Review

- SIMD Parallelism via Intel SSE Instructions
- Use of SSE intrinsics to get access to assembly instructions from C code
- Restructuring data to provide aligned access for SSE loads and stores

New-School Machine Structures (It’s a bit more complicated!)

- Parallel Requests
  - Assigned to computer
  - e.g., Search “Katz
- Parallel Threads
  - Assigned to core
  - e.g., Lookup, Ads
- Parallel Instructions
  - >1 instruction @ one time
  - e.g., 5 pipelined instructions
- Parallel Data
  - >1 data item @ one time
  - e.g., Add of 4 pairs of words
- Hardware descriptions
  - All gates @ one time
- Programming Languages

Simple Multiprocessor

Multiprocessor Execution Model

- Each processor has its own PC and executes an independent stream of instructions (MIMD)
- Different processors can access the same memory space
  - Processors can communicate via shared memory by storing/loading to/from common locations
- Two ways to use a multiprocessor:
  1. Deliver high throughput for independent jobs via job-level parallelism
  2. Improve the run time of a single program that has been specially crafted to run on a multiprocessor - a parallel-processing program

Now use term core for processor ("Multicore") because "Multiprocessor Microprocessor" too redundant

Transition to Multicore
Parallelism Only Path to Higher Performance

- Sequential processor performance not expected to increase much, and might go down
- If want apps with more capability, have to embrace parallel processing (SIMD and MIMD)
- In mobile systems, use multiple cores and GPUs
- In warehouse-scale computers, use multiple nodes, and all the MIMD/SIMD capability of each node

Potential Parallel Performance (assuming SW can use it)

<table>
<thead>
<tr>
<th>Year</th>
<th>Cores</th>
<th>SIMD bits /Core</th>
<th>Core * SIMD bits</th>
<th>Peak GFLOPs/Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>2003</td>
<td>MIMD 2</td>
<td>SIMD 128</td>
<td>256 MIMD 4</td>
<td>4</td>
</tr>
<tr>
<td>2005</td>
<td>+2/ 4 yrs</td>
<td>128</td>
<td>512 SIMD 8</td>
<td>8</td>
</tr>
<tr>
<td>2007</td>
<td>6</td>
<td>128</td>
<td>768</td>
<td>12</td>
</tr>
<tr>
<td>2009</td>
<td>8</td>
<td>128</td>
<td>1024</td>
<td>16</td>
</tr>
<tr>
<td>2011</td>
<td>10</td>
<td>256</td>
<td>2560</td>
<td>40</td>
</tr>
<tr>
<td>2013</td>
<td>12</td>
<td>256</td>
<td>3072</td>
<td>48</td>
</tr>
<tr>
<td>2015</td>
<td>14/ 8 yrs</td>
<td>512</td>
<td>7168 MIMD 20X</td>
<td>112</td>
</tr>
<tr>
<td>2017</td>
<td>16</td>
<td>512</td>
<td>8192</td>
<td>128</td>
</tr>
<tr>
<td>2019</td>
<td>18</td>
<td>1024</td>
<td>18432</td>
<td>288</td>
</tr>
<tr>
<td>2021</td>
<td>20</td>
<td>1024</td>
<td>20480</td>
<td>320</td>
</tr>
</tbody>
</table>

Multiprocessors and You

- Only path to performance is parallelism
  - Clock rates flat or declining
  - SIMD: 2X width every 3-4 years
    - 128b wide now, 256b in 2017, 512b in 2017, 1024b in 2018?
  - MIMD: Add 2 cores every 2 years: 2, 4, 6, 8, 10, ...
- A key challenge is to craft parallel programs that have high performance on multiprocessors as the number of processors increase – i.e., that scale
  - Scheduling, load balancing, time for synchronization, overhead for communication
- Project 3: fastest code on 8-core computers
  - 2 chips/computer, 4 cores/chip

Shared Memory and Caches

- What if?
  - Processors 1 and 2 read Memory[1000] (value 20)

Shared Memory and Caches

- Now:
  - Processor 0 writes Memory[1000] with 40

Problem?
Keeping Multiple Caches Coherent

- Architect’s job: shared memory => keep cache values coherent
- Idea: When any processor has cache miss or writes, notify other processors via interconnection network
  - If only reading, many processors can have copies
  - If a processor writes, invalidate any other copies
- Write transactions from one processor “snoop” tags of other caches using common interconnect
  - Invalidate any “hits” to same address in other caches
  - If hit is to dirty line, other cache has to write back first!

Shared Memory and Caches

- Example, now with cache coherence
  - Processors 1 and 2 read Memory[1000]
  - Processor 0 writes Memory[1000] with 40

Flashcard Quiz: Which statement is true?

- Using write-through caches removes the need for cache coherence
- Every processor store instruction must check contents of other caches
- Most processor load and store accesses only need to check in local private cache
- Only one processor can cache any memory location at one time

Administrivia

- Midterm Tuesday Oct 9, 8PM:
  - Two rooms: 1 Pimentel and 2050 LSB
  - Check your room assignment!
  - Covers everything through lecture Wednesday 10/3
  - Closed book, can bring one sheet notes, both sides
  - Copy of Green card will be supplied
  - No phones, calculators, …; just bring pencils & eraser
- NO LECTURE ON WEDNESDAY OCTOBER 10!!!

Midterm Room Assignment by Login

1 Pimentel = logins ab – mk
2050 VLSB = logins mm - xm

Cache Coherency Tracked by Block

- Suppose block size is 32 bytes
- Suppose Processor 0 reading and writing variable X, Processor 1 reading and writing variable Y
- Suppose in X location 4000, Y in 4012
- What will happen?
Coherency tracked by cache line

- Block ping-pongs between two caches even though processors are accessing disjoint variables
- Effect called false sharing
- How can you prevent it?

Fourth “C” of Cache Misses: Coherence Misses

- Misses caused by coherence traffic with other processor
- Also known as communication misses because represents data moving between processors working together on a parallel program
- For some parallel programs, coherence misses can dominate total misses

Threads

- A thread is a unit of work described by a sequential flow of instructions
- Each thread has a program counter and processor registers, and accesses the shared memory
- Each processor provides one (or more) hardware threads that actively execute instructions
- An operating system can multiplex multiple software threads onto the available hardware threads

Operating System Threads

Give the illusion of many active threads by time-multiplexing hardware threads among software threads

- Remove a software thread from a hardware thread by interrupting its execution and saving its registers and PC into memory
  - Also if one thread is blocked waiting for network access or user input
- Can make a different software thread active by loading its registers into processor and jumping to its saved PC

Hardware Multithreading

- Basic idea: Processor resources are expensive and should not be left idle
- Long memory latency to memory on cache miss?
- Hardware switches threads to bring in other useful work while waiting for cache miss
- Cost of thread context switch must be much less than cache miss latency
- Put in redundant hardware so don’t have to save context on every thread switch:
  - PC, Registers
- Attractive for apps with abundant TLP
  - Commercial multi-user workloads

Hardware Multithreading

- Two copies of PC and Registers inside processor hardware
- Looks like two processors to software (hardware thread 0, hardware thread 1)
- Control logic decides which thread to execute an instruction from next
Multithreading vs. Multicore

- Multithreading => Better Utilization
  - ≈1% more hardware, 1.10X better performance?
  - Share integer adders, floating-point adders, caches (L1 I S, L1 D$, L2 cache, L3 cache), Memory Controller
- Multicore => Duplicate Processors
  - ≈50% more hardware, ≈2X better performance?
  - Share outer caches (L2 cache, L3 cache), Memory Controller

10/7/12

Fall 2012 - Lecture #19

Machines in (old) 61C Lab

- `/usr/sbin/sysctl -a | grep hw`.
  - `hw.model` = MacPro4,1
  - `hw.physicalcpu` = 8
  - `hw.logicalcpu` = 16
  - `hw.physmem` = 2,147,483,648

Therefore, should try up to 16 threads to see if performance gain even though only 8 cores

10/7/12

Fall 2012 - Lecture #19

And in Conclusion, ...

- Sequential software is slow software
  - SIMD and MIMD only path to higher performance
- Multiprocessor/Multicore uses Shared Memory
  - Cache coherency implements shared memory even with multiple copies in multiple caches
  - False sharing a concern; watch block size!
- Multithreading increases utilization, Multicore more processors (MIMD)