Caching

DS 5110/CS 5501: Big Data Systems

Spring 2024

Lecture 2d

Yue Cheng

University of Virginia

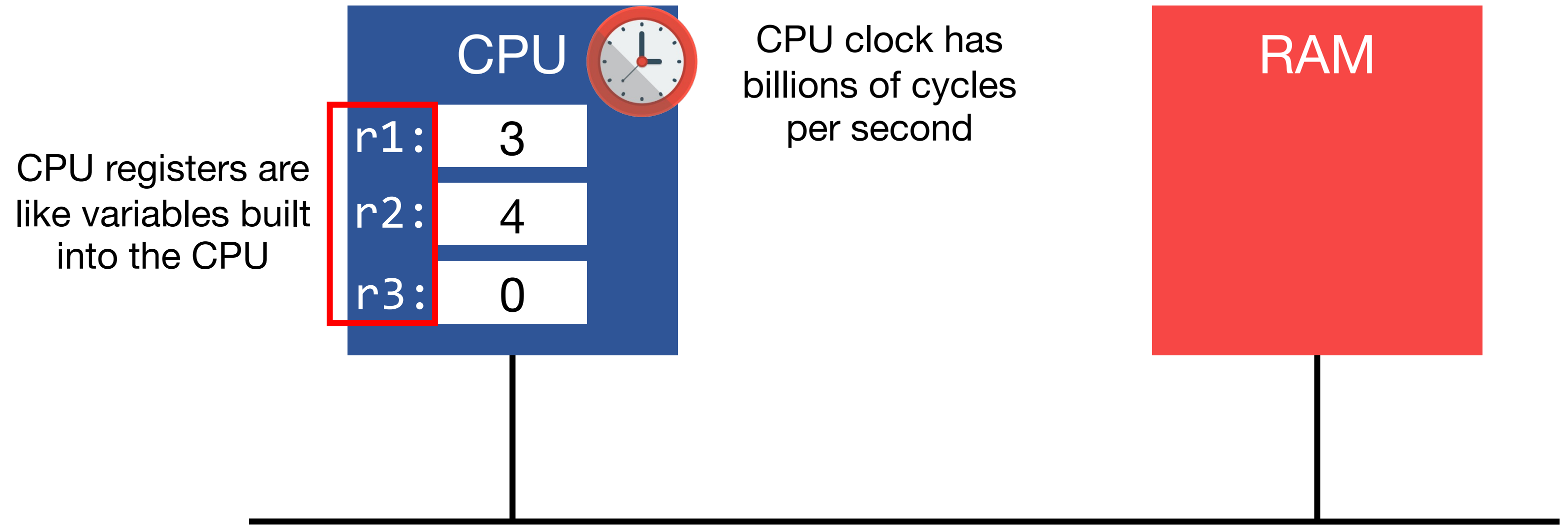
Learning objectives

- Describe the cache hierarchy
- Understand spatial locality and temporal locality
- Trace through access patterns with FIFO and LRU caching policies
    - Calculate cache performance metrics

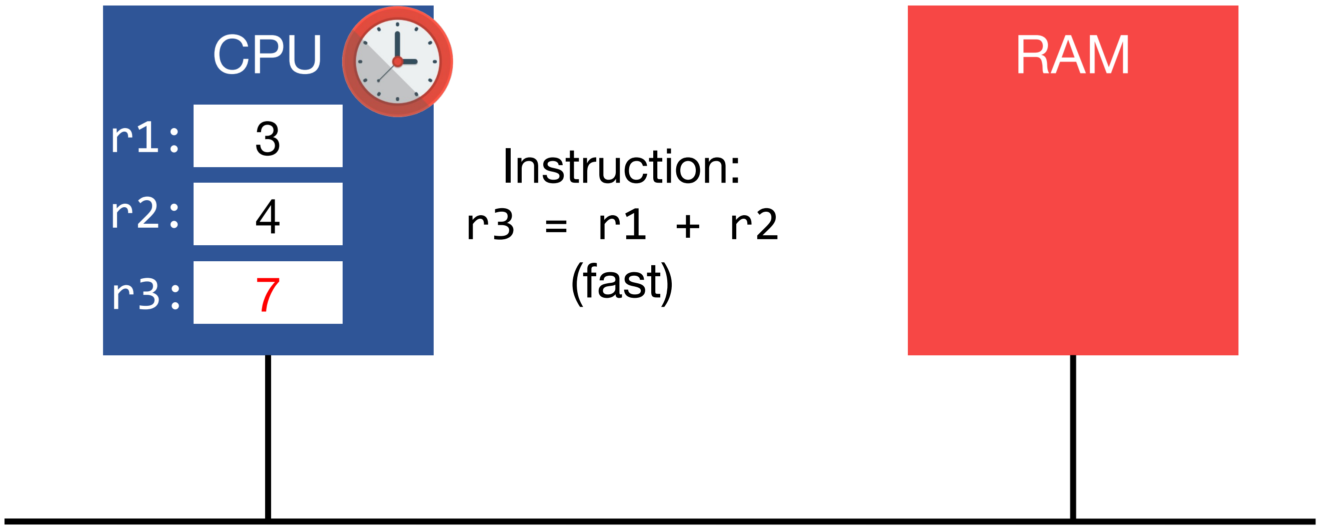
Outline

- Challenge: latency
- Cache hierarchy
  - CPU, RAM, SSD, Disk, Network
  - Tradeoffs
- Data access patterns, data locality, data access granularity
  - Spatial locality
  - Temporal locality
  - Cache lines and locality optimization
- What data should be cached?
  - Eviction policies: FIFO, LRU

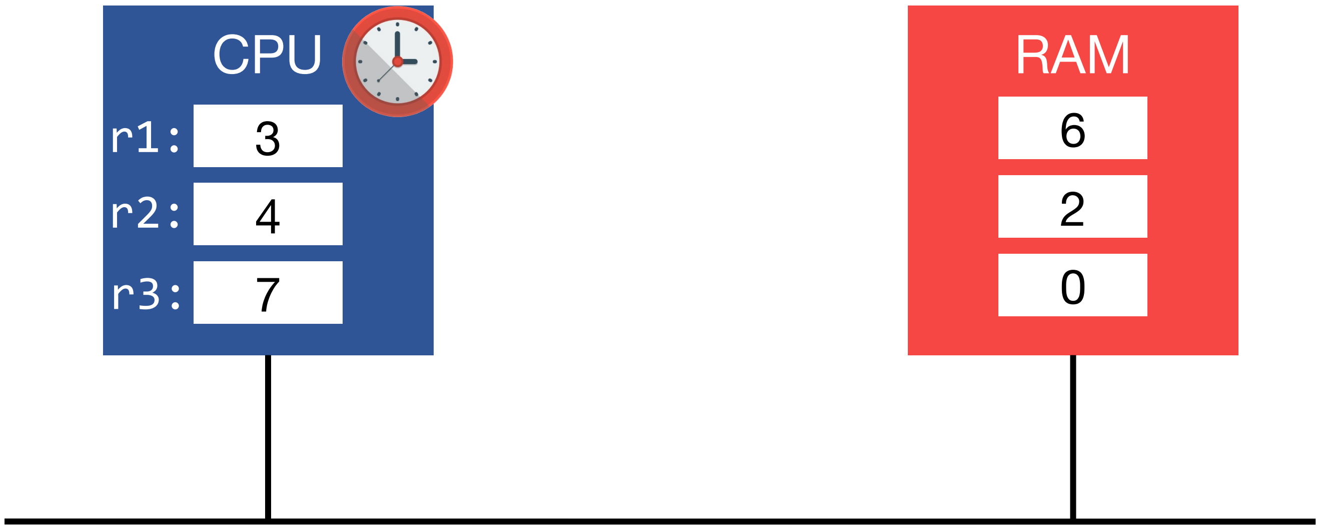
Interaction between CPU and RAM



Interaction between CPU and RAM

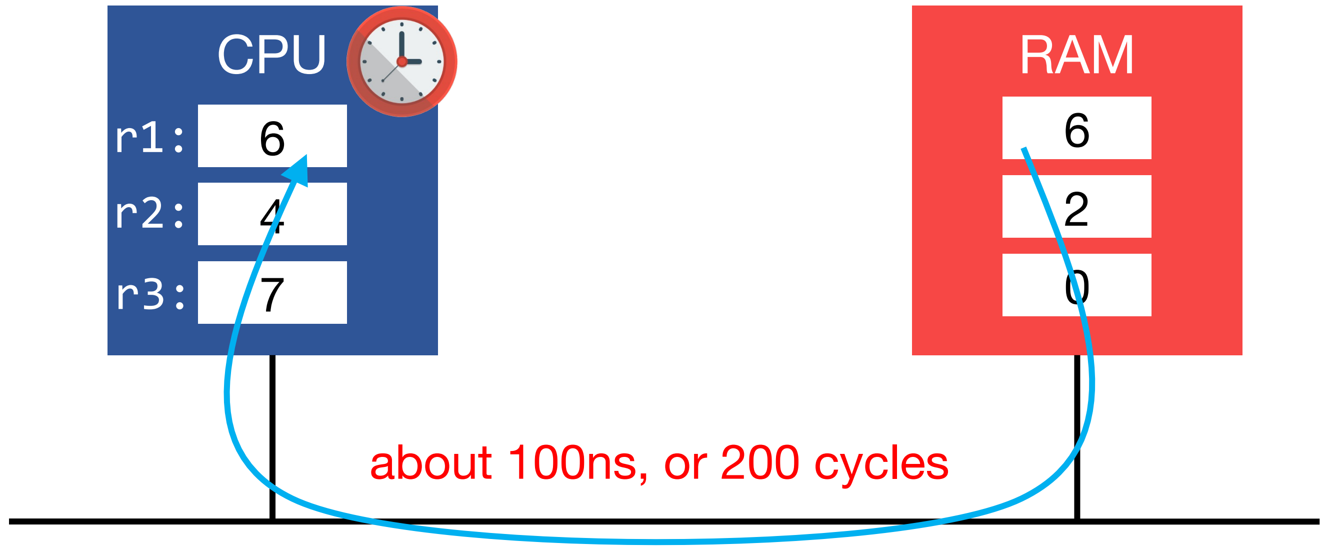


Load and store



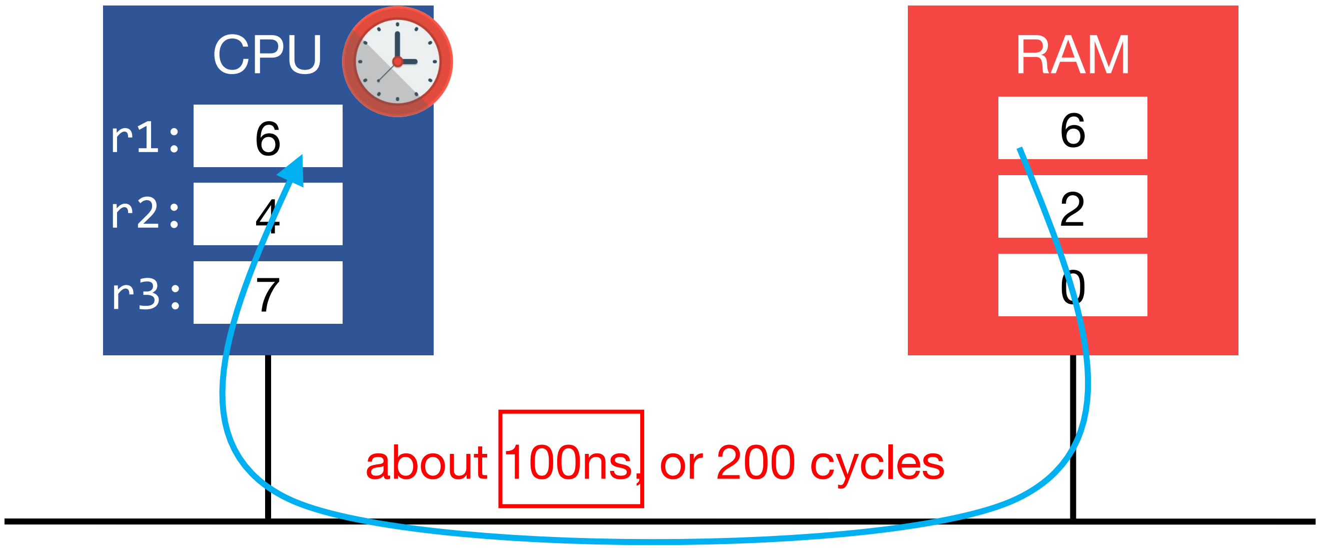
Challenge: If we want to add some numbers stored in RAM, we need to **load** before adding and **store** after

Latency to load from RAM



Very slow, but not long enough to switch to a different thread…

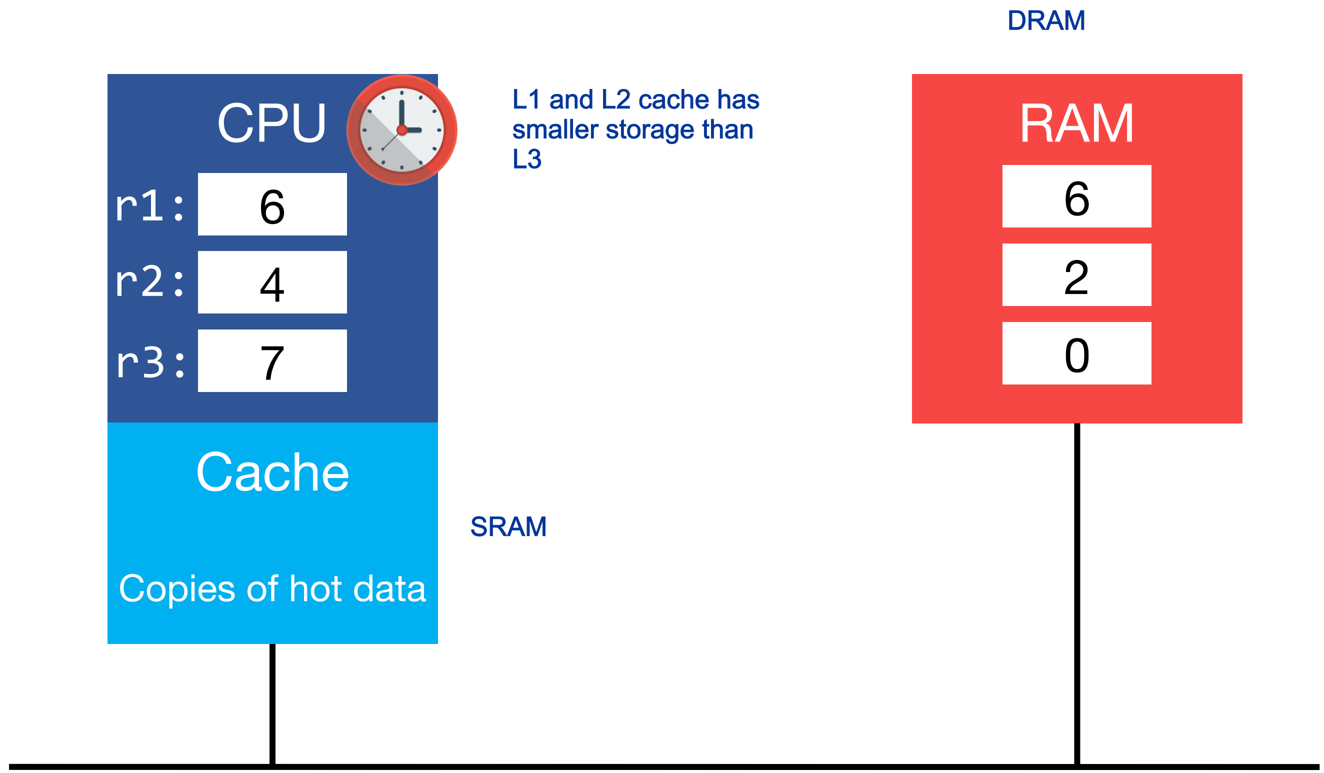
Latency



“How much time” is a **latency** measure.

Throughput (bytes/second) depends on how many loads we can do simultaneously.

CPU Cache



Idea: CPUs can have a small but very fast memory built in for data that is frequently accessed

Latency measurements

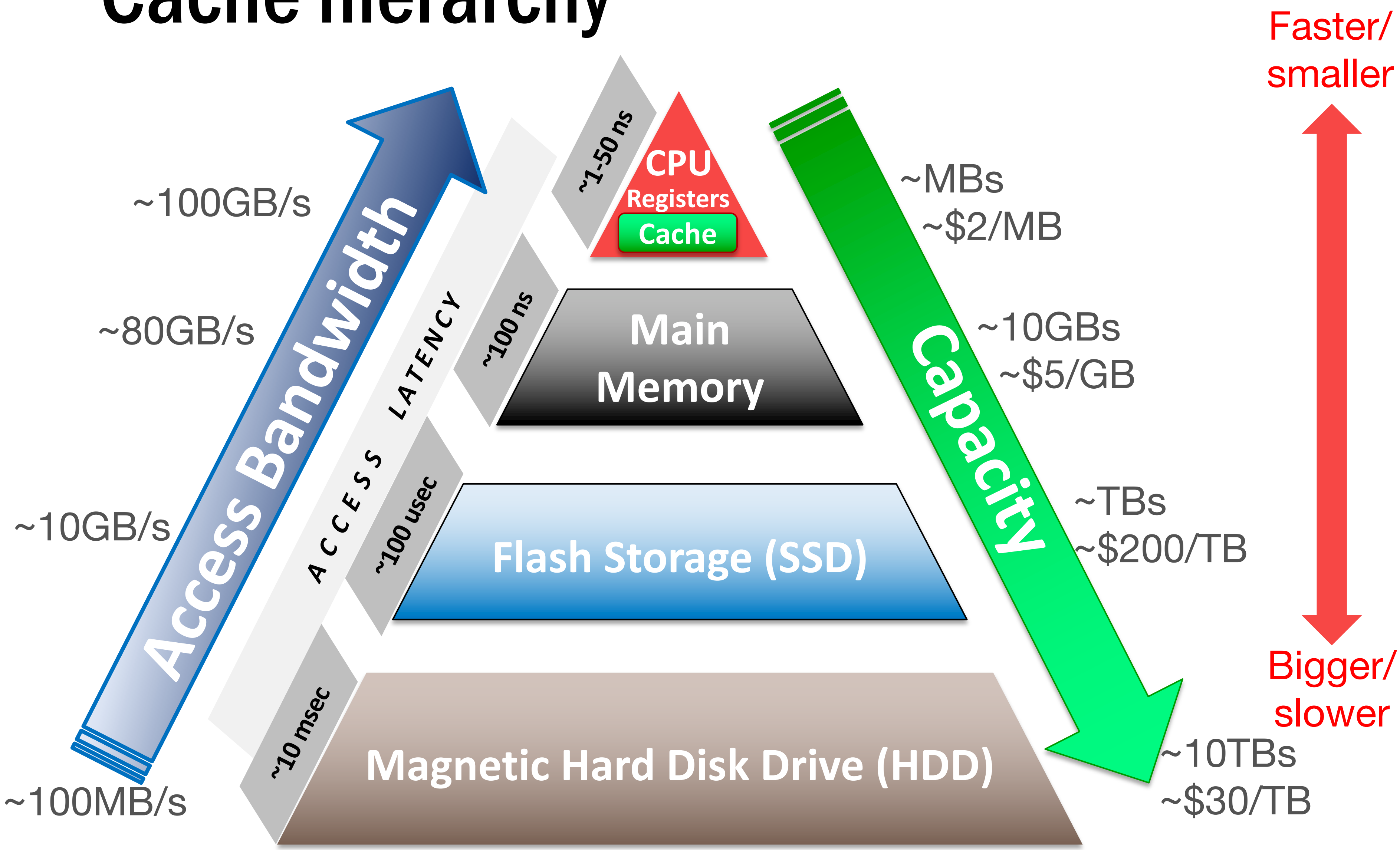
- Latency metrics
  - Average latency
  - Median latency
  - "Tail" latency (99th percentile, 99.9th percentile, etc.)

    looking at the extreme occurances of latency

- Which metrics do we expect **caching** to help with the most?

  average and sometimes median! Caching is a hit or miss so it can optimize the average latency (or sometimes median)

Cache hierarchy



*UCSD DSC 102: Systems for scalable analysis. Arun Kumar

Resource tradeoffs

- File system caches file data in RAM
  - Uses memory
  - Avoids storage reads

- Browser caches recently visited pages as disk files
  - Uses local storage space
  - Avoids network transfers

- Python dictionary caches return values in a `dict` (`key=args, val=return`)
  - Uses memory space
  - Avoids repeated compute

```
cache = {}
def f(x):
    if not x in cache:
        cache[x] = g(x)
    return cache[x]
```

Workload characteristics

Application A

```
sum = 0
for i in range(0,1024):
  sum += a[i]
```

Workload characteristics

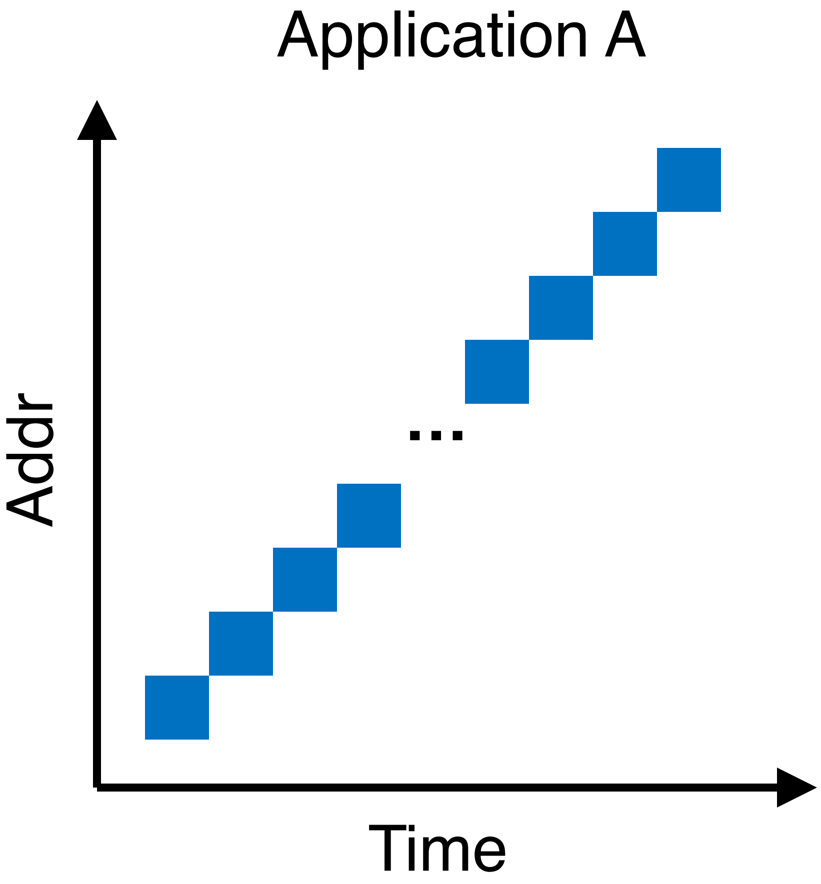
Application A

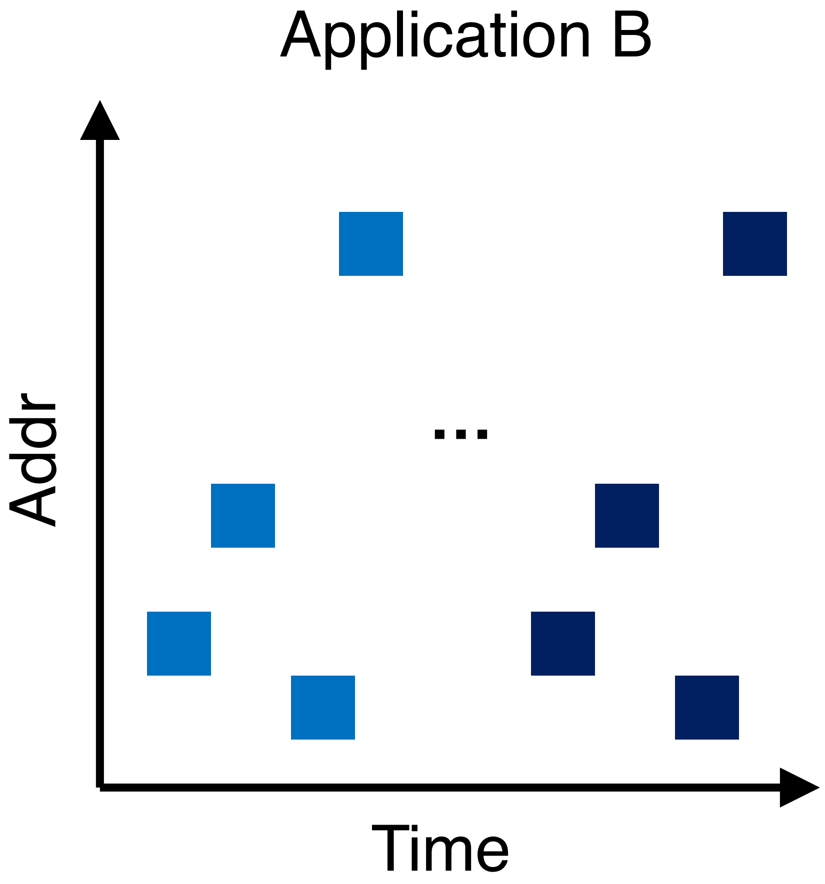
```
sum = 0
for i in range(0,1024):
  sum += a[i]
```

Application B

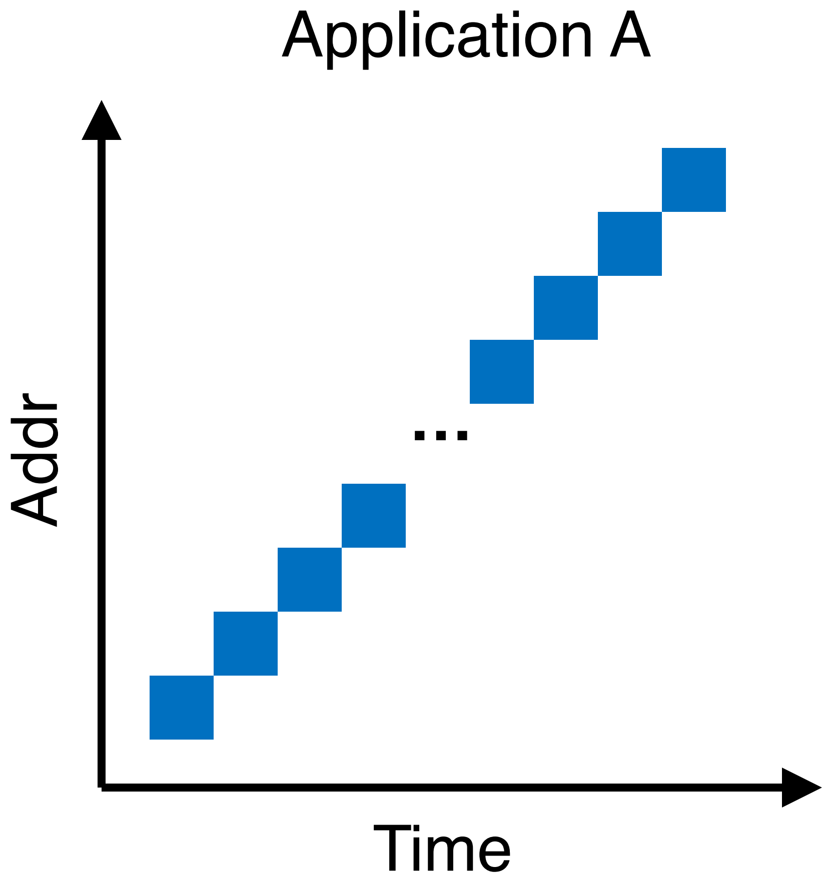
```
import random

sum = 0
random.seed(1234);
for i in range(0,512):
  sum += a[random.randint(0,1023)]

random.seed(1234) # same seed
for i in range(0,512):
  sum += a[random.randint(0,1023)]
```

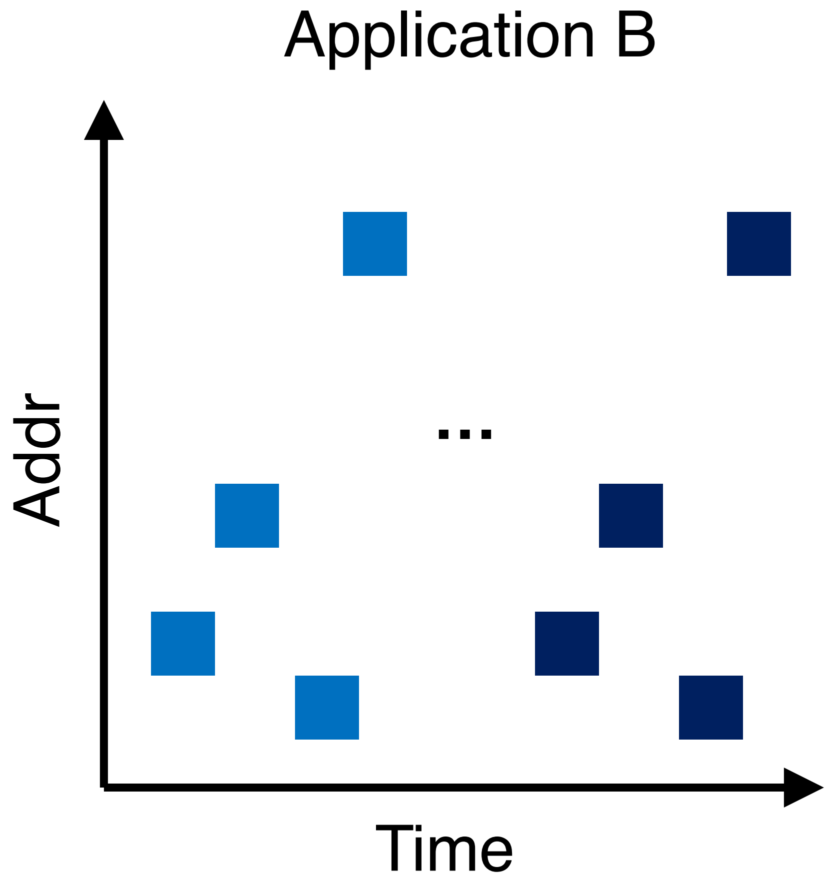
Access patterns





Access patterns





Spatial Locality **Temporal Locality**

Locality of data accesses

- Spatial locality:
  - Future access will be to nearby addresses
- Temporal locality:
  - Future access will be repeated to the same data

Locality of data accesses

- Spatial locality:
  - Future access will be to nearby addresses

- Temporal locality:
  - Future access will be repeated to the same data

- Q: What is the **implication of data locality** to data systems applications?

Locality optimization in Data Science

- Consider a matrix named data with `16*16` elements
- Each row is of size 16 floats and prefetching+caching means 1/2 row of accessed data item is brought to CPU cache at a time

Locality optimization in Data Science

- Consider a matrix named data with `16*16` elements
- Each row is of size 16 floats and prefetching+caching means 1/2 row of accessed data item is brought to CPU cache at a time
- Program 1

```
for i in range(len(data[0]):
    for row in data:
        sum += row[i]
```

16 x 16 = 256 CPU cache misses

Not too hardware-efficient (not able to exploit prefetching+caching)

Locality optimization in Data Science

- Consider a matrix named data with `16*16` elements
- Each row is of size 16 floats and prefetching+caching means 1/2 row of accessed data item is brought to CPU cache at a time
- Program 1

```
for i in range(len(data[0])):
    for row in data:
        sum += row[i]
```

16 x 16 = 256 CPU cache misses

Not too hardware-efficient (not able to exploit prefetching+caching)

- Program 2

```
for row in data:
  for element in row:
    sum += element
```

Only 16*2 CPU cache misses

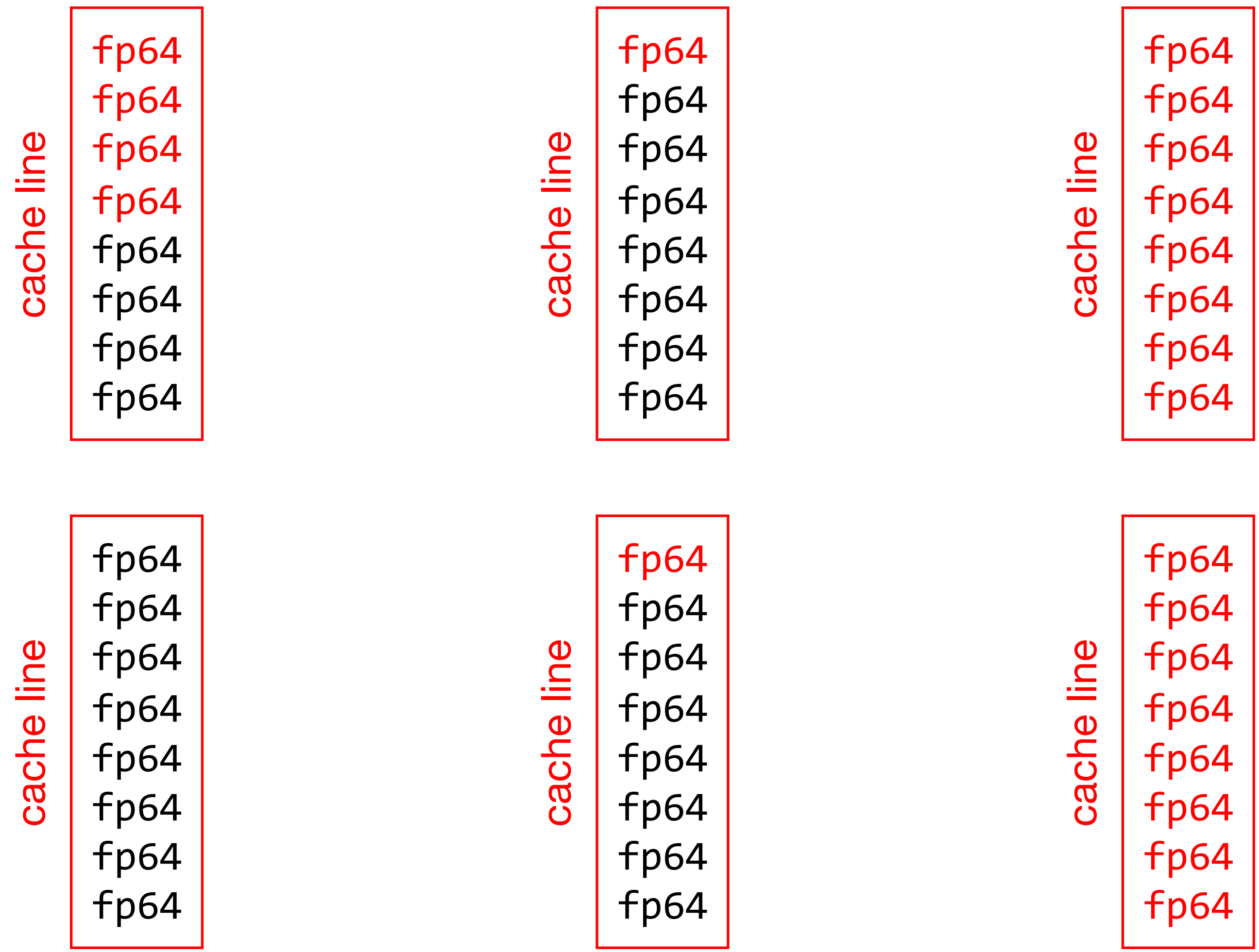
- Each time ½ row of `data[i]` is prefetched to cache so subsequent accesses are hits!

Peeking behind the scene…

- Data access granularity
  - If a process reads one byte and misses, how much data should the CPU bring into the CPU cache?
  - Tradeoff:
    - **Too little?** Will have many more misses if we read nearby bytes soon (recall spatial locality)
    - **Too much?** Wasteful to load data to cache that might never be accessed

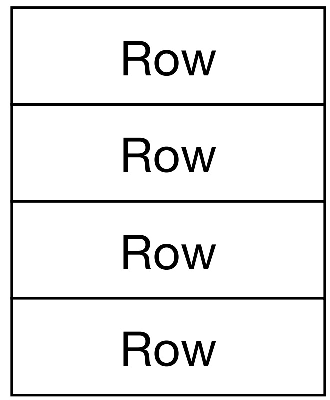
- CPU caches data in units called **cache lines**
  - Typically, 64 bytes for modern CPUs (8 float64 numbers)

Cache lines and misses



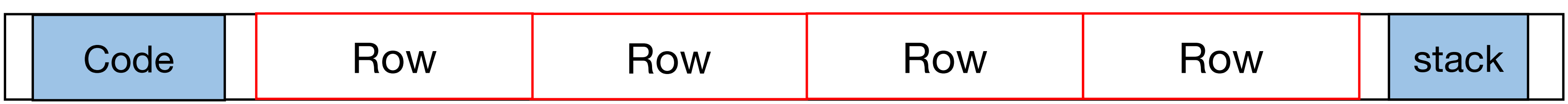
How many misses? How many misses? How many misses?

Memory layout of a matrix



Matrix of numbers
Logically, 2-dimensional

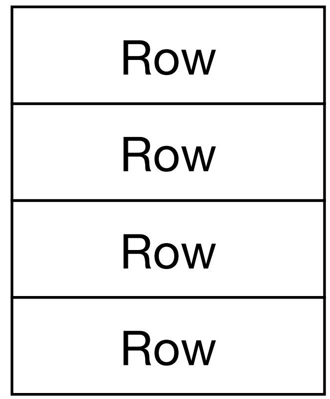
Physically, those rows are arranged along **1-dimension** in the virtual address space



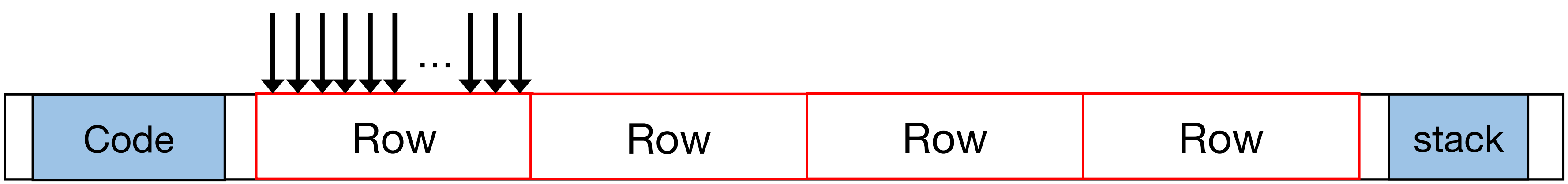
Virtual address space

Memory layout of a matrix

Matrix of numbers
Logically, 2-dimensional

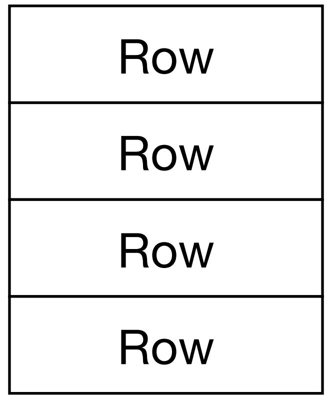


Summing over row:
data consolidated into a few cache lines (CPU cache friendly)

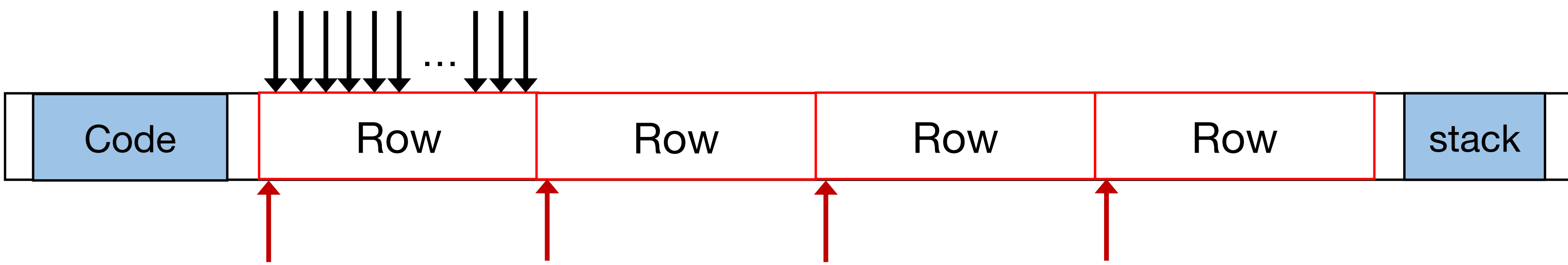


Memory layout of a matrix

Matrix of numbers
Logically, 2-dimensional



Summing over row:
data consolidated into a few cache lines (CPU cache friendly)



Summing over column: each number is in its own cache line and triggers a **cache miss**

Demo ...

Caching policies

- When to load data to a cache?
  - Whenever the program reads something, add it to cache
- When to evict data from a cache (eviction policy)? Several policies:
  - Random: select any data at random for eviction
  - **FIFO** (first-in, first-out): evict whichever data that has been in the cache the longest
  - **LRU** (least recently used): evict which data that has been used the least recently

Worksheet …