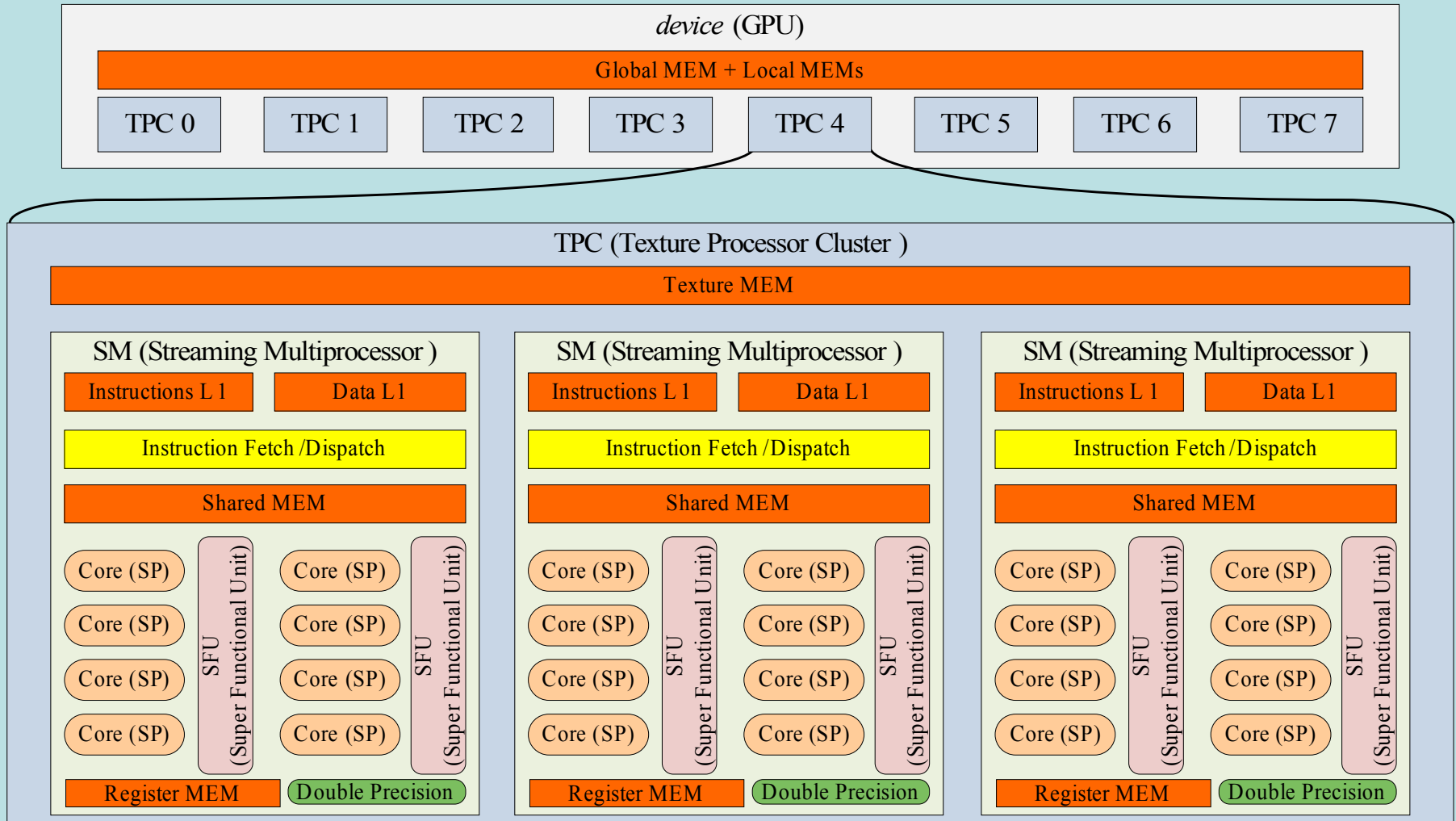
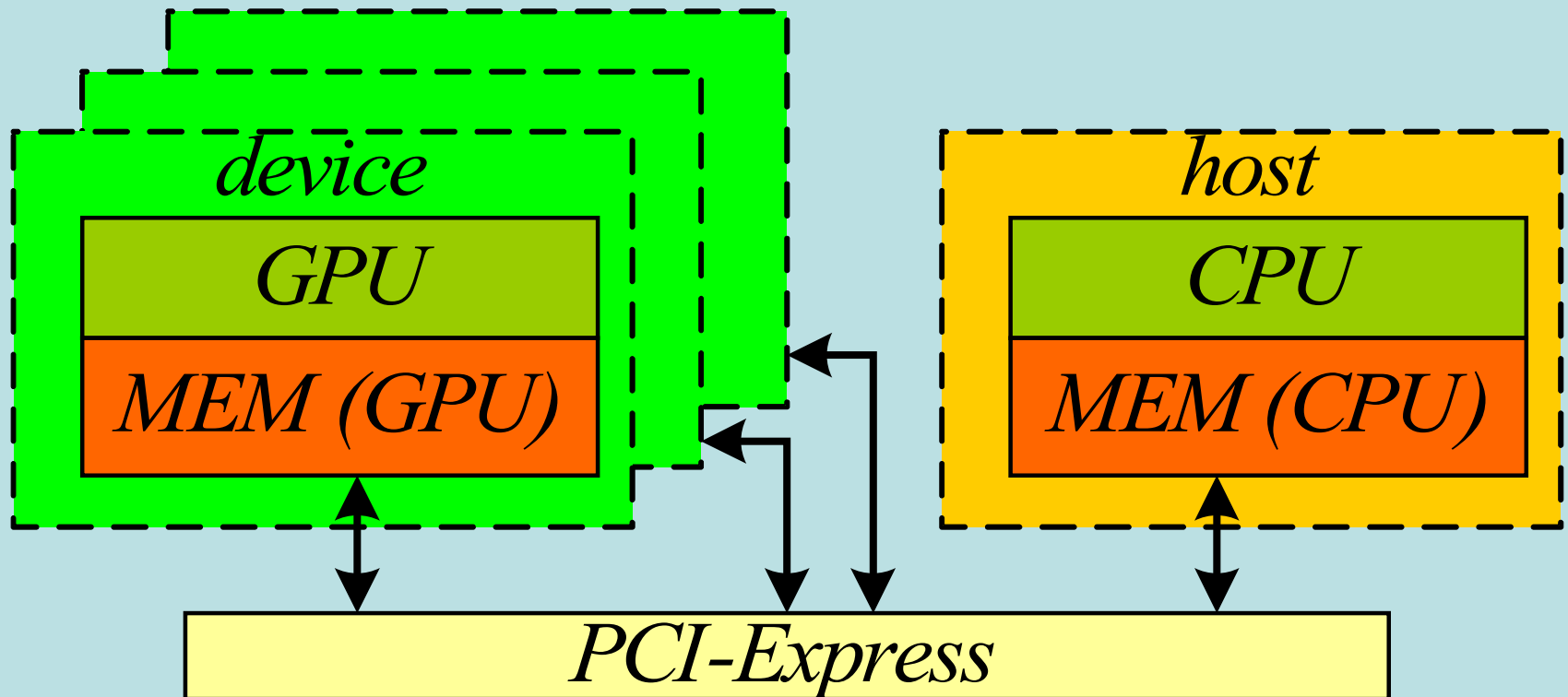


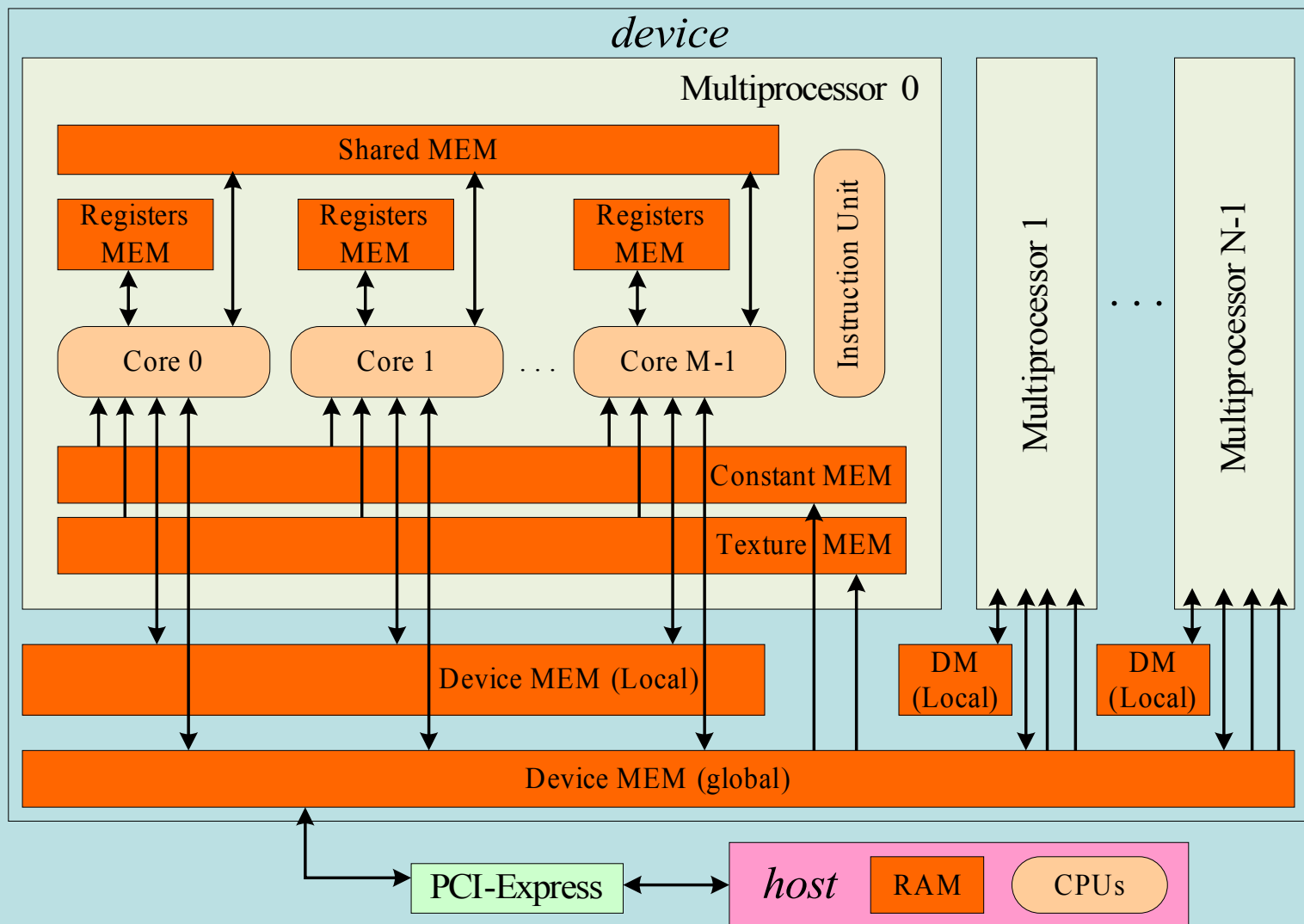
Структура GPU NVIDIA



device-s и host



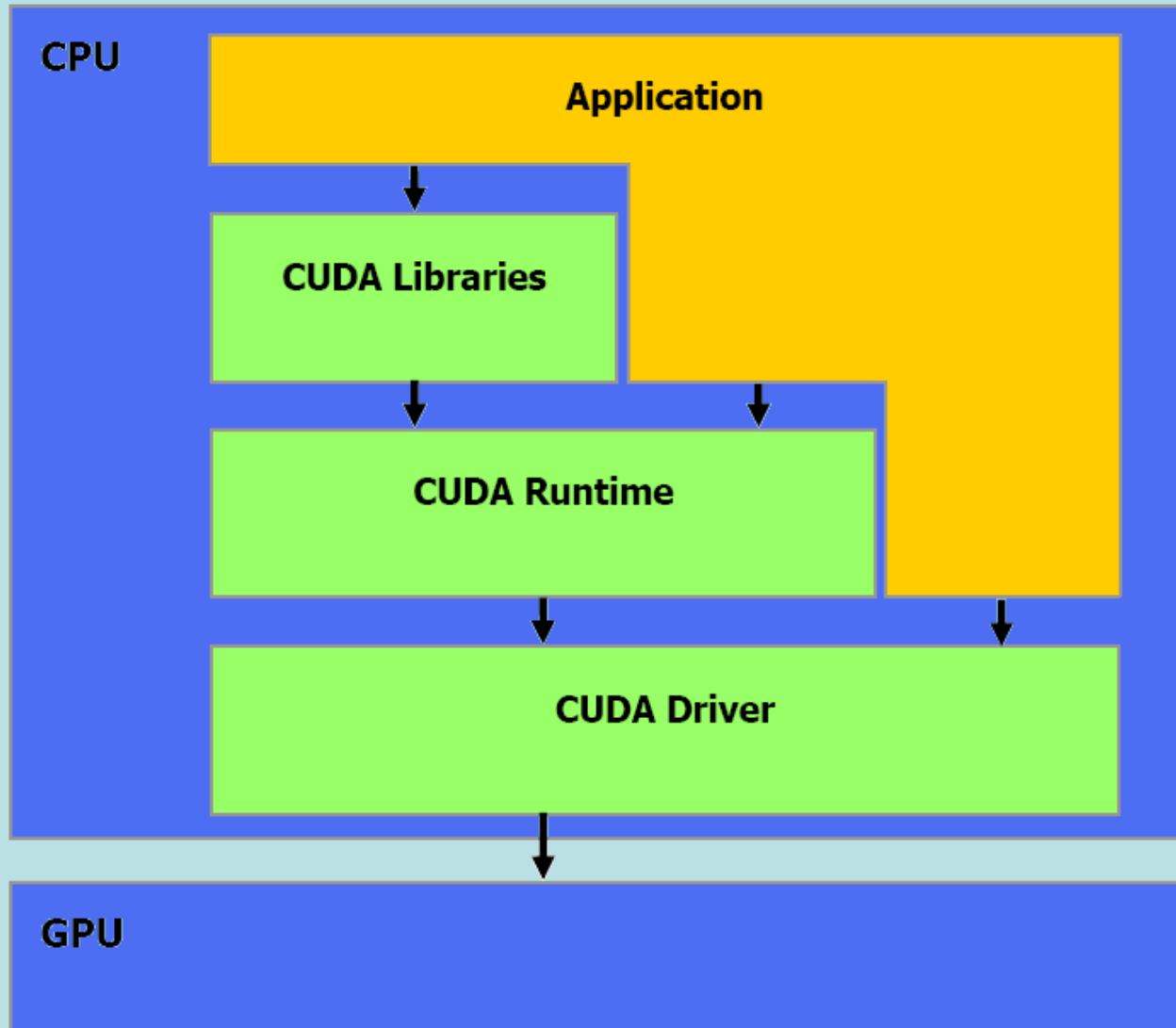
Доступ к памяти в NVIDIA GPU (CUDA)



Типы памяти NVIDIA GPU (CUDA)

Тип памяти	Доступ	Уровень выделения	Скорость работы
Регистры	R/W	Per-thread	Высокая(on-chip)
Локальная	R/W	Per-thread	Низкая (DRAM)
Shared	R/W	Per-block	Высокая(on-chip)
Глобальная	R/W	Per-grid	Низкая (DRAM)
Constant	R/O	Per-grid	Высокая(L1 cache)
Texture	R/O	Per-grid	Высокая(L1 cache)

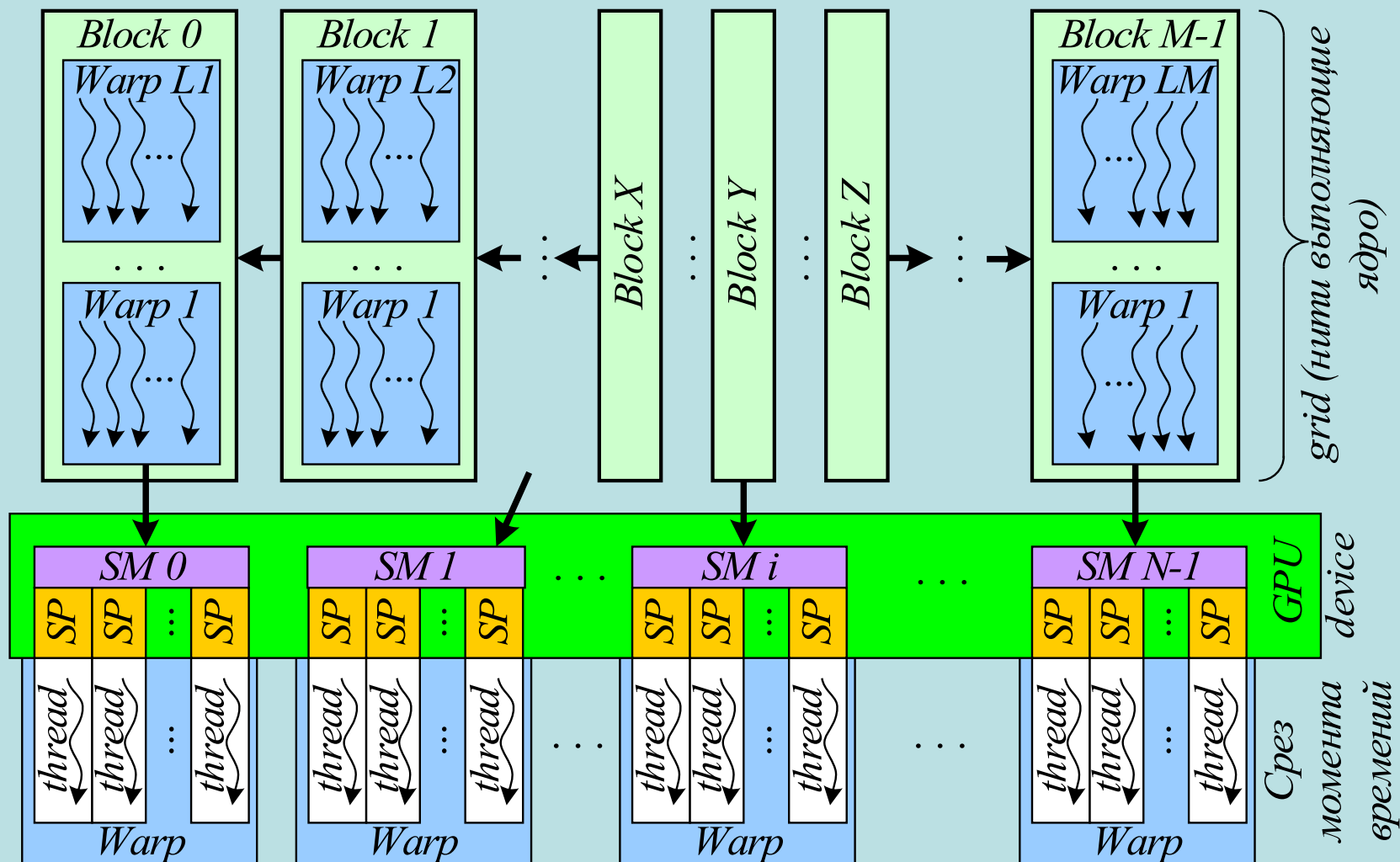
Идеология CUDA



Иерархия потоков CUDA

- 1) Параллельная часть кода выполняется как большое количество потоков (*threads*)
- 2) Потоки группируются в блоки (*blocks*) фиксированного размера (*blockDim*)
- 3) Блоки объединяются в сеть блоков (*grid*)
- 4) Ядро (*kernel*) выполняется на сетке из блоков
- 5) Каждый поток и блок имеют свой уникальный идентификатор (*threadIdx* и *blockIdx*)

Выполнение ядра на GPU



Структура CUDA программы

```
#include<stdio.h>

__global__ void FUN_KERNEL (формальные параметры функции-ядра)
{
    //Тело функции-ядра, Которое будет по умолчанию выполняться всеми запущенными
    потоками на GPU
}

int main ( int argc, char * argv [] )
{
    ПОДГОТОВКА ДАННЫХ НА host

    cudaMalloc ( указатель памяти device, размер);

    cudaMemcpy ( указатель памяти device, указатель памяти host, размер данных,
    cudaMemcpyHostToDevice);

    // ЗАПУСК ЯДРА НА device
    FUN_KERNEL<<< параметры исполнения >>> (параметры функции-ядра);

    cudaMemcpy ( указатель памяти CPU, указатель памяти GPU, размер данных,
    cudaMemcpyDeviceToHost );
    ИСПОЛЬЗОВАНИЕ РЕЗУЛЬТАТОВ РАБОТЫ device cudaFree (указатель памяти device);
    return 0;
}
```


Функция-ядро

<u>__global__</u>	<u>void</u>	<u>FUN_KERNEL</u>	<i>(параметры)</i>
префикс	всегда void	имя функции-ядра	Формальные параметры функции-ядра

{

//Тело функции-ядра

<внутренние переменные ядра>

<действия, выполняемые всеми потоками *grid*>

}

Функция main

```
int main ( int argc, char * argv [] )
{
    ...
    //Выделение памяти на GPU
    cudaMalloc ( указатель памяти device, размер);

    // Копирование данных в память device из памяти host
    cudaMemcpy ( указатель памяти device, указатель памяти host, размер данных,
                cudaMemcpyHostToDevice );
        Направление копирования
    // ЗАПУСК ЯДРА НА device
    FUN_KERNEL<<< параметры исполнения >>> (параметры функции-ядра);
    имя функции-ядра           параметры grid-а           фактические параметры

    //Копирование данных в память host из памяти device
    cudaMemcpy ( указатель памяти CPU, указатель памяти GPU, размер данных,
                cudaMemcpyDeviceToHost );

    ....
    cudaFree (указатель памяти device); //Очистка памяти на устройстве
    return 0;
}
```

Общий вид команды для запуска ядра

имя ядра <<<*bl, th, ns, st*>>> (*data*);

имя функции-ядра

параметры *grid*-а

фактические
параметры

bl – число блоков в *grid*

th – число потоков в блоке

ns – количество дополнительной
shared- памяти, выделяемое блоку

st – поток, в котором нужно
запустить ядро

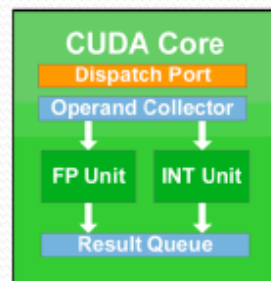
1.2 Развитие архитектур с CUDA.

Compute Capability

- Поколение **Tesla** (не путать с линией продуктов для HPC)
 - 1.1 – базовые возможности CUDA, атомарные операции с глобальной памятью
 - 1.2 - атомарные операции с общей памятью, warp vote-функции
 - 1.3 – вычисления с двойной точностью
- Поколение **Fermi**
 - 2.0 - новая архитектура чипа, ECC, кеши L1 и L2, асинхронное выполнение ядер, UVA и др.
 - 2.1 - новая архитектура warp scheduler-ов
- Поколение **Kepler**
 - 3.0, 3.2 – Новая архитектура чипа, Unified memory programming, warp shfl и др.
 - 3.0 - Динамический параллелизм, Hyper Queue и др.
- Поколение **Maxwell**
 - sm_50 and sm_52 – Новая архитектура чипа
-

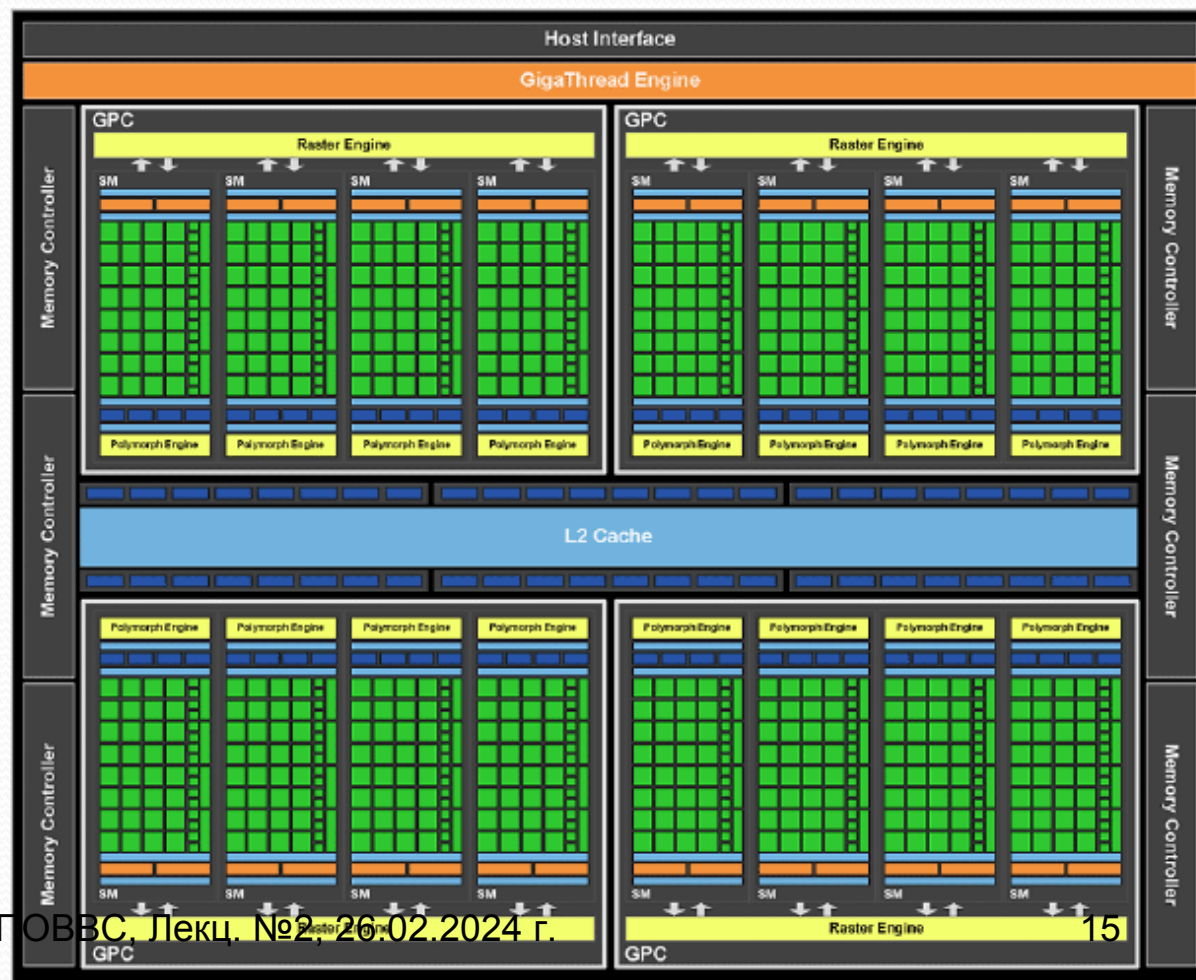
Fermi: Streaming Multiprocessor (SM)

- Поточковый мультипроцессор
- «Единица» построения устройства (как ядро в CPU):
 - 32 скалярных ядра CUDA Core, ~1.5ГГц
 - 2 Warp Scheduler-а
 - Файл регистров, 128KB
 - 3 Кэша – текстурный, глобальный (L1), константный(uniform)
 - PolyMorphEngine – графический конвейер
 - Текстуры юниты
 - 16 x Special Function Unit (SFU) – интерполяция и трансцендентная математика одинарной точности
 - 16 x Load/Store



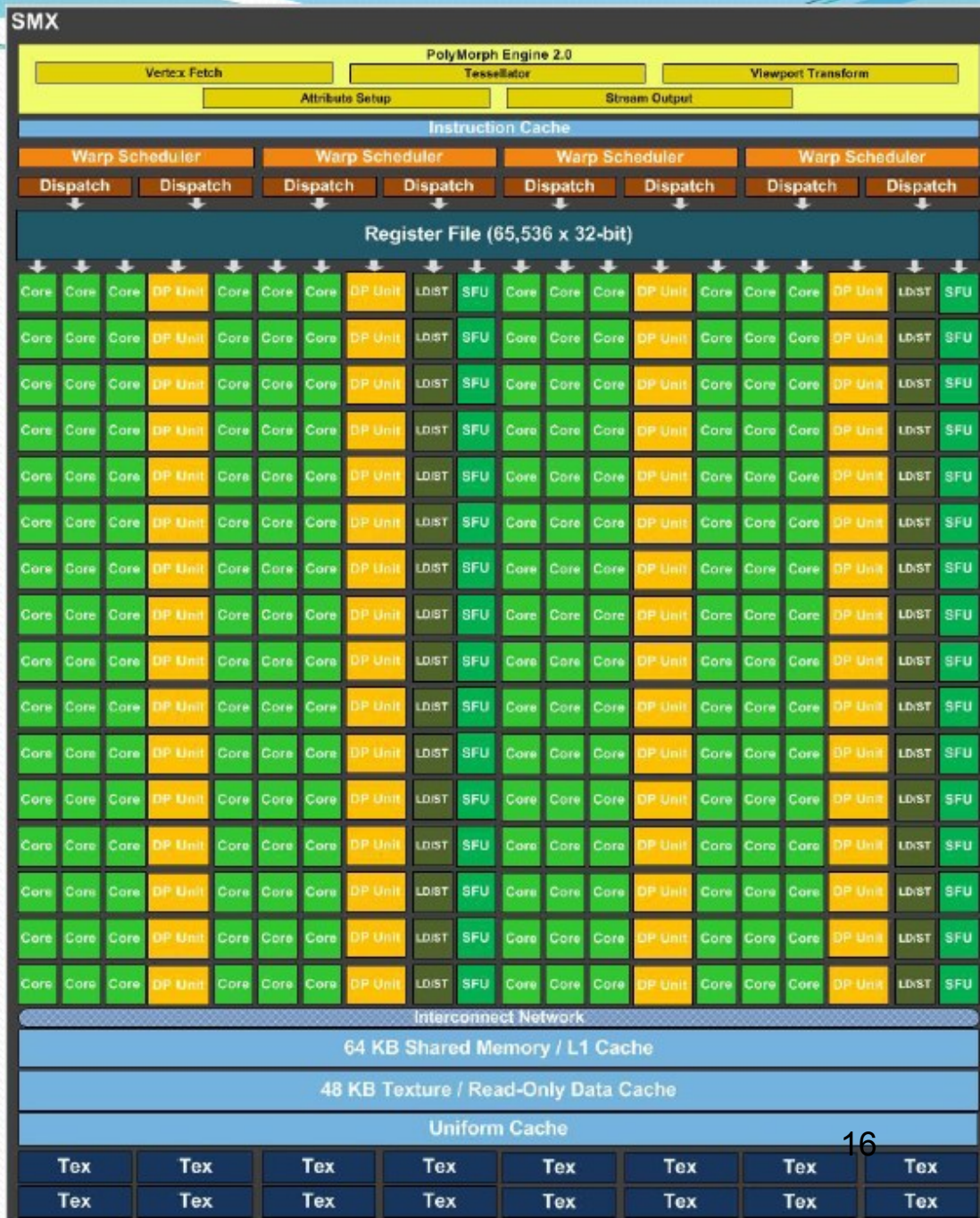
Fermi: Чип в максимальной конфигурации

- 16 SM
- 512 ядер CUDA Core
- Кеш L2 758KB
- GigaThreadEngine
- Контроллеры памяти DDR5
- Интерфейс PCI



Kepler: SMX

- 192 cuda core
- 64 x DP Unit
- 32 x SFU
- 32x load/store Unit
- 4 x warp scheduler
- 256KB регистров



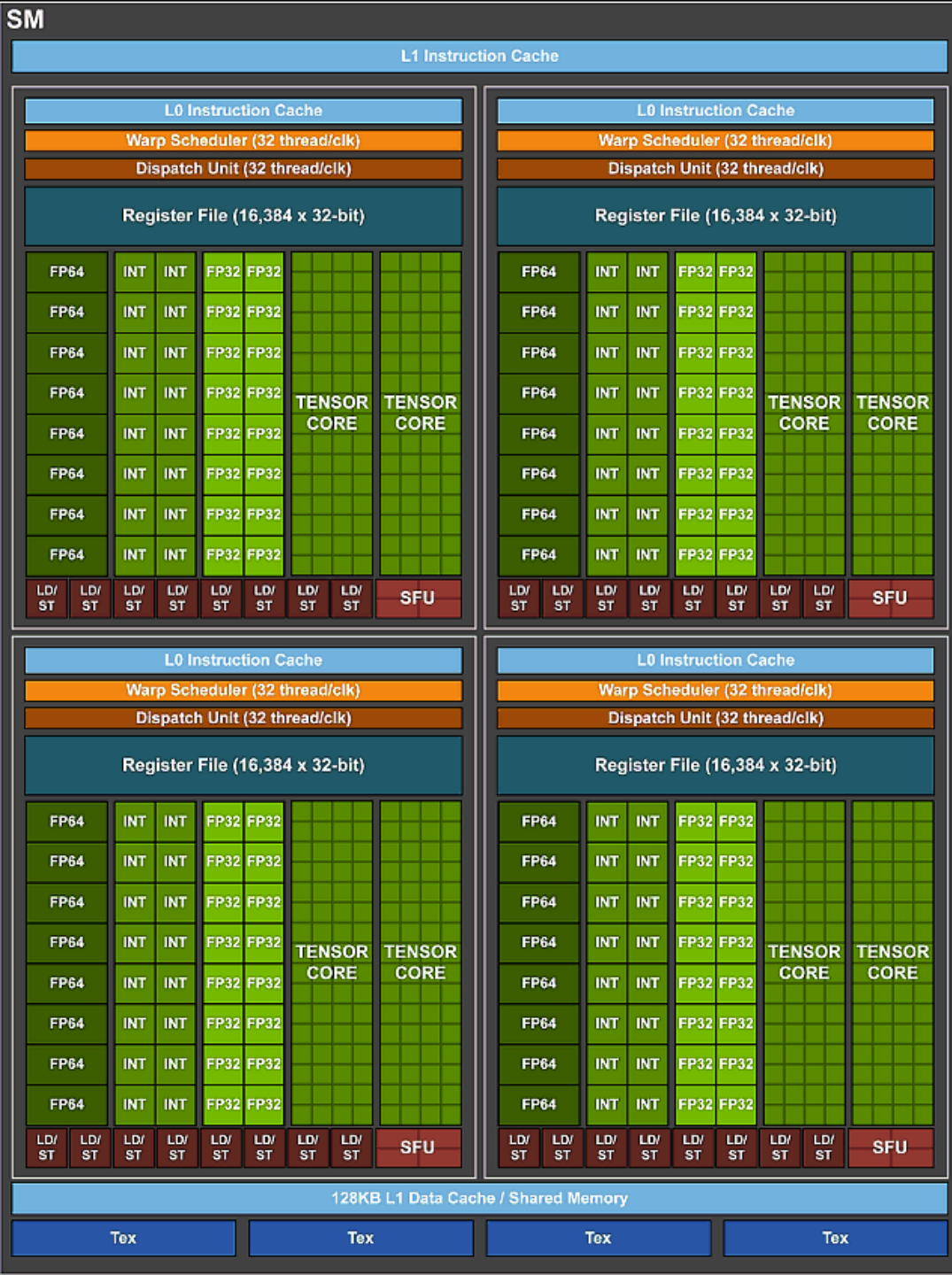
Kepler: Чип в максимальной конфигурации

- 15 SXM = 2880 cuda core



Nvidia Tesla V100 (Volta)





SM Nvidia Tesla V100 (Volta)

Параметры системы V100

Модель GPU	GV100
Архитектура	Volta
Кол-во SM	80
Кол-во TPC	40
Ядер FP32 на SM	64
Ядер FP32 всего	5120
Ядер FP64 на SM	32
Ядер FP64 всего	2560
Тензорные ядра всего	640
Турбо-частота GPU, МГц	1455
Пиковая пр-сть FP32, терафлопс	15,0
Пиковая пр-сть FP64, терафлопс	7,5
Пиковая пр-сть тензор, терафлопс	120
Кол-во TMU	320
Шина памяти, бит	4096
Тип памяти	HBM2
Объем памяти, ГБ	16 ГБ
Объем L2-кэша, КБ	6144
Объем разделяемой памяти на SM, КБ	До 96 КБ
Объем регистрового файла, КБ	20480

1.3. Выполнение и взаимодействие потоков ядра в CUDA

Сетка потоков (Grid)

При запуске ядра создаётся следующая иерархия совокупности запущенных потоков:

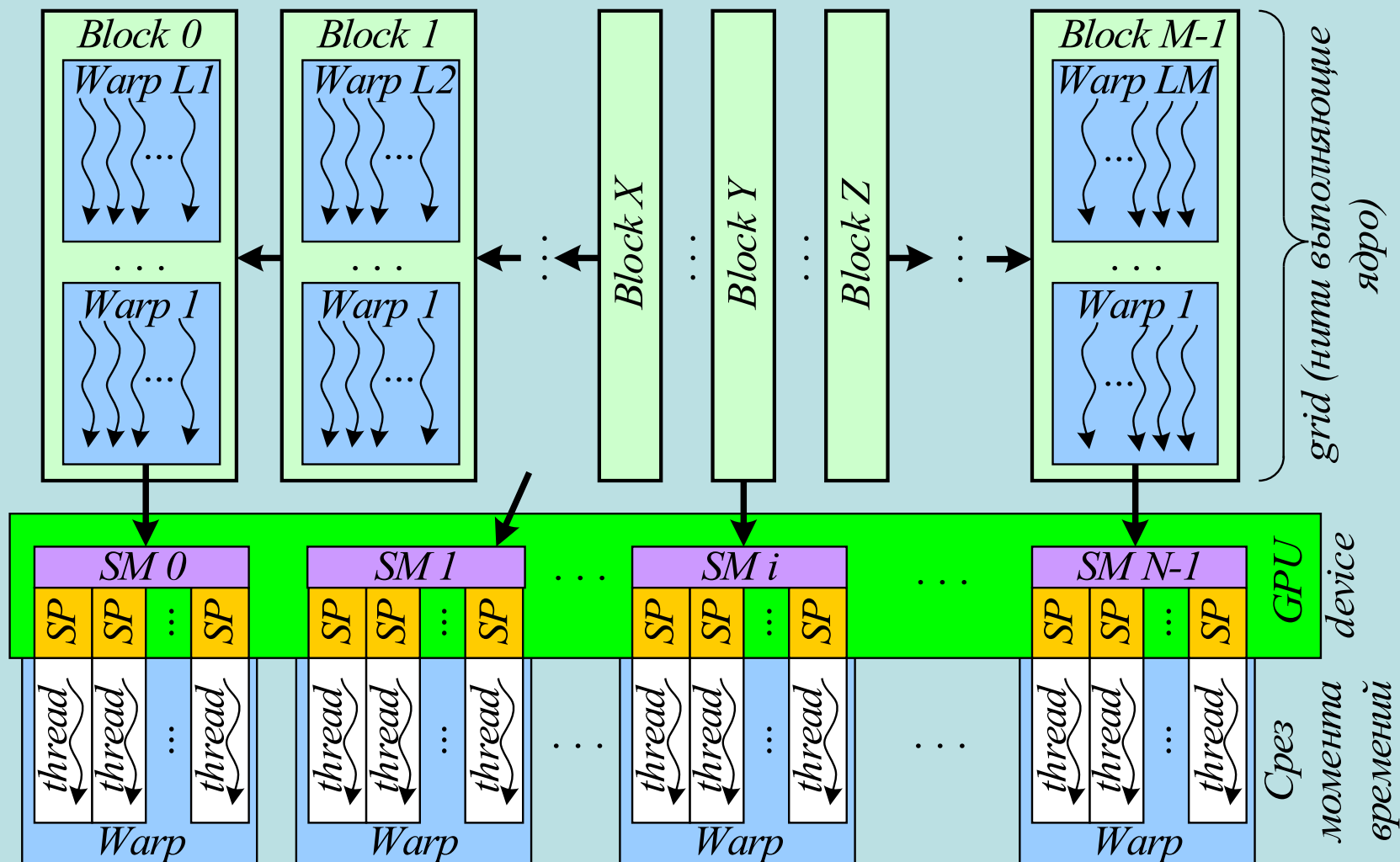
- **Grid** (грид) – все потоки сгруппированные по блокам;
- **Block** (блок) – группа потоков в рамках grid
- **Warp** (варп) – совокупность потоков одного блока, выполняемые одновременно;
- **Thread** (поток) – отдельный поток.

Сетка потоков (Grid) классика

Grid – все потоки функции-ядра

Block 0,0			Block 1,0			Block 2,0		
Thread 0,0,0	Thread 1,0,0	Thread 2,0,0	Thread 0,0,0	Thread 1,0,0	Thread 2,0,0	Thread 0,0,0	Thread 1,0,0	Thread 2,0,0
Thread 0,1,0	Thread 1,1,0	Thread 2,1,0	Thread 0,1,0	Thread 1,1,0	Thread 2,1,0	Thread 0,1,0	Thread 1,1,0	Thread 2,1,0
Thread 0,2,0	Thread 1,2,0	Thread 2,2,0	Thread 0,2,0	Thread 1,2,0	Thread 2,2,0	Thread 0,2,0	Thread 1,2,0	Thread 2,2,0
Block 0,1			Block 1,1			Block 2,1		
Thread 0,0,0	Thread 1,0,0	Thread 2,0,0	Thread 0,0,0	Thread 1,0,0	Thread 2,0,0	Thread 0,0,0	Thread 1,0,0	Thread 2,0,0
Thread 0,1,0	Thread 1,1,0	Thread 2,1,0	Thread 0,1,0	Thread 1,1,0	Thread 2,1,0	Thread 0,1,0	Thread 1,1,0	Thread 2,1,0
Thread 0,2,0	Thread 1,2,0	Thread 2,2,0	Thread 0,2,0	Thread 1,2,0	Thread 2,2,0	Thread 0,2,0	Thread 1,2,0	Thread 2,2,0

Выполнение ядра на GPU



Функция-ядро

<u>__global__</u>	<u>void</u>	<u>FUN_KERNEL</u>	<u>(параметры)</u>
префикс	всегда void	имя функции-ядра	Формальные параметры функции-ядра

{

//Тело функции-ядра

<внутренние переменные ядра>

<действия, выполняемые всеми потоками *grid*>

}

Функция main

```
int main ( int argc, char * argv [] )
{
    ...
    //Выделение памяти на GPU
    cudaMalloc ( указатель памяти device, размер);

    // Копирование данных в память device из памяти host
    cudaMemcpy ( указатель памяти device, указатель памяти host, размер данных,
                cudaMemcpyHostToDevice );
        Направление копирования
    // ЗАПУСК ЯДРА НА device
    FUN_KERNEL<<< параметры исполнения >>> (параметры функции-ядра);
    имя функции-ядра           параметры grid-а           фактические параметры

    //Копирование данных в память host из памяти device
    cudaMemcpy ( указатель памяти CPU, указатель памяти GPU, размер данных,
                cudaMemcpyDeviceToHost );

    ....
    cudaFree (указатель памяти device); //Очистка памяти на устройстве
    return 0;
}
```

Общий вид команды для запуска ядра

имя ядра <<<*bl, th, ns, st*>>> (*data*);

имя функции-ядра

параметры *grid*-а

фактические
параметры

bl – число блоков в *grid*

th – число потоков в блоке

ns – количество дополнительной
shared- памяти, выделяемое блоку

st – поток, в котором нужно
запустить ядро

Задача №1

Сложить поэлементно два вектора:
vect1 и *vect2*, длиной в *N* элементов
с плавающей точкой

Решение задачи №1

host


device

Задаём два вектора типа *float*


Выделяем память на

GPU

vect1 

devA 

vect2 

devB 

Получаем указатели

devA и *devB*

Копируем данные на *device*

vect1 



devA 

vect2 



devB 

Запускаем ядро и
передаём ему указатели

devA и *devB*

Ядро вычисляет

devA 

+

devB 

=

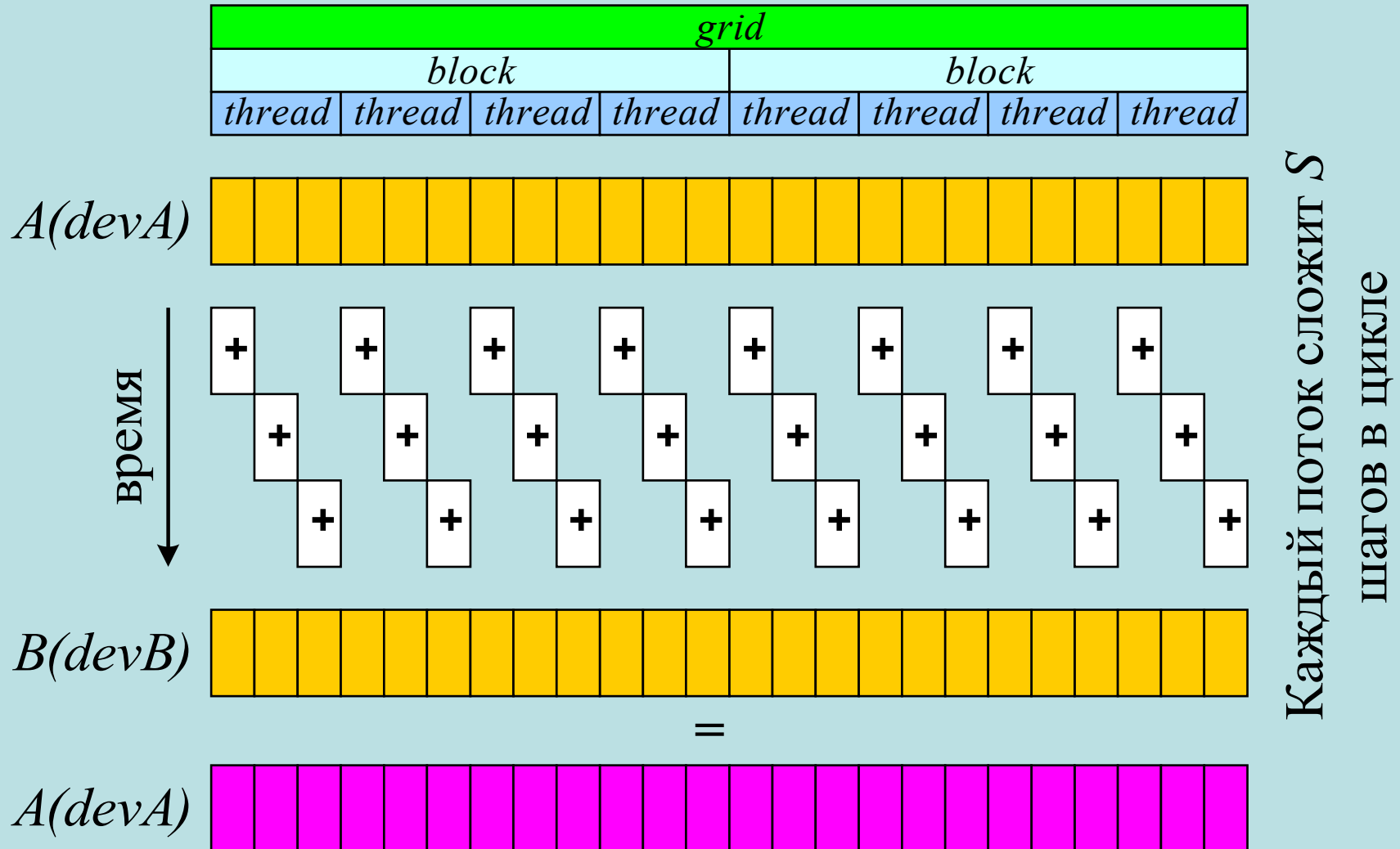
vect1 



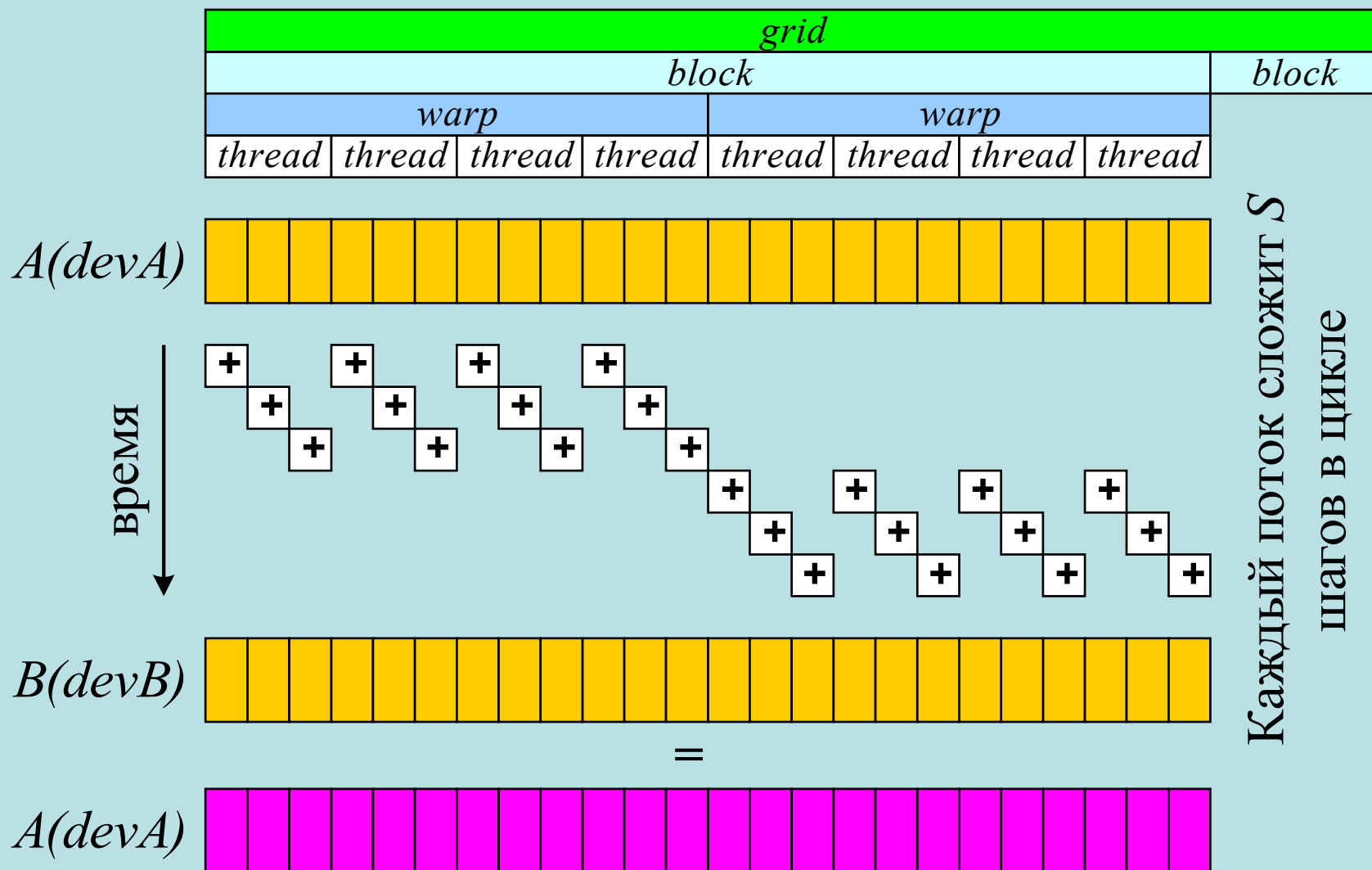
devA 

Копируем результат на *host*

Решение задачи Ядром



Выполнение блока варпами



Текст ядра

```
__global__ void FUN_KERNEL ( float * A, float * B, int S)
{
    int k;
    int idx_thread = blockIdx.x * blockDim.x + threadIdx.x;
                    «сквозной»      номер блока      размер блока      номер потока в
                    номер потока                                блоке

    for (k=0;k<S;k++)
        A[idx_thread*S+k]=A[idx_thread*S+k]+B[idx_thread*S+k];
}
```

Подготовка в вызову ядра

```
#include<stdio.h>
```

```
#define  N 4096
```

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

```
    int i,blocks,blocksize,steps;
```

```
    float vect1[N],vect2[N];
```

```
    FILE *f;
```

```
    float * devA, *devB;
```

```
    for (i=0; i<N; i++) { vect1[i]=i; vect2[i]=i;}
```

```
    cudaMalloc ( (void**)&devA, N * sizeof ( float ) ) ;
```

```
    cudaMalloc ( (void**)&devB, N * sizeof ( float ) ) ;
```

```
    blocks=4; blocksize=64; steps=(int)N/(blocks*blocksize);
```

```
    cudaMemcpy ( devA, vect1, N * sizeof ( float ), cudaMemcpyHostToDevice);
```

```
    cudaMemcpy ( devB, vect2, N*sizeof ( float ), cudaMemcpyHostToDevice);
```


Вызов ядра и завершение

//Вызов ядра с именем *FUN_KERNEL*

```
FUN_KERNEL<<<blocks,blocksize>>> ( devA, devB, steps);
```

```
cudaMemcpy ( vect1, devA, N * sizeof ( float ), cudaMemcpyDeviceToHost );
```

```
cudaFree ( devA );
```

```
for (i = 0; i < N; i++) printf("vect1[%d] = %.5f\n", i, vect1[i]);
```

```
return 0;
```

```
}
```

Сетка потоков (Grid) (повтор)

Grid – все потоки функции-ядра

Block 0,0			Block 1,0			Block 2,0		
Thread 0,0,0	Thread 1,0,0	Thread 2,0,0	Thread 0,0,0	Thread 1,0,0	Thread 2,0,0	Thread 0,0,0	Thread 1,0,0	Thread 2,0,0
Thread 0,1,0	Thread 1,1,0	Thread 2,1,0	Thread 0,1,0	Thread 1,1,0	Thread 2,1,0	Thread 0,1,0	Thread 1,1,0	Thread 2,1,0
Thread 0,2,0	Thread 1,2,0	Thread 2,2,0	Thread 0,2,0	Thread 1,2,0	Thread 2,2,0	Thread 0,2,0	Thread 1,2,0	Thread 2,2,0
Block 0,1			Block 1,1			Block 2,1		
Thread 0,0,0	Thread 1,0,0	Thread 2,0,0	Thread 0,0,0	Thread 1,0,0	Thread 2,0,0	Thread 0,0,0	Thread 1,0,0	Thread 2,0,0
Thread 0,1,0	Thread 1,1,0	Thread 2,1,0	Thread 0,1,0	Thread 1,1,0	Thread 2,1,0	Thread 0,1,0	Thread 1,1,0	Thread 2,1,0
Thread 0,2,0	Thread 1,2,0	Thread 2,2,0	Thread 0,2,0	Thread 1,2,0	Thread 2,2,0	Thread 0,2,0	Thread 1,2,0	Thread 2,2,0

Добавленные типы CUDA

1, 2, 3 и 4-х мерные вектора

- **char1, char2, char3, char4,**
- **uchar1, uchar2, uchar3, uchar4,**
- **short1, short2, short3, short4,**
- **ushort1, ushort2, ushort3, ushort4,**
- **int1, int2, int3, int4,**
- **uint1, uint2, uint3, uint4,**
- **long1, long2, long3, long4,**
- **ulong1, ulong2, ulong3, ulong4,**
- **float1, float2, float3, float2,**
- **double2.**

Обращение к компонентам вектора идет по именам - x, y, z и w.

Значения задаются так:

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

```
float3 u = make_float3 ( 1, 2, 3.4f );
```

Добавленные переменные

В язык C добавлены следующие специальные переменные

gridDim - размер *grid*'а (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*'а (имеет тип **int**)

Объявление Grid

```
dim3 blocks=dim3(3, 2, 5);
```

```
dim3 threads=dim3(3, 3, 1);
```

```
kernel <<blocks, threads>>(...);
```

Итого, в ядре будут следующие размеры:

```
gridDim.x = 3, gridDim.y = 2, gridDim.z = 5;
```

```
blockDim.x = 3, blockDim.y = 3;
```

Идентификация будет осуществляться переменными:

```
blockIdx.x, blockIdx.y и blockIdx.z
```

```
threadIdx.x, threadIdx.y и threadIdx.z
```

Идентификация потока (классика)

Определение номера потока в блоке:

```
idx_block=threadIdx.x+blockDim.x*threadIdx.y+  
          +blockDim.x*blockDim.y*threadIdx.z;
```

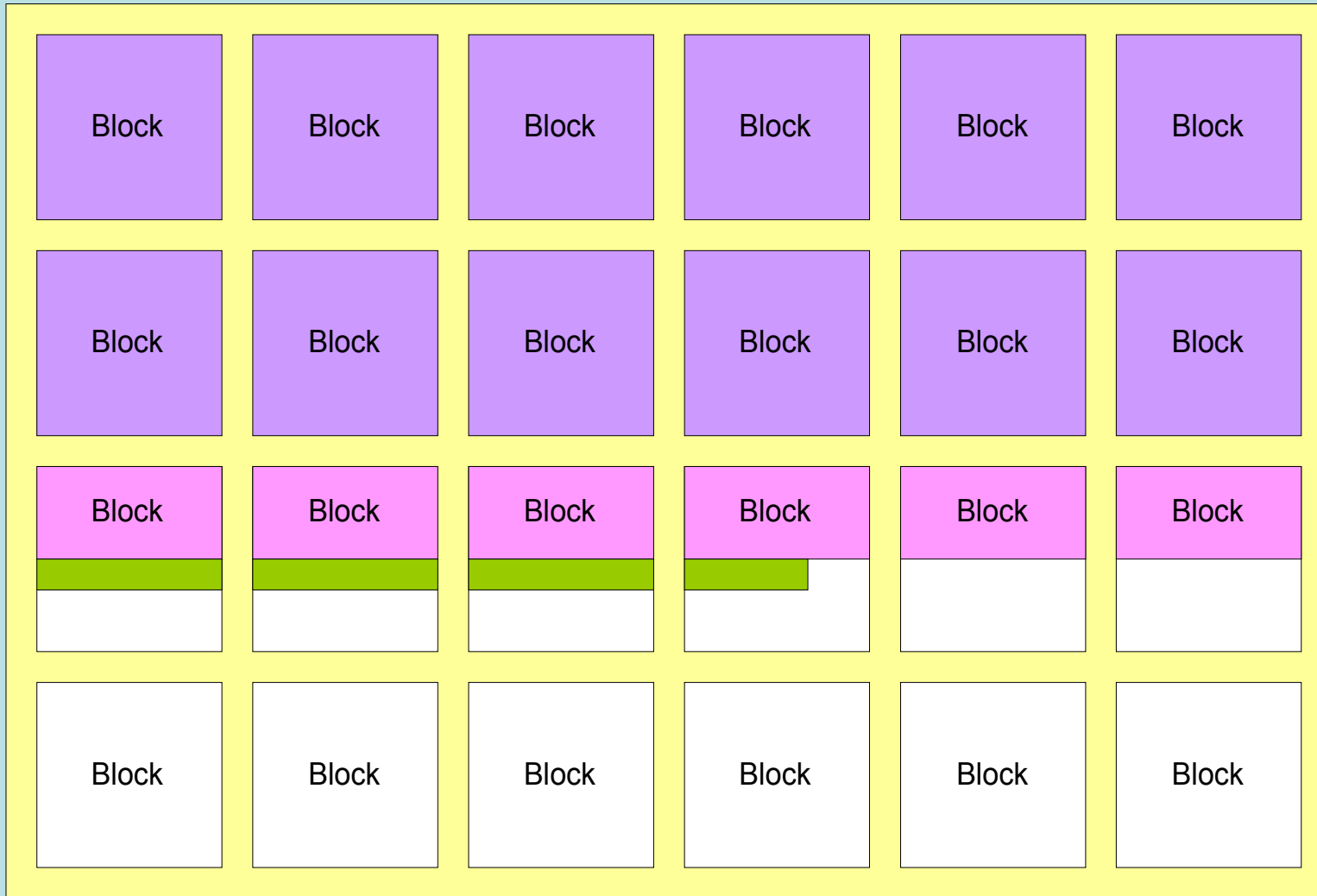
Определение номера потока в grid:

```
idx_grid=idx_block+(blockIdx.x+gridDim.x*blockIdx.y)*  
              *blockDim.x*blockDim.y*blockDim.z;
```

Двухкоординатное сквозное определение номера потока в grid:

```
idx_grid2=threadIdx.x+blockDim.x* blockIdx.x + +  
          (blockDim.x*gridDim.x)*threadIdx.y+ +  
          (blockDim.x*blockDim.y)*blockIdx.y;
```

Идентификация потока



III

$(\text{blockDim.x} * \text{blockDim.y}) * \text{blockIdx.y};$

II

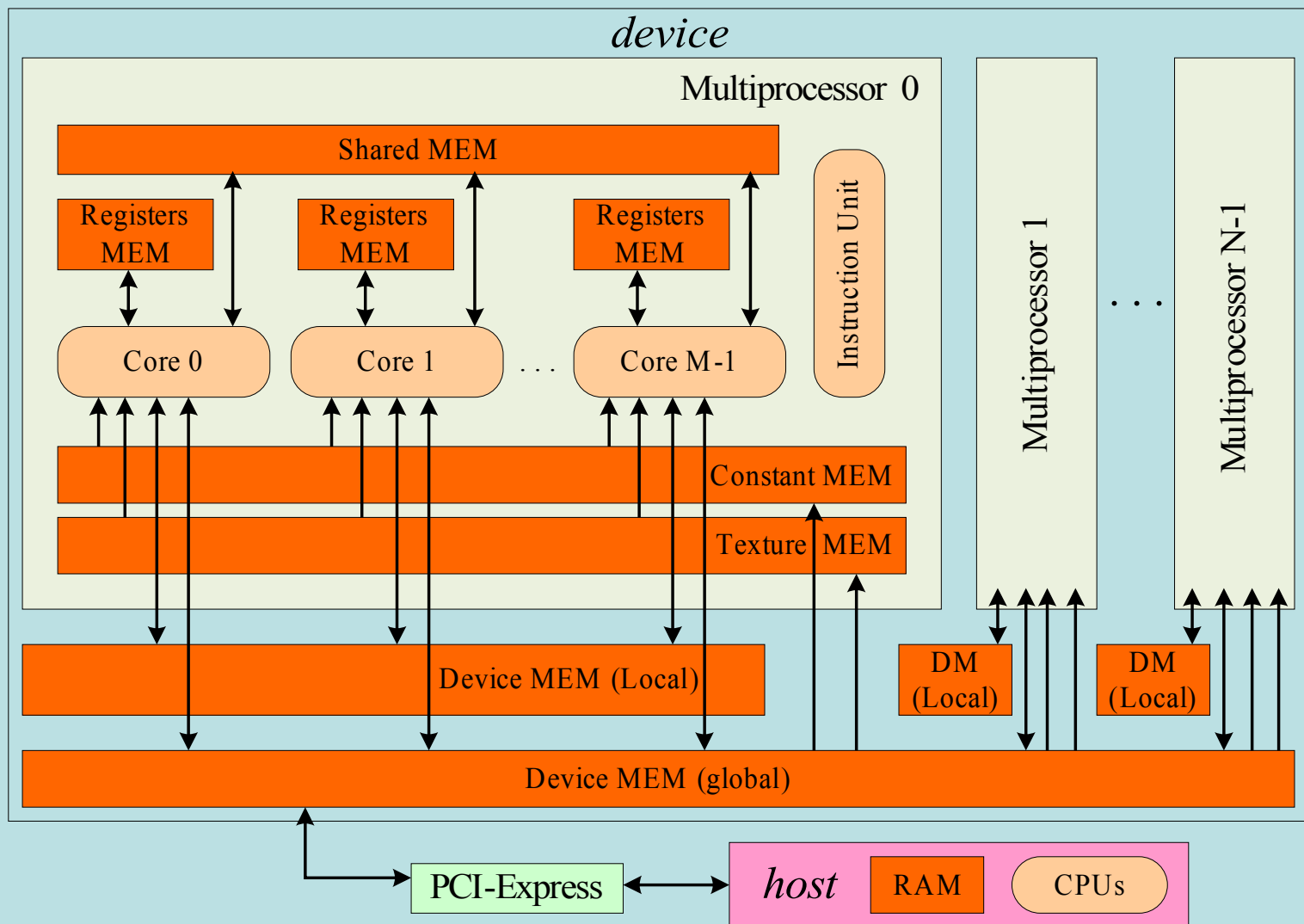
I

$\text{threadIdx.x} + \text{blockDim.x} * \text{blockIdx.x}$
 $(\text{blockDim.x} * \text{gridDim.x}) * \text{threadIdx.y}$

Функции в CUDA и их спецификаторы

	Спецификатор	Откуда вызывается	Где выполняется
Функция только CPU	<code>__host__</code>	<code>host</code>	<code>host (CPU)</code>
Функция ЯДРО GPU	<code>__device__</code>	<code>host</code>	<code>device (GPU)</code>
Функция только GPU	<code>__device__</code>	<code>device</code>	<code>device (GPU)</code>
Функция двойного назначения	<code>__host__ __device__</code>	<code>host</code> или <code>device</code>	<code>host</code> или <code>device</code>

Доступ к памяти в NVIDIA GPU (CUDA)



Типы памяти NVIDIA GPU (CUDA)

Тип памяти	Доступ	Уровень выделения	Скорость работы
Регистры	R/W	Per-thread	Высокая(on-chip)
Локальная	R/W	Per-thread	Низкая (DRAM)
Shared	R/W	Per-block	Высокая(on-chip)
Глобальная	R/W	Per-grid	Низкая (DRAM)
Constant	R/O	Per-grid	Высокая(L1 cache)
Texture	R/O	Per-grid	Высокая(L1 cache)

Назначение памяти

Регистровая память (register) является самой быстрой из всех видов. Она программно недоступна и, как правило, используется для хранения локальных переменных потока.

Локальная память (local memory) может быть использована компилятором при большом количестве локальных переменных потока в какой-либо функции выполняемой на device.

Глобальная память (global memory) – самый медленный тип памяти, из доступных GPU. Она является основной и самой ёмкой. Глобальные переменные можно выделить с помощью спецификатора `__global__`, а так же динамически, с помощью функций из семейства `cudaMallocXXX`.

Разделяемая память (shared memory) относится к быстрому типу памяти. Она является общедоступной для всех потоков одного блока. Поскольку они выполняются на одном мультипроцессоре.

Назначение памяти

Константная память (constant memory) является достаточно быстрой из доступных GPU. Отличительной особенностью константной памяти является возможность записи данных с хоста, но при этом в пределах GPU возможно лишь чтение из этой памяти, что и обуславливает её название. Для размещения данных в константной памяти предусмотрен спецификатор `__constant__`. Если необходимо использовать массив в константной памяти, то его размер необходимо указать заранее, так как динамическое выделение в отличие от глобальной памяти в константной не поддерживается. Для записи с хоста в константную память используется функция `cudaMemcpyToSymbol`, и для копирования с device'a на хост `cudaMemcpyFromSymbol`.

Текстурная память (texture memory) предназначена главным образом для работы с текстурами. Текстурная память имеет специфические особенности в адресации, чтении и записи данных.

Транспонирование матрицы

Ниже будут приведены примеры ядер транспонирования одной строки в один столбец. Одно ядро напрямую копирует данный из глобальной памяти в глобальную, а другая использует *shared* память.

Эти ядра могут быть вызваны программой транспонирования матрицы выполняемой на host-e.

Время выполнения с первым ядром: **116 мс.**

со вторым: **398 мс.**

Прямое копирование в global память

```
__global__ void transposeM_G(float* inputMatrix, float* outputMatrix, int width,  
int height)  
{  
    int xIndex = blockDim.x * blockIdx.x + threadIdx.x;  
    int yIndex = blockDim.y * blockIdx.y + threadIdx.y;  
  
    if ((xIndex < width) && (yIndex < height))  
    {  
        //Линейный индекс элемента строки исходной матрицы  
        int inputIdx = xIndex + width * yIndex;  
  
        //Линейный индекс элемента столбца матрицы-результата  
        int outputIdx = yIndex + height * xIndex;  
  
        outputMatrix[outputIdx] = inputMatrix[inputIdx];  
    }  
}
```

Копирование через shared память

```
__global__ void transposeM_S(float* inputMatrix, float* outputMatrix, int width,
int height)
{
    __shared__ float temp[BLOCK_DIM][BLOCK_DIM];

    int xIndex = blockIdx.x * blockDim.x + threadIdx.x;
    int yIndex = blockIdx.y * blockDim.y + threadIdx.y;

    if ((xIndex < width) && (yIndex < height))
    {
        // Линейный индекс элемента строки исходной матрицы
        int idx = yIndex * width + xIndex;

        //Копируем элементы исходной матрицы
        temp[threadIdx.y][threadIdx.x] = inputMatrix[idx];
    }

    //Синхронизация всех потоков в блоке
    __syncthreads();
}
```

Копирование через shared память

```
xIndex = blockIdx.y * blockDim.y + threadIdx.x;  
yIndex = blockIdx.x * blockDim.x + threadIdx.y;  
  
if ((xIndex < height) && (yIndex < width))  
{  
    // Линейный индекс элемента строки исходной матрицы  
    int idx = yIndex * height + xIndex;  
  
    //Копируем элементы исходной матрицы  
    outputMatrix[idx] = temp[threadIdx.x][threadIdx.y];  
}  
}
```