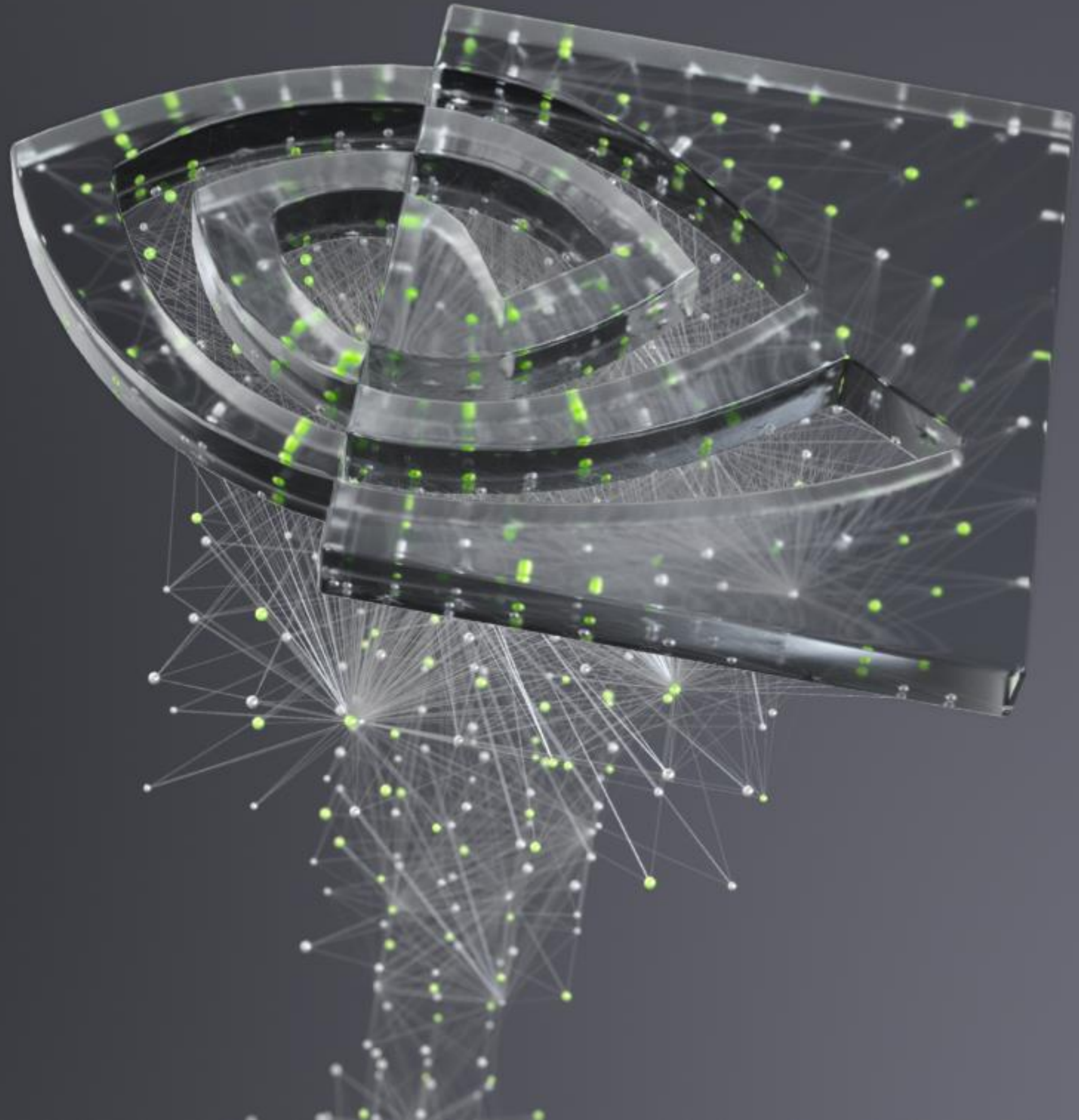
CUDA C++ BASICS



WHAT IS CUDA?

- CUDA Architecture
  - Expose GPU parallelism for general-purpose computing
  - Expose/Enable performance
- CUDA C++
  - Based on industry-standard C++
  - Set of extensions to enable heterogeneous programming
  - Straightforward APIs to manage devices, memory etc.
- This session introduces CUDA C++
  - Other languages/bindings available: Fortran, Python, Matlab, etc.

GPU KERNELS: DEVICE CODE

```
__global__ void mykernel(void) {

}
```

- CUDA C++ keyword `__global__` indicates a function that:
    - Runs on the device
    - Is called from host code (can also be called from other device code)
- `nvcc` separates source code into host and device components
    - Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
    - Host functions (e.g. `main()`) processed by standard host compiler (e.g. gcc)

GPU KERNELS: DEVICE CODE

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

- Triple angle brackets mark a call to *device* code
  - Also called a “kernel launch”
  - We’ll return to the parameters (1,1) in a moment
  - The parameters inside the triple angle brackets are the CUDA kernel **execution configuration**

RUNNING CODE IN PARALLEL

- GPU computing is about massive parallelism
  - So how do we run code in parallel on the device, for example adding one vector to another?

```
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 all 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) {
        b[blockIdx.x] = b[blockIdx.x] + a[blockIdx.x];
}
```

- By using `blockIdx.x` to index into the array, each block handles a different index
- Built-in variables like `blockIdx.x` are zero-indexed (C/C++ style), 0..**N**-1, where **N** is from the kernel execution configuration indicated at the kernel launch

VECTOR ADDITION ON THE DEVICE

```
int main() {
    int *a, *b;
    int size = N * sizeof(int);
    // Allocate memory
    cudaMallocManaged((void **)&a, size);
    cudaMallocManaged((void **)&b, size);
    // Set up input values
    random_ints(a, N); random_ints(b, N);
    // Launch add() kernel on GPU with N blocks
    add<<<N, 1>>>(a, b);
    // Cleanup
    cudaFree(a); cudaFree(b);
    return 0;
}
```

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) {
    b[threadIdx.x] = b[threadIdx.x] + a[threadIdx.x];
}
```

- We use `threadIdx.x` instead of `blockIdx.x`
- Need to make one change in `main()`:

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

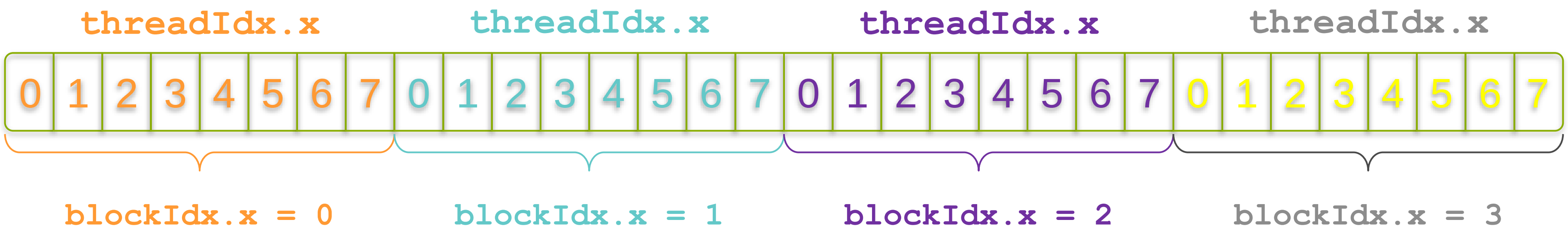
COMBINING BLOCKS *AND* THREADS

- 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):

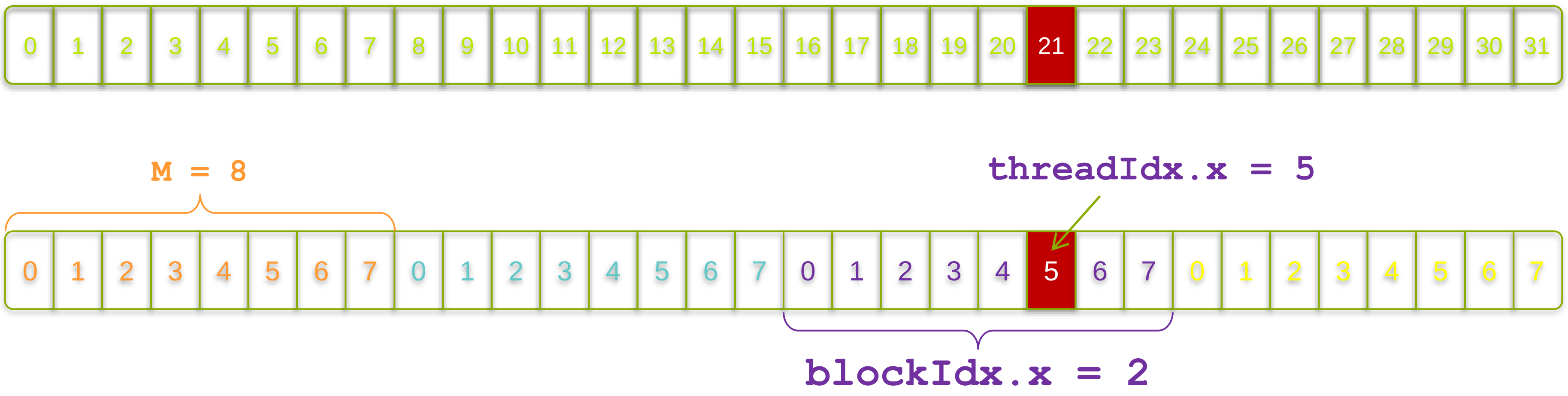


- With M threads/block a unique index for each thread is given by:

```
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 index = threadIdx.x + blockIdx.x * blockDim.x;
    b[index] = b[index] + a[index];
}
```

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

VECTOR ADDITION WITH BLOCKS AND THREADS

```
int main() {
   int *a, *b;
   int size = N * sizeof(int);
   // Allocate memory
   cudaMallocManaged((void **)&a, size);
   cudaMallocManaged((void **)&b, size);
   // Set up input values
   random_ints(a, N); random_ints(b, N);
   // Launch add() kernel on GPU with N blocks
   add<<<N / THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(a, b);
   // Cleanup
   cudaFree(a); cudaFree(b);
   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 n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n)
        b[index] = b[index] + a[index];
}
```

- Update the kernel launch:

```
add<<<(N + M - 1) / M, M>>>(a, b, N);
```

WHY DO WE HAVE HIERARCHICAL PARALLELISM?

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

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

- Two level hierarchy maps more appropriately to GPU hardware design

FURTHER STUDY

- An introduction to CUDA:
    - https://devblogs.nvidia.com/easy-introduction-cuda-c-and-c/
- Another introduction to CUDA:
    - https://devblogs.nvidia.com/even-easier-introduction-cuda/
- CUDA Programming Guide:
    - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
- CUDA Documentation:
    - https://docs.nvidia.com/cuda/index.html
- OLCF CUDA Training Series
    - https://www.olcf.ornl.gov/cuda-training-series/

NVIDIA