CS 261
Fall 2016

Mike Lam, Professor

Caching

Topics

- Locality
- Caching
- Performance impact
- General strategies

Locality

- Temporal locality: frequently-accessed items will continue to be accessed in the future
  - Theme: **repetition is common**
- Spatial locality: nearby addresses are more likely to be accessed soon
  - Theme: **sequential access is common**
- Why do we care?
  - *Programs with good locality run faster than programs with poor locality*

Data locality

- Temporal locality: keep often-used values in higher tiers of the memory hierarchy
- Spatial locality: use predictable access patterns
  - Stride-1 reference pattern (sequential access)
  - Stride-k reference pattern (every k elements)
  - Closely related to row-major vs. column-major
  - Allows for prefetching (predicting the next needed element and preloading it)

Instruction locality

- Normal execution exhibits spatial locality
  - Instructions execute **in sequence**
  - Control flow instructions inhibit locality
- Loops exhibit both temporal and spatial locality
  - The body statements execute **repeatedly** (temporal locality) and **in sequence** (spatial locality)
  - Short loops are better

Caching

- A cache is a small, fast memory that acts as a buffer or staging area for a larger, slower memory
  - Fundamental CS system design concept
  - Example: web browser cache
  - Data is transferred between the cache and the slower memory in blocks
  - Slower caches use larger block sizes
  - Cache hit vs. cache miss – did the cache have to transfer a value from the lower level?

Cache misses

- Cache misses require blocks to be replaced or evicted
- Policies:
  - Random replacement
  - Least recently used
  - Least frequently used

Cache misses

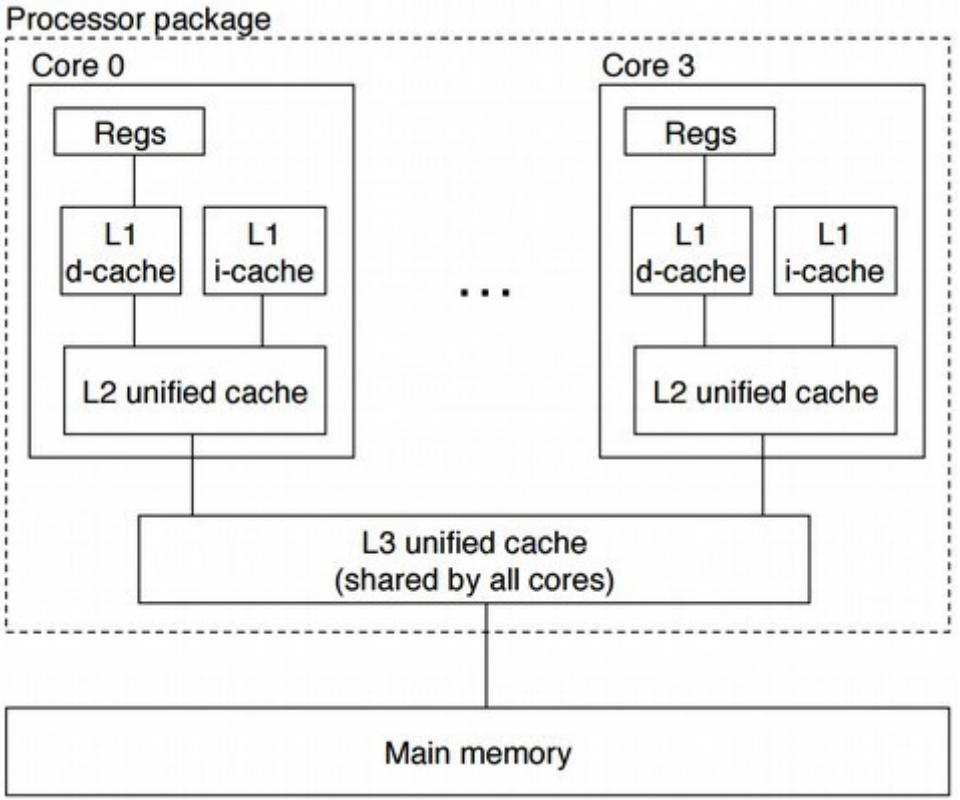
- A cache always begins cold (empty)
  - Every request will be a miss initially
- As the cache loads data, it is warmed up
  - This effect can cause performance measurement variation during experiments if not controlled for
- A working set is a collection of elements needed repeatedly for a particular computation
  - If the working set doesn't fit in cache, this is called a capacity miss

Cache writes

- How should we handle writes to a cached value?
  - Write-through: immediately update to lower level
    - Typically used for higher levels of memory hierarchy
  - Write-back: defer update until eviction
    - Typically used for lower levels of memory hierarchy
- How should we handle write misses?
  - Write-allocate: load then update
    - Typically used for write-back caches
  - No-write-allocate: update without loading
    - Typically used for write-through caches

Cache architecture

- Example: Intel Core i7
- Per-core:
  - Registers
  - L1 d-cache and i-cache
    - Data and instructions
  - L2 unified cache
- Shared:
  - L3 unified cache
  - Main memory

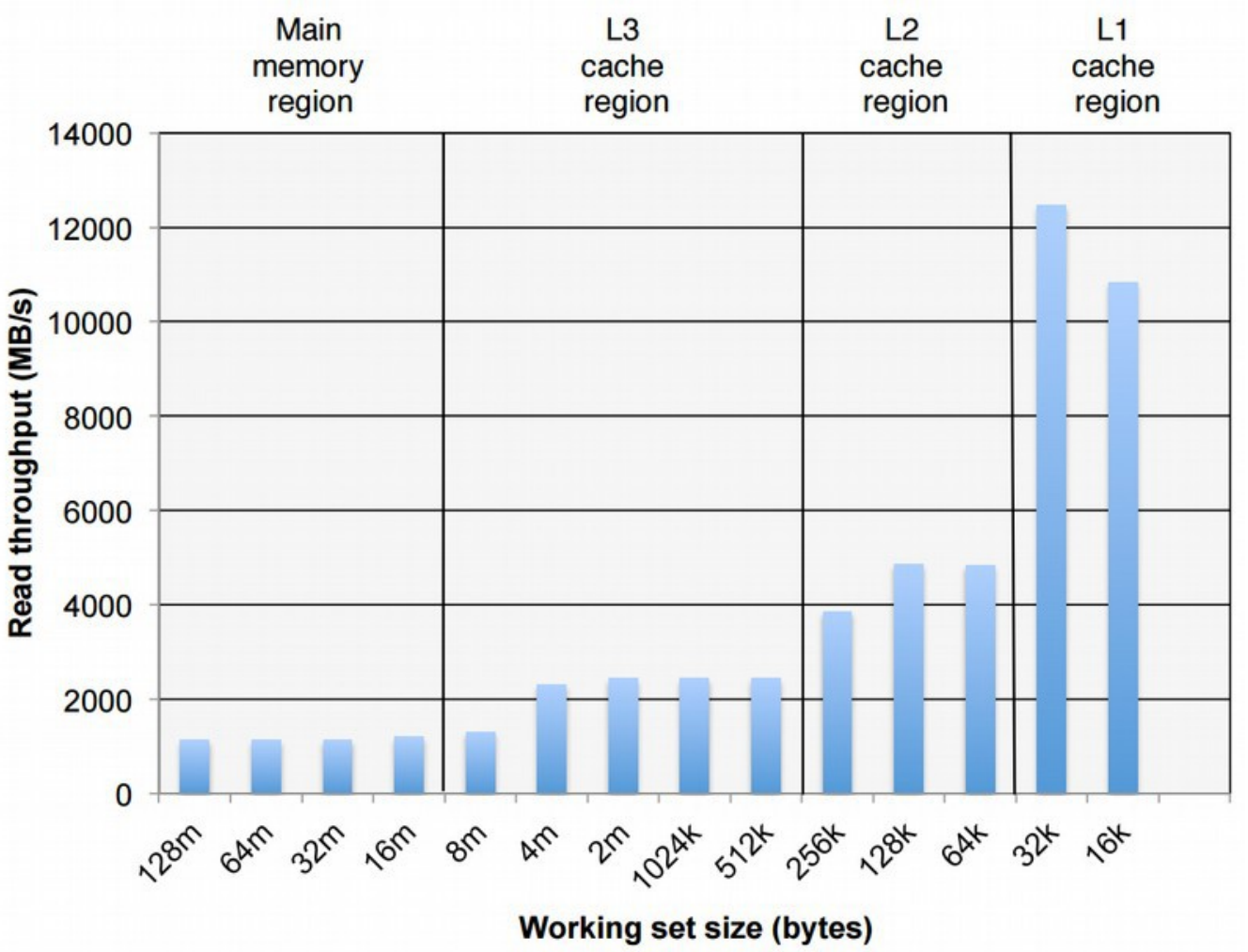


Performance impact

- Metrics
  - Miss rate: # misses / # memory accesses
  - Hit rate: 1 – miss rate
  - Hit time: delay in accessing data for a cache hit
  - Miss penalty: delay in loading data for a cache miss
  - Read throughput (or "bandwidth"): the rate that a program reads data from a memory system
- General observations:
  - Larger cache = higher hit rate but higher hit time
  - Lower miss rates = higher read throughput

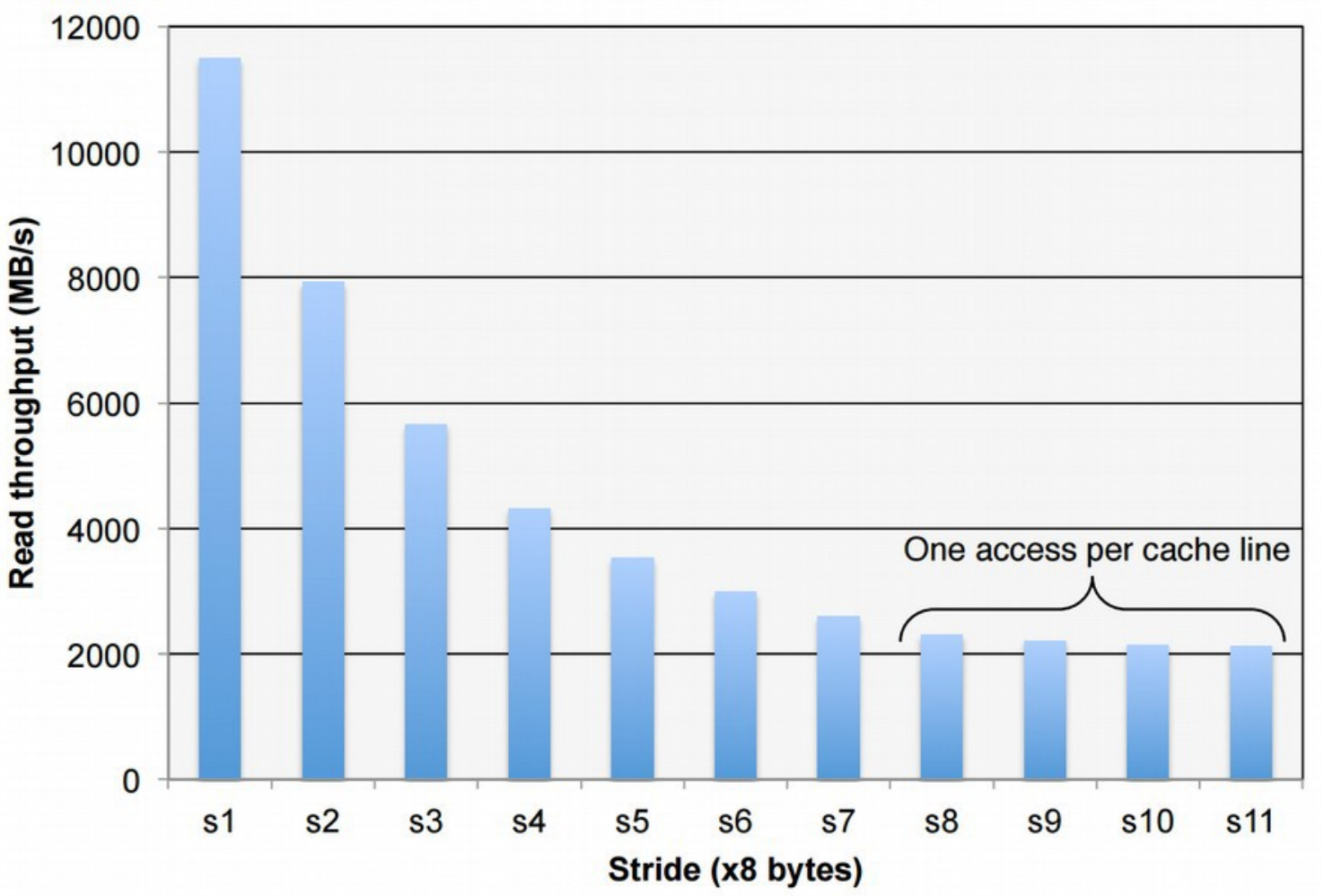
Temporal locality

- Working set size vs. throughput



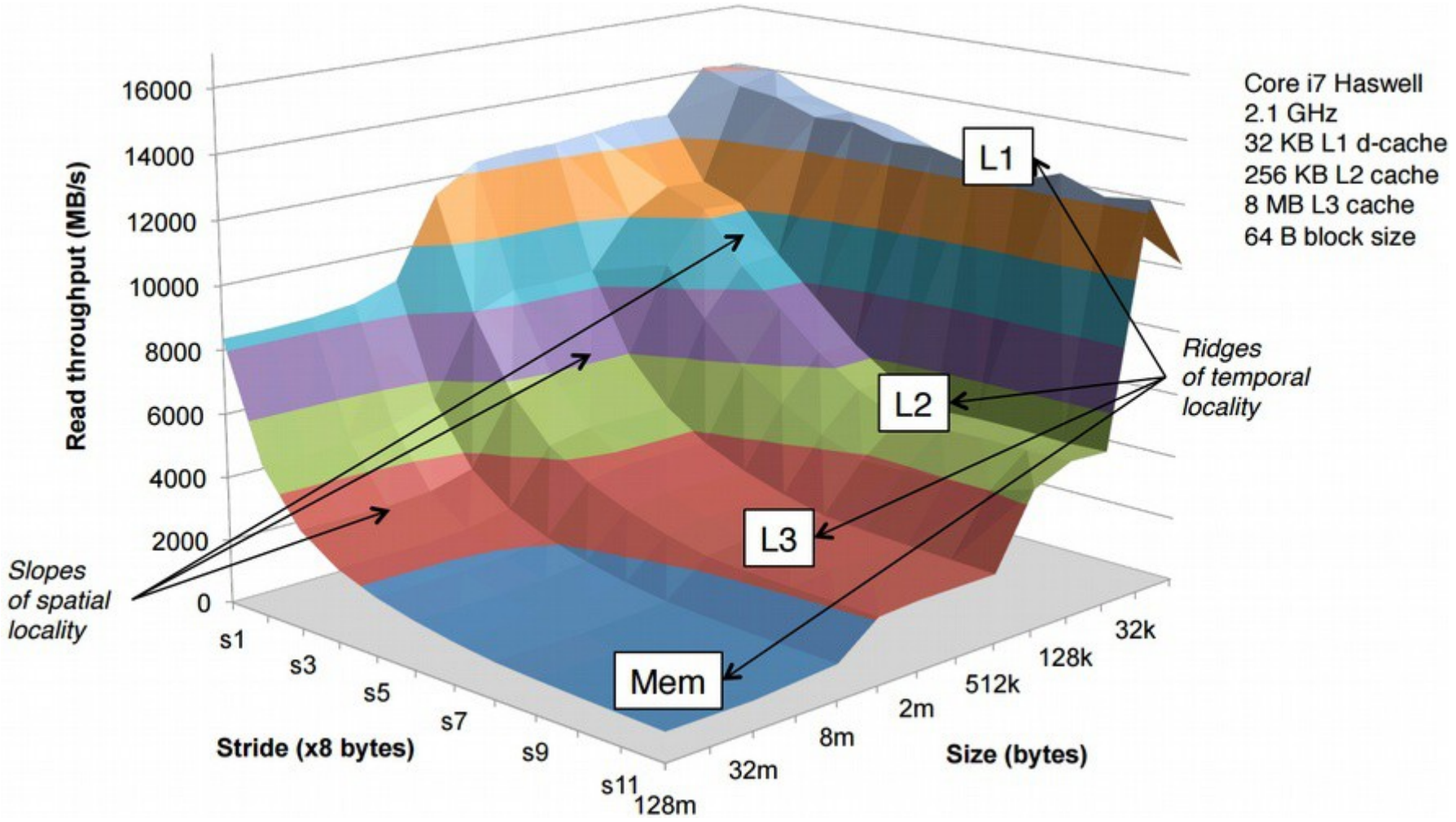
Spatial locality

- Stride vs. throughput



Memory mountain

- Stride and WSS vs. read throughput



Case study: matrix multiply

(a) Version *ijk*

code/mem/matmult/mm.c

```
1   for (i = 0; i < n; i++)
2       for (j = 0; j < n; j++) {
3           sum = 0.0;
4           for (k = 0; k < n; k++)
5               sum += A[i][k]*B[k][j];
6           C[i][j] += sum;
7       }
```

code/mem/matmult/mm.c

(b) Version *jik*

code/mem/matmult/mm.c

```
1   for (j = 0; j < n; j++)
2       for (i = 0; i < n; i++) {
3           sum = 0.0;
4           for (k = 0; k < n; k++)
5               sum += A[i][k]*B[k][j];
6           C[i][j] += sum;
7       }
```

code/mem/matmult/mm.c

(c) Version *jki*

code/mem/matmult/mm.c

```
1   for (j = 0; j < n; j++)
2       for (k = 0; k < n; k++) {
3           r = B[k][j];
4           for (i = 0; i < n; i++)
5               C[i][j] += A[i][k]*r;
6       }
```

code/mem/matmult/mm.c

(d) Version *kji*

code/mem/matmult/mm.c

```
1   for (k = 0; k < n; k++)
2       for (j = 0; j < n; j++) {
3           r = B[k][j];
4           for (i = 0; i < n; i++)
5               C[i][j] += A[i][k]*r;
6       }
```

code/mem/matmult/mm.c

(e) Version *kij*

code/mem/matmult/mm.c

```
1   for (k = 0; k < n; k++)
2       for (i = 0; i < n; i++) {
3           r = A[i][k];
4           for (j = 0; j < n; j++)
5               C[i][j] += r*B[k][j];
6       }
```

code/mem/matmult/mm.c

(f) Version *ikj*

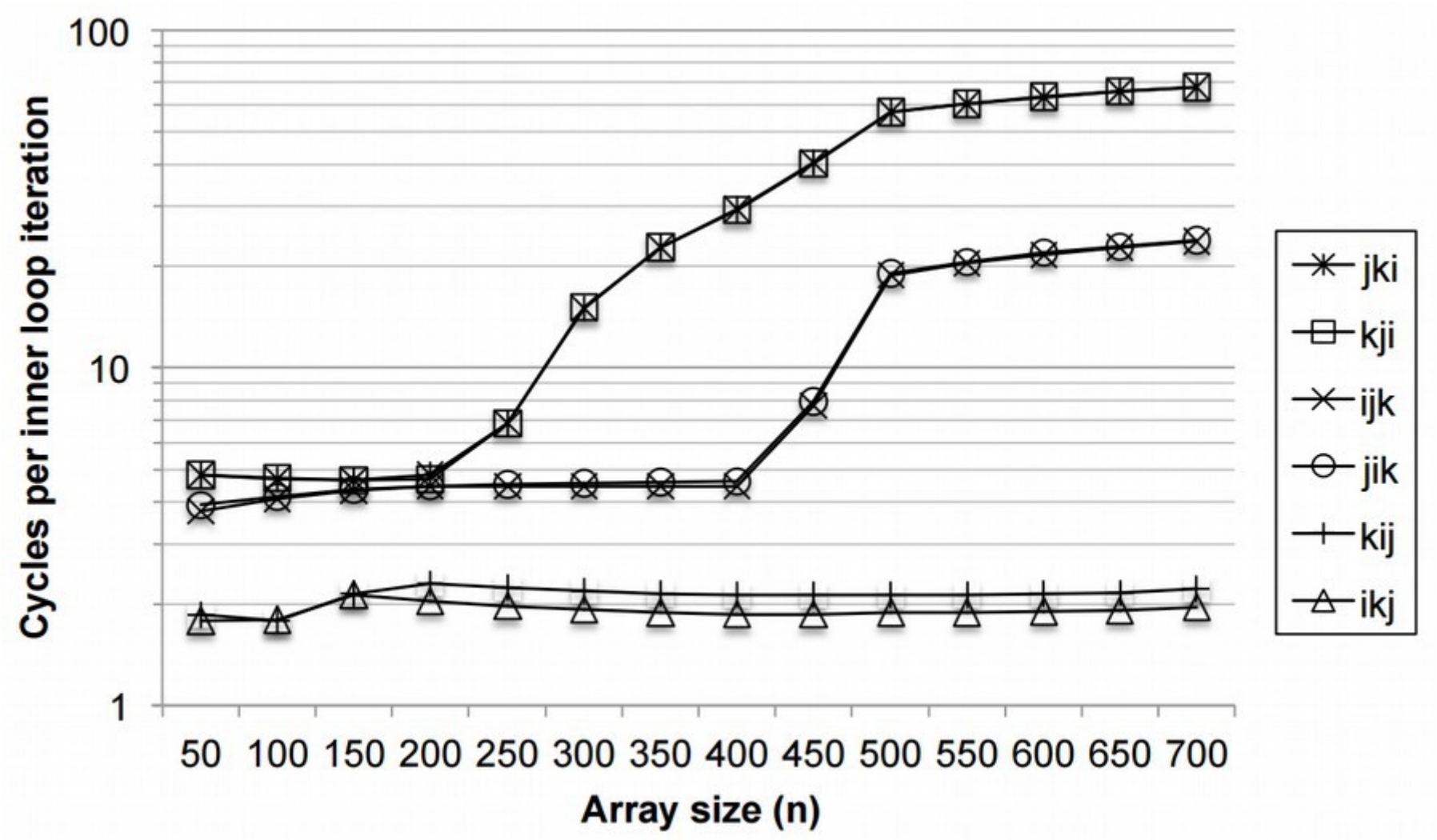
code/mem/matmult/mm.c

```
1   for (i = 0; i < n; i++)
2       for (k = 0; k < n; k++) {
3           r = A[i][k];
4           for (j = 0; j < n; j++)
5               C[i][j] += r*B[k][j];
6       }
```

code/mem/matmult/mm.c

Figure 6.44 **Six versions of matrix multiply.** Each version is uniquely identified by the ordering of its loops.

Case study: matrix multiply



Optimization strategies

- Focus on the common cases
- Focus on the code regions that dominate runtime
- Focus on inner loops and minimize cache misses
- Favor repeated local accesses (temporal locality)
- Favor stride-1 access patterns (spatial locality)

Next time

- Virtual memory: an OS-level memory cache