

# Caching and Performance

Kai Shen

1

## 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 caches (e.g., L1, L2, and L3), then in main memory.
- Typical system structure:

2

## Cache Set Association

$E = 2^n$  lines per set

set

$S = 2^s$  sets

- E-way set-associative cache.
  - tradeoff on E
- Special cases:
  - Direct-mapped: One line per set.
  - Fully-mapped: One set for the whole cache.

3

## Cache Read

$E = 2^n$  lines per set

set

$S = 2^s$  sets

- 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	line offset

valid bit

$B = 2^b$  bytes per cache line (the data)

4

### Cache Helps Programs with Locality

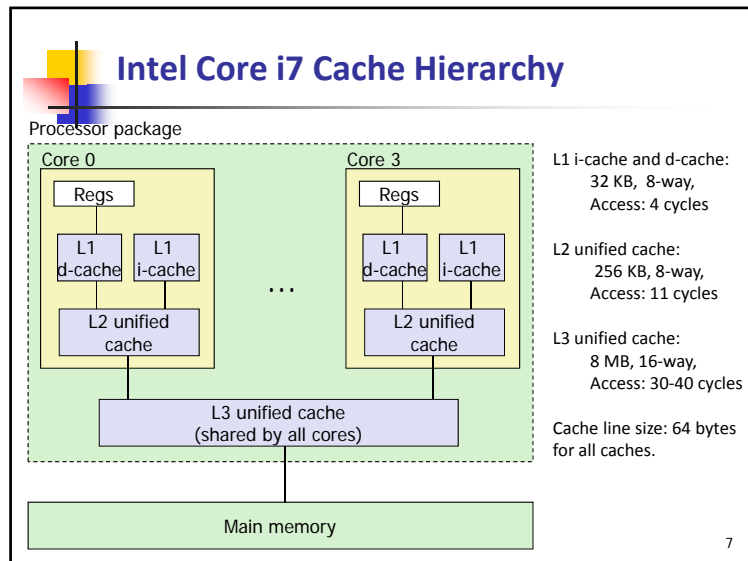
- Cache improves the performance of programs with temporal locality
  - How?
- Cache improves the performance of programs with spatial locality
  - Cacheline-grained memory access (implicit prefetching)
  - Explicit prefetching

5

### What about writes?

- **Write-through**
  - write immediately to memory
- **Write-back**
  - write to cache only (most of time)
  - defer write to memory until replacement of line

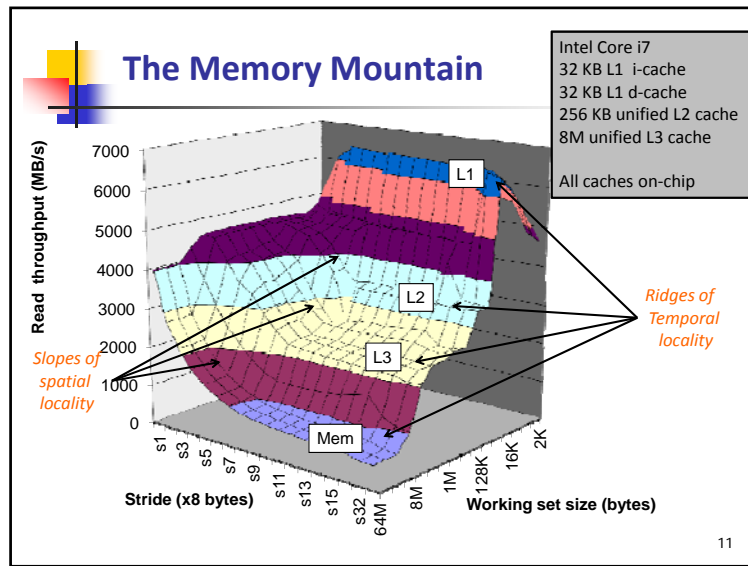
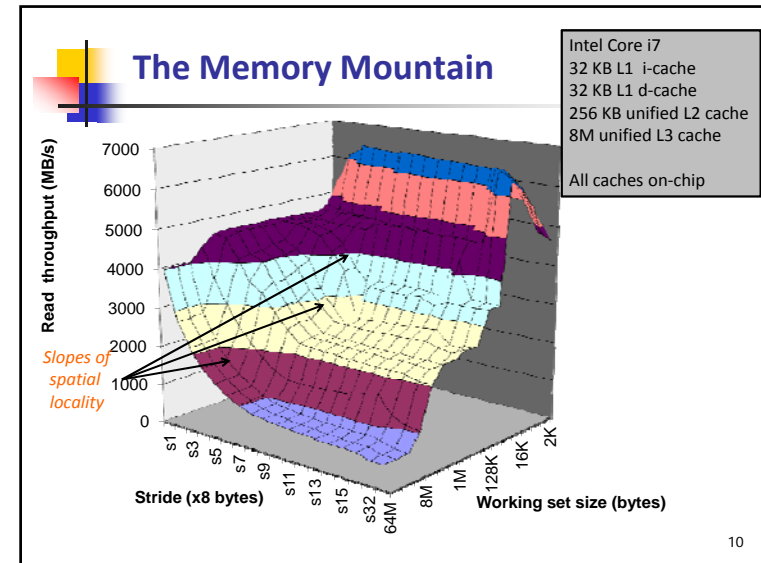
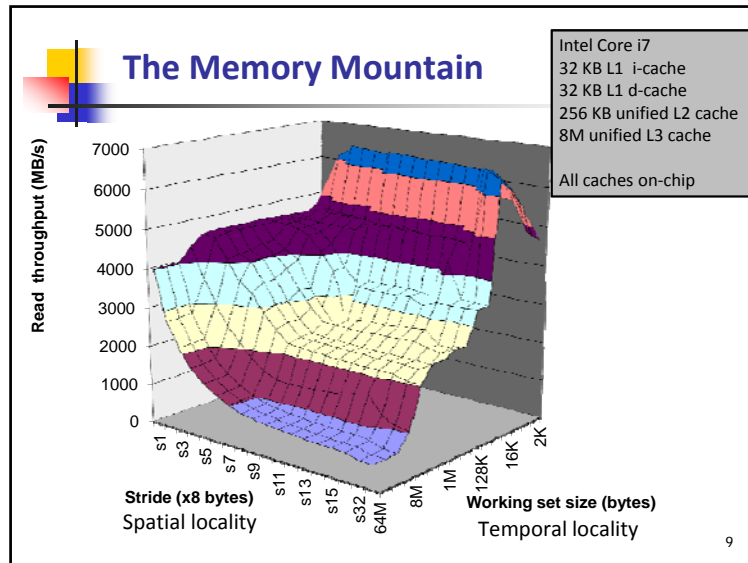
6



### Cache Performance

- Huge difference between a hit and a miss
  - Could be 100x, if just L1 and main memory
- Would you believe 99% hits (fraction of memory accesses in cache) 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:  $1 \text{ cycle} + 0.01 * 100 \text{ cycles} = 2 \text{ cycles}$
- "Miss rate" is more illustrative than "hit rate"

8



- ### 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**)
    - Sequential reference patterns are good (**spatial locality**)
  - Case studies
    - Rearrange loops to improve spatial locality
    - Use blocking to improve temporal locality
- 12

### Matrix Multiplication Performance

```

/* 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*

- Multiply  $n \times n$  matrices
  - $O(n^3)$  total operations
  - $3n^3$  data accesses
- Cache misses dominate the performance
  - Some data accesses go to registers
  - Some memory accesses hit in cache

13

### 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
  - recall that we load a cacheline at a time
  - if cache line size (64) > 8 bytes, exploit spatial locality
    - miss rate = 8 / 64
- 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%)

14

### 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;
  }
}
    
```

Inner loop:

Row-wise    Column-wise    Fixed

Assume matrix dimension ( $n$ ) is so large that a single row can't fit into cache.

Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>	Other implementations?
0.125	1.0	0.0	

15

### 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];
  }
}
    
```

Inner loop:

Fixed    Row-wise    Row-wise

Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.0	0.125	0.125

16

### 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;
  }
}
    
```

Inner loop:

Column-wise      Fixed      Column-wise

Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

17

### 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):

- misses/iter = 1.125

```

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):

- misses/iter = 0.25

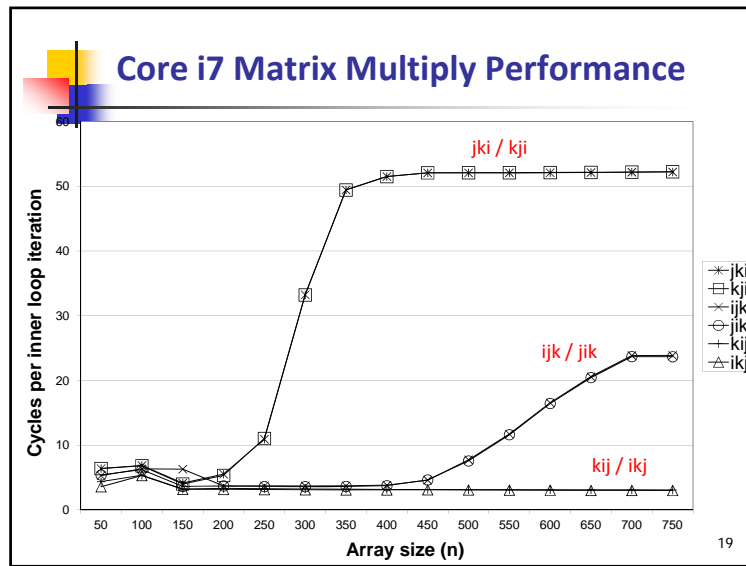
```

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):

- misses/iter = 2.0

18



### Blocked Matrix Multiplication


```

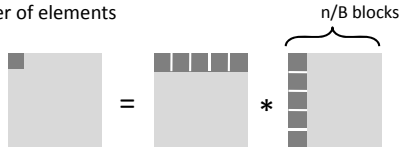
c = (double *) calloc(sizeof(double), n*n);
/* Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
  int i, j, k;
  for (i = 0; i < n; i+=B)
    for (j = 0; j < n; j+=B)
      for (k = 0; k < n; k+=B)
        /* B x B mini matrix multiplications */
        for (i1 = 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];
}
    
```

Block size B x B

20

## Cache Miss Analysis

- Assume:
  - Cache line = 8 doubles (elements)
  - Three blocks  fit into cache
  - B is the block size in number of elements
- First (block) iteration:
  - $B^2/8$  misses for each block
  - $2n/B * B^2/8 = nB/4$  (omitting matrix C)
- Total misses
  - $nB/4 * (n/B)^2 = n^3/(4B)$



21

## Summary

- Blocking:  $n^3/(4B)$
- No blocking:  $n^3 * 0.25$
- Suggest largest possible block size B, but limit it so three blocks fit into cache
  - I once used block size B=25 elements (doubles), what was the cache size?
- Foundation for performance enhancement:
  - 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

22

## Disclaimer

These slides were adapted from the CMU course slides provided along with the textbook of "Computer Systems: A programmer's Perspective" by Bryant and O'Hallaron. The slides are intended for the sole purpose of teaching the computer organization course at the University of Rochester.

23