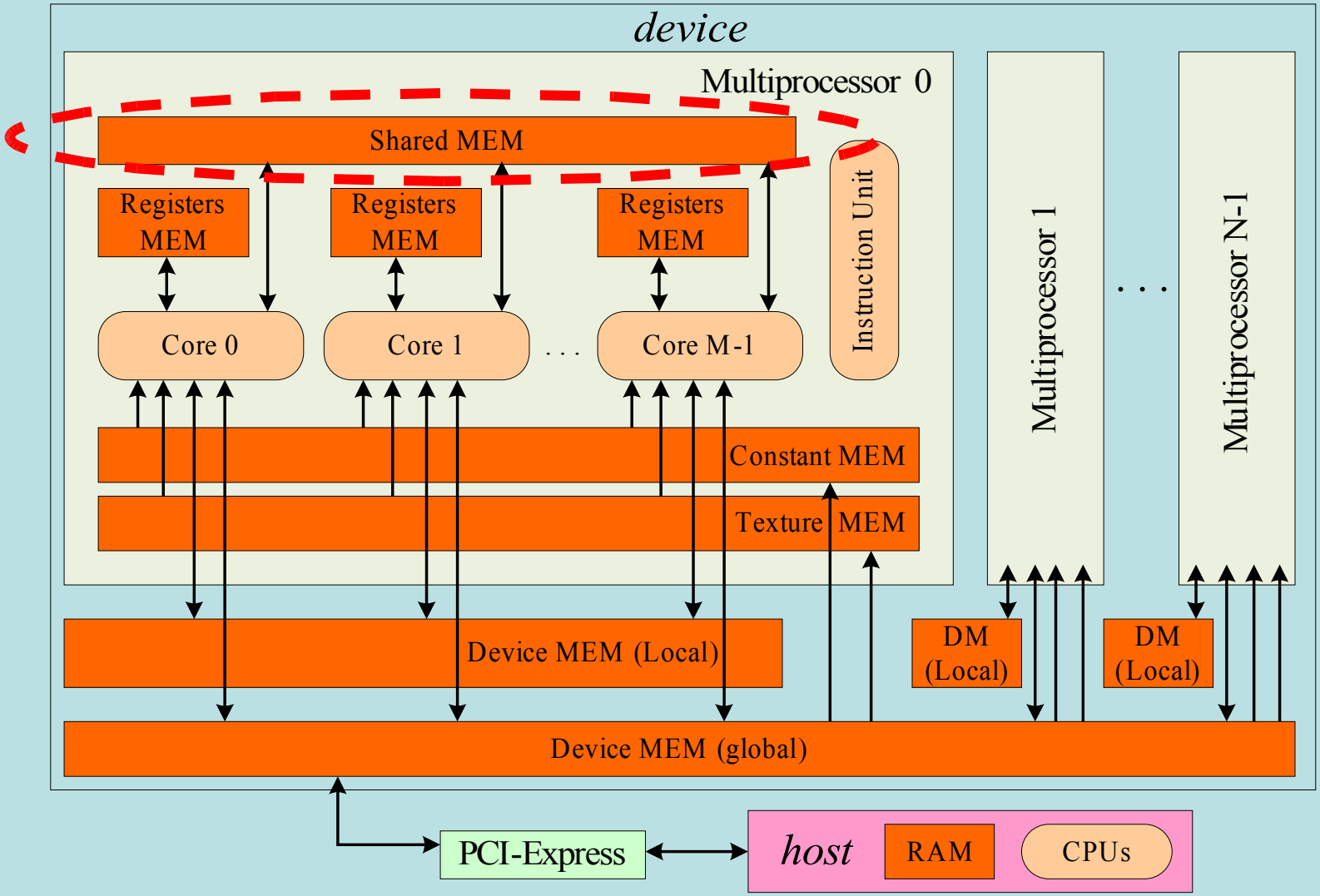
1.3. Выполнение и взаимодействие потоков ядра в CUDA (окончание)

Доступ к памяти в NVIDIA GPU (CUDA)



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-е.

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];
  }
 }
```

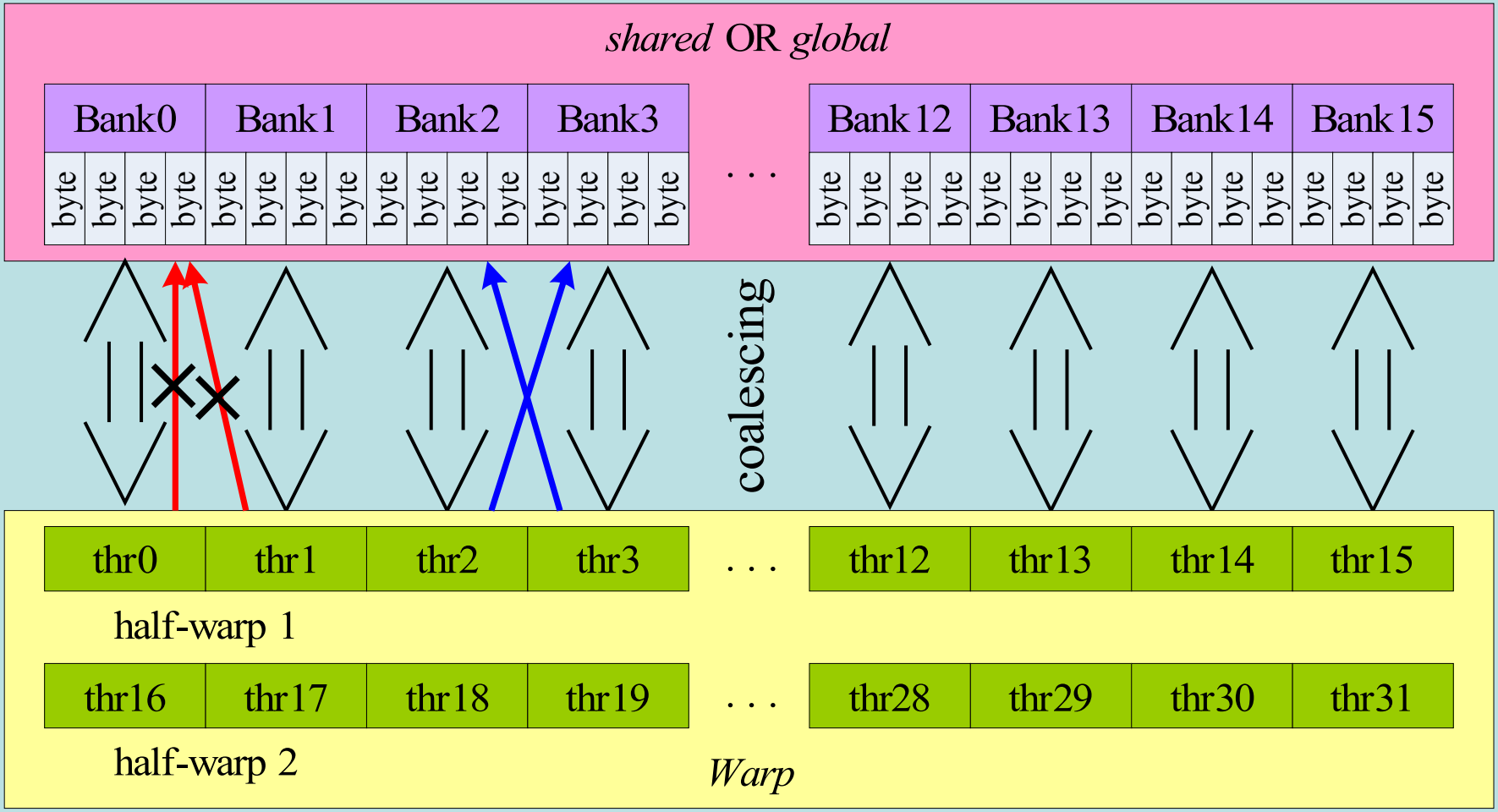
Копирование через shared память

```
__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];
  }
}
```

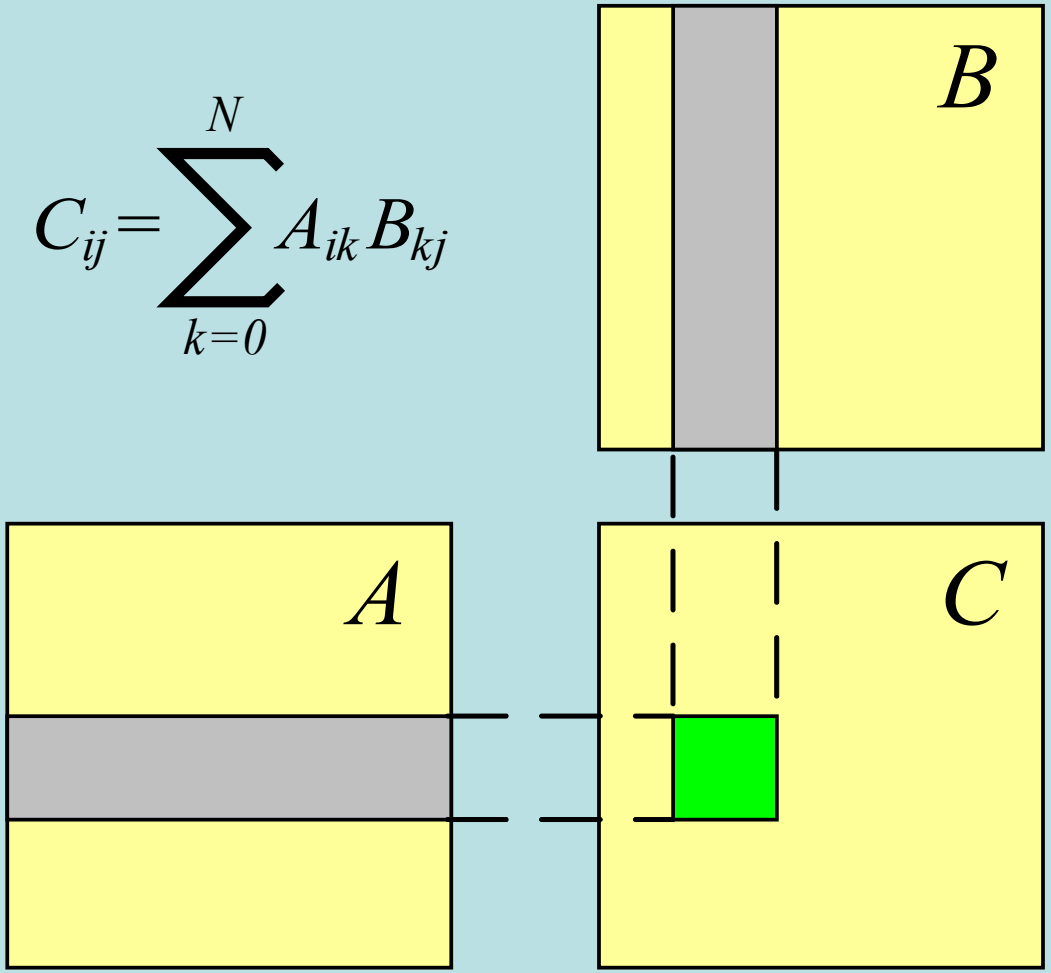
Использование объединенного запроса к памяти (*coalescing*)



```
__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;


	int aBegin = n * BLOCK_SIZE * by;
	int aEnd = aBegin + n - 1;

	int aStep = BLOCK_SIZE;
	int bBegin = BLOCK_SIZE * bx;
	int bStep = BLOCK_SIZE * n;
	float sum = 0.0f;

	for ( int ia = aBegin, ib = bBegin; ia <= aEnd; ia += aStep, ib += bStep )
	{
		__shared__ float as [BLOCK_SIZE][BLOCK_SIZE];
		__shared__ float bs [BLOCK_SIZE][BLOCK_SIZE];

		as [ty][tx] = a [ia + n * ty + tx];
		bs [ty][tx] = b [ib + n * ty + tx];

		__syncthreads();


		for ( int k = 0; k < BLOCK_SIZE; k++ ) sum += as [ty][k] * bs [k][tx];


		__syncthreads();
	}
	int ic = n * BLOCK_SIZE * by + BLOCK_SIZE * bx;
	c [ic + n * ty + tx] = sum;
}
```

Перемножение матриц **A*B**

$$C_{ij} = \sum_{k=0}^{N} A_{ik} B_{kj}$$



```
__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;


    int aBegin = n * BLOCK_SIZE * by;
    int aEnd = aBegin + n - 1;

    int aStep = BLOCK_SIZE;
    int bBegin = BLOCK_SIZE * bx;
    int bStep = BLOCK_SIZE * n;
    float sum = 0.0f;
```

```
        for ( int ia = aBegin, ib = bBegin; ia <= aEnd; ia +=
    aStep, ib += bStep )
    {
        __shared__ float as [BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float bs [BLOCK_SIZE][BLOCK_SIZE];

        as [ty][tx] = a [ia + n * ty + tx];
        bs [ty][tx] = b [ib + n * ty + tx];

        __syncthreads();
        for ( int k = 0; k < BLOCK_SIZE; k++ )
                sum += as [ty][k] * bs [k][tx];
        __syncthreads();
    }
    int ic = n * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    c [ic + n * ty + tx] = sum;
}
```

Функция main

```
#include <stdio.h>

#define BLOCK_SIZE      16
#define  N              1024

int main ( int argc, char *  argv [] )
{
    int     numBytes = N * N * sizeof ( float );

    float * a = new float [N*N];
    float * b = new float [N*N];
    float * c = new float [N*N];

    for ( int i = 0; i < N; i++ )
        for ( int j = 0; j < N; j++ )
        {
            a [i] = 0.0f;
            b [i] = 1.0f;
        }
```

```
float * adev = NULL;
float * bdev = NULL;
float * cdev = NULL;

cudaMalloc ( (void**)&adev, numBytes );
cudaMalloc ( (void**)&bdev, numBytes );
cudaMalloc ( (void**)&cdev, numBytes );


 dim3 threads ( BLOCK_SIZE, BLOCK_SIZE );
 dim3 blocks  ( N / threads.x, N / threads.y);


cudaEvent_t start, stop;
float gpuTime = 0.0f;

cudaEventCreate ( &start );
cudaEventCreate ( &stop );
```

```
cudaEventRecord ( start, 0 );
cudaMemcpy      ( adev, a, numBytes, cudaMemcpyHostToDevice );
cudaMemcpy      ( bdev, b, numBytes, cudaMemcpyHostToDevice );

matMult<<<blocks, threads>>> ( adev, bdev, N, cdev );

cudaMemcpy      ( c, cdev, numBytes, cudaMemcpyDeviceToHost );
cudaEventRecord ( stop, 0 );

 cudaEventSynchronize ( stop );
 cudaEventElapsedTime ( &gpuTime, start, stop );


printf("time spent executing by the GPU: %.2f millseconds\n", gpuTime );
```

Конец программы

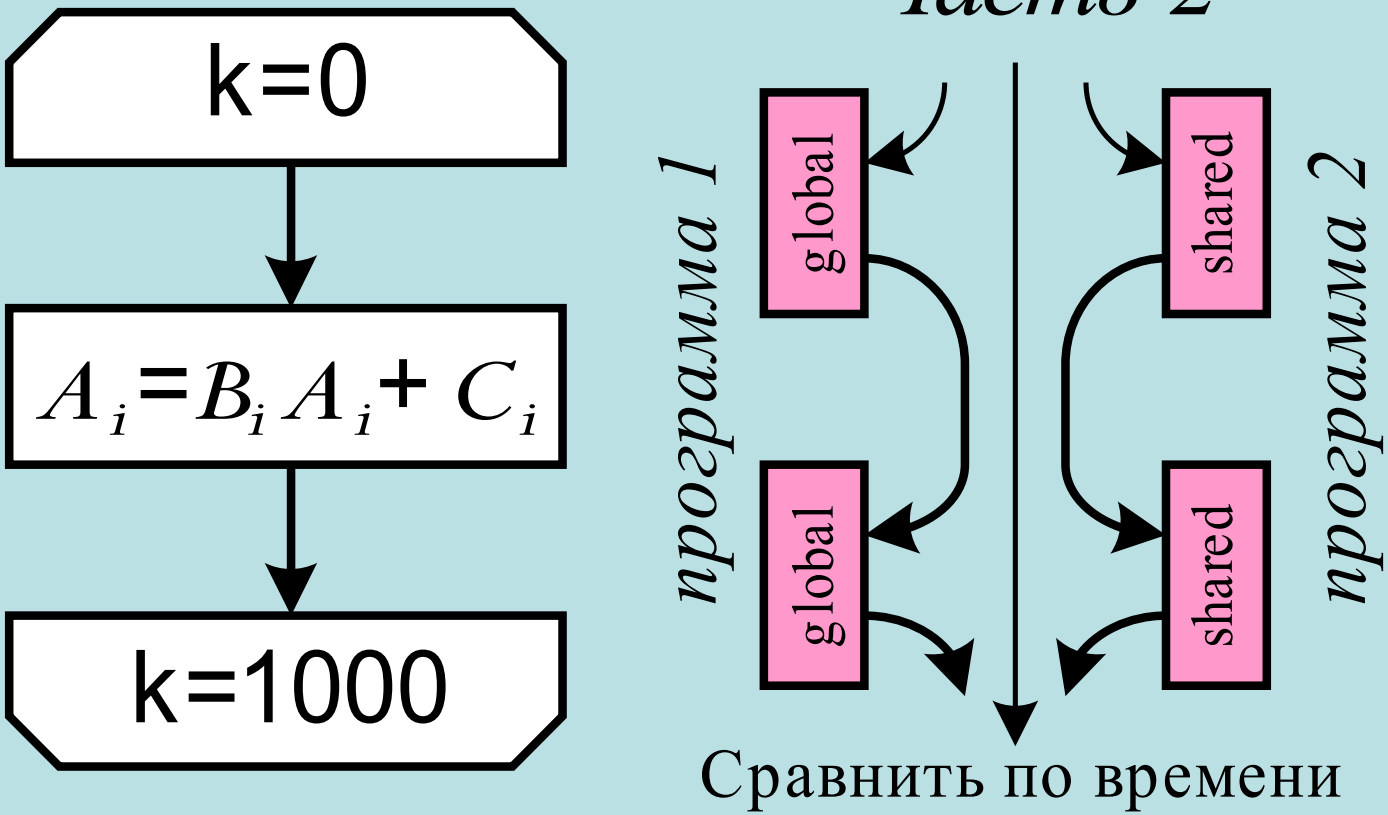
```
    cudaEventDestroy ( start );
    cudaEventDestroy ( stop  );
    cudaFree         ( adev  );
    cudaFree         ( bdev  );
    cudaFree         ( cdev  );

    delete a; delete b; delete c;

    return 0;
}
```

```
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;
}
```

ЛР №4 (по книге ЛР №3)

$Y_i = B_i A_i + C_i$ *Часть 1*

Часть 2

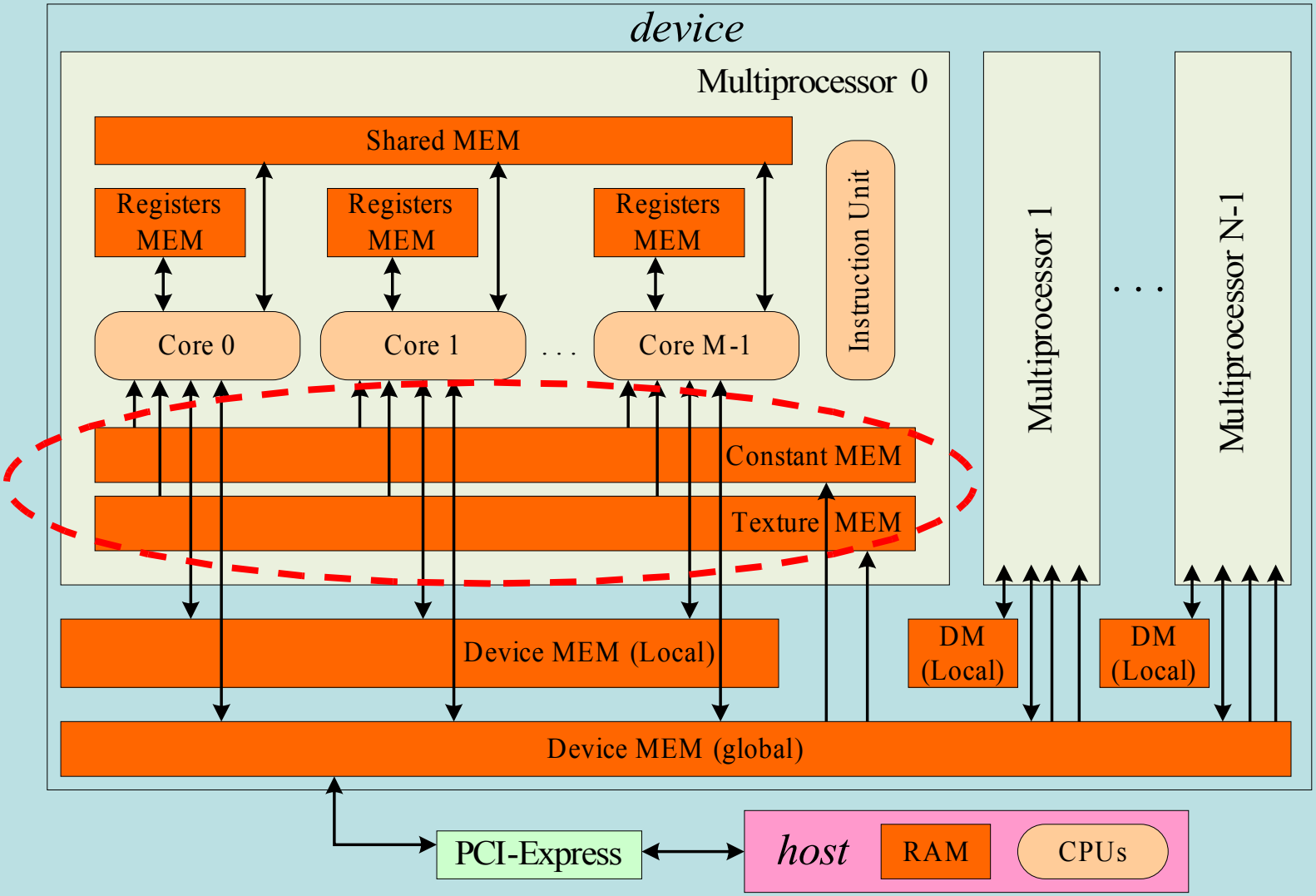


1.4. Константная память

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)

Доступ к памяти в NVIDIA GPU (CUDA)



Шаблон программы с константной памятью

```
#define N 100

__constant__ int gpu_buffer[N];

__global__ void  kernel()
{
   int a = gpu_buffer[0];
   int b = gpu_buffer[1] + gpu_buffer[2];
}

int main()
{
   int cpu_buffer[N];
   ...
   cudaMemcpyToSymbol(gpu_buffer, cpu_buffer, sizeof(int)*N);
   // Вызов ядра
}
```

1.5. Текстурная память
1.6. Трассировка лучей

Основные авторы презентаций: **Боресков А.В., Харламов А.А.**

Каталог презентаций
http://nvidia.esyr.org/files/presentations/

Текстурная память. Лекция №4
http://nvidia.esyr.org/files/presentations/0830_CUDA_Texture.pdf

Трассировка лучей. Лекция №6
http://nvidia.esyr.org/files/presentations/0901_CUDA_Raytracing.pdf