CUDA C/C++ BASICS

NVIDIA Corporation
CUDA PROGRAMMING MODEL
Anatomy of a CUDA C/C++ Application

- **Serial** code executes in a **Host** (CPU) thread
- **Parallel** code executes in many **Device** (GPU) threads across multiple processing elements
void serial_function(… ) {
    ...
}
void other_function(int ... ) {
    ...
}
void saxpy_serial(float ... ) {
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
void main( ) {
    float x;
    saxpy_serial(…);
    ...
}
CUDA C : C with a few keywords

void saxpy_serial(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);

__global__ void saxpy_parallel(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}
// Invoke parallel SAXPY kernel with 256 threads/block
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nbblocks, 256>>>(n, 2.0, x, y);
CUDA Kernels

- Parallel portion of application: execute as a **kernel**
  - Entire GPU executes kernel, many threads

- CUDA threads:
  - Lightweight
  - Fast switching
  - 1000s execute simultaneously

<table>
<thead>
<tr>
<th></th>
<th>Host</th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Executes functions</td>
<td></td>
<td>Executes kernels</td>
</tr>
</tbody>
</table>
CUDA Kernels: Parallel Threads

- A **kernel** is a function executed on the GPU as an array of threads in parallel.
- All threads execute the same code, can take different paths.
- Each thread has an ID:
  - Select input/output data
  - Control decisions

```c
float x = input[threadIdx.x];
float y = func(x);
output[threadIdx.x] = y;
```
CUDA Kernels: Subdivide into Blocks
CUDA Kernels: Subdivide into Blocks

Threads are grouped into **blocks**
CUDA Kernels: Subdivide into Blocks

- Threads are grouped into blocks
- Blocks are grouped into a grid
CUDA Kernels: Subdivide into Blocks

- Threads are grouped into blocks
- Blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads
CUDA Kernels: Subdivide into Blocks

- Threads are grouped into blocks
- Blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads
Kernel Execution

- Each kernel is executed on one device
- Multiple kernels can execute on a device at one time

CUDA-enabled GPU

CUDA thread block

CUDA thread

CUDA core

CUDA Streaming Multiprocessor

CUDA kernel grid

- Each thread is executed by a core
- Each block is executed by one SM and does not migrate
- Several concurrent blocks can reside on one SM depending on the blocks’ memory requirements and the SM’s memory resources
- Each kernel is executed on one device
- Multiple kernels can execute on a device at one time
Thread blocks allow cooperation

- Threads may need to cooperate:
  - Cooperatively load/store blocks of memory all will use
  - Share results with each other or cooperate to produce a single result
  - Synchronize with each other
Thread blocks allow scalability

- Blocks can execute in any order, concurrently or sequentially
- This independence between blocks gives scalability:
  - A kernel scales across any number of SMs
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!

CONCEPTS

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

- Terminology:
  - **Host**: The CPU and its memory (host memory)
  - **Device**: The GPU and its memory (device memory)
```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(void) {
    int *in, *out;
    int *d_in, *d_out;

    // host copies of a, b, c
    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
    cudaMalloc((void**)&d_in,  size);
    cudaMalloc((void**)&d_out, size);

    // Copy to device
    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 Code**: The `stencil_1d` function is written in CUDA and runs on the GPU. It reads input elements into shared memory, applies the stencil, and stores the result.

**Serial Code**: Functions like `fill_ints` and `main` are written in serial code and run on the host CPU. They allocate memory, copy to the device, and manage memory deallocation.

**Parallel FN**: The parallel part of the code is highlighted, showing how data is read from host memory, processed on the GPU, and results are copied back to the host.

**Serial Code**: The serial part of the code includes function calls for memory allocation and deallocation, data transfer between host and device, and handling of memory cleanup.

**Heterogeneous Computing**: The diagram illustrates the parallel execution on the GPU and serial execution on the CPU, emphasizing the cooperation between the two for efficient computation.
Simple Processing Flow

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:

```bash
$ nvcc hello_world.cu
$ a.out
Hello World!
$ 
```
__global__ void mykernel(void) {
    printf("Hello World from device!\n");
}

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

- Two new syntactic elements...
__global__ void mykernel(void) {
    printf(“Hello world from device!\n”);
}

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
Hello World! with Device Code

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!
Access to BigRed2

- ssh <username>@bigred2.uits.iu.edu
- cp -r /N/u/jbentz/BigRed2/oct2/cuda .
- cd cuda
- module load cudatoolkit

Use batch system for job submission
- qsub—submit a job to the queue
- qstat—show all jobs in the queue
- qdel—delete a job from the queue
Exercises: General Instructions (compiling)

- Exercises are in “cuda/exercises” directory
  - Solutions are in “cuda/exercise_solutions” directory

- To compile, use one of the provided makefiles
  ```
  C:
  > make
  ```
Exercises: General Instructions (running)

To run, just execute on the command line

> qsub runit
Hello world

- Login to BigRed 2
- Each coding project in a separate folder in the following dir
  - ~cuda/exercises
- cd cuda/exercises/hello_world
- All dirs have Makefiles for you
- Try building/running the code
  - make
    - Make sure you’ve loaded the cudatoolkit module!
  - qsub runit
Screenshot

```
jbentz@login1:~/oct2/cuda/exercises/hello_world> make
nvcc -arch sm_20 -c kernel.cu
nvcc -arch sm_20 -o x_hello_world kernel.o
jbentz@login1:~/oct2/cuda/exercises/hello_world> qsub runit
123582
jbentz@login1:~/oct2/cuda/exercises/hello_world> cat he
hello.o123582
jbentz@login1:~/oct2/cuda/exercises/hello_world> cat hello.o123582
Hello world from device!
Hello World from Host
Application 1140594 resources: utime ~0s, stime ~1s
==================================
submit_args : submit_args = runit
NIDS : 922
NID Placement : 922/16
jbentz@login1:~/oct2/cuda/exercises/hello_world>
```
Hello World! with Device Code

```c
__global__ void mykernel(void) {
    printf("Hello from device!\n");
}

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

**Output:**

```
$ nvcc hello.cu
$ a.out
Hello from device!
Hello from Host!
$ mykernel() does nothing interesting, 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

\[ a + b = c \]
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
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()`
Returning to our `add()` kernel

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

Let’s take a look at `main()`...

Open exercises/simple_add/kernel.cu

Fill-in missing code as indicated.

- Need to replace “FIXME” with code. Comments should help.
- If something isn’t clear, PLEASE ASK! 😊
Addition on the Device: main()

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()

// 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?

\[
\text{add}<<<1, 1 >>>();
\]

\[
\text{add}<<<N, 1 >>>();
\]

Instead of executing \( \text{add}() \) once, execute \( N \) times in parallel
Vector Addition on the Device

With \texttt{add()} running in parallel we can do vector addition.

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

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

By using \texttt{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**
  - \( c[0] = a[0] + b[0] \)

- **Block 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}()...

Open exercises/simple_add_blocks/kernel.cu

Fill-in missing code as indicated.

- Should be clear from comments where you need to add some code
- Need to replace “FIXME” with the proper piece of code.
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()

// 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()`...

- Open exercises/simple_add_threads/kernel.cu
# 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

CONCEPTS

- Heterogeneous Computing
- 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

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:

```plaintext
int index = threadIdx.x + blockIdx.x * M;
```
Indexing Arrays: Example

Which thread will operate on the red element?

```c
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()`?

- Open `exercises/simple_add_blocks_threads/kernel.cu`
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
    cudaMemcpy((void **)d_a, size);
    cudaMemcpy((void **)d_b, size);
    cudaMemcpy((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()`

```c
// 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:

  ```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;
```
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
Simple Stencil in 1d

- Open exercises/simple_stencil/kernel.cu
- Finish the kernel and the kernel launch
  - Each thread calculates one stencil value
  - Reads 2*RADIUS + 1 values
  - dim3 type: CUDA 3 dimensional struct used for grid/block sizes
- Inserted GPU timers into code to time the execution of the kernel

- Try various sizes of N, RADIUS, BLOCK
- Time a large (over a million) value of N with a RADIUS of 7
Can we do better?

Input elements are read multiple times
- With RADIUS=3, each input element is read seven times!
- Neighbouring threads read most of the same elements.
  - Thread 7 reads elements 4 through 10
  - Thread 8 reads elements 5 through 11

Can we avoid redundant reading of data?
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 (user managed scratch-pad)
  - 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

![Diagram showing halo at boundaries and blockDim.x output elements]
__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;
Simple Stencil 1d with shared memory

cd exercises/simple_stencil_smem/

Run the code. It will build/run without modification.
  If Errors occur, each offending element will be printed to the screen

What is the result with N=10,000 and BLOCK=32?
What is the result with N=10,000 and BLOCK=64?
  Why?
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

- Insert __syncthreads() into the kernel in the proper location
- Compare timing of previous simple stencil with the current shared memory implementation for same (large N) and BLOCK=512
Stencil Kernel

```c
__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:
```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```
Review (2 of 2)

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
- Using large shared mem size impacts number of blocks that can be scheduled on an SM (48K total smem size)

Use **__syncthreads()** as a barrier
- Use to prevent data hazards
Thank you!