CS 61C: Great Ideas in Computer Architecture

The Flynn Taxonomy, Intel SIMD Instructions

Guest Lecturer: Alan Christopher

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

Neuromorphic Chips

Researchers at IBM and HRL laboratories are looking into building computer chips that attempt to mimic natural thought patterns in order to more effectively solve problems in AI.

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 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

Multiple Instruction/Multiple Data Stream

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

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)

Agenda

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

- Midterm next Wednesday
  - May have 1 double sided 8.5" x 11" sheet of hand written notes
  - May also bring an unedited copy of the MIPS green sheet.
    - We will provide a copy if you forget yours
- Proj2 (MapReduce) to be released soon
  - Part 1 due date pushed later
  - Work in partners, 1 of you must know Java

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

```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
}
```

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>Saturation 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
Intel Architecture SSE2+ 128-Bit SIMD Data Types

### 128-Bit Packed SIMD Data Types

<table>
<thead>
<tr>
<th>Packed 16 Bytes</th>
<th>/ 128 Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>127</td>
<td></td>
</tr>
<tr>
<td>122 121</td>
<td>96 95</td>
</tr>
<tr>
<td>80 79</td>
<td>64 63</td>
</tr>
<tr>
<td>48 47</td>
<td>32 31</td>
</tr>
<tr>
<td>16 15</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Packed 8 Words</th>
<th>/ 128 Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>127</td>
<td></td>
</tr>
<tr>
<td>122 121</td>
<td>96 95</td>
</tr>
<tr>
<td>80 79</td>
<td>64 63</td>
</tr>
<tr>
<td>48 47</td>
<td>32 31</td>
</tr>
<tr>
<td>16 15</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Packed Doublewords</th>
<th>/ 128 Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>127</td>
<td></td>
</tr>
<tr>
<td>96 95</td>
<td></td>
</tr>
<tr>
<td>64 63</td>
<td></td>
</tr>
<tr>
<td>32 31</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Packed Quadswords</th>
<th>/ 128 Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>127</td>
<td></td>
</tr>
<tr>
<td>96 95</td>
<td></td>
</tr>
<tr>
<td>64 63</td>
<td></td>
</tr>
<tr>
<td>32 31</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

- 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 (XMM0 – 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 Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMM7</td>
</tr>
<tr>
<td>XMM6</td>
</tr>
<tr>
<td>XMM5</td>
</tr>
<tr>
<td>XMM4</td>
</tr>
<tr>
<td>XMM3</td>
</tr>
<tr>
<td>XMM2</td>
</tr>
<tr>
<td>XMM1</td>
</tr>
<tr>
<td>XMM0</td>
</tr>
</tbody>
</table>

---

### SSE/SSE2 Floating Point Instructions

#### Data Transfer

<table>
<thead>
<tr>
<th>Data Transfer</th>
<th>Arithmetic</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (PS/PD)</td>
<td>xmm, mem, mem</td>
<td>move from mem to XMM register, memory aligned, packed single precision</td>
</tr>
<tr>
<td>MOV (PS/PD)</td>
<td>mem, xmm</td>
<td>move from mem to XMM register, memory aligned, packed single precision</td>
</tr>
<tr>
<td>ADD (PS/PD)</td>
<td>xmm, xmm</td>
<td>add from mem to XMM register, packed single precision</td>
</tr>
<tr>
<td>ADD (PS/PD)</td>
<td>mem, xmm</td>
<td>add from mem to XMM register, packed single precision</td>
</tr>
<tr>
<td>XOR (PS/PD)</td>
<td>xmm, xmm</td>
<td>xor from XMM register to mem, memory aligned, packed single precision</td>
</tr>
<tr>
<td>XOR (PS/PD)</td>
<td>mem, xmm</td>
<td>xor from XMM register to mem, memory aligned, packed single precision</td>
</tr>
<tr>
<td>MOVAPS (PS/PD)</td>
<td>mem, xmm</td>
<td>move from mem to XMM register, memory aligned, packed single precision</td>
</tr>
<tr>
<td>MOVAPS (PS/PD)</td>
<td>xmm, mem</td>
<td>move from mem to XMM register, memory aligned, packed single precision</td>
</tr>
</tbody>
</table>

#### SSE Instruction Sequence

Example: Add Single Precision FP Vectors

Computation to be performed:

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

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
movaps %xmm0, address-of-vec_res
```

---

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

#### Packed Double (PD)

<table>
<thead>
<tr>
<th>K1</th>
<th>K2</th>
</tr>
</thead>
<tbody>
<tr>
<td>PD</td>
<td>PD</td>
</tr>
</tbody>
</table>

#### Scalar Double (SD)

<table>
<thead>
<tr>
<th>K1</th>
<th>K2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD</td>
<td>SD</td>
</tr>
</tbody>
</table>

---

3/08/2014 Spring 2014 -- Lecture #19
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:
   ```
   //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:
   ```
   fmaddps %xmm0, %xmm1, %xmm2, %xmm0
   // xmm2 * xmm1 + xmm0 -> xmm0
   // multiply xmm1 x 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
You are responsible for the material contained on the following slides, though we may not have enough time to get to them in lecture. They have been prepared in a way that should be easily readable and the material will be touched upon in the following lecture.

**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
- 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:
- `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**
- `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 Scheduled**
- 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:
  for(i=0; i<1000; i++)
    x[i] = x[i] + s;

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 downsides of doing this in C?

Generalizing Loop Unrolling

• Take a loop of n iterations and perform a k-fold unrolling of the body of the loop:
  – First run the loop with k copies of the body 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)