UC San Diego

DSC 102
Systems for Scalable Analytics

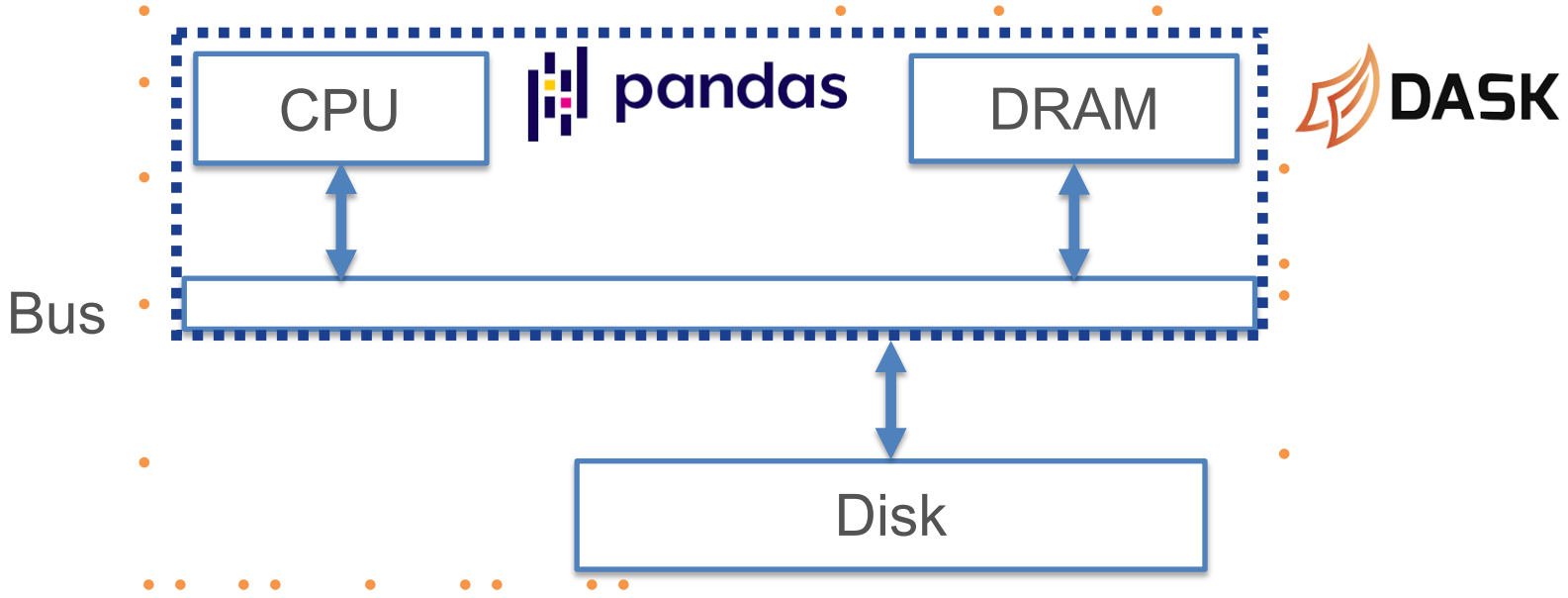
Spring 2024

Haojian Jin

Ch. 1, 2.1-2.3, 2.12, 4.1, and 5.1-5.5 of CompOrg Book

Memory Hierarchy in PA0

- Pandas DataFrame needs data to fit entirely in DRAM
- **Dask DataFrame** automatically manages Disk vs DRAM for you
  - Full data sits on Disk, brought to DRAM upon compute()
  - Dask stages out computations using Pandas



- **Tradeoff:** Dask may throw memory configuration issues. :)

I am looking for junior/novice software developers (aged 18+) for a study I'm running to evaluate a prototype to inform the trustworthiness of an open source project.

Software experience is a must, some industry experience (including internships) and some experience with open source software is a plus!

Compensation is $15 for a 40-45min Zoom call. If interested, please fill out this screener and I'll reach out to schedule a time if you qualify!
https://forms.gle/fYvg8miNXp6PgCAE8

Outline

- Basics of Computer Organization
  - Digital Representation of Data
  - Processors and Memory Hierarchy
- Basics of Operating Systems
  - Process Management: Virtualization; Concurrency
  - Filesystem and Data Files
  - Main Memory Management
- Persistent Data Storage

Basics of Processors

- **Processor:** Hardware to orchestrate and *execute instructions* to *manipulate data* as specified by a program
  - Examples: CPU, GPU, FPGA, TPU, embedded, etc.
- **ISA:** The vocabulary of commands of a processor

Program in PL

Compile/Interpret ↓

Program in Assembly Language

Assemble ↓

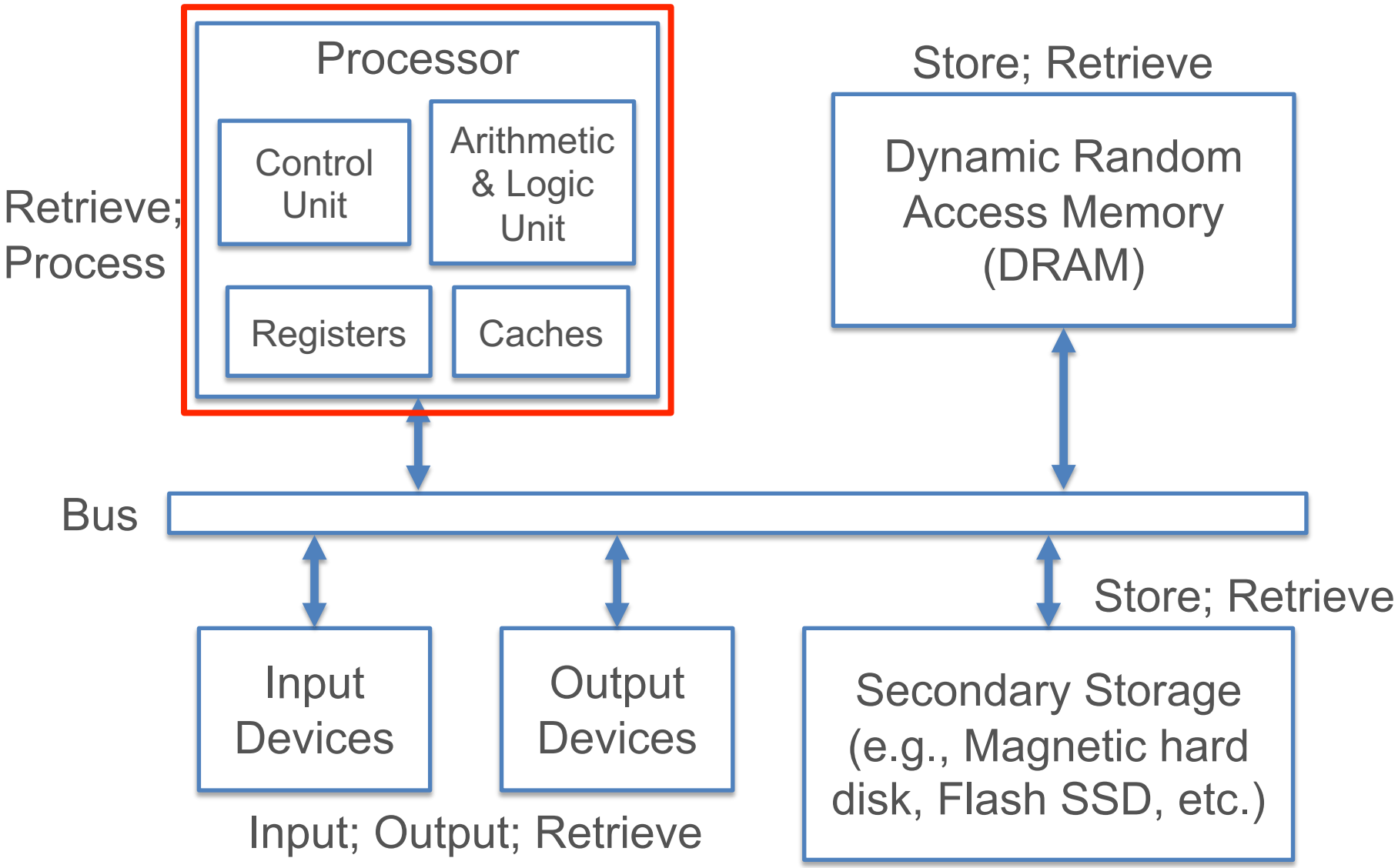
Machine code tied to ISA

↓

Run on processor

```
80483b4:    55                      push   %ebp
80483b5:    89 e5                   mov    %esp,%ebp
80483b7:    83 e4 f0                and    $0xfffffff0,%esp
80483ba:    83 ec 20                sub    $0x20,%esp
80483bd:    c7 44 24 1c 00 00 00    movl   $0x0,0x1c(%esp)
80483c4:    00
80483c5:    eb 11                   jmp    80483d8 <main+0x24>
80483c7:    c7 04 24 b0 84 04 08    movl   $0x80484b0,(%esp)
80483ce:    e8 1d ff ff ff          call   80482f0 <puts@plt>
80483d3:    83 44 24 1c 01          addl   $0x1,0x1c(%esp)
80483d8:    83 7c 24 1c 09          cmpl   $0x9,0x1c(%esp)
80483dd:    7e e8                   jle    80483c7 <main+0x13>
80483df:    b8 00 00 00 00          mov    $0x0,%eax
80483e4:    c9                      leave
80483e5:    c3                      ret
80483e6:    90                      nop
80483e7:    90                      nop
80483e8:    90                      nop
80483e9:    90                      nop
80483ea:    90                      nop
```

Abstract Computer Parts and Data



Basics of Processors

***Q:** How does a processor execute machine code?*

- Most common approach: **load-store architecture**
- **Registers:** Tiny local memory (“scratch space”) on proc. into which instructions and data are copied
- ISA specifies bit length/format of machine code commands
- ISA has several commands to manipulate register contents

Basics of Processors

***Q:** How does a processor execute machine code?*

- Types of ISA commands to manipulate register contents:
  - ***Memory access*: load** (copy bytes from a DRAM address to register); **store** (reverse); put constant
  - ***Arithmetic & logic*** on data items in registers: add/multiply/etc.; bitwise ops; compare, etc.; handled by ALU
  - ***Control flow*** (branch, call, etc.); handled by CU
- **Caches:** Small local memory to buffer instructions/data

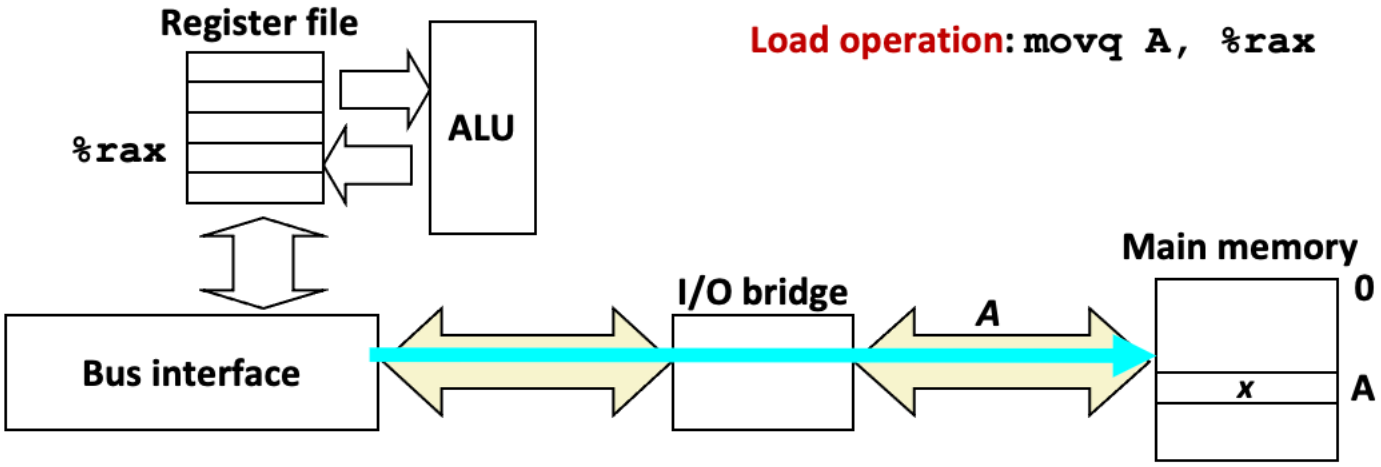
If interested in more details: https://www.youtube.com/watch?v=cNN_tTXABUA

```
80483b4:        55                      push   %ebp
80483b5:        89 e5                   mov    %esp,%ebp
80483b7:        83 e4 f0                and    $0xfffffff0,%esp
80483ba:        83 ec 20                sub    $0x20,%esp
80483bd:        c7 44 24 1c 00 00 00    movl   $0x0,0x1c(%esp)
80483c4:        00
80483c5:        eb 11                   jmp    80483d8 <main+0x24>
80483c7:        c7 04 24 b0 84 04 08    movl   $0x80484b0,(%esp)
80483ce:        e8 1d ff ff ff          call   80482f0 <puts@plt>
80483d3:        83 44 24 1c 01          addl   $0x1,0x1c(%esp)
80483d8:        83 7c 24 1c 09          cmpl   $0x9,0x1c(%esp)
80483dd:        7e e8                   jle    80483c7 <main+0x13>
80483df:        b8 00 00 00 00          mov    $0x0,%eax
80483e4:        c9                      leave
80483e5:        c3                      ret
80483e6:        90                      nop
80483e7:        90                      nop
80483e8:        90                      nop
80483e9:        90                      nop
80483ea:        90                      nop
```

Writing & Reading Memory

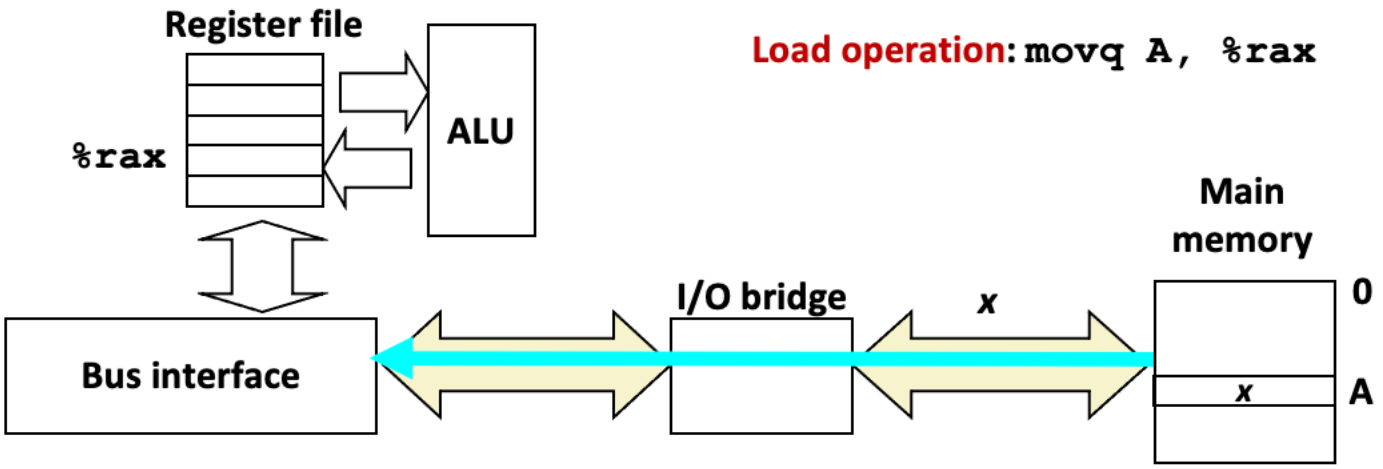
- Write
  - Transfer data from memory to CPU
    movq %rax, %rsp
  - "Store" operation
- Read
  - Transfer data from CPU to memory
    movq %rsp, %rax
  - "Load" operation

Memory Read Transaction (1)



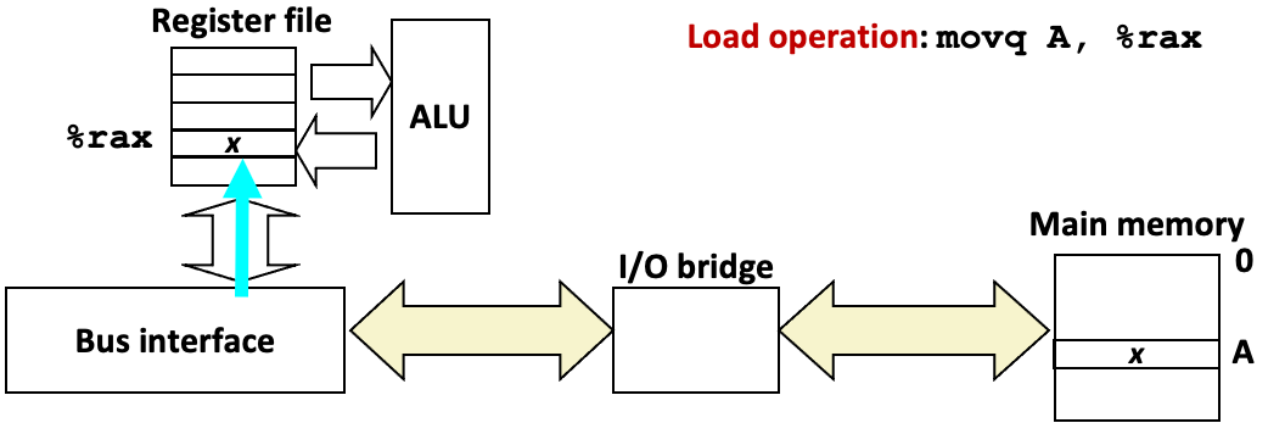
CPU places address A on the memory bus.

Memory Read Transaction (2)



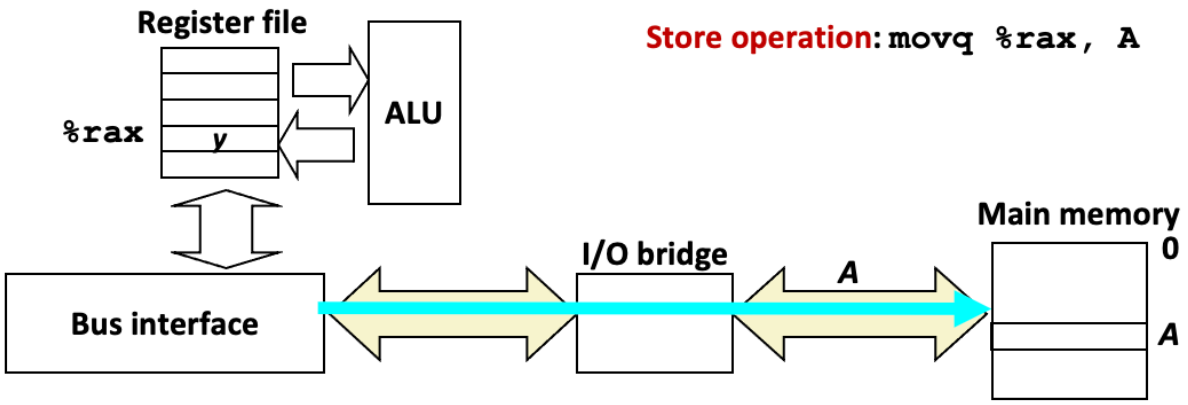
Main memory reads A from the memory bus, retrieves word x, and places it on the bus.

Memory Read Transaction (3)



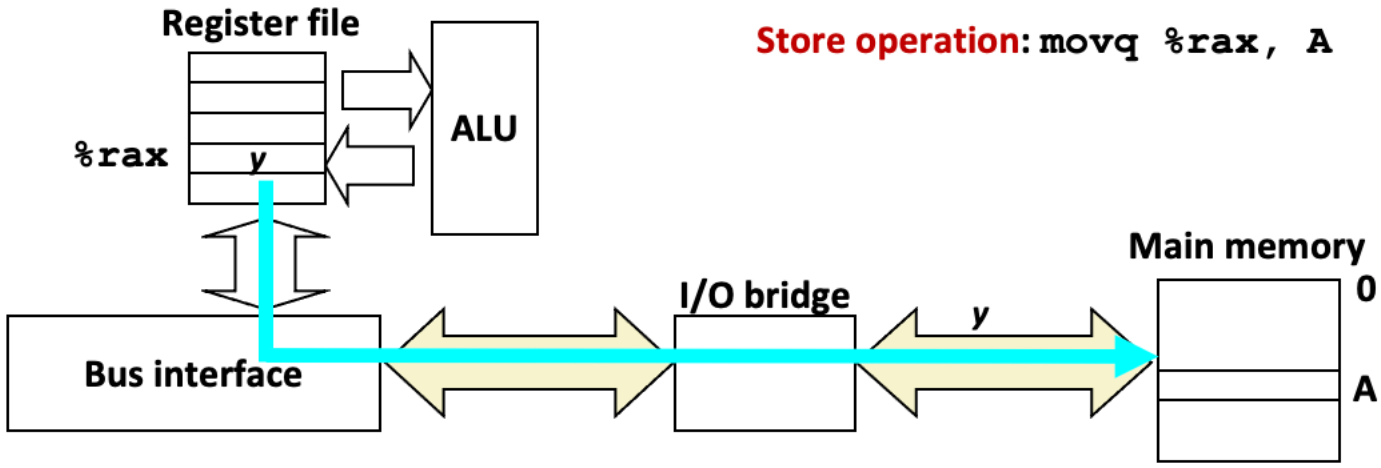
CPU reads word x from the bus and copies it into register `%rax`.

Memory Write Transaction (1)



CPU places address A on bus. Main memory reads it and waits for the corresponding data word to arrive.

Memory Write Transaction (2)



CPU places data word y on the bus.

Recall byte in data representation

- A **Byte** (B; 8 bits) is typically the basic unit of data types
  - CPU can't address anything smaller than a byte.

	Size (# of bytes)	
C Data Type	Typical 32-bit	Typical 64-bit
`char`	1	1
`short`	2	2
`int`	4	4
`long`	4	8
`float`	4	4
`double`	8	8
pointer	4	8
	"ILP32"	"LP64"

x86 v.s. x64

- x86 is the name of an ISA (80X86). 32-bit CPU.
- x64 is the name of an ISA(x86-64). 64-bit CPU
- 32-bit CPU can handle 32 bits information in each instruction.
- When we represent the memory address in bits, 32-bit CPU can support at most 2^32 bytes = 4 GB.
- You can install a 32-bit OS on a 64-bit CPU. But not a 64-bit OS on a 32-bit CPU.

Processor Performance

***Q:** How fast can a processor process a program?*

- Modern CPUs can run millions of instructions per second!
  - ISA tells #**clock cycles** per instruction
  - CPU's **clock rate** helps map that to runtime (ns)
- Alas, most programs do not keep CPU always busy:
  - Memory access commands **stall** the processor; ALU and CU are *idle* during DRAM-register transfer
  - Worse, data may not be in DRAM—wait for (disk) I/O!
  - So, actual *runtime* of program may be OOM higher than what clock rate calculation suggests

Key Principle: Optimizing access to DRAM and use of processor caches is critical for processor performance!

Processor-Optimized Math Libraries

***Q:** Would you like to write ML code in a cache-aware manner? :)*

- Matrices/tensors are ubiquitous in statistics/ML/DL programs
- Processor-optimized libraries for matrix/tensor arithmetic (*linear algebra*) studied for decades
- They *reduce memory stalls* and *increase parallelism* (more on parallelism later) automatically:
  - **Multi-core CPUs:** BLAS/LAPACK (C), Eigen (C++), la4j (Java), NumPy/SciPy (Python; can wrap BLAS)
  - **GPUs:** cuBLAS, cuSPARSE, cuDNN, cuDF, cuGraph

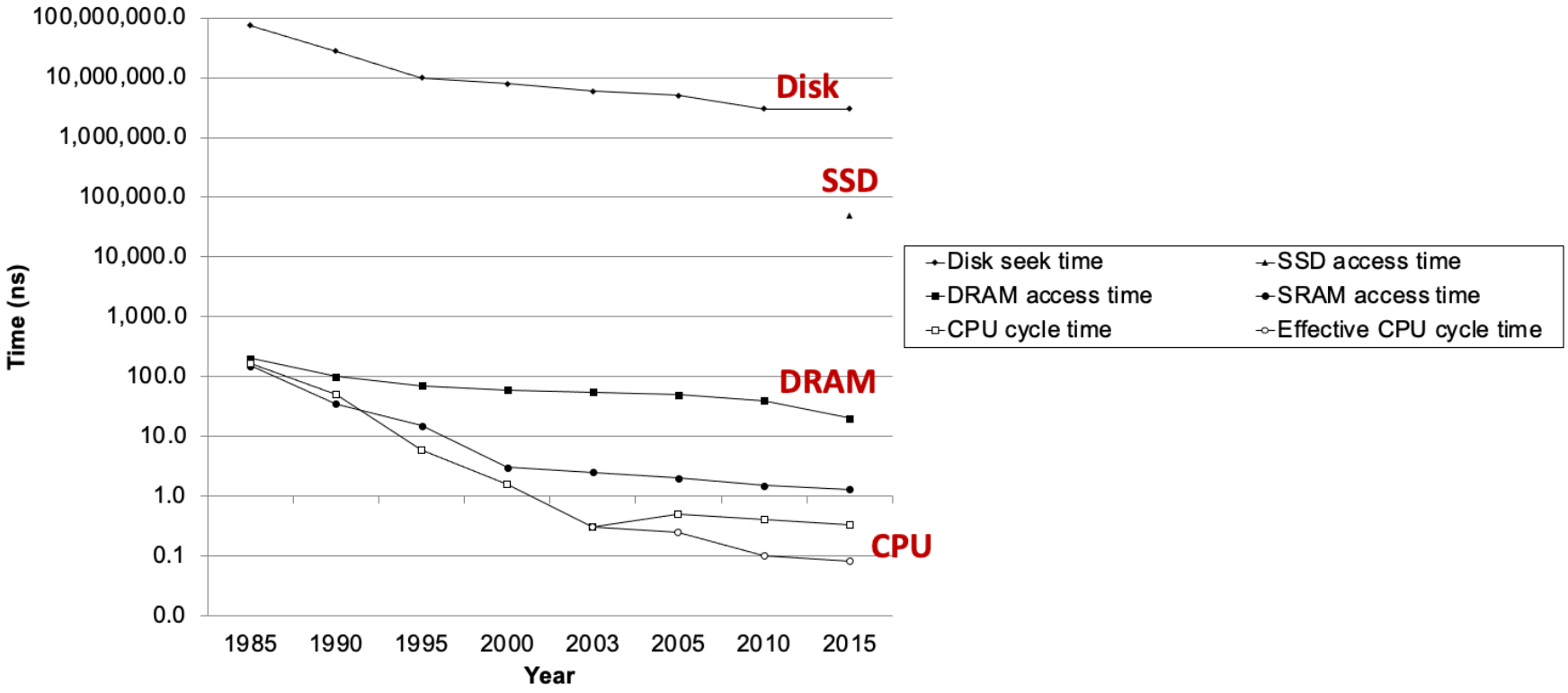
If interested, some benchmark empirical comparisons:

https://medium.com/datathings/benchmarking-blas-libraries-b57fb1c6dc7

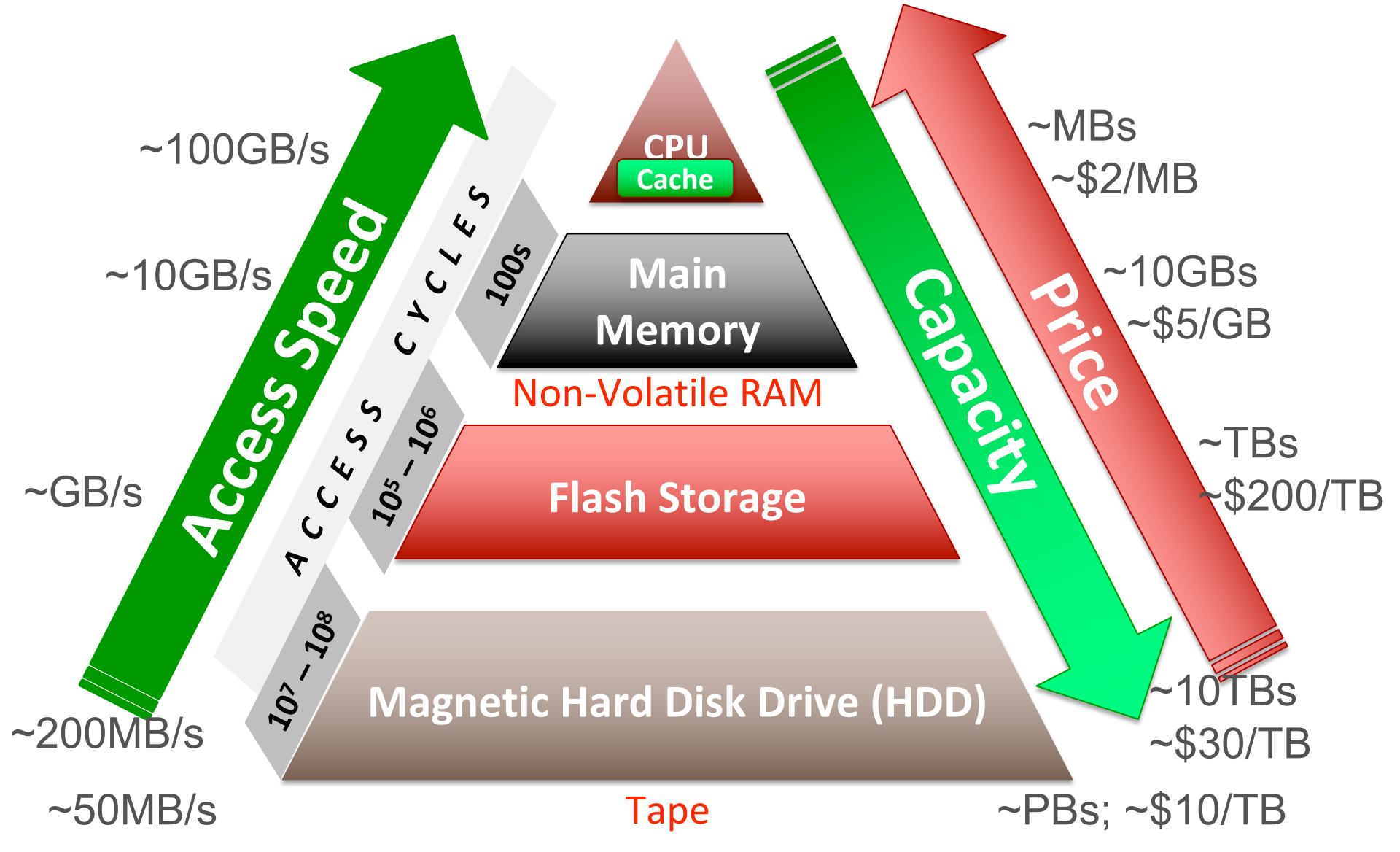
https://github.com/andre-weiteveiz/blas-benchmarks

The CPU-Memory Gap

The gap *widens* between DRAM, disk, and CPU speeds.



Memory/Storage Hierarchy



Memory Hierarchy in Action

Q: *What does this program do when run with 'python'? (Assume tmp.csv is in current working directory)*

tmp.py

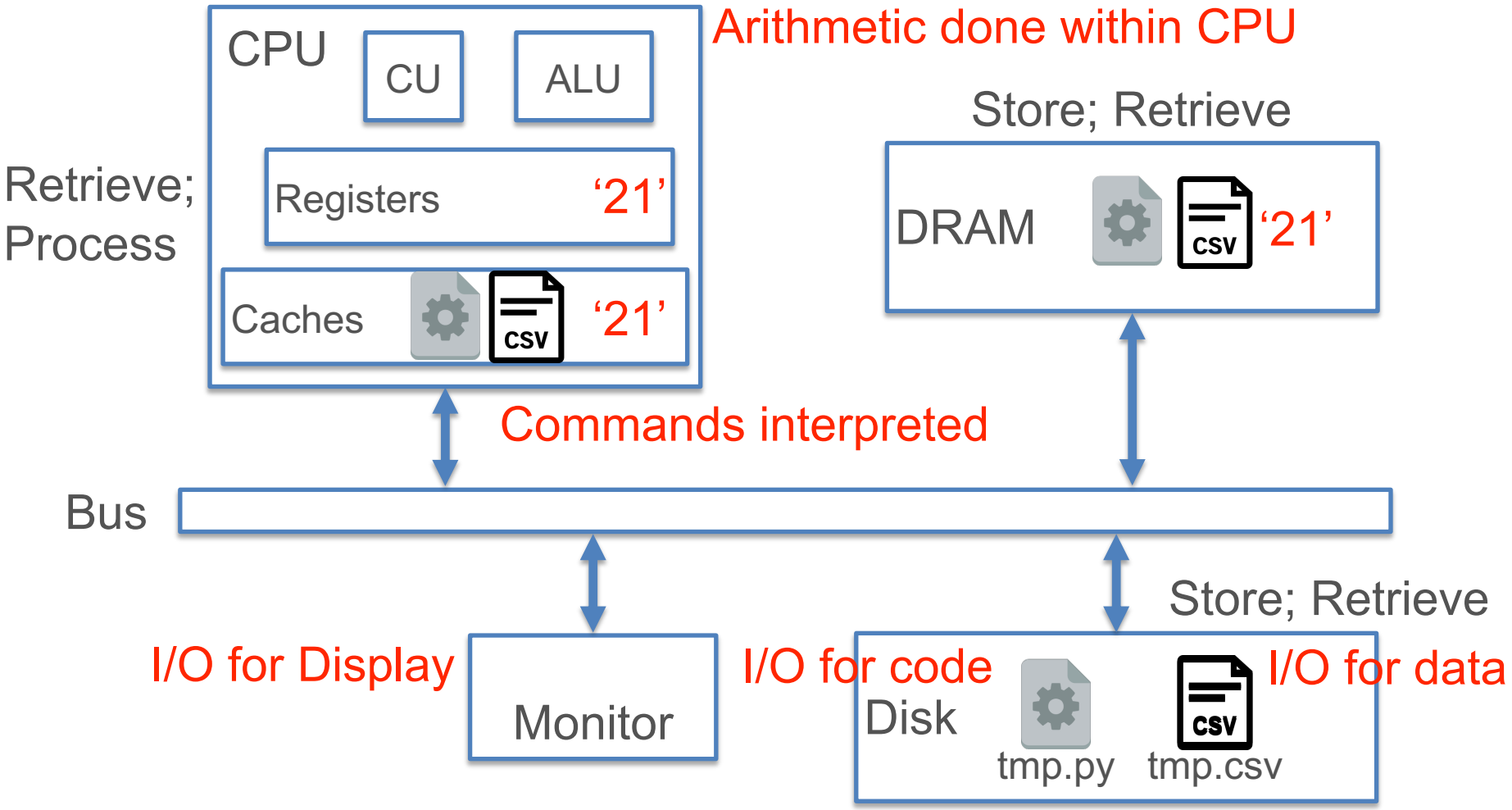
```
import pandas as p
m = p.read_csv('tmp.csv',header=None)
s = m.sum().sum()
print(s)
```

tmp.csv

```
1,2,3
4,5,6
```

Memory Hierarchy in Action

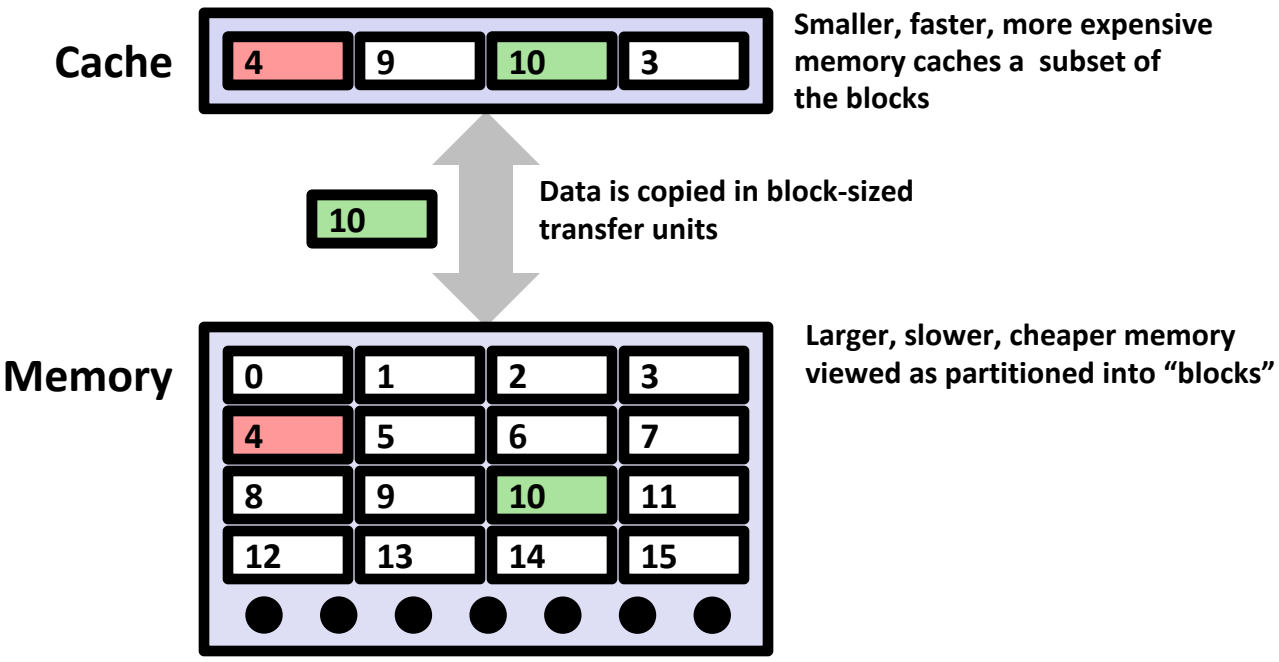
Rough sequence of events when program is executed



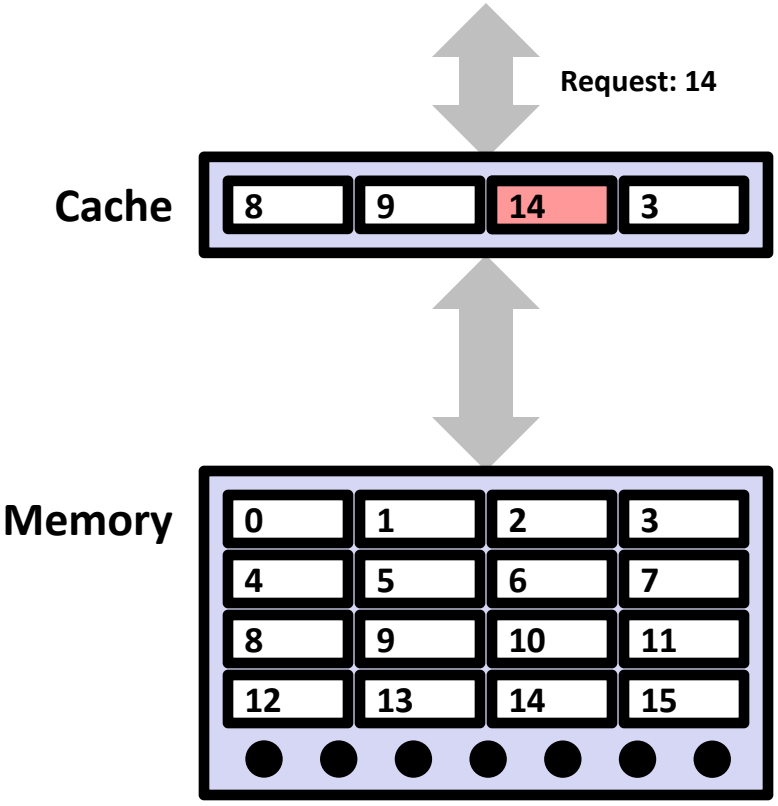
Concepts of Memory Management

- **Caching:** Buffering a copy of bytes (instructions and/or data) from a lower level at a higher level to exploit locality
- **Prefetching:** Preemptively retrieving bytes (typically data) from addresses not explicitly asked yet by program
- **Spill/Miss/Fault:** Data needed for program is not yet available at a higher level; need to get it from lower level
  - **Register Spill** (register to cache); **Cache Miss** (cache to main memory); **"Page" Fault** (main memory to disk)
- **Hit:** Data needed is already available at higher level
- **Cache Replacement Policy:** When new data needs to be loaded to higher level, which old data to evict to make room? Many policies exist with different properties

General Cache Concepts



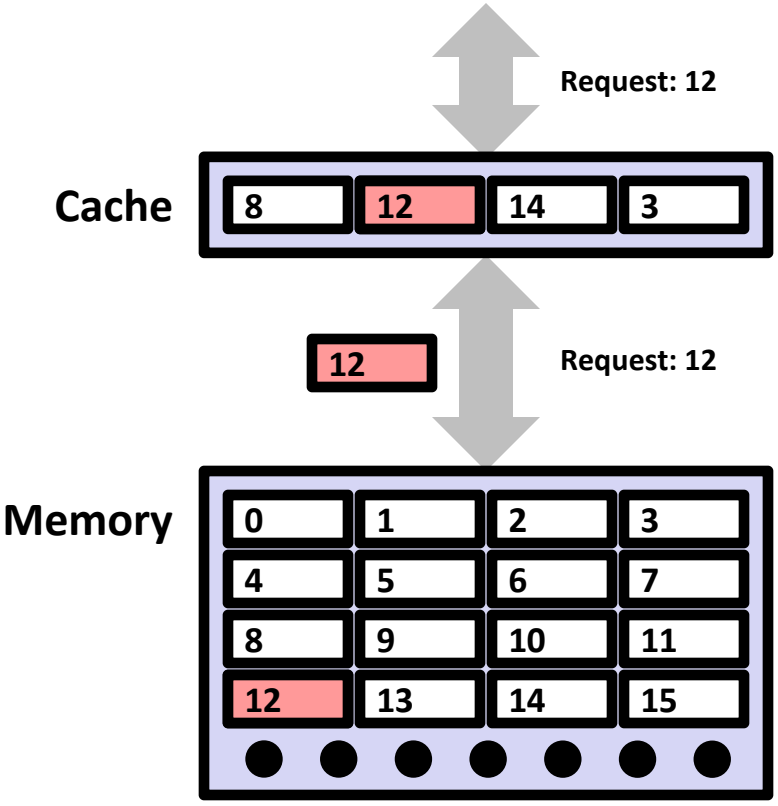
General Cache Concepts: Hit



Data in block b is needed

Block b is in cache:
Hit!

General Cache Concepts: Miss



Data in block b is needed

Block b is not in cache:
Miss!

Block b is fetched from memory

Block b is stored in cache

- Placement policy: determines where b goes
- Replacement policy: determines which block gets evicted (victim)

Memory/Storage Hierarchy

- Typical desktop computer today ($700):
  - 1 TB magnetic hard disk (SATA HDD); 32 GB DRAM
  - 3.4 GHz CPU; 4 cores; 8MB cache
- High-end enterprise rack server for RDBMSs ($8,000):
  - 12 TB Persistent memory; 6 TB DRAM
  - 3.8 GHz CPU; 28-core per proc.; 38MB cache
- Renting on Amazon Web Services (AWS):
  - EC2 m5.large: 2-core, 8GiB: $0.096 / hour
  - EC2 m5.24xlarge: 96-core, 384 GiB, $4.608 per hour
  - EBS general SSD: $0.10 per GB-month
  - S3 store / read: $0.023 / 0.013-0.023 per GB-month

copyij v.s copyji: copy a 2048 X 2048 integer array

```
void copyij(long int src[2048][2048], long int dst[2048][2048])
{
  long int i,j;
  for (i = 0; i < 2048; i++)
    for (j = 0; j < 2048; j++)
      dst[i][j] = src[i][j];
}

void copyji(long int src[2048][2048], long int dst[2048][2048])
{
  long int i,j;
  for (j = 0; j < 2048; j++)
    for (i = 0; i < 2048; i++)
      dst[i][j] = src[i][j];
}
```

4.3 milliseconds

81.8 milliseconds

Key Principle: Locality of Reference

Carefully handling/optimizing access to DRAM and use of processor caches is critical for processor performance!

Due to OOM access latency gaps across memory hierarchy, optimizing access to lower levels and careful use of higher levels is critical for overall system performance!

- **Locality of Reference:** Many programs tends to access DRAM locations in a somewhat *predictable* manner
  - **Spatial:** Nearby locations will be accessed soon
  - **Temporal:** Same locations accessed again soon
- Locality can be exploited to reduce runtimes using **caching** and/or **prefetching** across all levels in the hierarchy

Locality of Reference for Data

- **Data Layout:**
  - The *order* in which data items of a complex data structure/ADT are laid out in memory/disk
- **Data Access Pattern** (of a program on a data object):
  - The *order* in which a program has to access items of a complex data structure/ADT in memory
- **Hardware Efficiency** (of a program):
  - How close *actual execution runtime* is to best possible runtime given the proc. clock rate and ISA
  - Improved with careful data layout of all data objects used by a program based on its data access patterns
  - **Key Principle:** Raise cache hits; reduce memory stalls!

Locality of Reference in Data Science

- Common example: matrix multiplication (>1m cells each)
- Suppose data layout in DRAM is in ***row-major*** order

$$C_{n\times m} = A_{n\times p}\ B_{p\times m}$$

DRAM A[1;] A[2;] A[3;] … B[1;] B[2;] …

Caches A[1;]. B[1;]

```
for i = 1 to n
  for j = 1 to m
    for k = 1 to p
      C[i][j] += A[i][k] * B[k][j]
```

- Not too hardware-efficient
- Prefetching+caching means full row based on innermost loop is brought to proc. cache
- A[i][.] Hits but B[k][j] Misses
- So each * op is a stall! :(

Locality of Reference in Data Science

- Common example: matrix multiplication (>1m cells each)
- Suppose data layout in DRAM is in ***row-major*** order

$$C_{n\times m} = A_{n\times p}\ B_{p\times m}$$

DRAM A[1;] A[2;] A[3;] … B[1;] B[2;] …

Caches A[1;]. B[1;]. C[1;]

```
for i = 1 to n
  for k = 1 to p
    for j = 1 to m
      C[i][j] += A[i][k] * B[k][j]
```

- *Logically equivalent* computation but different order of ops!
- C[i][.] and B[k][.] Hits
- A[i][k] also Hit (unaffected by j)
- Orders of magnitude fewer stalls!
- Lot more hardware-efficient

Locality of Reference in Data Science

- Common example: matrix multiplication (>1m cells each)
- Suppose data layout in DRAM is in ***row-major*** order

$$C_{n\times m} = A_{n\times p}\ B_{p\times m}$$

```
for i = 1 to n
  for j = 1 to m
    for k = 1 to p
      C[i][j] += A[i][k] * B[k][j]
```

Rewrite ⇩

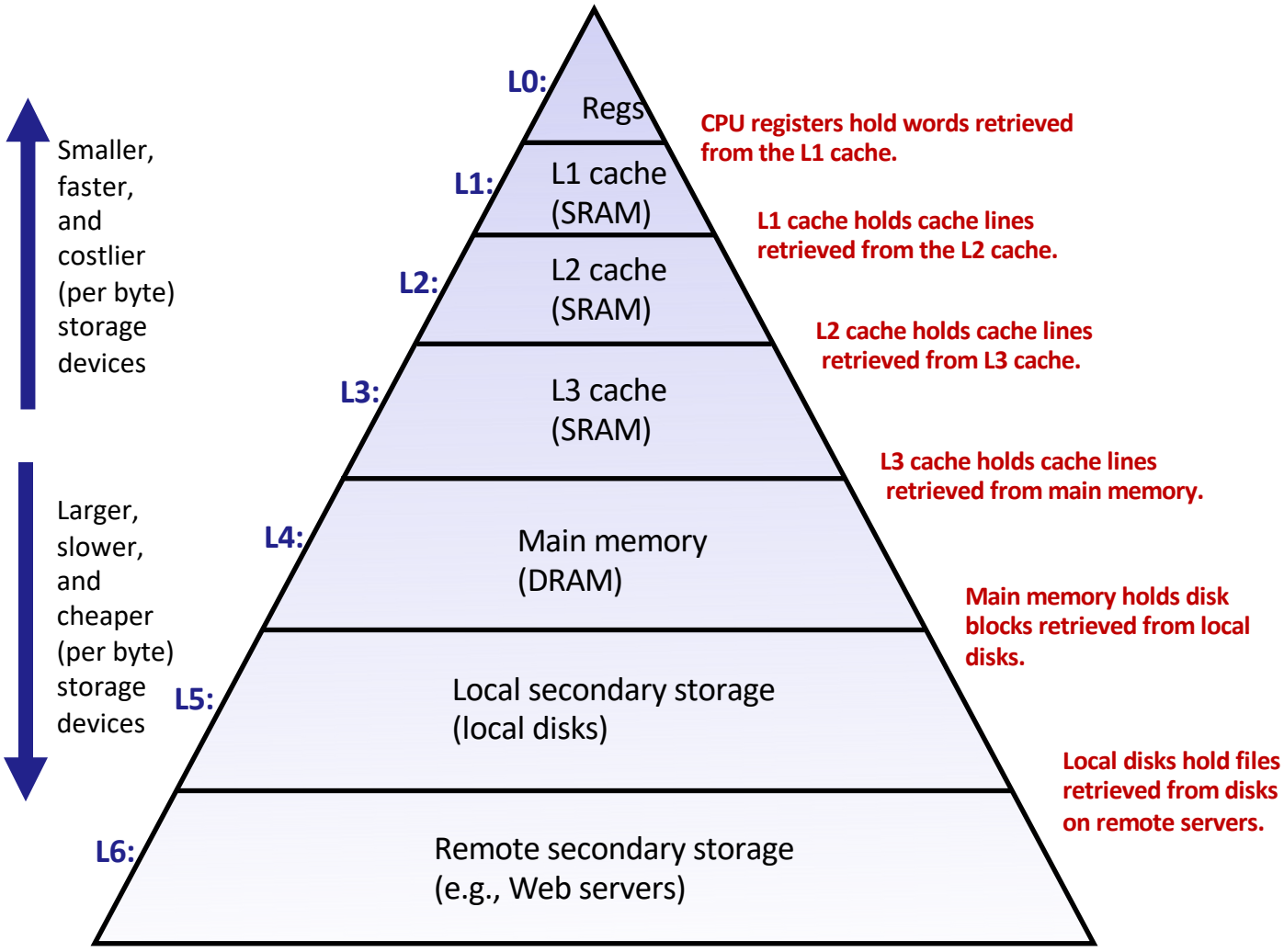
```
for i = 1 to n
  for k = 1 to p
    for j = 1 to m
      C[i][j] += A[i][k] * B[k][j]
```

- Although the math is the same and gives the same results ("logically equivalent"), the physical properties of program execution are vastly different
- Commonly used in *compiler optimization* and later on, also in *query optimization*

Recap: Memory Hierarchies

- Fast storage technologies cost more per byte, have less capacity, and require more power (heat!).
- The gap between CPU and main memory speed is widening.
- They suggest an approach for organizing memory and storage systems known as a memory hierarchy.

Example Memory Hierarchy



Why does it work?

- For each k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1.
- Why do memory hierarchies work?
  - Because of locality: programs tend to access the data at level k more often than they access the data at level k+1.
  - Thus, the storage at level k+1 can be slower, and thus larger and cheaper per bit.
- Big Idea (Ideal): The memory hierarchy creates a large pool
  of storage that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.

Review Questions

- What is an ISA?
- What are the 3 main kinds of commands in an ISA?
- Why do CPUs have both registers and caches?
- Why is it typically impossible for data processing programs to achieve 100% processor utilization?
- Which of these memory hierarchy layers is the most expensive: CPU cache, DRAM, flash disks, or magnetic hard disks?
- Which of the above layers is the slowest for data access?
- Which library helps ML users avoid need for writing GPU cache-aware computations?

Outline

- Basics of Computer Organization
  - Digital Representation of Data
  - Processors and Memory Hierarchy
- Basics of Operating Systems (OS)
  - Process Management: Virtualization; Concurrency
  - Filesystem and Data Files
  - Main Memory Management
- Persistent Data Storage