Архитектура и программирование массивно-параллельных вычислительных систем

Лектор:

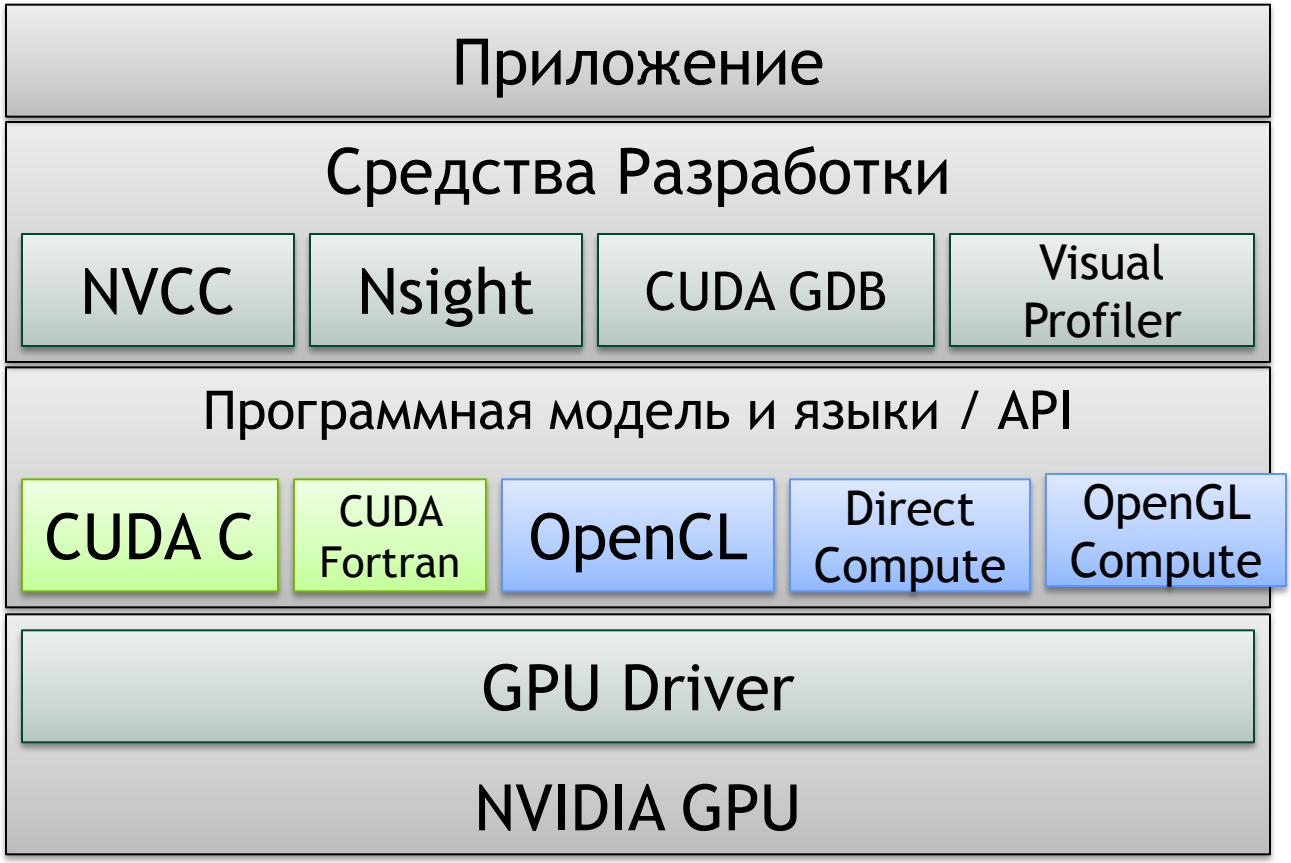
Боресков А.В. (ВМиК МГУ)

План

- CUDA
- Архитектура
- Hello, World
- Несколько слов о курсе
- Дополнительные слайды

Compute Unified Device Architecture (CUDA)

- CUDA – программно-аппаратный стек для программирования GPU



Программная модель CUDA

- Код состоит из последовательных и параллельных частей
- Последовательные части кода выполняются на CPU (*host*)
- Массивно-параллельные части кода выполняются на GPU (*device*)
- GPU
  - Является сопроцессором к CPU (*host*)
  - Имеет собственную память (DRAM)
  - Выполняет одновременно ***очень много*** нитей

Программная модель CUDA

- Параллельная часть кода выполняется как большое количество нитей (*threads*)
- Нити группируются в блоки (*blocks*) фиксированного размера
- Блоки объединяются в сеть блоков (*grid*)
- Ядро выполняется на сетке из блоков
- Каждая нить и блок имеют свой уникальный идентификатор

Программная модель CUDA

- Компромисс между желанием дать возможность каждой нити взаимодействовать с каждой и аппаратными возможностями
- Исходная задача разбивается на набор независимо решаемых подзадач
- Каждая подзадача решается параллельно блоков взаимодействующих нитей

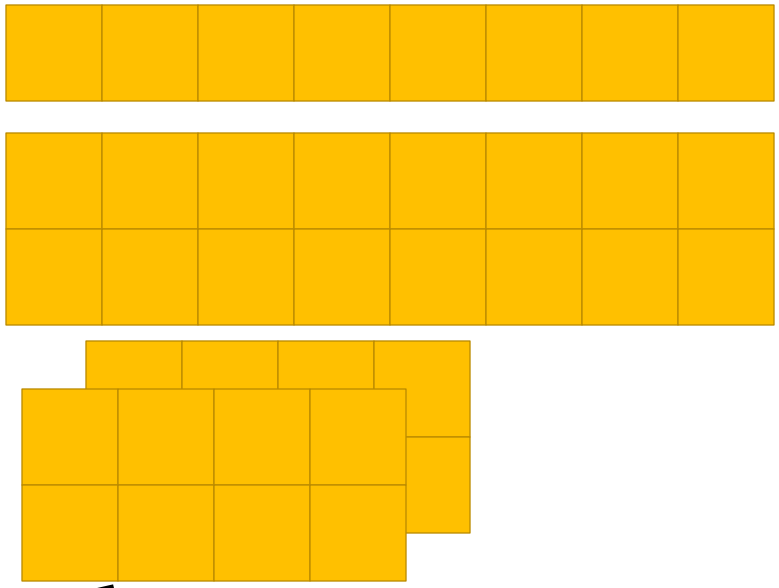
Программная модель CUDA

- Десятки тысяч нитей

```
for ( int ix = 0; ix < nx; ix++ )
{
   pData[ix] = f(ix);
}

for ( int ix = 0; ix < nx; ix++ ){
  for ( int iy = 0; iy < ny; iy++ )
  {
     pData[ix+iy*nx] = f(ix)*g(iy);
  }
}

for ( int ix = 0; ix < nx; ix++ ){
  for ( int iy = 0; iy < ny; iy++ ){
    for ( int iz = 0; iz < nz; iz++ )
    {
       pData[ix+(iy+iz*ny)*nx] = f(ix)*g(iy)*h(iz);
    }
  }
}
```

Программная модель CUDA

- Нити в CUDA объединяются в блоки:
  - 1D топология блока
  - 2D топология блока
  - 3D топология блока



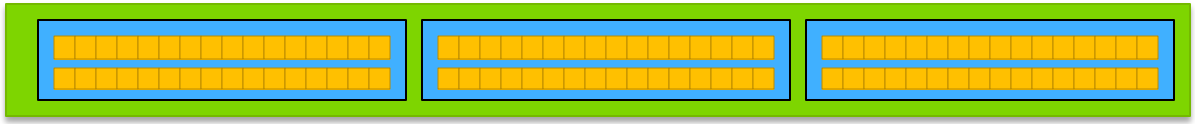
- Общее кол-во нитей в блоке ограничено
- В текущем HW это 1024 нитей

Программная модель CUDA

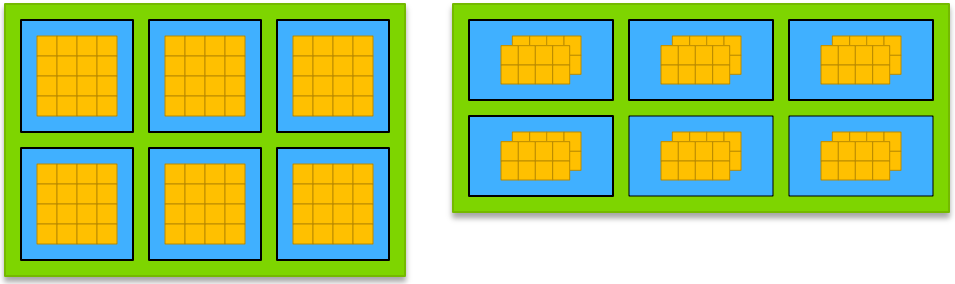
- Блоки могут использовать *shared* память
  - Нити могут обмениваться общими данными
- Внутри блока потоки могут синхронизоваться
  - Барьерная синхронизация

Программная модель CUDA

- Блоки потоков объединяются в сеть (*grid*) блоков потоков
  - 1D топология сетки блоков



  - 2D/3D топология сетки блоков



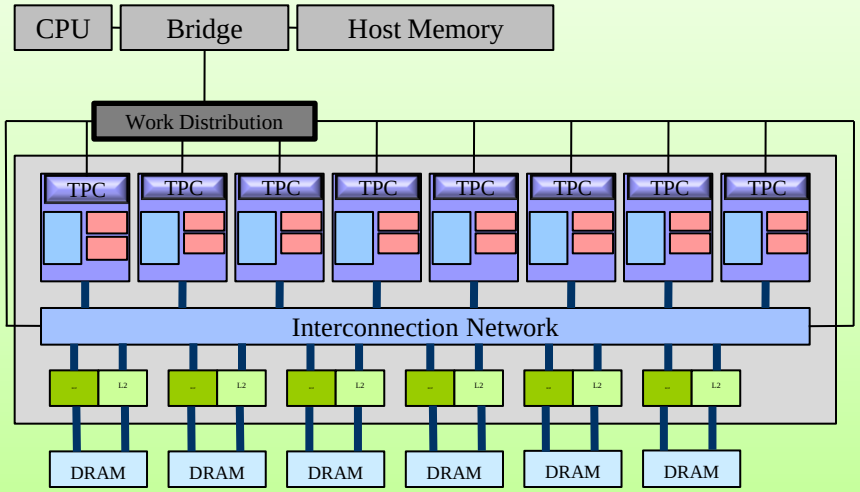
- Блоки в сети выполняются независимо друг от друга

Связь программной модели с HW

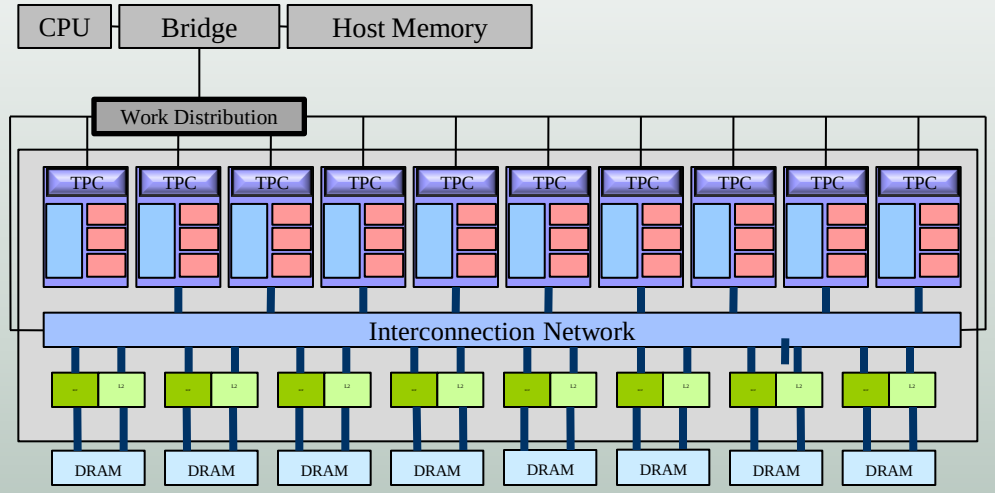
- Блоки могут использовать *shared* память
  - Т.к. блок целиком выполняется на одном SM
  - Объем *shared* памяти ограничен и зависит от HW
- Внутри блока нити могут синхронизоваться
  - Т.к. блок целиком выполняется на одном SM
- Масштабирование архитектуры и производительности

Масштабирование архитектуры Tesla

Tesla 8

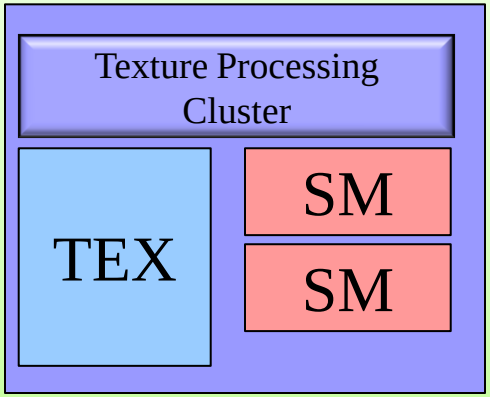


Tesla 10

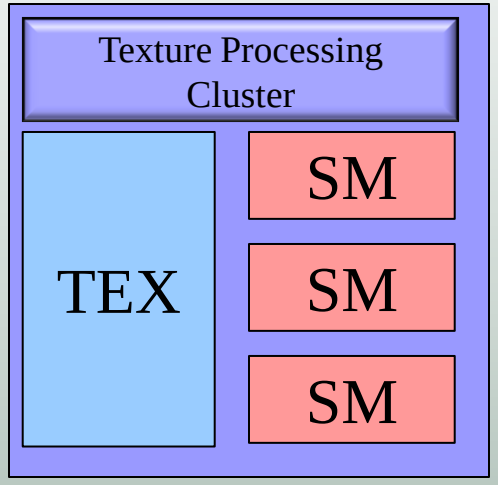


Масштабирование мультипроцессора Tesla 10

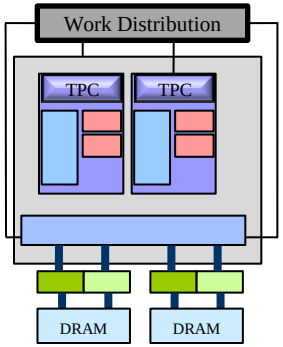
Tesla 8

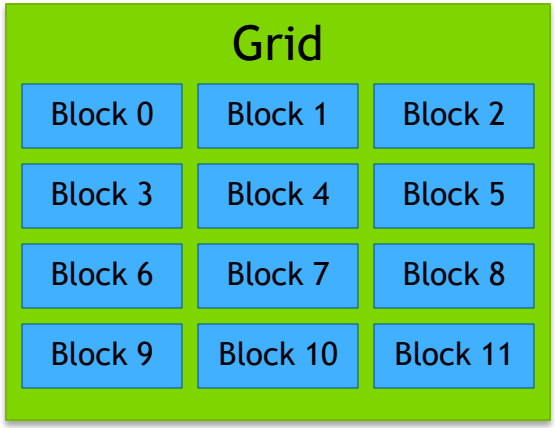


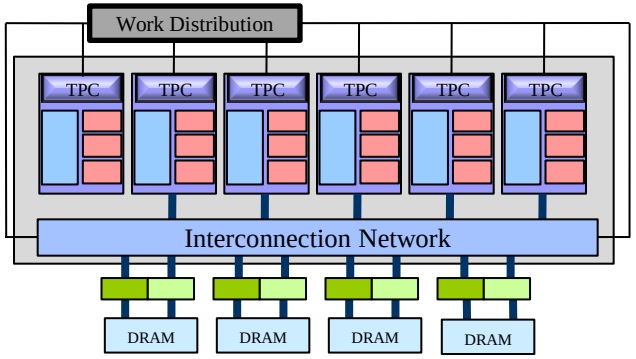
Tesla 10

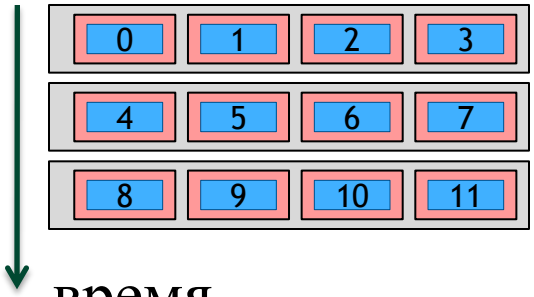


Масштабирование производительности

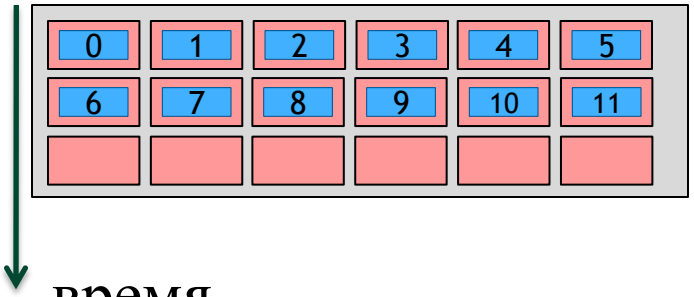








время



время

Легенда:
-блок
-сеть
-SM
-HW планировщик

Связь программной модели с HW

- Очень высокая степень параллелизма
  - Десятки тысяч потоков на чипе
  - Потоки на GPU очень «легкие» (очень дешевое переключение)
  - HW планировщик задач
- Основная часть чипа занята логикой, а не кэшем
- Для полноценной загрузки GPU нужны тысячи потоков
  - Для покрытия латентностей операций чтения / записи
  - Для покрытия латентностей sfu инструкций

Warp

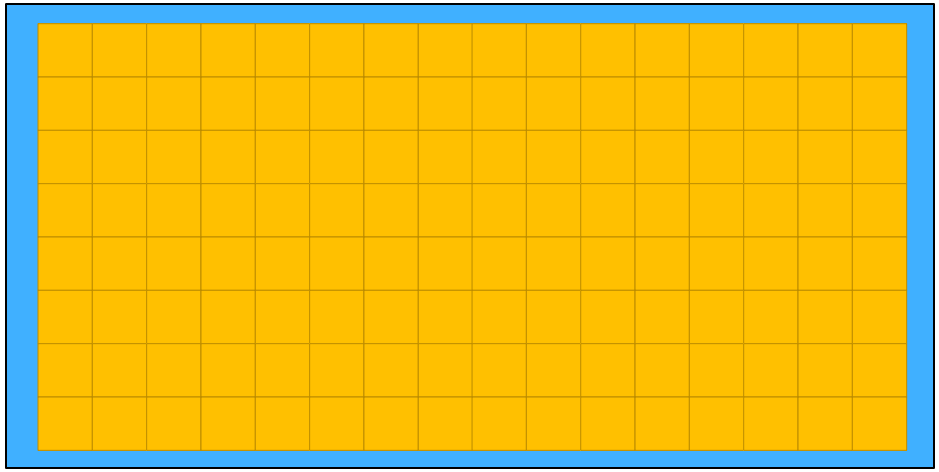
- Все нити, выполняемые на SM перенумеровываются и разбиваются на warp’ы – группы из 32 подряд идущих нитей
- Все нити warp’а считаются выполняемыми физически параллельно и всегда выполняют одну и ту же команду

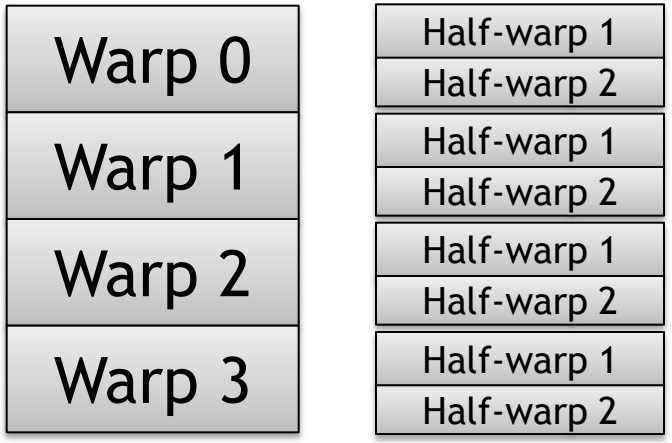
Warp

- SM содержит приоретизированную очередь всех warp’ов и по очереди выполняет по одной команде для каждого готового к выполнению warp’а
- Тем самым переключение warp’ов происходит на каждой команде

Блоки и warp’ы?

- Блоки – абстракция программной модели
- Warp – реальная единица исполнения HW





Single Instruction Multiple Threads (SIMT)

- Параллельно на каждом SM выполняется большое число отдельных нитей (*threads*)
- Нити в пределах одного *warp*'а выполняются физически параллельно (SIMD)
- Разные *warp*'ы могут исполнять разные команды
- Большое число *warp*'ов покрывает латентность

Язык CUDA C

- CUDA C – это расширение языка C/C++
  - спецификаторы для функций и переменных
  - новые встроенные типы
  - встроенные переменные (внутри ядра)
  - директива для запуска ядра из C кода
- Как скомпилировать CUDA код
  - **nvcc** компилятор
  - .cu расширение файла

Язык CUDA C
Спецификаторы

Спецификатор функций

Спецификатор	Выполняется на	Может вызываться из
__device__	device	device
__global__	device	host
__host__	host	host

Спецификатор переменных

Спецификатор	Находится	Доступна	Вид доступа
__device__	device	device	RW
__constant__	device	device / host	R / W
__shared__	device	block	RW / **__syncthreads()**

Язык CUDA C
Спецификаторы

- Спецификатор __global__ соответствует ядру
  - Может возвращать только void
- Спецификаторы __host__ и __device__ могут использоваться одновременно
  - Компилятор сам создаст версии для CPU и GPU
- Спецификаторы __global__ и __host__ не могут быть использованы одновременно

Язык CUDA C
Ограничения

- Ограничения на функции, выполняемые на GPU:
  - Не поддерживаются static-переменные внутри функции
  - Не поддерживается переменное число входных аргументов
  - Не поддерживается RTTI
  - Не поддерживаются exceptions

Язык CUDA C
Ограничения

- Ограничения на спецификаторы переменных:
  - Нельзя применять к полям структуры или union
  - Не могут быть extern
  - Запись в __constant__ может выполнять только CPU через специальные функции
  - __shared__ - переменные не могут инициализироваться при объявлении

Язык CUDA C
Типы данных

- Новые типы данных:
  - 1/2/3/4-мерные вектора из базовых типов
    - (u)char, (u)int, (u)short, (u)long, longlong
    - float, double
  - dim3 – uint3 с нормальным конструкторов, позволяющим задавать не все компоненты
    - Не заданные инициализируются единицей

Язык CUDA C
Встроенные переменные

Сравним CPU vs CUDA C код:

```
float * data;
for ( int i = 0; i < n; i++ )
{
    data [i] = data[i] + 1.0f;
}
```

Пусть nx = 2048
Пусть в блоке 256 потоков

→ кол-во блоков = 2048 / 256 = 8

```
__global__ void incKernel ( float * data )
{                    [ 0 .. 7 ]          [ == 256]          [ 0 .. 255 ]
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  data [idx] = data [idx] + 1.0f;
}
```

Язык CUDA C
Встроенные переменные

- В любом CUDA kernel’е доступны:
  - `dim3 gridDim;`
  - `uint3 blockIdx;`
  - `dim3 blockDim;`
  - `uint3 threadIdx;`
  - `int warpSize;`

dim3 – встроенный тип, который используется для задания размеров kernel’а
По сути – это uint3.

Язык CUDA C
Директивы запуска ядра

- Как запустить ядро с общим кол-во тредов равным nx?

```
float * data;
dim3 threads ( 256 );
dim3 blocks  ( nx / 256 );
incKernel<<<blocks, threads>>> ( data );
```

<<< , >>> угловые скобки, внутри которых задаются параметры запуска ядра

Можно для одномерного случая запустить и так

```
incKernel<<<nx/256, 256>>> ( data );
```

Язык CUDA C
Директивы запуска ядра

- Общий вид команды для запуска ядра

```
incKernel<<<bl, th, ns, st>>> ( data );
```

- *bl* – число блоков в сетке
- *th* – число нитей в блоке
- *ns* – количество дополнительной shared-памяти, выделяемое блоку
- *st* – поток, в котором нужно запустить ядро

Как скомпилировать CUDA код

- NVCC – компилятор для CUDA
  - Основными опциями команды **nvcc** являются:
  - **--use_fast_math** - заменить все вызовы стандартных математических функций на их быстрые (но менее точные) аналоги
  - **-o <outputFileName>** - задать имя выходного файла
- CUDA файлы обычно носят расширение .cu

CUDA “Hello World”

```
#define   N    (1024*1024)
__global__ void kernel ( float * data )
{
  int   idx = blockIdx.x * blockDim.x + threadIdx.x;
  float x   = 2.0f * 3.1415926f * (float) idx / (float) N;
  data [idx] = sinf ( sqrtf ( x ) );
}

int main ( int argc, char *  argv [] )
{
  float a [N];
  float * dev = NULL;
  cudaMalloc ( (void**)&dev, N * sizeof ( float ) );
  kernel<<<dim3((N/512),1), dim3(512,1)>>> ( dev );
  cudaMemcpy ( a, dev, N * sizeof ( float ), cudaMemcpyDeviceToHost );
  cudaFree   ( dev   );
  for (int idx = 0; idx < N; idx++)  printf("a[%d] = %.5f\n", idx, a[idx]);
  return 0;
}
```

CUDA “Hello World”

```
__global__ void kernel ( float * data )
{
   int idx = blockIdx.x * blockDim.x + threadIdx.x; // номер текущей нити
   float x = 2.0f * 3.1415926f * idx / N; // значение аргумента
   data [idx] = sinf ( sqrtf ( x ) ); // найти значение и записать в массив
}
```

- Для каждого элемента массива (всего N) запускается отдельная нить, вычисляющая требуемое значение.
- Каждая нить обладает уникальным id

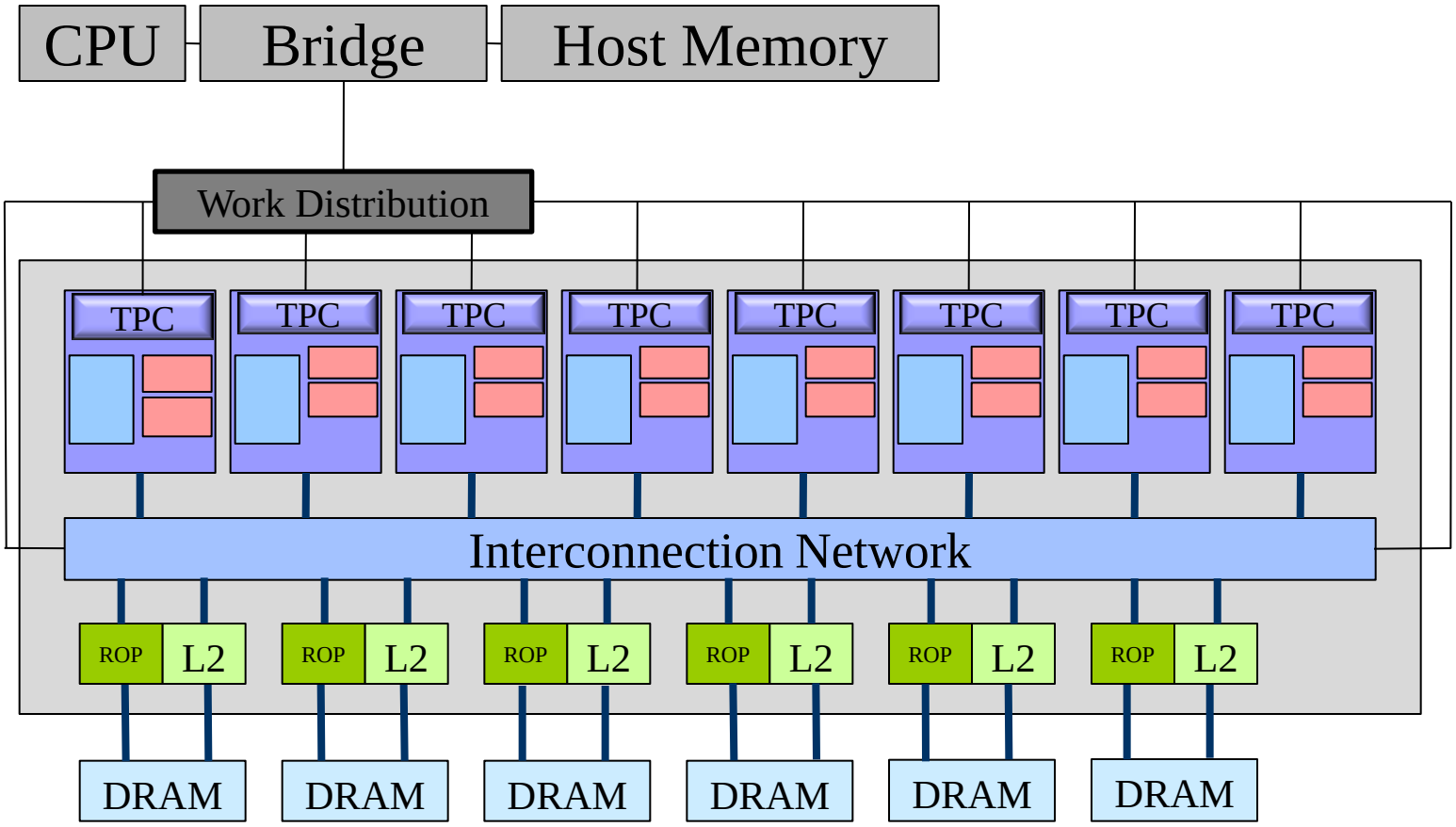
CUDA “Hello World”

```
float   a [N];
float * dev = NULL;

                    // выделить память на GPU под N элементов
cudaMalloc ( (void**)&dev, N * sizeof ( float ) );

                    // запустить N нитей блоками по 512 нитей
                    // выполняемая на нити функция - kernel
                    // массив данных - dev
kernel<<<dim3((N/512),1), dim3(512,1)>>> ( dev );

                    // скопировать результаты из памяти GPU (DRAM) в
                    // память CPU (N элементов)
cudaMemcpy ( a, dev, N * sizeof ( float ), cudaMemcpyDeviceToHost );

                    // освободить память GPU
cudaFree    ( dev   );
```

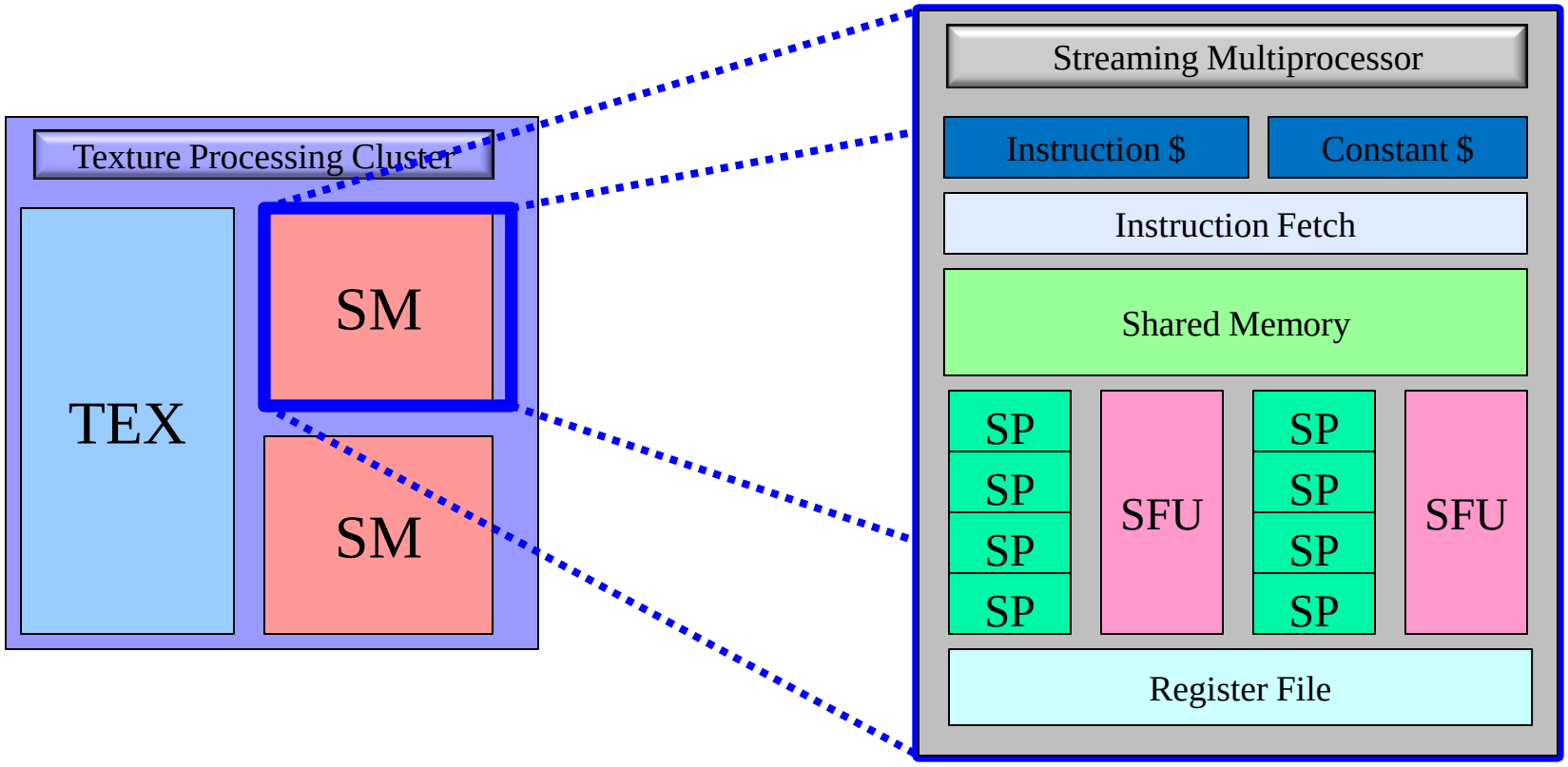
План

- CUDA
- Архитектура
- Hello, World
- Дополнительные слайды
  - Архитектура Tesla 8
  - Архитектура Tesla 20

Архитектура Tesla 8



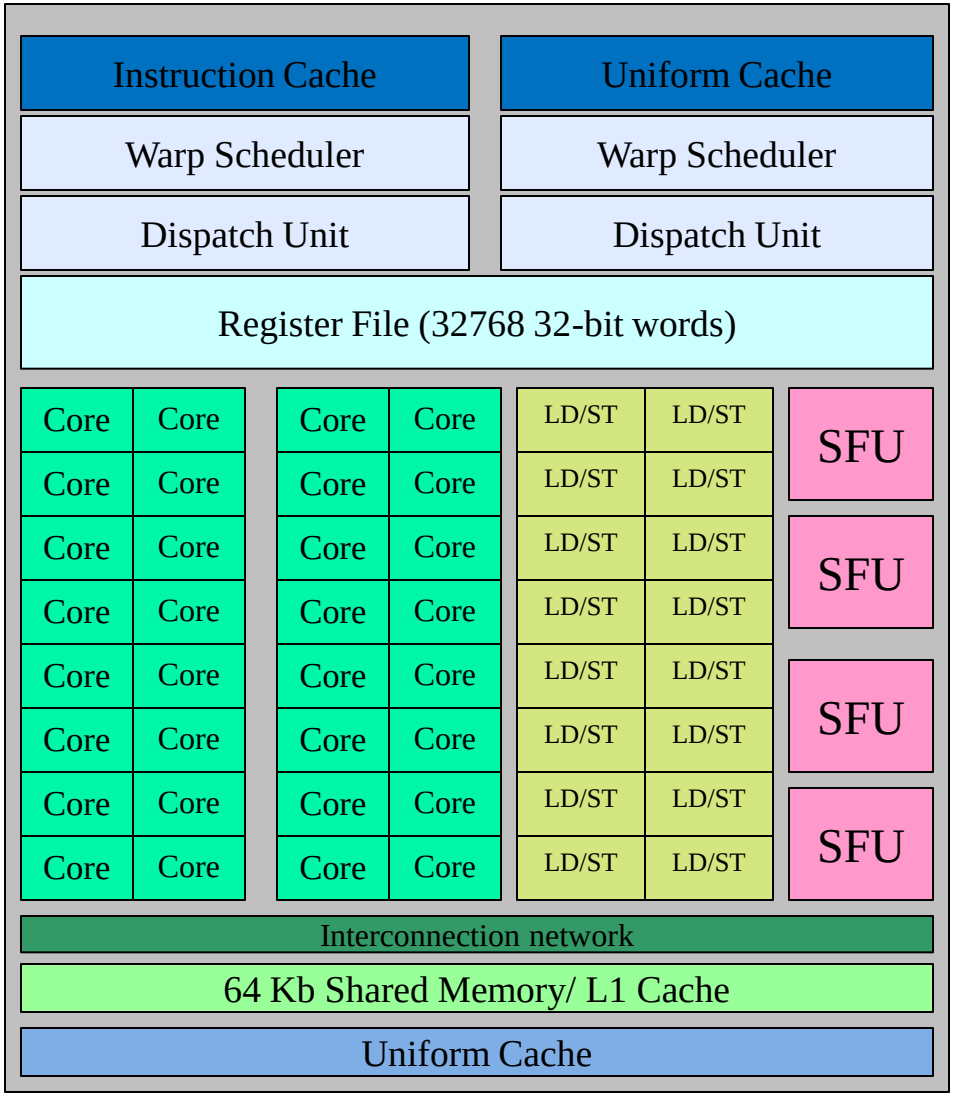
Архитектура Tesla: Мультипроцессор Tesla 8



Архитектура Tesla 20

- Объединенный L2 кэш (768 Kb)
- До 1 Tb памяти (64-битная адресация)
- Общее адресное пространство памяти
- ККО (DRAM, регистры, разделяемая память, кэш)
- Одновременное исполнение ядер, копирования памяти (CPU->GPU, GPU->CPU)
- Быстрая смена контекста (10x)
- Одновременное исполнение ядер (до 16)

Архитектура Tesla 20
Потоковый мультипроцессор



Архитектура Tesla 20 (Fermi)

- 32 ядра на SM
- Одновременное исполнение 2х варпов.
- 48 Kb разделяемой памяти
- 16 Kb кэш
  - или 16 Kb разделяемй + 48 Kb кэш
- Дешевые атомарные операции

Архитектура 3.0 (Kepler)

- 192 ядра на SM
- Одновременное исполнение 4х варпов.
- 48 Kb разделяемой памяти
- 32 SFU
- 16 Kb кэш
  - или 16 Kb разделяемй + 48 Kb кэш
- Регистры 2X

Архитектура 5.0 (Maxwell)

- 128 ядра на SM
- Одновременное исполнение 4х варпов.
- 48 Kb разделяемой памяти
- 32 SFU
- 16 Kb кэш
  - или 16 Kb разделяемй + 48 Kb кэш
- Read-only constant cache

Архитектура 6.0 (Pascal)

- 64/128 ядра на SM
- Одновременное исполнение 2/4х варпов.
- 16/32 SFU
- Shared/cache 64/128 Kb
- Read-only constant cache