Lecture 7

CUDA Performance Programming
Divergence
Matrix Multiplication
Announcements

• Re-run your A1 results on Forge on up to 16 cores
• Groups of 3: optimizations as described in the IPDPS 2010 paper by Kamil et al.
Tips for the lab writeups

- Develop the discussions: goals, methods, commentary, speculation, reflection
- Don’t present data that adds no new information
- Plotting
  - Put related information together in the same plot
  - How will your plots look in Black and White?
  - Logarithmic scales
Projects

• Counts for 60% of your grade
• Complete in 3 weeks
• See the (growing) list of projects at cseweb.ucsd.edu/classes/wi12/cse260-a/Projects/ProjectList.html
• A limited number of self-proposed projects, requires a proposal: 2/24
• Progress report: 3/5
• Final Report: 3/15
Today’s lecture

• Digging deeper into the GPU’s hardware
• Matrix Multiplication with Global Memory
Warp scheduling on Fermi

- Threads assigned to an SM in units of a thread block
- Can be multiple blocks per SM
- Blocks are divided into warps of 32 (SIMD) threads, a schedulable unit
  - A warp becomes eligible for execution when all its operands are available
  - Dynamic instruction reordering: eligible warps selected for execution using a prioritized scheduling policy
  - All threads in a Warp execute the same instruction
- Multiple warps simultaneously active, hiding data transfer delays
- All registers in all the warps are available, 0 overhead scheduling
- Hardware is free to assign blocks to any SM
- How does scheduling work?
Scoreboarding

• Keep track of all register operands of all instructions in the Instruction Buffer
  ♦ Instruction becomes ready after the needed values are written
  ♦ Eliminates hazards
  ♦ Ready instructions are eligible for issue

• Decouples the Memory/Processor pipelines
  ♦ Threads can issue instructions until the scoreboard prevents issue
  ♦ Allows Memory/Processor ops to proceed in parallel with other waiting Memory/Processor ops

Instruction:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>TB1</td>
<td>W1</td>
<td>TB2</td>
<td>W1</td>
<td>TB3</td>
<td>W2</td>
</tr>
</tbody>
</table>

Time

TB = Thread Block, W = Warp

Static scheduling limits performance

- The ADDD instruction is stalled on the DIVide ..
- stalling further instruction issue, e.g. the SUBD

\[
\begin{align*}
\text{DIV} & \quad F0, \quad F2, \quad F4 \\
\text{ADDD} & \quad F10, \quad F0, \quad F8 \\
\text{SUBD} & \quad F12, \quad F8, \quad F14
\end{align*}
\]

- But SUBD doesn’t depend on ADDD or DIV
- If we have two adder/subtraction units, one will sit idle uselessly until the DIV finishes

Dynamic scheduling

• Idea: modify the pipeline to permit instructions to execute as soon as their operands become available
• This is known as *out-of-order execution (classic dataflow)*
• The SUBD can now proceed normally
• Complications: dynamically scheduled instructions also complete out of order
Dynamic scheduling splits the ID stage

- Issue sub-stage
  - Decode the instructions
  - Check for structural hazards
- Read operands substage
  - Wait until there are no data hazards
  - Read operands
- We need additional registers to store pending instructions that aren’t ready to execute
Consequences of a split ID stage

- We distinguish between the time when an instruction begins execution, and when it completes.
- Previously, an instruction stalled in the ID stage, and this held up the entire pipeline.
- Instructions can now be in a suspended state, neither stalling the pipeline, nor executing.
- They are waiting on operands.
Two schemes for dynamic scheduling

- **Scoreboard**
  - CDC 66000
- **Tomasulo’s algorithm**
  - IBM 360/91
- We’ll vary the number of functional units, their latency, and functional unit pipelining
What is a scoreboard?

- A technique that allows instructions to execute out of order...
  - So long as there are sufficient resources and
  - No data dependencies
- The goal of scoreboard
  - Maintain an execution rate of one instruction per clock cycle
Multiple execution pipelines in DLX with scoreboard.

<table>
<thead>
<tr>
<th>Unit</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>0</td>
</tr>
<tr>
<td>Memory</td>
<td>1</td>
</tr>
<tr>
<td>FP Add</td>
<td>2</td>
</tr>
<tr>
<td>FP Multiply</td>
<td>10</td>
</tr>
<tr>
<td>FP Div</td>
<td>40</td>
</tr>
</tbody>
</table>
What are the requirements?

• Responsibility for instruction issue and execution, including hazard detection
• Multiple instructions must be in the EX stage simultaneously
• Either through pipelining or multiple functional units
• DLX has: 2 multipliers, 1 divider, 1 integer unit (memory, branch, integer arithmetic)
How is a scoreboard implemented?

• A centralized bookkeeping table
• Tracks instructions, along with register operand(s) they depend on and which register they modify
• Status of result registers (who is going to write to a given register)
• Status of the functional units
How does it work?

- As each instruction passes through the scoreboard, construct a description of the data dependencies (Issue)
- Scoreboard determines when the instruction can read operands and begin execution
- If the instruction can’t begin execution, the scoreboard keeps a record, and it listens for one the instruction *can* execute
- Also controls when an instruction may write its result
- All hazard detection is centralized
Warp scheduling on Fermi

- 2 schedulers find an eligible warp
  - Each issues 1 instruction per cycle
  - Issue selection based on round-robin/age of warp
  - Odd/even warps
  - Warp scheduler can issue instruction to \( \frac{1}{2} \) the cores, each scheduler issues: 1 (2) instructions for capability 2.0 (2.1)
  - Scheduler must issue the instruction over 2 clock cycles for an integer or floating-point arithmetic instruction
  - Only 1 double prec instruction at a time

- All registers in all the warps are available, 0 overhead scheduling
- Overhead may be different when switching blocks
Dynamic behavior – resource utilization

- Threads assigned to an SM in units of a thread block
- Can be multiple blocks per SM
- Each vector core (SM): 1024 thread slots and 8 block slots
- Hardware partitions slots into blocks at run time, accommodates different processing capacities
- Registers are split dynamically across all blocks assigned to the vector core
- A register is private to a single thread within a single block
Occupancy

• A minimum number of warps needed to hide memory latency
• Occupancy: \( \frac{\text{# active warps}}{\text{max # warps supported by vector unit}} \)
• Limited by vector unit resources
  - Amount of shared memory
  - Number of registers
  - Maximum number of threads
• Consider a kernel (16x16 block size)
  - Shared memory/block = 2648 bytes
  - Reg/thread=38 \([38*256 = 9728 < 16k]\)
  - # available registers is the limiting factor
• Tradeoff: more blocks with fewer threads or more threads with fewer blocks
  - Locality: want small blocks of data (and hence more plentiful warps) that fit into fast memory
  - Register consumption
Thread divergence

- All the threads in a warp execute the same instruction
- Different control paths are serialized
Example of thread Divergence

• Divergence when predicate is a function of the threadId, branch granularity > warp size
  if (threadId < 2) {} {} 

• No divergence if all follow the same path within a warp
  branch granularity > warp size
  if (threadId / WARP_SIZE < 2) {} {} 

• But there are different control paths within the block
A perspective on programming

- Vector length is not built into the instruction: we can run a program on a GPUs supporting different vector lengths
- A thread block is a single thread of vector instructions with a programmable vector length (the block size)
- The number of warps in a block is configurable
Strip mining

- Partitioning the iteration space into chunks

\[
\text{for } i = 0 \text{ to } N-1 \\
\quad a[i] = b[i] + c[i];
\]

\[
\text{for } j = 0 \text{ to } N-1 \text{ by } VL \\
\quad \text{for } i = j \text{ to } \min(N, j+VL) - 1 \\
\quad \quad a[i] = b[i] + c[i];
\]

```c
int idx = blockIdx.x*blockDim.x + threadIdx.x;
if (idx<N) a[idx] = a[idx]+1.f;
```
Strip mining on the GPU

• Partitioning long vectors into warps corresponds to strip-mining into *independent* instruction streams

• Traditionally: render independent instructions in the *same* instruction stream

```c
int idx = blockIdx.x*blockDim.x + threadIdx.x;
if (idx<N) a[idx] = a[idx]+1.f;
```

```c
for j = 0 to N-1 by VL
  for i = j to min(N, j+VL) − 1
    a[i] = b[i] + c[i];
```
Data motion cost

• Communication performance is a major factor in determining the overall performance of an application

• The $\alpha-\beta$ model: $\alpha + \beta^{-1}$

  $n = \text{message length}$

  $\alpha = \text{message startup time}$

  $\beta_{\infty} = \text{peak bandwidth (bytes / second)}$

<table>
<thead>
<tr>
<th>Machine</th>
<th>$\beta_{\infty}$ (Dev)</th>
<th>H-D</th>
<th>D-H</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forge</td>
<td>103 GB/s</td>
<td>2.3</td>
<td>1.3</td>
</tr>
<tr>
<td>Lilliput</td>
<td>73.6</td>
<td>3.3</td>
<td>2.8</td>
</tr>
<tr>
<td>CseClass01</td>
<td>122</td>
<td>3.4</td>
<td>2.7</td>
</tr>
<tr>
<td>CseClass04</td>
<td>56.1</td>
<td>5.2</td>
<td>4.1</td>
</tr>
</tbody>
</table>

As reported by bandwidthTest
Consequences of data motion cost

- Consider saxpy $z[i] = a*x[i] + y[i]$
- This is a bandwidth bound kernel
- Running time under the $\alpha-\beta$ model: $\alpha + \beta^{-1}$
  $\alpha = 4 \ \mu s$
  $\beta\_\infty = 127 \ \text{GB/sec}$
- Flop rate bounded by $(2n \ \text{flops}/12n \ \text{bytes}) * \beta\_\infty$
  - 27 Gflops/sec
- $N^{1/2}$ Half bandwidth point: $N \approx 42,000$
  - Matrix multiplication takes 4 $\mu s$
  - But the largest matrix that fits into memory is 1GB $\sim (16K)^2$
  - Consequence: saxpy takes constant time to run