Review

- Flynn Taxonomy of Parallel Architectures
  - SIMD: Single Instruction Multiple Data
  - MIMD: Multiple Instruction Multiple Data
  - MISD: Single Instruction Single Data (unused)
  - MISO: 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
  - Embed the SSE machine instructions directly into C programs through use of intrinsics
  - Achieve efficiency beyond that of optimizing compiler

Agenda

- Amdahl’s Law
- SIMD and Loop Unrolling
- Administrivia
- Memory Performance for Caches
- Review of 1st Half of 61C

Big Idea: Amdahl’s (Heartbreaking) Law

- Speedup due to enhancement E is:
  \[ \text{Speedup} = \frac{\text{Exec time w/o E}}{\text{Exec time w/ E}} \]
  \[ \text{Executive Time w/ E} = \text{Executive Time w/o E} \times \frac{1}{(1-F) + F/S} \]

Example #1: Amdahl’s Law

- Consider an enhancement which runs 20 times faster but which is only usable 25% of the time
  \[ \text{Speedup} = \frac{1}{1/(.75 + .25/20)} = 1.31 \]

- What if its usable only 15% of the time?
  \[ \text{Speedup} = \frac{1}{1/(.85 + .15/20)} = 1.17 \]

- Amdahl’s Law tells us that to achieve linear speedup with 100 processors, none of the original computation can be scalar!
  - To get a speedup of 90 from 100 processors, the percentage of the original program that could be scalar would have to be 0.1% or less
  \[ \text{Speedup} = \frac{1}{1/0.001 + .999/100} = 90.99 \]
**Parallel Speed-up Example**

\[
\begin{array}{c|c|c}
\text{Parallel part} & \text{Non-parallel part} & \text{Total} \\
\hline
z_0 + z_1 + \ldots + z_{19} & X_{10} & Y_{10} \\
& X_{11}, Y_{11} & X_{12}, Y_{12} \\
\end{array}
\]

| Partition 10 ways | Perform on 10 parallel processing units |
---|---|
To get good speedup on a multprocessor 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 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 => 33x33 Matrix on 100) - **Load balancing** is another important factor: every processor doing same amount of work - Just 1 unit with twice the load of others cuts speedup almost in half

- 10 “scalar” operations (non-parallelizable)
- 100 parallelizable operations
- 110 operations
- 100/110 = .909 Parallelizable, 10/110 = .091 Scalar

**Example #2: Amdahl’s Law**

Speedup w/E = 1 / (1 - F + F/S)

- Consider summing 10 scalar variables and two 10 by 10 matrices (matrix sum) on 10 processors
  Speedup w/E = 1/(.091 + .909/10) = 1/0.1819 = 5.5
- What if there are 100 processors?
  Speedup w/E = 1/(.009 + .991/100) = 1/0.1009 = 10.0
- Get 55% potential from 10, but only 10% potential of 100!
- What if the matrices are 33 by 33 (or 1019 adds in total) on 10 processors? (increase parallel data by 10x)
  Speedup w/E = 1/(.009 + .991/10) = 1/0.108 = 9.2
- What if there are 100 processors?
  Speedup w/E = 1/(.009 + .991/100) = 1/0.019 = 52.6
- Get 92% potential from 10 and 53% potential of 100

**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 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 => 33x33 Matrix on 100)
- **Load balancing** is another important factor: every processor doing same amount of work
  - Just 1 unit with twice the load of others cuts speedup almost in half

**Data Level Parallelism and SIMD**

- SIMD wants adjacent values in memory that can be operated in parallel
- Usually specified in programs as loops
  \[
  \text{for}(i=1000; i>0; i=1) \\
  x[i] = x[i] + s;
  \]
- 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:
- $s1$ is initially the address of the element in the array with the highest address
- $s0$ contains the scalar value $s$
- R($s2$) is the address of the last element to operate on

CODE:

1. ld $s2, $s1
2. add d $s10, $s12
3. add $s10, $s11
4. addui $s11, $s1, -8
5. bne $s11, $s12, Loop

```
Loop: 1. ld $s2, $s1 ; $s2-array element
  2. add d $s10, $s12 ; add s to $s2
  3. add $s10, $s11 ; store result
  4. addui $s11, $s1, -8
  5. bne $s11, $s12, Loop
```
Loop Unrolled

Loop:

```assembly
add d $10, $12, $0
s d $10, $9, $0
l d $8, $5, $1
add d $10, $9, $0
s d $12, $2, $0
l d $8, $5, $1
add d $10, $9, $0
s d $14, $3, $0
l d $8, $5, $1
add d $10, $9, $0
s d $16, $4, $0
l d $8, $5, $1
add d $10, $9, $0
s d $18, $5, $0
l d $8, $5, $1
```

Notes:
1. Only 1 loop overhead every 4 iterations
2. This unrolling works if loop_limit(mod 4) = 0
3. (Different registers eliminate stalls in pipeline; we’ll see later in course)

Loop Unrolled Scheduled

```assembly
Loop:

```
add d $10, $12, $0
s d $10, $9, $0
l d $8, $5, $1
add d $10, $9, $0
s d $12, $2, $0
l d $8, $5, $1
add d $10, $9, $0
s d $14, $3, $0
l d $8, $5, $1
add d $10, $9, $0
s d $16, $4, $0
l d $8, $5, $1
add d $10, $9, $0
s d $18, $5, $0
l d $8, $5, $1
```

• Loads side-by-side: Could replace with 4 wide SIMD Load
• Adds side-by-side: Could replace with 4 wide SIMD Add
• Stores side-by-side: Could replace with 4 wide SIMD Store

Generalizing Loop Unrolling

A loop of \( n \) iterations

• \( k \) copies of the body of the loop

Assume \((n \mod k) \neq 0\)

Then we will run the loop with 1 copy of the body \((n \mod k)\) times and

with \( k \) copies of the body \( \text{floor}(n/k) \) times

• (Will revisit loop unrolling again when get to pipelining later in semester)

Loop Unrolling in C

• Instead of compiler doing loop unrolling, could do it yourself in C
• Could be rewritten
• What is downside of doing it in C?

```c
for(i=1000; i>0; i--)
   x[i] = x[i] + 5;
```

Administrivia

• Lab #7 posted
• Midterm in 5 days:
  – Exam: Tu, Mar 6, 6:40-9:40 PM, 2050 VLSB
  – Covers everything through lecture today
  – 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 DAY OF EXAM, NO DISCUSSION SECTIONS
  – HKN Review: Sat, March 3, 3-5PM, 30E Soda Hall
  – TA Review: Sun, Mar 4, Starting 2PM, 2050 VLSB
• Will send (anonymous) 61C midterm survey before Midterm

Reading Miss Penalty: Memory Systems that Support Caches

• The off-chip interconnect and memory architecture affects overall system performance in dramatic ways

One word wide organization (one word wide bus and one word wide memory)

Assume
• 1 memory bus clock cycle to send address
• 15 memory bus clock cycles to get the 1st word in the block from DRAM (row cycle time), 5 memory bus clock cycles for 2nd, 3rd, 4th words (subsequent column access time)—note effect of latency!
• 1 memory bus clock cycle to return a word of data

Memory-Bus to Cache bandwidth
• Number of bytes accessed from memory and transferred to cache/CPU per memory bus clock cycle
Great Idea #1: Levels of Representation/Interpretation

- High Level Language (e.g., C)
  - Program (e.g., `int x = 5`)
- Compiler
- Assembly Language
  - Program (e.g., `mov $10, x`)
- Assembler
- Machine Language
  - Program (e.g., `1000 0010 0000 0000 0000`)

**temp = v[k];**
\[ v[k+1] = \text{temp}; \]
\[ \text{anything can be represented as a number, i.e., data or instructions} \]
\[ \text{v[10]: x, v[15]: y, v[2]: z} \]

1. **Pointer**
- Pointer is a C version (abstraction) of a data address
  - *"follows"* a pointer to its value
  - `&` gets the address of a value
  - Arrays and strings are implemented as variations on pointers
- Pointers are used to point to any kind of data (`int, char, a struct, etc.`)
- Normally a pointer only points to one type (`int, char, a struct, etc.`)
  - `void *` is a type that can point to anything (generic pointer)
If $t1$ and $t3$ represents the int pointers $p$ and $q$, and $t2$ represents int $x$, which statements about C compiled to MIPS instructions are true?

- $x = *p; \Rightarrow lw \ t2,0(t1)$
- $p = &q; \Rightarrow addiu t2,t3,0$
- $*p = x; \Rightarrow sw t2,0(t1)$
- All of the above are true

If $t1$ and $t3$ represents the int pointers $p$ and $q$, and $t2$ represents int $x$, which statements about C compiled to MIPS instructions are true?

- $q = p; \Rightarrow mov t3,t1$
- $q = p+1; \Rightarrow addiu t3,t1,4$
- $x = *(p+1); \Rightarrow lw t2,4(t1)$
- All of the above are true

What is output?

```c
int main() {
    int *p, x=5, y; // init
    int s;
    y = *(p = &x) + 1;
    printf("x=%d,y=%d,\n", x,y);
    flip_sign(p);
}
```

- $x=5, y=6, *p=-5$
- $x=5, y=6, *p=-5$
- $x=-5, y=4, *p=-5$
- $x=-3, y=-4, *p=-3$

Which of the following is TRUE?

- $addu t0,t1,4(t2)$ is valid MIPS
- $lw t0, t1(t2)$ is valid MIPS
- in $addiu t0,t1,imm$
  - imm is considered an unsigned number that is zero-extended to make it 32 bits wide
- Sign extension means replicating the sign bit when converting from an $n$-bit number to an $n$-bit number, where $n > m$

Which statement is FALSE?

- MIPS uses `jal` to invoke a function and `jr` to return from a function
- `jal` saves PC+1 in `%ra`
- The callee can use temporary registers ($%t$) without saving and restoring them
- The caller can rely on save registers ($%e$) without fear of callee changing them

Pointers in C

- Why use pointers?
  - If we want to pass a large struct or array, it's easier / faster / etc. to pass a pointer than the whole thing
  - In general, pointers allow cleaner, more compact code
- So what are the drawbacks?
  - Pointers are probably the single largest source of bugs in C, so be careful anytime you deal with them
  - Most problematic with dynamic memory management—
    which you will know by the end of the semester, but not for the projects (there will be a lab later in the semester)
  - Dangling references and memory leaks
In MIPS, what is the minimum number of bits does it take to represent 
-1.0 x 2\(^{127}\)?

- 16 bits
- 32 bits
- 64 bits
- 128 bits

---

**Moore’s Law**

“... complexity for minimum component costs has increased at a rate of roughly a factor of two per year... That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000.” (from 50 in 1965)

P = C V\(^2\) f

- Power is proportional to Capacitance \(\times\) Voltage\(^2\) \(\times\) Frequency of switching
- What is the effect on power consumption of:
  - “Simpler” implementation (fewer transistors)?
  - Smaller implementation (shrunk down design)?
  - Reduced voltage?
  - Increased clock frequency?

---

**Great Ideas #5: Measuring Performance**

Restating Performance Equation

\[
\text{Time} = \frac{\text{Seconds}}{\text{Instructions / Program}} \times \text{Clock cycles / Instruction} \times \text{Seconds / Clock Cycle}
\]

---

**What Affects Each Component?**

Instruction Count, CPI, Clock Rate

<table>
<thead>
<tr>
<th>Hardware or software component?</th>
<th>Affects What?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm</td>
<td>Instruction Count, CPI</td>
</tr>
<tr>
<td>Programming Language</td>
<td>Instruction Count, CPI</td>
</tr>
<tr>
<td>Compiler</td>
<td>Instruction Count, CPI</td>
</tr>
<tr>
<td>Instruction Set Architecture</td>
<td>Instruction Count, Clock Rate, CPI</td>
</tr>
</tbody>
</table>
Mapping a 6-bit Memory Address

```
<table>
<thead>
<tr>
<th>Mem Block Within $ Block</th>
<th>Block Within 5 Index</th>
<th>Byte Offset Within Block (e.g., Word)</th>
</tr>
</thead>
</table>
```

- **Note:** $ = Cache
- In example, block size is 4 bytes/1 word (it could be multi-word)
- Memory and cache blocks are the same size, unit of transfer between memory and cache
- # Memory blocks >> # Cache blocks
  - # Memory blocks/16 words/64 bytes/6 bits to address all bytes
  - 4 Cache blocks, 4 bytes (1 word) per block
  - 4 Memory blocks map to each cache block
- Byte within block: low order 2 bits, ignore! (nothing smaller than a block)
- Memory block to cache block, aka **index**: middle two bits
- Which memory block is in a given cache block, aka **tag**: top two bits

---

Caching: A Simple Example

```
<table>
<thead>
<tr>
<th>Cache</th>
<th>Main Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>Valid</td>
</tr>
<tr>
<td>00</td>
<td>0001x</td>
</tr>
<tr>
<td>01</td>
<td>0011x</td>
</tr>
<tr>
<td>01</td>
<td>0101x</td>
</tr>
<tr>
<td>10</td>
<td>1011x</td>
</tr>
<tr>
<td>11</td>
<td>1101x</td>
</tr>
<tr>
<td>11</td>
<td>1111x</td>
</tr>
</tbody>
</table>
```

- One word blocks
- Two lower order bits define the byte in the word (32b words)

Q: Where is the memory block in the cache?

Use next 2 low order memory address bits – the index – to determine which cache block (i.e., modulo the number of blocks in the cache)

(block address % 4) >> 2 (# of blocks in the cache)
Caching: A Simple First Example

Cache

Index | Valid | Tag | Data
---|---|---|---
00 | ✔ | 0x | 0x
01 | ✔ | 1x | 1x
10 | ✔ | 0x | 0x
11 | ✔ | 1x | 1x

Main Memory

0x00xx
0x10xx
0x20xx
0x30xx
0x40xx
0x50xx
0x60xx
0x70xx
0x80xx
0x90xx
0xA0xx
0xB0xx
0xC0xx
0xD0xx
0xE0xx
0xF0xx

Q: Where in the cache is the mem block?

Compare the cache tag to the high order 2 memory address bits to tell if the memory block is in the cache.

Direct Mapped Cache

Start with an empty cache - all blocks initially marked as not valid

For L1 cache

AMAT = Time for a hit + Miss rate x Miss penalty

What is AMAT for L2 cache?

- Time for L2 hit + L2 Miss rate x L2 Miss penalty
- Time for L1 hit + L1 Miss rate x L2 Miss rate x Miss penalty
- Time for L1 hit + L1 Miss rate x (Time for L2 hit + L2 Miss rate x Miss Penalty)
- Time for L1 hit + L1 Miss rate x Miss penalty + Time for L2 hit + L2 Miss rate x Miss penalty

Flynn Taxonomy

<table>
<thead>
<tr>
<th>Instruction Streams</th>
<th>Single</th>
<th>Multiple</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSE, Intel Pentium 4</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>SIMD, Intel Xeon-5345 (Clewell)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- In 2012, SIMD and MIMD most common parallel computers
- SIMD (aka hw-level data parallelism): specialized function units for handling lock-step calculations involving arrays
  - Scientific computing, signal processing, multimedia (audio/video processing)

Intel Architecture SSE2+ 128-Bit SIMD Data Types

- Note: in Intel Architecture ( unlike MIPS) a word is 16 bits
  - Single precision FP: Double word (32 bits)
  - Double precision FP: Quad word (64 bits)

<table>
<thead>
<tr>
<th>Packed Bytes</th>
<th>16 / 128 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>127 126 125 124 123 122 121 120 119 118 117</td>
<td>0</td>
</tr>
<tr>
<td>127 126 125 124 123 122 121 120 119 118 117</td>
<td>0</td>
</tr>
<tr>
<td>127 126 125 124 123 122 121 120 119 118 117</td>
<td>0</td>
</tr>
<tr>
<td>127 126 125 124 123 122 121 120 119 118 117</td>
<td>0</td>
</tr>
<tr>
<td>127 126 125 124 123 122 121 120 119 118 117</td>
<td>0</td>
</tr>
<tr>
<td>127 126 125 124 123 122 121 120 119 118 117</td>
<td>0</td>
</tr>
<tr>
<td>127 126 125 124 123 122 121 120 119 118 117</td>
<td>0</td>
</tr>
</tbody>
</table>

Great Idea #4: Parallelism

- Data Level Parallelism in 1st half 61C
  - Lots of data in memory that can be operated on in parallel (e.g., adding together 2 arrays)
  - Lots of data on many disks that can be operated on in parallel (e.g., searching for documents)
- 1st project: DLP across 10s of servers and disks using MapReduce
- Next week's lab, 3rd project: DLP in memory
Summary

• Amdhal’s Cruel Law: Law of Diminishing Returns
• Loop Unrolling to Expose Parallelism
• Optimize Miss Penalty via Memory system
• As the field changes, cs61c has to change too!
• Still about the software-hardware interface
  – Programming for performance via measurement!
  – Understanding the memory hierarchy and its impact on application performance
  – Unlocking the capabilities of the architecture for performance: SIMD