CS472 - Computer Architecture Lecture 10

D. Kevin McGrath

May 2, 2012
Pipelining and C++ AMP

- Overview of pipelining
- Pipelined data path and control
- C++ AMP introduction
Outline

1. Pipelining overview
2. Data path
3. C++ AMP
Steps of instruction execution

1. IF: Fetch instructions from memory
2. ED: Read registers while decoding instructions
3. EX: Execute the operation or calculate an address
4. MEM: Access an operand in data memory
5. WB: Write the result to a register
Pipeline performance

• Assume time for stages is
  • 100 ps for register read or write
  • 200 ps for other stages

• Total time for each class would be (in ps)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>200</td>
<td>100</td>
<td>200</td>
<td>200</td>
<td>100</td>
<td>800</td>
</tr>
<tr>
<td>sw</td>
<td>200</td>
<td>100</td>
<td>200</td>
<td>200</td>
<td></td>
<td>700</td>
</tr>
<tr>
<td>R-format</td>
<td>200</td>
<td>100</td>
<td>200</td>
<td></td>
<td>100</td>
<td>600</td>
</tr>
<tr>
<td>beq</td>
<td>200</td>
<td>100</td>
<td>200</td>
<td></td>
<td></td>
<td>500</td>
</tr>
</tbody>
</table>

• This assumes muxes, control, PC access, and sign extension have 0 delay

• lw longest instruction, defines clock cycle in single cycle data path
Pipelining speedup

- Under ideal conditions, speedup is by a factor of the number of stages in the pipeline
- For MIPS, this would imply we would see a $5\times$ speedup
- Longest time to clear a stage defines clock cycle for pipelined data path
- $200\,\text{ps}$ would thus be the clock cycle
- Speedup is actually only a factor of 4, as stages are not balanced
- Instruction *throughput* is increased, rather than decreasing the time to execute an instruction
ISA design for pipelining

- All instructions of the same length
- Small number of instruction formats, with source registers in the same location for all formats
- Memory operands only exist in loads in stores
- Operands are all aligned, so transferring only takes a single pipeline stage
Pipeline hazards

- Situations where the next instruction cannot execute in the following clock cycle
- Known as *hazards*
- 3 types:
  - Structural hazards
  - Data hazards
  - Control hazards
Outline

1. Pipelining overview
2. Data path
3. C++ AMP
Outline

1. Pipelining overview

2. Data path

3. C++ AMP
High performance programming

- Historically, just wait a little while, and performance will double
- This trend ended not that long ago
- Instead of increasing single thread performance, multi-core was the way forward
  - OpenMP
  - pthreads
  - Vectorization
- With the increasing use of threads, heterogeneous computing environments have been proposed as a performance improvement
  - Makes use of what is known as an “accelerator”
  - GPGPU programming
  - Co-processors
Multi-core libraries

- OpenMP
- pthreads
- Concurrency Runtime (ConcRT)
  - Parallel Patterns Library (PPL)
  - Asynchronous Agents Library
- Task Parallel Library (TPL)
- WARP - Windows Advanced Rasterization Platform
- Intel’s Thread Building Blocks (TBB)
GPGPU and heterogeneous computing

- Languages and libraries for general purpose GPU programming
- Only certain classes of problems can be enhanced
- Early versions often only support single precision floating point
- Often domain specific language, requires translation prior to compilation
- Examples:
  - CUDA
  - OpenCL
  - Direct3D
C++ AMP

- C++ Accelerated Massive Parallelism
- Targets GPUs, as well as other accelerators
  - First revision limited to CPUs and GPUs
  - Work ongoing to allow for other accelerators, including custom ASICs and FPGAs
  - Any device which presents a standard interface can be used in the future
- Templated, type- and exception-safe OO code
- Open standard, developed by Microsoft with input from the C++ community
- Almost entirely library based, rather than a new language
• Standard C/C++:

```c++
void AddArrays(int n, int *pA, int *pB, int *pC){
    for (int i=0; i<n; i++){
        pC[i] = pA[i] + pB[i];
    }
}
```
C++ AMP:

```cpp
#include <amp.h>
using namespace concurrency;

void AddArrays(int n, int *pA, int *pB, int *pC){
    array_view<int, 1> a(n, pA);
    array_view<int, 1> b(n, pB);
    array_view<int, 1> sum(n, pC);

    parallel_for_each(sum.grid,
        [=](index<1> idx) restrict(amp) {
            sum[idx] = a[idx] + b[idx];
        });
}
```
Details

- In the event of wanting to run code on platforms that do not have the necessary hardware accelerator, graceful fallback to CPU occurs
- C++ AMP is *not* tied to any specific platform
- C++ AMP does *not* require a special compiler
- C++ AMP is more general than OpenMP, but has similar programmer effort requirements
- It is important to determine if the effort, small as it is, is worth the tradeoff
Details

- For small problems, CPU is often faster
- For large problems, accelerators are often faster
- Matrix multiplication:
  - $1024 \times 1024$ matrices
    - CPU time: $\approx 21$ seconds
    - AMP time: $\approx 9.7$ seconds
    - Tiled AMP time: $\approx 5$ seconds
  - $512 \times 512$ matrices
    - CPU time: $\approx 10$ seconds
    - AMP time: $\approx 6$ seconds
    - Tiled AMP time: $\approx 7.5$ seconds