Review

- To access cache, Memory Address divided into 3 fields: Tag, Index, Block Offset
- Cache size is Data + Management (tags, valid, dirty bits)
- Write misses trickier to implement than reads
  -- Write back vs. Write through
  -- Write allocate vs. No write allocate
- Cache Performance Equations:
  -- CPU time = IC \times CPI_{ideal} \times CC
  -- AMAT = Time for a hit + Miss rate \times Miss penalty
- If understand caches, can adapt software to improve cache performance and thus program performance

Agenda

- Flynn Taxonomy
- Administrivia
- DLP and SIMD
- Technology Break
- Intel SSE
- (Amdahl’s Law if time permits)
Alternative Kinds of Parallelism:

- **Parallelism: Hardware vs. Software**
  
<table>
<thead>
<tr>
<th>Hardware</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>Concurrent</td>
</tr>
<tr>
<td>Parallel</td>
<td>Concurrent</td>
</tr>
</tbody>
</table>
  
  - Concurrent software can also run on serial hardware
  - Sequential software can also run on parallel hardware
  - Focus is on parallel processing software: sequential or concurrent software running on parallel hardware

- **Single Instruction/Single Data Stream**
  
  - Single Instruction, Single Data stream (SISD)
    - Sequential computer that exploits no parallelism in either the instruction or data streams. Examples of SISD architecture are traditional uniprocessor machines

- **Single Instruction/Multiple Data Streams**
  
  - Single Instruction, Multiple Data streams (SIMD)
    - Computer that exploits multiple instruction streams against a single data stream for data operations that can be naturally parallelized. For example, certain kinds of array processors.
    - No longer commonly encountered, mainly of historical interest only

- **Multiple Instruction/Single Data Stream**
  
  - Multiple Instruction, Single Data streams (MISD)
    - Multiple autonomous processors simultaneously executing different instructions on different data.
    - MISD architectures include multicores and Warehouse Scale Computers
    - (Discuss after midterm)

- **Multiple Instruction/Multiple Data Streams**
  
  - Multiple Instruction, Multiple Data streams (MIMD)
    - Multiple autonomous processors simultaneously executing different instructions on different data.
    - MIMD architectures include multicores and Warehouse Scale Computers
    - (Discuss after midterm)

- **Flynn Taxonomy**
  
  - In 2011, SIMD and MIMD most common parallel computers
  - Most common parallel processing programming style:
    - Single Program Multiple Data (“SPMD”)
      - Single program that runs on all processors of an MIMD
      - Cross-processor execution coordination through conditional expressions (thread parallelism after midterm)
    - SIMD (aka hw-level data parallelism): specialized function units, for handling lock-step calculations involving arrays
      - Scientific computing, signal processing, multimedia (audio/video processing)
Data-Level Parallelism (DLP) (from 2nd lecture, January 20)

• 2 kinds of DLP
  – 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)
• 2nd lecture (and 1st project) did DLP across 10s of servers and disks using MapReduce
• Today's lecture (and 3rd project) does Data Level Parallelism (DLP) in memory

SIMD Architectures

• Data parallelism: executing one operation on multiple data streams
  
  • Example to provide context:
    – Multiplying a coefficient vector by a data vector (e.g., in filtering)
      \[ y[i] := c[i] \times x[i], \ 0 \leq i < n \]
  
  • Sources of performance improvement:
    – One instruction is fetched & decoded for entire operation
    – Multiplications are known to be independent
    – Pipelining/concurrency in memory access as well

“Advanced Digital Media Boost”

• To improve performance, Intel’s SIMD instructions
  – Fetch one instruction, do the work of multiple instructions
  – MMX (MMultimedia eXtension, Pentium II processor family)
  – SSE (Streaming SIMD Extension, Pentium III and beyond)

Example: SIMD Array Processing

for each f in array
  \( f = \text{sqrt}(f) \)

for each f in array
  \{ 
  load f to the floating-point register
  calculate the square root
  write the result from the register to memory
  \}

for each 4 members in array
  \{ 
  load 4 members to the SSE register
  calculate 4 square roots in one operation
  write the result from the register to memory
  \}

Administrivia

• Lab #7 posted
• Midterm in 1 week:
  – Exam: Tu, Mar 8, 6-9 PM, 145/155 Dwinelle
    • Split: A-low in 145, I-J in 155
  – Covers everything through lecture March 3
  – Closed book, can bring one sheet notes, both sides
  – Copy of Green card will be supplied
  – No phones, calculators, ... just bring pencils & eraser
  – TA Review: Su, Mar 6, 2-5 PM, 2050 VLSB
• Sent (anonymous) 61C midway survey before Midterm

Scores on Project 2 Part 2

- Top 25%: ≥79 / 85
- Next 50%: ≥60, <79 / 85
Inclusive: all welcome, it works!

- 82%: reafirms CS major, will finish degree
- 30%: undergrads, 40%: grads, 30%

Luminaries: Deborah Estrin UCLA, Blaise Aguera y Arcas Microsoft, Alan Eustace Google, Bill Wulf UVA, Irving Wladawsky-Berger IBM, John Kubiatowicz UC Berkeley

Rising Stars: Hicks Rice, Howard Georgia Tech, Lopez Intel

General Chair: Dave Patterson

If you care, come!

Volunteer poster for student opportunities (work remote or import student)

Encourage grad students to apply doctoral consortium in beautiful San Francisco tapiaconference.org/2011

CDC is a joint org. of ACM,IEEE/CS,CRA

8 great speakers
- Workshops on Grad School Success, Early Career Success, Resume Preparation + BOFs
- Banquet and Dance
- San Francisco Activity: Alcatraz Tour, Chinatown, Bike over Golden Gate Bridge, ...

If interested in diversity, by today (3/1) email Sheila Humphrys with name, year, topic interest + 2 to 3 sentences why want to go to Tapia

humphrys@EECS.Berkeley.EDU

SSE Instruction Categories for Multimedia Support

<table>
<thead>
<tr>
<th>Instruction category</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signed add/subtract</td>
<td>8/128-bit</td>
</tr>
<tr>
<td>Signed compare</td>
<td>8/128-bit</td>
</tr>
<tr>
<td>Average had-min</td>
<td>8/16-bit</td>
</tr>
<tr>
<td>Shift left/right</td>
<td>8/16-bit</td>
</tr>
</tbody>
</table>

• SSE-2+ supports wider data types to allow 16 x 8-bit and 8 x 16-bit operands

Intel Architecture SSE2+

128-Bit SIMD Data Types

Fundamental 128-Bit Packed SIMD Data Types

<table>
<thead>
<tr>
<th>Packed Bytes</th>
<th>16 / 128 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packed Words</td>
<td>8 / 128 bits</td>
</tr>
<tr>
<td>Packed Doublewords</td>
<td>4 / 128 bits</td>
</tr>
</tbody>
</table>

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)

XMM Registers

<table>
<thead>
<tr>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMM7</td>
</tr>
<tr>
<td>XMM6</td>
</tr>
<tr>
<td>XMM5</td>
</tr>
<tr>
<td>XMM4</td>
</tr>
<tr>
<td>XMM3</td>
</tr>
<tr>
<td>XMM2</td>
</tr>
<tr>
<td>XMM1</td>
</tr>
<tr>
<td>XMM0</td>
</tr>
</tbody>
</table>

Architecture extended with eight 128-bit data registers: XMM registers
- IA 64-bit address architecture: available as 16 64-bit registers (XMM8 – XMM15)
- E.g., 128-bit packed single-precision floating-point data type (doublewords) allow four single-precision operations to be performed simultaneously
SSE/SSE2 Floating Point Instructions

Example: Add Two Single Precision FP Vectors

Computation to be performed:

\[
\begin{align*}
vec_{res}.x &= v1.x + v2.x; \\
vec_{res}.y &= v1.y + v2.y; \\
vec_{res}.w &= v1.w + v2.w;
\end{align*}
\]

SSE Instruction Sequences:

\[
\begin{align*}
\text{mov} a.p : \text{mov from mem to XMM register,} \\
\text{memory aligned, packed single precision}
\end{align*}
\]

Example: Image Converter

- Converts BMP (bitmap) image to a YUV (color space) image format:
  - Read individual pixels from the BMP image, convert pixels into YUV format
  - Can pack the pixels and operate on a set of pixels with a single instruction
- E.g., bitmap image consists of 8 bit monochrome pixels
  - Pack these pixel values in a 128 bit register (8 bit * 16 pixels), can operate on 16 values at a time
  - Significant performance boost

Example: Image Converter

Floating point numbers \( f(n) \) and \( x(n) \) in src1 and src2; \( p \) in dest;

C implementation for \( N = 4 \) (128 bits):

```c
for (int i = 0; i < 4; i++)
    p = p + src1[i] * src2[i];
```

Regular x86 instructions for the inner loop:

```assembly
// src1 is on the top of the stack, src2 -> src1
fmul w8 %xmm1, %xmm2 // xmm2 * xmm1 -> xmm0
addps %xmm2, %xmm0 // xmm0 + xmm2 -> xmm0
```

Number regular instructions executed: 2 SSE2 instructions vs. 8 x86

SSE5 instruction accomplishes same in one instruction:

```assembly
fmulps %xmm0, %xmm1, %xmm2, %xmm3 // xmm2 * xmm1 -> xmm0
addps %xmm2, %xmm0 // xmm0 + xmm2 -> xmm0
```

Number regular instructions executed: 1 SSE5 instruction vs. 8 x86
Intel SSE Intrinsics

- Intrinsics are C functions and procedures for putting in assembly language, including SSE instructions
  - With intrinsics, can program using these instructions indirectly
  - One-to-one correspondence between SSE instructions and intrinsics

Example: 2 x 2 Matrix Multiply

Definition of Matrix Multiply:

\[
C_{ij} = (A \times B)_{ij} = \sum_{k=1}^{2} A_{ik} \times B_{kj}
\]

\[
\begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix}
\begin{bmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{bmatrix}
\Rightarrow
\begin{bmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{bmatrix}
\]

Example: 2 x 2 Matrix Multiply

- Initialization

C_1 = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}

C_2 = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}

- I = 1

\[
\text{_mm_load_1 pd: Load 2 doubles into XMM reg, stored in memory in Column order}
\]

\[
\begin{bmatrix}
A_{11} & A_{12} \\
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{bmatrix}
\]

Example SSE Intrinsics

Intrinsics:
- Vector data type: _m128d
- Load and store operations:
  - _mm_load_pd: MOVAPD/aligned, packed double
  - _mm_loadu_pd: MOVUPD/unaligned, packed double
- Load and broadcast across vector
  - _mm_load1_pd: MOVSD/aligned, packed double
- Arithmetic:
  - _mm_add_pd: ADDPD/add, packed double
  - _mm_mul_pd: MULPD/multiply, packed double

Corresponding SSE instructions:
- _mm_load1_pd: MOVSD/aligned, packed double
- _mm_add_pd: ADDPD/add, packed double
- _mm_mul_pd: MULPD/multiply, packed double

Example: 2 x 2 Matrix Multiply

- Using the XMM registers
  - 64-bit/double precision/two doubles per XMM reg

\[
\begin{bmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{bmatrix}
\]

Stored in memory in Column order

\[
\begin{bmatrix}
A_{11} & A_{12} \\
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{bmatrix}
\]
Example: 2 x 2 Matrix Multiply

- First iteration intermediate result

\[
\begin{bmatrix}
A_1 & A_2 \\
B_1 & B_2
\end{bmatrix}
\]

\[
\text{c1} = \text{mm_load1_pd}(C[0])
\]

\[
\text{c2} = \text{mm_add_pd}(\text{c1}, \text{mm_mul_pd}(A_1, B_1))
\]

\[
\text{c1} = \text{mm_load1_pd}(C[1])
\]

\[
\text{c2} = \text{mm_add_pd}(\text{c1}, \text{mm_mul_pd}(A_2, B_2))
\]

\[
\text{C} = \begin{bmatrix}
C[0] & C[1] \\
\end{bmatrix}
\]

- Second iteration intermediate result

\[
\begin{bmatrix}
A_1 & A_2 \\
B_1 & B_2
\end{bmatrix}
\]

\[
\text{c1} = \text{mm_load1_pd}(C[0])
\]

\[
\text{c2} = \text{mm_add_pd}(\text{c1}, \text{mm_mul_pd}(A_1, B_1))
\]

\[
\text{c1} = \text{mm_load1_pd}(C[1])
\]

\[
\text{c2} = \text{mm_add_pd}(\text{c1}, \text{mm_mul_pd}(A_2, B_2))
\]

\[
\text{C} = \begin{bmatrix}
C[0] & C[1] \\
\end{bmatrix}
\]

Live Example: 2 x 2 Matrix Multiply

Definition of Matrix Multiply:

\[
C_{ij} = \sum_{k=1}^{2} A_{ik} B_{kj}
\]

Example: 2 x 2 Matrix Multiply (Part 1 of 2)

Example: 2 x 2 Matrix Multiply (Part 2 of 2)
Inner loop from gcc -O -S

L2: movapd (%rax,%rsi), %xmm1 //Load aligned A[i,i+1] -> m1
movddup (%rdx), %xmm0 //Load B[j], duplicate -> m0
mulpd %xmm1, %xmm0 //Multiply m0*m1 -> m0
addpd %xmm0, %xmm3 //Add m0+m3 -> m3
movddup 16(%rdx), %xmm0 //Load B[j+1], duplicate -> m0
mulpd %xmm0, %xmm1 //Multiply m0*m1 -> m1
addpd %xmm1, %xmm2 //Add m1+m2 -> m2
addq $16,%rax //rax+16 -> rax (i+=2)
addq $8,%rdx //rdx+8 -> rdx (j+=1)
cmpq $32,%rax //rax == 32?
jne L2 //jump to L2 if not equal
movapd %xmm3, (%rcx) //store aligned m3 into C[k,k+1]
movapd %xmm2, (%rdi) //store aligned m2 into C[l,l+1]

Performance-Driven ISA Extensions

• Subword parallelism, used primarily for multimedia applications
  – Intel MMX: multimedia extension
  – 64-bit registers can hold multiple integer operands
  – Intel SSE: Streaming SIMD extension
  – 128-bit registers can hold several floating-point operands
• Adding instructions that do more work per cycle
  – Shift add: replace two instructions with one (e.g., multiply by 5)
  – Multiply-add: replace two instructions with one (x := c + a \times b)
  – Multiply-accumulate: reduce round-off error (s := s + a \times b)
  – Conditional copy: to avoid some branches (e.g., in if-then-else)

Big Idea: Amdahl’s (Heartbreaking) Law

• Speedup due to enhancement E is
  \[
  \text{Speedup} = \frac{\text{Execution time w/o E}}{\text{Execution time w/ E}}
  \]
• Suppose that enhancement E accelerates a fraction \(F\) of the task by a factor \(S\) and the remainder of the task is unaffected

Execution Time w/ E = Execution Time w/o E \times [1-F] + F/S

\[
\text{Speedup} = \frac{1}{\left[\frac{1-F}{S}\right]} + F/S
\]

Example: the execution time of half of the program can be accelerated by a factor of 2. What is the program speed-up overall?

\[
\frac{1}{0.5 + 0.25} = \frac{1}{0.75} = 1.33
\]

Big Idea: Amdahl’s Law

Example: the execution time of half of the program can be accelerated by a factor of 2. What is the program speed-up overall?

\[
\frac{1}{0.5 + 0.25} = \frac{1}{0.75} = 1.33
\]
### Example #1: Amdahl’s Law

\[
\text{Speedup } w/E = 1 / ( (1-F) + F/S )
\]

- Consider an enhancement which runs 20 times faster but which is only usable 25% of the time
  \[
  \text{Speedup } w/E = 1/(.75 + .25/20) = 1.31
  \]
- What if its usable only 15% of the time?
  \[
  \text{Speedup } w/E = 1/(.85 + .15/20) = 1.17
  \]
- 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
  \[
  \text{Speedup } w/E = 1/(.001 + .999/100) = 90.99
  \]

### Example #2: Amdahl’s Law

\[
\text{Speedup } w/E = 1 / ( (1-F) + F/S )
\]

- Consider summing 10 scalar variables and two 10 by 10 matrices (matrix sum) on 10 processors
  \[
  \text{Speedup } w/E = 1/(.091 + .909/10) = 5.5
  \]
- What if there are 100 processors?
  \[
  \text{Speedup } w/E = 1/(.091 + .909/100) = 10.0
  \]
- What if the matrices are 100 by 100 (or 10,010 adds in total) on 10 processors?
  \[
  \text{Speedup } w/E = 1/(.001 + .999/10) = 9.9
  \]
- What if there are 100 processors?
  \[
  \text{Speedup } w/E = 1/(.001 + .999/100) = 91
  \]

### 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
  - **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
- **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

### 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**
  - Embed the SSE machine instructions directly into C programs through use of intrinsics
  - Achieve efficiency beyond that of optimizing compiler