GPU Computing with CUDA
Lecture 2 - CUDA Memories

Christopher Cooper
Boston University
August, 2011
UTFSM, Valparaíso, Chile
Outline of lecture

- Recap of Lecture 1
- Warp scheduling
- CUDA Memory hierarchy
- Introduction to the Finite Difference Method
Recap

GPU

- A real alternative in scientific computing
- Massively parallel commodity hardware
  - Market driven development
  - Cheap!

CUDA

- NVIDIA technology capable of programming massively parallel processors with general purpose
- C/C++ with extensions
Recap

GPU chip

- Designed for high throughput tasks
- Many simple cores
- Fermi:
  - 16 SMs
  - 32 SPs (cores)
  - Shared, registers, cache
  - SFU, CU
  - 1TFLOP peak throughput
  - 144GB/s peak bandwidth
Recap

- Programming model
  - Based on **threads**
  - Thread hierarchy: grouped in **thread blocks**
  - Threads in a thread block can communicate

- Challenge: parallel thinking
  - Data parallelism
  - Load balancing
  - Conflicts
  - Latency hiding
CUDA - Programming model

- Connecting the hardware and the software

Software representation

Hardware
CUDA - Programming model

- Connecting the hardware and the software

![Diagram showing software representation and hardware block]

- Thread
- Thread block

Software representation

Hardware
CUDA - Programming model

- Connecting the hardware and the software
CUDA - Programming model

- Connecting the hardware and the software

Software representation

Hardware
CUDA - Programming model

- Connecting the hardware and the software
How a Streaming Multiprocessor (SM) works

- CUDA Threads are grouped in thread blocks
  - All threads of the same thread block are executed in the same SM at the same time
    - SMs have shared memory, then threads within a thread block can communicate
  - The entirety of the threads of a thread block must be executed before there is space to schedule another thread block
How a Streaming Multiprocessor (SM) works

- Hardware schedules thread blocks onto available SMs
  - No guarantee of order of execution
  - If a SM has more resources the hardware will schedule more blocks
How a Streaming Multiprocessor (SM) works

- Hardware schedules thread blocks onto available SMs
  - No guarantee of order of execution
  - If a SM has more resources the hardware will schedule more blocks
How a Streaming Multiprocessor (SM) works

- Hardware schedules thread blocks onto available SMs
  - No guarantee of order of execution
  - If a SM has more resources the hardware will schedule more blocks

Diagram:
- SM
  - Block 1
  - Block 7
  - Block 100
  - Block 15
How a Streaming Multiprocessor (SM) works

- Hardware schedules thread blocks onto available SMs
  - No guarantee of order of execution
  - If a SM has more resources the hardware will schedule more blocks
How a Streaming Multiprocessor (SM) works

- Hardware schedules thread blocks onto available SMs
  - No guarantee of order of execution
  - If a SM has more resources the hardware will schedule more blocks

Diagram:

```
Block 7
Block 15
SM
Block 100
```
How a Streaming Multiprocessor (SM) works

- Hardware schedules thread blocks onto available SMs
  - No guarantee of order of execution
  - If a SM has more resources the hardware will schedule more blocks
Warps

- Inside the SM, threads are launched in groups of 32 called **warps**
  - Warps share the control part (warp scheduler)
  - At any time, only one warp is executed per SM
  - Threads in a warp will be executing the same instruction
  - Half warps for compute capability 1.X

- Fermi:
  - Maximum number of active threads $1024 \times 8 \times 32 = 262144$
Warps

- Inside the SM, threads are launched in groups of 32 called **warps**
  - Warps share the control part (warp scheduler)
  - At any time, only one warp is executed per SM
  - Threads in a warp will be executing the same instruction
  - Half warps for compute capability 1.X

- Fermi:
  - Maximum number of active threads $1024 \times 8 \times 32 = 262144$
Warps

- Inside the SM, threads are launched in groups of 32 called **warps**
  - Warps share the control part (warp scheduler)
  - At any time, only one warp is executed per SM
  - Threads in a warp will be executing the same instruction
  - Half warps for compute capability 1.X

- Fermi:
  - Maximum number of active threads $1024 \times 8 \times 32 = 262144$
Warps

- Inside the SM, threads are launched in groups of 32 called **warps**
  - Warps share the control part (warp scheduler)
  - At any time, only one warp is executed per SM
  - Threads in a warp will be executing the same instruction
  - Half warps for compute capability 1.X

- Fermi:
  - Maximum number of active threads $1024 \times 8 \times 32 = 262144$
Warp designation

- Hardware separates threads of a block into warps
  - All threads in a warp correspond to the same thread block
  - Threads are placed in a warp sequentially
  - Threads are scheduled in warps
Warp scheduling

- The SM implements a zero-overhead warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible warps are selected for execution on a prioritized scheduling policy
  - All threads in a warp execute the same instruction
Warp scheduling

- Fermi
  - Double warp scheduler
  - Each SM has two warp schedulers and two instruction units
Memory hierarchy

- CUDA works in both the CPU and GPU
  - One has to keep track of which memory is operating on (host - device)
  - Within the GPU there are also different memory spaces

Each thread can:
- Read/write per-thread registers
- Read/write per-thread local memory
- Read/write per-block shared memory
- Read/write per-grid global memory
- Read/only per-grid constant memory

Figure 4.2 GeForce 8800GTX Implementation of CUDA Memories
Memory hierarchy

- Global memory
  - Main GPU memory
  - Communicates with host
  - Can be seen by all threads
  - Order of GB
  - Off chip, slow (~400 cycles)

```c
__device__ float variable;
```
Memory hierarchy

- Shared memory
  - Per SM
  - Seen by threads of the same thread block
  - Order of kB
  - On chip, fast (~4 cycles)

- Registers
  - Private to each thread
  - On chip, fast

```c
__shared__ float variable;

float variable;
```
Memory hierarchy

- Local memory
  - Private to each thread
  - Off chip, slow
  - Register overflows

- Constant memory
  - Read only
  - Off chip, but fast (cached)
  - Seen by all threads
  - 64kB with 8kB cache

```c
float variable [10];
__constant__ float variable;
```
Memory hierarchy

- Texture memory
  - Seen by all threads
  - Read only
  - Off chip, but fast (cached) if cache hit
  - Cache optimized for 2D locality
  - Binds to global

```c
texture<type, dim> tex_var;
cudaChannelFormatDesc();
cudaBindTexture2D(...);
tex2D(tex_var, x_index, y_index);
```
Memory hierarchy

- Texture memory
  - Seen by all threads
  - Read only
  - Off chip, but fast (cached) if cache hit
  - Cache optimized for 2D locality
  - Binds to global

```
texture<type, dim> tex_var;
cudaChannelFormatDesc();
cudaBindTexture2D(...);
tex2D(tex_var, x_index, y_index);
```
Memory hierarchy

- Texture memory
  - Seen by all threads
  - Read only
  - Off chip, but fast (cached) if cache hit
  - Cache optimized for 2D locality
  - Binds to global

```cpp
texture<type, dim> tex_var;
cudaChannelFormatDesc();
cudaBindTexture2D(...);
tex2D(tex_var, x_index, y_index);
```
Memory hierarchy

- Texture memory
  - Seen by all threads
  - Read only
  - Off chip, but fast (cached) if cache hit
  - Cache optimized for 2D locality
  - Binds to global

```c++
texture<type, dim> tex_var;
cudaChannelFormatDesc();
cudaBindTexture2D(...);
tex2D(tex_var, x_index, y_index);
```

- Initialize
- Options
- Bind
Memory hierarchy

- Texture memory
  - Seen by all threads
  - Read only
  - Off chip, but fast (cached) if cache hit
  - Cache optimized for 2D locality
  - Binds to global

```c
texture<type, dim> tex_var;
cudaChannelFormatDesc();
cudaBindTexture2D(...);
tex2D(tex_var, x_index, y_index);
```

Initialize  Options  Bind  Fetch
## Memory hierarchy

<table>
<thead>
<tr>
<th>Memory</th>
<th>Location on/off chip</th>
<th>Cached</th>
<th>Access</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>On</td>
<td>n/a</td>
<td>R/W</td>
<td>1 thread</td>
<td>Thread</td>
</tr>
<tr>
<td>Local</td>
<td>Off</td>
<td>†</td>
<td>R/W</td>
<td>1 thread</td>
<td>Thread</td>
</tr>
<tr>
<td>Shared</td>
<td>On</td>
<td>n/a</td>
<td>R/W</td>
<td>All threads in block</td>
<td>Block</td>
</tr>
<tr>
<td>Global</td>
<td>Off</td>
<td>†</td>
<td>R/W</td>
<td>All threads + host</td>
<td>Host allocation</td>
</tr>
<tr>
<td>Constant</td>
<td>Off</td>
<td>Yes</td>
<td>R</td>
<td>All threads + host</td>
<td>Host allocation</td>
</tr>
<tr>
<td>Texture</td>
<td>Off</td>
<td>Yes</td>
<td>R</td>
<td>All threads + host</td>
<td>Host allocation</td>
</tr>
</tbody>
</table>

*Cached only on devices of compute capability 2.x.

![Memory hierarchy diagram](image)
Memory hierarchy

- Case of Fermi
  - Added an L1 cache to each SM
  - Shared + cache = 64kB:
    ‣ Shared = 48kB, cache = 16kB
    ‣ Shared = 16kB, cache = 48kB
Memory hierarchy

- Use memory strategically
  - Read only: `__constant__` (fast)
  - Read/write and communicate within a block: `__shared__` (fast and communication)
  - Read/write inside thread: registers (fast)
  - Data locality: texture
Resource limits

- Number of thread blocks per SM at the same time is limited by
  - Shared memory usage
  - Registers
  - No more than 8 thread blocks per SM
  - Number of threads
**Resource limits - Examples**

Number of blocks

How big should my blocks be? 8x8, 16x16 or 64x64?

- **8x8**
  8*8 = 64 threads per block, 1024/64 = 16 blocks. An SM can have up to 8 blocks: only 512 threads will be active at the same time.

- **16x16**
  16*16 = 256 threads per block, 1024/256 = 4 blocks. An SM can take all blocks, then all 1024 threads will be active and achieve full capacity unless other resource overrule.

- **64x64**
  64*64 = 4096 threads per block: doesn’t fit in a SM
Resource limits - Examples

Registers

We have a kernel that uses 10 registers. With 16x16 block, how many blocks can run in G80 (max 8192 registers per SM, 768 threads per SM)?

10*16*16 = 2560. SM can hold 8192/2560 = 3 blocks, meaning we will use 3*16*16 = 768 threads, which is within limits.

If we add one more register, the number of registers grows to 11*16*16 = 2816. SM can hold 8192/2816 = 2 blocks, meaning we will use 2*16*15 = 512 threads. Now as less threads are running per SM is more difficult to have enough warps to have the GPU always busy!
Introduction to Finite Difference Method

- Throughout the course many examples will be taken from Finite Difference Method
- Also, next lab involves implementing a stencil code
- Just want to make sure we’re all in the same page!
Introduction to Finite Difference Method

- Finite Difference Method (FDM) is a numerical method for solution of differential equations
- Domain is discretized with a mesh
- Derivative at a node are approximated by the linear combination of the values of points nearby (including itself)

\[
\frac{\partial y}{\partial x_i} \approx \frac{y_i - y_{i-1}}{x_i - x_{i-1}}
\]

Example using first order one sided difference
FDM - Accuracy and order

- From Taylor’s polynomial, we get

\[
f(x_0 + h) = f(x_0) + f'(x_0) \cdot h + R_1(x)
\]

\[
f'(x_0) = \frac{f(a + h) - f(a)}{h} - R(x)
\]

- Sources of error
  - Roundoff (Machine)
  - Truncation (R(x)): gives the order of the method
FDM - Stability

- Stability of the algorithm may be conditional

\[ \frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \]

- Explicit method

\[ \frac{u_{j}^{n+1} - u_{j}^{n}}{k} = \frac{u_{j+1}^{n} - 2u_{j}^{n} + u_{j-1}^{n}}{h^2} \text{ Conditionally stable } \frac{k}{h^2} < 0.5 \]

- Implicit

\[ \frac{u_{j}^{n+1} - u_{j}^{n}}{k} = \frac{u_{j+1}^{n+1} - 2u_{j}^{n+1} + u_{j-1}^{n+1}}{h^2} \text{ Unconditionally stable} \]

- Crank Nicolson

\[ \frac{u_{j}^{n+1} - u_{j}^{n}}{k} = \frac{1}{2} \left( \frac{u_{j+1}^{n} - 2u_{j}^{n} + u_{j-1}^{n}}{h^2} + \frac{u_{j+1}^{n+1} - 2u_{j}^{n+1} + u_{j-1}^{n+1}}{h^2} \right) \text{ Unconditionally stable} \]
FDM - 2D example

- We’re going to be working with this example in the labs

- Explicit diffusion

\[
\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}
\]

\[
u^{n+1}_{i,j} = u^n_{i,j} + \frac{\alpha k}{h^2} (u^n_{i,j+1} + u^n_{i,j-1} + u^n_{i+1,j} + u^n_{i-1,j} - 4u^n_{i,j})
\]
FDM on the GPU

- Mapping the physical problem to the software and hardware
- Each thread will operate on one node: node indexing groups them naturally!

\[ u_{i,j}^{n+1} = u_{i,j}^n + \frac{\alpha k}{h^2} (u_{i,j+1}^n + u_{i,j-1}^n + u_{i+1,j}^n + u_{i-1,j}^n - 4u_{i,j}^n) \]
FDM on the GPU

- Mapping the physical problem to the software and hardware
- Each thread will operate on one node: node indexing groups them naturally!

\[ u_{i,j}^{n+1} = u_{i,j}^{n} + \frac{\alpha k}{h^2} (u_{i,j+1}^{n} + u_{i,j-1}^{n} + u_{i+1,j}^{n} + u_{i-1,j}^{n} - 4u_{i,j}^{n}) \]
FDM on the GPU

- Mapping the physical problem to the software and hardware

- Each thread will operate on one node: node indexing groups them naturally!

\[ u^{n+1}_{i,j} = u^n_{i,j} + \frac{\alpha k}{h^2} \left( u^n_{i,j+1} + u^n_{i,j-1} + u^n_{i+1,j} + u^n_{i-1,j} - 4u^n_{i,j} \right) \]