

# Технология CUDA для высокопроизводительных вычислений на кластерах с графическими процессорами

Колганов Александр  
[alexander.k.s@mail.ru](mailto:alexander.k.s@mail.ru)

часть 1

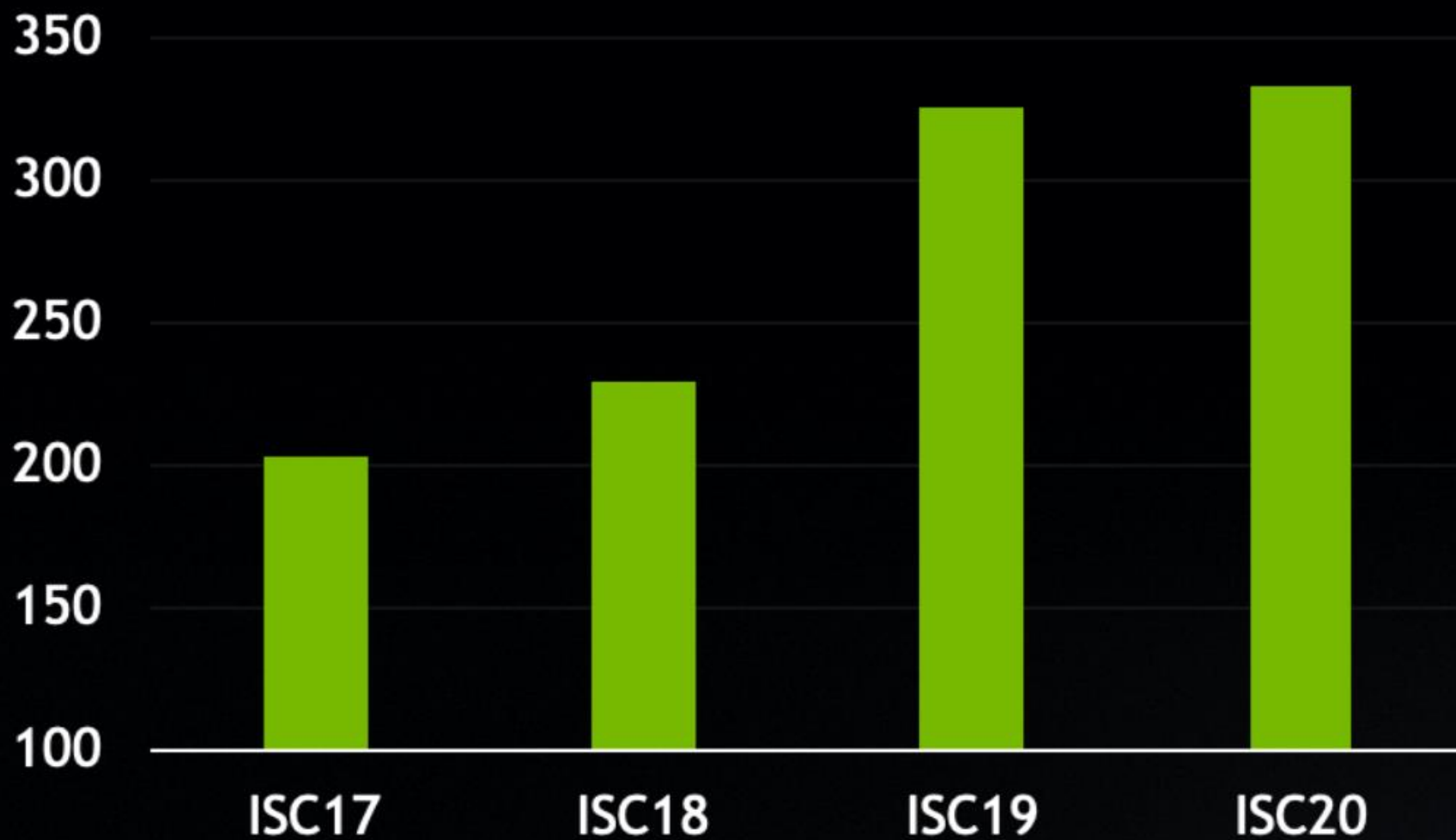


# Введение

# GPGPU и CUDA

- GPU - *Graphics Processing Unit*
- GPGPU - *General-Purpose* computing on GPU, вычисления общего вида на GPU;  
Первые GPU от NVIDIA с поддержкой GPGPU – GeForce восьмого поколения, G80 (2006 г);
- CUDA - *Compute Unified Device Architecture*, Программно-аппаратная архитектура от Nvidia, позволяющая производить вычисления с использованием графических процессоров

## NVIDIA Accelerates 66% of Top500 Systems



# Основные преимущества GPU по сравнению с CPU

- Высокое соотношение  
цена / производительность
- Высокое соотношение  
производительность / энергопотребление

# Рейтинг Green500 (www.top500.org)

## 2017 год

TOP500 Rank	System	Cores	Pflops /s	Power KW	Gflops /W
61	Xeon E5-2680v4, NVIDIA Tesla P100	36 288	1,9	142	14.1
465	Xeon E5-2650Lv4, NVIDIA Tesla P100	10 080	0,46	33	14.04
148	Xeon E5-2630Lv4, NVIDIA Tesla P100	23 400	0,9	76	12.6
305	NVIDIA DGX-1 Tesla P100, Fujitsu	11 712	0,6	60	10.6
100	Xeon E5-2650v4, NVIDIA Tesla P100	21 240	1,19	114	10.4
3	Xeon E5-2690v3, NVIDIA Tesla P100, Cray Inc.	361 760	16,5	2 272	10.39
69	Xeon D-1571 16C 1.3GHz, PEZY-SC2	3 176 000	1,6	164	10.22
220	Xeon E5-2650v4, NVIDIA Tesla P100	16 320	0,77	79	9.79
31	NVIDIA DGX-1 Tesla P100, Facebook	60 512	3,3	350	9.46
32	NVIDIA DGX-1 Tesla P100, Nvidia	60 512	3,3	350	9.46

# Рейтинг Green500 (www.top500.org)

## 2018 год

TOP500 Rank	System	Cores	Tflops/s	Power KW	Gflops/W
359	Xeon D-1571, PEZY-SC2	794 400	857	47	18.404
419	Xeon D-1571, PEZY-SC2	762 624	798	47	16.835
385	Xeon E5-2618Lv3, PEZY-SC2	794 400	824	50	16.657
227	Xeon E5-2698v4, NVIDIA Tesla V100	22 440	1 070	97	15.113
1	<b>SUMMIT IBM P9, NVIDIA Tesla V100</b>	2 282 544	122,300	8 806	13.889
19	Xeon E5-2680v4, NVIDIA Tesla P100	135 828	8,125	792	13.704
287	Xeon E5-2630Lv4, NVIDIA Tesla P100	23 400	961	76	12.6
5	Xeon Gold, NVIDIA Tesla V100	391 680	19,880	1,649	12.054
255	IBM P9, NVIDIA Tesla V100	19 440	1,018	86	11.865
171	NVIDIA DGX-1 Tesla V100	35 360	1,213	107	11.363

# Рейтинг Green500 ([www.top500.org](http://www.top500.org))

## 2022 год

TOP500 Rank	System	Cores	Tflops/s	Power KW	Gflops/W
29	EPYC 64C 2GHz, AMD MI250X	120,832	19.20	309	62.6
1	EPYC 64C 2GHz, AMD MI250X	8,730,112	1,102.00	21,100	52.2
3	EPYC 64C 2GHz, AMD MI250X	1,110,144	151.90	2,942	51.6
10	EPYC 64C 2GHz, AMD MI250X	319,072	46.10	921	50.0
326	Xeon Platinum 8260M 24C 2.4GHz	1,664	2.18	53	40.9
315	EPYC 7543 32C 2.8GHz, NVIDIA A100	16,704	2.27	103	33.9
319	EPYC 7742 64C 2.25GHz, NVIDIA A100	19,840	2.25	72	31.5
304	EPYC 7763 64C 2.45GHz, NVIDIA A100	26,880	2.29	74	30.7
105	EPYC 7742 64C 2.25GHz, NVIDIA A100	47,616	5.05	147	29.9
363	Xeon Platinum 8360Y 36C 2.4GHz, NVIDIA A100	20,160	2.07	69	29.9



# Рейтинг TOP500 ([www.top500.org](http://www.top500.org))

## 2022 год

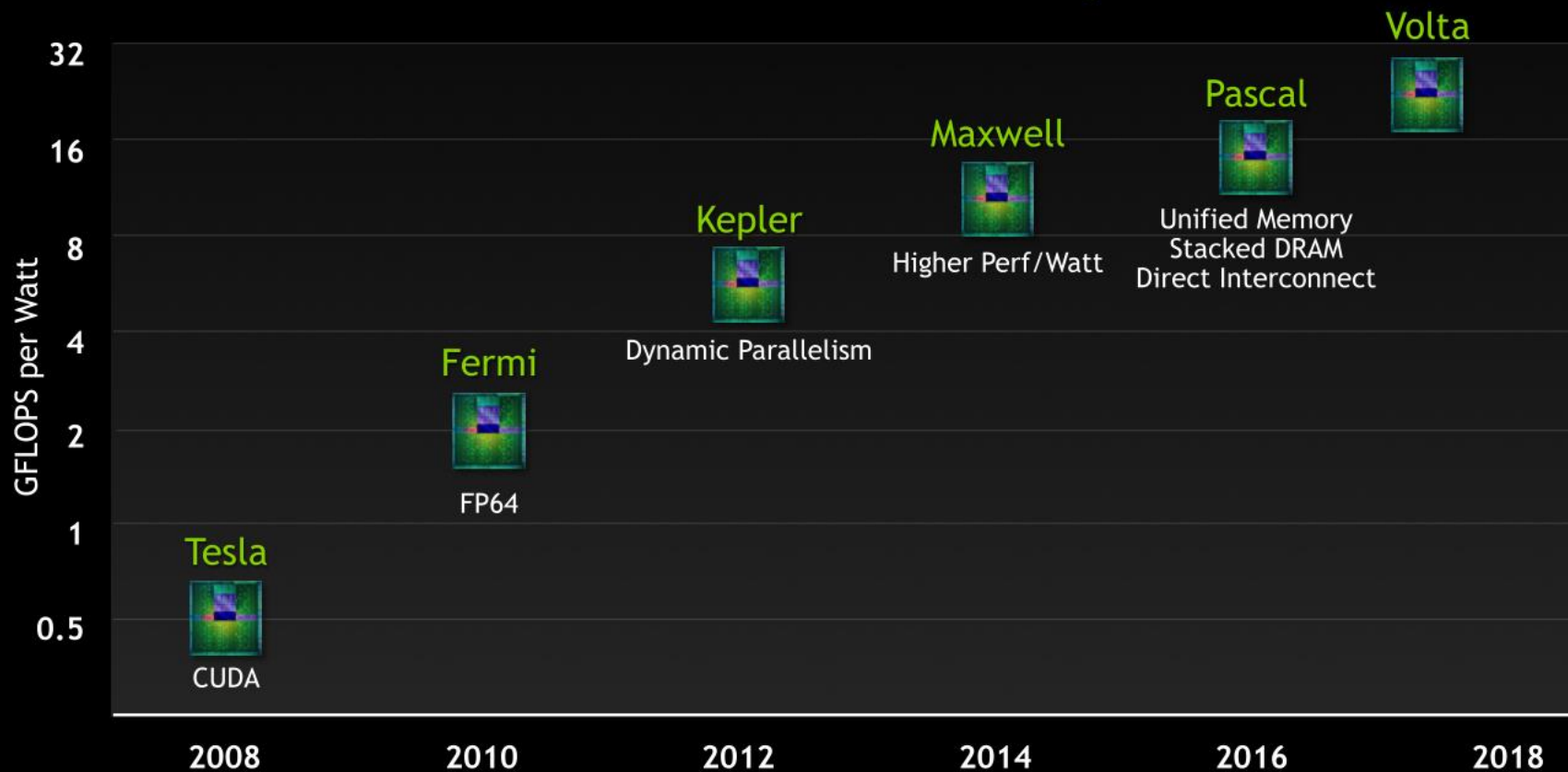
TOP500 Rank	System	Cores	Rmax PFlop/s	Rpeak PFlop/s	Power mW
1	Frontier - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X	8,730,112	1,102	1,685.65	21.1
2	Supercomputer Fugaku - Supercomputer Fugaku, A64FX 48C 2.2GHz	7,630,848	442	537.21	29.8
3	LUMI - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X	1,110,144	151	214.35	2.9
4	Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100	2,414,592	148	200.79	10
5	Sierra - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100	1,572,480	94	125.71	7.4
6	Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz	10,649,600	93	125.44	15.3

# СК Ломоносов 1 (2012 год) и СК К60 ИПМ (2019 год)

- Ломоносов 1: (потребление ~1600 КВт, произв. ~1700 Tflops)  
узлов: 5130 [2x Intel Xeon X5570/X5670]  
узлов: 1065 [2x Intel Xeon E5630 + 2x NVIDIA Tesla X2070]
- К60: (потребление ~16 КВт, произв. ~470 Tflops)  
узлов: 16 [2x Intel Xeon Gold 6142 + 4x GPU Tesla V100]

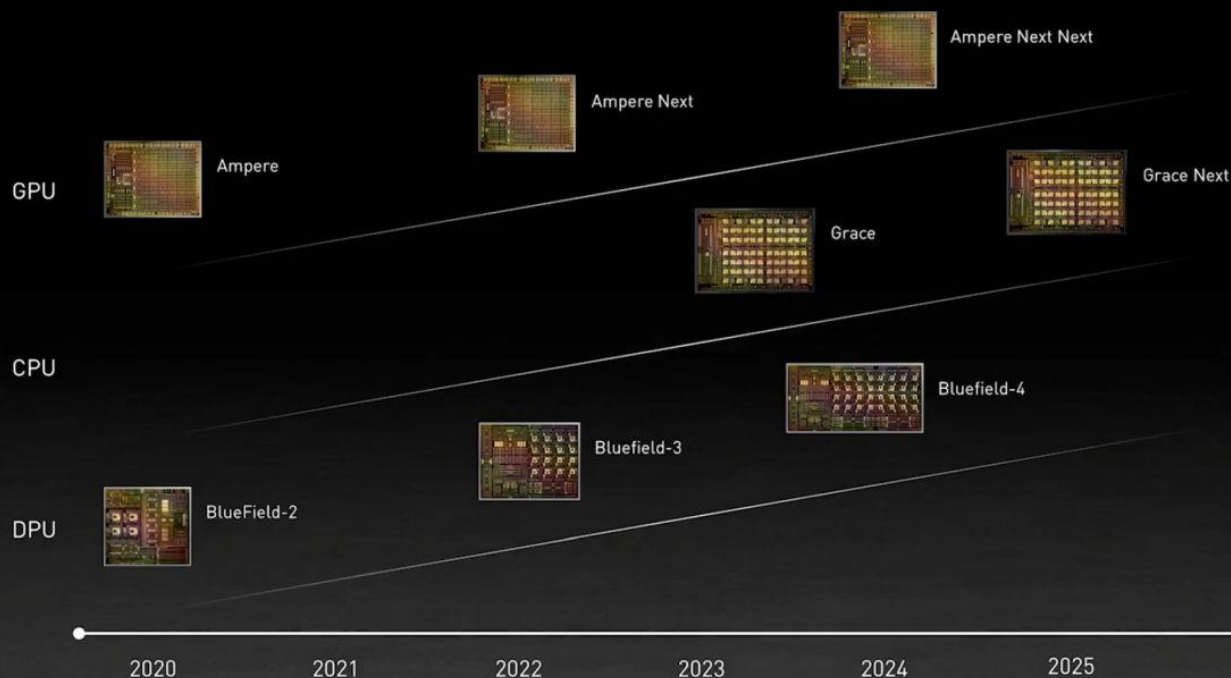
# Эффективность растёт (road map поколений)

## CUDA GPU Roadmap



# Эффективность растёт (road map поколений)

3 CHIPS. YEARLY LEAPS. ONE ARCHITECTURE.



# Эффективность растёт

## NVIDIA Data-Center GPUs Specifications

NVIDIA H100

NVIDIA A100

NVIDIA Tesla V100

NVIDIA Tesla P100



Picture

GPU

Transistors

Die Size

Architecture

Fabrication Node

GPU Clusters

CUDA Cores

L2 Cache

Tensor Cores

Memory Bus

Memory Size

TDP

Interface

Launch Year

GH100

80B

814 mm<sup>2</sup>

Hopper

TSMC N4

132/114\*

16896/14592\*

50MB

528/456\*

5120-bit

80 GB HBM3/HBM2e\*

700W/350W\*

SXM5/\*\*PCIe Gen5

2022

GA100

54.2B

828 mm<sup>2</sup>

Ampere

TSMC N7

108

6912

40MB

432

5120-bit

40/80GB HBM2e

250W/300W/400W

SXM4/PCIe Gen4

2020

GV100

21.1B

815 mm<sup>2</sup>

Volta

12nm FFN

80

5120

6MB

320

4096-bit

16/32 HBM2

250W/300W/450W

SXM2/PCIe Gen3

2017

GP100

15.3B

610 mm<sup>2</sup>

Pascal

16nm FinFET+

56

3584

4MB

-


4096-bit

16GB HBM2

250W/300W

SXM/PCIe Gen3

2016



# Программно-аппаратная модель: архитектура GPU NVidia



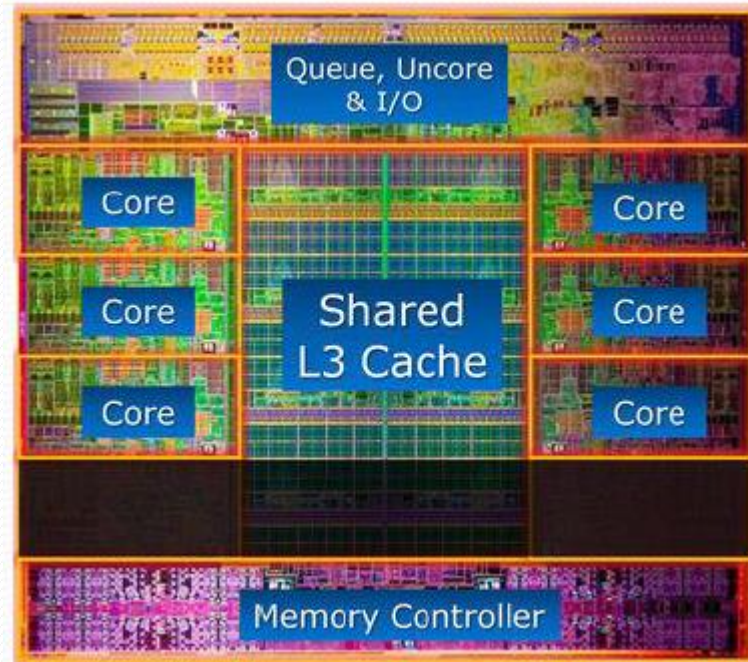
# Семейства GPU устройств



# CPU Intel Core i7 / i9

- Небольшое число мощных независимых ядер;
- До 36 ядер, ~2.0-3.0 ГГц каждое;
- Поддержка виртуальных потоков (Hyper-Threading)
- 3х уровневый, большой кеш L3 до ~64 МБ;
- На каждое ядро L1=32KB (data) + 32KB ( Instructions), L2=256KB;
- Обращения в память обрабатываются «отдельно» для каждого процесса\нити

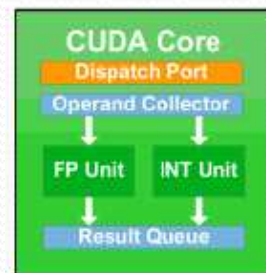
Core i7 3960x,  
6 ядер, 15MB L3





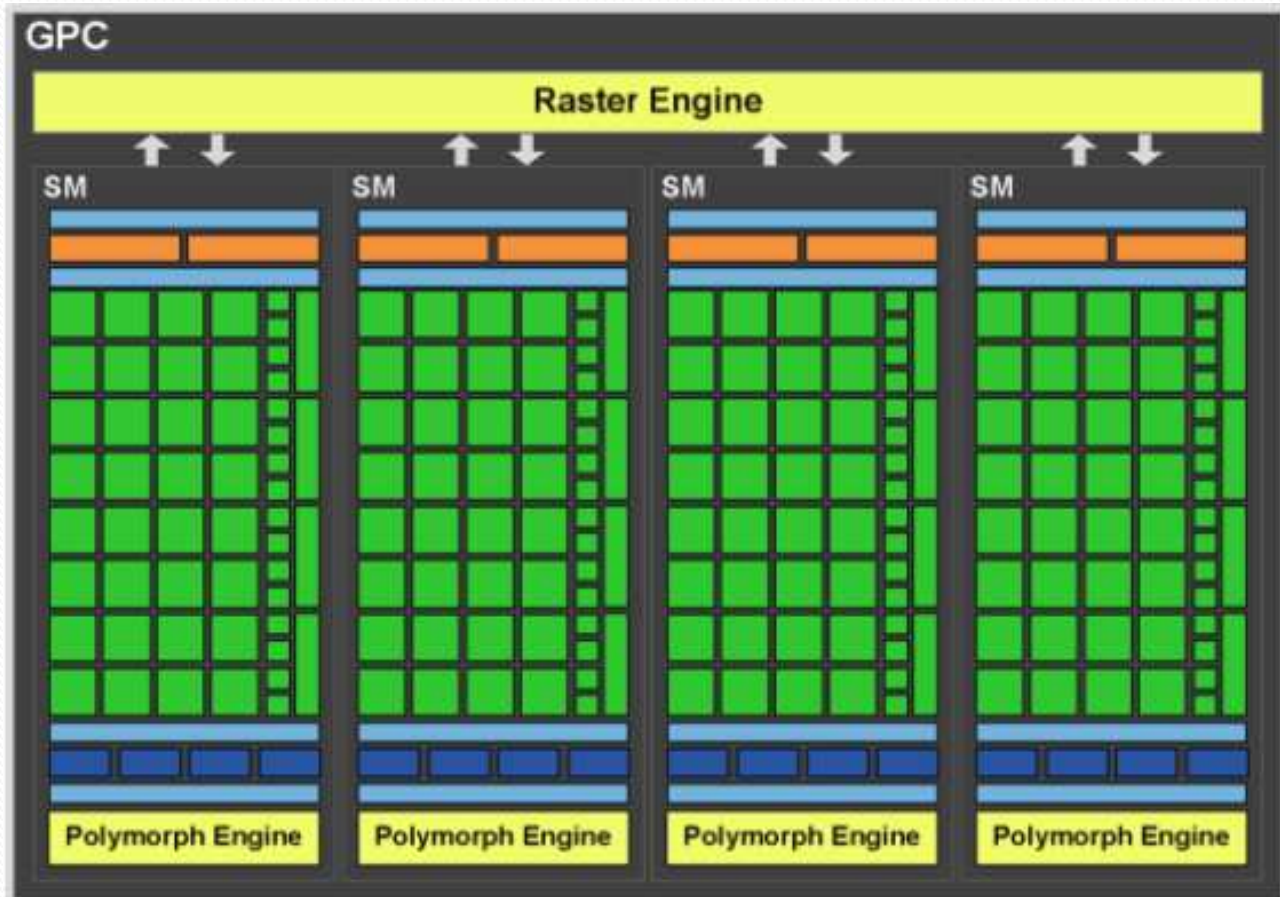
# GPU Streaming Multiprocessor (SM)

- Поточковый мультипроцессор
- «Единица» построения устройства (как ядро в CPU):
  - 32 скалярных ядра CUDA Core, ~2.0ГГц
  - 2 Warp Scheduler
  - Файл регистров, 128KB
  - 2х уровневый кэш
  - Текстурные юниты
  - 16 x Special Function Unit (SFU) – интерполяция и трансцендентная математика одинарной точности
  - 16 x Load/Store



# GPC - Graphics Processing Cluster

- Объединение потоковых мультипроцессоров в блоки



# Чип поколения Fermi в максимальной конфигурации

- 16 SM
- 512 ядер CUDA Core
- Кеш L2 758KB
- Контроллеры памяти GDDR5
- Интерфейс PCI 2.0





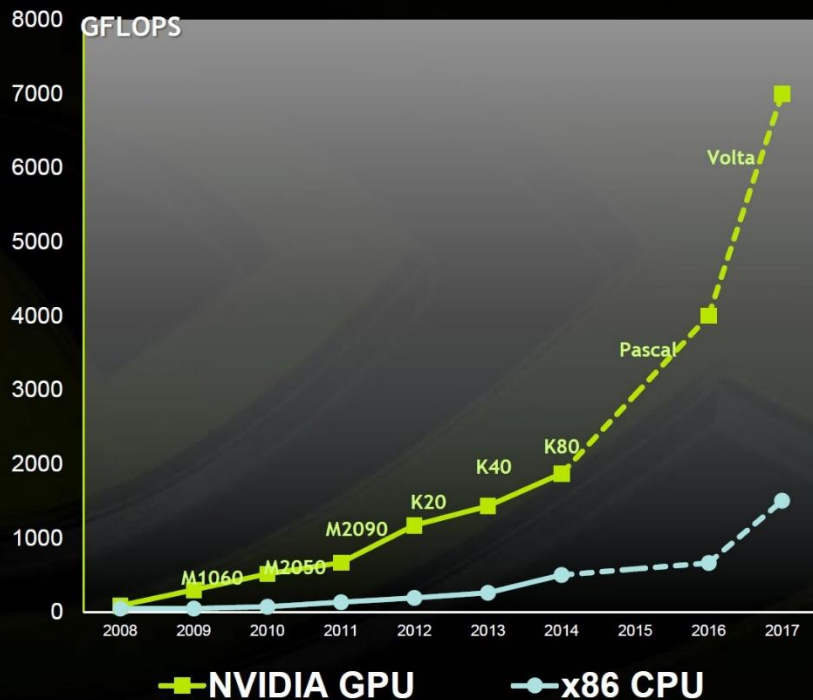
# Вычислительная мощность



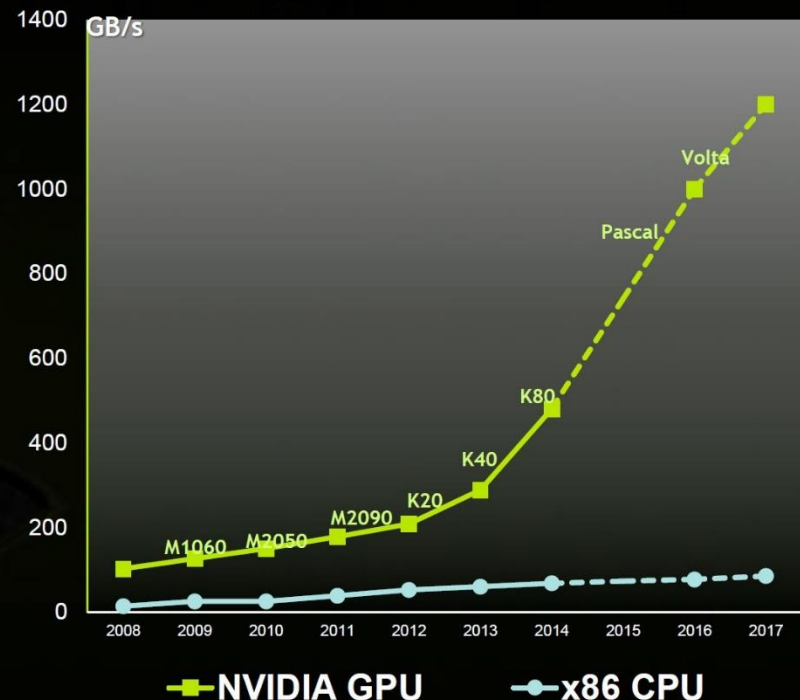
## GPU Motivation

## Performance Trends

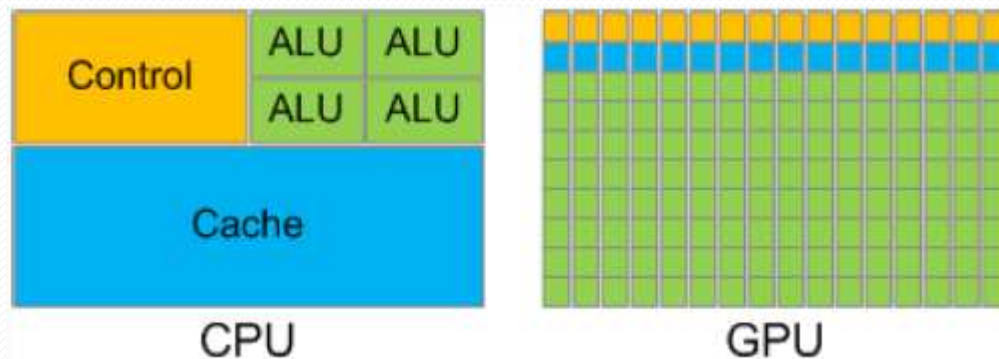
### Peak Double Precision FLOPS



### Peak Memory Bandwidth



# Сравнение GPU и CPU



- Сотни упрощённых вычислительных ядер, работающих на небольшой тактовой частоте **~2.5 ГГц**;
- Небольшие кеши на GPU
  - 32 CUDA-ядра разделяют 64 КБ L1
  - L2 общий для всех CUDA ядер, **L3 отсутствует**
- Оперативная память с высокой пропускной способностью и **высокой латентностью**, оптимизированная для коллективного доступа;
- Поддержка миллионов виртуальных нитей, быстрое переключение контекста для групп нитей.

# Утилизация латентности памяти

- Цель: эффективно загружать CUDA-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;
- За счёт наличия сотен ядер и поддержки миллионов нитей (потребителей) на GPU легче утилизировать всю полосу пропускания



# CUDA: гибридное программирование

# 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, но существует исключение!
- 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 код, в котором содержатся вызовы функций, побочный эффект которых – манипуляции с устройством.

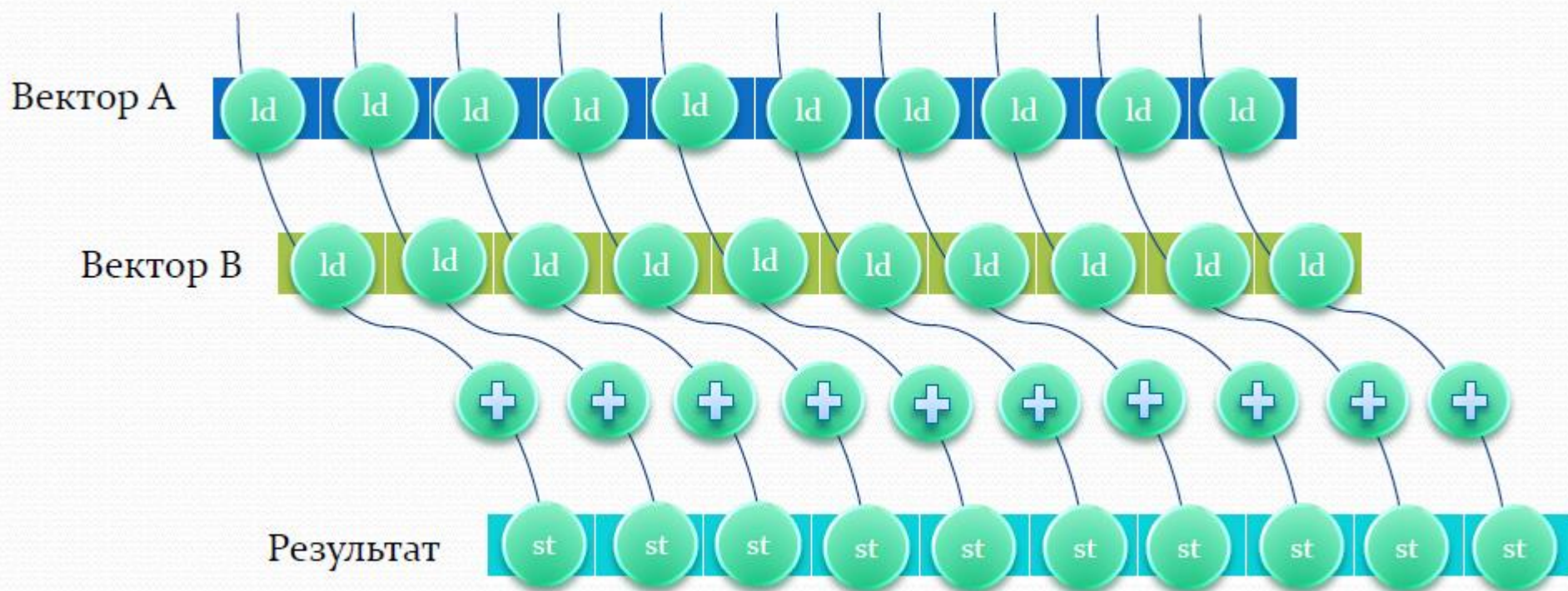
# Код для GPU (device-code)

- Код для **GPU** пишется на C++ с некоторыми 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
  - ...
- Компилируется специальным компилятором nvcc

# Код для CPU (host-code)

- Код для **CPU** дополняется вызовами специальных функций для работы с устройством;
- Код для **CPU** компилируется обычным компилятором gcc/icc/cl;
- Кроме конструкции запуска ядра <<<...>>> !
- Функции для **GPU** линкуются из динамических библиотек

# Сложение векторов



# Сложение векторов

- Без GPU:

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

- С GPU:

```
{ //на CPU
```

```
    <Переслать данные с CPU на GPU>;
```

```
    <Запустить вычисления на N GPU-нитех>;
```

```
    <Скопировать результат с GPU на CPU>;
```

```
}
```

```
{//в нити с номером IDX
```

```
    c[IDX] = a[IDX] + b[IDX];
```

```
}
```

# 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).

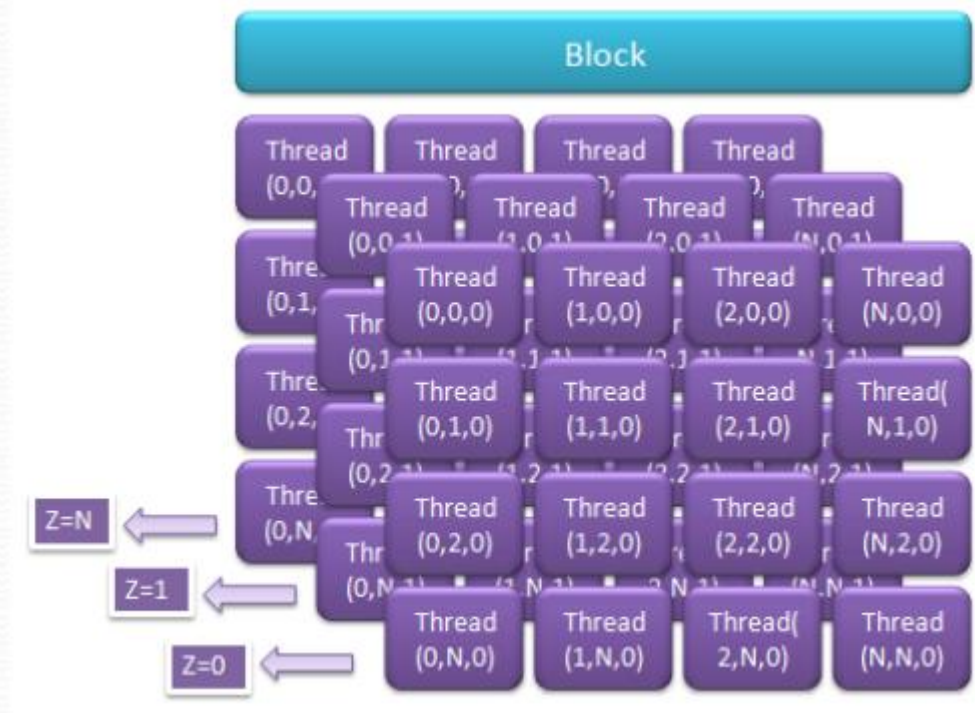
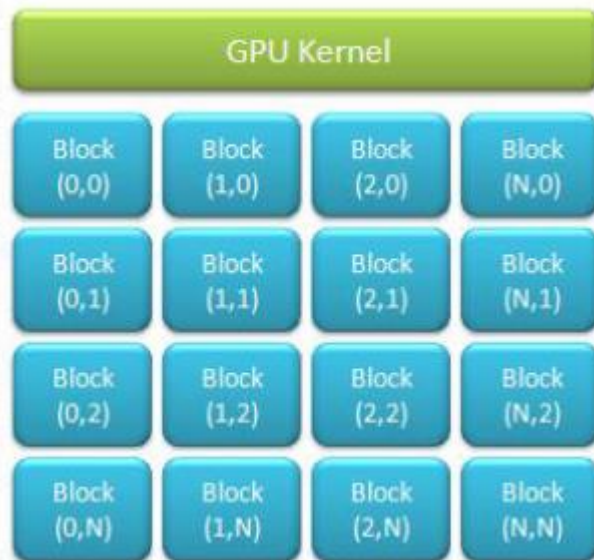
# CUDA Grid

- Грид задаётся количеством блоков по  $[X, Y, Z]$  (размер грида в блоках) и размерами каждого блока по  $[X, Y, Z]$ ;
- Например, 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»)

- Каждая нить выполняет копию специально оформленных функций «ядер», компилируемых для GPU.
  - Нет возвращаемого значения (**void**);
  - Обязательный атрибут **\_\_global\_\_**.

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

# CUDA Kernel («Ядро»)

- Терминология:
  - 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
  - «Ядро» – что делать
  - «Грид» – сколько делать

# Запуск 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,
  - Три поля: **unsigned x,y,z**
  - Конструктор **dim3(unsigned x=1, unsigned y=1, unsigned z=1)**

# Ориентация нити в гриде

- Осуществляется за счёт встроенных переменных:
  - *threadIdx.x threadIdx.y threadIdx.z* - индексы нити в блоке
  - *blockIdx.x blockIdx.y blockIdx.z* – индексты блока в гриде
  - *blockDim.x blockDim.y blockDim.z* – размеры блоков в нитях
  - *gridDim.x gridDim.y gridDim.z* – размеры грида в блоках

- Линейный индекс нити в гриде:

```
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

```
__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 вектором на GPU);
- Определяет своё положение в гриде **threadLinearIdx** ;
- Считывает из входных векторов элементы с индексом **threadLinearIdx** и записывает их сумму в выходной вектор по индексу **threadLinearIdx** ;
- рассчитывает один 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: Копирование памяти

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

```
float * inputDataOnHost = (float *)malloc( nb ) ;// память на хосте для входных данных
float * resultOnHost = (float *)malloc( nb ); // память на хосте для результата
float * inputDataOnDevice= NULL , *resultOnDevice = NULL; // память на устройстве
getInputData(inputDataOnHost); // получить входные данные
```

```
cudaMalloc( (void**)& inputDataOnDevice, nb) ; // выделить память на устройстве для входных данных
cudaMalloc( (void**)& resultOnDevice, nb) ; // выделить память на устройстве для хранения результата
```

```
cudaMemcpy(inputDataOnDevice, inputDataOnHost , nb, cudaMemcpyHostToDevice); // переслать на устройство
                                                                    входные данные
```

```
//запустить ядро. Выходные данные получим в resultOnDevice
```

```
cudaMemcpy(resultOnHost , resultOnDevice , nb, cudaMemcpyDeviceToHost); // переслать результаты на хост
```

```
cudaFree(inputDataOnDevice) ; // освободить память на устройстве
cudaFree(resultOnDevice ) ; // освободить память на устройстве
```

# Пример: Запуск ядра

```
int n = getSize(); // размер задачи
```

```
//определения указателей, получение входных данных на хосте
```

```
cudaMalloc( (void**)& inputDataOnDevice, nb); // выделить память на устройстве для входных данных
```

```
cudaMalloc( (void**)& resultOnDevice, nb); // выделить память на устройстве для хранения результата
```

```
cudaMemcpy(inputDataOnDevice, inputDataOnHost, nb, cudaMemcpyHostToDevice); // переслать на устройство  
входные данные
```

```
dim3 blockDim = dim3(512), gridDim = dim3( (n - 1) / 512 + 1 ); // рассчитать конфигурацию запуска
```

```
kernel <<< gridDim, blockDim >>> (inputDataOnDevice, resultOnDevice, n); // запустить ядро с рассчитанной  
конфигурацией и параметрами
```

```
cudaMemcpy(resultOnHost, resultOnDevice, nb, cudaMemcpyDeviceToHost); // переслать результаты на хост
```

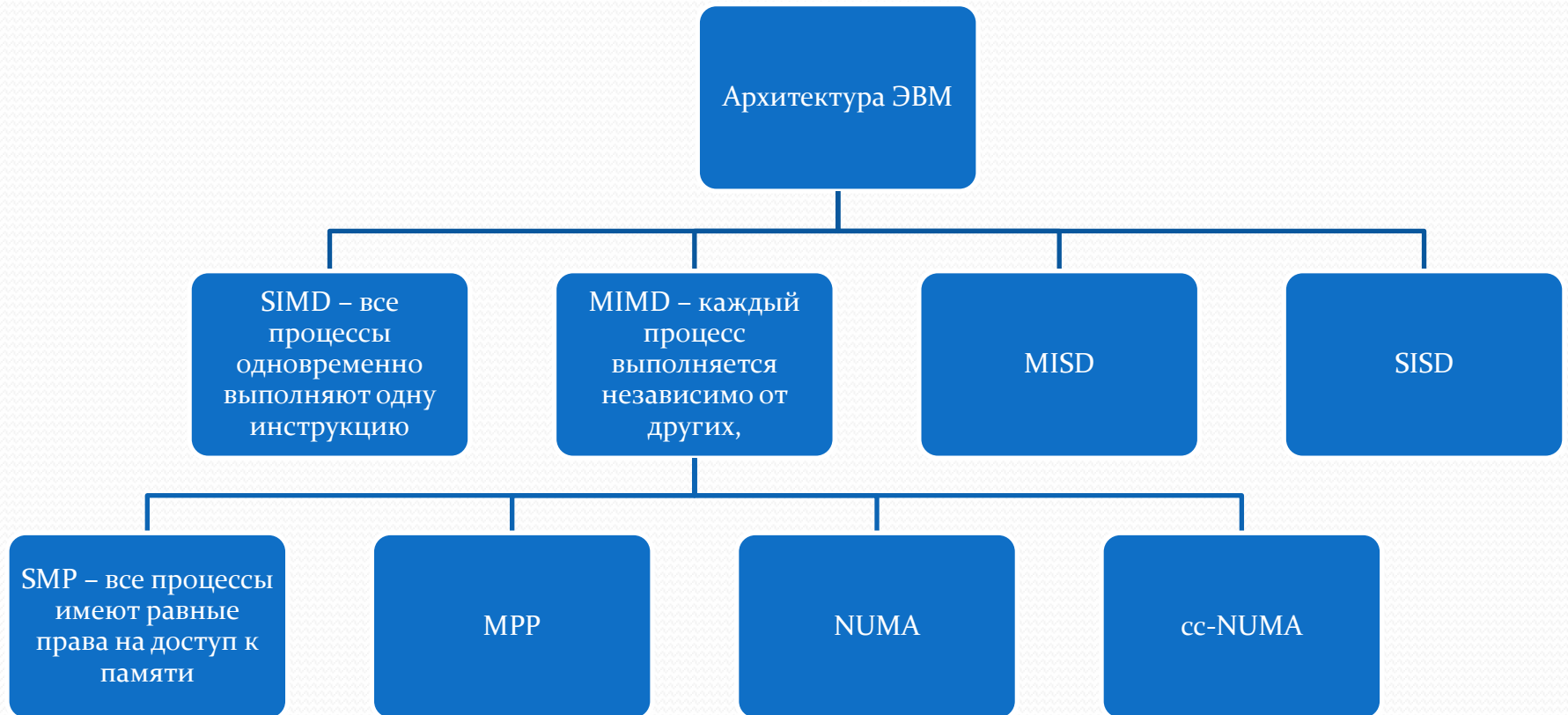
```
cudaFree(inputDataOnDevice); // освободить память на устройстве
```

```
cudaFree(resultOnDevice); // освободить память на устройстве
```



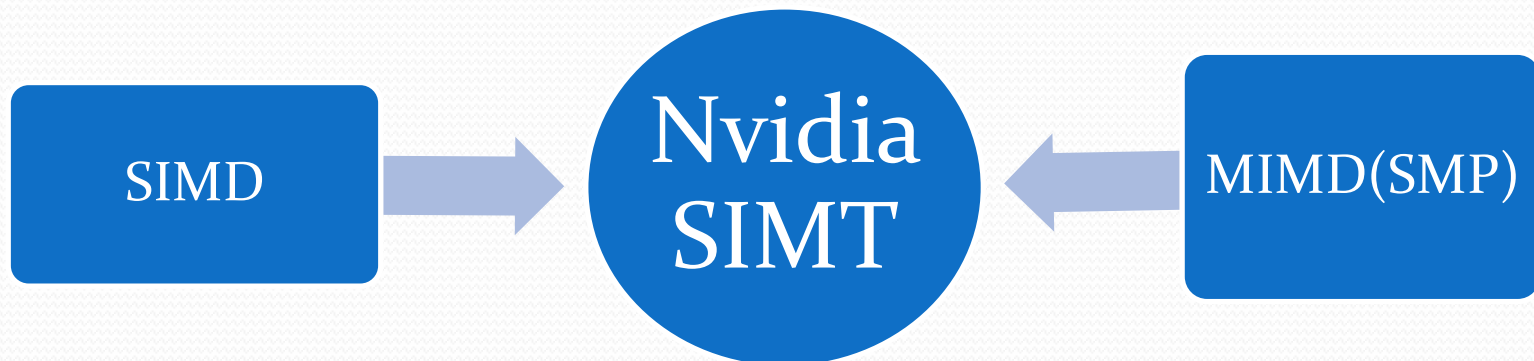
# Модель исполнения SIMT

# 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





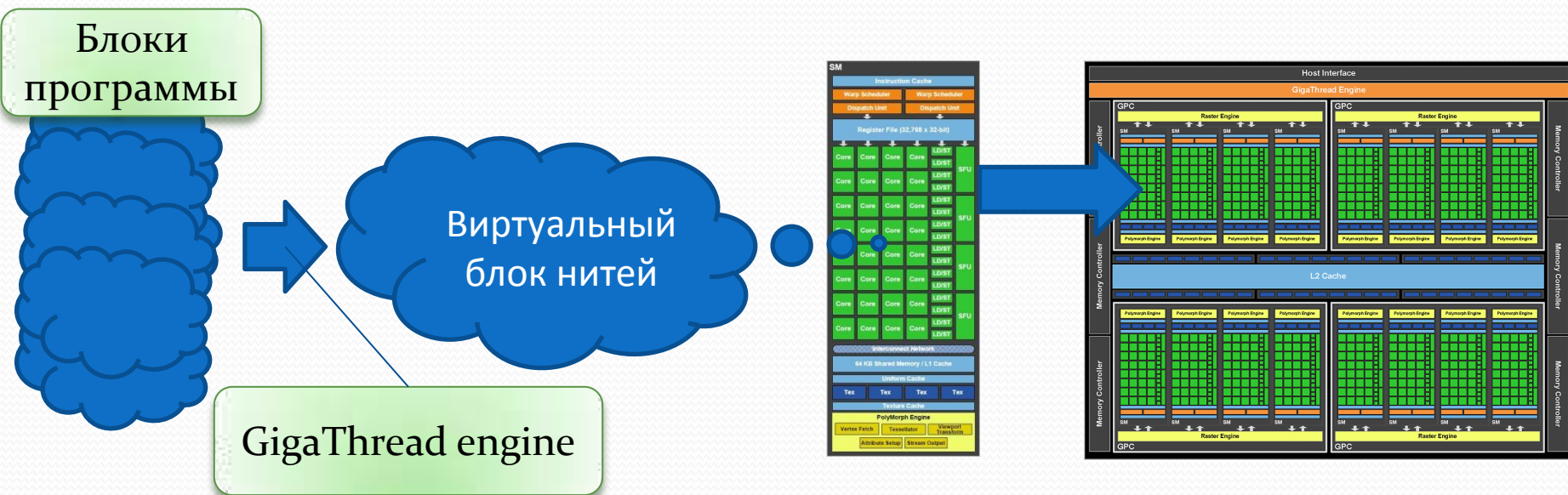
# SIMT: виртуальные нити, блоки

- 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 (есть исключение) , глобальная синхронизация всех нитей невозможна, нити из разных блоков выполняются полностью независимо
  - 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.



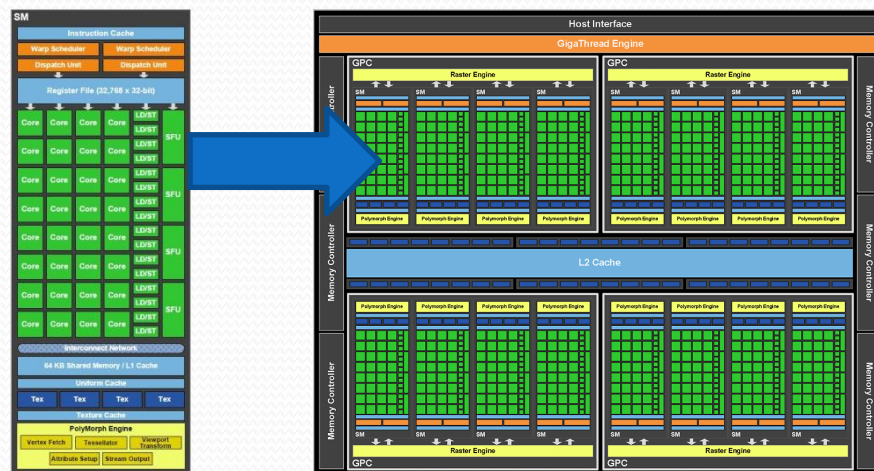
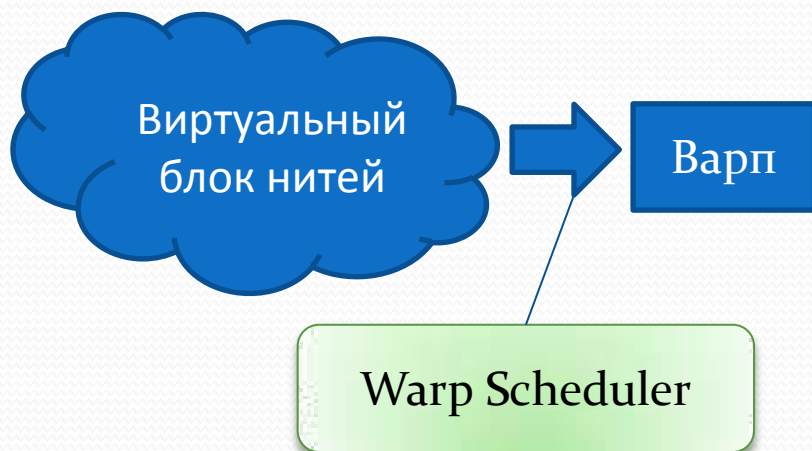
# SIMT: аппаратное выполнение

- Все нити из одного блока выполняются на одном мультипроцессоре (SM)
- Максимальное число нитей в блоке – 1024
- Блоки не мигрируют между SM
- Распределение блоков по мультипроцессорам непредсказуемо
- Каждый SM работает **независимо от других**



# Блоки и варпы

- Блоки нитей по фиксированному правилу разделяются на группы по 32 нити, называемые **варпами (warp)**
- Все нити варпа **одновременно** выполняют **одну общую** инструкцию (в точности SIMD-выполнение) !
- Warp Scheduler на каждом цикле работы выбирает варп, все нити которого готовы к выполнению следующей инструкции и запускает весь варп

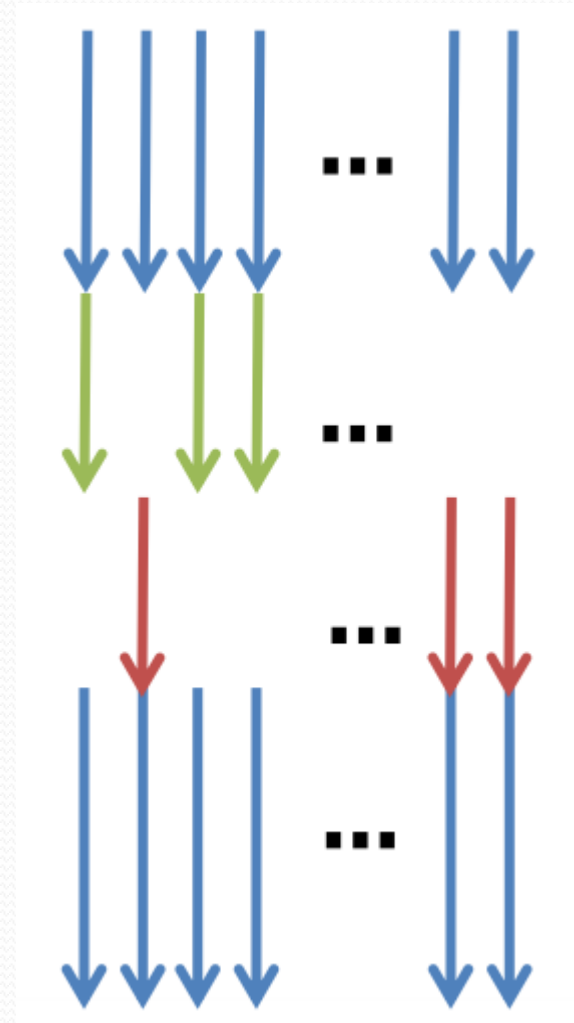
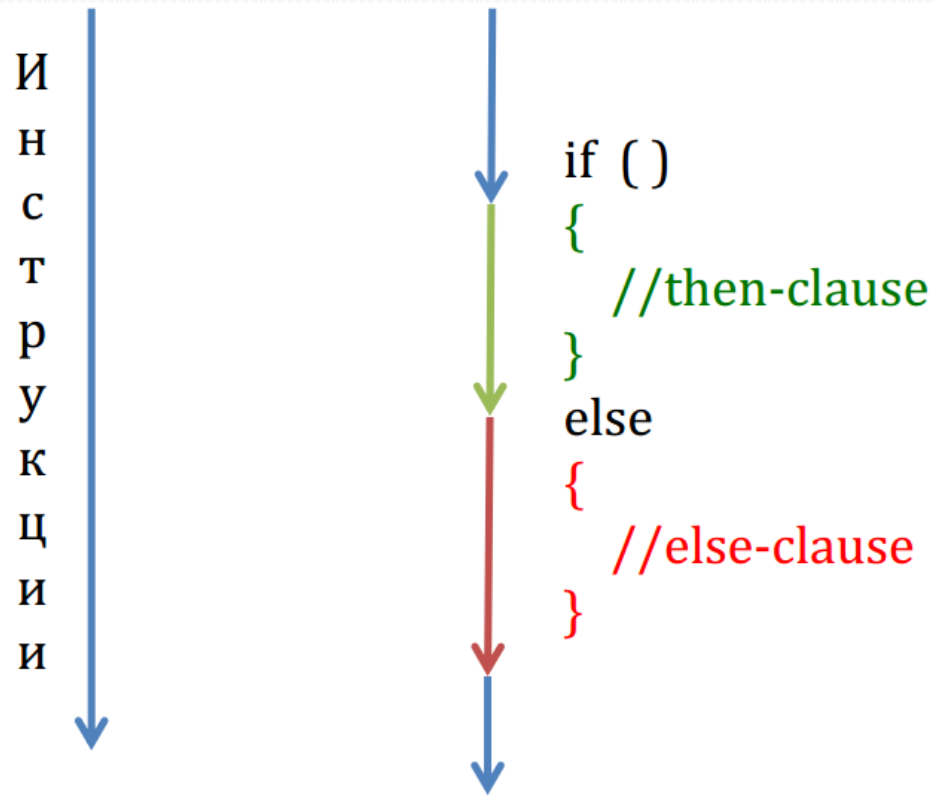


# Ветвление (branching)

- Все нити варпа 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 варпа

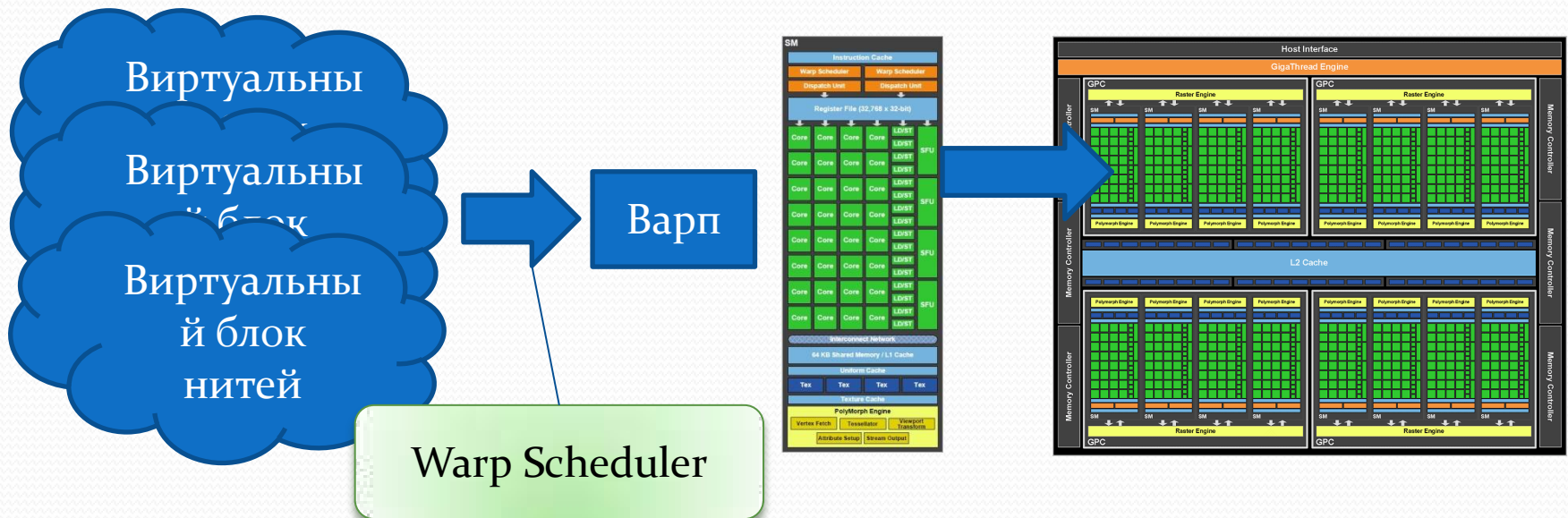
Эти нити «замаскируются» нулями в специальном наборе регистров и не будут её выполнять, т.е. **будут простаивать**

# Ветвление (branching)



# Несколько блоков на одном SM

- SM может работать с варпами нескольких блоков одновременно
  - Максимальное число резидентных блоков на одном мультипроцессоре – 8
  - Максимальное число резидентных варпов – 48 = 1536 нитей



# Загруженность (Оссирансу)

- Чем больше нитей активно на мультипроцессоре, тем эффективнее используется оборудование
  - Блоки по 1024 нити – 1 блок на SM, 1024 нити, 66% от максимума
  - Блоки по 100 нитей – 8 блоков на SM, 800 нитей, 52%
  - Блоки по 512 нитей – 3 блока на SM, 1536 нитей, 100%

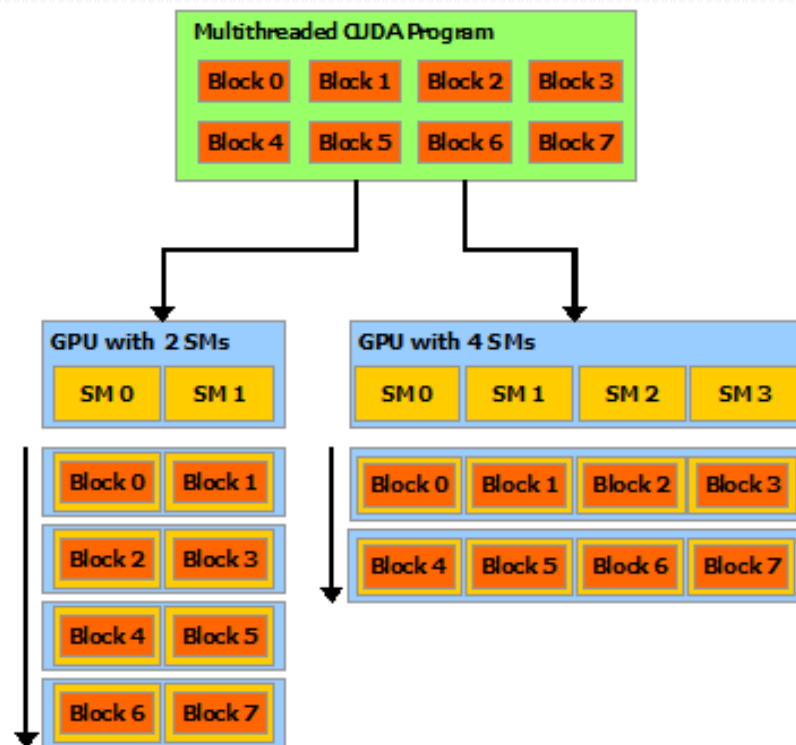


# SIMT и глобальная синхронизация

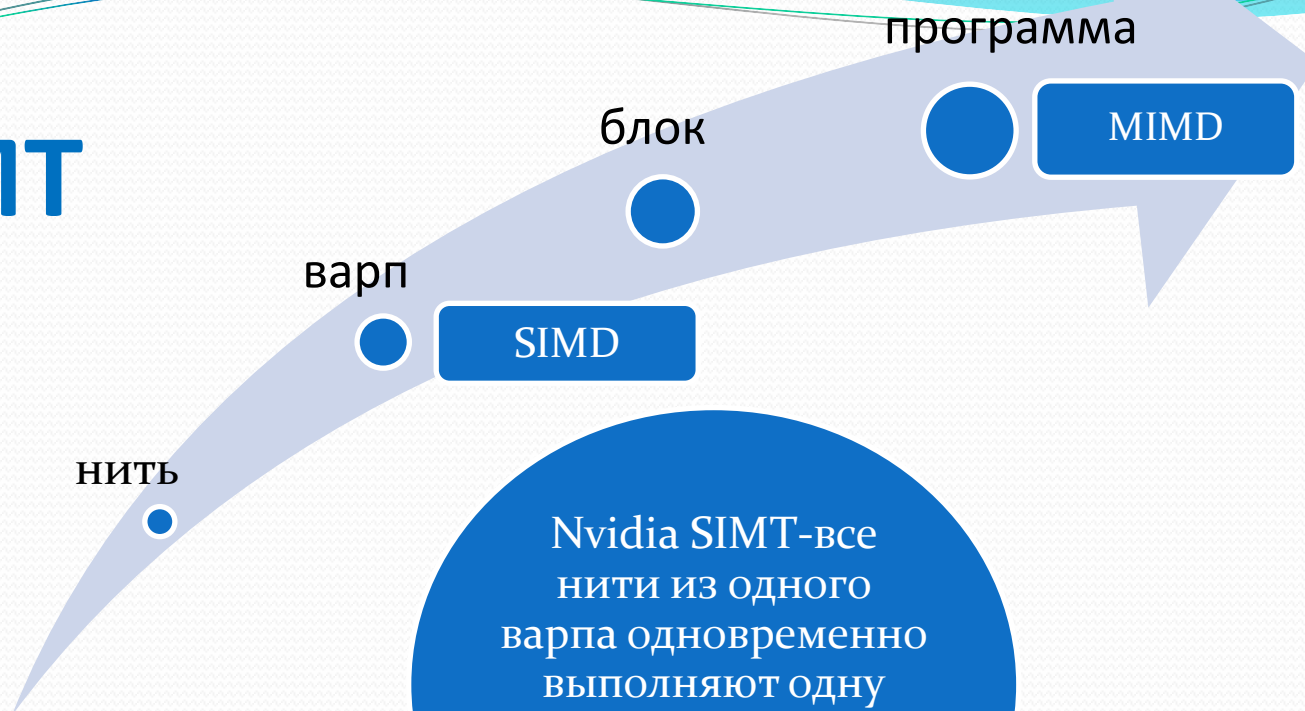
- В общем случае, из-за ограничений по числу нитей и блоков на одном SM, не удаётся разместить сразу все блоки программы на GPU
  - Часть блоков ожидает выполнения
    - Поэтому в общем случае невозможна глобальная синхронизация
  - Блоки выполняются по мере освобождения ресурсов
    - Нельзя предсказать порядок выполнения блоков
- Если все блоки программы удалось разместить, то возможна глобальная синхронизация через атомарные операции
  - Вручную, специальная техника «Persistent Threads»

# SIMT и масштабирование

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



# SIMT



Nvidia SIMT-все  
нити из одного  
варпа одновременно  
выполняют одну  
инструкцию, варпы  
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

# Выводы

Хорошо распараллеливаются на GPU задачи, которые:

- Имеют параллелизм по данным
  - Одна и та же последовательность вычислений, применяемая к разным данным
- Могут быть разбиты на подзадачи 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