GPU Computing with CUDA

Mathias Wagner

based on training material from NVIDIA
GPU Computing at IU

#84 on current (Nov. 2014) Top 500 list
## Big Red II - Specs

<table>
<thead>
<tr>
<th>System configuration</th>
<th>Aggregate information</th>
<th>Per node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine type</td>
<td>Hybrid (x86_64/NVIDIA Kepler)</td>
<td></td>
</tr>
<tr>
<td>Operating system</td>
<td>Cray Linux Environment (based on SUSE Linux SLES 11)</td>
<td></td>
</tr>
<tr>
<td>Memory model</td>
<td>Distributed</td>
<td></td>
</tr>
<tr>
<td>Processor cores</td>
<td>21.824</td>
<td>32 (compute) / 16 (GPU)</td>
</tr>
<tr>
<td>CPUs</td>
<td>1.364</td>
<td>2 (compute) / 1 (GPU)</td>
</tr>
<tr>
<td>Nodes</td>
<td>1,020 (344 compute + 676 GPU)</td>
<td></td>
</tr>
<tr>
<td>Rmax</td>
<td>596.4 teraFLOPS</td>
<td>0.884 teraFLOPS</td>
</tr>
<tr>
<td>Rpeak</td>
<td>1.006 petaFLOPS</td>
<td>320 gigaFLOPS (compute)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1,317 gigaFLOPS (GPU)</td>
</tr>
<tr>
<td>RAM</td>
<td>43,648 GB</td>
<td>64 GB (compute)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>32 GB (GPU)</td>
</tr>
</tbody>
</table>
The accelerator age

- High-Performance Computing increasingly dominated by accelerators (GPUs, Intel MIC, ...)
- Increasing number of systems in TOP 500 use accelerators
- Accelerators provide about 40% of performance
US to Build Two Flagship Supercomputers

Partnership for Science
100-300 PFLOPS Peak Performance
10x in Scientific Applications
2017

Major Step Forward on the Path to Exascale
GPU as accelerators

- typical CPU data
  - performance ~ 200 GFlops
  - bandwidth ~ 50 GB/s
  - power consumption ~ 125W
  - price ~ $500 - 1000

- typical GPU data
  - performance ~ 4 TFlops
  - bandwidth ~ 250 GB/s
  - power consumption ~ 250W
  - price ~ $2000 - 6000 (Tesla)
    $ 500-1000 (GTX)
Low Latency or High Throughput?

CPU
- Optimized for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution

GPU
- Optimized for data-parallel, throughput computation
- Architecture tolerant of memory latency
- More transistors dedicated to computation
Low Latency or High Throughput?

- **CPU** architecture must **minimize latency** within each thread.
- **GPU** architecture **hides latency** with computation.
Typical GPU system

different transfer rates and latencies

- GPU Mem. B/W / CPU Mem. B/W ~6.9x
- GPU Peak Flops (SP) / CPU Peak Flops (SP) ~8.4x
- PCIe Gen2 serious bottleneck for multi-GPU

'Speeds and Feeds' come from comparing a 6 core Westmere EX, running at 3.33 GHz, with a Tesla M2090 - using respective datasheets.

- PCIe 2 'Network' 8+8 GB/s per x16 bus
- GPU Memory buses ~177 GB/s
- multicore CPU: Westmere EX, 6 cores, 3.3 GHz -> 159 GFlops
- Off-node 'Network' ~5GB/s (40Gb/s)
- GPU: M2090: 1.3 TFlop (SP)

JLab 10G cluster
Thursday, July 14, 2011
Small Changes, Big Speed-up

Use GPU to Parallelize Compute-Intensive Functions

Rest of Sequential CPU Code
3 Ways to Accelerate Applications

- Libraries
  - “Drop-in” Acceleration

- OpenACC Directives
  - Easily Accelerate Applications

- Programming Languages
  - Maximum Performance
GPU Accelerated Libraries
“Drop-in” Acceleration for your Applications

- NVIDIA cuBLAS
- NVIDIA cuRAND
- NVIDIA cuSPARSE
- NVIDIA NPP
- GPU VSIPL
- CULA tools
- MAGMA
- NVIDIA cuFFT
- Rogue Wave Software
- CUSP
- ArrayFire
- Thrust

Vector Signal Image Processing
GPU Accelerated Linear Algebra
Matrix Algebra on GPU and Multicore
NVIDIA cuFFT
IMSL Library
Sparse Linear Algebra
Building-block Algorithms
C++ Templated Parallel Algorithms
OpenACC Directives

Program myscience
... serial code ...
!$acc kernels
do k = 1,n1
do i = 1,n2
... parallel code ...
enddo
enddo
!$acc end kernels
... End Program myscience

CPU
GPU

Simple Compiler hints
Compiler Parallelizes code
Works on many-core GPUs & multicore CPUs

Your original Fortran or C code
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];
}

// Perform SAXPY on 1M elements
saxpy_serial(4096*256, 2.0, x, y);

// Standard C Code

__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];
}

// Perform SAXPY on 1M elements
saxpy_parallel<<<4096,256>>>(n,2.0,x,y);

// Parallel C Code

## Recommended Approaches

<table>
<thead>
<tr>
<th>Category</th>
<th>Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numerical analytics</td>
<td>MATLAB, Mathematica, LabVIEW</td>
</tr>
<tr>
<td>Fortran</td>
<td>OpenACC, CUDA Fortran</td>
</tr>
<tr>
<td>C</td>
<td>OpenACC, CUDA C, OpenCL</td>
</tr>
<tr>
<td>C++</td>
<td>Thrust, CUDA C++</td>
</tr>
<tr>
<td>Python</td>
<td>PyCUDA</td>
</tr>
<tr>
<td>C#</td>
<td>GPU.NET</td>
</tr>
</tbody>
</table>
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
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

Threads
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
Kernel Execution

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

CUDA - enabled GPU

- 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
Memory hierarchy

- Thread:
  - Registers
Memory hierarchy

- Thread:
  - Registers
  - Local memory
Memory hierarchy

- Thread:
  - Registers
  - Local memory

- Block of threads:
  - Shared memory
Memory hierarchy

- **Thread:**
  - Registers
  - Local memory

- **Block of threads:**
  - Shared memory

- **All blocks:**
  - Global memory
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)
Heterogeneous Computing

```
#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];
    }

    // 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;
}

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

int main()
{
    int *in, *out;
    // host copies of a, b, c
    int *d_in, *d_out;
    // device copies of a, b, c

    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
    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;
}
```
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
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
#include <stdio.h>

__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

```c
__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`
Hello World! with Device Code

```c
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

\[
\begin{align*}
  a & \quad + \quad b \\
& \quad = \\
& \quad c
\end{align*}
\]
Addition on the Device

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
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()`
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;
```
Since CUDA 6: unified memory

**CPU Code**

```c
void sortfile(FILE *fp, int N) {
    char *data;
    data = (char *)malloc(N);
    fread(data, 1, N, fp);
    qsort(data, N, 1, compare);
    use_data(data);
    free(data);
}
```

**CUDA 6 Code with Unified Memory**

```c
void sortfile(FILE *fp, int N) {
    char *data;
    cudaMallocManaged(&data, N);
    fread(data, 1, N, fp);
    qsort<<<...>>>(data, N, 1, compare);
    cudaDeviceSynchronize();
    use_data(data);
    cudaFree(data);
}
```
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 \text{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:

<table>
<thead>
<tr>
<th>Block 0</th>
<th>Block 1</th>
<th>Block 2</th>
<th>Block 3</th>
</tr>
</thead>
</table>
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()...
#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_device.c:

```c
#include <stdio.h>
#include <cuda_runtime.h>

// Define constants
#define N 256
#define size 8

// Main function
int 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()`...
#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);
// 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:

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

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

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

```cpp
__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:

```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...
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];
  }
}
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;
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()

```c
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;
```
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

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
Reporting Errors

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
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()`
Questions?
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
Textures

- Read-only object
  - Dedicated cache

- Dedicated filtering hardware
  (Linear, bilinear, trilinear)

- Addressable as 1D, 2D or 3D

- Out-of-bounds address handling
  (Wrap, clamp)
Topics we skipped

We skipped some details, you can learn more:
- CUDA Programming Guide
- CUDA Zone - tools, training, webinars and more

Need a quick primer for later:
- Multi-dimensional indexing
- Textures