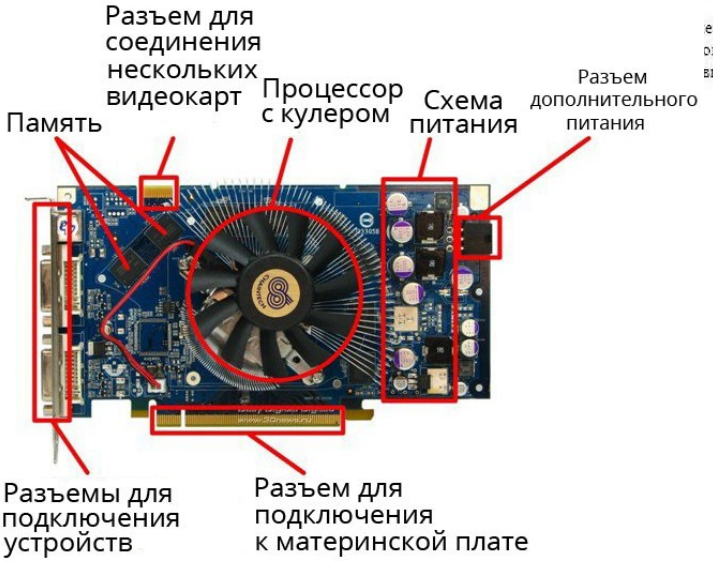
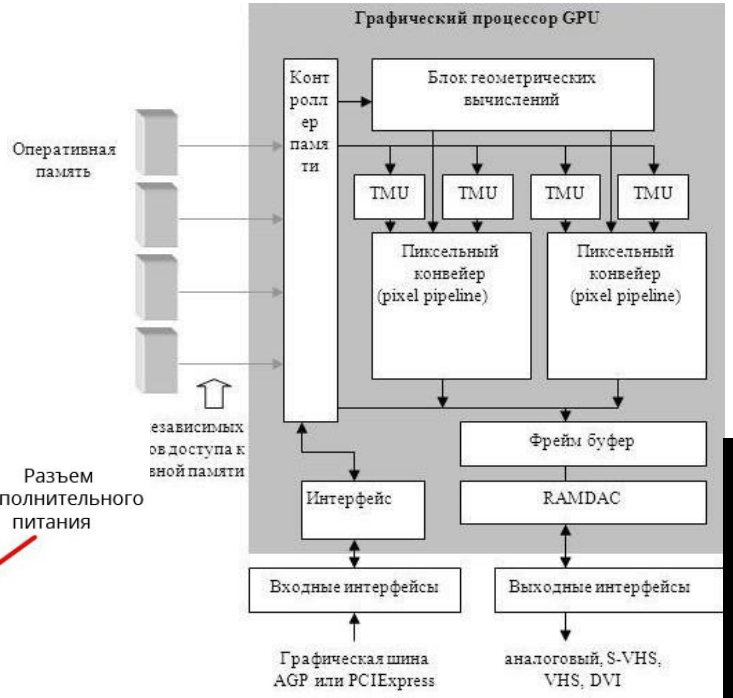
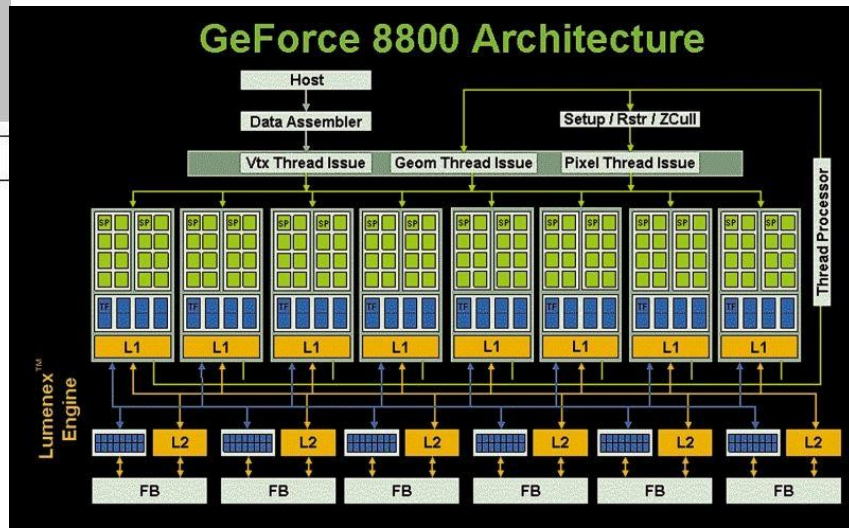
# Архитектуры параллельных ВС
## Graphics processing unit (GPU)



# *Определения*

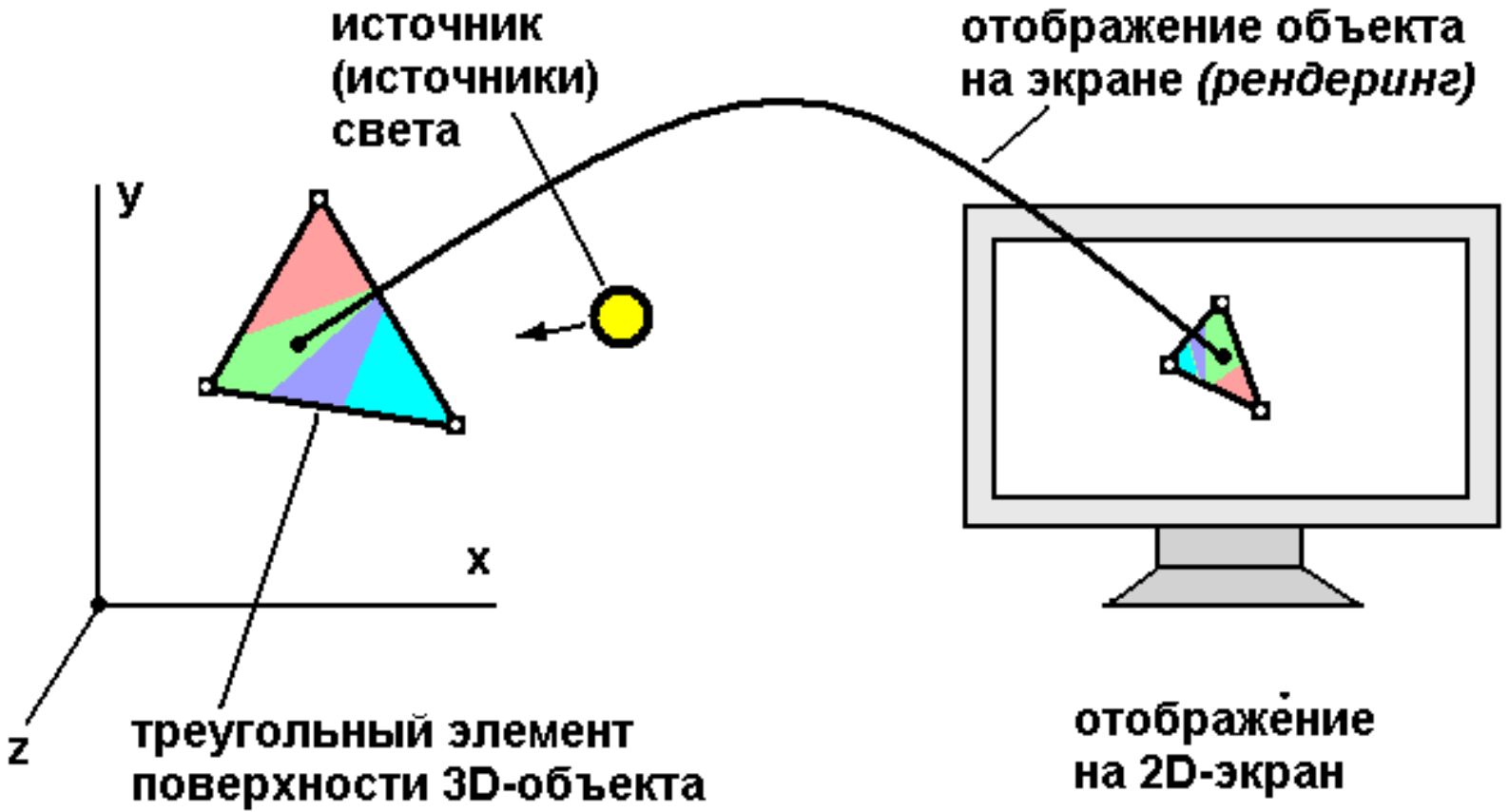
**Графический процессор (graphics processing unit, GPU)** — отдельное устройство, выполняющее графический рендеринг.

Отличительные особенности по сравнению с CPU:

- архитектура, максимально нацеленная на увеличение скорости расчёта текстур и сложных графических объектов;
- ограниченный набор команд.

Изначально создавался как многопоточная структура с множеством ядер.

# Основное назначение



# *Определения*

**Ре́ндеринг (rendering, “визуализация”)** - процесс получения 2D-изображения по некоторой модели с помощью компьютерной программы. Для представле́ния поверхностей 3D-объектов используется метод аппроксимации криволинейных поверхностей плоскими многоугольниками (обычно треугольниками).

**Ше́йдер (shader)** - программа для одной из ступеней графического конвейера, используемая в 3D-графике для определения окончательных параметров объекта или изображения. Верши́нный шейдер (vertex shader) оперирует данными, сопоставленными с вершинами многогранников. Пиксельный шейдер (pixel shader) работает с фрагментами изображения - пикселами, каждому из которых поставлен в соответствие набор атрибутов – в первую очередь цвет). Существуют шейдерные языки для описания поверхности объектов и рендеринга.

# *Определения*

Процесс рендеринга (построения 2D-объекта для визуализации) по сути параллельный (атрибуты отдельных пикселов независимы друг от друга) и осуществляется по одинаковым алгоритмам. Для аппаратного шейдинга известные фирмы (AMD, NVidia) выпускают графические карты.

Важным шагом стало осозна́ние того, что подобные методы эффективны не только при определении положения и цвета пикселов, но и для арифметических вычислений.

# *Определения*

**GPGPU (GPGP, GP²U, General-purpose computing on graphics processing units, неспециализированные вычисления на графических процессорах)** — техника использования графического процессора, предназначенного для компьютерной графики, в целях производства математических вычислений, которые обычно проводит центральный процессор.

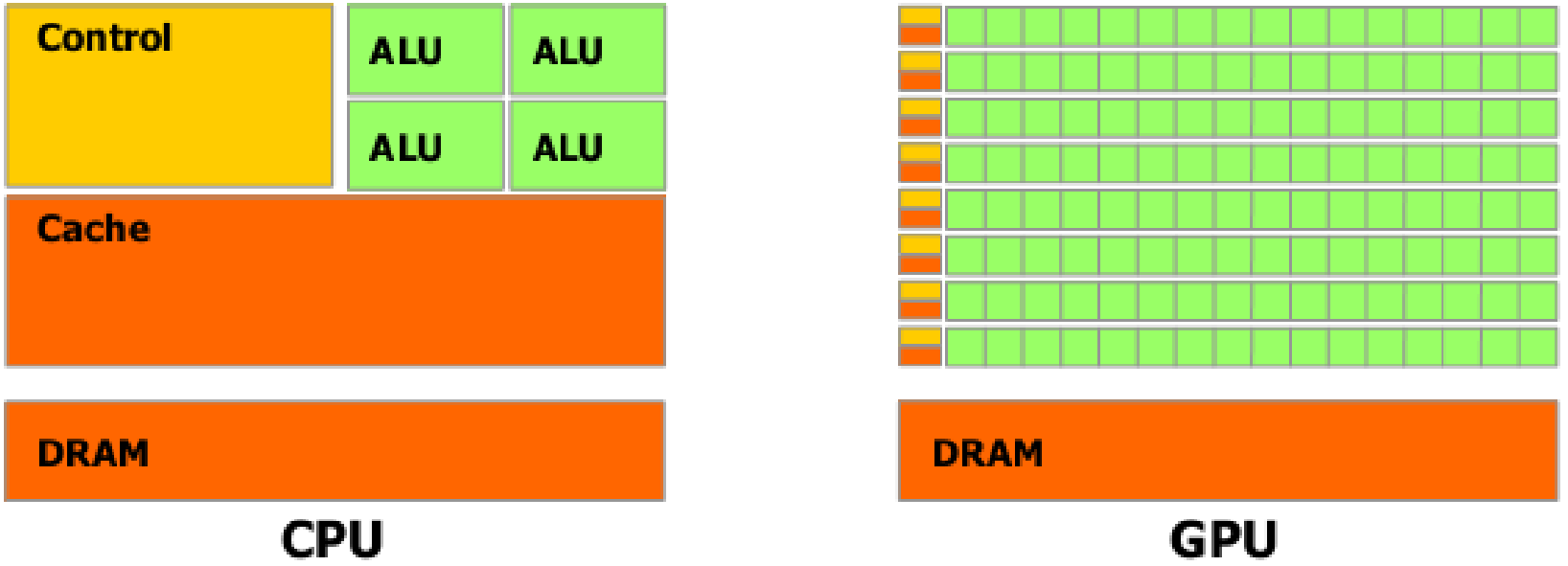
Стало возможным благодаря добавлению программируемых шейдерных блоков и более высокой арифметической точности растровых конвейеров, что позволяет разработчикам ПО использовать потоковые процессоры видеокарт для выполнения нерафических вычислений.

# *Определения*

**CUDA (Compute Unified Device Architecture)** — программно-аппаратная архитектура параллельных вычислений, которая позволяет существенно увеличить вычислительную производительность благодаря использованию графических процессоров фирмы Nvidia.

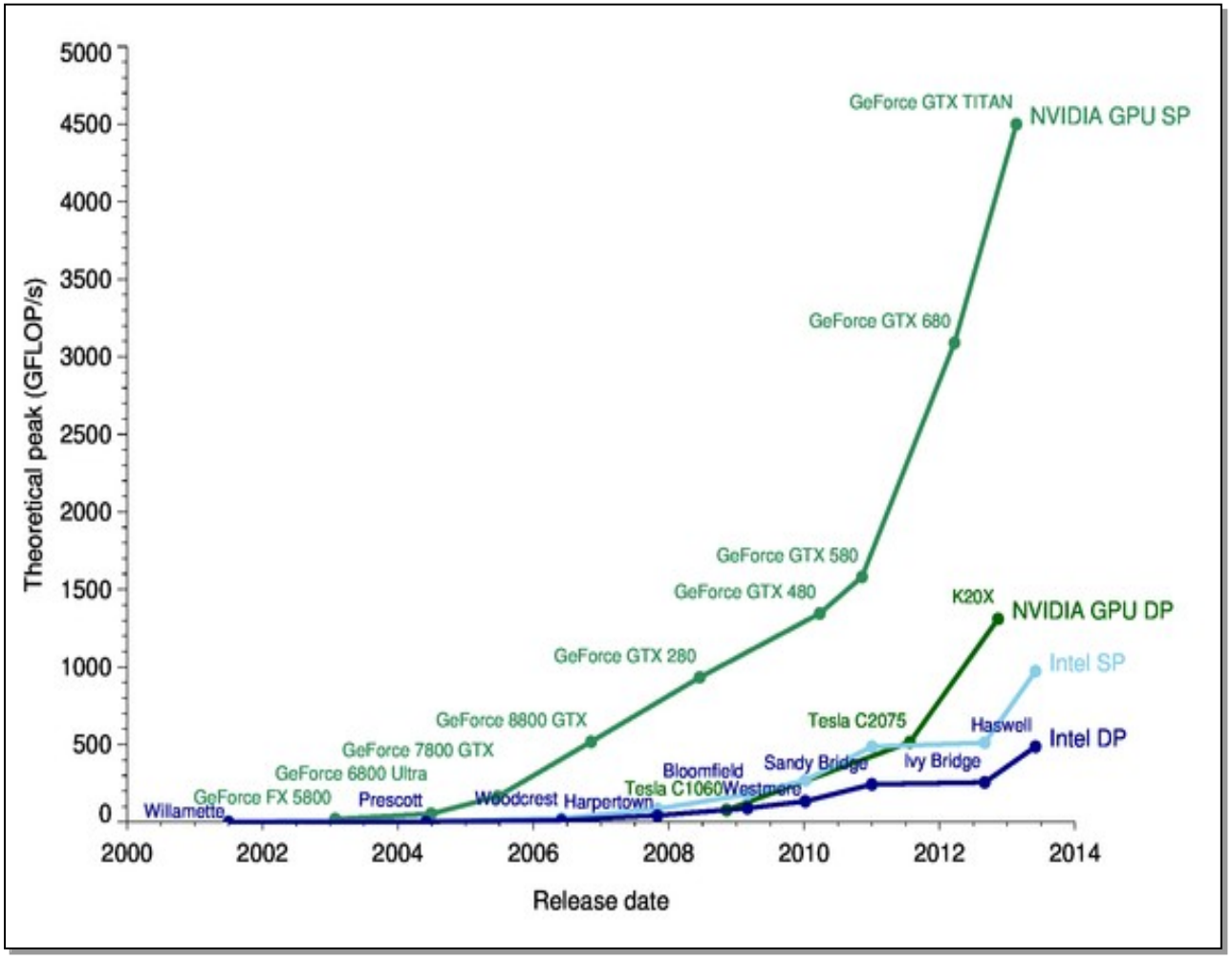
**CUDA SDK** позволяет программистам реализовывать на специальных упрощённых диалектах языков программирования Си, C++ и Фортран алгоритмы, выполнимые на графических и тензорных процессорах Nvidia. Архитектура CUDA даёт разработчику возможность по своему усмотрению организовывать доступ к набору инструкций графического или тензорного ускорителя и управлять его памятью. Функции, ускоренные при помощи CUDA, можно вызывать из различных языков, в т.ч. Python, MATLAB и др.

# Отличия в функциональном разделении ресурсов




**DRAM (англ. dynamic random access memory, динамическая память с произвольным доступом)** — тип компьютерной памяти, отличающийся использованием полупроводниковых материалов, энергозависимостью и возможностью доступа к данным, хранящимся в произвольных ячейках памяти.

# GPU vs CPU




Сравнение теоретической производительности CPU и GPU за период их развития с 2000 по 2014 год (производительность одной карты достигла нескольких Tflops).

Начиная с GT200 присутствует поддержка арифметики с плавающей точкой над числами типа double по IEEE 754.

Тепловыделение микросхем пропорционально четвёртой степени рабочей частоты, поэтому увеличение числа ядер всегда выгодно с точки зрения минимизации разогрéва процессора.

С использованием CUDA эффективно решаются задачи линейной алгебры, трассировки лучей при обсчете видеосцен, задачи моделирования в механике и физике (сеточными методами), проблемы прогнозирования природных катастроф (цунами, землетрясения), сложные задачи гидродинамики, моделирования динамики пучков заряженных частиц в ускорителях, реализации нейронных сетей для задач классификации, кластеризации, регрессии и прогнозирования.

# GPU Computing Applications

## Libraries and Middleware

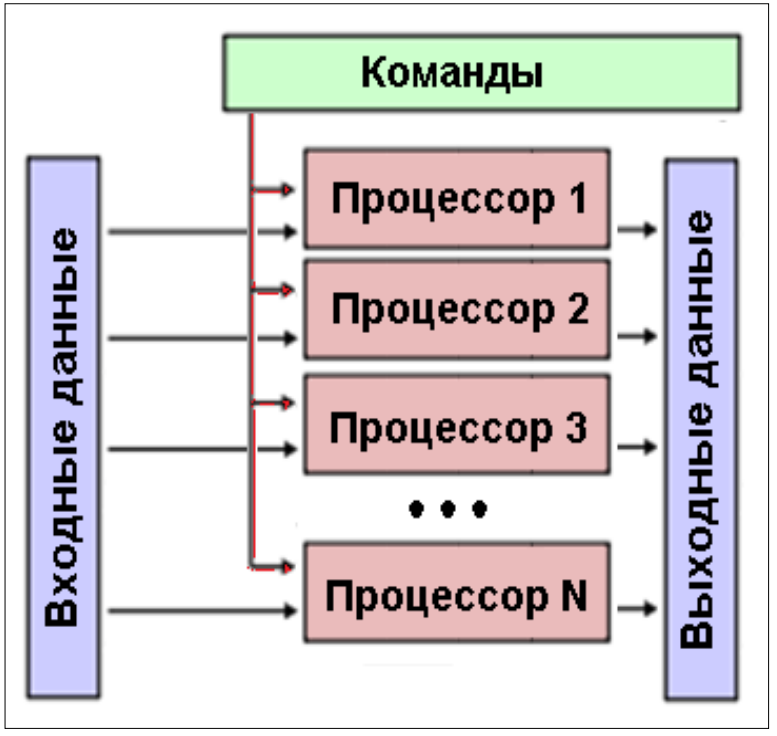
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| cuFFT | cuBLAS | CULA LAPACK | NPP & cuDPP | Video | PhysX Physics | OptiX Ray Tracing | mental ray iray Rendering | Reality Server 3D Web Services |

| C | C++ | OpenCL™ | Direct Compute | Fortran | Java and Python |
|---|---|---|---|---|---|

## NVIDIA GPU
with the **CUDA** Parallel Computing Architecture

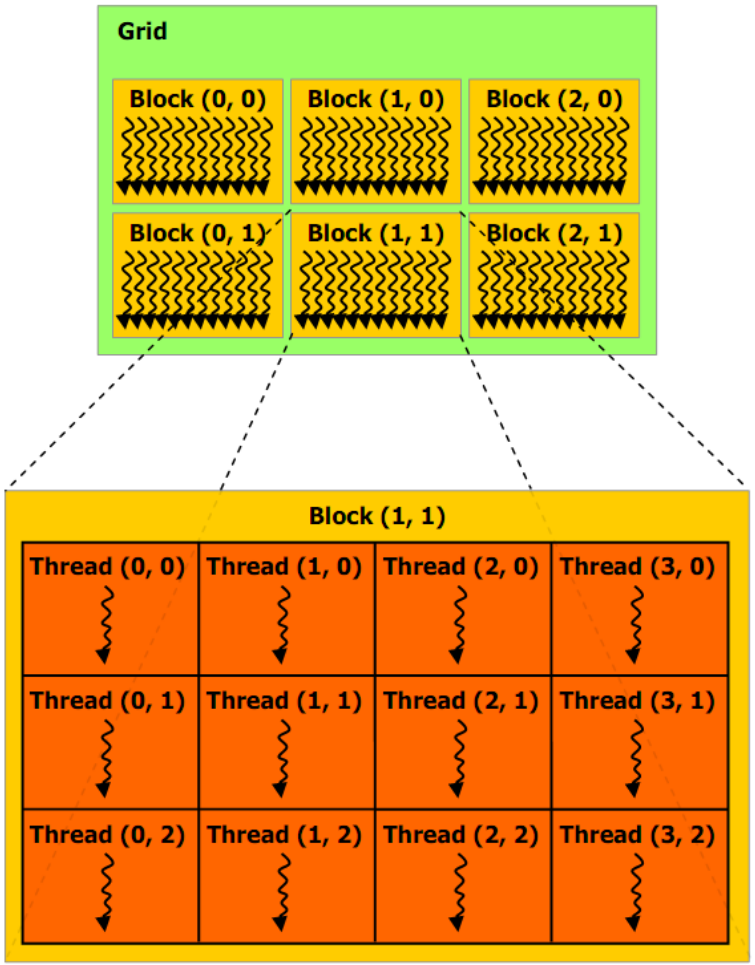
| | | | |
|---|---|---|---|
| **Fermi Architecture** (Compute capabilities 2.x) | GeForce 500 Series<br>GeForce 400 Series | Quadro Fermi Series | Tesla 20 Series |
| **Tesla Architecture** (Compute capabilities 1.x) | GeForce 200 Series<br>GeForce 9 Series<br>GeForce 8 Series | Quadro FX Series<br>Quadro Plex Series<br>Quadro NVS Series | Tesla 10 Series |
| |  |  |  |

# Технология CUDA (Compute Unified Device Architecture)


Устройство (device) – ма́ссово-параллельный (MPP) сопроцессор как дополне́ние к обычному CPU (host). Программа использует как CPU, так и GPU. Последовательная часть кода выполняется на CPU, а параллельная – на GPU в виде огромного (миллионы!) набора (только частично одновременно) выполняющихся ни́тей (threads - упрощённых пото́ков команд).

# Организация потоков в CUDA



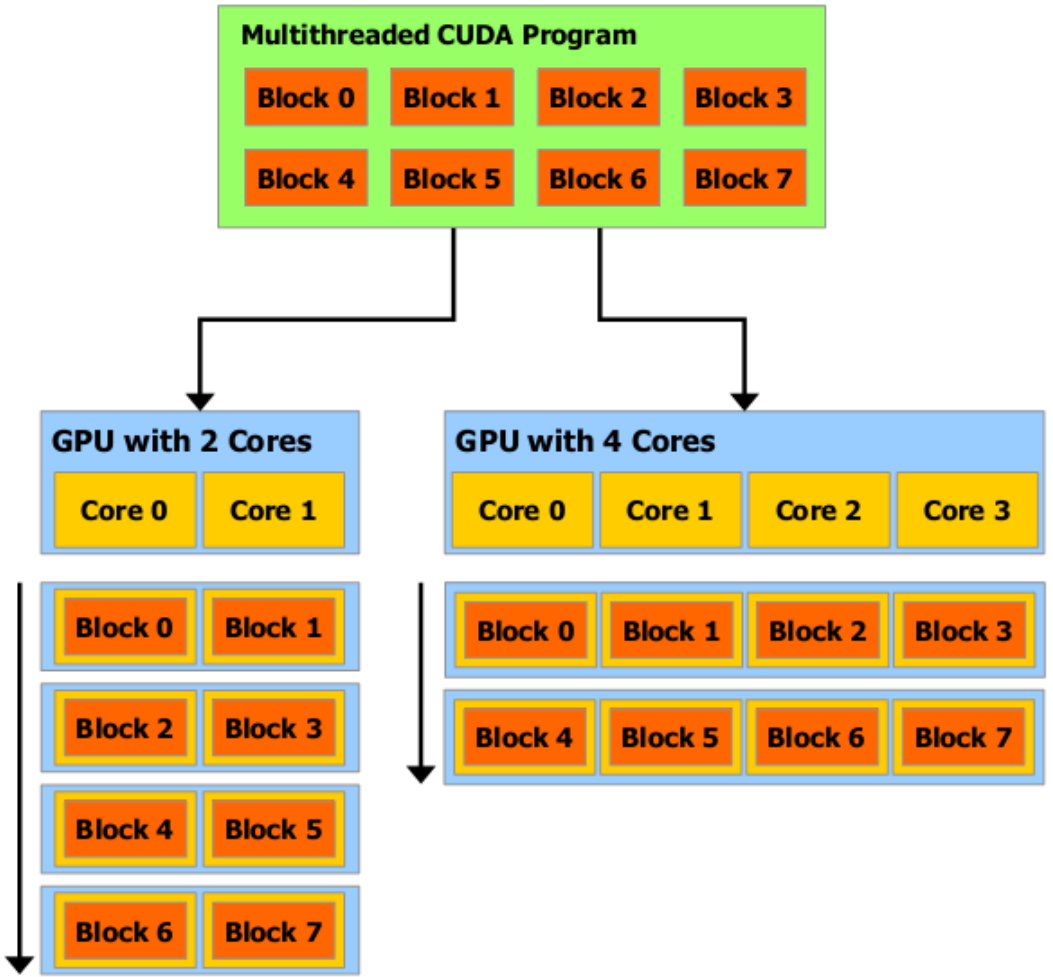

CUDA использует большое количество параллельно выполняющихся нитей (потоков). Каждой нити соответствует один блок (block) обрабатываемых данных.

Иерархия нитей:

- верхний уровень – сетка (grid) соответствует всем нитям, выполняющим программу. Сетка – одномерный или двухмерный массив из блоков;
- каждый блок – одномерный, двухмерный или трёхмерный массив нитей.

Все образующие сетку блоки имеют одинаковый размер. Каждый блок в сетке имеет свой адрес, состоящий из одного или двух неотрицательных целых чисел. Каждая нить внутри блока адресу́ется одним, двумя или тремя неотрицательными целыми числами.

# Распределение блоков по ядрам



# Решаемые задачи

*Табулирование функций от одной или многих переменных:*

- интегрирование (классическое или по методу Монте-Карло),
- вычисление производных и дифференциалов,
- поиск экстремумов функций.

*Статистика:*

- обработка сверхбольших объёмов данных,
- проверка статистических гипотез.

*Линейная алгебра* (есть BLAS – Basic Linear Algebra Subprograms).

*Анализ сигналов* (БПФ и др., FFT - Fast Fourier Transform).

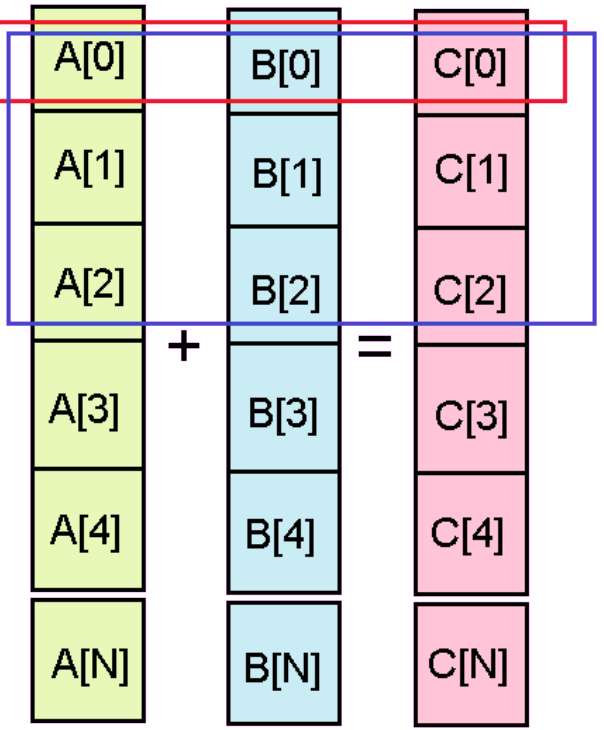
*Управление базами данных* (параллельная обработка запросов к таблицам БД).

Нейронные сети (существуют библиотеки поддержки сторонних фирм).

*…и многие другие задачи, характеризующиеся обработкой различных порций данных по единому алгоритму. Стало возможным благодаря добавлению программируемых шейдерных блоков и более высокой арифметической точности растровых конвейеров, что позволяет разработчикам ПО использовать потоковые процессоры видеокарт для выполнения неграфических вычислений.*

# Управление ресурсами GPU

1-й подход:
каждая нить (*thread*) обрабатывает (суммирует) один элемент массива A с элементом B и присваивает результат элементу C (индексы массивов одинаковы)




2-й подход:
каждая нить суммирует несколько элементов массивов A и B с присвоением результата элементам массива C

**Номер нити (*i_thread*) равен номеру индекса обрабатываемого элемента массивов A, B и C; инкремент номера нити = 1**

**Каждая нить обрабатывает элементы массива не по одному индексу, а по нескольким (на схеме слева - трём); инкремент номера нити > 1**

Все вычисления выполняются с помощью нитей (сотни тысяч и более).
Физически нити выполняются на вычислительных ядрах (сотни и более) графических карт.
Каждое ядро выполняет свой экземпляр одной и той же функции ядра.

# Программирование GPU

**Задача программиста** – распределить обработку исходных данных (практически всегда одно- , двух- или трёхмерные массивы) между нитями разумным образом.

Внутри функции ядра доступна (через предопределённые переменные) информация о номере нити, которая выполняет данный экземпляр кода.

Например, часто (внутри функции ядра) используется выражение для определения номера нити:

i_thread = threadIdx.x + blockIdx.x × blockDim.x

threadIdx.x, blockIdx.x и blockDim.x – индекс текущей нити в блоке, индекс текущего блока в сетке и размер блока соответственно
– все значения по координате x при двумерной индексации).

Т.к. все нити не могут выполняться синхронно, требуется применение принудительной синхронизации вызовом __synctreads(); При наличии операторов условного перехода внутри функции ядра нарушается принцип SIMD и ядро выполняет все ветви программы, выбирая нужный вариант в конце (при этом время выполнения увеличивается).

# Шаблон программы по технологии CUDA

```
#include <stdio.h>
#include <cuda.h>

__global__ void kernel(void) {
  // код программы ядра
}

int main(void) {
  // вызов программы ядра
  kernel<<<blocks,threads>>>();
  printf("It's end, fellows…");
  return 0;
}
```

**__global__** - объявленная функция (*ядро*) должна исполняться именно на **GPU**. В данном случае утилита **nvcc** передаст **пустую** функцию kernel() компилятору для GPU, а функцию main() – компилятору для CPU.
В угловых скобках **<<<blocks, threads>>>** располагаются параметры, передаваемые исполняющей среде GPU.
**blocks** – число параллельных экземпляров ядра (*блоков*),
**threads** – число нитей (потоков) в каждом блоке.
CUDA-программы (стандартное расширение файла .cu) компилируются с помощью внешнего компилятора
Исходный код для CPU и GPU разделяется:
- для CPU используется внешний компилятор,
- для GPU – специализированный компилятор **nvcc**).

При использовании арифметики плавающей запятой двойной точности (8 байт согласно IEEE 754) необходимо применять NVIDIA GPU с уровнем вычислительных возможностей (*Compute Capability*) не ниже **1.3**.

Получить информацию об имеющихся в системе GPU можно с помощью вызовов *cudaGetDeviceCount()* и *cudaGetDeviceProperties();* текущим GPU становится после выполнения *cudaSetDevice().*

# Пример. C = A + B

```
#include <iostream>
#include <cuda.h>

#define N 100 // размер массивов A[], B[], C[]
#define TYPE double // type of operands

// программа для GPU (device)
__global__ void add (TYPE *A,   TYPE *B, TYPE *C)   {
    int index = blockIdx.x; // обработка данных по по индексу блока x
    if (index < N)  {
        C[index] = A[index] + B[index];
    }
}
// конец программы для GPU (device)
```

# Пример. C = A + B

```
// код для CPU (host)
int main() {
    TYPE A[N], B[N], C[N], *dev_A, *dev_B, *dev_C;

    // выделить память на GPU
    cudaMalloc((void**)&dev_A, N*sizeof(TYPE));
    cudaMalloc((void**)&dev_B, N*sizeof(TYPE));
    cudaMalloc((void**)&dev_C, N*sizeof(TYPE));

    // инициализируем массивы A и B на CPU
    for (int i=0; i<N; i++) {
        A[i] = -i;
        B[i] = i*i;
    }
```

# Пример. C = A + B

```
// копируем массивы [A] и [B] с CPU на GPU
cudaMemcpy(dev_A, A, N*sizeof(TYPE), cudaMemcpyHostToDevice);
cudaMemcpy(dev_B, B, N*sizeof(TYPE), cudaMemcpyHostToDevice);
cudaMemcpy(dev_C, C, N*sizeof(TYPE), cudaMemcpyHostToDevice);

// задéйствуем N ядер (блоков) по одной нити в каждом
add<<<N,1>>>(dev_A,dev_B,dev_C);

// копируем массив  [C] с GPU на CPU
cudaMemcpy(C, dev_C, N*sizeof(TYPE), cudaMemcpyDeviceToHost);
```

# Пример. C = A + B

```
    // распечатываем результат с CPU
    for (int i=0; i<N; i++) {
        std::cout << A[i] << " + " << B[i]
                  << " = " << C[i] << "\n";
    }

    // освобождаем память на GPU
    cudaFree( dev_A );
    cudaFree( dev_B );
    cudaFree( dev_C );

    return 0;
}
// конец программы для CPU (host)
```

# CUDA и суперкомпьютеры

| Позиция | Rmax Rpeak (PFLOPS) | Название | Модель | Процессор | Сеть | Производитель | Размещение страна, год |
|---|---|---|---|---|---|---|---|
| 1 ▲ | 415.53 513.855 | Фугаку | Фугаку | Fujitsu A64FX[en] | Tofu interconnect D | Fujitsu | Институт физико-химических исследований Япония, 2020 |
| 2 ▼ | 148.6 200.795 | Summit | IBM Power Systems AC922 | POWER9, Tesla V100 | Infiniband EDR | IBM | Ок-Риджская национальная лаборатория США, 2018 |
| 3 ▼ | 94.64 125.712 | Sierra[20] | IBM Power Systems S922LC | POWER9, Tesla V100 | Infiniband EDR | IBM | Ливерморская национальная лаборатория США, 2018 |
| 4 ▼ | 93.015 125.436 | Sunway TaihuLight | Sunway MPP | SW26010 | Sunway[21] | NRCPC | National Supercomputing Center in Wuxi Китай, 2016[21] |
| 5 ▼ | 61.445 100.679 | Tianhe-2A | TH-IVB-FEP | Xeon E5-2692 v2, Matrix-2000 | TH Express-2 | NUDT | National Supercomputer Center in Guangzhou[en] Китай, 2013 |
| 6 ▲ | 35.45 51.721 | HPC5 | Dell | Xeon Gold 6252, Tesla V100 | Mellanox InfiniBand EDR | EMC | Eni Италия, 2020 |
| 7 ▲ | 27.58 34.569 | Selene | Nvidia | EPYC 7742, Ampere A100[en] | Mellanox InfiniBand EDR | Nvidia | Nvidia США, 2020 |
| 8 ▼ | 23,516 38,746 | Frontera | Dell C6420 | Xeon Platinum 8280[en], POWER9 | InfiniBand HDR | EMC | Texas Advanced Computing Center[en] , 2019 |
| 9 ▲ | 21.64 29.354 | Marconi-100 | IBM Power Systems AC922 | POWER9, Volta V100[en] | Dual-rail Mellanox EDR Infiniband | IBM | CINECA[en] Италия, 2020 |
| 10 ▼ | 21.23 27.154 | Piz Daint | Cray XC50[en] | Xeon E5-2690 v3, Tesla P100 | Aries | Cray | Swiss National Supercomputing Centre[en] , 2016 |

# *Используемые источники*

1. Руководство по программированию NVIDIA CUDA C (NVIDIA CUDA C Programming Guide) - http://masters.donntu.org/2013/fknt/vodolazskiy/library/translate1.htm

2. Cuda - основы, примеры - https://ksmlab.ru/page/cuda-osnovy-primery

3. Википедияя. Графический процессор - https://ru.wikipedia.org/wiki/Графический_процессор

4. Блог программиста. Особенности архитектуры и программирования графических ускорителей - https://pro-prof.com/forums/topic/architecture-and-programming-of-graphics-accelerators