NVIDIA CUDA И OPENACC
ЛЕКЦИЯ 1

Перепёлкин Евгений

СОДЕРЖАНИЕ

Лекция 1

- Введение
- Гибридная модель вычислений
- Типы вычислительных архитектур
- Архитектура графического процессора GPU

СОДЕРЖАНИЕ

Лекция 1

- Программная модель CUDA
- Гибридная модель программного кода
- Понятие потока, блока, сети блоков
- Функция-ядро как параллельный код на GPU
- Пример программы на CUDA

Введение

ПРОБЛЕМЫ

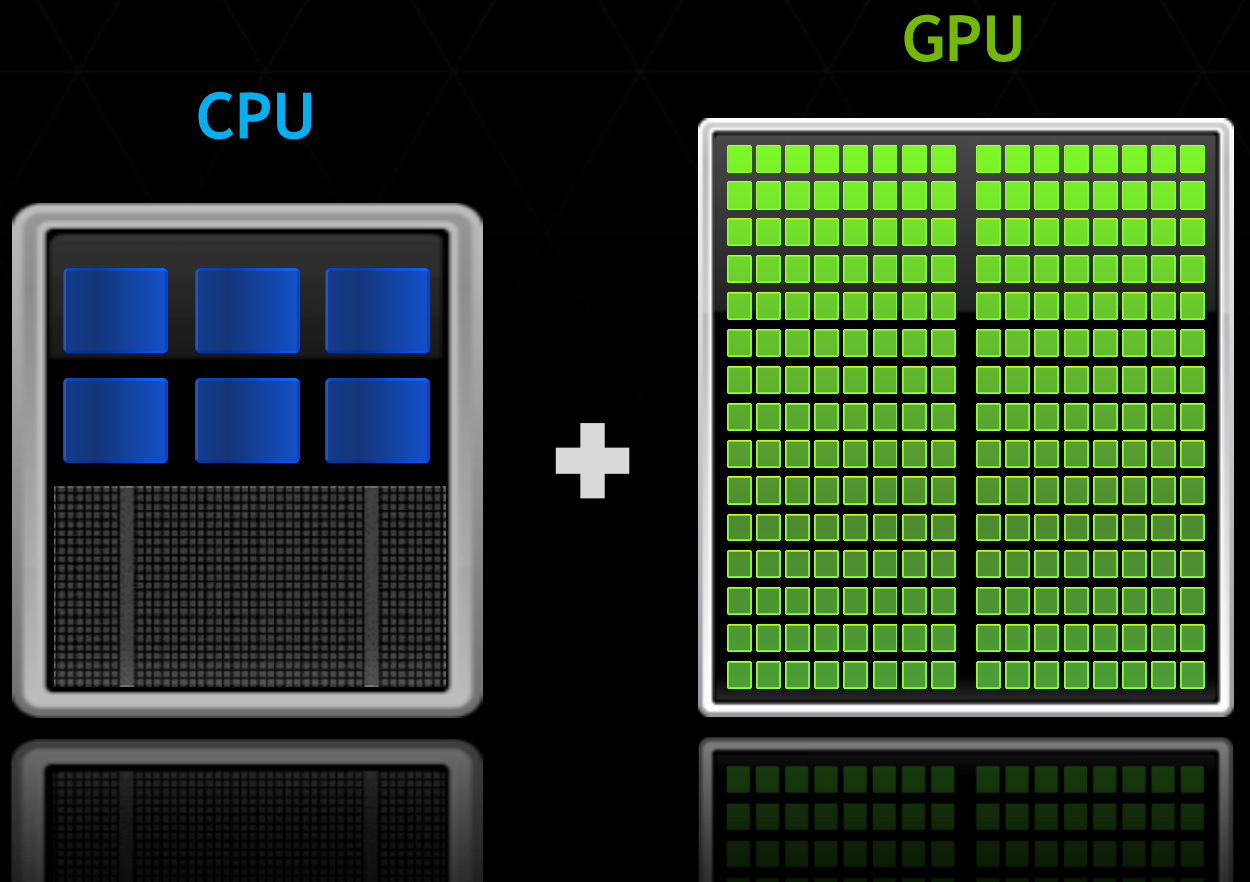
Рост частоты практически отсутствует

- Энерговыделение ~ второй степени частоты
- Ограничения техпроцесса

Гибридная модель вычислений

РАБОТА В ТАНДЕМЕ

Распределение вычислительной нагрузки между архитектурами



ГИБРИДНАЯ МОДЕЛЬ ВЫЧИСЛЕНИЙ

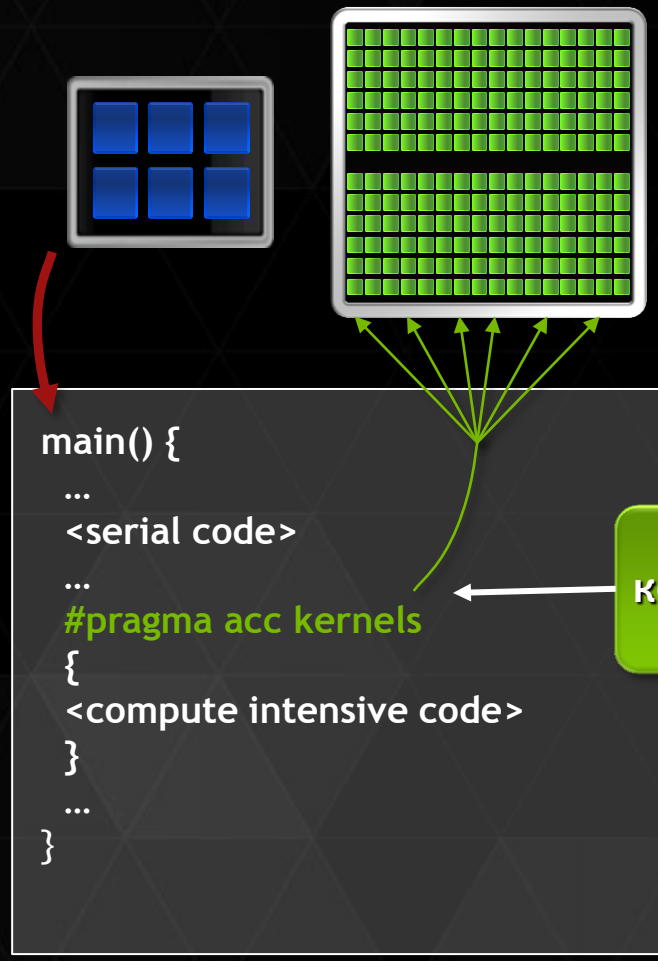
ПЕРВЫЙ ШАГ

Библиотеки

- cuFFT (Быстрое Преобразование Фурье)
- cuBLAS (библиотека линейной алгебры)
- cuRAND (генератор случайных чисел)
- cuSPARSE (работа с разряженными матрицами)
- NPP (библиотека примитивов)
- Plugin - MATLAB, Mathematica

ВТОРОЙ ШАГ

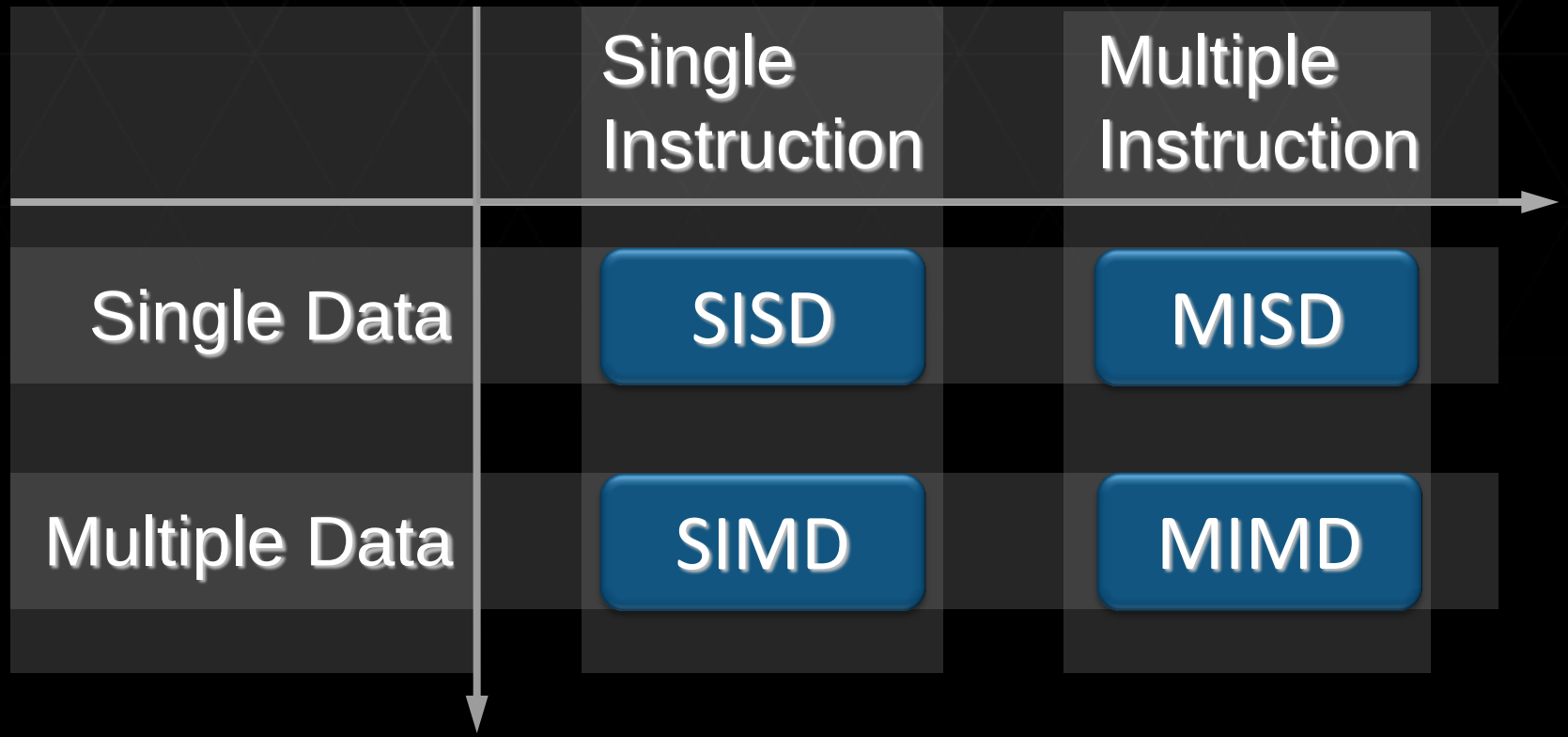
OpenACC



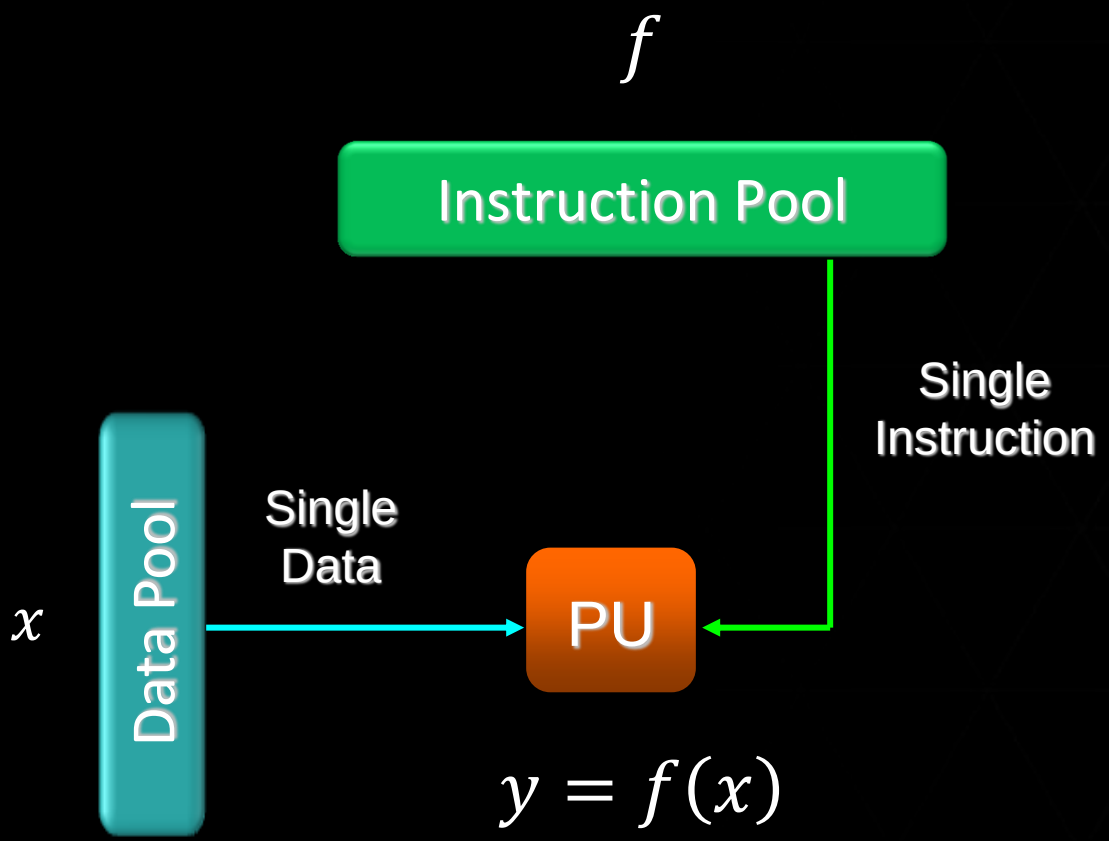
- Открытый стандарт
- Простота
- Использование на GPUs

Типы вычислительных архитектур

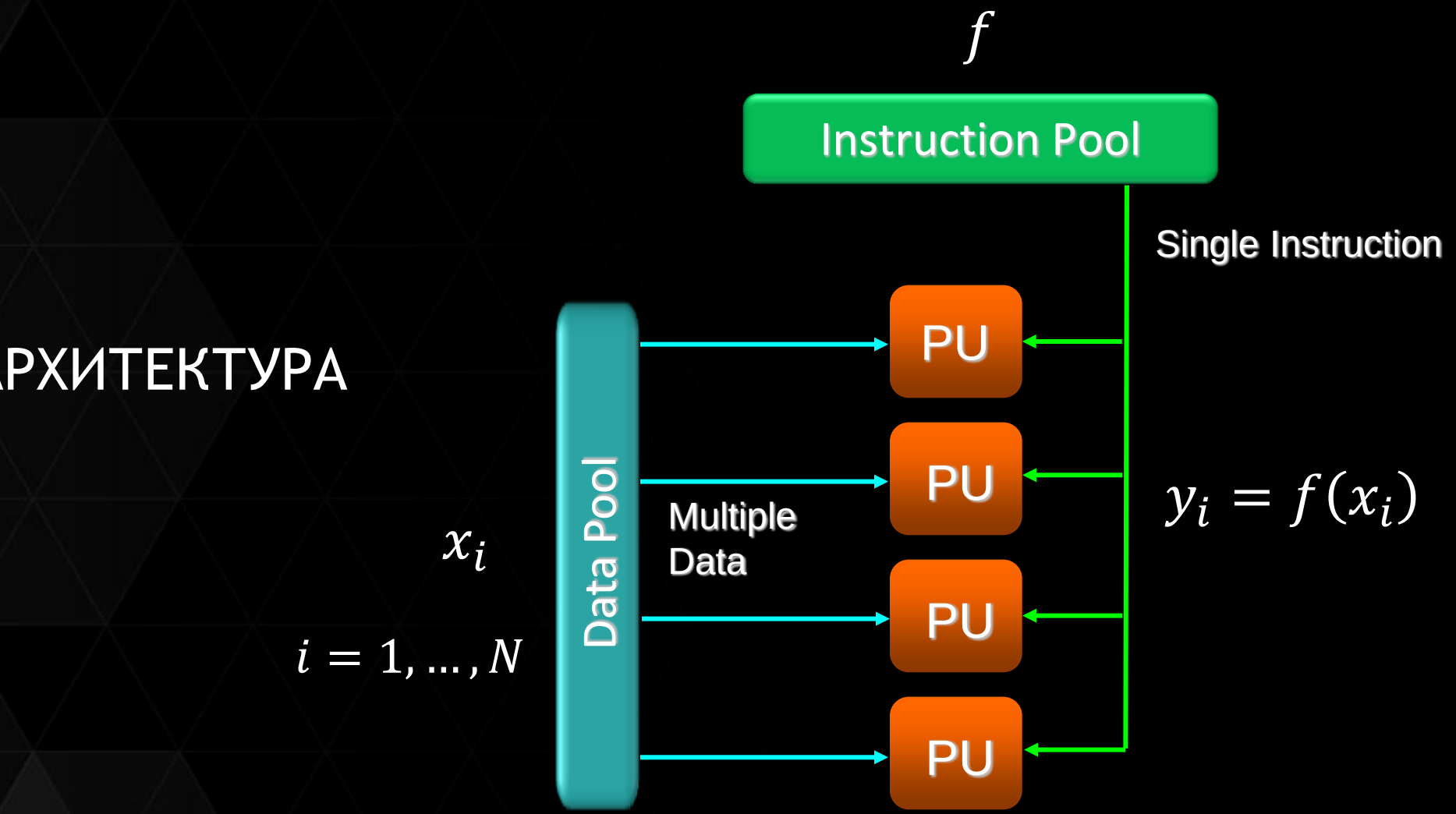
КЛАССИФИКАЦИЯ



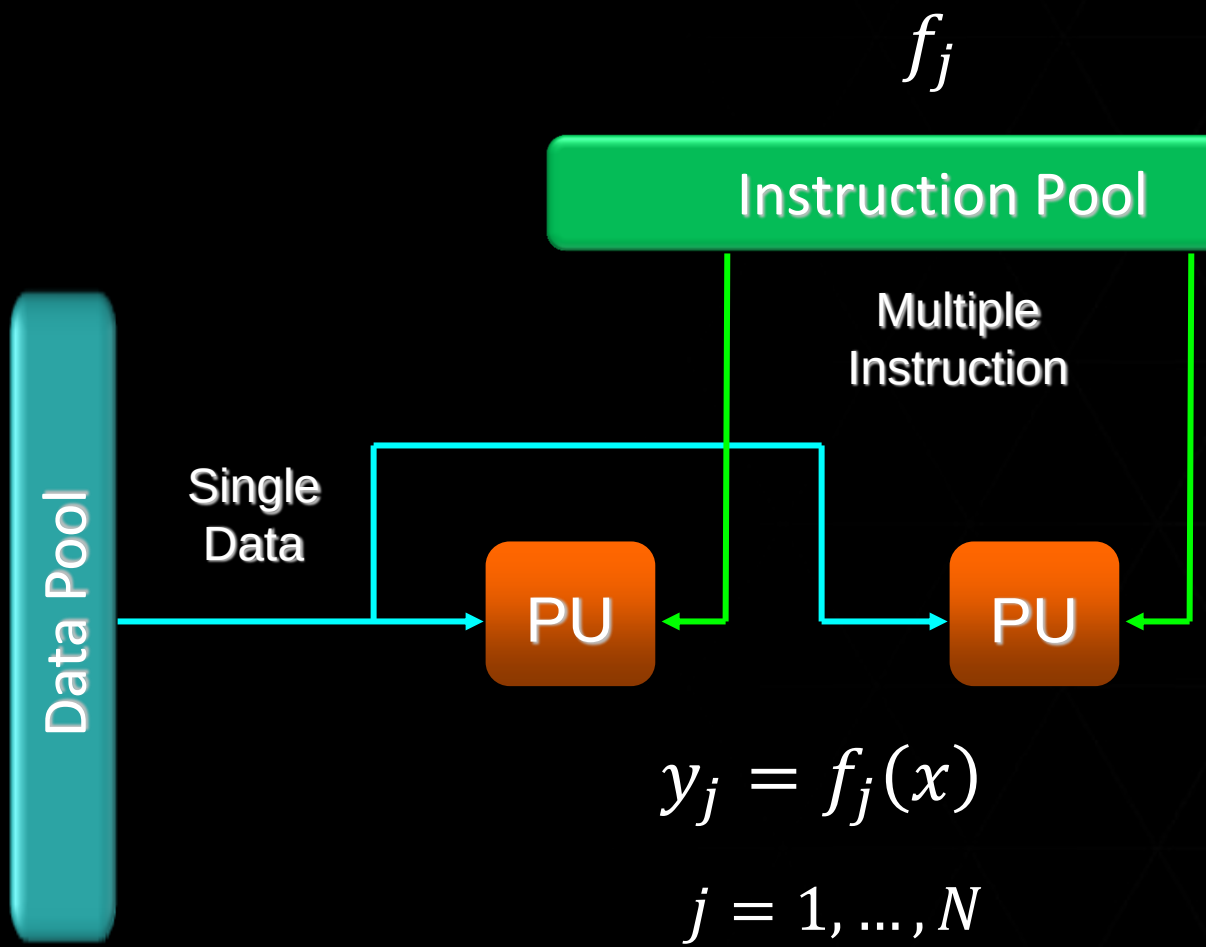
SISD АРХИТЕКТУРА



SIMD АРХИТЕКТУРА



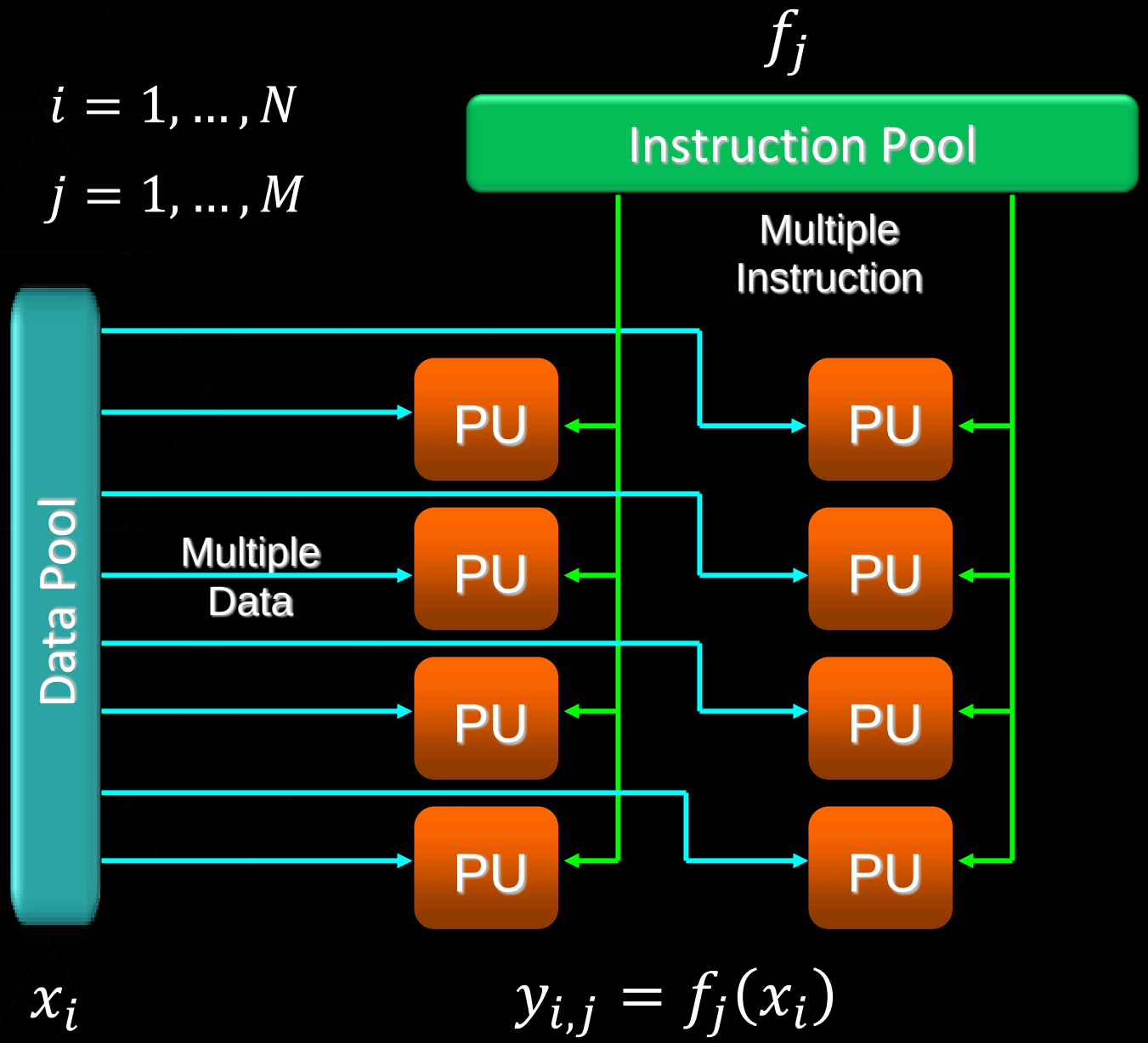
MISD АРХИТЕКТУРА



MIMD АРХИТЕКТУРА

$i = 1, \dots, N$

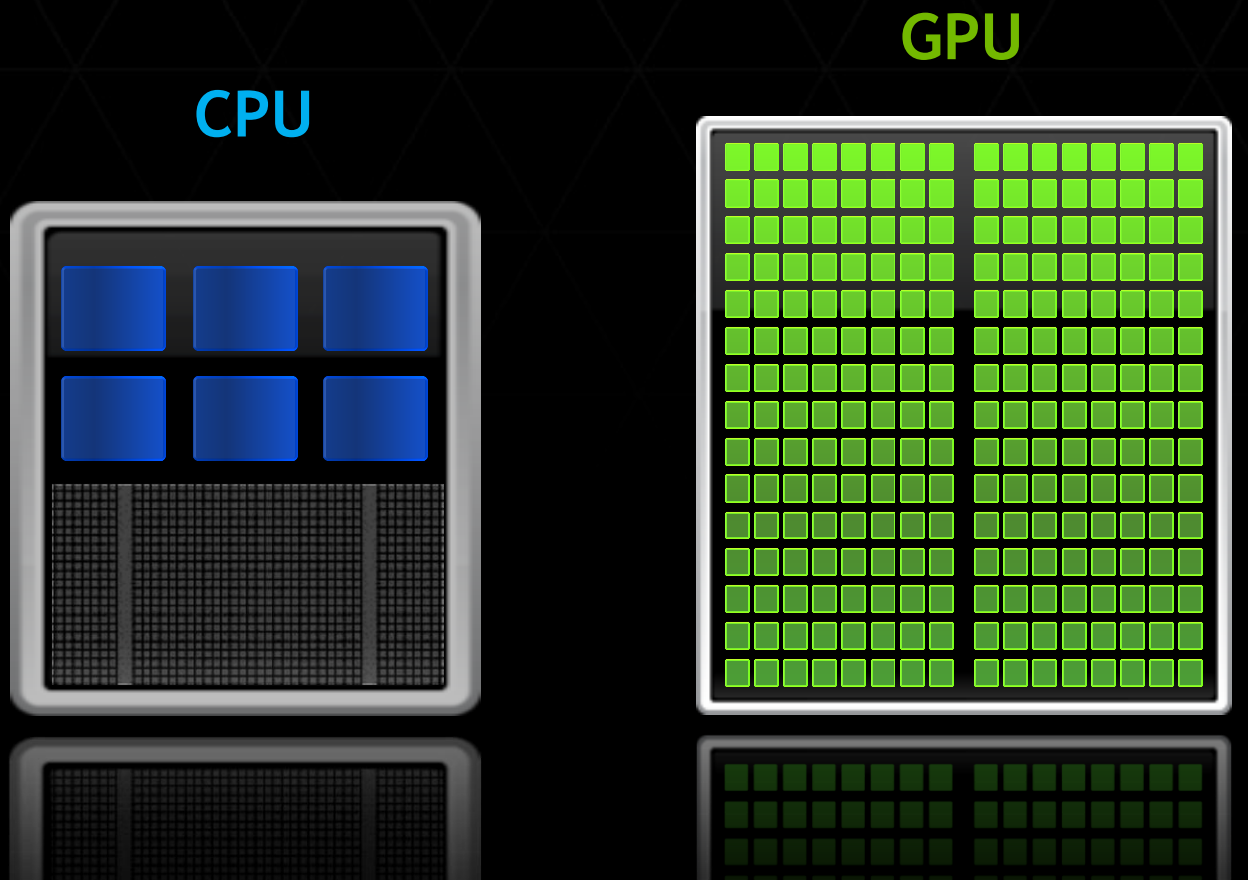
$j = 1, \dots, M$



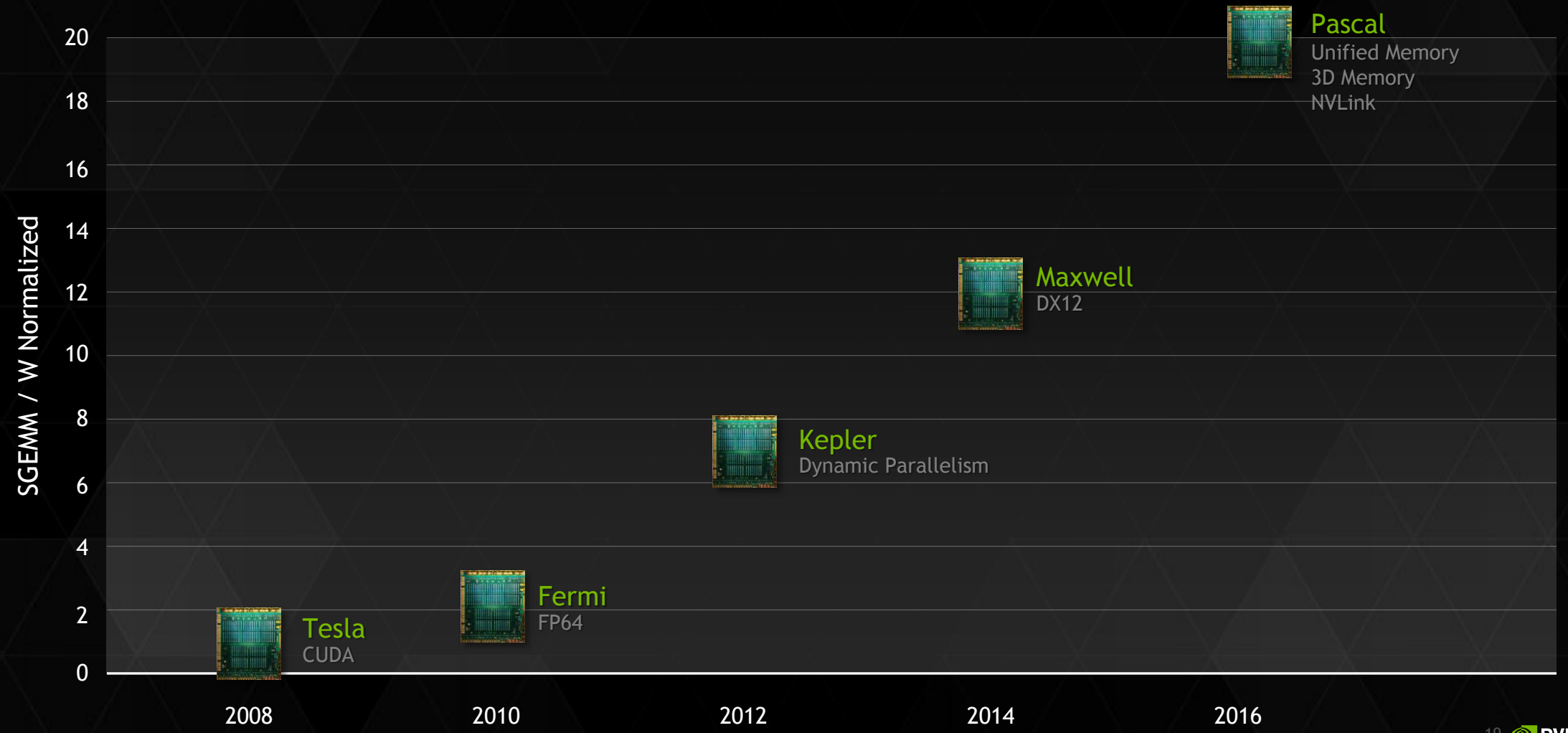
Архитектура графического процессора GPU

ОСОБЕННОСТИ АРХИТЕКТУР

Сравнение



РАЗВИТИЕ GPU АРХИТЕКТУРЫ



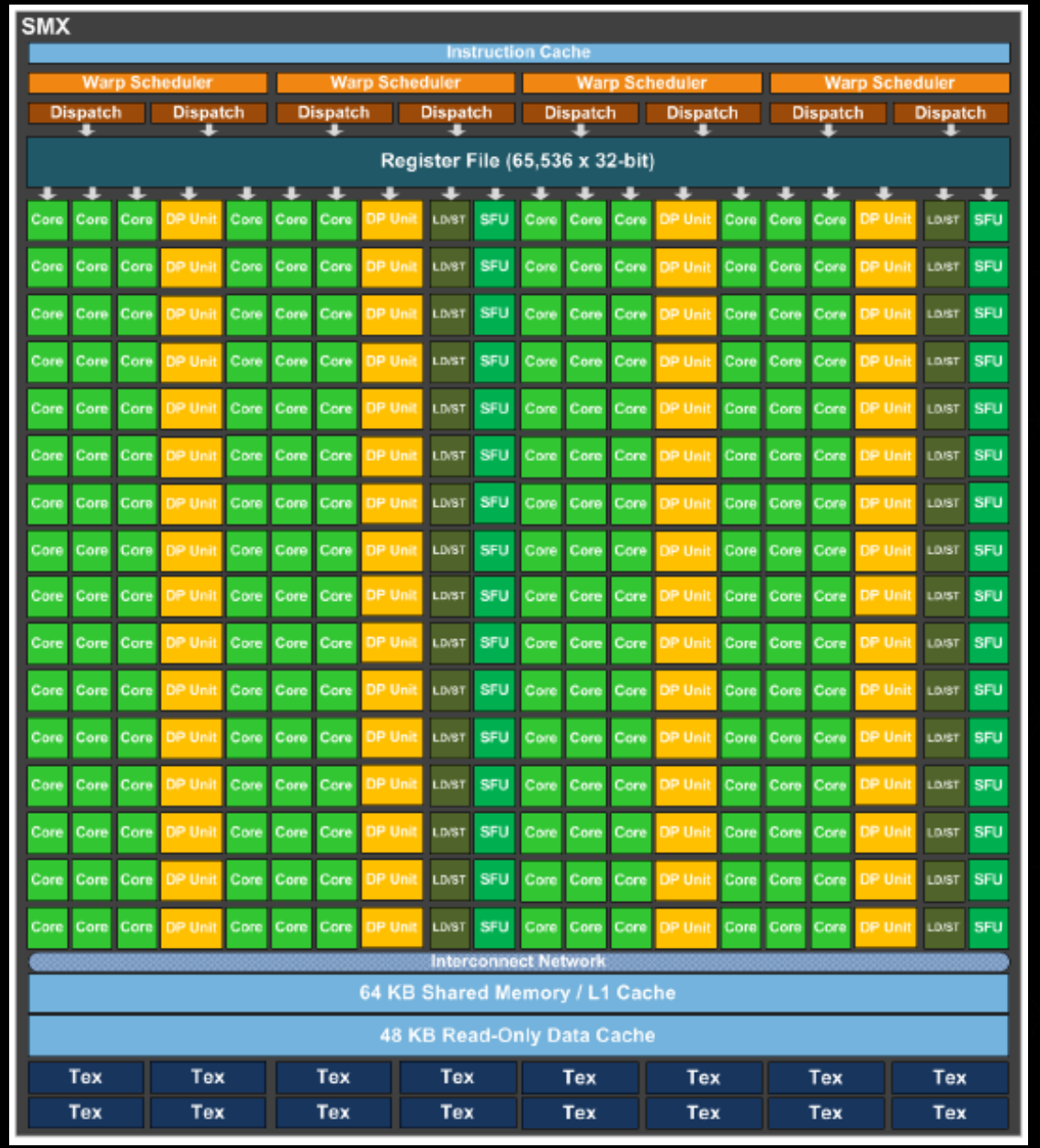
NVIDIA TESLA

Tesla C

Tesla M

Tesla S

АРХИТЕКТУРА KEPLER GK 110



СТРУКТУРА SMX

Core - 192 ядра

DP - 64 блока для вычислений с двойной точностью

SFU - 32 блока для вычислений специальных функций

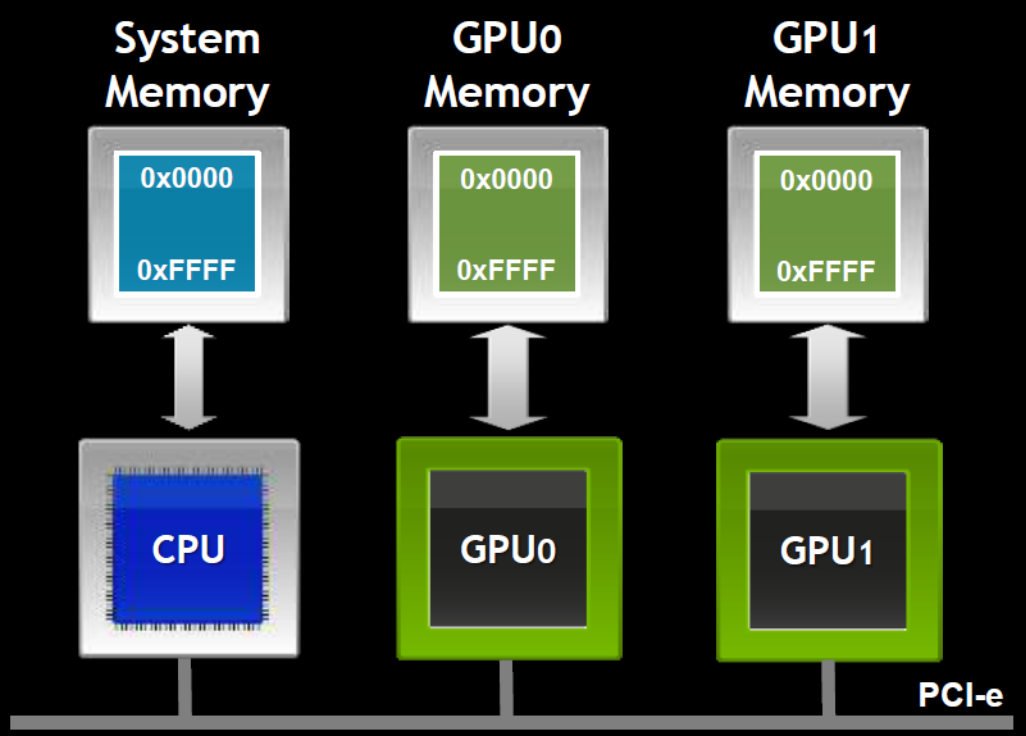
LD/ST - 32 блока загрузки / выгрузки данных

Warp Scheduler - 4 планировщика варпов

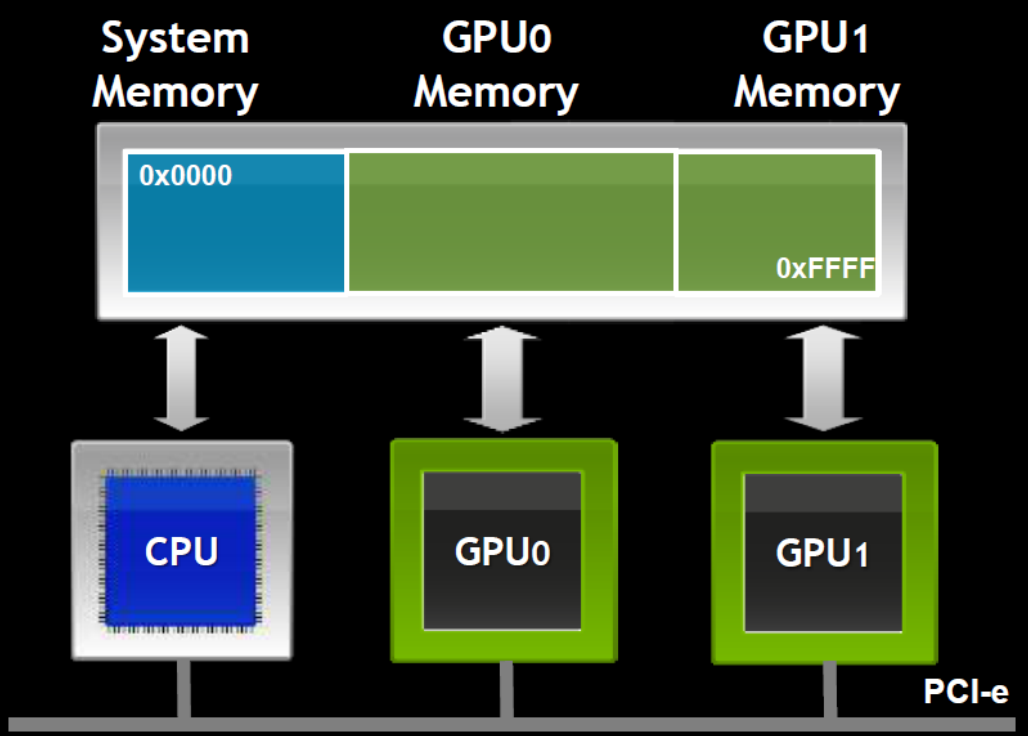
ПЛАНИРОВЩИК ВАРПОВ

UNIFIED VIRTUAL ADDRESSING

Без UVA

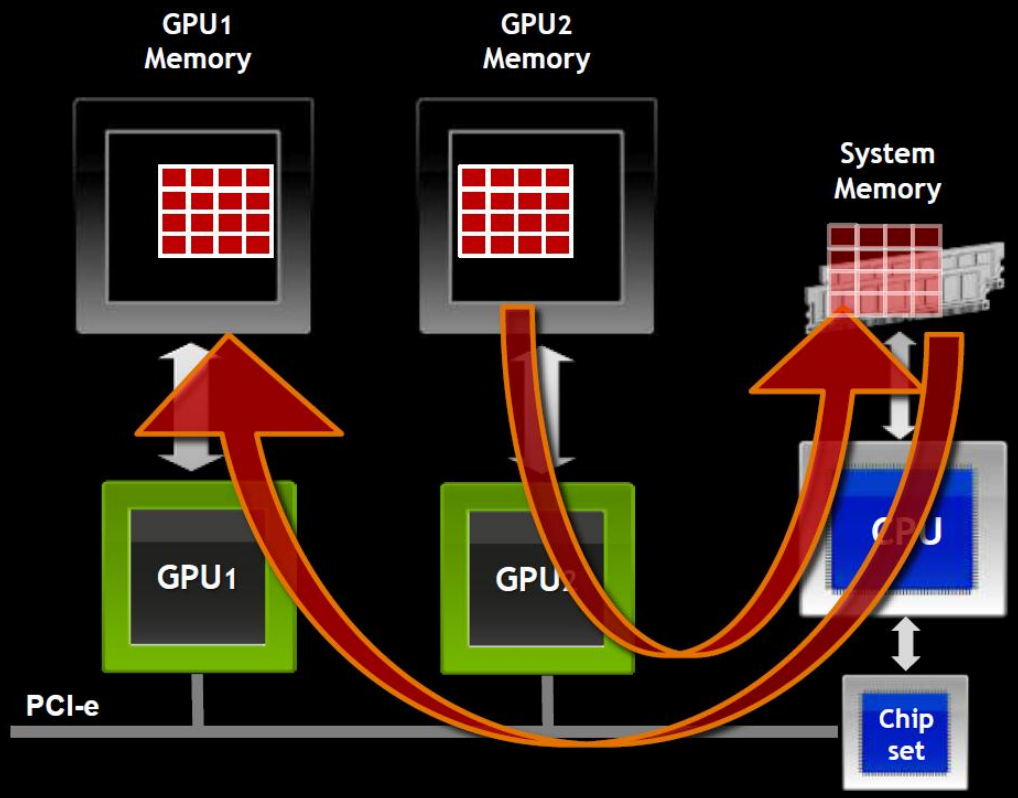


UVA

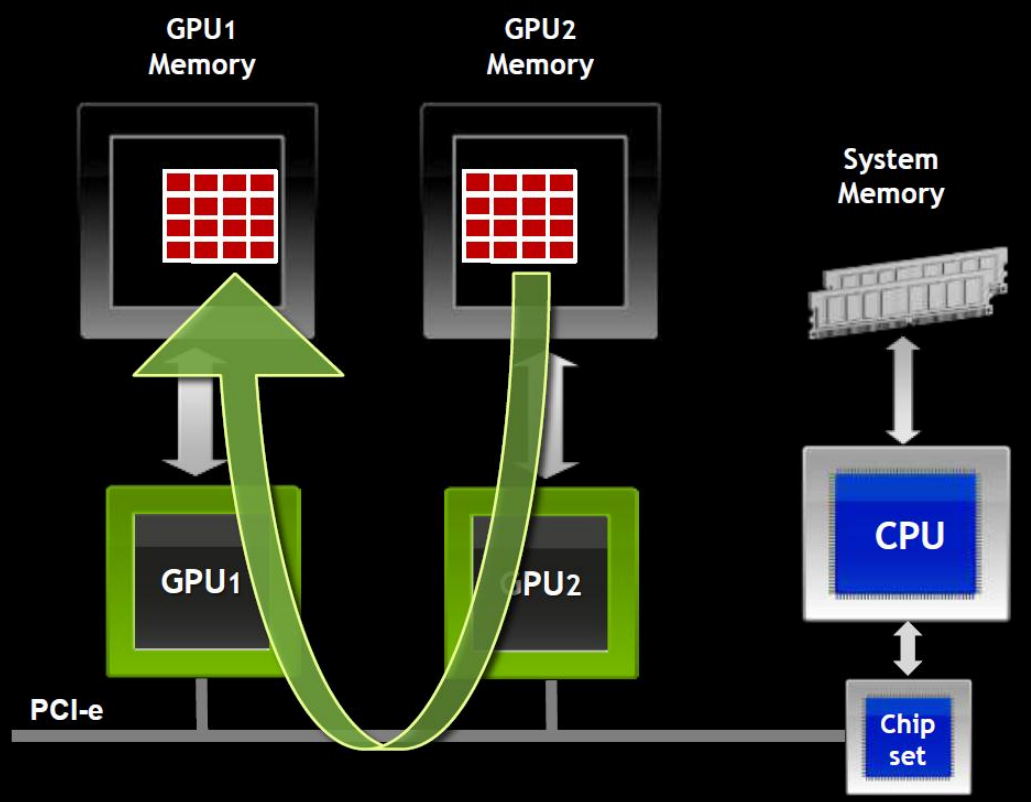


КОПИРОВАНИЕ ДАННЫХ

GPUDirect 1.0



GPUDirect 2.0



Compute Capability	
3.0/3.5	TESLA K10/20/20X/40X
2.0/2.1	TESLA Fermi C2070/2090
1.3	TESLA C1060
1.1	Geforce 8800GT
1.0	TESLA C870

Сравнение

	Fermi GF100	Fermi GF104	Kepler GK104	Kepler GK110
Compute Capability	2.0	2.1	3.0	3.5
Threads/Warp	32	32	32	32
Max Warps/Multiprocessor	48	48	64	64
Max Threads/Multiprocessor	1536	1536	2048	2048
Max Thread Blocks/Multiprocessor	8	8	16	16
32-bit Register/Multiprocessor	32768	32768	65536	65536
Max Register/Thread	63	63	63	255
Max Threads/Thread Block	1024	1024	1024	1024
Shared Memory Size Configurations (KB)	16/48	16/48	16/32/48	16/32/48

Программная модель CUDA

ПРИЛОЖЕНИЯ НА GPU

Compute Unified Device Architecture

GPU Computing Applications

Libraries and Middleware

cuFFT cuBLAS cuRAND cuSPARSE	CULA MAGMA	Thrust NPP	VSIPL SVM OpenCurrent	PhysX OptiX	iray	MATLAB Mathematica

Programming Languages

C	C++	Fortran	Java Python Wrappers	Directives (e.g. OpenACC)

NVIDIA GPU
CUDA Parallel Computing Architecture

С ЧЕГО НАЧАТЬ?

http://developer.nvidia.com

- Драйвер для видеокарты «NVIDIA»
- CUDA SDK
- CUDA ToolKit
- Parallel Nsight + MS VS
- Документация по CUDA

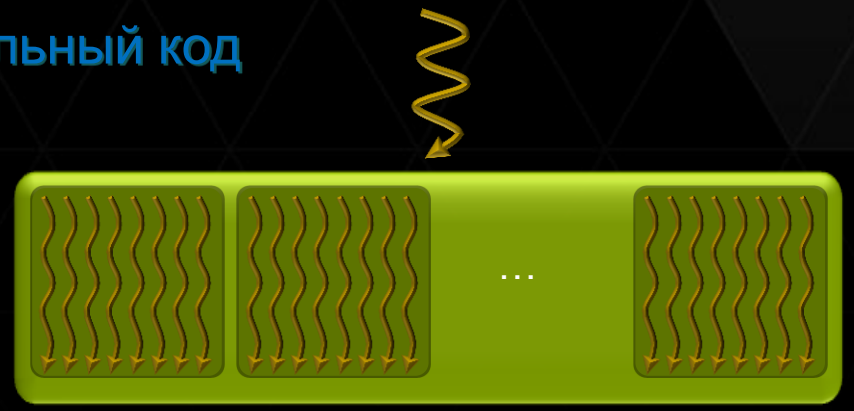
Гибридная модель программного кода

СТРУКТУРА КОДА

Последовательный код

Параллельное ядро A

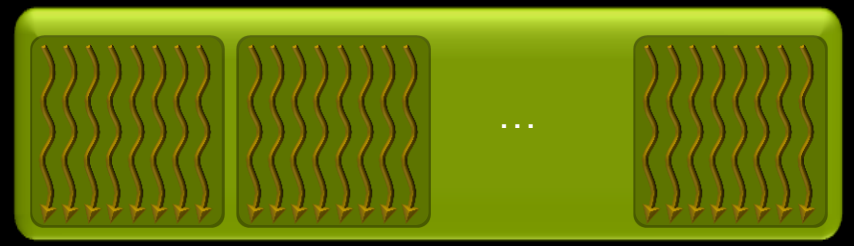
```
KernelA <<< nBlk, nTid >>> (args);
```



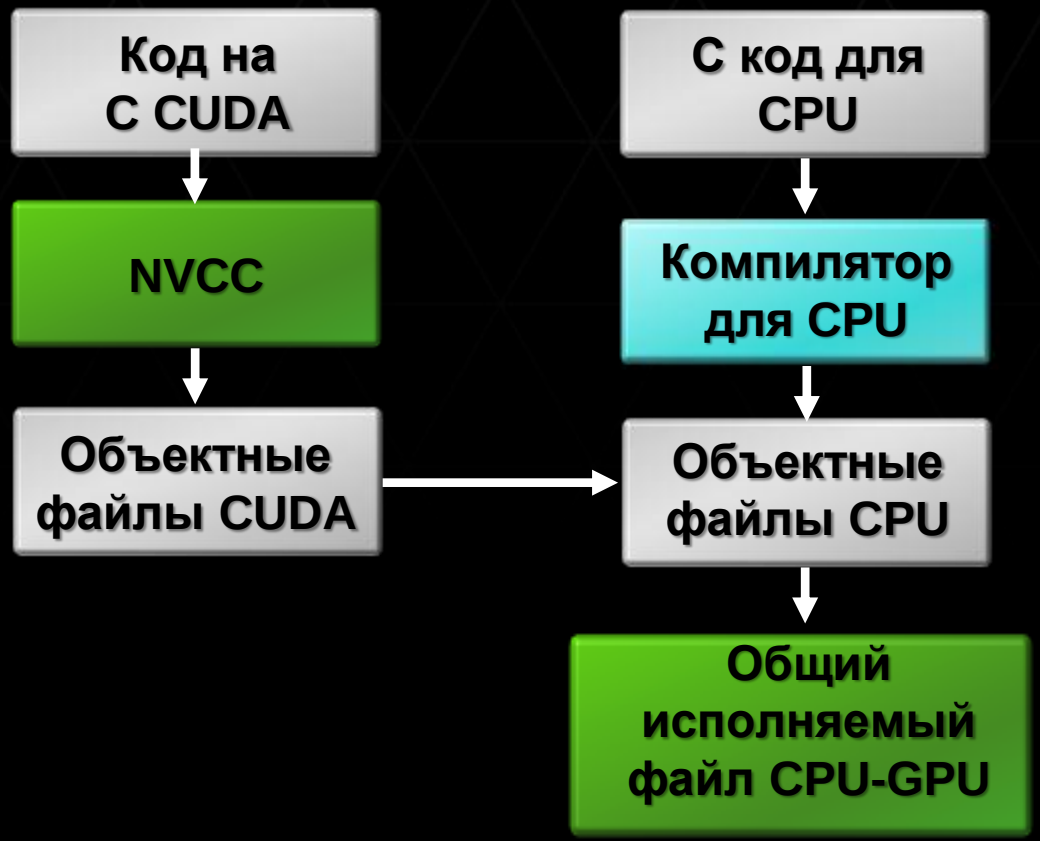
Последовательный код

Параллельное ядро B

```
KernelB <<< nBlk, nTid >>> (args);
```



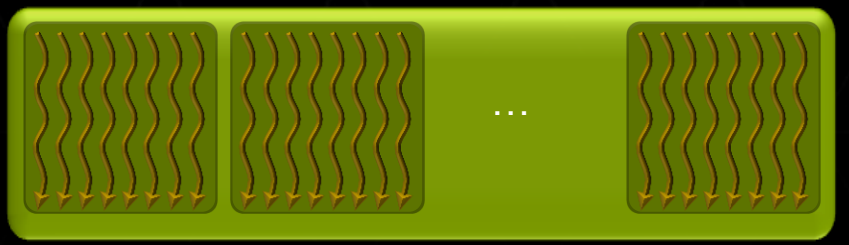
СБОРКА ПРИЛОЖЕНИЯ



ОСНОВНЫЕ ПОНЯТИЯ

- «host» - CPU
- «device» - GPU

```
Kernel <<< nBlk, nTid >>> (args);
```



N_{max} – максимальное число потоков на GPU

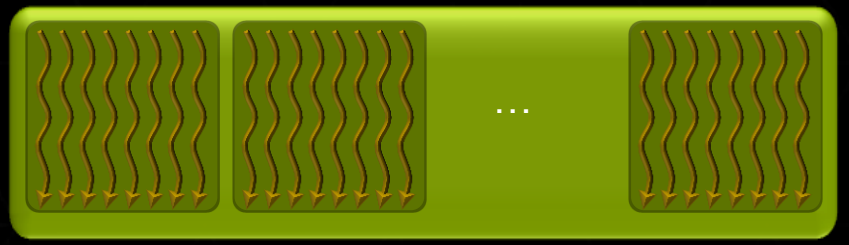
$$y_i = f(x_i), i = 1, \dots, N$$

$N \leq N_{max}$ или $N > N_{max}$

Понятие потока, блока, сети блоков

ПОТОКИ И БЛОКИ ПОТОКОВ

```
Kernel <<< nBlk, nTid >>> (args);
```



```
nBlk = N / nTid
```

или

```
nBlk = ( int ) ( N / nTid ) + 1
```

Warp состоит из 32 потоков

ПОТОКИ И БЛОКИ ПОТОКОВ

```
dim3 grid (10,1,1);
dim3 block (16,16,1);                    (*)
My_kernel <<< grid, block >>> ( param );
```

или

```
dim3 grid (10);
dim3 block (16,16);
```

ПОТОКИ И БЛОКИ ПОТОКОВ

threadIdx - номер нити в блоке
blockIdx - номер блока, в котором находится нить
blockDim - размер блока

Глобальный номер нити внутри сети:

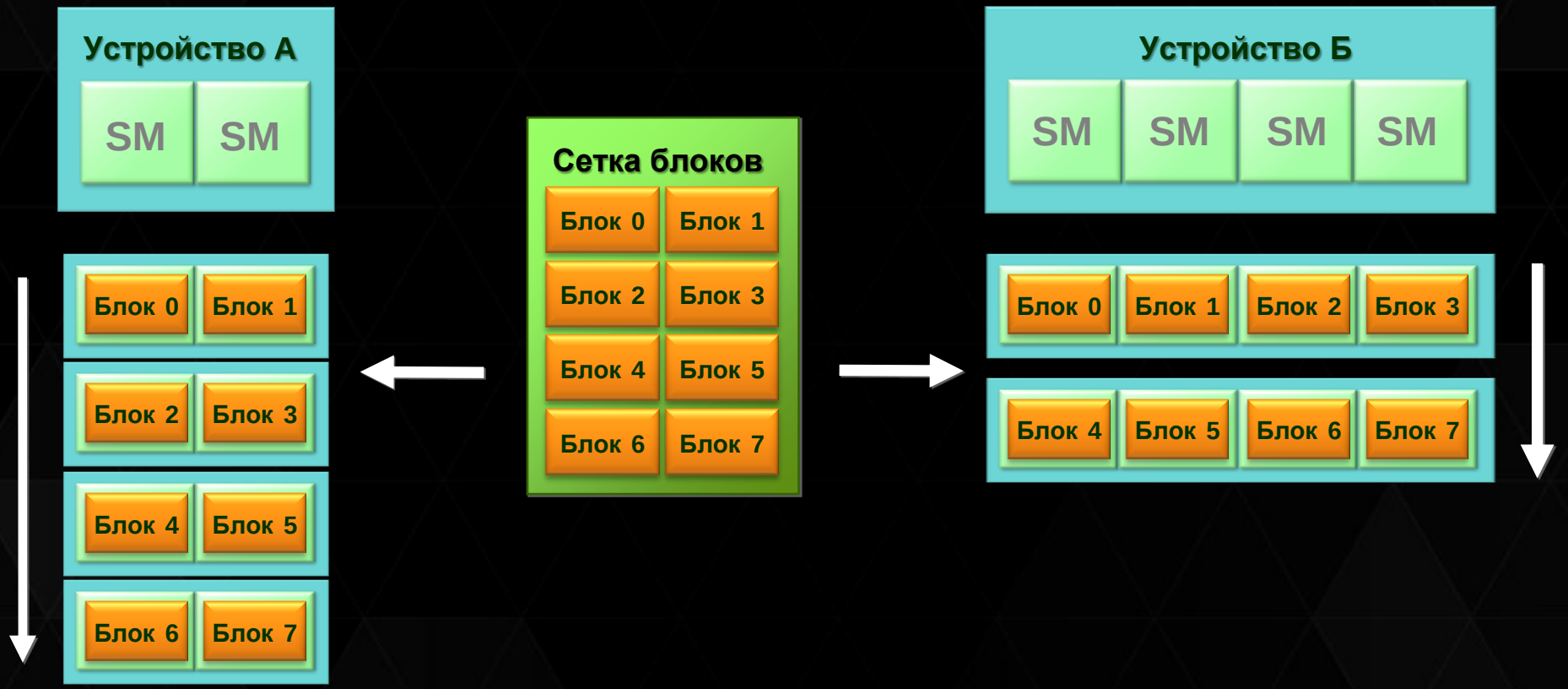
```
threadID = threadIdx.x + blockIdx.x * blockDim.x
```

В общем (3D) случае:

threadIdx – { `threadIdx.x, threadIdx.y, threadIdx.z` }
blockIdx – { `blockIdx.x, blockIdx.y, blockdx.z` }
blockDim – { `blockDim.x, blockDim.y, blockDim.z` }

ПОТОКИ И БЛОКИ ПОТОКОВ

Запуск блоков на различных GPU



Функция-ядро как параллельный код на GPU

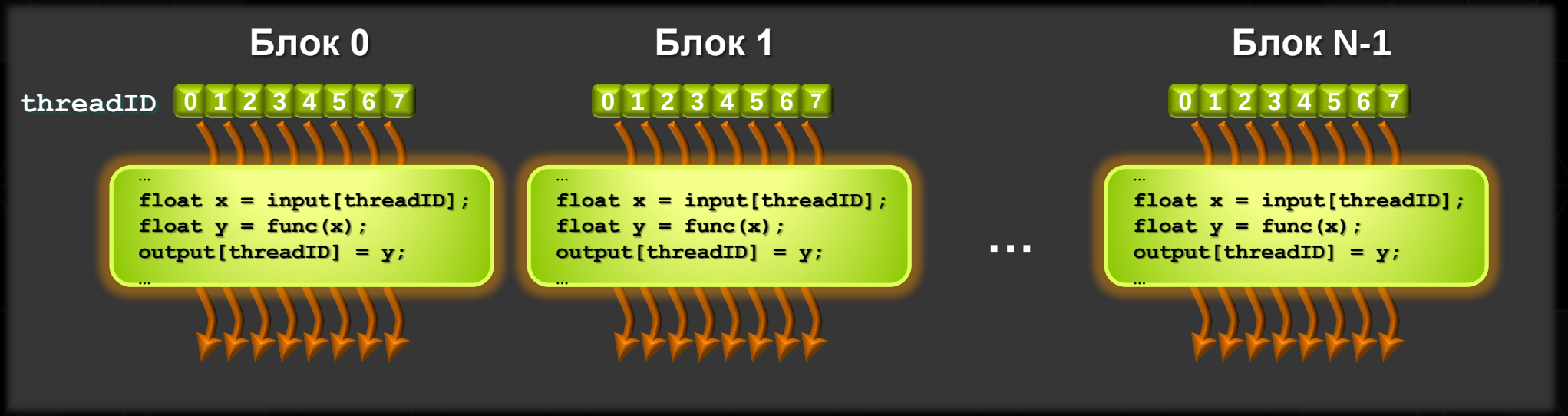
ФУНКЦИЯ-ЯДРО

```
My_Kernel <<< nBlock, nThread,
              nShMem, nStream >>> ( param )
```

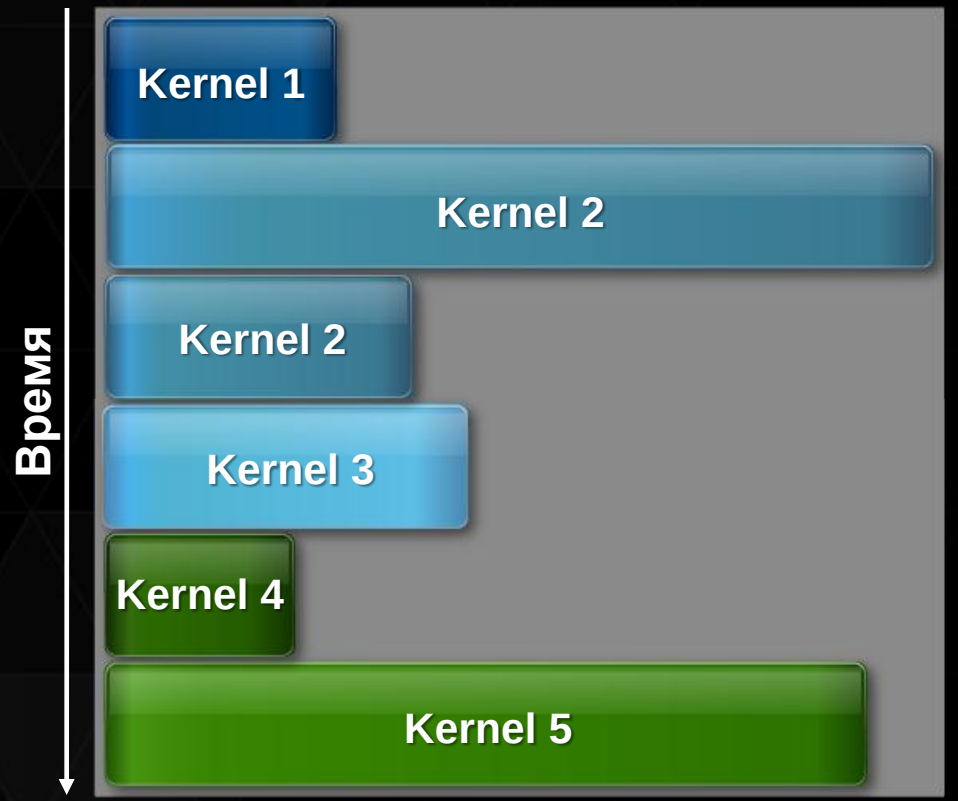
My_Kernel - название функции-ядра
nBlock - число блоков сети (grid)
nThread - число нитей в блоке
nShMem - количество дополнительной разделяемой памяти, выделяемой на блок
nStream - номер потока из которого запускается функция ядро

cudaDeviceSynchronize() - синхронизация потоков

ПАРАЛЛЕЛЬНОЕ ВЫПОЛНЕНИЕ КОДА



ВЫПОЛНЕНИЕ НЕСКОЛЬКИХ ФУНКЦИЙ-ЯДЕР



Последовательное исполнение

Параллельное исполнение

Спецификатор функций

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

*только на устройствах с CC 3.5 и выше

Спецификатор переменных

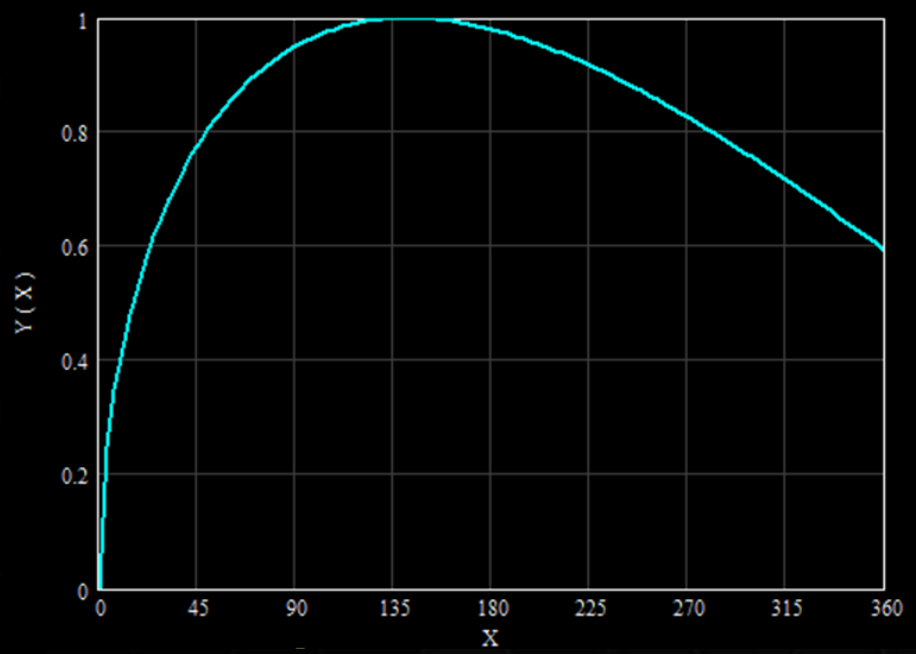
Спецификатор	Находится	Доступна	Вид доступа
__device__	device	device/host	R/W
__constant__	device	device/host	R/W
__shared__	device	block	RW __syncthreads()

Пример программы на CUDA

ПРИМЕР

Параллельного вычисления функции y (x)

$$y_i = sin(\sqrt{x_i}),\ x_i = \frac{2\pi}{N} i,$$
$$i = 1, \dots, N$$



- N = 1024 * 1024
- 512 нитей в блоке, тогда 2048 блоков
- Массивы dA (device), hA (host)
- cudaMalloc, cudaMemcpy

ПРИМЕР

Код программы на CUDA (1 часть)

```
#include <stdio.h>

#define N (1024*1024)

__global__ void kernel ( float * dA )

{ int idx = blockIdx.x * blockDim.x + threadIdx.x;

  float x = 2.0f * 3.1415926f * (float) idx / (float) N;

  dA [idx] = sinf (sqrtf ( x ) );

}
```

ПРИМЕР

Код программы на CUDA (2 часть)

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

ОШИБКИ

Выделения памяти на GPU

```
cudaError_t errMem;
errMem = cudaMalloc ((void**)&dA, N * sizeof ( float ));
if (errMem != cudaSuccess)
{fprintf (stderr, “Cannot allocate GPU memory: %s\n”,
cudaGetErrorString (errMem) );
 return 1;
}
```

ОШИБКИ

Копирования данных «host» — «device»

```
cudaError_t errMem;
errMem = cudaMemcpy ( hA, dA, N * sizeof ( float ),
                                        cudaMemcpyDeviceToHost );
if (errMem != cudaSuccess)
{fprintf (stderr, "Cannot copy data device/host : %s\n",
                                        cudaGetErrorString(errMem) );
 return 1;
}
```

ОШИБКИ

Запуска функции-ядра

```
cudaError_t err;
cudaDeviceSynchronize ( );
err = cudaGetLastError ( );
if (err != cudaSuccess)
{fprintf (stderr, “Cannot launch CUDA kernel : %s\n”,
                                    cudaGetErrorString(err) );
 return 1;
}
```

ОШИБКИ

Синхронизации

```
cudaError_t errSync;
errSync = cudaDeviceSynchronize ( );
if (errSync != cudaSuccess)
{ fprintf (stderr, "Cannot synchronize CUDA kernel : %s\n",
                                    cudaGetErrorString(errSync) );
  return 1;
}
```