КАЗАНСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

Д.Н. Тумаков, Д.Е. Чикрин, А.А. Егорчев, С.В. Голоусов

ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ CUDA

Учебное пособие

КАЗАНЬ
2017

УДК 004.4, 681.3
ББК 32.81

Печатается по постановлению Редакционно-издательского совета ФГАОУВПО «Казанский (Приволжский) федеральный университет»

Научный редактор
доктор физ.-мат. наук, проф. Н. Б. Плещинский

Рецензенты
кандидат физ.-мат.наук, доцент И.Е. Плещинская
кандидат физ.-мат.наук, доцент Е.В. Рунг

Тумаков Д.Н.
Технология программирования CUDA: учебное пособие / Д.Н. Тумаков, Д.Е. Чикрин, А.А. Егорчев, С.В. Голоусов. – Казань: Казанский государственный университет, 2017. – 112 с.

ISBN 978-5-00019-913-8

В пособии описаны технология CUDA и основные принципы работы с ней. Рассмотрены вопросы использования разделяемой, константной и текстурной памяти. Приведены примеры реализации различных алгоритмов. Описаны пакеты для работы с линейной алгеброй: cuBLAS и cuBLAS-Xt. В конце глав содержатся упражнения для лабораторных и домашних занятий.

Пособие предназначено для студентов, магистрантов и аспирантов ВУЗов, специализирующихся в области параллельного программирования и численных методов.

ISBN

Оглавление

Оглавление ... 3

Введение ... 5

Глава 1. Основы работы с CUDA ... 7

Основные понятия ... 7

Сетки, блоки, нити ... 8

Варпы ... 10

Первая программа ... 13

Версии Compute Capability ... 14

Глава 2. Свойства устройства ... 16

Архитектура видеокарт NVidia ... 16

Типы видеокарт, поддерживающих CUDA ... 19

Функция cudaGetDeviceProperties() ... 21

Глава 3. Простейшая программа на CUDA ... 27

Функция копирования ... 28

Замер времени работы части кода программы ... 30

Обработка ошибок ... 31

Глава 4. Атомарные операции в CUDA ... 34

Атомарные арифметические операции ... 34

Атомарные побитовые операции ... 36

Дополнительные возможности Compute Capability 6.x ... 37

Пример использования атомарных операций ... 38

Глава 5. Работа с векторами. Математические функции ... 40

Сложение векторов ... 40

Вычисление математических функций ... 42

Глава 6. Работа с матрицами ... 47

Создание матриц ... 48

Транспонирование матрицы ... 51

Сложение матриц ... 57

Умножение матриц ... 58

Глава 7. Типы памяти. Разделяемая память ... 64

Банки данных ... 67

Пример. Оптимизация программы по перемножению двух матриц ... 69

Глава 8. Константая и текстурная память ... 75

Константная память ... 75

Текстурная память ... 76

Примеры использования одномерной текстуры ... 78

Глава 9. Пакеты для работы с векторами и матрицами 83
Пакет cuBLAS 83
Пакет cuBLAS-Xt 102
Литература 111

Введение

Параллельные вычисления давно уже перестали быть специальными технологиями, которыми пользовались лишь в исследовательских центрах. Сегодня методы параллельных вычислений в той или иной степени используются при решении практически всех сложных задач. Совершенствование технической базы компьютеров в сочетании с параллельной обработкой данных позволило существенно повысить производительность вычислений.

В данном пособии рассмотрим одну из технологий высокопроизводительных вычислений, использующую графический процессор (GPU). Изначально этот класс устройств разрабатывался для обработки графики. Техника совершенствовалась, GPU наращивали производительность, и в какой-то момент оказалось, что GPU можно успешно использовать не только для задач компьютерной графики, но и как математический сопроцессор для CPU, получая при этом существенный прирост в производительности.

Этот класс технологий получил название GPGPU (General-purpose computing for graphics processing units) – использование графического процессора для различных вычислительных задач. Популярный представитель этого класса – CUDA (Compute Unified Device Architecture), который был впервые представлен компанией NVIDIA в 2007 году. CUDA может использоваться на GPU производства NVIDIA, таких как GeForce, Quadro, Tesla, Fermi, Kepler, Maxwell, Pascal и Volta. Некоторые из этих устройств не имеют видеовыхода, и эти видеокарты предназначены исключительно для высокопроизводительных вычислений.

CUDA работает только с устройствами производства NVIDIA, но отметим, что помимо CUDA существуют и другие аналогичные технологии, например OpenCL, AMD FireStream, CUDAfy и PyCUDA.

В качестве дополнительной литературы можно порекомендовать [1-3]. В [1] разобрана как сама технология CUDA, так и вопросы оптимизации кода с использованием ptx-ассемблера. В книге также рассмотрены задачи по моделированию ряда физических процессов. Задачи моделирования и использование стандартных библиотек рассмотрены и в [2]. В [3] подробно описана работа с несколькими графическими устройствами.

Глава 1. Основы работы с CUDA

В первой главе рассмотрим базовые понятия технологии CUDA – хост, устройство, ядро, сетка, блок и нить.

Основные понятия

Хост (host) – это центральный процессор (CPU) компьютера, ***устройство (device)*** – графический процессор (GPU), расположенный на видеокарте. Каждый процессор имеет свою отдельную память, память хоста и память устройства физически разделены. Хост управляет вычислительным процессом – с хоста вызываются функции, организующие обмен данными между различными видами памяти, и запускаются задачи на устройстве. Каждой такой задаче соответсвует ***ядро (kernel)*** – функция, оформленная по особым правилам.

В настоящее время технологию CUDA поддерживают такие языки программирования, как C/C++, Fortran и Phyton. В программу, написанную для CPU, можно добавить дополнительные элементы для работы с GPU. Компилятор CUDA работает как препроцессор. После предварительной обработки исходного кода вызывается стандартный компилятор и генерируется исполняемый код отдельно для хоста и отдельно для устройства.

Обычно в CUDA-программах при работе с устройством содержатся следующие этапы:

- выделяется память под данные на устройстве;
- копируются данные из памяти хоста в память устройства;
- на устройстве выполняются ядра;
- результаты пересылаются из памяти устройства в память хоста.

В расширении языка C/C++ используют три спецификатора функций, которые определяют, откуда функции вызываются и где выполняются:

__host__ – вызываются с хоста, выполняются на хосте (можно не указывать действует по умолчанию);

__global__ – вызываются с хоста, выполняются на устройстве;

__device__ – вызываются с устройства, выполняются на устройстве.

Сетки, блоки, нити

При вычислениях на GPU одновременно запускается большое число параллельных процессов, их принято называть нитями. Все запущенные на выполнение нити объединены в сложную структуру – ***сетку (grid).*** Сетка представляет собой одномерный, двухмерный или трехмерный массив ***блоков (block)***. Каждый блок – это одномерный, двухмерный или трехмерный массив ***нитей (thread)***. При этом все блоки, образующие сетку, имеют одинаковую размерность и размер (Рис. 1).

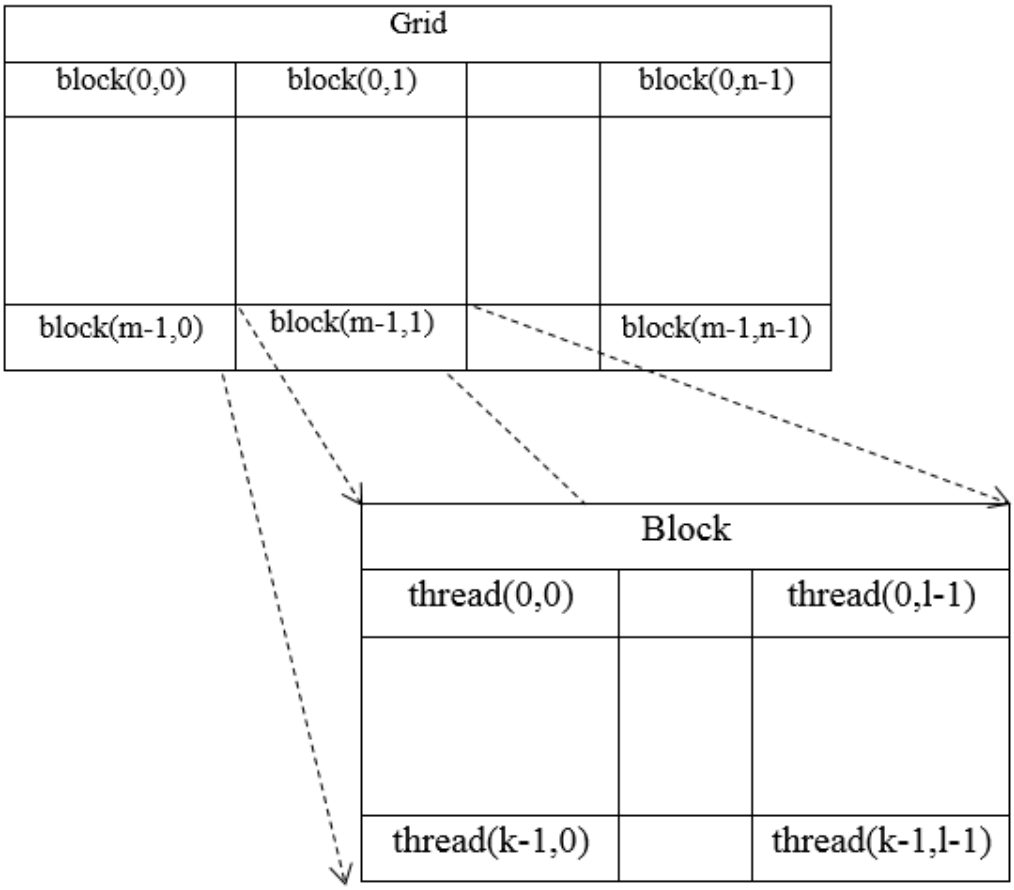

Рис. 1. *Иерархия нитей в CUDA*

Каждый блок в сетке имеет свой адрес, состоящий из одного или двух неотрицательных целых чисел (индекс блока в сетке). Аналогично каждая нить

внутри блока также имеет также свой адрес – одно, два или три неотрицательных целых числа, задающих индекс нити внутри блока.

Поскольку одно и то же ядро выполняется одновременно очень большим числом нитей, то для того, чтобы ядро могло однозначно определить номер нити (а значит, и элемент данных, который нужно обрабатывать), используются встроенные переменные ***threadIdx*** (индекс текущей нити в вычислении на GPU, имеет тип uint3) и ***blockIdx*** (индекс текущего блока в вычислении на GPU, имеет тип uint3). Каждая из этих переменных является трехмерным целочисленным вектором. Обратим внимание, что они доступны только для функций, выполняемых на GPU; для функций, выполняющихся на CPU, они не имеют смысла.

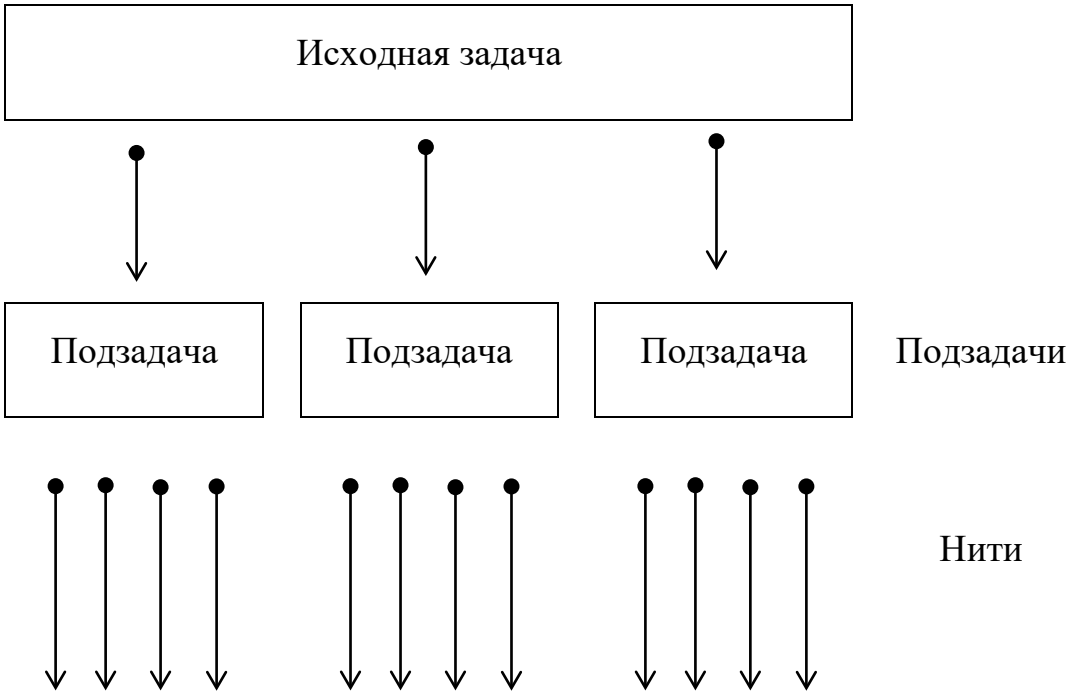

Рис. 2. *Разбиение исходной задачи на набор независимо решаемых подзадач*

Также ядро может получить размеры сетки и блока через встроенные переменные ***gridDim*** (размерность сетки, имеет тип dim3) и ***blockDim*** (размерность блока, также имеет тип dim3).

Подобное разделение всех нитей является еще одним общим приемом использования CUDA: исходная задача разбивается на набор отдельных

подзадач, решаемых независимо друг от друга (Рис. 2). Каждой такой подзадаче соответствует блок нитей.

При этом каждая подзадача совместно решается всеми нитями своего блока. Объединение нитей в варпы происходит отдельно для каждого блока; таким образом, все нити одного варпа всегда принадлежат одному блоку. При этом нити могут взаимодействовать между собой только в пределах блока. Нити разных блоков взаимодействовать между собой не могут.

Подобный подход является удачным компромиссом между необходимостью обеспечить взаимодействие нитей между собой и стоимостью подобного взаимодействия – обеспечить возможность взаимодействия каждой нити с каждой было бы слишком сложно и дорого.

Однако есть способы косвенного взаимодействия нитей в блоке. Они могут взаимодействовать друг с другом через разделяемую (shared) память (об использовании разделяемой памяти речь пойдет в Главе 7).

Каждый блок получает в свое распоряжение определенный объем быстрой разделяемой памяти, которую все нити блока могут совместно использовать. Поскольку нити блока необязательно выполняются физически параллельно (то есть мы имеем дело не с чистой SIMD-архитектурой, а имеет место прозрачное управление нитями), то для того, чтобы не возникало проблем с одновременной работой с shared-памятью, необходим некоторый механизм синхронизации нитей блока.

CUDA предлагает довольно простой способ синхронизации – это так называемая барьерная синхронизация. Для ее осуществления используется вызов встроенной функции ***__syncthreads()***, которая блокирует вызывающие нити блока до тех пор, пока все нити блока не войдут в эту функцию. Таким образом, при помощи __syncthreads() 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) (Рис. 3)

Варпы

Варп (Warp) – группировка потоков по 32 штуки, которые оказываются

частями более крупных образований – блоков (blocks). Все потоки каждого блока запускаются строго на одном потоковом мультипроцессоре (SM, Streaming Multiprocessors), поэтому имеют доступ только к его ресурсам. Однако, на одном SM может запускаться более одного блока, и тогда ресурсы будут разделяться между ними поровну.

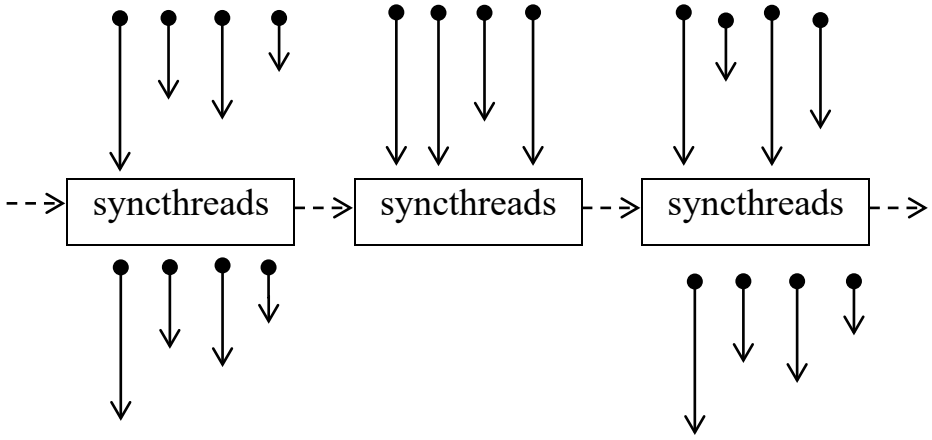

Рис. 3. *Барьерная синхронизация*

В каждом SM существует блок управления, который занимается распределением ресурса процессорного времени. Делается это так, что в каждый момент времени все ядра одного SM исполняют строго один варп. По его завершению работы варпа оптимальным способом выбирается следующий варп, приписанный к данному SM. Таким образом, оказывается, что потоки одного варпа синхронизируются за счет аппаратной особенности CUDA и исполняются по еще более близкому к SIMD (Single Instruction, Multiple Data) методу. Но потоки даже одного блока из разных варпов могут оказаться заметно рассинхронизированными.

Можно сделать вывод, что взаимодействие между потоками одного блока следует стараться осуществлять через их общую быструю разделяемую память, а между потоками двух различных блоков – только с использованием глобальной памяти. Тут возникает проблема: слежение за актуальностью данных в общедоступной разным потокам для чтения и записи области памяти. Иначе говоря – проблема синхронизации потоков. Как уже отмечалось, в

рамках одного блока потоки каждого варпа синхронизированы между собой. Для синхронизации потоков блока вне зависимости от принадлежности к варпам существует несколько команд барьерного типа.

Как была сказано выше, основная из них – **__syncthreads().** Эта функция заставляет каждый поток ждать, пока все остальные потоки этого блока достигнут этой точки и все операции по доступу к разделяемой и глобальной памяти, совершенные потоками этого блока, завершатся и станут видны потокам этого блока. Не желательно размещать эту команду внутри условного оператора **if**, а также следует обеспечивать безусловный вызов этой функции всеми потоками блока.

Выпишем остальные «барьерные команды»:

- **__threadfence_block()** заставляет ждать вызвавший её поток, пока все совершенные операции доступа к разделяемой и глобальной памяти завершатся и станут видны потокам этого блока.
- **__threadfence()** заставляет ждать вызвавший её поток, пока все совершенные операции доступа к разделяемой памяти станут видны потокам этого блока, а операции с глобальной памятью – всем потокам на девайсе.
- **__threadfence_system()** подобна **__threadfence()**, но включает синхронизацию с потоками на CPU («хосте»), при использовании page-locked памяти.

Параметры запуска ядра служат для описания количества блоков, нитей и памяти, которые необходимо выделить при расчете на GPU. Например, запуск ядра может быть осуществлен следующим образом:

```
    myKernelFunc<<<gridSize, blockSize>>>(float *param1,
int *param2)
```

где

- **gridSize** – размерность сетки блоков (тип dim3), выделенную для расчетов,
- **blockSize** – размер блока (тип dim3), выделенного для расчетов,

а `myKernelFunc` – функция ядра (спецификатор __global__).

Дополнительные типы переменных и их спецификаторы будут рассмотрены непосредственно в примерах работы с памятью.

Первая программа

В любом учебном пособии первая программа, которую предлагают вам написать - это программа, выводящая на консоль «Hello, world!». Данная программа на языке программирования C++ выглядит следующим образом:

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>

__global__ void HelloWorld()
{
    printf("Hello world, %d, %d\n", blockIdx.x,
threadIdx.x);
}

int main()
{
    HelloWorld << <1, 1 >> >();
// ожидаем нажатия любой клавиши
    getchar();
    return 0;
}
```

Первые две строки в программе (два `#include`) не обязательно. В среде Visual Studio они подключаются по умолчанию.

В отличие от файлов с программами на языке программирования C++, имеющих расширения .cpp, файлы с программами, которые используют технологию CUDA, должны иметь расширение .cu.

Версии Compute Capability

Перечислим поддержку возможностей в версиях вычислительной производительности (Compute Capability):

- с версии 1.1 – атомарные операции над 32-битными целыми числами и atomicExch() над 32-битными числами с плавающей точкой в глобальной памяти;
- с версии 1.2 – тоже в разделяемой памяти, также в глобальной над 64-разрядными целыми, функции голосования в варпе (warp vote functions);
- с версии 1.3 – операции над числами с плавающей точкой двойной точности;
- с версии 2.0 – атомарные операции над 64-битными целыми числами в разделяемой памяти и атомарное сложение 32-битных чисел с плавающей точкой, функции _ballot(), _threadfence_system(), _syncthreads_count(), _syncthreads_and(), _syncthreads_or(), поверхностные функции, трехмерная сетка блоков;
- с версии 3.0 – функции варп-перемешивания (warp shuffle functions);
- с версии 3.2 – сдвиги __funnelshift_lc() и __funnelshift_rc() (funnel shift);
- с версии 3.5 – динамический параллелизм;
- с версии 5.3 – операции над числами половинной точности;
- с версии 6.0 - атомарное сложение 64-битных чисел с плавающей точкой в глобальной и разделяемой памяти.

Заметим, что версия производительности – не то же самое, что версия CUDA SDK; она зависит он архитектуры конкретной видеокарты.

- CUDA SDK 6.5 – последняя версия, поддерживающая производительность 1.x;
- CUDA SDK 7.5 поддерживает 2.0 – 5.x;
- CUDA SDK 8.0 поддерживает 2.0 – 6.x;
- В версии 9.0 поддержка производительности 2.x удалена.

Вопросы к разделу

1. Какие основные части содержит CUDA-программа?
2. Что такое ядро? Опишите все возможные спецификаторы функций и параметры ядра.
3. Как организована иерархия нитей в CUDA?
4. Какие типы синхронизации потоков существуют?

Лабораторная работа

1. Определите характеристики используемой видеокарты: число SM, объем памяти, частоту работы видеопроцессоров и памяти, ширину полосы пропускания памяти. Сделайте выводы о производительности видеокарты.
2. Напишите программу, которая выводит на экран результат сложения двух чисел.

Глава 2. Свойства устройства

В этом разделе опишем, какие существуют видеокарты, и приведем характеристики некоторых из них. Также рассмотрим функцию, благодаря которой можно узнать параметры и характеристики видеокарты, установленной непосредственно на устройстве.

Архитектура видеокарт NVidia

Рассмотрим подробнее физическое устройство видеокарт. На рис. 4 приведена схема архитектуры видеокарты GeForce 8800 GT. Центральный процессор обменивается данными с GPU через мост (Bridge). Вычислительная часть видеокарты состоит из семи текстурных процессорных кластеров (Texture Processor Cluster), каждый из которых содержит два мультипроцессора (SM), блок текстур (Texture Unit) и совмещенный текстурный кэш и кэш первого уровня (Tex/L1).

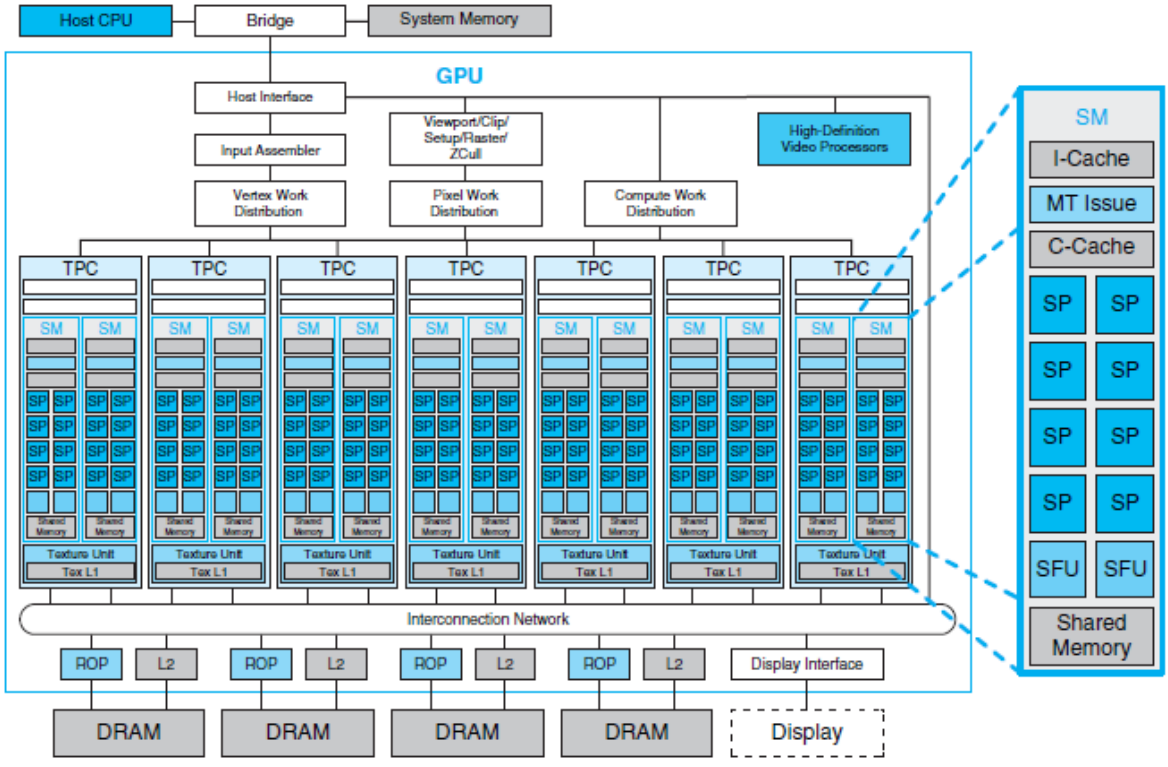

Рис. 4. *Архитектура видеокарты GeForce 8800 GT*

Блоки текстур осуществляют выборку и фильтрацию текстурных данных, необходимых для построения сцены. Число текстурных блоков в видеочипе определяет текстурную производительность и скорость выборки из текстур. Текстурная производительность является одним из важнейших параметров видеочипов. Особое влияние этот параметр оказывает на скорость при использовании трилинейной и анизотропной фильтраций, требующих дополнительных текстурных выборок.

Мультипроцессор, в свою очередь, содержит 8 потоковых процессоров (Streaming Processors) или ядер CUDA (CUDA cores). Таким образом, общее число SP = 14 * 8 = 112. Также каждый SM содержит по два блока для вычисления специальных (трансцендентных) функций (SFU), таких как синус, косинус и т.д. В состав SM входит разделяемая память (Shared Memory), блок инструкций нитей (Multithreaded Instruction Unit), константный кэш (C-Cache) и кэш инструкций (L-Cache).

Таблица 2. *Сравнительные характеристики видеокарт GF 8800*

	GeForce 8800 Ultra	**GeForce 8800 GTX**	**GeForce 8800 GTS**	**GeForce 8800 GT**	**GeForce 8800 GS**
Stream процессоры	128	128	96	112	96
Частота ядра (МГц)	612	575	500	600	550
Частота шейдерного блока (МГц)	1500	1350	1200	1500	1375
Частота памяти (МГц)	1080	900	800	900	800
Объем памяти	768MB	768MB	640MB и 320MB	512MB	384MB
Интерфейс памяти	384-bit	384-bit	320-bit	256-bit	192-bit
Полоса пропускания памяти (ГБ/с)	103.7	86.4	64	57.6	38.4

Вычислительная часть устройства соединяется с видеопамятью (DRAM) посредством кэша второго уровня (L2) и блоков операций растеризации (ROP). Блоки опреаций растеризации осуществляют операции записи рассчитанных видеокартой пикселей в буферы и операции их смешивания (блендинга). Производительность блоков ROP влияет на филлрейт (скорость заполнения пикселами), а это является одной из основных характеристик видеокарт.

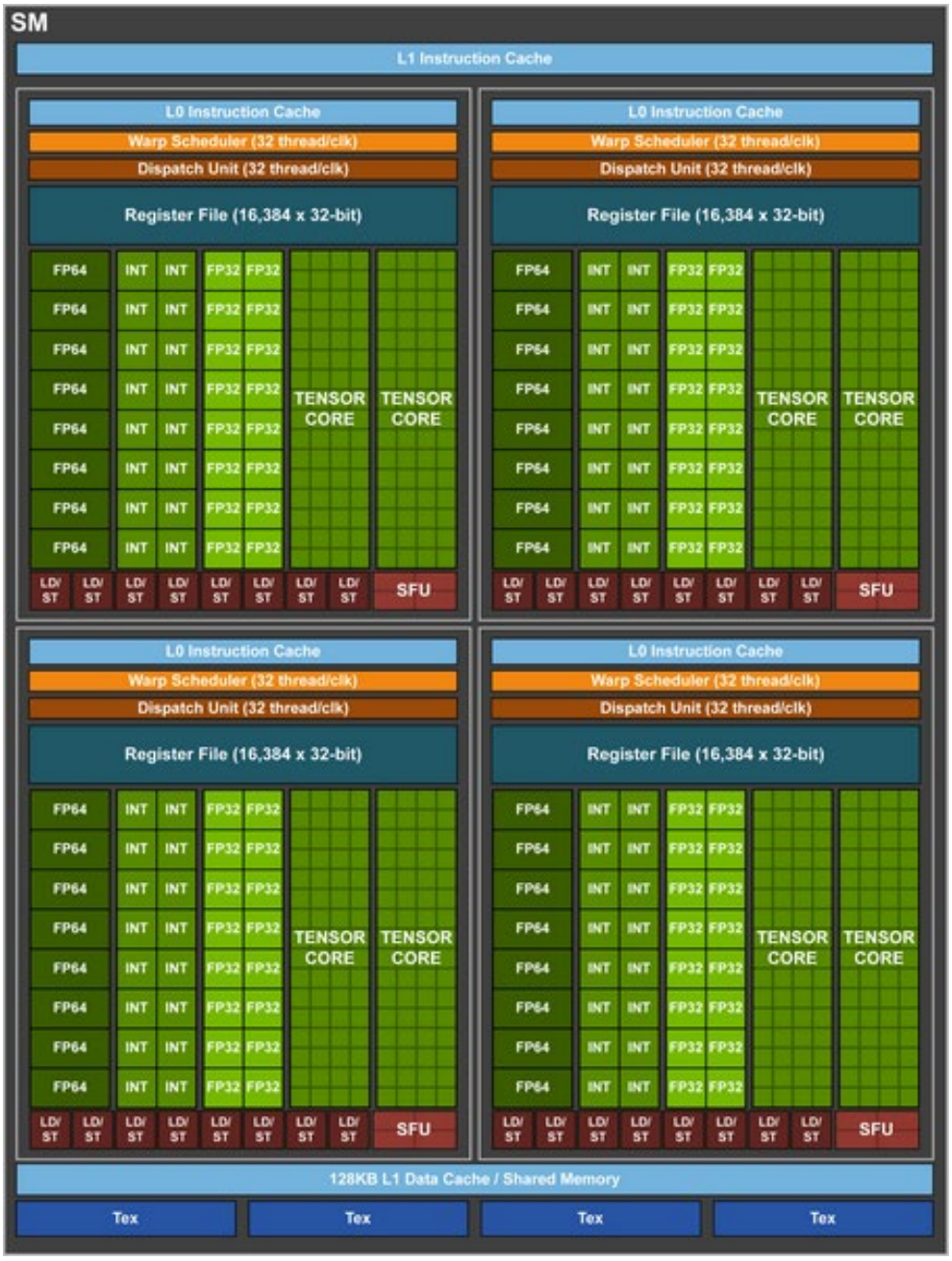

Рис. 5. *Архитектура нового мультипроцессора семейства Volta GV100*

Надо иметь в виду, что видеокарты одной линейки имеют одинаковую производительность, и, следовательно, одинаковое число SP в MP. Однако остальные характеристики могут отличаться. В таблице 2 приведены характеристики для GeForce 8800.

Рассмотрим также новый, недавно анонсированный, мультипроцессор семейства Volta (Рис. 5). Каждый SM содержит четыре блока, в которых есть помимо ядер выполняющих целочисленные вычисления, ядра для работы с вещественными 32 и 64-битными числами, а также новые, появившиеся в данной линейке процессоров, тензорные ядра (Tensor Cores), разработанные специально для обучения нейросетей в задачах глубокого обучения.

На Рис. 5 таже представлены планировщик варпов (Warp Scheduler), блок выборки (Dispatch Unit), регистры мультипроцессора (Register File) и блоки загрузки и сохранения (LD/ST – Load/Store Units).

Типы видеокарт, поддерживающих CUDA

Приведем в таблице 1 характеристики некоторых видеокарт, поддерживающих технологию CUDA.

Первая колонка таблицы – название видеокарты: указывается модель и объем памяти, в названии GT обозначает обычные видеокарты низкого уровня производительности, GTX обозначает видеоадаптеры среднего и высокого уровня, Ti или Titan обозначает десктопные карты.

Вторая колонка обозначает маркировку кристаллов: GT – Tesla, GF – Fermi, GK – Kepler, GM – Maxwell, GP – Pascal. Третья и четверая колонки фактически показывают производительность видеокарты. Цифры в последней колонке, разрядности шины памяти, указывают на пропускную способность памяти.

Отметим, что частота ядра, как правило, немного ниже частот шейдерных блоков. Частота же памяти в современных видеокартах существенно выше частоты ядра.

Таблица 1. *Характеристики некоторых видеокарт*

GeForce	GPU / кристалл	Кол-во процессоров-ядер / текстурных блоков / ROP – блоков растеризации	Частота ядра GPU / шейдерного блока / памяти в MHz.	Разрядность шины пямяти
Titan X 12gb	GP102	3584/224/96	1417/1530/10000	384 bit
GTX 1080 8gb	GP104	2560/160/64	1607/1733/10000	256 bit
GTX 1060 6gb	GP106	1280/80/48	1506/1709/8000	192 bit
GTX 980 Ti 4gb	GM200	2816/176/96	1000/1075/7000	384 bit
GTX 980 4 gb	GM204	2048/104/64	1126/1216/7000	256 bit
GTX 960 2 gb	GM206	1024/64/32	1126/1178/7000	128 bit
Titan Z 12gb	2xGK110	5760/448/96	705/876/7000	2×384
GTX 780 Ti 3gb	GK110	2880/240/48	875/928/7000	384 bit
GTX 760 2 gb	GK104	1152/96/32	980/1033/6008	256 bit
GTX 750 1 gb	GM107	512/32/16	1020/1085/5000	128 bit
GTX 680 2 gb	GK104	1538/128/32	1006/1058/6008	256 bit
GTX 660 2 gb	GK106	960/80/24	980/1033/6008	192 bit
GT 640 2 gb	GK107	384/32/16	900/900/1784	128 bit
GT 630 1 gb	GF108	96/16/4	810/1620/3200	128 bit
GT 610 1 gb	GF119	48/8/4	810/1620/1800	64 bit
GTX 580 1.5 gb	GF110	512/64/48	772/1544/4008	384 bit
GTX 480 1.5 gb	GF100	480/60/48	701/1402/3700	384 bit
GTX 560 1 gb	GF114	336/56/32	675/1620/4004	256 bit
GTX 460 1 gb	GF104	336/56/32	675/1350/3600	256 bit
GTX 550 Ti 1gb	GF116	192/32/24	900/1800/4100	192 bit
GTS 450 1 gb	GF106	192/32/16	783/1566/3600	128 bit
GT 440 1 gb	GF108	96/16/4	810/1620/3200	128 bit
GT 520 1 gb	GF119	48/8/4	810/1620/1800	64 bit
GTX 280 1 gb	GT200	240/80/32	602/1296/2215	512 bit
GTS 250 1 gb	G92b	128/64/16	738/1836/2200	256 bit
GT 240 1 gb	GT215	96/32/8	550/1340/3400	128 bit
GT 220 1 gb	GT216	48/16/8	625/1360/1580	128 bit
GT 210 1 gb	GT218	16/8/4	589/1402/1000	64 bit
9800 GT 1 gb	G92	112/56/16	600/1500/1800	256 bit
9600 GT 1 gb	G94	64/32/16	650/1625/1800	128 bit
9500 GT 1 gb	G96	32/16/8	550/1400/1600	128 bit
8400 GS 512 mb	G86	16\8\4	450/900/800	64 bit

Функция cudaGetDeviceProperties()

Получим от видеокарты информацию о ее возможностях и разберемся, что они дают. Для определения свойств видеокарты используется функция cudaGetDeviceProperties(), принимающая в качестве параметра номер видеокарты и возвращающая в структуре cudaDeviceProp интересующие нас значения. Выведем на экран полученные переменные.

Приведем основные переменные структуры cudaDeviceProp:

```
struct cudaDeviceProp {
        char name[256];
        size_t totalGlobalMem;
        size_t sharedMemPerBlock;
        int regsPerBlock;
        int warpSize;
        size_t memPitch;
        int maxThreadsPerBlock;
        int maxThreadsDim[3];
        int maxGridSize[3];
        size_t totalConstMem;
        int clockRate;
        int deviceOverlap;
        int multiProcessorCount;
        int kernelExecTimeoutEnabled;
        int integrated;
        int canMapHostMemory;
        int concurrentKernels;
    …
    }
```

Здесь:

- name[256] представляет собой ASCII строку идентификации устройства;
- totalGlobalMem – общий объем глобальной памяти, доступной на устройстве в байтах;

- sharedMemPerBlock – максимальный объем общей памяти, доступной для нити блока в байтах;
- regsPerBlock – максимальное количество 32-битных регистров, доступных для нити блока;
- warpSize – размер Варпа в потоках;
- maxThreadsPerBlock – максимальное количество потоков на блок;
- maxThreadsDim[3] содержит максимальный размер каждого измерения блока;
- maxGridSize[3] содержит максимальный размер каждого измерения сетки;
- totalConstMem – общий объем константной памяти, доступной на устройстве в байтах;
- clockRate – тактовая частота в килогерцах;
- deviceOverlap равно 1, если устройство может одновременно копировать память между хостом и устройством во время выполнения ядра, или 0, если нет;
- multiProcessorCount – число мультипроцессоров на устройстве;
- kernelExecTimeoutEnabled равно 1, если существует ограничение времени выполнения для ядер, выполненных на устройстве, или 0, если нет.
- integrated равен 1, если устройство является интегрированным (материнская плата) GPU и 0, если оно является дискретным.
- canMapHostMemory равно 1, если устройство может отображать память хоста в адресное пространство CUDA для использования с cudaHostAlloc() / cudaHostGetDevicePointer(), или 0, если нет;
- concurrentKernels равно 1, если устройство поддерживает выполнение нескольких ядер в том же контексте одновременно, или 0, если нет.

Приведем пример использования функции cudaGetDeviceProperties:

```
cudaDeviceProp deviceProp;
cudaGetDeviceProperties(&deviceProp, 0);//определение
параметров GPU с номером 0
printf("Device name : %s\n", deviceProp.name);
printf("Total global memory : %d MB\n",
deviceProp.totalGlobalMem / 1024 / 1024);
printf("Shared memory per block : %d\n",
deviceProp.sharedMemPerBlock);
printf("Registers per block : %d\n",
deviceProp.regsPerBlock);
printf("Warp size : %d\n", deviceProp.warpSize);
printf("Memory pitch : %d\n", deviceProp.memPitch);
printf("Max threads per block : %d\n",
deviceProp.maxThreadsPerBlock);
printf("Max threads dimensions : x = %d, y = %d, z =
%d\n", deviceProp.maxThreadsDim[0],
deviceProp.maxThreadsDim[1],
deviceProp.maxThreadsDim[2]);
printf("Max grid size: x = %d, y = %d, z = %d\n",
deviceProp.maxGridSize[0], deviceProp.maxGridSize[1],
deviceProp.maxGridSize[2]);
printf("Clock rate: %d\n", deviceProp.clockRate);
printf("Total constant memory: %d\n",
deviceProp.totalConstMem);
printf("Compute capability: %d.%d\n",
deviceProp.major, deviceProp.minor);
printf("Texture alignment: %d\n",
deviceProp.textureAlignment);
printf("Device overlap: %d\n",
deviceProp.deviceOverlap);
printf("Multiprocessor count: %d\n",
deviceProp.multiProcessorCount);
```

```
    printf("Kernel execution timeout enabled: %s\n",
    deviceProp.kernelExecTimeoutEnabled ? "true" :
    "false");
    scanf("");
```

Посмотрим, что выдала программа:

```
Device name : GeForce 940M
Total global memory : 2048 MB
Shared memory per block : 49152
Registers per block : 65536
Warp size : 32
Memory pitch : 2147483647
Max threads per block : 1024
Max threads dimensions : x = 1024, y = 1024, z = 64
Max grid size: x = 2147483647, y = 65535, z = 65535
Clock rate: 1176000
Total constant memory: 65536
Compute capability: 5.0
Texture alignment: 512
Device overlap: 1
Multiprocessor count: 3
Kernel execution timeout enabled: true
```

Также важным моментом является и определение версии видеокарты. Следующий код выводит производительность видеокарты

```
cudaDeviceProp deviceProp;

cudaGetDeviceProperties(&deviceProp, 0);

printf("Device has Compute Capability %d.%d\n",
deviceProp.major, deviceProp.minor);
```

Вопросы к разделу

1. Опишите примерную архитектуру современных видеокарт NVidia.
2. Какие параметры видеокарты характеризуют её производительность?
3. Какие основные характеристики видеокарты можно узнать с помощью функции `cudaGetDeviceProperties()`?

Лабораторная работа

1. Определите следующие параметры видеокарты с поддержкой технологии CUDA:
   - наименование;
   - общий объем графической памяти;
   - объем разделяемой памяти в пределах блока;
   - число регистров в пределах блока;
   - размер варпа;
   - максимально допустимое число потоков в блоке;
   - версию вычислительных возможностей;
   - число потоковых мультипроцессоров;
   - тактовую частоту ядра;
   - объем кэша второго уровня;
   - ширину шины памяти видеокарты;
   - максимальную размерность при конфигурации потоков в блоке и блоков в сетке.
2. Сравните результаты, полученные программно, с информацией от производителя.

Домашняя работа

1. Доработайте лабораторное задание 1, добавив в него вывод объема константной памяти и пиковую частоту видеокарты в МГц.

Глава 3. Простейшая программа на CUDA

Каждая программа, использующая технологию CUDA, делится на две части: последовательная часть (код, выполняющийся на CPU) и параллельная часть (код, выполняющийся на GPU). Напишем программу «Hello, world!», которая будет вызвана на двух блоках и пяти нитях.

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>

__global__ void HelloWorld()
{
    printf("Hello world, %d, %d\n", blockIdx.x,
    threadIdx.x);
}

int main()
{
    HelloWorld << <2, 5 >> >();

// хост ожидает завершения работы девайса
    cudaDeviceSynchronize();

// ожидаем нажатия любой клавиши
    getchar();
    return 0;
}
```

Ядро, так называется функция, выполняющаяся на GPU. Эта функция имеет спецификатор __global__, который сообщает компилятору, что функция вызывается с CPU и выполняется на GPU. На ядро налагается ряд некоторых ограничений: не содержать рекурсию, не иметь переменное число аргументов, не иметь переменные static внутри себя. Вызов функции осуществляются следующим образом: имя_функции <<<dimGrid, dimBlock>>> (параметры).

Тройные угловые скобки перед функцией, выполняющейся на GPU, – это синтаксис CUDA. Параметры в угловых скобках определяют количество и конфигурацию параллельных процессов GPU: размер сетки и размер блока.

Функция копирования

Рассмотрим более сложный пример использования технологии CUDA. Для написания программы, приведенной ниже, необходимо изучить принципы работы с глобальной памятью GPU.

Глобальная память является самой медленной, и если необходима высокая производительность, то количество операций с глобальной памятью следует свести к минимуму.

Назначение функции	Синтаксис функции
Выделение памяти	cudaMalloc(void **devPtr, size_t size);
Освобождение памяти	cudaFree(void **devPtr);
Копирование данных	cudaMemcpy(void *dst, const void *src, size_t size, enum cudaMemcpyKind kind) В качестве параметра kind, задающее направление копирования, могут выступать: cudamemcpyHostToDevice, cudamemcpyDeviceToHost, cudamemcpyDeviceToDevice, cudamemcpyHostToHost.

Итак, рассмотрим немного более сложную программу. Данная программа складывает два целых числа на GPU.

```
//подключение библиотек
#include "cuda_runtime.h"
```

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

// ядро
__global__ void add( int *a, int *b, int *c ) {
    *c = *a + *b;
}

//главная функция
int main()
{
    // переменные на CPU
    int a, b, c;

    // переменные на GPU
    int *dev_a, *dev_b, *dev_c;
    int size = sizeof( int );      //размерность

    // выделяем память на GPU
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, size );

    // инициализация переменных
    a = 2;
    b = 7;

    // копирование информации с CPU на GPU
    cudaMemcpy( dev_a, &a, size, cudaMemcpyHostToDevice
    );
    cudaMemcpy( dev_b, &b, size, cudaMemcpyHostToDevice
    );

    // вызов ядра
    add<<< 1, 1 >>>( dev_a, dev_b, dev_c );
    // копирование результата работы ядра с GPU на CPU
```

```
    cudaMemcpy( &c, dev_c, size, cudaMemcpyDeviceToHost
    );

    // вывод информации
    printf("%d + %d = %d\n", a, b, c);

    // очищение памяти на GPU
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
    return 0;
}
```

Замер времени работы части кода программы

Одним из важных параметров при написании программы является время, затраченное на ее выполнение. Для замера времени выполнения операций на CPU есть множество возможностей. Для того, чтобы замерить время, которое было затрачено на выполнение вычислений на GPU, можно воспользоваться механизмом событий (CUDA events). Событие – это объект типа cudaEvent_t, используемый для обозначения «точки» среды вызовов CUDA.

Приведем пример фрагмента кода, замеряющий время выполнения ядра на GPU:

```
    // инициализируем события
    cudaEvent_t start, stop;
    float elapsedTime;

    // создаем события
    cudaEventCreate(&start);
    cudaEventCtreate(&stop);

    // запись события
    cudaEventRecord(start, 0);

    // вызов ядра
```

```
cudaEventRecord(stop,0);

// ожидание завершения работы ядра
cudaEventSynchronize(stop);
cudaEventElapsedTime(&elapsedTime, start, stop);

// вывод информации
printf("Time spent executing by the GPU: %.2f
millseconds\n", elapsedTime);

// уничтожение события
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

Обработка ошибок

После выполнения функций на GPU, они возвращают значения типа cudaError_t. Функции, отработавшие на девайсе, возвращают значение cudaSuccess, в противном случае возвращается код ошибки.

Назначение функции	Синтаксис функции
Возвращает описание ошибки в виде строки	cudaGetErrorString(cudaError_t error);
Возвращает код последней ошибки	cudaGetLastError();

Основные выводы ответа при обработке ошибок:

1. cudaSuccess – при удачном выполнении.
2. cudaErrorMemoryAllocation – при ошибке выделения памяти.
3. cudaErrorInvalidValue – неверные параметры аргумента (например, размер копирования отрицателен).
4. cudaErrorInvalidDevicePointer – неверный указатель памяти в видеокарте.
5. cudaErrorInvalidMemcpyDirection – неверное направление (например, перепутан источник и место-назначение копирования).
6. cudaErrorInitializationError – ошибка инициализации.
7. cudaErrorPriorLaunchFailure – ошибка при предыдущем асинхронном

запуске функции.

8. cudaErrorInvalidResourceHandle – неверный обработчик события.

Приведем пример использования функции cudaGetLastError().

```
cudaError_t err = cudaGetLastError();
if (err != cudaSuccess) printf("%s ",
cudaGetErrorString(err));
```

В данном примере проверяется, есть ли ошибка у последней команды, выполненной на девайсе. Если ошибка есть, то выводится ее описание.

Вопросы к разделу

1. На какие условно две части можно разделить CUDA-программу?
2. Для чего нужна глобальная память? Какие имеются функции в CUDA для работы с глобальной памятью?
3. Опишите механизм, использующийся в CUDA-программах, для замера времени работы ядра.
4. Какие механизмы в CUDA-программах используются для обработки ошибок?

Лабораторная работа

1. Измерьте скорость копирования данных (ГБ/сек) между CPU и GPU.
2. Напишите программу, вычисляющую число пи методом интегрирования четверти окружности единичного радиуса (можно использовать формулу для площади четверти окружности: $S = \int_0^1 \sqrt{1-x^2}\,dx = \pi/4$). В программе предусмотрите проверку на ошибку выполнения функции.

Домашняя работа

1. Напишите программу вычисления дзета-функции Римана путем суммирования ряда из обратных степеней.

Глава 4. Атомарные операции в CUDA

CUDA для GPU, начиная с производительности 1.1 и выше, поддерживает атомарные операции над глобальной и разделяемой памятью. Атомарность заключается в том, что гарантируется корректность выполнения операции в случае, когда много нитей пытаются ее выполнить одновременно.

Атомарные арифметические операции

Наиболее часто используемыми атомарными арифметическими операциями являются atomicAdd и atomicSub, служащие для увеличения или уменьшения величины на заданное значение. При этом функция возвращает старое значение. Нужно обратить внимание, что atomicAdd поддерживает операции над 64-битовыми величинами, но только расположенными в глобальной памяти.

```
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 );

int atomicSub( int *addr, int value );

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

Операция atomicExch осуществляет атомарный обмен значениями – передаваемое значение записывается по указанному адресу, а предыдущее значение возвращается. При этом подобный обмен происходит как одна транзакция, то есть ни одна нить не может «вклиниться» между шагами этого

обмена. Операции над 64-битовыми значениями поддерживаются только для глобальной памяти.

```
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 нуль, иначе значение по адресу увеличивается на единицу. Возвращается старое значение.

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

Следующая функция (CAS – Compare And Swap) читает старое 32- или 64-битовое 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. Возвращается всегда 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 int *addr,
unsigned long long compare, unsigned long long value
);
```

Атомарные побитовые операции

Побитовые атомарные операции читают слово по заданному адресу, применяют к нему заданную побитовую операцию с заданным параметром и записывают результат обратно. Возвращается всегда старое значение, находившееся по переданному 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 );
```

Дополнительные возможности Compute Capability 6.x

Начиная с производительности 6.x, существует ряд нововведений. В частности, представлен новый тип атомарных операций. Например, atomicAdd_system гарантирует, что инструкция является атомарной относительно других CPU и GPU в системе. Функция atomicAdd_block подразумевает, что инструкция является атомарной только с учетом атомистики от других потоков в одном блоке потока. В следующем примере как CPU, так и GPU могут атомарно обновлять целочисленное значение по адресу addr:

```
__global__ void add10(int *addr) {
    // only available on devices with compute
    // capability 6.x
    atomicAdd_system(addr, 10);
}

void foo()
{
    int *addr;
    cudaMallocManaged(&addr, 4);
    *addr = 0;
```

```
    add10<<<2, 5>>>(addr);
    // CPU atomic operation
    _sync_fetch_and_add(addr, 10);
}
```

Также в версиях производительности 6.x добавлено сложение 64 битных чисел.

Пример использования атомарных операций

Пусть задана двумерная сетка нитей, и требуется сосчитать количество нитей, индексы которых удовлетворяют некоторому условию. Эту задачу можно реализовать следующим способом: каждая нить вызывает некую булеву функцию, и если результат вызова этой функции равна true, добавляет единицу некой глобальной переменной res.

Приведем ядро такой программы.

```
__global__ void Kernel(int * res) {

    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    if ( Condition(i, j) )// проверка условия
    {
        atomicAdd(res, 1);
    }
}
```

Здесь `Condition(i, j)` некоторое условие, например, `i>=j`.

Вопросы к разделу

1. Что такое атомарная операция? В чем ее ключевое отличие от обычных операций?
2. Какие атомарные операции вы знаете?

Лабораторная работа

1. Вычислите приблизительное значение числа π методом Монте Карло. Постройте равномерную сетку нитей в квадрате [0, 1]X[0, 1] и посчитайте количество нитей, которые попали в круг с центром в нуле и с радиусом, равным единице. Для вычисления случайного числа напишите __device__ функцию.

Домашняя работа

1. Вызов атомарной функции, в целом, замедляет работу программы. Например, если очень много нитей должны добавить единицу одной и той же переменной, то они встанут в очередь, и будут простаивать, пока все по очереди не выполнят эту операцию. Одним из подходов для решения этой проблемы является уменьшение количества нитей, вызывающих атомарную операцию. Учитывая выше сказанное, подумайте, как можно уменьшить количество нитей, вызывающих атомарную операцию в методе Монте Карло. Оптимизируйте таким образом программу, и посмотрите, насколько уменьшается время ее работы. Также посмотрите, как меняется точность вычисления числа π.

Глава 5. Работа с векторами. Математические функции

В языке Си (в частности, и в диалекте CUDA C) векторы обычно представляются массивами, как правило, динамическими (память выделяется с помощью функций семейства malloc() и освобождается вызовом free(), аналоги в CUDA см. ниже).

В функции же передается не сам массив целиком, а лишь *указатель* на его начало (переменная фактически целочисленного типа, размер которого зависит от архитектуры (как правило, 32 или 64 разряда), содержащая адрес начала массива). Зная размер одного элемента, легко вычислить адрес любого элемента по его смещению.

Сложение векторов

Напишем программу, которая генерирует два вектора целых чисел, копирует их в память GPU, выполняет сложение на GPU, результат копируем обратно в память CPU. GPU обычно не работает на прямую с памятью CPU, хотя это технически возможно. Для начала необходимо выделить память под данные на device, для этого используется фукция cudaMalloc(), после этого копируем данные из памяти host в память device с помощью cudaMemcpy(). Далее для каждого элемента вектора запускается отдельный процесс сложения, т.е. в данном случае – 128 нитей. На CPU это могло быть накладно, но GPU как раз рассчитан на работу с большим количеством потоков, и всё происходит достаточно быстро. После того, как операция завершена с помощью cudaMemcpy(), копируем результат обратно на host и освобождаем память device с помощью cudaFree(). В тексте ядра есть переменная threadIdx, это системная переменная CUDA, содержащая номер нити.

Приведем программу сложения двух векторов с помощью технологии CUDA.

```
#define N 128 //количество элементов массива
// Ядро
```

```
__global__ void add( int *a, int *b, int *c )
{
    //вычисление индекса элемента
    int tid = threadIdx.x;
    //проверка на выход за пределы массива
    if(tid > N-1) return;
    //поэлементное сложение массивов
    c[tid] = a[tid] + b[tid];
}

int main() {
    // выделение памяти под массивы на CPU
    int host_a[N], host_b[N], host_c[N];
    // выделение памяти под массивы для копирования
    // на GPU
    int *dev_a, *dev_b, *dev_c;
    // заполнение массивов
    for (int i=0; i<N; i++)
    {
        host_a[i] = i * i;
        host_b[i] = - i;
    }
    // выделение памяти под массивы на GPU
    cudaMalloc( (void**)&dev_a, N * sizeof(int) );
    cudaMalloc( (void**)&dev_b, N * sizeof(int) );
    cudaMalloc( (void**)&dev_c, N * sizeof(int) );

    // копирование данных в память GPU
    cudaMemcpy( dev_a, host_a, N * sizeof(int),
    cudaMemcpyHostToDevice ) ;
    cudaMemcpy( dev_b, host_b, N * sizeof(int),
    cudaMemcpyHostToDevice ) ;

    // вызов ядра
    add<<<1, N>>>( dev_a, dev_b, dev_c );

    // получаем результат расчета
```

```
    cudaMemcpy( host_c, dev_c, N * sizeof(int),
    cudaMemcpyDeviceToHost ) ;

    // вывод результатов
    for (int i=0; i<N; i++)
    {
        printf( "%d + %d = %d\n", host_a[i],
        host_b[i], host_c[i] );
    }

    // освобождение памяти
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
    return 0;
}
```

Вычисление математических функций

В CUDA существуют аналоги ряда математических функций, включенных в стандартную библиотеку языка Си (<math.h>). Так, например, прототип CUDA-версии синуса (sinf) имеет вид

```
__device__ float __sinf(float x);
```

Перечислим основные различия:

- функции CUDA вычисляются непосредственно на GPU;

- большинство функций работают только с float-аргументами (GPU, как правило, в основном работают с float, точности достаточно для основных задач, возлагаемых на GPU);

- имеют модификатор __device__, вследствие чего могут вызываться только с GPU (т.е. из kernel-функций);

Рассмотрим следующую задачу:

1. Выделить на GPU массив из n = 10^9 элементов типа float.

2. Инициализировать их в ядре следующим образом:

```
arr[i] = sin((i % 360) * pi / 180)
```

где `pi = 3.141592653`.

3. Скопировать массив в память CPU, посчитать ошибку

```
err = sum_i(abs(sin((i % 360) * pi / 180) - arr[i]))
/ n
```

4. Провести исследование при использовании массива типа double.

Основной частью данной задачи является написание ядра, в котором будет инициализироваться массив (и вызываться синус).

```
#define BASE_TYPE float
#define MAX_GRIDSIZE 1

__global__ void sinMass(BASE_TYPE *A, int arraySize)
{
    int index = (blockIdx.y * MAX_GRIDSIZE + blockIdx.x)
    * blockDim.x + threadIdx.x;
    if (index < arraySize)
        [index] = sin((BASE_TYPE)((index % 360) * M_PI /
        180));

    // другие варианты вычисления синуса
    // A[index] = sinf((index % 360) * M_PI / 180);
    // A[index] = __sinf((index % 360) * M_PI
    // / 180);
}
```

Приведем список встроенных математических функций для работы с типом float:

Функция	**Операция**
`__fadd_[rn,rz,ru,rd](x,y),`	x + y
`__fsub_[rn,rz,ru,rd](x,y),`	x – y
`__fmul_[rn,rz,ru,rd](x,y),`	x * y
`__fmaf_[rn,rz,ru,rd](x,y,z)`	x * y + z
`__frcp_[rn,rz,ru,rd](x)`	1 / x
`__fsqrt_[rn,rz,ru,rd](x)`	$\sqrt{x}$
`__frsqrt_rn(x)`	$1 / \sqrt{x}$
`__fdiv_[rn,rz,ru,rd](x,y)`	x / y
`__fdividef(x,y)`	x / y; при $y>2^{126}$ результат = 0
`__expf(x)`	e^x
`__exp10f(x)`	10^x
`__logf(x)`	ln x
`__log2f(x)`	$\log_2 x$
`__log10f(x)`	lg x
`__sinf(x)`	sin x
`__cosf(x)`	cos x
`__sincosf(x,sptr,cptr)`	*sptr = sin x, cptr = cos x
`__tanf(x)`	tg x
`__powf(x, y)`	x^y

Здесь `rn,rz,ru,rd` - способы округления, например, `__fadd_rn` округляет сумму к ближайшему, `__fadd_rz` – к нулю, `__fadd_ru` – до большего, `__fadd_rd` – до меньшего.

Список встроенных функций для работы с типом double несколько меньше:

Функция	**Операция**
`__dadd_[rn,rz,ru,rd](x,y),`	x + y
`__dsub_[rn,rz,ru,rd](x,y),`	x – y
`__dmul_[rn,rz,ru,rd](x,y),`	x * y
`__fma_[rn,rz,ru,rd](x,y,z)`	x * y + z
`__drcp_[rn,rz,ru,rd](x)`	1 / x
`__dsqrt_[rn,rz,ru,rd](x)`	√x
`__ddiv_[rn,rz,ru,rd](x,y)`	x / y

Вопросы к разделу

1. Как организовать обращение к элементу массива в теле ядра?
2. Какие основные этапы работы с векторами можно выделить?
3. Какие встроенные (быстрые) математические функции есть на GPU? В чем отличие их от обычных функций?

Лабораторная работа

1. Используя другие функции вместо синуса (__expf` (e^x), __exp10f` (10^x) и т.д.), рассчитайте относительную ошибку (отношение абсолютной ошибки к значению функции).
2. Напишите программу, реализующую скалярное произведение двух векторов.

Домашняя работа

1. Сравните скорость и точность вычислений с типами float и double для функций из лабораторной работы. Какой тип для каких задач лучше использовать и почему? Сделайте вывод.
2. Напишите программу, реализующую процесс ортогонализации Грама-Шмидта. При этом на хосте создается набор из N векторов вида a_1 = (1,1,1,1,…), a_2 = (0,1,1,1,…), a_3 = (0,0,1,1,…), … , a_N = (…,0,0,0,1). Далее эти вектора передаются на девайс, после чего производится

ортогонализация. Алгоритм процесса ортогонализации (получение ортогональных векторов b_i по a_i) следующий:

$$b_1 = a_1, \qquad b_i = a_i - \sum_{k=1}^{i-1} \frac{(a_i, b_k)}{(b_k, b_k)} b_k, \quad i = 2..N.$$

.

Глава 6. Работа с матрицами

Перед тем, как начать работать с матрицами, скажем несколько слов о типе dim3 – одном из базовых типов CUDA. Он представляет собой не что иное, как вектор из 3-х целочисленных значений. Создается он следующим образом:

```
dim3 v1(512, 2, 4);
dim3 v2(512, 128);
dim3 v3(1024).
```

Если какие-то значения опущены (как в примере с v2 и v3), то они по умолчанию равны 1 (а не 0). То есть, вышеприведенный код эквивалентен следующему:

```
dim3 v1(512, 2, 4);
dim3 v2(512, 128, 1);
dim3 v3(1024, 1, 1).
```

В CUDA есть несколько векторных типов для доступа к данным. Например, такой тип, как [u]int может быть 1-, 2-, 3-, 4-мерным вектором. Для создания переменной такого типа применяется функция специального вида, например:

```
int3 a = make_int3 ( 1, 2, 3);
```

Так же существуют типы (u)char, (u)short, (u)long, float, которые могут быть 1-, 2-, 3-, 4-мерными векторами, а вот типы long long и double могут быть только 1- и 2-мерными.

На основе типа uint3, в CUDA существует специальный тип dim3, который имеет конструктор. Конструктор этого типа позволяет задавать не все компоненты 3-мерного вектора. Недостающие компоненты этого типа в конструкторе инициализируются единицами. Главное назначение типа dim3 – это задание параметров запуска ядра.

Внутри каждого вычислительного ядра содержатся встроенные переменные, которые используют вышеописанные типы. С помощью этих

встроенных переменных можно оперировать отдельными нитями и различать их. Эти переменные были описаны выше, кроме переменной ***warpSize*** типа int – эта переменная содержит значение размера связки нитей (варп), которое на текущий момент всегда равно 32.

Используя встроенные переменные, можно задавать разные параметры запуска ядра и различать отдельные нити в блоках или сами блоки.

Создание матриц

Создадим на CPU матрицу A размером 10x10 и зададим элементы матрицы следующим образом:

$$A = \begin{pmatrix} 0 & 1 & 2 & ... & 9 \\ 10 & 11 & 12 & ... & 19 \\ 20 & 21 & 22 & ... & 29 \\ ... & ... & ... & ... & ... \\ 90 & 91 & 92 & ... & 99 \end{pmatrix}$$

Создадим на GPU матрицу B размером 10x10. Элементы матрицы B зададим такими же, как и у матрицы A. При этом не будем использовать копирование данных с хоста на девайс. Необходимо скопировать матрицу B с девайса на хост и сравнить на хосте две матрицы – A и B. Нужно добиться, чтобы обе матрицы совпадали. Попробуйте запускать программу с разными размерами grid и block.

Для решения этой задачи используем один блок с десятью нитями так, чтобы каждая нить создавала один элемент матрицы. Для удобства, блок нитей мы сделаем двумерным. Таким образом, индекс элемента матрицы вычислим как threadIdx.y * n + threadIdx.x. В итоге код ядра будет выглядеть так:

```
__global__ void createMatrix(int *A, const int n)
{
```

```
    A[threadIdx.y * n + threadIdx.x] = 10 *
    threadIdx.y + threadIdx.x;
}
```

На CPU матрицу A будем создавать с помощью двойного цикла следующим образом:

```
size_t size = n * n * sizeof(int);
int *h_A = (int *)malloc(size);
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        h_A[j * n + i] = 10 * j + i;
```

Вызов ядра, описанного выше, осуществляется таким образом:

```
dim3 threadsPerBlock = dim3(10, 10);
dim3 blocksPerGrid = dim3(1);

createMatrix<<<blocksPerGrid, threadsPerBlock>>>(d_B,
n);
```

Проверку равенства двух матриц осуществляем следующим образом с помощью двойного цикла:

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
    if (h_A[j * n + i] != h_B[j * n + i]) {
        printf("h_A[%d] != h_B[%d]\n", j * n + i, j
        * n + i);
    }
```

Листинг кода.

```
// Сравнение двух матриц
#include "cuda_runtime.h"
```

```
#include "device_launch_parameters.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
// Ядро
__global__ void createMatrix(int *A, const int n)
{
    // Создание элементов матрицы на GPU
    A[threadIdx.y * n + threadIdx.x] = 10 *
    threadIdx.y + threadIdx.x;
}
int main()
{
    // кол-во строк и столбцов матрицы
    const int n = 10;

    // размер матрицы
    size_t size = n * n * sizeof(int);

    // выделяем память для матрицы на CPU
    int *h_A = (int *)malloc(size);

    // инициализируем матрицу
    for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        h_A[j * n + i] = 10 * j + i;
    int *d_B = NULL;
    // выделяем память для матрицы на GPU
    cudaMalloc((void **)&d_B, size);

    // определение размеров сетки и блоков
    dim3 threadsPerBlock = dim3(10, 10);
    dim3 blocksPerGrid = dim3(1);

    // вызов ядра
    createMatrix<<<blocksPerGrid,
    threadsPerBlock>>>(d_B, n);
```

```
    // выделяем память для матрицы B, чтобы
    // скопировать из GPU на CPU
    int *h_B = (int *)malloc(size);

    // копируем матрицу из GPU на CPU
    cudaMemcpy(h_B, d_B, size,
    cudaMemcpyDeviceToHost);

    // проверяем совпадение матрицы A и матрицы B
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            if (h_A[j * n + i] != h_B[j * n + i]) {
                printf("h_A[%d] != h_B[%d]\n", j * n
                + i, j * n + i);
            }

    // освобождаем память на GPU
    cudaFree(d_B);

    // освобождаем память на CPU
    free(h_A);
    free(h_B);
    return 0;
}
```

Транспонирование матрицы

Для решения этой задачи используем блоки размером 16x16. Каждая нить блока создает один элемент транспонированной матрицы. Код ядра будет выглядеть так:

```
__global__ void matrixTranspose (const BASE_TYPE *A,
BASE_TYPE *AT, int rows, int cols)
{
    // Индекс элемента в исходной матрице
```

```
    int iA = cols * (blockDim.y * blockIdx.y +
    threadIdx.y) + blockDim.x * blockIdx.x +
    threadIdx.x;

    // Индекс соответствующего элемента в
    // транспонированной матрице
    int iAT = rows * (blockDim.x * blockIdx.x +
    threadIdx.x) + blockDim.y * blockIdx.y +
    threadIdx.y; AT[iAT] = A[iA];
}
```

Для удобства вычислений размеры исходной матрицы делаем кратными размеру блока. Для этого используем функцию toMultiple(int a, int b), которая возвращает число, кратное b и наиболее близкое к a. Матрицу инициализируем случайными числами.

```
// Инициализация матрицы
for (int i = 0; i < rows * cols; ++i)
{
    h_A[i] = rand()/(BASE_TYPE)RAND_MAX;
}
```

Вызов ядра, описанного выше, осуществляется таким образом:

```
dim3 threadsPerBlock = dim3(BLOCK_SIZE, BLOCK_SIZE);

dim3 blocksPerGrid = dim3(cols / BLOCK_SIZE, rows /
BLOCK_SIZE);

//вызов ядра
matrixTranspose<<<blocksPerGrid,
threadsPerBlock>>>(d_A, d_AT, rows, cols);
```

Осуществляем проверку правильности работы ядра.

```
for (int i = 0; i < rows; i++)
    for (int j = 0; j < cols; j++)
        if (h_A[i * cols + j] != h_AT[j * rows + i])
        {
            fprintf(stderr, "Result verification
            failed at element [%d, %d]!\n", i, j);
            exit(EXIT_FAILURE);
        }
```

Листинг кода:

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <stdlib.h>

// размер блока
#define BLOCK_SIZE 16

// тип, который будут иметь элементы матриц
#define BASE_TYPE float

// Ядро
// Функция транспонирования матрицы
__global__ void matrixTranspose(const BASE_TYPE *A,
BASE_TYPE *AT, int rows, int cols)
{
    // Индекс элемента в исходной матрице
    int iA = cols * (blockDim.y * blockIdx.y +
    threadIdx.y) + blockDim.x * blockIdx.x +
    threadIdx.x;

    // Индекс соответствующего элемента в
    // транспонированной матрице
    int iAT = rows * (blockDim.x * blockIdx.x +
    threadIdx.x) + blockDim.y * blockIdx.y +
    threadIdx.y;
```

```
    AT[iAT] = A[iA];
}

// Функция вычисления числа, которое больше числа a
// и кратное числу b
int toMultiple(int a, int b)
{
    int mod = a % b;
    if (mod != 0)
    {
        mod = b - mod;
        return a + mod;
    }
    return a;
}

int main()
{
    // Объекты событий
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    // Количество строк и столбцов матрицы
    int rows = 1000;
    int cols = 2000;

    // Меняем количество строк и столбцов матрицы
    // на число, кратное размеру блока (16)
    rows = toMultiple(rows, BLOCK_SIZE);
    printf("rows = %d\n", rows);
    cols = toMultiple(cols, BLOCK_SIZE);
    printf("cols = %d\n", cols);
    size_t size = rows * cols * sizeof(BASE_TYPE);

    // Выделение памяти под матрицы на хосте
    // Исходная матрица
    BASE_TYPE *h_A = (BASE_TYPE *)malloc(size);
```

```
// Транспонированная матрица
BASE_TYPE *h_AT = (BASE_TYPE *)malloc(size);

// Инициализация матрицы
for (int i = 0; i < rows * cols; ++i)
{
    h_A[i] = rand()/(BASE_TYPE)RAND_MAX;
}
// Выделение глобальной памяти на девайсе
// для исходной матрицы
BASE_TYPE *d_A = NULL;
cudaMalloc((void **)&d_A, size);

// Выделение глобальной памяти на девайсе для
// транспонированной матрицы
BASE_TYPE *d_AT = NULL;
cudaMalloc((void **)&d_AT, size);

// Копируем матрицу из CPU на GPU
cudaMemcpy(d_A, h_A, size,
cudaMemcpyHostToDevice);

// Определяем размер блока и сетки
dim3 threadsPerBlock = dim3(BLOCK_SIZE,
BLOCK_SIZE);
dim3 blocksPerGrid = dim3(cols / BLOCK_SIZE,
rows / BLOCK_SIZE);

// Начать отсчета времени
cudaEventRecord( start, 0);

// Запуск ядра
matrixTranspose<<<blocksPerGrid,
threadsPerBlock>>>(d_A, d_AT, rows, cols);

// Окончание работы ядра, остановка времени
cudaEventRecord( stop, 0);
cudaEventSynchronize( stop );
```

```
    float KernelTime;
    cudaEventElapsedTime( &KernelTime, start, stop);
    printf("KernelTime: %.2f milliseconds\n",
    KernelTime);

    // Копируем матрицу из GPU на CPU
    cudaMemcpy(h_AT, d_AT, size,
    cudaMemcpyDeviceToHost);

    // Проверка правильности работы ядра
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++)
        {
            if (h_A[i * cols + j] != h_AT[j * rows +
            i])
            fprintf(stderr, "Result verification
            failed at element [%d, %d]!\n", i, j);
            exit(EXIT_FAILURE);
        }
    printf("Test PASSED\n");

    // Освобождаем память на GPU
    cudaFree(d_A);
    // Освобождаем память на GPU
    cudaFree(d_AT);
    // Освобождаем память на CPU
    free(h_A);
    free(h_AT);
    // Удаляем объекты событий
    cudaEventDestroy( start );
    cudaEventDestroy( stop );
    printf("Done\n");
    return 0;
}
```

Сложение матриц

Поскольку сложение матриц выполняется поэлементно (еще проще, чем умножение), то все сказанное в следующем разделе (по поводу параллельного вычисления и представления массивов в памяти) справедливо и для сложения матриц.

Для решения задачи используем функцию __global__ void matrixAdd.

```
// Ядро
__global__ void matrixAdd(const BASE_TYPE *A, const
BASE_TYPE *B, BASE_TYPE *C, int cols)
{
    // Вычисление индекса элемента матрицы на GPU
    int ind = cols * (blockDim.y * blockIdx.y +
    threadIdx.y) + blockDim.x * blockIdx.x + threadIdx.x;

    C[ind] = A[ind] + B[ind];
}

// Функция вычисление числа, которое больше
// числа a и кратное числу b
int toMultiple(int a, int b)
{
     int mod = a % b;
     if (mod != 0)
     {
     mod = b - mod;
     return a + mod;
     }
     return a;
}
```

Умножение матриц

Пусть даны матрицы $A = (a_{ij})$ и $B = (b_{ij})$ размерами $m \times n$ и $n \times p$ соответственно. Тогда произведение $C = (c_{ij}) = AB$ размерностью $m \times p$, как известно, определяется по формуле

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

Элементы матрицы-произведения зависят только от элементов исходных матриц , поэтому их вычисление может выполняться параллельно, не требуя дополнительных "ухищрений". Поэтому рассмотрим лишь вопрос о вариантах представления матрицы в памяти – в виде двухмерного (например, float** a, затем обращение вида a[i][j]) или одномерного массива (float* a и затем обращение вида a[i * N + j], где N - длина строки (количество столбцов)).

Рассмотрим вопрос о том, как работает операция индексирования:

1) в случае с одномерным массивом:
   a) вычисляется выражение i * N + j (в современных CPU это выполняется за одну операцию, так называемую fused multiply-add);
   b) полученное число (смещение) умножается на размер типа и складывается с базовым адресом (адресом начала массива);
   c) как правило, переменные i, j и базовый адрес будут храниться в регистрах CPU, поэтому вышеприведенные операции выполняются быстро.
2) в случае с двухмерным массивом:
   a) значение i умножается на размер указателя и складывается с базовым адресом, вычисляется адрес a[i] (начала i-й строки);
   b) Значение j умножается на размер элемента и складывается с адресом a[i], вычисляется адрес a[i][j];
   c) Адрес a[i], очевидно, не хранится в регистре, поэтому операция с ним будет медленнее.

С другой стороны, одномерный массив требует большого *последовательного* участка памяти.

Для решения задачи опишем функцию __global__ void matrixMult.

```
__global__ void matrixMult(const BASE_TYPE *A, const
BASE_TYPE *B, BASE_TYPE *C, int Acols, int Bcols)
{
    int i0 = Acols * (blockDim.y * blockIdx.y +
    threadIdx.y);
    int j0 = blockDim.x * blockIdx.x + threadIdx.x;
    BASE_TYPE sum = 0;

    for (int k = 0; k < Acols; k++)
        sum += A[i0 + k] * B[k * Bcols + j0];

    int ind = Bcols * (blockDim.y * blockIdx.y +
    threadIdx.y) + blockDim.x * blockIdx.x + threadIdx.x;

    C[ind] = sum;
}
```

Листинг кода:

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <stdlib.h>

#define BLOCK_SIZE 16
// тип, который будут иметь элементы матриц
#define BASE_TYPE double
// функция перемножения матриц
__global__ void matrixMult(const BASE_TYPE *A, const
BASE_TYPE *B, BASE_TYPE *C, int Acols, int Bcols)
{
    int i0 = Acols * (blockDim.y * blockIdx.y +
    threadIdx.y);
    int j0 = blockDim.x * blockIdx.x + threadIdx.x;
    BASE_TYPE sum = 0;
```

```
    for (int k = 0; k < Acols; k++)
        sum += A[i0 + k] * B[k * Bcols + j0];

    int ind = Bcols * (blockDim.y * blockIdx.y +
    threadIdx.y) + blockDim.x * blockIdx.x + threadIdx.x;

    C[ind] = sum;
}

int toMultiple(int a, int b) {

    int mod = a % b;

    if (mod != 0) {
        mod = b - mod;
        return a + mod;
    }
    return a;
}

int main()
{
    //start, stop - for Kernel time
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    // количество строк и столбцов матрицы
    int Arows = 100;
    int Acols = 200;
    int Brows = Acols;
    int Bcols = 150;

    Arows = toMultiple(Arows, BLOCK_SIZE);
    printf("Arows = %d\n", Arows);

    Acols = toMultiple(Acols, BLOCK_SIZE);
    printf("Acols = %d\n", Acols);
```

```
Brows = toMultiple(Brows, BLOCK_SIZE);
printf("Brows = %d\n", Brows);

Bcols = toMultiple(Bcols, BLOCK_SIZE);
printf("Bcols = %d\n", Bcols);

size_t Asize = Arows * Acols * sizeof(BASE_TYPE);
size_t Bsize = Brows * Bcols * sizeof(BASE_TYPE);
size_t Csize = Arows * Bcols * sizeof(BASE_TYPE);

BASE_TYPE *h_A = (BASE_TYPE *)malloc(Asize);
BASE_TYPE *h_B = (BASE_TYPE *)malloc(Bsize);
BASE_TYPE *h_C = (BASE_TYPE *)malloc(Csize);

for (int i = 0; i < Arows * Acols; ++i) {
    h_A[i] = rand()/(BASE_TYPE)RAND_MAX;
}
for (int i = 0; i < Brows * Bcols; ++i) {
    h_B[i] = rand()/(BASE_TYPE)RAND_MAX;
}

BASE_TYPE *d_A = NULL;
cudaMalloc((void **)&d_A, Asize);

BASE_TYPE *d_B = NULL;
cudaMalloc((void **)&d_B, Bsize);

BASE_TYPE * d_C = NULL;
cudaMalloc((void **)&d_C, Csize);
cudaMemcpy(d_A, h_A, Asize, cudaMemcpyHostToDevice);

cudaMemcpy(d_B, h_B, Bsize, cudaMemcpyHostToDevice);

dim3 threadsPerBlock = dim3(BLOCK_SIZE, BLOCK_SIZE);
dim3 blocksPerGrid = dim3(Bcols / BLOCK_SIZE, Arows /
BLOCK_SIZE);
```

```
cudaEventRecord( start, 0);

matrixMult<<<blocksPerGrid, threadsPerBlock>>>(d_A,
d_B, d_C, Acols, Bcols);

cudaEventRecord( stop, 0);
cudaEventSynchronize( stop );
float KernelTime;
cudaEventElapsedTime( &KernelTime, start, stop);
printf("KernelTime: %.2f milliseconds\n",
KernelTime);

cudaMemcpy(h_C, d_C, Csize, cudaMemcpyDeviceToHost);

printf("Test STARTED\n");
for (int i = 0; i < Arows; i++) {
    for (int j = 0; j < Bcols; j++) {
        BASE_TYPE sum = 0;
        for (int k = 0; k < Acols; k++)
            sum += h_A[i * Acols + k] * h_B[k *
            Bcols + j];

        if (fabs(h_C[i * Bcols + j] - sum) > 1e-3)
        {
            fprintf(stderr, "Result verification
            failed at element [%d, %d]!\n", i, j);
            printf("sum = %f, h_C[i * Bcols + j] =
            %f\n", sum, h_C[i * Bcols + j]);
            exit(EXIT_FAILURE);
        }
    }
}
printf("Test PASSED\n");

cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
free(h_A);
```

```
    free(h_B);
    free(h_C);
    cudaEventDestroy( start );
    cudaEventDestroy( stop );

    return 0;
}
```

Вопросы к разделу

1. Чем отличается создание матрицы на GPU от создания матрицы на CPU и последующего копирования ее элементов на GPU?
2. В каких участках кода при операциях над матрицами возникают потери в производительности?

Лабораторная работа

1. Проверьте, является ли данная квадратная матрица ортогональной, путем умножения на транспонированную (не выделяя под нее дополнительной памяти) и сравнения результата с единичной матрицей.

Домашняя работа

1. Проверьте, являются ли две заданные квадратные матрицы A и B коммутирующими (т.е. их произведение не зависит от порядка умножения: AB = BA).
2. Напишите функцию для сложения двух матриц M×N, заданных как вектор векторов (float a[M][N], b[M][N]). Матрицы задаются на хосте, после копируются на девайс, и там производится сложение.

Глава 7. Типы памяти. Разделяемая память

В CUDA предусмотрены несколько типов памяти, каждый из которых предназначен под определенные нужды. Ниже приведена таблица типов памяти GPU.

Таблица 3. *Типы памяти GPU*

Тип памяти	**Доступ**	**Выделяется**	**Скорость работы**
Регистры	чтение/запись	на поток	Высокая
Локальная	чтение/запись	на поток	Низкая
Разделяемая	чтение/запись	на блок	Высокая
Глобальная	чтение/запись	на все решетки	Низкая
Константная	чтение	на все решетки	Низкая
Текстурная	чтение	на все решетки	Низкая

Регистры используется для хранения локальных переменных. Скорость доступа для данного типа памяти самая быстрая, и выделение происходит на каждую нить. **Локальная память** используется для хранения локальных переменных, когда регистров не хватает, скорость доступа низкая. Память выделяется так же, как и для регистров. **Разделяемая память** используется для хранения массивов данных, используемых совместно всеми нитями в блоке. Эта память имеет чуть меньшую скорость доступа, чем регистры, и выделяется на блок. В **глобальной памяти** хранятся исходно все данные, это общая память для всех блоков. **Константная память** используется для передачи параметров в ядро, превышающих допустимые размеры для параметров ядра, выделение целиком на решетку. **Текстурная память** используется для хранения больших массивов данных, выделяется целиком на решетку.

Обращение к памяти в CUDA осуществляется одновременно из половины количества нитей в блоке. Все типы памяти, кроме регистровой и локальной,

можно использовать как эффективно, так и не эффективно. Поэтому необходимо четко понимать их особенности и области применения.

Рассмотрим более подробно разделяемую память. Основной особенностью разделяемой памяти, является то, что она оптимизирована под обращение к ней одновременно половины количества нитей в блоке (16 нитей). Вся разделяемая память разбита на 32 банка, состоящих из 32 битовых слов, причем, последовательно идущие слова попадают в последовательно идущие блоки. Обращение к каждому банку происходит независимо. Второй важной особенностью разделяемой памяти является то, что она выделяется на блок. Для задания разделяемой памяти используется спецификатор ***__shared__***. Часто использование разделяемой памяти требует явной синхронизации внутри блока.

При работе с разделяемой памятью необходимо обеспечить синхронизацию потоков, что в CUDA довольно просто – она предоставляет примитивный барьер, функцию __syncthreads(). При ее вызове потоки приостанавливаются до того момента, пока все потоки не достигнут этой точки. Важно, чтобы вызов __syncthreads() у всех потоков находился в одном и том же месте (в частности, был безусловным, т. е. не находился внутри оператора if и т.п.), в противном случае результаты могут быть непредсказуемыми, в частности, может возникнуть "мертвая блокировка" (deadlock).

Приведем фрагмент кода создания массива данных в разделяемой памяти.

```
#define BS 256 //Размер блока

__global__ void Kernel (float* data)
{
// Создание массива в разделяемой памяти
    __shared__ float a[BS];
    int i = blockIdx.x * BS + threadIDx.x;
// Копирование из глобальной памяти в разделяемую
    a[i] = data[i];
// Синхронизируем нити.
```

```
// Перед использованием нужно быть уверенным, что все
данные скопированились
    __syncthreads();
// Использонание данных разделяемой памяти
}
```

Данный пример демонстрирует только работу с разделяемой памятью. Для того, чтобы понять, как же разделяемая память помогает в ускорении работы напишем программу для вычисления скалярного произведения векторов. Попробуем написать эту программу с использованием глобальной памяти. Используя механизм событий, замерим время работы ядра.

Рассмотрим, как изменится ядро, если, вместо глобальной памяти, использовать разделяемую память.

```
__global__ void scalMult(const BASE_TYPE *A, const
BASE_TYPE *B, BASE_TYPE *C, int numElem)
{
// Переменная для хранения суммы элементов
    BASE_TYPE sum = 0.0;
// Создание массивов в разделяемой памяти
    __shared__ BASE_TYPE ash[BLOCK_SIZE];
    __shared__ BASE_TYPE bsh[BLOCK_SIZE];
// Копирование из глобальной памяти
    ash[threadIdx.x] = A[blockIdx.x * blockDim.x +
    threadIdx.x];
    bsh[threadIdx.x] = B[blockIdx.x * blockDim.x +
    threadIdx.x];
// Синхронизация нитей
    __syncthreads();
// Вычисление скалярного произведения
    if (threadIdx.x == 0)
    {
        sum = 0.0;
        for (int j = 0; j < blockDim.x; j++)
            sum += ash[j] * bsh[j];
```

```
            C[blockIdx.x] = sum;
        }
}
```

Заменим ядро программы и оценим время работы ядра, использующего разделяемую память. Так как каждая нить теперь работает с памятью, чья скорость работы более высокая, большую часть времени работы программы занимают вычисления, а не доступ к памяти. Более наглядно это можно наблюдать в случае перемножения матриц.

Банки данных

Доступ к памяти осуществляется через 32 банка данных. Причем номер банка можно посчитать по формуле

Номер банка = (Адрес в байтах / 4) % 32.

Каждый банк в определенный момент времени обрабатывает одну транзакцию (чтение или запись). Оптимальный доступ к данным обеспечивается, когда каждая нить последовательно соотносится со своим банком (см. Рис. 6а).

Также конфликта между банками нет, если каждой нити в любом порядке соответствует свой банк. Пример такого доступа изображен на Рис. 6б. Конфликт имеет место, когда несколько нитей пытаются обратиться к одному банку данных, как изображено на Рис. 6в. Однако, в случае, когда все нити записывают или считывают с одного банка, то конфликта не происходит.

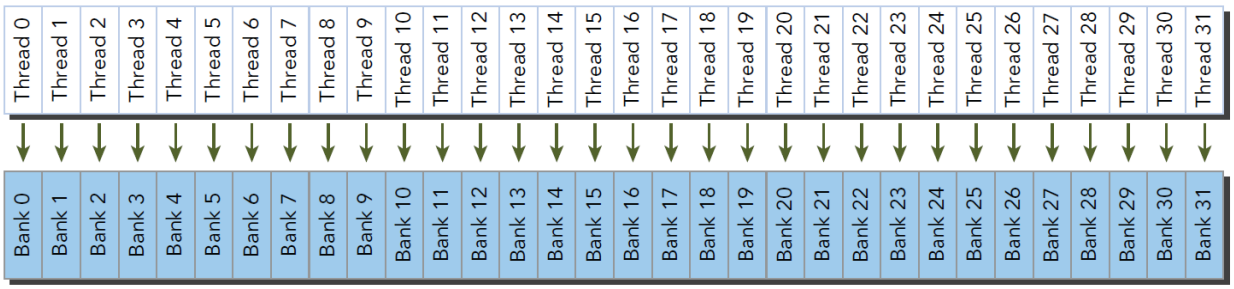

Рис. 6а. *Оптимальный доступ к банкам данных*

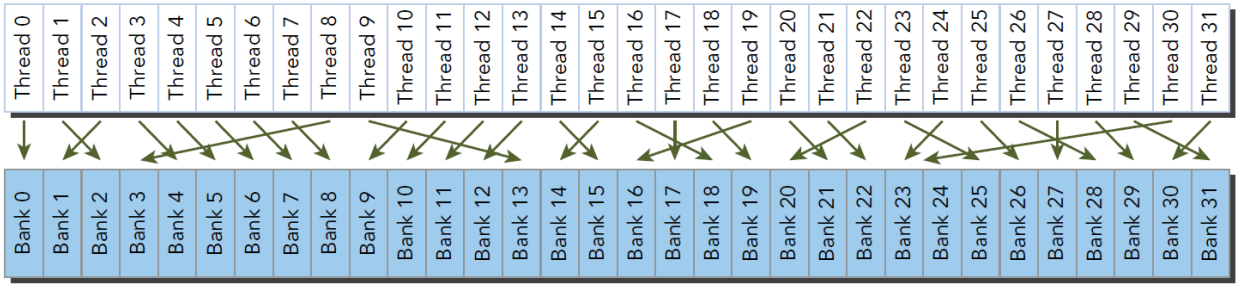

Рис. 6б. *Бесконфликтный доступ к банкам данных*

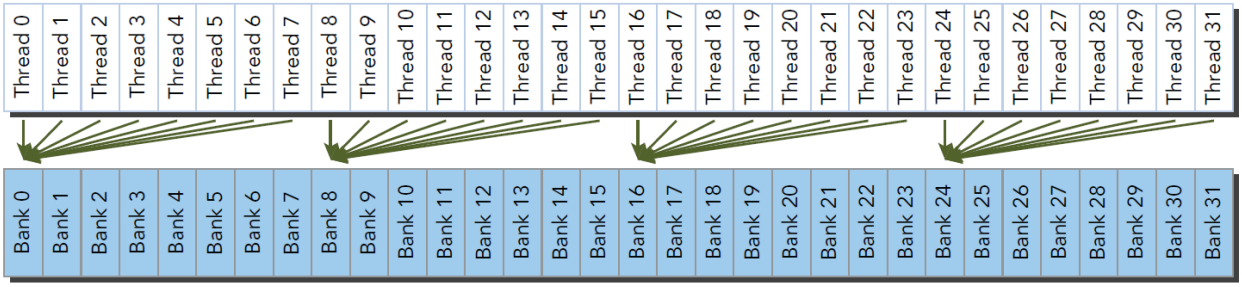

Рис. 6в. *Конфликтный доступ к банкам данных*

Рассмотрим наиболее распространенные случаи, когда могут иметь место конфликты банков данных. В следующем примере

```
__shared__ char s[N];
char data = s[threadIdx.x];
```

имеем однобайтовый массив, и, так как в одном банке содержится четыре байта, то одновременно в один банк будут обращаться четыре нити. Это приведет к конфликту четвертого порядка (4 нити на 1 банк). Отметим, что на Рис. 6в приведен пример конфликтов шестого порядка.

Один из способов решения такого конфликта состоит в добавлении избыточных данных:

```
__shared__ char s[4 * N];
char data = s[threadIdx.x * 4];
```

В следующем примере будет конфликт второго порядка (`short` – двухбайтовый тип)

```
__shared__ short s[N];
short data = s[threadIdx.x];
```

Чаще возникают конфликты, когда используются четырехбайтовые числа, и номер в массиве вычисляется по формуле. Например, в следующем коде:

```
__shared__ float s[N];
float data = s[threadIdx.x * L];
```

при четных `L` возникают конфликты. Действительно, при `L` = 2 нулевая нить и 16-натая нить обращаются в нулевой банк, а при `L` = 4 уже нити с номерами 0, 8, 16 и 24 обращаются в нулевой банк, создавая конфликт четвертого порядка.

Пример. Оптимизация программы по перемножению двух матриц

Пусть даны две матрицы размером $N * N$ (для удобства будем считать, что N кратно 16). Если решать данную задачу, получая данные постоянно из глобальной памяти, то понадобится $2N$ чтений из глобальной памяти. Становится ясно, что основным тормозящим программу фактором является не что иное, как чтение из глобальной памяти, которая имеет низкую скорость передачи.

Заметно ускорить программу поможет правильная организация вычислений и разделяемая память. Обратим внимание на то, что для вычисления элемента новой матрицы необходимы лишь небольшие части исходных матриц. Для повышения быстродействия программы разобьем новую матрицу на подматрицы таким образом, чтобы обработкой данной подматрицы занимался только один блок. В данной задаче разобьем на подматрицы (C') размером 16*16 элементов (см. Рис. 7).

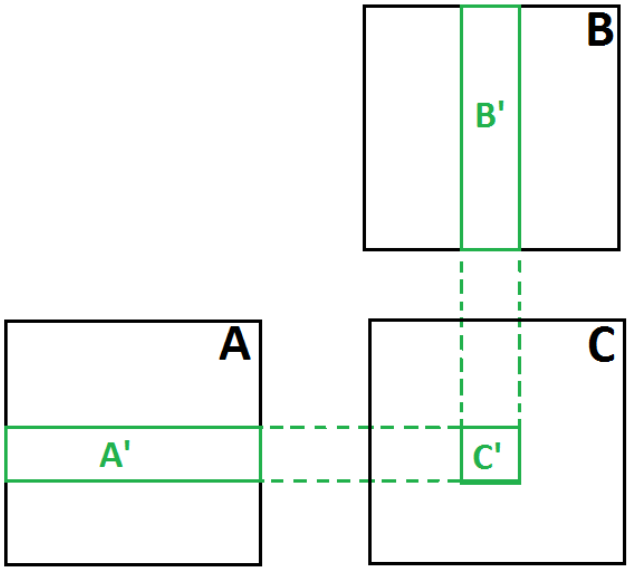

Рис. 7. *Вычисление подматрицы новой матрицы*

Однако, из-за малого объема разделяемой памяти, подматрицы A' и B' не могут быть целиком скопированы. Поступим с A' и B' так же, как поступили с подматрицей C', разобьем каждую матрицу на подматрицы размером 16*16. В итоге вычисление C' будем проводить в $N/16$ шагов (см. Рис. 8).

Вычисление подматрицы C' можно записать следующим образом:

$$C' = A'_1 * B'_1 + A'_2 * B'_2 + \cdots + A'_{\frac{N}{16}} * B'_{\frac{N}{16}}$$

Теперь на каждом шаге будем подгружать в разделяемую память по одной подматрице A'_i и B'_i, вычислять соответствующую им сумму для элементов произведения и по окончанию всех вычислений записывать в результирующую матрицу.

Укажем в начале программы размер блока и тип элементов массива:

```
// размер блока или размер подматрицы
#define BLOCK_SIZE 16
//тип, который будут иметь элементы матриц
#define BASE_TYPE double
```

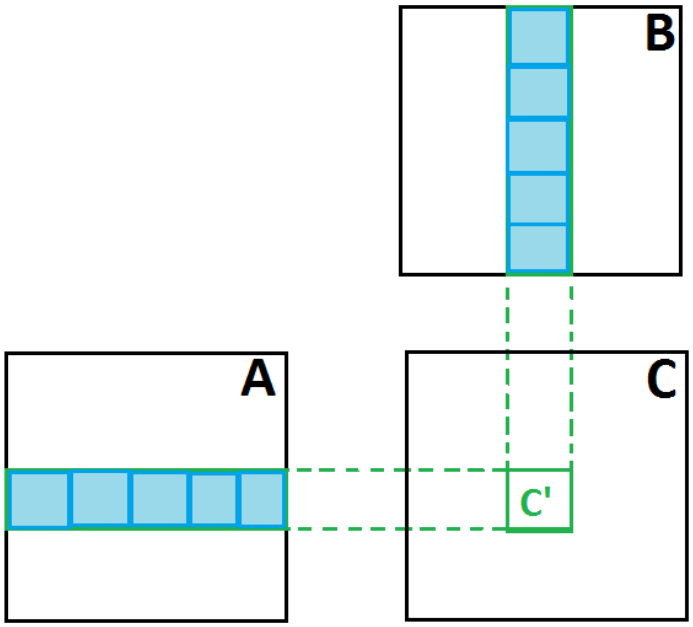

Рис. 8. *Разбиение исходных матриц*

Учитывая все вышесказанное об организации вычислений, ядро будет выглядеть следующим образом:

```
__global__ void matrixMult(const BASE_TYPE *A, const
 BASE_TYPE *B, BASE_TYPE *C, int Acols, int Bcols)
{
// индекс начала первой подматрицы A, которую
// обрабатывает блок
     int aBegin = Acols * blockDim.y * blockIdx.y;
 // индекс конца подматрицы A, которую обрабатывает блок
     int aEnd = aBegin + Acols - 1;
// шаг для перебора подматриц A
     int aStep = blockDim.x;
// индекс начала первой подматрицы B, которую
// обрабатывает блок
     int bBegin = blockDim.x * blockIdx.x;
// шаг для перебора подматриц B
     int bStep = blockDim.y * Bcols;

 // Выделение разделяемой памяти для подматриц
     __shared__ BASE_TYPE as [BLOCK_SIZE][BLOCK_SIZE];
     __shared__ BASE_TYPE bs [BLOCK_SIZE][BLOCK_SIZE];
```

```
// переменная для вычисления элемента подматрицы
    BASE_TYPE sum = 0.0;

    for (int ia = aBegin, ib = bBegin; ia < aEnd; ia +=
    aStep, ib += bStep)
    {
// загрузка подматриц A и B из глобальной памяти в
// разделяемую
        as[threadIdx.y][threadIdx.x] = A[ia + Acols *
        threadIdx.y + threadIdx.x];
        bs[threadIdx.y][threadIdx.x] = B[ib + Bcols *
        threadIdx.y + threadIdx.x];

// синхронизация нитей
        __syncthreads();

// перемножение двух матриц
        for (int k = 0; k < blockDim.x; k++)
            sum += as[threadIdx.y][k] *
            bs[k][threadIdx.x];

// синхронизация нитей
        __syncthreads();
    }

// индекс результирующего элемента в глобальной памяти
    int ind = Bcols * (blockDim.y * blockIdx.y +
    threadIdx.y) + blockDim.x * blockIdx.x + threadIdx.x;

// запись элемента в глобальную память
    C[ind] = sum;
}
```

После любой работы с разделяемой памятью необходимо синхронизировать нити. В данном примере важно произвести синхронизацию до момента начала расчетов, чтобы убедиться в том, что все данные загружены,

а также после расчетов, чтобы убедиться в том, что текущие подматрицы не нужны, и можно подгружать новые подматрицы.

Вызов ядра из главной функции:

```
// параметры запуска ядра
dim3 threadsPerBlock = dim3(BLOCK_SIZE, BLOCK_SIZE);
dim3 blocksPerGrid = dim3(Bcols / BLOCK_SIZE, Arows / 
BLOCK_SIZE);

// старт времени работы ядра
cudaEventRecord( start, 0);

// вызов ядра
matrixMult<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, 
d_C, Acols, Bcols);

// стоп времени работы ядра
cudaEventRecord( stop, 0);
cudaEventSynchronize( stop );

// вычисление времени работы
float KernelTime;
cudaEventElapsedTime( &KernelTime, start, stop);
```

Использование такой алгоритм для перемножении двух матриц требует 2 * N / 16 чтений из глобальной памяти. Понятно, что правильное использование разделяемой памяти может поднять быстродействие программы даже более чем на порядок. Также метод вычисления суммы произведений элементов по блокам, который был применен в ядре для вычисления произведения двух матриц, ускоряет вычисления и часто применяется в подобных задачах.

Если размер разделяемой памяти не задан изначально, и определяется в процессе работы программы, то необходимо его указывать при вызове ядра

```
myKernelFunc<<<gridSize, blockSize,
sharedMemSize>>>(float *param1, int *param2)
```

где `sharedMemSize` – размер дополнительной памяти, выделяемой при запуске ядра,

Вопросы к разделу

1. Какие типы памяти выделяют в технологии CUDA?
2. Какой спецификатор применяется для задания разделяемой памяти? Перечислите основные особенности разделяемой памяти.

Лабораторная работа

1. Напишите программу для скалярного умножения двух векторов, используя разделяемую память. Разделяемая память выделяется с учетом ограничения ее размеров на блок. Количество нитей при вызове ядра равно числу элементов массива с разделяемой памятью. Замерьте время работы ядра программы.
2. Напишите программу вычисления определенного интеграла функции одной переменной по квадратурной формуле центральных прямоугольников.

Домашняя работа

1. Доработайте программу вычисления определенного интеграла, используя другие квадратурные формулы: формулы трапеций и Симпсона. Сравните точность результатов.
2. Напишите программу вычисления евклидовой нормы вектора (путем скалярного умножения его на себя и вычисления корня из произведения).

Глава 8. Константая и текстурная память

Помимо использования разделяемой памяти существует еще два вида памяти, которые можно использовать в CUDA – это константная и текстурная память. Эти виды памяти являются медленнее разделяемой, но за счет кэширования они превосходят по скорости глобальную память.

Константная память

Константная память используется тогда, когда в ядро необходимо передать много различных данных, которые будут одинаково использоваться всеми нитями ядра. Эта память выделяется в глобальной памяти в пределах 64 кБ. При этом кэширование происходит отдельного для каждого мультипроцессора, размер кэша равен 8 кБ для всех версий производительности кроме 6.0, для 6.0 объем кэша составляет 4 кБ.

Объявление переменной делается глобально, используя слово `__constant__`. Пример использования константного массива:

```
__device__ __constant__ float contsData[4];
float hostData[4];
……
// присваивание значений в hostData
// копирование данных с центрального процессора в
// константную память
cudaMemcpyToSymbol (constData, hostData, 4*sizeof(float),
0, cudaMemcpyHostToDevice);
```

Слово `__device__` перед `__constant__` можно опускать, так как по умолчанию константная память выделяется на графическом устройстве. Использование внутри ядра константной памяти ничем не отличается от использования любой глобальной переменной. Рассмотрим следующий пример использования константы:

```
__constant__ int add;

// функция сложения двух векторов с прибавлением
// постоянной add
__global__ void addKernel(int *c, const int *a, const int
*b)
{
    int i = threadIdx.x;
    c[i] = a[i] + b[i] + add;
}
……
// инициализация и вызов функции сложения в основной
// программе, size – размер массивов
int r = 2;
cudaMemcpyToSymbol(add, &r, sizeof(int), 0,
cudaMemcpyHostToDevice);

addKernel<<<1, size>>>(dev_c, dev_a, dev_b);
```

Текстурная память

При рассмотрении разделяемой памяти, мы предполагали, что использование памяти внутри блока сильно локализовано, то есть что блоку для его работы требуется лишь какой-то определенный участок массива, если же локализовать обращение не удается, а скорость обращения к памяти является узким местом программы, можно попробовать использовать текстурную память.

Основная особенность текстурной памяти – это использования кэша. Однако существуют дополнительные стадии конвейера (преобразование адресов, фильтрация, преобразование данных), которые снижают скорость первого обращения. Поэтому текстурную память разумно использовать с следующих случаях:

- Объем данных не влезает в shared память;

- Паттерн доступа хаотичный;
- Данные переиспользуются разными потоками.

Для использования текстурной памяти необходимо задать объявление текстуры как глобальную переменную:

```
texture < type, dim, tex_type> g_TexRef ;
```

Type – тип хранимых переменных, dim – размерность текстуры (1, 2, 3) Tex_type – тип возвращаемых значений

- cudaReadModeNormalizedFloat,
- cudaReadModeElementType.

Кроме того, для более полного использования возможностей текстурной памяти можно задать описание канала:

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

Задает формат возвращаемого значения

int x, y, z, w – числа в интервале [0, 32], проекция исходного значения по битам

cudaChannelFormatKind – тип возвращаемого значения

- cudaChannelFormatKindSigned – знаковые int
- cudaChannelFormatKindUnsigned – беззнаковые int
- cudaChannelFormatKindFloat – float

Текстурная память, как и константная, кэшируется. Причем, размер кэша зависит от производительности. До производительности 6.0 размер кэша изменяется от 12 до 48 КБ, для 6.x – от 24 до 48 КБ, для 7.0 – от 32 до 128 КБ.

Примеры использования одномерной текстуры

Приведем пример программы для работы с одномерной текстурой, привязанной к CUDA массиву.

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>

// объявляем одномерный текстурный объект
texture<float, 1, cudaReadModeElementType> texRef;

__global__ void readTexels(int n, float *d_out)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n)
    {
        float x = tex1D(texRef, float(idx));
        d_out[idx] = x;
    }
}

#define NUM_THREADS 256

int main()
{
    int N = 10;
    int nBlocks = N / NUM_THREADS + ((N % NUM_THREADS) ?
    1 : 0);
    float *d_out;

    // выделяем память на GPU
    cudaMalloc((void**)&d_out, sizeof(float) * N);

    // выделяем память на CPU
    float *h_out = (float*)malloc(sizeof(float)*N);

    // заполняем массив data
    float *data = (float*)malloc(N * sizeof(float));
```

```
    for (int i = 0; i < N; i++) data[i] = float(2 * i);

    // создаем CUDA массив на device
    cudaArray* cuArray;
    cudaMallocArray(&cuArray, &texRef.channelDesc, N, 1);
    cudaMemcpyToArray(cuArray, 0, 0, data,
    sizeof(float)*N, cudaMemcpyHostToDevice);

    // привязываем текстуру к CUDA массиву
    cudaBindTextureToArray(texRef, cuArray);

    // устанавливаем атрибуты текстуры
    texRef.normalized = false;
    texRef.filterMode = cudaFilterModePoint;

    readTexels << <nBlocks, NUM_THREADS >> >(N, d_out);

    // копируем с девайса на хост
    cudaMemcpy(h_out, d_out, sizeof(float)*N,
    cudaMemcpyDeviceToHost);

    for (int i = 0; i < N; i++)
    printf("%f\n", h_out[i]);


    free(h_out);
    cudaFree(d_out);
    cudaFreeArray(cuArray);
    cudaUnbindTexture(texRef);
    getchar();
}
```

Программа выделяет память на GPU, привязывает к ней текстуру. Затем копирует из текстуры в массив. Ответ должен быть следующим:

```
0.000000
2.000000
```

```
    4.000000
    6.000000
    8.000000
    10.000000
    12.000000
    14.000000
    16.000000
    18.000000
```

Следующий пример кода связывает текстуру с линейной памятью. Эта программа заполняет на девайсе обыкновенной массив из текстуры, после этого копирует его на хост и сравнивает с исходным массивом. Программа должна выдать сообщение «Массивы совпали».

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <assert.h>

void checkCUDAError(const char *msg)
{
    cudaError_t err = cudaGetLastError();
    if (cudaSuccess != err)
    {
        fprintf(stderr, "Cuda error: %s: %s.\n", msg,
cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }
}

texture<float, 1, cudaReadModeElementType> texRef;

__global__ void kernel(int n, float *d_out)
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    if (idx < n)
```

```
            d_out[idx] = tex1Dfetch(texRef, idx);
}

#define NUM_THREADS 256
int main()
{
    int N = 5000;
    int nBlocks = N / NUM_THREADS + ((N % NUM_THREADS) ?
    1 : 0);
    int memSize = N * sizeof(float);

    // data fill array with increasing values
    float *data;
    data = (float*)malloc(memSize);
    for (int i = 0; i < N; i++) data[i] = float(i);

    float *d_a;
    cudaMalloc((void **)&d_a, memSize);
    cudaMemcpy(d_a, data, memSize,
    cudaMemcpyHostToDevice);

    cudaBindTexture(0, texRef, d_a, memSize);
    checkCUDAError("bind");

    kernel << <nBlocks, NUM_THREADS >> >(N, d_a);

    float *h_out = (float*)malloc(memSize);
    cudaMemcpy(h_out, d_a, memSize,
    cudaMemcpyDeviceToHost);
    checkCUDAError("cudaMemcpy");

    for (int i = 0; i << N; i++)
        assert(data[i] == h_out[i]);

    printf("Массивы совпали \n");

    cudaUnbindTexture(texRef);
    checkCUDAError("cudaUnbindTexture");
```

```
    free(h_out);
    free(data);
    getchar();
}
```

Вопросы к разделу

1. Где выделяется константная и текстурная память?
2. Где расположен кэш для константной и текстурной памяти? Какие размеры кэша?

Лабораторная работа

1. Напишите программу для скалярного умножения двух векторов, используя константную память. Замерьте время работы ядра программы.
2. Напишите программу вычисления определенного интеграла функции одной переменной по квадратурной формуле центральных прямоугольников. Используйте тектстурную память, привязанную к линейной памяти.

Домашняя работа

1. Напишите программу для скалярного умножения двух векторов, расположенных в текстурной памяти. Испольуйте три алгоритма: 1. В обоих случаях привязка к CUDA массиву; 2. В одном – к CUDA массиву, в другом – к линейной памяти; 3. В обоих случаях привязаны к линейной памяти. Сравните время работы для различных размеров массивов.

Глава 9. Пакеты для работы с векторами и матрицами

В этой главе рассмотрим два пакета: cuBLAS и cuBLAS-Xt.

Пакет cuBLAS

cuBLAS – реализация интерфейса программирования приложений для создания библиотек, выполняющих основные операции линейной алгебры BLAS (Basic Linear Algebra Subprograms) для CUDA. Этот пакет позволяет получить доступ к вычислительным ресурсам графических процессоров NVIDIA. Перед тем, как начать изучение непосредственно библиотеки cuBLAS, стоит ознакомится с интерфейсом BLAS, который реализует данная библиотека.

BLAS *(Basic Linear Algebra Subprograms*) – стандарт интерфейса программирования приложений (API) для создания библиотек, выполняющих основные операции линейной алгебры, такие как умножение векторов и матриц. Впервые опубликован в 1979 году.

Функциональность BLAS делится на три уровня.

Уровень 1:

Первый уровень содержит векторные операции вида:

$$y \leftarrow \alpha x + y,$$

операции скалярного произведения, взятия нормы вектора и другие операции.

Уровень 2:

Этот уровень содержит операции матрица-вектор вида:

$$y \leftarrow \alpha Ax + \beta y,$$

решение $Tx = y$ для x с треугольной матрицей T и другие операции.

Уровень 3:

Содержит операции матрица-матрица вида:

$$C \leftarrow \alpha AB + \beta C,$$

решение $B \leftarrow \alpha T^{-1} B$ для треугольной матрицы T и другие операции.

Введение в cuBLAS

CuBLAS является самодостаточной библиотекой на уровне API, то есть, прямого взаимодействия с драйвером CUDA не происходит. cuBLAS прикрепляется к одному GPU и автоматически не распараллеливается между несколькими GPU.

Каждая реализуемая функция представлена в нескольких вариантах: для данных одинарной и двойной точности, вещественных и комплексных.

Поскольку пакет BLAS изначально был написан на языке Fortran, NVIDIA приложила все усилия к тому, чтобы обеспечить максимальную совместимость. Соответственно, массивы в cuBLAS хранятся по столбцам, а не по строкам, как принято в C и C++. При этом сохраняется индексирование с 1. Чтобы упростить написание программ в рамках тех соглашений, которые приняты в языках C и C++, достаточно ввести макрос для вычисления элемента двумерного массива, хранимого в одномерном виде и в FORTRAN-овской нотации

```
#define IDX2F(i,j,ld) ((((j)-1)*(ld))+((i)-1))
```

Здесь ld – это размерности матрицы, и в случае хранения в столбцах, является количеством строк. Для кода изначально написанного на C и C++, можно было бы использовать индексирование с 0, в этом случае макрос выглядит так:

```
#define IDX2C(i,j,ld) (((j)*(ld))+(i))
```

В старых версиях библиотеки основные функции cuBLAS (в отличие от вспомогательных функций) не возвращают статус ошибки напрямую (из соображений совместимости с существующими библиотеками BLAS). В этом случае, чтобы помочь в отладке, cuBLAS предоставляет отдельную функцию, которая возвращает последнюю записанную ошибку. Мы будем рассматривать

только вторую версию библиотеки. Об особенностях старой (legacy) версии можно прочитать в документации (*раздел 1.2. New and Legacy cuBLAS API*)

Использование cuBLAS API

Перед использованием функций библиотеки cuBLAS, нужно инициализировать контекст (т.е. содержимое самой библиотеки) и передавать его при каждом вызове функций библиотеки. Контекст хранится в переменной типа cublasHandle_t. Инициализация контекста производиться вызовом функции cublasCreate(), который возвращает значение структуры типа cublasHandle_t. После завершения работы с библиотекой нужно вызвать функцию cublasDestory(), передав переменную, которую вернула функция cublasCreate(), чтобы освободить ресурсы, связанные с контекстом библиотеки cuBLAS.

Такой подход к использованию контекстов позволяет явно контролировать настройки при использовании нескольких нитей в хосте и нескольких графических ускорителей. При этом предполагается, что контекст не будет меняться между вызовами cublasCreate() и cublasDestory(). Для того, чтобы cuBLAS мог использовать разные GPU в одной и той же нити хоста, нужно инициализировать другой контекст, предварительно установив соответствующий девайс вызовом cudaSetDevice().

В дальнейшем, при приведении примеров без полного листинга, будем иметь ввиду, что все необходимые библиотеки подключены, макросы для доступа к элементам матрицы определены и приводимый код находится между вызовами функций создания и уничтожения контекста. Сам контекст хранится в переменной handle. Так же будем считать, что используемые переменные были определены и инициализированы случайными значениями.

Все функции библиотеки cuBLAS возвращают статус ошибки cublasStatus_t. При удачном завершении операции функция возвращает значение CUBLAS_STATUS_SUCCESS. Информацию о других статусах можно получить в официальной документации cuBLAS.

Вспомогательные функции cuBLAS

Эти функции не входят в интерфейс BLAS, они реализованы для взаимодействия с графическим ускорителем. Мы здесь приведем лишь краткое описание некоторых функций и их входных параметров. Описание возвращаемых статусов, причины тех или иных ошибок и способы их исправления смотрите в официальной документации.

```
cublasCreate() и cublasDestroy();

cublasStatus_t cublasCreate(cublasHandle_t *handle);

cublasStatus_t cublasDestroy(cublasHandle_t handle);
```

Как было описано ранее, эти функции отвечают за инициализацию контекста. Перед тем, как начать работать с функциями cuBLAS, нужно вызвать cublasCreate(). Так как cublasCreate() выделяет некоторые внутренние ресурсы, нужно в конце работы их освободить при помощи функции cublasDestroy().

Рекомендуется минимизировать использование данных функций в программах (желательно вызывать только один раз для каждого устройства).

```
cublasSetVector();

cublasStatus_t cublasSetVector(int n, int elemSize,
const void *x, int incx, void *y, int incy);
```

Функция копирует n элементов вектора x из памяти хоста в память девайса GPU. Предполагается, что элементы обоих векторов имеют размер elemSize байтов. Интервал между элементами у исходного вектора задается параметром incx, а у вектора получателя – incy.

В общем случае, переменная y должна указывать на объект или часть объекта, который был выделен на девайсе до этого.

```
cublasGetVector();

cublasStatus_t cublasGetVector(int n, int elemSize,
const void *x, int incx, void *y, int incy);
```

Функция копирует n элементов вектора x из памяти девайса в память хоста. Предполагается, что элементы обоих векторов имеют размер elemSize байтов. Интервал между элементами у исходного вектора задается параметром `incx`, а у вектора получателя – `incy`.

```
cublasSetMatrix(); и cublasGetMatrix();

cublasStatus_t cublasSetMatrix(int rows, int cols,
int elemSize, const void *A, int lda, void *B, int
ldb);

cublasStatus_t cublasGetMatrix(int rows, int cols,
int elemSize, const void *A, int lda, void *B, int
ldb);
```

Аналогично функциям cublasSetVector() и cublasGetVector(), cublasSetMatrix() и cublasGetMatrix() производят копирование матриц из хоста на девайс и обратно.

Функции BLAS

Для того, чтобы понять особенности использования библиотеки cuBLAS, опишем некоторые функции базовой подпрограммы линейной алгебры.

Все функции для математических операций имеют следующий вид:

```
cublasStatus_t cublas<T><operation>(cublasHandle_t
handle, ...);
```

Буква T обозначает тип данных

- S = float;
- D = double;
- C = complex float;
- Z = complex double.

и operation – принятое в стандарте BLAS название операции.

Например, функция вычисления нормы вектора с элементами типа float имеет вид:

```
cublasStatus_t cublasSnrm2(cublasHandle_t handle, int
n, const float *x, int incx, float *result)
```

Реализация той же операции для чисел с двойной точностью будет следующей:

```
cublasStatus_t cublasDnrm2(cublasHandle_t handle, int
n, const double *x, int incx, double *result)
```

Описание параметров для функций `cublas<t>nrm2()`.

Параметр	**Размещение в памяти**	**In/out**	**Описание**
handle		input	Контекст библиотеки cuBLAS
n		input	Количество элементов вектора x
x	device	input	Вектор типа <type> с n элементами.

Параметр	Размещение в памяти	In/out	Описание
incx		input	Расстояние между элементами вектора x.
result	host или device	output	Результат вычисления нормы, который равен 0.0 если n,incx<=0.

Также при вычислениях с матрицами, часто к матрицам добавляются некоторые преобразования. То есть вычисления имеют, например, вид $y = \alpha op(A)x + \beta y$, где β и α – скаляры, y и x – векторы, A – матрица, $op()$ - некоторая операция, применяемая к матрице (без преобразования, транспонирование или эрмитово сопряжение). Вид этой операции задается как параметр функции типа cublasOperation_t и может принимать следующие значения:

- CUBLAS_OP_N – матрица не преобразовывается;
- CUBLAS_OP_T – к матрице применяется транспонирование;
- CUBLAS_OP_C – к матрице применяется эрмитово сопряжение.

Также, некоторые функции выполняют лево- или правостороннее умножение матриц, в зависимости от входных параметров. Параметр типа cublasSideMode_t определяет тип умножения и принимает значения:

- CUBLAS_SIDE_LEFT – матрица с левой стороны;
- CUBLAS_SIDE_RIGHT – матрица, соответственно, с правой стороны.

cuBLAS функции первого уровня

cublasI<t>amax() функция поиска индекса (наименьшего) максимального элемента;

cublasI<t>amin() функция поиска индекса (наименьшего) минимального элемента;

cublas<t>asum() функция находит сумму абсолютных значений элементов вектора;

cublas<t>axpy() функция умножает вектор *x* на скаляр *alpha* и прибавляет результат к вектору *y*. Значение вектора *y* презаписывается;

cublas<t>copy() функция поэлементно копирует вектор *x* в вектор *y;*

cublas<t>dot() функция производит скалярное умножение векторов *x* и *y;*

cublas<t>nrm2() функция вычисляет Евклидову норму вектора;

cublas<t>rot() функция применяет поворот Гивенса (поворот вектора на некоторый заданный угол) к векторам *x* и *y;*

cublas<t>rotg() функция применяет поворот Гивенса к векторам *x* и *y* (с обнулением второго вхождения);

cublas<t>rotm() функция применяет модифицированную трансформацию Гивенса к векторам *x* и *y;*

cublas<t>rotmg() функция применяет модифицированную трансформацию Гивенса к векторам *x* и *y* (с обнулением второго вхождения);

cublas<t>scal() функция умножает вектор на скаляр и перезаписывает вектор новыми значениями;

cublas<t>swap() функция меняет вектора местами.

Пример 1. Реализовать следующее вычисление: $x = \alpha xx$, где x – вектор, α – скаляр. Вычисления проводятся для чисел с двойной точностью.

```
// векторы x и y определены на CPU и скопированы
// на GPU (dev_x, dev_y)
cublasDcopy(handle, n, dev_x, 1, dev_y, 1);
cublasDscal(handle, n, &alpha, dev_y, 1);
cublasDdot(handle, n, dev_y, 1, dev_x, 1, &result);
```

cuBLAS функции второго уровня

cublas<t>gbmv() функция применяет окаймленное умножение матрицы на вектор вида

$y = \alpha\, op(A)x + \beta y,$

где A – ленточная матрица;

cublas<t>gemv() функция применяет окаймленное умножение матрицы на вектор вида

$y = \alpha\, op(A)x + \beta y;$

cublas<t>ger() функция выполняет операцию вида

$A = \alpha x y^T + A,$

где A – матрица размерности n на $m;$

cublas<t>sbmv() функция применяет умножение матрицы на вектор вида

$y = \alpha Ax + \beta y,$

где A – симметричная ленточная матрица;

cublas<t>spmv() функция применяет умножение матрицы на вектор вида

$y = \alpha Ax + \beta y,$

где A – симметричная матрица, хранимая в пакетном формате;

cublas<t>spr() функция выполняет операцию вида

$A = \alpha x x^T + A;$

cublas<t>spr2() функция выполняет операцию вида

$A = \alpha(xy^T + yx^T) + A,$

где A – симметричная матрица, хранимая в пакетном формате;

cublas<t>symv() функция применяет умножение матрицы на вектор вида

$y = \alpha Ax + \beta y,$

где A – симметричная матрица;

cublas<t>syr() функция выполняет операцию вида

$A = \alpha x x^T + A,$

где A – симметричная матрица;

cublas<t>syr2() функция выполняет операцию вида

$A = \alpha(xy^T + yx^T) + A,$

где A – симметричная матрица;

cublas<t>tbmv() функция выполняет операцию вида

$x = op(A)x,$

где A – треугольная матрица;

cublas<t>tbsv() функция решает уравнение вида

$op(A)x = b,$

где A – треугольная матрица;

cublas<t>tpmv() функция выполняет операцию вида

$x = op(A)x,$

где A – треугольная матрица, хранимая в пакетном формате;

cublas<t>tpsv() функция решает уравнение вида

$op(A)x = b,$

где A – треугольная матрица, хранимая в пакетном формате;

cublas<t>trmv() функция выполняет окаймленное умножение матрицы на вектор вида

$x = op(A)x\ ,$

где A – треугольная матрица;

cublas<t>trsv() функция решает уравнение вида

$op(A)x = b,$

где A – треугольная матрица;

cublas<t>hemv() функция применяет умножение матрицы на вектор вида

$y = \alpha Ax + \beta y,$

где A – эрмитова матрица;

cublas<t>hbmv() функция применяет умножение матрицы на вектор вида

$y = \alpha Ax + \beta y,$

где A – эрмитова ленточная матрица;

cublas<t>hpmv() функция применяет умножение матрицы на вектор вида

$y = \alpha Ax + \beta y,$

где A – эрмитова матрица, хранимая в пакетном формате;

cublas<t>her() функция выполняет операцию вида

$A = \alpha x y^H + A,$

где A – эрмитова матрица;

cublas<t>her2() функция выполняет операцию вида

$A = \alpha(xy^T + yx^T) + A,$

где A – эрмитова матрица;

cublas<t>hpr() функция выполняет операцию вида

$A = \alpha xy^H + A,$

где A – эрмитова матрица, хранимая в пакетном формате;

cublas<t>hpr2() функция выполняет операцию вида

$A = \alpha(xy^T + yx^T) + A,$

где A – эрмитова матрица, хранимая в пакетном формате.

Пример 2. Вычислить следующее выражение: $y = \alpha Ax$.

```
// вектор x и матрица A определены на CPU и скопированы
// на GPU (dev_x, dev_A)
// вектор y инициализирован 0-ми и так же скопирован на
// GPU (dev_y)

cublasDgemv()(handle, n, m, &alpha, dev_A, m, dev_x, 1,
&beta, dev_y, 1);
```

cuBLAS функции третьего уровня

cublas<t>gemm() функция умножения матриц вида

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

где A – m на k, B – k на n, C – m на n;

cublas<t>gemm3m() функция умножения матриц вида

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

где A – m на k, B – k на n, C – m на n. При этом применяется метод Гаусса для ускорения операции;

cublas<t>gemmBatched() функция умножения матриц вида

$C[i] = \alpha\, op(A[i])op(B[i]) + \beta\, op(C[i])$,

для массива матриц;

cublas<t>symm() функция умножения симметричных матриц вида

$C = \alpha AB + \beta C$ или $C = \alpha BA + \beta C$;

cublas<t>syrk() функция выполняет для симметричных матриц операцию вида

$C = \alpha\, op(A)op(A)^T + \beta C$;

cublas<t>syr2k() функция выполняет для симметричных матриц операцию вида

$C = \alpha\, (op(A)op(B)^T + op(B)op(A)^T) + \beta C$;

cublas<t>syrkx() функция выполняет для симметричных матриц операцию вида

$C = \alpha\, op(A)op(B)^T + \beta C$;

cublas<t>trmm() функция умножения треугольных матриц вида

$C = \alpha op(A)B$ или $C = \alpha Bop(A)$;

cublas<t>trsm() функция решает для треугольных матриц уравнение вида

$op(A)X = \alpha B$ или $X\, op(A) = \alpha B$;

cublas<t>trsmBatched() функция решает для массива треугольных матриц уравнение вида

$op(A[i])X[i] = \alpha B[i]$ или $X[i]\, op(A[i]) = \alpha B[i]$;

cublas<t>hemm() функция умножения для эрмитовых матриц вида

$C = \alpha AB + \beta C$ или $C = \alpha BA + \beta C;$

cublas<t>herk() функция реализует для эрмитовых матриц операцию вида

$C = \alpha\ op(A)op(A)^H + \beta C;$

cublas<t>her2k() функция реализует для эрмитовых матриц операцию вида

$C = \alpha\ op(A)op(B)^H + \alpha\ op(B)op(A)^H + \beta C;$

cublas<t>herkx() функция выполняет для эрмитовых матриц операцию вида

$C = \alpha\ op(A)op(B)^H + \beta C.$

Пример 3. Умножить две матрицы. Результат записать в третью матрицу.

```
// матрицы A и B определены на CPU и скопированы на GPU
// (dev_A, dev_B)
// матрица C инициализирован 0-ми и скопирован на GPU
// (dev_C)
// alpha и beta равны 1
cublasDdot(handle, CUBLAS_OP_N, CUBLAS_OP_N, n, n, n,
&alpha, dev_A, n, dev_B, n, &beta, dev_A, n);
```

BLAS-подобное расширение

cublas<t>geam() функция выполняет над матрицами операцию вида

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

где A, B, C – m на $n;$

cublas<t>dgmm() функция выполняет операцию вида

$C = A\ diag(X)$ или $C = diag(X)\ A$,

где A – матрица m на n, X – вектор из n элементов;

cublas<t>getrfBatched() функция выполняет LU-разложение для массива матриц:

$P\ A[i] = LU;$

cublas<t>getrsBatched() функция решает систему линейных уравнений для массива LU-разложенных матриц;

cublas<t>getriBatched() функция вычисляет обратные матрицы для массива матриц;

cublas<t>matinvBatched() функция вычисляет обратные матрицы для массива матриц;

cublas<t>geqrfBatched() функция выполняет для массива матриц QR-разложение;

cublas<t>gelsBatched() функция решает для массива матриц следующую задачу:

$min||Carr[i] - Aarr[i]*Xarr[i]\ ||;$

cublas<t>tpttr() функция выполняет преобразование треугольной матрицы из пакетного формата в «обычный»;

cublas<t>trttp() функция выполняет преобразование треугольной матрицы из «обычного» формата в пакетный;

cublas<t>gemmEx() функция умножения матриц вида

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

где A – m на k, B – k на n, C – m на n. Входные данные могут иметь более низкую точность, чем выходные;

cublasCsyrkEx() функция выполняет для симметричных матриц операцию вида

$$C = \alpha\, op(A) op(A)^T + \beta C,$$

Причем, входные данные могут иметь более низкую точность, но выходные будут типа cuComplex.

cublasCsyrk3mEx() функция выполняет модификацию предыдущей операции;

cublasCherkEx() функция выполняет для симметричных матриц операцию вида

$$C = \alpha\, op(A) op(A)^H + \beta C,$$

входные данные могут иметь более низкую точность , но выходные будут типа cuComplex.

cublasCsyrk3mEx() функция выполняет модификацию предыдущей операции;

cublasCherkEx() функция выполняет ту же операцию, что и cublasCherk. Входные данные могут иметь более низкую точность, но выходные будут типа cuComplex.

cublasCherk3mEx() функция выполняет модификацию предыдущей операции;

cublasNrm2Ex() функция вычисляет Евклидову норму вектора. Входные данные могут иметь более низкую точность, чем выходные (тип входных и выходных параметров передаются дополнительно).

cublasAxpyEx() функция является обобщением cublas<t>axpy, где тип входных и выходных параметров передаются дополнительно.

cublasDotEx() функция является обобщением cublas<t>dot, где тип входных и выходных параметров передаются дополнительно.

cublasScalEx() функция является обобщением cublas<t>scal, где тип входных и выходных параметров передаются дополнительно.

Пример 4. Выполнить скалярное произведение двух векторов одинарной точности. Результат должен быть типа double.

```
cublasDotEx(handle, n, dev_x, CUDA_R_32F, 1, dev_y,
CUDA_R_32F, 1, &result, CUDA_R_64F, CUDA_R_64F);
```

Приведем пример решения системы линейных уравнений с нижней треугольной матрицей.

```
#include <cstdio>
#include <cstdlib>
#include <ctime>
#include <cuda_runtime.h>

// подключение библиотеки cuBLAS
#include <cublas_v2.h>
// макрос для работы с индексами в стиле FORTRAN
```

```
#define IDX2C(i,j,ld) (((j)*(ld))+(i))
int main()
{
    const int N = 6;
    cublasHandle_t handle;
    float *dev_A, *dev_b;
    float *x, *A, *b;
    x = (float *)malloc(N * sizeof(*x));
    b = (float *)malloc(N * sizeof(*b));
    A = (float *)malloc(N * N * sizeof(*A));
    // инициализация матрицы и вектора правой части
    int ind = 11;
    for (int j = 0; j < N; j++)
    {
        for (int i = 0; i < N; i++)
            if (i >= j)
                A[IDX2C(i, j, N)] = (float)ind++;
            else A[IDX2C(i, j, N)] = 0.0f;
        b[j] = 1.0f;
    }
    // выделяем память на GPU соответствующего размера
    // для каждой переменной
    cudaMalloc((void**)&dev_b, N * sizeof(*x));
    cudaMalloc((void**)&dev_A, N * N * sizeof(*A));
    // инициализируем контекст cuBLAS
    cublasCreate(&handle);
    // копируем вектор и матрицу из CPU в GPU
    cublasSetVector(N, sizeof(*b), b, 1, dev_b, 1);
    cublasSetMatrix(N, N, sizeof(*A), A, N, dev_A, N);
    // решаем нижнюю треугольню матрицу
```

```
    cublasStrsv(handle, CUBLAS_FILL_MODE_LOWER,
    CUBLAS_OP_N, CUBLAS_DIAG_NON_UNIT, N, dev_A, N,
    dev_b, 1);
    // копируем результат из GPU в CPU
    cublasGetVector(N, sizeof(*x), dev_b, 1, x, 1);
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
            printf("%3.0f ", A[IDX2C(i, j, N)]);
        printf(" = %f %4.6f\n", b[i], x[i]);
    }
    // освобождаем память в GPU
    cudaFree(dev_b);
    cudaFree(dev_A);
    // уничтожаем контекс cuBLAS
    cublasDestroy(handle);
    // освобождаем память в CPU
    free(x);
    free(b);
    free(A);
    getchar();
    return EXIT_SUCCESS;
}
```

При компиляции могут возникать следующие ошибки:

LNK2019 unresolved external symbol cublasCreate_v2 referenced in function main

Для их устранения необходимо подключить библиотеку cublas.lib, добавив ее в *Project→Properties→Linker→Input→Additional Dependencies*.

После этого возможна следующая ошибка для 64-битных систем: *LNK1104 cannot open file 'cublas.lib'*. Эта проблема устраняется изменением значения *Build→Configuration Manager…→Platform* на *x64*.

Пакет cuBLAS-Xt

cuBLAS-Xt – это набор рутинных подпрограмм, которые ускоряют вызовы в Level 3 BLAS (базовые подпрограммы линейной алгебры) за счет распределения нагрузки на нескольких GPU. Благодаря потоковой структуре, cuBLAS-Xt эффективно автоматически управляет пересылками на шине PCI-Express, что позволяет хранить входные и выходные данные в системной памяти. Это приводит к тому, что операции выполняются с использованием внешней памяти, размер операндов памяти ограничен только размером системной памяти, а не памятью GPU. cuBLAS-XT поддерживается на всех 64-битных платформах. cuBLASXt API заботится о распределении памяти между назначенными графическими процессорами и распределяет нагрузку между ними и, наконец, возвращает результаты обратно на хост. cuBLAS-Xt API поддерживает только вычисление операций вида BLAS3 (матрица-матрица). Начиная с версии CUDA 6.0, бесплатная версия cuBLAS-Xt входит в состав CUDA Toolkit как часть библиотеки cuBLAS. Cublas-Xt API имеет файл заголовка cuBLAS-Xt.h. Примеры cuBLAS-Xt находятся в каталоге: C:\ProgramData\NVIDIA Corporation\CUDA Samples\v8.0\7_CUDALibraries\simpleCUBLASXT

Инициализация контекста cuBLAS-XT API

Для возможности использования функций библиотеки cuBLAS-Xt требуется проинициализировать контекст. Тип cublasXtHandle_t является типом указателя на cuBLAS-Xt API контекст. cublasXt API контекст инициализируется с помощью cublasXtCreate(), и возвращенный дескриптор должен быть передан на все последующие функции cuBLAS-Xt API. Контекст должен быть уничтожен в конце, используя cublasXtDestroy().

```
cublasStatus_t status;
cublasXtHandle_t handle;
status = cublasXtCreate(&handle);
```

```
-----
status = cublasXtDestroy(handle);
```

После инициализации контекста выбираются графические устройства, между которыми будут распределяться вычисления.

```
int devices[1] = { 0 };
cublasXtDeviceSelect(handle, 1, devices);
```

Функция cublasXtDeviceSelect позволяет пользователю указать количество устройств GPU и соответствующие им идентификаторы, которые будут участвовать в последующих вызовах функций API cuBLAS-Xt Math.

```
cublasXtDeviceSelect(cublasXtHandle_t handle, int nbDevices, int DeviceId[])
```

Эта функция создает контекст cuBLAS для каждого GPU, представленного в этом списке. Конфигурация устройства не может быть изменена между вызовами функций Math. В этой связи, эта функция должна вызываться только один раз после функции cublasXtCreate. Для того, чтобы иметь возможность запускать несколько конфигураций, требуется несколько контекстов.

Распределение нагрузки

Для распределения нагрузки между несколькими GPU, API cuBLAS-Xt использует стратегию черепицы (“tiling strategy”) каждая матрица делится на квадратные плитки размерностью, выбранной пользователем. Функция cublasXtSetBlockDim позволяет пользователю установить размер блока, используемого для разбиения матриц для последующих вызовов функций Math.

```
cublasXtSetBlockDim(cublasXtHandle_t handle, int blockDim)
```

Матрицы разделяются в квадратные плитки blockDimxblockDim. Эта функция может быть вызвана в любое время и вступит в силу для следующих вызовов функций Math. Размер блока должен быть выбран таким образом, чтобы оптимизировать работу и убедиться в том, что переводы PCI хорошо покрываются вычислениями. Функция cublasXtGetBlockDim() позволяет пользователю запрашивать размер блока.

Функции cuBLAS-Xt API Math

Все функции для математических операций имеют вид:

cublasStatus_t cublasXt<T><operation>(cublasXtHandle_t handle, ...)

где буква T обозначает тип данных

<type>	<T>
float	'S'
double	'D'
cuComplex	'C'
cuDoubleComplex	'Z'

1. cublasXt<T>gemm

```
cublasStatus_t cublasXtSgemm(cublasXtHandle_t handle, cublasOperation_t transa, cublasOperation_t transb, size_t m, size_t n, size_t k, const float *alpha, const float *A, int lda, const float *B, int ldb, const float *beta, float *C, int ldc)
```

Эта функция выполняет перемножение матриц.

C=αop(A)op(B)+βC,

$$op(A) = \begin{cases} A, & \text{if transa} == \text{CUBLAS\_OP\_N}, \\ A^{T}, & \text{if transa} == \text{CUBLAS\_OP\_T}, \\ A^{H}, & \text{if transa} == \text{CUBLAS\_OP\_C}, \end{cases}$$

где α и β – скаляры, A, B и C являются матрицами с размерами m × k , k × n и m × n, соответственно, lda, ldb, ldc – первые размерности двумерных массивов, используются для хранения матриц.

2. cublasXt<t>hemm()

```
cublasStatus_t cublasXtChemm(cublasXtHandle_t handle, cublasSideMode_t side, cublasFillMode_t uplo, size_t m, size_t n, const cuComplex *alpha, const cuComplex *A, size_t lda, const cuComplex *B, size_t ldb, const cuComplex *beta, cuComplex *C, size_t ldc)
```

Эта функция выполняет умножение Эрмитовой матрицы на матрицу.

$$C = \begin{cases} \alpha AB + \beta C, & \text{if side} == \text{CUBLAS\_SIDE\_LEFT}, \\ \alpha BA + \beta C, & \text{if side} == \text{CUBLAS\_SIDE\_RIGHT}, \end{cases}$$

где α и β скаляры, A – Эрмитова матрица, m × n размер матриц B и C соответствующие матрице A, uplo указывает, с какой частью матрицы производить вычисления – нижней треугольной или верхней (т.к нижнюю треугольную матрицу эрмитовой матрицы можно получить из верхней и наоборот).

3. cublasXt<t>symm()

```
cublasStatus_t cublasXtSsymm(cublasXtHandle_t handle, cublasSideMode_t side, cublasFillMode_t uplo, size_t m, size_t n, const float *alpha, const float *A, size_t lda, const float *B, size_t ldb, const float *beta, float *C, size_t ldc)
```

Эта функция выполняет умножение симметричной матрицы на матрицу

$$C = \begin{cases} \alpha AB + \beta C, & \text{if side} == \text{CUBLAS\_SIDE\_LEFT}, \\ \alpha BA + \beta C, & \text{if side} == \text{CUBLAS\_SIDE\_RIGHT}, \end{cases}$$

где α и β – скаляры, A, B и C являются матрицами с размерами m × k , k × n и m × n, соответственно, lda, ldb, ldc – первые размерности двумерных массивов, используются для хранения матрицы, uplo указывает, с какой частью матрицы производить вычисления – нижней треугольной, либо верхней.

4. cublasXt<t>trsm()

```
cublasStatus_t cublasXtStrsm(cublasXtHandle_t handle, cublasSideMode_t side, cublasFillMode_t uplo, cublasOperation_t trans, cublasXtDiagType_t diag, size_t m, size_t n, const float *alpha, const float *A, size_t lda, float *B, size_t ldb)
```

Эта функция решает треугольную линейную систему с несколькими правыми сторонами.

$$\begin{cases} \mathrm{op(A)X} = \alpha \mathrm{B}, & \text{if side} == \text{CUBLAS\_SIDE\_LEFT}, \\ X\mathrm{op(A)} = \alpha \mathrm{B}, & \text{if side} == \text{CUBLAS\_SIDE\_RIGHT}, \end{cases}$$

где A – треугольная матрица с или без главной диагонали, uplo указывает, с какой частью матрицы производить вычисления – нижней треугольной или верхней. X и B – матрицы размера m×n, и α скаляр. Кроме того, для матрицы A:

$$\mathrm{op(A)} = \begin{cases} \mathrm{A}, & \text{if transa} == \text{CUBLAS\_OP\_N}, \\ \mathrm{A}^{\mathrm{T}}, & \text{if transa} == \text{CUBLAS\_OP\_T}, \\ \mathrm{A}^{\mathrm{H}}, & \text{if transa} == \text{CUBLAS\_OP\_C}. \end{cases}$$

Матрица решений X записывается поверх матрицы B на выходе.

5. cublasXt<t>trmm()

Функция выполняет умножение треугольной матрицы на матрицу.

$$\mathrm{C} = \begin{cases} \alpha\, \mathrm{op(A)B}, & \text{if side} == \text{CUBLAS\_SIDE\_LEFT}, \\ \alpha\, \mathrm{B\, op(A)}, & \text{if side} == \text{CUBLAS\_SIDE\_RIGHT}. \end{cases}$$

6. cublasXt<t>spmm()

Функция выполняет умножение симметрично упакованной матрицы на матрицу.

$$C = \begin{cases} \alpha AB + \beta C, & \text{if side} == \text{CUBLAS\_SIDE\_LEFT}, \\ \alpha BA + \beta C, & \text{if side} == \text{CUBLAS\_SIDE\_RIGHT}. \end{cases}$$

Гибридные вычисления CPU-GPU

В случае очень больших проблем API cuBLAS-Xt предлагает возможность переложить часть вычислений на CPU. Эта функция может быть настроена с процедуры cublasXtSetCpuRoutine() и cublasXtSetCpuRatio().

Эту функцию следует использовать с осторожностью, поскольку он может помешать потокам процессора, работающим с видеокартой. На данный момент только cublasXt<t>gemm() поддерживает эту функцию.

Документация библиотеки расположена на сайте

http://docs.nvidia.com/cuda/cublas/#using-the-cublasXt-api

Приведем листинг кода перемножения матриц:

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

/* Подключаем требуемые библиотеки cuda */
#include <cuda_runtime.h>
#include <cublasXt.h>
#define N (2)
int main(int argc, char **argv)
{
    // статус ошибки возвращается всеми cuBLAS функциями
    cublasStatus_t status;
```

```
    float *h_A;
    float *h_B;
    float *h_C;
    // Скаляры
    float alpha = 1.0f;
    float beta = 0.0f;
    // Размер матрицы
    int n2 = N * N;
    // Указатель на контекст CUBLAS-XT
    cublasXtHandle_t handle;
    int devices[1] = { 0 };
    // Инициализируем контекст CUBLAS-XT
    status = cublasXtCreate(&handle);
    if (status != CUBLAS_STATUS_SUCCESS)
    {
        fprintf(stderr, "!!!! CUBLASXT initialization
        error\n");
        return EXIT_FAILURE;
    }
    /* Выбираем устройства для запуска функций CUBLAS-XT
между которыми будет распределяться нагрузка */
    status = cublasXtDeviceSelect(handle, 1, devices);
    if (status != CUBLAS_STATUS_SUCCESS)
    {
        fprintf(stderr, "!!!! CUBLASXT device selection
        error\n");
        return EXIT_FAILURE;
    }
/* Устанавливаем размер блоков (blockDim x blockDim) на
которые будут разбиваться матрицы при распределение между
устройствами */
    status = cublasXtSetBlockDim(handle, 64);
    if (status != CUBLAS_STATUS_SUCCESS)
    {
        fprintf(stderr, "!!!! CUBLASXT set block
        dimension error\n");
        return EXIT_FAILURE;
    }
```

```
// Выделяем память для матриц в системной памяти
h_A = (float *)malloc(n2 * sizeof(h_A[0]));
h_B = (float *)malloc(n2 * sizeof(h_B[0]));
h_C = (float *)malloc(n2 * sizeof(h_C[0]));
// Заполняем матрицы тестовыми данными
srand((unsigned int)time(NULL));
for (int i = 0; i < n2; i++)
{
    h_A[i] = rand() / (float)RAND_MAX;
    h_B[i] = rand() / (float)RAND_MAX;
}
// Выполняем операцию перемножения матриц
status    =    cublasXtSgemm(handle,    CUBLAS_OP_N,
CUBLAS_OP_N, N, N, N, &alpha, h_A, N, h_B, N, &beta,
h_C, N);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf(stderr,    "!!!!    kernel    execution
    error.\n");
    return EXIT_FAILURE;
}
// Вывод результата перемножения матриц
for (int i = 0; i < N; i++)
{
    for (int j = 0; j < N; j++)
        printf("%4.4f ", h_B[i*N + j]);
    printf("  *  ");
    for (int j = 0; j < N; j++)
        printf("%4.4f ", h_A[i*N + j]);
    printf("  =  ");
    for (int j = 0; j < N; j++)
        printf("%4.4f ", h_C[i*N + j]);
    printf("\n");
}
// Очищаем память
free(h_A);
free(h_B);
free(h_C);
```

```
    getchar();
}
```

Контрольные вопросы

1. Как происходит инициализация контекста в библиотеке cuBLAS?
2. Как можно оптимизировать первый пример? Можно ли в этом примере обойтись одним вектором вместо двух?
3. Где должна размещаться (девайс или хост) переменная, в которую возвращается результат вычисления скалярного произведения (cublasDdot())?
4. Перечислите основные этапы работы с cuBLAS-Xt API.
5. Назначение типа cublasXtHandle_t. Возможно ли обойтись без него?
6. Какой вид имеют функции для математических операций?
7. В чем разница библиотек cublas и cuBLAS-Xt?
8. Назначение функции cublasXtSetBlockDim.

Лабораторная работа

1. Дополните пример обработкой ошибок. Учитывайте как ошибки при выделении памяти на CPU и GPU (функции CUDA), так и возвращаемый статус функций cuBLAS.
2. Вычислите матрицу C = op(A + B) * (A * B), где A и B квадратные матрицы, используя библиотеку cuBlasXT.

Домашнее задание

1. Расширьте пример для решения любых матриц. Для этого использовать LU-разложение из BLAS-like функций.
2. Найдите значения матрицы X. α (A + C) * C * X = B, где α – скаляр, матрицы A и C верхнетреугольные.

   A=((1,1), (0,1)), C=((1, -1), (0, 1)), B=((12, 10) (6, 8)).

3. Сравните время работы перемножения матриц с использованием библиотек cuBLAS, cuBLAS-Xt и без библиотек.

Литература

1. Боресков А.В., Харламов А.А. *Основы работы с технологией CUDA*. – М.: ДМК Пресс, 2010. – 232 с.
2. Сандерс Дж., Кэндрот Э. *Технология CUDA в примерах: Введение в программирование графических процессоров*. – М.: ДМК Пресс, 2013. – 232 с.
3. Cheng J., Grossman M., McKercher T. *Professional CUDA C Programming*. – Indianapolis: John Wiley & Sons, Inc., 2014. – 528 p.

Тумаков Дмитрий Николаевич

Чикрин Дмитрий Евгеньевич

Егорчев Антон Александрович

Голоусов Святослав Владимирович

ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ CUDA

Учебное пособие

Подписано в печать 14.11.2017

Форм. Бум. 60 X 84 1/16. Гарнитура “Таймс”. Печать цифровая.

Печ. л. 8,25. Т. 100. Заказ 15.

Издательство КФУ,

420008, Казань, ул. пр Нужина, 1/37,

+7 (843) 233-73-28, +7 (843) 233-73-59