CoE 163

Computing Architectures and Algorithms

Running Linear Algebra Operations in a Computer
(and things we need to consider)

Previous discussion was on linear algebra

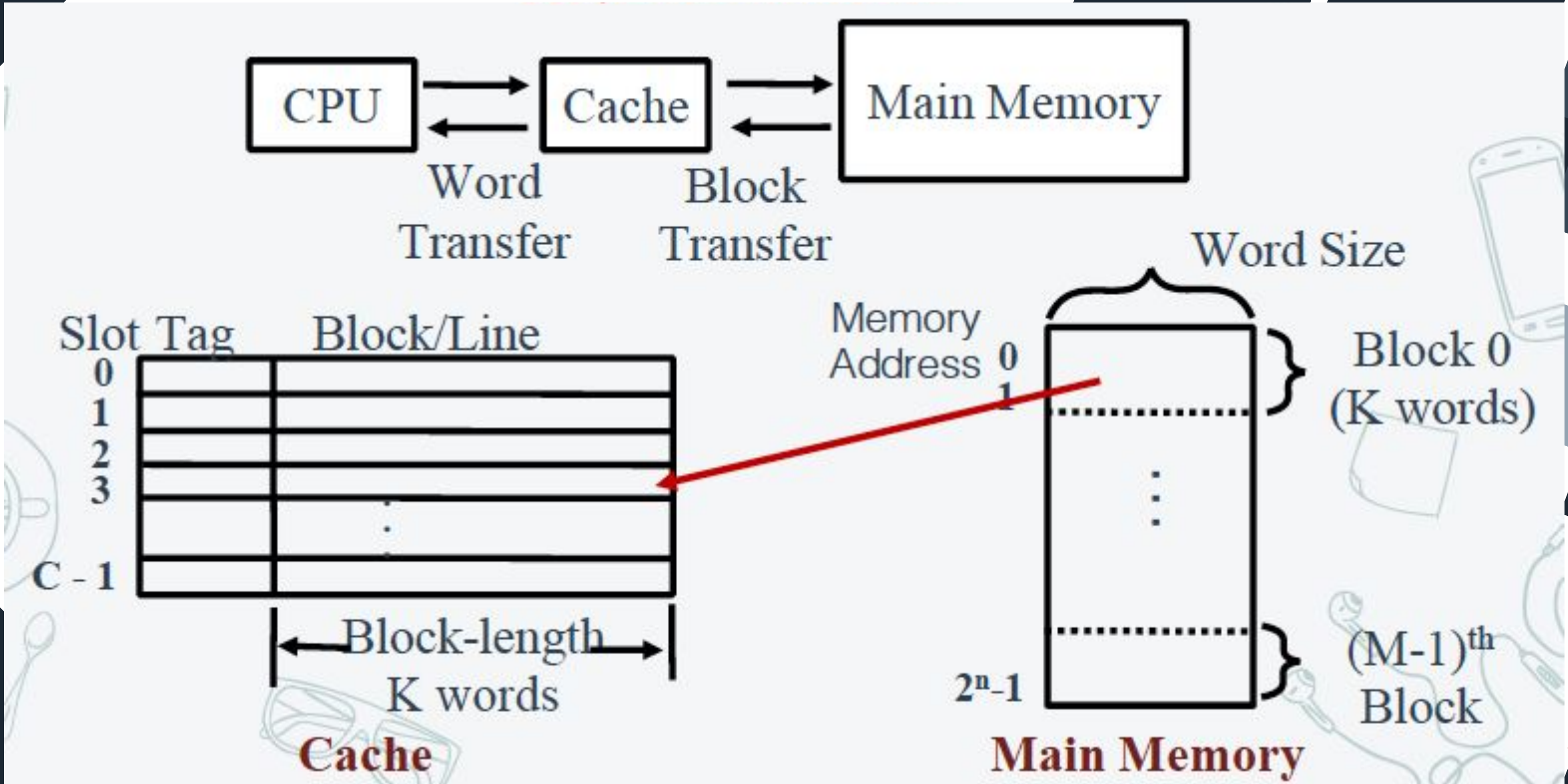
- We learn linear algebra understanding how to do the computations by hand
- Many considerations arise when we have to create computer algorithms for these operations
- With larger matrices / data, we need to consider how to optimize our algorithms

Numerical Linear Algebra

- Specific branch of linear algebra that deals with the following questions:
  - How can we create computer algorithms around matrix operations?
  - How can these algorithms efficiently and accurately solve problems?
  - How can these algorithms approximate the answers that can be obtained in **continuous** mathematics?

First: brief review of computer memory organization and behavior

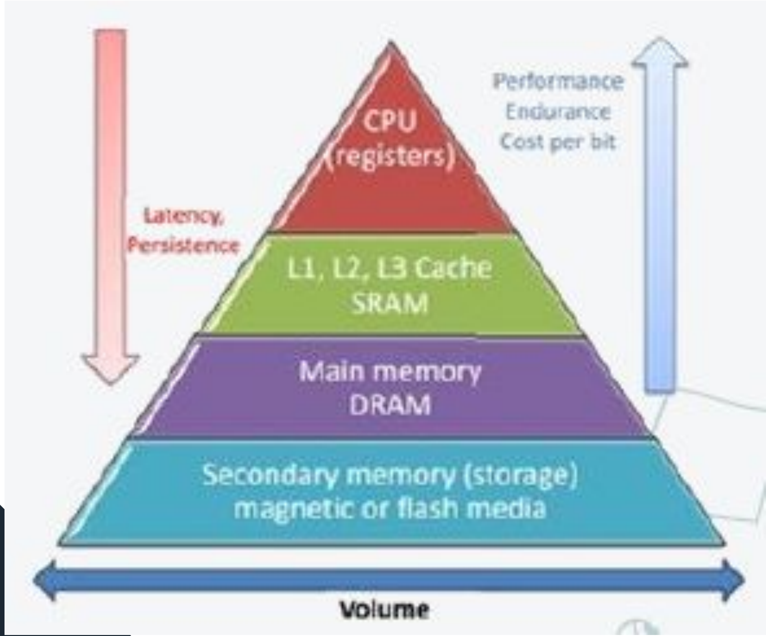
Typical Organization of Computer Memory

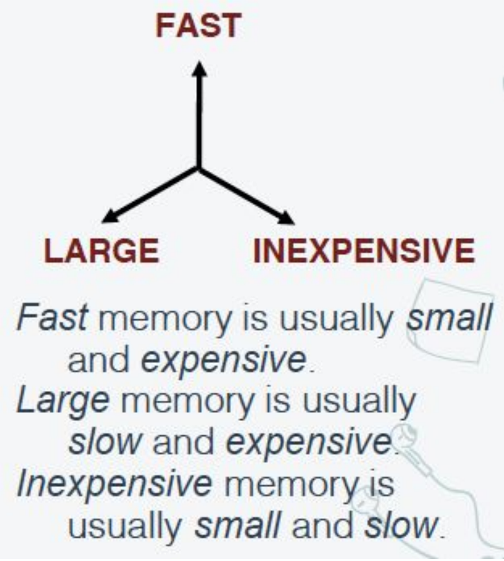


("stolen" from EEE 153 materials)

Memory Hierarchy

Caches (faster memory) are introduced to speed up computer operations





("stolen" from EEE 153 materials)

Locality of Reference

Temporal locality - recently executed instructions (or accessed data) are likely to be executed (or accessed) soon

Spatial locality – instructions/data in close proximity to a recently executed (or accessed) instruction/data are likely to be executed (or accessed) soon

How can we do matrix computations with acceptable speed and acceptable accuracy?

Key question asked in fast.ai course: *Computational Linear Algebra* by Rachel Thomas, 2017. [https://www.fast.ai/2017/07/17/num-lin-alg/]

Things to consider when doing matrix operations on computers

- Accuracy
- Memory use
- Speed
- Scalability

Accuracy

Exercise

Look at the function below. On paper, determine the expected output if we set `f = 0.1`

```
def f(x):
    if x <= 1/2:
        return 2 * x
    if x > 1/2:
        return 2*x - 1
```

Example from *Numerical Methods*, by Greenbaum and Chartier

Accuracy

Exercise

Run the code below in python:

```
def f(x):
    if x <= 1/2:
        return 2 * x
    if x > 1/2:
        return 2*x - 1


x = 0.1
for i in range(80):
    print(x)
    x = f(x)
```

Example from *Numerical Methods*, by Greenbaum and Chartier

Accuracy

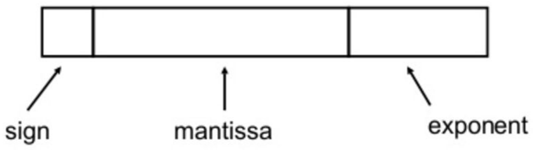
- Did you get the expected output?
- What went wrong?

Example from *Numerical Methods*, by Greenbaum and Chartier

Accuracy

- Math is infinite and continuous while computers are finite and discrete
- Limitations in storing/representing numbers
  - Remember floating point representation from your EEE 143 lessons?

- Floating point numbers have three parts:
  sign, **mantissa**, and **exponent**
- The base (radix) is assumed (usually base 2).
- The sign is a single bit (0 for positive number, 1 for negative).



Memory use

- We can save memory space if we store only the non-zero elements of matrices
- This is especially useful for sparse matrices where most of the elements are zero
- Will go back to this in the succeeding weeks

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & 0 & \frac{1}{4} & \frac{1}{4} \end{pmatrix}$$

Speed

- How can you make the computation/algorithm faster?
  - Choice of algorithm
  - Opportunities for parallelization
  - Locality (moving things around in memory, using what is in the cache immediately instead of discarding and reloading to cache)

More on Locality

- Computers have fast storage and slow storage
  - Check out: https://colin-scott.github.io/personal_website/research/interactive_latency.html
- When data is in fast storage (cache) we want to run our computations right away, before it gets bumped off (we don’t want to have to reload it into cache)
- For some storage, it is faster to access data items that are next to each other
- Trade off for optimizing locality: may lose opportunity to parallelize (see next slides)

Scalability

- Can we scale our algorithm over multiple cores or multiple computers over a network?
- Can we parallelize?
- Scalable algorithms:
    - input can be broken up into smaller pieces, can be handled by a different core/computer, and then are put back together at the end

To further demonstrate the impact of locality, and tradeoff with parallelization, take a moment to watch the following talk (~25 minutes)

https://youtu.be/3uiEyEKji0M

Takeaways

- We can improve the accuracy and efficiency of linear algebra algorithms if we consider that computers are finite and discrete when we craft our algorithms
- Memory considerations:
    - size limits
    - speed at different levels of memory hierarchy
    - optimizing for locality in memory can reduce scalability across cores