Review: Parallelism - The Challenge

- Only path to performance is parallelism
  - Clock rates flat or declining
- Key challenge is to craft parallel programs that have high performance on multiprocessors as the number of processors increase – i.e., that scale.
- Can exploit multiple types of parallelism
  - Request Level Parallelism (RLP)
  - Thread Level Parallelism (TLP)
  - Data Level Parallelism (DLP)
  - Instruction Level Parallelism (ILP)
- **Project #2**: fastest matrix multiply code on 8 processor (8 cores) computers
  - Will use DLP and TLP (and cache blocking).

Agenda

- Kinds of Parallelism, The Flynn Taxonomy
- Administrivia
- Data Level Parallelism and SIMD
- Break
- Intel SSE Intrinsics
- Amdahl’s Law (if time)

Kinds of Parallelism: The Programming Viewpoint

- Job-level parallelism/process-level parallelism
  - Running independent programs on multiple processors simultaneously
  - **Example**?
- Parallel processing program
  - Single program that runs on multiple processors simultaneously
  - **Example**?

Kinds of Parallelism: Hardware vs. Software

<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 (Downtown)</td>
<td>Windows Vista Operating System running on an Intel Xeon e5345 (Downtown)</td>
<td></td>
</tr>
</tbody>
</table>

- Concurrent software can also run on serial hardware
- Sequential software can also run on parallel hardware
- Focus is on **parallel processing software**: sequential or concurrent software running on parallel hardware

Kinds of Parallelism: Single Instruction/Single Data Stream

- Single Instruction, Single Data stream (SISD)
  - Sequential computer that exploits no parallelism in either the instruction or data streams. Examples of SISD architecture are traditional uniprocessor machines
Kinds of Parallelism:

Multiple Instruction/Single Data Stream

- Multiple Instruction, Single Data streams (MISD)
  - Computer that exploits multiple instruction streams against a single data stream for data operations that can be naturally parallelized. For example, certain kinds of array processors.
  - No longer commonly encountered, mainly of historical interest only.

Single Instruction/Multiple Data Stream

- Single Instruction, Multiple Data streams (SIMD)
  - Computer that exploits multiple data streams against a single instruction stream to operations that may be naturally parallelized, e.g., SIMD instruction extensions or Graphics Processing Unit (GPU).

Multiple Instruction/Multiple Data Streams

- Multiple Instruction, Multiple Data streams (MIMD)
  - Multiple autonomous processors simultaneously executing different instructions on different data.
  - MIMD architectures include multicore and Warehouse Scale Computers.

Flynn Taxonomy

- In 2011, SIMD and MIMD most commonly encountered
- 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 (aka hw-level data parallelism): specialized function units, for handling lock-step calculations involving arrays
  - Scientific computing, signal processing, multimedia (audio/video processing)

Data-Level Parallelism (DLP)

- 2 kinds
  - Lots of data in memory that can be operated on in parallel (e.g., adding together 2 arrays)
  - Lots of data on many disks that can be operated on in parallel (e.g., searching for documents)
- Today and 2nd project do first type of DLP.
- Later, will look at DLP across many disks or servers.

SIMD Architectures

- Execute one operation on multiple data streams
  - Example to provide context:
    - Vector multiply
      \[ z[i] := x[i] \times y[i], \quad 0 \leq i < n \]
  - Sources of performance improvement:
    - One instruction is fetched & decoded for entire operation
    - Multiplications are independent, executed in parallel.
    - 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
}

Agenda

• Kinds of Parallelism, The Flynn Taxonomy
• Administrivia
• Data Level Parallelism and SIMD
• Break
• Intel SSE Intrinsics
• Amdahl’s Law (if time)

Administrivia

• Midterm
  – Friday 7/15, 9am-12pm, 2050 VLSB
  – How to study:
    • Studying in groups can help.
    • Take old exams for practice (link at top of main webpage)
      – I’ve linked to the Spring 2011 midterm as well.
    • Look at lectures, section notes, projects, hw, labs, etc.
    • There are still a few OH left before the test...
      – No calculators allowed.

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>

• 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

- 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: XMM 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

SSE/SSE2 Floating Point Instructions

Example: Add Two Single Precision FP Vectors

Example: Image Converter

- 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
  - E.g., bitmap image consists of 8 bit monochrome pixels
    - Pack these pixel values in a 128 bit register (8 bit * 16 pixels), can operate on 16 values at a time
    - Significant performance boost
Example: Image Converter

- `FMADDPS` – Multiply and add packed single precision floating point instruction
- One of the typical operations computed in transformations (e.g., DFT of FFT)

\[ P = \sum_{n=1}^{N} f(n) \times x(n) \]

Example: Image Converter

Floating point numbers \( f(n) \) and \( x(n) \) in src1 and src2; \( p \in \text{dest} \);

C implementation for \( N = 4 \) (128 bits):

```c
for (int i = 0; i < 4; i++)
    p += src1[i] * src2[i];
```

Regular x86 instructions for the inner loop:

- `fmul` ...
- `faddp` ...

Number regular x86 Fl. Pt. instructions executed: \( 4 \times 2 = 8 \)

Example: Image Converter

Floating point numbers \( f(n) \) and \( x(n) \) in src1 and src2; \( p \in \text{dest} \);

C implementation for \( N = 4 \) (128 bits):

```c
for (int i = 0; i < 4; i++)
    p += src1[i] * src2[i];
```

- SSE2 instructions for the inner loop:
  - `mulps %xmm1, %xmm2` // \( \text{ xmm2} \times \text{ xmm1} \)
  - `addps %xmm2, %xmm0` // \( \text{ xmm0} + \text{ xmm2} \rightarrow \text{ xmm0} \)

Number regular instructions executed: 2 SSE2 instructions vs. 8 x86

SSE5 instruction accomplishes same in one instruction:

```c
fmaddps %xmm0, %xmm1, %xmm2, %xmm0
```

Number regular instructions executed: 1 SSE5 instruction vs. 8 x86

Agenda

- Kinds of Parallelism, The Flynn Taxonomy
- Administrivia
- Data Level Parallelism and SIMD
- Break
- Intel SSE Intrinsics
- Amdahl’s Law (if time)

Agenda

- 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.
Example 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/multiply, packed double

Example: 2 x 2 Matrix Multiply

Definition of Matrix Multiply:

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

Using the XMM registers

– 64-bit/double precision/two doubles per XMM reg

\[ C_1 = \begin{bmatrix} c_{1,1} \\ c_{1,2} \end{bmatrix} \]
\[ C_2 = \begin{bmatrix} c_{2,1} \\ c_{2,2} \end{bmatrix} \]

\[ A = \begin{bmatrix} \text{stored in memory in column order} \\ a_{1,1} \\ a_{1,2} \\ a_{2,1} \\ a_{2,2} \end{bmatrix} \]
\[ B_1 = \begin{bmatrix} b_{1,1} \\ b_{1,2} \end{bmatrix} \]
\[ B_2 = \begin{bmatrix} b_{2,1} \\ b_{2,2} \end{bmatrix} \]

\[ C_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \]
\[ C_2 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \]

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

Example: 2 x 2 Matrix Multiply

• Initialization

\[ C_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \]
\[ C_2 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \]

• \( i = 1 \)

\[ A_1 = \begin{bmatrix} a_{1,1} \\ a_{1,2} \end{bmatrix} \]
\[ B_{1,1} = \begin{bmatrix} b_{1,1} \\ b_{1,2} \end{bmatrix} \]
\[ B_{1,2} = \begin{bmatrix} b_{1,1} \\ b_{1,2} \end{bmatrix} \]

\[ a_{1,1} \text{\_mm_load_pd: stored in memory in column order} \]

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

```c
#include <emmintrin.h>  // header file for SSE compiler intrinsics
#include <stdio.h>      // header file for SSE compiler

int main() {
    double B[4] __attribute__((aligned (16)));  // double precision floating point numbers
    double C[4] __attribute__((aligned (16)));  // allocate A,B,C aligned on 16-byte boundaries

    a = _mm_load_pd(A+0*lda);    // load A[0] into high double words of the XMM register
    b1 = _mm_load1_pd(B+i+0*lda);  // load B[0] into low double words of the XMM register
    b2 = _mm_load1_pd(B+i+1*lda);  // load B[1] into low double words of the XMM register

    c1 = _mm_add_pd(c1, _mm_mul_pd(a, b1));  // add: replace two instructions with one (x := c + a * b)
    c2 = _mm_add_pd(c2, _mm_mul_pd(a, b2));  // add: replace two instructions with one (x := c + a * b)

    c1 = _mm_add_pd(c1, _mm_mul_pd(a, b1));  // add: replace two instructions with one (x := c + a * b)
    c2 = _mm_add_pd(c2, _mm_mul_pd(a, b2));  // add: replace two instructions with one (x := c + a * b)

    return 0;
}
```

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

```c
// used aligned loads to set
for (i = 0; i < 2; i++) {  // loop over the rows of the matrices
    for (j = 0; j < 2; j++) {  // loop over the columns of the matrices
        c[i][j] = _mm_add_pd(c[i][j], _mm_mul_pd(a[i][j], b[i][j]));  // add: replace two instructions with one (x := c + a * b)
    }
    c[i][2] = _mm_add_pd(c[i][2], _mm_mul_pd(a[i][2], b[i][2]));  // add: replace two instructions with one (x := c + a * b)
    c[i][3] = _mm_add_pd(c[i][3], _mm_mul_pd(a[i][3], b[i][3]));  // add: replace two instructions with one (x := c + a * b)
}
```

### Performance-Driven ISA Extensions

- **Subword parallelism, used primarily for multimedia applications**
  - Intel MMX: multimedia extension
  - 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 (x := c + a * b)
  - **Multiply-accumulate**: reduce round-off error (s := s + a * b)
  - **Conditional copy**: to avoid some branches (e.g., if-then-else)
Big Idea: Amdahl’s (Heartbreaking) Law

- Speedup due to enhancement \( E \) is
  
  \[
  \text{Speedup w/} \ E = \frac{\text{Exec time w/o} \ E}{\text{Exec time w/} \ E}
  \]

- Suppose that enhancement \( E \) accelerates a fraction \( F \) (\( F < 1 \)) of the task by a factor \( S \) (\( S > 1 \)) and the remainder of the task is unaffected

\[
\text{Execution Time w/} \ E = \text{Execution Time w/o} \ E \times \left[ (1-F) + \frac{F}{S} \right]
\]

\[
\text{Speedup w/} \ E = \frac{1}{(1-F) + \frac{F}{S}}
\]

Example: the execution time of half of the program can be accelerated by a factor of 2. What is the program speed-up overall?

\[
\frac{1}{0.5 + 0.25} = 1.33
\]

Example #1: Amdahl’s Law

- Consider an enhancement which runs 20 times faster but which is only usable 25% of the time
  
  \[
  \text{Speedup w/} \ E = \frac{1}{0.75 + 0.25/20} = 1.31
  \]

- What if its usable only 15% of the time?
  
  \[
  \text{Speedup w/} \ E = 1/(0.85 + 0.15/20) = 1.17
  \]

- Amdahl’s Law tells us that to achieve linear speedup with 100 processors, none of the original computation can be scalar!

- To get a speedup of 90 from 100 processors, the percentage of the original program that could be scalar would have to be 0.1% or less

\[
\text{Speedup w/} \ E = 1/(0.001 + 0.999/100) = 90.99
\]
Example #2: Amdahl’s Law

Speedup w/ E = 1 / [(1-F) + F/S]

- Consider summing 10 scalar variables and two 10 by 10 matrices (matrix sum) on 10 processors
  Speedup w/ E = 1 / [(0.91 + 0.09/10)] = 1 / 0.1009 = 10.0

- What if there are 100 processors?
  Speedup w/ E = 1 / [(0.91 + 0.09/100)] = 1 / 0.10009 = 9.9

- What if the matrices are 100 by 100 (or 10,010 adds in total) on 10 processors?
  Speedup w/ E = 1 / [(0.001 + 0.999/10)] = 1 / 0.1009 = 9.9

- What if there are 100 processors?
  Speedup w/ E = 1 / [(0.001 + 0.999/100)] = 1 / 0.01099 = 91

Strong and Weak Scaling

- To get good speedup on a multiprocessor while keeping the problem size fixed is harder than getting good speedup by increasing the size of the problem.
  - **Strong scaling:** when speedup can be achieved on a parallel processor without increasing the size of the problem.
  - **Weak scaling:** when speedup is achieved on a parallel processor by increasing the size of the problem proportionally to the increase in the number of processors.

- **Load balancing** is another important factor: every processor doing same amount of work
  - Just 1 unit with twice the load of others cuts speedup almost in half

"And in Conclusion.."

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

- Intel SSE SIMD Instructions
  - One instruction fetch that operates on multiple operands simultaneously
  - 128/64 bit XMM registers

- SSE Instructions in C
  - Embed the SSE machine instructions directly into C programs through use of intrinsics