The Memory Hierarchy

Today

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

Random-Access Memory (RAM)

- **Key features**
    - RAM is traditionally packaged as a chip.
    - Basic storage unit is normally a cell (one bit per cell).
    - Multiple RAM chips form a memory.

- **RAM comes in two varieties:**
    - SRAM (Static RAM)
    - DRAM (Dynamic RAM)

SRAM vs DRAM Summary

	Trans. per bit	Access time	Needs refresh?	Needs EDC?	Cost	Applications
SRAM	4 or 6	1X	No	Maybe	100x	Cache memories
DRAM	1	10X	Yes	Yes	1X	Main memories, frame buffers

Nonvolatile Memories

- **DRAM and SRAM are volatile memories**
  - Lose information if powered off.
- **Nonvolatile memories retain value even if powered off**
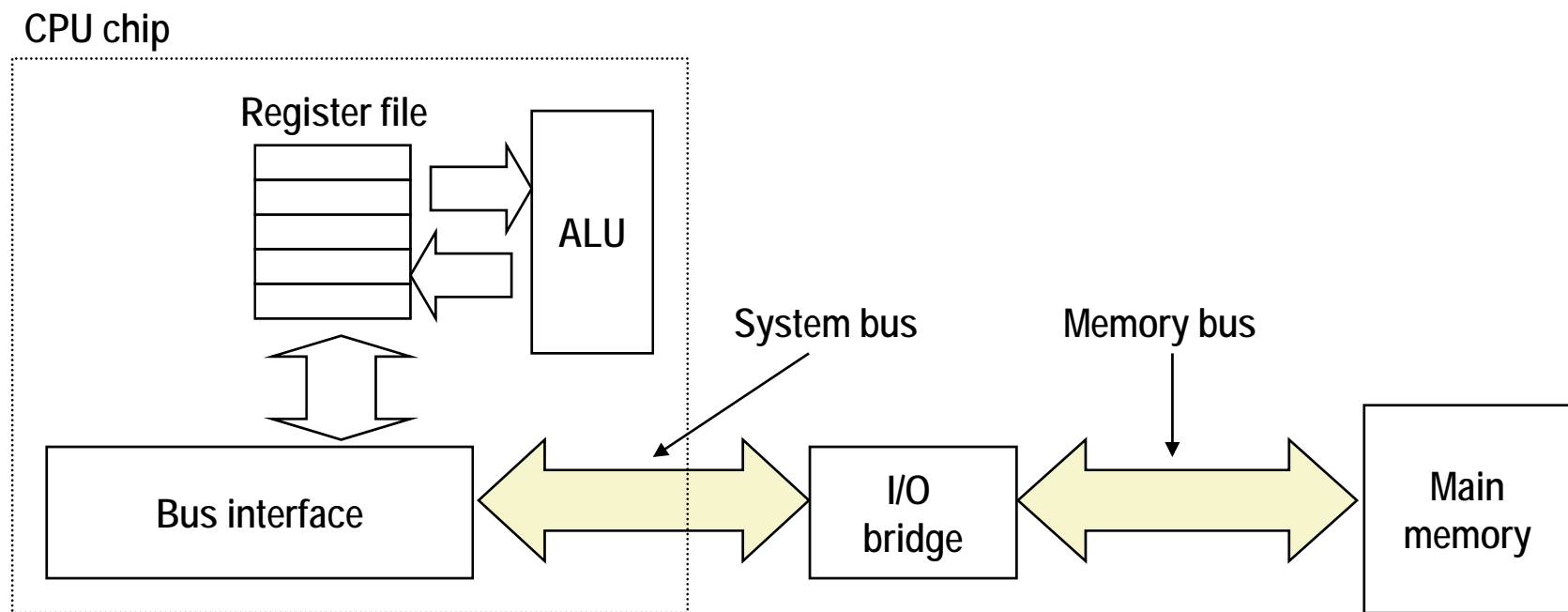 - Read-only memory (ROM): programmed during production
  - Programmable ROM (PROM): can be programmed once
  - Eraseable PROM (EPROM): can be bulk erased (UV, X-Ray)
  - Electrically eraseable PROM (EEPROM): electronic erase capability
  - Flash memory: EEPROMs. with partial (block-level) erase capability
    - Wears out after about 100,000 erasings
- **Uses for Nonvolatile Memories**
  - Firmware programs stored in a ROM (BIOS, controllers for disks, network cards, graphics accelerators, security subsystems,…)
  - Solid state disks (replace rotating disks in thumb drives, smart phones, mp3 players, tablets, laptops,…)
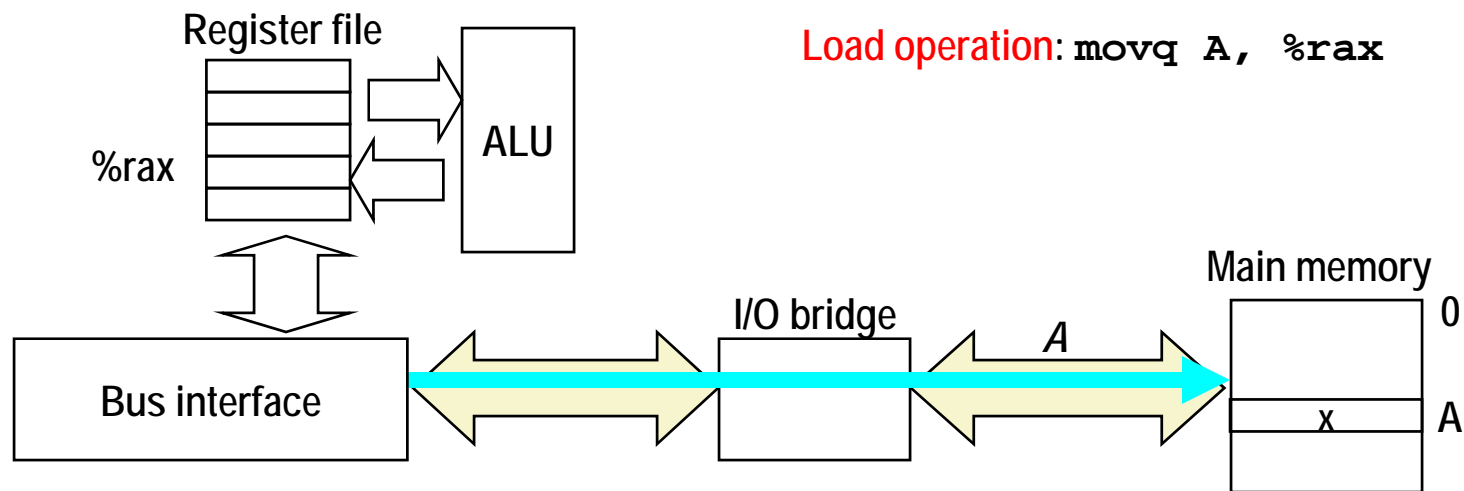 - Disk caches

Traditional Bus Structure Connecting CPU and Memory

- A **bus** is a collection of parallel wires that carry address, data, and control signals.
- Buses are typically shared by multiple devices.

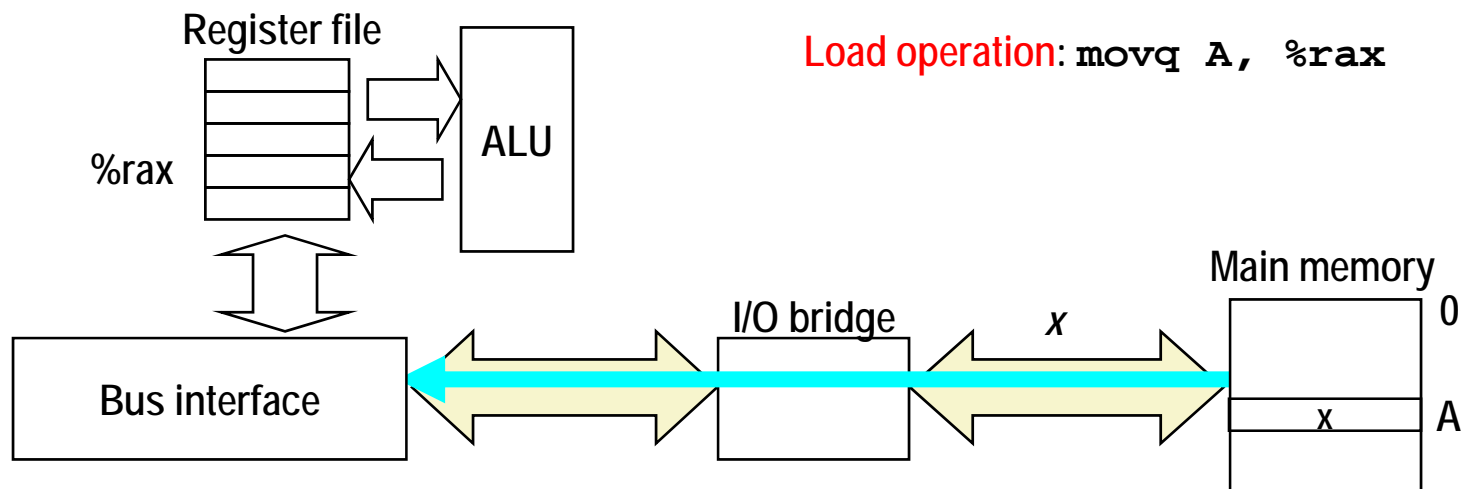CPU chip

Register file

ALU

System bus

Memory bus

Bus interface

I/O bridge

Main memory

Memory Read Transaction (1)

CPU places address A on the memory bus.

Load operation: `movq A, %rax`

Register file

%rax

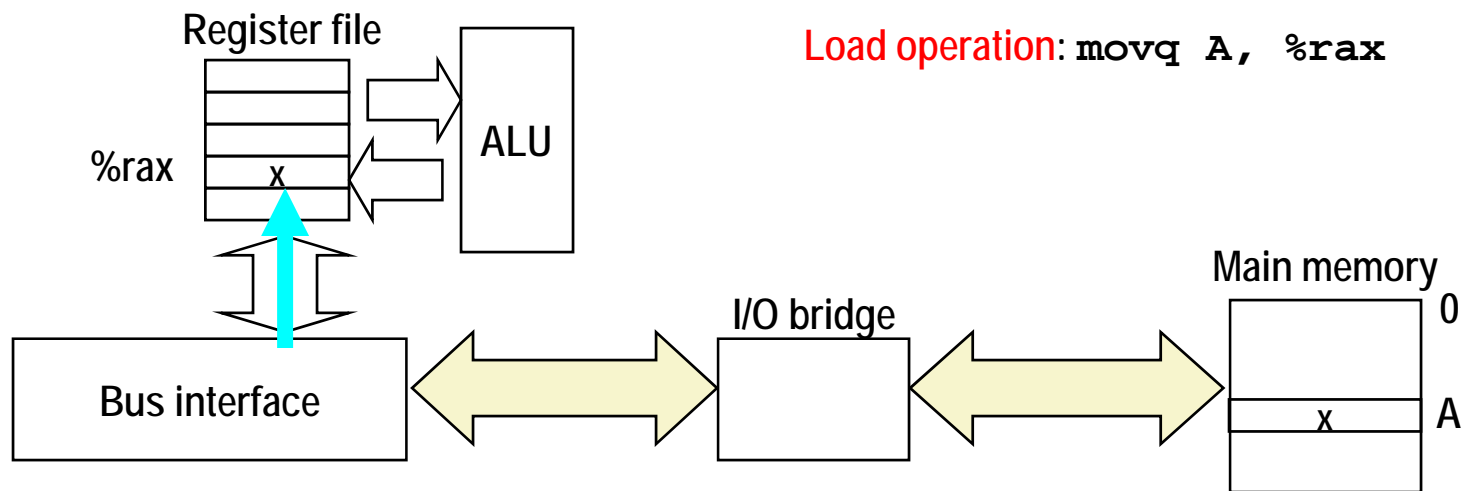ALU

Bus interface

I/O bridge

A

Main memory

0

x

A

Memory Read Transaction (2)

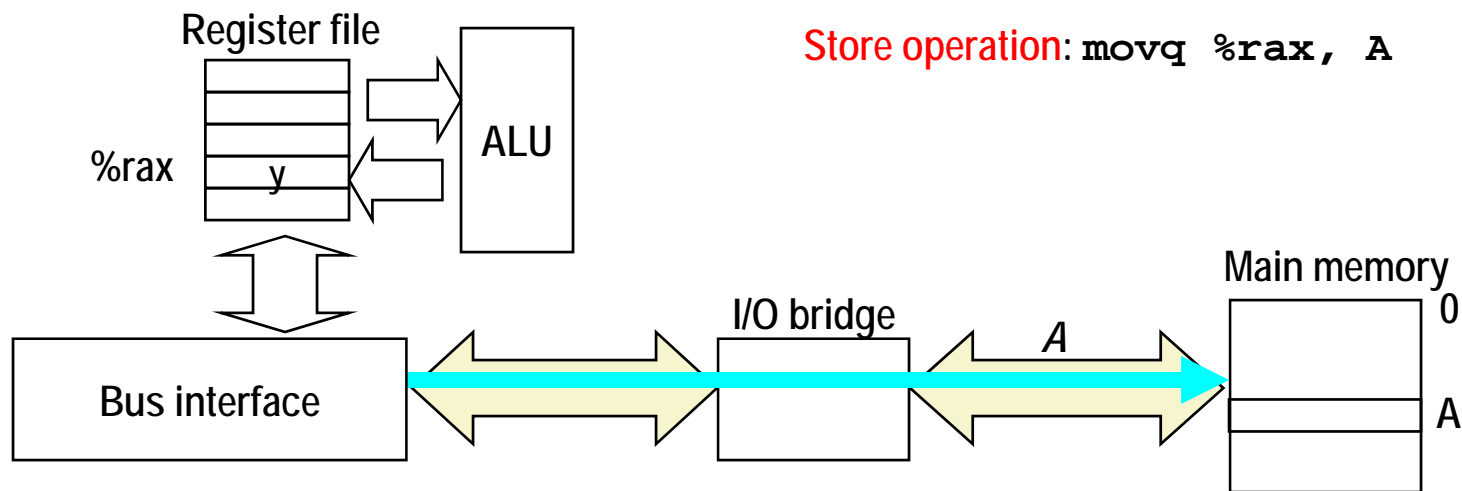- **Main memory reads A from the memory bus, retrieves word x, and places it on the bus.**

Register file

Load operation: `movq A, %rax`

ALU

%rax

Bus interface

I/O bridge

x

Main memory

0

x   A

Memory Read Transaction (3)

- CPU read word x from the bus and copies it into register %rax.

Register file

Load operation: `movq A, %rax`
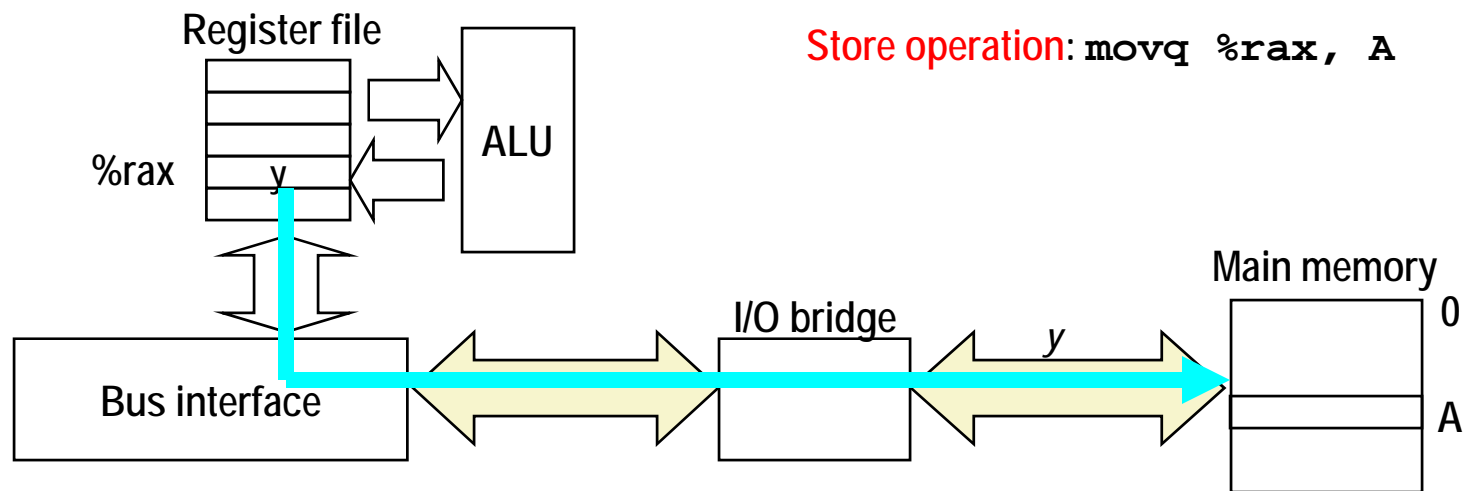
%rax | x

ALU

Bus interface

I/O bridge

Main memory

0

x | A

Memory Write Transaction (1)

CPU places address A on bus. Main memory reads it and waits for the corresponding data word to arrive.

Register file
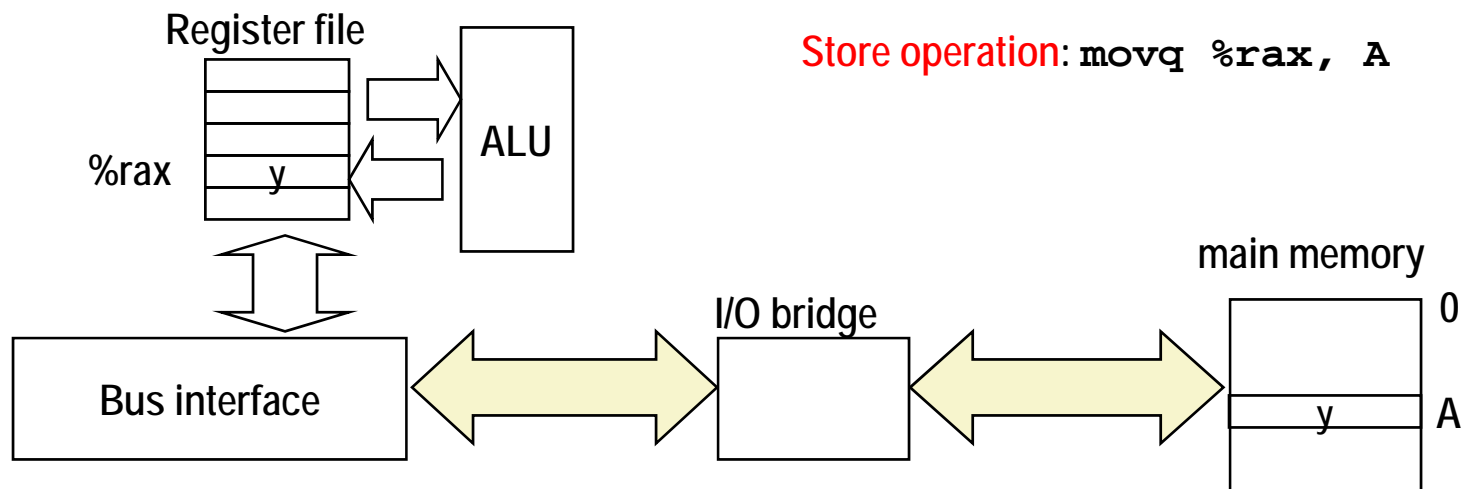
Store operation: `movq %rax, A`

%rax | y

ALU

Main memory
0

I/O bridge

A

Bus interface

A

Memory Write Transaction (2)

- CPU places data word y on the bus.

Register file

%rax   y

ALU

Store operation: `movq %rax, A`

Bus interface

I/O bridge   y

Main memory
0

A

Memory Write Transaction (3)

Main memory reads data word y from the bus and stores it at address A.

Store operation: `movq %rax, A`

Register file

%rax | y

ALU

Bus interface

I/O bridge

main memory

0

y | A

What's Inside A Disk Drive?

Spindle

Arm

Platters

Actuator

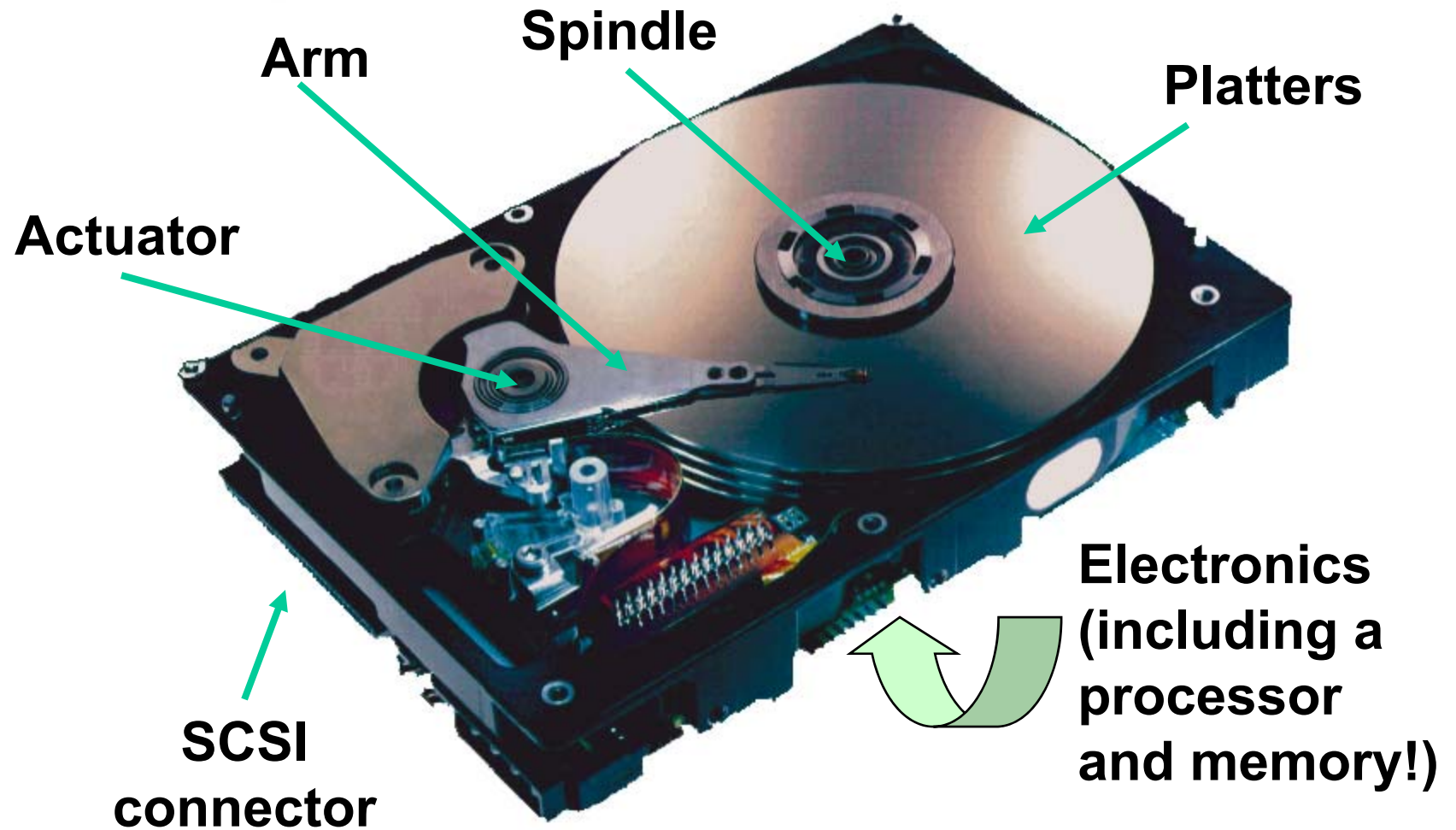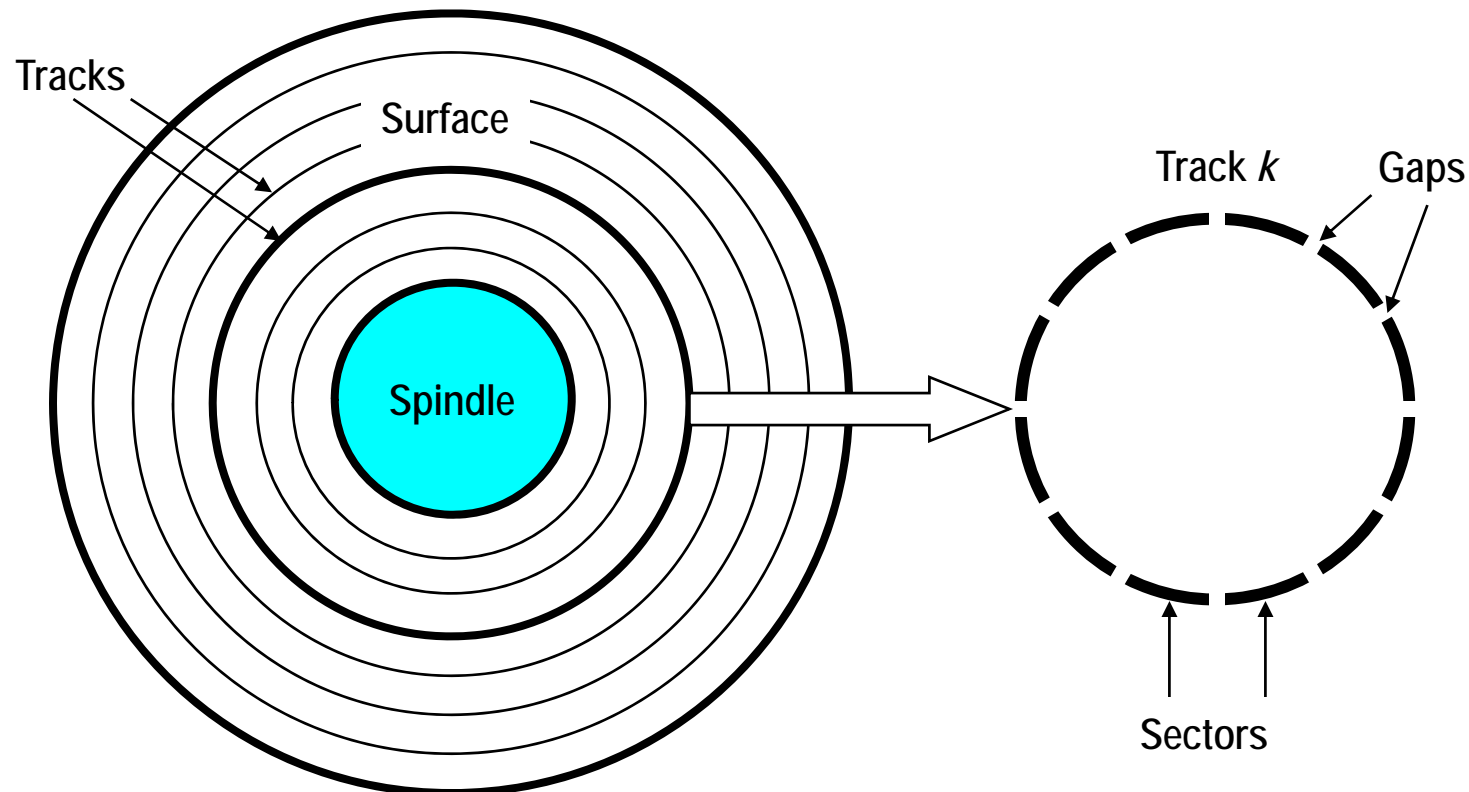Electronics (including a processor and memory!)
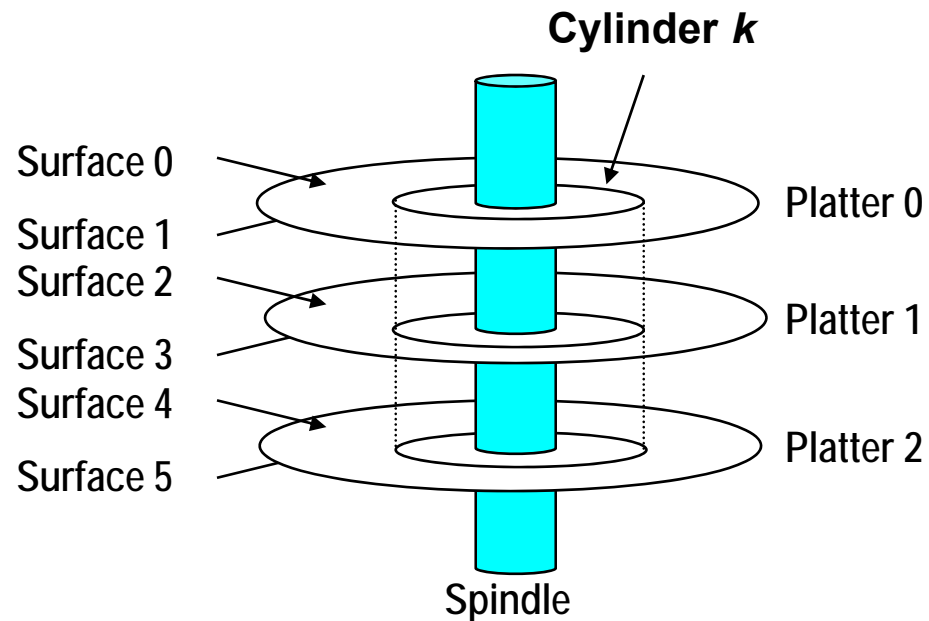
SCSI connector

Image courtesy of Seagate Technology

Disk Geometry

- **Disks consist of platters, each with two surfaces.**
- **Each surface consists of concentric rings called tracks.**
- **Each track consists of sectors separated by gaps.**

Disk Geometry (Muliple-Platter View)
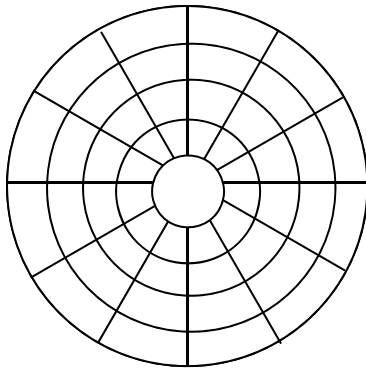
- **Aligned tracks form a cylinder.**

Disk Structure - top view of single platter

Surface organized into tracks

Tracks divided into sectors

Disk Access
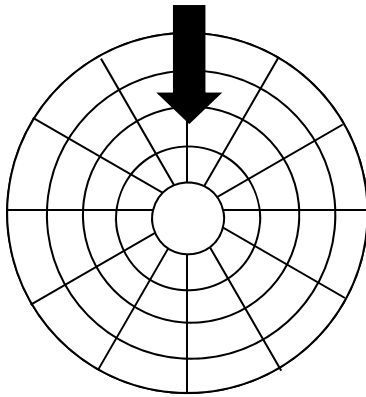
Head in position above a track

Disk Access
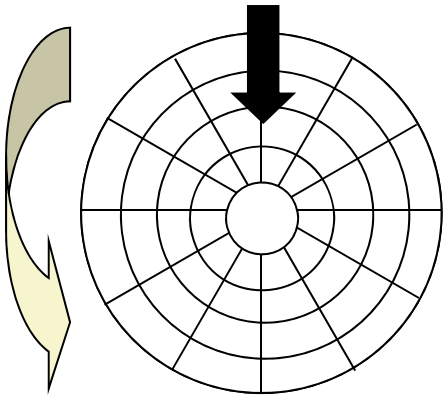
Rotation is counter-clockwise

Disk Access – Read
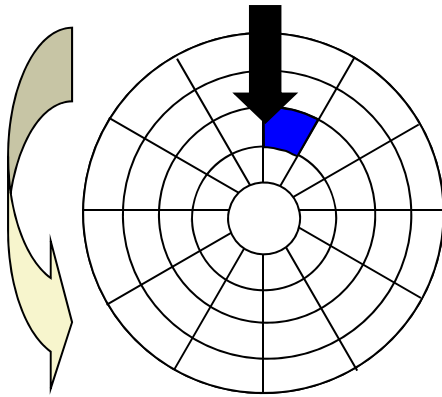
About to read blue sector

Disk Access – Read
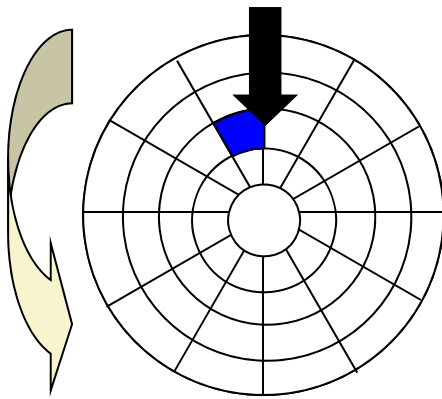
After BLUE read

After reading blue sector

Disk Access – Read
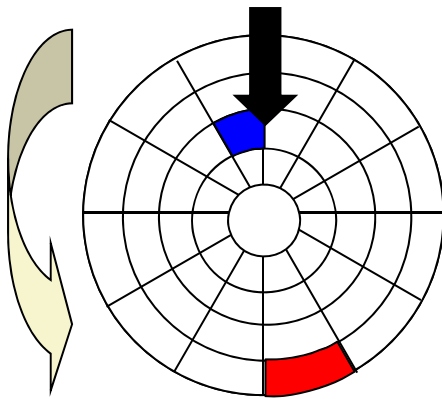
After BLUE read

Red request scheduled next

Disk Access – Seek

After BLUE read   Seek for RED
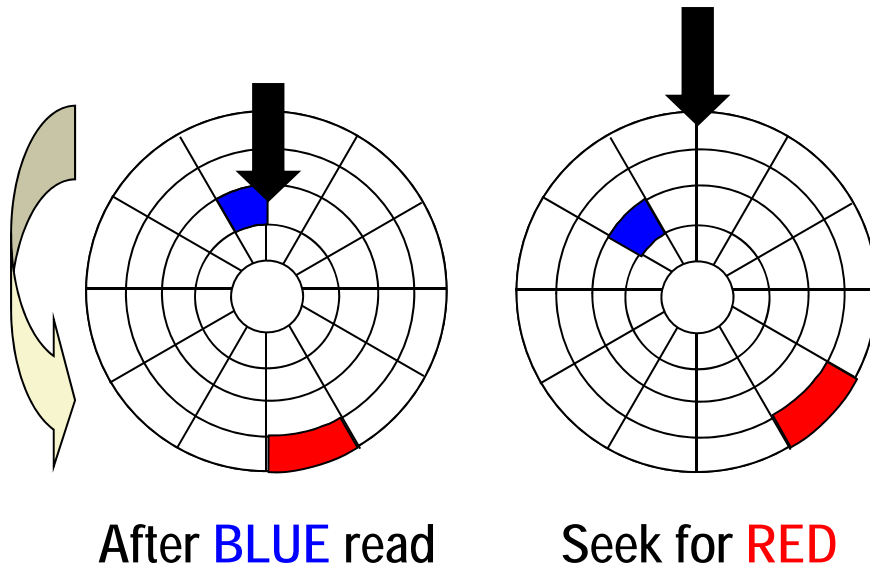
Seek to red's track

Disk Access – Rotational Latency

After BLUE read   Seek for RED   Rotational latency
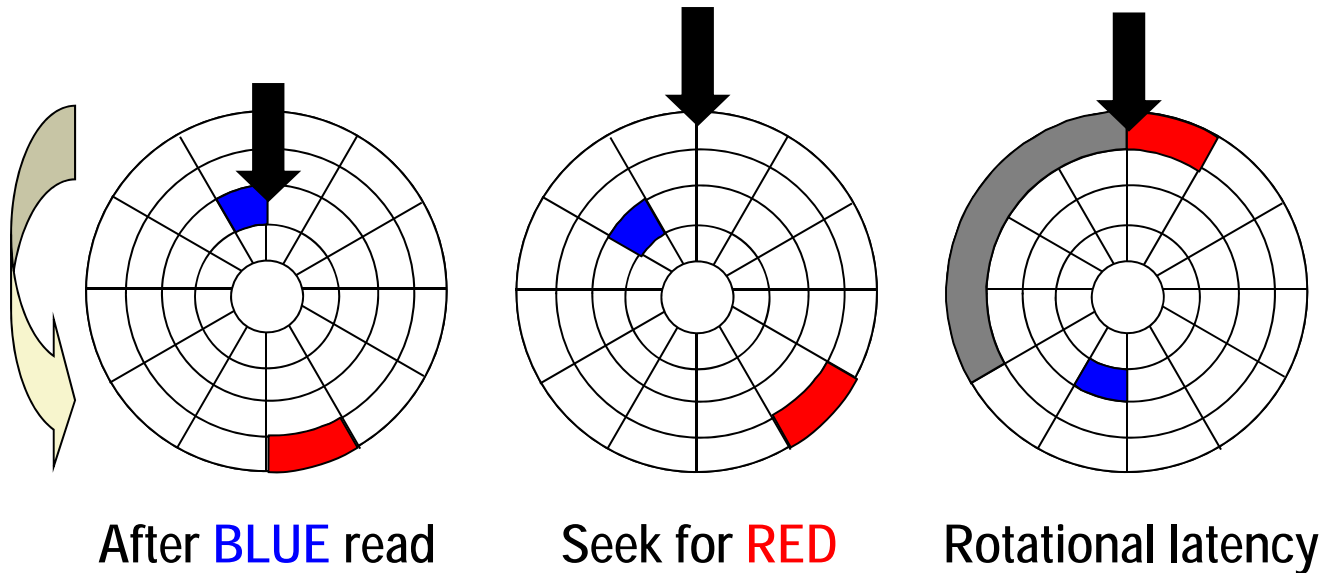
Wait for red sector to rotate around

Disk Access – Read

After BLUE read   Seek for RED   Rotational latency   After RED read
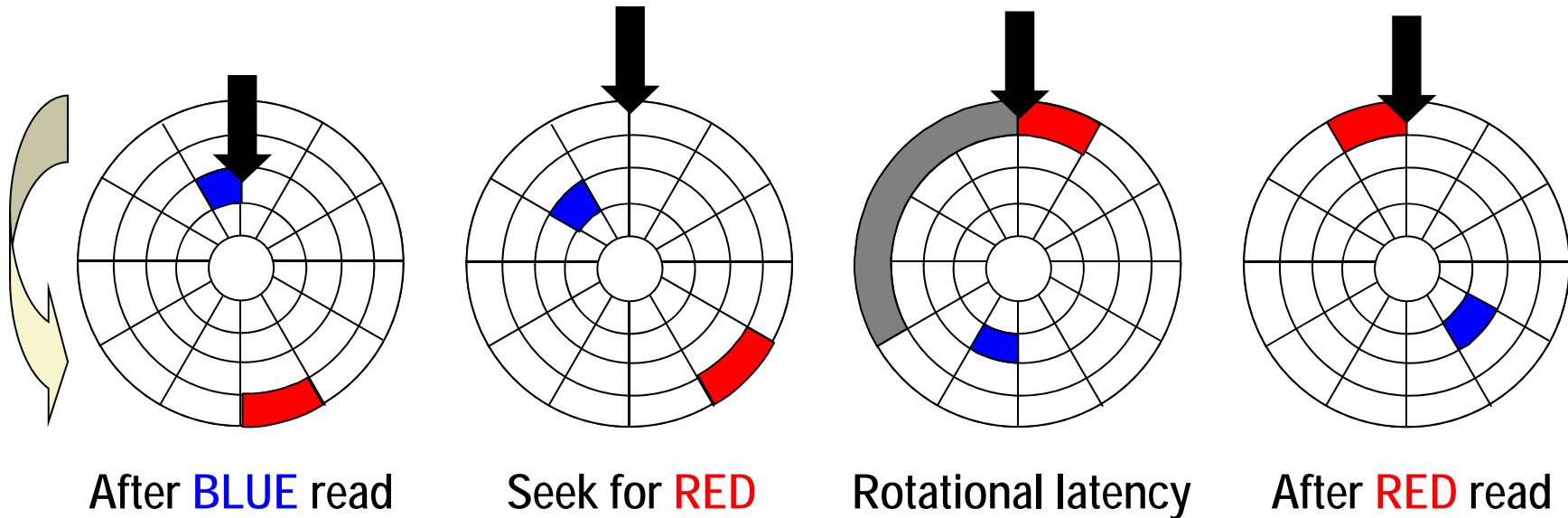
Complete read of red

Disk Access – Service Time Components

After BLUE read   Seek for RED   Rotational latency   After RED read

Data transfer   **Seek**   **Rotational latency**   **Data transfer**
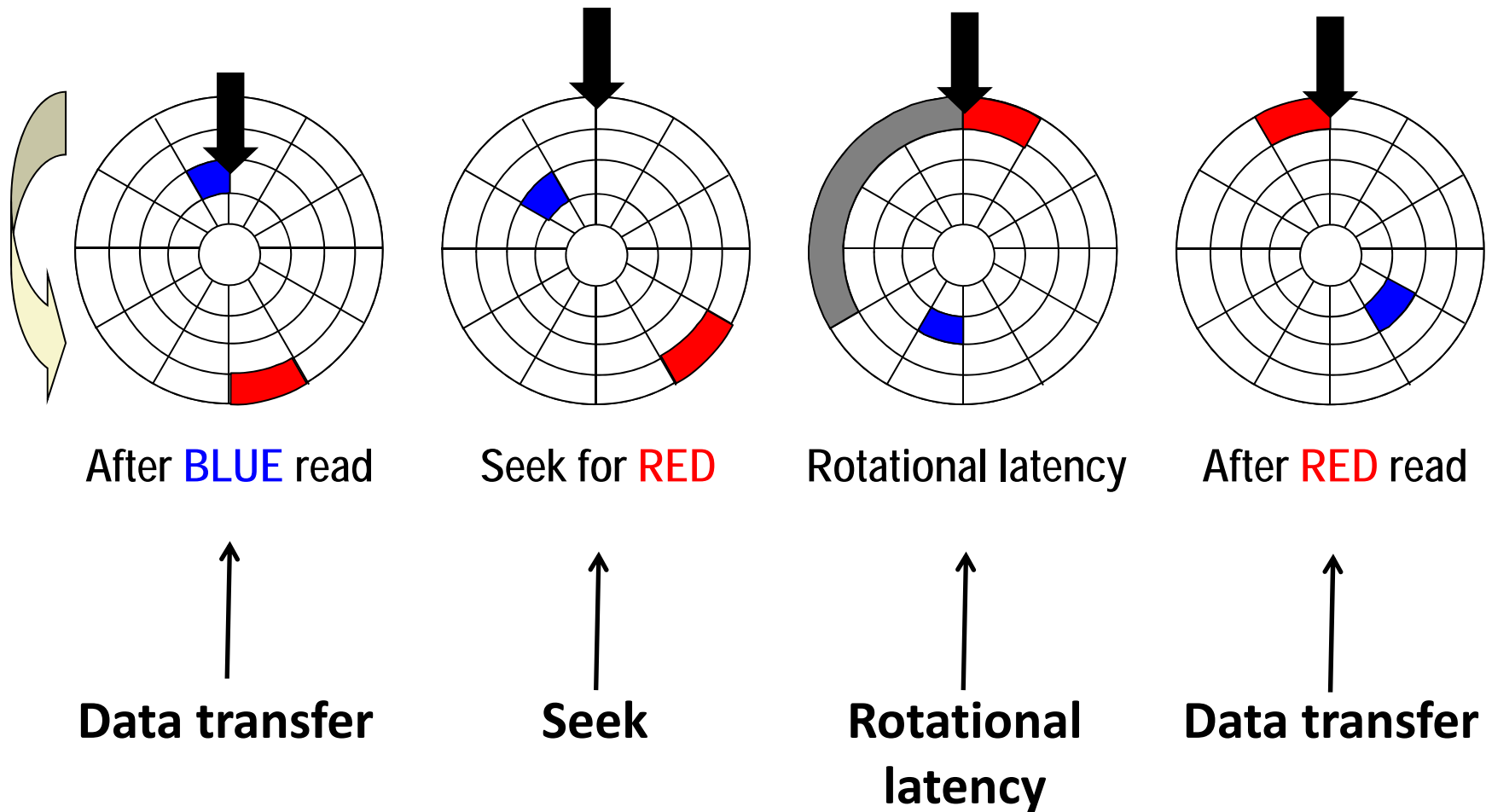
Logical Disk Blocks

- **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, ...)
- **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.
- **Allows controller to set aside spare cylinders for each zone.**
  - Accounts for the difference in "formatted capacity" and "maximum capacity".

I/O Bus

CPU chip

Register file

ALU

System bus   Memory bus

Bus interface

I/O bridge

Main memory

I/O bus

Expansion slots for other devices such as network adapters.

USB controller

Graphics adapter

Disk controller

Mouse   Keyboard

Monitor

Disk

Reading a Disk Sector (1)

CPU chip

Register file

ALU

Bus interface

CPU initiates a disk read by writing a command, logical block number, and destination memory address to a port (address) associated with disk controller.

Main memory

I/O bus

USB controller

Graphics adapter

Disk controller

mouse keyboard

Monitor

Disk

Reading a Disk Sector (2)

CPU chip

Register file

ALU

Bus interface

Disk controller reads the sector and performs a direct memory access (DMA) transfer into main memory.

Main memory

I/O bus

USB controller

Graphics adapter

Disk controller

Mouse   Keyboard   Monitor

Disk

Reading a Disk Sector (3)

CPU chip

Register file

ALU

Bus interface

Main memory

I/O bus

USB controller

Graphics adapter

Disk controller

Mouse   Keyboard

Monitor

Disk

When the DMA transfer completes, the disk controller notifies the CPU with an *interrupt* (i.e., asserts a special "interrupt" pin on the CPU)

Solid State Disks (SSDs)

I/O bus

Requests to read and write logical disk blocks

Solid State Disk (SSD)

Flash translation layer

Flash memory

Block 0			...	Block B-1		
Page 0	Page 1	... Page P-1		Page 0	Page 1	... Page P-1
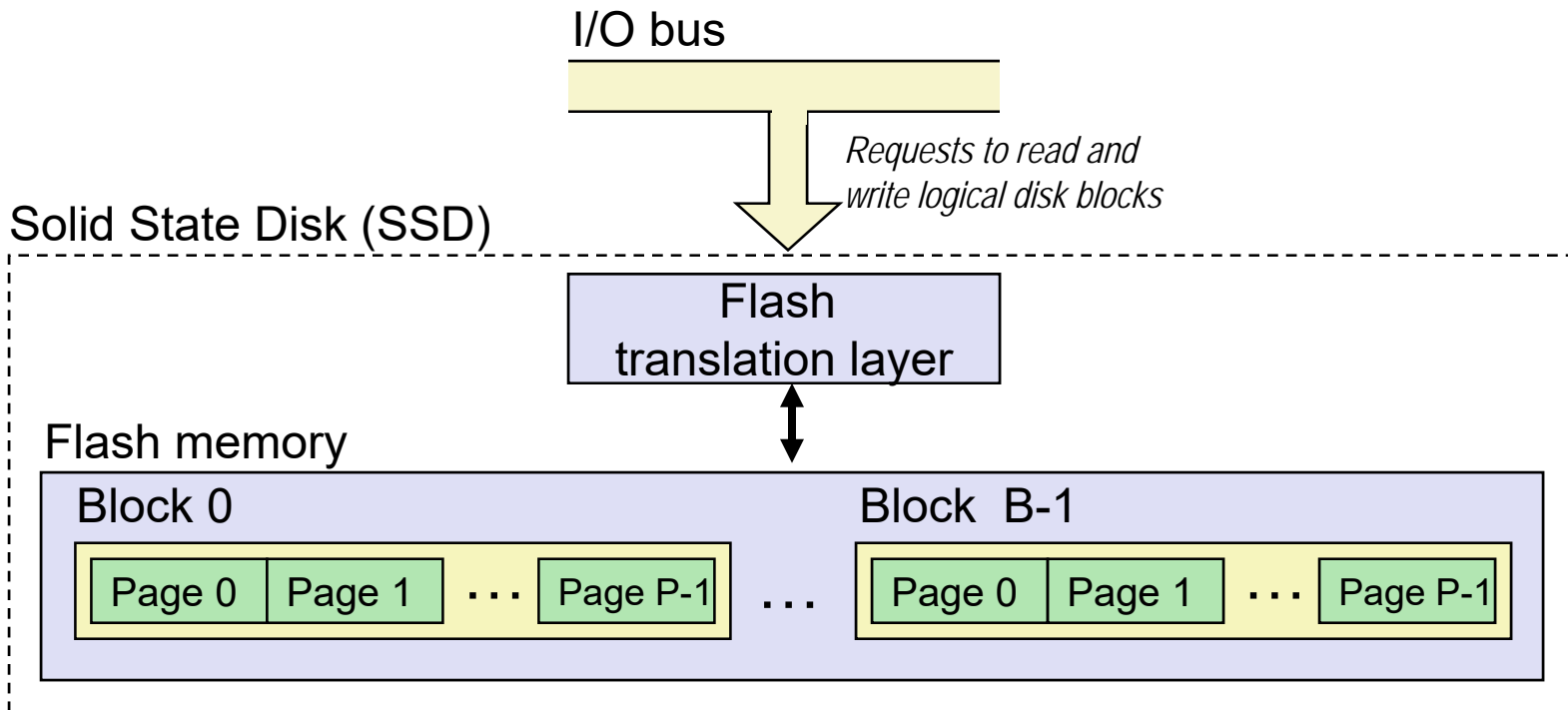
- **Pages: 512KB to 4KB, Blocks: 32 to 128 pages**

- **Data read/written in units of pages.**

- **Page can be written only after its block has been erased**

- **A block wears out after about 100,000 repeated writes.**

SSD Performance Characteristics

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**
  - Common theme in the memory hierarchy
- **Random writes are somewhat slower**
  - Erasing a block takes a long time (~1 ms)
  - Modifying a block page requires all other pages to be copied to new block
  - In earlier SSDs, the read/write gap was much larger.

Source: Intel SSD 730 product specification.

SSD Tradeoffs vs Rotating Disks

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

- **Disadvantages**
  - Have the potential to wear out
    - Mitigated by "wear leveling logic" in flash translation layer
    - E.g. Intel SSD 730 guarantees 128 petabyte (128 x 10^{15} bytes) of writes before they wear out
  - In 2015, about 30 times more expensive per byte

- **Applications**
  - MP3 players, smart phones, laptops
  - Beginning to appear in desktops and servers

The CPU-Memory Gap

The gap widens between DRAM, disk, and CPU speeds.
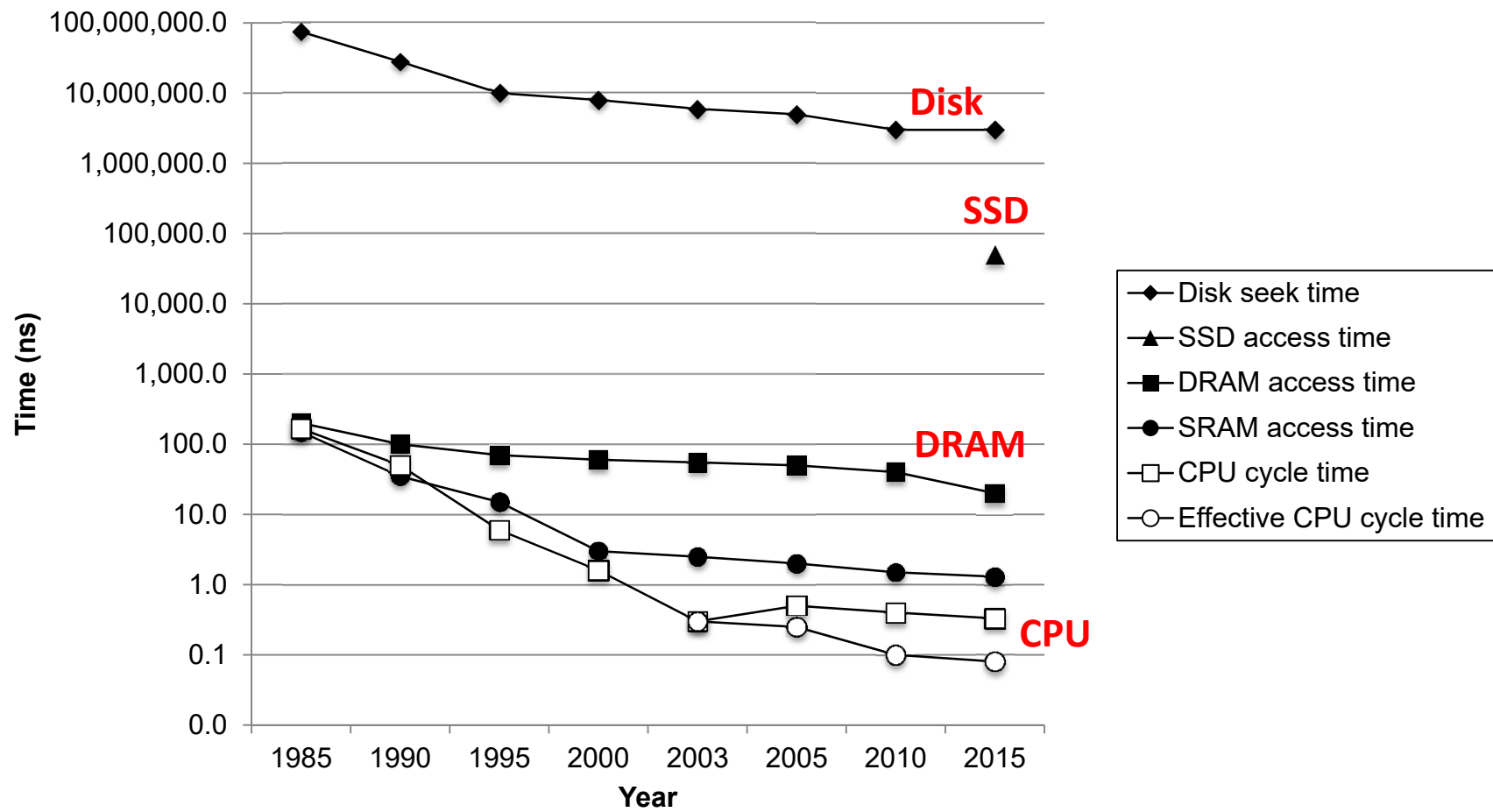
Locality to the Rescue!

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 tend to 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
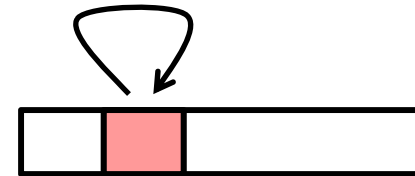
- **Spatial locality:**

  - Items with nearby addresses tend
    to be referenced close together in time
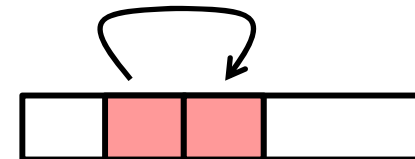
Locality Example

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

- **Data references**
  - Reference array elements in succession (stride-1 reference pattern). **Spatial locality**
  - Reference variable `sum` each iteration. **Temporal locality**

- **Instruction references**
  - Reference instructions in sequence. **Spatial locality**
  - Cycle through loop repeatedly. **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;
}
```

Locality Example

Question: Does this function have good locality with respect to array a?

```
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;
}
```

Locality Example

⟳ **Question**: Can you permute the loops so that the function scans the 3-d array a with a stride-1 reference pattern (and thus has good spatial locality)?

```
int sum_array_3d(int a[M][N][N])
{
    int i, j, k, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
                sum += a[k][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.**

Today

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

Example Memory Hierarchy

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

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

L0: Regs

L1: L1 cache (SRAM)

L2: L2 cache (SRAM)

L3: L3 cache (SRAM)

L4: Main memory (DRAM)

L5: Local secondary storage (local disks)

L6: Remote secondary storage (e.g., Web servers)

CPU registers hold words retrieved from the L1 cache.

L1 cache holds cache lines retrieved from the L2 cache.

L2 cache holds cache lines retrieved from L3 cache

L3 cache holds cache lines retrieved from main memory.

Main memory holds disk blocks retrieved from local disks.

Local disks hold files retrieved from disks on remote servers

Notes ad
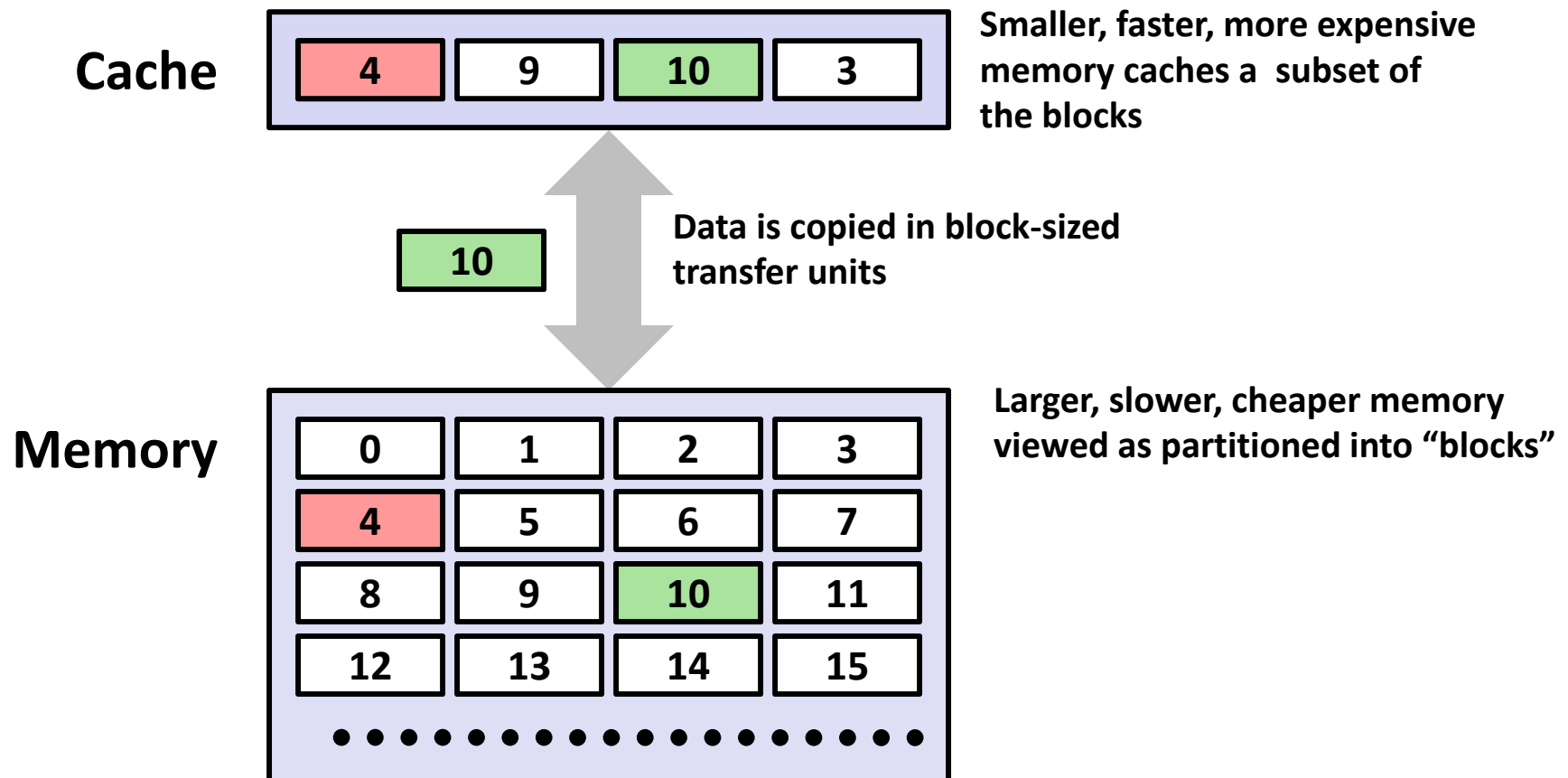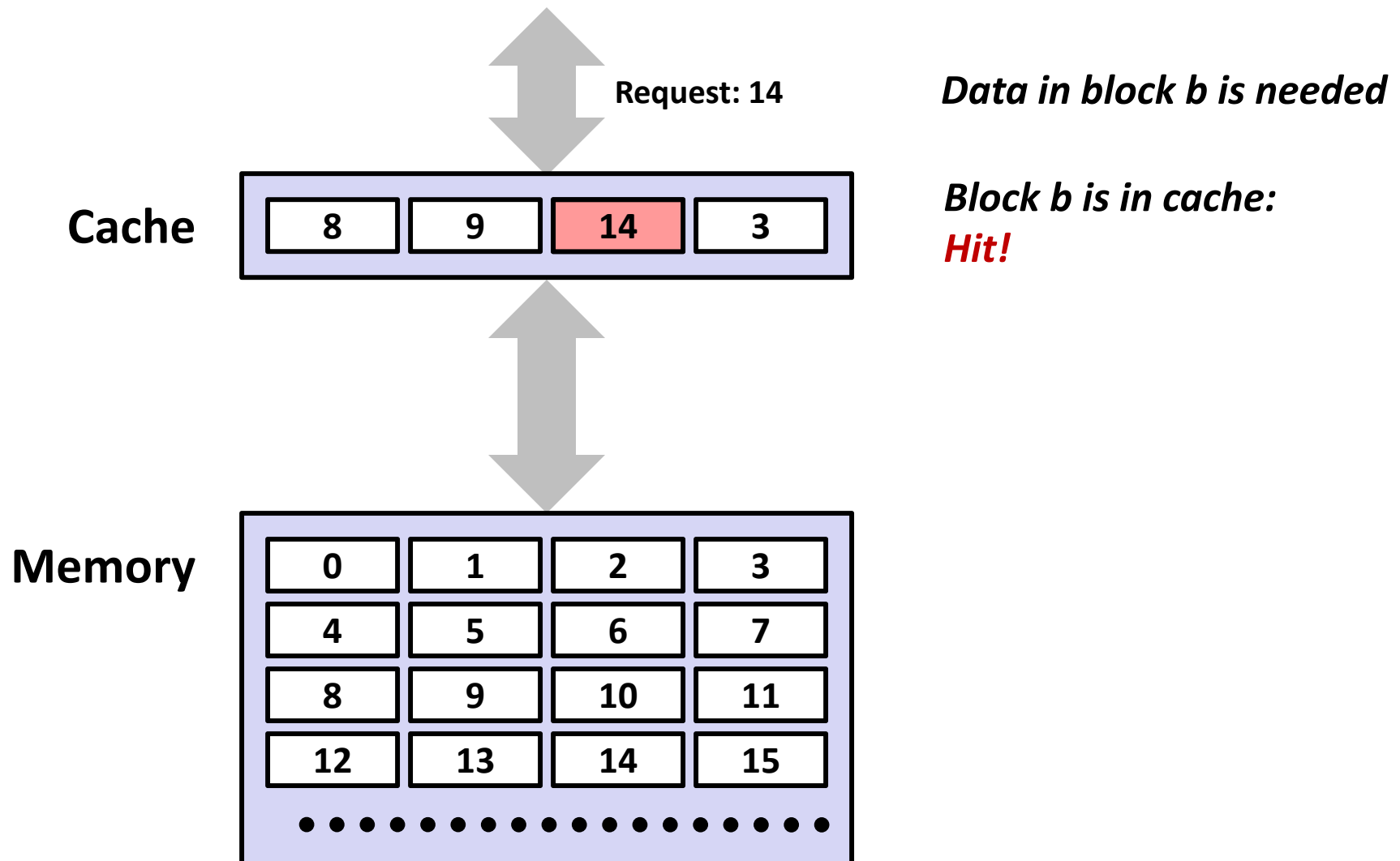
Caches

- *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

Cache

4	9	10	3

Smaller, faster, more expensive memory caches a subset of the blocks

10

Data is copied in block-sized transfer units

Memory

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Larger, slower, cheaper memory viewed as partitioned into "blocks"

General Cache Concepts: Hit

Request: 14

Data in block b is needed

Block b is in cache:
Hit!

Cache

| 8 | 9 | 14 | 3 |

Memory

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

● ● ● ● ● ● ● ● ● ● ● ● ● ● ● ● ●

General Cache Concepts: Miss

Request: 12

Data in block b is needed

Cache

8	12	14	3

Block b is not in cache:
Miss!

12   **Request: 12**

Block b is fetched from memory

Memory

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Block b is stored in cache
- Placement policy:
  determines where b goes
- Replacement policy:
  determines which block
  gets evicted (victim)

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

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

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**
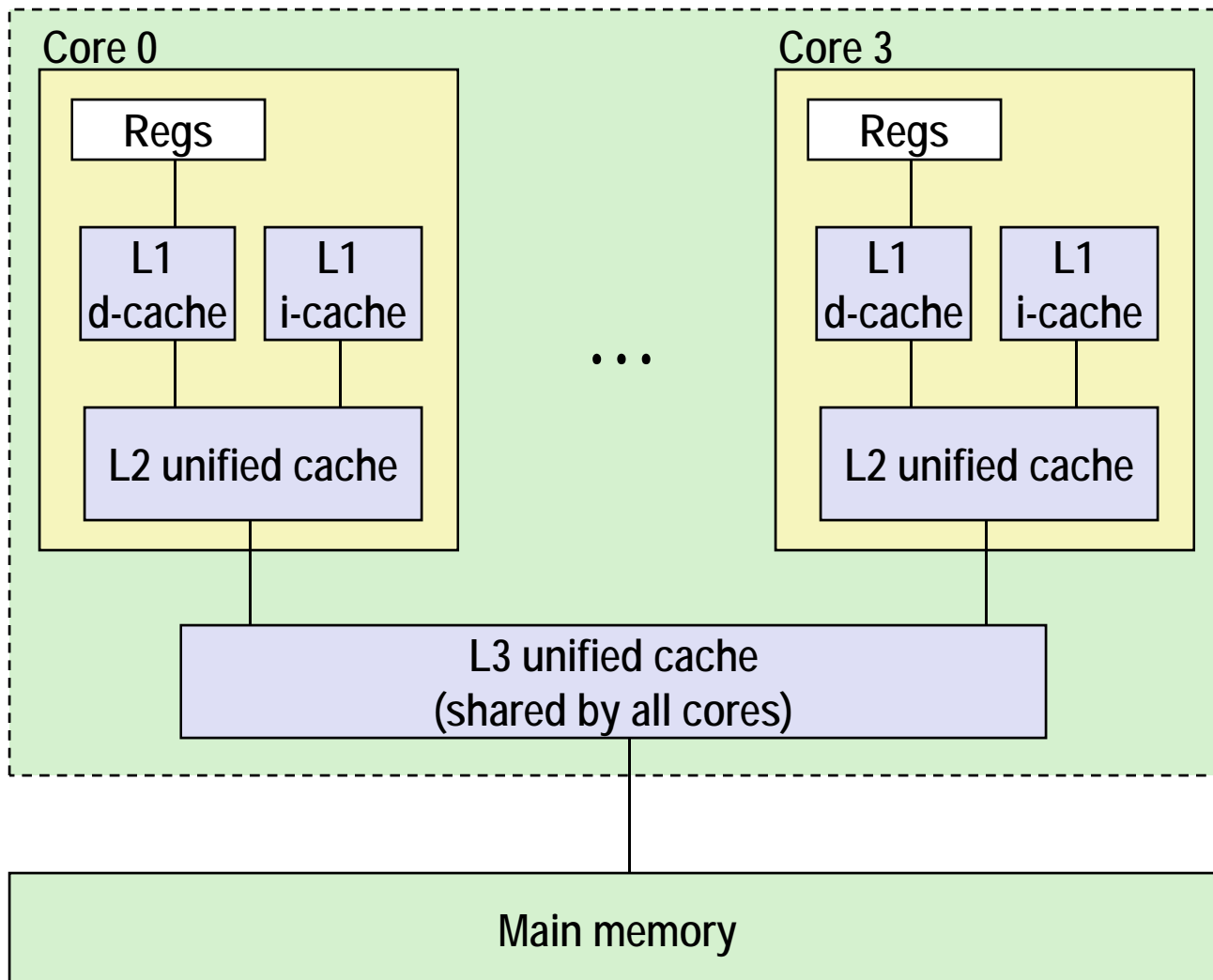- **Typical system structure:**

CPU chip

Register file

Cache memory

ALU

System bus

Memory bus

Bus interface

I/O bridge

Main memory
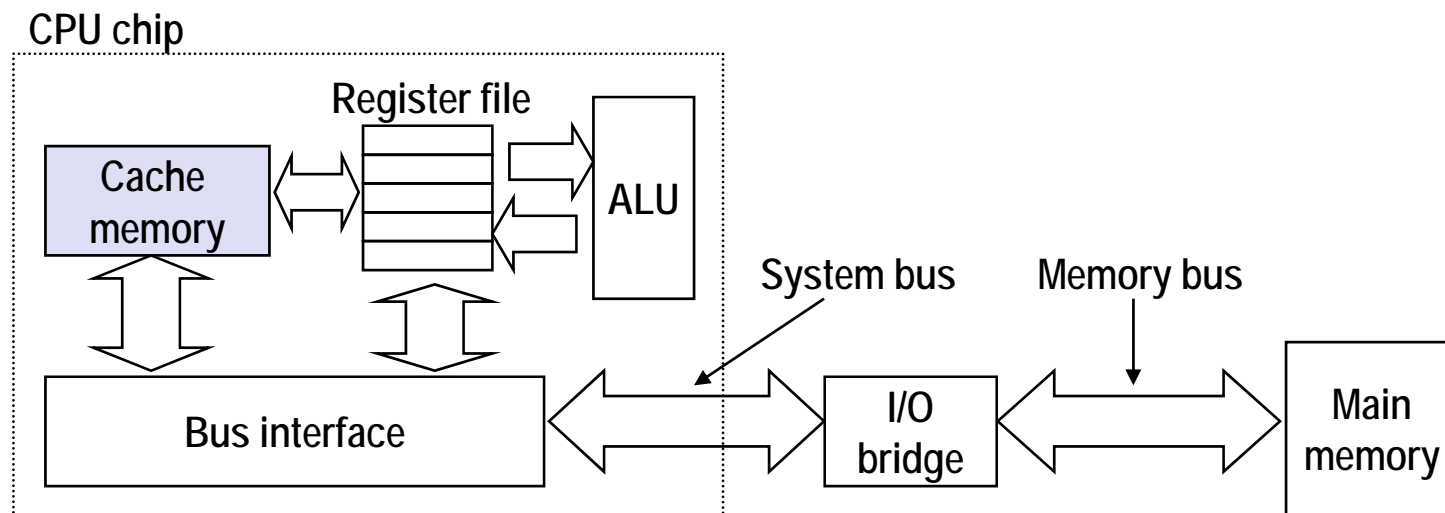
General Cache Organization (S, E, B)

E = 2^e lines per set

set

line

S = 2^s sets

Cache size:
C = S x E x B data bytes

| v | tag | 0 | 1 | 2 | ······ | B-1 |

valid bit

B = 2^b bytes per cache block (the data)
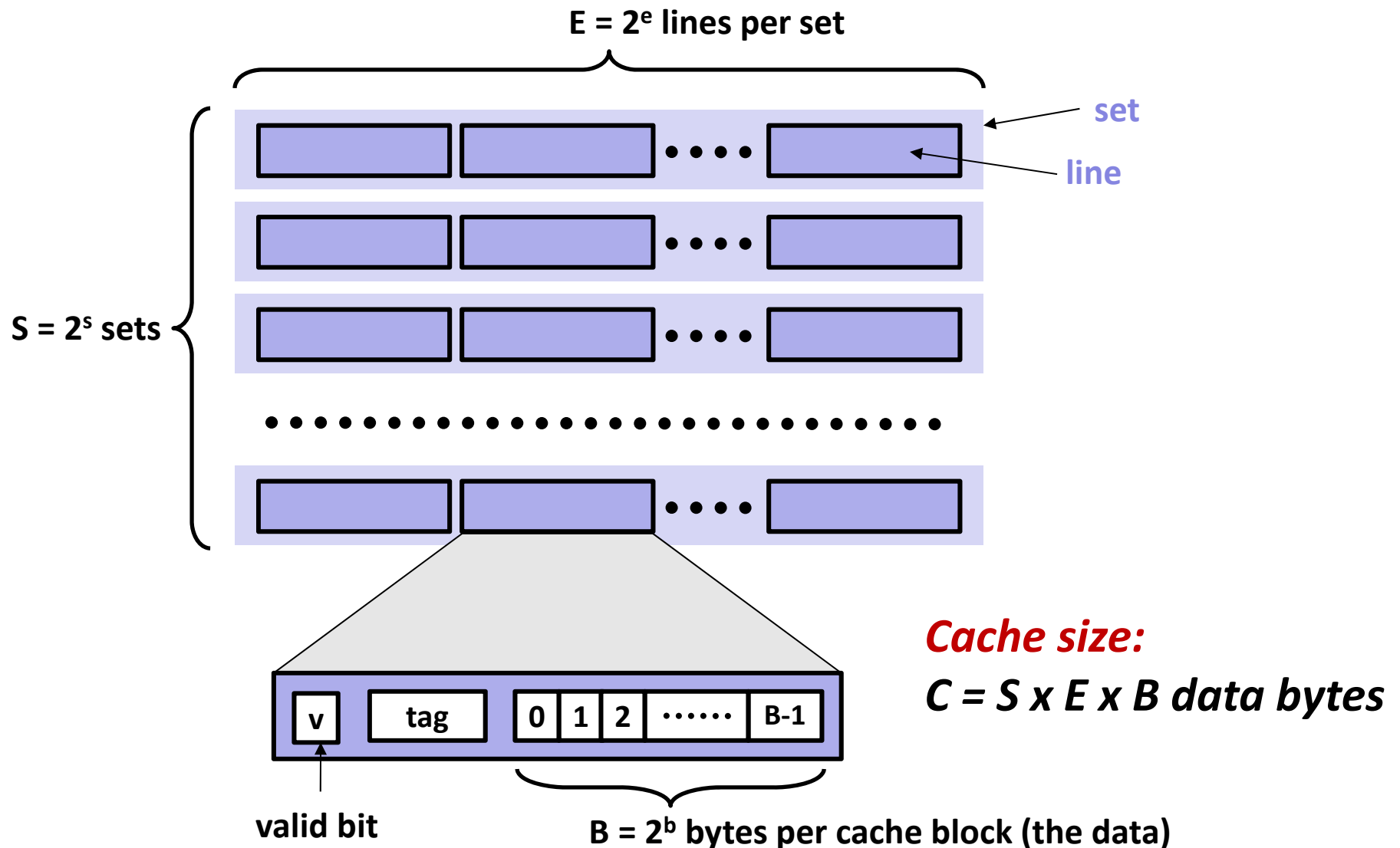
Intel Core i7 Cache Hierarchy

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 – 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 cycle + 0.03 * 100 cycles = **4 cycles**
    99% hits: 1 cycle + 0.01 * 100 cycles = **2 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

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));
}
```
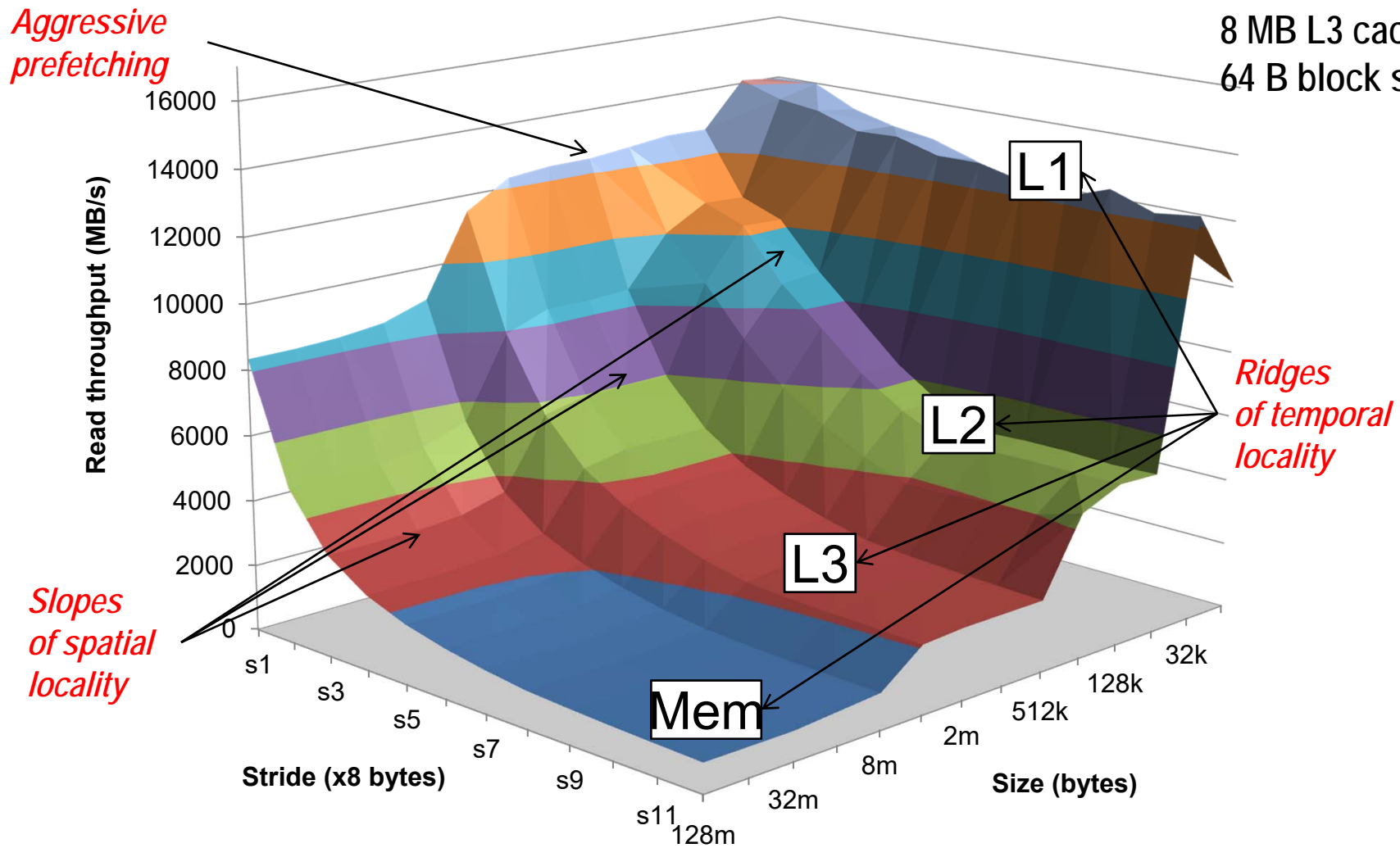*mountain/mountain.c*

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

Aggressive prefetching

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

L1

L2

L3

Mem

Ridges of temporal locality

Slopes of spatial locality

Read throughput (MB/s)

16000
14000
12000
10000
8000
6000
4000
2000
0

Stride (x8 bytes)

s1 s3 s5 s7 s9 s11

Size (bytes)

32k 128k 512k 2m 8m 32m 128m

Matrix Multiplication Example

Description:

- Multiply N x 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

Variable sum held 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;
  }
}                        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
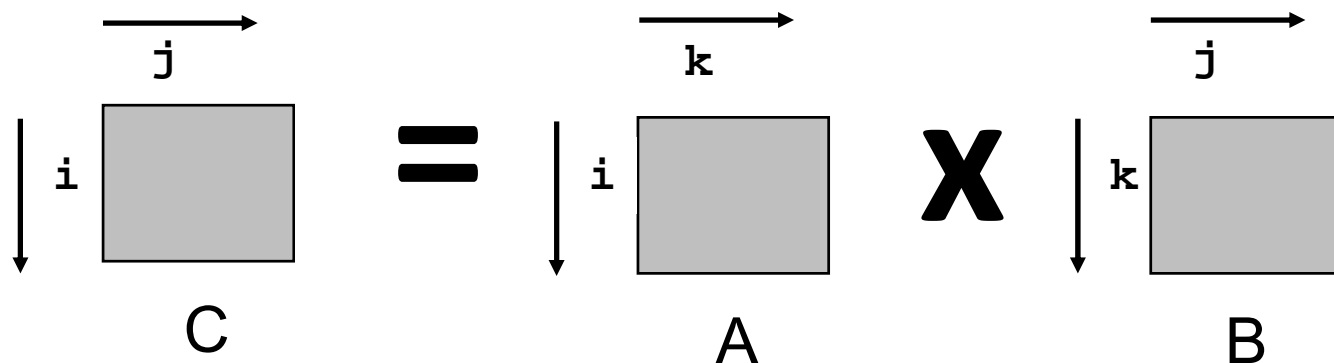
Layout of C Arrays in Memory (review)

- **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_{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
```
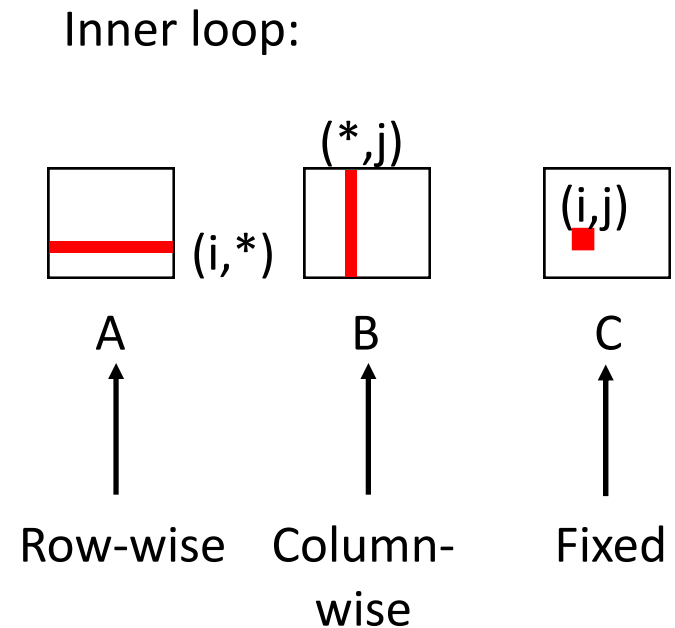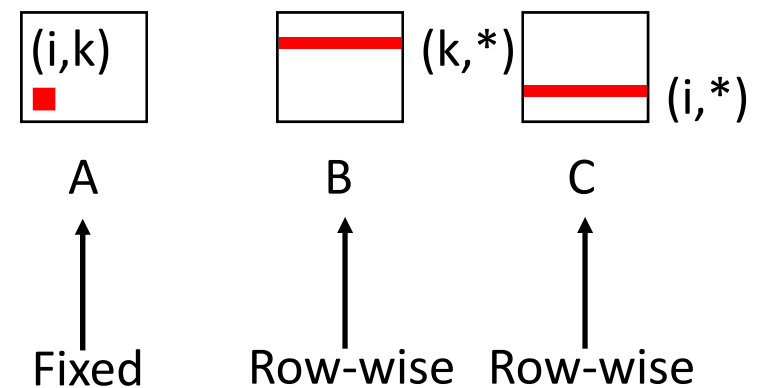
Inner loop:

(*,j)

(i,*)

(i,j)

A   B   C

Row-wise   Column-   Fixed
           wise

Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0
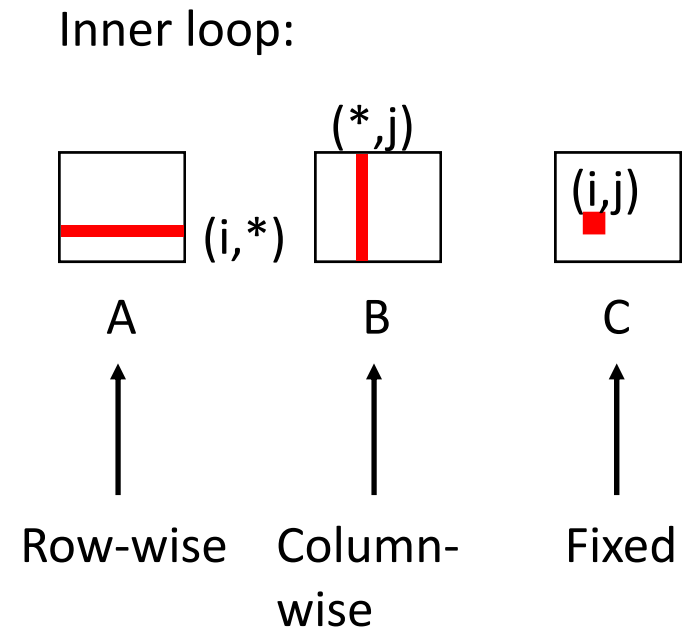
Matrix Multiplication (jik)

```
/* 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:

(*,j)

(i,*)

(i,j)

A   B   C

Row-wise   Column-   Fixed
           wise

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
```
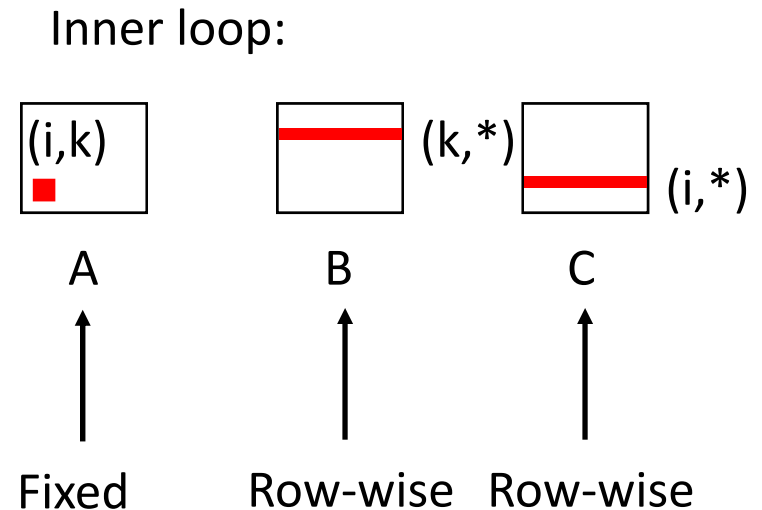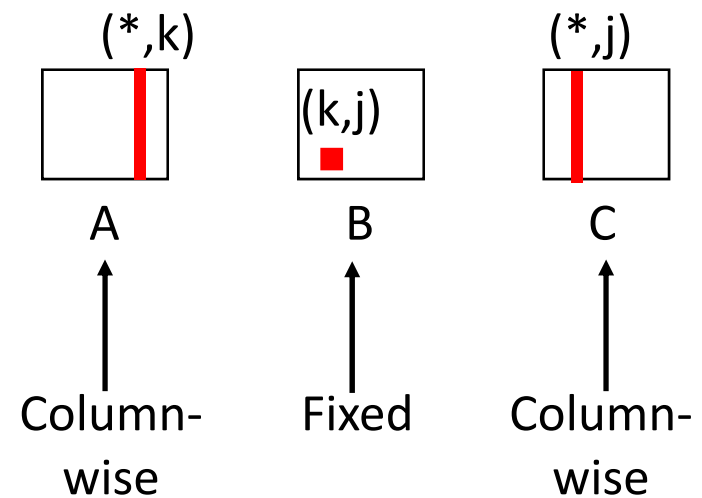
Inner loop:

A — Fixed
B — Row-wise
C — Row-wise

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
```

Inner loop:

(i,k)	(k,*)	(i,*)
A	B	C
↑	↑	↑
Fixed	Row-wise	Row-wise

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
```
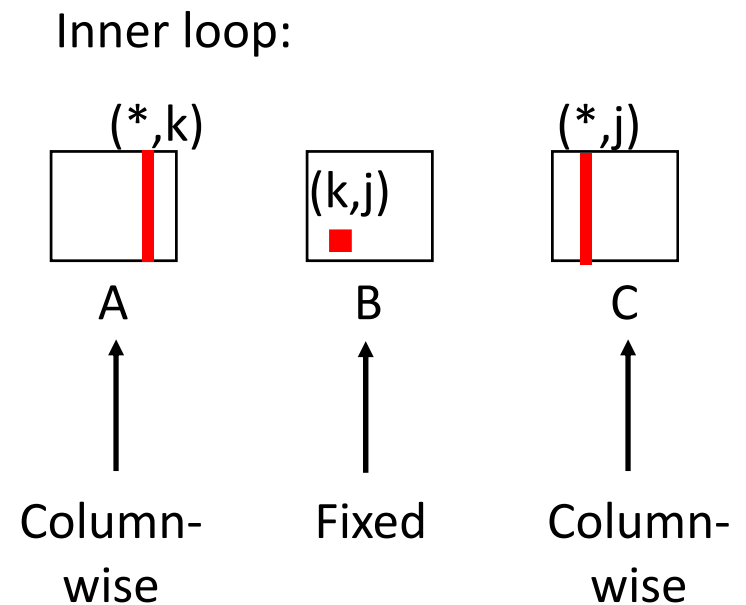
Inner loop:

```
        (*,k)                              (*,j)
                        (k,j)

         A             B              C

      Column-         Fixed        Column-
       wise                         wise
```

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
```

Inner loop:

(*,k)	(k,j)	(*,j)
A	B	C
Column-wise	Fixed	Column-wise

Misses per inner loop iteration:

A	B	C
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];
  }
 }
```
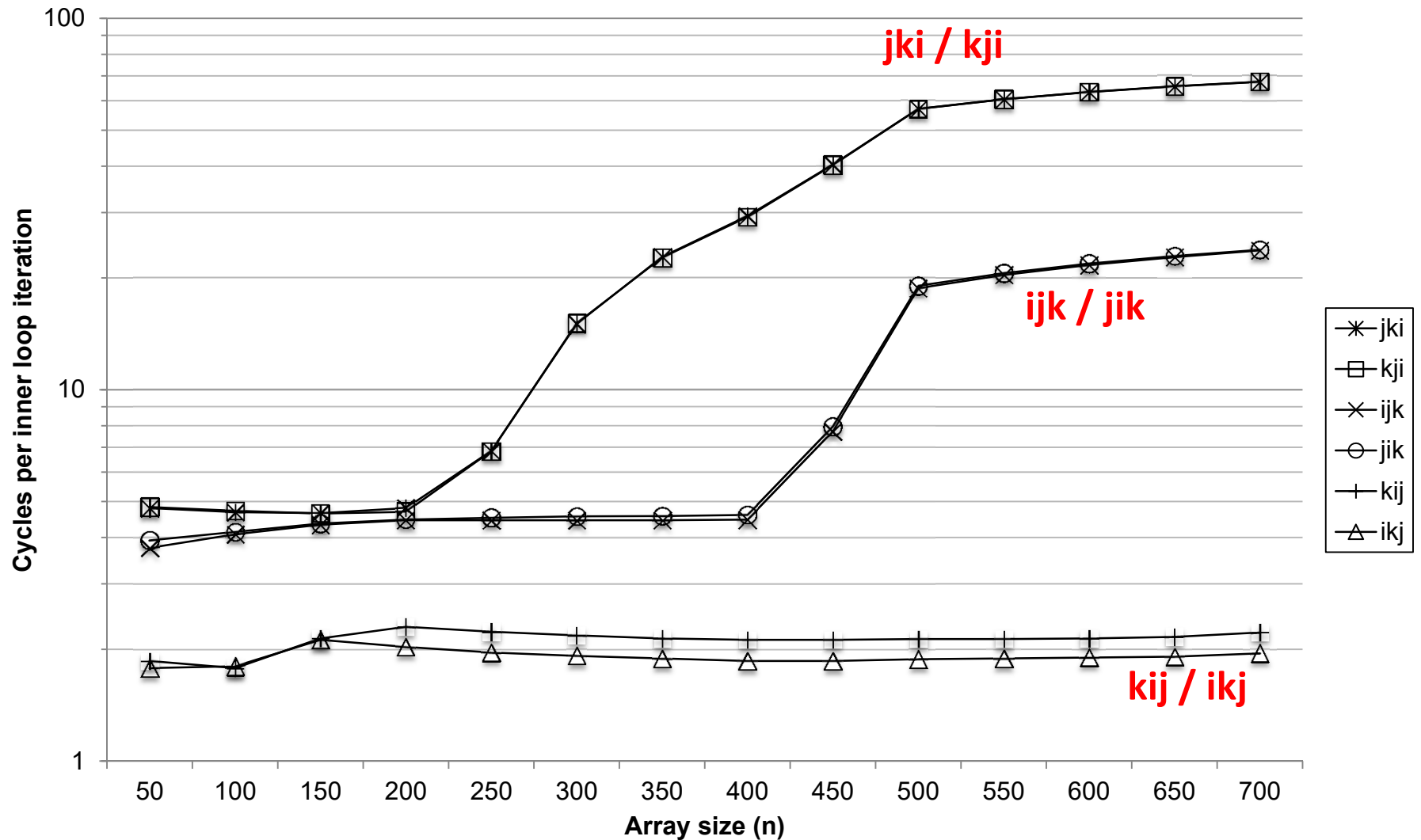
kij (& ikj):
- 2 loads, 1 store
- misses/iter = **0.5**

```
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

jki / kji

ijk / jik

kij / ikj

- jki
- kji
- ijk
- jik
- kij
- ikj

Cycles per inner loop iteration

100

10

1

50 100 150 200 250 300 350 400 450 500 550 600 650 700

Array size (n)

Example: Matrix Multiplication

```c
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];
}
```
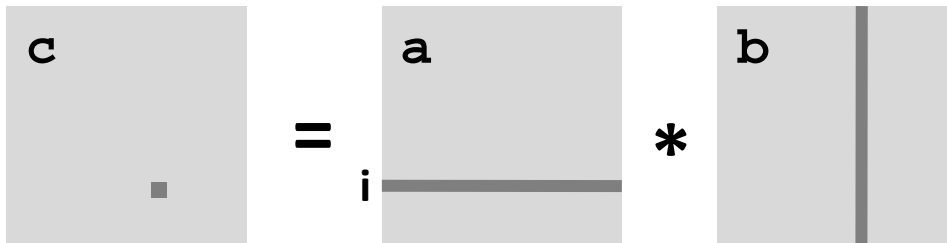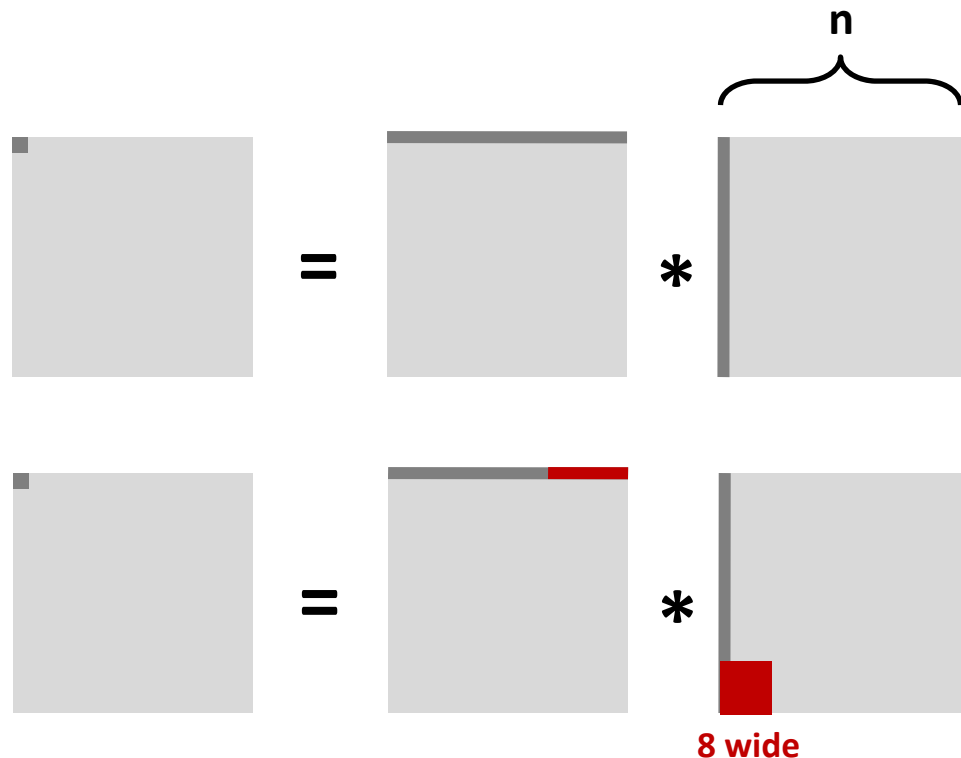
c   =   a   *   b

i

Cache Miss Analysis

Assume:

- Matrix elements are doubles
- Cache block = 8 doubles
- Cache size C << n (much smaller than n)

First iteration:

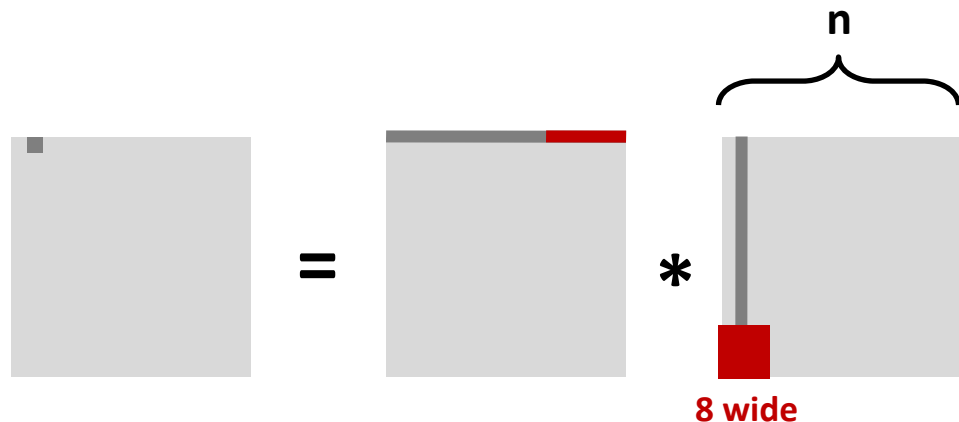- n/8 + n = 9n/8 misses

- Afterwards in cache: (schematic)

$$= \quad * $$

8 wide

Cache Miss Analysis

Assume:

- Matrix elements are doubles
- Cache block = 8 doubles
- Cache size C << n (much smaller than n)

Second iteration:

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

$$n$$
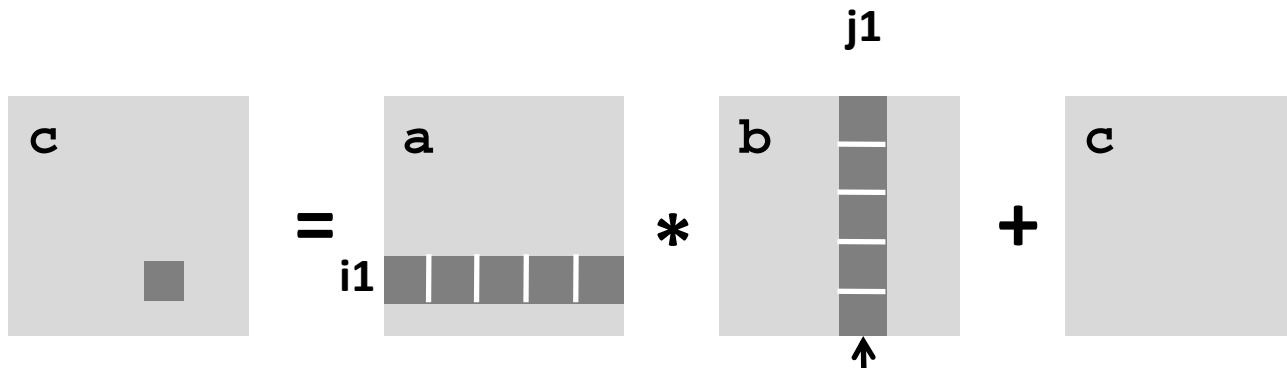
$$= \quad * \quad$$

8 wide

Total misses:

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

Blocked Matrix Multiplication

```c
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];
}
```
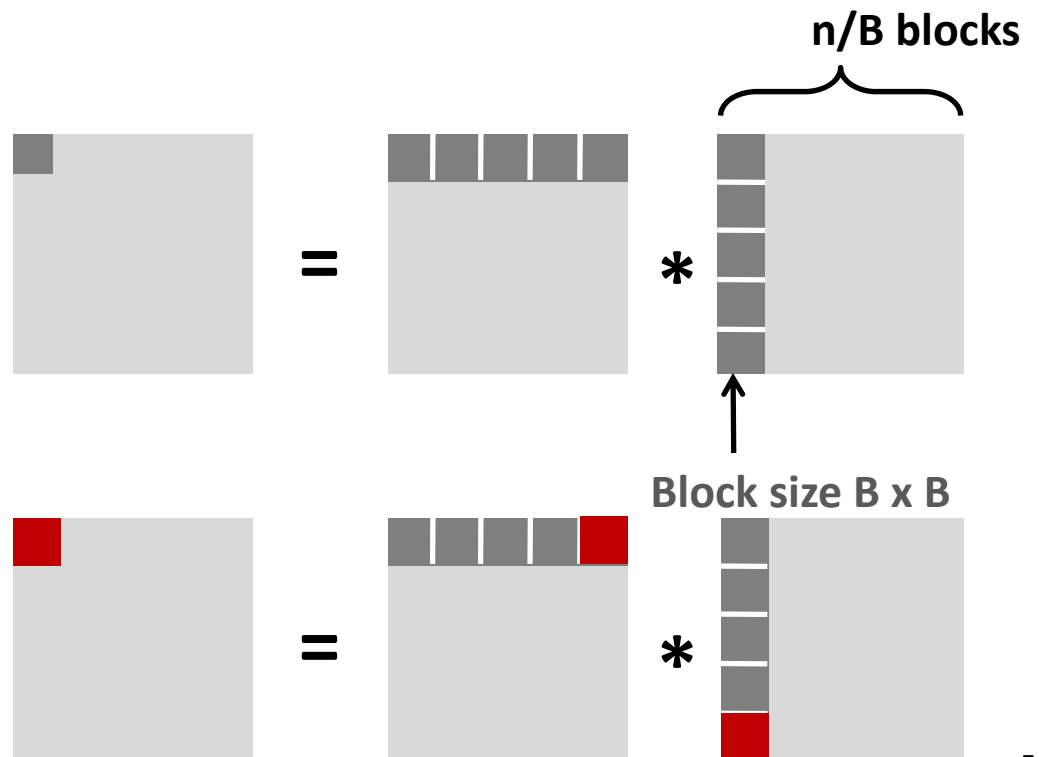*matmult/bmm.c*

Block size B x B

Cache Miss Analysis

- **Assume:**
  - Cache block = 8 doubles
  - Cache size C << 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)
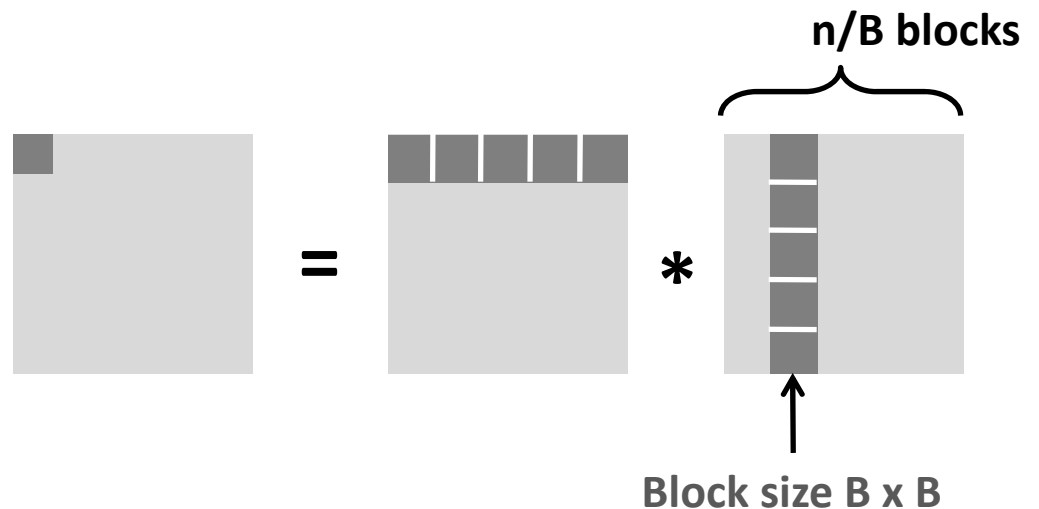
n/B blocks

Block size B x B

Cache Miss Analysis

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

- **Second (block) iteration:**
  - Same as first iteration
  - $2n/B * B^2/8 = nB/4$

n/B blocks

= * **Block size B x B**

- **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

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.**

  - 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.