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 from “Hello World!”
- Write and launch CUDA C/C++ kernels
- Manage GPU memory
- Manage communication and synchronization
Prerequisites

- You (probably) need experience with C or C++
- You don’t need GPU experience
- You don’t need parallel programming experience
- You don’t need graphics experience
CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices
HELLO WORLD!
Heterogeneous Computing

- **Terminology:**
  - *Host*  The CPU and its memory (host memory)
  - *Device* The GPU and its memory (device memory)
Heterogeneous Computing

```cpp
#include <iostream>
#include <algorithm>

using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__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];
    }

    __syncthreads();

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

    // Store the result
    out[gindex] = result;
}

void fill_ints(int* x, int n) {
    fill_n(x, n, 1);
}

int main() {
    int* in, *out;
    fill_ints(in,  N + 2*RADIUS);
    fill_ints(out, N + 2*RADIUS);

    int size = (N + 2*RADIUS) * sizeof(int);
    Alloc space for host copies and setup values
    in  = (int*)malloc(size);
    fill_ints(in,  N + 2*RADIUS);
    out = (int*)malloc(size);
    fill_ints(out, N + 2*RADIUS);

    Alloc space for device copies
    cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

    // Launch stencil_1d() kernel on GPU
    stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS, d_out + RADIUS);

    // Copy result back to host
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

    Cleanup
    free(in); free(out);
    cudaFree(d_in);
    cudaFree(d_out);
    return 0;
}
```

parallel fn
serial code
parallel code
serial code
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
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
3. Copy results from GPU memory to CPU memory
Hello World!

```c
int main(void) {
    printf("Hello World!\n");
    return 0;
}
```

- Standard C that runs on the host
- NVIDIA compiler (nvcc) can be used to compile programs with no device code

Output:

```
$ nvcc hello_world.cu
$ a.out
Hello World!
$ 
```
Hello World! with Device Code

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

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;

    
- Two new syntactic elements...
Hello World! with Device Code

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

- **CUDA C/C++ keyword** `__global__` indicates a function that:
  - Runs on the device
  - Is called from host 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`
mykernel<<<1,1>>>();

- Triple angle brackets mark a call from host code 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!
Hello World! with Device Code

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

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}
```

Output:

```
$ nvcc hello.cu
$ a.out
Hello World!
$
```

- `mykernel()` does nothing, somewhat anticlimactic!
Parallel Programming in CUDA C/C++

- But wait... GPU computing is about massive parallelism!

- We need a more interesting example...

- We’ll start by adding two integers and build up to vector addition
A simple kernel to add two integers

```c
__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}
```

As before, `__global__` is a CUDA C/C++ keyword meaning:
- `add()` will execute on the device
- `add()` will be called from the host
Addition on the Device

Note that we use pointers for the variables

```c
__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}
```

`add()` runs on the device, so `a`, `b` and `c` must point to device memory

We need to allocate memory on the GPU
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()`, `cudaMemcpys()`
- Similar to the C equivalents `malloc()`, `free()`, `memcpy()`
Addition on the Device: \texttt{add()} \\

Returning to our \texttt{add()} kernel

\begin{verbatim}
__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}
\end{verbatim}

Let’s take a look at \texttt{main()}...
Addition on the Device: `main()`

```c
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 = sizeof(int);

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

    // Setup input values
    a = 2;
    b = 7;
}
```
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
add<<<1,1>>>(d_a, d_b, d_c);

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

// Cleanup
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
```

RUNNING IN PARALLEL

CONCEPTS

- Heterogeneous Computing
- 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?

```
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`.

```c
__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**
  
  \[ c[0] = a[0] + b[0]; \]

- **Block 1**
  
  \[ c[1] = a[1] + b[1]; \]

- **Block 2**
  

- **Block 3**
  
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 \texttt{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

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
INTRODUCING THREADS

CONCEPTS

- Heterogeneous Computing
- 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 `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()`...
Vector Addition Using Threads: \texttt{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 Using Threads: `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 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 THREADS AND BLOCKS
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...
No longer as simple as using `blockIdx.x` and `threadIdx.x`

Consider indexing an array with one element per thread (8 threads/block)

```
<table>
<thead>
<tr>
<th>threadIdx.x</th>
<th>threadIdx.x</th>
<th>threadIdx.x</th>
<th>threadIdx.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7</td>
<td>0 1 2 3 4 5 6 7</td>
<td>0 1 2 3 4 5 6 7</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
</tbody>
</table>
```

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?

\[
\text{int } \text{index} = \text{threadIdx.x} + \text{blockIdx.x} \times M; \\
= 5 + 2 \times 8; \\
= 21;
\]
Vector Addition with Blocks and Threads

- 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: \texttt{main()}

\begin{verbatim}
// 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;
\end{verbatim}
Handling Arbitrary Vector Sizes

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

- Avoid accessing beyond the end of the arrays:

  ```c
  __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:

  ```c
  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

Allocate elements to threads:

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

CONCEPTS

Heterogeneous Computing
Blocks
Threads
Indexing
Shared memory
__syncthreads()
Asynchronous operation
Handling errors
Managing devices
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:
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];
if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}
int result = 0;
result += temp[lindex + 1];
```

- Store at `temp[18]`

- Skipped, `threadIdx > RADIUS`

- 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();
}
/ 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
MANAGING THE DEVICE

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices
Coordinating Host & Device

- **Kernel launches are asynchronous**
  - Control returns to the CPU immediately

- CPU needs to synchronize before consuming the results

  **cudaMemcpy()**
  - Blocks the CPU until the copy is complete
  - Copy begins when all preceding CUDA calls have completed

  **cudaMemcpyAsync()**
  - Asynchronous, does not block the CPU

  **cudaDeviceSynchronize()**
  - Blocks the CPU until all preceding CUDA calls have completed
All CUDA API calls return an error code (**cudaError_t**)

- Error in the API call itself
- OR
- Error in an earlier asynchronous operation (e.g. kernel)

Get the error code for the last error:

```c
cudaError_t cudaGetLastError(void)
```

Get a string to describe the error:

```c
char *cudaGetErrorString(cudaError_t)
```

```c
printf("%s\n", cudaGetErrorString(cudaGetLastError()));
```
Device Management

- Application can query and select GPUs
  - `cudaGetDeviceCount(int *count)`
  - `cudaSetDevice(int device)`
  - `cudaGetDevice(int *device)`
  - `cudaGetDeviceProperties(cudaDeviceProp *prop, int device)`

- Multiple threads can share a device

- A single thread can manage multiple devices
  - `cudaSetDevice(i)` to select current device
  - `cudaMemcpy(...)` for peer-to-peer copies†

† requires OS and device support
Introduction to CUDA C/C++

What have we learned?

- Write and launch CUDA C/C++ kernels
  - `__global__`, `blockIdx.x`, `threadIdx.x`, `<<<>>>`
- Manage GPU memory
  - `cudaMalloc()`, `cudaMemcpy()`, `cudaFree()`
- Manage communication and synchronization
  - `__shared__`, `__syncthreads()`
  - `cudaMemcpy()` vs `cudaMemcpyAsync()`, `cudaDeviceSynchronize()`
The **compute capability** of a device describes its architecture, e.g.

- Number of registers
- Sizes of memories
- Features & capabilities

The following presentations concentrate on Fermi + Kepler

**Compute Capability**

<table>
<thead>
<tr>
<th>Compute Capability</th>
<th>Selected Features (see CUDA C Programming Guide for complete list)</th>
<th>Tesla models</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>Fundamental CUDA support</td>
<td>870</td>
</tr>
<tr>
<td>1.3</td>
<td>Double precision, improved memory accesses, atomics</td>
<td>10-series</td>
</tr>
<tr>
<td>2.0</td>
<td>Caches, fused multiply-add, 3D grids, surfaces, ECC, P2P, concurrent kernels/copies, function pointers, recursion</td>
<td>Fermi-series</td>
</tr>
<tr>
<td>3.0</td>
<td>Expanded grid, warp shuffle</td>
<td>Kepler-series</td>
</tr>
<tr>
<td>3.5</td>
<td>Dynamic Parallelism, Hyper-Q</td>
<td>Kepler-series</td>
</tr>
</tbody>
</table>

The following presentations concentrate on Fermi + Kepler

**Compute Capability >= 2.0**
IDs and Dimensions

A kernel is launched as a grid of blocks of threads
- `blockIdx` and `threadIdx` are 3D
- We showed only one dimension (x)

**Built-in variables:**
- `threadIdx`
- `blockIdx`
- `blockDim`
- `gridDim`
Topics we skipped

We skipped some details, you can learn more:

- CUDA Programming Guide
- CUDA Zone - tools, training, webinars and more
  http://developer.nvidia.com/cuda

Need a quick primer for later:

- Multi-dimensional indexing
- Textures
Questions?