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

- Exploiting Data-Level Parallelism with SIMD instructions

Intel SIMD Instructions

- Fetch one instruction, do the work of multiple instructions

<table>
<thead>
<tr>
<th>Source 1</th>
<th>X3</th>
<th>X2</th>
<th>X1</th>
<th>X0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source 2</td>
<td>V3</td>
<td>V2</td>
<td>V1</td>
<td>V0</td>
</tr>
<tr>
<td>Destination</td>
<td>X3 OP Y3</td>
<td>X2 OP Y2</td>
<td>X1 OP Y1</td>
<td>X0 OP Y0</td>
</tr>
</tbody>
</table>

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

XMM 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)

Fundamental 128-Bit Packed SPFP Data Types:

<table>
<thead>
<tr>
<th>Paired Bytes</th>
<th>Paired Doublewords</th>
<th>Paired Quadwords</th>
</tr>
</thead>
<tbody>
<tr>
<td>127 126 125 96 95 80 79 64 63 48 47 32 31 16 15</td>
<td>16 / 128 bits</td>
<td>8 / 128 bits</td>
</tr>
<tr>
<td>127 126 125 96 95 80 79 64 63 48 47 32 31 16 15</td>
<td>4 / 128 bits</td>
<td>2 / 128 bits</td>
</tr>
</tbody>
</table>
First SIMD Extensions:
MIT Lincoln Labs TX-2, 1957

SSE/SSE2 Floating Point Instructions

Example: Add Two Single-Precision Floating-Point Vectors

Computation to be performed:
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;

SSE Instruction Sequence:
mov a ps: move from mem to XMM register, memory aligned, packed single precision
add ps: add from mem to XMM register, packed single precision

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

- Vector data type: _m128d
- Load and store operations:
  _mm_load_pd
  _mm_store_pd
- Load and broadcast across vector
  _mm_load1_pd
- Arithmetic:
  _mm_add_pd
  _mm_mul_pd

Packed and Scalar Double-Precision Floating-Point Operations

Example:

```
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;
```
Example: 2 x 2 Matrix Multiply

Definition of Matrix Multiply:

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

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

First iteration intermediate result

\[ I = 1 \]

\[
\begin{bmatrix}
A & B \\
C & D
\end{bmatrix}
\]

Example: 2 x 2 Matrix Multiply

• Initialization

\[
\begin{bmatrix}
C_{1} & 0 & 0 \\
C_{2} & 0 & 0
\end{bmatrix}
\]

Example: 2 x 2 Matrix Multiply

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

\[
\begin{bmatrix}
C_{1} & C_{12} \\
C_{2} & C_{22}
\end{bmatrix}
\]

Stored in memory in Column-major order

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

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

\[
\begin{bmatrix}
C_{1} & C_{2} \\
C_{3} & C_{4}
\end{bmatrix}
\]

I = 1

\[
\begin{bmatrix}
A & B \\
C & D
\end{bmatrix}
\]

\[
\begin{bmatrix}
C & C_{2} \\
C_{2} & C_{4}
\end{bmatrix}
\]

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

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

\[
\begin{bmatrix}
C_{1} & C_{2} \\
C_{3} & C_{4}
\end{bmatrix}
\]

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

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

\[
\begin{bmatrix}
C_{1} & C_{2} \\
C_{3} & C_{4}
\end{bmatrix}
\]

Example: 2 x 2 Matrix Multiply

• First iteration intermediate result

\[
\begin{bmatrix}
C & C_{2} \\
C_{2} & C_{4}
\end{bmatrix}
\]

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

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

\[
\begin{bmatrix}
C_{1} & C_{2} \\
C_{3} & C_{4}
\end{bmatrix}
\]

Example: 2 x 2 Matrix Multiply

• First iteration intermediate result

\[
\begin{bmatrix}
C & C_{2} \\
C_{2} & C_{4}
\end{bmatrix}
\]

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

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

\[
\begin{bmatrix}
C_{1} & C_{2} \\
C_{3} & C_{4}
\end{bmatrix}
\]

Example: 2 x 2 Matrix Multiply

• First iteration intermediate result

\[
\begin{bmatrix}
C_{1} & C_{12} \\
C_{2} & C_{22}
\end{bmatrix}
\]

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

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

\[
\begin{bmatrix}
C_{1} & C_{2} \\
C_{3} & C_{4}
\end{bmatrix}
\]

Example: 2 x 2 Matrix Multiply

• First iteration intermediate result

\[
\begin{bmatrix}
C_{1} & C_{12} \\
C_{2} & C_{22}
\end{bmatrix}
\]

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

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

\[
\begin{bmatrix}
C_{1} & C_{2} \\
C_{3} & C_{4}
\end{bmatrix}
\]
Example: 2 x 2 Matrix Multiply

- Second iteration intermediate result
- $A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$
- $B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$
- $C = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$

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

```c
#include <stdio.h>

float A[4], B[4], C[4];

int main(void)
{
    A[0] = 0.0;
    A[1] = 0.0;
    A[2] = 0.0;
    A[3] = 0.0;
    B[0] = 1.0;
    B[1] = 2.0;
    B[2] = 3.0;
    B[3] = 4.0;
    C[0] = 0.0;
    C[1] = 0.0;
    C[2] = 0.0;
    C[3] = 0.0;

    for (c2 = _mm_load_pd(C+1*8); c1 = _mm_load_pd(C+0*8);
        c1 + a_11*b_12 |
        c2 + a_22*b_22)
        for (c2 += 2)
            for (c1 += 1)
                C += 8; (C[0] += 4.0; C[1] += 4.0; C[2] += 4.0; C[3] += 4.0);

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

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

```c
#include <stdio.h>

float A[4], B[4], C[4];

int main(void)
{
    A[0] = 0.0;
    A[1] = 0.0;
    A[2] = 0.0;
    A[3] = 0.0;
    B[0] = 1.0;
    B[1] = 2.0;
    B[2] = 3.0;
    B[3] = 4.0;
    C[0] = 0.0;
    C[1] = 0.0;
    C[2] = 0.0;
    C[3] = 0.0;

    /*
        c1 = _mm_add_pd(c1, c2);
        c2 = _mm_add_pd(c2, c1);
        C += 8; (C[0] += 4.0; C[1] += 4.0; C[2] += 4.0; C[3] += 4.0);
        
        printf("%g,%g
%g,%g", C[0], C[2], C[1], C[3]);
        return 0;
    */
}
```

Inner loop from gcc –O -S

- L2: movapdp (%rax,%rsi), %xmm0
- movdqu (%rax), %xmm0
- mulps %xmm0, %xmm3
- addps %xmm0, %xmm3
- store aligned m3 into C[k+1]
- store aligned m2 into C[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: replace two instructions with one ($c := c + a \times b$)
  - Multiply-accumulate: reduce round-off error ($s := s + a \times b$)
  - Conditional copy: to avoid some branches (e.g., if-then-else)
10/5/12

Administrivia
• Lab #6, Project #2b posted
• Midterm Tuesday Oct 9, 8PM:
  – Two rooms: 1 Pimentel and 2050 LSB
  – Check your room assignment!
  – Covers everything through lecture Wednesday 10/3
  – Closed book, can bring one sheet notes, both sides
  – Copy of Green card will be supplied
  – No phones, calculators, …; just bring pencils & eraser

Midterm Room Assignment by Login
1 Pimentel = logins ab – mk
2050 VLSB = logins mm - xm

Midterm Review
Topics we’ve covered

New-School Machine Structures
(It’s a bit more complicated!)

6 Great Ideas in Computer Architecture
1. Layers of Representation/Interpretation
2. Moore’s Law
3. Principle of Locality/Memory Hierarchy
4. Parallelism
5. Performance Measurement & Improvement
6. Dependability via Redundancy

Great Idea #1: Levels of Representation/Interpretation
Great Idea #2: Moore’s Law

Predicts: 2X Transistors / chip every 2 years

Great Idea #3: Principle of Locality/ Memory Hierarchy

Great Idea #4: Parallelism

Great Idea #5: Performance Measurement and Improvement

- Match application to underlying hardware to exploit:
  - Locality
  - Parallelism
  - Special hardware features, like specialized instructions (e.g., matrix manipulation)
- Latency
  - How long to set the problem up
  - How much faster does it execute once it gets going
  - It is all about time to finish
- Make common case fast!

Great Idea #6: Dependability via Redundancy

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

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

Components of a Computer

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
...in Conclusion

• Intel SSE SIMD Instructions
  – 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