Lecture 14 – Architecture and Performance

Prof. Brendan Kochunas

NERS/ENGR 570 - Methods and Practice of Scientific Computing (F20)

COLLEGE OF ENGINEERING
NUCLEAR ENGINEERING & RADIOLOGICAL SCIENCES
UNIVERSITY OF MICHIGAN

Outline

- Motivating example--Understanding results of Lab 4
- Basic notions of performance (for serial execution)
- Computer Architecture
  - Idealized Models
  - Really what's going on
  - Practical Mental Models of Architecture

Learning Objectives: By the end of Today's Lecture you should be able to

- (Knowledge) explain the results observed in lab 4
- (Knowledge) describe the basic pieces of a single core architecture
- (Knowledge) define some performance metrics
- (Value) understand what things are important to focus on when trying to improve code performance

Motivation

Matrix-matrix Multiply

- Lab 4
  - Write your own matrix-matrix multiply
  - Compare to `dgemm` from BLAS
    - For system BLAS library and OpenBLAS
    - Compare to NumPy

$$\mathbf{AB} = \mathbf{C} \rightarrow c_{i,j} = \sum_{k} a_{i,k} b_{k,j}$$

Results for Lab 4

Matrix Size	My Implementations			System BLAS	OpenBLAS
100	0.001	0.001	0.003	0.001	0.064
500	0.062	0.068	0.465	0.071	0.069
1000	0.505	0.559	4.109	0.575	0.291
2000	5.516	6.160	39.028	5.787	0.749
4000	47.922	50.341	>100	46.229	2.822

Results obtained on Great Lakes

Clearly something is going on here...

Matrix-matrix Multiply

- Lab 4
  - Write your own matrix-matrix multiply
  - Compare to `dgemm` from BLAS
    - For system BLAS library and OpenBLAS
    - Compare to NumPy

$$\mathbf{AB} = \mathbf{C} \rightarrow c_{i,j} = \sum_k a_{i,k} b_{k,j}$$

```
do i=1,n
  do j=1,n
    do k=1,n
      c(i,j)=c(i,j)+a(i,k)*b(k,j)
    enddo
  enddo
enddo
```

```
do k=1,n
  do j=1,n
    do i=1,n
      c(i,j)=c(i,j)+a(i,k)*b(k,j)
    enddo
  enddo
enddo
```

```
c=MATMUL(A,B)
```
Fortran one-liner

What is NumPy doing?

```
$ module load python3.7-anaconda
$ python
>>> import numpy
>>> numpy.__config__.show()
blas_mkl_info:
    libraries = ['mkl_rt', 'pthread']
    library_dirs = ['/sw/arcts/centos7/python3.7-anaconda/2020.02/lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['/sw/arcts/centos7/python3.7-anaconda/2020.02/include']
blas_opt_info:
    libraries = ['mkl_rt', 'pthread']
    library_dirs = ['/sw/arcts/centos7/python3.7-anaconda/2020.02/lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['/sw/arcts/centos7/python3.7-anaconda/2020.02/include']
lapack_mkl_info:
    libraries = ['mkl_rt', 'pthread']
    library_dirs = ['/sw/arcts/centos7/python3.7-anaconda/2020.02/lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['/sw/arcts/centos7/python3.7-anaconda/2020.02/include']
lapack_opt_info:
    libraries = ['mkl_rt', 'pthread']
    library_dirs = ['/sw/arcts/centos7/python3.7-anaconda/2020.02/lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['/sw/arcts/centos7/python3.7-anaconda/2020.02/include']
```

- Turns out...
  - NumPy uses Intel MKL!
- Anaconda (and probably Canopy) distributions of NumPy also include Intel MKL libraries!
- MKL has a free community edition library.

Performance Basics

To understand the performance you observe,
you first must understand what performance to expect

To have an expectation of performance,
we all must be on the same page about what performance *is*

What is performance?

What does it mean for a code to be fast?

- The real metric: Time
- Derived metrics
  - *FLOPS* = FLoating Point OPerations per Second
  - *Bandwidth* = data per unit time (sort of like a flow rate)
  - *Latency* = Minimum time for data to travel from point A to point B
- Theoretical Peak Performance
  - *Very* difficult to achieve in practice
  - Can be computed from hardware specs
- Do things efficiently in time

How do you get fast code?

- First: Choose the right algorithm
- Second: Understand how to express that algorithm in the programming language
- Third: Understand how the source code will get mapped to the hardware
- Fourth: Tune the code to the hardware
- A lot of this can be done with pen and paper

Definitions of Performance Metrics

- FLOPS: Floating Point Operations per Second

$$\text{FLOPS} = \frac{\text{FLOPs}}{\text{Time}}$$

- Bandwidth: data transmitted per unit time

$$\text{MB/s} = \frac{\text{Data [MB]}}{\text{Time } [s]}$$

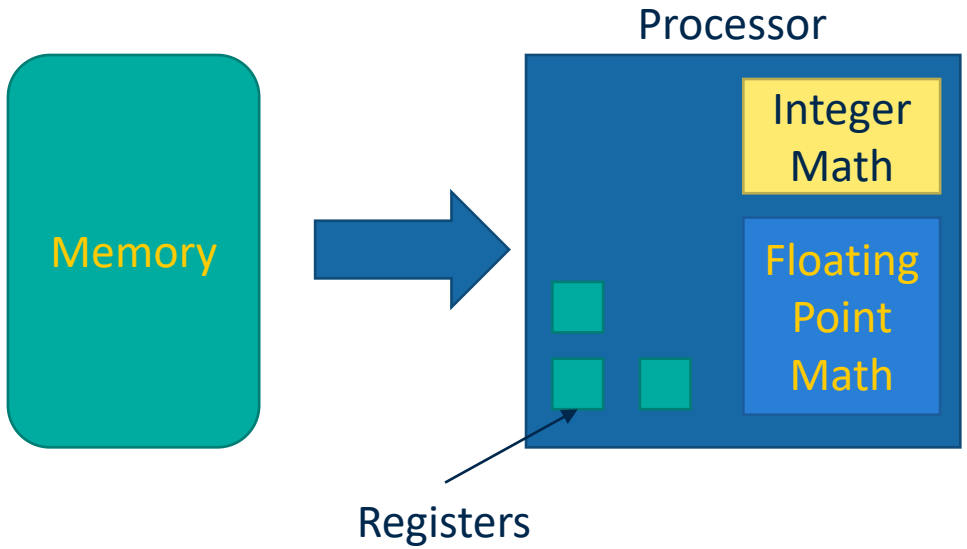
- Latency: time it takes data to go from A to B

- Fraction of Peak Performance $P_{peak} = \frac{\text{FLOPS}_{observed}}{\text{FLOPS}_{theoretical}}$

Computer Architecture

Idealized Processor Model

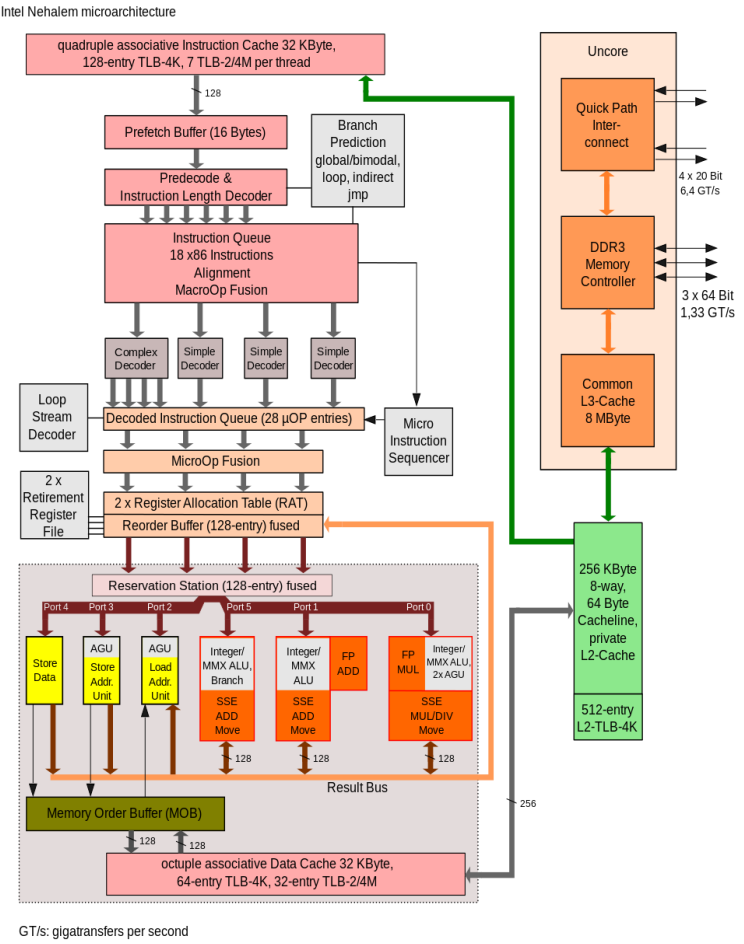
- Processor names bytes, words, etc. in its address space
  - These represent integers, floats, pointers, arrays, etc.
- Operations include
  - Read and write into very fast memory called registers
  - Arithmetic and other logical operations on register
- Order specified by program
  - Read and returns the most recently written data
  - Compiler and architecture translate high level expressions into “obvious” Lower level instructions
  - Hardware executes instructions in order specified by compiler
- Idealized Cost
  - Each operation has roughly the same cost (read, write, add, multiply, etc.)



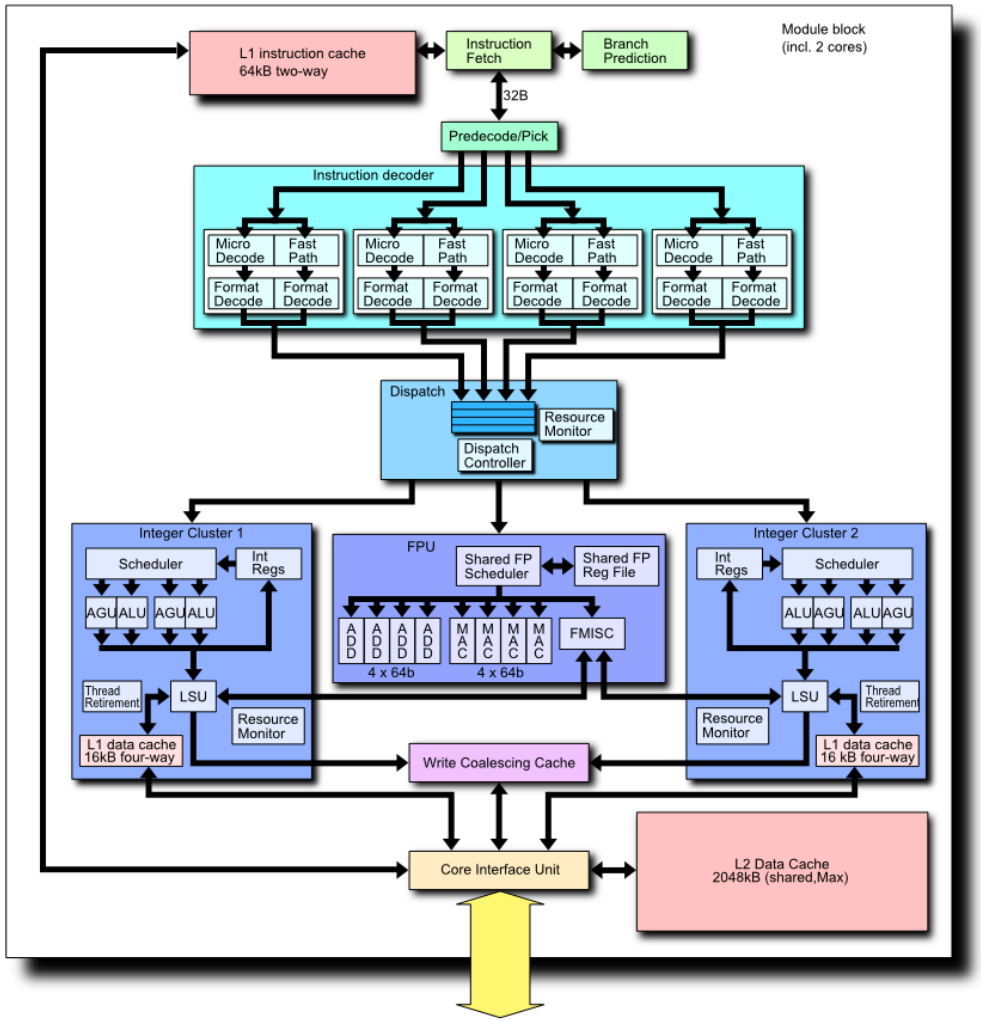
A+B=C

Read address(A) into R1
Read address(B) into R2
R3 = R1 + R2
Write R3 to address(C)

Real World Processors



Intel “Nehalem”

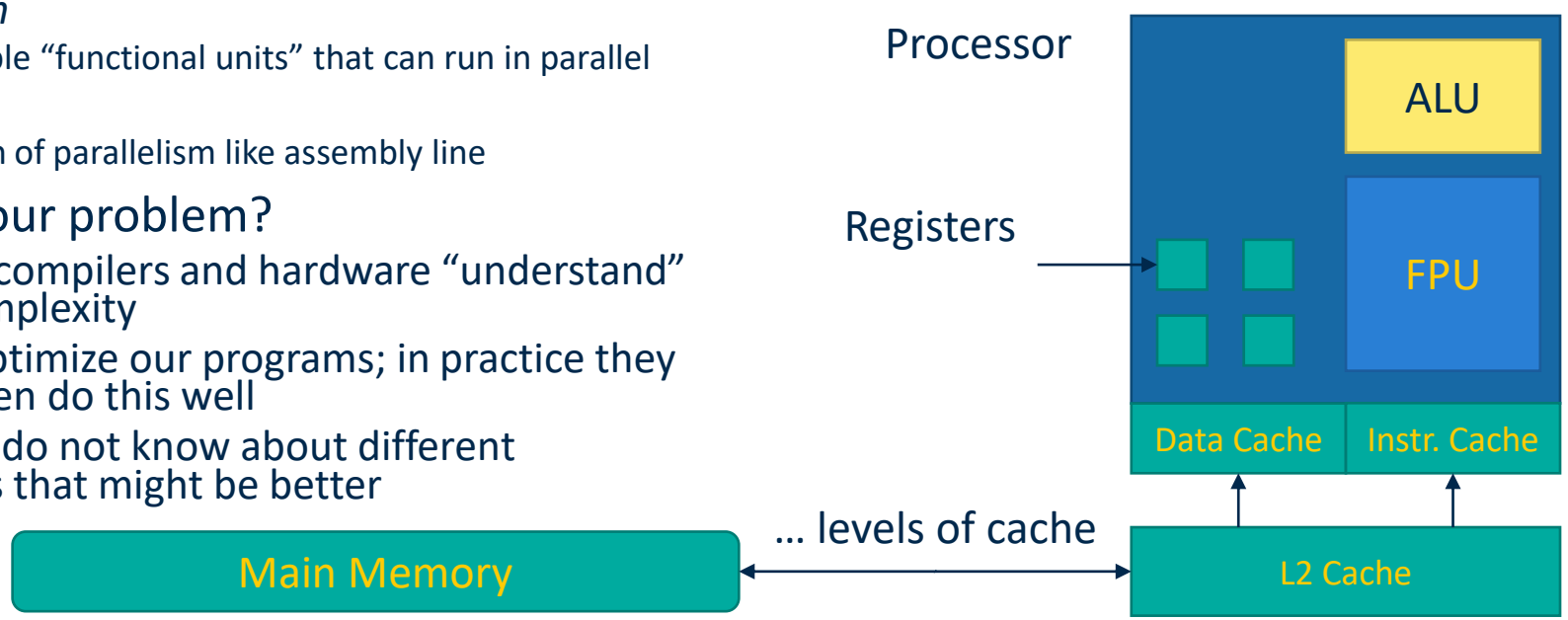


AMD “Bulldozer”

"Single" Processor Concept

- Real world processors have
  - *Registers and caches*
    - Small amounts of fast memory
    - Stores values of recently used data nearby
    - Different memory operations can have very different costs
  - *Parallelism*
    - Multiple "functional units" that can run in parallel
  - *Pipelining*
    - A form of parallelism like assembly line
- Why is this your problem?
  - In theory, compilers and hardware "understand" all this complexity
  - and can optimize our programs; in practice they do not often do this well
  - Compilers do not know about different algorithms that might be better

- We want to know the details to use processors effectively
- Don't want to know all the details
- Don't want to have an incomplete model.



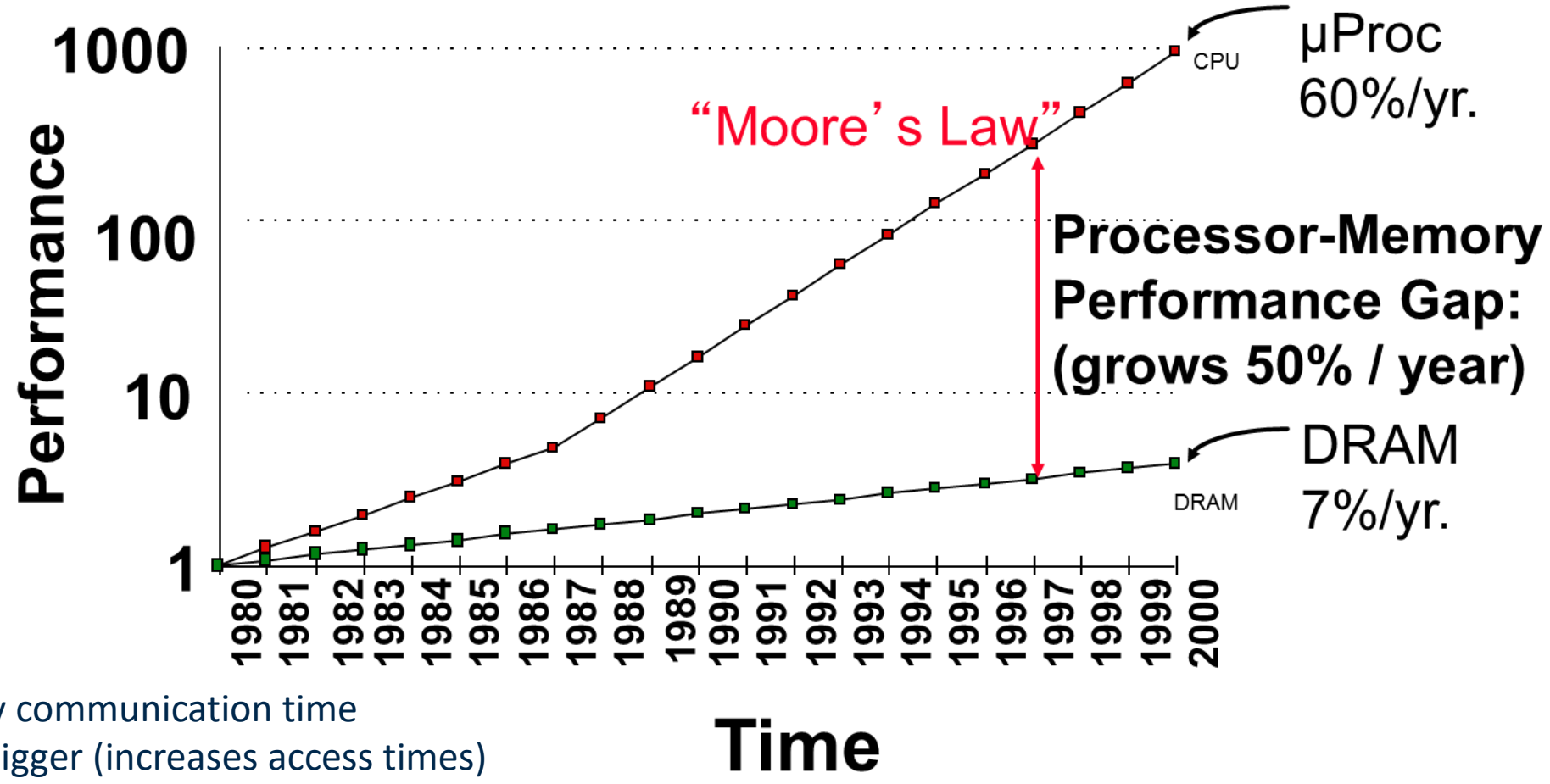
Cache Basics

- *Cache* is fast (expensive) memory which keeps copy of data in main memory;
  - Typically it is hidden from software (e.g. no standard way of programming directly)
  - Simplest example: data at memory address xxxxx1101 is stored at cache location 1101
- *Cache hit*: in-cache memory access—cheap
- *Cache miss*: non-cached memory access—expensive
  - Need to access next, slower level of cache
- *Cache line length*: # of bytes loaded together in one entry
  - Ex: If either xxxxx1100 or xxxxx1101 is loaded, both are
- *Associativity*
  - direct-mapped: only 1 address (line) in a given range in cache
    - Data stored at address xxxxx1101 stored at cache location 1101, in 16 word cache
  - *n*-way: $n \geq 2$ lines with different addresses can be stored
    - Up to $n \leq 16$ words with addresses xxxxx1101 can be stored at cache location 1101 (so cache can store 16n words)

Why have multiple levels of cache/memory?

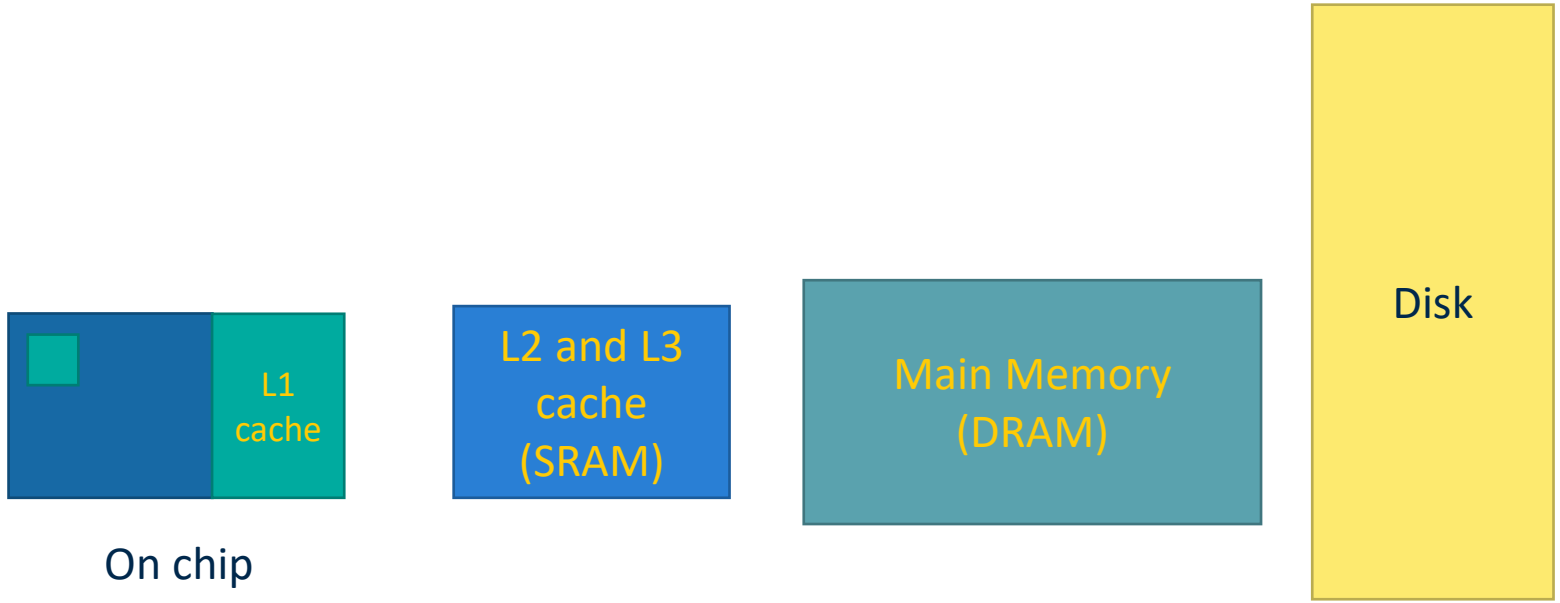
- Most programs have a high degree of locality in their memory access patterns
  - Spatial Locality: accessing data nearby previously accessed data
  - Temporal Locality: access data and reuse that data a lot
  - A memory hierarchy attempts to exploit locality to improve overall average access time.
- Cache is small and fast (speed = $$$)
  - A large cache always has delays: time to check addresses is longer
  - There are other parts to memory hierarchy (TLB, pages, swap, etc.)
- Attempts to reconcile Processor/Memory Gap

Processor-DRAM Gap (latency)



Main delay is mostly communication time
Memory is getting bigger (increases access times)

Memory Hierarchy



	Register	L1	L2	L3	DRAM	Disk	Tape
Size	< 1 KB	~1KB	1 MB	10's MB	1-100's GB	TB	PB
Speed	< 1ns	<1 ns	~1 ns	~1-10 ns	10-100 ns	10 ms	~10s

Fundamental Performance Model Concept

Execution time = time to perform arithmetic + *time to move data*

In the next lecture we'll work on developing simple performance models to understand how algorithmic choice and architecture influence performance metrics.