LAU
الجامعة اللبنانية الأميركية
Lebanese American University

CSC 447: Parallel Programming for Multi-Core and Cluster Systems

CUDA Threads

Instructor: Haidar M. Harmanani

Spring 2018

CUDA Blocks

- Memory Management
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

Moving to Parallel

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

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

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

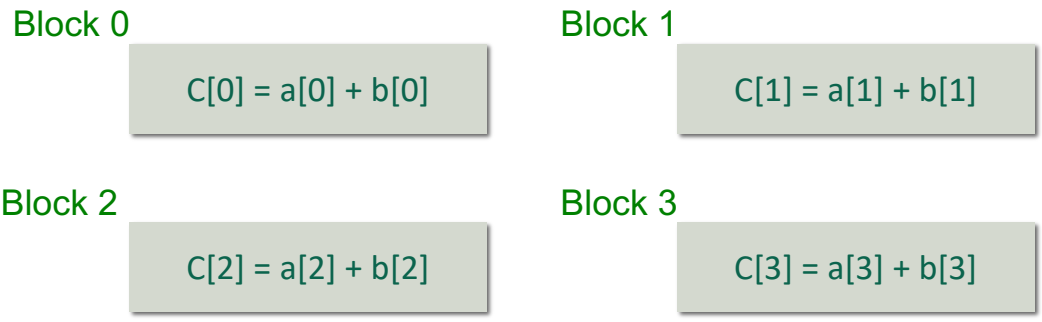
- Instead of executing `vecAdd()` once, execute N times in parallel

Vector Addition on the GPU

- Well, since GPUs are about massive parallelism we will rewrite the `vecAdd()` method such that
  - Each invocation of `vecAdd()` is referred to as a *block*
  - Kernel can refer to its block's index with the variable blockIdx.x
  - Each block adds a value from a[] and b[], storing the result in c[]
- By using `blockIdx.x` to index into the array, each block handles a different index

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

Vector Addition on the GPU




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


Vector Addition on the GPU: N Blocks

```
#define N 512
int main( void )
{
      int *a, *b, *c;                    // a, b, c
      int size = N * sizeof( int);       // The total number of bytes per vector

      // Allocate memory
      cudaMallocManaged(&a, size);
      cudaMallocManaged(&b, size);
      cudaMallocManaged(&c, size);

      random_ints( a, N );
      random_ints( b, N );

      // launch add() kernel on GPU using N Parallel Blocks
      vecAdd<<< N, 1 >>>( a, b, d);

      cudaDeviceSynchronize(); // Wait for the GPU to finish


      // Free all our allocated memory
      cudaFree( a ); cudaFree( b ); cudaFree( c );
}
```

Launch N blocks with 1 thread each

CUDA Threads

- Memory Management
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

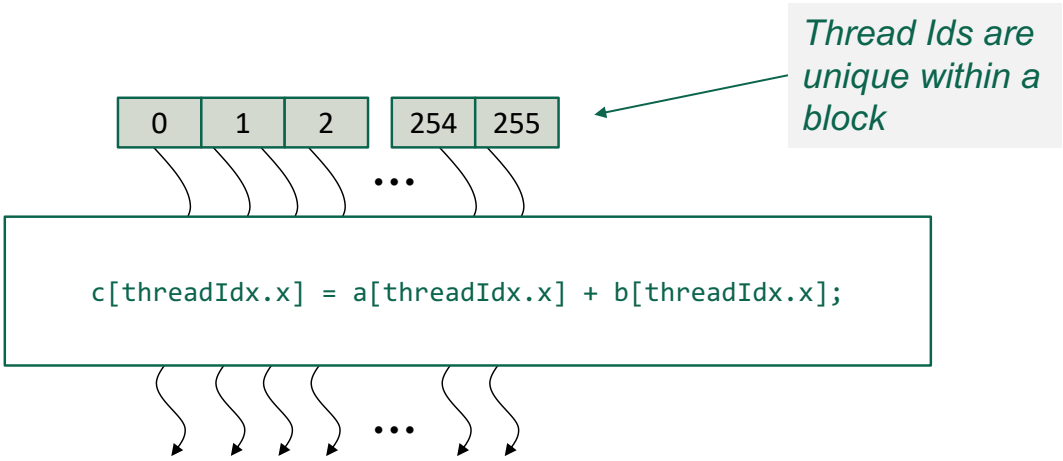
CUDA Threads

- Terminology: a block can be split into parallel *threads*
- Let's change `vecAdd()` to use parallel *threads* instead of parallel *blocks*

```
__global__ void vecAdd(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()` …

Arrays of Parallel Threads



Arrays of Parallel Threads

- To execute kernels in parallel with CUDA
  - Launch a grid of blocks of threads, specifying the number of blocks per grid (`bpg`) and threads per block (`tpb`).
  - Total number of threads launched will be the product of `bpg` × `tpb`
  - This can be in the millions!

Caveat: Threads in CUDA

- Hardware limits
  - The number of blocks in a single launch to 65,535
  - The number of threads per block with which we can launch a kernel to a maximum of `maxThreadsPerBlock`
  - Number is hardware dependent
    - 1024 threads per block on a Kepler, 512 on most older GPUs
- How do we use a thread-based approach to add two vectors of size greater than 512 or 1024?
- Each thread has a gloabl ID
  - This is basically just finding an offset given a 2D grid of 3D blocks of 3D threads, but can get very confusing!

GPU *n* Numbers Thread Addition

- Divide thread array into multiple blocks
  - Threads within a block cooperate via shared memory, atomic operations and barrier synchronization
  - Threads in different blocks cannot cooperate

GPU *n* Numbers Thread Addition

- A block can be split into parallel threads
- A CUDA kernel is executed by a grid (array) of threads
  - All threads in a grid run the same kernel code (SPMD)
  - Each thread has an index that it uses to compute memory addresses and make control decisions

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


Vector Addition on the GPU: 1 Block with N Threads

```
#define N 512
int main( void )
{
      int *a, *b, *c;                    // a, b, c
      int size = N * sizeof( int);       // The total number of bytes per vector

      // Allocate memory
      cudaMallocManaged(&a, size);
      cudaMallocManaged(&b, size);
      cudaMallocManaged(&c, size);

      random_ints( a, N );
      random_ints( b, N );

      // launch vecAdd() kernel on GPU using N Parallel Blocks
      vecAdd<<< 1, N >>>( a, b, d);

      cudaDeviceSynchronize(); // Wait for the GPU to finish

      // Free all our allocated memory
      cudaFree( a ); cudaFree( b ); cudaFree( c );
}
```

Launch 1 block with N threads each

Combining Blocks and Threads

- Memory Management
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

Combining Blocks and Threads

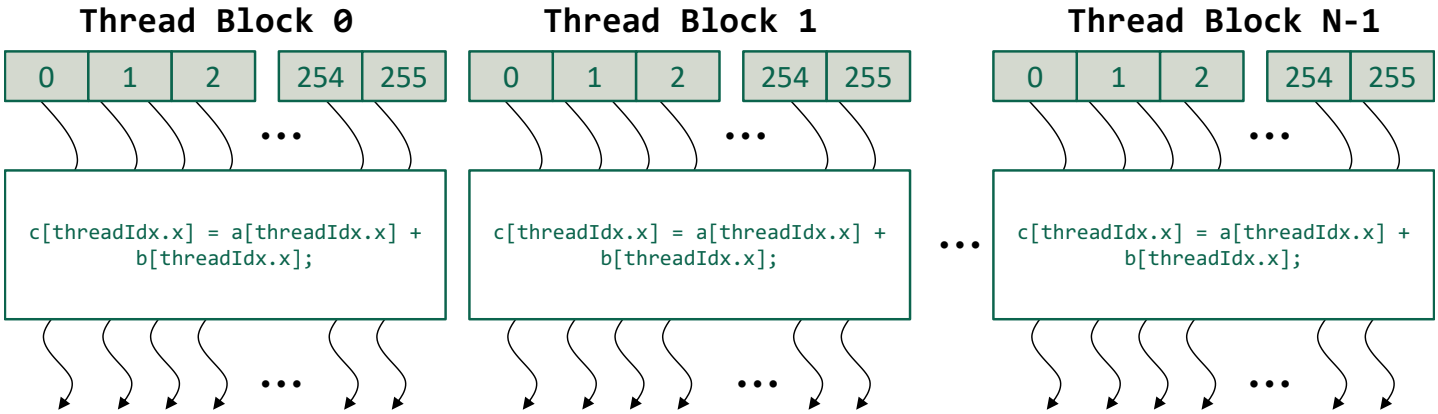
- We've seen parallel vector addition using:
  - Many blocks with one thread each
  - One block with many threads
- Let's adapt vector addition to use both blocks and threads

Blocks, Grids, and Threads

- Kernels are launched as *grids* of *blocks* of threads.
  - Threads can further be divided into 32-thread *warps*, and each thread in a warp is called a *lane*
- Thread blocks are separately scheduled onto SMs, and threads within a given block are executed by the same SM

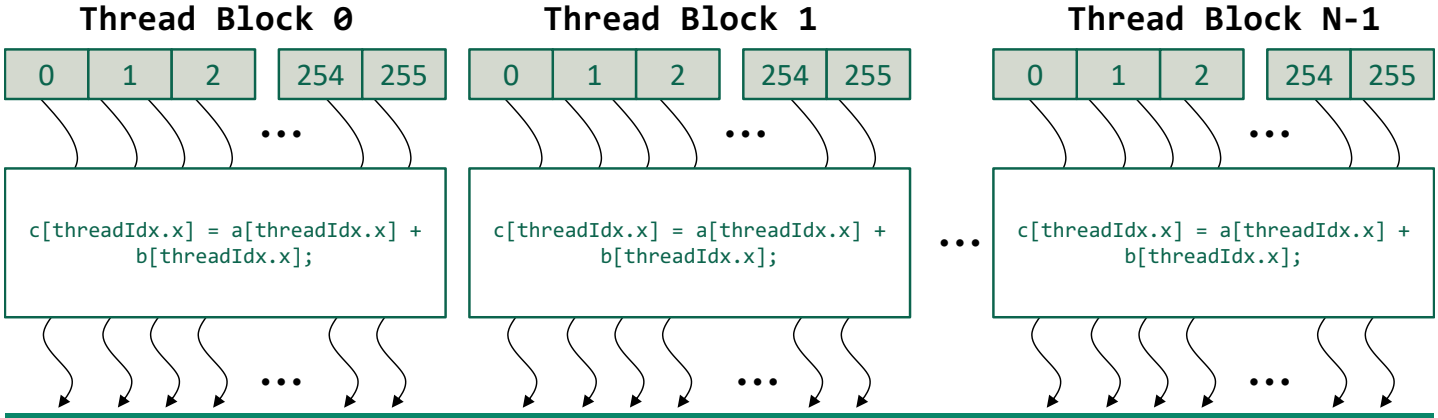
Combining Threads and Blocks

Thread Ids are not unique across all blocks???



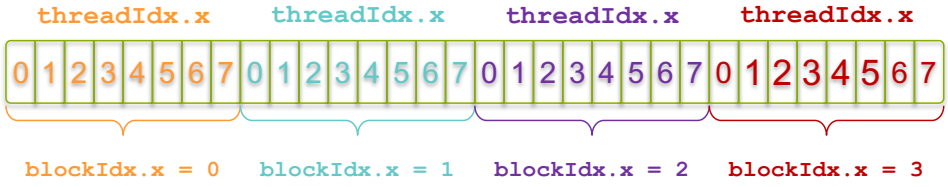
Combining Threads and Blocks

- Indexing arrays with threads and blocks is now problematic!
- Solution?



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)

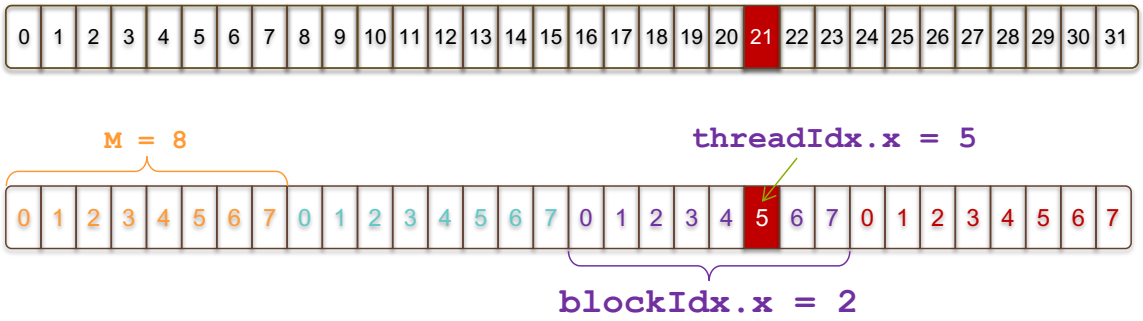


- With M threads/block a unique index for each thread is given by:

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

Indexing Arrays: Example

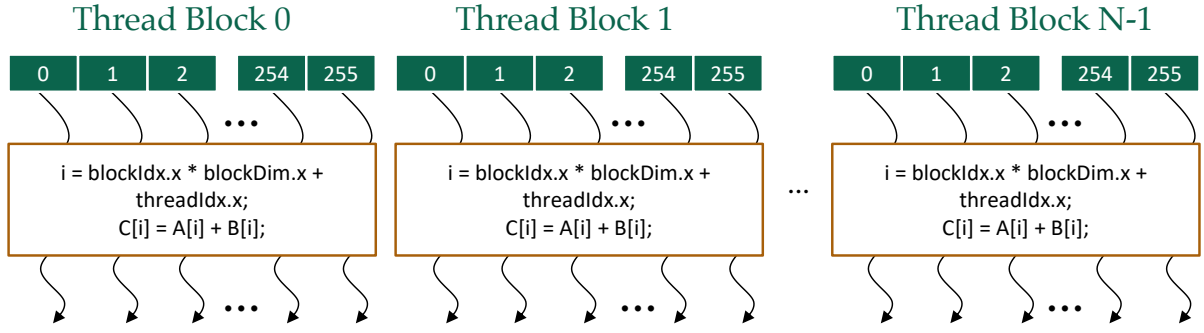
- Which thread will operate on the red element?




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

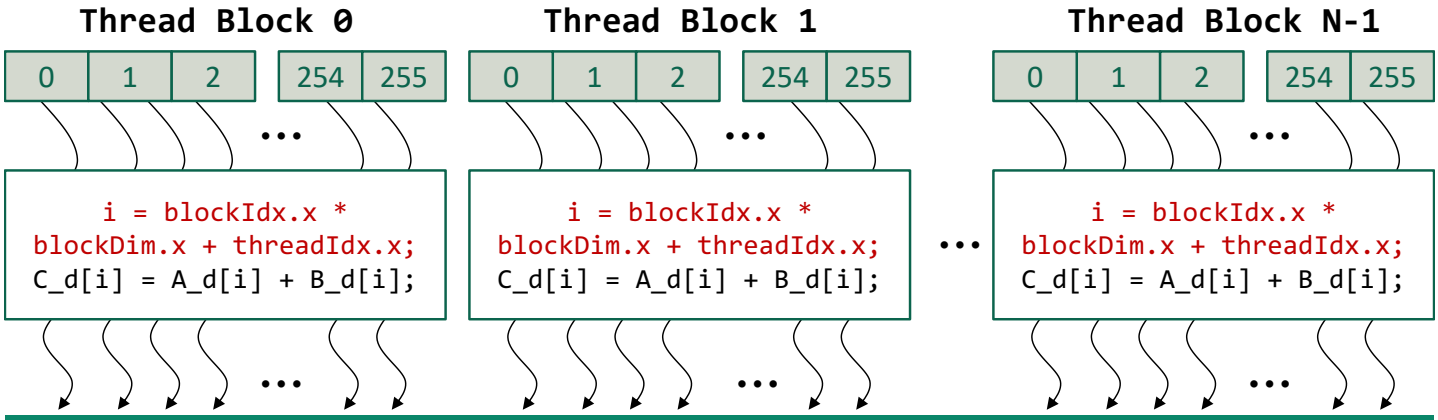

Thread Blocks: Scalable Cooperation

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



Combining Threads and Blocks

- Solution?



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 `vecAdd()` to use parallel threads *and* parallel blocks
- What changes need to be made in `main()`?

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

```
#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);
```


Launch N blocks with 512 threads each

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

    // Launch add() kernel on GPU
    vecAdd<<<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 vecAdd(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:

```
vecAdd<<<(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 but that will wait till next week!