

CMSC216: Memory Systems

Chris Kauffman

Last Updated:
Tue Apr 22 09:16:22 AM EDT 2025

Logistics

Announcements

Measuring Time in Code

- ▶ Measure CPU time with the standard `clock()` function; measure time difference and convert to seconds
- ▶ Measure Wall (real) time with `gettimeofday()` or related functions; fills struct with info on time of day (duh)

CPU Time

```
#include <time.h>

clock_t begin, end;
begin = clock(); // current cpu moment

do_something();

end = clock(); // later moment

double cpu_time =
((double) (end-begin)) / CLOCKS_PER_SEC;
```

Real (Wall) Time

```
#include <sys/time.h>

struct timeval tv1, tv2;
gettimeofday(&tv1, NULL); // early time

do_something();

gettimeofday(&tv2, NULL); // later time

double wall_time =
((tv2.tv_sec-tv1.tv_sec)) +
((tv2.tv_usec-tv1.tv_usec) / 1000000.0);
```

Exercise: Time and Throughput

Consider the following simple loop to sum elements of an array from `stride_throughput.c`

```
int *data = ...; // global array

int sum_simple(int len, int stride){
    int sum = 0;
    for(int i=0; i<len; i+=stride){
        sum += data[i];
    }
    return sum;
}

int main(){
    ...
    int x1 = sum_simple(n,1);
    int x2 = sum_simple(n,2);
    int x3 = sum_simple(n,3);
    // total time for each stride?
    // throughput for each stride?
}
```

- ▶ Param `stride` controls step size through loop
- ▶ Interested in two features of the `sum_simple()` function:

1. Total Time to complete
2. **Throughput:**

$$\text{Throughput} = \frac{\#Additions}{\text{Second}}$$

- ▶ How would one **measure and calculate** these two in a program?
- ▶ As stride increases, **predict** how **Total Time** and **Throughput** change

Answers: Time and Throughput

Measuring Time/Throughput

Most interested in CPU time so

```
begin = clock();
sum_simple(length,stride);
end = clock();
cpu_time = ((double) (end-begin))
           / CLOCKS_PER_SEC;

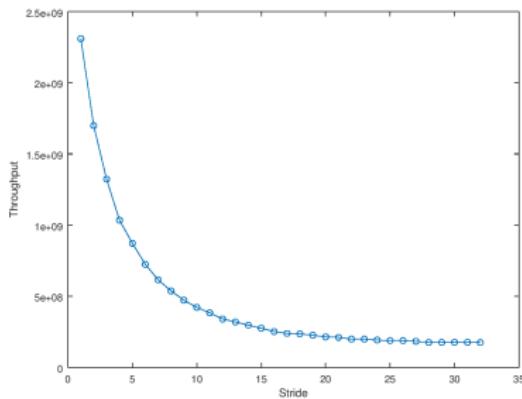
throughput = ((double) length) /
             stride /
             cpu_time;
```

Time vs Throughput

As stride increases . . .

- ▶ Time decreases: doing fewer additions (duh)
- ▶ Throughput **decreases**

Plot of Stride vs Throughput

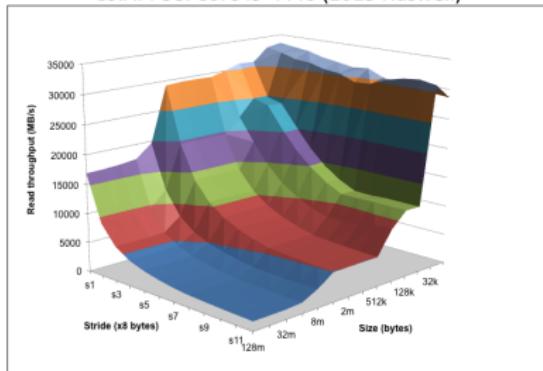


- ▶ Stride = 1: consecutive memory accesses
- ▶ Stride = 16: jumps through memory, more time

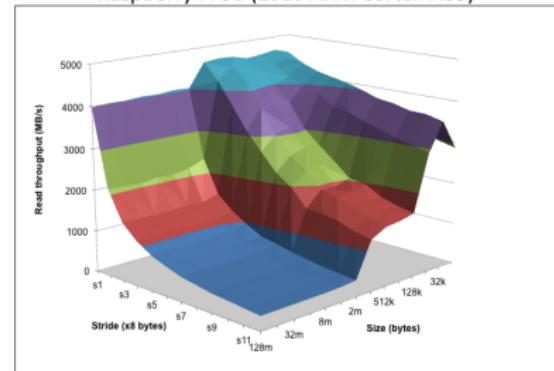
Memory Mountains from Bryant/O'Hallaron

- ▶ Varying stride for a fixed length leads to decreasing performance, 2D plot
- ▶ Can also vary length for size of array to get a 3D plot
- ▶ Illustrates features of CPU/memory on a system
- ▶ The “Memory Mountain” on the cover of our textbook
- ▶ What **interesting structure** do you see?

CS:APP3e: Core i5-4440 (2013 Haswell)



Raspberry Pi 3B (2016 ARM Cortex-A53)



CPU vs Memory Speed

- ▶ Early Computing Systems had a CPU Chips and Memory Chips, little if any data storage in the CPU (e.g. no registers)
- ▶ CPU and Memory Chips ran at similar speeds / clock frequencies: CPU would fetch data from Memory, perform arithmetic, store answers back to Memory
- ▶ Engineers found it easier to increase CPU Chip speed than Memory Chip speed: could now perform 100s of arithmetic operations in the time that a single Memory Fetch / Store could take place
- ▶ **Registers and Cache** were developed in response to the growing speed difference between CPU and Memory Chips
- ▶ Registers can be directly controlled by programmers (if the code in Assembly)
- ▶ Cache memory is (mostly) managed by the hardware itself, the **Main Memory System**

Cache Favors Temporal and Spatial Locality

Hardware folks noticed programmers often write loops like

```
for(int i=0; i<len; i++){  
    sum += array[i];  
}
```

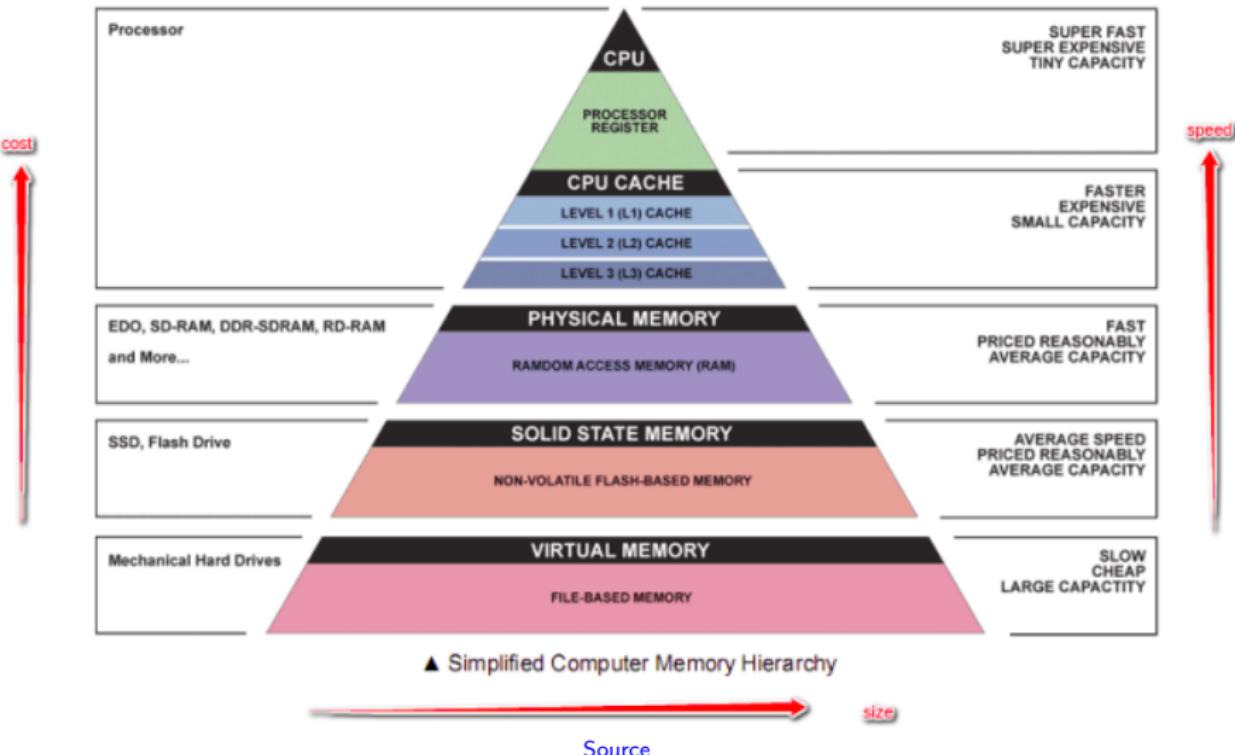
which exhibits two Memory Locality features

1. **Temporal Locality**: memory recently used likely to be used again soon (like `sum` and `i` used in every loop iteration)
2. **Spatial Locality**: nearby addresses to recently used memory likely to be used (like `arr[0]` first then `arr[1],arr[2]`)

Hardware engineers began adding chunks of Memory to CPUs to exploit these code tendencies giving rise to Cache Memory

- ▶ Code that utilizes Cache well will run faster

The Memory Pyramid



Numbers Everyone Should Know

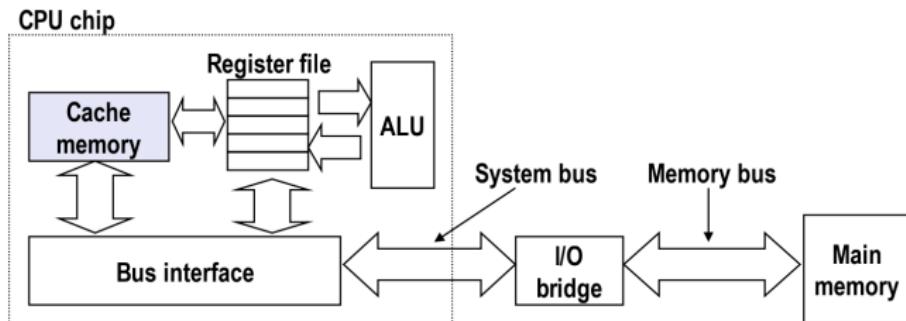
- ▶ “Main Memory” is comprised of many different physical devices that work together and have differing sizes/speeds
- ▶ Accessing memory at #4096 may involve some or all of...
 - ▶ Several Levels of Cache Memory on CPU (SRAM)
 - ▶ DRAM memory on separate chips
 - ▶ Permanent storage (SSDs and HDDs)

Edited Excerpt of [Jeff Dean's](#) talk on data centers.

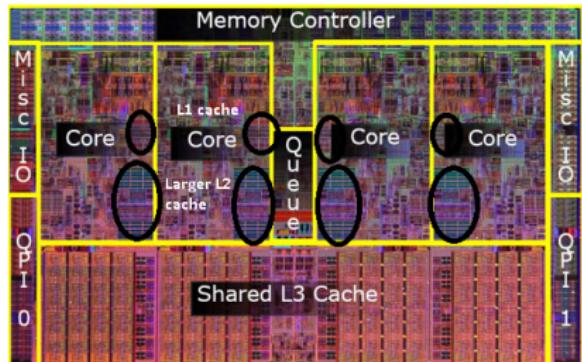
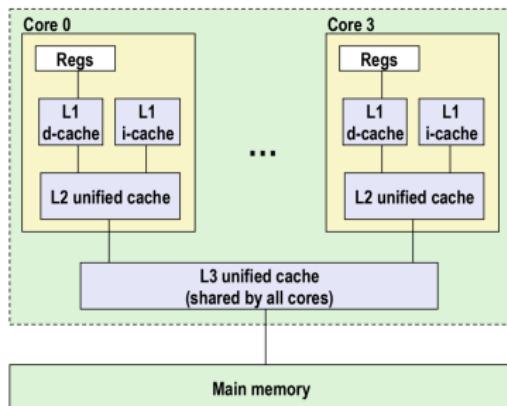
Reference	Time	Analogy
Register	-	Your brain
L1 cache reference	0.5 ns	Your desk
L2 cache reference	7 ns	Neighbor's Desk
DRAM memory reference	100 ns	This Room
Disk seek	10,000,000 ns	Salt Lake City

Big-O Analysis does NOT capture these; proficient programmers do

Diagrams of Memory Interface and Cache Levels



Source: Bryant/O'Hallaron CS:APP 3rd Ed.



Source: SO "Where exactly L1, L2 and L3 Caches located in computer?"

Why isn't Everything Cache?

Metric	1985	1990	1995	2000	2005	2010	2015	2015/1985
SRAM \$/MB	2,900	320	256	100	75	60	25	116
SRAM access (ns)	150	35	15	3	2	1.5	1.3	115
DRAM \$/MB	880	100	30	1	0.1	0.06	0.02	44,000
DRAM access (ns)	200	100	70	60	50	40	20	10

Source: Bryant/O'Hallaron CS:APP 3rd Ed., Fig 6.15, pg 603

1 bit SRAM = 6 transistors

1 bit DRAM = 1 transistor + 1 capacitor

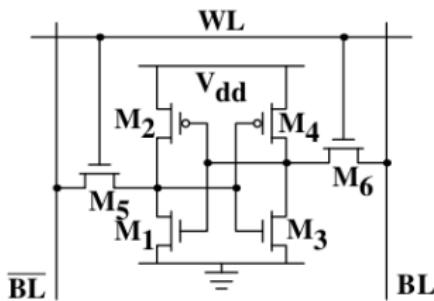


Figure 2.4: 6-T Static RAM

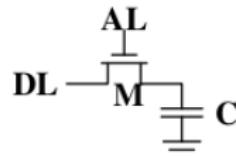
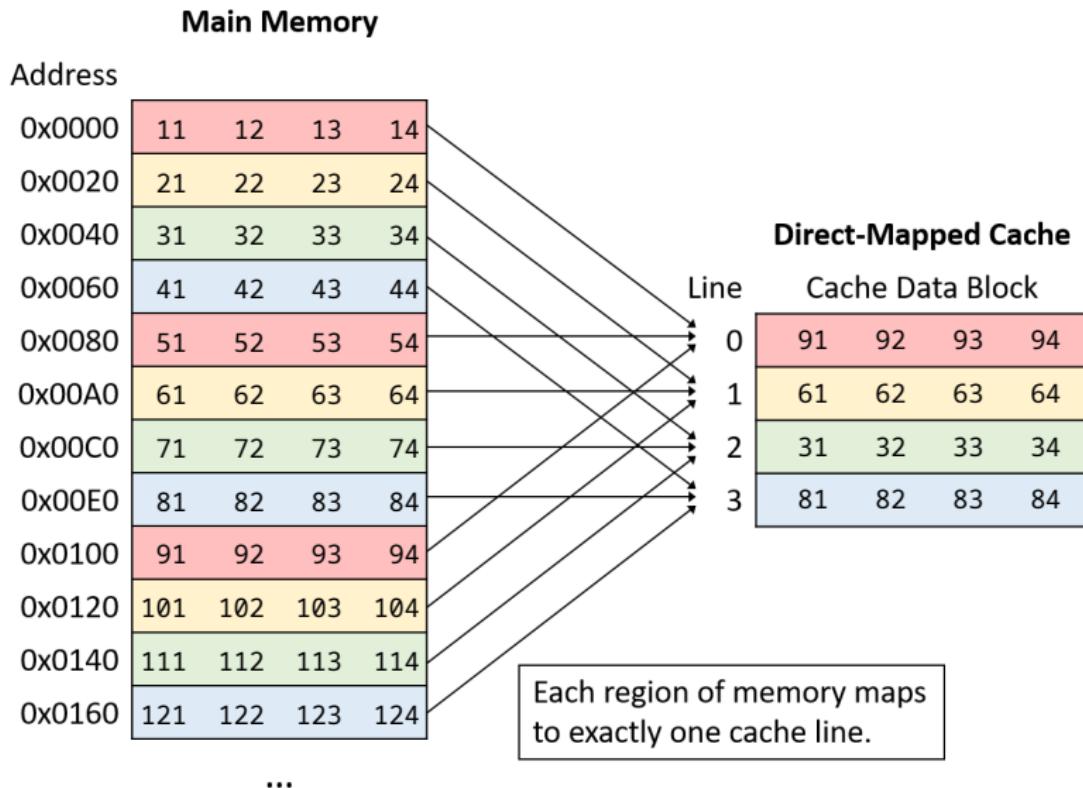


Figure 2.5: 1-T Dynamic RAM

"What Every Programmer Should Know About Memory" by Ulrich Drepper, Red Hat, Inc.

Diagram of Direct Mapped Cache



Source: Dive into Systems dot org, with modifications

How big is your cache? Check Linux System special Files

lscpu Utility

Handy Linux program that summarizes info on CPU(s)

```
> lscpu
Architecture: x86_64
CPU op-mode(s): 32-bit, 64-bit
Byte Order: Little Endian
Address sizes: 36 bits physical,
                48 bits virtual
CPU(s):        4
Vendor ID:    GenuineIntel
CPU family:   6
Model:        58
Model name:   Intel(R) Core(TM)
              i7-3667U CPU @ 2.00GHz
...
L1d cache:    64 KiB
L1i cache:    64 KiB
L2 cache:    512 KiB
L3 cache:    4 MiB
Vulnerability Meltdown: Mitigation; ...
Vulnerability Spectre v1: Mitigation ...
...
```

Detailed Hardware Info

Files under /sys/devices/... show hardware info (caches)

```
> cd /sys/devices/system/cpu/cpu0/cache/
> ls
index0  index1  index2  index3 ...
> ls index0/
number_of_sets  type  level  size
ways_of_associativity ...
> cd index0
> cat level type number_* ways_* size
1 Data 64 8 32K
> cd ../index1
> cat level type number_* ways_* size
1 Instruction 64 8 32K
> cd ../index3
> cat level type number_* ways_* size
3 Unified 8192 20 10240K
```

Exercise: 2D Arrays

- ▶ Several ways to construct “2D” arrays in C
- ▶ All must *embed* a 2D construct into 1-dimensional memory
- ▶ Consider the 2 styles below: how will the picture of memory look different?

```
// REPEATED MALLOC
// allocate
int rows=100, cols=30;
int **mat =
    malloc(rows * sizeof(int*));

for(int i=0; i<rows; i++){
    mat[i] = malloc(cols*sizeof(int));
}

// do work
mat[i][j] = ...

// free memory
for(int i=0; i<rows; i++){
    free(mat[i]);
}
free(mat);
```

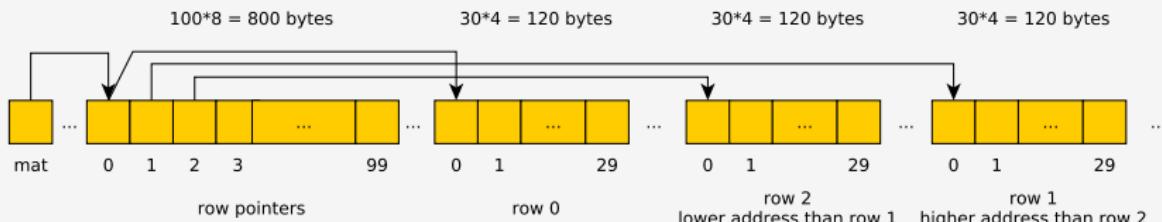
```
// TWO MALLOCs
// allocate
int rows=100, cols=30;
int **mat =
    malloc(rows * sizeof(int*));
int *data =
    malloc(rows*cols*sizeof(int));
for(int i=0; i<rows; i++){
    mat[i] = data+i*cols;
}

// do work
mat[i][j] = ...

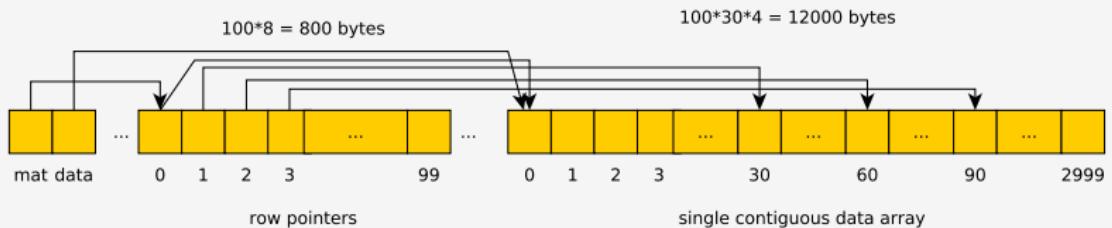
// free memory
free(data);
free(mat);
```

Answer: 2D Arrays

Repeated Mallocs



Two Mallocs



Single Malloc Matrices

Somewhat common to use a 1D array as a 2D matrix as in

```
int *matrix =
    malloc(rows*cols*sizeof(int));

int i=5, j=20;
int elem_ij = matrix[ i*cols + j ]; // retrieve element i,j
```

HWs / Labs / P4 will use this technique along with some structs and macros to make it more readable:

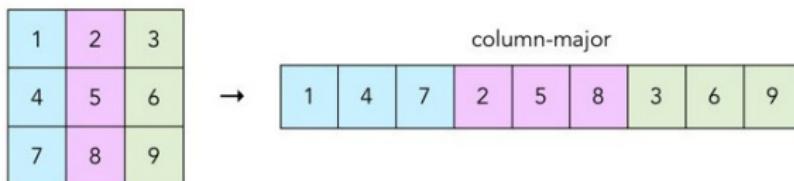
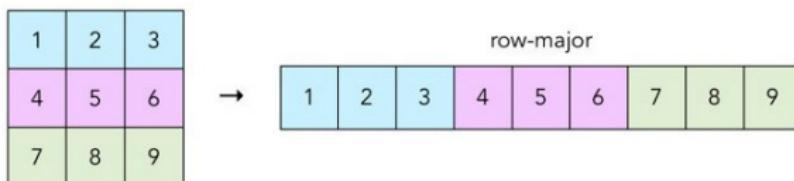
```
matrix_t mat;
matrix_init(&mat, rows, cols);

int elij = MGET(mat,i,j);
// elij = mat.data[ mat.cols*i + j ]

MSET(mat,i,j, 55);
// mat.data[ mat.cols*i + j ] = 55;
```

Aside: Row-Major vs Col-Major Layout

- ▶ Many languages use **Row-Major** order for 2D arrays/lists
 - ▶ C, Java, Python, Ocaml,...
 - ▶ $\text{mat}[i]$ is a contiguous row, $\text{mat}[i][j]$ is an element
- ▶ Numerically-oriented languages use **Column-Major** order
 - ▶ Fortran, Matlab/Octave, R, Ocaml (?)...
 - ▶ $\text{mat}[j]$ is a contiguous **column**, $\text{mat}[i][j]$ is an element
- ▶ Being aware of language convention can increase efficiency



Source: The Craft of Coding

Exercise: Matrix Summing

- ▶ How are the two codes below different?
- ▶ Are they doing the same number of operations?
- ▶ Which will run faster?

```
int sumR = 0;  
for(int i=0; i<rows; i++){  
    for(int j=0; j<cols; j++){  
        sumR += mat[i][j];  
    }  
}
```

```
int sumC = 0;  
for(int j=0; j<cols; j++){  
    for(int i=0; i<rows; i++){  
        sumC += mat[i][j];  
    }  
}
```

Answer: Matrix Summing

- ▶ Show timing in `matrix_timing.c`
- ▶ `sumR` faster than `sumC`: caching effects
- ▶ Discuss timing functions used to determine duration of runs

```
> gcc -Og matrix_timing.c
```

```
> a.out 50000 10000
```

```
sumR: 1711656320 row-wise CPU time: 0.265 sec, Wall time: 0.265  
sumC: 1711656320 col-wise CPU time: 1.307 sec, Wall time: 1.307
```

- ▶ `sumR` runs about 6 times faster than `sumC`
- ▶ Understanding why requires knowledge of the memory hierarchy and cache behavior

(Optional) Tools to Measure Performance: perf

- ▶ The Linux `perf` tool is useful to measure performance of an entire program
- ▶ Shows variety of statistics tracked by the kernel about things like memory performance
- ▶ **Examine** examples involving the `matrix_timing` program: `sumR` vs `sumC`
- ▶ **Determine** statistics that explain the performance gap between these two?

(Optional Exercise): perf on sumR vs sumC

What stats below might explain the performance difference?

```
> perf stat $perfopts ./matrix_timing 8000 4000 row ## RUN sumR ROW SUMMING
sumR: 1227611136 row-wise CPU time: 0.019 sec, Wall time: 0.019
```

Performance counter stats for './matrix_timing 8000 4000 row':		%SAMPLED
135,161,407	cycles:u	(45.27%)
417,889,646	instructions:u # 3.09 insn per cycle	(56.22%)
56,413,529	L1-dcache-loads:u	(55.96%)
3,843,602	L1-dcache-load-misses:u # 6.81% of all L1-dcache hits	(50.41%)
28,153,429	L1-dcache-stores:u	(47.42%)
125	L1-icache-load-misses:u	(44.77%)
3,473,211	cache-references:u # last level of cache	(56.22%)
1,161,006	cache-misses:u # 33.427 % of all cache refs	(56.22%)

```
> perf stat $perfopts ./matrix_timing 8000 4000 col # RUN sumC COLUMN SUMMING
sumC: 1227611136 col-wise CPU time: 0.086 sec, Wall time: 0.086
```

Performance counter stats for './matrix_timing 8000 4000 col':		%SAMPLED
372,203,024	cycles:u	(40.60%)
404,821,793	instructions:u # 1.09 insn per cycle	(57.23%)
61,990,626	L1-dcache-loads:u	(60.21%)
39,281,370	L1-dcache-load-misses:u # 63.37% of all L1-dcache hits	(45.66%)
23,886,332	L1-dcache-stores:u	(43.24%)
2,486	L1-icache-load-misses:u	(40.82%)
32,582,656	cache-references:u # last level of cache	(59.38%)
1,894,514	cache-misses:u # 5.814 % of all cache refs	(60.38%)

Answers: perf stats for sumR vs sumC, what's striking?

Observations

- ▶ Similar number of instructions between row/col versions
- ▶ #cycles lower for row version → higher insn per cycle
- ▶ **L1-dcache-misses**: marked difference between row/col version
- ▶ **Last Level Cache Refs** : many, many more in col version
- ▶ Col version: much time spent waiting for memory system to feed in data to the processor

Notes

- ▶ The right-side percentages like (50.41%) indicate how much of the time this feature is measured; some items can't be monitored all the time.
- ▶ Specific perf invocation is in
`10-memory-systems-code/measure-cache.sh`

Flavors of Permanent Storage

- ▶ Have discussed a variety of fast memories which are **small**
- ▶ At the bottom of the pyramid are **disks**: slow but **large** memories, may contain copies of what is in higher parts of memory pyramid
- ▶ These are **persistent**: when powered off, they retain information
- ▶ Permanent storage often referred to as a “drive”
- ▶ Comes in many variants but these 3 are worth knowing about in the modern era
 1. Rotating Disk Drive
 2. Solid State Drive
 3. Magnetic Tape Drive
- ▶ Surveyed in the slides that follow

Ye Olde Rotating Disk

- ▶ Store bits “permanently” as magnetized areas on special platters
- ▶ Magnetic disks: moving parts → slow
- ▶ Cheap per GB of space

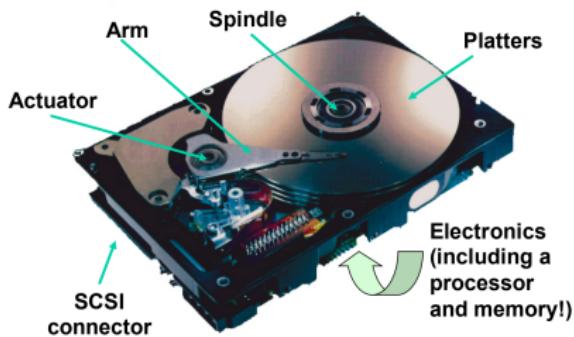
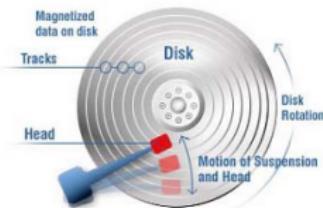


Image courtesy of Seagate Technology

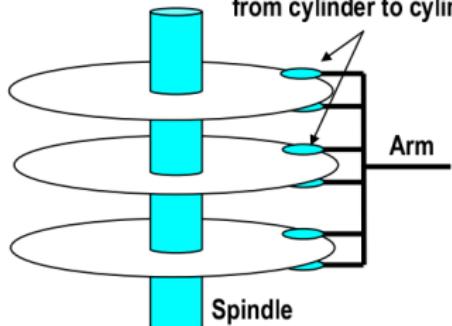
Source: CS:APP Slides

HARD DRIVE DATA READ & WRITE OPERATION MOTION DIAGRAM



Source: Realtechs.net

Read/write heads
move in unison
from cylinder to cylinder



Source: CS:APP Slides

Rotating Disk Drive Features of Interest

Measures of Quality

- ▶ Capacity: bigger is usually better
- ▶ Seek Time: delay before a head assembly reaches an arbitrary track of the disk that contains data
- ▶ Rotational Latency: time for disk to spin around to correct position; faster rotation → lower Latency
- ▶ Transfer Rate: once correct read/write position is found, how fast data moves between disk and RAM

Sequential vs Random Access

Due to the rotational nature of Magnetic Disks...

- ▶ Sequential reads/writes comparatively FAST
- ▶ Random reads/writes comparatively very SLOW

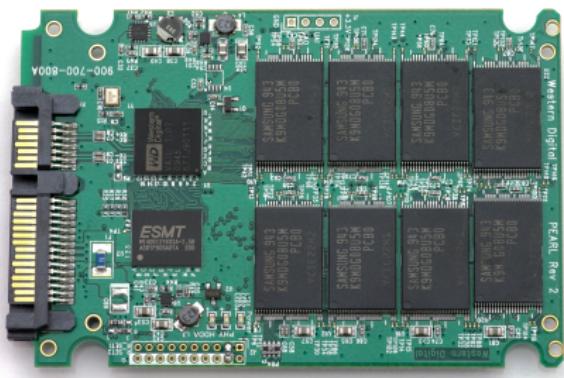
Solid State Drives

- ▶ No moving parts → speed
- ▶ Most use “flash” memory, non-volatile circuitry
- ▶ Major drawback: limited number of **writes**, disk wears out eventually
- ▶ Reads faster than writes
- ▶ Sequential somewhat faster than random access
- ▶ **Expensive:**

A 1TB internal 2.5-inch hard drive costs between \$40 and \$50, but as of this writing, an SSD of the same capacity and form factor starts at \$250. That translates into

- 4 to 5 cents/GB for HDD
- 25 cents/GB for the SSD.

PC Magazine, “SSD vs HDD” by Tom Brant and Joel Santo Domingo March 26, 2018

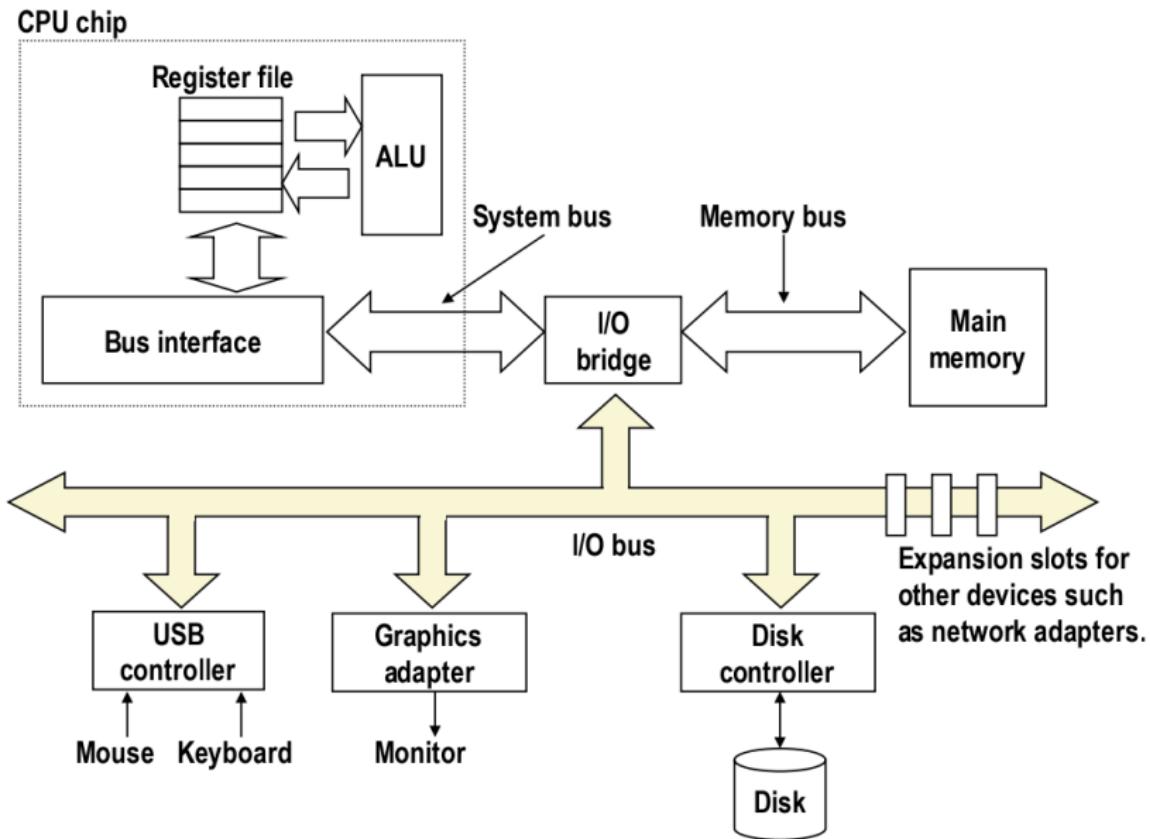


Tape Drives

- ▶ Slowest yet: store bits as magnetic field on a piece of “tape” a la 1980’s cassette tape / video recorder 
- ▶ Extremely cheap per GB so mostly used in backup systems
- ▶ Ex: CSELabs does nightly backups of home directories, recoverable from tape at request to Operator



The I/O System Connects CPU and Peripherals



Terminology

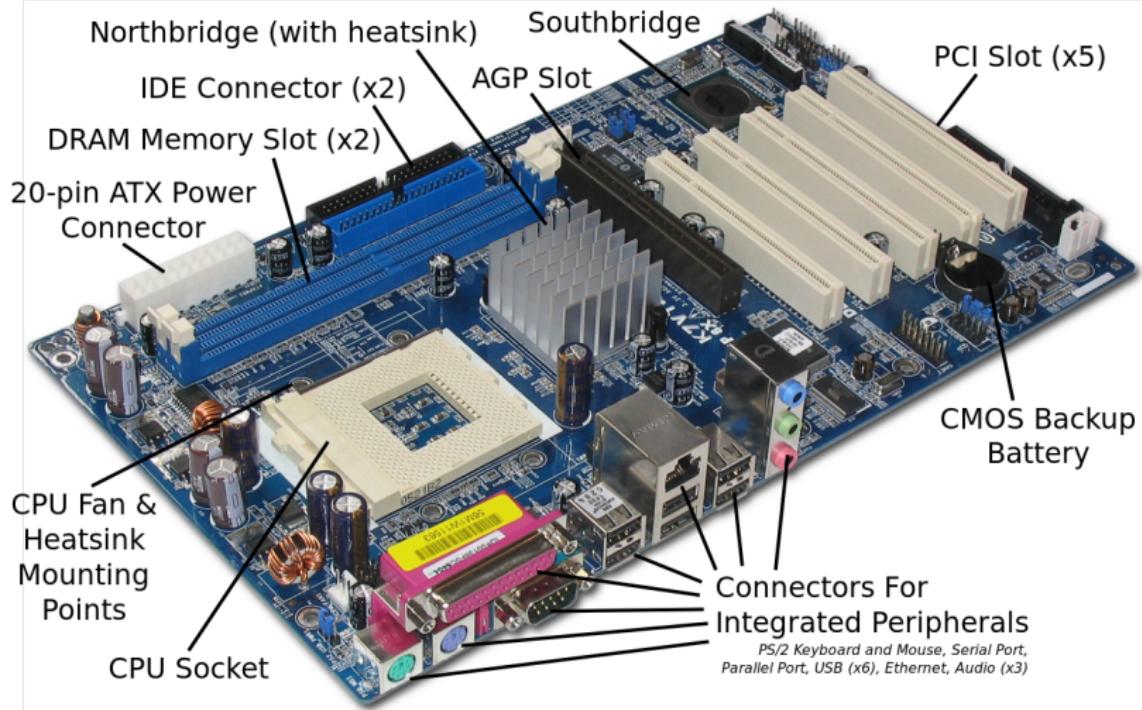
Bus A collection of wires which allow communication between parts of the computer. May be serial (single wire) or parallel (several wires), must have a communication protocol over it.

Bus Speed Frequency of the clock signal on a particular bus, usually different between components/buses requiring interface chips
CPU Frequency > Memory Bus > I/O Bus

Interface/Bridge Computing chips that manage communications across the bus possibly routing signals to correct part of the computer and adapting to differing speeds of components

Motherboard A printed circuit board connects to connect CPU to RAM chips and peripherals. Has buses present on it to allow communication between parts. *Form factor* dictates which components can be handled.

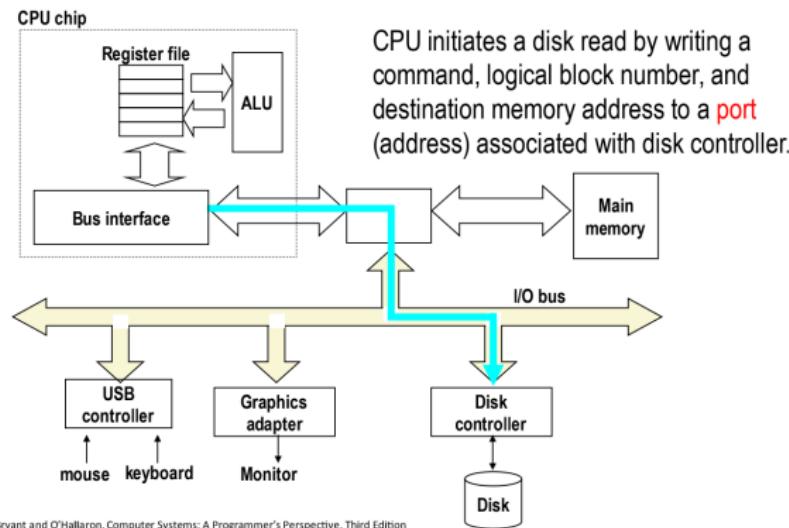
The Motherboard



Picture Source: Wikipedia
Live Props Courtesy of Free Geek Minneapolis

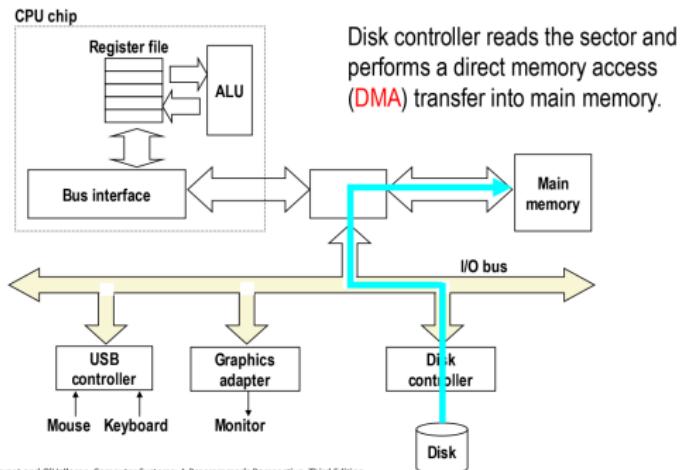
Memory Mapped I/O

- ▶ Modern systems are a collection of devices and microprocessors
- ▶ CPU usually uses **memory mapped I/O**: read/write certain memory addresses translated to communication with devices on I/O bus



Direct Memory Access

- ▶ Communication received by *other* microprocessors like a Disk Controller or Memory Management Unit (MMU)
- ▶ Other controllers may talk: Disk Controller loads data directly into Main Memory via **direct memory access**

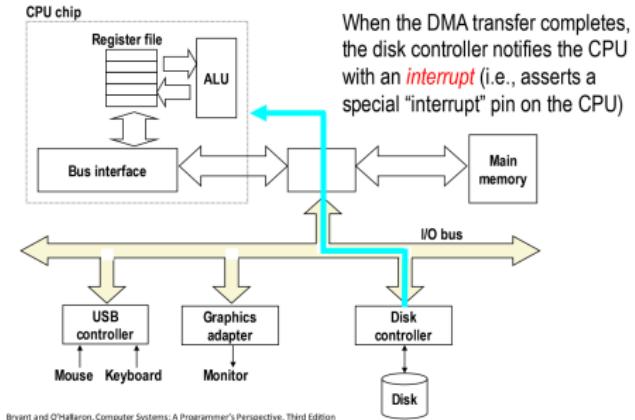


Interrupts and I/O

Recall access times

Place	Time
L1 cache	0.5 ns
RAM	100 ns
Disk	10,000,000 ns

- ▶ While running Program X, CPU reads an int from disk into %rax
- ▶ Communicates to disk controller to read from file
- ▶ Rather than wait, OS puts Program X to “sleep”, starts running program Y



Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

- ▶ When disk controller completes read, signals the CPU via an **interrupt**, electrical signals indicating an event
- ▶ OS handles interrupt, schedules Program X as “ready to run”

Interrupts from Outside and Inside

- ▶ Examples of events that generate interrupts
 - ▶ Integer divide by 0
 - ▶ I/O Operation complete
 - ▶ Memory address not in RAM (Page Fault)
 - ▶ User generated: x86 instruction int 80
- ▶ Interrupts are mainly the business of the Operating System
- ▶ Usually cause generating program to immediately transfer control to the OS for handling
- ▶ When building your own OS, must write “interrupt handlers” to deal with above situations
 - ▶ Divide by 0: **signal** program usually terminating it
 - ▶ I/O Complete: schedule requesting program to run
 - ▶ Page Fault: sleep program until page loaded
 - ▶ User generated: perform system call
- ▶ User-level programs will sometimes get a little access to interrupts via **signals**, a topic in many OS classes