CS 61C: Great Ideas in Computer Architecture (Machine Structures)

Thread Level Parallelism

Instructor:
Michael Greenbaum
Review

• Flynn Taxonomy of Parallel Architectures
  – *SIMD*: *Single Instruction Multiple Data*
  – *MIMD*: *Multiple Instruction Multiple Data*
  – SISD: Single Instruction Single Data (unused)
  – MISD: Multiple Instruction Single Data

• Intel SSE SIMD Instructions
  – One instruction fetch that operates on multiple operands simultaneously
  – 128/64 bit XMM registers
  – SSE Instructions in C
    • Compiler intrinsics map directly to SIMD instructions.
Amdahl’s Law and Parallelism

\[
\text{Speedup} = \frac{1}{(1 - F) + \frac{F}{S}}
\]

- Non-speed-up part
- Speed-up part

Table:

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Parallel portion</th>
<th>Serial portion</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Graph:
- Time (vertical axis)
- Parallel portion
- Serial portion
- Number of Processors (horizontal axis)

7/18/2011
Parallel Speed-up Example

\[ Z_1 + Z_2 + \ldots + Z_{10} \]

\[
\begin{bmatrix}
X_{1,1} & X_{1,10} \\
\vdots & + \\
X_{10,1} & X_{10,10}
\end{bmatrix}
\]

\[
\begin{bmatrix}
Y_{1,1} & Y_{1,10} \\
\vdots & \\
Y_{10,1} & Y_{10,10}
\end{bmatrix}
\]

Partition 10 ways and perform on 10 parallel processing units.

Non-parallel part

Parallel part

- 10 “scalar” operations (non-parallelizable)
- 100 parallelizable operations
  - Say, element-wise addition of two 10x10 matrices.
- 110 operations
  - \( \frac{100}{110} = 0.909 \) Parallelizable, \( \frac{10}{110} = 0.091 \) Scalar
Parallel Speed-up Example

Speedup w/ \( E \) = \( \frac{1}{(1-F) + F/S} \)

• Consider summing 10 scalar variables and two 10 by 10 matrices (matrix sum) on 10 processors
  
  Speedup w/ \( E \) = \( \frac{1}{.091 + .909/10} \) = \( 1/0.1819 = 5.5 \)

• What if there are 100 processors?
  
  Speedup w/ \( E \) = \( \frac{1}{.091 + .909/100} \) = \( 1/0.10009 = 10.0 \)

• Now, what if the matrices are 32 by 32 (1024 ops) on 10 processors? (10x the parallel data)
  
  Speedup w/ \( E \) = \( \frac{1}{.009 + .991/10} \) = \( 1/0.108 = 9.2 \)

• What if there are 100 processors?
  
  Speedup w/ \( E \) = \( \frac{1}{.009 + .991/100} \) = \( 1/0.019 = 52.6 \)
Strong and Weak Scaling

• To get good speedup on a multiprocessor while keeping the problem size fixed is harder than getting good speedup by increasing the size of the problem.
  
  – **Strong scaling**: when speedup can be achieved on a parallel processor without increasing the size of the problem
  – (e.g., 10x10 Matrix on 10 processors to 10x10 on 100)
  
  – **Weak scaling**: when speedup is achieved on a parallel processor by increasing the size of the problem proportionally to the increase in the number of processors
  – (e.g., 10x10 Matrix on 10 processors =>32x32 Matrix on 100)
Agenda

• Loop Unrolling For SIMD
• Administrivia
• Multiprocessor Systems
• Break
• Multiprocessor Cache Consistency
• Synchronization - A Crash Course
• Summary
Data Level Parallelism and SIMD

• SIMD wants adjacent values in memory that can be operated in parallel
• Usually specified in programs as loops
  
  ```c
  for(i=0; i>1000; i=i-1)
    x[i] = x[i] + s;
  ```

• How can reveal more data level parallelism than available in a single iteration of a loop?

• *Unroll loop* and adjust iteration rate
Looping in MIPS

Assumptions:
- $s0$ is initially the address of the element in the array with the highest address
- $s1$ contains the scalar value $s$
- $s2$ termination address (bottom of array)

CODE:
Loop:
1. lw $t0, 0($s0)
2. addu $t0,$t0,$s1 # add $s$ to array element
3. sw $t0,0($s0) # store result
4. addui $s0,$s0,-4 # decrement pointer 4 bytes
5. bne $s0,$s2,Loop # repeat Loop if not finished
Loop Unrolled

Loop:

lw   $t0,0($s0)
addu  $t0,F0,$s1
sw   $t0,0($s0)
lw   $t1,-4($s0)
addy  $t1,$t1,$s1
sw   $t1,-4($s0)
lw   $t2,-8($s0)
addu  $t2,$t2,$s1
sw   $t2,-8($s0)
lw   $t3,-12($s0)
addu  $t3,$t3,$s1
sw   $t3,-12($s0)
addui  $s0,$s0,-16
bne  $s0,$s2,Loop

NOTE:
1. Different Registers eliminate stalls
2. Only 1 Loop Overhead every 4 iterations
3. This unrolling works if
   loop_limit(mod 4) = 0
Loop Unrolled Scheduled

Loop:

```
.l.d $t0,0($s0)
.l.d $t1,-4($s0)
.l.d $t2,-8($s0)
.l.d $t3,-12($s0)
add.d $t0,$t0,$s1
add.d $t1,$t1,$s1
add.d $t2,$t2,$s1
add.d $t3,$t3,$s1
.s.d $t0,0($s0)
.s.d $t1,-8($s0)
.s.d $t2,-16($s0)
.s.d $t3,-24($s0)
addui $s0,$s0,#-32
bne $s0,R2,Loop
```

4 Loads side-by-side: Could replace with 4 wide SIMD Load

4 Adds side-by-side: Could replace with 4 wide SIMD Add

4 Stores side-by-side: Could replace with 4 wide SIMD Store
Loop Unrolling in C

• Instead of compiler doing loop unrolling, could do it yourself in C
  
  ```c
  for(i=1000; i>0; i=i-1)
      x[i] = x[i] + s;
  ```

• Could be rewritten

  ```c
  for(i=1000; i>0; i=i-4) {
      x[i] = x[i] + s;
      x[i-1] = x[i-1] + s;
      x[i-2] = x[i-2] + s;
      x[i-3] = x[i-3] + s;
  }
  ```

What is downside of doing it in C?
Generalizing Loop Unrolling

• A loop of \textit{n iterations} \\
• \textbf{k-fold} unrolling of the body of the loop \\
Then we will run the loop with 1 copy of the body \textit{n(mod k)} times and \\
with k copies of the body \textit{floor(n/k)} times \\
• (Will revisit loop unrolling again when get to pipelining later in semester)
Agenda

- Loop Unrolling For SIMD
- Administrivia
- Multiprocessor Systems
- Break
- Multiprocessor Cache Consistency
- Synchronization - A Crash Course
- Summary
Midterm Results

Mean: 63.6
Standard Dev: 16.6
Max: 94
Administrivia

• Regrade Policy
  – Rubric will be posted shortly
  – Any questions? Ask in discussion section.
  – **Written** appeal process
    • Explain rationale for regrade request
    • Attach rationale to exam
    • Submit to your TA by next Tuesday’s lab.

• Remember, you can overwrite your score on the midterm if you do better on the matching portion of the final.
Administrivia

• Project #2: Matrix Multiply Performance Improvement
  – Work in groups of two!
  – Part 1: Due July 24 (this Sunday)
  – Part 2: Due July 31
• HW #3 also due July 27
• Closely packed due dates, try to get ahead of schedule for the project.
cs61c in... Game Development

- Knowledge of Floating Point needed when building a game! (3d block building game, Minecraft)
- “For example, at extreme distances, the player may move slower than near the center of the world, due to rounding errors (the position has a huge mantissa, the movement delta has a tiny, so it gets cut off faster).”
- “Many of these problems can be solved by changing the math into a local model centered around the player so the numbers all have vaguely the same magnitude.”

You Are Here!

**Software**

- **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 functioning in parallel at same time

**Hardware**

- Warehouse Scale Computer
- Memory (Cache)
- Input/Output
- Logic Gates
- Functional Unit(s)
  \( A_0 + B_0 \)
  \( A_1 + B_1 \)
  \( A_2 + B_2 \)
  \( A_3 + B_3 \)
- Core
- ... Core
- Main Memory
- Project 3
- Smart Phone

Today’s Lecture
Parallel Processing:
Multiprocessor Systems (MIMD)

- Multiprocessor (MIMD): a computer system with at least 2 processors

1. Deliver high throughput for independent jobs via request-level or task-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” is redundant
Transition to Multicore

- AMD Phenom (4 cores)
- Intel Pentium 4
- Intel Pentium Pro
- MIPS R2000

Graph showing the following:
- Transistors (Thousands)
- Parallel App Performance
- Sequential App Performance
- Frequency (MHz)
- Typical Power (Watts)
- Number of Cores

Data partially collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond
Three Key Questions about Multiprocessors

• Q1 – How do they share data?

• Q2 – How do they coordinate?

• Q3 – How many processors can be supported?
Three Key Questions about Multiprocessors: Q1

• Q1 – How do they share data?
• Single address space shared by all processors/cores

• eg, Core 1 writes data to a given address, Core 2 reads data from that address; it will get the same data.
Three Key Questions about Multiprocessors: Q2

• Q2 – How do they coordinate?
• Processors coordinate/communicate through shared variables in memory (via loads and stores)
  – Use of shared data must be coordinated via synchronization primitives (locks) that allow access to data to only one processor at a time
• All multicore computers today are Shared Memory Multiprocessors (SMPs)
Example: Sum Reduction

- Sum 100,000 numbers on 100 processor SMP
  - Each processor has ID: $0 \leq P_n \leq 99$
  - Partition 1000 numbers per processor
  - Initial summation on each processor:
    $$\text{sum}[P_n] = 0;$$
    $$\text{for } (i = 1000*P_n; \ i < 1000*(P_n+1); \ i = i + 1)$$
    $$\text{sum}[P_n] = \text{sum}[P_n] + A[i];$$

- Now need to add these partial sums
  - Reduction: divide and conquer
    - Half the processors add pairs, then quarter, ...
    - Need to synchronize between reduction steps

- Sometimes just adding them serially is fastest
Example: Sum Reduction

This code executes simultaneously in P0, P1, ..., P7

```
half = 8;
repeat
    synch();
    if (half%2 != 0 && Pn == 0)
        sum[0] = sum[0] + sum[half-1];
    /* Special case when half is odd; Processor0 gets extra element */
    half = half/2; /* dividing line on who sums */
    if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1);
```
An Example with 10 Processors


half = 10
half = 5
half = 2
half = 1

7/18/2011
Spring 2011 -- Lecture #15
Three Key Questions about Multiprocessors: Q3

• Q3 – How many processors can be supported?
• Key bottleneck in an SMP is the memory system
• Caches can effectively increase memory bandwidth/open the bottleneck
• But what happens to the memory being actively shared among the processors through the caches?
Shared Memory and Caches

• What if?
  – Processors 1 and 2 read Memory[1000] (value 20)
Shared Memory and Caches

• What if?
  – Processors 1 and 2 read Memory[1000]
  – Processor 0 writes Memory[1000] with 40
Agenda

• Loop Unrolling For SIMD
• Administrivia
• Multiprocessor Systems
• Break
• Multiprocessor Cache Consistency
• Synchronization - A Crash Course
• Summary
Keeping Multiple Caches Coherent

- HW 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 all other copies
- Shared written result can “ping-pong” between caches
How Does HW Keep $ Coherent?

• Each cache tracks state of each block in cache:
  1. **Shared**: up-to-date data, other caches may have a copy
  2. **Modified**: up-to-date data, changed (dirty), no other cache has a copy, OK to write, memory out-of-date
Two Optional Performance Optimizations of Cache Coherency via New States

• Each cache tracks state of each block in cache:

3. **Exclusive**: up-to-date data, no other cache has a copy, OK to write, memory up-to-date
   – Avoids writing to memory if block replaced
   – Supplies data on read instead of going to memory

4. **Owner**: up-to-date data, other caches may have a copy (they must be in Shared state)
   – Only cache that supplies data on read instead of going to memory
Name of Common Cache Coherency Protocol: MOESI

- Memory access to cache is either **Modified** (in cache)
- **Owner**ed (in cache)
- **Exclusive** (in cache)
- **Shared** (in cache)
- **Invalid** (not in cache)

Snooping/Snoopy Protocols
  e.g., the Berkeley Ownership Protocol
Cache Coherency and Block Size

• 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?
• Effect called *false sharing*
• *How can you prevent it?*
Agenda

• Loop Unrolling For SIMD
• Administrivia
• Multiprocessor Systems
• Break
• Multiprocessor Cache Consistency
• **Synchronization - A Crash Course**
• Summary
Threads

• *Thread of execution*: smallest unit of processing scheduled by operating system

• On single/uni-processor, multithreading occurs by *time-division multiplexing*:
  – Processor switched between different threads
  – *Context switching* happens frequently enough user perceives threads as running at the same time

• On a multiprocessor, threads run at the same time, with each processor running a thread
Data Races and Synchronization

• Two memory accesses form a data race if from different threads to same location, and at least one is a write, and they occur one after another
• If there is a data race, result of program can vary depending on chance (which thread ran first?)
• Avoid data races by synchronizing writing and reading to get deterministic behavior
• Synchronization done by user-level routines that rely on hardware synchronization instructions
Example: Buying Milk for the Apartment

• Your fridge has no milk. You and your roommate will return from classes at some point and check the fridge.

• Whoever gets home first will check the fridge, go and buy milk, and return.

• What if the other person gets back while the first person is buying milk?
  – You’ve just bought twice as much milk as you need!

• It would’ve helped to have left a note...
Lock and Unlock Synchronization

• Lock used to create region *(critical section)* where only one thread can operate

• Given shared memory, use memory location as synchronization point: the *lock*

• Processors read lock to see if must wait, or OK to go into critical section (and set to locked)
  – 0 => lock is free / open / unlocked / lock off
  – 1 => lock is set / closed / locked / lock on

Set the lock
Critical section (only one thread gets to execute this section of code at a time)
e.g., change shared variables

Unset the lock
Possible Lock/Unlock Implementation

• **Lock (aka busy wait):**

  ```
  addiu $t1,$zero,1 ; t1 = Locked value
  Loop:  lw $t0,lock($s0) ; load lock
         bne $t0,$zero,Loop ; loop if locked
  Lock:  sw $t1,lock($s0) ; Unlocked, so lock
  ```

• **Unlock:**

  ```
  sw $zero,lock($s0)
  ```

• **Any problems with this?**
Possible Lock Problem

- Thread 1
  ```
  addiu $t1,$zero,1
  Loop: lw $t0,lock($s0)
  
  bne $t0,$zero,Loop
  
  Lock: sw $t1,lock($s0)
  ```

- Thread 2
  ```
  addiu $t1,$zero,1
  Loop: lw $t0,lock($s0)
  
  bne $t0,$zero,Loop
  
  Lock: sw $t1,lock($s0)
  ```

Both threads think they have set the lock
Exclusive access not guaranteed!
Help! Hardware Synchronization

• Hardware support required to prevent interloper (either thread on other core or thread on same core) from changing the value
  – *Atomic* read/write memory operation
  – No other access to the location allowed between the read and write

• Could be a single instruction
  – E.g., atomic swap of register ↔ memory
  – Or an atomic pair of instructions
Synchronization in MIPS

- Load linked: \texttt{ll rt,offset(rs)}
- Store conditional: \texttt{sc rt,offset(rs)}
  - Succeeds if location not changed since the \texttt{ll}
    - Returns 1 in rt (clobbers register value being stored)
  - Fails if location has changed
    - Returns 0 in rt (clobbers register value being stored)

- Example: atomic swap (to test/set lock variable)

Exchange contents of reg and mem: $s4 \leftrightarrow \text{Mem($s1$)}$

\begin{verbatim}
try: add $t0,$zero,$s4 ;copy exchange value
ll $t1,0($s1) ;load linked
sc $t0,0($s1) ;store conditional
beq $t0,$zero,try ;branch store fails
add $s4,$zero,$t1 ;put load value in $s4
\end{verbatim}
Test-and-Set

• In a single atomic operation:
  – *Test* to see if a memory location is set (contains a 1)
  – *Set* it (to 1) If it isn’t (it contained a zero when tested)
  – Otherwise indicate that the Set failed, so the program can try again
  – No other instruction can modify the memory location, including another Test-and-Set instruction

• Useful for implementing lock operations
Test-and-Set in MIPS

- Example: MIPS sequence for implementing a T&S at ($s1)

  Try:
  - addiu $t0, $zero, 1
  - `ll` $t1, 0($s1)
  - bne $t1, $zero, Try
  - `sc` $t0, 0($s1)
  - beq $t0, $zero, try

  Locked:
  - critical section

  sw $zero, 0($s1)
And In Conclusion, ...

• SIMD and MIMD for higher performance
• Multiprocessor (Multicore) uses Shared Memory (single address space)
• Cache coherency implements shared memory even with multiple copies in multiple caches
  – False sharing a concern
• Synchronization via hardware primitives:
  – MIPS does it with Load Linked + Store Conditional