CS 61C: Great Ideas in Computer Architecture (Machine Structures)

The Flynn Taxonomy, Data Level Parallelism

Instructor: Michael Greenbaum

7/14/2011
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></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>
</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>

- 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.
Kinds of Parallelism: 
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)
Kinds of Parallelism: 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 2\textsuperscript{rd} 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)

<table>
<thead>
<tr>
<th>Source 1</th>
<th>X3</th>
<th>X2</th>
<th>X1</th>
<th>X0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Source 2</td>
<td>Y3</td>
<td>Y2</td>
<td>Y1</td>
<td>Y0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Destination</td>
<td>X3 OP Y3</td>
<td>X2 OP Y2</td>
<td>X1 OP Y1</td>
<td>X0 OP Y0</td>
</tr>
</tbody>
</table>

7/14/2011 Summer 2011 -- Lecture #15
Example: SIMD Array Processing

for each \( f \) in array
\[
f = \sqrt{f}
\]

pseudocode

SID

SIMD
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.
Agenda

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

<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>

- Architecture extended with eight 128-bit data registers: XMM 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>

- xmm: one operand is a 128-bit SSE2 register
- mem/xmm: other operand is in memory or an SSE2 register
- {SS} Scalar Single precision FP: one 32-bit operand in a 128-bit register
- {PS} Packed Single precision FP: four 32-bit operands in a 128-bit register
- {SD} Scalar Double precision FP: one 64-bit operand in a 128-bit register
- {PD} Packed Double precision FP, or two 64-bit operands in a 128-bit 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 Two 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:

\[
\begin{align*}
\text{movaps address-of-v1, } &\%\text{xmm0} \\
&// v1.w \mid v1.z \mid v1.y \mid v1.x \rightarrow \text{xmm0} \\
\text{addps address-of-v2, } &\%\text{xmm0} \\
&// v1.w+v2.w \mid v1.z+v2.z \mid v1.y+v2.y \mid v1.x+v2.x \rightarrow \text{xmm0} \\
\text{movaps } &\%\text{xmm0, address-of-vec}_\text{res}
\end{align*}
\]

\textbf{mov a ps}: move from mem to XMM register, memory aligned, packed single precision

\textbf{add ps}: add from mem to XMM register, packed single precision

\textbf{mov a ps}: move from XMM register to mem, memory aligned, packed single precision
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 monochromed 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 dest;

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

$$
\text{for (int } i = 0; i < 4; i++) \\
\quad p = p + \text{src1}[i] \times \text{src2}[i];
$$

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

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

- 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
  ```

- Number regular instructions executed: 2 SSE2 instructions vs. 8 x86
- SSE5 instruction accomplishes same in one instruction:
  ```
  fmaddps %xmm0, %xmm1, %xmm2, %xmm0  // xmm2 * xmm1 + xmm0 -> xmm0
  // multiply xmm1 x xmm2 paired single,
  // then add product paired single to sum in 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

• Kinds of Parallelism, The Flynn Taxonomy
• Administrivia
• Data Level Parallelism and SIMD
• Break
• Intel SSE Intrinsics
• Amdahl’s Law (if time)
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/multiple, packed double
Example: 2 x 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 x 2 Matrix Multiply

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

\[
\begin{align*}
C_1 & : \begin{bmatrix} C_{1,1} & | & C_{2,1} \end{bmatrix} \\
C_2 & : \begin{bmatrix} C_{1,2} & | & C_{2,2} \end{bmatrix}
\end{align*}
\]

\[
\begin{align*}
A & : \begin{bmatrix} A_{1,i} & | & A_{2,i} \end{bmatrix} \\
B_1 & : \begin{bmatrix} B_{i,1} & | & B_{i,2} \end{bmatrix} \\
B_2 & : \begin{bmatrix} B_{i,1} & | & B_{i,2} \end{bmatrix}
\end{align*}
\]

Stored in memory in Column order
Example: 2 x 2 Matrix Multiply

• Initialization

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

_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 x 2 Matrix Multiply

• Initialization

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

• \( l = 1 \)

\[
\begin{array}{ccc}
A_1 & A_{1,1} & A_{2,1} \\
_\text{mm_load_pd}: & \text{Stored in memory in} & \text{Column order} \\
B_{11} & B_{1,1} & B_{1,1} \\
_\text{mm_load1_pd}: & \text{SSE instruction that loads} & \text{a double word and stores it in the high and} \\
B_{12} & B_{1,2} & B_{1,2} \\
_\text{low double words of the XMM register} & \end{array}
\]
Example: 2 x 2 Matrix Multiply

- First iteration intermediate result

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

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

- \( I = 1 \)

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

\_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 x 2 Matrix Multiply

- First iteration intermediate result

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

\[
c1 = \_mm\_add\_pd(c1,\_mm\_mul\_pd(a,b1));
\]
\[
c2 = \_mm\_add\_pd(c2,\_mm\_mul\_pd(a,b2));
\]

- \( I = 2 \)

\[
\begin{array}{c|c}
A_2 & A_{1,2} & A_{2,2} \\
B_{21} & B_{2,1} & B_{2,1} \\
B_{22} & B_{2,2} & B_{2,2}
\end{array}
\]

\_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 x 2 Matrix Multiply

- Second iteration intermediate result

\[
\begin{array}{c|c}
\text{C}_1 & \text{C}_2 \\
\hline
\text{C}_1 & \begin{array}{c}
A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\
A_{1,1}B_{1,1} + A_{2,1}B_{2,1}
\end{array} & \begin{array}{c}
A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\
A_{2,1}B_{1,1} + A_{2,2}B_{2,2}
\end{array} \\
\text{C}_2 & \begin{array}{c}
A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\
A_{1,1}B_{1,1} + A_{2,1}B_{2,1}
\end{array} & \begin{array}{c}
A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \\
A_{2,1}B_{1,2} + A_{2,2}B_{2,2}
\end{array}
\end{array}
\]

\[c_1 = _\text{mm}_\text{add}_\text{pd}(c_1, _\text{mm}_\text{mul}_\text{pd}(a,b_1));\]
\[c_2 = _\text{mm}_\text{add}_\text{pd}(c_2, _\text{mm}_\text{mul}_\text{pd}(a,b_2));\]

- \(I = 2\)

\[
\begin{array}{c|c}
\text{A}_2 & \text{A}_1 \text{,} 2 & \text{A}_2 \text{,} 2 \\
\hline
\text{B}_2 \text{,} 1 & \text{B}_2 \text{,} 1 & \text{B}_2 \text{,} 1 \\
\text{B}_2 \text{,} 2 & \text{B}_2 \text{,} 2 & \text{B}_2 \text{,} 2
\end{array}
\]

\text{\texttt{mm}}\text{\textunderscore\texttt{load}}\text{\textunderscore\texttt{pd}}: Stored in memory in Column order

\text{\texttt{mm}}\text{\textunderscore\texttt{load1}}\text{\textunderscore\texttt{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 <stdio.h>  
#include <emmintrin.h>  

// header file for SSE 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;  

    /* A =                           (note column order!)
       1 0  
       0 1  */

    /* B =                              (note column order!)
       1 3  
       2 4  */
    B[0] = 1.0; B[1] = 2.0; B[2] = 3.0; B[3] = 4.0;

    /* C =                             (note column order!)
       0 0  
       0 0  */
    C[0] = 0.0; C[1] = 0.0; C[2] = 0.0; C[3] = 0.0;
}
```
Example: 2 x 2 Matrix Multiply
(Part 2 of 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 = */
    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;
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: 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., in if-then-else)
Agenda

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

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

\[
\text{Speedup w/ E} = \frac{1}{[ (1-F) + \frac{F}{S} ]}
\]
Big Idea: Amdahl’s Law

Speedup =

Example: the execution time of half of the program can be accelerated by a factor of 2. What is the program speed-up overall?
Big Idea: Amdahl’s Law

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

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

\[
\frac{1}{0.5 + 0.5} = \frac{1}{0.5 + 0.25} = 1.33
\]
<table>
<thead>
<tr>
<th>Time</th>
<th>Number of Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel portion</td>
<td>Serial portion</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

The diagram shows the relationship between the number of processors and the time taken for a certain task, with a distinction between the parallel and serial portions of the process.
Big Idea: Amdahl’s Law

If the portion of the program that can be parallelized is small, then the speedup is limited.

The non-parallel portion limits the performance.
Example #1: Amdahl’s Law

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

- Consider an enhancement which runs 20 times faster but which is only usable 25% of the time
  \[
  \text{Speedup w/ E} = \frac{1}{(.75 + .25/20)} = 1.31
  \]

- What if its usable only 15% of the time?
  \[
  \text{Speedup w/ E} = \frac{1}{(.85 + .15/20)} = 1.17
  \]

- 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
  \[
  \text{Speedup w/ E} = \frac{1}{(.001 + .999/100)} = 90.99
  \]
Example #2: Amdahl’s Law

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

• Consider summing 10 scalar variables and two 10 by 10 matrices (matrix sum) on 10 processors
  
  \[ \text{Speedup w/ } E = \frac{1}{(.091 + .909/10)} = \frac{1}{0.1819} = 5.5 \]

• What if there are 100 processors?
  
  \[ \text{Speedup w/ } E = \frac{1}{(.091 + .909/100)} = \frac{1}{0.10009} = 10.0 \]

• What if the matrices are 100 by 100 (or 10,010 adds in total) on 10 processors?
  
  \[ \text{Speedup w/ } E = \frac{1}{(.001 + .999/10)} = \frac{1}{0.1009} = 9.9 \]

• What if there are 100 processors?
  
  \[ \text{Speedup w/ } E = \frac{1}{(.001 + .999/100)} = \frac{1}{0.01099} = 91 \]
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