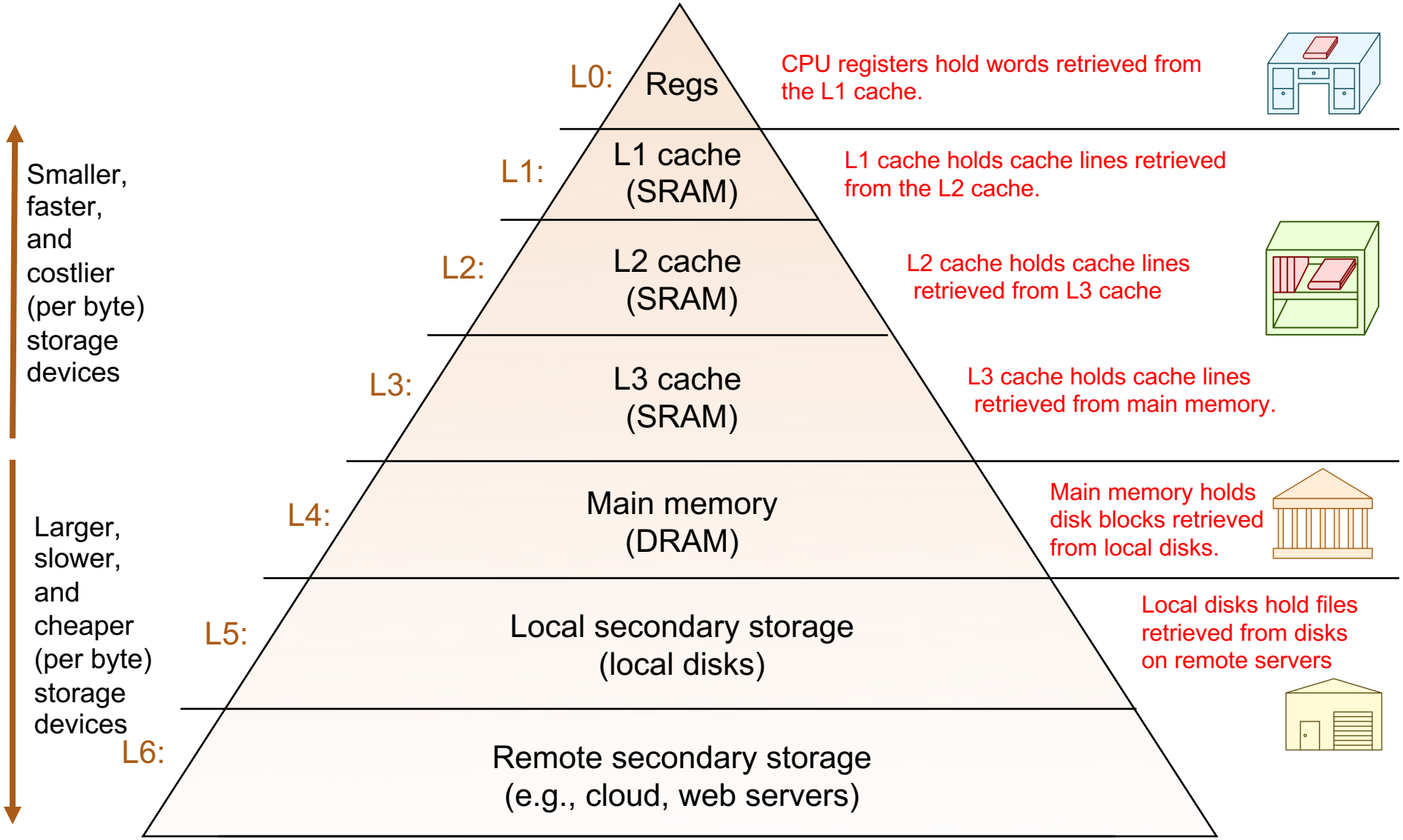
Cache

Write-Through, Write-Allocate, and Performance

Memory Hierarchy



Caching Vocabulary

- Size: the total number of bytes that can be stored in the cache

- Cache Hit: the desired value is in the cache and quickly returned
- Hite rate: the fraction of accesses that are hits
- Hit time: the time to process a hit

- Cache Miss: the desired value is not in the cache and must be fetched elsewhere
- Miss rate: the fraction of accesses that are misses
- Miss penalty: the additional time to process a miss

- Average access time: hit-time + miss-rate * miss-penalty

Caching and Writes

What to do on a write-hit?

- Write-through: write immediately to memory
- Write-back: defer write to memory until replacement of line (requires use of a dirty bit to know if the cache values are different than the memory values)

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

Write-through: write immediately to memory
No-write-allocate: writes straight to memory, does not load into cache

Practice Write-through + No-write-allocate

Memory

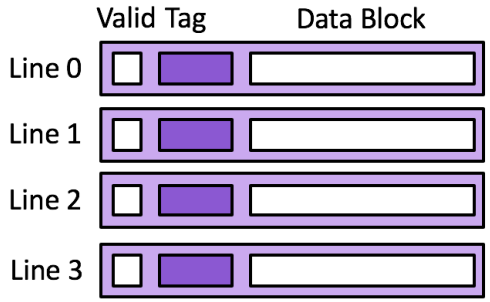
0x24	22
0x20	21
0x1c	20
0x18	19
0x14	18
0x10	17

How many bits for offset?

How many bits for index?

How many bits for tag?

Cache



Assume direct-mapped and 4-byte data blocks

Binary	Access	tag	idx	off	h/m
0001 0000	rd 0x10				
0001 0000	wr 0x10,8				
0010 0100	wr 0x24,9				
0010 0100	rd 0x24				
0010 0000	rd 0x20				

Line 0			Line 1			Line 2			Line 3			W
0	0	47	0	1	47	0	2	47	0	3	47	

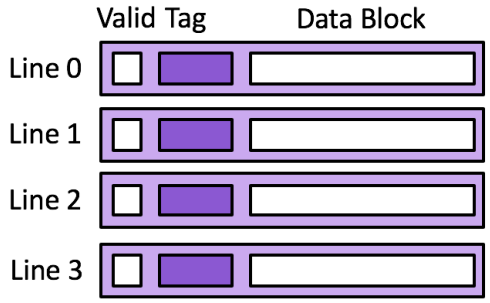
Write-through: write immediately to memory
No-write-allocate: writes straight to memory, does not load into cache

Practice Write-through + No-write-allocate

Memory

0x24	22
0x20	21
0x1c	20
0x18	19
0x14	18
0x10	17

Cache



Assume direct-mapped and 4-byte data blocks

Binary	Access	tag	idx	off	h/m
0001 0000	rd 0x10	0001	00	00	m
0001 0000	wr 0x10,8	0001	00	00	h
0010 0100	wr 0x24,9				
0010 0100	rd 0x24				
0010 0000	rd 0x20				

Line 0			Line 1			Line 2			Line 3			W
0	0	47	0	1	47	0	2	47	0	3	47	
1	1	17										N
1	1	8										Y

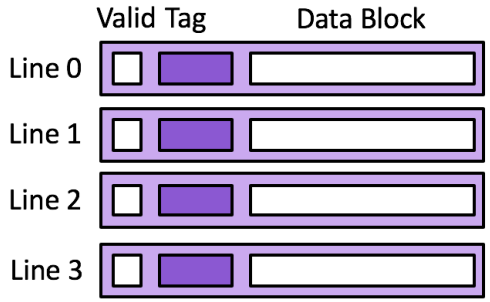
Write-through: write immediately to memory
No-write-allocate: writes straight to memory, does not load into cache

Practice Write-through + No-write-allocate

Memory

0x24	22
0x20	21
0x1c	20
0x18	19
0x14	18
0x10	08

Cache



Assume direct-mapped and 4-byte data blocks

Binary	Access	tag	idx	off	h/m
0001 0000	rd 0x10	0001	00	00	m
0001 0000	wr 0x10,8	0001	00	00	h
0010 0100	wr 0x24,9	0010	01	00	m
0010 0100	rd 0x24				
0010 0000	rd 0x20				

Line 0			Line 1			Line 2			Line 3			W
0	0	47	0	1	47	0	2	47	0	3	47	
1	1	17										N
1	1	8										Y
												Y

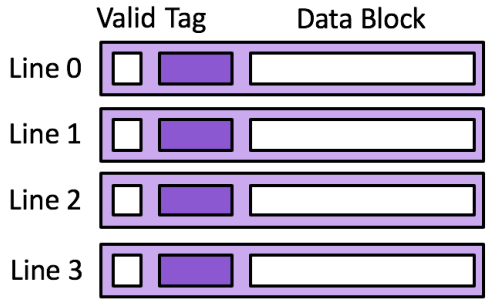
Write-through: write immediately to memory
No-write-allocate: writes straight to memory, does not load into cache

Practice Write-through + No-write-allocate

Memory

0x24	09
0x20	21
0x1c	20
0x18	19
0x14	18
0x10	08

Cache



Assume direct-mapped and 4-byte data blocks

Binary	Access	tag	idx	off	h/m
0001 0000	rd 0x10	0001	00	00	m
0001 0000	wr 0x10,8	0001	00	00	h
0010 0100	wr 0x24,9	0010	01	00	m
0010 0100	rd 0x24	0010	01	00	m
0010 0000	rd 0x20	0010	00	00	m

Line 0			Line 1			Line 2			Line 3			W
0	0	47	0	1	47	0	2	47	0	3	47	
1	1	17										N
1	1	8										Y
												Y
			1	2	9							N
1	2	21										N

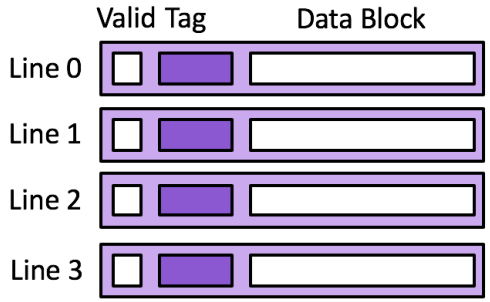
Write-back: defer write to memory until replacement of line
Write-allocate: load into cache, update line in cache

Practice Write-back + Write-allocate

Memory

0x24	22
0x20	21
0x1c	20
0x18	19
0x14	18
0x10	17

Cache



Assume direct-mapped and 4-byte data blocks

Binary	Access	tag	idx	off	h/m
0001 0000	rd 0x10				
0001 0000	wr 0x10,8				
0010 0100	wr 0x24,9				
0010 0100	rd 0x24				
0010 0000	rd 0x20				

Line 0			Line 1			Line 2			Line 3			W
0	0	47	0	1	47	0	2	47	0	3	47	

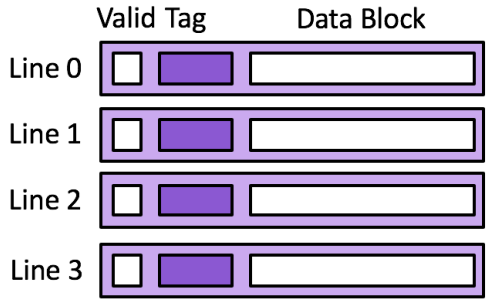
Write-back: defer write to memory until replacement of line
Write-allocate: load into cache, update line in cache

Practice Write-back + Write-allocate

Memory

0x24	22
0x20	21
0x1c	20
0x18	19
0x14	18
0x10	17

Cache



Assume direct-mapped and 4-byte data blocks

Binary	Access	tag	idx	off	h/m
0001 0000	rd 0x10	0001	00	00	m
0001 0000	wr 0x10,8	0001	00	00	h
0010 0100	wr 0x24,9	0010	01	00	m
0010 0100	rd 0x24	0010	01	00	h
0010 0000	rd 0x20	0010	00	00	m

Line 0			Line 1			Line 2			Line 3			W
0	0	47	0	1	47	0	2	47	0	3	47	
1	1	17										N
1	1	8										N
			1	2	9							N
												N
1	2	21										Y

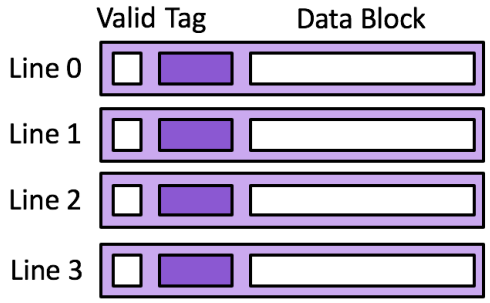
Write-back: defer write to memory until replacement of line
Write-allocate: load into cache, update line in cache

Practice Write-back + Write-allocate

Memory

0x24	22
0x20	21
0x1c	20
0x18	19
0x14	18
0x10	8

Cache



Assume direct-mapped and 4-byte data blocks

Binary	Access	tag	idx	off	h/m
0001 0000	rd 0x10	0001	00	00	m
0001 0000	wr 0x10,8	0001	00	00	h
0010 0100	wr 0x24,9	0010	01	00	m
0010 0100	rd 0x24	0010	01	00	h
0010 0000	rd 0x20	0010	00	00	m

Line 0			Line 1			Line 2			Line 3			W
0	0	47	0	1	47	0	2	47	0	3	47	
1	1	17										N
1	1	8										N
			1	2	9							N
												N
1	2	21										Y

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 memory_mountain(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));
}
```

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 memory_mountain(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 `memory_mountain()` with many combinations of `elems` and `stride`.

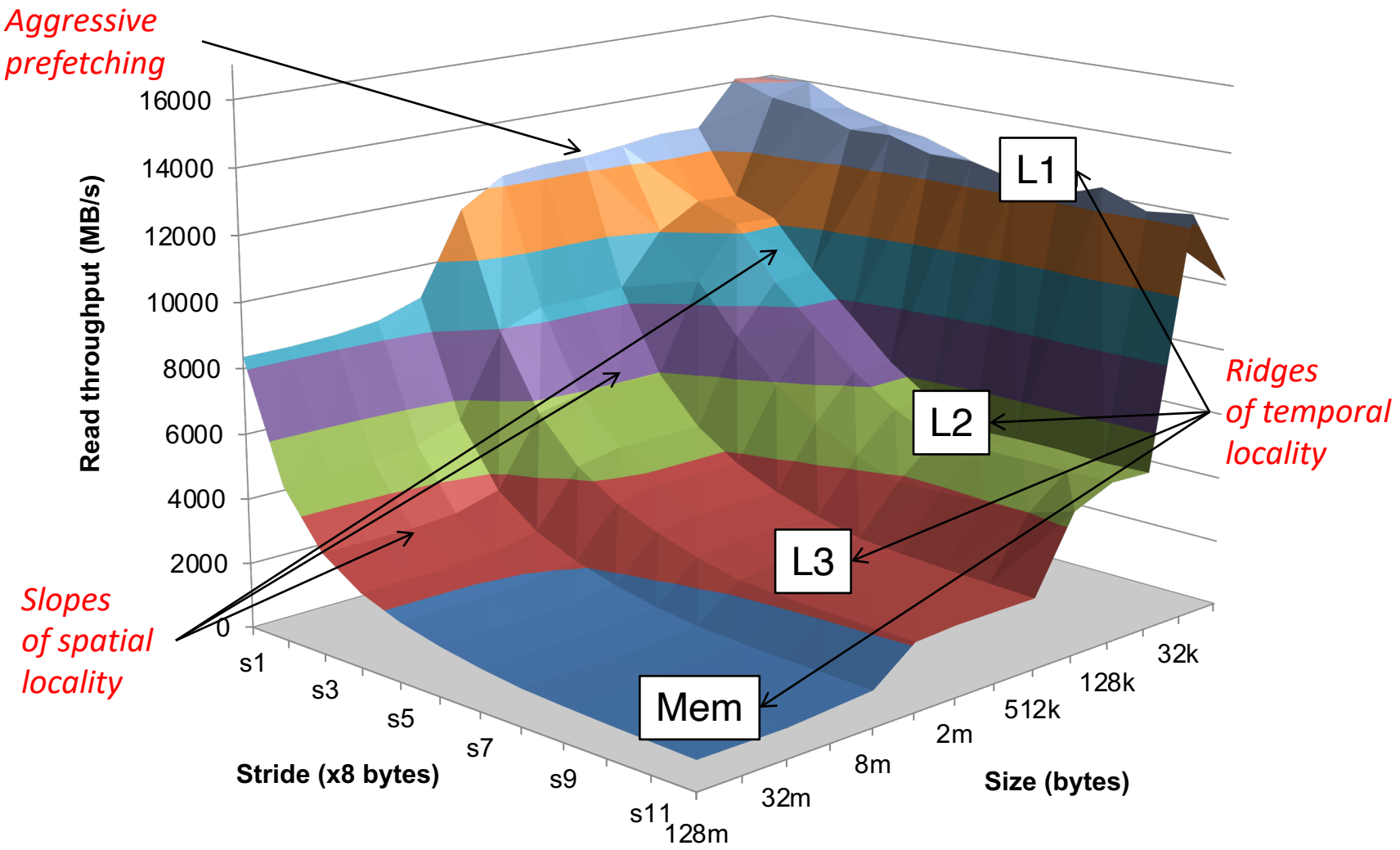
For each `elems` and `stride` :

1. Call `memory_mountain()` once to warm up the caches.

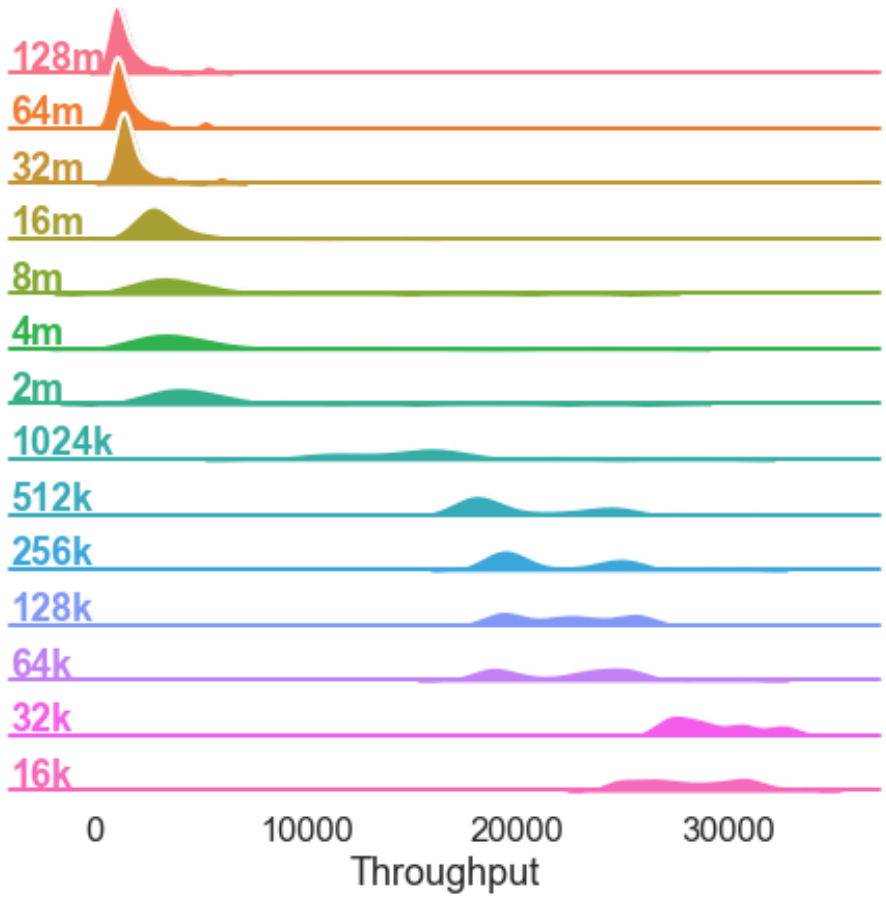
2. Call `memory_mountain()` again and measure the read throughput (MB/s)

The Memory Mountain

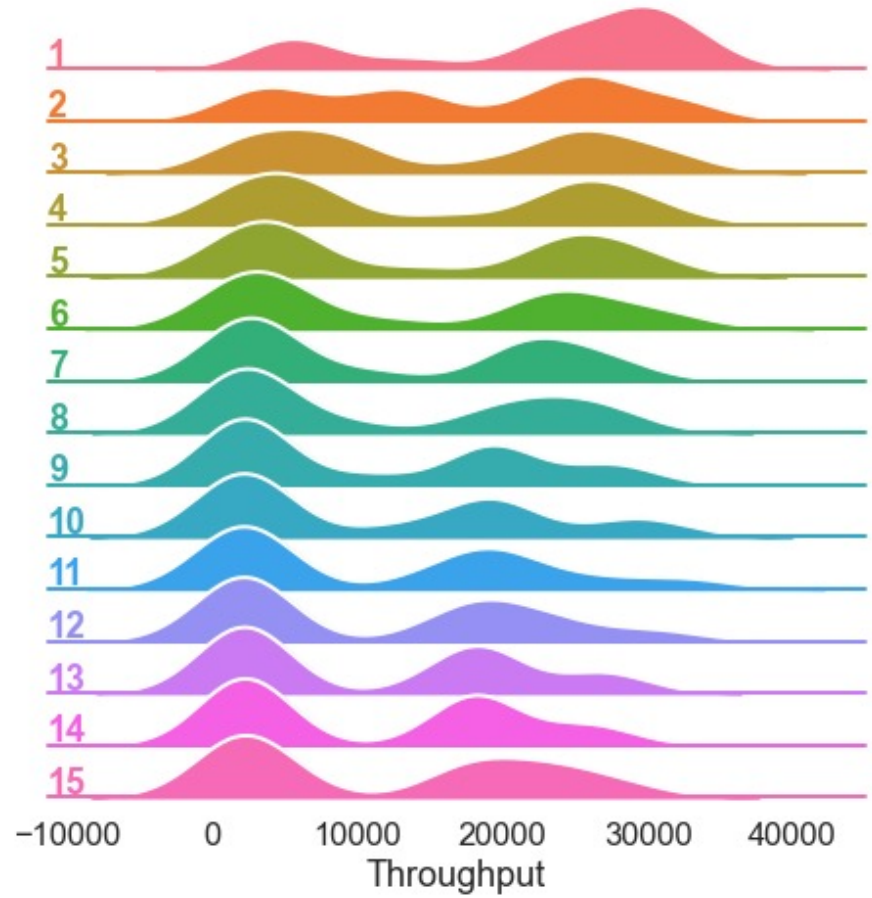
Core i7 Haswell
2.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size



Size


128m
64m
32m
16m
8m
4m
2m
1024k
512k
256k
128k
64k
32k
16k
0
10000
20000
30000
Throughput

Stride


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
−10000
0
10000
20000
30000
40000
Throughput

Practice with Locality

Which of the following functions is better in terms of locality with respect to array src?

```
void copyij(int src[2048][2048],
            int dst[2048][2048])
{
  int i,j;
  for (i = 0; i < 2048; i++)
    for (j = 0; j < 2048; j++)
      dst[i][j] = src[i][j];
}
```

```
void copyji(int src[2048][2048],
            int dst[2048][2048])
{
  int i,j;
  for (j = 0; j < 2048; j++)
    for (i = 0; i < 2048; i++)
      dst[i][j] = src[i][j];
}
```

Practice with Locality

Which of the following functions is better in terms of locality with respect to array src?

```
void copyij(int src[2048][2048],
            int dst[2048][2048])
{
  int i,j;
  for (i = 0; i < 2048; i++)
    for (j = 0; j < 2048; j++)
      dst[i][j] = src[i][j];
}
```

4.3ms

```
void copyji(int src[2048][2048],
            int dst[2048][2048])
{
  int i,j;
  for (j = 0; j < 2048; j++)
    for (i = 0; i < 2048; i++)
      dst[i][j] = src[i][j];
}
```

2.0 GHz Intel Core i7 Haswell

81.8ms

Writing Cache-Friendly Code

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

- Minimize the misses in inner loops
  - Repeated references to variables are good (**temporal locality**)
  - Stride-1 reference patterns are good (**spatial locality**)

Example: Matrix Multiplication

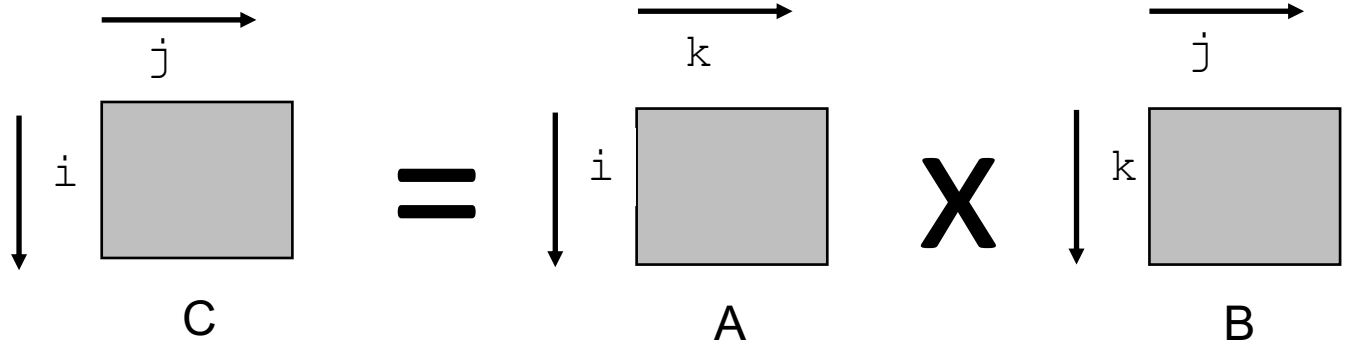
- Multiply N x N matrices
- Matrix elements are doubles (8 bytes)
- $O(N^3)$ total operations
- N reads per source element
- N values summed per destination

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

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



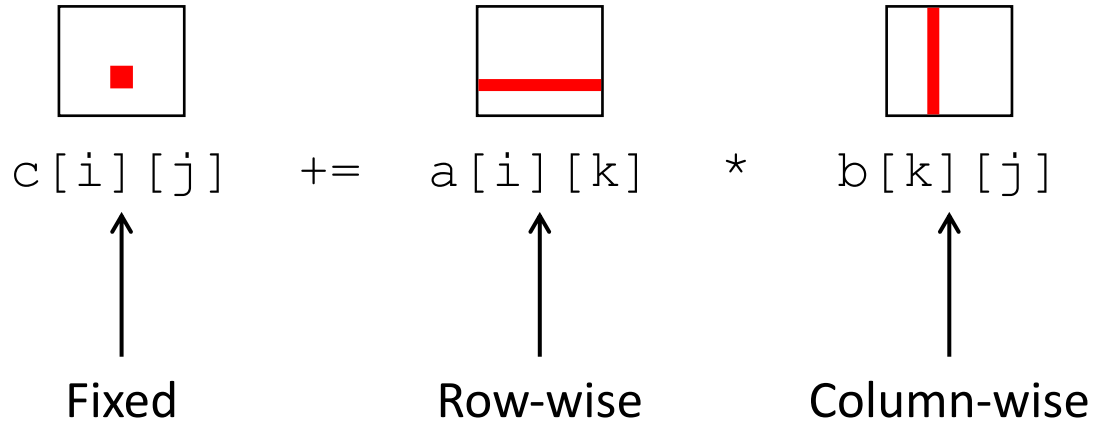
Layout of C Arrays in Memory (review)

- C arrays allocated in row-major order
  - Each row in contiguous memory locations

- Stepping through columns in one row:
  - Accesses successive elements
  - If cache data block "$B > s_{ij}$", then we can exploit spatial locality
    - miss rate = $\frac{s_{ij}}{B}$
- Stepping through rows in one column:
  - Accesses distant elements
  - No spatial locality!
    - miss rate = 1 (i.e., 100%)

Matrix Multiplication: ijk

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

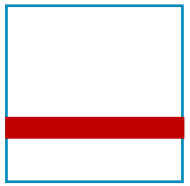


Misses per inner loop iteration:

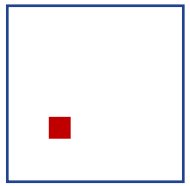
c	a	b
0.0	0.25	1.0

Matrix Multiplication

```
for (k = 0; k < n; k++)
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      c[i][j] += a[i][k] * b[k][j];
```



c[i][j]



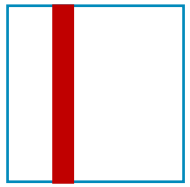
a[i][k]

b[k][j]

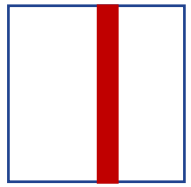
Misses per inner loop iteration

c	a	b
0.25	0.0	0.25

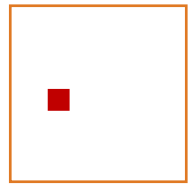
```
for (j = 0; j < n; j++)
  for (k = 0; k < n; k++)
    for (i = 0; i < n; i++)
      c[i][j] += a[i][k] * b[k][j];
```



c[i][j]



a[i][k]

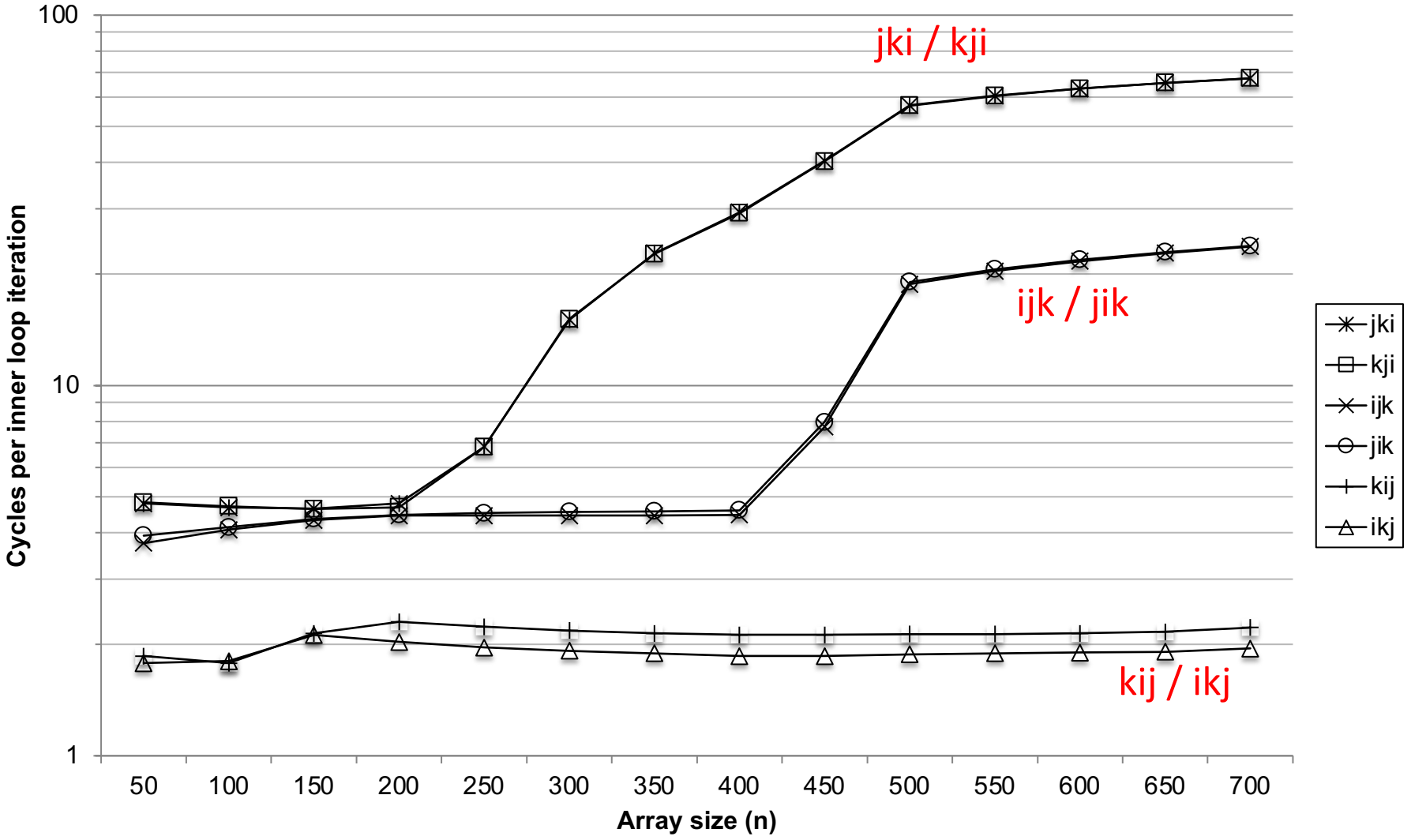


b[k][j]

Misses per inner loop iteration

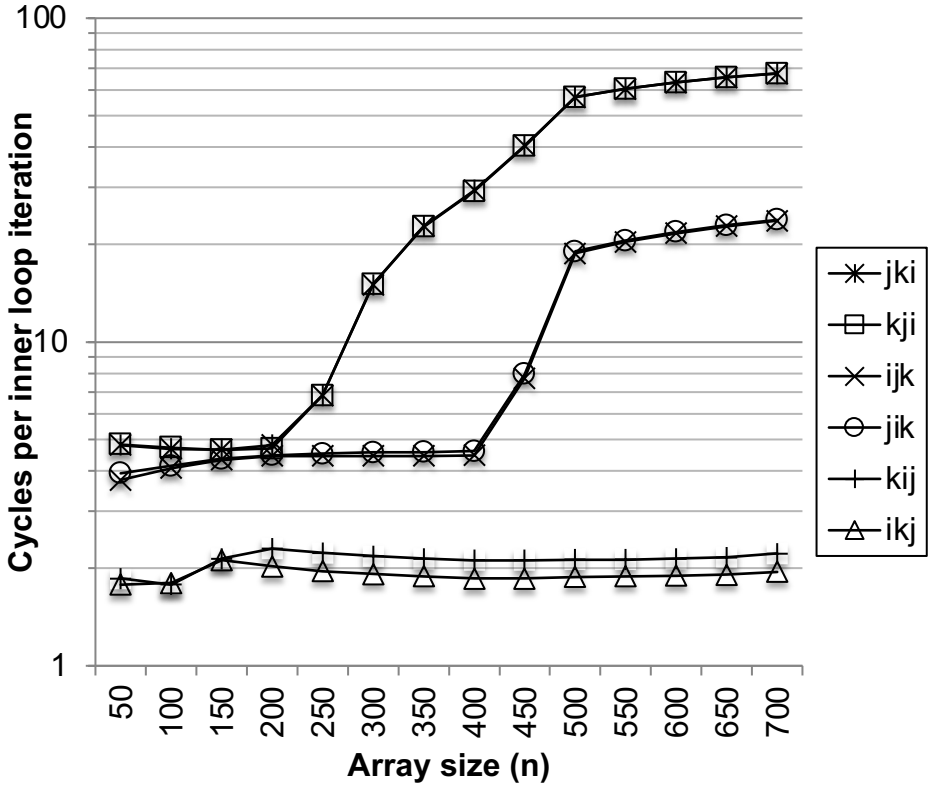
c	a	b
1.0	1.0	0.0

Core i7 Matrix Multiply Performance



Matrix Multiply Performance

Core i7



Pentium III Xeon