The Memory Hierarchy

Prof. Hyuk-Yoon Kwon
https://sites.google.com/view/seoultech-bigdata

Most parts are based on slides written by Brayant and O'Hallaon, CMU
(http://csapp.cs.cmu.edu/3e/instructors.html)

Today

- **Storage technologies and trends**
- Locality of reference
- Caching in the memory hierarchy

Logical Disk Blocks

access units in disk = sectors c tracks c surfaces

(physical level / physical disk) ⟷ (logical view / disk blocks)

- **Modern disks present a simpler abstract view of the complex sector geometry:**
  - The set of available sectors is modeled as a sequence of b-sized logical blocks (0, 1, 2, ...)

have same size!

- **Mapping between logical blocks and actual (physical) sectors**
  - Maintained by hardware/firmware device called disk controller.
  - Converts requests for logical blocks into (surface,track,sector) triples.

interface to communicate with disk when computer system requires logical blocks to controller

disk controller

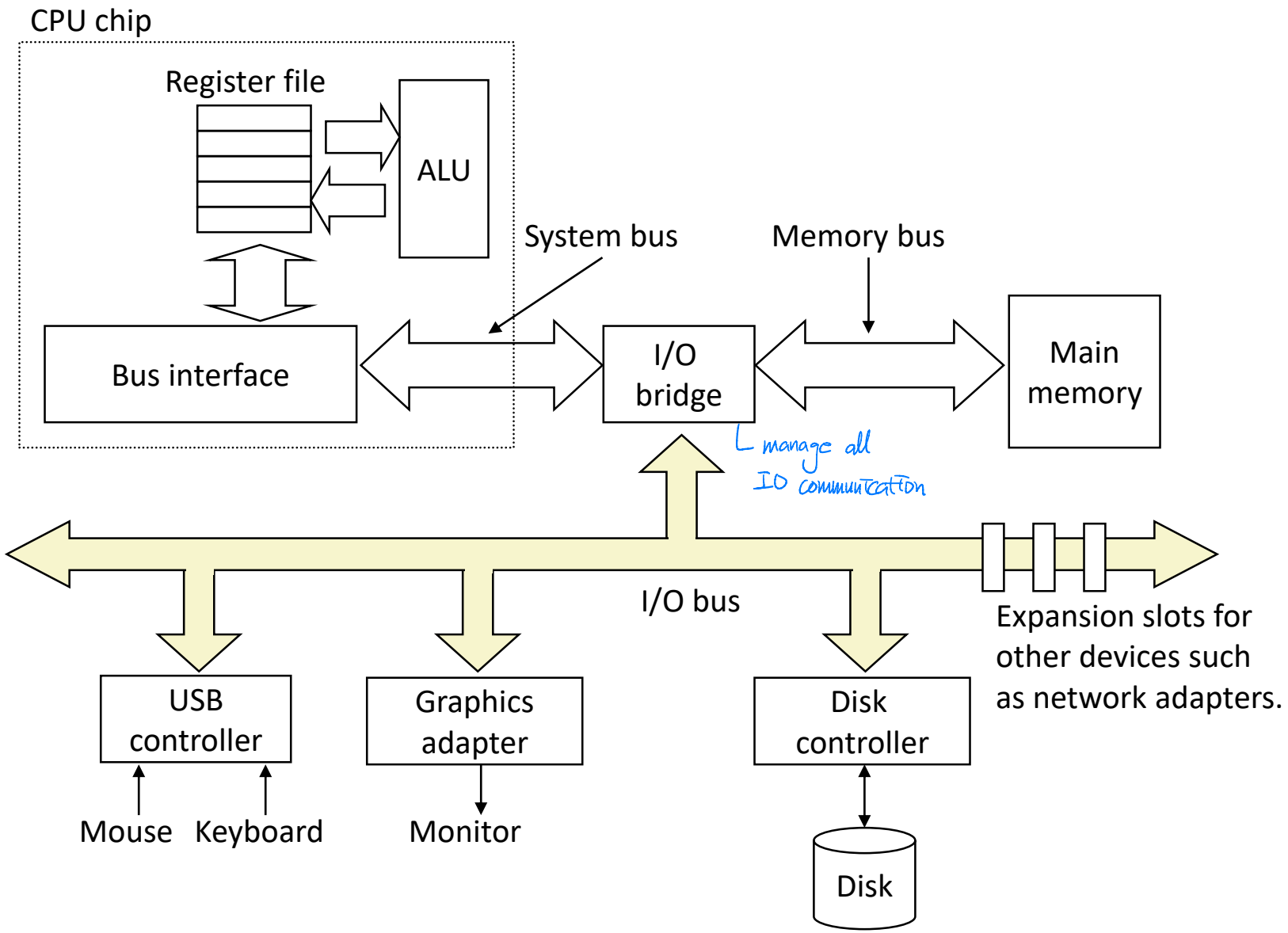
in physical disk

just consider disk as logical block.

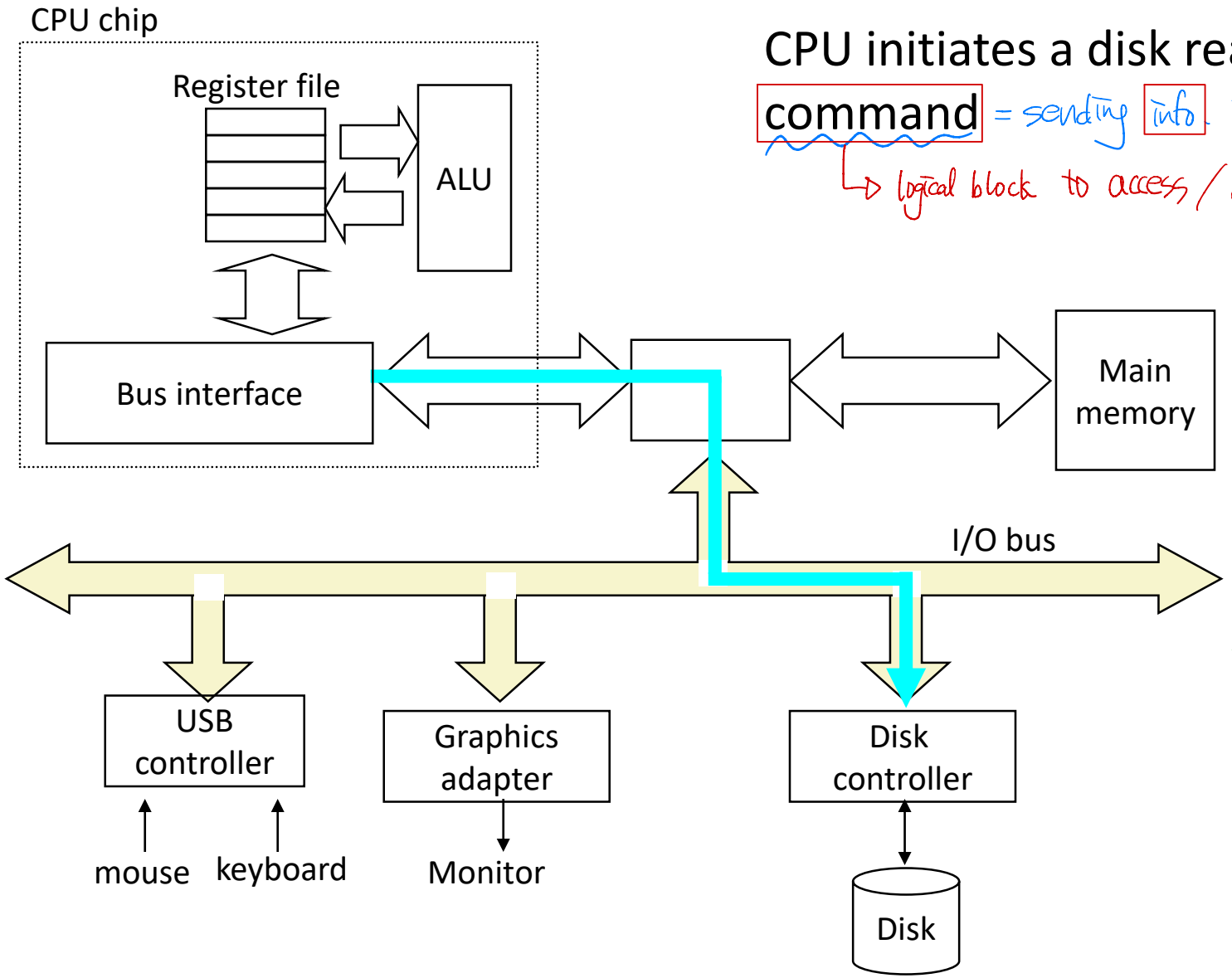
do not care about # of triples due to mapping relationship

I/O Bus

→ support communication between IO device and computer systems



Reading a Disk Sector (1)



CPU initiates a disk read by writing a command = sending info. to disk controller

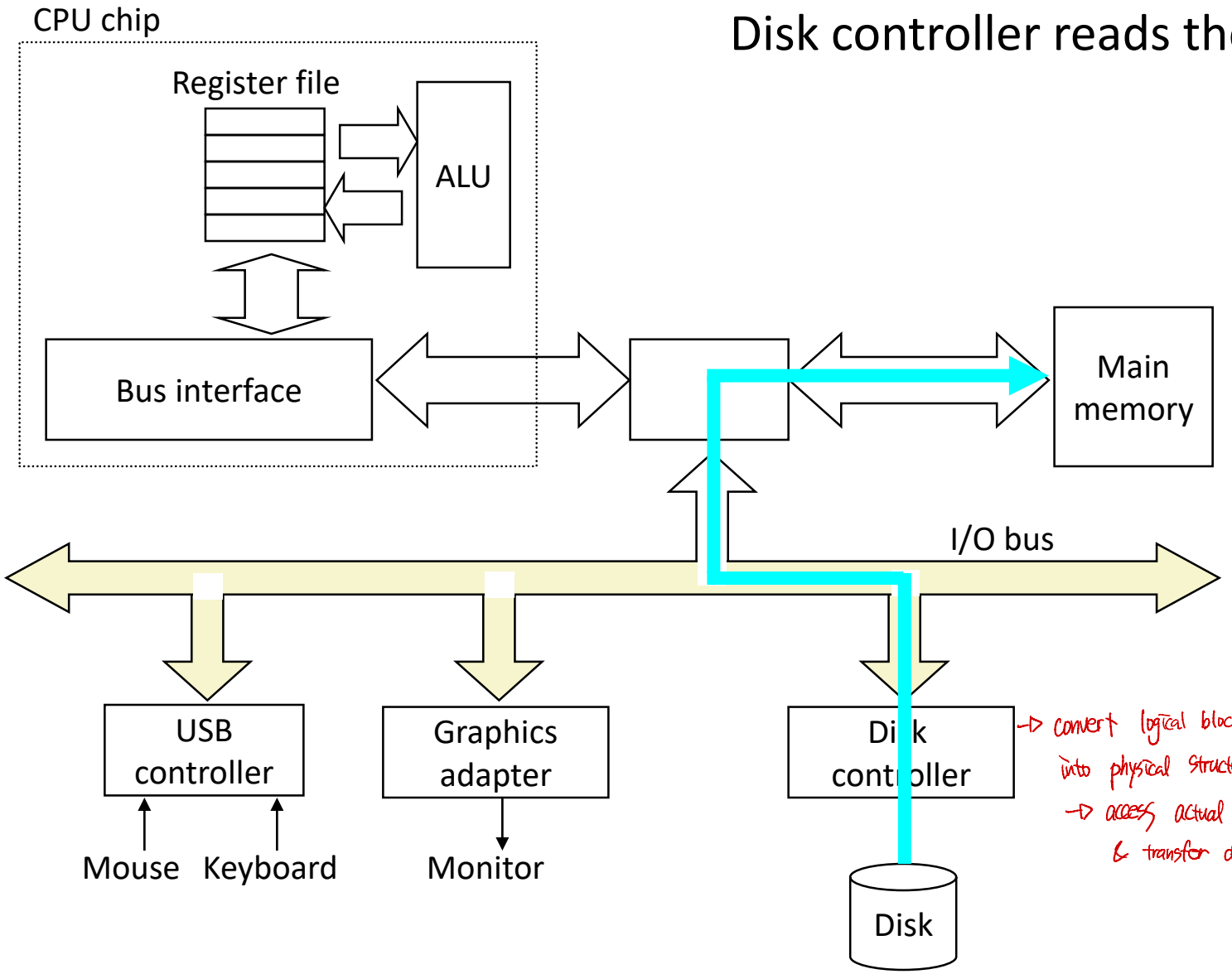
↳ logical block to access / destination memory address

1. communication between CPU & disk controller for read from disk operation

2. CPU use logical block number to manage data in disk, not physical info of disk. CPU doesn't care!
logical block number would be converted into physical structures by disk controller.

3. data in disk will be transferred into memory
→ CPU can read data from memory to register
not directly access between CPU – disk

Reading a Disk Sector (2)



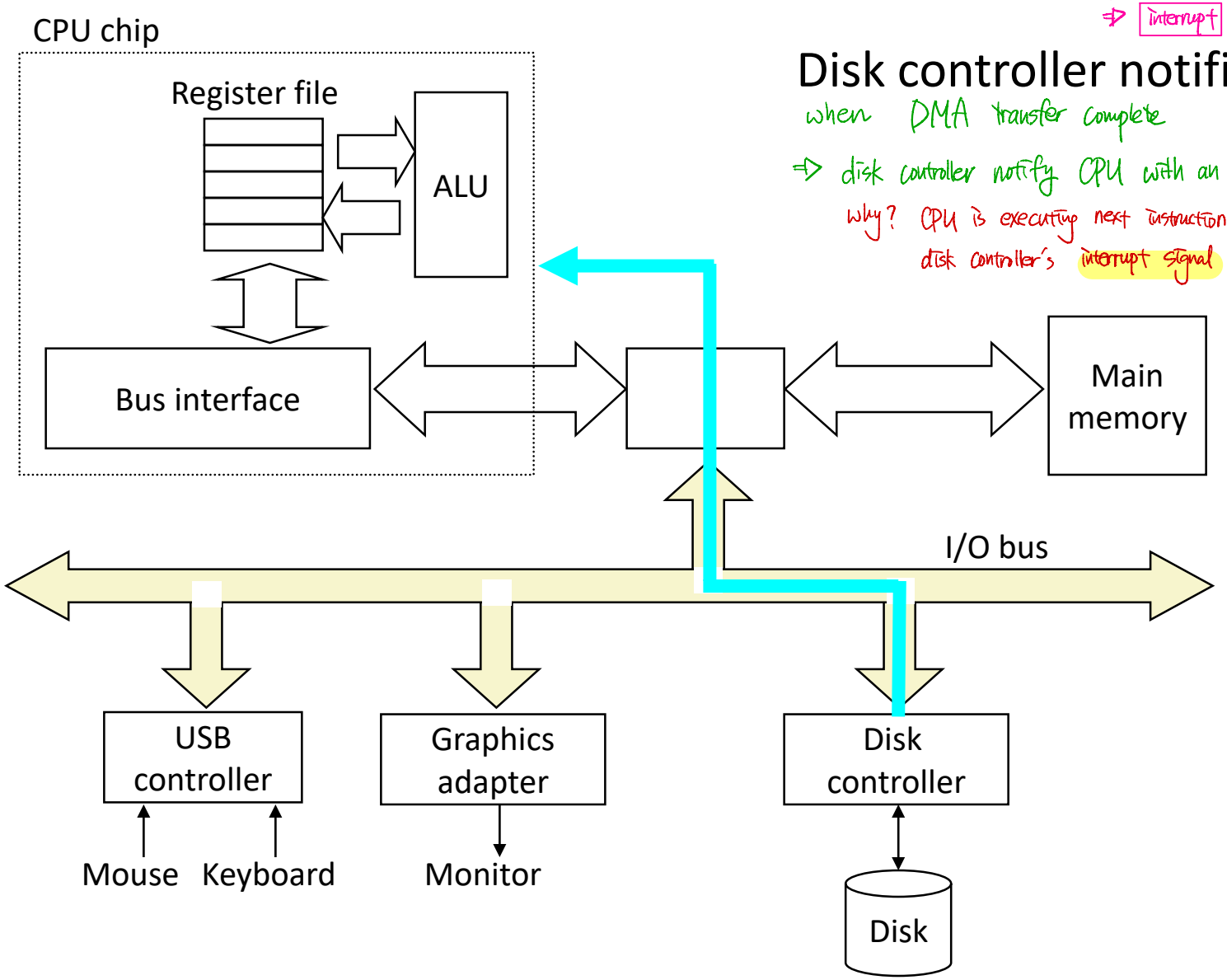
Disk controller reads the sector

Computer system has a separate device which has responsibility to access main memory
-> Direct Memory Access (DMA) without CPU
-> CPU just initiate transfer perform next instruction, no wait end of IO operation managed by DMA.

-> convert logical blocks for the data into physical structure.
-> access actual sector in the disk & transfer data into main memory **

Reading a Disk Sector (3)

why 'interrupt' is needed? { CPU cannot wait until IO instruction is finished. time ↑, inefficient
CPU initiates the command to proper controllers & CPU perform next instruction without waiting for previous command
⇒ need a way how CPU knows when ongoing IO instructions are finished.
⇒ interrupt occur, execute interrupt handler in CPU!

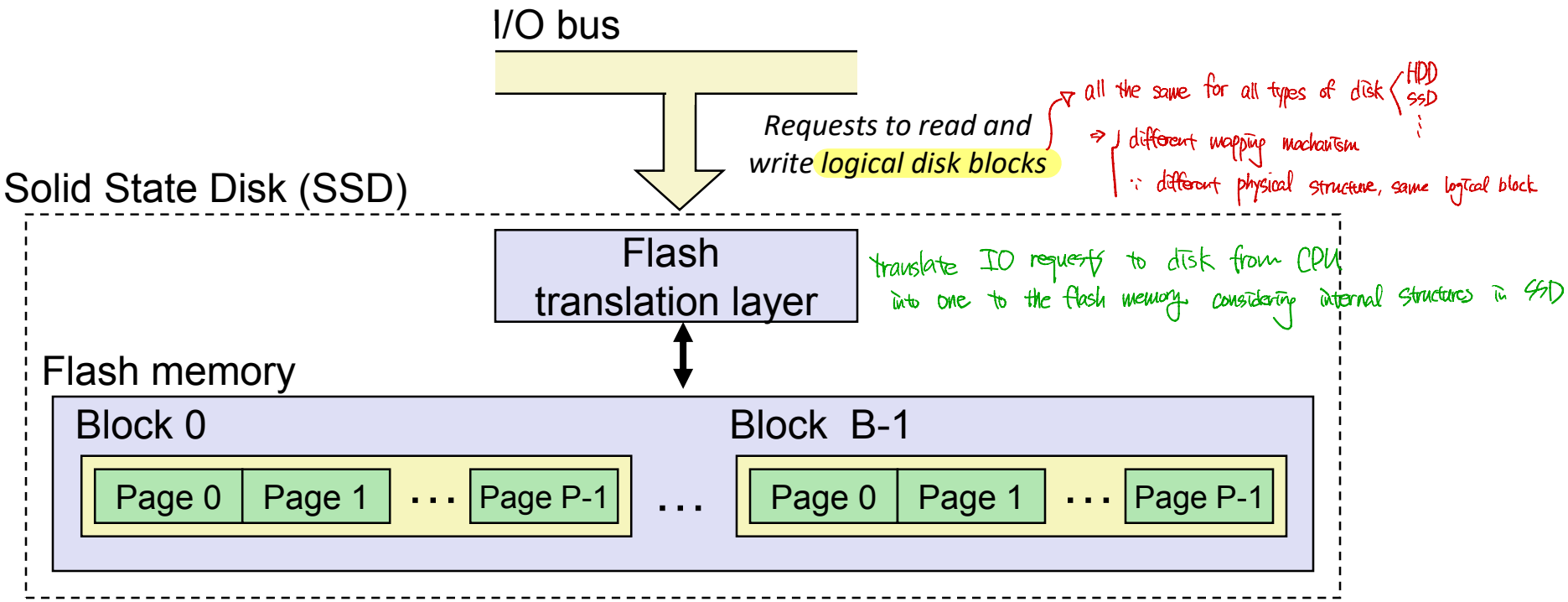


Disk controller notifies the CPU

when DMA transfer complete
⇒ disk controller notify CPU with an interrupt.
why? CPU is executing next instruction,
disk controller's interrupt signal to notice given IO instructions are finished.

DMA will perform read operation,
instead of CPU, from the disk
→ store data in main memory

Solid State Disks (SSDs)



- **Pages: 512KB to 4KB, Blocks: 32 to 128 pages**
- **A block wears out after about 100,000 repeated writes.**

SSD Performance Characteristics

HDD : Sequential access → only transfer time, seek time, rotational delay = 0
random access → seek time & rotational delay occur for every access
speed ⇒ sequential ⋙ random

Sequential read tput	550 MB/s	Sequential write tput	470 MB/s
Random read tput	365 MB/s	Random write tput	303 MB/s
Avg seq read time	50 us	Avg seq write time	60 us

- **Sequential access faster than random access**

  but the gap is { smaller than HDD / not significant } SSD support fast random access compared to HDD

- **Random writes are somewhat slower**
  - Erasing a block takes a long time (~1 ms)

    modify block page ⇒ require all other pages are copied to new block
    ⇒ overhead for write operation in SSD

Source: Intel SSD 730 product specification.

SSD Tradeoffs vs Rotating Disks in Hard Disk Drives

Advantages

- No moving parts → faster, less power, more rugged

Disadvantages

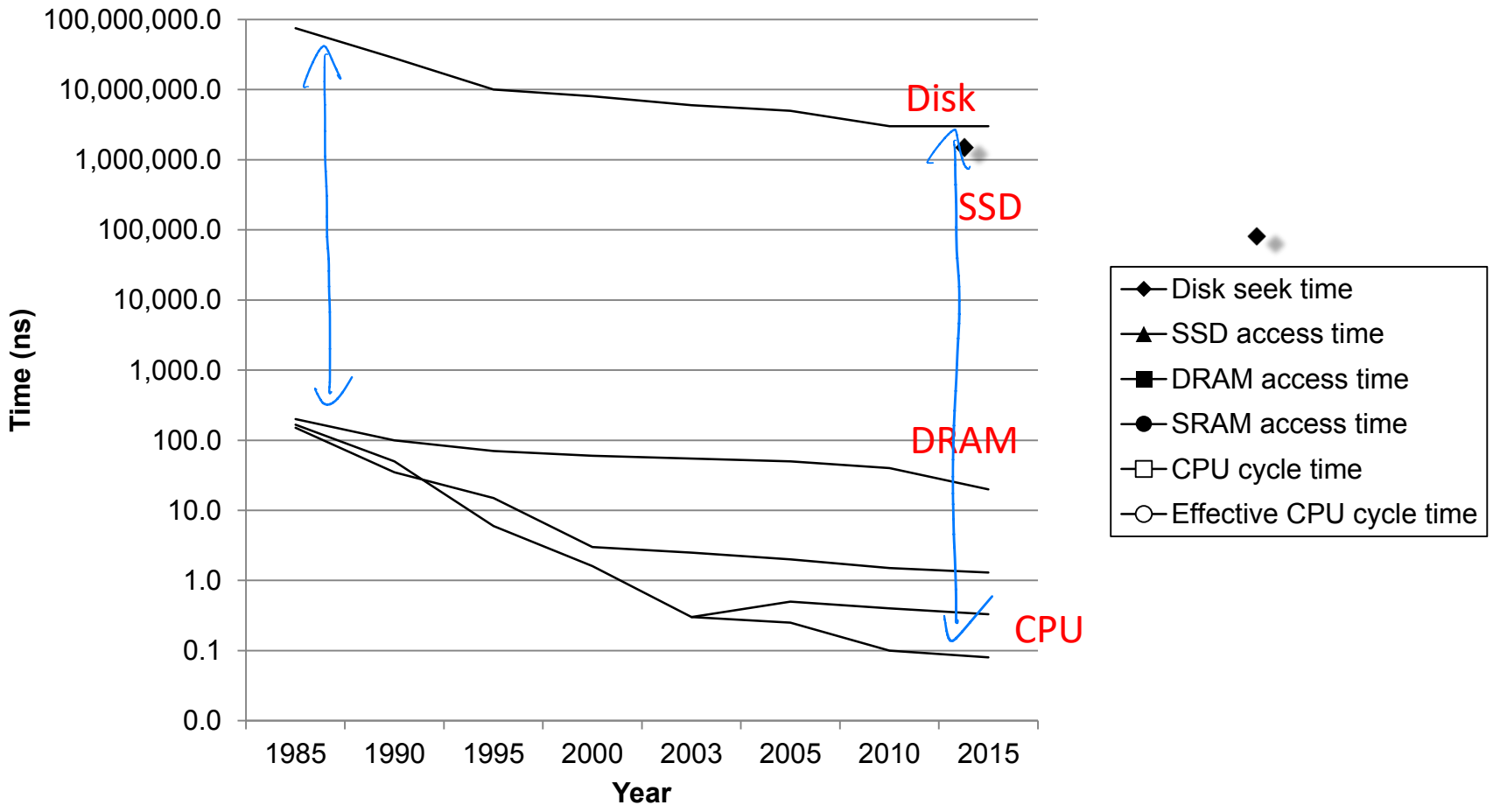
- Have the potential to wear out
  - E.g. Intel SSD 730 guarantees 128 petabyte (128×10^{15} bytes) of writes before they wear out
- In 2015, about 30 times more expensive per byte

Applications

- Smart phones, laptops
- Beginning to appear in desktops and servers

The CPU-Memory Gap

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



Locality to the Rescue!

mitigate performance gap between storages
optimize overall performance in computer systems

The key to bridging this CPU-Memory gap is a fundamental property of computer programs known as locality

Today

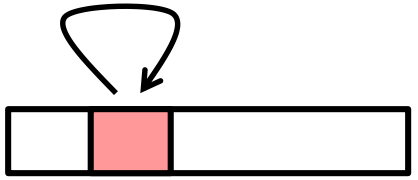
- Storage technologies and trends
- **Locality of reference**
- Caching in the memory hierarchy

Locality

- **Principle of Locality:** **Programs use data and instructions with addresses near or equal to those they have used recently**

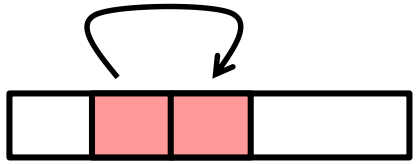
- **Temporal locality:**
  - Recently referenced items are likely to be referenced again in the near future

Criteria for temporal locality : time



- **Spatial locality:**
  - Items with nearby addresses tend to be referenced close together

Criteria for spatial locality : space



Locality Example

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- **Data references**
  - Reference array elements in succession.
  - Reference variable `sum` each iteration.

Spatial locality

Temporal locality

- **Instruction references**
  - Reference instructions in sequence.
  - Cycle through loop repeatedly.

Spatial locality

Temporal locality

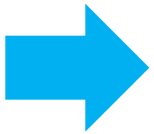
Practice: Qualitative Estimates of Locality

- **Claim:** **Being able to look at code and get a qualitative sense of its locality is a key skill for a professiona l programmer.**

- **Question:** **Does this function have good locality with respect to array** `a`**?** Good!

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```



Address	0	4	8	12	16	20
Contents	a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]
Access order	1	2	3	4	5	6

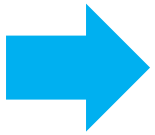
M = 2, N = 3

Practice: Qualitative Estimates of Locality (Continued)

- **Question:** Does this function have good locality with respect to array `a`? Bad !

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```



Address	0	4	8	12	16	20
Contents	a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]
Access order	1	3	5	2	4	6

M = 2, N = 3

Memory Hierarchies

- **Some fundamental and enduring properties of hardware and software**
  - The gap between CPU and main memory speed is widening.
  - Well-written programs tend to exhibit good locality.

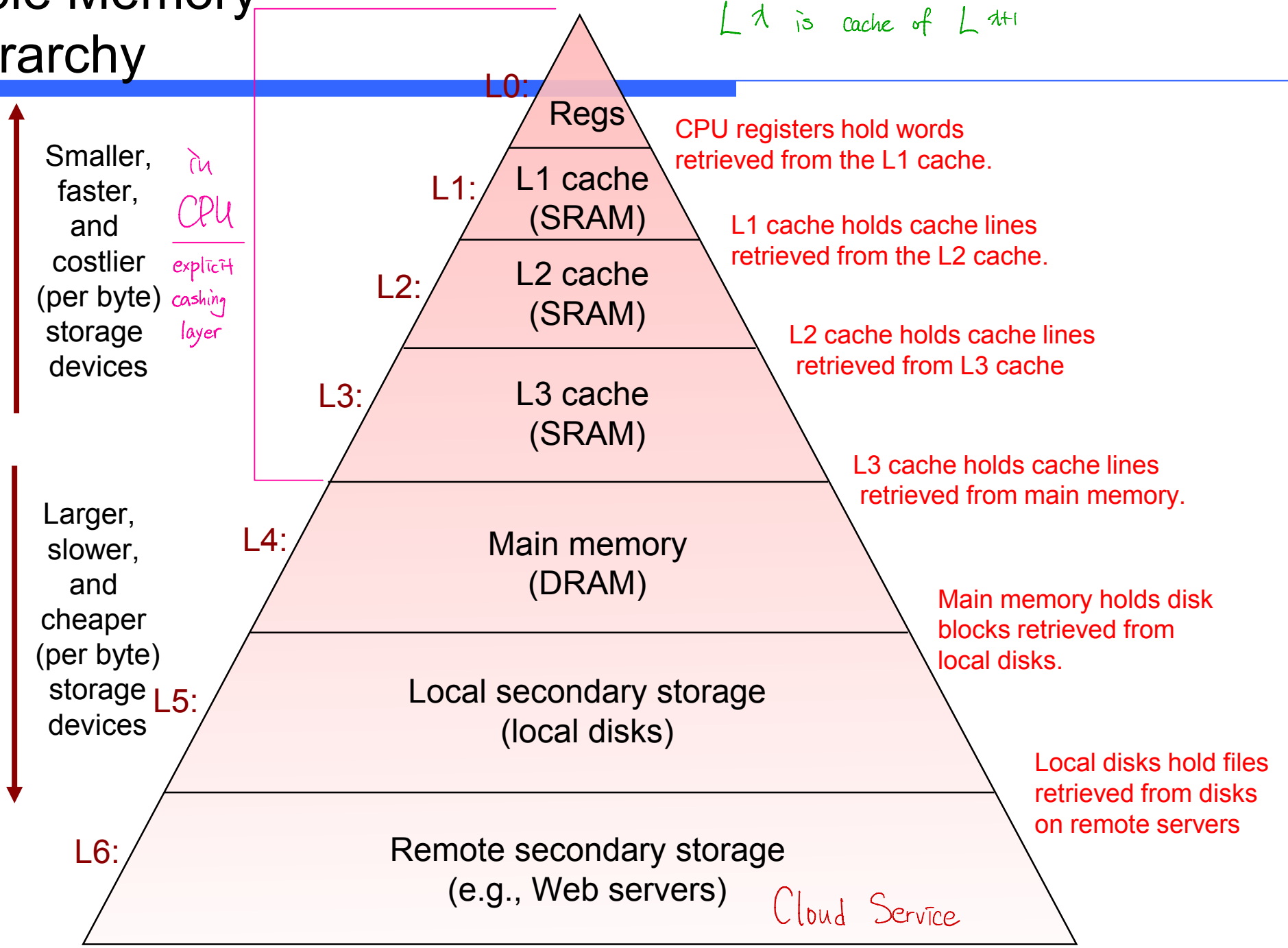
- **These fundamental properties complement each other beautifully.**

- **They suggest an approach for organizing memory and storage systems known as a memory hierarchy.**

Today

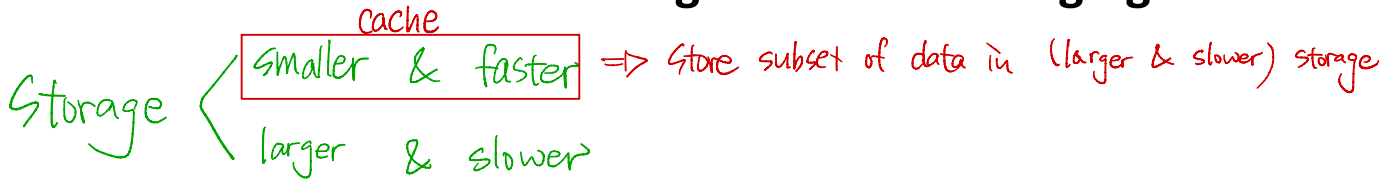
- Storage technologies and trends
- Locality of reference
- **Caching in the memory hierarchy**

Example Memory Hierarchy



Caches

- ***Cache:*** **A smaller and faster storage device as a staging area for a subset of the entire data**

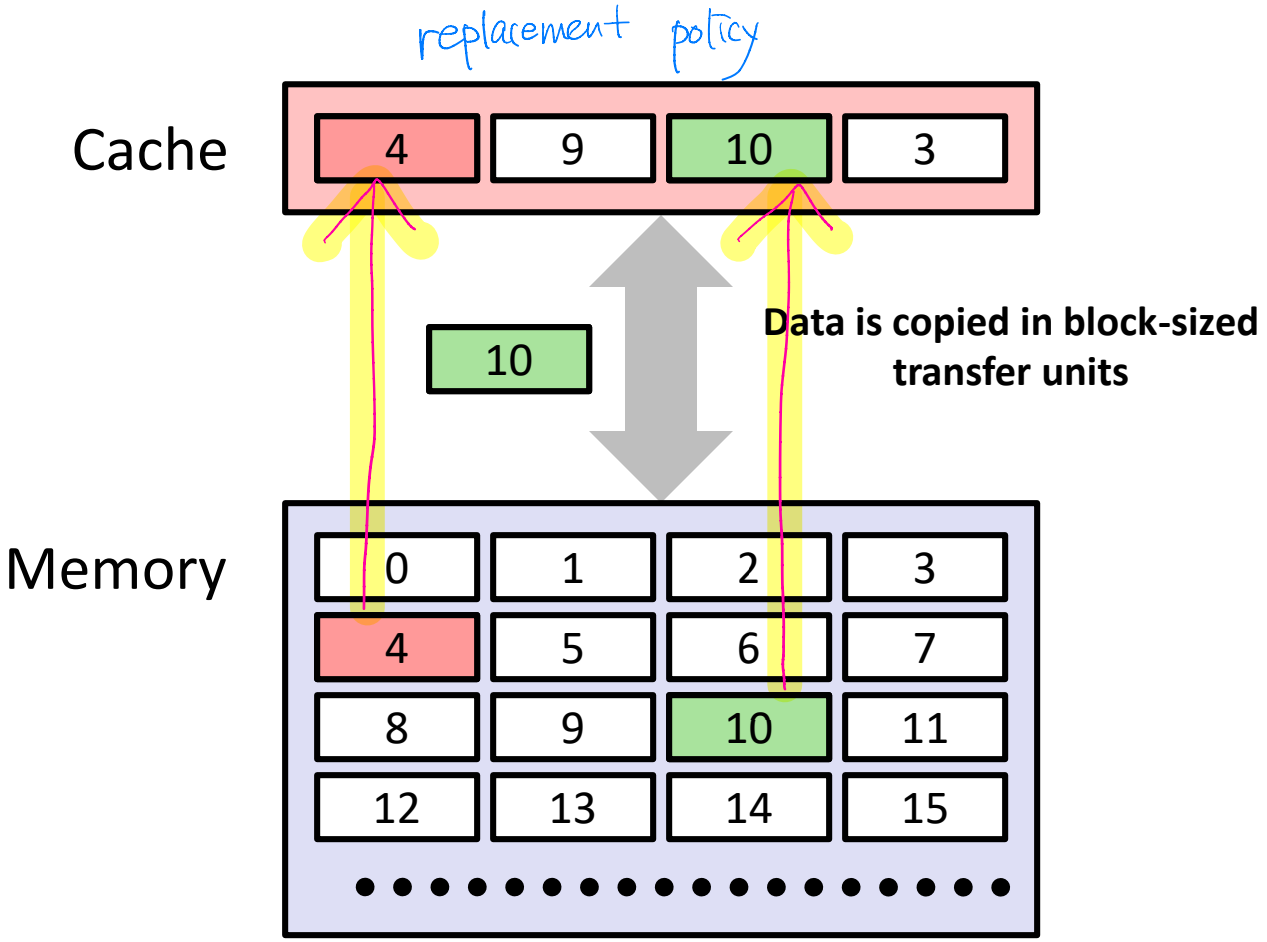


- **Fundamental idea of a memory hierarchy:**
  - For each *k*, device at level *k* serves as a cache for the device at level *k*+1

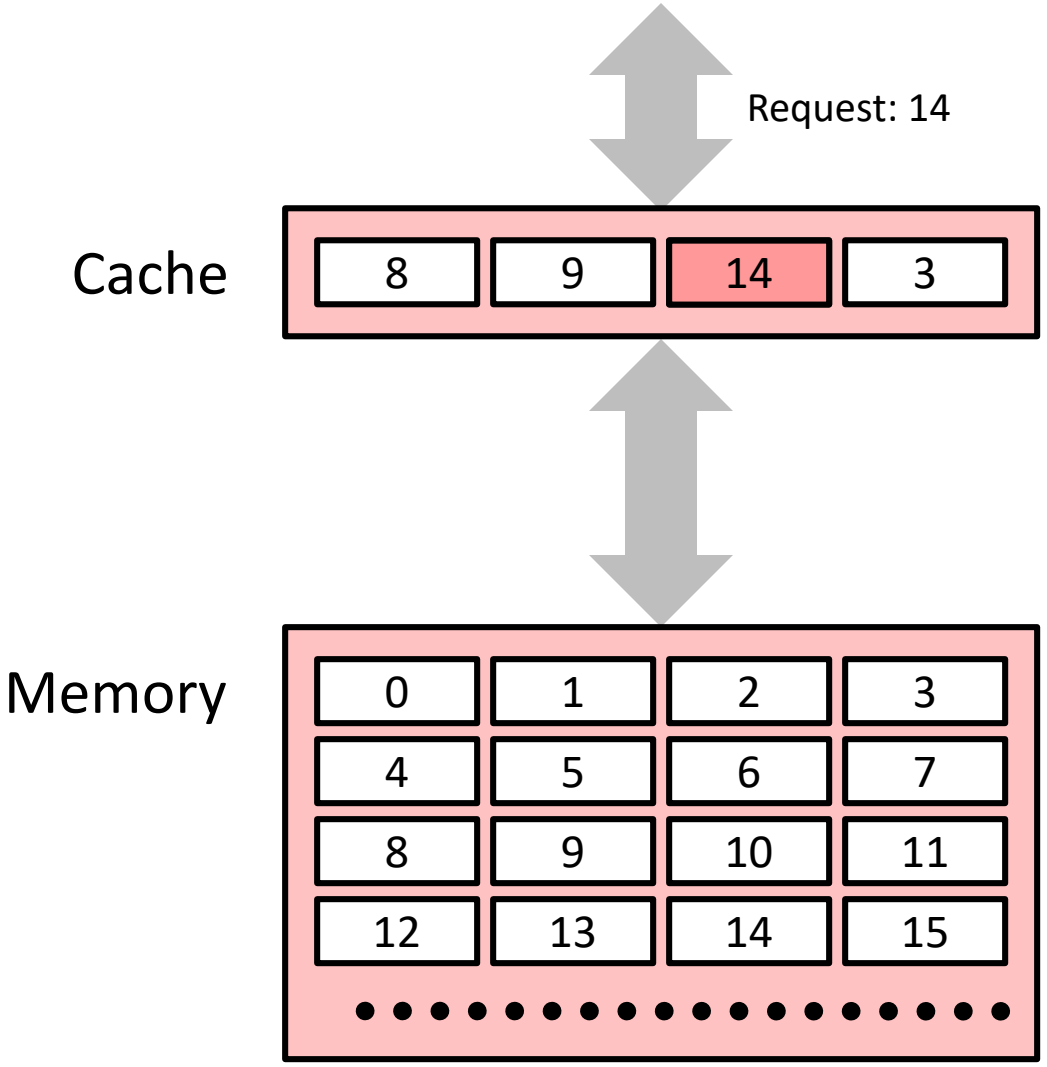
- **Why do memory hierarchies work?**
  - Because of locality, programs access the data at level *k* more often than the data at level *k*+1.
  - Thus, the storage at level *k*+1 can be slower.

- ***Big Idea:*** **The memory hierarchy creates a large pool of storage that costs the cheap storage near the bot tom, but that serves data at the rate of the fast storage near the top.**

General Cache Concepts



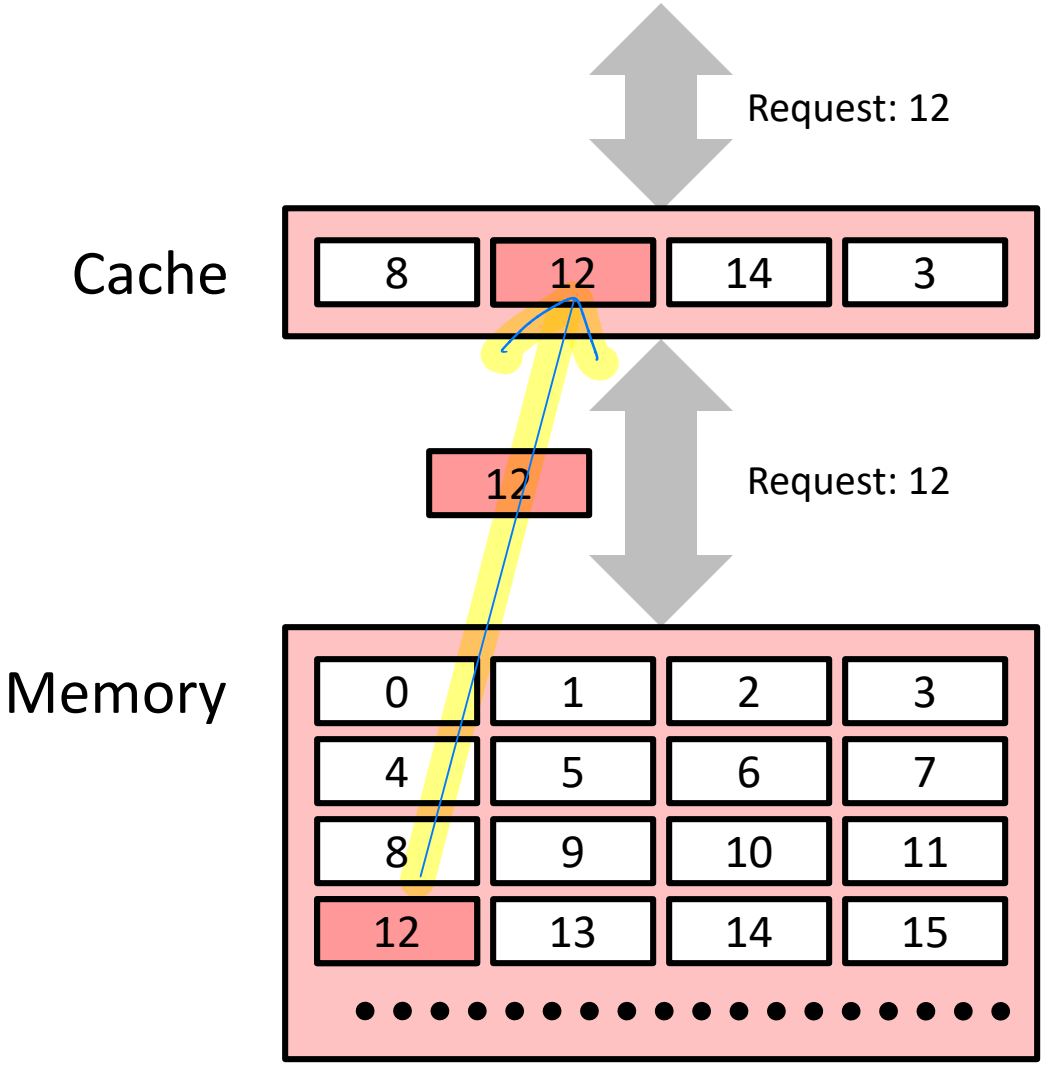
General Cache Concepts: Hit



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)

affect performance!
ratio of cache 'hit' is affected by two policies

General Caching Concepts: Types of Cache Misses

- **Cold (compulsory) miss** initial state of all accesses to data
  - This occurs because the cache is empty.

- **Conflict miss**
  - Most caches limit blocks to a small subset
    - E.g. Block i at level k+1 must be placed in block (i mod 4) at level k.
  - Multiple data objects all map to the same level k block
    - E.g. Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time.

- **Capacity miss** inherently occurence ⇒ capacity issue between level k & level k+1
  - Occurs when the set of active cache blocks (working set) is larger than the cache.

Examples of Caching in the Mem. Hierarchy

Cache Type	What is Cached?	Where is it Cached?	Latency (cycles)	Managed By
Registers	4-8 bytes words	CPU core	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware MMU
L1 cache	64-byte blocks	On-Chip L1	4	Hardware
L2 cache	64-byte blocks	On-Chip L2	10	Hardware
Virtual Memory	4-KB pages	Main memory	100	Hardware + OS
Buffer cache	Parts of files	Main memory	100	OS
Disk cache	Disk sectors	Disk controller	100,000	Disk firmware
Network buffer cache	Parts of files	Local disk	10,000,000	NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

on-chip storage → in CPU

in main memory

in disk controller

in local storage

in remote server disk

Summary

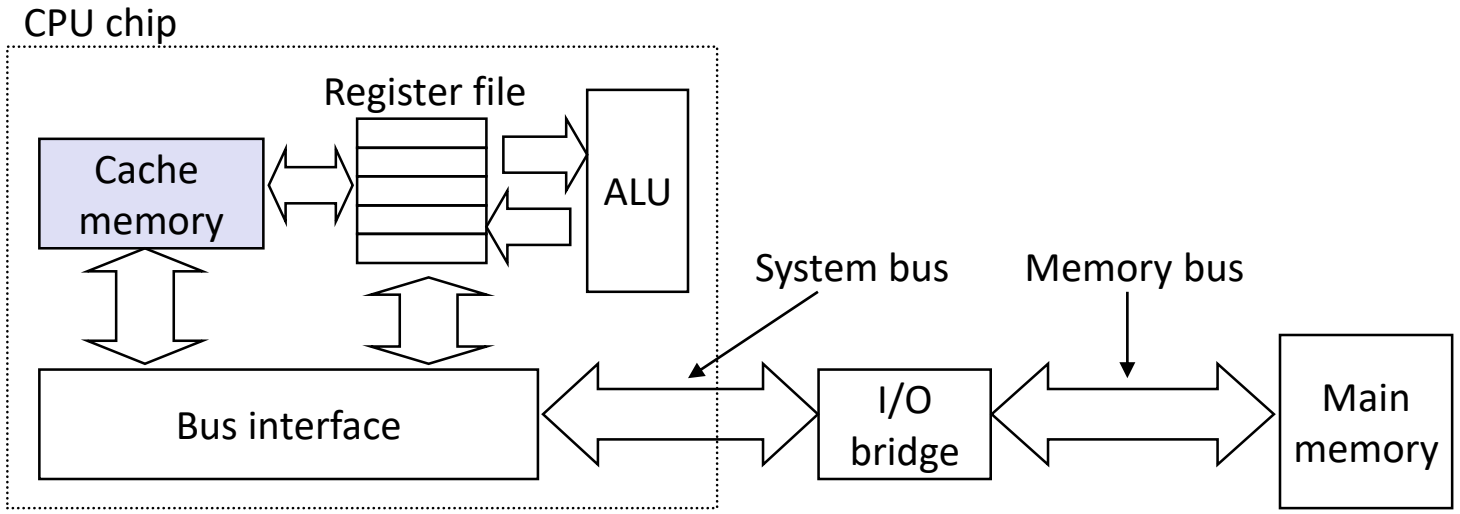
- The speed gap between CPU, memory and mass storage continues to widen.
- Well-written programs exhibit a property called *locality*.
- Memory hierarchies based on *caching* close the gap by exploiting locality.

Today

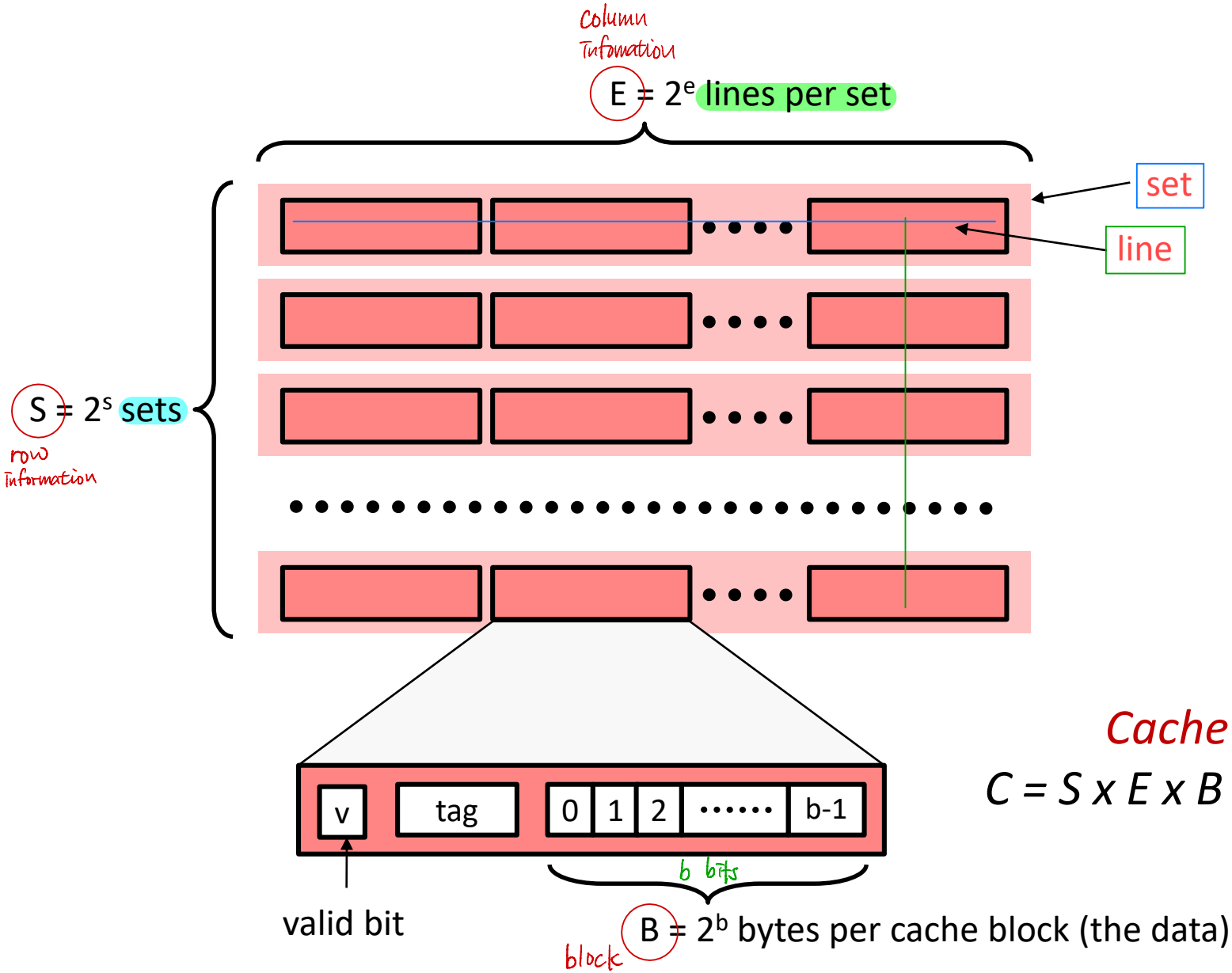
- Caching in the memory hierarchy
- **Cache memory organization and operation**
- Performance impact of caches
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality

Cache Memories

- **Cache memories are small, fast SRAM-based memories** ⇨ managed by HW
  - Hold frequently accessed blocks of main memory
- **CPU looks first for data in cache**
- **Typical system structure:**



General Cache Organization (S, E, B)

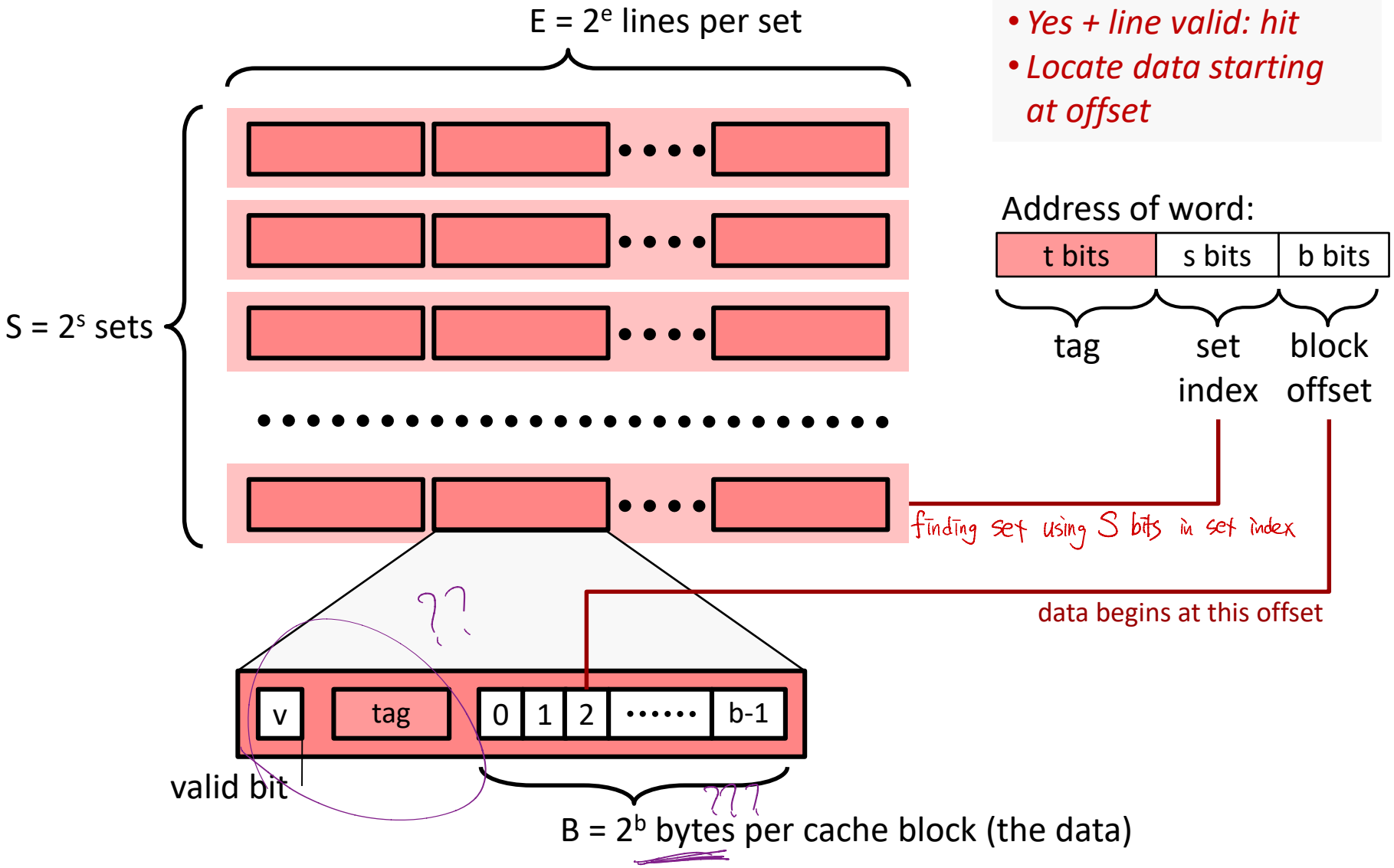


Cache size:

C = S x E x B data bytes

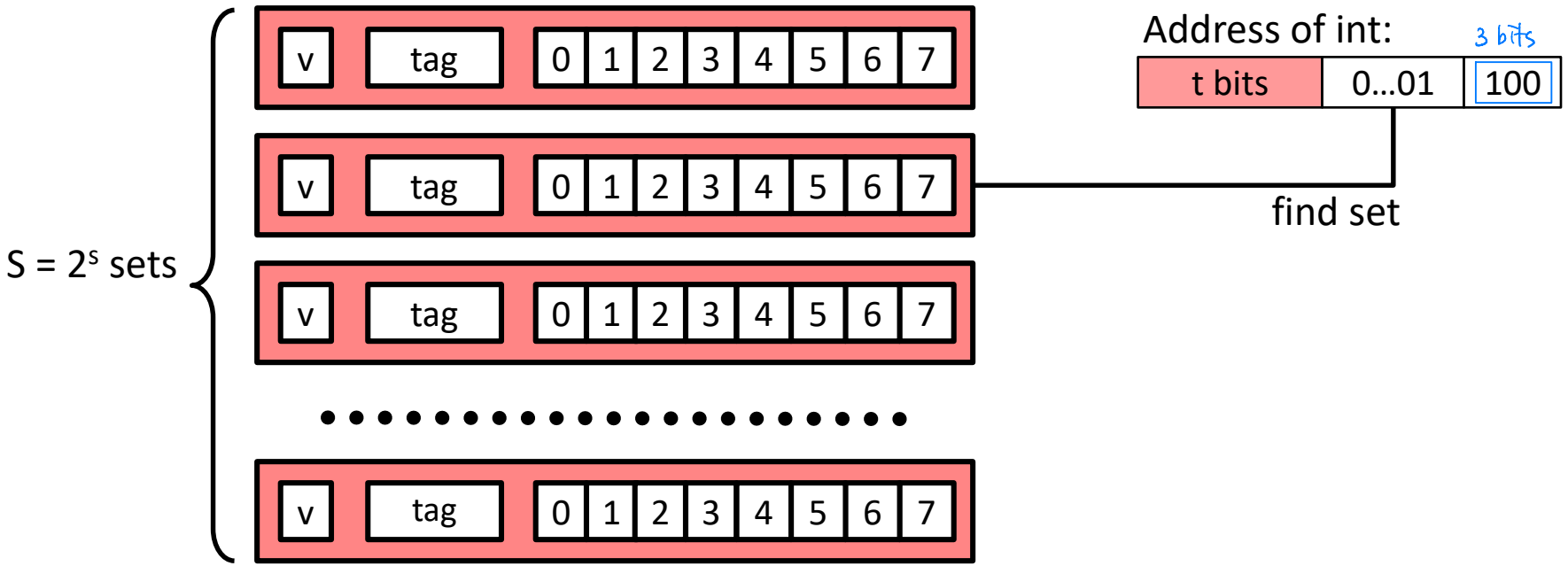
Cache Read

- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*
- *Locate data starting at offset*



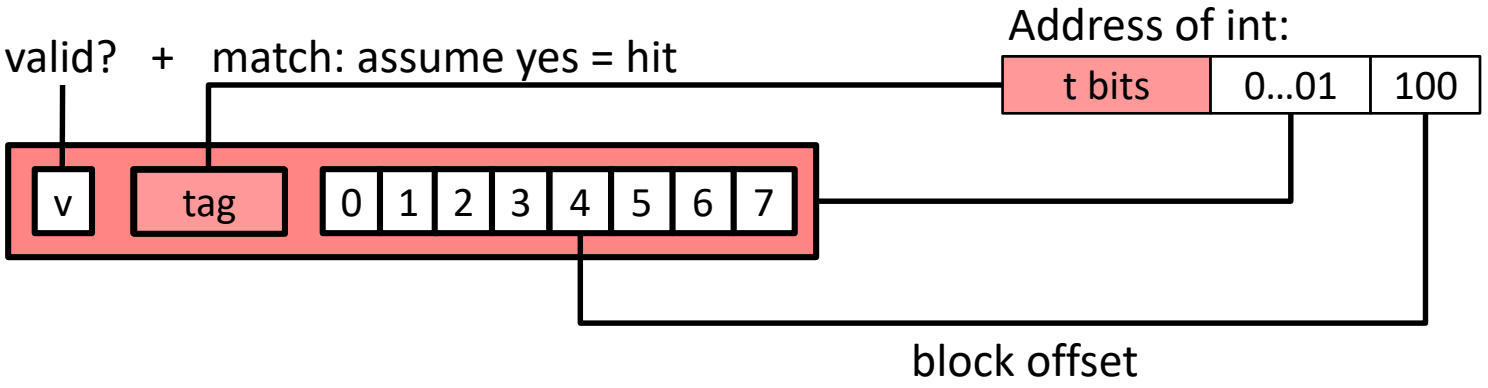
Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes
$= 2^3$



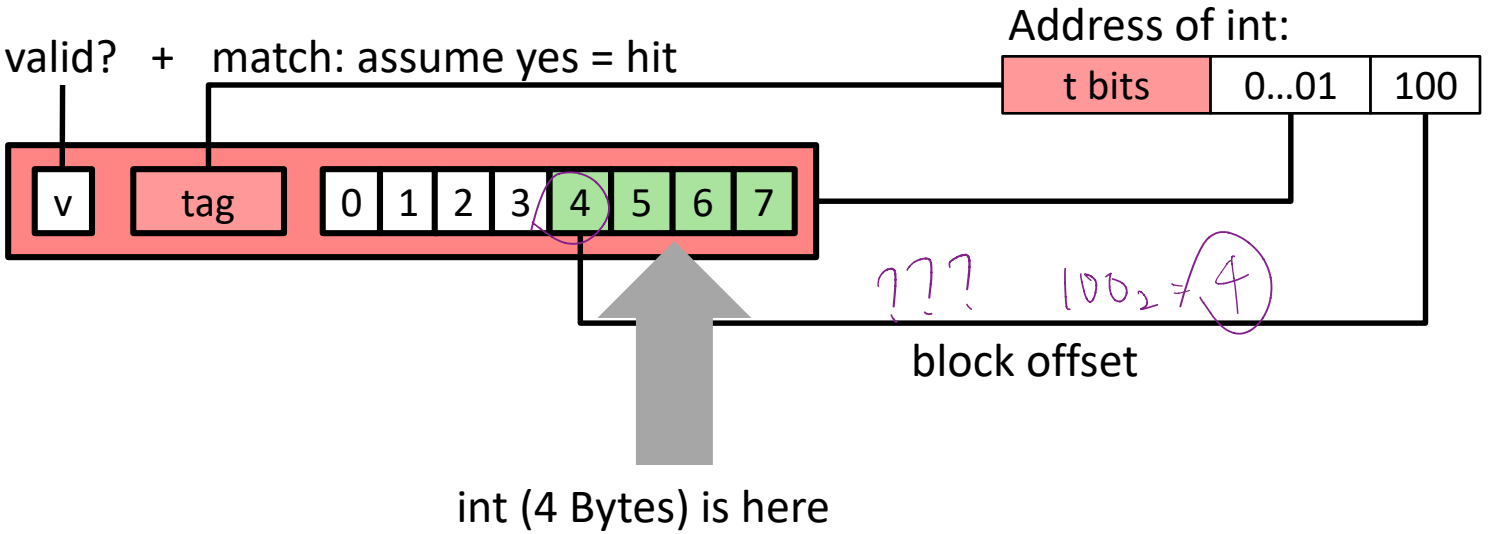
Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes



Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes



If tag doesn't match: old line is evicted and replaced

Direct-Mapped Cache Simulation

t=1	s=2	b=1
x	xx	x

M=16 bytes (4-bit addresses), B=2 bytes/block,
S=4 sets, E=1 Blocks/set

Address trace (reads, one byte per read):

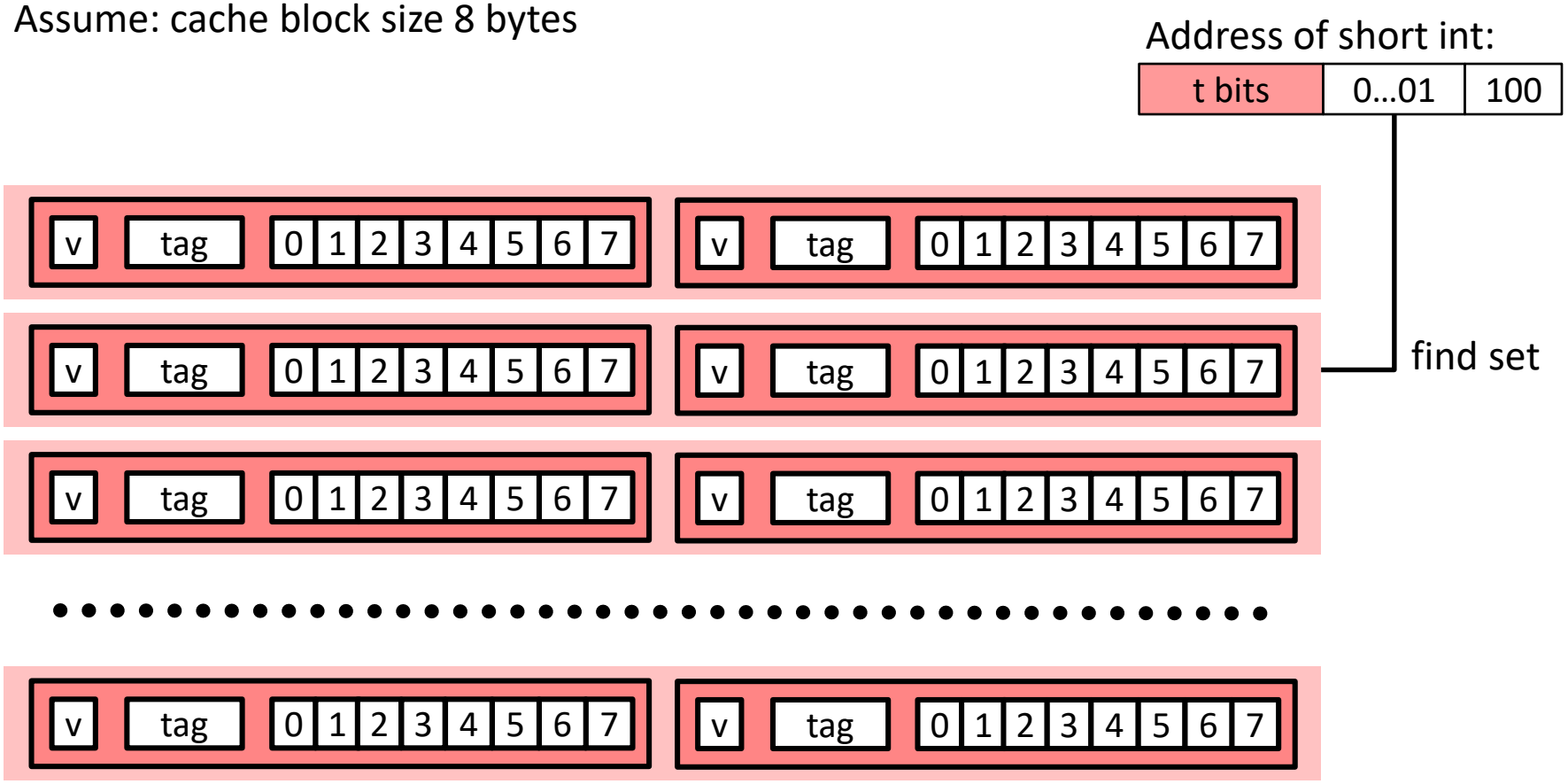
0	[$0\underline{00}0_2$],	miss
1	[$0\underline{00}1_2$],	hit
7	[$0\underline{11}1_2$],	miss
8	[$1\underline{00}0_2$],	miss
0	[$0\underline{00}0_2$]	miss

	v	Tag	Block
Set 0	1	0(1)	M[0-1] (M[8-9])
Set 1			
Set 2			
Set 3	1	0	M[6-7]

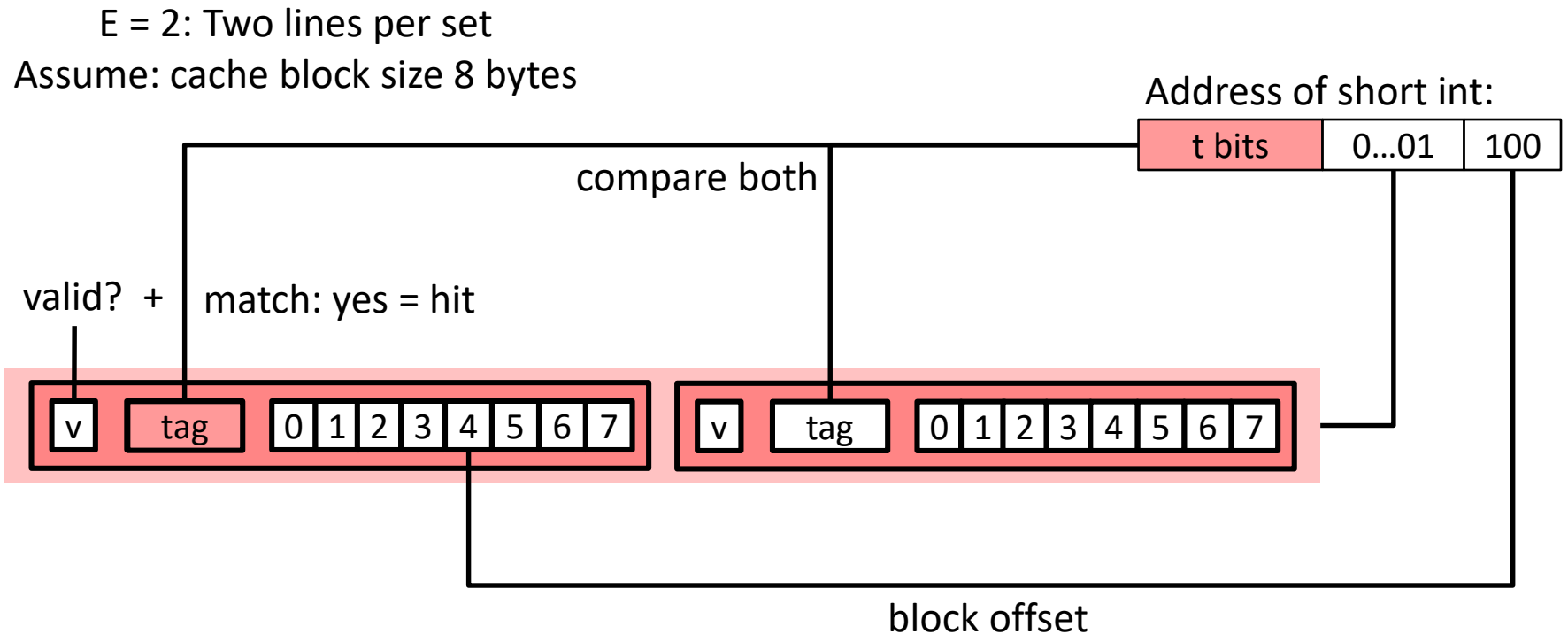
E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

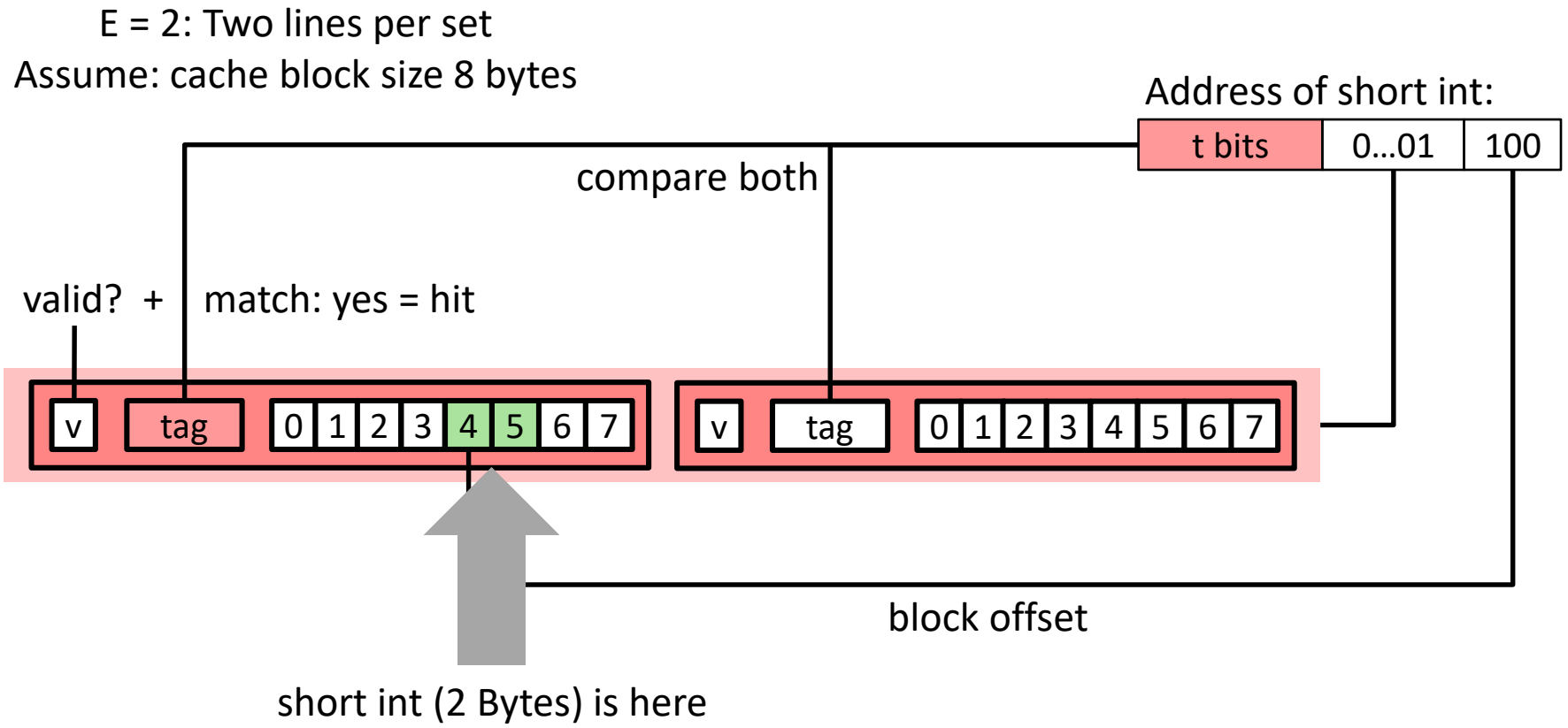
Assume: cache block size 8 bytes



E-way Set Associative Cache (Here: E = 2)



E-way Set Associative Cache (Here: E = 2)



No match:

- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), …

2-Way Set Associative Cache Simulation

t=2	s=1	b=1
xx	x	x

M=16 byte addresses, B=2 bytes/block,
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

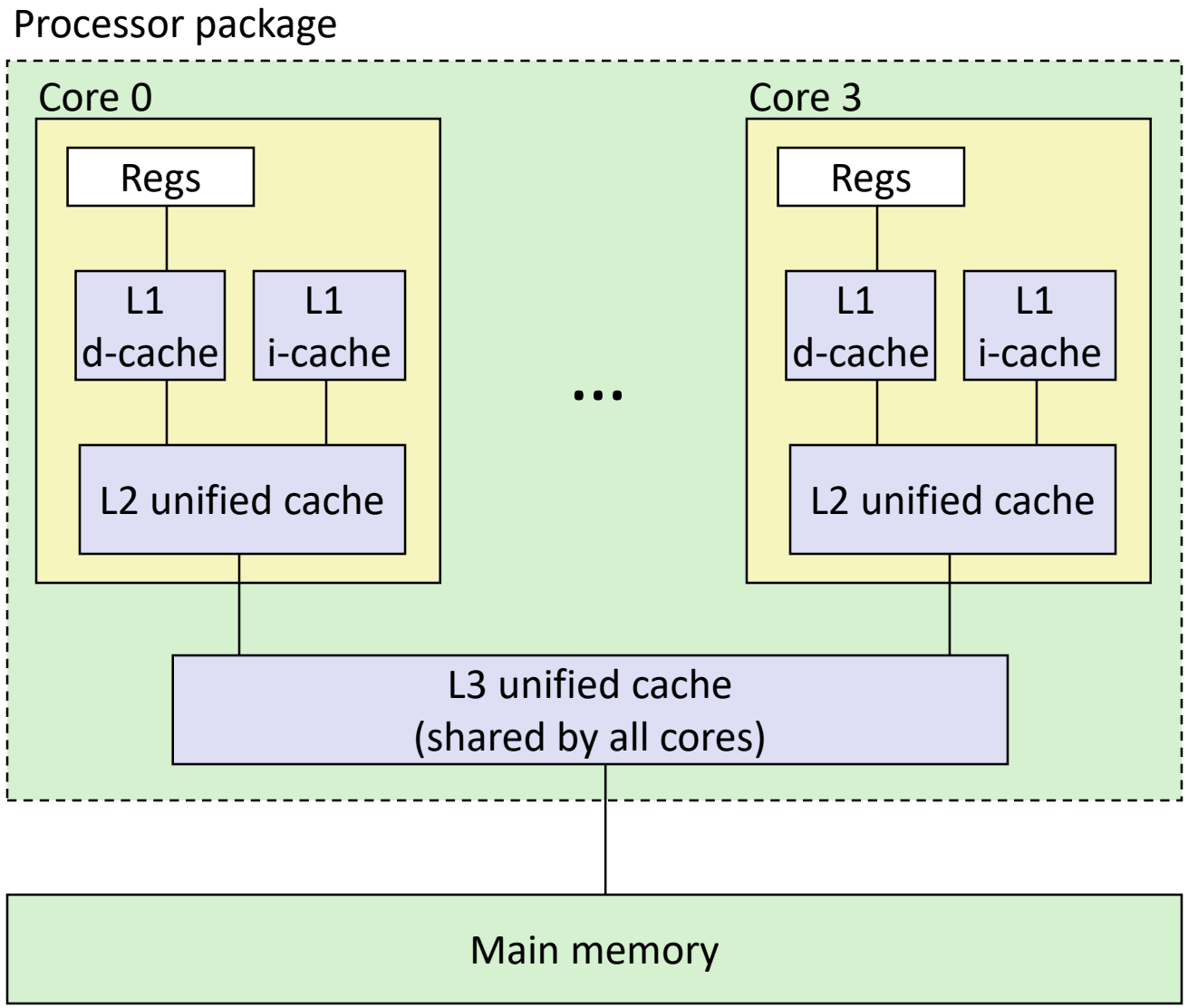
0	[00$\underline{0}$0$_2$],	miss
1	[00$\underline{0}$1$_2$],	hit
7	[01$\underline{1}$1$_2$],	miss
8	[10$\underline{0}$0$_2$],	miss
0	[00$\underline{0}$0$_2$]	hit

	v	Tag	Block
Set 0	1	00	M[0-1]
	1	10	M[8-9]
Set 1	1	01	M[6-7]
	0		

What about writes?

- **Multiple copies of data exist:** Sync problem
  - L1, L2, L3, Main Memory, Disk
- **What to do on a write-hit?** finding data block in cache
  - Write-through (write immediately to memory) Sync both when update occur
  - Write-back (defer write to memory until replacement of line) less write operation — efficient
    - Need a dirty bit (line different from memory or not)
- **What to do on a write-miss?**
  - Write-allocate (load into cache, update line in cache) more write operations! ∵ involve cache!
    - Good if more writes to the location follow
  - No-write-allocate (writes straight to memory, does not load into cache)
- **Typical** aggressive passive
  - Write-through + No-write-allocate
  - **Write-back + Write-allocate** passive aggressive

Intel Core i7 Cache Hierarchy



L1 i-cache and d-cache:
32 KB, 8-way,
Access: 4 cycles

L2 unified cache:
256 KB, 8-way,
Access: 10 cycles

L3 unified cache:
8 MB, 16-way,
Access: 40-75 cycles

Block size: 64 bytes for all caches.

Cache Performance Metrics

Miss Rate

- Fraction of memory references not found in cache (misses / accesses)

  = 1 – hit rate

- Typical numbers (in percentages):
  - 3-10% for L1
  - can be quite small (e.g., < 1%) for L2

Hit Time

- Time to deliver a line in the cache to the processor
- Typical numbers:
  - 4 clock cycle for L1
  - 10 clock cycles for L2

Miss Penalty

- Additional time required because of a miss
  - typically 50-200 cycles for main memory (Trend: increasing!)

Let’s think about those numbers

- **Huge difference between a hit and a miss**
  - Could be 100x, if just L1 and main memory

- **Would you believe 99% hits is twice as good as 97%?**
  - Consider:
    cache hit time of 1 cycle
    miss penalty of 100 cycles

  - Q. Average access time?
    97% hits: 1 cycle + 0.03 * 100 cycle = 4 cycle
    99% hits: 1 cycle + 0.01 * 100 cycle = 2 cycle

- **This is why “miss rate” is used instead of “hit rate”**

Writing Cache Friendly Code

- **Make the common case go fast**
  - E.g., inner loops

- **Minimize the misses in the inner loops**
  - Temporal locality -> repeated reference to variables
  - Spatial locality: Stride-1 reference patterns -> sequencial access to array in a consecutive way without skipping elements

Good performance => accessing order = storing order

Today

- Cache organization and operation
- **Performance impact of caches**
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality

The Memory Mountain

- **Read throughput (read bandwidth)**
  - Number of bytes read from memory per second (MB/s)

- **Memory mountain**
  - Measured read throughput as a function of spatial and temporal locality.

⇒ understand effect of spatial & temporal locality of caching

Memory Mountain Test Function

```
long data[MAXELEMS];  /* Global array to traverse */

/* test - Iterate over first "elems" elements of
 *      array “data” with stride of "stride",
 *      using 4x4 loop unrolling.
 */
int test(int elems, int stride) {
    long i, sx2=stride*2, sx3=stride*3, sx4=stride*4;
    long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
    long length = elems, limit = length - sx4;

    /* Combine 4 elements at a time */
    for (i = 0; i < limit; i += sx4) {
        acc0 = acc0 + data[i];
        acc1 = acc1 + data[i+stride];
        acc2 = acc2 + data[i+sx2];
        acc3 = acc3 + data[i+sx3];
    }

    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc0 = acc0 + data[i];
    }
    return ((acc0 + acc1) + (acc2 + acc3));
}
```

mountain/mountain.c

Call `test()` with many combinations of `elems` and `stride`.

For each `elems` and `stride`:

1. Call `test()` once to warm up the caches.

2. Call `test()` again and measure the read throughput (MB/s)

Practice: Design Memory Mountain

- **Using the previous test function, design the functions for calculating the memory mountain**
  - Use the clock() to measure the spent time for the target operations

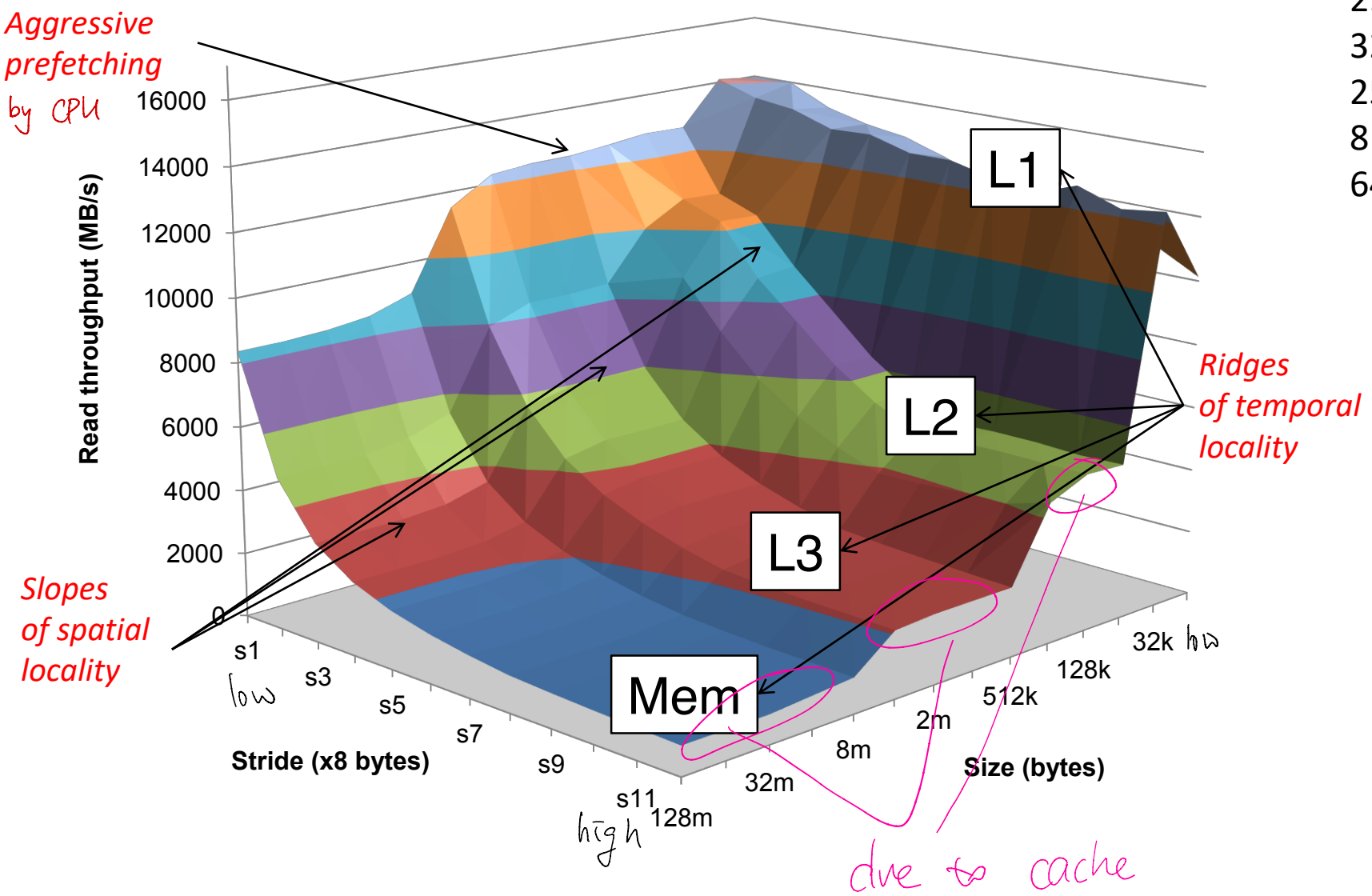
```
clock_t start, end;
double cpu_time_used;

start = clock();
... /* Do the work. */
end = clock();
cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
```

  - Watch out for the following condition
    - `Call test() once to warm up the caches.`
  - You can assume any sample combinations for the parameters
  - You can program just pseudo code by filling the necessary logics for calculating the memory mountain
    - Not necessary to make the program in C

The Memory Mountain

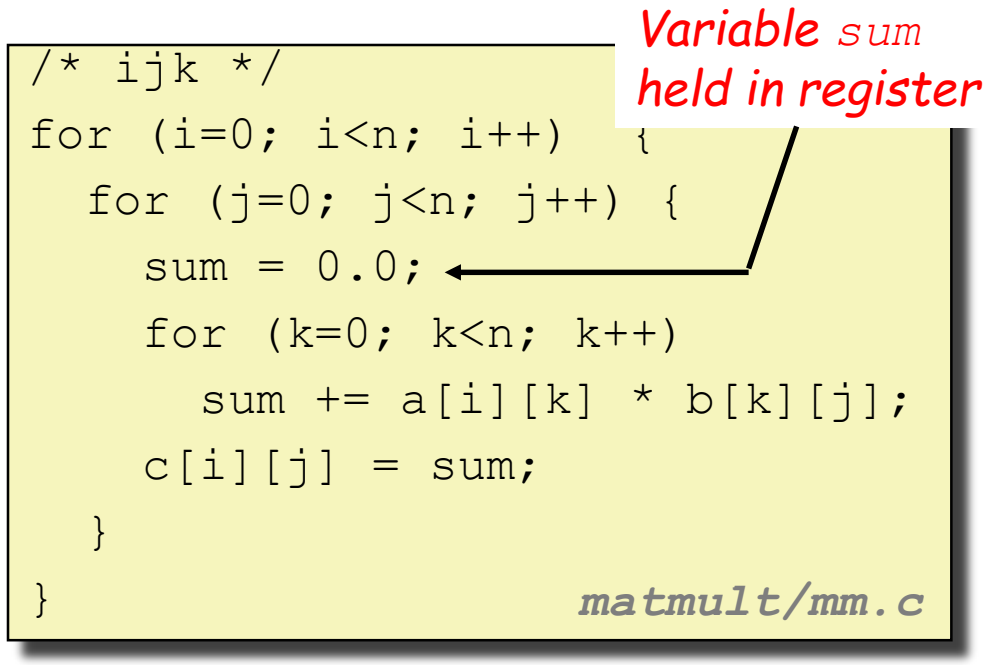
Core i7 Haswell
2.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size



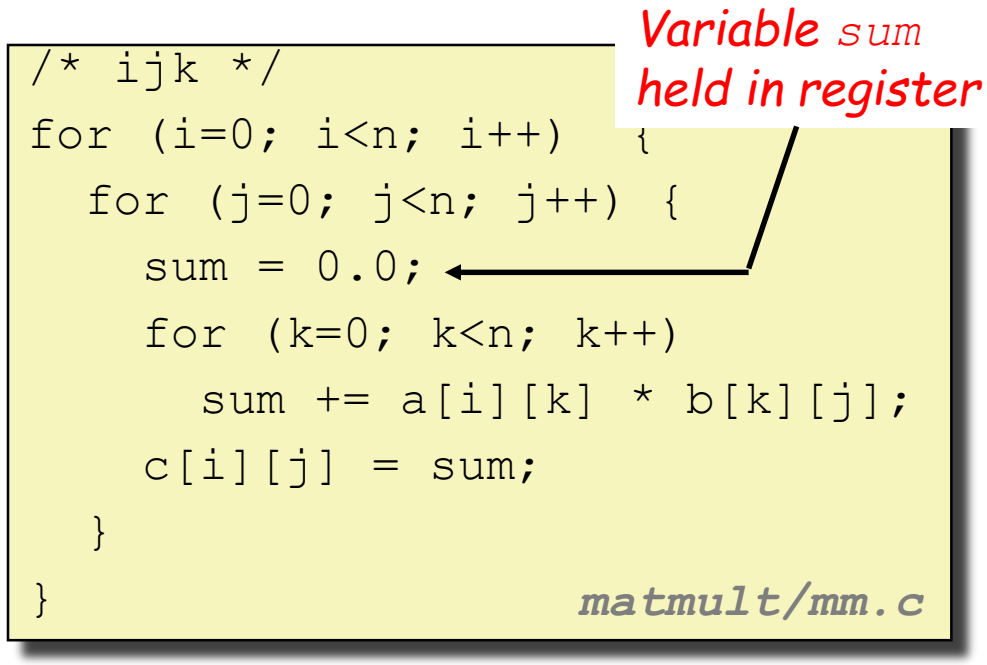
Today

- Cache organization and operation
- Performance impact of caches
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality

Matrix Multiplication Example

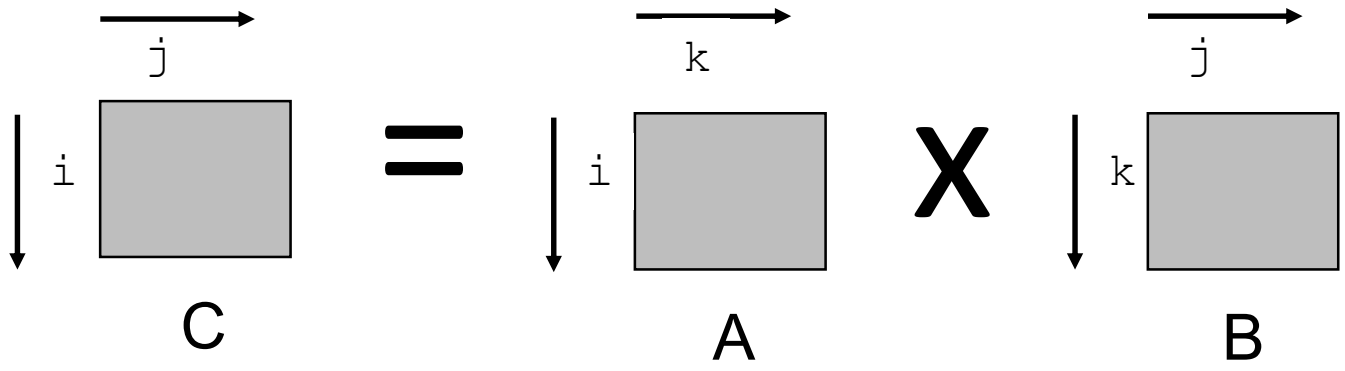


Practice: Matrix Multiplication Example



Miss Rate Analysis for Matrix Multiply

- **Assume:**
  - Block size = 32B (big enough for four doubles)
  - Matrix dimension (N) is very large
    - Approximate 1/N as 0.0
  - Cache is not even big enough to hold multiple rows
- **Analysis Method:**
  - Look at access pattern of inner loop



Layout of C Arrays in Memory (review)

- **C arrays allocated in row-major order**

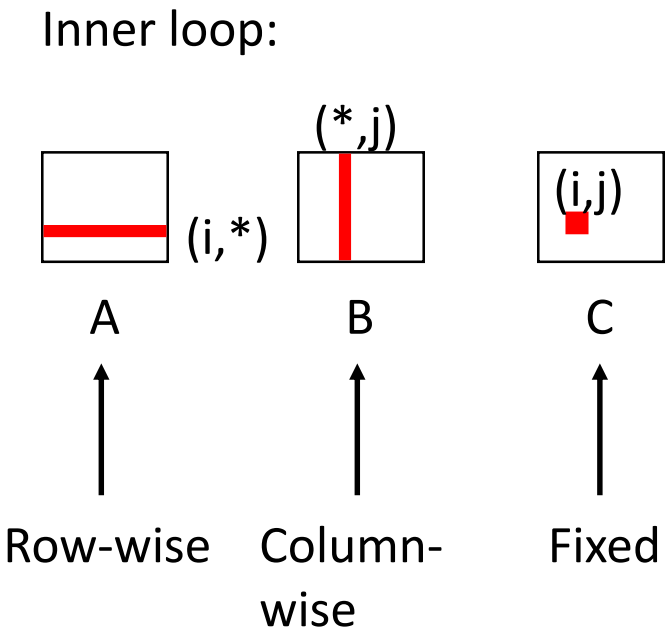
- **Stepping through columns in one row:**
  - `for (i = 0; i < N; i++)`
    `sum += a[0][i];`
  - accesses successive elements
  - if block size (B) > sizeof(a_{ij}) bytes, exploit spatial locality
    - miss rate = sizeof(a_{ij}) / B

- **Stepping through rows in one column:**
  - `for (i = 0; i < n; i++)`
    `sum += a[i][0];`
  - accesses distant elements
  - no spatial locality!
    - miss rate = 1 (i.e. 100%)

Matrix Multiplication (ijk)

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

matmult/mm.c



Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0

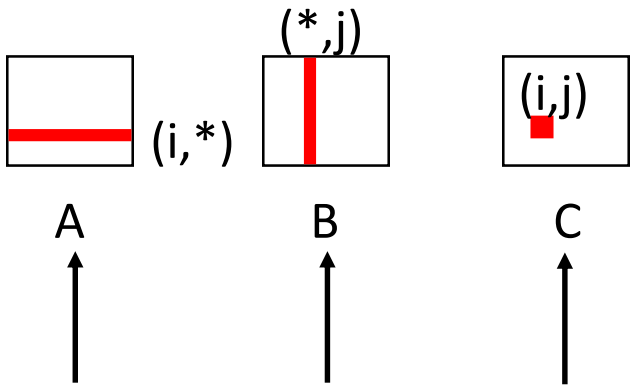
8 byte / double
32 byte block size

Matrix Multiplication (jik)

```
/* jik */
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum
  }
}
```

matmult/mm.c

Inner loop:



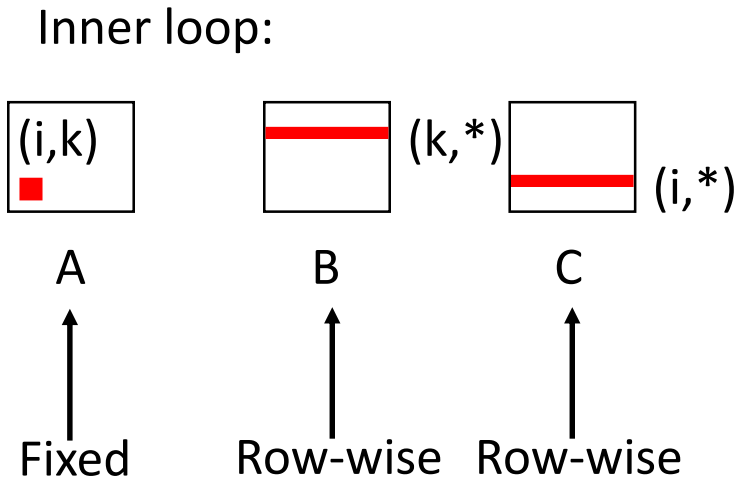
Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (kij)

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

matmult/mm.c



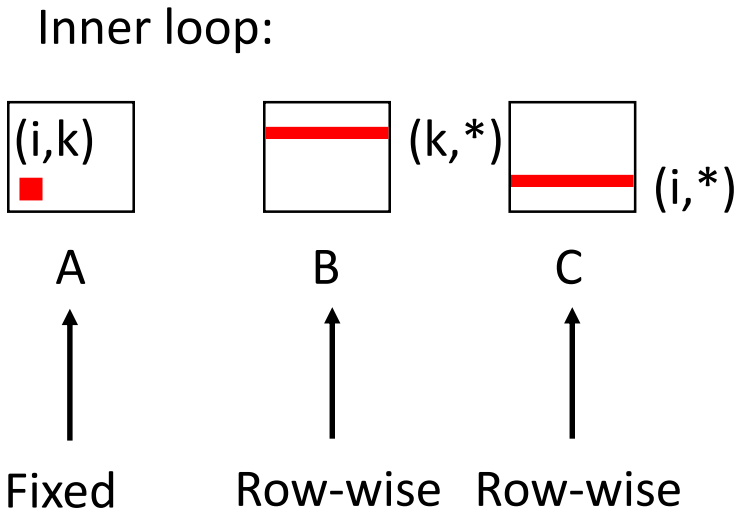
Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (ikj)

```
/* ikj */
for (i=0; i<n; i++) {
  for (k=0; k<n; k++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

matmult/mm.c



Misses per inner loop iteration:

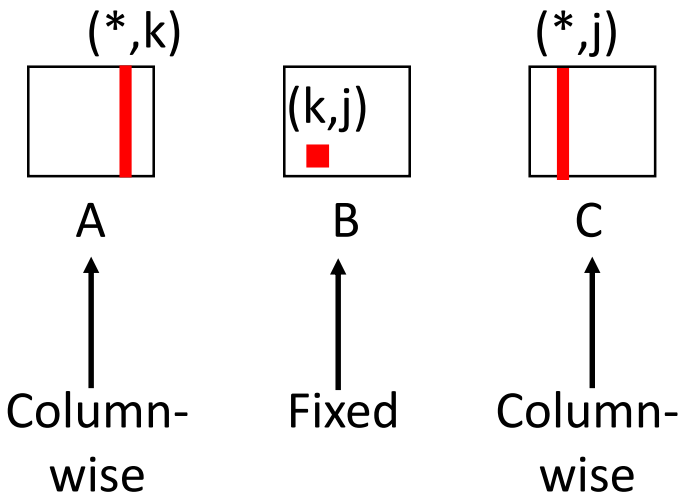
A	B	C
0.0	0.25	0.25

Matrix Multiplication (jki)

```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

matmult/mm.c

Inner loop:



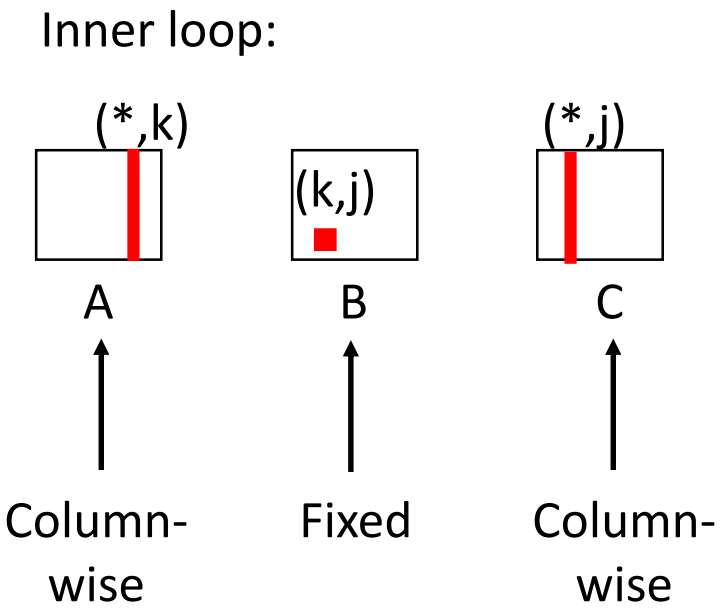
Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Matrix Multiplication (kji)

```
/* kji */
for (k=0; k<n; k++) {
  for (j=0; j<n; j++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

matmult/mm.c



Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Core i7 Matrix Multiply Performance