

CS 110

Computer Architecture

Amdahl's Law, Data-level Parallelism

Instructor:
Sören Schwertfeger

<https://robotics.shanghaitech.edu.cn/courses/ca>

School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkley's CS61C

Admin

- Midterm II
 - May 10 (This Friday!)
 - Topics: SDS, RISC-V Pipelining, Superscalar, Caches, Floating Point
- Same rules as Midterm I except:
 - 2 cheat A4 sheets are allowed (both handwritten in English by you)
 - RISC-V green sheet will be provided again

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

Software

Hardware

Warehouse
Scale
Computer



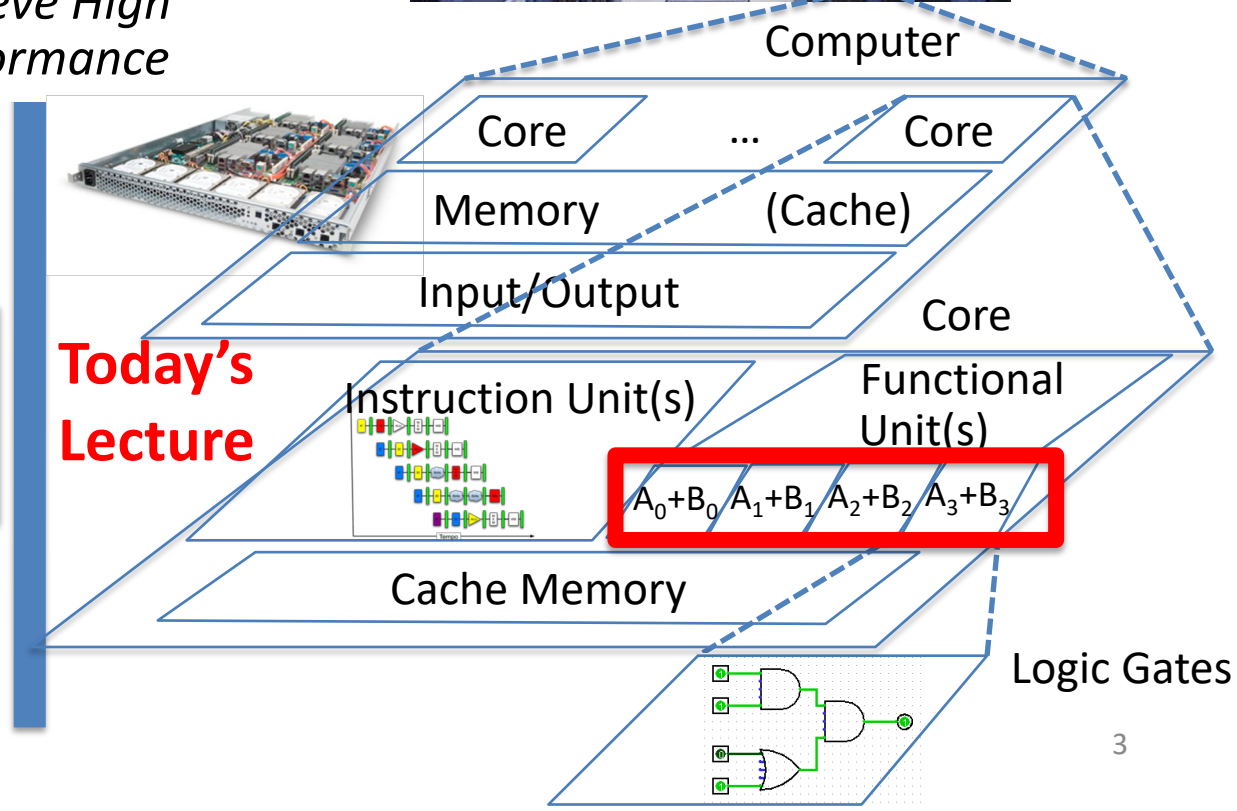
Smart
Phone



*Harness
Parallelism &
Achieve High
Performance*

- 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

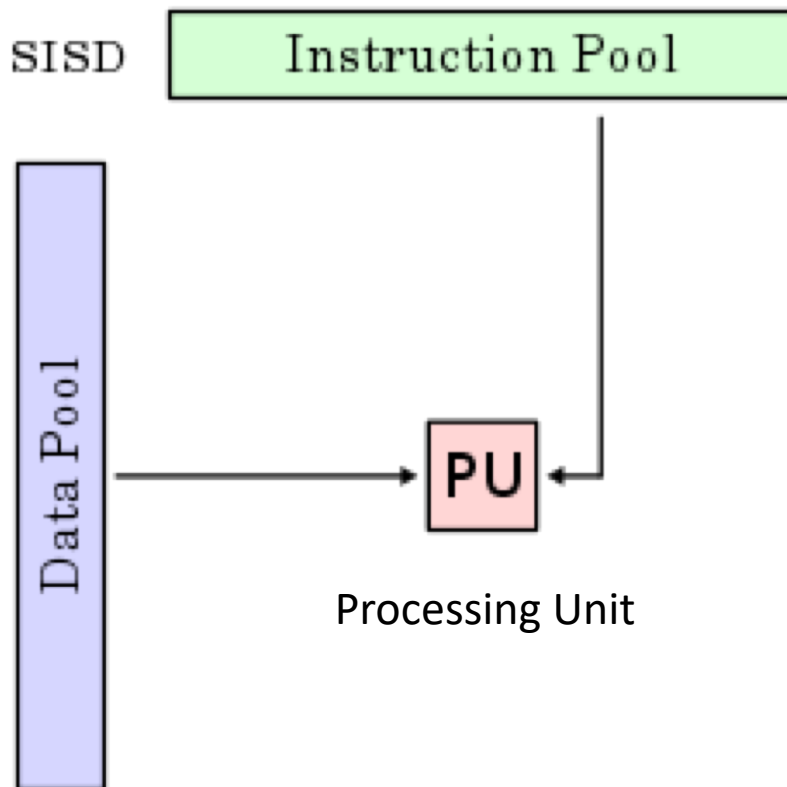
**Today's
Lecture**



Using Parallelism for Performance

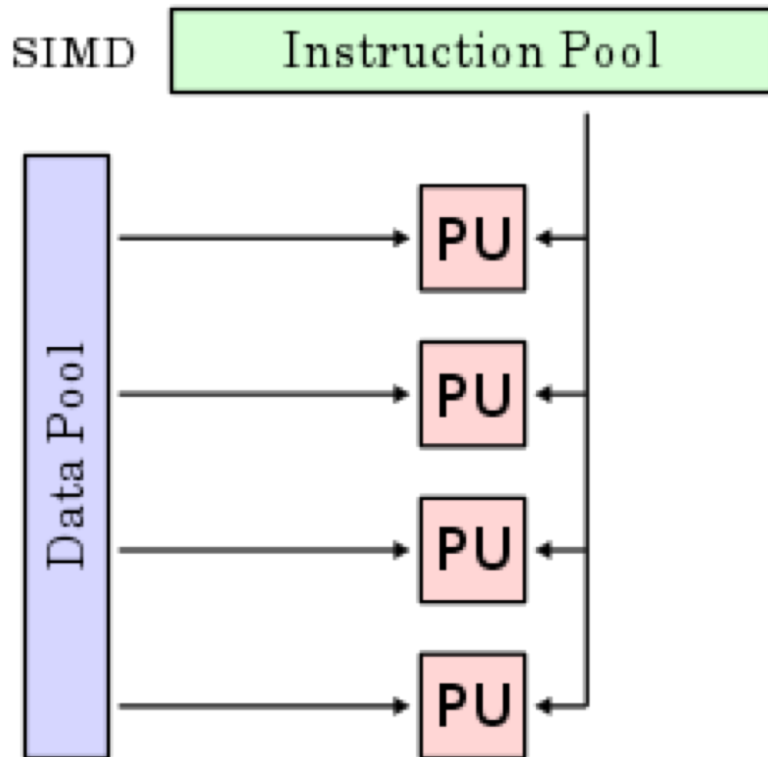
- Two basic ways:
 - Multiprogramming
 - run multiple independent programs in parallel
 - “Easy”
 - Parallel computing
 - run one program faster
 - “Hard”
- We’ll focus on parallel computing for next few lectures

Single-Instruction/Single-Data Stream (SISD)



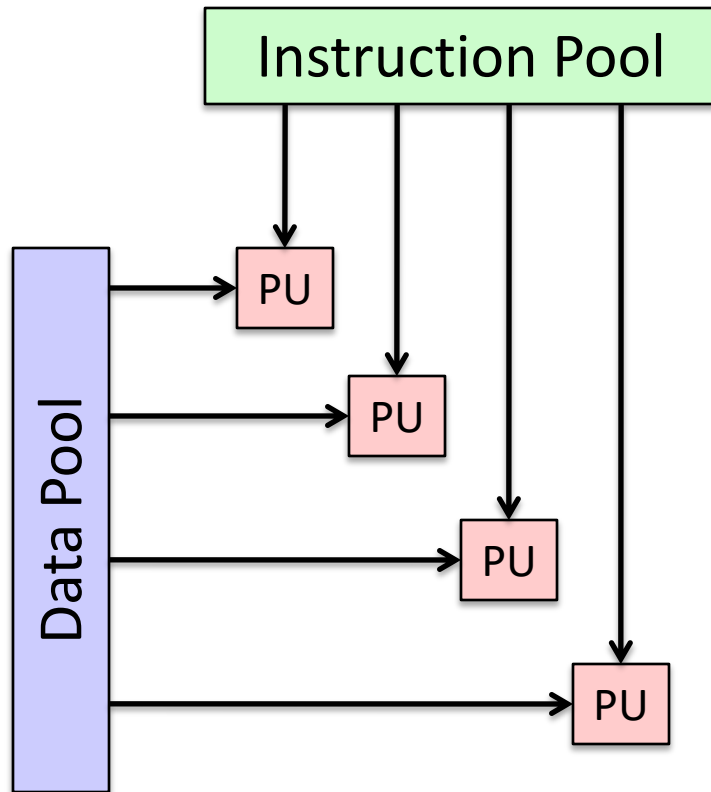
- Sequential computer that exploits no parallelism in either the instruction or data streams. Examples of SISD architecture are traditional uniprocessor machines

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



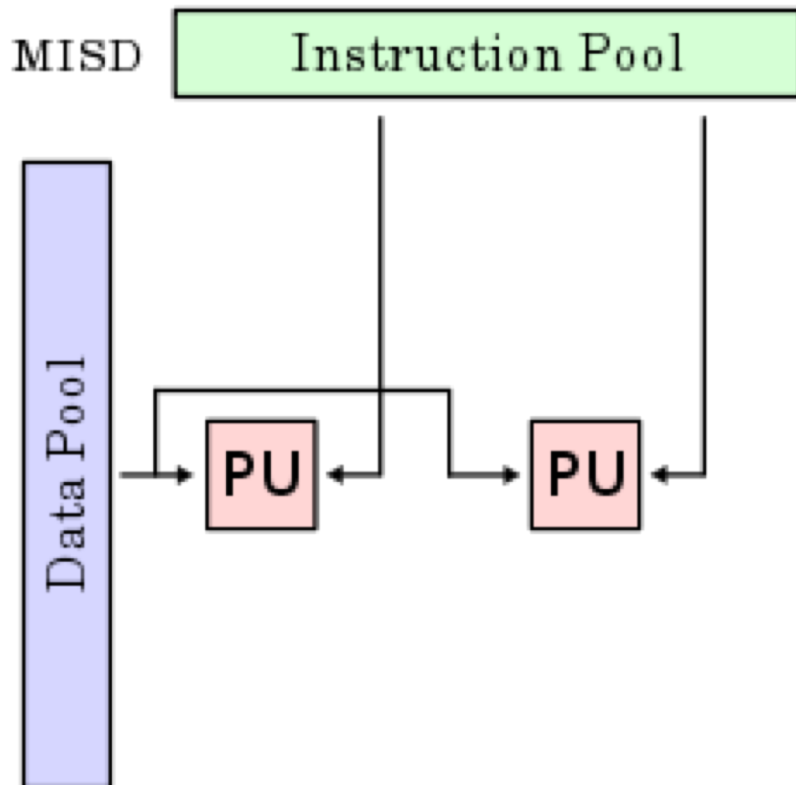
- SIMD computer exploits multiple data streams against a single instruction stream to operations that may be naturally parallelized, e.g., Intel SIMD instruction extensions or NVIDIA Graphics Processing Unit (GPU)

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



- 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)

- Since about 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

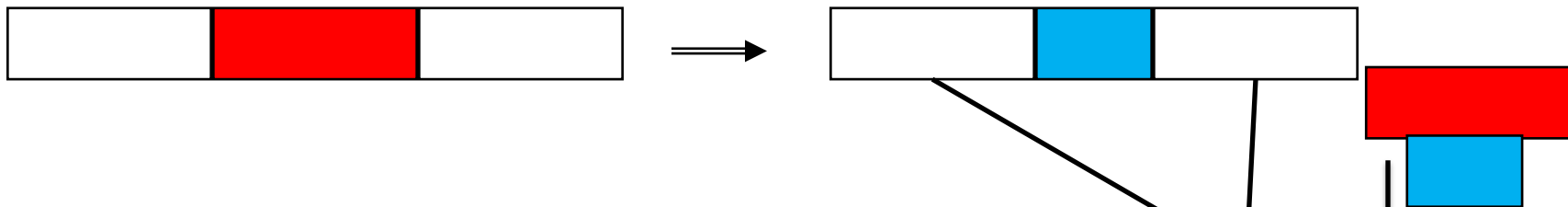


Big Idea: Amdahl's (Heartbreaking) Law

- 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



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

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

Big Idea: Amdahl's Law

$$\text{Speedup} = \frac{1}{\underbrace{(1 - F)}_{\text{Non-speed-up part}} + \underbrace{\frac{F}{S}}_{\text{Speed-up part}}}$$

Example: the execution time of half of the program can be accelerated by a factor of 2.

What is the program speed-up overall?

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

Example #1: Amdahl's Law

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

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

$$\text{Speedup w/ E} = 1 / (.75 + .25/20) = 1.31$$

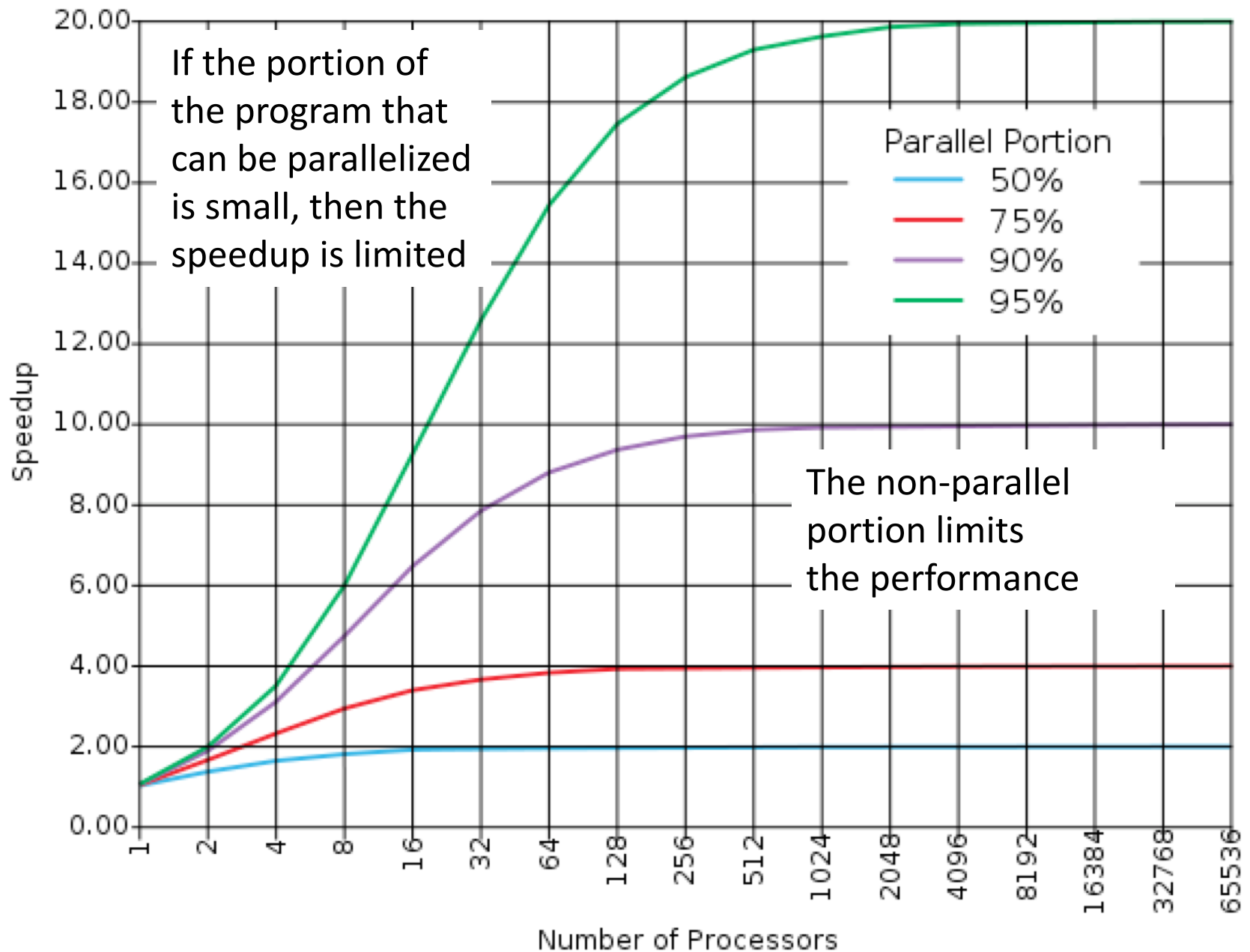
- What if its usable only 15% of the time?

$$\text{Speedup w/ E} = 1 / (.85 + .15/20) = 1.17$$

- 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

$$\text{Speedup w/ E} = 1 / (.001 + .999/100) = 90.99$$

Amdahl's Law



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

Question

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?

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

A: 5

B: 16

C: 20

D: 100

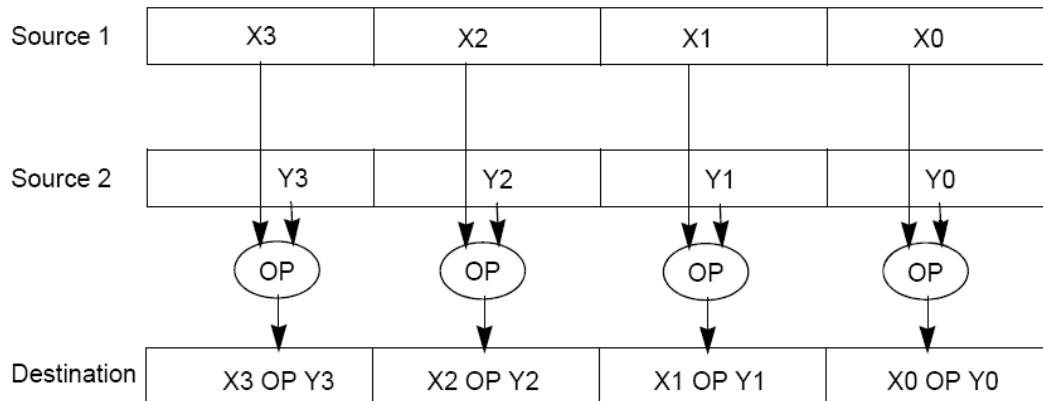
E: None of the above

SIMD Architectures

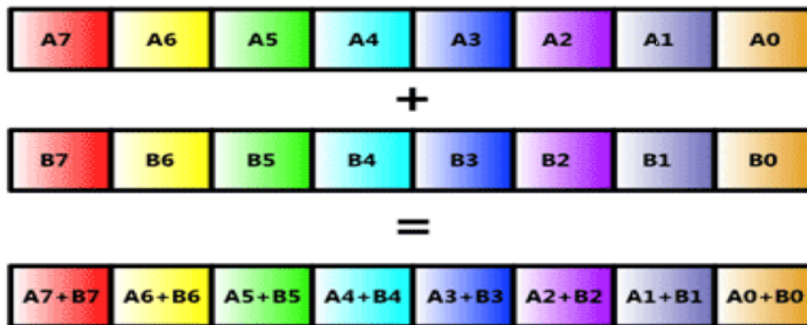
- *Data parallelism*: executing same operation on multiple data streams
- Example to provide context:
 - Multiplying a coefficient vector by a data vector (e.g., in filtering)
$$y[i] := c[i] \times x[i], \quad 0 \leq i < n$$
- Sources of performance improvement:
 - One instruction is fetched & decoded for entire operation
 - Multiplications are known to be independent
 - Pipelining/concurrency in memory access as well

Intel “Advanced Digital Media Boost”

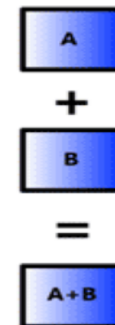
- To improve performance, Intel’s SIMD instructions
 - Fetch one instruction, do the work of multiple instructions



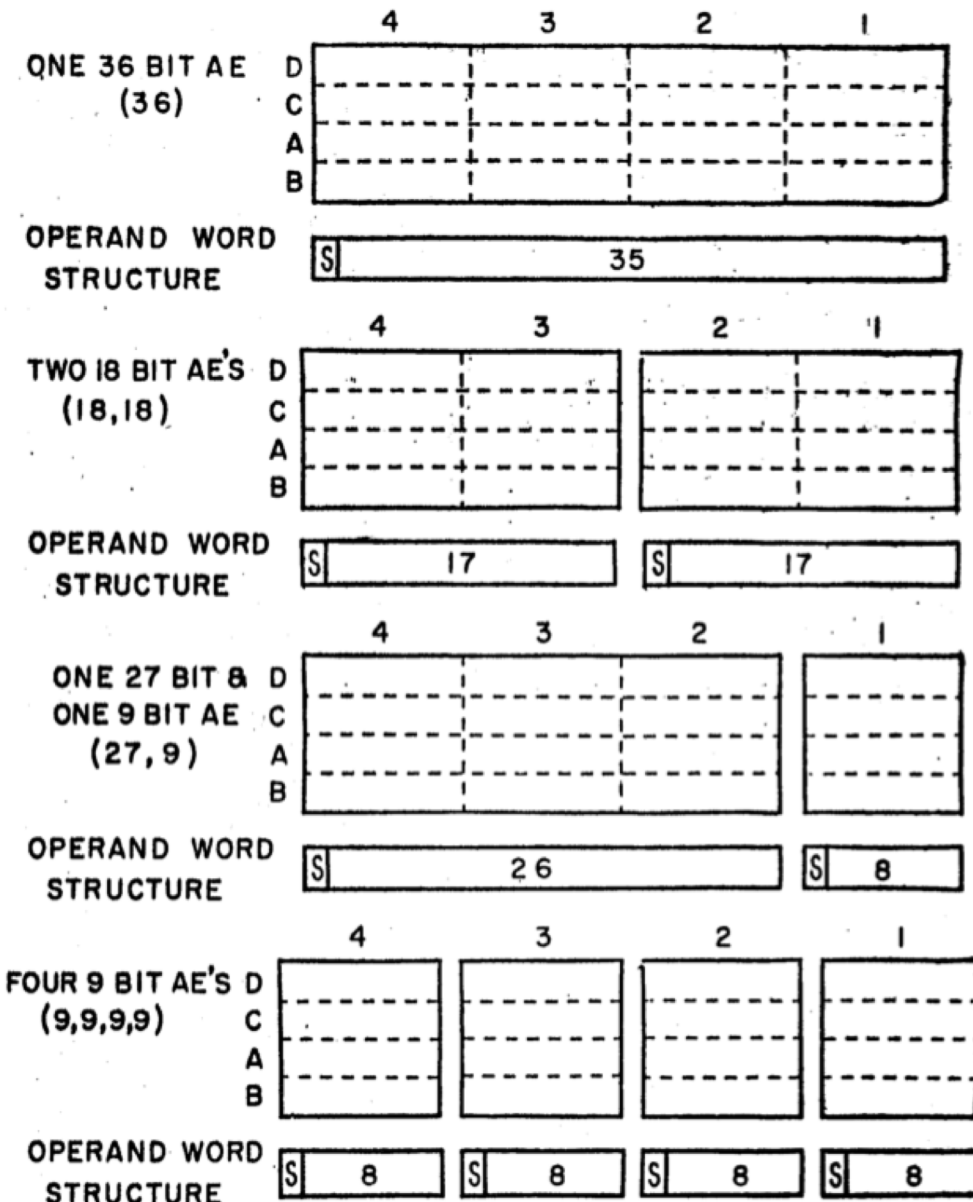
SIMD Mode



Scalar Mode



First SIMD Extensions: MIT Lincoln Labs TX-2, 1957



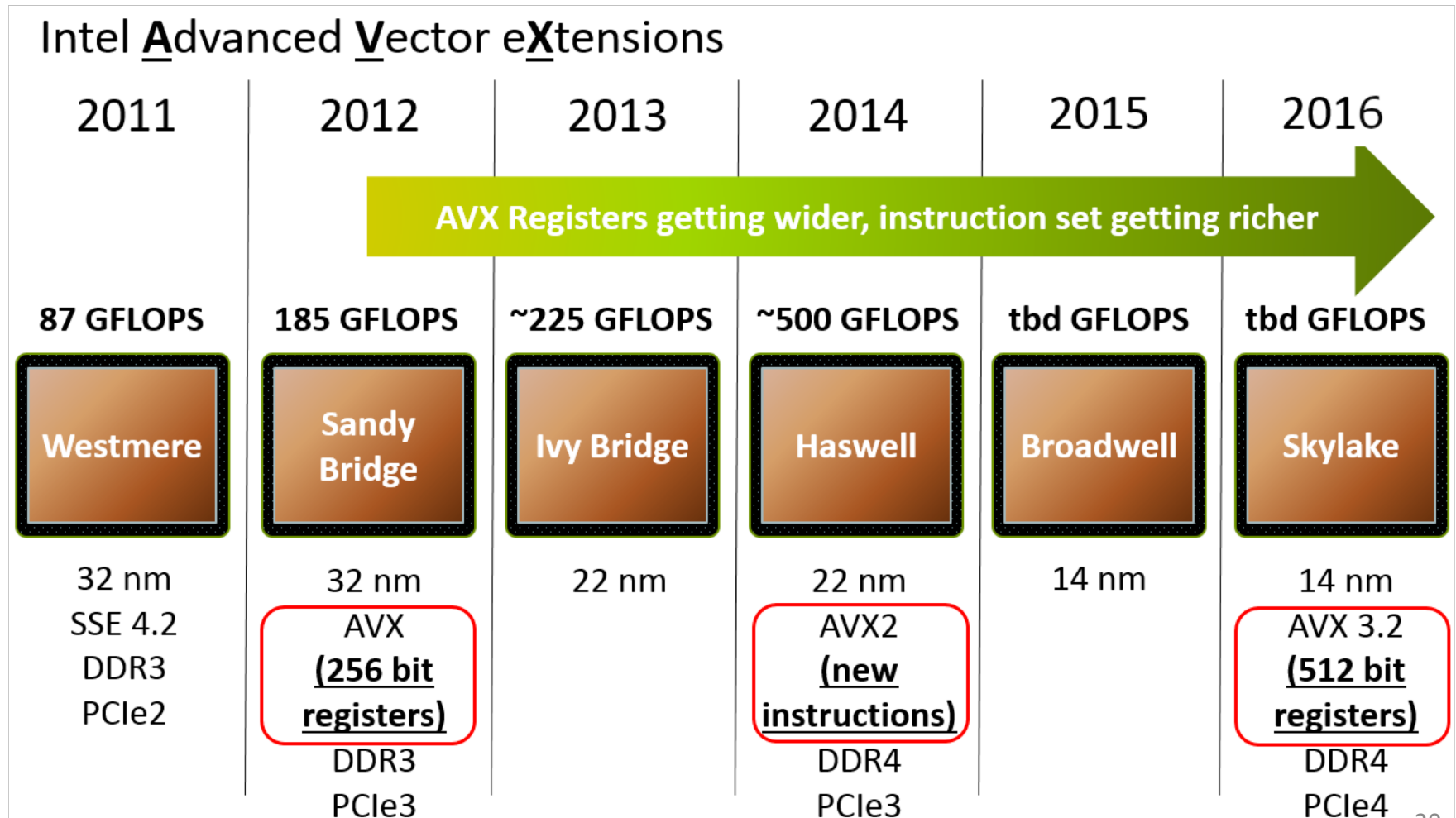
Intel SIMD Extensions

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

MMX 1997

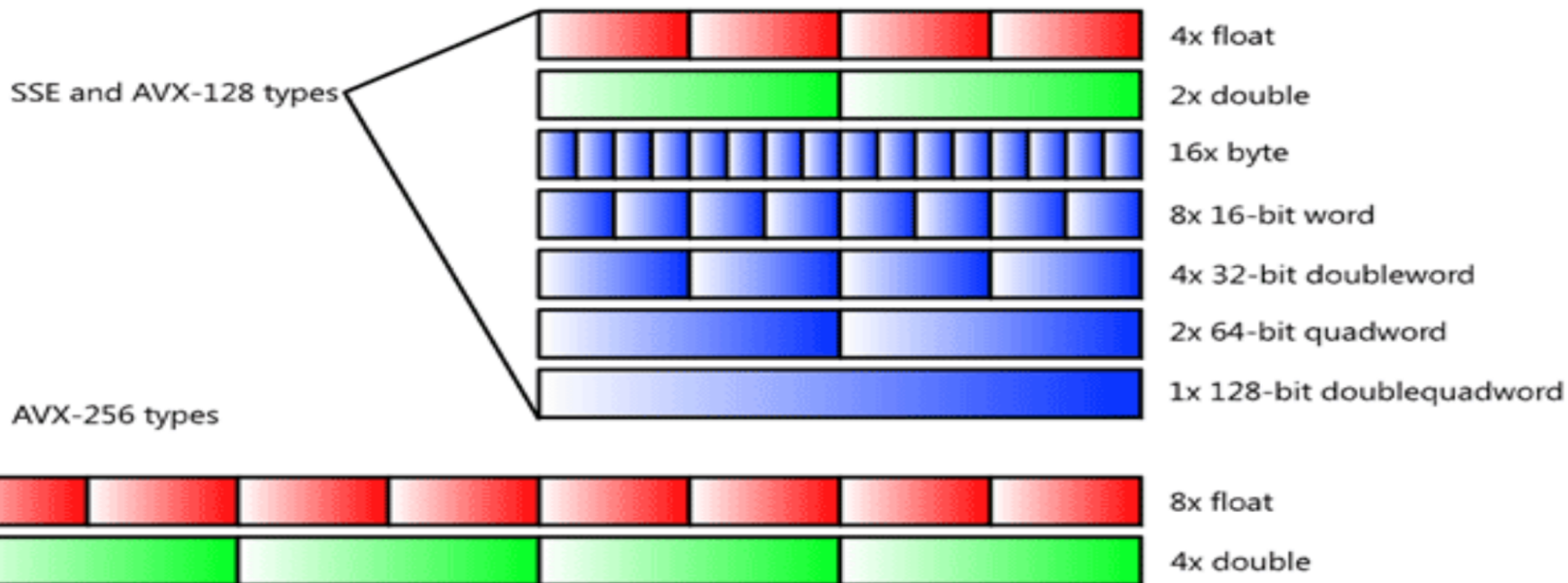
1999	2000	2004	2006	2007	2008	2009	2010\11
SSE	SSE2	SSE3	SSSE3	SSE4.1	SSE4.2	AES-NI	AVX
70 instr Single-Precision Vectors Streaming operations	144 instr Double-precision Vectors 8/16/32 64/128-bit vector integer	13 instr Complex Data	32 instr Decode	47 instr Video Graphics building blocks Advanced vector instr	8 instr String/XML processing POP-Count CRC	7 instr Encryption and Decryption Key Generation	~100 new instr. ~300 legacy sse instr updated 256-bit vector 3 and 4- operand instructions

Intel Advanced Vector eXtensions AVX



Intel Architecture SSE SIMD Data Types

- Note: in Intel Architecture (unlike RISC-V) a word is 16 bits
 - Single-precision FP: Double word (32 bits)
 - Double-precision FP: Quad word (64 bits)
 - AVX-512 available 16x float and 8x double)



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

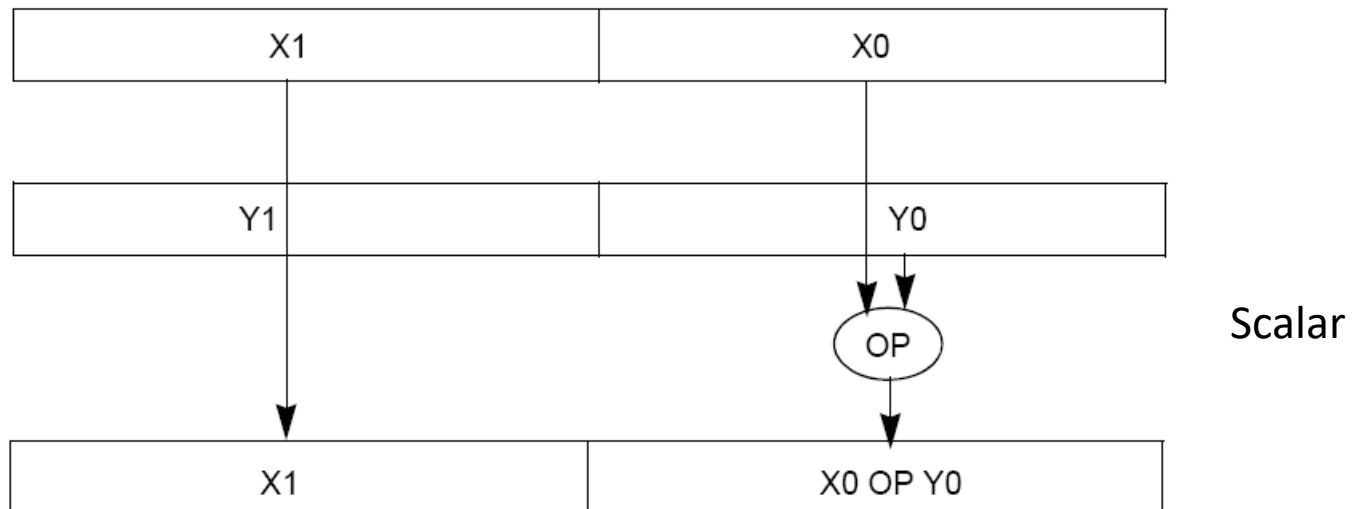
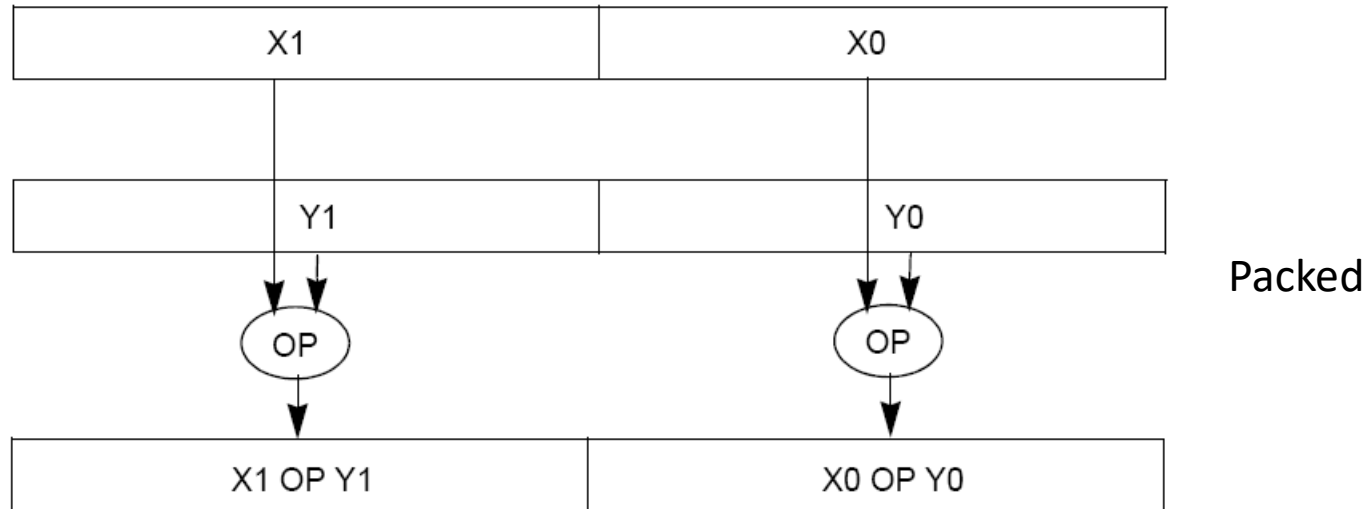
{A} 128-bit operand is aligned in memory

{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



X86 SIMD Intrinsics



Technologies

- ☐ MMX
- ☐ SSE
- ☐ SSE2
- ☐ SSE3
- ☐ SSSE3
- ☐ SSE4.1
- ☐ SSE4.2
- ☒ AVX
- ☐ AVX2
- ☐ FMA
- ☐ AVX-512
- ☐ KNC
- ☐ SVML
- ☐ Other

Categories

- ☐ Application-Targeted
- ☐ Arithmetic
- ☐ Bit Manipulation
- ☐ Cast
- ☐ Compare

mul_pd

```
__m256d _mm256_mul_pd (__m256d a, __m256d b)
```

Synopsis

```
__m256d _mm256_mul_pd (__m256d a, __m256d b)  
#include "immintrin.h"  
Instruction: vmulpd ymm, ymm, ymm  
CPUID Flags: AVX
```

Intrinsic

assembly instruction

Description

Multiply packed double-precision (64-bit) floating-point elements in **a** and **b**, and store the results in **dst**.

Operation

```
FOR j := 0 to 3  
  i := j*64  
  dst[i+63:i] := a[i+63:i] * b[i+63:i]  
ENDFOR  
dst[MAX:256] := 0
```

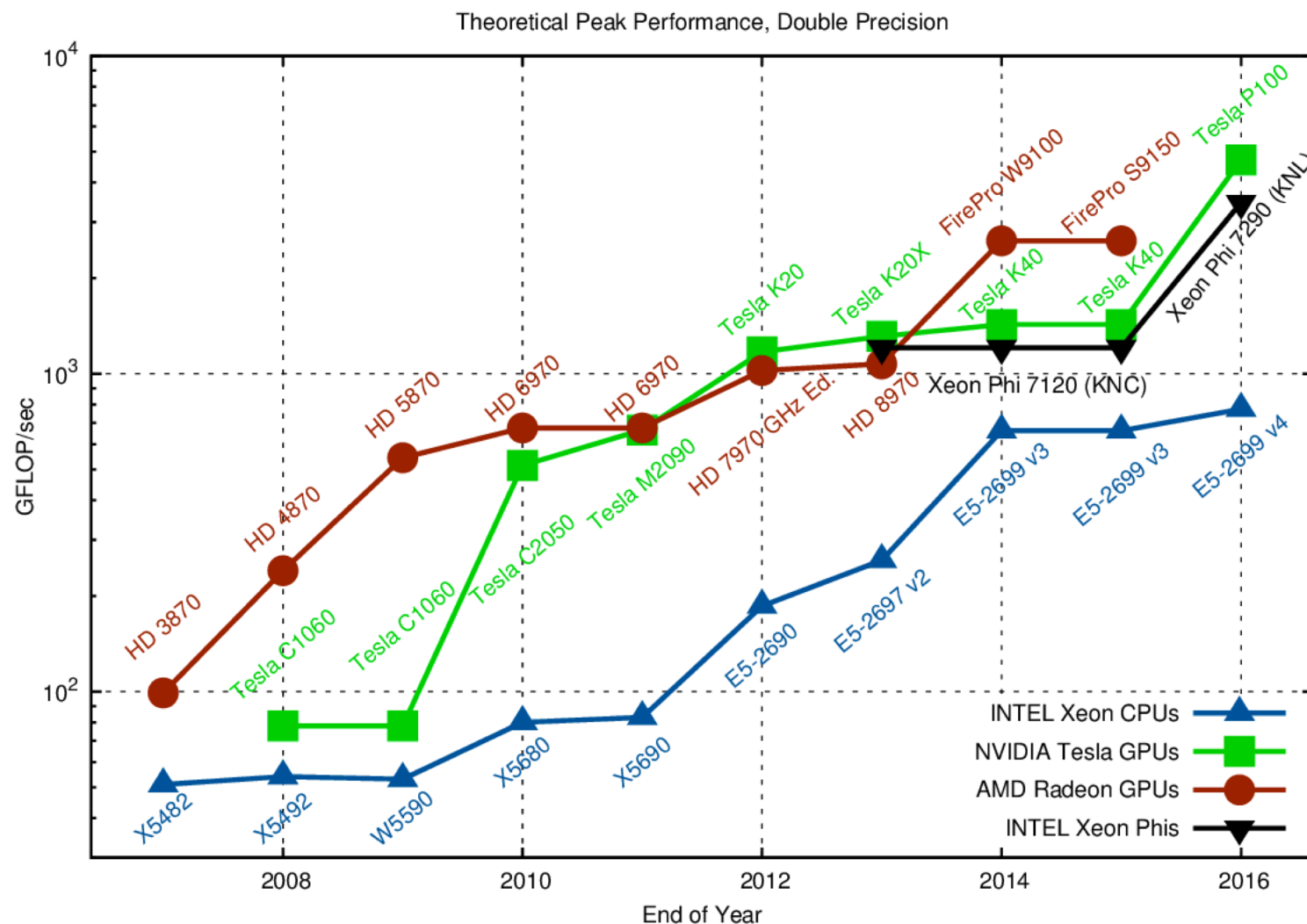
4 parallel multiplies

Performance

Architecture	Latency	Throughput
Haswell	5	0.5
Ivy Bridge	5	1
Sandy Bridge	5	1

2 instructions per clock cycle (CPI = 0.5)


Raw Double-Precision Throughput



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. addiu    $t1,$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)**
 addiu **\$t1,\$t1,#-32**
 bne **\$t1,\$t2,Loop**

NOTE:

1. Only 1 Loop Overhead every 4 iterations
2. This unrolling works if
 $\text{loop_limit} \pmod{4} = 0$
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)	
	addiu	\$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 $(n \bmod k) \neq 0$**

Then we will run the loop with 1 copy of the body **$(n \bmod k)$** times and with k copies of the body **$\text{floor}(n/k)$** 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 : **move** from mem to XMM register,
memory **aligned**, **packed** single precision

add ps : **add** from mem to XMM register,
packed single precision

mov a ps : **move** from XMM register to mem,
memory **aligned**, **packed** single 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
```

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

Intrinsics:

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:

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

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

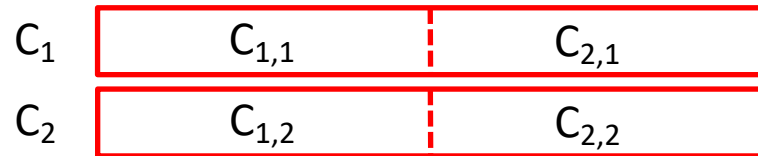
$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

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

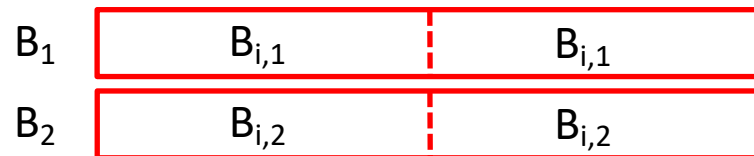
$$\begin{bmatrix} C_{1,1} = 1*1 + 0*2 = 1 & C_{1,2} = 1*3 + 0*4 = 3 \\ C_{2,1} = 0*1 + 1*2 = 2 & C_{2,2} = 0*3 + 1*4 = 4 \end{bmatrix}$$

Example: 2 x 2 Matrix Multiply

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



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

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

$C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$ $C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$
 $C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$ $C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$

- 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

- $l = 2$

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 (Part 1 of 2)

```
#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;
```

```
// 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;
}
```

DGEMM Speed Comparison

- Double precision general matrix matrix multiply: DGEMM
- Intel Core i7-5557U CPU @ 3.10 GHz
 - Instructions per clock (mul_pd) 2; Parallel multiplies per instruction 4
 - => 24.8 GFLOPS
- Python:

```
def dgemm(N, a, b, c):  
    for i in range(N):  
        for j in range(N):  
            c[i+j*N] = 0  
            for k in range(N):  
                c[i+j*N] += a[i+k*N] * b[k+j*N]
```

N	Python [Mflops]
32	5.4
160	5.5
480	5.4
960	5.3

- 1 MFLOP = 1 Million floating-point operations per second (fadd, fmul)
- **dgemm**(N ...) takes $2 \cdot N^3$ flops

C versus Python

- $c = a * b$
- a, b, c are $N \times N$ matrices

```
// Scalar; P&H p. 226
void dgemm_scalar(int N, double *a, double *b, double *c) {
    for (int i=0; i<N; i++)
        for (int j=0; j<N; j++) {
            double cij = 0;
            for (int k=0; k<N; k++)
                //      a[i][k] * b[k][j]
                cij += a[i+k*N] * b[k+j*N];
            // c[i][j]
            c[i+j*N] = cij;
        }
}
```

N	C [GFLOPS]	Python [GFLOPS]
32	1.30	0.0054
160	1.30	0.0055
480	1.32	0.0054
960	0.91	0.0053



240x!

Vectorized dgemm

```
// AVX intrinsics; P&H p. 227
void dgemm_avx(int N, double *a, double *b, double *c) {
    // avx operates on 4 doubles in parallel
    for (int i=0; i<N; i+=4) {
        for (int j=0; j<N; j++) {
            // c0 = c[i][j]
            __m256d c0 = {0,0,0,0};
            for (int k=0; k<N; k++) {
                c0 = _mm256_add_pd(
                    c0, // c0 += a[i][k] * b[k][j]
                    _mm256_mul_pd(
                        _mm256_load_pd(a+i+k*N),
                        _mm256_broadcast_sd(b+k+j*N)));
            }
            _mm256_store_pd(c+i+j*N, c0); // c[i,j] = c0
        }
    }
}
```



N	Gflops	
	scalar	avx
32	1.30	4.56
160	1.30	5.47
480	1.32	5.27
960	0.91	3.64

- 4x faster
- Still << theoretical 25 GFLOPS

Loop Unrolling

// Loop unrolling; P&H p. 352

```
const int UNROLL = 4;
```

```
void dgemm_unroll(int n, double *A, double *B, double *C) {  
    for (int i=0; i<n; i+= UNROLL*4) {  
        for (int j=0; j<n; j++) {  
            __m256d c[4];  4 registers  
            for (int x=0; x<UNROLL; x++)  
                c[x] = _mm256_load_pd(C+i+x*4+j*n);  
            for (int k=0; k<n; k++) {  
                __m256d b = _mm256_broadcast_sd(B+k+j*n);  
                for (int x=0; x<UNROLL; x++)  Compiler does the unrolling  
                    c[x] = _mm256_add_pd(c[x],  
                                         _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));  
            }  
            for (int x=0; x<UNROLL; x++)  
                _mm256_store_pd(C+i+x*4+j*n, c[x]);  
        }  
    }  
}
```

N	GFlops		
	scalar	avx	unroll
32	1.30	4.56	12.95
160	1.30	5.47	19.70
480	1.32	5.27	14.50
960	0.91	3.64	6.91 ?

FPU versus Memory Access

- How many floating-point operations does matrix multiply take?
 - $F = 2 \times N^3$ (N^3 multiplies, N^3 adds)
- How many memory load/stores?
 - $M = 3 \times N^2$ (for A, B, C)
- Many more floating-point operations than memory accesses
 - $q = F/M = 2/3 * N$
 - Good, since arithmetic is faster than memory access
 - Let's check the code ...

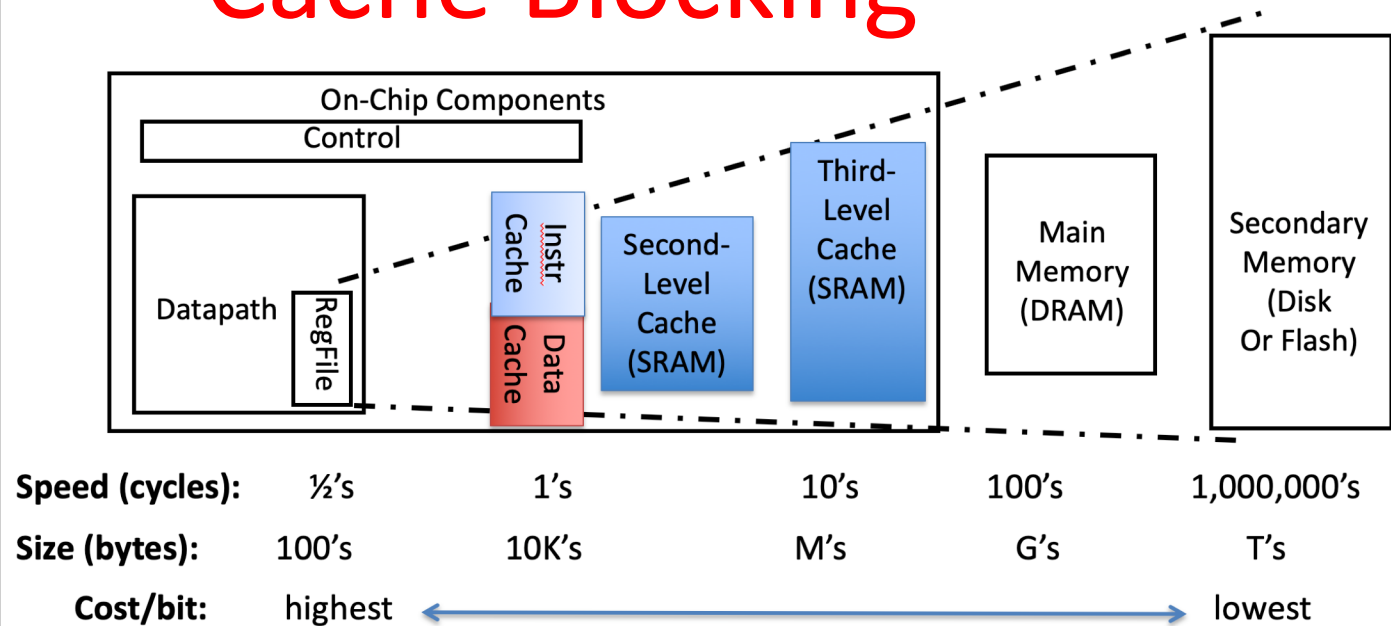
But memory is accessed repeatedly

- $q = F/M = 1.6!$ (1.25 loads and 2 floating-point operations)

Inner loop:

```
for (int k=0; k<N; k++) {  
    c0 = _mm256_add_pd(  
        c0, // c0 += a[i][k] * b[k][j]  
        _mm256_mul_pd(  
            _mm256_load_pd(a+i+k*N),  
            _mm256_broadcast_sd(b+k+j*N)));  
}
```

Cache Blocking



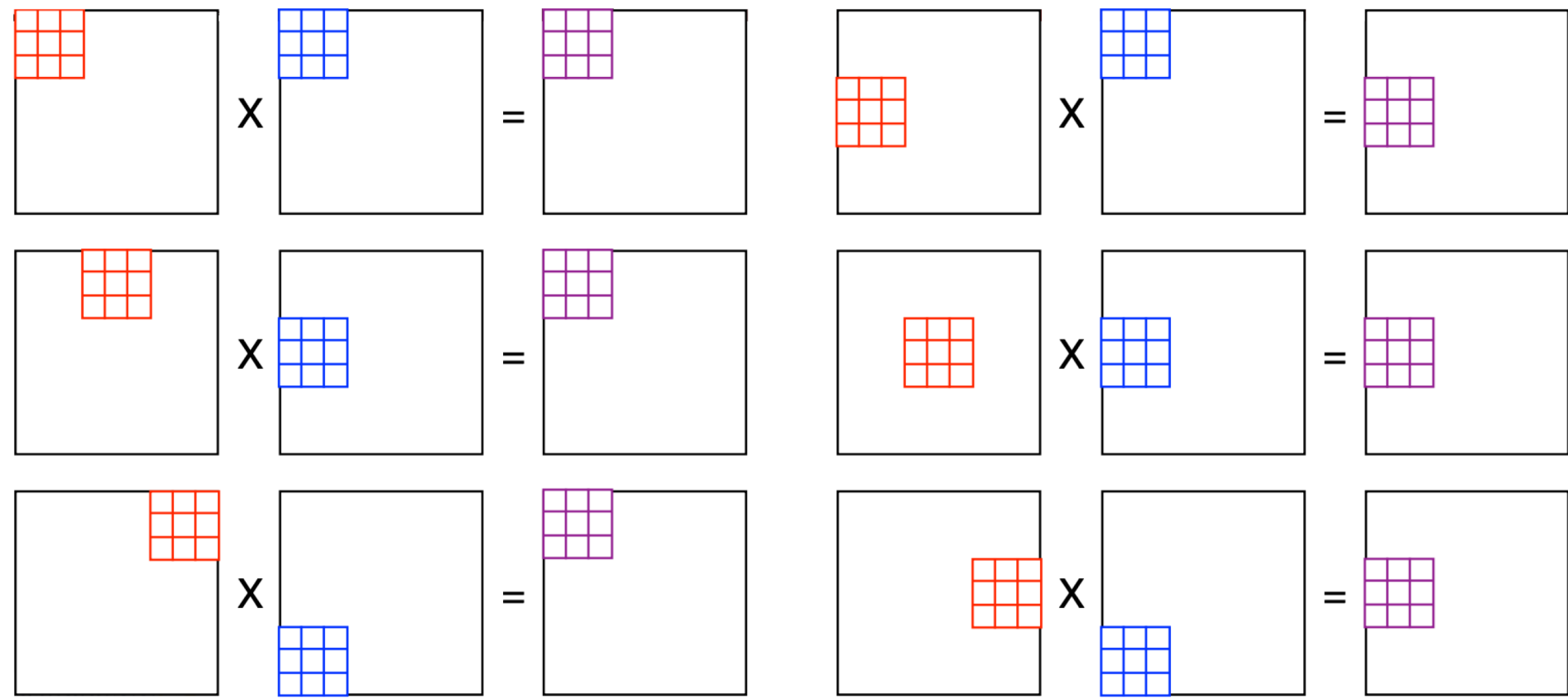
- Where are the operands (A, B, C) stored?
- What happens as N increases?
- Idea: arrange that most accesses are to fast cache!
- Rearrange code to use values loaded in cache many times
- Only “few” accesses to slow main memory (DRAM) per floating point operation

P&H, RISC-V edition p. 465

— -> throughput limited by FP hardware and cache, not slow DRAM

Blocking Matrix Multiply

(divide and conquer: sub-matrix multiplication)



Memory Access Blocking

```
// Cache blocking; P&H p. 556
const int BLOCKSIZE = 32;

void do_block(int n, int si, int sj, int sk, double *A, double *B, double *C) {
    for (int i=si; i<si+BLOCKSIZE; i+=UNROLL*4)
        for (int j=sj; j<sj+BLOCKSIZE; j++) {
            __m256d c[4];
            for (int x=0; x<UNROLL; x++)
                c[x] = _mm256_load_pd(C+i+x*4+j*n);
            for (int k=sk; k<sk+BLOCKSIZE; k++) {
                __m256d b = _mm256_broadcast_sd(B+k+j*n);
                for (int x=0; x<UNROLL; x++)
                    c[x] = _mm256_add_pd(c[x],
                                         _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));
            }
            for (int x=0; x<UNROLL; x++)
                _mm256_store_pd(C+i+x*4+j*n, c[x]);
        }
}

void dgemm_block(int n, double* A, double* B, double* C) {
    for(int sj=0; sj<n; sj+=BLOCKSIZE)
        for(int si=0; si<n; si+=BLOCKSIZE)
            for (int sk=0; sk<n; sk += BLOCKSIZE)
                do_block(n, si, sj, sk, A, B, C);
}
```

Performance

- Intel i7-5557U theoretical limit (AVX2): 24.8 GFLOPS
- Cache:
 - L3: 4 MB 16-way set associative shared cache
 - L2: 2 x 256 KB 8-way set associative caches
 - L1 Cache: 2 x 32KB 8-way set associative caches (2x: D & I)
- Maximum memory bandwidth (GB/s): 29.9

N	Size	GFlops			
		scalar	avx	unroll	blocking
32	3x 8KB	1.30	4.56	12.95	13.80
160	3x 205KB	1.30	5.47	19.70	21.79
480	3x 1.8MB	1.32	5.27	14.50	20.17
960	3x 7.3MB	0.91	3.64	6.91	15.82

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