Вычисления на GPU

План курса

- Введение в GPU и его роль в современных вычислениях
- Основы программирования на GPU
- Обзор фреймворков для работы с GPU
- Примеры кода на CUDA/OpenCL
- Резюме (???)

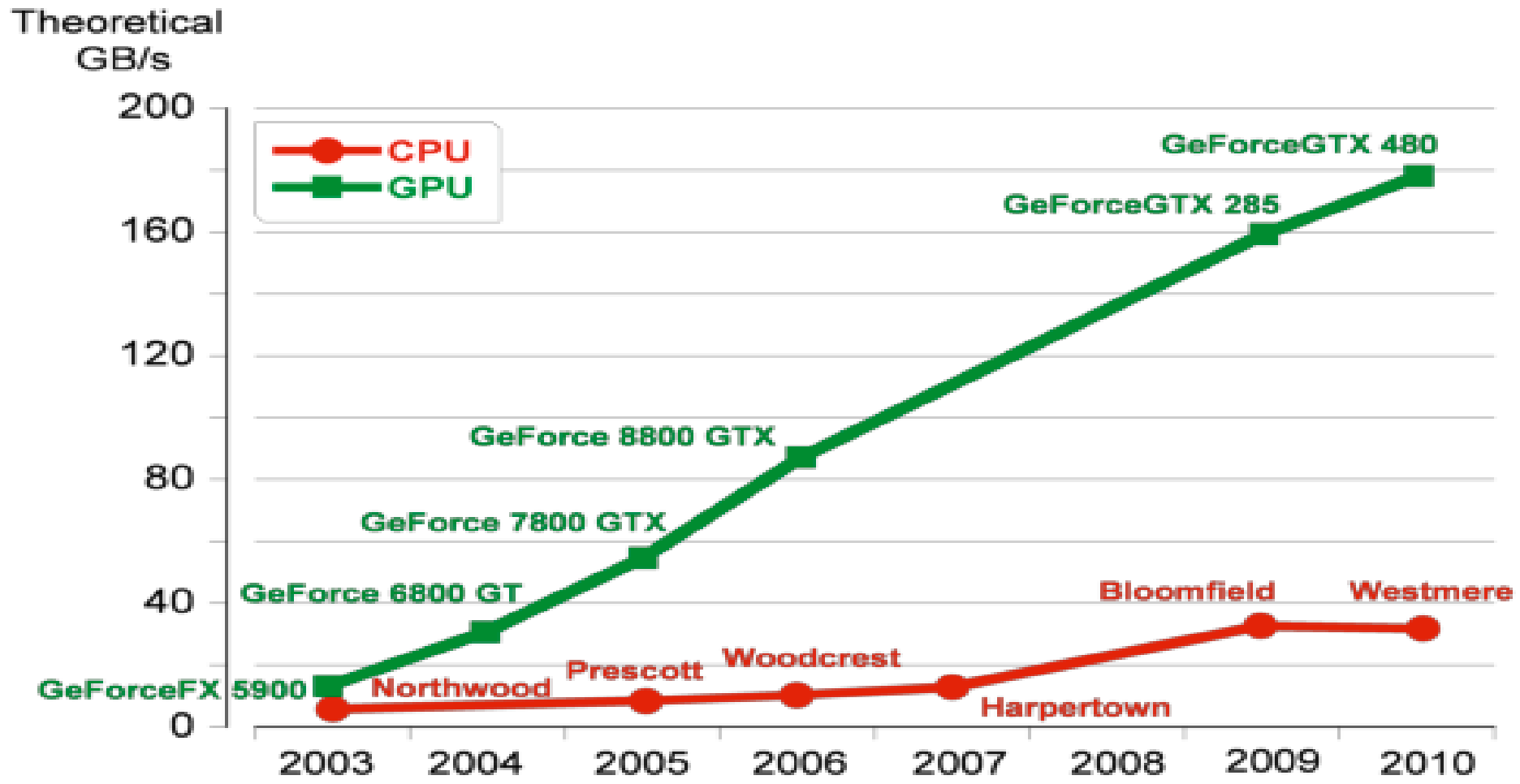
- А ЧЕ ПО ОЦЕНКЕ И ЗАДАЧКАМ

Что такое GPU?

- GPU (Graphics Processing Unit) — это специализированный процессор, изначально разработанный для обработки графики и визуализации.

- Современные GPU используются не только для рендеринга, но и для выполнения общих вычислений (GPGPU — General-Purpose computing on Graphics Processing Units).

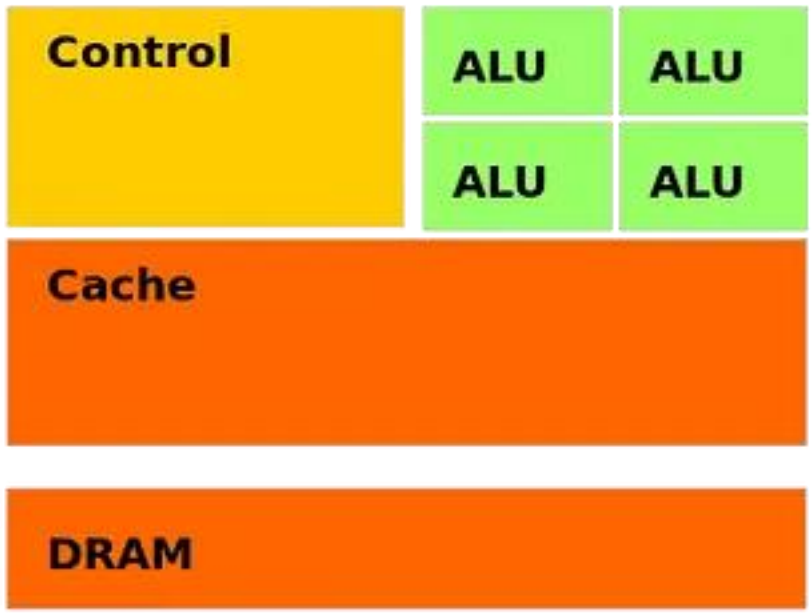
Но ведь есть CPU!



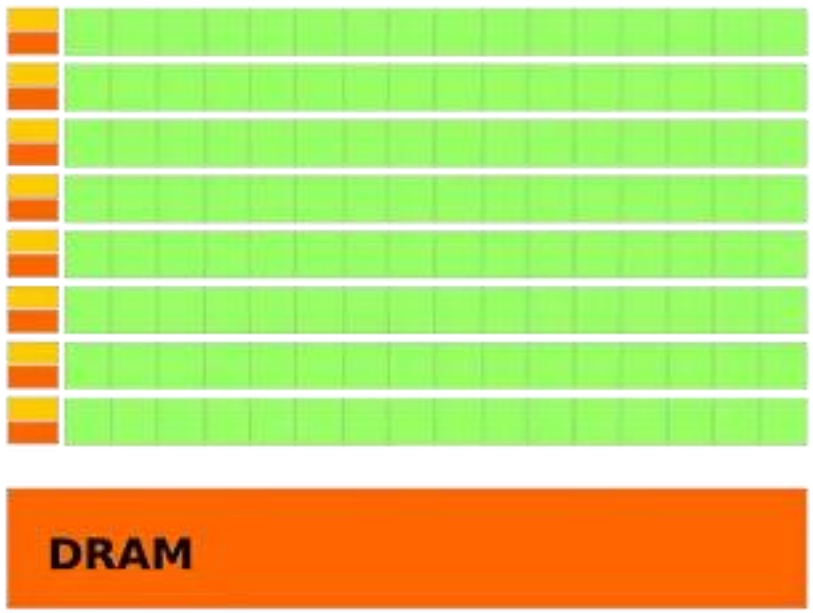
GPU vs. CPU

	GPU	CPU
Архитектура	**Тысячи более простых ядер**, оптимизированных для параллельных вычислений	**Несколько мощных ядер**, оптимизированных для последовательных задач
Параллелизм	Эффективен в задачах, где можно выполнять **множество одинаковых операций одновременно** (например, обработка пикселей, матричные операции)	Хорошо справляется с задачами, требующими **сложной логики и ветвления**
Производительность	Высокая производительность на **множестве потоков**, но только при условии высокой степени параллелизма	Высокая производительность на **одном потоке**

Архитектура



CPU



GPU

Зачем нам GPU??

- Машинное обучение и искусственный интеллект
- Научные вычисления
- Рендеринг и графика
- Криптография и блокчейн

Не все так однозначно…

Преимущества	Ограничения
Высокая производительность: GPU может выполнять тысячи операций одновременно, что делает его идеальным для задач с высокой степенью параллелизма	**Сложность программирования:** Работа с GPU требует понимания параллельной архитектуры и специфических фреймворков (CUDA, OpenCL)
Энергоэффективность: GPU потребляет меньше энергии на выполнение параллельных задач по сравнению с CPU	**Неэффективность для последовательных задач:** Если задача не может быть распараллелена, GPU не даст преимущества перед CPU
Масштабируемость: Современные GPU поддерживают тысячи потоков, что позволяет эффективно решать задачи любого масштаба	**Ограниченная память:** Глобальная память GPU меньше, чем у CPU, что может стать проблемой для задач с большими объемами данных

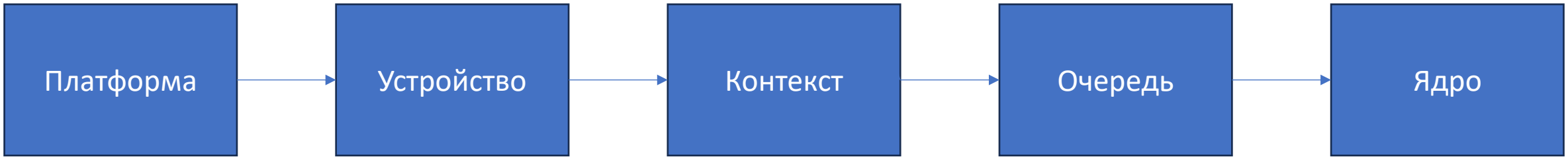
Обзор фрейморков для работы с GPU

- Фреймворки для работы с GPU — это наборы инструментов и библиотек, которые позволяют программистам использовать вычислительные мощности GPU для выполнения задач, отличных от обработки графики (GPGPU)
- Фреймворки упрощают взаимодействие с GPU, предоставляя высокоуровневые API для управления памятью, выполнения ядер (kernels) и синхронизации.
- Без фреймворков программирование на GPU было бы крайне сложным и требовало бы глубокого понимания аппаратной архитектуры.

OpenCL (Open Computing Language)

- OpenCL — это открытый стандарт для параллельных вычислений на расличных системах (CPU, GPU и других устройствах). Он разработан Khronos Group и поддерживается на устройствах от разных производителей (NVIDIA, AMD, Intel, ARM и др.).
- Основные концепции OpenCL включают платформу, устройство, контекст, очередь команд и ядро.

Основные концепции OpenCL



Абстракция для устройства или группы устройств (например, GPU NVIDIA или AMD)

Конкретное устройство, на котором выполняются вычисления (например, видеокарта)

Окружение, в котором выполняются команды (включает устройства, память и очереди команд)

Очередь задач, которые выполняются на устройстве

Функция, которая выполняется на GPU. Пишется на языке, похожем на C

CUDA

- CUDA — это проприетарная технология NVIDIA для параллельных вычислений на GPU
- Разработана исключительно для GPU NVIDIA, поэтому оптимизирована для GPU NVIDIA

Основные концепции CUDA

- Потоки (Threads): Базовая единица выполнения. Потоки объединяются в блоки.
- Блоки (Blocks): Группа потоков, которые могут взаимодействовать через shared memory.
- Сетки (Grids): Группа блоков, выполняющих одно ядро.

Основные концепции CUDA

- Растение – поток (thread)
- Ряд в грядке – **warp (32 threads)**
- Грядка – блок (block)

- Ряд из грядок – сетка (grid)

Основные концепции программирования на OpenCL и CUDA

- Код выполняется отдельно на **хосте (host)** и отдельно на **устройстве (device)**
- Создание “окружения” для работы кода (платформа, контекст, очереди и др.)
- **Выделение памяти** на хосте и устройстве, **копирование** данных с хоста на устройство
- **Выполнение ядра**
- **Копирование результатов** ядра на хост
- **Удаление** всех созданных сущностей, чтобы не было утечек памяти

Примеры кода на OpenCL (ядро)

```
__kernel void vector_add(__global const float *A, __global const float *B, __global float *C) {
    int id = get_global_id(0); // Получаем уникальный идентификатор потока
    C[id] = A[id] + B[id];    // Складываем элементы векторов
}
```

Примеры кода на OpenCL (хост)

```
#include <iostream>
#include <vector>
#include <CL/cl.hpp>

int main() {
    // Исходные данные
    std::vector<float> A = {1, 2, 3, 4, 5};
    std::vector<float> B = {10, 20, 30, 40, 50};
    std::vector<float> C(A.size());

    // Получаем платформы и устройства
    std::vector<cl::Platform> platforms;
    cl::Platform::get(&platforms);
    auto platform = platforms.front();
    std::vector<cl::Device> devices;
    platform.getDevices(CL_DEVICE_TYPE_GPU, &devices);
    auto device = devices.front();

    // Создаем контекст и очередь команд
    cl::Context context(device);
    cl::CommandQueue queue(context, device);

    // Выделяем память на GPU
    cl::Buffer bufferA(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(float) * A.size(),
A.data());
    cl::Buffer bufferB(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(float) * B.size(),
B.data());
    cl::Buffer bufferC(context, CL_MEM_WRITE_ONLY, sizeof(float) * C.size());
```

Примеры кода на OpenCL (хост)

```
    // Компилируем ядро
    const char* kernel_code = R"(
        __kernel void vector_add(__global const float *A, __global const float *B, __global float *C) {
            int id = get_global_id(0);
            C[id] = A[id] + B[id];
        }
    )";
    cl::Program program(context, kernel_code);
    program.build("-cl-std=CL1.2");

    // Создаем ядро и устанавливаем аргументы
    cl::Kernel kernel(program, "vector_add");
    kernel.setArg(0, bufferA);
    kernel.setArg(1, bufferB);
    kernel.setArg(2, bufferC);

    // Запускаем ядро
    queue.enqueueNDRangeKernel(kernel, cl::NullRange, cl::NDRange(A.size()), cl::NullRange);

    // Копируем результат обратно на CPU
    queue.enqueueReadBuffer(bufferC, CL_TRUE, 0, sizeof(float) * C.size(), C.data());

    // Выводим результат
    for (float c : C) {
        std::cout << c << " ";
    }
    return 0;
}
```

Примеры кода на CUDA (ядро)

```
__global__ void vector_add(const float *A, const float *B, float *C, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N) {
        C[idx] = A[idx] + B[idx];
    }
}
```

Примеры кода на CUDA (хост)

```
#include <iostream>
#include <vector>

__global__ void vector_add(const float *A, const float *B, float *C, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N) {
        C[idx] = A[idx] + B[idx];
    }
}

int main() {
    // Исходные данные
    int N = 5;
    std::vector<float> A = {1, 2, 3, 4, 5};
    std::vector<float> B = {10, 20, 30, 40, 50};
    std::vector<float> C(N);

    // Выделяем память на GPU
    float *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, N * sizeof(float));
    cudaMalloc(&d_B, N * sizeof(float));
    cudaMalloc(&d_C, N * sizeof(float));

    // Копируем данные на GPU
    cudaMemcpy(d_A, A.data(), N * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B.data(), N * sizeof(float), cudaMemcpyHostToDevice);
```

Примеры кода на CUDA (хост)

```
    // Запускаем ядро
    int blockSize = 256;
    int gridSize = (N + blockSize - 1) / blockSize;
    vector_add<<<gridSize, blockSize>>>(d_A, d_B, d_C, N);

    // Копируем результат обратно на CPU
    cudaMemcpy(C.data(), d_C, N * sizeof(float), cudaMemcpyDeviceToHost);

    // Освобождаем память на GPU
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Выводим результат
    for (float c : C) {
        std::cout << c << " ";
    }
    return 0;
}
```