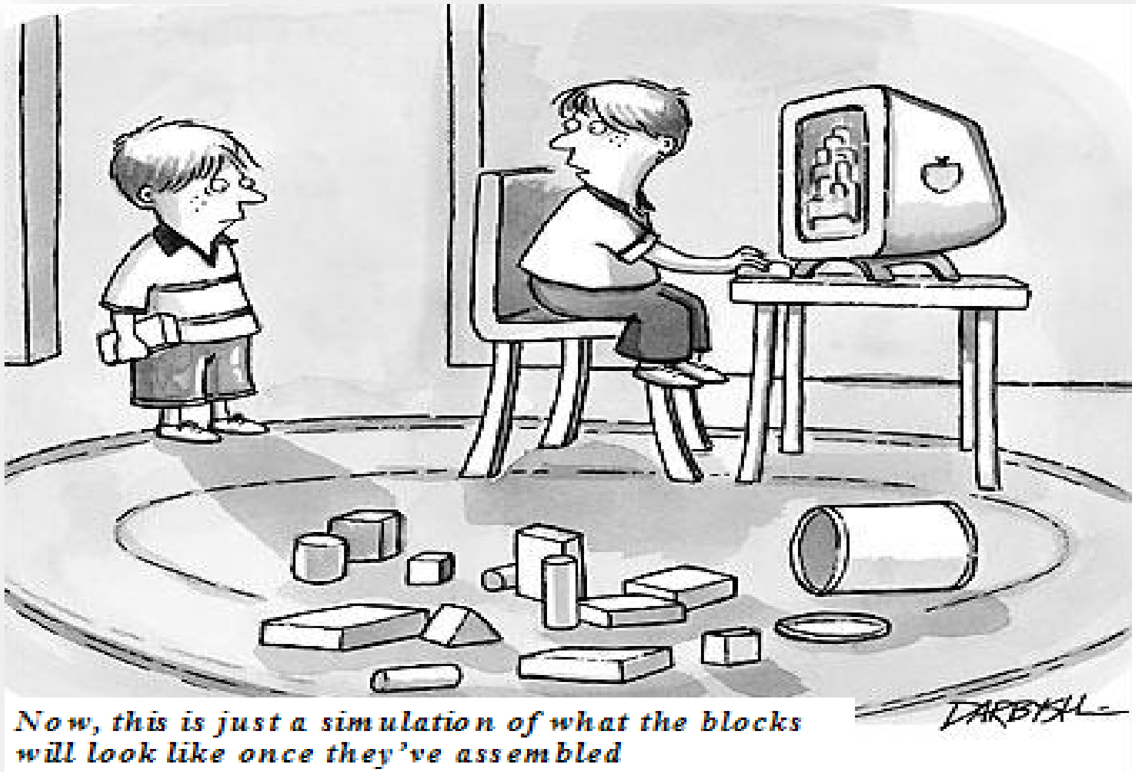
ECE569
Module 8



- Vector addition kernel code

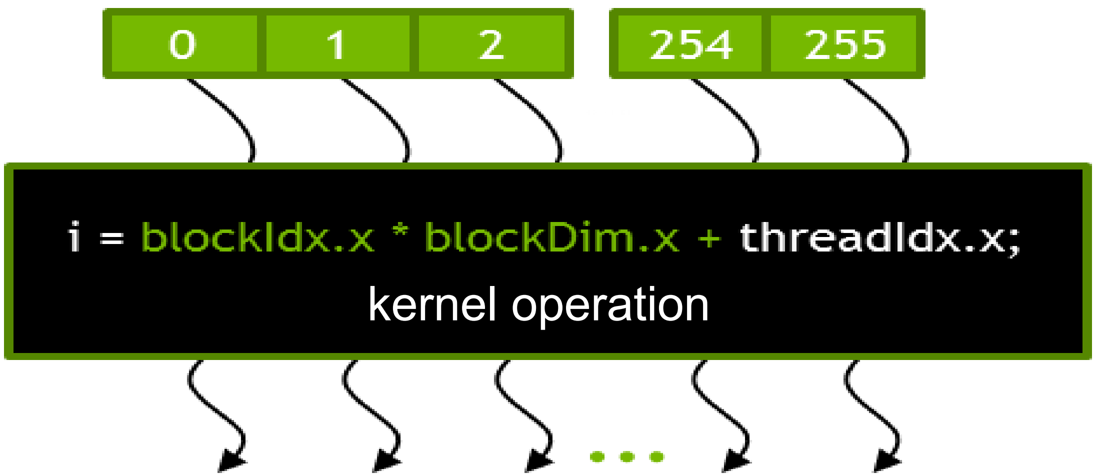
Kernel

- **Looks like serial program**
  - Write your program as if it will run on one thread
    - C[i] = A[i] + B[i]
- **From the CPU we decide the number of instances for the kernel**
  - GPU will run the program on many threads
- **GPU is good at**
  - Launching large number of parallel threads
    - Latency for a thread may be longer than CPU!

Arrays of Parallel Threads

- **A CUDA kernel is executed by a grid (array) of threads**
  - All threads in a grid run the same kernel code (Single Program Multiple Data)
  - Each thread has indexes that it uses to compute memory addresses and make control decisions

Example with 1 block,
With 256 threads per blocks
  blockIdx =?
  blockDim=?



Assume array size is 2048 targeting Tesla K20

	Compute Capability											
Technical Specifications	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.0	7.5
Maximum number of resident grids per device (Concurrent Kernel Execution)	16	4	32				16	128	32	16	128	
Maximum dimensionality of grid of thread blocks	3											
Maximum x- or y-dimension of a block	1024											
Maximum z-dimension of a block	64											
Maximum number of threads per block	1024											
Warp size	32											
Maximum number of resident blocks per multiprocessor	16				32							16
Maximum number of resident warps per multiprocessor	64											32
Maximum number of resident threads per multiprocessor	2048											1024
Maximum amount of shared memory per multiprocessor	48 KB			112 KB	64 KB	96 KB	64 KB		96 KB	64 KB	96 KB	64 KB
Maximum amount of shared memory per thread block	48 KB										96 KB	64 KB
Cache working set per multiprocessor for texture memory	Between 12 KB and 48 KB							Between 24 KB and 48 KB			32 ~ 128 KB	32 or 64 KB

What is the proper thread block configuration?

a) 1 block, 2048 threads per block

b) 2 blocks, 1024 threads per block

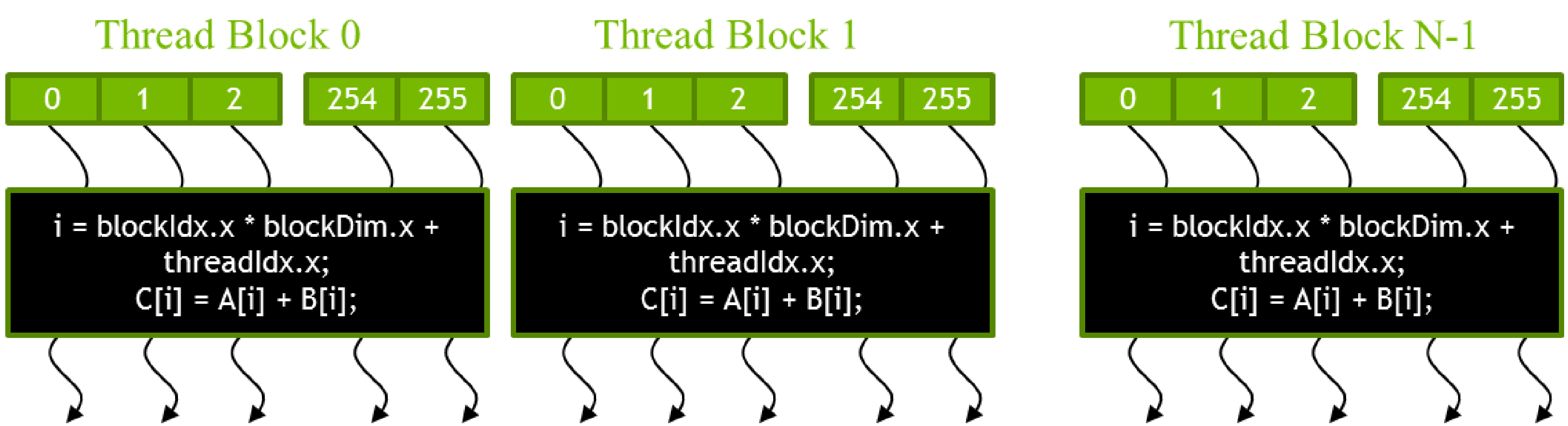
c) 4 blocks, 512 threads per block

d) 8 blocks, 256 threads per block

Tesla K20	3.5

Thread Blocks: Scalable Cooperation

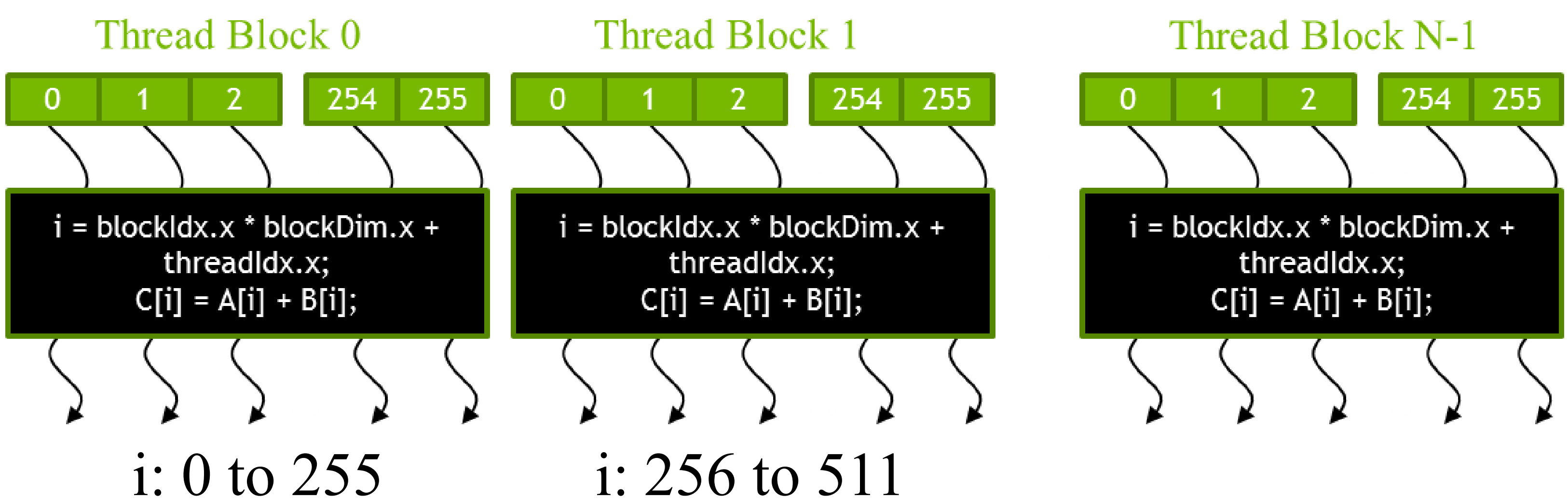
- **Divide thread array into multiple blocks**
  - Threads within a block cooperate via shared memory and barrier synchronization
  - Threads in different blocks do not interact



i: **a** to **b** i: **c** to **d** What are the values of a,b,c,d?

Thread Blocks: Scalable Cooperation

- **Divide thread array into multiple blocks**
  - Threads within a block cooperate via shared memory and barrier synchronization
  - Threads in different blocks do not interact
    - **blockDim.x allows striding with size of 256 (block size)**



Vector addition kernel

// Compute vector sum C = A+B

// Each thread performs one pair-wise addition

```
__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
      int i = threadIdx.x + blockDim.x * blockIdx.x;
       C[i] = A[i] + B[i]
}
```

CUDA Function Declarations

__global__ defines a kernel function,

– Must return void

	Executed on the:	Only callable from the:
`__device__ float myDeviceFunc()`	device	device
`__global__ void myKernelFunc()`	device	host
`__host__ float myHostFunc()`	host	host

- By default, all functions in a CUDA program are host functions if they do not have any of the CUDA keywords in their declaration.

CUDA Function Declarations

- one can use **both** "__host__" and "__device__" in a function declaration.
  - to generate **<u>two versions</u>** of object files for the same function.
    - One is executed on the host and the on the device
  - Many user library functions will likely fall into this category.

Launching the kernel from host

```
// Assume array size 1000 and block size is 256 threads
vecAddKernel<<<???, ???>>>(d_A, d_B, d_C, n);
```

- set the grid and thread block dimensions

Launching the kernel from host

```
// Run ceil(n/256) blocks of 256 threads each
vecAddKernel<<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);
```

- first parameter: **number of thread blocks** in the grid.
  - To ensure that we have enough threads to cover all the vector elements, we apply the C ceiling function to n/256.0
  - Using floating point value 256.0 ensures that we generate a floating value for the division so that ceiling function rounds up correctly.
- second parameter: **number of threads** per block.
- What is the total number of threads that we launch if n is 1000?

Launching the kernel from host

```
// Run ceil(n/256) blocks of 256 threads each
vecAddKernel<<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);
```

- first parameter: **number of thread blocks** in the grid.
  - To ensure that we have enough threads to cover all the vector elements, we apply the C ceiling function to n/256.0
  - Using floating point value 256.0 ensures that we generate a floating value for the division so that ceiling function rounds up correctly.
- second parameter: **number of threads** per block.
- if n is 1000
  - launch ceil(1000/256.0) = 4 thread blocks.
  - statement will launch 4*256=1024 threads.
- In this case which elements of the A and B arrays will be accessed by thread (threadidx.x = 232, blockIdx.x =3)?

Launching the kernel from host

```
// Run ceil(n/256) blocks of 256 threads each
vecAddKernel<<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);
```

- if n is 1000
  - launch ceil(1000/256.0) = 4 thread blocks.
  - statement will launch 4*256=1024 threads.

- In this case which elements of the A and B arrays will be accessed by thread (threadidx.x = 232, blockIdx.x =3)?

```
i = threadIdx.x + blockDim.x * blockIdx.x; // i=1000
C[i] = A[i] + B[i]   // out of boundary!!!
```

Threads 1000-1023 should not participate in the computation! **How do we achieve that?**

Vector addition kernel

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
      int i = threadIdx.x + blockDim.x * blockIdx.x;
      if (i<n)
            C[i] = A[i] + B[i];
}
```

Observations:

1. No more loop in the code! Replaced with grid of threads
2. With (i<n) we can operate on arbitrary length arrays

CUDA threads

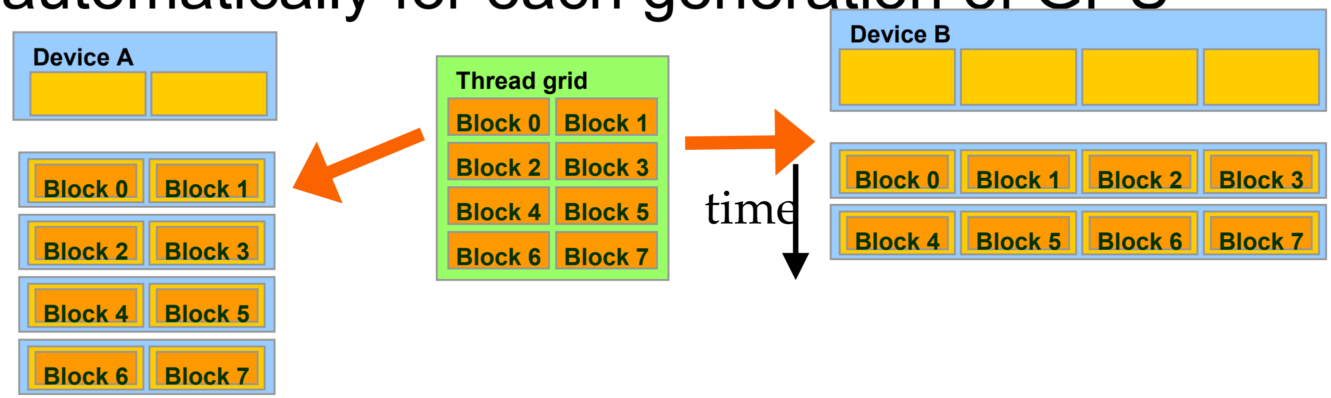
- **main mechanism for exploiting of data parallelism**
  - Hierarchical thread organization
  - Launching parallel execution
  - Thread index to data index mapping

Threads and Blocks: Why an hierarchy?

- **Why not launch millions of threads instead of organizing them into blocks?**
  – Each block can have 1024 threads, but you can launch massive number of blocks 2^{32}-1
  – Each GPU can run some number of blocks concurrently, executing some number of threads simultaneously
  – With the extra level of abstraction, higher performance GPUs can simply run more blocks concurrently and chew through the workload quicker **with absolutely no change to the code!**
    - **Scalability!**

Transparent Scalability

- **Each block can execute in any order relative to others.**
- **Hardware is free to assign blocks to any processor at any time**
  - A kernel scales to any number of parallel processors automatically for each generation of GPU

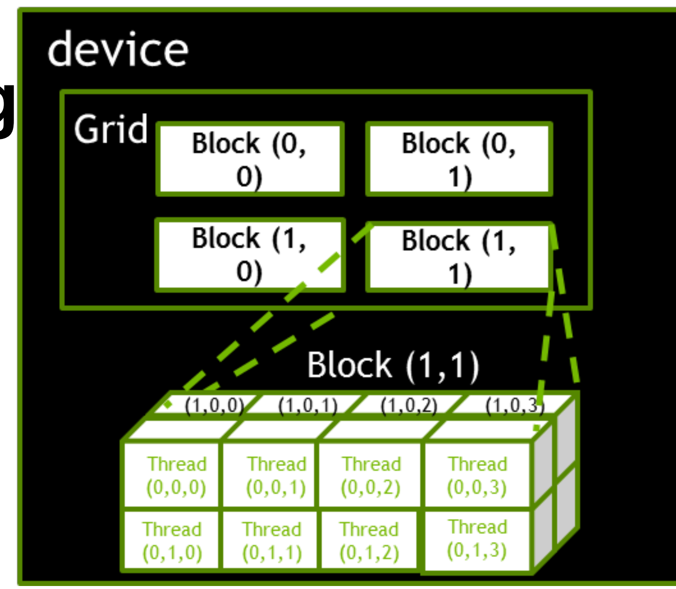


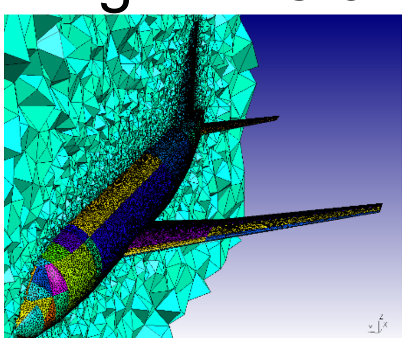
Reading

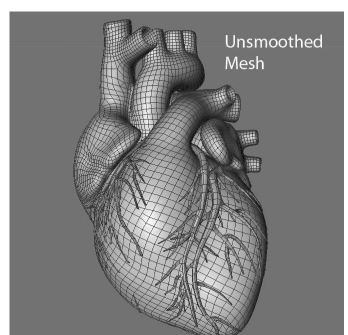
- CUDA Programming Guide
  - Chapter 1: Introduction
  - Chapter 2: Programming Model
  - Appendix A: CUDA-enabled GPUs
  - Appendix B, sections B.1 – B.4
  - Appendix I, section I.1: features of different GPUs

blockIdx and threadIdx

- **Each thread uses indices to decide what data to work on**
  - blockIdx: 1D, 2D, or 3D
  - threadIdx: 1D, 2D, or 3D
- **Simplifies memory addressing when processing multidimensional data**
  - Image processing
  - Solving PDEs on volumes
  - …







Compute Capability

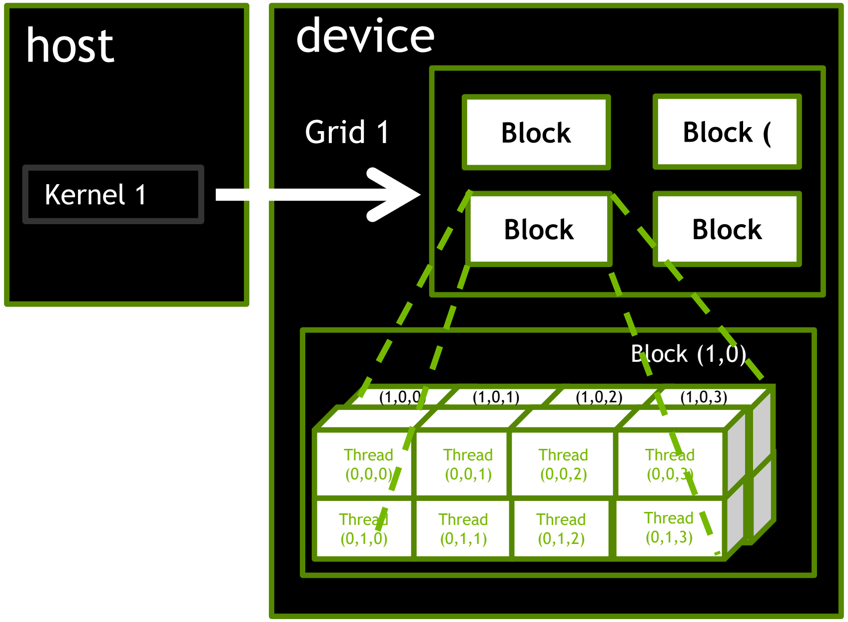
<table>
<tr><th></th><th colspan="12">Compute Capability</th></tr>
<tr><td>Technical Specifications</td><td>3.0</td><td>3.2</td><td>3.5</td><td>3.7</td><td>5.0</td><td>5.2</td><td>5.3</td><td>6.0</td><td>6.1</td><td>6.2</td><td>7.0</td><td>7.5</td></tr>
<tr><td>Maximum number of resident grids per device (Concurrent Kernel Execution)</td><td>16</td><td>4</td><td colspan="4">32</td><td>16</td><td>128</td><td>32</td><td>16</td><td colspan="2">128</td></tr>
<tr><td>Maximum dimensionality of grid of thread blocks</td><td colspan="12">3</td></tr>
<tr><td>Maximum x- or y-dimension of a block</td><td colspan="12">1024</td></tr>
<tr><td>Maximum z-dimension of a block</td><td colspan="12">64</td></tr>
<tr><td>Maximum number of threads per block</td><td colspan="12">1024</td></tr>
<tr><td>Warp size</td><td colspan="12">32</td></tr>
<tr><td>Maximum number of resident blocks per multiprocessor</td><td colspan="4">16</td><td colspan="7">32</td><td>16</td></tr>
<tr><td>Maximum number of resident warps per multiprocessor</td><td colspan="11">64</td><td>32</td></tr>
<tr><td>Maximum number of resident threads per multiprocessor</td><td colspan="11">2048</td><td>1024</td></tr>
<tr><td>Maximum amount of shared memory per multiprocessor</td><td colspan="3">48 KB</td><td>112 KB</td><td>64 KB</td><td>96 KB</td><td colspan="2">64 KB</td><td>96 KB</td><td>64 KB</td><td>96 KB</td><td>64 KB</td></tr>
<tr><td>Maximum amount of shared memory per thread block</td><td colspan="10">48 KB</td><td>96 KB</td><td>64 KB</td></tr>
<tr><td>Cache working set per multiprocessor for texture memory</td><td colspan="7">Between 12 KB and 48 KB</td><td colspan="3">Between 24 KB and 48 KB</td><td>32 ~ 128 KB</td><td>32 or 64 KB</td></tr>
</table>

dim3 Data Type

- **3D structure (C `struct`) or vector type with three unsigned integers (x,y,z)**
  - dim3 threads(256); // x=256, y and z=1
  - dim3 blocks(100,100); // x and y = 100, z=1
  - dim3 another(10,20,40);
- **Each thread is individual and knows the following:**
  - threadIdx : Thread index within the block
  - blockIdx: Block index within the grid
  - blockDim: Block dimensions in threads
  - gridDim: Grid dimensions in blocks
    - Each of these are **dim3 structures** and can be read in the kernel to assign particular workloads to a thread.

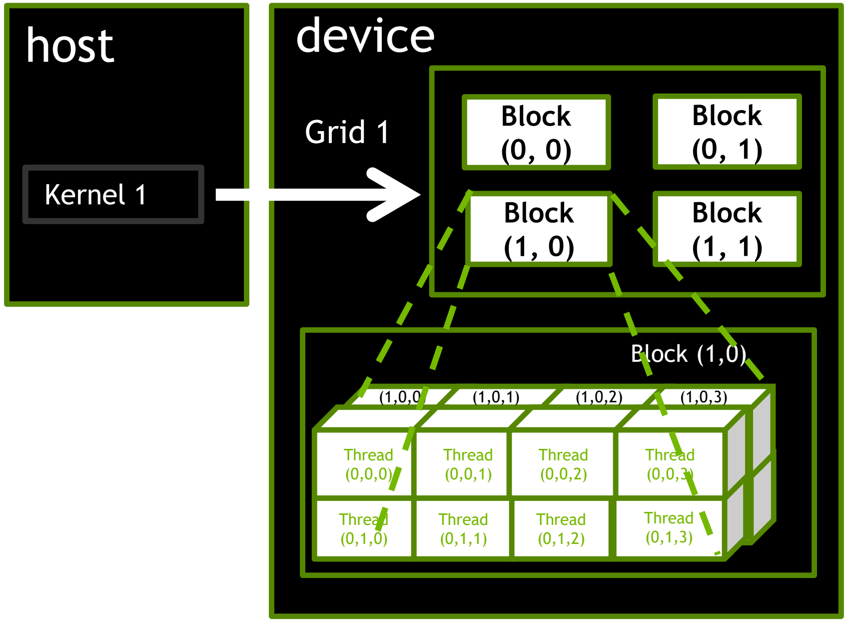
A Multi-Dimensional Grid Example

- **dim3 dimGrid(?,?,?)**
  - ? blocks
- **dim3 dimBlock(?,?,?)**
  - ? threads/block



A Multi-Dimensional Grid Example

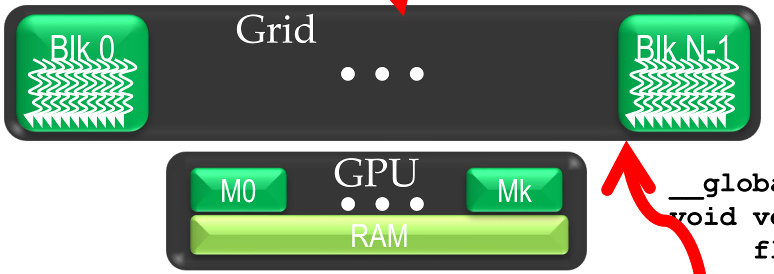
- **dim3 dimGrid(2,2,1)**
  - 4 blocks
- **dim3 dimBlock(4,2,2)**
  - 16 threads/block



CUDA Parallelism Model – Multidimensional

- **Vector Addition Kernel Launch (Host Code)**

```
__host__ void vecAdd(float* h_A, float* h_B, float*
  h_C, int n)
{
   dim3 DimGrid((n-1)/256 + 1, 1, 1);
   dim3 DimBlock(256, 1, 1);
   vecAddKernel<<<DimGrid,DimBlock>>>(d_A, d_B, d_C, n);
}
```




```
__global__
void vecAddKernel(float *A,
     float *B, float *C, int n)
{
   int i = blockIdx.x * blockDim.x
             + threadIdx.x;

   if( i<n ) C[i] = A[i]+B[i];
}
```

Next

- **Launching interactive jobs**
  - Printing from kernel code, time utility, thread block configuration experiment
- **Debugging and profiling tools**