Оглавление

1. Введение

1.1. От обработки графики до параллельных вычислений общего назначения

Благодаря ненасытному рыночному спросу на 3D-графику в программируемый графический процессор или графический процессор превратился в многопоточный многоядерный процессор с потрясающей вычислительной мощностью и очень высокой пропускной способностью памяти, как показано на рисунке 2 и рисунке 3.

Причина различия возможностей с плавающей запятой между процессором и графическим процессором заключается в том, что графический процессор специализирован для интенсивных вычислений, высокопараллельных вычислений - именно то, что представляет собой графический рендеринг, и поэтому сконструирован таким образом, что большее количество транзисторов занимается обработкой данных, а не кэшированием данных и управлением потоком, как схематически показано на рисунке 1.

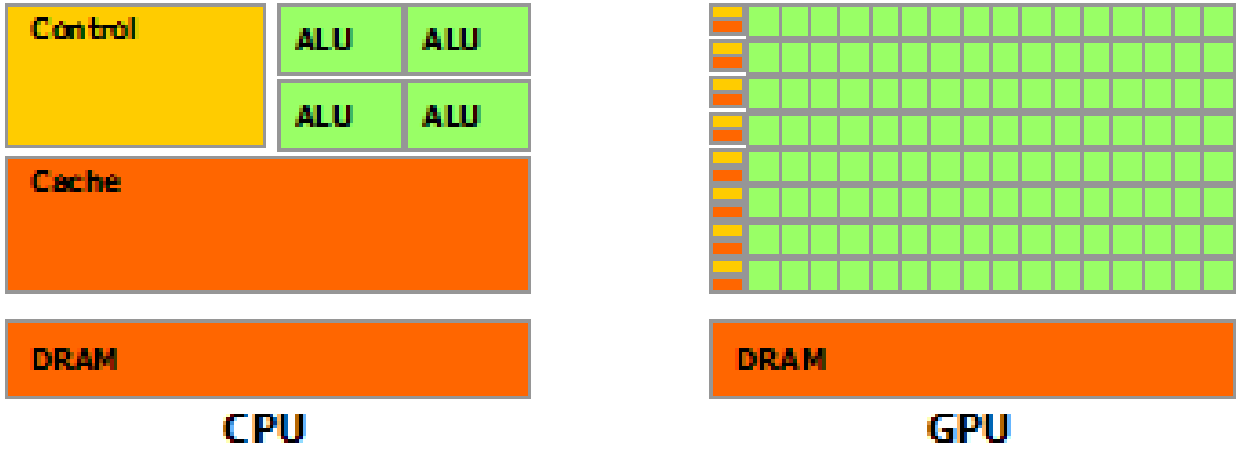

Рисунок 1. Графический процессор выделяет больше транзисторов для обработки данных

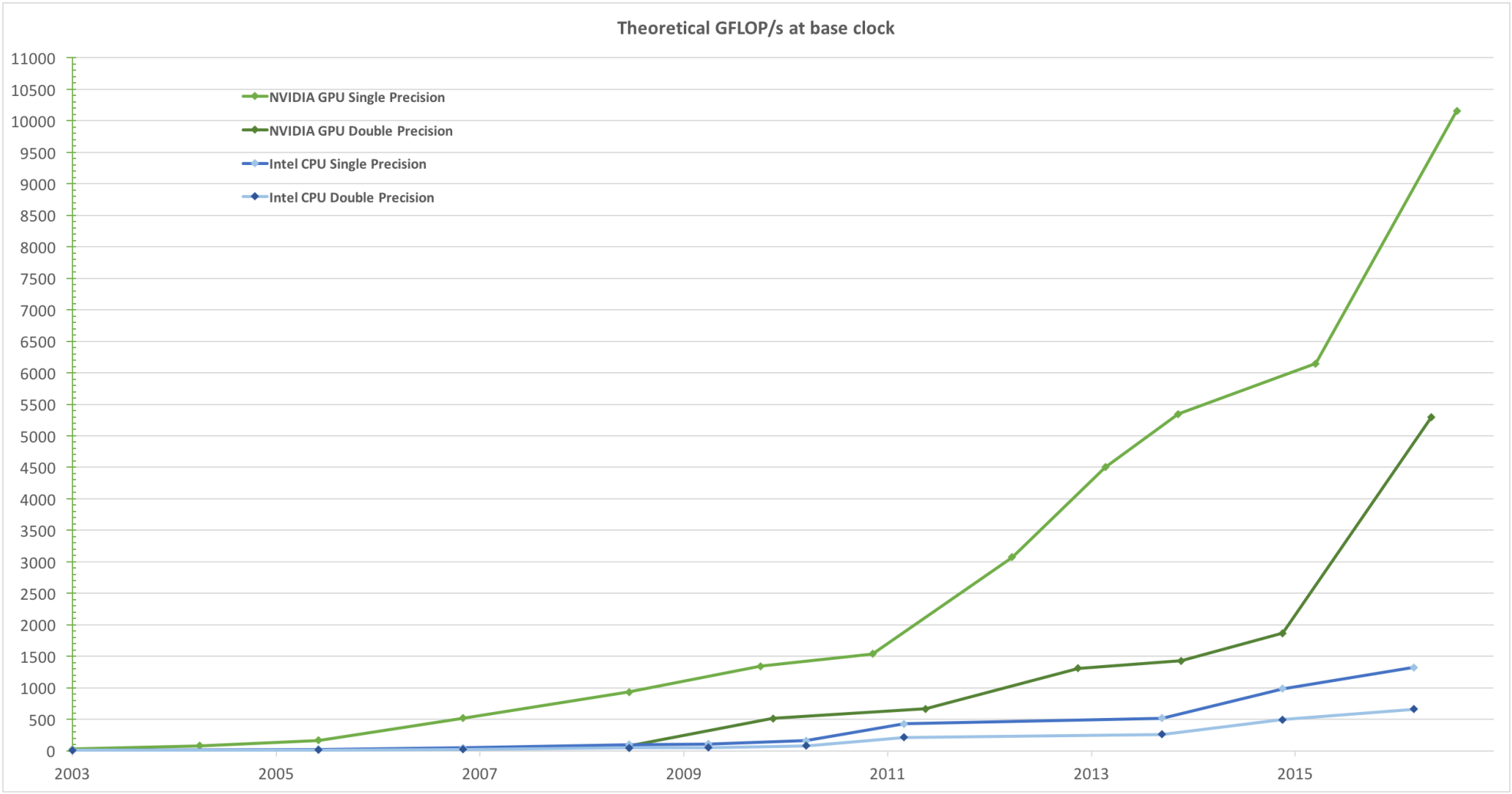

Рисунок 2. Операции с плавающей запятой в секунду для центрального процессора и графического процессора

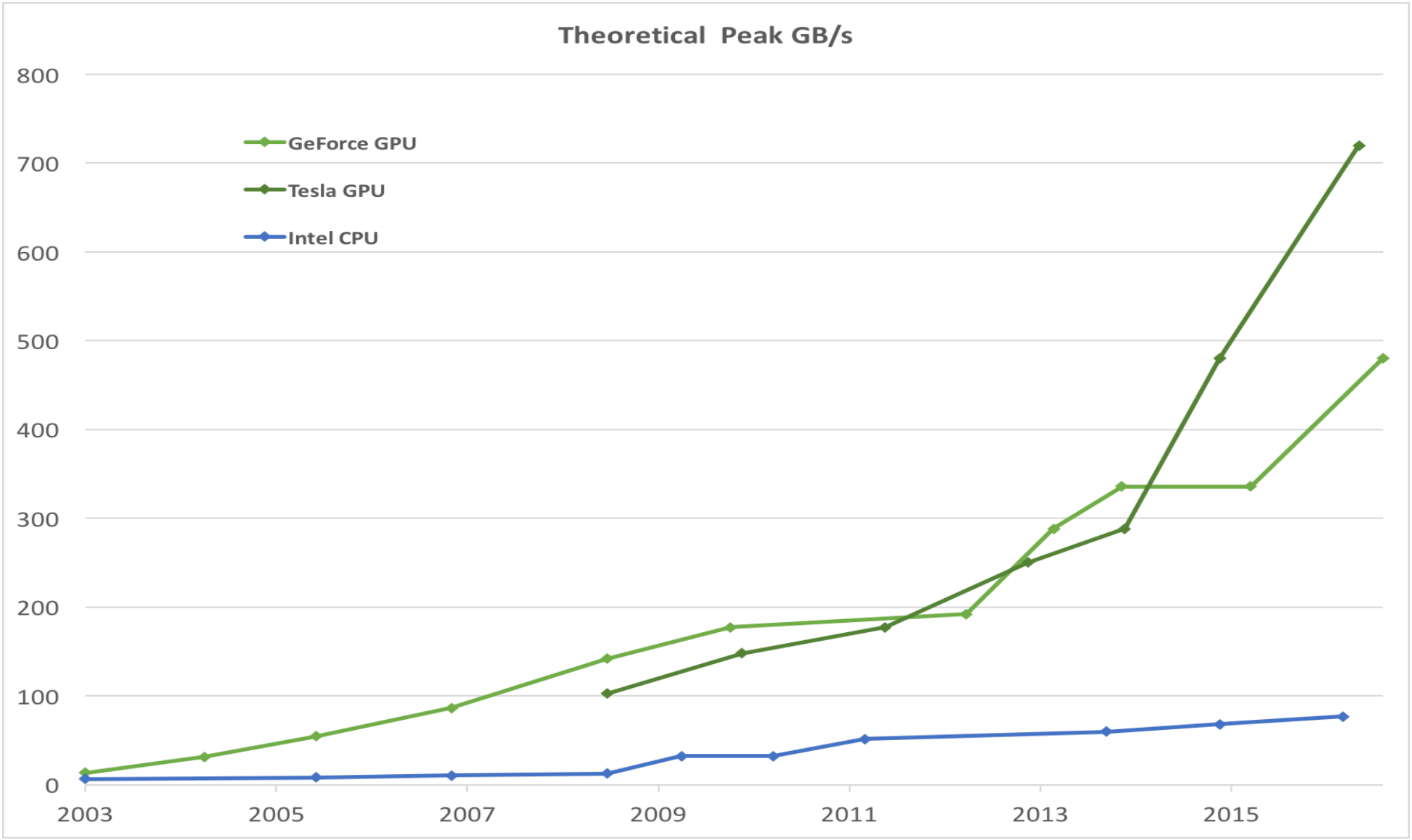

Рисунок 3. Пропускная способность памяти для центрального процессора и графического процессора

Графический процессор особенно хорошо подходит для решения задач, которые могут быть выражены в виде параллельных вычислений данных - одна и та же программа выполняется для многих параллельных элементов данных, с большим количеством арифметических операций по отношению к операциям с памятью. Поскольку одна и та же программа выполняется для каждого элемента данных, существует более низкая потребность в сложном управлении потоками, и поскольку она выполняется над многими элементами данных и имеет большое количество арифметических операций, латентность доступа к памяти может быть скрыта с помощью вычислений, а не за счёт кэширования данных.

Параллельная обработка данных подразумевает распределение данных по независимым потокам, которые параллельно занимаются обработкой своей части данных. Многие приложения, обрабатывающие большие массивы данных, могут использовать модель параллельного программирования данных для ускорения вычислений. В 3D-рендеринге большие множества пикселей и вершин распределяются по параллельным потокам. Подобным образом приложения обработки изображений и мультимедиа, такие как пост-обработка визуализированных изображений, кодирование и декодирование видео, масштабирование изображения, распознавание образов, могут распределять блоки изображения и пиксели по потокам для параллельной обработки. Фактически, многие алгоритмы ускоряются путем параллельной обработки данных, от общей обработки сигналов или моделирования физики до вычислительного финансирования или вычислительной биологии.

1.2. CUDA®: платформа параллельных вычислений общего назначения и модель программирования

В ноябре 2006 года NVIDIA представила CUDA®, платформу параллельных вычислений общего назначения и модель программирования, которая использует параллельный вычислительный движок в графических процессорах NVIDIA для решения многих сложных вычислительных задач более эффективным способом, чем на центральном процессоре.

CUDA поставляется с программной средой, которая позволяет разработчикам использовать C как высокоуровневый язык программирования. Как показано на рисунке 4, поддерживаются другие языки, интерфейсы прикладного программирования или подходы, основанные на директивах, такие как FORTRAN, DirectCompute, OpenACC.

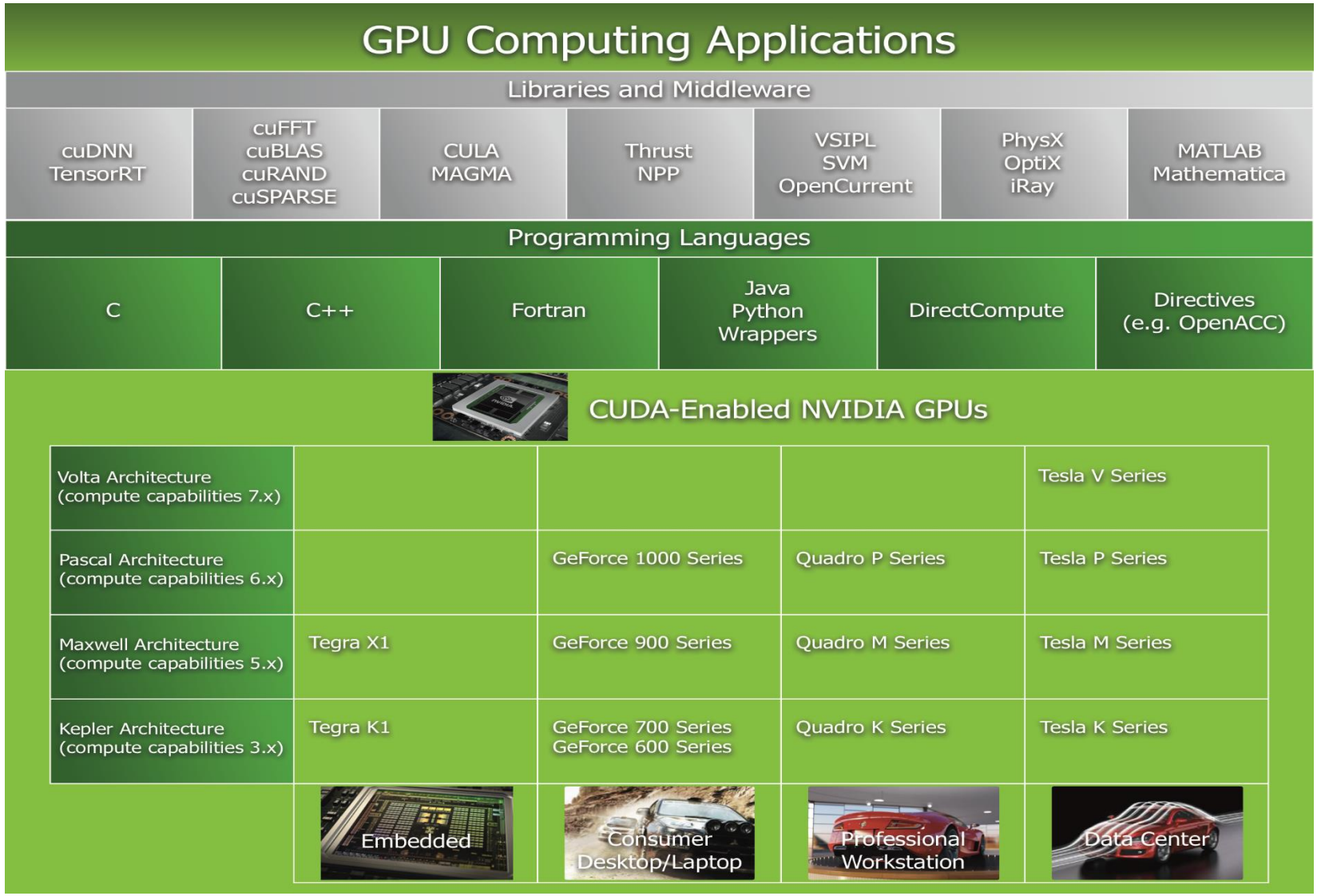

Рисунок 4. Приложения для графических процессоров. CUDA предназначен для поддержки различных языков и интерфейсов прикладного программирования.

1.3. Масштабируемая модель программирования

Появление многоядерных процессоров и многопользовательских графических процессоров означает, что основные процессоры теперь являются параллельными системами. Более того, их параллелизм продолжает действовать в соответствии с законом Мура. Задача состоит в том, чтобы разработать прикладное программное обеспечение, которое прозрачно масштабирует свой параллелизм, чтобы использовать все большее число процессорных ядер, так же, как приложения 3D-графики прозрачно масштабируют их параллелизм для многих чистых графических процессоров с различным количеством ядер.

Модель параллельного программирования CUDA предназначена для преодоления этой проблемы, сохраняя при этом низкую кривую обучения для программистов, знакомых со стандартными языками программирования, такими как C.

По сути, это три ключевые абстракции - иерархия групп потоков, разделяемая память и барьерная синхронизация, которые просто представлены программисту как минимальный набор языковых расширений.

Эти абстракции обеспечивают параллелизм данных и параллелизм потоков, вложенные параллелизм данных и параллелизм задач. Они подталкивают программиста на то, чтобы разделить проблему на грубые подзадачи, которые могут решаться независимо параллельно блоками потоков, а каждая подзадача разбивается на более мелкие куски, которые могут решаться совместно всеми потоками внутри блока.

Это разложение сохраняет выразительность языка, позволяя потокам взаимодействовать при решении каждой подзадачи и в то же время обеспечивает автоматическую масштабируемость. Действительно, каждый блок потоков может быть запланирован на любом из доступных мультипроцессоров в графическом процессоре в любом порядке одновременно или последовательно, так что скомпилированная программа CUDA может выполняться на любом количестве мультипроцессоров, как показано на рисунке 5, и только runtime система должна знать количество физических мультипроцессоров.

Эта масштабируемая модель программирования позволяет архитектуре GPU охватывать широкий диапазон рынка, просто масштабируя количество мультипроцессоров и разделов памяти: от высокопроизводительных энтузиастов графических процессоров GeForce и профессиональных вычислительных продуктов Quadro и Tesla до самых недорогих основных графических процессоров GeForce.

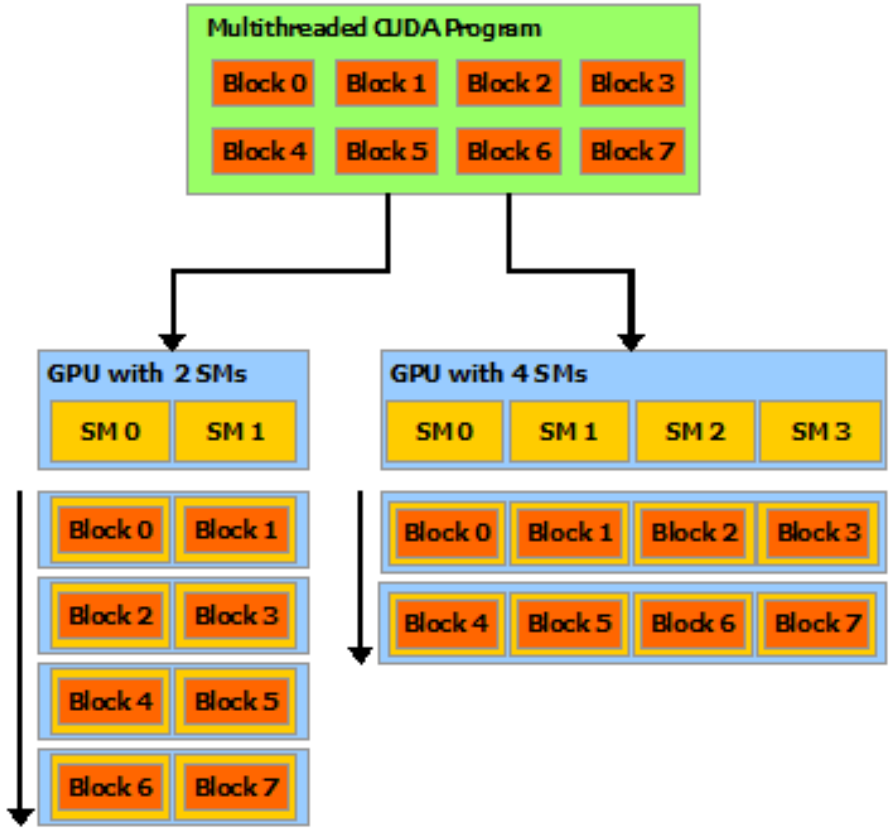

Рисунок 5. Автоматическая масштабируемость

Примечание. Графический процессор построен вокруг массива потоковых мультипроцессоров (SMs). Многопоточная программа разбивается на блоки потоков, которые выполняются независимо друг от друга, так что GPU с большим количеством процессоров будет автоматически выполнять программу за меньшее время, чем графический процессор с меньшим количеством процессоров.

1.4. Вопросы для самопроверки

- Назовите отличия GPU от CPU
- Для решения каких задач хорошо подходят графические процессоры? Почему?

2. Модель программирования

В этой главе представлены основные концепции модели программирования CUDA, излагая, как они отображаются в языке программирования C. Подробное описание CUDA C приведено в интерфейсе программирования.

2.1 Расширение языка C

Вводимые в CUDA расширения языка C состоят из:

- спецификаторов функций, определяющих откуда вызывать и где выполнять функцию: __device__, __global__ и __host__.
- спецификаторов переменных, задающих тип памяти, используемый для данных переменных: __device__, __shared__ и __constant__.
- встроенных векторных типов;
- встроенных переменных для задания размера сетки / блока и индексов блока/нити;
- директив вызова ядра, задающих иерархию нитей: kernel<<<GridDim,BlockDim>>>()

2.1.1 Спецификаторы функций

В CUDA используются следующие спецификаторы функций (Табл. 1).

Таблица 1. Спецификаторы функций в CUDA

Спецификатор	Функция выполняется на	Функция вызывается из
__device__	device (GPU)	device (GPU)
__global__	device (GPU)	host (CPU)
__host__	host (CPU)	host (CPU)

Спецификатор ___ global____ обозначает ядро, и соответствующая функция должна возвращать значение типа void. При вызове ядра нужно обязательно указывать конфигурацию вызова.

На функции, выполняемые на GPU (спецификаторы __device__ и __global__), накладываются следующие ограничения:

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

Адрес функции __device__нельзя использовать при указании на функцию (а __global__ можно).

Спецификаторы __host__ и __device__ могут быть использованы вместе. Это означает, что соответствующая функция может выполняться как на GPU, так и на CPU.

Спецификаторы __global__ и __host__ не могут быть использованы вместе.

2.1.2 Спецификаторы переменных

Для размещения в памяти GPU переменных используются следующие спецификаторы: __device__, __shared__ и __constant__. В таблице 2 приводятся основные характеристики добавленных переменных.

Таблица 2. Спецификаторы переменных

Спецификатор	Область действия	Срок жизни	Память устройства
__device__	сетка	приложение	глобальная
__shared__	блок	блок	разделяемая
__constant__	сетка	приложение	константная

На использование добавленных переменных накладываются следующие ограничения:

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

2.1.3 Встроенные векторные типы

Добавлены 1 / 2 / 3 / 4-мерные векторы из базовых типов: charl, ucharl, char2, uchar2, char3, uchar3, char4, uchar4; short1, ushortl, short2, ushort2, short3, ushort3, short4, ushort4; inti, uintl, int2, uint2, int3, uint3, int4, uint4;
longl, ulongl, long2, ulong2, long3, ulong3, long4, ulong4; floatl, float2, float3, float4; doublel, double2.

2.2. Ядра

CUDA C расширяет C, позволяя программисту определять функции C, называемые ядрами, которые при вызове выполняются N раз параллельно N различными потоками CUDA, а не только как обычные функции C.
Ядро определяется с помощью спецификатора объявления __global__, а число потоков CUDA, выполняющих это ядро для данного вызова ядра, определяется с помощью нового синтаксиса конфигурации выполнения <<< ... >>> . Каждому потоку, выполняющему ядро, присваивается уникальный идентификатор потока, доступный внутри ядра через встроенную переменную threadIdx.
В качестве иллюстрации следующий пример кода добавляет два вектора A и B размера N и сохраняет результат в вектор C:

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
```

```
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Здесь каждый из N потоков, которые выполняют VecAdd (), выполняет одно сложение пары элементов векторов.

2.3. Иерархия нитей

Для удобства threadIdx представляет собой трехкомпонентный вектор, так что потоки могут быть идентифицированы с использованием одномерного, двумерного или трехмерного индекса потока, образуя одномерный, двумерный или трехмерный блок потоков. Это обеспечивает естественный способ вызова вычислений через элементы в области, такие как вектор или матрица.

Индекс потока и его идентификатор потока связаны друг с другом простым способом: для одномерного блока они одинаковы; для двумерного блока размера (Dx, Dy) идентификатор потока равен (x + y Dx); для трехмерного блока размера (Dx, Dy, Dz) идентификатор потока (x, y, z) равен (x + y Dx + z Dx Dy).

В качестве примера следующий код складывает две матрицы A и B размера NxN и сохраняет результат в матрицу C:

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Существует ограничение на количество потоков на блок, поскольку ожидается, что все потоки блока будут находиться в одном ядре процессора и должны совместно использовать ресурсы ограниченной памяти этого ядра. На текущих графических процессорах блок потоков может содержать до 1024 потоков.

Однако ядро может быть выполнено несколькими одинаковыми блоками потоков, так что общее число потоков равно числу потоков на каждый блок, умноженное на количество блоков.

Блоки организованы в одномерную, двумерную или трехмерную сетку блоков потоков, как показано на рисунке 6. Количество блоков потоков в сетке обычно определяется размером обрабатываемых данных или количеством процессоров в системе.

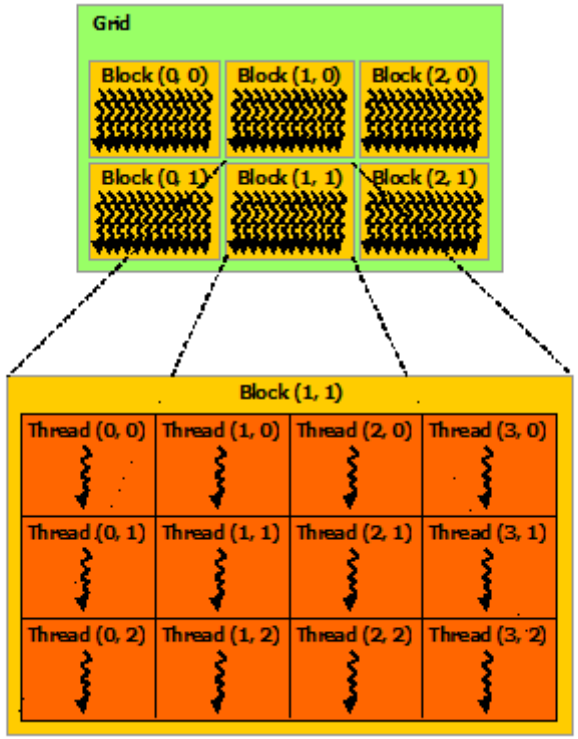

Рисунок 6. Сетка блоков потоков

Количество потоков на блок и количество блоков на каждую сетку, указанные в синтаксисе <<< ... >>>, могут быть типа int или dim3. Двумерные блоки или сетки могут быть указаны как в приведенном выше примере.

Каждый блок в сетке может быть идентифицирован одномерным, двумерным или трехмерным индексом, доступным внутри ядра через встроенную переменную blockIdx. Размер блока потока доступен в ядре через встроенную переменную blockDim.

После модификации предыдущего примера MatAdd () для обработки нескольких блоков, код становится следующим.

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
```

```
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Размер блока нити 16x16 (256 потоков). Сетка создается с достаточным количеством блоков, чтобы иметь один поток на элемент матрицы, как и раньше. Для простоты в этом примере предполагается, что количество потоков на каждую сетку в каждом измерении равномерно делится на количество потоков на блок в этом измерении, хотя это не обязательно.

Блоки потоков должны выполняться независимо: они должны выполняться в любом порядке параллельно или последовательно. Это требование независимости требует, чтобы блоки потоков были запланированы в любом порядке по любому количеству ядер, как показано на рисунке 5, что позволяет программистам писать код, который масштабируется с количеством ядер.

Потоки внутри блока могут взаимодействовать путем совместного использования данных через некоторую общую память и путем синхронизации их выполнения для координации доступа к памяти. Точнее, можно указать точки синхронизации в ядре, вызвав внутреннюю функцию __syncthreads (); __syncthreads () действует как барьер, при котором все потоки в блоке должны ждать до того, как разрешено продолжить. Совместная память дает пример использования разделяемой памяти. В дополнение к __syncthreads () API предоставляет богатый набор примитивов синхронизации потоков.

2.4. Иерархия памяти

Потоки CUDA могут обращаться к данным из нескольких пространств памяти во время их выполнения, как показано на рисунке 7. Каждый поток имеет частную локальную память. Каждый блок потока имеет общую память, видимую для всех потоков блока и с тем же временем жизни, что и блок. Все потоки имеют доступ к одной и той же глобальной памяти.

Есть также два дополнительных пространства для чтения, доступных для всех потоков: пространства постоянной и текстурной памяти. Глобальные, постоянные и текстурные пространства памяти оптимизированы для разных применений памяти. Память текстур также предлагает различные режимы адресации, а также фильтрацию данных для некоторых конкретных форматов данных.

Глобальные, постоянные и текстурные пространства памяти сохраняются при запуске ядра одним и тем же приложением.

2.5. Гетерогенное программирование

Как показано на рисунке 8, модель программирования CUDA предполагает, что потоки CUDA выполняются на физически отдельном устройстве, которое работает как сопроцессор к хосту, на котором запущена программа C. Это так, например, когда ядра выполняются на графическом процессоре, а остальная часть программы C выполняется на процессоре.

Модель программирования CUDA также предполагает, что как хост, так и устройство поддерживают свои собственные пространства памяти в DRAM, называемые памятью хоста и памятью устройства, соответственно. Таким образом, программа управляет глобальными, постоянными и текстурными пространствами памяти, видимыми для ядер через вызовы во время выполнения CUDA). Это включает в себя распределение памяти и освобождение памяти, а также передачу данных между памятью хоста и устройства.

Унифицированная память обеспечивает управляемую память для смены пространства памяти хоста и устройства. Управляемая память доступна из всех процессоров и графических процессоров в системе как единое целочисленное изображение памяти с общим адресным пространством. Эта возможность позволяет переназначить память устройства и может значительно упростить задачу портирования приложений, устраняя необходимость явно отражать данные на хосте и устройстве.

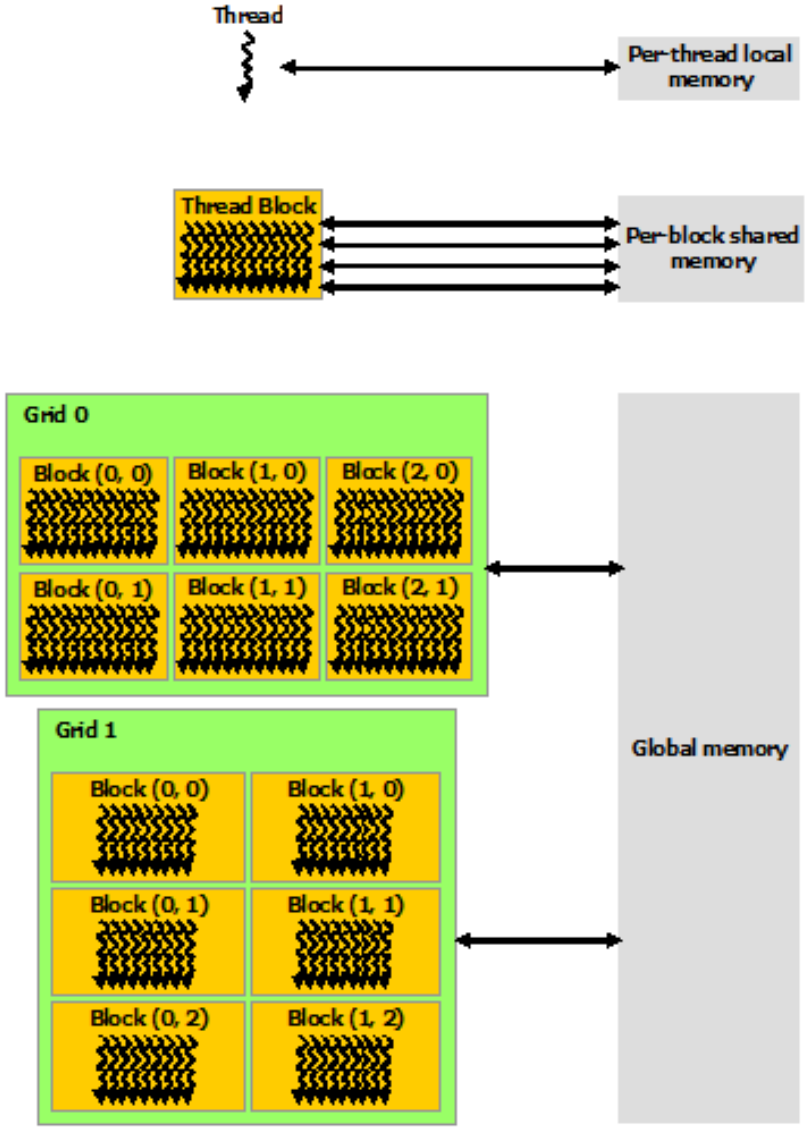

Рисунок 7. Иерархия памяти

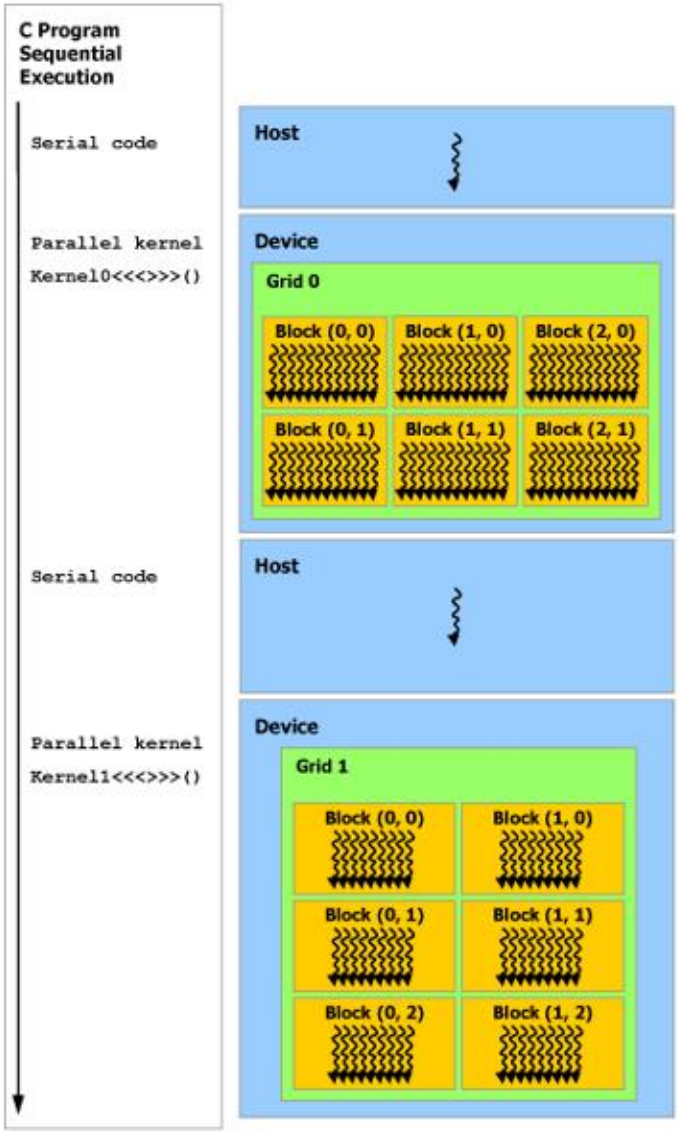

Рисунок 8. Гетерогенное программирование

Примечание. Последовательный код выполняется на хосте, а на устройстве выполняется параллельный код.

2.6. Compute Capability

Compute Capability устройства представлена номером версии, также иногда называемым «версией SM». Этот номер версии идентифицирует функции, поддерживаемые аппаратным обеспечением графического процессора, и используется приложениями во время выполнения, чтобы определить, какие аппаратные функции и / или инструкции доступны на данном графическом процессоре.

Compute Capability содержит основной номер версии X и младший номер версии Y и обозначается X.Y.

Устройства с тем же самым основным номером ревизии имеют одну и ту же основную архитектуру. Основной номер версии - 7 для устройств, основанных на архитектуре Volta, 6 для устройств на основе архитектуры Pascal, 5 для устройств на основе архитектуры Maxwell, 3 для устройств на основе архитектуры Kepler, 2 для устройств на основе архитектуры Fermi, и 1 для устройств на основе архитектуры Tesla.

Минорный номер версии соответствует постепенному улучшению базовой архитектуры, возможно, включая новые функции.

Примечание: Compute Capability конкретного графического процессора не следует путать с версией CUDA (например, CUDA 7.5, CUDA 8, CUDA 9), которая является версией программной платформы CUDA. Платформа CUDA используется разработчиками приложений для создания приложений, работающих на многих поколениях графических архитектур, включая будущие архитектуры графических процессоров, которые еще предстоит изобрести. В то время как новые версии платформы CUDA часто добавляют встроенную поддержку новой архитектуры графического процессора, поддерживая версию Compute Capability этой архитектуры, новые версии платформы CUDA обычно также включают в себя программные функции, которые не зависят от аппаратной генерации.

Архитектуры Tesla и Fermi больше не поддерживаются, начиная с CUDA 7.0 и CUDA 9.0 соответственно.

2.6. Вопросы для самопроверки

- Что означает спецификатор __global__ ?
- Что означает спецификатор __host__ ?
- Что означает спецификатор __device__ ?
- Какие типы памяти существуют в GPU?
- Как вызвать функцию-ядро?
- Что такое нить?
- Что такое блок?
- Как получить номер нити в блоке?
- Как получить глобальный номер нити?

3. Интерфейс программирования

CUDA C предоставляет простой путь для пользователей, знакомых с языком программирования C, чтобы легко создавать программы для выполнения устройством.

Он состоит из минимального набора расширений языка C и библиотеки времени выполнения.

Расширения основного языка позволяют программистам определять ядро как функцию C и использовать какой-то новый синтаксис, чтобы определять размер сетки и блока каждый раз, когда вызывается функция. Любой исходный файл, содержащий некоторые из этих расширений, должен быть скомпилирован с помощью nvcc.

Runtime предоставляет функции C, которые выполняются на хосте для распределения и освобождения памяти устройства, передачи данных между памятью хоста и памятью устройства, управления системами с несколькими устройствами и т. д.

Runtime построено поверх API C более низкого уровня C, API-интерфейса драйвера CUDA, который также доступен для приложения. API-интерфейс драйвера обеспечивает дополнительный уровень управления, предоставляя концепции нижнего уровня, такие как контексты CUDA - аналог хост-процессов для устройства - и модули CUDA - аналог динамически загружаемых библиотек для устройства. Большинство приложений не используют API-интерфейс драйвера, так как они не нуждаются в этом дополнительном уровне управления, и при использовании среды выполнения контекст и управление модулем являются неявными, что приводит к получению более сжатого кода.

3.1 Основы работы с CUDA API

Технология CUDA предоставляет в распоряжение программиста ряд функций, которые могут быть использованы только CPU (CUDA host API). CUDA API для CPU выступает в двух формах: низкого уровня (CUDA driver API) и высокого уровня (CUDA runtime API, реализованный через CUDA driver API). В программе можно использовать только один из них.

Низкоуровневый CUDA driver API дает больше возможностей программисту, а также требует большего объема кода, явных настроек, явной инициализации.

Высокоуровневый CUDA runtime API не требует явной инициализации она происходит автоматически при первом вызове какой-либо его функции. В дальнейшем будем использовать именно CUDA runtime API.

3.1.1 Установка CUDA на компьютер

Для установки CUDA на компьютер необходимо установить:

- Драйвер видеокарты NVIDIA с поддержкой CUDA;
- CUDA Toolkit, CUDA Tools;
- CUDA SDK примеры программных проектов.

Для компиляции программ на CUDA потребуется установленный компилятор с C/C . В качестве такого компилятора в Microsoft Windows может выступать компилятор cl, входящий в состав Microsoft Visual Studio, а также компилятор cygwin, в Linux компилятор gcc.

3.1.2 Компиляция и запуск программ

Для компиляции программ на CUDA существует компилятор nvcc, использующий внешний компилятор для компиляции частей кода, выполняемых на CPU. Функции, составляющие ядро, помещаются в файл с расширением cu, который компилируется с использованием программы nvcc.

Для того чтобы просто откомпилировать программу, состоящую из одного или нескольких файлов, сразу в выполняемый файл, можно воспользоваться следующей командой (для Microsoft Windows):

```
nvcc filel.cu file2.cu file3.cpp -o program.exe
```

Для Linux команда выглядит аналогично, только расширение. exe для выполняемого файла не указывается:

```
nvcc filel.cu file2.cu file3.cpp -o program
```

Для сборки проектов, состоящих из многих файлов, можно также воспользоваться утилитой make (или ее аналогом в Microsoft Windows утилитой nmake).

3.1.3 Обработка ошибок

Каждая функция CUDA runtime API возвращает значение типа cudaError_t. При успешном выполнении функции возвращается значение cudaSuccess, в противном случае возвращается код ошибки. Получить описание ошибки в виде строки можно при помощи функции cudaGetErrorString():

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

Также можно получить код последней ошибки при помощи функции cudaGetLastError():

```
cudaError_t cudaGetLastError();
```

3.1.4 Замеры времени на GPU

Для точного измерения времени выполнения различных операций на GPU можно воспользоваться так называемыми событиями (CUDA events). Событие — это объект типа cudaEvent_t, используемый для обозначения «точки» среди вызовов CUDA. Каждое событие, привязанное к точке, характеризуется тем, пройдена GPU данная точка или нет.

CUDA runtime API обеспечивает точный замер времени, позволяя приложению асинхронно записывать события в любой точке программы, запрашивать наступило ли данное событие, ждать наступления события, а также получать интервал времени в миллисекундах между наступлениями событий.

Ниже приводится простой пример кода, замеряющий время выполнения ядра на GPU.

```
cudaEvent_t start, stop; //init events
float elapsedTime;
cudaEventCreate(festart);//create events
cudaEventCreate(festop);
cudaEventRecord(start, 0);//write event
```

```
//call kernel
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);//
cudaEventElapsedTime(&elapsedTime, start, stop);
cudaEventDestroy(start);//delete events
cudaEventDestroy(stop);
```

3.1.5 Синхронизация

Для синхронизации текущей нити на CPU с GPU используется функция cudaThreadSynchronize (), которая дожидается завершения выполнения всех операций CUDA, ранее вызванных с данной нити CPU.

Функция CPU cudaDeviceSynchronizeO блокирует выполнение кода CPU пока все GPU ядра не выполнены.

Для синхронизации всех нитей в одном и том же блоке используется функция __synctbreads() , которая блокирует вызывающие нити до тех пор, пока все нити блока не войдут в эту функцию.

При помощи этой функции можно организовать барьеры внутри ядра, гарантирующие, что если хотя бы одна нить прошла такой барьер, то не осталось ни одной за барьером (не прошедшей его). Такая синхронизация называется барьерной.

3.2. Компиляция с NVCC

Ядра могут быть записаны с использованием архитектуры набора инструкций CUDA, называемой PTX. Однако, как правило, более эффективно использовать высокоуровневый язык программирования, такой как C. В обоих случаях ядра должны быть скомпилированы в двоичный код nvcc для выполнения на устройстве.

nvcc - это драйвер компилятора, который упрощает процесс компиляции кода C или PTX: он предоставляет простые и знакомые параметры командной строки и выполняет их, вызывая инструменты, которые реализуют различные стадии компиляции.

3.2.1. Рабочий процесс компиляции

3.2.1.1. Автономная компиляция

Исходные файлы, скомпилированные с помощью nvcc, могут включать в себя сочетание хост-кода (то есть кода, выполняемого на хосте) и кода устройства (то есть кода, который выполняется на устройстве). Основной рабочий процесс nvcc состоит в разделении кода устройства с хост-кода, а затем:

- компиляция кода устройства в форму сборки (код PTX) и / или двоичную форму
- и модификация кода хоста, заменив синтаксис <<< ... >>>, введенный в ядрах с помощью необходимых функций функции выполнения CUDA C для загрузки и запуска каждого скомпилированного ядра из кода PTX.

Измененный код хоста выводится либо как код C, который остается компилировать с помощью другого инструмента, либо как объектный код напрямую, позволяя nvcc вызывать компилятор хоста во время последнего этапа компиляции.

Приложения могут затем:

- Либо ссылаться на скомпилированный код хоста (это самый распространенный случай)
- Или использовать API-интерфейс драйвера CUDA для загрузки и выполнения кода PTX.

3.2.1.2. JIT Компиляция

Любой код PTX, загружаемый приложением во время выполнения, далее компилируется двоичным кодом драйвером устройства. Это называется JIT компиляцией. JIT компиляция увеличивает время загрузки приложений, но позволяет приложению получить прирост производительности от любых улучшений компилятора, поступающих с каждым новым драйвером устройства.

Когда драйвер устройства компилирует некоторый код PTX для некоторого приложения, он автоматически кэширует копию сгенерированного двоичного кода, чтобы избежать повторения компиляции в последующих вызовах приложения. Кэш, называемый вычислительным кэшем, автоматически недействителен при обновлении драйвера устройства, так что приложения могут получить прирост производительности из-за нового JIT компилятора.

3.3 CUDA C Runtime

3.3.1. Инициализация

Нет явной функции инициализации для среды выполнения; он инициализирует первый раз, когда вызывается функция времени выполнения.

Во время инициализации среда выполнения создает контекст CUDA для каждого устройства в системе. Этот контекст является основным контекстом для этого устройства и является общим для всех потоков хоста приложения. В рамках создания этого контекста код устройства компилируется по необходимости и загружается в память устройства. Все это происходит под капотом, и среда выполнения не раскрывает основной контекст приложения.

Когда хост-поток вызывает cudaDeviceReset (), это разрушает основной контекст устройства, на котором в настоящий момент работает хост-поток. Следующий вызов функции времени выполнения, созданный любым потоком хоста, который имеет это устройство как текущий, создаст новый первичный контекст для этого устройства.

3.3.2. Память устройств

Как упоминалось, модель программирования CUDA предполагает систему, состоящую из хоста и устройства, каждая со своей собственной отдельной памятью. Ядра работают вне памяти устройства, поэтому среда выполнения предоставляет функции для распределения, освобождения и копирования памяти устройства, а также для передачи данных между памятью хоста и памятью устройства.

Память устройств может быть выделена либо как линейная память, либо как массивы CUDA.

Массивы CUDA - это непрозрачные макеты памяти, оптимизированные для работы с текстурами.

Линейная память существует на устройстве в 40-битном адресном пространстве, поэтому отдельно выделенные объекты могут ссылаться друг на друга посредством указателей, например, в двоичном дереве.

Линейная память обычно выделяется с помощью cudaMalloc () и освобождается с использованием cudaFree (), а передача данных между памятью хоста и памятью устройства обычно выполняется с использованием cudaMemcpy ().

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

cudaError_t cudaFree (void * devPtr);

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

Функции cudaMalloc и cudaFree являются стандартными функциями выделения и освобождения глобальной памяти. Поскольку для эффективного доступа к глобальной памяти важным требованием является выравнивание данных в памяти, то для выделения памяти под двухмерные массивы лучше использовать функцию cudaMallocPitch, которая при выделении памяти может увеличить объем памяти под каждую строку, чтобы гарантировать выравнивание всех строк. При этом дополнительный объем памяти в байтах, служащий для выравнивания строки, возвращается через параметр pitch.

В образце кода сложения вектора ячеек векторы необходимо скопировать из памяти хоста в память устройства:

```
// Device code
__global__ void VecAdd(float* A, float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}

// Host code
int main()
{
    int N = ...;
    size_t size = N * sizeof(float);

    // Allocate input vectors h_A and h_B in host memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);

    // Initialize input vectors
    ...

    // Allocate vectors in device memory
    float* d_A;
    cudaMalloc(&d_A, size);
    float* d_B;
```

```
    cudaMalloc(&d_B, size);
    float* d_C;
    cudaMalloc(&d_C, size);

    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid =
            (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free host memory
    ...
}
```

Линейную память также можно выделить через cudaMallocPitch () и cudaMalloc3D (). Эти функции рекомендуются для размещения двумерных или трехмерных массивов, так как он гарантирует, что распределение надлежащим образом дополняется, чтобы соответствовать требованиям к выравниванию, поэтому обеспечивает максимальную производительность при доступе к адресам строк или выполнению копирования между 2D-массивами и другими регионами памяти (используя функции cudaMemcpy2D () и cudaMemcpy3D ()). Возвращаемый шаг (или шаг) должен использоваться для доступа к элементам массива. В следующем примере кода выделяется двумерный массив значений с плавающей запятой и показано, как перемещаться по элементам массива в коде устройства:

```
// Host code
int width = 64, height = 64;
float* devPtr;
size_t pitch;
cudaMallocPitch(&devPtr, &pitch,
```

```
                width * sizeof(float), height);
MyKernel<<<100, 512>>>(devPtr, pitch, width, height);

// Device code
__global__ void MyKernel(float* devPtr,
                         size_t pitch, int width, int height)
{
    for (int r = 0; r < height; ++r) {
        float* row = (float*)((char*)devPtr + r * pitch);
        for (int c = 0; c < width; ++c) {
            float element = row[c];
        }
    }
}
```

В следующем примере кода выделяется 3D-массив значений с плавающей запятой и показано, как перемещаться по элементам массива в коде устройства:

```
// Host code
int width = 64, height = 64, depth = 64;
cudaExtent extent = make_cudaExtent(width * sizeof(float),
                                    height, depth);
cudaPitchedPtr devPitchedPtr;
cudaMalloc3D(&devPitchedPtr, extent);
MyKernel<<<100, 512>>>(devPitchedPtr, width, height, depth);

// Device code
__global__ void MyKernel(cudaPitchedPtr devPitchedPtr,
                         int width, int height, int depth)
{
    char* devPtr = devPitchedPtr.ptr;
    size_t pitch = devPitchedPtr.pitch;
    size_t slicePitch = pitch * height;
    for (int z = 0; z < depth; ++z) {
        char* slice = devPtr + z * slicePitch;
        for (int y = 0; y < height; ++y) {
            float* row = (float*)(slice + y * pitch);
            for (int x = 0; x < width; ++x) {
                float element = row[x];
```

```
            }
        }
    }
}
```

Следующий пример кода иллюстрирует различные способы доступа к глобальным переменным через API среды выполнения:

```
__constant__ float constData[256];
float data[256];
cudaMemcpyToSymbol(constData, data, sizeof(data));
cudaMemcpyFromSymbol(data, constData, sizeof(data));

__device__ float devData;
float value = 3.14f;
cudaMemcpyToSymbol(devData, &value, sizeof(float));

__device__ float* devPointer;
float* ptr;
cudaMalloc(&ptr, 256 * sizeof(float));
cudaMemcpyToSymbol(devPointer, &ptr, sizeof(ptr));
```

cudaGetSymbolAddress () используется для извлечения адреса, указывающего на память, выделенную для переменной, объявленной в глобальной памяти. Размер выделенной памяти получается через cudaGetSymbolSize ().

3.3.3. Разделяемая память

Разделяемая память выделяется с использованием спецификатора __shared__.

Ожидается, что общая память будет намного быстрее, чем глобальная память. Поэтому любая возможность заменить глобальные обращения к памяти с помощью доступа к общей памяти должна быть использована, как показано в следующем примере умножения матрицы.

Следующий пример кода представляет собой прямую реализацию матричного умножения, которая не использует преимущества общей памяти. Каждый поток считывает одну строку A и один столбец B и вычисляет соответствующий элемент C, как показано на рисунке 9. Следовательно, A считывает B.width раз из глобальной памяти, а B читается A.height раз.

```
// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.width + col)
typedef struct {
    int width;
    int height;
    float* elements;
} Matrix;

// Thread block size
```

```
#define BLOCK_SIZE 16

// Forward declaration of the matrix multiplication kernel
__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);

// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    // Load A and B to device memory
    Matrix d_A;
    d_A.width = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc(&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size,
               cudaMemcpyHostToDevice);
    Matrix d_B;
    d_B.width = B.width; d_B.height = B.height;
    size = B.width * B.height * sizeof(float);
    cudaMalloc(&d_B.elements, size);
    cudaMemcpy(d_B.elements, B.elements, size,
               cudaMemcpyHostToDevice);

    // Allocate C in device memory
    Matrix d_C;
    d_C.width = C.width; d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    cudaMalloc(&d_C.elements, size);

    // Invoke kernel
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

    // Read C from device memory
    cudaMemcpy(C.elements, Cd.elements, size,
               cudaMemcpyDeviceToHost);
```

```
    // Free device memory
    cudaFree(d_A.elements);
    cudaFree(d_B.elements);
    cudaFree(d_C.elements);
}

// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Each thread computes one element of C
    // by accumulating results into Cvalue
    float Cvalue = 0;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    for (int e = 0; e < A.width; ++e)
        Cvalue += A.elements[row * A.width + e]
                * B.elements[e * B.width + col];
    C.elements[row * C.width + col] = Cvalue;
}
```

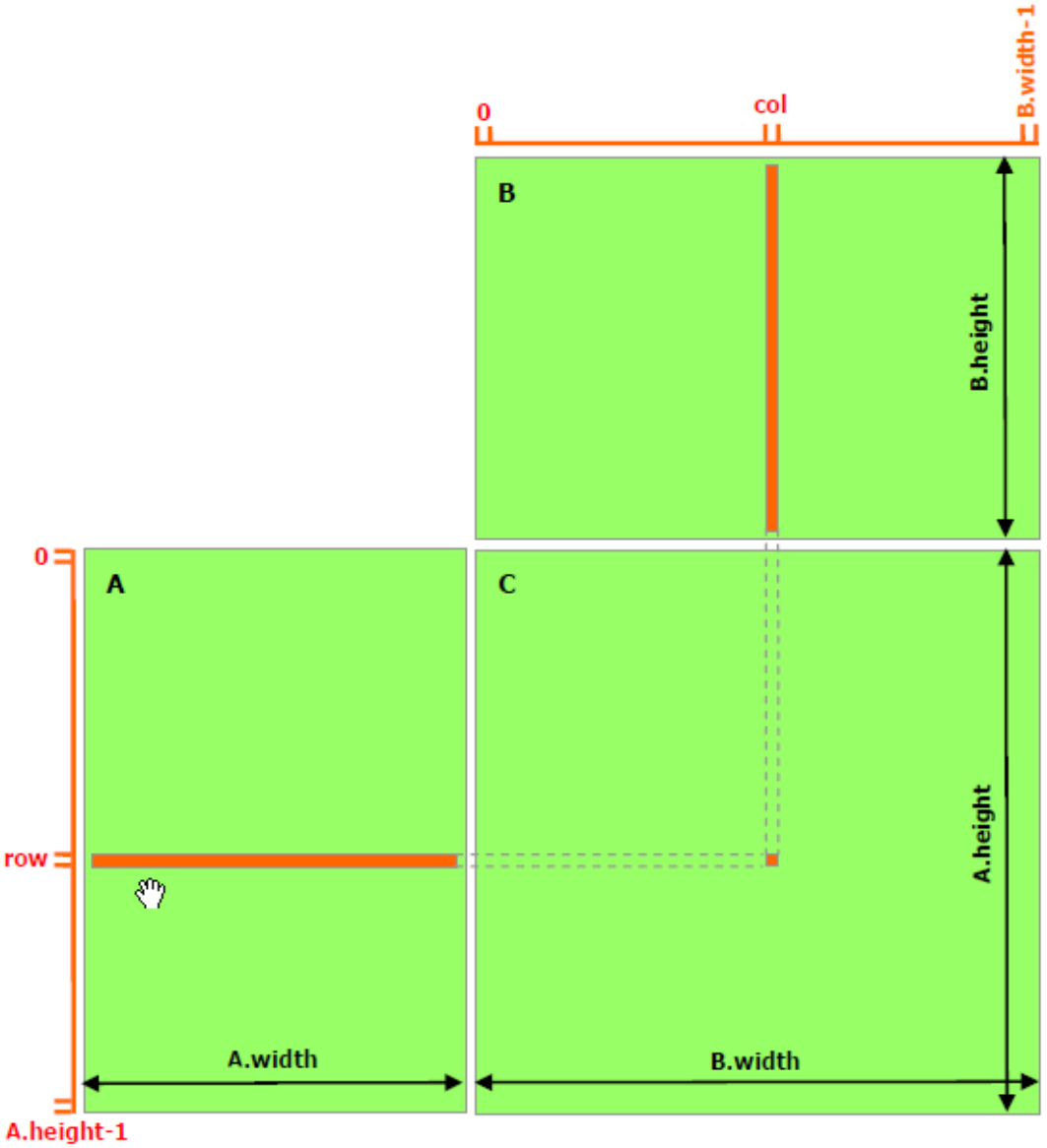

Рисунок 9. Умножение матриц без общей памяти

Следующий пример кода представляет собой реализацию матричного умножения, которая использует преимущества общей памяти. В этой реализации каждый блок потока отвечает за вычисление одной квадратной подматрицы Csub C, и каждый поток в блоке отвечает за вычисление одного элемента Csub. Как показано на рисунке 10, Csub равен произведению двух прямоугольных матриц: подматрицы A размерности (A.width, block_size), которая имеет те же индексы строк, что и Csub, и подматрицу B размерности (block_size, A.width), который имеет те же индексы столбцов, что и Csub. Чтобы вписаться в ресурсы устройства, эти две прямоугольные матрицы делятся на столько квадратных матриц размерности block_size, сколько необходимо, и Csub вычисляется как сумма произведений этих квадратных матриц. Каждый из этих произведений выполняется путем первой загрузки двух соответствующих квадратных матриц из глобальной памяти в общую память с одним потоком, загружающим один элемент каждой матрицы, а затем путем вычисления каждого потока одного элемента произведений. Каждый поток накапливает результат каждого из этих произведений в регистр и после этого записывает результат в глобальную память.

Таким образом, мы используем быструю разделяемую память и сохраняем большую пропускную способность глобальной памяти, так как A только считывает (B.width / block_size) раз из глобальной памяти, а B читается (A.height / block_size) раз .

```
// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.stride + col)
typedef struct {
    int width;
    int height;
    int stride;
    float* elements;
} Matrix;

// Get a matrix element
__device__ float GetElement(const Matrix A, int row, int col)
{
    return A.elements[row * A.stride + col];
}

// Set a matrix element
__device__ void SetElement(Matrix A, int row, int col,
                           float value)
{
    A.elements[row * A.stride + col] = value;
}

// Get the BLOCK_SIZExBLOCK_SIZE sub-matrix Asub of A that is
// located col sub-matrices to the right and row sub-matrices down
```

```
// from the upper-left corner of A
 __device__ Matrix GetSubMatrix(Matrix A, int row, int col)
{
    Matrix Asub;
    Asub.width    = BLOCK_SIZE;
    Asub.height   = BLOCK_SIZE;
    Asub.stride   = A.stride;
    Asub.elements = &A.elements[A.stride * BLOCK_SIZE * row
                                         + BLOCK_SIZE * col];
    return Asub;
}

// Thread block size
#define BLOCK_SIZE 16

// Forward declaration of the matrix multiplication kernel
__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);

// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    // Load A and B to device memory
    Matrix d_A;
    d_A.width = d_A.stride = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc(&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size,
               cudaMemcpyHostToDevice);
    Matrix d_B;
    d_B.width = d_B.stride = B.width; d_B.height = B.height;
    size = B.width * B.height * sizeof(float);
    cudaMalloc(&d_B.elements, size);
    cudaMemcpy(d_B.elements, B.elements, size,
    cudaMemcpyHostToDevice);

    // Allocate C in device memory
    Matrix d_C;
```

```
    d_C.width = d_C.stride = C.width; d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    cudaMalloc(&d_C.elements, size);

    // Invoke kernel
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

    // Read C from device memory
    cudaMemcpy(C.elements, d_C.elements, size,
               cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A.elements);
    cudaFree(d_B.elements);
    cudaFree(d_C.elements);
}

// Matrix multiplication kernel called by MatMul()
 __global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Block row and column
    int blockRow = blockIdx.y;
    int blockCol = blockIdx.x;

    // Each thread block computes one sub-matrix Csub of C
    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);

    // Each thread computes one element of Csub
    // by accumulating results into Cvalue
    float Cvalue = 0;

    // Thread row and column within Csub
    int row = threadIdx.y;
    int col = threadIdx.x;

    // Loop over all the sub-matrices of A and B that are
```

```
    // required to compute Csub
    // Multiply each pair of sub-matrices together
    // and accumulate the results
    for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {

        // Get sub-matrix Asub of A
        Matrix Asub = GetSubMatrix(A, blockRow, m);

        // Get sub-matrix Bsub of B
        Matrix Bsub = GetSubMatrix(B, m, blockCol);

        // Shared memory used to store Asub and Bsub respectively
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

        // Load Asub and Bsub from device memory to shared memory
        // Each thread loads one element of each sub-matrix
        As[row][col] = GetElement(Asub, row, col);
        Bs[row][col] = GetElement(Bsub, row, col);

        // Synchronize to make sure the sub-matrices are loaded
        // before starting the computation
        __syncthreads();
        // Multiply Asub and Bsub together
        for (int e = 0; e < BLOCK_SIZE; ++e)
            Cvalue += As[row][e] * Bs[e][col];

        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();
    }

    // Write Csub to device memory
    // Each thread writes one element
    SetElement(Csub, row, col, Cvalue);
}
```

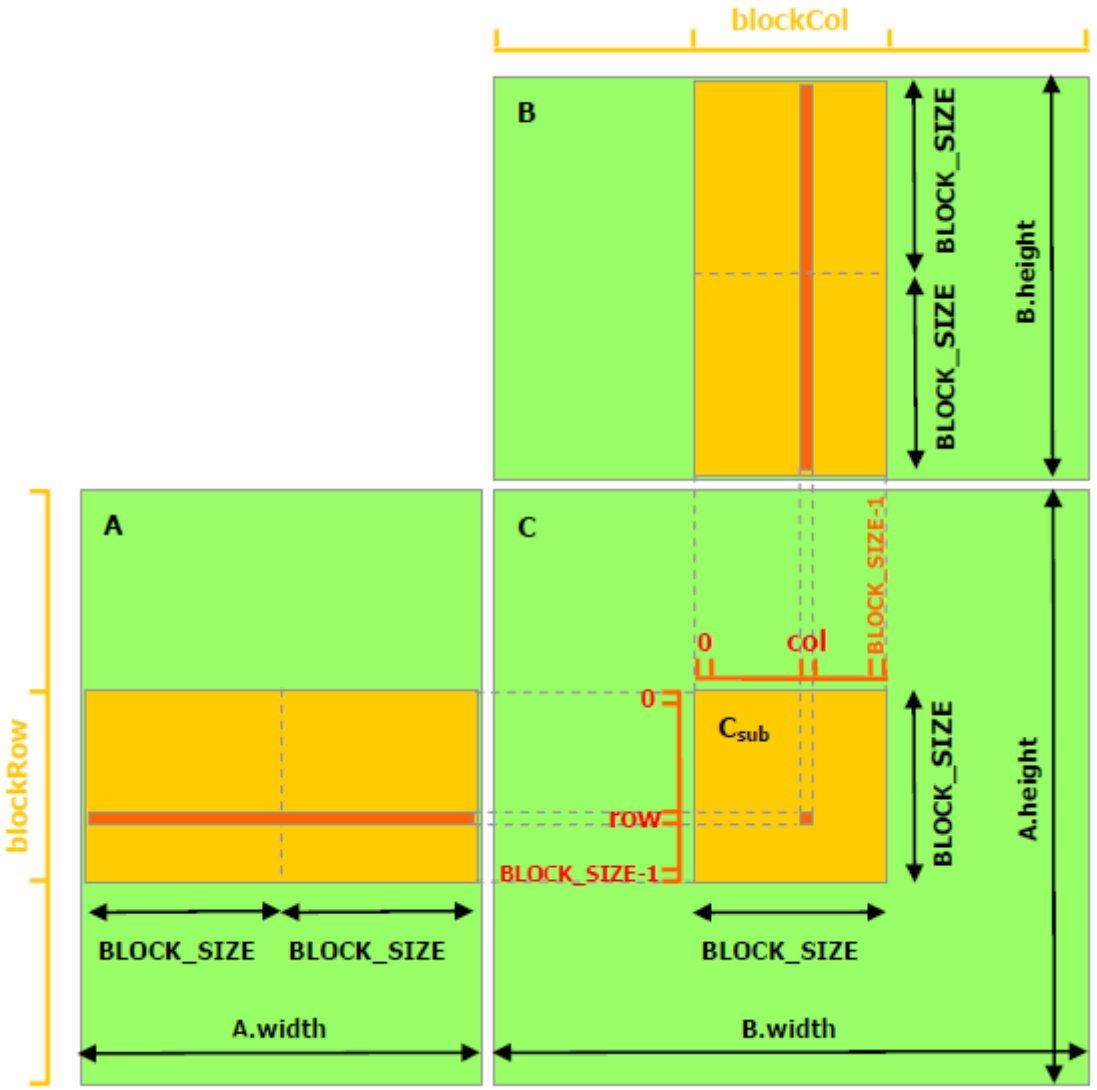

Рисунок 10. Умножение матриц с разделяемой памятью

3.3.4. Закреплённая хост-память

Среда выполнения предоставляет функции, позволяющие использовать память хоста с записями (также известную как закрепленная) (в отличие от обычной доступной памяти хоста, выделенной malloc ()):

- cudaHostAlloc () и cudaFreeHost () выделяют и освобождают память хоста на странице;
- cudaHostRegister () закрепляет область памяти, выделенную malloc ().

Использование памяти с закреплением страниц имеет несколько преимуществ:

Копирование между памятью с закреплением страниц и памятью устройства могут выполняться одновременно с выполнением ядра для некоторых устройств.

На некоторых устройствах память узла с записями страниц может быть отображена в адресное пространство устройства, что исключает необходимость ее скопировать в память устройства или из памяти устройства.

В системах с фронтальной шиной пропускная способность между памятью хоста и памятью устройства выше, если память хоста выделяется как закреплением страниц.

Выделение большого количества закреплённой памяти снижает производительность системы

3.3.4.1. Портативная память

Блок памяти с закреплёнными страницами может использоваться совместно с любым устройством в системе, но по умолчанию преимущества использования закрепления страниц, описанные выше, являются доступными для устройства, которое было текущим, когда был выделен блок (и со всеми устройствами, использующими одно и то же унифицированное адресное пространство, если таковые имеются). Чтобы сделать эти преимущества доступными для всех устройств, блок необходимо выделить, передав флаг cudaHostAllocPortable в cudaHostAlloc () или закрепив страницы, передав флаг cudaHostRegisterPortable в cudaHostRegister ().

3.3.4.3. Отображение памяти

Блок памяти с закреплением страниц также можно отобразить в адресное пространство устройства, передав флаг cudaHostAllocMapped в cudaHostAlloc () или передав флаг cudaHostRegisterMapped в cudaHostRegister (). Таким образом, такой блок имеет, как правило, два адреса: один в памяти хоста, который возвращается cudaHostAlloc () или malloc (), и один в памяти устройства, который может быть получен с помощью cudaHostGetDevicePointer (), а затем используется для доступа к блоку из ядра. Единственное исключение - для указателей, выделенных cudaHostAlloc ().

Доступ к памяти хоста непосредственно из ядра имеет несколько преимуществ:

- Нет необходимости выделять блок в памяти устройства и копировать данные между этим блоком и блоком в память хоста; передача данных неявно выполняется по мере необходимости ядром;
- Нет необходимости использовать потоки для перекрытия передачи данных с выполнением ядра; передача данных, инициированная ядром, автоматически перекрывается с выполнением ядра.
- Поскольку отображённая память с записями страниц используется совместно с хостом и устройством, приложение должно синхронизировать обращения к памяти с помощью потоков или событий, чтобы избежать ошибок при записи в память.

3.3.5. Асинхронное параллельное выполнение

CUDA предоставляет следующие операции в качестве самостоятельных задач, которые могут работать одновременно друг с другом:

- Вычисление на хосте;
- Вычисление на устройстве;
- Копирование памяти с хоста на устройство;
- Копирование памяти с устройства на хост;
- Копирование памяти с устройства на устройство;

3.3.5.1. Параллельное выполнение между хостом и устройством

Параллельное выполнение на хосте облегчается с помощью асинхронных функций, которые возвращают управление хост-потоку до того, как устройство выполнит запрошенную задачу. Используя асинхронные вызовы, многие операции с устройством могут быть поставлены в очередь вместе, чтобы их выполнил драйвер CUDA, когда доступны соответствующие ресурсы устройства. Это избавляет основной поток от ответственности за управление устройством, оставляя его свободным для других задач. Следующие операции устройства являются асинхронными по отношению к хосту:

- Запуск ядра;
- Копирование в память устройства;
- Копирование памяти с хоста на устройство блока памяти объемом не более 64 КБ;
- Копирование памяти, выполняемые функциями, которые имеют суффикс Async;

Программисты могут глобально отключить асинхронность запуска ядра для всех приложений CUDA, работающих в системе, установив переменную среды CUDA_LAUNCH_BLOCKING в 1. Эта функция предоставляется только для целей отладки и не должна использоваться в качестве способа надежного запуска производственного программного обеспечения.

3.2.5.2. Параллельное выполнение ядра

Некоторые устройства могут одновременно выполнять несколько ядер. Приложения могут запрашивать эту возможность, проверяя свойство устройства concurrentKernels, которое равно 1 для устройств, которые её поддерживают.

Ядро из одного контекста CUDA не может выполняться одновременно с ядром из другого контекста CUDA.

Ядра, которые используют много текстур или большую часть локальной памяти, с меньшей вероятностью выполняются одновременно с другими ядрами.

3.3.5.3. Перекрытие передачи данных и выполнение ядра

Некоторые устройства могут выполнять асинхронное копирование памяти на или с графического процессора одновременно с выполнением ядра. Приложения могут запрашивать эту возможность, проверяя свойство устройства asyncEngineCount, которое больше нуля для устройств, которые его поддерживают. Если в копировании задействована память хоста, она должна быть закреплена.

3.3.5.4. Потоки(Streams)

Приложения управляют параллельными операциями, описанными выше, через потоки. Поток представляет собой последовательность команд (возможно, выпущенных разными хост-потоками), которые выполняются по порядку. Различные потоки, с другой стороны, могут выполнять свои команды не по порядку относительно друг друга или одновременно; это поведение не гарантируется.

3.3.5.5.1. Создание и уничтожение

Поток определяется созданием объекта потока и указанием его как параметра потока для последовательности запуска ядра и копированием памяти с хоста на устройство. В следующем примере кода создается два потока и выделяется массив hostPtr из float в закреплённой странице.

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
float* hostPtr;
cudaMallocHost(&hostPtr, 2 * size);
```

Каждый из этих потоков определяется как последовательность одной операции копирования памяти с хоста на устройство, один запуск ядра и одна операция копирования памяти с устройства на хост:

```
for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);
    MyKernel <<<100, 512, 0, stream[i]>>>
          (outputDevPtr + i * size, inputDevPtr + i * size, size);
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
                    size, cudaMemcpyDeviceToHost, stream[i]);
}
```

Каждый поток копирует свою часть входного массива hostPtr в массив inputDevPtr в памяти устройства, обрабатывает inputDevPtr на устройстве, вызывая MyKernel (), и копирует результат outputDevPtr обратно в ту же часть hostPtr.

Потоки освобождаются, вызывая cudaStreamDestroy ().

```
for (int i = 0; i < 2; ++i)
    cudaStreamDestroy(stream[i]);
```

3.3.5.5.2. Явная синхронизация

Существуют различные способы явной синхронизации потоков друг с другом.

- cudaDeviceSynchronize () ожидает завершения всех предыдущих команд во всех потоках всех потоков хоста.
- cudaStreamSynchronize () принимает поток как параметр и ждет, пока все предыдущие команды в данном потоке не будут завершены. Он может использоваться для синхронизации хоста с определенным потоком, позволяя другим потокам продолжать выполнение на устройстве.
- cudaStreamWaitEvent () принимает поток и событие в качестве параметров и выполнчет все команды, добавленные в данный поток после вызова cudaStreamWaitEvent (), задерживает их выполнение до тех пор, пока данное событие не завершится. Поток может быть 0, и в этом случае все команды, добавленные в любой поток после вызова cudaStreamWaitEvent (), ждут события.
- cudaStreamQuery () предоставляет приложениям способ узнать, завершены ли все предыдущие команды в потоке.

3.3.5.5.3. Неявная синхронизация

Две команды из разных потоков не могут выполняться одновременно, если какая-либо из следующих операций выдается между ними потоком хоста:

- выделение памяти с закреплением страницы
- распределение памяти устройства
- копирование памяти между двумя адресами в одну и ту же память устройства
- любая команда CUDA в поток NULL

3.3.5.6. События

Время выполнения также обеспечивает возможность тщательного отслеживания прогресса устройства, позволяя приложению асинхронно записывать события в любой момент программы и запрашивать, когда эти события будут завершены. Событие завершено, когда все задачи (или, необязательно, все команды в данном потоке - перед событием завершены). События в потоке NULL завершаются после завершения всех предыдущих задач и команд во всех потоках.

3.3.5.6.1. Создание и уничтожение

Следующий пример кода создает два события:

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
```

Они уничтожаются таким образом:

```
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

3.3.5.6.2. Измерение времени

События, могут использоваться для измерения времени работы кода следующим образом:

```
cudaEventRecord(start, 0);
for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(inputDev + i * size, inputHost + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);
    MyKernel<<<100, 512, 0, stream[i]>>>
               (outputDev + i * size, inputDev + i * size, size);
    cudaMemcpyAsync(outputHost + i * size, outputDev + i * size,
                    size, cudaMemcpyDeviceToHost, stream[i]);
}
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
```

3.3.6. Проверка ошибок

Все функции времени выполнения возвращают код ошибки, но для асинхронных функций этот код ошибки не может сообщать о каких-либо асинхронных ошибках, которые могут возникнуть на устройстве, поскольку функция возвращается до того, как устройство завершило задачу; код ошибки сообщает только ошибки, которые происходят на хосте перед выполнением задачи, как правило, связанные с проверкой параметров; если возникает асинхронная ошибка, о ней будет сообщено с помощью некоторого последующего вызова функции времени исполнения.

Таким образом, единственный способ проверить на асинхронные ошибки сразу после некоторого вызова асинхронной функции, выполнить синхронизацию сразу после вызова, вызывая cudaDeviceSynchronize и проверяя код ошибки, возвращаемый cudaDeviceSynchronize ( ).

Среда выполнения поддерживает переменную ошибки для каждого потока хоста, которая инициализируется cudaSuccess и перезаписывается кодом ошибки каждый раз при возникновении ошибки (будь то ошибка проверки параметров или асинхронная ошибка). cudaPeekAtLastError () возвращает эту переменную. cudaGetLastError () возвращает эту переменную и сбрасывает ее до cudaSuccess.

Запуск ядра не возвращает код ошибки, поэтому cudaPeekAtLastError () или cudaGetLastError () необходимо вызывать сразу после запуска ядра для извлечения любых ошибок перед запуском. Чтобы гарантировать, что любая ошибка, возвращаемая cudaPeekAtLastError () или cudaGetLastError (), не возникает из вызовов до запуска ядра, необходимо убедиться, что переменная переменной времени выполнения установлена в cudaSuccess непосредственно перед запуском ядра, например, путем вызова cudaGetLastError () перед запуском ядра. Ядро запускается асинхронно, поэтому для проверки асинхронных ошибок приложение должно синхронизировать между запуском ядра и вызовом cudaPeekAtLastError () или cudaGetLastError ().

Обратите внимание: cudaErrorNotReady, который может быть возвращен cudaStreamQuery () и cudaEventQuery (), не считается ошибкой и поэтому не сообщается cudaPeekAtLastError () или cudaGetLastError ().

3.3.7 Текстурная память

CUDA поддерживает подмножество оборудования, которое графический процессор использует для графики для доступа к текстурной памяти. Чтение данных из текстурной памяти вместо глобальной памяти может обеспечить некоторое повышение производительности.

Существует два разных API для доступа к текстурной и поверхностной памяти:

- API-интерфейс текстуры, поддерживаемый на всех устройствах,
- API-интерфейс текстурного объекта, который поддерживается только на устройствах вычислительной способности 3.x.

3.2.7.1. API объектов текстуры

Объект текстуры создается с помощью cudaCreateTextureObject () из описания ресурса типа struct cudaResourceDesc, который определяет текстуру и описание текстуры:

```
struct cudaTextureDesc
{
    enum cudaTextureAddressMode addressMode[3];
    enum cudaTextureFilterMode  filterMode;
    enum cudaTextureReadMode    readMode;
    int                         sRGB;
    int                         normalizedCoords;
    unsigned int                maxAnisotropy;
    enum cudaTextureFilterMode  mipmapFilterMode;
    float                       mipmapLevelBias;
    float                       minMipmapLevelClamp;
    float                       maxMipmapLevelClamp;
};
```

- addressMode указывает режим адресации;
- filterMode указывает режим фильтра;
- readMode указывает режим чтения;
- normalizedCoords указывает, нормализуются ли координаты текстуры или нет;

Следующий пример кода применяет к текстуре простое преобразование:

```
// Simple transformation kernel
__global__ void transformKernel(float* output,
                                cudaTextureObject_t texObj,
```

```
                                int width, int height,
                                float theta)
{
    // Calculate normalized texture coordinates
    unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;

    float u = x / (float)width;
    float v = y / (float)height;

    // Transform coordinates
    u -= 0.5f;
    v -= 0.5f;
    float tu = u * cosf(theta) - v * sinf(theta) + 0.5f;
    float tv = v * cosf(theta) + u * sinf(theta) + 0.5f;

    // Read from texture and write to global memory
    output[y * width + x] = tex2D<float>(texObj, tu, tv);
}
// Host code
int main()
{
    // Allocate CUDA array in device memory
    cudaChannelFormatDesc channelDesc =
                cudaCreateChannelDesc(32, 0, 0, 0,
                                      cudaChannelFormatKindFloat);
    cudaArray* cuArray;
    cudaMallocArray(&cuArray, &channelDesc, width, height);

    // Copy to device memory some data located at address h_data
    // in host memory
    cudaMemcpyToArray(cuArray, 0, 0, h_data, size,
                      cudaMemcpyHostToDevice);

    // Specify texture
    struct cudaResourceDesc resDesc;
    memset(&resDesc, 0, sizeof(resDesc));
    resDesc.resType = cudaResourceTypeArray;
```

```
    resDesc.res.array.array = cuArray;

    // Specify texture object parameters
    struct cudaTextureDesc texDesc;
    memset(&texDesc, 0, sizeof(texDesc));
    texDesc.addressMode[0]   = cudaAddressModeWrap;
    texDesc.addressMode[1]   = cudaAddressModeWrap;
    texDesc.filterMode       = cudaFilterModeLinear;
    texDesc.readMode         = cudaReadModeElementType;
    texDesc.normalizedCoords = 1;

    // Create texture object
    cudaTextureObject_t texObj = 0;
    cudaCreateTextureObject(&texObj, &resDesc, &texDesc, NULL);

    // Allocate result of transformation in device memory
    float* output;
    cudaMalloc(&output, width * height * sizeof(float));

    // Invoke kernel
    dim3 dimBlock(16, 16);
    dim3 dimGrid((width  + dimBlock.x - 1) / dimBlock.x,
                 (height + dimBlock.y - 1) / dimBlock.y);
    transformKernel<<<dimGrid, dimBlock>>>(output,
                                           texObj, width, height,
                                           angle);

    // Destroy texture object
    cudaDestroyTextureObject(texObj);

    // Free device memory
    cudaFreeArray(cuArray);
    cudaFree(output);

    return 0;
}
```

3.3.7.2. Ссылки на текстуру

Некоторые атрибуты ссылки на текстуру являются неизменными и должны быть известны во время компиляции; они указываются при объявлении ссылки на текстуру. Ссылка на текстуру объявляется в области файлов как переменная текстуры типа:

```
texture<DataType, Type, ReadMode> texRef;
```

где:

DataType определяет тип texel;

Type задает тип ссылки на текстуру и равен cudaTextureType1D, cudaTextureType2D или cudaTextureType3D для одномерной, двумерной или трехмерной текстуры соответственно или cudaTextureType1DLayered или cudaTextureType2DLayered для одномерной или двумерной слоистой текстуры соответственно; Type - необязательный аргумент, по умолчанию - cudaTextureType1D;

ReadMode указывает режим чтения; это необязательный аргумент, по умолчанию используется cudaReadModeElementType.

Ссылка на текстуру может быть объявлена только как статическая глобальная переменная и не может быть передана в качестве аргумента функции.

Другие атрибуты ссылки на текстуру являются изменяемыми и могут быть изменены во время выполнения через среду выполнения хоста.

```
struct textureReference {
    int                          normalized;
    enum cudaTextureFilterMode   filterMode;
    enum cudaTextureAddressMode  addressMode[3];
    struct cudaChannelFormatDesc channelDesc;
    int                          sRGB;
    unsigned int                 maxAnisotropy;
    enum cudaTextureFilterMode   mipmapFilterMode;
    float                        mipmapLevelBias;
    float                        minMipmapLevelClamp;
    float                        maxMipmapLevelClamp;
}
```

- normalized указывает, нормализуются ли координаты текстуры или нет;
- filterMode задает режим фильтрации;
- addressMode указывает режим адресации;
- channelDesc описывает формат текселя; он должен соответствовать аргументу DataType объявления текстурной ссылки; channelDesc имеет следующий тип:

```
struct cudaChannelFormatDesc {
  int x, y, z, w;
  enum cudaChannelFormatKind f;
```

```
};
```

где x, y, z и w равны числу бит каждого компонента возвращаемого значения, а f равно:

- cudaChannelFormatKindSigned, если эти компоненты имеют целочисленный тип со знаком,
- cudaChannelFormatKindUnsigned, если они имеют целочисленный тип без знака,
- cudaChannelFormatKindFloat, если они имеют тип с плавающей точкой.

Прежде чем ядро сможет использовать ссылку на текстуру для чтения из памяти текстур, ссылка на текстуру должна быть привязана к текстуре с использованием cudaBindTexture () или cudaBindTexture2D () для линейной памяти или cudaBindTextureToArray () для массивов CUDA. cudaUnbindTexture () используется для открепления ссылки на текстуру. После того, как ссылка на текстуру была откреплена, ее можно безопасно привязать к другому массиву, даже если ядра, которые используют ранее связанную текстуру, не завершены. Рекомендуется использовать двумерные текстуры в линейной памяти с помощью cudaMallocPitch () и использовать шаг, возвращаемый cudaMallocPitch () в качестве входного параметра, в cudaBindTexture2D ().

Следующие примеры кода связывают двухмерную ссылку на текстуру с линейной памятью, на которую указывает devPtr:

- Использование низкоуровневого API:

```
texture<float, cudaTextureType2D,
        cudaReadModeElementType> texRef;
textureReference* texRefPtr;
cudaGetTextureReference(&texRefPtr, &texRef);
cudaChannelFormatDesc channelDesc =
                              cudaCreateChannelDesc<float>();
size_t offset;
cudaBindTexture2D(&offset, texRefPtr, devPtr, &channelDesc,
                  width, height, pitch);
```

- Использование высокоуровнего API:

```
texture<float, cudaTextureType2D,
        cudaReadModeElementType> texRef;
cudaChannelFormatDesc channelDesc =
                              cudaCreateChannelDesc<float>();
size_t offset;
cudaBindTexture2D(&offset, texRef, devPtr, channelDesc,
                  width, height, pitch);
```

Следующие примеры кода связывают ссылку на 2D текстуру с массивом CUDA cuArray:

- Использование низкоуровневого API:

```
texture<float, cudaTextureType2D,
        cudaReadModeElementType> texRef;
textureReference* texRefPtr;
```

```
cudaGetTextureReference(&texRefPtr, &texRef);
cudaChannelFormatDesc channelDesc;
cudaGetChannelDesc(&channelDesc, cuArray);
cudaBindTextureToArray(texRef, cuArray, &channelDesc);
```

- Использование высокоуровнего API:

```
texture<float, cudaTextureType2D,
        cudaReadModeElementType> texRef;
cudaBindTextureToArray(texRef, cuArray);
```

Пример использования текстурной памяти:

```
// 2D float texture
texture<float, cudaTextureType2D, cudaReadModeElementType> texRef;

// Simple transformation kernel
__global__ void transformKernel(float* output,
                                int width, int height,
                                float theta)
{
    // Calculate normalized texture coordinates
    unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;

    float u = x / (float)width;
    float v = y / (float)height;

    // Transform coordinates
    u -= 0.5f;
    v -= 0.5f;
    float tu = u * cosf(theta) - v * sinf(theta) + 0.5f;
    float tv = v * cosf(theta) + u * sinf(theta) + 0.5f;


    // Read from texture and write to global memory
    output[y * width + x] = tex2D(texRef, tu, tv);
}

// Host code
int main()
```

```
{
    // Allocate CUDA array in device memory
    cudaChannelFormatDesc channelDesc =
              cudaCreateChannelDesc(32, 0, 0, 0,
                                    cudaChannelFormatKindFloat);
    cudaArray* cuArray;
    cudaMallocArray(&cuArray, &channelDesc, width, height);

    // Copy to device memory some data located at address h_data
    // in host memory
    cudaMemcpyToArray(cuArray, 0, 0, h_data, size,
                      cudaMemcpyHostToDevice);

    // Set texture reference parameters
    texRef.addressMode[0] = cudaAddressModeWrap;
    texRef.addressMode[1] = cudaAddressModeWrap;
    texRef.filterMode     = cudaFilterModeLinear;
    texRef.normalized     = true;

    // Bind the array to the texture reference
    cudaBindTextureToArray(texRef, cuArray, channelDesc);

    // Allocate result of transformation in device memory
    float* output;
    cudaMalloc(&output, width * height * sizeof(float));

    // Invoke kernel
    dim3 dimBlock(16, 16);
    dim3 dimGrid((width  + dimBlock.x - 1) / dimBlock.x,
                 (height + dimBlock.y - 1) / dimBlock.y);
    transformKernel<<<dimGrid, dimBlock>>>(output, width, height,
                                           angle);

    // Free device memory
    cudaFreeArray(cuArray);
    cudaFree(output);

    return 0;
```

```
}
```

3.3.8 Работа с константной памятью

Константная память выделяется непосредственно в коде программы при по мощи спецификатора constant . Все нити сетки могут читать из нее данные, и чтение из нее кешируется. CPU имеет доступ к ней как на чтение, так и на запись 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) ; В качестве значения параметра kind выступает одна из следующих констант, задающих направление копирования, - cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost и cudaMemcpyDeviceToDevice. Параметр stream позволяет организовывать несколько потоков команд, если вы не используете эту возможность, то следует использовать поток по умолчанию - 0.

Поскольку константная память кешируется, то она является идеальным местом для размещения небольшого объема часто используемых неизменяемых данных, которые должны быть доступны всем нитям сетки сразу.

3.4 Вопросы для самопроверки

- Как скомпилировать и запустить программу для CUDA?
- Какие функции для работы с различными типами памяти вы знаете?
- В чём различие между типами памяти?
- Что такое поток? Для чего нужны потоки?
- Что такое события?
- Как измерить время выполнения кода для GPU?
- Для чего нужна синхронизация? Какие средства для неё имеются в CUDA C?
- Как происходит поиск ошибок в программе, которая использует GPU для вычислений?

4. Дополнительно

4.1 Полезные материалы

- https://docs.nvidia.com/cuda/index.html – официальная документация CUDA.
- https://developer.nvidia.com/cuda-downloads – ссылка на скачивание CUDA Toolkit

- *Модель программирования CUDA [Электронный ресурс] : учебник / В.В. Коробицын [и др.]. — Электрон. текстовые данные. — Омск: Омский государственный университет им. Ф.М. Достоевского, 2012. — 256 c. — 978-5-7779-1489-7. — Режим доступа: http://www.iprbookshop.ru/24903.htmlВойтов А.Г. Экономическая теория [Электронный ресурс]: учебник для бакалавров/ Войтов А.Г.— Электрон.текстовые данные.— М.: Дашков и К, 2015.— 391 c.— Режим доступа: http://www.iprbookshop.ru/11012 — ЭБС «IPRbooks»*
- *Параллельные вычисления на GPU. Архитектура и программная модель CUDA [Электронный ресурс] : учебное пособие / А.В. Боресков [и др.]. — Электрон. текстовые данные. — М. : Московский государственный университет имени М.В. Ломоносова, 2015. — 336 c. — 978-5-19-011058-6. — Режим доступа:* http://www.iprbookshop.ru/54647.html
- *Параллельные вычисления общего назначения на графических процессорах [Электронный ресурс] : учебное пособие / К.А. Некрасов [и др.]. — Электрон. текстовые данные. — Екатеринбург: Уральский федеральный университет, 2016. — 104 c. — 978-5-7996-1722-6. — Режим доступа: http://www.iprbookshop.ru/69657.html*
- *Метод Монте-Карло на графических процессорах [Электронный ресурс] : учебное пособие / К.А. Некрасов [и др.]. — Электрон. текстовые данные. — Екатеринбург: Уральский федеральный университет, 2016. — 60 c. — 978-5-7996-1723-3. — Режим доступа: http://www.iprbookshop.ru/69634.html.*
- *Куликов И.М. Технологии разработки программного обеспечения для математического моделирования физических процессов. Часть 1. Использование суперкомпьютеров, оснащенных графическими ускорителями [Электронный ресурс] : учебное пособие / И.М. Куликов. — Электрон. текстовые данные. — Новосибирск: Новосибирский государственный технический университет, 2013. — 40 c. — 978-5-7782-2195-6. — Режим доступа: http://www.iprbookshop.ru/45044.html*

4.2 Контакты с преподавателем

Эл. почта: mileschko.sibsutis@yandex.ru