CS 61C: Great Ideas in Computer Architecture
SIMD

Instructor:
Randy H. Katz
http://inst.eecs.Berkeley.edu/~cs61c/fa13

Review

• Three C’s of cache misses
  – Compulsory
  – Capacity
  – Conflict
• Amdahl’s Law: Speedup = 1/((1-F)+F/S)
• Flynn’s Taxonomy: SISD/SIMD/MISD/MIMD
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*)

![Diagram](image-url)
Example: SIMD Array Processing

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

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  
  store the 4 results from the register to memory
\}

SIMD style

Data-Level Parallelism and SIMD

- SIMD wants adjacent values in memory that can be operated in parallel
- Usually specified in programs as loops
  \[
  \text{for}(i=1000; i>0; i=i-1)
  \quad \text{x}[i] = \text{x}[i] + s;
  \]
- How can reveal more data-level parallelism than available in a single iteration of a loop?
- \textit{Unroll loop} and adjust iteration rate
Looping in MIPS

Assumptions:
- $t1$ is initially the address of the element in the array with the highest address
- $f0$ contains the scalar value $s$
- $8(t2)$ is the address of the last element to operate on

CODE:

1. l.d $f2,0(t1)$ ; $f2$ = array element
2. add.d $f10,f2,f0$ ; add $s$ to $f2$
3. s.d $f10,0(t1)$ ; store result
4. addui $t1,t1,#-8$ ; decrement pointer 8 byte
5. bne $t1,t2,Loop$ ; repeat loop if $t1 != t2$

Loop Unrolled

1. l.d $f2,0(t1)$
2. add.d $f10,f2,f0$
3. s.d $f10,0(t1)$
4. l.d $f4,-8(t1)$
5. add.d $f12,f4,f0$
6. s.d $f12,-8(t1)$
7. l.d $f6,-16(t1)$
8. add.d $f14,f6,f0$
9. s.d $f14,-16(t1)$
10. l.d $f8,-24(t1)$
11. add.d $f16,f8,f0$
12. s.d $f16,-24(t1)$
13. addui $t1,t1,#-32$
14. bne $t1,t2,Loop$

NOTE:
1. Only 1 Loop Overhead every 4 iterations
2. This unrolling works if
   \[ \text{loop\_limit(mod 4)} = 0 \]
3. (Different Registers eliminate stalls in pipeline; we'll see later in course)
Loop Unrolled Scheduled

Loop:

```
    l.d $f2,0($t1)
    l.d $f4,-8($t1)
    l.d $f6,-16($t1)
    l.d $f8,-24($t1)
    add.d $f10,$f2,$f0
    add.d $f12,$f4,$f0
    add.d $f14,$f6,$f0
    add.d $f16,$f8,$f0
    s.d $f10,0($t1)
    s.d $f12,-8($t1)
    s.d $f14,-16($t1)
    s.d $f16,-24($t1)
    addui $t1,$t1,#-32
    bne $t1,$t2,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
  ```
  for(i=1000; i>0; i=i-1)
      x[i] = x[i] + s;
  ```
- Could be rewritten
  ```
  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 \( n \) iterations
- \( k \) copies of the body of the loop
- Assuming \((n \mod k) \neq 0\)

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)

Intel SIMD Extensions

- MMX 64-bit registers, reusing floating-point registers [1992]
- SSE2/3/4, new 128-bit registers [1999]
- AVX, new 256-bit registers [2011]
  - Space for expansion to 1024-bit registers
• Architecture extended with eight 128-bit data registers: XMM registers
  – x86 64-bit address architecture adds 8 additional registers (XMM8 – XMM15)

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)
### SSE/SSE2 Floating Point Instructions

<table>
<thead>
<tr>
<th>Data transfer</th>
<th>Arithmetic</th>
<th>Compare</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV{A}/UI</td>
<td>{SS/PS/SD/PD} xmm, mem/xmm</td>
<td>ADD{SS/PS/SD/PD} xmm, mem/xmm</td>
</tr>
<tr>
<td>SUB{SS/PS/SD/PD} xmm, mem/xmm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV (H/L) (PS/PD) xmm, mem/xmm</td>
<td>MUL{SS/PS/SD/PD} xmm, mem/xmm</td>
<td></td>
</tr>
<tr>
<td>DIV{SS/PS/SD/PD} xmm, mem/xmm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQRT{SS/PS/SD/PD} mem/xmm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAX {SS/PS/SD/PD} mem/xmm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MIN{SS/PS/SD/PD} mem/xmm</td>
<td></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, or two 64-bit operands in a 128-bit register
- (A) 128-bit operand is aligned in memory
- (U) means the 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

---

First SIMD Extensions: MIT Lincoln Labs TX-2, 1957
Example: Add Two Single-Precision Floating-Point Vectors

Computation to be performed:

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

SSE Instruction Sequence:
(Note: Destination on the right in x86 assembly)

\[
\begin{align*}
\text{movaps address-of-v1, %xmm0} \\
\text{addps address-of-v2, %xmm0} \\
\text{movaps %xmm0, address-of-vec\_res}
\end{align*}
\]
Intel SSE Intrinsics

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

Example SSE Intrinsics

Intrinsics: Corresponding SSE instructions:
- Vector data type: _m128d
- Load and store operations: 
  _mm_load_pd MOVAPD/aligned, packed double
  _mm_store_pd MOVUPD/unaligned, packed double
  _mm_loadu_pd MOVUPD/unaligned, packed double
  _mm_storeu_pd
- Load and broadcast across vector
  _mm_load1_pd MOVSD + shuffling/duplicating
- 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:

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

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

Example: 2 x 2 Matrix Multiply

• Using the XMM registers
  – Two 64-bit doubles per XMM reg

\[
\begin{array}{c|c|c}
\hline
C_1 & C_{1,1} & C_{1,2} \\
\hline
C_2 & C_{2,1} & C_{2,2} \\
\hline
\end{array}
\]

\[
\begin{array}{c|c}
\hline
A & A_{1,i} & A_{2,i} \\
\hline
B_1 & B_{i,1} & B_{i,2} \\
B_2 & B_{i,1} & B_{i,2} \\
\hline
\end{array}
\]

Stored in memory in Column-major order
Example: 2 x 2 Matrix Multiply

• Initialization

\[
\begin{array}{c|c}
C_1 & 0 \mid 0 \\
C_2 & 0 \mid 0 \\
\end{array}
\]

• \( I = 1 \)

\[
\begin{array}{c|c}
A & A_{1,1} \mid A_{2,1} \\
B_1 & B_{1,1} \mid B_{1,1} \\
B_2 & B_{1,2} \mid B_{1,2} \\
\end{array}
\]

__mm_load_pd__: Load 2 doubles into XMM register, Stored in memory in Column-major 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 (duplicates value in both halves of XMM)
**Example: 2 x 2 Matrix Multiply**

- **First iteration intermediate result**

<table>
<thead>
<tr>
<th>C1</th>
<th>C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 + A1,1 B1,1</td>
<td>0 + A2,1 B1,1</td>
</tr>
</tbody>
</table>

  \[ c1 = \text{mm\_add\_pd}(c1, \text{mm\_mul\_pd}(a, b1)); \]

  \[ c2 = \text{mm\_add\_pd}(c2, \text{mm\_mul\_pd}(a, b2)); \]

  SSE instructions first do parallel multiplies and then parallel adds in XMM registers

- **I = 1**

<table>
<thead>
<tr>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1,1</td>
</tr>
</tbody>
</table>

  \[ \text{mm\_load\_pd}: \text{Stored in memory in Column order} \]

<table>
<thead>
<tr>
<th>B1</th>
<th>B2</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1,1</td>
<td>B1,2</td>
</tr>
<tr>
<td>B2,1</td>
<td>B2,2</td>
</tr>
</tbody>
</table>

  \[ \text{mm\_load1\_pd}: \text{SSE instruction that loads a double word and stores it in the high and low double words of the XMM register (duplicates value in both halves of XMM)} \]

---

**Example: 2 x 2 Matrix Multiply**

- **First iteration intermediate result**

<table>
<thead>
<tr>
<th>C1</th>
<th>C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 + A1,1 B1,1</td>
<td>0 + A2,1 B1,1</td>
</tr>
</tbody>
</table>

  \[ c1 = \text{mm\_add\_pd}(c1, \text{mm\_mul\_pd}(a, b1)); \]

  \[ c2 = \text{mm\_add\_pd}(c2, \text{mm\_mul\_pd}(a, b2)); \]

  SSE instructions first do parallel multiplies and then parallel adds in XMM registers

- **I = 2**

<table>
<thead>
<tr>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1,2</td>
</tr>
</tbody>
</table>

  \[ \text{mm\_load\_pd}: \text{Stored in memory in Column order} \]

<table>
<thead>
<tr>
<th>B1</th>
<th>B2</th>
</tr>
</thead>
<tbody>
<tr>
<td>B2,1</td>
<td>B2,2</td>
</tr>
</tbody>
</table>

  \[ \text{mm\_load1\_pd}: \text{SSE instruction that loads a double word and stores it in the high and low double words of the XMM register (duplicates value in both halves of XMM)} \]
Example: 2 x 2 Matrix Multiply

- Second iteration intermediate result

\[
\begin{array}{c|c}
C_{1,1} & C_{2,1} \\
\hline
C_1 & A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\
C_2 & A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\
C_{1,2} & A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\
C_{2,2} & A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \\
\end{array}
\]

\[c1 = \_mm_add_pd(c1, _mm_mul_pd(a,b1));\]
\[c2 = \_mm_add_pd(c2, _mm_mul_pd(a,b2));\]
SSE instructions first do parallel multiplies and then parallel adds in XMM registers.

- \( I = 2 \)

\[
\begin{array}{c|c}
A & \_mm_load_pd: Stored in memory in Column order \\
\hline
A_{1,2} & A_{2,2} \\
\end{array}
\]

\[
\begin{array}{c|c}
B_1 & B_{2,1} \\
B_2 & B_{2,2} \\
\end{array}
\]

\_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).

Live Example: 2 x 2 Matrix Multiply

Definition of Matrix Multiply:

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

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

\[
\begin{bmatrix}
1 & 0 \\
0 & 1 \\
\end{bmatrix}
\begin{bmatrix}
1 & 3 \\
2 & 4 \\
\end{bmatrix}
\begin{bmatrix}
C_{1,1} = 1 \times 1 + 0 \times 2 = 1 \\
C_{1,2} = 1 \times 3 + 0 \times 4 = 3 \\
C_{2,1} = 0 \times 1 + 1 \times 2 = 2 \\
C_{2,2} = 0 \times 3 + 1 \times 4 = 4 \\
\end{bmatrix}
\]
Example: 2 x 2 Matrix Multiply
(Part 1 of 2)

```c
#include <stdio.h>
#include <emmintrin.h>

// Initialize A, B, C for example
/* 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
0 0 */
C[0] = 0.0; C[1] = 0.0; C[2] = 0.0; C[3] = 0.0;

int main(void) {
    // allocate A, B, C aligned on 16-byte boundaries
    double B[4] __attribute__((aligned(16)));
    double C[4] __attribute__((aligned(16)));
    int lda = 2;
    int i = 0;
    // declare several 128-bit vector variables
    __m128d c1,c2,a,b1,b2;

    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_21]
           i = 1: [b_12 / b_22] */
        b1 = _mm_load1_pd(B+i*lda);
        /* b2 = 
           i = 0: [b_11 / b_12]
           i = 1: [b_22 / b_22] */
        b2 = _mm_load1_pd(B+i*lda);
        a = _mm_add_pd(a, _mm_mul_pd(b1, b2));
    }
    // store c1, c2 back into C for completion
    _mm_store_pd(C, c1);
    _mm_store_pd(C+lda, c2);
    // print C
    printf("%g,%g
%g,%g", C[0], C[1], C[2], C[3]);
    return 0;
}
```

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

```c
#include <stdio.h>
#include <emmintrin.h>

// Initialize A, B, C for example
/* 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
0 0 */
C[0] = 0.0; C[1] = 0.0; C[2] = 0.0; C[3] = 0.0;

int main(void) {
    // allocate A, B, C aligned on 16-byte boundaries
    double B[4] __attribute__((aligned(16)));
    double C[4] __attribute__((aligned(16)));
    int lda = 2;
    int i = 0;
    // declare several 128-bit vector variables
    __m128d c1,c2,a,b1,b2;

    for (i = 0; i < 2; i++) {
        /* a = 
           i = 0: [a_11 / a_11]
           i = 1: [a_12 / a_22] */
        a = _mm_load_pd(A+i*lda);
        /* b1 = 
           i = 0: [b_11 / b_11]
           i = 1: [b_21 / b_21] */
        b1 = _mm_load1_pd(B+i*lda);
        /* b2 = 
           i = 0: [b_12 / b_12]
           i = 1: [b_22 / b_22] */
        b2 = _mm_load1_pd(B+i*lda);
        c1 = _mm_add_pd(c1, _mm_mul_pd(a, b1));
        c2 = _mm_add_pd(c2, _mm_mul_pd(a, b2));
    }
    // store c1, c2 back into C for completion
    _mm_store_pd(C, c1);
    _mm_store_pd(C+lda, c2);
    // print C
    printf("%g,%g
%g,%g", C[0], C[1], C[2], C[3]);
    return 0;
}
```
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 × b)
  - Multiply-accumulate: reduce round-off error (s := s + a × b)
  - Conditional copy: to avoid some branches (e.g., in if-then-else)
Administrivia

Midterm Review

Topics we’ve covered
New-School Machine Structures (It’s a bit more complicated!)

- **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

Great Ideas in Computer Architecture

1. **Design for Moore’s Law**
2. **Abstraction to Simplify Design**
3. **Make the Common Case Fast**
4. **Dependability via Redundancy**
5. **Memory Hierarchy**
6. **Performance via Parallelism/Pipelining/Prediction**
Moore’s Law

Predicts: 2X Transistors / chip every 2 years

Curve shows ‘Moore’s Law’: transistor count doubling every two years

Gordon Moore, Intel Cofounder
B.S. Cal 1950
Cal Alumni of Year 1997

Abstraction via Layers of Representation

High Level Language Program (e.g., C)  
Compiler

Assembly Language Program (e.g., MIPS)  
Assembler

Machine Language Program (MIPS)  
Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)

Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)

anything can be represented as a number, i.e., data or instructions

0000 1001 1100 0110 1010 1111 0101 1000  
1010 1111 0101 1000 0000 1001 1100 0110  
1100 0110 1010 1111 0101 1000 0000 1001  
0101 1000 0000 1001 1100 0110 1010 1111
Make the Common Case Fast

- In making a design tradeoff, favor the common over the infrequent case
- Don’t spend time optimizing code that is run infrequently
- Choose your performance metric and use measurement to determine the common case

Dependability via Redundancy

- Redundancy so that a failing piece doesn’t make the whole system fail

Increasing transistor density reduces the cost of redundancy
Memory Hierarchy

Fast, Expensive, but Small

Parallelism/Pipelining/Prediction

Cheap, Large, but Small
The XBOX One vs. Real Performance

Warehouse-Scale Computers

- Power Usage Effectiveness
- Request-Level Parallelism
- MapReduce
- Handling failures
- Costs of WSC
C Language and Compilation

- C Types, including Structs, Consts, Enums
- Arrays and strings
- C Pointers
- C functions and parameter passing

MIPS Instruction Set

- ALU operations
- Loads/Stores
- Branches/Jumps
- Registers
- Memory
- Function calling conventions
- Stack
Everything is a Number

- Binary
- Signed versus Unsigned
- One’s Complement/Two’s Complement
- Floating-Point numbers
- Character Strings

- Instruction Encoding
Caches

- Spatial/Temporal Locality
- Instruction versus Data
- Block size, capacity
- Direct-Mapped cache
- 3 C’s
- Cache-aware performance programming

Parallelism

- SIMD/MIMD/MISD/SISD
- Amdahl’s Law
- Strong vs. weak scaling
- Data-Parallel execution
And in Conclusion, ...

- Intel SSE SIMD Instructions
  - Exploit data-level parallelism in loops
  - One instruction fetch that operates on multiple operands simultaneously
  - 128-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