**Содержание**

[Введение 2](#_Toc279697584)

[1. Обзор архитектуры CUDA 3](#_Toc279697585)

[1.1. Архитектура ГП, поддерживающего CUDA 3](#_Toc279697586)

[1.2. Организация памяти 4](#_Toc279697587)

[1.3. Модель программирования 7](#_Toc279697588)

[1.4. Среда программирования 10](#_Toc279697589)

[2. Разработка программы 12](#_Toc279697590)

[2.1. Разработка алгоритма 12](#_Toc279697591)

[2.2. Оптимизация алгоритма 13](#_Toc279697592)

[3. Результаты тестирования 24](#_Toc279697593)

[Выводы 26](#_Toc279697594)

[Список использованной литературы 27](#_Toc279697595)

# Введение

На сегодняшний день, графические процессоры (ГП) – это высокопроизводительные многоядерные процессоры, которые предназначены для обработки больших объемов данных (карты текстур, полигоны и т.п.), над которыми выполняется одна и та же операция. Следовательно, для достижения максимальной производительности, современные ГП построены на основе архитектуры SIMD. Благодаря этому, они намного эффективнее в обработке графической информации, чем типичный центральный процессор (ЦП).

Отличительными особенностями ГП, по сравнению с ЦП являются:

* архитектура, максимально нацеленная на увеличение скорости расчёта текстур и сложных графических объектов;
* ограниченный набор команд.

Графический процессор в современных видеоадаптерах применяется в качестве ускорителя трёхмерной графики, однако его можно использовать в некоторых случаях и для универсальных вычислений (General-purpose graphics processing units — «ГП общего назначения»)[1]. Это направление является весьма перспективным и позволяет получить большое ускорение для задач с естественным параллелизмом. При этом, ГП выступает в качестве математического сопроцессора для центрального процессора (ЦП). Выполнение выглядит следующим образом:

* Данные для обработки создаются в оперативной памяти (ОП) при участии ЦП.
* Затем данные пересылаются в видеопамять графического ускорителя.
* Данные обрабатываются на ГП.
* Результат вычислений пересылается обратно в ОП.

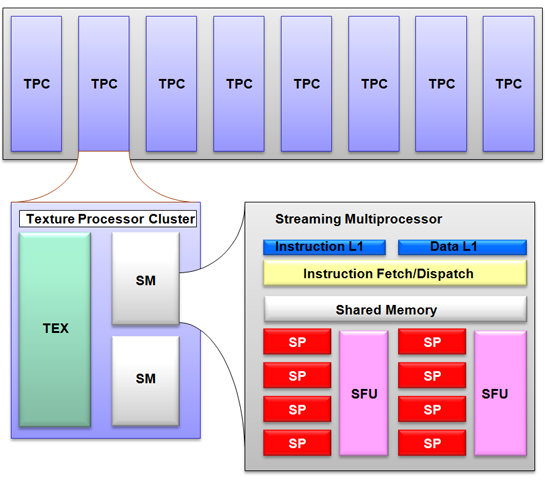
При этом, ЦП практически не несет никакой нагрузки и может быть использован для каких-либо других вычислений.

# Обзор архитектуры CUDA

NVidia Compute Unified Device Architecture (CUDA) – это программная архитектура для организации и управления вычислениями для ГП как для параллельного вычислительного устройства без потребности проецирования их на графический программный интерфейс приложения. [15]

## Архитектура ГП, поддерживающего CUDA

Главным вычислительным элементом видеокарты, поддерживающей архитектуру **CUDA,** является ядро шейдеров (рис. 1.1.) [1], которое состоит из нескольких кластеров текстурных процессоров (Texture Processor Cluster, TPC).



* 1. Ядро шейдеров

Различные модели видеокарт имеют разное число кластеров [7]. Каждый кластер, по сути, состоит из текстурного блока и двух потоковых мультипроцессоров (streaming multiprocessor). Последние включают начало конвейера (front end), выполняющее чтение и декодирование инструкций, а также отсылку их на выполнение, и конец конвейера (back end), состоящий из восьми вычислительных устройств и двух суперфункциональных устройств SFU (Super Function Unit), где инструкции выполняются по принципу SIMT, то есть одна инструкция применяется ко всем потокам в варпе.

Каждый мультипроцессор обладает определённым набором ресурсов: eсть небольшая область памяти под названием "Разделяемая память/Shared Memory", по 16 кбайт на мультипроцессор. Данная область памяти открывает возможность обмена информацией между потоками в одном блоке. Потоки также могут использовать видеопамять, но с меньшей пропускной способностью и большими задержками. Поэтому, чтобы снизить частоту обращения к этой памяти, nVidia оснастила мультипроцессоры кэш памятью (8 кбайт на мультипроцессор), хранящим константы и текстуры. [3]

Мультипроцессор имеет 8192 регистров, которые являются общими для потоков всех блоков, активных на мультипроцессоре. Число активных блоков на мультипроцессор не может превышать восьми, а число активных варпов ограничено 24 (768 потоков). Знание этих ограничений позволяет оптимизировать алгоритмы под доступные ресурсы. [8]

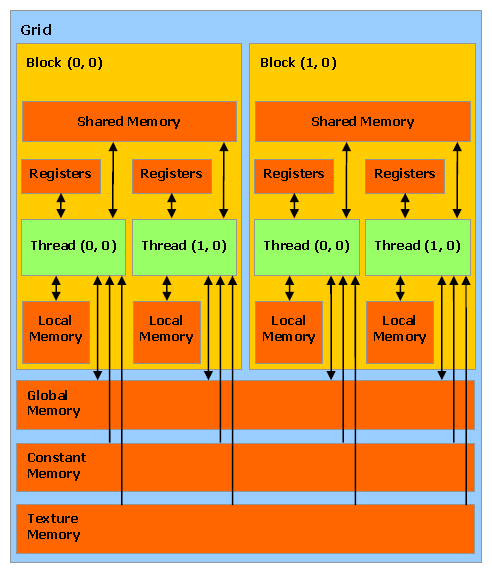
## Организация памяти

Модель памяти в CUDA отличается возможностью побайтной адресации, поддержкой как gather, так и scatter. Доступно довольно большое количество регистров на каждый потоковый процессор, до 1024 штук. Доступ к ним очень быстрый, хранить в них можно 32-битные целые или числа с плавающей точкой. [9]

Каждый поток имеет доступ к следующим типам памяти (рис 1.2):

Глобальная память — самый большой объём памяти, доступный для всех мультипроцессоров на видеочипе, размер составляет от 256 мегабайт до 1.5 гигабайт на текущих решениях (и до 4 Гбайт на Tesla). Обладает высокой пропускной способностью, более 100 гигабайт/с для топовых решений NVIDIA, но очень большими задержками в несколько сот тактов. Не кэшируется, поддерживает обобщённые инструкции load и store, и обычные указатели на память.

Локальная память — это небольшой объём памяти, к которому имеет доступ только один потоковый процессор. Она относительно медленная — такая же, как и глобальная.



* 1. Схема организации памяти.

Разделяемая память — это 16-килобайтный (в видеочипах нынешней архитектуры) блок памяти с общим доступом для всех потоковых процессоров в мультипроцессоре. Эта память весьма быстрая, такая же, как регистры. Она обеспечивает взаимодействие потоков, управляется разработчиком напрямую и имеет низкие задержки. Преимущества разделяемой памяти: использование в виде управляемого программистом кэша первого уровня, снижение задержек при доступе исполнительных блоков (ALU) к данным, сокращение количества обращений к глобальной памяти.

Память констант — область памяти объемом 64 килобайта (то же — для нынешних GPU), доступная только для чтения всеми мультипроцессорами. Она кэшируется по 8 килобайт на каждый мультипроцессор. Довольно медленная — задержка в несколько сот тактов при отсутствии нужных данных в кэше.

Текстурная память — блок памяти, доступный для чтения всеми мультипроцессорами. Выборка данных осуществляется при помощи текстурных блоков видеочипа, поэтому предоставляются возможности линейной интерполяции данных без дополнительных затрат. Кэшируется по 8 килобайт на каждый мультипроцессор. Медленная, как глобальная — сотни тактов задержки при отсутствии данных в кэше. [11]

Типы памяти в CUDA. Таблица 1.1.

|  |  |  |  |
| --- | --- | --- | --- |
| Тип памяти | Доступ | Уровень выделения | Скорость работы |
| регистры (registers) | R/W | per-thread | высокая (on chip) |
| local | R/W | per-thread | низкая (DRAM) |
| shared | R/W | per-block | высокая (on-chip) |
| global | R/W | per-grid | низкая(DRAM) |
| constant | R/O | per-grid | высокая(on chip L1 cache) |
| texture | R/O | per-grid | высокая(on chip L1 cache) |

Естественно, что глобальная, локальная, текстурная и память констант — это физически одна и та же память, известная как локальная видеопамять видеокарты [10]. Их отличия в различных алгоритмах кэширования и моделях доступа. Центральный процессор может обновлять и запрашивать только внешнюю память: глобальную, константную и текстурную.

## Модель программирования

CUDA использует параллельную модель вычислений, когда каждый из SIMD процессоров выполняет ту же инструкцию над разными элементами данных параллельно. GPU является вычислительным устройством, сопроцессором (device) для центрального процессора (host), обладающим собственной памятью и обрабатывающим параллельно большое количество потоков. Ядром (kernel) называется функция для GPU, исполняемая потоками (аналогия из 3D графики — шейдер). [13]

Основной процесс приложения CUDA работает на универсальном процессоре (host), он запускает несколько копий процессов kernel на видеокарте. Код для CPU делает следующее: инициализирует GPU, распределяет память на видеокарте и системе, копирует константы в память видеокарты, запускает несколько копий процессов kernel на видеокарте, копирует полученный результат из видеопамяти, освобождает память и завершает работу.

Видеочип отличается от CPU тем, что может обрабатывать одновременно десятки тысяч потоков, что обычно для графики, которая хорошо распараллеливается. Каждый поток скалярен, не требует упаковки данных в 4-компонентные векторы, что удобнее для большинства задач. Количество логических потоков и блоков потоков превосходит количество физических исполнительных устройств, что даёт хорошую масштабируемость для всего модельного ряда решений компании. [14]

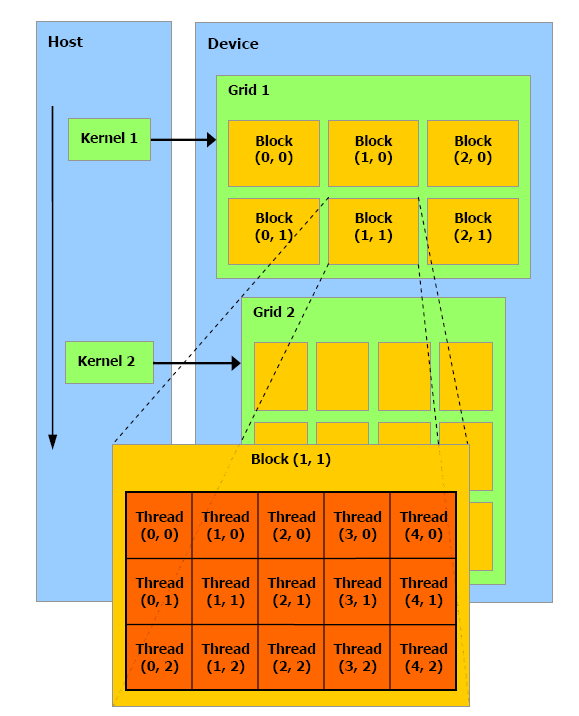
Функции, исполняемые видеочипом, имеют следующие ограничения: отсутствует рекурсия, нет статических переменных внутри функций и переменного числа аргументов. Поддерживается два вида управления памятью: линейная память с доступом по 32-битным указателям, и CUDA-массивы с доступом только через функции текстурной выборки. [4]

Модель программирования в CUDA предполагает группирование потоков. Потоки объединяются в блоки потоков (thread block) — одномерные или двумерные сетки потоков, взаимодействующих между собой при помощи разделяемой памяти и точек синхронизации. Программа (ядро, kernel) исполняется над сеткой (grid) блоков потоков (thread blocks) (рис 1.3.). Одновременно исполняется одна сетка. Каждый блок может быть одно-, двух- или трехмерным по форме, и может состоять из 512 потоков на текущем аппаратном обеспечении.

Блоки потоков выполняются в виде небольших групп, называемых варп (warp). Варп в CUDA представляет собой группу из 32 потоков и является минимальным объёмом данных, обрабатываемых SIMT-способом в мультипроцессорах CUDA. Также используется понятие half-warp'а - это первая или вторая половина варпа. Подобное разбиение варпа на половины связано с тем, что обычно обращение к памяти делаются отдельно для каждого варпа.

Преимущество подобной группировки заключается в том, что число блоков, одновременно обрабатываемых GPU, тесно связано с аппаратными ресурсами, как мы увидим ниже. Группировка блоков в сетки позволяет полностью абстрагироваться от этого ограничения и применить ядро/kernel к большему числу потоков за один вызов, не думая о фиксированных ресурсах. Если у GPU недостаточно ресурсов, он будет выполнять блоки последовательно. В обратном случае, блоки могут выполняться параллельно, что важно для оптимального распределения работы на видеочипах разного уровня, начиная от мобильных и интегрированных. [11]

Программы на CUDA могут взаимодействовать с графическими API: для рендеринга данных, сгенерированных в программе, для считывания результатов рендеринга и их обработки средствами CUDA (например, при реализации фильтров постобработки). Для этого ресурсы графических API могут быть отображены (с получением адреса ресурса) в пространство глобальной памяти CUDA. Поддерживаются следующие типы ресурсов графических API: Buffer Objects (PBO / VBO) в OpenGL, вершинные буферы и текстуры (2D, 3D и кубические карты) Direct3D9. [9]

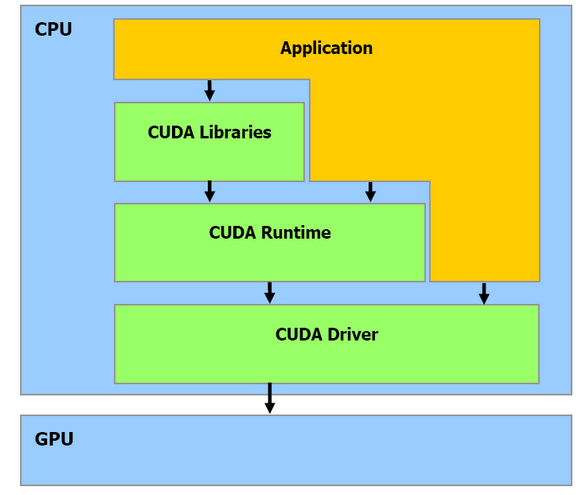


* 1. Структура потоков CUDA

## Среда программирования

В CUDA имеется набор программных уровней для работы с GPU высокоуровневый API: CUDA Runtime API, низкоуровневый API: CUDA Driver API и уровень библиотек [2] (рис. 1.4).

Программы для CUDA (соответствующие файлы обычно имеют расширение *.cu*) пишутся на "расширенном" С и компилируются при помощи компилятора nvcc. [12]

[](http://www.thg.ru/graphic/nvidia_cuda/nvidia_cuda_screenshots_6_1.html)

* 1. **Программные уровни CUDA**

Компилируется программа CUDA весьма интересным способом: компиляция выполняется в несколько этапов. Сначала извлекается код, относящийся к CPU, который передаётся стандартному компилятору. Код, предназначенный для GPU, сначала преобразовывается в промежуточный язык PTX. Он подобен ассемблеру и позволяет изучать код в поисках потенциальных неэффективных участков. Наконец, последняя фаза заключается в трансляции промежуточного языка в специфические команды GPU и создании двоичного файла [10] (рис. 1.4.2.).



* 1. Схема компиляции программы CUDA

# Разработка программы

## Разработка алгоритма

Одной из часто встречающихся задач являются так называемая редукция массива. Пусть у нас есть массив *a0,a1,...,aN-1* и некоторая бинарная операция или функция (например сложение).

Тогда следующее выражение будет называться редукцией массива *a0,a1,...,aN-1* относительно заданной операции (в нашем случае сложения):

*A=((((a0+a1)+(a2+a3))+((a4+a5)+(a6+a7)))+...+aN-1*

В качестве бинарной операции также может выступать умножение, минимум или максимум, давая в результате произведение всех элементов массива или минимальный/максимальный элемент массива. Точно также скалярное произведение двух больших массивов является редукцией поэлементного произведения этих массивов.

На примере реализации редукции в CUDA мы рассмотрим основные элементы оптимизации программы.

Обратите внимание на то, что в CUDA есть ограничение - количество блоков по каждому измерению (т.е. размер *grid*'а по каждому измерению не может превышать 65535). В нашем случае, поскольку мы работаем с одномерным массивом и одномерный *grid* является наиболее подходящим способом организации блоков, это накладывает ограничения на размер входного массива. Однако данное ограничение не принципиально - можно легко переписать рассматриваемые ниже варианты для двух- или трехмерных *grid* 'ов.

Первое с чем нужно определиться - это что является лимитирующим быстродействие фактором - арифметические операции или доступ к памяти. В нашем случае очевидно, что лимитировать нас будет именно доступ к памяти и именно этот доступ и нужно оптимизировать в первую очередь.

Первый шаг довольно прост - каждому блоку соответствует часть массива (делим массив поровну между всеми блоками). Задача блока найти сумму всех элементов своей части и записать результирующее значение в выходной массив. При этом каждой нити соответствует по одному элементу массива, сначала каждая нить загружает свой элемент в *shared*-память, а затем иерархически суммирует, как показано на рис 6.

Рис. 2.1. Иерархическое суммирование элементов внутри блока.

## Оптимизация алгоритма

Далее мы рассмотрим именно оптимизацию этого шага - нахождение каждым блоком своей суммы наиболее эффективным образом. Будем использовать одномерный массив блоков, где каждый блок является одномерным массивом нитей. Ниже приводится ядро, осуществляющее параллельную редукцию описанным методом:

\_\_global\_\_ void reduce0(int \*g\_idata, int \*g\_odata) {

extern \_\_shared\_\_ int sdata[];

// Каждый поток загружает по одному элементу с глобальной в общую память

unsigned int tid = threadIdx.x;

unsigned int i = blockIdx.x\*blockDim.x + threadIdx.x;

sdata[tid] = g\_idata[i];

\_\_syncthreads();

// выполнение редукции в общей памяти

for(unsigned int s=1; s < blockDim.x; s \*= 2) {

if (tid % (2\*s) == 0) {

sdata[tid] += sdata[tid + s];

}

\_\_syncthreads();

}

// запись результата редукции в глобальную память

if (tid == 0) g\_odata[blockIdx.x] = sdata[0];

}

На следующем рисунке проиллюстрирована схема работы данного подхода.

Рис. 2.2. Схема работы

##### Минимизация ветвлений

Сразу виден большой недостаток этого подхода - условный оператор приводит к сильному ветвлению кода внутри каждого *warp*'а. Его можно избежать переписав ядро следующим образом:

Заменим ветвление

for (unsigned int s=1; s < blockDim.x; s \*= 2) {

**if (tid % (2\*s) == 0) {**

**sdata[tid] += sdata[tid + s];**

**}**

\_\_syncthreads();

}

На чередующуюся индексную адресацию

for (unsigned int s=1; s < blockDim.x; s \*= 2) {

**int index = 2 \* s \* tid;**

**if (index < blockDim.x) {**

**sdata[index] += sdata[index + s];**

**}**

\_\_syncthreads();

}

Рис. 2.3. Схема работы

Как видно из рисунка, получается почти такая же схема, но распределение операций и элементов по нитям изменилось, за счет чего и удалось получить минимальное ветвление.

##### Устранение конфликтов банков памяти

Однако данный подход имеет серьезный недостаток - он приводит в большому числу конфликтов банков при обращении к *shared*-памяти - так при *s>1* у нас не будет ни одного обращения к банкам с нечетными номерами, а на банки с четными номерами придется двойная нагрузка. По мере увеличения *s* количество неиспользуемых банков (а, значит, и нагрузка на остальные) будет только расти..

Для того, чтобы избежать подобной проблемы реорганизуем саму схему суммирования - начнем суммировать не с соседних элементов, а наоборот - с элементов, удаленных друг от друга на *dimBlock.x/2*. На следующем шаге будем суммировать элементы, удаленные друг от друга на *dimBlock.x/4* и т.д. (см. рис 2.4.).

Рис. 2.4. Схема работы

Соответствующая схема реализуется следующим образом:

Заменяем вложенный цикл:

for (unsigned int s=1; s < blockDim.x; s \*= 2) {

**int index = 2 \* s \* tid;**

**if (index < blockDim.x) {**

**sdata[index] += sdata[index + s];**

**}**

\_\_syncthreads();

}

Соответствующим циклом с обратной индексацией:

for (unsigned int s=blockDim.x/2; s>0; s>>=1) {

**if (tid < s) {**

**sdata[tid] += sdata[tid + s];**

**}**

\_\_syncthreads();

}

##### Устранение простоя нитей

Хотя мы заметно сократили число конфликтов банков в *shared*-памяти, но в результате получили что на первой итерации цикла по *s* половина всех нитей простаивает.

Хотя с точки зрения ветвления в этом никаких проблем нет (при большом размере блока, кратном 64, у нас все простаивающие нити будут собраны в *warp*'ы), но все равно имеется неэффективность, которую хочется удалить.

Хочется, чтобы уже на первой итерации цикла все нити были загружены. Для этого необходимо уменьшить вдвое количество блоков, но при этом каждому блоку выделить вдвое больше слов. При этом самое первое суммирование может быть выполнено сразу же при загрузке данных в *shared*-память (т.е. мы не увеличиваем требуемый объем *shared*-памяти):

Вместо подгрузки одного слова:

unsigned int tid = threadIdx.x;

**unsigned int i = blockIdx.x\*blockDim.x + threadIdx.x;**

**sdata[tid] = g\_idata[i];**

\_\_syncthreads();

Будем загружать сразу 2 и выполнять их суммирование:

unsigned int tid = threadIdx.x;

unsigned int i = blockIdx.x**\*(blockDim.x\*2)** + threadIdx.x;

**sdata[tid] = g\_idata[i] + g\_idata[i+blockDim.x];**

\_\_syncthreads();

##### Разворачивание циклов

Заметим, что при *s<=32* у нас в каждом блоке останется всего по одному *warp*'у, поэтому синхронизация уже не нужна и проверка *tid<s* также не нужна (она все равно ничего в этом случае не делает). Поэтому развернем цикл для *s<=32*:

for (unsigned int s=blockDim.x/2; s>32; s>>=1)

{

if (tid < s)

sdata[tid] += sdata[tid + s];

\_\_syncthreads();

}

if (tid < 32)

{

sdata[tid] += sdata[tid + 32];

sdata[tid] += sdata[tid + 16];

sdata[tid] += sdata[tid + 8];

sdata[tid] += sdata[tid + 4];

sdata[tid] += sdata[tid + 2];

sdata[tid] += sdata[tid + 1];

}

Если бы знать количество итераций во время компиляции – можно было бы полностью развернуть всю редукцию. Но мы знаем, что максимальное количество потоков в блоке – 512, поэтому мы легко можем написать шаблон, в котором развернем циклы для всех возможных вариантов размера блока:

template <unsigned int blockSize>  
\_\_global\_\_ void reduce5(int \*g\_idata, int \*g\_odata)

…

if (blockSize >= 512) {

if (tid < 256) { sdata[tid] += sdata[tid + 256]; } \_\_syncthreads();

}

if (blockSize >= 256) {

if (tid < 128) { sdata[tid] += sdata[tid + 128]; } \_\_syncthreads();

}

if (blockSize >= 128) {

if (tid < 64) { sdata[tid] += sdata[tid + 64]; } \_\_syncthreads();

}

if (tid < 32) {

if (blockSize >= 64) sdata[tid] += sdata[tid + 32];

if (blockSize >= 32) sdata[tid] += sdata[tid + 16];

if (blockSize >= 16) sdata[tid] += sdata[tid + 8];

if (blockSize >= 8) sdata[tid] += sdata[tid + 4];

if (blockSize >= 4) sdata[tid] += sdata[tid + 2];

if (blockSize >= 2) sdata[tid] += sdata[tid + 1];

}

##### Объединение запросов к памяти

Крайне важной особенностью GPU является возможность объединения (*coalescing global memory accesses*) нескольких запросов в глобальную память в одну операция над блоком (транзакцию). Подобное объединение запросов в один запрос чтения/записи одного блока длиной 32/64/128 байт может происходить в пределах одного *half-warp*'а. Для того, чтобы такое объединение произошло, должны быть выполнены специальные требования на то, как отдельные нити *half-warp*'а обращаются к памяти.

Прежде всего получающийся общий блок обязательно должен быть выровнен в памяти(т.е. его адрес должен быть кратен его размеру). Также должны быть выполнены дополнительные требования, зависящие от *Compute Capability* GPU.

Для GPU с *Compute Capability* 1.0 и 1.1 объединения запросов в одну транзакцию будет происходит при выполнении следующих условий:

* нити должны обращаться либо к 32-битовым словам, давая при этом в результате один 64-байтовый блок (транзакцию), либо к 64-битовым словам, давая при этом один 128-байтовый блок (транзакцию);
* все 16 слов должны лежать в пределах данного блока (64 или 128 байт)
* нити должны обращаться к словам последовательно, т.е. *k*-ая нить должна обращаться к *k*-му слову (при этом допускается что какие-то нити вообще не производят обращения к соответствующим им словам).

Если нити *half-warp*'а не удовлетворяют какому-либо из данных условий, то каждое обращение к памяти происходит как отдельная транзакция. На следующих рисунках приводятся типичные паттерны обращения дающие объединения и не дающие объединения.

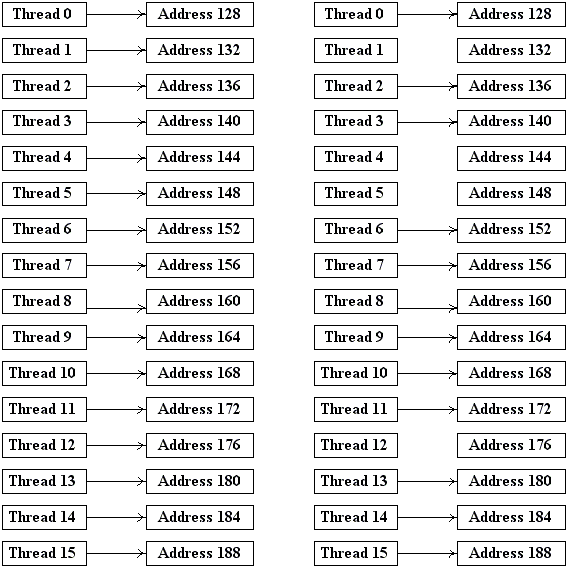


Рис 2.5. Паттерны обращения к памяти, дающие объединение для *Compute Capability* 1.0 и 1.1.

На рис. 2.5. приведены типичные паттерны обращения к памяти, приводящие к объединению запросов в одну транзакцию. Слева у нас выполнены все условия, справа - просто для части нитей пропущено обращение к соответствующим словам (что равно позволяет добавить фиктивные обращения и свести к случаю слева).

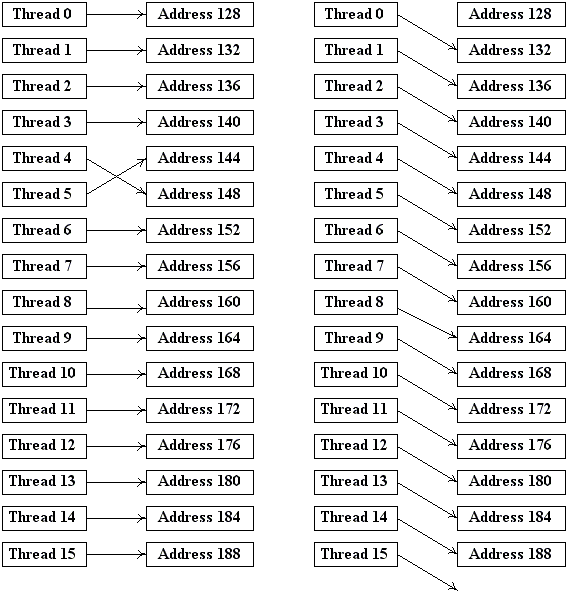


Рис 2.6. Паттерны обращения к памяти, не дающие объединение для *Compute Capability* 1.0 и 1.1.

На рис. 2.6. слева для нитей 4 и 5 нарушен порядок обращения к словам, а справа нарушено условие выравнивание - хотя слова, к которым идет обращения и образуют непрерывный блок из 64 байт, но начало этого блока (по адресу 132) не кратно его размеру (16 байт).

Для GPU с *Compute Capability* 1.2 и выше объединения запросов в один будет происходить, если слова, к которым идет обращение нитей, лежат в одном сегменте размера 32 байта (если все нити обращаются к 8-битовым словам), 64 байта (если все нити обращаются к 16-битовым словам) и 128 байт (если все нити обращаются к 32-битовым или 64-битовым словам). Получающийся сегмент (блок) должен быть выровнен по 32/64/128 байтам.

Обратите внимание, что в этом случае порядок, в котором нити обращаются к словам, не играет никакой роли и ситуация на рис. 2.6. слева приведет к объединению всех запросов в одну транзакцию.

Если идет обращения к *n* соответствующим сегментам, то происходит группировка запросов в *n* транзакций (только для GPU с *Compute Capability* 1.2 и выше).

Итак, используем эту особенность в нашей программе: будем подгружать из глобальной памяти не по 2 элемента, а по столько, сколько нужно, но не больше, чем размер grid'a, для возможности объединения запросов:

Заменим

unsigned int tid = threadIdx.x;

unsigned int i = blockIdx.x\*(blockDim.x\*2) + threadIdx.x;

**sdata[tid] = g\_idata[i] + g\_idata[i+blockDim.x];**

\_\_syncthreads();

На

unsigned int tid = threadIdx.x;

unsigned int i = blockIdx.x\*(blockSize\*2) + threadIdx.x;

**unsigned int gridSize = blockSize\*2\*gridDim.x;**

**sdata[tid] = 0;**

**while (i < n) {**

**sdata[tid] += g\_idata[i] + g\_idata[i+blockSize];**

**i += gridSize;**

**}**

\_\_syncthreads();

# Результаты тестирования

Тестирование проводилось на машине с ЦП Intel Pentium D (2 ядра) и видеокартой NVidia GeForce 8600 GT (32 потоковых процессоров).

Программа была протестирована на массивах разной размерности.

Замерялось время выполнения программы с использованием каждого ядра. Также было замеряно время выполнения последовательного алгоритма на ЦП.

Таблица 3.1. Результаты тестирования

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Время выполнения (мс) | | | | | | | | |
| Размерность | 8192 | 16384 | 32768 | 65536 | 131072 | 262144 | 524288 | 1048576 |
| CPU | 0,07919 | 0,14591 | 0,32811 | 0,59829 | 1,20061 | 2,64868 | 4,72510 | 9,69142 |
| Kernel 1  Начальный алгоритм | 0,33886 | 0,41405 | 0,57479 | 0,90329 | 1,44599 | 2,56013 | 4,84184 | 9,33381 |
| Kernel 2  Минимизация ветвлений | 0,34721 | 0,31249 | 0,39696 | 0,57431 | 0,83252 | 1,36320 | 2,37241 | 4,41617 |
| Kernel 3  Устранение конфликтов банков памяти | 0,29224 | 0,31467 | 0,35086 | 0,38677 | 0,54119 | 0,80315 | 1,32973 | 2,33175 |
| Kernel 4  Устранение простоя нитей | 0,26801 | 0,31780 | 0,32291 | 0,33796 | 0,41261 | 0,58053 | 0,87313 | 1,45935 |
| Kernel 5  Развертывание последнего цикла | 0,26890 | 0,26280 | 0,32082 | 0,30993 | 0,37913 | 0,49700 | 0,71343 | 1,10759 |
| Kernel 6  Полное развертывание | 0,26905 | 0,28971 | 0,30328 | 0,33449 | 0,33729 | 0,45331 | 0,60846 | 0,92909 |
| Kernel 7  Объединение запросов к памяти | 0,27285 | 0,25897 | 0,27817 | 0,27298 | 0,28820 | 0,34645 | 0,39312 | 0,49982 |

Рис. 3.1. График времен выполнения программы

# Выводы

В данном курсовом проекте был реализован параллельный алгоритм редукции массива, а также выполнена его оптимизация для выполнения на графических процессорах с поддержкой архитектуры CUDA.

Важнейшим аспектом написания программ на CUDA является их оптимизация. В первую очередь, оптимизация программы CUDA состоит в получении оптимального баланса между количеством блоков и их размером. Больше потоков на блок будут полезны для снижения задержек работы с памятью, но и число регистров, доступных на поток, уменьшается. Также эффективными приемами является минимизация ветвлений, выравнивание данных и объединение запросов к глобальной памяти, разворачивание циклов, минимизация перемещений данных host-device, использование разделяемой памяти вместо глобальной.

Результаты экспериментов показали, что на данных малой размерности ЦП намного эффективнее чем ГП. Это можно объяснить тем, что частота ЦП намного выше, чем частота одного потокового процессора. С увеличением размерности входных данных, эффективность ГП значительно возрастает.

Минимизация ветвлений ускорила выполнение программы в 1.9 раз, разрешение конфликтов банков памяти – в 1.8 раз, устранение простоя нитей – в 1.6 раз, развертывание последнего цикла – в 1.3 раз, полное развертывание - 1,2 раз, объединение запросов к памяти – в 1,9 раз. Суммарное ускорение за счет минимизации – 16,2 раз. Максимальный полученный коэффициент ускорения - 20.

Таким образом, ГП зарекомендовали себя как эффективное и недорогое средство вычислений для задач с естественным параллелизмом.

# Список использованной литературы

1. Frederic P. Miller, Agnes F. Vandome, John McBrewster, GPGPU: Stream Processing, Graphics Processing Unit, Comparison of AMD Graphics Processing Units, Comparison of Nvidia Graphics Processing Units, Graphics Pipeline, Video Card, BrookGPU, Physics Engine: Alphascript Publishing, 2009, **–** 212 p.
2. NVIDIA Corporation, NVIDIA CUDA Development Tools 2.3: 2701 Tomas Expressway, Santa Clara, CA 95050: July 2009, **–** 15р.
3. NVIDIA Corporation, NVIDIA CUDA Programming Guide version 2.3.1: 2701 Tomas Expressway, Santa Clara, CA 95050: July 2009, **–** 145р.
4. NVIDIA Corporation, NVIDIA CUDA Reference manual version 2.3: 2701 Tomas Expressway, Santa Clara, CA 95050: July 2009, **–** 256р.
5. CUDA Zone [Электронный ресурс] : официальный сайт CUDA. — Электрон. дан. — М.: NVIDIA Corporation, 2010. — Режим доступа: http://www.nvidia.ru/object/cuda\_home\_new\_ru.html свободный — Загл. с экрана.
6. [CUDA GPU Computing](http://forums.nvidia.com/index.php?showforum=62) [Электронный ресурс] : официальные веб-форумы, посвящённые вычислениям CUDA. — Электрон. дан. — М.: NVIDIA Corporation, 2010. — Режим доступа: <http://forums.nvidia.com> свободный — Загл. с экрана.
7. [GPGPU.ru](http://www.gpgpu.ru) [Электронный ресурс] : Использование видеокарт для вычислений. — Электрон. дан. — М.: [GPGPU.ru](http://www.gpgpu.ru), 2010. — Режим доступа: <http://www.gpgpu.ru/> свободный — Загл. с экрана.
8. Берилло А. [NVIDIA CUDA — неграфические вычисления на графических процессорах. Часть 1](http://www.ixbt.com/video3/cuda-1.shtml). — М.: iXBT.com (23 сентября 2008 г.).
9. Берилло А. [NVIDIA CUDA — неграфические вычисления на графических процессорах. Часть 2](http://www.ixbt.com/video3/cuda-2.shtml). — М.: iXBT.com (22 октября 2008 г.). — Примеры внедрения NVIDIA CUDA.
10. Боресков А. В. [Основы CUDA](http://www.steps3d.narod.ru/tutorials/cuda-tutorial.html) . — Электрон. дан. — М.: Steps3D, 20 января 2009 г. — Режим доступа: <http://www.steps3d.narod.ru/tutorials/cuda-tutorial.html> свободный — Загл. с экрана.
11. Боресков А.В. [Архитектура и программирование массивно-параллельных вычислительных систем на основе технологии CUDA](http://groups.google.ru/group/cudacsmsusu?lnk=srg&hl=ru&ie=UTF-8). — **Курс лекций на ВМК МГУ. — М.: Москва, 2009**.
12. Осколков И. [NVIDIA CUDA – доступный билет в мир больших вычислений](http://www.computerra.ru/interactive/423392/). — М.: [Компьютерра](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D1%80%D0%B0) (30 апреля 2009 г.).
13. Фролов В. [Введение в технологию CUDA](http://cgm.computergraphics.ru/issues/issue16/cuda). — М.:  Сетевой журнал «Компьютерная графика и мультимедиа» (19 декабря 2008 г.).
14. Фролов В. [Введение в технологию CUDA](http://www.uraldev.ru/articles/id/33). — М.:  Сетевой журнал «Компьютерная графика и мультимедиа» (1 августа 2009 г.).
15. Чеканов Д. [nVidia CUDA: вычисления на видеокарте или смерть CPU](http://www.thg.ru/graphic/nvidia_cuda/print.html). — М.: Tom's Hardware (22 июня 2008 г.).
16. Чеканов Д. Чеканов. [nVidia CUDA: тесты приложений на GPU для массового рынка](http://www.thg.ru/graphic/nvidia_cuda_test/print.html). — М.: Tom's Hardware (19 мая 2009 г.).