Lecture 9

Divergence, scheduling and floating point
Announcements

• Mac Mini lab (APM 2402)
  ♦ Friday at 4pm to 6pm
  ♦ Next week: Tues and Fri 4pm to 6pm

• Project proposals due on November 9
Projects

• Counts for 60% of your grade
• Complete in 3 weeks
• See the (growing) list of projects at cseweb.ucsd.edu/classes/fa12/cse260-b/Projects/ProjectList.html
• CUDA, MPI or CUDA + MPI

• Select your project from the list by 11/9
  • A limited number of self-proposed projects, requires a proposal

• Progress report: 11/21 (Weds)
• Final Report: 12/7 (Friday)
Project Proposals

• Due 11/9
  ♦ What are the goals of your project? Are they realistic?
  ♦ What are your hypotheses?
  ♦ What is your experimental method for proving or disproving your hypotheses?
  ♦ What experimental result(s) do you need to demonstrate?
  ♦ What would be the significance of those results?
  ♦ What code will you need to implement? What software packages or previously written software will use?
  ♦ A tentative division of labor among the team members
  ♦ A preliminary list of milestones—with completion dates
Projects!

- Stencil method in 3 dimensions
- Multigrid
- Communication avoiding matrix multiplication (MPI)
- Algorithm based fault tolerance (MPI)
- 3D Fast Fourier Transform (MPI or CUDA)
- Particle simulation (MPI)
- Groups of 3 will do a more ambitious project
  - MPI projects can add communication overlap
  - MPI + CUDA
- Propose your own
- Make your choice by 11/9

www-cse.ucsd.edu/classes/fa12/cse260-b/Projects/ProjectList.html
Today’s lecture

• Thread divergence
• Scheduling
• Floating Point
Thread divergence

- All the threads in a warp execute the same instruction
- Different control paths are serialized
if (threadIdx >= 2)
    a=100;
else
    a=-100;

compare threadIdx,2

Mary Hall
Divergence example

if (threadIdx >= 2)
  a=100;
else
  a=-100;

Mary Hall
Divergence example

if (threadIdx >= 2)
a=100;
else
a=-100;

Mary Hall

©2012 Scott B. Baden /CSE 260/ Fall 2012
Thread Divergence

- All the threads in a warp execute the same instruction
- Different control paths are serialized
- *Divergence* when a predicate is a function of the threadId
  
  \[
  \text{if (threadId < 2) \{ \}}
  \]

- No divergence if all follow the same path within a warp
  
  \[
  \text{if (threadId / WARP\_SIZE < 2) \{ \}}
  \]

- We can have different control paths within the thread block

©2012 Scott B. Baden /CSE 260/ Fall 2012
Example – reduction – thread divergence

Thread 0   Thread 2   Thread 4   Thread 6   Thread 8   Thread 10

0   1   2   3   4   5   6   7   8   9   10   11

0+1   2+3   4+5   6+7   8+9   10+11

0...3   4..7   8..11

0..7   8..15

©2012 Scott B. Baden /CSE 260/ Fall 2012
The naïve code

```c
__global__ void reduce(int *input, unsigned int N, int *total) {
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

    __shared__ int x[BSIZE];
    x[tid] = (i < N) ? input[i] : 0;
    __syncthreads();

    for (unsigned int stride = 1; stride < blockDim.x; stride *= 2) {
        __syncthreads();
        if (tid % (2*stride) == 0)
            x[tid] += x[tid + stride];
    }

    if (tid == 0) atomicAdd(total, x[tid]);
}
```
Reducing divergence and avoiding bank conflicts
The improved code

- No divergence until stride < 32
- All warps active when stride ≥ 32

```c
for (stride = blockDim.x/2;
    stride > 1;
    stride /= 2) {
    __syncthreads();
    if (tid < stride)
        x[tid] += x[tid + stride];
}
```

```c
for (stride = 1;
    stride < blockDim.x;
    stride *= 2) {
    __syncthreads();
    if (tid % (2*stride) == 0)
        x[tid] += x[tid + stride];
}
```
Predication on Fermi

- All instructions support predication in 2.x
- Condition code or *predicate* per thread:
  set to true or false
- Execute only if the predicate is true
  
  \[
  \text{if (x>1)} \\
  y = 7;
  \]

  \[
  \text{test = (x>1)} \\
  \text{test: y=7}
  \]

- Compiler replaces a branch instruction with predicated instructions only if the number of instructions controlled by branch condition is not too large
- If the compiler predicts too many divergent warps….
  threshold = 7, else 4
Concurrency – Host & Device

- Nonbocking operations
  - Kernel launch
  - Device ↔ {Host, Device}
  - Async memory copies
- Multiple kernel invocations: certain capability 2.x devices
- CUDA Streams (§3.2.6.5), with limitations
  ```
  cudaStream_t stream[2];
  for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
  for (int i = 0; i < 2; ++i)
    Kernel<<<100, 512, 0, stream[i]>>> ( ... );
  ```
Today’s lecture

• Thread divergence
• Scheduling
• Floating Point
Warp scheduling on Fermi

• 2 schedulers find an eligible warp
  ♦ Each issues 1 instruction per cycle
  ♦ Dynamic instruction reordering: eligible
    warps selected for execution using a
    prioritized scheduling policy
  ♦ Issue selection: round-robin/age of warp
  ♦ Odd/even warps
  ♦ Warp scheduler can issue instruction to ½
    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
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

TB1, W1 stall — TB2, W1 stall — TB3, W2 stall

TB1 W1
TB2 W1
TB3 W1
TB3 W2
TB2 W1
TB1 W1
TB1 W2
TB1 W3
TB1 W2
TB3 W2

Instruction:

1 2 3 4 5 6 1 2 1 2 3 4 7 8 1 2 1 2 3 4

—Time—

TB = Thread Block, W = Warp

©2012 Scott B. Baden /CSE 260/ Fall 2012
Static scheduling limits performance

• The ADDD instruction is stalled on the DIVide..
• Stalling further instruction issue, e.g. the SUBD

\[
\begin{align*}
\text{DIV} & \quad F_0, \quad F_2, \quad F_4 \\
\text{ADDD} & \quad F_{10}, \quad F_0, \quad F_8 \\
\text{SUBD} & \quad F_{12}, \quad F_8, \quad F_{14}
\end{align*}
\]
• 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 scoreboarding?

• A technique that allows instructions to execute out of order…
  ♦ So long as there are sufficient resources and
  ♦ No data dependencies

• The goal of scoreboarding
  ♦ Maintain an execution rate of one instruction per clock cycle
Multiple execution pipelines in DLX with scoreboard
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
Stripmining
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

```plaintext
for i = 0 to N-1
    a[i] = b[i] + c[i];

for j = 0 to N-1 by VL
    for i = j to min(N, j+VL) – 1
        a[i] = b[i] + c[i];

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

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

```cpp
for j = 0 to N-1 by VL
    for i = j to min(N, j+VL) – 1
        a[i] = b[i] + c[i];
```
Today’s lecture

• Thread divergence
• Scheduling
• Floating Point
What is floating point?

• A representation
  - $\pm2.5732\ldots \times 10^{22}$
  - NaN $\infty$
  - Single, double, extended precision

• A set of operations
  - $+=*;/\sqrt{\text{rem}}$
  - Comparison $<\leq=\neq\geq$
  - Conversions between different formats, binary to decimal
  - Exception handling

• IEEE Floating point standard P754
  - Universally accepted
  - W. Kahan received the Turing Award in 1989 for design of IEEE Floating Point Standard
  - Revision in 2008
IEEE Floating point standard P754

- Normalized representation \( \pm 1.d\ldots d \times 2^{\text{esp}} \)
  - Macheps = Machine epsilon = \( \varepsilon = 2^{-\#\text{significand bits}} \)
    relative error in each operation
  - OV = overflow threshold = largest number
  - UN = underflow threshold = smallest number
- \( \pm \text{Zero: } \pm \text{significand and exponent } = 0 \)

<table>
<thead>
<tr>
<th>Format</th>
<th># bits</th>
<th>#significand bits</th>
<th>macheps</th>
<th>#exponent bits</th>
<th>exponent range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>32</td>
<td>23+1</td>
<td>(2^{-24} (~10^{-7}))</td>
<td>8</td>
<td>(2^{-126} - 2^{127} (~10^{+38}))</td>
</tr>
<tr>
<td>Double</td>
<td>64</td>
<td>52+1</td>
<td>(2^{-53} (~10^{-16}))</td>
<td>11</td>
<td>(2^{-1022} - 2^{1023} (~10^{+308}))</td>
</tr>
<tr>
<td>Double</td>
<td>(\geq 80)</td>
<td>(\geq 64)</td>
<td>(\leq 2^{-64} (~10^{-19}))</td>
<td>(\geq 15)</td>
<td>(2^{-16382} - 2^{16383} (~10^{+4932}))</td>
</tr>
</tbody>
</table>

Jim Demmel
©2012 Scott B. Baden /CSE 260/ Fall 2012
What happens in a floating point operation?

- Round to the nearest representable floating point number that corresponds to the exact value (correct rounding)
- Round to nearest value with the lowest order bit = 0 (rounding toward nearest even)
- Others are possible
- We don’t need the exact value to work this out!
- Applies to \(+ = \ast / \sqrt{\text{rem}}\)
- Error formula: \(\text{fl}(a \op b) = (a \op b)(1 + \delta)\) where
  - \(\op\) one of \(+, -, \ast, /\)
  - \(|\delta| \leq \varepsilon\)
  - assuming no overflow, underflow, or divide by zero
- Addition example
  - \(\text{fl}(\sum x_i) = \sum_{i=1:n} x_i(1+e_i)\)
  - \(|e_i| \sim (n-1)\varepsilon\)
Exception Handling

• An exception occurs when the result of a floating point operation is not representable as a normalized floating point number
  - $1/0$, $\sqrt{-1}$

• P754 standardizes how we handle exceptions
  - **Overflow**: exact result $> \text{OV}$, too large to represent
  - **Underflow**: exact result nonzero and $< \text{UN}$, too small to represent
  - **Divide-by-zero**: nonzero/0
  - **Invalid**: $0/0$, $\sqrt{-1}$, log(0), etc.
  - **Inexact**: there was a rounding error (common)

• Two possible responses
  - Stop the program, given an error message
  - Tolerate the exception
An example

• Graph the function

\[ f(x) = \frac{\sin(x)}{x} \]

• \( f(0) = 1 \)

• But we get a singularity @ \( x=0 \): \( 1/x = \infty \)

• This is an “accident” in how we represent the function (W. Kahan)

• We *catch* the exception (divide by 0)

• Substitute the value \( f(0) = 1 \)
Denormalized numbers

- We compute if $(a \neq b)$ then $x = a/(a-b)$
- We should never divide by 0, even if $a-b$ is tiny
- Underflow exception occurs when exact result $a-b <$ underflow threshold UN
- We return a denormalized number for $a-b$
  - $\pm 0.d\ldots d \times 2^{\text{min\_exp}}$
  - sign bit, nonzero significand, minimum exponent value
  - Fills in the gap between 0 and UN
NaN (Not a Number)

- Invalid exception
  - Exact result is not a well-defined real number
- We can have a quiet NaN or an sNan
  - Quiet – does not raise an exception, but propagates a distinguished value
    - E.g. missing data: max(3, NAN) = 3
  - Signaling - generate an exception when accessed
    - Detect uninitialized data
Exception handling

- An important part of the standard, 5 exceptions
  - Overflow and Underflow
  - Divide-by-zero
  - Invalid
  - Inexact
- Each of the 5 exceptions manipulates 2 flags
  - Sticky flag set by an exception
    - Remains set until explicitly cleared by the user
  - Exception flag: should a trap occur?
    - If so, we can enter a trap handler
    - But requires precise interrupts, causes problems on a parallel computer
- We can use exception handling to build faster algorithms
  - Try the faster but “riskier” algorithm
  - Rapidly test for accuracy (possibly with the aid of exception handling)
  - Substitute slower more stable algorithm as needed
When compiler optimizations alter precision

• Let’s say we support 79+ bit extended format in registers
• When we store values into memory, values truncated to the lower precision format, e.g. 64 bits
• Compilers can keep things in registers and we may lose *referential transparency*
• An example
  
  ```
  float x, y, z;
  int j;
  ....
  x = y + z;
  if (x >= j) replace x by something smaller than j
  y=x;
  ```

• With optimization turned on, *x* is computed to extra precision; it is not a float
• If *x < j* and held in a register…. no guarantee the condition *x < j* will be preserved … when *x* is stored in *y*, i.e. *y >= j*
P754 on the GPU

- Cuda Programming Guide (4.1)
  “All compute devices follow the IEEE 754-2008 standard for binary floating-point arithmetic with the following deviations”
  - There is no mechanism for detecting that a floating-point exception has occurred and all operations behave as if the exceptions are always masked… SNaN … are handled as quiet

- Cap. 2.x: FFMA … is an IEEE-754-2008 compliant fused multiply-add instruction … the full-width product … used in the addition & a single rounding occurs during generation of the final result
  - $\text{rnd}(A \times A + B)$ with FFMA (2.x) vs $\text{rnd}(\text{rnd}(A \times A) + B)$ FMAD for 1.x

- FFMA can avoid loss of precision during subtractive cancellation when adding quantities of similar magnitude but opposite signs

- Also see *Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs,* ” by N. Whitehead and A. Fit-Florea