CS 110
Computer Architecture
Paralellism, Amdahl's Law

Instructors:
Chundong Wang, Siting Liu & Yuan Xiao
Course website: https://toast-lab.sist.shanghaitech.edu.cn/courses/CS110@ShanghaiTech/Spring-2025/index.html
School of Information Science and Technology (SIST)
ShanghaiTech University

2025/4/29

Administratives

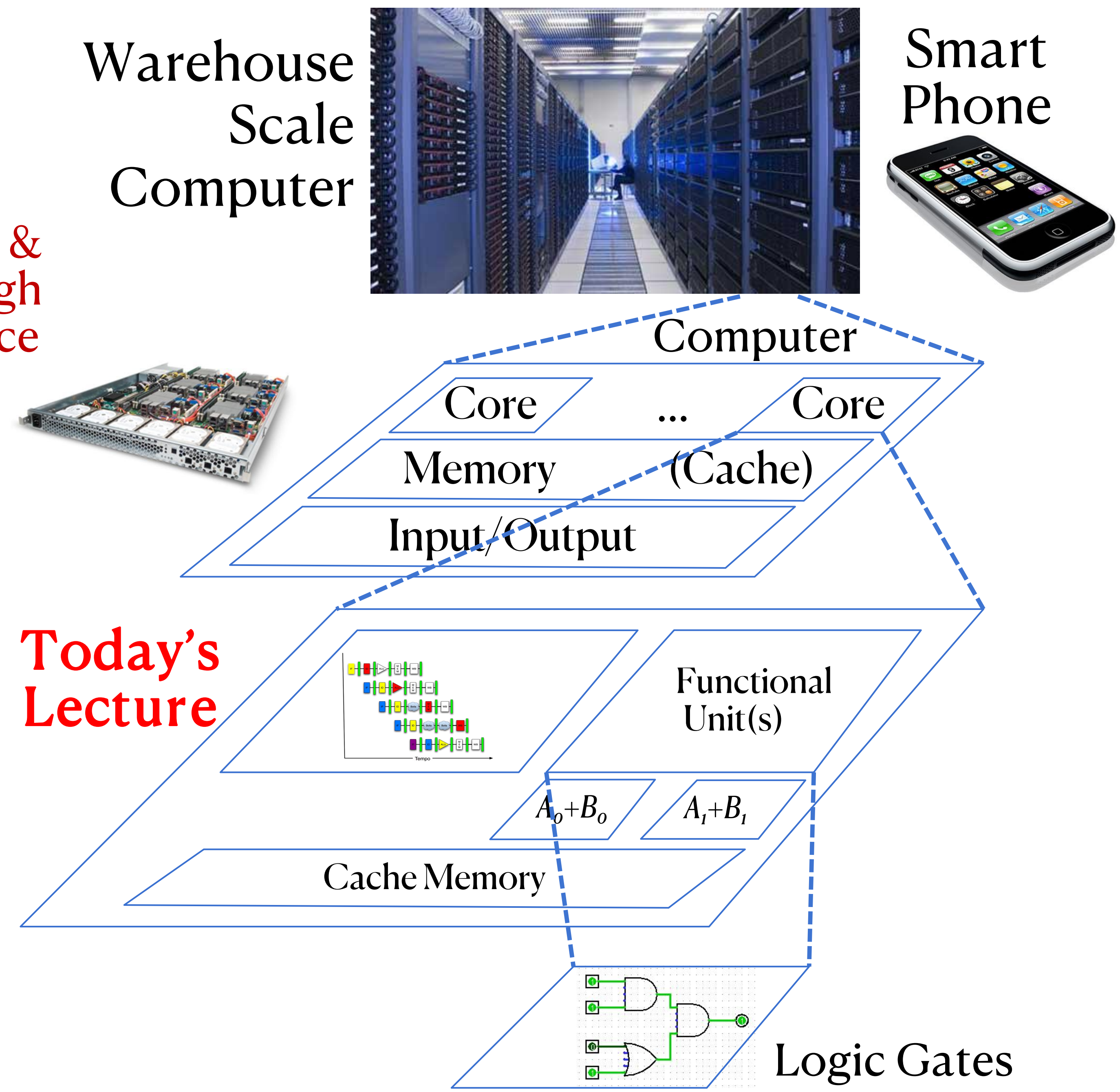
- Mid-term II tentatively May 15th 8am-10am; you can bring 2-page A4-sized double-sided cheat sheet, **handwritten** only! (Teaching center 201/301/303); From start to May 13th lecture.
- Project 2.1 ddl approaching, May 5th!!!
- Project 2.2 released, ddl May 19th.
- HW5 ddl approaching, May 7th.
- HW6 will be released, ddl May 12th!
- Lab 11 to be released. Prepare in advance!
  - Keep the boards really well, because you have to return the board after lab/project checking;
- Discussion May 9th & 12th on SIMD.

Parallelism Overview

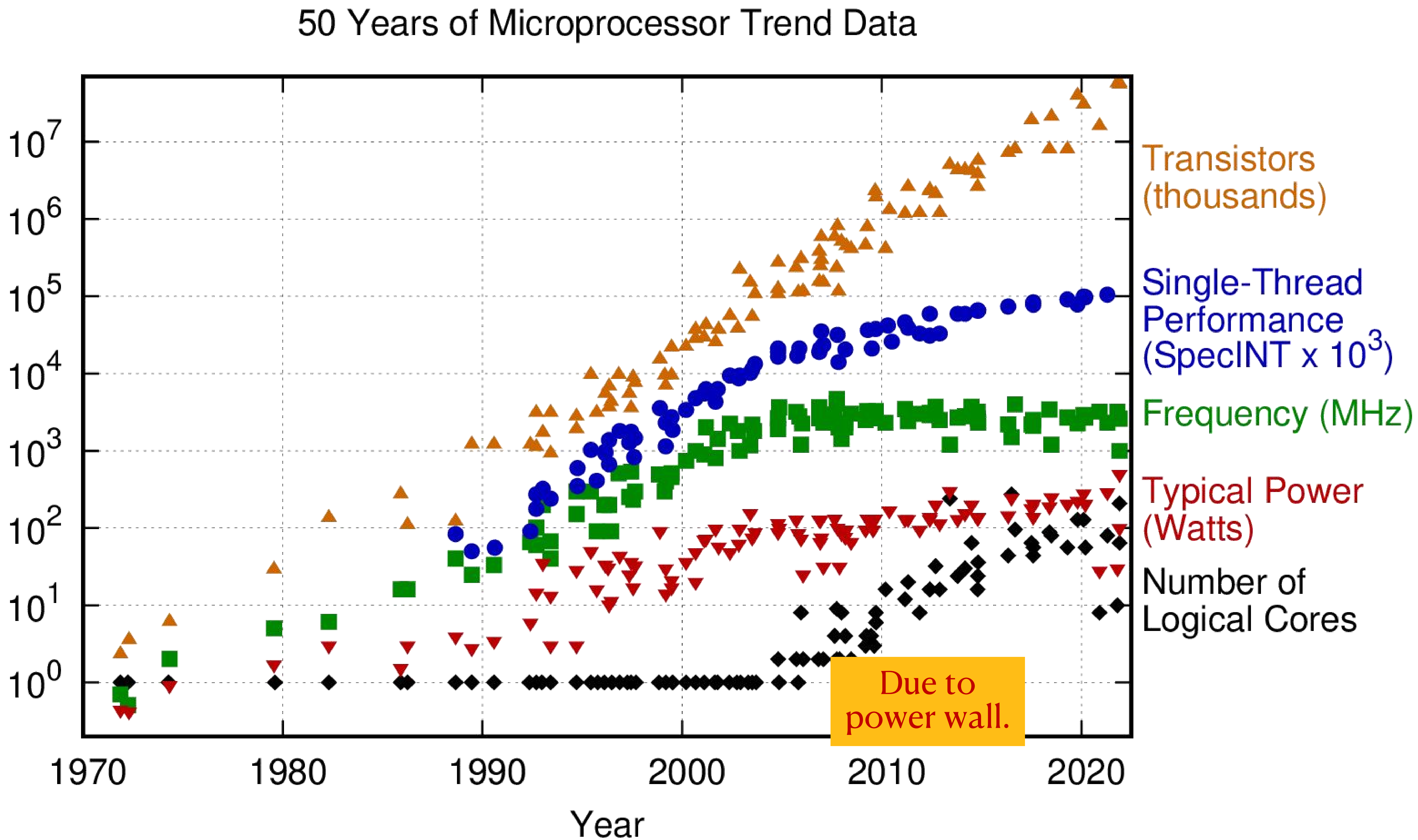
Software | *Hardware*

- Parallel Requests
  Assigned to computer
  e.g., Search “CS110”
- 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



CPU Trends

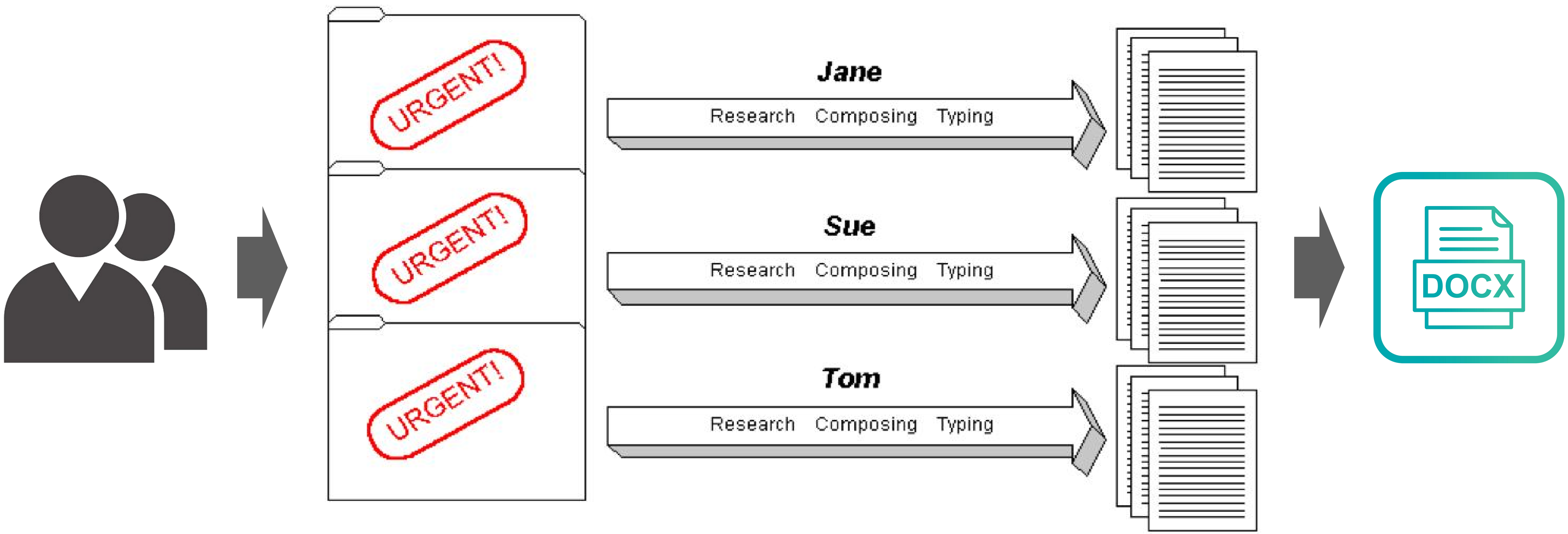


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp

Using Parallelism for Performance

- Two basic ways:
  - Multiprogramming
    - Run multiple independent programs in parallel
    - “Easy”
  - Parallel computing (parallel processing program)
    - Run one program (single task or job) faster
    - “Hard”
  - We’ll focus on parallel computing for next few lectures

Challenges for Parallel Processing Programs



Task partition Task sceduling Communication between writers … …

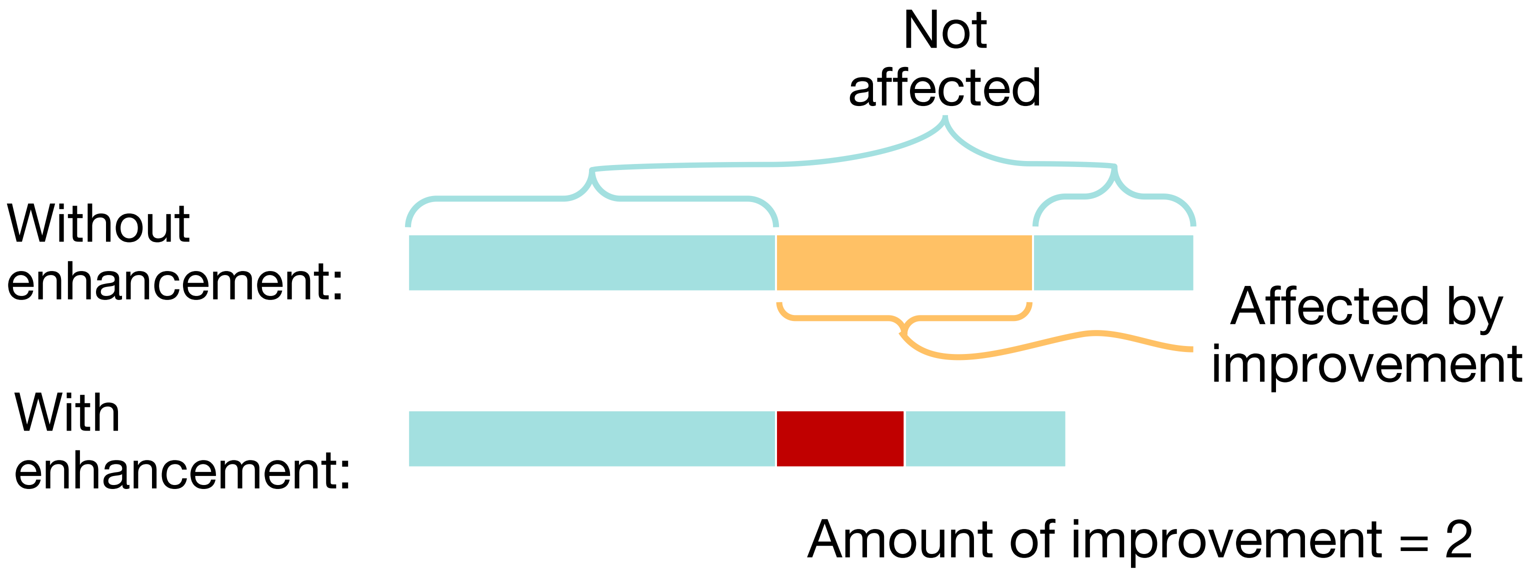
Recall Amdahl's Law

Gene Amdahl
Computer Pioneer

- We have learnt in the very first lecture.

Execution time after improvement

$$= \frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}} + \text{Execution time not affected}$$

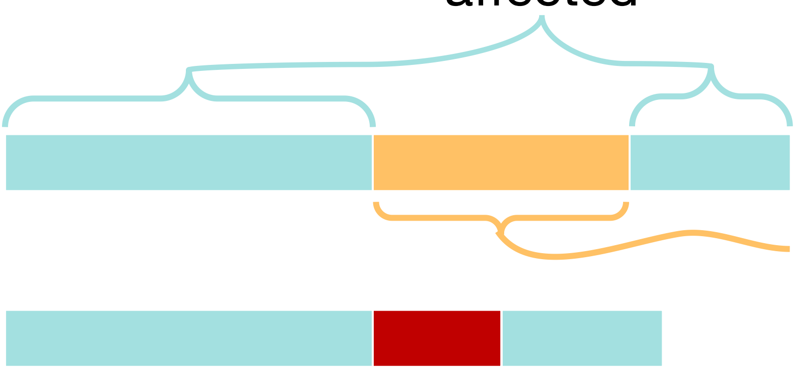


Speed-up Challenges

Gene Amdahl
Computer Pioneer

Speed-up

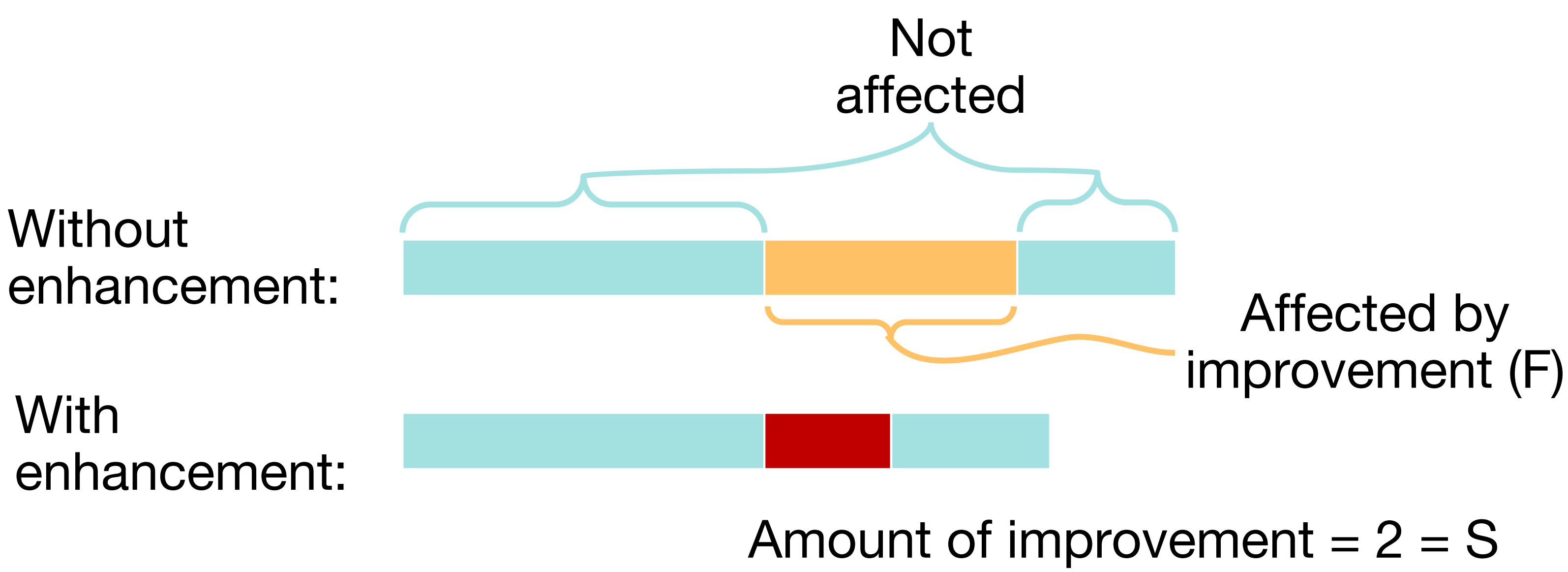
$$= \frac{\text{Original execution time}}{\text{Execution time after improvement}} = \frac{1}{(1-F) + \frac{F}{S}}$$



Amdahl's Law Example

- Suppose to achieve a speed-up of 90 times faster with 100 processors, what percentage of the original computation can be sequential?

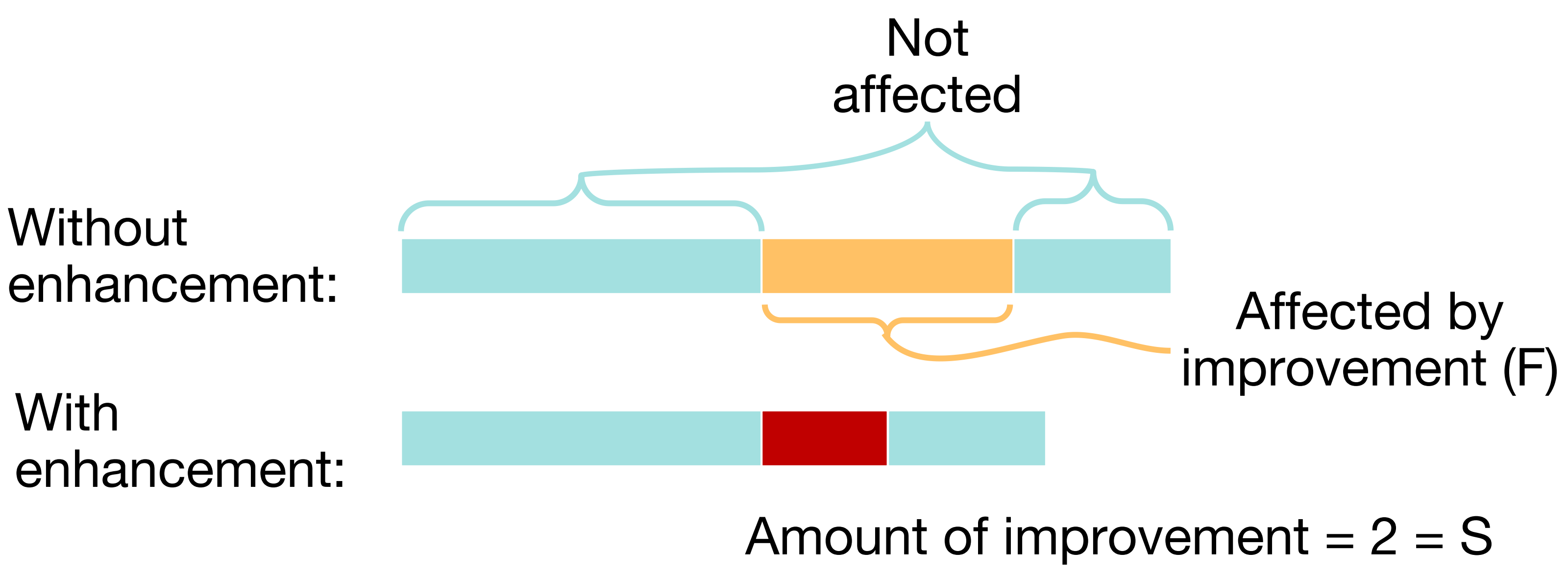
$$\text{Speed-up} = \frac{1}{(1-F) + \dfrac{F}{S}}$$



Amdahl's Law Example

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

$$\text{Speed-up} = \frac{1}{(1-F) + \dfrac{F}{S}}$$



Another Example

- Assume that we perform two sums: one to sum 10 scalar variables, and one to add two-dimensional arrays (element-wise), with dimensions 10 by 10. Assume an addition takes time t.

  Single processor execution time: 110 * t

  10 processors execution time: 20 * t

  50 processors execution time: 12 * t

Another Example

- Assume that we perform two sums: one to sum 10 scalar variables, and one to add two-dimensional arrays (element-wise), with dimensions 10 by 10. Assume an addition takes time t.

  Single processor execution time: 110 * t

  10 processors execution time: 20 * t

  50 processors execution time: 12 * t

- What if it is a 20 by 20 matrix addition?

  Single processor execution time: 410 * t

  10 processors execution time: 50 * t

  50 processors execution time: 18 * t

Strong and Weak Scaling

- It is harder to obtain good speed-up while keeping the problem size fixed than to obtain good speed-up by increasing the size of 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
- Memory hierarcy also interfere with scaling;
  - e.g. when problem does not fit in last level cache for weakly scaled data

Gustafson's law

A post on scaling

Load Balancing

- Assume that we perform add two-dimensional arrays (element-wise), with dimensions 10 by 10. Assume an addition takes time t.

- Case 1: balanced load for 10 processors on matrix addition

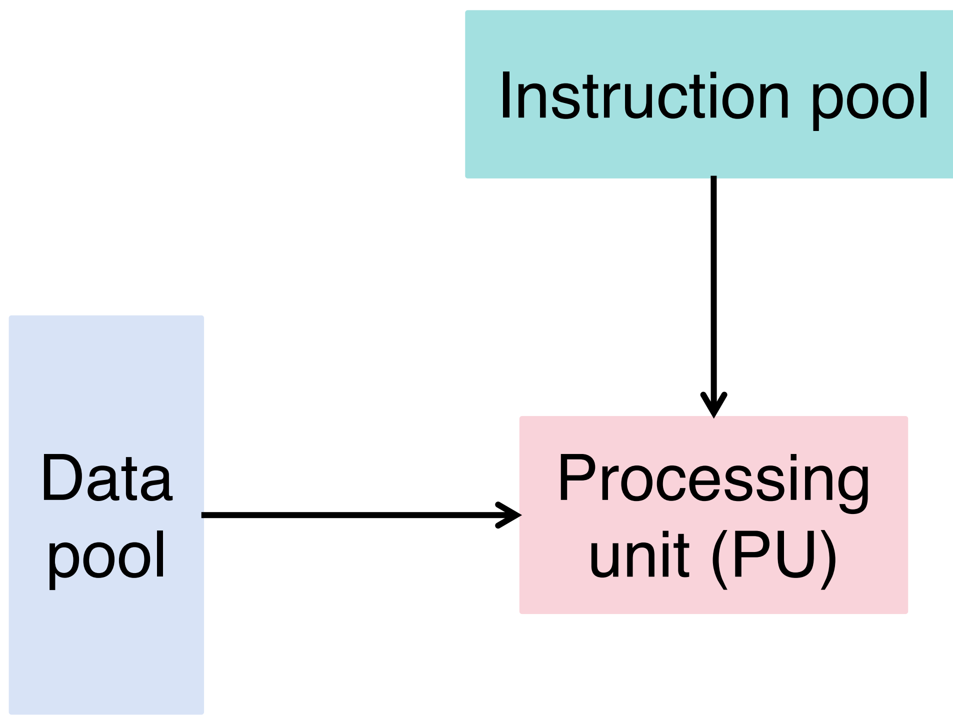
- Case 2: 5 processors take 100%, while the other 5 processors take 0% on matrix addition

10 * t

20 * t

Flynn’s Taxonomy

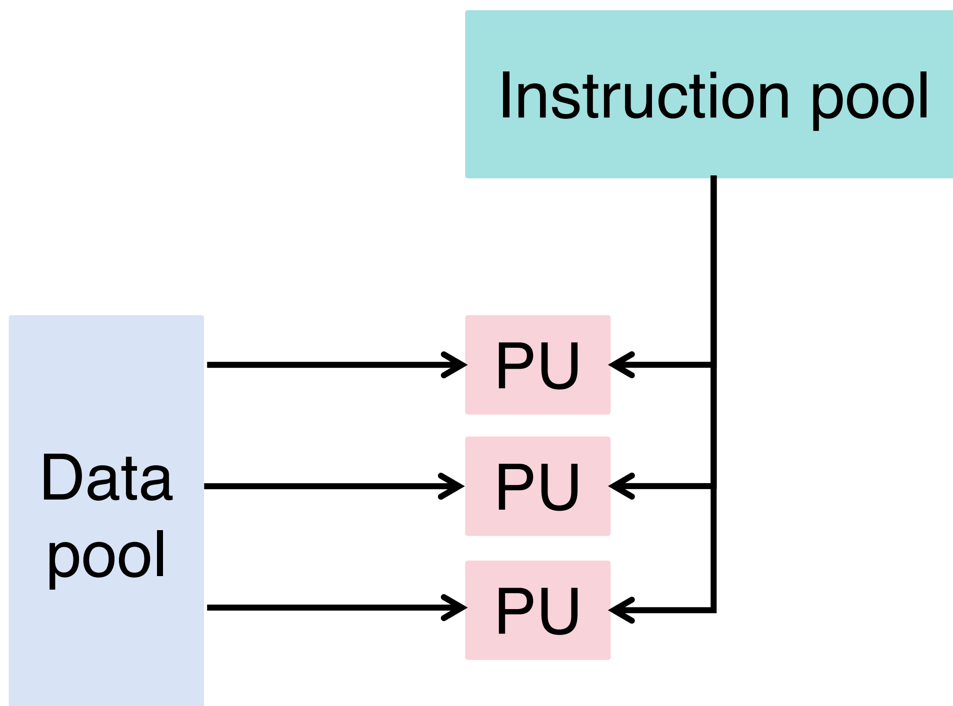
- Sequential computer that exploits no parallelism in either the instruction or data streams. Examples of SISD architecture are traditional uniprocessor machines.
  - E.g. Our RISC-V processor up to now;
  - Superscalar is SISD because programming model is sequential



Single instruction, single data (SISD)

Single Instruction, Multiple Data (SIMD)

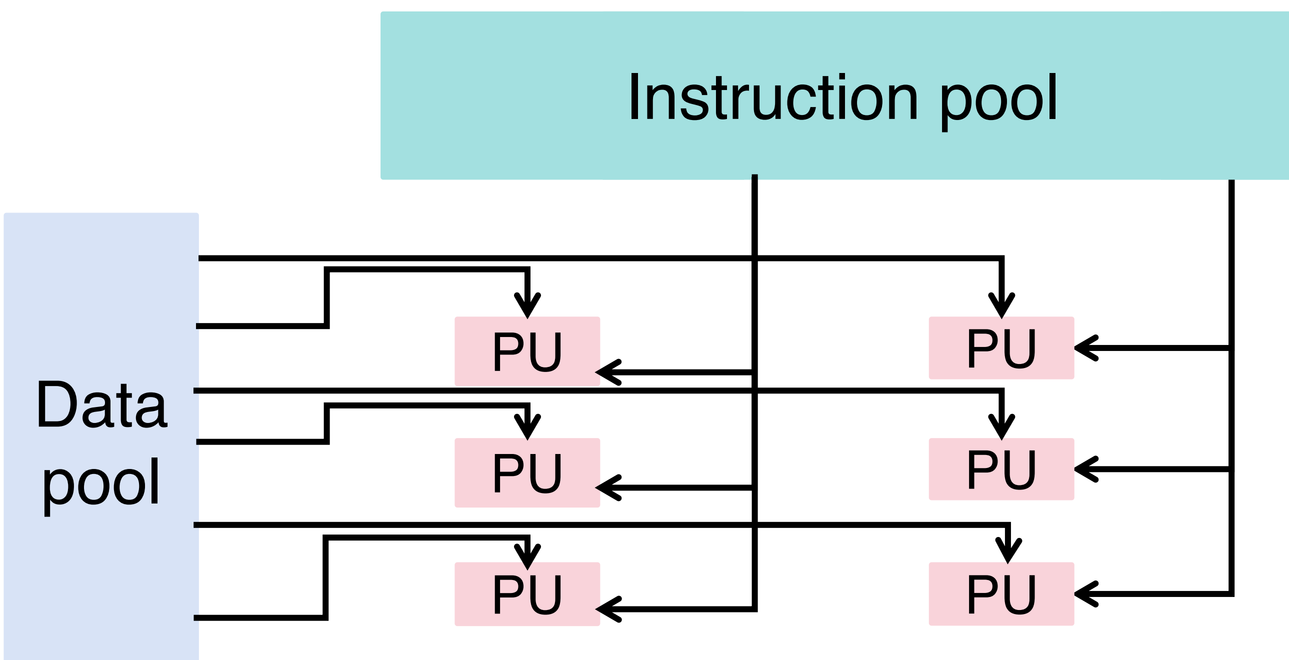
- SIMD computer exploits multiple data streams against a single instruction stream to operations that may be naturally parallelized.
  - Intel SIMD instruction extensions
  - NVIDIA Graphics Processing Unit (GPU)
  - Vector processors



SIMD pronouced as “sim-dee”.
Main topic today!

Multiple Instruction, Multiple Data (MIMD)

- Multiple autonomous processors simultaneously executing different instructions on different data.
  - Multicore
  - Warehouse-scale computers (WSC)

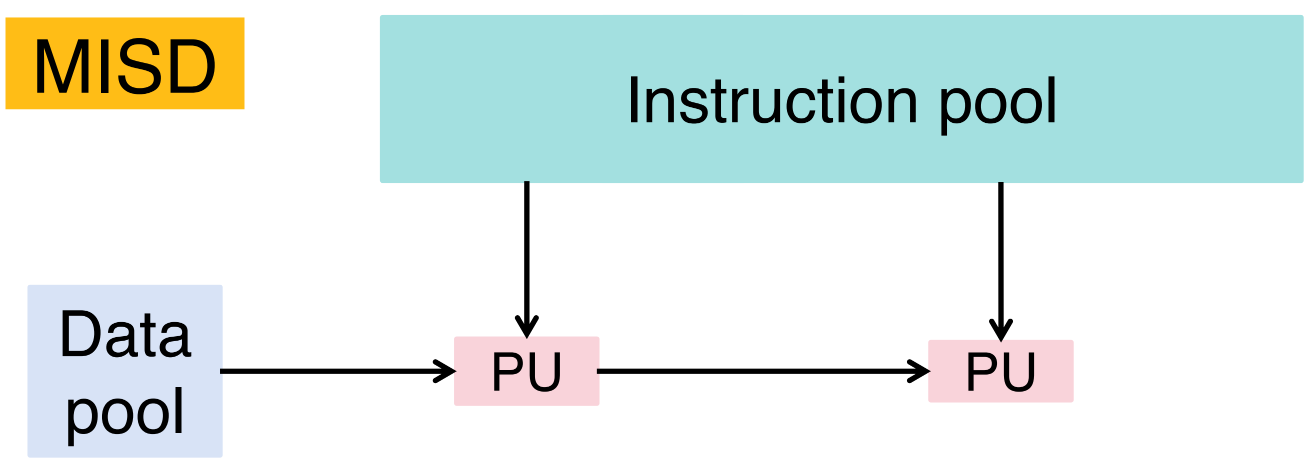


MIMD pronouced as “mim-dee”.
Will be covered in later lectures!

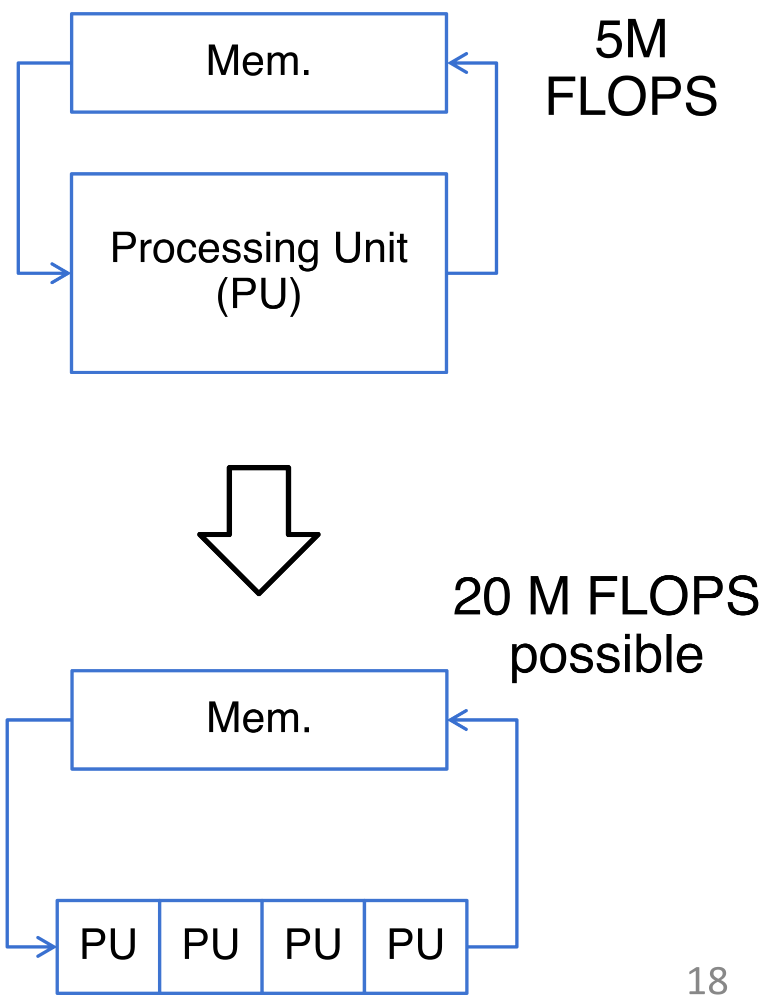
Apple M4 [Tech Insights]

Multiple Instruction, Single Data (MISD)

- Multiple-Instruction, Single-Data stream computer that exploits multiple instruction streams against a single data stream.
  - Rare, mainly of historical interest only
  - Some literatures categorize systolic array as MISD



H. T. Kung, Why systolic Architectures? IEEE Computer, 1982



Flynn’s Taxonomy, 1966

- 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 (a.k.a. hw-level *data parallelism*): specialized function units, for handling lock-step calculations involving arrays
  - Scientific computing, signal processing, multimedia (audio/video processing)

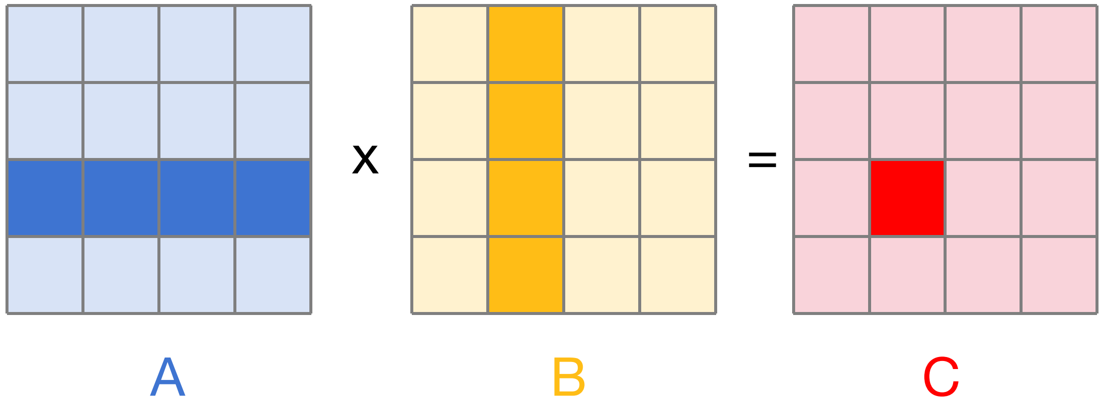
		Data streams	
		Single	Multiple
Instruction streams	Single	SISD: Intel Pentium 4	SIMD: SSE instructions of x86
	Multiple	MISD: No examples today	MIMD: Intel i7/Apple M4, etc.

Data-Level Parallelism (DLP)

- Executing the same operation on multiple data streams
- Example: element-wise vector multiplication (e.g., in filtering, GEMM, etc.)

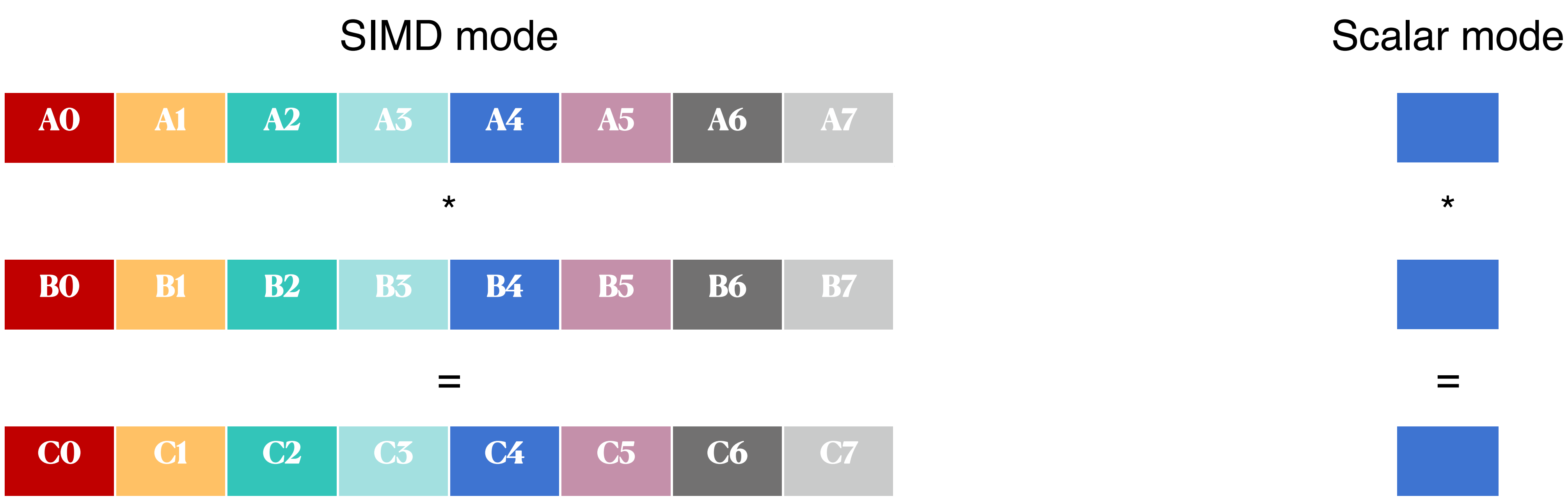
$$y[i] := c[i] \times x[i],\ 0 \le i < n$$

```
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++) {
        Cij += A[i+k*N] * B[k+j*N];
      }
      C[i+j*N] = Cij;
    }
  }
}
```



SIMD Architecture

- SIMD architectures provide performance improvement for DLP
  - One instruction is fetched & decoded for entire operation
  - Multiplications are known to be independent
  - Pipelining/concurrency in memory access as well
  - Special functional units may be faster



SIMD Applications & Implementations

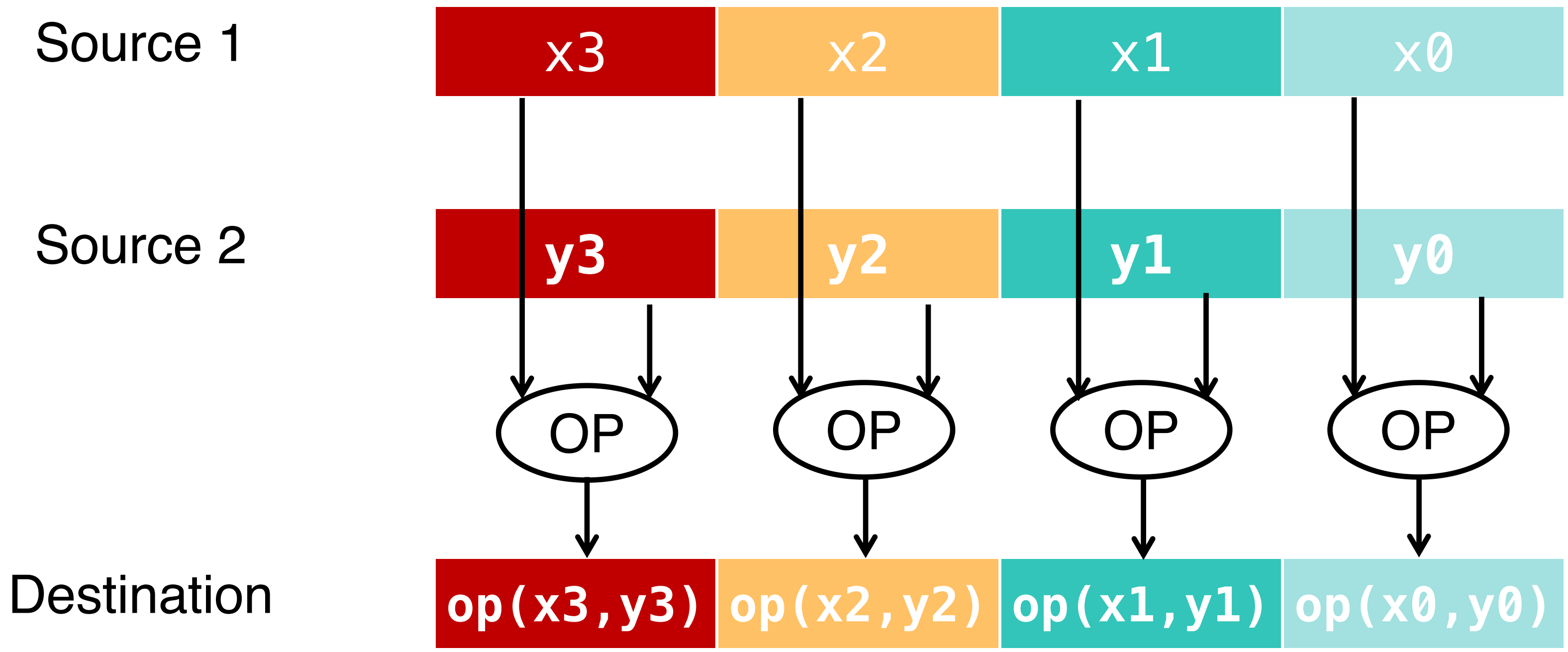
- Applications
  - Scientific computing (Matlab, NumPy)
  - Graphics and video processing (Photoshop, …)
  - Big Data (Deep learning)
  - Gaming

- Implementations
  - x86 Intel Intrinsics
  - ARM
  - RISC-V vector extensions
    - More in CA II & EE219
  - Video cards

SIMD instructions can often be accessed via extensions to a given ISA, e.g., Intel x86 SSE/AVX, RISC-V vector extension.

SIMD Instructions

- Fetch one instruction, do the work of multiple instructions



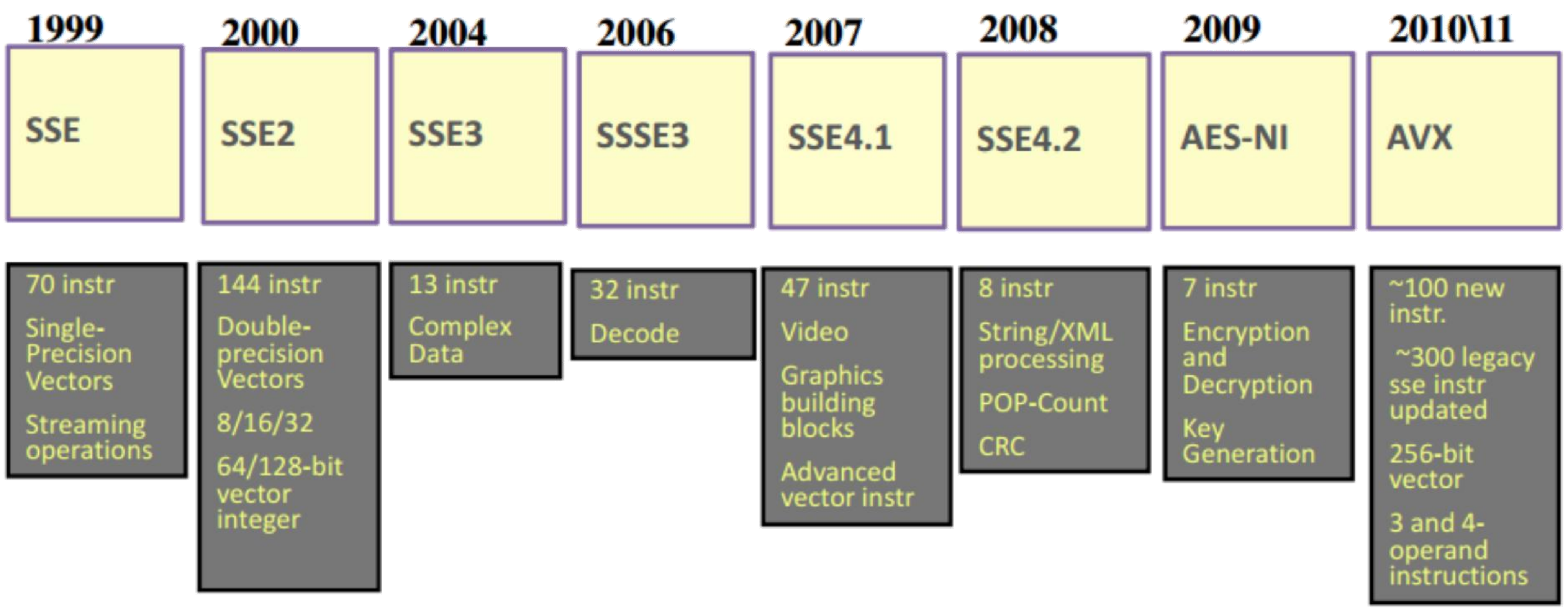
  - Source operands + destination registers wide enough to fit multiple values (e.g., four 64-bit floating point numbers)
  - Apply single operation (e.g., OP:= add) to all operands in register

Intel SIMD Instructions

- Multi-Media eXtension (MMX)
- Streaming SIMD Extension (SSE)
- Advanced Vector eXtension (AVX)

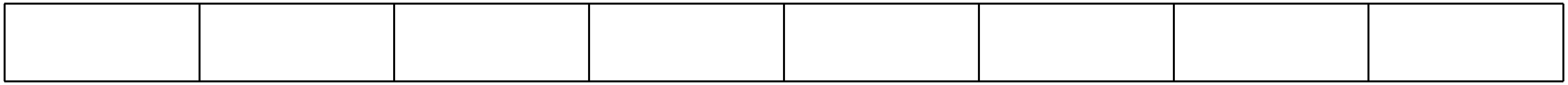
MMX 1997

MMX 64-bit registers, reusing floating-point registers



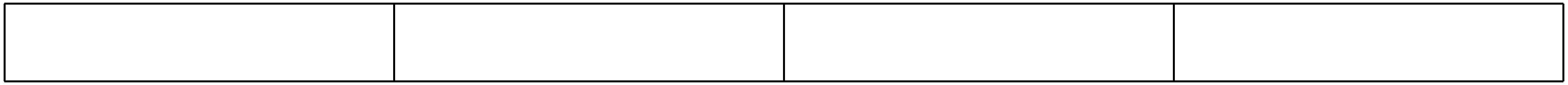
MMX Datatype

Packed byte



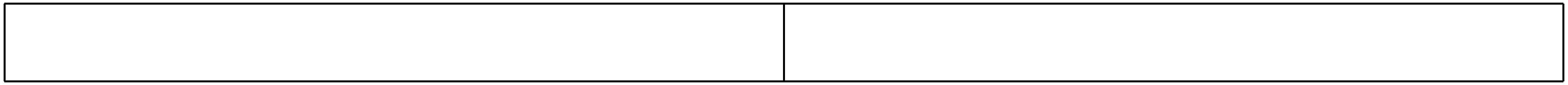
63 8 7 0

Packed word (Intel has 16-bit words)



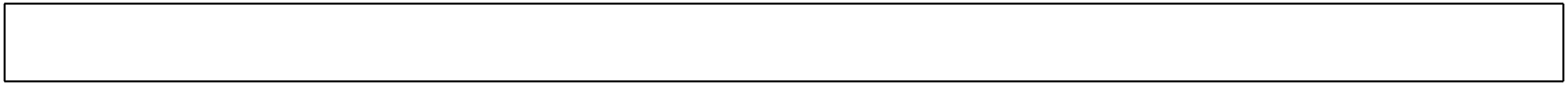
63 16 15 0

Packed doubleword



63 32 31 0

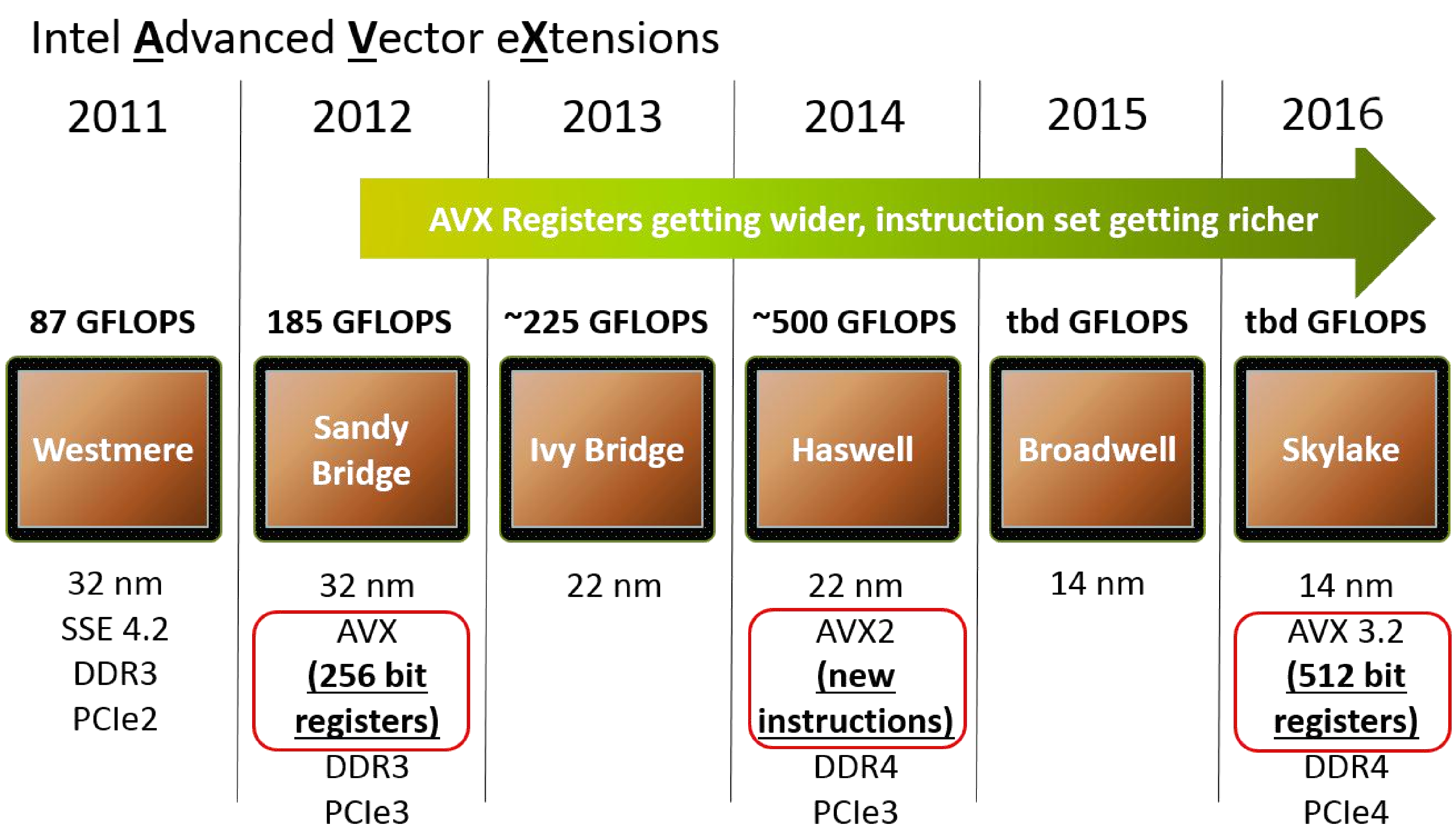
Packed quadword



63 0

Intel SIMD Instructions

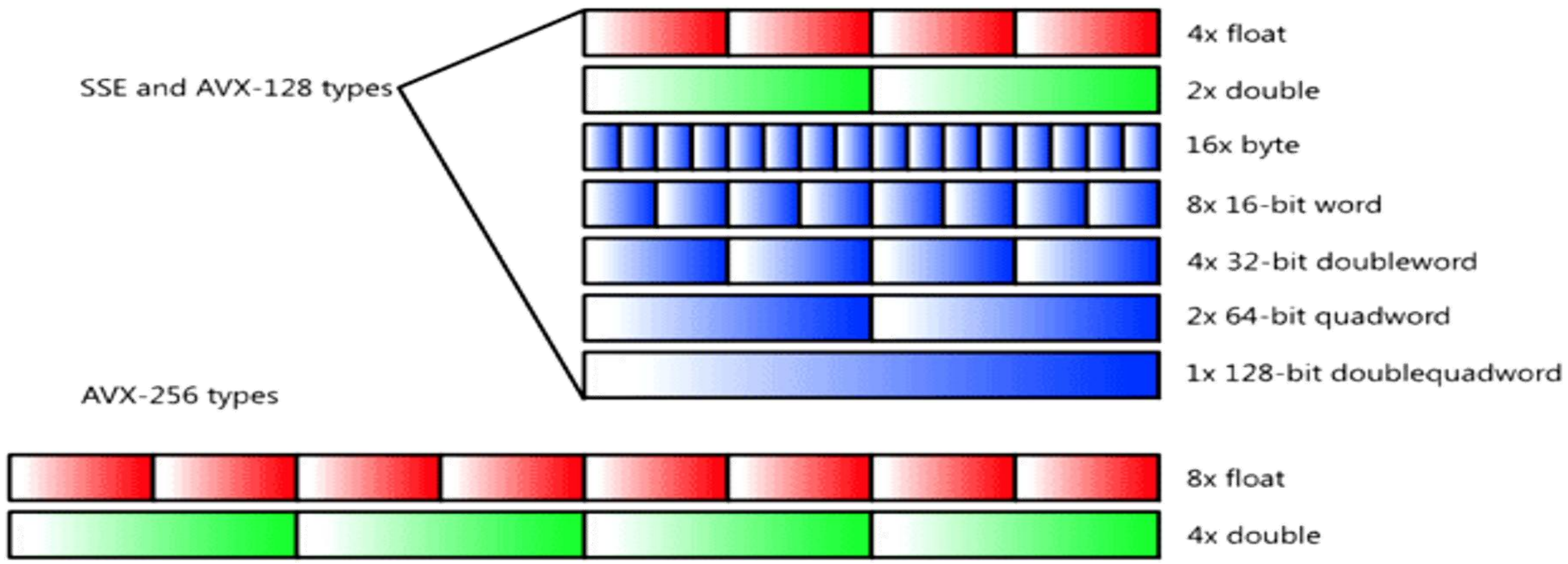
- Multi-Media eXtension (MMX)
- Streaming SIMD Extension (SSE)
- Advanced Vector eXtension (AVX)



https://chrisadkin.io/2015/06/04/under-the-hood-of-the-batch-engine-simd-with-sql-server-2016-ctp/

Intel SSE SIMD Data Types

- Multi-Media 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

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

X86 SIMD Intrinsics

Instruction Set

- ☐ MMX
- ☐ SSE family
- ☐ AVX family
- ☐ AVX-512 family
- ☐ AMX family
- ☐ SVML
- ☐ Other

Categories

- ☐ Application-Targeted
- ☐ Arithmetic
- ☐ Bit Manipulation
- ☐ Cast
- ☐ Compare
- ☐ Convert
- ☐ Cryptography
- ☐ Elementary Math Functions
- ☐ General Support
- ☐ Load
- ☐ Logical
- ☐ Mask
- ☐ Miscellaneous
- ☐ Move
- ☐ OS-Targeted
- ☐ Probability/Statistics
- ☐ Random

Search Intel Intrinsics

```
__m256d _mm256_add_pd (__m256d a, __m256d b)
```

Synopsis

```
__m256d _mm256_add_pd (__m256d a, __m256d b)
#include <immintrin.h>
Instruction: vaddpd ymm, ymm, ymm
CPUID Flags: AVX
```

Intrinsic

Instruction

Description

Add 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 additions

Latency and Throughput

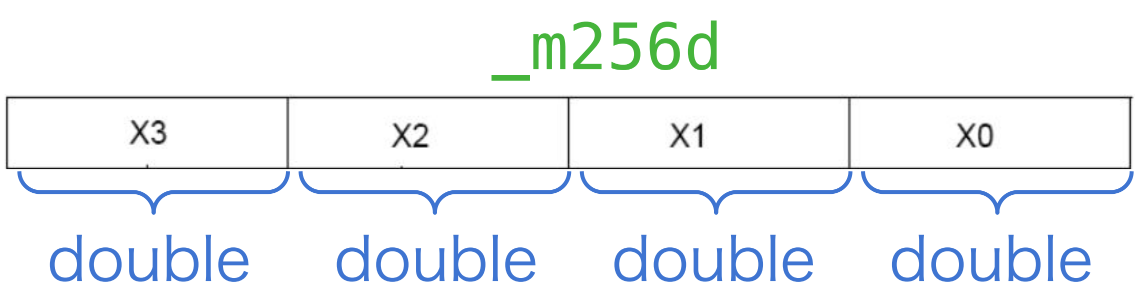
Architecture	Latency	Throughput (CPI)
Alderlake	2	0.5
Icelake Intel Core	4	0.5
Icelake Xeon	4	0.5
Sapphire Rapids	2	0.5
Skylake	4	0.5

CPI = 0.5

https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

SIMD Instructions and Intrinsic Functions

- **SIMD instructions** can often be accessed via **extensions** to a given ISA, e.g., Intel x86.
  - Sometimes known as vector instructions
- Use specialized "vector" registers
- Use extended SIMD instructions to load/store, do arithmetic, etc.



Instead of writing assembly, use **intrinsics** to write in a higher-level language, C.

- **Intrinsics are C functions and procedures that provide access to assembly language.**
  - With intrinsics, can program using assembly instructions indirectly.
  - One-to-one correspondence between SIMD extension assembly instruction (e.g., Intel AVX or SSE) and intrinsics.

Example: SIMD Array Processing

- For each f in array perform: `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

Loop Unrolling

- 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** DLP than available in a **single** iteration of a loop?
- *Unroll loop* and adjust iteration rate

Looping in RISC-V

- D Standard Extension (double) – builds upon F standard extension (float)

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: fld     f2 , 0(t1)      # $f2=array element
      fadd.d f10, f2, f0      # add s to $f2
      fsd     f10, 0(t1)      # store result
      addi    t1, t1, -8      # t1 = t1 -8
      bne     t1, t2, Loop    # repeat loop if t1 != t2
```

Loop Unrolled

- 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

4 Loads side-by-side:
Could replace with 4-wide SIMD Load

```
Loop:
        fld     f2 , 0(t1)
        fld     f3 , -8(t1)
        fld     f4 , -16(t1)
        fld     f5 , -24(t1)

        fadd.d f10, f2, f0
        fadd.d f11, f3, f0
        fadd.d f12, f4, f0
        fadd.d f13, f5, f0

        fsd     f10, 0(t1)
        fsd     f11, -8(t3)
        fsd     f12, -16(t1)
        fsd     f13, -24(t1)

        addi    t1, t1, -32
        bne     t1, t2, Loop
```

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 Unrolled
Scheduled

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

```
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

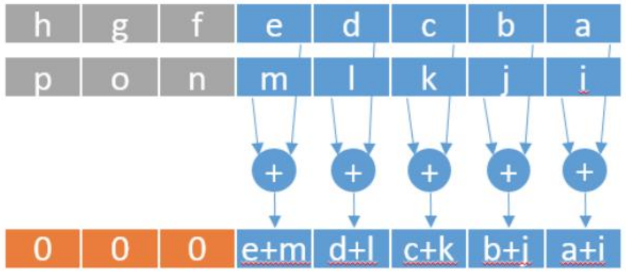
RISC-V Vector Extension

- 32 vector registers
- Need to setup length of data and number of parallel registers to work on before usage (`vconfig`)! `vflw.s`: vector float load word . stride: load a single word, put in `v1` ‘vector length’ times
- `vsetvl`: ask for certain vector length – hardware knows what it can do (maxvl)!

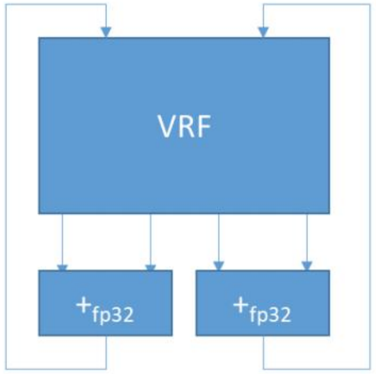
```
    # assume x1 contains size of array
    # assume t1 contains address of array
    # assume x4 contains address of scalar s
    vconfig 0x63        # 4 vregs, 32b data (float)
    vflw.s v1.s, 0(x4)  # load scalar value into v1

loop:
    vsetvl x2, x1       # will set vl and x2 both to min(maxvl, x1)
    vflw v0, 0(t1)      # will load ‘vl’ elements out of ‘vec’
    vfadd.s v2, v1, v0  # do the add
    vsw v2, 0(t1)       # store result back to ‘vec’
    slli x5, x2, 2      # bytes consumed from ‘vec’ (x2 * sizeof(float))
    add t1, t1, x5      # increment ‘vec’ pointer
    sub x1, x1, x2      # subtract from total (x1) work done this iteration (x2)
    bne x1, x0, loop    # if x1 not yet zero, still work to do
```

Hardware Support for CPU

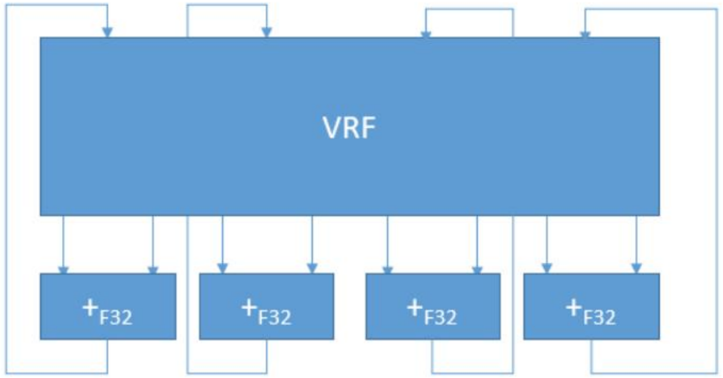


2-lane implementation



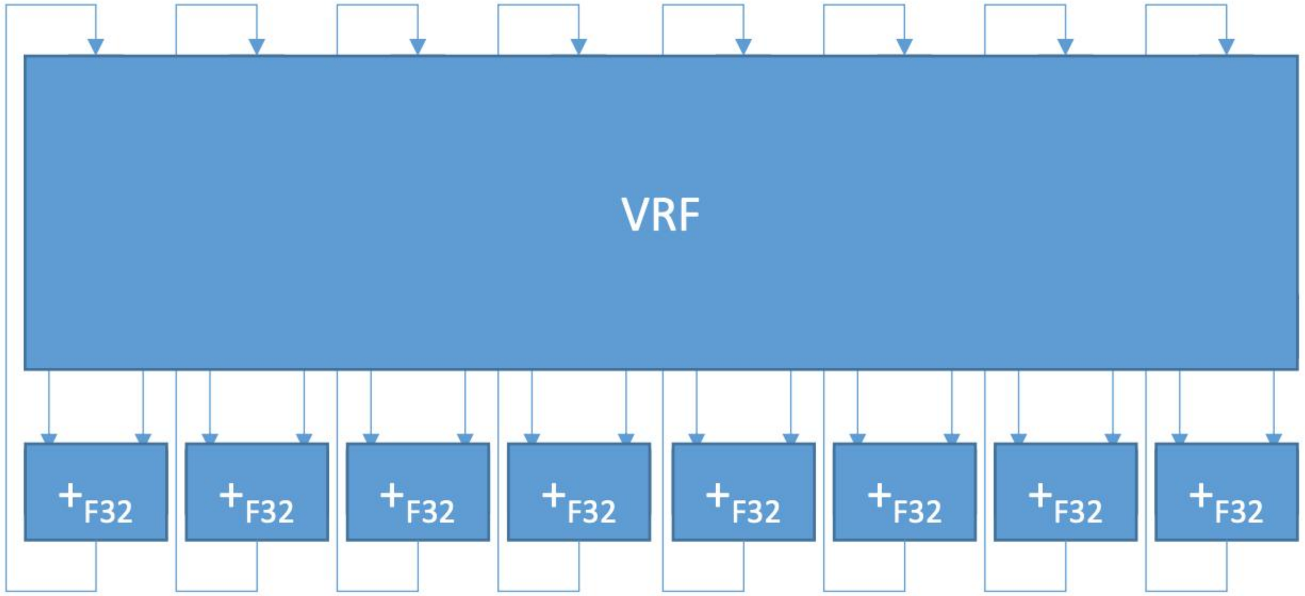
1st clock: a+i, b+j
2nd clock: c+k, d+l
3rd clock: e+m, 0
4th clock: up to you

4-lane implementation



1st clock: a+i, b+j, c+k, d+l
2nd clock: e+m, 0, 0, 0

8-lane implementation (a.k.a. SIMD)



1st clock: a+i, b+j, c+k, d+l, e+m, 0, 0, 0

Number of lanes is transparent to programmer
Same code runs independent of # of lanes

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;
```

- SSE Instruction Sequence:

(Note: Destination on the right in x86 assembly)

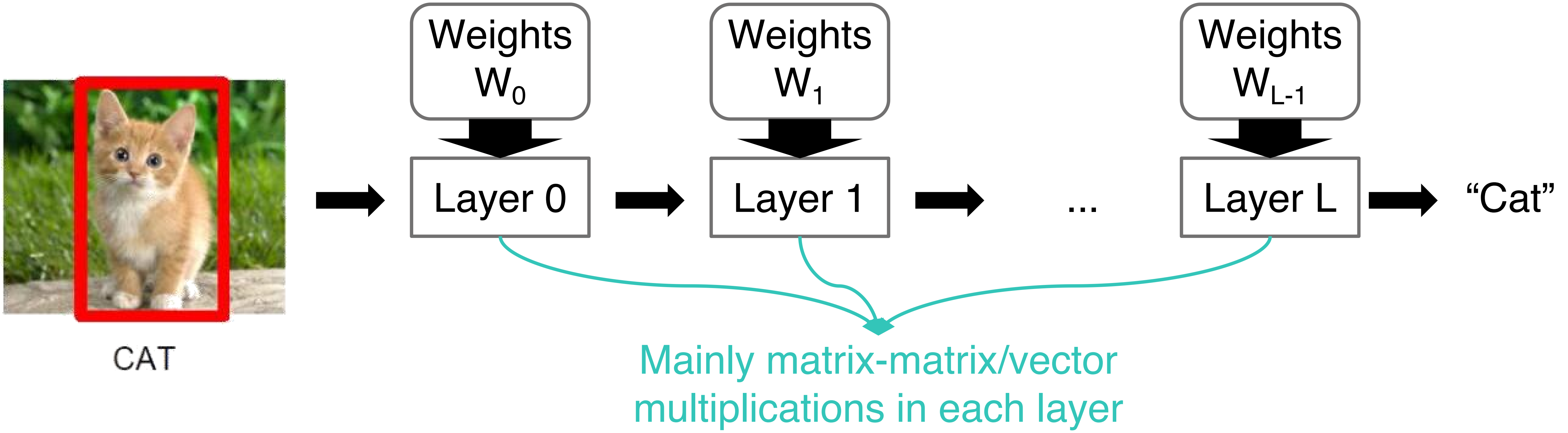
```
movaps address-of-v1, %xmm0
// v1.w | v1.z | v1.y | v1.x -> xmm0
addps address-of-v2, %xmm0
// v1.w+v2.w | v1.z+v2.z | v1.y+v2.y | v1.x+v2.x -> xmm0
movaps %xmm0, address-of-vec_res
```

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

Matrix Multiplication Performance Benchmark

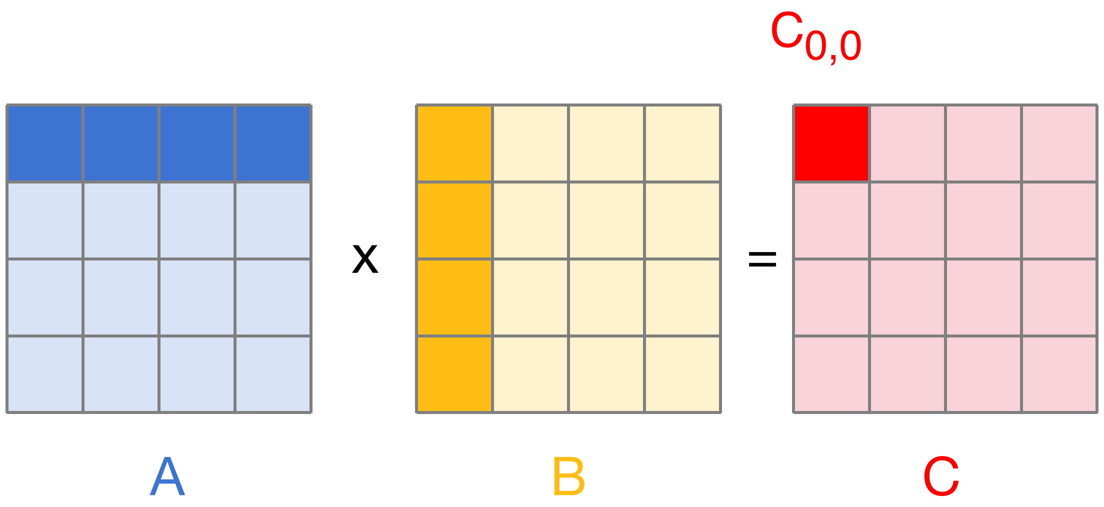
- Matrix multiplication is a basic operation in many engineering, data, and imaging processing tasks.
  - Image filtering, noise reduction, machine learning…
  - Many closely related operations
- Task (and implementation): **dgemm**
  - Double(-Precision floating-point) GEneral Matrix Multiplication



Recall Matrix Multiplication

- Basic implementation in C

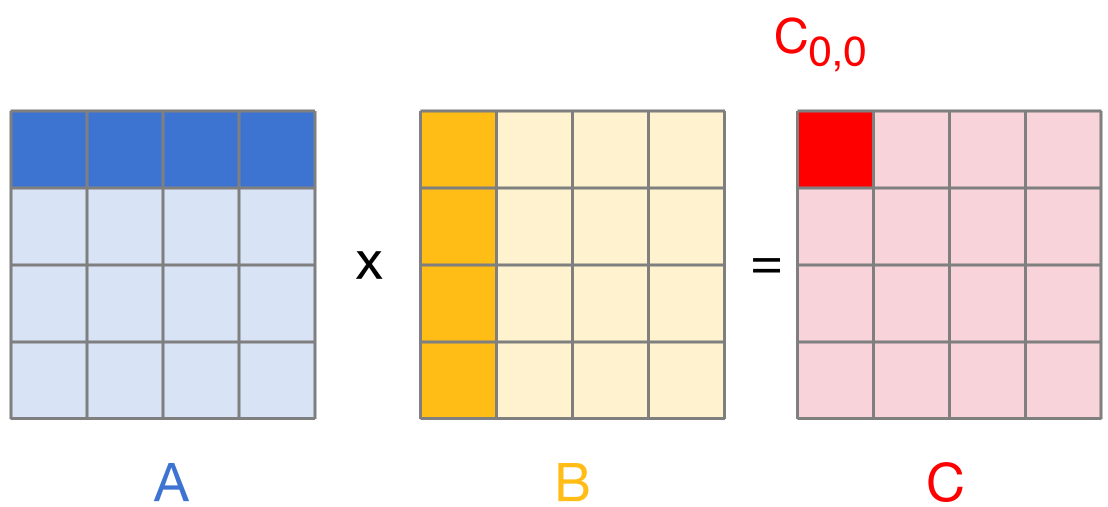
```
for (int n = 0; n < N; n++) {
    for (int k = 0; k < K; k++) {
        C[n][k] = 0; // Initialization
        for (int m = 0; m < M; m++) {
            C[n][k] += A[n][m] * B[m][k];
        }
    }
}
```

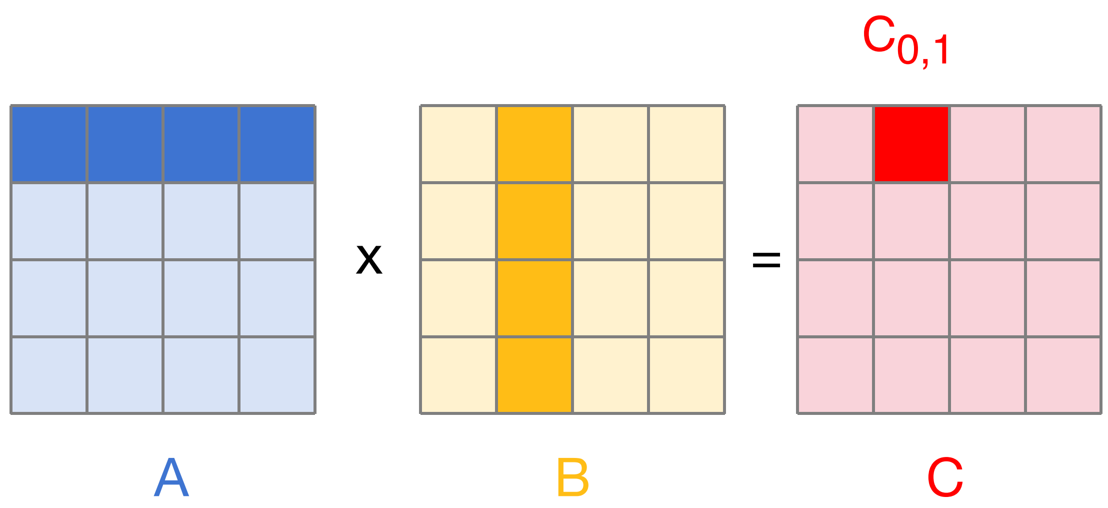


Recall Matrix Multiplication

- Basic implementation in C

```
for (int n = 0; n < N; n++) {
    for (int k = 0; k < K; k++) {
        C[n][k] = 0; // Initialization
        for (int m = 0; m < M; m++) {
            C[n][k] += A[n][m] * B[m][k];
        }
    }
}
```

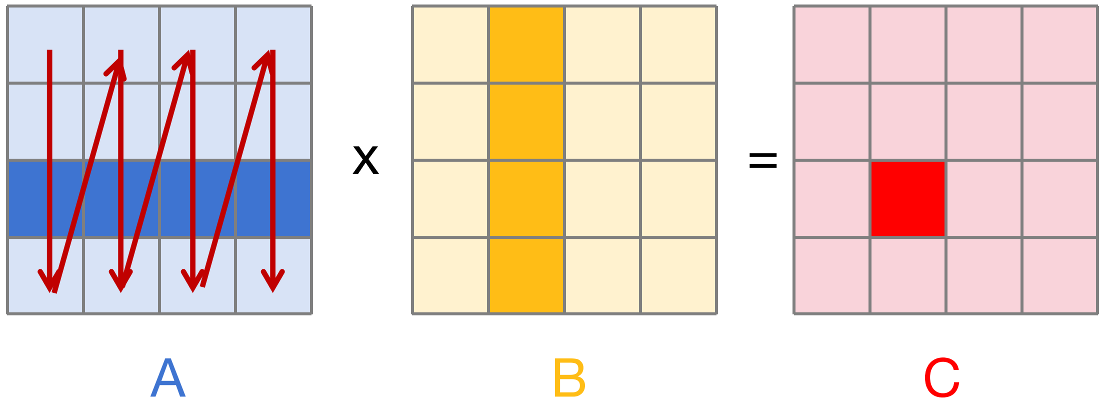




DGEMM in C

- Set C = A x B, and A, B, C are N x N matrices in column-major order.
- In C, they are actually stored in row-major order.
- FLOPS: Floating Point Operations Per Second.
  - DGEMM has $2*N^3$ Floating Point Operations (`fadd`, `fmul`)

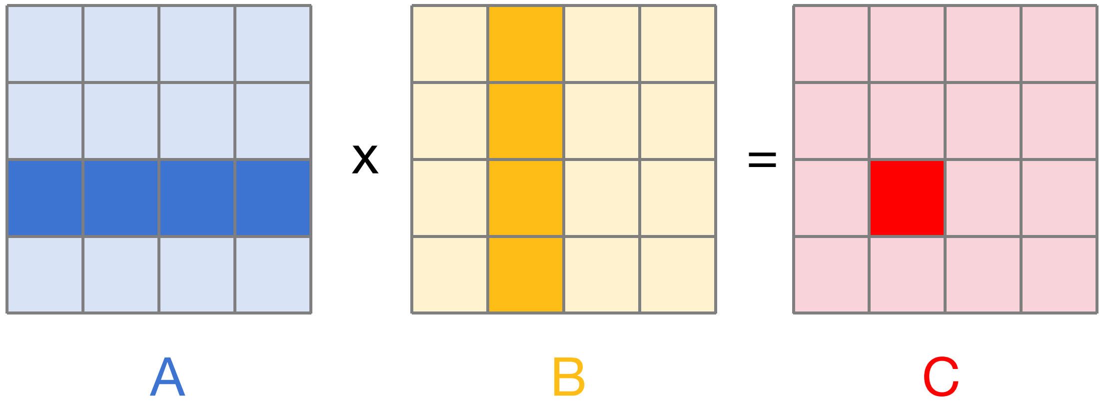
```
for (int n = 0; n < N; n++) {
    for (int k = 0; k < K; k++) {
        C[n][k] = 0; // Initialization
        for (int m = 0; m < M; m++) {
            C[n][k] += A[n][m] * B[m][k];
        }
    }
}
```



Column-major

Observations

```
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++) {
        Cij += A[i+k*N] * B[k+j*N];
      }
      C[i+j*N] = Cij;
    }
  }
}
```



- Parallelism opportunities in DGEMM
  - Element-wise computation of `Cij;`
  - Multiplication of `Aik` and `Bkj;`

A 2 x 2 Matrix Multiplication

- Definition of Matrix Multiply:

$$C_{i,j} = \sum_{k=1}^{2} A_{i,k} \times B_{k,j}$$

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

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

A 2 x 2 Matrix Multiplication

- Definition of Matrix Multiply:

$$C_{i,j} = \sum_{k=1}^{2} A_{i,k} \times B_{k,j}$$

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

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

A 2 x 2 Matrix Multiplication

- Use the XMM registers (contain two doubles per reg.)

$$C_{i,j} = \sum_{k=1}^{2} A_{i,k} \times B_{k,j}$$

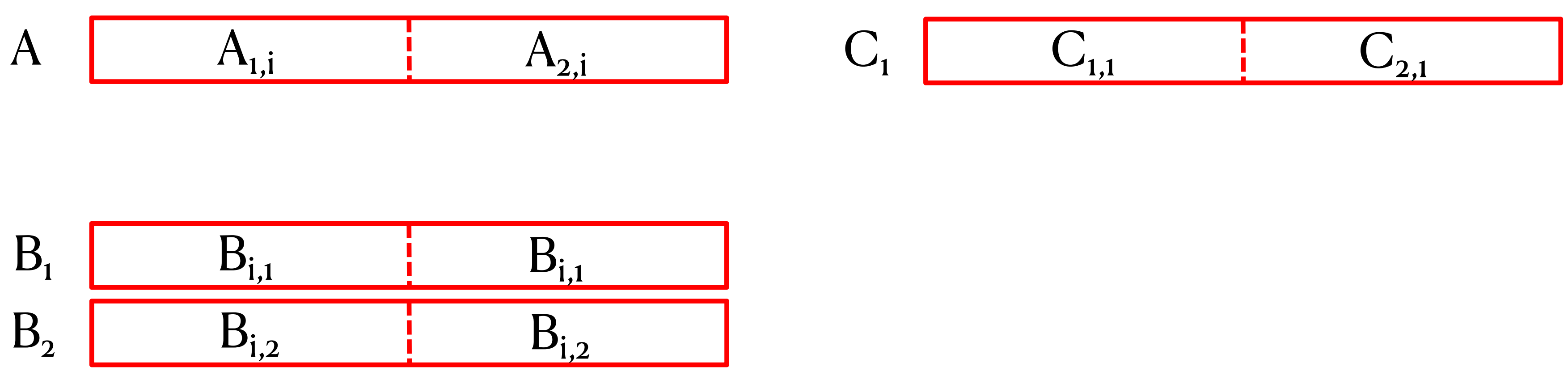
$$\begin{bmatrix} \boxed{A_{1,1}} & \boxed{A_{1,2}} \\ \boxed{A_{2,1}} & \boxed{A_{2,2}} \end{bmatrix} \text{x} \begin{bmatrix} \boxed{B_{1,1}} & B_{1,2} \\ \boxed{B_{2,1}} & B_{2,2} \end{bmatrix} = \begin{bmatrix} C_{1,1}=A_{1,1}B_{1,1}+A_{1,2}B_{2,1} & C_{1,2}=A_{1,1}B_{1,2}+A_{1,2}B_{2,2} \\ C_{2,1}=A_{2,1}B_{1,1}+A_{2,2}B_{2,1}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

A	$A_{1,i}$	$A_{2,i}$
C_1	$C_{1,1}$	$C_{2,1}$
B_1	$B_{i,1}$	$B_{i,1}$
B_2	$B_{i,2}$	$B_{i,2}$

A 2 x 2 Matrix Multiplication

- Use the XMM registers (contain two doubles per reg.)

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




```
double A[4] __attribute__ ((aligned (16)));
double B[4] __attribute__ ((aligned (16)));
double C[4] __attribute__ ((aligned (16)));
//double arrays declared and initialized (not shown)

__m128d c1,c2,a,b1,b2;
//vector variables declared
```

A 2 x 2 Matrix Multiplication

- Use the XMM registers (contain two doubles per reg.)

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

A: $[A_{1,i} \mid A_{2,i}]$

C_1: $[C_{1,1} \mid C_{2,1}]$

B_1: $[B_{i,1} \mid B_{i,1}]$

B_2: $[B_{i,2} \mid B_{i,2}]$

```
// used aligned loads to set
// c1 = [c_11 | c_21]
c1 = _mm_load_pd(C+0);
// c2 = [c_12 | c_22]
c2 = _mm_load_pd(C+2);
```

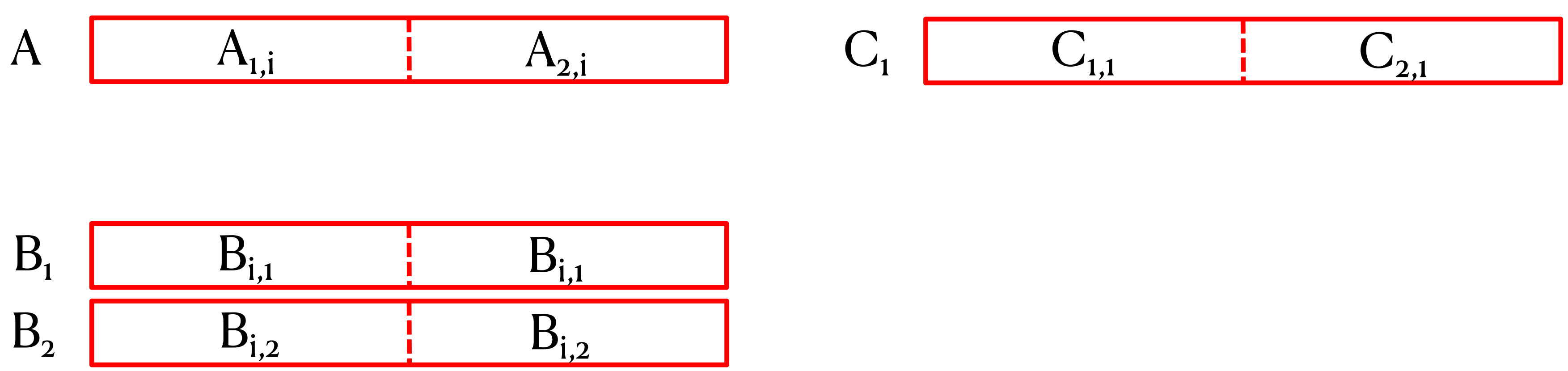
```
double A[4] __attribute__ ((aligned (16)));
double B[4] __attribute__ ((aligned (16)));
double C[4] __attribute__ ((aligned (16)));
//double arrays declared and initialized (not shown)

__m128d c1,c2,a,b1,b2;
//vector variables declared
```

A 2 x 2 Matrix Multiplication

- Use the XMM registers (contain two doubles per reg.)

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

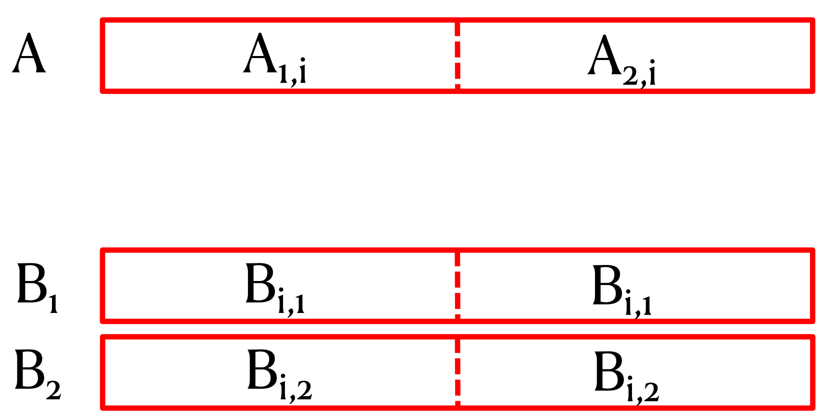



```
// used aligned loads to set a/b1
a = _mm_load_pd(A); //A11 A21
b1 = _mm_load1_pd(B); //B11 B11
//compute partial sum
c1 = _mm_add_pd(c1,_mm_mul_pd(a,b1));//A11B11 A21B11
```

A 2 x 2 Matrix Multiplication

- Use the XMM registers (contain two doubles per reg.)

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




```
// used aligned loads to set a/b1
a = _mm_load_pd(A); //A11 A21
b1 = _mm_load1_pd(B); //B11 B11
//compute partial sum
c1 = _mm_add_pd(c1,_mm_mul_pd(a,b1));//A11B11 A21B11
// used aligned loads to set a/b2
```

```
a = _mm_load_pd(A+2); //A12 A22
b2 = _mm_load1_pd(B+1); //B21 B21
//compute C11,C21
c1 = _mm_add_pd(c1,_mm_mul_pd(a,b2));
//store C11,C21
_mm_store_pd(C,c1);
```

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 the use of intrinsics
  - Achieve efficiency beyond that of optimizing compiler

Appendix

- `lscpu` in terminal to check if your CPU supports certain SIMD extensions;

```
siting@siting-ThinkPad-T14p-Gen-1: ~/Downloads
siting@siting-ThinkPad-T14p-Gen-1:~/Downloads$ lscpu
架构：                    x86_64
  CPU 运行模式：          32-bit, 64-bit
  Address sizes:          46 bits physical, 48 bits virtual
  字节序：                Little Endian
CPU:                      20
  在线 CPU 列表：         0-19
厂商 ID：                 GenuineIntel
  型号名称：              13th Gen Intel(R) Core(TM) i9-13900H
    CPU 系列：            6
    型号：                186
    每个核的线程数：      2
    每个座的核数：        14
    座：                  1
    步进：                2
    CPU(s) scaling MHz:   39%
    CPU 最大 MHz：        2600.0000
    CPU 最小 MHz：        400.0000
    BogoMIPS:             5990.40
    标记：                fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx f
                          xsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts
                          rep_good nopl xtopology nonstop_tsc cpuid aperfmperf tsc_known_freq pni pclmulqdq dtes64 monit
                          or ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt
                           tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb ssbd i
                          brs ibpb stibp ibrs_enhanced tpr_shadow flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1
                          avx2 smep bmi2 erms invpcid rdseed adx smap clflushopt clwb intel_pt sha_ni xsaveopt xsavec xg
                          etbv1 xsaves split_lock_detect user_shstk avx_vnni dtherm ida arat pln pts hwp hwp_notify hwp_
                          act_window hwp_epp hwp_pkg_req hfi vnmi umip pku ospke waitpkg gfni vaes vpclmulqdq tme rdpid
                          movdiri movdir64b fsrm md_clear serialize pconfig arch_lbr ibt flush_l1d arch_capabilities
```

Appendix

- Toy example, 2x2 matrix multiplication

```
//Toy example, 2x2 matrix multiplication for CS110 2025, ShanghaiTech University, all rights reserved
#include <stdio.h>
#include <time.h>
#include <emmintrin.h>
#include <immintrin.h>
void main(){
  double A[4]__attribute__((aligned(16)));
  double B[4]__attribute__((aligned(16)));
  double C[4]__attribute__((aligned(16)));
  int ida = 2;
  int i = 0;
  __m128d c1, c2, a, b1, b2;
  A[0] = 1.0;A[1] = 1.1;A[2] = 0.0;A[3] = 1.0;
  B[0] = 1.0;B[1] = 2.0;B[2] = 5.0;B[3] = 4.0;
  C[0] = 0.0;C[1] = 0.0;C[2] = 0.0;C[3] = 0.0;
  c1 = _mm_load_pd(C+0*ida);
  c2 = _mm_load_pd(C+1*ida);
  for (i=0;i<2;i++){
   a = _mm_load_pd(A+i*ida);
   b1 = _mm_load1_pd(B+i+0*ida);
   b2 = _mm_load1_pd(B+i+1*ida);
   c1 = _mm_add_pd(c1,_mm_mul_pd(a,b1));
   c2 = _mm_add_pd(c2,_mm_mul_pd(a,b2));
   }
   _mm_store_pd(C+0*ida,c1);
   _mm_store_pd(C+1*ida,c2);
   printf("[%g,%g,%g,%g]\n",C[0],C[1],C[2],C[3]);
  return;
}
```