Applied Parallel Computing

parallel-computing.pro

# Высокопроизводительные вычисления, среда программирования CUDA

Шевченко Александр



# Введение

# GPGPU & CUDA

- **GPU** — **G**raphics **P**rocessing **U**nit
- **GPGPU** — **G**eneral-**P**urpose computing on **GPU**, вычисления общего вида на GPU
  - Первые GPU от NVIDIA с поддержкой GPGPU — GeForce восьмого поколения, G80 (2006 г.)
- **CUDA** — **C**ompute **U**nified **D**evice **A**rchitecture
  - Программно-аппаратная архитектура от NVIDIA, позволяющая производить вычисления с использованием графических процессоров

# Семейства GPU NVIDIA

Высокопроизводительные вычисления

Профессиональная графика

Развлечения





# TOP-500

| Rank | Site | System | Cores | Rmax (TFlop/s) | Rpeak (TFlop/s) | Power (kW) |
|---|---|---|---|---|---|---|
| 1 | National University of Defense Technology<br>China | **Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P<br>NUDT | 3,120,000 | 33,862.7 | 54,902.4 | 17,808 |
| 2 | DOE/SC/Oak Ridge National Laboratory<br>United States | **Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x<br>Cray Inc. | 560,640 | 17,590.0 | 27,112.5 | 8,209 |
| 3 | DOE/NNSA/LLNL<br>United States | **Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom<br>IBM | 1,572,864 | 17,173.2 | 20,132.7 | 7,890 |

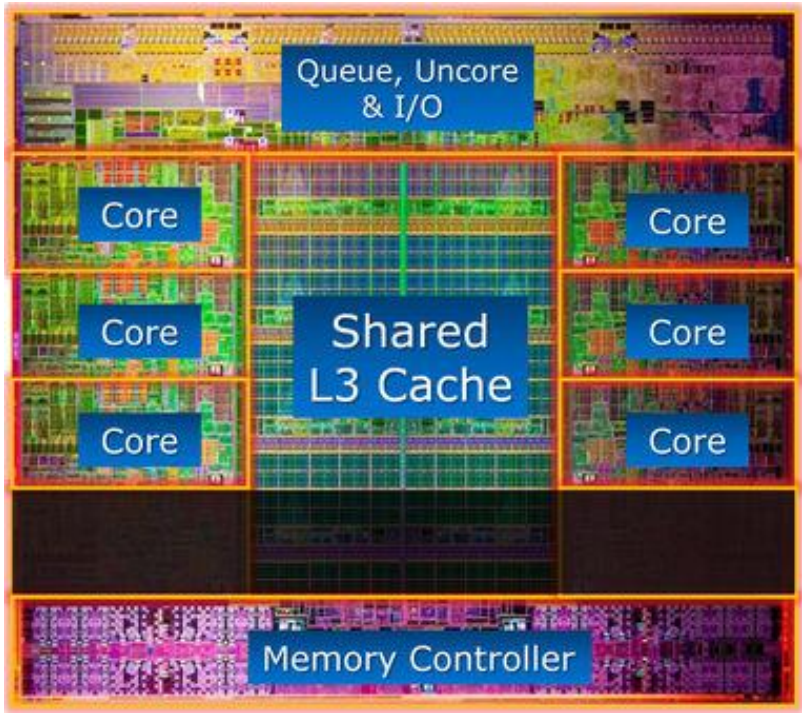


# Аппаратная архитектура GPU NVIDIA

# Особенности CPU Intel Core I-7

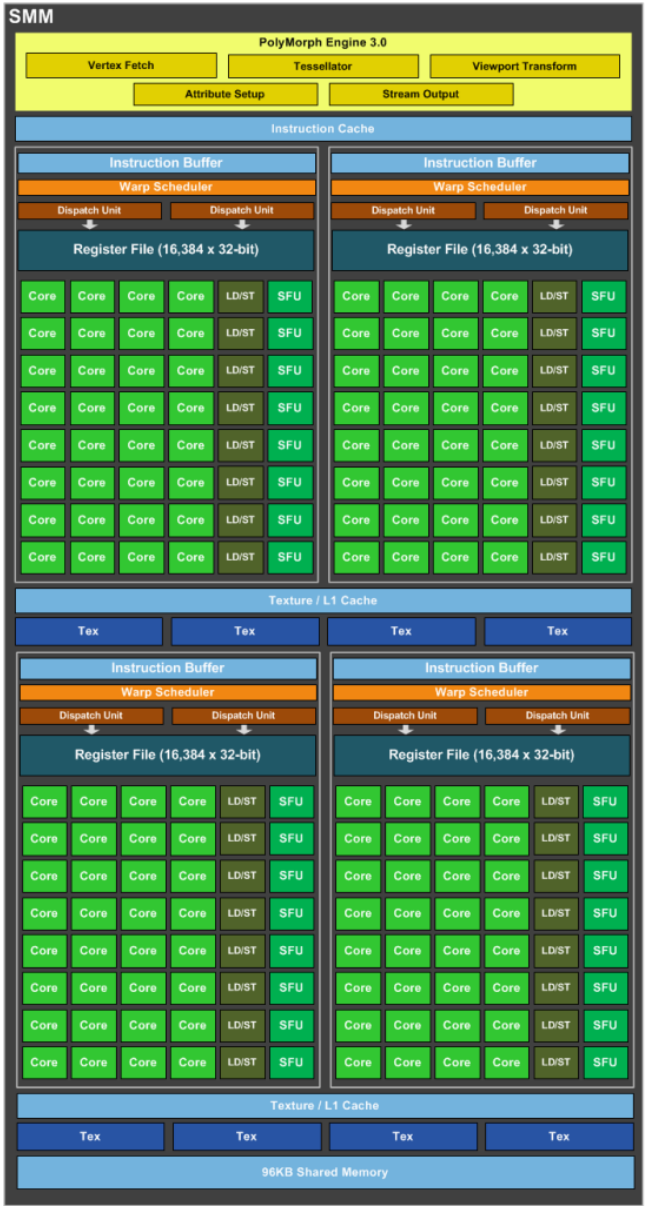
- Небольшое число мощных независимых ядер
  - 2, 4, 6, 8 ядер, 2.66 – 3.6 ГГц каждое
  - Каждое физическое ядро определяется системой как 2 логических и может параллельно выполнять два потока (Hyper-Threading)
- 3 уровня кэшей, большой кэш L3
  - На каждое ядро L1 = 32 КВ (data) + 32 КВ (instructions), L2 = 256 КВ
  - Разделяемый L3 до 20 Mb
- Обращения в память обрабатываются отдельно для каждого процесса/нити

Core I7-3960x,
6 ядер, 15MB L3



# GPU Streaming Multiprocessor (SM)

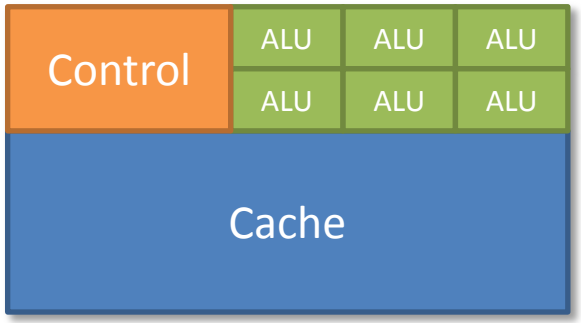
- Потоковый мультипроцессор
- «Единица» построения устройства (как ядро в CPU):
  - 128 скалярных ядра CUDA Core, ~1.2 ГГц
  - 4 Warp Scheduler-а
  - Файл регистров, 256 КВ
  - 3 кэша: текстурный, глобальный (L1), константный (uniform)
  - Текстурные юниты
  - Special Function Unit (SFU) — интерполяция и трансцендентная математика одинарной точности
  - 16x Load/Store unit

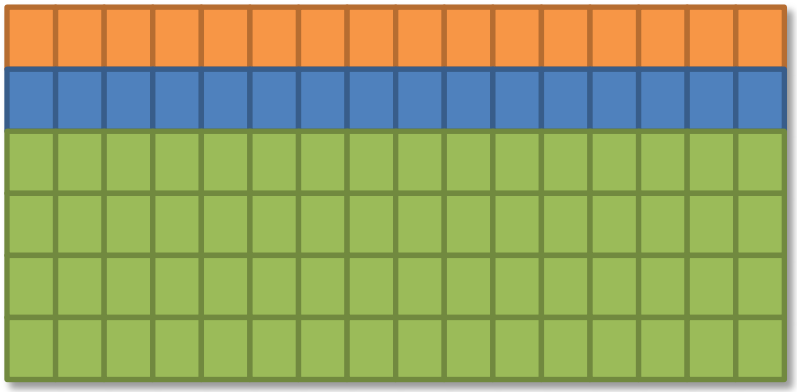


# Чип в максимальной конфигурации

- 16 SM
- 2048 ядер CUDA Core
- Кэш L2 2048 КВ
- GigaThreadEngine
- Контроллеры памяти DDR5
- Интерфейс PCIe3.0

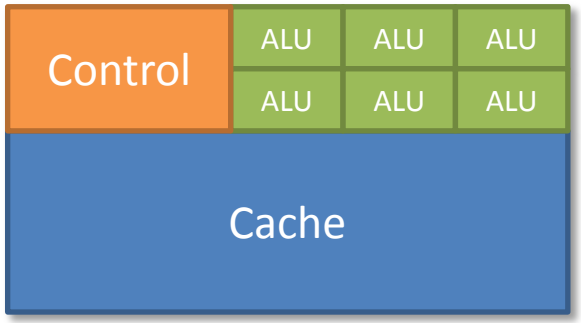
# Отличия GPU от CPU

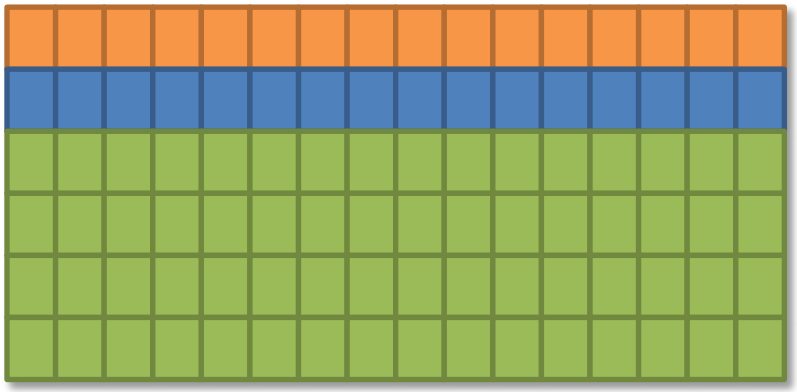


- Тысячи вычислительных ядер
- Небольшие кэши
- Оперативная память с высокой пропускной способностью и высокой латентностью
  - Оптимизирована для коллективного доступа
- Поддержка миллионов виртуальных нитей, быстрое переключение контекста для групп нитей

# Латентность памяти



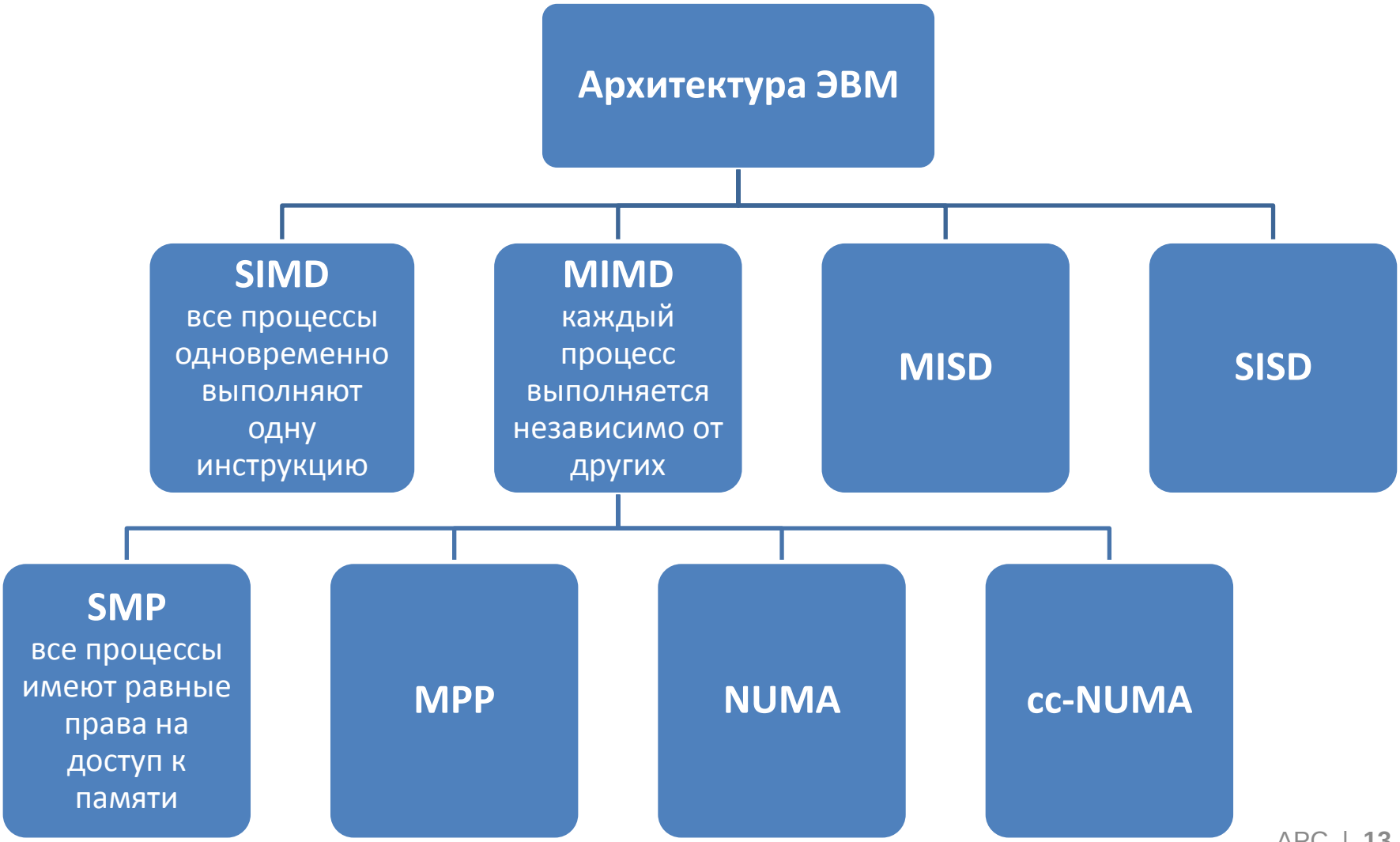



- Цель: **эффективно загружать ядра**
- Проблема: **латентность памяти**
- Решение:
  - **CPU**: Сложная иерархия кэшей
  - **GPU**: Много нитей, покрывать обращения одних нитей в память вычислениями в других за счёт быстрого переключения контекста



# Модель исполнения SIMT

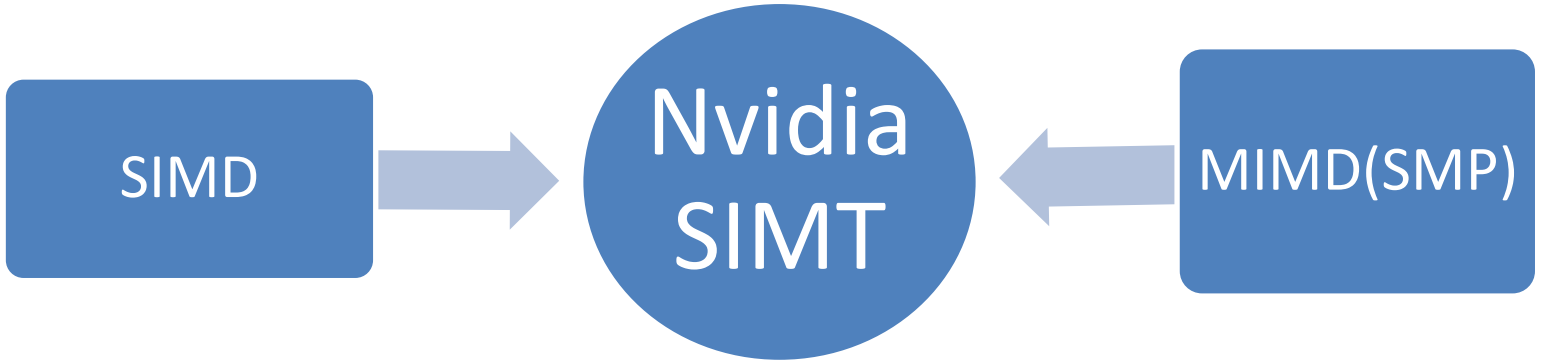
# CUDA и классификация Флинна



# CUDA и классификация Флинна

- У NVIDIA собственная модель исполнения, имеющая черты как SIMD, так и MIMD:

- **NVIDIA SIMT**: Single Instruction — Multiple Thread



# SIMT: виртуальные нити, блоки

- **Виртуально** все нити:
  - Выполняются параллельно (MIMD)
  - Имеют одинаковые права на доступ к памяти (MIMD:SMP)

- Нити разделены на группы одинакового размера (**блоки**):
  - В общем случае, **глобальная синхронизация всех нитей невозможна**, нити из разных блоков выполняются полностью независимо и не могут управляемо взаимодействовать
  - **Есть локальная синхронизация внутри блока**, нити из одного блока могут взаимодействовать через специальную память

- Нити не мигрируют между блоками. Каждая нить находится в своём блоке с начала выполнения и до конца

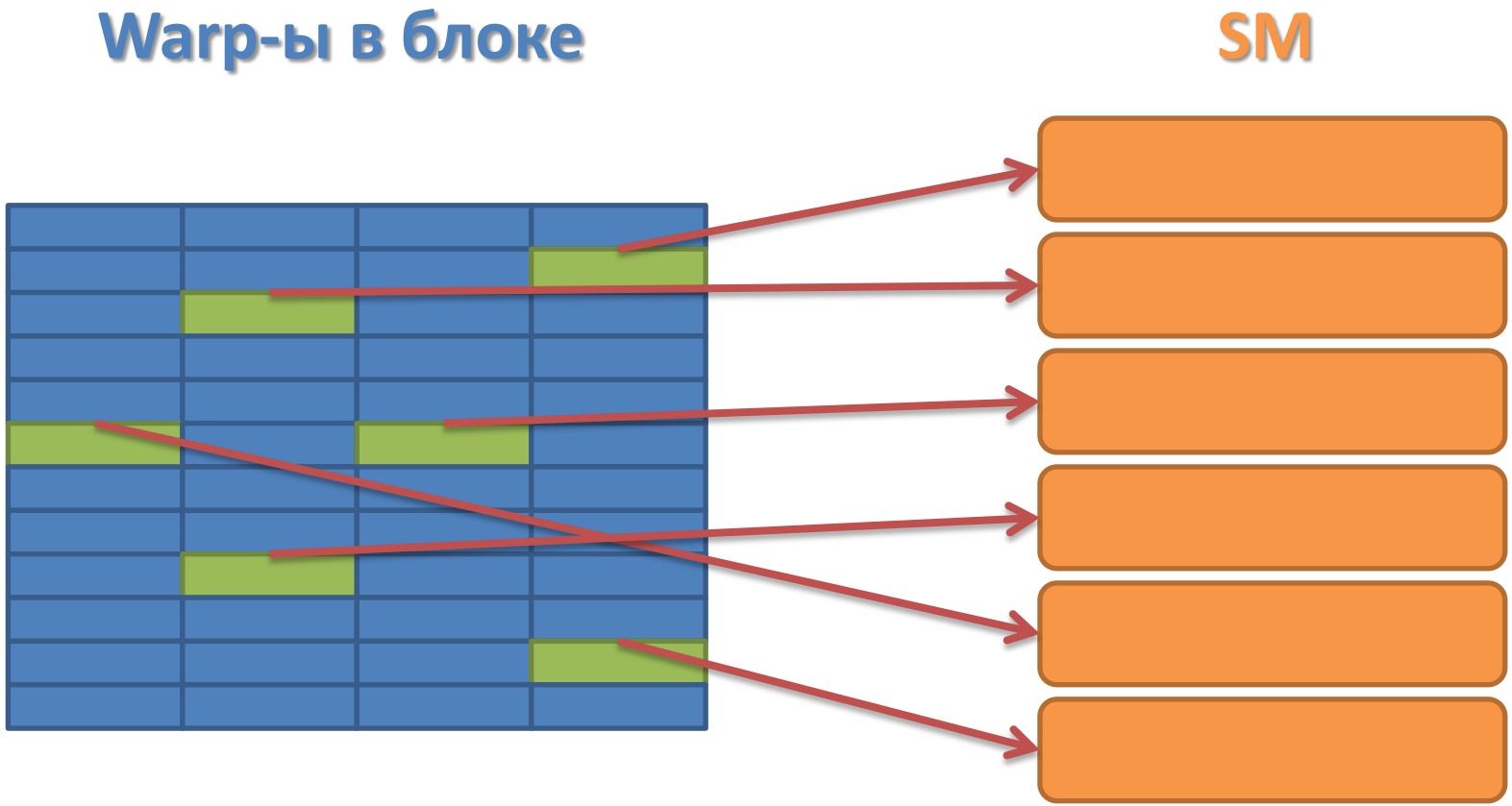
# SIMT: аппаратное выполнение

- Все нити из одного блока выполняются на одном мультипроцессоре (SM)
- Блоки не мигрируют между SM
- Максимальное число нитей в блоке **ограничено**
- Каждый SM работает **независимо от других**

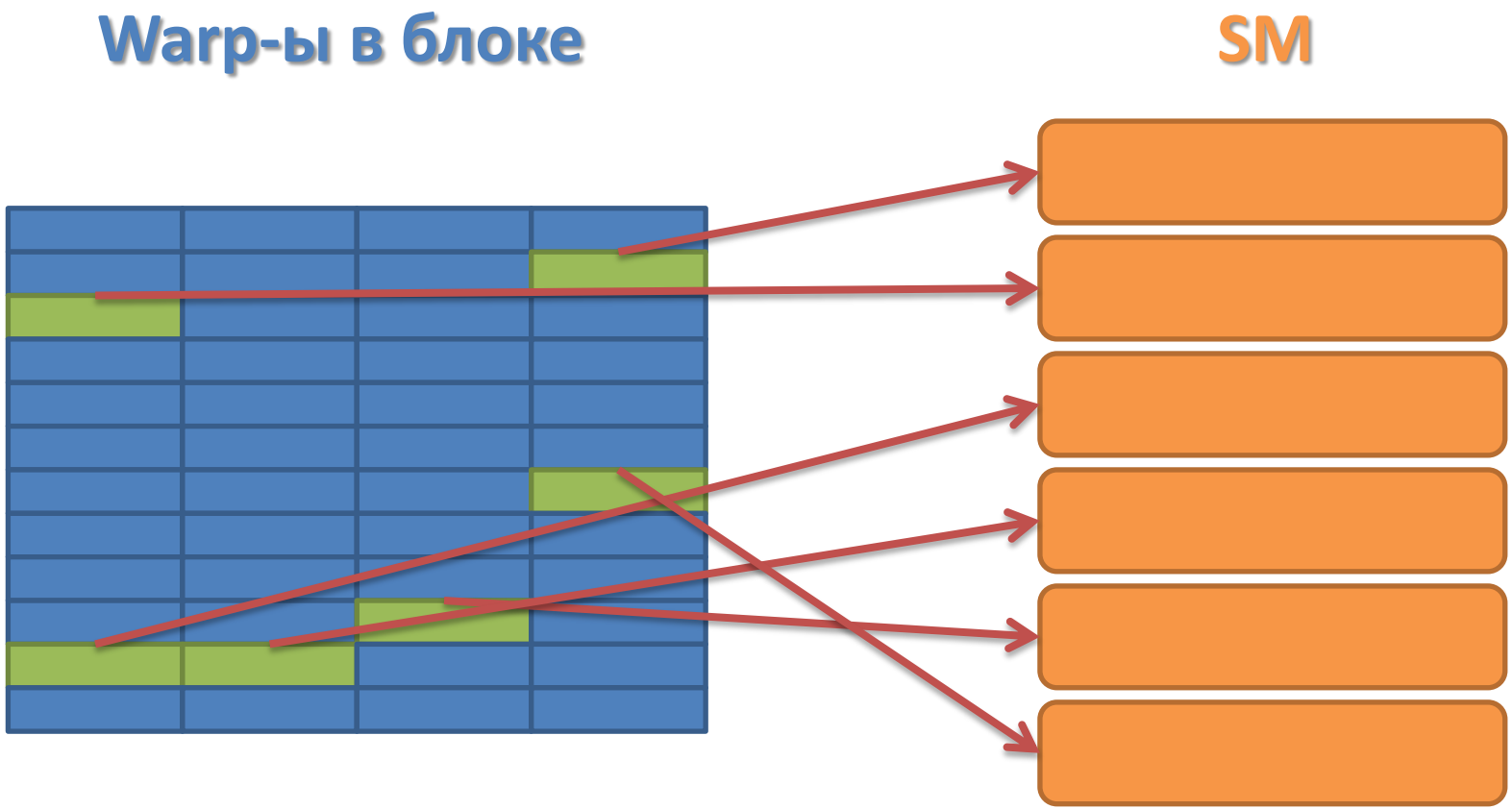
# SIMT: аппаратное выполнение

- Блоки разделяются на группы по 32 нити, называемые **warp**
- Все нити warp-а **одновременно** выполняют **одну общую** инструкцию (в точности SIMD-выполнение)
- Планировщик warp-ов на каждом цикле работы выбирает warp, все нити которого готовы к выполнению следующей инструкции, и запускает его

# Пример



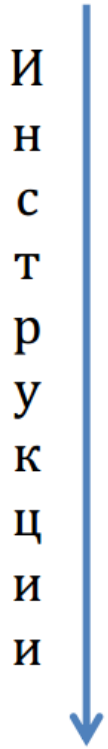
# Пример

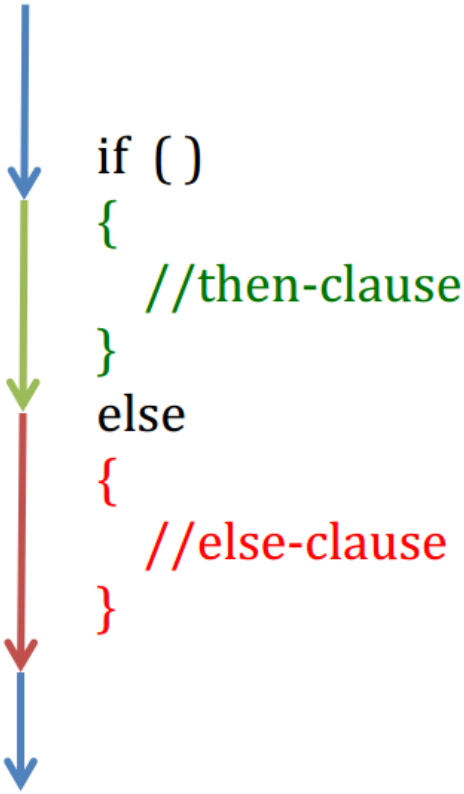


# Ветвление (branching)

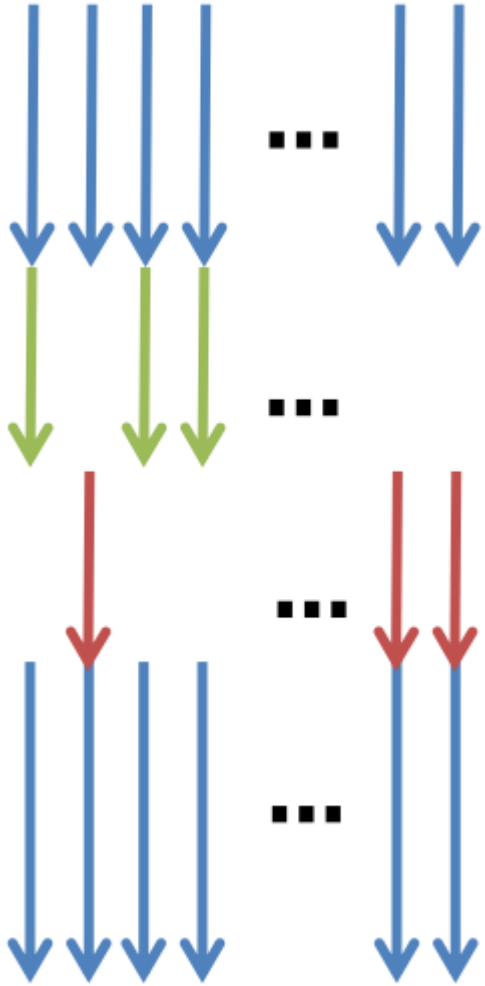
- Все нити warp-а одновременно выполняют одну и ту же инструкцию
- Как быть, если часть нитей эту инструкцию выполнять не должна?
  - if(<условие>), где значение условия различается для нитей одного warp-а
- Эти нити «замаскируются» нулями в специальном наборе регистров и не будут её выполнять, т.е. **будут простаивать**

# Ветвление (branching)



```
if ( )
{
   //then-clause
}
else
{
   //else-clause
}
```

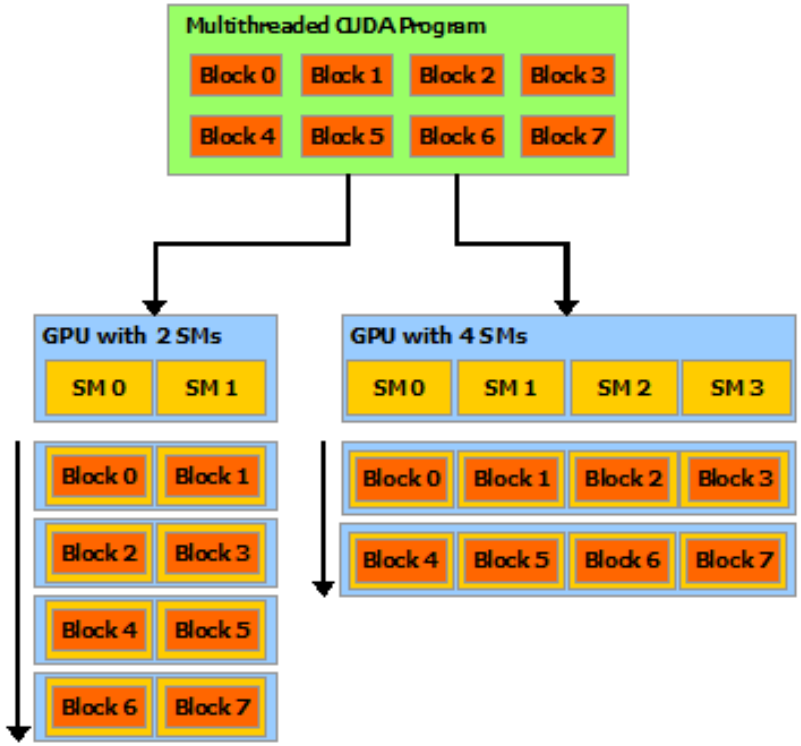


# Глобальная синхронизация

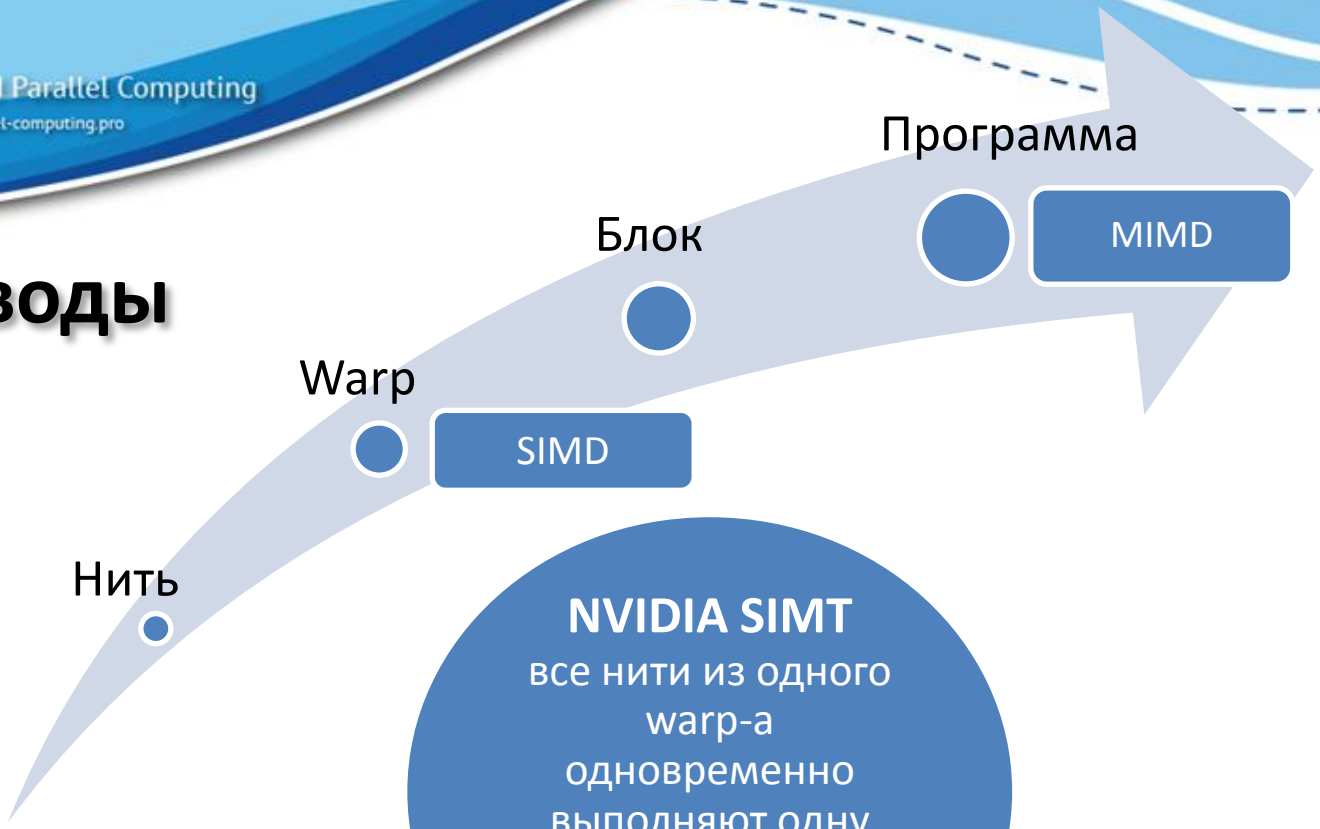
- В общем случае, из-за ограничений по числу нитей и блоков на одном SM не удаётся разместить сразу все блоки программы на GPU
  - Часть блоков ожидает выполнения
    - ✓ **Поэтому в общем случае невозможна глобальная синхронизация**
  - Блоки выполняются по мере освобождения ресурсов
    - ✓ Нельзя предсказать порядок выполнения блоков

# SIMT и масштабирование

- Виртуальная часть
  - GPU может поддерживать миллионы виртуальных нитей
  - Виртуальные блоки независимы
    - ✓ Программу можно запустить на любом количестве SM
- Аппаратная часть
  - Мультипроцессоры независимы
    - ✓ Можно «нарезать» GPU с различным количеством SM



# Выводы




**NVIDIA SIMT**
все нити из одного warp-а одновременно выполняют одну инструкцию, warp-ы выполняются независимо

SIMD — все нити одновременно выполняют одну инструкцию

MIMD — каждая нить выполняется независимо от других, SMP — все нити имеют равные возможности для доступа к памяти



# CUDA: Гибридное программирование CPU+GPU

# Вычисления с использованием GPU

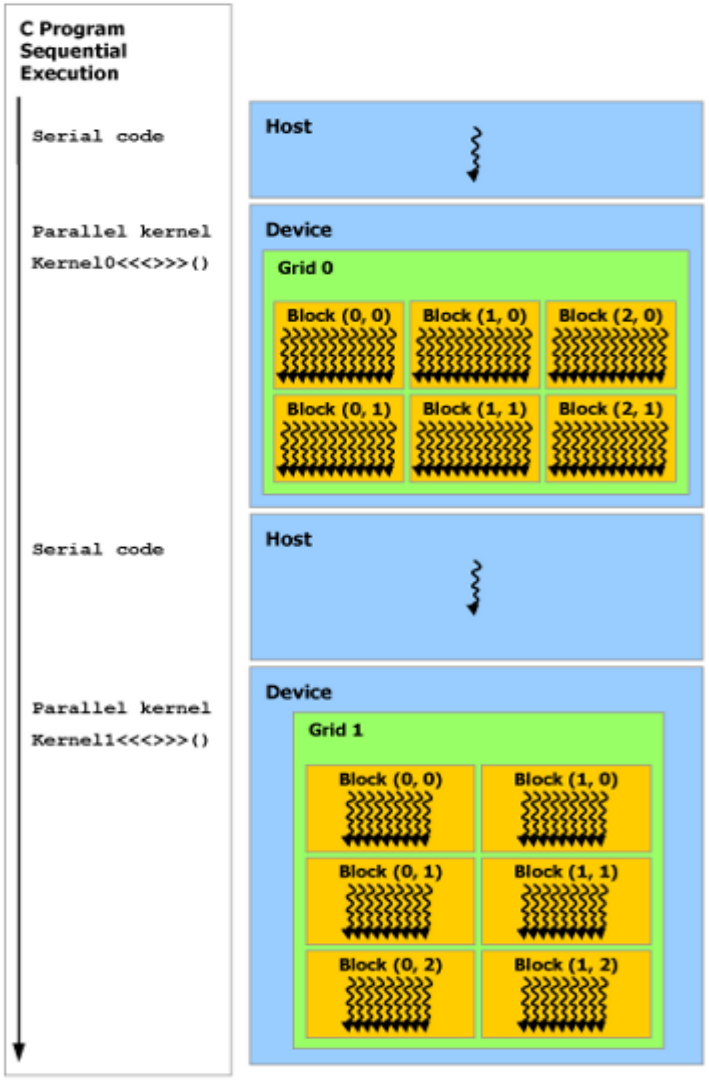
- Программа, использующая **GPU**, состоит из:
  - Кода для **GPU**, описывающего необходимые вычисления и работу с памятью устройства
  - Кода для **CPU**, в котором осуществляется
    - ✓ **Управление памятью GPU** — выделение/освобождение
    - ✓ **Обмен данными** между **GPU**/**CPU**
    - ✓ **Запуск кода** для **GPU**
    - ✓ Обработка результатов и прочий последовательный код

# Вычисления с использованием GPU

- **GPU** рассматривается как периферийное устройство, управляемое центральным процессором
  - **GPU** «пассивно», т.е. не может само загрузить себя работой
- Код для **GPU** можно запускать из любого места программы как обычную функцию
  - «Точечная», «инкрементная» оптимизация/портирование программ

# Используемая терминология

- **CPU** = «хост» (от англ. host)
  - код для CPU — код для хоста, «хост-код» (host-code)
- **GPU** = «устройство» или «девайс» (от англ. device)
  - код для GPU — «код для устройства», «девайс-код» (device-code)



# Код для GPU (device-code)

- Код для GPU пишется на C++ с некоторыми надстройками:
  - 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
  - …
- Компилируется специальным компилятором **cicc**

# Код для CPU (host-code)

- Код для CPU дополняется вызовами специальных функций для работы с устройством
- Код для CPU компилируется обычным компилятором
  - Кроме конструкции запуска ядра <<<. . .>>>
- Функции линкуются из динамических библиотек

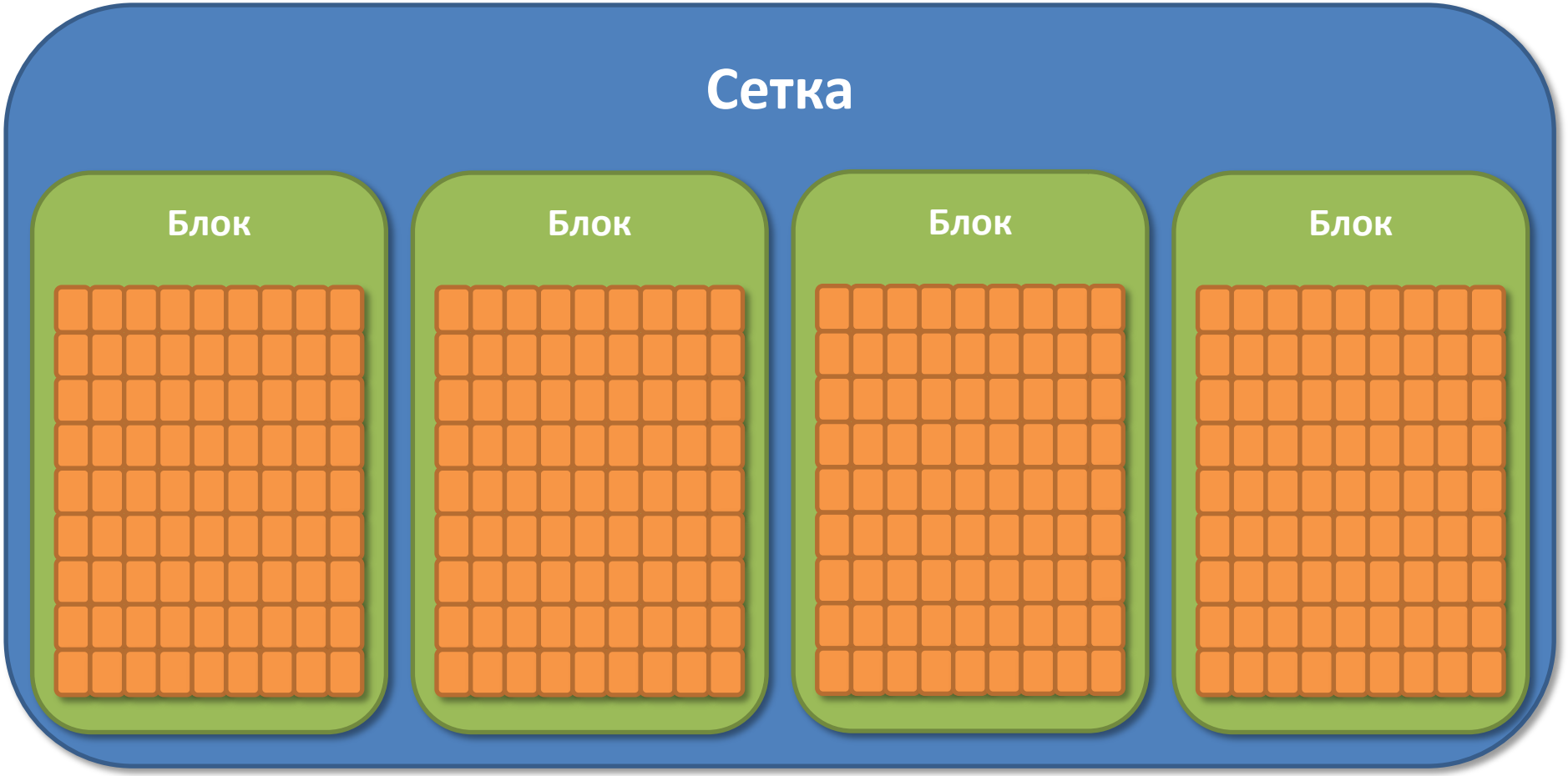
# CUDA Kernel («Ядро»)

- Специальная функция, являющая **входной точкой** для кода на **GPU**
  - ✓ Нет возвращаемого значения (void)
  - ✓ Выделена атрибутом **__global__**
- Объявления параметров и их использование такое же, как и для обычных функций

```
__global__ void kernel (int *ptr) {
    ptr = ptr + 1;
    ptr[0] = 100;
    ….; //other code for GPU
}
```

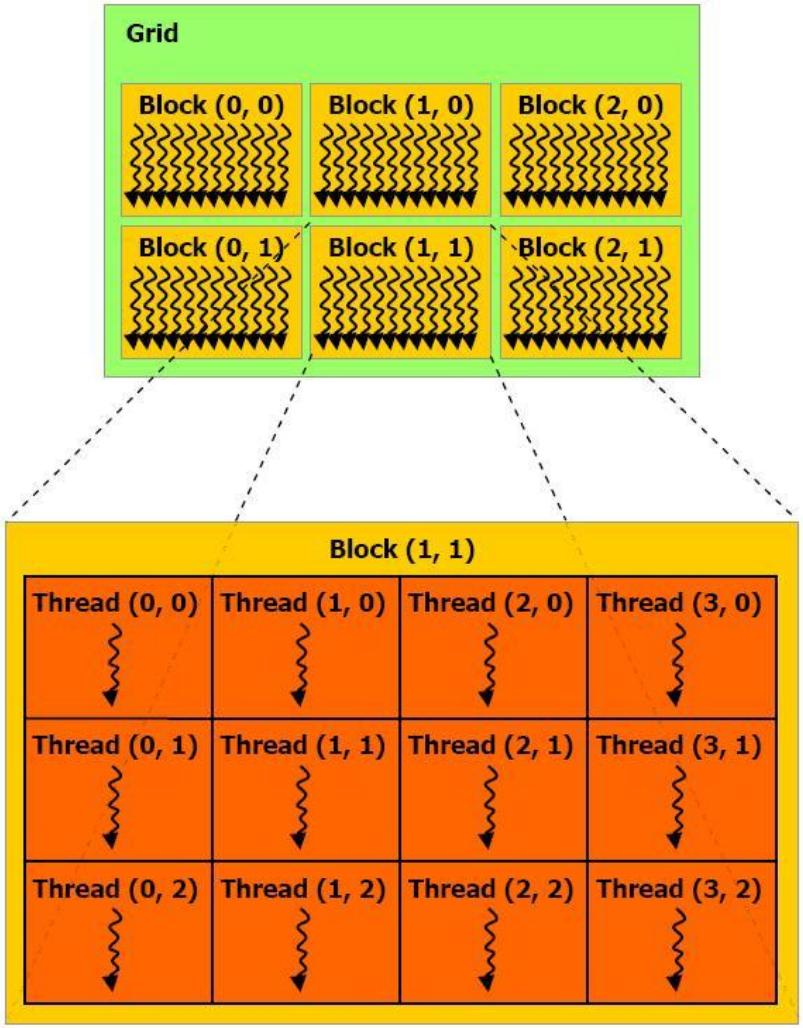
- **CPU** запускает именно **ядра**, **GPU** их выполняет

# Иерархия нитей исполнения



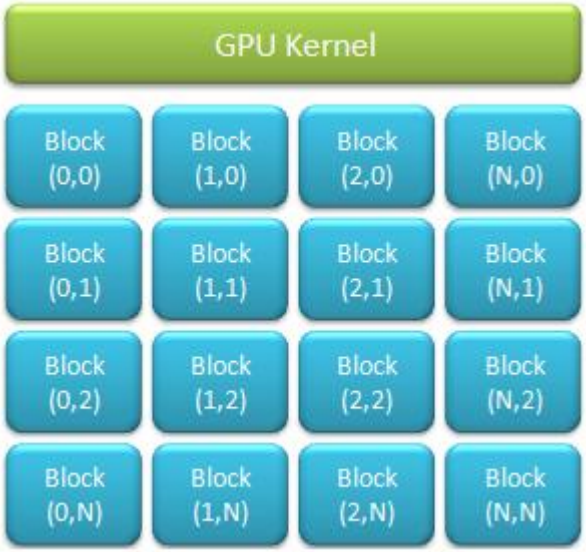
# CUDA Grid

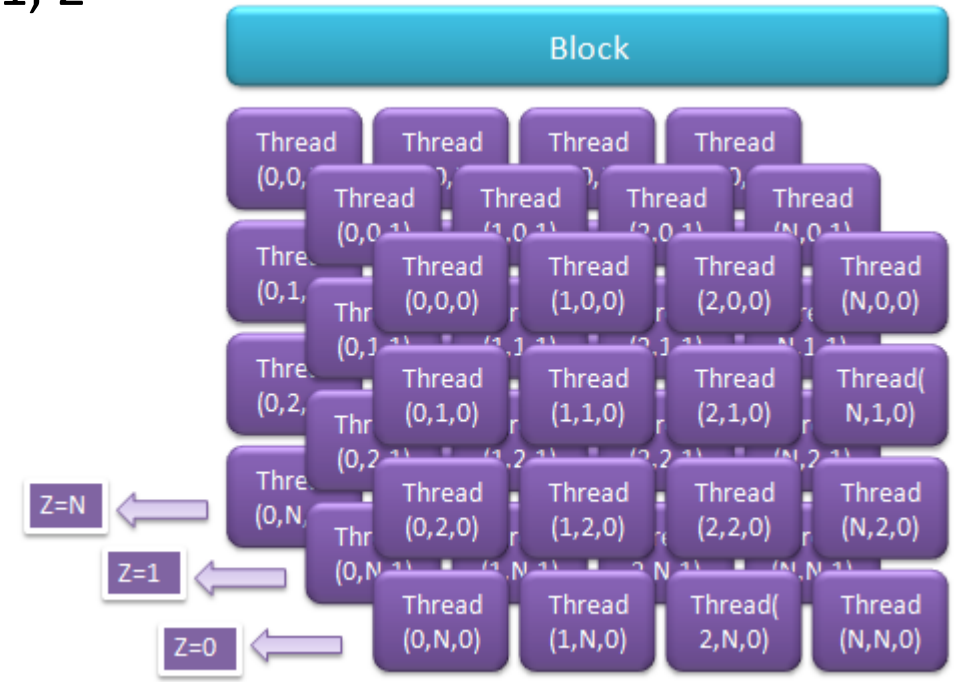
- Положение нити в блоке и блока в сетке индексируются по трём измерениям (*x, y, z*)
- Сетка задаётся количеством блоков по *x*, *y*, *z* (размер в блоках) и размерами каждого блока по *x*, *y*, *z*
- Если по *z* размер сетки и блоков равен единице, то получаем плоскую прямоугольную сетку нитей



# CUDA Grid

- Двумерная сетка из трёхмерных блоков
  - Логический индекс по переменной *z* у всех блоков равен нулю
  - Каждый блок состоит из трёх «слоёв» нитей, соответствующих *z* = 0, 1, 2



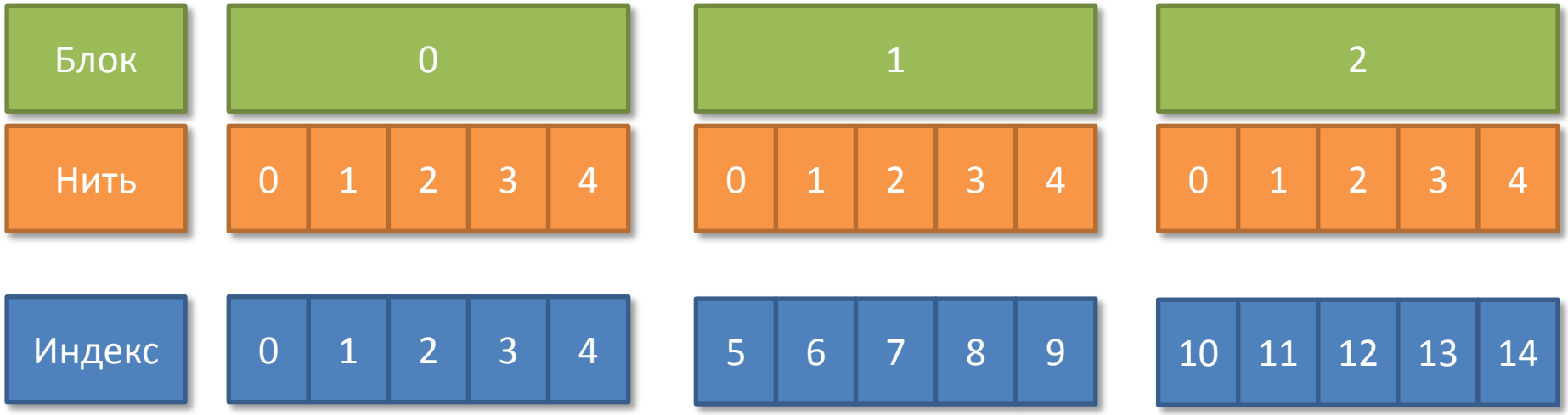
# Управление нитями

- Осуществляется за счёт встроенных переменных:
  - **dim3 threadIdx** – индексы нити в блоке
  - **dim3 blockIdx** – индексы блока в сетке
  - **dim3 blockDim** – размеры блоков в нитях
  - **dim3 gridDim** – размеры сетки в блоках
  - Линейный индекс нити в сетке:

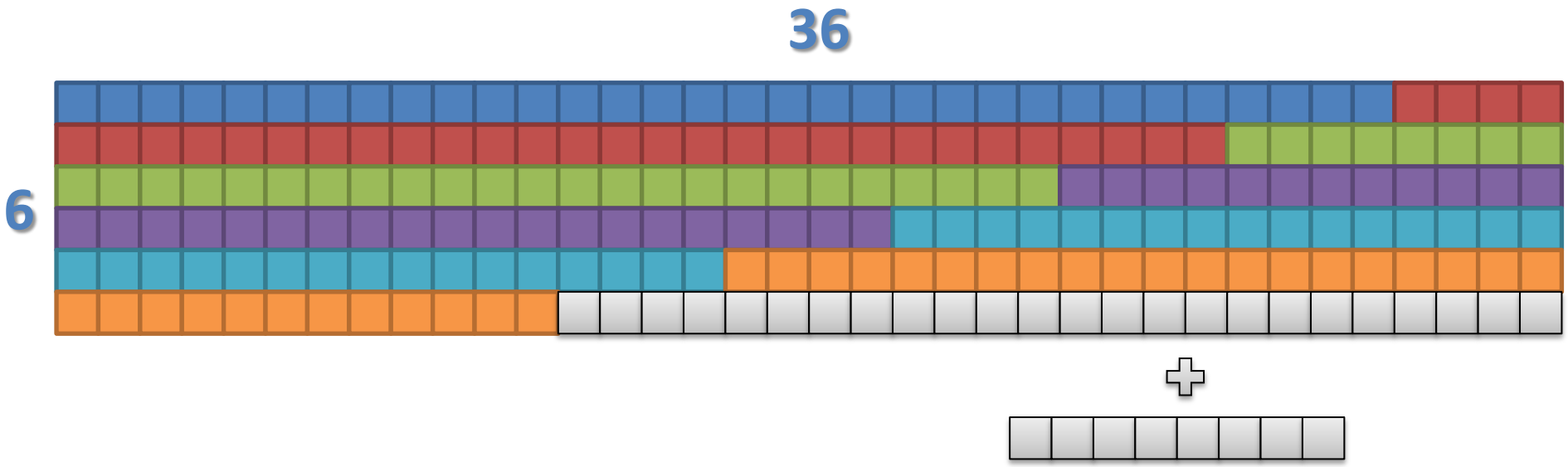
```
int threadLinearIdx =
  blockIdx.x * blockDim.x + threadIdx.x;
```

# Управление нитями

```
int threadLinearIdx = blockIdx.x * blockDim.x + threadIdx.x;
```

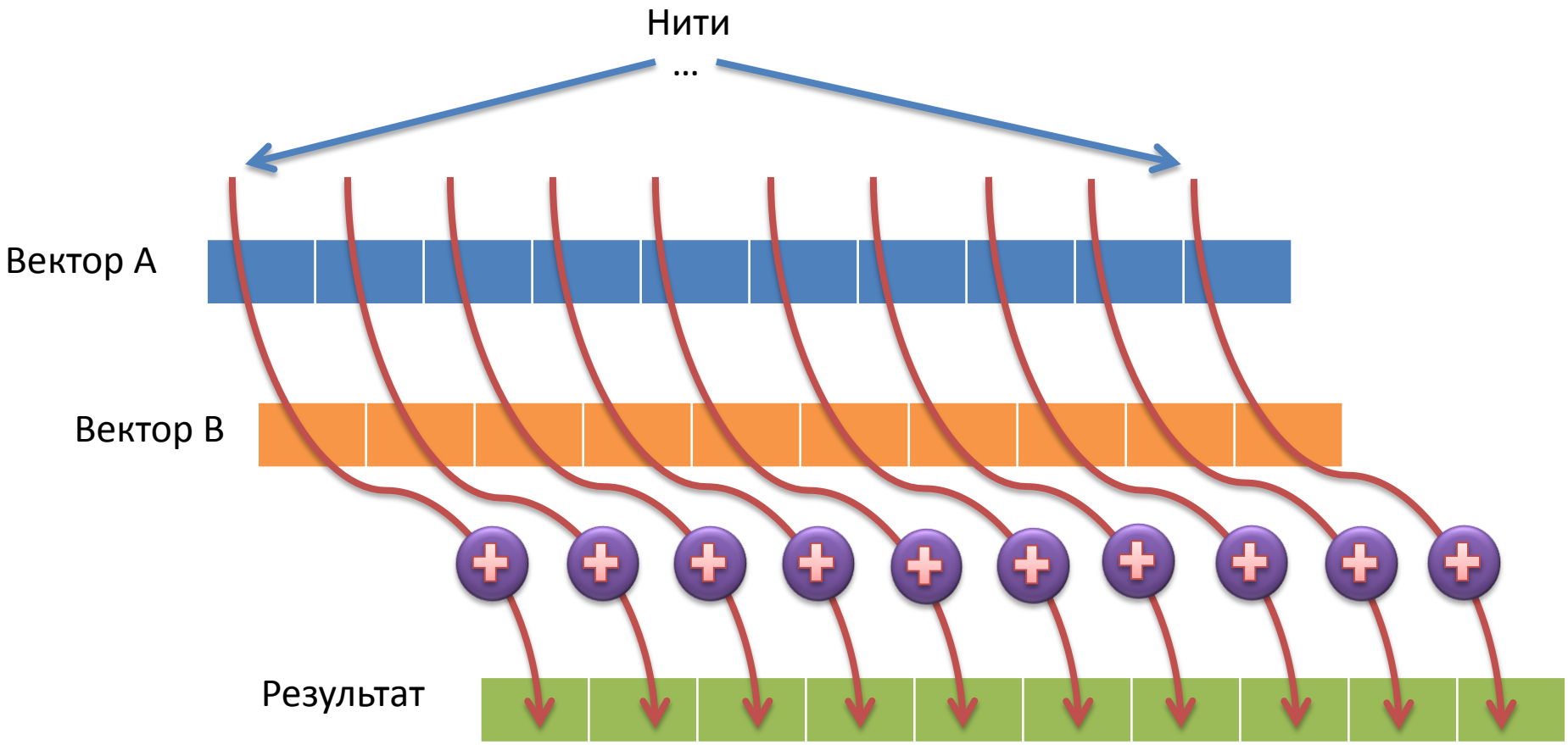


# Распределение по warp-ам




```
int threadLinearIdx =
  threadIdx.y * blockDim.x + threadIdx.x;
```

# Сложение векторов



# Сложение одномерных векторов

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

- Каждая нить
  - Получает копию параметров
    - ✓ В данном случае это адреса векторов на GPU
  - Определяет своё положение в сетке **threadLinearIdx**
  - Считывает из входных векторов элементы с индексом **threadLinearIdx** и записывает их сумму в выходной вектор по индексу **threadLinearIdx**

# Host Code

- Выбрать устройство
  - По умолчанию устройство с номером 0
- Выделить память на устройстве
- Переслать на устройство входные данные
- Рассчитать размеры сетки
- Запустить ядро
- Переслать с устройства на хост результат

# Выделение 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**

# Копирование

- `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
  - Вызов **cudaMemcpy()** с **kind**, не соответствующим **dst** и **src** , приводит к непредсказуемому поведению

# Запуск ядра

- `kernel <<<execution configuration>>> (params);`
  - “kernel” — имя ядра,
  - “params” — параметры ядра, копию которых получит каждая нить
- `execution configuration` — `Dg, Db (базовая)`
  - `dim3 Dg` — размеры сетки в блоках
  - `dim3 Db` — размер каждого блока
- `struct dim3` — структура, определённая в CUDA Toolkit
  - Три поля: `uint x, y, z`
  - Конструктор
    `dim3(uint x = 1, uint y = 1, uint z = 1)`



# Компиляция и запуск

# .cu файлы

При работе с CUDA используются расширения C++:

- Конструкция запуска ядра **<<< …. >>>**
- Встроенные переменные **threadIdx**, **blockIdx**
- Идентификаторы **__global__** **__device__** и т.д.
- …

- Эти расширения могут быть обработаны только в ***.cu** файлах!
  - **cudafe не запускается для файлов с другим расширением**
  - В этих файлах можно **не** делать **#include <cuda_runtime.h>**
- Вызовы библиотечных функций вида **cuda*** можно располагать в ***.cpp** файлах
  - Они будут скомпонованы обычным линковщиком из библиотеки libcudart.so

# Компиляция С кода

Рассмотрим **test.cpp**:

- Вызов ядра нельзя поместить в *.cpp:

```
#include <cuda_runtime.h>  // Toolkit functions declarations
void launchKernel(params); // define this function in *.cu
int main(){
        … // typical host code
        cudaSetDevice(0); // Allowed usage of cudart library functions
        … // typical host code
        launchKernel(params); // This function contains kernel invocation
                              // Defined in *.cu
        … // typical host code
}
```

- Компиляция:

```
g++ -I /toolkit_install_dir/include test.cpp –c –o test.o
```

  - ✓ /toolkit_install_path/include — путь к CUDA Toolkit
  - ✓ -c -o test.o — компиляция в test.o

При использовании nvcc путь к Toolkit можно не указывать: `nvcc test.cpp –c –o test.o`

# Компиляция CUDA кода

Рассмотрим **kernel.cu**:

- Функция-ядро и функция, которая её запускает. При этом должна указываться конфигурация запуска

```
__global __ void kernel(params)  {
        ...; kernel code
}
void launchKernel(params)  {
        ...; // launch parameters configuring
        kernel<<< configuration >>> (params); // kernel launch
}
```

- Компиляция:

```
nvcc –arch=sm_52 –Xptxas –v kernel.cu -c –o kernel.o
```

# Компоновка проекта

- `g++ -L/toolkit_install_dir/lib64 -lcudart test.o kernel.o –o test`
  - Использование libcudart.so: указали, где она находится

- `nvcc test.o kernel.o –o test`
  - nvcc –v test.o kernel.o –o test  покажет, какая конкретно команда вызвалась

**Также можно расположить весь код в *.cu файле и не пользоваться *.cpp вообще**

Подробнее см. в CUDA Compiler Driver NVCC

# Запуск

- В результате сборки проекта получаем обычный исполняемый файл

- Запускаем из командной строки обычным способом
  - $./test 1024



# Обработка ошибок

# Обработка ошибок

- Большая часть функций из runtime библиотеки возвращает `cudaError_t`
- Проверяя эти ошибки, можно идентифицировать некоторые проблемы исполнения

```
#define CUDA_CALL(x) do{ \
  cudaError_t err = (x); \
  if (err != cudaSuccess) { \
    printf ("Error \"%s\"\n", \
      cudaGetErrorString(err)); \
    exit(-1); \
}} while (0)
```

# Обработка ошибок

- Коды ошибок записываются в специальную переменную типа `enum cudaError_t`
  - Эта переменная равна коду **последней** ошибки
  - `cudaError_t cudaPeekAtLastError()` — возвращает текущее значение этой переменной
  - `cudaError_t cudaGetLastError()` — возвращает текущее значение этой переменной и присваивает ей `cudaSuccess`
  - `const char* cudaGetErrorString (cudaError_t error)` — по коду ошибки возвращает её текстовое описание

# Обработка ошибок

- Простейший способ быть уверенным, что в программе не произошло CUDA-ошибки:
  - Добавить в конце main.c

```
std::cout << cudaGetErrorString(
  cudaGetLastError());
```



# Ошибки работы с памятью

# Ошибки работы с памятью

- В отличии от CPU не идентифицируются автоматически при исполнении

- Использование утилиты **`cuda-memcheck`** упрощает их поиск



# Асинхронность в CUDA

# Асинхронность в CUDA

- Некоторые CUDA вызовы являются асинхронными
  - Отправляют команду на устройство и сразу возвращают управление CPU

- В том числе:
  - Конструкция вызова функции-ядра
  - Функции копирования памяти `*Async`
  - Другие

# Асинхронность в CUDA

- Почему тогда верно работает код?

```
//запуск ядра (асинхронно)
sum_kernel<<<blocks,  threads>>>(aDev, bDev, cDev);
//переслать результаты обратно
cudaMemcpy(cHost, cDev, nb, cudaMemcpyDeviceToHost);
```

- CPU вызывает **cudaMemcpy** до завершения выполнения ядра

# CUDA Stream

- **CUDA Stream** (очередь исполнения) — последовательность команд для GPU (запуски ядер, копирования памяти и т.д.), исполняемая строго последовательно, следующая выполняется после завершения предыдущей
- Команды из разных очередей могут выполняться параллельно, не зависит от исполнения команд в других очередях
- По умолчанию все команды помещаются в **Default Stream**, имеющий номер 0

# Асинхронность в CUDA

- Почему тогда верно работает код?

```
//запуск ядра (асинхронно)
sum_kernel<<<blocks,  threads>>>(aDev, bDev, cDev);
//переслать результаты обратно
cudaMemcpy(cHost, cDev, nb, cudaMemcpyDeviceToHost);
```

- Вызов функции ядра и **cudaMemcpy** попадают в один поток (Default Stream)

# Явная синхронизация

- `cudaError_t cudaDeviceSynchronize()`
  - Ожидает завершения выполнения всех задач, отправленных к моменту вызова на GPU

- Синхронизация выполнения ядра:

```
kernel<<<configuration>>>(…); // Async
cudaDeviceSynchronize(); // Sync
cudaMemcpyAsync(…); // Works after kernel
```



# Работа с сервером

# Putty (Windows)

- Скачать и установить программу Putty
- При запуске установить свойства:
  - Session -> Host name: satok605.mipt.su
  - Session -> Port: 22
  - Window -> Translation -> RCS: UTF-8
- Сохраните сессию
- Скачать и установить WinSCP для копирования файлов

# Выводы

Хорошо распараллеливаются на GPU задачи, которые:

- Имеют параллелизм по данным
  - одна и та же последовательность вычислений, применяемая к разным данным
- Состоят из подзадач примерно одинаковой сложности
  - подзадача будет решаться блоком нитей
- Каждая подзадача может быть выполнена независимо от всех остальных
  - нет потребности в глобальной синхронизации
- Число арифметических операций велико по сравнению с операциями доступа в память
  - для покрытия латентности памяти вычислениями
- Если алгоритм итерационный, то его выполнение может быть организовано без пересылок памяти между хостом и GPU после каждой итерации
  - пересылки данных между  хостом и GPU накладны

# Спасибо за внимание!

**Шевченко Александр**
aleksandr.shevchenko@phystech.edu