

# Cache Memories

主讲人: 邓倩妮

上海交通大学

内容来自：

《深入理解计算机系统》第三版，机械工业出版社，作者：Bryant,R.E.  
等



上海交通大学  
SHANGHAI JIAO TONG UNIVERSITY



## Objectives of this lecture

- Master the concepts of hierarchical memory organization.
- Understand how each level of memory contributes to system performance, and how the performance is measured.
- Master the concepts behind cache memory.
- Understand the performance of cache memory

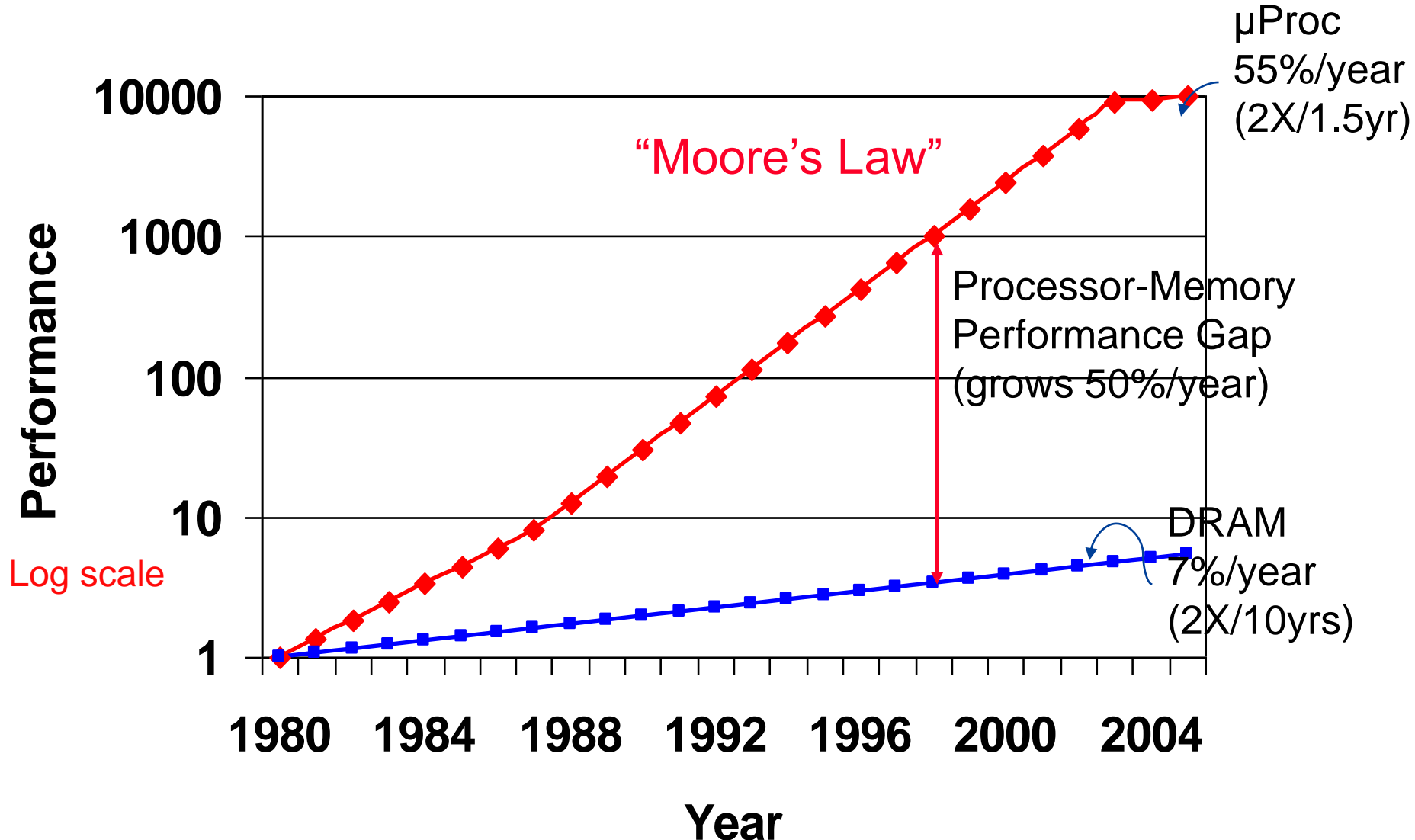


# Today

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



# Memory Wall : Processor-Memory Performance Gap



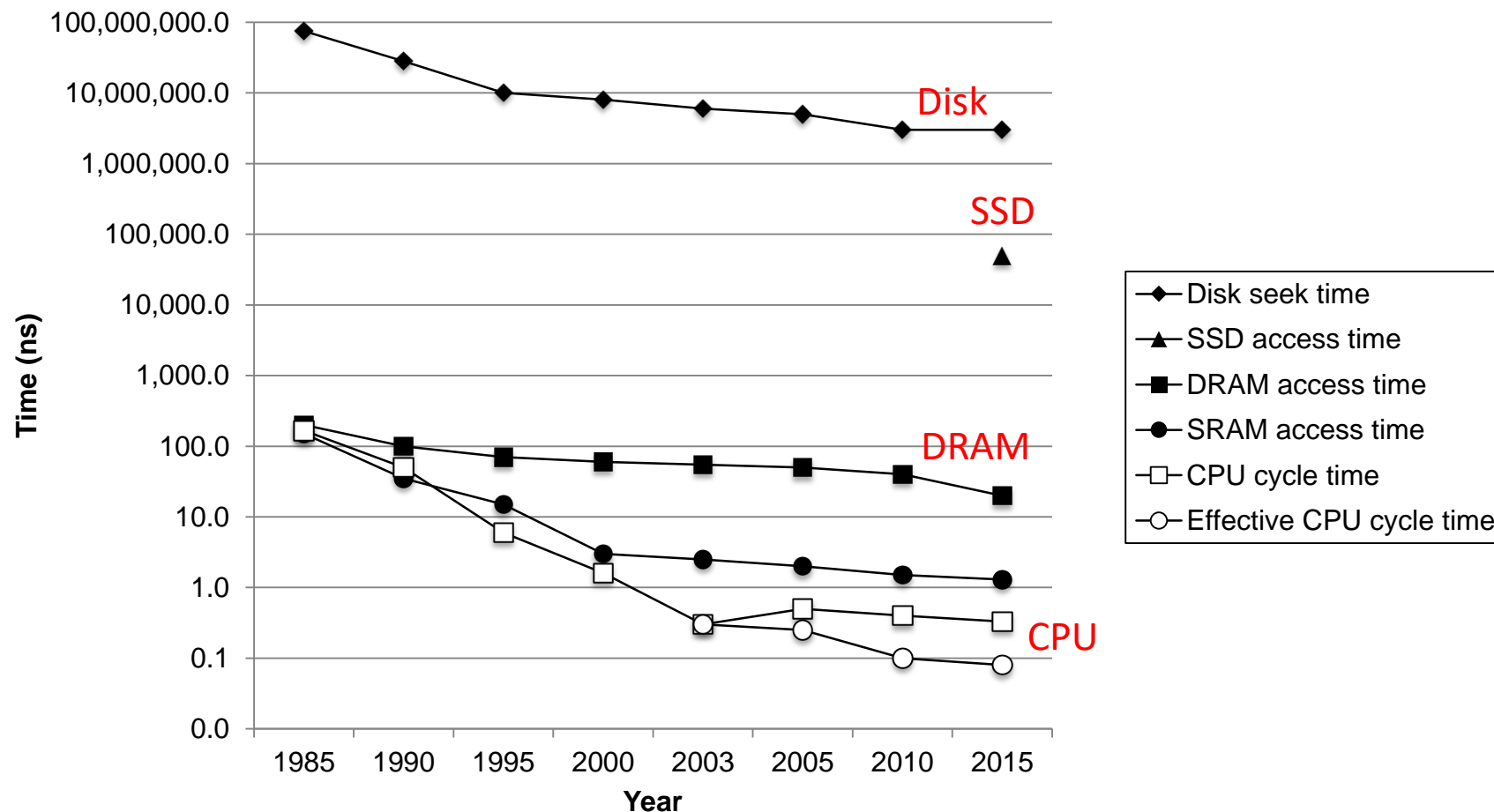
# Memory Technology Trend

	<u>Capacity</u>	<u>Speed (latency)</u>
<b>DRAM:</b>	4 x in 3 years	2 x in 10 years
<b>Disk:</b>	4 x in 3 years	2 x in 10 years

DRAM		
Year	Size	Cycle Time
1980	64 kbit	250 nsec
1983	256 kb	220 ns
1986	1 Mb	190 ns
1989	4 Mb	165 ns
1992	16 Mb	145 ns
1995	64 Mb	120 ns
1999	128 Mb	110 ns
2003	512 Mb	90 ns
2007	2 Gb	60 ns

# The CPU-Memory Gap

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



# Locality to the Rescue!



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

# 局部性原理：



1968年， Denning.P 指出：

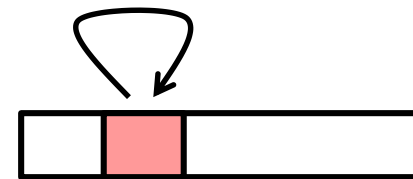
- ① 程序执行时，除了少部分的转移和过程调用指令外，在大多数情况下仍是顺序执行的。
- ② 过程调用将会使程序的执行轨迹由一部分区域转至另一部分区域，但过程调用的深度在大多数情况下都不超过5。
- ③ 程序中存在许多循环结构，只由少数指令构成，但是它们将多次执行。
- ④ 程序中包括许多对数据结构的处理，如对数组进行操作，它们往往都局限于很小的范围内。



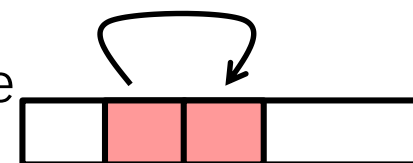
# Locality



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



- **Temporal locality (时间局部性) :**
  - Recently referenced items are likely to be referenced again in the near future



- **Spatial locality (空间局部性) :**
  - Items with nearby addresses tend to be referenced close together in time

# Locality Example



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

- Data references

- Reference array elements in succession (stride-1 reference pattern).
- Reference variable sum each iteration.

Spatial locality

Temporal locality

- Instruction references

- Reference instructions in sequence.
- Cycle through loop repeatedly.

Spatial locality

Temporal locality

# 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 professional programmer.**
- **Question:** Does this function have good locality with respect to array a?

```
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;
}
```

# Memory Hierarchies

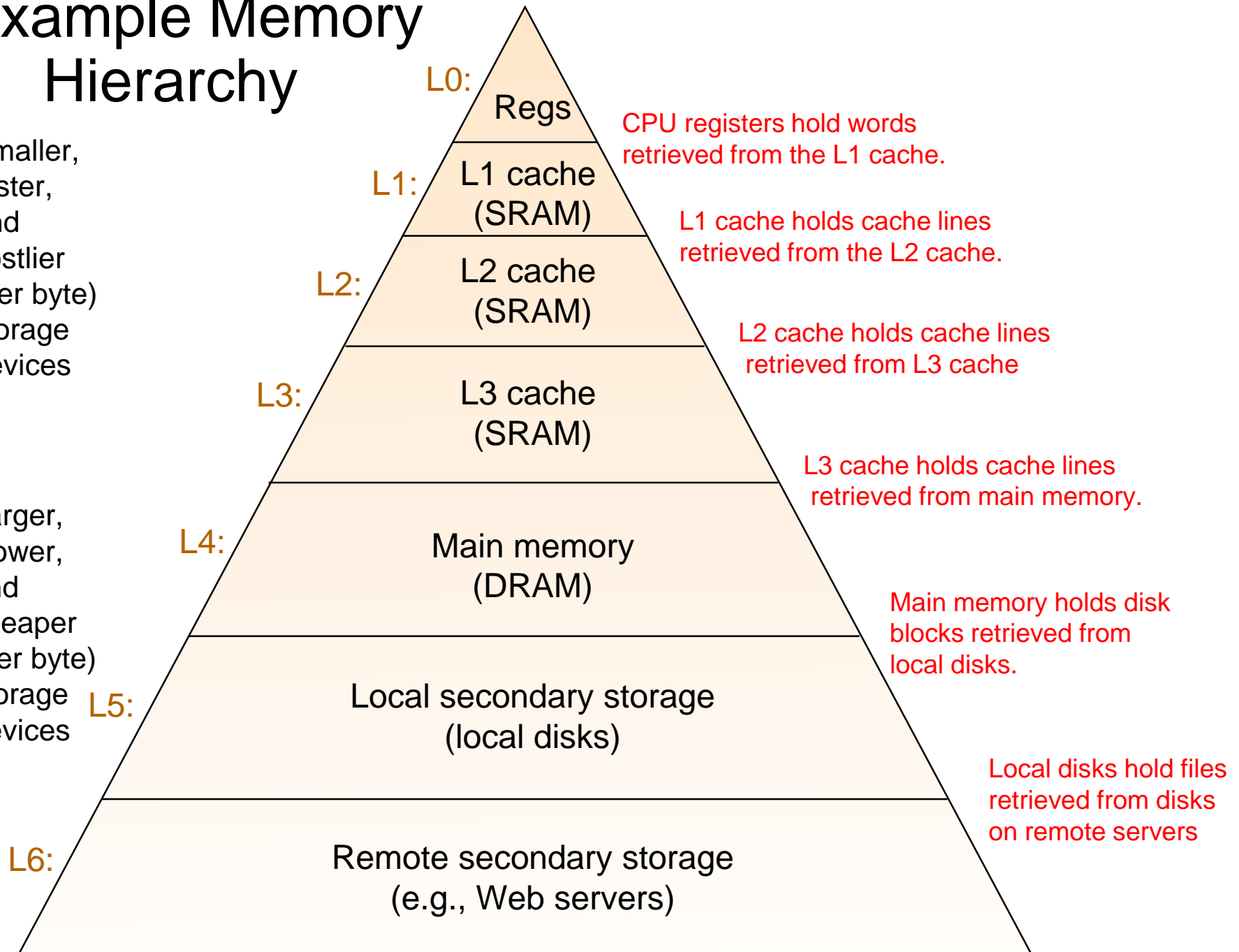


- Some fundamental and enduring properties of hardware and software:
  - Fast storage technologies cost more per byte, have less capacity, and require more power (heat!).
  - The gap between CPU and main memory speed is widening.
  - Well-written programs tend to exhibit good locality.
- These fundamental properties complement each other beautifully.
- They suggest an approach for organizing memory and storage systems known as a **memory hierarchy**.

# Example Memory Hierarchy

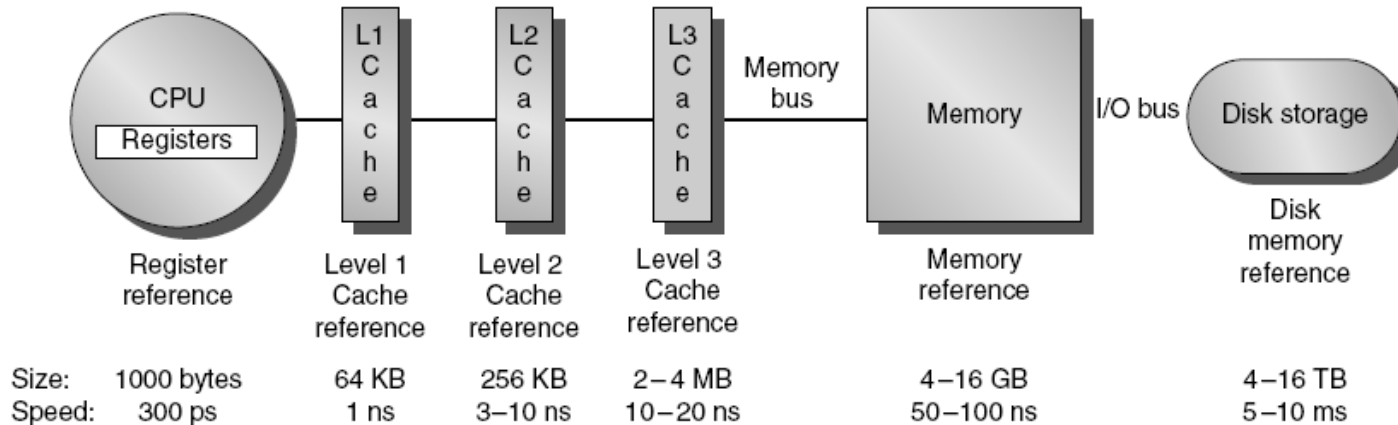
↑  
Smaller,  
faster,  
and  
costlier  
(per byte)  
storage  
devices

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

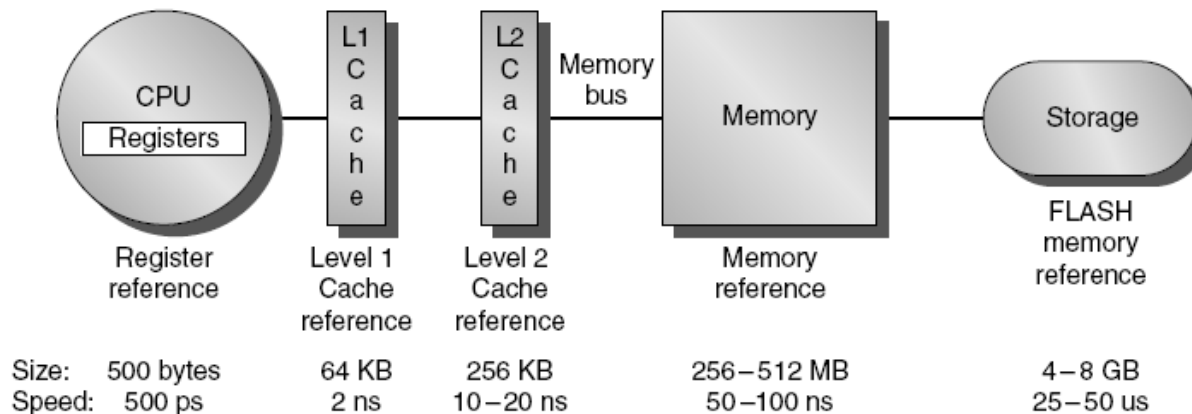




# Memory Hierarchy

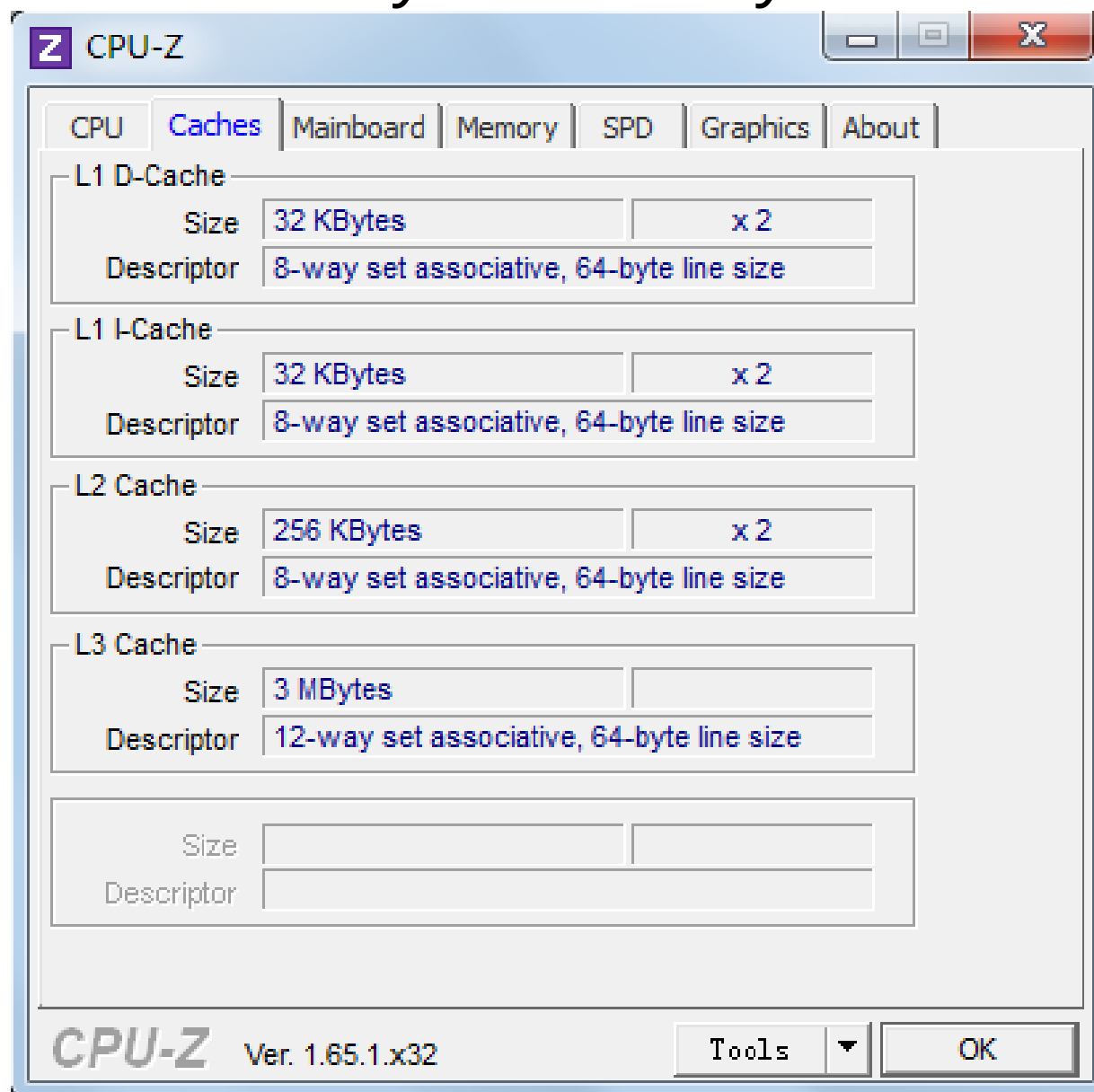


(a) Memory hierarchy for server



(b) Memory hierarchy for a personal mobile device

# Memory hierarchy on PC





# Today

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



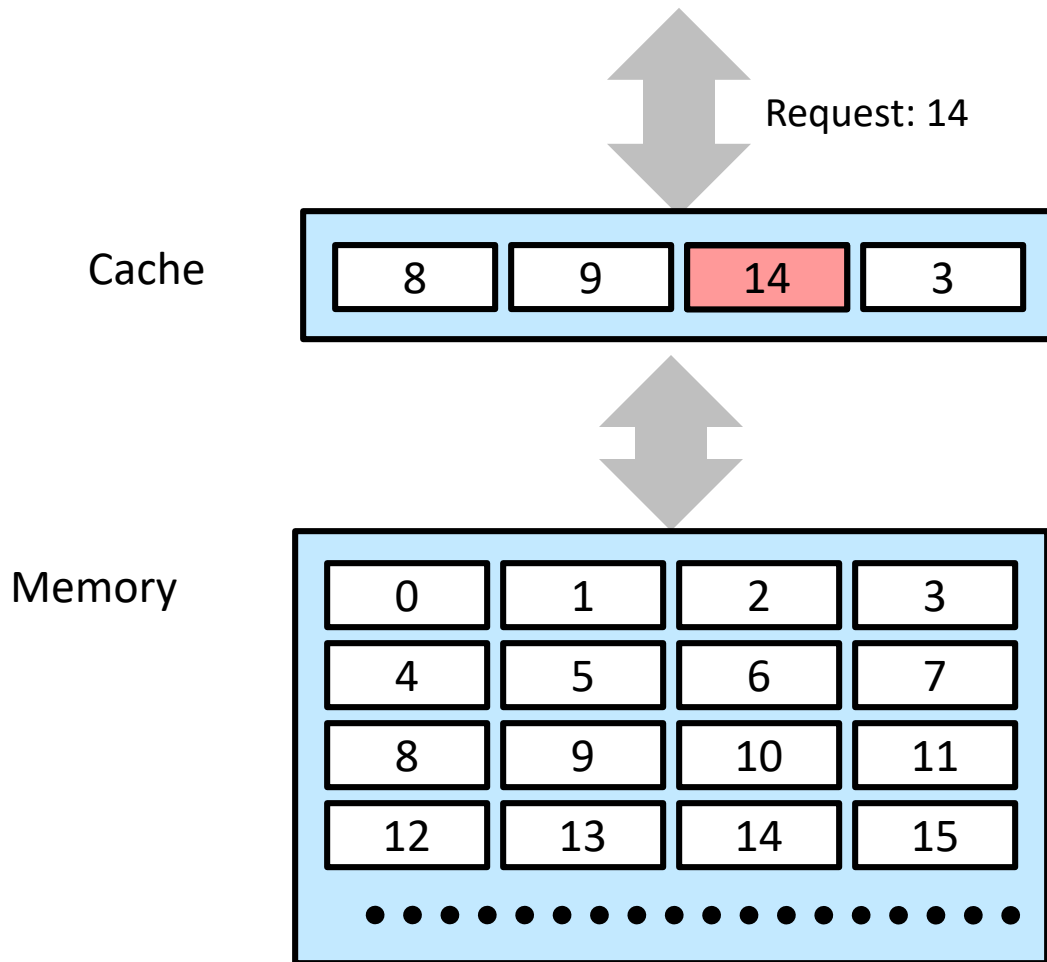
# Caches



- *Cache:* A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.
- Fundamental idea of a memory hierarchy:
  - For each  $k$ , the faster, smaller device at level  $k$  serves as a cache for the larger, slower device at level  $k+1$ .
- Why do memory hierarchies work?
  - Because of locality, programs tend to access the data at level  $k$  more often than they access the data at level  $k+1$ .
  - Thus, the storage at level  $k+1$  can be slower, and thus larger and cheaper per bit.
- *Big Idea:* The memory hierarchy creates a large pool of storage that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.



# General Cache Concepts: Hit



*Data in block b is needed*

*Block b is in cache:*

*Hit!*



# General Concepts: 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.**

# 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



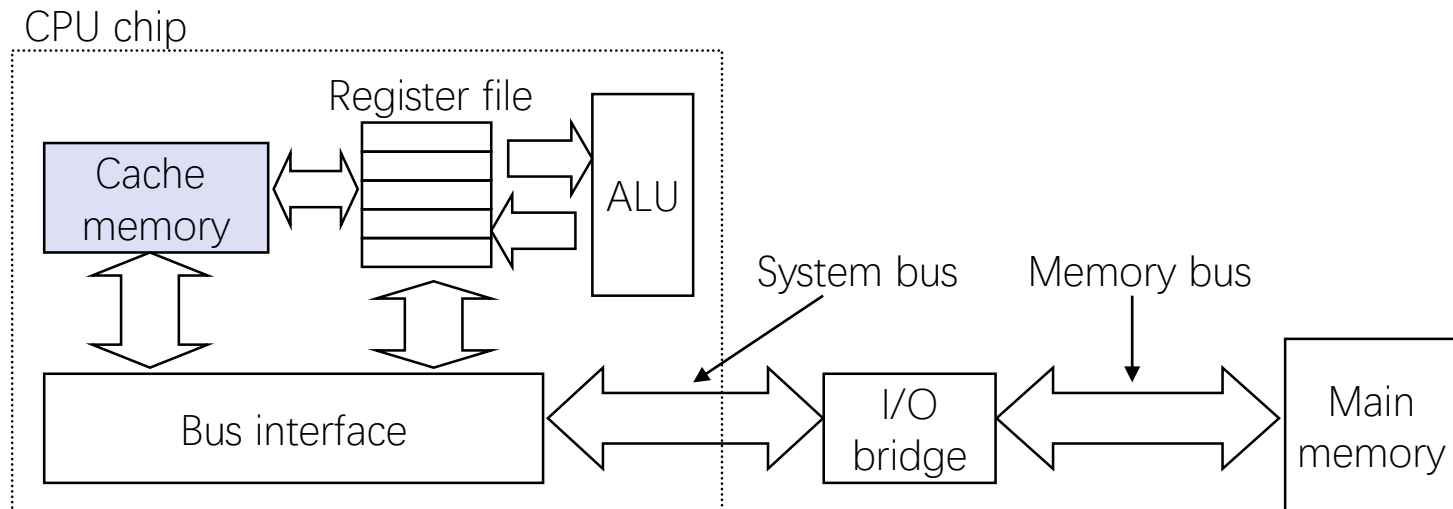
# Today

- Memory hierarchy
- Basic idea of Cache
- 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

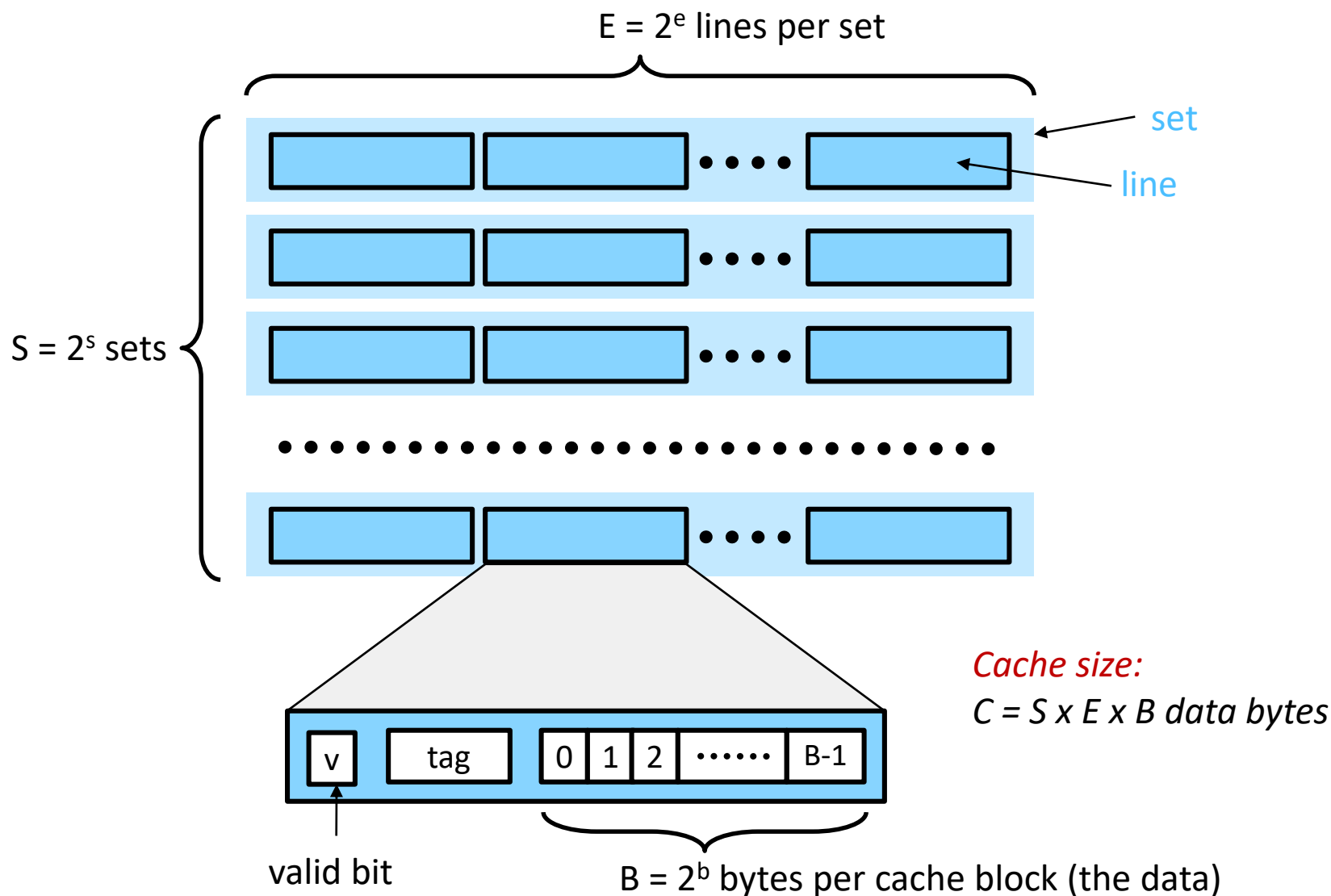


- **Cache memories** are small, fast SRAM-based memories managed automatically in hardware
  - Hold frequently accessed blocks of main memory
- CPU looks first for data in cache
- Typical system structure:

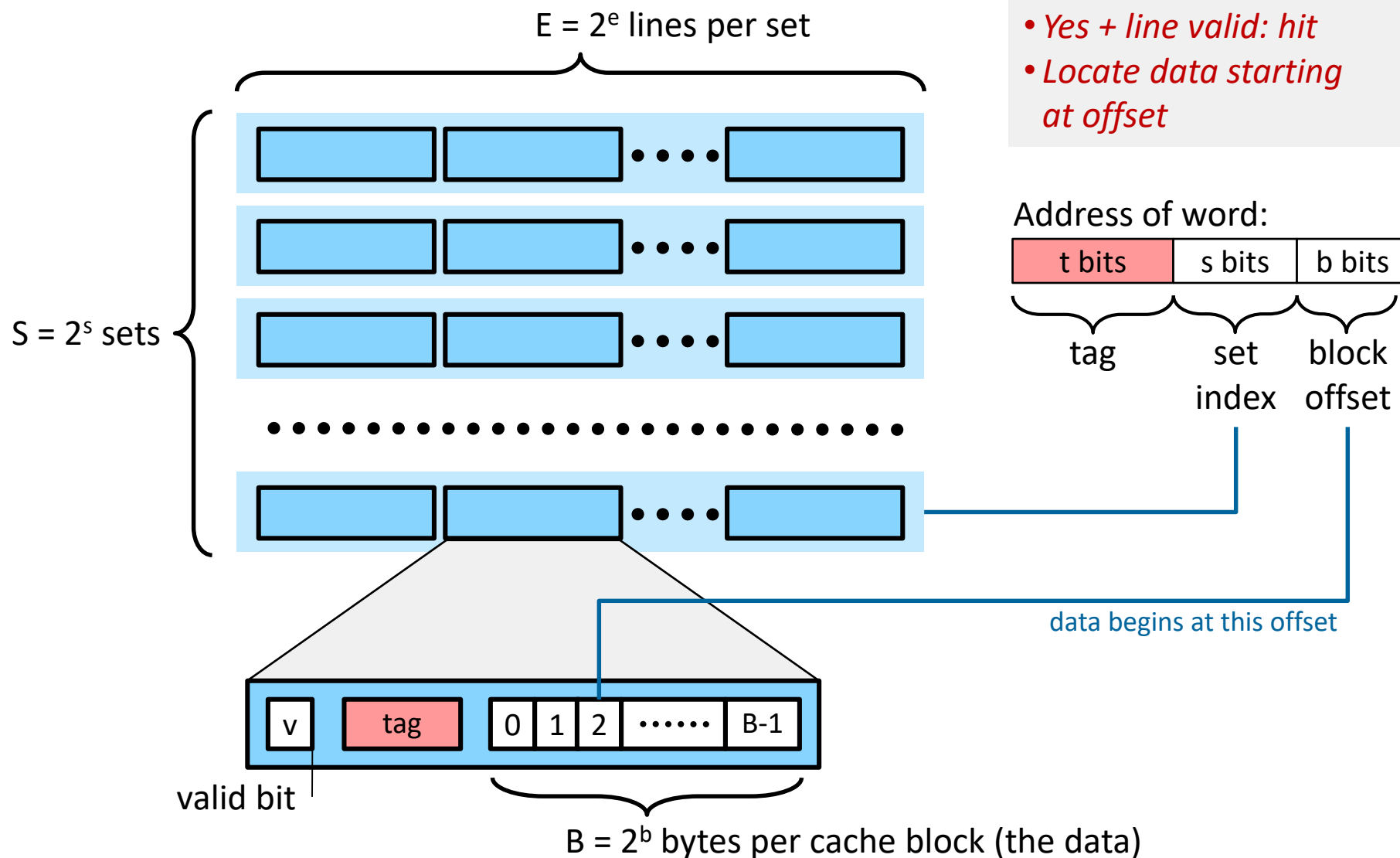




# General Cache Organization (S, E, B)



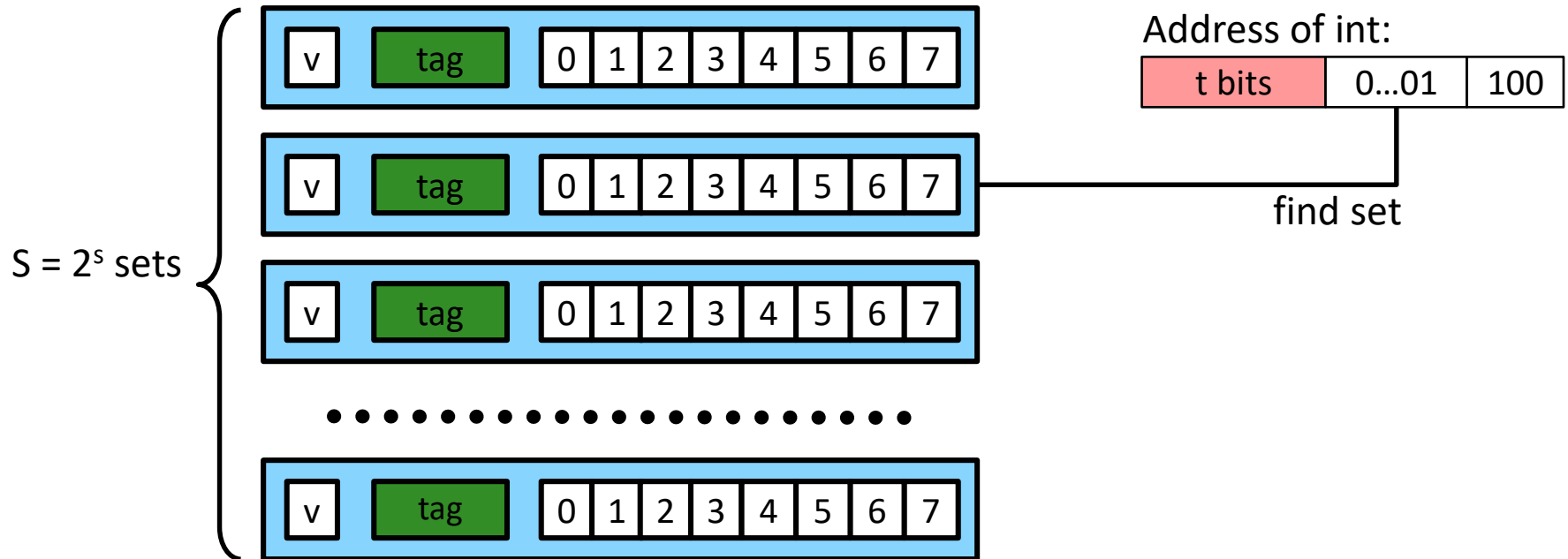
# Cache Read



# Example: Direct Mapped Cache ( $E = 1$ )

Direct mapped: One line per set

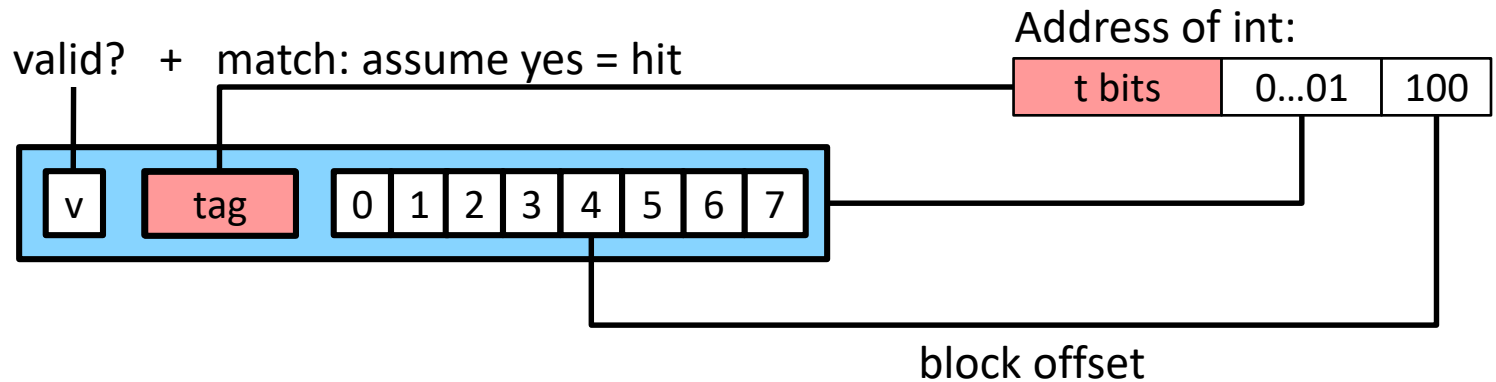
Assume: cache block size 8 bytes



# Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set

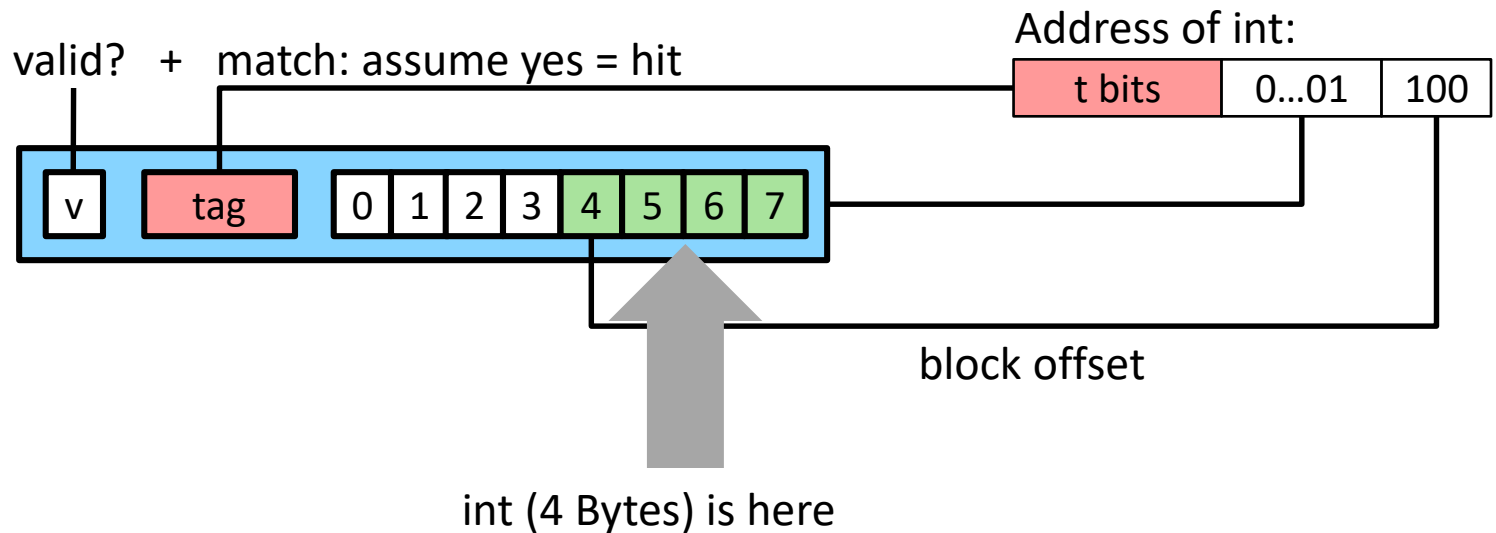
Assume: cache block size 8 bytes



# 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

Address trace (reads, one byte per read):

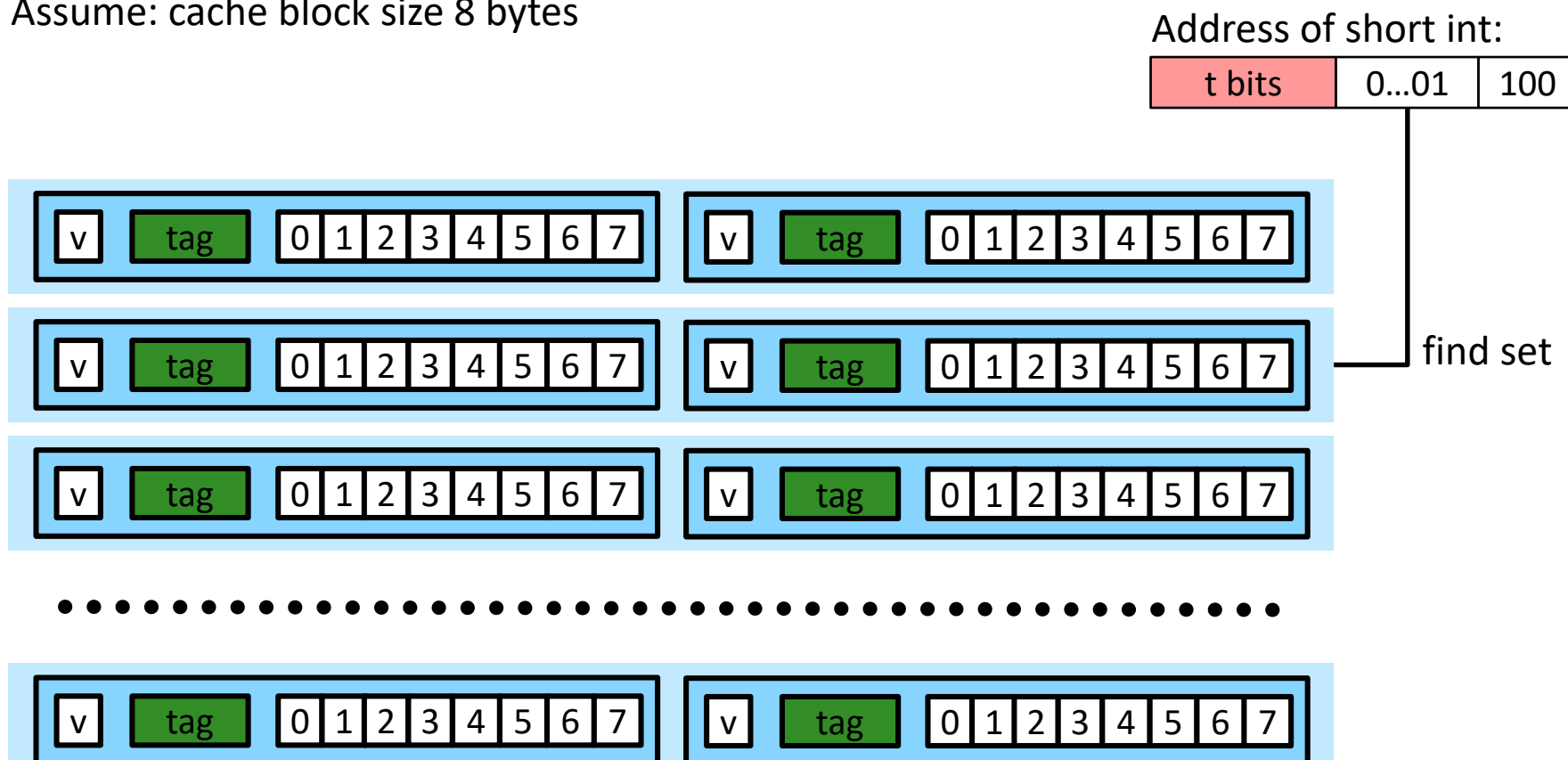
0	[0000 <sub>2</sub> ],	miss
1	[0001 <sub>2</sub> ],	hit
7	[0111 <sub>2</sub> ],	miss
8	[1000 <sub>2</sub> ],	miss
0	[0000 <sub>2</sub> ]	miss

	v	Tag	Block
Set 0	1	0	M[0-1]
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

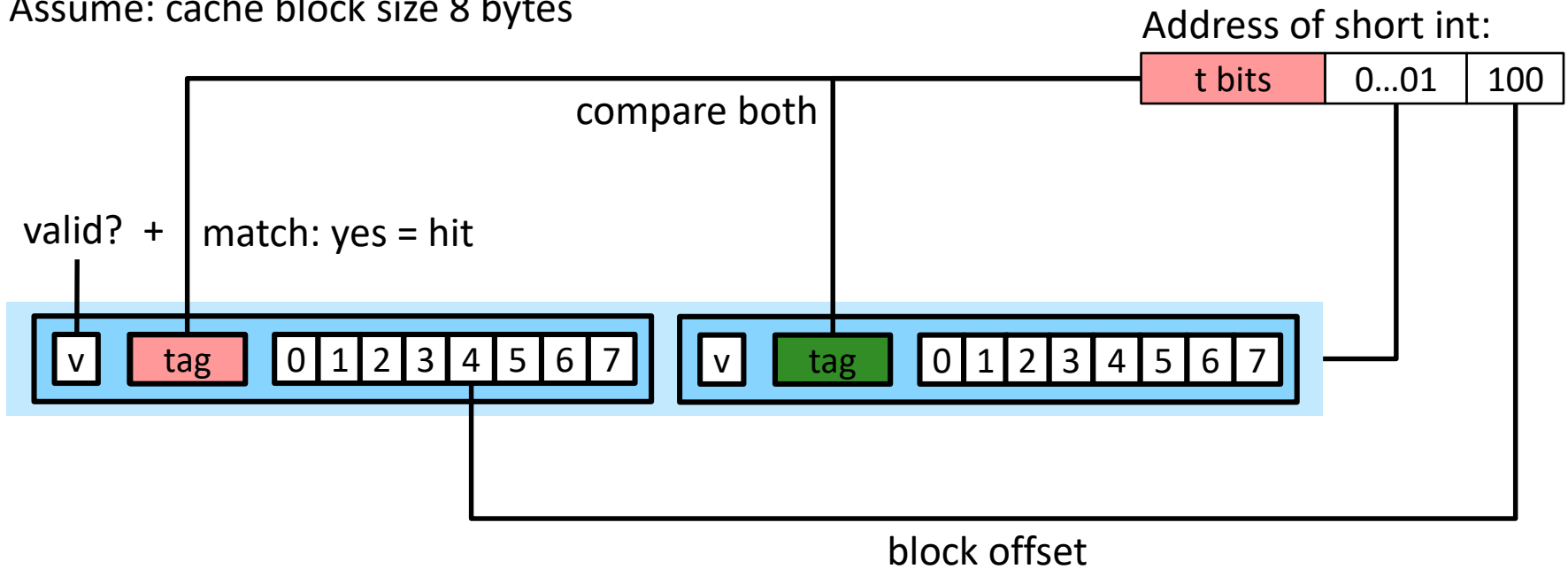
Assume: cache block size 8 bytes



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

E = 2: Two lines per set

Assume: cache block size 8 bytes

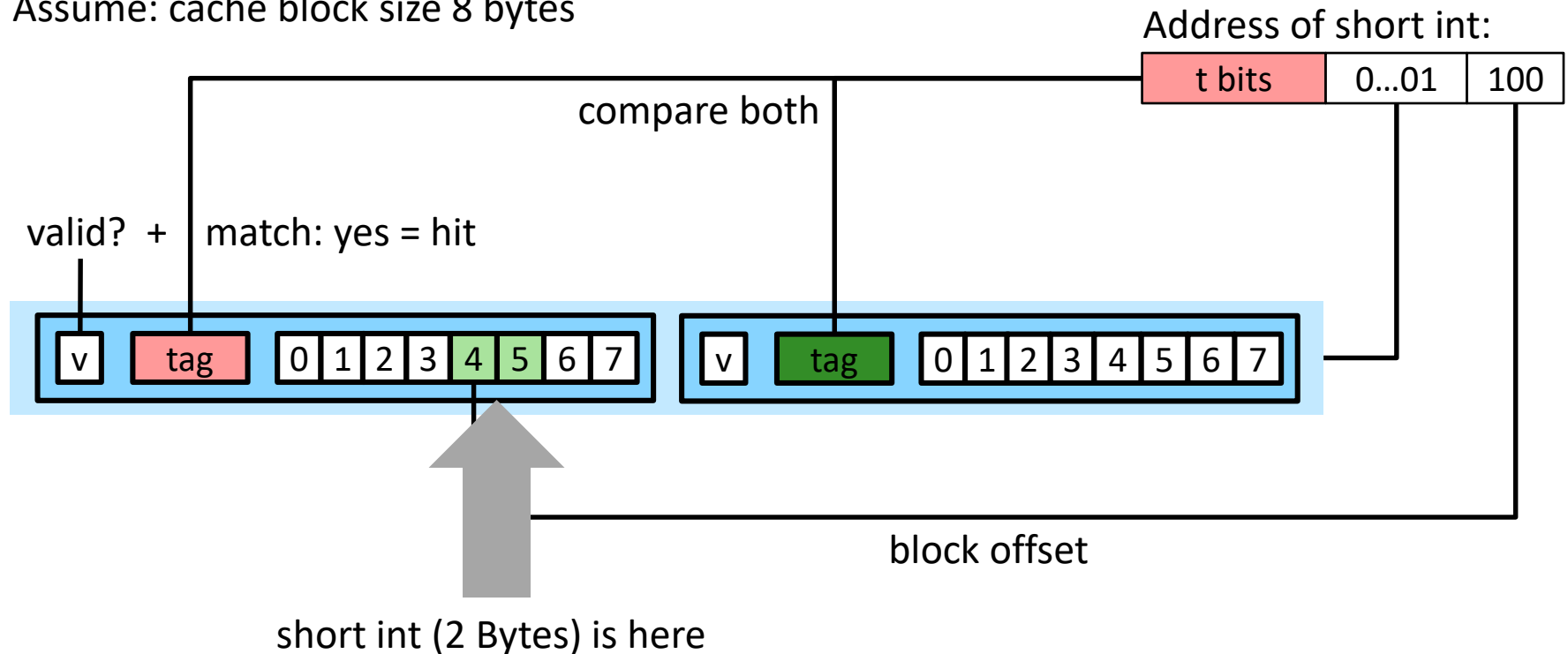




# E-way Set Associative Cache (Here: E = 2)

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

# Which block should be replaced on a miss?

- No choice in a direct mapped cache
- In an associative cache, which block from set should be evicted when the set becomes full?
  - **Random**
  - **Least Recently Used (LRU)**
    - LRU cache state must be updated on every access
    - True implementation only feasible for small sets (2-way)
    - **Pseudo-LRU binary tree often used for 4-8 way**
  - **First In, First Out (FIFO)** aka Round-Robin
    - Used in highly associative caches
  - **Not Most Recently Used (NMRU)**
    - FIFO with exception for most recently used block(s)

# LRU policy



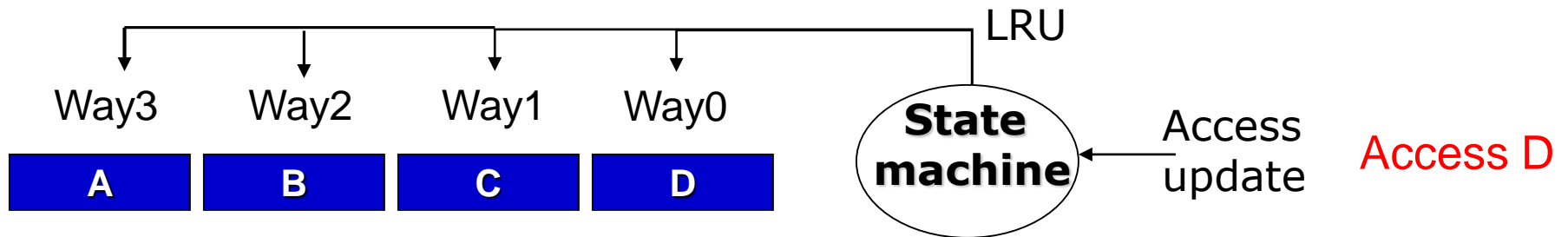
	MRU	MRU-1	LRU+1	LRU
	A	B	C	D
Access C	C	A	B	D
Access D	D	C	A	B
Access E	E	D	C	A
Access C	C	E	D	A
Access G	G	C	E	D

Red arrows indicate the movement of the accessed element from the LRU column to the MRU column in each row.

**MISS, replacement needed** (for Access E)

**MISS, replacement needed** (for Access C)

# LRU From Hardware Perspective



**LRU policy increases cache access times**

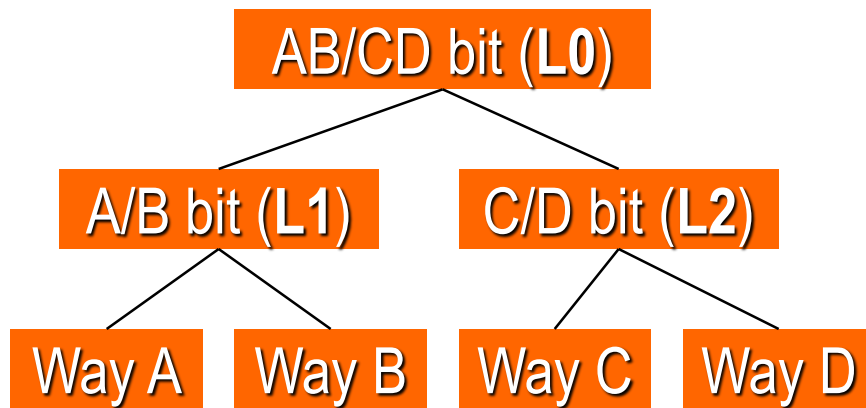
**Additional hardware bits needed for LRU state machine**

# LRU Algorithms

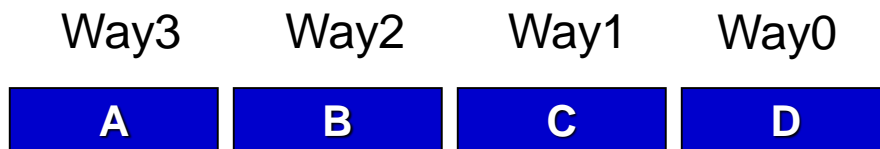


- True LRU
  - **Expensive** in terms of speed and hardware
  - Need to remember the order in which all  $N$  lines were last accessed
  - $N!$  scenarios –  **$O(\log N!) \approx O(N \log N)$**  LRU bits
    - 2-ways  $\rightarrow$  AB BA = 2 = 2!
    - 3-ways  $\rightarrow$  ABC ACB BAC BCA CAB CBA = 6 = 3!
- Pseudo LRU:  **$O(N)$** 
  - Approximates LRU policy with a binary tree

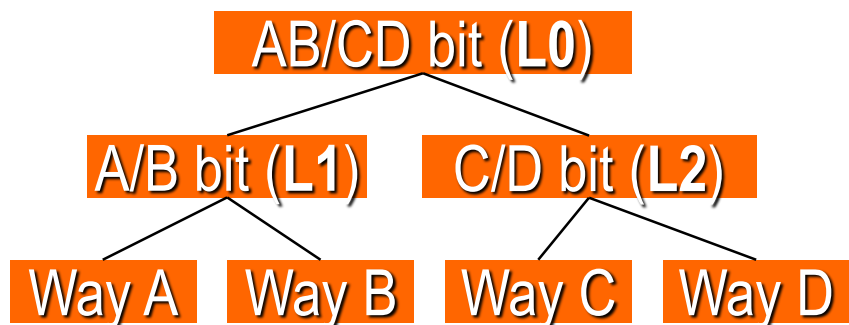
# Pseudo LRU Algorithm (4-way SA)



- Tree-based
- $O(N)$ : 3 bits for 4-way
- Cache ways are the leaves of the tree
- Combine ways as we proceed towards the root of the tree



# Pseudo LRU Algorithm



•  $L2L1L0 = 000$ ,  
there is a hit in Way B, what  
is the new updated  $L2L1L0$ ?

## LRU update algorithm

	CD	AB	AB/CD
Way hit	L2	L1	L0
Way A	---	1	1
Way B	---	0	1
Way C	1	---	0
Way D	0	---	0

- Less hardware than LRU
- Faster than LRU

•  $L2L1L0 = 001$ ,  
a way needs to be replaced,  
which way would be chosen?

## Replacement Decision

CD	AB	AB/CD	
L2	L1	L0	Way to replace
X	0	0	Way A
X	1	0	Way B
0	X	1	Way C
1	X	1	Way D



# Not Recently Used (NRU)

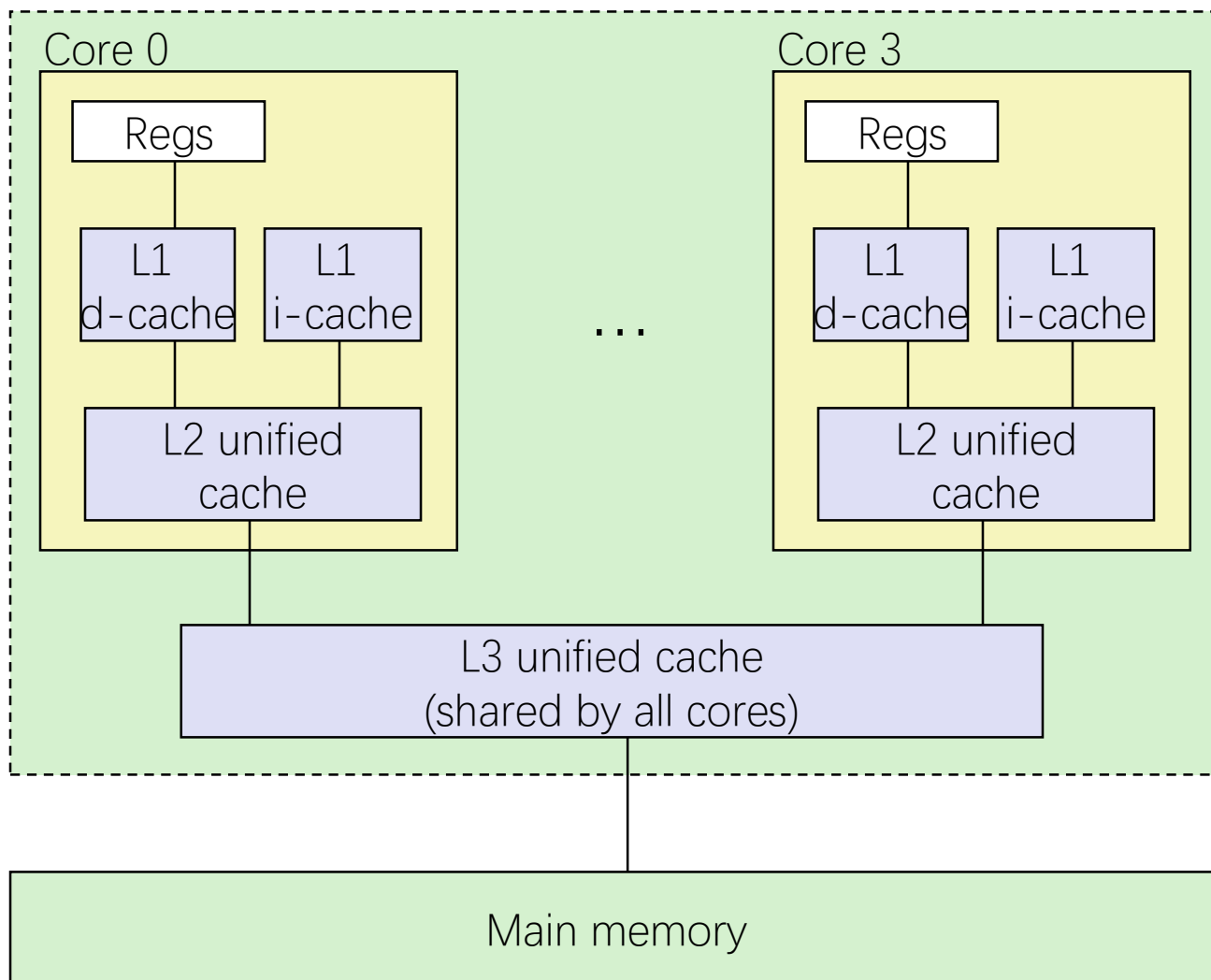
- Use R(eferenced) and M(odified) bits
  - 0 (not referenced or not modified)
  - 1 (referenced or modified)
- Classify lines into
  - C0: R=0, M=0
  - C1: R=0, M=1
  - C2: R=1, M=0
  - C3: R=1, M=1
- Chose the victim from the lowest class
  - $(C3 > C2 > C1 > C0)$
- Periodically clear R and M bits

# 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

Processor package



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)  
=  $1 - \text{hit rate}$
- 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



# Today

- Memory hierarchy
- Basic idea of Cache
- Cache memory 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));
}
```

*mountain/mountain.c*

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

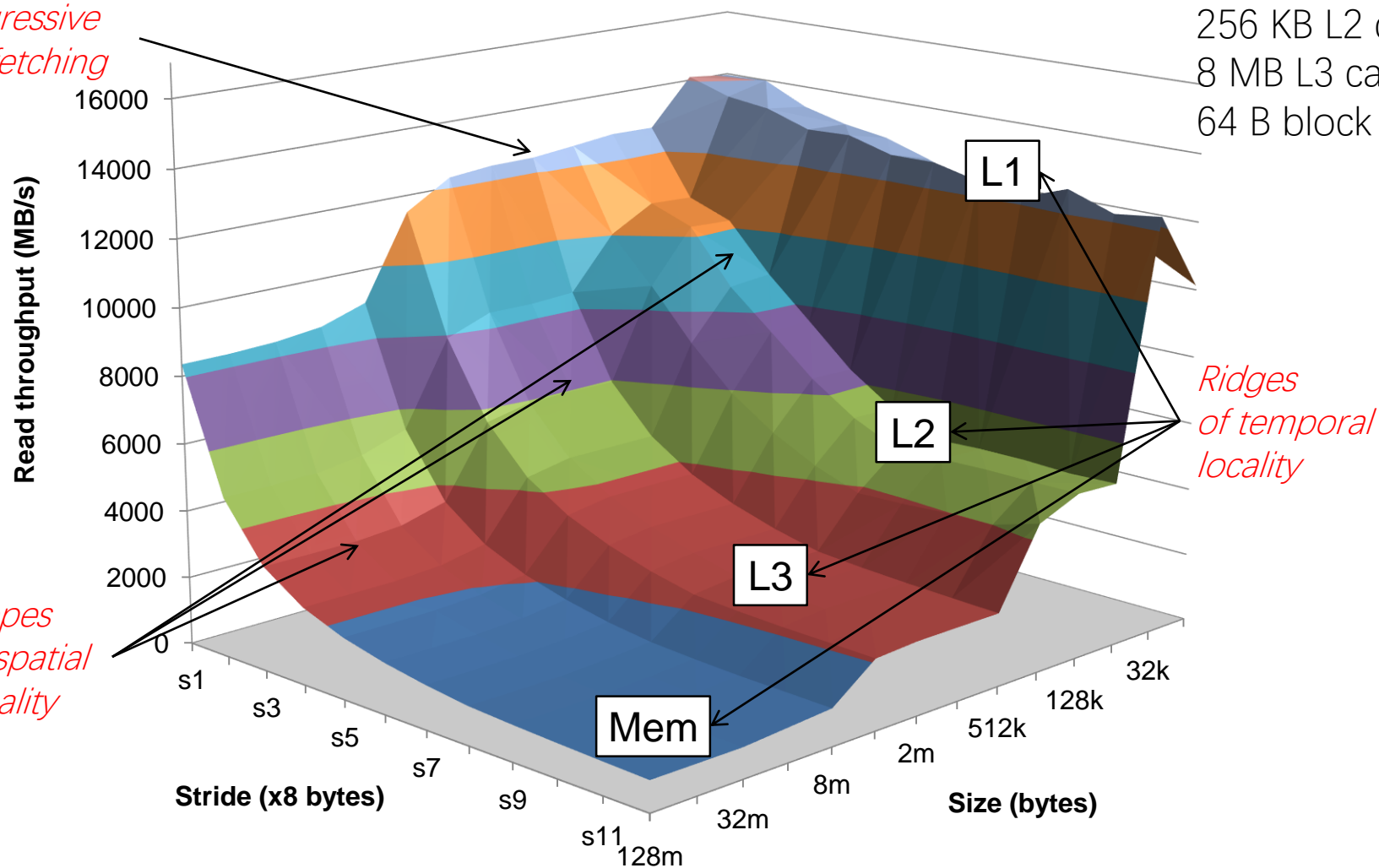
For each `elems` and `stride`:

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

# 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

*Aggressive  
prefetching*





# Today

- Memory hierarchy
- Basic idea of Cache
- Cache memory 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

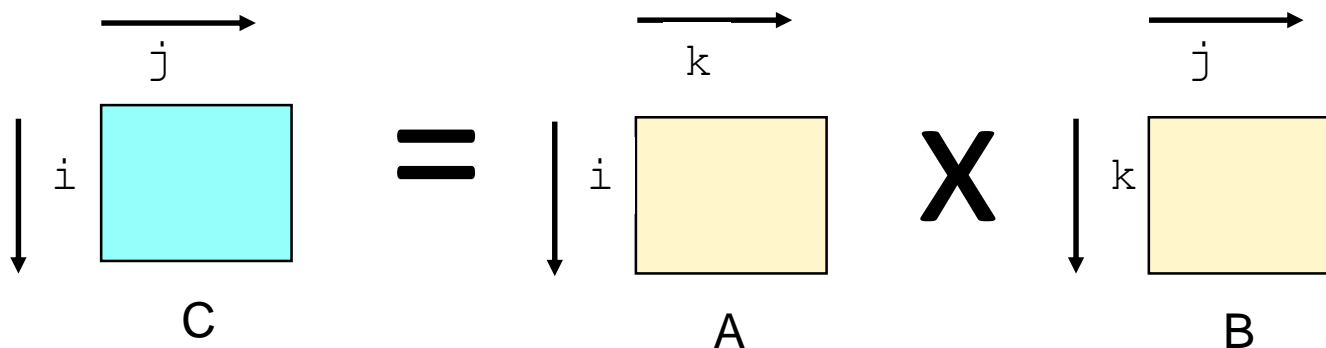
```
/* ijk */
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

*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 ( $i = 0; i < N; i++$ )  
     $\text{sum} += a[0][i];$
  - accesses successive elements
  - if block size ( $B$ ) >  $\text{sizeof}(a_{ij})$  bytes, exploit spatial locality
    - $\text{miss rate} = \text{sizeof}(a_{ij}) / B$
- Stepping through rows in one column:
  - for ( $i = 0; i < n; i++$ )  
     $\text{sum} += a[i][0];$
  - accesses distant elements
  - no spatial locality!
    - $\text{miss rate} = 1$  (i.e. 100%)

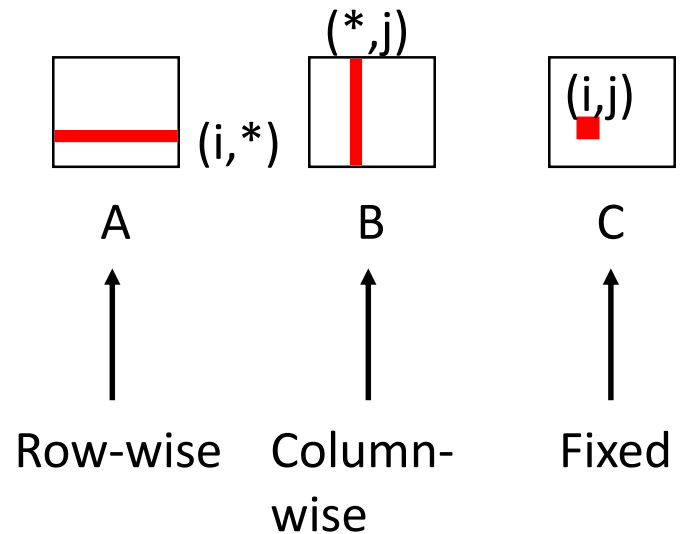
# Matrix Multiplication (ijk)



```
/* 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*

Inner loop:



Misses per inner loop iteration:

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

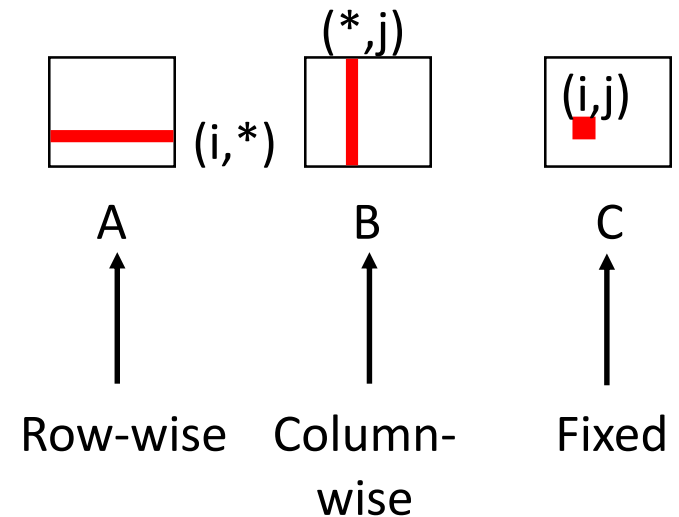
# Matrix Multiplication (jik)



```
/* 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:

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

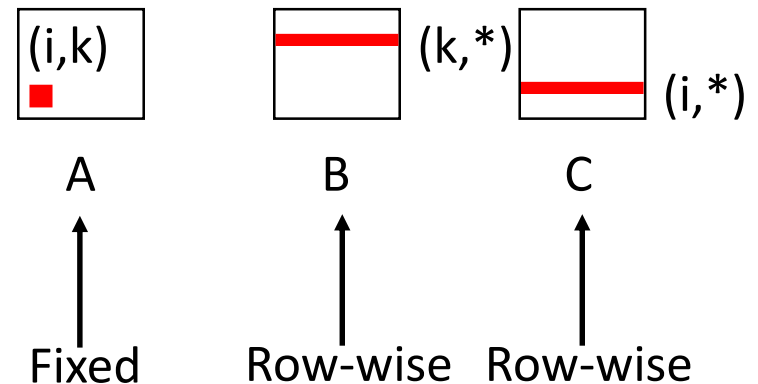


# Matrix Multiplication (kij)

```
/* 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*

Inner loop:



Misses per inner loop iteration:

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

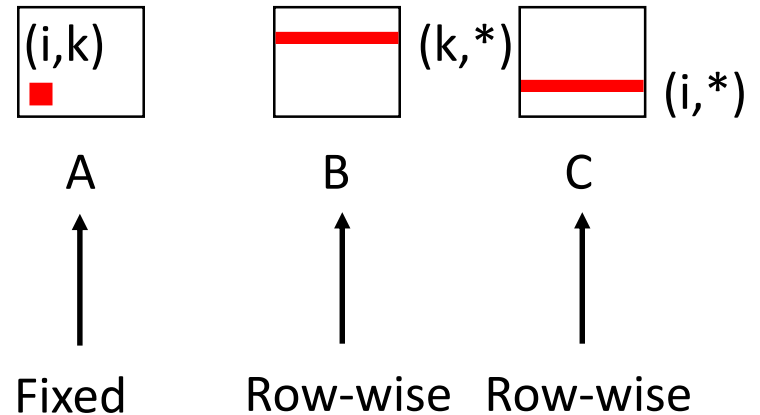
# Matrix Multiplication (ikj)



```
/* 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*

Inner loop:



Misses per inner loop iteration:

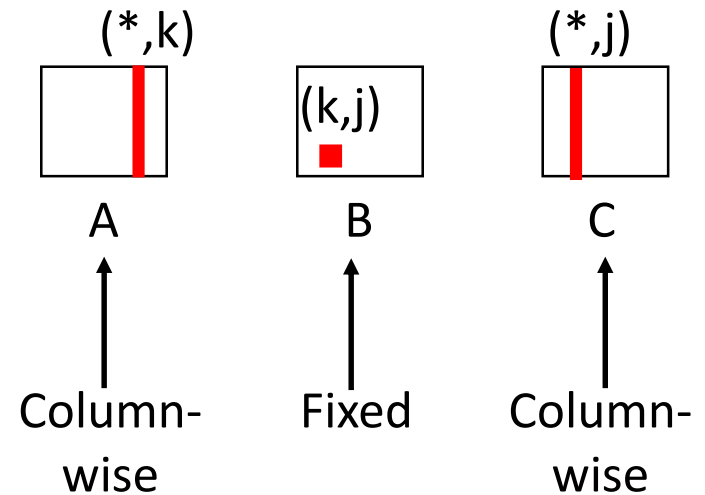
<u>A</u>	<u>B</u>	<u>C</u>
0.0	0.25	0.25

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

*matmult/mm.c*

Inner loop:



Misses per inner loop iteration:

A  
1.0

B  
0.0

C  
1.0

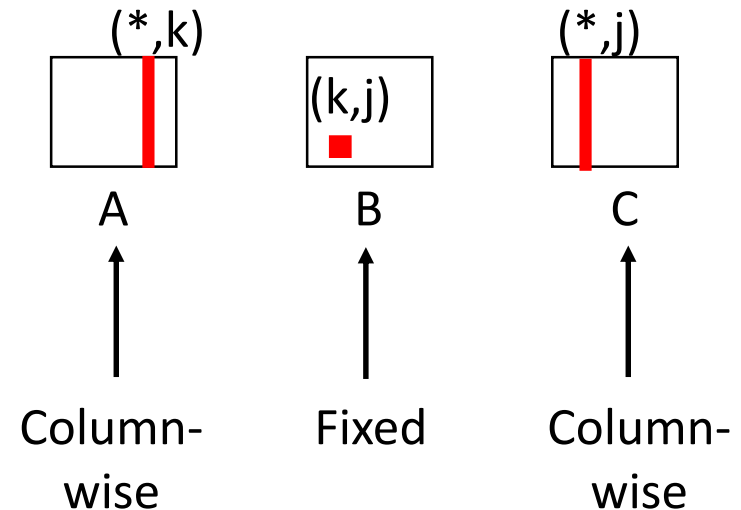
# Matrix Multiplication (kji)



```
/* 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;  
    }  
}
```

ijk (& jik):

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

```
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];  
    }  
}
```

kij (& ikj):

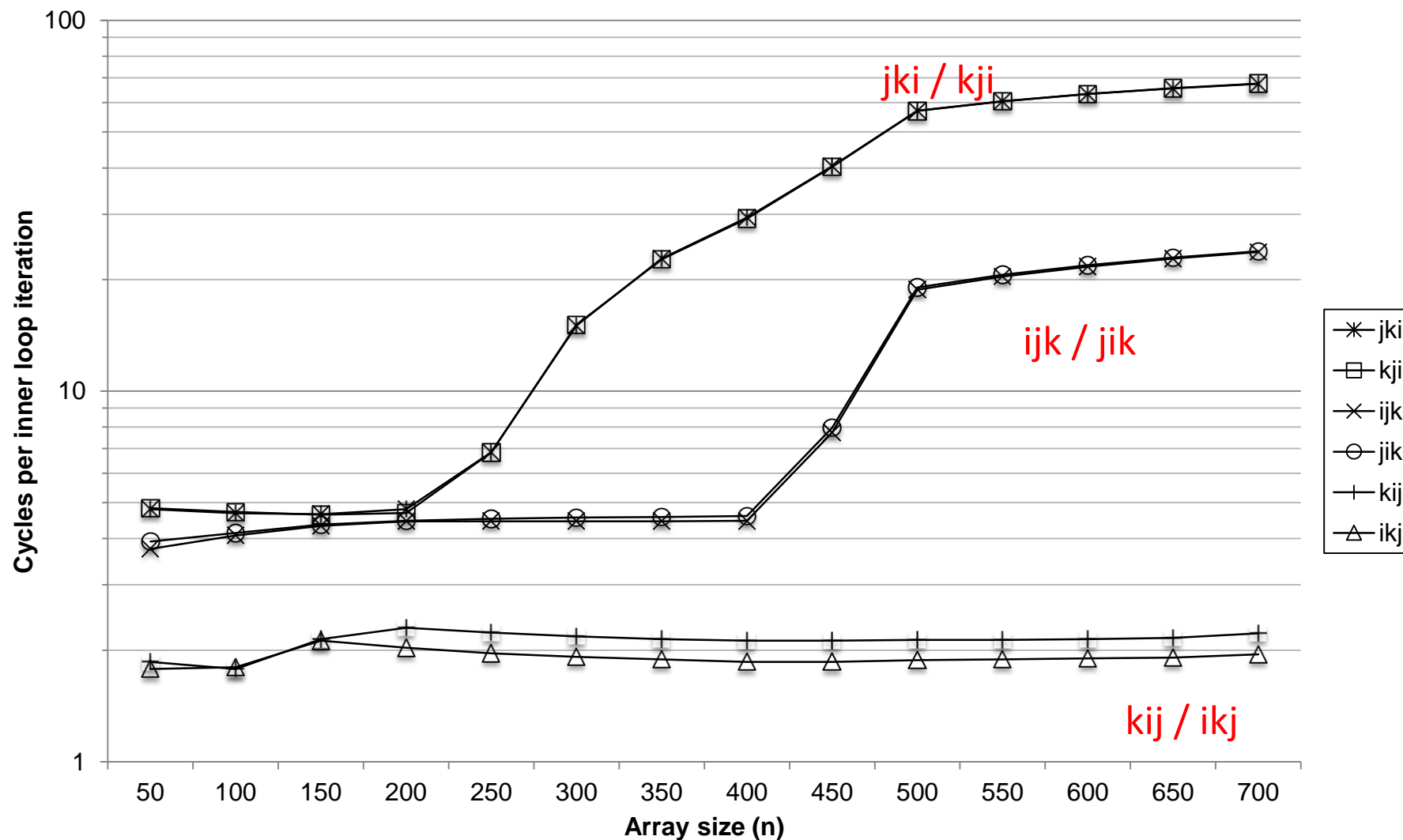
- 2 loads, 1 store
- misses/iter = 0.5

```
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;  
    }  
}
```

jki (& kji):

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

# Core i7 Matrix Multiply Performance





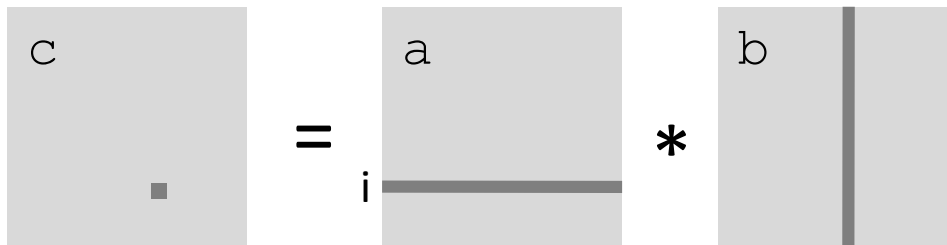
# Today

- Memory hierarchy
- Basic idea of Cache
- 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: 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];
}
```



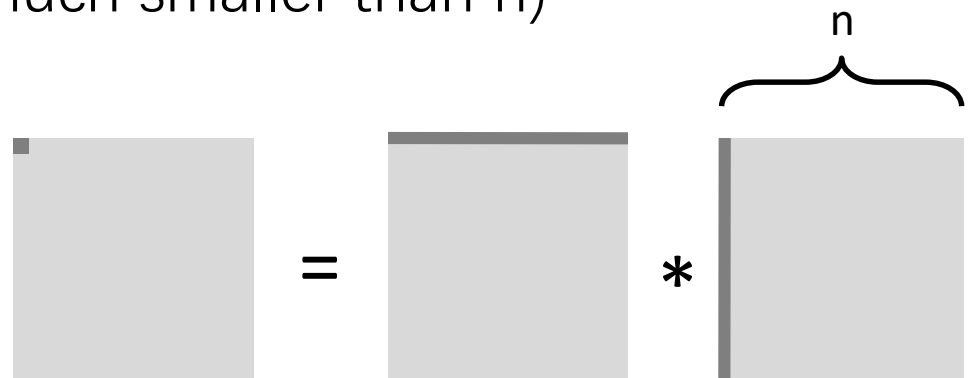


# Cache Miss Analysis

- 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)

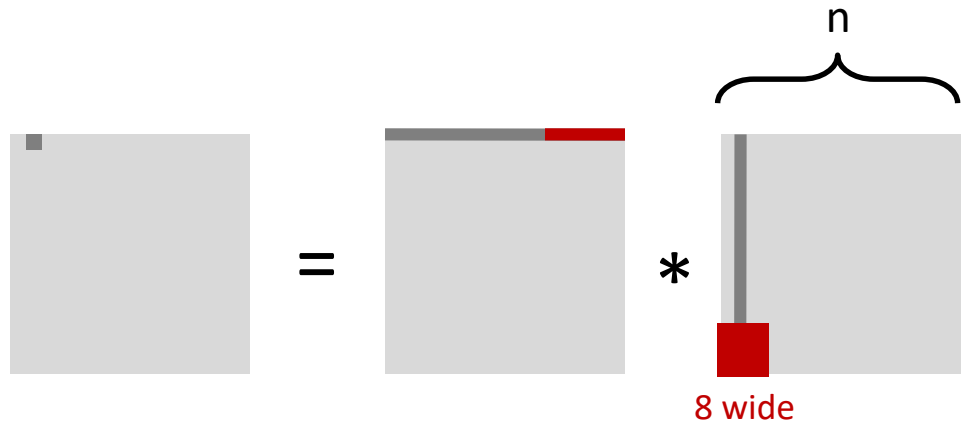


# Cache Miss Analysis

- 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

```

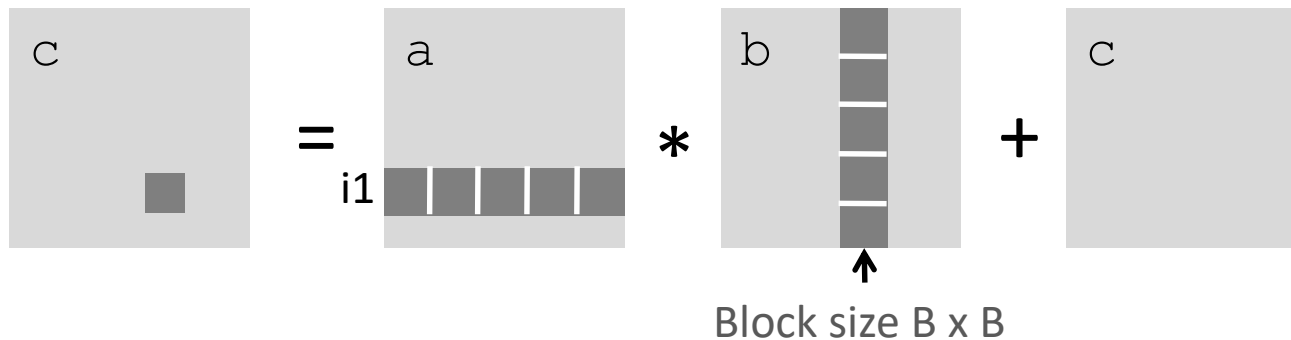
c = (double *) calloc(sizeof(double), n*n);

/* 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*

j1



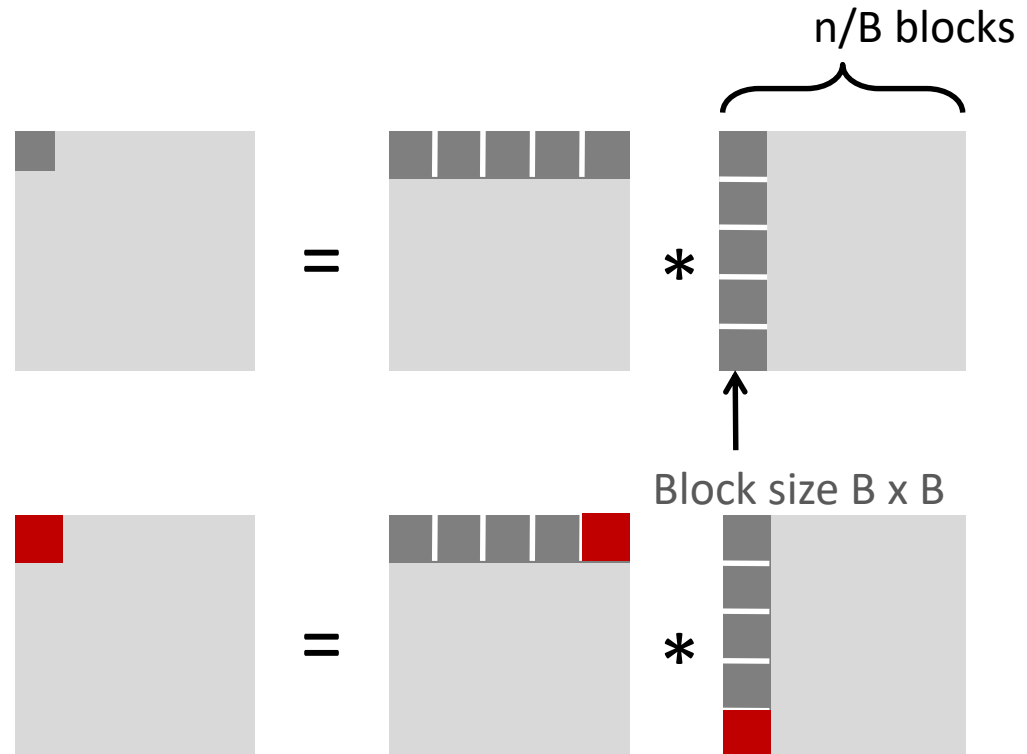
# Cache Miss Analysis

- 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)

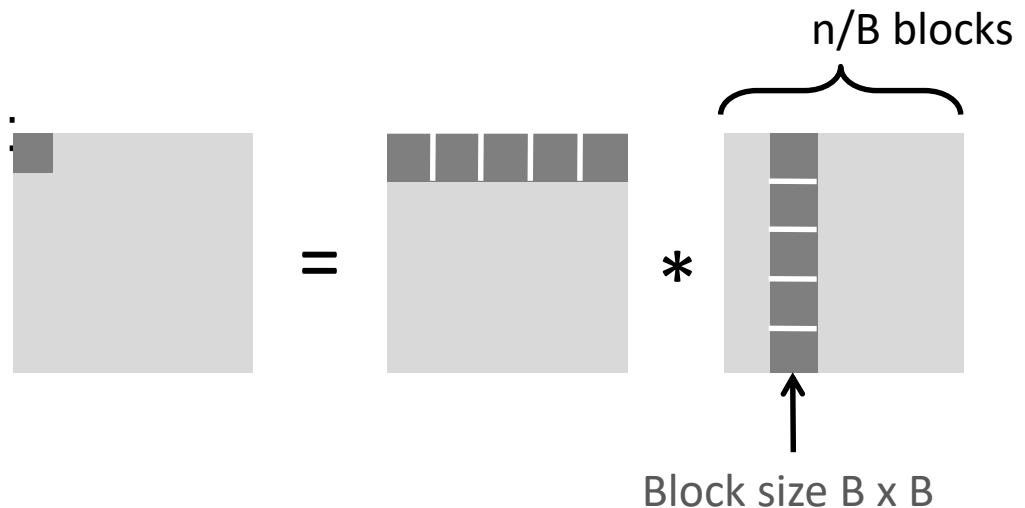
# Cache Miss Analysis

- 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

# Cache Summary



- 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 sequentially with stride 1.
  - Try to maximize temporal locality by using a data object as often as possible once it's read from memory.

谢谢！

