Технология CUDA для высокопроизводительных вычислений на кластерах с GPU

Лихогруд Николай
n.lihogrud@gmail.com

Часть первая

Введение

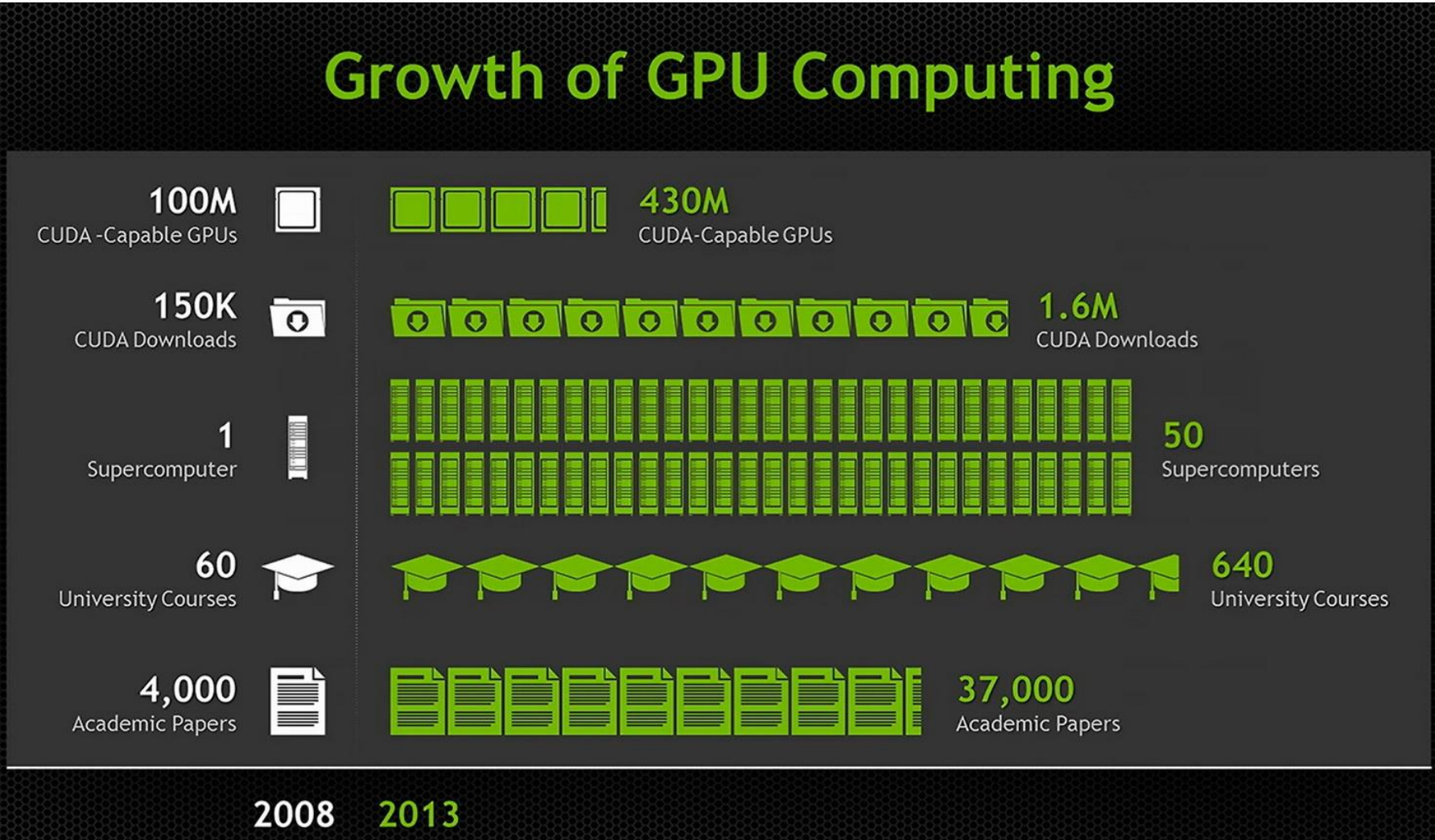
GPGPU & CUDA

- **GPU - *G**raphics **P**rocessing **U**nit***

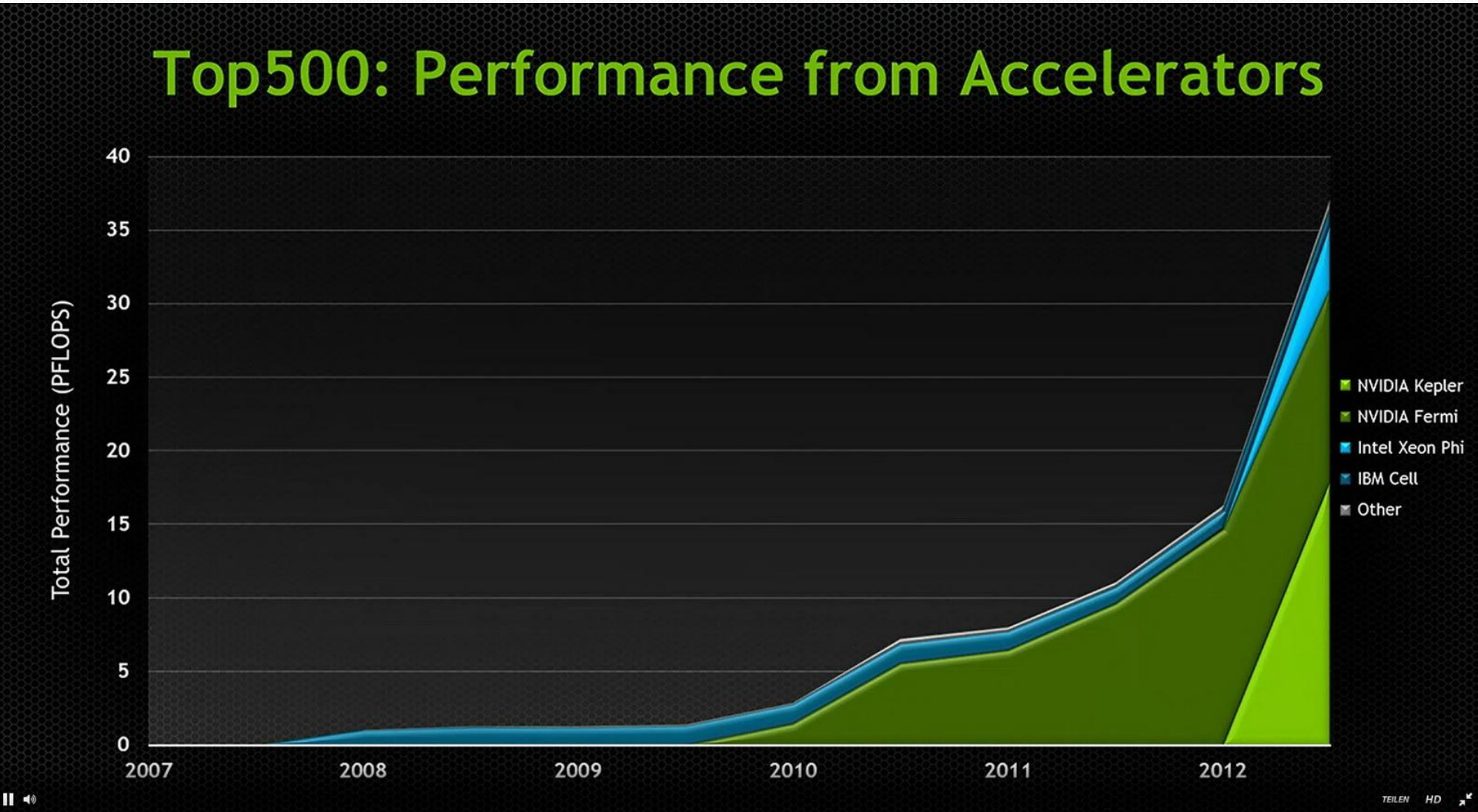
- **GPGPU** - ***G**eneral-**P**urpose* computing on **GPU**, вычисления общего вида на GPU
  - Первые **GPU** от NVIDIA с поддержкой **GPGPU** – GeForce восьмого поколения, G80 (2006 г)

- ***CUDA - **C**ompute **U**nified **D**evice **A**rchitecture***
  - Программно-аппаратная архитектура от Nvidia, позволяющая производить вычисления с использованием графических процессоров

CUDA растет


Growth of GPU Computing
100M
CUDA -Capable GPUs
430M
CUDA-Capable GPUs
150K
CUDA Downloads
1.6M
CUDA Downloads
1
Supercomputer
50
Supercomputers
60
University Courses
640
University Courses
4,000
Academic Papers
37,000
Academic Papers
2008
2013

Ускорители в top500



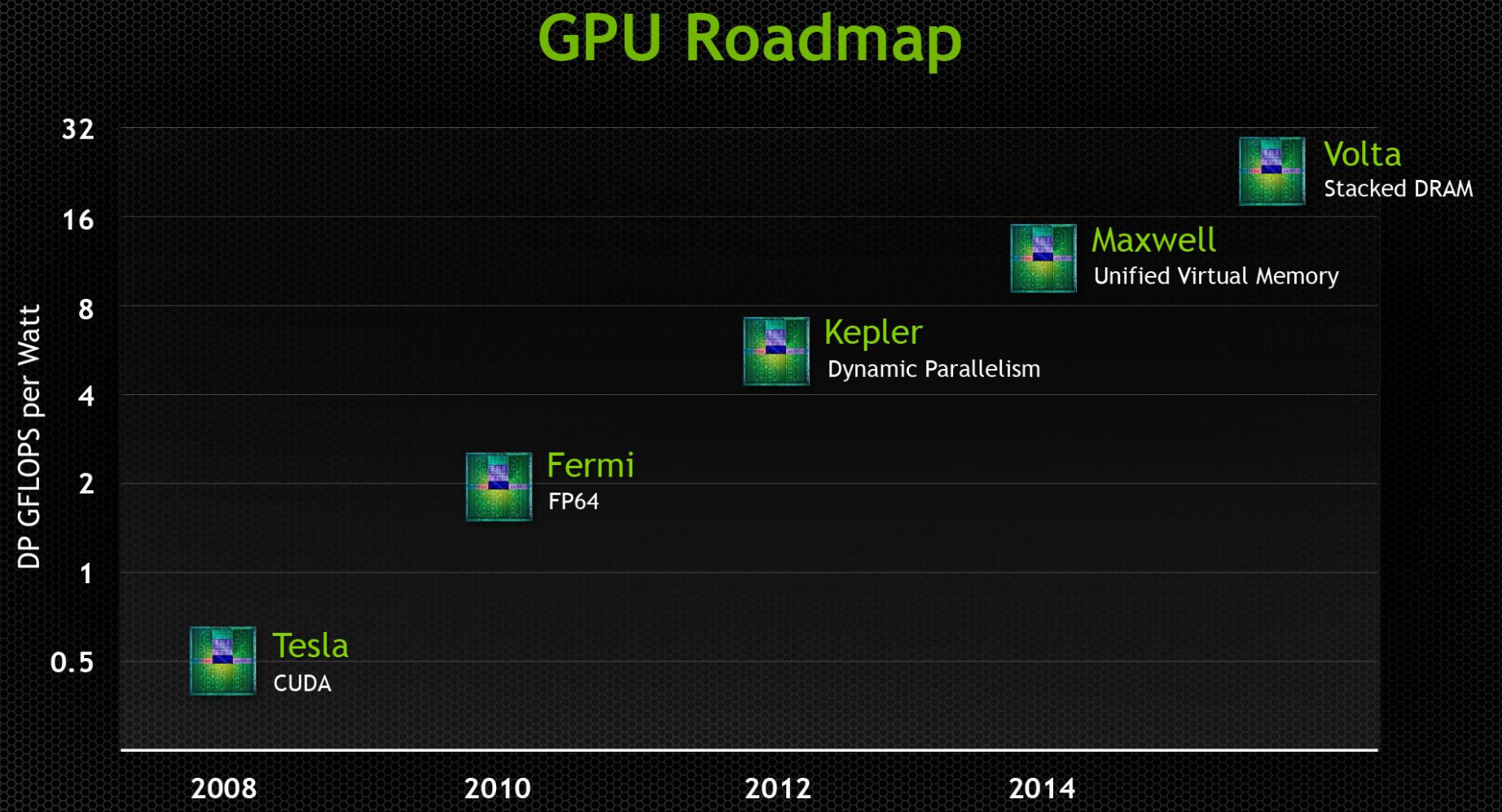
Преимущества GPGPU

- Соотношения
  - Цена\производительность
  - Производительность\энергопотребление

Green500

Green500 Rank	MFLOPS/W	Site*	Computer*	Total Power (kW)
1	4,389.82	GSIC Center, Tokyo Institute of Technology	TSUBAME-KFC - LX 1U-4GPU/104Re-1G Cluster, Intel Xeon E5-2620v2 6C 2.100GHz, Infiniband FDR, NVIDIA K20x	34.58
2	3,631.70	Cambridge University	Wilkes - Dell T620 Cluster, Intel Xeon E5-2630v2 6C 2.600GHz, Infiniband FDR, NVIDIA K20	52.62
3	3,517.84	Center for Computational Sciences, University of Tsukuba	HA-PACS TCA - Cray 3623G4-SM Cluster, Intel Xeon E5-2680v2 10C 2.800GHz, Infiniband QDR, NVIDIA K20x	78.77
4	3,459.46	SURFsara	Cartesius Accelerator Island - Bullx B515 cluster, Intel Xeon E5-2450v2 8C 2.5GHz, InfiniBand 4× FDR, Nvidia K40m	44.40
5	3,185.91	Swiss National Supercomputing Centre (CSCS)	Piz Daint - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Level 3 measurement data available	1,753.66
6	3,131.06	ROMEO HPC Center - Champagne-Ardenne	romeo - Bull R421-E3 Cluster, Intel Xeon E5-2650v2 8C 2.600GHz, Infiniband FDR, NVIDIA K20x	81.41
7	3,019.72	CSIRO	CSIRO GPU Cluster - Nitro G16 3GPU, Xeon E5-2650 8C 2GHz, Infiniband FDR, Nvidia K20m	86.20
8	2,951.95	GSIC Center, Tokyo Institute of Technology	TSUBAME 2.5 - Cluster Platform SL390s G7, Xeon X5670 6C 2.93GHz, Infiniband QDR, NVIDIA K20x	927.86
9	2,813.14	Exploration & Production - Eni S.p.A.	HPC2 - iDataPlex DX360M4, Intel Xeon E5-2680v2 10C 2.8GHz, Infiniband FDR, NVIDIA K20x	1,067.49
10	2,678.41	Financial Institution	iDataPlex DX360M4, Intel Xeon E5-2680v2 10C 2.800GHz, Infiniband, NVIDIA K20x	54.60

Эффективность энергопотребления



Семейства GPU Nvidia

Высокопроизводительные вычисления

Профессиональная графика

Развлечения

Часть 1: Аппаратно-программная модель

Аппаратная архитектура GPU Nvidia

Как устроено GPU?

Compute Capability

- CUDA развивалась постепенно, многие возможности API недоступны на старых архитектурах
- Возможности устройства определяются его **Compute Capability**

<***номер поколения***>.<***номер модификации***>

- Для эффективного программирования с использованием GPU нужно учитывать Compute Capability используемого устройства

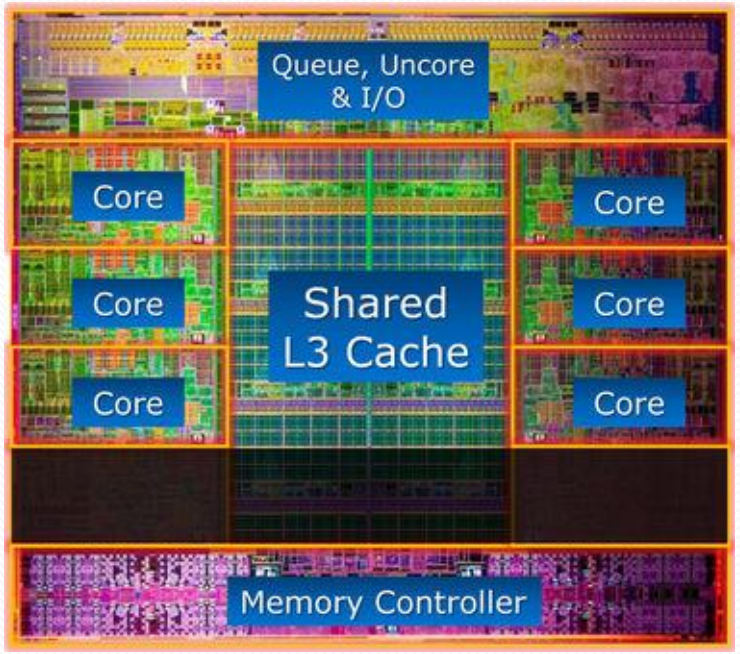
Compute Capability

- Поколение **Tesla** (не путать с линией продуктов для HPC)
  - 1.1 – базовые возможности CUDA, атомарные операции с глобальной памятью
  - 1.2 - атомарные операции с общей памятью, warp vote-функции
  - 1.3 – вычисления с двойной точностью
- Поколение **Fermi**
  - 2.0 - новая архитектура чипа, ECC, кеши L1 и L2, асинхронное выполнение ядер, UVA и др.
  - 2.1 - новая архитектура warp scheduler-ов
- Поколение **Kepler**
  - 3.0, 3.2 – Новая архитектура чипа, Unified memory programming, warp shfl и др.
  - 3.0 - Динамический параллелизм, Hyper Queqe и др.
- Поколение **Maxwell**
  - sm_50 and sm_52 – Новая архитектура чипа
- ….

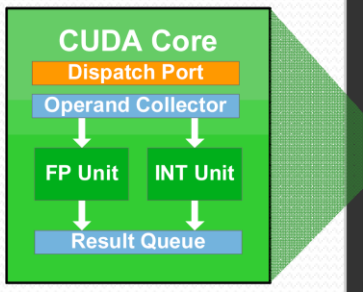
CPU Intel Core I-7

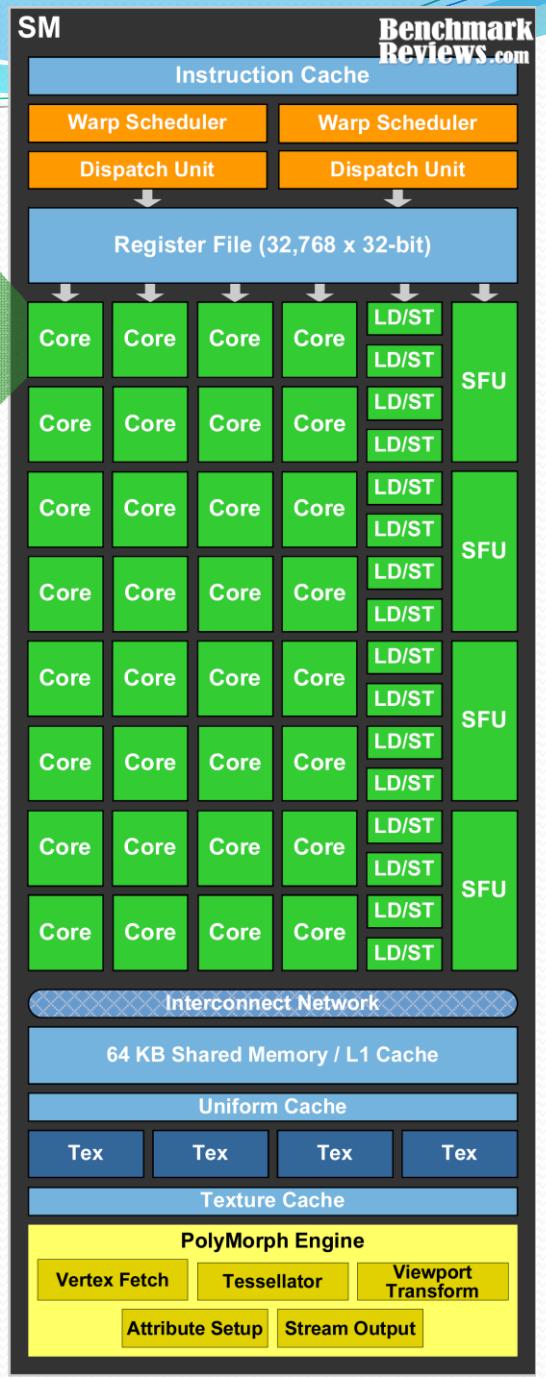
- Небольшое число мощных независимых ядер
  - 2,4,6,8 ядер, 2,66—3,6ГГц каждое
  - Каждое физическое ядро определяется системой как 2 логических и может параллельно выполнять два потока (Hyper-Threading)
- 3 уровня кешей, большой кеш L3
  - На каждое ядро L1=32KB (data) + 32KB (Instructions), L2=256KB
  - Разделяемый L3 до 20 mb
- Обращения в память обрабатываются отдельно для каждого процесса\нити

Core I7-3960x,
6 ядер, 15MB L3



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

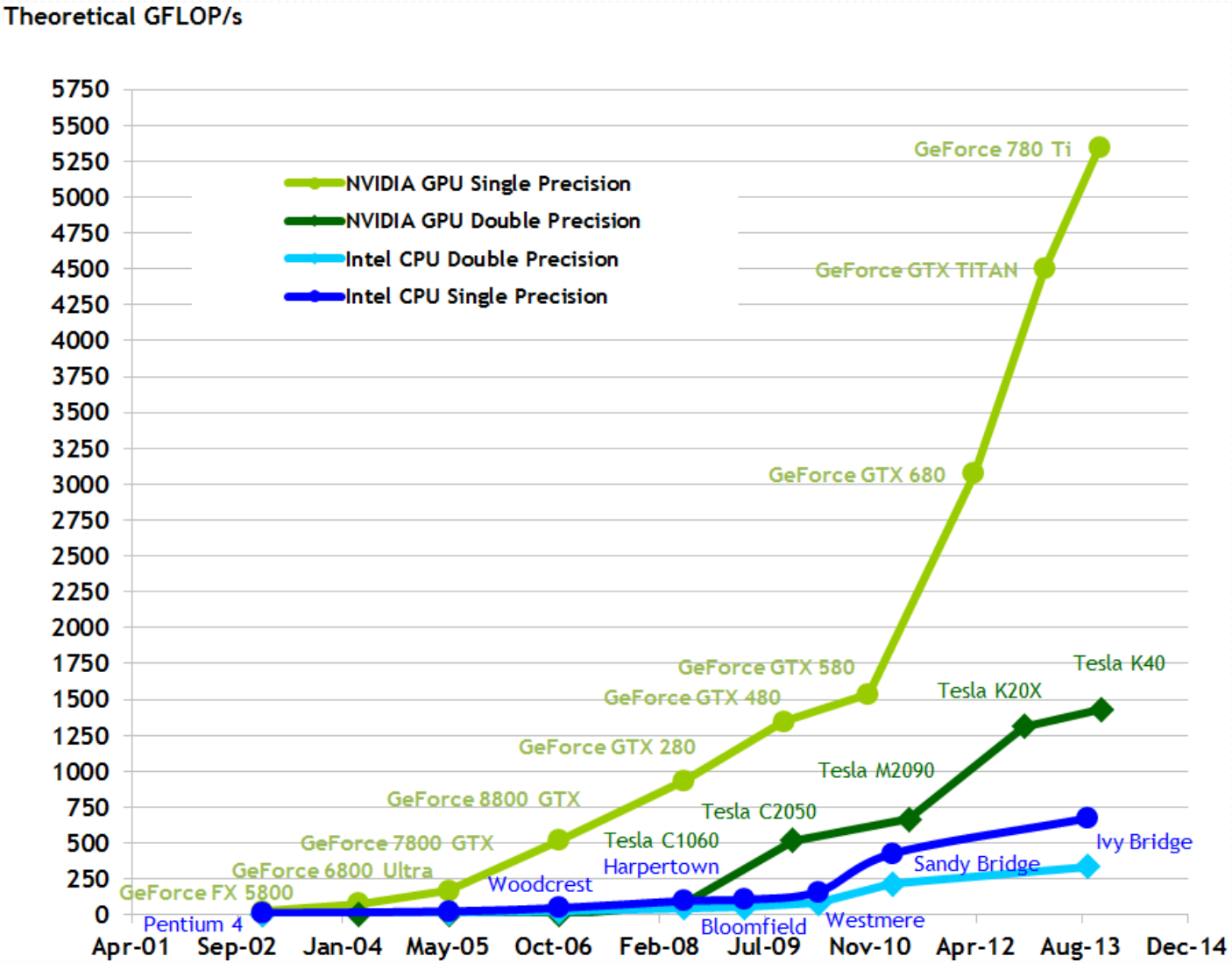
Kepler: SMX

- 192 cuda core
- 64 x DP Unit
- 32 x SFU
- 32x load/store Unit
- 4 x warp scheduler
- 256KB регистров

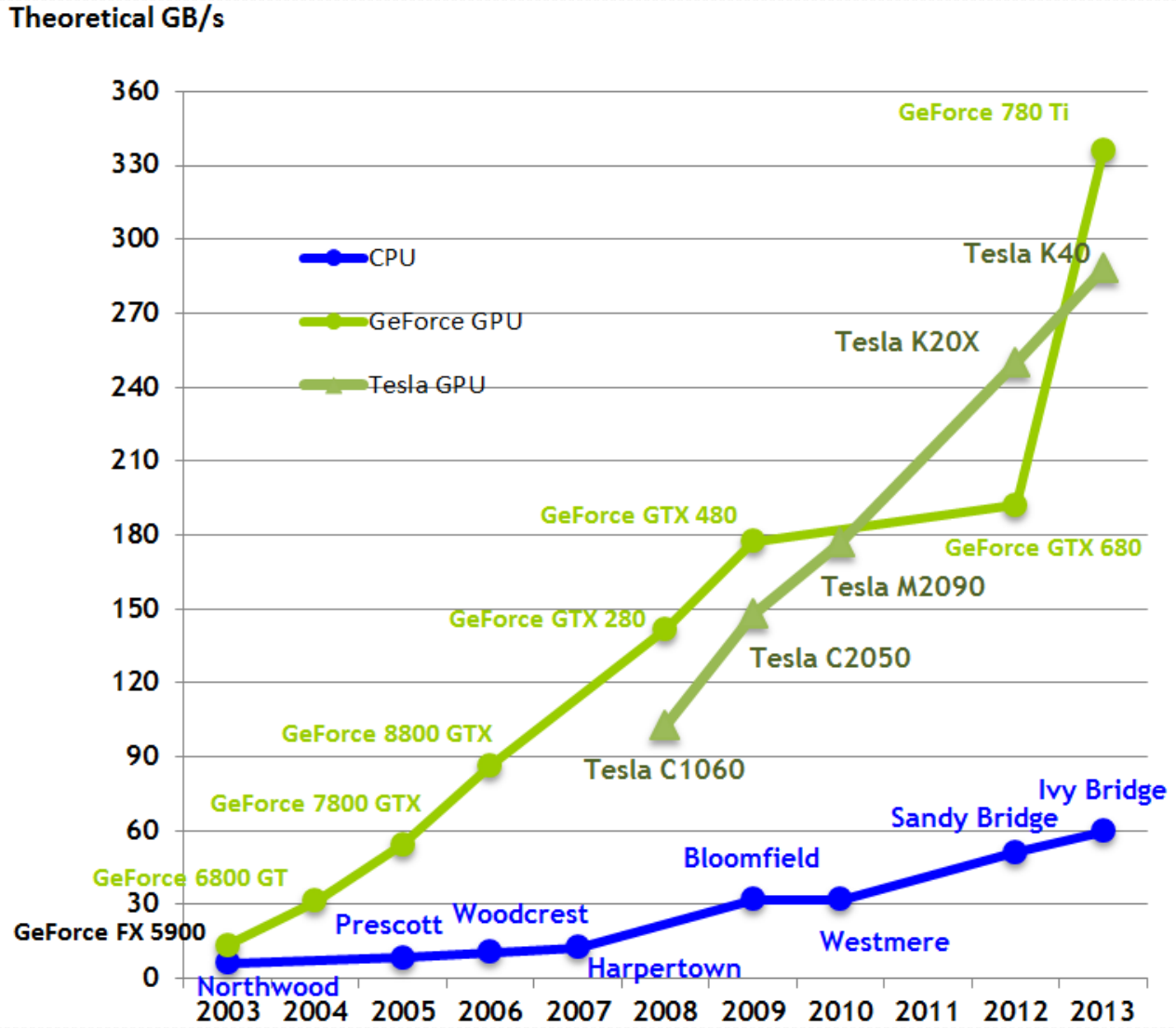
Kepler: Чип в максимальной конфигурации

- 15 SXM = **2880 cuda core**

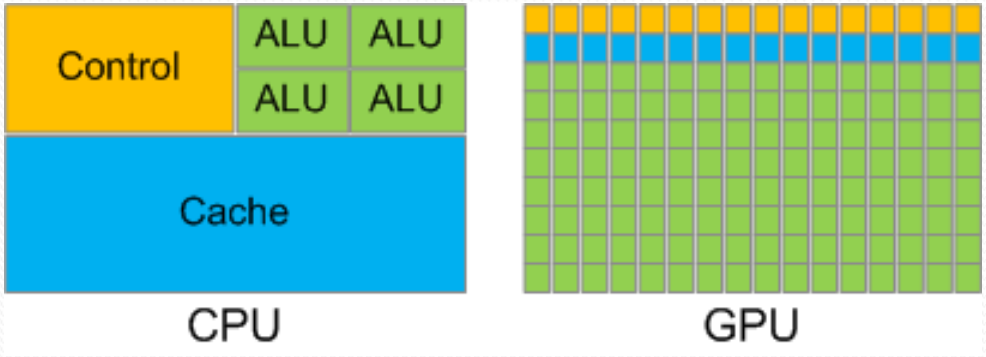
Вычислительная мощность



Пропускная способность памяти

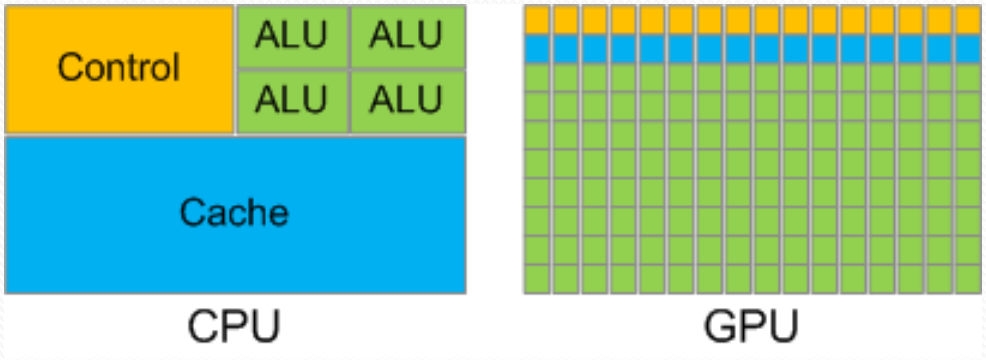


Сравнение GPU и CPU



- 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



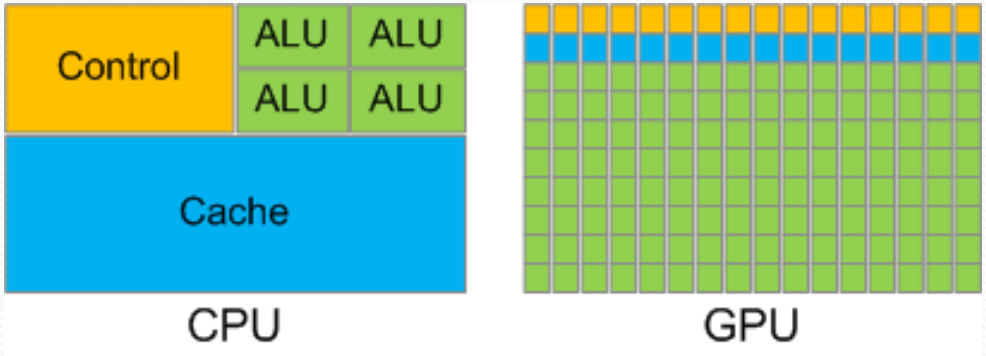
- Цель: эффективно загружать Ядра

  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

CUDA: Гибридное программирование CPU+GPU

Из чего состоит программа с использованием CUDA?

Вычисления с использованием GPU

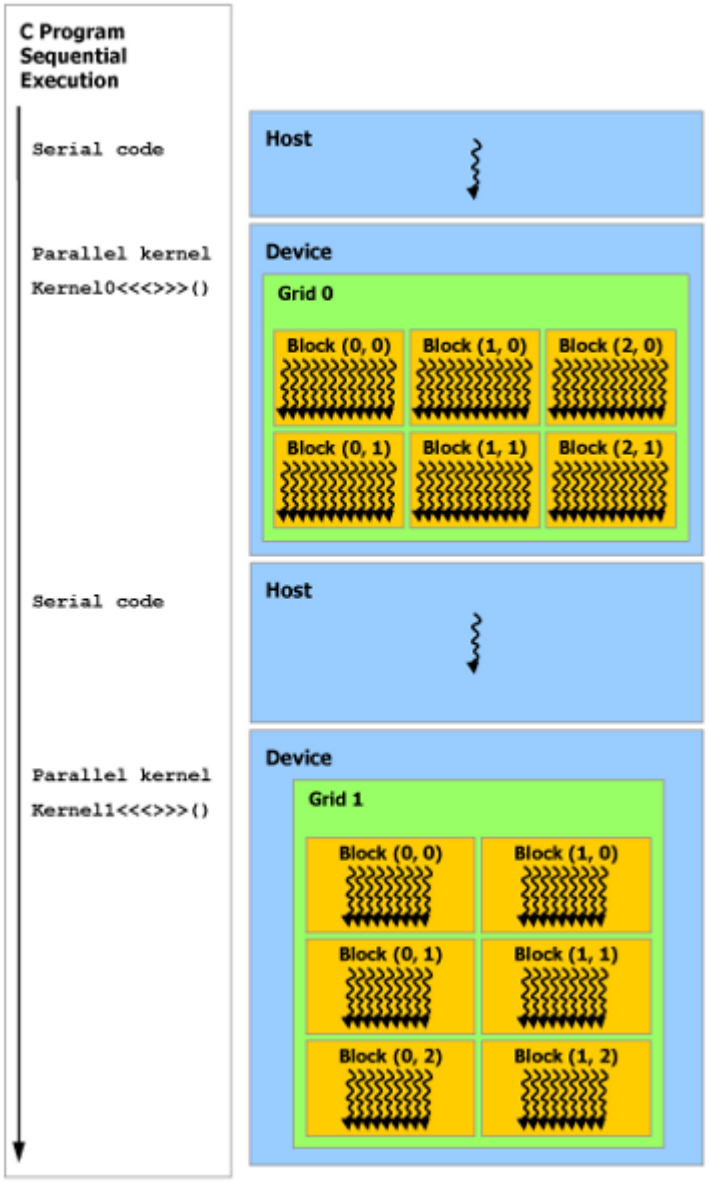
- Программа, использующая GPU, состоит из:
  - Кода для GPU, описывающего необходимые вычисления и работу с памятью устройства
  - Кода для CPU, в котором осуществляется
    - Управление памятью GPU – выделение / освобождение
    - Обмен данными между GPU/CPU
    - Запуск кода для GPU
    - Обработка 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.



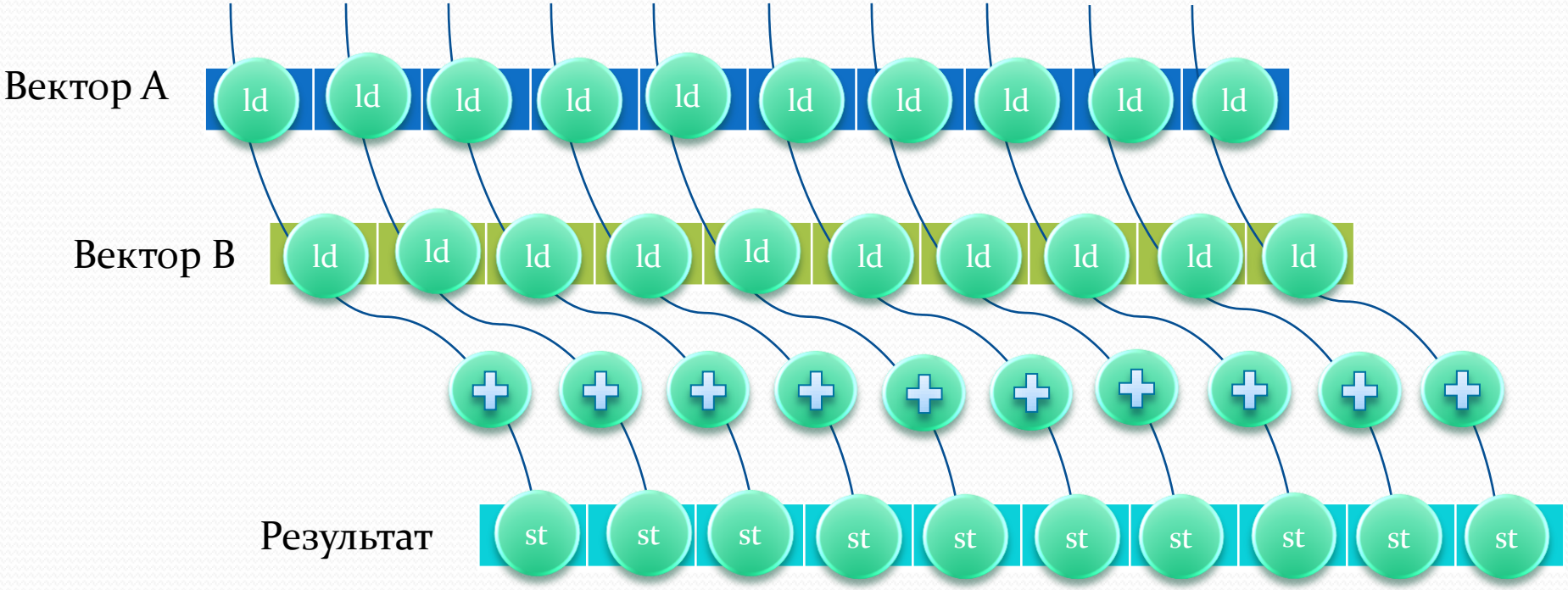
Код для GPU (device-code)

- 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`

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

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-нитях>;
    <Скопировать результат с GPU на CPU>;
}

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

SPMD & CUDA

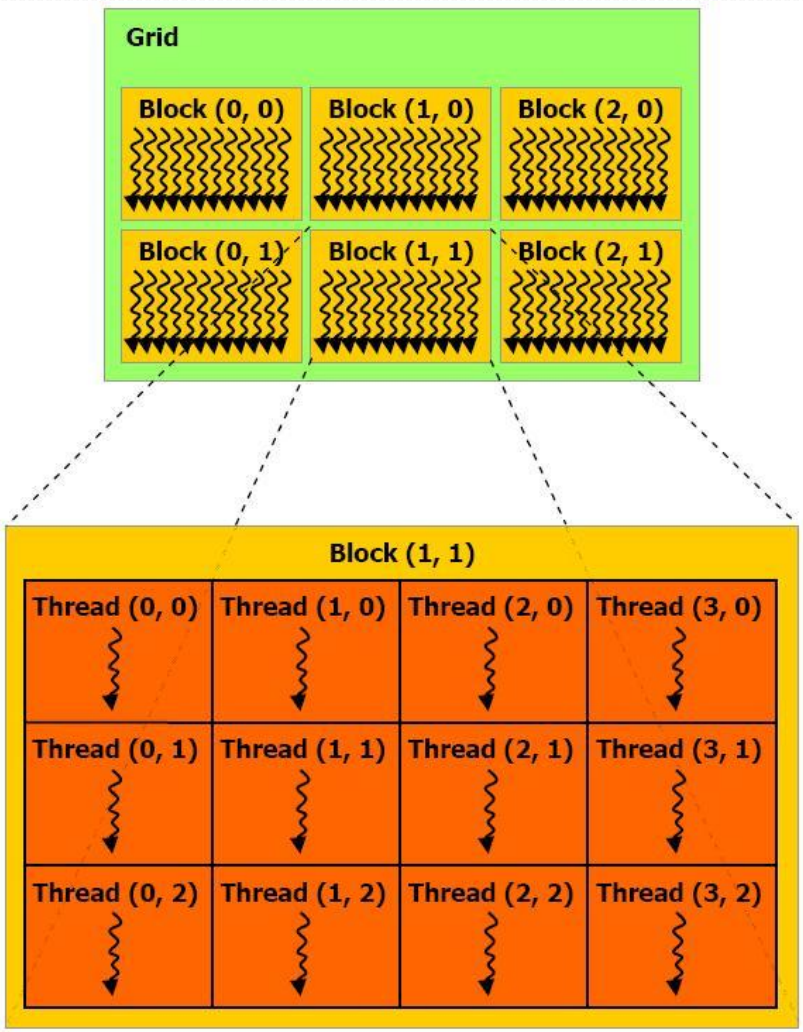
- GPU работает по методу **SPMD** - единая программа, множество данных
  - Задается программа (**CUDA kernel**)
  - Запускается множество нитей (**CUDA grid**)
  - Каждая нить выполняет копию программы над своими данными

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)

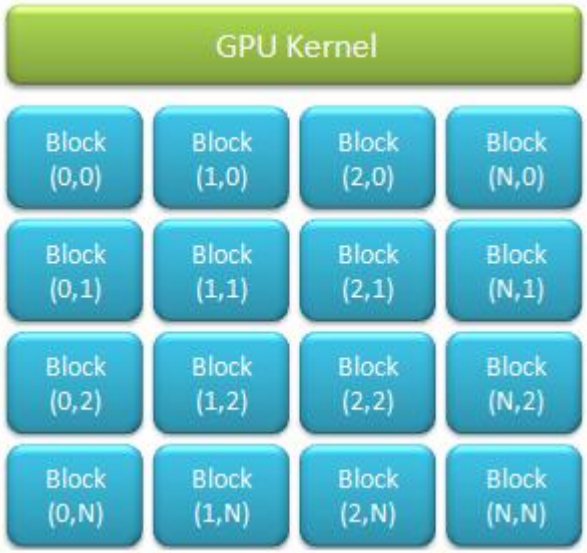
CUDA Grid

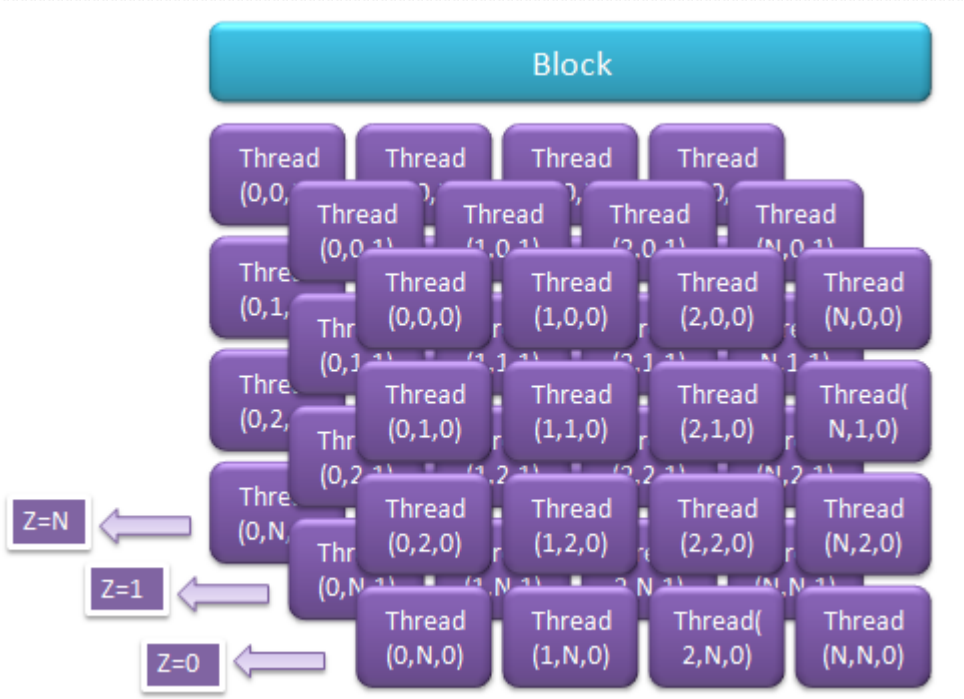
- Грид задаётся количеством блоков по x,y,z (размер грида в блоках) и размерами каждого блока по x,y,z
- Ели по z размер грида и блоков равен единице, то получаем плоскую прямоугольную сетку нитей



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

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 – 1)  /  512 + 1
```

- 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  - индексты блока в гриде
dim3 blockDim  - размеры блоков в нитях
dim3 gridDim   - размеры грида в блоках
```

- Линейный индекс нити в гриде:

```
int gridSizeX = blockDim.x*gridDim.x;
int gridSizeAll = gridSizeX * gridSizeY * gridSizeZ
int threadLinearIdx =
    (threaIdx.z * gridSizeY + threadIdx.y) * gridSizeX +
                                            threadIdx.x
```

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

Запуск ядра

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

Копирование результата на хост

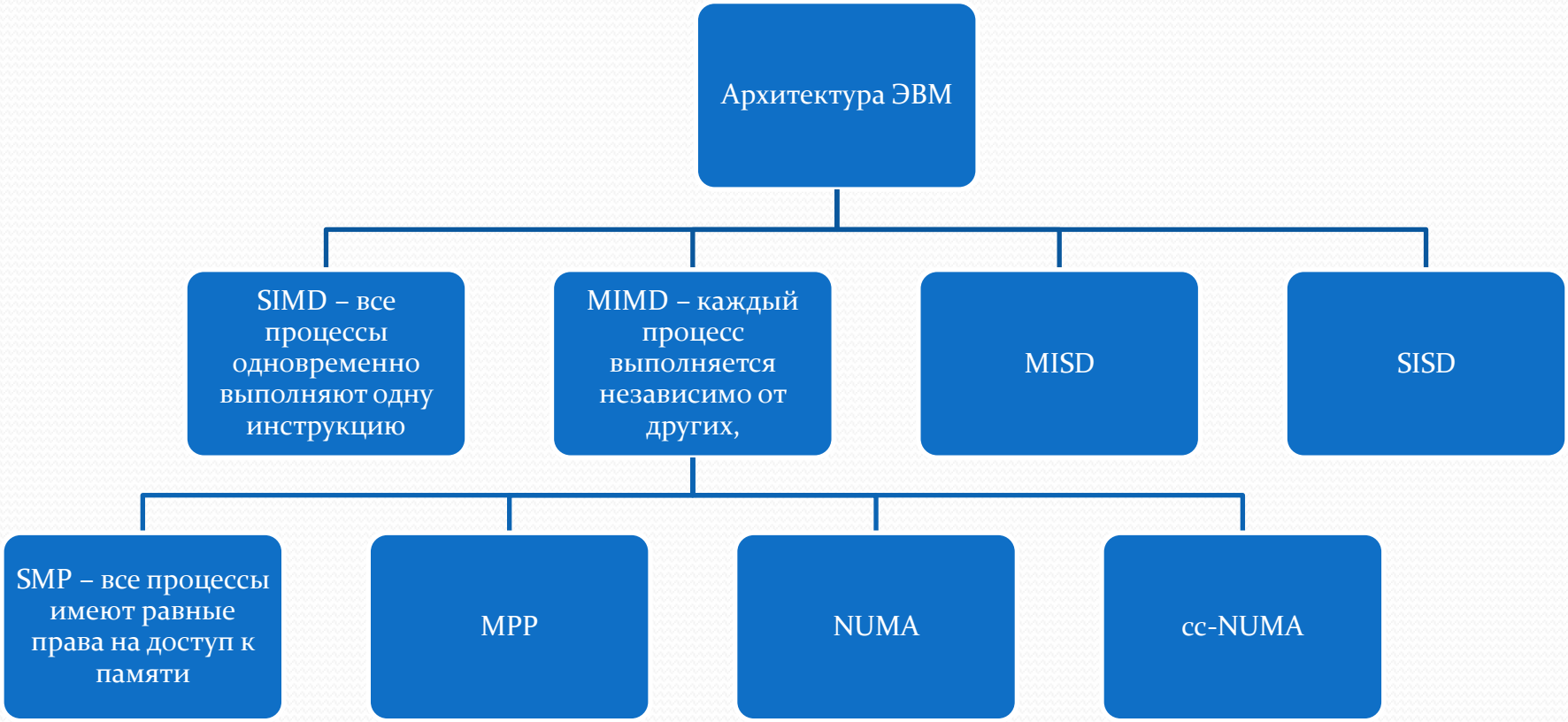
```
cudaFree(inputDataOnDevice);
cudaFree(resultOnDevice);
```

Освободить память

Модель исполнения SIMT

Как реализовать выполнение миллионов нитей на имеющейся архитектуре?

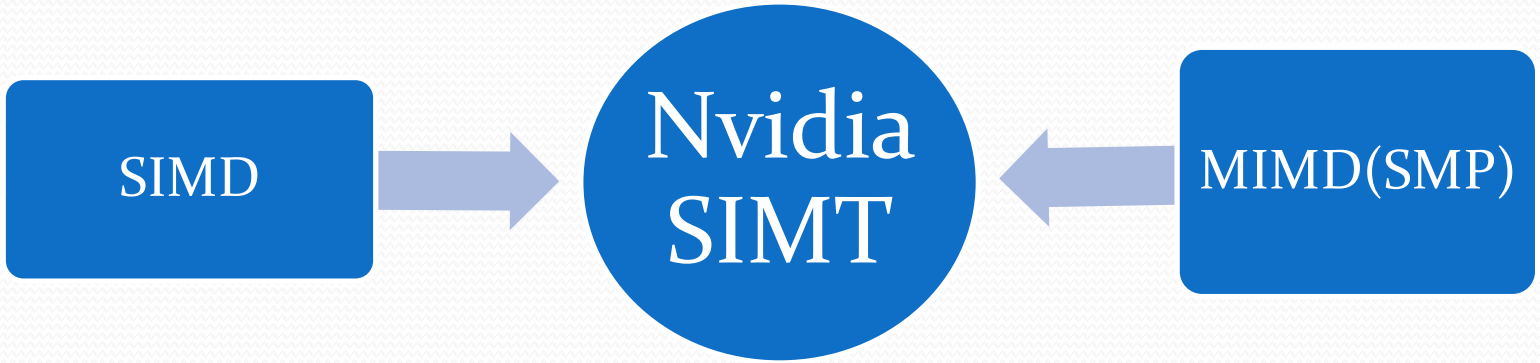
CUDA и классификация Флинна



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

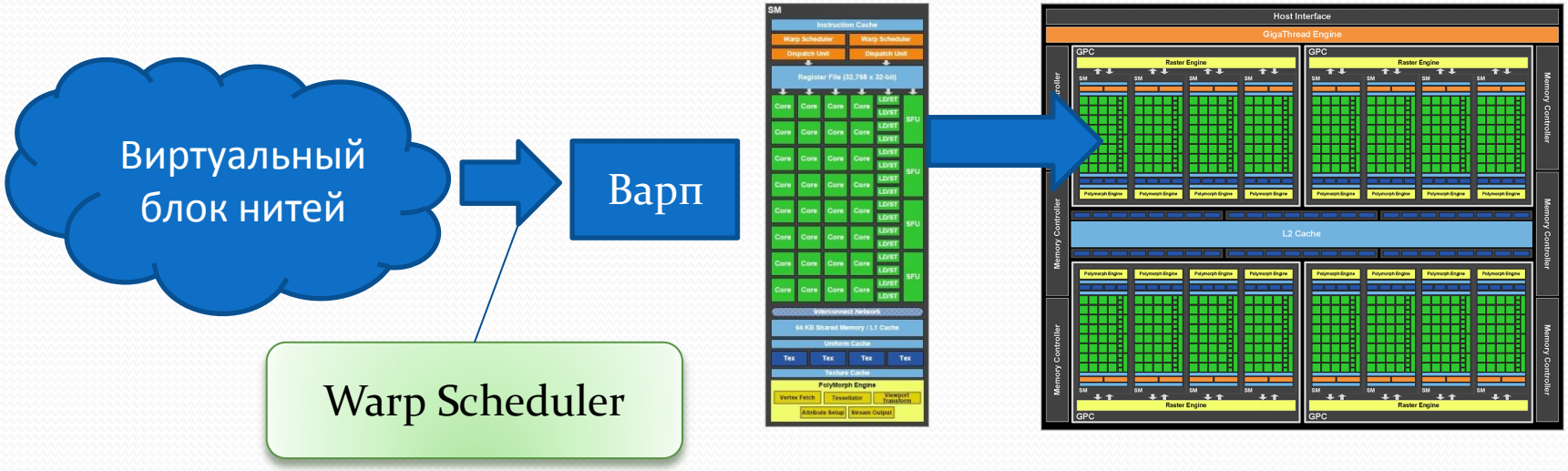
- Нити не мигрируют между блоками. Каждая нить находится в своём блоке с начала выполнения и до конца.

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 мультироцесссорам 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



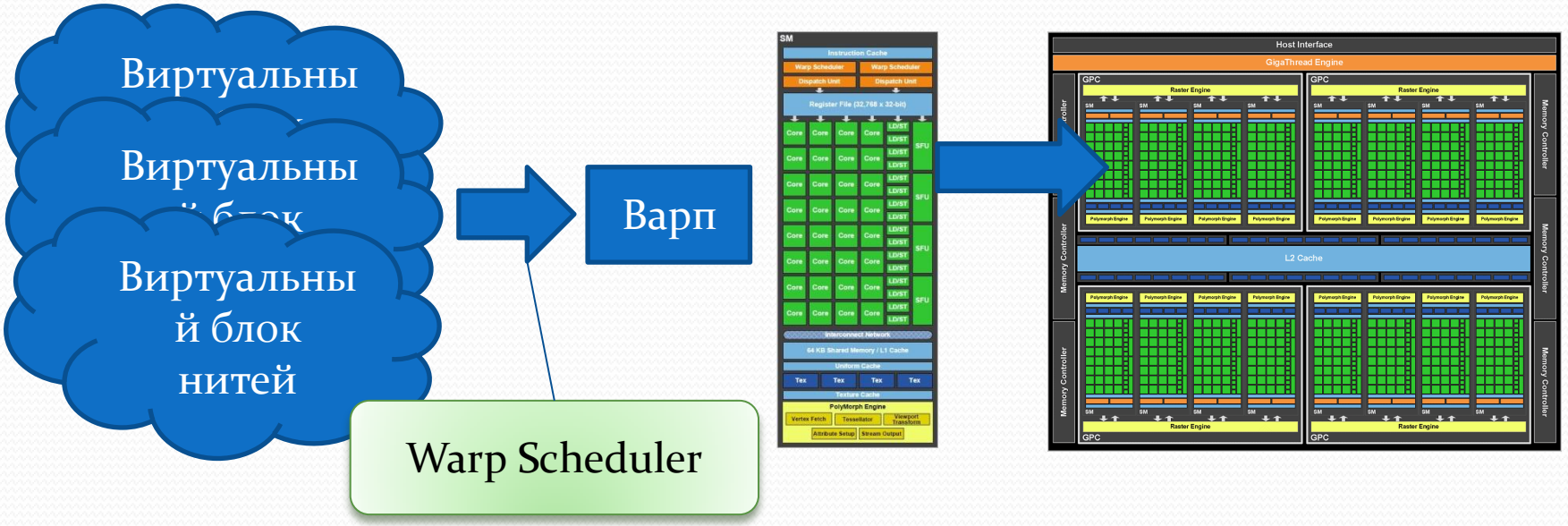
Ветвление (branching)

- 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

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 (Occupancy)

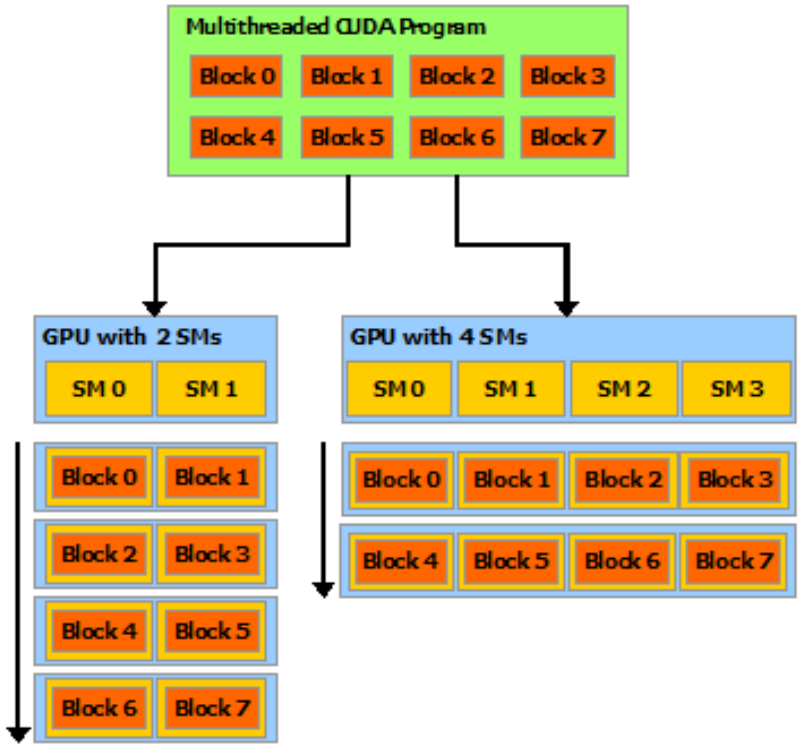
- Чем больше нитей активно на мультипроцессоре, тем эффективнее используется оборудование
  - Блоки по 1024 нити – 1 блок на SM, 1024 нити, 66% от максимума
  - Блоки по 100 нитей – 8 блоков на SM, 800 нитей, 52%
  - Блоки по 512 нитей – 3 блока на SM, 1536 нитей, 100%

SIMT и глобальная синхронизация

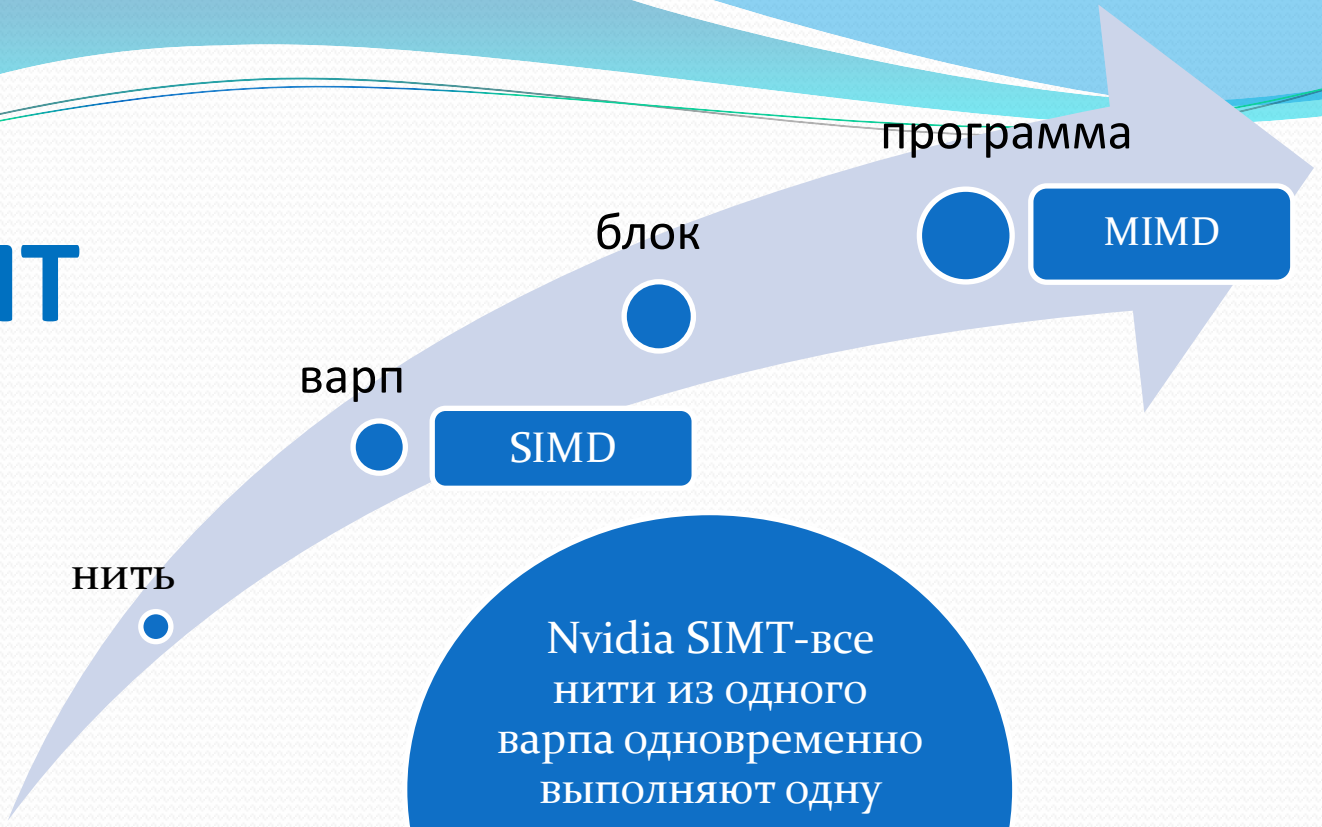
- В общем случае, из-за ограничений по числу нитей и блоков на одном SM, не удаётся разместить сразу все блоки программы на GPU
  - Часть блоков ожидает выполнения
    - Поэтому в общем случае невозможна глобальная синхронизация
  - Блоки выполняются по мере освобождения ресурсов
    - Нельзя предсказать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

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

Выводы

Хорошо распараллеливаются на GPU задачи, которые:

- 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