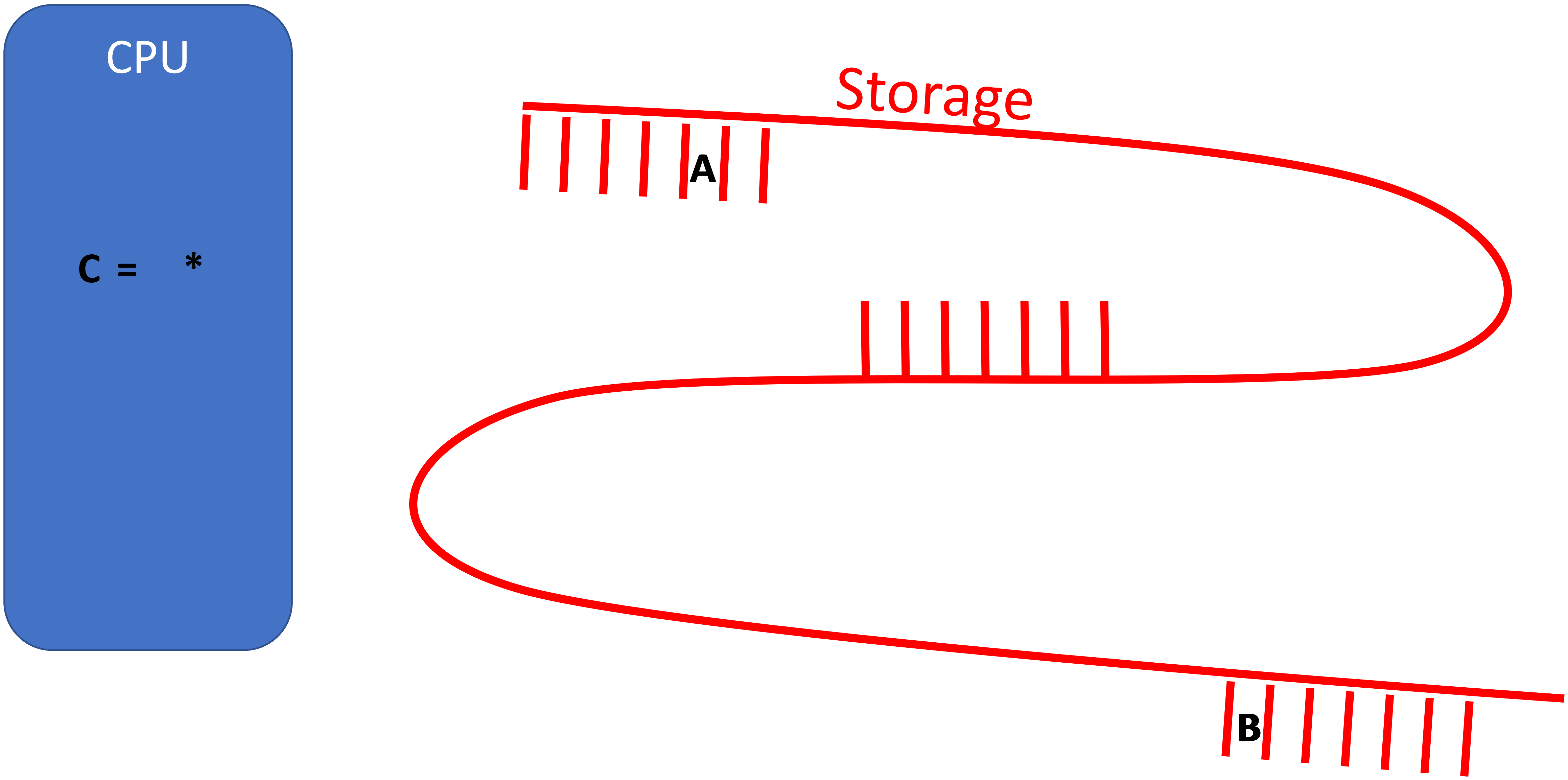
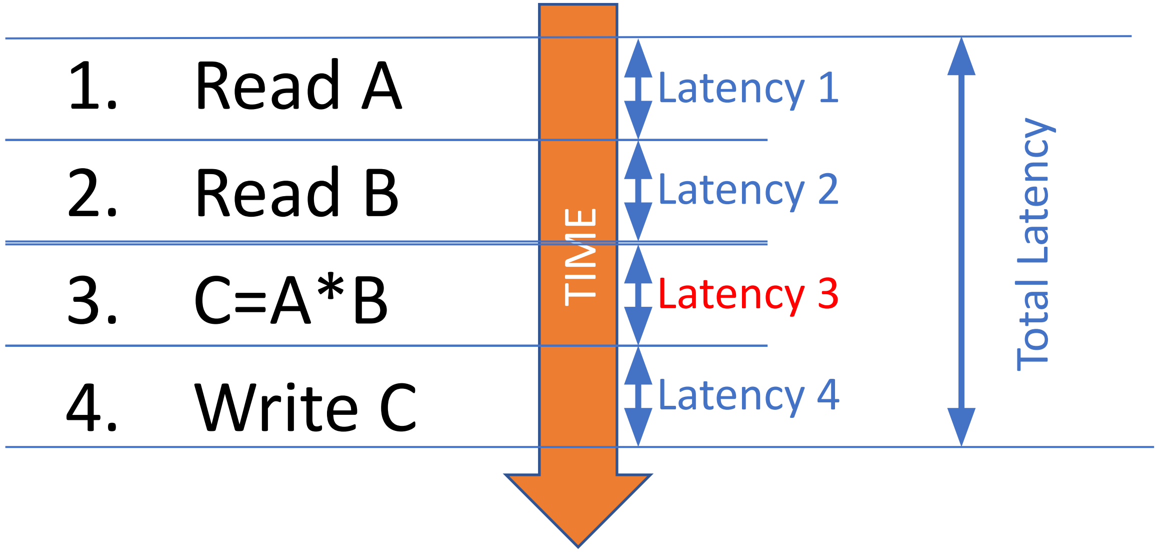
2: Storage Latency


CPU
C = *
Storage
A
B

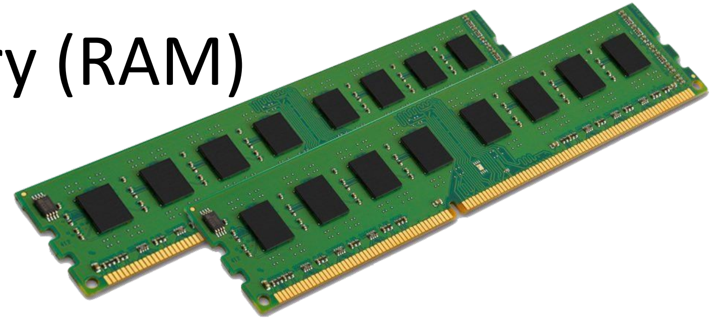
Latencies



With big data, most of the latency is memory latency (1,2,4), not computation (3)

Storage Types

- Main Memory (RAM)



- Spinning disk

- Remote computer

Summary for part 2

- The major source of latency in data analysis is reading and writing to storage
- Different types of storage offer different latency, capacity and price.
- Big data analytics revolves around methods for organizing storage and computation in ways that maximize speed while minimizing cost.
- Next, storage locality.

3: Caching

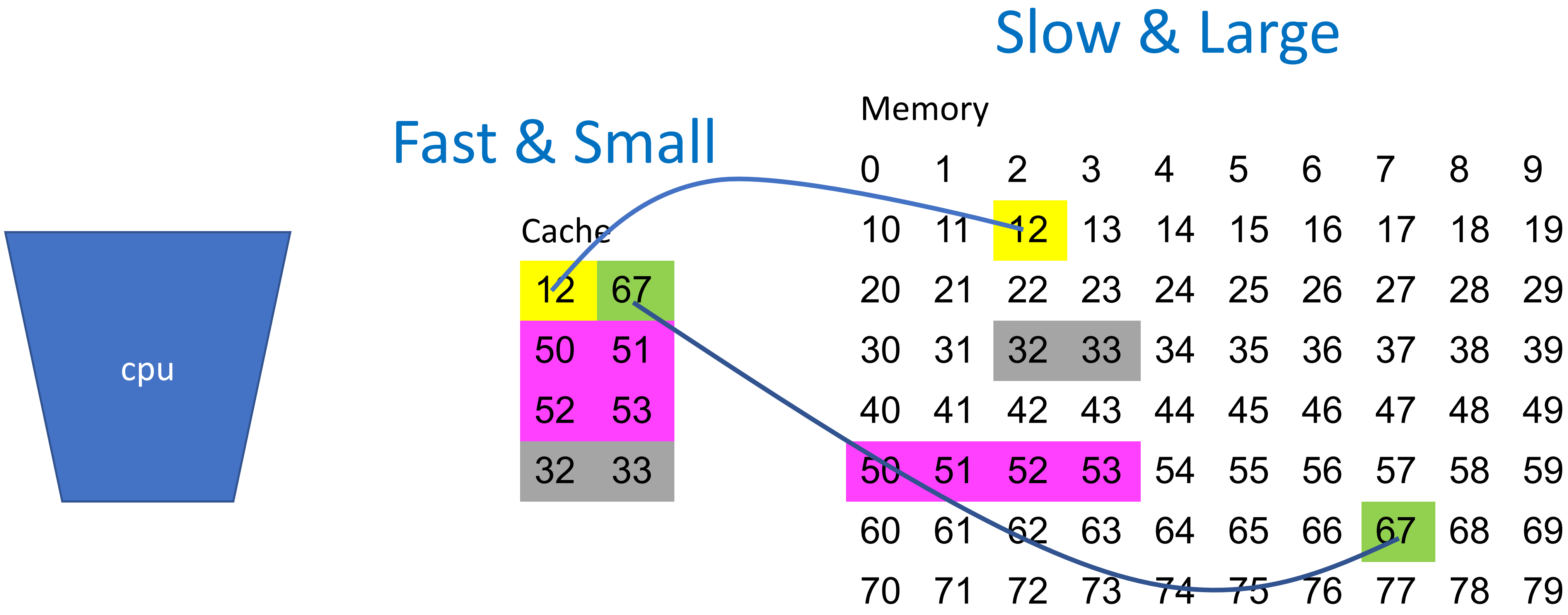
Latency, size and price of computer memory

Given a budget, we need to trade off

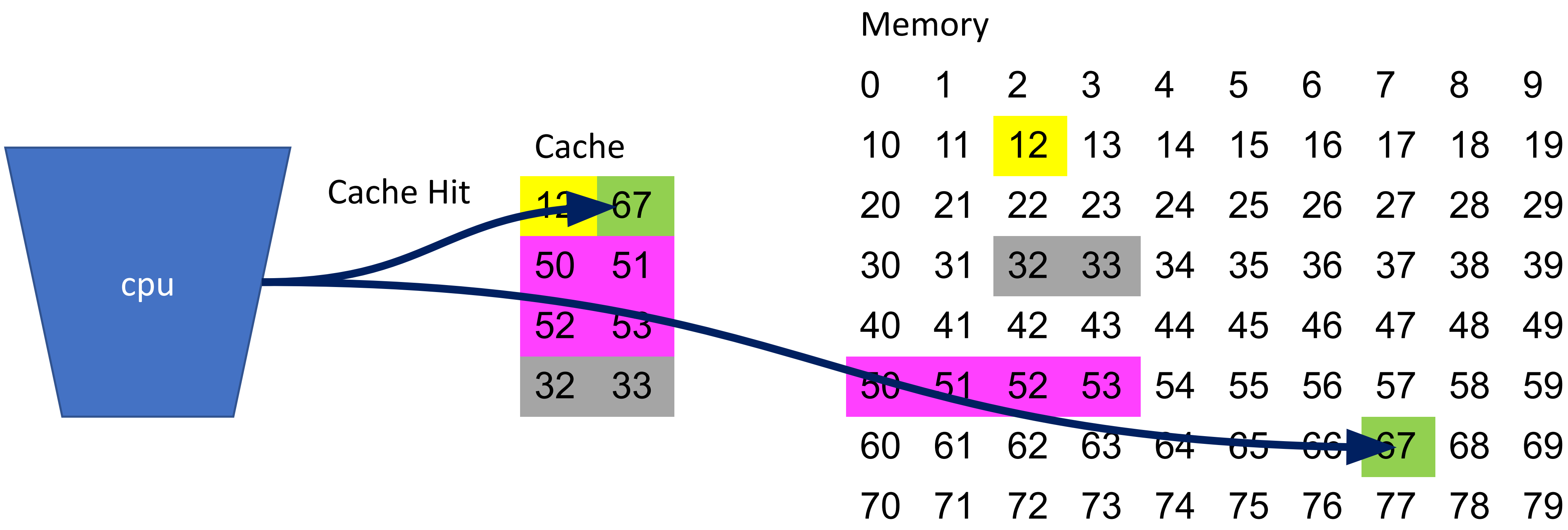
$10: Fast & Small

$10: Slow & Large

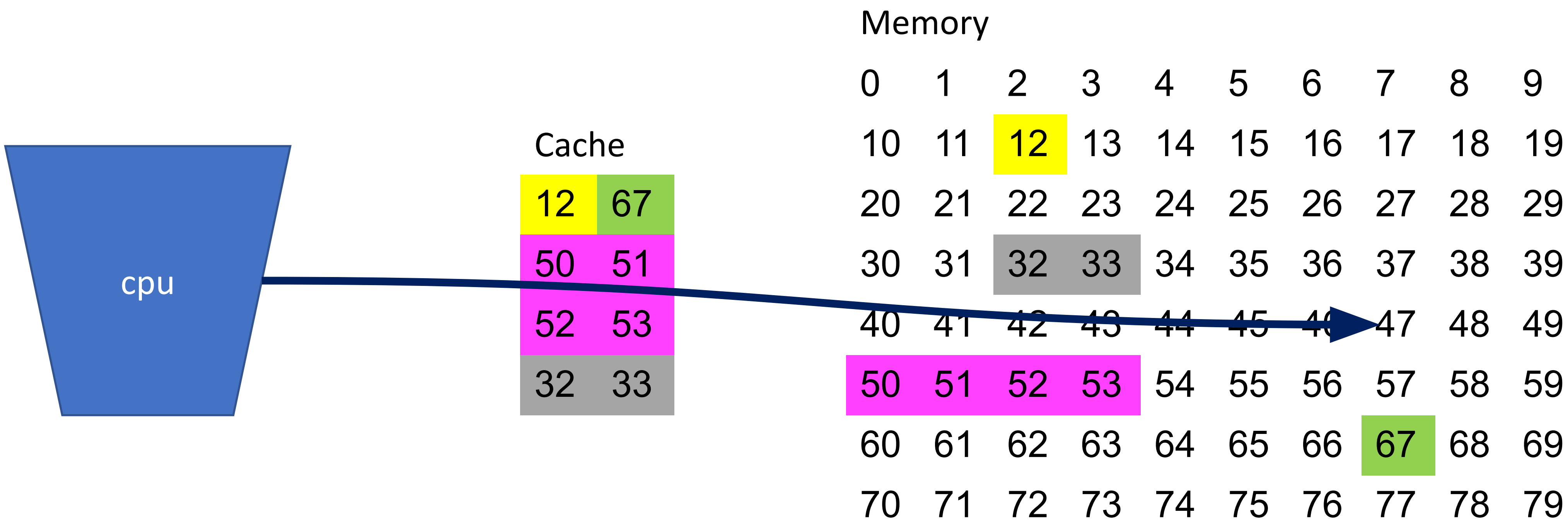
Cache: The basic idea



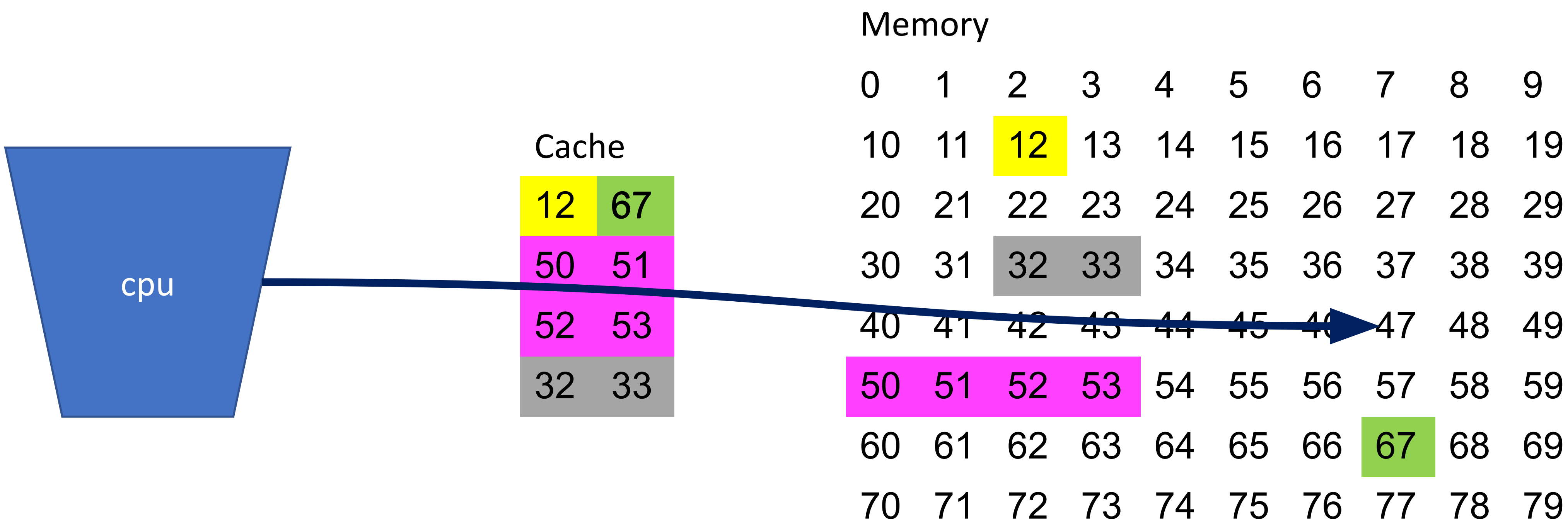
Cache Hit


Memory
0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79
Cache
12 67
50 51
52 53
32 33
Cache Hit
cpu

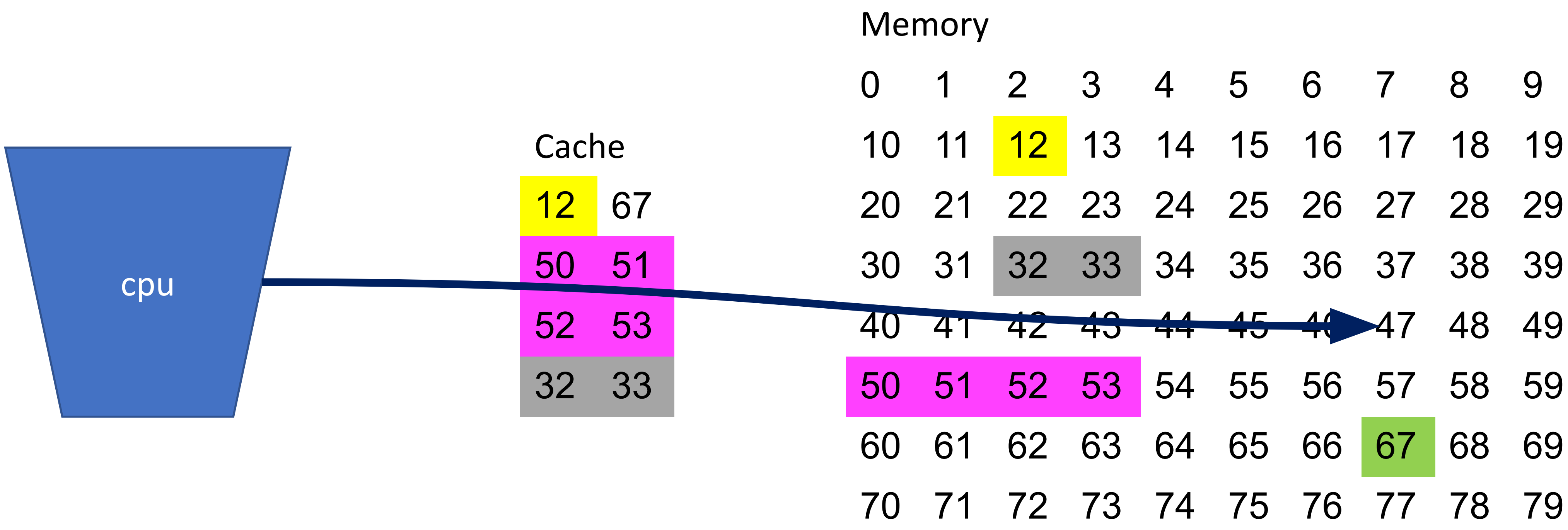
Cache Miss


cpu
Cache
12 67
50 51
52 53
32 33
Memory
0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79

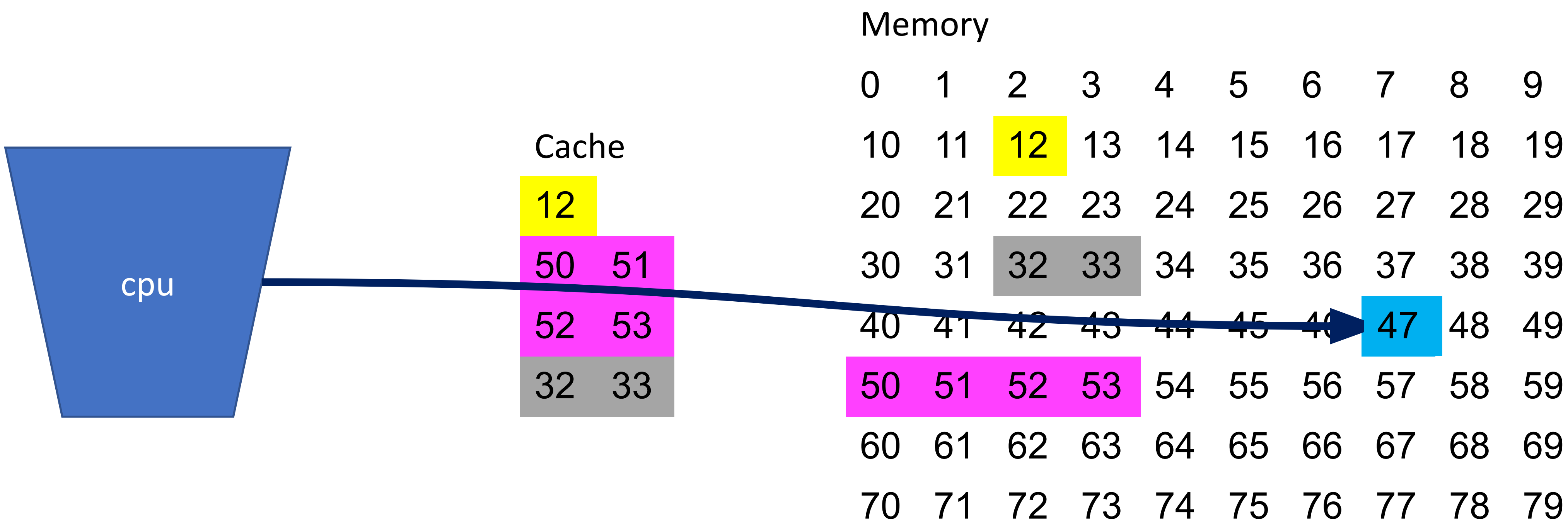
Cache Miss Service: 1) Choose byte to drop



Cache Miss Service: 2) write back



Cache Miss Service: 3) Read In



Summary of part 3

- Cache is much faster than main memory
- Cache hit = the needed value is already in the cache
- Cache miss = the needed value is not in the cache – needs to be brought in from memory
- If there is no space in cache:
  - need to make space
  - If dirty, need to write value back to memory first.
- Cache miss – latency is much bigger than cache miss.

4: Locality of storage access

Access Locality

- The cache is effective If most accesses are hits.
  - Cache Hit Rate is high.
- **Cache effectiveness** depends on patterns (statistics) of memory access.
- **Temporal Locality**: Multiple accesses to **same** address within a short time period
- **Spatial Locality**: Multiple accesses to **near-by** addresses within a short time period

Temporal Locality

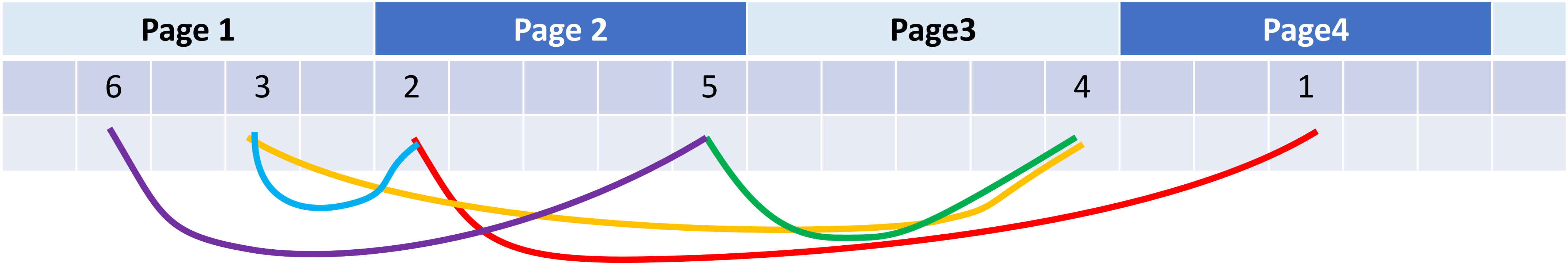
- **Task:** compute the function $f_\theta(x)$ on a long sequence $x_1, x_2, \dots, x_n$
- θ is a parameter vector – example: the weights in a neural network.
- The parameters θ are needed for each computation.
- If θ fits in the cache – access is fast
- If θ does **not** fit in the cache – each x_i causes at least one cache miss – program will be much slower.
- **Temporal Locality:** repeated access to the same memory location

Spatial locality

-
- **Task:** compute the function $\sum_{i=1}^{n-1}(x_i - x_{i+1})^2$ on $x_1, x_2, \ldots, x_n$
- Contrast two ways to store $x_1, x_2, \ldots, x_n$:
- Linked list (poor locality)
- Indexed array (good locality)

Linked List

Let $x_1, x_2, \ldots, x_n$ be 1,2,3,4,5,6



Traversal of 6 elements touches 4 pages

array

Let $x_1, x_2, \ldots, x_n$ be 1,2,3,4,5,6

Page 1					Page 2					Page3					Page4					
							1	2	3	4	5	6								

Traversal of 6 elements touches 2 pages

Summary of Part 4

- Caching Is effective when memory access is local
- Temporal locality: accessing the same location many times in a short period of time.
- Spatial locality: accessing close-by locations many times in a short period of time.
- Hardware and compilers have a symbiotic relationship: success if compiler generates machine code that has good locality.

Word Count

- Task: given a (large) text
- Count the number of times each word
- Output (word,count) sorted in decreasing order by Count.

Unsorted word count / poor locality

```
=== unsorted list:
the,vernacular,but,as,for,you,ye,carrion,rogues,turning,to,
```

```
Dict={}
For word in list:
    if word in Dict:
        Dict[word]+=1
    else:
        Dict[word]=1
```

Suppose

len(list)=1,000,000

len(Dict) = 100,000

Access to list: spatially local

Access to Dict: **random**

sorted word count / good locality

```
=== sorted list:
lines,lingered,lingered,lingered,lingered,lingered,lingerin
g,lingering,lingering,lingering,lingering,lingering,lingeri
ng,lingering,lingers,lingo,lingo,lining,link,link,linked,li
nked,linked,linked,links,links
```

```
Dict={}
Sort(list)
For word in list:
    if word in Dict:
        Dict[word]+=1
    else:
        Dict[word]=1
```

Suppose

len(list)=1,000,000

len(Dict) = 100,000

Access to list: spatially local

Access to Dict: **Spatially local**

But what about the sort step?

Sorting can be done in time **O(n)**

Efficient in distributed setup

Summary

- Improved memory locality reduces run-time
- Why?
  - Because computer memory is organized in pages.
  - And caching retrieves a page at a time.