CS 61C: Great Ideas in Computer Architecture

The Flynn Taxonomy, Intel SIMD Instructions

Instructor: Justin Hsia

Review of Last Lecture

- Amdahl’s Law limits benefits of parallelization
- Request Level Parallelism
  - Handle multiple requests in parallel (e.g. web search)
- MapReduce Data Level Parallelism
  - Framework to divide up data to be processed in parallel
  - Mapper outputs intermediate key-value pairs
  - Reducer “combines” intermediate values with same key

Great Idea #4: Parallelism

- 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

Agenda

- Flynn’s Taxonomy
- Administrivia
- Data Level Parallelism and SIMD
- Bonus: Loop Unrolling

Hardware vs. Software Parallelism

- 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

- 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 a 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
- Bonus: Loop Unrolling

**Administrivia**

- HW3 due Sunday
- Proj2 (MapReduce) to be released soon
  - Part 1 due 3/17
  - Part 2 due 3/24
  - Work in partners, preferably at least 1 knows Java
- Midterms graded
  - Collect after lecture today or from Lab TA next week
Agenda

- Flynn’s Taxonomy
- Administrivia
- Data Level Parallelism and SIMD
- Bonus: 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

**Example: SIMD Array Processing**

```plaintext
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 every 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
}
```

**“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)

**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 x 8-bit and 8 x 16-bit operands
• 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>(SS) Scalar single precision FP:</td>
<td>3 32-bit operand in a 128-bit register</td>
<td>(PS) Packed single precision FP:</td>
</tr>
<tr>
<td>(PD) Packed double precision FP:</td>
<td>2 64-bit operands in a 128-bit register</td>
<td></td>
</tr>
</tbody>
</table>

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

### SSE/SSE2 Floating Point Instructions

<table>
<thead>
<tr>
<th>Data Transfer</th>
<th>Arithmetic</th>
<th>Compare</th>
</tr>
</thead>
</table>
| movaps address-of-v1, %xmm0
// v1.x | v1.y | v1.z | v1.w -> %xmm0 |
| addps address-of-v2, %xmm0
// v1.x+v2.x | v1.y+v2.y | v1.z+v2.z | v1.w+v2.w |
| movaps %xmm0, address-of-vec_res |

### 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>XMM Registers</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>127 0</td>
<td>XMM7</td>
</tr>
<tr>
<td>126 4</td>
<td>XMM6</td>
</tr>
<tr>
<td>125 8</td>
<td>XMM5</td>
</tr>
<tr>
<td>124 12</td>
<td>XMM4</td>
</tr>
<tr>
<td>123 16</td>
<td>XMM3</td>
</tr>
<tr>
<td>122 20</td>
<td>XMM2</td>
</tr>
<tr>
<td>121 24</td>
<td>XMM1</td>
</tr>
<tr>
<td>120 28</td>
<td>XMM0</td>
</tr>
</tbody>
</table>

### Packed and Scalar Double-Precision Floating-Point Operations

<table>
<thead>
<tr>
<th>Packed Double (PD)</th>
<th>Scalar Double (SD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>x1</td>
</tr>
<tr>
<td>y1</td>
<td>y1</td>
</tr>
<tr>
<td>z1</td>
<td>z1</td>
</tr>
<tr>
<td>x1+y2</td>
<td>y1+x2</td>
</tr>
<tr>
<td>x1+y2+2.2</td>
<td>y1+x2+2.2</td>
</tr>
<tr>
<td>x1+y2+2.2+1.4</td>
<td>y1+x2+2.2+1.4</td>
</tr>
<tr>
<td>movaps %xmm0, address-of-vec_res</td>
<td>movaps %xmm0, address-of-vec_res</td>
</tr>
</tbody>
</table>
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):
  
  ```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):
  
  ```c
  for (int i = 0; i < 4; i++)
      p = p + src1[i] * src2[i];
  ```

2) SSE2 instructions for the inner loop:

  ```c
  // 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 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];
  ```

3) SSE5 accomplishes the same in one instruction:

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

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
### Agenda

- Flynn’s Taxonomy
- Administrivia
- Data Level Parallelism and SIMD
- Bonus: 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
  
  ```
  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 (beginning of array)
- $s1 \rightarrow$ scalar value s
- $s2 \rightarrow$ termination address (end of array)

**Loop:**

```mips
lw $t0,0($s0)
addu $t0,$t0,$s1    # add s to array element
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 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 loop_limit mod 4 = 0

**Loop Unrolled Scheduled**

```
Loop:  lw $t0,0($s0)
     lwc1 $t0,0($s0)
     lwc1 $t1,4($s0)
     lwc1 $t2,8($s0)
     lwc1 $t3,12($s0)
     add.s $t0,$t0,$s1    # add s to array element
     lwc1 $t0,0($s0)
     lwc1 $t1,4($s0)
     lwc1 $t2,8($s0)
     lwc1 $t3,12($s0)
     add.s $t0,$t0,$s1    # store result
     lwc1 $t0,0($s0)
     lwc1 $t1,4($s0)
     lwc1 $t2,8($s0)
     lwc1 $t3,12($s0)
     add.s $t0,$t0,$s1    # move to next element
     bne $s0,$s2,Loop    # repeat Loop if not done
```

**Note:** We just switched from integer instructions to single-precision FP instructions!
Loop Unrolling in C

• Instead of compiler doing loop unrolling, could do it yourself in C:

\[
\text{for}(i=0; i<1000; i++) \quad x[i] = x[i] + s; \\
\text{for}(i=0; i<1000; i=i+4) \quad \{ \\
\quad x[i] = x[i] + s; \\
\quad x[i+1] = x[i+1] + s; \\
\quad x[i+2] = x[i+2] + s; \\
\quad x[i+3] = x[i+3] + s; \}
\]

What is downside of doing this in C?

Generalizing Loop Unrolling

• Take a loop of \textit{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)