CS 61C:
Great Ideas in Computer Architecture

Lecture 18:
Parallel Processing – SIMD

John Wawrzynek & Nick Weaver
http://inst.eecs.berkeley.edu/~cs61c

61C Survey

It would be nice to have a review lecture every once in a while, actually showing us how things fit in the bigger picture

Agenda

- **61C – the big picture**
- Parallel processing
- Single instruction, multiple data
- SIMD matrix multiplication
- Loop unrolling
- Memory access strategy - blocking
- And in Conclusion, …

61C Topics so far …

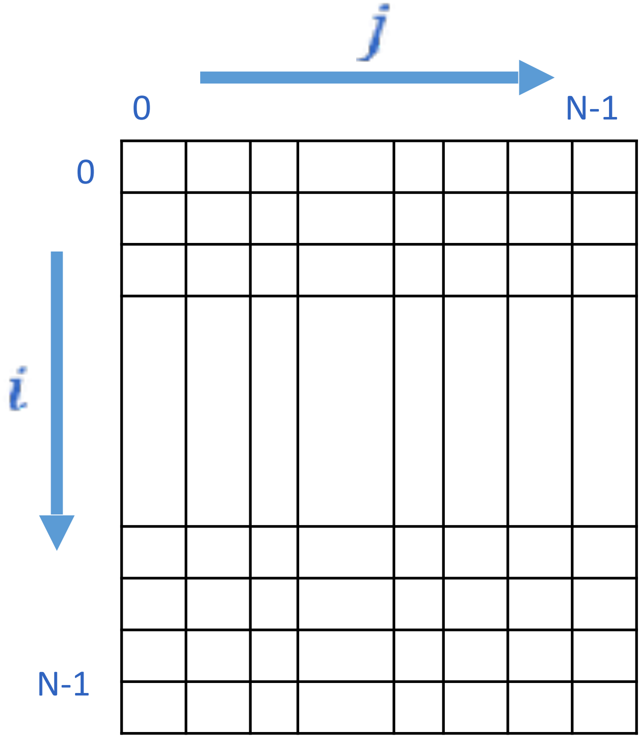
- What we learned:
  1. Binary numbers
  2. C
  3. Pointers
  4. Assembly language
  5. Processor micro-architecture
  6. Pipelining
  7. Caches
  8. Floating point
- What does this buy us?
  - Promise: execution speed
  - Let’s check!

Reference Problem

- Matrix multiplication
  - Basic operation in many engineering, data, and imaging processing tasks
    - Ex:, Image filtering, noise reduction, …
  - Core operation in Neural Nets and Deep Learning
    - Image classification (cats …)
    - Robot Cars
    - Machine translation
    - Fingerprint verification
    - Automatic game playing
- **`dgemm`**
  - **d**ouble-precision floating-point **ge**neral **m**atrix-**m**ultiply
  - Standard well-studied and widely used routine
  - Part of Linpack/Lapack

2D-Matrices

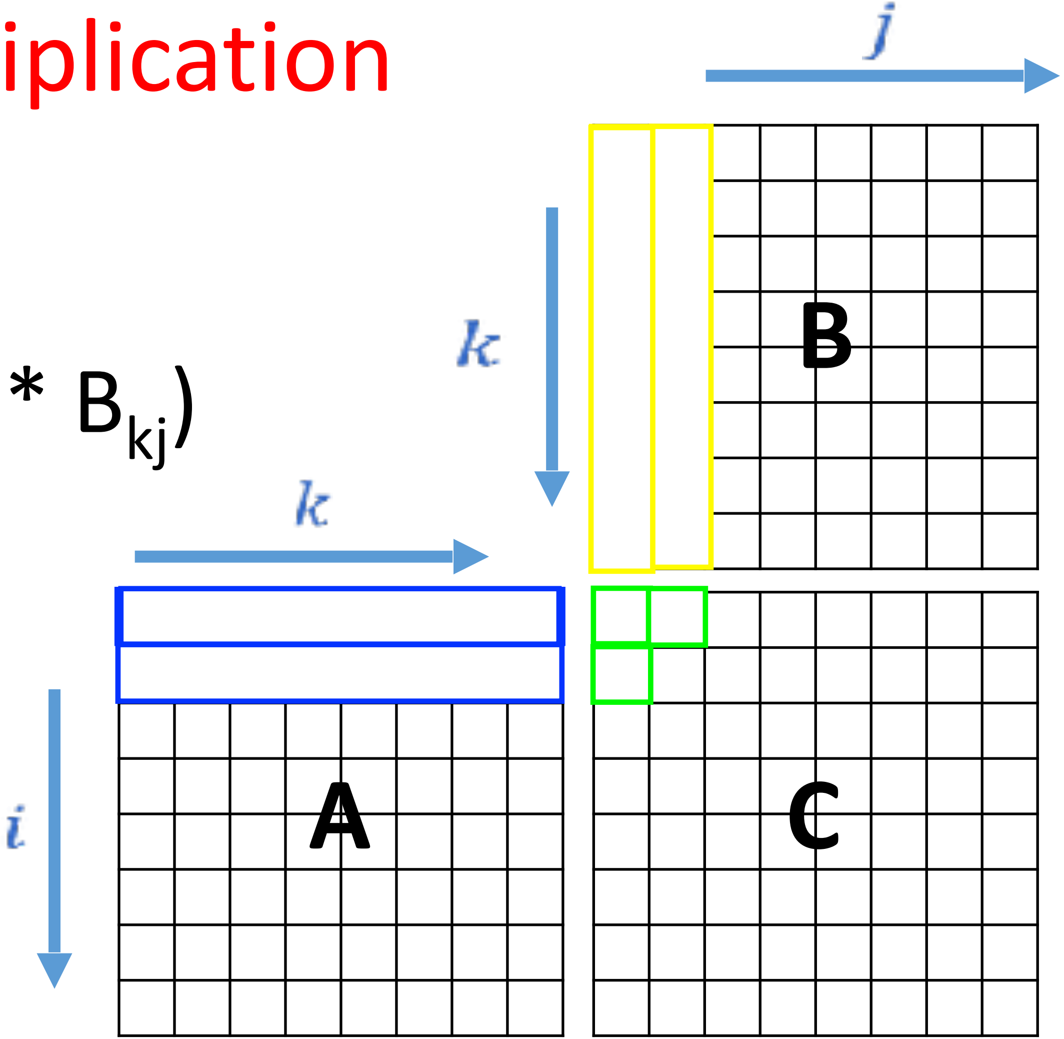
- Square matrix of dimension NxN
  - i indexes through rows
  - j indexes through columns



Matrix Multiplication

C = A*B

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

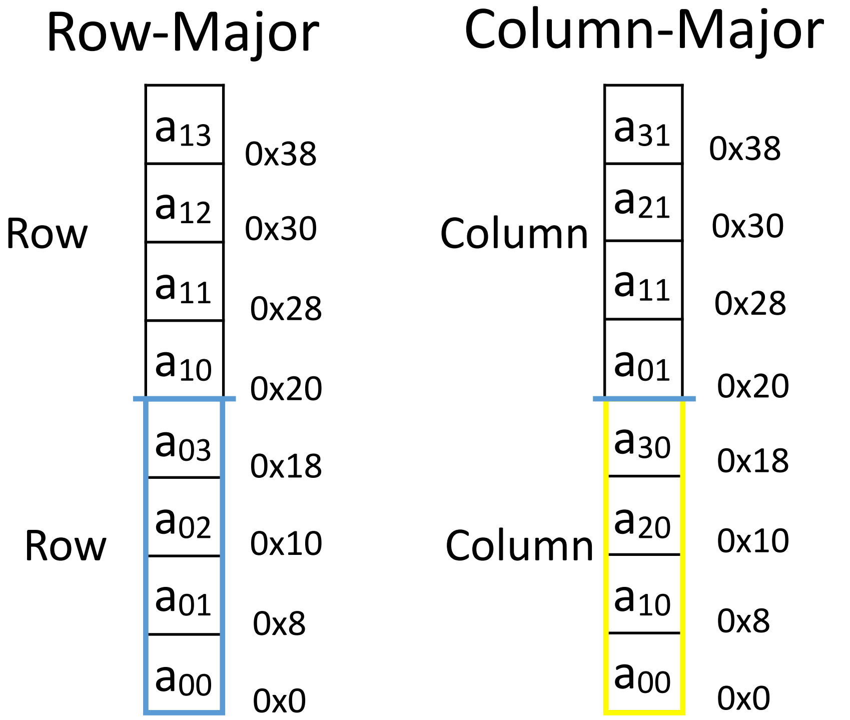


2D Matrix Memory Layout

a_{ij}

a_{00}	a_{01}	a_{02}	a_{03}
a_{10}	a_{11}	a_{12}	a_{13}
a_{20}	a_{21}	a_{22}	a_{23}
a_{30}	a_{31}	a_{32}	a_{33}

- a[][] in C uses row-major
- Fortran uses column-major
- Our examples us column-major



Row-Major

Row

a_{13}	0x38
a_{12}	0x30
a_{11}	0x28
a_{10}	0x20

Row

a_{03}	0x18
a_{02}	0x10
a_{01}	0x8
a_{00}	0x0

Column-Major

Column

a_{31}	0x38
a_{21}	0x30
a_{11}	0x28
a_{01}	0x20

Column

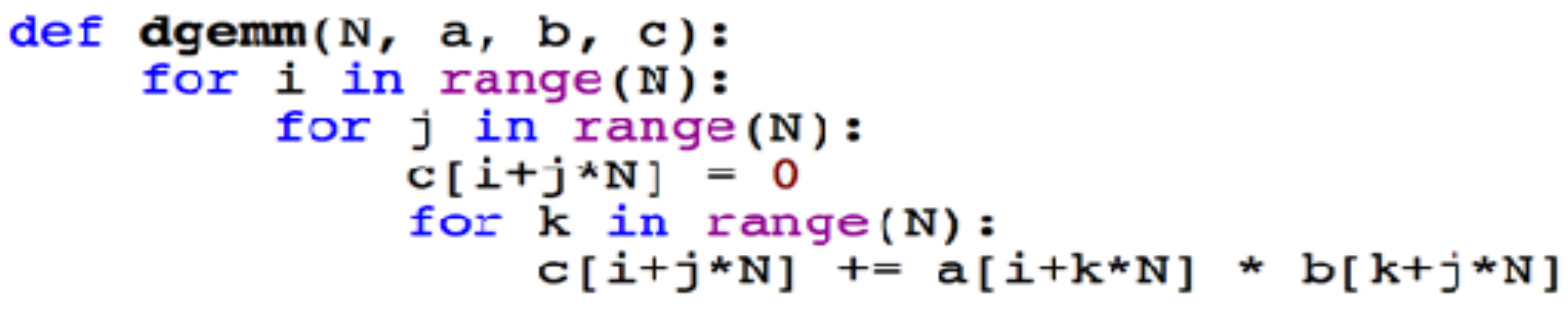
a_{30}	0x18
a_{20}	0x10
a_{10}	0x8
a_{00}	0x0

a_{ij} : a[i∗N + j]

a_{ij} : a[i + j∗N]

dgemm Reference Code: Python

- Linear addressing, assumes “column-major” memory layout



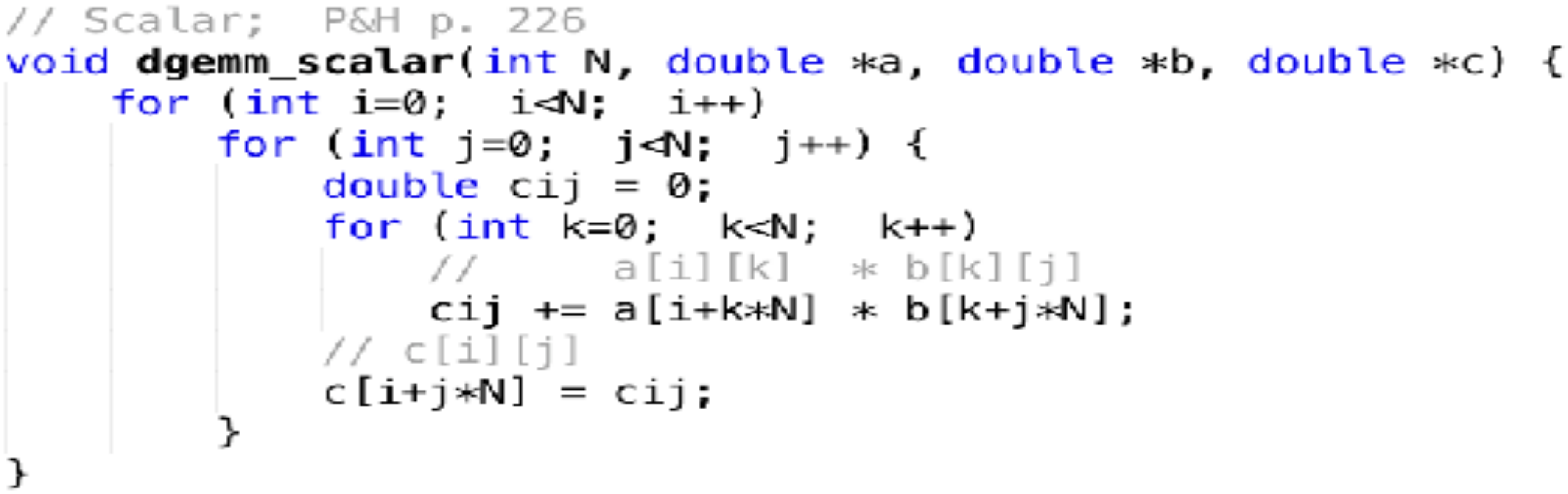
```
def dgemm(N, a, b, c):
    for i in range(N):
        for j in range(N):
            c[i+j*N] = 0
            for k in range(N):
                c[i+j*N] += a[i+k*N] * b[k+j*N]
```

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*N^3 flops

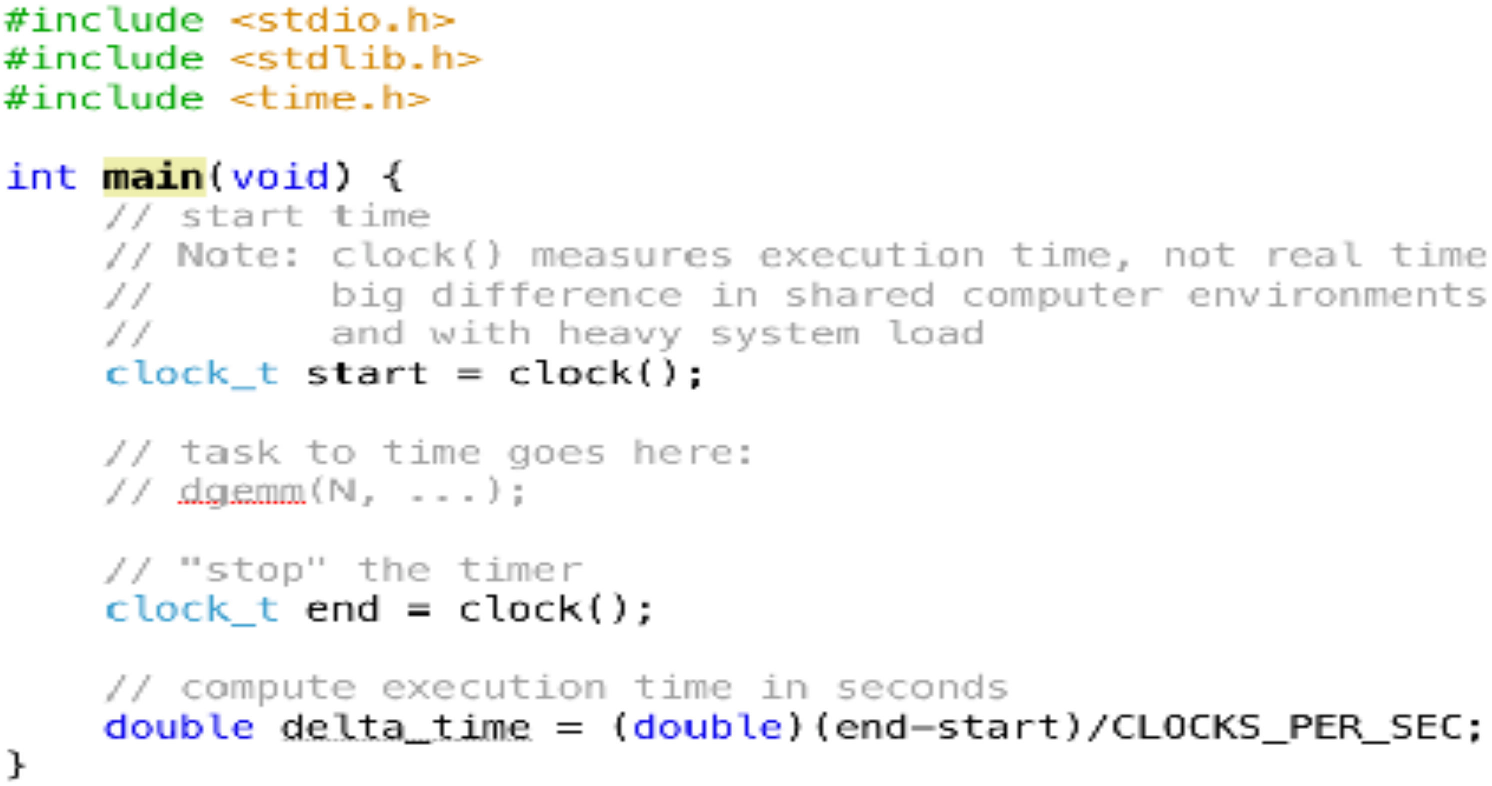
C

- c = a * b
- a, b, c are N x N matrices



```c
// 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

240x!

Which other class gives you this kind of power?

We could stop here … but why? Let’s do better!

Agenda

- 61C – the big picture
- **Parallel processing**
- Single instruction, multiple data
- SIMD matrix multiplication
- Amdahl’s law
- Loop unrolling
- Memory access strategy - blocking
- And in Conclusion, …

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 air-speed limited by economics
    - Use more and larger airplanes to increase throughput
    - (And smaller seats …)

Using Parallelism for Performance

- Two basic approaches to parallelism:
  - 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

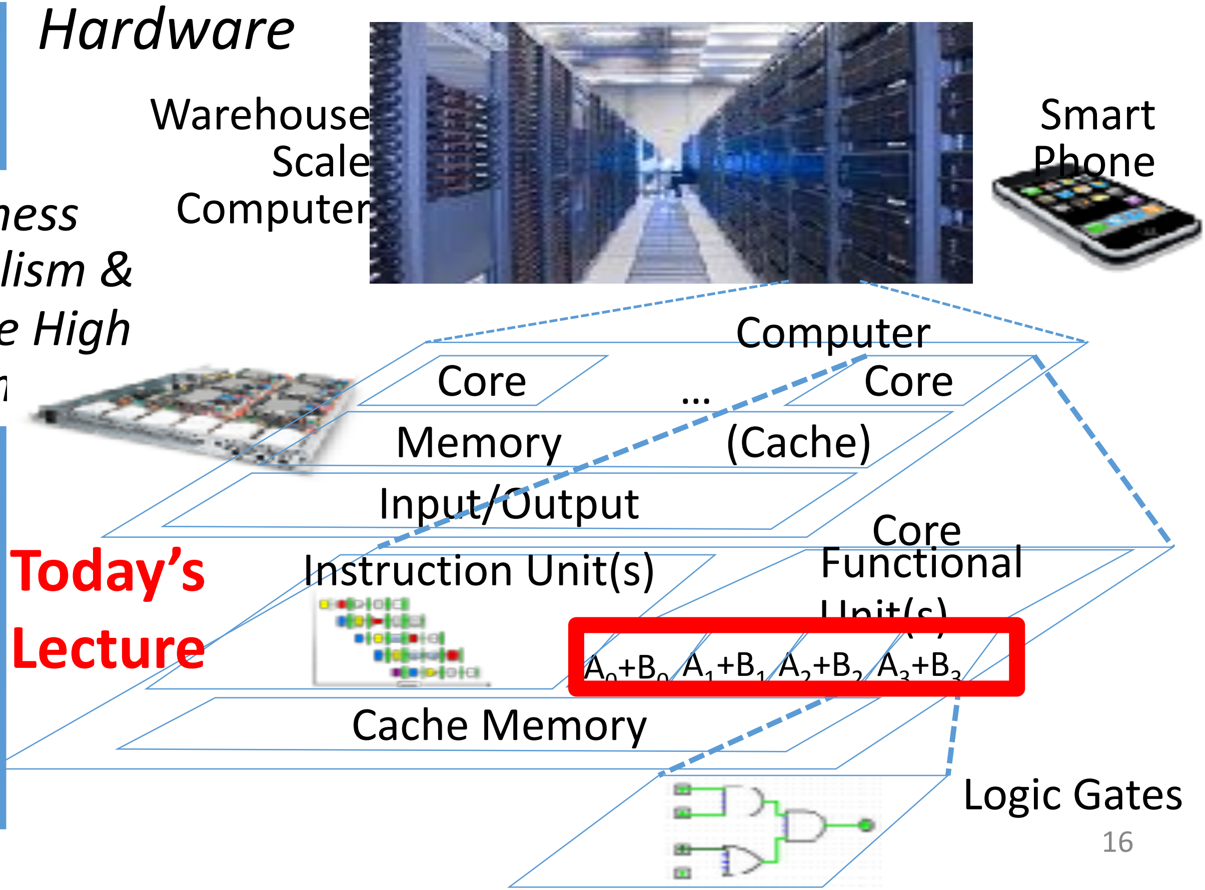
New-School Machine Structures
(It’s a bit more complicated!)

Software | *Hardware*

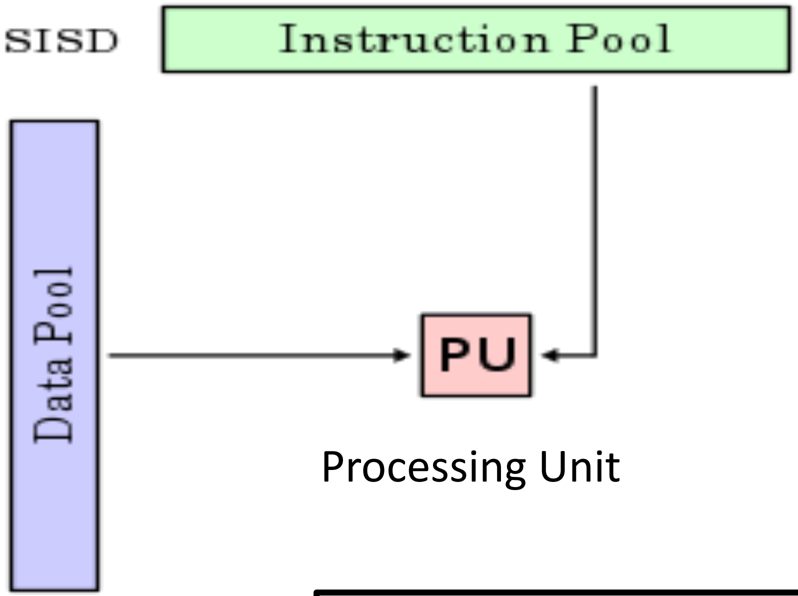
- Parallel Requests
  Assigned to computer
  e.g., Search “Katz”
- Parallel Threads
  Assigned to core
  e.g., Lookup, Ads
- Parallel Instructions
  >1 instruction @ one time
  e.g., 5 pipelined instructions
- Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words
- Hardware descriptions
  All gates @ one time
- Programming Languages

Harness Parallelism & Achieve High Perforn

Today’s Lecture



Single-Instruction/Single-Data Stream (SISD)

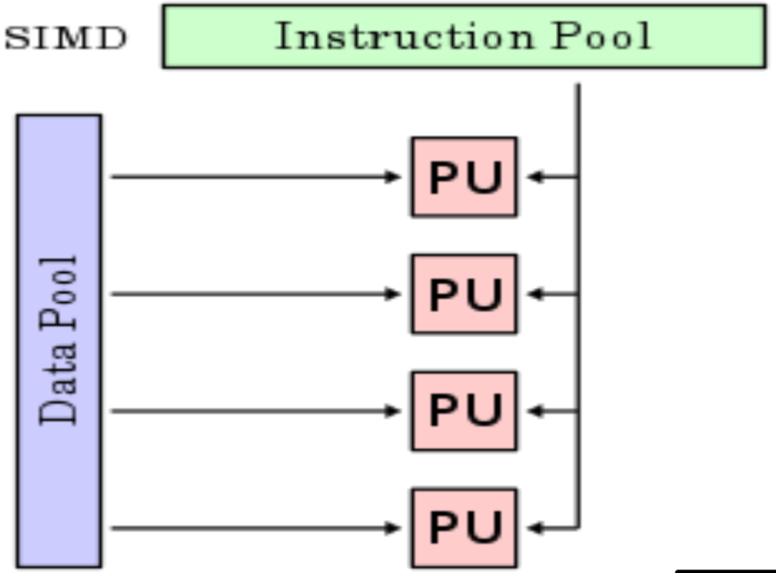


Processing Unit

- 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

This is what we did up to now in 61C

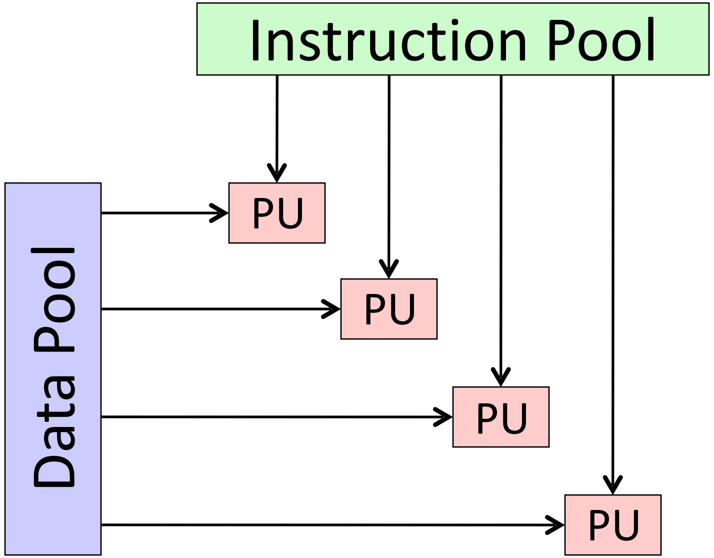
Single-Instruction/Multiple-Data Stream (SIMD or "sim-dee")



- SIMD computer processes multiple data streams using a single instruction stream, e.g., Intel SIMD instruction extensions or NVIDIA Graphics Processing Unit (GPU)

Today's topic.

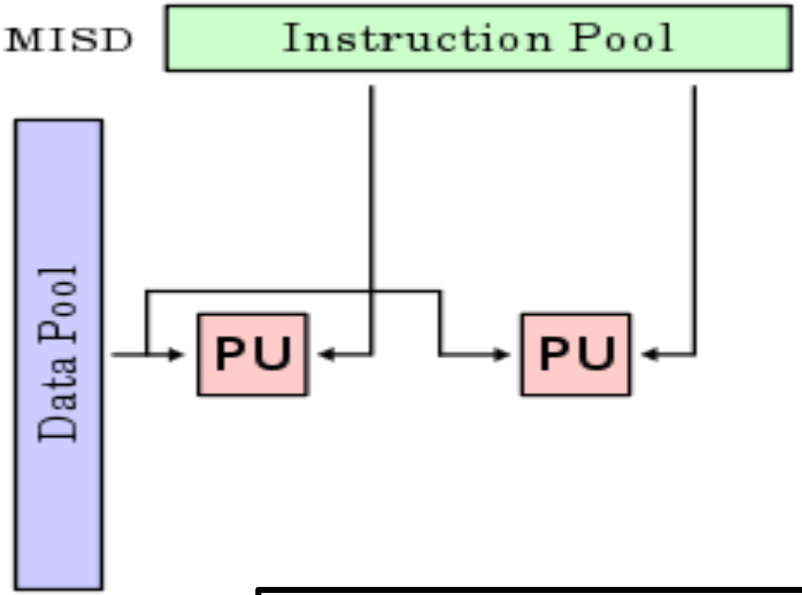
Multiple-Instruction/Multiple-Data Streams (MIMD or “mim-dee”)



- Multiple autonomous processors simultaneously executing different instructions on different data.
    - MIMD architectures include multicore and Warehouse-Scale Computers

Topic of Lecture 19 and beyond.

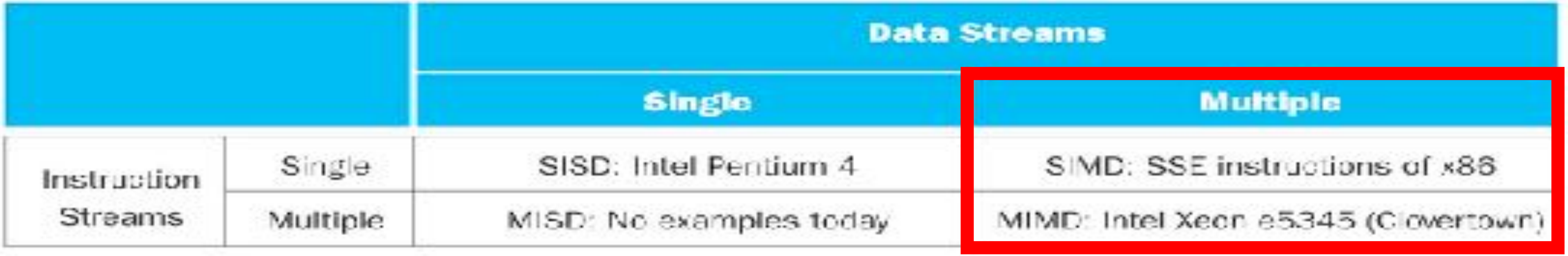
Multiple-Instruction/Single-Data Stream (MISD)



- Multiple-Instruction, Single-Data stream computer that processes multiple instruction streams with a single data stream.
  - Historical significance

This has few applications. Not covered in 61C.

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

*Prof. Michael Flynn, Stanford

Patterson and Hennessy win Turning Award!

acm TURING AWARD SINCE 1966

CELEBRATING 50 YEARS OF COMPUTING'S GREATEST ACHIEVEMENTS

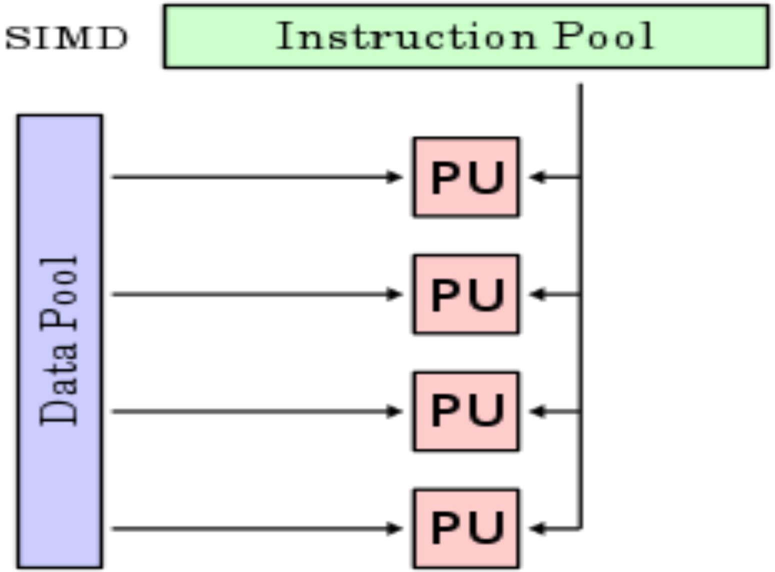
Dave Patterson, right, and John Hennessy in the early 1990s. The men won the Turing Award for their pioneering work on a computer chip design that is now used by most of the tech industry. [illegible]

NEW YORK, NY, March 21, 2018 – ACM, the Association for Computing Machinery, today named John L. Hennessy, former President of Stanford University, and David A. Patterson, retired Professor of the University of California, Berkeley, recipients of the 2017 ACM A.M. Turing Award for pioneering a systematic, quantitative approach to the design and evaluation of computer architectures with enduring impact on the microprocessor industry. Hennessy and Patterson created a systematic and quantitative approach to designing faster, lower power, and reduced instruction set computer (RISC) microprocessors.

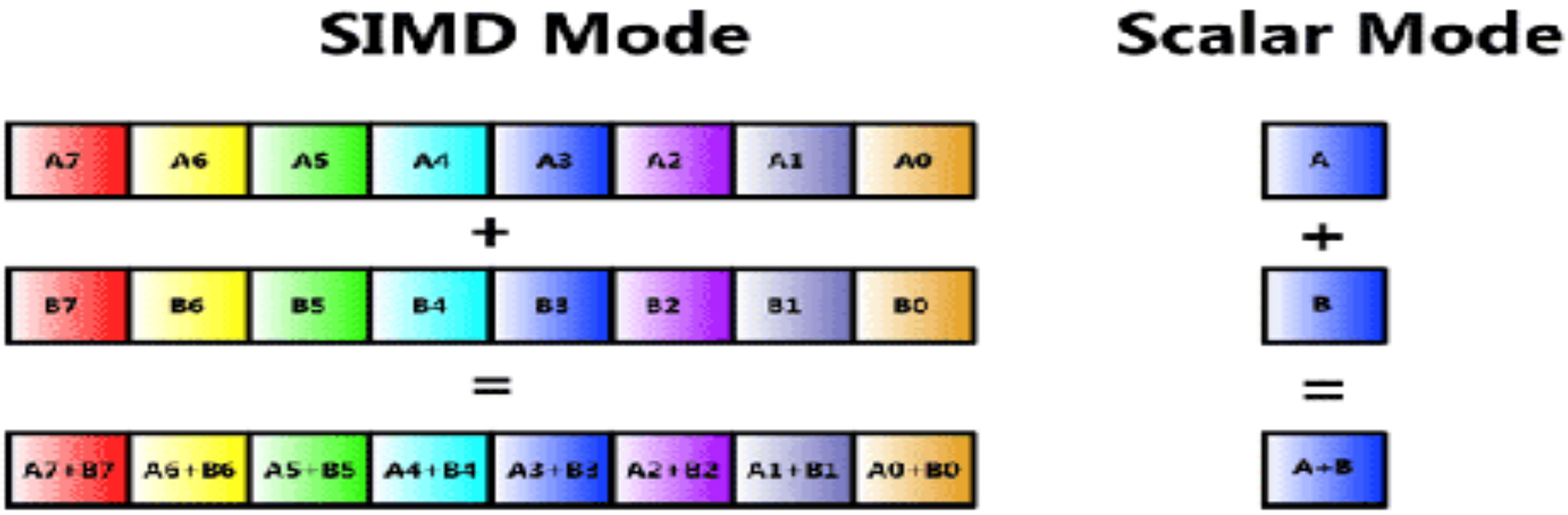
Agenda

- 61C – the big picture
- Parallel processing
- **Single instruction, multiple data**
- SIMD matrix multiplication
- Amdahl’s law
- Loop unrolling
- Memory access strategy - blocking
- And in Conclusion, …

SIMD – "Single Instruction Multiple Data"



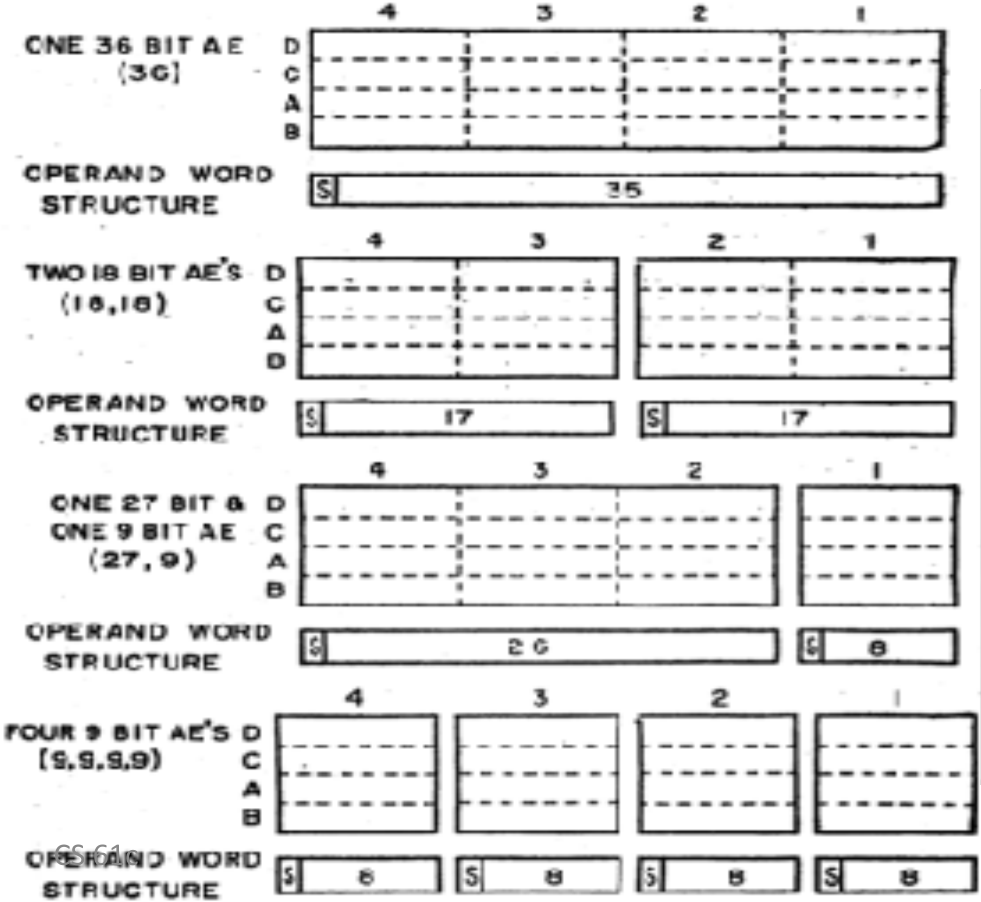
SIMD (Vector) Mode



SIMD Applications & Implementations

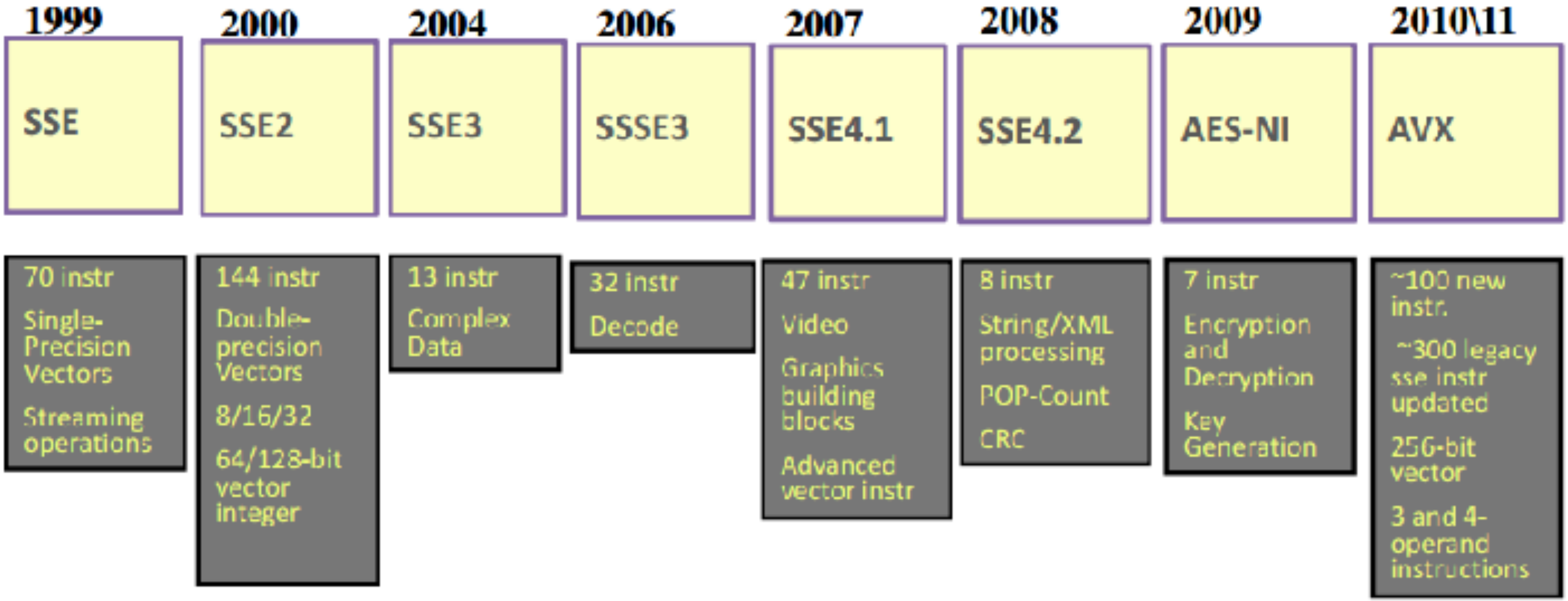
- Applications
  - Scientific computing
    - Matlab, NumPy
  - Graphics and video processing
    - Photoshop, …
  - Big Data
    - Deep learning
  - Gaming
  - …
- Implementations
  - x86
  - ARM
  - RISC-V vector extensions

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



Intel x86 SIMD: Continuous Evolution

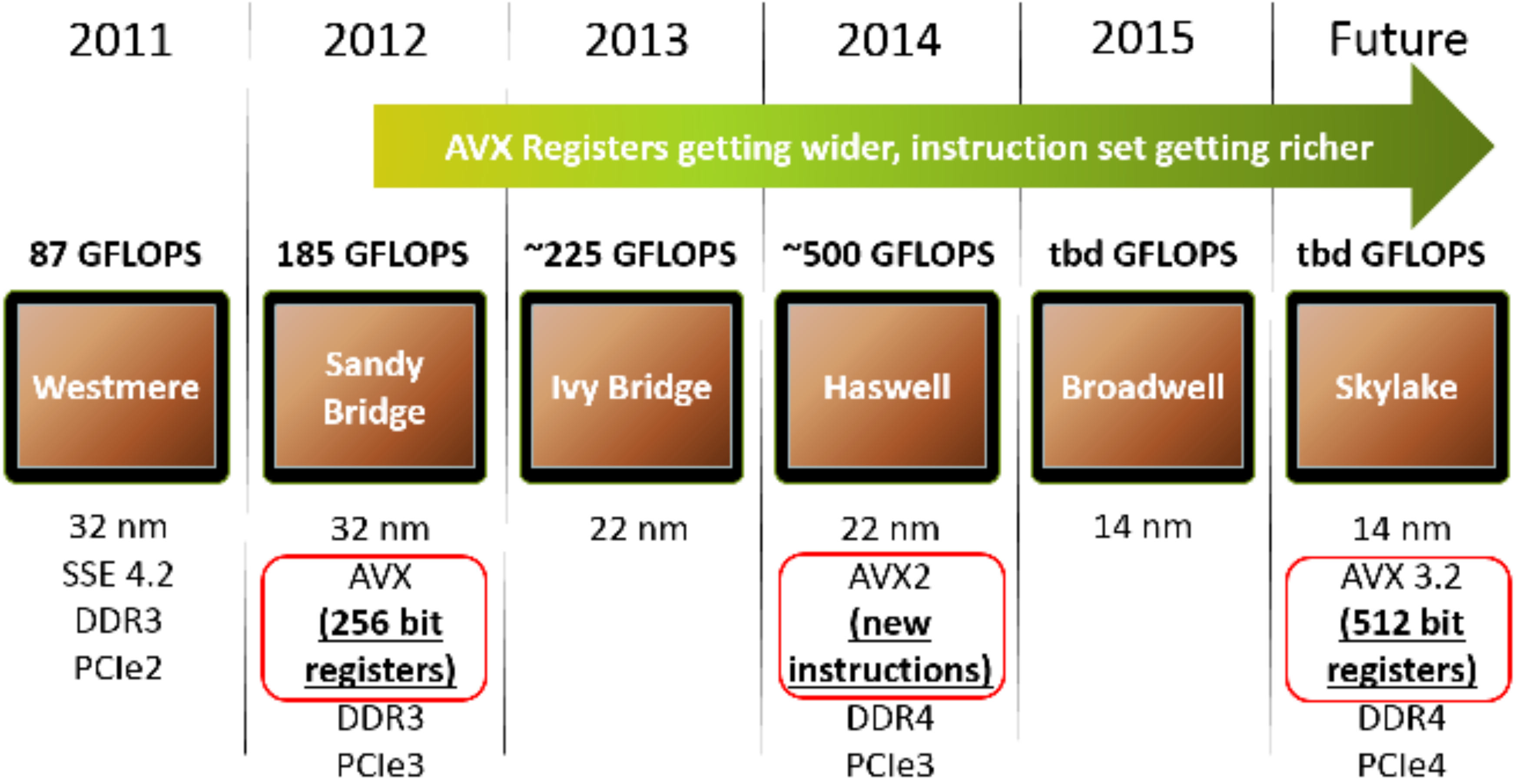
MMX 1997



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 also supported by AMD processors



AVX Registers getting wider, instruction set getting richer

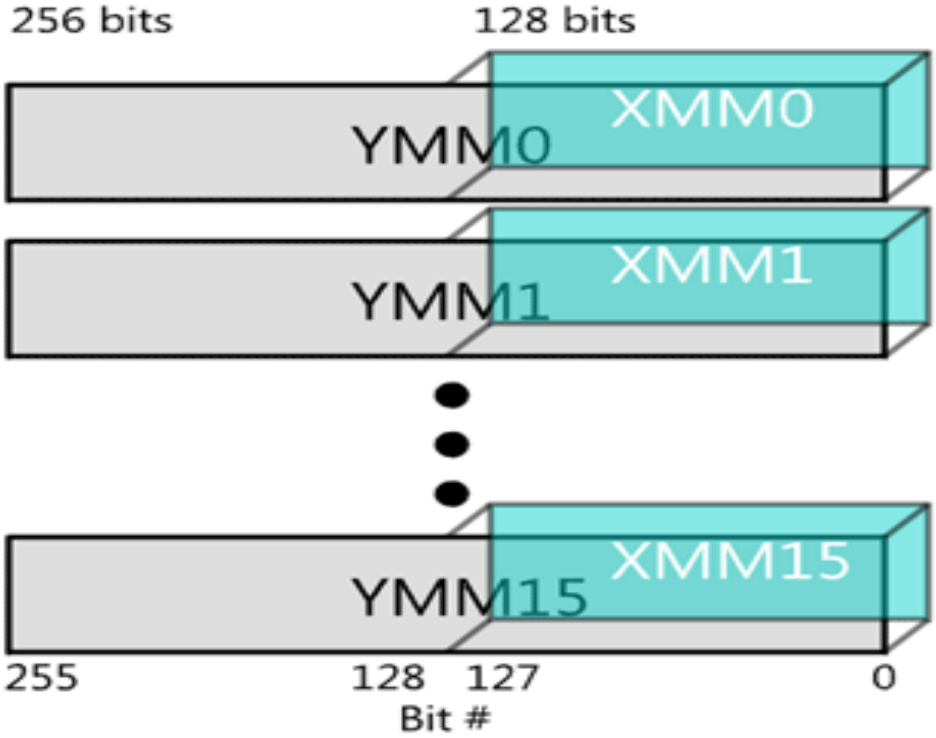
2011	2012	2013	2014	2015	Future
87 GFLOPS	185 GFLOPS	~225 GFLOPS	~500 GFLOPS	tbd GFLOPS	tbd GFLOPS
Westmere	Sandy Bridge	Ivy Bridge	Haswell	Broadwell	Skylake
32 nm SSE 4.2 DDR3 PCIe2	32 nm AVX (256 bit registers) DDR3 PCIe3	22 nm	22 nm AVX2 (new instructions) DDR4 PCIe3	14 nm	14 nm AVX 3.2 (512 bit registers) DDR4 PCIe4

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

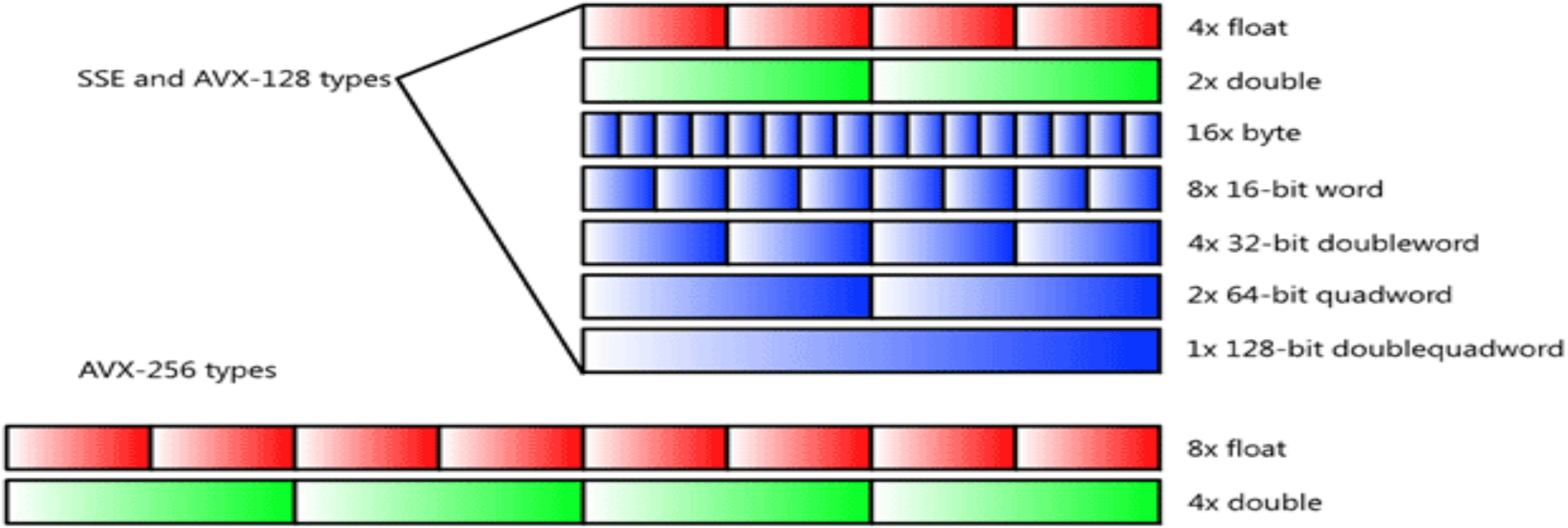
Laptop CPU Specs

```
$ 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
```

AVX SIMD Registers



SIMD Data Types



(Now also AVX-512 available)

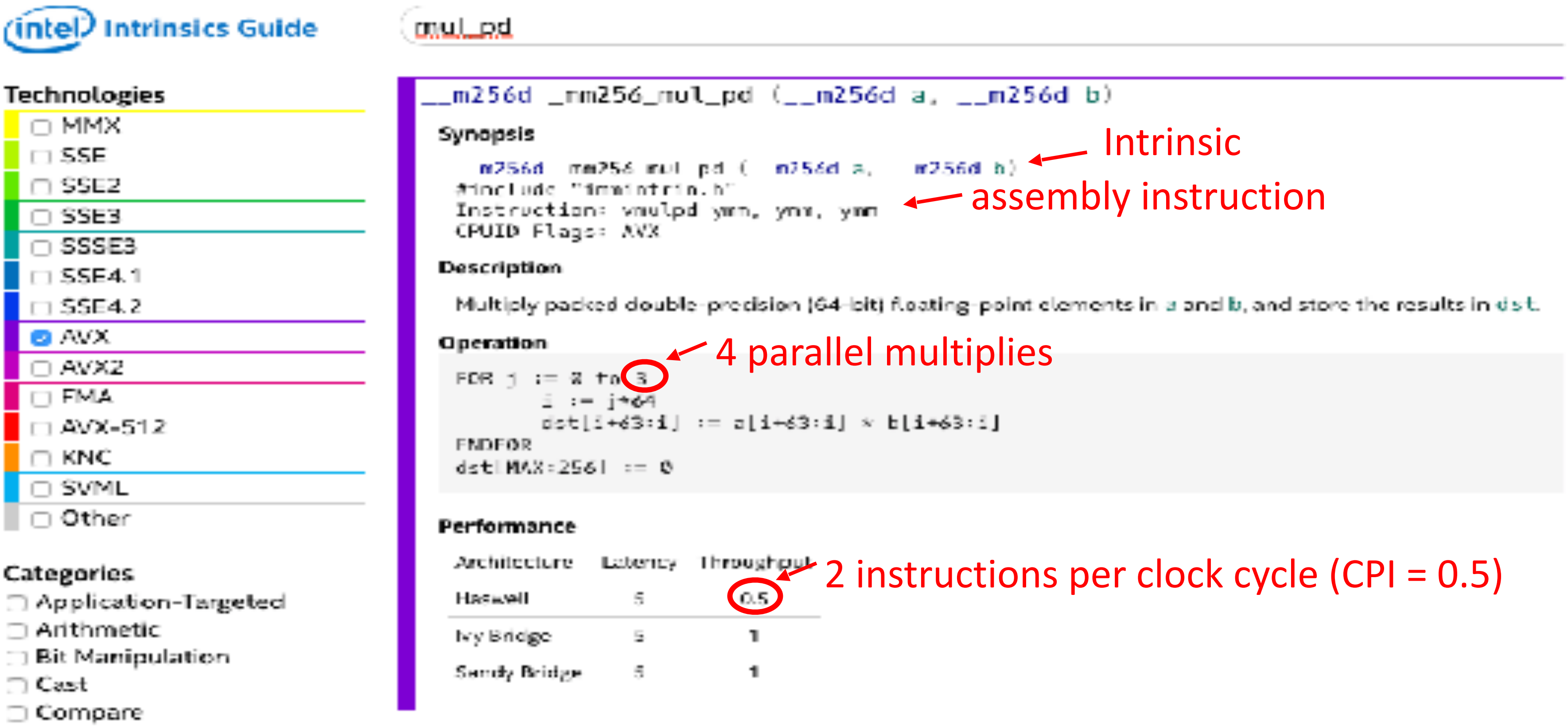
Agenda

- 61C – the big picture
- Parallel processing
- Single instruction, multiple data
- **SIMD matrix multiplication**
- Amdahl’s law
- Loop unrolling
- Memory access strategy - blocking
- And in Conclusion, …

Problem

- Today’s compilers can generate SIMD code
- But in some cases, better results by hand (assembly)
- We will study x86 (not using RISC-V as no vector hardware widely available yet)
  - Over 1000 instructions to learn …
- Can we use the compiler to generate all non-SIMD instructions?

x86 SIMD “Intrinsics”



Intel Intrinsics Guide

mul_pd

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

```
__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

4 parallel multiplies

```
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

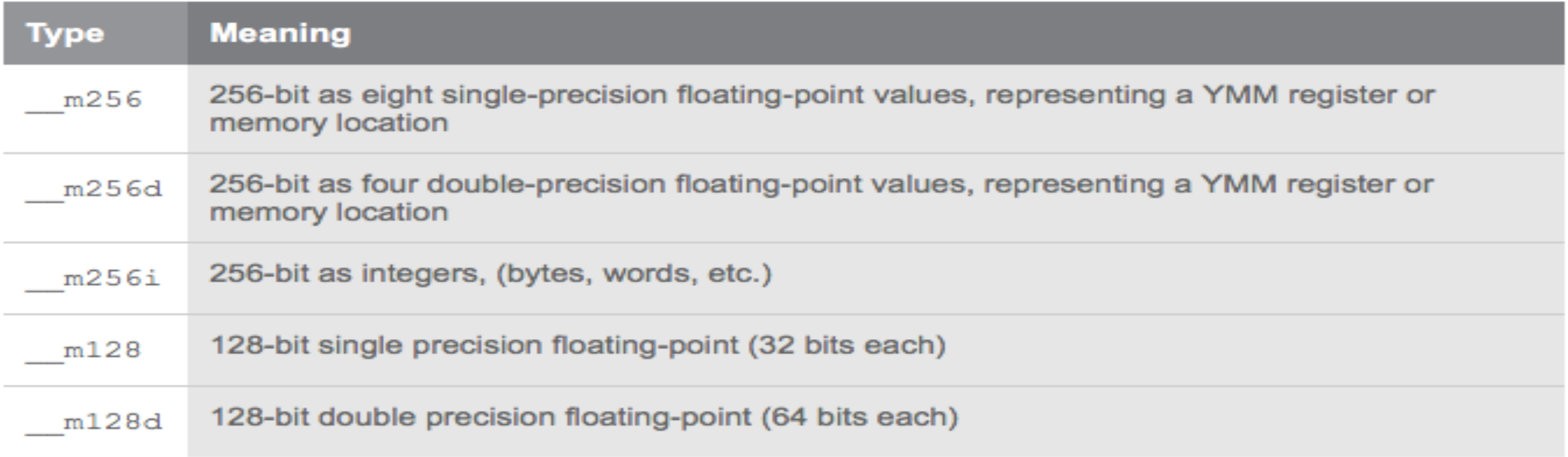
Architecture	Latency	Throughput
Haswell	5	0.5
Ivy Bridge	5	1
Sandy Bridge	5	1

2 instructions per clock cycle (CPI = 0.5)

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

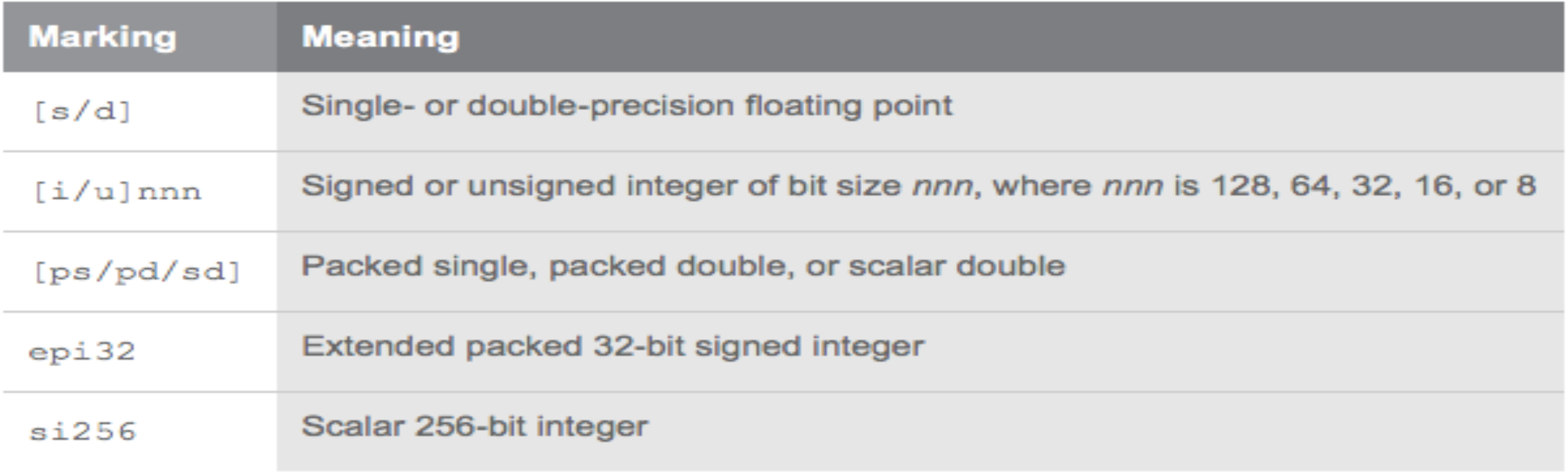
x86 Intrinsics AVX Data Types

Intrinsics: Direct access to assembly from C



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

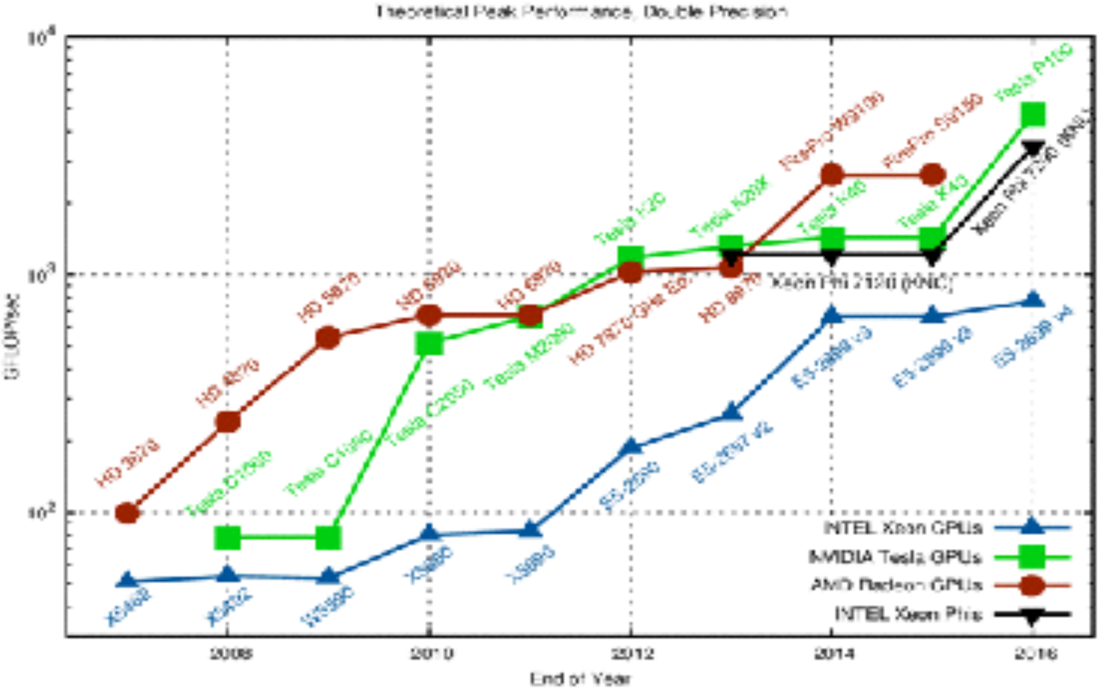
Intrinsics AVX Code Nomenclature



Marking	Meaning
[s/d]	Single- or double-precision floating point
[i/u]nnn	Signed or unsigned integer of bit size *nnn*, where *nnn* 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

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.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

Actual performance is lower because of overhead

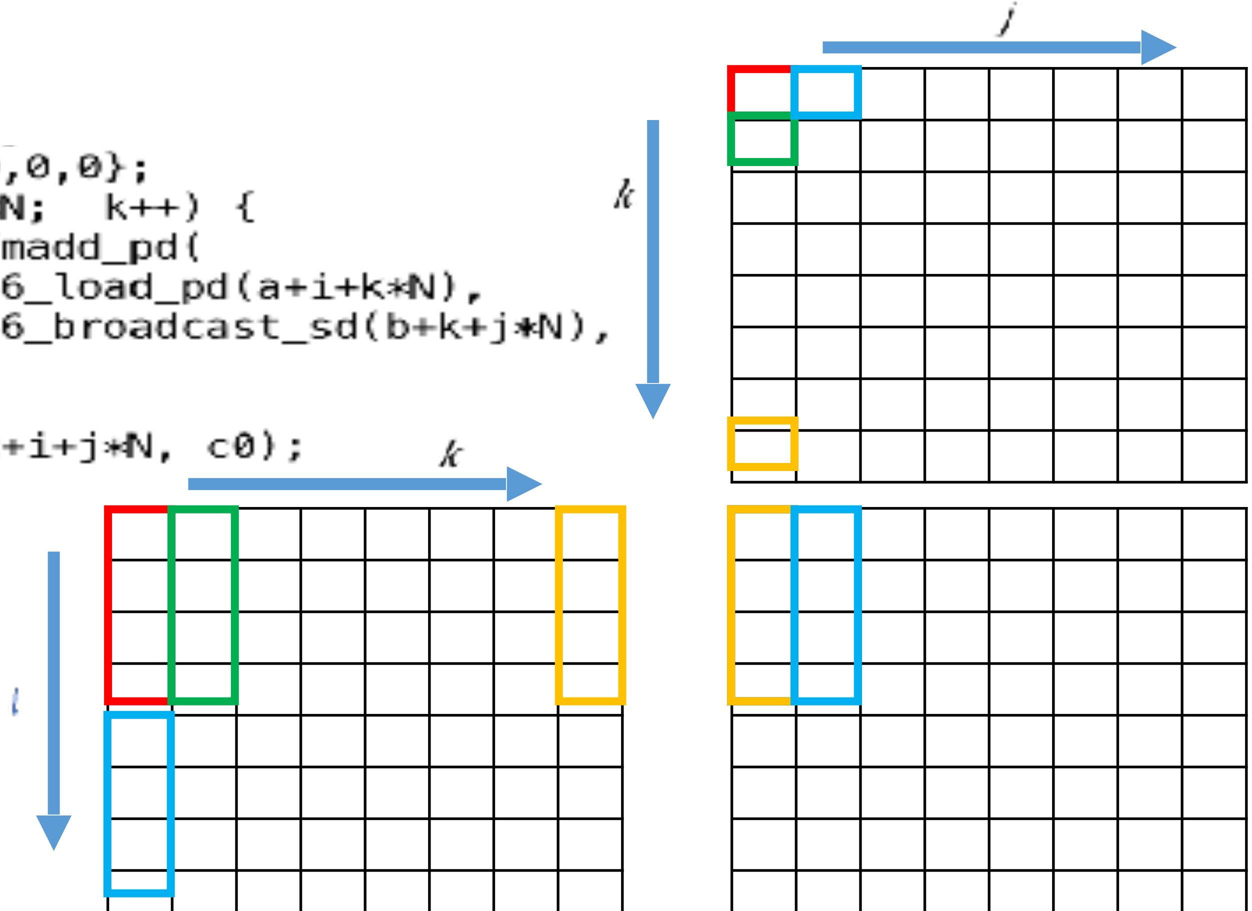
Vectorized Matrix Multiplication

for i …; **i+=4**

for j …

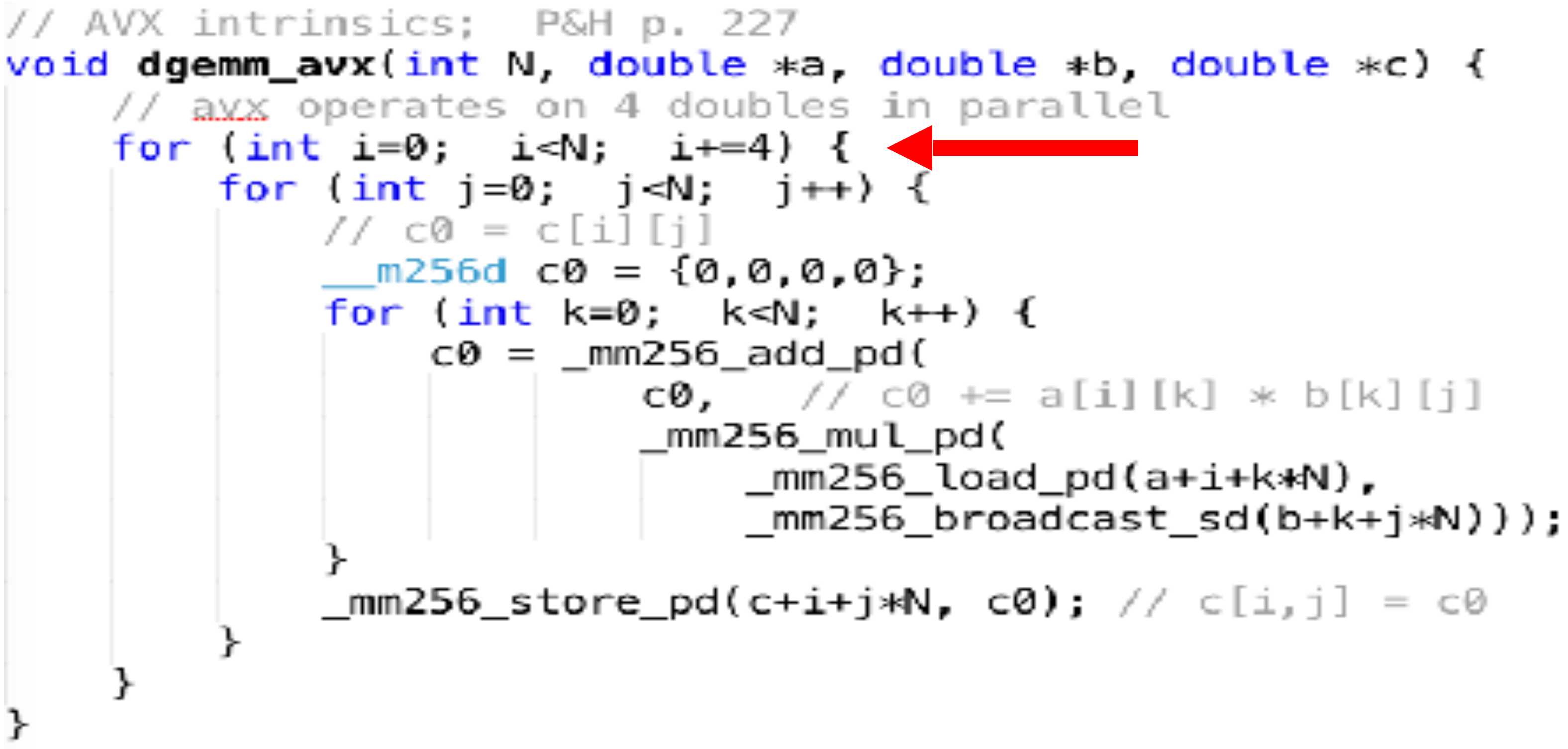
Inner Loop:

```
__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);
```



i += 4

“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!

Agenda

- 61C – the big picture
- Parallel processing
- Single instruction, multiple data
- SIMD matrix multiplication
- **Loop unrolling**
- Memory access strategy - blocking
- And in Conclusion, …

Loop Unrolling

- On high performance processors, optimizing compilers performs “loop unrolling” operation to expose more parallelism and improve performance:

```
for(i=0; i<N; i++)
    x[i] = x[i] + s;
```

- Could become:

```
for(i=0; i<N; 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;
   }
```

1. Expose data-level parallelism for vector (SIMD) instructions or super-scalar multiple instruction issue
2. Mix pipeline with unrelated operations to help with reduce hazards
3. Reduce loop “overhead”
4. Makes code size larger

Amdahl’s Law* applied to **`dgemm`**

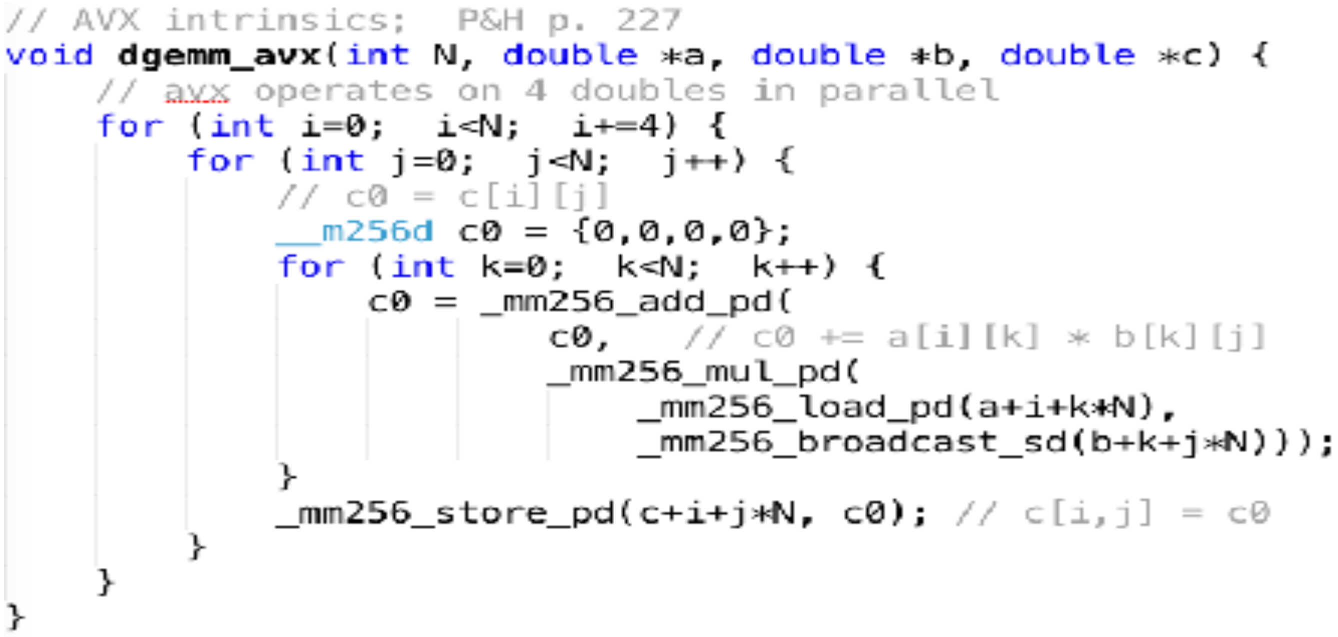
- Measured **`dgemm`** performance

– 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!

** Amdahl’s Law states that the amount a speedup possible through parallelism is limited by the sequential (non-parallel) work.*

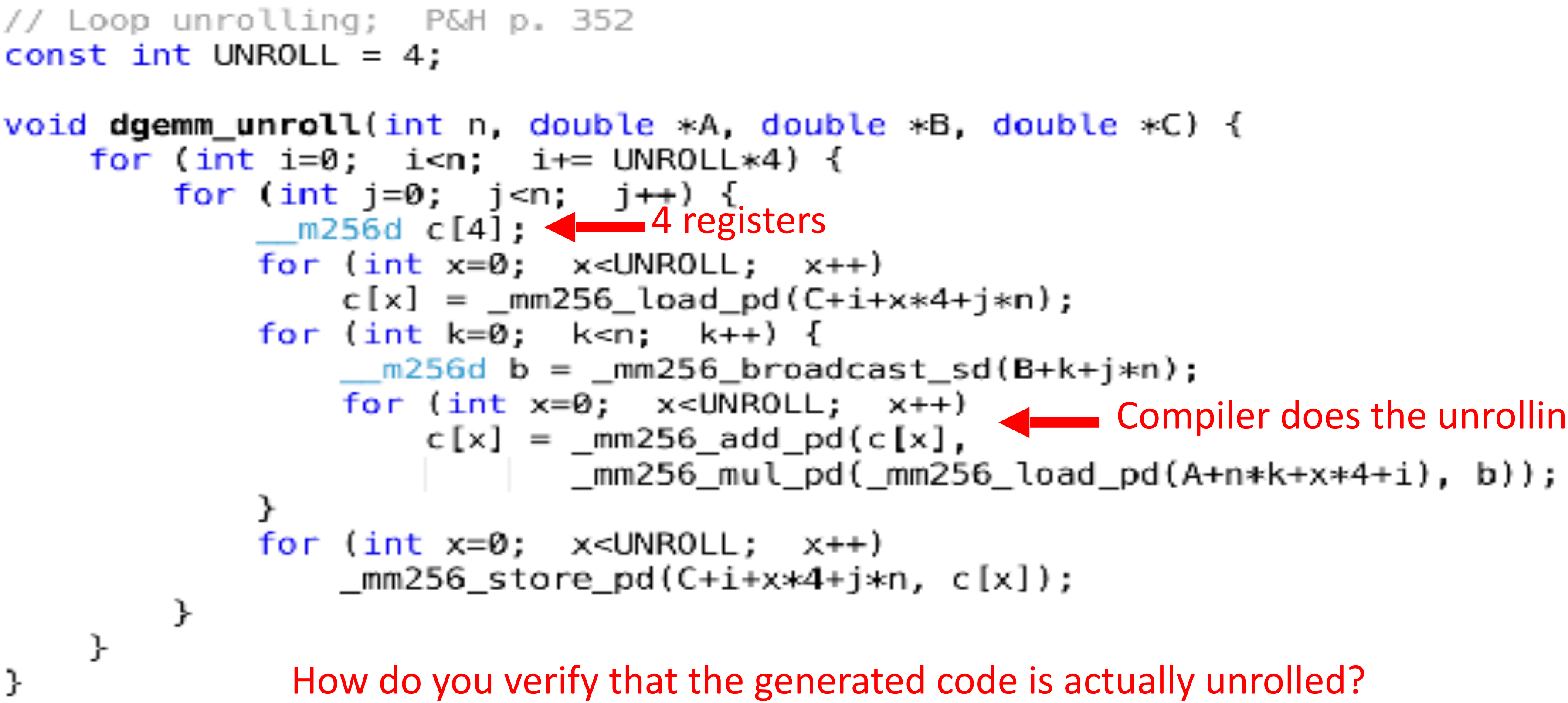
Pipeline Hazards – **dgemm**



```
// 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
        }
    }
}
```

“add_pd” depends on result of “mult_pd” which depends on “load_pd”

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];
            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++)
                    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]);
        }
    }
}
```

4 registers

Compiler does the unrolling

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!

?

Agenda

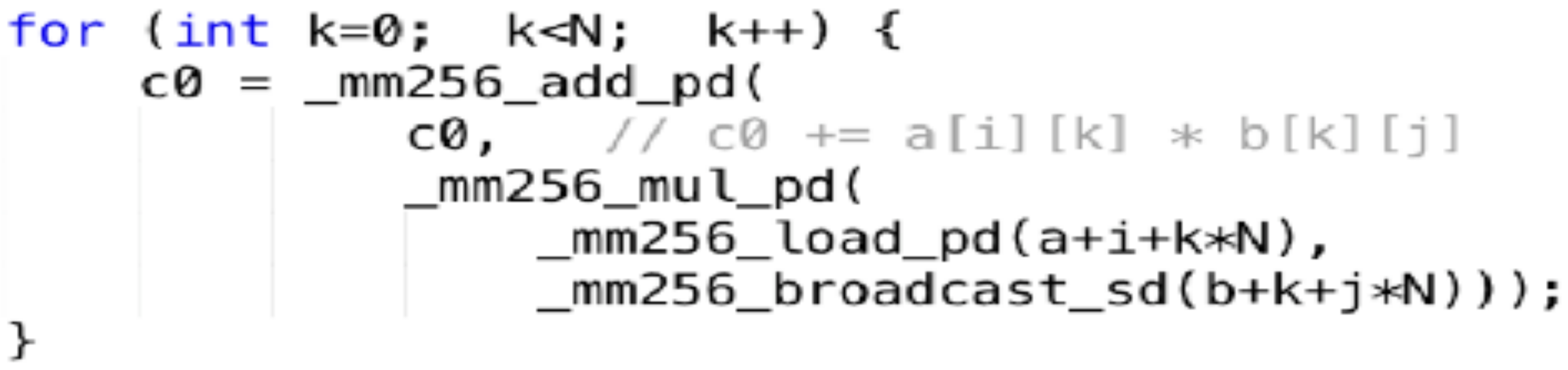
- 61C – the big picture
- Parallel processing
- Single instruction, multiple data
- SIMD matrix multiplication
- Amdahl’s law
- Loop unrolling
- **Memory access strategy - blocking**
- And in Conclusion, …

FPU versus Memory Access

- How many floating-point operations does matrix multiply take?
  - F = 2 x N^3 (N^3 multiplies, N^3 adds)
- How many memory load/stores?
  - M = 3 x 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

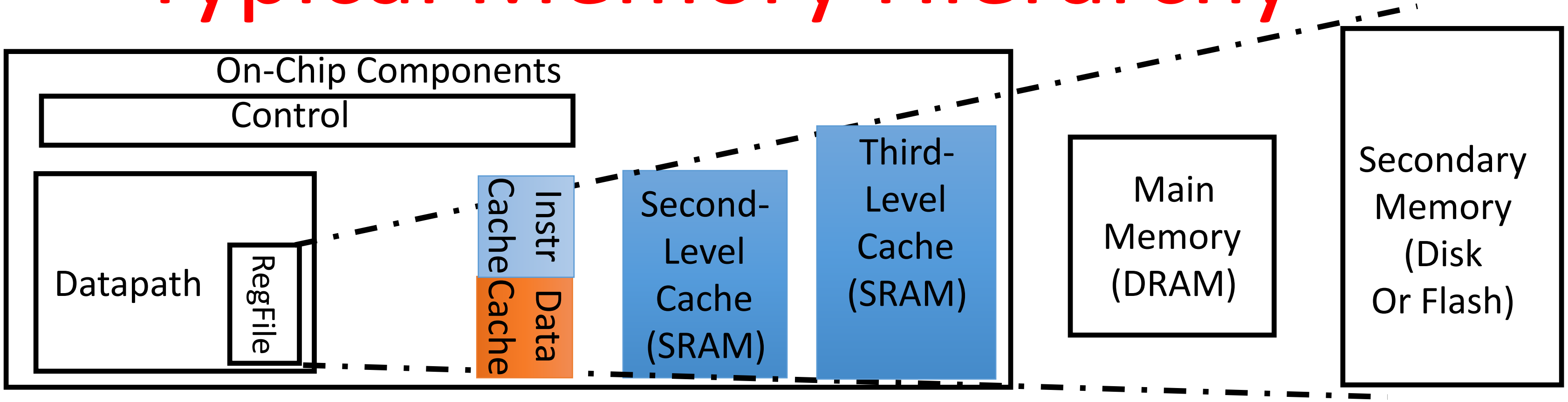
Inner loop:



```
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* = 1.6! (1.25 loads and 2 floating-point operations)

Typical Memory Hierarchy



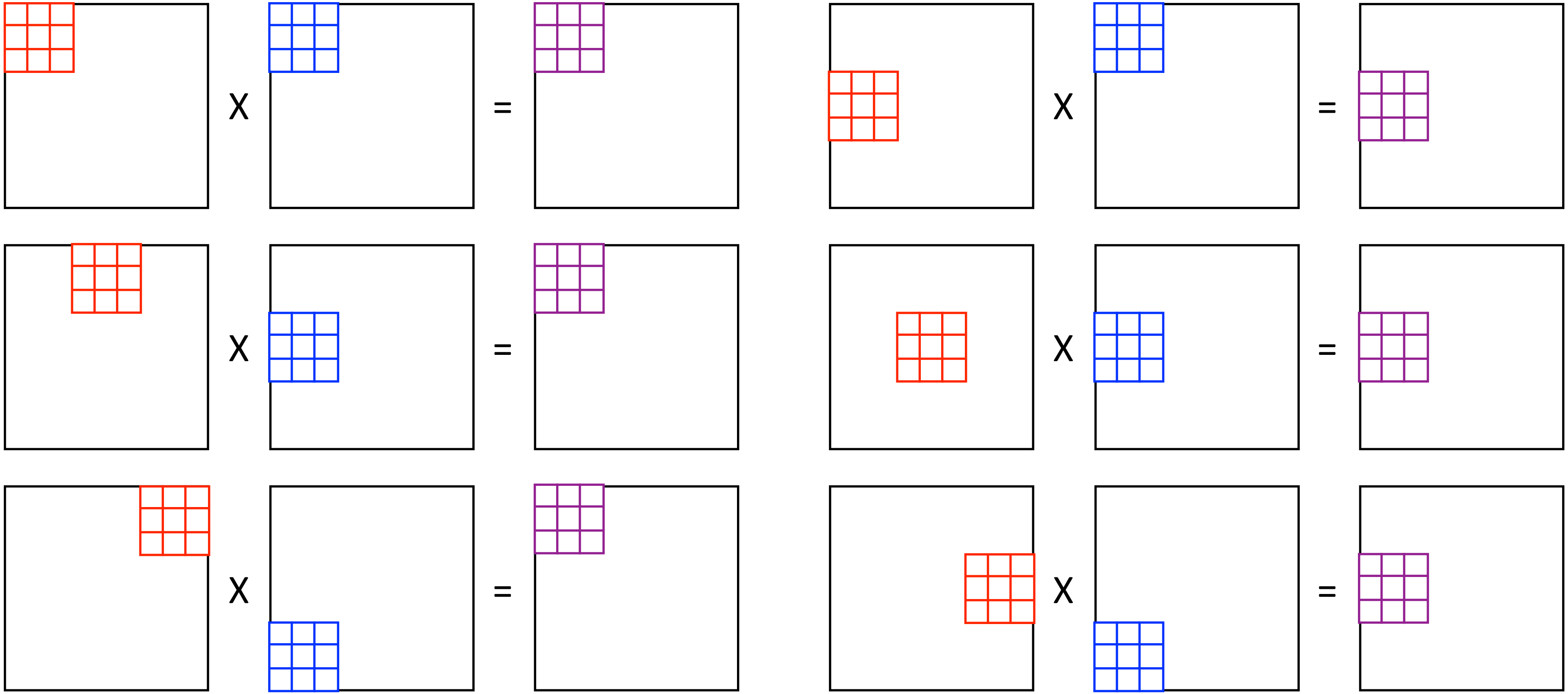
Speed (cycles):	½'s	1's	10's	100's-1000	1,000,000's
Size (bytes):	100's	10K's	M's	G's	T's
Cost/bit:	highest	↔			lowest

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

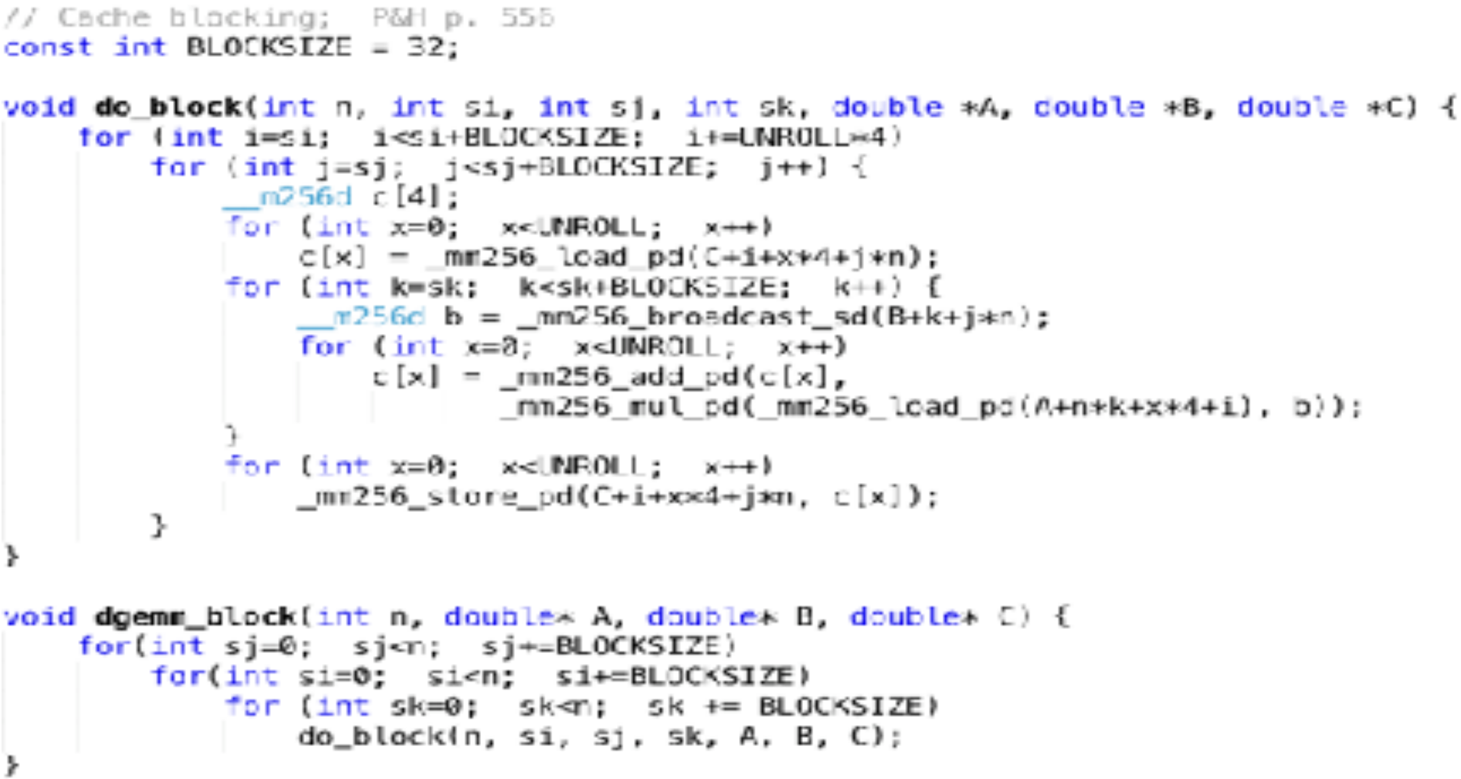
Blocking

- Idea:
  - 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

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

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

Agenda

- 61C – the big picture
- Parallel processing
- Single instruction, multiple data
- SIMD matrix multiplication
- Amdahl's law
- Loop unrolling
- Memory access strategy - blocking
- And in Conclusion, …

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