Петрозаводский государственный университет
Математический факультет
Кафедра прикладной математики и кибернетики

Курсовая работа

Программирование на графических процессорах с использованием технологии CUDA

Выполнил:
студент 3-го курса группы 22303
М. М. Пробичев

_____
(подпись)

Научный руководитель:
к. ф.-м. н., ст. преподаватель А. М. Караваев

_____
(оценка руководителя)

_____
(подпись)

Представлена на кафедру:
« ____ » _____ 2014 г.

_____
(подпись принявшего работу)

Итоговая оценка _____

_____
(подпись)

Петрозаводск
2014

Содержание

Введение

На сегодняшний день, высокопроизводительные параллельные вычисления являются одной из актуальных и перспективных тем исследования. В нашем распряжении появилось много различных технологий для этого, например OpenMP, MPI, Nvidia CUDA. Эта тема, по большей части, актуальна тем, что с каждым годом объём обрабатываемых данных растёт, и для ускорения обработки требуется более мощные средства, а также перенос имеющихся эффективных алгоритмов на новые, многоядерные платформы.

Существует огромное многообразие задач, для исследования которых необходимы высокопроизводительные системы вычислений, например анализ и обработка графического или видео материала, комбинаторные задачи, моделирование физических процессов, финансовые расчёты и прогнозы. Часть может быть решена за счёт использования облачных вычислений, часть на суперкомпьютерах обладающих производительностью в сотни терафлопов. Но часть задач требует решения на домашних машинах. Раньше решение данных задач в домашних условиях было весьма проблематично, так как в нашем распоряжении был только центральный процессор, не способный на быстрые трудоёмкие вычисления.

Но современные многопроцессорные системы дают возможность справляться с этими задачами. Помимо многоядерных процессоров, современные компьютеры оснащаются видеокартами, которые позволяют решать трудоёмкие вычислительные задачи.

Графические процессоры устроены таким образом, чтобы эффективно обрабатывать огромные объёмы данных, благодаря большому количеству вычислительных ядер. Они имеют производительность в тысячи миллиардов операций над вещественными числами в секунду. Я считаю в этом и актуальность темы, использование графического процессора для высокопроизводительных вычислений, благодаря возможности быстро обрабатывать большие объёмы данных. Специалистов в

данной области не так много и нужно развивать данное направление.

Данная курсовая работа носит реферативный характер. Целью работы является изучение возможностей графических процессоров, и обучение программированию на видеокартах.

Также были поставлены следующие задачи:

1. Изучить парадигму программирования на видеокартах Nvidia.
2. Написать несколько типовых программ с использованием Nvidia CUDA и сравнить их с CPU реализациями.
3. Планирую написание учебного пособия для студентов.

Практическая значимость состоит в последующем написании учебного пособия для студентов.

1 Архитектура и парадигма программирования на GPU

В данной работе не будут описаны архитектурные различия центрального процессора и графического процессора, основные отличия, области применения графических карт, возможности, преимущества и ограничения Nvidia CUDA. А также модель программирования на GPU и несколько типовых задач. Также много полезной информации на данную тему есть в презентации на офф. сайте [2].

1.1 Сравнение CPU и GPU

Центральные процессоры являются универсальными устройствами, которые позволяют работать с задачами разного рода, в то время как графические процессоры рассчитаны на узкий спектр задач. Ядра CPU созданы для исполнения одного потока последовательных инструкций с наибольшей производительностью, а GPU для быстрого выполнения большого числа параллельно выполняемых потоков. Для сравнения: максимальное число потоков CPU составляет 20 на 10 ядер (Xeon E7) против 2048 потоков на каждый из 8 мультипроцессоров (5760 активных ядр CUDA) (Nvidia Titan Z). При этом для CPU переключение потоков стоит сотни тактов, то GPU переключает несколько потоков за такт. Однако ядро CPU превосходит по мощности ядро GPU в несколько раз.

Большая проблема вычислительных систем заключается в быстродействии памяти, когда память работает медленнее процессора. В CPU это решается использованием кэшей. В GPU они тоже есть, но менее мощные: кэшируется не все типы памяти, и кэши работают только на чтение. Однако, медленное обращение к памяти отлично скрывается параллельными вычислениями.

Отсюда получаем, что основное отличие универсальных CPU от GPU в том, что видеочипы предназначены для параллельных вычислений с большим количеством арифметических операций, и GPU куда лучше справляется с обработкой массивов данных, чем с управлением выполнения малого количества вычислительных потоков. Для эффективного использования GPU необходимо распараллелить алгоритмы на большое количество исполнительных блоков и максимально избегать сложных ветвлений и частого обращения к памяти. На следующем графике изображено отличие в производительности лучших моделей CPU и GPU

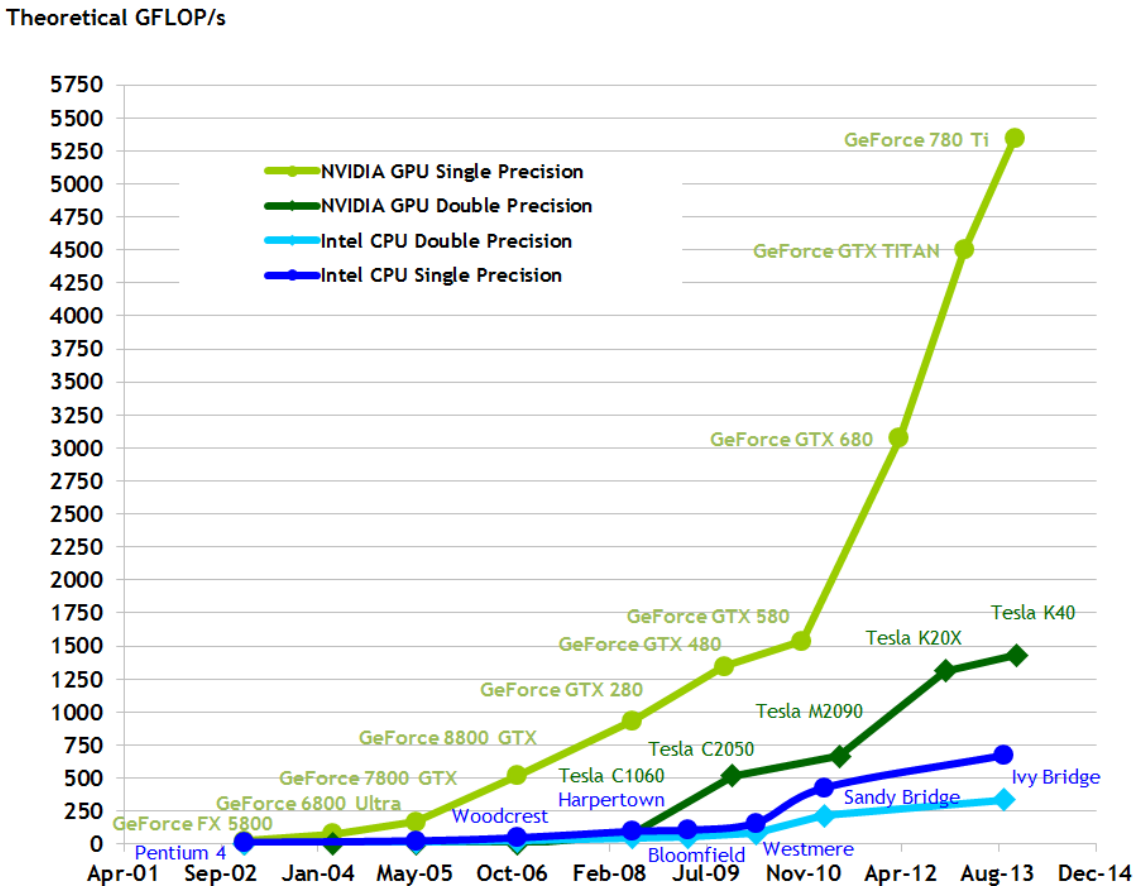

Рисунок 1 — График количества операций в секунду для CPU и GPU

Как видно по теоретическим оценкам, графические процессоры намного обходят универсальные центральные процессоры.

1.2 Потоковая модель GPU

Архитектура CUDA основана на концепции SIMD и понятии мультипроцессора. Данная концепция подразумевает,что одна инструкция позволяет одновременно обработать множество данных.

Мультипроцессор, в свою очередь, это многоядерный SIMD процессор, который позволяет в каждый момент времени выполнять на всех ядрах только одну инструкцию, каждое ядру мультипроцессора скалярно и не поддерживает векторные операции в чистом виде.

Устройство (device), в терминологии CUDA, это видеоадаптер поддерживающий драйвер CUDA. Хост(host), в данной терминологии, это программа в оперативной памяти компьютера, которая использует CPU, для выполнения управляющих функций по работе с устройством.

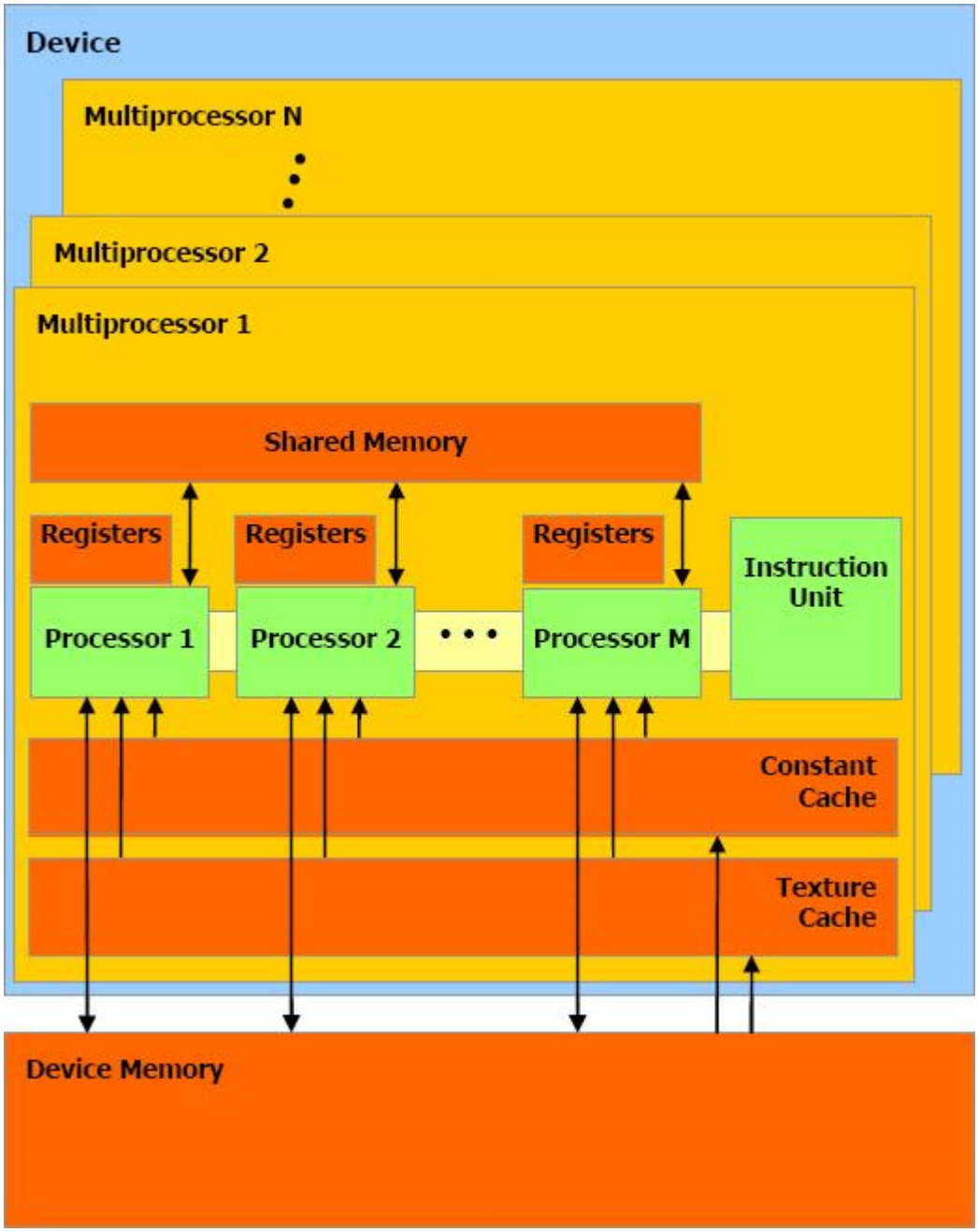

Рисунок 2 — Устройство графической карты

Главная особенность архитектуры CUDA, является блочно-сеточная организация, которая представлена на рис. 2. Все исполняемые на ядре потоки объединяются в блоки, а они объединяются в сетку.

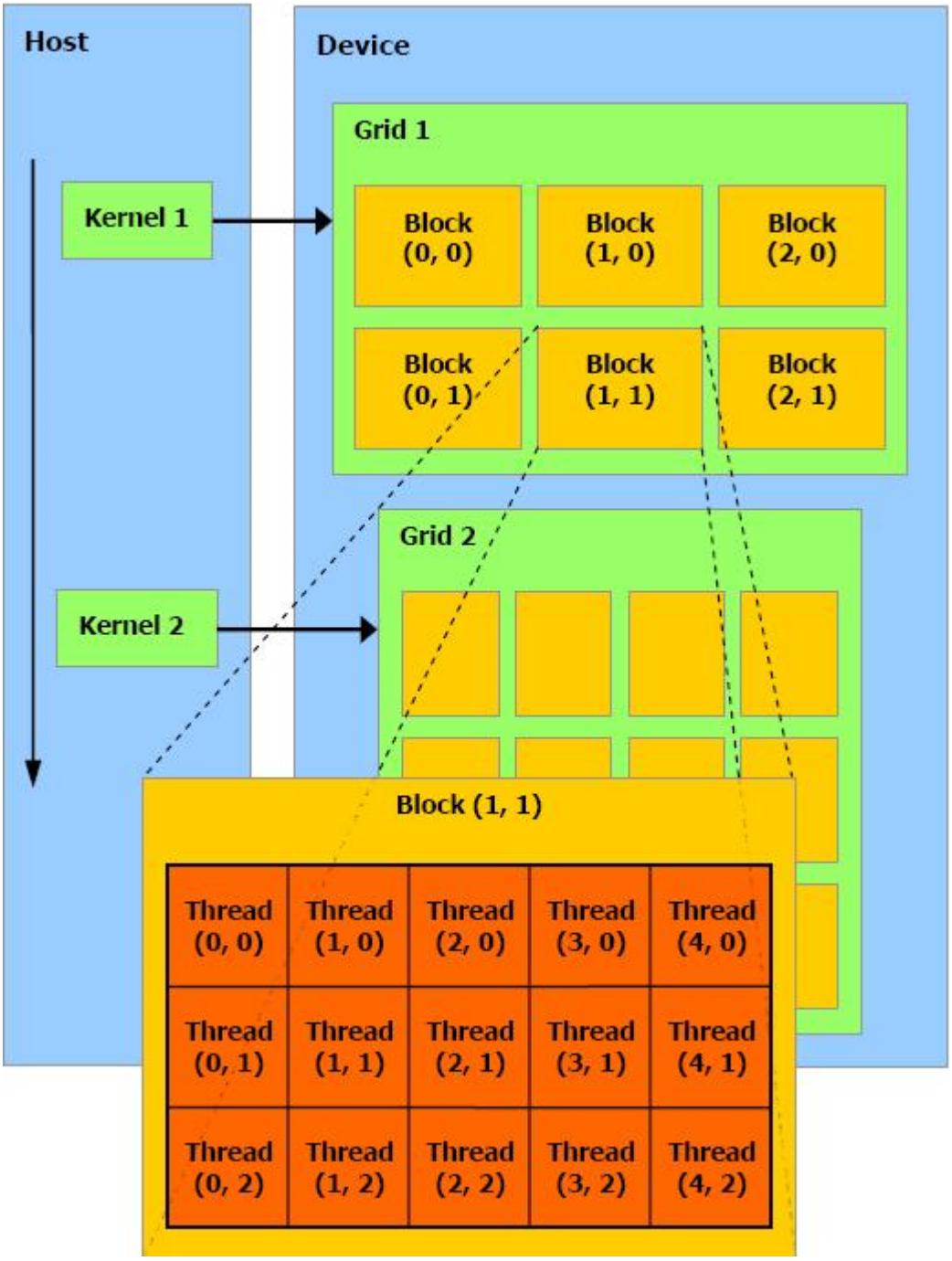

Рисунок 3 — Организация потоков GPU

Есть возможность работать как с двухмерными индексами потоков, так и с трёхмерными либо с простыми (одномерными). В общем случае, мы работаем с трёхмерными индексами. Для каждого потока мы можем узнать индекс потока внутри блока (threadIdx) и индекс блока внутри сетки (blockIdx), при помощи которых и сможем осуществлять управление.

Блок потоков выполняется на мультипроцессоре частями (warp), состоящии из 32 потоков. Во всех потоках внутри warp'а одновременно может выполняться только одна инструкция [3].

1.3 Память

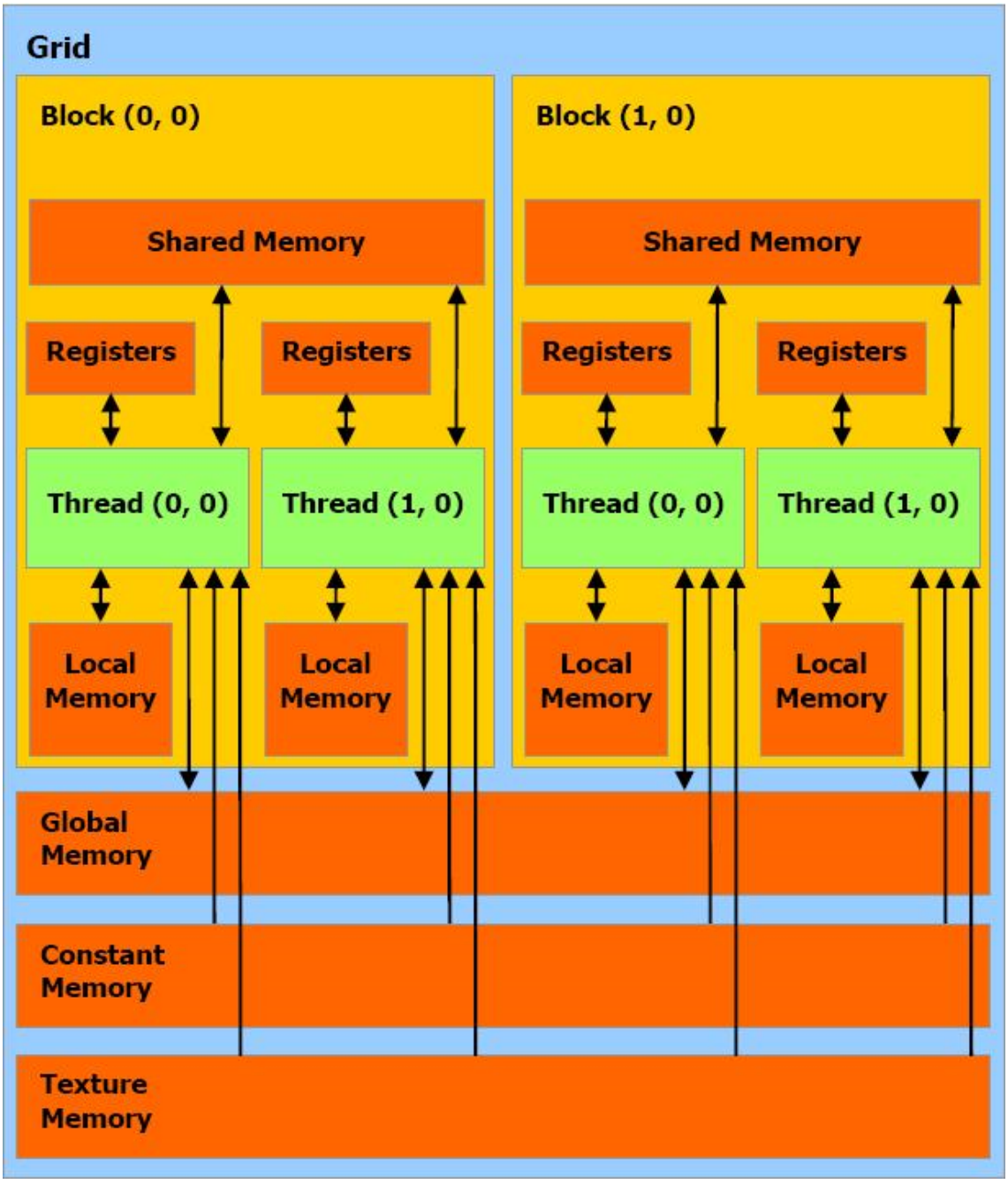

Рисунок 4 — Виды памяти GPU

В Nvidia CUDA выделяется 6 видов памяти, указанных на рис. 4. Доступ к переменным, размещённым в регистрах, осуществляется с максимальной скоростью, и компилятор по возможности старается размещать все локальные переменные в регистрах. На данный момент, для одного мультипроцессора доступно 65536 32-битных регистра (KEPLER GK110). Для одновременного выполнения 4 блоков на одном мультипроцессоре нужно стараться занимать не более 64 регистра на блок [3].

Если локальные данные занимают слишком много места, или компилятор не может вычислить для них некоторый постоянный шаг при обращении, он может поместить их в локальную память. В целях оптимизации, не рекомендуется использовать данный вид памяти [4].

Следующим видом памяти, является глобальная память. Обычно в GPU нет произвольной адресации глобальной памяти, но технология CUDA позволяет читать из любой ячейки памяти и писать в любую ячейку памяти. Однако это работает очень медленно и применяется для сохранения результата перед отправкой его на хост. Переменные, объявленные с квалификатором __global__, размещаются в глобальной памяти. Глобальную память также можно выделить динамически, вызвав функцию cudaMalloc(void* mem, int size) на хосте. Из устройства эту функцию вызывать нельзя, т.е. распределением памяти должна заниматься программа-хост, работающая на CPU. Данные с хоста можно отправлять в устройство вызовом функции cudaMemcpy.

Разделяемая память — это некэшируемая, но быстрая память. Ее рекомендуется использовать как управляемый кэш. На один мультипроцессор доступно 16-48KB разделяемой памяти. Разделив это число на количество потоков в блоке, получим максимальное количество разделяемой памяти, доступной на один поток (при независимом использовании). Данная память является одинаковой для всех задач внутри блока и при помощи её можно обмениваться данными между блоков.

Переменные, объявленные с квалификатором __shared__, размещаются в разделяемой памяти.

Рассмотрим константную память. Константная память кэшируется и кэш существует в единственном экземпляре для одного мультипроцессор. На хосте в константную память можно что-то записать, вызвав функцию cudaMemcpyToSymbol. Из устройства константная память доступна только для чтения. Константная память очень удобна в использовании. Можно размещать в ней данные любого типа и читать их при помощи простого присваивания.

Последний вид памяти, это текстурный. Текстурная память кэшируется. Для каждого мультипроцессора имеется один кэш. Текстурная память оптимизирована под выборку 2D данных и имеет следующие возможности: [4]

1. быстрая выборка значений фиксированного размера из одномерного или двухмерного массива;

2. нормализованная адресация числами типа float в интервале [0,1);

3. аппаратная линейная или билинейная (в случае 2D) интерполяция соседних значений в случае нормализованной адресации;

4. аппаратная обработка выхода за границу массива с использованием двух режимов: clamp и wrap.

Чтобы выделить текстурную память, необходимо выделить функцией cudaMalloc участок глобальной памяти, а затем указать с помощью функции cudaBindTexture, что данный участок теперь будет текстурной памятью.

Размер текстурной памяти ограничивается только максимальным размером памяти, выделяемым устройством .

1.4 Общая схема программироваия на CUDA

В каждом приложении CUDA необходимо аписать код исполняемый host'ом (CPU) и код непосредственно исполняемый device'ом (GPU). Опишем общие щаги CPU части приложения:

1. Подготовка памяти GPU. Необходимо выделить память на устройстве и перенести необходимые данные. Основные используемые функции cudaMalloc,cudaMemcpy и cudaFree (подробности смотрите в офф. документацию [3]).

2. Настройка блоков и сетки. Необходимо задать размеры блоков и сеток для них, соблюдая баланс между размером и количеством блоков. Рекомендуется использовать блоки по 128 или 256 потоков [3].

3. Запуск ядра. Вызываем ядро как обычную функцию языка C, и передаём в качестве параметров размерности блока и сетки.

4. Получение результатов и освобождение памяти. После завершения работы ядра, необходимо скопировать результаты в оперативную память (используя cudaMemcpy) и освободить все выделенные ресурсы.

Далее рассмотрим общие принципы GPU части приложения. CUDA, в плане написания кода, является расшрением языка C. При помощи данных расширений можно получить доступ к аппаратным возможностям GPU.

Часть программы, составляющая ядро, помещается в файл с расширением .cu, который компилируется при помощи NVCC. Устройства разделяются по типу следующими спецификаторами:

1. __host__ — выполняется на CPU. Вызывается с CPU.

2. __global__ — выполняется на GPU, вызывается с CPU.

3. __device__ — выполняется на GPU, вызывается с GPU.

Для переменных, описанных выше, также существуют спецификаторы:

1. __global__ — переменная находится в глобальной памяти устройства.

2. __constant__ — переменная в константной памяти.

3. __shared__ — переменная в разделяемой памяти блока.

Функция ядра может иметь произвольные параметры. Чаще всего, в качестве параметров передаются ссылки на области памяти с необходимой информацией, или размерность данных. Можно использовать переменные blockDim и blockIdx для обращения к конкретному блоку текущего экземпляра ядра.

2 Реализация типовых задач на Nvidia CUDA

В данной работе были рассмотрены такие типовые задачи как сложение двух многомерных векторов и перемножение двух матриц. Помимо программ, вычисляющих на GPU, были реализованы программы для суммирования векторов и перемножения матриц на CPU. Это было необходимо для сравнения правильности решений и для сравнения времени расчётов. Также для сравнения времени расчётов использовалось время вычислений программ для одного ядра CPU с сайта fastcomputing.org.

При сложении двух векторов на видеокарте, каждая пара координат вектора складывается на отдельном потоке и записывается в результирующий вектор.

Далее будет приведена более сложная задача — премножение двух матриц.

2.1 Перемножение матриц

Пусть у нас есть две квадратные матрицы A и B размера $N \times N$, пусть N кратно 16, если нет, то дополняем матрицу нулями до кратной 16. Требуется перемножить две матрицы и получить результирующую матрицу C размера $N \times N$.

Простое решение.

Простейший вариант использует по одной нити на каждый элемент получающейся матрицы C, при этом поток извлекает все необходимые элементы из глобальной памяти и производит требуемые вычисления. $C_{ij} += A_{ik} \cdot B_{kj}$ где $k = 1..N$. Каждый поток перемножает по одной строке и столбцу и суммирует результат.

Тем самым для вычисления одного элемента матрицы C нужно выполнить $2 \cdot N$ арифметических операций и $2 \cdot N$ чтений из глобальной памяти, скорость доступа к которой очень низкая. Арифметические операции будут выполняться мгновенно, а вот частое чтение из памяти будет сильно влиять на производительность. Но есть возможность ускорить работу программы за счёт меньшего числа дорогих по времени обращений к памяти.

Блочный метод умножения.

Можно заметно повысить быстродействие программы если использовать shared-памяти сократив количество обращений к глобальной. Для этого разбиваем результирующую матрицу C на подматрицы размером 16×16. Каждую такую подматрицу будет вычислять один блок потоков. Для вычисления такой подматрицы будем использовать «полоски» матриц A и B, обозначим их за A' и B'.

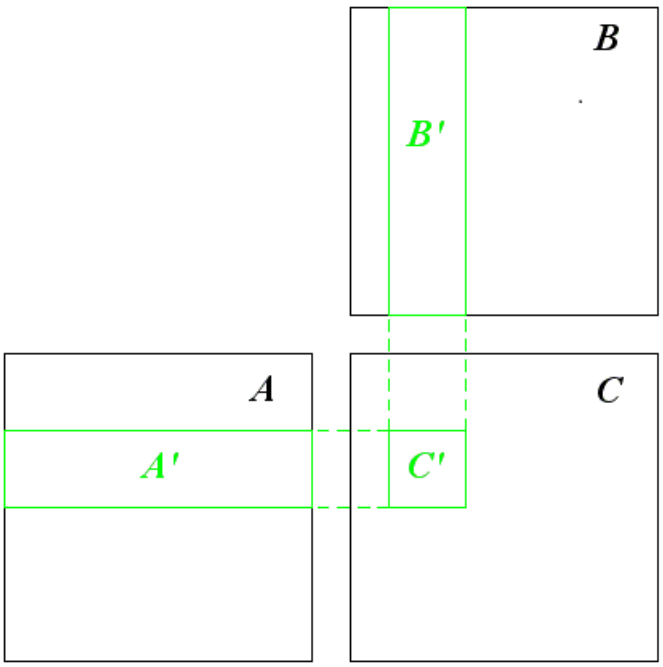

Рисунок 5 — Разбиение результирующей и выделение полос

Так как целиком полосы в shared-память не поместятся из-за небольшого объема shared-памяти. Поэтому разбиваем эти «полоски» на матрицы размером 16×16 и вычисление подматрицы произведения матриц будем проводить в $N/16$ шагов. На каждом шаге загружаем в shared-память по одной 16×16 подматрице A' и одной 16×16 подматрице B', вычисляя соответствующую им сумму для элементов произведения. На каждом шаге один поток загружает по одному элементу из каждой из матриц A и B и вычисляет соответствующую им сумму и после всех вычислений сохраняем результат.

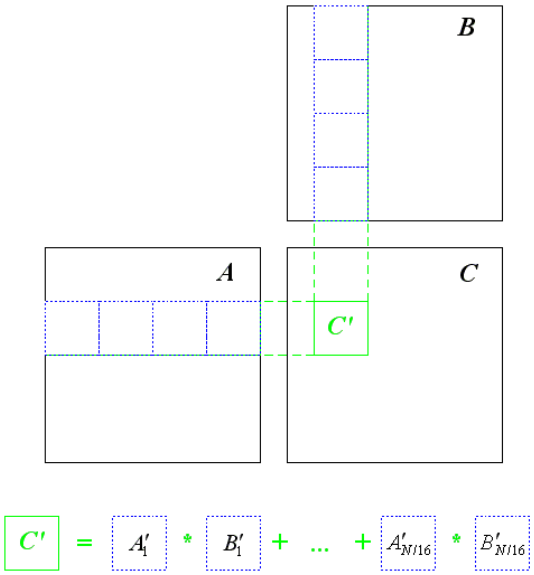

Рисунок 6 — Разбиение результирующей и выделение полос

В итоге результирующая подматрица $C' = A'_1 \cdot B'_1 + ... + A'_{N/16} \cdot B'_{N/16}$ и так для всех подматриц C'.

Также, важно синхронизировать потоки (synchronize) после загрузки элементов из A и B и после обработки уже загруженных. Это нужно для того, чтобы к моменту начала расчётов все нужные элементы были загружены и чтобы загружать следующие после обработки предыдущих.

Теперь для вычисления одного элемента произведения матриц нам нужно всего $2 \cdot N/16$ чтений из глобальной памяти. И по результатам сразу видно за счет использования shared-памяти нам удалось поднять быстродействие более чем на порядок.

Далее будут представлены результаты обеих программ.

2.2 Результаты и сравнение для перемножения матриц

Ниже представлена таблица времени выполнения умножения матриц в двух вариантах и для разных систем.

Таблица 1 — Простое перемножение матриц

Размер:	1000	3000	5000
CPU:	8.892c	340.094c	1734.437c
GPU:	0.093c	2.277c	10.468c

Для блочного метода привожу результаты для размера 5000 на 5000 при работе с CPU. Результат взят с fastcomputing.org

Таблица 2 — Блочное перемножение матриц

Размер:	1000	3000	5000
CPU:	-	-	28.407c
GPU:	0.066c	1.592c	7.233c

Как видно по результатам, вычисления на видиокарте обходят по времени вычисления на одном ядре процессора даже при сильной оптимизации (блочный метод).

2.3 Нахождение обратной матрицы

Постановка задачи

Пусть у нас есть квадратная матрица A размера $N \times N$, пусть N кратно 16. Необходима найти такую матрицу A^{-1},что $A^{-1} \cdot A = A \cdot A^{-1} = E$, где E — единичная матрица размера $N \times N$. A^{-1} будет являться обратной матрицей по отношению к матрице A.

Нахождение обратной матрици методом Жордана-Гаусса

К матрице A приписываем справа единичную матрицу E того же порядка, получая расширенную матрицу $(A|E)$ размерности $N \times 2 \cdot N$, после этого, с

помощью элементарных преобразований над расширенной матрицей, добиваемся получения, на месте исходной матрицы A,единичную матрицу, при этом расширен - ная матрица приобретает вид $(E|A^{-1})$ и мы получаем искомую обратную матрицу.

Мы можем применять следующие элементарные преобразования:

1. Обмен местами двух строк;
2. Умножение всех элементов строки на некоторое, не равное нулю, число;
3. Прибавление к элементам одной строки соответствующих элементов другой, умноженных на любой отличный от нуля множитель.

Метод Жордана-Гаусса состоит из следующих шагов [6]:

1. Выбираем первый слева столбец матрицы, с хотя бы одним отличным от нуля значением.
2. Если самое верхнее число в этом столбце ноль, то меняем всю первую строку матрицы с другой строкой матрицы, где в данном столбце нет нуля.
3. Все элементы первой строки делим на верхний элемент выбранного столбца.
4. Из оставшихся строк вычитают первую строку, умноженную на первый элемент соответствующей строки, с целью получить первым элементом каждой строки (кроме первой) ноль.
5. Далее проводим такую же процедуру с матрицей, получающейся из исходной матрицы после вычёркивания первой строки и первого столбца.
6. После повторения этой процедуры $N-1$ раз получаем верхнюю треугольную матрицу.

7. Вычитаем из предпоследней строки последнюю строку, умноженную на соответствующий коэффициент, так , чтобы в предпоследней строке осталась только 1 на главной диагонали.

8. Повторяем предыдущий шаг для последующих строк. В итоге получаем единичную матрицу и решение на месте исходной единичной матрицы (с ней необходимо проводить все те же преобразования).

Реализация

Далее приводится общие шаги для метода Жордана-Гаусса адаптированного для использования на GPU с указанием отвечающих за данный шаг функций:

1. Определяем необходимость в перестановке строк, нахождение строки на замену (функция validateRows);

2. Переставляем строки матрицы,выбранные на предыдущем шаге (функция swapRows);

3. Приводим строку в матрице к единичному виде. Получам еденицу в сроке на месте диагонального элемента, путём деления всех элементов строки на одно и тоже число (функция normalizeRow);

4. Выполнить проход, обнуляя элементы над и под диагональю (функция inversRow);

Первый шаг алгоритма линеен, но выполняется всё равно на GPU в связи с медленной скоростью обращения к памяти. На данном шаге проверяем, равняется ли нуля диагональный элемент матрицы и ищется строка для замены.

Последний шаг объединяет в себе прямой и обратный проходы классического метода Гаусса.

Все перечисленные шаги последовательно выполняются для каждой строки матрицы. Вся польза GPU раскрывается на последнем шаге,когда приходится параллельно обрабатывать все элементы матрицы,приводя её к единичному виду.

Сравнение результатов обращения матрици

Ниже представлены результаты нахождения обратной матрицы методом Жордана-Гаусса.

Таблица 3 — Нахождение обратной матриц

Размер:	512	1024	2048	4096
CPU:	1.661с	13.501с	107.956с	890.183с
GPU:	1.201с	1.402с	8.261с	64.609с

Как видно из таблицы, для небольших размерностей матрицы, время выполнения примерно одинаковое. Однако с увеличением размера матрицы, разница во времени выполнения становиться всё заметнее. Это связано со скоростью обработки видеокартой больших объёмов данных на последнем шаге алгоритма, и чем больше данных, тем сильнее виден результат распараллеливания последнего шага.

Заключение

На данном этапе работы, мной изучены основы работы с графической картой Nvidia и программирование с использованием технологии CUDA. Были реализованы программы сложения двух векторов, перемножения матриц(в дух вариантах) на GPU и нахождение обратной матрицы. Результаты программ были сравнены с результатами аналогичных программ написанных для CPU. Судя по времени вычислений разница ощутима, и видеокарта несомненно выигрывает одно ядро процессора и чем больше данных для обработки, тем больше видна разница.

В дальнейшем необходимо продолжить знакомство с технологией Nvidia CUDA и следующей задачей для реализации и сравнение будет нахождение обратной по модулю матрицы.

Список использованных источников

[1] Nvidia Cuda programming guide. From graphics processing to general purpose parallel computing [Электронный ресурс] // URL:http://docs.nvidia.com/cuda/cuda-c-programming-guide/#from-graphics-processing-to-general-purpose-parallel-computing. Яз. англ.

[2] NVIDIA CUDA — неграфические вычисления на графических процессорах [Электронный ресурс] // URL:http://www.ixbt.com/video3/cuda-1.shtml. Яз.русс.

[3] NVIDIA CUDA Programming Guide [Электронный ресурс] // URL:http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf. Яз.англ.

[4] Введение в технологию CUDA [Электронный ресурс] // URL:http://cgm.computergraphics.ru/issues/issue16/cuda Яз.русс.

[5] CUDA: Новая архитектура для вычислений на GPU [Электронный ресурс] // URL:http://www.nvidia.ru/content/cudazone/download/ru/CUDA_for_games.pdf. Яз.русс.

[6] Метод Гаусс-Жордана [Электронный ресурс] // URL:http://ru.wikipedia.org/wiki/Метод_Жордана-Гаусса. Яз.русс.