CSCI 2021: Memory Systems

Chris Kauffman

Last Updated:
Mon Nov 21 01:16:25 PM CST 2022
Logistics

Reading Bryant/O’Hallaron

- Ch 4: Finish / Skim
- Ch 6: Memory

Lab / HW 11

- Lab 11: `clock()` function
- HW 11: Memory Optim
  Used in various ways for P4

Goals

- Timing code
- Cache Basics + Details
- 2D arrays + Cache
- Permanent Storage

Schedule

<table>
<thead>
<tr>
<th>Date</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fri 18-Nov</td>
<td>Memory+Storage</td>
</tr>
<tr>
<td>Mon 21-Nov</td>
<td>Storage+Micro Opt</td>
</tr>
<tr>
<td></td>
<td>Lab 12: Video Demo</td>
</tr>
<tr>
<td></td>
<td>No In-person Demo</td>
</tr>
<tr>
<td></td>
<td>TAs help w/ P4+Lab</td>
</tr>
<tr>
<td></td>
<td>Video Lecture</td>
</tr>
<tr>
<td>Thu 24-Nov</td>
<td>Thanksgiving</td>
</tr>
<tr>
<td>Fri 25-Nov</td>
<td>No classes</td>
</tr>
<tr>
<td>Mon 28-Nov</td>
<td>Micro-Opts</td>
</tr>
<tr>
<td>Tue 29-Nov</td>
<td>Unified Office Hours</td>
</tr>
<tr>
<td></td>
<td>Keller 3-180</td>
</tr>
<tr>
<td>Wed 30-Nov</td>
<td>Lab: Review</td>
</tr>
<tr>
<td></td>
<td>Lecture: Practice Exam 3</td>
</tr>
<tr>
<td></td>
<td>Project 4 Due</td>
</tr>
<tr>
<td>Fri 02-Dec</td>
<td>Exam 3</td>
</tr>
</tbody>
</table>
Announcements

None
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

```c
#include <time.h>

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

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

### Real (Wall) Time

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

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

```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;
}
```

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

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

   \[
   \text{Throughput} = \frac{\#\text{Additions}}{\text{Second}}
   \]

- 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

```c
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

- 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?
Increasing Efficiency

- Can increase the efficiency of loop summing with tricks
- B/O’H use multiple accumulators: multiple variables for summing
- Facilitates pipelining / superscalar processor
- Code is significantly faster BUT less readable
- This optimization can be performed by the compiler, will discuss later (among the many gcc optimization options, ~67 pages)

```c
// From Bryant/O'Hallaron
int sum_add4(int elems, int stride){
    int i,
        sx1 = stride*1, sx2 = stride*2,
        sx3 = stride*3, sx4 = stride*4,
        acc0 = 0, acc1 = 0,
        acc2 = 0, acc3 = 0;
    int length = elems;
    int limit = length - sx4;

    /* Combine 4 elements at a time */
    for (i = 0; i < limit; i += sx4) {
        acc0 = acc0 + data[i];
        acc1 = acc1 + data[i+sx1];
        acc2 = acc2 + data[i+sx2];
        acc3 = acc3 + data[i+sx3];
    }

    /* Finish any remaining elements */
    for (; i < length; i += stride) {
        acc0 = acc0 + data[i];
    }

    return acc0+acc1+acc2+acc3;
}
```
Temporal and Spatial Locality

- In the beginning, there was only CPU and Memory
- Both ran at about the same speed (same clock frequency)
- CPUs were easier to make faster, began outpacing speed of memory
- Hardware folks noticed programmers often write loops like
  
  ```
  for(int i=0; i<len; i++){
      sum += array[i];
  }
  ```
  
  which exhibits two Memory Locality features
  - **Temporal Locality**: memory recently used likely to be used again soon (like `sum` and `i` used in every loop iteration)
  - **Spatial Locality**: memory near to recently used memory likely to be used (like `arr[0]` first then `arr[1],arr[2]`)

- Register file and Cache were developed to exploit locality
The Memory Pyramid

- Processor
- CPU
  - Processor Register
  - CPU Cache
    - Level 1 (L1) Cache
    - Level 2 (L2) Cache
    - Level 3 (L3) Cache
- Physical Memory
  - RAM-DOM Access Memory (RAM)
- Solid State Memory
  - Non-Volatile Flash-Based Memory
- Virtual Memory
  - File-Based Memory

Source
Numbers Everyone Should Know

Edited Excerpt of Jeff Dean’s talk on data centers.

<table>
<thead>
<tr>
<th>Reference</th>
<th>Time</th>
<th>Analogy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>-</td>
<td>Your brain</td>
</tr>
<tr>
<td>L1 cache reference</td>
<td>0.5 ns</td>
<td>Your desk</td>
</tr>
<tr>
<td>L2 cache reference</td>
<td>7 ns</td>
<td>Neighbor’s Desk</td>
</tr>
<tr>
<td>Main memory reference</td>
<td>100 ns</td>
<td>This Room</td>
</tr>
<tr>
<td>Disk seek</td>
<td>10,000,000 ns</td>
<td>Salt Lake City</td>
</tr>
</tbody>
</table>

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.
Why isn’t Everything Cache?

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM $/MB</td>
<td>2,900</td>
<td>320</td>
<td>256</td>
<td>100</td>
<td>75</td>
<td>60</td>
<td>25</td>
<td>116</td>
</tr>
<tr>
<td>SRAM access (ns)</td>
<td>150</td>
<td>35</td>
<td>15</td>
<td>3</td>
<td>2</td>
<td>1.5</td>
<td>1.3</td>
<td>115</td>
</tr>
<tr>
<td>DRAM $/MB</td>
<td>880</td>
<td>100</td>
<td>30</td>
<td>1</td>
<td>0.1</td>
<td>0.06</td>
<td>0.02</td>
<td>44,000</td>
</tr>
<tr>
<td>DRAM access (ns)</td>
<td>200</td>
<td>100</td>
<td>70</td>
<td>60</td>
<td>50</td>
<td>40</td>
<td>20</td>
<td>10</td>
</tr>
</tbody>
</table>

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

---

“What Every Programmer Should Know About Memory” by Ulrich Drepper, Red Hat, Inc.
Cache Principles: Hits and Misses

CPU-Memory is a Client-Server

▶ CPU makes requests
▶ Memory system services request as fast as possible

Cache Hit

▶ CPU requests memory at address 0xFFFF1234 be loaded into register %rax
▶ Finds valid data for 0xFFFF1234 in L1 Cache: L1 Hit
▶ Loads into register fast

Cache Miss

▶ CPU requests memory at address 0xFFFF7890 be loaded into register %rax
▶ 0xFFFF7890 not in L1 Cache: L1 Miss
▶ Search L2: if found move into L1, then %rax
▶ Search L3: if found move into L2, L1, %rax
▶ Search main memory: if found, move into caches, if not...

Wait, how could 0xFFFF7890 not be in main memory...?
Types of Cache Misses

Compulsory “Cold” Miss: Program Getting Started

- All cache entries start with valid=0: cache contains leftover garbage from previous program runs
- After the cache “warms up” most entries will have Valid=1, data for running program

Capacity Miss: Data Too Big to Fit

- **Working set** is set of memory being frequently accessed in a particular phase of a program run
- Large working set may exceed the size of a cache causing misses

Conflict Miss: This Stall Occupied

- Internal **placement policy** of cache dictates where data goes
- If two needed piece of data both go to the same position in cache, leads to misses as they overwrite each other
Diagram of Direct Mapped Cache

Main Memory

Address

0x0000
0x0020
0x0040
0x0060
0x0080
0x00A0
0x00C0
0x00E0
0x0100
0x0120
0x0140
0x0160
...

Direct-Mapped Cache

Line

0
1
2
3

Cache Data Block

Each region of memory maps to exactly one cache line.

Source: Dive into Systems dot org
Cache is like a **Hash Table**

- Each Main Memory address has some bits indicating
  - **Set** - where in cache data should go
  - **Tag** - identifier to track what’s in cache

Each cache Set can hold 1 or more **Lines** of data with a specific Tag

Main Memory divides into cache **Blocks** which share Tag/Set and move in/out of cache together

- Bits from address determine location for memory in cache
- Direct-Mapped cache, 4 sets and 16 byte blocks/lines
- Load address 0x28

\[
028 \quad \begin{array}{c|c|c}
0 & \text{Offset} & 8 \\
\hline
2 & \text{Set} & 10 \\
\hline
8 & \text{Tag} & 1000 \\
\end{array}
\]

- 0x28 = 00101000
- 0x20 in the same line, will also be loaded into set #2
## Exercises: Anatomy of a Simple CPU Cache

### MAIN MEMORY

<table>
<thead>
<tr>
<th>Addr</th>
<th>Addr Bits</th>
<th>Value</th>
<th>Blocks/Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00 00 0000</td>
<td>331</td>
<td></td>
</tr>
<tr>
<td>08</td>
<td>00 00 1000</td>
<td>332</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>00 01 0000</td>
<td>333</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>00 01 1000</td>
<td>334</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>00 10 0000</td>
<td>335</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>00 10 1000</td>
<td>336</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>00 11 0000</td>
<td>337</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>00 11 1000</td>
<td>338</td>
<td></td>
</tr>
<tr>
<td>C0</td>
<td>11 00 0000</td>
<td>551</td>
<td></td>
</tr>
<tr>
<td>C8</td>
<td>11 00 1000</td>
<td>552</td>
<td></td>
</tr>
<tr>
<td>D0</td>
<td>11 01 0000</td>
<td>553</td>
<td></td>
</tr>
<tr>
<td>D8</td>
<td>11 01 1000</td>
<td>554</td>
<td></td>
</tr>
<tr>
<td>E0</td>
<td>11 10 0000</td>
<td>555</td>
<td></td>
</tr>
<tr>
<td>E8</td>
<td>11 10 1000</td>
<td>556</td>
<td></td>
</tr>
<tr>
<td>F0</td>
<td>11 11 0000</td>
<td>557</td>
<td></td>
</tr>
<tr>
<td>F8</td>
<td>11 11 1000</td>
<td>558</td>
<td></td>
</tr>
</tbody>
</table>

### CACHE

<table>
<thead>
<tr>
<th>Set</th>
<th>V</th>
<th>Tag</th>
<th>Blocks/Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>00</td>
<td>333 334</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>11</td>
<td>555 556</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>00</td>
<td>337 338</td>
</tr>
</tbody>
</table>

- Direct-mapped Cache
  - Direct-mapped: 1 Line per Set
  - 16-byte lines = 4-bit offset
  - 4 Sets = 2-bit index
  - 8-bit Address = 2-bit tag
  - Total Cache Size = 64 bytes
    - 4 sets * 16 bytes

**HITS OR MISSES? Show effects**

1. Load 0x08
2. Load 0xF0
3. Load 0x18
## Answers: Anatomy of a Simple CPU Cache

### MAIN MEMORY

<table>
<thead>
<tr>
<th>Addr</th>
<th>Addr Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00 00 0000</td>
<td>331</td>
</tr>
<tr>
<td>08</td>
<td>00 00 1000</td>
<td>332</td>
</tr>
<tr>
<td>10</td>
<td>00 01 0000</td>
<td>333</td>
</tr>
<tr>
<td>18</td>
<td>00 01 1000</td>
<td>334</td>
</tr>
<tr>
<td>20</td>
<td>00 10 0000</td>
<td>335</td>
</tr>
<tr>
<td>28</td>
<td>00 10 1000</td>
<td>336</td>
</tr>
<tr>
<td>30</td>
<td>00 11 0000</td>
<td>337</td>
</tr>
<tr>
<td>38</td>
<td>00 11 1000</td>
<td>338</td>
</tr>
<tr>
<td></td>
<td>.. .</td>
<td></td>
</tr>
<tr>
<td>C0</td>
<td>11 00 0000</td>
<td>551</td>
</tr>
<tr>
<td>C8</td>
<td>11 00 1000</td>
<td>552</td>
</tr>
<tr>
<td>D0</td>
<td>11 01 0000</td>
<td>553</td>
</tr>
<tr>
<td>D8</td>
<td>11 01 1000</td>
<td>554</td>
</tr>
<tr>
<td>E0</td>
<td>11 10 0000</td>
<td>555</td>
</tr>
<tr>
<td>E8</td>
<td>11 10 1000</td>
<td>556</td>
</tr>
<tr>
<td>F0</td>
<td>11 11 0000</td>
<td>557</td>
</tr>
<tr>
<td>F8</td>
<td>11 11 1000</td>
<td>558</td>
</tr>
</tbody>
</table>

### CACHE

<table>
<thead>
<tr>
<th>Set</th>
<th>V</th>
<th>Tag</th>
<th>Blocks/Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>*00</td>
<td>331 332</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>00</td>
<td>333 334</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>11</td>
<td>555 556</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>*11</td>
<td>557 558</td>
</tr>
</tbody>
</table>

**DIRECT-MAPPED Cache**
- Direct-mapped: 1 line per set
- 16-byte lines = 4-bit offset
- 4 Sets = 2-bit index
- 8-bit Address = 2-bit tag
- Total Cache Size = 64 bytes
  - 4 sets * 16 bytes

**HITS OR MISSES? Show effects**
1. Load 0x08: MISS to set 00
2. Load 0xF0: MISS overwrite set 11
3. Load 0x18: HIT in set 01 no change
### Direct vs Associative Caches

#### Direct Mapped

One line per set

<table>
<thead>
<tr>
<th>Set</th>
<th>V</th>
<th>Tag</th>
<th>Blocks/Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>00</td>
<td>333 334</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>11</td>
<td>555 556</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>00</td>
<td>337 338</td>
</tr>
</tbody>
</table>

- **Simple circuitry**
- **Conflict misses** may result: 1 slot for many possible tags
- **Thrashing**: need memory with overlapping tags

vv

\[
0x10 = 00 01 0000 : \text{in cache}
\]

\[
0xD8 = 11 01 1000 : \text{conflict}
\]

#### N-Way Associative Cache

Ex: 2-way = 2 lines per set

<table>
<thead>
<tr>
<th>Set</th>
<th>V</th>
<th>Tag</th>
<th>Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>11</td>
<td>551 552</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>11</td>
<td>553 554</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>00</td>
<td>337 338</td>
</tr>
</tbody>
</table>

- **Complex circuitry → $$**
- Requires an **eviction policy**, usually least recently used
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?

```c
// 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);
```

```c
// 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 Mallocls**
- $100 \times 8 = 800$ bytes
- $30 \times 4 = 120$ bytes
- $30 \times 4 = 120$ bytes
- $30 \times 4 = 120$ bytes

**Two Mallocls**
- $100 \times 8 = 800$ bytes
- $100 \times 30 \times 4 = 12000$ bytes
Single Malloc Matrices

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

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

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

HW11/P4 will use this technique along with some structs and macros to make it more readable:

```c
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,…
  - `mat[i]` is a contiguous row, `mat[i][j]` is an element

- Numerically-oriented languages use **Column-Major** order
  - Fortran, Matlab/Octave, R, Ocaml (?)...  
  - `mat[j]` is a contiguous **column**, `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?

```java
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

```bash
> 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
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?
Exercise: perf stats for sumR vs sumC, what’s striking?

> 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%)

29
**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 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
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

- CPU chip
  - Register file
  - ALU
- Bus interface
- System bus
- Memory bus
- I/O bridge
- Main memory
- I/O bus
- Expansion slots for other devices such as network adapters.
- USB controller
  - Mouse
  - Keyboard
- Graphics adapter
  - Monitor
- Disk controller
  - Disk
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

- Northbridge (with heatsink)
- Southbridge
- IDE Connector (x2)
- AGP Slot
- DRAM Memory Slot (x2)
- PCI Slot (x5)
- CMOS Backup Battery
- 20-pin ATX Power Connector
- Connectors For Integrated Peripherals: PS/2 Keyboard and Mouse, Serial Port, Parallel Port, USB (x6), Ethernet, Audio (x3)
- CPU Fan & Heatsink Mounting Points
- CPU Socket

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

CPU initiates a disk read by writing a command, logical block number, and destination memory address to a port (address) associated with disk controller.
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

<table>
<thead>
<tr>
<th>Place</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache</td>
<td>0.5 ns</td>
</tr>
<tr>
<td>RAM</td>
<td>100 ns</td>
</tr>
<tr>
<td>Disk</td>
<td>10,000,000 ns</td>
</tr>
</tbody>
</table>

- 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

- 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 for CSCI 4061