Multiple blocks

Grid concept, Synchronization

GPU Programming

http://cuda.nik.uni-obuda.hu

Szénási Sándor
szenasi.sandor@nik.uni-obuda.hu

GPU Education Center of Óbuda University

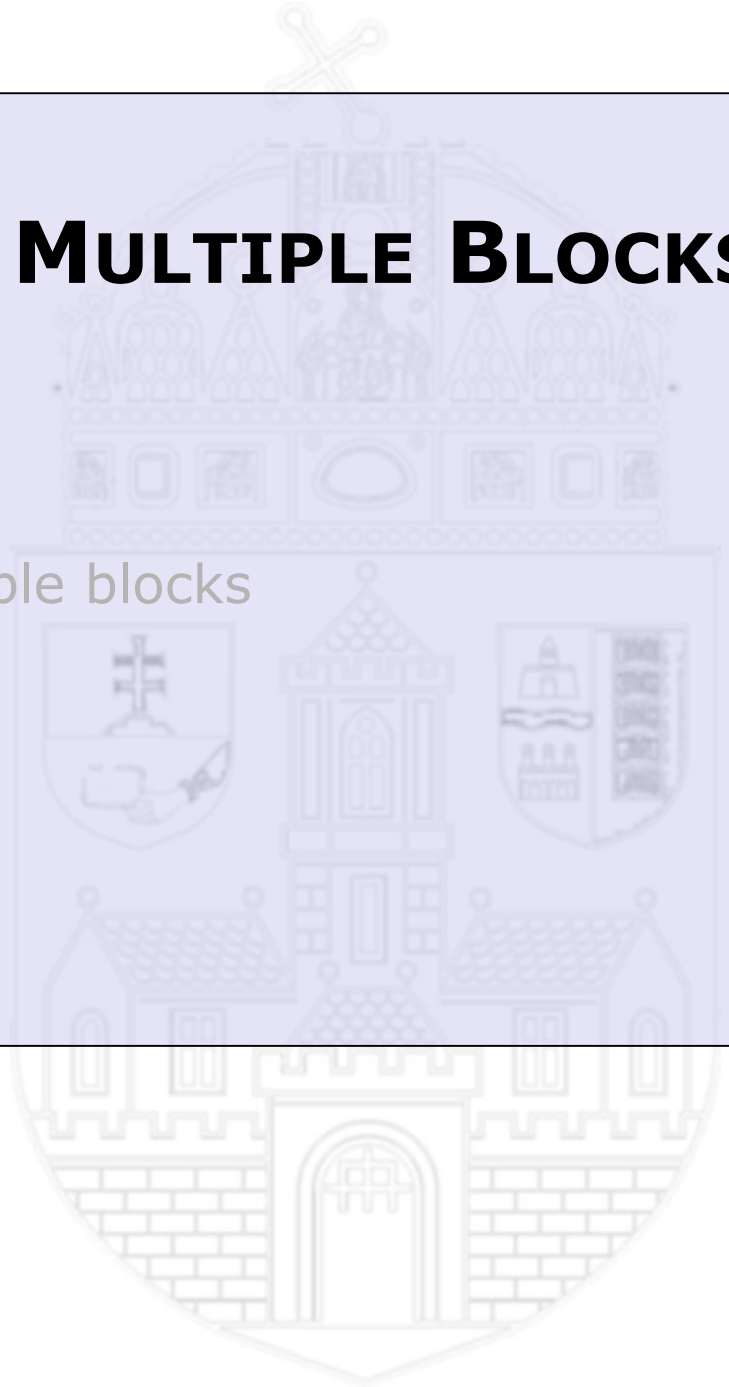
NVIDIA GPU EDUCATION CENTER

Multiple Blocks

Grid concept

Launching multiple blocks

Shared memory



Blocks and grid

Thread blocks

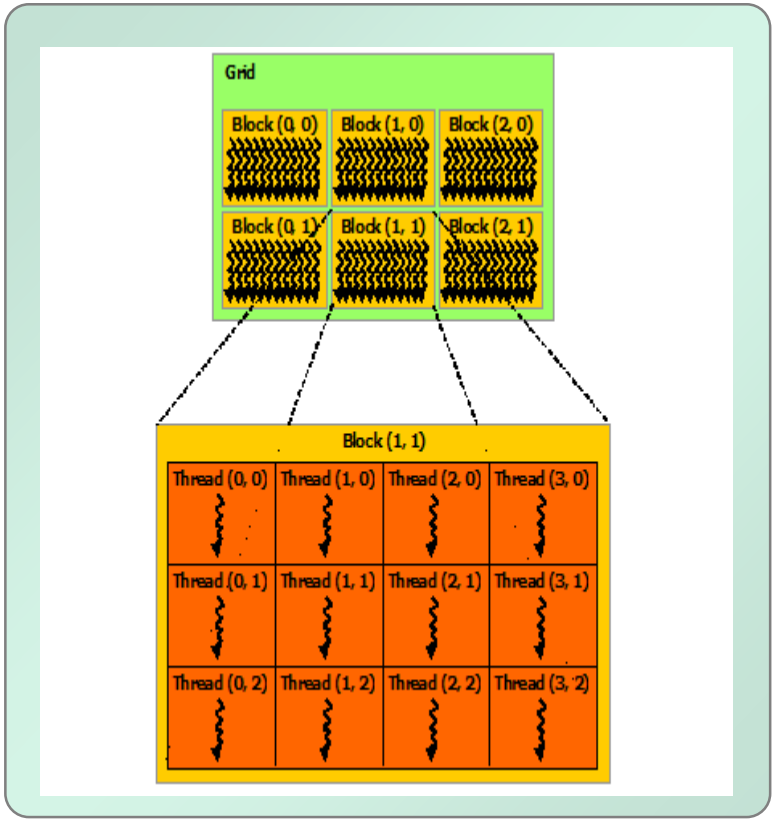
- CUDA devices has a limitation for the maximal number of parallel executable threads. The index space of a complex task can be greater than this limit (for example maximum 512 thread ↔ 100x100 matrix = 10000 threads)
- In these cases the device will split the index space to smaller thread blocks. The scheduling mechanism will process all of these blocks and it will decide the processing order (one-by-one or in case of more than one multiprocessors in a parallel way)
- The hierarchy of blocks is the grid

Block splitting method

- In CUDA, the framework will create, initialize and start all of the threads. The creation, initialization of the blocks is the framework's task too
- The programmer can influence this operation via the following parameters (kernel start parameters):
  - number of threads within a single block (1,2 or 3 dimension)
  - number of blocks in the grid (1, 2 or 3 dimension)

Thread block indices

- Thread blocks also have a unique ID. So a thread can reach the owner block data
- It can be
  - 1 dimensional
  - 2 dimensional
  - 3 dimensional (Fermi and later)
- Block ID is available in the kernel via blockIdx variable
- In case of multidimensional index space, the *threadIdx* is a structure with the following fields:
  - blockIdx.x
  - blockIdx.y
  - blockIdx.z



http://docs.nvidia.com/cuda/cuda-c-programming-guide/graphics/grid-of-thread-blocks.png

Global and local indices

Local identifier

- Every thread have a local identifier, it is stored in the previously introduced threadIdx variable
- This number shows the thread's place within the block
- The identifier of the „first" thread is (based on the block dimensions):
  0 or [0,0] or [0,0,0]

Global identifier

- In case of more than one block, the local identifier is not unique anymore
- We know the identifier of the block (the owner of the thread), the previously introduced blockIdx variable and the size of the blocks (blockDim *variable*), we can calculate the global identifier of the thread:

  *Global_x_component = blockIdx.x * blockDim.x + threadIdx.x*
- The programmer can not send unique parameters to the threads (for example, which matrix element to process). Therefore the thread must use it's unique global identifier to get it's actual parameters

Some useful formulas

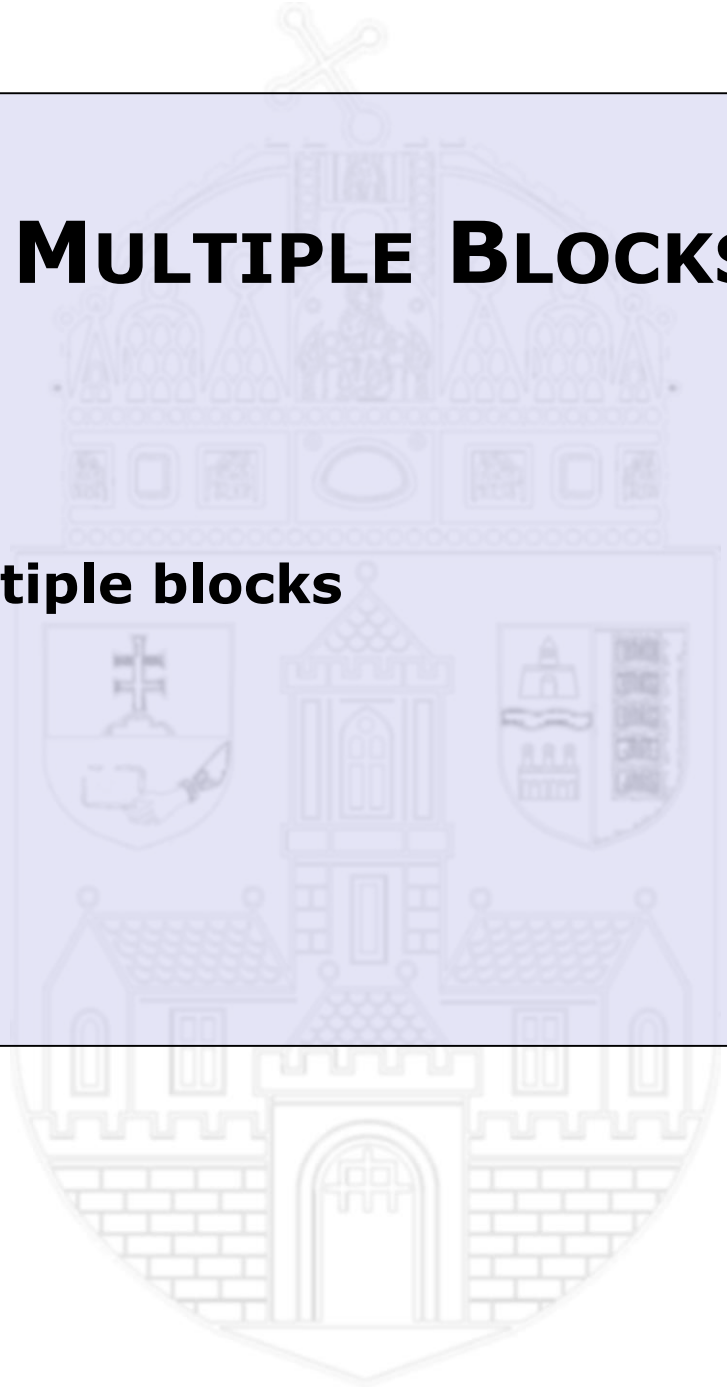
- Size of the index space: G_x, G_y
  (derived from the problem space)
- Block size: S_x, S_y
  (based on the current hardware)
- Number of threads: $G_x * G_y$
  (number of all threads)
- Global identifiers: $(0..G_x - 1, 0..G_y - 1)$
  (unique identifier for all threads)
- Number of blocks: $(W_x, W_y) = ((G_x - 1)/ S_x + 1, (G_y - 1)/ S_y + 1)$
  (number of block for the given block size)
- Global identifier: $(g_x, g_y) = (w_x * S_x + s_x, w_y * S_y + s_y)$
- Local identifier: $(w_x, w_y) = ((g_x - s_x) / S_x, (g_y - s_y) / S_y)$

Multiple Blocks

Grid concept

Launching multiple blocks

Shared memory



Using multiple-block kernel

- If we want to process 2000 items which is more than the number of maximum threads in a single block, we have to create more than one blocks in the device

```
1  __global__ void vectorMul(float* A, int N)
2  {
3     int i = blockIdx.x * blockDim.x + threadIdx.x;
4     if (i < N)
5     {
6        A[i] = A[i] * 2;
7     }
8  }
```

- In the first line the kernel calculates it's global identifier. This will be a globally unique number for each threads in each blocks

Invoking a multiple-block kernel

- If we want to process 1000 element and the maximum block size is 512 (with Compute Capability 1.0), we can use the following parameters:
- 4 blocks (identifiers are 0, 1, 2 and 3)
- 250 threads (local identifiers are 0 .. 249)

```
1   float*A = ...
2   ... transfer data CPU→GPU ...
3   vectorMul<<<4, 250>>>(A, 1000);
4   ... transfer results GPU→CPU ...
```

- If we don't know the number of elements at compile time, we can calculate the correct block and thread numbers (N – vector size, BM – chosen block size)
  - Number of blocks: (N-1) / BM + 1
  - Size of blocks: BM

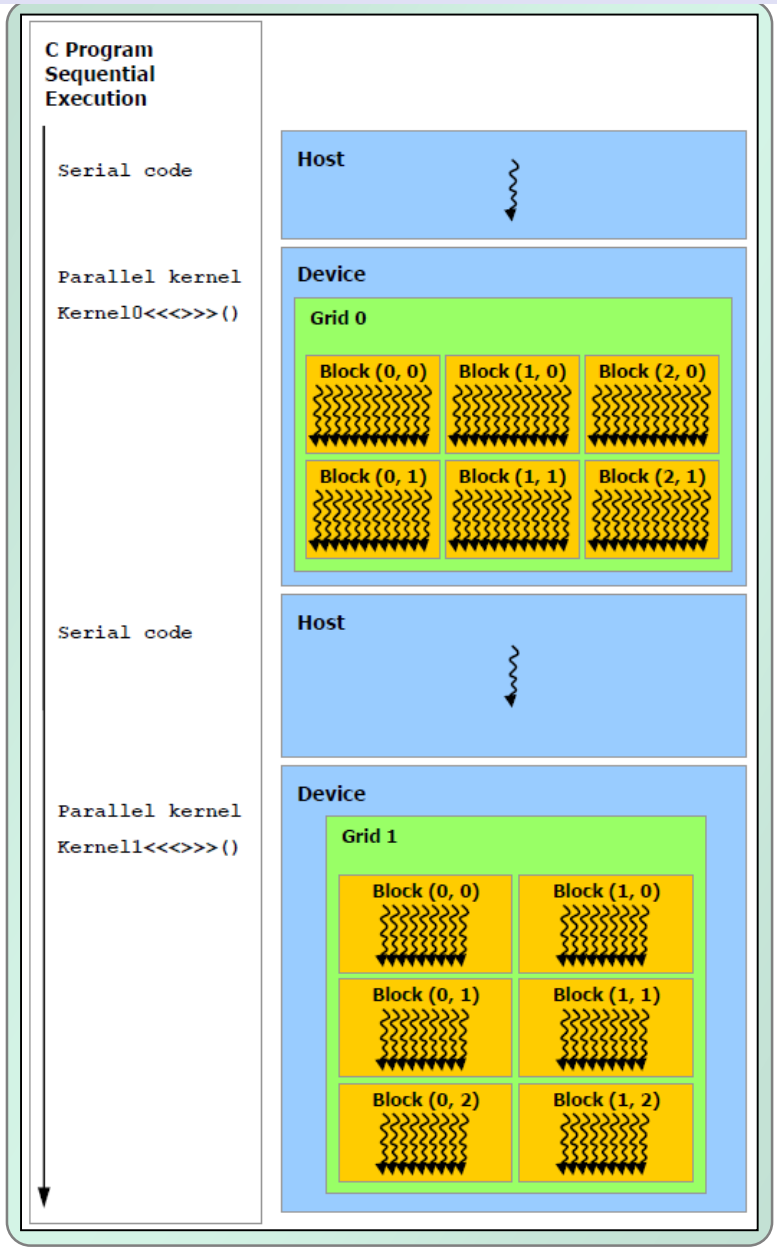
Kernel execution modes

Synchronous execution

- There is no parallelism between the host and device codes
- We can use this method by explicit synchronization calls

Asynchronous execution

- In case of modern devices, the followings can run parallel
  - host code
  - device kernel code
  - host to device memory transfer
  - device to host memory transfer
- With multiple devices one host can control more than one CUDA device
- In case of Fermi an later cards, one device can run more than one thread groups asynchronously
- In case of Kepler and later cards, any kernel can start other kernels
- We will discuss these later



Synchronization – host library

Device level synchronization

- cudaError_t cudaDeviceSynchonize()
  - effect: blocks until the device has completed all preceding requested tasks
  - scope: all operations in the given device
  - Affected operations:
    - kernel launches
    - memory transfers
- The result of the function is an errorcode if one of the preceding tasks has failed

```
1  cudaMemcpy(devA, A, sizeof(float) * N, cudaMemcpyHostToDevice);
2  ...
3  cudaDeviceSynchronize();
4  ...
```

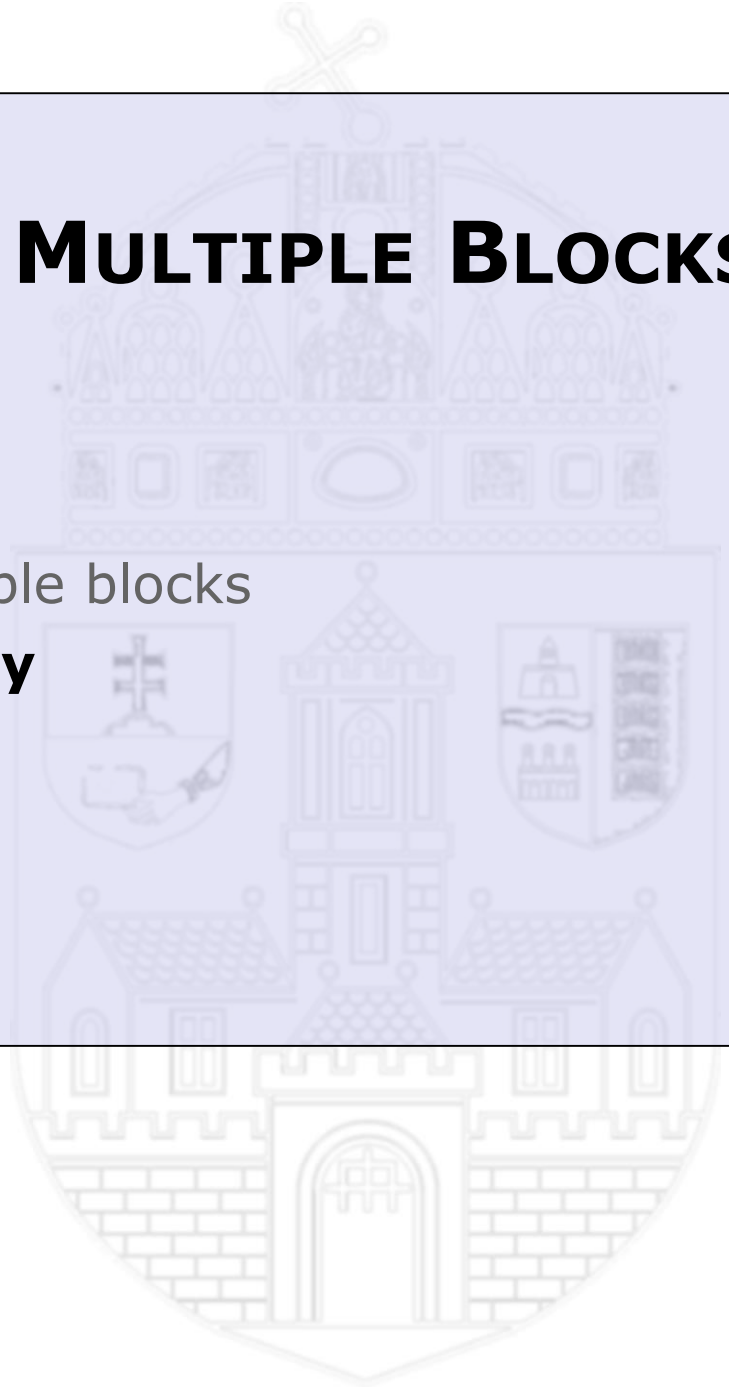
- There are some operations using an implicit synchronization, like
  - *some kind of memory allocations*
  - *some configuration changes (cache, etc.)*
  - *etc.*
- Streams will give a more sophisticated way to synchronize operations

Multiple Blocks

Grid concept

Launching multiple blocks

Shared memory



Shared memory

- On-chip, fast memory region
- Declare it with __shared__ keyword

```
__shared__ int a;
```

Block level access

- All threads of a block can access variables in the shared memory region
- Threads of other blocks cannot access these variables
- Host cannot access shared memory regions

```
__shared__ int a;

…

Kernel<<<5,10>>>( );
```

- In this example, there will be 5 integer variables in the device
- There will be 50 threads grouped into blocks of 10 threads
- All threads in the same group can access the same „a” variable