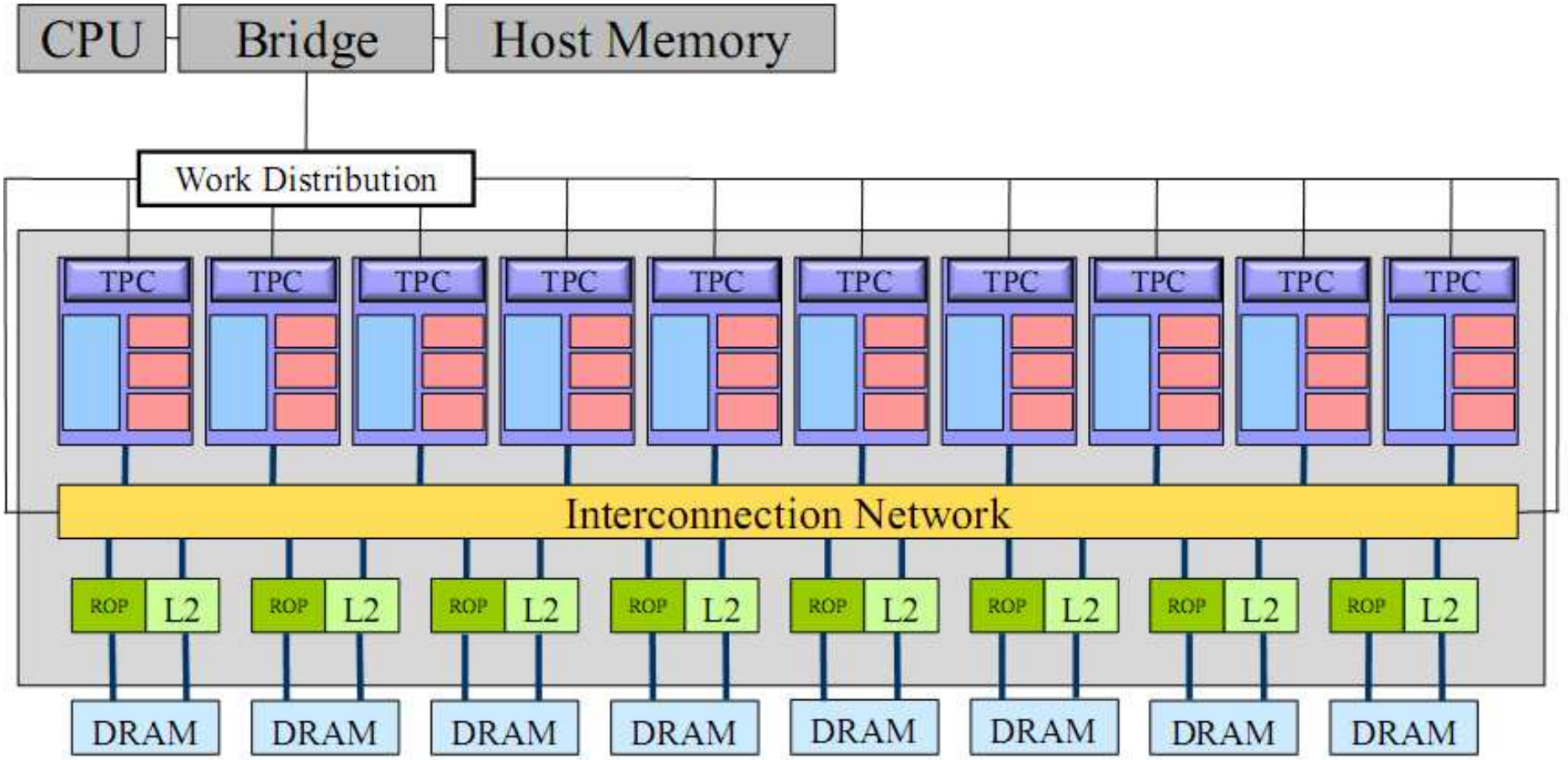
Центр микро- и наномасштабной динамики дисперсных систем

Модель программирования CUDA

Марьин Д. Ф.

Уфа, 2011г.

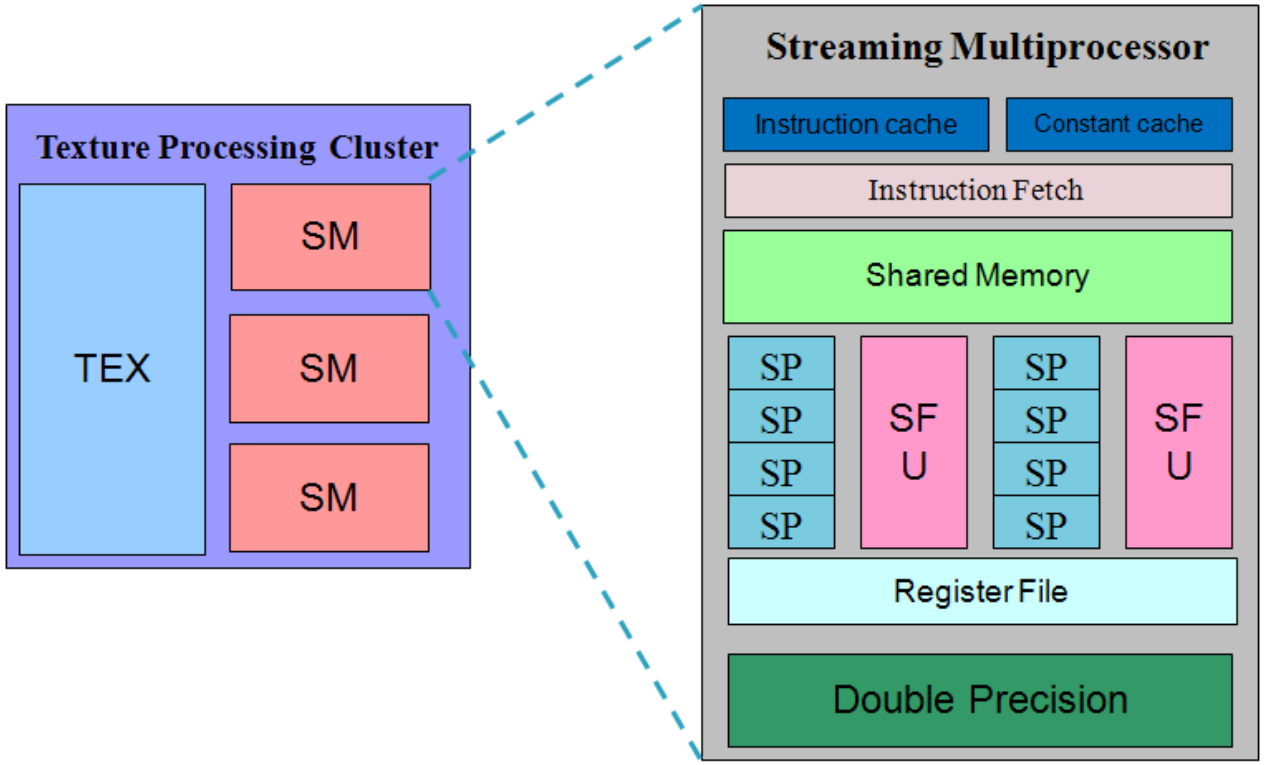
Общая архитектура GPU Tesla 10 series



SPA (Streaming Processor Array) — набор TPC.

Архитектура TPC (Texture Processing Cluster) Tesla 10

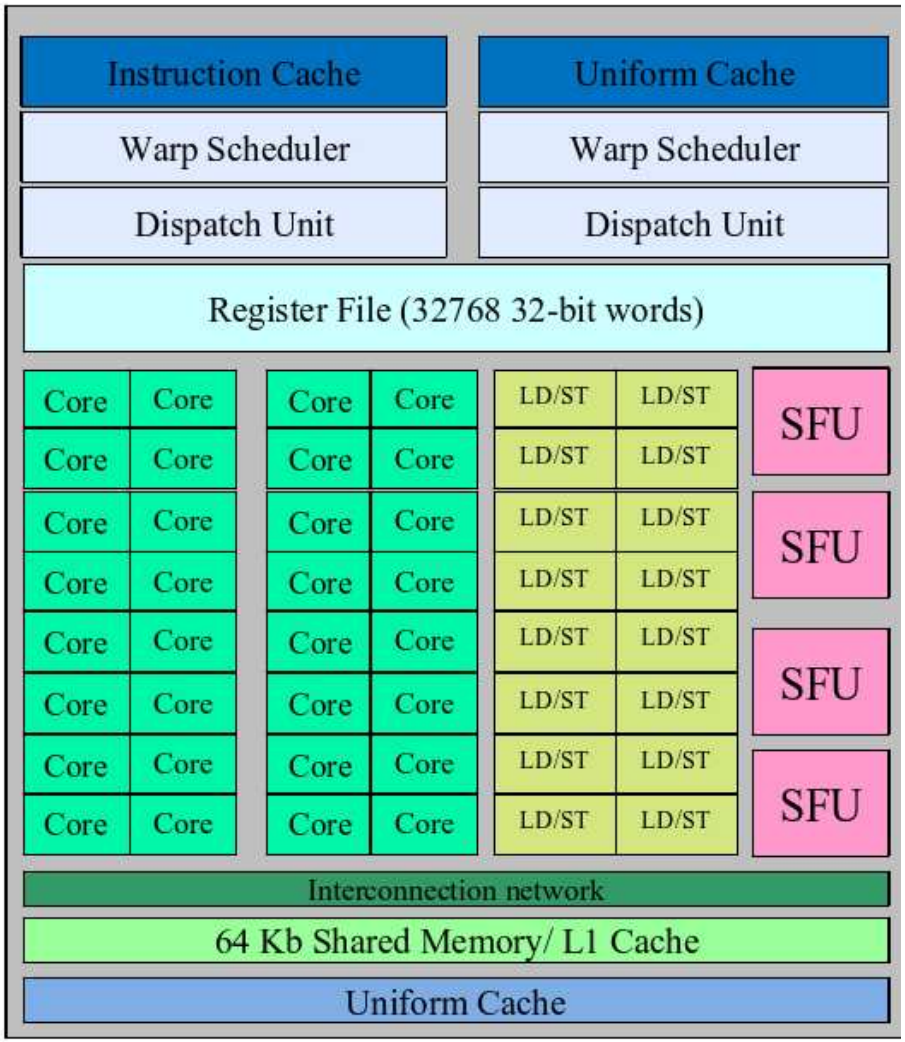
TEX — Texture block
SM — Streaming Multiprocessor
SP — Scalar Processor (CUDA–ядро)
SFU — Super Function Unit



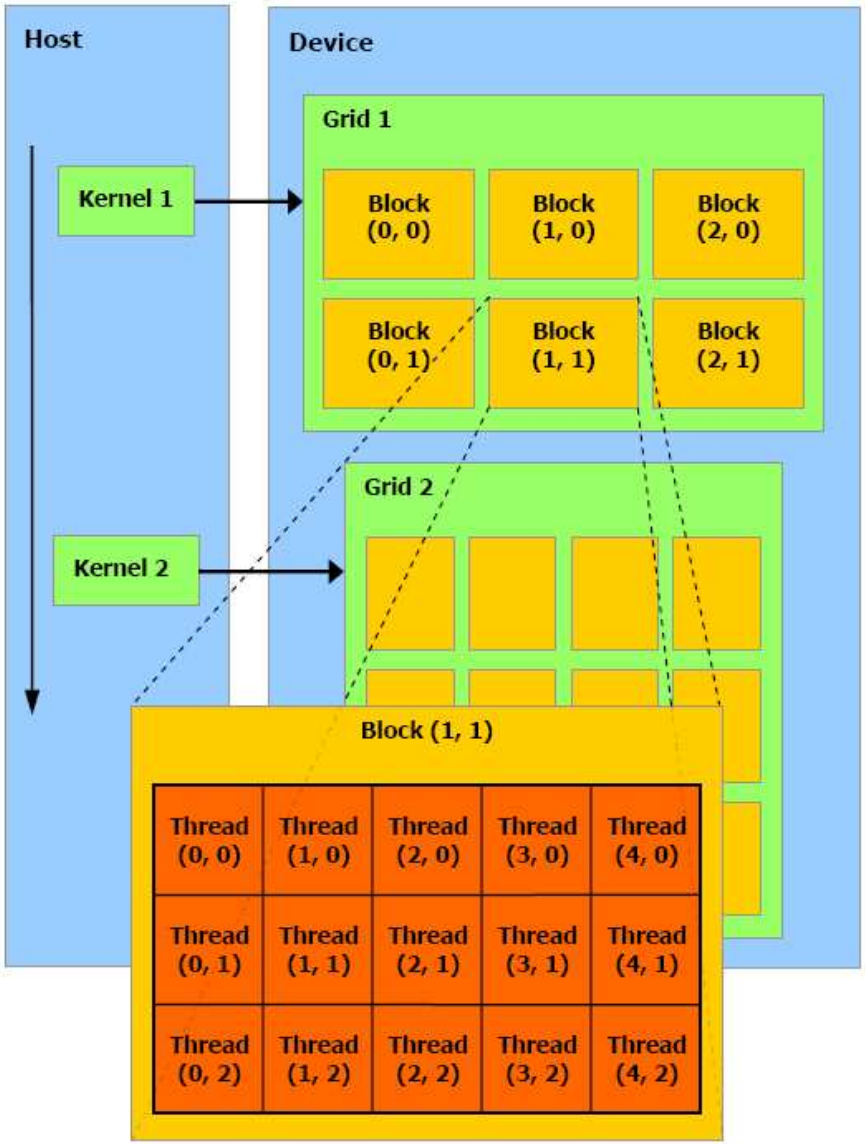
Архитектура Tesla 20

- Объединенный L2 кэш (768 Kb)
- До 1 Tb памяти (64-битная адресация)
- Общее адресное пространство памяти
- Одновременное исполнение ядер, копирования памяти (CPU→GPU, GPU→CPU)
- Одновременное исполнение ядер (до 16)

Архитектура SM Tesla 20



- 32 ядра на SM
- Одновременное исполнение 2x варпов
- 48 Kb разделяемой памяти + 16 Kb кэш;
  или 16 Kb разделяемой + 48 Kb кэш



Каждый поток и блок потоков имеют идентификаторы.
blockIdx — индекс *block*'а внутри *grid*'а (1D, 2D).
threadIdx — индекс *thread*'а внутри *block*'а (1D, 2D, 3D).

Модель программирования CUDA. CPU

Основной процесс приложения CUDA работает на CPU (host):

1. инициализирует GPU
2. распределяет память на видеокарте и системе
3. копирует данные в память видеокарты
4. запускает несколько копий процессов kernel на видеокарте
5. копирует полученные результаты из видеопамяти
6. освобождает память
7. завершает работу

Модель программирования CUDA. GPU

GPU выполняет следующую работу:

1. задача разбивается на подзадачи
2. входные данные делятся на блоки, которые вмещаются в разделяемую память
3. каждый блок обрабатывается блоком потоков
4. над данными в разделяемой памяти проводятся соответствующие вычисления
5. результаты копируются из разделяемой памяти обратно в глобальную

Расширения языка

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

Спецификаторы функций

Определяют, где может выполняться функция и откуда она может быть запущена:

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

- **__host__** опционален;
- **__host__** и **__device__** могут быть использованы вместе.

Пример:

```
__global__ void Func(float* arg1, int arg2, ...);
```

Ограничения на спецификаторы функций

Ограничения на функции выполняемые на GPU (**__device__** и **__global__**):

- не поддерживается рекурсия;
- не поддерживается переменное число входных аргументов;
- не поддерживаются static-переменные внутри функции;
- **__device__** функция не поддерживает взятие адреса;
- **__global__** обозначает **kernel** и соответствующая функция должна возвращать значение типа **void**;
- **__device__** функция является встроенной.
  Можно использовать **__noinline__**
  — тело функции должно быть в том же файле;
  — компилятор не среагирует, если функция с параметрами–указателями, или большое число аргументов.

Директива вызова ядра

```
Func <<< grid, block[, bytes[, streamid]] >>> ([arg1, ...]);
```

grid — переменная/значение типа dim3, задаёт размер grid'а (в блоках);
block — переменная/значение типа dim3, задаёт размер блока (в нитях);
bytes — переменная/значение типа size_t, задаёт дополнительный объём shared-памяти, которая должна быть динамически выделена; **streamid** — переменная/значение типа cudaStream_t, задаёт поток (CUDA stream), в котором должен произойти вызов, по умолчанию — 0.

Пример

```
__global__ void kernel( void )
{
}
int main( void )
{
  kernel <<< 16,32 >>> ();
  return 0;
}
```

Добавлены 1,2,3,4-мерные вектора из базовых типов:

- (u)char1, (u)char2, (u)char3, (u)char4,
- (u)short1, (u)short2, (u)short3, (u)short4,
- (u)int1, (u)int2, (u)int3, (u)int4,
- (u)long1, (u)long2, (u)long3, (u)long4,
- float1, float2, float3, float4,
- longlong1, longlong2,
- double1, double2.

Не поддерживаются векторные покомпонентные операции — это необходимо явно делать для каждой компоненты.

Для создания значений-векторов заданного типа **typeName** служит конструкция вида **make_<typeName>**.
Обращение к компонентам вектора производится по именам — **x, y, z** и **w**.

```
int2    a = make_int2   ( 1, 7 );
int2    b = make_int2   ( 2, 0 );

// a = a + b
a.x += b.x;
a.y += b.y;

float4  c = make_float4 ( 1, 2, 3.4f, 0.1f );

c.y = a.x + c.x;
a.x = a.y + (int)c.z;
c.x += c.y + c.z + c.w;
```

dim3 — uint3 тип для задания размерности. Обладает конструктором, инициализирующим все не заданные компоненты единицами:

```
dim3 varname ( [ x [, y [, z ] ] ] );
```

Пример:

```
dim3 blocks ( 16, 16 ); // blocks ( 16, 16, 1 )
dim3 grid ( 256 );  // grid ( 256, 1, 1 )

int blockDimX = blocks.x;
```

Добавлены следующие специальные переменные:

- gridDim — размер grid'a (имеет тип dim3)
- blockDim — размер блока (имеет тип dim3)
- blockIdx — индекс текущего блока в grid'е (имеет тип uint3)
- threadIdx — индекс текущей нити в блоке (имеет тип uint3)
- warpSize — размер warp'a (имеет тип int)

Ограничения:

- нельзя брать адрес встроенных переменных
- нельзя присвоить значение встроенной переменной

CPU C code:

```
float * a;
...
for ( int i = 0; i < N; i++ ) {
  a [i] = a [i] + 1.5f;
}
```

CUDA C code:

```
__global__ void kernel ( float * a, int N )
{
  // global thread index
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if ( i < N )
    a [i] = a [i] + 1.5f;
}
```

Спецификаторы переменных

Указывают на область видео–памяти, где они будут храниться.

- **__device__**
- **__constant__**
- **__shared__**

Пример:

```
__device__ int a[100];
```

__device__

Описывает переменную, которая будет находиться на устройстве.
Может быть уточнен другими спецификаторами.
Переменная:

- если не указано особо, то находится в глобальной памяти
- имеет время жизни всего приложения
- доступна всем потокам и хосту через runtime library

__constant__

опционально используется с **__device__**
Переменная:

- находится в константной памяти
- имеет время жизни всего приложения
- доступна всем потокам и хосту через runtime library

__shared__

опционально используется с **__device__**
Переменная:

- находится в разделяемой памяти блока потоков
- имеет время жизни блока
- доступна только потокам этого блока

```
short array0[128];
float array1[64];
int   array2[256];
```

```
extern __shared__ char array[];
__device__ void Func()
{
  short * array0 = (short*) array;
  float * array1 = (float*) &array0[128];
  int *   array2 = (int*)   &array1[64];
}
```

Ограничения на спецификаторы переменных

- не могут быть применены к полям структуры (struct или union)
- не могут быть применены к формальным и локальным переменным функций, исполняемых на хосте
- **__device__** и **__constant__** переменные могут использоваться только в пределах одного файла, их нельзя объявлять как extern
- запись в переменные типа **__constant__** может осуществляться только CPU при помощи специальных функций
- **__shared__** переменные не могут быть инициализированы при объявлении

CUDA поддерживает все математические функции из стандартной библиотеки C. (sin)
Лучше использовать float–аналоги стандартных функций. (sinf)
CUDA предоставляет специальный набор функций пониженной точности. (__sinf)
Для ряда функций можно задать способ округления при помощи суффикса:

- rn — округление к ближайшему
- rz — округление к нулю
- ru — округление вверх
- rd — округление вниз

```
float x = 1.72; __sinf_rz(x);
```

Есть ряд оптимизированных функций для работы с целыми числами.

```
#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 ( __fsqrt_rz(x) );
}
int main ( int argc, char *argv [] ) {
  float a [N];
  float * dev = NULL;
  dim3 threadsPerBlock = dim3 ( 128, 1 );
  dim3 blocksPerGrid ( N / threadsPerBlock.x, 1 );
  cudaMalloc ( (void**) & dev, N * sizeof ( float ) );
  kernel <<< threadsPerBlock, blocksPerGrid >>> ( dev );
  cudaMemcpy ( a, dev, N * sizeof ( float ),
      cudaMemcpyDeviceToHost );
  cudaFree ( dev );
  return 0;
}
```

```
float a [N];
float * dev = NULL;
// allocate memory on the GPU by N elements
cudaMalloc ( (void**) & dev, N * sizeof ( float ) );

// run N threads in blocks of 128 threads
// performed on the thread function - kernel
// data array - dev

dim3 threadsPerBlock = dim3 ( 128, 1 );
dim3 blocksPerGrid ( N / threadsPerBlock.x, 1 );
kernel <<< threadsPerBlock, blocksPerGrid >>> ( dev );

// copy data from GPU memory (DRAM) to CPU memory
cudaMemcpy ( a, dev, N * sizeof ( float ),
      cudaMemcpyDeviceToHost );

cudaFree ( dev ); // release GPU memory
```

```
#define N (1024*1024)
__global__ void kernel ( float * data ) {
  // number (id) of the current thread
  int idx = blockIdx.x * blockDim.x + threadIdx.x;

  float x = 2.0f * 3.1415926f * (float) idx / (float) N;
  // find the value and write it into array
  data [idx] = sinf ( __fsqrt_rz(x) ); //
}
```

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

Асинхронные функции

Асинхронность — управление возвращается до реального завершения требуемой операции.

- запуск ядра
- функции копирования памяти, имена которых оканчиваются на Async
- функции копирования памяти device $\leftrightarrow$ device
- функции инициализации памяти

Возвращаемые значения функций

Каждая функция CUDA runtime API (кроме запуска ядра) возвращает значение типа cudaError_t.
При успешном выполнении возвращается cudaSuccess, иначе — код ошибки.
Получить описание ошибки в виде строки по ее коду можно с помощью функции:

```
char * cudaGetErrorString ( cudaError_t code );
```