое количество находится за пределами наиболее эффективных значений как для устройств архитектуры GT200, так и для Fermi. Более того, помимо расчета пересечений, необходимо выполнять множество других действий: «траверсить» (выполнять поиск) ускоряющие структуры, делать затенение, генерировать новые теневые, отраженные и преломленные лучи.

Один из способов оптимизации – это использование алгоритмов, которые позволяют сэкономить регистры. Например, в качестве теста пересечения луча и треугольника можно использовать Woop's Unit Test вместо наиболее распространенного барицентрического теста. Другой способ – компоновка функциональности в ядра с использованием одной из приведенных ниже стратегий:

- *Uber kernel* – в случае критической нехватки регистров в качестве стандартного решения используется паттерн *uber-kernel* (другое название – *mega-kernel*). Сложный код делится некоторым образом на части. В ядре присут-

ствуют все части, но каждая в своей ветке if. Также имеется флаг, отвечающий за то, какая часть кода должна выполняться. Во время выполнения процессор может периодически переключаться с одной части кода на другую, сохраняя некоторые важные данные в разделяемой или локальной памяти. Таким образом, можно использовать одни и те же регистры для разных переменных.

- *Separate kernel* – основная идея этого подхода состоит в разбиении сложного кода на несколько ядер. Причем критичный по времени выполнения код следует помещать в небольшое, как можно более оптимизированное ядро. Так, для трассировки лучей в отдельное ядро следует выделить наиболее ресурсоемкую часть – поиск пересечений. Затенение, текстурирование и генерацию новых лучей можно выполнять в одном ядре, но лучше также в нескольких отдельных ядрах.

Использование оператора «?» вместо if-ветвлений. Ветвления, как правило, сами по себе немного уменьшают скорость выполнения программы, т.к. их наличие заставляет GPU выполнять некоторые дополнительные действия по маскированию потоков и синхронизации. Также, в отличие от CPU, на GPU тернарный оператор (“cond ? a : b”) с высокой вероятностью может быть транслирован в более эффективные предикатные инструкции. Один из важных примеров, где можно обойтись без ветвлений, – это расчет пересечения луча и выровненного по координатным осям ограничивающего бокса:

```
inline __device__ bool RayBoxIntersection(
    float3 ray_pos, float3 inv_ray_dir, float3 boxMin, float3 boxMax, float& ↩
        tmin)
{
    float lo = inv_ray_dir.x*(boxMin.x - ray_pos.x);
    float hi = inv_ray_dir.x*(boxMax.x - ray_pos.x);

    float tmax;
    tmin = fminf(lo, hi);
    tmax = fmaxf(lo, hi);

    float lo1 = inv_ray_dir.y*(boxMin.y - ray_pos.y);
    float hi1 = inv_ray_dir.y*(boxMax.y - ray_pos.y);
```

```
    tmin = fmaxf(tmin, fminf(lo1, hi1));
    tmax = fminf(tmax, fmaxf(lo1, hi1));

    float lo2 = inv_ray_dir.z*(boxMin.z - ray_pos.z);
    float hi2 = inv_ray_dir.z*(boxMax.z - ray_pos.z);

    tmin = fmaxf(tmin, fminf(lo2, hi2));
    tmax = fminf(tmax, fmaxf(lo2, hi2));

    return (tmin <= tmax) && (tmax > 0.f);
}
```

Другой пример – упрощение тела цикла поиска в kd-дереве:

```
while(!node.Leaf())
{
    int axis = node.GetAxis();

    float rdir;
    rdir = (axis==0) ? ray_dir.x : ray_dir.y;
    rdir = (axis==2) ? ray_dir.z : rdir;

    float rpos;
    rpos = (axis==0) ? ray_pos.x : ray_pos.y;
    rpos = (axis==2) ? ray_pos.z : rpos;

    int nearNodeOffset = (rdir >= 0) ? node.GetLeftOffset() : node.↩
        GetLeftOffset()+1;
    int farNodeOffset  = (rdir >= 0) ? node.GetLeftOffset()+1 : node.↩
        GetLeftOffset();

    float t_split = (node.GetSplitPos() - rpos)/rdir;

    bool traverseNearNode = (t_near  <= t_split);
    bool traverseFarNode  = (t_split <= t_far);

    node = getKdTreeNode(traverseNearNode ? nearNodeOffset : farNodeOffset);

    if(traverseNearNode && traverseFarNode)
    {
      stack[top].offset = farNodeOffset;
      stack[top].t_far  = t_far;
      top++;
      t_far = t_split;
    }
}
```

Минимизация нагрузки на текстурные блоки. Большим количеством зависимых выборок в цикле поиска пересечений трассировка лучей создает значительную нагрузку на текстурные блоки. Поскольку архитектура GPU рассчитана в большей степени на высокую долю вычислений, то невыгоден любой значительный поток данных, пусть даже почти всегда кэшируемых. В частности, по этой причине поиск пересечений с использованием BVH-деревьев может быть более эффективен, нежели поиск пересечений с использованием kd-деревьев.

Использование константной памяти. Константную память следует использовать с некоторой осторожностью, поскольку она рассчитана на то, чтобы все нити варпа обращались строго по одному адресу. Если разные нити варпа обращаются по разным адресам, то происходит сериализация, и чтения из константной памяти выполняются последовательно.

Структуры данных без косвенного доступа. Большое число обращений к данным через указатели внутри структур (a->b->c->d->e) нарушает независимость текстурных выборок, повышая латентность доступа к памяти. Поэтому данные желательно группировать в простые структуры.

16-битное представление вещественных данных. Одним из способов снизить нагрузку на подсистему памяти является использование так называемой «половинной точности». 16-битной точности, как правило, достаточно для представления нормалей, тангент-, битангент-векторов, отношений различных величин и, в некоторых случаях, цветов. В CUDA отсутствует встроенная поддержка соответствующего типа *half*, но упаковка и распаковка значений доступна посредством функций *__half2float(ushort)* и *__float2half_rn(float)*.

Список ссылок

[1] Примеры программ к книге «Параллельные вычисления на GPU: Архитектура и программная модель CUDA». URL: http://parallel-compute.ru/book.

[2] Инфраструктура компилятора Open64. URL: http://www.open64.net/.

[3] NVOPENCC – вариант компилятора Open64 с поддержкой CUDA. URL: ftp://download.nvidia.com/CUDAOpen64/.

[4] Nath Rajib, Tomov Stanimire, Dongarra Jack. An Improved MAGMA GEMM for Fermi GPUs: LAPACK Working Note: 227. inst-UT-CS:adr: Department of Computer Science, University of Tennessee, Knoxville, 2010. UT-CS-10-655. URL: http://www.netlib.org/lapack/lawnspdf/lawn227.pdf.

[5] Беклимишев Д.В. Курс аналитической геометрии и линейной алгебры. Москва: ФИЗМАТЛИТ, 2006.

[6] Беклимишев Д.В. Дополнительные главы линецной алгебры. Москва: Наука, 1983.

[7] Perron Oskar. Grundlagen für eine Theorie des Jacobischen Kettenbruchalgorithmus // Mathematische Annalen. 1907. T. 64. C. 1–76.

[8] Parallel Versions of FFTW. URL: http://www.fftw.org/parallel/parallel-fftw.html.

[9] Implementing FFTs in Practice. URL: http://cnx.org/content/m16336/latest.

[10] Marsaglia George. Xorshift RNGs // Journal of Statistical Software. 2003. T. 8, № 14. C. 1–6. See [38] for corrections and the equivalence of xorshift generators and the well-understood linear feedback shift register generators. URL: http://www.jstatsoft.org/v08/i14; http://www.jstatsoft.org/v08/i14/xorshift.pdf.

[11] Соболь И. М. О распределении точек в кубе и приближенном вычислении интегралов // Ж. вычисл. матем. и матем. физ. 1967. Т. 7, № 4. С. 784–802. URL: http://mi.mathnet.ru/zvmmf7334.

[12] Hoberock Jared, Bell Nathan. Thrust: A Parallel Template Library. 2010. Version 1.3.0. URL: http://www.meganewtons.com/.

[13] Bell Nathan. Rapid Problem Solving Using Thrust. URL: http://www.gputechconf.com/object/gtc-express-webinar.htm.

[14] PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation: Tech. Rep.: 2009-40 / Andreas Klöckner, Nicolas Pinto, Yunsup Lee [и др.]. Providence, RI, USA: Scientific Computing Group, Brown University, 2009. URL: http://arxiv.org/abs/0911.3456.

[15] PyCUDA and PyOpenCL. URL: http://mathema.tician.de/software/pycuda.

[16] Python tutorial. URL: http://docs.python.org/tutorial/.

[17] PyCUDA Installation Guides. URL: http://wiki.tiker.net/PyCuda/Installation.

[18] Флетчер К. Вычислительные методы в динамике жидкостей. Том 2. Москва: Мир, 1991.

[19] Sagaut Pierre. Large eddy simulation for incompressible flows. Third изд. Springer, 2006.

[20] Пасконов В. М., Березин С. Б. Неклассические решения классической задачи о течении вязкой несжимаемой жидкости в плоском канале // Прикладная математика и информатика. Москва, 2004. Т. 17.

[21] Пасконов В. М., Березин С. Б., Корухова Е. С. Динамическая система визуализации для многопроцессорных систем с общей памятью и ее применение для численного моделирования турбулентных течений вязких жидкостей // Вестник Московского университета. 2007. Т. Серия 15: Вычислительная математика и кибернетика. С. 7–16.

[22] Sakharnykh Nikolai, Berezin Sergey, Paskonov Vilen M. [и др.]. Fluid solver based on Navier-Stokes equations using finite difference methods in areas with dynamic boundaries. URL: http://code.google.com/p/cmc-fluid-solver/.

[23] Morinishi Y., Lund T. S., Vasilyev O. V. et al. Fully Conservative Higher Order Finite Difference Schemes for Incompressible Flow. 1998. URL: http://citeseer.ist.psu.edu/381728.html; http://landau.mae.missouri.edu/ vasilyev/Publications/JCP1998b.pdf.

[24] Su S.-W., Lai M.-C., Lin C.-A. An immersed boundary technique for simulating complex flows with rigid boundary // Computers and Fluids. 2007. Т. 36. С. 313–324.

[25] Mittal R., Iaccarino G. Immersed boundary methods // Annual Review of Fluid Mechanics. 2005. Т. 37. С. 239–261.

[26] Saad Yousef. Iterative Methods for Sparse Linear Systems. PWS publishing, 1996.

[27] van der Vorst Henk A. Iterative Krylov methods for large linear systems. Cambridge, UK: Cambridge University Press, 2003. Т. 13 из *Cambridge Monographs on Applied and Computational Mathematics*. С. xiv + 221.

[28] Ortega James M. Introduction to Parallel and Vector Solution of Linear Systems. Plenum Press, 1988.

[29] Harrar David L., Ortega James M. Optimum m-step SSOR preconditioning // J. of Computational and Applied Mathematics. 1988. Т. 24. С. 195–198.

[30] Benzi M., Meyer C. D., Tuma M. A sparse approximate inverse preconditioner for the conjugate gradient method // SIAM J. on Scientific Computing. 1996. Т. 17(5). С. 1135–1149.

[31] A parallel preconditioned conjugate gradient solver for the poisson problem on a multi-GPU platform / M. Ament, G. Knittel, D. Weiskopf [и др.] // Proceedings of the 18th Euromicro Conference on Parallel Disturbed and Networkbased Processing. 2010. С. 583–592.

[32] Demmel James W., Heath Michael T., van der Vorst Henk A. Parallel numerical linear algebra // Acta Numerica. Cambridge, 1993. С. 111–197.

[33] Ольшанский М.А. Лекции и упражнения по многосеточным методам. Москва: Издательство Московского государственного университета, 2003.

[34] Mortikov Evgeny. Navier-Stokes Equations solver with GPU support. URL: http://tesla.parallel.ru/trac/nse.

[35] Pharr Matt, Humphreys Greg. Physically-Based Rendering: From Theory to Implementation. Morgan Kaufmann Publishers, 2004.

[36] Jensen Henrik Wann. A Practical Guide to Global Illumination Using Photon Maps // SIGGRAPH 2000 Course Notes CD-ROM / Association for Computing Machinery. ACM SIGGRAPH, 2000. Course 8.

[37] Wald Ingo, Havran Vlastimil. On building fast kd-trees for ray tracing, and on doing that in O(N log N) // Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing. 2006. С. 61–69.

[38] Brent Richard P. Note on Marsaglia's Xorshift Random Number Generators // Journal of Statistical Software. 2004. Т. 11, № 5. С. 1–5. See [10],[39]. This article shows the equivalence of xorshift generators and the well-understood linear feedback shift register generators. URL: http://www.jstatsoft.org/counter.php?id=101&url=v11/i05/v11i05.pdf&ct=1.

[39] Panneton François, L'Ecuyer Pierre. On the xorshift random number generators // ACM Transactions on Modeling and Computer Simulation. 2005. Т. 15, № 4. С. 346–361. See [10],[38].

Приложение А

Установка и настройка CUDA

Реализация NVIDIA CUDA состоит из трех отдельных пакетов:

- “developer”-драйвер GPU;
- CUDA Toolkit – компилятор, профилировщик, отладчик (для Linux, для Windows – NVIDIA Parallel Nsight поставляется отдельно), системые и прикладные библиотеки;
- CUDA SDK – коллекция примеров приложений, использующих CUDA.

Пакеты распространяются отдельно для различных операционных систем, драйвер и CUDA Toolkit – в бинарном виде, CUDA SDK – в виде исходных файлов. Загрузка всех частей производится со страницы `http://developer.nvidia.com/cuda-downloads`, рис. А.1 (при установке для Linux необходимо учитывать конкретную версию дистрибутива и замечения в разделе А.2).

Помимо “developer”-драйвера также существует «обычный» драйвер для GPU NVIDIA, который можно загрузить со страницы драйверов `http://www.nvidia.com/drivers`. Различия между этими двумя типами драйверов минимальны: «обычный» драйвер содержит самые новые обновления и оптимизации, тогда как “developer”-драйвер имеет несколько более раннюю «стабильную» версию. Для работы CUDA-приложений, как правило, не важно, используется ли «обычный» или “developer”-драйвер. Существенное же значение имеет то, реализован ли в установленной версии CUDA весь необходимый функционал, т.е. соответствует ли она той версии, для которой написано приложение.

Ниже приведены инструкции для типовой установки CUDA для ОС Windows и Linux.

А.1. Windows 7

Известной проблемой CUDA для Windows являются ошибки, вызванные наличием кириллических символов в путях проекта Microsoft Visual Studio или компо-

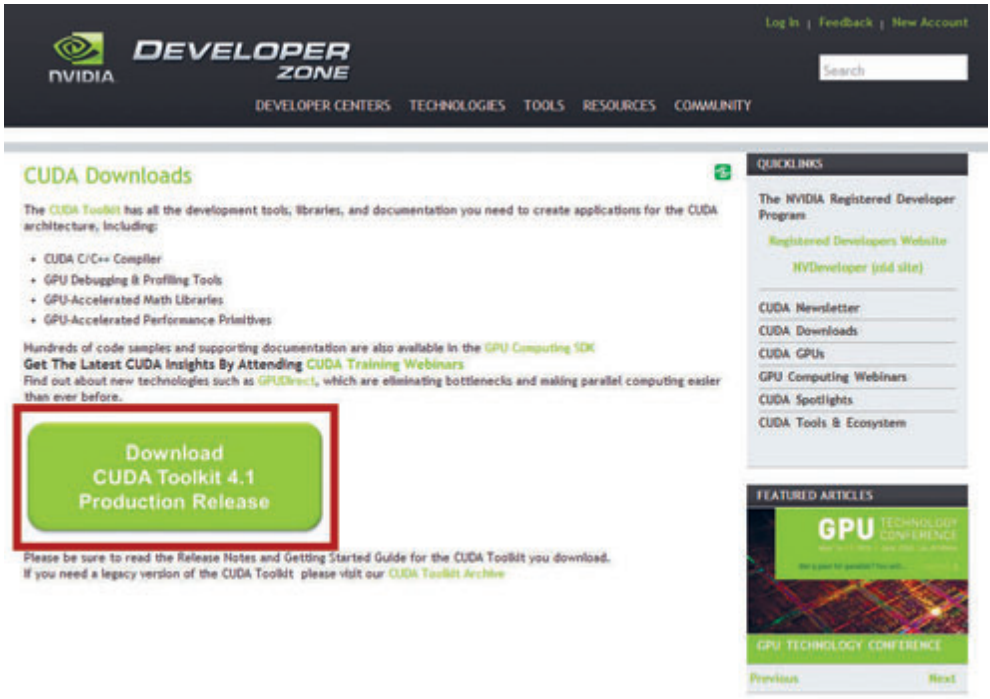


Рис. А.1. Страница установки компонент CUDA

нентов CUDA. Они могут быть, в частности, связаны с использованием кириллического имени пользователя. Перед установкой рекомендуется убедиться в отсутствии условий для данной проблемы.

Прежде всего, выполним загрузку и установку “developer”-драйвера CUDA (рис. А.2). При установке необходимо подтвердить установку драйвера, который «не может быть проверен Windows» (рис. А.3).

После установки драйвера в контекстном меню Рабочего Стола появится элемент “NVIDIA Control Panel” (рис. А.4). По ссылке “System Information” в нижнем углу можно открыть окно с информацией об установленных в системе видеокартах.

Разработка для CUDA под Windows возможна только при наличии компилятора и библиотек из состава Microsoft Visual Studio, кроме того, периодически появляется поддержка взаимодействия с компилятором Intel. Технически, существует

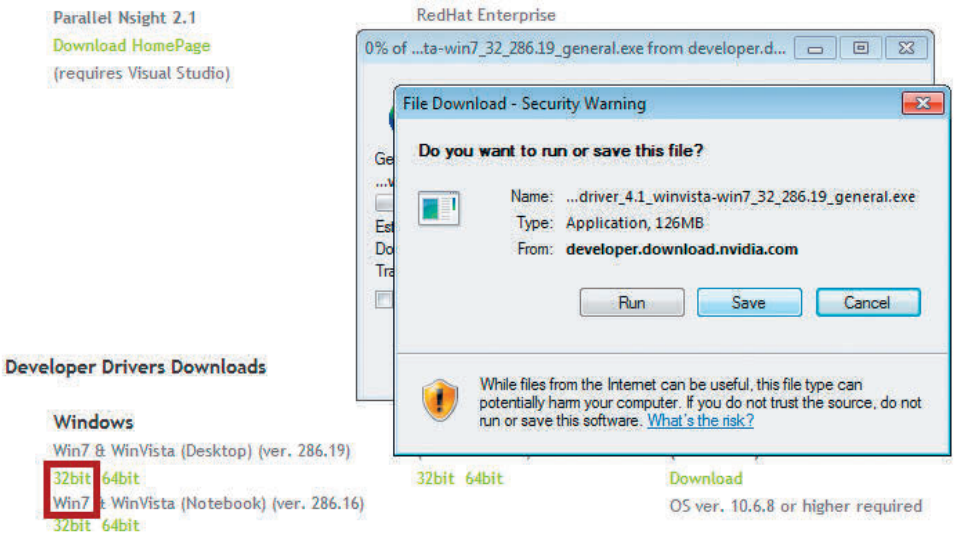


Рис. А.2. Загрузка драйвера CUDA

Рис. А.3. Установка драйвера CUDA

возможность сборки CUDA для Windows в бесплатных окружениях Cygwin или mingw, но официально такие сборки никогда не выпускались. По адресу `http://www.microsoft.com/visualstudio/en-us/try` можно загрузить полнофункциональную версию Microsoft Visual Studio для ознакомительного использования в течение 90 дней (рис. А.5). Поскольку большинство современных браузеров поддерживают загрузку больших файлов и возобновление загрузки с точки останова, установочный файл можно загрузить напрямую (рис. А.6). Microsoft Visual Studio распространяется в виде образа DVD-диска, что не всегда удобно. Чтобы им воспользоваться, данный образ придется либо записать на DVD- или USB-носитель специальной программой, либо монтировать с помощью виртуального привода, например, Daemon Tools Lite, `http://www.daemon-tools.cc/eng/downloads` (рис. А.7). Для монтирования необходимо открыть программу через контекстное меню и выбрать ISO-образ. В случае успеха, содержимое этого образа будет доступно в появившемся логическом DVD-приводе. Для работы CUDA подходит стандартная установка Microsoft Visual Studio (рис. А.9).

Основные компоненты CUDA объединены в пакет CUDA Toolkit (рис. А.10). Для интеграции с Microsoft Visual Studio также понадобятся некоторые компоненты из состава CUDA SDK (рис. А.11).

Для первого CUDA-приложения создадим в Microsoft Visual Studio новый проект типа “Win32 Console Application” (рис. А.12). Из автоматически созданного дерева проекта можно удалить все файлы и через свойства проекта выключить использование Precompiled Headers (рис. А.18), взамен добавив *sum_kernel.cu* с соответствующим примером на CUDA C из данной книги (рис. А.13). Первой строчкой в *sum_kernel.cu* дополнительно следует добавить “#define BLOCK_SIZE 512”. Затем в меню “Build Customization” свойств проекта необходимо выбрать “CUDA Build Customization 4.1” (рис. А.14). В свойствах самого исходного файла *sum_kernel.cu* должен быть установлен тип “CUDA C/C++” (рис. А.15). В разделе *Linker/Input* свойств проекта необходимо добавить подключение библиотеки CUDA Runtime – *$(CUDA_LIB_PATH)/cudart.lib* (рис. А.16). После того как данные шаги выполнены, CUDA-приложение должно успешно собираться (комбинацией клавиш “Ctrl B”) и запускаться (клавишей “F5”). Для удобства можно вопользоваться консолью Microsoft Visual Studio (рис. А.17–А.19).

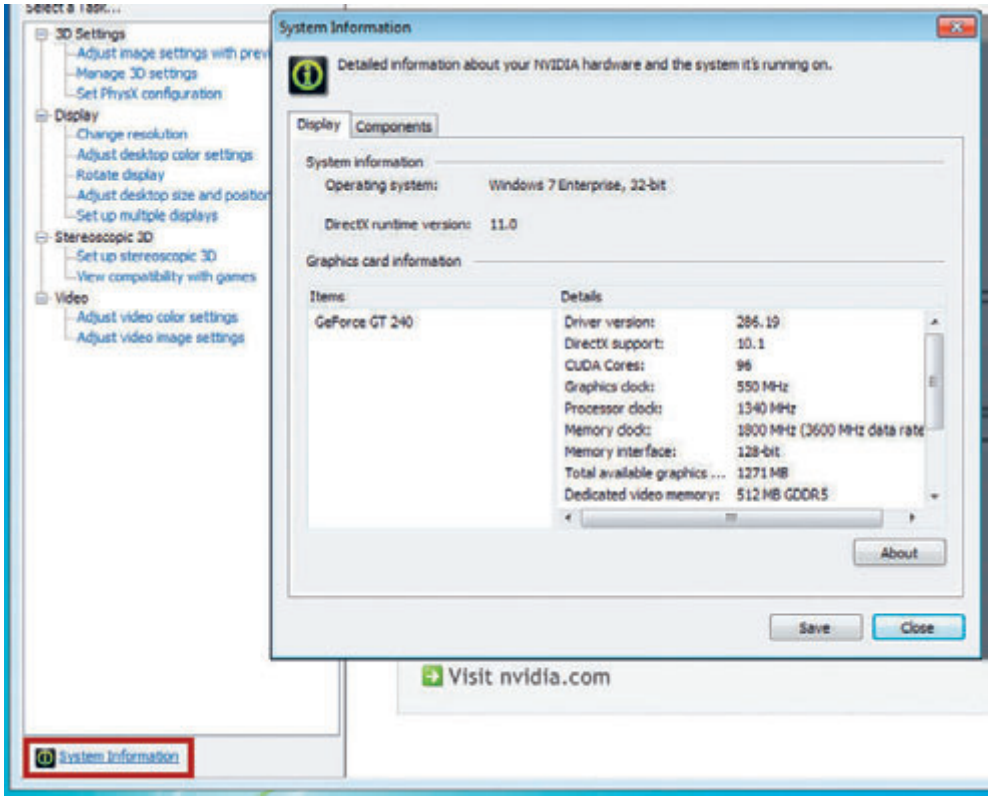


Рис. А.4. Предупреждение безопасности при установке драйвера CUDA

Рис. А.5. Загрузка Microsoft Visual Studio

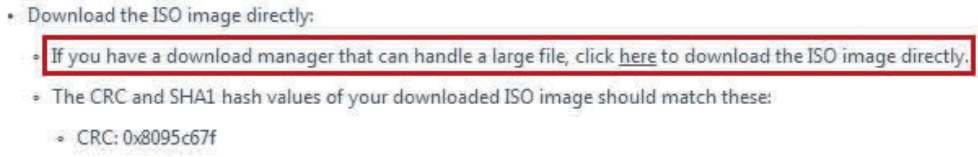


Рис. А.6. Загрузка iso-образа с Microsoft Visual Studio

Рис. А.7. Загрузка виртуального привода Daemon Tools Lite

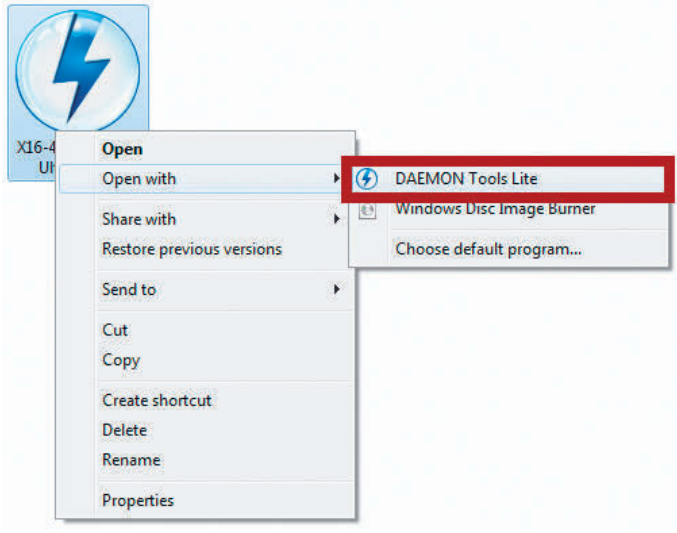


Рис. А.8. Доступ к Daemon Tools Lite через контекстное меню

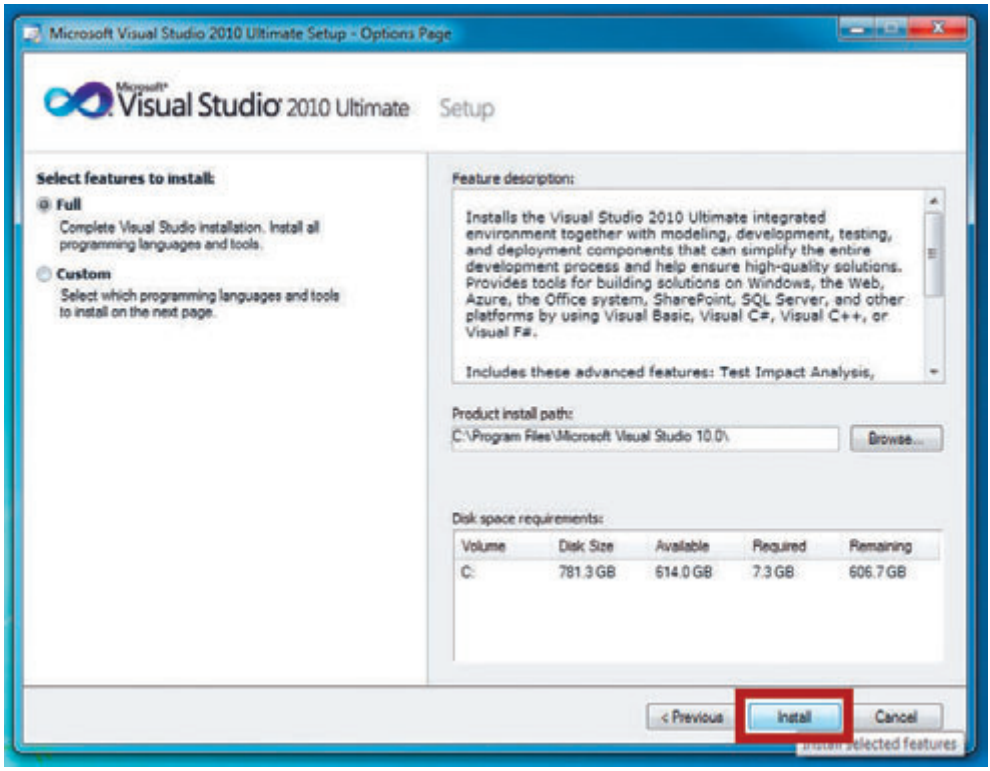


Рис. А.9. Установка Microsoft Visual Studio

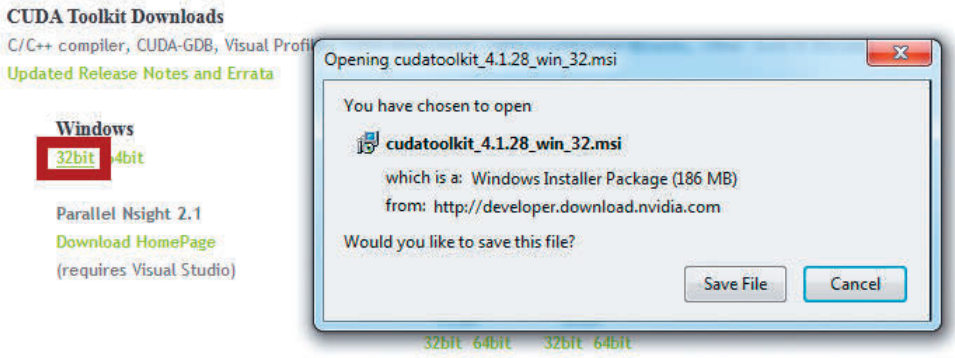


Рис. А.10. Установка CUDA Toolkit

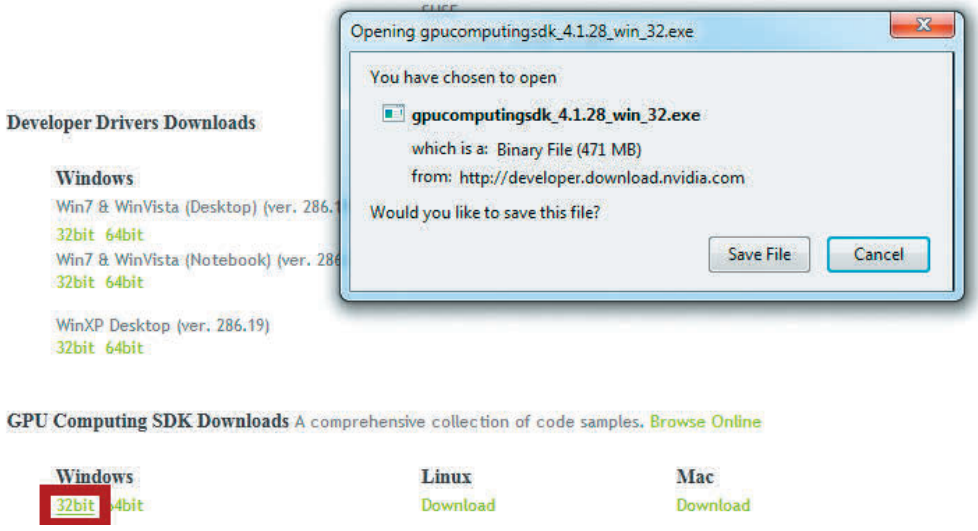


Рис. А.11. Установка CUDA SDK

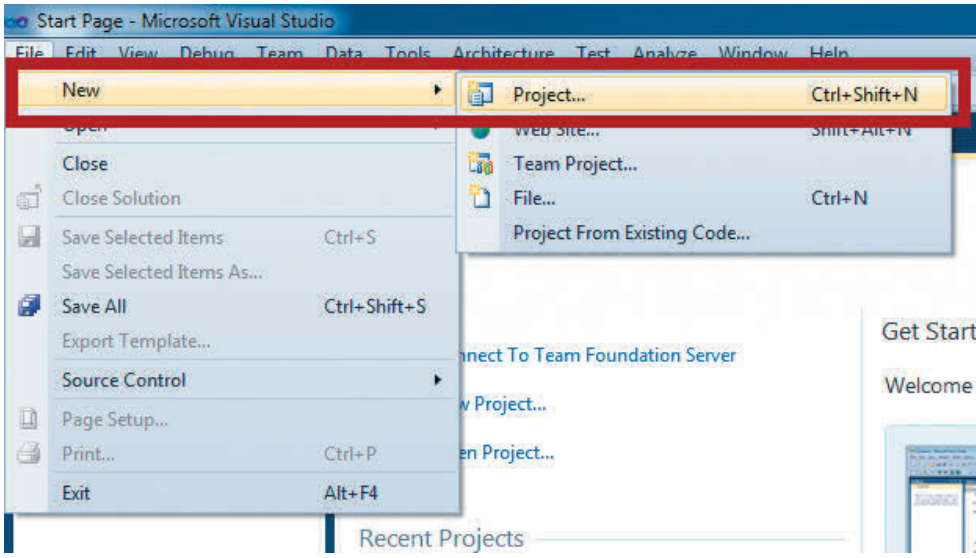


Рис. А.12. Создание нового проекта в Microsoft Visual Studio

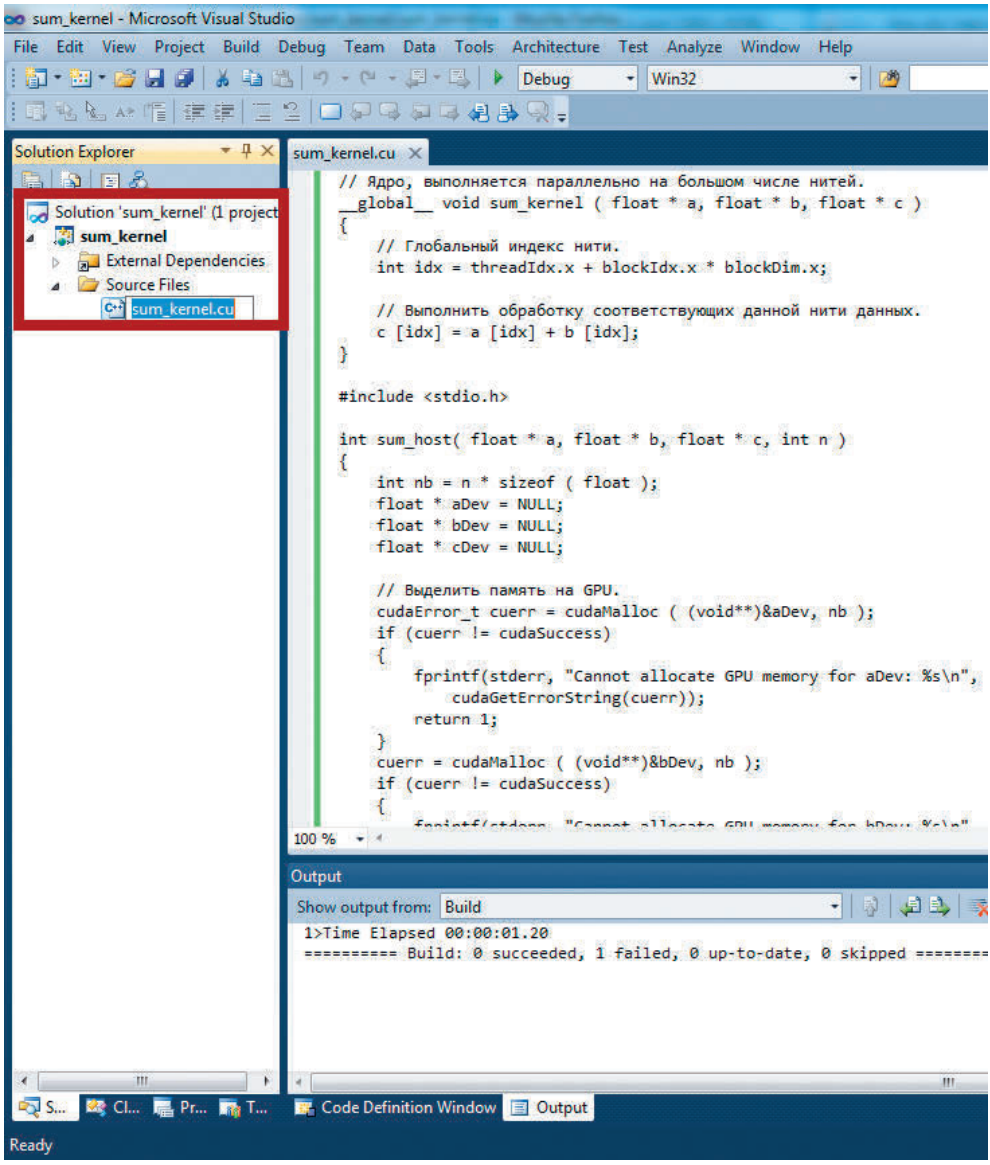


Рис. А.13. Содержимое проекта для CUDA-приложения

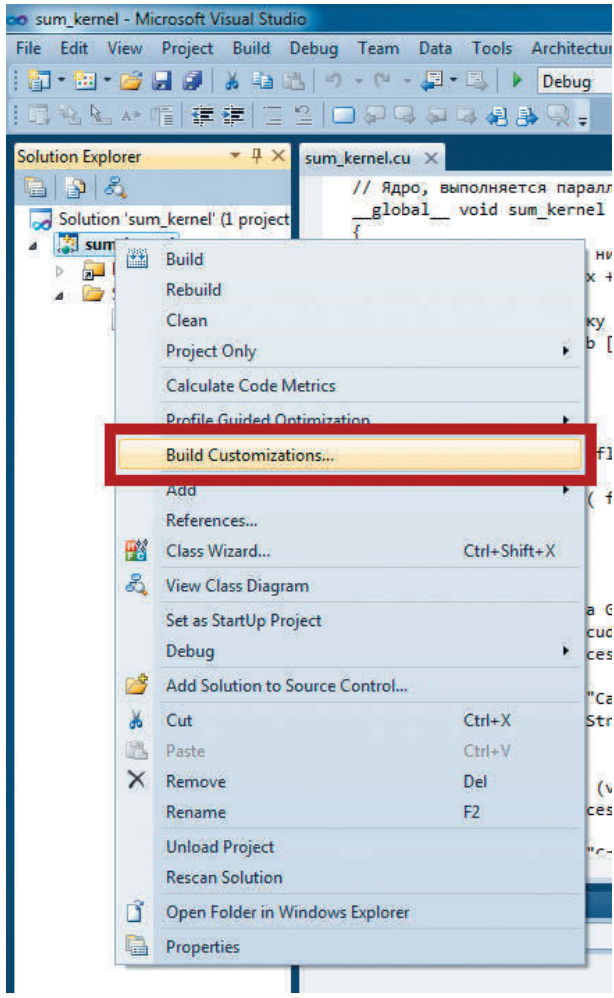


Рис. А.14. Настройка CUDA Build Customization

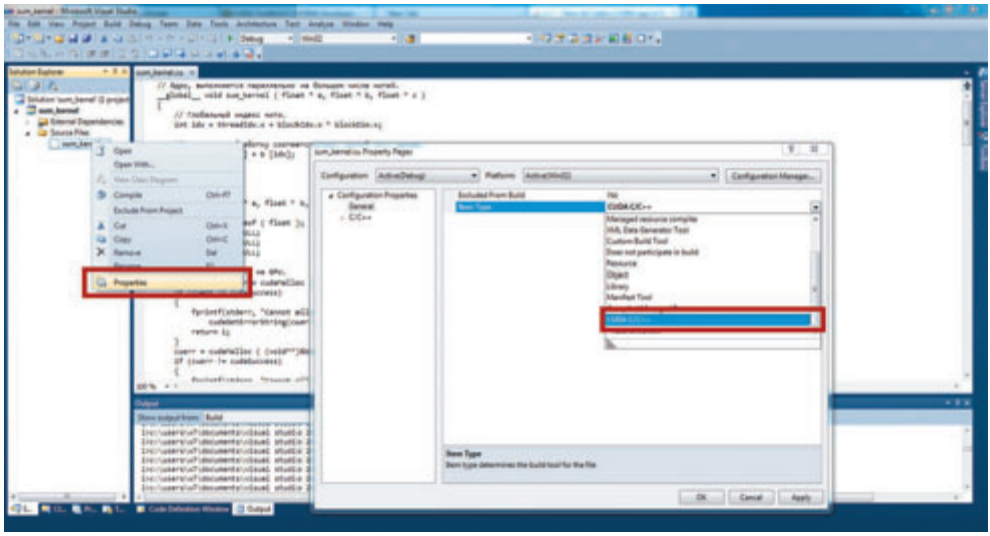


Рис. А.15. Настройка правила компиляции исходного файла CUDA

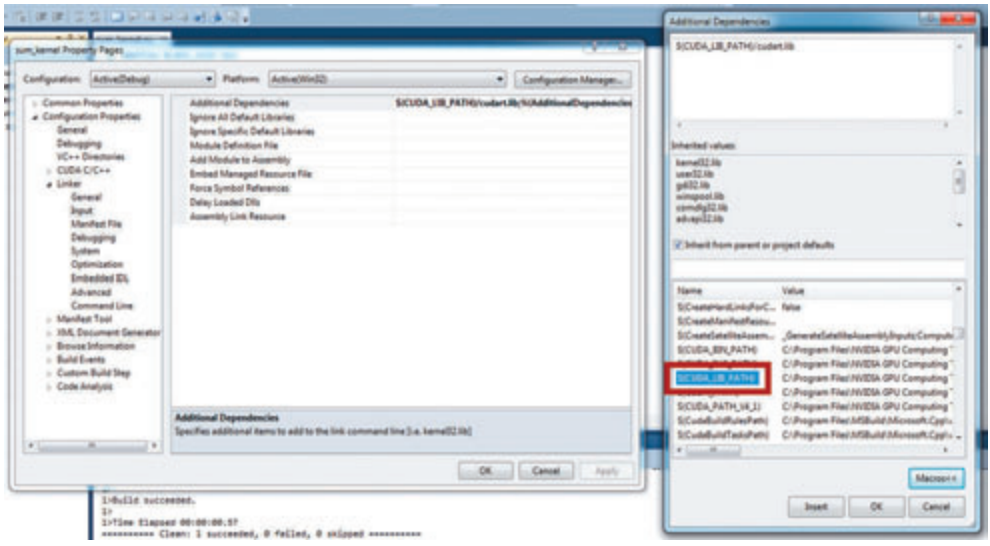


Рис. А.16. Настройка опций линковщика

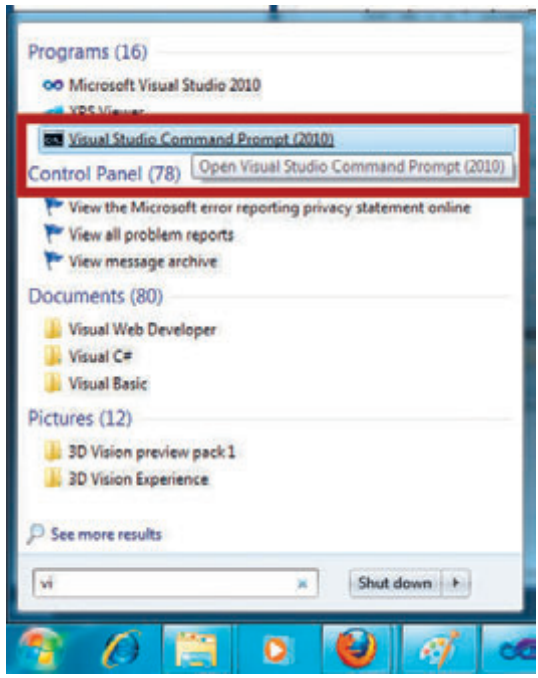


Рис. A.17. Запуск консоли Microsoft Visual Studio

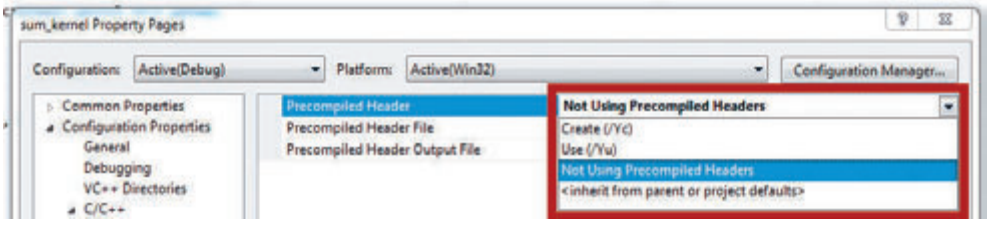


Рис. A.18. Выключить Precompiled Headers

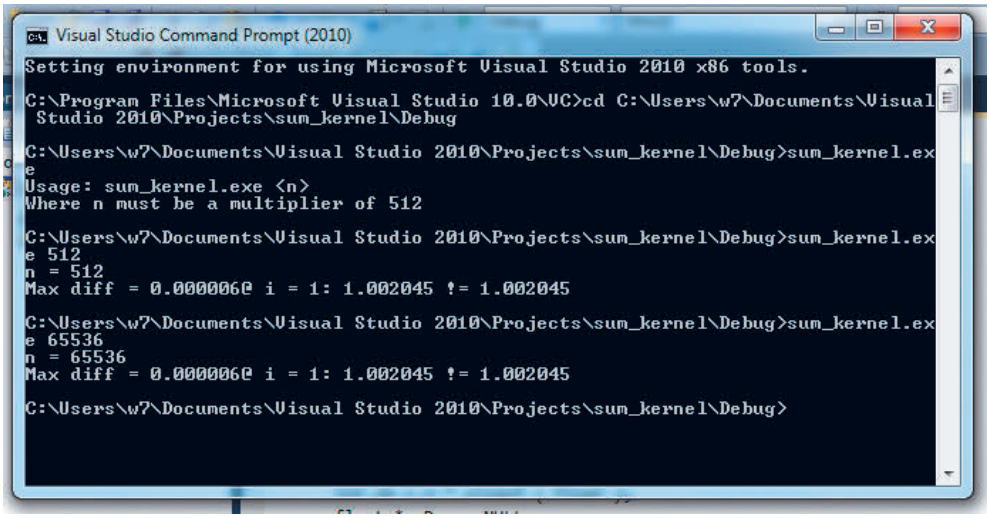


Рис. А.19. Выполнение CUDA-приложения в консоли Microsoft Visual Studio

Исходные файлы CUDA пока не имеют подсветки синтаксиса в редакторе кода. Подсветку можно подключить, скопировав файл *usertype.dat* из каталога NVIDIA SDK в каталог установки Microsoft Visual Studio, как это показано на рис. А.20. Для получения прав записи, копирование производится с помощью административной консоли (рис. А.21). Далее необходимо активировать подсветку через верхнее меню Visual Studio → Options, Text Editor → File Extension, как показано на рис. А.22. Тогда CUDA-код должен начать отображаться с подсветкой (рис. А.23).

А.1.1. PGI Visual Fortran

Компилятор PGI Visual Fortran выпускается с поддержкой CUDA-расширений, описанных в главе 4. После создания учетной записи на странице `http://www.pgroup.com/support/trial.htm` или входа в существующую учетную запись, в разделе *Download* `https://www.pgroup.com/support/downloads.php` можно загрузить PGI Visual Fortran для Microsoft Visual Studio (рис. А.24). После его установки среди типов новых проектов в Visual Studio появится PGI Visual Fortran (рис. А.25). В данном случае в новый проект добавлен исходный файл с расширением *.CUF*, содержащий код примера *sum_kernel* на языке CUDA

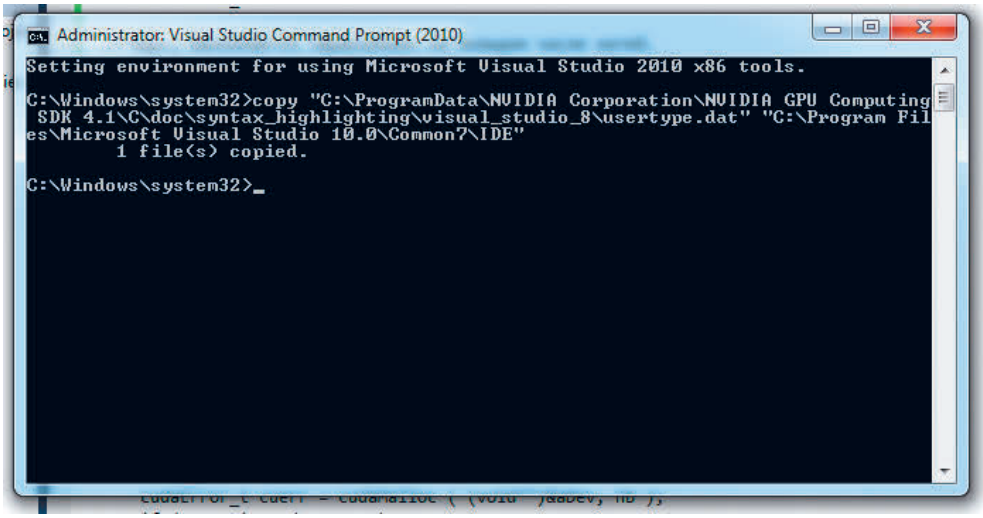


Рис. А.20. Копирование файла синтаксиса CUDA в Microsoft Visual Studio

Fortran (рис А.26). Необходимо активировать CUDA Fortran через свойства проекта → Fortran → Language → Enable CUDA Fortran → Yes (рис. А.27) и в строке Fortran → Preprocessing → Preprocessor Definitions указать *BLOCK_SIZE=512* (рис. А.28). Далее проект может быть скомпилирован и запущен обычным образом.

А.2. Linux

Методы установки CUDA в Linux могут быть различными в зависимости от дистрибутива. Во многих случаях дистрибутивы уже включают автоматический установщик проприетарного драйвера NVIDIA. В системах типа Ubuntu 12.04 программу установки драйверов можно найти по фильтру “addi” в строке Dash home (рис. А.29). Следует запустить утилиту Applications → Additional Drivers и активировать “NVIDIA accelerated graphics driver (version current)” или “NVIDIA accelerated graphics driver (post-release updates)” (рис. А.30). После завершения установки – перезагрузиться. Важно заметить, что установка драйвера с сайта NVIDIA может нарушить работоспособность графической оболочки, поскольку в Ubuntu файлы драйвера размещены в нестандартной папке */usr/nvidia-current*. По этой причине открытым является вопрос о наличии простого и надежного способа установки

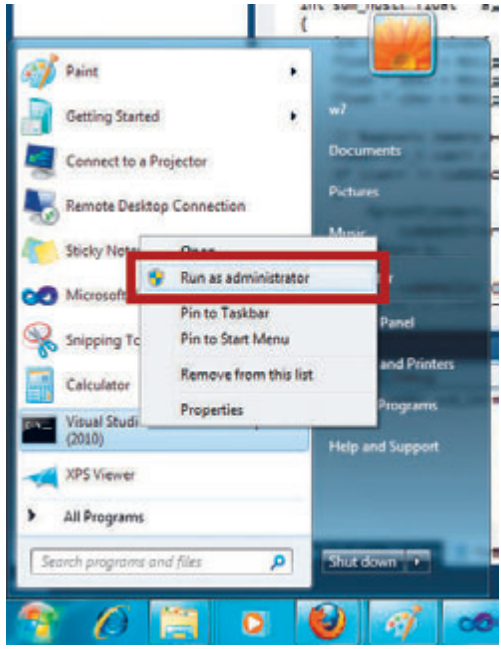


Рис. А.21. Запуск административной консоли Microsoft Visual Studio

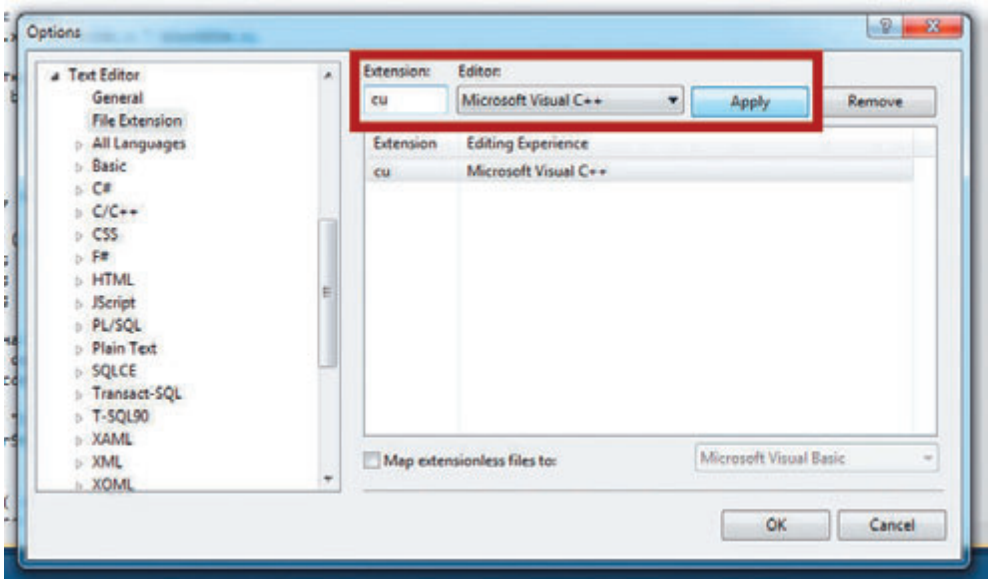


Рис. А.22. Активация подсветки синтаксиса CUDA

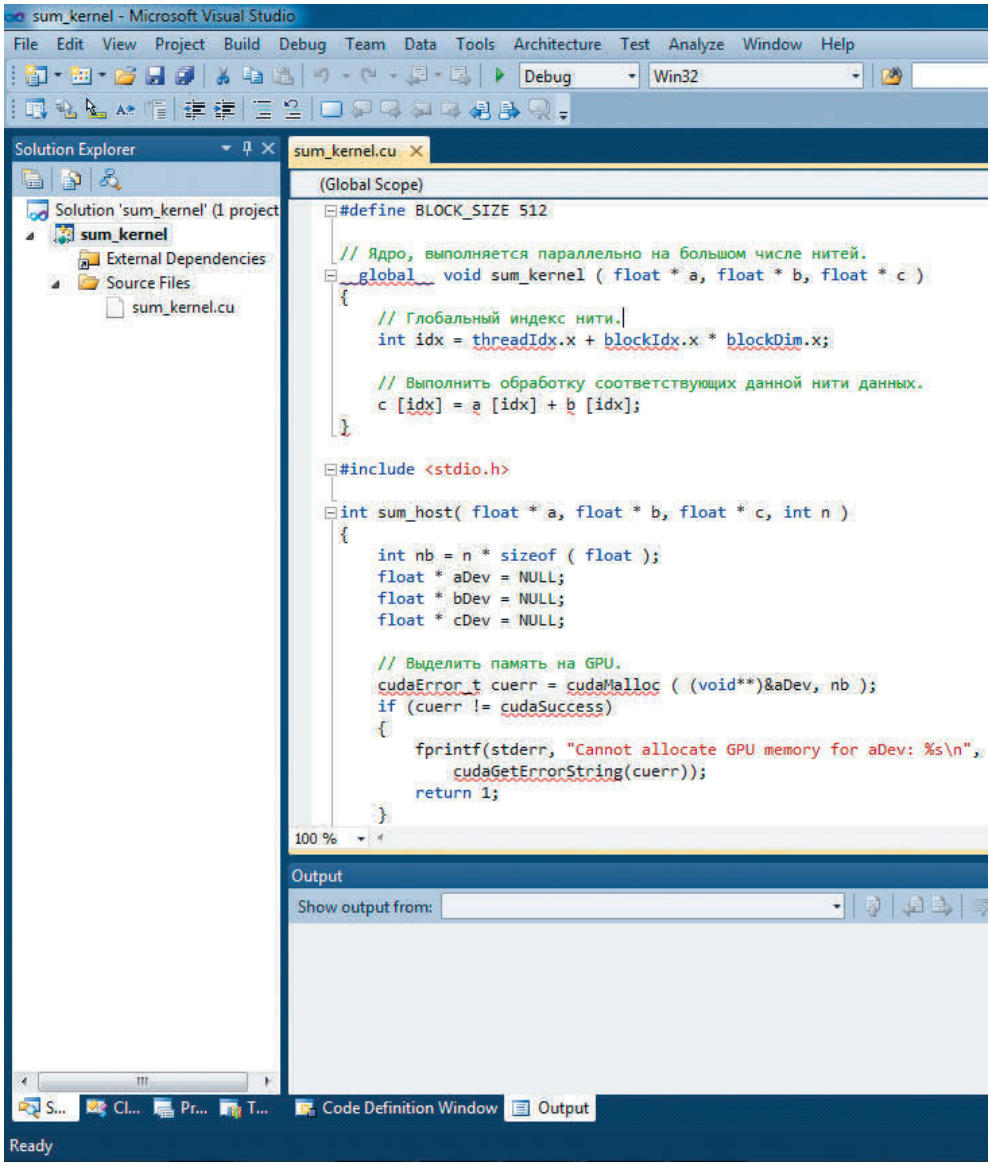


Рис. А.23. Редактор исходного кода CUDA с включенной подсветкой

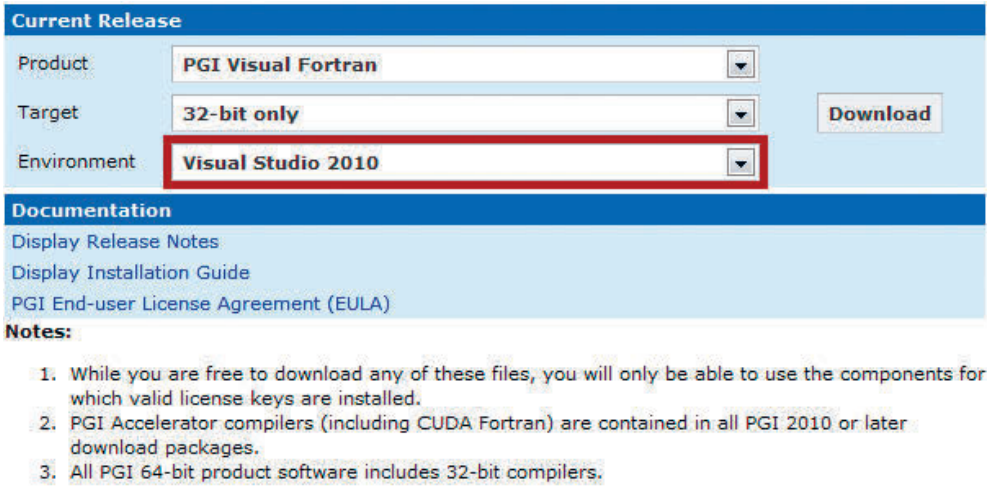


Рис. А.24. Загрузка дистрибутива PGI Visual Fortran

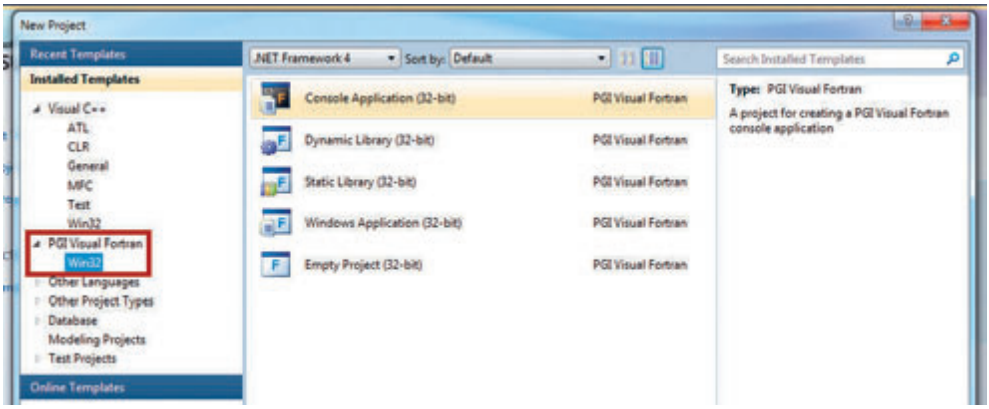


Рис. А.25. Новый проект PGI Visual Fortran в Microsoft Visual Studio

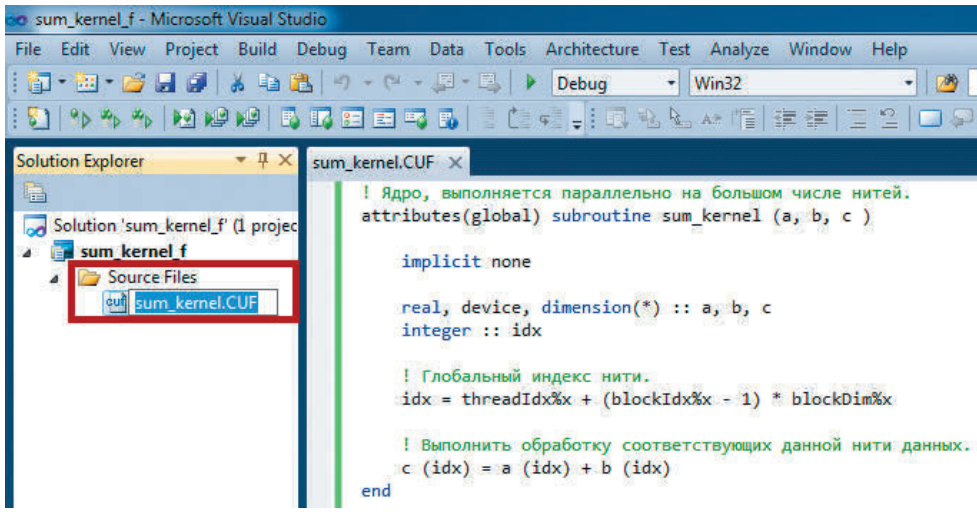


Рис. А.26. Добавление файла с расширением *.CUF*

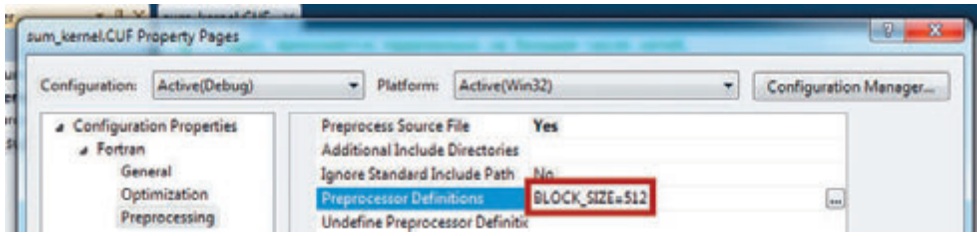


Рис. А.27. Подключение поддержки CUDA Fortran

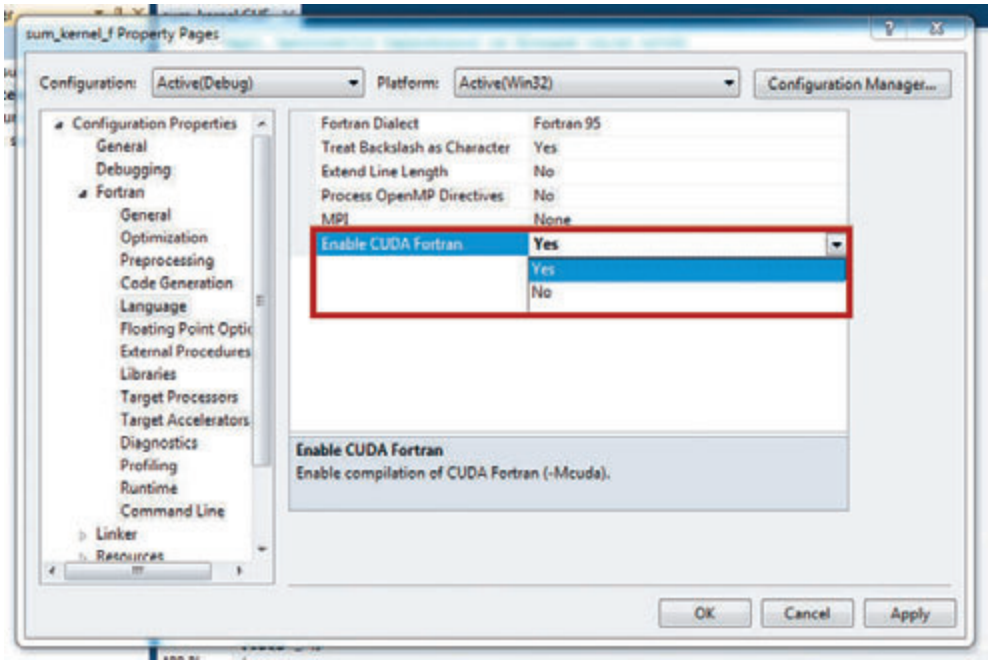


Рис. А.28. Определение символа препроцессора *BLOCK_SIZE*

Рис. А.29. Навигация в системном меню для поиска программы установки драйвера

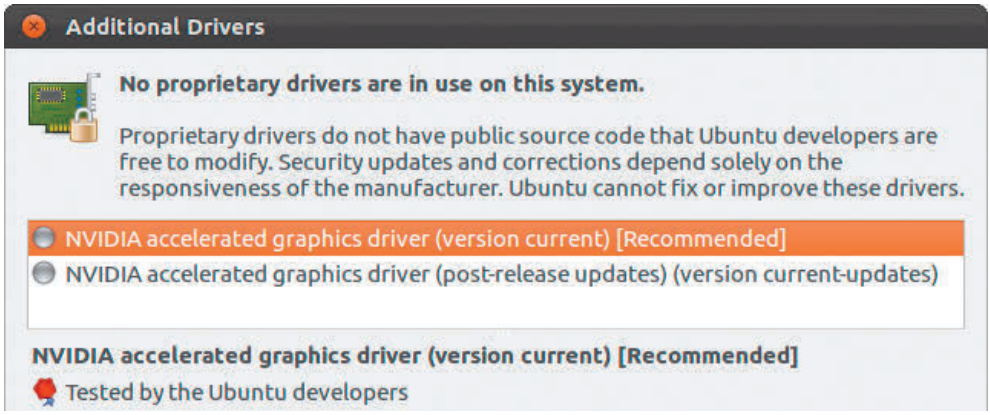


Рис. А.30. Запуск утилиты Additional Drivers

версии драйвера, недоступной в репозиториях Ubuntu.

Если при следующем запуске системы драйвер был успешно загружен, то информацию о нем можно посмотреть через каталоги */proc*:

```
ubi@ubi:~$ cat /proc/driver/nvidia/version
NVRM version: NVIDIA UNIX x86 Kernel Module  295.20  Mon Feb  6 20:25:38 PST ↩
    2012
GCC version:  gcc version 4.6.2 (Ubuntu/Linaro 4.6.2-16ubuntu1)
```

К драйверу прилагается ряд утилит, например, *nvidia-smi*, сообщающая информацию о состоянии установленных в системе GPU:

```
ubi@ubi:~$ nvidia-smi
Sat Mar 17 21:19:09 2012
+------------------------------------------------------+
| NVIDIA-SMI 3.295.20   Driver Version: 295.20         |
|-------------------------------+----------------------+----------------------+
| Nb.  Name                     | Bus Id        Disp.  | Volatile ECC SB / DB |
| Fan   Temp   Power Usage      | Memory Usage         | GPU Util. Compute M. |
|===============================+======================+======================|
| 0.  GeForce GT 240            | 0000:01:00.0  N/A    |        N/A        N/A |
|  35%   38 C  N/A   N/A        | 33%  171MB /  511MB  |  N/A      Default    |
|-------------------------------+----------------------+----------------------|
| Compute processes:                                               GPU Memory |
|  GPU  PID     Process name                                       Usage      |
|=============================================================================|
|  0.           Not Supported                                                 |
+-----------------------------------------------------------------------------+
```

В отличие от драйвера NVIDIA с поддержкой CUDA, CUDA Toolkit в дистрибутивы Linux обычно не входит. Его можно скачать с сайта NVIDIA и установить в заданную папку:

```
$ wget http://developer.download.nvidia.com/compute/cuda/4_1/rel/toolkit/↩
    cudatoolkit_4.1.28_linux_32_ubuntu11.04.run
$ sudo sh ./cudatoolkit_4.1.28_linux_32_ubuntu11.04.run
Enter install path (default /usr/local/cuda, '/cuda' will be appended): /opt
```

Чтобы компилятор CUDA был доступен по имени файла из командной строки пользователя, добавьте в папку */etc/profile.d/* файл *cuda.sh* следующего содержания:

```
export PATH=$PATH:/opt/cuda/bin
```

Аналогично, можно добавить в папку */etc/ld.so.conf.d/* файл *cuda.conf* с путями поиска динамических библиотек, требуемых при запуске CUDA-приложений:

```
/opt/cuda/lib
/opt/cuda/lib64
```

В Ubuntu данные изменения будут иметь эффект только после перезагрузки.

Работа компилятора CUDA требует наличия системных компиляторов gcc и g++ для CPU:

```
$ sudo apt—get install g++
```

В CUDA Toolkit официально поддерживаются версии gcc/g++ только вполть до конкретной версии, для этого установлено следующее ограничение в заголовочном файле */opt/cuda/inclide/host_config.h*:

```
#error — unsupported GNU version! gcc 4.6 and up are not supported!
```

Если системные компиляторы gcc/g++ новее, чем 4.6, то произойдет ошибка компиляции. Избежать ее можно, используя компиляторы более старых версий, или же закомментировав саму строку с *#error*. В последнем случае код для CUDA, не использующий C++, с высокой вероятностью будет успешно компилироваться и корректно работать.

С сайта NVIDIA также можно установить CUDA SDK с набором примеров:

```
$ wget http://developer.download.nvidia.com/compute/cuda/4_1/rel/sdk/↩
    gpucomputingsdk_4.1.28_linux.run
$ sh ./gpucomputingsdk_4.1.28_linux.run
Enter CUDA install path (default /usr/local/cuda): /opt/cuda_sdk
```

А.3. Использование визуальной среды Eclipse совместно с CUDA

Визуальная среда Eclipse может служить аналогом Microsoft Visual Studio в Linux и входит в поставку Ubuntu:

```
$ sudo apt-get install eclipse
$ sudo apt-get install eclipse-cdt
```

К сожалению, нередко поставляемые в составе дистрибутивов версии Eclipse оказываются нестабильными. В этом случае заменой может служить бинарный пакет с сайта программы: `http://www.eclipse.org/downloads/`.

При загрузке Eclipse потребует выбрать рабочее пространство, для начала это может быть */home/username/workspace*. Далее необходимо установить Eclipse Marketplace, если он не установлен. В меню Help → Install New Software... → Work with: выбрать "Indigo Update Site" и в поле "type filter text" набрать "Marketplace". После установки Marketplace потребуется перезапуск Eclipse.

В Eclipse Marketplace по фильтру "CUDA" можно найти и установить "CUDA Plugin (Helios)". Далее можно создать новый проект: New → Project → C/C++ → C++ project → Next → Project Name → sum_kernel, Location → */home/ubi/sum_kernel*, Project Type → Makefile Project → Empty Project → Next → Finish.

Папка */home/ubi/sum_kernel* должна содержать исходные файлы и makefile следующего содержания (отступы должны быть сделаны одной табуляцией):

```
all: sum_kernel

sum_kernel: sum_kernel.cu
    nvcc -DBLOCK_SIZE=512 $< -o $@

clean:
    rm -rf sum_kernel
```

Вид визуального редактора Eclipse может меняться в зависимости от типа проекта. Вручную тип можно выбрать через меню Window → Open Perspective → C/C++.

Завершающий шаг – настройка ранее созданного проекта для работы с CUDA. В меню sum_kernel → Properties → C/C++ General → Paths and Symbols → Languages → NVIDIA CUDA Language → Include Directories необходимо заменить "/usr/local/cuda/include" на "/opt/cuda/include", в меню Toolchains выбрать

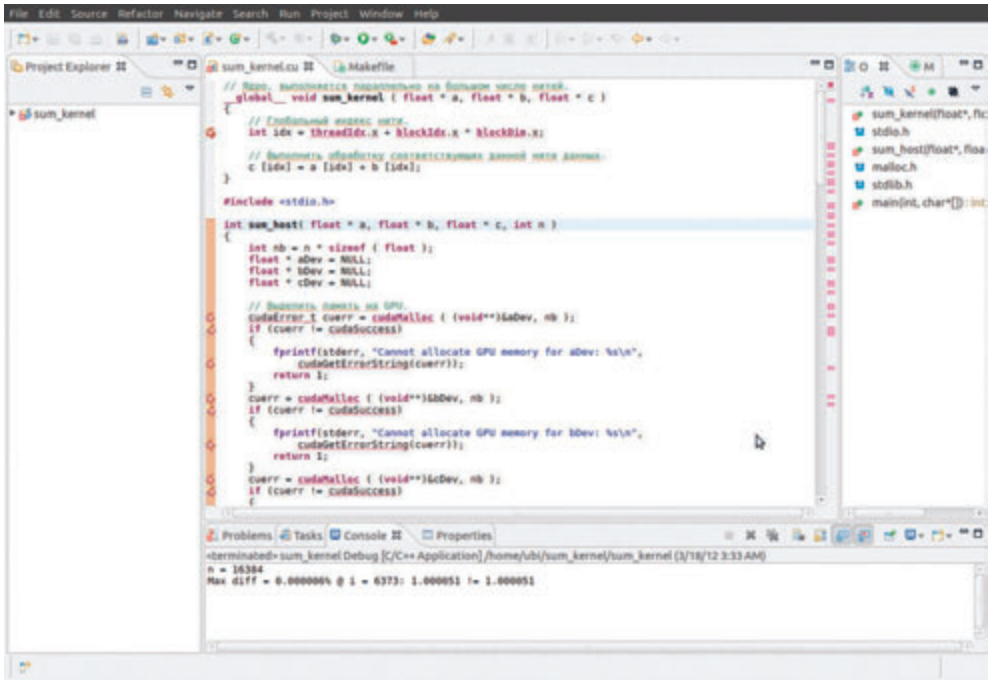


Рис. А.31. Разработка CUDA-приложения в визуальной среде Eclipse CDT

“CUDA Toolchain” и нажать “Next”.

Сборка проекта производится сочетанием клавиш “Ctrl+B”. Для быстрого запуска можно установить аргументы программы через меню Run → Run Configurations ... → C/C++ Applications → sum_kernel Debug → Arguments → Program Arguments → 16384 → Close.

Наконец, запуск программы: Run → Run As... → Local C/C++ Application или сочетание клавиш “Ctrl+F11” (рис. А.31).

Приложение Б

Счетчики профилирования

Обозначения: SM – мультипроцессор, TPC – текстурный блок вместе с использующими его мультипроцессорами (Texture Processing Cluster).

Таблица Б.1. Счетчики *domain_a* для устройств с compute capability 1.x

Имя и описание счетчика	Тип	Capability			
		1.0	1.1	1.2	1.3
tex_cache_hit – число попаданий в текстурный кэш	SM	Y	Y	Y	Y
tex_cache_miss – число промахов в текстурном кэше	SM	Y	Y	Y	Y

Таблица Б.2. Счетчики *domain_b* для устройств с compute capability 1.x

Имя и описание счетчика	Тип	Capability			
		1.0	1.1	1.2	1.3
branch – количество ветвей условных операторов, пройденных нитями. Счетчик увеличивается на единицу, если хотя бы одна нить выполняет данную ветвь. Барьерные синхронизации (__syncthreads()) так же считаются ветвями	SM	Y	Y	Y	Y
divergent_branch – количество дивергентных ветвлений – случаев, когда хотя бы одна нить варпа выполняет ветвь условного оператора, отличную от ветвей, выполняемых всеми другими нитями	SM	Y	Y	Y	Y
instructions – количество выполненных инструкций	SM	Y	Y	Y	Y

warp_serialize – если два запроса в константную или разделяемую память попадают в один и тот же банк памяти, то происходит банк-конфликт, и запросы сериализуются (выполняются последовательно). Этот счетчик хранит число варпов, в которых произошла сериализация доступа	SM	Y	Y	Y	Y
gld_incoherent – количество необъединенных (non-coalesced) чтений из глобальной памяти	TPC	Y	Y	N	N
gld_coherent – количество объединенных (coalesced) чтений из глобальной памяти	TPC	Y	Y	N	N
gld_32b – количество 32-байтовых чтений из глобальной памяти	TPC	N	N	Y	Y
gld_64b – количество 64-байтовых чтений из глобальной памяти	TPC	N	N	Y	Y
gld_128b – количество 128-байтовых чтений из глобальной памяти	TPC	N	N	Y	Y
gst_incoherent – количество необъединенных (non-coalesced) записей в глобальную память	TPC	Y	Y	N	N
gst_coherent – количество объединенных (coalesced) записей в глобальную память	TPC	Y	Y	N	N
gst_32b – количество 32-байтовых записей в глобальную память	TPC	N	N	Y	Y
gst_64b – количество 64-байтовых записей в глобальную память	TPC	N	N	Y	Y
gst_128b – количество 128-байтовых записей в глобальную память	TPC	N	N	Y	Y
local_load – количество чтений произвольного размера из локальной памяти	TPC	Y	Y	Y	Y
local_store – количество записей в локальную память: +2 на каждую 32-байтовую транзакцию, +4 – на 64-байтовую и +8 – на 128-байтовую	TPC	Y	Y	Y	Y
cta_launched – количество блоков нитей, запущенных на TPC	TPC	Y	Y	Y	Y
sm_cta_launched – количество блоков нитей, запущенных на SM	SM	Y	Y	Y	Y

prof_trigger_XX – 8 счетчиков для событий (00-07), контролируемых пользователем вызовами prof_trigger(int)	SM	Y	Y	Y	Y

Таблица Б.3. Счетчики *domain_a* для устройств с compute capability 2.x

Имя и описание счетчика	**Тип**	**Capability**	
		2.0	2.1
branch – количество ветвей условных операторов, пройденных нитями. Счетчик увеличивается на единицу, если хотя бы одна нить выполняет данную ветвь	SM	Y	Y
divergent_branch – количество дивергентных ветвлений – случаев, когда хотя бы одна нить варпа выполняет ветвь условного оператора, отличную от ветвей, выполняемых всеми другими нитями	SM	Y	Y
warps_launched – количество запущенных варпов	SM	Y	Y
threads_launched – количество запущенных нитей	SM	Y	Y
active_warps – аккумулированное количество варпов за такт. На каждом такте счетчик увеличивается на число активных варпов, которое может быть от 0 до 48	SM	Y	Y
active_cycles – количество тактов, в течение которых мультипроцессор исполнял хотя бы один варп	SM	Y	Y
sm_cta_launched – количество запущенных блоков нитей	SM	Y	Y
local_load – количество инструкций чтения из локальной памяти на варп	SM	Y	Y
local_store – количество инструкций записи в локальную память на варп	SM	Y	Y

gld_request – количество инструкций чтения из глобальной памяти на варп	SM	Y	Y
shared_load – количество инструкций чтения из разделяемой памяти на варп	SM	Y	Y
shared_store – количество инструкций записи в разделяемую память на варп	SM	Y	Y
l1_local_load_hit – количество попаданий в L1-кэш при чтении из локальной памяти. Счетчик увеличивается на 1, 2 или 4 соответственно для 32, 64 или 128-битного доступа	SM	Y	Y
l1_local_load_miss – количество промахов мимо L1-кэша при чтении из локальной памяти. Счетчик увеличивается на 1, 2 или 4 соответственно для 32, 64 или 128-битного доступа	SM	Y	Y
l1_local_store_hit – количество попаданий в L1-кэш при записи в локальную память. Счетчик увеличивается на 1, 2 или 4 соответственно для 32, 64 или 128-битного доступа	SM	Y	Y
l1_local_store_miss – количество промахов мимо L1-кэша при записи в локальную память. Счетчик увеличивается на 1, 2 или 4 соответственно для 32, 64 или 128-битного доступа	SM	Y	Y
l1_global_load_hit – количество попаданий в L1-кэш при чтении из глобальной памяти. Счетчик увеличивается на 1, 2 или 4 соответственно для 32, 64 или 128-битного доступа	SM	Y	Y
l1_global_load_miss – количество промахов мимо L1-кэша при чтении из глобальной памяти. Счетчик увеличивается на 1, 2 или 4 соответственно для 32, 64 или 128-битного доступа	SM	Y	Y
uncached_global_load_transaction – количество некэшируемых транзакций чтения из глобальной памяти. Счетчик увеличивается на 1, 2 или 4 соответственно для 32, 64 или 128-битного доступа	SM	Y	Y

global_store_transaction – количество транзакций записи в глобальную память. Счетчик увеличивается на 1, 2 или 4 соответственно для 32, 64 или 128-битного доступа	SM	Y	Y
l1_shared_bank_conflict – количество конфликтов банков, вызванных попаданием в один и тот же банк двух или более обращений в разделяемую память	SM	Y	Y
prof_trigger_XX – всего существует 8 триггеров общего назначения (00-07), которые пользователь может профилировать. Эти триггеры могут быть вставлены в любую часть кода для сбора требуемой статистики	SM	Y	Y
inst_executed – количество исполненных инструкций, не включая повторы	SM	Y	Y

Lomonosov Moscow State University

A. V. Boreskov, A.A. Kharlamov, N.D. Markovskiy,
D. N. Mikushin, E. V. Mortikov, A. A. Myltsev,
N. A. Sakharnykh, V. A. Frolov

Parallel Computing on GPU

Architecture and CUDA Programming Model

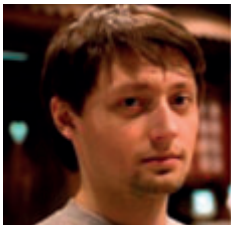
Manual

This book is a detailed practical guide on developing applications with NVIDIA CUDA technology, version 4. The first part of the book is dedicated to the CUDA programming model basics in context of C and Fortran languages and the GPU memory hierarchy. Efficient strategies of shared memory utilization are illustrated in typical computational algorithms. The second part of the book covers GPU-enabled mathematical libraries and frameworks. Another part of the book describes tools for professional CUDA applications analysis, debugging and diagnostics. Approaches of combining CUDA with cluster programming models are presented. The last part consists of several articles on advanced CUDA user experience in computational fluid dynamics and computer graphics. The book is intended for engineers and researchers who use parallel computations.

Key words: CUDA, GPU, clustering, debugging, Navier – Stockes, raytracing, multi-GPU.

Боресков Алексей Викторович
факультет вычислительной математики и кибернетики
Московского государственного университета имени
М. В. Ломоносова, к.ф.-м.н., мл. научный сотрудник

Alexey Boreskov, PhD
Dept. of Computational mathematics and cybernetics
Lomonosov Moscow State University,
junior scientific fellow



Харламов Александр Александрович
NVIDIA, инженер
факультет вычислительной математики и кибернетики
Московского государственного университета имени
М. В. Ломоносова, аспирант

Alexander Kharlamov
NVIDIA, engineer
Dept. of Computational mathematics and cybernetics
Lomonosov Moscow State University, PhD Student

Марковский Николай Дмитриевич, PhD
NVIDIA, инженер по технологиям в HPC

Nikolay Markovskiy, PhD
NVIDIA, HPC Devtech

Микушин Дмитрий Николаевич
Факультет Информатики Университета Лугано,
аспирант
Applied Parallel Computing LLC, Tech Lead

Dmitry Mikushin
Universita della Svizzera italiana,
Facolta di scienze informatiche, PhD student

Мортиков Евгений Валерьевич
Научно-исследовательский вычислительный центр МГУ имени М. В. Ломоносова, мл. научный сотрудник
Институт Океанологии РАН им. П. П. Ширшова, аспирант

Evgeny Mortikov
Moscow State University Research Computing Center, junior scientific fellow; Shirshov Institute of Oceanology RAS, PhD student

Мыльцев Александр Анатольевич
Институт прикладной математики РАН им. М. В. Келдыша, аспирант

Alexander Myltsev
Keldysh Institute of Applied Mathematics RAS, PhD student
http://pat.keldysh.ru/~myltsev/

Сахарных Николай Александрович
NVIDIA, инженер
факультет вычислительной математики и кибернетики Московского государственного университета имени М. В. Ломоносова, аспирант

Nikolai Sakharnykh
NVIDIA, engineer
Dept. of Computational mathematics and cybernetics Lomonosov Moscow State University, PhD student

Фролов Владимир Александрович
NVIDIA, инженер
Институт прикладной математики им. М. В. Келдыша РАН, аспирант

Vladimir Frolov
NVIDIA, engineer
Keldysh Institute of Applied Mathematics RAS, PhD student
http://ray-tracing.ru

CONTENTS

Серия
Суперкомпьютерное
Образование

Воеводин В. В.
Вычислительная математика и структура алгоритмов: 10 лекций о том, почему трудно решать задачи на вычислительных системах параллельной архитектуры и что надо знать дополнительно, чтобы успешно преодолевать эти трудности: Учебное пособие. — 2-е издание, стереотипное. — М.: Издательство Московского университета, 2010. — 168 с. — (Серия «Суперкомпьютерное образование»).

ISBN 978-5-211-05933-7

В книге представлены лекции, прочитанные автором в различных учебных заведениях, институтах и на научных конференциях. Все они посвящены вопросам эффективного решения задач на вычислительных системах параллельной архитектуры. Особое внимание уделяется изучению информационной структуры алгоритмов и ее влиянию на разработку эффективно реализуемых программ. Обсуждаются особенности математического образования по отношению к требованиям параллельных вычислений.

Для студентов, аспирантов и научных работников, специализирующихся в области исследования структуры алгоритмов, решения больших задач и создания программного обеспечения для параллельных вычислительных систем.

Ключевые слова: Вычислительная математика, структура алгоритмов, информационная структура программ и алгоритмов, параллельные вычисления, суперкомпьютеры, отображение программ и алгоритмов на вычислительные системы, математическое образование, суперкомпьютерное образование.

Серия
Суперкомпьютерное
Образование

Гергель В. П.
Высокопроизводительные вычисления для многопроцессорных многоядерных систем: Учебник – М.: Издательство Московского университета, 2010. – 544 с., илл. – (Серия «Суперкомпьютерное образование»)
ISBN 978-5-211-05937-5
ISBN 978-5-9221-1312-0

В работе излагается учебный материал, достаточный для успешного начала работ в области параллельного программирования. Для этого в пособии дается краткая характеристика принципов построения параллельных вычислительных систем, рассматриваются математические модели параллельных алгоритмов и программ для анализа эффективности параллельных вычислений, приводятся примеры конкретных параллельных методов для решения типовых задач вычислительной математики.

Учебник предназначен для широкого круга студентов, аспирантов и специалистов, желающих изучить и практически использовать параллельные компьютерные системы для решения вычислительно трудоемких задач.

Рекомендовано Советом учебно-методическим объединением классических университетов по прикладной математике и информатике.

Подготовка учебника была выполнена в рамках работ программы развития Нижегородского университета как Национального исследовательского университета.

Ключевые слова: Вычислительная математика, параллельные вычисления, многоядерные процессоры, высокопроизводительные вычислительные системы, технологии параллельного программирования, ускорение и эффективность распараллеливания, параллельные методы и программы, суперкомпьютерное образование.

Учебное издание

Боресков Алексей Викторович, Харламов Александр Александрович,
Марковский Николай Дмитриевич, Микушин Дмитрий Николаевич,
Мортиков Евгений Валерьевич, Мыльцев Александр Анатольевич,
Сахарных Николай Александрович, Фролов Владимир Александрович

**Параллельные вычисления на GPU:
Архитектура и программная модель CUDA**

2-е издание

Автор дизайна и иллюстраций *Т. А. Смирнова*
Редактор *М. С. Кутасова*
Оформление переплета
П. А. Брызгалов, Ю. Н. Симоненко
Верстка *Д. Н. Микушин*

Подписано в печать 24.03.2015.
Формат 70×100/16. Гарнитура Times.
Бумага офсетная № 1. Усл. печ. л. 27,09.
Тираж 168 экз. (1-й завод)
Изд. № 10452. Заказ № 0135-15

Издательство Московского университета
125009, Москва, ул. Б. Никитская 5.

Тел.: 629-50-91. *Факс:* 697-66-71
939-34-93 *(отдел реализации)*
E-mail: secretary-msu-press@yandex.ru

Сайт Издательства МГУ:
www.msu.ru/depts/MSUPubl2005

Интернет-магазин: http://msupublishing.ru

Адрес отдела реализации:
Москва, ул. Хохлова, 11 (Воробьевы горы, МГУ).
E-mail: izd-mgu@yandex.ru Тел.: (495) 939-34-93

Типография МГУ
119991, ГСП-1, Москва, Ленинские горы, д. 1, стр. 15.
Отпечатано: Публичное акционерное общество
«Т8 Издательские Технологии».
109316 Москва, Волгоградский проспект,
дом 42, корпус 5.