Программистам на Fortran'е, которые понимают,
что хорошо, а что плохо.

Оглавление

Благодарности **10**

Предисловие **11**

ЧАСТЬ I
Программирование на CUDA Fortran **13**

Глава 1. Введение **14**

1.1. Краткая история вычислений на GPU 14
1.2. Параллельные вычисления 16
1.3. Основные идеи 17
1.3.1. Первая программа на CUDA Fortran 18
1.3.2. Обобщение на большие массивы 22
1.3.3. Многомерные массивы 25
1.4. Определение возможностей и ограничений оборудования с поддержкой CUDA ... 27
1.5. Обработка ошибок 38
1.6. Компиляция программы на CUDA Fortran 39
1.6.1. Раздельная компиляция 43

Глава 2. Измерение производительности и метрики производительности ... **48**

2.1. Измерение времени выполнения ядра 48
2.1.1. Синхронизация хоста и устройства и таймеры CPU ... 49
2.1.2. Хронометраж с помощью событий CUDA 50
2.1.3. Командный профилировщик 51
2.1.4. Профилировщик nvprof 53
2.2. Ядра, производительность которых, ограничена вычислениями, пропускной способностью памяти и задержкой ... 54
2.3. Пропускная способность памяти 58
2.3.1. Теоретически максимальная пропускная способность . 58
2.3.2. Эффективная пропускная способность 60

Глава 3. Оптимизация.................................... 63
3.1. Передача данных между хостом и устройством................. 63
3.1.1. Зафиксированная область памяти 64
3.1.2. Объединение мелких операций передачи в один пакет 69
3.1.3. Асинхронная передача данных (дополнительная тема)..........72
3.2. Память устройства 83
3.2.1. Объявление данных в коде, выполняемом на устройстве 85
3.2.2. Объединенный доступ к глобальной памяти 85
3.2.3. Текстурная память..99
3.2.4. Локальная память.. 105
3.2.5. Константная память 108
3.3. Внутрикристалльная память.. 112
3.3.1. L1-кэш... 112
3.3.2. Регистры.. 113
3.3.3. Разделяемая память 115
3.4. Пример оптимизации работы с памятью: транспонирование матрицы 122
3.4.1. Недогрузка разделов (дополнительная тема)...................... 128
3.5. Конфигурация выполнения 133
3.5.1. Параллелизм на уровне потоков.. 133
3.5.2. Параллелизм на уровне команд... 137
3.6. Оптимизация команд 140
3.6.1. Встроенные функции устройства 141
3.6.2. Флаги компилятора... 142
3.6.3. Расходящиеся варпы 143
3.7. Директивы генерации ядра из цикла 144
3.7.1. Редукция в CUF-ядрах 147
3.7.2. Потоки CUDA в CUF-ядрах 147
3.7.3. Параллелизм на уровне команд в CUF-ядрах...................... 148

Глава 4. Программирование компьютера с несколькими GPU 150
4.1. Средства CUDA для работы с несколькими GPU 150
4.1.1. Связь между равноправными устройствами....................... 152
4.1.2. Прямая передача данных между равноправными устройствами... 157
4.1.3. Транспонирование матрицы с применением равноправного доступа.. 168
4.2. Программирование нескольких GPU с применением библиотеки MPI 177
4.2.1. Сопоставление устройств рангам MPI 179
4.2.2. Транспонирование матрицы с применением MPI 186
4.2.3. Транспонирование матрицы с применением MPI, поддерживающей GPU... 188

ЧАСТЬ II
Примеры задач 191

Глава 5. Метод Монте-Карло 192
5.1. Библиотека CURAND 193
5.2. Вычисление π с помощью CUF-ядер 198
5.2.1. Стандарт IEEE-754 (дополнительная тема) 202
5.3. Вычисление π с помощью ядер редукции 205
5.3.1. Редукция с атомарными блокировками (дополнительная тема) 211
5.4. Точность суммирования 213
5.5. Опционное ценообразование 220

Глава 6. Метод конечных разностей 229
6.1. Девятиточечный шаблон конечно-разностной схемы для вычисления первой производной 229
6.1.1. Повторное использование данных и разделяемая память 231
6.1.2. Ядро производной по *x* 232
6.1.3. Производные по *y* и *z* 237
6.1.4. Неравномерные сетки 242
6.2. Двумерное уравнение Лапласа 246

Глава 7. Приложения быстрого преобразования Фурье 254
7.1. Библиотека CUFFT 254
7.2. Спектральное дифференцирование 263
7.3. Свертка 267
7.4. Решение уравнения Пуассона 276

ЧАСТЬ III
Приложение 283

Приложение A. Технические характеристики Tesla 284

Приложение B. Управление системой и окружением 287
B.1. Переменные окружения 287
B.1.1. Общие переменные окружения 287
B.1.2. Командный профилировщик 288
B.1.3. JIT-компиляция 288

B.2. Интерфейс управления системой nvidia-smi 289
B.2.1. Включение и выключение режима ECC................................290
B.2.2. Режим вычислений 292
B.2.3. Инерционный режим...293

Приложение C. Вызов CUDA C из CUDA Fortran..... 295

C.1. Вызовы библиотеки, написанной на CUDA C 295
C.2. Вызов написанной пользователем функции на CUDA C... 298

Приложение D. Исходный код 300

D.1. Текстурная память 300
D.2. Транспонирование матрицы 304
D.3. Параллелизм на уровне потоков и команд 311
D.4. Программирование с использованием нескольких GPU 315
D.4.1. Транспонирование с применением равноправного доступа к памяти...316
D.4.2. Транспонирование с применением библиотеки MPI для передачи данных между хостами 322
D.4.3. Транспонирование с применением библиотеки MPI для передачи данных между устройствами 327
D.5. Программирование метода конечных разностей 332
D.6. Решение уравнения Пуассона спектральным методом ... 352

Литература... 357

Предметный указатель 359

БЛАГОДАРНОСТИ

Работа над этой книгой приносила нам радость и удовлетворение – прежде всего, благодаря общению с людьми, которые помогли сделать книгу такой, как вы ее видите. Нам помогало – прямо или косвенно – множество людей; рискуя позабыть кого-то, мы хотим выразить благодарность следующим лицам и коллективам.

Разумеется, никакой книги о CUDA Fortran не было бы без самого языка CUDA Fortran, поэтому мы благодарны компании The Portland Group (PGI) и особенно Бренту Либэку (Brent Leback) и Майклу Вулфу (Michael Wolfe) – за предмет, о котором мы могли писать. Работа с PGI над CUDA Fortran стала для нас великолепным жизненным опытом.

Авторы часто задумывались над тем, что расчеты, выполненные в их диссертациях, которые занимали много-много часов на тогдашних больших векторных ЭВМ, сейчас – при использовании графических процессоров (GPU) NVIDIA – завершились бы раньше, чем вы успели бы выпить чашечку кофе. Мы благодарны сотрудникам NVIDIA, осуществившим этот технологический прорыв. Мы также выражаем признательность членам команды, разрабатывающей CUDA, – как нынешним, так и работающим уже в других местах: Филиппу Куадра (Philip Cuadra), Марку Хэргроуву (Mark Hairgrove), Стивену Джонсу (Stephen Jones), Тиму Маррею (Tim Murray) и Джоэлю Шерпельцу (Joel Sherpelz) за ответы на наши многочисленные вопросы.

Значительная часть материала этой книги – результат совместных усилий над оптимизацией приложений. Мы хотим поблагодарить всех, кто участвовал в этой работе, в том числе Норберта Джаффа (Norbert Juffa), Патрика Легресли (Patrick Legresley), Паулюса Мицикевичуса (Paulius Micikevicius) и Эверетт Филлипс (Everett Phillips).

Многие рецензировали рукопись этой книги на разных этапах работы над ней, мы благодарны Роберто Гомперцу (Roberto Gomperts), Марку Хэррису (Mark Harris), Норберту Джаффа, Бренту Либэку и Эверетт Филлипс за их замечания и предложения.

Мы также признательны Яну Баку (Ian Buck), который предоставил нам возможность уделить время этому предприятию, и своим семьям, мирившимся с тем, что мы работали и дома тоже.

Наконец, мы хотели сказать спасибо всем нашим учителям. Благодаря им мы смогли написать эту книгу и надеемся, что она в свою очередь поможет другим.

ПРЕДИСЛОВИЕ

Эта книга предназначена научным работникам и инженерам, которые занимаются разработкой или сопровождением компьютерных моделей и приложений на языке Fortran и хотели бы воспользоваться параллельными вычислениями на графических процессорах (GPU) для ускорения своего кода. Мы хотели познакомить читателей с основами программирования GPU с помощью CUDA Fortran, а также привести несколько типичных примеров, показывающих, что разработка кода на CUDA Fortran – не самоцель.

Архитектура CUDA разработана компанией NVIDIA, для того чтобы GPU можно было применять для вычислений общего вида, не требуя от программиста знаний в области компьютерной графики. Существует несколько способов доступа к этой архитектуре, в том числе из программ на языке C/C++ – с помощью CUDA C – и на языке Fortran – с помощью компилятора CUDA Fortran, разработанного компанией The Portland Group (PGI). В этой книге рассматривается второй подход. Компилятор CUDA Fortran следует отличать от ускорителя PGI Accelerator и от интерфейса между архитектурой CUDA и OpenACC Fortran, который представляет собой подход к использованию GPU на основе директив. CUDA Fortran – это просто аналог CUDA C, только на Fortran'е.

Предполагается, что читатель знаком с такими концепциями Fortran 90, как модули, производные типы и операции с массивами. Для тех, кто работал с прежними версиями Fortran и хотел бы перейти на более современные, имеется несколько прекрасных книг (например, Metcalf, 2011). В книге используются некоторые возможности, появившиеся в версии Fortran 2003, но они подробно объясняются. Хотя знакомство с Fortran 90 необходимо, опыт параллельного программирования (с помощью GPU или иных средств) необязателен. Своей привлекательностью модель программирования CUDA отчасти обязана простотой, которая позволяет начинающим очень быстро получить работающую параллельную программу.

Нередко причиной обращения к CUDA Fortran является желание переписать имеющийся, иногда довольно объемный, код на Fortran'е, воспользовавшись GPU. Поскольку CUDA – гибридная модель про-

граммирования в том смысле, что позволяет использовать одновременно GPU и CPU, то код, исполняемый на CPU, можно переносить на GPU постепенно. Компилятор CUDA Fortran используют и те, кто переносит приложения на GPU, в основном применяя подход на основе директив, принятый в OpenACC, но хотели бы повысить производительность немногих критических участков кода, вручную написав код на CUDA Fortran. OpenACC и CUDA Fortran могут мирно сосуществовать в одной программе.

Книга состоит из двух основных частей. Первая часть – учебное пособие по программированию на CUDA Fortran: от основ написания кода до некоторых советов по оптимизации. Во вторую часть мы включили несколько примеров для демонстрации применения описанных в первой части принципов к решению реальных задач.

В книге используются компиляторы PGI версий 13.x, которые можно скачать с сайта `http://pgroup.com`. Хотя все примеры компилируются и запускаются в любой поддерживаемой операционной системе из различных сред разработки, в книге упоминается только компиляция из командной строки, например в Linux или Mac OS X.

Сопроводительный сайт

Дополнительные материалы можно скачать с сайта издательства Elsevier по адресу `http://store.elsevier.com/product.jsp?isbn=9780124169708`

Часть I

Программирование на CUDA Fortran

ГЛАВА 1
Введение

1.1. Краткая история вычислений на GPU

Параллельные вычисления в той или иной форме существуют уже десятки лет. Поначалу они были вотчиной немногих программистов, имевших доступ к большим и дорогим компьютерам. Но теперь все круто изменилось. Почти все потребительские настольные ПК и ноутбуки оснащены *центральными процессорами* (CPU) с несколькими ядрами. Даже сотовые телефоны и планшеты по большей части оборудуются многоядерными процессорами. Основная причина чуть ли не повсеместного присутствия нескольких процессорных ядер состоит в том, что производители уже не могут повышать производительность одноядерных CPU за счет увеличения тактовой частоты. Поэтому, начиная примерно с 2005 года, применяется «масштабирование по горизонтали», т. е. увеличение количества ядер, а не «масштабирование по вертикали» (повышение тактовой частоты). Но хотя сейчас производятся CPU, имеющие от двух до нескольких десятков ядер, такая степень параллелизма – ничто по сравнению с количеством ядер в *графическом процессоре* (GPU). Например, NVIDIA Tesla® K20X содержит 2688 ядер. С момента своего появления в середине 1990-х годов GPU отличались массивно-параллельной архитектурой, потому что обработка графики – задача, легко поддающаяся распараллеливанию.

На первых порах применение GPU для вычислений общего вида (эту технику часто называют GPGPU) было далеко не тривиальным занятием. Приходилось использовать API для программирования графики, а он накладывал серьезные ограничения на типы алгоритмов, допускающих реализацию на GPU. И даже в тех случаях, когда реализация была возможна, программирование алгоритма было трудным и интуитивно неочевидным делом для научных работников

и инженеров, не сталкивающихся по роду занятий с компьютерной графикой. Поэтому GPU входили в обиход научных и инженерных расчетов медленно.

Все изменилось, когда в 2007 году компания NVIDIA предложила архитектуру CUDA®. Эта архитектура включала как аппаратные компоненты в составе GPU производства NVIDIA, так и среду программирования, которая устраняла препятствия, стоящие на пути широкого внедрения GPGPU. С момента своего появления CUDA распространилась настолько широко, что в ноябре 2010 года три из первой пятерки суперкомпьютеров в списке Top 500 использовали GPU. И самый быстрый в мире суперкомпьютер по состоянию на ноябрь 2012 года также был оборудован GPU. Одна из причин столь быстрого признания CUDA – исключительная простота модели программирования. CUDA C, первый интерфейс к архитектуре CUDA, по существу представляет собой язык C с несколькими расширениями, позволяющими переместить части алгоритма на GPU. Таким образом, мы имеем гибридное решение с использованием одновременно CPU и GPU, поэтому переносить вычисления на GPU можно постепенно.

В конце 2009 года в результате совместной работы компании The Portland Group® (PGI®) и NVIDIA был создан компилятор CUDA Fortran. Если CUDA C – это C с расширениями, то CUDA Fortran – это по сути Fortran 90 с несколькими расширениями, позволяющими задействовать в вычислениях всю мощь графических процессоров. Об эффективной разработке приложений с помощью CUDA C написано немало книг, статей и других документов (см., например, Sanders and Kandrot, 2011; Kirk and Hwu, 2012; Wilt, 2013). Что же касается CUDA Fortran, то в связи с его новизной количество пособий, посвященных созданию кода, сравнительно невелико. Многие приемы написания эффективного кода на CUDA C легко переносятся на CUDA Fortran, поскольку базовая архитектура одна и та же. Тем не менее, ощущается необходимость в материалах, где описывалось бы, как эффективно программировать на CUDA Fortran. Тому есть две причины. Во-первых, хотя CUDA C и CUDA Fortran похожи, между ними есть различия, оказывающие влияние на способы написания кода. Это неудивительно – ведь и код для CPU, написанный на C и Fortran, с ростом проекта расходится в разные стороны. Кроме того, в CUDA C есть возможности, отсутствующие в CUDA Fortran, в частности, некоторые аспекты текстур. Наоборот, в CUDA Fortran имеются особенности, которых в CUDA C нет, например, атрибут переменной `device`, обозначающий, что данные находятся в памяти GPU.

Эта книга предназначена читателям, для которых параллельные вычисления – не самоцель, а инструмент решения задач. Наша задача – предложить читателю базовые навыки, необходимые для написания разумно оптимизированного кода на CUDA Fortran, в котором были бы задействованы вычислительные возможности оборудования NVIDIA®. Мы остановились на таком подходе и не пытались научить тому, как выжимать из оборудования максимальную производительность, поскольку считаем, что пользователи CUDA Fortran рассматривают этот язык всего лишь как средство. Как правило, такие пользователи ценят ясный и удобный в сопровождении код, который было бы легко писать и который работает с приемлемой производительностью на нескольких поколениях процессоров с поддержкой CUDA и с разными версиями CUDA Fortran.

Но где проходит черта, отделяющая разумную оптимизацию от чрезмерной? В конечном итоге разработчик сам решает, сколько времени и сил потратить на оптимизацию кода. Принимая такое решение, мы должны знать, какого выигрыша ожидать от устранения различных узких мест и сколько усилий придется приложить для этого. Одна из целей этой книги – помочь читателю развить интуицию, необходимую для подобной оценки «окупаемости». Для этого мы обсудим проблемы, встречающиеся при реализации на CUDA Fortran типичных алгоритмов в научных и инженерно-технических приложениях. Там, где возможно, мы расскажем о нескольких обходных путях и о влиянии тех или иных приемов оптимизации на производительность.

1.2. Параллельные вычисления

Прежде чем приступать к написанию кода на CUDA Fortran, следует сказать несколько слов о месте CUDA среди других моделей параллельного программирования. Знать и понимать различные модели параллельного программирования для чтения этой книги необязательно, но тем читателям, которыми уже имеют опыт создания параллельных программ, этот раздел поможет решить, куда отнести CUDA.

Мы уже отмечали, что CUDA – гибридная модель вычислений, предполагающая одновременное использование CPU и GPU. Это удобно, потому что позволяет переносить написанный для CPU код на GPU постепенно. Вычисления на CPU и GPU могут перекрываться во времени, а это само по себе один из аспектов параллелизма.

Но гораздо более высокая степень параллелизма имеет место в самом GPU. Работающие на GPU подпрограммы исполняются параллельно многими потоками. Все они выполняют один и тот же код, но обычно обрабатывают разные данные. Такое *распараллеливание по данным* представляет собой мелкомасштабную (fine-grained) форму параллелизма, наиболее эффективную, когда соседние потоки работают с соседними данными, например с последовательными элементами массива. Эта модель параллелизма кардинально отличается от крупномасштабных (coarse-grained) моделей типа интерфейса передачи сообщений (Message Passing Interface, MPI). В модели MPI данные обычно разбиваются на крупные сегменты, или участки, и каждый процесс MPI выполняет вычисления над целым участком.

Есть ряд характеристик, по которым модель программирования CUDA существенно отличается от моделей программирования на основе CPU. Одно из таких отличий состоит в том, что накладные расходы на создание потоков GPU очень малы. Более того, контекстное переключение, при котором активный поток становится неактивным и наоборот, на GPU производится очень быстро по сравнению с CPU. Причина в том, что GPU не приходится сохранять состояние, как в случае контекстного переключения потоков на CPU. Благодаря столь быстрому контекстному переключению количество запущенных потоков может быть гораздо больше числа ядер GPU (это называется избыточным выделением), что позволяет замаскировать задержки доступа к памяти. Вполне обычна ситуация, когда количество запущенных потоков GPU на порядок превышает число располагаемых ядер. В модели программирования CUDA мы по существу пишем последовательный код, который параллельно исполняется многими потоками GPU. Каждый поток может идентифицировать себя, чтобы обрабатывать различные данные, но сам код, который в CUDA исполняется разными потоками, очень похож на тот, что мы написали бы для последовательно работающего CPU. С другой стороны, во многих других моделях параллельного программирования для CPU код существенно отличается от последовательного. Ниже мы еще будем возвращаться к этим аспектам модели программирования и архитектуры CUDA.

1.3. Основные идеи

В этом разделе мы приведем ряд простых примеров кода на CUDA Fortran, чтобы продемонстрировать основные идеи программирования на этом языке.

Но сначала дадим несколько определений. CUDA Fortran – гибридная модель программирования, то есть одни участки кода исполняются на CPU, а другие – на GPU. В контексте программирования на CUDA Fortran CPU с его памятью называют *хостом*, а GPU с его памятью – *устройством*. Термином *CPU-код* мы будем называть часть программы, исполняемую только на CPU. Подпрограмма, которая вызывается хостом, но выполняется на устройстве, называется *ядром*.

1.3.1. Первая программа на CUDA Fortran

Начнем с кода на Fortran 90, который увеличивает все элементы массива на одну и ту же величину. Собственно увеличение выполняется подпрограммой, которая сама является модулем в смысле Fortran 90. Эта подпрограмма вызывается в цикле и прибавляет к каждому элементу массива значение переданного ей параметра `b`.

```
 1 module simpleOps_m
 2 contains
 3  subroutine increment(a, b)
 4    implicit none
 5    integer, intent(inout) :: a(:)
 6    integer, intent(in) :: b
 7    integer :: i, n
 8
 9    n = size(a)
10    do i = 1, n
11      a(i) = a(i)+b
12    enddo
13
14  end subroutine increment
15 end module simpleOps_m
16
17
18 program incrementTestCPU
19  use simpleOps_m
20  implicit none
21  integer, parameter :: n = 256
22  integer :: a(n), b
23
24  a = 1
25  b = 3
26  call increment(a, b)
27
28  if (any(a /= 4)) then
29     write (*,*) '**** Program Failed ****'
30  else
```

```
31     write (*,*) 'Program Passed '
32  endif
33 end program incrementTestCPU
```

На практике мы решили бы эту задачу иначе. В главной программе мы воспользовались бы имеющимся в Fortran 90 синтаксисом массивов, позволяющим сделать то же самое в одной строчке. Но показанный выше вариант удобнее для сравнения с версией на CUDA Fortran и подчеркивает последовательную природу операций на CPU.

Эквивалентный код на CUDA Fortran выглядит следующим образом:

```
 1 module simpleOps_m
 2  contains
 3   attributes(global) subroutine increment(a, b)
 4   implicit none
 5   integer, intent(inout) :: a(:)
 6   integer, value :: b
 7   integer :: i
 8
 9   i = threadIdx%x
10   a(i) = a(i)+b
11
12  end subroutine increment
13 end module simpleOps_m
14
15
16 program incrementTestGPU
17  use cudafor
18  use simpleOps_m
19  implicit none
20  integer, parameter :: n = 256
21  integer :: a(n), b
22  integer, device :: a_d(n)
23
24  a = 1
25  b = 3
26
27  a_d = a
28  call increment <<<1,n>>>(a_d, b)
29  a = a_d
30
31  if (any(a /= 4)) then
32   write (*,*) '**** Program Failed ****'
33  else
34   write (*,*) 'Program Passed '
35  endif
36 end program incrementTestGPU
```

Первое различие между Fortran 90 и CUDA Fortran – префикс `attributes(global)` подпрограммы, начинающейся в строке 3 реализации на CUDA Fortran. Атрибут `global` означает, что код выполняется на устройстве, но вызывается из хоста. (Слово `global`, как и все прочие атрибуты подпрограмм, обозначает область видимости и говорит, что подпрограмма видна как хосту, так и устройству.)

Второе заметное различие – замена цикла do в строках 10–12 версии для Fortran 90 двумя предложениями: в первом (строка 9) инициализируется индекс `i`, а второе (строка 10) содержит тело прежнего цикла. Это следствие отличия между последовательным и параллельным выполнением. В коде для CPU увеличение элементов массива «`a`» производится последовательно в цикле do, исполняемом одним потоком CPU. В версии для CUDA Fortran подпрограмма параллельно выполняется несколькими потоками ГП. Каждый поток идентифицируется с помощью встроенной переменной `threadIdx`, доступной в любом месте исполняемого устройством кода; эта переменная используется как индекс элемента массива. Такой вид параллелизма, когда последовательные потоки модифицируют соседние элементы массива, называется *мелкомасштабным* (fine-grained) параллелизмом.

Главная программа в версии для CUDA Fortran выполняется на хосте. Определения и производные типы содержатся в модуле `cudafor`, который включается с помощью команды `use` в строке 17; затем, в строке 18, включается модуль `simpleOps_m`. Как мы уже отмечали, в CUDA Fortran имеются два независимых адресных пространства: на хосте и на устройстве. То и другое видно исполняемой на хосте программе, а чтобы показать, что переменная должна размещаться в памяти устройства, при ее объявлении указывается атрибут `device`, как, например, при объявлении переменной `a_d` в строке 22 версии для CUDA Fortran. Суффикс «`_d`» – не обязательное, но полезное соглашение, позволяющее отличить переменные, размещенные в памяти устройства. Поскольку в этом отношении язык CUDA Fortran строго типизированный, для передачи данных между хостом и устройством достаточно использовать предложения присваивания. Так мы и поступаем в строке 27, где уже инициализированный на хосте массив `a` передается в память устройства (типа DRAM – динамическое запоминающее устройство с произвольной выборкой).

После того как данные перемещены в память устройства, можно запускать ядро, то есть выполняемую на устройстве подпрограмму (строка 28). Заключенная в тройные угловые скобки группа параметров между именем подпрограммы и списком аргументов в строке 28

называется *конфигурацией выполнения* и определяет количество потоков GPU, исполняющих данное ядро. Подробнее о конфигурации выполнения мы будем говорить ниже, а пока скажем лишь, что запись `<<<1,n>>>` означает, что ядро выполняется `n` потоками GPU.

Массивы, переданные ядру в качестве аргументов, например `a_d`, должны размещаться в памяти устройства, но это не относится к скалярным аргументам, например второму аргументу `b`, который находится в памяти хоста. Исполняющая среда CUDA сама позаботится о перемещении скалярных аргументов из памяти хоста в память устройства, но ожидает, что аргументы передаются по значению. По умолчанию аргументы в Fortran передаются по ссылке, но могут быть переданы по значению, если указать в объявлении переменной атрибут `value`, как в строке 6 версии для CUDA Fortran. Атрибут `value` был включен в стандарт Fortran 2003 как часть механизма совместной работы с кодом на C.

При программировании в гибридной модели, каковой является CUDA, необходимо думать о *синхронизации* между хостом и устройством. Чтобы эта программа работала правильно, мы должны быть уверены, что передача данных от хоста устройству в строке 27 завершится до начала выполнения ядра, и что ядро закончит работу до начала передачи данных от устройства хосту в строке 29. В данном случае такое поведение гарантируется в силу того, что передача данных с помощью операторов присваивания в строках 27 и 29 – блокирующая, или синхронная операция. Такая операция передачи данных не начнется, пока не будут завершены все ранее запущенные операции на GPU, а следующие за ней не начнутся, пока не завершится эта передача данных. Блокирующая природа передачи данных полезна для неявной синхронизации CPU и GPU.

Передача данных с помощью оператора присваивания – блокирующая, или синхронная операция, тогда как запуск ядра – операция неблокирующая, или асинхронная. После запуска ядра в строке 28 управление немедленно возвращается хосту. Однако у нас все же есть гарантия требуемого поведения, потому что передача данных в строке 29, будучи блокирующей операцией, не начнется раньше завершения ядра.

Существуют процедуры, которые производят передачу данных асинхронно, позволяя выполнять вычисления на устройстве одновременно с обменом данными между ним и хостом. Имеются также средства явной синхронизации хоста и устройства. Все это мы будем обсуждать в разделе 3.1.3.

1.3.2. Обобщение на большие массивы

В предыдущем примере параметр `n` в конфигурации выполнения `<<<1,n>>>` налагал ограничение на размер массива: какое конкретно, зависит от используемого устройства CUDA. Для изделий на основе архитектуры Kepler™ или Fermi™, например карт Tesla K20 и C2050, максимальное значение `n` равно `1024`, а для карт предыдущего поколения – `512` (сведения об ограничениях см. в приложении А). Чтобы увеличить размер массива, необходимо изменить первый параметр конфигурации выполнения, поскольку количество потоков GPU, исполняющих код, на самом деле равно произведению обоих чисел. Почему так сделано? Для чего потоки GPU группируются подобным образом? Дело в том, что группировка потоков в этой модели программирования отражает группировку процессорных элементов в аппаратуре GPU.

Основным вычислительным элементом GPU является потоковый процессор, который чаще называют просто *процессорным ядром*. Потоковый процессор выполняет операции с плавающей точкой. Потоковые процессоры объединяются в мультипроцессоры, которые содержат ограниченное количество ресурсов, используемых запущенными потоками, а именно регистров и разделяемой памяти. Эта структура изображена на рис. 1.1, где показано устройство с поддержкой CUDA, содержащее GPU с четырьмя мультипроцессорами, каждый из которых состоит из 32 потоковых процессоров.

Аналогом мультипроцессора в модели программирования является блок потоков. Это группа потоков, которые назначены какому-то одному мультипроцессору и после назначения уже не могут перемещаться. На одном мультипроцессоре может размещаться несколько блоков потоков, но количество блоков, одновременно размещенных на одном мультипроцессоре, ограничено доступными этому мультипроцессору ресурсами, а также ресурсами, необходимыми каждому блоку потоков.

Но вернемся к нашему примеру. Вызванное ядро запускает *сетку* (grid) блоков потоков. Количество запускаемых блоков определяется первым параметром конфигурации выполнения, а количество потоков в одном блоке – вторым параметром. Таким образом, наша первая программа на CUDA Fortran запустила сетку, содержащую один блок из 256 потоков. Чтобы обработать больший массив, мы можем запустить несколько блоков потоков, как в следующем примере.

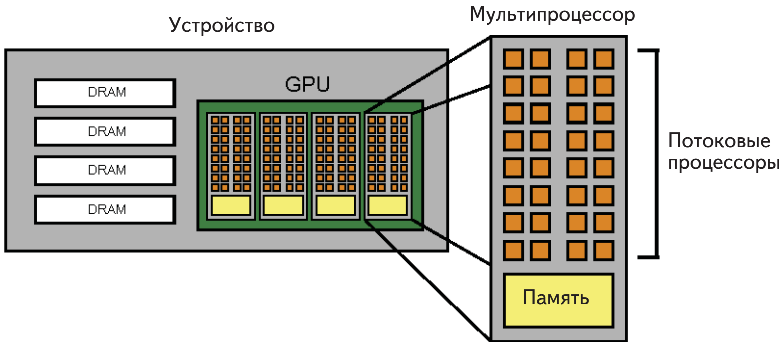

Рис. 1.1. Иерархия вычислительных элементов в GPU; потоковые процессоры объединяются в мультипроцессор

```
 1 module simpleOps_m
 2  contains
 3    attributes( global) subroutine increment(a, b)
 4    implicit none
 5    integer, intent(inout) :: a(:)
 6    integer, value :: b
 7    integer :: i, n
 8
 9    i = blockDim%x*( blockIdx%x-1) + threadIdx%x
10    n = size(a)
11    if (i <= n) a(i) = a(i)+b
12
13  end subroutine increment
14 end module simpleOps_m
15
16
17 program incrementTest
18  use cudafor
19  use simpleOps_m
20  implicit none
21  integer, parameter :: n = 1024*1024
22  integer, allocatable :: a(:)
23  integer, device, allocatable :: a_d (:)
24  integer :: b, tPB = 256
25
26  allocate(a(n), a_d(n))
27  a = 1
28  b = 3
29
30  a_d = a
31  call increment <<<ceiling(real(n)/tPB),tPB>>>(a_d, b)
32  a = a_d
```

Конец ознакомительного фрагмента.
Приобрести книгу можно
в интернет-магазине
«Электронный универс»
e-Univers.ru