

CUDA

(Grids, Blocks, Warps, Threads)

Overview

1. CUDA Thread Organization
2. Mapping Threads to Multidimensional Data
3. Synchronization and Transparent Scalability
4. Querying Device Properties
5. Thread Assignment
6. Thread Scheduling and Latency Tolerance
7. Conclusions

CUDA Thread Organization

CUDA Thread Organization

Grids consist of blocks.

Blocks consist of threads.

A grid can contain up to 3 dimensions of blocks, and a block can contain up to 3 dimensions of threads.

A grid can have 1 to 65535 blocks, and a block (on most devices) can have 1 to 512 threads.

CUDA Thread Organization

The number of total threads created will be:

$\text{total threads} = \text{number of grids} * \text{number of blocks in each grid} * \text{number of threads in each block}$

CUDA Thread Organization

In general use, grids tend to be two dimensional, while blocks are three dimensional. However this really depends the most on the application you are writing.

CUDA provides a `struct` called `dim3`, which can be used to specify the three dimensions of the grids and blocks used to execute your kernel:

```
dim3 dimGrid(5, 2, 1);  
dim3 dimBlock(4, 3, 6);  
KernelFunction<<<dimGrid, dimBlock>>> (...);
```

CUDA Thread Organization

In general use, grids tend to be two dimensional, while blocks are three dimensional. However this really depends the most on the application you are writing.

CUDA provides a `struct` called `dim3`, which can be used to specify the three dimensions of the grids and blocks used to execute your kernel:

```
dim3 dimGrid(5, 2, 1);  
dim3 dimBlock(4, 3, 6);  
KernelFunction<<<dimGrid, dimBlock>>> (...);
```

How many threads will this make?

CUDA Thread Organization

```
dim3 dimGrid(5, 2, 1);  
dim3 dimBlock(4, 3, 6);  
KernelFunction<<<dimGrid, dimBlock>>> (...);
```

The for dimGrid, $x = 5, y = 2, z = 1$, and for dimBlock, $x = 4, y = 3, z = 6$.

The threads created will have:

```
gridDim.x = 5, blockIdx.x = 0 ... 4  
gridDim.y = 2, blockIdx.y = 0 ... 1  
gridDim.z = 1, blockIdx.z = 0 ... 0
```

```
blockDim.x = 4, threadIdx.x = 0 ... 3  
blockDim.y = 3, threadIdx.y = 0 ... 2  
blockDim.z = 6, threadIdx.z = 0 ... 5
```

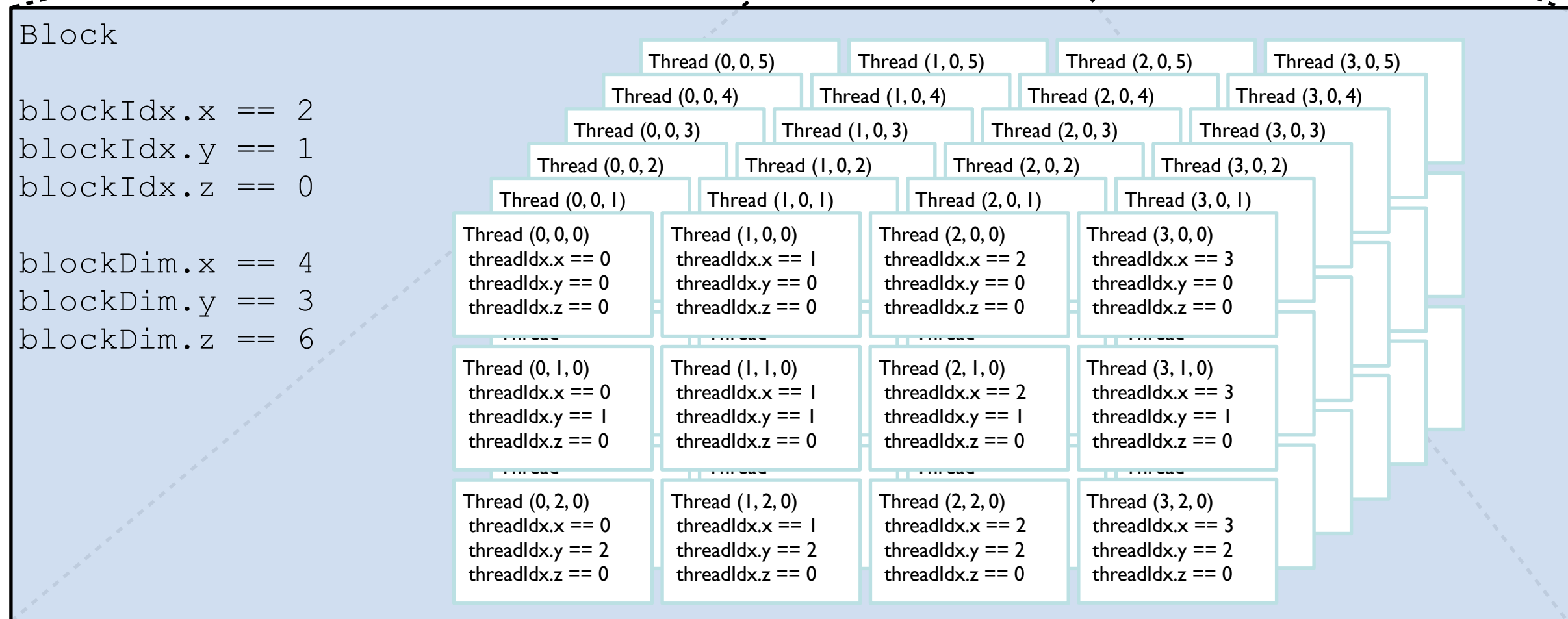
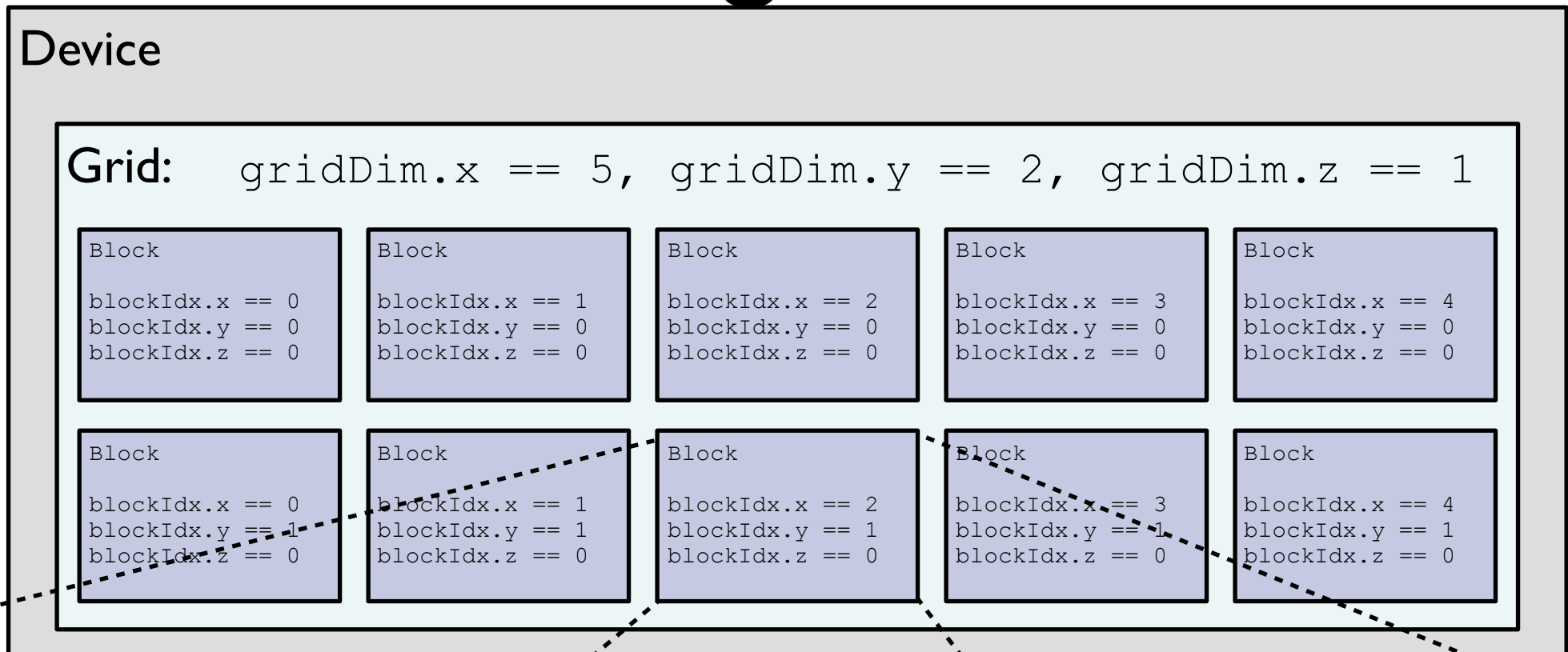
Therefore the total number of threads will be

$$5 * 2 * 1 * 4 * 3 * 6 = 720$$

CUDA Thread Organization

```
dim3 dimGrid(5, 2, 1);  
dim3 dimBlock(4, 3, 6);
```

Kernel



Mapping Threads to Multidimensional Data

Mapping Threads to Multidimensional Data

Using 1D, 2D or 3D thread/block organization is usually based on the nature of the data being used on the GPU.

For example, a black and white picture will be a 2D array of pixels, with each element in the 2D array being how dark that pixel is.

A color picture will be a 3D array of pixels, which each element in the 2D array being 3 values (typically) corresponding to the red, green and blue values of that pixel.

Mapping Threads to Multidimensional Data

Say we have a 2013×3971 pixel (black and white) picture, and we want to apply a blur function to each pixel. Our blur function assigns the value of each pixel to the average of itself and its neighbors. In this case, we need to preserve the 2D information of where the pixels are when creating the threads on the GPU.

Mapping Threads to Multidimensional Data

The standard process for performing this on the GPU is:

1. Determine an optimally or well sized block. Ideally we want our blocks to use as many threads as possible, with as few of those threads doing nothing as possible.
2. Determine how many blocks we want need. Here we need enough blocks to handle all data elements.

Mapping Threads to Multidimensional Data

So given our 2013 x 3971 pixel (black and white) picture, we may determine that a 16 x 32 block size (which gives us 512 threads) is the best block size.

Then we will need a 126 x 125 sized grid:

$$2013 / 16 = 125.8125$$

$$3971 / 32 = 124.09375$$

Note that some threads will be idle in this solution.

Mapping Threads to Multidimensional Data

Given 16 x 32 blocks in a 126 x 125 sized grid for a 2013 x 3971 pixel image:



This will make a grid of 2016 x 4000 threads.

In the right most (the last x dimension) and bottom most (last y dimension) blocks, some threads will be idle as there will be no pixels to operate on.

In this case, $(3 * 3971) + (29 * 2013) + (3 * 29) = 11,913 + 58,377 + 87 = 70,377$ threads will be idle of the $2016 * 4000 = 8,064,000$ threads created. So $\sim 0.87\%$ threads will be idle.

Mapping Threads to Multidimensional Data

CUDA doesn't allow the creation of multi-dimensional arrays with `cudaMalloc`, which means multi-dimensional arrays need to be *linearized*.

C and C++ use a *row-major layout* for their arrays in memory, while FORTRAN uses a *column-major layout*.

To access an element in a 2 dimensional array linearized in row-major layout:
$$\text{index} = \text{row} * \text{width} + \text{column}$$

To access an element in a 2 dimensional array linearized with column-major layout:
$$\text{index} = \text{column} * \text{height} + \text{row}$$


Mapping Threads to Multidimensional Data

An example of row major layout:

Conceptual Representation:

M(0,0)	M(0,1)	M(0,2)	M(0,3)
M(1,0)	M(1,1)	M(1,2)	M(1,3)
M(2,0)	M(2,1)	M(2,2)	M(2,3)
M(3,0)	M(3,1)	M(3,2)	M(3,3)

C/C++
Representation
in Memory:



M(0,0)	M(0,1)	M(0,2)	M(0,3)	M(1,0)	M(1,1)	M(1,2)	M(1,3)	M(2,0)	M(2,1)	M(2,2)	M(2,3)	M(3,0)	M(3,1)	M(3,2)	M(3,3)
--------	--------	--------	--------	--------	--------	--------	--------	--------	--------	--------	--------	--------	--------	--------	--------

Linearized:



M(0)	M(1)	M(2)	M(3)	M(4)	M(5)	M(6)	M(7)	M(8)	M(9)	M(10)	M(11)	M(12)	M(13)	M(14)	M(15)
------	------	------	------	------	------	------	------	------	------	-------	-------	-------	-------	-------	-------

Mapping Threads to Multidimensional Data

Note that this can also be expanded to 3, 4 and more dimensions.

In 2D (with x as width, y as height):

$$\text{index} = (y * \text{width}) + x$$

In 3D (with x as width, y as height, z as depth):

$$\text{index} = (z * \text{width} * \text{height}) + (y * \text{width}) + x$$

and so on...

Synchronization and Transparent Scalability

Synchronization and Transparent Scalability

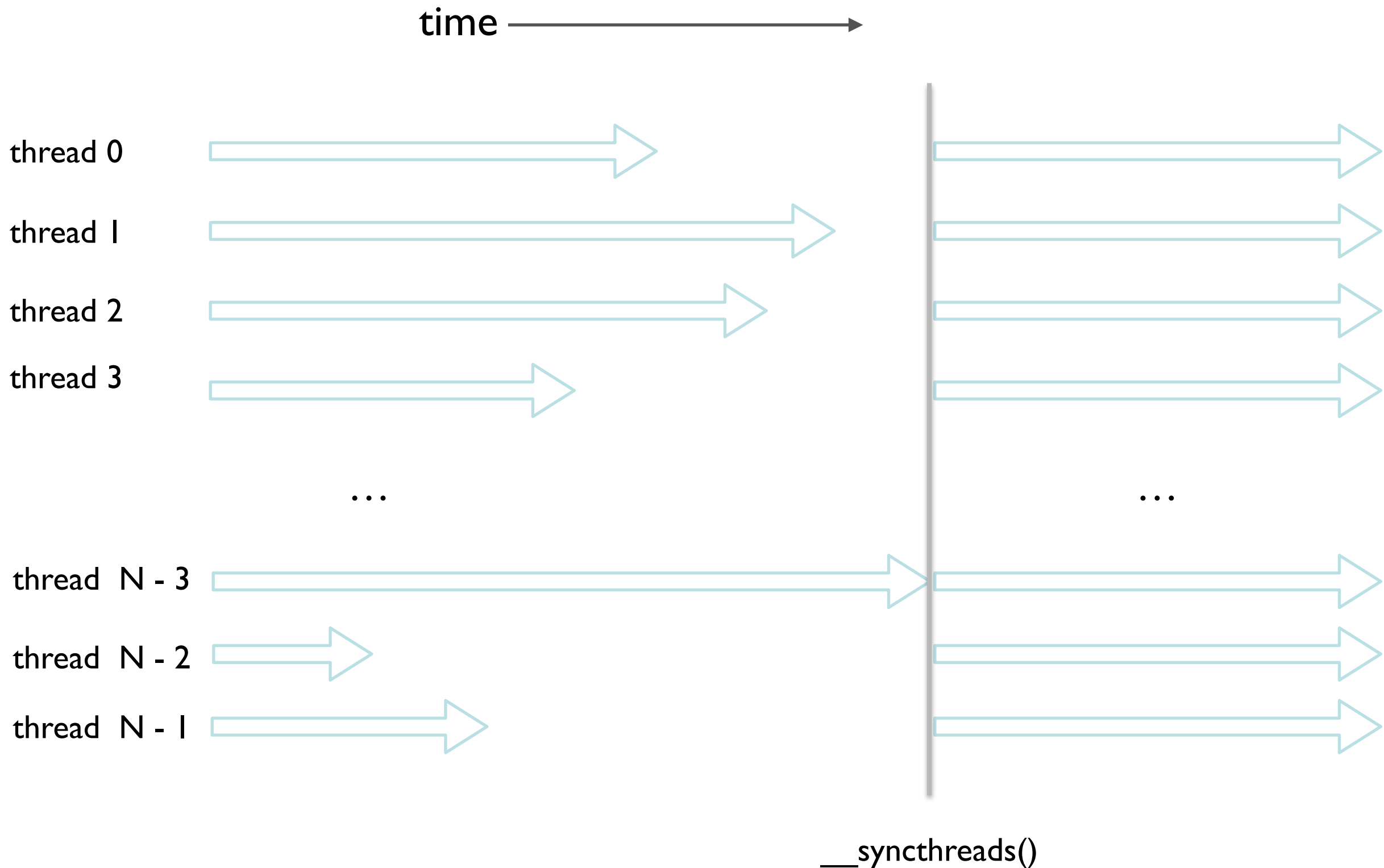
CUDA essentially provides one function to coordinate thread activities:

```
__syncthreads ()
```

This function ensures that all threads in the currently executing block have reached that function call.

__syncthreads()

Two things are very important to note with `syncthreads`. First, that it only applies to threads within the same block. Second, as it requires all threads to reach the same point before continuing, threads that complete faster will be idle until other threads catch up.



Synchronization and Transparent Scalability

`__syncthreads()` is barrier synchronization. All threads in the block *must* execute the `__syncthreads()` statement, otherwise the threads will end up blocking on the `__syncthreads()` call indefinitely. For example:

```
if (threadIdx.x % 2 == 0) {  
    __syncthreads()  
} else {  
    ...  
}
```

Will block indefinitely because the threads with an even `threadIdx.x` will have reached the `__syncthreads()` call will be waiting for the odd threads to reach that call; which they never will.

Synchronization and Transparent Scalability

Further, the `__syncthreads()` call must be the *same* `__syncthreads()` call. For example:

```
if (threadIdx.x % 2 == 0) {  
    __syncthreads()  
} else {  
    __syncthreads()  
}
```

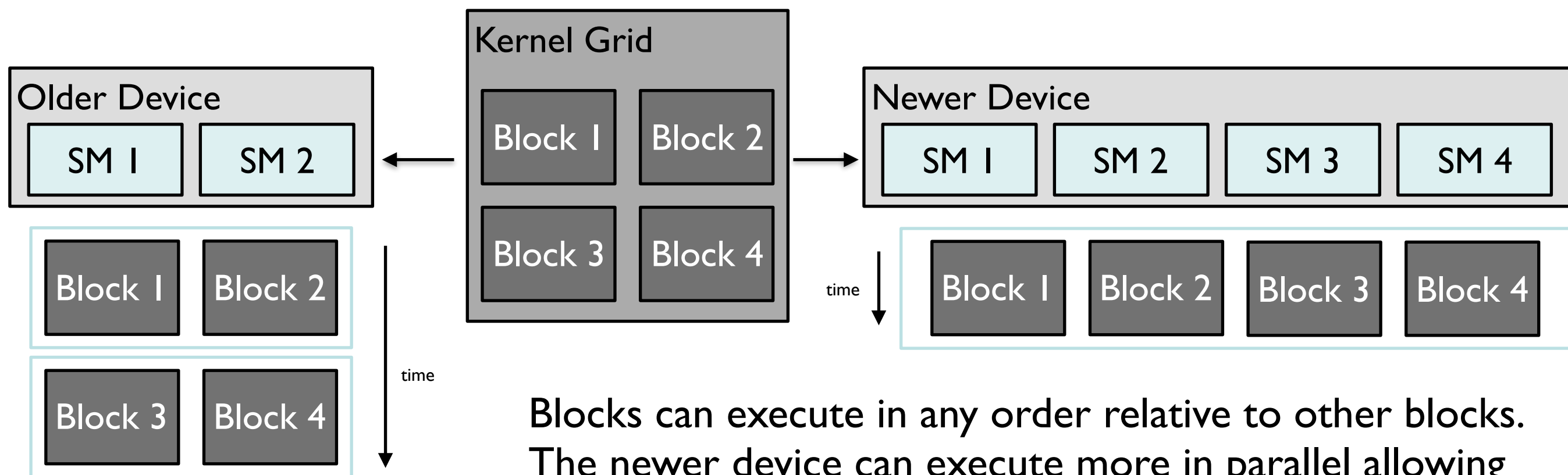
Will also block indefinitely, because half the threads will be waiting on the one call, while the other threads will be waiting on the other. Because of this, when you use `__syncthreads()`, *all* threads must execute the *same* `__syncthreads()` call. This means partial synchronization within a block is not possible.

Assigning Resources to Blocks

Block Synchronization

CUDA devices have the capability to process multiple blocks at the same time, however different devices can process different numbers of blocks simultaneously. As shown previously in the architecture diagrams, CUDA capable GPUs have different numbers of streaming multiprocessors (SMs); each of which can process a block at a time.

This is the main reason behind having `__syncthreads()` only synchronize threads within blocks. It also allows CUDA programs to have transparent scalability (assuming there are enough blocks within the grids).



Blocks can execute in any order relative to other blocks. The newer device can execute more in parallel allowing better performance.

Block Synchronization

It is important to know that Streaming Multiprocessors (SMs) can also each process multiple blocks.

Querying Device Properties

Synchronization and Transparent Scalability

In CUDA C there are built in function calls for determining the properties of the device(s) on the system:

```
int dev_count;
cudaGetDeviceCount( &dev_count );

cudaDeviceProp dev_prop;
for (int i = 0; i < dev_count; i++) {
    cudaGetDeviceProperties(&dev_prop, i);
    cout << "max threads per block: " << dev_prop.maxThreadsPerBlock << endl;
    cout << "max block x dim: " << dev_prop.maxThreadsDim[0] << endl;
    cout << "max block y dim: " << dev_prop.maxThreadsDim[1] << endl;
    cout << "max block z dim: " << dev_prop.maxThreadsDim[2] << endl;
    cout << "max grid x dim: " << dev_prop.maxGridSize[0] << endl;
    cout << "max grid y dim: " << dev_prop.maxGridSize[1] << endl;
    cout << "max grid z dim: " << dev_prop.maxGridSize[2] << endl;
}
```

An extensive example of this can be found in the CUDA SDK:
/GPU Computing/C/src/deviceQuery/deviceQuery.cpp

Thread Scheduling and Latency Tolerance

Synchronization and Transparent Scalability

In most implementations of CUDA capable GPUs to date, once a block is assigned to a SM it also then divided into 32-thread units called *warps*. (In general, this means it's probably a good idea to have the number of threads in your blocks be a multiple of 32, or whatever the warp size happens to be). `dev_prop.warpSize` can give you this value.

Synchronization and Transparent Scalability

CUDA schedules threads via these warps. Warps are executed SIMD (single instruction, multiple data) style, similar to a vector processor, across all the threads in the currently running warp.

When threads in the warp block on reach a *long-latency operation* (like a read from global memory) then the SM will execute other warps until the data for that operation is ready. This strategy is called *latency tolerance* or *latency hiding* and is used by CPUs scheduling multiple threads as well.

Synchronization and Transparent Scalability

Swapping between warps generally does not introduce any idle time into the execution timeline, because CUDA uses a *zero-overhead thread scheduling strategy*.

Generally, if there are enough warps in an SM, the time of long-latency operations can be masked by other warps being scheduled while those occur.

GPUs use this warp scheduling strategy as opposed to having more cache memory and branch prediction mechanisms (like in CPUs) to dedicate more chip area to floating point execution resources and parallelism.

Conclusions

Conclusions

Generally, a grid will be divided into blocks, the blocks will be assigned to different streaming multiprocessors, each of which will schedule warps of threads to their streaming processors.

Warps are run in SMs in a block by block basis (allowing for `__syncthreads()` to work).

While each GPU can have a different warp size, number of blocks that can be processed by a SM, and different numbers of SMs; CUDA transparently handles scheduling all the specified threads to the available hardware to allow the same code to execute on devices of different scales.