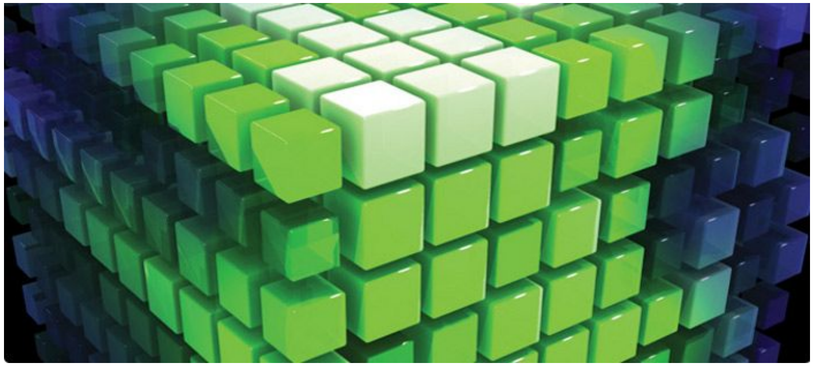
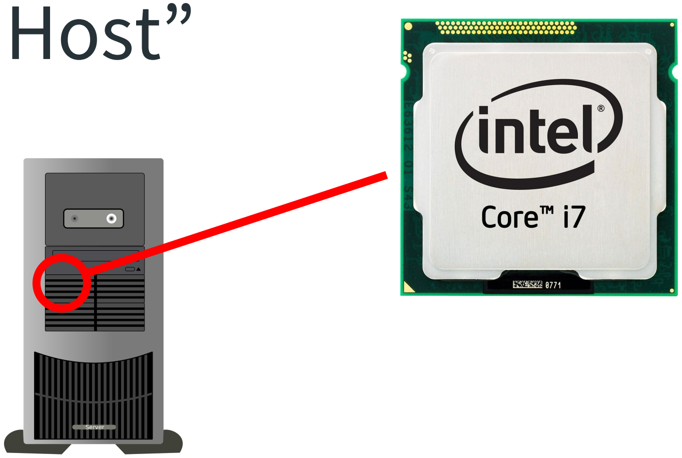
3. Programming in CUDA



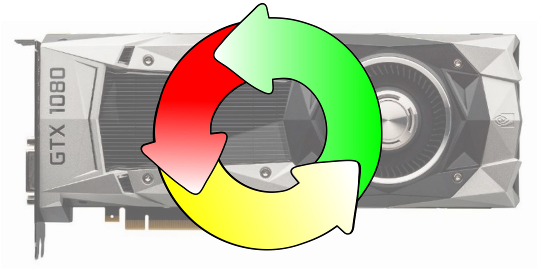
Words you should know:

“Host”



“Device”

“Kernel”

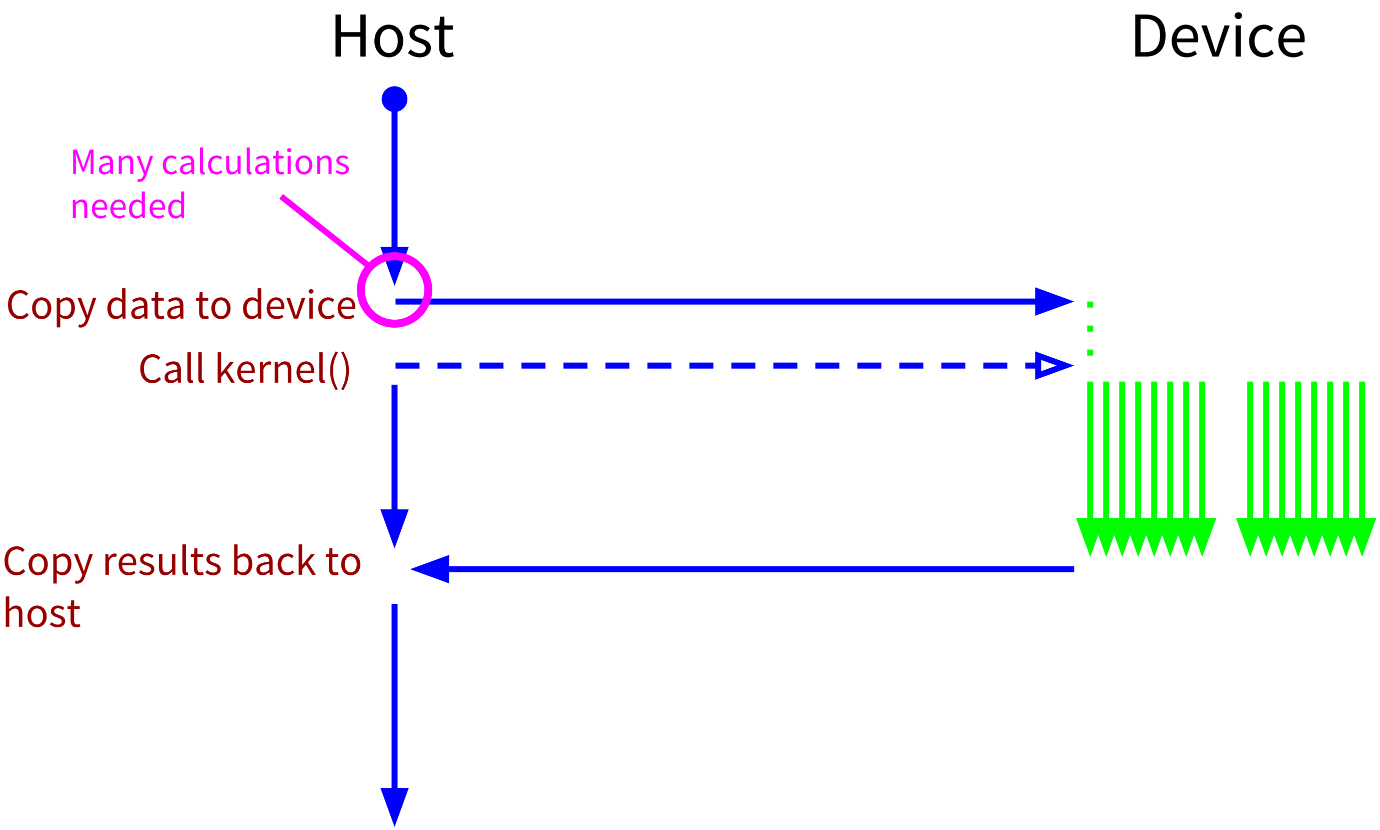


Introduction to CUDA

- CUDA Language
  - C/C++-like syntax with some minor extensions
  - we'll use the C flavour
- Recall: GPUs are *accelerators*
  - Can’t run a process on their own
  - Need the CPU's help!
- How it works:
  - We write a *Host program*
  - Periodically call kernel functions to *hand- off* compute-intensive tasks to the GPU

NVIDIA CUDA

Anatomy of a CUDA Program

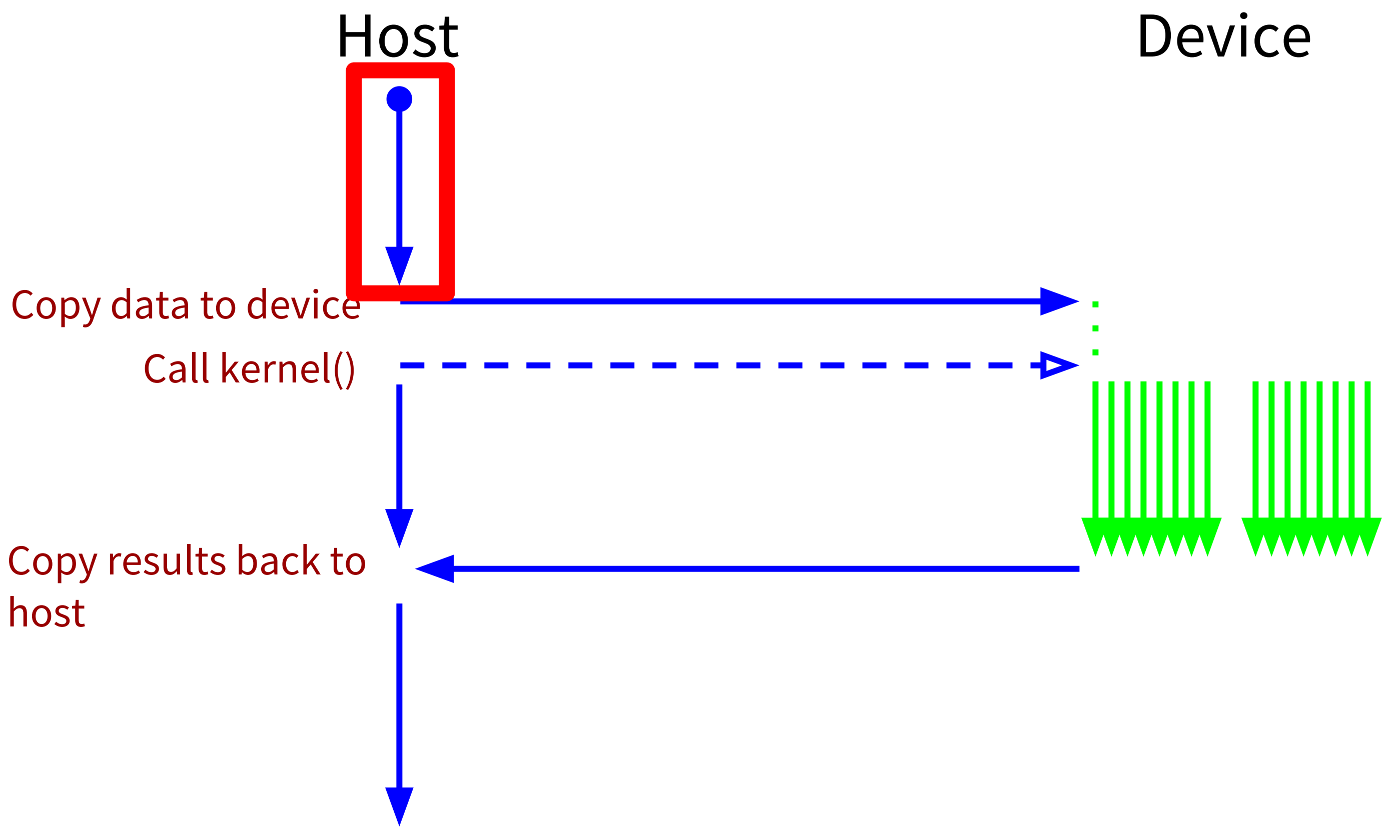


This cycle can be repeated for each data-parallel task in our algorithm.

Example - Vector Addition

A:	2	8	5	9	7	1
B: +	7	2	6	1	4	8
C:	9	10	11	10	11	9

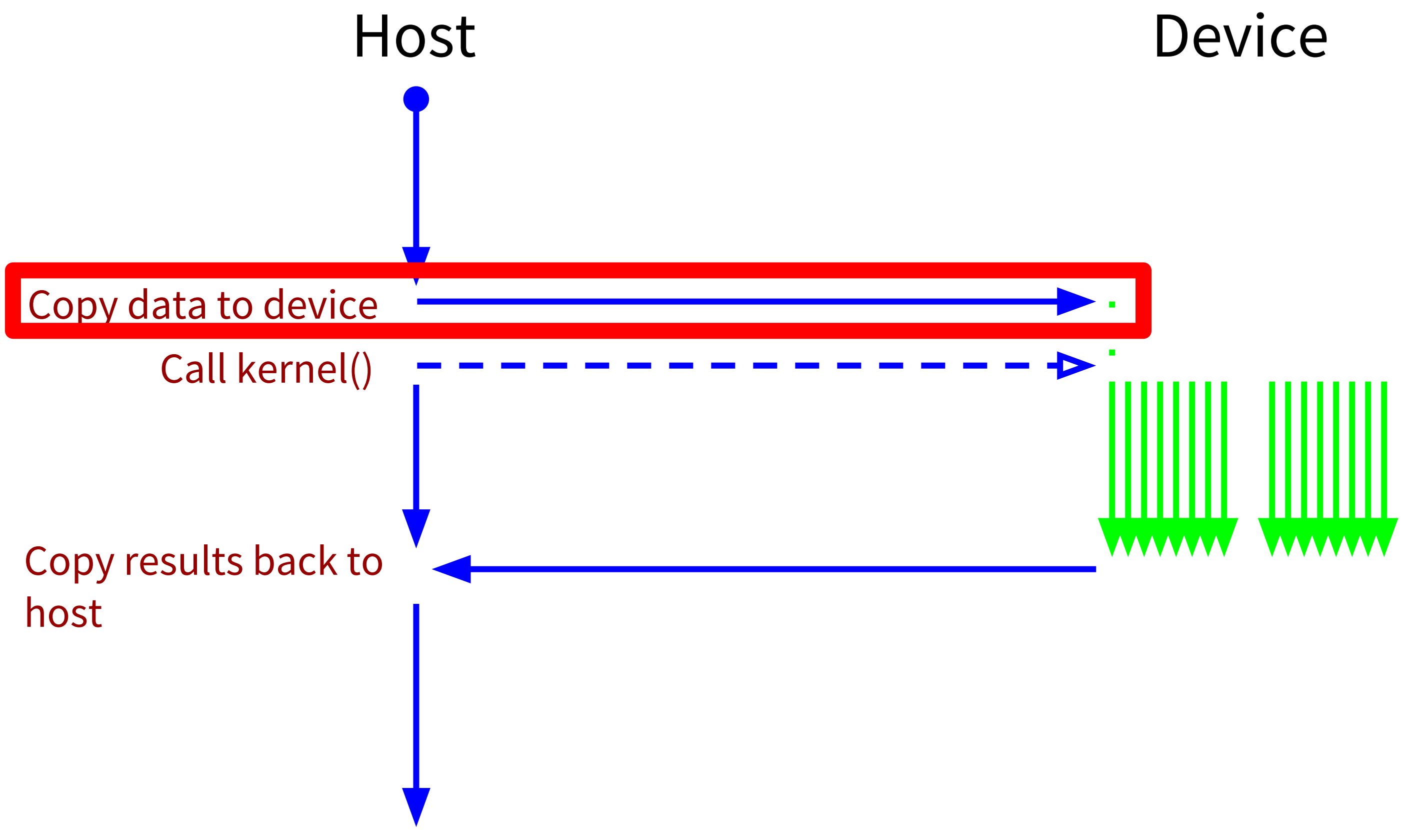
1. Starting Out



Starting Out

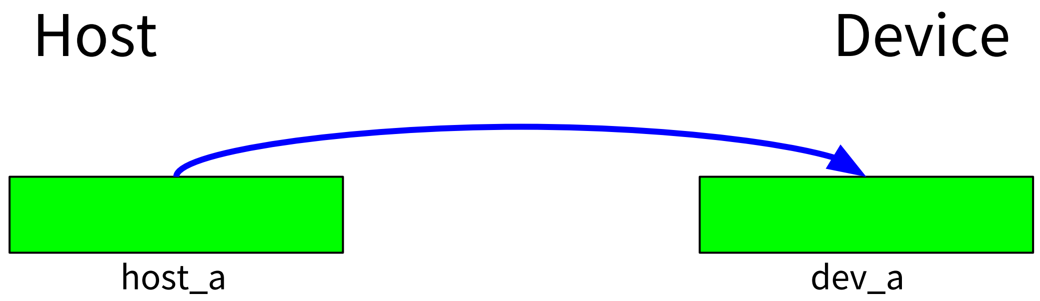
```c
int main(int argc, char *argv[]) {
   // grab n from the command line
   int n = atoi(argv[1]);

   // allocate host buffers
   float *host_a = (float *) malloc(n * sizeof(float));
   float *host_b = (float *) malloc(n * sizeof(float));
   float *host_c = (float *) malloc(n * sizeof(float));

   // fill A and B with random floats
   init_vec(host_a);
   init_vec(host_b);
   ...
```

2. Buffers & Transferring Data



2. Buffers & Transferring Data

- Before we can transfer data, we need to allocate a GPU buffer



Allocating Device Buffers

```
// cpu
float *host_a = (float *) malloc(n * sizeof(float));

// gpu
float *dev_a;
cudaError_t status;
status = cudaMalloc(&dev_a, n * sizeof(float));
```

- dev_a now points to a buffer in GPU's global memory
  - Do the same to create dev_b, dev_c

Transferring data

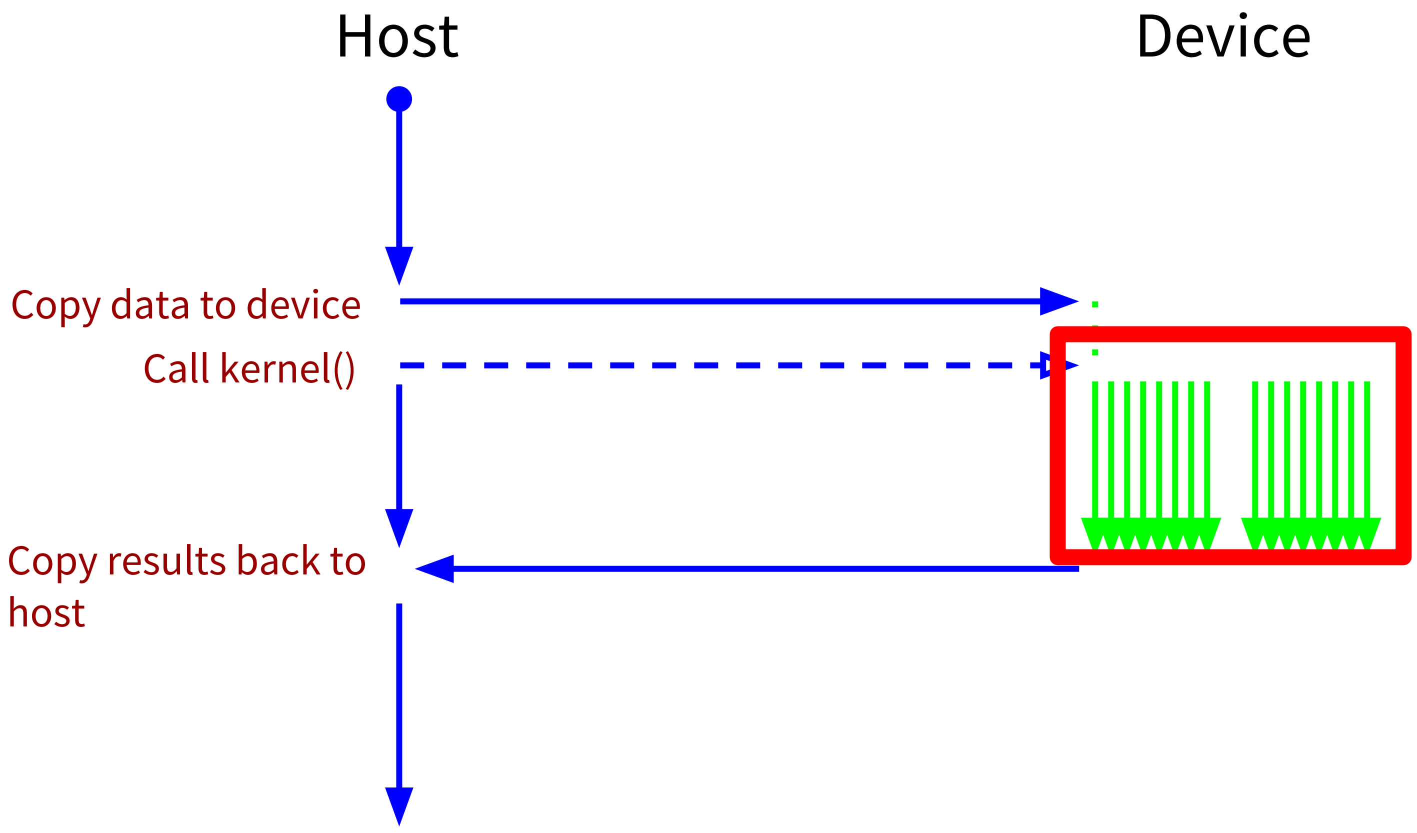
```
// Host -> Device
status = cudaMemcpy(
                 dev_a,                   // destination
                 host_a,                  // source
                 n * sizeof(float),       // size (bytes)
                 cudaMemcpyHostToDevice   // direction
         );
```

- Same for dev_b

Transferring data

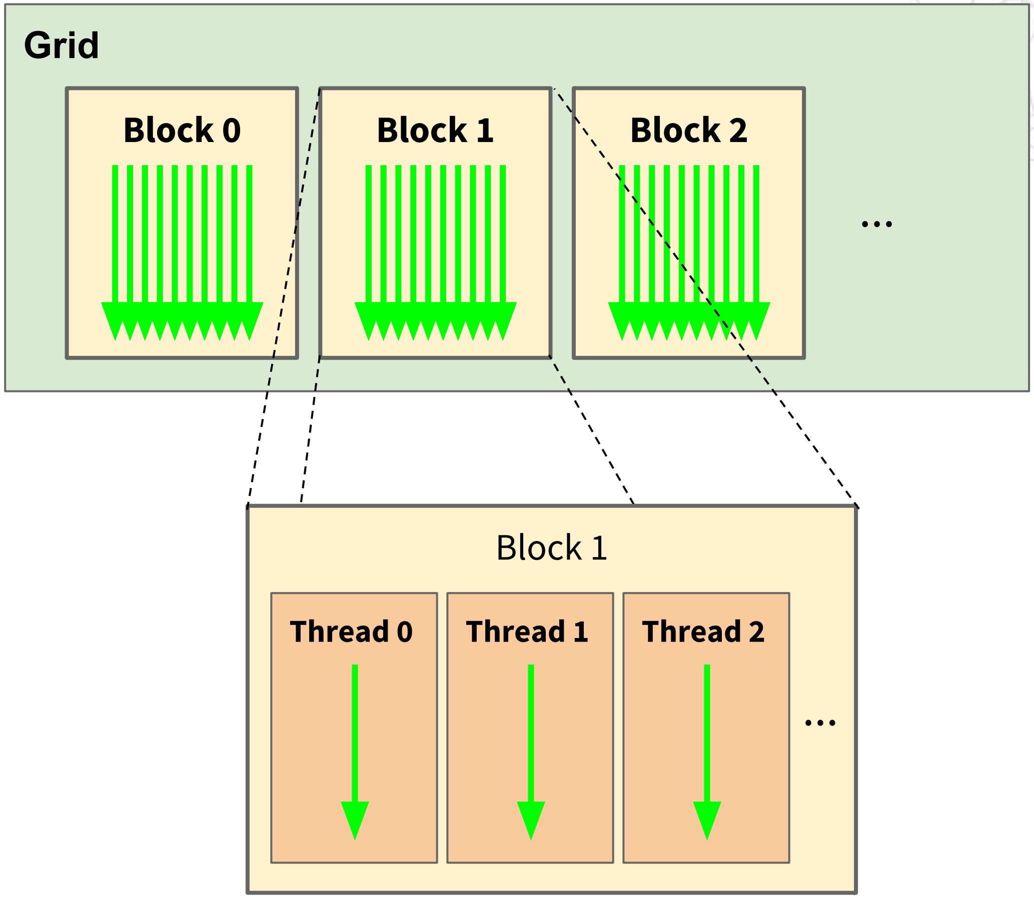
- `cudaMemcpy()` is *blocking*
  - Like `MPI_Send()`
  - Host waits…

3. Writing a Kernel



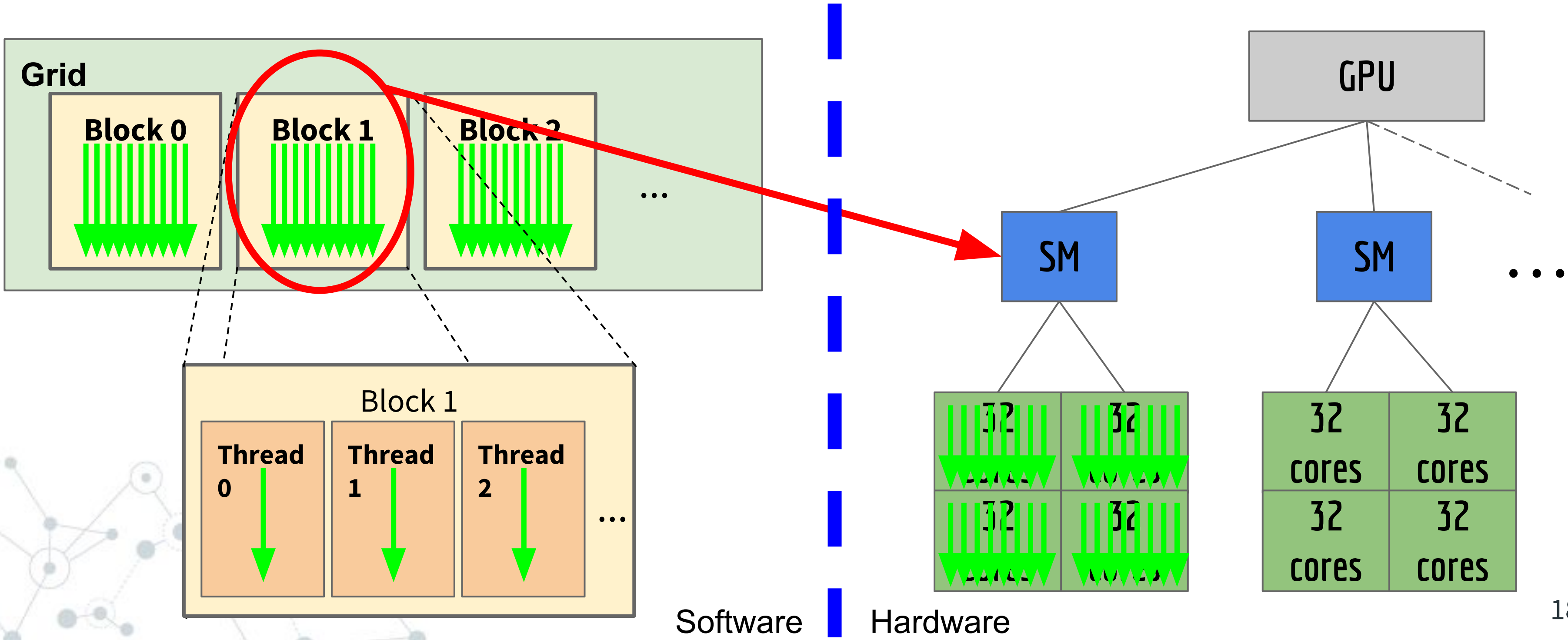
Threads in CUDA

- Thread grid
  - Contains all threads executing on GPU
  - Sub-divided into **thread blocks**



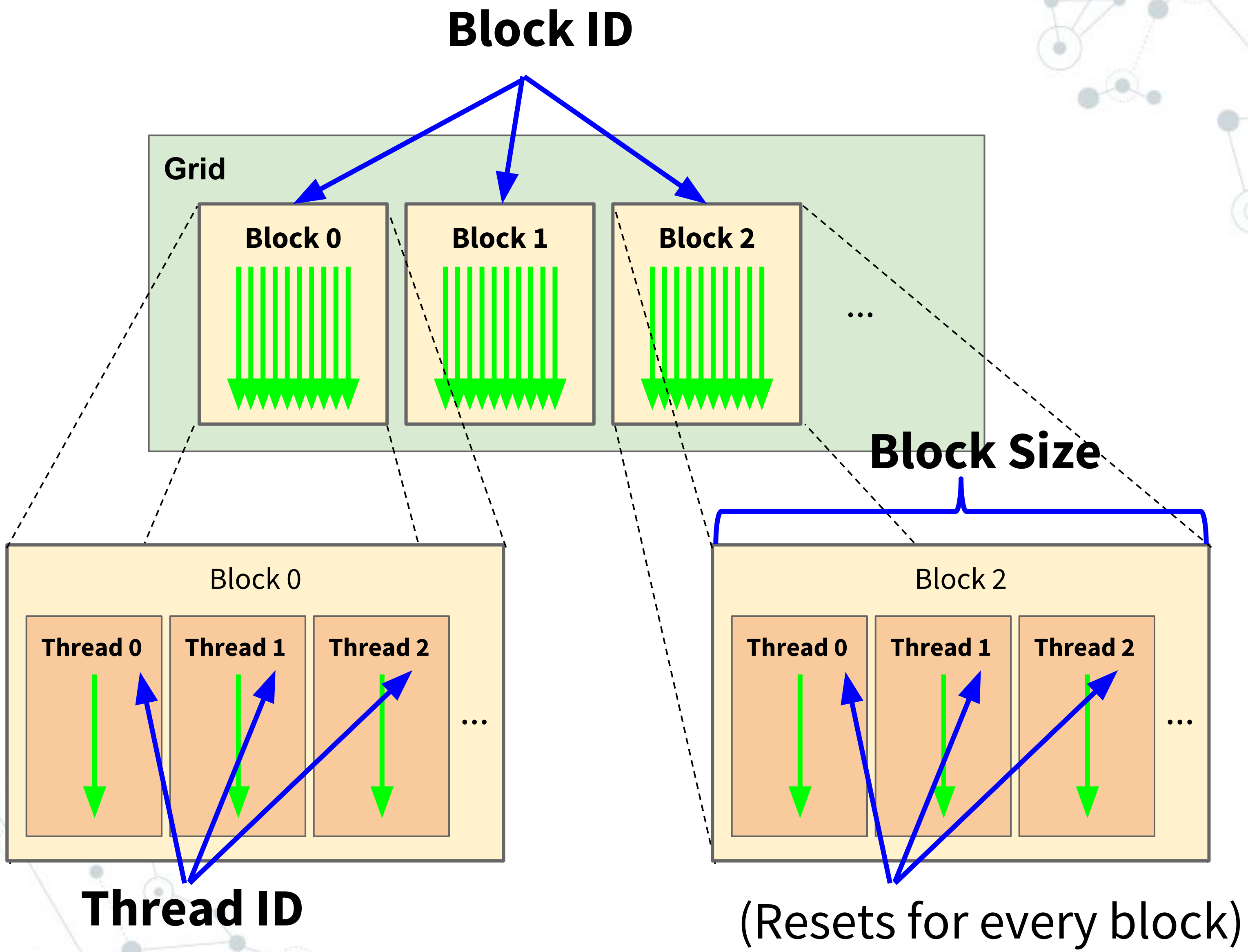
Why do we need blocks?

- Recall: GPU h/w is made up of multiple SMs.
  - GPU schedules each **s/w block** on a separate **h/w SM**
- The SM splits a block into warps
  - and schedules them on its 4 groups of cores



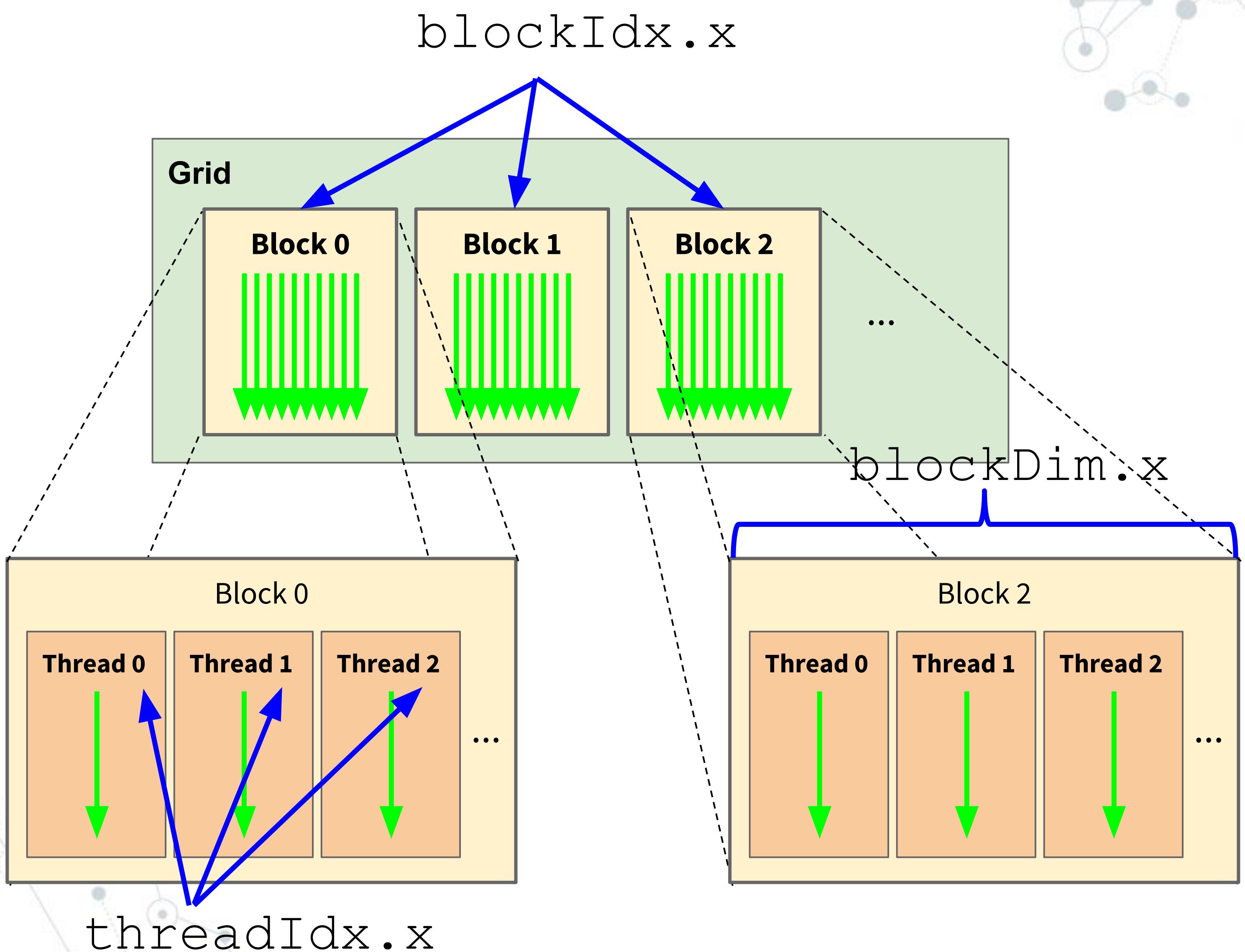
Thread IDs in CUDA

- Threads can access info about the thread grid:



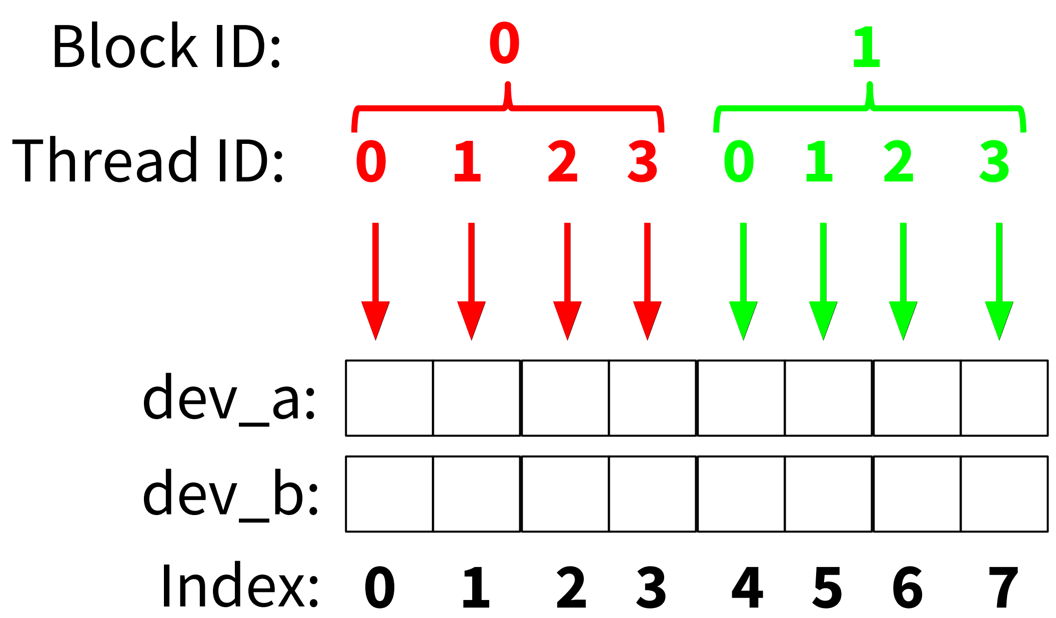
Thread IDs in CUDA

- In kernel functions, threads can use:



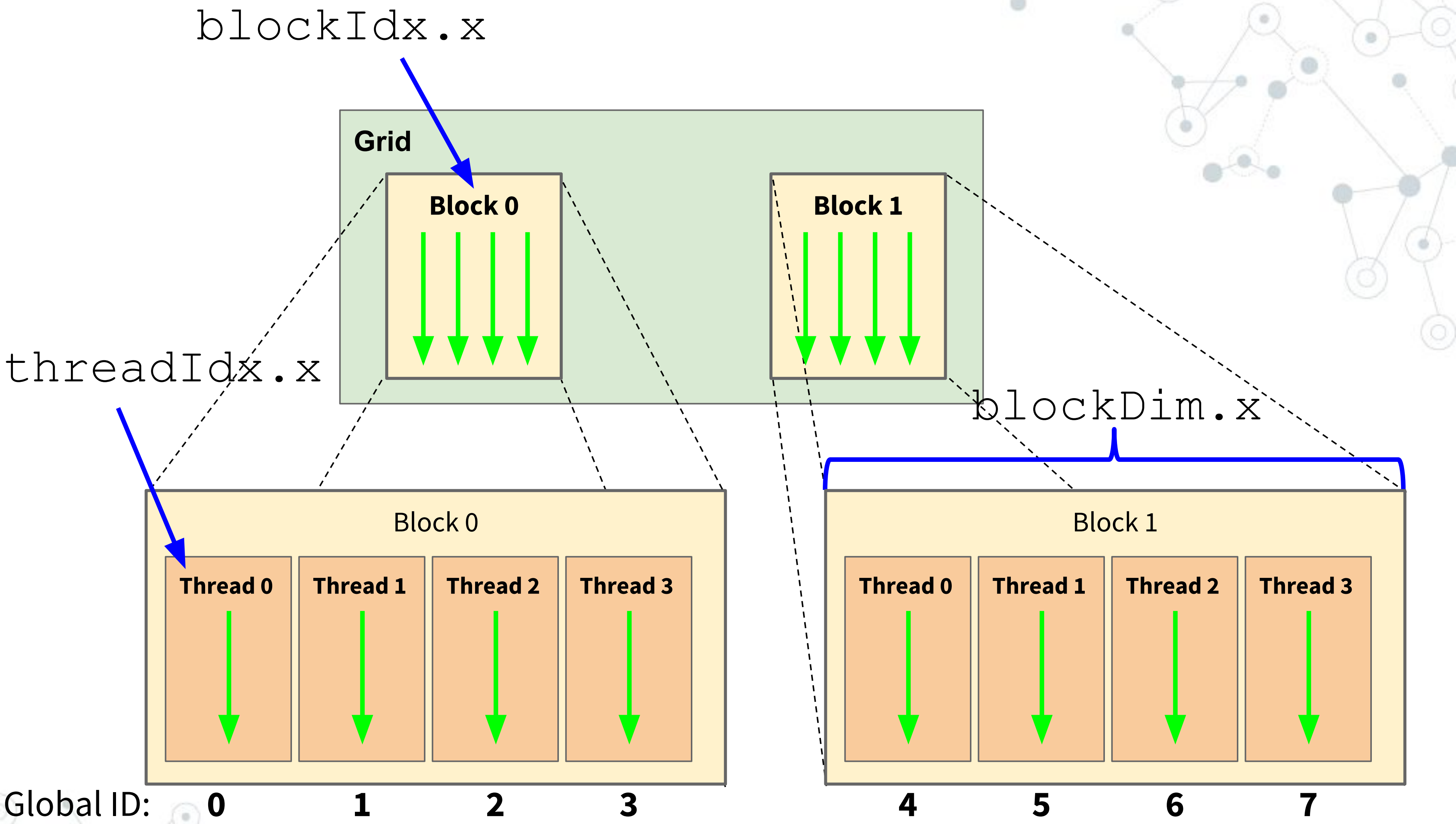
Example

- Suppose we have n = 8
  - So we launch 8 threads (one per column)...
  - Suppose our SM can only handle 4 threads, so we use 2 blocks (block size = 4)



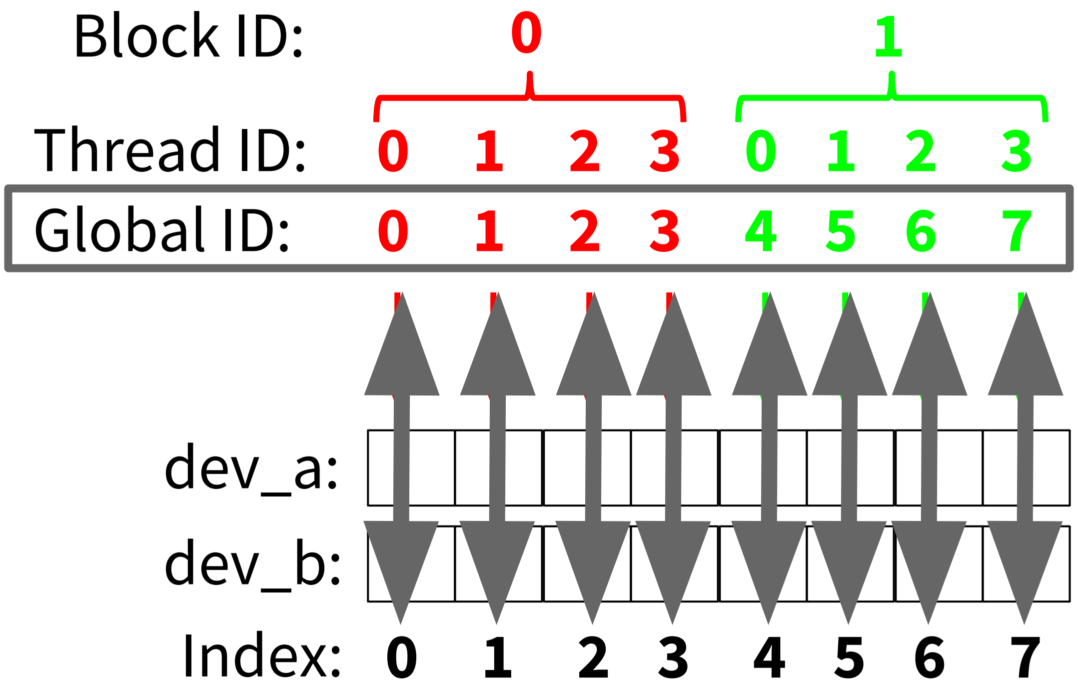
- How does a thread know which column to add?
  - Can't use Thread ID - resets in each block
  - Need a "**Global ID**" that keeps increasing *through* blocks

Calculating the Global ID




```
global_id = blockIdx.x * blockDim.x + threadIdx.x
```

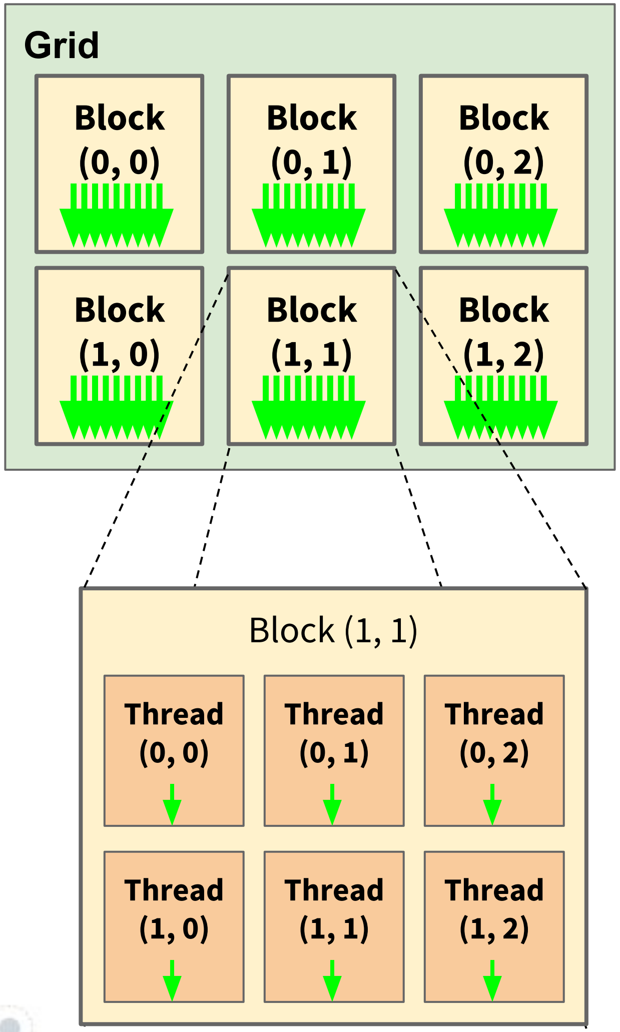
Using the Global ID



- Each thread adds the column given by its global ID

Why is there an "x" component in the IDs?

- Eg. `blockDim.x`
- The thread grid can also be multidimensional!

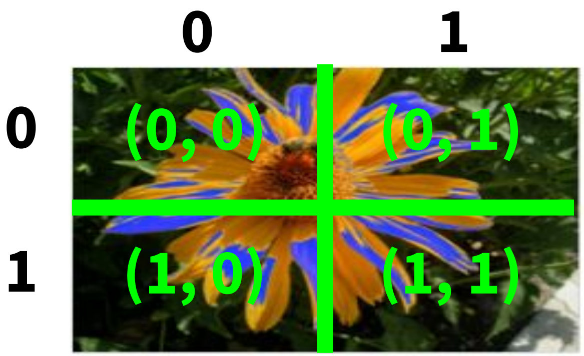


Multidimensional Thread Grids

- Why?

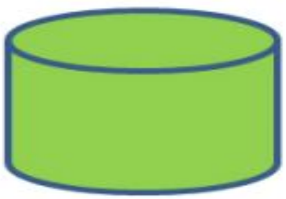
D = 1

ID format: **x**



D = 2

ID format: **(x, y)**



D = 3

ID format: **(x, y, z)**

Picking a grid and block size

Our problem is 1D - let's use a 1D grid.

To write our kernel, we need to know:

1. *How many threads do we need in our grid?*

```
int threads = n;
```

2. *How many threads in a block?*
   - Let's use the max # of threads supported by an SM
   - 1024 on our GPU

```
int block_size = 1024;
```

3. *How many blocks?*

```
int blocks = threads / block_size +
                 (threads % block_size > 0 ? 1 : 0);
```

- Note: This means we may have more threads than we need...
  - Eg. n = 1025

Writing a Kernel Function

```
__global__ void vec_add(float *a, float *b, float *c, int n)
{
  int global_id = blockIdx.x * blockDim.x + threadIdx.x;
  if (global_id < n)
  {
    c[global_id] = a[global_id] + b[global_id];
  }
}
```

Writing a Kernel Function

```
__global__ void vec_add(float *a, float *b, float *c, int n)
{
  int global_id = blockIdx.x * blockDim.x + threadIdx.x;
  if (global_id < n)
  {
    c[global_id] = a[global_id] + b[global_id];
  }
}
```

- Marks this as a kernel function

Writing a Kernel Function

```
__global__ void vec_add(float *a, float *b, float *c, int n)
{
  int global_id = blockIdx.x * blockDim.x + threadIdx.x;
  if (global_id < n)
  {
    c[global_id] = a[global_id] + b[global_id];
  }
}
```

- Kernel functions can’t return anything
  - All communication between host & device done through data transfers

Writing a Kernel Function

```
__global__ void vec_add(float *a, float *b, float *c, int n)
{
  int global_id = blockIdx.x * blockDim.x + threadIdx.x;
  if (global_id < n)
  {
    c[global_id] = a[global_id] + b[global_id];
  }
}
```

- Kernel name

Writing a Kernel Function

```
__global__ void vec_add(float *a, float *b, float *c, int n)
{
  int global_id = blockIdx.x * blockDim.x + threadIdx.x;
  if (global_id < n)
  {
    c[global_id] = a[global_id] + b[global_id];
  }
}
```

- Args are passed using “call by copy”
  - Pointers are shallow-copied
  - Args on stack are copied
  - Placed in constant memory

Writing a Kernel Function

```
__global__ void vec_add(float *a, float *b, float *c, int n)
{
  int global_id = blockIdx.x * blockDim.x + threadIdx.x;
  if (global_id < n)
  {
    c[global_id] = a[global_id] + b[global_id];
  }
}
```

- Calculate global ID

Writing a Kernel Function

```
__global__ void vec_add(float *a, float *b, float *c, int n)
{
  int global_id = blockIdx.x * blockDim.x + threadIdx.x;
  if (global_id < n)
  {
    c[global_id] = a[global_id] + b[global_id];
  }
}
```

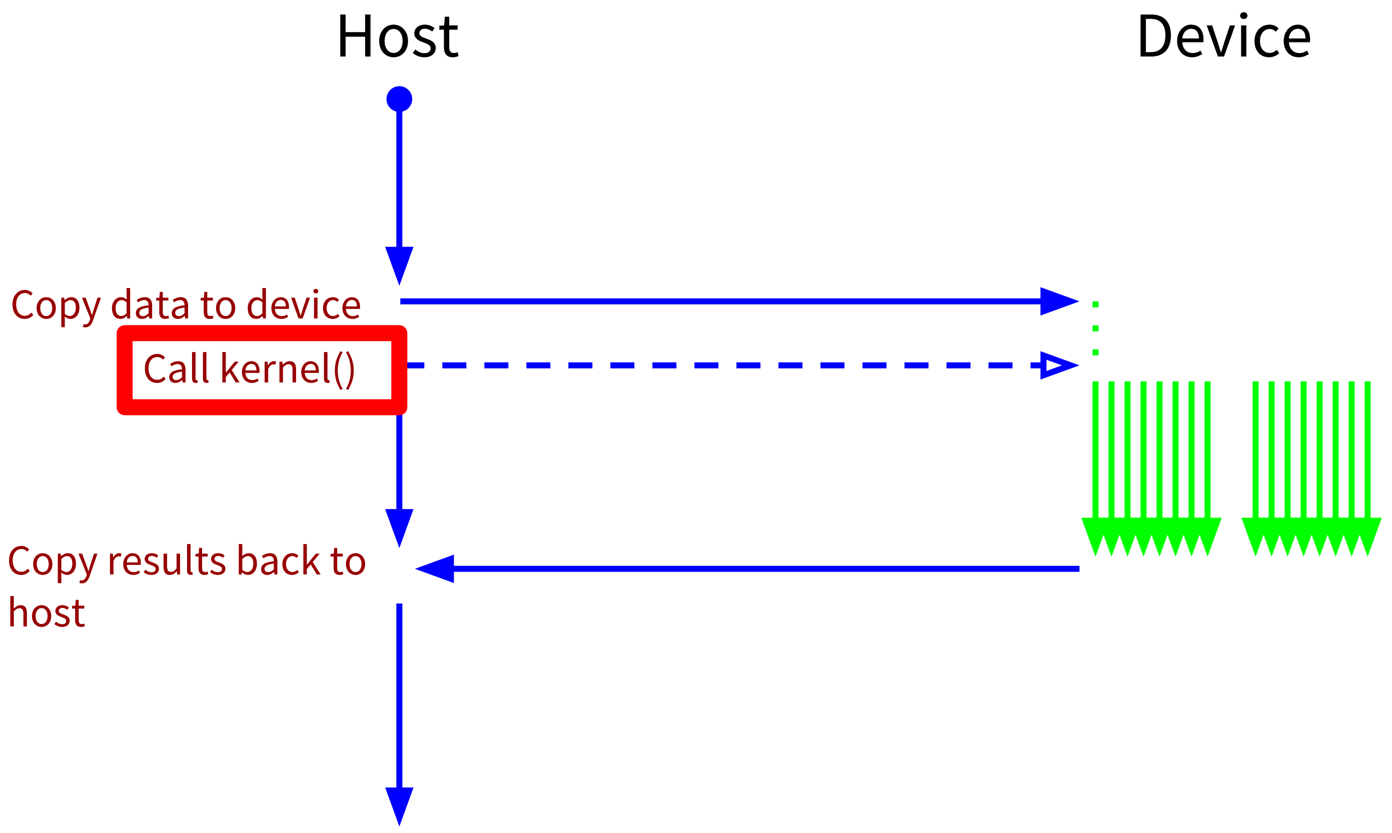
- Recall: we may have more threads than we need
  - last block...

Writing a Kernel Function

```
__global__ void vec_add(float *a, float *b, float *c, int n)
{
  int global_id = blockIdx.x * blockDim.x + threadIdx.x;
  if (global_id < n)
  {
    c[global_id] = a[global_id] + b[global_id];
  }
}
```

- global_id used to index vector
  - Each thread adds one column of vectors
- Result written to c (in dev memory)

4. Calling the Kernel Function



Launching the Kernel

- "***Launching***": calling a kernel function from the host
- Like a C function call…
  - plus some syntax to tell CUDA how many threads & blocks to use!

```
vec_add<<<blocks, block_size>>>(dev_a, dev_b, dev_c, n);
```

function name

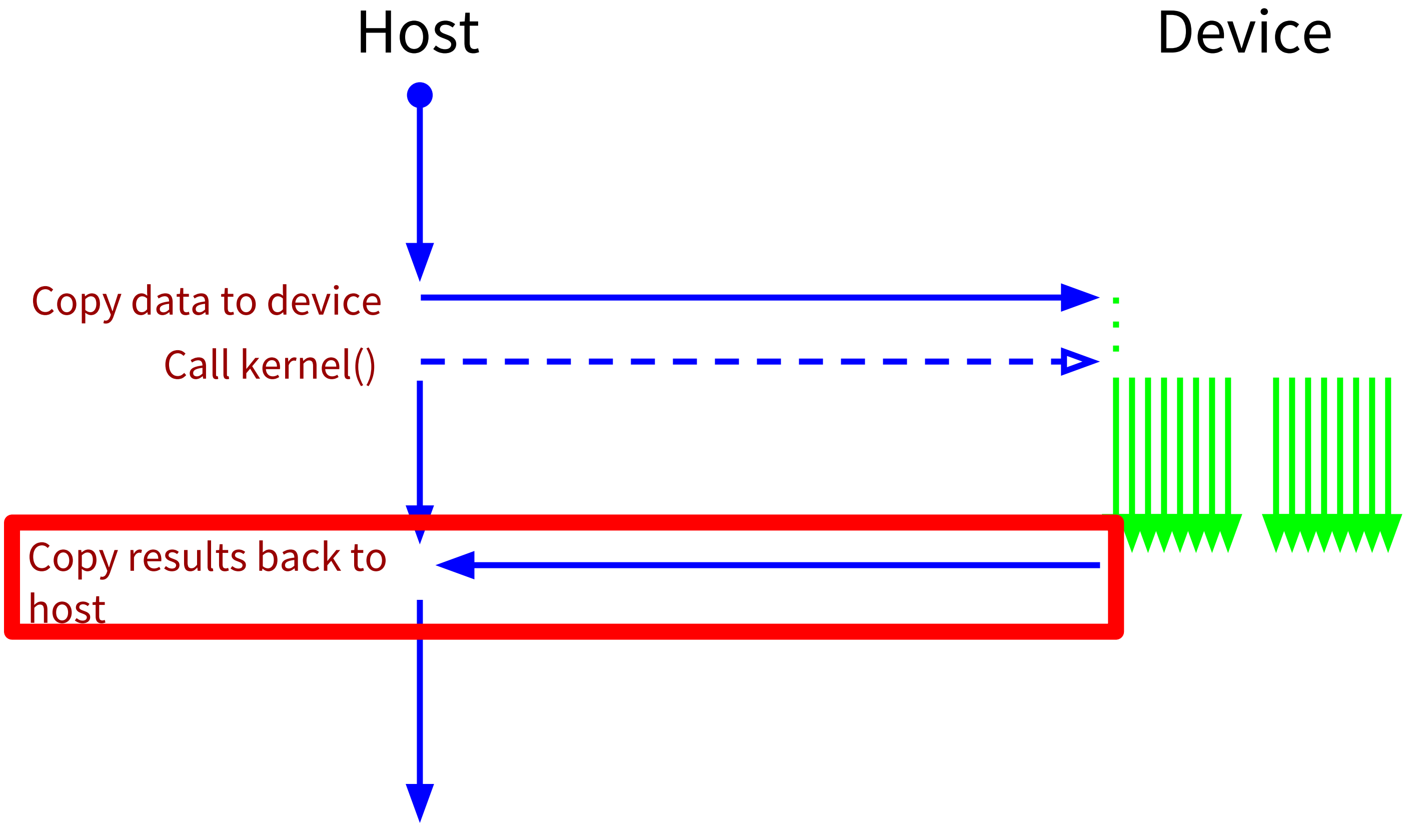
Blocks in grid

Threads in block

Pointers to our device buffers

Vector length

5. Retrieving the Result



Synchronization

- Kernel calls are non-blocking!
  - Host program continues on to next instruction
  - Can sync up at end using:
    1. `cudaDeviceSynchronize()`, OR
    2. Issuing a (blocking) `cudaMemcpy()`

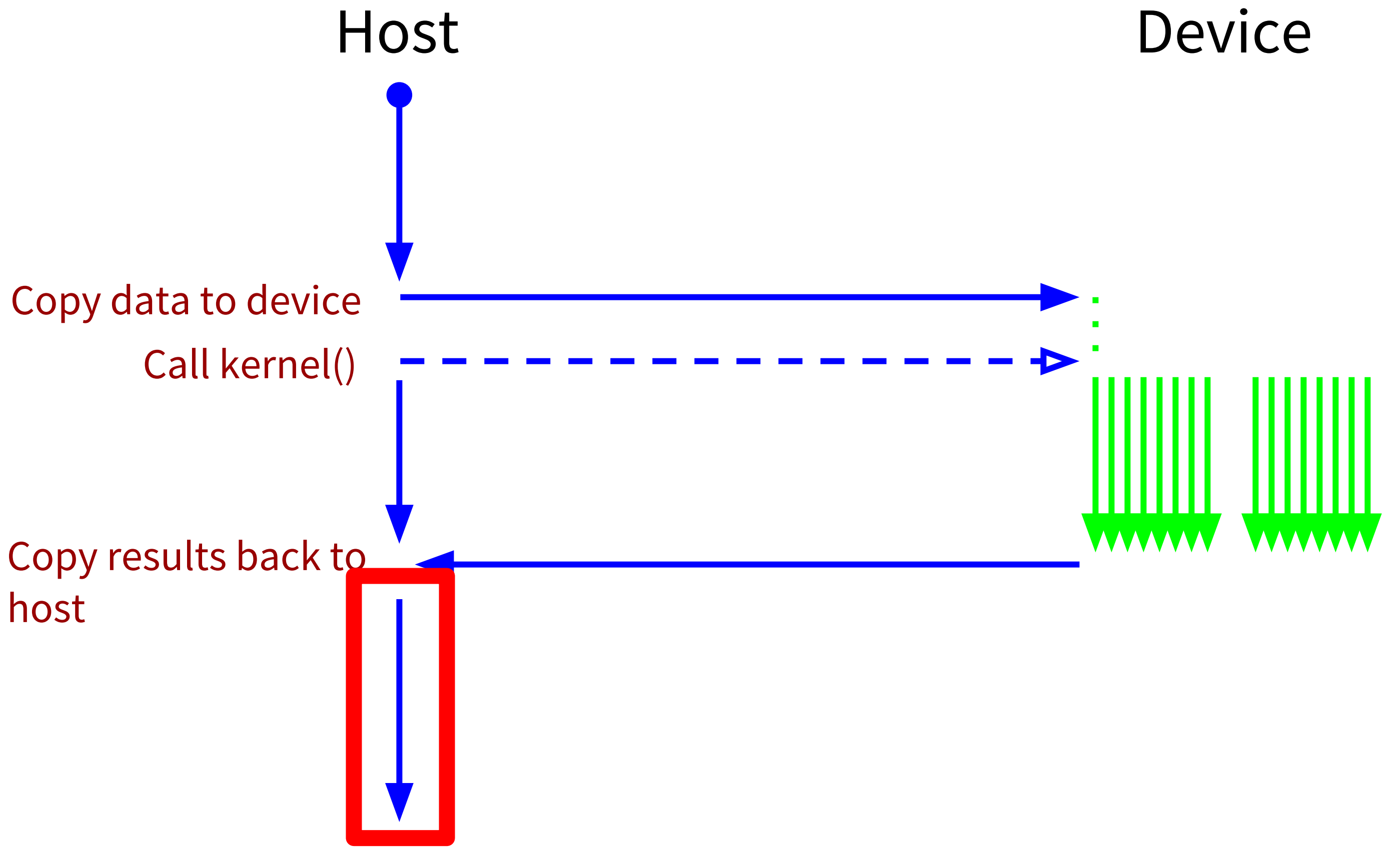
Preferred method if you need results back (avoids redundant sync)

```
vec_add<<<blocks, threads>>>(dev_a, dev_b, dev_c, n);
// host continues immediately...
```

Transferring the Result

```
// Device -> Host
status = cudaMemcpy(
                 host_c,                      // destination
                 dev_c,                       // source
                 n * sizeof(float),           // size (bytes)
                 cudaMemcpyDeviceToHost // direction
         );
```

5. Cleaning Up



Freeing Device Buffers

```
// cpu
free(host_a);
```

```
// gpu
cudaError_t status;
status = cudaFree(dev_a);
```

- Do the same for dev_b, dev_c

Example Code

- Complete vector sum code up on course website
  - vec_add.zip
- GPUs available on **`aviary`** machines
  - See the "**CUDA Programming Environments**" document for instructions