# Networking & Parallelism 1

Berkeley EECS
ELECTRICAL ENGINEERING & COMPUTER SCIENCES

# Pedagogical Notes: Workload



- Yes, we know workload for this class is STILL a problem. 😞
  - And COVID is making it worse
- Fortunately this is near the end of the semester
  - And we deliberately wind down the work
- And sorry for the too-hard exam:
  - I know it is demoralizing, which is why we are going to allow clobbering... And remember, curve to 3.4-3.5 ***including*** P/NP students in calculating the curve

# Pedagogical Notes: COVID Support



- Remote instruction is ***really*** kicking our ass in 61C
  - We rely so much on in-person office hours/labs/project parties for a good 1/4 to 1/3rd of our students
- But next semester we won't have that problem...
  - So for those in that 1/4-1/3rd the help will (finally) be available
- Simplified COVID incomplete policy
  - Designed to help those students who really benefit from the in-person side of the class
  - Will up my workload in the fall, but...

# If you want/need an Incomplete



- We will provide a standard Google form
- Project 1, 2, Midterm, labs and homework from before spring break are carried forward
- Grades for Project 3, 4, Homeworks after Spring Break:
  - We will maintain an open autograder until the versions are available for Fall
    - Until then, you would submit the current one
    - After then, you have to do the new one
- Grades for Lab > spring break:
  - We will maintain an open autograder you submit to
- Grade for Final is summer or fall final:
  - Z-score scaling for grade calculation
    Z-score scaling for clobbering the Midterm
- Nick will have ***dedicated*** 1 hr/week office hours for this in the fall so you can do all but the final at the ***start*** of the semester

# And "Me Bad" on Low Rate I/O...



- I said for low rate (mice, etc) interrupts make the most sense in the slides last spring
  - Which just got copied
  - Polling overhead is low, but so is interrupt overhead...
- But that makes sense only if the devices support interrupts
  - But our keyboards and mice don't actually ***support interrupts!***
- Old days: PS2 interface directly plugged in a keyboard or mouse
  - And the keyboard/mouse generated an interrupt on keypress or movement
  - No overhead if nothing going on...

# But recently I learned that USB doesn't actually support interrupts!!!



- USB is a tree
  - With the controller at the root...
  - Devices are addressed by the path through the tree
- For simplicity, the controller controls ***everything***
  - It sends out requests to the devices to read/write data and the device responds
  - An "interrupt device" (low rate, high priority device like a keyboard or mouse) ***doesn't actually send interrupts***!
- Instead, every request starts with the controller
  - And it just guarantees that it will be able to poll the devices at a given rate...
    It schedules any bulk transfers so that it still has time to ask for data
- Thus USB uses polling, ***because it doesn't have the ability to generate interrupts!***

# And Further:
# Where IS the overhead...



- An interrupt is a huge amount of overhead
  - Flush the pipeline: Which on a modern processor might be 40+ instructions in process
  - Switching to the OS: OS now has an empty cache, so enjoy the compulsory misses
    - And not just data cache, but branch-prediction-buffer, TLB, and a gazillion other caches
  - Swiching back to user program: All caches ***MUST be cleared***!
    - Necessary for security to prevent "side channels" from occurring
- But the OS is already taking a regular interrupt (the "timer interrupt")
  - Commonly every 1-10ms
  - Used to determine when to switch programs or other housekeeping
- So you do the polling when the timer interrupt occurs
  - So the net ***additional*** overhead for polling a low rate source is very low

# Networks: Talking to the Outside World



- Originally sharing I/O devices between computers
  - E.g., printers
- Then communicating between computers
  - E.g., file transfer protocol
- Then communicating between people
  - E.g., e-mail
- Then communicating between networks of computers
  - E.g., file sharing, www, …
- Then turning multiple cheap systems into a single computer
  - Warehouse scale computing

# The Internet (1962) www.computerhistory.org/internet_history



- History
  - 1963: JCR Licklider, while at DoD's ARPA, writes a memo describing desire to connect the computers at various research universities: Stanford, Berkeley, UCLA, ...
  - 1969 : ARPA deploys 4 "nodes" @ UCLA, SRI, Utah, & UCSB
  - 1973 Robert Kahn & Vint Cerf invent TCP, now part of the Internet Protocol Suite
- Internet growth rates
  - Exponential since start
  - But finally starting to hit human scale limits although lots of silicon cockroaches...


www.greatachievements.org/?id=3736
en.wikipedia.org/wiki/Internet_Protocol_Suite

# The World Wide Web (1989)

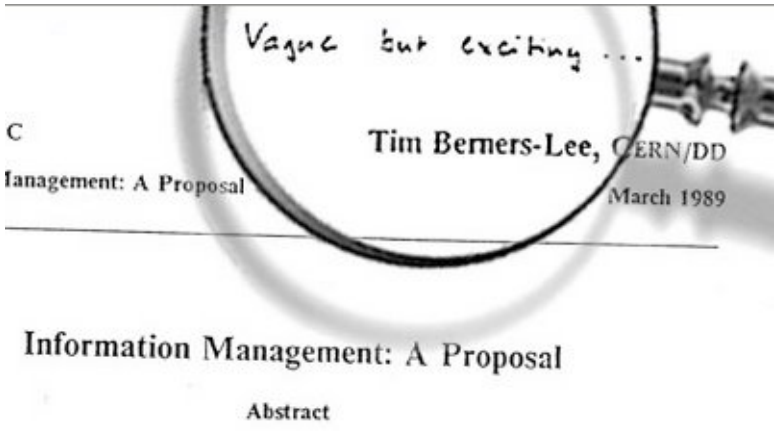
**en.wikipedia.org/wiki/History_of_the_World_Wide_Web**



- "System of interlinked hypertext documents on the Internet"
- History
  - 1945: Vannevar Bush describes hypertext system called "memex" in article
  - 1989: Sir Tim Berners-Lee proposed and implemented the first successful communication between a Hypertext Transfer Protocol (HTTP) client and server using the internet.
  - 1993: NCSA Mosaic: A graphical HTTP client
  - ~2000 Dot-com entrepreneurs rushed in, 2001 bubble burst
- Today : Access anywhere!

Tim Berners-Lee

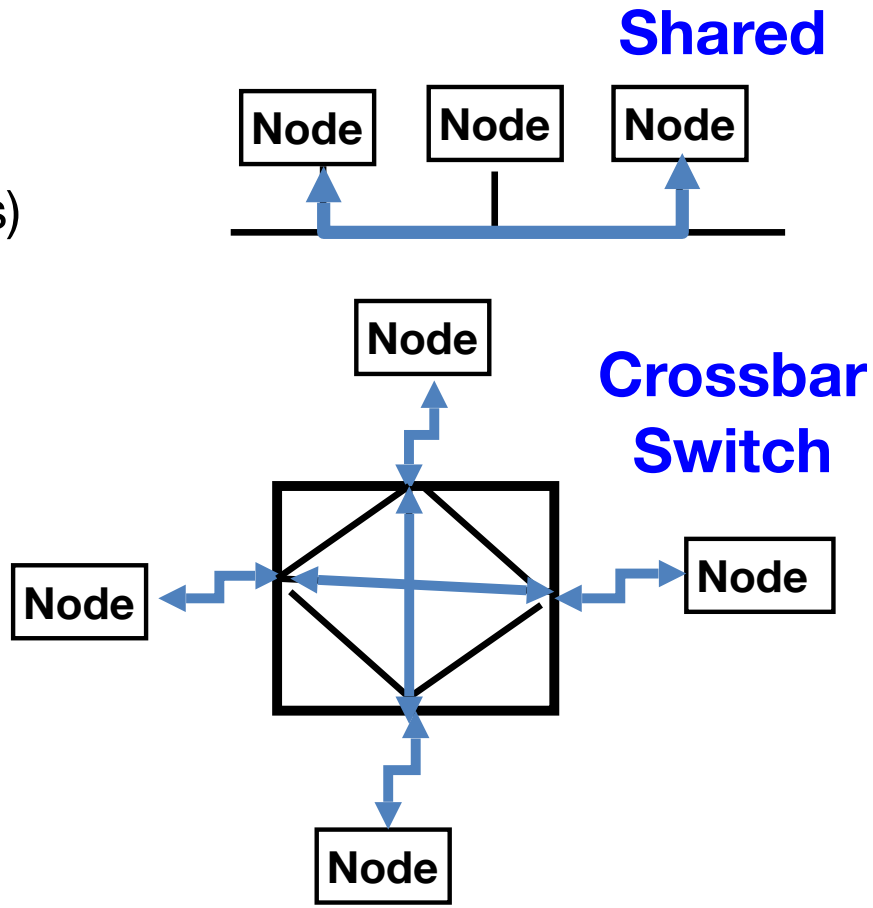
World's First web server in 1990



# Shared vs. Switch-Based Networks



- Shared vs. Switched:
  - Shared: 1 at a time (CSMA/CD)
  - Switched: pairs ("point-to-point" connections) communicate at same time
- Aggregate bandwidth (BW) in switched network is many times that of shared:
  - point-to-point faster since no arbitration, simpler interface
- Wired is almost always switched
- Wireless is by definition shared



# Shared Broadcast

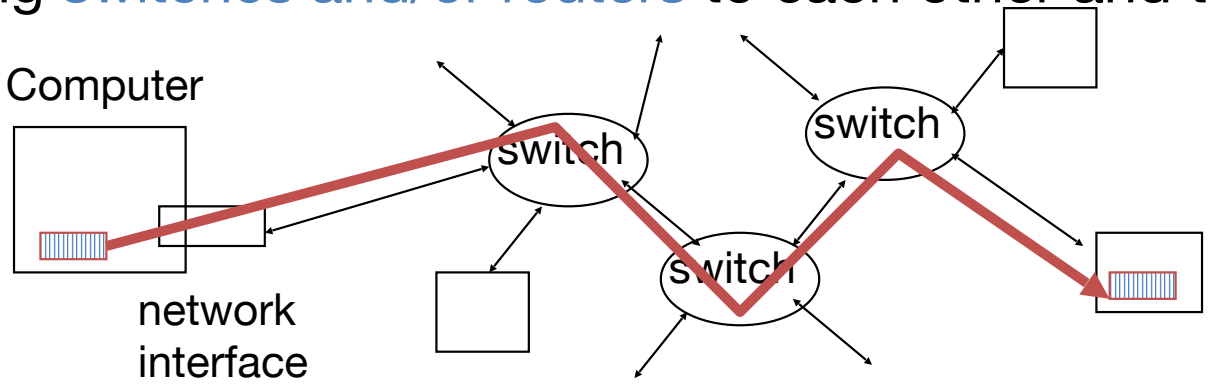


- Old-school Ethernet and Wireless
  - It doesn't just share but all others can see the request?
- How to handle access:
  - Old when I was old skool: Token ring
    - A single "token" that is passed around
  - Ethernet:
    - Listen and send
    - Randomized retry when someone else interrupts you
  - Cable Modem:
    - "Request to send": small request with a listen and send model
    - Big transfers then arbitrated

# What makes networks work?



- links connecting switches and/or routers to each other and to computers or devices

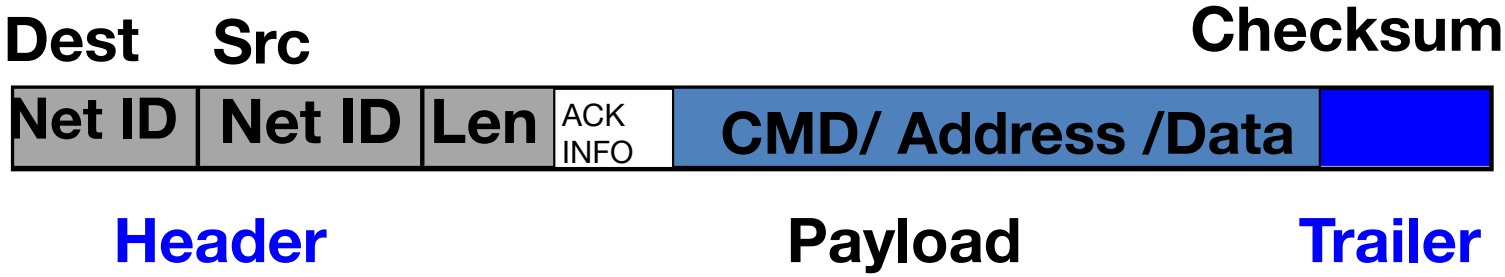



- ability to name the components and to route packets of information - messages - from a source to a destination
- Layering, redundancy, protocols, and encapsulation as means of abstraction (61C big idea)

# Software Protocol to Send and Receive



- SW Send steps
  - 1: Application copies data to OS buffer
  - 2: OS calculates checksum, starts timer
  - 3: OS sends DMA request to network interface HW and says start
- SW Receive steps
  - 3: Network interface copies data from network interface HW to OS buffer, triggers interrupt
  - 2: OS calculates checksum, if OK, send ACK; if not, delete message (sender resends when timer expires)
  - 1: If OK, OS copies data to user address space, & signals application to continue



# ***Protocols*** for Networks of Networks?



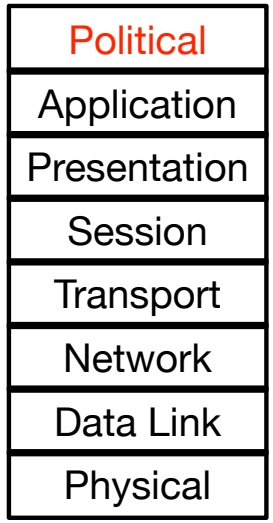
What does it take to send packets across the globe?

- Bits on wire or air
- Packets on wire or air
- Delivery packets within a single physical network
- Deliver packets across multiple networks
- Ensure the destination received the data
- Create data at the sender and make use of the data at the receiver

# The OSI 7 Layer Network Model



- A conceptual "Stack"
  - Physical Link: eg, the wires/wireless
  - Data Link: Ethernet
  - Network Layer: IP
  - Transport Layer: TCP/UDP
  - ~~Session Layer/Presentation Layer~~/Application Layer
    - Session Layer/Presentation Layer really never got used
  - Political Layer: "Feinstein/Burr 'thou shalt not encrypt'"
    - Nick is starting to spend way too much time on "layer 8" issues



# Protocol Family Concept



- Protocol: packet structure and control commands to manage communication
- Protocol families (suites): a set of cooperating protocols that implement the network stack
- Key to protocol families is that communication occurs logically at the same level of the protocol, called peer-to-peer…
  …but is implemented via services at the next lower level
- Encapsulation: carry higher level information within lower level "envelope"

# Inspiration…



- CEO A writes letter to CEO B
  - Folds letter and hands it to assistant
- Assistant:
  - Puts letter in envelope with CEO B's full name
  - Takes to FedEx
- FedEx Office
  - Puts letter in larger envelope
  - Puts name and street address on FedEx envelope
  - Puts package on FedEx delivery truck
- FedEx delivers to other company

***Dear John,***

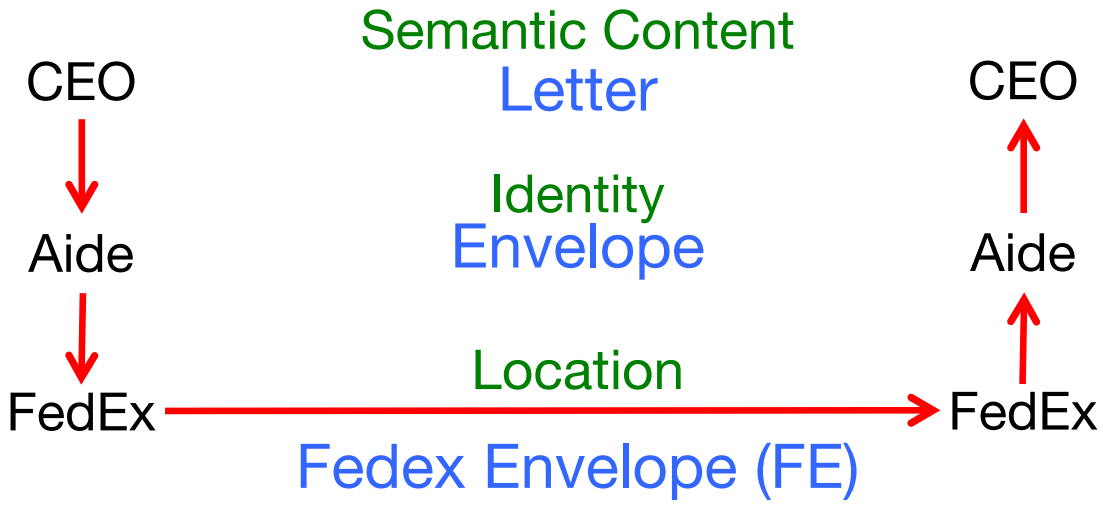
***Your days are numbered.***

***--Pat***

# The Path of the Letter

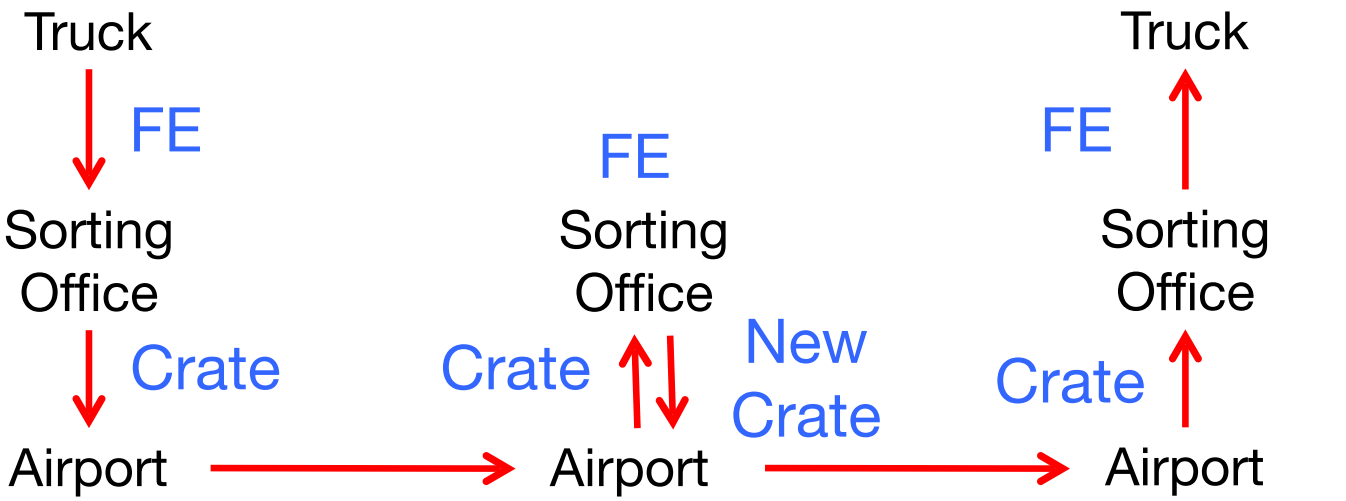


"Peers" on each side understand the same things
No one else needs to
Lowest level has most packaging



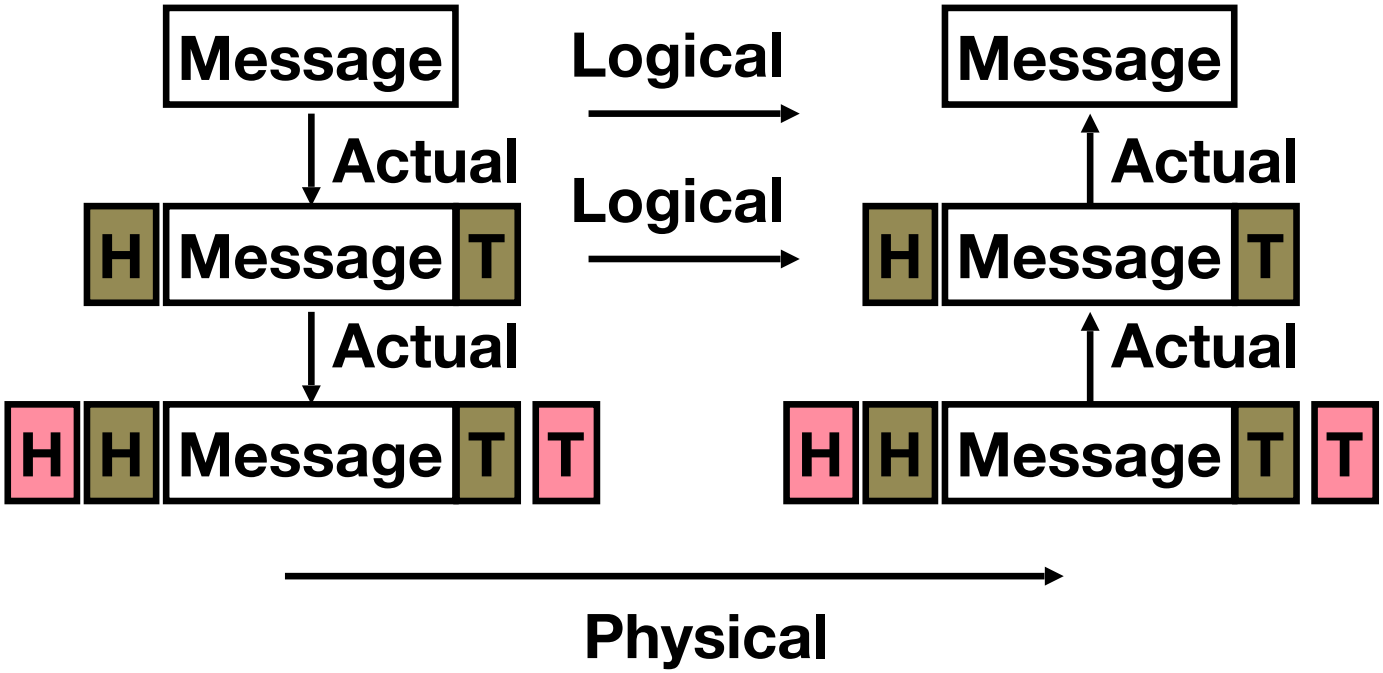
# The Path Through FedEx





# Protocol Family Concept






*Each lower level of stack "encapsulates" information from layer above by adding header and trailer.*

# Most Popular Protocol for Network of Networks

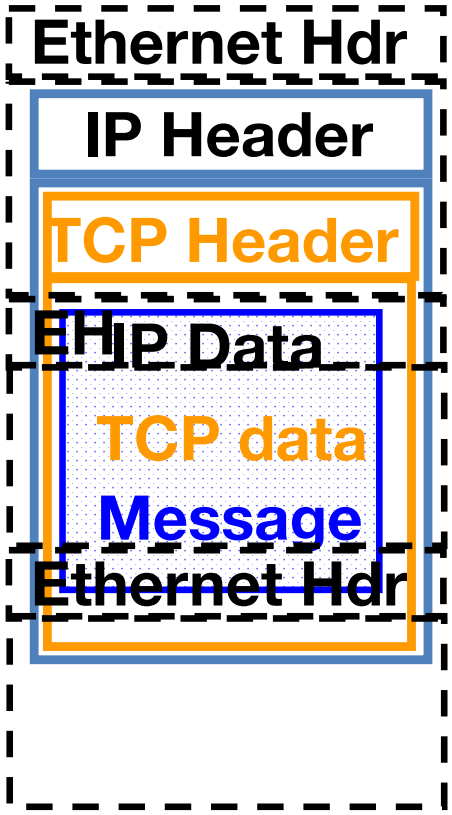


- Transmission Control Protocol/Internet Protocol (TCP/IP)
- This protocol family is the basis of the Internet, a WAN (wide area network) protocol
  - IP makes best effort to deliver
    - Packets can be lost, corrupted
      - But corrupted packets should be turned into lost packets
  - TCP guarantees ***reliable, in-order*** delivery of a ***bytestream***
    - Programs don't see packets, they just read and write strings of bytes
  - TCP/IP so popular it is used even when communicating locally: even across homogeneous LAN (local area network)

# TCP/IP packet, Ethernet packet, protocols



- Application sends message
  - **TCP breaks into 64KiB segments, adds 20B header**
  - **IP adds 20B header, sends to network**
  - **If Ethernet, broken into 1500B packets with headers, trailers**



# TCP and UDP
# The Two Internet Transfer Protocols



- TCP: Transmission Control Protocol
  - Connection based
    - SYN->SYN/ACK->ACK 3-way handshake
  - In order & reliable
    - All data is acknowledged
    - Programming interface is streams of data
- UDP: Universal Datagram Protocol
  - Datagram based
    - Just send messages
  - Out of order & unreliable
    - Datagrams can arrive in any order or be dropped (but not corrupted)
  - Needed for realtime protocols

# And Switching Gears: GPIO



- We see how to do high performance I/O
  - CPU has data it wants to send in main memory
  - Configures device & DMA controller to initiate transfer
    - Device then receives the data through DMA
- We have moderate bandwidth, flexible I/O
  - Universal Serial Bus is really a lightweight network for your slower peripheral devices
- But what about human scale?
  - With people, we only need to react in milliseconds to hours

# Reminder: Amdahl's Law and Programming Effort

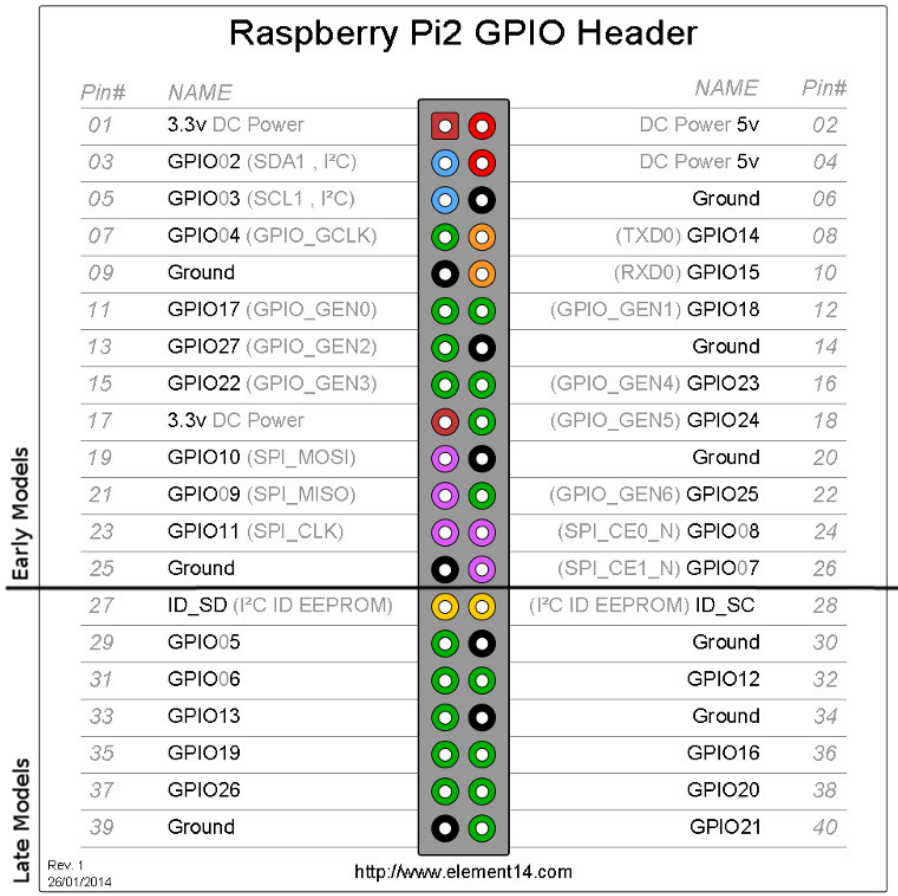


- Don't optimize where you don't need to
  - And if I only need to react at kHz granularity...
    But my processor is a GHz...
- I have 1 million clock cycles to actually decide what to do!
- So lets provide a simple interface
  - Because lets face it, ***my time*** is more valuable than the computer's time!

# Raspberry Pi GPIO



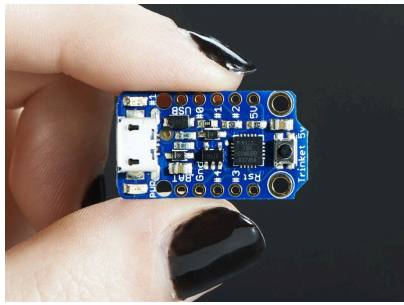
- A set of physical pins hooked up to the CPU
  - The CPU can write and read these pins as memory, like any other I/O device
- But that is a low level pain for us humans...
  - So the Linux instillation provides "files" that can access GPIO
  - You can literally write a 1 or a 0 to a pin or read the value at a pin
- Plus faster & still simple APIs



# Using GPIO



- There are a lot of add-on cards...
  - EG, ones for controlling servos
- Or you can build your own
- Combined with USB provides very powerful glue...
- Similarly some even smaller devices:
  - Adafruit "Trinket": 8 MHz 8-bit microcontroller, 5 GPIO pins
    Get it for $8 at the Jacobs Hall store...
- Big application: Serial LED strings
  - Color LEDs that have a bit-serial interface



# 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:
  - Binary numbers
  - C
  - Pointers
  - Assembly language
  - Processor micro-architecture
  - Pipelining
  - Caches
  - Floating point
- What does this buy us?
  - Promise: execution speed
  - Let's check!

# Reference Problem

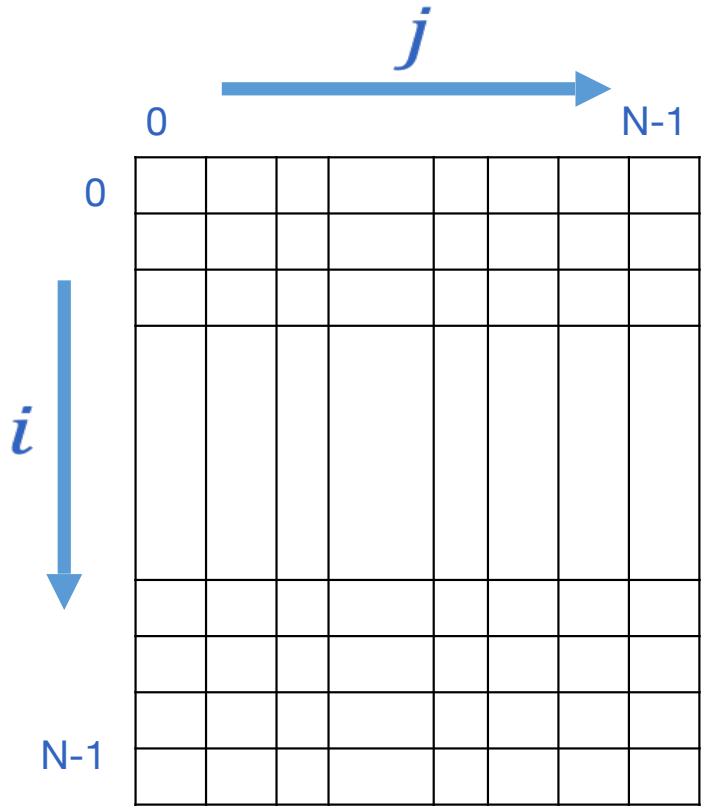


- Matrix multiplication
  - Basic operation in many engineering, data, and imaging processing tasks
    - Ex:, Image filtering, noise reduction, …
  - Core operation in Neural Nets and Deep Learning
    - Image classification (cats …)
    - Robot Cars
    - Machine translation
    - Fingerprint verification
    - Automatic game playing
- **`dgemm`**
  - double-precision floating-point general matrix-multiply
  - 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

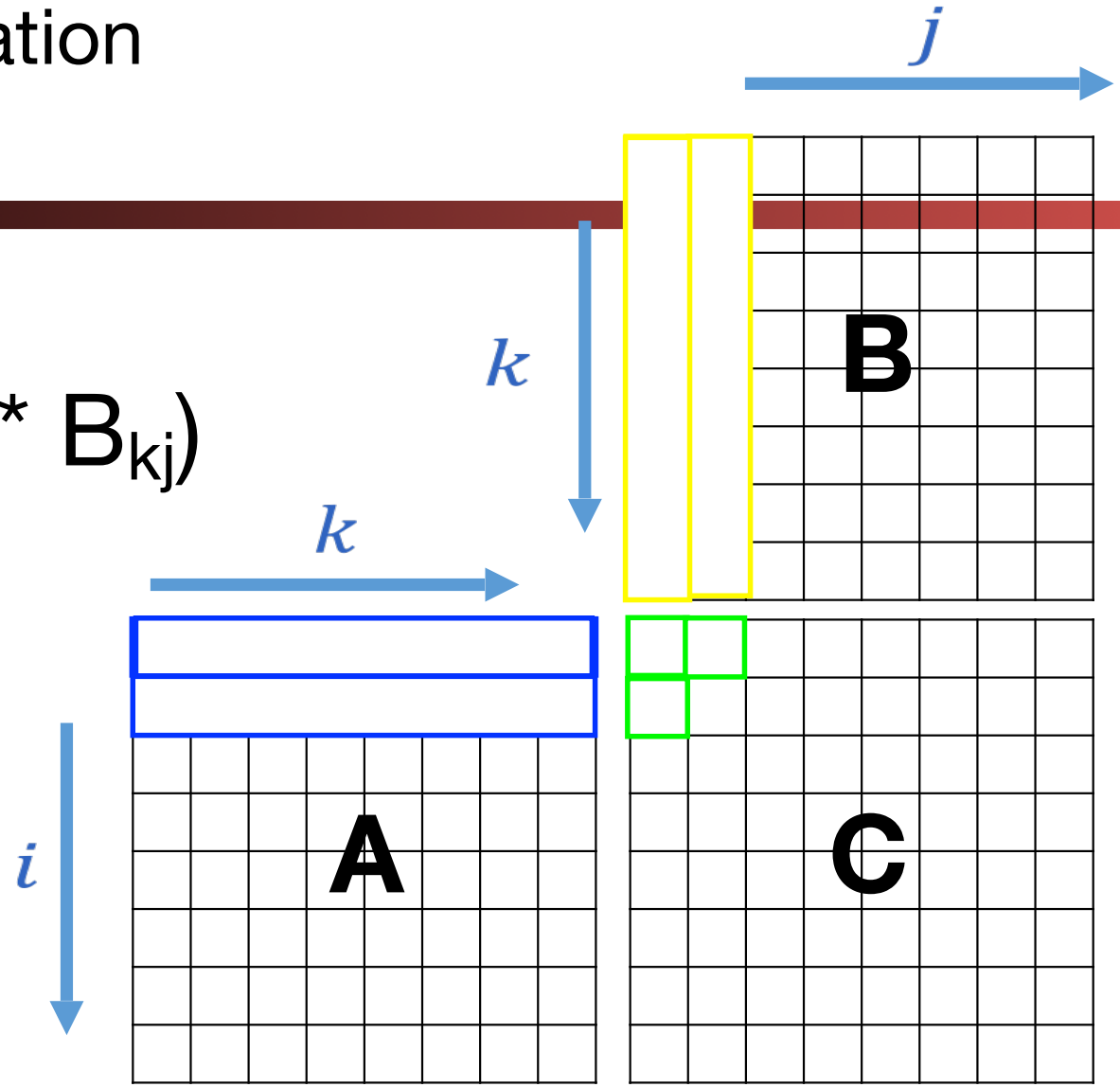


# Matrix Multiplication



**C = A*B**

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



# 2D Matrix Memory Layout

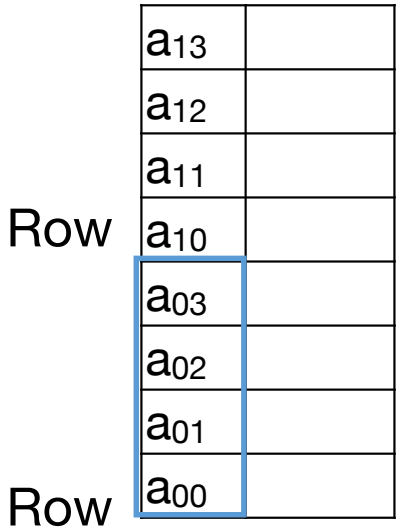


- `a[ ][ ]` in C uses row-major
- Fortran uses column-major
- Our examples use column-major

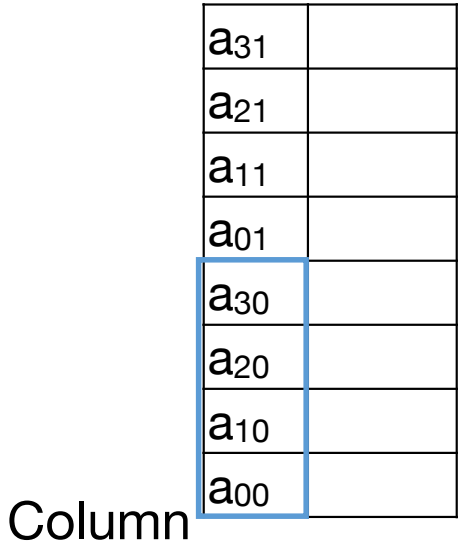
$a_{ij}$

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

Row-Major




Column-Major




$a_{ij} : a[i*N + j]$

$a_{ij} : a[i + j*N]$

# Simplifying Assumptions…



- We want to keep the examples (somewhat) manageable…
- We will keep the matrixes square
  - So both matrixes are the same size
    with the same number of rows and columns
- We will keep the matrixes reasonably aligned
  - So size % a reasonable power of 2 == 0

# `dgemm` Reference Code: 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 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 other class gives you this kind of power?**
**We could stop here … but why? Let's do better!**

# But This Is Part 1 of Project 4...



- You are developing a C-based library for python
- First is going to be implementing the C functions in sequential code
  - So right here the goal is the 200x speedup over naive python
- We've received complaints that 61C students aren't good enough C coders...
  - So we have structured project 4 to require a fair amount of C writing before you get to the parallel coding...

# 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

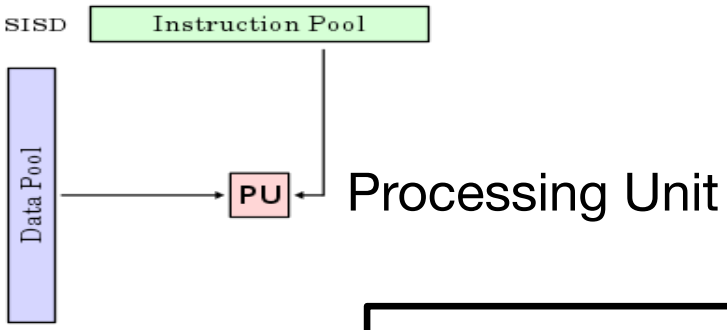


- Two basic approaches to parallelism:
  - Multiprogramming
    - run multiple independent programs in parallel
    - “Easy”
  - Parallel computing
    - run one program faster
    - “Hard”
- We’ll focus on parallel computing in the next few lectures

# 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 RISC-V processor
  - We consider superscalar as SISD because the ***programming model*** is sequential

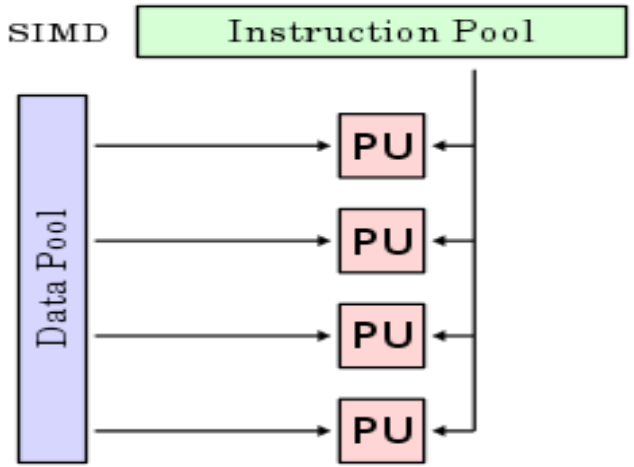



This is what we did up to now in 61C

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



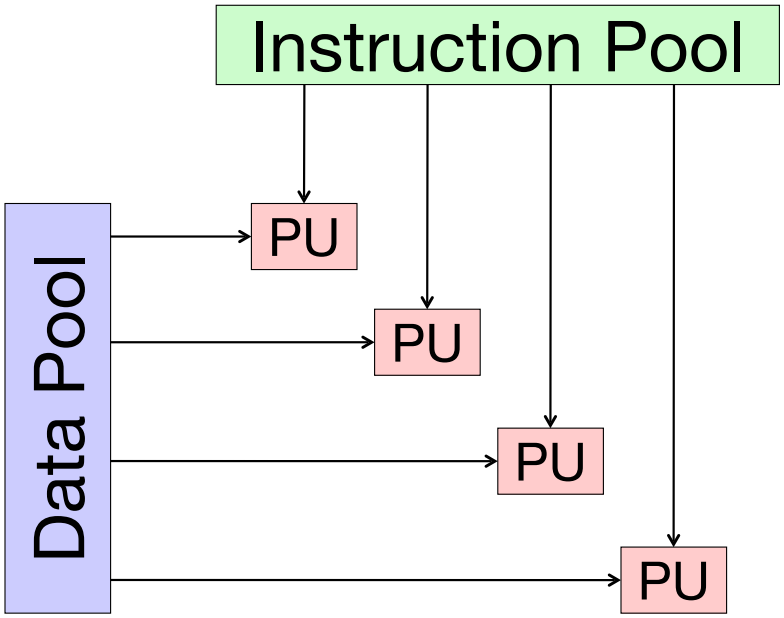
- SIMD computer processes multiple data streams using a single instruction stream, e.g., Intel SIMD instruction extensions or NVIDIA Graphics Processing Unit (GPU)




Today’s topic.

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



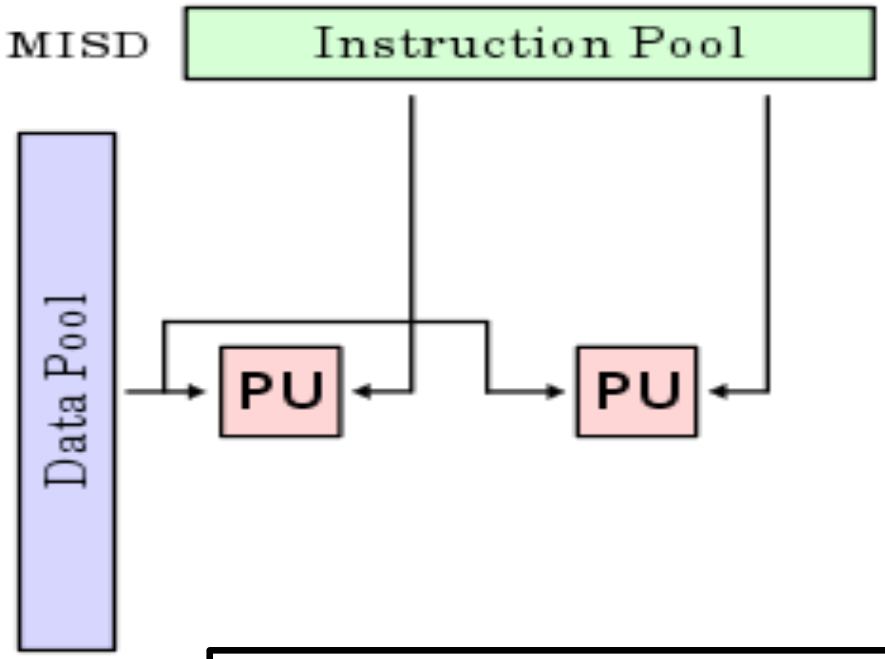



- Multiple autonomous processors simultaneously executing different instructions on different data.
  - MIMD architectures include multicore and Warehouse-Scale Computers

Topic of Lecture 22 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

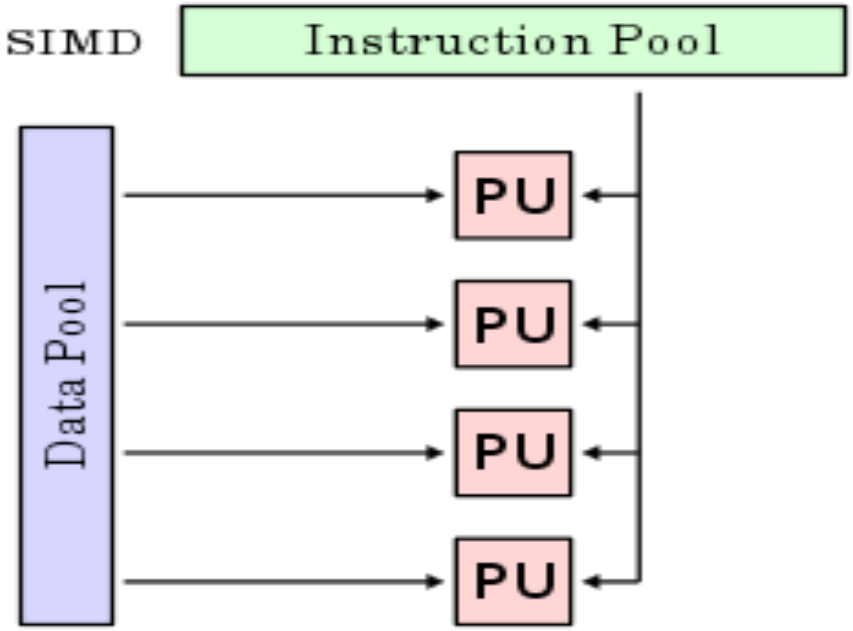
# Agenda



- 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, …

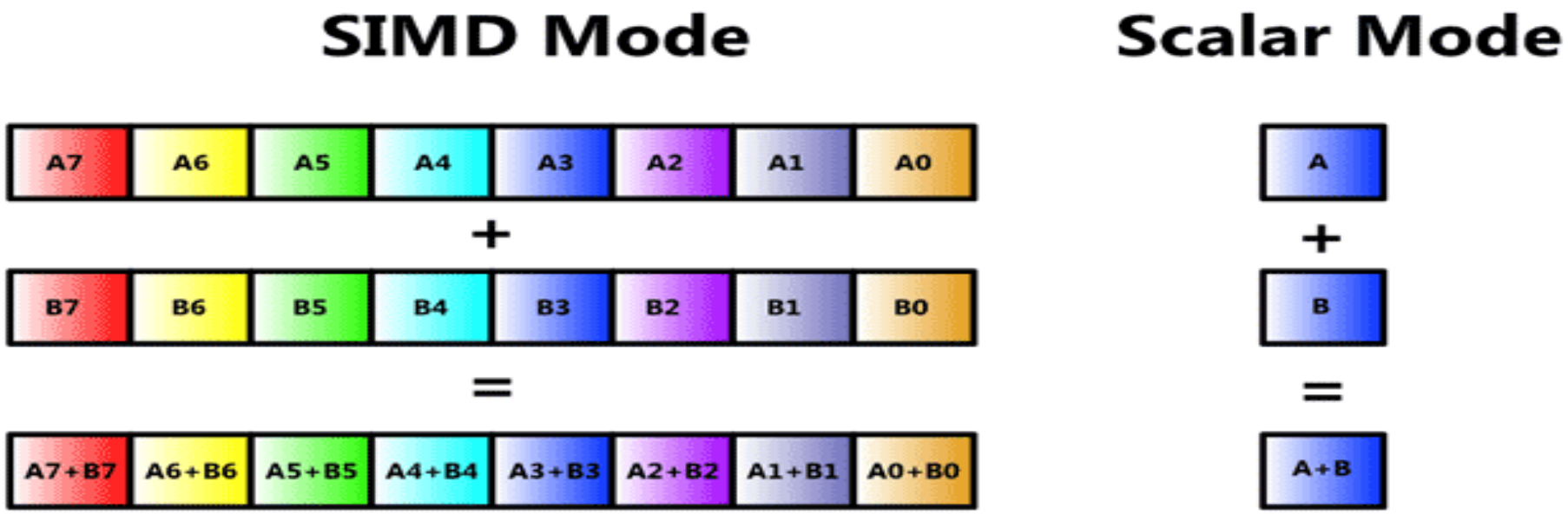
# SIMD – “Single Instruction Multiple Data”





# SIMD (Vector) Mode





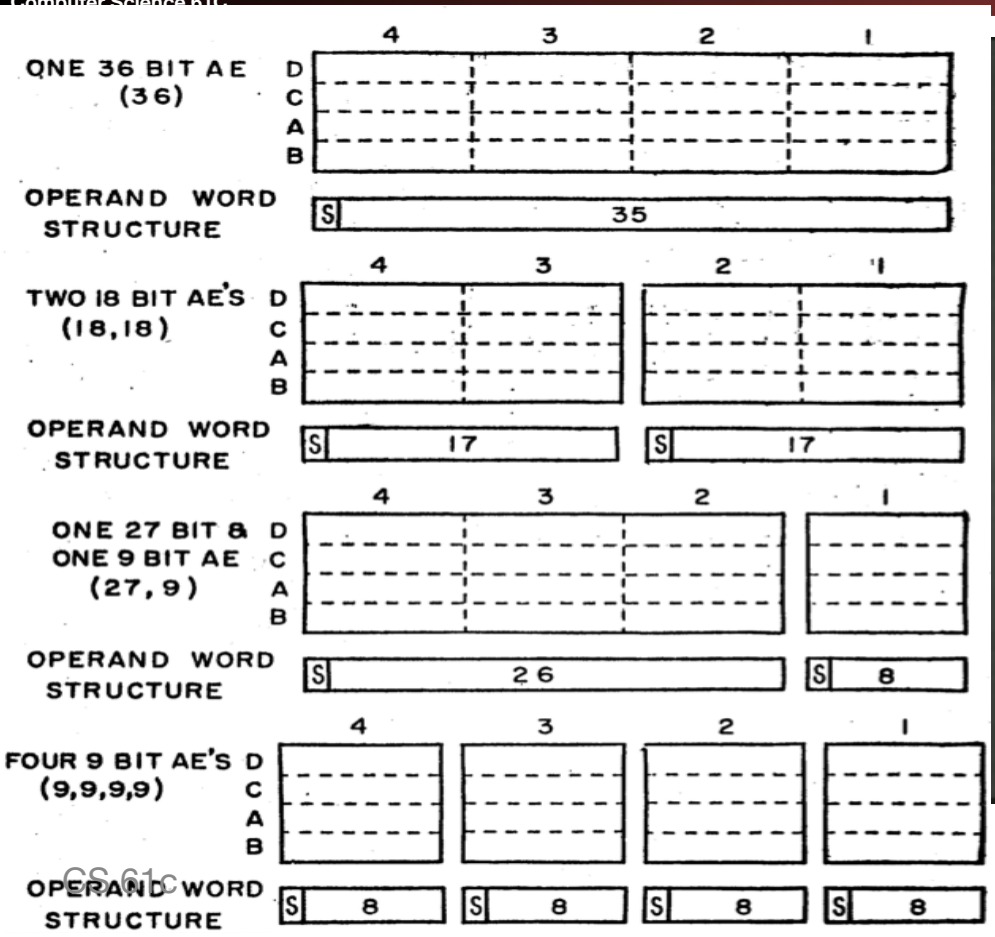
# SIMD Applications & Implementations



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

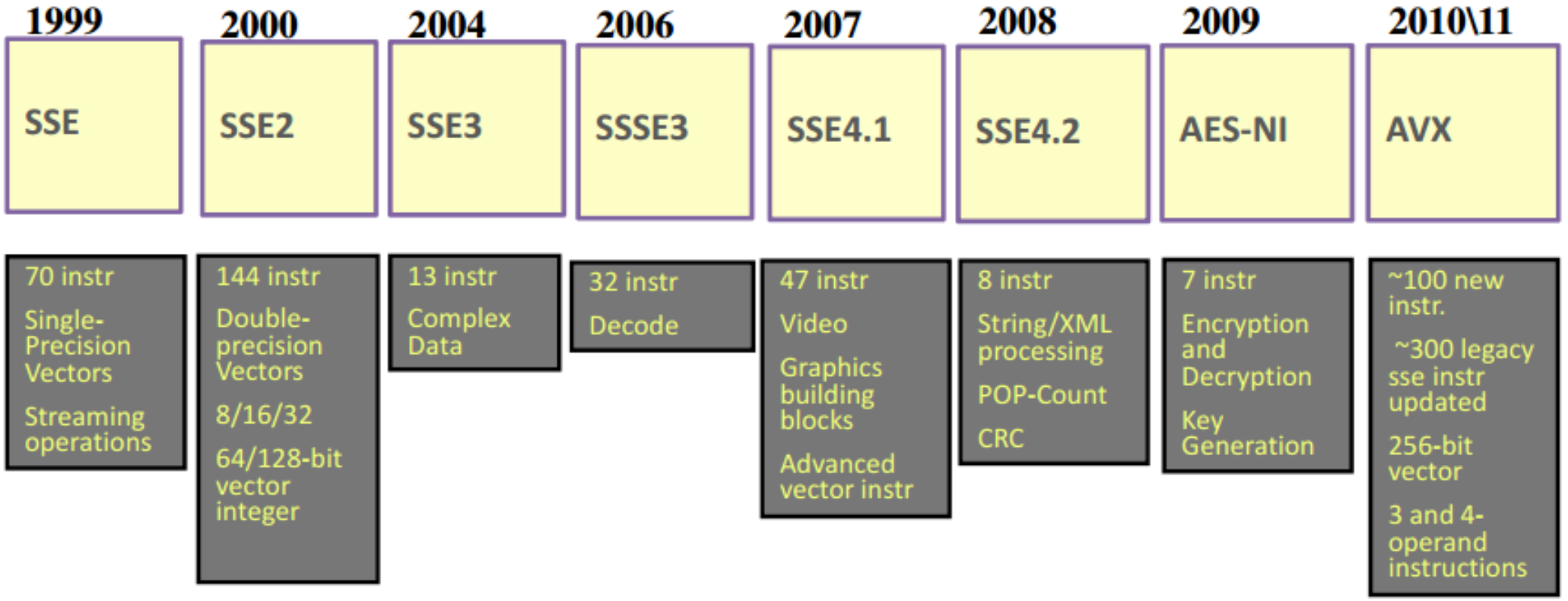
# First SIMD Extensions: MIT Lincoln Labs TX-2, 1957

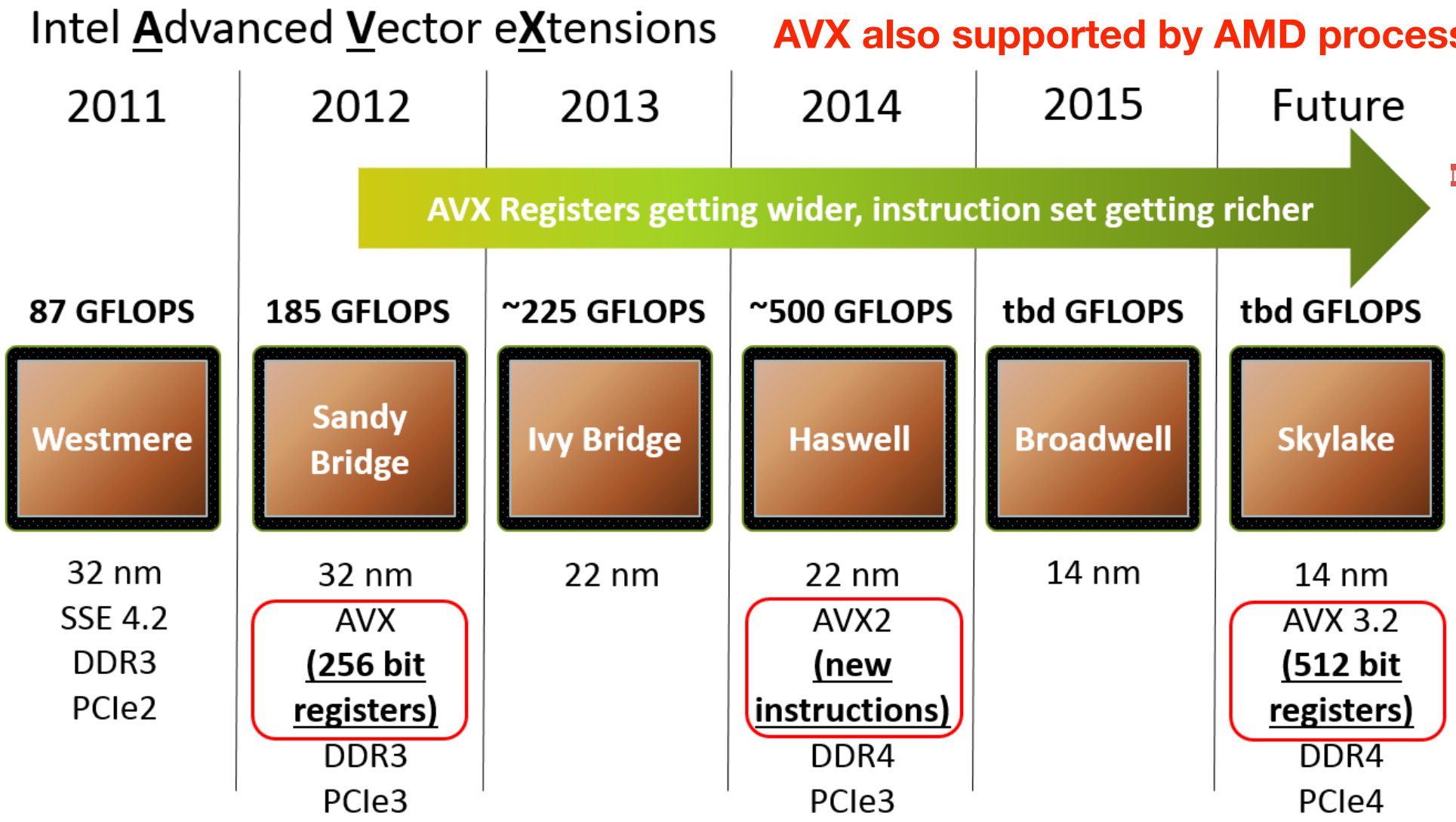





# Intel x86 SIMD: Continuous Evolution

**MMX 1997**






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

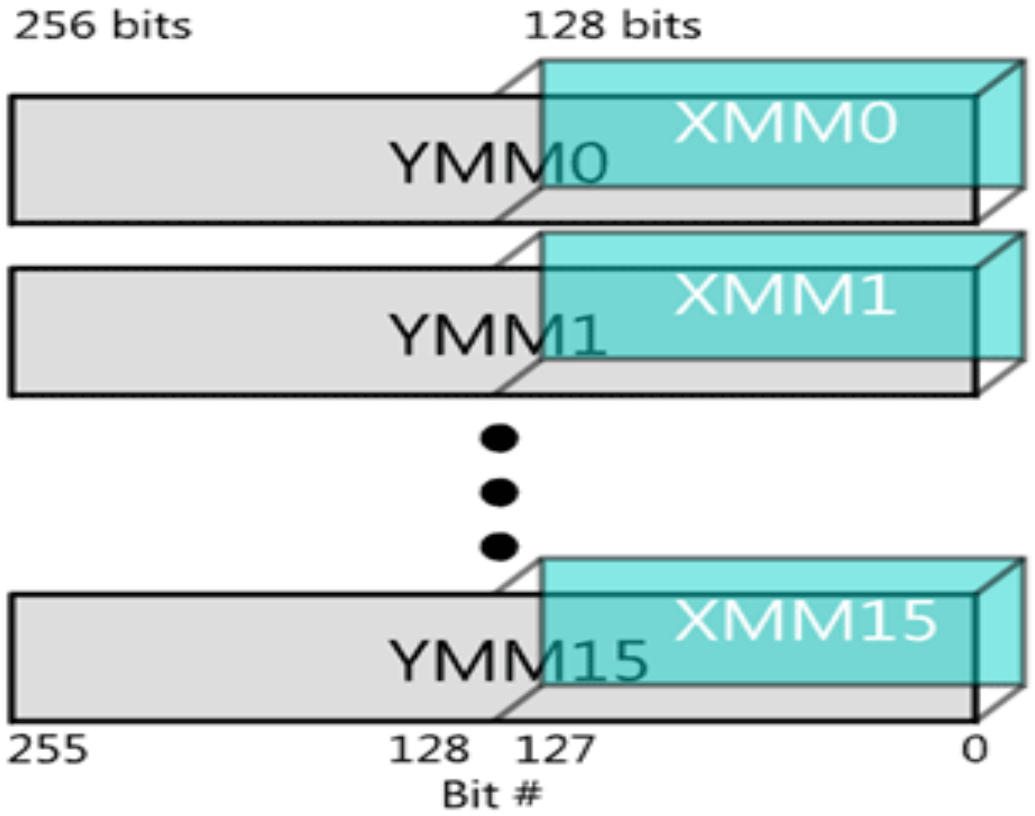
# Laptop CPU Specs



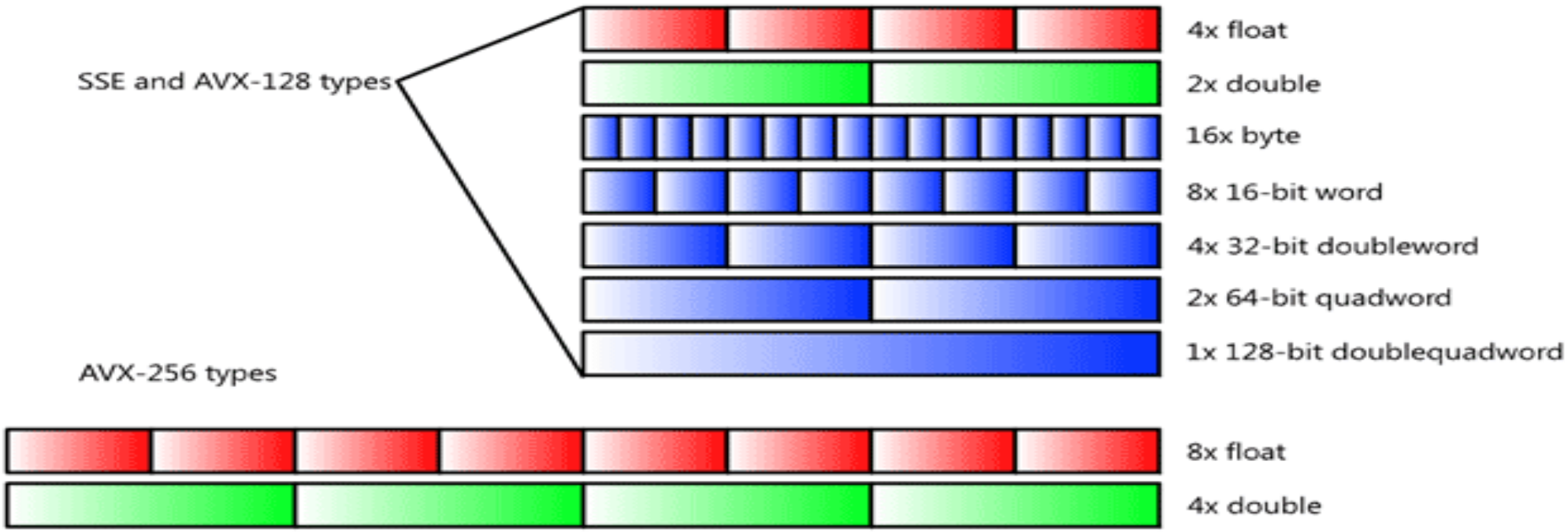
```
$ sysctl -a | grep cpu

hw.physicalcpu: 4
hw.logicalcpu: 8


machdep.cpu.brand_string: Intel(R) Core(TM) i5-1038NG7 CPU @ 2.00GHz


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: RDWRFSGS TSC_THREAD_OFFSET SGX BMI1 AVX2 FDPEO SMEP BMI2
ERMS INVPCID FPU_CSDS AVX512F AVX512DQ RDSEED ADX SMAP AVX512IFMA CLFSOPT IPT
AVX512CD SHA AVX512BW AVX512VL AVX512VBMI UMIP PKU GFNI VAES VPCLMULQDQ AVX512VNNI
AVX512BITALG AVX512VPOPCNTDQ RDPID SGXLC FSREPMOV MDCLEAR IBRS STIBP L1DF ACAPMSR
SSBD
machdep.cpu.extfeatures: SYSCALL XD 1GBPAGE EM64T LAHF LZCNT PREFETCHW RDTSCP TSCI
```

# AVX SIMD Registers:
# Greater Bit Extensions Overlap Smaller Versions





# Intel SIMD Data Types




(Now also AVX-512 available (***but not on Hive so you can't use on Proj 4***): 16x float and 8x double)

# Agenda



- 61C – the big picture
- Parallel processing
- Single instruction, multiple data
- **SIMD matrix multiplication**
- 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 …
  - Or to google, either one...
- Can we use the compiler to generate all non-SIMD instructions?

# x86 SIMD “Intrinsics”

intel Intrinsics Guide

The Intel Intrinsics Guide is an interactive reference tool for Intel intrinsic instructions, which are C style functions that provide access to many Intel instructions - including Intel® SSE, AVX, AVX-512, and more - without the need to write assembly code.

**Technologies**

- [ ] MMX
- [ ] SSE
- [ ] SSE2
- [ ] SSE3
- [ ] SSSE3
- [ ] SSE4.1
- [ ] SSE4.2
- [x] AVX
- [ ] AVX2
- [ ] FMA
- [ ] AVX-512
- [ ] KNC
- [ ] SVML
- [ ] Other

**Categories**

- [ ] Application-Targeted
- [ ] Arithmetic
- [ ] Bit Manipulation
- [ ] Cast
- [ ] Compare
- [ ] Convert
- [ ] Cryptography
- [ ] Elementary Math
- [ ] Functions
- [ ] General Support

mul_pd

Intrinsic

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

vmulpd

**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**

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 (CPI) |
|---|---|---|
| Icelake | 4 | 0.5 |
| Skylake | 4 | 0.5 |
| Broadwell | 3 | 0.5 |
| Haswell | 5 | 0.5 |

2 instructions per clock cycle (CPI = 0.5)

4 cycles latency (data hazard time...)

# 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

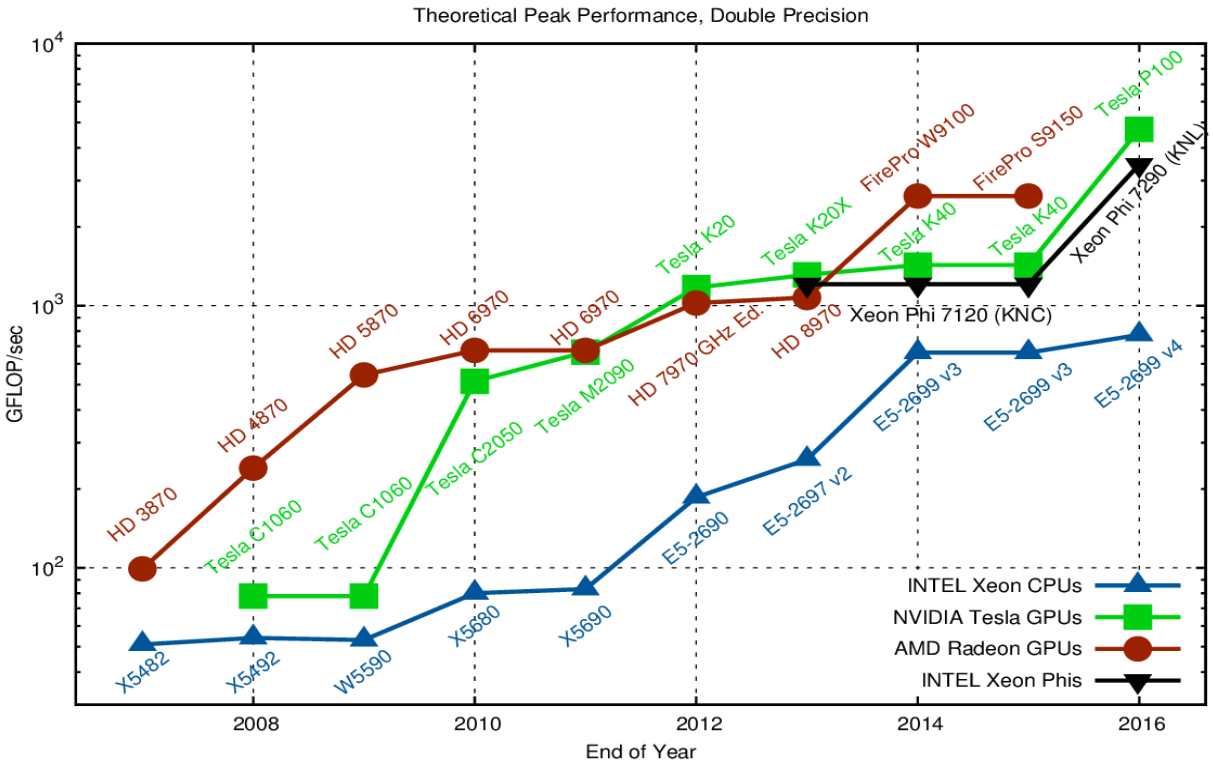


| Marking | Meaning |
|---|---|
| `[s/d]` | Single- or double-precision floating point |
| `[i/u]nnn` | Signed or unsigned integer of bit size *nnn*, where *nnn* is 128, 64, 32, 16, or 8 |
| `[ps/pd/sd]` | Packed single, packed double, or scalar double |
| `epi32` | Extended packed 32-bit signed integer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

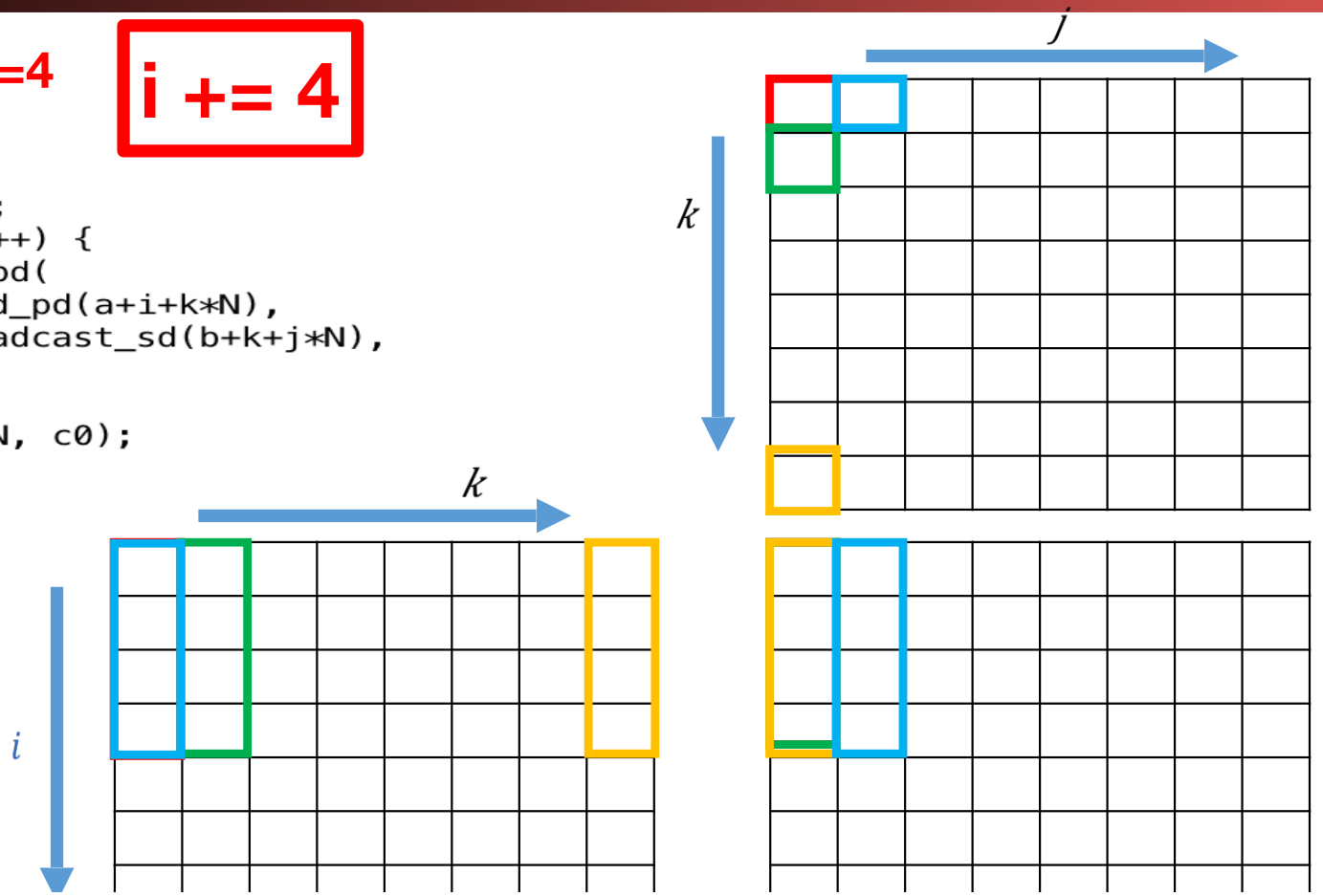


for i …; **i+=4**

***Inner Loop:*** for j …

**i += 4**

```
__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!

# 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!

# “Vectorized” dgemm: Pipeline Hazards



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

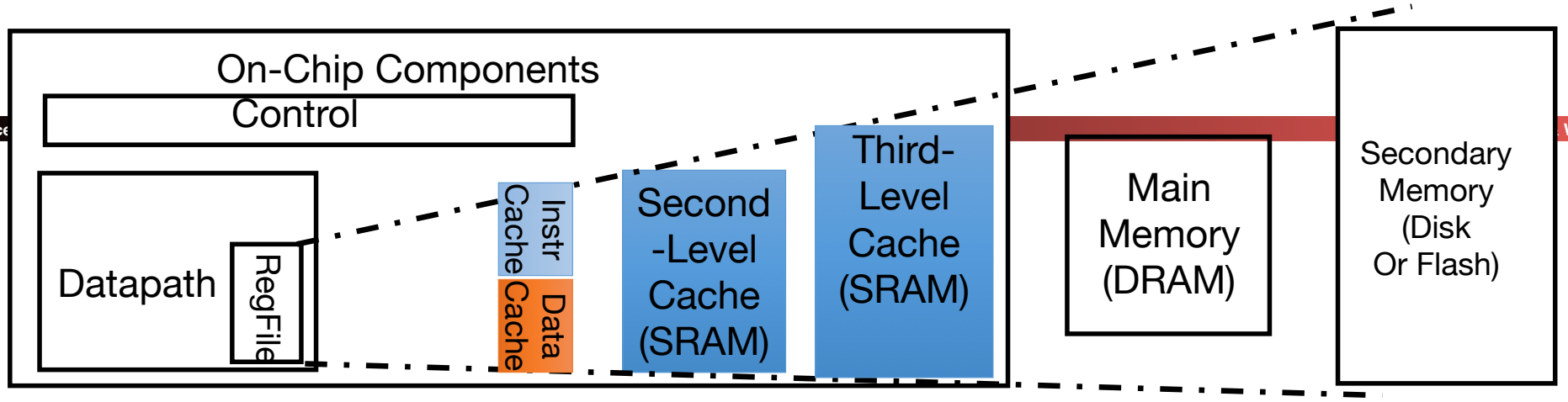
# But memory is accessed repeatedly



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

**Inner loop:**

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




| | | | | | |
|---|---|---|---|---|---|
| **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 |

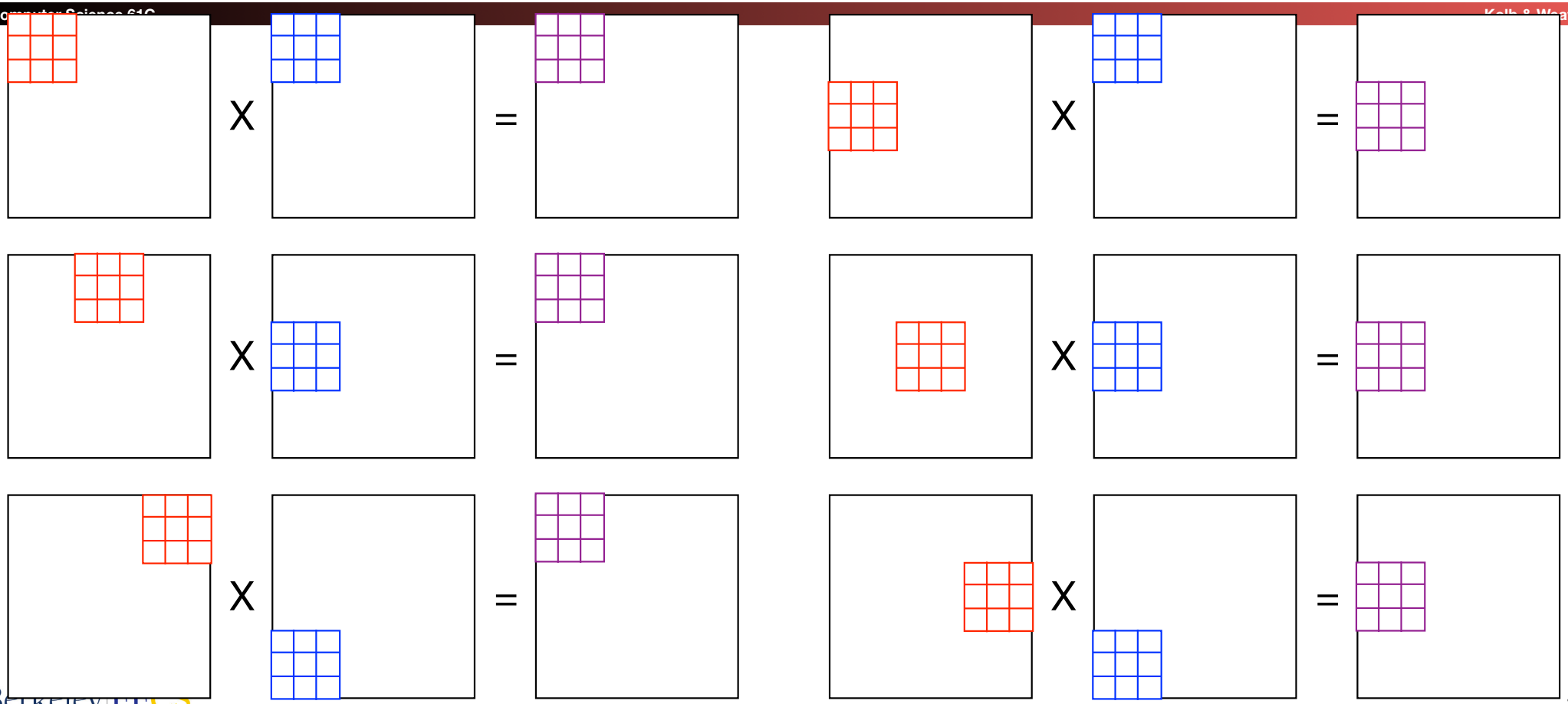
- Where are the operands (A, B, C) stored?
- What happens as N increases?
- Idea: arrange that most accesses are to fast cache!

# 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 |

# 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