# Технология CUDA

- Графические процессоры для расчетов общего назначения: история развития и технологии
- Технология CUDA для программирования гибридных систем ЦПУ-ГПУ

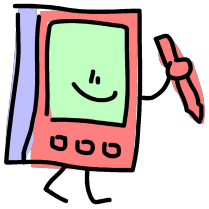
# Этапы эволюции графических процессоров (1)

**ГПУ 1го поколения** (середина 90х) – «ответ» на возрастающее потребление вычислительных ресурсов компьютерными играми.

Специализированные процессоры для ускорения операций с 3-мерной графикой предназначались для построения 2-мерных изображений 3-мерных сцен в режиме реального времени. Аппаратная реализация алгоритмов и аппаратное распараллеливание. Принимали от ЦПУ на вход описание 3-мерной сцены в виде массивов вершин и треугольников, а также параметры наблюдателя, и строили по ним на экране 2-мерное изображение сцены для этого наблюдателя (рендеринг).

Поддерживалось отсечение невидимых граней, задание цвета вершин и интерполяционная закраска, текстуры объектов и вычисление освещенности без учета теней. Тени можно было добавить при помощи алгоритмов расчета теней.

Первые ГПУ обеспечивали растеризацию (перевод треугольников в массивы пикселей), поддержку буфера глубины, наложения текстур и альфа-блендинга (эффект «прозрачности»). Эти (относительно простые) задачи ГПУ выполняли быстро, существенно обгоняя ЦПУ, поскольку ГПУ мог одновременно обрабатывать сразу много пикселов. Это и привело к широкому распространению графических ускорителей для 3D-графики.



# Этапы эволюции графических процессоров (2)

**ГПУ 2го поколения** (2001-2005). Возможность программирования ГПУ. Изначально фиксированный алгоритм вычисления освещенности и преобразования координат вершин заменен на алгоритм, задаваемый пользователем. Возможность писать программы на специальном ассемблере (**шейдеры**, от англ. shade – закрашивать) для вычисления цвета пикселя на экране. Программа выполнялась параллельно для каждой вершины.

Первые шейдеры не могли иметь длину больше 20 команд, не поддерживались команды переходов, вычисления только с фиксированной точкой. Позже появились высокоуровневые шейдерные языки и первые приложения ГПУ для высокопроизводительных вычислений.

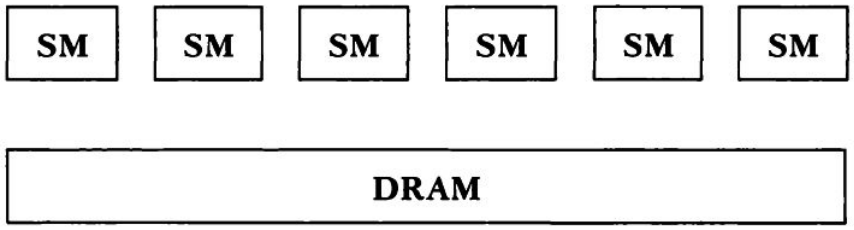
Складывается направление ОВГПУ (GPGPU – general purpose graphic processor units). Для программирования ГПУ предложен подход потокового программирования, предполагающий разбиение программы на относительно небольшие этапы (ядра, kernel), которые обрабатывают элементы потоков данных. Ядра отображаются на шейдеры, а потоки данных – на текстуры в ГПУ.

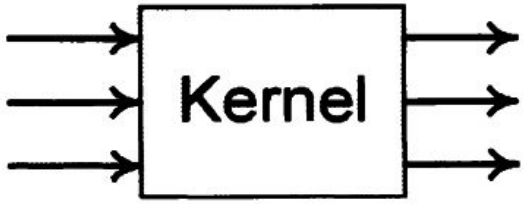
# Этапы эволюции графических процессоров (3)

**ГПУ 3го поколения** (с 2005) характеризуются расширенными возможностями программирования. Появляются операции ветвления и циклов, что позволяет создавать более сложные шейдеры. Поддержка 32-битных вычислений с плавающей точкой становится повсеместной, что способствует активному росту направления ОВГПУ.

К этому времени ГПУ – мощные SIMD-процессоры, способные одновременно выполнять одну и ту же операцию на множеством данных: получая на вход поток однородных данных, ГПУ параллельно обрабатывает их и порождает выходной поток.

Каждый потоковый мультипроцессор (Streaming multiprocessor) в составе ГПУ способен одновременно выполнять более 1000 нитей



# Этапы эволюции графических процессоров (4)

OpenGLv2.0: поддержка высокоуровневого шейдерного языка GLSL. Производительность ГПУ на реальных задачах достигает сотен Гфлопс. Поддержка целочисленных операций, а также операций с двойной точностью.

Инструментарий, позволяющий взаимодействовать с ГПУ напрямую, минуя уровень интерфейса программирования 3D-графики

Создание специализированных средств программирования для ОВГПУ (CUDA). Потоковые библиотеки программирования ГПУ (RapidMind, Accelerator). Первые коммерческие применения ОВГПУ.

Этот этап продолжается по настоящее время.

Сегодня графический ускоритель – гибко программируемый массивно-параллельный процессор с высокой производительностью и пропускной способностью памяти, востребованный в решении целого ряда вычислительно трудоемких задач: проблемы физического моделирования, инженерного анализа, финансовой математики + «машинное зрение», обработка изображений, высококачественная визуализация и др.

# Почему ГПУ ? (1)

Почему ОВГПУ активно развивается, хотя программирование ГПУ существенно отличается от традиционного программирования ЦПУ?

(1) На данный момент ГПУ обеспечивают максимальное соотношение производительности к цене по сравнению с другими решениями. Производительность наращивается существенно быстрее по сравнению с ЦПУ. Последние поколения ГПУ получили унифицированную архитектуру, в которой специализированные блоки заменены на универсальные программируемые процессоры, что положительно сказалось на общей производительности.

(2) ЦПУ всегда были ориентированы на максимально быстрое исполнение последовательного кода, в то время как ГПУ изначально создавались для решения задач визуализации компьютерных сцен, характеризующихся высокой степенью параллелизма на всех стадиях; соответственно ГПУ ориентированы на одновременное исполнения большого числа относительно простых однотипных блоков и на повышение производительности за счет увеличение числа этих блоков.

# Средства программирования ГПУ

- Шейдерные языки (OpenGL, OpenAL) позволяют лаконично описывать некоторые алгоритмы.
- Специализированные средства от производителей (Compute Unified Driver Architecture, CUDA) от NVIDIA и Stream Computing от ATI\AMD
- **OpenCL – первый открытый межплатформенный стандарт** для параллельных вычислений на современных процессорах (включая многоядерные и графические), доступных в ПК, наладонных устройствах и серверах. В OpenCL входят язык программирования для параллельных вычислений на современных процессорах (включая многоядерные и графические), доступных в ПК, наладонных устройствах и серверах. В OpenCL входят язык программирования, который базируется на стандарте C99 для параллельных вычислений на современных процессорах (включая многоядерные и графические), доступных в ПК, наладонных устройствах и серверах. В OpenCL входят язык программирования, который базируется на стандарте C99, и интерфейс программирования приложений (*API*).

Ключевыми отличиями используемого в OpenCL языка от Си являются:

- не поддерживаются указатели на функции, рекурсия, битовые поля,

# Технология CUDA: зачем? (1)

**CUDA** (*Compute Unified Device Architecture*) — технология GPGPU (*General-Purpose computing on Graphics Processing Units*), которая даёт возможность организации доступа к набору инструкций графического ускорителя и управления его памятью при организации параллельных вычислений.

- Позволяет программистам реализовывать на упрощённом Си-подобном языке алгоритмы, выполнимые на графическихПозволяет программистам реализовывать на упрощённом Си-подобном языке алгоритмы, выполнимые на графических_Позволяет программистам реализовывать на упрощённом Си-подобном языке алгоритмы, выполнимые на графических процессорахПозволяет программистам реализовывать на упрощённом Си-подобном языке алгоритмы, выполнимые на графических процессорах ускорителей GeForceПозволяет программистам реализовывать на упрощённом Си-подобном языке алгоритмы, выполнимые на графических процессорах ускорителей GeForce 8го поколения и старше фирмы Nvidia.
- Технология CUDA разработана компанией nVidia.
- Фактически CUDA позволяет включать в текст C\C++ программы

# Технология CUDA: зачем? (2)

Современные видеочипы содержат сотни математических исполнительных блоков, эта мощь может использоваться для значительного ускорения множества вычислительно интенсивных приложений.

Конечно, максимальная скорость достигается лишь в ряде удобных задач и имеет ограничения, но такие устройства начали широко применяться в сферах, для которых они изначально не предназначались.

В середине 2000х появились первые технологии неграфических расчётов общего назначения GPGPU (General-Purpose computation on GPUs).

Для этого использовались графические API: OpenGL и Direct3D, когда данные к видеочипу передавались в виде текстур, а расчётные программы загружались в виде шейдеров.

Программа писалась сразу на двух языках: традиционный язык для ЦП: подготовка и передача данных в память ГПУ, запуск кода на ГПУ (kernel). Код для ГПУ пишется на языке шейдеров.

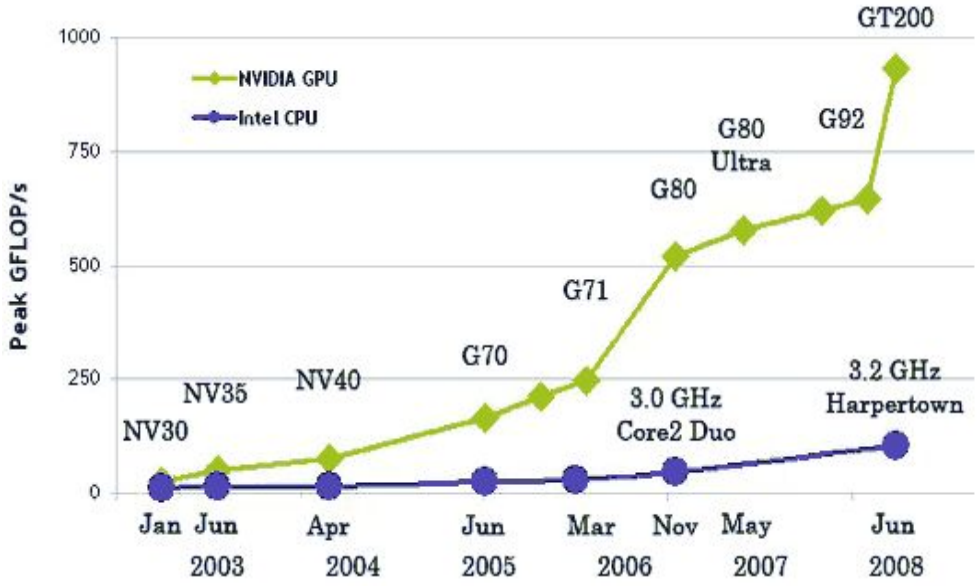
Недостатки: сравнительно высокая сложность программирования, низкая скорость обмена данными между CPU и GPU и другие ограничения.

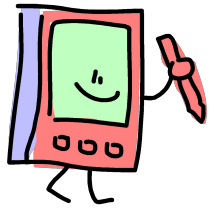
# Технология CUDA: зачем? (3)

Именно поэтому компания NVIDIA выпустила платформу CUDA — Си-подобный язык программирования со своим компилятором и библиотеками для вычислений на GPU.

За счет того, что программа пишется для системы ЦПУ – ГПУ на едином языке – процесс создания кода значительно упрощается, что способствовало активному распространению вычислений общего назначения на ГПУ.

Другой фактор – технологические ограничения роста тактовой частоты и числа ядер в «традиционных» микропроцессорах.



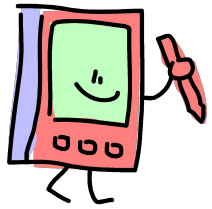


# CUDA: общие положения (1)

- **GPU (device) – сопроцессор для CPU (хоста)**
- **У GPU есть собственная память (device memory),** имеющая сложную иерархическую структуру. DRAM - *dynamic random access memory* (Динамическая память с произвольным доступом — тип энергозависимой полупроводниковой памяти. При отсутствии подачи электроэнергии происходит разряд конденсаторов и память обнуляется.
- **GPU способен одновременно обрабатывать множество процессов (threads) данных одним и тем же алгоритмом**
- **Для осуществления расчётов при помощи GPU хост должен осуществить запуск вычислительного ядра (kernel),** который определяет конфигурацию GPU в вычислениях и способ получения результатов (алгоритм)
- **GPU обладает возможностью параллельного выполнения огромного количества нитей.** По сравнению с ЦПУ-нитями. ГПУ-нити

- обладают крайне невысокой стоимостью создания.

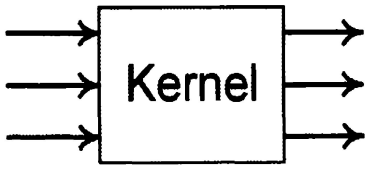
- для эффективной загрузки необходимо использовать много тысяч нитей



# CUDA: общие положения (2)

**Общая схема кода:**

- Выделяется общая память на ГПУ
- Копируются необходимые данные из памяти ЦПУ в память ГПУ
- Осуществляется запуск ядра (или последовательный запуск нескольких ядер)
- Результаты вычислений копируются в память ЦПУ
- Освобождается память ГПУ.




**Основные характеристики CUDA:**

- унифицированное программно-аппаратное решение для параллельных вычислений на видеочипах NVIDIA;
- стандартный язык программирования Си;
- стандартные библиотеки – в том числе для численного анализа FFT (быстрое преобразование Фурье) и BLAS (линейная алгебра);
- оптимизированный обмен данными между CPU и GPU;
- поддержка 32- и 64-битных OS (WindowsXP,WindowsVista,Linux,MacOSX)

# **CUDA:** Основы создания программ (1)

- Первый шаг при переносе существующего приложения на CUDA – определение участков кода, являющихся «бутылочным горлышком», тормозящим работу.
- Если среди таких участков есть подходящие для быстрого параллельного исполнения, они переносятся на C-расширения CUDA для выполнения на GPU.
- Программа компилируется при помощи поставляемого NVIDIA компилятора **nvcc**, который генерирует код и для CPU, и для GPU.
- При исполнении кода, CPU выполняет свои порции кода, а GPU выполняет CUDA-код (kernel) с наиболее тяжелыми параллельными вычислениями. В ядре определяются операции, которые будут исполнены над данными.
- GPU получает ядро и создает его копии для каждого элемента данных. Эти копии называются потоками (thread). Каждый поток содержит счётчик, регистры и состояние. Потоки выполняются группами по 32 штуки, называемыми **warp**'ы. **Warp**'ам назначается исполнение на определенных потоковых мультипроцессорах (SM)

# **CUDA:** Основы создания программ (2)

Каждый SM состоит из восьми и более ядер — потоковых процессоров, которые выполняют одну инструкцию за один такт.

SM – не традиционный многоядерный процессор, он приспособлен для многопоточности, поддерживая до 32 warp'ов единовременно.

Если проводить аналогию с CPU, это похоже на одновременное исполнение 32 программ и переключение между ними каждый такт без потерь тактов. Ядра же CPU поддерживают единовременное выполнение одной программы и переключаются на другие с задержкой в сотни тактов.

Типичный (но не обязательный) шаблон решения задач:

- задача разбивается на подзадачи;
- входные данные делятся на блоки, чтобы вмещались в разделяемую память;
- каждый блок данных обрабатывается блоком потоков;
- над данными в разделяемой памяти проводятся вычисления;
- результаты копируются из разделяемой памяти обратно в глобальную.

# **CUDA:** Модель памяти (1)

- Модель памяти в CUDA отличается возможностью побайтной адресации, с поддержкой gather и scatter.
- Доступно до 1024 регистров на каждый потоковый процессор, которых до 1024 штук. Доступ к ним очень быстрый, хранить в них можно 32-битные целые или числа с плавающей точкой.
- Каждый поток имеет доступ к следующим типам памяти:

**Глобальная память:** самый большой объём памяти, доступный для всех МП на видеочипе (размер от 256Мбайт до 4Гбайт). Высокая пропускная способность, но очень большие задержки (несколько сот тактов). Не кэшируется.

**Локальная память:** небольшой объём памяти, к которому имеет доступ только один потоковый процессор. Относительно медленная, как и глобальная.

**Разделяемая память:** (16 килобайт) блок памяти с общим доступом для всех потоковых процессоров в МП. Весьма быстрая, такая же, как регистры. Обеспечивает взаимодействие потоков, управляется разработчиком кода напрямую, имеет низкие задержки.

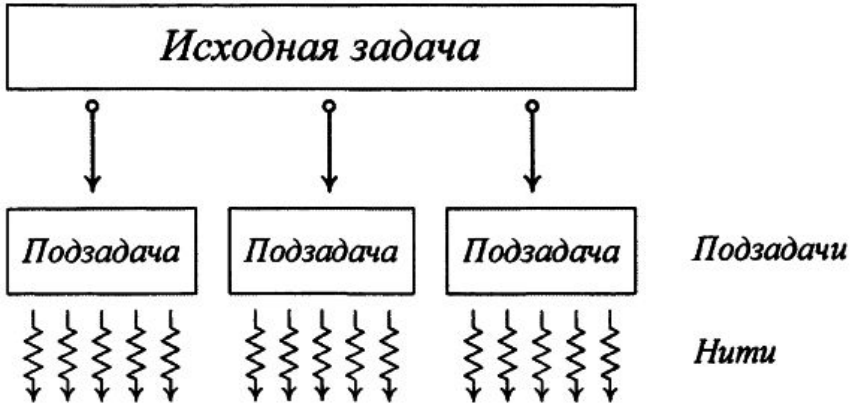
# **CUDA**: Модель памяти (2)

**Память констант:** область памяти (64 килобайт), доступная только для чтения всеми МП. Кэшируется. Довольно медленная — задержка в несколько сот тактов при отсутствии нужных данных в кэше.

**Текстурная память:** блок памяти, доступный для **чтения** всеми МП. Кэшируется. Медленная, как глобальная — сотни тактов задержки при отсутствии данных в кэше.

- Естественно, что глобальная, локальная, текстурная и память констант — это физически одна и та же память, известная как локальная видеопамять видеокарты. Их отличия – в различных алгоритмах кэширования и моделях доступа.
- CPU может обновлять и запрашивать только внешнюю память: глобальную, константную и текстурную.
- При разработке CUDA приложений нужно помнить о разных типах памяти, о том, что локальная и глобальная память не кэшируются и задержки при доступе к ней гораздо выше, чем у регистровой памяти, так как она физически находится в отдельных микросхемах.

# CUDA: спецификаторы




Спецификаторы функций

__device__ (выполняется на ГПУ, вызывается из ГПУ)
__global__ (выполняется на ГПУ, вызывается из ЦПУ) → kernel
__host__ (выполняется на ЦПУ, вызывается из ЦПУ)

Спецификаторы __host__ и __device__ могут быть использованы вместе.

Ограничения на функции, выполняемые на GPU:

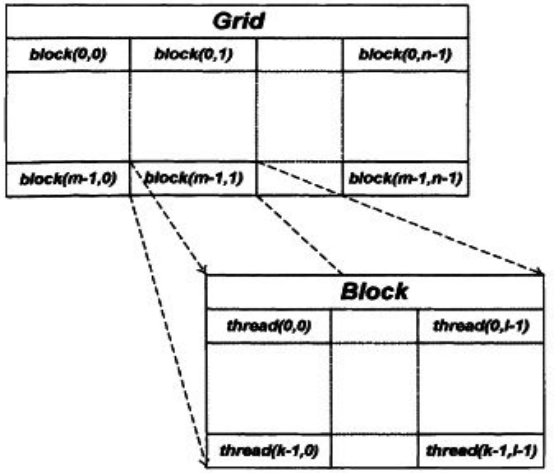
- Не поддерживается рекурсия
- Нельзя брать их адрес
- Не поддерживаются static переменные внутри функции
- Не поддерживается переменное число входных аргументов

Спецификаторы переменных: __device__, __constant__, __shared__

Ограничения:

- Спецификаторы не могут применяться к полям структуры
- Специфицированные переменные не могут быть объявлены extern
- Запись в __constant__ осуществляет только ЦПУ

# CUDA: добавленные типы и функции




**Добавленные типы: 1\2\3\4-мерные векторы базовых типов.**

Специальные переменные:

gridDim (размер сетки); blockDim (размер блока); warpsize (размер warp'a);
blockIdx (индекс текущего блока); threadIdx (индекс текущей нити в блоке)

**Директива вызова ядра: Name <<< … >>> (args)**

- Размерность и кол-во блоков в сетке
- Размерность и кол-во нитей в блоке
- Дополнительный объем разделяемой памяти (необязательно)
- Поток, в котором должен произойти вызов (по умолчанию 0)

Добавленные функции

Float-аналоги стандарных функций
«быстрые» функции пониженной точности
Функции для разных способов округления
«быстрые» функции для целочисленной арифметики

Встроенная функция барьерной синхронизации _synchronthreads( )

# Пример: поэлементное суммирование двух векторов

```
__global__ void kernel (float *a, float *b, float *c)
{
// глобальный индекс нити
Int idx = theadIdx.x + blockIdx.x * blockDim.x
C[idx] = a[idx] + b[idx];
}

Void sum (float *a, float *b, float *c, int n)
{
int numBytes = n*sizeof(float)
float *adev=NULL;
float *bdev=NULL;
float *cdev=NULL;
//выделяем память  на GPU; копируем
    adev,bdev в память GPU
cudaMalloc ( void**)&adev, numbytes);
cudaMalloc ( void**)&bdev, numbytes);
cudaMalloc ( void**)&cdev, numbytes);
cudaMemcpy (adev, a, numbytes,
    cudaMemcpyHostToDevice);
cudaMemcpy (bdev, b, numbytes,
    cudaMemcpyHostToDevice);
```

```
//конфигурация запуска n нитей
Dim3 threads = dim3(512,1)
Dim3 blocks = dim3(n/threads.x,1)
//вызываем ядро
sumKernel <<<blocks, threads>>>
          (adev, bdev, cdev);
//копируем cdev в память CPU
cudaMemcpy (cdev, c, numbytes,
cudaMemcpyDeviceToHost);
//освобождаем выделенную
//          память GPU
cudaFree (adev);
cudaFree (bdev);
cudaFree (cdev);
}
```

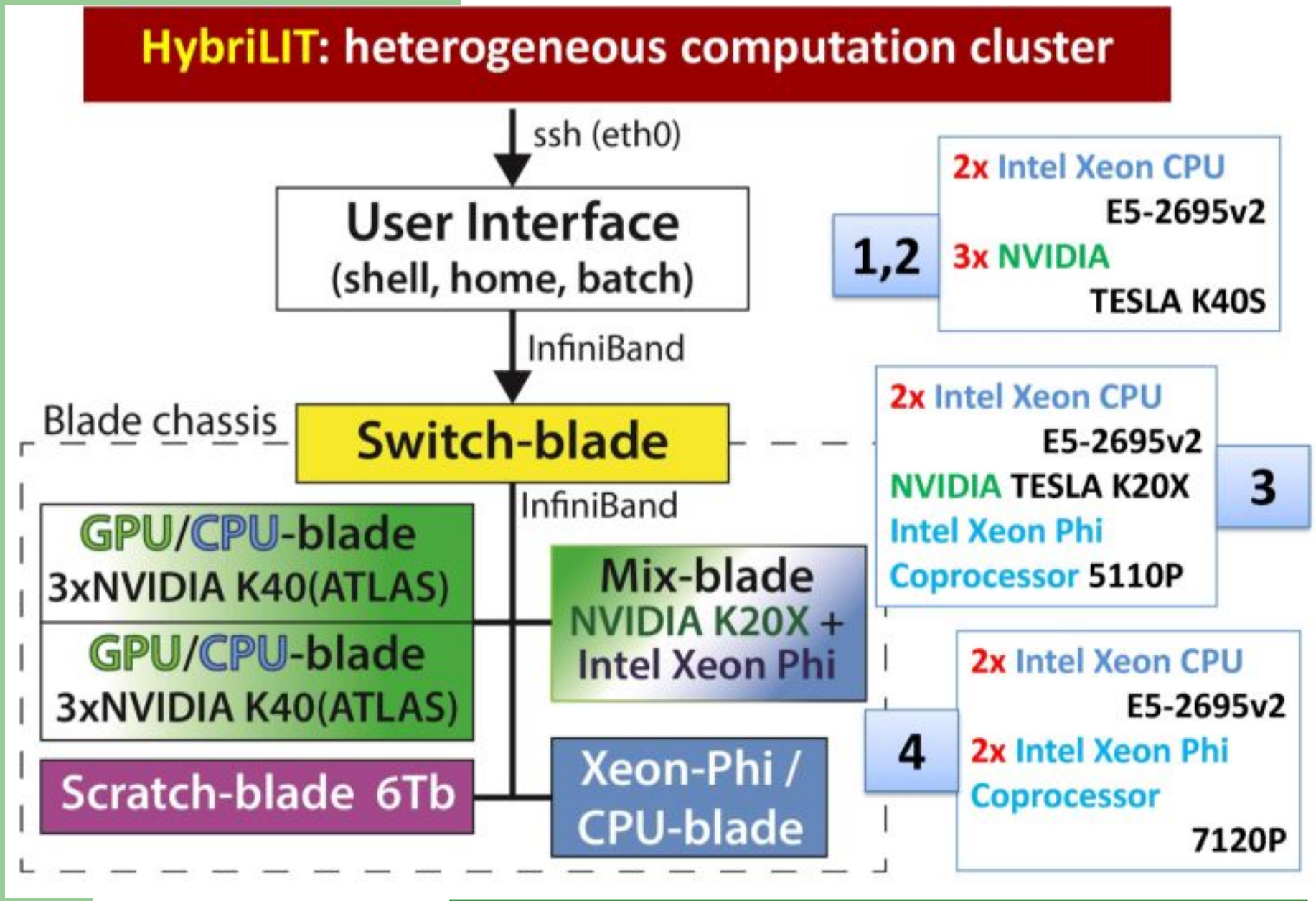
# Resumé

Cuda строится на концепции, что GPU выступает в роли массивно-параллельного сопроцессора к CPU. Cuda-код задействует как CPU, так и GPU. При этом GPU-ядро (kernel) выполняется как набор большого числа одновременно выполняющихся нитей (потоков, threads).

**Преимущества:** простота; наличие хорошей документации; набор готовых инструментов; набор готовых библиотек; кроссплатформенность; полностью бесплатный продукт (developer.nvidia.com)

Один из немногочисленных **недостатков** CUDA — слабая переносимость. Эта архитектура работает только на видеочипах этой компании, да ещё не на всех, а начиная с серии GeForce 8,9 и соответствующих Quadro и Tesla.

По данным NVIDIA, в мире 90млн CUDA-совместимых видеочипов. Но и конкуренты предлагают свои решения, отличные от CUDA. Так, у AMD есть CTM (AMD Stream Computing), у Intel в будущем будет Ct.




Podgainy D.V., Presentation on 41st PAC, 22/01/2015