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?

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

**Software**

- **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

**Hardware**

- **Warehouse Scale Computer**

  **Leverage Parallelism & Achieve High Performance**

- **Core**
  Instruction Unit(s)
  Functional Unit(s)
  Cache Memory

- **Memory**
  Input/Output

- **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>Sequential</th>
<th>Concurrent</th>
</tr>
</thead>
<tbody>
<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>
<td></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>
<td></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>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>SIMD: Intel Pentium 4</td>
</tr>
<tr>
<td>Multiple</td>
<td>MISD: No examples today</td>
</tr>
<tr>
<td>Single</td>
<td>SIMD: SSE instructions of x86</td>
</tr>
<tr>
<td>Multiple</td>
<td>MIMD: Intel Xeon e5345 (Clovertown)</td>
</tr>
</tbody>
</table>

- SIMD and MIMD most commonly encountered today
- Most common parallel processing programming style: Single Program Multiple Data (“SPMD”)
  - Single program that runs on all processors of an MIMD
  - Cross-processor execution coordination through conditional expressions (will see later in Thread Level Parallelism)
- SIMD: specialized function units (hardware), for handling lock-step calculations involving arrays
  - Scientific computing, signal processing, multimedia (audio/video processing)
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
Agenda

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

• HW3 due tonight
• Midterm
  – Friday 7/13, 9am-12pm, 245 Li Ka Shing
  – One-sided handwritten sheet; MIPS Green Sheet
  – No calculators
• Justin’s OH tonight 5-7pm in 200 SDH
• Old midterm problems tomorrow
  – Taking requests via Piazza
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)
  \[ y[i] := c[i] \times x[i], \ 0 \leq i < n \]

- **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 $16 \times 8$-bit and $8 \times 16$-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)
XMMM 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
### 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:

```
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:**

```
movaps  address-of-v1, %xmm0
       // v1.w | v1.z | v1.y | v1.x -> xmm0
addps  address-of-v2, %xmm0
       // v1.w+v2.w | v1.z+v2.z | v1.y+v2.y | v1.x+v2.x
       -> xmm0
movaps  %xmm0, address-of-vec_res
```

*move* from mem to XMM register, memory aligned, packed single precision

*add* from mem to XMM register, packed single precision

*move* from XMM register to mem, memory aligned, packed single precision
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 \( \frac{128}{8} = 16 \) 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):
  ```c
  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 * 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)
• 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 + \text{src1}[i] \times \text{src2}[i];
```

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

```assembly
fmaddps %xmm0, %xmm1, %xmm2, %xmm0
// \text{xmm2} \times \text{xmm1} + \text{xmm0} \rightarrow \text{xmm0}
// multiply \text{xmm1} \times \text{xmm2} paired single,
// then add product paired single to sum in \text{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: $2 \times 2$ 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>C_1</th>
<th>C_{1,1}</th>
<th>C_{2,1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_2</td>
<td>C_{1,2}</td>
<td>C_{2,2}</td>
</tr>
</tbody>
</table>

Memory is column major

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

<table>
<thead>
<tr>
<th>B_1</th>
<th>B_{i,1}</th>
<th>B_{i,1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>B_2</td>
<td>B_{i,2}</td>
<td>B_{i,2}</td>
</tr>
</tbody>
</table>
Example: $2 \times 2$ Matrix Multiply

- **Initialization**

  \[
  \begin{array}{cc}
    C_1 & 0 & 0 \\
    C_2 & 0 & 0 \\
  \end{array}
  \]

- **$i = 1$**

  \[
  \begin{array}{cc}
    A_1 & A_{1,1} & A_{2,1} \\
    B_{11} & B_{1,1} & B_{1,1} \\
    B_{12} & B_{1,2} & B_{1,2} \\
  \end{array}
  \]

  _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

\[
\begin{array}{c|c}
C_1 & 0 + A_{1,1}B_{1,1} & 0 + A_{2,1}B_{1,1} \\
\hline
C_2 & 0 + A_{1,1}B_{1,2} & 0 + A_{2,1}B_{1,2} \\
\end{array}
\]

\[
c_1 = \_\text{mm\_add\_pd}(c_1, \_\text{mm\_mul\_pd}(a, b_1));
\]
\[
c_2 = \_\text{mm\_add\_pd}(c_2, \_\text{mm\_mul\_pd}(a, b_2));
\]

• \( i = 1 \)

\[
\begin{array}{c|c}
A_1 & A_{1,1} & A_{2,1} \\
\hline
B_{11} & B_{1,1} & B_{1,1} \\
B_{12} & B_{1,2} & B_{1,2} \\
\end{array}
\]

\_\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
Example: $2 \times 2$ Matrix Multiply

- **First iteration intermediate result**

  \[ C_1 = 0 + A_{1,1}B_{1,1} \quad 0 + A_{2,1}B_{1,1} \]
  \[ C_2 = 0 + A_{1,1}B_{1,2} \quad 0 + A_{2,1}B_{1,2} \]

  \[ c_1 = \_mm\_add\_pd(c_1, \_mm\_mul\_pd(a, b_1)); \]
  \[ c_2 = \_mm\_add\_pd(c_2, \_mm\_mul\_pd(a, b_2)); \]

- **i = 2**

  - _mm_load_pd: Stored in memory in Column order
  
  \[ A_2 = A_{1,2} \quad A_{2,2} \]

  - _mm_load1_pd: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register
  
  \[ B_{11} = B_{2,1} \quad B_{1,1} \]
  \[ B_{12} = B_{2,2} \quad B_{2,2} \]
Example: 2 × 2 Matrix Multiply

- Second iteration intermediate result

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

\[
c_1 = \text{_mm_add_pd}(c_1, \text{_mm_mul_pd}(a, b1));
\]

\[
c_2 = \text{_mm_add_pd}(c_2, \text{_mm_mul_pd}(a, b2));
\]

- i = 2

\[
\begin{array}{ccc}
& A_{1,2} & A_{2,2} \\
A_2 & & \\
B_{11} & B_{2,1} & B_{2,1} \\
B_{12} & B_{2,2} & B_{2,2}
\end{array}
\]

\[
\text{_mm_load_pd: Stored in memory in Column order}
\]

\[
\text{_mm_load1_pd: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register}
\]
2 x 2 Matrix Multiply Code (1/2)

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

// header file for SSE3 compiler intrinsics

// 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;
    /* continued on next slide */
}

/* 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;
```

A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \text{(note column order!)}

B = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \quad \text{(note column order!)}

C = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad \text{(note column order!)}
// 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_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 =
    i = 0: [c_11 + a_11*b_11 | c_21 + a_21*b_11]
    i = 1: [c_11 + a_21*b_21 | c_21 + a_22*b_21]
    */
    c1 = _mm_add_pd(c1, _mm_mul_pd(a, b1));
    /* c2 =
    i = 0: [c_12 + a_11*b_12 | c_22 + a_21*b_12]
    i = 1: [c_12 + a_21*b_22 | c_22 + a_22*b_22]
    */
    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 * b)
  – Multiply-accumulate: reduce round-off error (s := s + a * b)
  – Conditional copy: avoid some branches (e.g. if-then-else)
Get To Know Your Staff

• Category: Food
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
  \[
  \text{for}(i=0; \ i<1000; \ i=i+1) \\
  \hspace{1em} x[i] = x[i] + s;
  \]
• How can we reveal more data level parallelism than is available in a single iteration of a loop?
  – \textit{Unroll the loop} and adjust iteration rate
Looping in MIPS

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

Loop:

\begin{align*}
\text{lw} & \quad \$t0, 0(\$s0) \\
\text{addu} & \quad \$t0, \$t0, \$s1 \quad \# \text{add } s \text{ to array element} \\
\text{sw} & \quad \$t0, 0(\$s0) \quad \# \text{store result} \\
\text{addiu} & \quad \$s0, \$s0, 4 \quad \# \text{move to next element} \\
\text{bne} & \quad \$s0, \$s2, \text{Loop} \quad \# \text{repeat Loop if not done}
\end{align*}
### Loop Unrolled

Loop: \( \text{lw} \quad $t0,0($s0) \)
\( \text{addu} \quad $t0,$t0,$s1 \)
\( \text{sw} \quad $t0,0($s0) \)
\( \text{lw} \quad $t1,4($s0) \)
\( \text{addu} \quad $t1,$t1,$s1 \)
\( \text{sw} \quad $t1,4($s0) \)
\( \text{lw} \quad $t2,8($s0) \)
\( \text{addu} \quad $t2,$t2,$s1 \)
\( \text{sw} \quad $t2,8($s0) \)
\( \text{lw} \quad $t3,12($s0) \)
\( \text{addu} \quad $t3,$t3,$s1 \)
\( \text{sw} \quad $t3,12($s0) \)
\( \text{addiu} \quad $s0,$s0,16 \)
\( \text{bne} \quad $s0,$s2,Loop \)

### NOTE:

1. Using different registers eliminate stalls

2. Loop overhead encountered only once every 4 data iterations

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

Loop: l.d $t0,0($s0)
l.d $t1,8($s0)
l.d $t2,16($s0)
l.d $t3,24($s0)
add.d $t0,$t0,$s1
add.d $t1,$t1,$s1
add.d $t2,$t2,$s1
add.d $t3,$t3,$s1
s.d $t0,0($s0)
s.d $t1,8($s0)
s.d $t2,16($s0)
s.d $t3,24($s0)
addiu $s0,$s0,32
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=i+1)
    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