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

Instructors:
Randy H. Katz
David A. Patterson
http://inst.eecs.Berkeley.edu/~cs61c/fa10

Agenda

• Intel SSE SIMD Instructions
• Administrivia
• Technology Break
• SSE in C
Agenda

• Intel SEE SIMD Instructions
• Administrivia
• Technology Break
• SSE in C

Single Instruction/Multiple Data Stream

• Single Instruction, Multiple Data streams (SIMD)
  - Computer that exploits multiple data streams against a single instruction stream to operations that may be naturally parallelized, e.g., an array processor or Graphics Processing Unit (GPU)
“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
\[
f = \sqrt{f}
\]
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>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 128-Bit SIMD Data Types

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

<table>
<thead>
<tr>
<th>127</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMM7</td>
<td></td>
</tr>
<tr>
<td>XMM6</td>
<td></td>
</tr>
<tr>
<td>XMM5</td>
<td></td>
</tr>
<tr>
<td>XMM4</td>
<td></td>
</tr>
<tr>
<td>XMM3</td>
<td></td>
</tr>
<tr>
<td>XMM2</td>
<td></td>
</tr>
<tr>
<td>XMM1</td>
<td></td>
</tr>
<tr>
<td>XMM0</td>
<td></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), allows four single-precision operations to be performed simultaneously

### SSE/SSE2 Floating Point Instructions

<table>
<thead>
<tr>
<th>Data transfer</th>
<th>Arithmetic</th>
<th>Compare</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (U) (SS/PS/SD/ PD) xmm, mem/xmm</td>
<td>ADD (SS/PS/SD/ PD) xmm, xmm</td>
<td>CMP (SS/PS/SD/ PD)</td>
</tr>
<tr>
<td>MOV (H/L) (PS/PD) xmm, mem/xmm</td>
<td>SUB (SS/PS/SD/ PD) xmm, xmm</td>
<td></td>
</tr>
<tr>
<td>MUL (SS/PS/SD/ PD) xmm, xmm</td>
<td>INT (SS/PS/SD/ PD) mem/xmm</td>
<td></td>
</tr>
<tr>
<td>DIV (SS/PS/SD/ PD) xmm, xmm</td>
<td>SQRT (SS/PS/SD/ PD) mem/xmm</td>
<td></td>
</tr>
<tr>
<td>MAX (SS/PS/SD/ PD) mem/xmm</td>
<td>MIN (SS/PS/SD/ PD) mem/xmm</td>
<td></td>
</tr>
</tbody>
</table>

- xmm: one operand is a 128-bit SSE2 register
- mem/xmm: other operand is in memory or an SSE2 register
- (SS) Scalar Single precision FP: one 32-bit operand in a 128-bit register
- (PS) Packed Single precision FP: four 32-bit operands in a 128-bit register
- (SD) Scalar Double precision FP: one 64-bit operand in a 128-bit register
- (PD) Packed Double precision FP: four 64-bit operands in a 128-bit register
- (U) 128-bit operand is unaligned in memory
- (H) means move the high half of the 128-bit operand
- (L) means move the low half of the 128-bit operand
Example: Add Two Quad Word Vectors

Computation to be performed:

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

SSE Instruction Sequence:

\[
\begin{align*}
\text{mov a ps:} & \text{move from mem to XMM register, memory aligned, packed single precision} \\
\text{movaps xmm0, address-of-v1} & ; \text{xmm0}=v1.w | v1.z | v1.y | v1.x \\
\text{addps xmm0, address-of-v2} & ; \text{xmm0}=v1.w+v2.w | v1.z+v2.z | v1.y+v2.y | v1.x+v2.x \\
\text{movaps address-of-vec_res, xmm0}
\end{align*}
\]

Displays and Pixels

- Each coordinate in frame buffer on left determines shade of corresponding coordinate for the raster scan CRT display on right. Pixel (X0, Y0) contains bit pattern 0011, a lighter shade on the screen than the bit pattern 1101 in pixel (X1, Y1)
Example: Image Converter

• Converts BMP image to a YUV 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

• 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: Image Converter

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

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

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

SSE2 instructions for the inner loop:

```c
//\( \text{xmm0} = p, \text{xmm1} = \text{src1}, \text{xmm2} = \text{src2} \)
mulps xmm1, xmm2
addps xmm0, xmm1
```

SSE5 instruction accomplishes same in one instruction:

```c
//\( \text{xmm0} = p, \text{xmm1} = \text{src1}, \text{xmm2} = \text{src2} \)
fmaddps xmm0, xmm1, xmm2, xmm0
```

Agenda

- Intel SEE SIMD Instructions
- Administrivia
- Technology Break
- SSE in C
Administrivia

• Midterm regrade requests one week from tomorrow (19 October 2010)
• EC2 Project #2 posted, Part I due Saturday, Part II following Saturday, 1 sec to Midnight
• This week: Intel SSE/Data Parallelism Lab; Next week: Thread Parallelism Lab;
• Note Project 4 “Competition” coming at end of semester
• Clarification of “doing your own work”
• Comments from the Course Survey

Agenda

• Intel SSE SIMD Instructions
• Administrivia
• Technology Break
• SSE in C
Agenda

- Intel SEE SIMD Instructions
- Administrivia
- Technology Break
- SSE in C

Intel SSE Intrinsics

- Intrinsics are C functions and procedures for SSE instructions
  - With intrinsics, can program using these instructions indirectly
  - One-to-one correspondence between SSE instructions and intrinsics
Example SSE Intrinsics

- Vector data type: 
  \_m128d

Load and store operations:
- \_mm_load_pd	MOVAPD/aligned, packed double
- \_mm_store_pd	MOVAPD/aligned, packed double
- \_mm_loadu_pd	MOVUPD/unaligned, packed double
- \_mm_storeu_pd	MOVUPD/unaligned, packed double

Load and broadcast across vector
- \_mm_load1_pd	MOVSD + shuffling

Arithmetic:
- \_mm_add_pd	ADDPD/add, packed double
- \_mm_mul_pd	MULPD/multiple, packed double

Example: 2 x 2 Matrix Multiply

Definition of Matrix Multiply:
\[
C_{i,j} = (A \times B)_{i,j} = \sum_{k=1}^{2} A_{i,k} \times B_{k,j}
\]

\[
\begin{bmatrix}
A_{1,1} & A_{1,2} \\
A_{2,1} & A_{2,2}
\end{bmatrix}
\times
\begin{bmatrix}
B_{1,1} \\
B_{2,1}
\end{bmatrix}
= \begin{bmatrix}
C_{1,1}=A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\
C_{2,1}=A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\
C_{1,2}=A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\
C_{2,2}=A_{2,1}B_{1,2} + A_{2,2}B_{2,2}
\end{bmatrix}
\]
Example: 2 x 2 Matrix Multiply

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

\[
\begin{align*}
\mathbf{C} & = \begin{pmatrix}
C_{1,1} & C_{1,2} \\
C_{2,1} & C_{2,2}
\end{pmatrix} & \text{Stored in memory in Column order} \\
\mathbf{A} & = \begin{pmatrix}
A_{1,1} \\
A_{2,1}
\end{pmatrix} \\
\mathbf{B} & = \begin{pmatrix}
B_{1,1} & B_{1,2} \\
B_{2,1} & B_{2,2}
\end{pmatrix}
\end{align*}
\]

Example: 2 x 2 Matrix Multiply

• Initialization

\[
\begin{align*}
\mathbf{C} & = \begin{pmatrix}
0 & 0 \\
0 & 0
\end{pmatrix}
\end{align*}
\]
Example: 2 x 2 Matrix Multiply

• Initialization

| C₁ | 0 | 0 |
| C₂ | 0 | 0 |

• I = 1

| A  | A₁₁ | A₁₂ |
| B₁ | B₁₁ | B₁₂ |
| B₂ | B₂₁ | B₂₂ |

_mm_load_pd: Stored in memory in Column order
_mm_load1_pd: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register

Example: 2 x 2 Matrix Multiply

• First iteration intermediate result

| C₁ | 0+A₁₁B₁₁ | 0+A₁₂B₁₂ |
| C₂ | 0+A₁₁B₂₁ | 0+A₁₂B₂₂ |

c₁ = _mm_add_pd(c₁, _mm_mul_pd(a, b₁));
c₂ = _mm_add_pd(c₂, _mm_mul_pd(a, b₂));

• I = 1

| A  | A₁₁ | A₁₂ |
| B₁ | B₁₁ | B₁₂ |
| B₂ | B₂₁ | B₂₂ |

_mm_load_pd: Stored in memory in Column order
_mm_load1_pd: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register
Example: 2 x 2 Matrix Multiply

• First iteration intermediate result

<table>
<thead>
<tr>
<th></th>
<th>C_1</th>
<th>C_2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0+A_{1,1}B_{1,1}</td>
<td>0+A_{1,1}B_{1,1}</td>
</tr>
<tr>
<td></td>
<td>0+A_{1,1}B_{1,2}</td>
<td>0+A_{2,1}B_{1,2}</td>
</tr>
</tbody>
</table>

c_1 = _mm_add_pd(c_1, _mm_mul_pd(a, b));
c_2 = _mm_add_pd(c_2, _mm_mul_pd(a, b));

• \( I = 2 \)

<table>
<thead>
<tr>
<th>A</th>
<th>A_{1,2}</th>
<th>A_{2,2}</th>
</tr>
</thead>
</table>

\_mm_load\_pd: Stored in memory in Column order

<table>
<thead>
<tr>
<th>B_1</th>
<th>B_{2,1}</th>
<th>B_{2,1}</th>
</tr>
</thead>
</table>

| B_2 | B_{2,2} | B_{2,2} |

\_mm_load1\_pd: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register

Example: 2 x 2 Matrix Multiply

• Second iteration intermediate result

<table>
<thead>
<tr>
<th></th>
<th>C_{1,1}</th>
<th>C_{2,1}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A_{1,1}B_{1,1}+A_{1,1}B_{1,2}</td>
<td>A_{2,1}B_{1,1}+A_{2,1}B_{1,2}</td>
</tr>
<tr>
<td></td>
<td>A_{1,2}B_{1,2}+A_{2,2}B_{2,2}</td>
<td>A_{2,1}B_{2,2}+A_{2,2}B_{2,2}</td>
</tr>
</tbody>
</table>

c_1 = _mm_add_pd(c_1, _mm_mul_pd(a, b));
c_2 = _mm_add_pd(c_2, _mm_mul_pd(a, b));

• \( I = 2 \)

<table>
<thead>
<tr>
<th>A</th>
<th>A_{1,2}</th>
<th>A_{2,2}</th>
</tr>
</thead>
</table>

\_mm_load\_pd: Stored in memory in Column order

<table>
<thead>
<tr>
<th>B_1</th>
<th>B_{2,1}</th>
<th>B_{2,1}</th>
</tr>
</thead>
</table>

| B_2 | B_{2,2} | B_{2,2} |

\_mm_load1\_pd: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register
Example: 2 x 2 Matrix Multiply
(Part 1 of 2)

```c
#include <stdio.h>
// header file for SSE compiler intrinsics
#include <emmintrin.h>

// NOTE: vector registers will be represented in comments as v1 = [a | b]
// where v1 is a variable of type __m128d and a, b are doubles

int main(void)
{
    int lda = 2;
    int i = 0;
    // declare a couple 128-bit vector variables
    __m128d c1, c2, a, b1, b2;

    // allocate A, B, C aligned on 16-byte boundaries
    double B[4] __attribute__((aligned(16)));
    double C[4] __attribute__((aligned(16)));

    // used aligned loads to set
    c1 = _mm_load_pd(C+0*lda);
    c2 = _mm_load_pd(C+1*lda);

    for (i = 0; i < 2; i++) {
        a = _mm_load_pd(A+i*lda);
        b1 = _mm_load1_pd(B+i*lda);
        b2 = _mm_load1_pd(B+(i+1)*lda);
        c1 = _mm_mul_pd(a, b1);  // A[i] * B[i]
        c2 = _mm_mul_pd(a, b2);  // A[i] * B[i+1]
        c1 = _mm_add_pd(c1, _mm_add_pd(c2, a));
    }

    // store c1, c2 back into C for completion
    _mm_store_pd(C+0*lda, c1);
    _mm_store_pd(C+1*lda, c2);

    // print C
    printf("%g,%g\n%g,%g\n", C[0], C[1], C[2], C[3]);
    return 0;
}
```

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

```c
/* A =
  1 0
  0 1
*/

/* B =
  1 3
  2 4
*/
B[0] = 1.0; B[1] = 2.0; B[2] = 3.0; B[3] = 4.0;

/* C =
  0 0
*/
C[0] = 0.0; C[1] = 0.0; C[2] = 0.0; C[3] = 0.0;

// used aligned loads to set
// c1 = [c_11 | c_21]
c1 = _mm_load_pd(C+0*lda);
// c2 = [c_12 | c_22]
c2 = _mm_load_pd(C+1*lda);

for (i = 0; i < 2; i++) {
    /* a =
     i = 0: [a_11 | a_21]
     i = 1: [a_12 | a_22]
     */
    a = _mm_load_pd(A+i*lda);
    /* b1 =
     i = 0: [b_11 | b_12]
     i = 1: [b_21 | b_22]
     */
    b1 = _mm_load1_pd(B+i*lda);
    /* b2 =
     i = 0: [b_12 | b_21]
     i = 1: [b_22 | b_22]
     */
    b2 = _mm_load1_pd(B+(i+1)*lda);
    c1 = _mm_add_pd(c1, _mm_mul_pd(a, b1));
    c2 = _mm_add_pd(c2, _mm_mul_pd(a, b2));
}
```

10/11/10 Fall 2010 -- Lecture #18
29
Summary

• 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