

计算机组成与系统结构

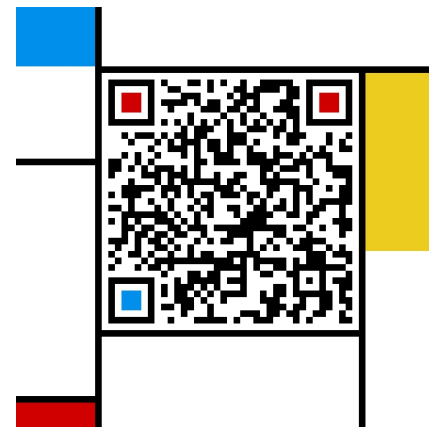
Computer Organization & System Architecture

Huang Kejie (黄科杰) 百人计划研究员

Office: 玉泉校区老生仪楼 304

Email address: huangkejie@zju.edu.cn

HP: 17706443800



Agenda

- Parallel processing
- Single instruction, multiple data
- SIMD matrix multiplication
- Amdahl's law
- Loop unrolling
- Memory access strategy - blocking
- And in Conclusion, ...

Reference Problem

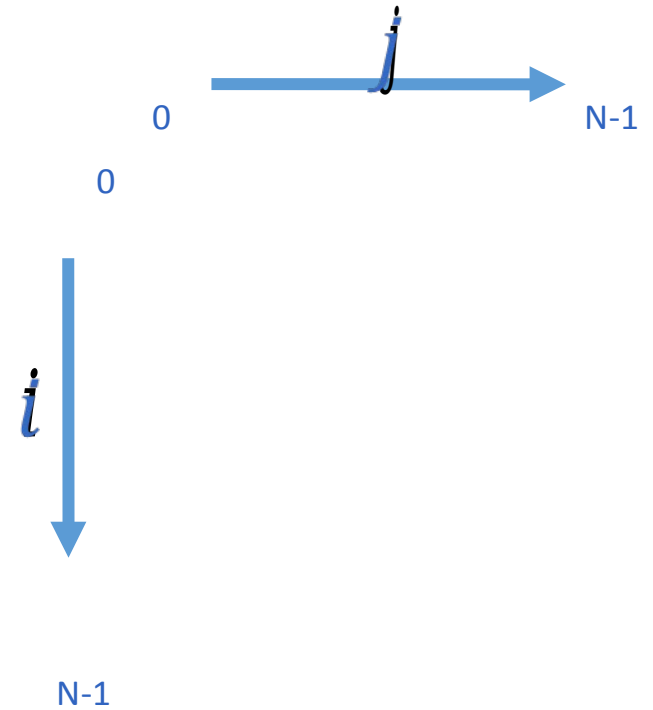
- Matrix multiplication
 - Basic operation in many engineering, data, and imaging processing tasks
 - Image filtering, noise reduction, ...
 - Many closely related operations
- **dgemm**
 - double-precision floating-point matrix multiplication

Application Example: Deep Learning

- Image classification (cats ...)
- Pick “best” vacation photos
- Machine translation
- Clean up accent
- Fingerprint verification
- Automatic game playing

Matrices

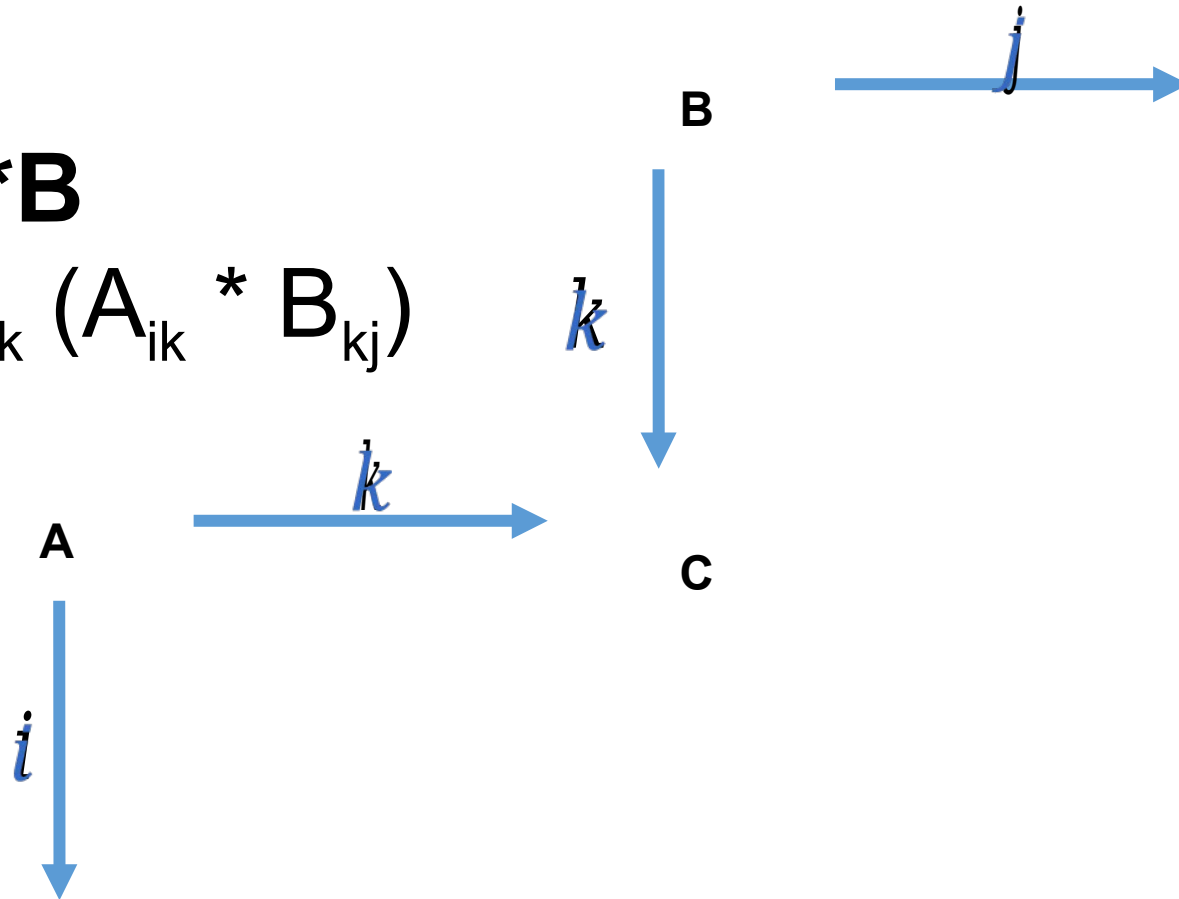
- Square matrix of dimension $N \times N$



Matrix Multiplication

$$\mathbf{C} = \mathbf{A} * \mathbf{B}$$

$$C_{ij} = \sum_k (A_{ik} * B_{kj})$$



Reference: Python

- Matrix multiplication in 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

- $c = a * b$
- 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;
        }
}
```


Timing Program Execution

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void) {
    // start time
    // Note: clock() measures execution time, not real time
    //       big difference in shared computer environments
    //       and with heavy system load
    clock_t start = clock();

    // task to time goes here:
    // dgemm(N, ...);

    // "stop" the timer
    clock_t end = clock();

    // compute execution time in seconds
    double delta_time = (double)(end-start)/CLOCKS_PER_SEC;
}
```

C versus Python

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



Which class gives you this kind of power?
We could stop here ... but why? Let's do better!

Why Parallel Processing?

- CPU Clock Rates are no longer increasing
 - Technical & economic challenges
 - Advanced cooling technology too expensive or impractical for most applications
 - Energy costs are prohibitive
- Parallel processing is only path to higher speed
 - Compare airlines:
 - Maximum speed limited by speed of sound and economics
 - Use more and larger airplanes to increase throughput
 - And smaller seats ...

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 in the next few lectures

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

- 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

Software

Hardware

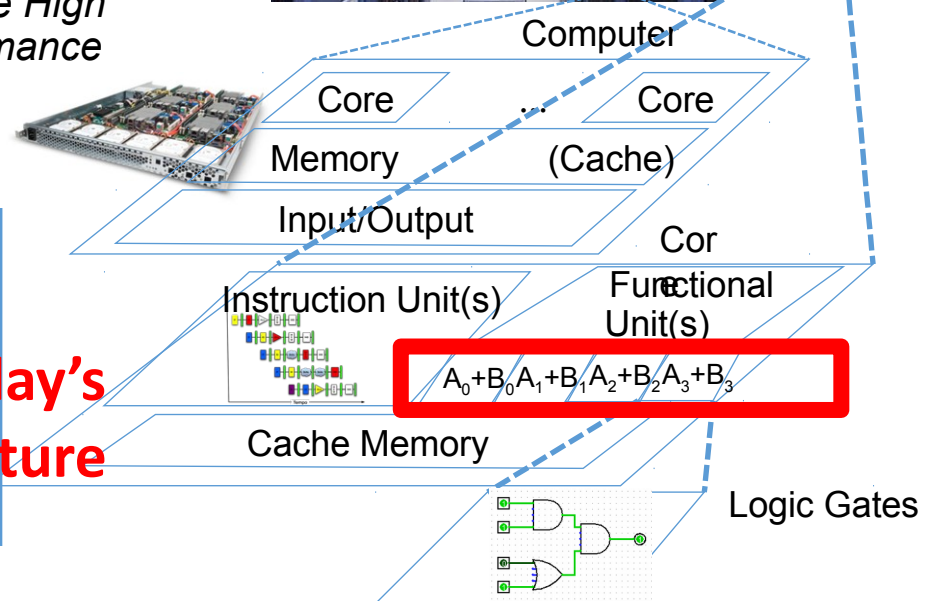
*Harness
Parallelism &
Achieve High
Performance*

Warehouse
Scale
Computer

Smart
Phone



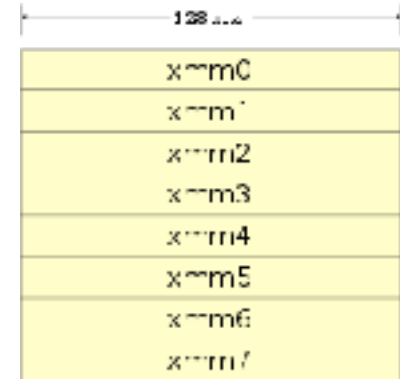
**Today's
Lecture**



Types of Parallelism

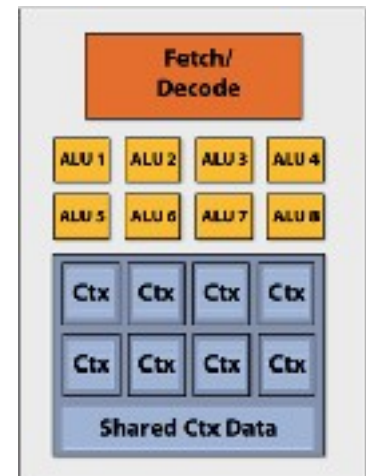
- Parallelism in Hardware (Uniprocessor)

- Parallelism in a Uniprocessor
 - Pipelining
 - Superscalar, VLIW etc.
- SIMD instructions, Vector processors, GPUs
- Multiprocessor
 - Symmetric shared-memory multiprocessors
 - Distributed-memory multiprocessors
 - Chip-multiprocessors a.k.a. Multi-cores
- Multicomputers a.k.a. clusters

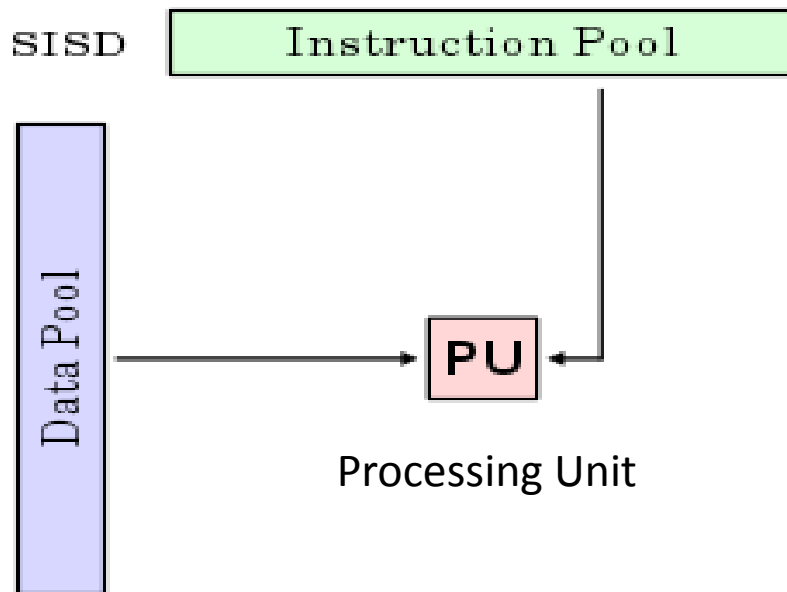


- Parallelism in Software

- Instruction level parallelism
- Task-level parallelism
- Data parallelism
- Transaction level parallelism



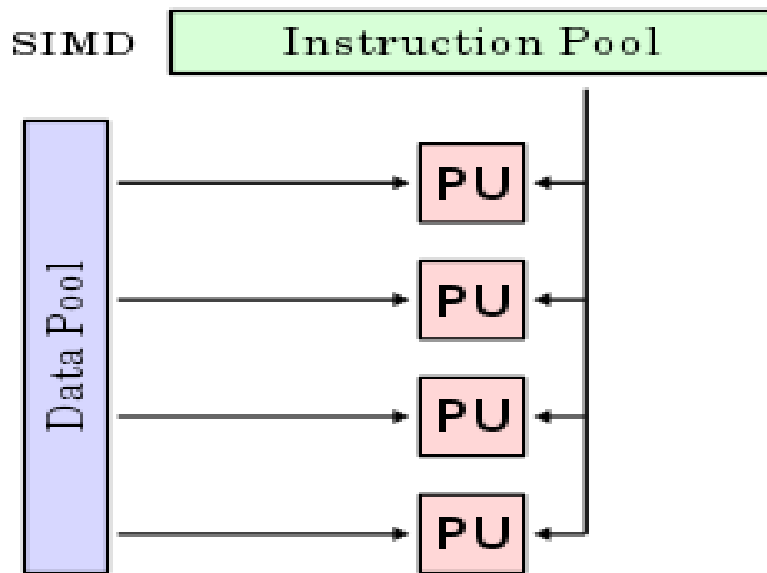
Single-Instruction/Single-Data Stream (SISD)



- Sequential computer that exploits no parallelism in either the instruction or data streams. Examples of SISD architecture are traditional uniprocessor machines
 - E.g. our trusted RISC-V pipeline

This is what we did up to now in this course

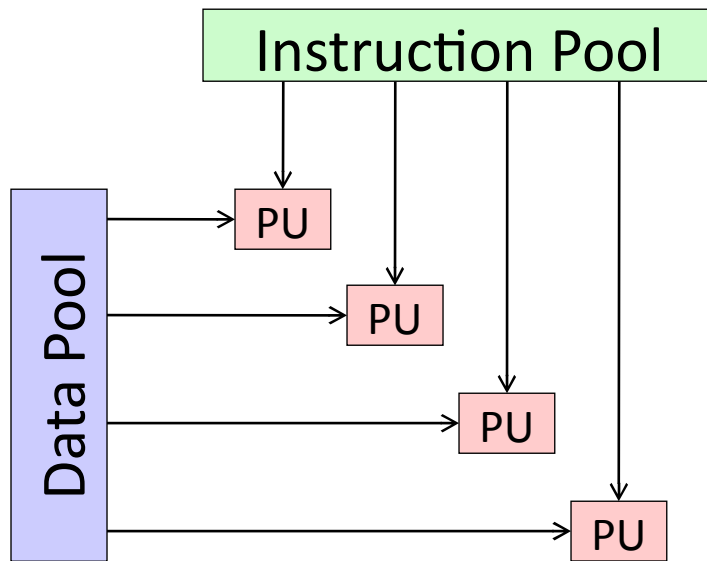
Single-Instruction/Multiple-Data Stream (SIMD or “sim-dee”)



- 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)

Today's topic.

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

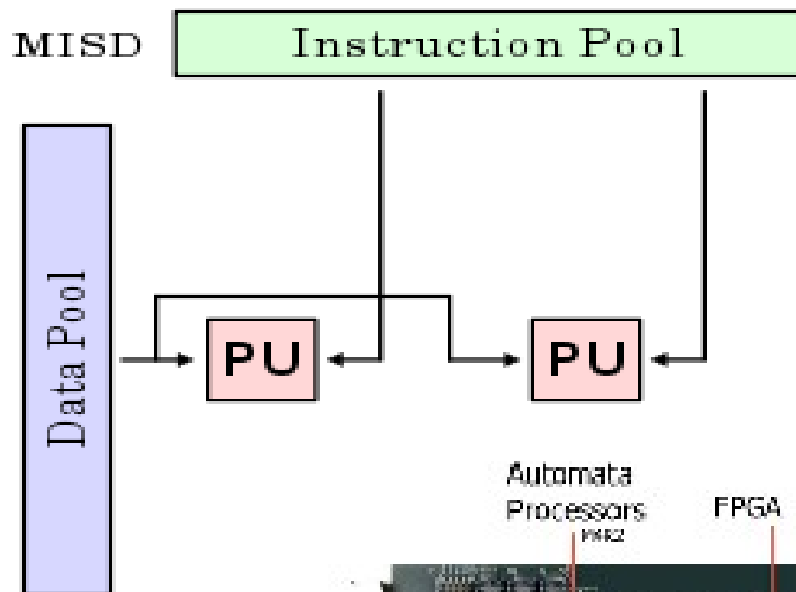


- Multiple autonomous processors simultaneously executing different instructions on different data.

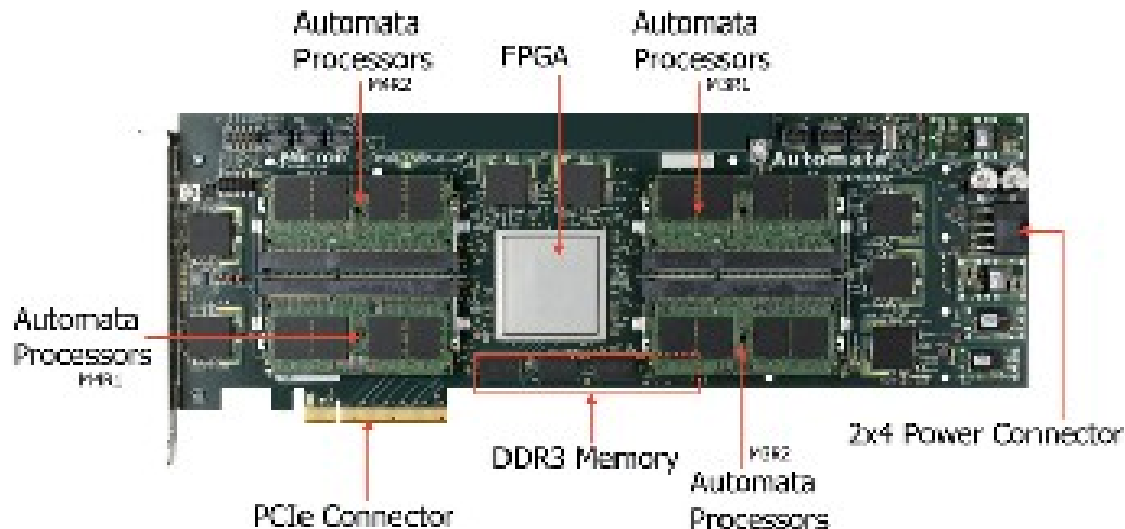
- MIMD architectures include multicore and Warehouse-Scale Computers

Succeeding topic and beyond.

Multiple-Instruction/Single-Data Stream (MISD)



- Different instructions on the same data
- Fault-tolerant computers, Near memory computing (Micron Automata processor).



This has few applications. Not covered in this course

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)

- SIMD and MIMD are currently the 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

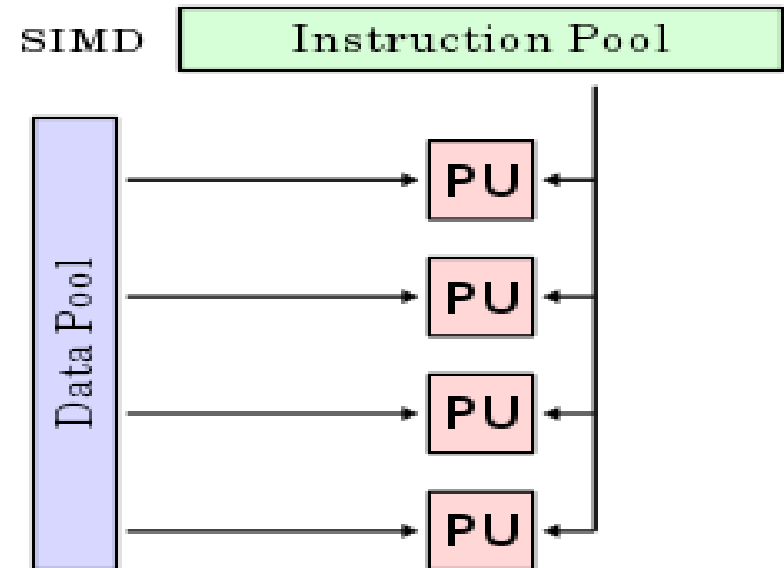
SIMD Applications & Implementations

- Applications

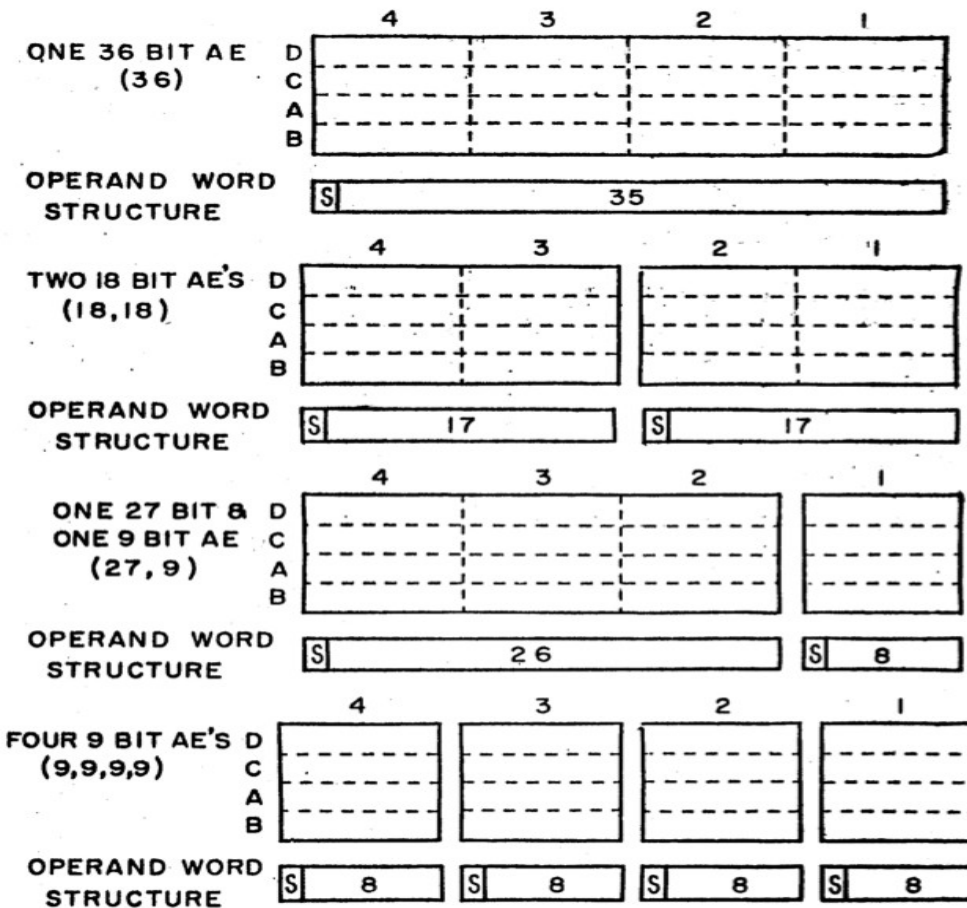
- Scientific computing
 - Matlab, NumPy
- Graphics and video processing
 - Photoshop, ...
- Big Data
 - Deep learning
- Gaming
- ...

- Implementations

- x86
- ARM
- RISC-V vector extensions

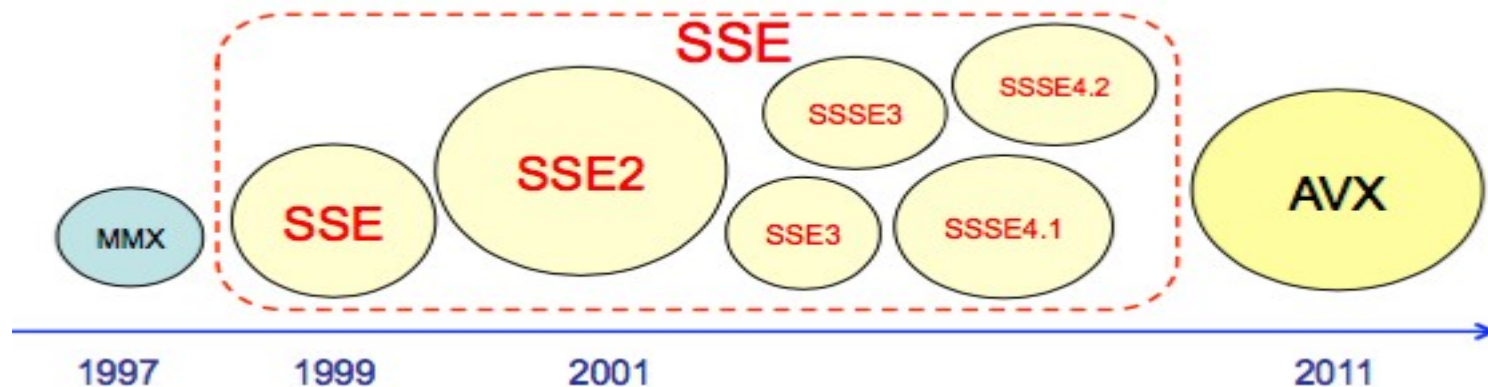


First SIMD Extensions: MIT Lincoln Labs TX-2, 1957



x86 SIMD Evolution

- New instructions
- New, wider, more registers
- More parallelism



<http://svmoore.pbworks.com/w/file/fetch/70583970/VectorOps.pdf>

Laptop CPU Specs

```
$ sysctl -a | grep cpu
```

```
hw.physicalcpu: 2
```

```
hw.logicalcpu: 4
```

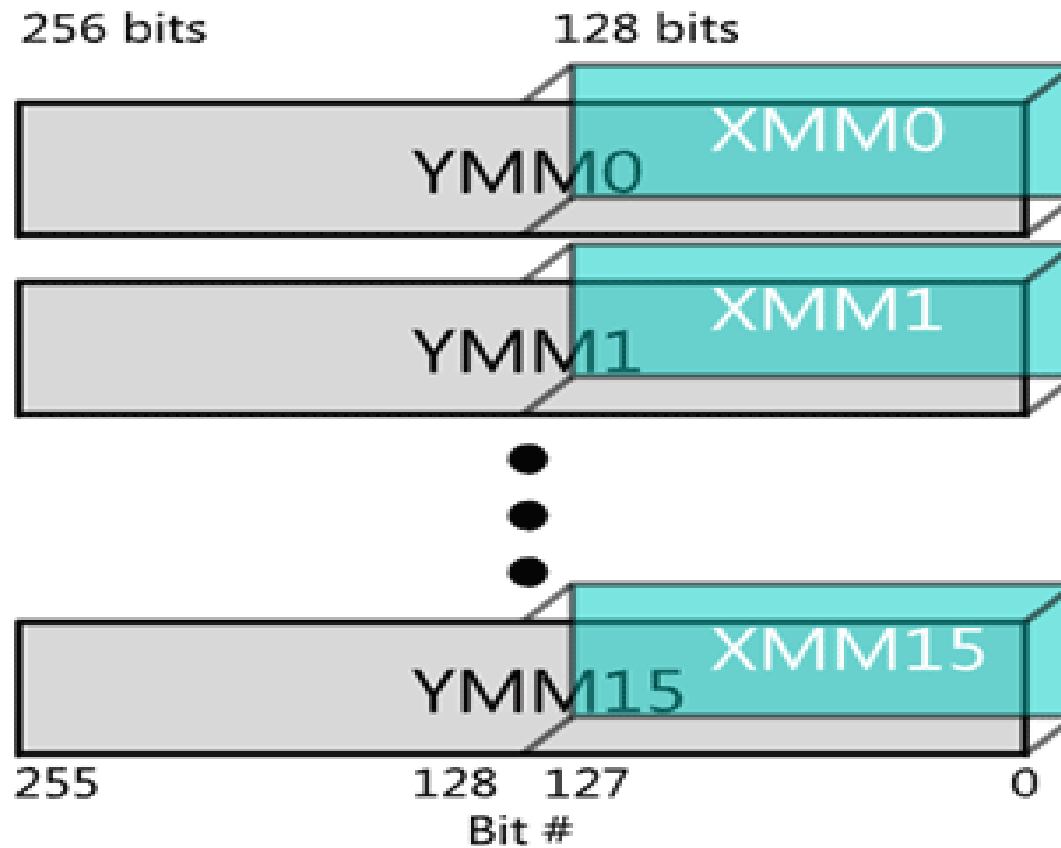
```
machdep.cpu.brand_string:
```

```
Intel(R) Core(TM) i7-5557U CPU @ 3.10GHz
```

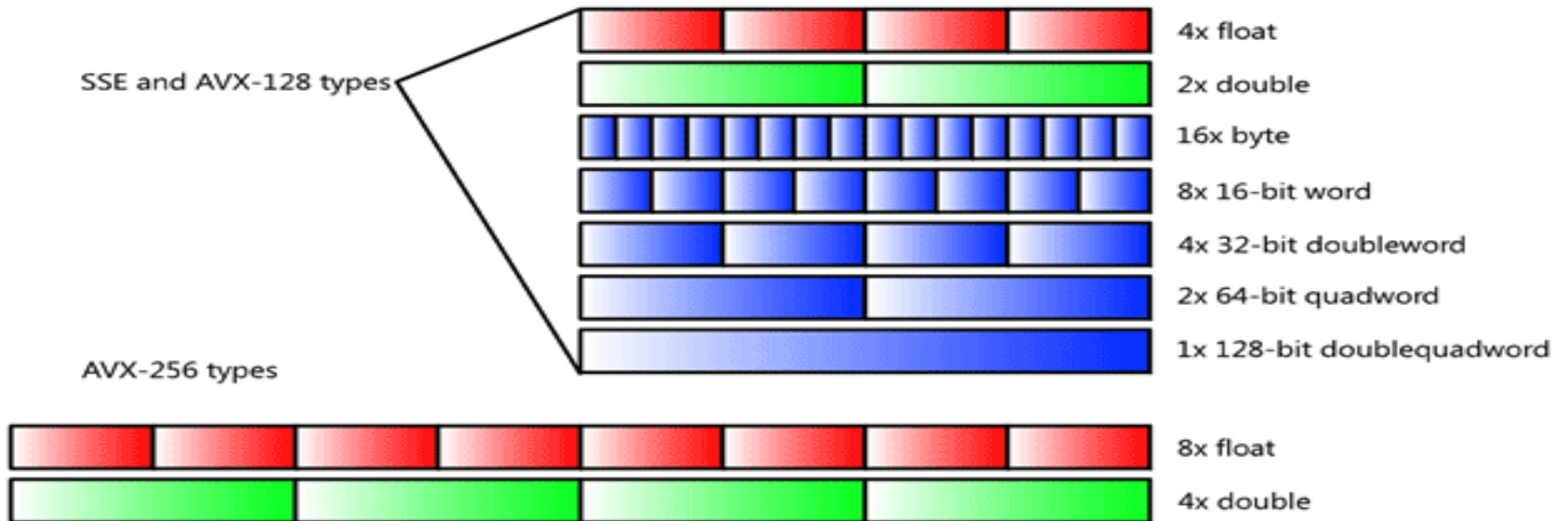
```
machdep.cpu.features: FPU VME DE PSE TSC MSR PAE MCE  
CX8 APIC SEP MTRR PGE MCA CMOV PAT PSE36 CLFSH DS  
ACPI MMX FXSR SSE SSE2 SS HTT TM PBE SSE3 PCLMULQDQ  
DTES64 MON DSCPL VMX EST TM2 SSSE3 FMA CX16 TPR  
PDCM SSE4.1 SSE4.2 x2APIC MOVBE POPCNT AES PCID  
XSAVE OSXSAVE SEGLIM64 TSCTMR AVX1.0 RDRAND F16C
```

```
machdep.cpu.leaf7_features: SMEP ERMS RDWRFSGS  
TSC_THREAD_OFFSET BMI1 AVX2 BMI2 INVPCID SMAP  
RDSEED ADX IPT FPU_CSDS
```

SIMD Registers



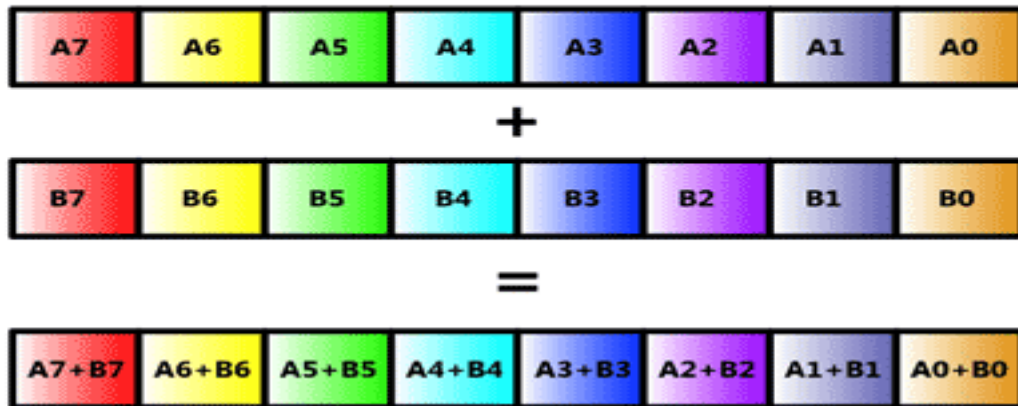
SIMD Data Types



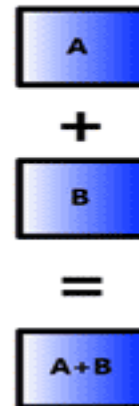
(Now also AVX-512 available)

SIMD Vector Mode

SIMD Mode



Scalar Mode



Problem

- Today's compilers (largely) do not generate SIMD code
- Back to assembly ...
- x86 (not using RISC-V as no vector hardware yet)
 - Over 1000 instructions to learn ...
 - Green Book
- Can we use the compiler to generate all non-SIMD instructions?

x86 Intrinsics AVX Data Types

- Intrinsics: Direct access to registers & assembly from C

Register

Type	Meaning
<code>__m256</code>	256-bit as eight single-precision floating-point values, representing a YMM register or memory location
<code>__m256d</code>	256-bit as four double-precision floating-point values, representing a YMM register or memory location
<code>__m256i</code>	256-bit as integers, (bytes, words, etc.)
<code>__m128</code>	128-bit single precision floating-point (32 bits each)
<code>__m128d</code>	128-bit double precision floating-point (64 bits each)

Intrinsics AVX Code Nomenclature

Marking	Meaning
[s/d]	Single- or double-precision floating point
[i/u]nnn	Signed or unsigned integer of bit size <i>nnn</i> , where <i>nnn</i> is 128, 64, 32, 16, or 8
[ps/pd/sd]	Packed single, packed double, or scalar double
epi32	Extended packed 32-bit signed integer
si256	Scalar 256-bit integer

x86 SIMD “Intrinsics”



Technologies

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

← 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

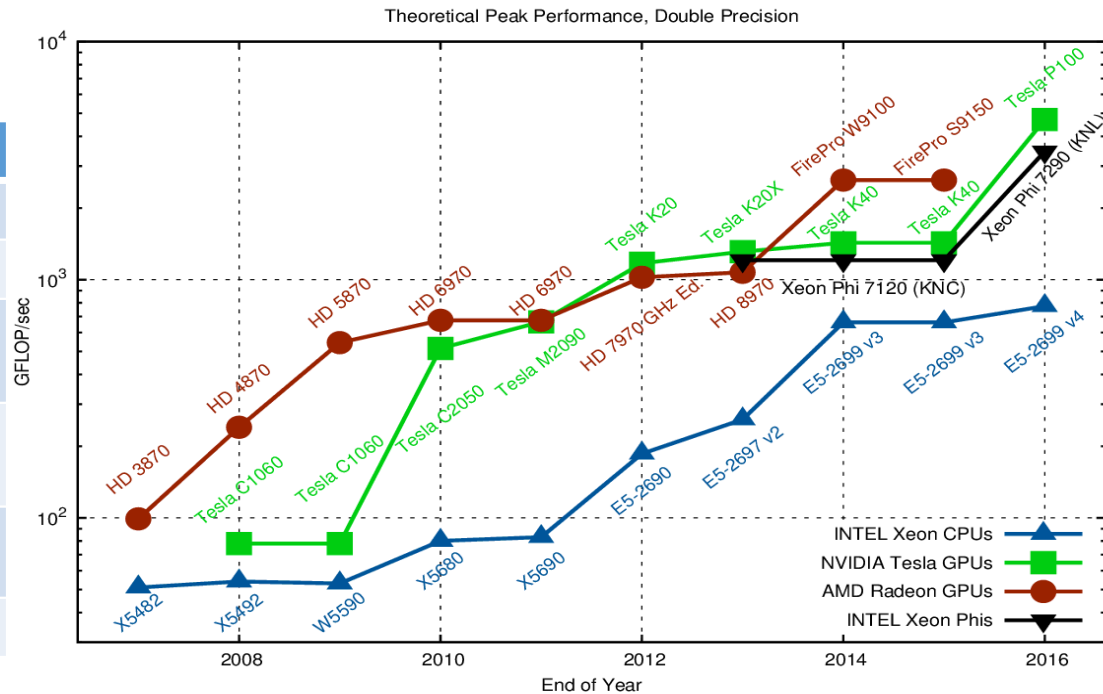
Architecture	Latency	Throughput
Haswell	5	0.5
Ivy Bridge	5	1
Sandy Bridge	5	1

← 2 instructions per clock cycle (CPI = 0.5)

<https://software.intel.com/sites/landingpage/IntrinsicsGuide/>

Raw Double-Precision Throughput

Characteristic	Value
CPU	i7-5557U
Clock rate (sustained)	3.1 GHz
Instructions per clock (mul_pd)	2
Parallel multiplies per instruction	4
Peak double FLOPS	24.8 GFLOPS



<https://www.karlsruhp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/>

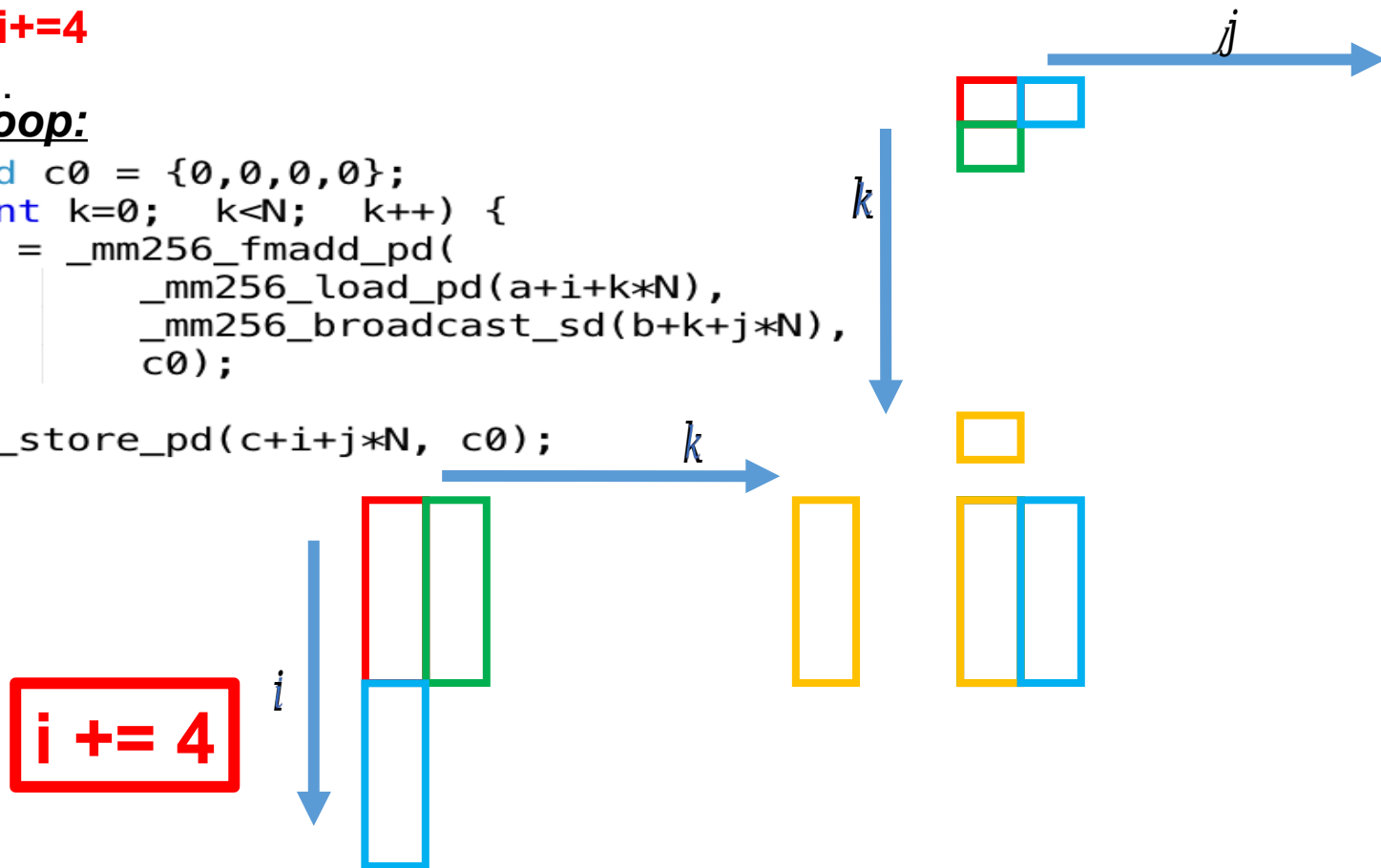
Actual performance is lower because of overhead

Vectorized Matrix Multiplication


for i ...; **i+=4**

for j ...
Inner Loop:

```
__m256d c0 = {0,0,0,0};  
for (int k=0; k<N; k++) {  
    c0 = _mm256_fmadd_pd(  
        _mm256_load_pd(a+i+k*N),  
        _mm256_broadcast_sd(b+k+j*N),  
        c0);  
}  
_mm256_store_pd(c+i+j*N, c0);
```



“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
        }
    }
}
```

Performance

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
- But still << theoretical 25 GFLOPS!

A trip to LA

Commercial airline:

Get to SFO & check-in	SFO → LAX	Get to destination
3 hours	1 hour	3 hours

Total time: 7 hours

Supersonic aircraft:

Get to SFO & check-in	SFO → LAX	Get to destination
3 hours	6 min	3 hours

Total time: 6.1 hours

Speedup:

Flying time

$$S_{flight} = 60 / 6 = 10x$$

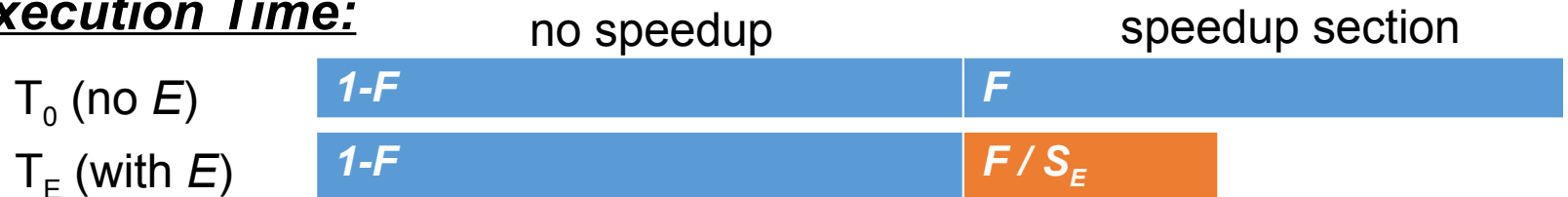
Trip time

$$S_{trip} = 7 / 6.1 = 1.15x$$

Amdahl's Law

- Get enhancement E for your new PC
 - E.g. floating-point rocket booster
- E
 - Speeds up some task (e.g. arithmetic) by factor S_E
 - F is fraction of program that uses this "task"

Execution Time:



Speedup:

$$\text{Speedup} = T_0 / T_E = 1 / ((1-F) + F/S_E)$$

Big Idea: Amdahl's Law

$$\text{Speedup} = T_O / T_E = 1 / ((1-F) + F/S_E)$$

Part not sped up

Part sped up

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

$$T_O / T_E = 1 / ((1-F) + F/S_E) = 1 / ((1 - 0.5) + 0.5/2) = 1.33 \ll 2$$

Maximum “Achievable” Speed-Up

$$\text{Speedup} = 1 / ((1-F) + F/S_E)$$

$$(S_E \rightarrow \infty) = 1/(1-F)$$

Question: What is a reasonable # of parallel processors to speed up an algorithm with $F = 95\%$? (i.e. 19/20th can be sped up)

a) Maximum speedup: $S_{\max} = 1/(1-0.95) = 20$

but needs $S_E = \infty$!

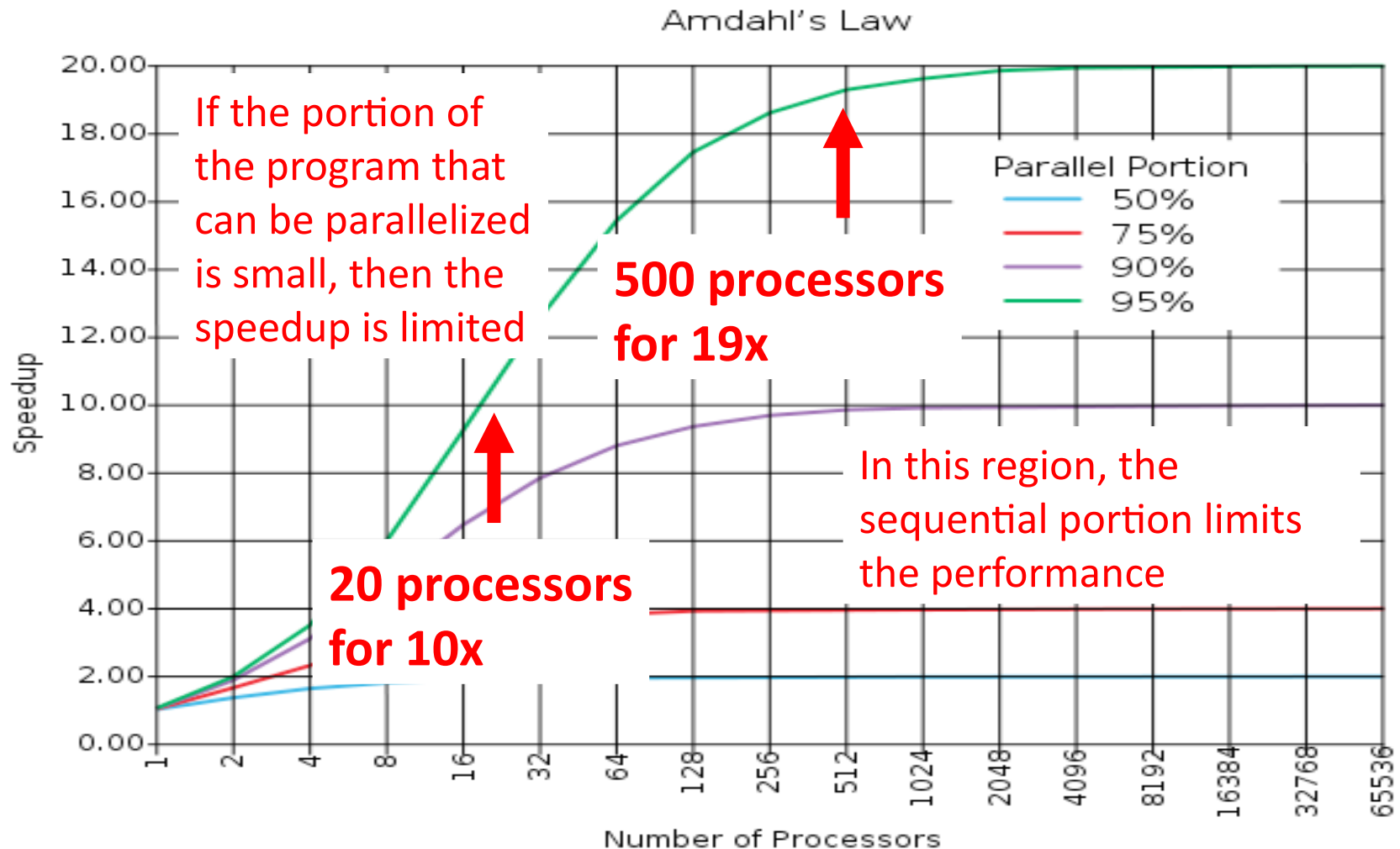
b) Reasonable “engineering” compromise:

Equal time in sequential and parallel code: $(1-F) = F/S_E$

$$S_E = F/(1-F) = 0.95/0.05 = 20$$

$$S = 1/(0.05+0.05) = 10$$

Maximum “Achievable” Speed-Up



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

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?

$$\text{Speedup} = 1 / ((1-F) + F/S_E)$$

RED: 5

GREEN: 20

ORANGE: 500

YELLOW: None of above

Amdahl's Law applied to dgemm

- Measured **dgemm** performance
 - Peak 5.5 GFLOPS
 - Large matrices 3.6 GFLOPS
 - Processor 24.8 GFLOPS
- Why are we not getting (close to) 25 GFLOPS?
 - Something else (not floating-point ALU) is limiting performance!
 - But what? Possible culprits:
 - Cache
 - Hazards
 - Let's look at both!

Pipeline Hazards – dgemm



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
```

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
```

Performance

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

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]);
        }
    }
}
```

How do you verify that the generated code is actually unrolled?

Performance

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

WOW!

?

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
 - $q = F/M = 2/3 \times N$
 - Good, since arithmetic is faster than memory access
 - Let's check the code ...

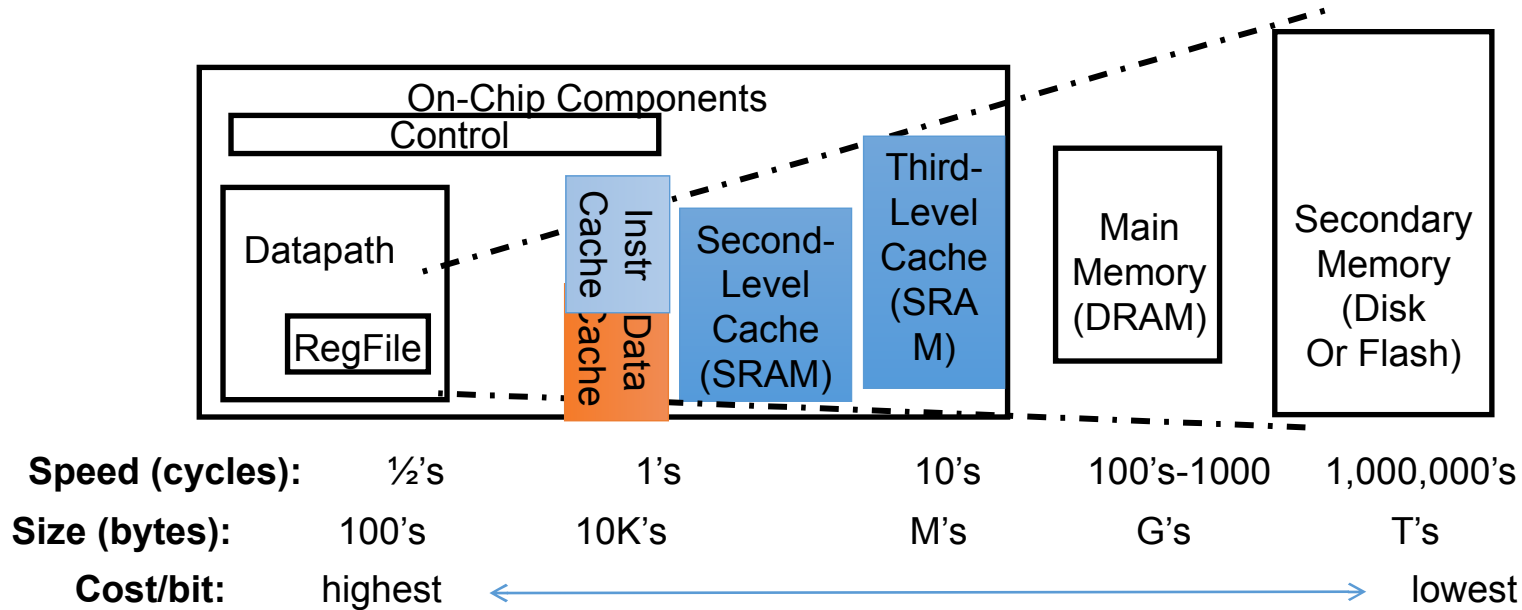
But memory is accessed repeatedly

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)));  
}
```

- $q = F/M = 1!$ (2 loads and 2 floating-point operations)

Typical Memory Hierarchy



- Where are the operands (A, B, C) stored?
- What happens as N increases?
- Idea: arrange that most accesses are to fast cache!

Sub-Matrix Multiplication or: Beating Amdahl's Law

- Blocking:
 - Rearrange code to use values loaded in cache many times
 - Only “few” accesses to slow main memory (DRAM) per floating point operation
 - → throughput limited by FP hardware and cache, not slow DRAM
 - P&H, RISC-V edition p. 465

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

N	Gflops			
	scalar	avx	unroll	blocking
32	1.30	4.56	12.95	13.80
160	1.30	5.47	19.70	21.79
480	1.32	5.27	14.50	20.17
960	0.91	3.64	6.91	15.82

And in Conclusion, ...

- Approaches to Parallelism
 - SISD, SIMD, MIMD (next lecture)
- SIMD
 - One instruction operates on multiple operands simultaneously
- Example: matrix multiplication
 - Floating point heavy → exploit Moore's law to make fast
- Amdahl's Law:
 - Serial sections limit speedup
 - Cache
 - Blocking
 - Hazards
 - Loop unrolling