CS 61C: Great Ideas in Computer Architecture (Machine Structures)
Amdahl's Law and Data-Level Parallelism

Instructors:

Krste Asanovic & Vladimir Stojanovic

Review

- Performance
  - Bandwidth, measured in tasks/second
  - Latency, time to complete one task
- "Iron Law" of computer performance:
  - Secs/program = insts/program * clocks/inst * secs/clock
- IEEE-754 Floating-Point Standard
  - Sign-magnitude significand* 2 ^ biased exponent
  - Special values, NaN, Infinity, Denormals

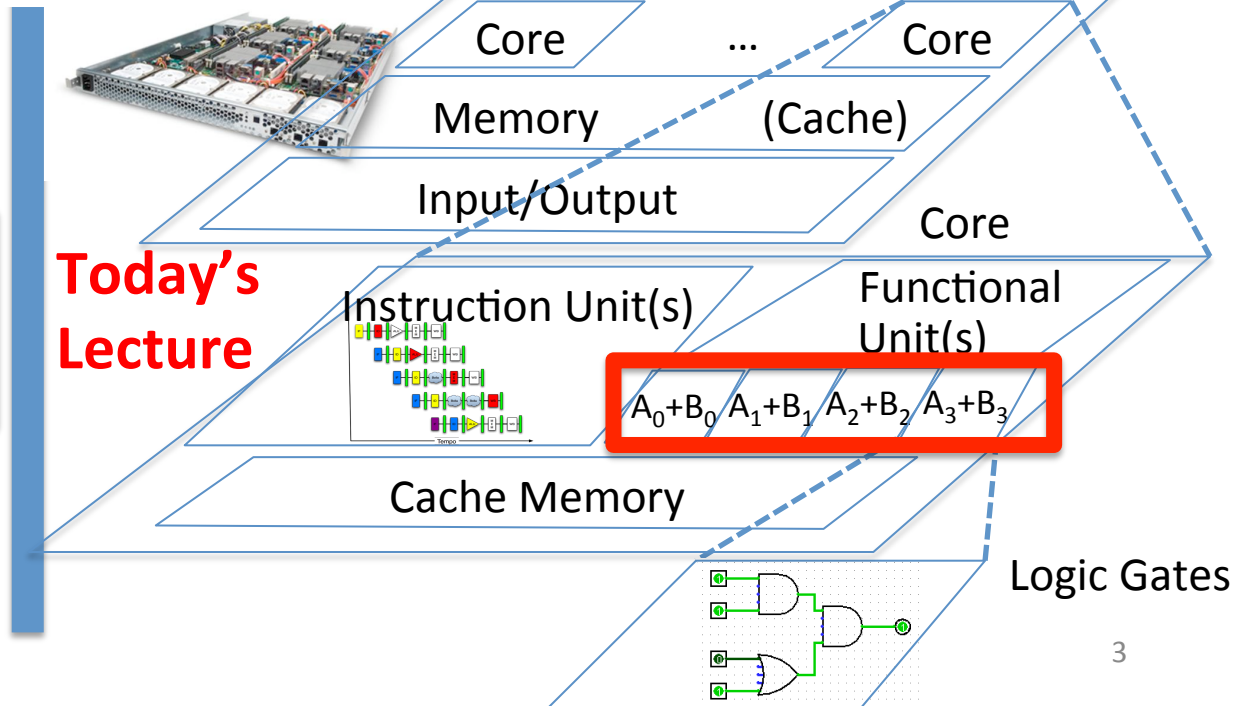# New-School Machine Structures (It's a bit more complicated!)

Software   *Hardware*

- **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 @ one time

- **Programming Languages**

Harness Parallelism & Achieve High Performance

Warehouse Scale Computer

Smart Phone

Today's Lecture

Computer

Core   ...   Core

Memory   (Cache)

Input/Output

Core

Instruction Unit(s)   Functional Unit(s)

A_0+B_0 A_1+B_1 A_2+B_2 A_3+B_3

Cache Memory

Logic Gates

Using Parallelism for Performance

- Two basic ways:
  - Multiprogramming
    - run multiple independent programs in parallel
    - "Easy"
  - Parallel computing
    - run one program faster
    - "Hard"

- We'll focus on parallel computing for next few lectures

Single-Instruction/Single-Data Stream (SISD)

SISD

Instruction Pool

Data Pool

PU
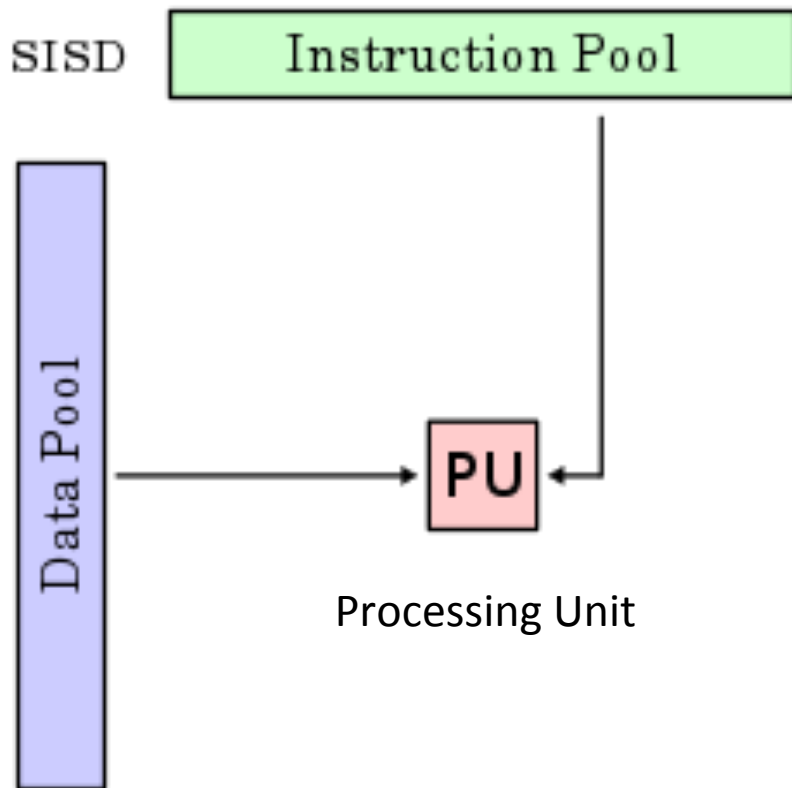
Processing Unit

- Sequential computer that exploits no parallelism in either the instruction or data streams. Examples of SISD architecture are traditional uniprocessor machines

Single-Instruction/Multiple-Data Stream (SIMD or "sim-dee")

SIMD

Instruction Pool

Data Pool

PU

PU

PU

PU
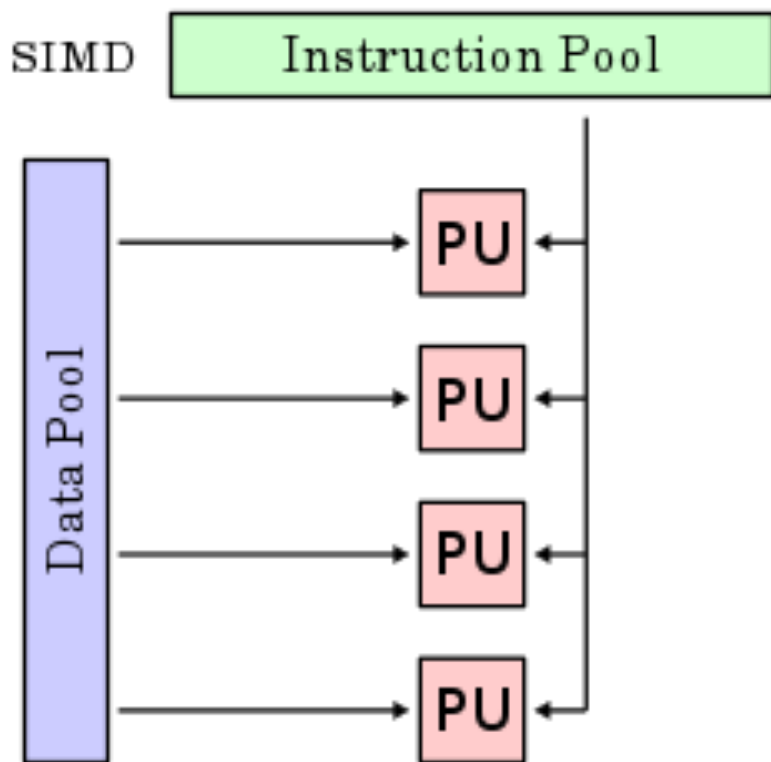
- SIMD computer exploits multiple data streams against a single instruction stream to operations that may be naturally parallelized, e.g., Intel SIMD instruction extensions or NVIDIA Graphics Processing Unit (GPU)

Multiple-Instruction/Multiple-Data Streams (MIMD or "mim-dee")

Instruction Pool

Data Pool

PU
PU
PU
PU
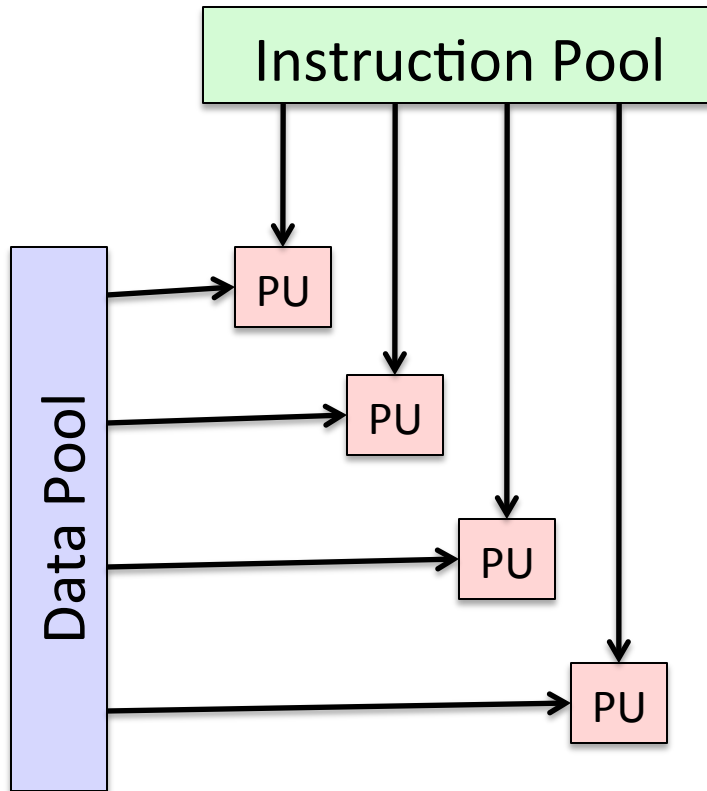
- \Multiple autonomous processors simultaneously executing different instructions on different data.

  - MIMD architectures include multicore and Warehouse-Scale Computers

Multiple-Instruction/Single-Data Stream (MISD)
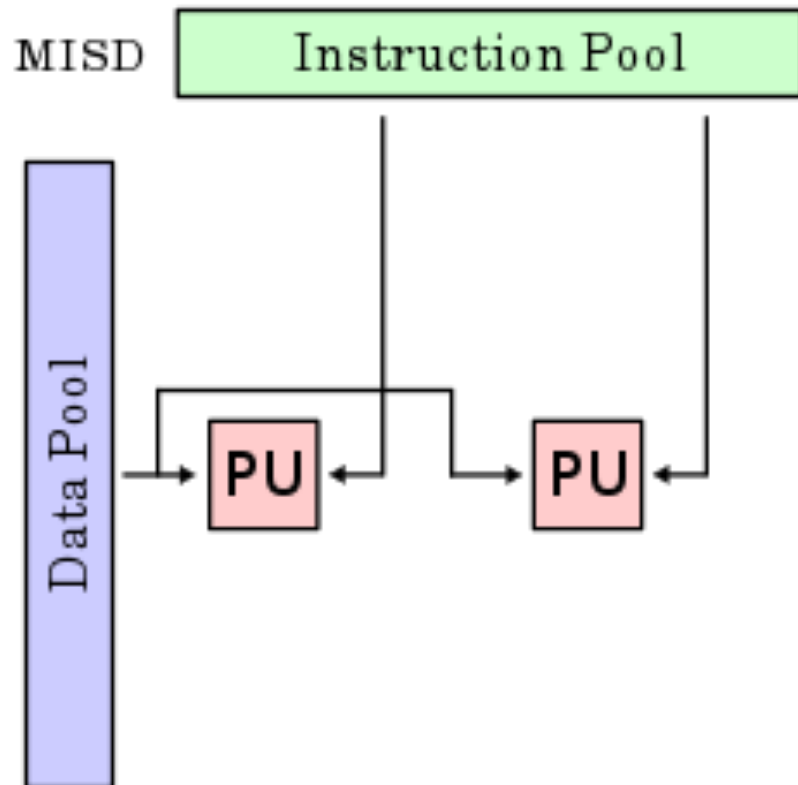
- Multiple-Instruction, Single-Data stream computer that exploits multiple instruction streams against a single data stream.
  - Rare, mainly of historical interest only

Flynn* Taxonomy, 1966

		Data Streams	
		Single	**Multiple**
Instruction Streams	Single	SISD: Intel Pentium 4	SIMD: SSE instructions of x86
	Multiple	MISD: No examples today	MIMD: Intel Xeon e5345 (Clovertown)

- In 2013, SIMD and MIMD most common parallelism in architectures – usually both in same system!
- 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 using synchronization primitives
- SIMD (aka hw-level *data parallelism*): specialized function units, for handling lock-step calculations involving arrays
  - Scientific computing, signal processing, multimedia (audio/video processing)
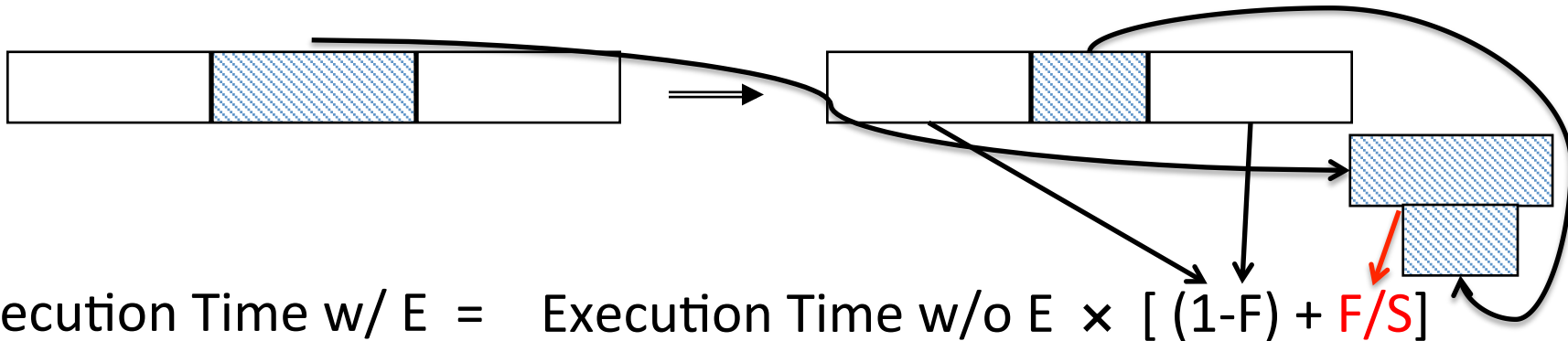
*Prof. Michael Flynn, Stanford

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

Execution Time w/ E =   Execution Time w/o E × [(1-F) + F/S]

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

$$\text{Speedup} = \cfrac{1}{(1-F) + \cfrac{F}{S}}$$

Non-speed-up part → (1 - F)

Speed-up part → 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?

$$\cfrac{1}{0.5 + \cfrac{0.5}{2}} = \frac{1}{0.5 + 0.25} = 1.33$$

Example #1: Amdahl's Law

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

- Consider an enhancement which runs 20 times faster but which is only usable 25% of the time

  Speedup w/ E = $1/(.75 + .25/20)$ = 1.31
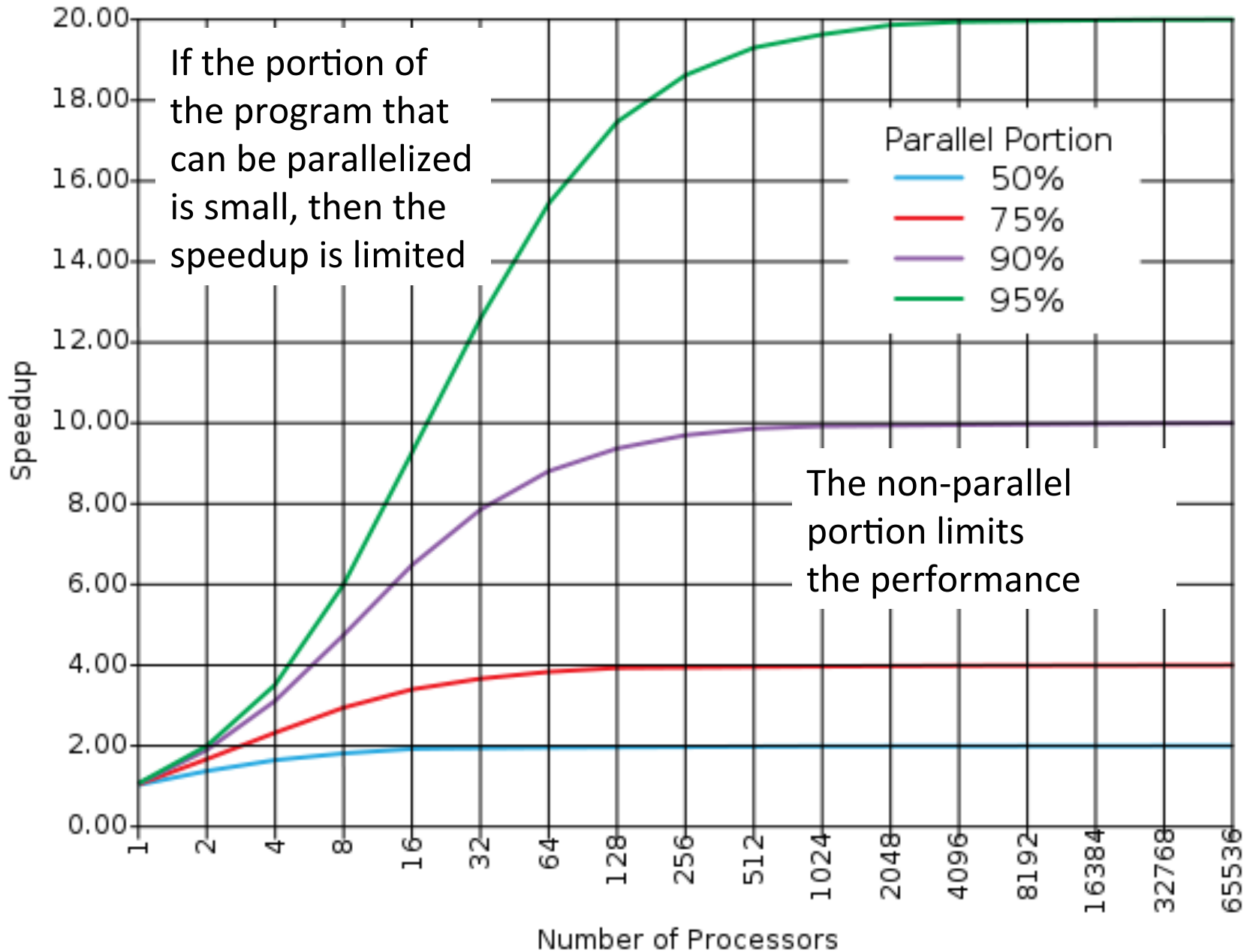

- What if its usable only 15% of the time?

  Speedup w/ E = $1/(.85 + .15/20)$ = 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

  Speedup w/ E = $1/(.001 + .999/100)$ = 90.99

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

Parallel Portion
- 50%
- 75%
- 90%
- 95%

Speedup (y-axis): 0.00, 2.00, 4.00, 6.00, 8.00, 10.00, 12.00, 14.00, 16.00, 18.00, 20.00

Number of Processors (x-axis): 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536

Strong and Weak Scaling

- To get good speedup on a parallel processor 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 one unit with twice the load of others cuts speedup almost in half

Clickers/Peer Instruction

Suppose a program spends 80% of its time in a square root routine. How much must you speedup square root to make the program run 5 times faster?

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

A: 5

B: 16

C: 20

D: 100

E: None of the above

Administrivia

- MT2 is Thursday, April 9th:
  - Conflict: Email Sagar by midnight **today**
  - TA Review Session:
    - April 4th, 1-3pm, 145 Dwinelle
  - Guerrilla Section (caches + FP):
    - April 5th, 3-5pm, 2nd floor Soda labs
  - HKN Review Session: (info coming soon)
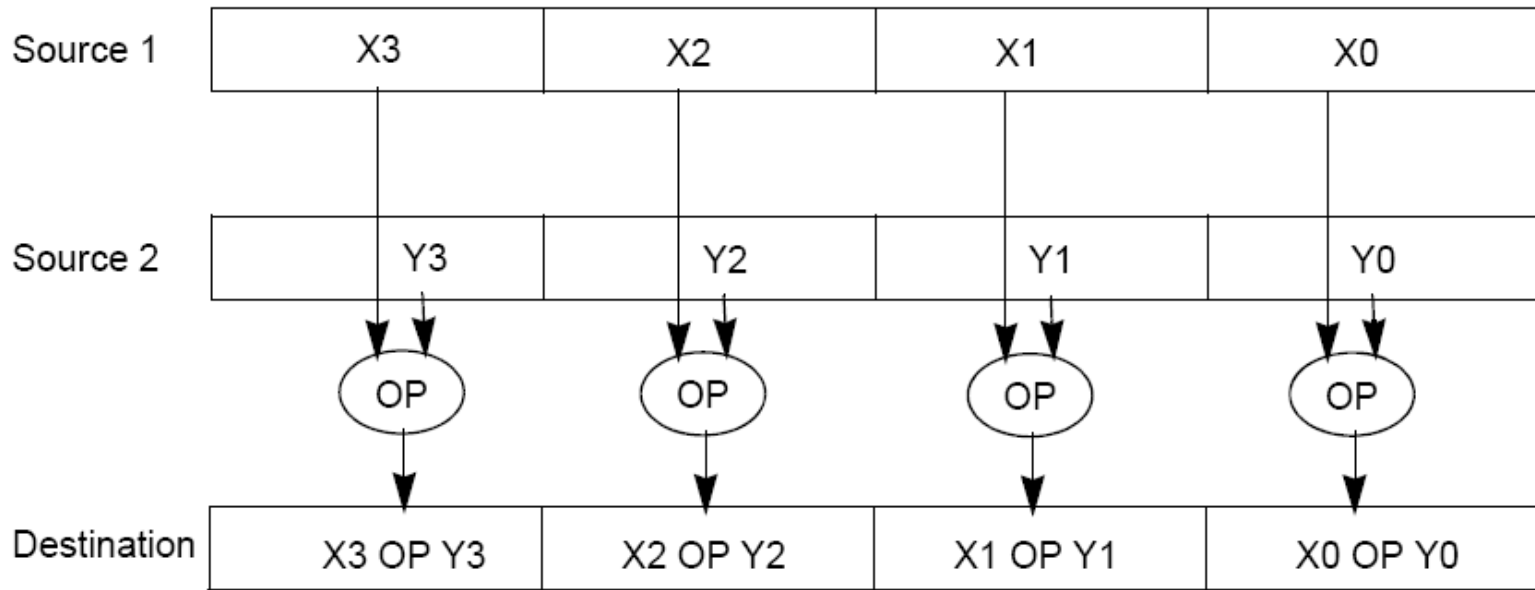
SIMD Architectures

- *Data parallelism*: executing same operation on multiple data streams
- Example to provide context:
  - Multiplying a coefficient vector by a data vector (e.g., in filtering)

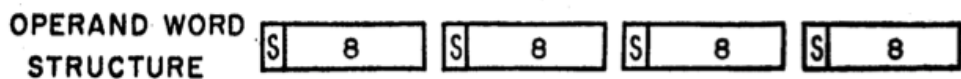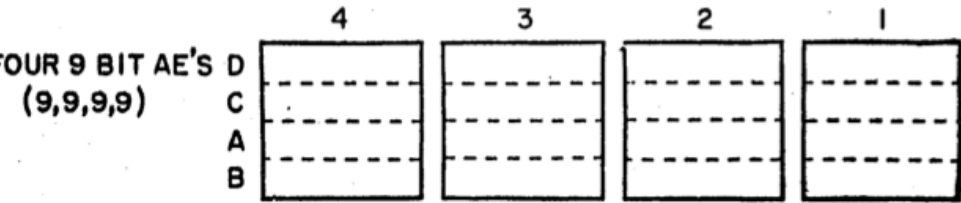$$y[i] \; := \; c[i] \; \times \; x[i], \; 0 \; \leq \; i \; < \; n$$
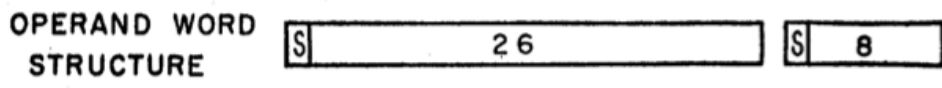
- Sources of performance improvement:
  - One instruction is fetched & decoded for entire operation
  - Multiplications are known to be independent
  - Pipelining/concurrency in memory access as well

Intel "Advanced Digital Media Boost"

- To improve performance, Intel's SIMD instructions
  - Fetch one instruction, do the work of multiple instructions

Source 1	X3	X2	X1	X0
Source 2	Y3	Y2	Y1	Y0
	OP	OP	OP	OP
Destination	X3 OP Y3	X2 OP Y2	X1 OP Y1	X0 OP Y0
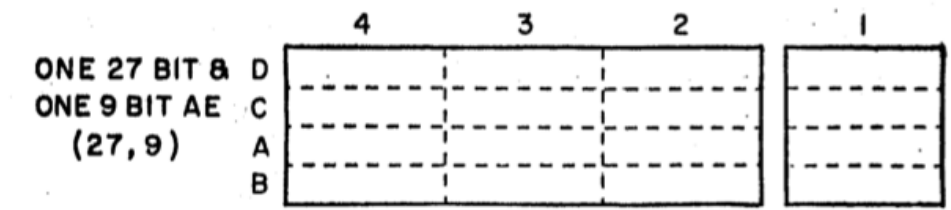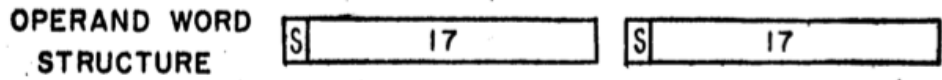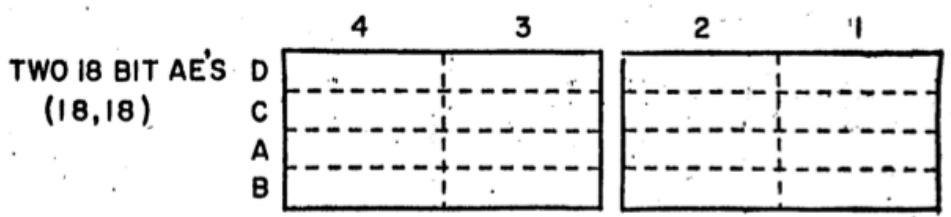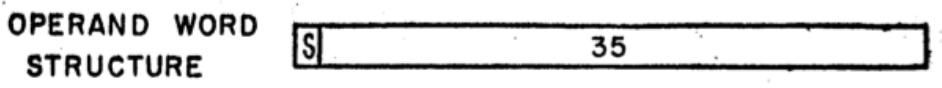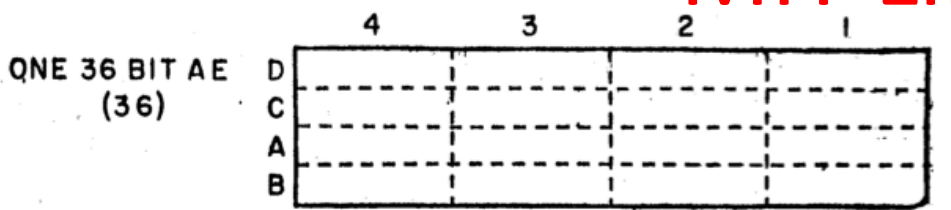
Intel SIMD Extensions

- MMX 64-bit registers, reusing floating-point registers [1992]

- SSE2/3/4, new 128-bit registers [1999]

- AVX, new 256-bit registers [2011]
  - Space for expansion to 1024-bit registers

XMM Registers
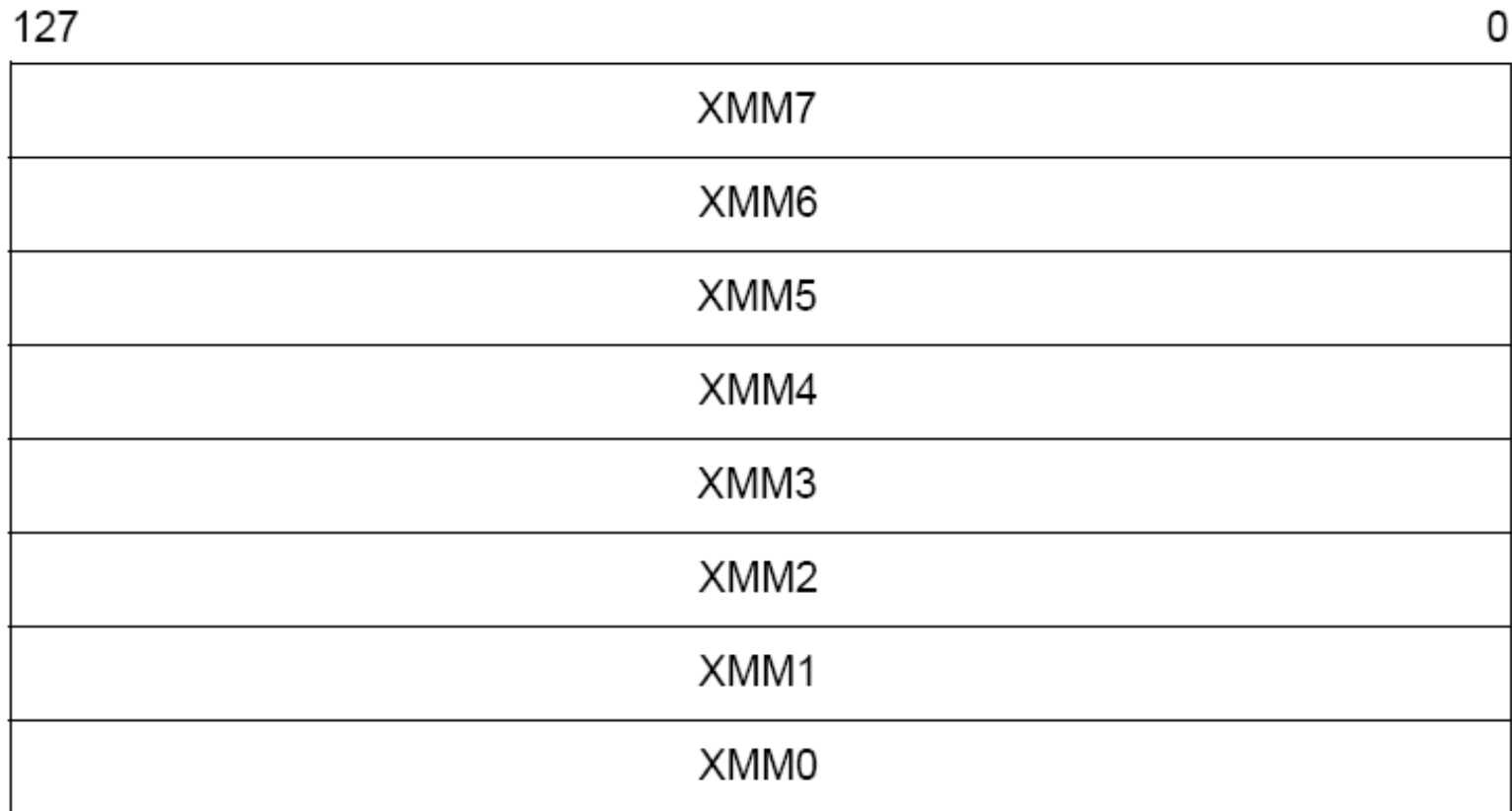
- Architecture extended with eight 128-bit data registers: XMM registers
  - x86 64-bit address architecture adds 8 additional registers (XMM8 – XMM15)

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)

Fundamental 128-Bit Packed SIMD Data Types

Packed Bytes																16 / 128 bits

127 122 121 96 95 80 79 64 63 48 47 32 31 16 15 0

Packed Words — 8 / 128 bits

127 122 121 96 95 80 79 64 63 48 47 32 31 16 15 0

Packed Doublewords — 4 / 128 bits

127 96 95 64 63 32 31 0

Packed Quadwords — 2 / 128 bits

127 64 63 0
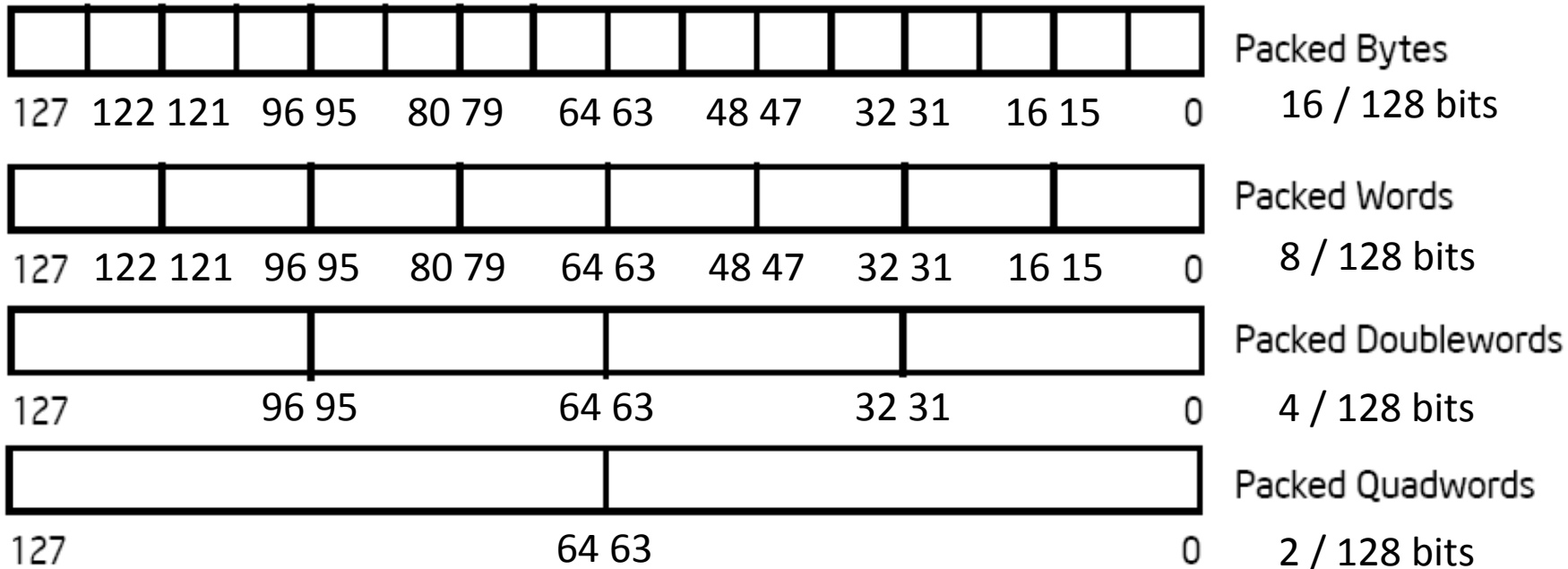
SSE/SSE2 Floating Point Instructions

Move does both load and store

Data transfer	Arithmetic	Compare
MOV{A/U}{SS/PS/SD/PD} xmm, mem/xmm	ADD{SS/PS/SD/PD} xmm, mem/xmm	CMP{SS/PS/SD/PD}
	SUB{SS/PS/SD/PD} xmm, mem/xmm	
MOV {H/L} {PS/PD} xmm, mem/xmm	MUL{SS/PS/SD/PD} xmm, mem/xmm	
	DIV{SS/PS/SD/PD} xmm, mem/xmm	
	SQRT{SS/PS/SD/PD} mem/xmm	
	MAX {SS/PS/SD/PD} mem/xmm	
	MIN{SS/PS/SD/PD} mem/xmm	

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

Packed and Scalar Double-Precision Floating-Point Operations

Packed

Scalar
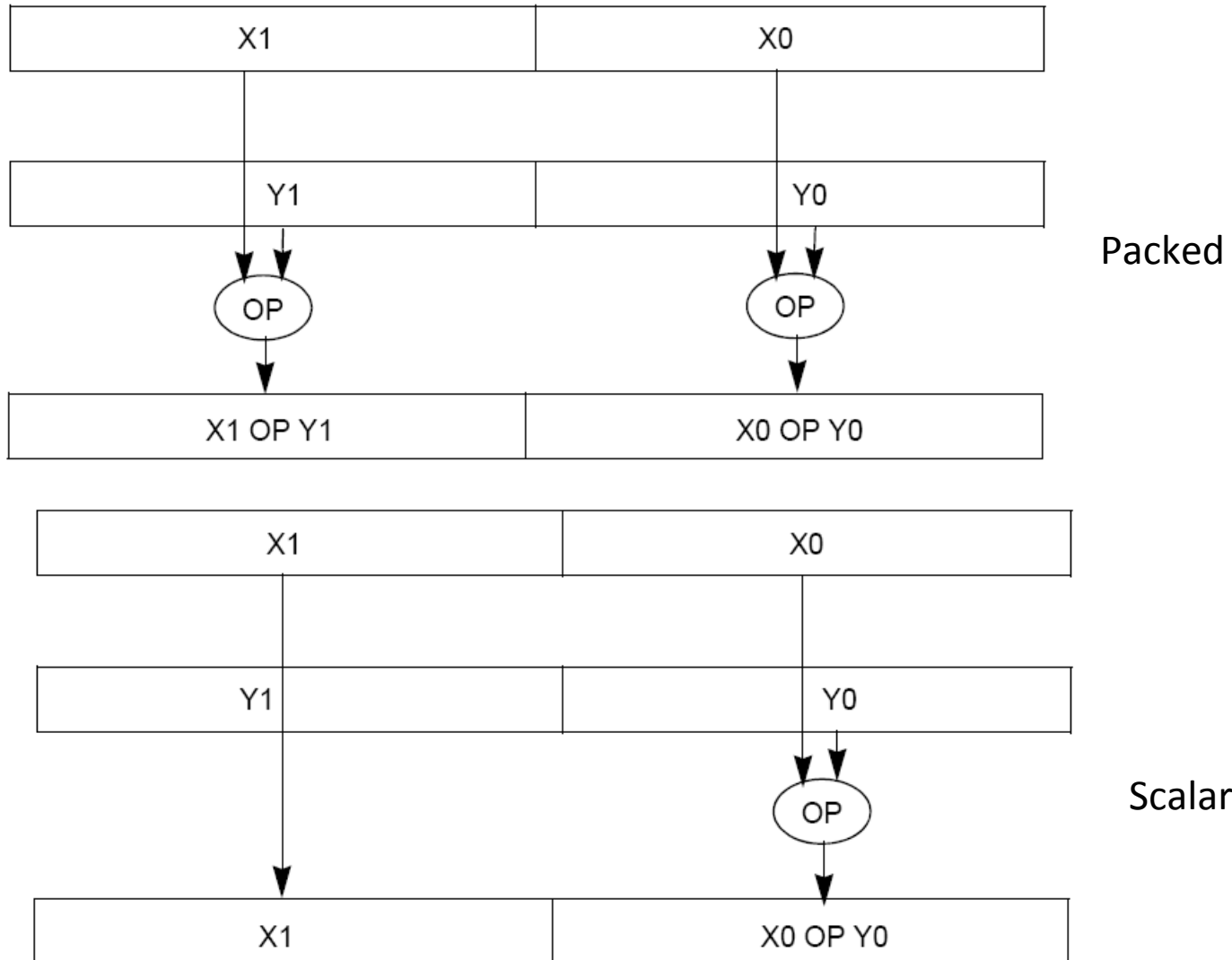
Example: SIMD Array Processing

```
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
    store the 4 results from the register to memory

}
```

SIMD style

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=1000; i>0; i=i-1)
    x[i] = x[i] + s;
```

- How can reveal more data-level parallelism than available in a single iteration of a loop?

- *Unroll loop* and adjust iteration rate

Looping in MIPS

Assumptions:

- $t1 is initially the address of the element in the array with the highest address

- $f0 contains the scalar value s

- 8($t2) is the address of the last element to operate on

CODE:

```
Loop: 1. l.d        $f2,0($t1)        ; $f2=array element
      2. add.d     $f10,$f2,$f0      ; add s to $f2
      3. s.d        $f10,0($t1)       ; store result
      4. addui     $t1,$t1,#-8       ; decrement pointer 8 byte
      5. bne        $t1,$t2,Loop     ;repeat loop if $t1 != $t2
```

Loop Unrolled

```
Loop:   l.d      $f2,0($t1)
        add.d    $f10,$f2,$f0
        s.d      $f10,0($t1)
        l.d      $f4,-8($t1)
        add.d    $f12,$f4,$f0
        s.d      $f12,-8($t1)
        l.d      $f6,-16($t1)
        add.d    $f14,$f6,$f0
        s.d      $f14,-16($t1)
        l.d      $f8,-24($t1)
        add.d    $f16,$f8,$f0
        s.d      $f16,-24($t1)
        addui    $t1,$t1,#-32
        bne      $t1,$t2,Loop
```

NOTE:
1. Only 1 Loop Overhead every 4 iterations
2. This unrolling works if
   loop_limit(mod 4) = 0
3. Using different registers for each iteration eliminates data hazards in pipeline

Loop Unrolled Scheduled

```
Loop: l.d    $f2,0($t1)
      l.d    $f4,-8($t1)
      l.d    $f6,-16($t1)
      l.d    $f8,-24($t1)
      add.d  $f10,$f2,$f0
      add.d  $f12,$f4,$f0
      add.d  $f14,$f6,$f0
      add.d  $f16,$f8,$f0
      s.d    $f10,0($t1)
      s.d    $f12,-8($t1)
      s.d    $f14,-16($t1)
      s.d    $f16,-24($t1)
      addui  $t1,$t1,#-32
      bne    $t1,$t2,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
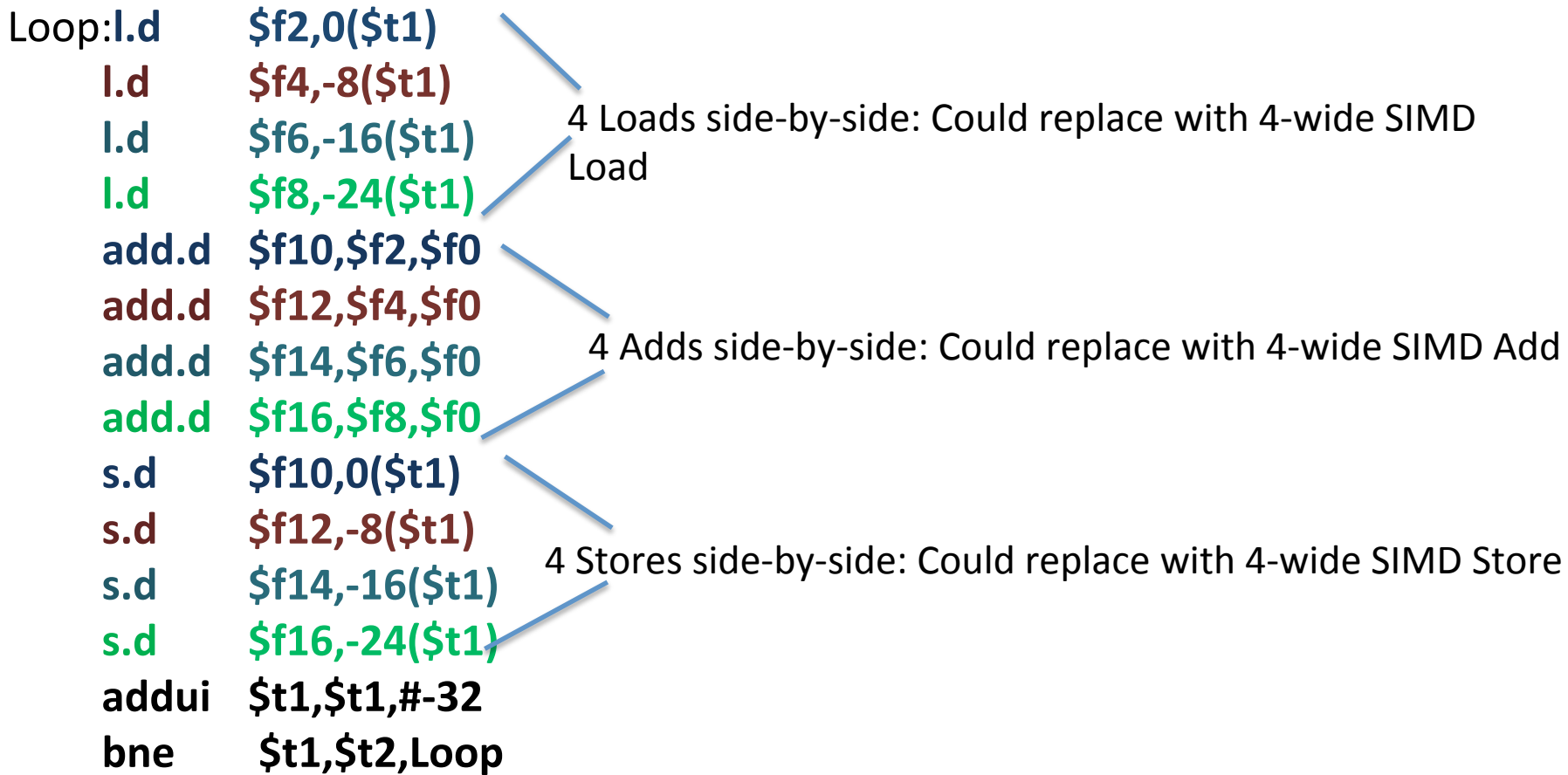
Loop Unrolling in C

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

```
for(i=1000; i>0; i=i-1)
    x[i] = x[i] + s;
```

- Could be rewritten   What is downside of doing it in C?

```
for(i=1000; i>0; 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;
    }
```

Generalizing Loop Unrolling

- A loop of **n iterations**

- **k copies** of the body of the loop

- **Assuming (n mod k) ≠ 0**

  Then we will run the loop with 1 copy of the body **(n mod k)** times and with k copies of the body **floor(n/k)** times

Example: Add Two Single-Precision Floating-Point 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;
```

mov a ps : **mov**e from mem to XMM register, memory **a**ligned, **p**acked **s**ingle precision

add ps : **add** from mem to XMM register, **p**acked **s**ingle precision

mov a ps : **mov**e from XMM register to mem, memory **a**ligned, **p**acked **s**ingle precision

SSE Instruction Sequence:

(Note: Destination on the right in x86 assembly)

```
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
```

In The News: Intel to buy Altera?

- Altera is 2nd biggest FPGA maker after Xilinx
  - FPGA (Field-Programmable Gate Array)
- Altera already has fabrication deal to use Intel's 14nm technology
- Intel experimenting with FPGA next to server processor
- Microsoft to use programmable logic chips to accelerate Bing search engine

Intel SSE Intrinsics

- Intrinsics are C functions and procedures for inserting assembly language into C code, including SSE instructions

  – With intrinsics, can program using these instructions indirectly

  – One-to-one correspondence between SSE instructions and intrinsics

Example SSE Intrinsics

Instrinsics: Corresponding SSE instructions:

- 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/duplicating

- 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:

$$C_{i,j} = (A \times B)_{i,j} = \sum_{k=1}^{2} A_{i,k} \times B_{k,j}$$
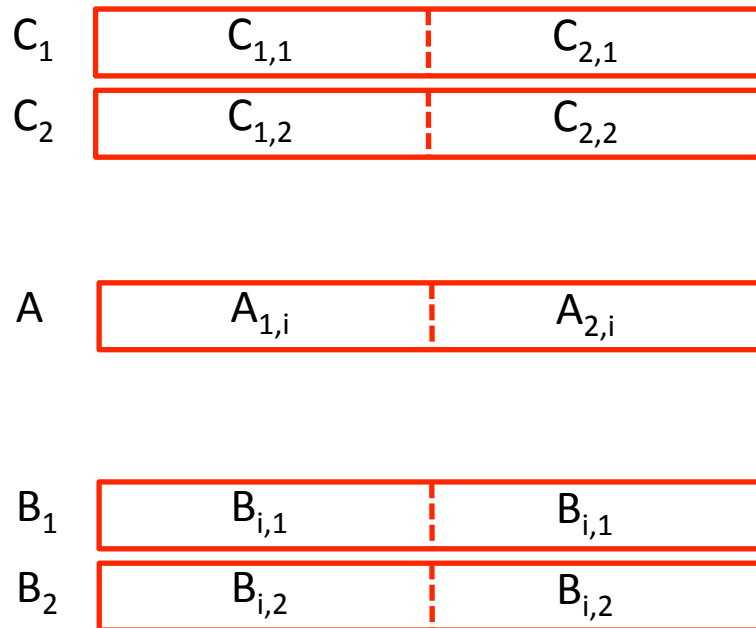
$$
\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}
\times
\begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}
=
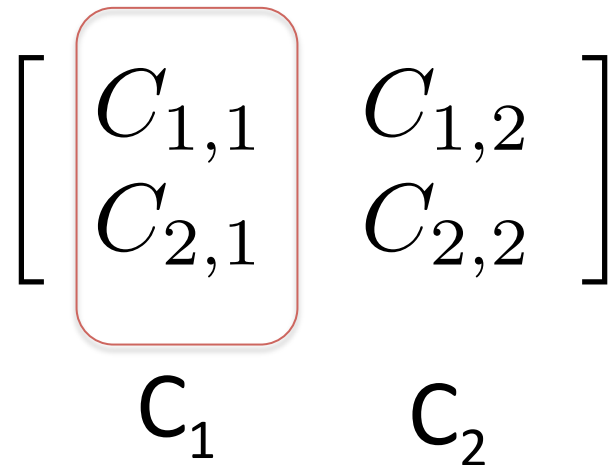\begin{bmatrix} C_{1,1}=A_{1,1}B_{1,1}+A_{1,2}B_{2,1} & C_{1,2}=A_{1,1}B_{1,2}+A_{1,2}B_{2,2} \\ C_{2,1}=A_{2,1}B_{1,1}+A_{2,2}B_{2,1} & C_{2,2}=A_{2,1}B_{1,2}+A_{2,2}B_{2,2} \end{bmatrix}
$$

$$
\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}
\times
\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}
=
\begin{bmatrix} C_{1,1}= 1*1 + 0*2 = 1 & C_{1,2}= 1*3 + 0*4 = 3 \\ C_{2,1}= 0*1 + 1*2 = 2 & C_{2,2}= 0*3 + 1*4 = 4 \end{bmatrix}
$$

Example: 2 x 2 Matrix Multiply

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

C_1 | $C_{1,1}$ | $C_{2,1}$
C_2 | $C_{1,2}$ | $C_{2,2}$

A | $A_{1,i}$ | $A_{2,i}$

B_1 | $B_{i,1}$ | $B_{i,1}$
B_2 | $B_{i,2}$ | $B_{i,2}$

Stored in memory in Column order

$$\begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

C_1 C_2

Example: 2 x 2 Matrix Multiply

- Initialization

C_1
0	0

C_2
0	0

Example: 2 x 2 Matrix Multiply

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \times \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix} = \begin{bmatrix} C_{1,1}=A_{1,1}B_{1,1}+A_{1,2}B_{2,1} & C_{1,2}=A_{1,1}B_{1,2}+A_{1,2}B_{2,2} \\ C_{2,1}=A_{2,1}B_{1,1}+A_{2,2}B_{2,1} & C_{2,2}=A_{2,1}B_{1,2}+A_{2,2}B_{2,2} \end{bmatrix}$$

- Initialization

C_1 | 0 | 0
C_2 | 0 | 0

- I = 1
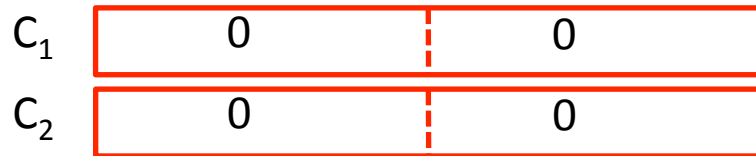
A | $A_{1,1}$ | $A_{2,1}$

_mm_load_pd: Load 2 doubles into XMM reg, Stored in memory in Column order

B_1 | $B_{1,1}$ | $B_{1,1}$
B_2 | $B_{1,2}$ | $B_{1,2}$

_mm_load1_pd: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register (duplicates value in both halves of XMM)

Example: 2 x 2 Matrix Multiply

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \times \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix} = \begin{bmatrix} C_{1,1}=A_{1,1}B_{1,1}+A_{1,2}B_{2,1} & C_{1,2}=A_{1,1}B_{1,2}+A_{1,2}B_{2,2} \\ C_{2,1}=A_{2,1}B_{1,1}+A_{2,2}B_{2,1} & C_{2,2}=A_{2,1}B_{1,2}+A_{2,2}B_{2,2} \end{bmatrix}$$

- First iteration intermediate result

C_1	$0+A_{1,1}B_{1,1}$	$0+A_{2,1}B_{1,1}$
C_2	$0+A_{1,1}B_{1,2}$	$0+A_{2,1}B_{1,2}$

c1 = _mm_add_pd(c1,_mm_mul_pd(a,b1));
c2 = _mm_add_pd(c2,_mm_mul_pd(a,b2));
SSE instructions first do parallel multiplies and then parallel adds in XMM registers

- I = 1

A	$A_{1,1}$	$A_{2,1}$

_mm_load_pd: Stored in memory in Column order

B_1	$B_{1,1}$	$B_{1,1}$
B_2	$B_{1,2}$	$B_{1,2}$

_mm_load1_pd: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register (duplicates value in both halves of XMM)

Example: 2 x 2 Matrix Multiply

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \times \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix} = \begin{bmatrix} C_{1,1}=A_{1,1}B_{1,1}+A_{1,2}B_{2,1} & C_{1,2}=A_{1,1}B_{1,2}+A_{1,2}B_{2,2} \\ C_{2,1}=A_{2,1}B_{1,1}+A_{2,2}B_{2,1} & C_{2,2}=A_{2,1}B_{1,2}+A_{2,2}B_{2,2} \end{bmatrix}$$

- First iteration intermediate result

C_1	$0+A_{1,1}B_{1,1}$	$0+A_{2,1}B_{1,1}$
C_2	$0+A_{1,1}B_{1,2}$	$0+A_{2,1}B_{1,2}$

c1 = _mm_add_pd(c1,_mm_mul_pd(a,b1));
c2 = _mm_add_pd(c2,_mm_mul_pd(a,b2));
SSE instructions first do parallel multiplies and then parallel adds in XMM registers

- I = 2

A	$A_{1,2}$	$A_{2,2}$

_mm_load_pd: Stored in memory in Column order

B_1	$B_{2,1}$	$B_{2,1}$
B_2	$B_{2,2}$	$B_{2,2}$

_mm_load1_pd: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register (duplicates value in both halves of XMM)

Example: 2 x 2 Matrix Multiply

- Second iteration intermediate result

$C_{1,1}$ $C_{2,1}$

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 | $A_{1,1}B_{1,2}+A_{1,2}B_{2,2}$ | $A_{2,1}B_{1,2}+A_{2,2}B_{2,2}$ |

$C_{1,2}$ $C_{2,2}$

c1 = _mm_add_pd(c1,_mm_mul_pd(a,b1));
c2 = _mm_add_pd(c2,_mm_mul_pd(a,b2));
SSE instructions first do parallel multiplies and then parallel adds in XMM registers

- I = 2

A | $A_{1,2}$ | $A_{2,2}$ |

_mm_load_pd: Stored in memory in Column order

B_1 | $B_{2,1}$ | $B_{2,1}$ |

B_2 | $B_{2,2}$ | $B_{2,2}$ |

_mm_load1_pd: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register (duplicates value in both halves of XMM)

Example: 2 x 2 Matrix Multiply

Definition of Matrix Multiply:

$$C_{i,j} = (A \times B)_{i,j} = \sum_{k=1}^{2} A_{i,k} \times B_{k,j}$$

$$
\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}
\times
\begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}
=
\begin{bmatrix} C_{1,1}=A_{1,1}B_{1,1}+A_{1,2}B_{2,1} & C_{1,2}=A_{1,1}B_{1,2}+A_{1,2}B_{2,2} \\ C_{2,1}=A_{2,1}B_{1,1}+A_{2,2}B_{2,1} & C_{2,2}=A_{2,1}B_{1,2}+A_{2,2}B_{2,2} \end{bmatrix}
$$

$$
\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}
\times
\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}
=
\begin{bmatrix} C_{1,1}=1*1+0*2=1 & C_{1,2}=1*3+0*4=3 \\ C_{2,1}=0*1+1*2=2 & C_{2,2}=0*3+1*4=4 \end{bmatrix}
$$

Example: 2 x 2 Matrix Multiply (Part 1 of 2)

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

// 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 A[4] __attribute__ ((aligned (16)));
    double B[4] __attribute__ ((aligned (16)));
    double C[4] __attribute__ ((aligned (16)));
    int lda = 2;
    int i = 0;
    // declare several 128-bit vector variables
    __m128d c1,c2,a,b1,b2;
```

```c
// Initialize A, B, C for example
/* A =                 (note column order!)
    1 0
    0 1
 */
A[0] = 1.0; A[1] = 0.0;  A[2] = 0.0;  A[3] = 1.0;

/* 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 =
      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+0*lda);
    /* b2 =
      i = 0: [b_12 | b_12]
      i = 1: [b_22 | b_22]
    */
    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;
}
```

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]
```

And in Conclusion, …

- Amdahl's Law: Serial sections limit speedup
- Flynn Taxonomy
- Intel SSE SIMD Instructions
  - Exploit data-level parallelism in loops
  - One instruction fetch that operates on multiple operands simultaneously
  - 128-bit XMM registers
- SSE Instructions in C
  - Embed the SSE machine instructions directly into C programs through use of intrinsics
  - Achieve efficiency beyond that of optimizing compiler