

# COMP201

## Computer Systems & Programming

### Lecture #30 – Cache Memories



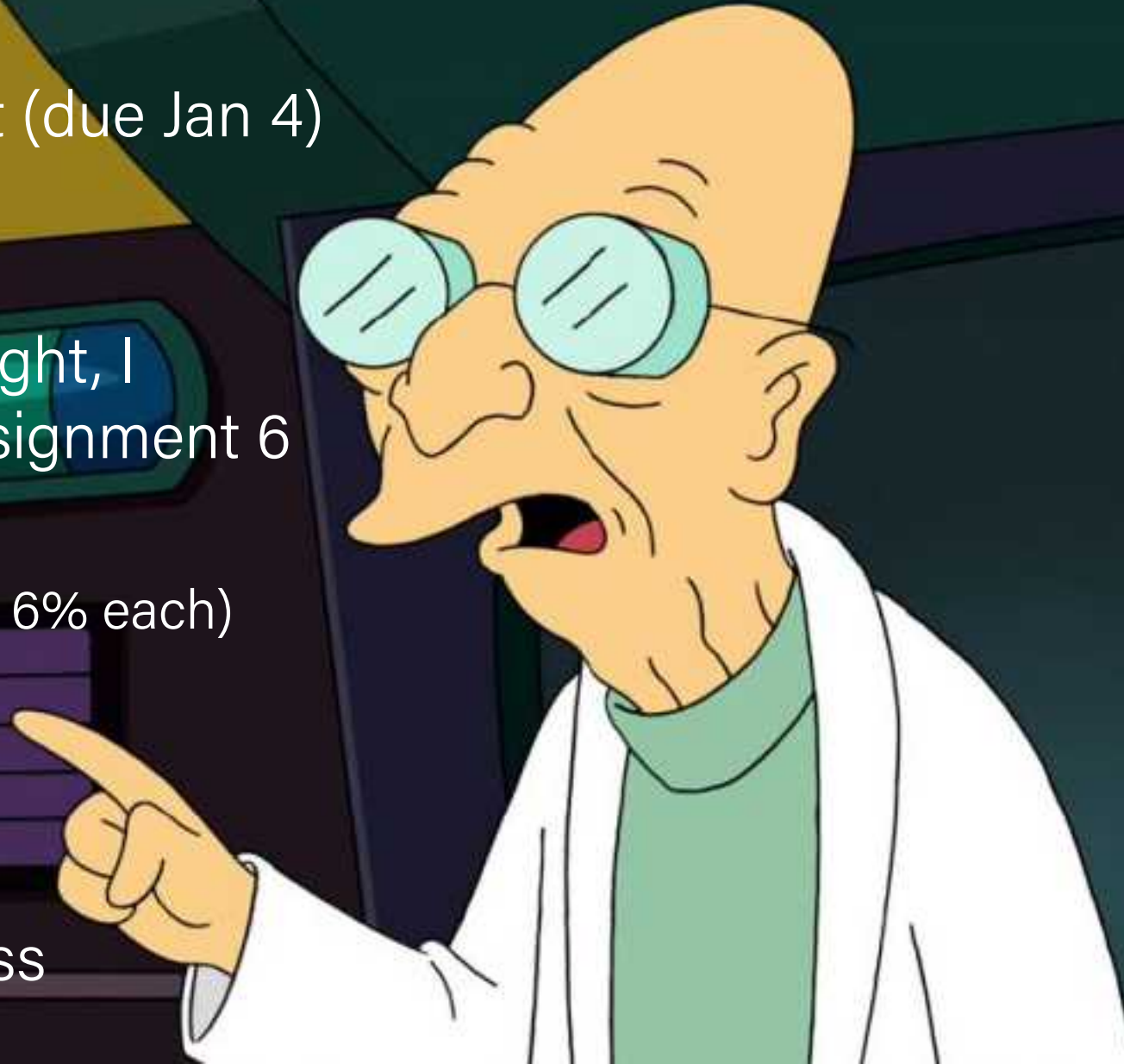
**KOÇ  
UNIVERSITY**

Aykut Erdem // Koç University // Fall 2020



# Good news, everyone!

- Assignment 5 is out (due Jan 4)
- No labs this week!
- After a second thought, I decided to drop Assignment 6
  - Assignment 0 2%,  
Assignment 1-5 5% 6% each)
- I finished grading your exam papers though I want to give it one more pass



# Recap

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

# Plan for Today

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

**Disclaimer:** Slides for this lecture were borrowed from  
—Randal E. Bryant and David R. O'Hallaron's CMU 15-213 class

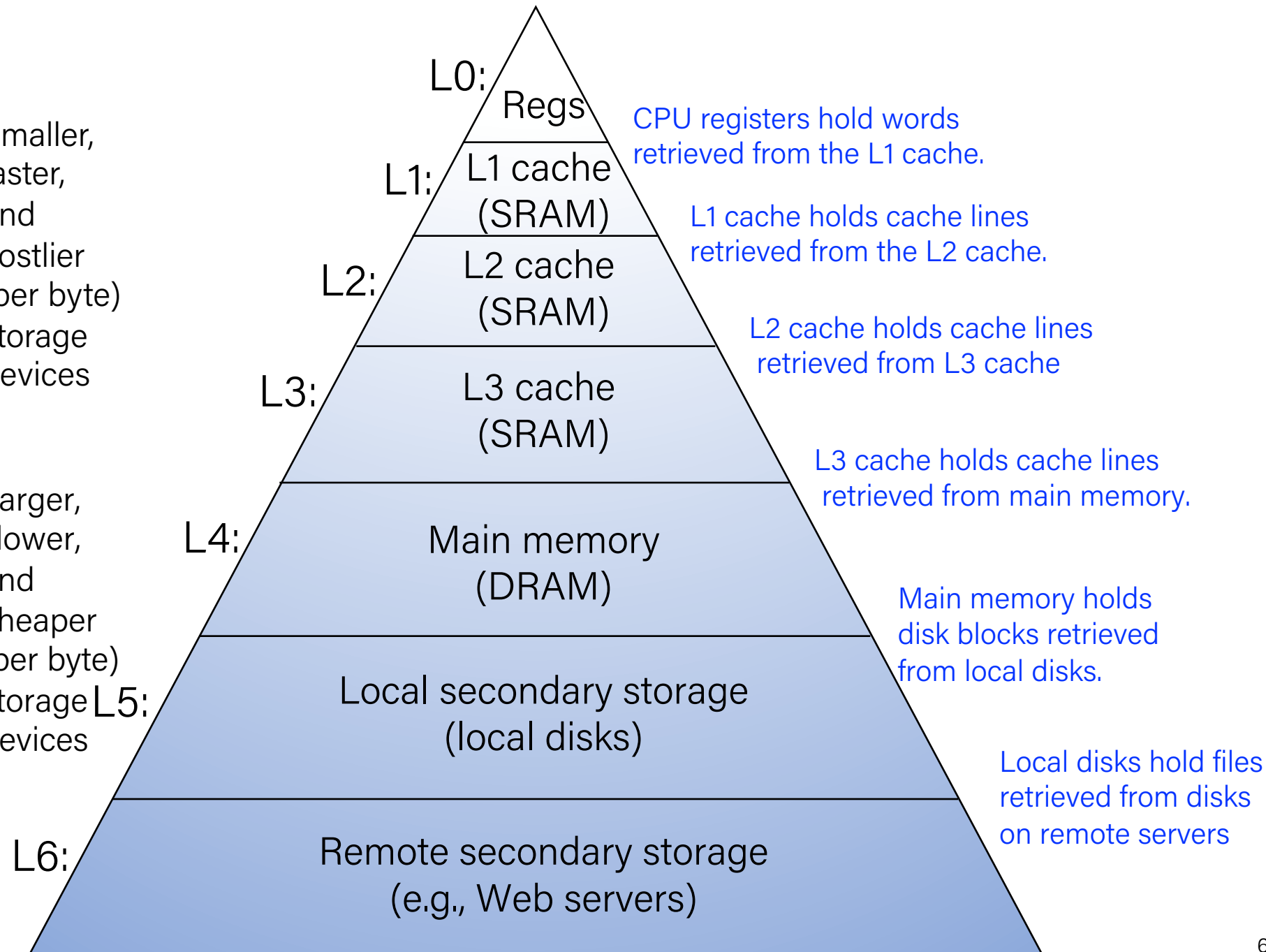
# Lecture Plan

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

# Example Memory Hierarchy

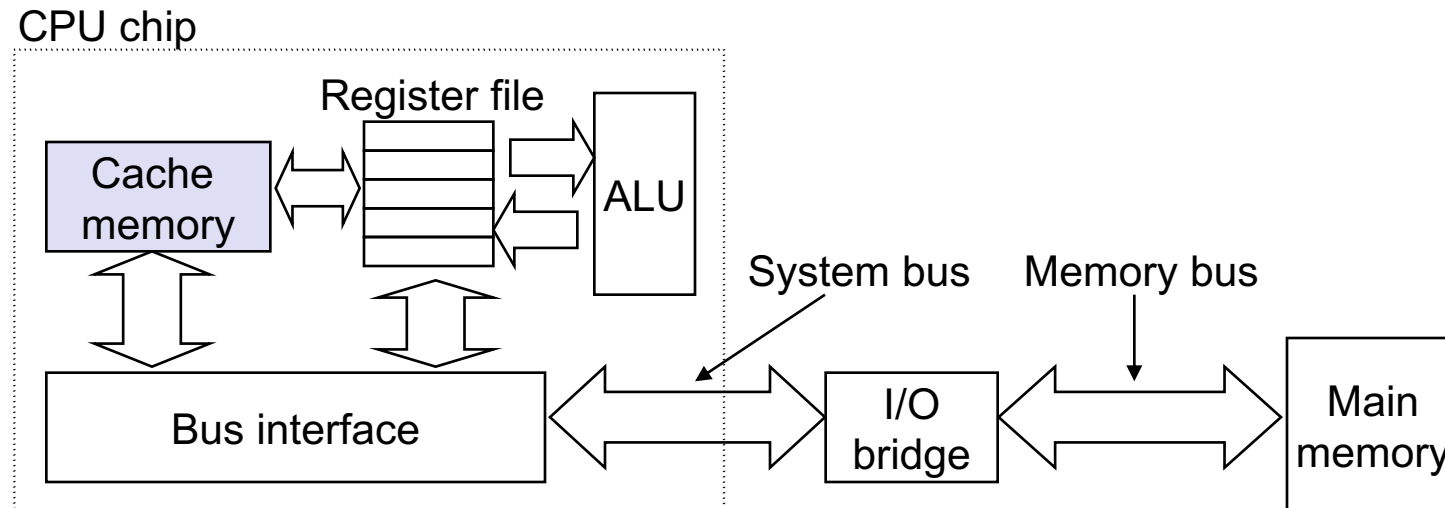
↑  
Smaller,  
faster,  
and  
costlier  
(per byte)  
storage  
devices

↓  
Larger,  
slower,  
and  
cheaper  
(per byte)  
storage  
devices



# Cache Memories

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

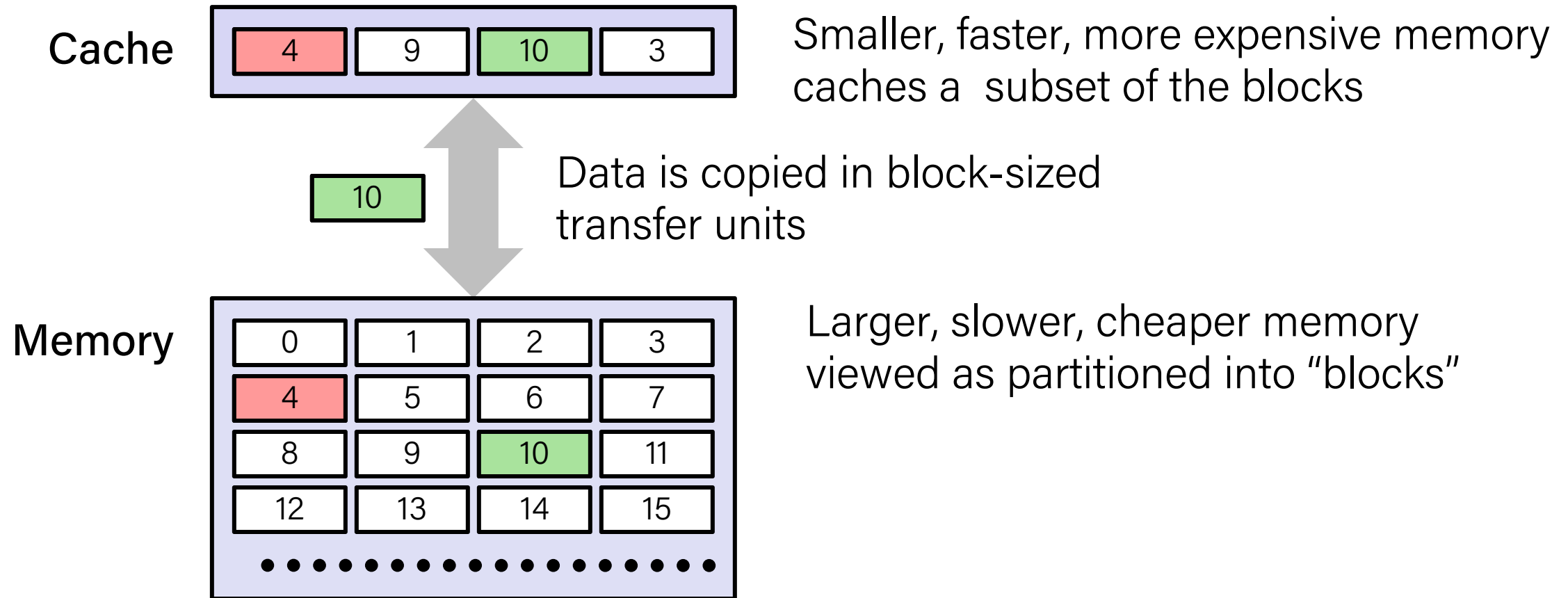


# Examples of Caching in the Mem. Hierarchy

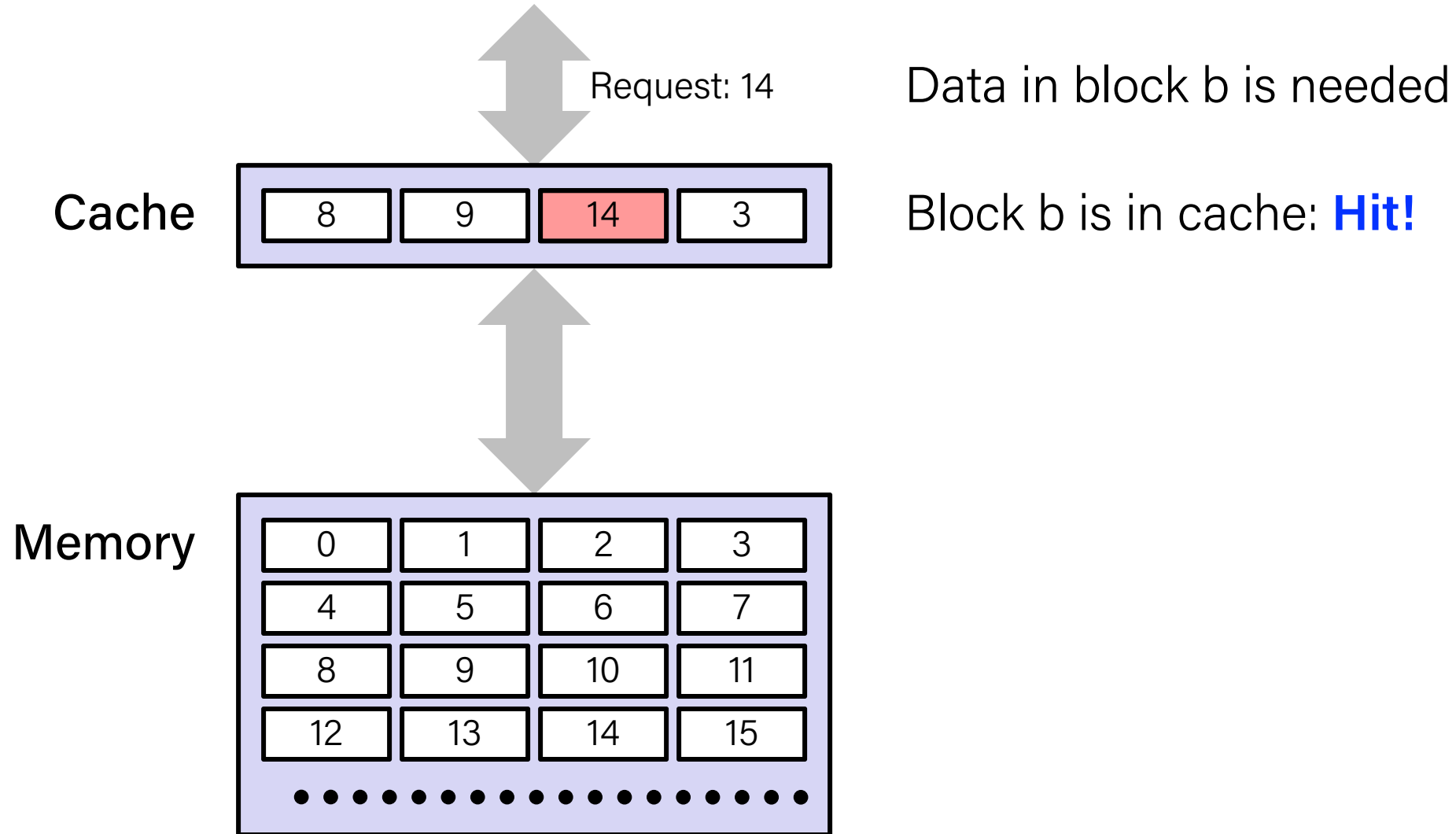
| Cache Type           | What is Cached?      | Where is it Cached? | Latency (cycles) | Managed By       |
|----------------------|----------------------|---------------------|------------------|------------------|
| Registers            | 4-8 bytes words      | CPU core            | 0                | 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 |



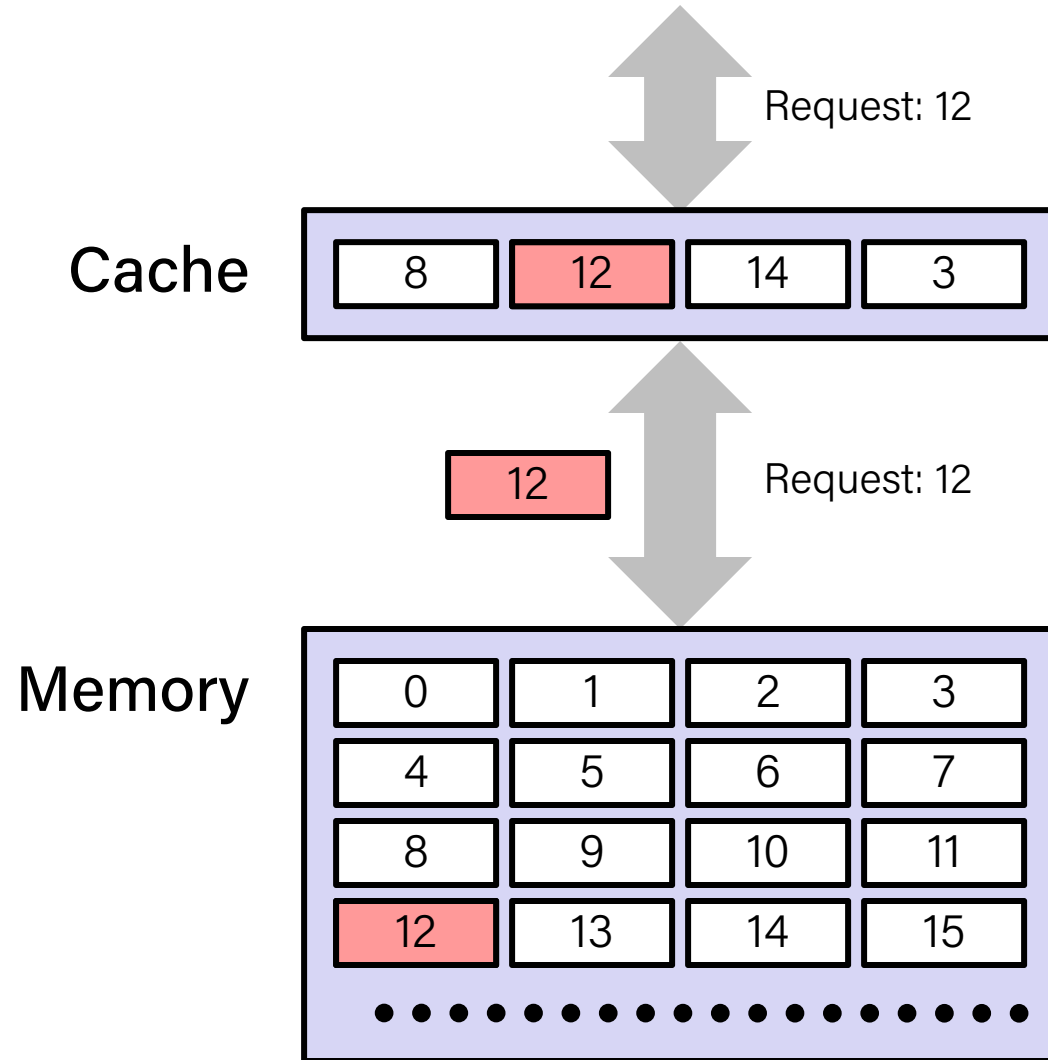
# General Cache Concepts



# General Cache Concepts: 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)

# Types of Cache Misses

- **Cold (compulsory) miss**

- Cold misses occur because the cache is empty.

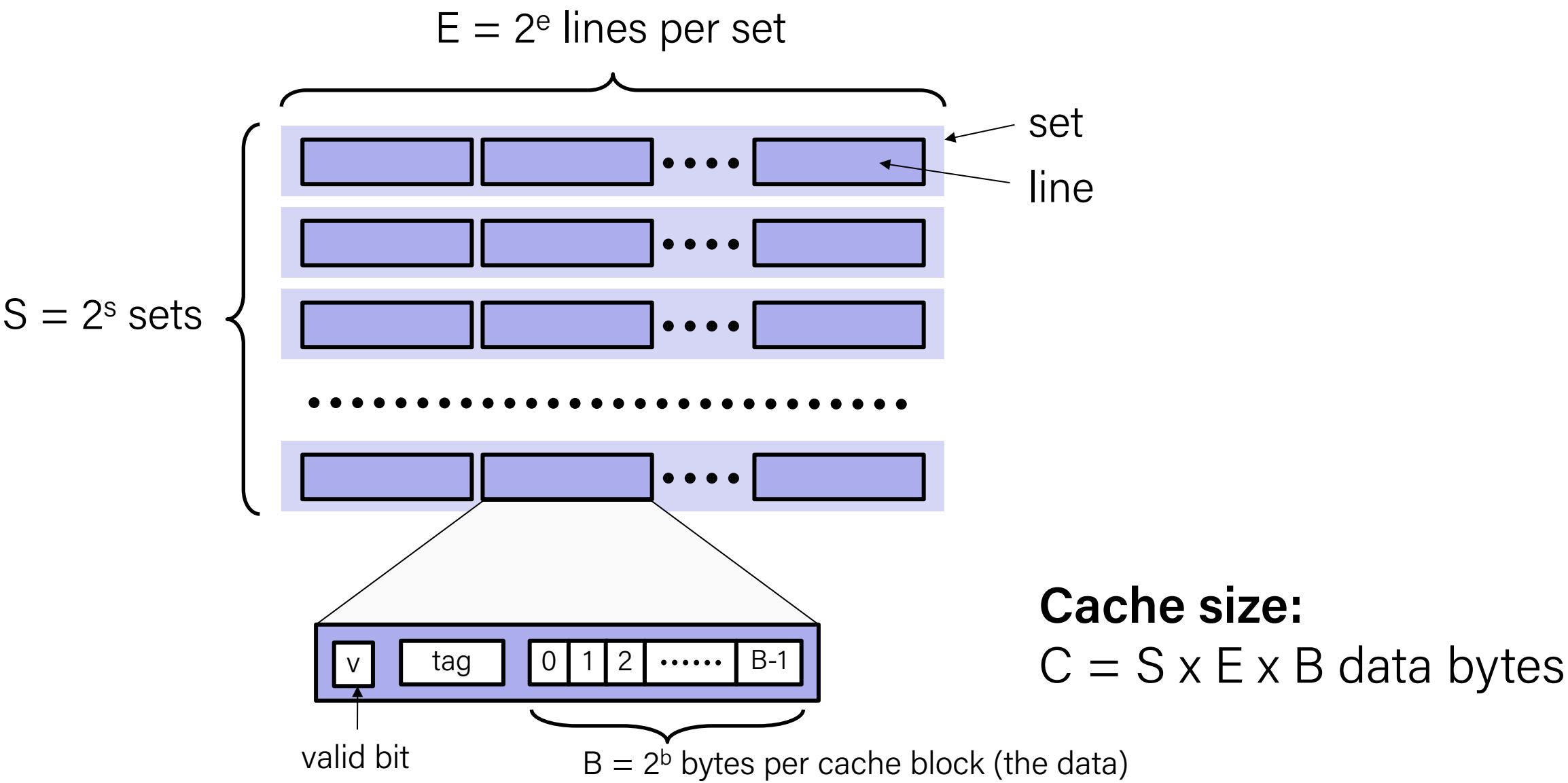
- **Conflict miss**

- Most caches limit blocks at level  $k+1$  to a small subset (sometimes a singleton) of the block positions at level  $k$ .
  - E.g. Block  $i$  at level  $k+1$  must be placed in block  $(i \bmod 4)$  at level  $k$ .
- Conflict misses occur when the level  $k$  cache is large enough, but 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**

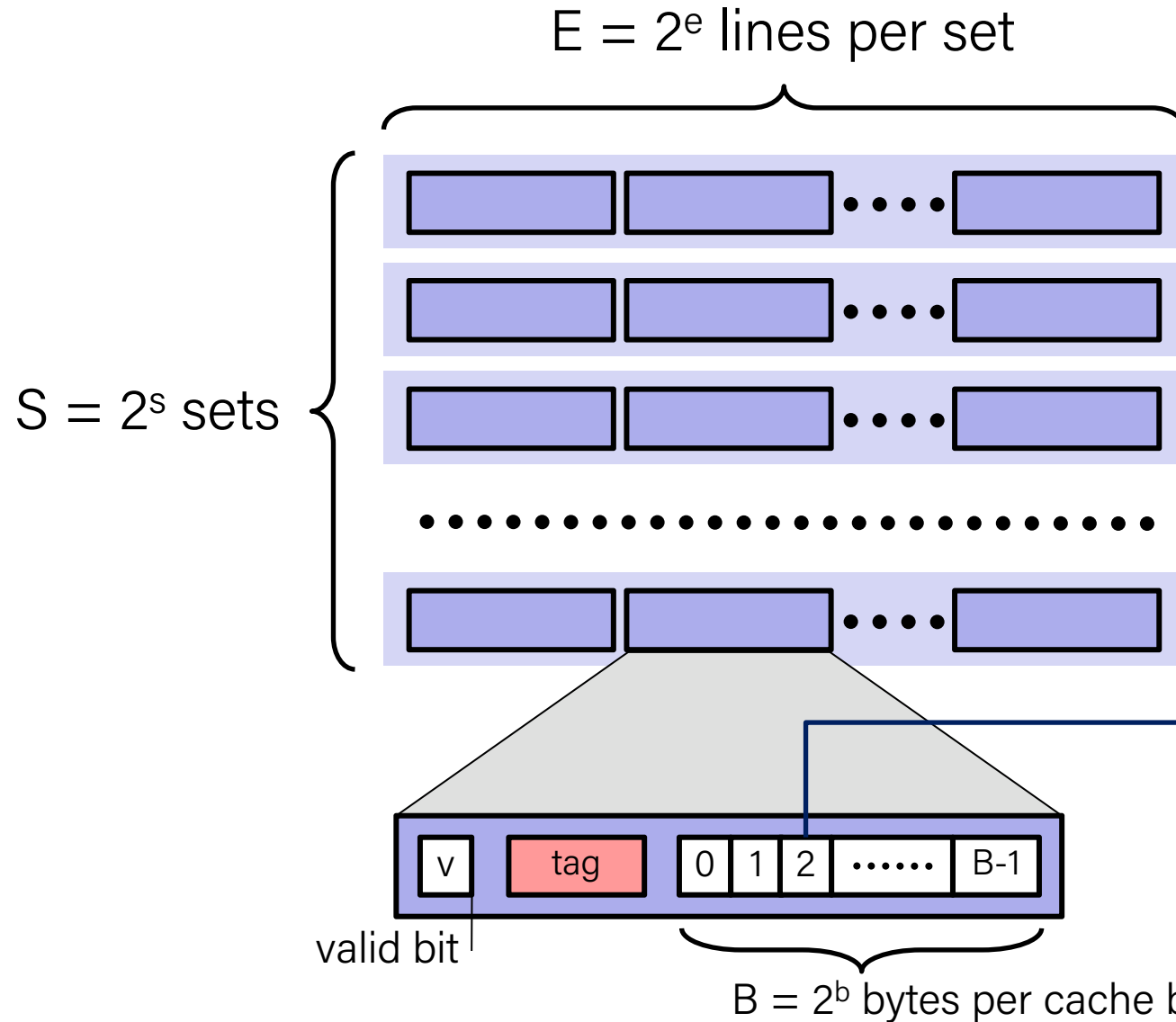
- Occurs when the set of active cache blocks (working set) is larger than the cache.

# General Cache Organization (S, E, B)



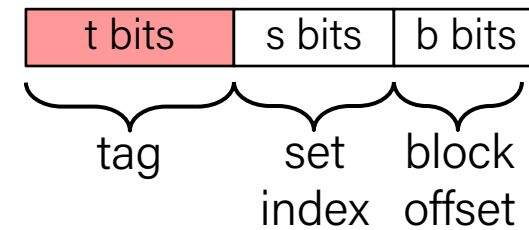


# Cache Read



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

Address of word:

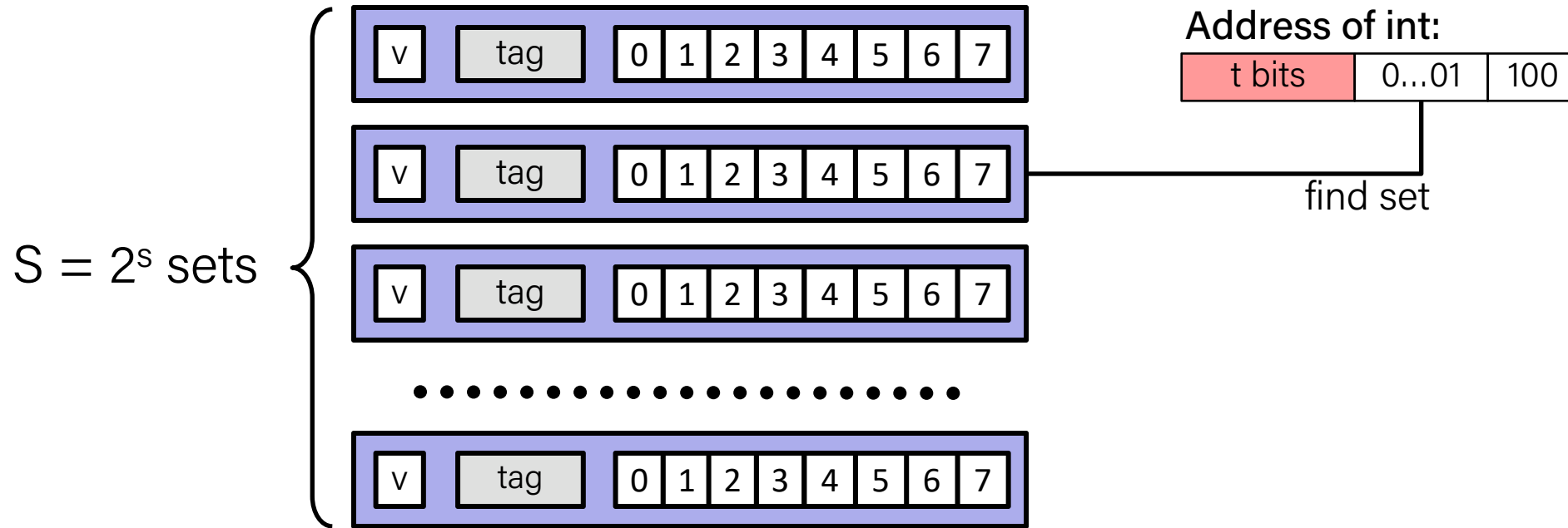


data begins at this offset

# Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set

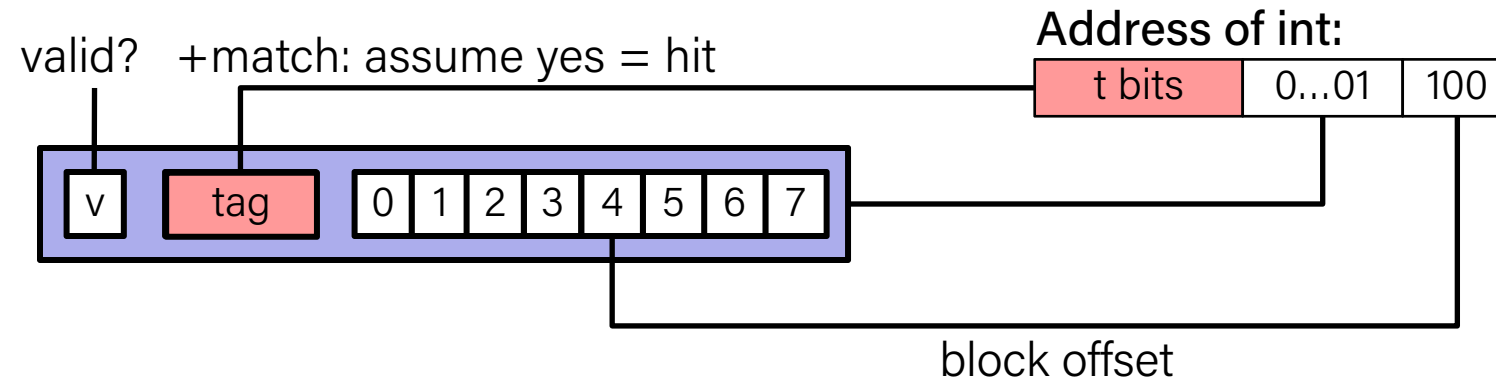
Assume: cache block size 8 bytes



# Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set

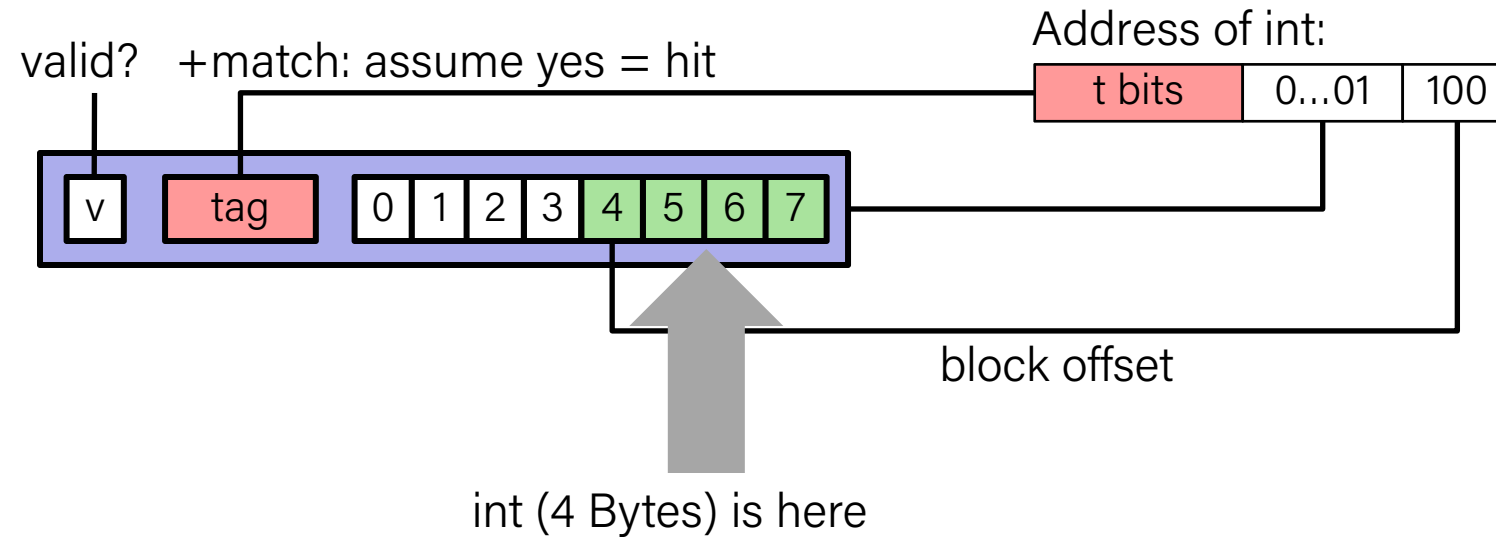
Assume: cache block size 8 bytes



# 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

|       | v | Tag | Block  |
|-------|---|-----|--------|
| Set 0 | 1 | 0   | M[0-1] |
| Set 1 |   |     |        |
| Set 2 |   |     |        |
| Set 3 | 1 | 0   | M[6-7] |

Address trace (reads, one byte per read):

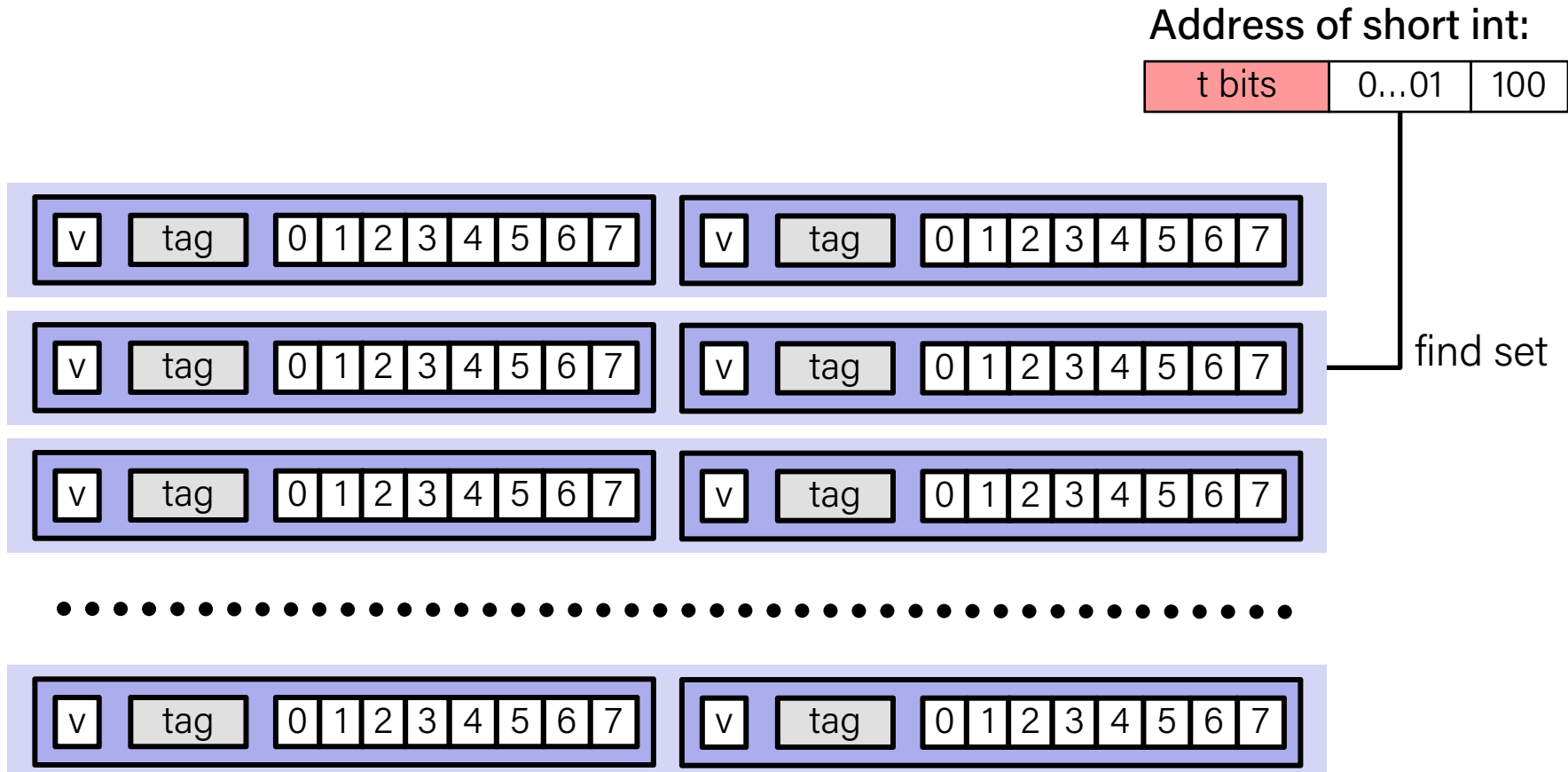
|   |                               |      |
|---|-------------------------------|------|
| 0 | [ <u>0000</u> <sub>2</sub> ], | miss |
| 1 | [ <u>0001</u> <sub>2</sub> ], | hit  |
| 7 | [ <u>0111</u> <sub>2</sub> ], | miss |
| 8 | [ <u>1000</u> <sub>2</sub> ], | miss |
| 0 | [ <u>0000</u> <sub>2</sub> ]  | miss |



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

E = 2: Two lines per set

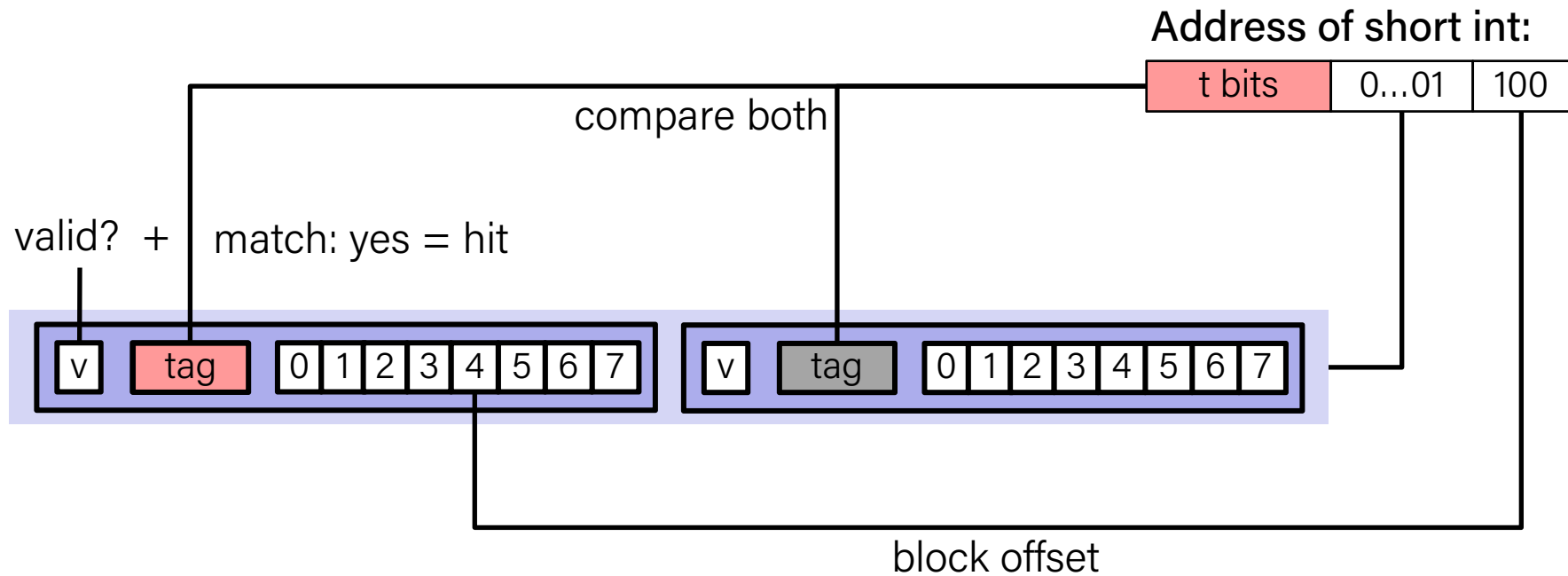
Assume: cache block size 8 bytes



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

E = 2: Two lines per set

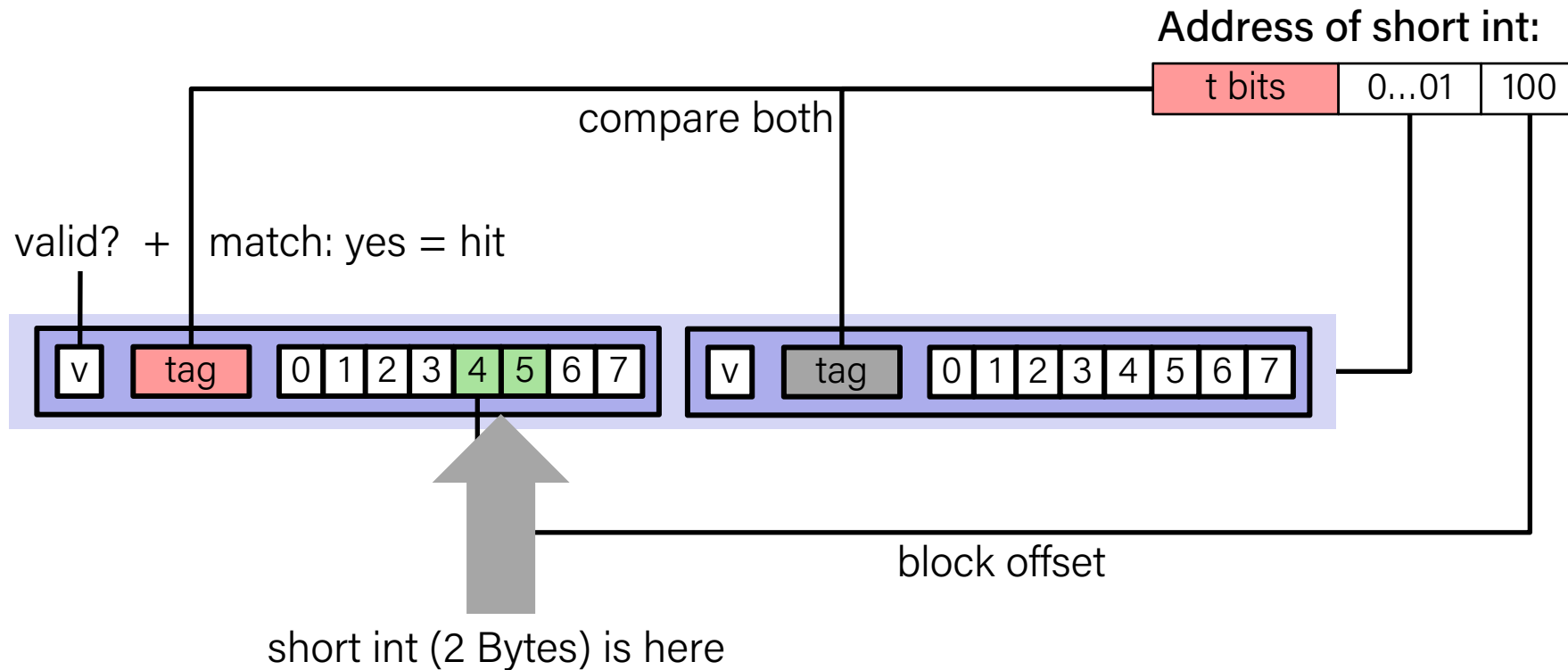
Assume: cache block size 8 bytes



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

E = 2: Two lines per set

Assume: cache block size 8 bytes



## 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 <u>0</u> 0 <sub>2</sub> ], | miss |
| 1 | [00 <u>0</u> 1 <sub>2</sub> ], | hit  |
| 7 | [01 <u>1</u> 1 <sub>2</sub> ], | miss |
| 8 | [10 <u>0</u> 0 <sub>2</sub> ], | miss |
| 0 | [00 <u>0</u> 0 <sub>2</sub> ]  | hit  |

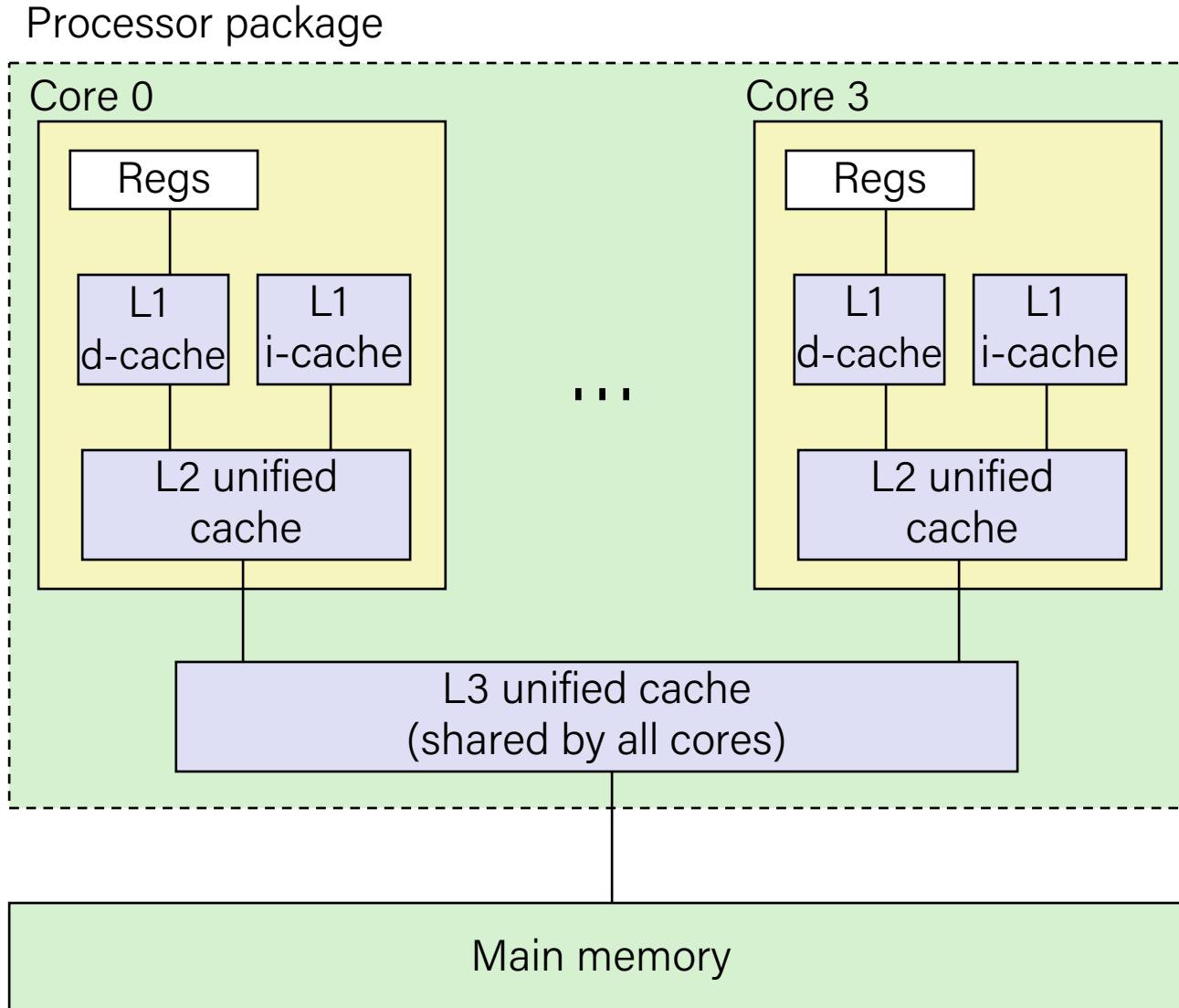
|       | v | Tag | Block  |
|-------|---|-----|--------|
| Set 0 | 1 | 00  | M[0-1] |
|       | 1 | 10  | M[8-9] |
| Set 1 | 1 | 01  | M[6-7] |
|       | 0 |     |        |

# What about writes?

- **Multiple copies of data exist:**
  - L1, L2, L3, Main Memory, Disk
- **What to do on a write-hit?**
  - **Write-through** (write immediately to memory)
  - **Write-back** (defer write to memory until replacement of line)
    - 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)
    - Good if more writes to the location follow
  - **No-write-allocate** (writes straight to memory, does not load into cache)
- **Typical**
  - Write-through + No-write-allocate
  - Write-back + Write-allocate



# 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, depending on size, etc.

- **Hit Time**

- Time to deliver a line in the cache to the processor
  - includes time to determine whether the line is in the cache
- 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
  - Average access time:
    - 97% hits:  $1 \text{ cycle} + 0.03 * 100 \text{ cycles} = 4 \text{ cycles}$
    - 99% hits:  $1 \text{ cycle} + 0.01 * 100 \text{ cycles} = 2 \text{ cycles}$
- **This is why “miss rate” is used instead of “hit rate”**

# Writing Cache Friendly Code

- Make the common case go fast
  - Focus on the inner loops of the core functions
- Minimize the misses in the inner loops
  - Repeated references to variables are good (temporal locality)
  - Stride-1 reference patterns are good (spatial locality)

**Key idea:** Our qualitative notion of locality is quantified through our understanding of cache memories

# Lecture Plan

- 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.
  - Compact way to characterize memory system performance.

# 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
 *      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));
}
```

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

For each elems and stride:

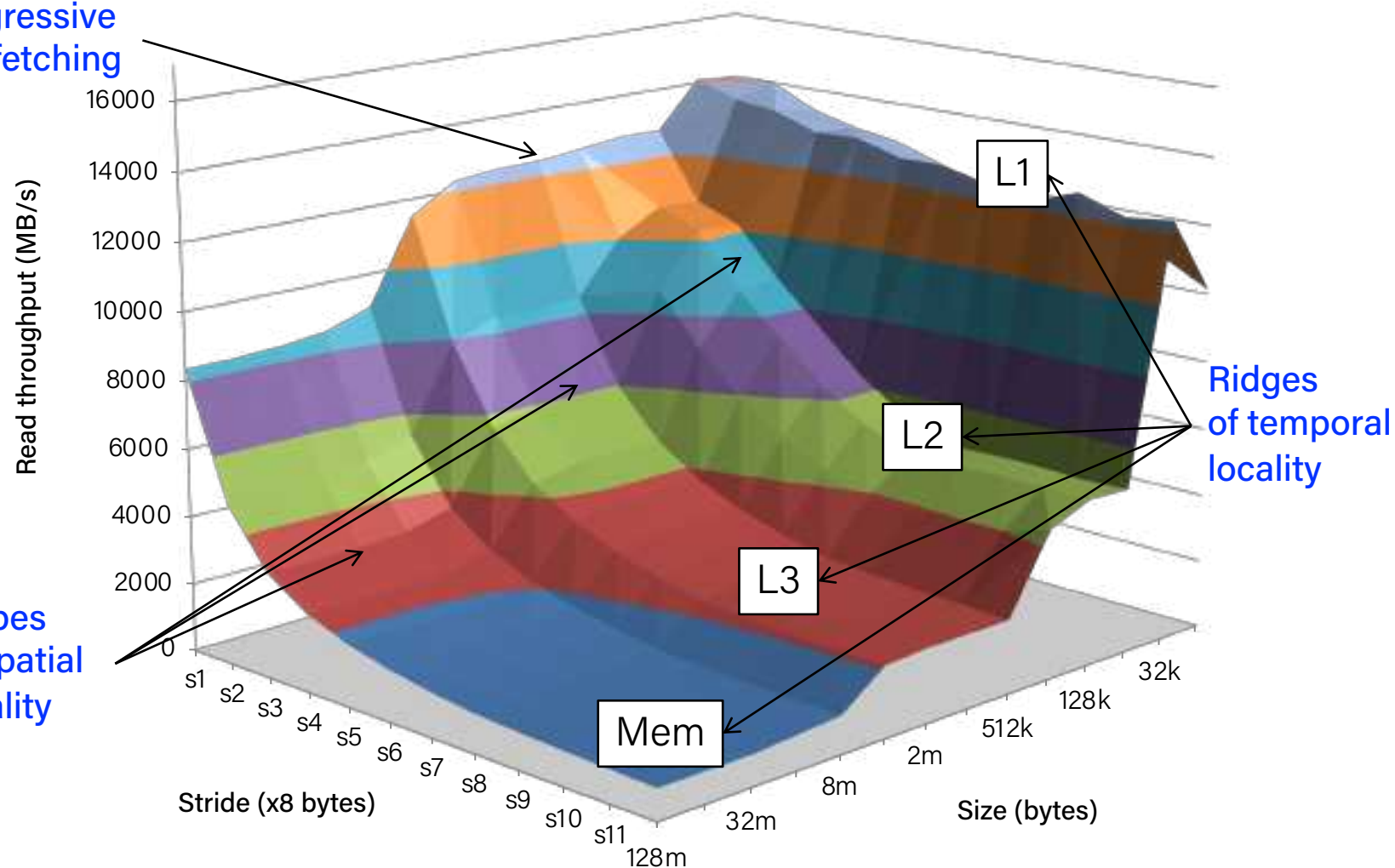
1. Call test() once to warm up the caches.
2. Call test() again and measure the read throughput(MB/s)

mountain/mountain.c

# The Memory Mountain

Aggressive  
prefetching

Slopes  
of spatial  
locality



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

# Lecture Plan

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

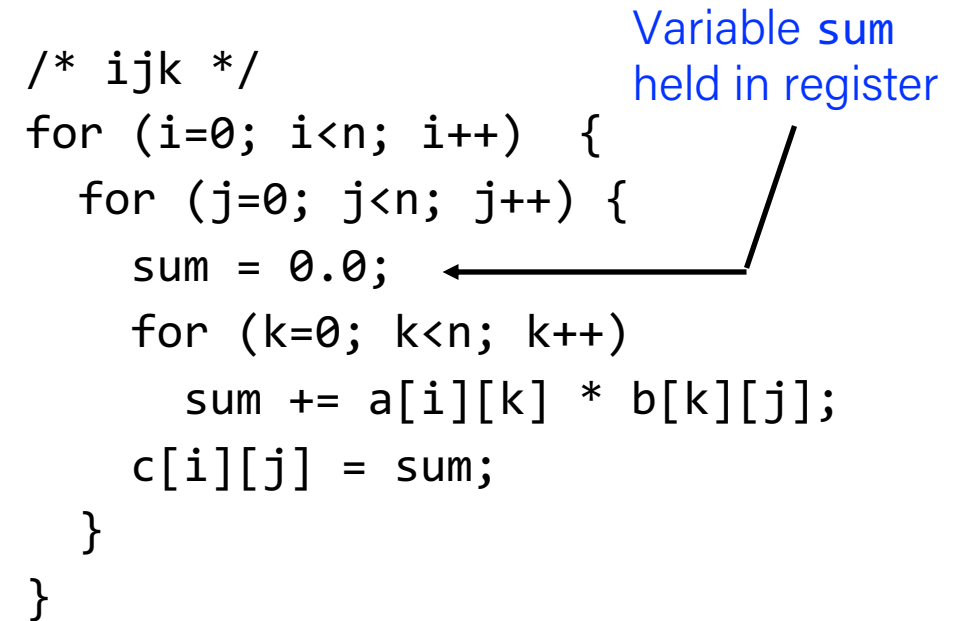
# Matrix Multiplication Example

- Description:

- Multiply  $N \times N$  matrices
- Matrix elements are doubles (8 bytes)
- $O(N^3)$  total operations
- $N$  reads per source element
- $N$  values summed per destination
  - but may be able to hold in register

```
/* 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;
    }
}
```

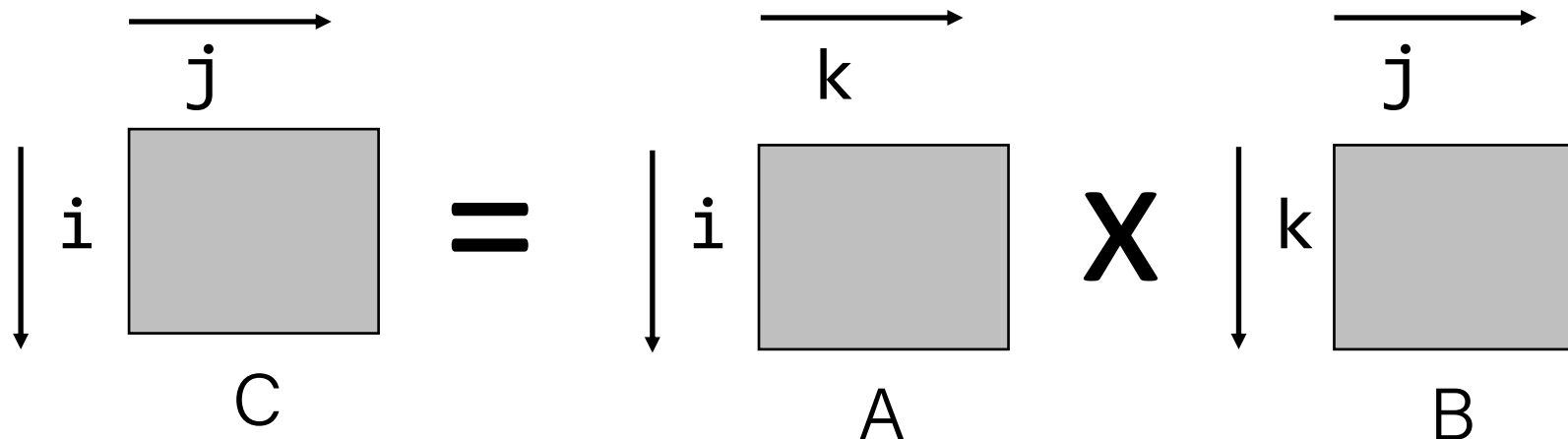
Variable sum  
held in register



matmult/mm.c

# 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
  - each row in contiguous memory locations
- 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<sub>ij</sub>) bytes, exploit spatial locality
  - miss rate = sizeof(a<sub>ij</sub>) / 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)

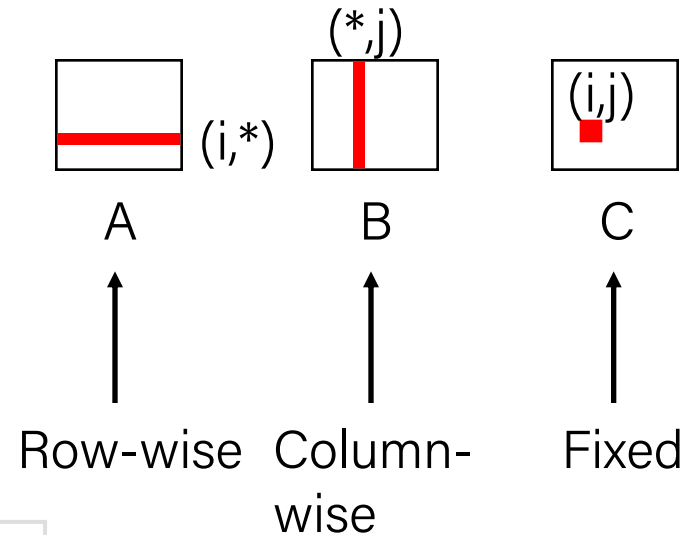
```
/* ijk */  
for (i=0; i<n; i++) {  
    for (j=0; j<n; j++) {  
        sum = 0.0;  
        for (k=0; k<n; k++)  
            sum += a[i][k] * b[k][j];  
        c[i][j] = sum;  
    }  
}
```

Misses per inner loop iteration:

| <u>A</u> | <u>B</u> | <u>C</u> |
|----------|----------|----------|
| 0.25     | 1.0      | 0.0      |

matmult/mm.c

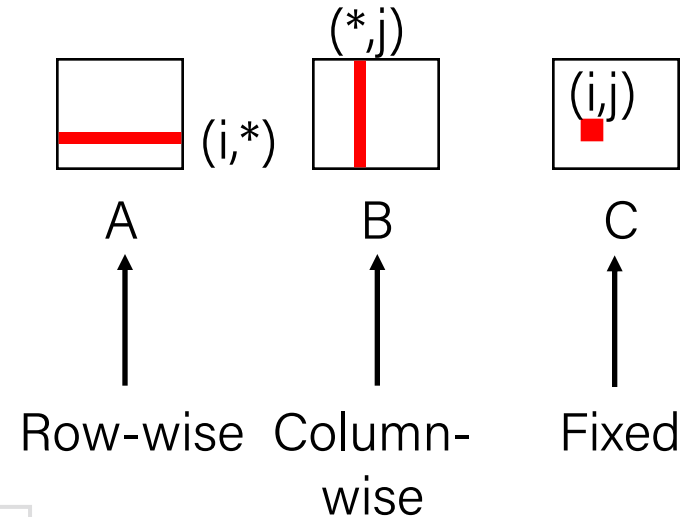
Inner loop:



# 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
    }
}
```

Inner loop:



matmult/mm.c

Misses per inner loop iteration:

| <u>A</u> | <u>B</u> | <u>C</u> |
|----------|----------|----------|
| 0.25     | 1.0      | 0.0      |

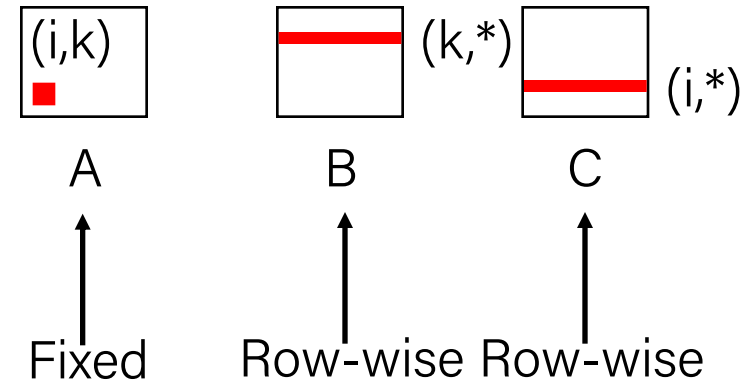
# Matrix Multiplication (kij)

```
/* 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];
    }
}
```

Misses per inner loop iteration:

| <u>A</u> | <u>B</u> | <u>C</u> |
|----------|----------|----------|
| 0.0      | 0.25     | 0.25     |

Inner loop:

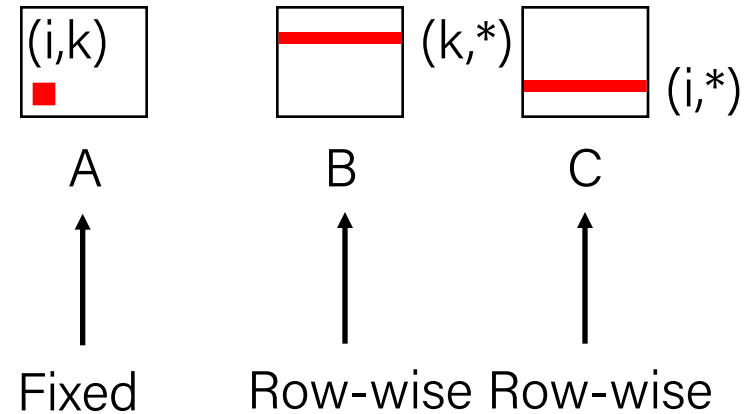


matmult/mm.c

# 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];
    }
}
```

Inner loop:



matmult/mm.c

Misses per inner loop iteration:

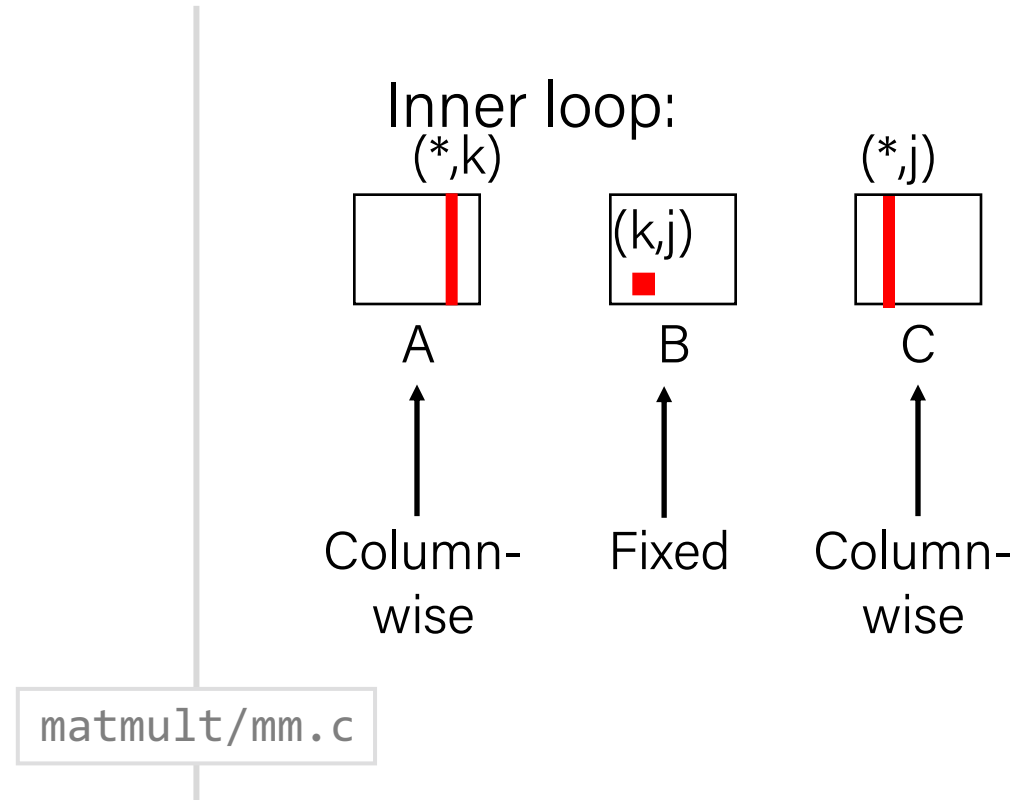
| <u>A</u> | <u>B</u> | <u>C</u> |
|----------|----------|----------|
| 0.0      | 0.25     | 0.25     |

# 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++)
            * r;    c[i][j] += a[i][k]
    }
}
```

Misses per inner loop iteration:

| <u>A</u> | <u>B</u> | <u>C</u> |
|----------|----------|----------|
| 1.0      | 0.0      | 1.0      |

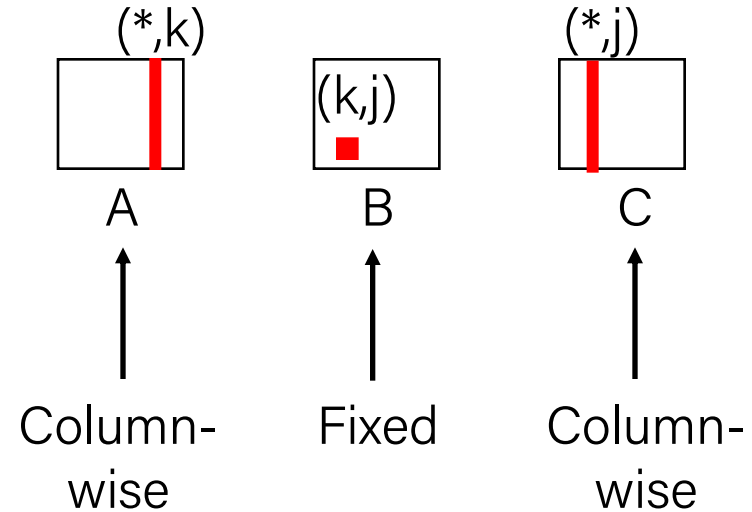


# Matrix Multiplication (kji)

```
/* 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

Inner loop:



Misses per inner loop iteration:

| <u>A</u> | <u>B</u> | <u>C</u> |
|----------|----------|----------|
| 1.0      | 0.0      | 1.0      |

# Summary of Matrix Multiplication

```
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;  
    }  
}
```

```
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];  
    }  
}
```

```
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;  
    }  
}
```

ijk (& jik):

- 2 loads, 0 stores
- misses/iter = 1.25

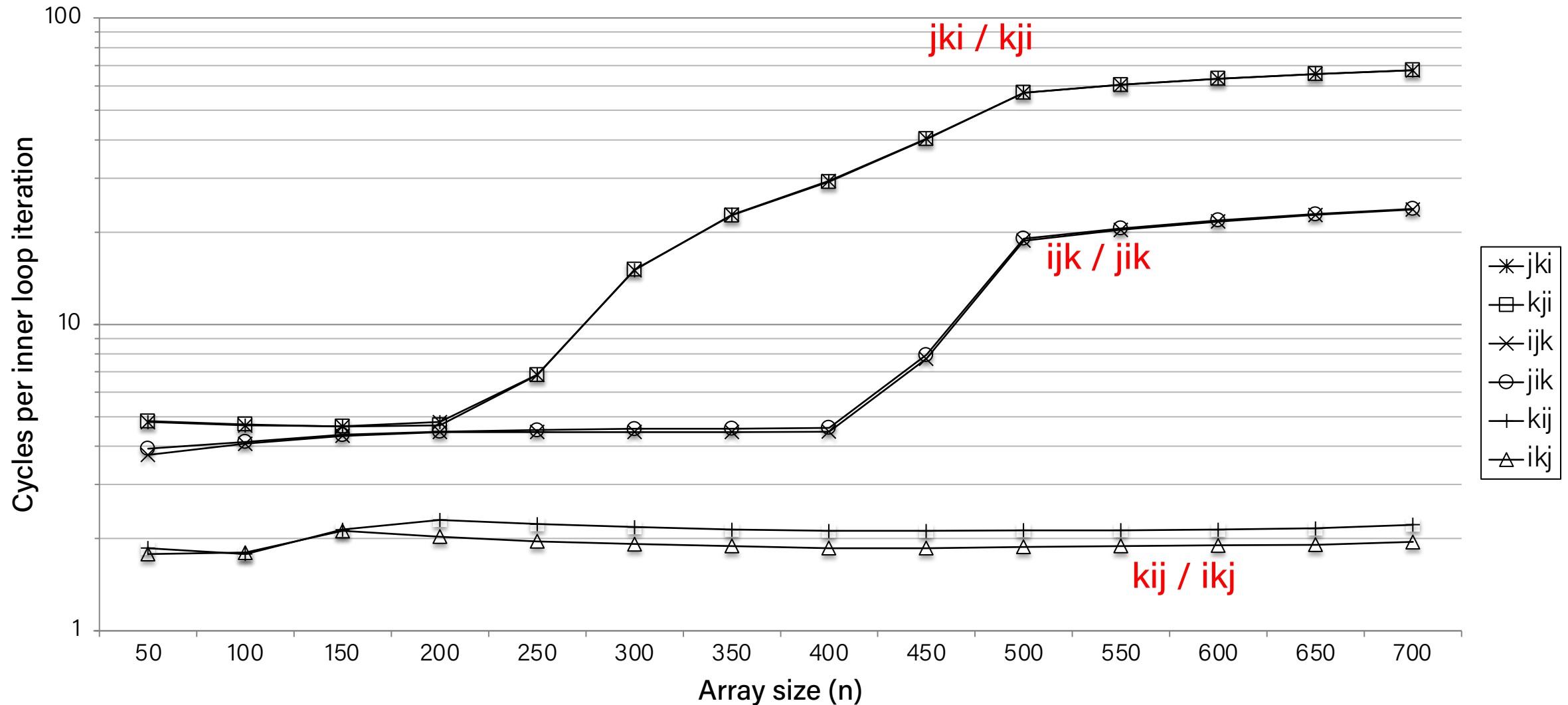
kij (& ikj):

- 2 loads, 1 store
- misses/iter = 0.5

jki (& kji):

- 2 loads, 1 store
- misses/iter = 2.0

# Core i7 Matrix Multiply Performance





# Lecture Plan

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

# Example: Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);
```

```
/* Multiply n x n matrices a and b */  
void mmm(double *a, double *b, double *c, int n) {  
    int i, j, k;  
    for (i = 0; i < n; i++)  
        for (j = 0; j < n; j++)  
            for (k = 0; k < n; k++)  
                c[i*n + j] += a[i*n + k] * b[k*n + j];  
}
```

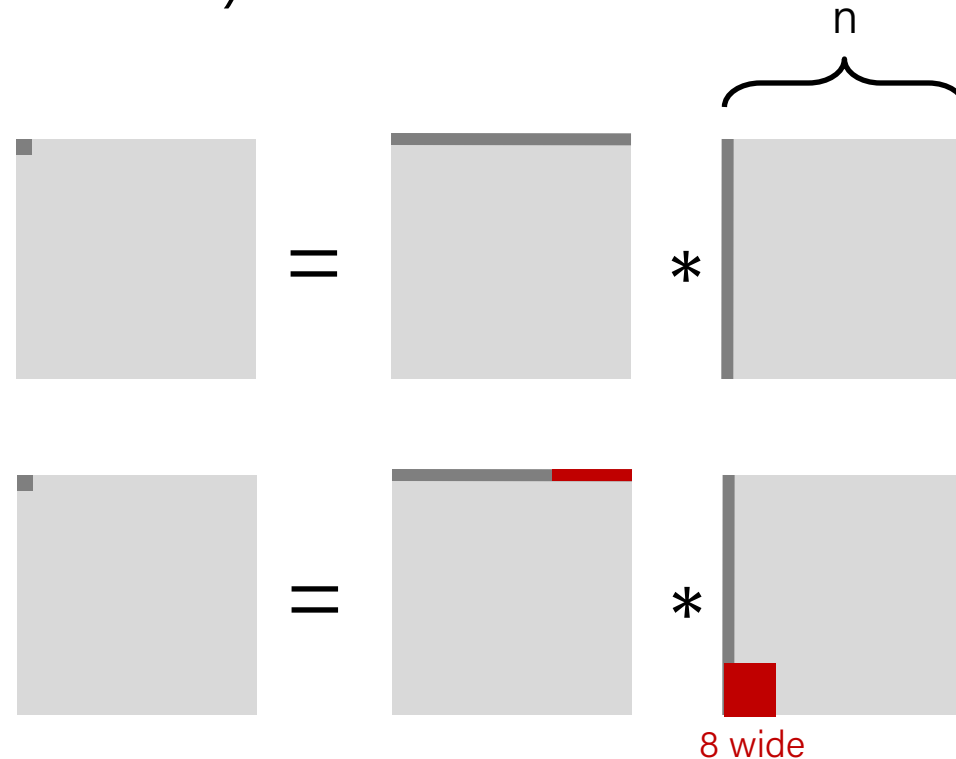


# Cache Miss Analysis

- Assume
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size  $C \ll n$  (much smaller than  $n$ )

- **First iteration:**

- $n/8 + n = 9n/8$  misses



- Afterwards **in cache:**  
(schematic)

# Cache Miss Analysis

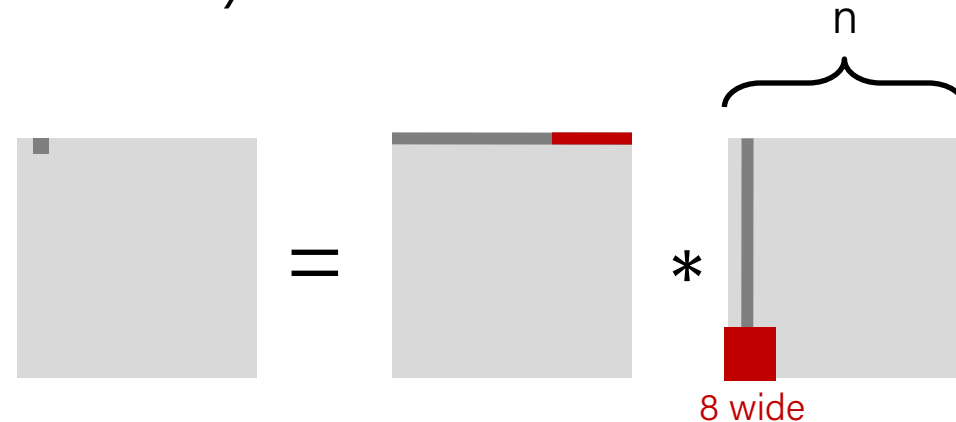
- Assume
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size  $C \ll n$  (much smaller than  $n$ )

- **Second iteration:**

- Again:  
 $n/8 + n = 9n/8$  misses

- **Total misses:**

- $9n/8 * n^2 = (9/8) * n^3$

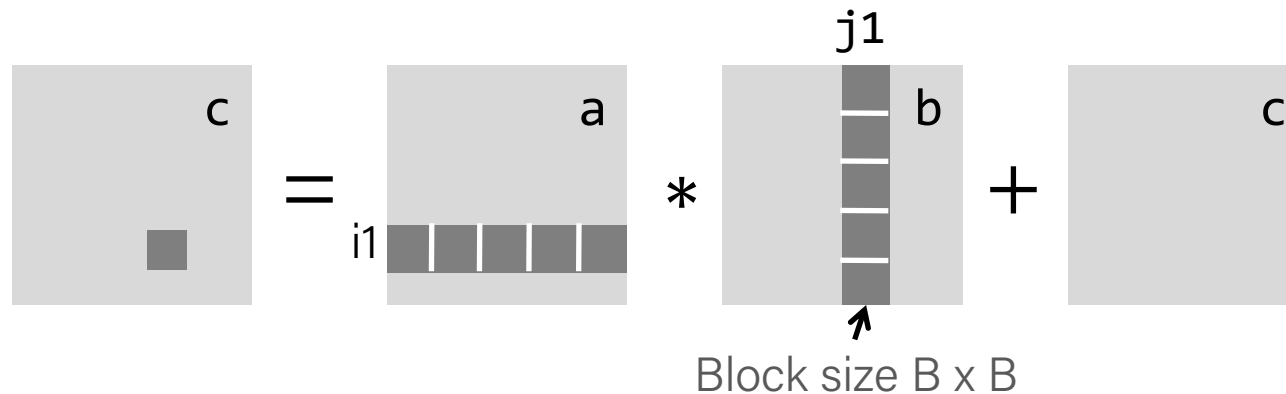


# Blocked Matrix Multiplication


```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                /* B x B mini matrix multiplications */
                for (i1 = i; i1 < i+B; i++)
                    for (j1 = j; j1 < j+B; j++)
                        for (k1 = k; k1 < k+B; k++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```

matmult/bmm.c

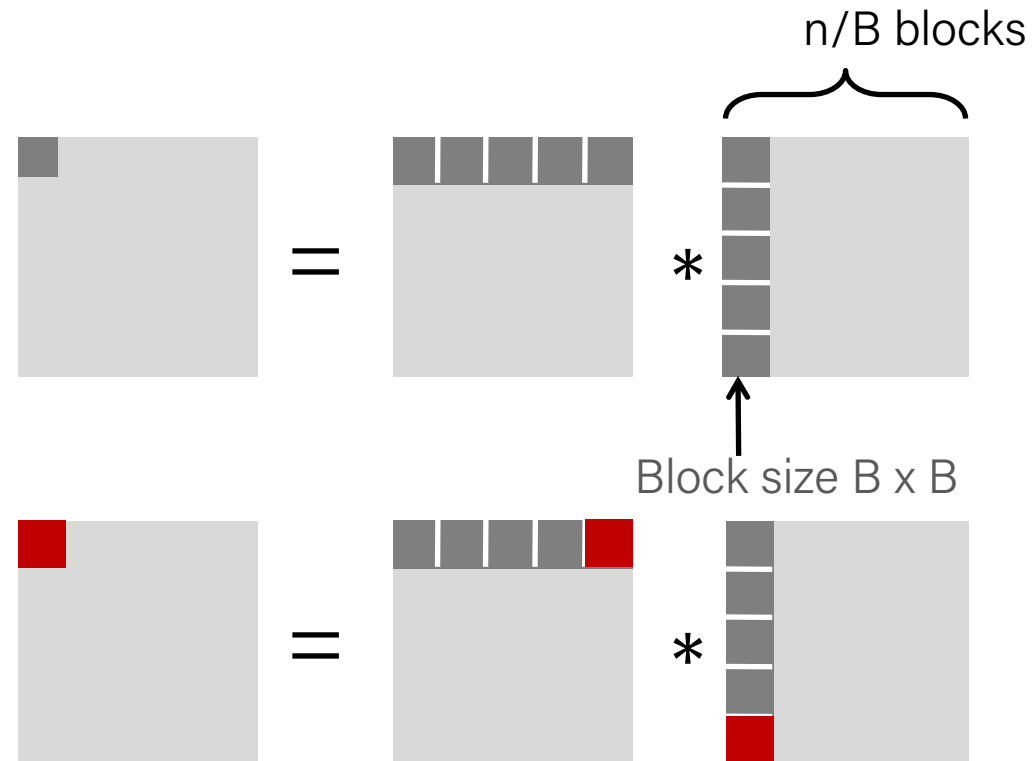


# Cache Miss Analysis


- Assume
  - Cache block = 8 doubles
  - Cache size  $C \ll n$  (much smaller than  $n$ )
  - Three blocks  fit into cache:  $3B^2 < C$

- **First (block) iteration:**

- $B^2/8$  misses for each block
- $2n/B * B^2/8 = nB/4$   
(omitting matrix  $c$ )
- Afterwards in cache  
(schematic)



# Cache Miss Analysis

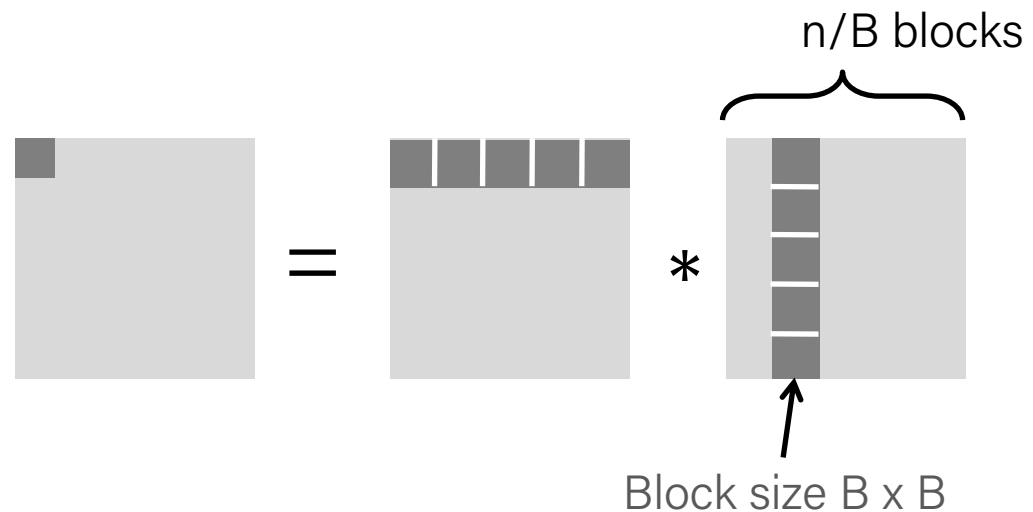
- Assume:
  - Cache block = 8 doubles
  - Cache size  $C \ll n$  (much smaller than  $n$ )
  - Three blocks  fit into cache:  $3B^2 < C$

- **Second (block) iteration:**

- Same as first iteration
- $2n/B * B^2/8 = nB/4$

- Total misses:

- $nB/4 * (n/B)^2 = n^3/(4B)$



# Blocking Summary

- **No blocking:**  $(9/8) * n^3$
- **Blocking:**  $1/(4B) * n^3$
- **Suggest largest possible block size B, but limit  $3B^2 < C$ !**
- **Reason for dramatic difference:**
  - Matrix multiplication has inherent temporal locality:
    - Input data:  $3n^2$ , computation  $2n^3$
    - Every array elements used  $O(n)$  times!
  - But program has to be written properly



# Recap

- Cache memories can have significant performance impact
- You can write your programs to exploit this!
  - Focus on the inner loops, where bulk of computations and memory accesses occur.
  - Try to maximize spatial locality by reading data objects with sequentially with stride 1.
  - Try to maximize temporal locality by using a data object as often as possible once it's read from memory.

**Next time:** Debugging and Design