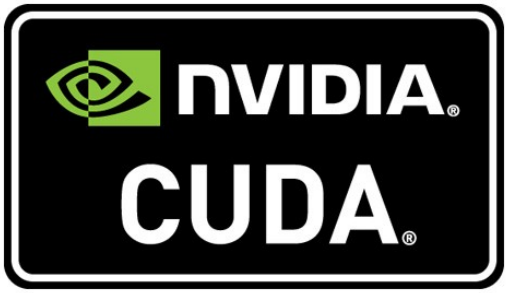
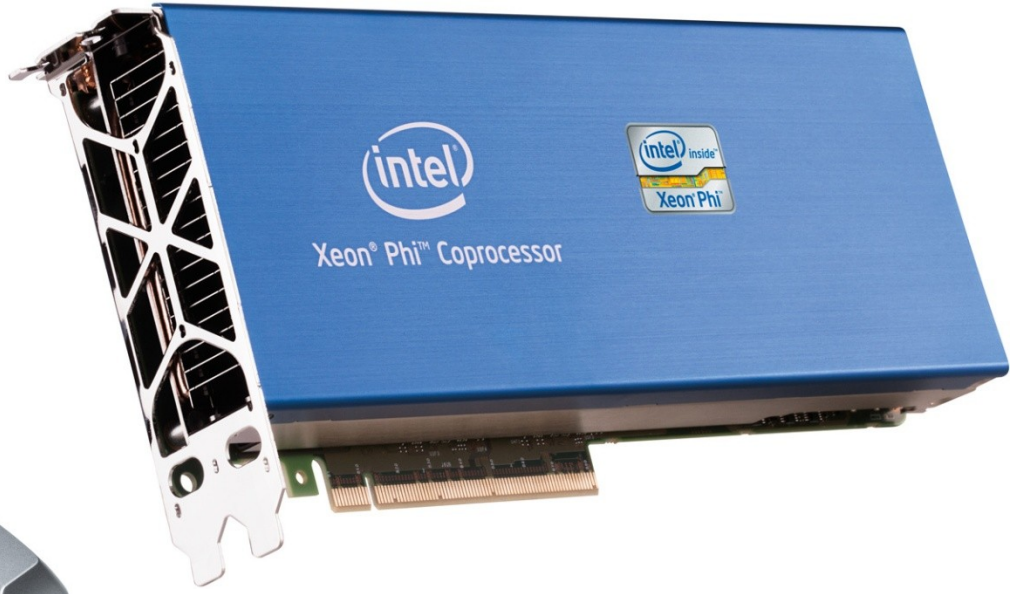
Программирование ускорителей параллельных вычислений


NVIDIA
CUDA


intel
Xeon® Phi™ Coprocessor
intel inside
Xeon Phi


TITAN
NVIDIA


AMD

План курса

- Введение в технологию программирования CUDA.
- Потоки и блоки. SIMT режим выполнения потоков. Атомарные функции
- Многоуровневая память видеокарты. Оптимизация доступа к памяти.
- Работа с разделяемой памятью видеокарты.
- CUDA-библиотеки
- Программирование тензорных ядер
- Технология OpenCL

Лабораторные работы

1. Простое задание на обработку графики
2. Профилирование CUDA-программы
3. Работа с управляемым кешем видеокарты
4. Теоретическая оценка производительности
5. Программирование тензорных ядер
6. OpenCL

Оценивание

- 100 - выполнение лабораторных работ:
  - 1 – 10
  - 2 – 5
  - 3+4 – 27 + 10
  - 5 – 40
  - 6 – 8

Программное обеспечение

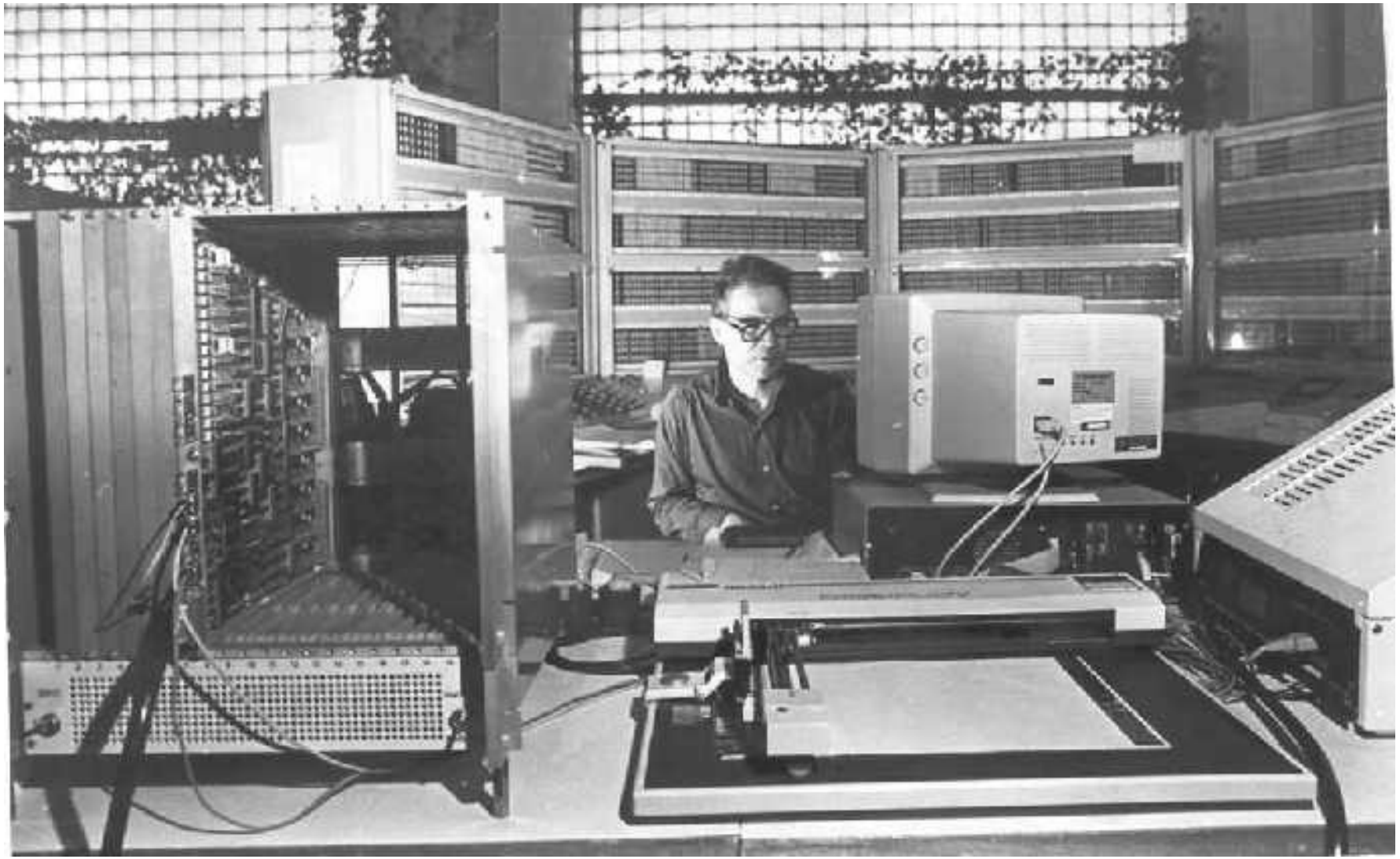
- CUDA SDK
  - компилятор nvcc
  - система разработки Nsight (VS)
  - профилировщик Nsight Compute
  - библиотеки

Доступные вычислители

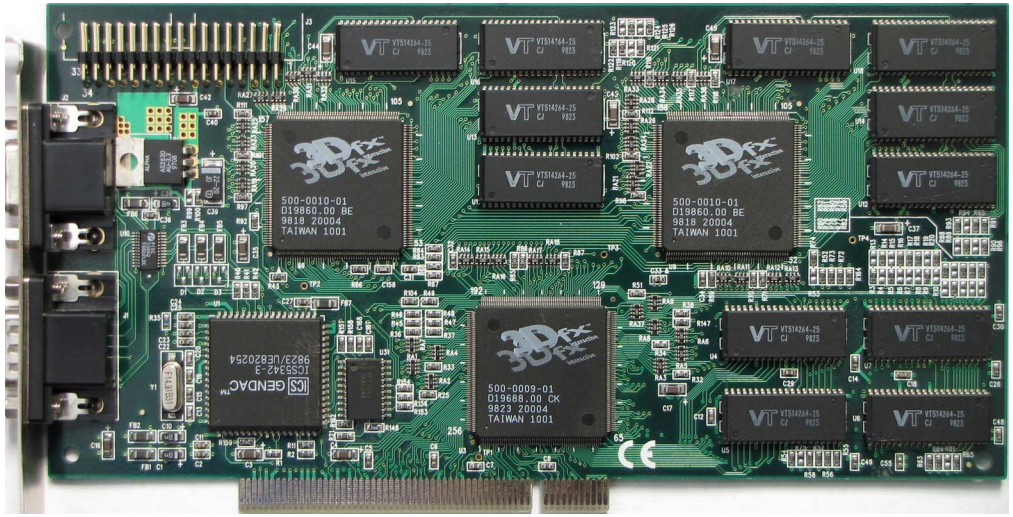
- GeForce RTX 2060 (1920 ядер, 14.2TFlops, 336ГБ/с)
- NVIDIA Titan (2880 ядер, 4.5 TFlops)
- NVIDIA Tesla C2075 (448 ядер, 1.03 TFlops)
- Xeon Phi 7120 (61 ядро, 2.4 TFlops)
- процессоры рабочих компьютеров (д/з – найти параметры и вычислить теоретическую производительность)

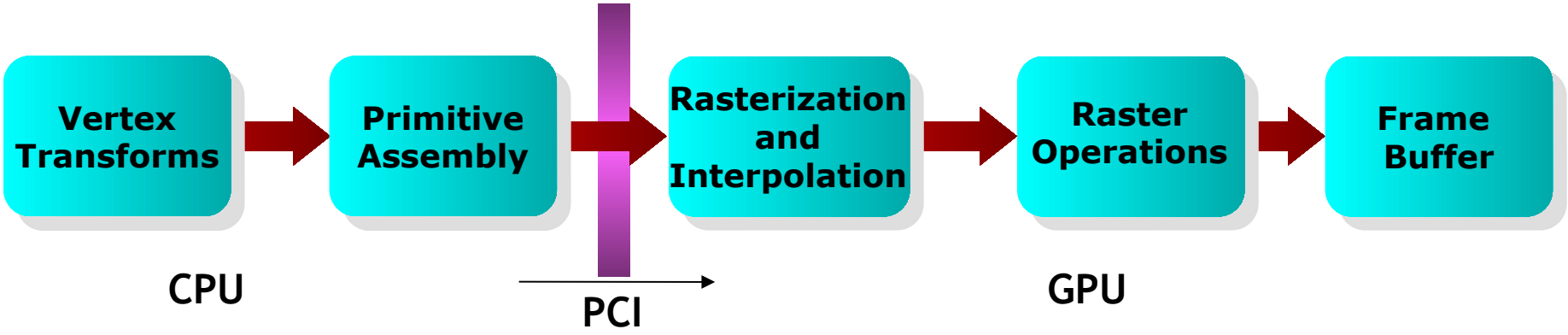
История

Далекие 60-70е годы

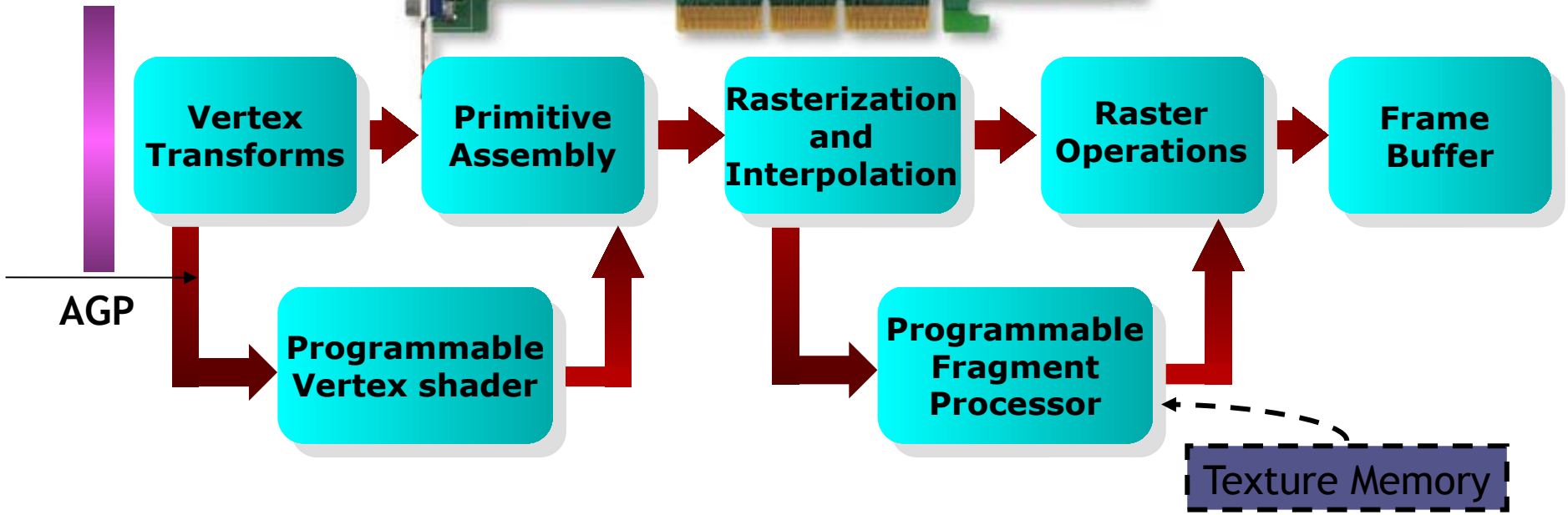


3dfx Voodoo (1996)
один из первых 3d ускорителей

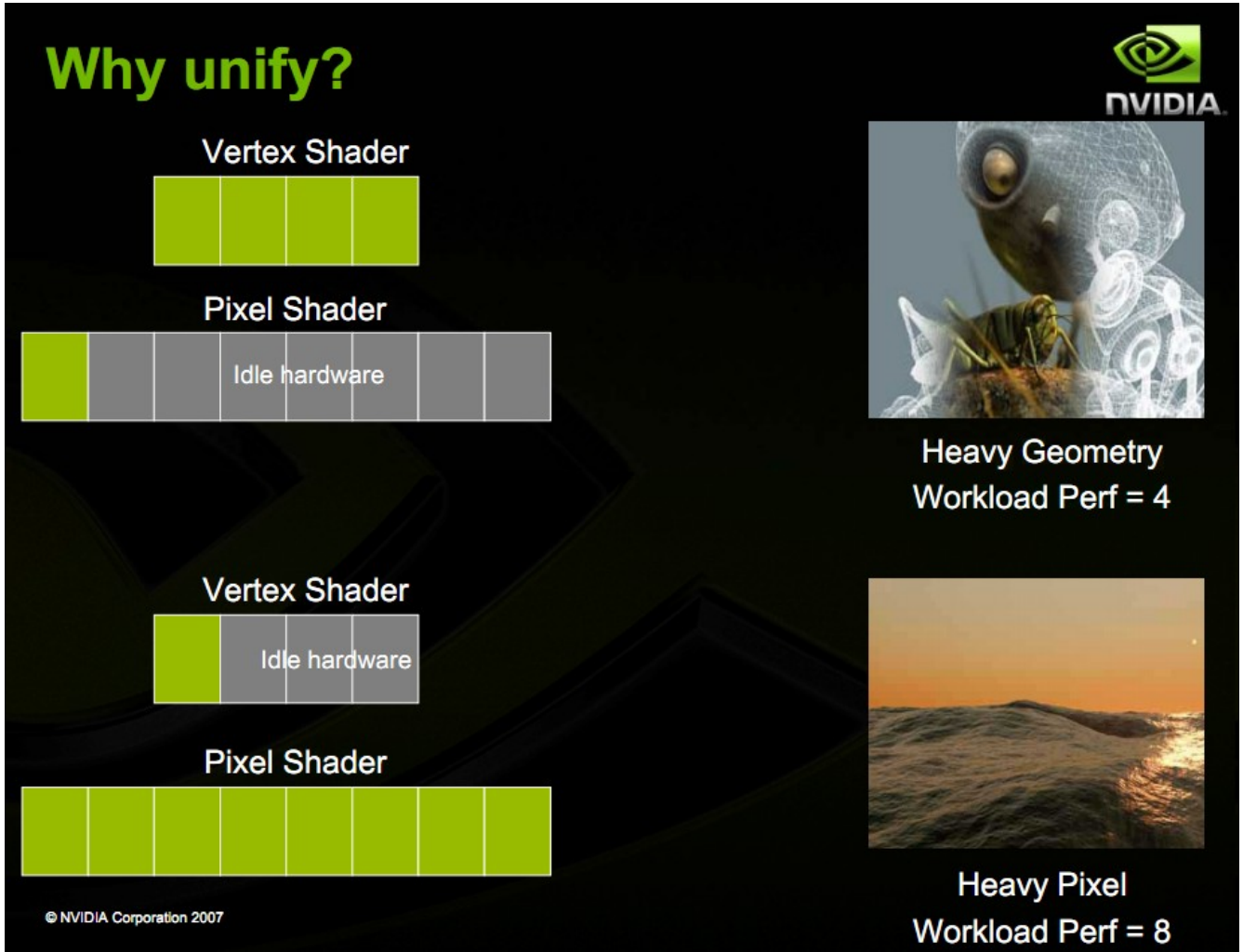




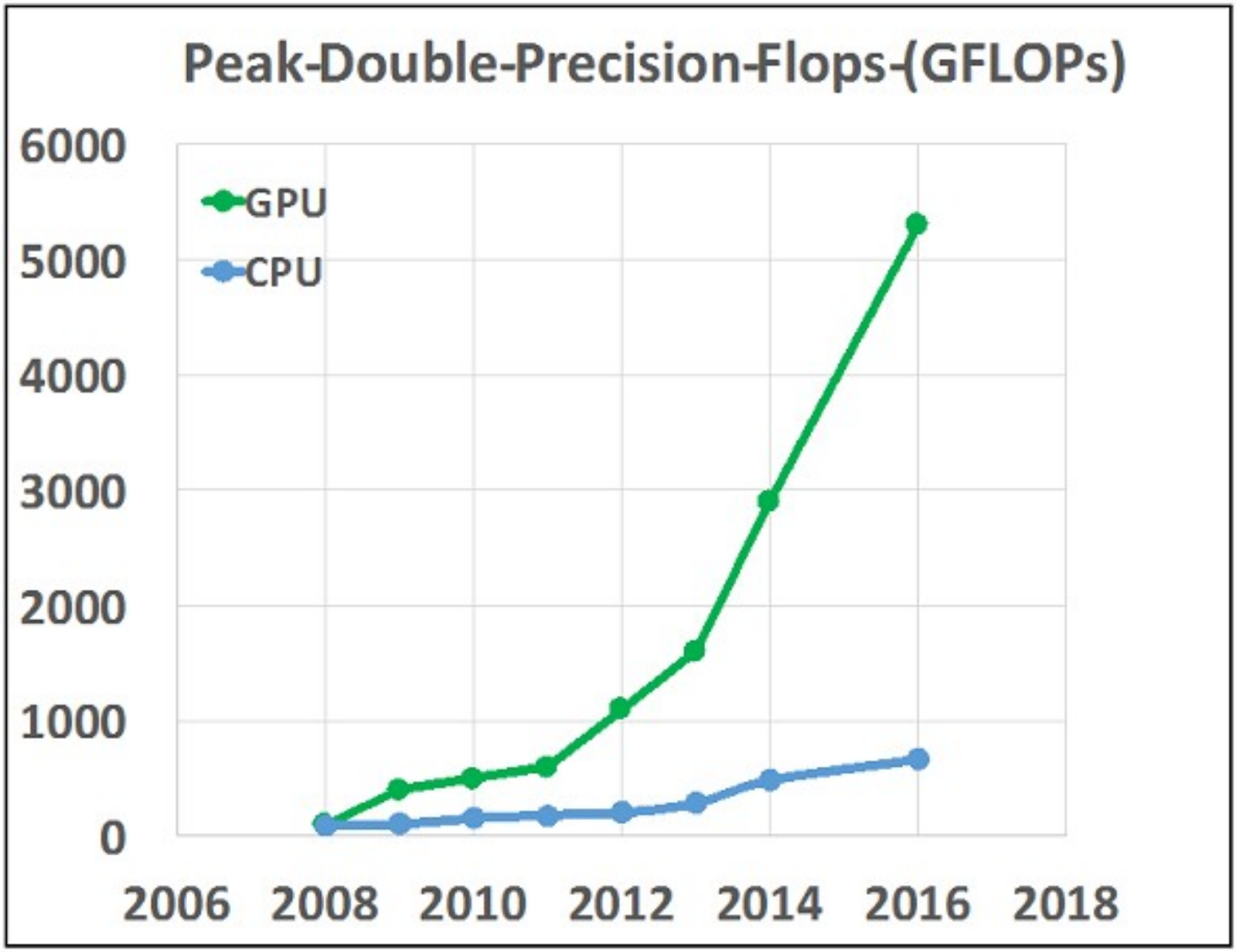
Radeon 9700/GeForce FX (2002) – первые программируемые видеокарты



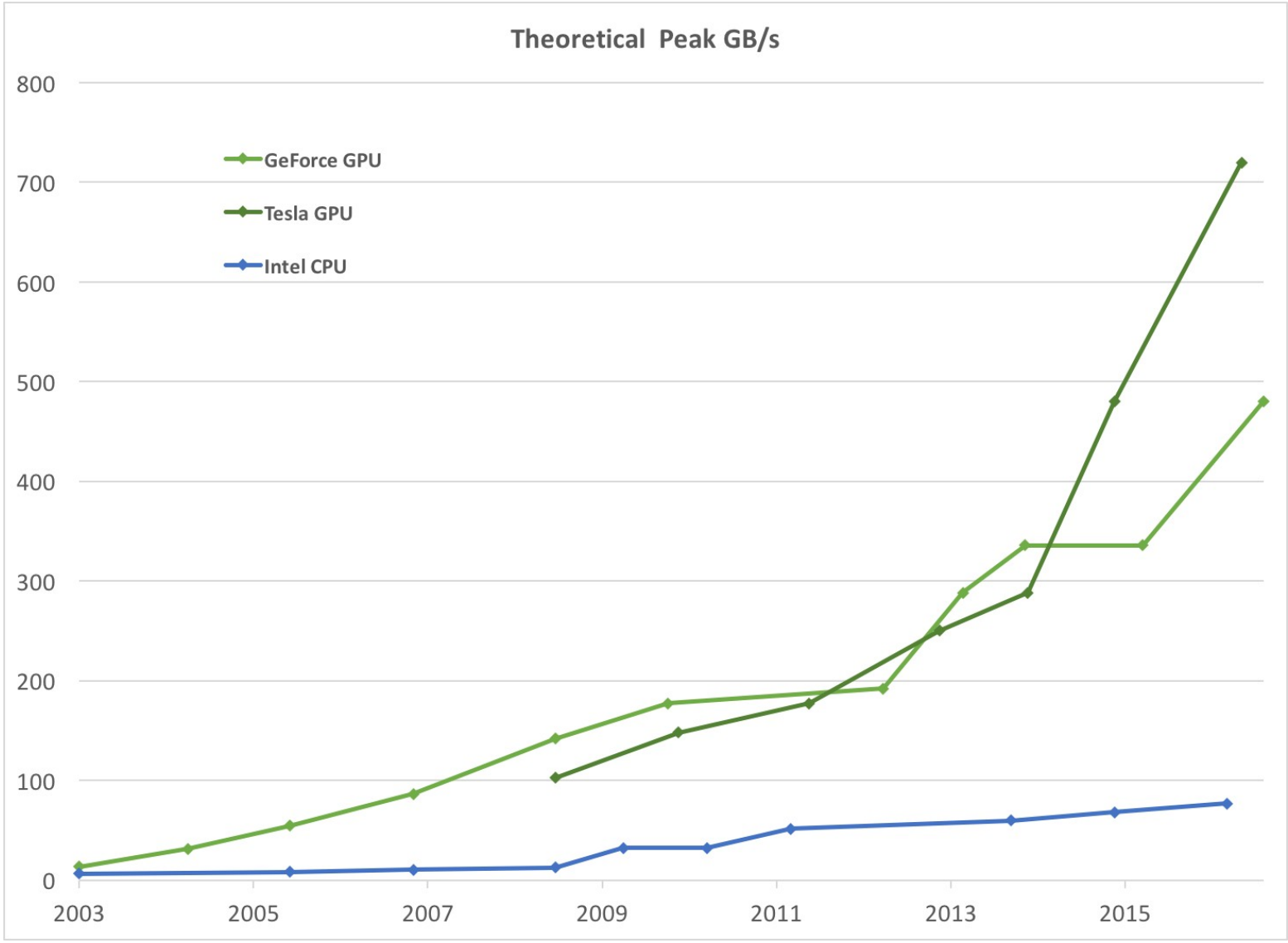
CUDA –
Compute Unified Device Architecture



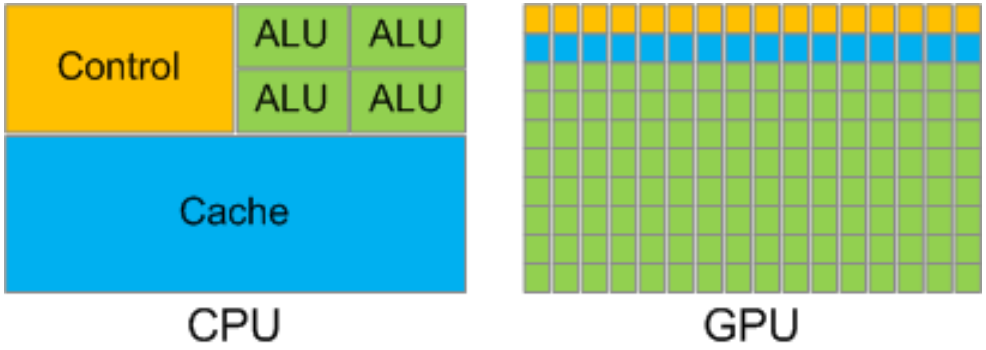
Вычислительная мощность



Пропускная способность памяти

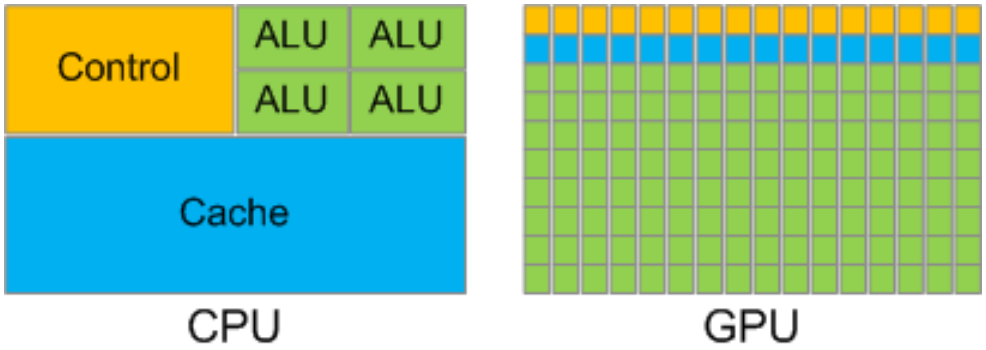


Сравнение GPU и CPU



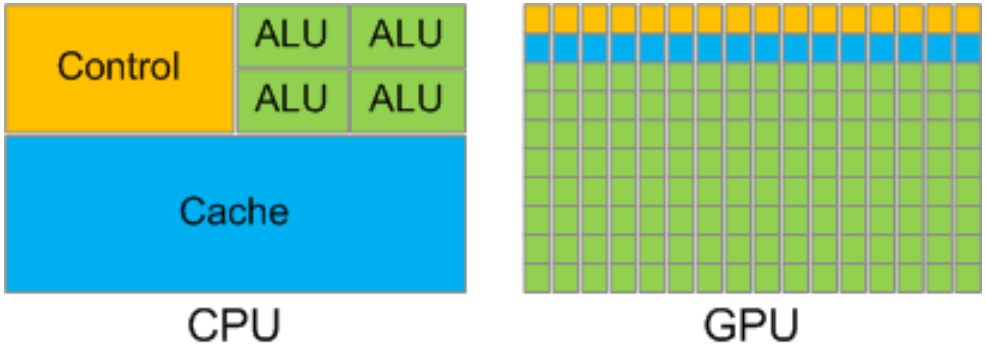
- Сотни упрощённых вычислительных ядер, работающих на небольшой тактовой частоте ~1.5ГГц (вместо 2-8 на CPU)
- Небольшие кеши
  - 32 ядра разделяют L1, с двумя режимами: 16KB или 48KB
  - L2 общий для всех ядер, 768 KB, L3 отсутствует
- Оперативная память с высокой пропускной способностью и высокой латентностью
  - Оптимизирована для коллективного доступа
- Поддержка миллионов виртуальных нитей, быстрое переключение контекста для групп нитей

Утилизация латентности памяти



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

Утилизация латентности памяти

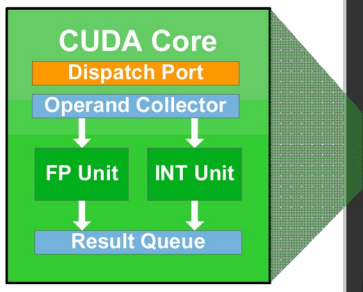


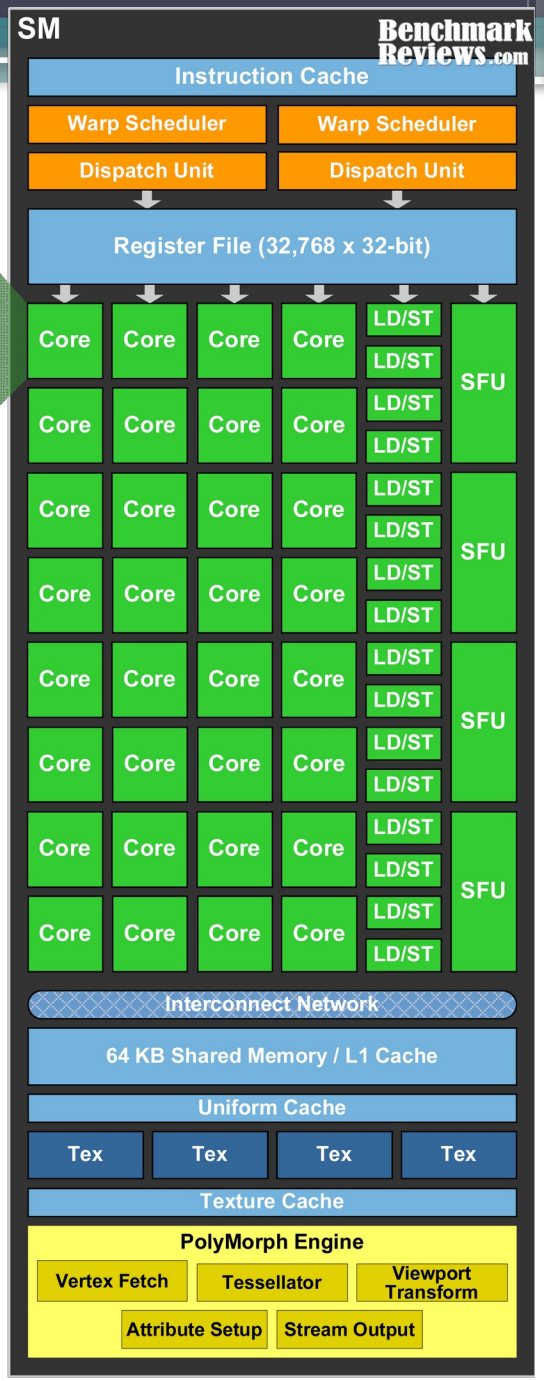
- GPU: Много нитей, покрывать обращения одних нитей в память вычислениями в других за счёт быстрого переключения контекста

- За счёт наличия сотен ядер и поддержки миллионов нитей (потребителей) на GPU легче утилизировать всю полосу пропускания !

Архитектура

Fermi: Streaming Multiprocessor (SM)





- Потоковый мультипроцессор
- «Единица» построения устройства (как ядро в CPU):
  - 32 скалярных ядра CUDA Core, ~1.5ГГц
  - 2 Warp Scheduler-а
  - Файл регистров, 128KB
  - 3 Кэша – текстурный, глобальный (L1), константный(uniform)
  - PolyMorphEngine – графический конвейер
  - Текстурные юниты
  - 16 x Special Function Unit (SFU) – интерполяция и трансцендентная математика одинарной точности
  - 16 x Load/Store

Fermi: Чип в максимальной конфигурации

- 16 SM
- 512 ядер CUDA Core
- Кеш L2 758KB
- GigaThreadEngine
- Контроллеры памяти DDR5
- Интерфейс PCI

Turing: SM

- 32 x DP Unit
- 64 x SFU, 64 x IU
- 8 Tensor Cores
- L0 кеш инструкций
- 32x load/store Unit
- 4 x warp scheduler
- 256KB регистров

Turing: Чип в максимальной конфигурации

84 SM = 5376 FP32 + 5376 INT32 + 2688 FP64 + 672 TC

Технологии программирования ускорителей

Шейдеры

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
   vec2 xy = fragCoord.xy / iResolution.xy;
   vec4 texColor = texture(iChannel0,xy);
   texColor.r *= abs(sin(iGlobalTime));
   texColor.g *= abs(cos(iGlobalTime));
   texColor.b *= abs(sin(iGlobalTime) *
                                        cos(iGlobalTime));
   fragColor = texColor;
}
```

Разнообразие языков программирования (приходилось писать компиляторы) →
SPIR-V (промежуточный универсальный язык, похожий на GLSL)

Другие технологии

- Close To Metal, AMD FireStream – разработаны AMD (устарели)
- OpenCL – открытый портабильный аналог CUDA, позволяет програм-мировать CPU, GPU, FPGA, DSP
- Android RenderScript (до Android 12)
- Vulkan (на смену OpenGL и Direct3D) — кросплатформенный, использует SPIR-V для прогр. шейдеров

Введение в технологию программирования CUDA

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

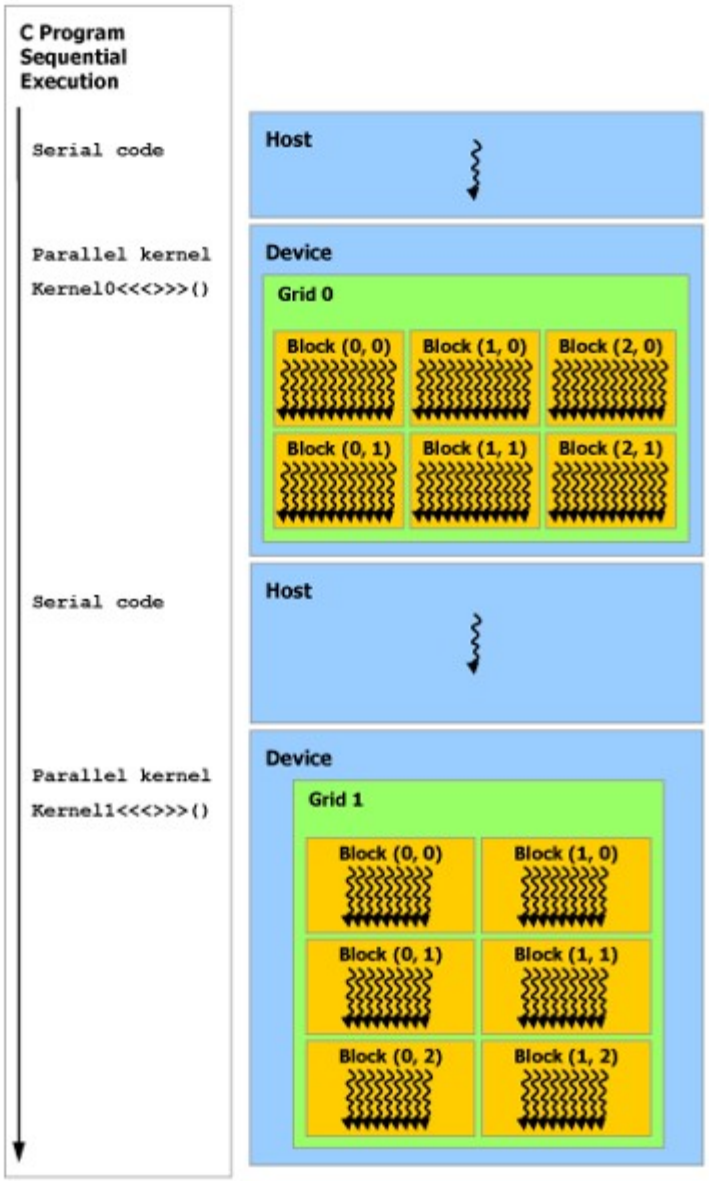
Вычисления с использованием GPU

- 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 программ

Терминология

- 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 хост-код, в котором содержатся вызовы функций, побочный эффект которых – манипуляции с устройством.



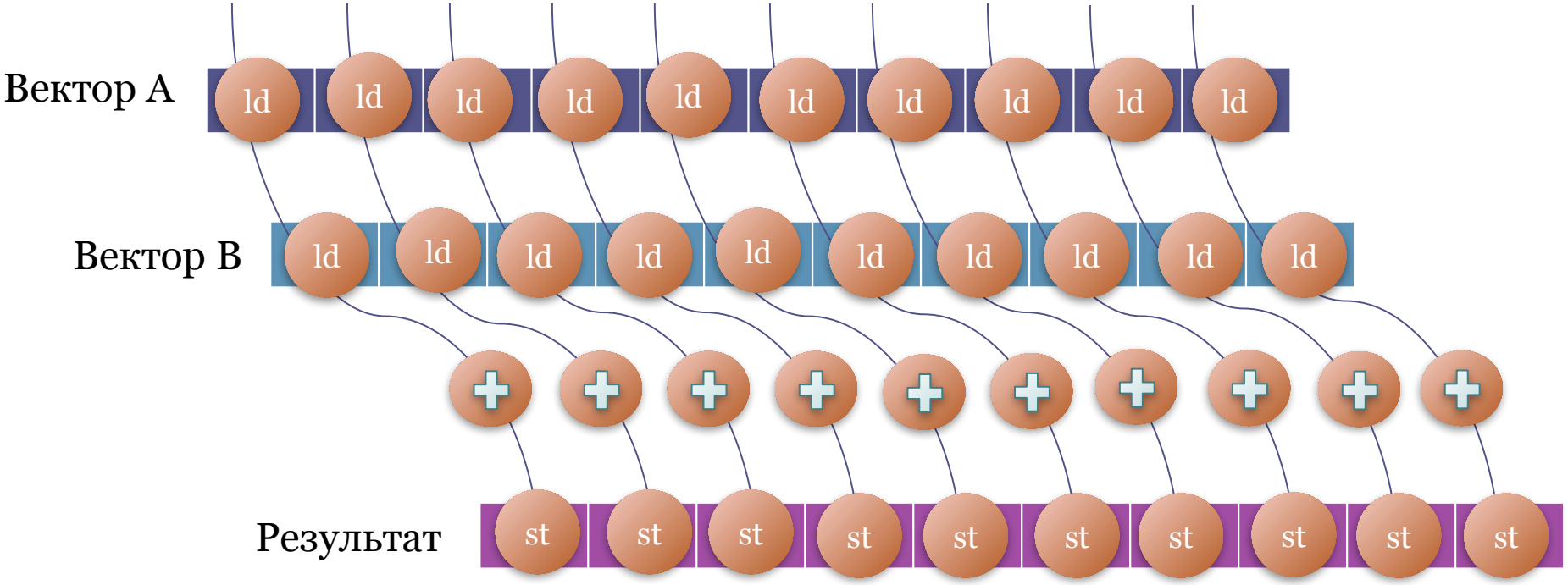
Код для GPU (device-code)

- 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, основанным на LLVM`

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

Пример: сложение векторов



Сложение векторов

- Без GPU:

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

- Используя GPU

```
{// на CPU:
   <Переслать данные с CPU на GPU>;
   <Запустить вычисления на N GPU-нитях>;
   <Скопировать результат с GPU на CPU>;
   }

   {// на GPU в нити с номером threadIndex:
      c[threadIndex] = a[theadIndex] +
   b[threadIndex];
   }
```

SPMD & CUDA

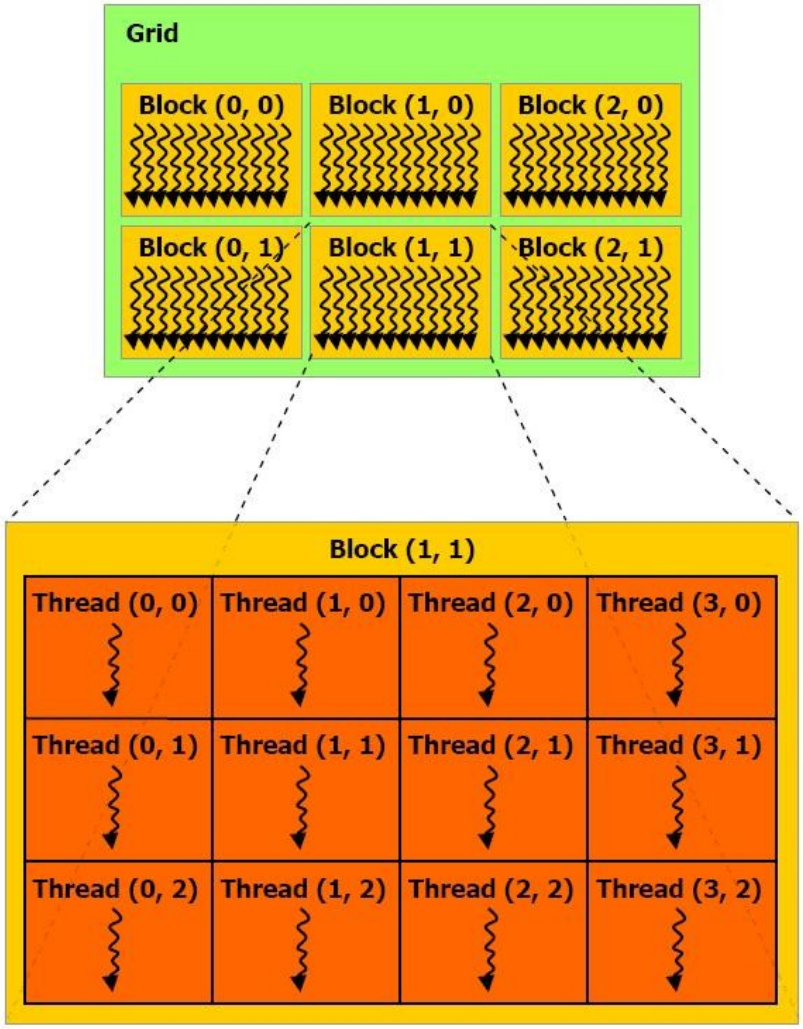
- GPU работает по методу **SPMD** - единая программа, множество данных
  - Задается программа (**CUDA kernel**)
  - Запускается множество нитей (**CUDA grid**)
  - Каждая нить выполняет копию программы над своими данными

CUDA Grid

- Хост может запускать на GPU множества виртуальных нитей
- Каждая нить приписана некоторому виртуальному блоку
- Грид (от англ. Grid-сетка) – множество блоков одинакового размера
- Положение нити в блоке и блока в гриде индексируются по трём измерениям (x,y,z)

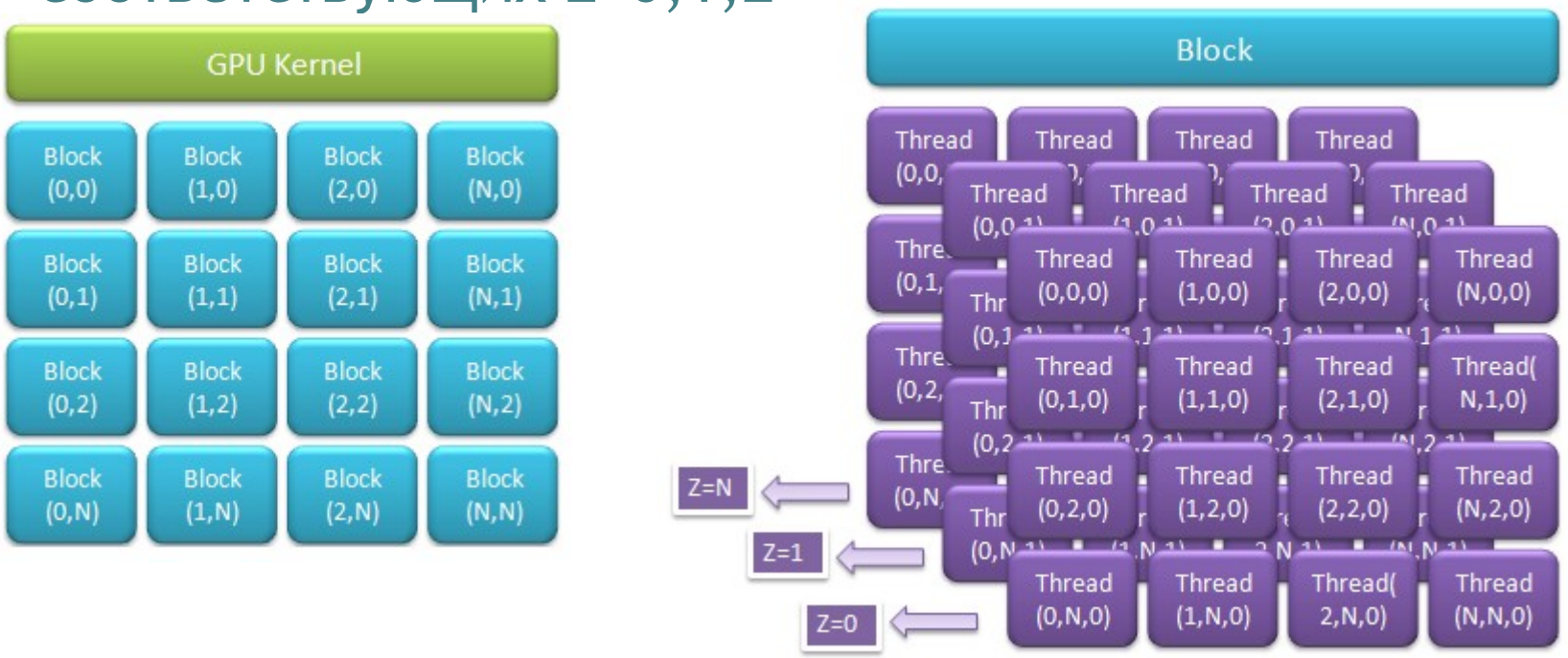
CUDA Grid

- Грид задаётся количеством блоков по x,y,z (размер грида в блоках) и размерами каждого блока по x,y,z
- Если по z размер грида и блоков равен единице, то получаем плоскую прямоугольную сетку нитей



CUDA Grid пример

- Двумерный грид 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



CUDA Kernel («Ядро»)

- Нити выполняют копии т.н. «ядер» - специально оформленных функций, компилируемых под GPU
  - Нет возвращаемого значения (void)
  - Атрибут **__global__**

```
__global__ void kernel (параметры) {
    код, исполняемый всеми
    ядрами видеокарты
}
```

Терминология

- Хост запускает вычисление ядра на гриде нитей
  - Иногда «на гриде нитей» опускается
- Одно и то же ядро может быть запущено на разных гридах

Запуск ядра

- **kernel<<< *execution configuration* >>>(params);**
  - **“kernel”** - имя ядра,
  - **“params”** - параметры ядра, копию которых получит каждая нить

- ***execution configuration:***

```
<<< dim3 gridDim, dim3 blockDim >>>
```

- `dim3` - структура, определённая в CUDA Toolkit

```
struct dim3 {
    unsigned x,y,z;
    dim3(unsigned vx=1, unsigned vy=1, unsigned
vz=1);
```

Запуск ядра

- kernel<<< *execution configuration* >>>(params);
  - **“kernel”** - имя ядра,
  - **“params”** - параметры ядра, копию которых получит каждая нить

- ***execution configuration:***

```
<<< dim3 gridDim, dim3 blockDim >>>
```

  - `dim3 gridDim` - размеры грида в блоках

    `число блоков = gridDim.x * gridDim.y * gridDim.z`

  - `dim3 blockDim` - размер каждого блока

    `число нитей в блоке = blockDim.x * blockDim.y * blockDim.z`

Запуск ядра

- Рассчитать грид:

```
dim3 blockDim = dim3(512);
gridDim = dim3(  (n + 512 - 1)  /  512 )
```

- Запустить ядро с именем “kernel”

```
kernel <<< gridDim, blockDim  >>>(…);
```

Ориентация нити в гриде

- Осуществляется за счёт встроенных переменных:

```
dim3 threadIdx - индексы нити в блоке
dim3 blockIdx  - индексы блока в гриде
dim3 blockDim  - размеры блоков в нитях
dim3 gridDim   - размеры грида в блоках
```

- Линейный индекс нити в гриде (для 1d-grid и 1d-block):

```
blockDim.x * blockIdx.x + threadIdx.x
```

Д/з:Написать линейные индексы для всех остальных случаев.

Сколько различных линейных индексов для случая 2d-grid и 2d-block ?

Пример: ядро сложения

```
__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
  - Рассчитывает свой элемент выходного массива

Host Code

- Выделить память на устройстве
- Переслать на устройство входные данные
- Рассчитать грид
  - Размер грида зависит от размера задачи
- Запустить вычисления на гриде
  - В конфигурации запуска указываем грид
- Переслать с устройства на хост результат

Выделение памяти на устройстве

- `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.
- Вызов `cudaMalloc(&p, N*sizeof(float))` соответствует вызову `p = malloc(N*sizeof(float));`

Копирование памяти

- `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

Пример кода хоста

```
int n = getSize(); // размер задачи
int nb = n * sizeof (float); // размер размер задачи в байтах
```

Приходится дублировать указатели для хоста и GPU

```
float *inputDataOnHost = getInputData(); // входные данные на хосте
float *resultOnHost = (float *)malloc( nb );
float *inputDataOnDevice = NULL, *resultOnDevice = NULL;
```

```
cudaMalloc( (void**)& inputDataOnDevice,  nb);
cudaMalloc( (void**)& resultOnDevice, nb);
```

Выделение памяти на GPU

Пример кода хоста

```
cudaMemcpy(inputDataOnDevice,inputDataOnHost,
                    nb, cudaMemcpyHostToDevice);
```

Копирование входных данных на GPU

```
dim3 blockDim = dim3(512);
dim3 gridDim = dim3((n + 512-1) / 512 );
kernel <<< gridDim, blockDim  >>> (inputDataOnDevice,
                                   resultOnDevice, n);
```

Запуск ядра

```
cudaMemcpy(resultOnHost, resultOnDevice,
          nb, cudaMemcpyDeviceToHost);
```

Копирование результата на хост

```
cudaFree(inputDataOnDevice);
cudaFree(resultOnDevice);
```

Освободить память

Задача

- Пусть матрица float* A; размера m x n хранится в памяти по строкам. Напишите индексное выражение и параметры конфигурации запуска ядра на видеокарте так, чтобы блок параллельных потоков обрабатывал столбец матрицы для случаев:
  a) m < 512, n < 65535
  b) m - любое, n < 65535
  c) m - любое, n - любое