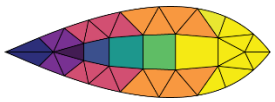

Thread mapping in CUDA

by Dr. Toby Potter

It can be confusing as to how CUDA hardware threads (CUDA cores) are mapped to the available work. This article shows how to disentangle the CUDA hardware thread mapping.

Blocks

When a CUDA kernel executes, it executes hardware threads in batches of 32 threads called warps. Threads are referenced according to 3D rectangular cuboid called a block, with axes (x,y,z). At present blocks are limited to a maximum 1024 threads. The maximum length of a block in the z direction is 64 threads and any of the axes for the block may have a "length" of 1 thread. A common choice for the dimensions of a block is (16,16,1) so that a block has 256 threads which is a multiple of the 32-thread warp size.

Grid

Blocks are referenced according to a 3D rectangular cuboid structure called a grid, also with axes (x,y,z). The maximum number of blocks along a dimension of the grid is 65536, and the minimum number is 1. Parameters that you specify to the kernel include both the dimensions of the block, and the dimensions of the grid.

Structures available during execution

During a CUDA kernel execution, the following structures are available to the kernel code:

- `gridDim.x, gridDim.y, gridDim.z`; the number of **blocks** along the x, y, and z dimensions of the **grid**
- `blockIdx.x, blockIdx.y, blockIdx.z`; the zero-based index of the **block** along the x,y, and z dimensions of the **grid**
- `blockDim.x, blockDim.y, blockDim.z`; the number of **threads** along the x, y, and z dimensions of the **block**.
- `threadIdx.x, threadIdx.y, threadIdx.z`; the zero-based index of the **thread** along the x, y, and z dimensions of the **bloc**k.

Mapping order

The order of threads in blocks and the order of blocks in grids are both arranged according to a FORTRAN-style column-major format. That is, they are arranged along x axis first, then the y axis, and finally the z axis. Care must be taken when CUDA kernels are applied to multidimensional arrays whose ordering is in row-based (C-style) format, as the thread ordering is opposite to the layout of the array. **Note**: The order in which the blocks and threads are executed in is **undefined** and cannot be relied upon in the execution of the kernel.

Example mapping

In the diagram below is a mapping where the grid has size (5,4,1), that is, it has 5 blocks in the x direction, 4 blocks in the y direction, and 1 block in the z direction. Each block is of size (5,5,1) with 5 threads along the x and y directions, and 1 thread along the z direction. At the grid level (on the left), the tuple for each block is the 3D index, e.g. (0,0,0), and below the 3D index is the 1D index of each block, e.g. [0]. At the block level (on the right), a similar indexing scheme applies, where the tuple is the 3D index of the thread within the block and the number in the square bracket is the 1D index of the thread within the block. During execution, the CUDA threads are mapped to the problem in an undefined manner. In the diagram below we show randomly completed threads and blocks as green to highlight the fact that the order of execution for threads is undefined.

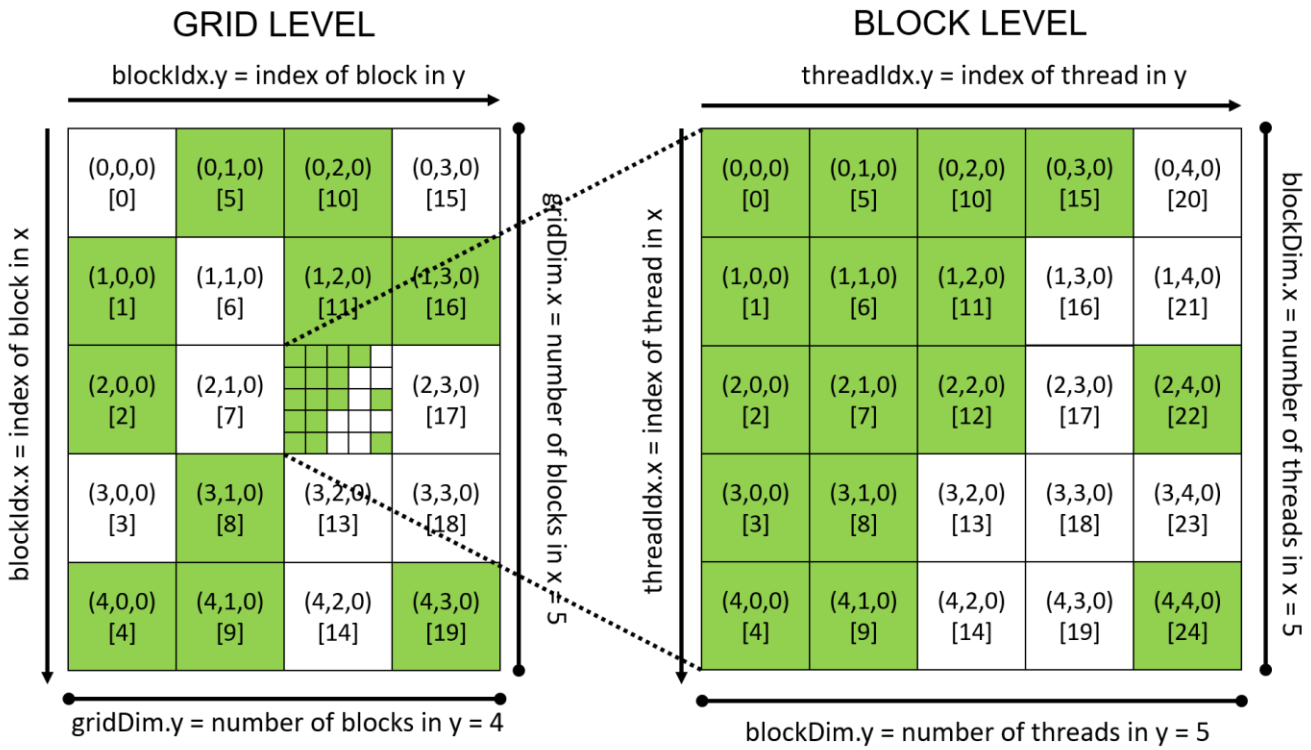

Figure 1. Layout of how blocks are arranged within grids (on the left), and how threads are arranged within blocks (on the right). Threads and blocks highlighted in green have completed, the tuple e.g. (0,0,0) is the 3D index of the block/thread relative to its parent grid/block, and the number in square brackets, e.g. [0] is the 1D index relative to its parent grid/block.

Obtaining a 3D index of a block relative to its grid

This is just (`blockIdx.x, blockIdx.y, blockIdx.z`)

Obtaining a 3D index of a thread relative to its block

This is just (`threadIdx.x, threadIdx.y, threadIdx.z`)

Obtaining a thread's 3D index relative to the grid.

In order to get a thread's 3D index relative to the grid, one has to multiply the index of the block by the number of threads along that dimension of the block, and add the component of 3D thread index. In a CUDA kernel it looks like this:

```
int x_index = blockIdx.x*blockDim.x + threadID.x;
int y_index = blockIdx.y*blockDim.y + threadID.y;
int z_index = blockIdx.z*blockDim.z + threadID.z;
```

Obtaining the 1D index of a block within a grid

The 1D index of a block within the grid is given by the dot product of the block indices with the vector containing the number of blocks to stride over in each dimension.

```
blockId_grid = [blockIdx.x, blockIdx.y, blockIdx.z]·[1, gridDim.x,
gridDim.x*gridDim.y]
```

Obtaining the 1D index of a thread within a block

Similarly, the 1D index of a thread within a block is obtained as the dot product between the vector of indices and the vector containing the number threads to stride over in each dimension.

```
threadId_block = [threadIdx.x, threadIdx.y, threadIdx.z]·[1,
blockDim.x, blockDim.x*blockDim.y]
```

Obtaining a thread's 1D index within the grid.

We obtain the 1D index of a **thread** within the **grid** by first obtaining the 1D index of the **block**, multiplying it by the number of **threads** per **block**, and adding the 1D index of the thread within the block.

```
threadId_grid = blockId_grid *(blockDim.x*blockDim.y*blockDim.z)
                +threadId_block
```

In C code within a kernel this could take the form:

```
size_t blockId_grid = blockIdx.x + blockIdx.y*gridDim.x
                    + blockIdx.z*gridDim.x*gridDim.y;

size_t threads_per_block = blockDim.x*blockDim.y*blockDim.z;

size_t threadId_grid= blockId_grid * threads_per_block
                    +threadIdx.x + threadIdx.y*blockDim.x
                    +threadIdx.z*blockDim.x*blockDim.y;
```

A slight performance improvement might be obtained by using `int` instead of `size_t`, however you do risk integer overflow if the number of elements exceeds 2147483647 or approximately 8.6GB on the GPU for 32-bit floating point arrays.

If any of the dimensions of the grid have size 1 (e.g. `gridDim.z==1`), then the corresponding `blockIdx` term for that dimension (e.g. `blockIdx.z*gridDim.x*gridDim.y),` is always zero and you might consider removing that term from the calculation of `blockId_grid`.

Similarly, if any of the dimensions of a block have size 1 (e.g. `blockDim.y==1`), then the corresponding `threadIdx` term (e.g `threadIdx.y*blockDim.x`), is always zero and may be removed from the calculation of `threadId_grid`.

Reference

Information in this sheet was sourced from the *CUDA C programming guide*, located at http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#axzz4qeshgxpy. No responsibility is assumed for the accuracy of this material.

License