



ECE408 /CS483/CSE408 Fall 2021

Applied Parallel Programming

Lecture 2:

Introduction to CUDA C and Data Parallel Programming

Course Reminders

- Lab 0 is due this Friday at 8pm US Central time (8pm Saturday for ZJ1/ZJ2 sections)
 - You should have submitted it by now :), it is an easy lab.
 - But if you signed up for the course in the last day or so, submit the lab ASAP.
 - If you miss this deadline, that's OK for Lab 0, but you must submit it ASAP. Remember, Lab 0 will be graded, but not counted towards your overall grade.
- Lab 1 will be out very soon, it is due next Friday

Objective

- To learn the basic concept of data parallel computing
- To learn the basic features of the CUDA C programming interface

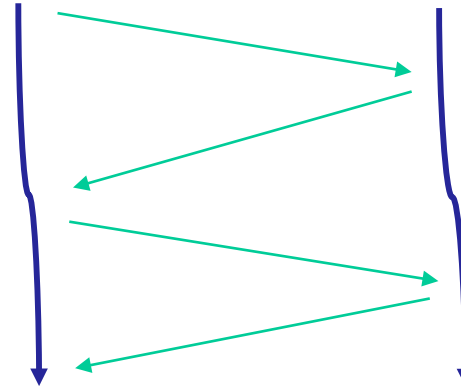
Thread as a basic unit of computing

- What is a thread?

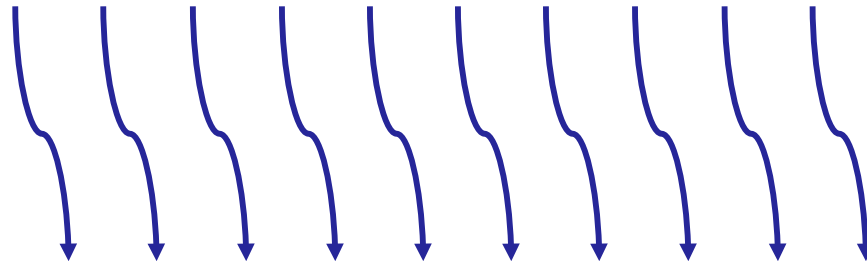
- Program
- PC
- Context
 - Memory
 - Registers
 - ...



- Multiple threads



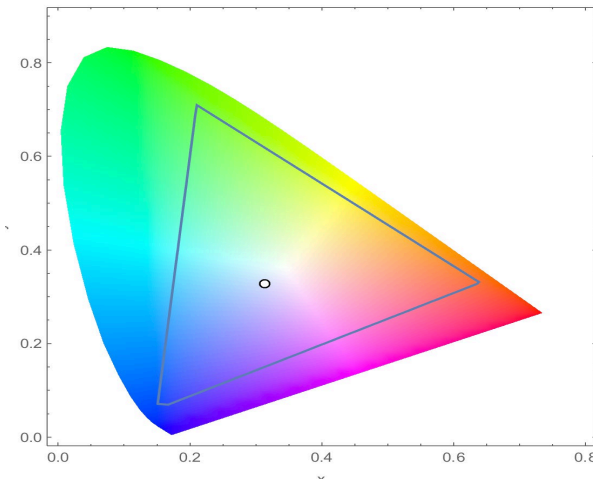
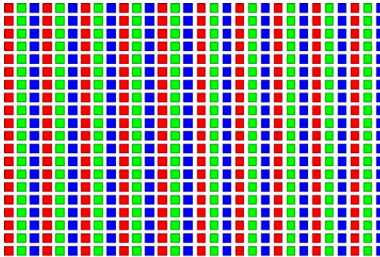
- Many threads



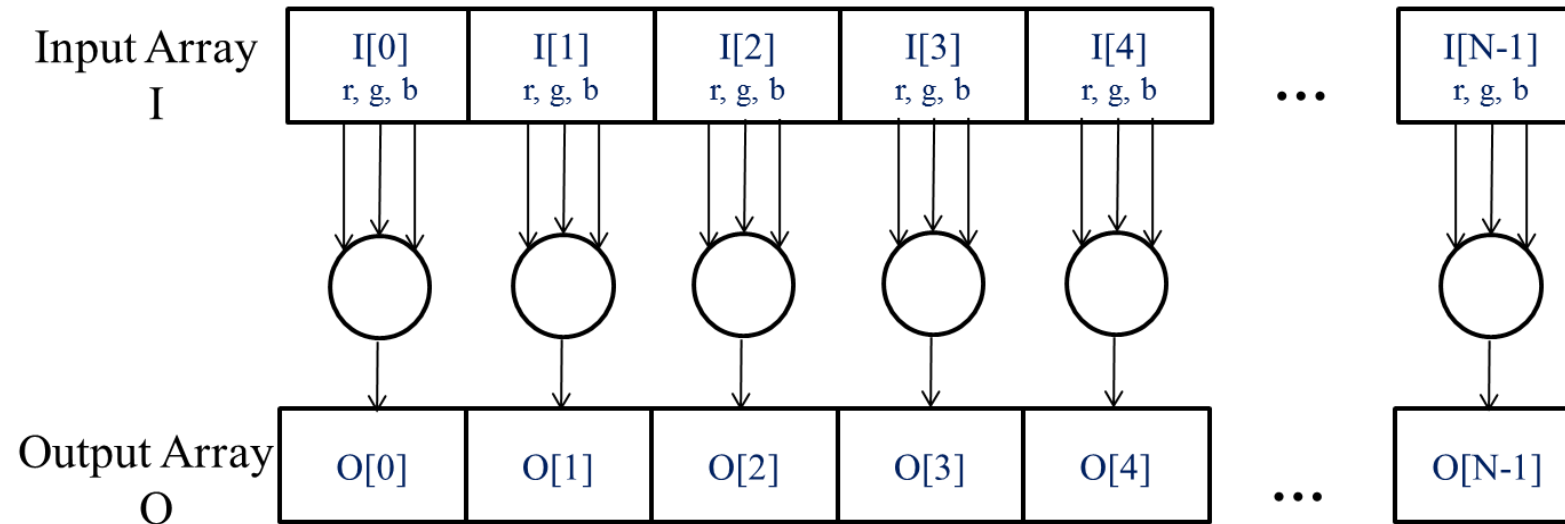
A Data Parallel Computation Example: Conversion of a color image to grey-scale image



```
for each pixel {  
    pixel = gsConvert(pixel)  
}  
// Every pixel is independent  
// of every other pixel
```



The pixels can be calculated independently of each other



```
for each pixel {  
    pixel = gsConvert(pixel)  
}  
// Every pixel is independent  
// of every other pixel
```

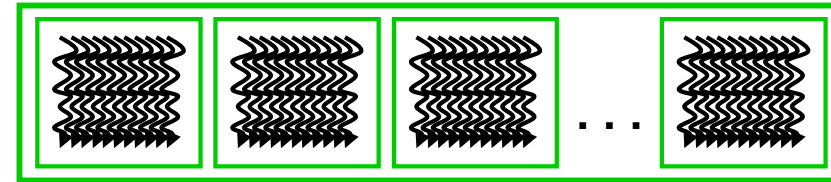
CUDA/OpenCL – Execution Model

- Integrated host+device app C program
 - Serial or modestly parallel parts in **host** C code
 - Highly parallel parts in **device** SPMD kernel C code

Serial Code (host)

Parallel Kernel (device)

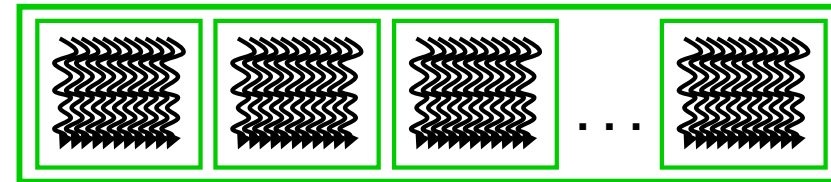
`KernelA<<< nBlk, nTid >>>(args);`



Serial Code (host)

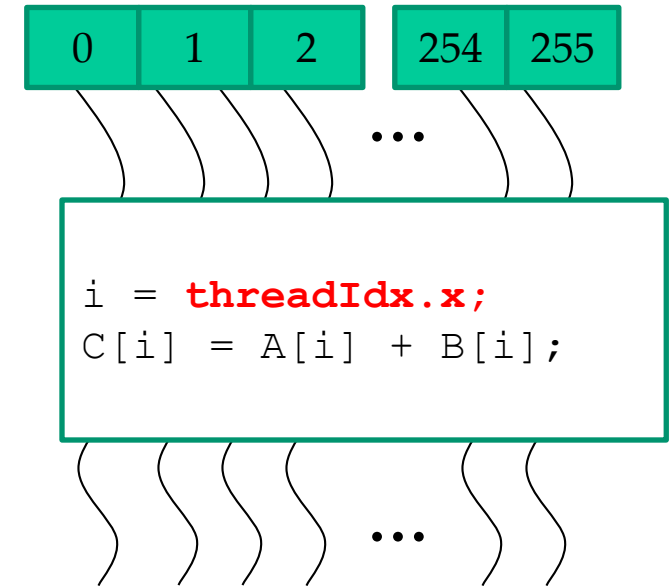
Parallel Kernel (device)

`KernelB<<< nBlk, nTid >>>(args);`



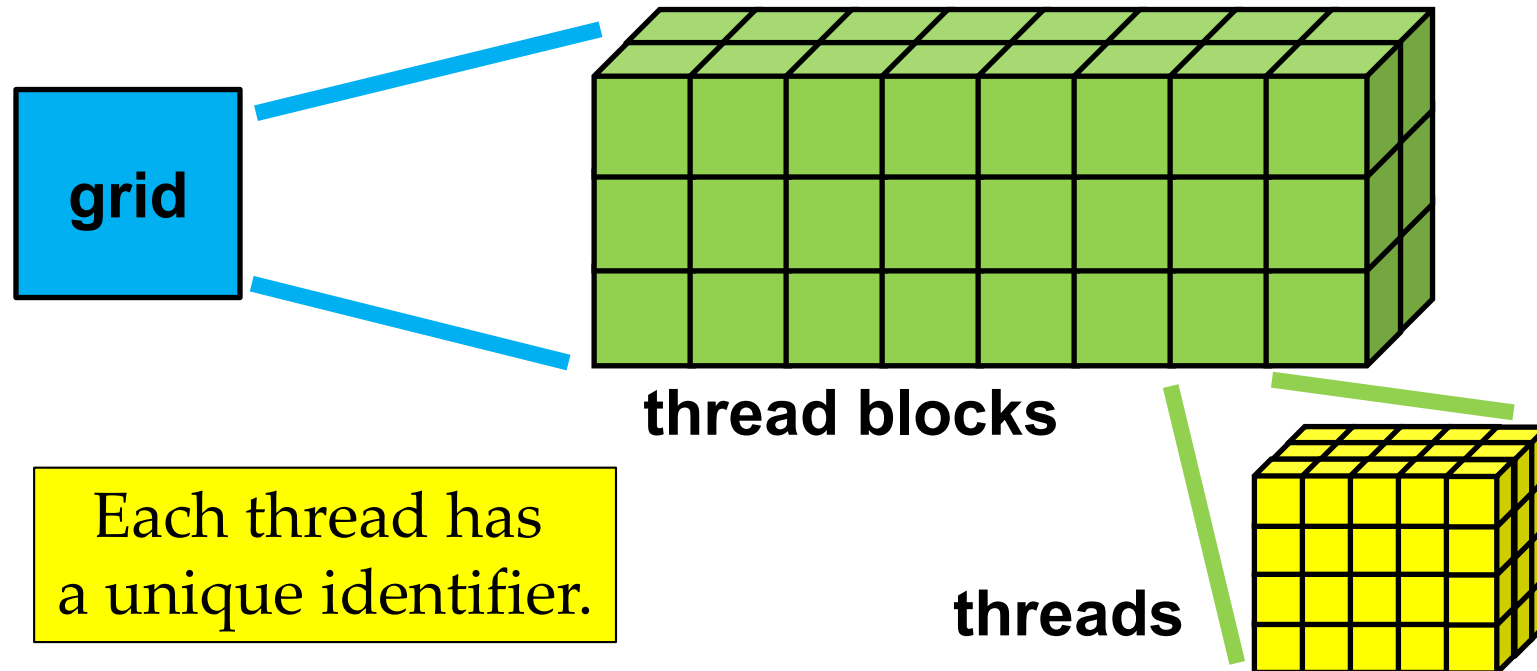
Arrays of Parallel Threads

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



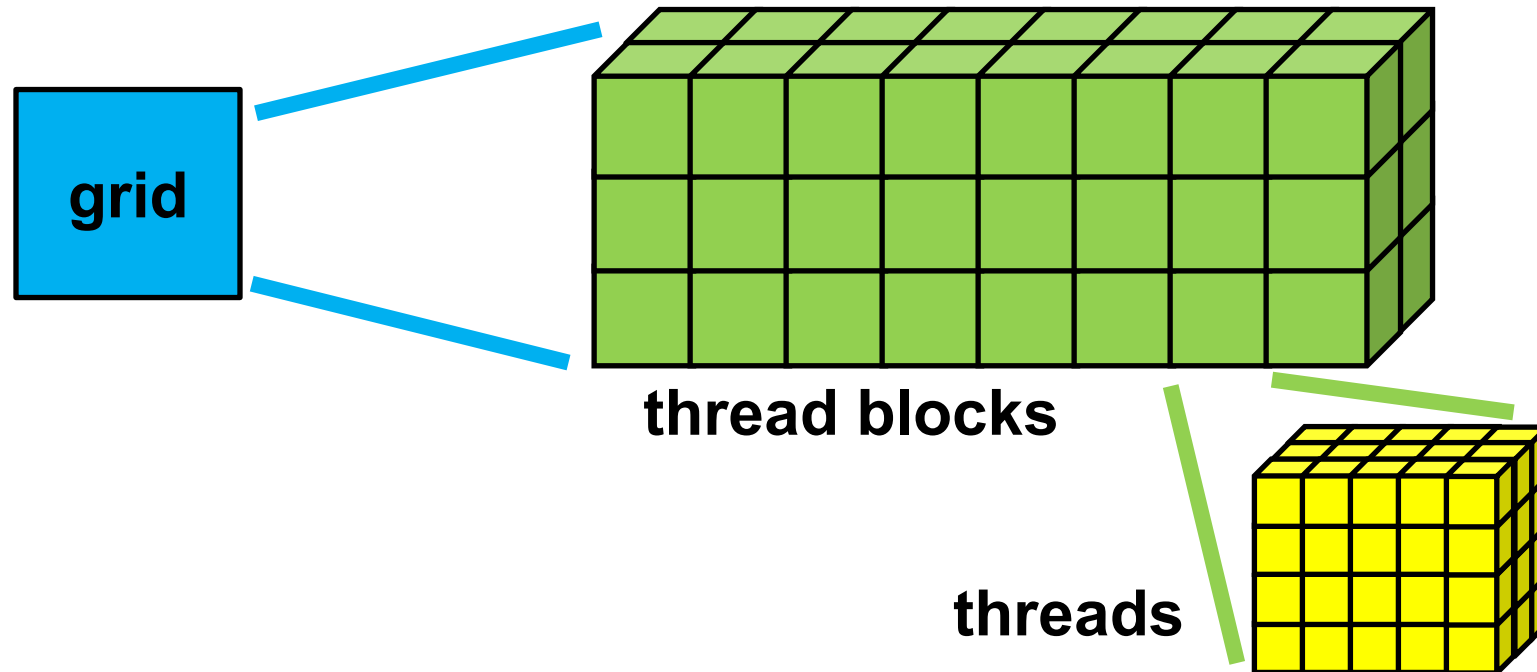
Logical Execution Model for CUDA

- Each CUDA kernel
 - is executed by a **grid**,
 - a 3D array of **thread blocks**, which are
 - 3D arrays of **threads**.



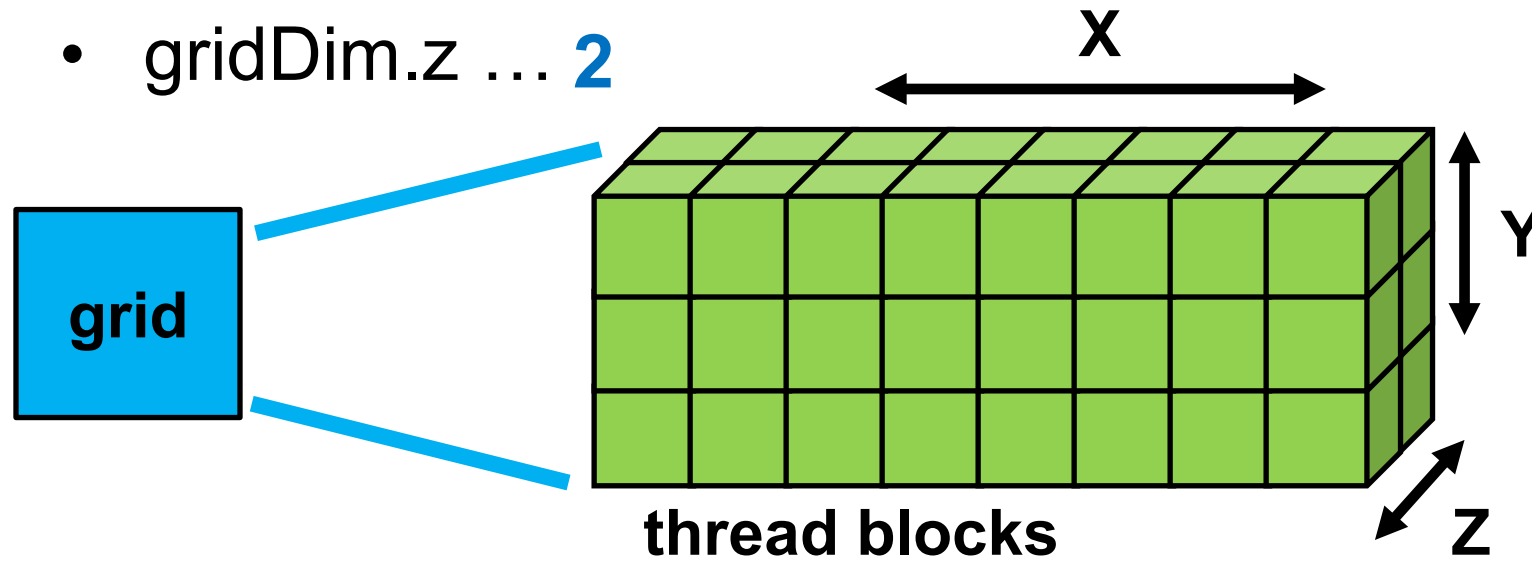
Single Program, Multiple Data

- Each thread
 - executes the **same program**
 - on **distinct data inputs**,
 - a single-program, multiple-data (**SPMD**) model



gridDim Gives Number of Blocks

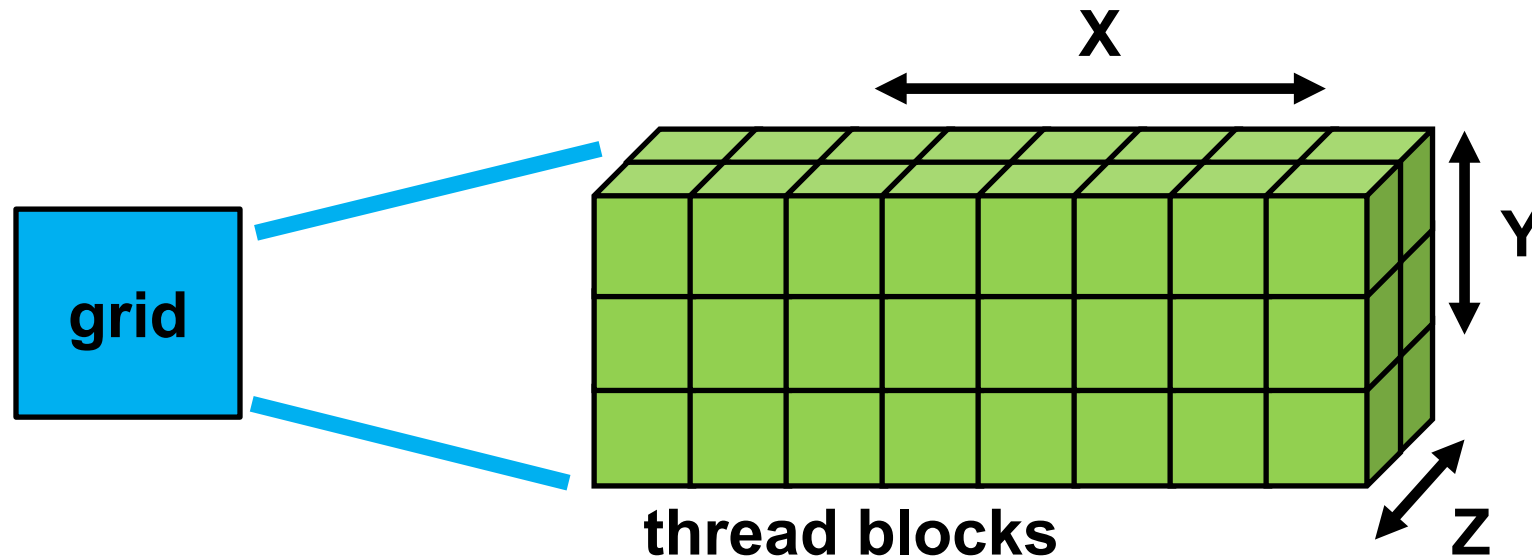
- Number of blocks in each dimension is
 - `gridDim.x` ... **8**
 - `gridDim.y` ... **3**
 - `gridDim.z` ... **2**



For 2D (and 1D grids), simply use
grid dimension 1 for Z (and Y).

blockIdx is Unique for Each Block

- Each block has a unique index tuple
 - blockIdx.x (from 0 to (gridDim.x – 1))
 - blockIdx.y (from 0 to (gridDim.y – 1))
 - blockIdx.z (from 0 to (gridDim.z – 1))

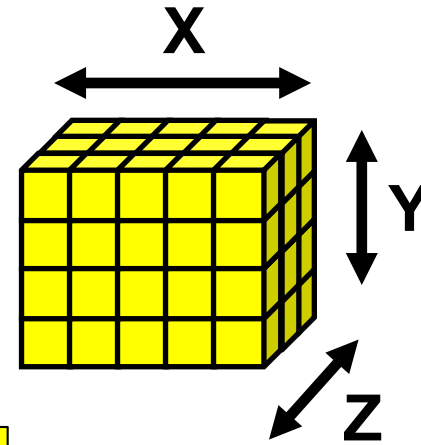


blockDim: # of Threads per Block

- Number of blocks in each dimension is

- blockDim.x ... **5**
- blockDim.y ... **4**
- blockDim.z ... **3**

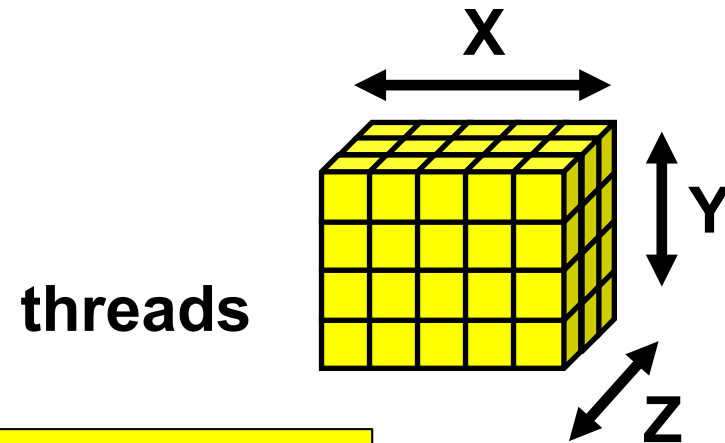
threads



For 2D (and 1D blocks), simply use block dimension 1 for Z (and Y).

threadIdx Unique for Each Thread

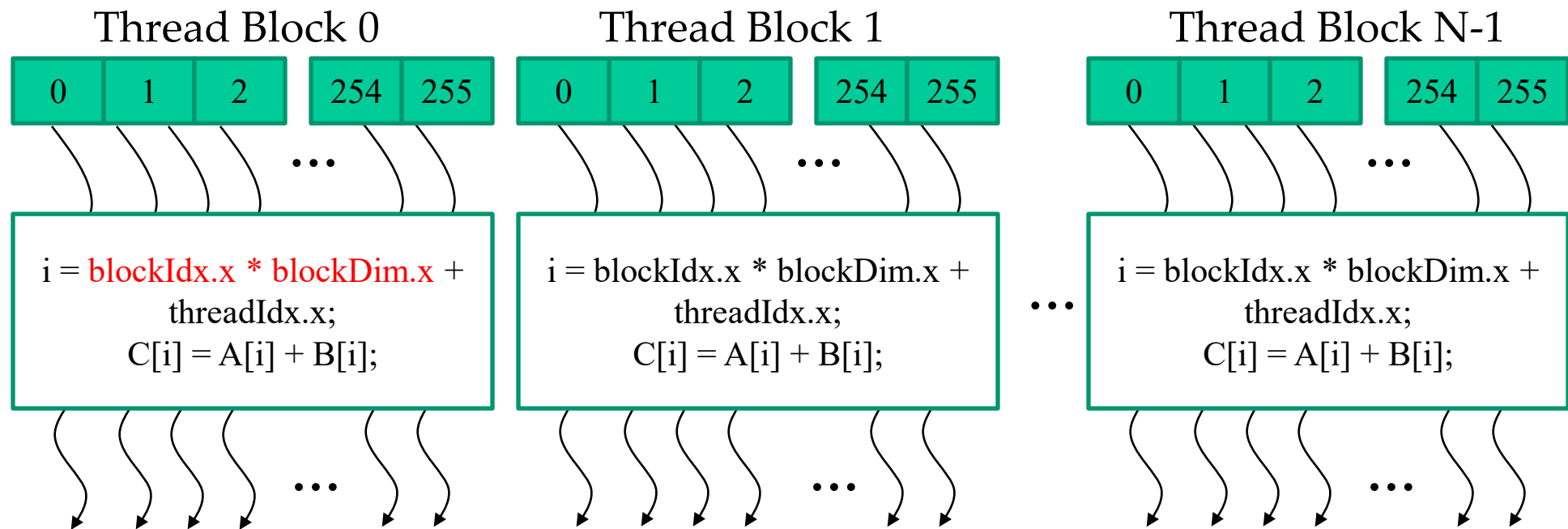
- Each thread has a unique index tuple
 - threadIdx.x (from 0 to (blockDim.x – 1))
 - threadIdx.y (from 0 to (blockDim.y – 1))
 - threadIdx.z (from 0 to (blockDim.z – 1))



threadIdx tuple is unique to each thread
WITHIN A BLOCK.

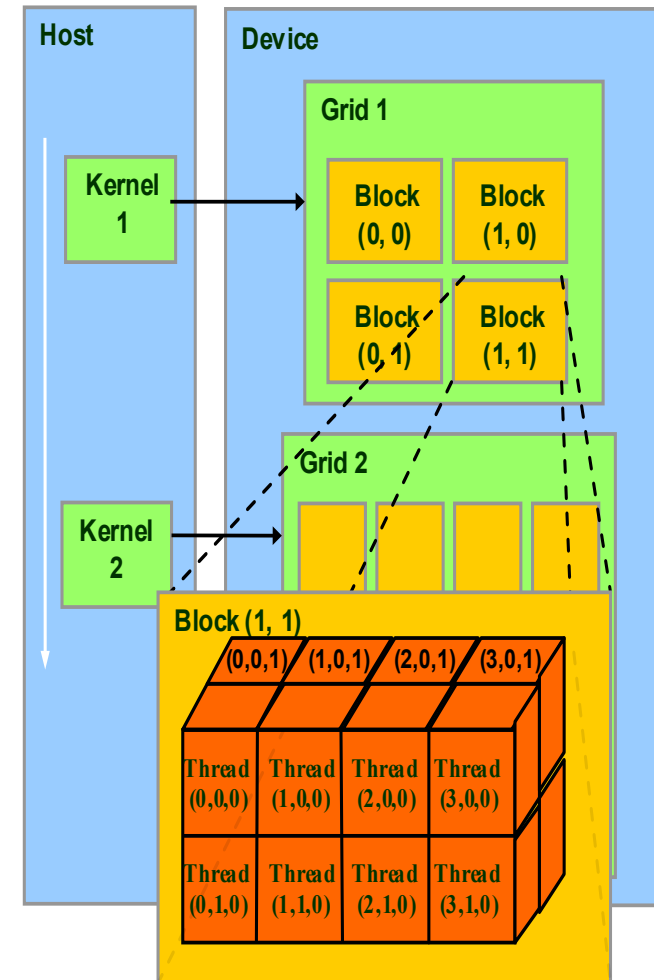
Thread Blocks: Scalable Cooperation

- Threads within a block cooperate via **shared memory, atomic operations** and **barrier synchronization** (to be covered later)
- Threads in different blocks cooperate less.

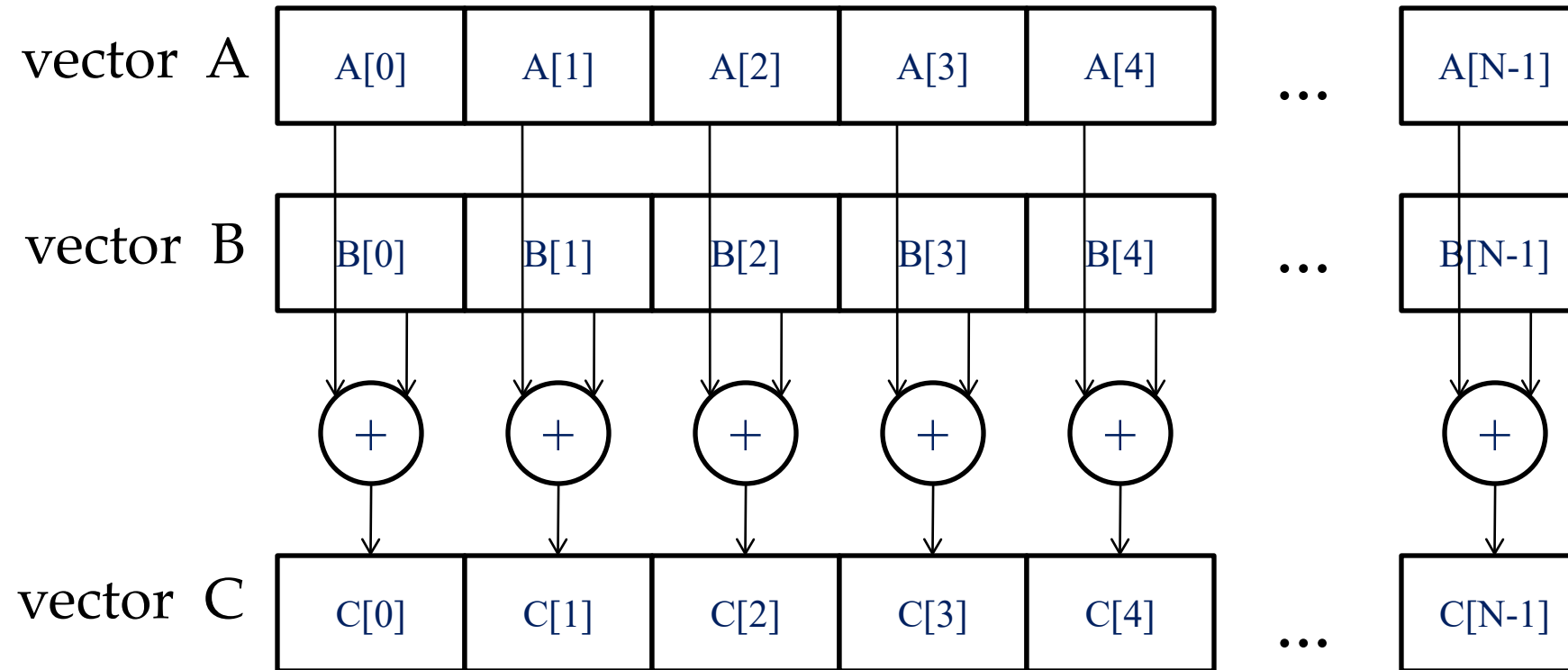


blockIdx and threadIdx

- Thread block and thread organization
 - simplifies memory addressing
 - when processing multidimensional data
- Image processing
- Vectors, matrices, tensors
- Solving PDEs on volumes
- ...



Vector Addition – Conceptual View



Vector Addition – Traditional C Code

```
// Compute vector sum C = A+B
void vecAdd(float* A, float* B, float* C, int n)
{
    for (i = 0, i < n, i++)
        C[i] = A[i] + B[i];
}

int main()
{
    // Memory allocation for A_h, B_h, and C_h
    // I/O to read A_h and B_h, N elements
    ...
    vecAdd(A_h, B_h, C_h, N);
}
```

Heterogeneous Computing: vecAdd Host Code

```
#include <cuda.h>
void vecAdd(float* A, float* B, float* C, int n)
{
    int size = n* sizeof(float);
    float *A_d, *B_d, *C_d;
    ...
1. // Allocate device memory for A, B, and C
   // copy A and B to device memory

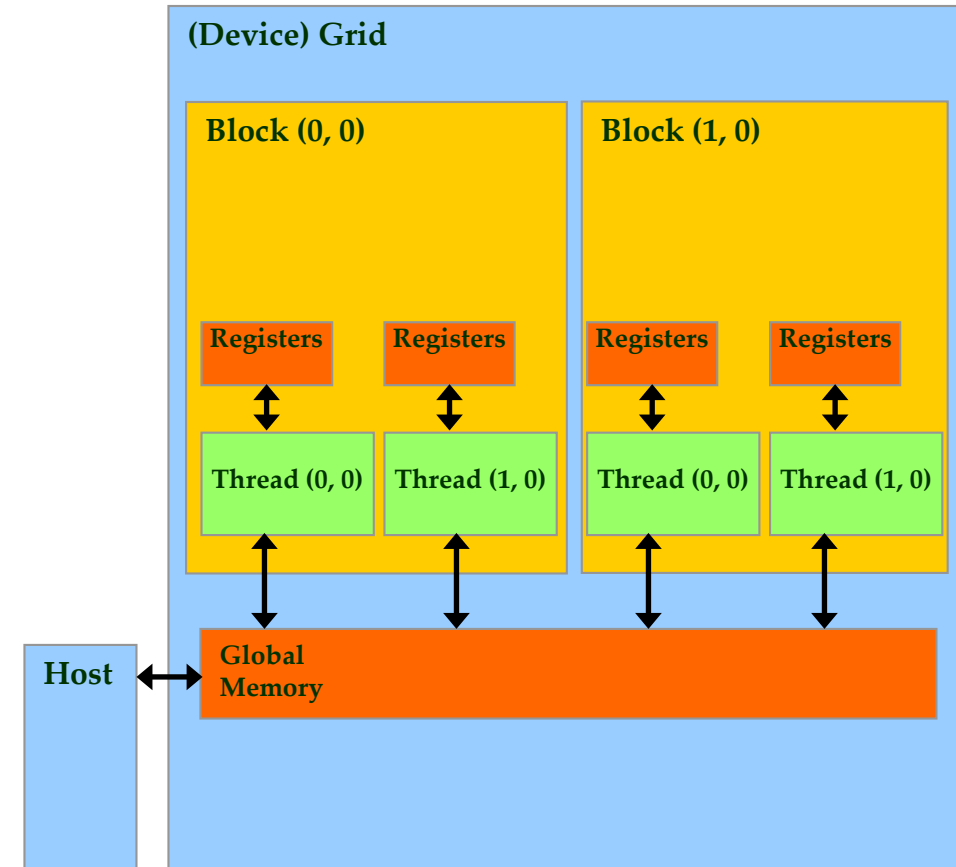
2. // Kernel launch code - to have the device
   // to perform the actual vector addition

3. // copy C from the device memory
   // Free device vectors
}
```

Partial Overview of CUDA Memories

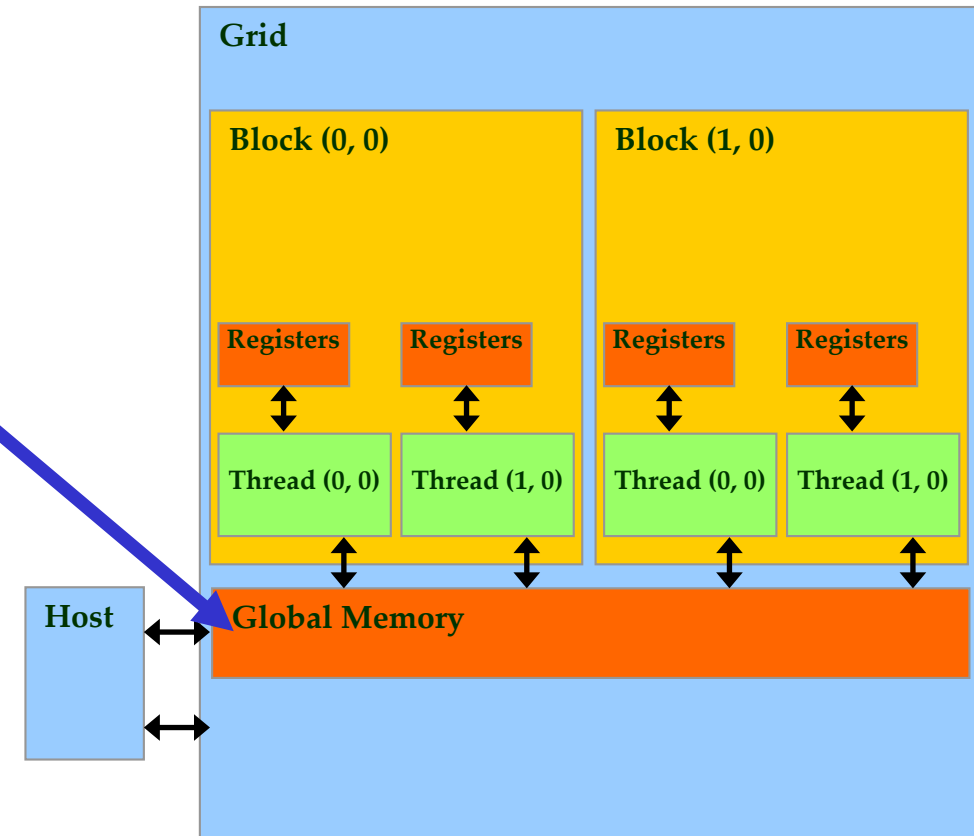
- Device code can:
 - R/W per-thread **registers**
 - R/W per-grid **global memory**
- Host code can
 - Transfer data to/from per grid **global memory**

We will cover more later.



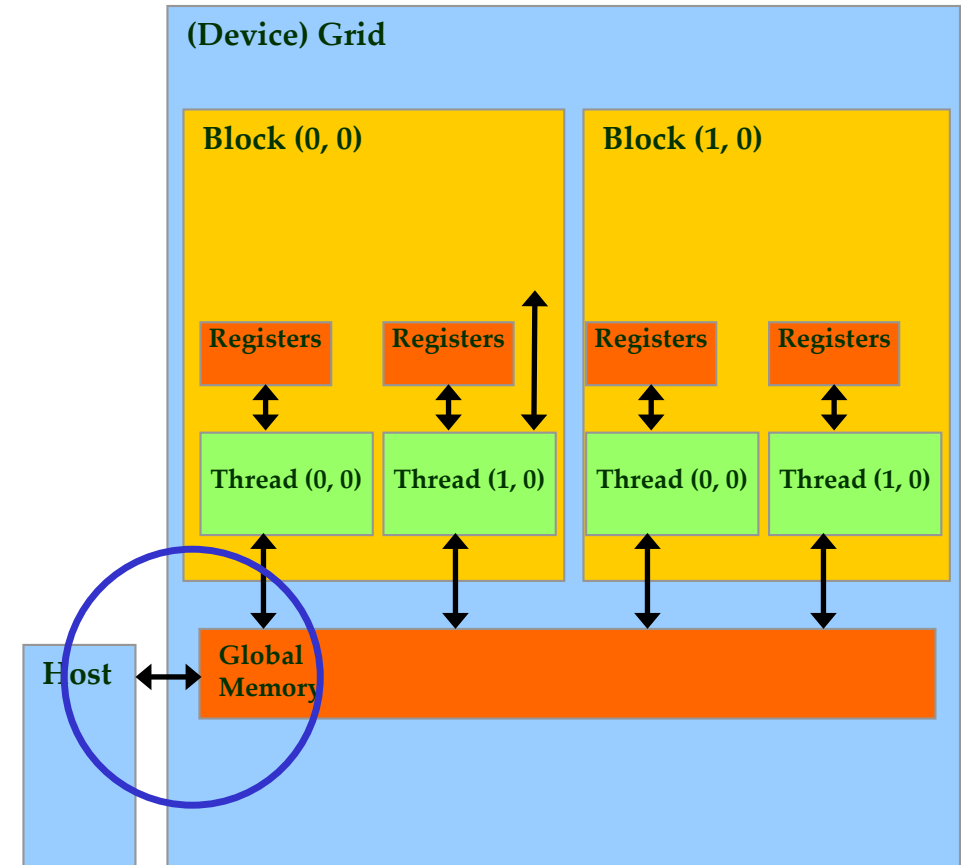
CUDA Device Memory Management API functions

- `cudaMalloc()`
 - Allocates object in the device global memory
 - Two parameters
 - **Address of a pointer** to the allocated object
 - **Size of** the allocated object in terms of bytes
- `cudaFree()`
 - Frees object from device global memory
 - **Pointer** to freed object



Host-Device Data Transfer API functions

- `cudaMemcpy()`
 - memory data transfer
 - Requires four parameters
 - Pointer to destination
 - Pointer to source
 - Number of bytes copied
 - Type/Direction of transfer



```

void vecAdd(float* A, float* B, float* C, int n)
{
    int size = n * sizeof(float);
    float *A_d, *B_d, *C_d;

1. // Transfer A and B to device memory
   // (error-checking omitted)
   cudaMalloc((void **) &A_d, size);
   cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
   cudaMalloc((void **) &B_d, size);
   cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);

   // Allocate device memory for
   cudaMalloc((void **) &C_d, size);

2. // Kernel invocation code - to be shown later
   ...
3. // Transfer C from device to host
   cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);
   // Free device memory for A, B, C
   cudaFree(A_d); cudaFree(B_d); cudaFree (C_d);
}

```

Example: Vector Addition Kernel

Device Code

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition

__global
void vecAddKernel(float* A_d, float* B_d, float* C_d, int n)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x ;
    if(i < n) C_d[i] = A_d[i] + B_d[i];
}

int vectAdd(float* A, float* B, float* C, int n)
{
    // A_d, B_d, C_d allocations and copies omitted
    // Run ceil(n/256) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256.0), 256>>>>(A_d, B_d, C_d, n);
}
```


Example: Vector Addition Kernel

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

Host Code

```
int vecAdd(float* A, float* B, float* C, int n)
{
    // A_d, B_d, C_d allocations and copies omitted
    // Run ceil(n/256) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256.0),256>>>(A_d, B_d, C_d, n);
}
```

More on Kernel Launch

Equivalent Host Code

```
int vecAdd(float* A, float* B, float* C, int n)
{
    // A_d, B_d, C_d allocations and copies omitted
    // Run ceil(n/256) blocks of 256 threads each
    dim3 DimGrid(n/256, 1, 1);
    if (0 != (n % 256)) { DimGrid.x++; }
    dim3 DimBlock(256, 1, 1);

    vecAddKernel<<<DimGrid,DimBlock>>>>(A_d, B_d, C_d, n);
}
```

- Any call to a kernel function is asynchronous from CUDA 1.0 on, explicit synch needed for blocking

Vector Addition Kernel

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

int vecAdd(float* A, float* B, float* C, int n)
{
    // A_d, B_d, C_d allocations and copies omitted
    // Run ceil(n/256) blocks of 256 threads each
    dim3 DimGrid(ceil(n/256), 1, 1);
    dim3 DimBlock(256, 1, 1);

    vecAddKernel<<<DimGrid, DimBlock>>>>(A_d, B_d, C_d, n);
}
```

A Number of blocks per dimension

B Number of threads per dimension in a block

C Unique block # in x dimension

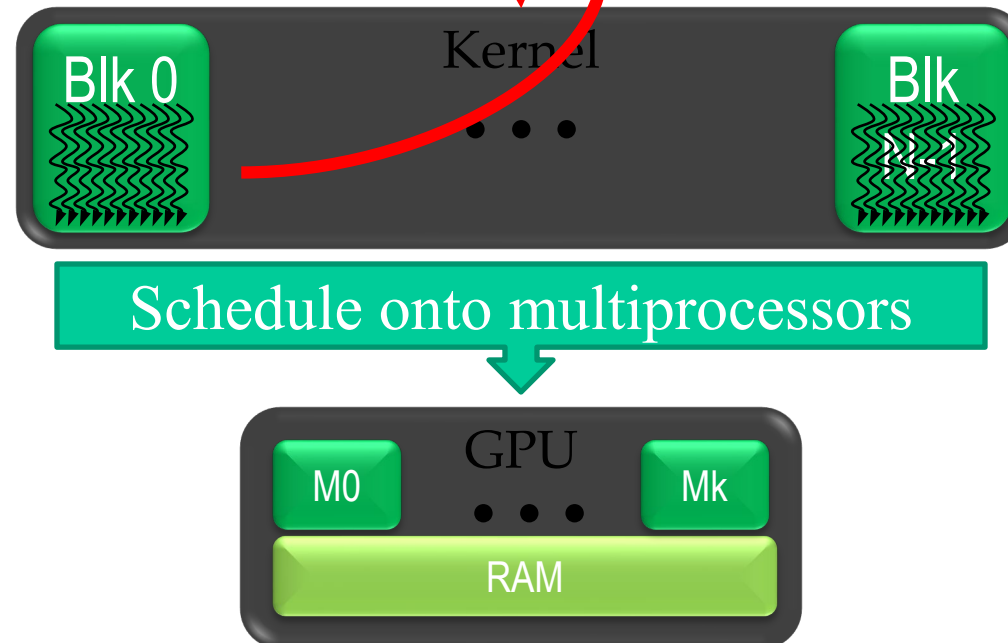
D Number of threads per block in x dimension

E Unique thread # in x dimension in the block

Kernel execution in a nutshell

```
__host__  
void vecAdd()  
{  
    dim3 DimGrid(ceil(n/256.0),1,1);  
    dim3 DimBlock(256,1,1);  
  
    vecAddKernel<<<DimGrid,DimBlock>>>  
    (A_d,B_d,C_d,n);  
}
```

```
__global__  
void vecAddKernel(float *A_d,  
                  float *B_d, float *C_d, int n)  
{  
    int i = blockIdx.x * blockDim.x  
           + threadIdx.x;  
  
    if( i < n ) C_d[i] = A_d[i] + B_d[i];  
}
```



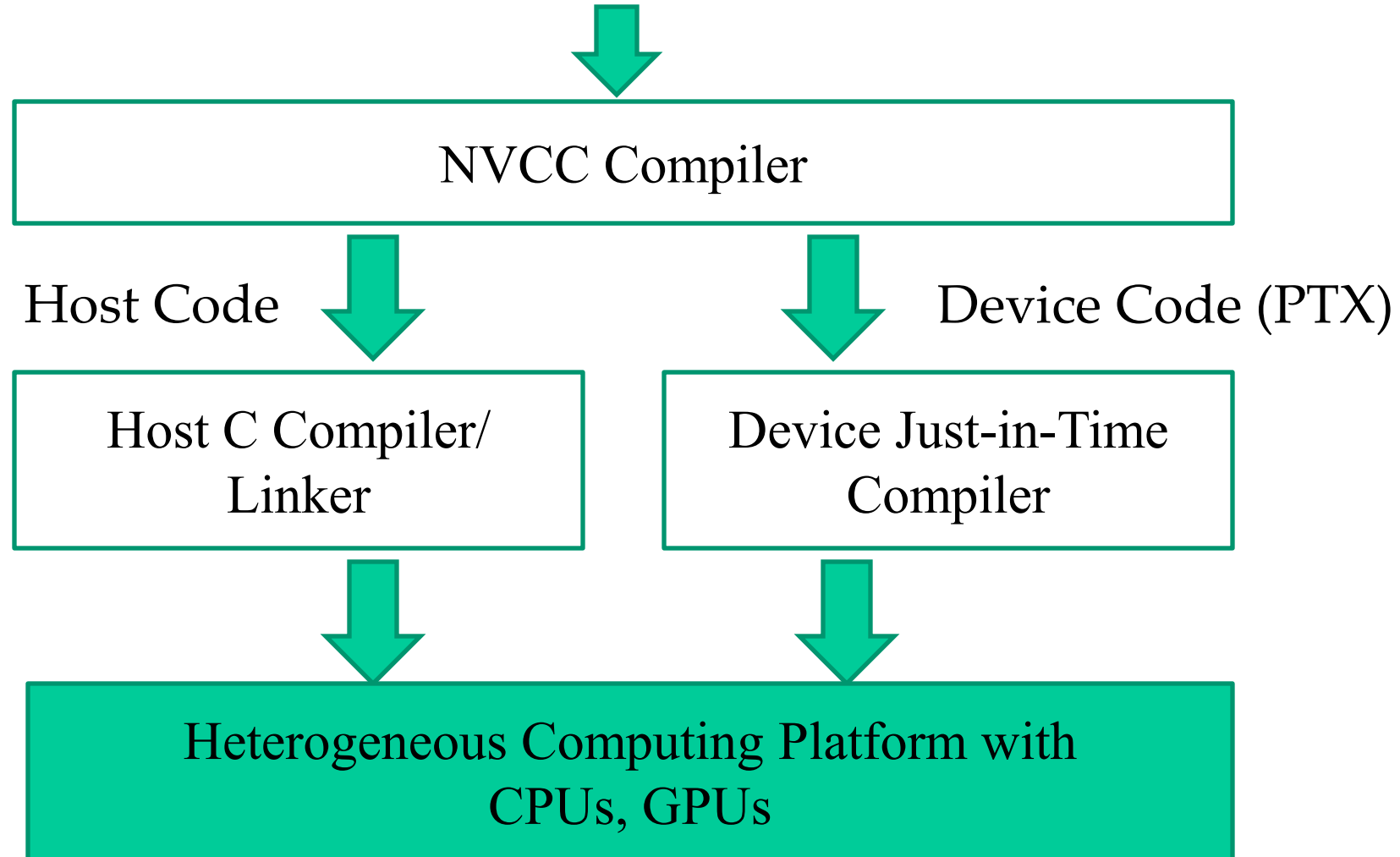
More on CUDA Function Declarations

	Executed on the:	Only callable from the:
<code>__device__ float DeviceFunc()</code>	device	device
<code>__global__ void KernelFunc()</code>	device	host
<code>__host__ float HostFunc()</code>	host	host

- `__global__` defines a kernel function
 - Each “__” consists of two underscore characters
 - A kernel function must return `void`
- `__device__` and `__host__` can be used together

Compiling A CUDA Program

Integrated C programs with CUDA extensions



Two vertical lines, one blue and one orange, are positioned on the left side of the slide.

ANY MORE QUESTIONS?
READ CHAPTER 2