CUDA C/C++ BASICS

NVIDIA Corporation
What is CUDA?

CUDA Architecture
- Expose GPU parallelism for general-purpose computing
- Retain performance

CUDA C/C++
- Based on industry-standard C/C++
- Small set of extensions to enable heterogeneous programming
- Straightforward APIs to manage devices, memory etc.

This session introduces CUDA C/C++
Introduction to CUDA C/C++

What will you learn in this session?
- Start with vector addition
- Write and launch CUDA C/C++ kernels
- Manage GPU memory
- Manage communication and synchronization

(Some knowledge of C programming is assumed.)
Heterogeneous Computing

- Terminology:
  - **Host**: The CPU and its memory (host memory)
  - **Device**: The GPU and its memory (device memory)
1. Copy input data from CPU memory to GPU memory
Simple Processing Flow

1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory
Parallel Programming in CUDA C/C++

▪ GPU computing is about massive parallelism!

▪ We need an interesting example...

▪ We’ll start with vector addition
GPU Kernels: Device Code

```c
__global__ void mykernel(void) { 
}
```

- CUDA C/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
    - `gcc`, `cl.exe`
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

That’s all that is required to execute a function on the GPU!
Memory Management

Host and device memory are separate entities

Device pointers point to GPU memory
- May be passed to/from host code
- May not be dereferenced in host code

Host pointers point to CPU memory
- May be passed to/from device code
- May not be dereferenced in device code

Simple CUDA API for handling device memory

- cudaMalloc(), cudaFree(), cudaMemcpy()
- Similar to the C equivalents malloc(), free(), memcpy()
Running code in parallel

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

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

```c
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

```c
__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: \( c[0] = a[0] + b[0] \)
Vector Addition on the Device: \texttt{add()} 

Returning to our parallelized \texttt{add()} kernel

\begin{verbatim}
\_\_global\_\_ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
\end{verbatim}

Let’s take a look at main()...
Vector Addition on the Device: `main()`

```c
#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()`

```c
// 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;
```
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 (or possibly from other device code)

Passing parameters from host code to a device function
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

```c
__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()`:
  ```c
  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):

```
threadIdx.x
```

```
0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
```

```
blockIdx.x = 0         blockIdx.x = 1         blockIdx.x = 2         blockIdx.x = 3
```

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?

$$\text{int index} = \text{threadIdx.x} + \text{blockIdx.x} \times M;$$

$$= 5 + 2 \times 8;$$

$$= 21;$$
Use the built-in variable `blockDim.x` for threads per block

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

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

```c
__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()`

```c
#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...
Review

- Launching parallel kernels
  - Launch $N$ copies of `add()` with `add<<<N/M, M>>>(...);
  - Use `blockIdx.x` to access block index
  - Use `threadIdx.x` to access thread index within block

- Assign elements to threads:

  ```
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  ```
1D Stencil

Consider applying a 1D stencil to a 1D array of elements

- Each output element is the sum of input elements within a radius

If radius is 3, then each output element is the sum of 7 input elements:

```
+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |
  |   |   |   |   |   |   | X |
  |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |
```
Implementing Within a Block

- Each thread processes one output element
  - blockDim.x elements per block

- Input elements are read several times
  - With radius 3, each input element is read seven times
Sharing Data Between Threads

- Terminology: within a block, threads share data via shared memory

- Extremely fast on-chip memory, user-managed

- Declare using __shared__, allocated per block

- Data is not visible to threads in other blocks
Implementing With Shared Memory

- Cache data in shared memory
  - Read \((\text{blockDim}.x + 2 \times \text{radius})\) input elements from global memory to shared memory
  - Compute \(\text{blockDim}.x\) output elements
  - Write \(\text{blockDim}.x\) output elements to global memory

- Each block needs a **halo** of \(\text{radius}\) elements at each boundary
__global__ void stencil_1d(int *in, int *out) {

    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] =
        in[gindex + BLOCK_SIZE];
    }
}
// Apply the stencil
int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];

// Store the result
out[gindex] = result;
Data Race!

- The stencil example will not work...
- Suppose thread 15 reads the halo before thread 0 has fetched it...

```c
temp[lindex] = in[gindex];  // Store at temp[18]
if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];  // Skipped, threadIdx > RADIUS
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}
int result = 0;
result += temp[lindex + 1];  // Load from temp[19]
```
__syncthreads()

void __syncthreads();

Synchronizes all threads within a block
  - Used to prevent RAW / WAR / WAW hazards

All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + radius;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();
}
Stencil Kernel

// Apply the stencil
int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];

// Store the result
out[gindex] = result;
Launching parallel threads

- Launch $N$ blocks with $M$ threads per block with `kernel<<<N,M>>>(...) ;`
- Use `blockIdx.x` to access block index within grid
- Use `threadIdx.x` to access thread index within block

Allocate elements to threads:

```c
int index = threadIdx.x + blockIdx.x * blockDim.x;
```
Use `__shared__` to declare a variable/array in shared memory
- Data is shared between threads in a block
- Not visible to threads in other blocks

Use `__syncthreads()` as a barrier
- Use to prevent data hazards
Further Study

An introduction to CUDA:

Another introduction to CUDA:

CUDA Programming Guide:

CUDA Documentation:
- https://docs.nvidia.com/cuda/index.html
Questions?