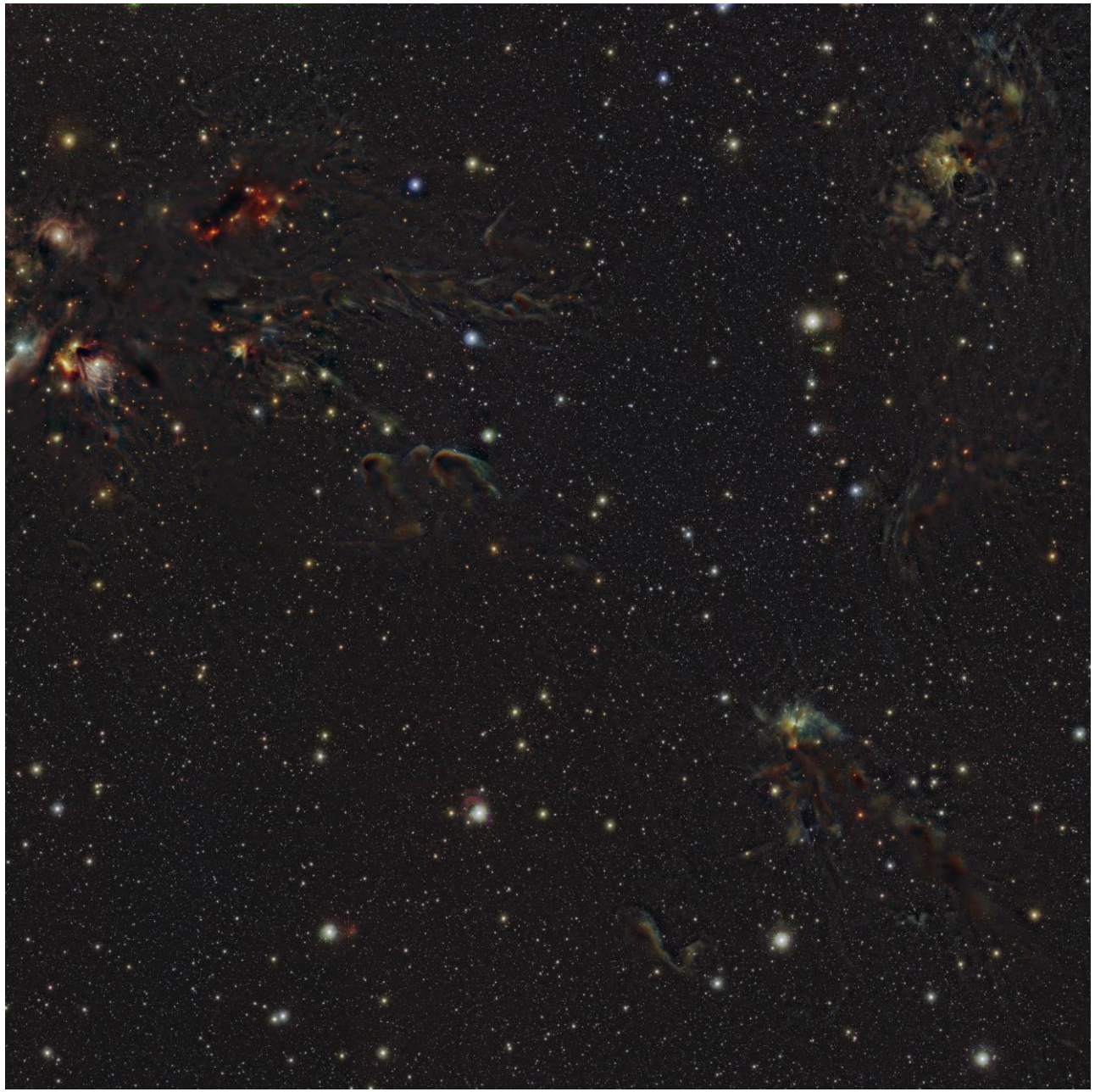
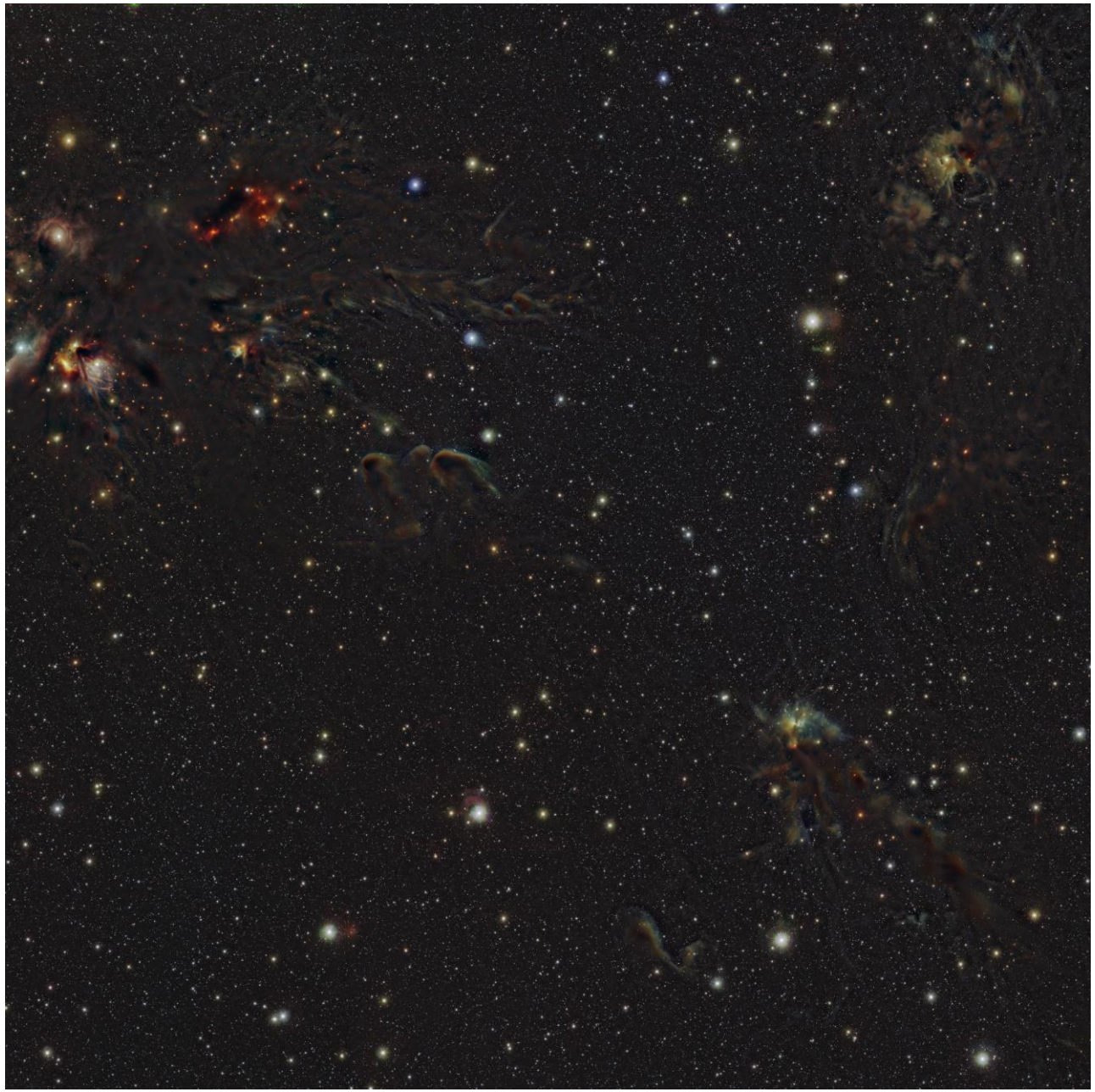
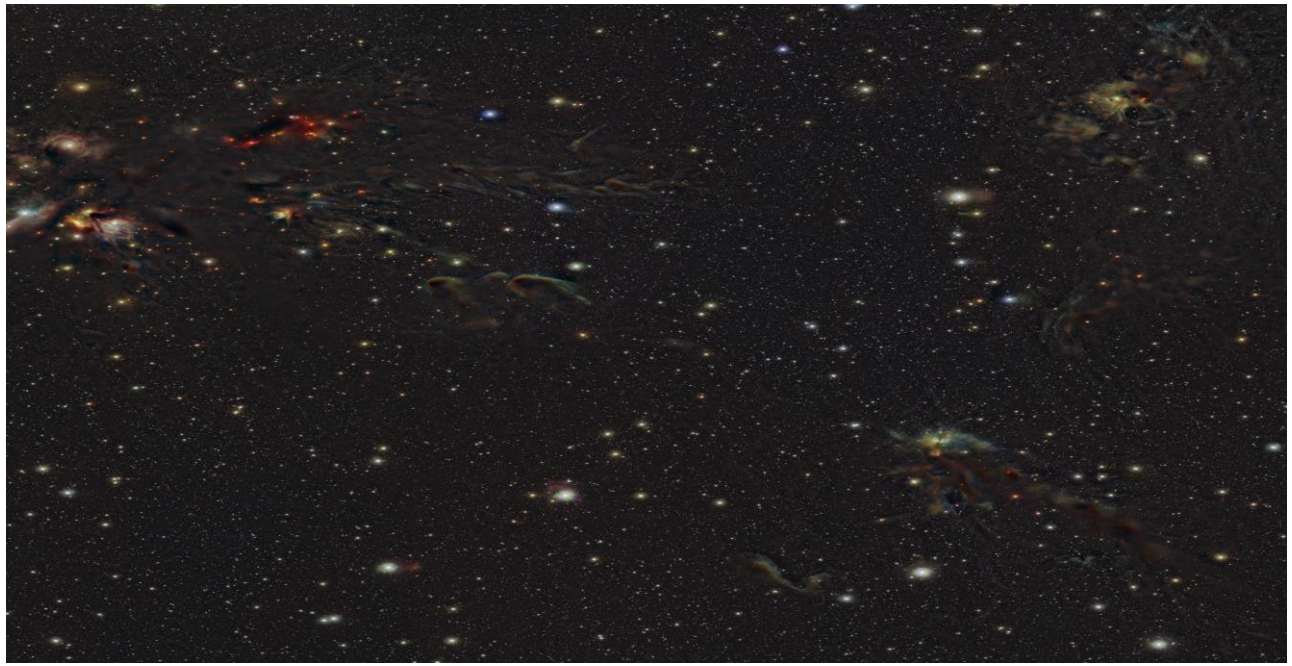
Week 3 – The Memory Hierarchy

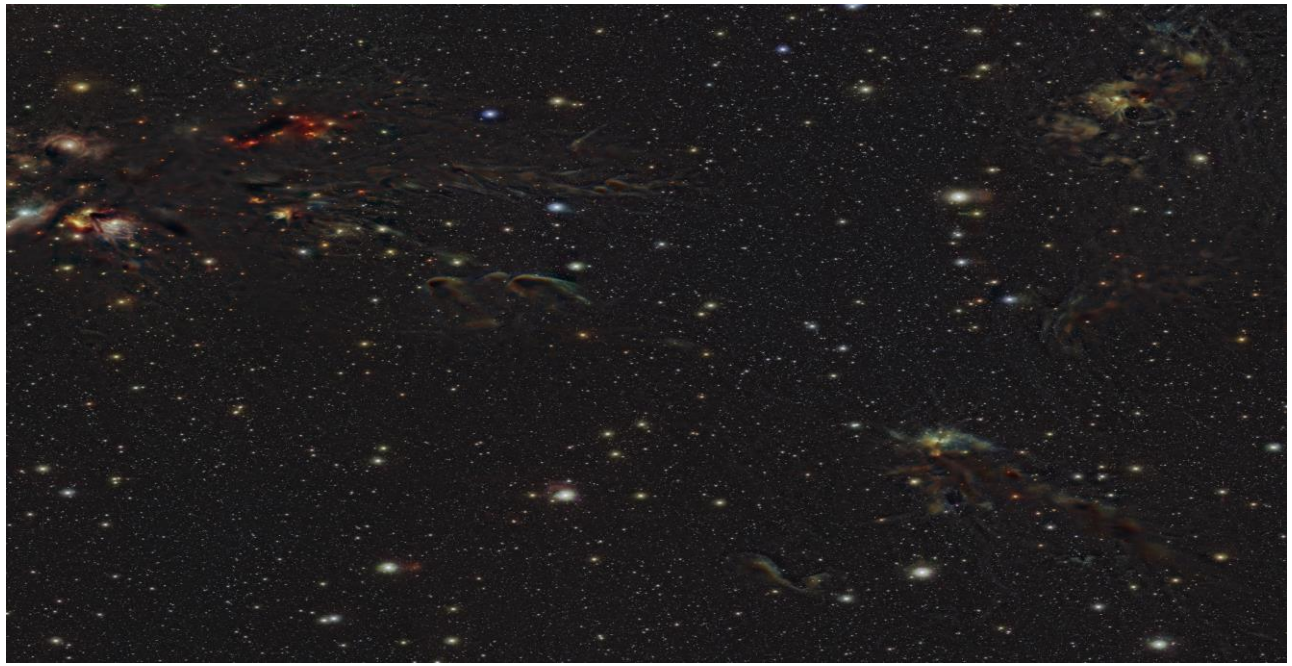
02613 Python and High-Performance Computing



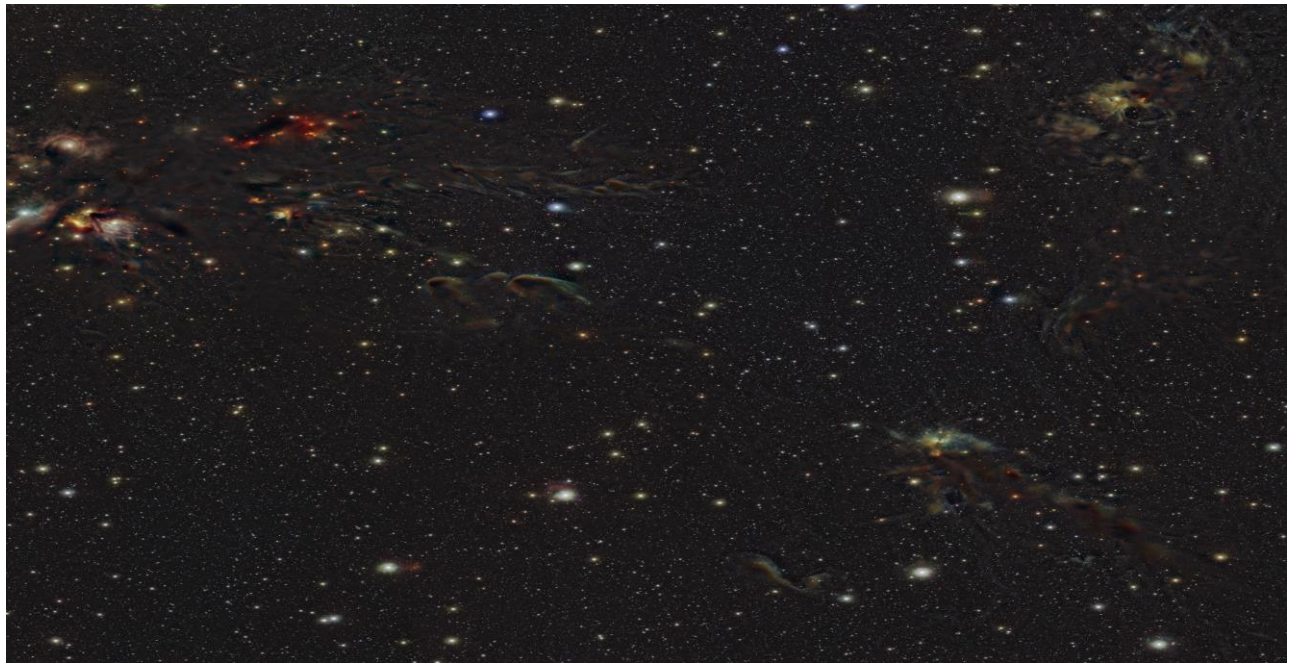

18K
18K





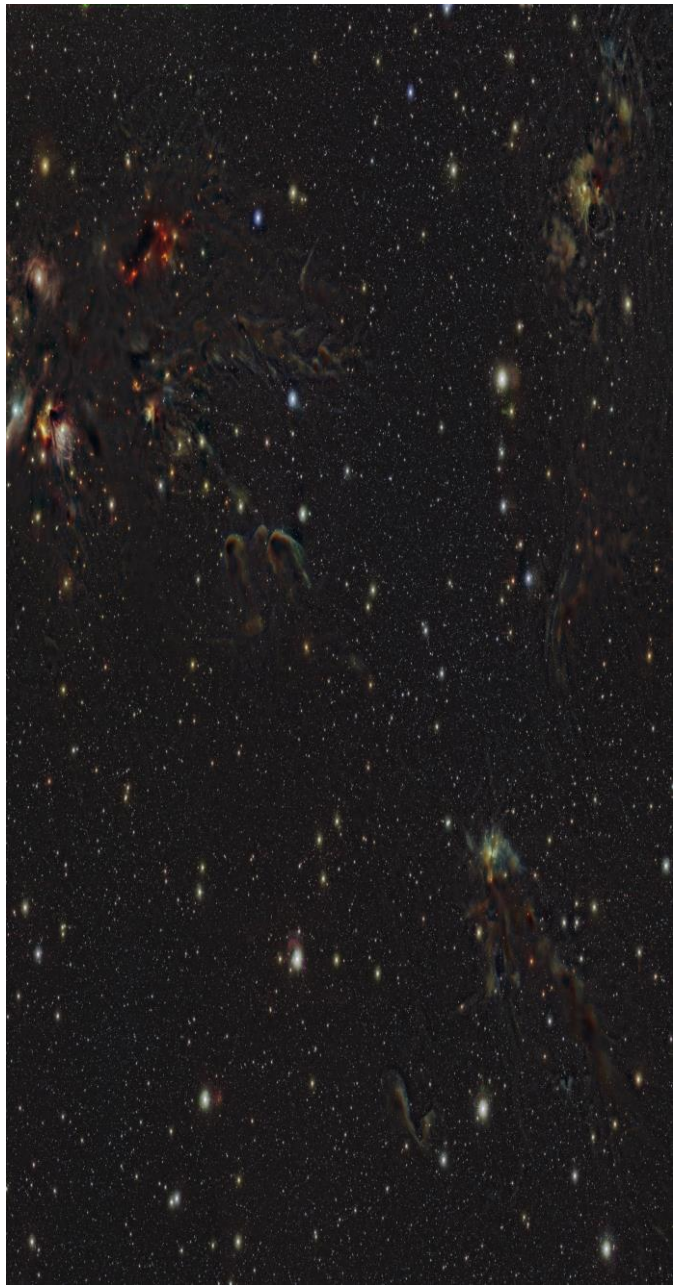


```
im = load_image('stars.png')
n, m, c = im.shape

small = np.empty((n // 2, m, c))
t = time()
for i in range(n // 2):
    small[i,:] = im[i*2,:] + im[i*2+1,:]
t = time() - t
print(t, 'sec')
```



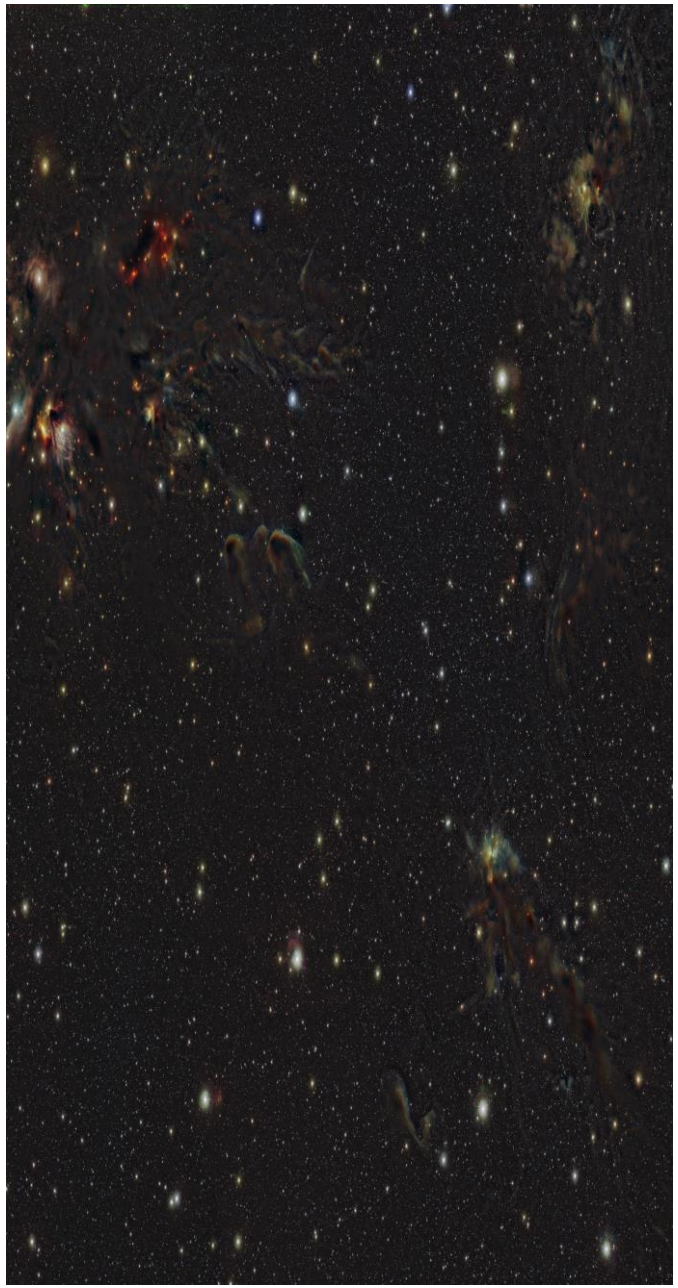
```
im = load_image('stars.png')
n, m, c = im.shape

small = np.empty((n // 2, m, c))
t = time()
for i in range(n // 2):
    small[i,:] = im[i*2,:] + im[i*2+1,:]
t = time() - t
print(t, 'sec')
```

```
1.6689006000001427 sec
```



```
im = load_image('stars.png')
n, m, c = im.shape

small = np.empty((n // 2, m, c))
t = time()
for i in range(n // 2):
    small[i,:] = im[i*2,:] + im[i*2+1,:]
t = time() - t
print(t, 'sec')
```

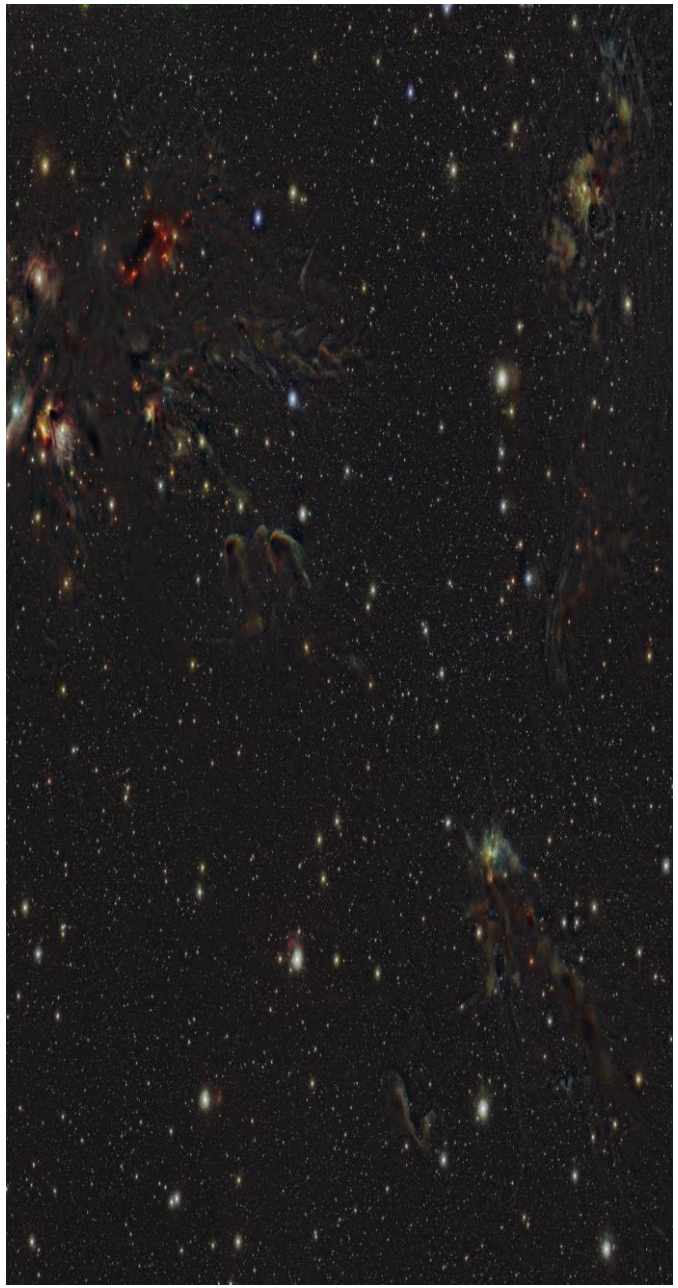
```
1.6689006000001427 sec
```



```
im = load_image('stars.png')
n, m, c = im.shape

small = np.empty((n // 2, m, c))
t = time()
for i in range(n // 2):
    small[i,:] = im[i*2,:] + im[i*2+1,:]
t = time() - t
print(t, 'sec')
```

```
1.6689006000001427 sec
```

```
im = load_image('stars.png')
n, m, c = im.shape

small = np.empty((n, m // 2, c))
t = time()
for i in range(m // 2):
    small[:,i] = im[:,i*2] + im[:,i*2+1]
t = time() - t
print(t, 'sec')
```



```
im = load_image('stars.png')
n, m, c = im.shape

small = np.empty((n // 2, m, c))
t = time()
for i in range(n // 2):
    small[i,:] = im[i*2,:] + im[i*2+1,:]
t = time() - t
print(t, 'sec')
```

```
1.6689006000001427 sec
```

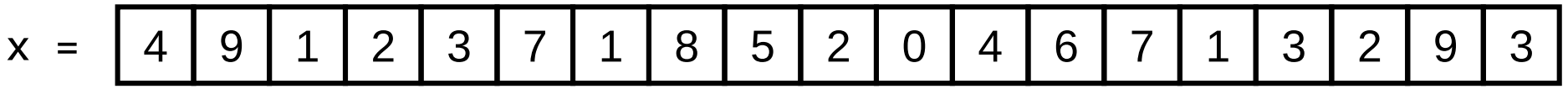
```
im = load_image('stars.png')
n, m, c = im.shape

small = np.empty((n, m // 2, c))
t = time()
for i in range(m // 2):
    small[:,i] = im[:,i*2] + im[:,i*2+1]
t = time() - t
print(t, 'sec')
```

```
20.149567999999817 sec
```

Today

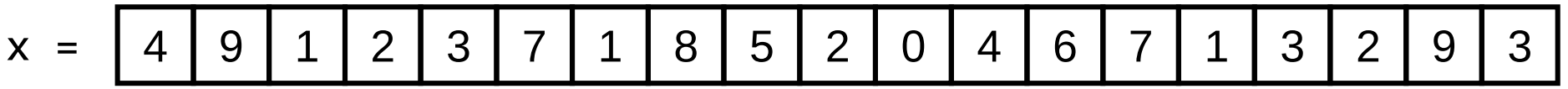
- What is performance
- Memory hierarchies and how to deal with them
  - Caches
  - Compressed data

Performance




```
s = sum(x)
```

Performance




```
s = sum(x)
```

Wall-Clock Time

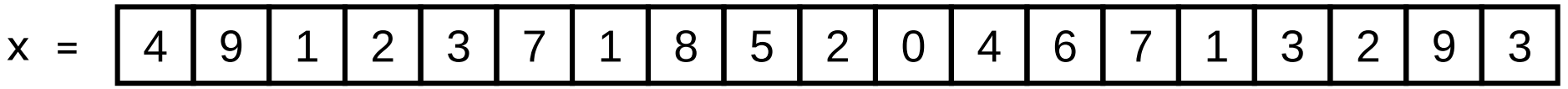
Performance

x =	4	9	1	2	3	7	1	8	5	2	0	4	6	7	1	3	2	9	3

```
s = sum(x)
```

Wall-Clock Time

Performance



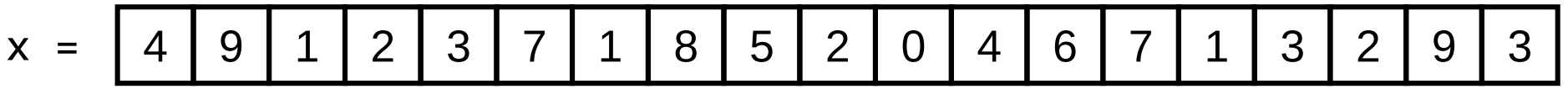

```
s = sum(x)
```

Wall-Clock Time

How long did we wait?

```
t = time()
s = sum(x)
t = time() - t
```

Performance



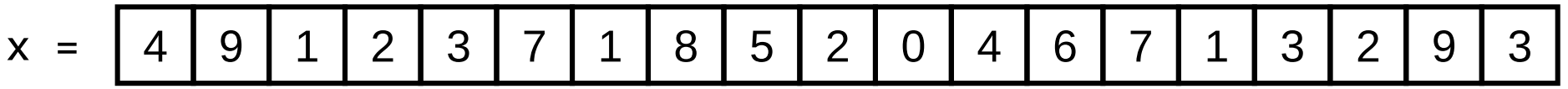

```
s = sum(x)
```

Wall-Clock Time

How long did we wait?

```
t = time()
for _ in range(100):
    s = sum(x)
t = time() - t
```

Performance




```
s = sum(x)
```

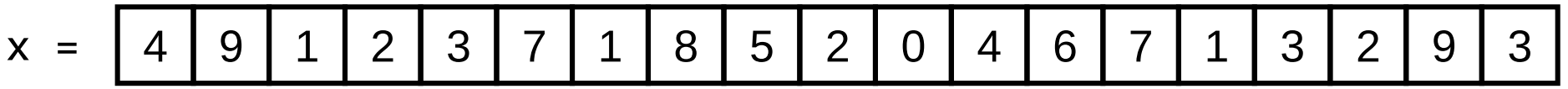
Wall-Clock Time > **CPU time**

How long did we wait?

How much did the CPU work?

Generally, a bit less than wall time

Performance




```
s = sum(x)
```

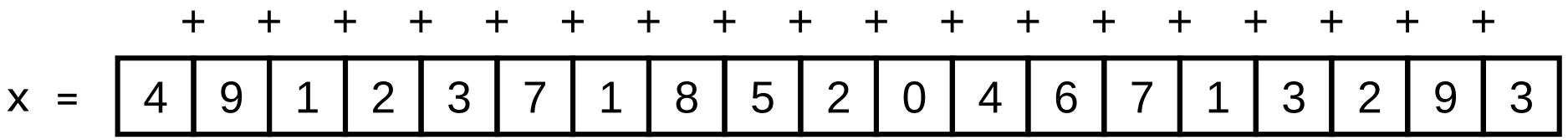
Wall-Clock Time < **CPU time**

How long did we wait?

How much did the CPU work?

Summed over all cores =
May be larger for parallel work

Performance




```
s = sum(x)
```

Wall-Clock Time

How long did we wait?

CPU time

How much did the CPU work?

FLOP/s

How fast did the CPU work?

Floating point operations / second

$$\text{FLOP/s} = \frac{\#\text{FLOPs}}{\text{Wall time}}$$

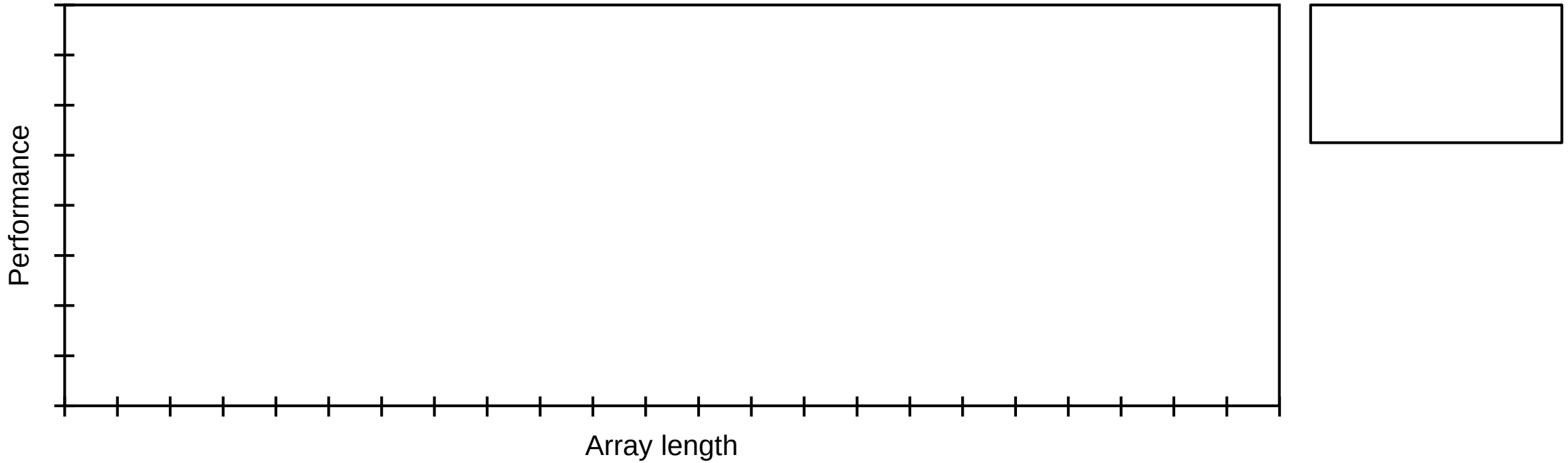
No auto. tool to measure #FLOPs

Must look at code!

Performance

x = | 4 | 9 | 1 | 2 | 3 | 7 | 1 | 8 | 5 | 2 | 0 | 4 | 6 | 7 | 1 | 3 | 2 | 9 | 3 |

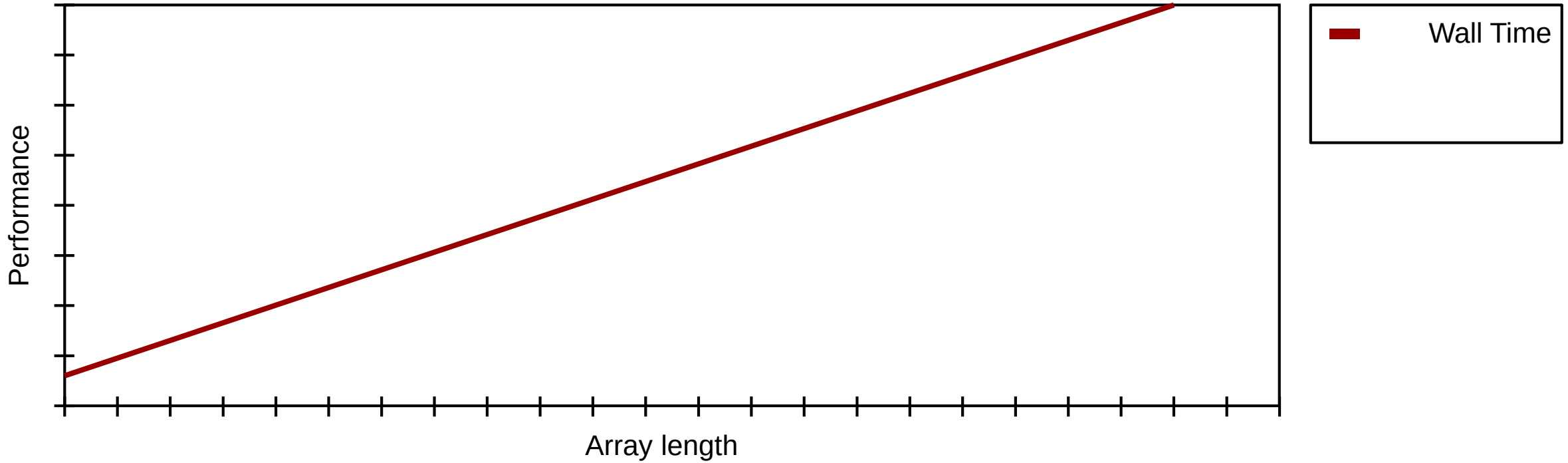
```
s = sum(x)
```



Performance

x = | 4 | 9 | 1 | 2 | 3 | 7 | 1 | 8 | 5 | 2 | 0 | 4 | 6 | 7 | 1 | 3 | 2 | 9 | 3 |

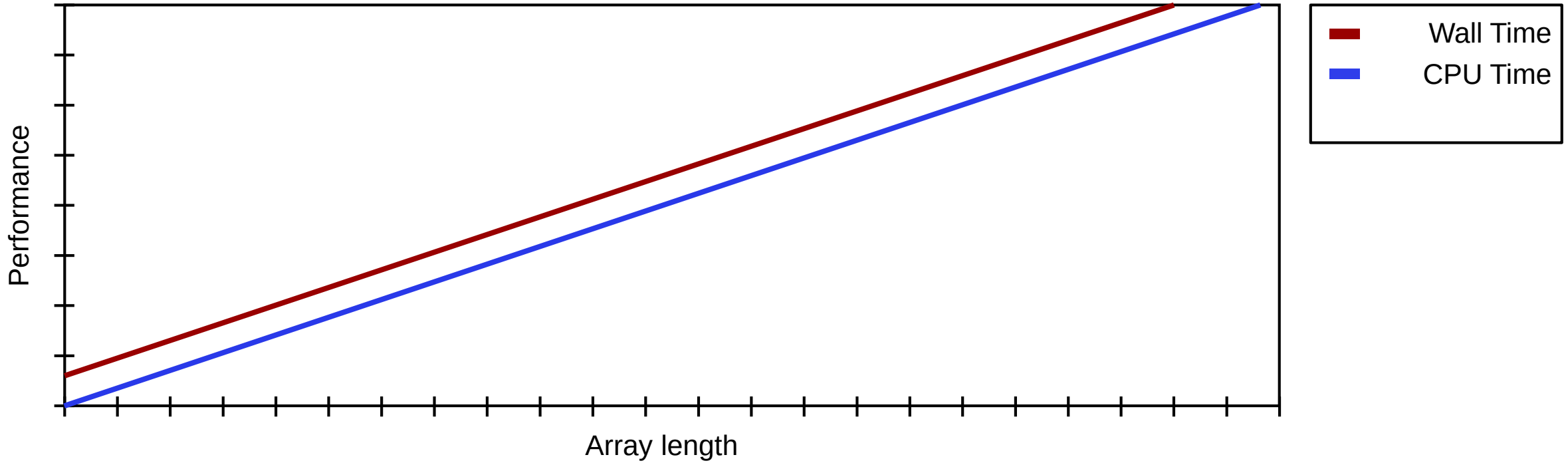
```
s = sum(x)
```



Performance

x =	4	9	1	2	3	7	1	8	5	2	0	4	6	7	1	3	2	9	3

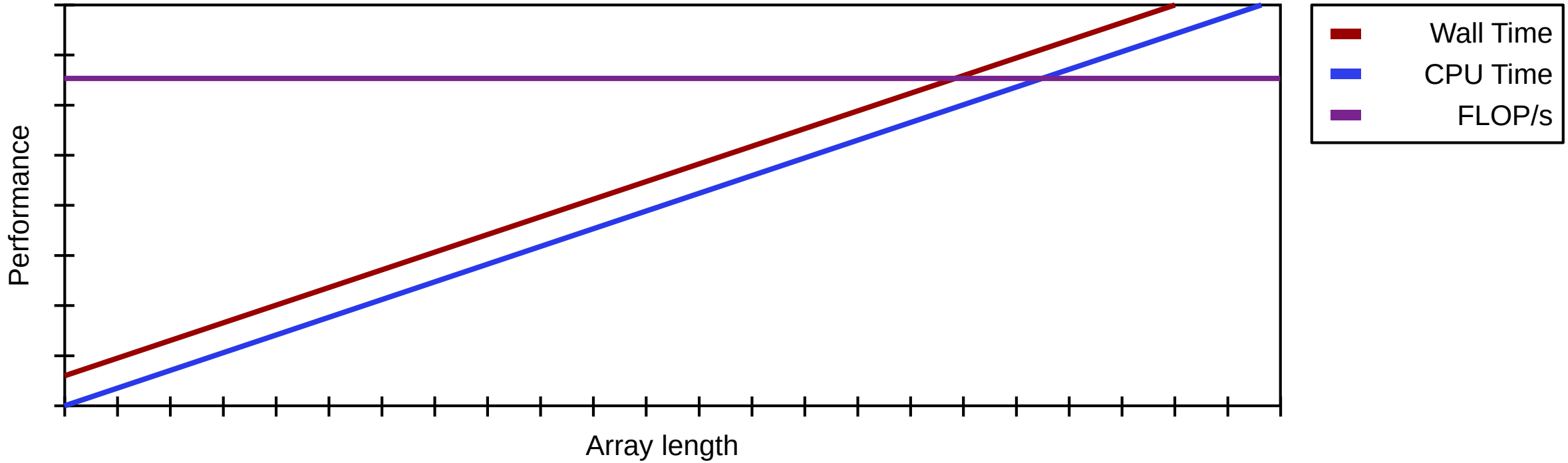
```
s = sum(x)
```



Performance

x = | 4 | 9 | 1 | 2 | 3 | 7 | 1 | 8 | 5 | 2 | 0 | 4 | 6 | 7 | 1 | 3 | 2 | 9 | 3 |

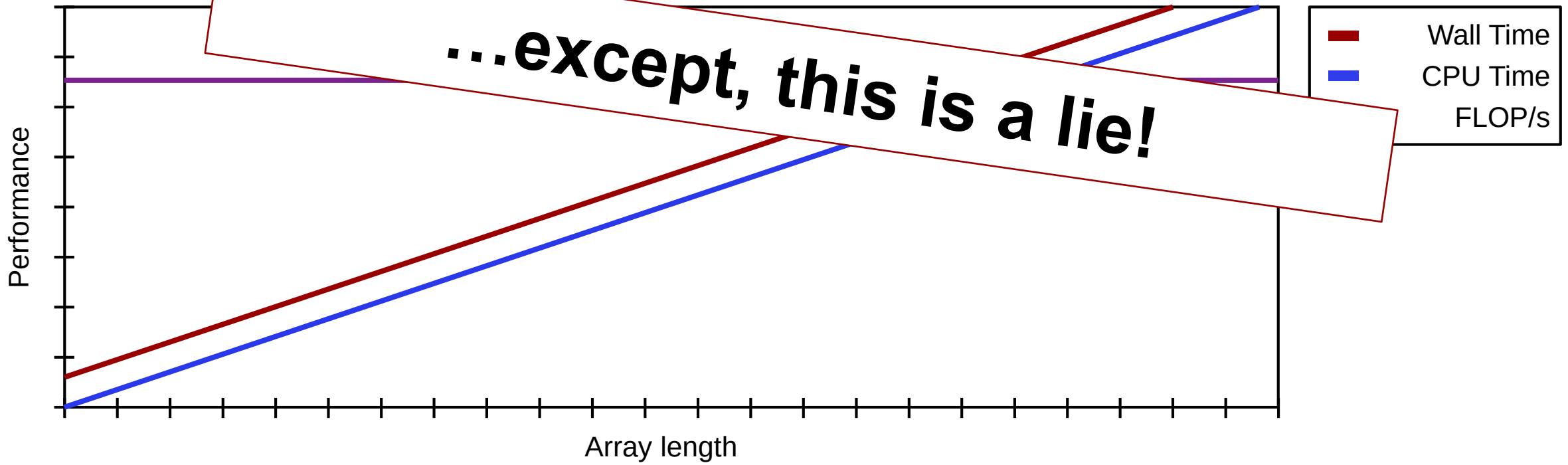
```
s = sum(x)
```



Performance

x = | 4 | 9 | 1 | 2 | 3 | 7 | 1 | 8 | 5 | 2 | 0 | 4 | 6 | 7 | 1 | 3 | 2 | 9 | 3 |

```
s = sum(x)
```

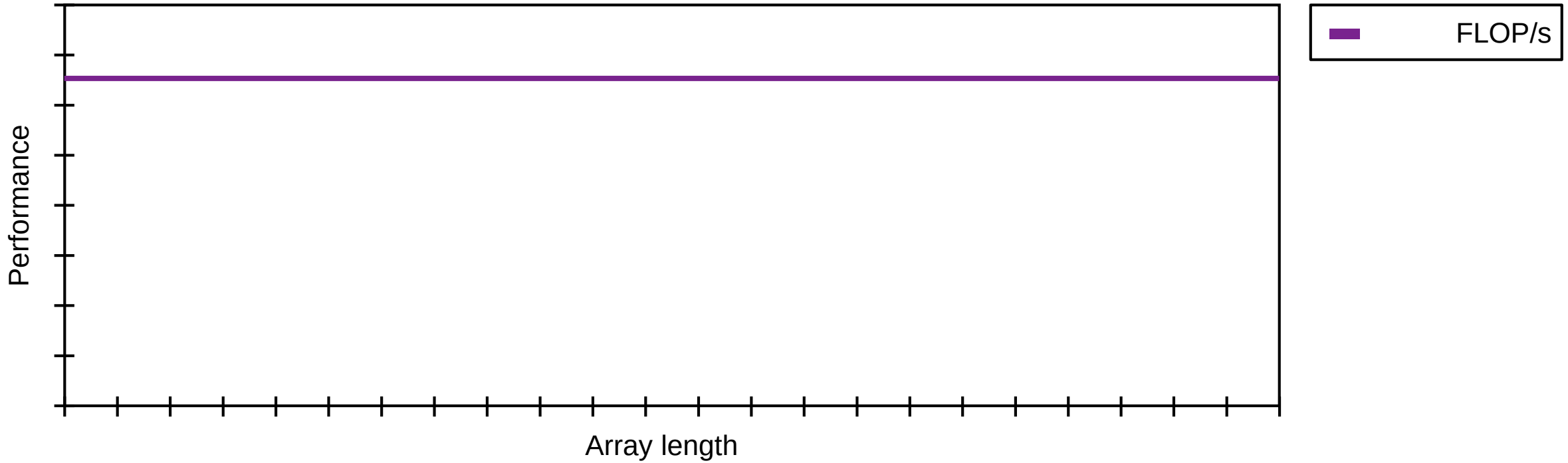


The Memory Hierarchy: Caches

Performance

x = | 4 | 9 | 1 | 2 | 3 | 7 | 1 | 8 | 5 | 2 | 0 | 4 | 6 | 7 | 1 | 3 | 2 | 9 | 3 |

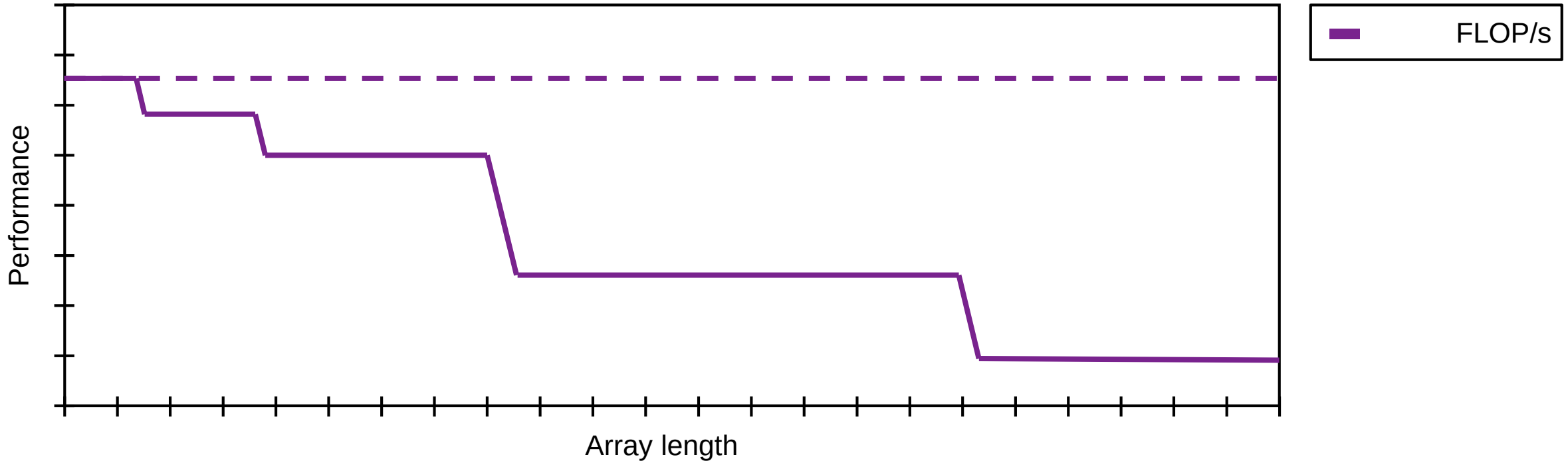
```
s = sum(x)
```



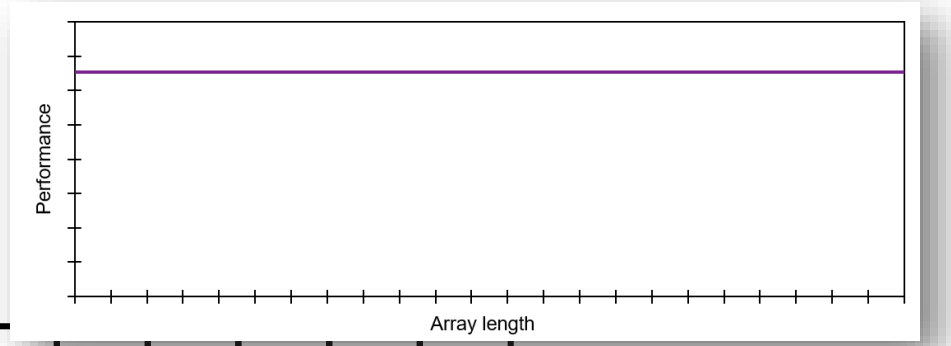
Performance

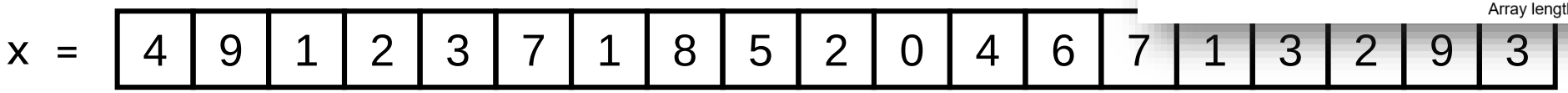
x = | 4 | 9 | 1 | 2 | 3 | 7 | 1 | 8 | 5 | 2 | 0 | 4 | 6 | 7 | 1 | 3 | 2 | 9 | 3 |

```
s = sum(x)
```

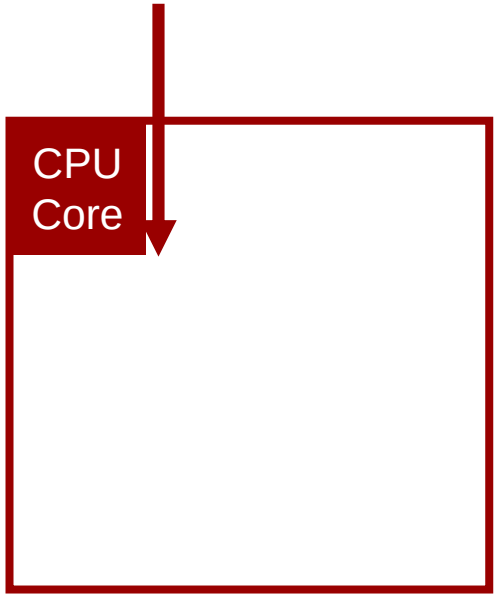


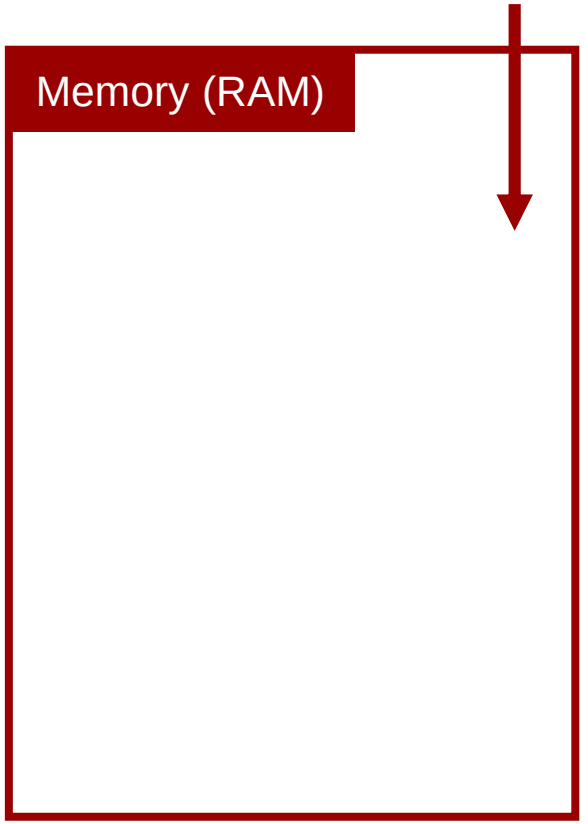
Performance: the fantasy



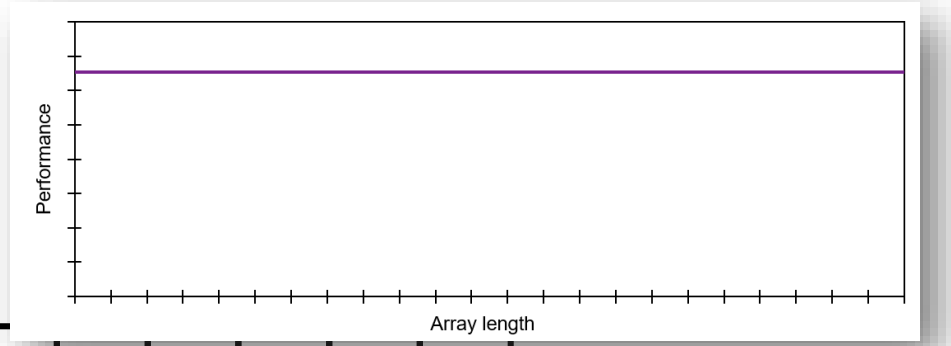


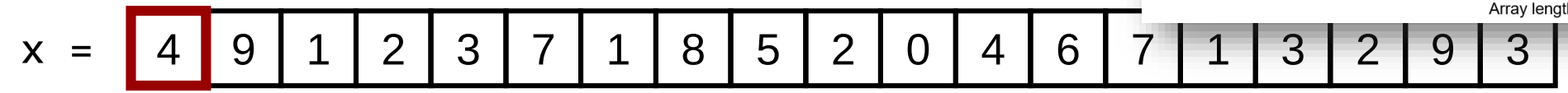

```
s = 0
```



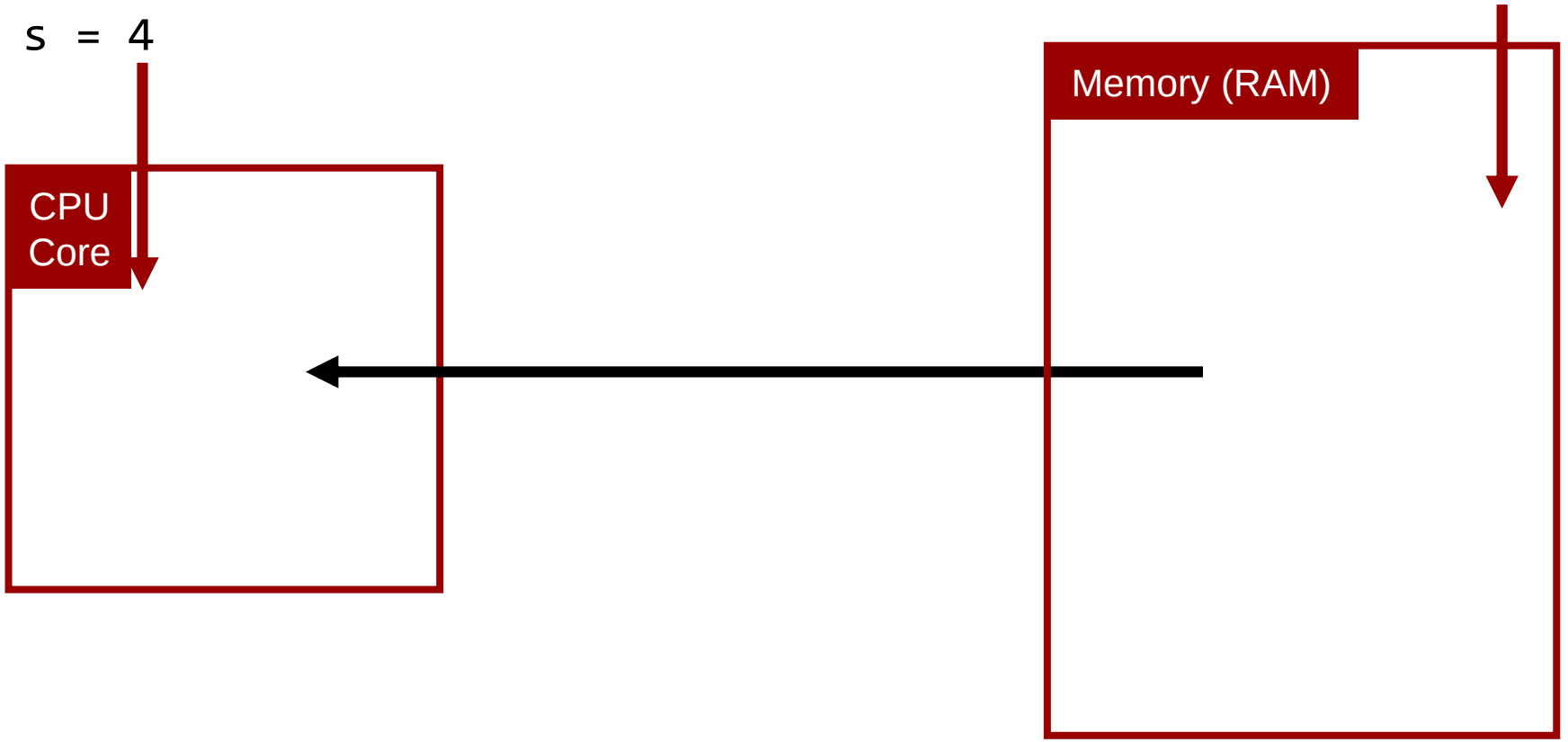


Performance: the fantasy

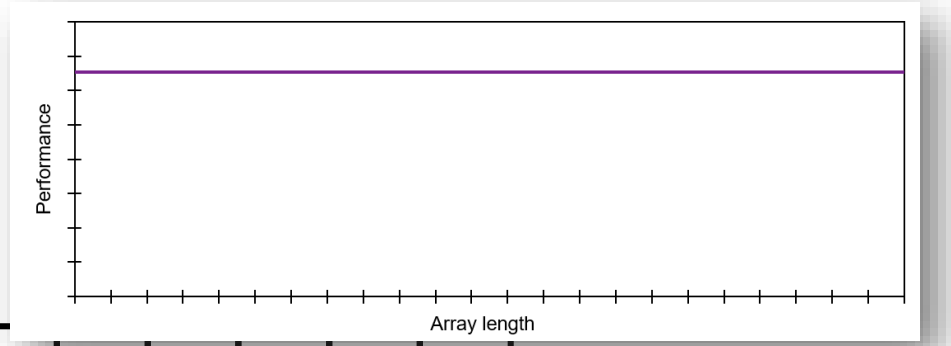


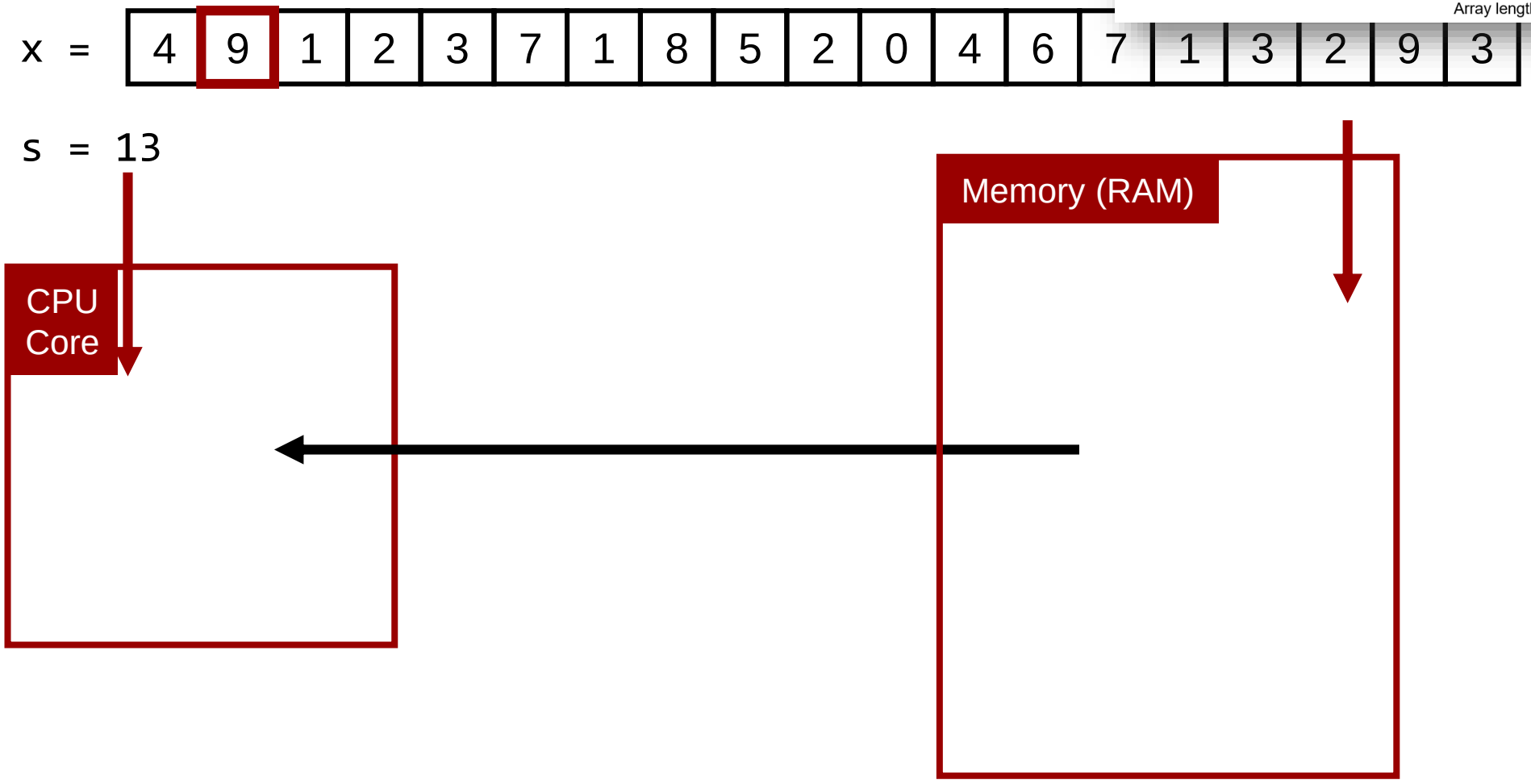


s = 4

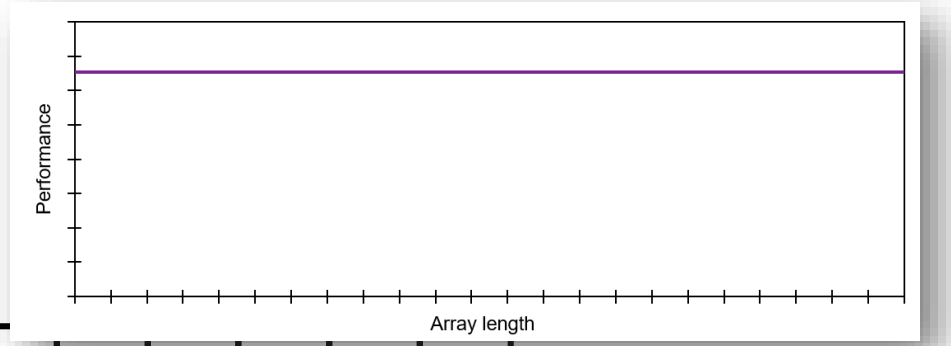


Performance: the fantasy



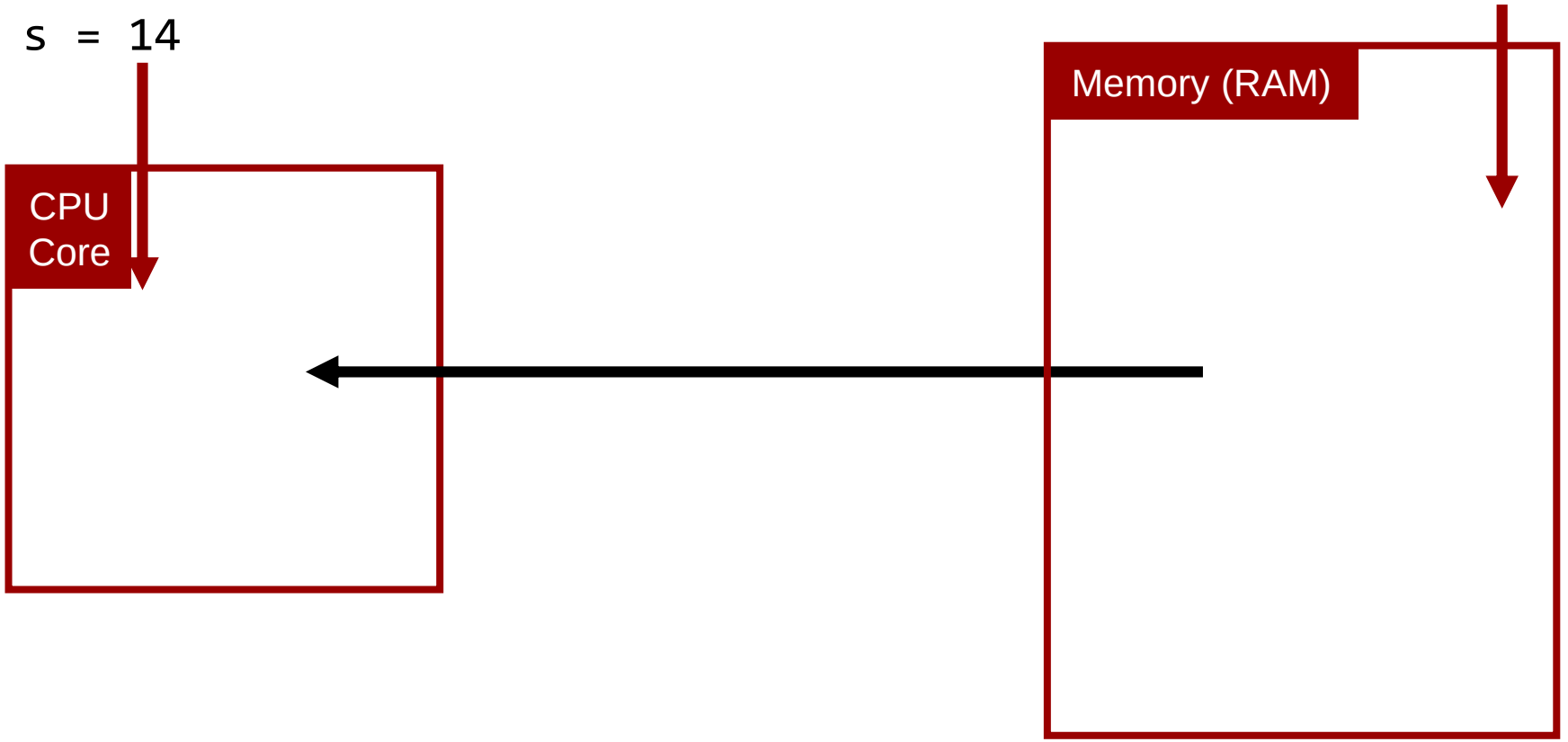


Performance: the fantasy

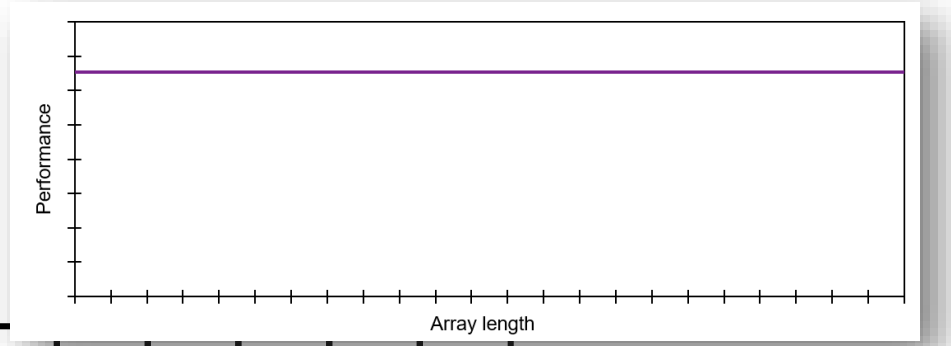


x = | 4 | 9 | 1 | 2 | 3 | 7 | 1 | 8 | 5 | 2 | 0 | 4 | 6 | 7 | 1 | 3 | 2 | 9 | 3 |

s = 14

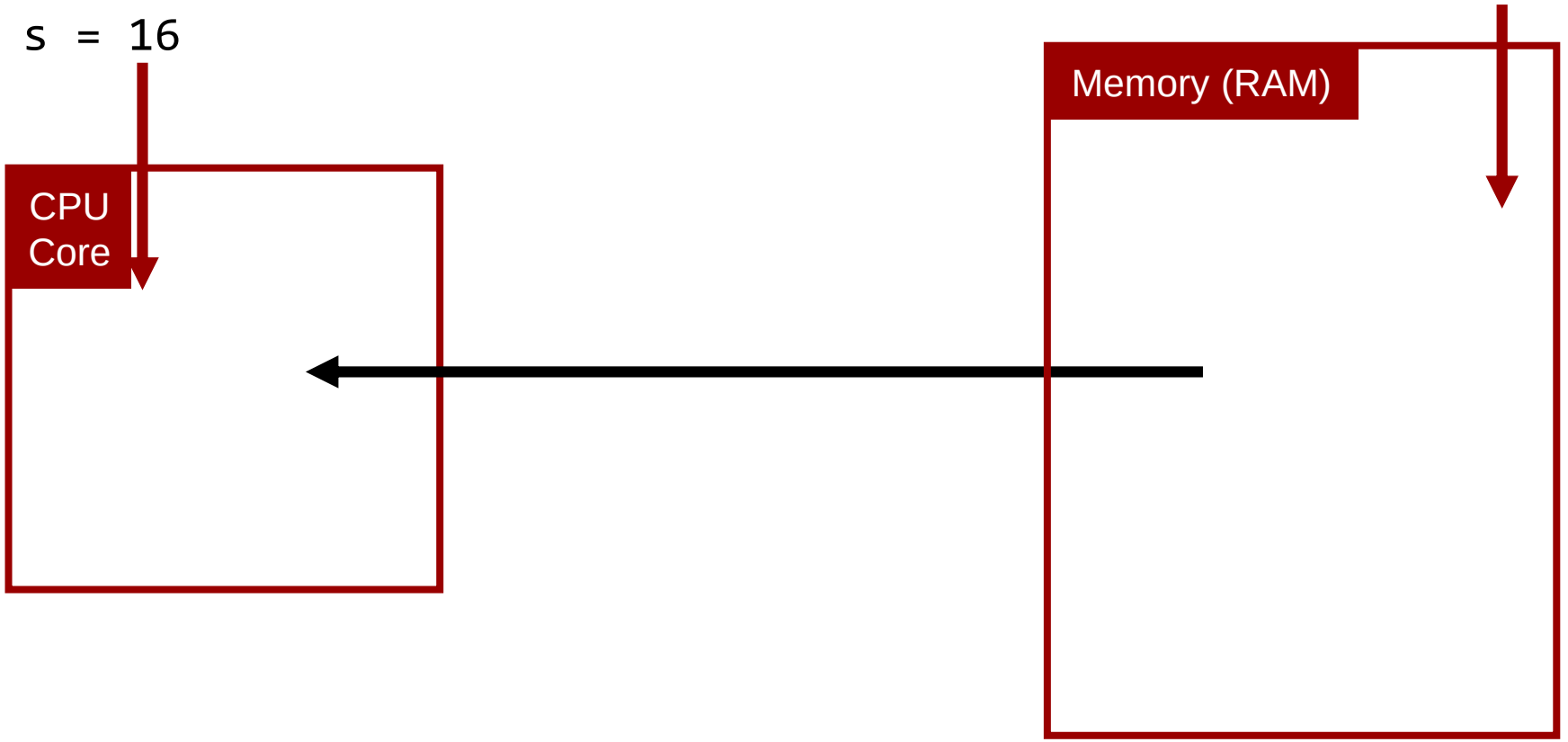


Performance: the fantasy

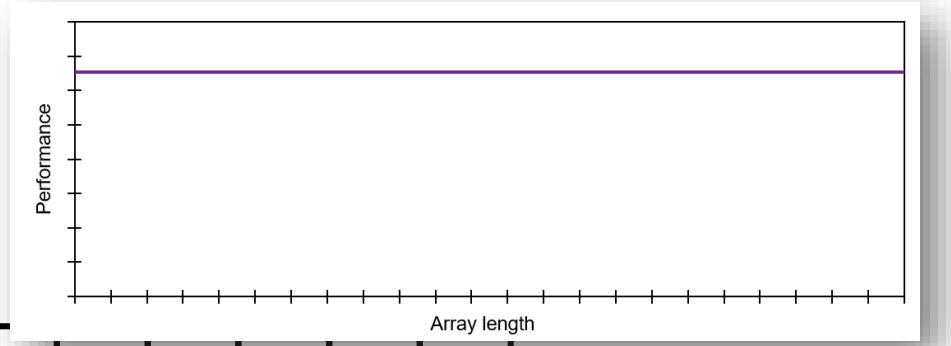


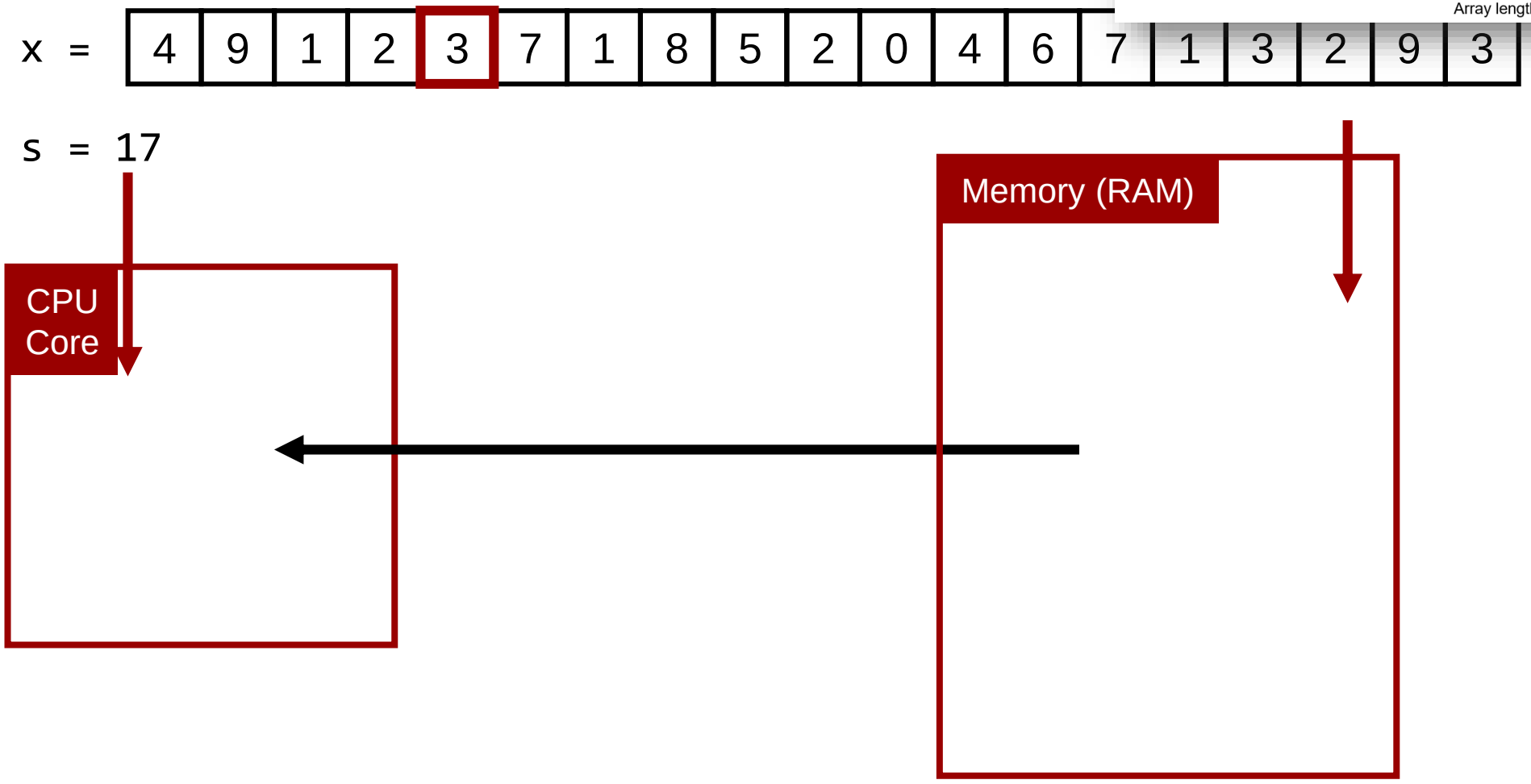
x = | 4 | 9 | 1 | 2 | 3 | 7 | 1 | 8 | 5 | 2 | 0 | 4 | 6 | 7 | 1 | 3 | 2 | 9 | 3 |

s = 16

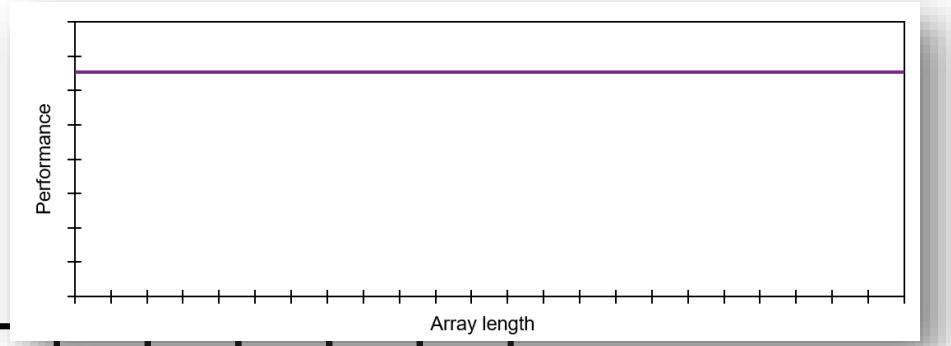


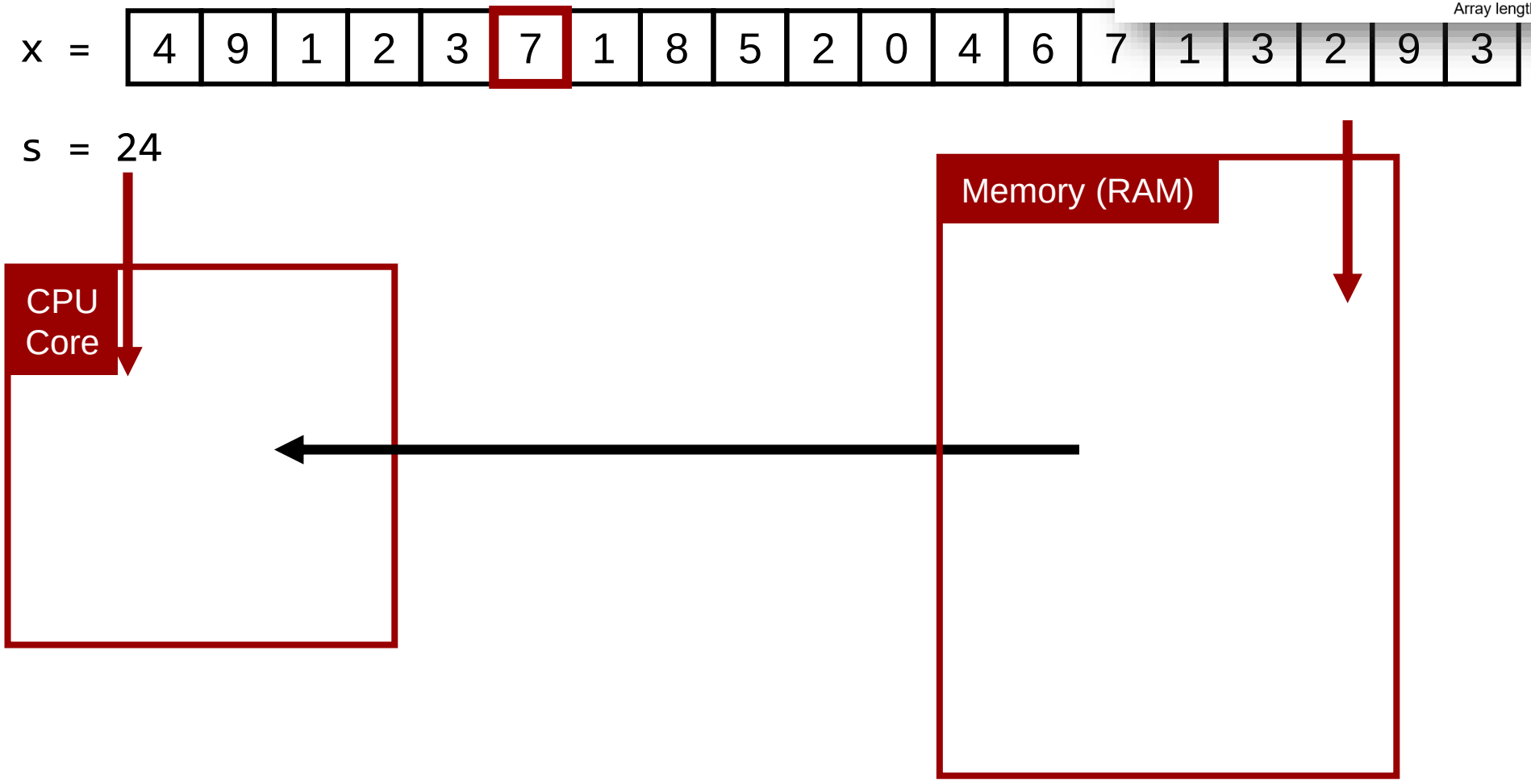
Performance: the fantasy



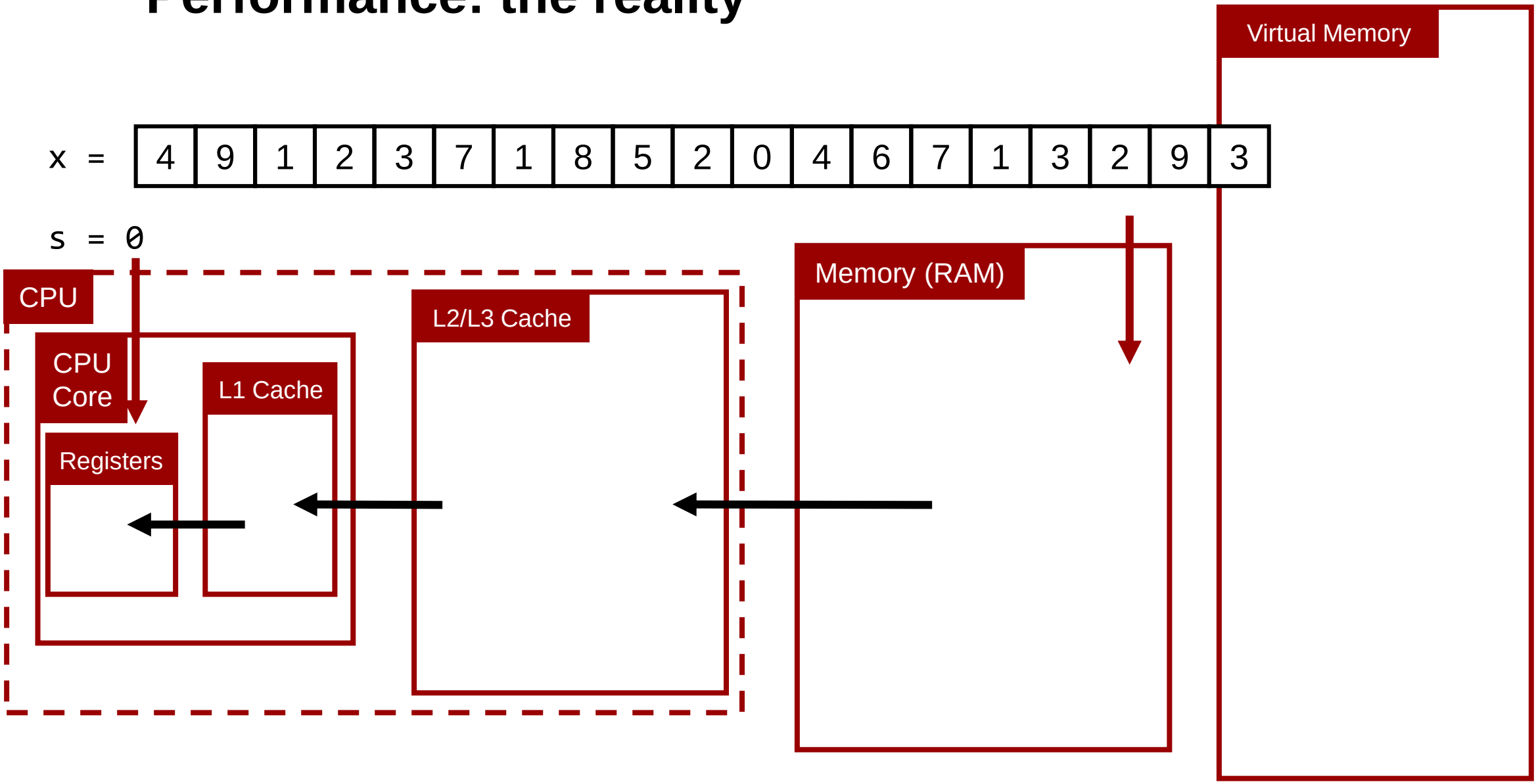


Performance: the fantasy

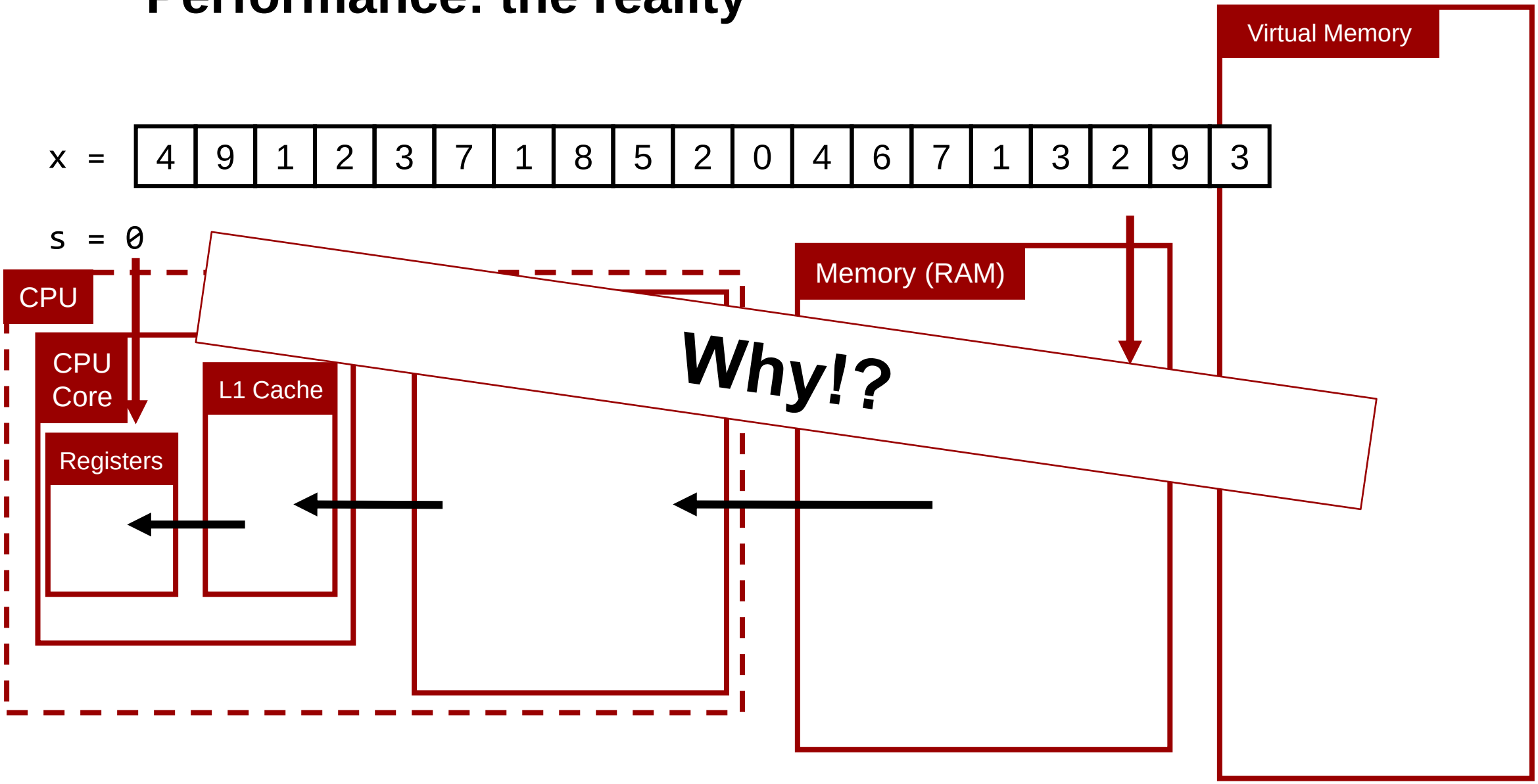




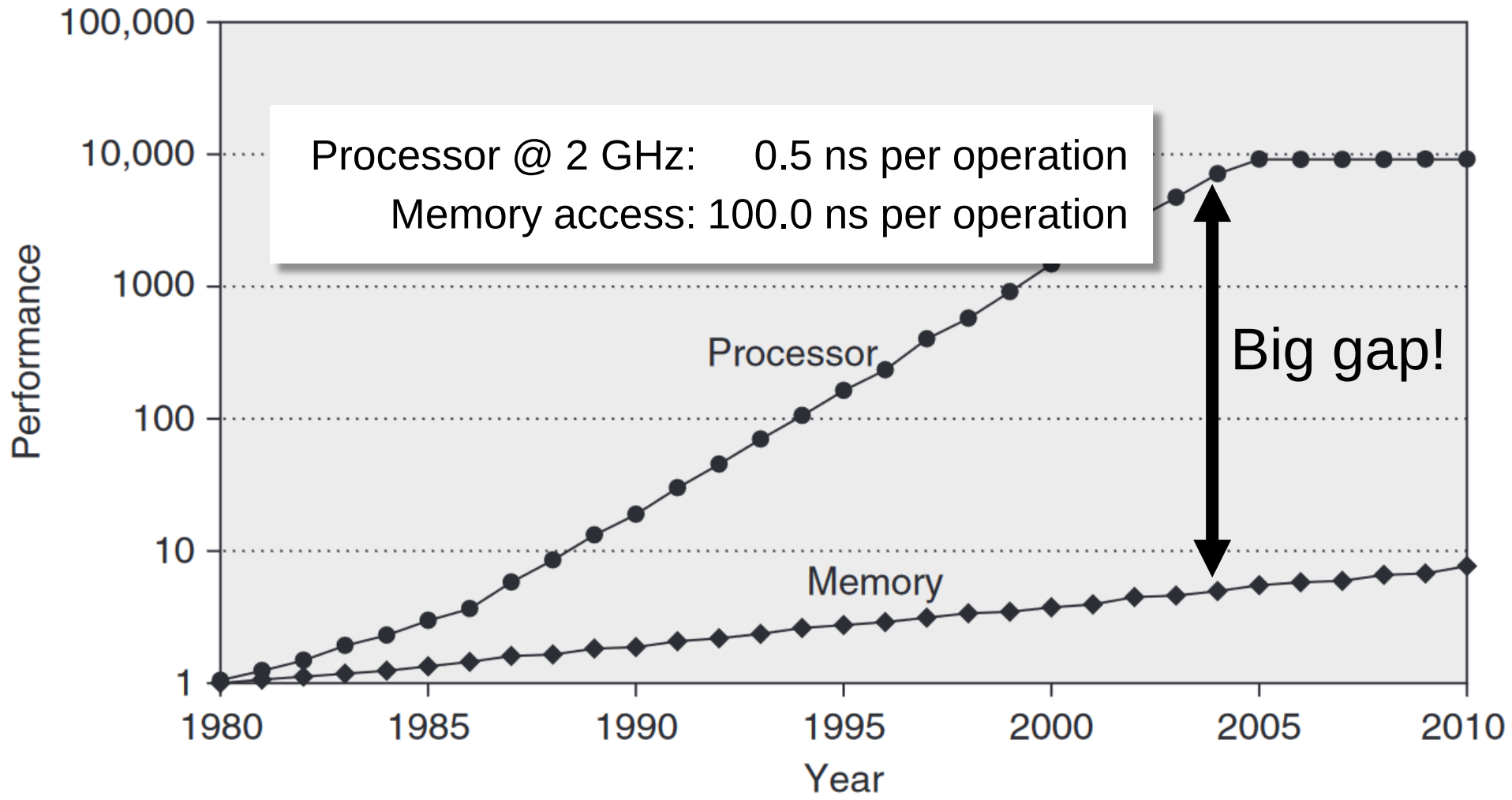
Performance: the reality



Performance: the reality

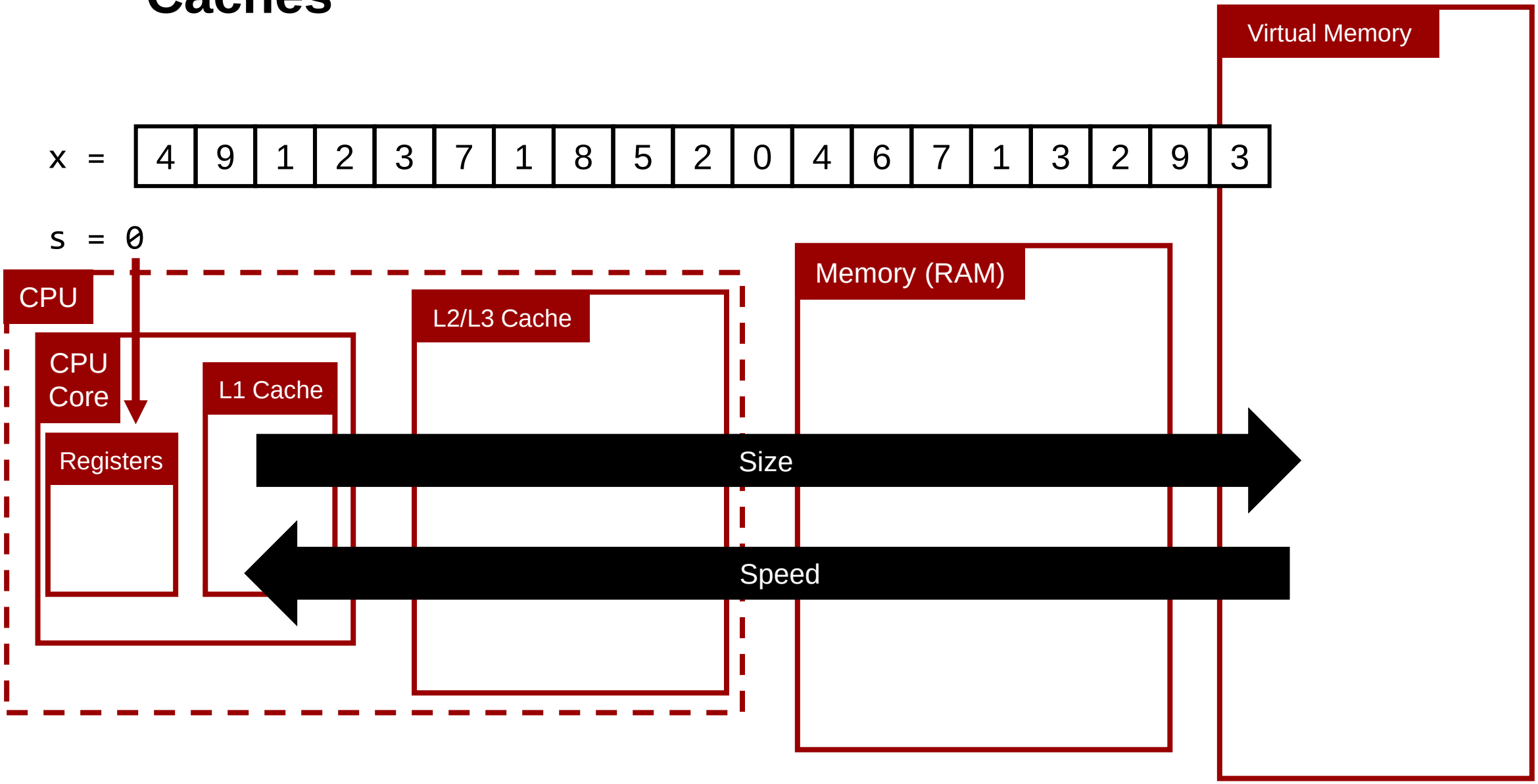


Performance: the reality

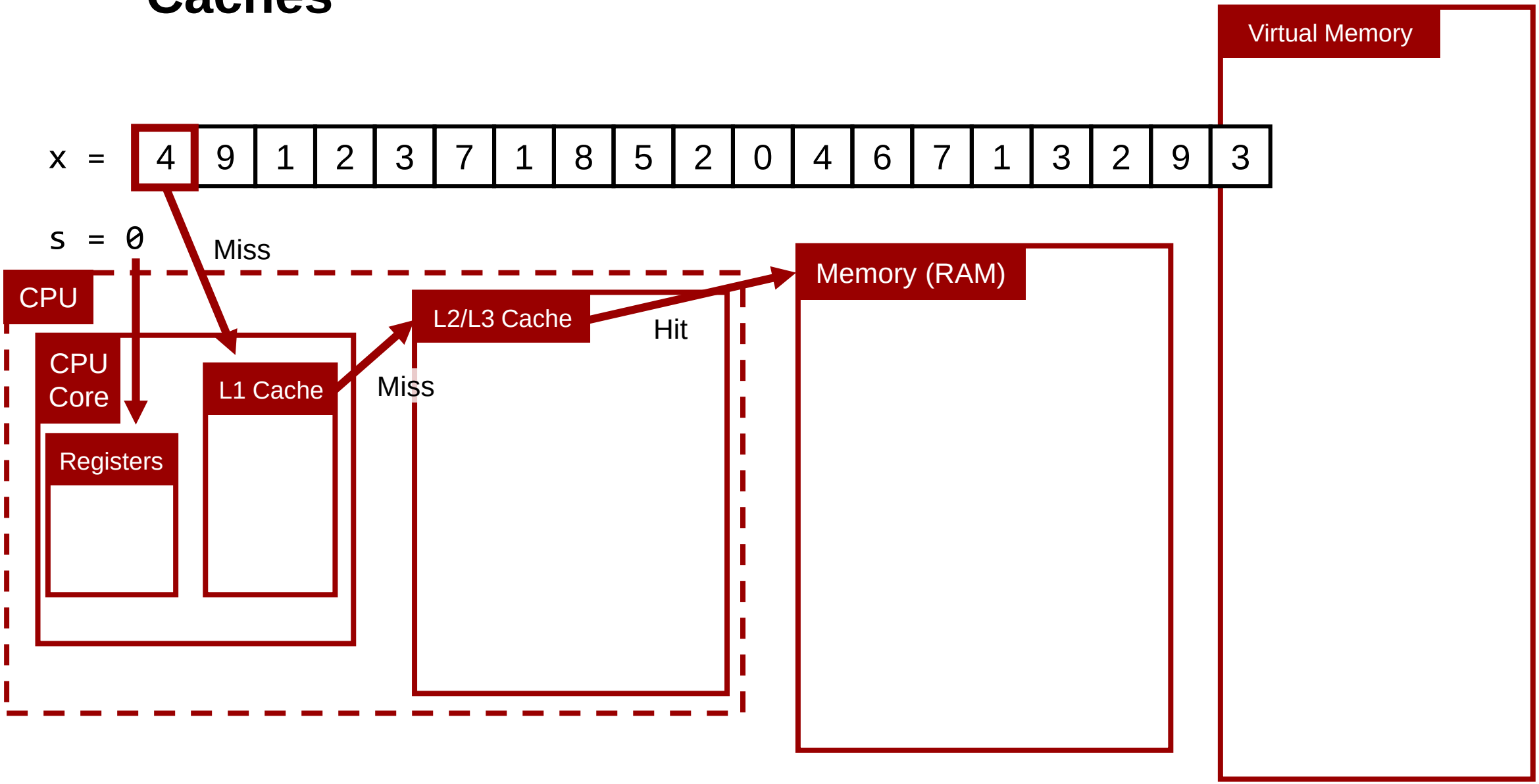


Hennesy & Patterson, ”Computer Architecture: A Quantitative Approach”, 5th Ed.

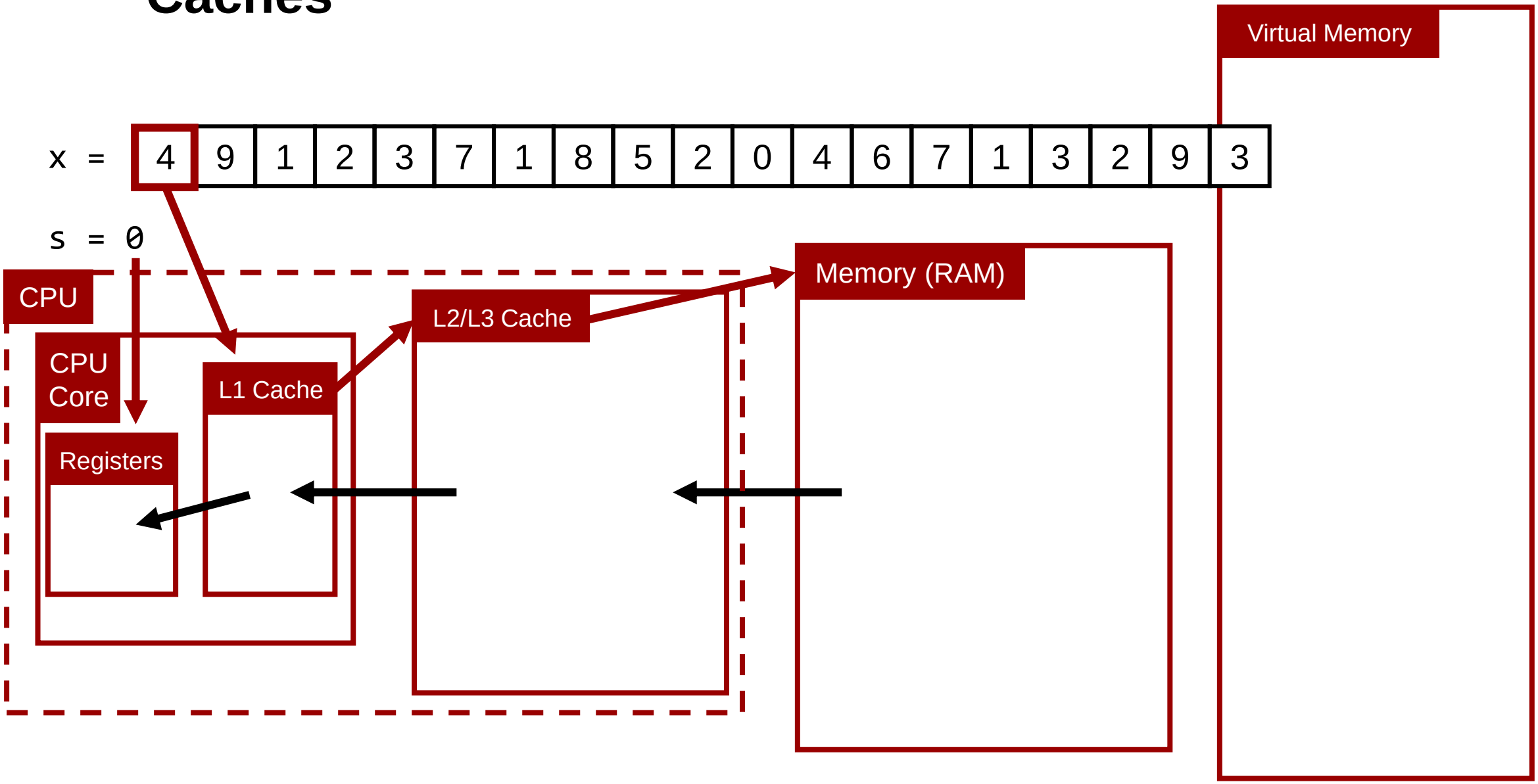
Caches


Virtual Memory
x =
4 9 1 2 3 7 1 8 5 2 0 4 6 7 1 3 2 9 3
s = 0
CPU
CPU Core
Registers
L1 Cache
L2/L3 Cache
Memory (RAM)
Size
Speed

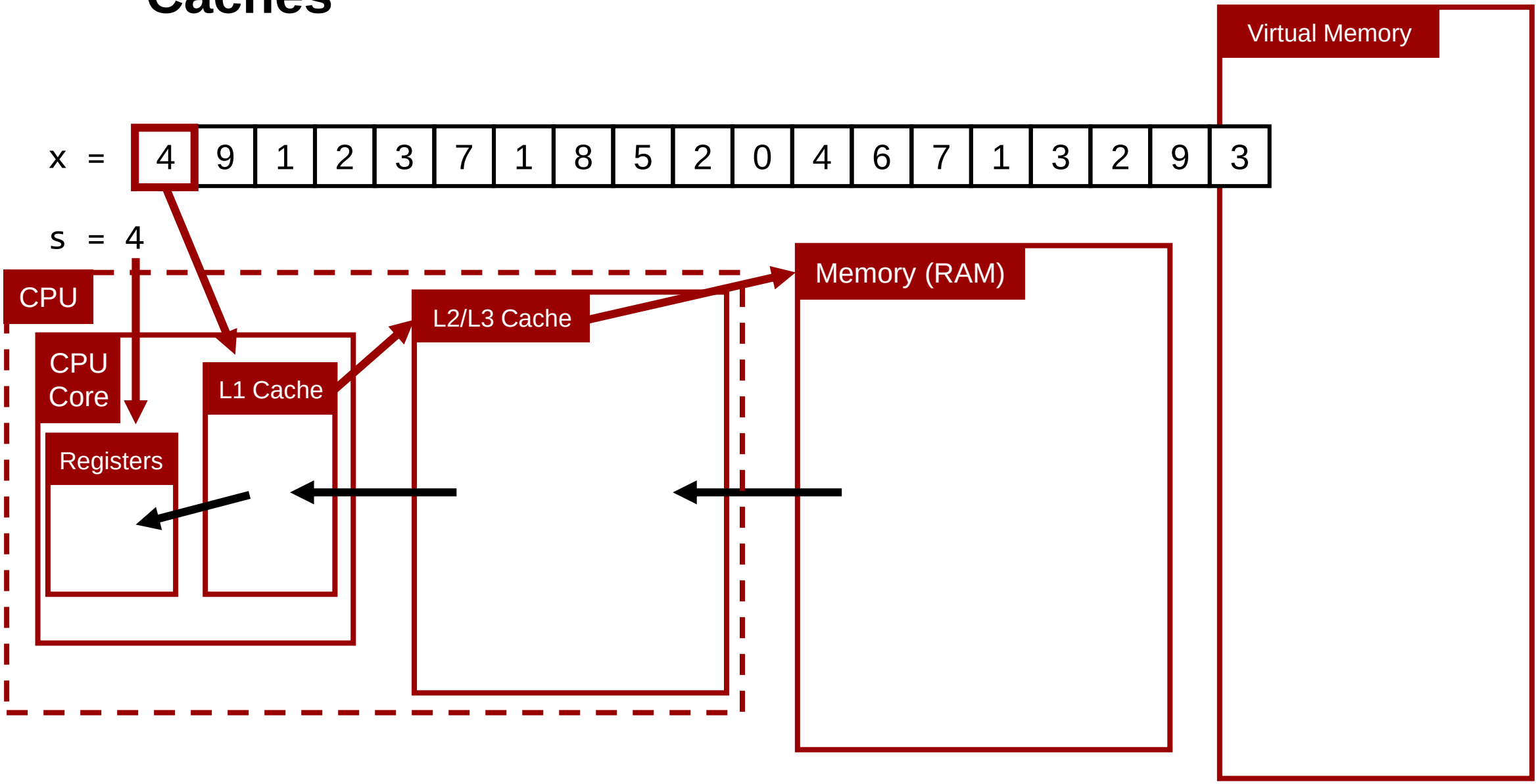
Caches


Virtual Memory
x =
4 9 1 2 3 7 1 8 5 2 0 4 6 7 1 3 2 9 3
s = 0
Miss
CPU
CPU Core
Registers
L1 Cache
Miss
L2/L3 Cache
Hit
Memory (RAM)

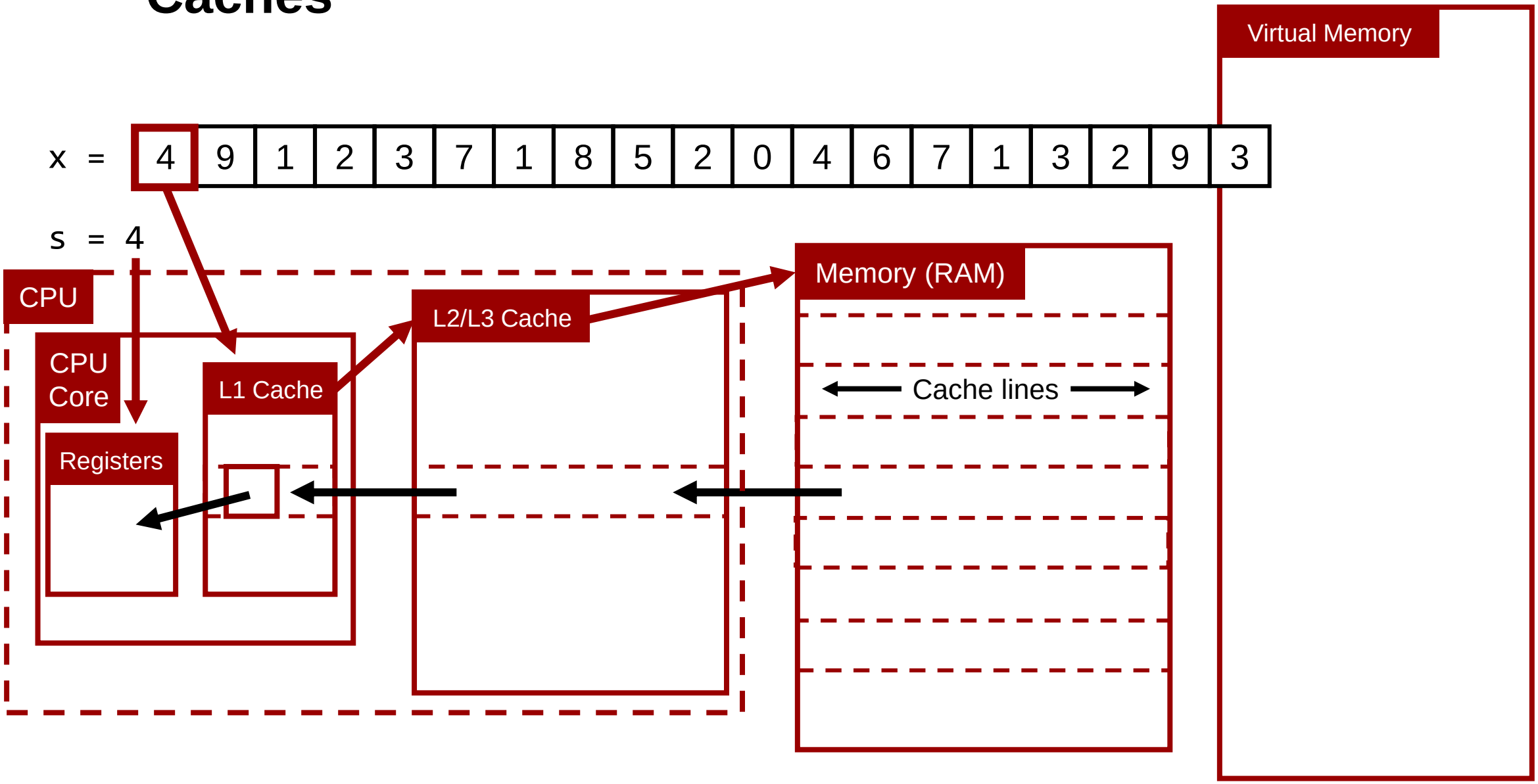
Caches


Virtual Memory
x =
4 9 1 2 3 7 1 8 5 2 0 4 6 7 1 3 2 9 3
s = 0
CPU
CPU Core
Registers
L1 Cache
L2/L3 Cache
Memory (RAM)

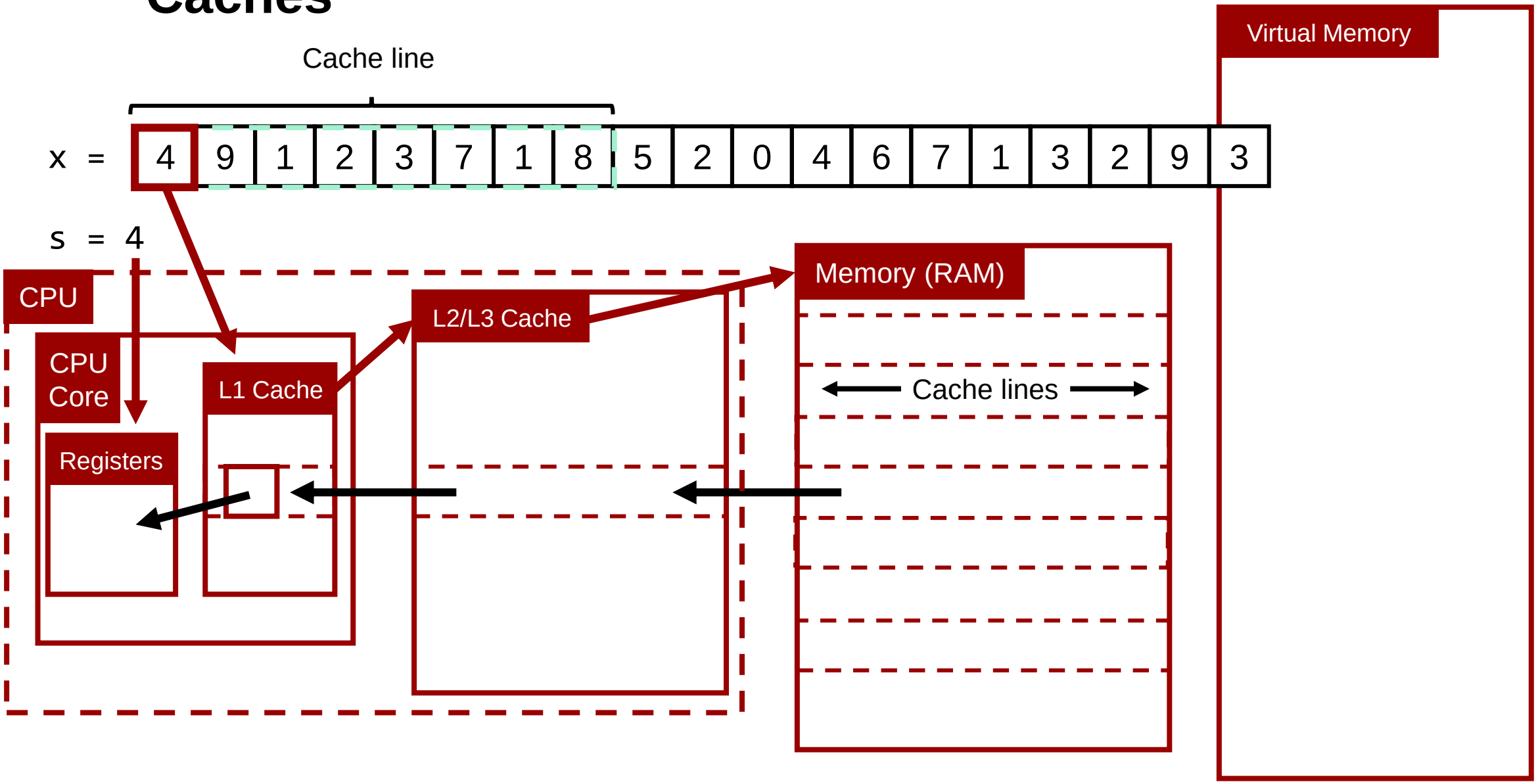
Caches


Virtual Memory
x = 4 9 1 2 3 7 1 8 5 2 0 4 6 7 1 3 2 9 3
s = 4
CPU
CPU Core
Registers
L1 Cache
L2/L3 Cache
Memory (RAM)

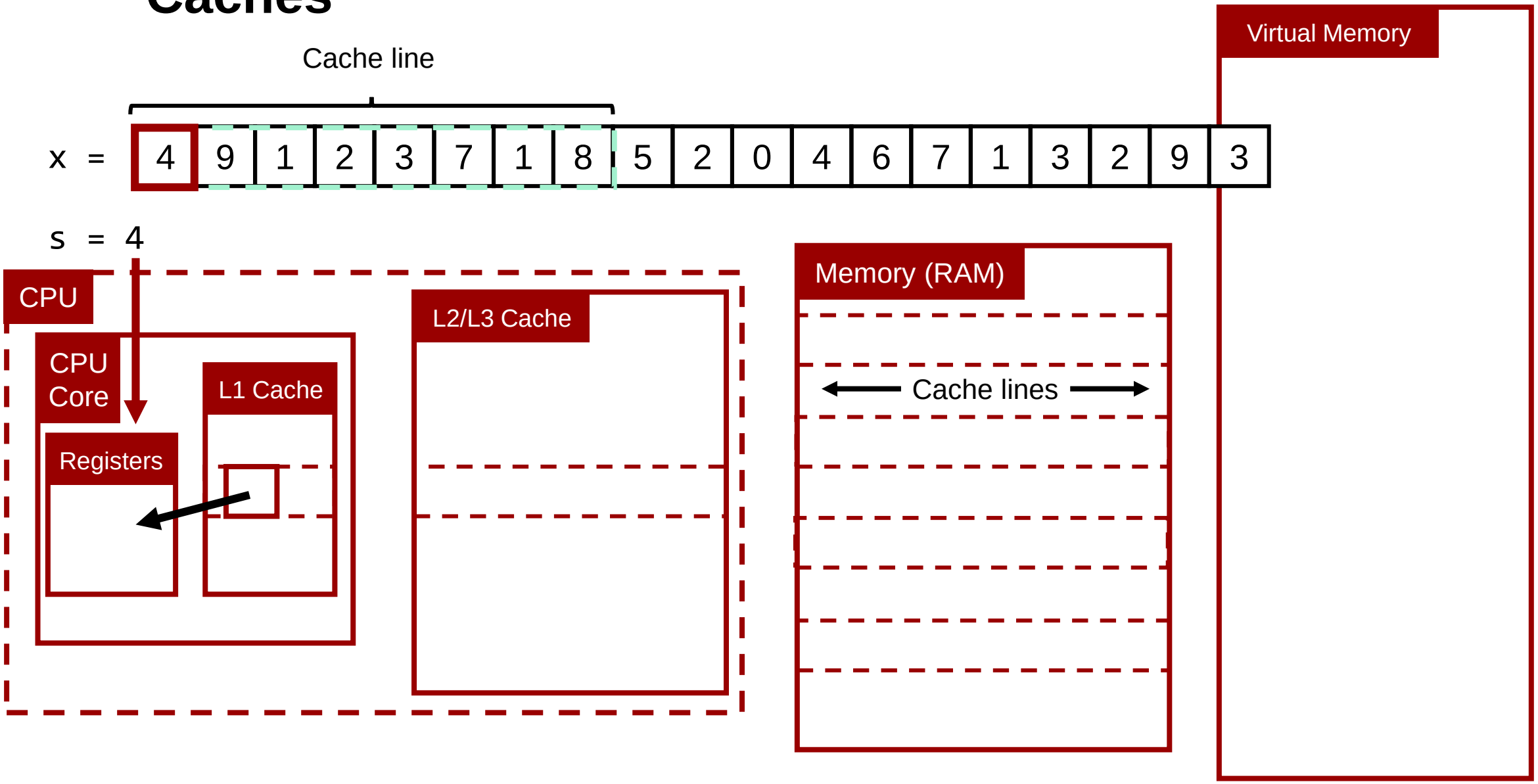
Caches


Virtual Memory
x =
4
9
1
2
3
7
1
8
5
2
0
4
6
7
1
3
2
9
3
s = 4
CPU
CPU Core
Registers
L1 Cache
L2/L3 Cache
Memory (RAM)
Cache lines

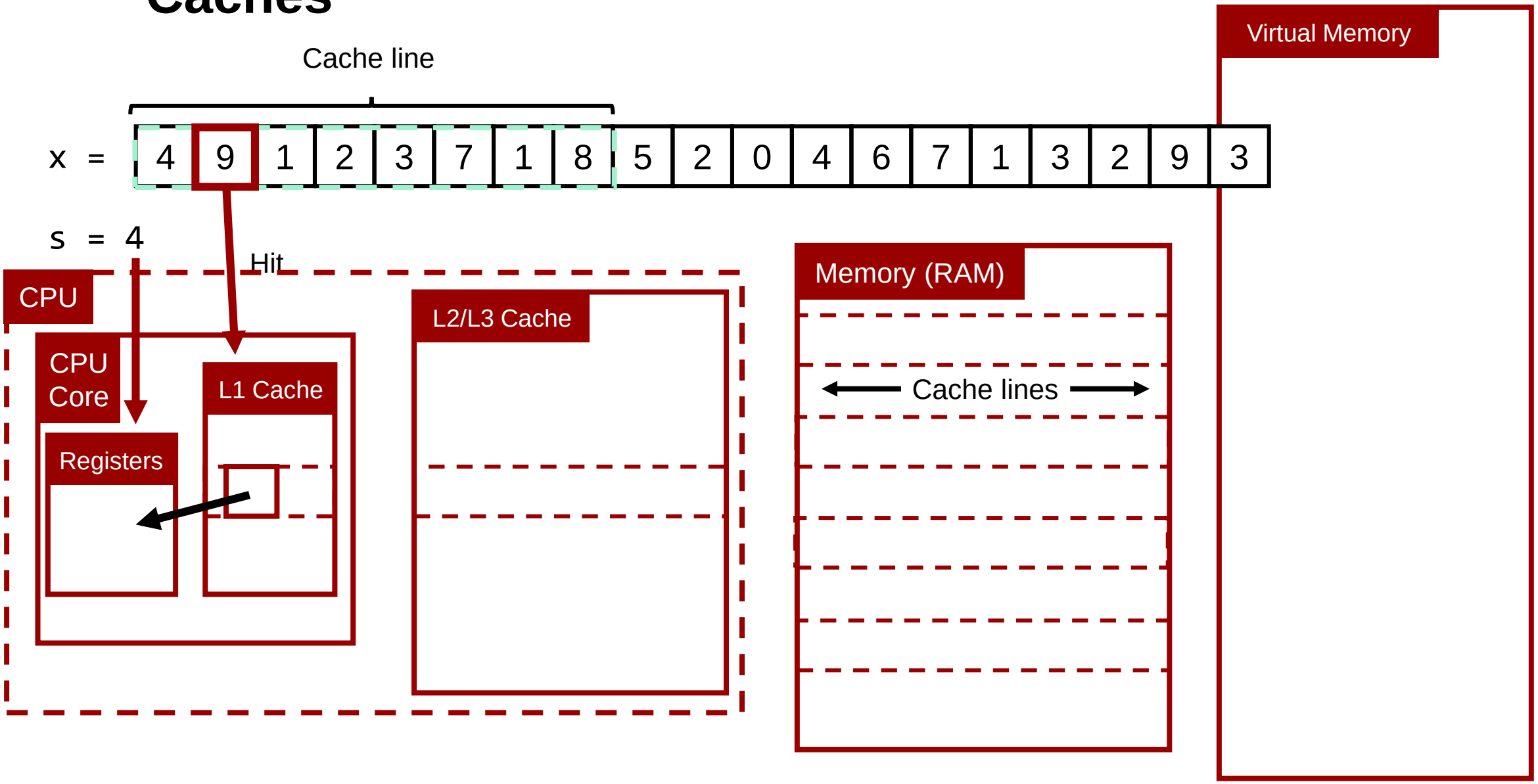
Caches


Virtual Memory
Cache line
x =
4 9 1 2 3 7 1 8 5 2 0 4 6 7 1 3 2 9 3
s = 4
CPU
CPU Core
Registers
L1 Cache
L2/L3 Cache
Memory (RAM)
Cache lines

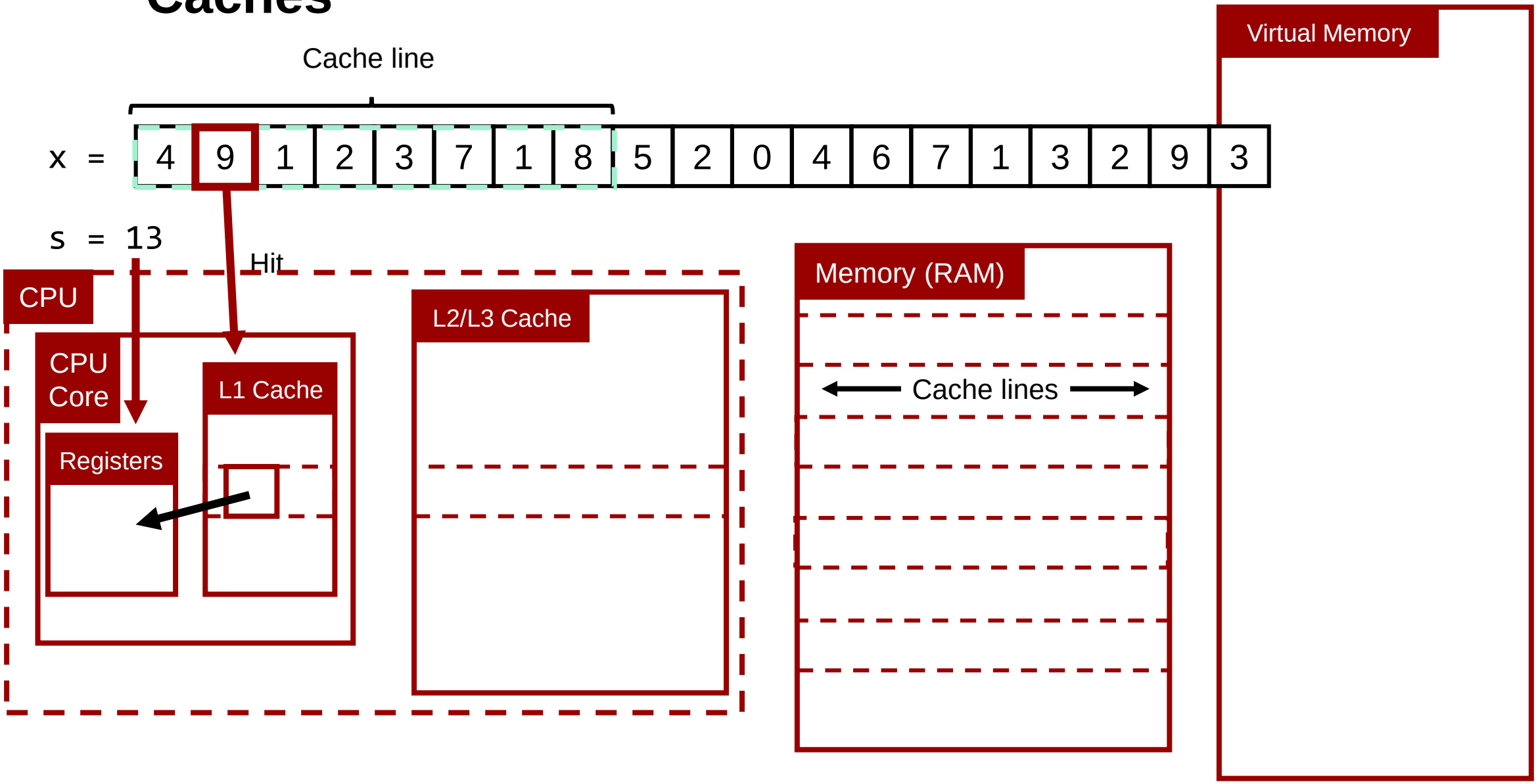
Caches


Virtual Memory
Cache line
x =
4 9 1 2 3 7 1 8 5 2 0 4 6 7 1 3 2 9 3
s = 4
CPU
CPU Core
Registers
L1 Cache
L2/L3 Cache
Memory (RAM)
Cache lines

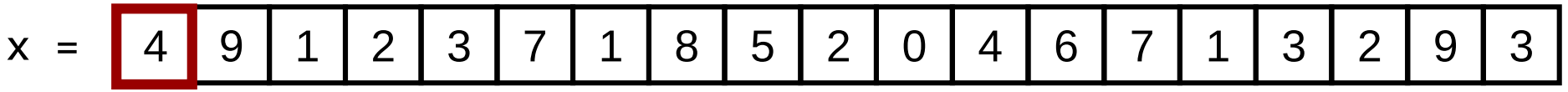
Caches


Cache line
x =
4 9 1 2 3 7 1 8 5 2 0 4 6 7 1 3 2 9 3
s = 4
Hit
CPU
CPU Core
Registers
L1 Cache
L2/L3 Cache
Memory (RAM)
Cache lines
Virtual Memory

Caches

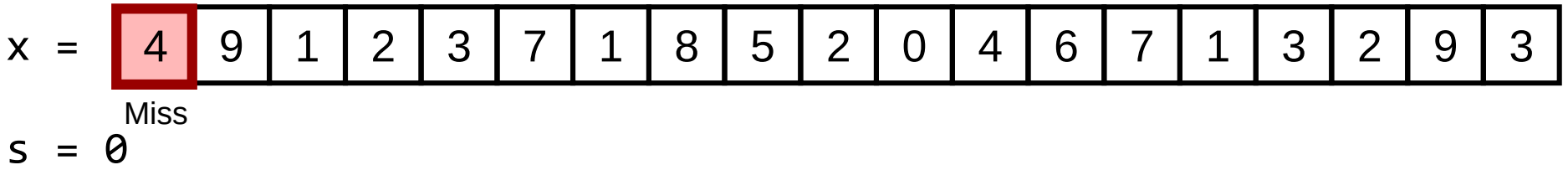

Virtual Memory
Cache line
x =
4 9 1 2 3 7 1 8 5 2 0 4 6 7 1 3 2 9 3
s = 13
Hit
CPU
CPU Core
L1 Cache
Registers
L2/L3 Cache
Memory (RAM)
Cache lines

Caches

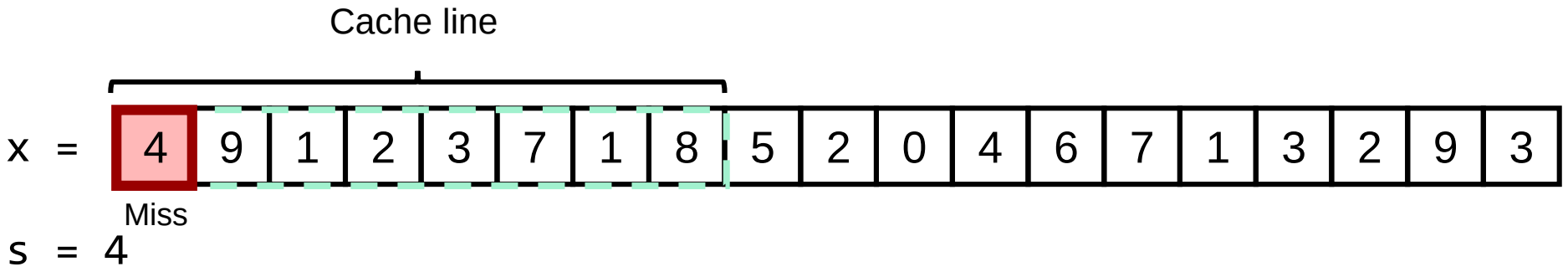



```
s = 0
```

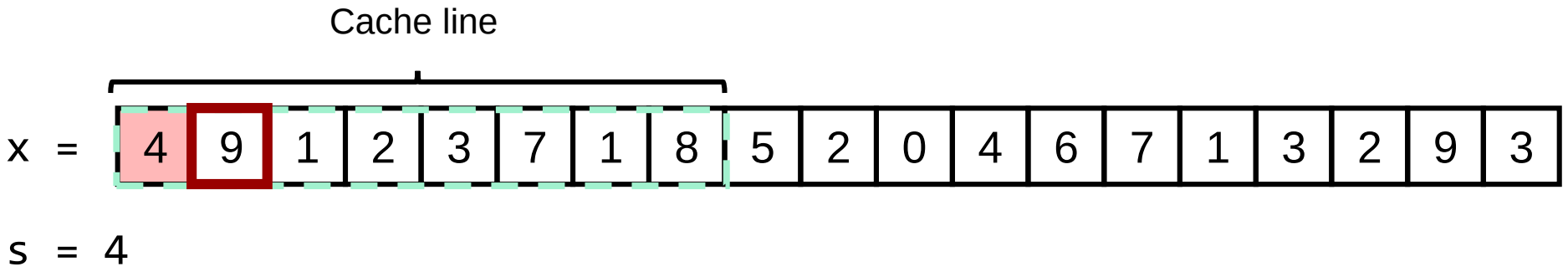
Caches



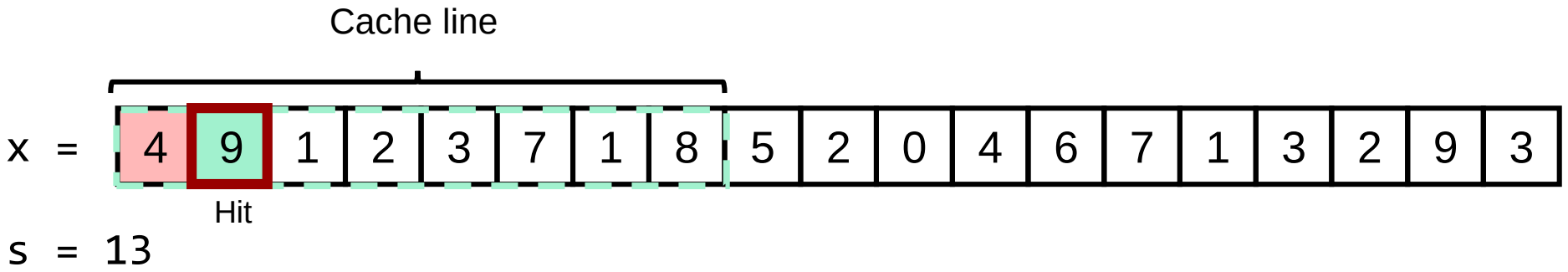
Caches



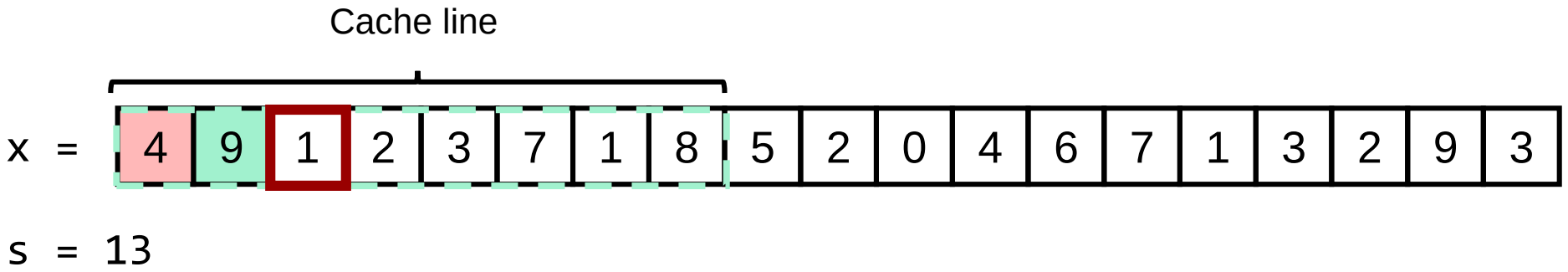
Caches



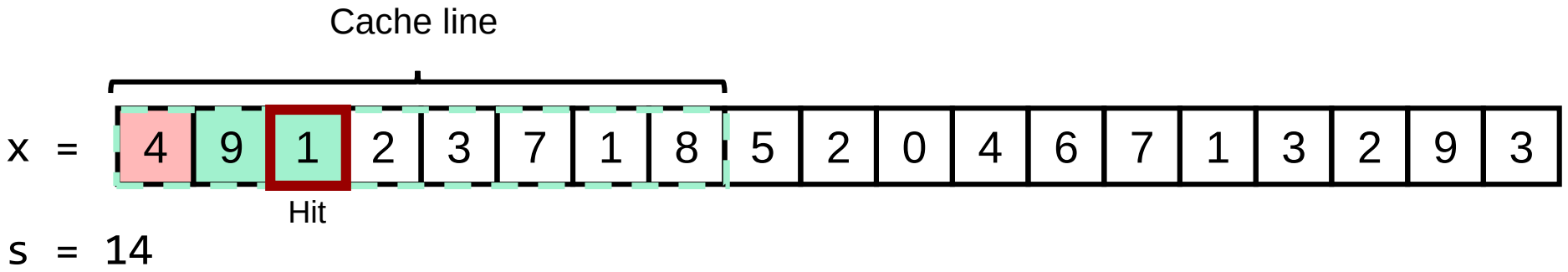
Caches



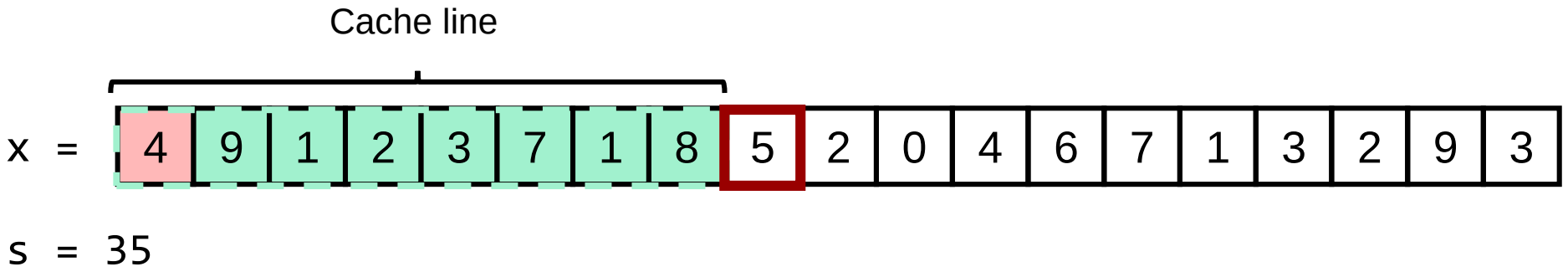
Caches



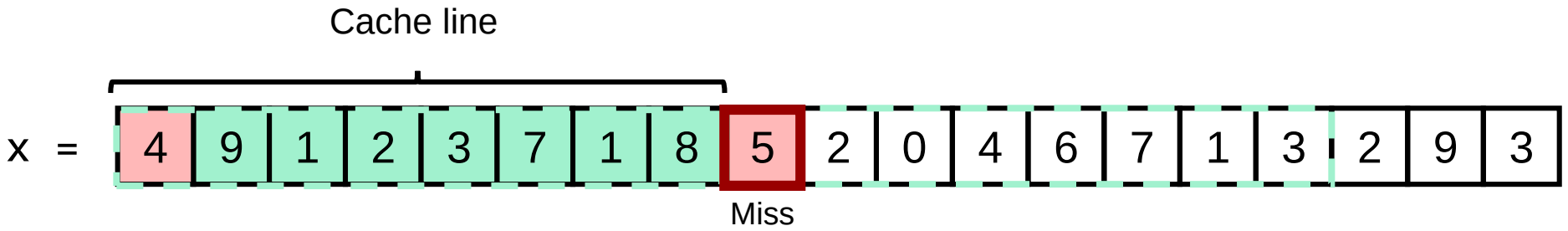
Caches



Caches

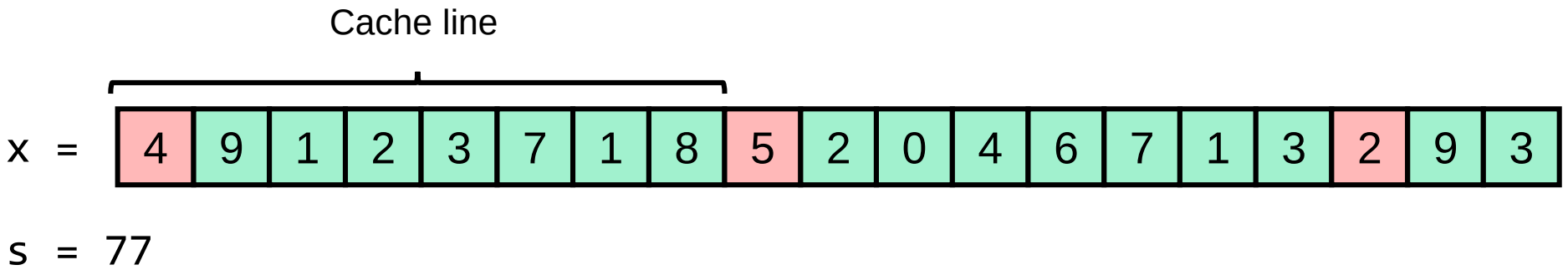


Caches



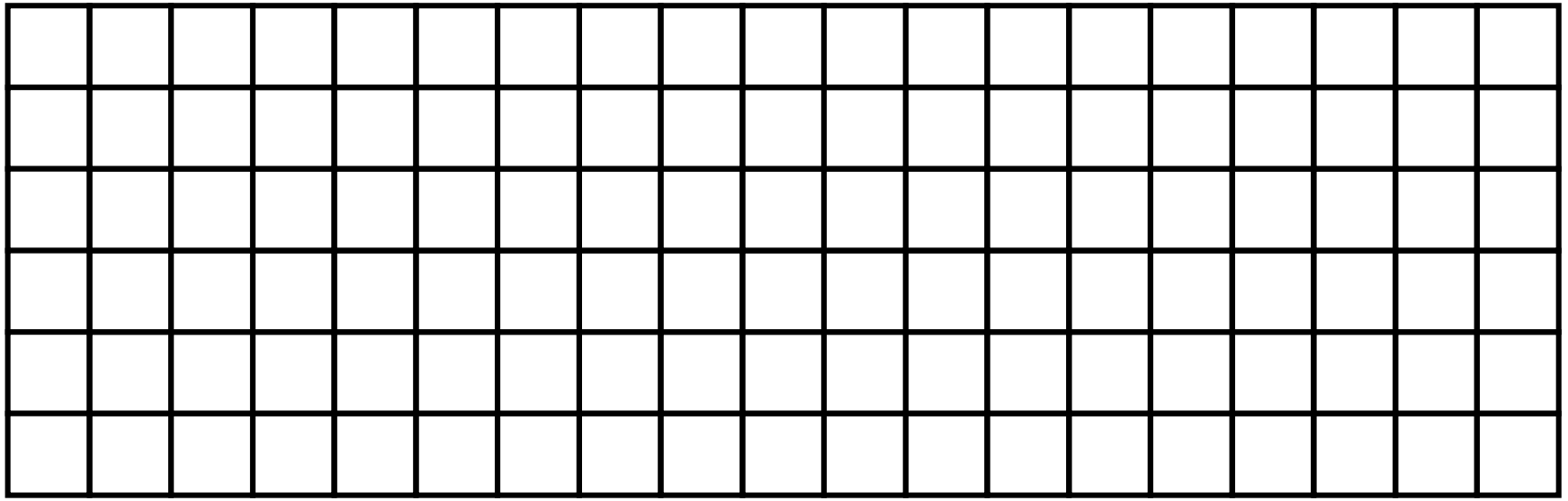

```
s = 40
```

Caches



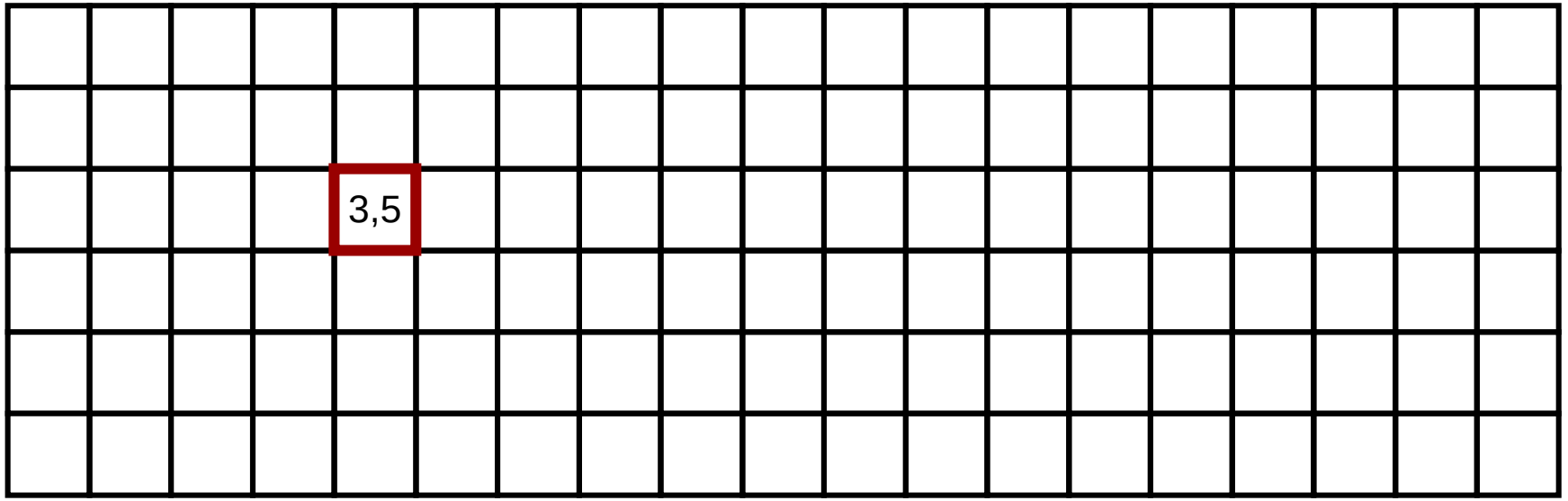
Caches: multidimensional arrays

x =



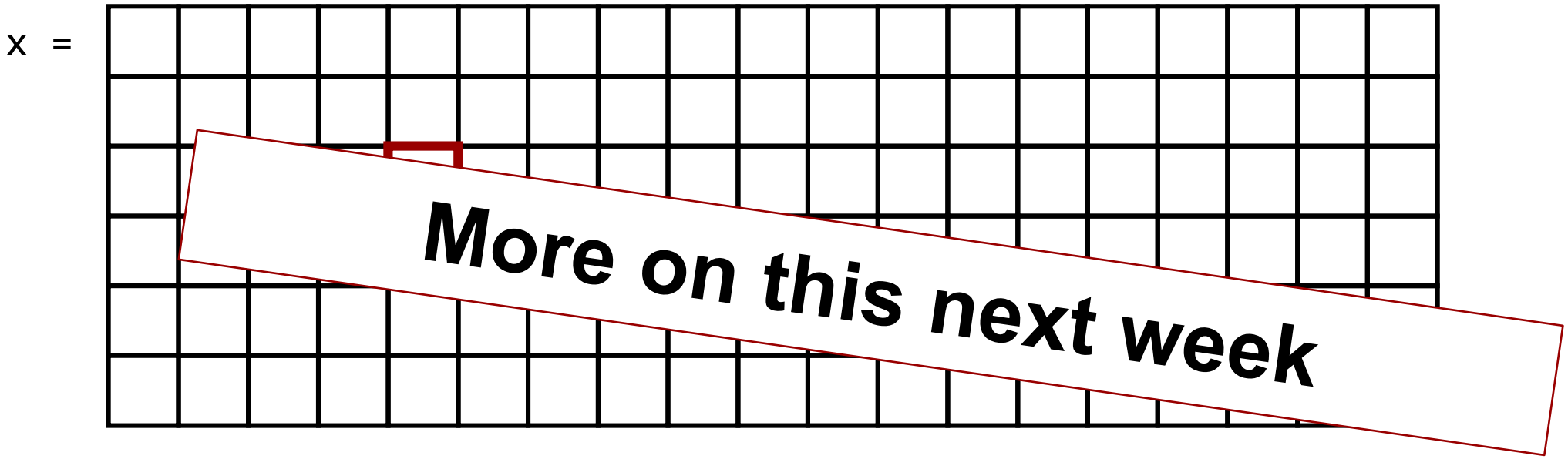
Caches: multidimensional arrays

x =



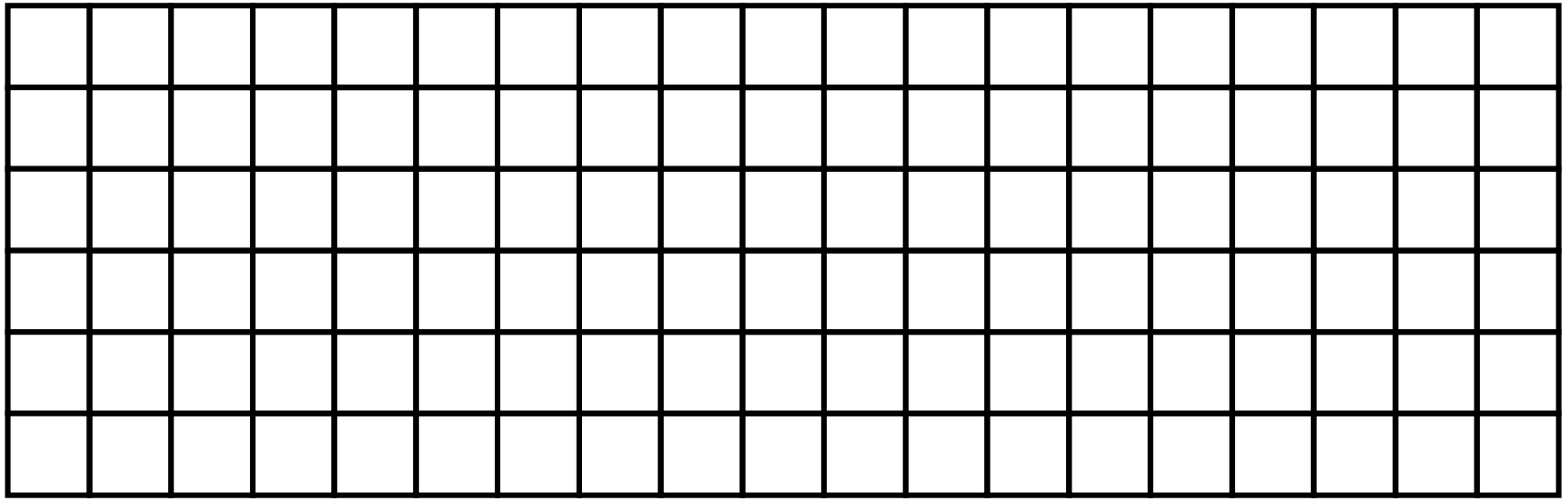
x =

Caches: multidimensional arrays



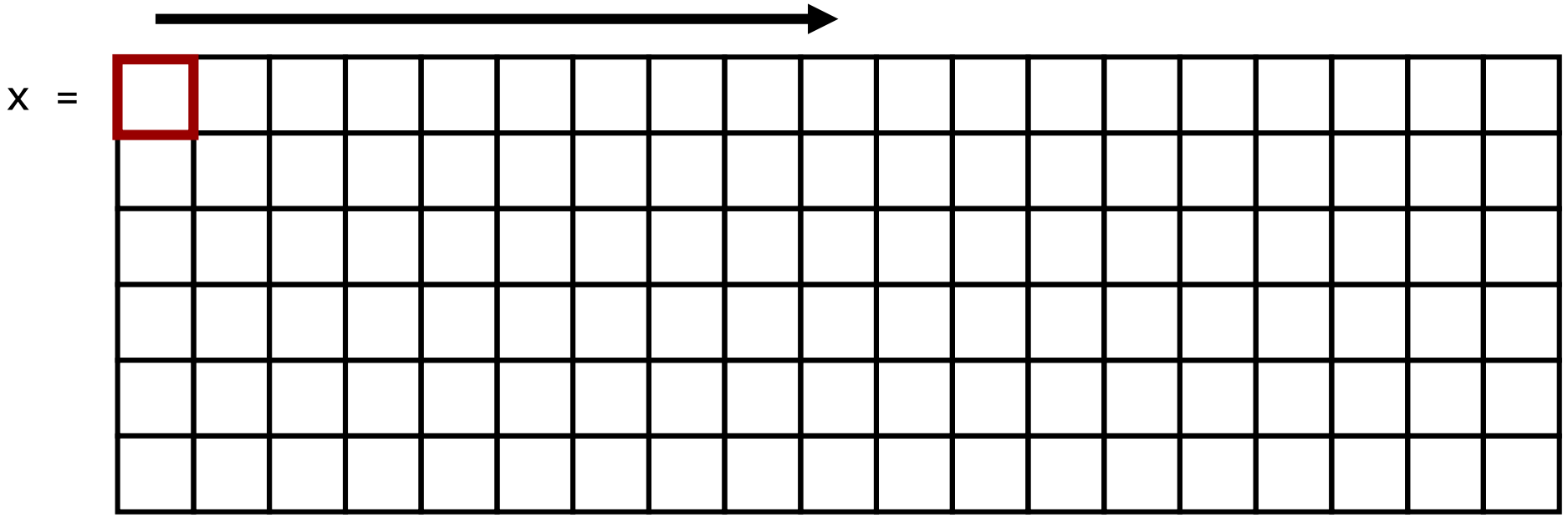
Caches: multidimensional arrays

x =



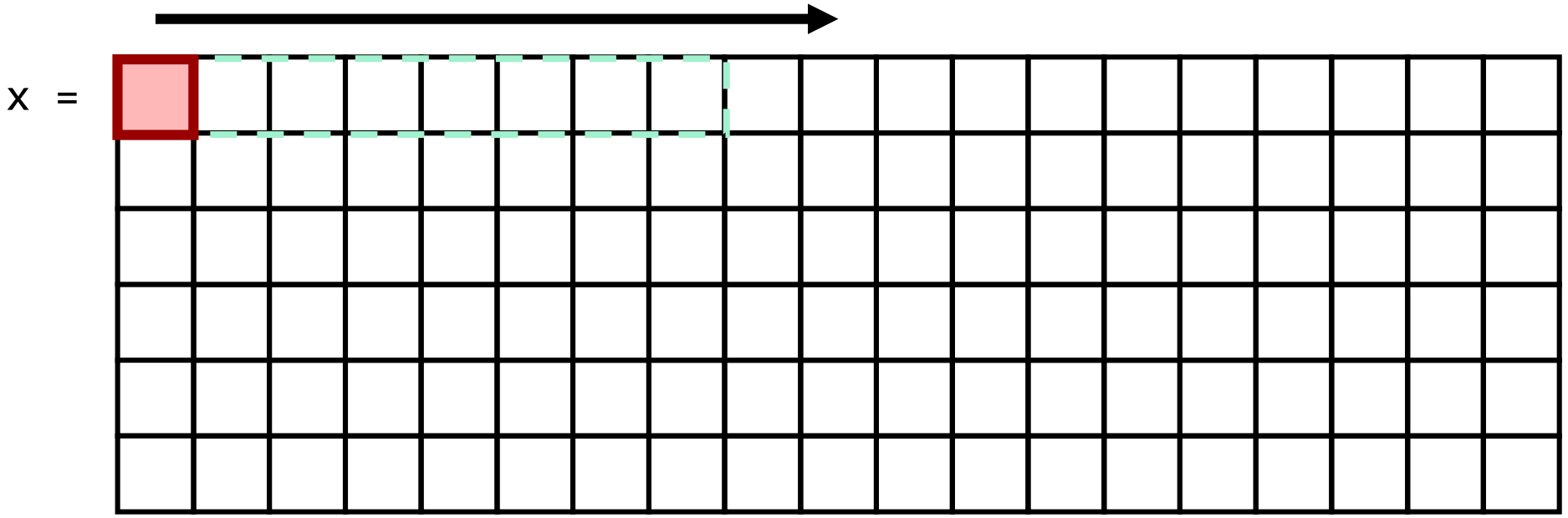
```
s = sum(x)
```

Caches: multidimensional arrays



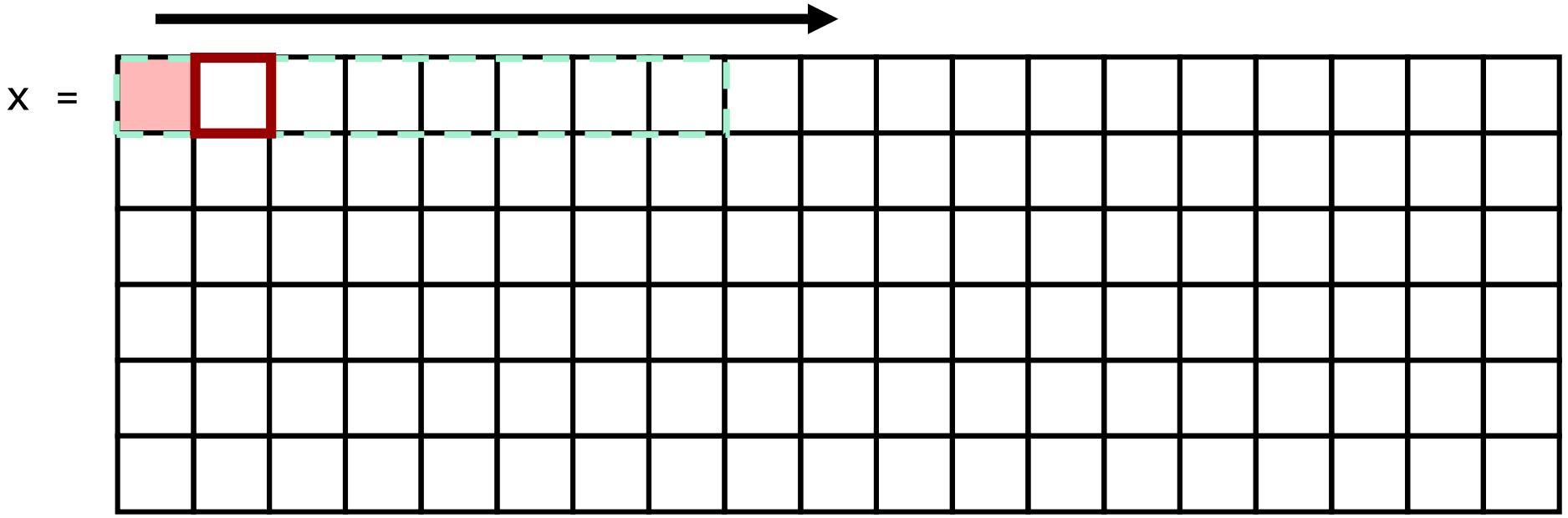

```
s = 0
for r in range(rows):
    for c in range(columns):
        s += x[r, c]
```

Caches: multidimensional arrays



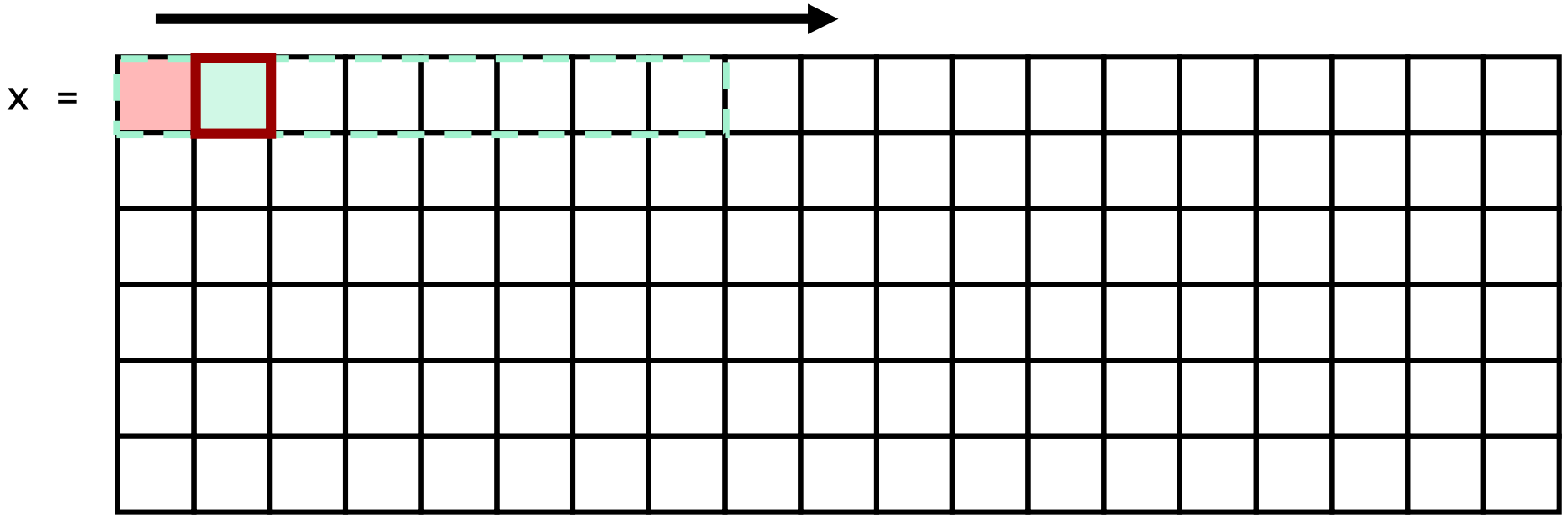

```
s = 0
for r in range(rows):
    for c in range(columns):
        s += x[r, c]
```

Caches: multidimensional arrays



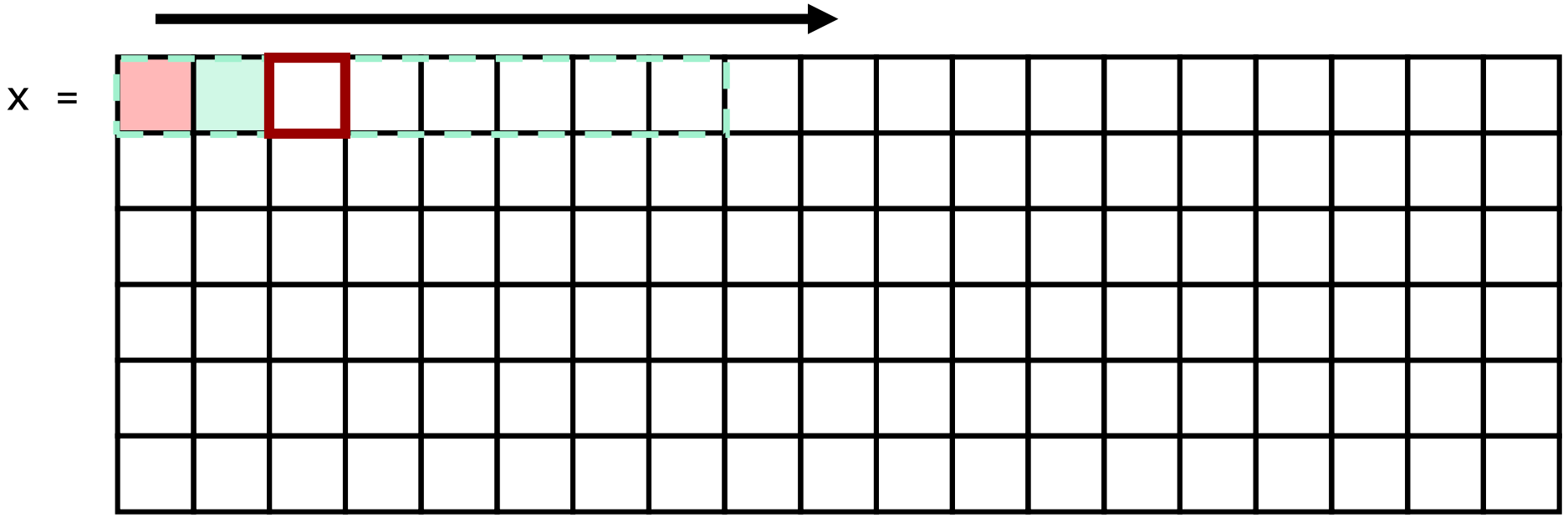

```
s = 0
for r in range(rows):
    for c in range(columns):
        s += x[r, c]
```

Caches: multidimensional arrays



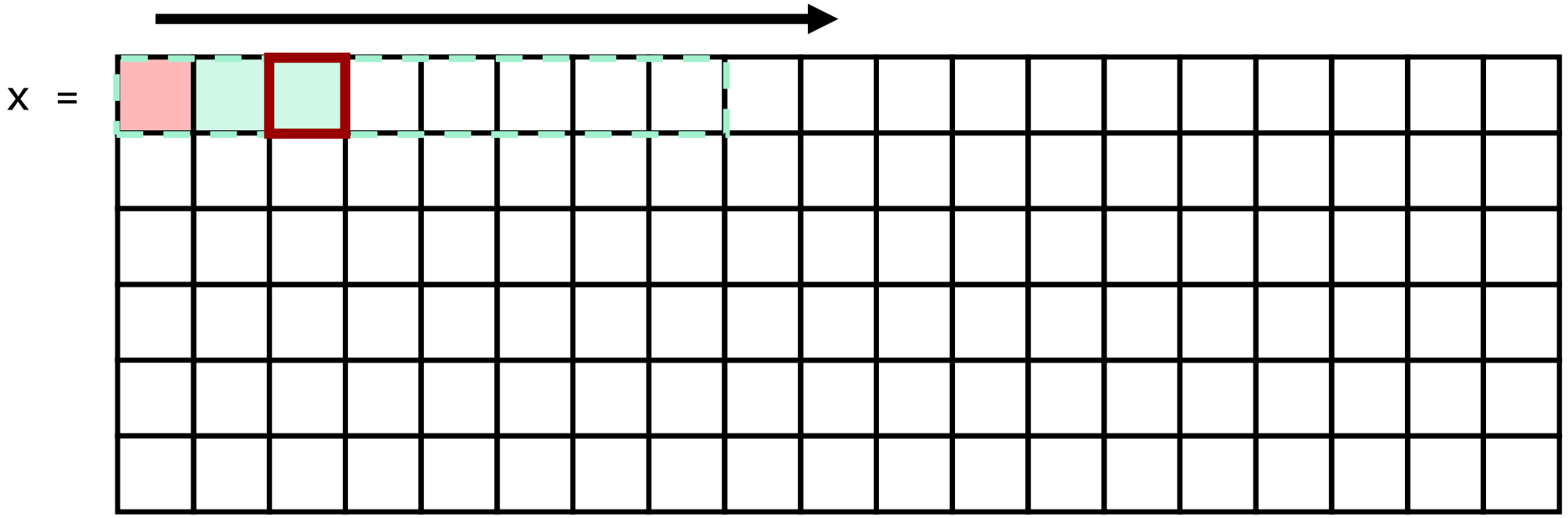

```
s = 0
for r in range(rows):
    for c in range(columns):
        s += x[r, c]
```

Caches: multidimensional arrays



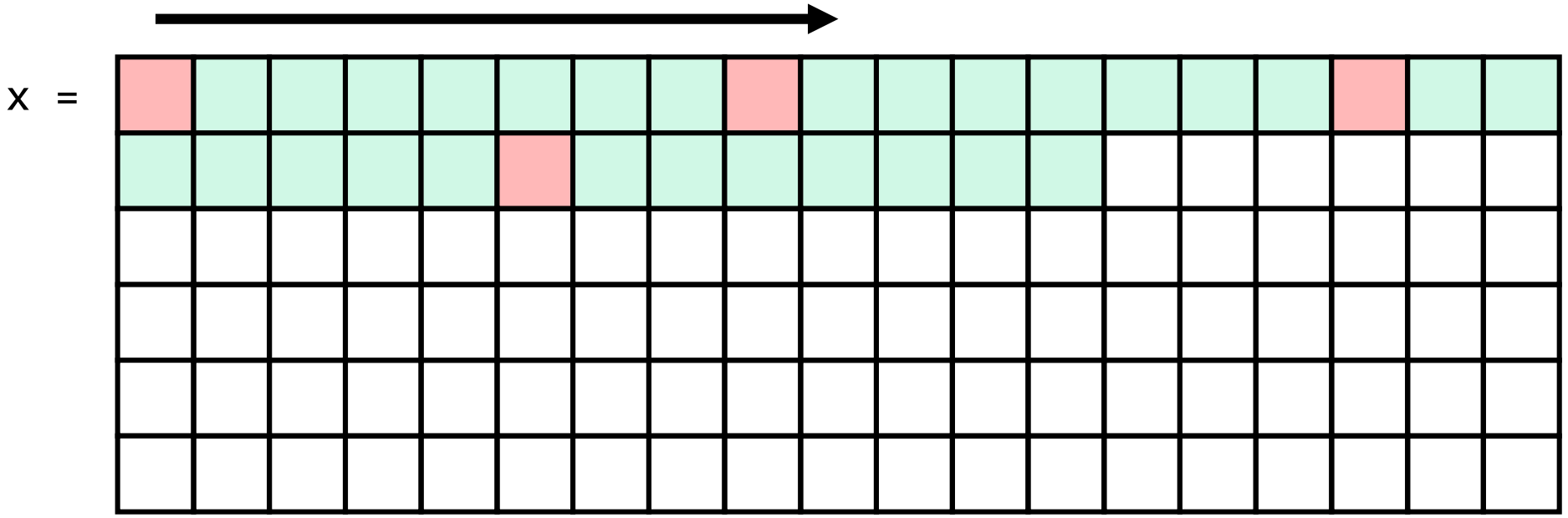

```
s = 0
for r in range(rows):
    for c in range(columns):
        s += x[r, c]
```

Caches: multidimensional arrays




```
s = 0
for r in range(rows):
    for c in range(columns):
        s += x[r, c]
```

Caches: multidimensional arrays

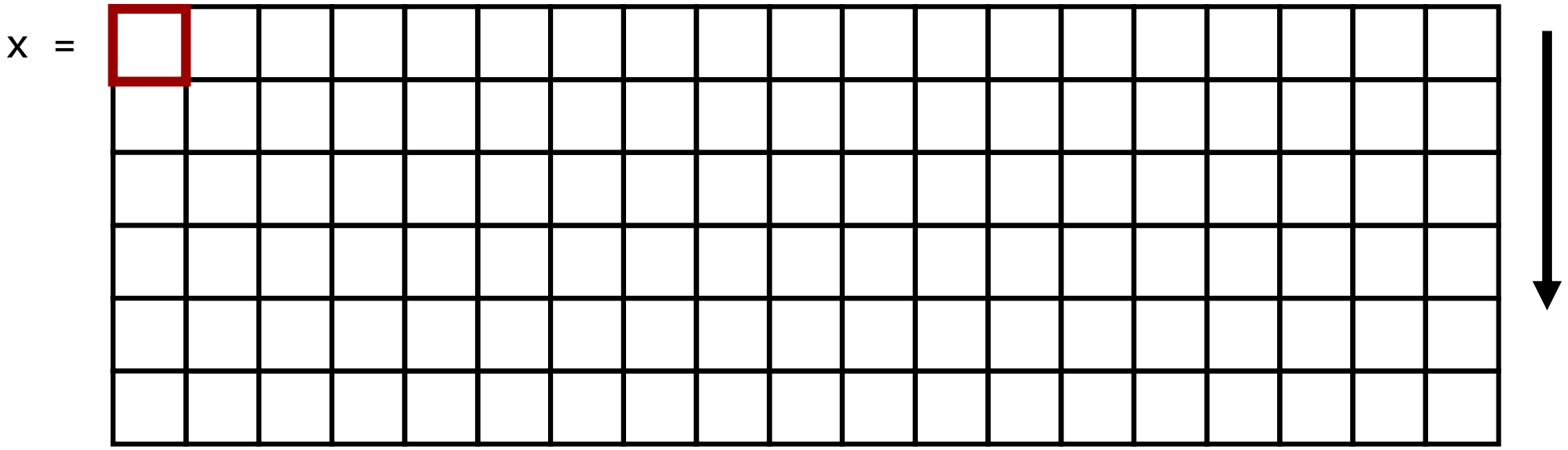



```
s = 0
for r in range(rows):
    for c in range(columns):
        s += x[r, c]
```

Access pattern matches data layout

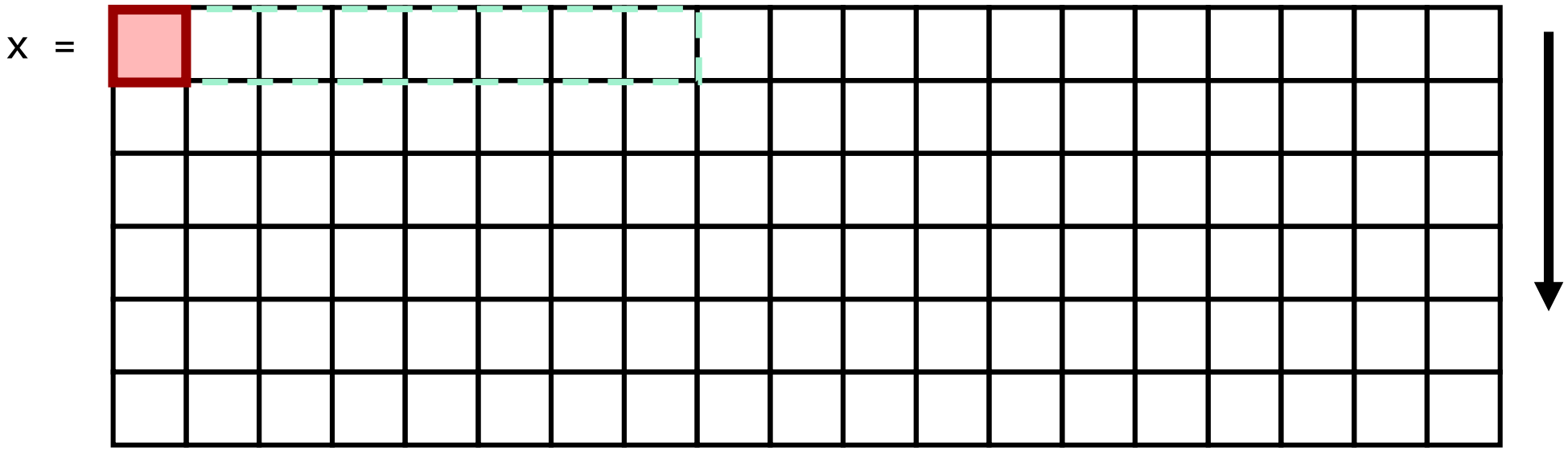
Miss rate: 12.5% (1/8)

Caches: multidimensional arrays




```
s = 0
for c in range(columns):
    for r in range(rows):
        s += x[r, c]
```

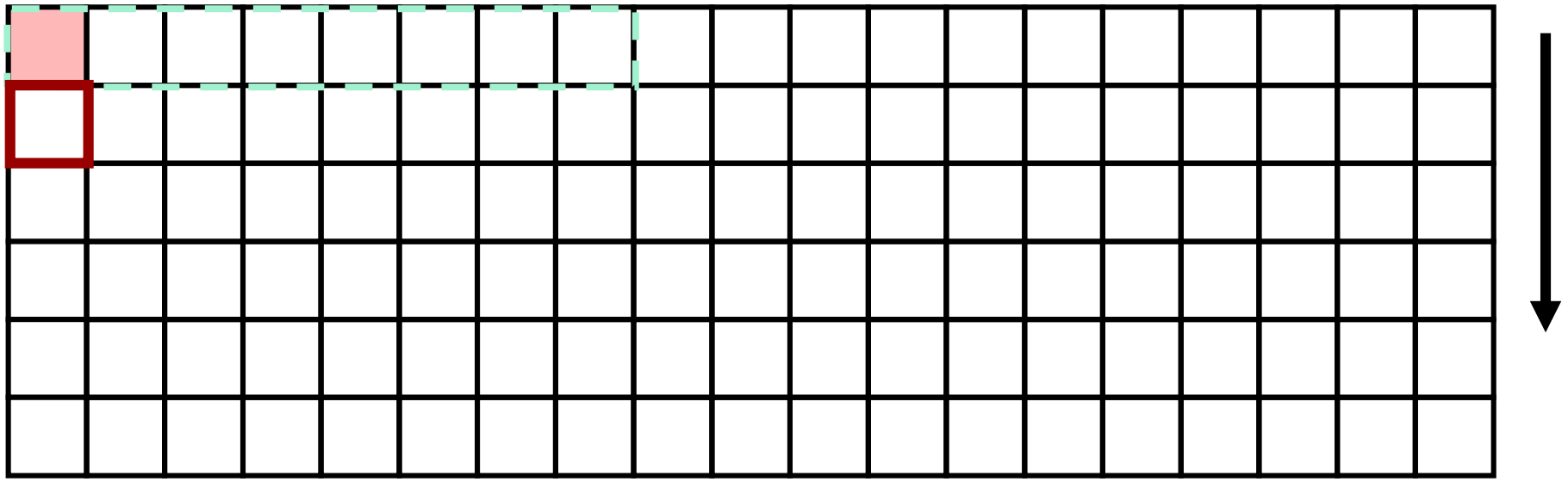
Caches: multidimensional arrays




```
s = 0
for c in range(columns):
    for r in range(rows):
        s += x[r, c]
```

Caches: multidimensional arrays

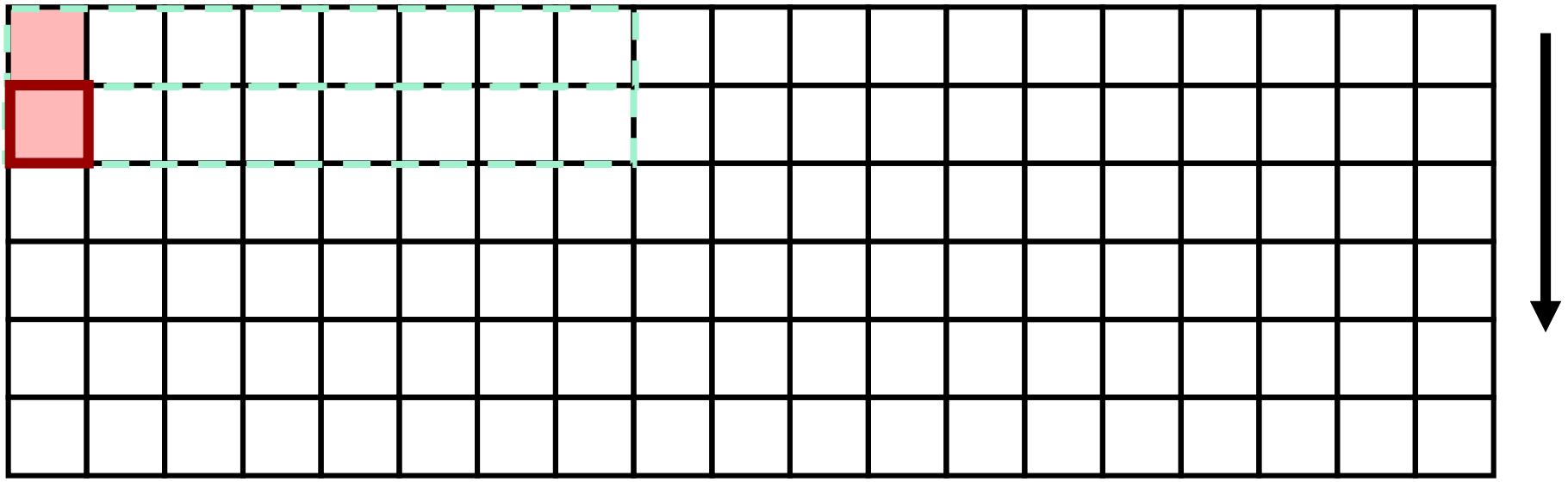
x =



```
s = 0
for c in range(columns):
    for r in range(rows):
        s += x[r, c]
```

Caches: multidimensional arrays

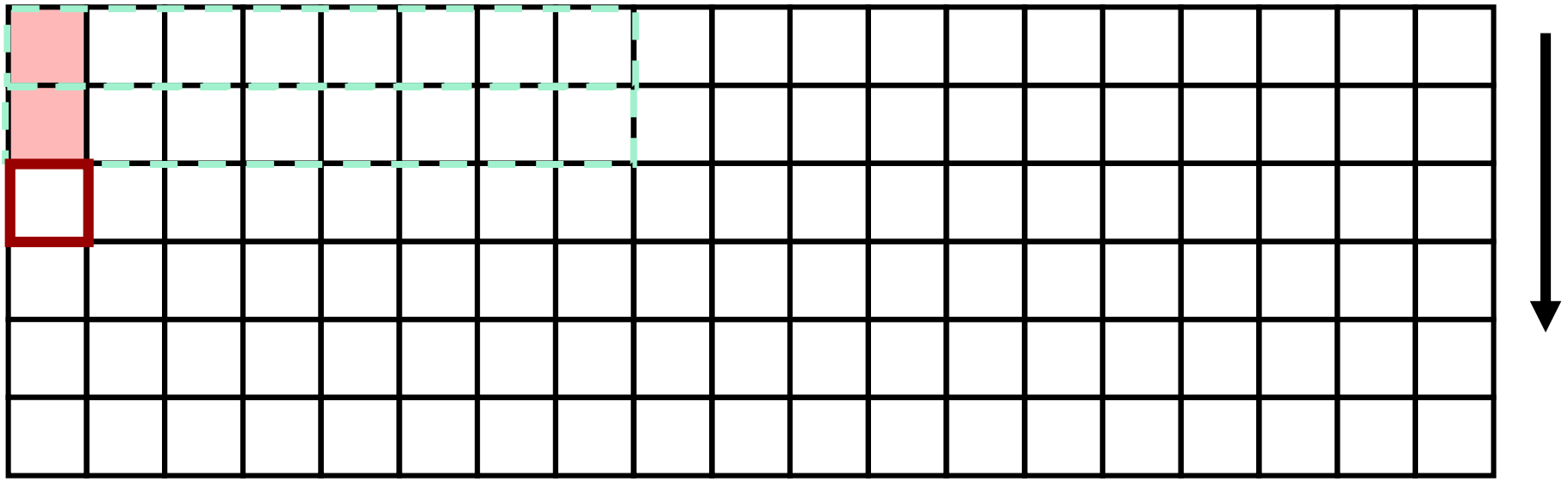
x =



```
s = 0
for c in range(columns):
    for r in range(rows):
        s += x[r, c]
```

Caches: multidimensional arrays

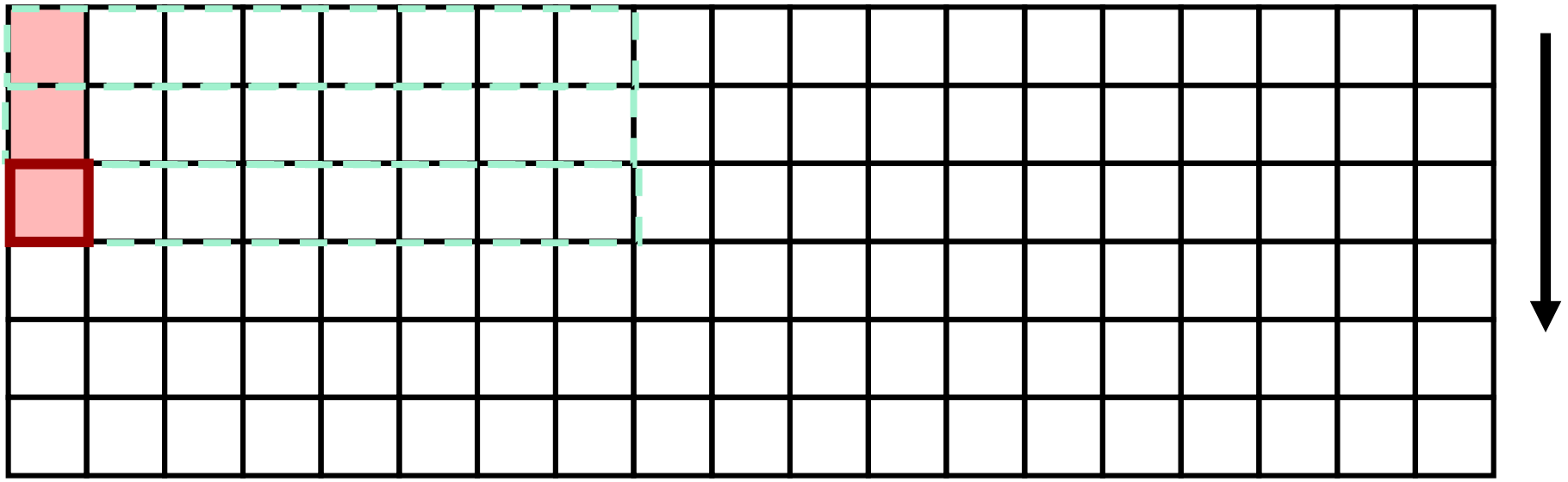
x =



```
s = 0
for c in range(columns):
    for r in range(rows):
        s += x[r, c]
```

Caches: multidimensional arrays

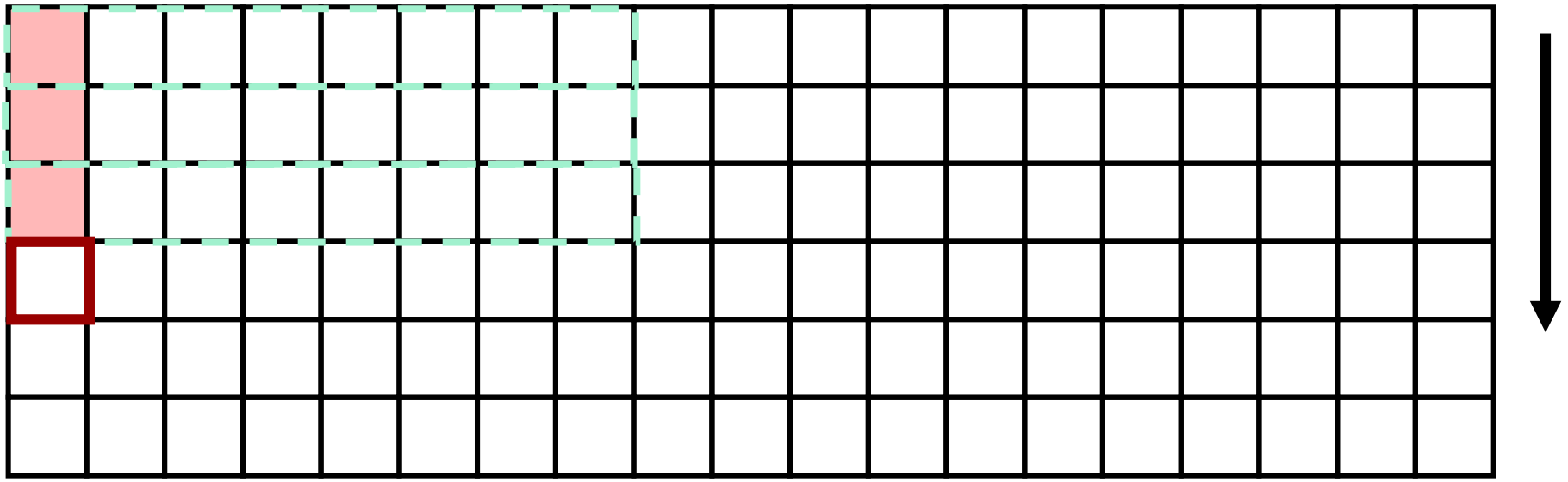
x =



```
s = 0
for c in range(columns):
    for r in range(rows):
        s += x[r, c]
```

Caches: multidimensional arrays

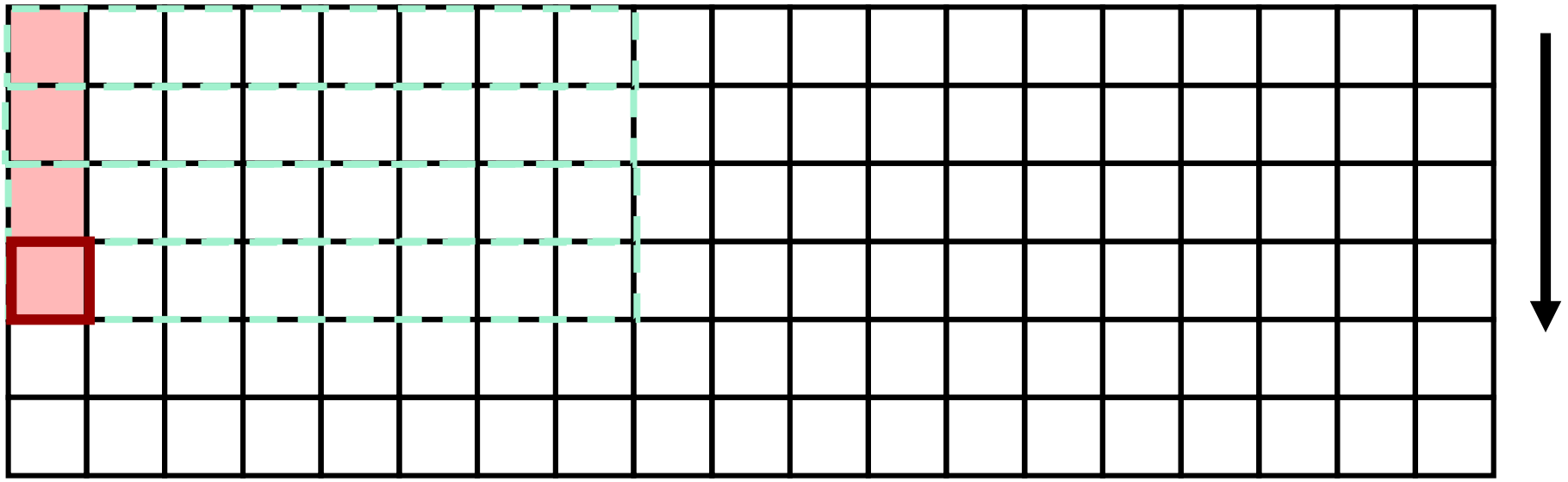
x =



```
s = 0
for c in range(columns):
    for r in range(rows):
        s += x[r, c]
```

Caches: multidimensional arrays

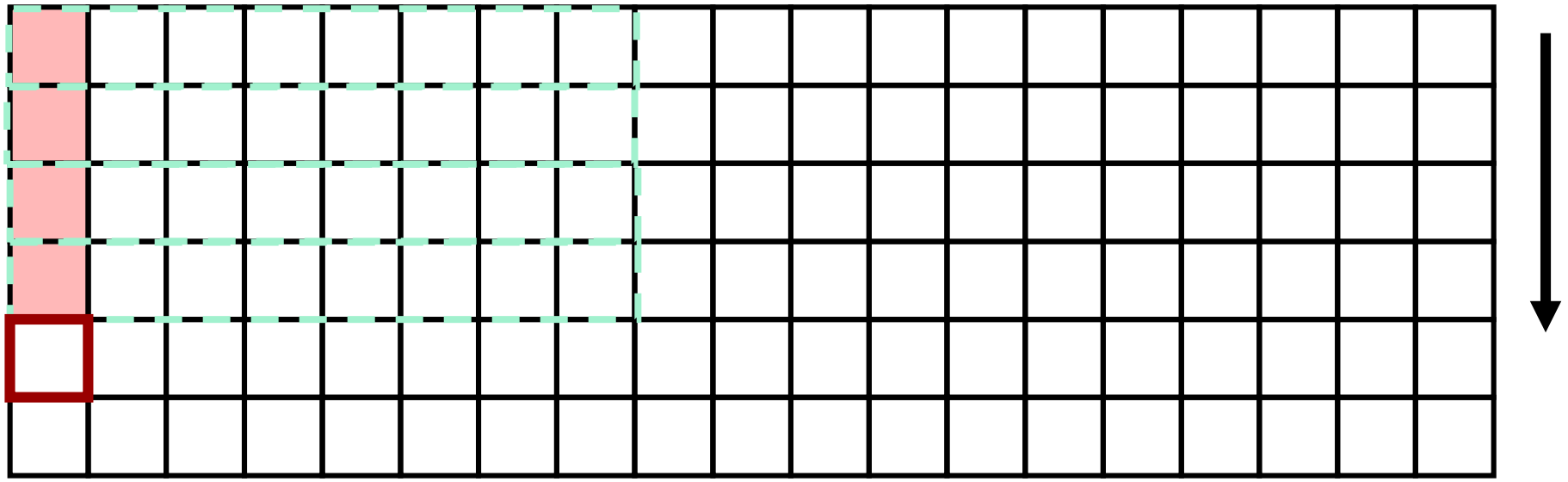
x =



```
s = 0
for c in range(columns):
    for r in range(rows):
        s += x[r, c]
```

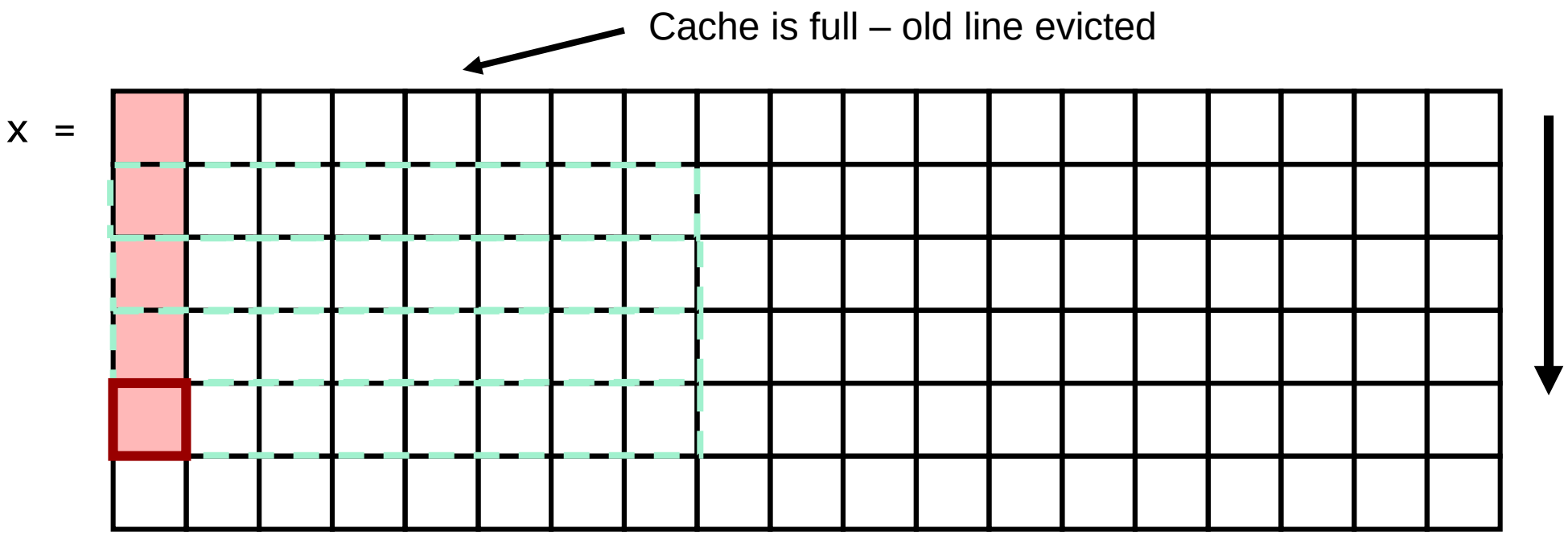
Caches: multidimensional arrays

x =



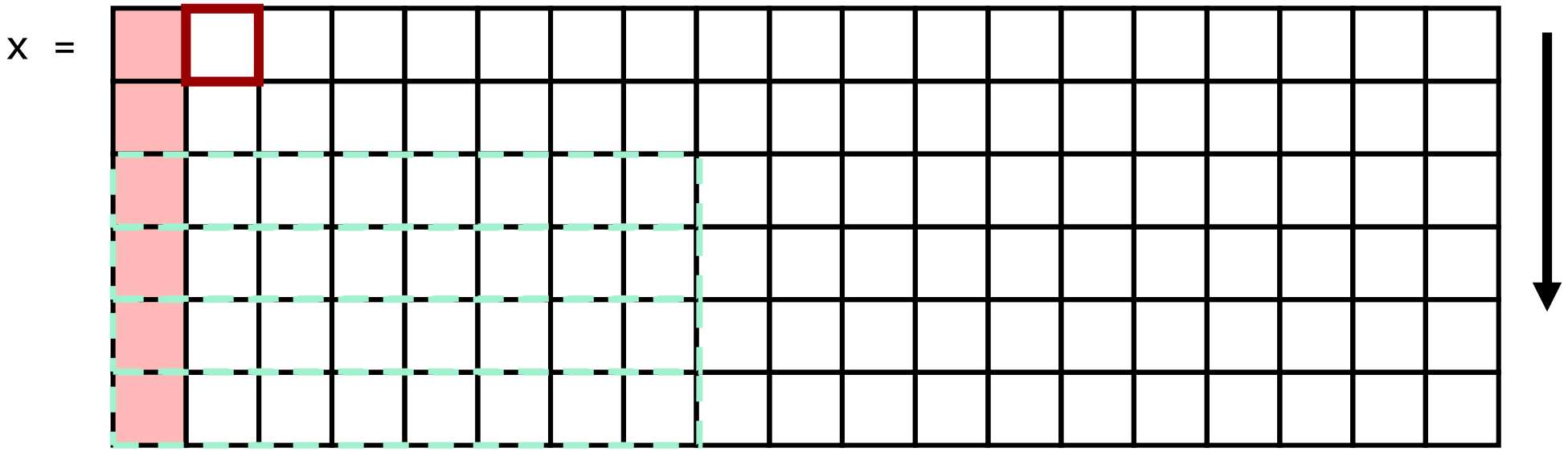
```
s = 0
for c in range(columns):
    for r in range(rows):
        s += x[r, c]
```

Caches: multidimensional arrays



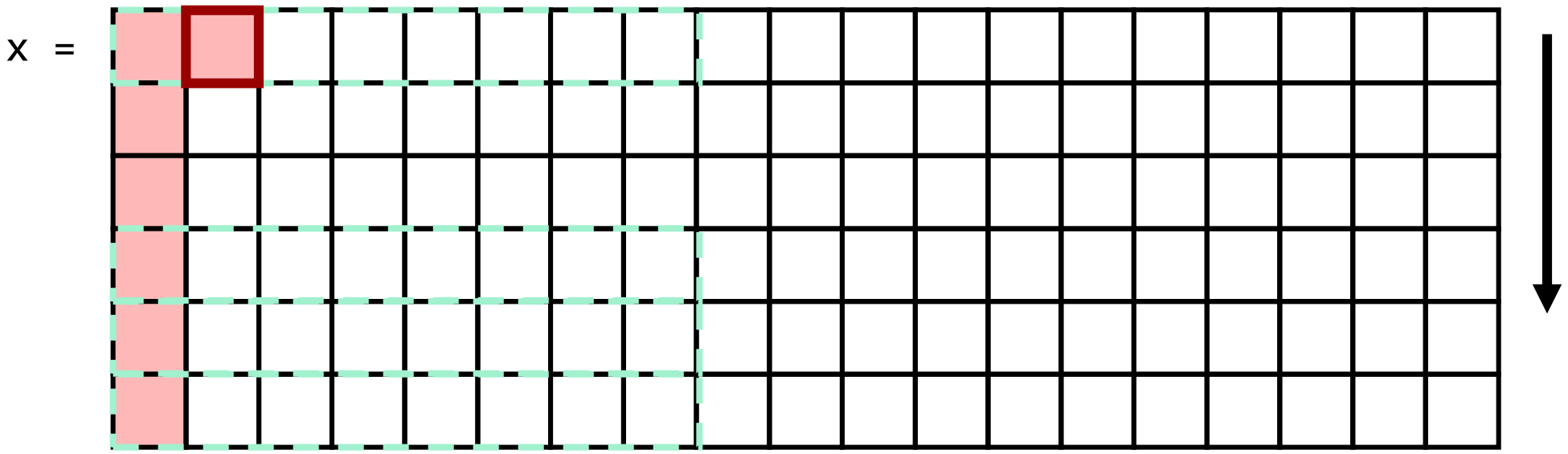

```
s = 0
for c in range(columns):
    for r in range(rows):
        s += x[r, c]
```

Caches: multidimensional arrays




```
s = 0
for c in range(columns):
    for r in range(rows):
        s += x[r, c]
```

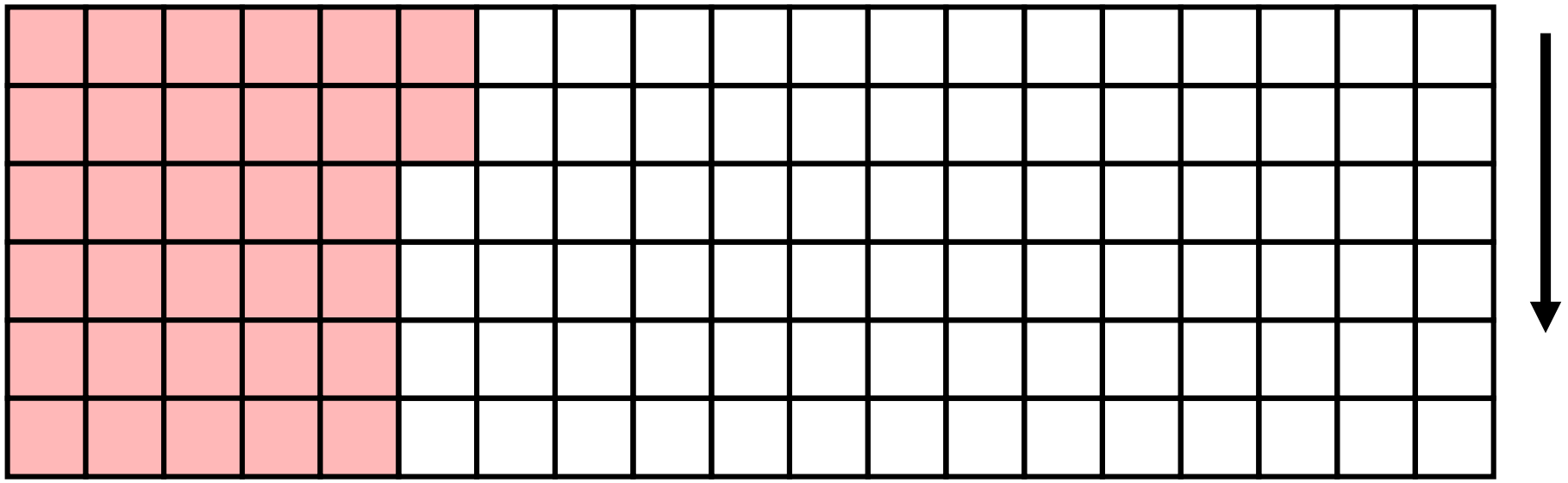
Caches: multidimensional arrays




```
s = 0
for c in range(columns):
    for r in range(rows):
        s += x[r, c]
```

Caches: multidimensional arrays

x =



```
s = 0
for c in range(columns):
    for r in range(rows):
        s += x[r, c]
```

Access pattern **does not** match data layout

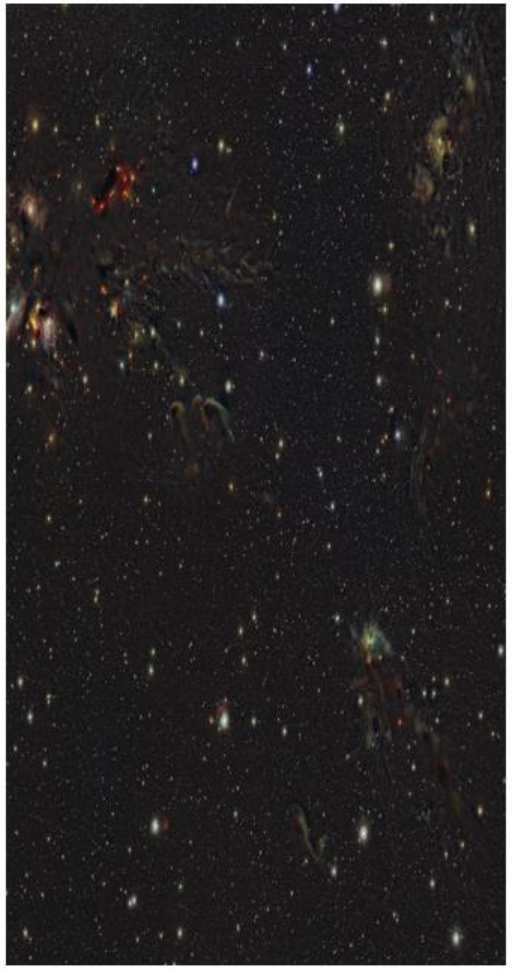
Miss rate: 100%

Caches: multidimensional arrays

x =

s =

for



```
im = load_image('stars.png')
n, m, c = im.shape

small = np.empty((n // 2, m, c))
t = time()
for i in range(n // 2):
    small[i,:] = im[i*2,:] + im[i*2+1,:]
t = time() - t
print(t, 'sec')
```

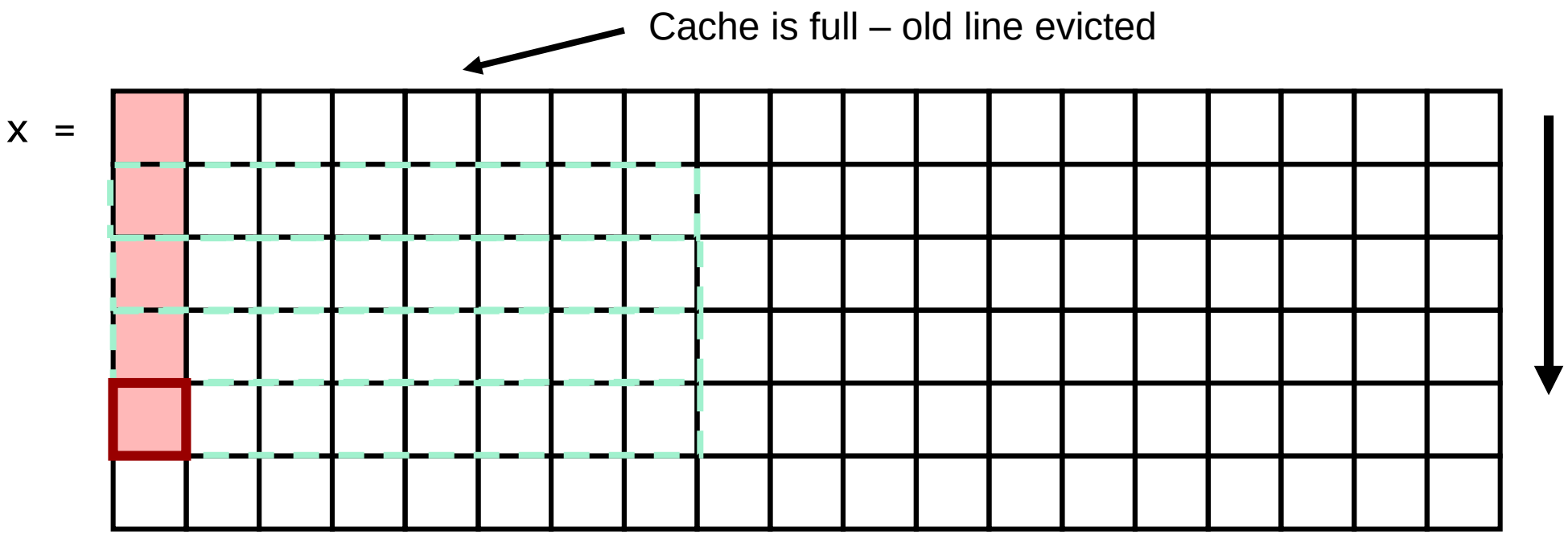
```
1.6689006000001427 sec
```

```
im = load_image('stars.png')
n, m, c = im.shape

small = np.empty((n, m // 2, c))
t = time()
for i in range(m // 2):
    small[:,i] = im[:,i*2] + im[:,i*2+1]
t = time() - t
print(t, 'sec')
```

```
20.149567999999817 sec
```

14.02.2024 DTU Compute Python and High-Performance Computing 10

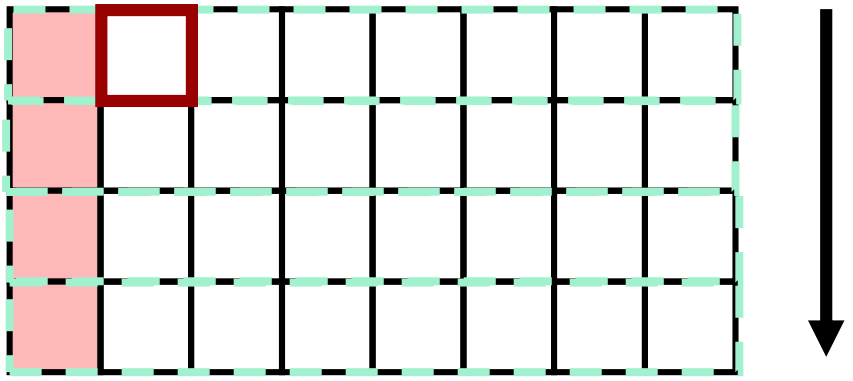
Caveat: small arrays




```
s = 0
for c in range(columns):
    for r in range(rows):
        s += x[r, c]
```

Caveat: small arrays

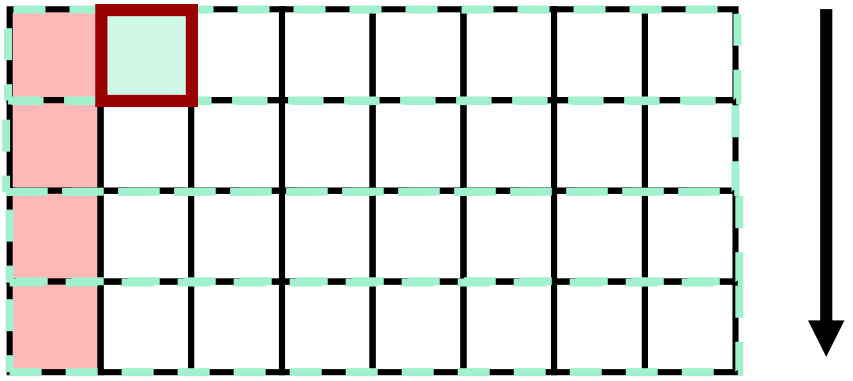
x =



```
s = 0
for c in range(columns):
    for r in range(rows):
        s += x[r, c]
```

Caveat: small arrays

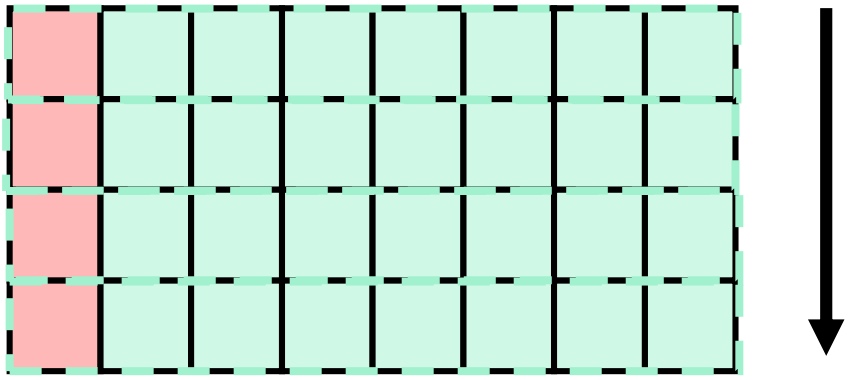
x =



```
s = 0
for c in range(columns):
    for r in range(rows):
        s += x[r, c]
```

Caveat: small arrays

x =

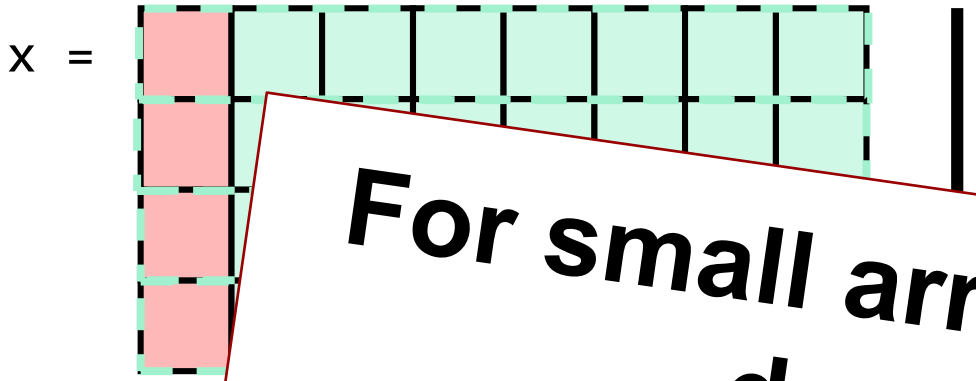


```
s = 0
for c in range(columns):
    for r in range(rows):
        s += x[r, c]
```

Access pattern **does not** match data layout

Miss rate: 12.5% (1/8)

Caveat: small arrays



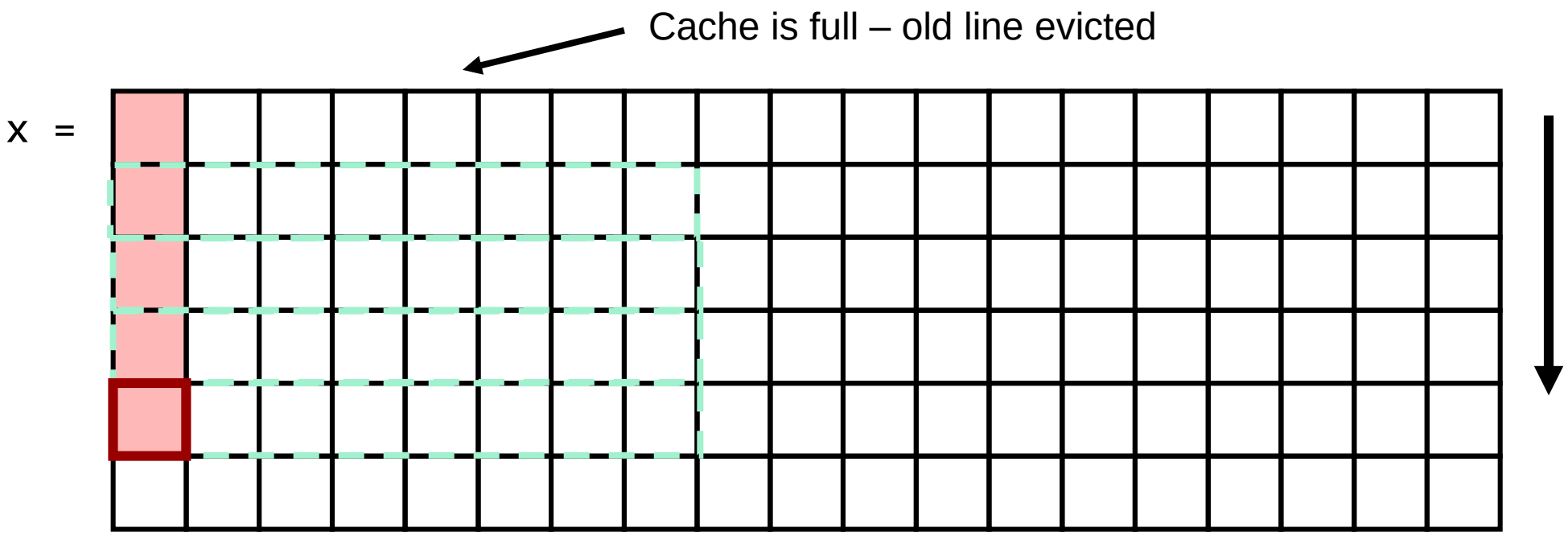
For small arrays access pattern does *not* matter

```
s = 0
for c in range(columns):
    for r in range(rows):
        s += x[r, c]
```

Access pattern **does not** match data layout

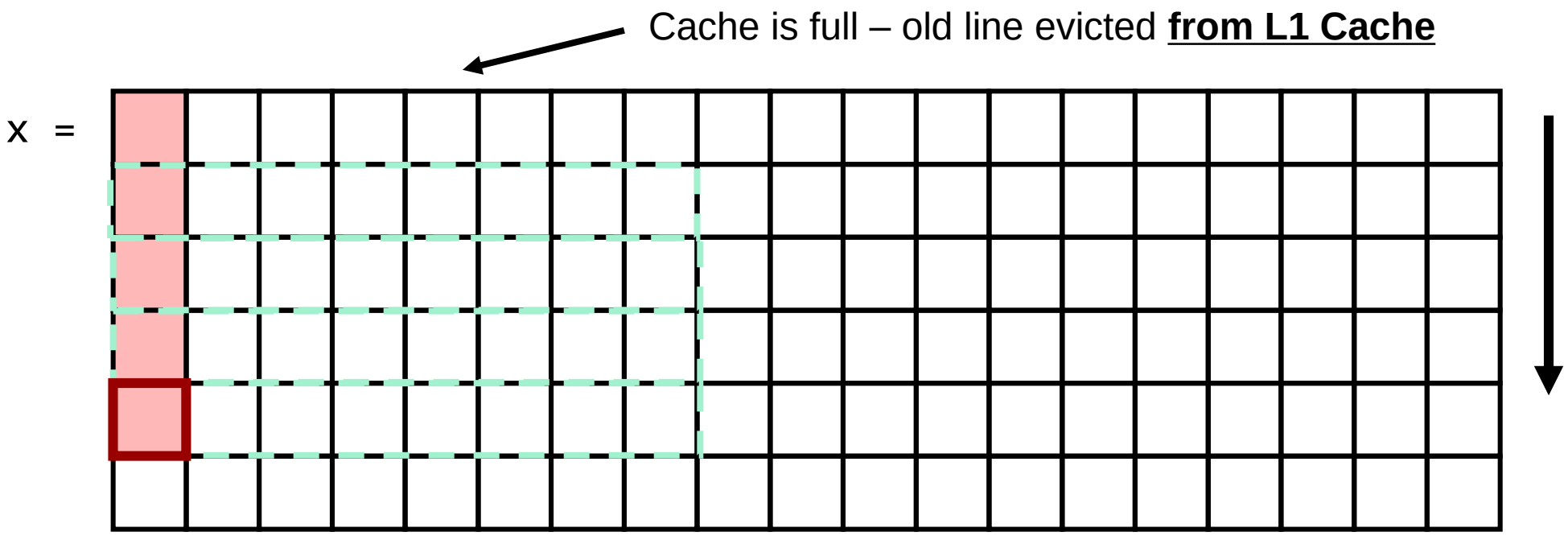
Miss rate: 12.5% (1/8)

Caveat: multiple cache levels



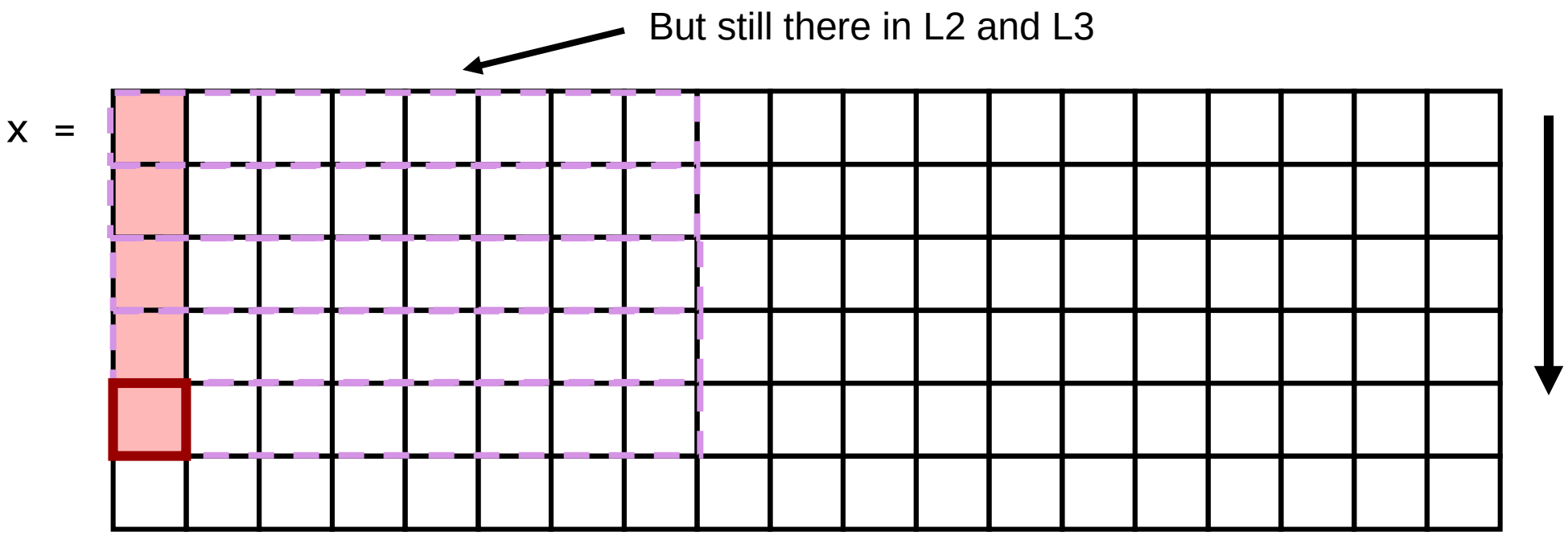

```
s = 0
for c in range(columns):
    for r in range(rows):
        s += x[r, c]
```

Caveat: multiple cache levels



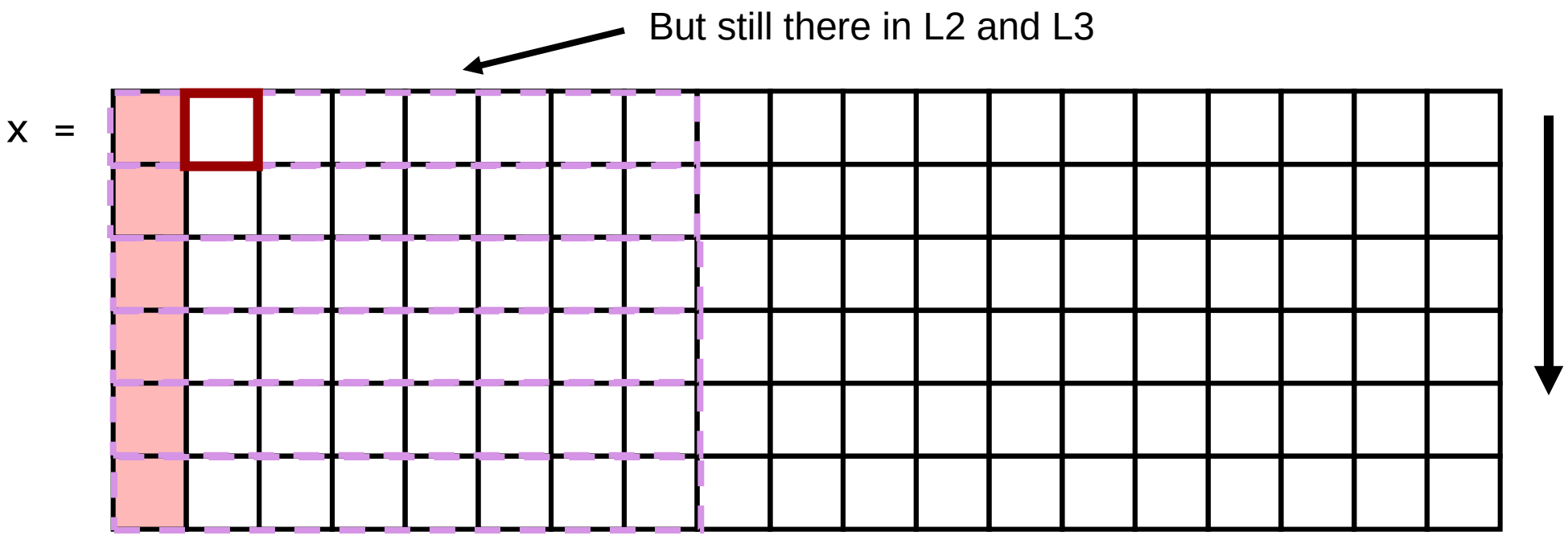

```
s = 0
for c in range(columns):
    for r in range(rows):
        s += x[r, c]
```

Caveat: multiple cache levels



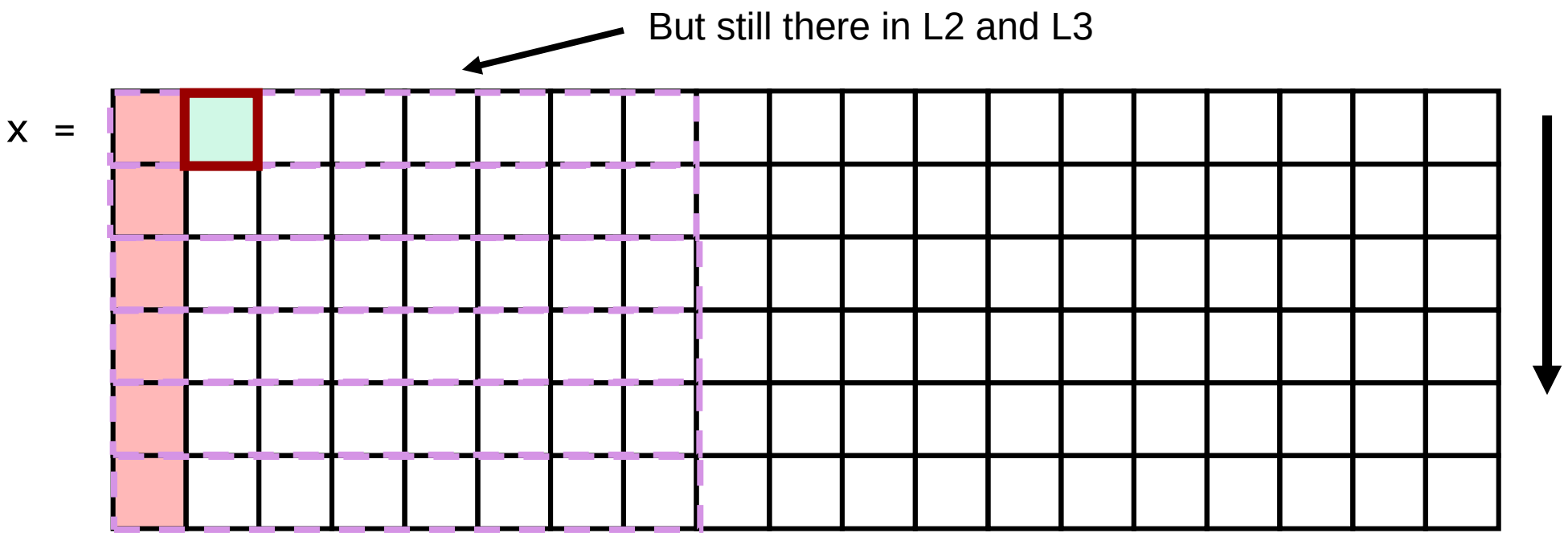

```
s = 0
for c in range(columns):
    for r in range(rows):
        s += x[r, c]
```

Caveat: multiple cache levels




```
s = 0
for c in range(columns):
    for r in range(rows):
        s += x[r, c]
```

Caveat: multiple cache levels

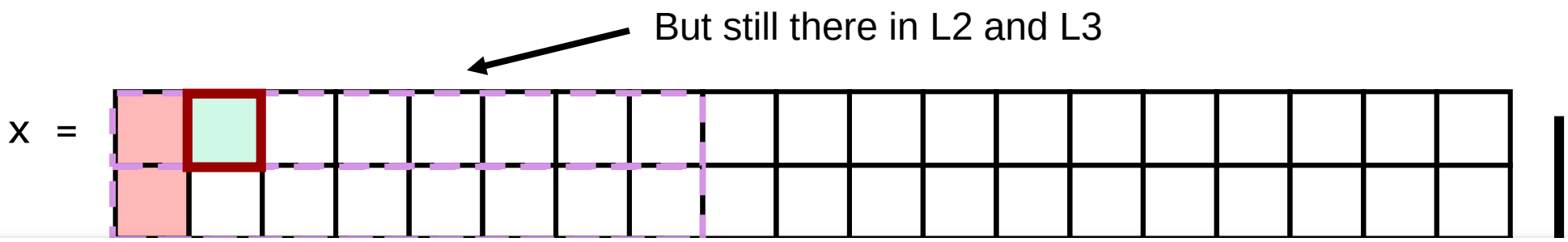


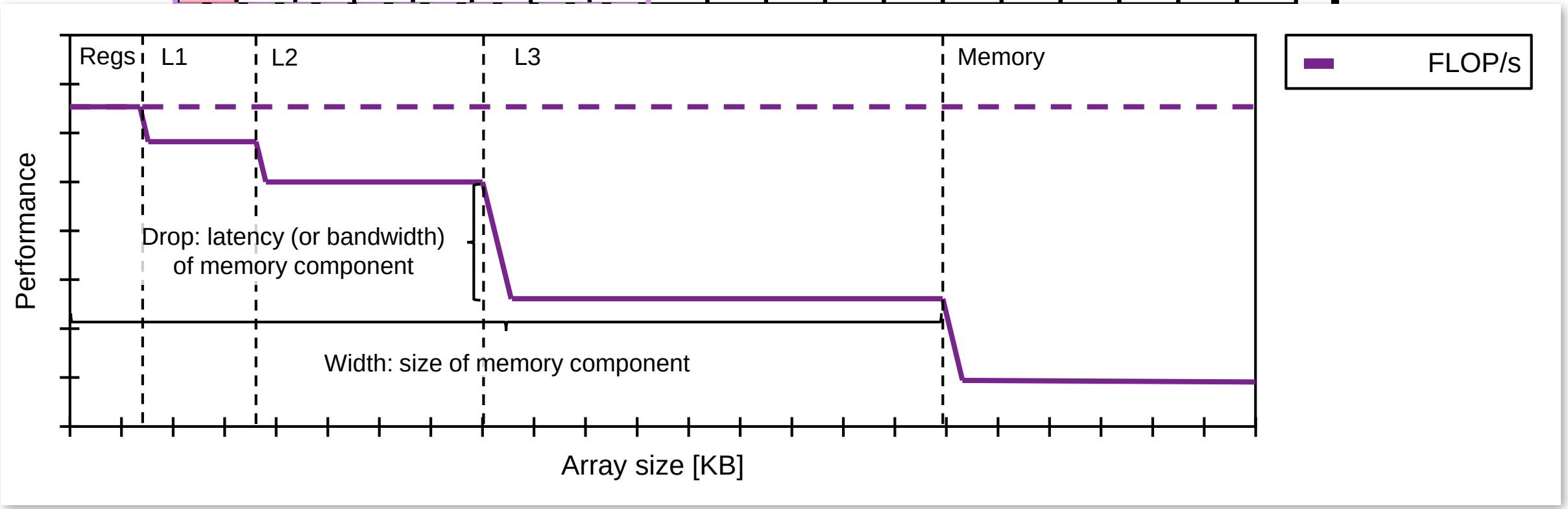

```
s = 0
for c in range(columns):
    for r in range(rows):
        s += x[r, c]
```

Access pattern **does not** match data layout

Miss rate L1: 100.0%
Miss rate L2: 12.5% (1/8)

Caveat: multiple cache levels





Caches: general rules

1. You cannot avoid cache misses

2. …but you should try minimize them

Caches: minimizing misses

1. **Spatial locality**
   Use all data in a cache line

2. **Temporal locality**
   Re-use data in a cache line

Caches: minimizing misses

1. **Spatial locality**
   Re-use data in a cache line
2. **Temporal locality**
   Re-use data in a cache line

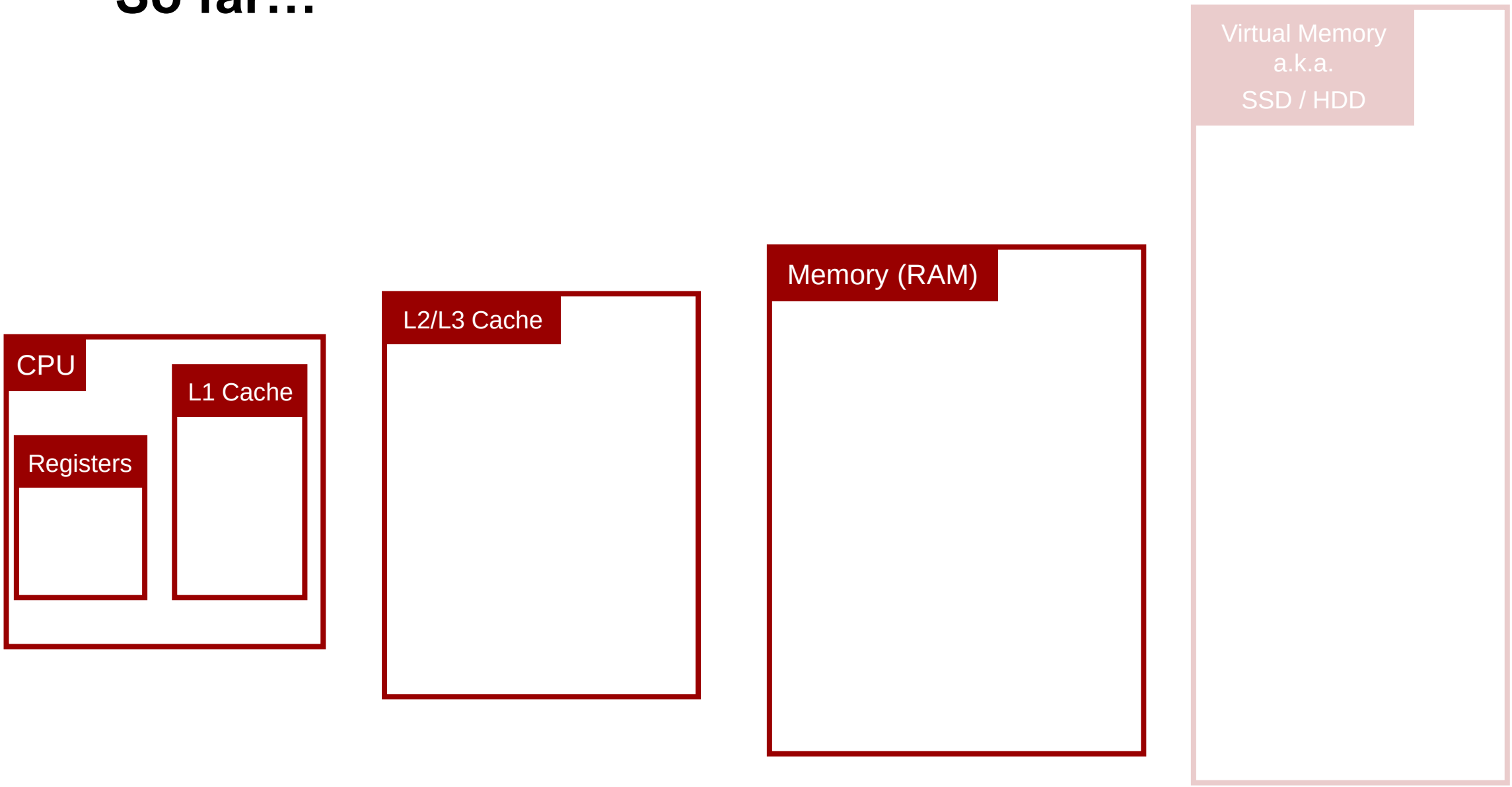
Note: *takeaway is not to care all the time!*
Measure first, then care (a lot)!

Quiz time!

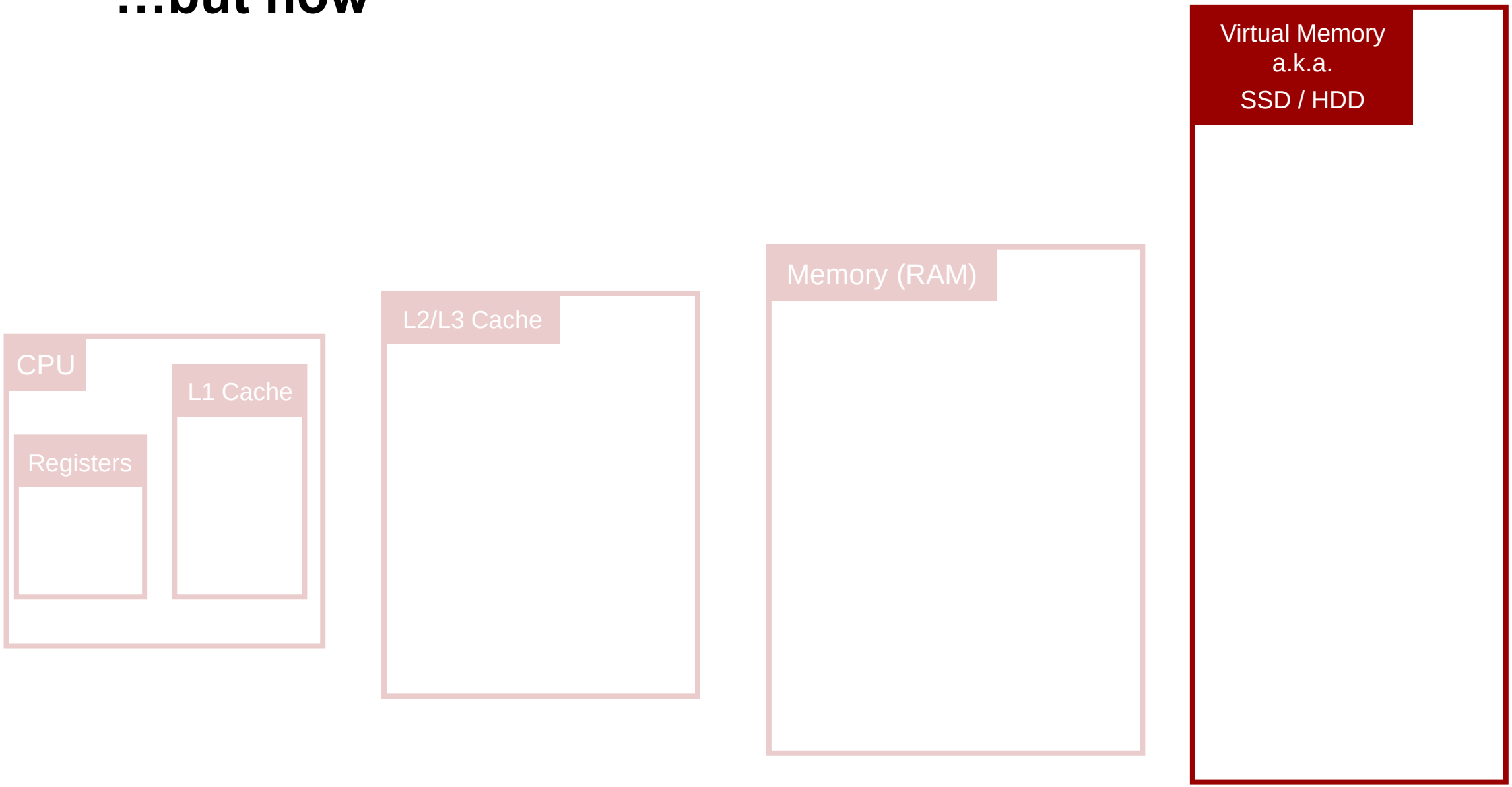
menti.com

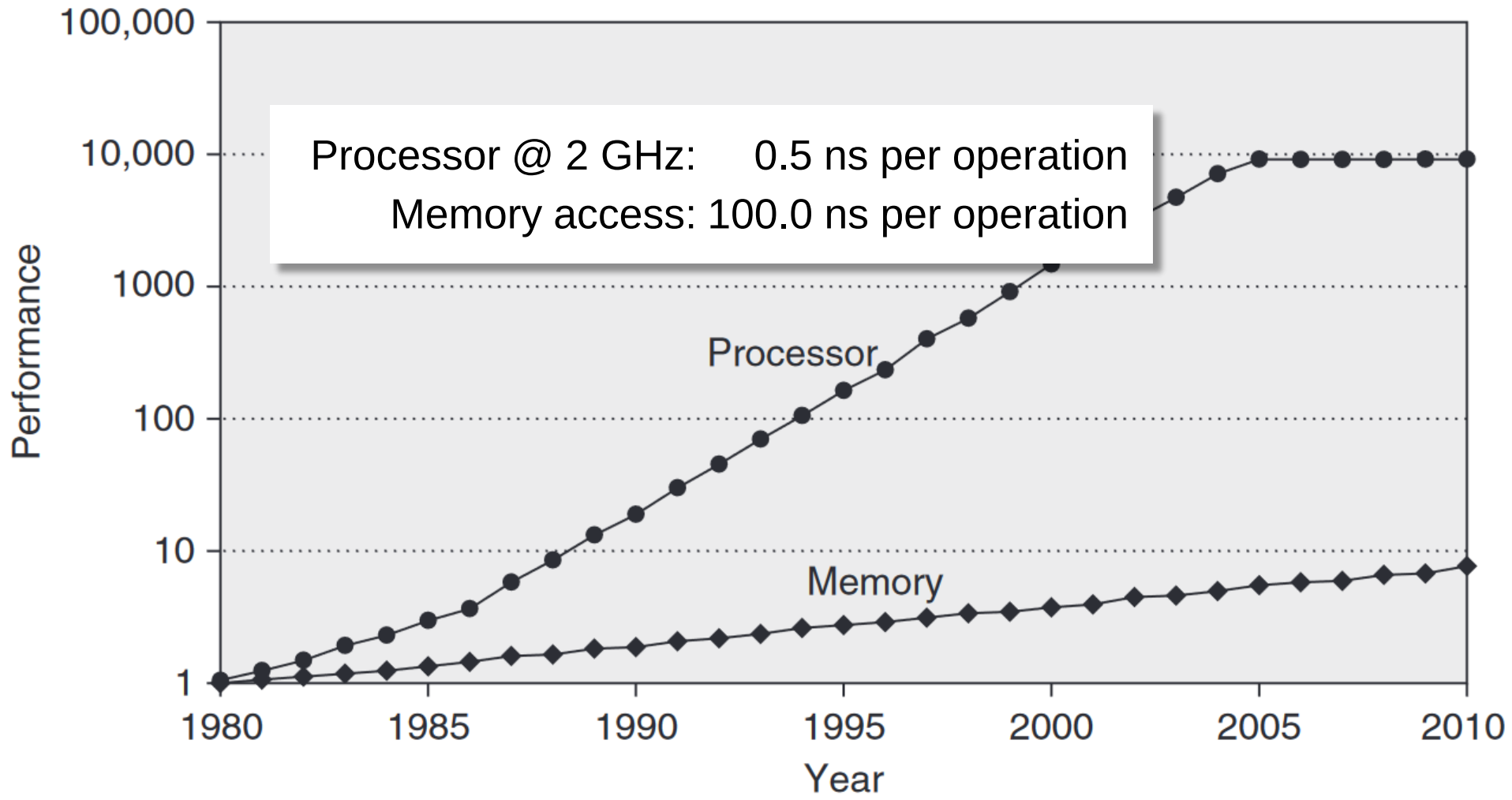
The Memory Hierarchy: Data Storage & Compression

So far…


Virtual Memory
a.k.a.
SSD / HDD
Memory (RAM)
L2/L3 Cache
CPU
L1 Cache
Registers

…but now


Virtual Memory
a.k.a.
SSD / HDD
Memory (RAM)
L2/L3 Cache
CPU
L1 Cache
Registers



Hennesy & Patterson, ”Computer Architecture: A Quantitative Approach”, 5th Ed.

Processor @ 2 GHz: 0.5 ns per operation

M thetical but realistic modern desktop

Type	Size	Access time	
CPU			
L1 cache	256 KB	2 ns	4 CPU ops
L2 cache	1 MB	5 ns	10 CPU ops
L3 cache	6 MB	30 ns	60 CPU ops
RAM			
DIMM	8 GB	100 ns	200 CPU ops
Secondary storage			
SSD	256 GB	50 µs	100,000 CPU ops
HDD	2 TB	5 ms	10,000,000 CPU ops

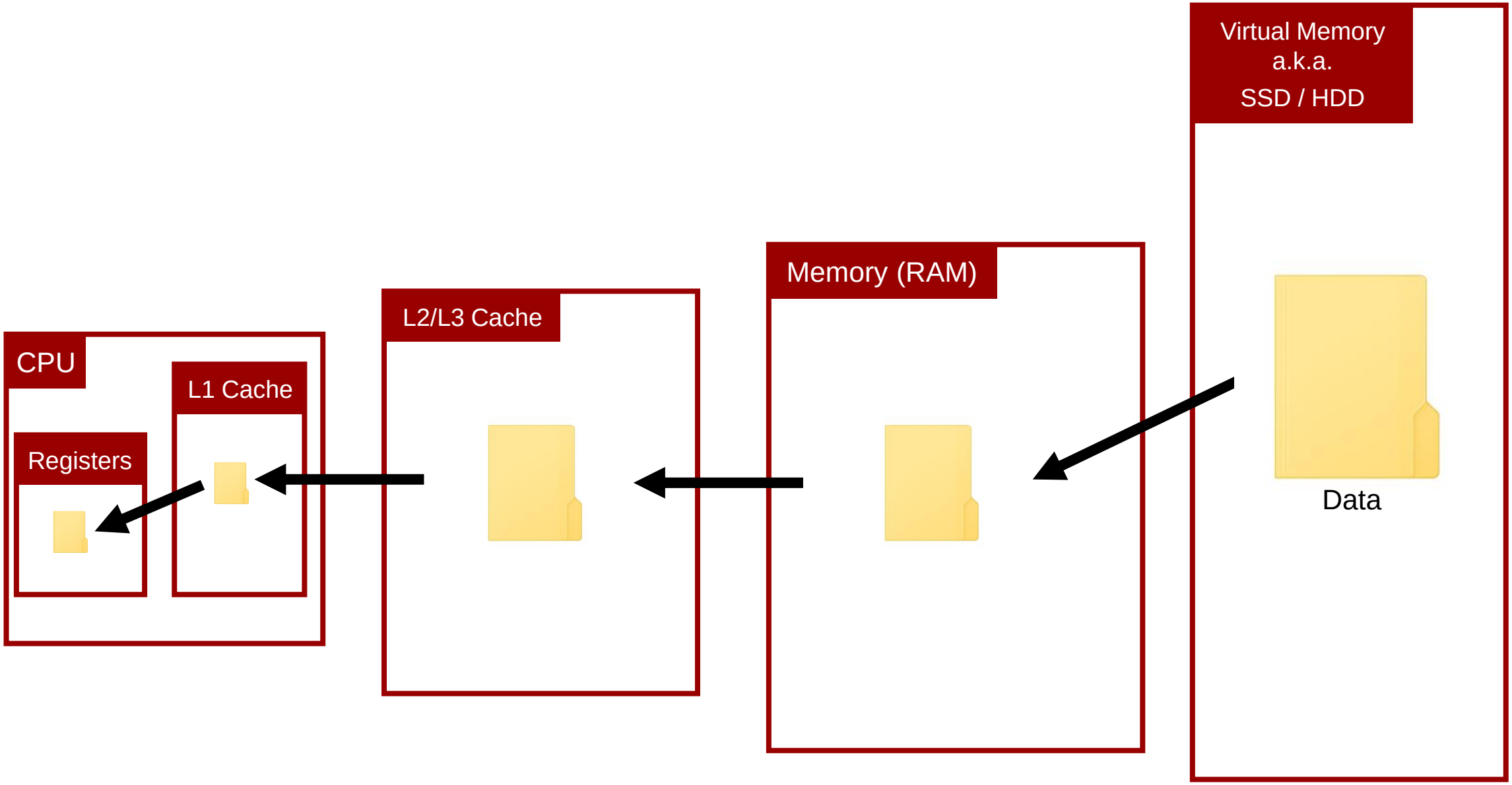
Antão, Fast Python, Table 1.1

Processor (scaled): 1 second per operation

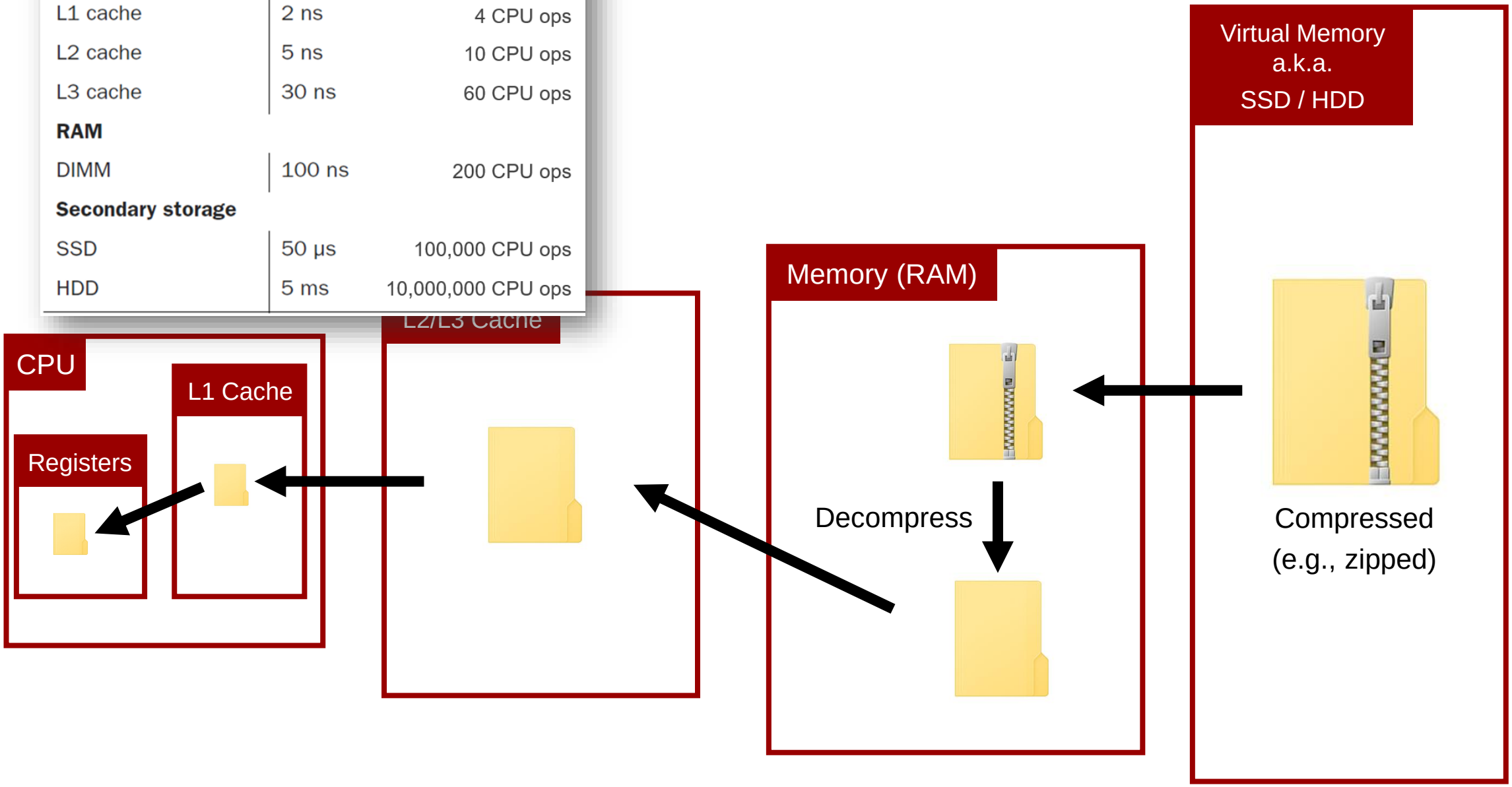
M ...etical but realistic modern desktop

Type	Size	Access time	
CPU			
L1 cache	256 KB	4 seconds	4 CPU ops
L2 cache	1 MB	10 seconds	10 CPU ops
L3 cache	6 MB	1 minute	60 CPU ops
RAM			
DIMM	8 GB	3:20 minutes	200 CPU ops
Secondary storage			
SSD	256 GB	27.8 hours	100,000 CPU ops
HDD	2 TB	3.8 months	10,000,000 CPU ops

Antão, Fast Python, Table 1.1

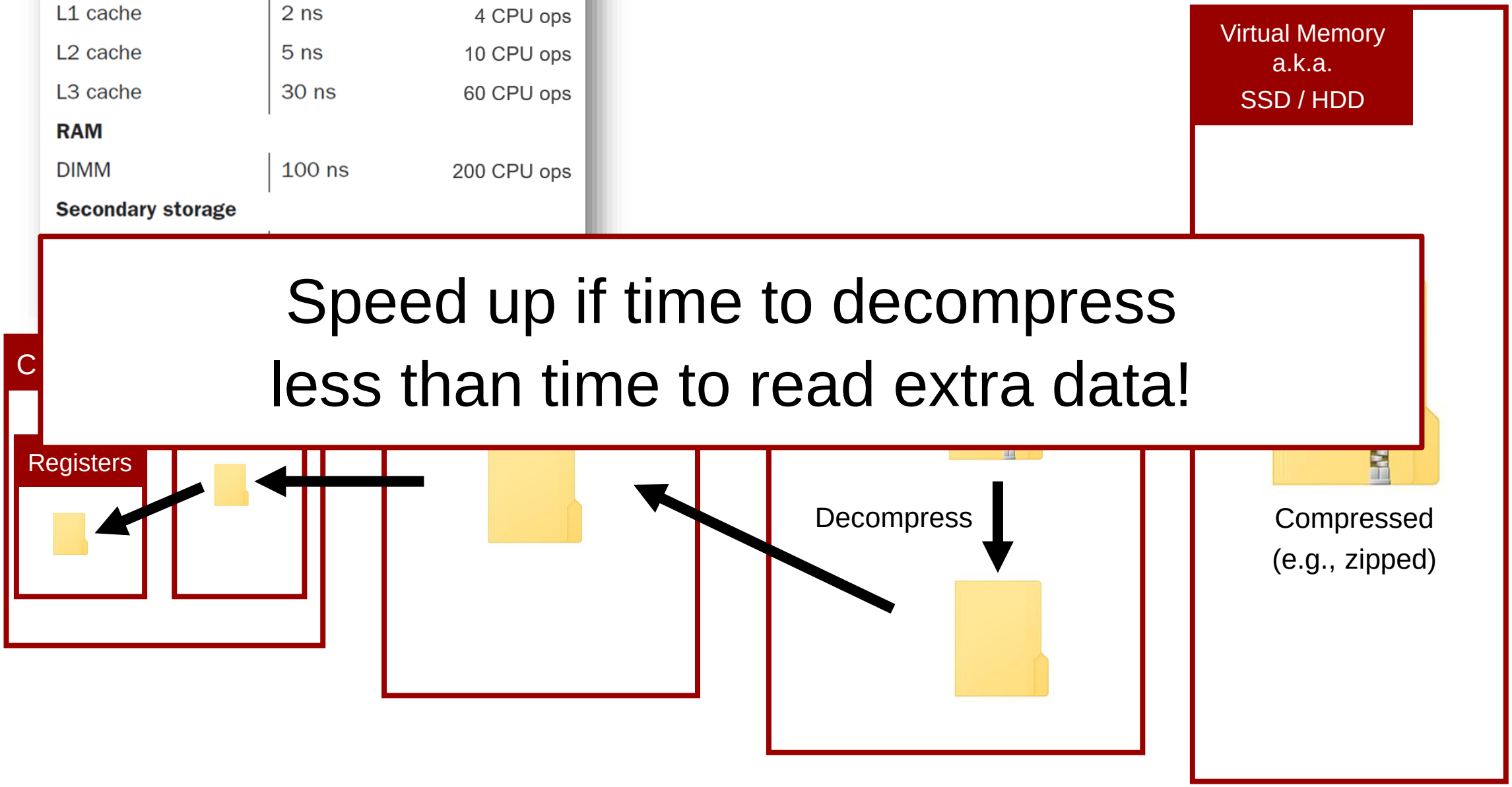

Virtual Memory
a.k.a.
SSD / HDD
Data
Memory (RAM)
L2/L3 Cache
CPU
L1 Cache
Registers

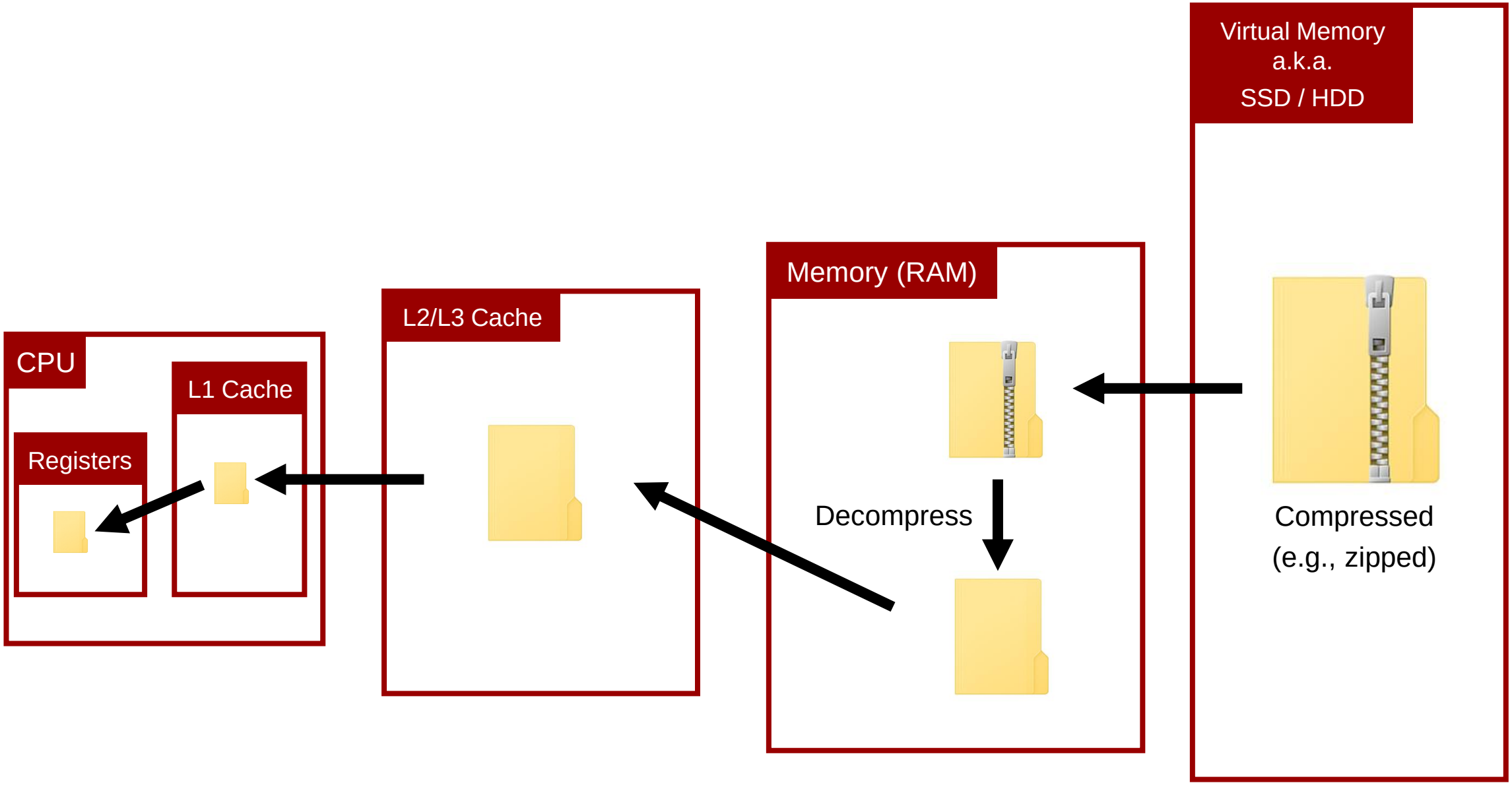
Type	Access time	
CPU		
L1 cache	2 ns	4 CPU ops
L2 cache	5 ns	10 CPU ops
L3 cache	30 ns	60 CPU ops
RAM		
DIMM	100 ns	200 CPU ops
Secondary storage		
SSD	50 µs	100,000 CPU ops
HDD	5 ms	10,000,000 CPU ops

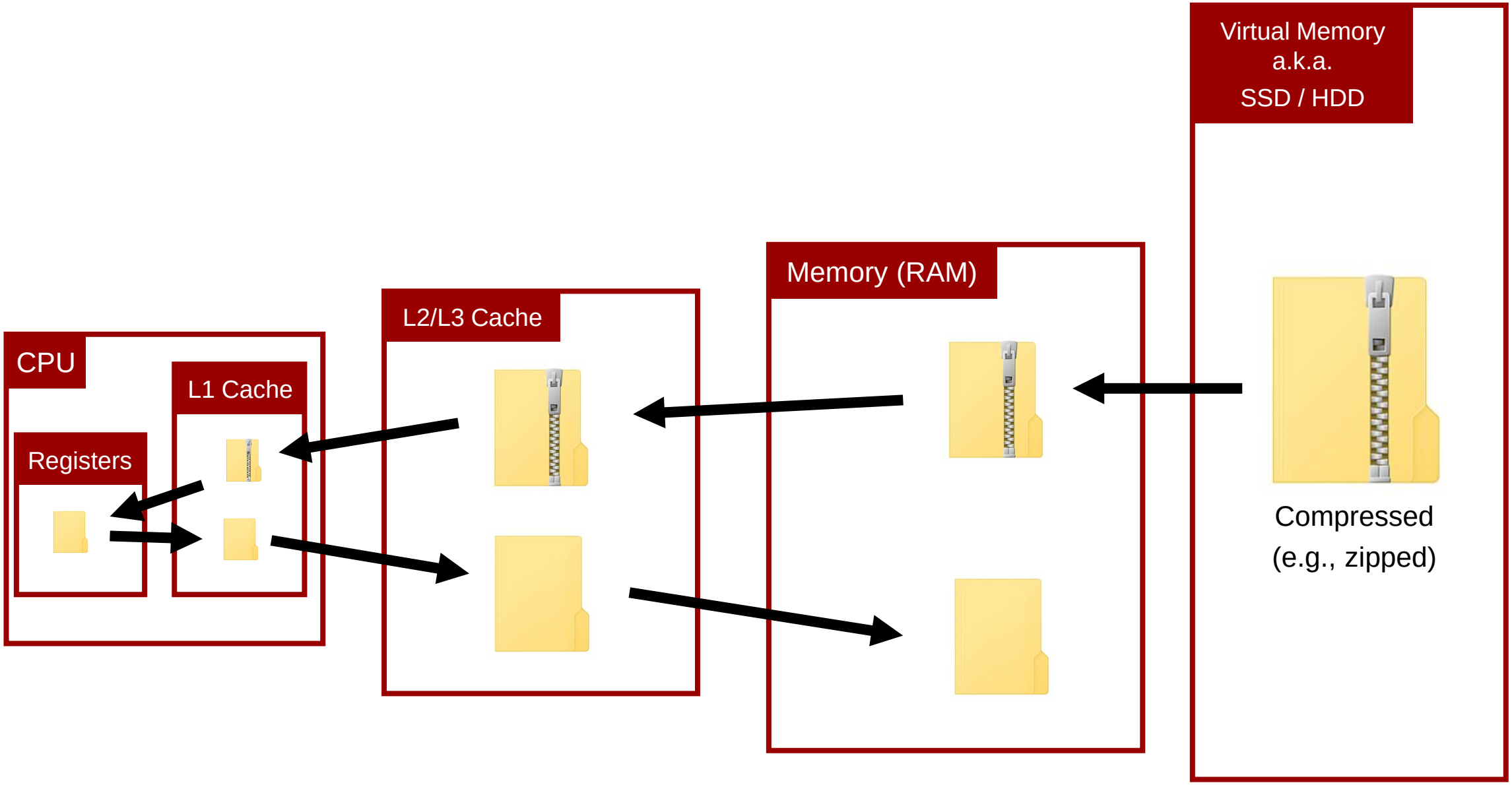


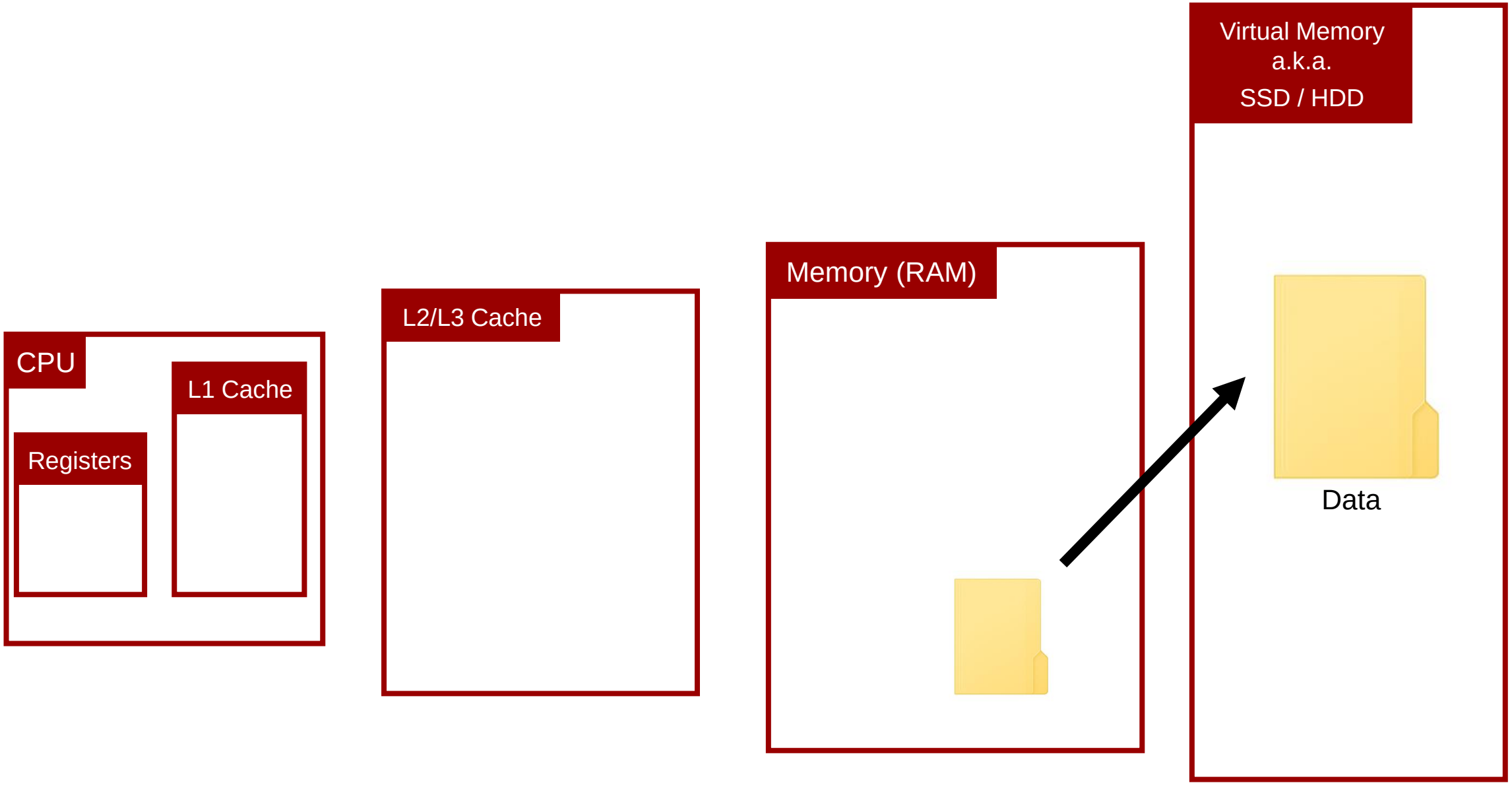
Type	Access time	
CPU		
L1 cache	2 ns	4 CPU ops
L2 cache	5 ns	10 CPU ops
L3 cache	30 ns	60 CPU ops
RAM		
DIMM	100 ns	200 CPU ops
Secondary storage		

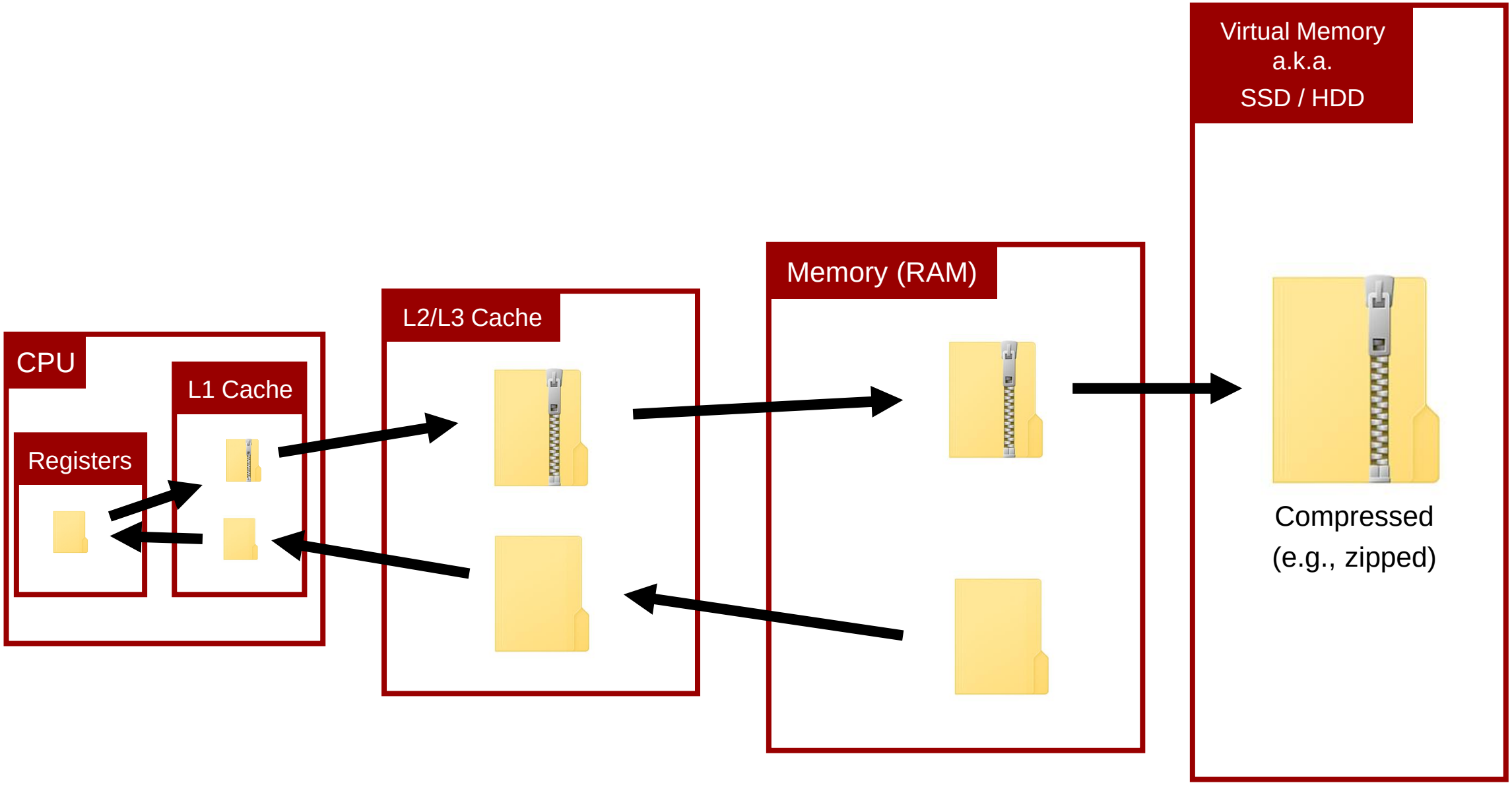
Speed up if time to decompress
less than time to read extra data!

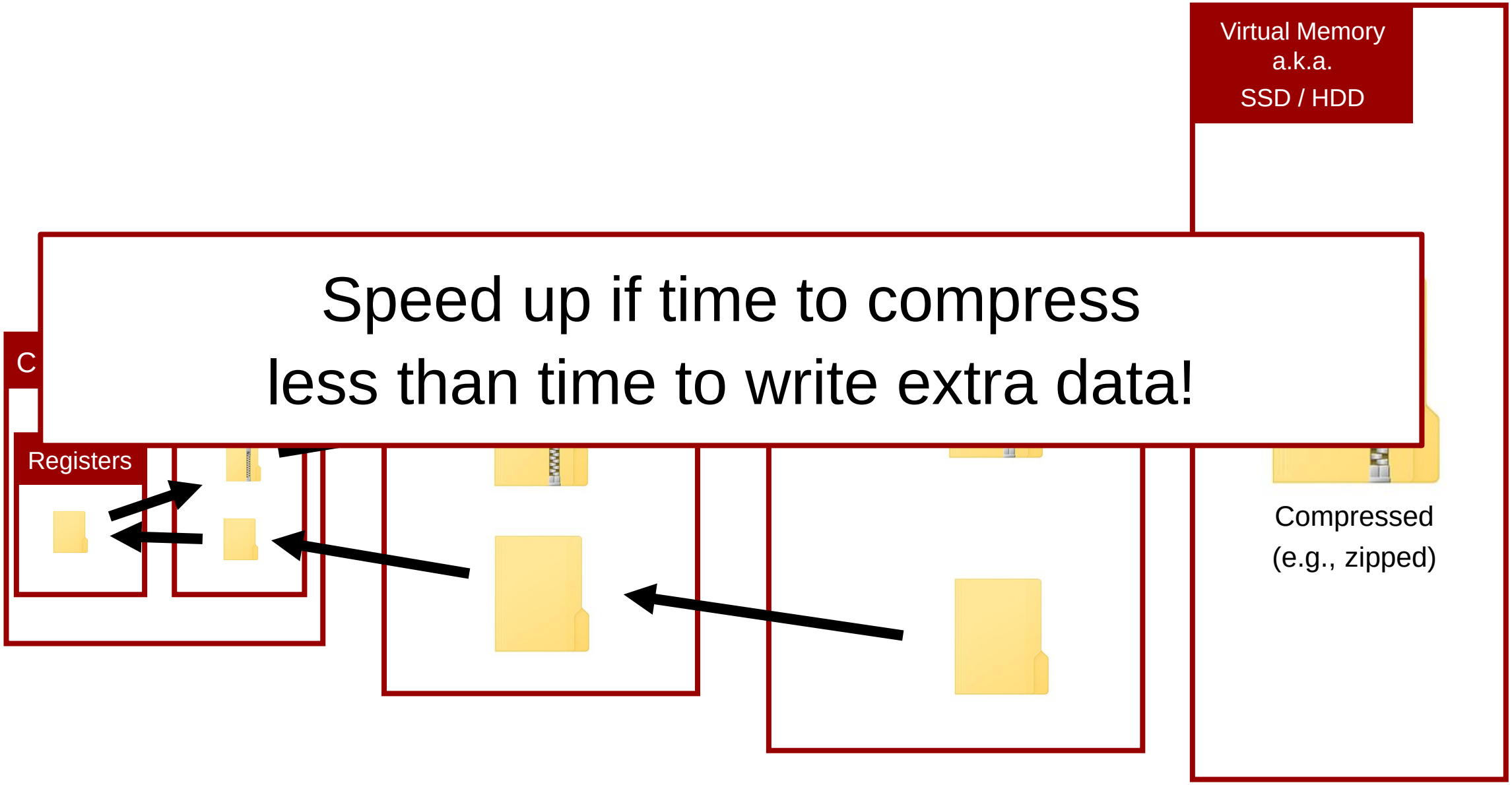


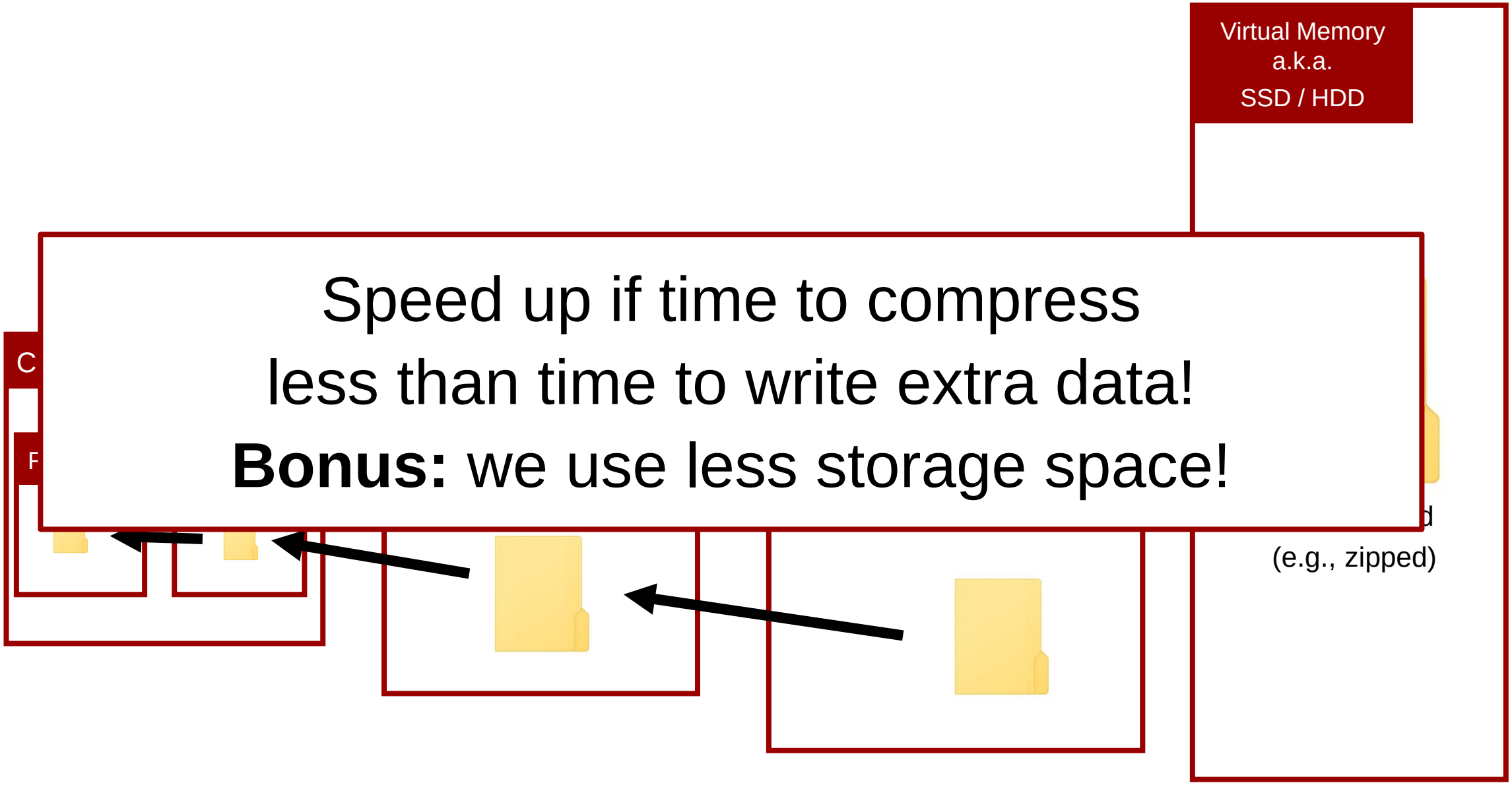

Virtual Memory
a.k.a.
SSD / HDD
Memory (RAM)
L2/L3 Cache
CPU
L1 Cache
Registers
Decompress
Compressed
(e.g., zipped)


Virtual Memory
a.k.a.
SSD / HDD
Memory (RAM)
L2/L3 Cache
CPU
L1 Cache
Registers
Compressed
(e.g., zipped)


Virtual Memory
a.k.a.
SSD / HDD
Memory (RAM)
L2/L3 Cache
CPU
L1 Cache
Registers
Data


Virtual Memory
a.k.a.
SSD / HDD
Memory (RAM)
L2/L3 Cache
CPU
L1 Cache
Registers
Compressed
(e.g., zipped)


Virtual Memory
a.k.a.
SSD / HDD
Speed up if time to compress
less than time to write extra data!
C
Registers
Compressed
(e.g., zipped)



Speed up if time to compress
less than time to write extra data!
Bonus: we use less storage space!

When to expect speed-ups

Speed up if time to compress/decompress less than time to write/read extra data

Depends on:

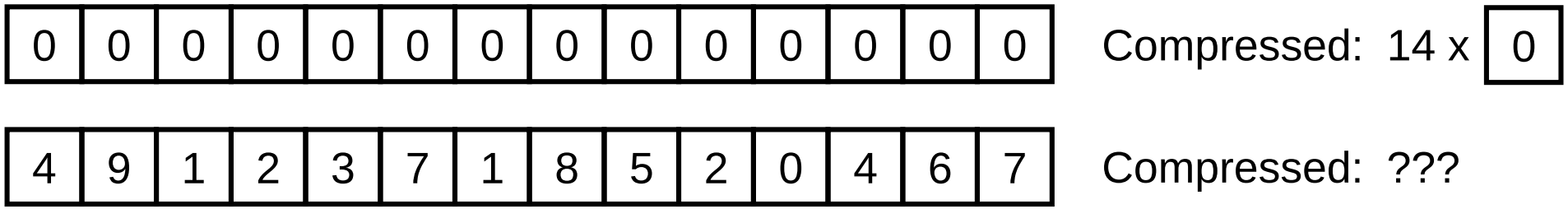
- Storage: HDD, SSD, NVMe, etc.,
- Algorithm: gzip, lz4, blosc, etc.,
- Data: random or with patterns

When to expect speed-ups

Speed up if time to compress/decompress less than time to write/read extra data

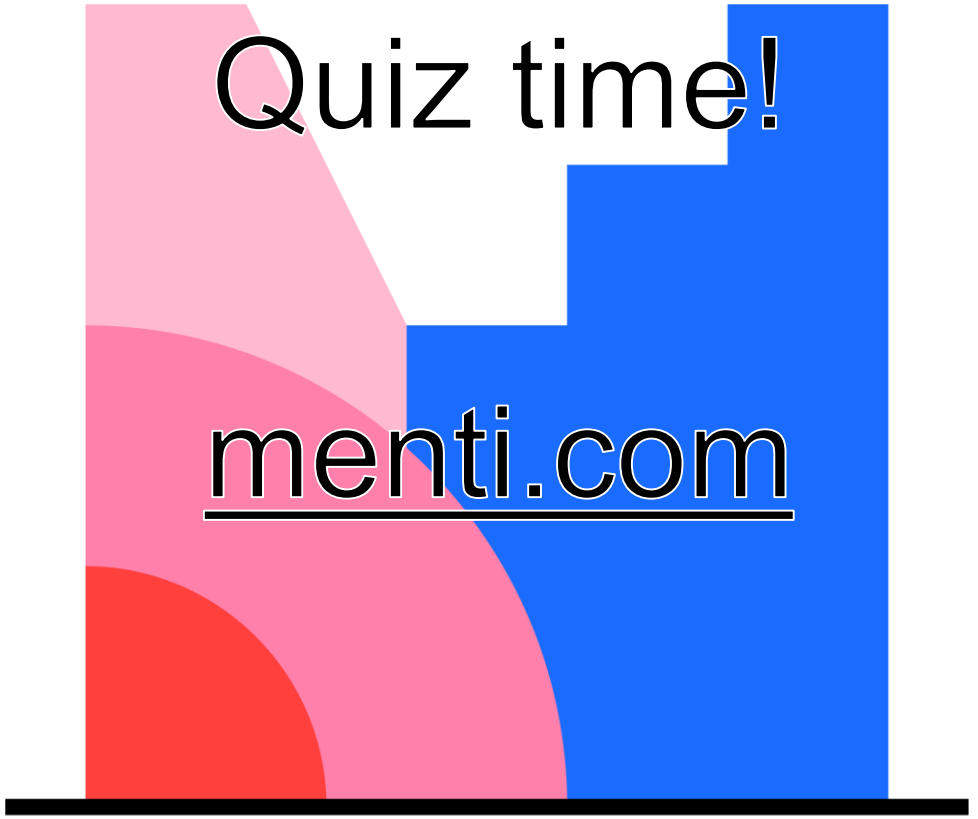
Depends on:

- Storage: HDD, SSD, NVMe, etc.,
- Algorithm: gzip, lz4, blosc, etc.,
- Data: random or with patterns



Quiz time!

menti.com



Mentimeter

Takeaways

- Hardware matters!
- Performance depends on the data not just code
- Minimizing data transfer = faster programs

Today’s exercise

Observe hardware effects with your own eyes

- Cache effects
  - Measuring performance (wall time, FLOP/s)
  - Effect of data access pattern
  - Effect of data size

- Compressed storage
  - Using the `blosc` package
  - Explore compression for different kinds of data

Observe hardware effects with your own eyes

- Cache effects

  Word of warning nr. 1

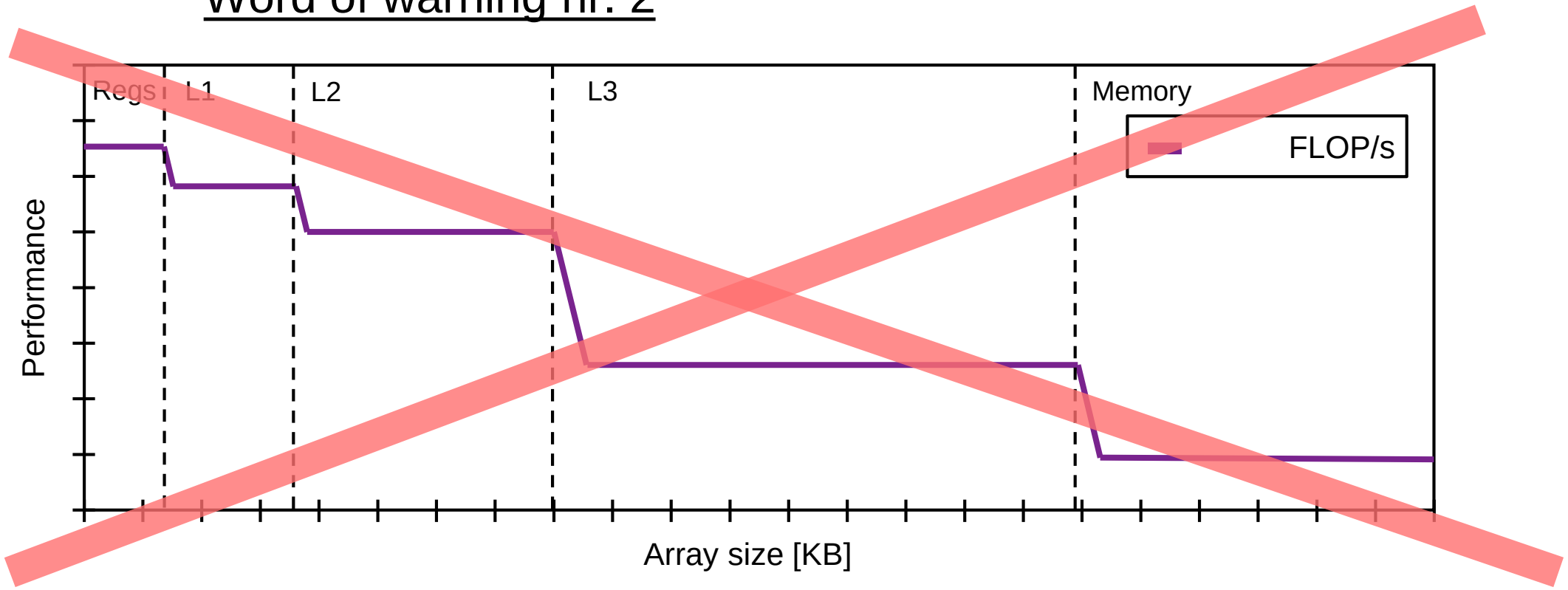
Do ***NOT*** try to solve these exercises on your own laptop!

PLEASE!

Observe hardware effects with your own eyes

- Cache effects

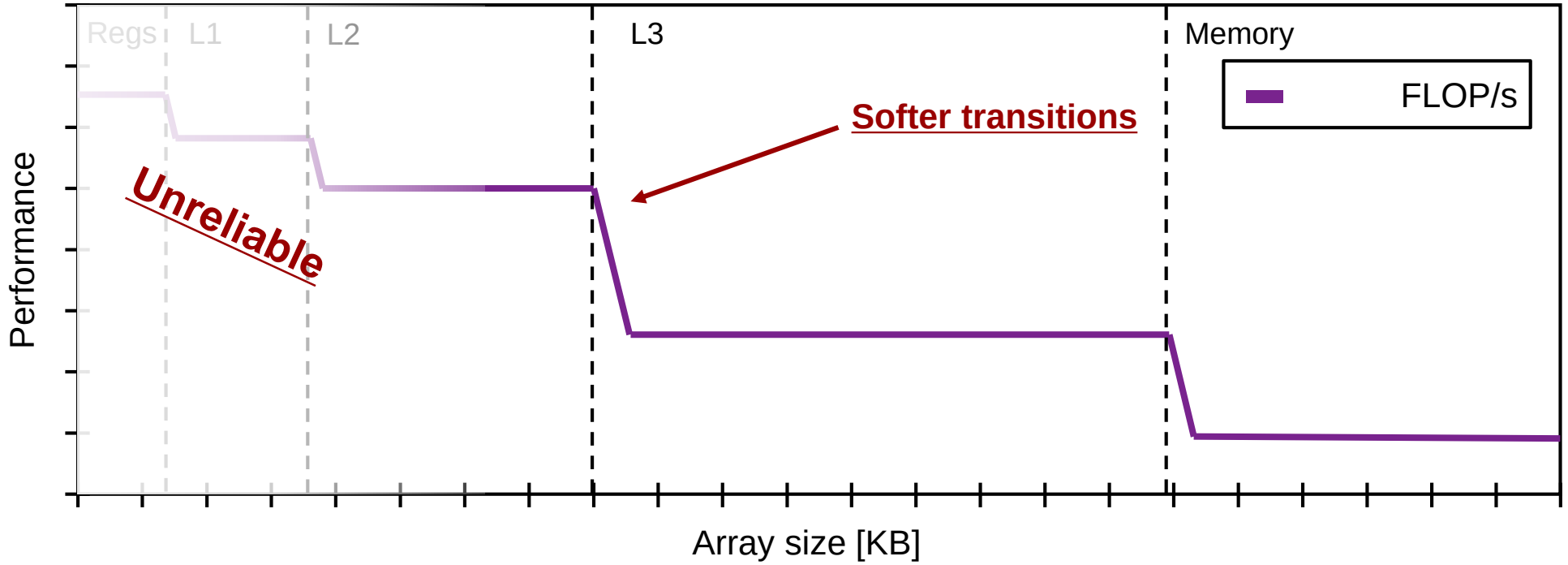
Word of warning nr. 2



Observe hardware effects with your own eyes

- Cache effects

  Word of warning nr. 2



Useful concepts

Python command line arguments
import sys
sys.argv # List of arguments

Python timing
from time import perf_counter as time
t = time()
… # Code to time
t = time() – t

Memory for NumPy arrays
arr = np.zeros(10, dtype='float64')
arr.size * arr.dtype.itemsize # Num. bytes = 80
arr.nbytes # Num. bytes = 80

MFLOP/s

Mega (1,000,000) floating point operations / second

Change to work node
linuxsh

Submit job script
bsub < submit.sh

Job status
bstat / bjobs

Check job output
bpeek / bpeek <JOBID>

Kill job
bkill <JOBID>