Содержание

[1. Архитектура GPU 1](#_Toc34656585)

[2. Отличие архитектуры GPU от архитектуры CPU 2](#_Toc34656586)

[3. Программная модель CUDA 3](#_Toc34656587)

[4. Иерархия памяти GPU 6](#_Toc34656588)

[5. Глобальная память GPU. Шаблон использования, оптимизация доступа, метрики производительности работы с глобальной памятью (в т.ч. метрики профилировщика) 6](#_Toc34656589)

[6. Разделяемая память GPU. Шаблон использования, оптимизация доступа, метрики производительности работы с разделяемой памятью (в т.ч. метрики профилировщика) 6](#_Toc34656590)

[7. Теоретическая и достигнутая заполняемость. Ограничивающие заполняемость факторы. 8](#_Toc34656591)

# **Архитектура GPU**

Разделяют два вида устройств – то которое управляет общей логикой – host, и то которое умеет быстро выполнить некоторый набор инструкций над большим объемом данных – device. В роли хоста обычно выступает центральный процессор (CPU).

В роли вычислительного устройства – видеокарта (GPU). Видеокарта содержит Compute Units – процессорные ядра. В современных игровых видеокартах – их 8-32. Процессорные ядра могут исполнять несколько потоков за счет того, что в каждом содержится несколько (8-16) потоковых процессоров (Stream Cores или Stream Processor). Для карт NVidia – вычисления производятся непосредственно на потоковых процессорах, но ATI ввели еще один уровень абстракции – каждый потоковый процессор, состоит из processing elements – PE (иногда называемых ALU – arithmetic and logic unit) – и вычисления происходят на них.

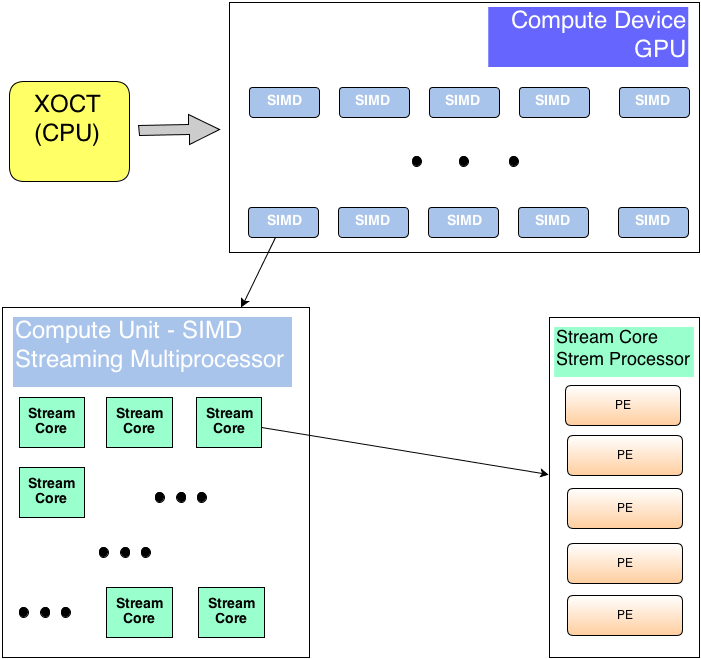


Рисунок 1.1 – Архитектура GPU

GPU предназначен для вычислений,

* параллельных по данным: одна и та же операция выполняется над многими данными параллельно,
* в которых отношение вычислительных операций к числу операций по доступу к памяти велико.

Вместо кэша и сложных элементов управления на кристалле размещено большее число вычислительных элементов.

Латентность памяти покрывается за счет большого количества легковесных потоков.

Постепенно сложность архитектуры GPU повышается.

Состоит из мультипроцессоров (streaming multiprocessor, MP), каждый из которых содержит несколько CUDA-ядер (CUDA core) и общую для них память.

# **Отличие архитектуры GPU от архитектуры CPU**

Основная задача CPU – это выполнение цепочки инструкций за максимально короткое время. CPU спроектирован таким образом, чтобы выполнять несколько таких цепочек одновременно или разбивать один поток инструкций на несколько и, после выполнения их по отдельности, сливать их снова в одну, в правильном порядке. Каждая инструкция в потоке зависит от следующих за ней, и именно поэтому в CPU так мало исполнительных блоков, а весь упор делается на скорость выполнения и уменьшение простоев, что достигается при помощи кэш-памяти и конвейера.

Основная функция GPU — рендеринг 3D графики и визуальных эффектов, следовательно, в нем все немного проще: ему необходимо получить на входе полигоны, а после проведения над ними необходимых математических и логических операций, на выходе выдать координаты пикселей. По сути, работа GPU сводится к оперированию над огромным количеством независимых между собой задач, следовательно, он содержит большой объем памяти, но не такой быстрой, как в CPU, и огромное количество исполнительных блоков: в современных GPU их 2048 и более, в то время как у CPU их количество может достигать 48, но чаще всего их количество лежит в диапазоне 2-8.

CPU отличается от GPU в первую очередь способами доступа к памяти. В GPU он связанный и легко предсказуемый — если из памяти читается тексел текстуры, то через некоторое время настанет очередь и соседних текселов. С записью похожая ситуация — пиксель записывается во фреймбуфер, и через несколько тактов будет записываться расположенный рядом с ним. Также графическому процессору, в отличие от универсальных процессоров, просто не нужна кэш-память большого размера, а для текстур требуются лишь 128–256 килобайт. Кроме того, на видеокартах применяется более быстрая память, и в результате GPU доступна в разы большая пропускная способность, что также весьма важно для параллельных расчетов, оперирующих с огромными потоками данных.

Есть множество различий и в поддержке многопоточности: CPU исполняет 1–2 потока вычислений на одно процессорное ядро, а GPU может поддерживать несколько тысяч потоков на каждый мультипроцессор, которых в чипе несколько штук! И если переключение с одного потока на другой для CPU стоит сотни тактов, то GPU переключает несколько потоков за один такт.

В CPU большая часть площади чипа занята под буферы команд, аппаратное предсказание ветвления и огромные объемы кэш-памяти, а в GPU большая часть площади занята исполнительными блоками. Вышеописанное устройство схематично изображено ниже:



Если подавать на GPU независимые простейшие математические задачи, то он справится значительно быстрее, чем центральный процессор.

# **Программная модель CUDA**

**Программная модель**

С программной стороны CUDA представляет собой расширения языка C/C++ и включает:

* Поток (Thread) — единица исполнения потока команд.
* Блок (Blok) — группа связанных между собой потоков.
* Варп (Warp) — группа потоков внутри потокового блока, которая исполняется физически одновременно (32 потока).
* Сеть (Grid) — набор блоков, который должен быть обработан прежде чем исполнение программы пойдет дальше.

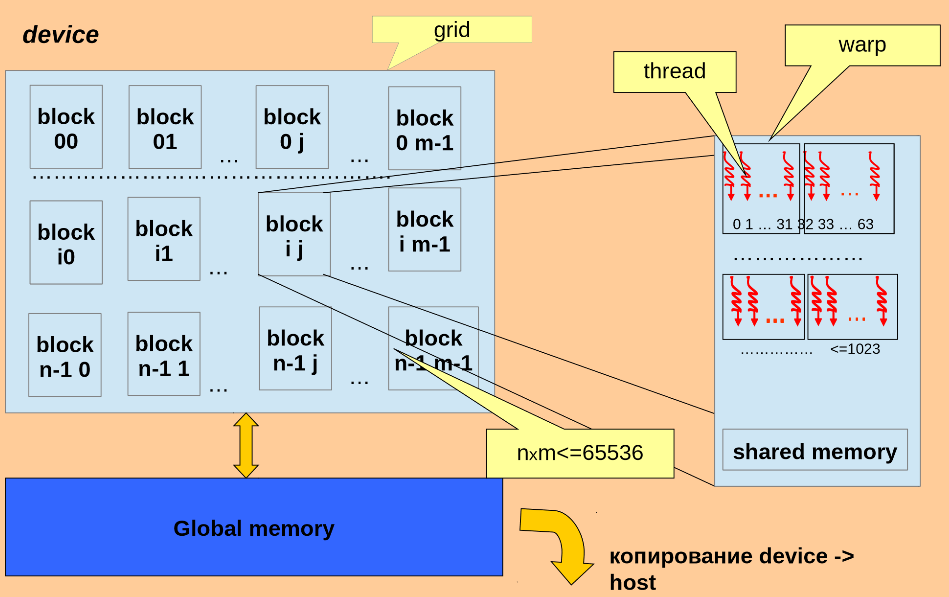


Рисунок 1 – Структура CUDA

Пример функции-ядра (device-часть кода):

\_\_global\_\_ void kernel\_function ( float\* data )

{

int globalIdx;

return;

}

При вызове ядра программист определяет количество нитей в блоке и количество блоков в grid. При этом допустима линейная, двумерная или трехмерная индексация нитей:

gFunc <<< dim3(bl\_xdim, bl\_ydim, bl\_zdim),

dim3(th\_xdim, th\_ydim, th\_zdim) >>> (<params>);

Пусть при запуске задана двумерная сеть из блоков размером H×W и каждый блок содержит M×K потоков. Таким образом область моделирования разбивается на:

* H∙M потоков по вертикали;
* W∙K потоков по горизонтали.

Тогда координаты потока в пространстве:

(blockID.x\*M + threadID.x, blockID.y\*K + threadID.y)

Встроенные переменные-векторы:

* uint3 threadIdx; //индекс нити в блоке
* dim3 blockDim; //размеры блока
* uint3 blockIdx; //индекс блока в сетке
* dim3 gridDim; //размеры сетки

**Модель выполнения**

Блоки выполняются на Stream Multiprocessor по одному блоку на MS. Последовательность исполнения блоков не определена.

Количество блоков на SM определяется количеством регистров, требуемых потоку и количеством разделяемой памяти на блок.

Планировщик потоков периодически передает управление от одного варпа к другому.

Количество нитей в блоке обычно выбирается в программе из соображений производительности, а количество блоков рассчитывается исходя из размерности задачи. Например, в одномерном случае:

blocks = N / threads или blocks = N / threads + 1 (если возникает остаток).

Можно также воспользоваться следующим выражением:

blocks = (N + threads - 1) / threads

Спецификаторы для функций:

|  |  |  |
| --- | --- | --- |
| **Спецификатор** | **Вызывается с** | **Выполняется на** |
| \_\_device\_\_ | device | device |
| \_\_global\_\_ | host, device\* | device |
| \_\_host\_\_ | host | host |

Спецификаторы для переменных:

|  |  |  |  |
| --- | --- | --- | --- |
| **Спецификатор** | **Расположение в памяти** | **Область видимости** | **Время жизни** |
| \_\_device\_\_ | device (global) | сетка | программа |
| \_\_global\_\_ | device (constant) | сетка | программа |
| \_\_host\_\_ | device (shared) | блок | ядро |

Базовые функции для работы с динамической памятью на GPU

cudaError\_t cudaMalloc ( void \*\* devPtr, size\_t size );

выделение size байт памяти на GPU

cudaError\_t cudaFree ( void \* devPtr );

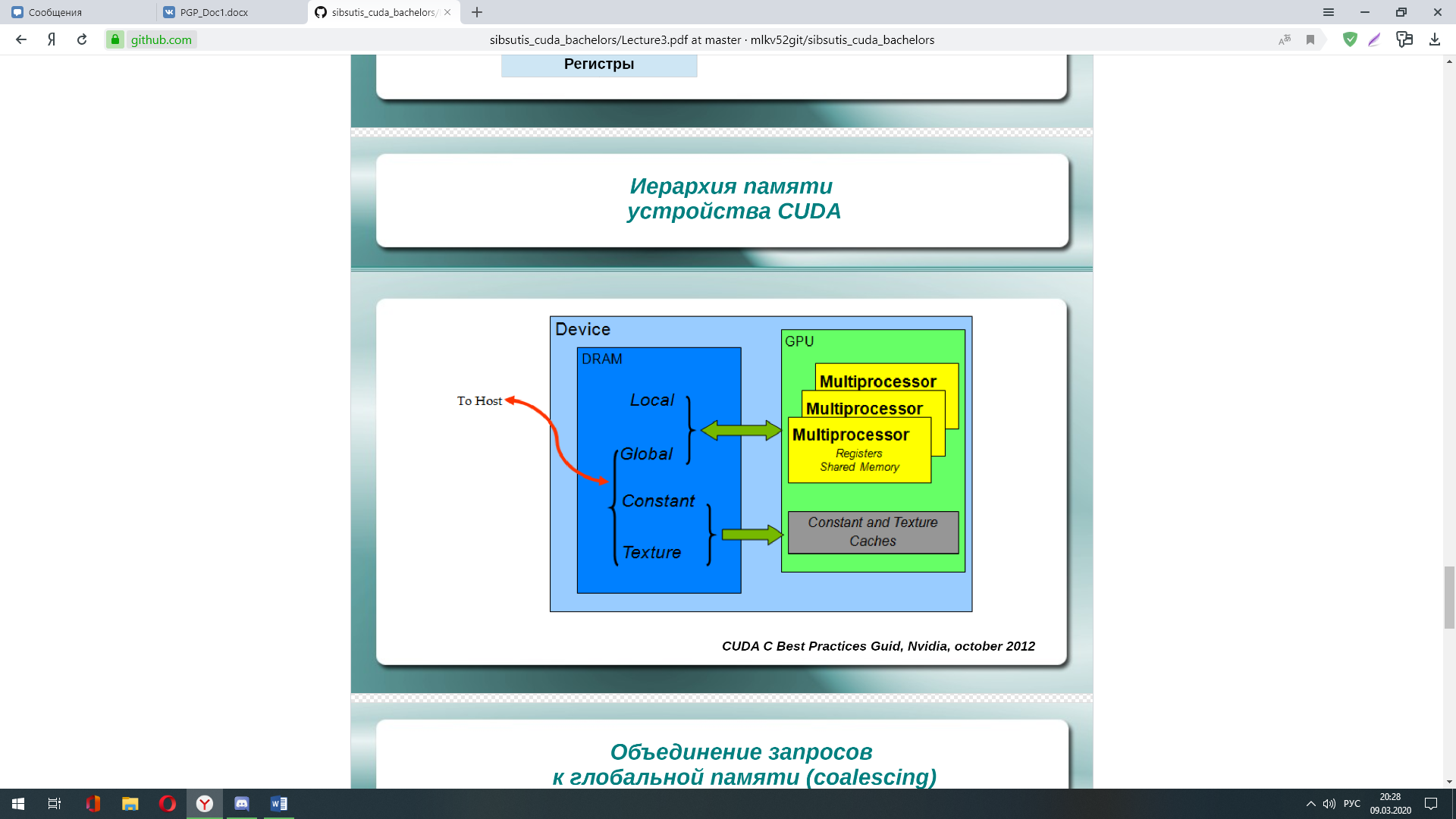
освобождение памяти по указателю devPtr

cudaError\_t cudaMemcpy ( void \* dst, const void \* src, size\_t size, enum cudaMemcpyKind kind );

копирование size байт памяти в направлении kind, которое может принимать следующие значения:

* cudaMemcpyHostToHost
* сudaMemcpyHostToDevice
* cudaMemcpyDeviceToHost
* cudaMemcpyDeviceToDevice
* cudaMemcpyDefault

# **Иерархия памяти GPU**



# **Глобальная память GPU. Шаблон использования, оптимизация доступа, метрики производительности работы с глобальной памятью (в т.ч. метрики профилировщика)**

Глобальная память не кэшируется. Она работает очень медленно, количество обращений к глобальной памяти следует минимизировать.

В CUDA имеется возможность произвольной адресации глобальной памяти. То есть можно читать из любой ячейки памяти, и писать можно тоже в произвольную ячейку.

Глобальная память необходима в основном для сохранения результатов работы программы перед отправкой их на хост (в обычную память DRAM). Причина этого в том, что глобальная память — единственный вид памяти, куда можно что-то записывать.

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. Отсюда следует, что распределением памяти должна заниматься программа-хост, работающая на CPU. Данные с хоста можно отправлять в устройство вызовом функции cudaMemcpy:

cudaMemcpy(void\* gpu\_mem, void\* cpu\_mem, int size, cudaMemcpyHostToDevice);

Точно таким же образом можно проделать и обратную процедуру:

cudaMemcpy(void\* cpu\_mem, void\* gpu\_mem, int size, cudaMemcpyDeviceToHost);

Этот вызов тоже осуществляется с хоста.

Запросы на чтение и запись к глобальной памяти нитями одного варпа (a warp) объединяются в транзакции, количество которых равно количеству необходимых для выполнения запросов блоков данных (cache lines) L1 кэша размером в 128 байт.

# **Разделяемая память GPU. Шаблон использования, оптимизация доступа, метрики производительности работы с разделяемой памятью (в т.ч. метрики профилировщика)**

**Работа с разделяемой памятью**

Общая память для всех потоков одного блока. содержит переменные, объявленные в ядре с квалификатором \_\_shared\_\_ (для каждого блока будет создан один экземпляр таких переменных), и аргументы ядер.

Обычно используется для взаимодействия потоков одного блока, в этом случае нужна синхронизация \_\_syncthreads().

Латентность 4 такта.

Малый размер (от 16 или 48kB).

Типичная схема использования разделяемой памяти для уменьшения времени доступа к глобальной памяти:

* загрузка интенсивно используемых данных из глобальной памяти;
* синхронизация (при необходимости);
* вычисления с использованием загруженных данных;
* синхронизация (при необходимости);
* запись результатов в глобальную память.

**Эффективная работа с разделяемой памятью**

Разделяемая память разбита на банки (страницы) таким образом, что последовательные 32-битные слова попадают в последовательные банки.

Каждый банк работает независимо от других, возможен параллельный доступ к различным банкам.

Доступ нескольких потоков к одному банку происходит конфликт банков и доступ сериализуется (выполняется отчасти последовательно для исключения конфликтов), исключение – чтение всеми потоками данных из одного и того же банка.

**Разделяемая память. Банк конфликты**

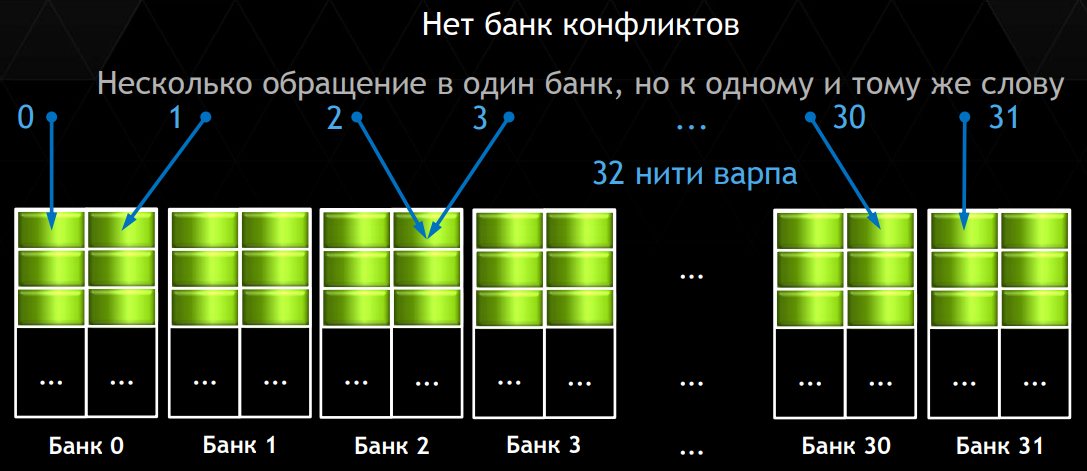
Банк конфликты возникают, когда:

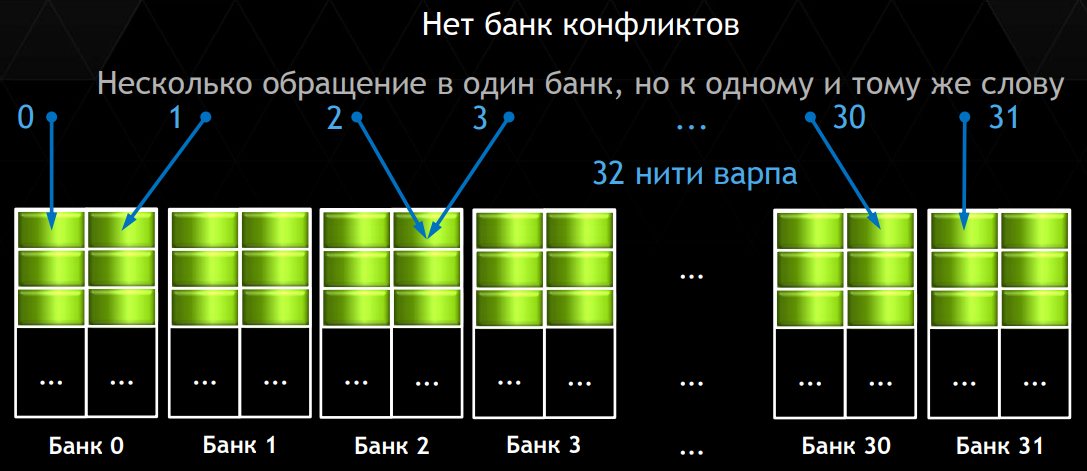
* две или более нитей одного варпа обращаются к разным 8-Байтовым словам, лежащим в одном банке
* банк-конфликт имеет порядок N когда конфликтуют N нитей одного варпа

Банк конфликтов нет, когда:

* разные нити варпа обращаются к одному слову
* разные нити варпа обращаются к различным байтам одного и того же слова









# **Теоретическая и достигнутая заполняемость. Ограничивающие заполняемость факторы.**