CS 61C: Great Ideas in Computer Architecture

The Flynn Taxonomy,
Data Level Parallelism

Instructor: Justin Hsia
Review of Last Lecture

• Performance programming
  – When possible, loops through arrays in chunks that “fit nicely” in your cache
  – Cache blocking will improve speed of Matrix Multiply with appropriately-sized blocks

• Processors have hit the power wall, the only option is to go parallel
  – $P = C \times V^2 \times f$
Question: Which statement is TRUE about cache blocking for matrix multiply?

(A) The same code will have the same performance whether the matrix is row or column major
(B) All choices of block size will produce a similar amount of speedup vs. naïve algorithm
(C) Cache blocking helps with both read and write hits in the cache
(D) For the product of two matrices, we need our cache to fit at least two matrix blocks at a time
Great Idea #4: Parallelism

- **Software**
  - Parallel Requests
    Assigned to computer
e.g. search “Garcia”
  - 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

- **Hardware**
  - Warehouse Scale Computer
  - Leverage Parallelism & Achieve High Performance
  - Core
  - Memory
  - Input/Output
  - Instruction Unit(s)
  - Functional Unit(s)
    - $A_0 + B_0$, $A_1 + B_1$, $A_2 + B_2$, $A_3 + B_3$
  - Cache Memory
  - Logic Gates

**Smart Phone**

We are here
Agenda

• Flynn’s Taxonomy
• Administrivia
• Data Level Parallelism and SIMD
• Intel SSE Intrinsics
• Loop Unrolling
Hardware vs. Software Parallelism

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Software</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Serial</td>
<td>Concurrent</td>
</tr>
<tr>
<td>Serial</td>
<td>Matrix Multiply written in MatLab running on an Intel Pentium 4</td>
<td>Windows Vista Operating System running on an Intel Pentium 4</td>
</tr>
<tr>
<td>Parallel</td>
<td>Matrix Multiply written in MATLAB running on an Intel Xeon e5345 (Clovertown)</td>
<td>Windows Vista Operating System running on an Intel Xeon e5345 (Clovertown)</td>
</tr>
</tbody>
</table>

- Choice of hardware and software parallelism are independent
  - Concurrent software can also run on serial hardware
  - Sequential software can also run on parallel hardware
- *Flynn’s Taxonomy* is for parallel hardware
Flynn’s Taxonomy

<table>
<thead>
<tr>
<th>Instruction Streams</th>
<th>Data Streams</th>
<th>Single</th>
<th>Multiple</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>SISD: Intel Pentium 4</td>
<td>SIMD: SSE instructions of x86</td>
<td></td>
</tr>
<tr>
<td>Multiple</td>
<td>MISD: No examples today</td>
<td>MIMD: Intel Xeon e5345 (Clovertown)</td>
<td></td>
</tr>
</tbody>
</table>

- SIMD and MIMD most commonly encountered today
- Most common parallel programming style:
  - Single program that runs on all processors of an MIMD
  - Cross-processor execution coordination through conditional expressions
- SIMD: specialized function units (hardware), for handling lock-step calculations involving arrays
  - Scientific computing, signal processing, audio/video
Single Instruction/Single Data Stream

- Sequential computer that exploits no parallelism in either the instruction or data streams
- Examples of SISD architecture are traditional uniprocessor machines
Multiple Instruction/Single Data Stream

- Exploits multiple instruction streams against a single data stream for data operations that can be naturally parallelized (e.g. certain kinds of array processors)

- MISD no longer commonly encountered, mainly of historical interest only
Single Instruction/Multiple Data Stream

• Computer that applies a single instruction stream to multiple data streams for operations that may be naturally parallelized (e.g. SIMD instruction extensions or Graphics Processing Unit)
Multiple Instruction/Multiple Data Stream

- Multiple autonomous processors simultaneously executing different instructions on different data
- MIMD architectures include multicore and Warehouse Scale Computers

7/18/2013
Agenda

- Flynn’s Taxonomy
- Administrivia
- Data Level Parallelism and SIMD
- Intel SSE Intrinsics
- Loop Unrolling
Administrivia

- HW4 due Sunday
- Midterm
  - Friday 7/19, 9am-12pm, 1 Pimentel
  - Double-sided handwritten sheet
  - MIPS Green Sheet provided
  - No calculators
- Justin’s OH tonight 5-7pm in 200 SDH
Agenda

• Flynn’s Taxonomy
• Administrivia
• **Data Level Parallelism and SIMD**
• Intel SSE Intrinsics
• Loop Unrolling
SIMD Architectures

• **Data-Level Parallelism (DLP):** Executing one operation on multiple data streams

• **Example:** Multiplying a coefficient vector by a data vector (e.g. in filtering)

\[ y[i] := c[i] \times x[i], \quad 0 \leq i < n \]

• 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
    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
    write the result from the register to memory
}
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>

- Intel processors are **CISC (complicated instrs)**
- SSE-2+ supports wider data types to allow 16 × 8-bit and 8 × 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)
XMM Registers

- Architecture extended with eight 128-bit data registers
  - 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

<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>
## 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/U}{SS/PS/SD/PD} xmm, mem/xmm</td>
<td>ADD{SS/PS/SD/PD} xmm, mem/xmm</td>
<td>CMP{SS/PS/SD/PD}</td>
</tr>
<tr>
<td></td>
<td>SUB{SS/PS/SD/PD} xmm, mem/xmm</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></td>
<td>DIV{SS/PS/SD/PD} xmm, mem/xmm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SQRT{SS/PS/SD/PD} mem/xmm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAX{SS/PS/SD/PD} mem/xmm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MIN{SS/PS/SD/PD} mem/xmm</td>
<td></td>
</tr>
</tbody>
</table>

{SS} Scalar Single precision FP: **1** 32-bit operand in a 128-bit register

{PS} Packed Single precision FP: **4** 32-bit operands in a 128-bit register

{SD} Scalar Double precision FP: **1** 64-bit operand in a 128-bit register

{PD} Packed Double precision FP, or **2** 64-bit operands in a 128-bit register
## 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/U}{SS/PS/SD/PD} xmm, mem/xmm</td>
<td>ADD{SS/PS/SD/PD} xmm, mem/xmm</td>
<td>CMP{SS/PS/SD/PD}</td>
</tr>
<tr>
<td></td>
<td>SUB{SS/PS/SD/PD} xmm, mem/xmm</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></td>
<td>DIV{SS/PS/SD/PD} xmm, mem/xmm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SQRT{SS/PS/SD/PD} mem/xmm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAX {SS/PS/SD/PD} mem/xmm</td>
<td></td>
</tr>
<tr>
<td></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
- `{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
Example: Add Single Precision FP Vectors

Computation to be performed:

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

SSE Instruction Sequence:

\[
\begin{align*}
\text{movaps} & \quad \text{address-of-v1, } %\text{xmm0}
\quad \text{move from mem to XMM register,} \\
& \quad \quad \text{memory aligned, packed single precision} \\
\text{addps} & \quad \text{address-of-v2, } %\text{xmm0}
\quad \text{add from mem to XMM register,} \\
& \quad \quad \text{packed single precision} \\
\text{movaps} & \quad %\text{xmm0, address-of-vec_res}
\quad \text{move from XMM register to mem,} \\
& \quad \quad \text{memory aligned, packed single precision}
\end{align*}
\]
Packed and Scalar Double-Precision Floating-Point Operations

Packed Double (PD)

Scalar Double (SD)
Example: Image Converter (1/5)

• 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

• Bitmap image consists of 8-bit monochrome pixels
  – By packing these pixel values in a 128-bit register, we can operate on 128/8 = 16 values at a time
  – Significant performance boost
Example: Image Converter (2/5)

• FMADDPS – Multiply and add packed single precision floating point instruction

• One of the typical operations computed in transformations (e.g. DFT or FFT)

\[
P = \sum_{n=1}^{N} f(n) \times x(n)
\]
Example: Image Converter (3/5)

- FP numbers f(n) and x(n) in src1 and src2; p in dest;
- C implementation for N = 4 (128 bits):
  
  ```
  for (int i = 0; i < 4; i++)
      p = p + src1[i] * src2[i];
  ```

1) Regular x86 instructions for the inner loop:

   - `fmul` [...]
   - `faddp` [...]

   - Instructions executed: $4 \times 2 = 8$ (x86)
Example: Image Converter (4/5)

- FP numbers f(n) and x(n) in src1 and src2; p in dest;
- C implementation for N = 4 (128 bits):
  
  for (int i = 0; i < 4; i++)
  
      p = p + src1[i] * src2[i];

2) 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

  — Instructions executed: 2 (SSE2)
Example: Image Converter (5/5)

- FP 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];
  ```

3) SSE5 accomplishes the same in **one** instruction:

  ```c
  fmaddps %xmm0, %xmm1, %xmm2, %xmm0
  // xmm2 * xmm1 + xmm0 -> xmm0
  // multiply xmm1 x xmm2 paired single,
  // then add product paired single to sum in xmm0
  ```
Agenda

• Flynn’s Taxonomy
• Administrivia
• Data Level Parallelism and SIMD
  • Intel SSE Intrinsics
• Loop Unrolling
Intel SSE Intrinsics

- Intrinsics are C functions and procedures that translate to assembly language, including SSE instructions
  - With intrinsics, can program using these instructions indirectly
  - One-to-one correspondence between intrinsics and SSE instructions
Sample of 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 × 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 × 2 Matrix Multiply

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

<table>
<thead>
<tr>
<th>c₁</th>
<th>C₁,₁</th>
<th>C₂,₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>c₂</td>
<td>C₁,₂</td>
<td>C₂,₂</td>
</tr>
<tr>
<td>a</td>
<td>A₁,i</td>
<td>A₂,i</td>
</tr>
<tr>
<td>b₁</td>
<td>Bₗ₁,i,₁</td>
<td>Bₗ₂,i,₁</td>
</tr>
<tr>
<td>b₂</td>
<td>Bₗ₁,i,₂</td>
<td>Bₗ₂,i,₂</td>
</tr>
</tbody>
</table>

Memory is column major
Example: $2 \times 2$ Matrix Multiply

• Initialization

\[
\begin{array}{cc}
  c_1 & 0 & 0 \\
  c_2 & 0 & 0 \\
\end{array}
\]

• $i = 1$

\[
\begin{array}{cc}
  a & A_{1,1} & A_{2,1} \\
  b_1 & B_{1,1} & B_{1,1} \\
  b_2 & B_{1,2} & B_{1,2} \\
\end{array}
\]

_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 × 2 Matrix Multiply

• First iteration intermediate result

<table>
<thead>
<tr>
<th></th>
<th>c₁</th>
<th>c₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b₁</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b₂</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ c₁ = _\text{mm\_add\_pd}(c₁, _\text{mm\_mul\_pd}(a, b₁)); \]
\[ c₂ = _\text{mm\_add\_pd}(c₂, _\text{mm\_mul\_pd}(a, b₂)); \]

• i = 1

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

\_\text{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 × 2 Matrix Multiply

• First iteration intermediate result

<table>
<thead>
<tr>
<th></th>
<th>0+A₁₁B₁₁</th>
<th>0+A₂₁B₁₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>c₁</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c₂</td>
<td>0+A₁₁B₁₂</td>
<td>0+A₂₁B₁₂</td>
</tr>
</tbody>
</table>

\[
c₁ = _\text{mm\_add\_pd}(c₁, _\text{mm\_mul\_pd}(a, b₁));
\]
\[
c₂ = _\text{mm\_add\_pd}(c₂, _\text{mm\_mul\_pd}(a, b₂));
\]

• \(i = 2\)

<table>
<thead>
<tr>
<th></th>
<th>(A₁₂)</th>
<th>(A₂₂)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>(B₂₁)</th>
<th>(B₂₁)</th>
</tr>
</thead>
<tbody>
<tr>
<td>b₁</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b₂</td>
<td>(B₂₂)</td>
<td>(B₂₂)</td>
</tr>
</tbody>
</table>

\[\text{\_mm\_load1\_pd}:\text{ SSE instruction that loads a double word and stores it in the high and low double words of the XMM register}\]
Example: $2 \times 2$ Matrix Multiply

- Second iteration intermediate result

\[
\begin{array}{c|c|c|c}
& c_1 & c_2 & c_3 \\
\hline
1 & A_{1,1}B_{1,1} + A_{1,2}B_{2,1} & A_{2,1}B_{1,1} + A_{2,2}B_{2,1} & \\
2 & A_{1,1}B_{1,2} + A_{1,2}B_{2,2} & A_{2,1}B_{1,2} + A_{2,2}B_{2,2} & \\
\end{array}
\]

\[
c_1 = \text{_mm_add_pd}(c_1, \text{_mm_mul_pd}(a,b_1));
\]
\[
c_2 = \text{_mm_add_pd}(c_2, \text{_mm_mul_pd}(a,b_2));
\]

- \(i = 2\)

\[
\begin{array}{c|c|c|c}
& a & b_1 & b_2 \\
\hline
1 & A_{1,2} & B_{2,1} & B_{2,1} \\
2 & A_{2,2} & B_{2,2} & B_{2,2} \\
\end{array}
\]

_\text{mm_load_pd}: Stored in memory in Column order

_\text{mm_load1_pd}: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register.
#include <stdio.h>
//# header file for SSE3 compiler intrinsics
#include <nmmintrin.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) {
    // 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 a couple 128-bit vector variables
    __m128d c1,c2,a,b1,b2;
    /* 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;
2 x 2 Matrix Multiply Code (2/2)

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

// 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[2], C[1], C[3]);
return 0;
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: two instructions in one (e.g. multiply by 5)
  – Multiply-add: two instructions in one (x := c + a \times b)
  – Multiply-accumulate: reduce round-off error (s := s + a \times b)
  – Conditional copy: avoid some branches (e.g. if-then-else)
Get To Know Your Instructor
Agenda

• Flynn’s Taxonomy
• Administrivia
• Data Level Parallelism and SIMD
• Intel SSE Intrinsics

• Loop Unrolling
Data Level Parallelism and SIMD

- SIMD wants adjacent values in memory that can be operated in parallel
- Usually specified in programs as loops
  
  ```cpp
  for (i=0; i<1000; i++)
    x[i] = x[i] + s;
  ```

- How can we reveal more data level parallelism than is available in a single iteration of a loop?
  - *Unroll the loop* and adjust iteration rate
Looping in MIPS

Assumptions:
- $s0 \rightarrow$ initial address (top of array)
- $s1 \rightarrow$ scalar value $s$
- $s2 \rightarrow$ termination address (end of array)

Loop:

```
lw $t0, 0($s0)  # add s to array element
addu $t0,$t0,$s1
sw $t0, 0($s0)  # store result
addiu $s0,$s0,4  # move to next element
bne $s0,$s2,Loop # repeat Loop if not done
```
Loop Unrolled

Loop:  

```
lw    $t0,0($s0)
addu  $t0,$t0,$s1
sw    $t0,0($s0)
lw    $t1,4($s0)
addu  $t1,$t1,$s1
sw    $t1,4($s0)
lw    $t2,8($s0)
addu  $t2,$t2,$s1
sw    $t2,8($s0)
lw    $t3,12($s0)
addu  $t3,$t3,$s1
sw    $t3,12($s0)
addiu $s0,$s0,16
bne   $s0,$s2,Loop
```

**NOTE:**

1. Using different registers eliminate stalls

2. Loop overhead encountered only once every 4 data iterations

3. This unrolling works if 
   
   \[ \text{loop\_limit \ mod \ 4} = 0 \]
Loop Unrolled Scheduled

Note: We just switched from integer instructions to single-precision FP instructions!

Loop: lwc1 $t0,0($s0)
lwc1 $t1,4($s0)
lwc1 $t2,8($s0)
lwc1 $t3,12($s0)
add.s $t0,$t0,$s1
add.s $t1,$t1,$s1
add.s $t2,$t2,$s1
add.s $t3,$t3,$s1
swc1 $t0,0($s0)
swc1 $t1,4($s0)
swc1 $t2,8($s0)
swc1 $t3,12($s0)
addiu $s0,$s0,16
bne $s0,$s2,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:

```c
for (i=0; i<1000; i++)
    x[i] = x[i] + s;
```

Loop Unroll

```c
for (i=0; i<1000; 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 this in C?
Generalizing Loop Unrolling

• Take a loop of \textbf{n iterations} and perform a \textbf{k-fold} unrolling of the body of the loop:
  – First run the loop with \( k \) copies of the body \( \text{floor}(n/k) \) times
  – To finish leftovers, then run the loop with 1 copy of the body \( n \mod k \) times

• (Will revisit loop unrolling again when get to pipelining later in semester)
Summary

• Flynn Taxonomy of Parallel Architectures
  – SIMD: Single Instruction Multiple Data
  – MIMD: Multiple Instruction Multiple Data
  – SISD: Single Instruction Single Data
  – MISD: Multiple Instruction Single Data (unused)

• Intel SSE SIMD Instructions
  – One instruction fetch that operates on multiple operands simultaneously
  – 128/64 bit XMM registers
  – Embed the SSE machine instructions directly into C programs through use of intrinsics

• Loop Unrolling: Access more of array in each iteration of a loop