UNIVERSITY OF WOLVERHAMPTON

6CS005

High Performance Computing

Lecture 10

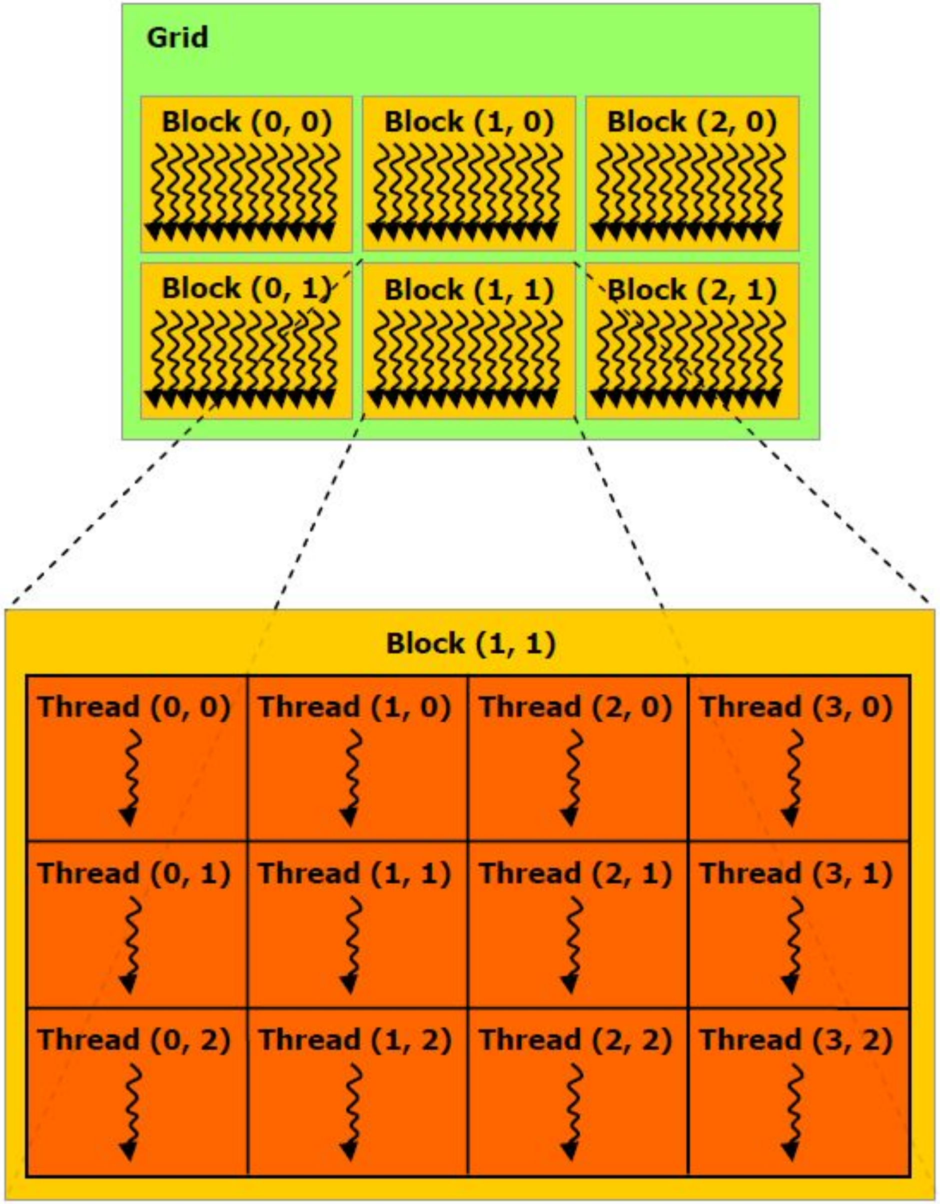
Concept of CUDA Threads, Blocks and Grids

CUDA Threads, Blocks and Grids

- A multithreaded program is partitioned into blocks of threads that execute independently from each other.
- A kernel function runs on the GPU using N parallel threads.
- Threads are grouped into 1, 2 or 3 dimensional blocks.
- Blocks are grouped into a 1, 2 or 3 dimensional grid.

CUDA Threads, Blocks and Grids

- Each thread's unique id is calculated from the thread index and block index.
- Here we have a 2 dimensional grid of blocks, with each grid being 2 dimensional.



CUDA Threads, Blocks and Grids

- Threads are grouped into blocks.
- Blocks are grouped into a grid.
- Both the grid and blocks are 3 dimensional
- So a specific thread needs to be indexed using the x, y, z index of the block in the grid and the x, y, z index of the block in the thread.

Moving to Parallel

- GPU computing is about massive parallelism
  - So how do we run code in parallel on the device?

```
add<<< 1, 1 >>>();

add<<< N, 1 >>>();
```

- Instead of executing `add()` once, execute N times in parallel

Vector Addition on the Device

- With `add()` running in parallel we can do vector addition

- Terminology: each parallel invocation of `add()` is referred to as a block
  - The set of blocks is referred to as a grid
  - Each invocation can refer to its block index using `blockIdx.x`

```
__global__ void add(int *a, int *b, int *c) {
        c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- By using `blockIdx.x` to index into the array, each block handles a different index

Vector Addition on the Device

```
__global  void add(int *a, int *b, int *c) {
        c[blockIdx.x] = a[blockIdx.x] +
        b[blockIdx.x];
}
```

- On the device, each block can execute in parallel:

Block 0	Block 1	Block 2	Block 3
`c[0]= a[0] + b[0];`	`c[1]= a[1] + b[1];`	`c[2]= a[2] + b[2];`	`c[3]= a[3] + b[3];`

Vector Addition on the Device: `add()`

- Returning to our parallelized **`add()`** kernel

```
__global__ void add(int *a, int *b, int *c) {
        c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- Let’s take a look at main()…

Vector Addition on the Device: main()

```
#define N 512
int main(void) {
    int *a  *b  *c          // host copies of a, b, c
    int *d_a, *d_b, *d_c;   // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a,  N);
    b = (int *)malloc(size); random_ints(b,  N);
    c = (int *)malloc(size);
```

Vector Addition on the Device: main()

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU with N blocks
    add<<<N,1>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Review (1 of 2)

- Difference between *host* and *device*
  - *Host* CPU
  - *Device* GPU

- Using `__global__` to declare a function as device code
  - Executes on the device
  - Called from the host

- Passing parameters from host code to a device function

Review (2 of 2)

- Basic device memory management
  - `cudaMalloc()`
  - `cudaMemcpy()`
  - `cudaFree()`

- Launching parallel kernels
  - Launch `N` copies of `add()` with `add<<<N,1>>>(…);`
  - Use `blockIdx.x` to access block index

CUDA Threads

- Terminology: a block can be split into parallel threads

- Let's change `add()` to use parallel *threads* instead of parallel *blocks*

```
__global__ void add(int *a, int *b, int *c) {
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

- We use `threadIdx.x` instead of `blockIdx.x`

- Need to make one change in `main()`...

Vector Addition Using Threads: main()

```
#define N 512
int main(void) {
    int *a, *b, *c;                 // host copies of a, b, c
    int *d_a, *d_b, *d_c;           // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b,  c and setup input values
    a = (int *)malloc(size); random_ints(a,  N);
    b = (int *)malloc(size); random_ints(b,  N);
    c = (int *)malloc(size);
```

Vector Addition Using Threads: main()

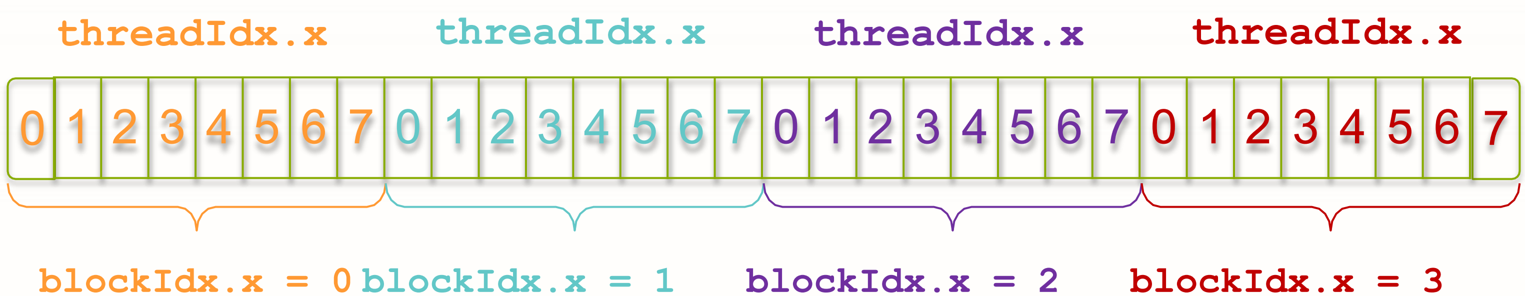
```
  // Copy inputs to device
 cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
 cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

 // Launch add() kernel on GPU with N threads
 add<<<1,N>>>(d_a, d_b, d_c);

 // Copy result back to host
 cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

 // Cleanup
 free(a); free(b); free(c);
 cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
 return 0;
}
```

Combining Blocks and Threads

- We've seen parallel vector addition using:
  - Many blocks with one thread each
  - One block with many threads
- Let's adapt vector addition to use both blocks and threads
- Why? We'll come to that…
- First let's discuss data indexing…

Indexing Arrays with Blocks and Threads

- No longer as simple as using `blockIdx.x` and `threadIdx.x`
  - Consider indexing an array with one element per thread (8 threads/block)

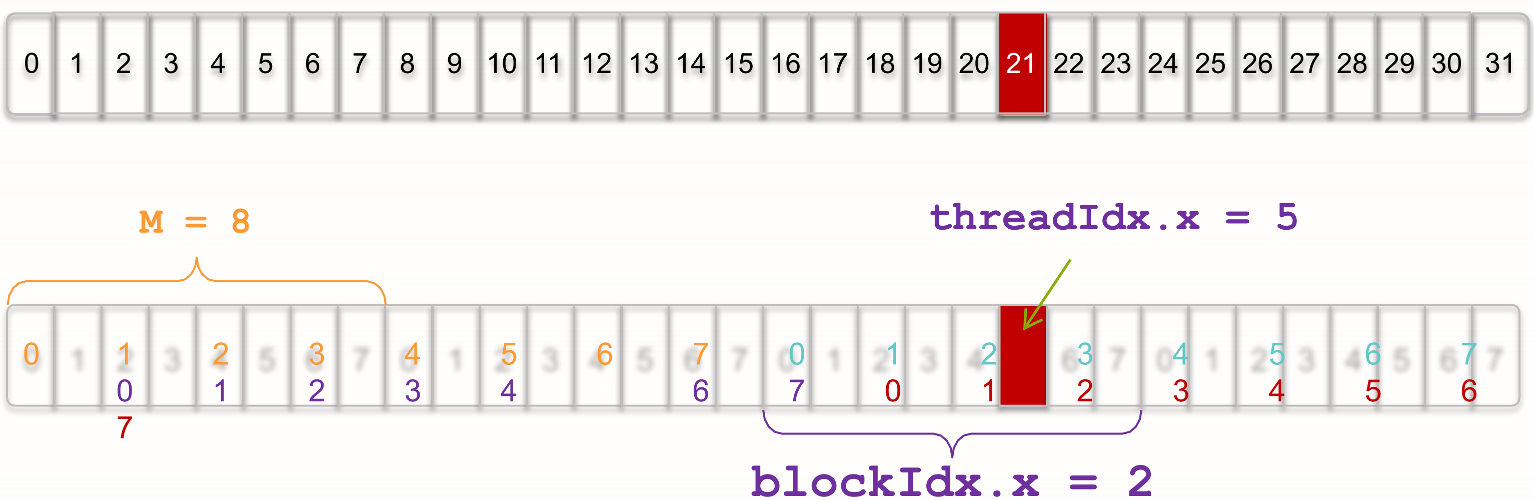


- With M threads/block a unique index for each thread is given by:

```
int index = threadIdx.x + blockIdx.x * M;
```

Indexing Arrays: Example

- Which thread will operate on the red element?




```
int index = threadIdx.x + blockIdx.x * M;
                        =   5  +  2  * 8;
          = 21;
```

Vector Addition with Blocks and Threads

- Use the built-in variable `blockDim.x` for threads per block

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

- Combined version of `add()` to use parallel threads *and* parallel blocks

```
__global void add(int *a, int *b, int *c) {
    int index = threadIdx.x + blockIdx.x *
    blockDim.x; c[index] = a[index] + b[index];
}
```

- What changes need to be made in `main()`?

Addition with Blocks and Threads: main()

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main(void) {
    int *a, *b, *c;                 // host copies of a, b, c
    int *d_a, *d_b, *d_c;           // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b,  c and setup input values
    a = (int *)malloc(size); random_ints(a,  N);
    b = (int *)malloc(size); random_ints(b,  N);
    c = (int *)malloc(size);
```

Addition with Blocks and Threads: main()

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<N/THREADS_PER_BLOCK  THREADS_PER_BLOCK>>>(d_a,  d_b,
    d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Handling Arbitrary Vector Sizes

- Typical problems are not friendly multiples of `blockDim.x`

- Avoid accessing beyond the end of the arrays:

```
__global  void add(int *a, int *b, int *c, int n) { int
    index = threadIdx.x + blockIdx.x * blockDim.x; if
    (index < n)
        c[index] = a[index] + b[index];
}
```

- Update the kernel launch:

```
add<<<(N + M-1) / M,M>>>(d_a, d_b, d_c, N);
```

Why Bother with Threads?

- Threads seem unnecessary
  - They add a level of complexity
  - What do we gain?

- Unlike parallel blocks, threads have mechanisms to:
  - Communicate
  - Synchronize

- To look closer, we need a new example…

Using multiple Block and Thread Dimensions in CUDA

Different types of pre-defined variables in CUDA – attributes of current thread

The variables below are defined within the CUDA library. They all represent different features of the block/thread system when running the kernel function. At least one of these variables will hold a different number for each thread spawned

therefore, we need to calculate thread ID's based on these variables

gridDim.x/y/z

blockDim.x/y/z

blockId.x/y/z

threadId.x/y/z

Different types of pre-defined variables in CUDA – attributes of current thread

gridDim.x/y/z – Represents the amount of blocks in a specific dimension within a large grid

blockDim.x/y/z – Represents the amount of threads within a block in a specific dimensions

blockId.x/y/z – Represents the current block ID this thread has spawned in

threadId.x/y/z – Represents the current thread ID

<<<dim3(1,1,1), dim3(100,1,1)

The above equation represents 1 block and 100 threads, this means 100 threads will spawn. We don't need to use all of those pre-defined variables to make each thread unique because threadIdx.x will give us 100 unique numbers from 0-99 for each time the kernel function is used (see demo in workshop)

<<<dim3(2,1,1), dim3(50,1,1)

The above equation represents 2 blocks and 50 threads, this means 100 threads will spawn (same as before but the setup is different). This time, we cannot use just threadIdx.x because this will only give us 2 batches of 50 unique numbers (0-49 x 2). However, the blockIdx.x will give us 2 unique numbers (0-1) for each batch of thread IDs. To calculate the exact thread ID for the function, we need to use the total amount of threads (blockDim), the current block ID and the current thread ID.

blockDim.x * blockIdx.x + threadIdx.x

<<<dim3(2,2,1), dim3(5,5,1)

The above equation represents 4 blocks (2 in x and 2 in y) and 25 threads (5 in x and 5 in y), this means 100 threads will spawn. This time, we cannot use just x dimension variables because now we are using blocks and threads in the y. However, we still need to use the equation we generated in the last slide (blockDim.x * blockIdx.x + threadIdx.x). We need to create the formula which uses the blockdim for x and y, block ID for x and y and thread ID in x and y to make sure we generate a unique ID for each thread. That’s your job, not mine! But remember, will it make a big time difference for the applications you are building?

End of Lecture 10