

ПРОГРАММНАЯ МОДЕЛЬ CUDA. ПАРАЛЛЕЛЬНОЕ СЛОЖЕНИЕ ВЕКТОРОВ НА GPU СРЕДСТВАМИ CUDA.

АФАНАСЬЕВ ИЛЬЯ
AFANASIEV_ILYA@ICLOUD.COM

CUDA: ГИБРИДНОЕ ПРОГРАММИРОВАНИЕ

- **CUDA** (изначально аббр. от англ. Compute Unified Device Architecture) – программно-аппаратная архитектура параллельных вычислений, которая позволяет существенно увеличить вычислительную производительность благодаря использованию графических процессоров фирмы Nvidia
- Программа, использующая GPU, состоит из:
 - Кода для GPU, описывающего необходимые вычисления и работу с памятью устройства
- Кода для CPU, в котором осуществляется:
 - Управление памятью GPU – выделение / освобождение
 - Обмен данными между GPU/CPU
 - Запуск кода для GPU
 - Обработка результатов и прочий последовательный код

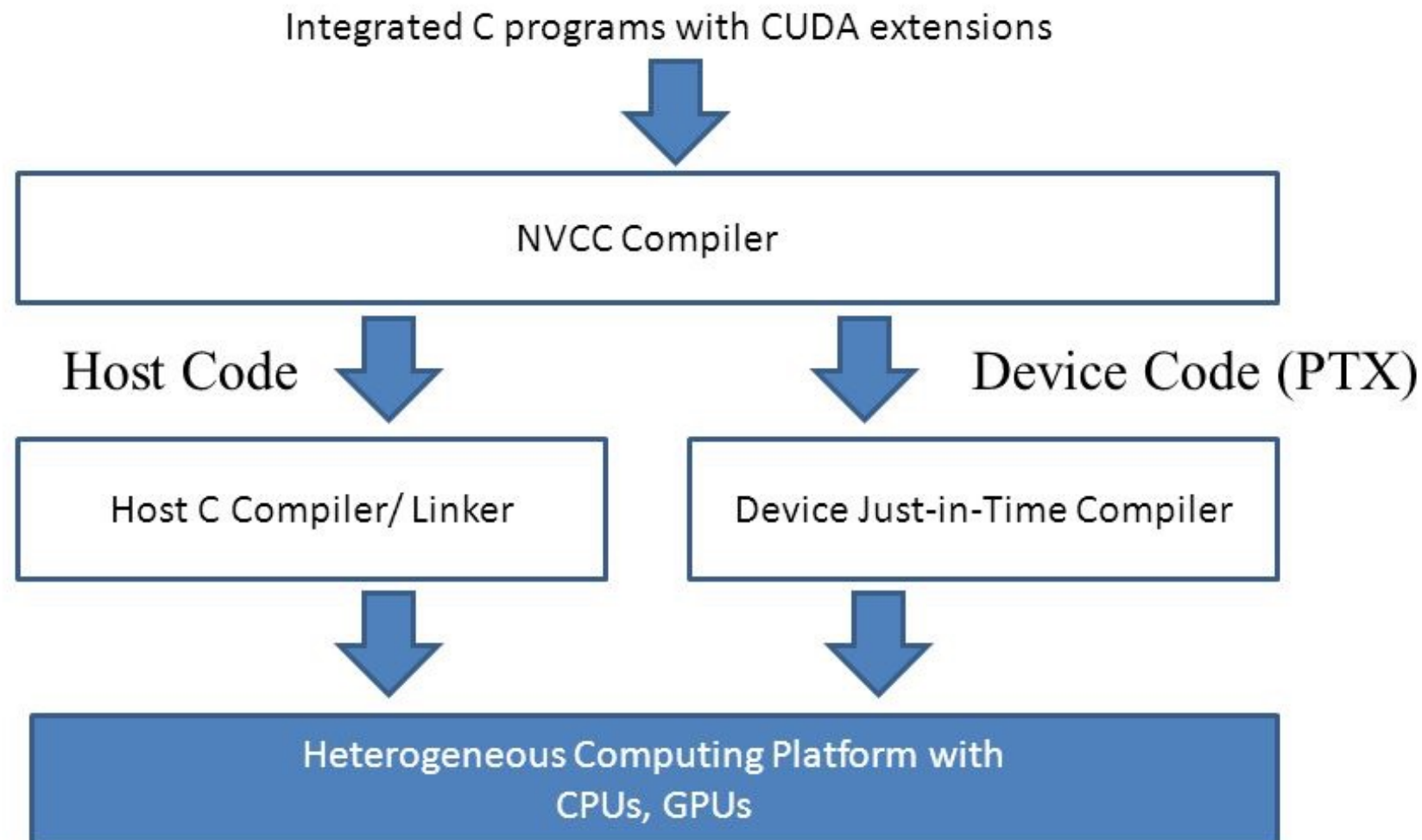
ТЕРМИНОЛОГИЯ (КОД)

- **CPU** будем далее называть «хостом» (от англ. host), код для CPU - код для хоста, «хост-код» (host-code)
- **GPU** будем далее называть «устройством» или «девайсом» (от англ. device), код для GPU – «код для устройства», «девайс-код» (device-code)
- Хост выполняет последовательный код (а при необходимости и параллельный), в котором содержатся вызовы функций, побочный эффект которых – манипуляции с устройством

CUDA PROGRAM

(HOST CODE + DEVICE CODE)

Compiling A CUDA Program



КОД ДЛЯ CPU (HOST-CODE)

- Код для **CPU** дополняется вызовами специальных функций для работы с устройством (выделение памяти, копирования данных, получение свойств GPU)
- Код для **CPU** компилируется обычным компилятором GCC/ICC
 - Кроме конструкции запуска ядра <<<...>>> !
- Возможна компиляция всего кода с использованием NVCC
- Функции для **GPU** линкуются из динамических библиотек

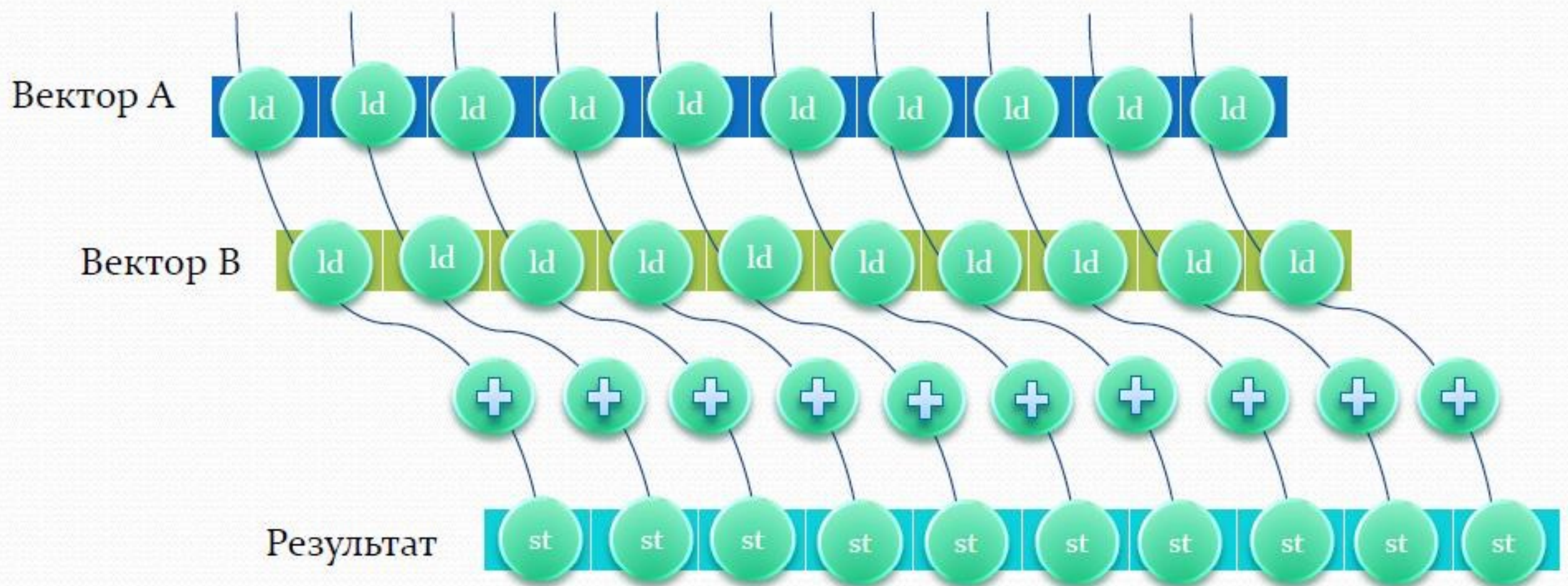
КОД ДЛЯ GPU (DEVICE-CODE)

- Код для GPU пишется на C++ с некоторыми расширениями:
 - Атрибуты функций, переменных и структур
 - Встроенные функции
 - Математика, реализованная на GPU
 - Синхронизации, коллективные операции
 - Векторные типы данных
 - Встроенные переменные:
threadIdx, blockIdx, gridDim, blockDim
 - Шаблоны для работы с текстурами
- GPU-код компилируется специальным компилятором NVCC (входящим в состав CUDA Toolkit)

ПРИМЕР: СЛОЖЕНИЕ ВЕКТОРОВ

CPU:

```
for (int i=0; i<N; ++i)  
    c[i] = a[i] + b[i];
```



ПРИМЕР: СЛОЖЕНИЕ ВЕКТОРОВ

Как будет выглядеть CUDA-программа, складывающая вектора?

Хост-код:

```
{  
    // Переслать данные с CPU на GPU  
    // Запустить вычисления на N GPU-нити  
    // Скопировать результат с GPU на CPU  
}
```

Девайс-код:

```
{  
    c[IDX] = a[IDX] + b[IDX]; // на каждой нити  
}
```

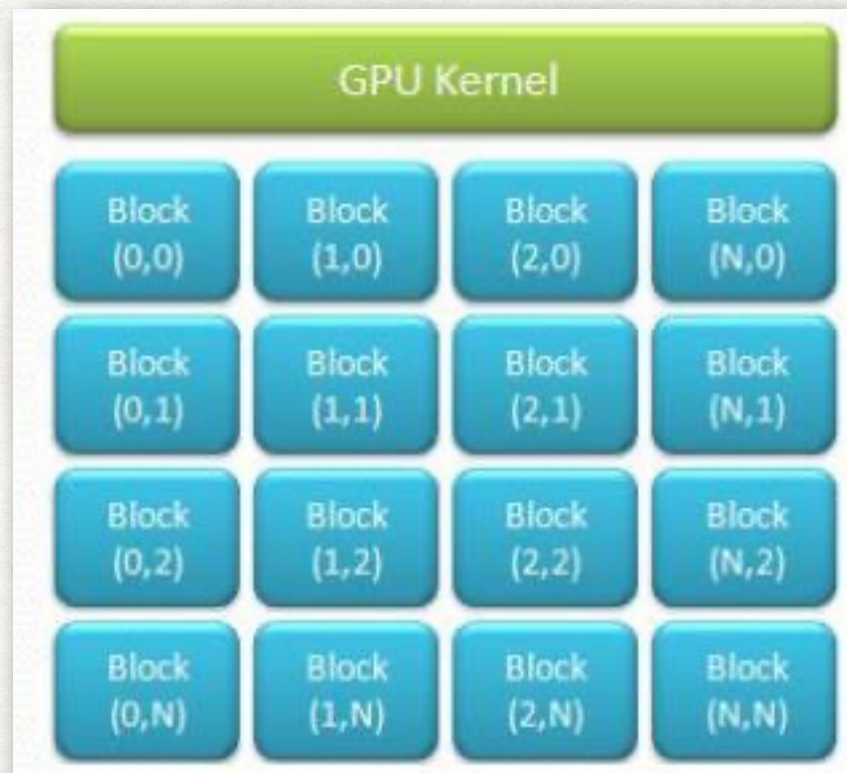

GPU: НИТИ, БЛОКИ, GRID

- Хост может запускать на GPU множества виртуальных нитей
 - Каждая из нитей использует для вычислений различные CUDA-ядра (в зависимости от типа выполняемой инструкции)
 - Нити группируются в виртуальные блоки (число блоков и размер блока задаёт пользователь, но с некоторыми аппаратными ограничениями, таким образом регулируя общее число нитей)
- Каждый потоковый мультипроцессор GPU (SM) обрабатывает несколько блоков
- Грид (от англ. Grid-сетка) – множество блоков одинакового размера, на которых запускается GPU-код
- Положение нити в блоке и блока в гриде индексируются по трём измерениям (x, y, z) - как у каждой нити, так и у каждого блока есть трехмерный индекс

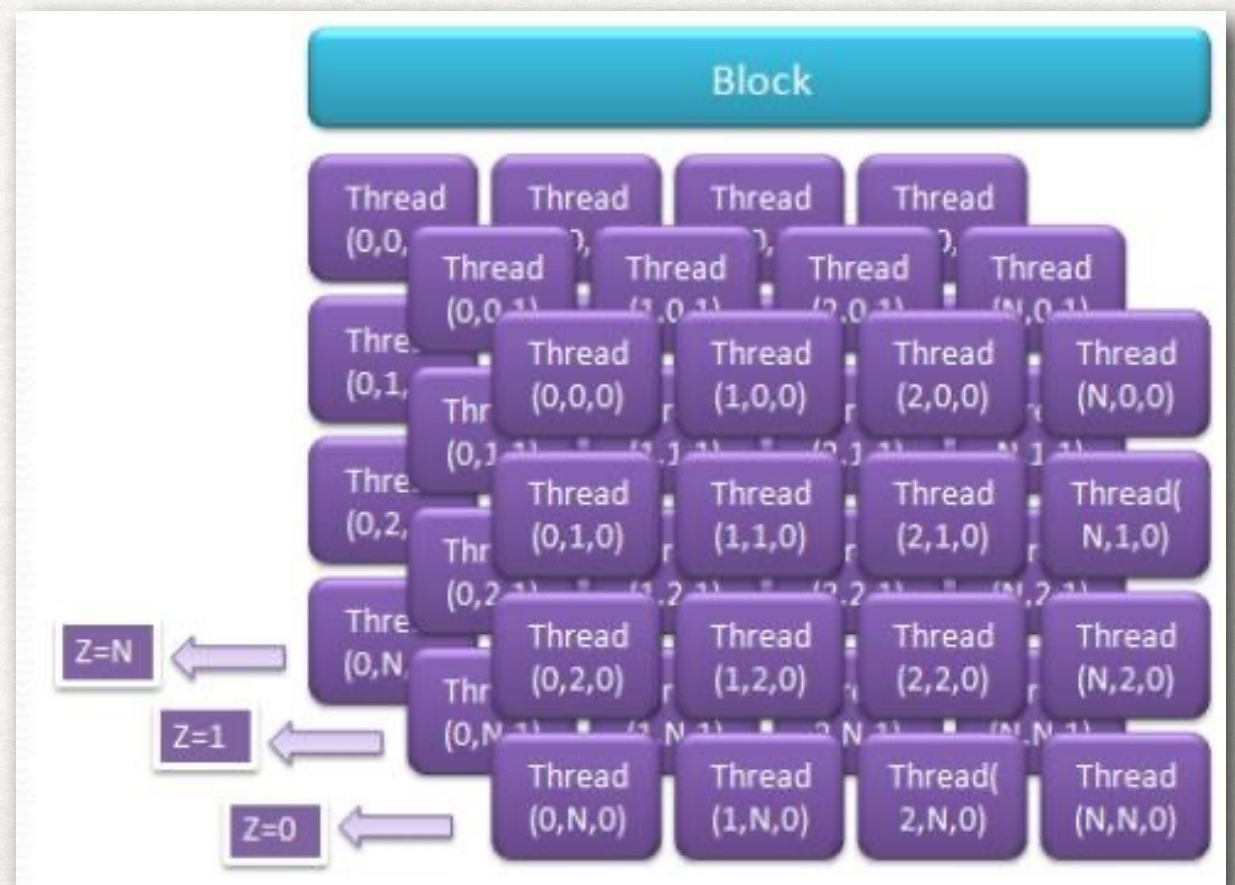
РАЗМЕРЫ GRID

- Грид задаётся количеством блоков по $[X, Y, Z]$ (размер грида в блоках) и размерами каждого блока по $[X, Y, Z]$
- Каждый из размеров может быть равен 1. Важные частые случаи - одномерный ($Z=1$) и двумерный грид ($Z=1, Y=1$), используемые для обработки векторов и матриц соответственно.

двумерный грид

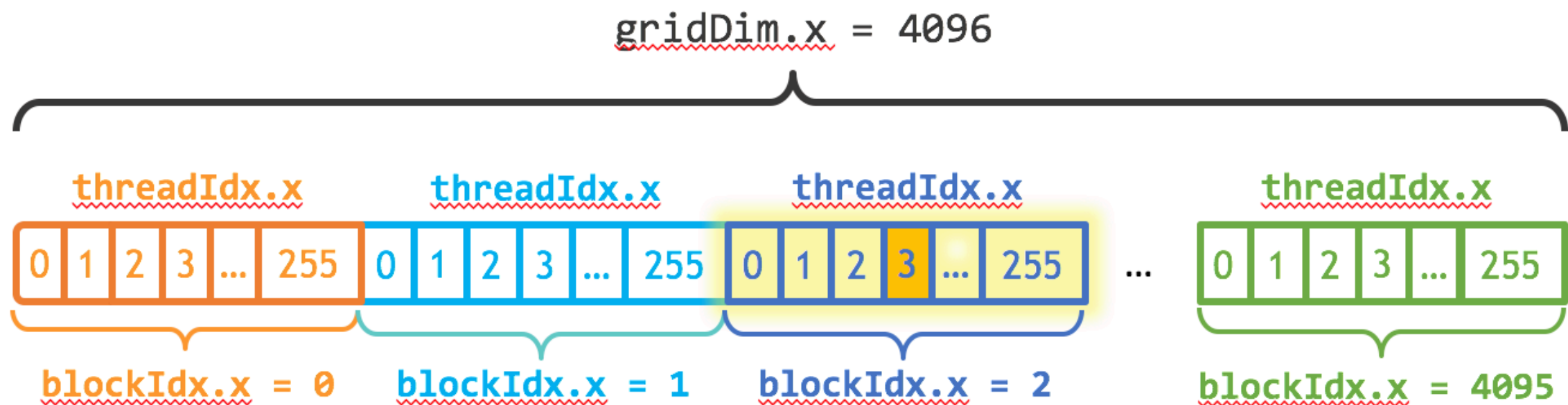


двумерный блок



ОДНОМЕРНЫЙ ГРИД

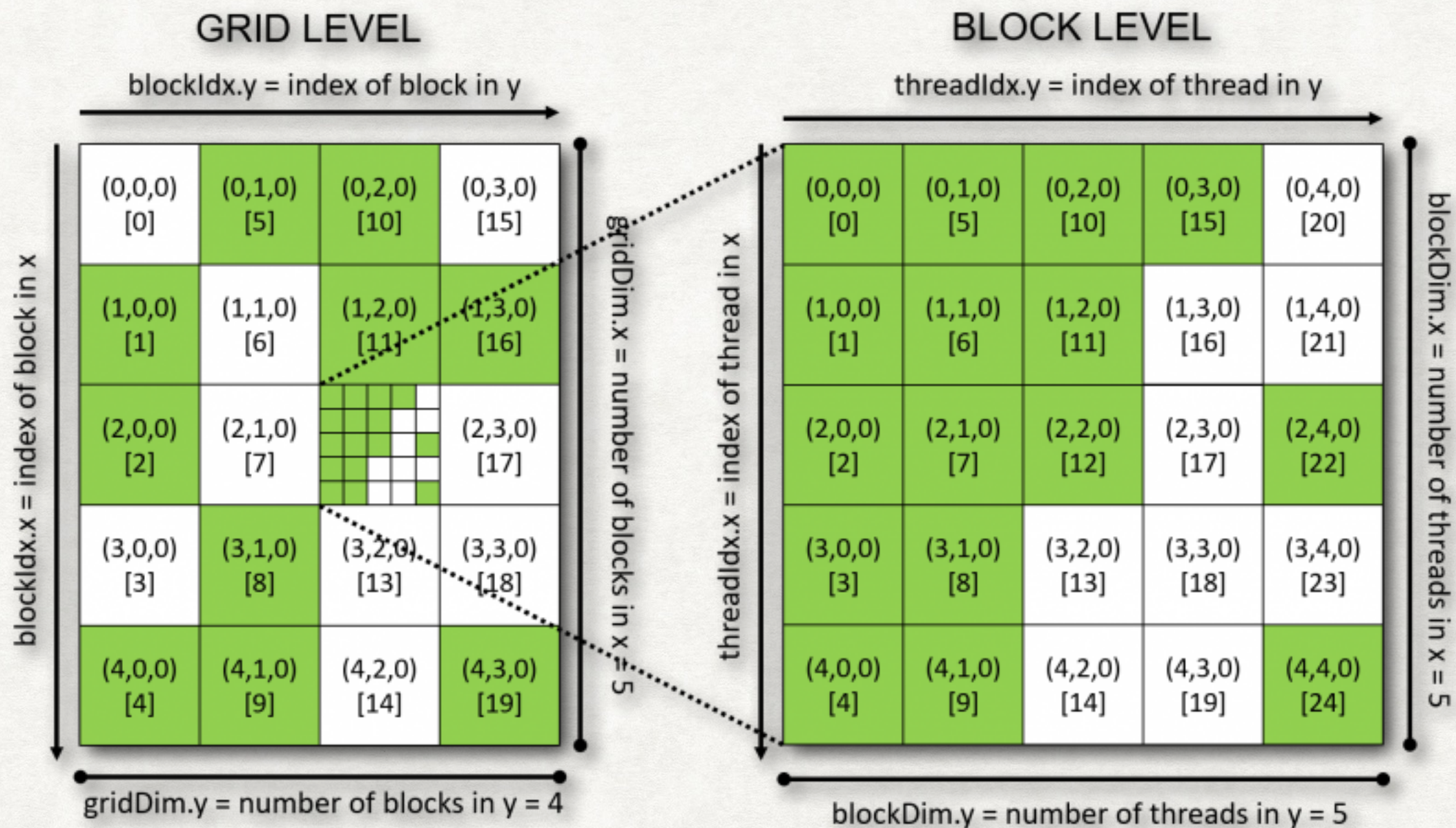
- 4096 блоков
- в каждом блок 256 нитей
- всего 1048576 нитей



$$\text{index} = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$$

$$\text{index} = (2) * (256) + (3) = 515$$

ДВУМЕРНЫЙ ГРИД И ДВУМЕРНЫЙ БЛОК



CUDA KERNEL («ЯДРО»)

- Поведение каждой из CUDA-нитей описывается специальными CUDA-ядрами
- Каждая нить выполняет копию специально оформленных функций «ядер», компилируемых для GPU
- В простейшем случае каждая из нитей выполняет одну и ту же версию кода над разными данными (data-driven параллелизм), однако в общем случае поведение нити может зависеть от её индекса
- У ядер нет возвращаемого значения (void)
- Перед описанием ядра должен присутствовать обязательный атрибут `__global__`
- CUDA-ядра описываются **только** в .cu файлах, компилируемых при помощи NVCC
- Пример:

```
__global__ void kernel (int param1, float *param2)  
{  
    // execute device code  
}
```


ТЕРМИНОЛОГИЯ (НИТИ И ЯДРА)

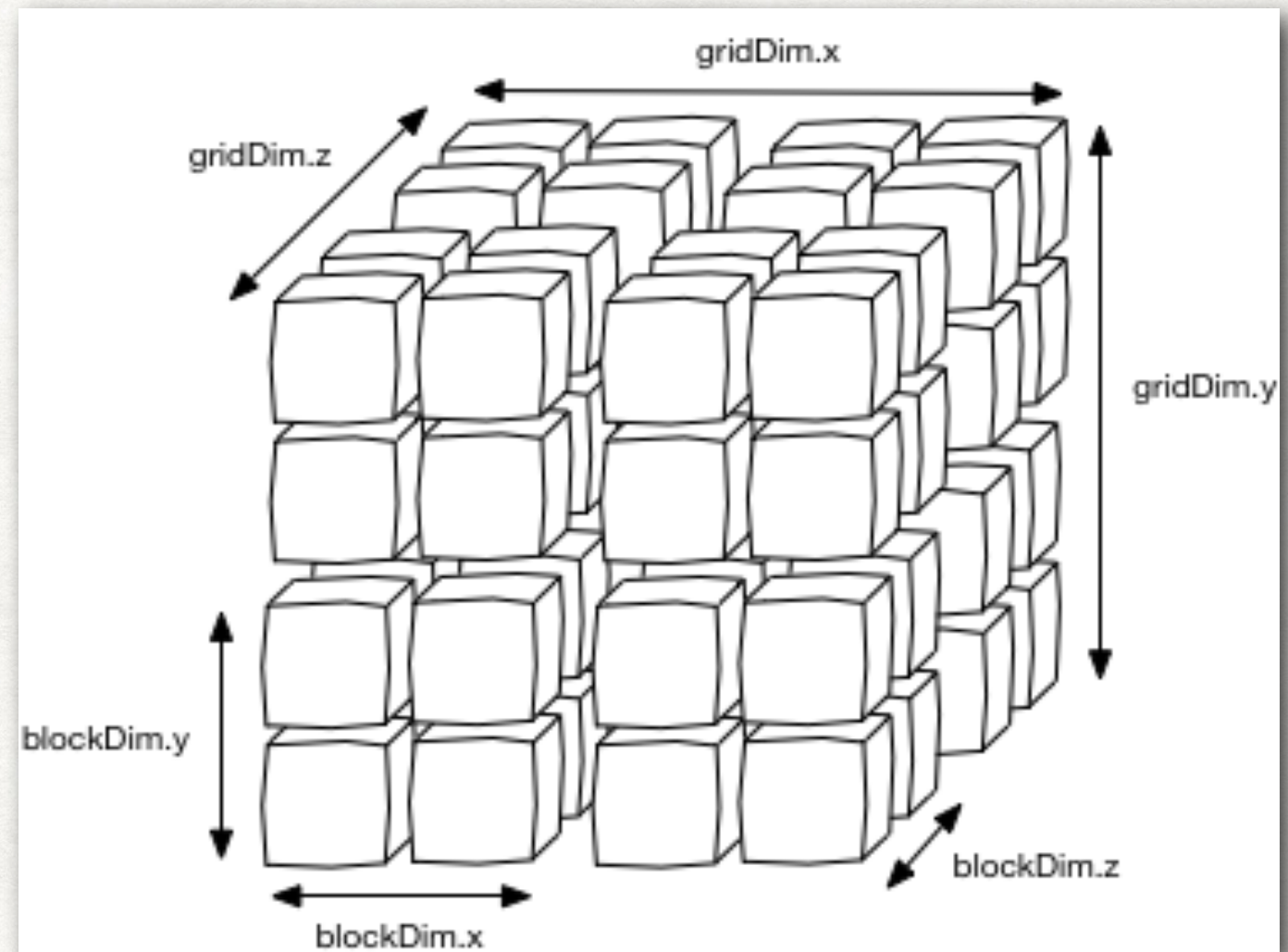
- **Хост** запускает вычисление ядра на **гриде** нитей (либо просто хост запускает ядро на GPU)
- Одно и то же ядро может быть запущено на разных **гридах** (с разной конфигурацией число блоков/размер блока)
- «Ядро» – программа, поток управления, определяет что делать
- «Грид» – определяет сколько делать

СИНТАКСИС ВЫЗОВА ЯДРА

- **kernel <<< execution configuration >>> (params);**
 - "kernel"-имя ядра
 - "params"-параметры ядра, копию которых получит каждая нить
 - execution configuration-Dg,Db,Ns,S
 - dim3 Dg - размеры грида в блоках, Dg.x * Dg.y * Dg.z число блоков
 - dim3 Db - размер каждого блока, Db.x * Db.y * Db.z - число нитей в блоке
 - size_t Ns - размер динамически выделяемой общей памяти (опционально)
 - cudaStream_t S - поток, в котором следует запустить ядро (опционально)
- struct dim3 - структура, определённая в CUDA Toolkit
 - Состоит из трех полей: unsigned x,y,z
 - Конструктор dim3(unsigned x=1, unsigned y=1, unsigned z=1)

КАК НИТЬ МОЖЕТ ОПРЕДЕЛИТЬ СВОЮ ПОЗИЦИЮ В ГРИДЕ?

- При помощи специализированных встроенных переменных:
 - `threadIdx.x` / `threadIdx.y` / `threadIdx.z` - индексы нити в блоке
 - `blockIdx.x` / `blockIdx.y` / `blockIdx.z` - индексы блока в гриде
 - `blockDim.x` / `blockDim.y` / `blockDim.z` - размеры блоков в НИТЯХ
 - `gridDim.x` / `gridDim.y` / `gridDim.z` - размеры грида в блоках



ПОЗИЦИЯ НИТИ В GRID

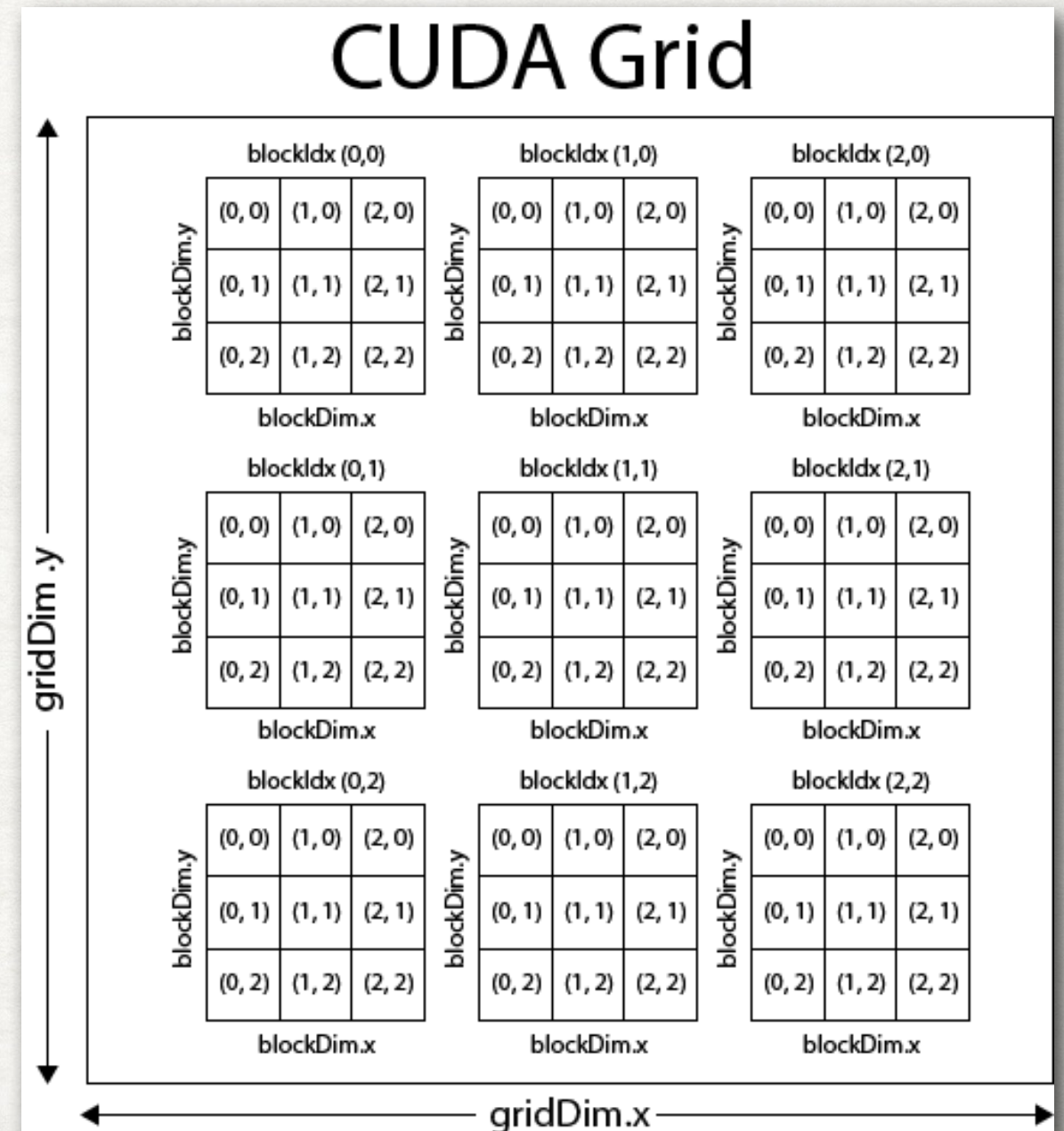
- Индекс нити в одномерном GRID:

int global_index = blockIdx.x * blockDim.x + threadIdx.x;

- Индекс нити в трехмерном GRID:

```
int gridSizeX = blockDim.x *  
gridDim.x; int gridSizeZ = ... ;  
gridSizeY = ... ;  
int gridSizeAll = gridSizeX * gridSizeY  
* gridSizeZ;
```

```
int threadLinearIdx = (threadIdx.z *  
gridSizeY + threadIdx.y) * gridSizeX +  
threadIdx.x;
```



ПРИМЕР: СЛОЖЕНИЕ ВЕКТОРОВ, ДЕВАЙС-КОД

- Как реализовать сложение векторов на GPU?
- Пусть данные уже находятся в памяти GPU
- Тогда:
 1. создадим грид, с числом нитей равным длине входных векторов
 2. каждая нить определит свой линейный индекс i в гриде и выполнит операцию $c[i] = a[i] + b[i]$
- Как определить рассчитать размер грида?
 - Возьмем максимальный размер блок (1024 для P100 GPU)
 - Число блоков = $(N - 1) / 1024 + 1$
 - При таком подходе будет создано достаточно блоков и нитей, чтобы каждая сложила соответствующие пары элементов векторов

ПРИМЕР: СЛОЖЕНИЕ ВЕКТОРОВ, ДЕВАЙС-КОД

```
__global__ void sum_kernel( int *A, int *B, int *C )
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x; //определить свой индекс
    int a = A[idx]; //считать нужный элемент A
    int b = B[idx]; // считать нужный элемент B
    C[idx] = a + b; //записать результат суммирования
}
```

- Каждая нить:
 - Получает копию параметров (В данном случае, это адреса вектором на GPU)
 - Определяет своё положение в гриде threadIdx
 - Считывает из входных векторов элементы с индексом threadIdx и записывает их сумму в выходной вектор по индексу threadIdx
 - рассчитывает один элемент выходного массива
- Вопрос: какой есть недостаток в данном коде?

ПРИМЕР: СЛОЖЕНИЕ ВЕКТОРОВ, ХОСТ-КОД

- А как же хост-код? Как скопировать данные на GPU и запустить ядро?
- Необходимо:
 1. Выделить память на устройстве
 2. Переслать на устройство входные данные
 3. Рассчитать грид, размер грида зависит от размера задачи
 4. Запустить вычисления на гриде, в конфигурации запуска указываем грид
 5. Переслать результат вычислений с устройства на хост

ВЫДЕЛЕНИЕ И ОСВОБОЖДЕНИЕ ПАМЯТИ

- `cudaError_t cudaMalloc (void** devPtr, size_t size)`
Выделяет `size` байтов линейной памяти на устройстве и возвращает указатель на выделенную память в `*devPtr`.
Память не обнуляется. Адрес памяти выровнен по 512 байт
- `cudaError_t cudaFree (void* devPtr)`
Освобождает память устройства на которую указывает `devPtr`.
- Аналогичны стандартным вызовам Malloc/free

КОПИРОВАНИЕ ДАННЫХ

- `cudaError_t cudaMemcpy (void* dst, const void* src, size_t count, cudaMemcpyKind kind)`
- Копирует *count* байтов из памяти, на которую указывает *src* в память, на которую указывает *dst*, *kind* указывает направление передачи
- `cudaMemcpyHostToHost` – копирование между двумя областями памяти на хосте
- `cudaMemcpyHostToDevice` – копирование с хоста на устройство
- `cudaMemcpyDeviceToHost` – копирование с устройства на хост
- `cudaMemcpyDeviceToDevice` – между двумя областями памяти на устройстве

СЛОЖЕНИЕ ВЕКТОРОВ, ПОЛНЫЙ ПРИМЕР

```
int main()
{
    // Size of vectors
    int n = 100000;

    // Host vectors
    double *h_a, *h_b, *h_c;

    // Size, in bytes, of each vector
    size_t bytes = n*sizeof(double);

    // Allocate memory for each vector on host
    h_a = (double*)malloc(bytes);
    h_b = (double*)malloc(bytes);
    h_c = (double*)malloc(bytes);

    int i;
    // Initialize vectors on host
    for( i = 0; i < n; i++ )
    {
        h_a[i] = sin(i)*sin(i);
        h_b[i] = cos(i)*cos(i);
    }
}
```


СЛОЖЕНИЕ ВЕКТОРОВ, ПОЛНЫЙ ПРИМЕР

```
// Device input vectors
double *d_a, *d_b, *d_c;

// Size, in bytes, of each vector
size_t bytes = n*sizeof(double);

// Allocate memory for each vector on GPU
cudaMalloc(&d_a, bytes);
cudaMalloc(&d_b, bytes);
cudaMalloc(&d_c, bytes);

// Copy host vectors to device
cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice);
cudaMemcpy( d_b, h_b, bytes, cudaMemcpyHostToDevice);

int blockSize, gridSize;
// Number of threads in each thread block
blockSize = 1024;
// Number of thread blocks in grid
gridSize = (n-1)/blockSize + 1;
// Execute the kernel
vecAdd<<gridSize, blockSize>>>(d_a, d_b, d_c, n);

// Copy array back to host
cudaMemcpy( h_c, d_c, bytes, cudaMemcpyDeviceToHost );

// Release device memory
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
```


КОМПИЛЯЦИЯ

(ОСОБОЕ ОТНОШЕНИЕ К .CU ФАЙЛАМ)

- При работе с CUDA используются расширения C/Си++:
 - Конструкция запуска ядра<<<....>>>
 - Встроенные переменные threadIdx, blockIdx
 - Квалификаторы __global__ __device__ и т.д.
 -
- Эти расширения могут быть обработаны только в *.cu файлах!
- В этих файлах можно не делать #include <cuda_runtime.h>
- Вызовы библиотечных функций вида cuda* можно располагать в *.cpp файлах
- Они будут слинкованы обычным линковщиком из библиотеки libcudart.so

НЕСКОЛЬКО ВОПРОСОВ...

- Пусть у нас есть массивно-параллельная задача (большой объем параллелизма над данными)
- Достаточно ли просто породить большое число нитей и запустить их на GPU для написания эффективной CUDA-программы?
- **Нет.** Крайне важно понимать, как нити обрабатываются аппаратурой GPU, о чём и пойдет речь в дальнейших лекциях.

ИЗМЕРЕНИЕ ВРЕМЕНИ РАБОТЫ ЯДРА

(БОЛЕЕ ПОДРОБНО О CUDA-СОБЫТИЯХ НА СЛЕДУЮЩИХ ЛЕКЦИЯХ)

```
cudaEvent_t start, stop;  
cudaEventCreate(&start);  
cudaEventCreate(&stop);  
  
cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);  
cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);  
  
cudaEventRecord(start);  
saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);  
cudaDeviceSynchronize();  
cudaEventRecord(stop);  
  
cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);  
  
cudaEventSynchronize(stop);  
float milliseconds = 0;  
cudaEventElapsedTime(&milliseconds, start, stop);
```


ЗАДАНИЯ

ЗАДАНИЕ 1

(СЛОЖЕНИЕ ВЕКТОРОВ)

- Компиляция и запуск алгоритма сложения векторов на системе Polus
- `nvcc -O3 -gencode arch=compute_52,code=sm_52 -gencode arch=compute_60,code=sm_60 main.cu`
- Самостоятельно реализовать проверку корректности результата (host и device)
- Доработать программу, чтобы она могла производить сложение векторов произвольного размера
- Сравнить время работы device ядра (без копирований данных) со сложением на CPU

ЗАДАНИЕ 2

(РАБОТА С ДВУМЕРНЫМ ГРИДОМ)

- Реализуем ядро транспонирования матрицы $a[i][j] = a[j][i]$ (можно без хост кода, итоговую версию запустим в конце дня)
- Как сконфигурировать грид для матрицы размера N ?
- Какую операцию будет производить каждая из нитей?
- Как необходимо хранить матрицу?
- Как скопировать матрицу в память GPU и обратно?

ВОПРОСЫ?