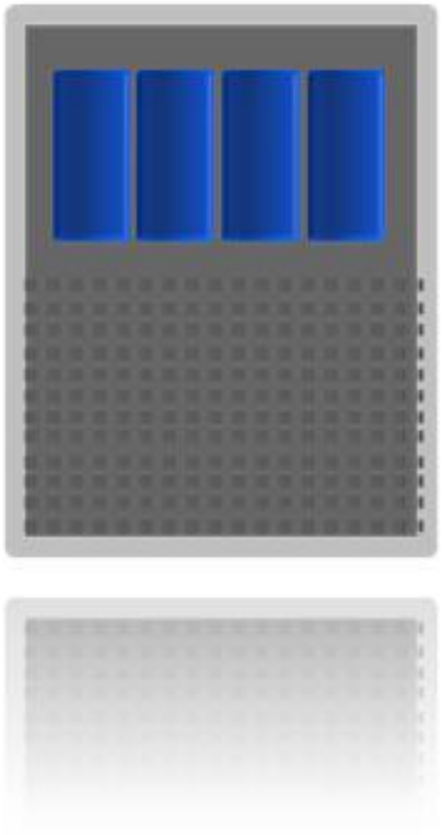
Лекция 1

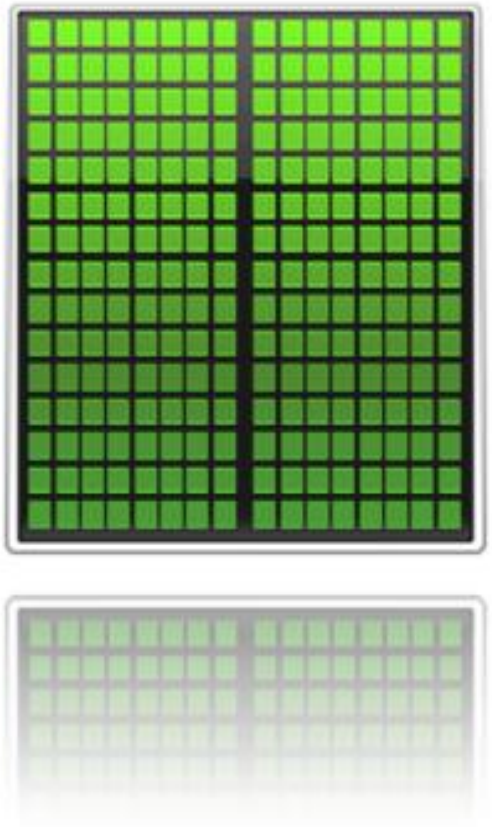
Принципы работы графических ускорителей. Программная модель CUDA

Перепёлкин Е.Е.

Отличия CPU от GPU

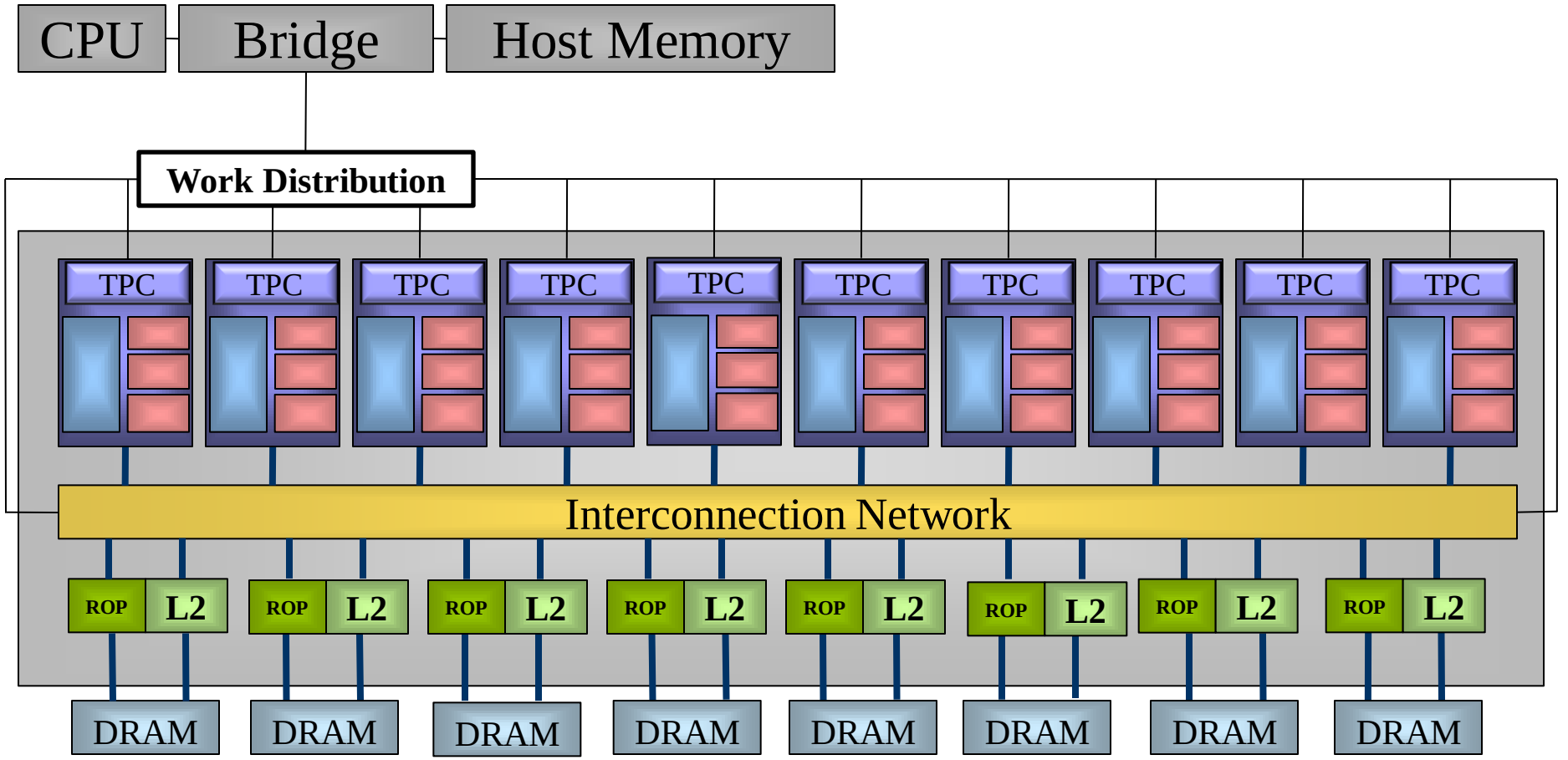


CPU

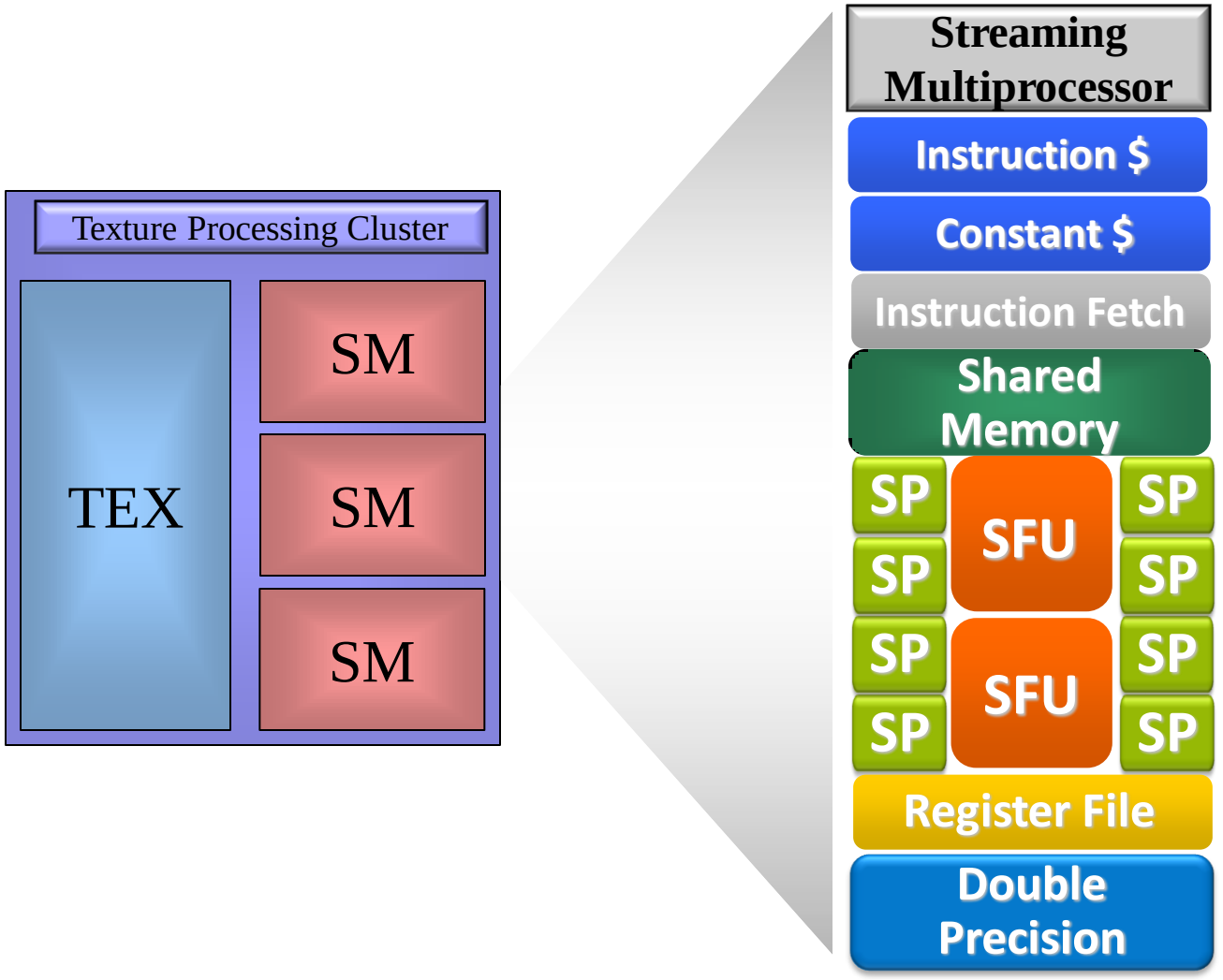


GPU

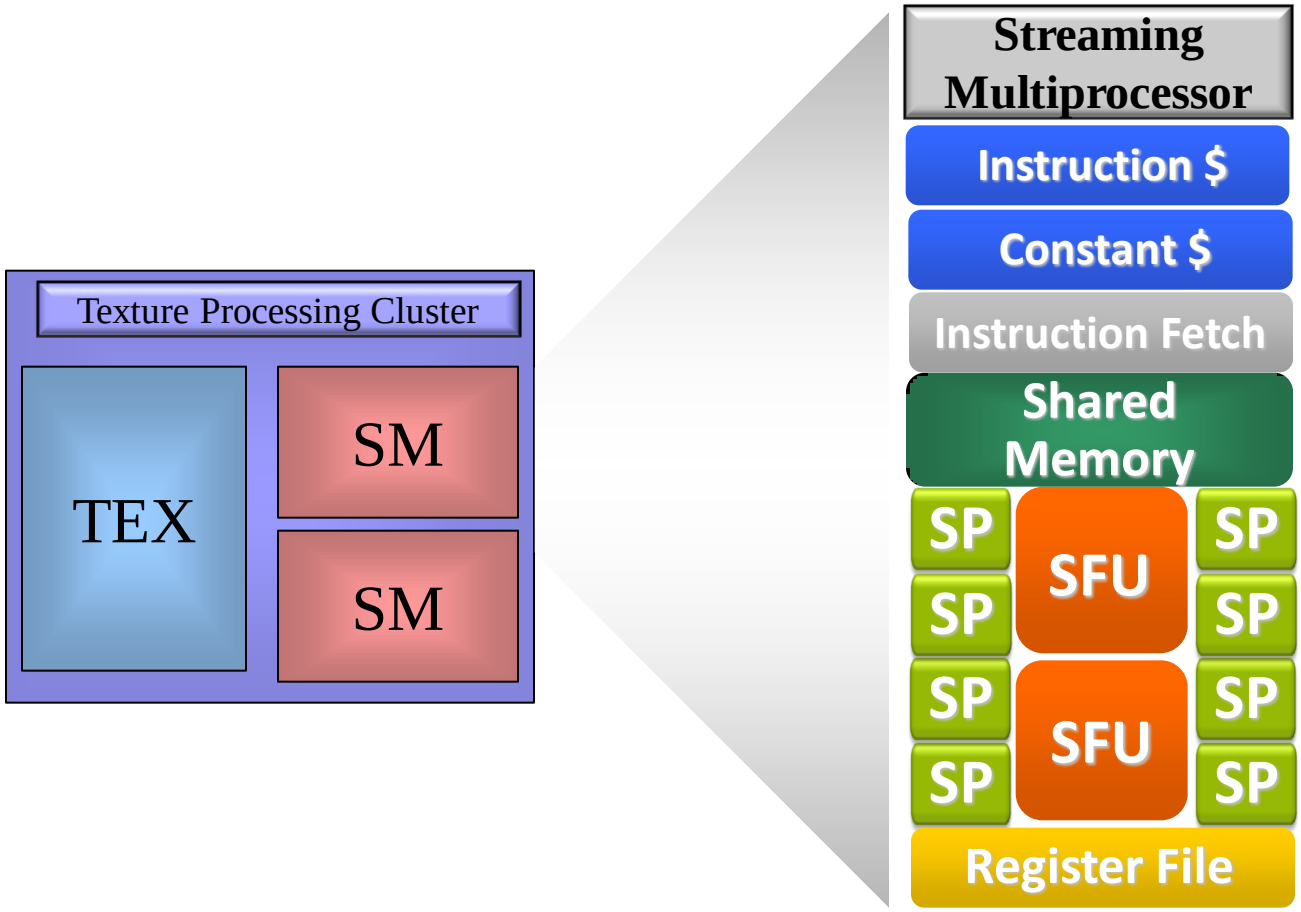
Архитектура Tesla 10



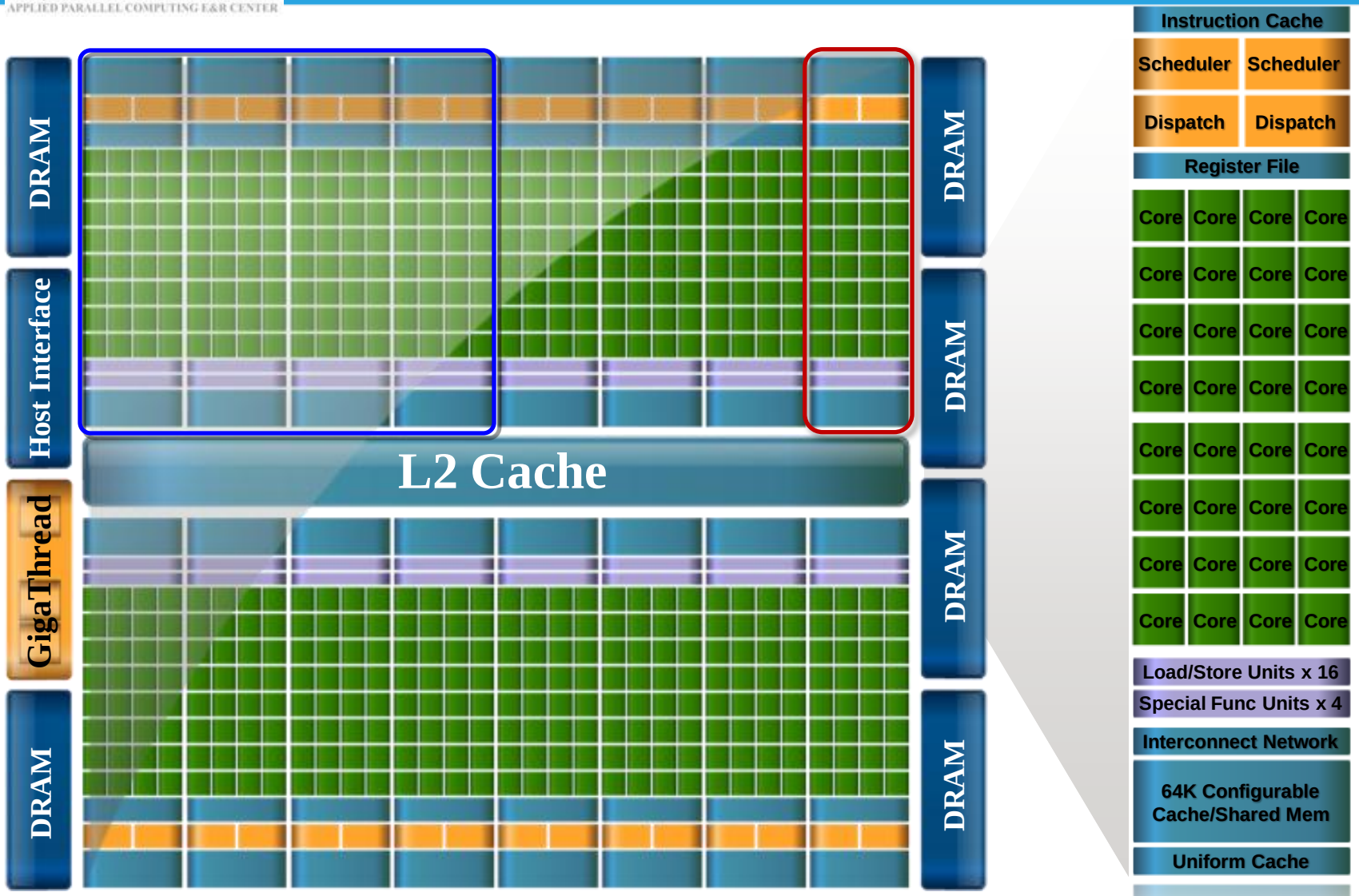
Архитектура Tesla 10



Архитектура Tesla 8



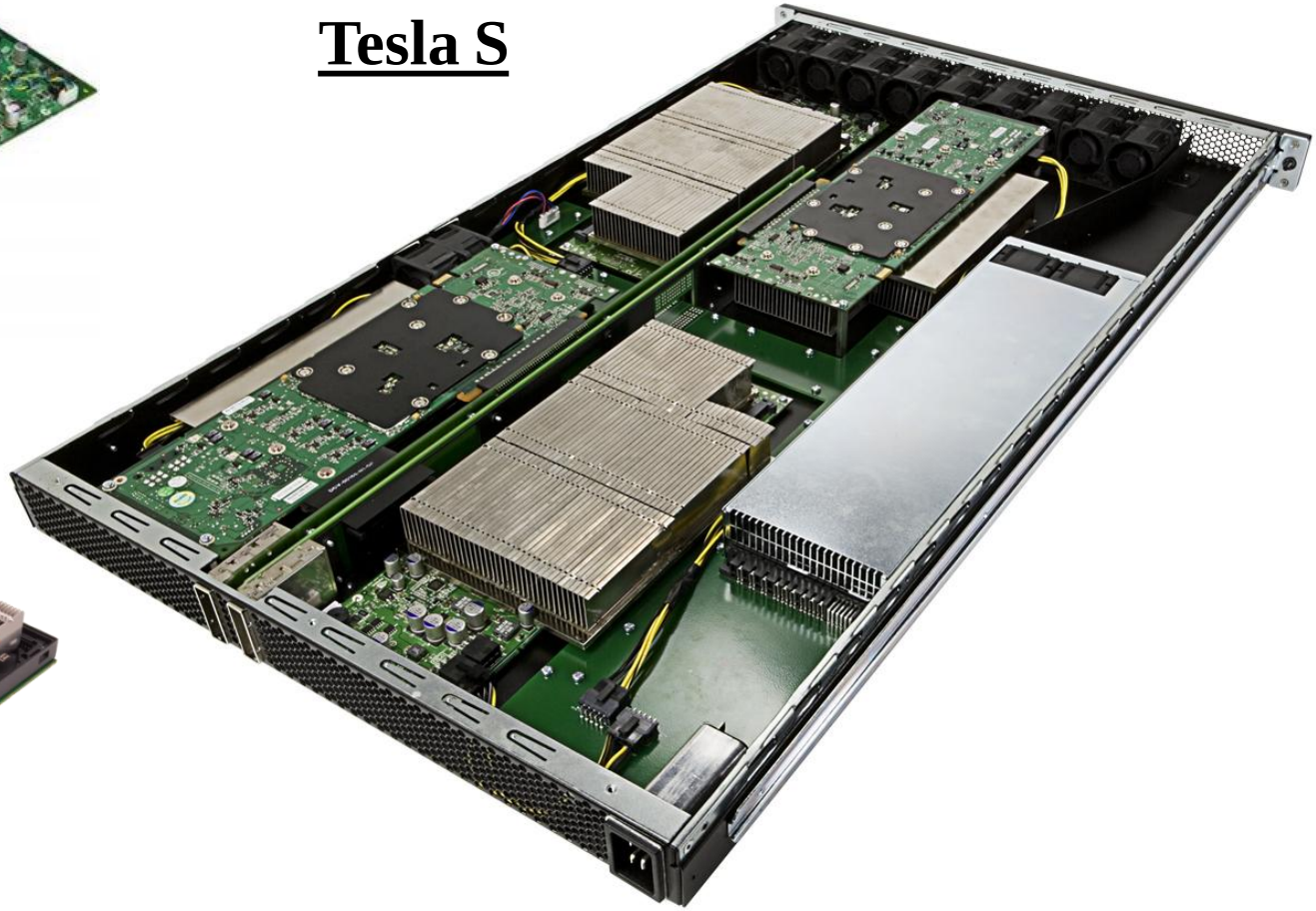
Архитектура Tesla 20 (Fermi)



NVIDIA Tesla

Tesla C

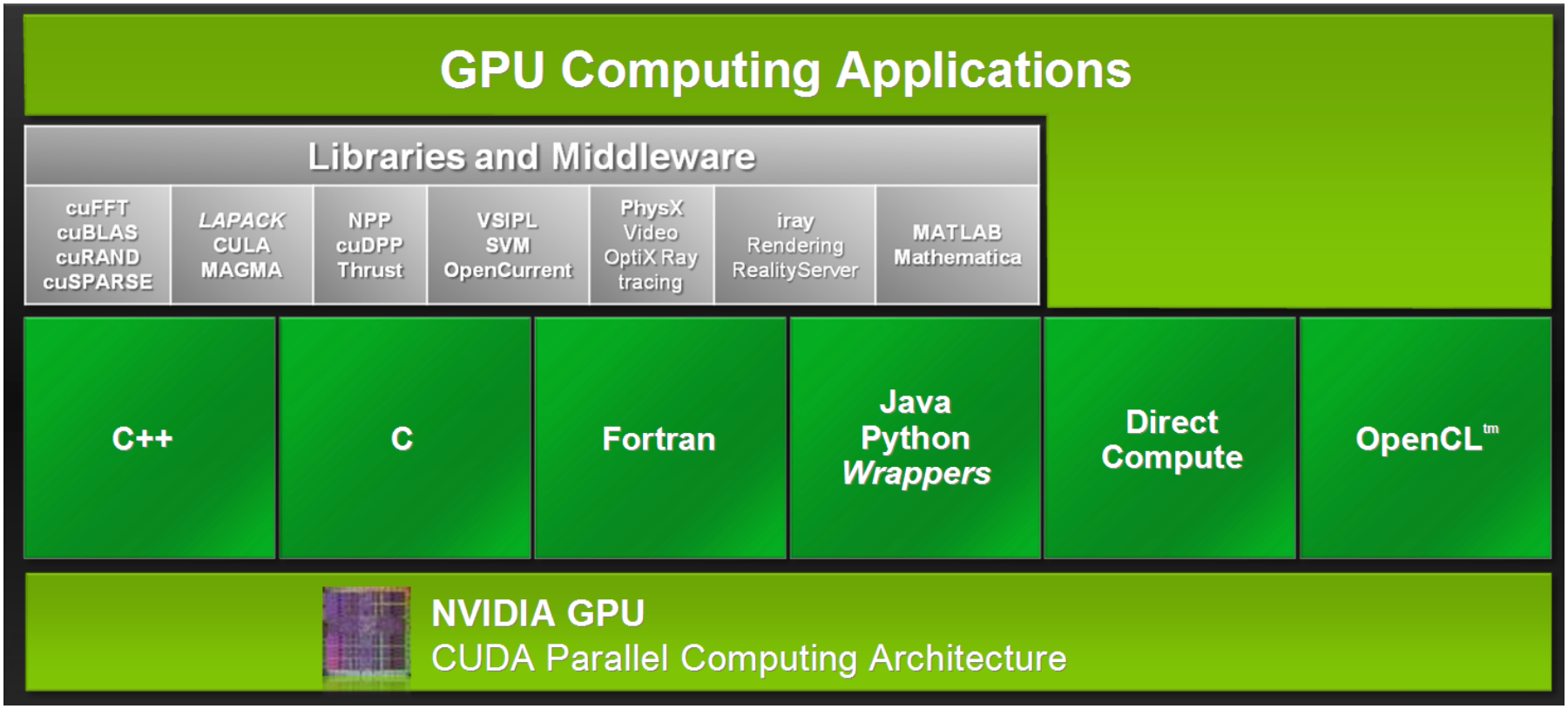
Tesla S



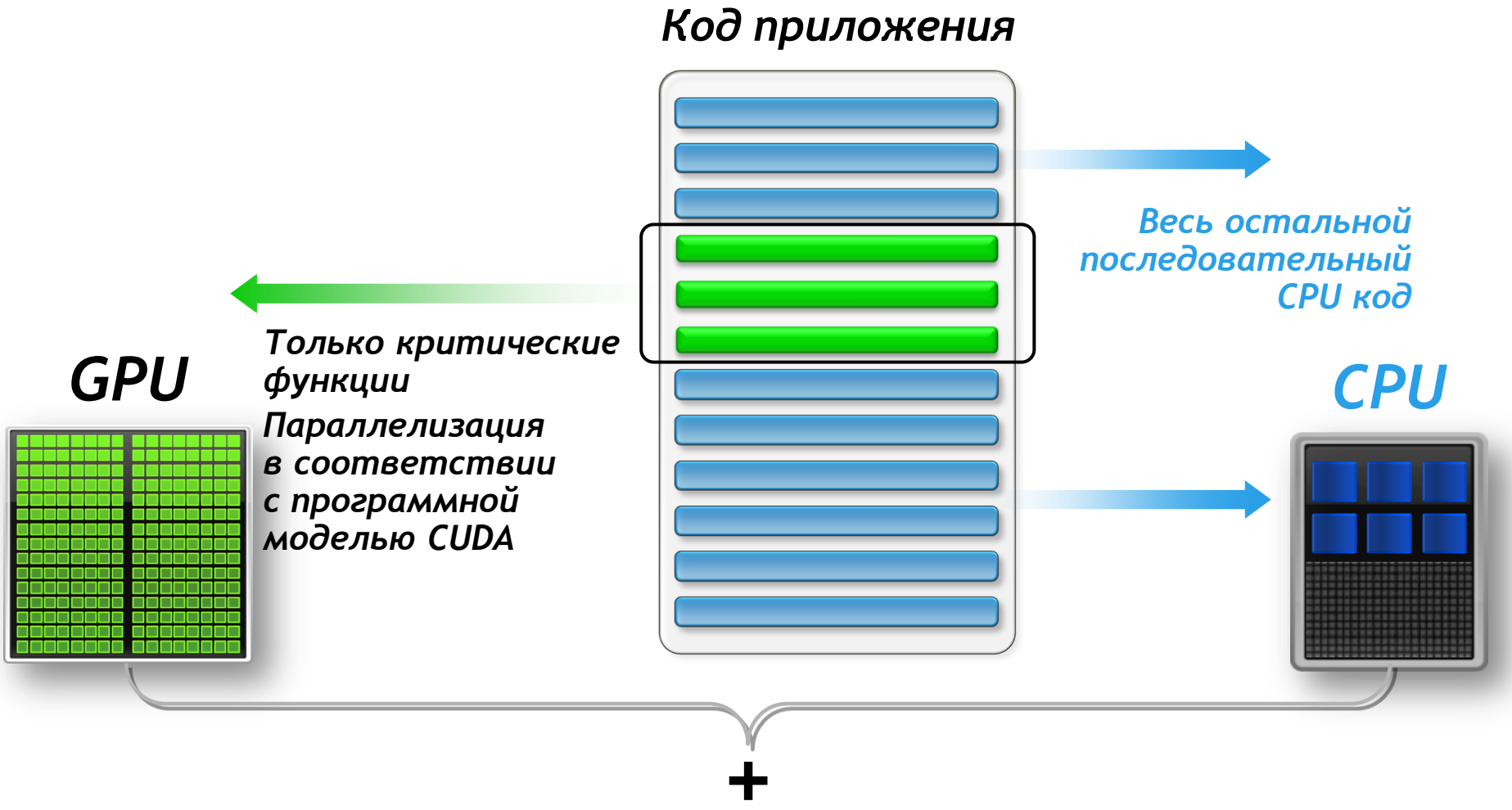
Tesla M

CUDA

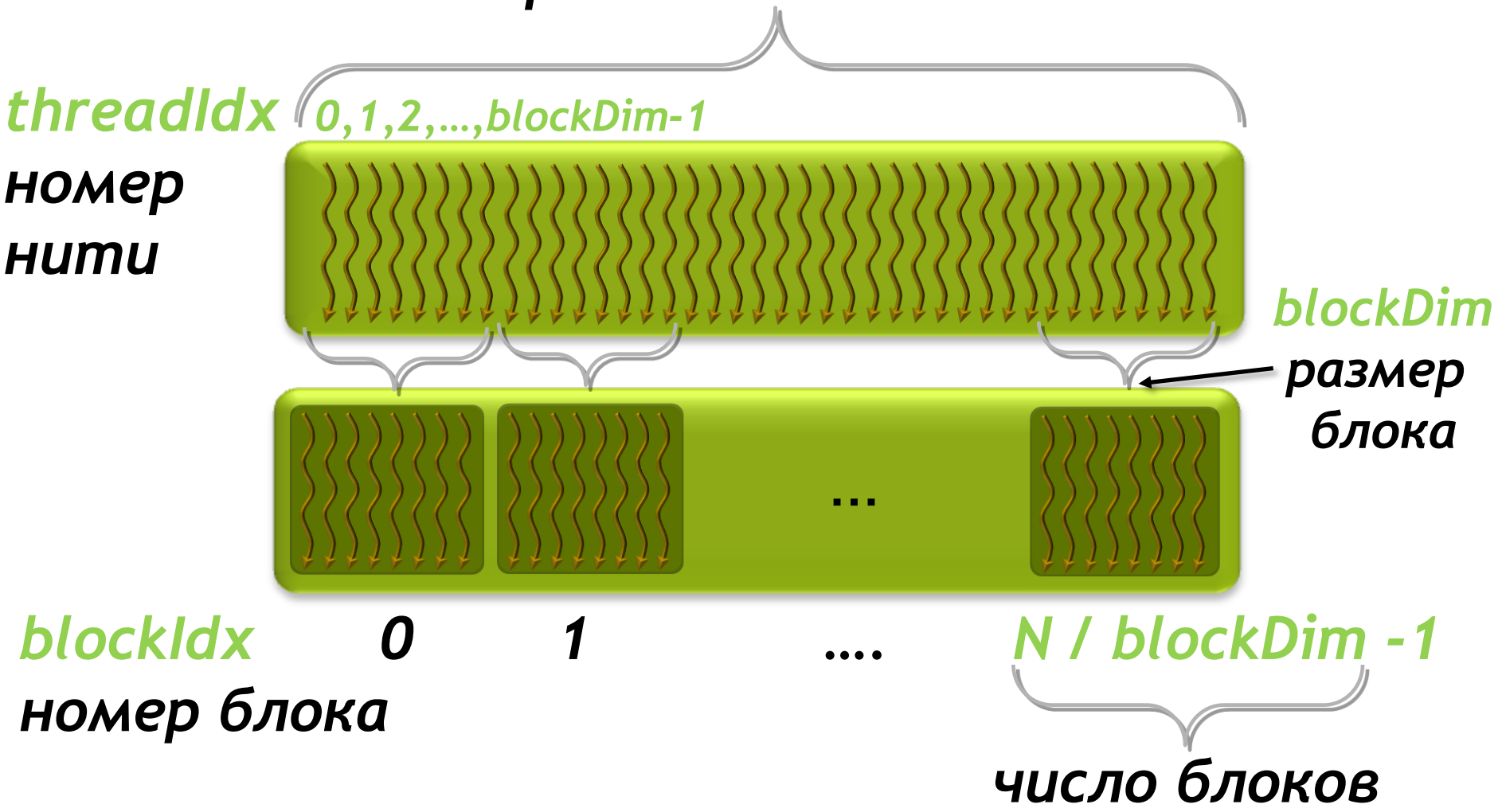
Compute Unified Device Architecture



С чего начать?



Группировка потоков

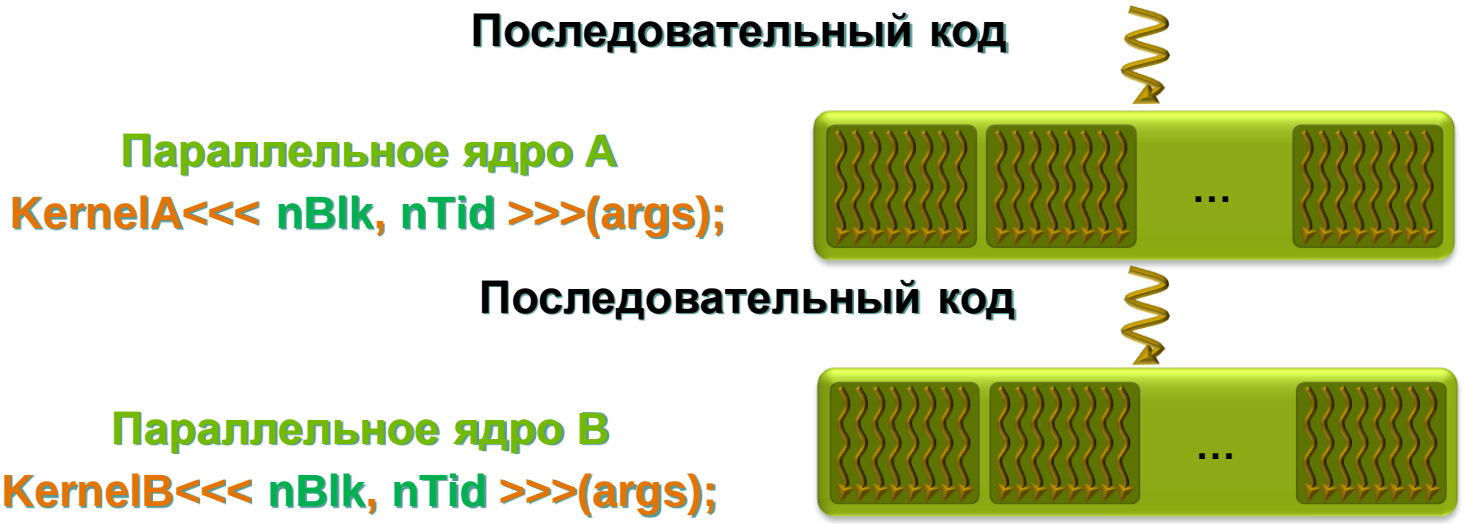


Смысл разбиения на блоки

- **Размер блока ограничен**
- **Блоки могут использовать shared память**
  - Так как блок целиком выполняется на одном SM
  - Объем shared памяти ограничен и зависит от HW
- **Внутри блока потоки могут синхронизироваться**
  - Так как блок целиком выполняется на одном SM

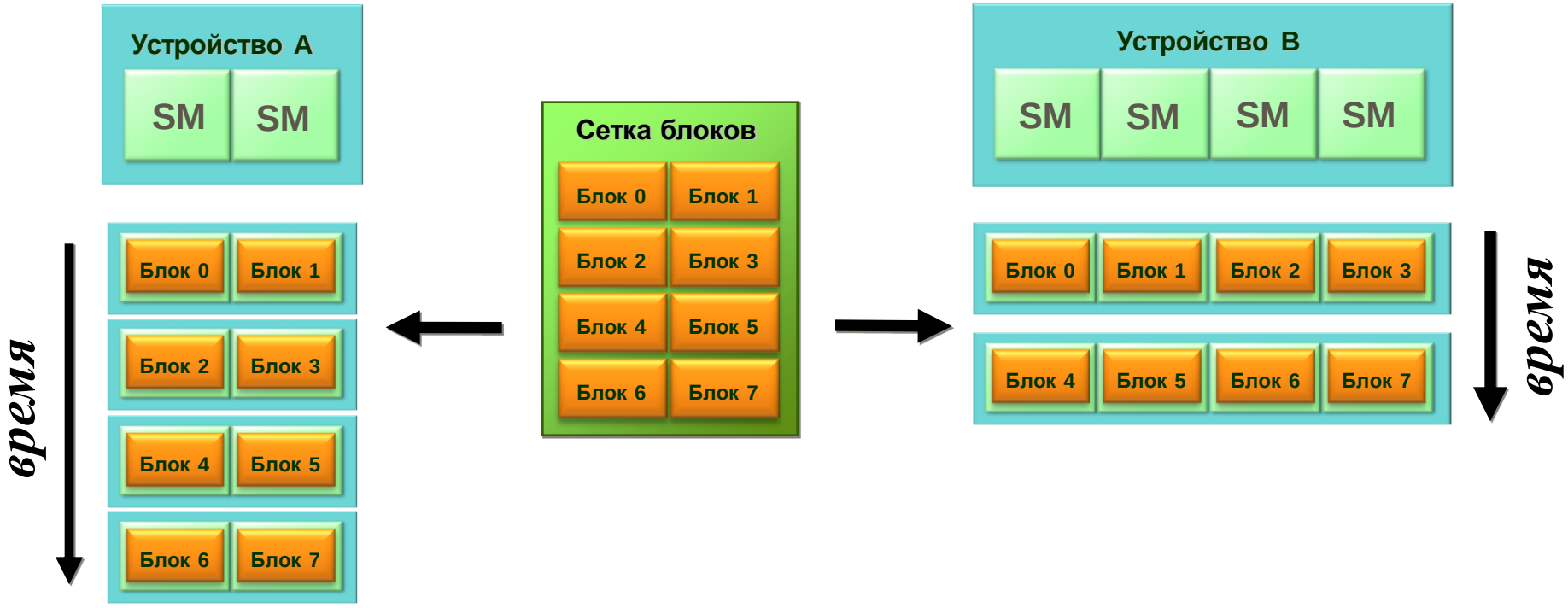
Гетерогенная структура кода

- Код состоит как из последовательных, так и из параллельных частей
- Последовательные части кода выполняются на CPU
- Массивно-параллельные части кода выполняются на GPU как функция-ядро (kernel function)



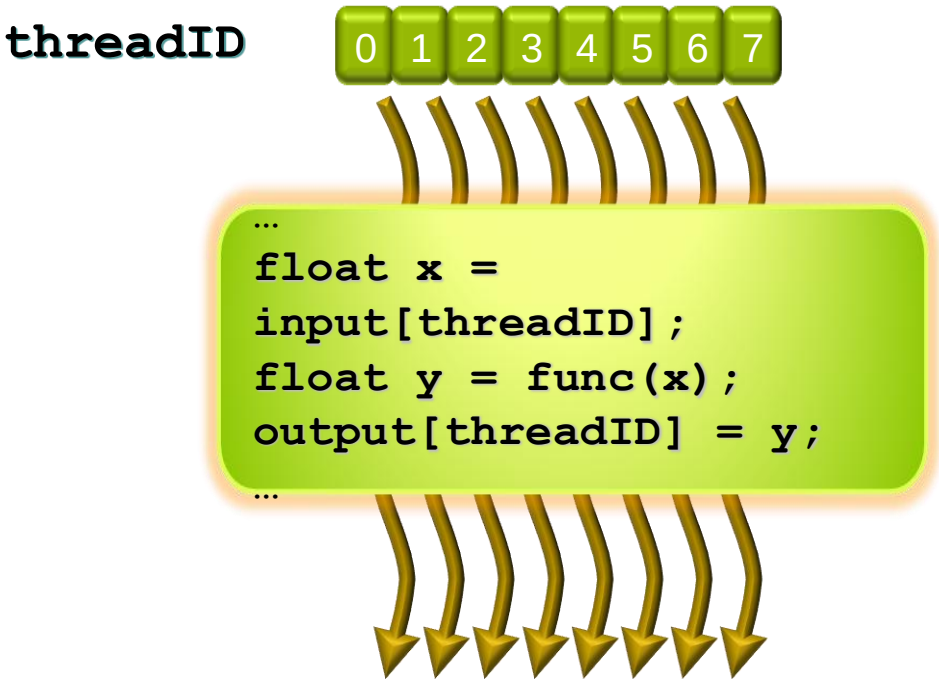
Прозрачная масштабируемость

- **Блоки могут быть распределены на любой процессор SM**
  - **код на CUDA масштабируется на любое количество ядер**



Выполнение одного блока

- **Ядро CUDA исполняется массивом потоков**
  - **Все потоки исполняют одну программу**
  - **Каждый поток использует свой индекс для вычислений и управления исполнением**



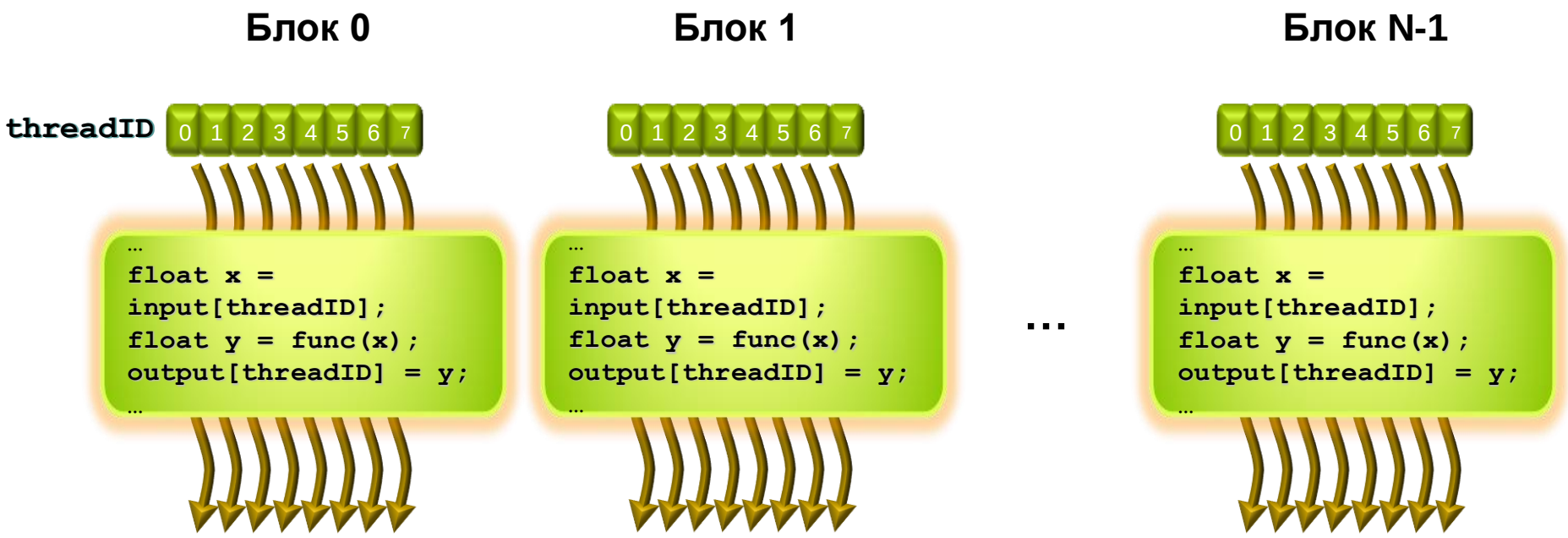
Взаимодействие потоков

- **Потоки могут быть не полностью независимы**
  - **Обмениваются результатами вычислений**
  - **Разделяют доступ к внешней памяти**

- **Возможность взаимодействия потоков – ключевая особенность программной модели CUDA**
  - **Потоки кооперируются, используя разделяемую память и примитивы синхронизации**

Выполнение сетки блоков

- Монолитный массив потоков разделяется на блоки
  - Потоки внутри блока взаимодействуют через разделяемую память
  - Потоки в разных блоках не могут синхронизироваться
- Позволяет программам прозрачно масштабироваться

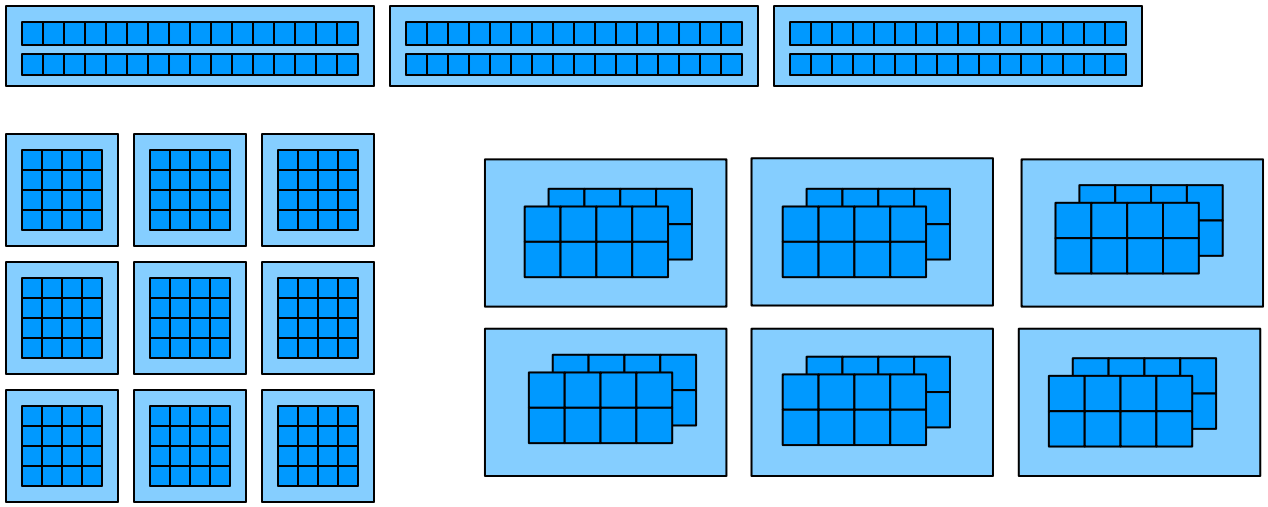


Иерархия нитей

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

Топология сети (grid)

- **Потоки в CUDA объединяются в блоки:**
  - **Возможна 1D, 2D, 3D топология блока и нитей**
  - Общее кол-во потоков в блоке ограничено
    - В Tesla 10 максимальное число потоков в блоке 512 (1024)
    - В Tesla 20 – 1536 потоков (?)



Номер нити в сети

- **Размер 3D блока**
  - **(D_x,D_y,D_z) → (blockDim.x, blockDim.y, blockDim.z)**
- **Координаты 3D нити внутри блока**
  - **(x,y,z) → (threadIdx.x, threadIdx.y, threadIdx.z)**
- **Линейный номер нити (потока) в блоке**
  - **ID thread = $x + y * D_x + z * D_x * D_y$**

Программная модель CUDA

- Выбор топологии сети (grid)

1D

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

2D

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

3D

```
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);
    }
```

CUDA 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	R
__constant__	device	device / host	R / W
__shared__	device	block	RW / **__syncthreads()**

Расширения языка C

```
int2   a = make_int2 ( 1, 7 );
float4 b = make_float4 ( a.x, a.y, 1.0f, 7 );
float2 x = make_float2 ( b.z, b.w );
dim3   grid   = dim3 ( 10 );
dim3   blocks = dim3 ( 16, 16 );
```

Для векторов не определены покомпонентные операции

Для double и longlong возможны только вектора размера 1 и 2.

Расширения языка C

Общий вид команды для запуска ядра

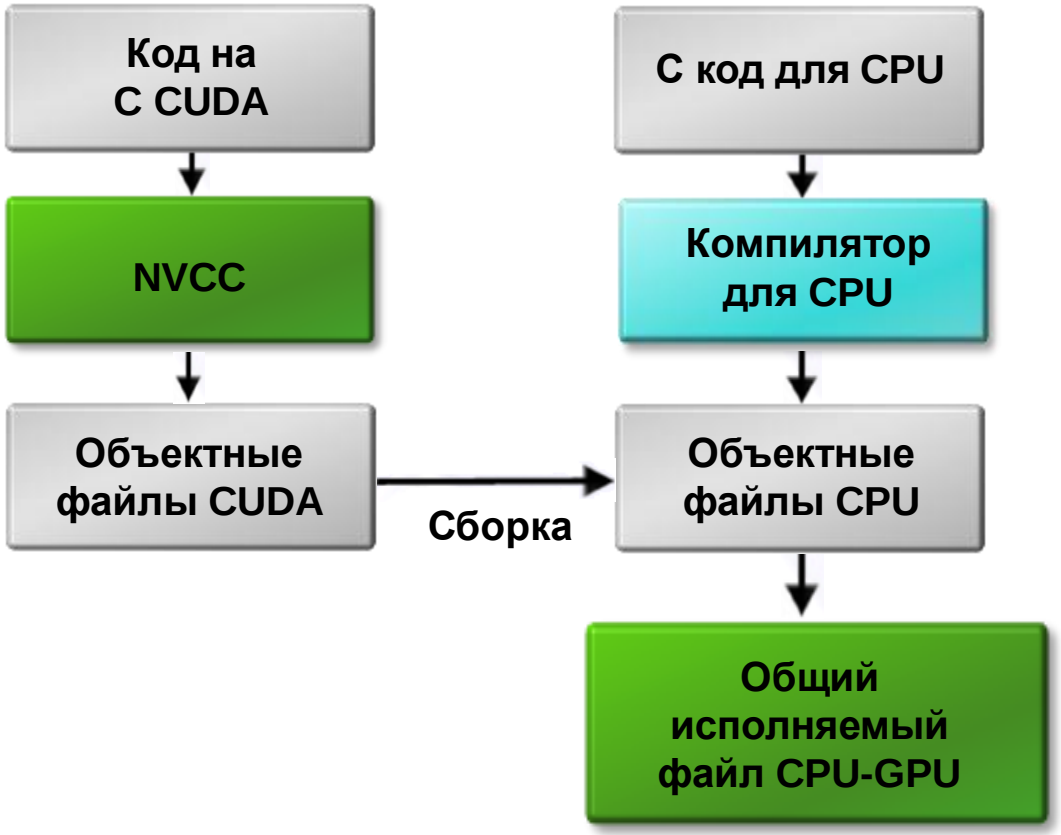
```
incKernel<<<bl, th, ns, st>>> ( data );
```

- *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 (например, sin) на их быстрые (но менее точные) аналоги (__sin)
  - **-o <outputFileName>** - задать имя выходного файла
- **CUDA файлы обычно носят расширение .cu**
- **Используем утилиту make/nmake, явно вызывающую nvcc**
- **Используем MS Visual Studio**
  - Подключаем cuda.rules
  - Используем CUDA Wizard (http://sourceforge.net/projects/cudawizard)

Процесс сборки приложения

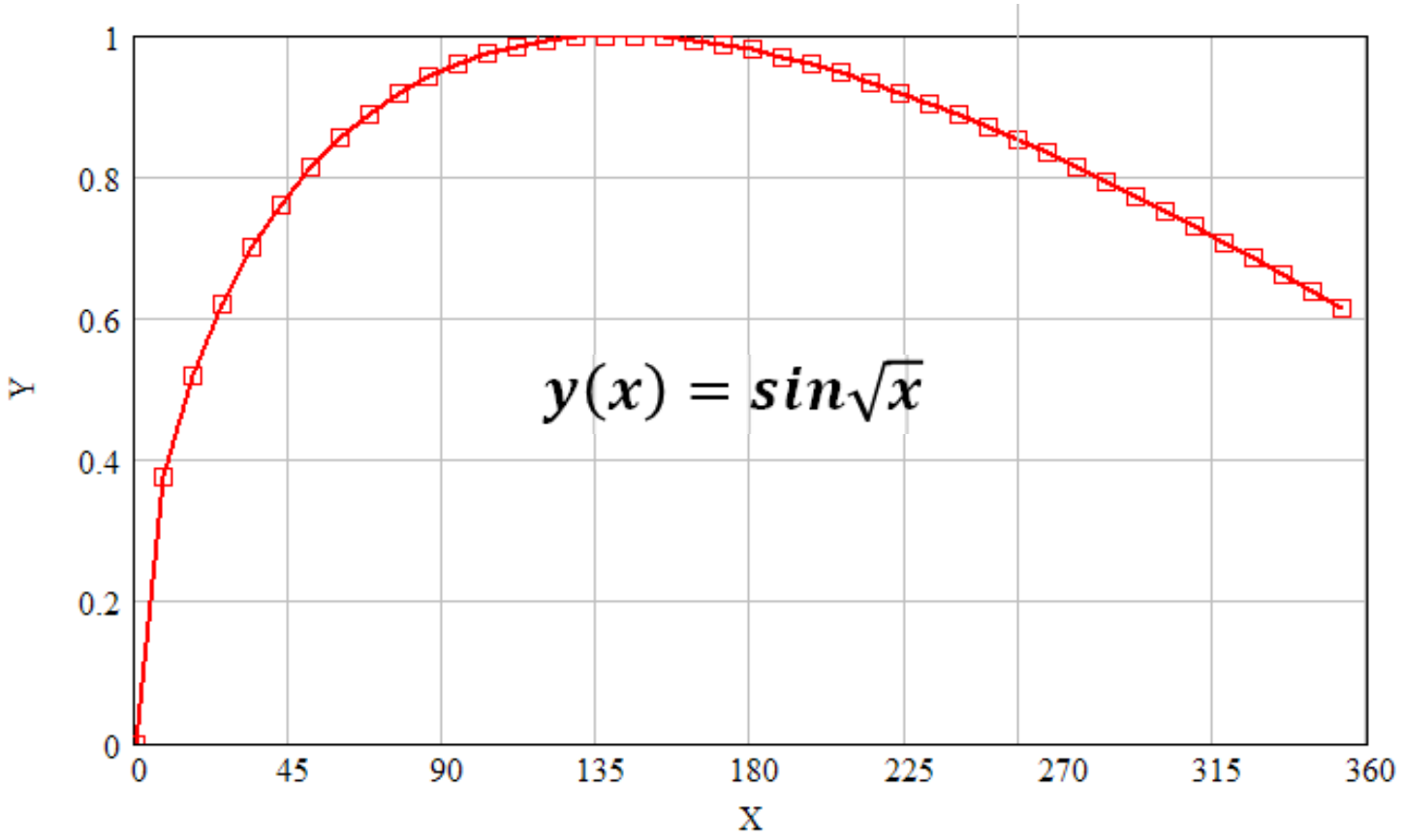


Получение информации о GPU

```
int main ( int argc, char *  argv [] )
{
    int           deviceCount;
    cudaDeviceProp         devProp;

    cudaGetDeviceCount ( &deviceCount );
    printf             ( "Found %d devices\n", deviceCount );

    for ( int device = 0; device < deviceCount; device++ )
    {
       cudaGetDeviceProperties ( &devProp, device );
       printf ( "Device %d\n", device );
       printf ( "Compute capability     : %d.%d\n", devProp.major, devProp.minor );
       printf ( "Name                   : %s\n", devProp.name );
       printf ( "Total Global Memory    : %d\n", devProp.totalGlobalMem );
       printf ( "Shared memory per block: %d\n", devProp.sharedMemPerBlock );
       printf ( "Registers per block    : %d\n", devProp.regsPerBlock );
       printf ( "Warp size              : %d\n", devProp.warpSize );
       printf ( "Max threads per block  : %d\n", devProp.maxThreadsPerBlock );
       printf ( "Total constant memory  : %d\n", devProp.totalConstMem );
    }
     return 0;
}
```

Задача

Построить график функции $y(x)$ *на отрезке* $[0, 2\pi]$



CUDA «Hello World»

```
#define       N      (1024*1024)

__global__ void kernel ( float * data )
{
                  [0…2047]        [512]          [0…511]
 int   idx = blockIdx.x * blockDim.x + threadIdx.x;

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

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

}
```

CUDA «Hello World»

```
int main ( int argc, char *  argv [] )
{

 float * a;

 float * dev = NULL;

 a = ( float* ) malloc ( N * sizeof ( float ) );

 cudaMalloc ( (void**)&dev, N * sizeof ( float ) );

 kernel<<<N/512, 512>>> ( dev );

 cudaMemcpy ( a, dev, N * sizeof ( float ), cudaMemcpyDeviceToHost );

 for ( int idx = 0; idx < N; idx++ )
                              printf ( "a[%d] = %.5f\n", idx, a[idx] );

 free( a ); cudaFree( dev );

 return 0;

}
```

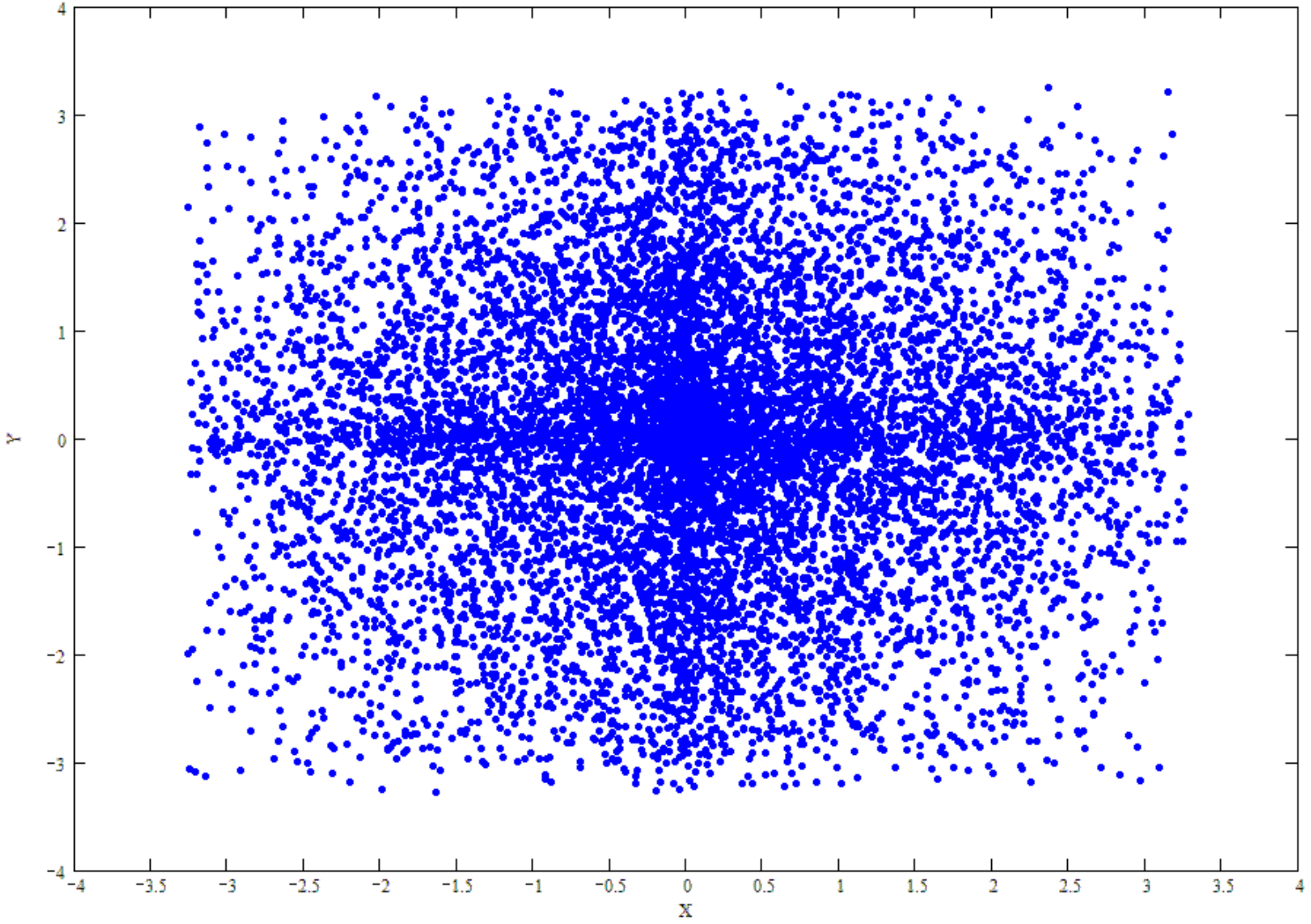
Задача N - тел

- Дано распределение N – точечных масс.
- Точечные массы движутся под действием гравитационного взаимодействия.
- Уравнения движения для численного моделирования имею вид:

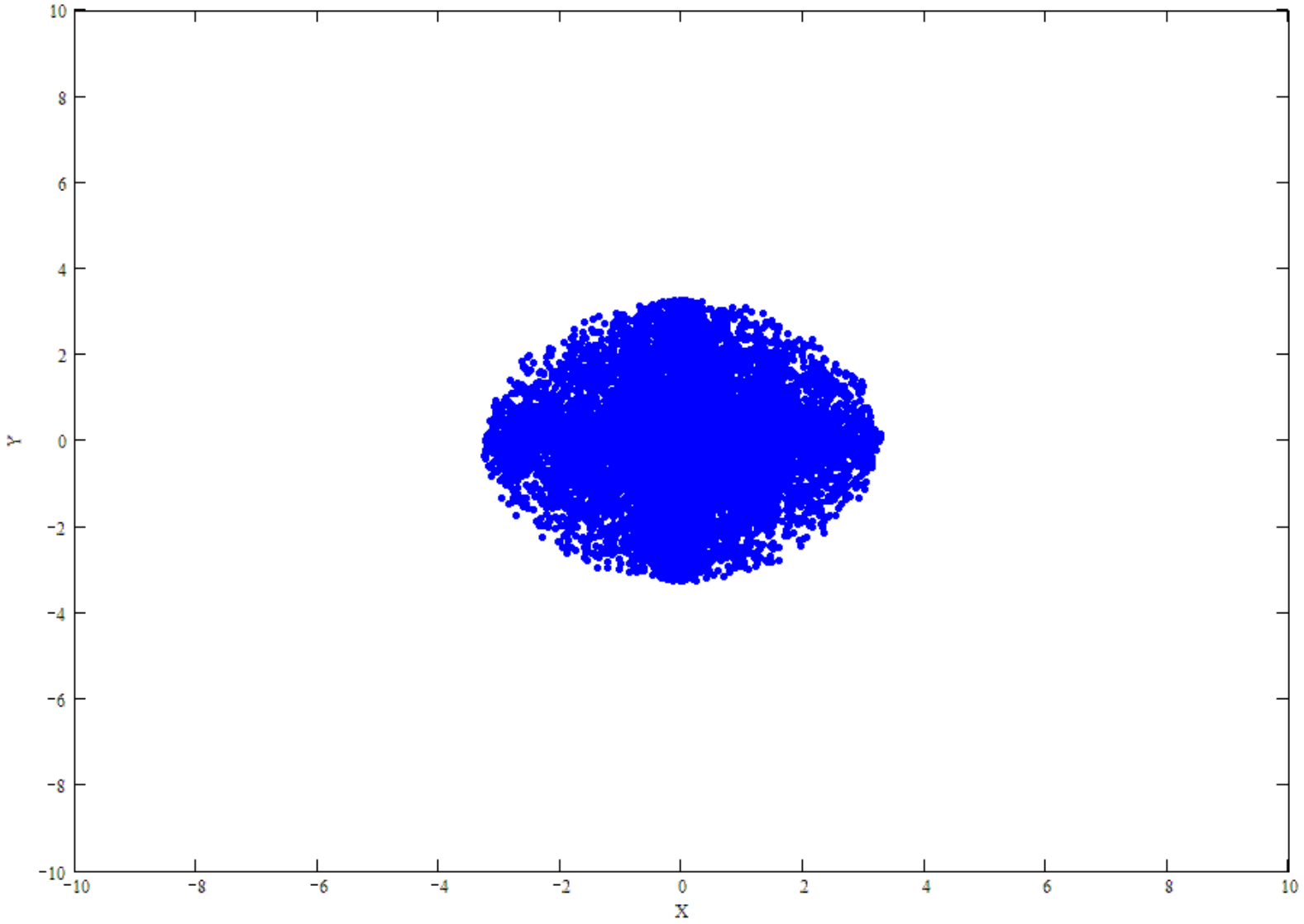
$$\vec{a}_{n,i} = \frac{\vec{F}_{n,i}}{m} = Gm \sum_{k \neq n}^{N} \frac{\vec{r}_k - \vec{r}_n}{|\vec{r}_k - \vec{r}_n|^3}$$

$$\vec{v}_{n,i+1} = \vec{v}_{n,i} + \vec{a}_{n,i}\tau \qquad \vec{r}_{n,i+1} = \vec{r}_{n,i} + \vec{v}_{n,i}\tau + \frac{\vec{a}_{n,i}\tau^2}{2}$$

Нулевые начальные условия



Не нулевые начальные условия



CPU вариант

```
void Acceleration_CPU ( float *X, float *Y, float *AX,
                        float *AY, int N, int id )
{
 float ax = 0.f;
 float ay = 0.f;
 float xx, yy, rr;

 for ( int j = 0; j < N;  j ++ )
 { if ( j ! = id )
    { xx = X [ j ] – X [ id ]; yy  = Y [ j ] – Y [ id ];
      rr = sqrtf ( xx * xx + yy * yy );
      if ( rr > 0.01f ) { rr  = 10.f / (rr * rr * rr );
                          ax + = xx * rr; ay + = yy * rr; }
    }
 }
 AX [ id ] = ax; AY [ id ] = ay;
}
```

CPU вариант

```
void Position_CPU  ( float *X, float *Y, float *VX,
                     float *VY,float *AX, float *AY,
                     float tau, int nt, int Np, int id)
{
  X [ id + nt * Np ] = X [ id + ( nt - 1 ) * Np ] +
  VX [ id ] * tau + AX [ id ] * tau * tau * 0.5f;

  Y [ id + nt * Np ] = Y [ id + ( nt - 1 ) * Np ] +
  VY [ id ] * tau + AY [ id ] * tau * tau * 0.5f;

  VX [ id ] + = AX [ id ] * tau;

  VY [ id ] + = AY [ id ] * tau;
}
```

GPU вариант 1

```
__global__ void Acceleration_GPU ( float *X, float *Y,
                                   float *AX, float *AY,  int N )
{int id = threadIdx.x + blockIdx.x*blockDim.x;
 float ax = 0.f;
 float ay = 0.f;
 float xx, yy, rr;

 for ( int j = 0; j < N;  j ++ )
 { if ( j ! = id )
    { xx = X [ j ] - X [ id ]; yy  = Y [ j ] - Y [ id ];
      rr = sqrtf ( xx * xx + yy * yy );
      if ( rr > 0.01f ) { rr  = 10.f / (rr * rr * rr );
                          ax + = xx * rr; ay + = yy * rr; }
    }
 }
 AX [ id ] = ax; AY [ id ] = ay;
}
```

GPU вариант 1

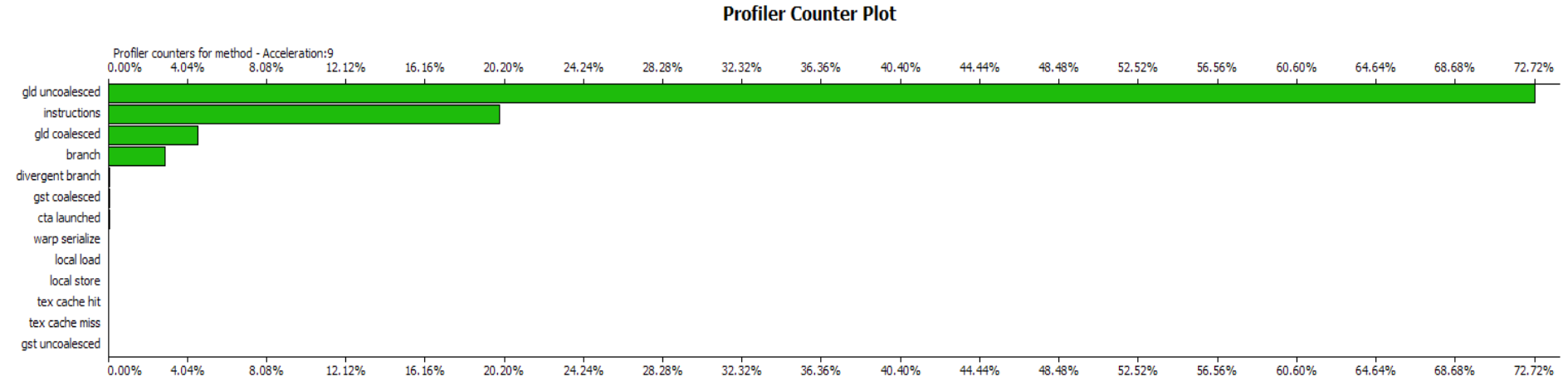
```
__global__ void Position_GPU  ( float *X, float *Y,
                  float *VX, float *VY, float *AX,
                  float *AY, float tau, int nt, int Np )
{
  int id = threadIdx.x + blockIdx.x*blockDim.x;

  X [ id + nt * Np ] = X [ id + ( nt - 1 ) * Np ] +
  VX [ id ] * tau + AX [ id ] * tau * tau * 0.5f;

  Y [ id + nt * Np ] = Y [ id + ( nt - 1 ) * Np ] +
  VY [ id ] * tau + AY [ id ] * tau * tau * 0.5f;

  VX [ id ] + = AX [ id ] * tau;

  VY [ id ] + = AY [ id ] * tau;
}
```

Оптимизация

- Имеется многократное считывание одних и тех же данных из глобальной памяти
- Функция-ядро **Acceleration_GPU** может быть существенно улучшена по средствам использования разделяемой памяти
- Вариант **Acceleration_GPU_Shared** рассмотрим на следующем занятии

Задача N-тел. Вариант 1

Kernel Acceleration



Kernel Position