Введение в параллельные вычисления

д.т.н. Мокрова Наталия Владиславовна

асп. Морунов Егор, Сырко Денис

пятница	ауд. 118, 119	
12:50 – 14:20	Лекция	
	1 неделя	2 неделя
14:35 - 17:55	Лаб – КС44	Лаб - КС40

Выбор модели программирования

Системы с общей памятью

- C++11 threads (pthreads)
- OpenMP

Системы с распределенной памятью

- MPI = message passing interface

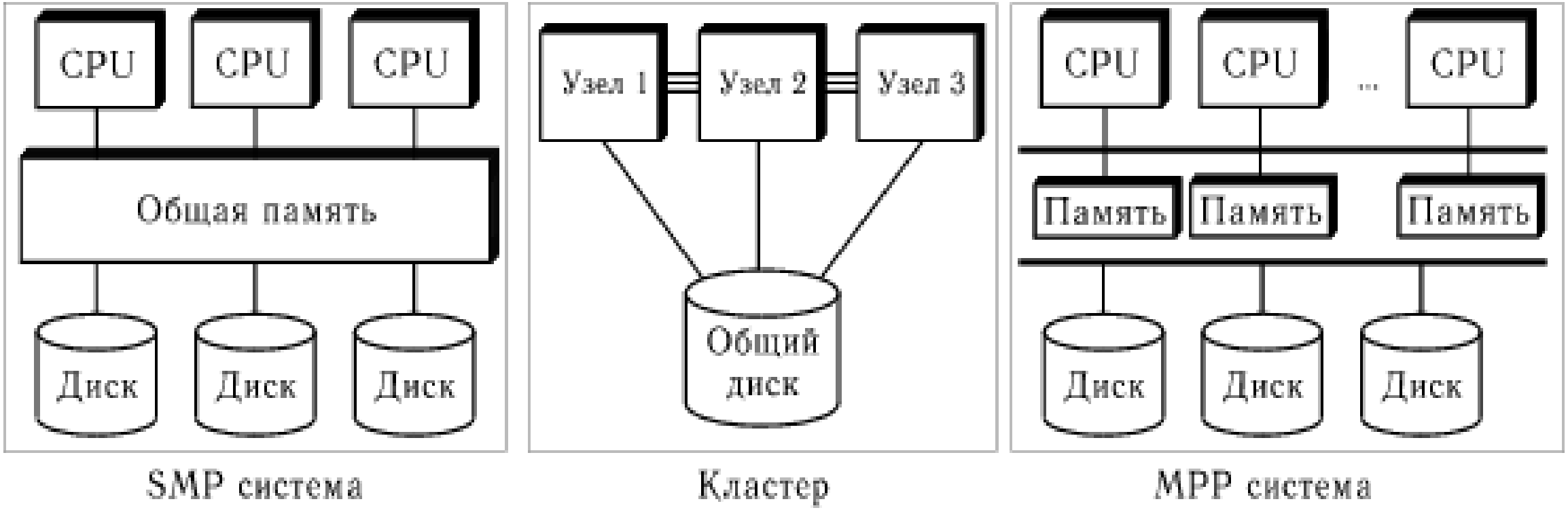
Гетерогенные системы (графические процессоры)

- CUDA

Лекция 15. CUDA
Основные понятия

- А. В. Боресков и др. Предисл.: В. А. Садовничий. **Параллельные вычисления на GPU. Архитектура и программная модель CUDA: Учебное пособие.** Издательство Московского университета, 2012.
- Перепёлкин Е.Е., Садовников Б.И., Иноземцева Н.Г. **Вычисления на графических процессорах (GPU) в задачах математической и теоретической физики**. Издательство: URSS, 2014/
- Джейсон Сандерс, Эдвард Кэндрот. **Технология CUDA в примерах. Введение в программирование графических процессоров.** ДМК Пресс, 2011.
- А. В. Боресков, А. А. Харламов. **Основы работы с технологией CUDA.** 2010.
- Параллельные вычисления CUDA | Что такое CUDA? | NVIDIA www.nvidia.ru › NVIDIA › Технологии › Вычисления на GPU.

Схемы параллельных вычислительных систем



Классификация параллельных вычислительных систем по способу доступа к памяти:

- MPP – системы распределенной памяти;
- SMP – симметричные мультипроцессорные системы общей памяти;
- Кластеры – экономичный вариант MPP.

Архитектура ускорителя Kepler

Модель Geforce GTX 680 основана на первом графическом процессоре компании Nvidia, имеющем новейшую архитектуру Kepler.

PU имеет в своём составе несколько блоков GPC (кластеры графической обработки — Graphics Processing Clusters), которые являются независимыми устройствами в составе видеочипа, способными работать сами как отдельные устройства, так как в их составе есть все необходимые собственные ресурсы: растеризаторы, геометрические движки и текстурные модули. Большинство функционала выполняется внутри блоков GPC.

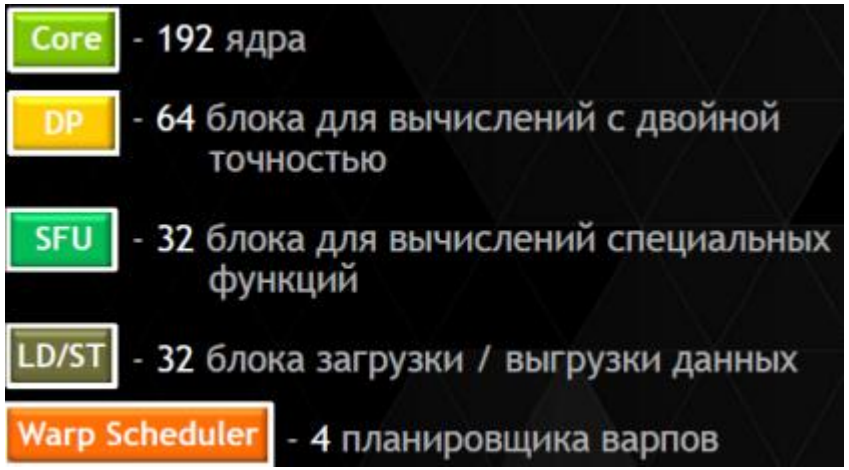
Блок-схема GK10:

http://www.ixbt.com/video3/gk104-part1.shtml

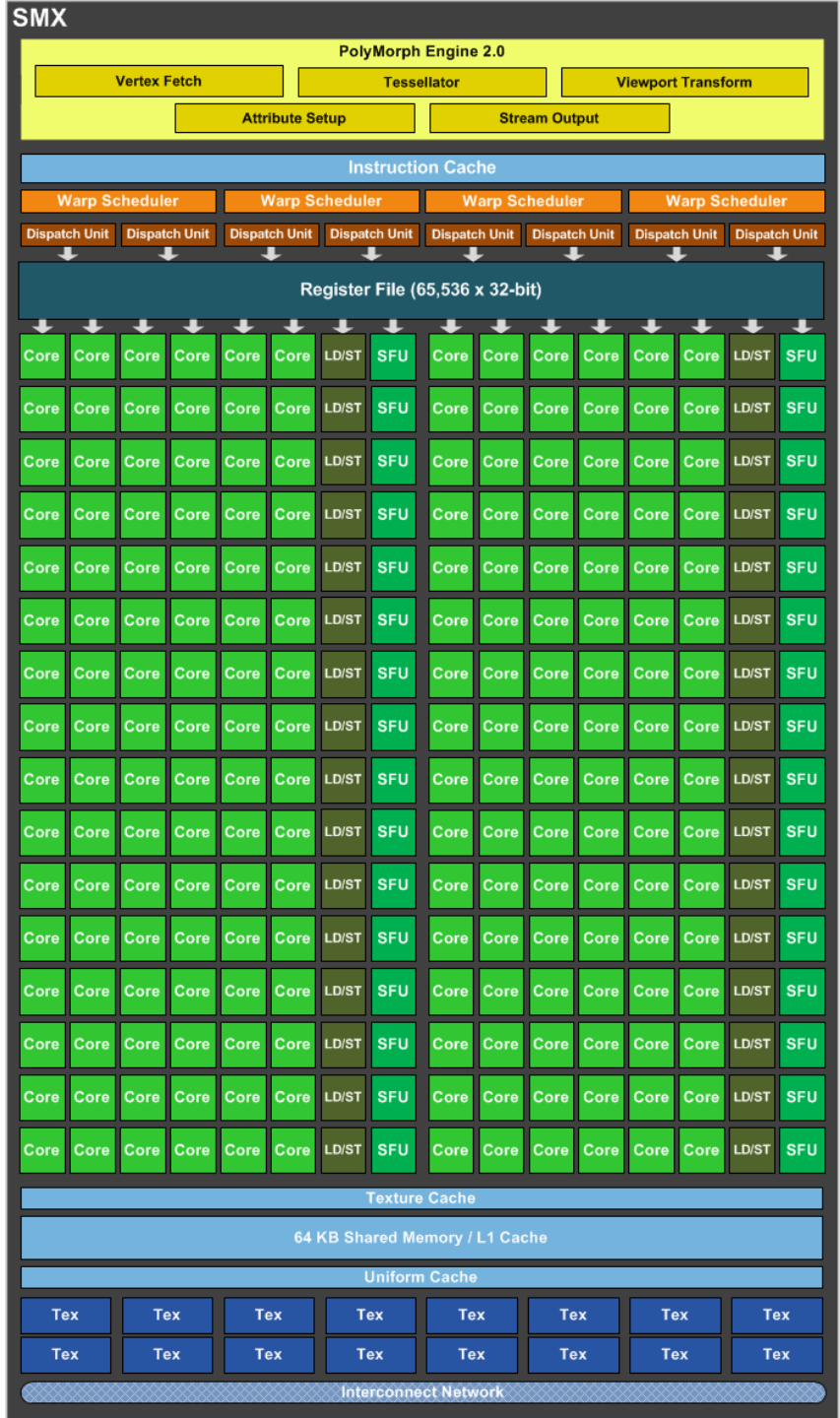
Структура SMX

Новый GPU имеет четыре блока GPC, как и предыдущий топовый чип GF100/GF110, но в отличие от них, каждый из этих блоков содержит по два потоковых мультипроцессора, отличающихся от предыдущих чипов Nvidia.

Новое решение использует следующее поколение потоковых мультипроцессоров (Streaming Multiprocessor – SMX), в отличие от SM в предыдущих чипах.



Важнейшие изменения произошли в SMX.



Изменения в SMX

- В SMX более высокая производительность, что видно по количеству функциональных устройств, но при этом потребляют значительно меньше энергии. А уменьшенное количество мультипроцессоров на GPU (8 в отличие от 16 в GF100/GF110) было продиктовано установленными рамками по площади ядра.
- Большая часть ключевых блоков GPU включена в состав SMX: потоковые процессоры (CUDA Cores) выполняют все математические операции над пикселями, вершинами и занимаются неграфическими вычислениями, текстурные модули (TMU) фильтруют текстурные данные, загружают и записывают их из/в видеопамять, блоки специальных функций (Special Function Units, SFU) выполняют сложные операции (вычисление синуса, косинуса, квадратного корня и т.п.) и интерполяции графических атрибутов. Ну а движок PolyMorph обеспечивает выборку вершин, занимается тесселяцией, преобразованием в экранные координаты, установкой атрибутов и потоковым выводом (stream output).
- Количество блоков загрузки-сохранения (Load-Store Unit – LSU) в GK104 в расчёте на каждые шесть потоковых блоков снизилось. Блоки LSU используются для передачи данных из/в кэш и разделяемую память, что негативно скажется на задачах GPU вычислений. Уменьшение количества LSU не должно повлиять на производительность в графических применениях.

CUDA

CUDA – архитектура параллельных вычислений от NVIDIA, позволяющая существенно увеличить вычислительную производительность благодаря использованию GPU (графических процессоров).

CUDA SDK позволяет программистам реализовывать на специальном упрощённом диалекте языка программирования Си алгоритмы, выполнимые на графических процессорах NVIDIA, и включать специальные функции в текст программы на Си.

Архитектура CUDA даёт разработчику возможность по своему усмотрению организовывать доступ к набору инструкций графического ускорителя и управлять его памятью.

История развития GPU-вычислений

- 80-90 г. Рост популярности графических ОС типа MS Windows, ускорители двумерной графики для ПК позволяли аппаратно выполнять операции с растровыми изображениями.
- 92 г. Silicon Graphics выпустила библиотеку OpenGL, как платформенно-независимым метод написания трехмерных графических приложений.
- Середина 90 г. Выход в рынок первых FPS (Doom, Duke Nukem 3D, Quake), создание реалистичных 3D-сцен. Компании NVIDIA, ATI Technologies и 3dfx Interactive начали выпуск доступных по цене графических ускорителей.
- Выпуск GeForce 256 – впервые вычисления геометрических преобразований и освещения сцены возможно производить непосредственно в графическом процессоре.
- Выпуск GeForce 3 – первой микросхемы, в которой был реализован стандарт Microsoft DirectX 8.0, который требовал, чтобы совместимое оборудование включало возможность программируемой обработки вершин и пикселов. Частичный контроль над вычислениями на GPU.
- Ноябрь 2006 г. – NVIDIA выпустила первую GPU с поддержкой стандарта DirectX 10 – GeForce 8800 GTX, построенный (GPU) на архитектуре CUDA.

Или немного об истории появления

Лекция:
Краснов Дмитрий
Маркин Иван
Пысин Максим

Особенности раннего программирования на GPU

- Единственный способ взаимодействия с GPU – через графические API типа OpenGL и DirectX. Задачи общего характера приходилось представлять как задачу рендеринга.
- GPU вычисляли цвет каждого пиксела экрана с помощью программируемых арифметических устройств, пиксельных шейдеров, которые в общем случае на вход получали координаты (x, y) точки на экране и некоторую дополнительную информацию, а на выходе выдавали конечный цвет этой точки.
- Вычисления записывались на специальных языках графического программирования – шейдерных языках. Программист «подсовывал» на вход шейдеру свои числовые данные вместо графических.
- Ограничения:
  - на ресурсы – программа могла получать входные данные только в виде горстки цветов и текстурных блоков;
  - невозможность записи в произвольные ячейки памяти;
  - разные GPU по-разному работали с числами с плавающей точкой (а то и вообще не работали);
  - …

Особенности архитектуры CUDA

- Наличие унифицированного шейдерного конвейера, позволяющего программе задействовать любое АЛУ, входящее в микросхему.
- Набор команд, ориентированный на вычисления общего назначения, а не только на графику
- Разрешение произвольного доступа к памяти для чтения и записи исполняющим устройствам GPU, а также доступ к программно-управляющему кэшу – разделяемой памяти.
- Вместо программирования на шейдерных языках типа GLSL/HLSL через OpenGL/DirectX применяется расширенный язык C – CUDA C.

Применения:

- Обработка медицинских изображений.
- Вычислительная гидродинамика.
- Науки об окружающей среде (климат).

Средства для разработки на CUDA

Необходимые средства для разработки на CUDA C:

- GPU, поддерживающий CUDA.
- Драйвер устройства NVIDIA.
- Комплект средств разработки CUDA (CUDA Toolkit).
- Стандартный компилятор языка C.

Все можно загрузить по адресу: http://developer.nvidia.com/cuda/cuda-downloads

Документация по программированию:

http://developer.nvidia.com/cuda/nvidia-gpu-computing-documentation

Вебинары: http://developer.nvidia.com/cuda/get-started-cuda-cc

Нужно два компилятора – один компилирует код на CUDA C для CPU, другой для GPU. В состав Toolkit входит компилятор CUDA C, компилирующий код для GPU.

Настройка VS проекта, варианты:

- Взять проект template из комплекта SDK.
- NVIDIA Parallel Nsight, после установки станет доступен проект CUDA.

Ключевые понятия

Ядро CUDA-программы – обычная Си функция использующая особый способ вызова языка CUDA C.

- ядро, как функция языка си, может принимать параметры;
- из ядра нельзя получить какие либо данные простой передачей указателя на память выделенную на CPU;
- так же, в ядро нельзя передавать массивы, и указатели на них, если они находятся в памяти CPU;
- ядро может вызывать любые другие функции, кроме другого ядра, невозможна рекурсия ядра, но возможна рекурсия функции;
- ядро создает локальные копии переменных объявляемых внутри него для каждой нити.

host – центральный процессор;

device – графический ускоритель;

block (блок) – параллельно выполняемый экземпляр ядра;

grid (сетка) – группа параллельных блоков (одномерная, двухмерная, трехмерная);

thread (нить) – при программировании под CPU называется потоком выполнения;

warp – объединение 32 нитей в пучок.

Hello world

```
#include <stdio.h>
__global__ void kernel() {
}
int main() {
      kernel<<<1,1>>>();
      printf("Hello, World!\n");
      return 0;
}
```

О ключевых словах

Ключевое слово __device__ означает, что код исполняется GPU. Такие функции можно вызывать из других функций с ключевыми словами __device__ и __global__.
Функции __global__ вызываются только из кода CPU.

В одном файле находится код предназначенный для исполнения как CPU (host), так и GPU (device).
Функция kernel передается компилятору, обрабатывающему код для устройства.
<<<x,y>>> – вызов кода (с созданием x параллельных блоков и y потоков), выполняемого GPU.

Работа с памятью на GPU

```
__global__ void add(int a, int b, int *c) {
        *c = a + b;
}
int main() {
        int c; int *dev_c;
        cudaMalloc( (void**)&dev_c, sizeof(int) );
        add<<<1,1>>>(2, 7, dev_c);
        cudaMemcpy(&c, dev_c, sizeof(int), cudaMemcpyDeviceToHost);
        printf("2 + 7 = %d\n", c);
        cudaFree(dev_c);
        return 0;
}
```

В программе:

- Передаются параметры ядру.
- Выделяется память, чтобы устройство через нее вернуло данные CPU.

cudaMalloc() выделяет память на GPU. Первый аргумент – указатель на указатель, в котором будет возвращен адрес выделенной области памяти, второй – размер этой памяти.

Правила работы с указателями на память

- Разрешается передавать указатели на память, выделенную cudaMalloc() функциям, исполняемым GPU.
- Разрешается использовать указатели на память, выделенную cudaMalloc() для чтения и записи в эту память в коде, исполняемом GPU.
- Разрешается передавать указатели на память, выделенную cudaMalloc() функциям, исполняемым CPU.
- Не разрешается использовать указатели на память, выделенную cudaMalloc(), для чтения и записи в эту память в коде, который исполняется CPU.

Поэтому область памяти GPU копируется функцией cudaMemcpy() в память CPU. Последний параметр – константа, описывающая направление копирования памяти:

cudaMemcpyDeviceToHost

cudaMemcpyHostToDevice

cudaMemcpyDeviceToDevice

Функции работы с памятью GPU возвращают коды ошибок.

Получение информации об устройствах

```
int main() {
        cudaDeviceProp prop;
        int count;
        cudaGetDeviceCount(&count);
        for(int i = 0; i < count; i++) {
                cudaGetDeviceProperties(&prop, i);
                // Сделать что-то со свойствами устройства
}
```

Структура cudaDeviceProp содержит поля с информацией:

- имя GPU,
- объём памяти,
- максимальное количество нитей в блоках и т.д.

Измерение производительности

Событие CUDA – временная метка GPU, запомненная пользователем в определенный момент времени. События позволяют измерять производительность.

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

    cudaEventRecord( start, 0 );
    // Выполнение ядра, работа с памятью GPU
    cudaEventRecord( stop, 0 );
    cudaEventSynchronize( stop );

float elapsedTime;
cudaEventElapsedTime( &elapsedTime, start, stop );
printf( "Time to generate: %3.1f ms\n", elapsedTime );

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

Причина использования cudaEventSynchronize() – из-за асинхронности некоторых обращений к исполняющей среде CUDA, например, при запуске ядра GPU начинает исполнять код ядра, но CPU продолжает выполнять дальше свой код, не дожидаясь окончания работы GPU. После возвращения управления функцией cudaEventSynchronize() есть уверенность, что вся работа до окончания события завершена.

Сложение векторов

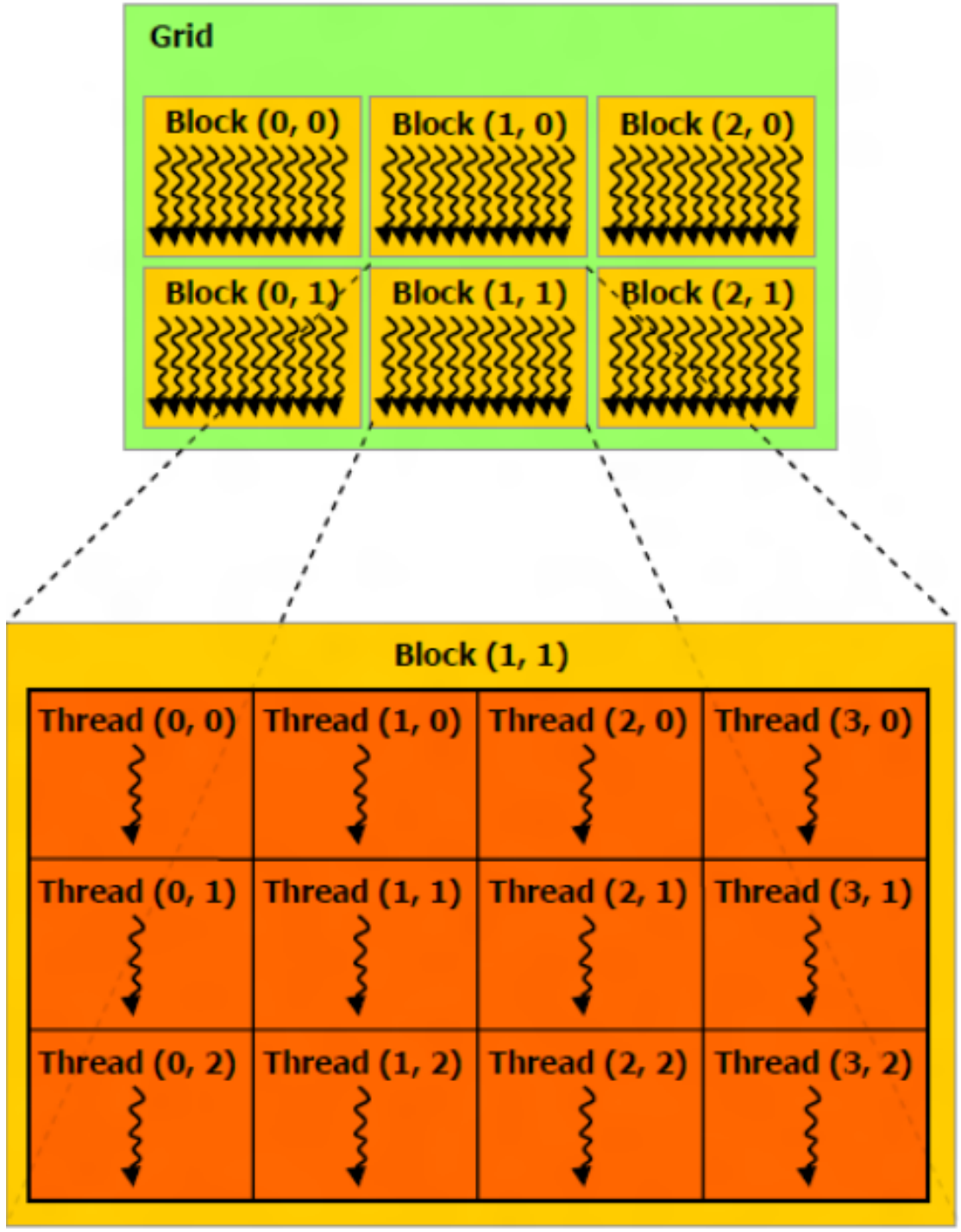
```
__global__ void add(int *a, int *b, int *c) {
        int i = threadIdx.x;
                if(i < N)
                        c[i] = a[i] + b[i];
}
int main() {
        int a[N], b[N], c[N];
        int *dev_a, *dev_b, *dev_c;
// Инициализация a, b, выделение памяти GPU, копирование a и b в память GPU
        // складываем вектора
        add<<<1,N>>>( dev_a, dev_b, dev_c );
        // Копирование dev_c в c, вывод результатов
}
```

Число N в записи <<<1,N>>> определяет число нитей в пределах одного блока. В функции add каждая нить узнает свой индекс (координату x трехмерной решетки (x, y, z)), на основе координаты вычисляет соответствующий элемент вектора.

Блоки и нити

При вызове __global__ функции в конструкции
<<< >>> в общем случае указываются два трехмерных вектора типа dim3, обозначающие размерность решетки блоков и размерности блока.

Пример для решетки блоков размерностью (2, 3, 1) и размерности блока размерностью (3, 4, 1).

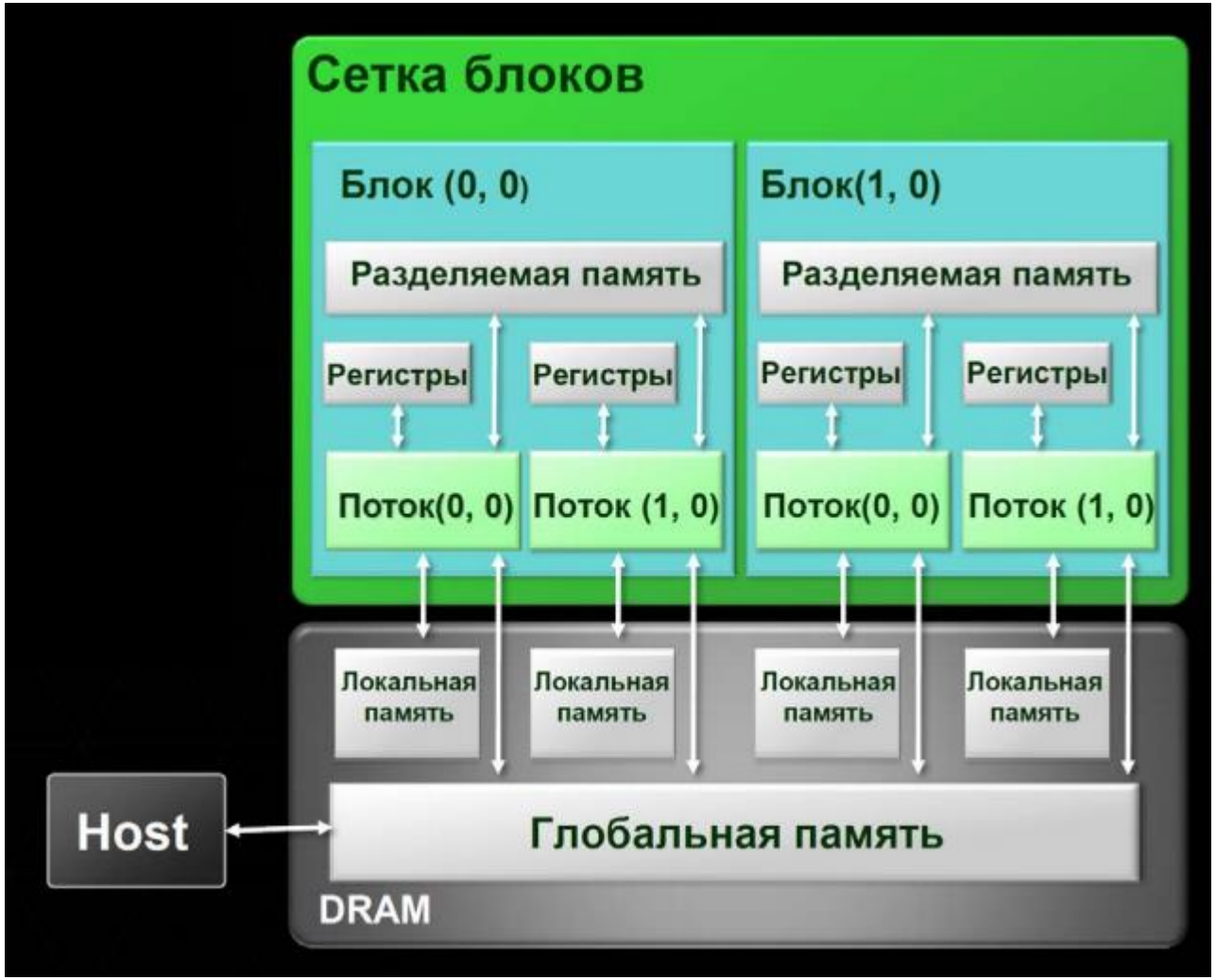


О размерности GPU

Каждая видеокарта имеет свои ограничения для максимальных размерностей решеток блока и размерности блока (512 x 512 x 64 для размерности блока и 65536 x 65536 x 1 для решетки блока).

Существует предел количества потоков на блок, поскольку все потоки блока располагаются на одном процессорном ядре GPU и должны разделять ограниченные ресурсы памяти этого ядра.

Размерность блока можно узнать с помощью встроенной переменной blockDim, размерность сетки блоков – с помощью gridDim.



Взаимодействие потоков

Если при объявлении переменной указать ключевое слово __shared__, то она будет размещена в разделяемой памяти.

Для каждой такой переменной создается копия в каждом блоке. Все нити, работающие в одном блоке, разделяют эту переменную, но не видят ее копии, размещенные в других блоках.

Буферы разделяемой памяти находятся на самом GPU, а не на DRAM, что уменьшает время доступа к ним, фактически они играют роль программно управляемого кэша.

```
__global__ void kernel() {
        …
__shared__ float cache[threadBerBlock];
// Действия с кэшем
__syncthreads(); // Синхронизация потоков
        …
}
```

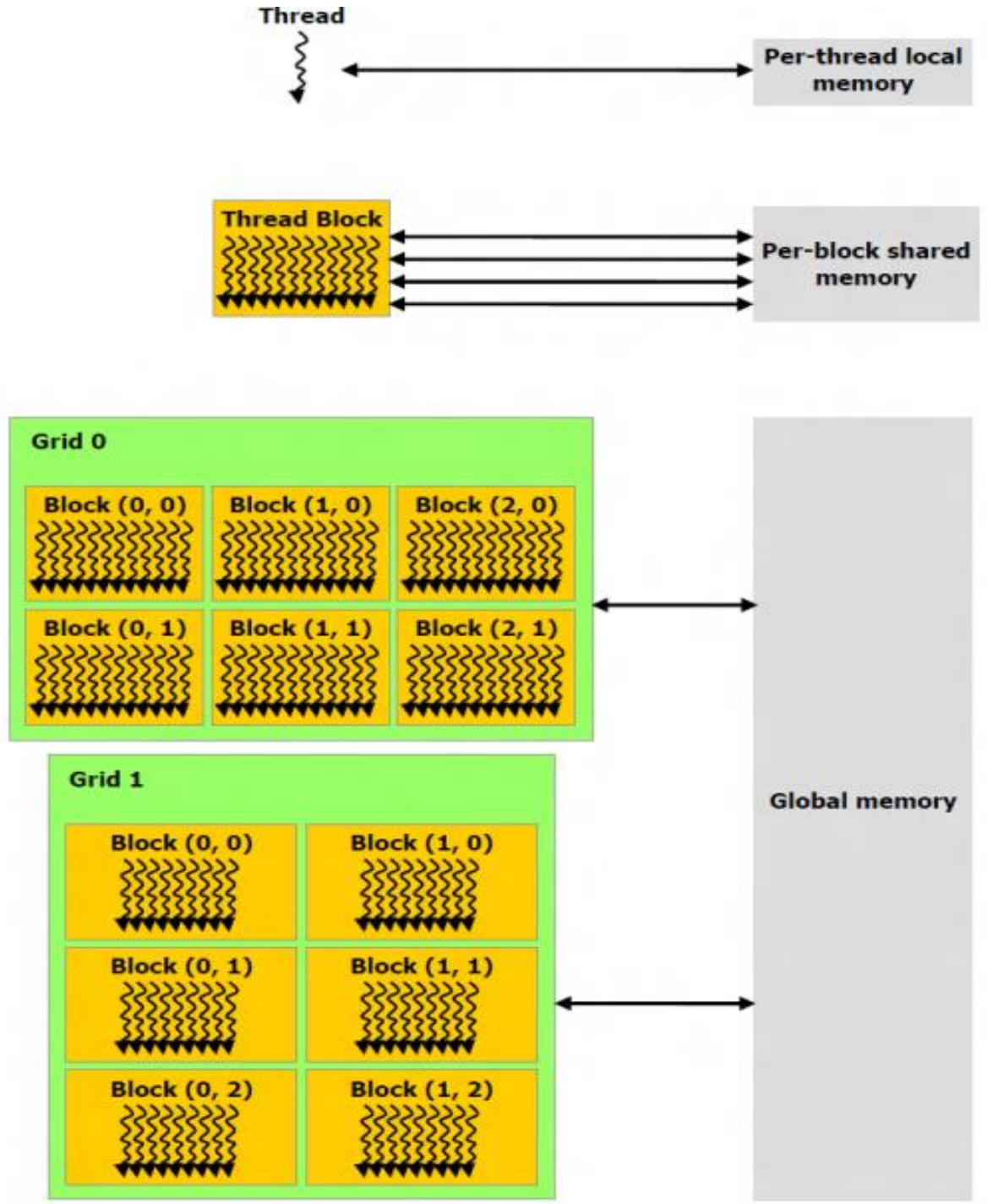
Функция __syncthreads() выполняет барьерную синхронизацию потоков. Предостережение: не стоит делать условный вызов __syncthreads().

Иерархия памяти

Каждая нить имеет собственную локальную память.

Каждый блок имеет разделяемую память, видимую для всех потоков этого блока. Все нити имеют доступ к глобальной памяти.

К глобальной памяти относится все, что в рассмотренных примерах выделялось с помощью функций cudaMalloc().



Константная память

В GPU очень много АЛУ, и узким местом вычисления на GPU часто становится скорость обмена с памятью. До этого мы рассматривали глобальную и разделяемую память, теперь рассмотрим константную память, которая служит для хранения данных, не изменяющихся в процессе исполнения ядра. Оборудование NVIDIA предоставляет 64 Кб константной памяти.

Копирование из памяти CPU в константную память. Размер копируемого массива должен быть известен на этапе компиляции!

```
__constant__ int arr[N];
…
cudaMemcpyToSymbol( arr, srcMemory, N * sizeof(int) );
```

Ключевое слово __constant__ помечает указатель на константную память.

Преимущества константной памяти

1. Одно чтение из константной памяти может быть передано (broadcast) другим «близким» нитям, что позволяет сэкономить до 15 операций чтения.
2. Константная память кэшируется (поскольку неизменна) в кэш константной памяти, поэтому последующие операции чтения из того же адреса не порождают дополнительного трафика.

«Близкие» нити, warp (канат, группа спряденных вместе нитей) – группа из 32 нитей, которые исполняются синхронно. Каждая нить, принадлежащая одному варпу, в любой момент времени исполняет одну и ту же команду над разными данными.

При чтении из константной памяти данные могут транслироваться на целый полуварп, т.е. на группу из 16 нитей – если каждая нить в полуварпе запрашивает данные из одного и того же адреса в константной памяти, то GPU выдает только один запрос на чтение, а затем передает данные каждому потоку.

Вследствие кэширования после прочтения данных по некоторому адресу, нити из других полуварпов, читающие тот же адрес, обнаружат данные в кэше, и повторного обращения к памяти не произойдет.

Текстурная память

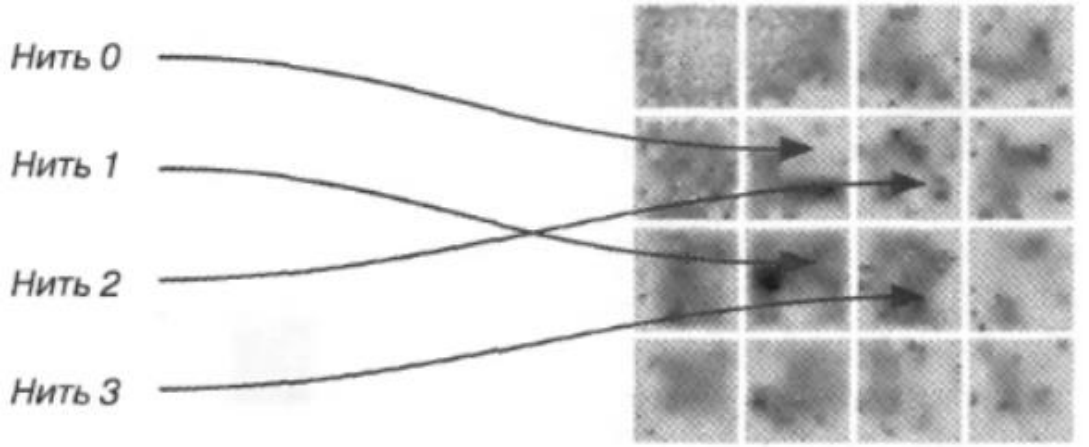
Текстурная память – еще один вид памяти, предназначенной только для чтения и позволяющей повысить производительность и сократить трафик между процессором и памятью при определенных способах доступа.

NVIDIA проектировала текстурные устройства для поддержки классических конвейеров рендеринга OpenGL и DirectX, у текстурной памяти есть некоторые свойства, делающие ее полезной для вычислений вообще.

Как и константная память, текстурная память кэшируется на кристалле, поэтому в некоторые случаях позволяет уменьшить количество обращений к внешнему DRAM.

Текстурные кэши предназначены для графических приложений, в которых доступ к памяти характеризуется высокой пространственной локальностью. В вычислительном приложении общего назначения это означает, что нить с большей вероятностью будет обращаться к адресам, расположенным "рядом" с адресами, к которым обращаются «близкие» нити.

С точки зрения арифметики, четыре показанных на рисунке адреса не являются соседними, поэтому не будут кэшироваться вместе при использовании стандартной схемы кэширования, применяемой в CPU. Но текстурные кэши GPU специально разработаны для ускорения доступа в таких ситуациях, поэтому применение текстурной памяти вместо глобальной может дать выигрыш.



CUDA память

Тип памяти	Тип доступа	Уровень выделения	Расположение	Скорость работы
Register (Регистровая)	RW (запись\чтение)	Per-thread (На нить)	On-chip (На чипе)	Самая высокая
Local (Локальная)	RW (запись\чтение)	Per-thread (На нить)	DRAM (В памяти GPU)	Низкая
Global (Глобальная)	RW (запись\чтение)	Per-grid (На сеть)	DRAM (В памяти GPU)	Самая низкая
Shared (Разделяемая)	RW (запись\чтение)	Per-block (На блок)	On-chip (На чипе)	Высокая (2 по скорости)
Constant (Константная)	RO (только чтение)	Per-grid (На сеть)	L1 cache (в кеше первого ур.)	Высокая (3 по скорости)
Texture (Текстурная)	RO (только чтение)	Per-grid (На сеть)	L1 cache (в кеше первого ур.)	Высокая (4 по скорости)

Сравнение кодов стандартного и параллельного языка C