Section Notes - Let's talk about caching

1. Caches - Some background

Cache - Fast storage that stores temporary copies of data from slower storage

An access (a read or write) is a ***cache hit*** when the cache can satisfy the access without contacting slower storage. Cache hits are good since they're fast. An access that's not a hit is a ***miss***. Misses are bad since they require contacting the next level of storage, which is slow.

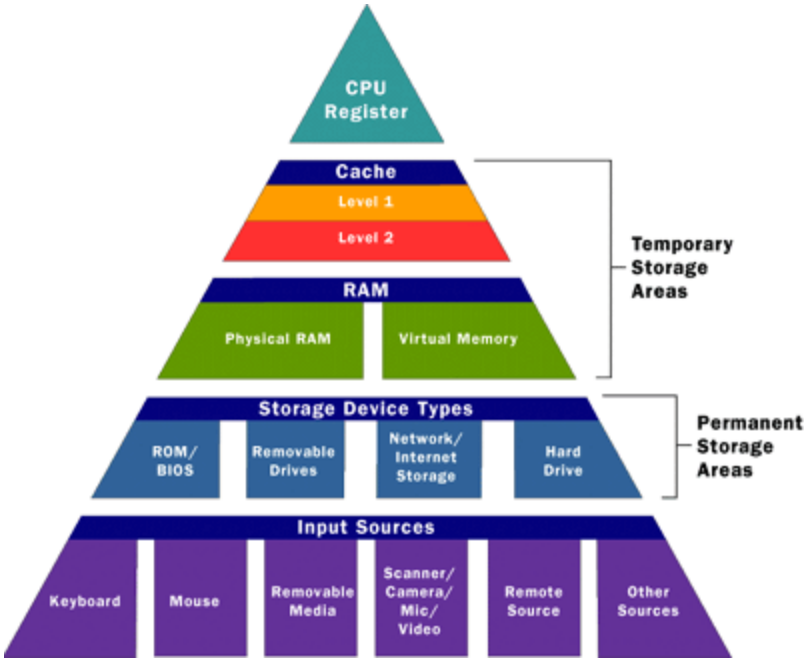
Caches allow us to store data closer to where it is actively being used, and therefore allow retrieval of the data to be faster. An analogy of a cache would be your backpack. You keep some books and other items in your bag so that you can quickly retrieve the items when you are at class. It would be extremely inconvenient and slow to go back to your room every time you needed an item. Thus, your backpack is a cache for your room.

Taking this to another level, you can think of your school dorm room as a cache for your room back in your hometown. If you didn't live in Boston, it would be even more slow and inconvenient to have to go back to your hometown every time you needed an item—instead, you ***speculate*** about what you would need for the semester and take what you need with you when you move into your dorm room. Note that for each level of our "cache"—your backpack for your dorm room, and your dorm room for your hometown—the capacity is smaller, but a lot faster to access.

The computer memory hierarchy acts in a very similar way. Below is a diagram. The higher you go up, the faster it is to access items. The lower you go up, the cheaper it is to add more storage in that layer.[1]

KEY INSIGHT: Each level is a cache for the levels below! Thus, L1 is a cache for L2 which is a cache for L3 … primary storage (RAM) can be thought of as a cache for disk.

[1] There are limits to the size of each layer, particularly in the lower levels of the cache, that are physical rather than purely financial. L1 cache is generally on the chip which has limited area.



2. Request Costs - Lecture review

If we have:

N = Number of requests in the series
U = Size of each unit in N, for some definition of “unit”
R = Per request cost
K = Per unit cost
C = total cost

Then:

C = NR + NUK

Exercise 1:
Suppose we are writing a file of size $\mathbf{10^6}$ bytes to disk. Similar to w01-syncbyte.c, we write each byte individually and synchronously to disk, and we observed the total time to write all the bytes was **1.01s**. Similar to w02-syncblock.c, we instead write blocks of **size 1000** to disk. We observed that the total time to write all the bytes to disk takes **0.011s**.

QUESTION: Write down an equation in terms of N, R, U, and K that describes how long it took to write each byte individually to disk.

QUESTION: Write down an equation in terms of N, R, U, and K that describes how long it took to write blocks of bytes of size 1000 to disk.

QUESTION: What is the per request cost (R) and per unit cost (K) of the system?

QUESTION: How long would it take if we instead wrote blocks of bytes of size 4000 to disk?

Exercise 2:
You decide it would be nice to hold your VM appliance image on a USB drive so you can do your cs61 programming everywhere you go. Its a pretty big file, 1627275264 bytes. You start copying it over but realize you used the wrong U size, you think 256 will take too long. **If R = 2.2 microseconds and K = 13 nanoseconds, how long after you start, will it still be faster to stop the current U=256 byte copy and restart from the beginning with U=4096 bytes?** (assume that you can switch the block size and restart the copy instantly)

3. Coherence

Note: For the remaining exercises we will be using the code in the section repository. You can pull the new code or add the repository if you haven't already:
git clone git@code.seas.harvard.edu:cs61/cs61-section.git

In lecture the concept of cache coherency was introduced. **Cache coherence** is the property that cached data is consistent with other copies of the same data in other layers of memory. We can explore the concept with a few programs. We have four programs:

1. readbytes.c - Reads one byte at a time and pauses for 0.5 seconds.
2. freadbytes.c - Also reads one byte at a time and pauses for 0.5 seconds but uses stdio.
3. writebytes.c - Writes one byte (the character '7') at a time and pauses for 0.25 seconds.
4. fwritebytes.c - Also writes one byte (the character '7') at a time and pauses for 0.25 seconds but uses stdio.

We are supplied a data file where each byte is the character '6'.

What would you expect to see if:
(A) you run (1) and (3) concurrently (coherence or incoherence)?

(B) you ran (2) and (3) concurrently?

(C) you ran (1) and (4) concurrently?

(D) you ran (2) and (4) concurrently?

4. Memory caches and locality

We spoke briefly last week about locality. Recall that there are two types of locality with regard to caches:

- Spatial
- Temporal

A concept that is related to to spatial locality is strided access patterns. A **stride-k** access pattern accesses every k^{th} byte. Stride-1 is sequential access. Stride-1024 accesses every 1024^{th} byte (or, more generally, accesses a word or so every 1024^{th} byte).

Maintaining data access locality throughout a program can have a significant impact on performance.

QUESTION: Does good locality help with latency? throughput? space (memory usage)?

QUESTION: What is thrashing?

Locality is important at all levels of the memory hierarchy. Although your problem set, and lecture so far, have focused on the buffer cache (the operating system's cache for disk files), modern processors are so fast that they have caches for memory too. All the ideas behind caching apply to processor caches! And there are programs where better use of processor caches can speed up program performance dramatically.

(Of course each different type of cache has some properties specific to that type. The book's presentation, for example, is very thorough on properties of processor caches.)

So let's look at a program where changing locality of references *to main memory* can have dramatic performance impact.

Consider matrix multiplication of two 3x3 matrices such that C = AB:

$$\begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}$$

where:

$$\begin{aligned} c_{11} &= a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} \\ c_{12} &= a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} \\ c_{13} &= a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33} \\ &\dots \\ c_{33} &= a_{31}b_{13} + a_{32}b_{23} + a_{33}b_{33} \end{aligned}$$

We can implement this matrix multiplication using three 'for' loops which we will identify by their index, i, j, and k. Note that there are 3! ways to permute the set of indices. With some slight changes we can implement the three loops in each of the 6 orders.

(a) Version ijk

code/mem/matmult/mm.c

```
1  for (i = 0; i < n; i++)
2      for (j = 0; j < n; j++) {
3          sum = 0.0;
4          for (k = 0; k < n; k++)
5              sum += A[i][k]*B[k][j];
6          C[i][j] += sum;
7      }
```

code/mem/matmult/mm.c

(b) Version jik

code/mem/matmult/mm.c

```
1  for (j = 0; j < n; j++)
2      for (i = 0; i < n; i++) {
3          sum = 0.0;
4          for (k = 0; k < n; k++)
5              sum += A[i][k]*B[k][j];
6          C[i][j] += sum;
7      }
```

code/mem/matmult/mm.c

(c) Version jki

code/mem/matmult/mm.c

```
1  for (j = 0; j < n; j++)
2      for (k = 0; k < n; k++) {
3          r = B[k][j];
4          for (i = 0; i < n; i++)
5              C[i][j] += A[i][k]*r;
6      }
```

code/mem/matmult/mm.c

(d) Version kji

code/mem/matmult/mm.c

```
1  for (k = 0; k < n; k++)
2      for (j = 0; j < n; j++) {
3          r = B[k][j];
4          for (i = 0; i < n; i++)
5              C[i][j] += A[i][k]*r;
6      }
```

code/mem/matmult/mm.c

(e) Version kij

code/mem/matmult/mm.c

```
1  for (k = 0; k < n; k++)
2      for (i = 0; i < n; i++) {
3          r = A[i][k];
4          for (j = 0; j < n; j++)
5              C[i][j] += r*B[k][j];
6      }
```

code/mem/matmult/mm.c

(f) Version ikj

code/mem/matmult/mm.c

```
1  for (i = 0; i < n; i++)
2      for (k = 0; k < n; k++) {
3          r = A[i][k];
4          for (j = 0; j < n; j++)
5              C[i][j] += r*B[k][j];
6      }
```

code/mem/matmult/mm.c

Figure 6.46 **Six versions of matrix multiply.** Each version is uniquely identified by the ordering of its loops.

We'll refer to each of the implementations by their three letter permutation.

Before we try to figure out which will be the best performing, let's recall how the arrays will be laid out in memory. Each of A, B, and C, are of type "array of array of double"[2] which means a matrix that we think of as:

$$\begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

will be laid out in memory as:

c_{11}	c_{12}	c_{13}	c_{21}	c_{22}	c_{23}	c_{31}	c_{32}	c_{33}

where each $c_{RowColumn}$ is a double. If we call this matrix C we would access element c_{23} with the array notation C[2][3].

QUESTION: Which loop order of matrix multiple do you expect to perform best? worst? why?

First lets look at the driver: matrixmultiply-main.c

Now open and inspect the first implementation: matrixmultiply-jki.c

Before you run anything, which implementation do you think will be fastest, and why?

Run each of the 6 basic matrix multiply programs and record the run time.

time ./matrixmultiply-jki

Explain the differences that you see in the trials and correlate them with program code.

Does immediately rerunning the program result in a speedup? Why or why not?

Do you see the same differences when you run with a smaller matrix? Why or why not?

time ./matrixmultiply-jki 100

Now that we have coalesced memory accesses, how might we make the code faster? Let's look at the cse versions of the tests. matrixmultiply-jki-cse.c

How do we match up against a matrix library? time ./matrixmultiply-blas

Can we beat the library? Look at matrixmultiply-ikj-extraopt.c

---

[2] All of our examples will use double. Though we could, of course, use int, unsigned, etc.