Управление группами и коммуникаторами

Разделение:

```
int MPI_Comm_split(    MPI_Comm comm,
                       int color,
                       int key,
                       MPI_Comm *newcomm);
```

Делит процессы **comm** на непересекающиеся подгруппы, **по одной** для каждого значения **color**.

В пределах каждой подгруппы процессы **пронумерованы в порядке**, определенном значением **key**.

Если **color** = MPI_UNDEFINED, тогда **newcomm** = MPI_COMM_NULL.

На базе коммуникатора можно создать группу:

```
int MPI_Comm_group(      MPI_Comm comm,
                         MPI_Group *group);
```

На базе группы можно создать коммуникатор:

```
int MPI_Comm_create(     MPI_Comm comm,
                         MPI_Group group,
                         MPI_Comm *newcomm);
```

Для групп имеются возможности:

- включения/исключения процессов;
- объединения/пересечения/разности групп.

Виртуальные топологии MPI

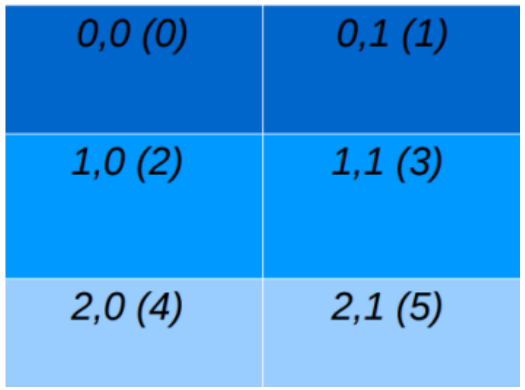
Топология в MPI ***является логической*** и никак не связана с физической средой передачи данных.

Поддерживаются два вида топологий:
– ***декартова топология***;
– ***граф***.

Декартова топология

Создание.

```
int MPI_Cart_create(  MPI_Comm oldComm,
                      int nDims,
                      int *dims,
                      int *periods,
                      int reorder,
                      MPI_Comm *grid_comm);
```



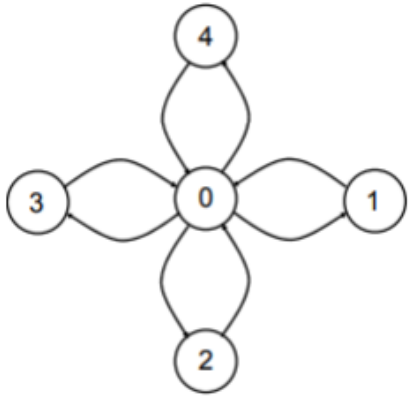
Работа.

- Определение координат по рангу.
- Определение ранга по координатам.
- Вычисление координат соседей.

Топология граф

Создание.

```
int MPI_Graph_create( MPI_Comm oldComm,
                      int nnodes,
                      int *index,
                      int *edges,
                      int reorder,
                      MPI_Comm *graphcomm);
```




```
int index[] = {4,5,6,7,8};
int edges[] = {1,2,3,4,0,0,0,0};
```

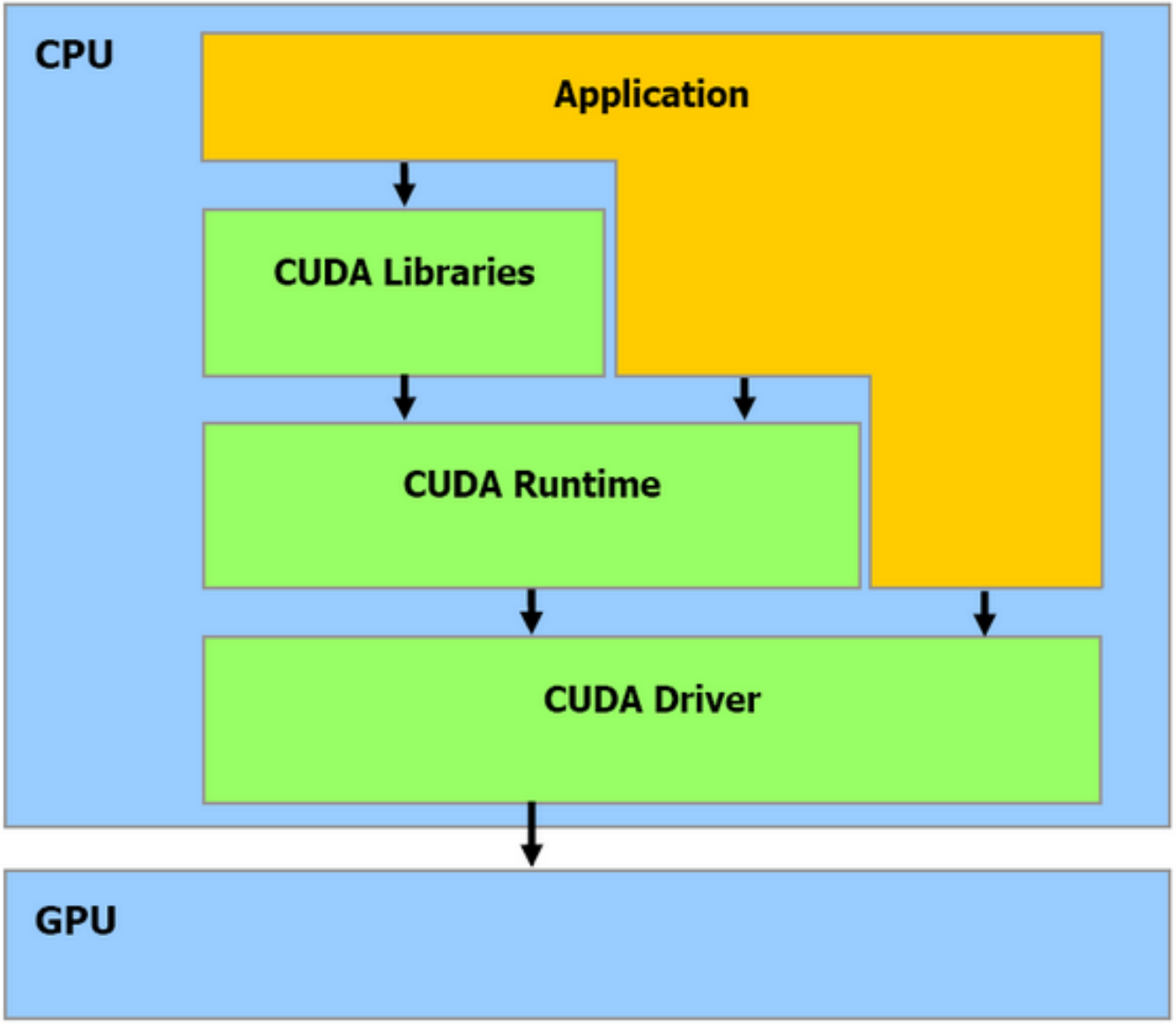
Работа.

- Получение рангов соседних вершин.

NVidia CUDA

(Compute Unified Device Architecture)

комбинация программных и аппаратных технологий

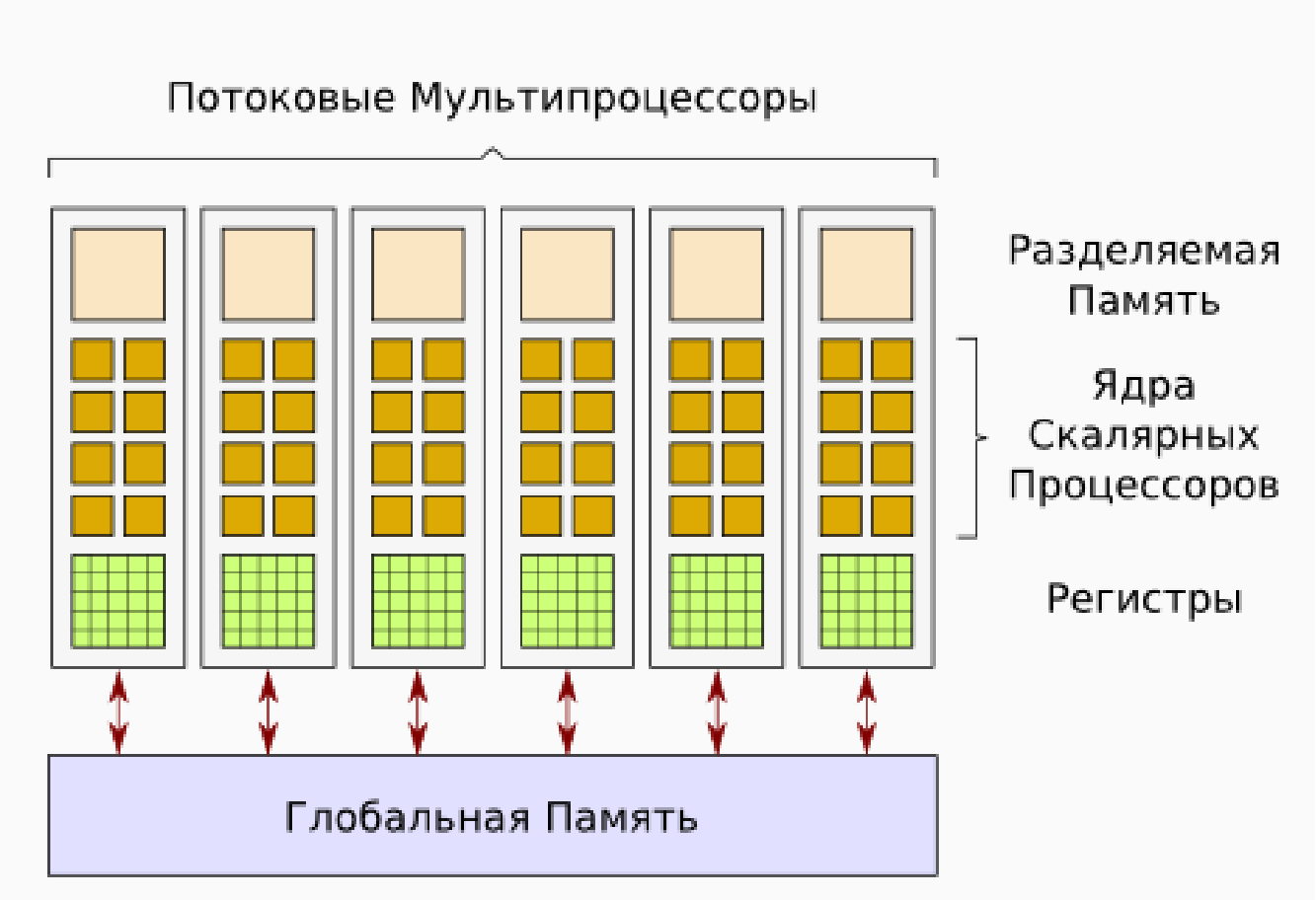


Библиотеки:
CUBLAS (Basic Linear Algebra Subprograms) - для вычислений задач линейной алгебры;
CUFFT (Fast Fourier Transform) - расчёт быстрого преобразования Фурье.

Высокоуровневый API: CUDA Runtime API;

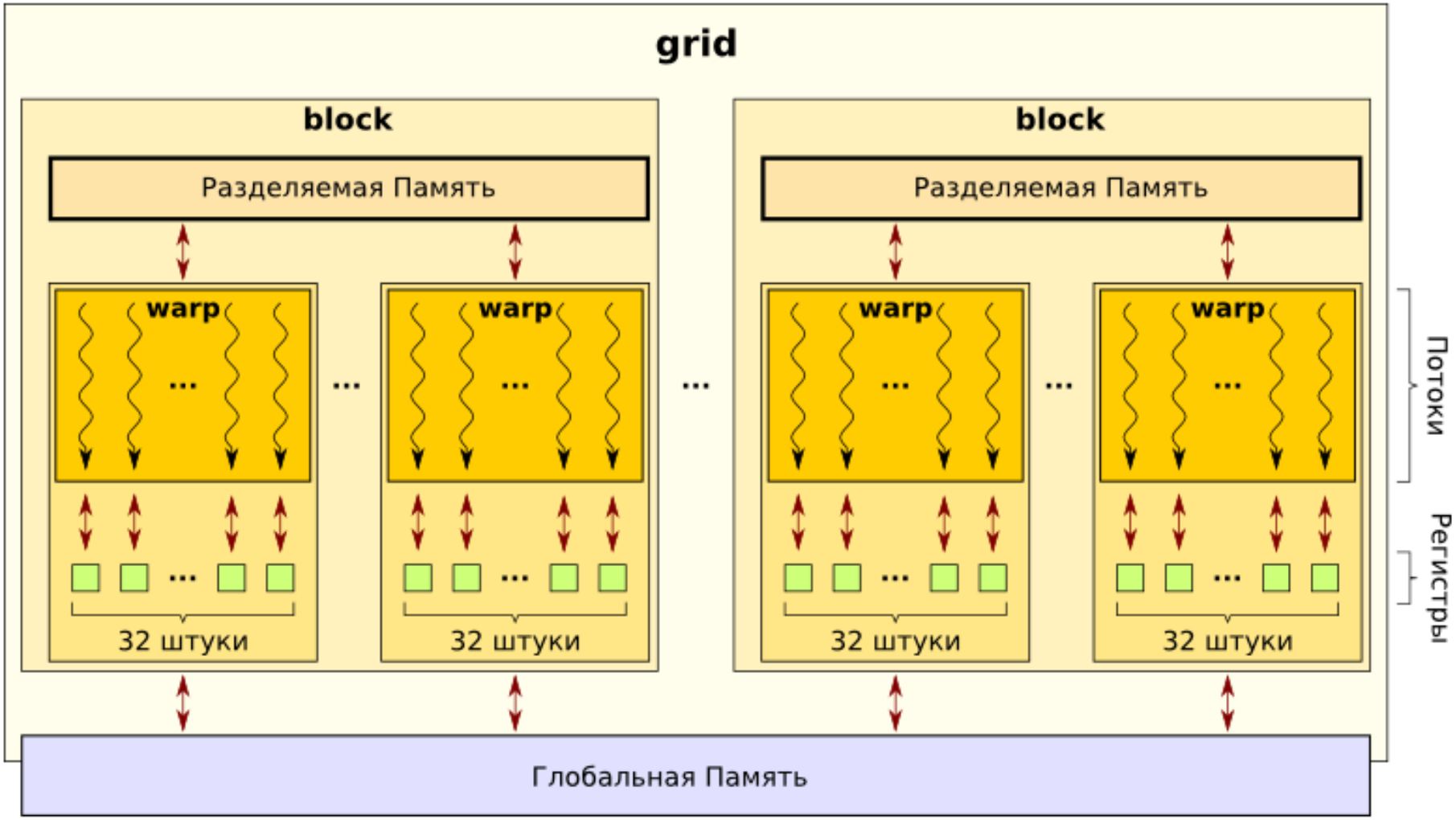
низкоуровневый API: CUDA Driver API.

Аппаратная часть



Sared Memory – разделяемая (общая) память **16 - 48 Кб**.

Иерархия потоков



- Все потоки warp’а выполняют **одинаковые инструкции**.
- Обращение **к памяти** для warp’а (или половины warp’а) может производится **одним запросом**.

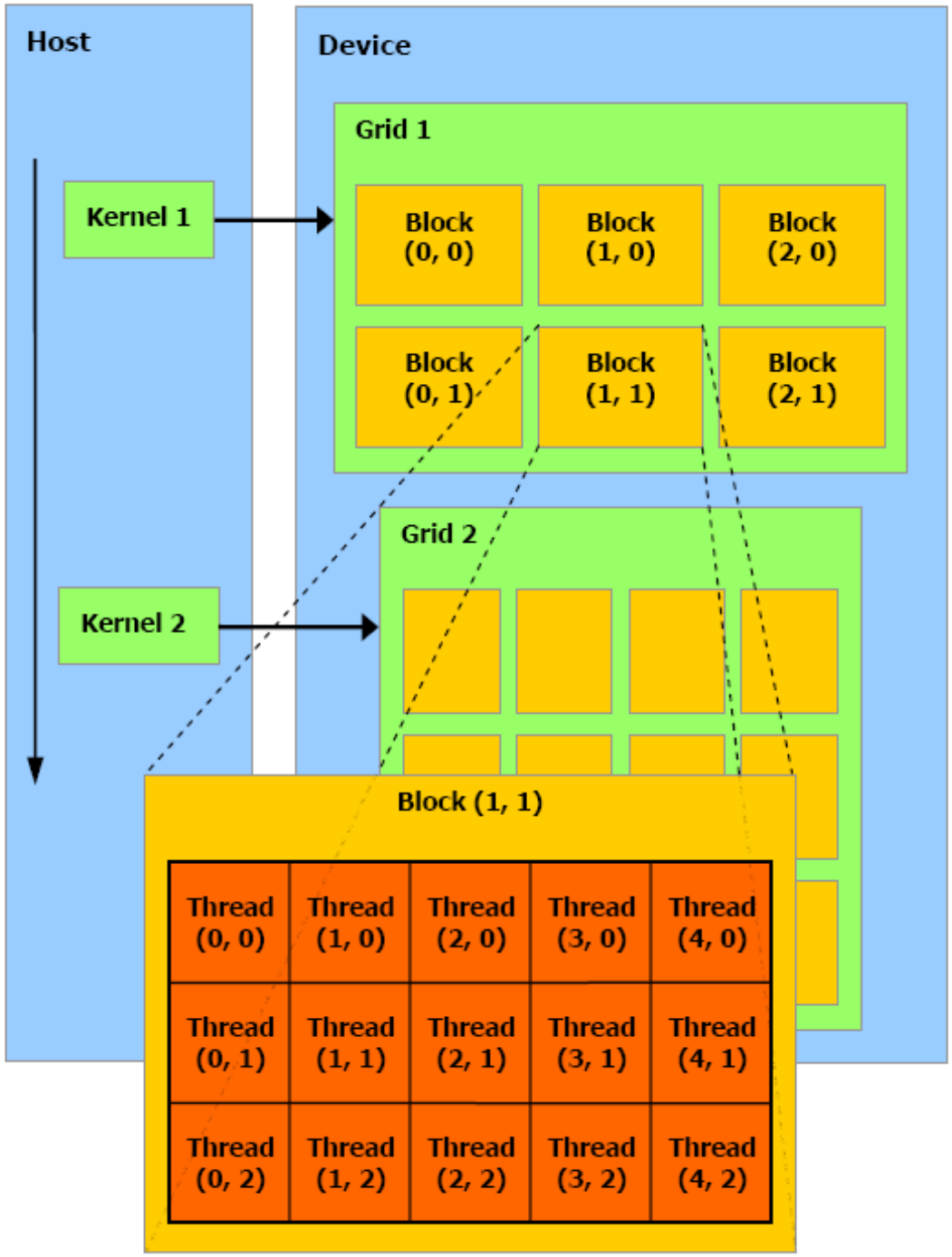
Блоки (block) и сетки (grid)

CPU - **хост/host;**

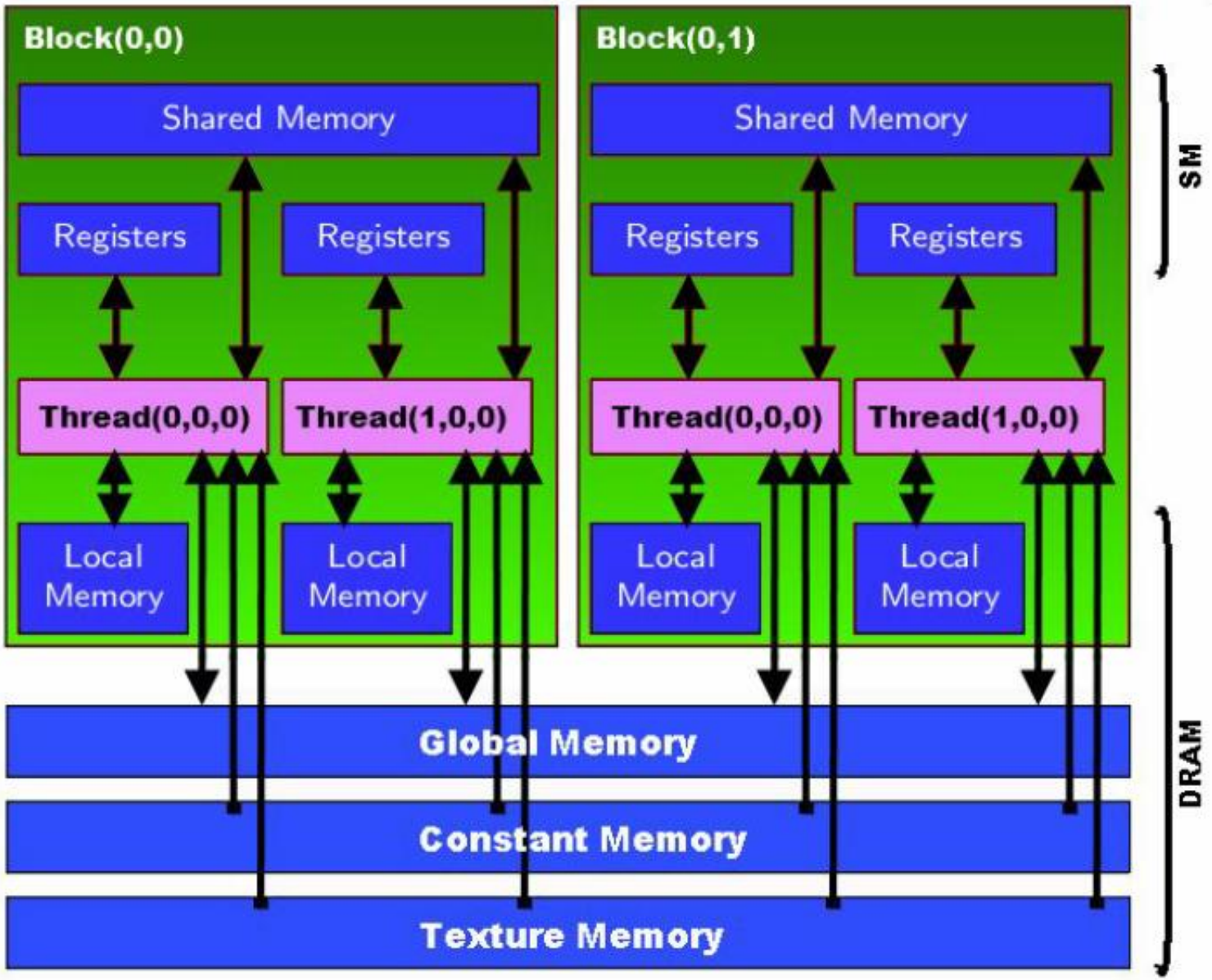
GPU - **устройство/device.**

Функция вызываемая на CPU и выполняемая на GPU – **ядро/kernel**.

Все потоки блока выполняются одним мультипроцессором.



Типы памяти



При доступе к DRAM возникают большие задержки.

Разбиение задачи по потокам

- Задача представляется в виде сетки блоков.
- Каждый блок – множество потоков, выполняемых на одном SM.
- Каждый SM может выполнять несколько блоков (если ресурсов недостаточно).
- Поток при выполнении знает координаты своего блока в сетке и свои координаты в блоке.

Различия версий CUDA

Технические спецификации	Версия CUDA						
	1.0	1.1	1.2	1.3	2.x	3.0	3.5
Максимальная размерность сетки блоков потоков	2				3		
Максимальные x-, y- или z- размерности сетки блоков	65535					2^{31}-1	
Максимальная размерность блока потоков	3						
Максимальные x- или y-размерности блока	512				1024		
Максимальная z-размерность блока	64						
Максимальное количество потоков в блоке	512				1024		
Размер warp’a (количество потоков)	32						
Максимальный объем разделяемой памяти на мультипроцессор	16 KB				48 KB		

Расширения языка C

- Директива запуска ядра.
- Новые типы данных.
- Встроенные переменные, доступные в ядре.
- Спецификаторы для функций и переменных.

Новые типы данных

• **1,2,3,4–мерные вектора** базовых типов.

Обращение по компонентам **x**, **y**, **z**, **w.**

Пример:

```
int2 a = make_int2(1, 7);
a.y = 7;
```

• **dim3** – тип, описывающий размерность сетки;

```
dim3 grid = dim3(10,1,1);
```

Встроенные переменные, доступные в ядре

- ✓`gridDim` – размерность сетки
- ✓`blockIdx` – координаты блока в сетке
- ✓`blockDim` – размерность блока
- ✓`threadIdx` – координаты потока в блоке
- ✓`warpSize` – размер warp-а.

Спецификаторы функций:

`__device__` - вызывается GPU, выполняется GPU;

`__global__` - вызывается CPU, выполняется GPU;

`__host__` - вызывается CPU, выполняется CPU.

На GPU не поддерживаются:

- взятие адреса;
- рекурсия;
- **static** переменные
- переменное число аргументов.

Спецификаторы переменных:

`__device__` - для размещения в глобальной памяти;

`__constant__` - для размещения в константной памяти;

`__shared__` - для размещения в разделяемой памяти.

Ограничения:

- не используются как поля структур;
- нельзя объявлять как `extern;`
- `__shared__` не могут инициализироваться при объявлении;
- запись в `__constant__` может осуществляться только CPU.

Директива запуска ядра

```
kernelName <<<bl, th, ns = 0, st = 0>>> (args);
```

dim3 bl – число блоков в сетке;

dim3 th – число нитей в блоке;

size_t ns – количество дополнительной разделяемой памяти, выделяемое блоку;

cudaStream_t st – поток, в котором нужно запустить ядро.

Работа с памятью

- выделение памяти на GPU:

```
cudaMalloc(&addr, size);
```

- копирование данных:

```
cudaMemcpy(&addr, &addr, size, direction);
```

- освобождение памяти на GPU:

```
cudaFree(&addr);
```

Синхронизация CPU - GPU

Многие функции являются асинхронными, в частности:
-запуск ядра;
-функции копирования памяти, имена которых оканчиваются на Async;
-функции копирования памяти device-device;
-функции инициализации памяти.

`cudaError_t cudaThreadSynchronize()` - дожидается завершения выполнения на GPU всех переданных запросов.

Синхронизация GPU - GPU

Барьер (только в пределах блока): `__syncthreads();`

Для отслеживания выполнения кода на GPU и синхронизации с CPU можно использовать объекты `cudaEvent_t`

```
cudaError_t cudaEventRecord      (cudaEvent_t event,
                                  CUstream stream);

cudaError_t cudaEventQuery       (cudaEvent_t event);


cudaError_t cudaEventElapsedTime ( float * time,
                                   cudaEvent_t startEvent,
                                   cudaEvent_t stopEvent );

cudaError_t cudaEventSynchronize (cudaEvent_t event);
```

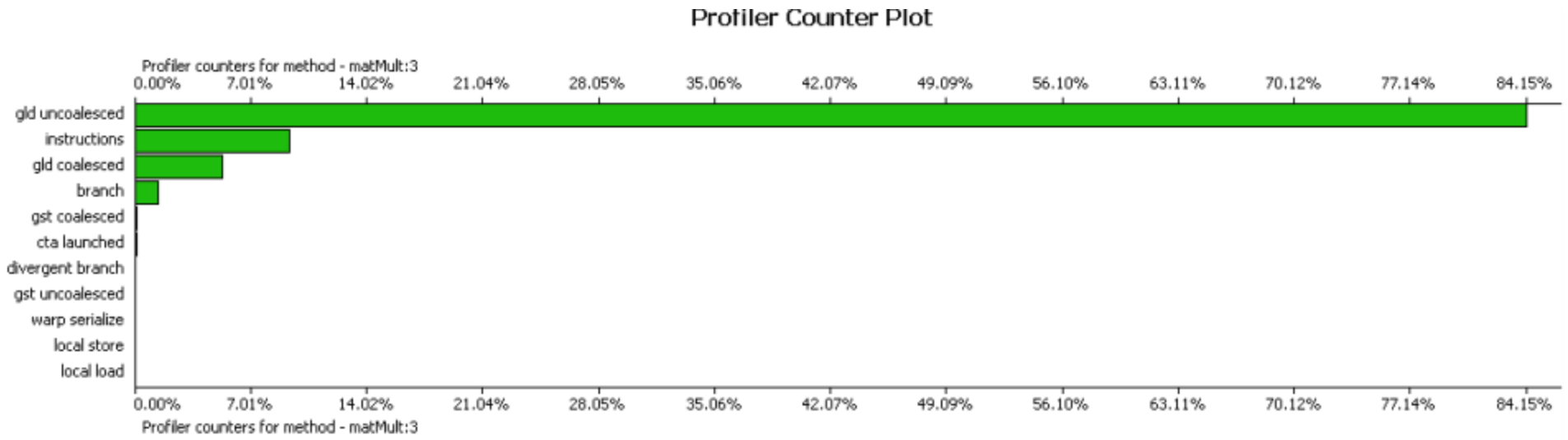
Пример. Умножение матриц:

```
int          numBytes = N * N * sizeof ( float );
float      * adev, * bdev, * cdev ;
dim3         threads ( BLOCK_SIZE, BLOCK_SIZE );
dim3         blocks ( N / threads.x, N / threads.y);
cudaMalloc       ( (void**)&adev, numBytes ); // allocate DRAM
cudaMalloc       ( (void**)&bdev, numBytes ); // allocate DRAM
cudaMalloc       ( (void**)&cdev, numBytes ); // allocate DRAM
                          // copy from CPU to DRAM
cudaMemcpy ( adev, a, numBytes, cudaMemcpyHostToDevice );
cudaMemcpy ( bdev, b, numBytes, cudaMemcpyHostToDevice );
matMult<<<blocks, threads>>> ( adev, bdev, N, cdev );
cudaThreadSynchronize();
cudaMemcpy             ( c, cdev, numBytes, cudaMemcpyDeviceToHost );
                           // free GPU memory
cudaFree               ( adev );
cudaFree               ( bdev );
cudaFree               ( cdev );
```

```
#define BLOCK_SIZE 16
__global__ void matMult ( float * a, float * b, int n, float * c )
{
  int   bx  = blockIdx.x;
  int   by  = blockIdx.y;
  int   tx  = threadIdx.x;
  int   ty  = threadIdx.y;
  float sum = 0.0f;
  int   ia  = n * BLOCK_SIZE * by + n * ty;
  int   ib  = BLOCK_SIZE * bx + tx;
  int   ic  = n * BLOCK_SIZE * by + BLOCK_SIZE * bx;

  for ( int k = 0; k < n; k++ )
    sum += a [ia + k] * b [ib + k*n];

  c [ic + n * ty + tx] = sum;
}
```

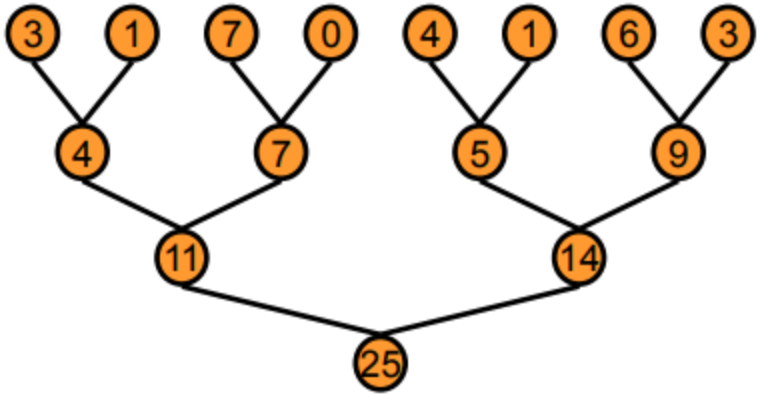


- Основное время (84.15%) ушло на чтение из глобальной памяти

Редукция

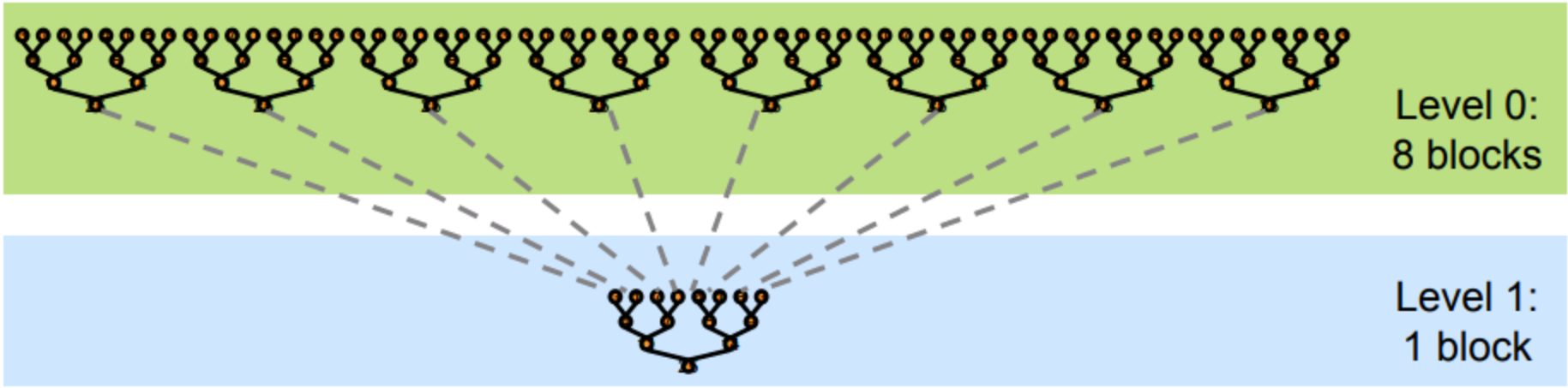
Parallel Reduction

- Tree-based approach used within each thread block

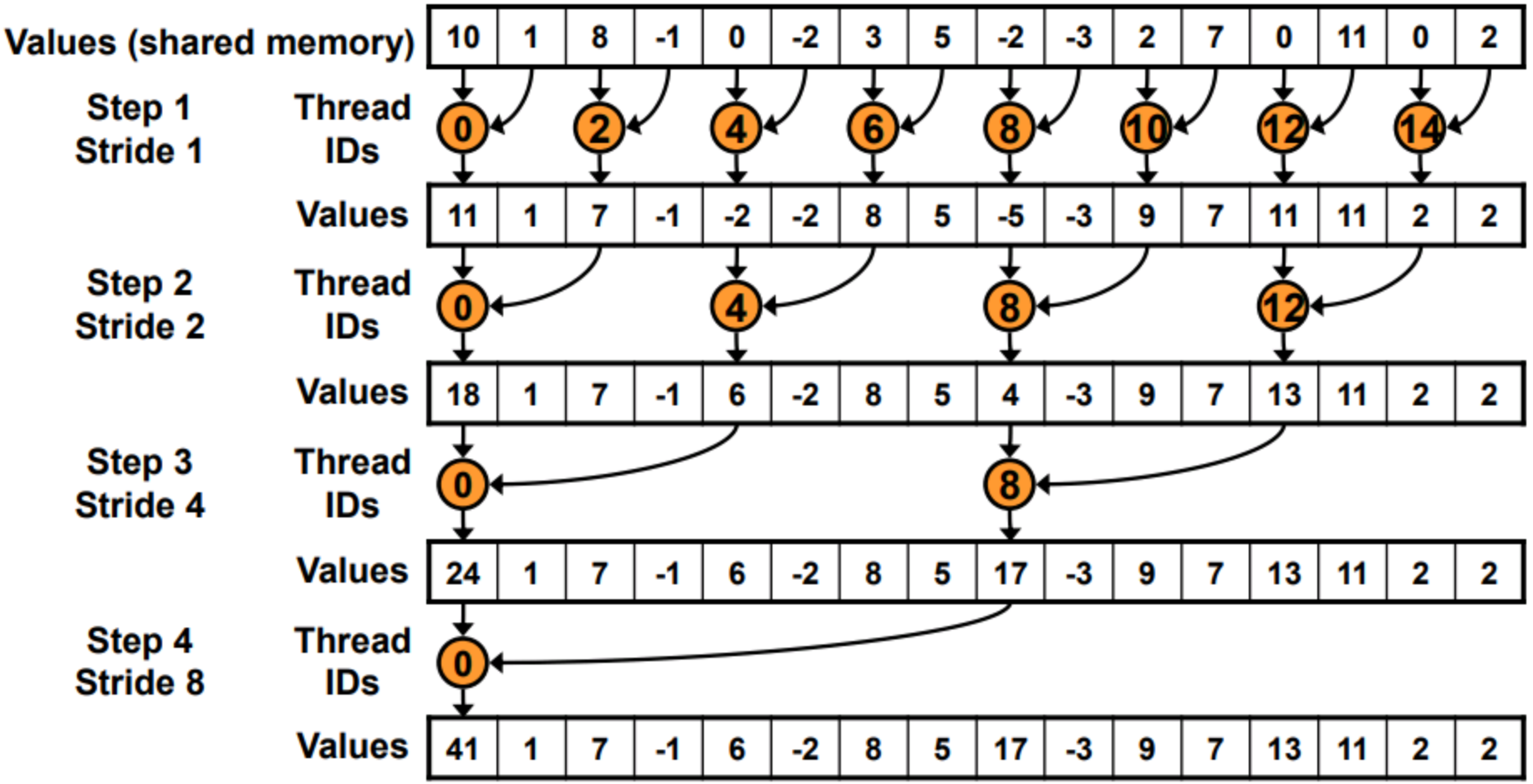


Solution: Kernel Decomposition

- **Avoid global sync by decomposing computation into multiple kernel invocations**



Parallel Reduction: Interleaved Addressing



Отличительные особенности работы с GPU

- Низкий уровень абстракции.
- Слабые возможности синхронизации.
- Ориентированность на массивно-параллельные программы.
- Хорошая масштабируемость.