UC Berkeley
Teaching Professor
Dan Garcia

CS61C

Great Ideas in Computer Architecture (a.k.a. Machine Structures)

UC Berkeley
Professor
Bora Nikolić

Parallelism

New-School Machine Structures

Software

Parallel Requests
Assigned to computer
e.g., Search "Cats"

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 work in parallel at same time

Harness Parallelism & Achieve High Performance

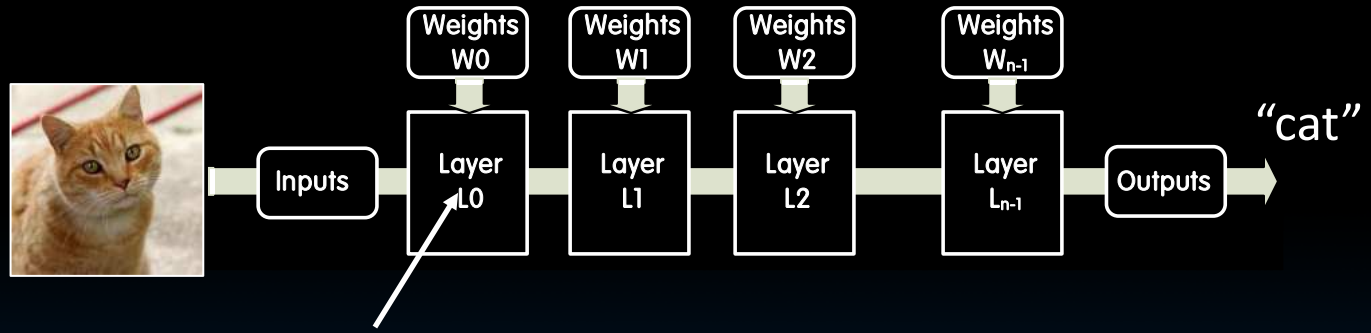
Hardware

Warehouse Scale Computer

Smart Phone

Application: Machine Learning

- Inference in machine learning applications

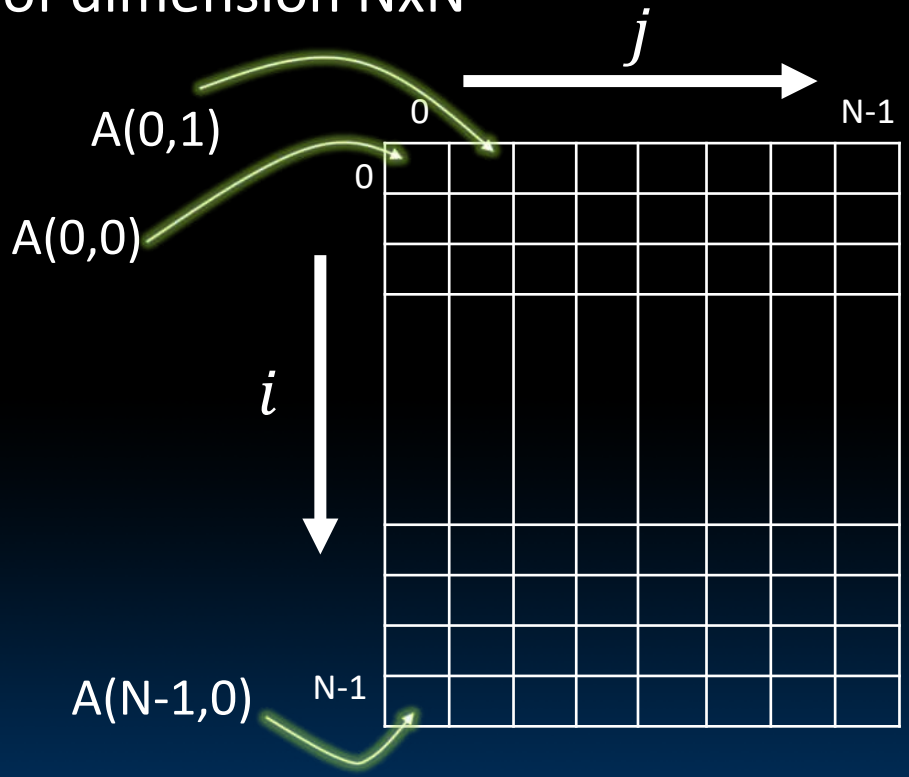


Reference Problem: Matrix Multiplication

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

Matrices

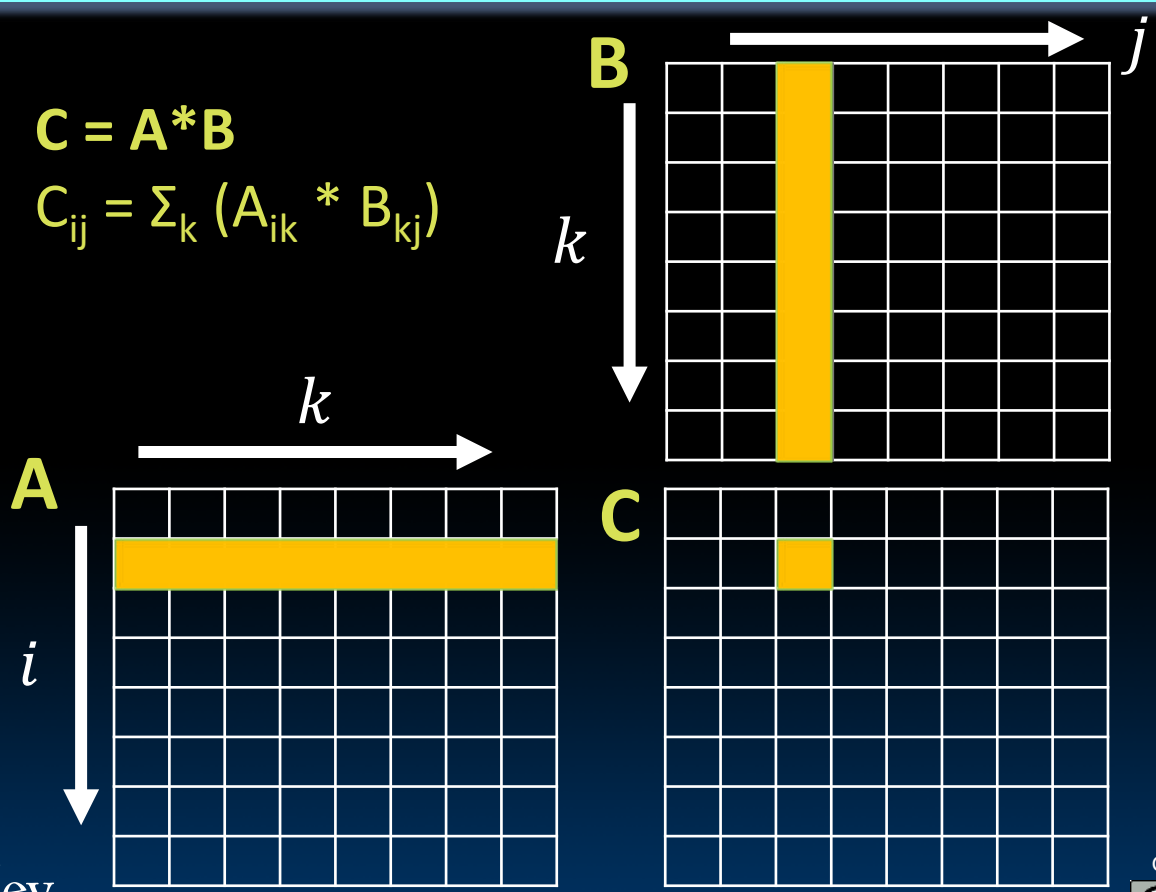
- Square matrix of dimension NxN



Matrix Multiplication

$C = A*B$

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



Matrix Multiplication

Example: 2 x 2 Matrix Multiply

Matrix Multiply:

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

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

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

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 $2*N^3$ FLOPs

C

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

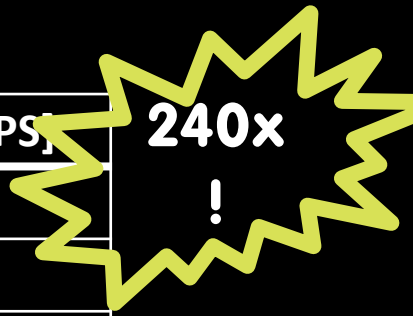
```
// Scalar;  P&H p. 226
void dgemm_scalar(int N, double *a, double *b, double *c) {
    for (int i=0;  i<N;  i++)
        for (int j=0;  j<N;  j++) {
            double cij = 0;
            for (int k=0;  k<N;  k++)
                //     a[i][k]  * b[k][j]
                cij += a[i+k*N] * b[k+j*N];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

Timing Program Execution

```c
#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 vs. 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!

Flynn's Taxonomy

Software vs. Hardware Parallelism

		Software	
		Sequential	**Concurrent**
Hardware	Serial	Matrix Multiply written in MatLab running on an Intel Pentium 4	Windows Vista Operating System running on an Intel Pentium 4
	Parallel	Matrix Multiply written in MATLAB running on an Intel Core i7	Windows Vista Operating System running on an Intel Core i7

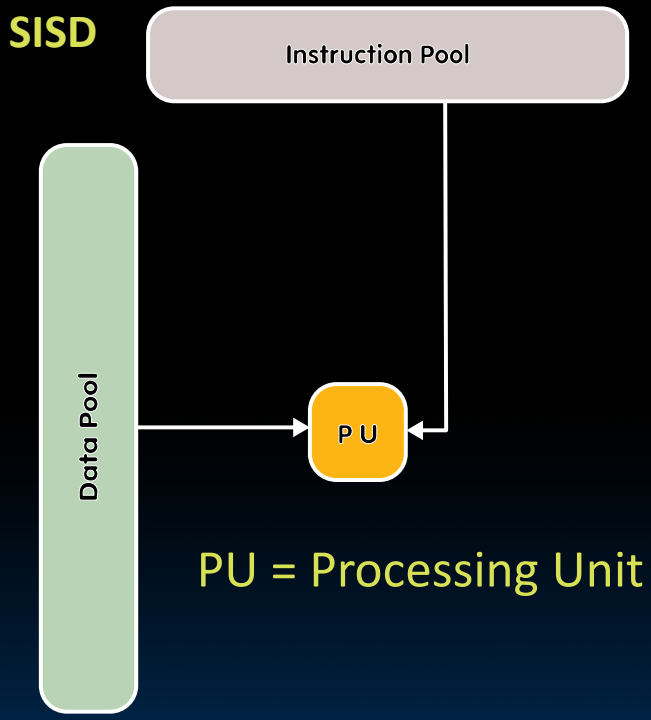
- Choice of hardware and software parallelism are independent
  - Concurrent software can also run on serial hardware
  - Sequential software can also run on parallel hardware
- *Flynn's Taxonomy* is for parallel hardware

Flynn's Taxonomy

		Data Streams: Single	Data Streams: 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 most commonly encountered today
- Most common parallel processing programming style: Single Program Multiple Data ("SPMD")
  - Single program that runs on all processors of an MIMD
  - Cross-processor execution coordination through conditional expressions (will see later in Thread Level Parallelism)
- SIMD: specialized function units (hardware), for handling lock-step calculations involving arrays
  - Scientific computing, machine learning, signal processing, multimedia (audio/video processing)

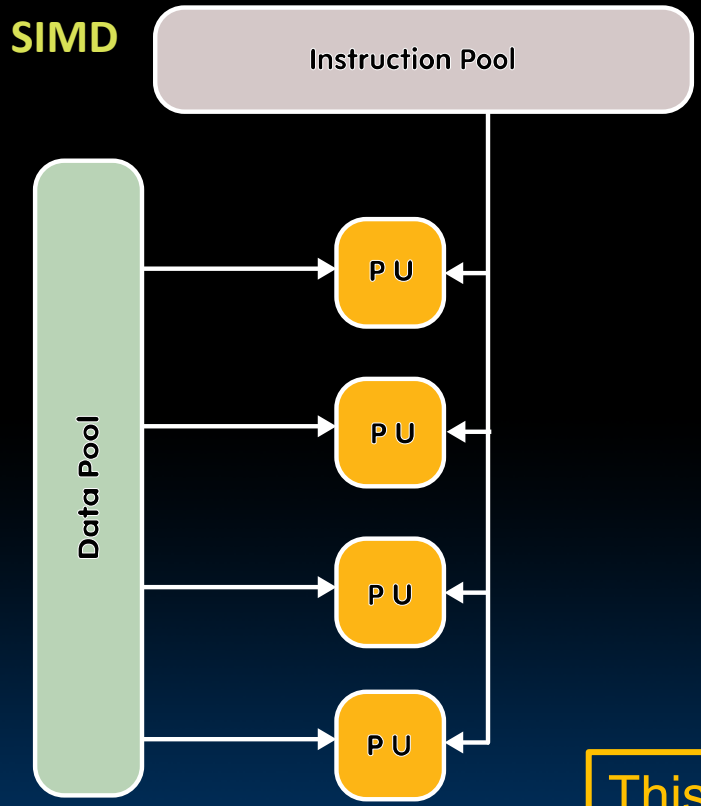
Single Instruction/Single Data Stream (SISD)



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

This is what we did up to now in 61C

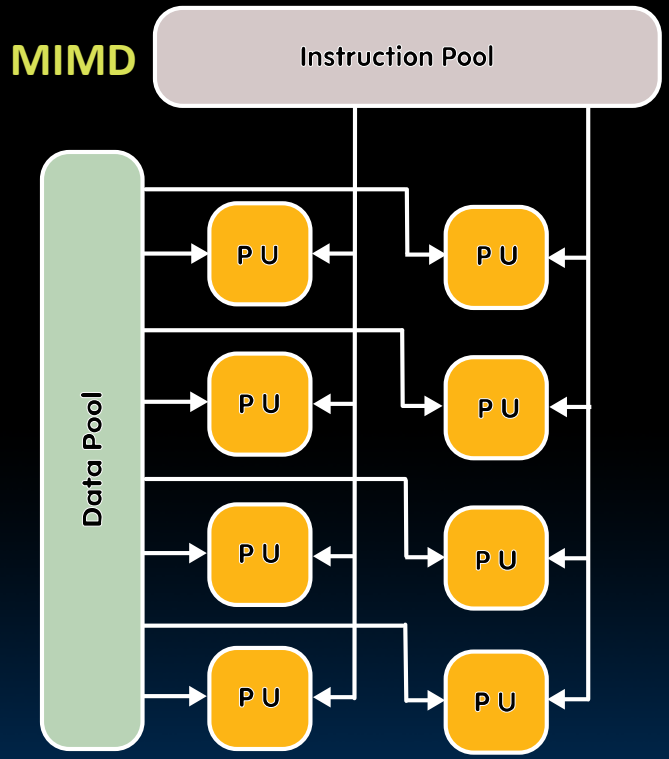
Single Instruction/Multiple Data Stream (SIMD)



- Computer that applies a single instruction stream to multiple data streams for operations that may be naturally parallelized (e.g. SIMD instruction extensions or Graphics Processing Unit)

This segment

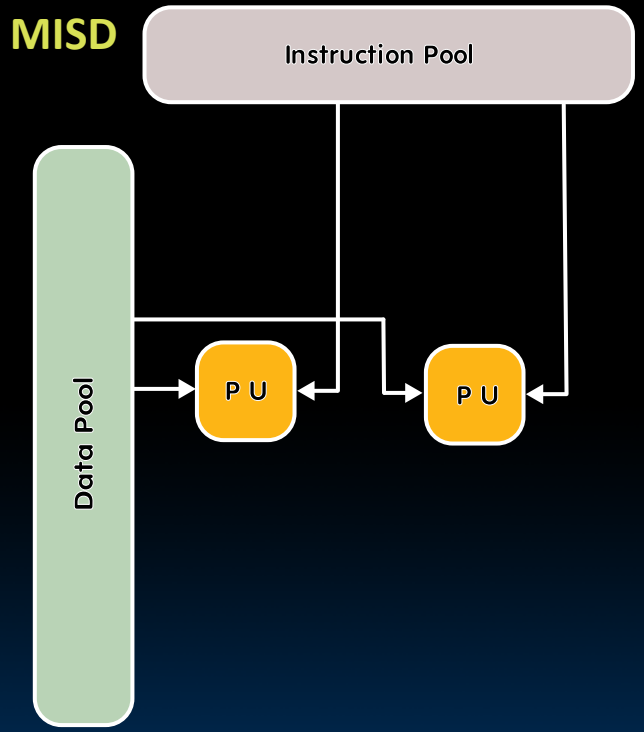
Multiple Instruction/Multiple Data Stream (MIMD)



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

Later in this module

Multiple Instruction/Single Data Stream (MISD)



- Exploits multiple instruction streams against a single data stream for data operations that can be naturally parallelized (e.g. certain kinds of array processors)
- MISD no longer commonly encountered, mainly of historical interest only

This has few applications. Not covered in 61C.

SIMD Architectures

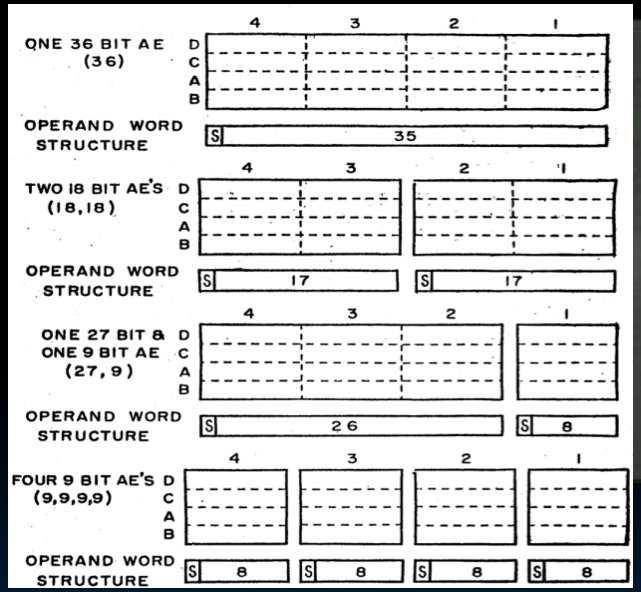
SIMD Architectures

- *Data-Level Parallelism (DLP):* operation on multiple data streams
- **Example:** Multiplying a coefficient vector by a data vector (e.g. in filtering)

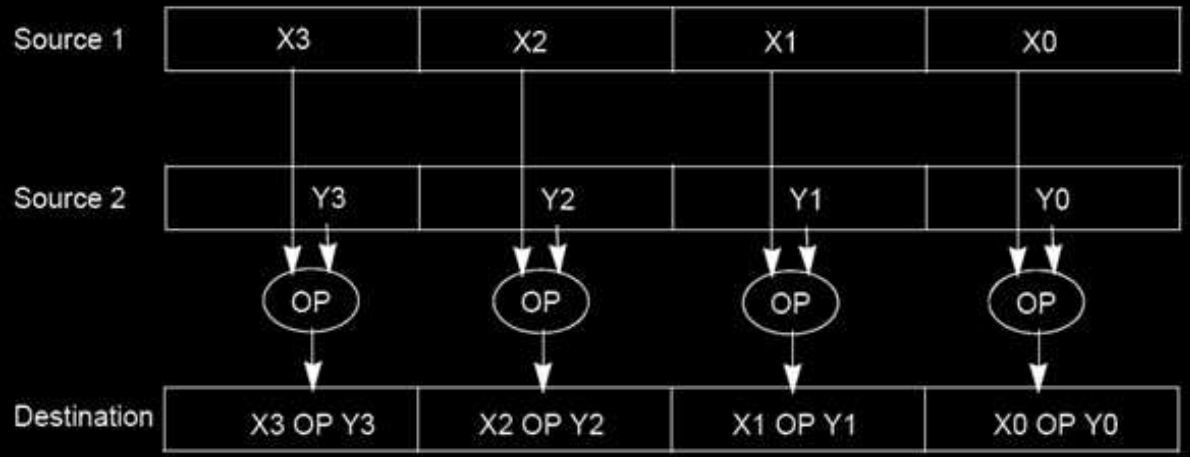
$$y[i] := c[i] \times x[i],\ 0 \le i < n$$

- 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

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



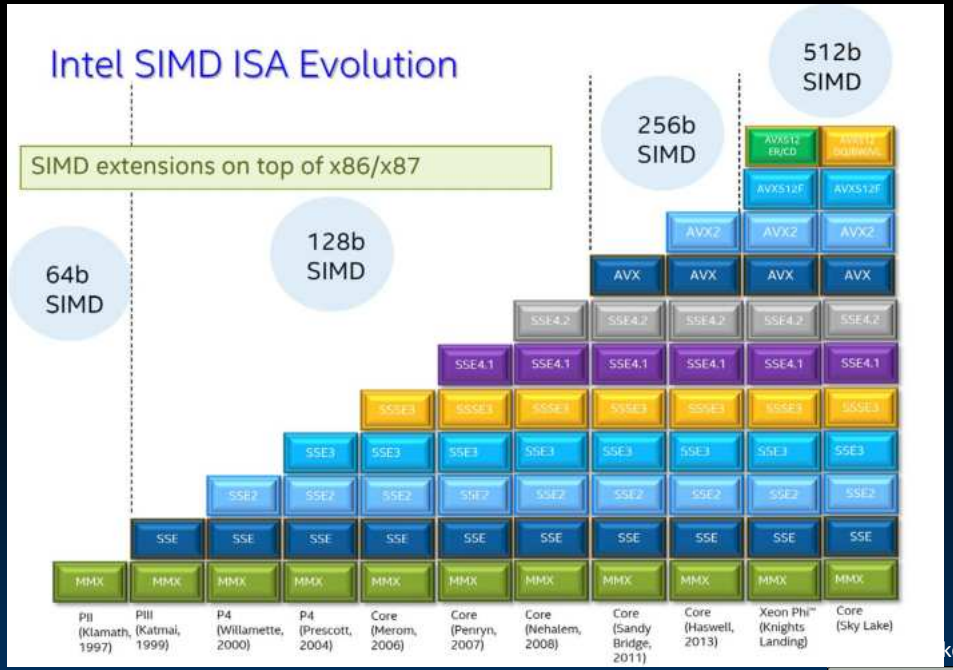
"Advanced Digital Media Boost"



- To improve performance, Intel's SIMD instructions
  - Fetch one instruction, do the work of multiple instructions
  - MMX (MultiMedia eXtension, Pentium II processor family)
  - *SSE (Streaming SIMD Extension, Pentium III and beyond)*

Intel x86 SIMD Evolution

- Started with multimedia extensions (MMX)
  - New instructions every few years
  - New and wider registers
  - More parallelism



C vs. Python

4x!

N	AVX [GFLOPS]	C [GFLOPS]	Python [GFLOPS]
32	4.56	1.30	0.0054
160	5.47	1.30	0.0055
480	5.27	1.32	0.0054
960	3.64	0.91	0.0053

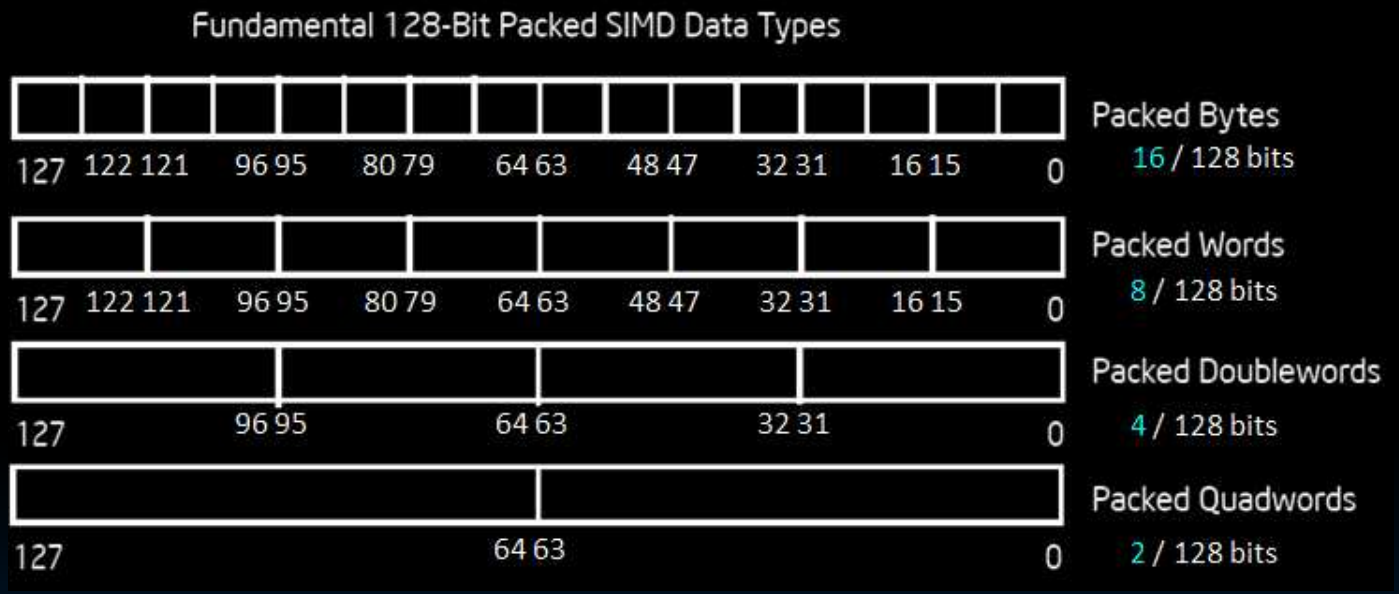
Theoretical Intel i7-5557U performance is ~25 GFLOPS

3.1GHz x 2 instructions/cycle x 4 mults/inst = 24.8GFLOPS

XMM Registers in SSE

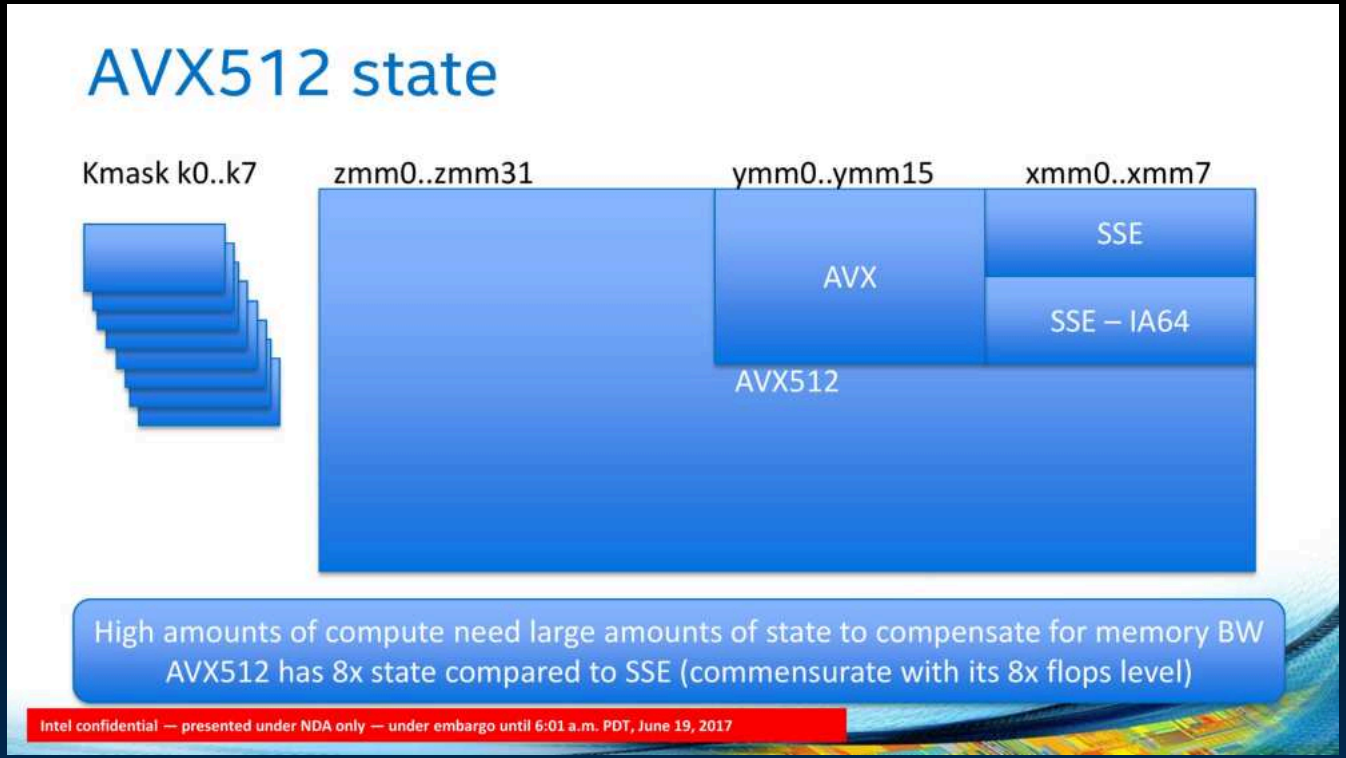
- Architecture extended with eight 128-bit data registers
  - 64-bit address architecture: available as 16 64-bit registers (XMM8 – XMM15)
  - e.g. 128-bit packed single-precision floating-point data type (doublewords), allows four single-precision operations to be performed simultaneously

Intel Architecture SSE2+128-Bit 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)

SIMD Registers in AVX512



Check Out My Laptop (`lscpu`)

```
Model:                           126
Model name:                      Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz
Stepping:                        5
CPU MHz:                         1497.605
BogoMIPS:                        2995.21
Hypervisor vendor:               Microsoft
Virtualization type:             full
L1d cache:                       192 KiB
L1i cache:                       128 KiB
L2 cache:                        2 MiB
L3 cache:                        8 MiB
Vulnerability Itlb multihit:     KVM: Vulnerable
Vulnerability L1tf:              Not affected
Vulnerability Mds:               Not affected
Vulnerability Meltdown:          Not affected
Vulnerability Spec store bypass: Mitigation; Speculative Store Bypass disabled via prctl and seccomp
Vulnerability Spectre v1:        Mitigation; usercopy/swapgs barriers and __user pointer sanitization
Vulnerability Spectre v2:        Mitigation; Enhanced IBRS, IBPB conditional, RSB filling
Vulnerability Srbds:             Not affected
Vulnerability Tsx async abort:   Not affected
Flags:                           fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxs
                                 r sse sse2 ss ht syscall nx pdpe1gb rdtscp lm constant_tsc rep_good nopl xtopology cpui
                                 d pni pclmulqdq ssse3 fma cx16 pcid sse4_1 sse4_2 movbe popcnt aes xsave avx f16c rdran
                                 d hypervisor lahf_lm abm 3dnowprefetch invpcid_single ssbd ibrs ibpb stibp ibrs_enhance
                                 d fsgsbase bmi1 avx2 smep bmi2 erms invpcid avx512f avx512dq rdseed adx smap avx512ifma
                                  clflushopt avx512cd sha_ni avx512bw avx512vl xsaveopt xsavec xgetbv1 xsaves avx512vbmi
                                  umip avx512_vbmi2 gfni vaes vpclmulqdq avx512_vnni avx512_bitalg avx512_vpopcntdq rdpi
                                 d flush_l1d arch_capabilities
```

SIMD Array Processing

Example: SIMD Array Processing

```
for each f in array:
    f = sqrt(f)
```

pseudocode

```
for each f in array {
    load f to the floating-point register
    calculate the square root
    write the result from the register to memory
}
```

SISD

```
for every 4 members in array {
    load 4 members to the SSE register
    calculate 4 square roots in one operation
    write the result from the register to memory
}
```

SIMD

Example: Add Single-Precision FP 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:

```
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
```

move from mem to XMM register
memory aligned, packed single precision

add from mem to XMM register
packed single precision

move from XMM register to mem
memory aligned, packed single precision

Intel SSE Intrinsics

- Intrinsics are C functions and procedures for putting in assembly language, including SSE instructions
  - With intrinsics, can program using these instructions indirectly
  - One-to-one correspondence between SSE instructions and intrinsics

Instrinsics:	Corresponding SSE instructions:
• Vector data type:	
_m128d	
• Load and store operations:	
_mm_load_pd	MOVAPD/aligned, packed double
_mm_store_pd	MOVAPD/aligned, packed double
• Arithmetic:	
_mm_add_pd	ADDPD/add, packed double
_mm_mul_pd	MULPD/multiple, packed double

Matrix Multiply Example

Example: 2 x 2 Matrix Multiply

Matrix Multiply:

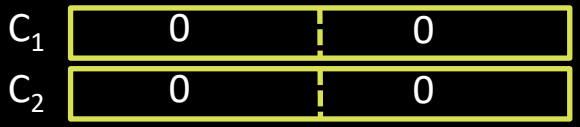
$$C_{i,j} = (A \times B)_{i,j} = \sum_{k=1}^{2} A_{i,k} \times B_{k,j}$$

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

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

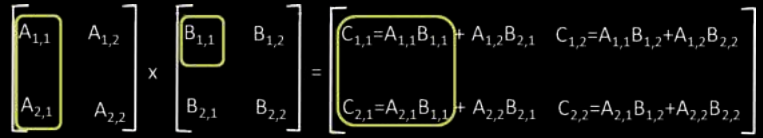
Example: 2 x 2 Matrix Multiply

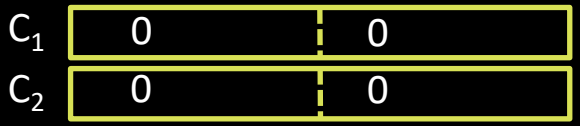
- Initialization



Example: 2 x 2 Matrix Multiply

- Initialization



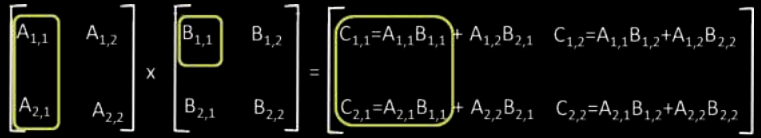


- I = 1

_mm_load_pd: Stored in memory in Column order

_mm_load1_pd: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register

Example: 2 x 2 Matrix Multiply



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

- I = 1, intermediate result

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

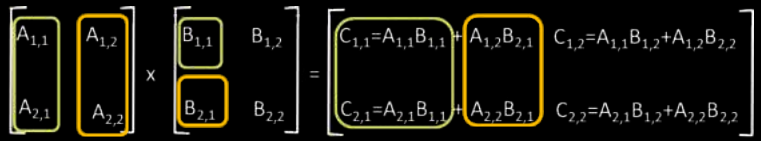
A	$A_{1,1}$	$A_{2,1}$

_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

Example: 2 x 2 Matrix Multiply



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, intermediate result

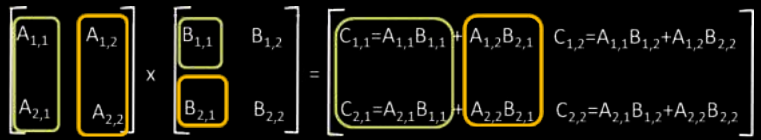
A	$A_{1,2}$	$A_{2,2}$

_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

Example: 2 x 2 Matrix Multiply



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

- I = 2, intermediate result

```
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

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

Example: 2 x 2 Matrix Multiply

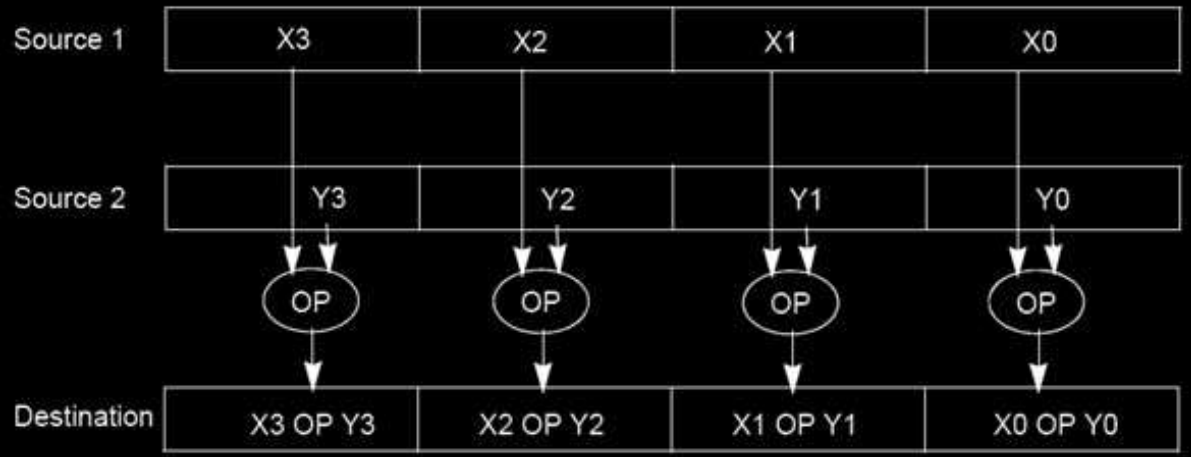
```
#include <stdio.h>
// header file for SSE compiler intrinsics
#include <emmintrin.h>

// 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

```
// 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;
}
```

Back to RISC-V: Vector Extensions (Draft)



- To improve RISC-V performance, add SIMD instructions (and hardware) – V extension
  - Fetch one instruction, do the work of multiple instructions
  - OP denotes a vector instruction, prefix v – vector register
  - **`vadd vd, vs1, vs2`** (adds two vectors stored in vector registers)
  - Assume vectors are 512-bits wide

"And in Conclusion..."

- Flynn Taxonomy of Parallel Architectures
  - SIMD: Single Instruction Multiple Data
  - MIMD: Multiple Instruction Multiple Data
  - SISD: Single Instruction Single Data
  - MISD: Multiple Instruction Single Data (unused)
- Intel AVX SIMD Instructions
  - One instruction fetch that operates on multiple operands simultaneously
  - 512/256/128/64-bit AVX registers
  - Use C intrinsics