CS 61C: Great Ideas in Computer Architecture
SIMD I

Instructor:
David A. Patterson
http://inst.eecs.Berkeley.edu/~cs61c/sp12

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 × CPIideal × CC
  - AMAT = Time for a hit + Miss rate × Miss penalty
- If understand caches, can adapt software to improve cache performance and thus program performance

Agenda

- Flynn Taxonomy
- Administrivia
- DLP and SIMD
- Intel Streaming SIMD Extensions (SSE)
- (Amdahl’s Law if time permits)

Alternative Kinds of Parallelism:
The Programming Viewpoint

- Job-level parallelism/process-level parallelism
  - Running independent programs on multiple processors simultaneously
    - Example?
- Parallel processing program
  - Single program that runs on multiple processors simultaneously
    - Example?
Alternative Kinds of Parallelism:
Multiple Instruction/Single Data Stream

- Multiple Instruction, Single Data streams (MISD)
  - 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.

Alternative Kinds of Parallelism:
Single Instruction/Multiple Data Stream

- Single Instruction, Multiple Data streams (SIMD or "sim-dee")
  - Computer that exploits multiple data streams against a single instruction stream to operations that may be naturally parallelized, e.g., Intel SIMD instruction extensions or NVIDIA Graphics Processing Unit (GPU)

Flynn Taxonomy

<table>
<thead>
<tr>
<th>Data Streams</th>
<th>Single</th>
<th>Multiple</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>SIMD: Intel Pentium 4</td>
<td>SIMD: SSE instructions of x86</td>
</tr>
<tr>
<td>Streams</td>
<td>MISD: No examples today</td>
<td>MIMD: Intel Xeon (or greater)</td>
</tr>
</tbody>
</table>

- In 2012, 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 a 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 19)

- 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 (MultiMedia eXtension, Pentium II processor family)
  – SSE (Streaming SIMD Extension, Pentium III and beyond)

Example: SIMD Array Processing

for each \( f \) in array
\[
E = \sqrt{f}
\]
for each \( f \) in array
\[
\{ \\
\quad \text{load } f \text{ to the floating-point register} \\
\quad \text{calculate the square root} \\
\quad \text{write the result from the register to memory}
\}
\]
for each 4 members in array
\[
\{ \\
\quad \text{load 4 members to the SSE register} \\
\quad \text{calculate 4 square roots in one operation} \\
\quad \text{store the 4 results from the register to memory}
\}
\]
SIMD style

Intel SSE Instruction Categories for Multimedia Support

<table>
<thead>
<tr>
<th>Instruction category</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsigned add/subtract</td>
<td>Eight 8-bit or Four 16-bit</td>
</tr>
<tr>
<td>Saturating add/subtract</td>
<td>Eight 8-bit or Four 16-bit</td>
</tr>
<tr>
<td>Max/Min/Minimum</td>
<td>Eight 8-bit or Four 16-bit</td>
</tr>
<tr>
<td>Average</td>
<td>Eight 8-bit or Four 16-bit</td>
</tr>
<tr>
<td>Shift right/left</td>
<td>Eight 8-bit or Four 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

• 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)

Administrivia

• Lab #7 posted
• Midterm in 1 week:
  – 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
  – TA Review: Su, Mar 4, Starting 2PM, 2050 VLSB
• Will send (anonymous) 61C midway survey before Midterm

Project 2, Part 2 Grades

• On-time submissions grades
• Majority got perfect or near-perfect scores
• Median 81.5, Mean 74.1
Most Common Test Failures

<table>
<thead>
<tr>
<th>Project</th>
<th>Test Case</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proj 2</td>
<td>SSE/SSE2</td>
<td>Floating Point Instructions</td>
</tr>
<tr>
<td></td>
<td>SSE/SSE2</td>
<td>Floating Point Instructions</td>
</tr>
</tbody>
</table>

61C in the News

“The smartphone market last year was a half billion units,” [Timothy D. Cook, the Apple chief executive] continued. “In 2015, it is projected to be a billion units. When you take it in the context of these numbers, the truth is, this is a jaw-dropping industry.” ...


SSE/SSE2 Floating Point Instructions

Example: Add Two Single Precision FP Vectors

Computation to be performed:

```
vec_res = v1.x + v2.x;
vec_res.y = v1.y + v2.y;
vec_res.z = v1.z + v2.z;
vec_res.w = v1.w + v2.w;
```

```asm
mov a p; mov from mem to XMM register.
add p; add from mem to XMM register;
packed single precision
```

SSE Instruction Sequence:

(Not: Destination on the right in x86 assembly)

```asm
madps address-of-v1, v1, v2.w
```

Example: Add Two Single Precision FP Vectors

```
vec_res = v1.x + v2.x;
vec_res.y = v1.y + v2.y;
vec_res.z = v1.z + v2.z;
vec_res.w = v1.w + v2.w;
```

Project 2

• Flynn Taxonomy
• Admintrivia
• DLP and SIMD
• Technology Break
• Intel Streaming SIMD Extensions (SSE)
• (Amdahl’s Law if time permits)
Example: Image Converter

- **FMADDPS** – Multiply and add packed single precision floating point instruction
- One of the typical operations computed in transformations (e.g., DFT of FFT)

\[
P = \sum_{n=1}^{N} f(n) \times x(n)
\]

Example: SSE Image Converter

Floating point numbers \( f(n) \) and \( x(n) \) in src1 and src2; \( p \) in dest; C implementation for \( N = 4 \) (128 bits):

\[
\begin{align*}
\text{for } (\text{int } i & = 0; \ i < 4; \ i++) \\
p & = p + \text{src1}[i] \times \text{src2}[i];
\end{align*}
\]

- SSE2 instructions for the inner loop:
  - xmm0 = p, xmm1 = src1[i], xmm2 = src2[i]
  - mulps %xmm1, %xmm2 // xmm2 = xmm1 * xmm2
  - addps %xmm2, %xmm0 // xmm0 = xmm2 + xmm0
- Number regular instructions executed: 2 SSE2 instructions vs. 8 x86
- SSE instruction accomplishes same in one instruction:
  - fmaddps %xmm0, %xmm1, %xmm2, %xmm0
  - // xmm2 * xmm1 + xmm0 -> xmm0
- Number regular instructions executed: 1 SSE instruction vs. 8 x86

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):

\[
\begin{align*}
\text{for } (\text{int } i & = 0; \ i < 4; \ i++) \\
p & = p + \text{src1}[i] \times \text{src2}[i];
\end{align*}
\]

- Regular x86 instructions for the inner loop:
  - // src1 is on the top of the stack; src1 * src2 -> src1
  - fmul DWORD PTR _src2[esp+148]
  - /p = ST(1), src1 = ST(0); ST(0)+ST(1) -> ST(1); ST-Stack Top
  - faddp %ST(0), %ST(1)

(Note: Destination on the right in x86 assembly)

Number regular x86 Fl. Pt. instructions executed: 4 * 2 = 8

Example: Image Converter

- 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

Footer:
2/28/12
Example SSE Intrinsics

- Vector data type: \_m128d
- Load and store operations:
  - \_mm\_load\_pd: MOVAPD/aligned, packed double
  - \_mm\_store\_pd: MOVUPD/unaligned, packed double
  - Load and broadcast across vector
  - \_mm\_load1\_pd: MOVSD + shuffling/duplicating
- Arithmetic:
  - \_mm\_add\_pd: ADDPD/add, packed double
  - \_mm\_mul\_pd: MMADDPD/multiply, packed double

Example: \(2 \times 2\) Matrix Multiply

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

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

\[
\begin{bmatrix}
A_{1,1} & A_{1,2} \\
A_{2,1} & A_{2,2}
\end{bmatrix}
\times
\begin{bmatrix}
B_{1,1} & B_{1,2} \\
B_{2,1} & B_{2,2}
\end{bmatrix}
= \begin{bmatrix}
C_{1,1} & C_{1,2} \\
C_{2,1} & C_{2,2}
\end{bmatrix}
\]

Example: \(2 \times 2\) Matrix Multiply

- Initialization
  - \(C_{1} = \begin{bmatrix} 0 & 0 \\
0 & 0 \end{bmatrix}\)
  - \(C_{2} = \begin{bmatrix} 0 & 0 \\
0 & 0 \end{bmatrix}\)

Example: \(2 \times 2\) Matrix Multiply

- First iteration intermediate result
  - \(c1 = \_mm\_add\_pd((c1, \_mm\_mul\_pd(a1,b1)));\)
  - \(c2 = \_mm\_add\_pd((c2, \_mm\_mul\_pd(a2,b2)));\)
  - SSE instructions first do parallel multiplies and then parallel adds in XMM registers

Example: \(2 \times 2\) Matrix Multiply

- \(i = 1\)
  - \(A = \begin{bmatrix} A_{11} & A_{12} \\
A_{21} & A_{22} \end{bmatrix}\)
  - \(B = \begin{bmatrix} B_{11} & B_{12} \\
B_{21} & B_{22} \end{bmatrix}\)
  - \(\_mm\_load1\_pd\): SSE instruction that loads a double word and stores it in the high and low double words of the XMM register (duplicates value in both halves of XMM)
Example: 2 x 2 Matrix Multiply

- First iteration intermediate result

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

- Second iteration intermediate result

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

Live Example: 2 x 2 Matrix Multiply

Definition of Matrix Multiply:

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

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
mulpd %xmm1, %xmm0
addpd %xmm0, %xmm3
movddup (%rdx), %xmm0
mulpd %xmm0, %xmm3
addpd %xmm0, %xmm3
movapd (%rdx, %rsi), %xmm2
mulpd %xmm2, %xmm1
addpd %xmm1, %xmm2
movq $8, %rdx
cmpq $32, %rax
jne L2
// jump to L2 if not equal
movapd %xmm3, (%rcx)
movapd %xmm2, (%rdi)
// store aligned m3 into C(k,k+1)
// store aligned m2 into C(j,j+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: reduce round-off error (e.g., in if-then-else)

Big Idea: Amdahl’s (Heartbreaking) Law

- Speedup due to enhancement $E$ is
  \[
  \text{Speedup} \left( \text{w/o } E \right) = \frac{1}{(1-F) + F/S} 
  \]

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

\[
\text{Non-speed-up part} \quad \text{Speed-up part} \\
\frac{1}{2} \cdot 2 = 1.00 
\]

Example #1: Amdahl’s Law

- Consider an enhancement which runs 20 times faster but which is only usable 25% of the time
  \[
  \text{Speedup} \left( \text{w/o } E \right) = \frac{1}{(1-0.25) + 0.25/20} = 1.31
  \]

- What if its usable only 15% of the time?
  \[
  \text{Speedup} \left( \text{w/o } E \right) = \frac{1}{(1-0.15) + 0.15/20} = 1.17
  \]

- 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
  \[
  \text{Speedup} \left( \text{w/o } E \right) = \frac{1}{(1-0.001 + 0.999/100)} = 90.99
  \]
Example #2: Amdahl’s Law

Speedup with E = \frac{1}{[1-(1-F)/F]}:

- Consider summing 10 scalar variables and two 10 by 10 matrices (matrix sum) on 10 processors:
  
  \text{Speedup with } E = \frac{1}{0.091 + 0.909/10} = 1/0.1819 = 5.5

- What if there are 100 processors?
  
  \text{Speedup with } E = \frac{1}{0.091 + 0.909/100} = 1/0.10009 = 10.0

- What if the matrices are 100 by 100 (or 10,010 adds in total) on 10 processors?
  
  \text{Speedup with } E = \frac{1}{0.001 + 0.999/10} = 1/0.1009 = 9.9

- What if there are 100 processors?
  
  \text{Speedup with } E = \frac{1}{0.001 + 0.999/100} = 1/0.01099 = 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