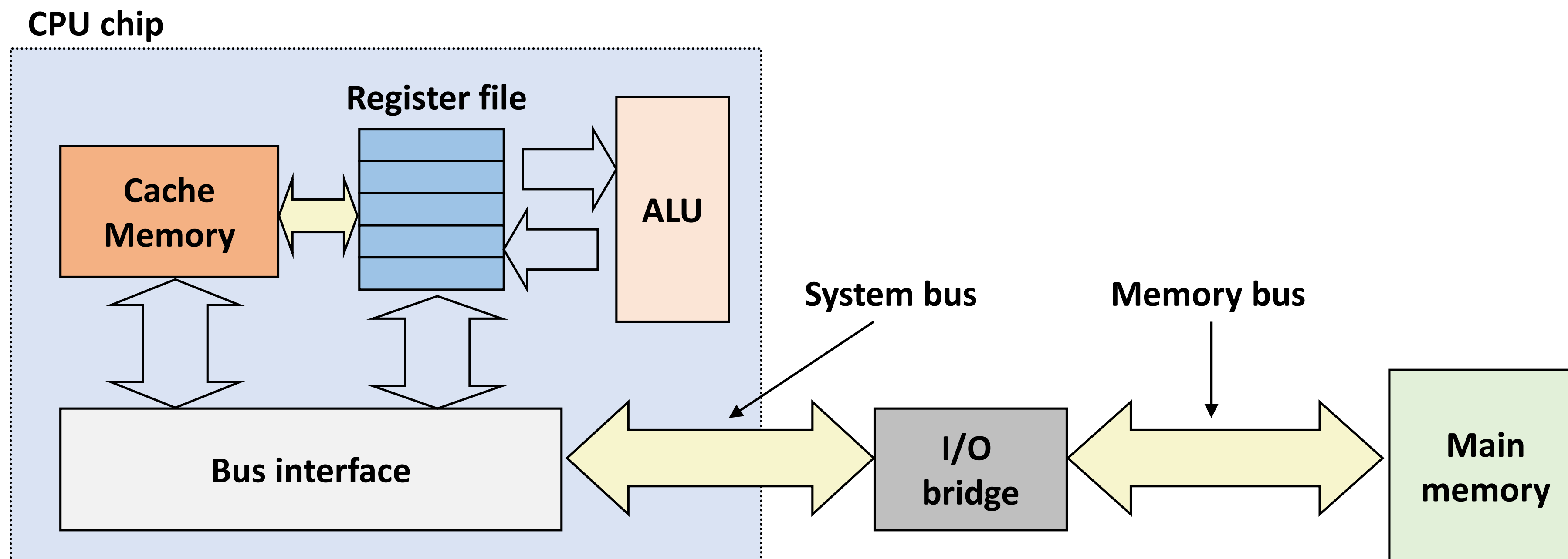
Cache Memories

Jin-Soo Kim
(jinsoo.kim@snu.ac.kr)
Systems Software &
Architecture Lab.
Seoul National University

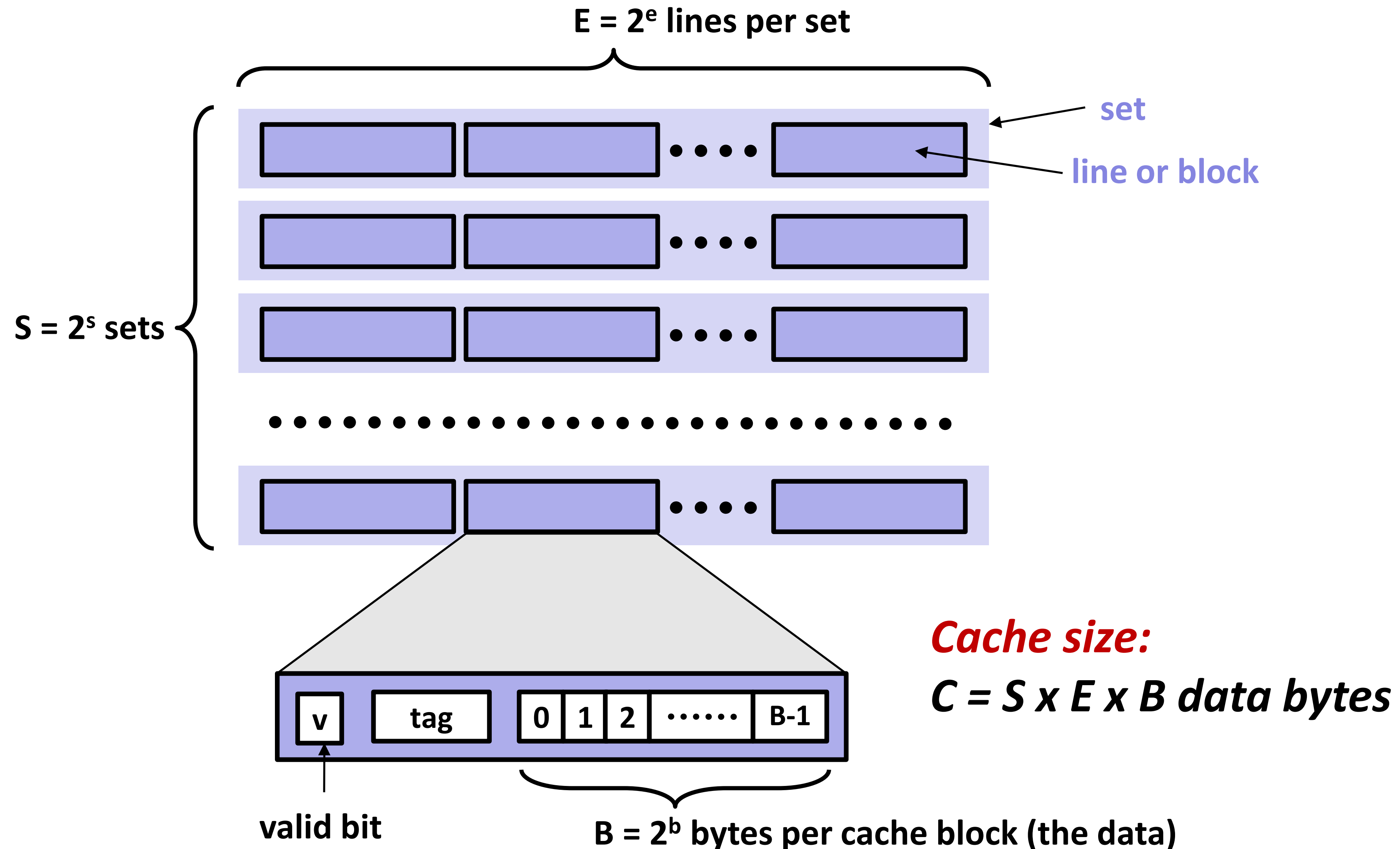
Spring 2019

Cache Memories

- 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)

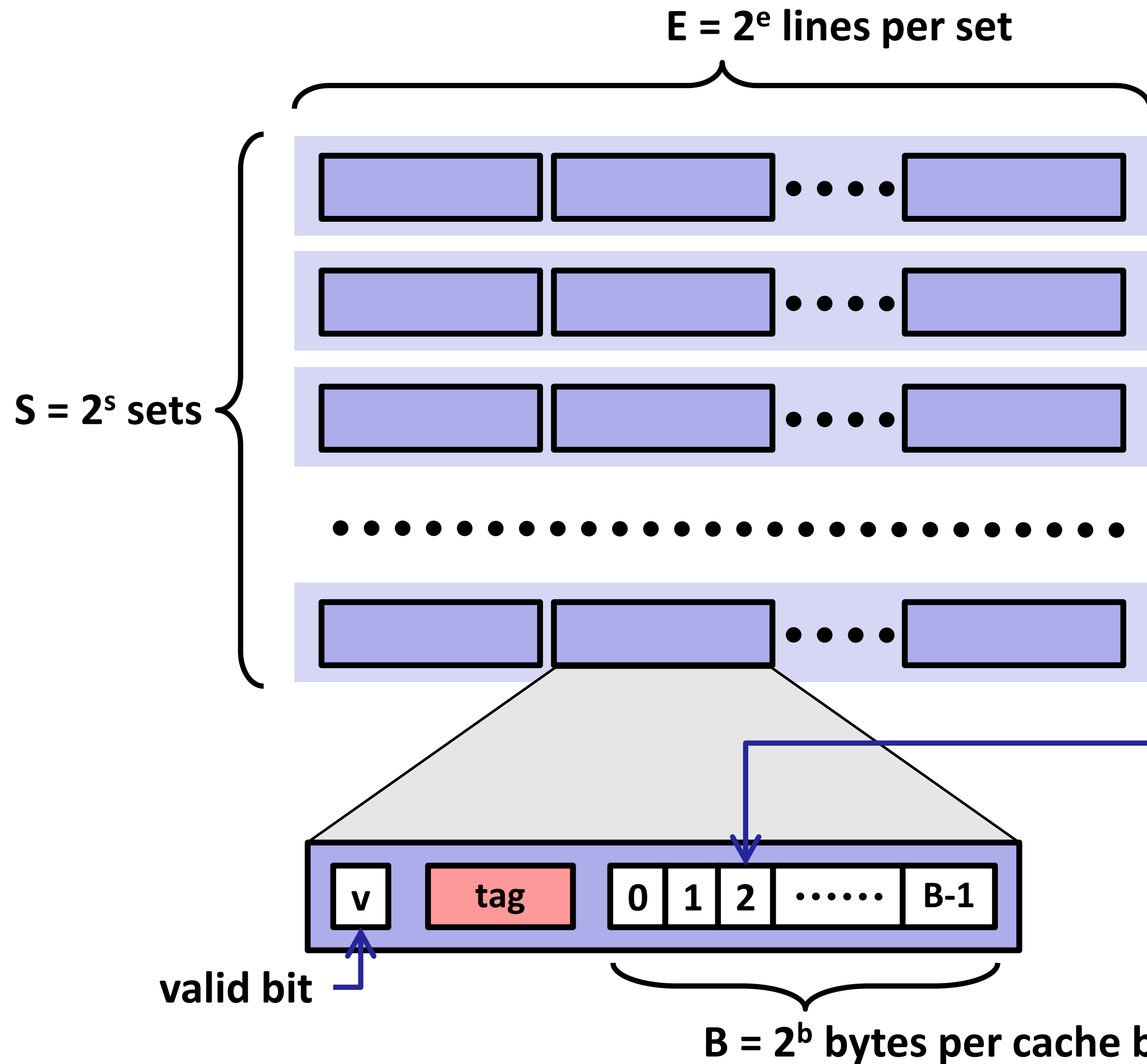


Cache size:

C = S x E x B data bytes

Cache Read

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



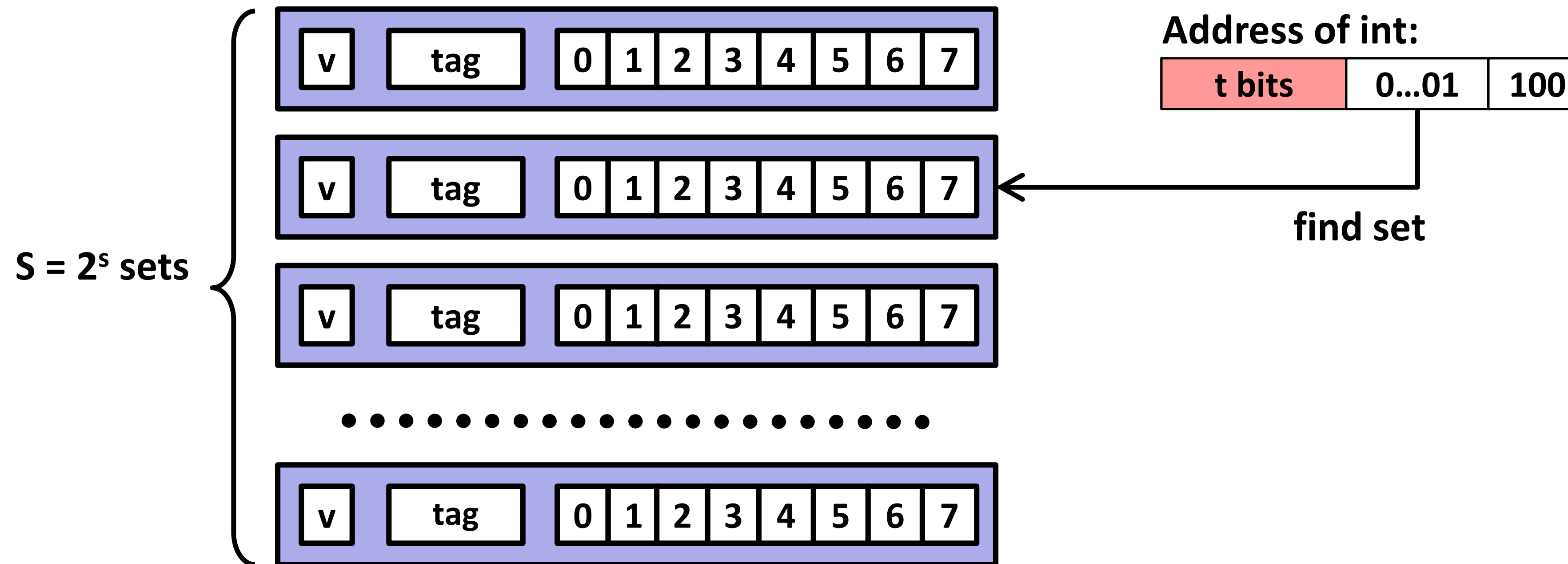
Address of word:

t bits	s bits	b bits
tag	set index	block offset

data begins at this offset

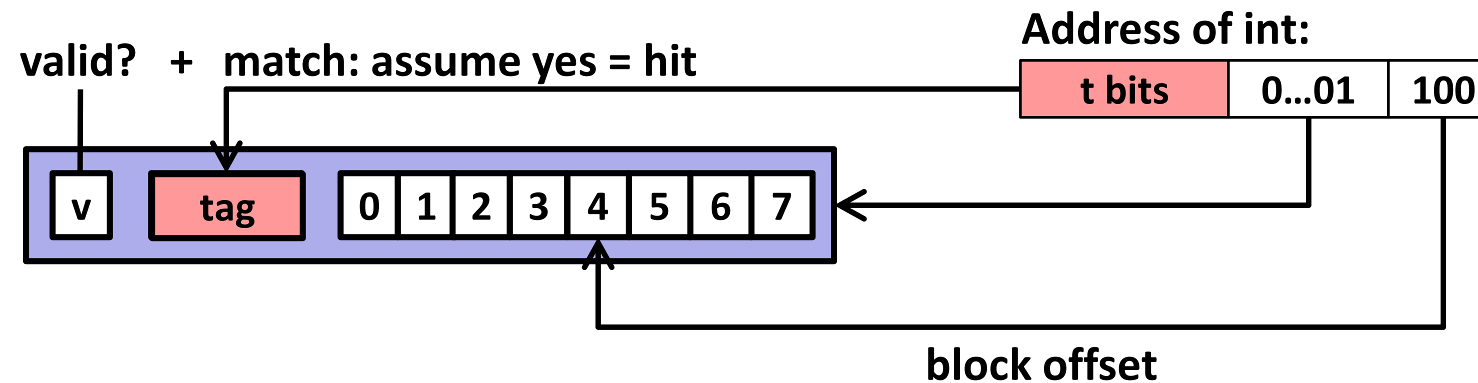
Direct Mapped Cache (E = 1)

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



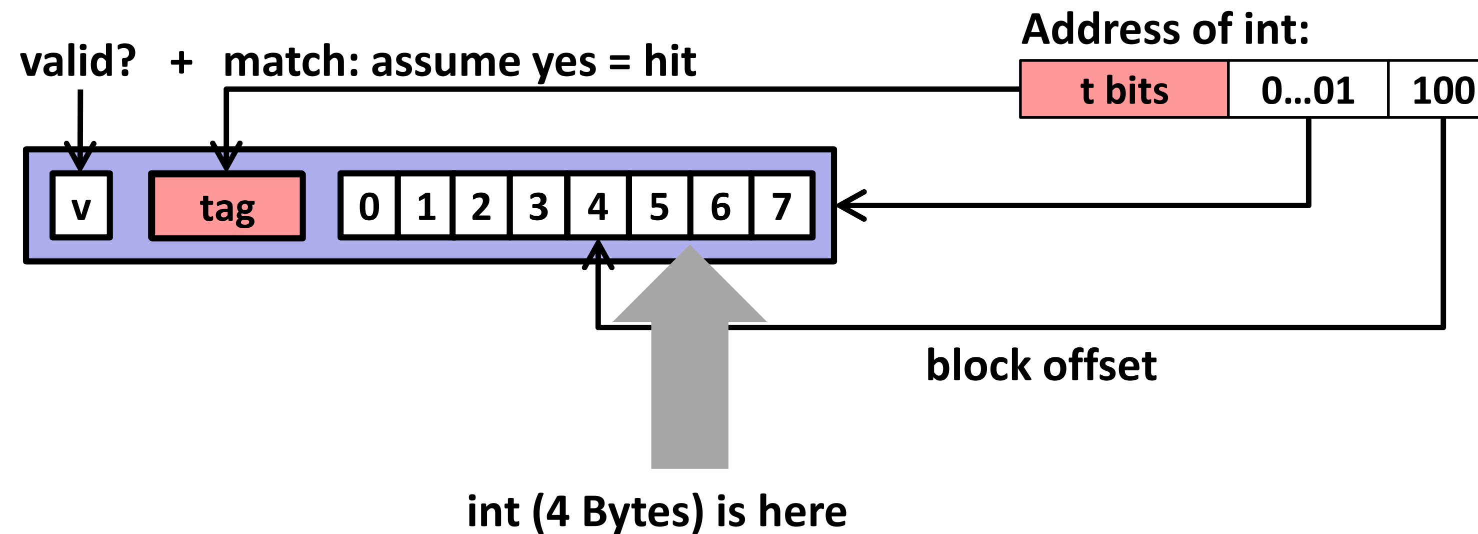
Direct Mapped Cache (E = 1)

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



Direct Mapped Cache (E = 1)

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



No match: old line is evicted and replaced

Direct Mapped Cache Simulation

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

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

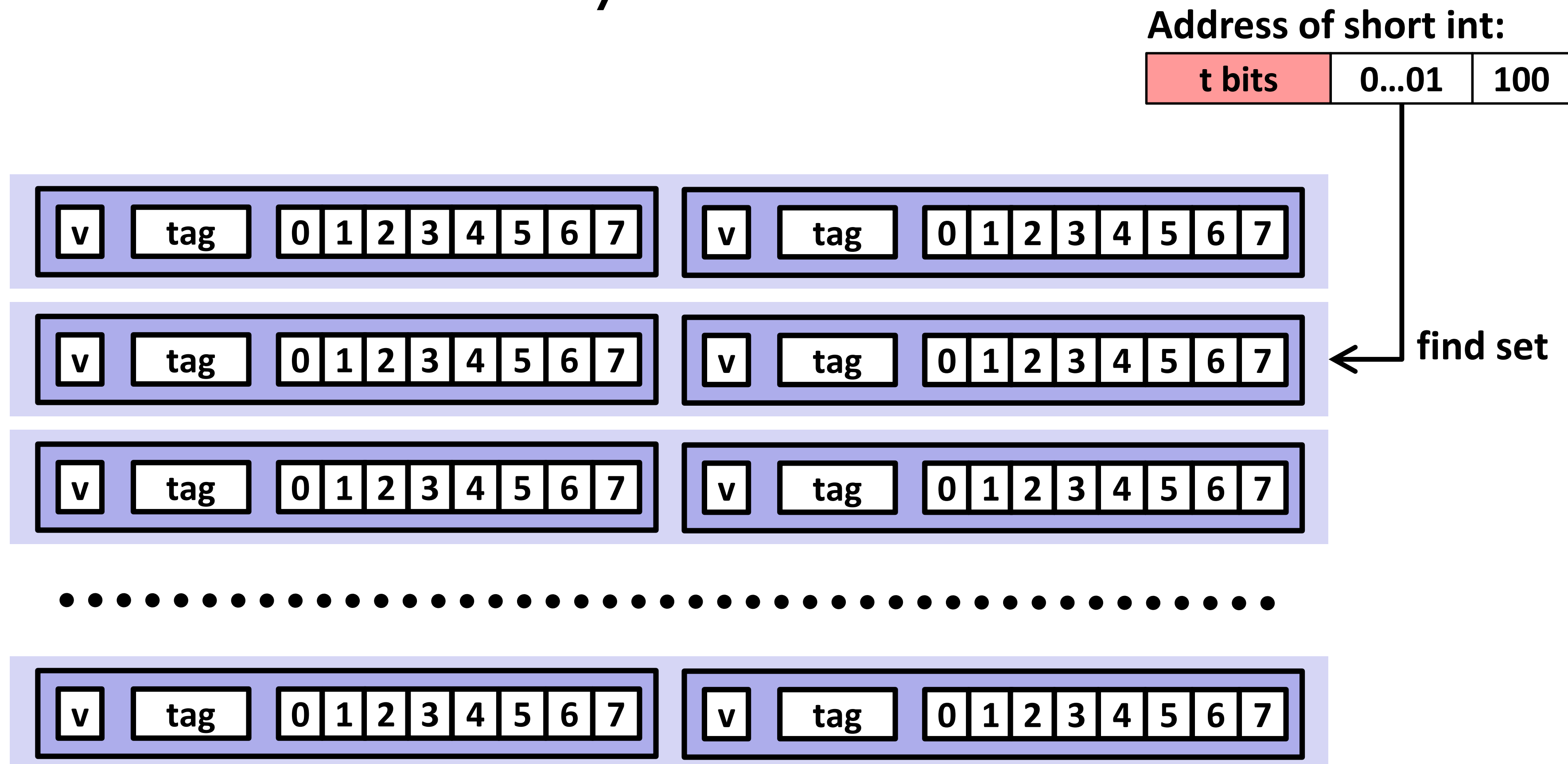
	v	Tag	Block
Set 0	1	0	M[0-1]
Set 1			
Set 2			
Set 3	1	0	M[6-7]

Address trace (reads, one byte per read):

0	**[0000$_2$],**	miss	*(cold)*
1	**[0001$_2$],**	hit	
7	**[0111$_2$],**	miss	*(cold)*
8	**[1000$_2$],**	miss	*(cold)*
0	**[0000$_2$]**	miss	*(conflict)*

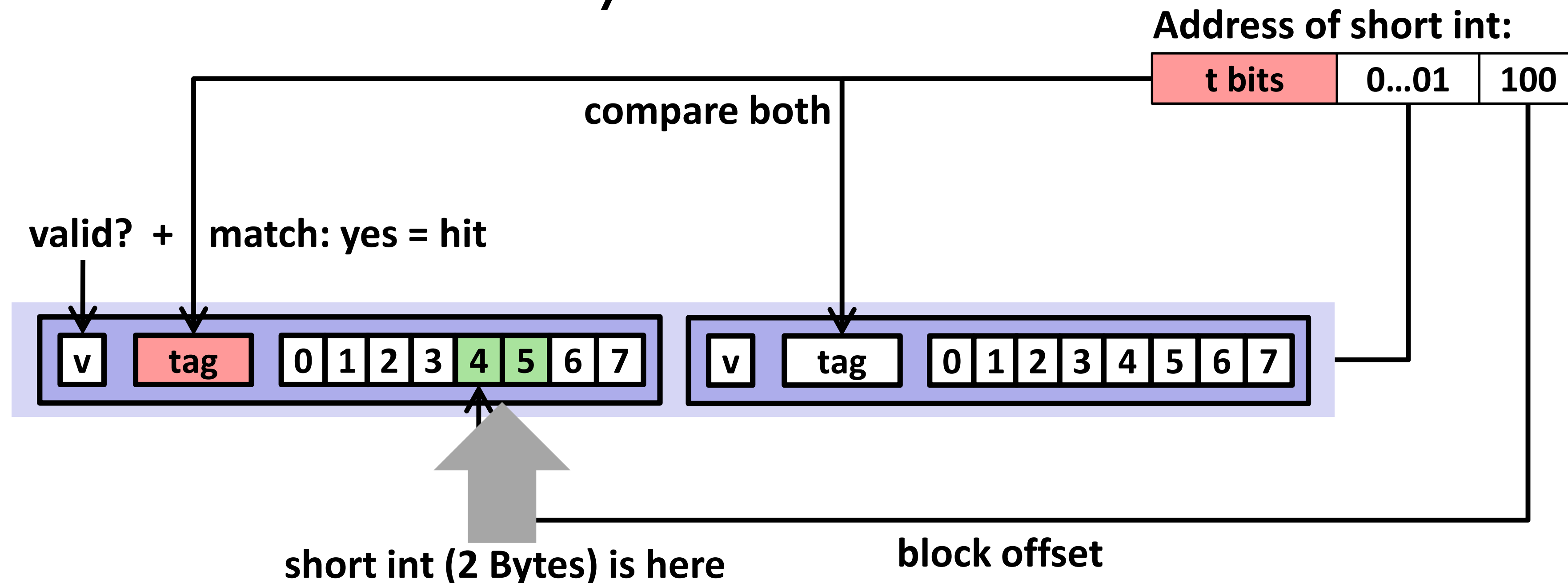
E-way Set Associative Cache

- Here E = 2: Two lines per set
- Assume cache block size 8 bytes



E-way Set Associative Cache

- Here 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, LRU, …

2-way Set Associative Cache Simulation

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

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

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

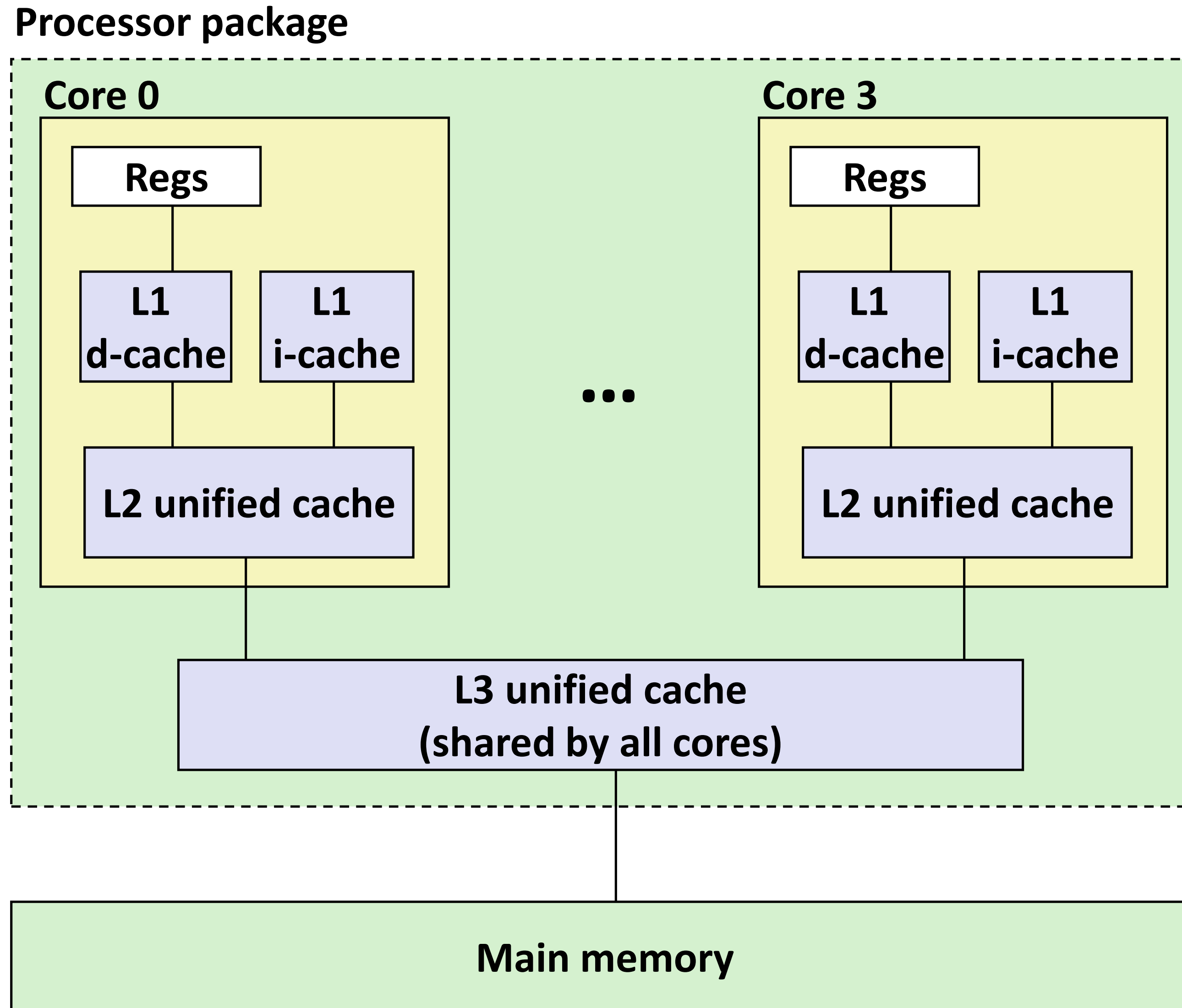
Address trace (reads, one byte per read):

0	**[0000$_2$],**	miss
1	**[0001$_2$],**	hit
7	**[0111$_2$],**	miss
8	**[1000$_2$],**	miss
0	**[0000$_2$]**	hit

What about Writes?

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

Intel Core i7 Cache Hierarchy



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

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

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

Block size:
64 bytes for all caches

Cache Performance Metrics

- Miss rate
  - Fraction of memory references not found in cache (misses / accesses) = 1 – hit rate
  - Typically, 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
  - Typically, 4 clock cycles for L1 and 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"

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

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

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

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

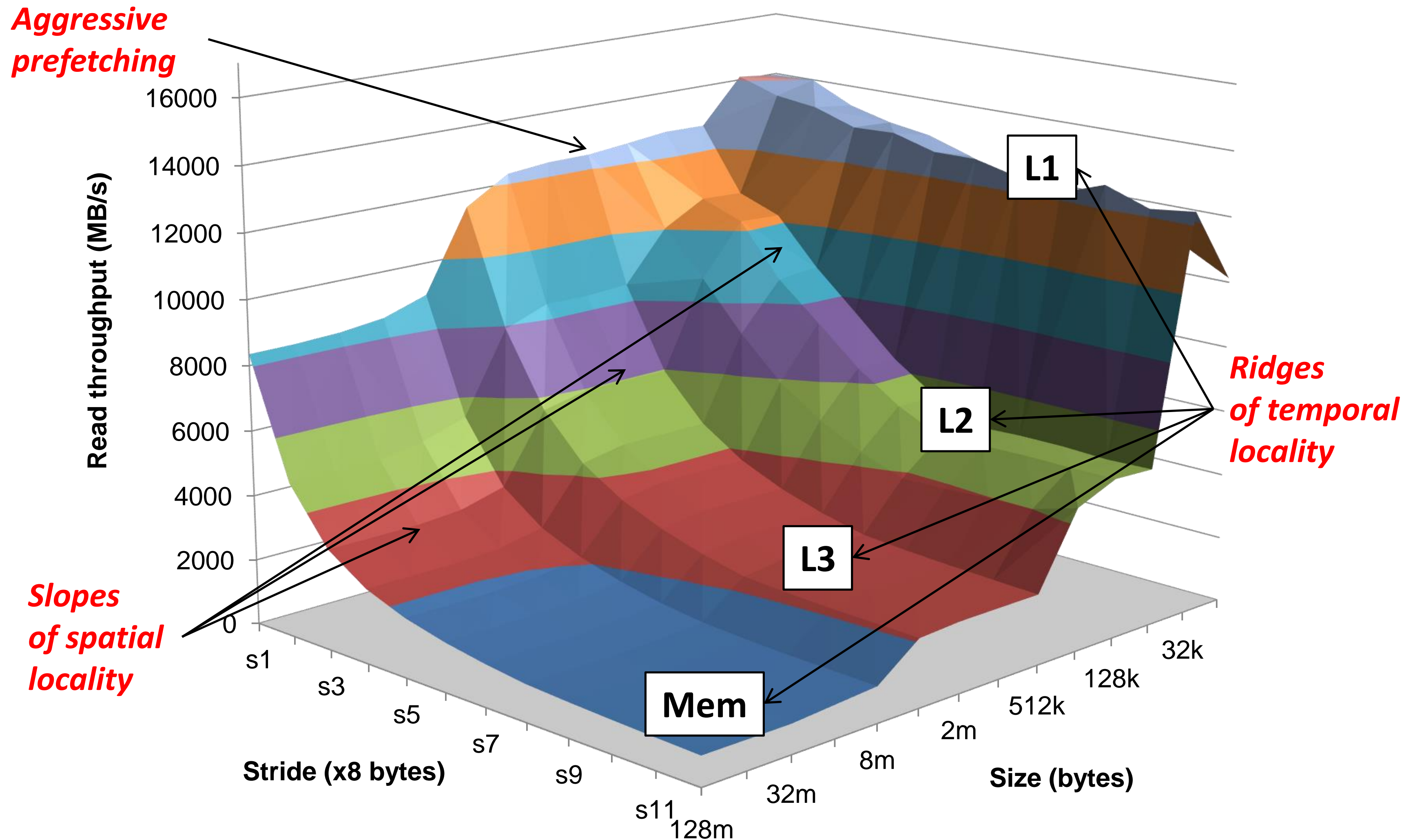
Call `test()` with many combinations of `elems` and `stride`.

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 Example



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

Writing Cache Friendly Code

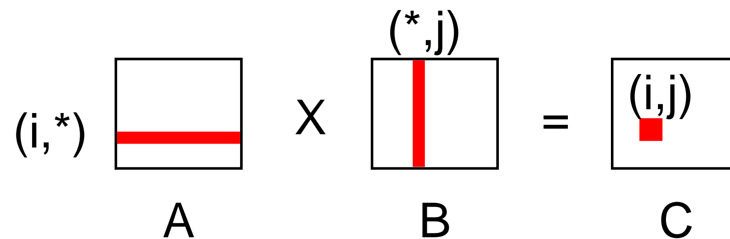
- Make the common case go fast
  - Focus on the inner loops of the core functions

- 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

Matrix Multiplication (1)

- **Description**
  - Multiply N x N matrices
  - $O(N^3)$ total operations




```
/* 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;
  }
}
```

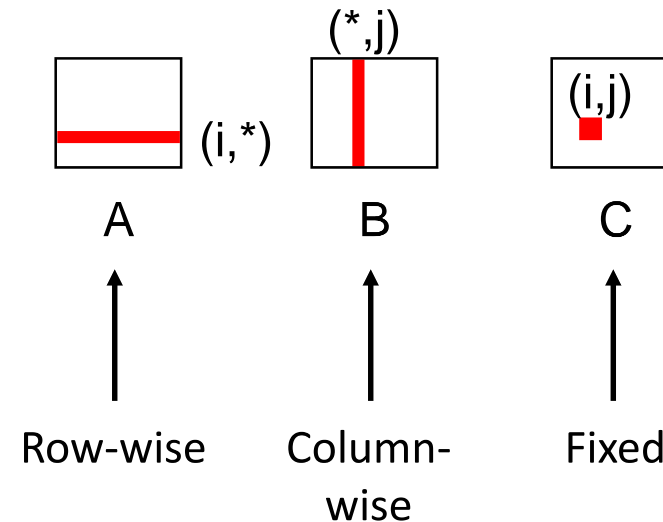
*Variable **sum** held in register*

- **Assumptions**
  - Line size = 32 bytes (big enough for 4 64-bit words)
  - Matrix dimension (N) is very large

Matrix Multiplication (2)

- Matrix multiplication (ijk)

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



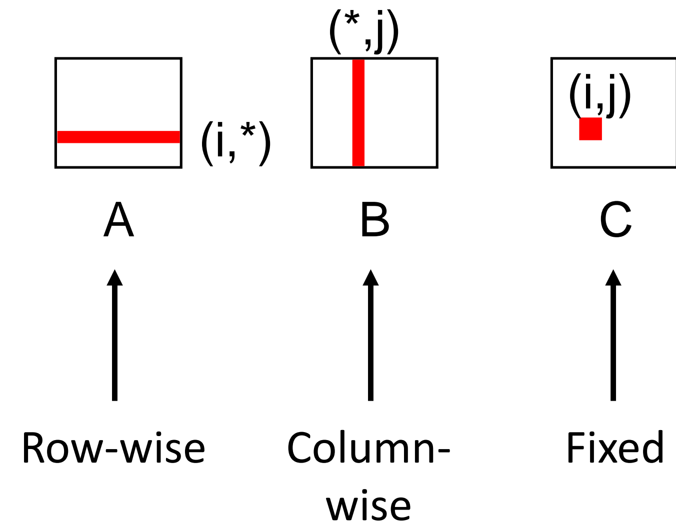
Misses per Inner Loop Iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (3)

- 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
  }
}
```



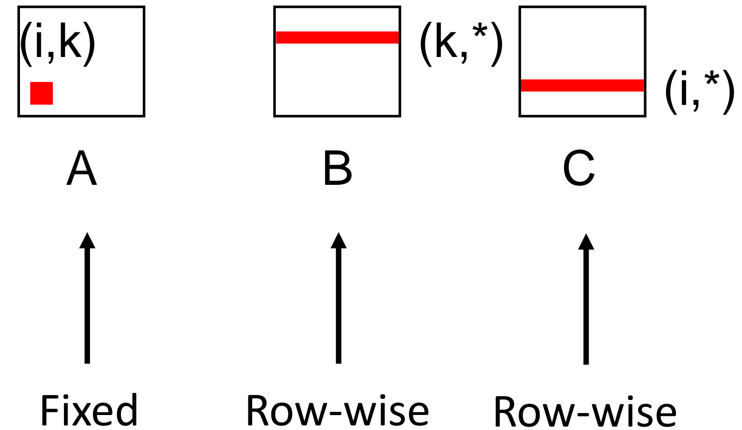
Misses per Inner Loop Iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (4)

- 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];
  }
}
```



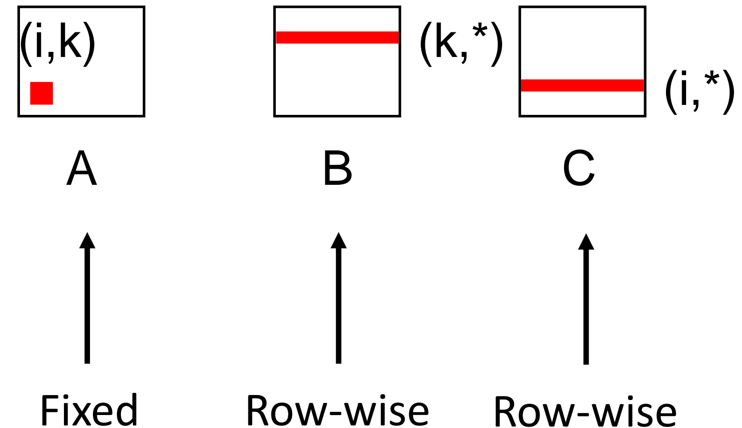
Misses per Inner Loop Iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (5)

- 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];
  }
}
```



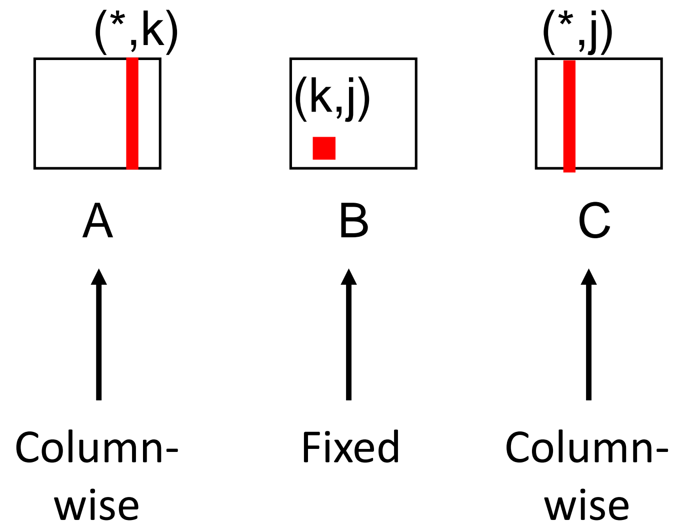
Misses per Inner Loop Iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (6)

- 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;
  }
}
```



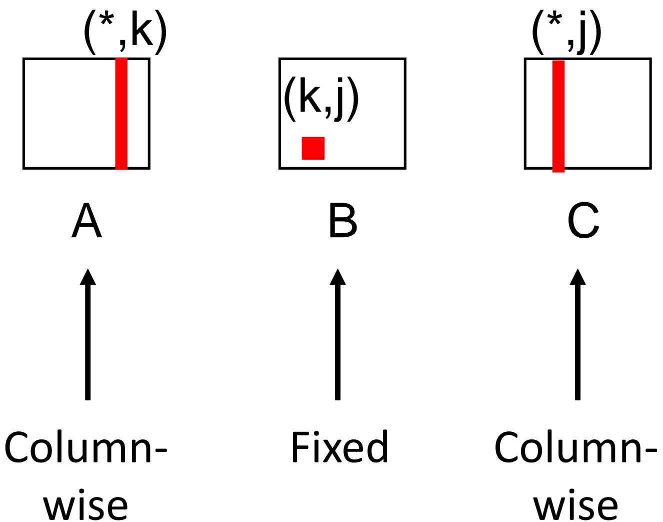
Misses per Inner Loop Iteration:

A	B	C
1.0	0.0	1.0

Matrix Multiplication (7)

- 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;
  }
}
```



Misses per Inner Loop Iteration:

A	B	C
1.0	0.0	1.0

Matrix Multiplication (8)

- Summary

ijk (& jik):
- 2 loads, 0 stores
- misses/iter = **1.25**

```
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;
 }
}
```

A	B	C
0.25	1.0	0.0

kij (& ikj):
- 2 loads, 1 store
- misses/iter = **0.5**

```
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];
 }
}
```

A	B	C
0.0	0.25	0.25

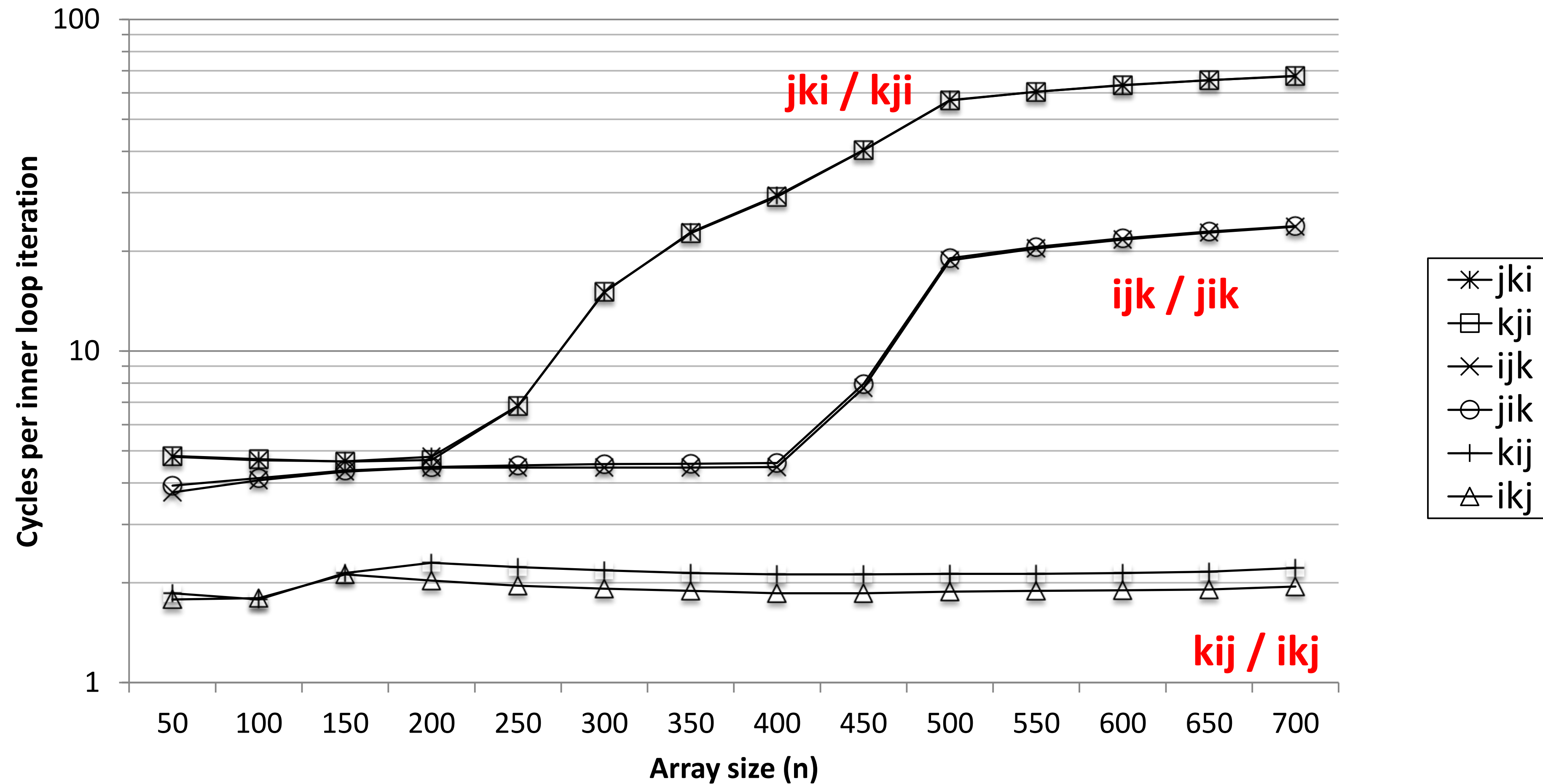
jki (& kji):
- 2 loads, 1 store
- misses/iter = **2.0**

```
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;
 }
}
```

A	B	C
1.0	0.0	1.0

Core i7 Matrix Multiplication Performance

- Performance in Core i7



Summary

- Cache memories can have significant performance impact

- You ~~can~~ *should* 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