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. – М.: Издательство Московского университета, 2012. – 336 с., илл. – (Серия «Суперкомпьютерное образование»)

ISBN 978-5-211-06340-2

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 978-5-211-06340-2

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-

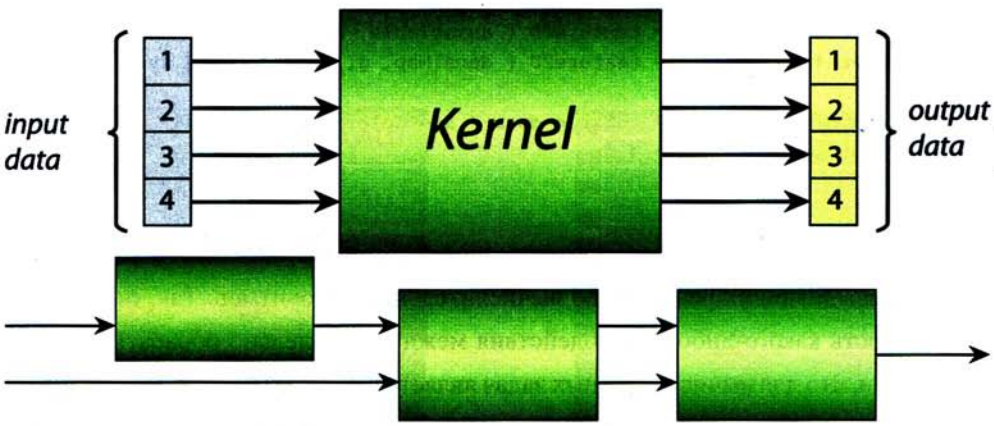

Рис. 1.1. Работа SIMD архитектуры

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).

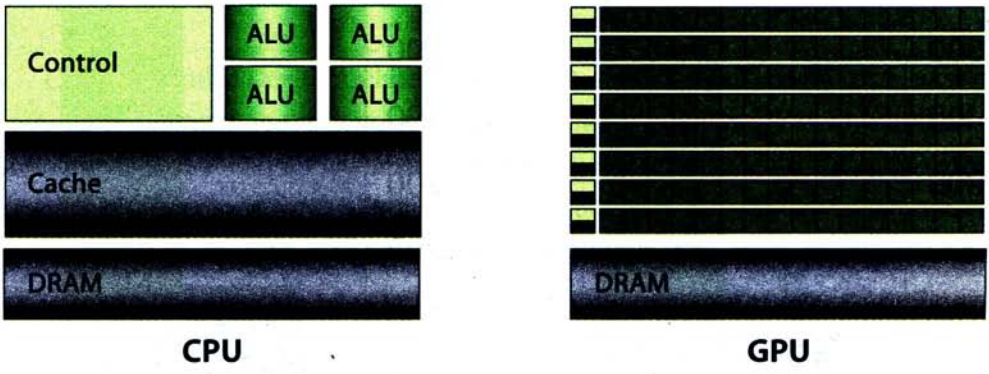

Рис. 1.2. GPU отводит большее колличество транзисторов под операции над данными

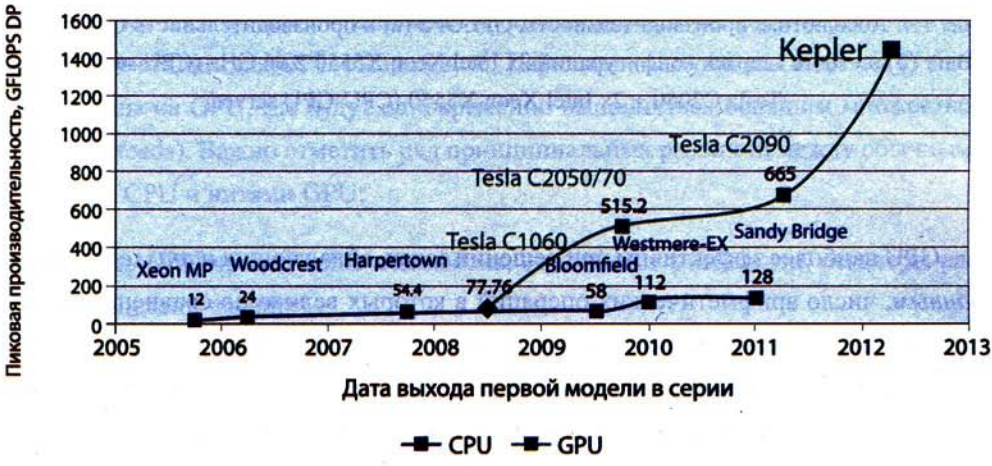

Рис. 1.3. Динамика роста пиковой производительности вычислений с двойной точностью CPU (Intel Xeon) и GPU (NVIDIA Tesla)

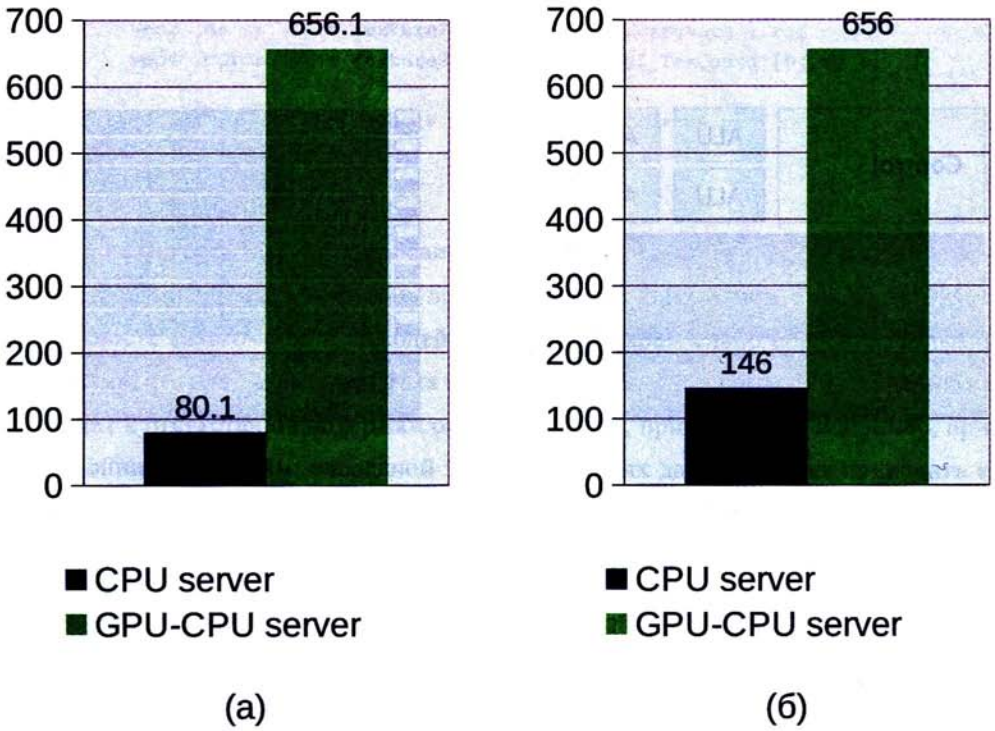

Рис. 1.4. Абсолютная производительность, GFLOPS (а) и производительность GFLOPS на Ватт (б) на тесте Linpack конфигураций 2x Intel Xeon X5550 2.66 GHz (CPU server) и 2x Tesla C2050 + 2x Intel Xeon X5550 (CPU-GPU server)

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*.

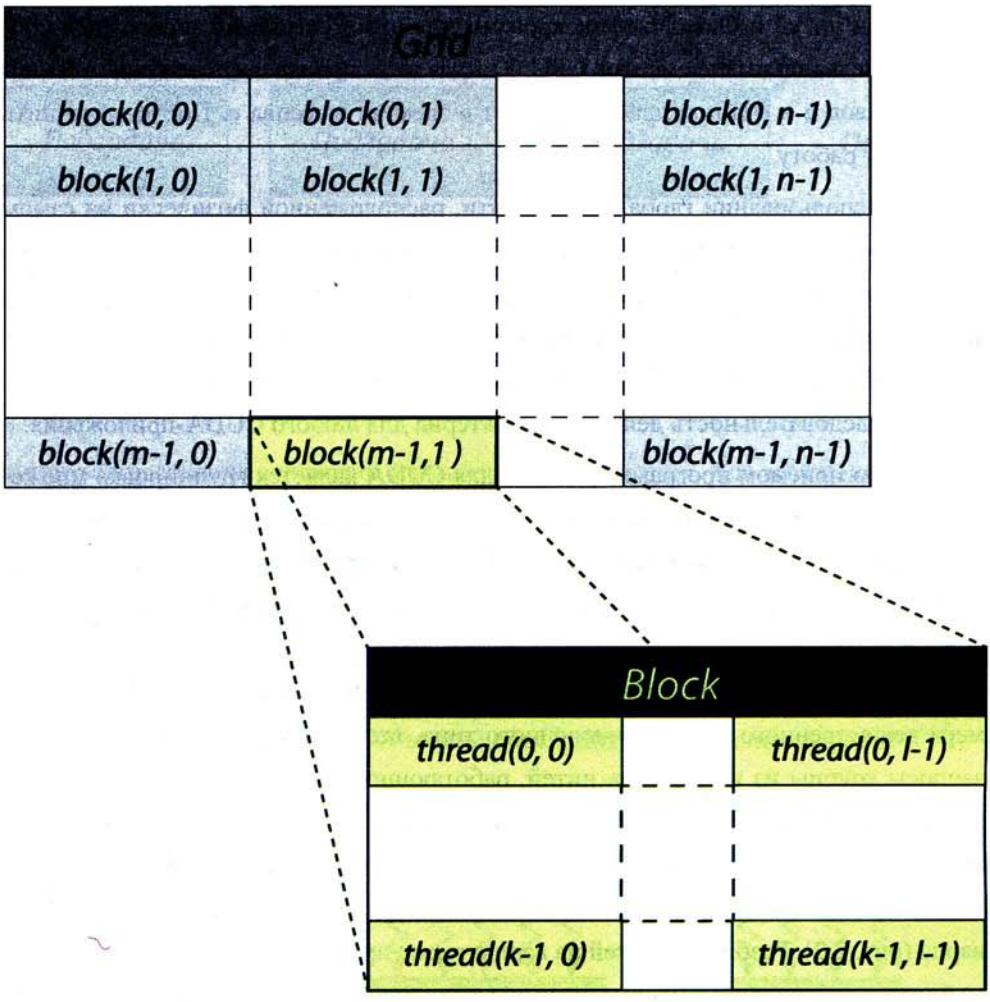

Рис. 2.1. Иерархия нитей в CUDA

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-

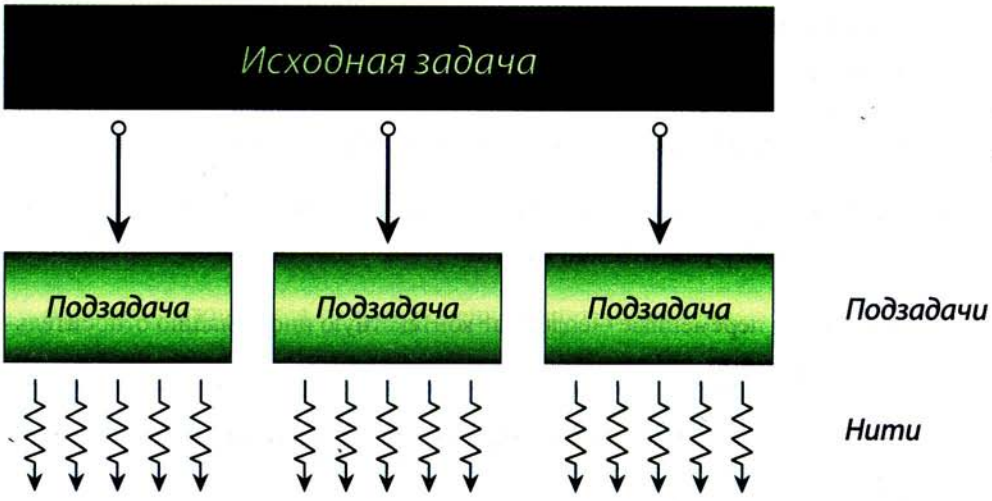

Рис. 2.2. Разбиение исходной задачи на набор независимо решаемых подзадач

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:

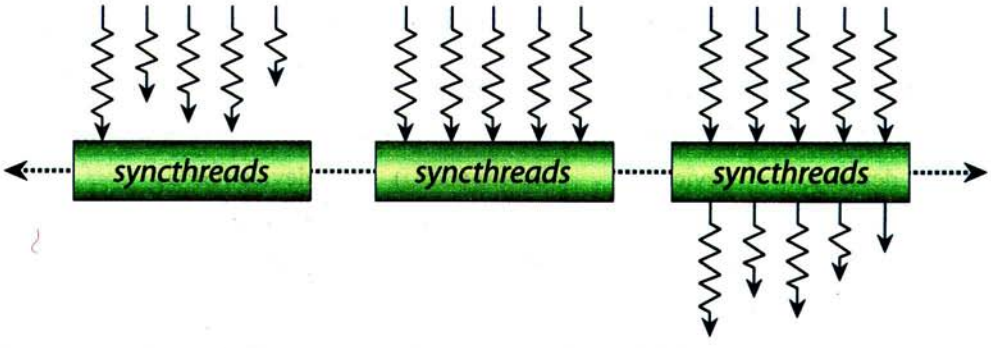

Рис. 2.3. Барьерная синхронизация

- 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

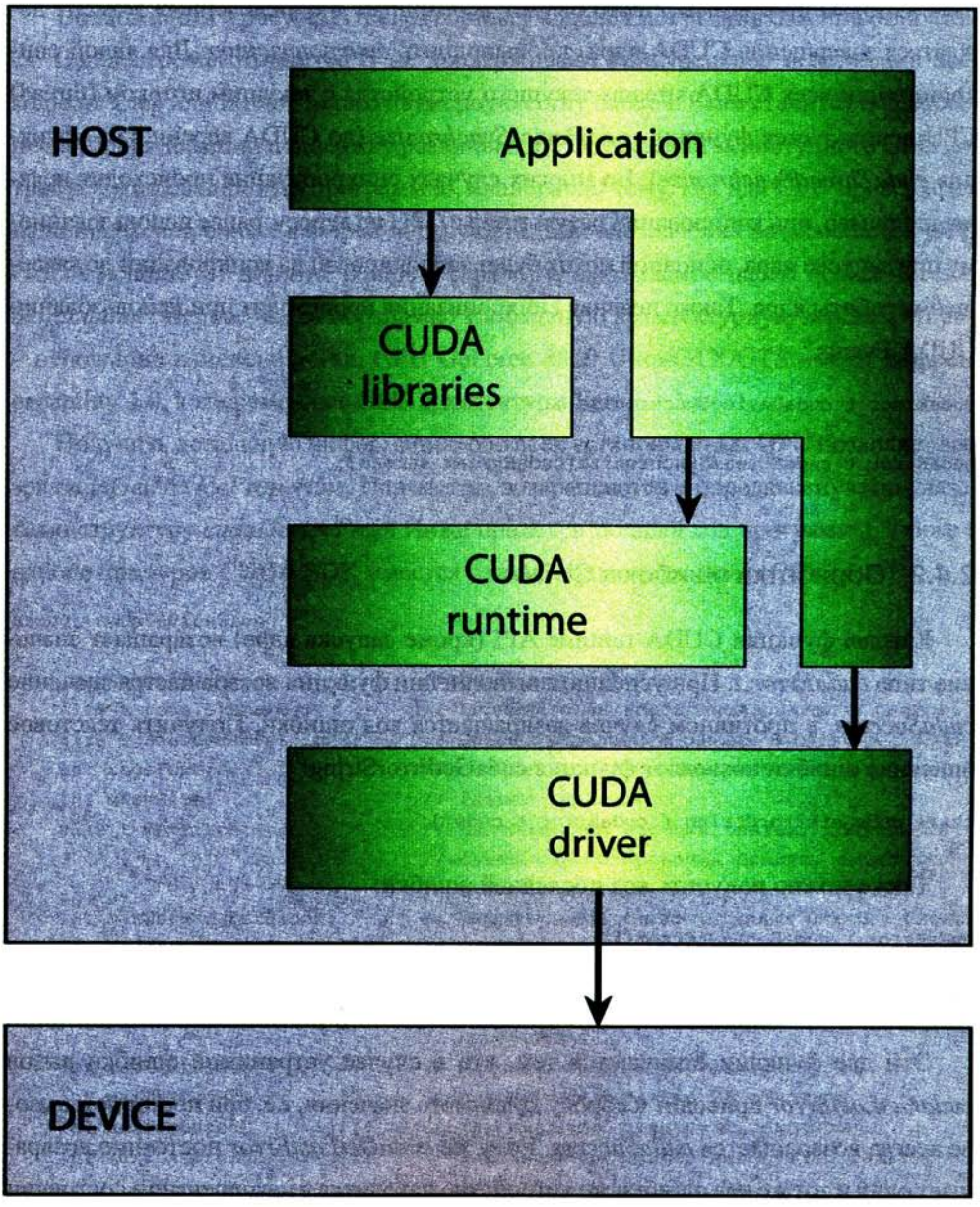

Рис. 2.4. Программный стек CUDA

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()
{
    ...
}
```

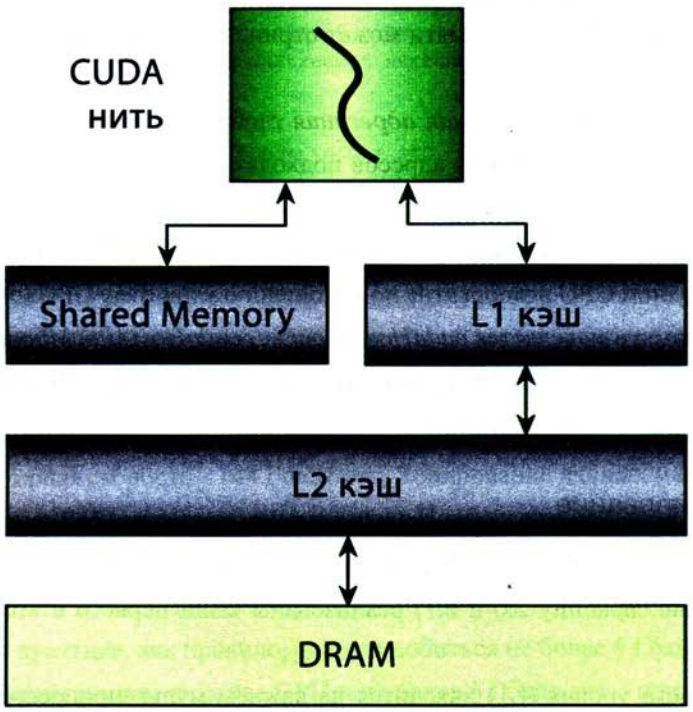

Рис. 3.1. Fermi – подсистема памяти

```
// 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:

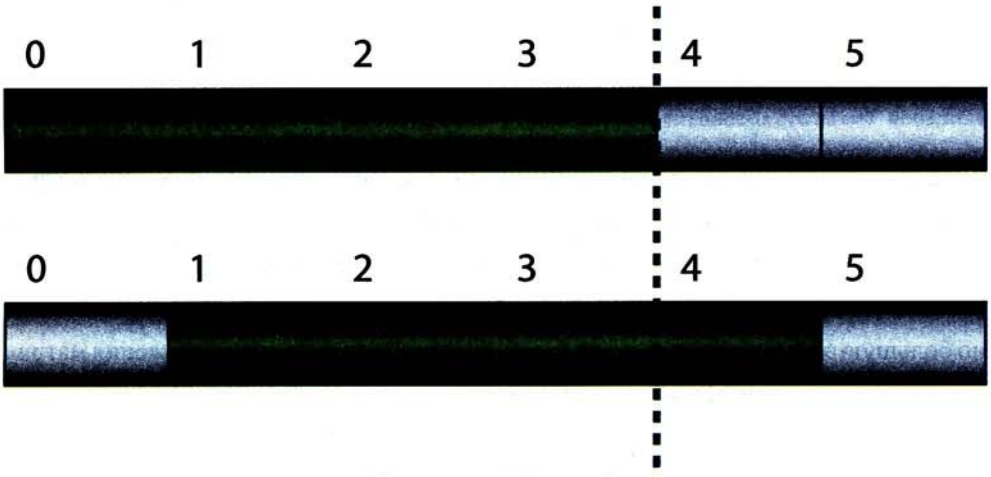

Рис. 3.2. Пример выровненного (сверху) и невыровненного (внизу) 4-байтового блока

```
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.

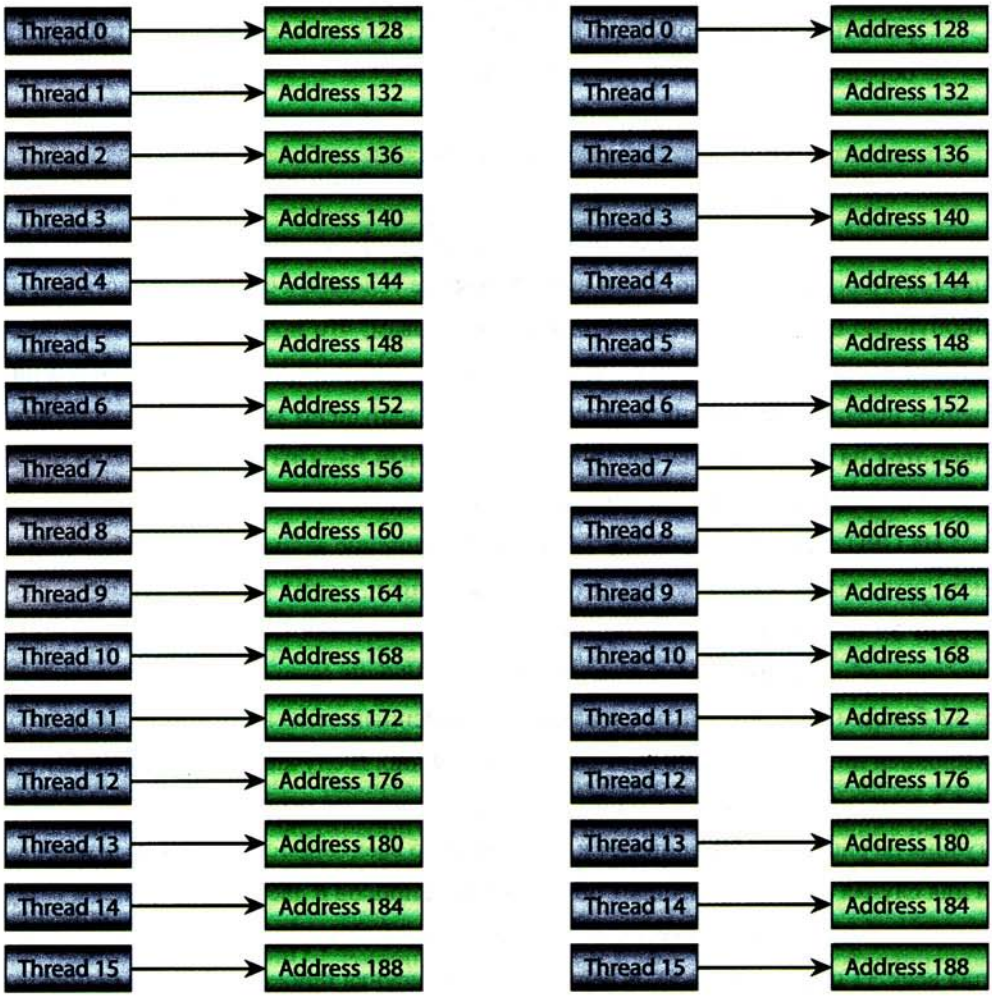

Рис. 3.3. Паттерны обращения к памяти, дающие объединение для GPU с compute capability 1.0 и 1.1

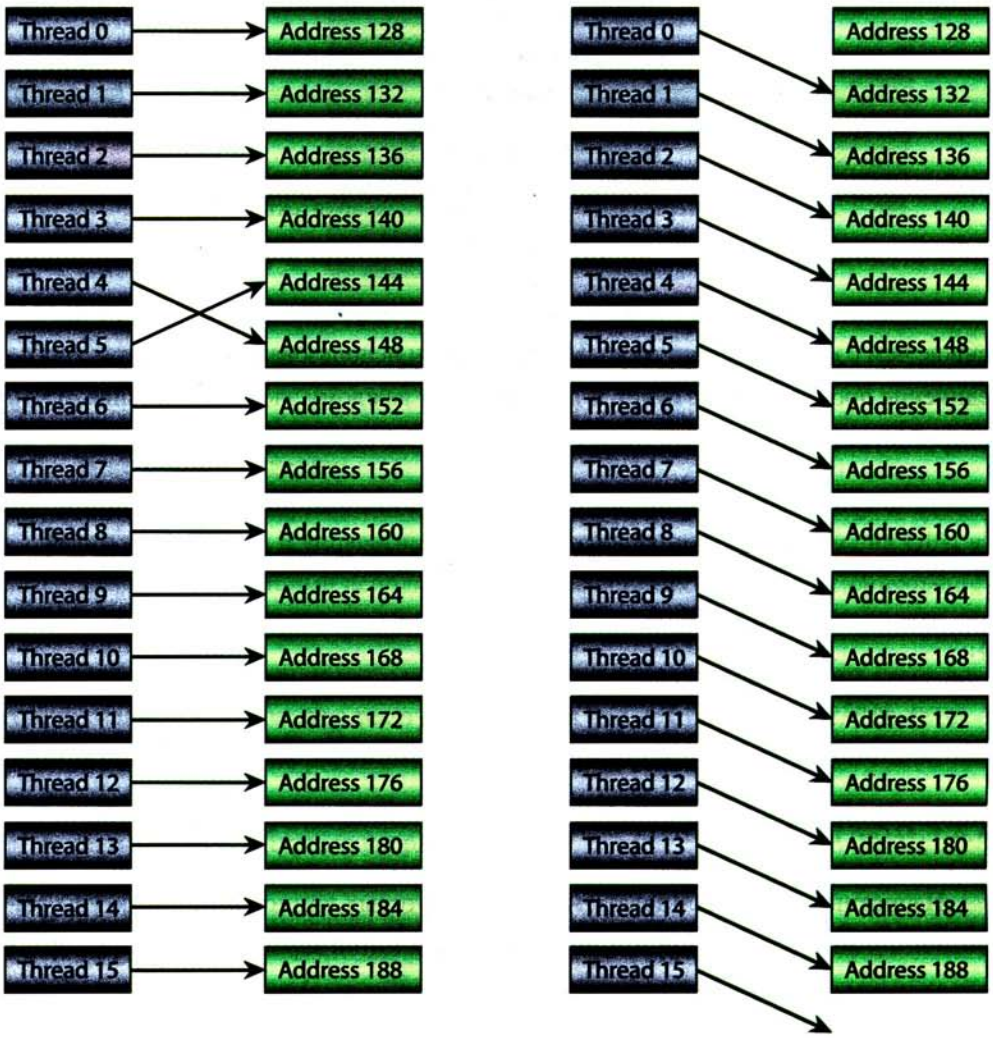

Рис. 3.4. Паттерны обращения к памяти, не дающие объединение для GPU с compute capability 1.0 и 1.1

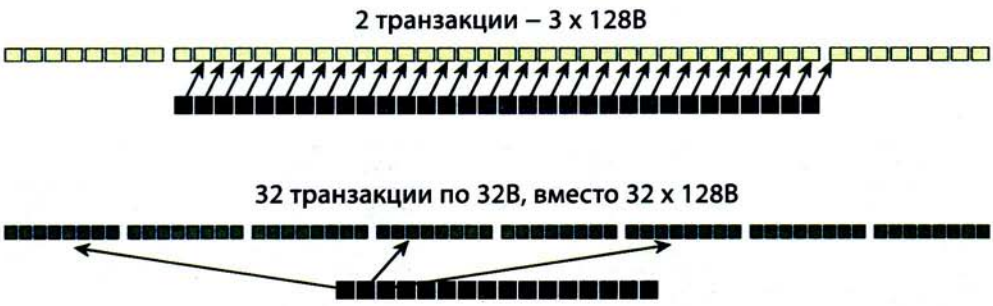

Рис. 3.5. Паттерны обращения к памяти на архитектуре Fermi

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];
```

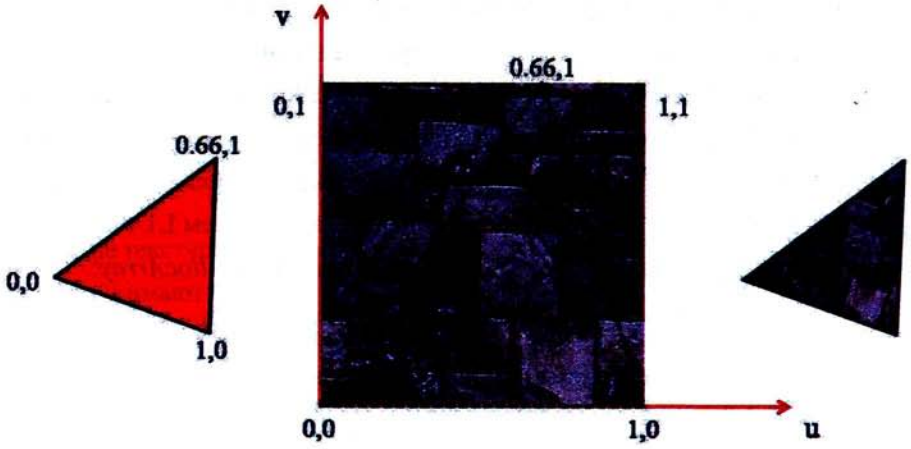

Рис. 3.6. Текстура

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.

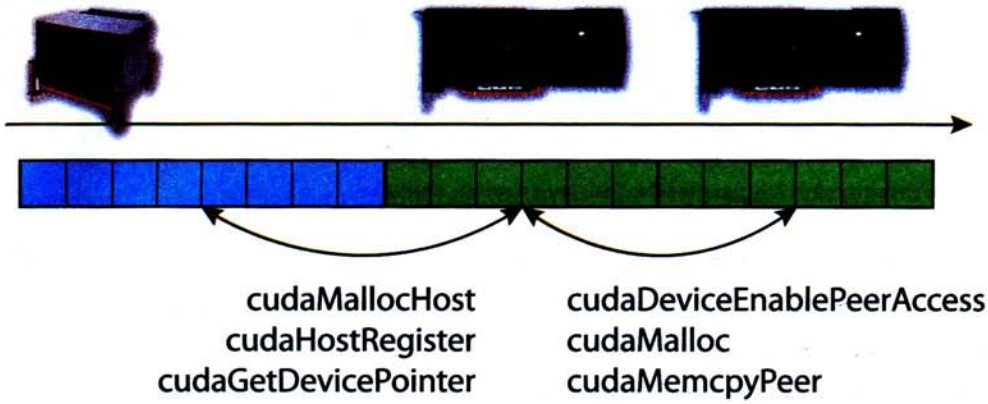

Рис. 3.7. Общее виртуальное адресное пространство (UVA)

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;
```

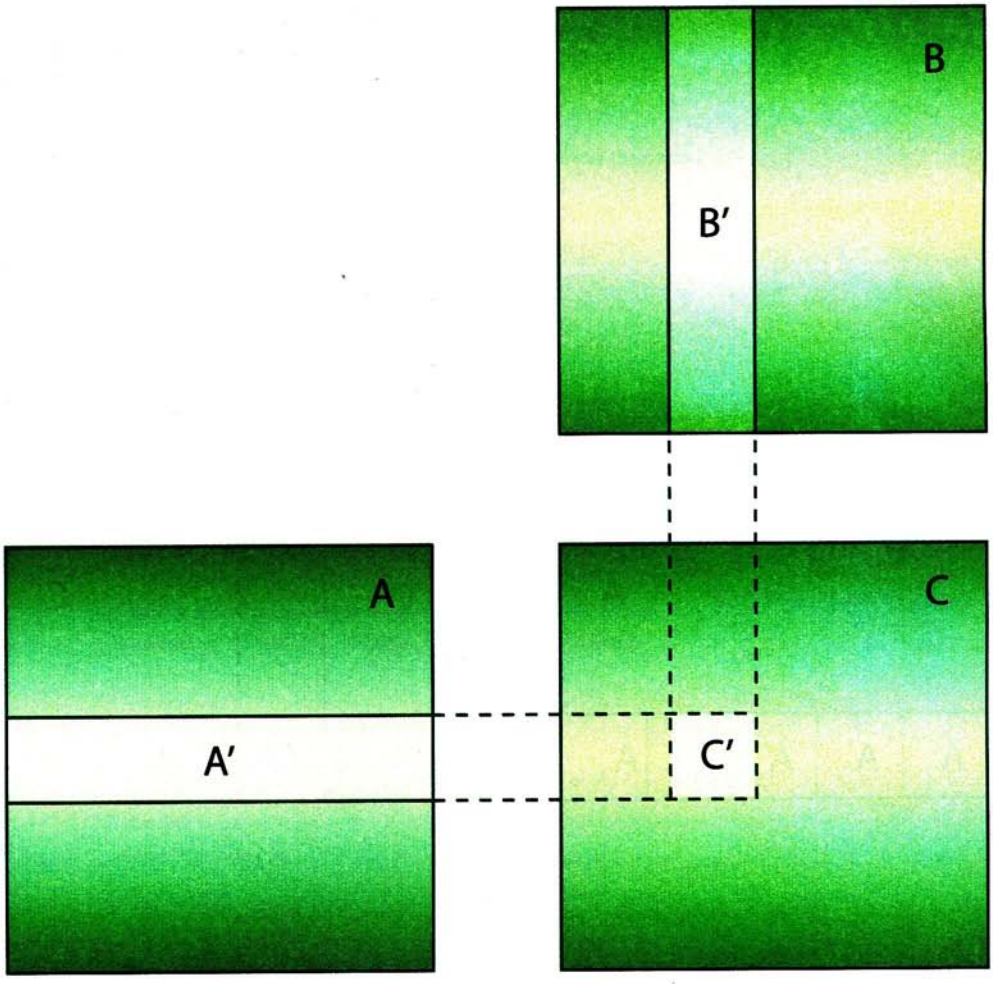

Рис. 3.8. Для вычисления элементов C' нужны только элементы из A' и B'

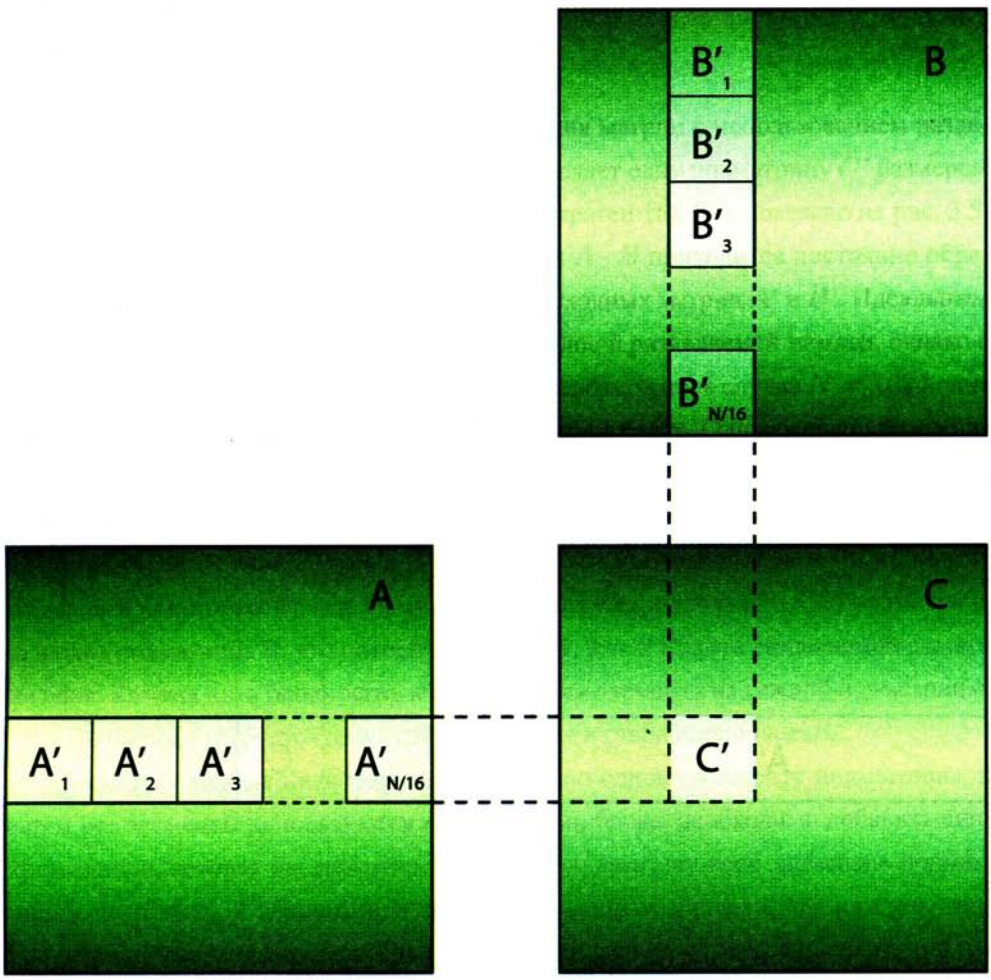

Рис. 3.9. Разложение требуемой подматрицы в сумму произведений матриц 16 × 16

```
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 $C = AB$, 2048×2048 на Tesla C2070

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

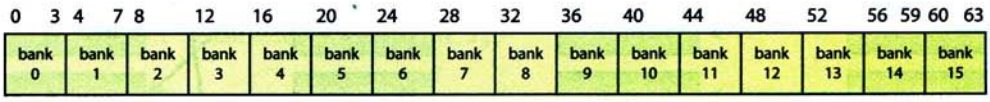

Рис. 3.10. Разбиение 64 байт разделяемой памяти на банки

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-

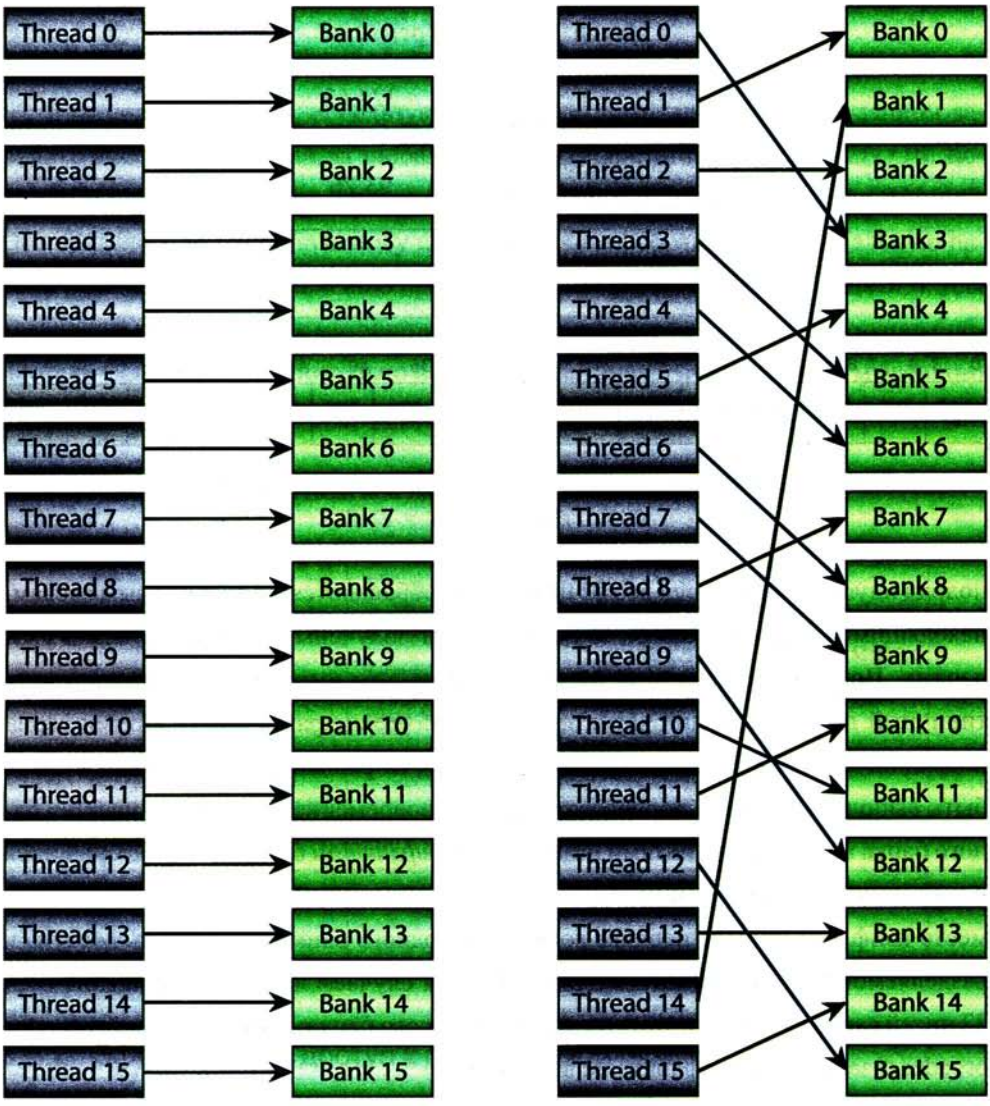

Рис. 3.11. Примеры доступа к разделяемой памяти, в которых не возникает конфликт банков

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;
```

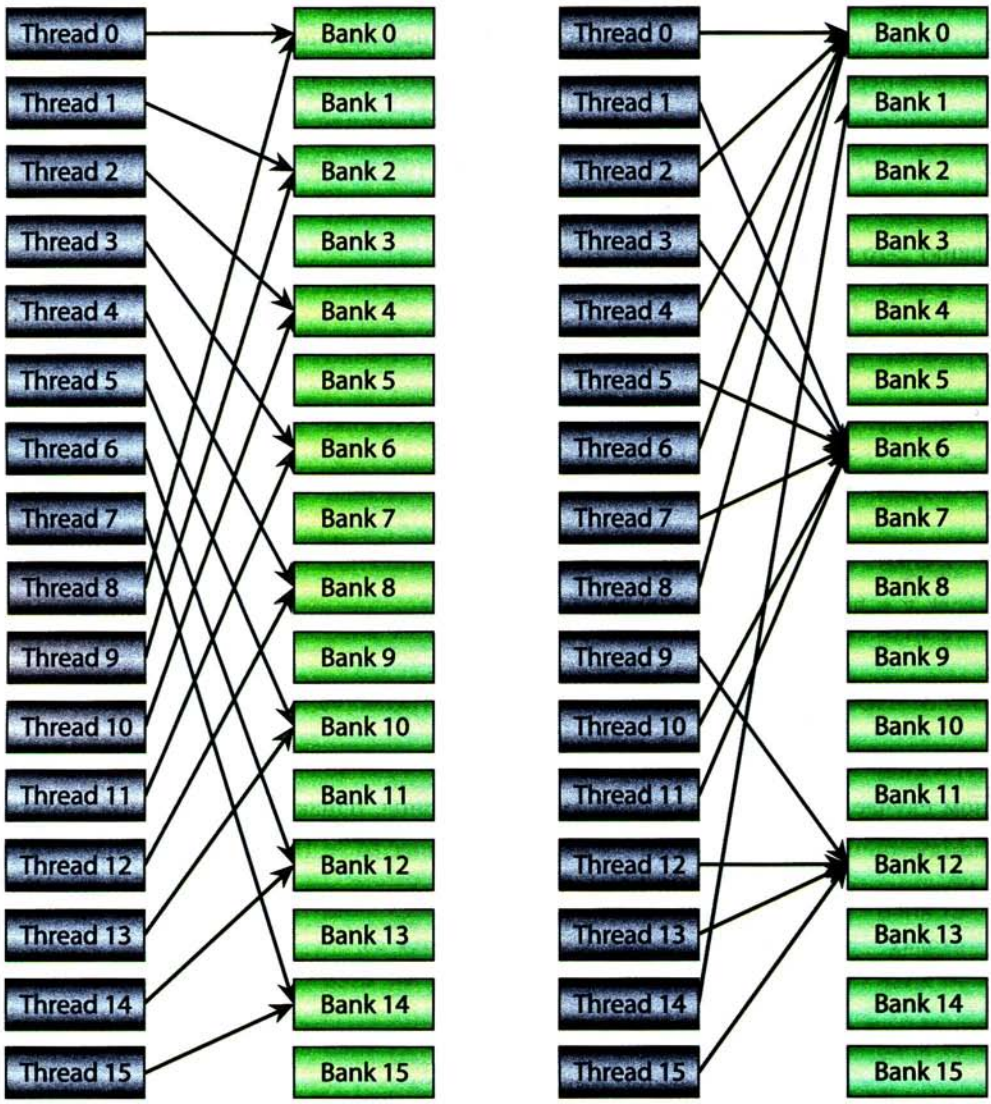

Рис. 3.12. Примеры доступа к разделяемой памяти, в которых возникает конфликт банков

```
    // 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

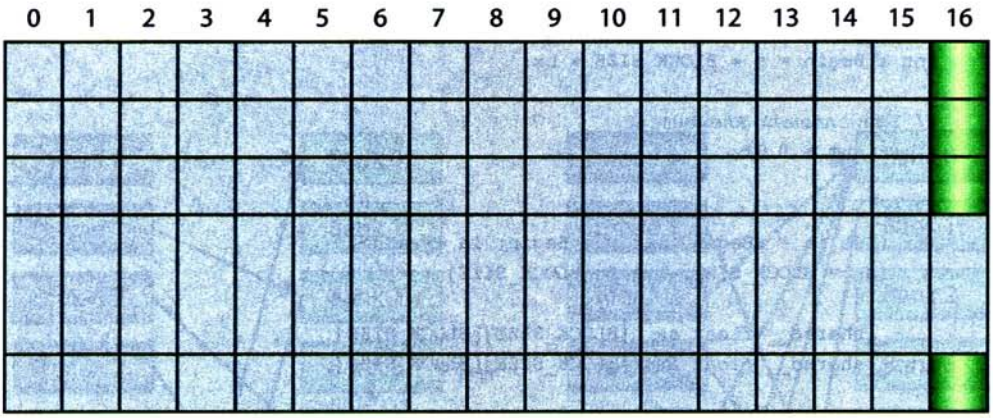

Рис. 3.13. Добавление к матрице 16 × 16 одного столбца для избавления от конфликта банков

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 $C = AA^T$, 2048×2048 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