

Cache Memories

COMP400727: Introduction to Computer Systems

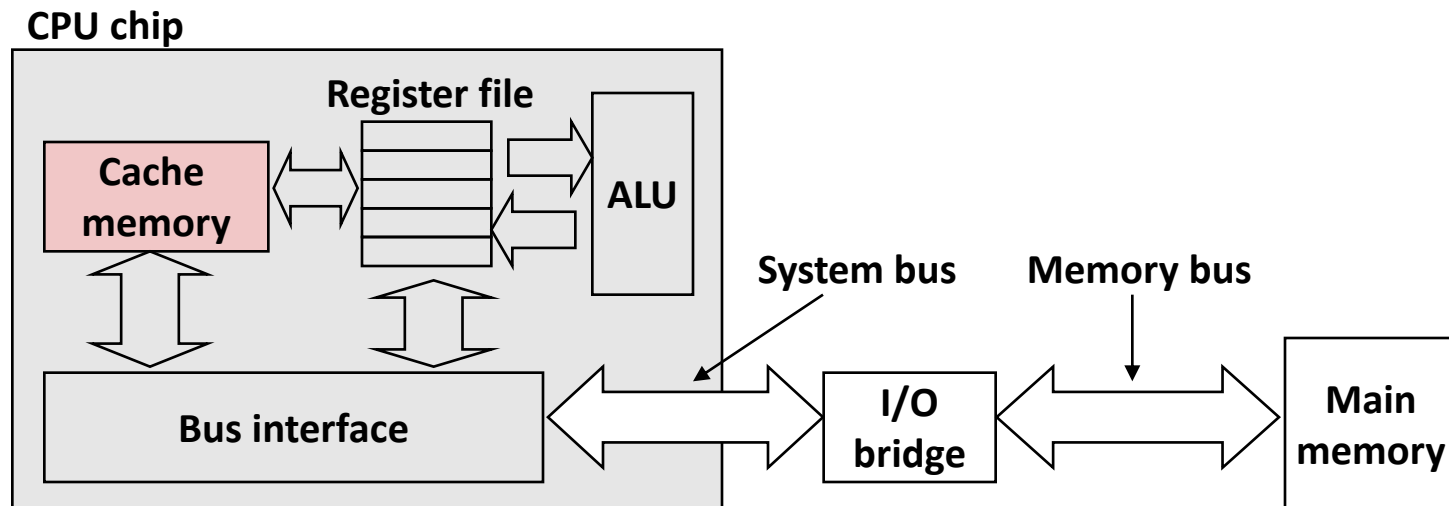
Danfeng Shan
Xi'an Jiaotong University

Today

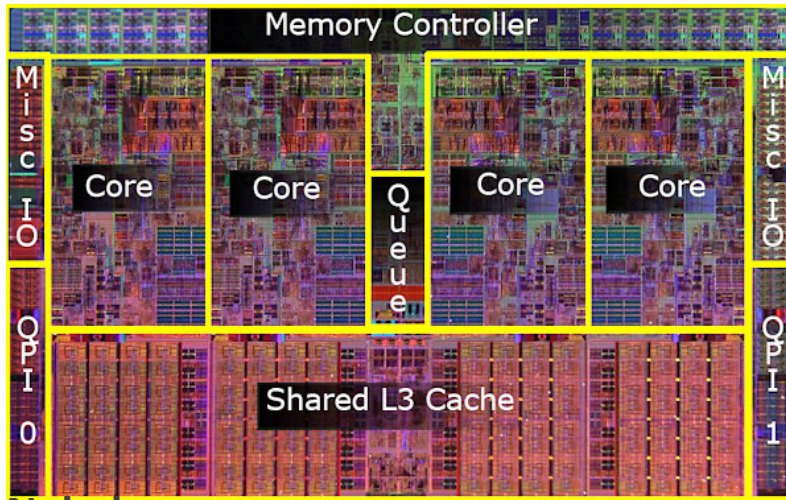
- **Cache memory organization and operation**
- **Performance impact of caches**
 - The memory mountain
 - Rearranging loops to improve spatial locality
 - Using blocking to improve temporal locality

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:

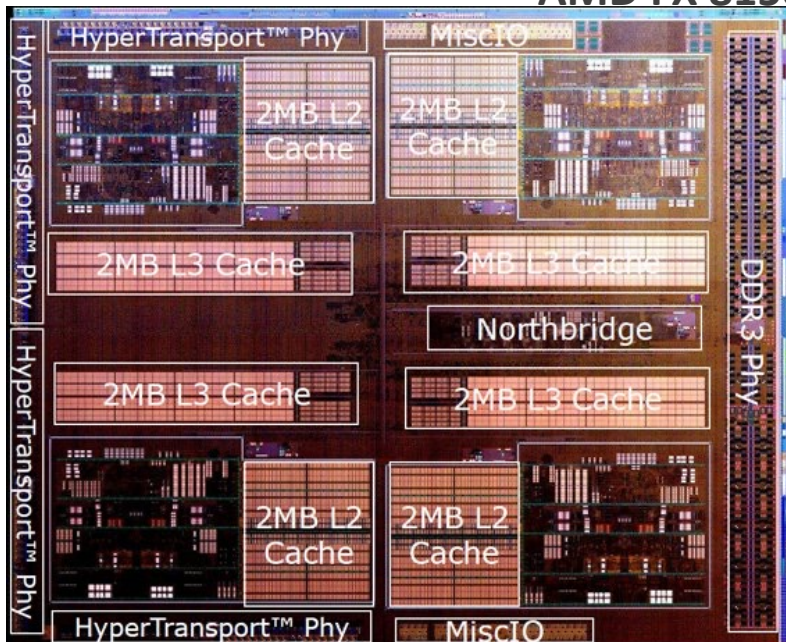


What it Really Looks Like

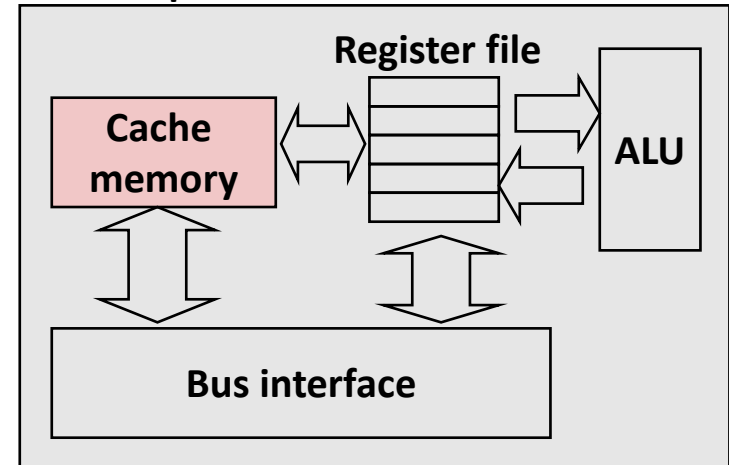


Nehalem

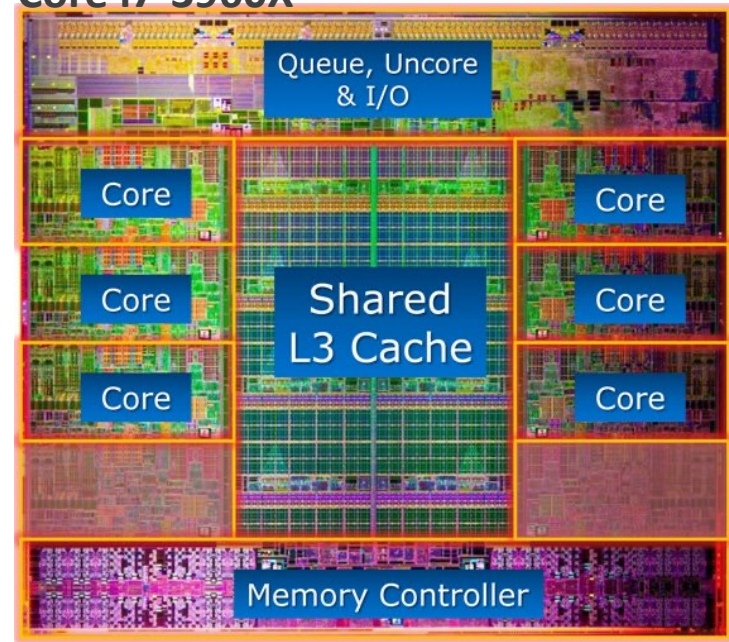
AMD FX 8150



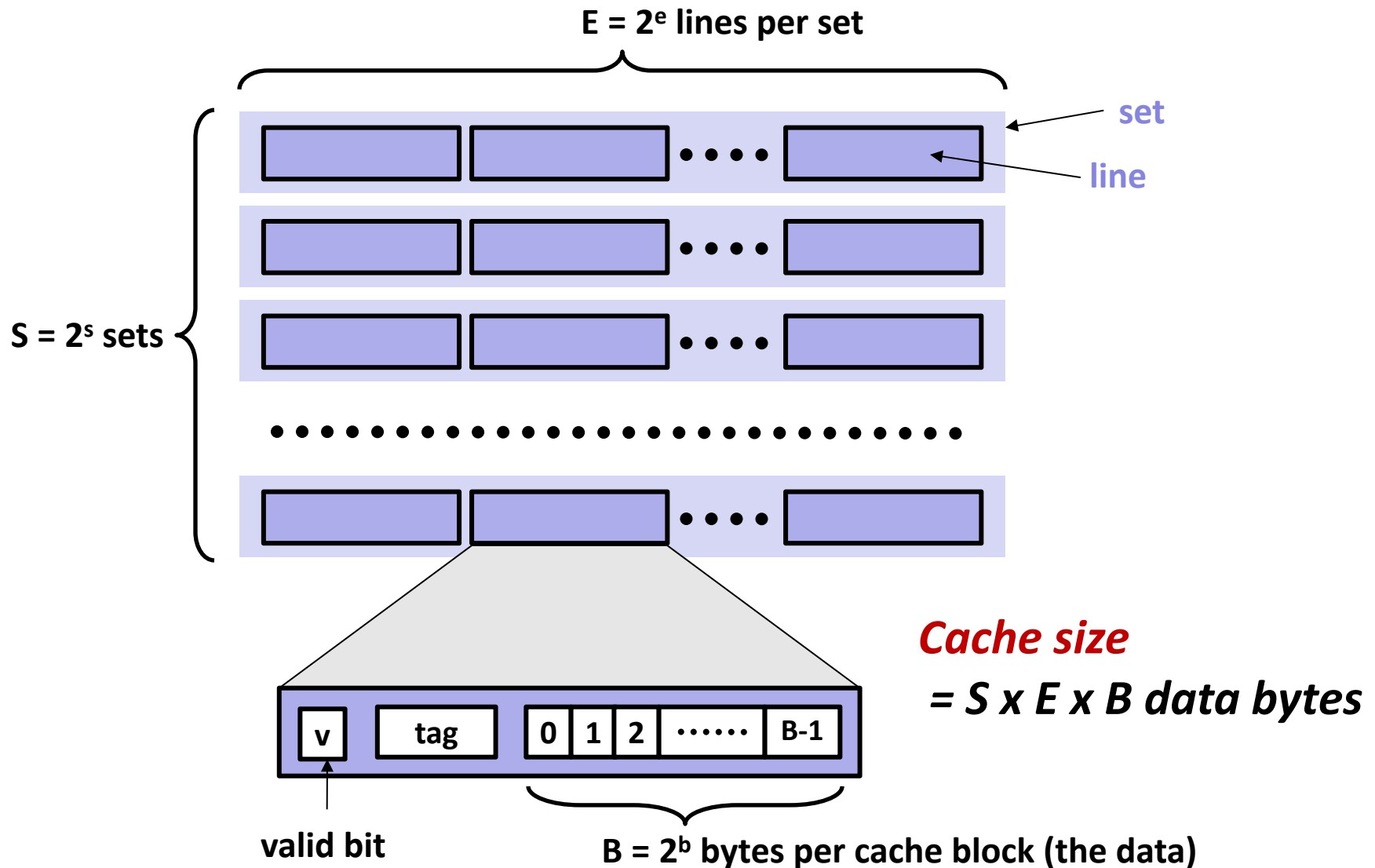
CPU chip



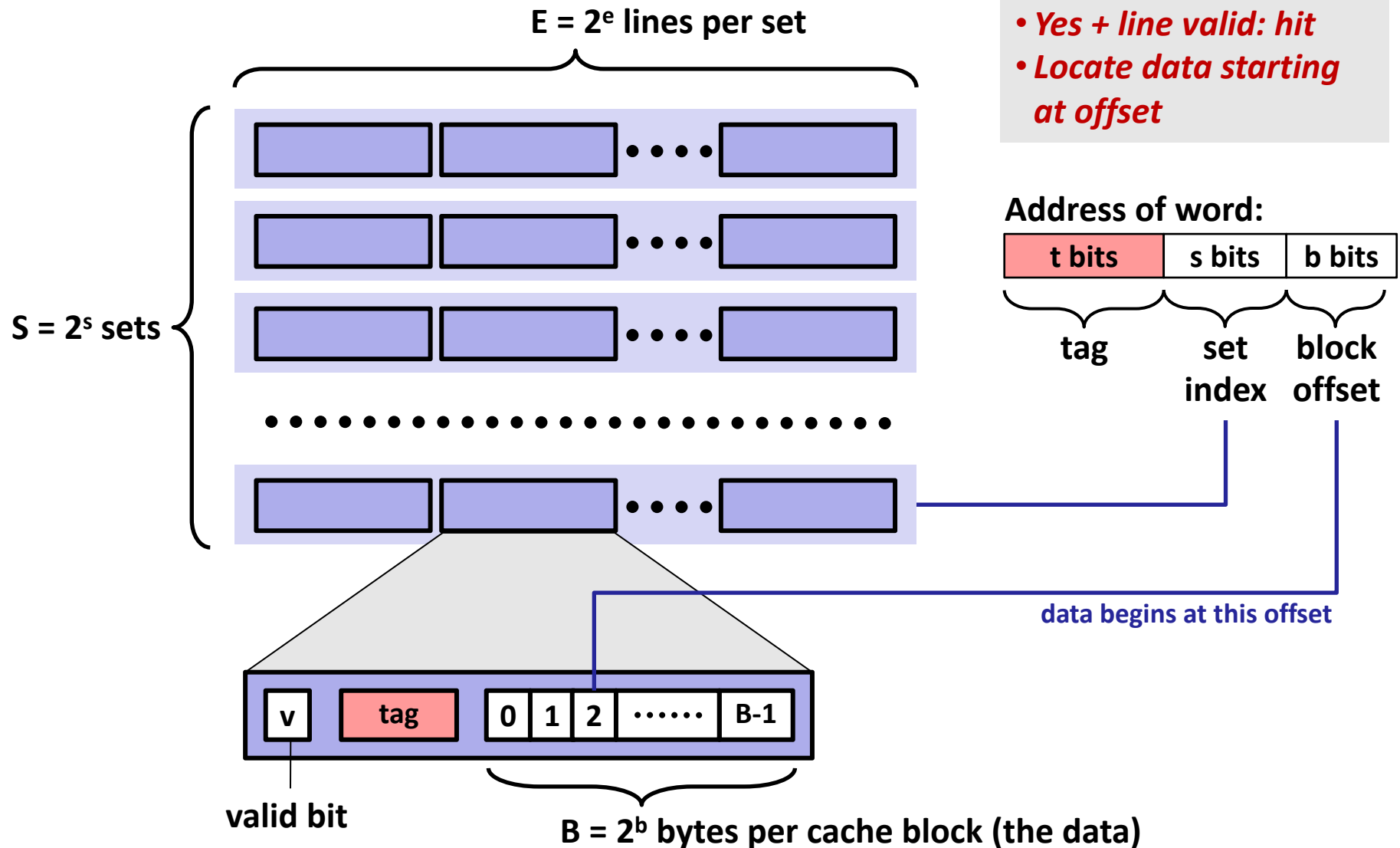
Core i7-3960X



General Cache Organization (S, E, B)



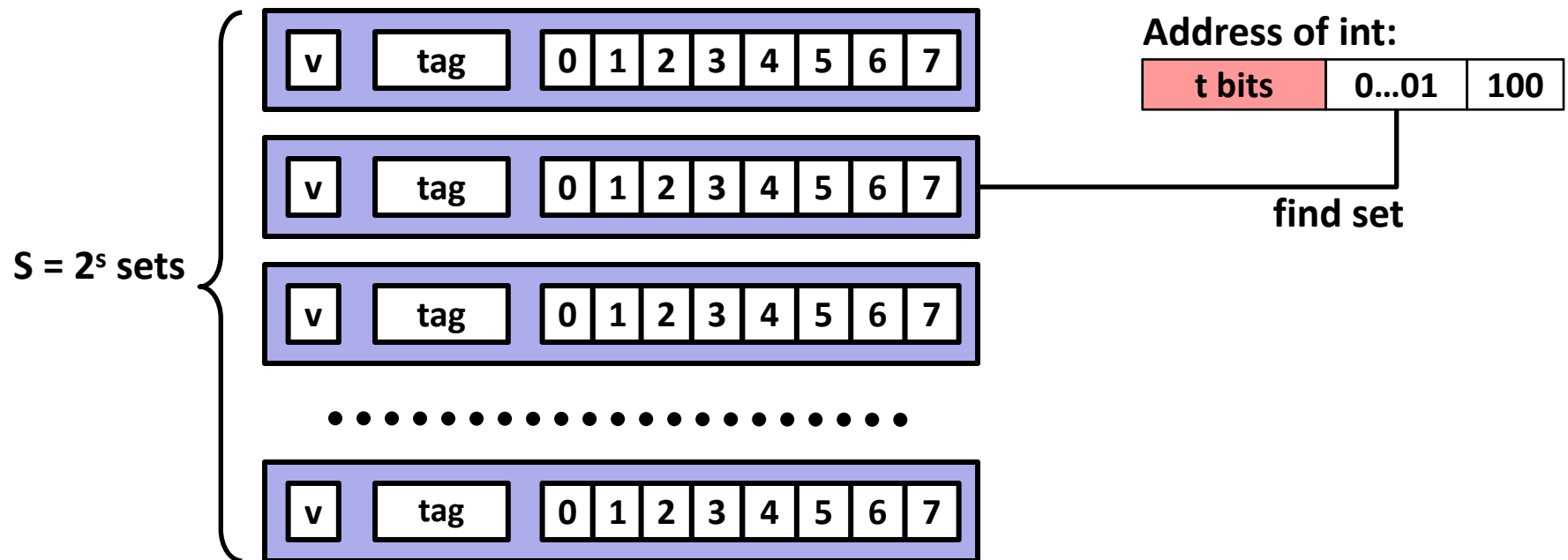
Cache Read



Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set

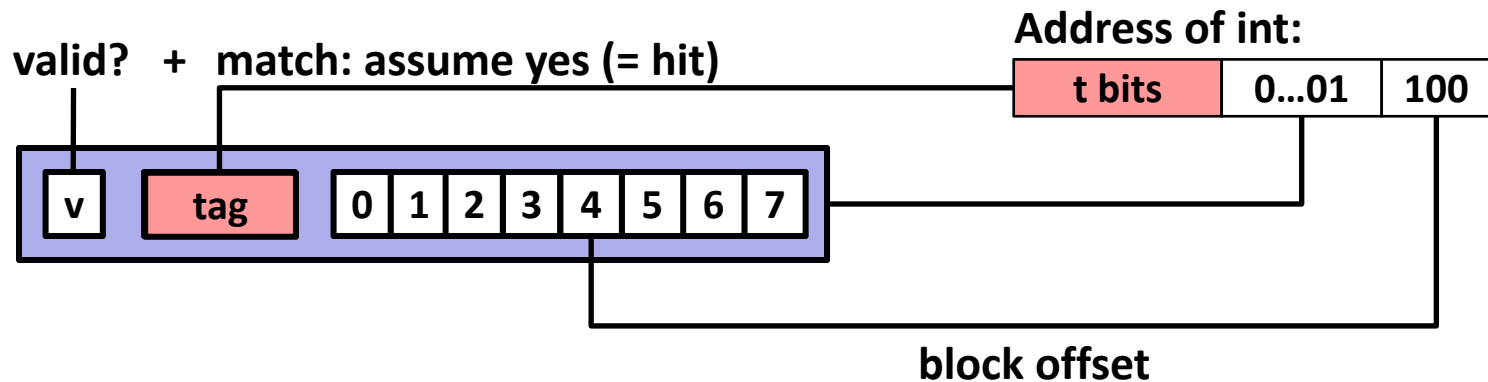
Assume: cache block size B=8 bytes



Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set

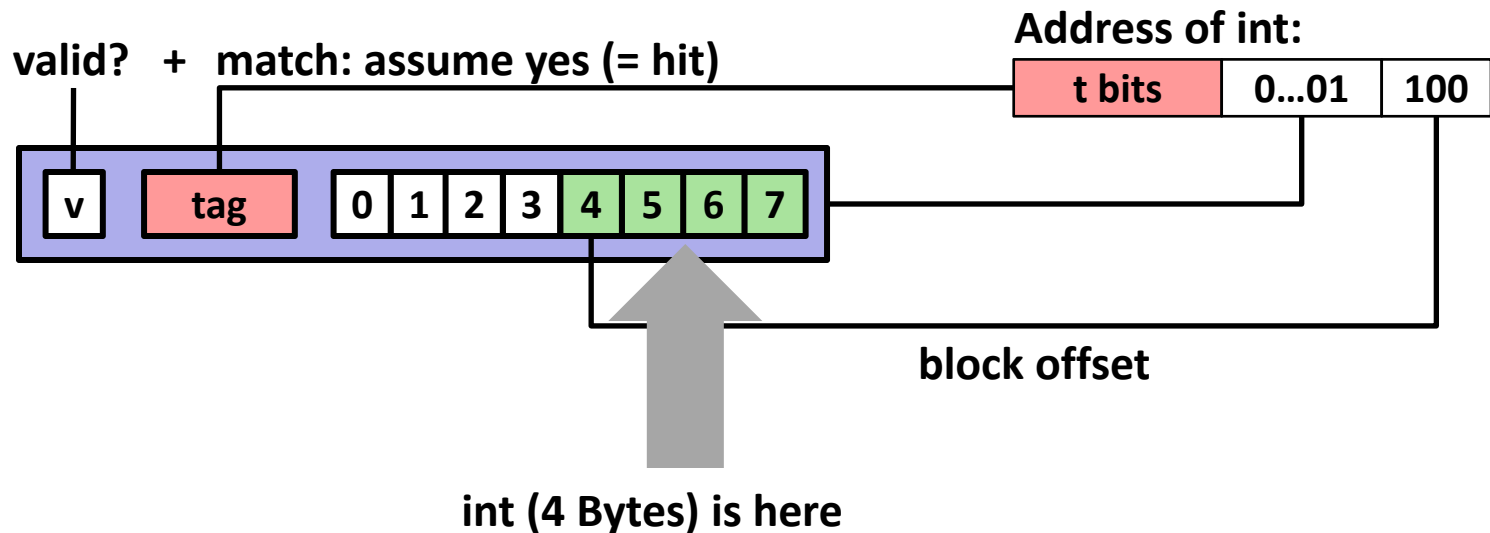
Assume: cache block size B=8 bytes



Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set

Assume: cache block size B=8 bytes



If tag doesn't match (= miss): old line is evicted and replaced

Direct-Mapped Cache Simulation

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

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

Address trace (reads, one byte per read):

0	[<u>0</u> 00 <u>0</u> ₂],	miss
1	[<u>0</u> 00 <u>1</u> ₂],	hit
7	[<u>0</u> 1 <u>1</u> 1 ₂],	miss
8	[<u>1</u> 00 <u>0</u> ₂],	miss
0	[<u>0</u> 00 <u>0</u> ₂]	miss

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

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

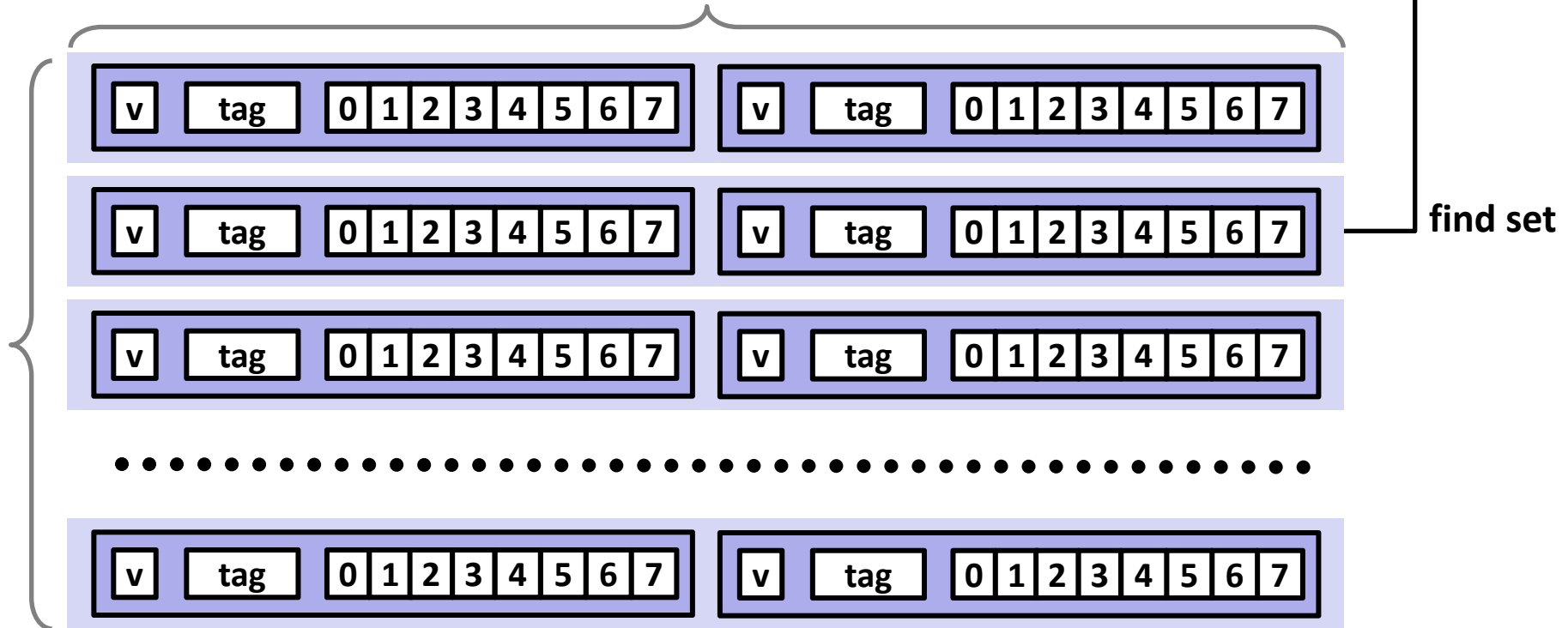
E = 2: Two lines per set

Assume: cache block size B=8 bytes

2 lines per set

Address of short int:

t bits	0...01	100
--------	--------	-----

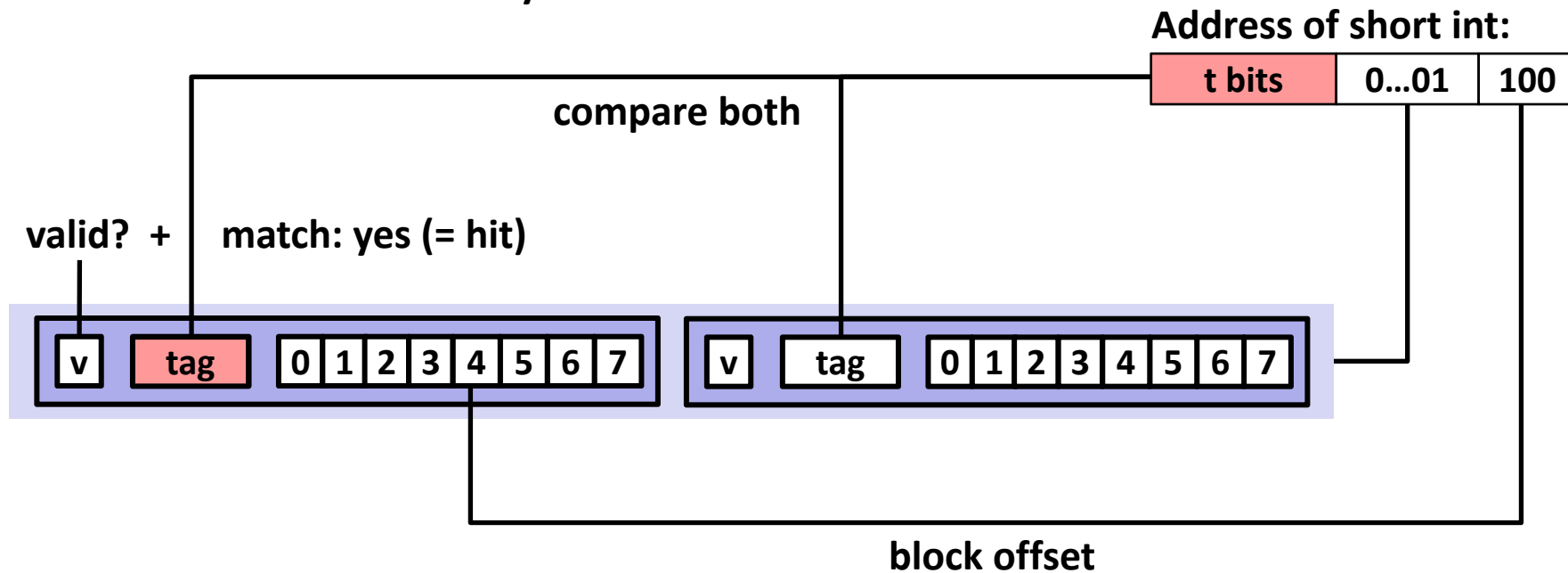


S sets

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

E = 2: Two lines per set

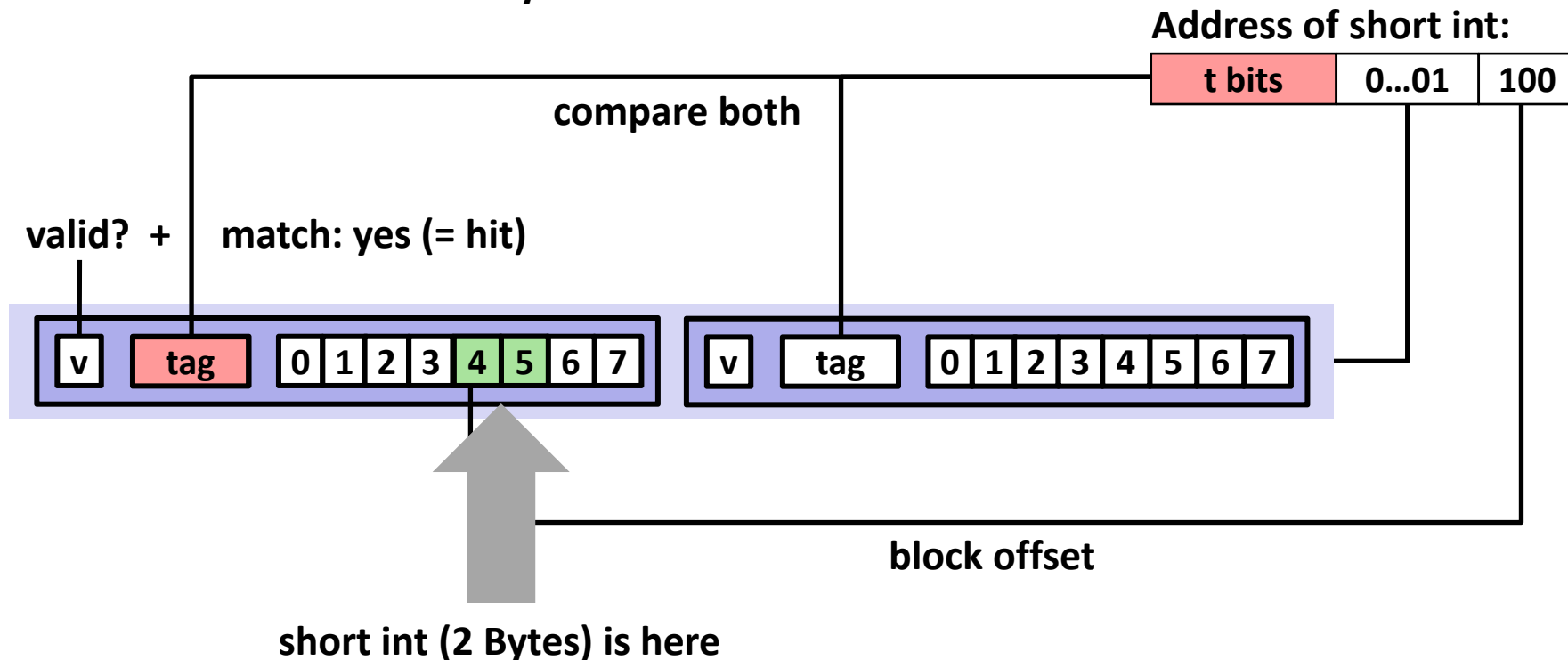
Assume: cache block size B=8 bytes



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

E = 2: Two lines per set

Assume: cache block size B=8 bytes



No match or not valid (= miss):

- 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

4-bit addresses (M=16 bytes)

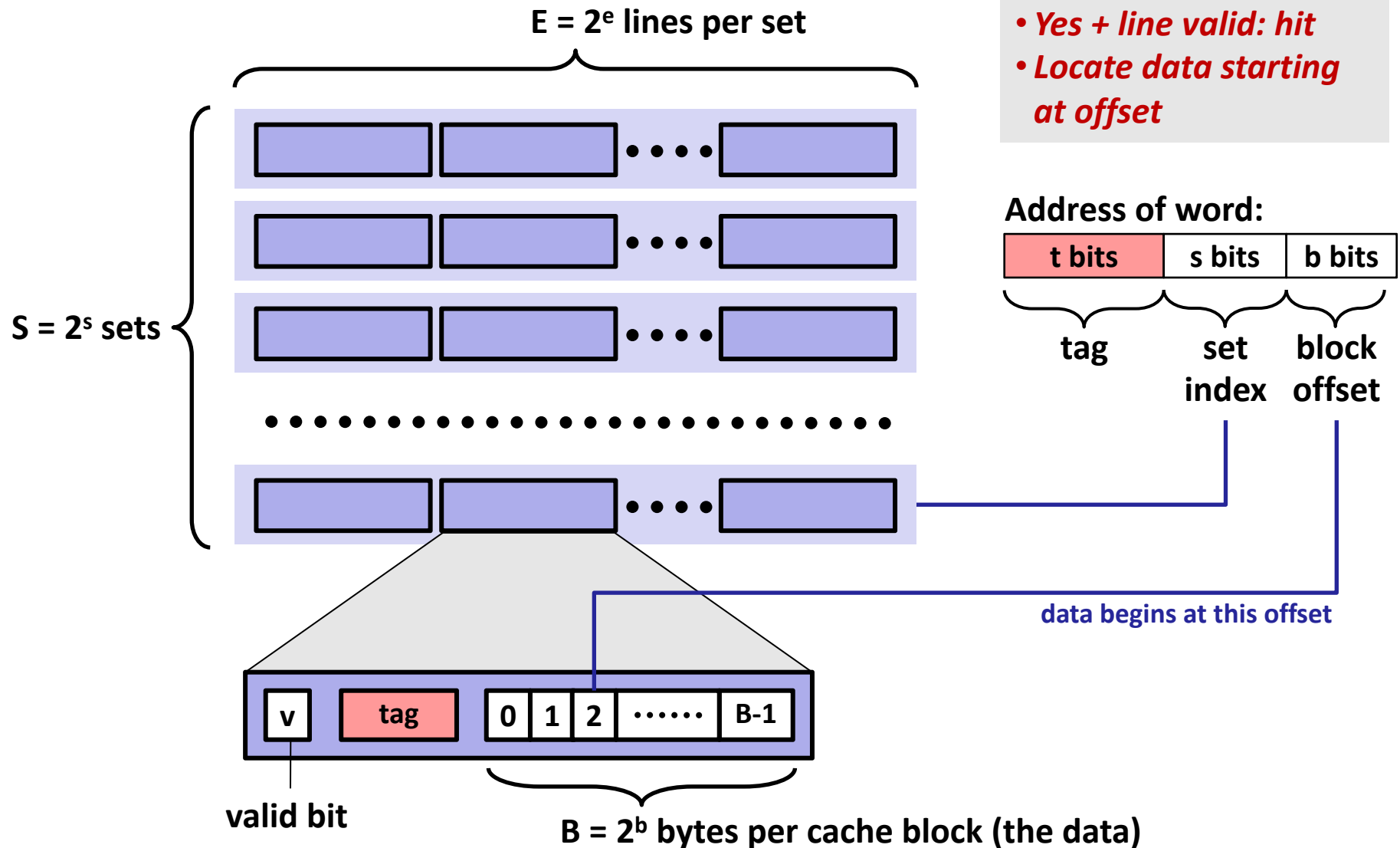
S=2 sets, E=2 blocks/set, B=2 bytes/block

Address trace (reads, one byte per read):

0	[00 <u>0</u> 0 ₂],	miss
1	[00 <u>0</u> 1 ₂],	hit
7	[0 <u>1</u> <u>1</u> 1 ₂],	miss
8	[<u>1</u> 0 <u>0</u> 0 ₂],	miss
0	[00 <u>0</u> 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		

Cache Read



What about writes?

■ Multiple copies of data exist:

- L1, L2, L3, Main Memory, Disk

■ What to do on a write-hit?

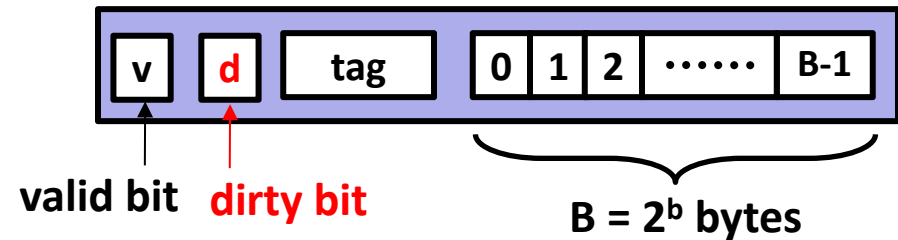
- **Write-through** (write immediately to memory)
- **Write-back** (defer write to memory until replacement of line)
 - Needs a dirty bit (set if data has been written to)

■ What to do on a write-miss?

- **Write-allocate** (load into cache, update line in cache)
 - Good if more writes to the location will follow
- **No-write-allocate** (writes straight to memory, does not load into cache)

■ Typical

- Write-through + No-write-allocate
- Write-back + Write-allocate

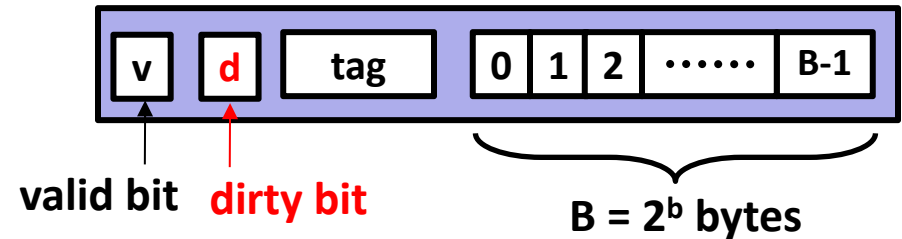


Practical Write-back Write-allocate

- A write to address X is issued

- If it is a hit

- Update the contents of block
 - Set dirty bit to 1 (bit is sticky and only cleared on eviction)



- If it is a miss

- Fetch block from memory (per a read miss)
 - The perform the write operations (per a write hit)

- If a line is evicted and dirty bit is set to 1

- The entire block of 2^b bytes are written back to memory
 - Dirty bit is cleared (set to 0)
 - Line is replaced by new contents

Why Index Using Middle Bits?

Direct mapped: One line per set

Assume: cache block size 8 bytes

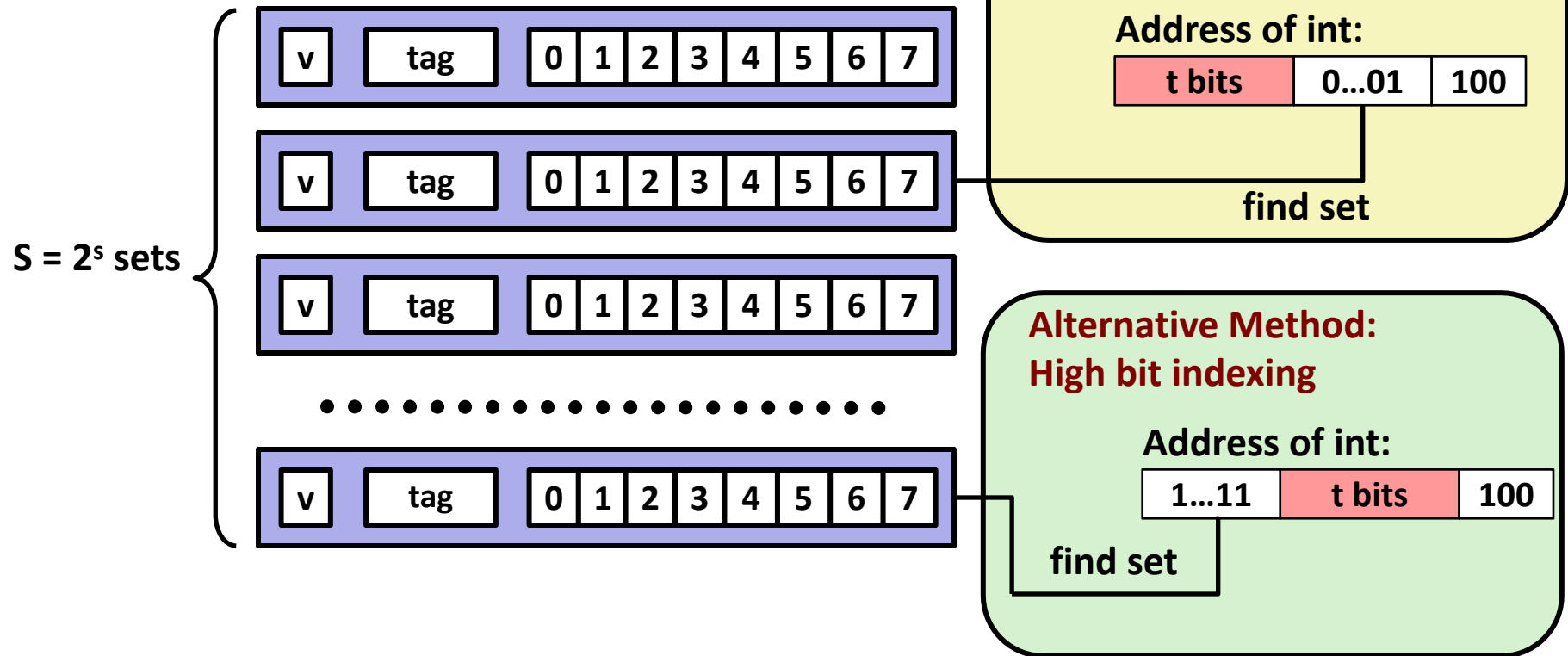
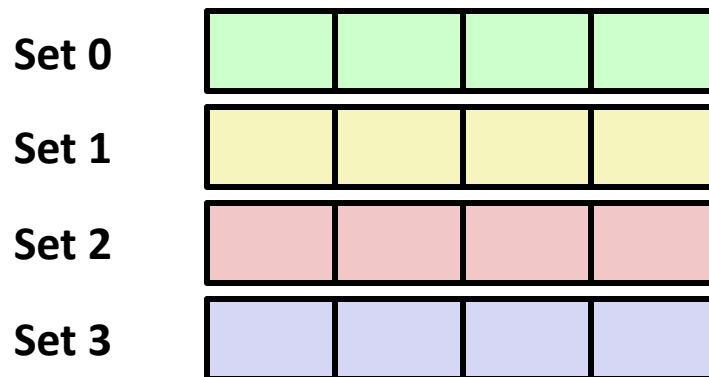


Illustration of Indexing Approaches

- 64-byte memory
 - 6-bit addresses
- 16 byte, direct-mapped cache
- Block size = 4. (Thus, 4 sets; why?)
- 2 bits tag, 2 bits index, 2 bits offset



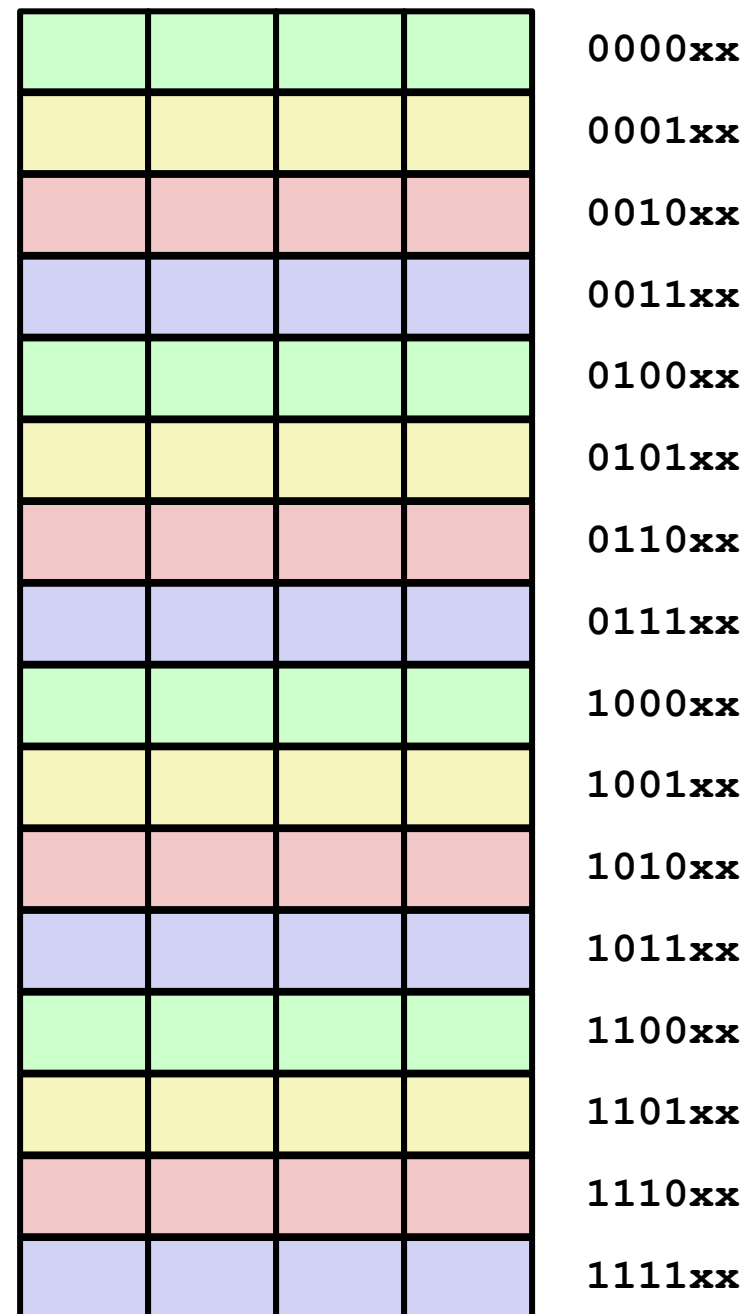
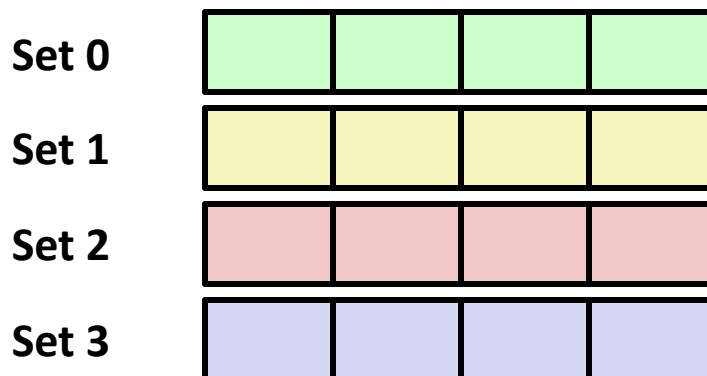
				0000xx
				0001xx
				0010xx
				0011xx
				0100xx
				0101xx
				0110xx
				0111xx
				1000xx
				1001xx
				1010xx
				1011xx
				1100xx
				1101xx
				1110xx
				1111xx

Middle Bit Indexing

■ Addresses of form **TTSSBB**

- **TT** Tag bits
- **SS** Set index bits
- **BB** Offset bits

■ Makes good use of spatial locality

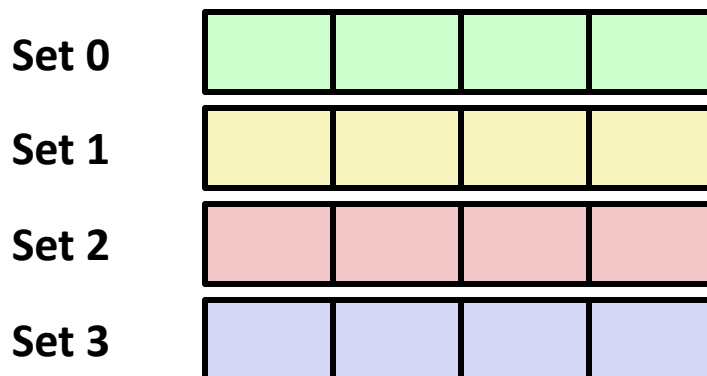


High Bit Indexing

■ Addresses of form **SS****TT****BB**

- **SS** Set index bits
- **TT** Tag bits
- **BB** Offset bits

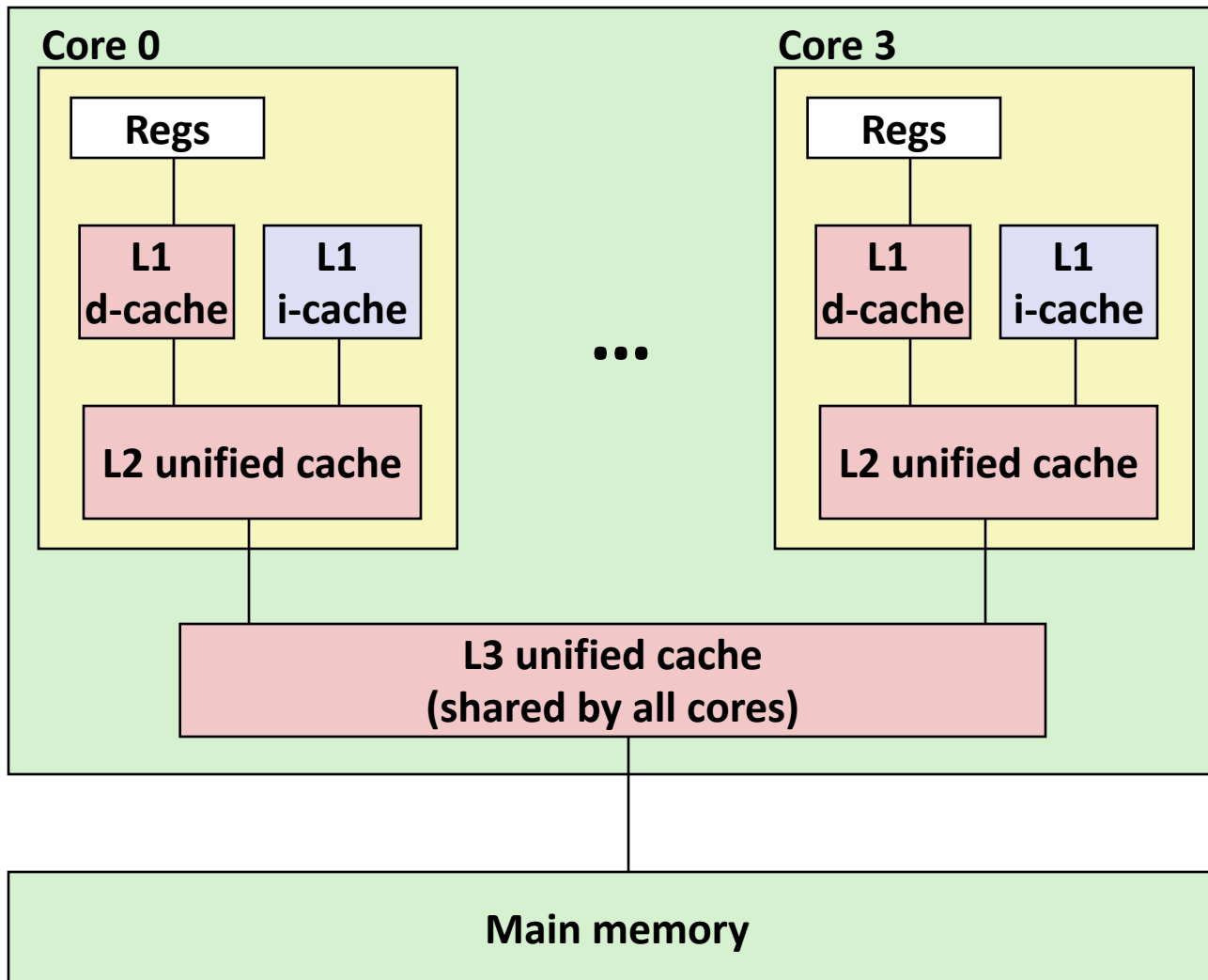
■ Program with high spatial locality would generate lots of conflicts



				0000xx
				0001xx
				0010xx
				0011xx
				0100xx
				0101xx
				0110xx
				0111xx
				1000xx
				1001xx
				1010xx
				1011xx
				1100xx
				1101xx
				1110xx
				1111xx

Intel Core i7 Cache Hierarchy

Processor package



L1 i-cache and d-cache:

32 KB, 8-way,
Access: 4 cycles

L2 unified cache:

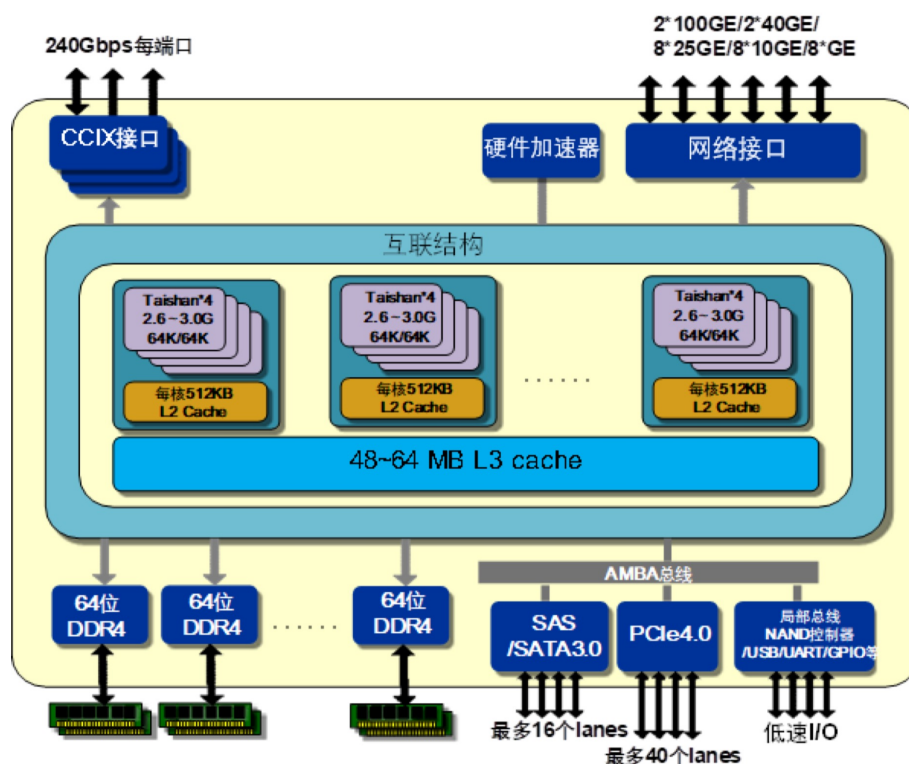
256 KB, 8-way,
Access: 10 cycles

L3 unified cache:

8 MB, 16-way,
Access: 40-75 cycles

Block size: 64 bytes for
all caches.

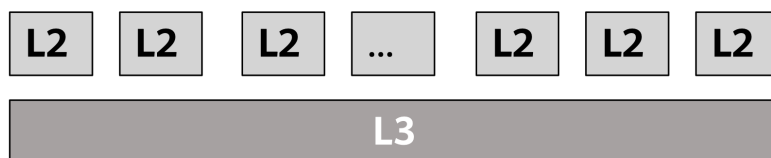
Kunpeng 920 Cache Hierarchy



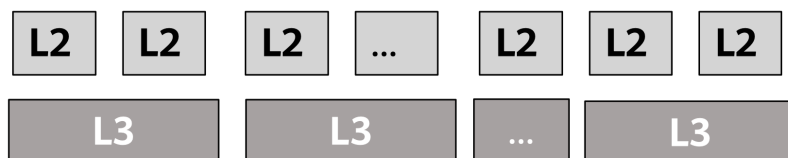
- 集成最多64 × 自研核
 - 指令集兼容ARMv8.2, 最高主频达3.0GHz
 - 每核集成64KB L1 I/D缓存
 - 每核独享512KB L2缓存, 单芯片共享48-64MB L3缓存
- 8 × DDR4控制器@2933MT/s
- 集成PCI-e/SAS接口
 - 支持PCI-e 4.0, 向下兼容PCI-e 3.0/2.0/1.0
 - 支持x16,x8,x4,x2,x1 PCI-e 4.0, 集成20 PCI-e控制器
 - 支持16 × SAS/SATA 3.0控制器
- 支持CCIX接口, 支持加速器的缓存一致性
- 支持2 × 100G RoCE v2, 支持25GE/50GE/100GE标准NIC
- 支持2P/4P扩展
- 封装大小: 60mm × 75mm

Kunpeng 920 Cache Hierarchy

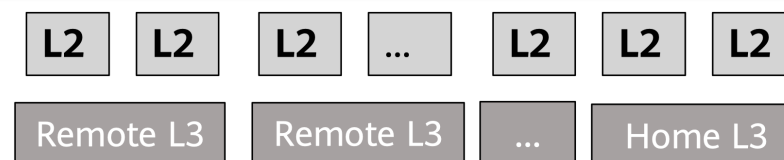
- Share Cache: 对所有的L2来说L3 cache是共享的，一个进程可以使用整个L3的容量



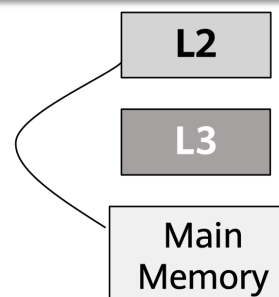
- Private Cache: 有N个Private的L3，每个Private L3只缓存对应的L2的数据。即一个进程只能使用对应的部分L3的容量，无法使用全部L3的容量，L3和L3之间不通信



- Partitioned Cache: 与Private相同的是，一个进程只能使用对应的部分L3容量；与Private不同的是，L3细分为一个Home的L3和N个Remote的L3，Home的L3类似L4，所以L3和L3之间会通信，由Home的L3来维护多个Partitioned L3之间的一致性



- Non-inclusive L3: 支持Non-inclusive模式，Memory和L2间直接数据访问



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 this simplified example:
cache hit time of 1 cycle
miss penalty of 100 cycles
 - Average access time:
97% hits: $1 \text{ cycle} + 0.03 \times 100 \text{ cycles} = 4 \text{ cycles}$
99% hits: $1 \text{ cycle} + 0.01 \times 100 \text{ cycles} = 2 \text{ cycles}$
- **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**)

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)
单位时间内的阅读量
- **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 4x4 loop unrolling.
 */
/* work set => 时间局部性      步长 => 空间局部性
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.

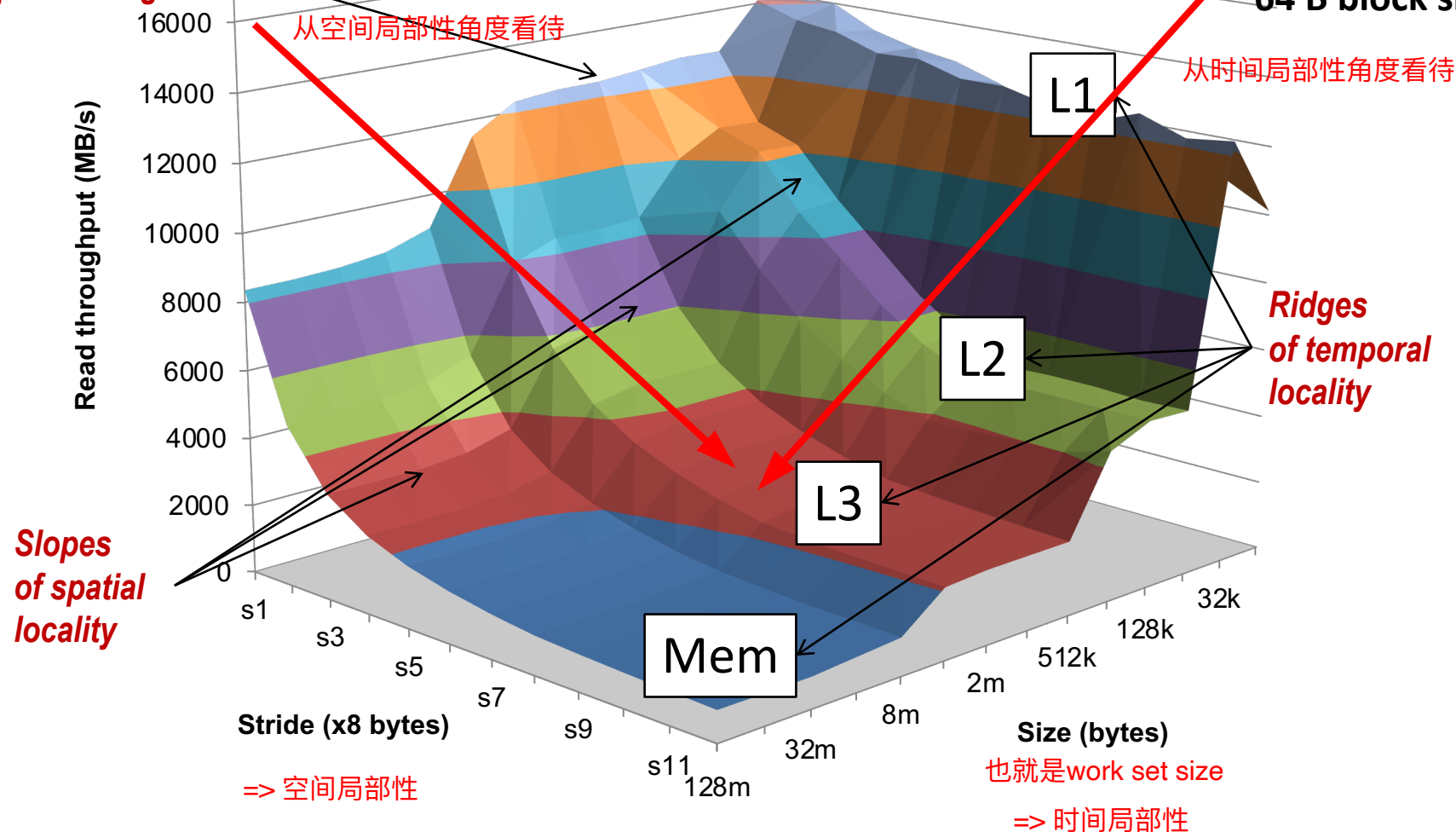
For each elems and stride:

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

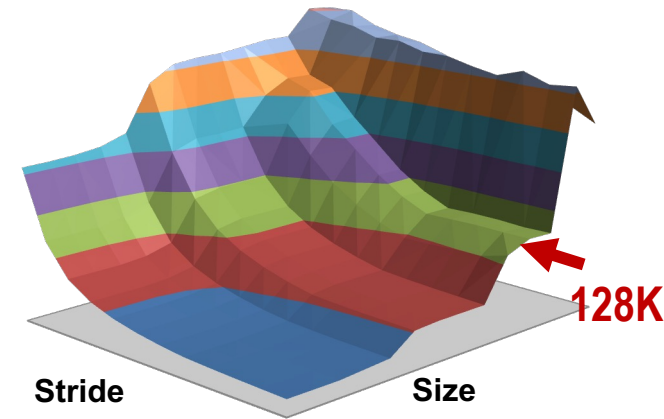
The Memory Mountain

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

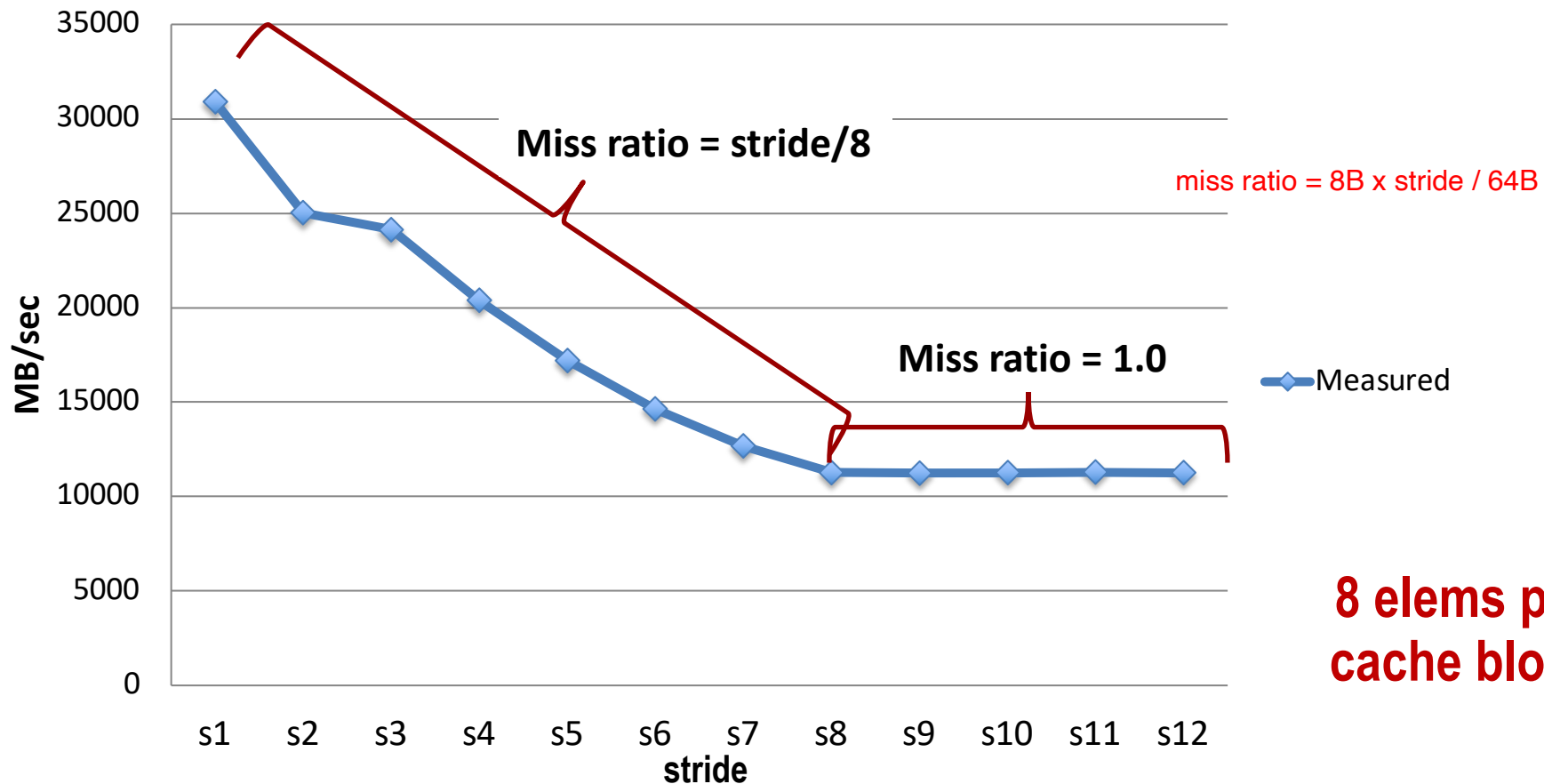
**Aggressive
prefetching**



Closer Look at Stride Effects



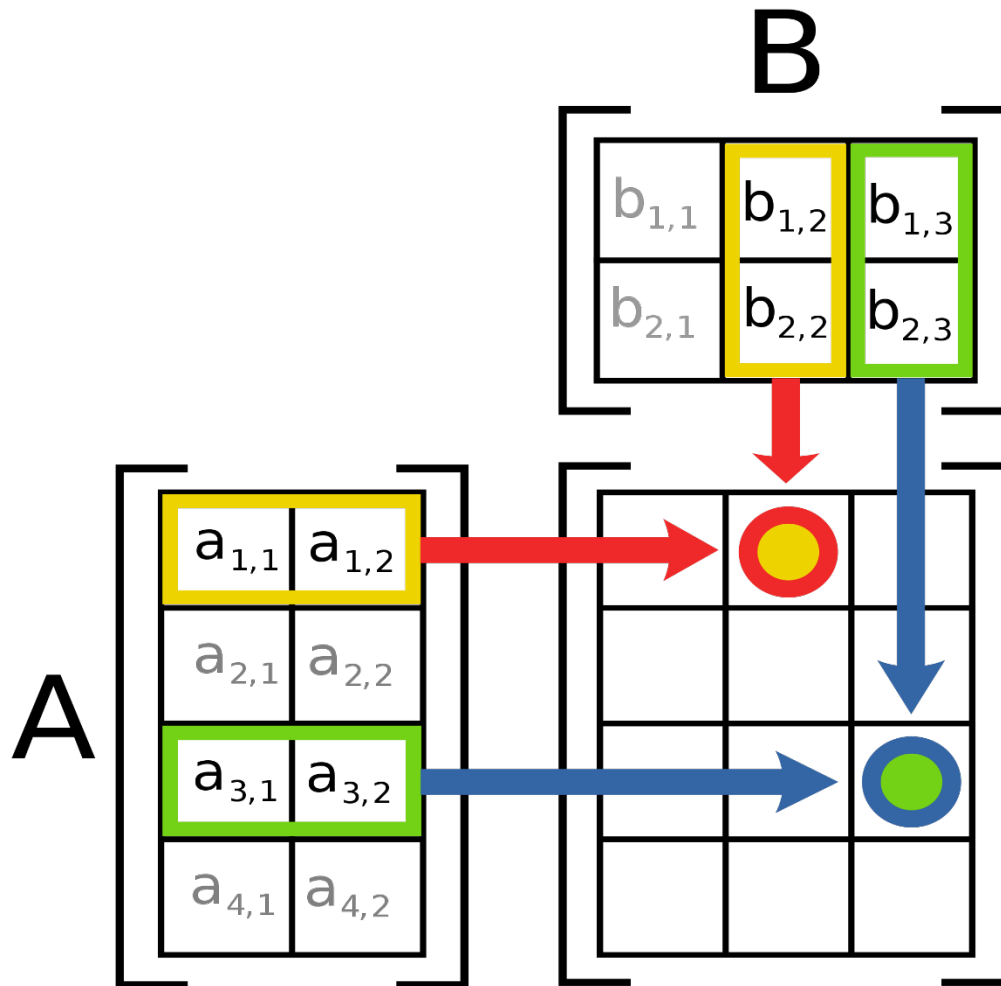
Throughput for size = 128K



Today

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

Remember matrix multiplication



$$\begin{aligned} \text{Out}[i, j] = & \text{dot product}(A[i, :], B[:, j]) \\ = & \text{sum} (\\ & a[i, 0] * b[0, j], \\ & a[i, 1] * b[1, j] \\ &) \end{aligned}$$

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

矩阵乘法的计算代码

```

/* 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*

访问b的空间局部性很差

matmult/mm.c

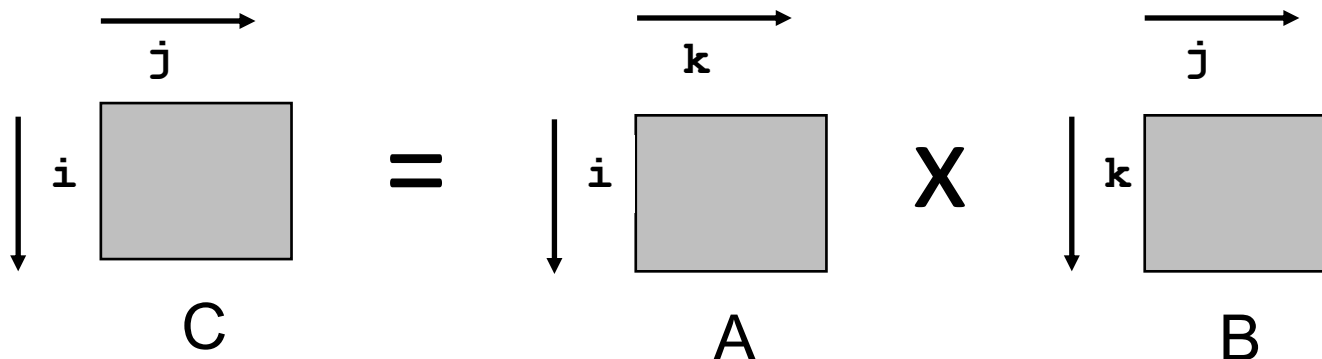
Miss Rate Analysis for Matrix Multiply

■ 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
- $a[i][j] = a[i*N + j]$ where N is the number of columns

■ 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)

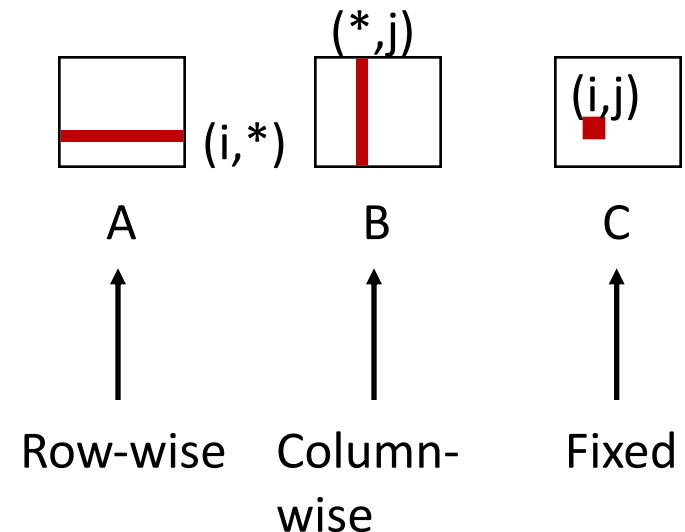
```

/* 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:



Miss rate for inner loop iterations:

A

B

C

Block size = 32B (four doubles)

Matrix Multiplication (ijk)

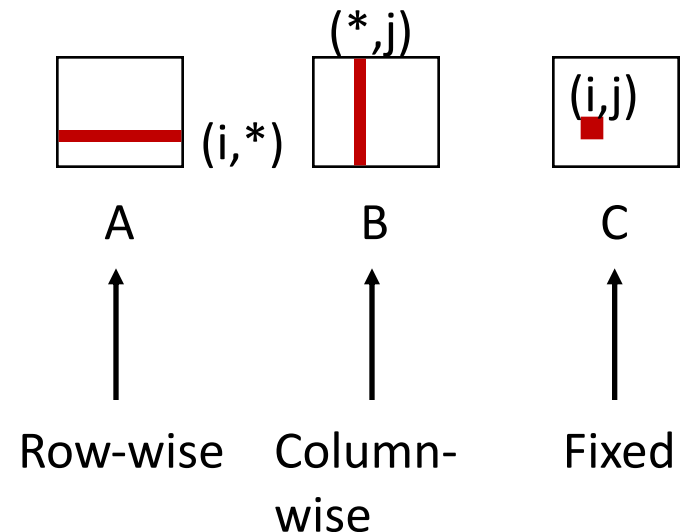
```

/* 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:



Miss rate for inner loop iterations:

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

Total = 1.25

Block size = 32B (four doubles)

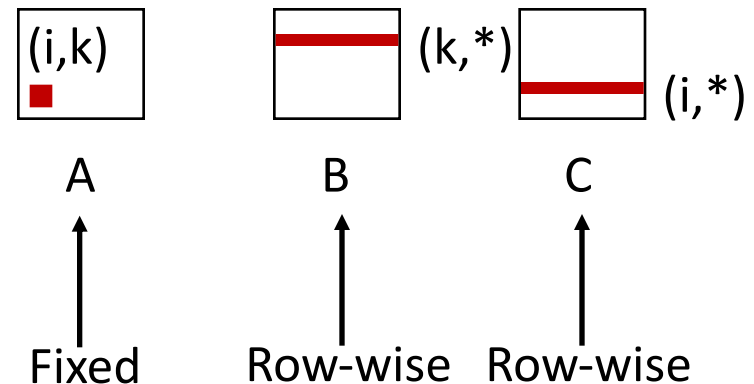
Matrix Multiplication (kij)

```

/* 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:



Miss rate for inner loop iterations:

A

B

C

Block size = 32B (four doubles)

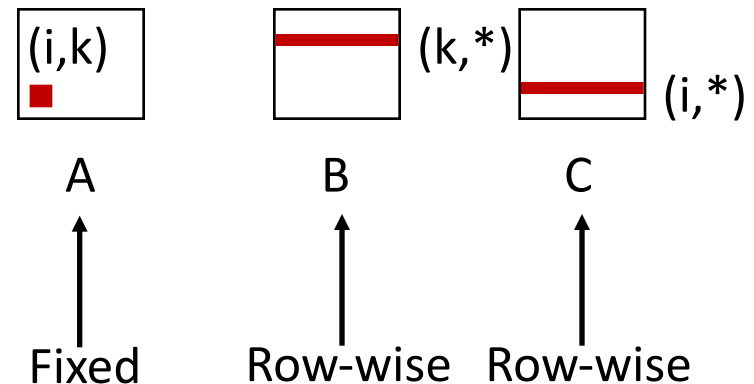
Matrix Multiplication (kij)

```

/* 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:



Miss rate for inner loop iterations:

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

Block size = 32B (four doubles)

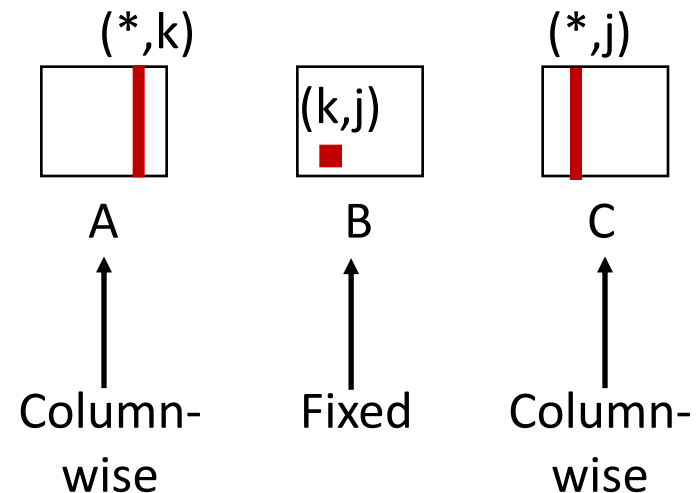
Matrix Multiplication (jki)

```

/* 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:



Miss rate for inner loop iterations:

A

B

C

Block size = 32B (four doubles)

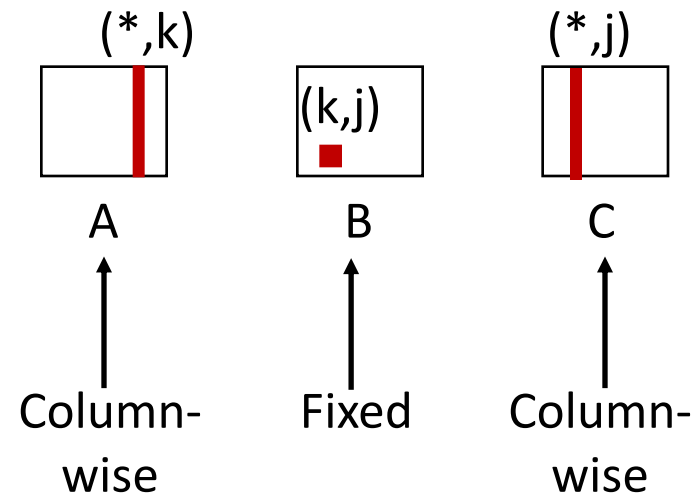
Matrix Multiplication (jki)

```

/* 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:



Miss rate for inner loop iterations:

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

Block size = 32B (four doubles)

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
- avg 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
- avg 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;
    }
}
```

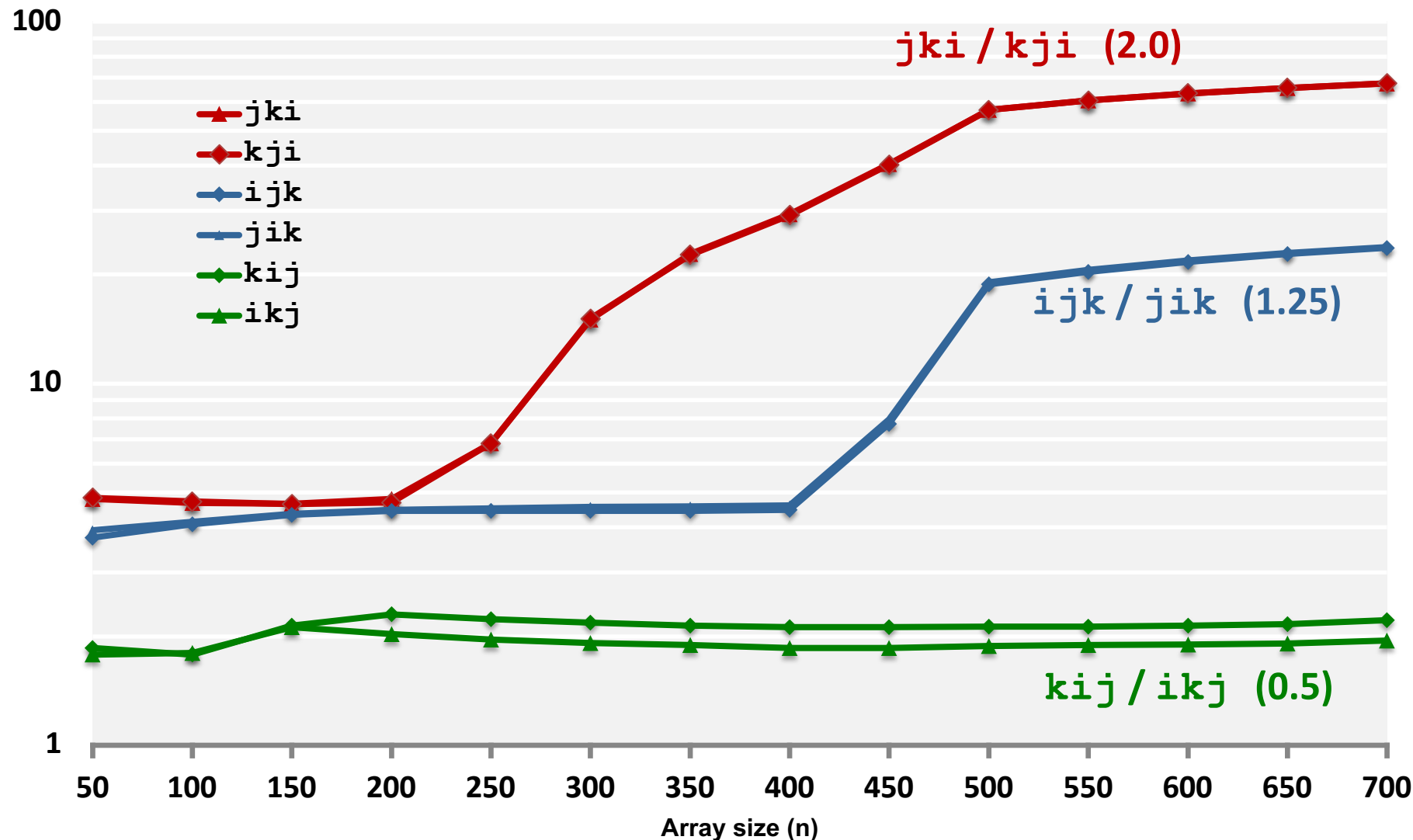
jki (& kji):

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

Core i7 Matrix Multiply Performance

Cycles per inner loop iteration

外面两层循环基本没有影响
内层循环影响大



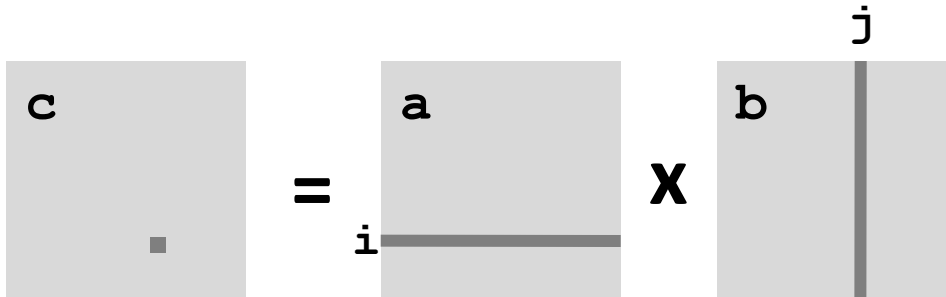
Today

- 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

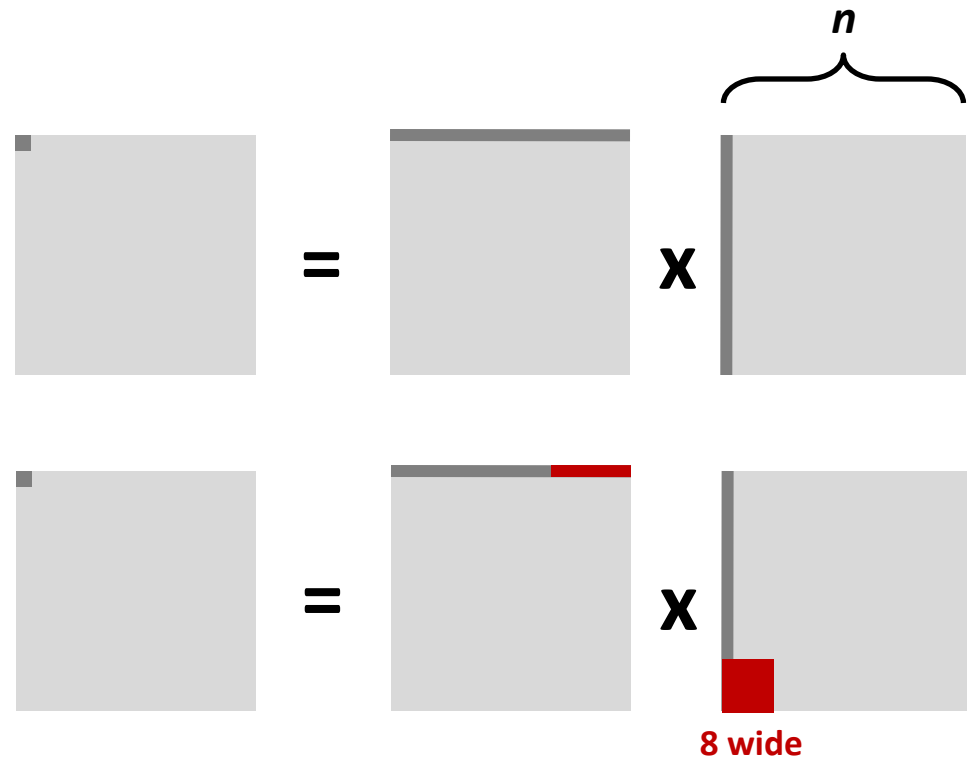
■ Assume:

- 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

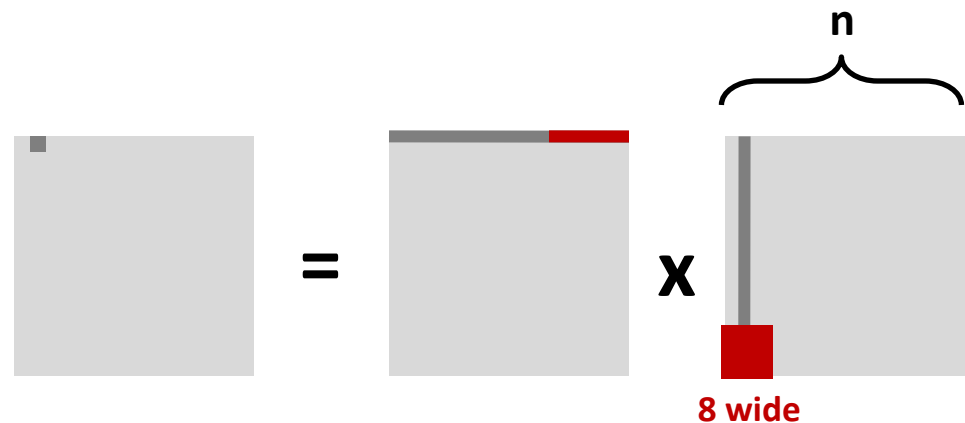
■ Assume:

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

■ Second iteration:

- Again:

$$n/8 + n = 9n/8 \text{ misses}$$



■ Total misses:

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

Blocked Matrix Multiplication

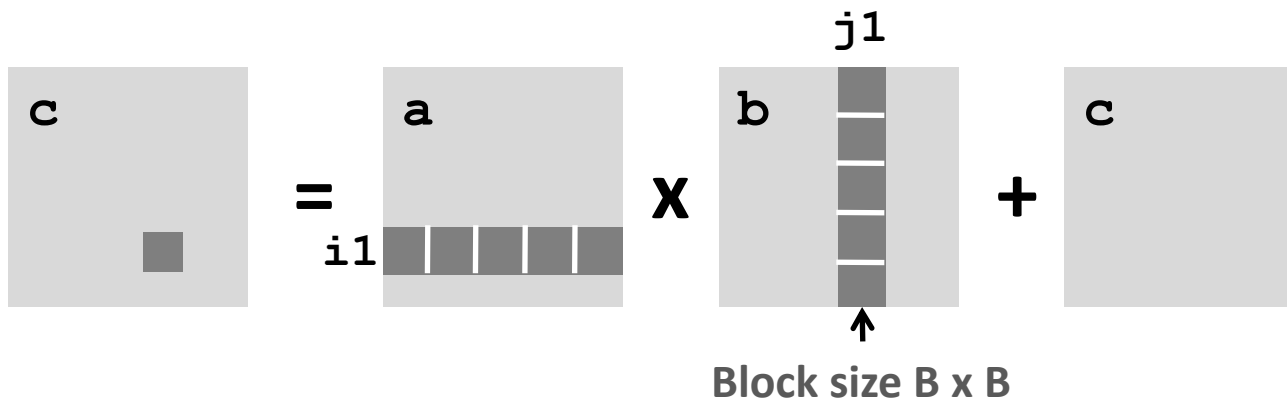
```

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; i1++)
                    for (j1 = j; j1 < j+B; j1++)
                        for (k1 = k; k1 < k+B; k1++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}


```

matmult/bmm.c



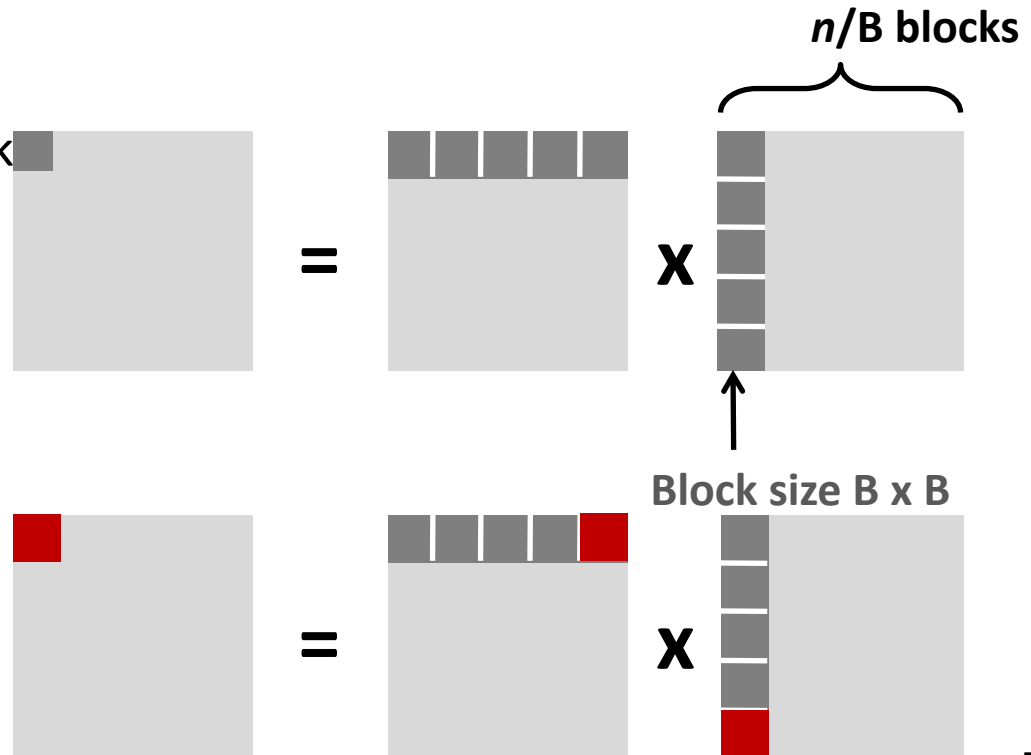
Cache Miss Analysis

■ 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:


- $B*B/8$ misses for each block
- $2n/B \times B^2/8 = nB/4$
(omitting matrix c)



- Afterwards in cache
(schematic)

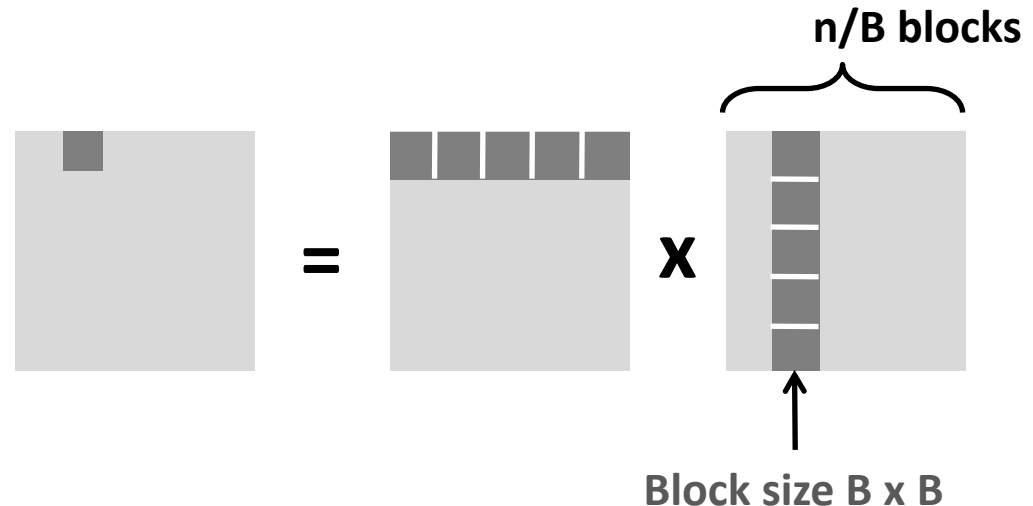
Cache Miss Analysis

■ 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
- $2n/B \times B^2/8 = nB/4$



■ Total misses:

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

Blocking Summary

- No blocking: $(9/8) n^3$ misses
- Blocking: $(1/(4B)) n^3$ misses
- **Use largest block size B , such that B satisfies $3B^2 < C$**
 - Fit three blocks in cache! Two input, one output.
- **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.