Lecture 11
CSE 260 – Parallel Computation
(Fall 2015)
Scott B. Baden

Thread Scheduling
Thread Divergence
Announcements

• Next Weds office hours will end at 3:30 instead of 4pm (11/4)
Recapping from last time: the warp scheduler

• Each SMX has 4 warp schedulers
• Schedular assigns independent instructions for processing … from same or different warp …. in same or different blocks
• Can dispatch 2 independent instructions/ cycle
• Scheduler choose the best warp to go next if there are multiple candidates
• How does the scheduler even know when an instruction is able to execute?
What makes a processor run faster?

- Registers and cache
- Vectorization, SSE
- Pipelining
- Instruction level parallelism
Pipelining

- **Laundry Example**
  
  wash (30 min) + dry (40 min) + fold (20 min) = 90 min

  - Sequential execution takes
    4 * 90 min = 6 hours
  - Pipelined execution takes
    30 + 4*40 + 20 = 3.5 hours
  - Bandwidth = loads/hour
  - Pipelining helps bandwidth but not latency (90 min)
  - Bandwidth limited by slowest pipeline stage
  - Potential speedup
    = Number pipe stages

Scott B. Baden / CSE 260, UCSD / Fall '15
Multi-execution pipeline

- **MIPS R4000**

```
\[ w = u / v; \]
\[ a = b + c; \]
\[ s = q * r; \]
\[ i++; \]
```

```
\[ x = y / z; \]
\[ t = c - q; \]
\[ a = x + c; \]
```

David Culler

Scott B. Baden / CSE 260, UCSD / Fall '15
Instruction level parallelism

- Execute more than one instruction at a time
  \[ w = u / z; \]
  \[ a = b + c; \]
- But what if we have dependencies?
  \[ x = y / z; \]
  \[ a = x + c; \]
  \[ t = c - q; \]
Pipeline Hazards

- A data dependence is one kind of pipeline hazard, a data hazard, that prevents the next instruction from executing when expected
- Structural hazards: physical resources aren’t available
- Control hazards: branches or other instructions that change the PC
- The simplest solution is to stall the pipeline, introducing bubble, but we can do better
Static scheduling limits performance

- The ADDD instruction is stalled on the DIVide …
- …stalling further instruction issue, e.g. the SUBD
  \[
  \text{DIV} \quad \text{ADD} \quad \text{SUBD}
  \]
  \[
  \begin{align*}
  \text{DIV} & : F0, F2, F4 \\
  \text{ADD} & : F10, F0, F8 \\
  \text{SUB} & : F12, F8, F14
  \end{align*}
  \]
- 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

• Let’s modify the pipeline to enable instructions to execute as soon as their operands become available: *out-of-order execution*

• Stalled instructions (SUBD) can now proceed normally

• Complications: dynamically scheduled instructions also complete out of order

\[
x = y / z; \quad x = y / z; \\
a = x + c; \quad t = c - q; \\
t = c - q; \quad a = x + c;
\]
Dynamic scheduling splits the ID stage

• Issue sub-stage
  ◆ Decode the instructions
  ◆ Check for structural hazards

• Read operands sub-stage
  ◆ Wait until there are no data hazards
  ◆ Read operands
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 - need additional registers to store pending instructions that aren’t ready to execute
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
Scoreboarding

• Hardware data structures keep track of
  - When instructions complete
  - Which instructions depend on the results
  - When it’s safe to write a reg.

• Deals with data hazards
  - WAR (Write after read)
  - RAW (Read after write)
  - WAW (Write after write)
Scoreboard controls the pipeline

Instruction Fetch → Instruction Issue

Pre-issue buffer

Instruction Decode

Read operands → Execution unit 1 → Write results

Read operands → Execution unit 2

Pre-execution buffers

Post-execution buffers

Scoreboard / Control Unit

WAW

RAW

WAR

Mike Frank

Scott B. Baden / CSE 260, UCSD / Fall '15
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
• Our processor (DLX) has: 2 multipliers, 1 divider, 1 integer unit (memory, branch, integer arithmetic)
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
Multiple execution pipelines in DLX with scoreboard

- A centralized bookkeeping table: tracks instructions + registers they depend or modify
- Status of result registers (who is going to write to a given register)
- Status of the functional units

### Registers

<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>

### Data buses

- FP mult
- FP divide
- FP add
- Integer unit

Scott B. Baden / CSE 260, UCSD / Fall '15
Scoreboarding on the GPU

- 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

TB = Thread Block, W = Warp
Today’s lecture

• Thread scheduling
• Thread Divergence
Thread divergence

• All the threads in a warp execute the same instruction

• Different control paths are serialized
Thread divergence

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

• Execution exhibits divergence when a \textit{predicate} is a function of the thread Id
  
  \begin{verbatim}
  if (threadId >= 2) {}
  \end{verbatim}

• No divergence if all follow the same path
  
  \begin{verbatim}
  if (threadId / WARP_SIZE >= 2) {}
  \end{verbatim}

• We can have different control paths within the thread block
Divergence example

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

Mary Hall
Divergence example

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

Mary Hall

Scott B. Baden / CSE 260, UCSD / Fall '15
if (threadIdx >= 2)
  a=100;
else
  a=-100;
Example – reduction – thread divergence ∑ᵢ xᵢ

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 2</th>
<th>Thread 4</th>
<th>Thread 6</th>
<th>Thread 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>2+3</td>
<td>4+5</td>
<td>6+7</td>
<td>8+9</td>
</tr>
<tr>
<td>2</td>
<td>0...3</td>
<td>4..7</td>
<td>8..11</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0..7</td>
<td>8..15</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Scott B. Baden / CSE 260, UCSD / Fall '15

David Kirk NVIDIA & Wen-mei Hwu UIUC
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

- All threads in a warp execute the same instruction
  reduceSum<<<N/512,512>>>(x,N)
- No divergence until stride < 32
- All warps active when stride ≥ 32

```c
__shared__ int x[ ];
    unsigned int tid = threadIdx.x;
    unsigned int s;

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

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

- s = 256: threads 0:255
- s = 128: threads 0:127
- s =  64: threads 0:63
- s =  32: threads 0:31
- s =  16: threads 0:15
  ...
CUDA implementation of predication

• All instructions support predication (started with 2.x)

• Condition code or *predicate* per thread:
  set to true or false

• Execute only if the predicate is true

if (x>1)
    y = 7;

  test = (x>1)
  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