Memory hierarchies: caches and their impact on the running time

Irene Finocchi

Dept. of Computer and Science

Sapienza University of Rome

Performance impact of caches

A high level example

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

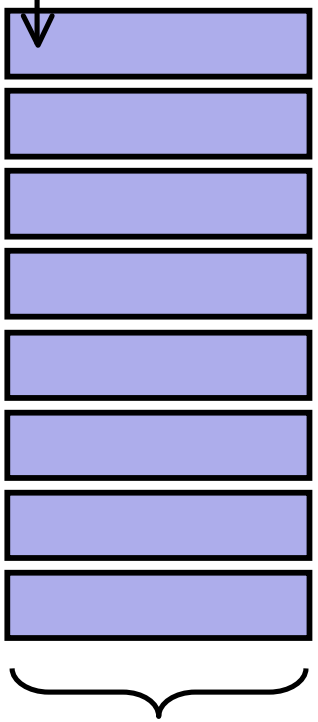
Stride-1 reference pattern

```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; i < 16; i++)
        for (i = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

Stride-16 reference pattern

Ignore the variables sum, i, j

assume: cold (empty) cache, a[0][0] goes here



32 B = 4 doubles

blackboard

Cache performance metrics

- **Miss Rate** = fraction of memory references not found in cache (misses / accesses) = 1 – 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:
    - 1-2 clock cycle for L1
    - 5-20 clock cycles for L2
- **Miss Penalty** = additional time required because of a miss
    - typically 50-200 cycles for main memory (trend: increasing!)

Lets 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**

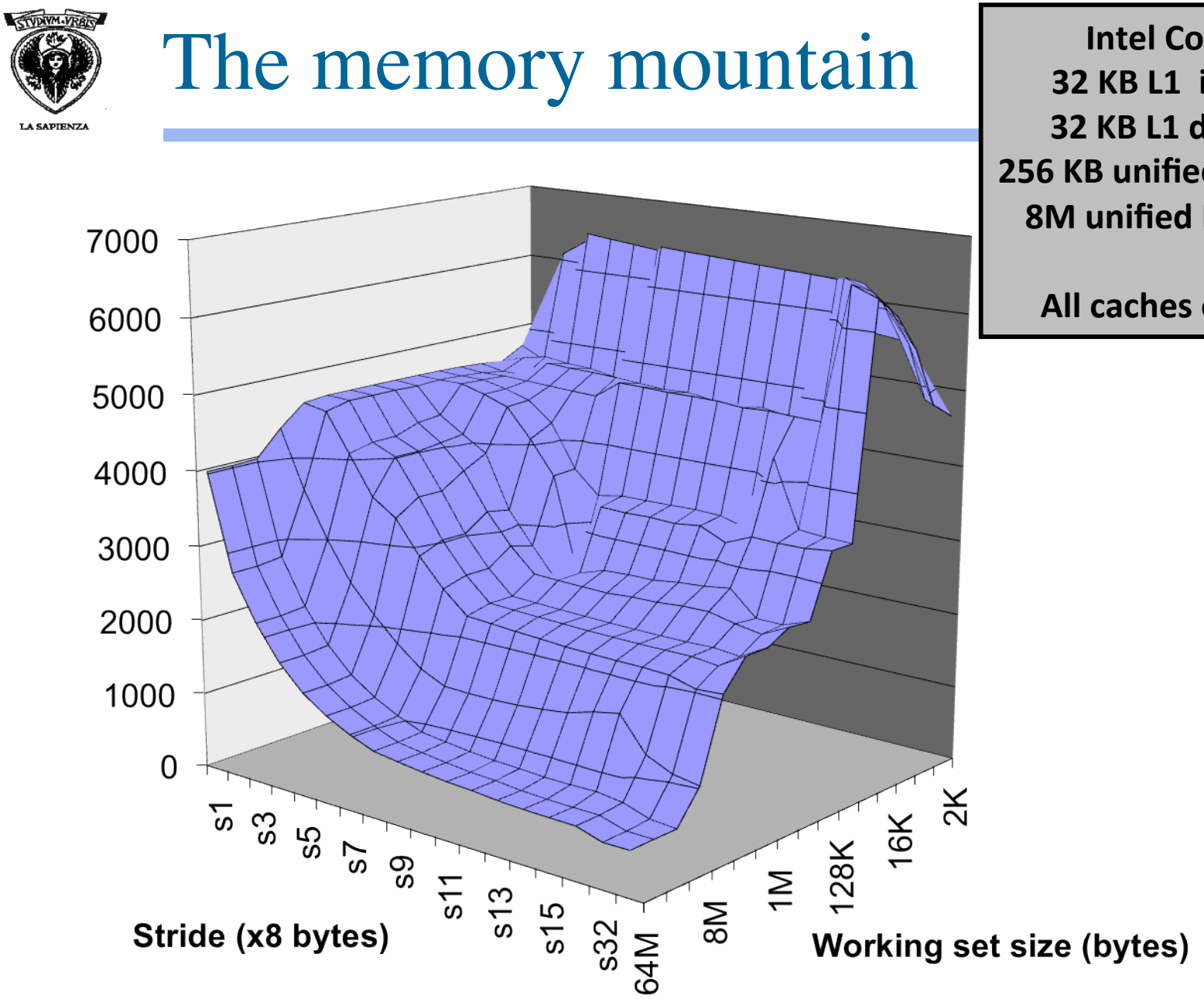
The memory mountain

- 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

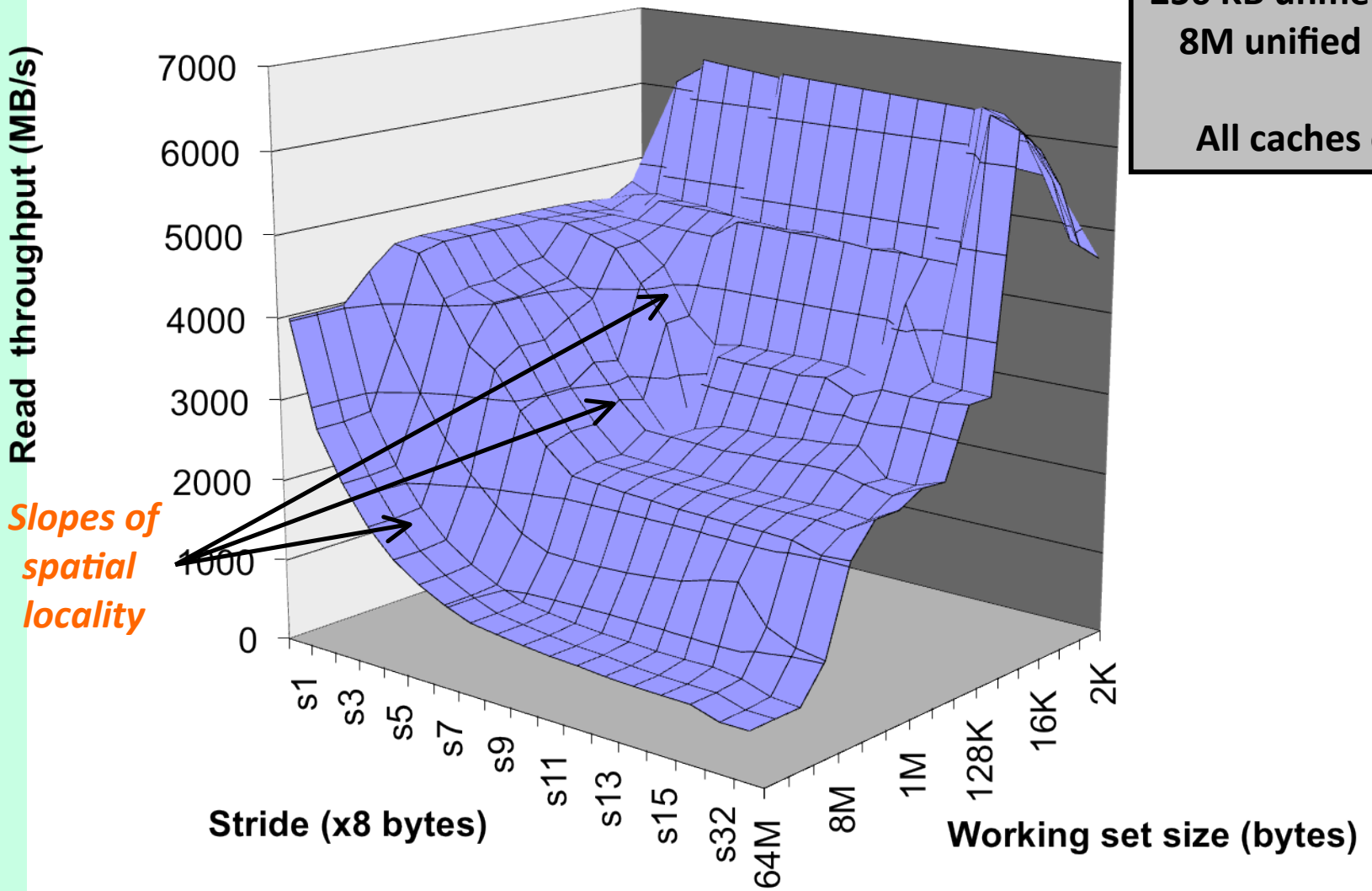
```
/* The test function */
void test(int elems, int stride) {
    int i, result = 0;
    volatile int sink;

    for (i = 0; i < elems; i += stride)
        result += data[i];
    sink = result; /* So compiler doesn't optimize away the loop */
}

/* Run test(elems, stride) and return read throughput (MB/s) */
double run(int size, int stride, double Mhz)
{
    double cycles;
    int elems = size / sizeof(int);

    test(elems, stride);                      /* warm up the cache */
    cycles = fcyc2(test, elems, stride, 0);   /* call test(elems,stride) */
    return (size / stride) / (cycles / Mhz); /* convert cycles to MB/s */
}
```

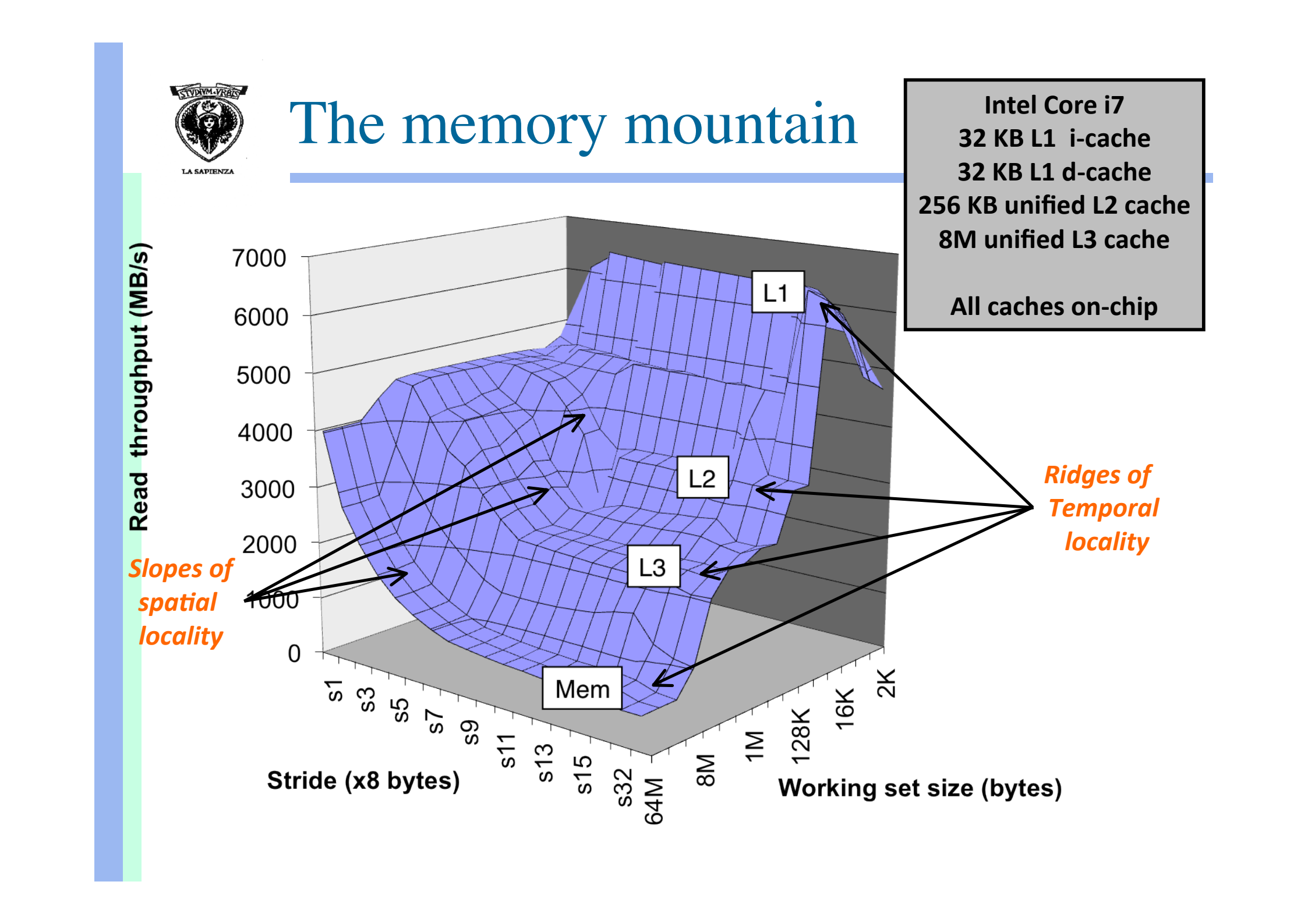

The memory mountain
Intel Core i7
32 KB L1 i-cache
32 KB L1 d-cache
256 KB unified L2 cache
8M unified L3 cache
All caches on-chip
Read throughput (MB/s)
7000
6000
5000
4000
3000
2000
1000
0
s1
s3
s5
s7
s9
s11
s13
s15
s32
Stride (x8 bytes)
64M
8M
1M
128K
16K
2K
Working set size (bytes)

The memory mountain

Intel Core i7
32 KB L1 i-cache
32 KB L1 d-cache
256 KB unified L2 cache
8M unified L3 cache

All caches on-chip



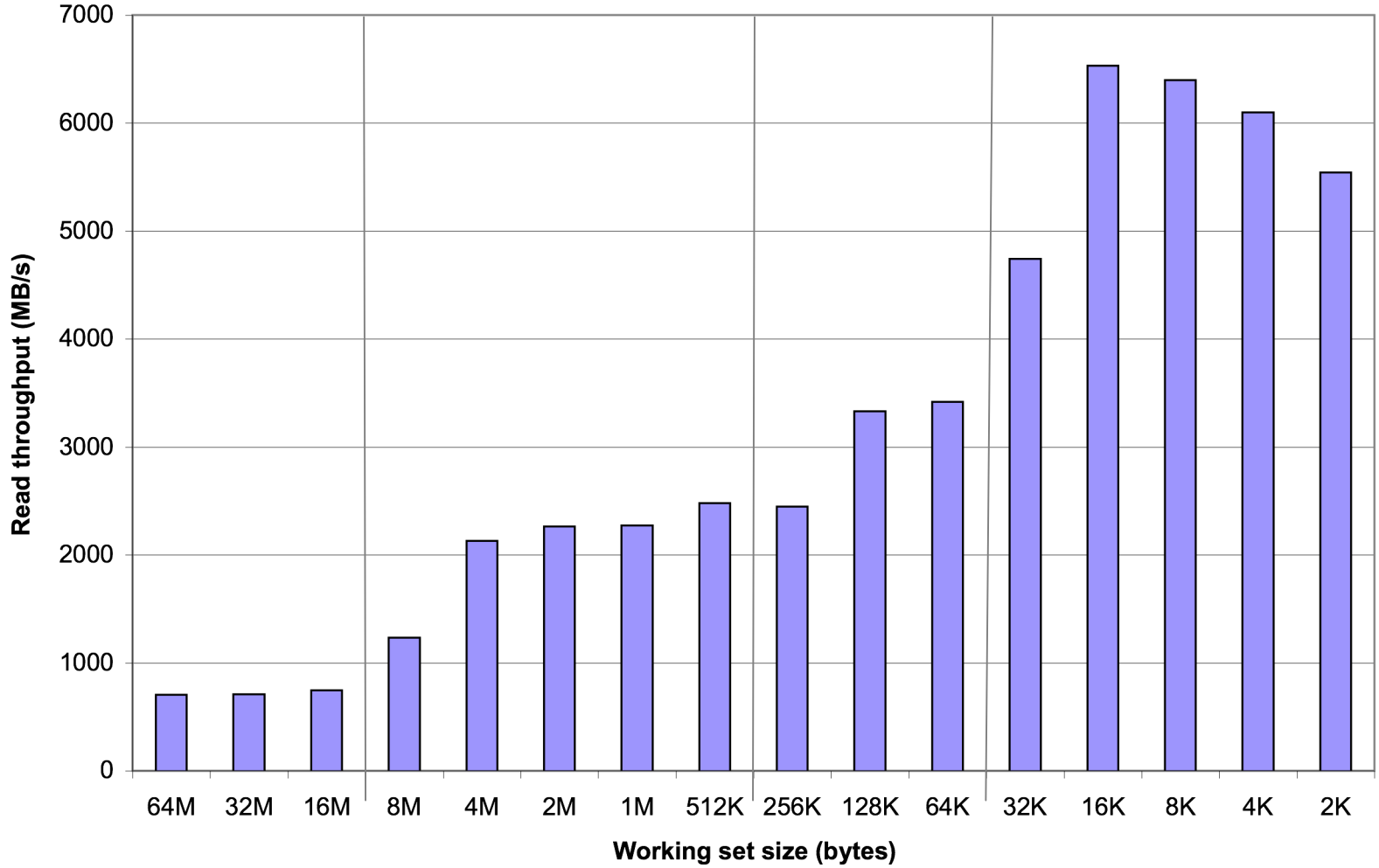

The memory mountain
Intel Core i7
32 KB L1 i-cache
32 KB L1 d-cache
256 KB unified L2 cache
8M unified L3 cache
All caches on-chip
Read throughput (MB/s)
7000
6000
5000
4000
3000
2000
1000
0
L1
L2
L3
Mem
Ridges of Temporal locality
Slopes of spatial locality
s1
s3
s5
s7
s9
s11
s13
s15
s32
Stride (x8 bytes)
64M
8M
1M
128K
16K
2K
Working set size (bytes)

Ridges of temporal locality

Intel Core i7
32 KB L1 i-cache
32 KB L1 d-cache
256 KB unified L2 cache
8M unified L3 cache

All caches on-chip

Stride fixed at 16

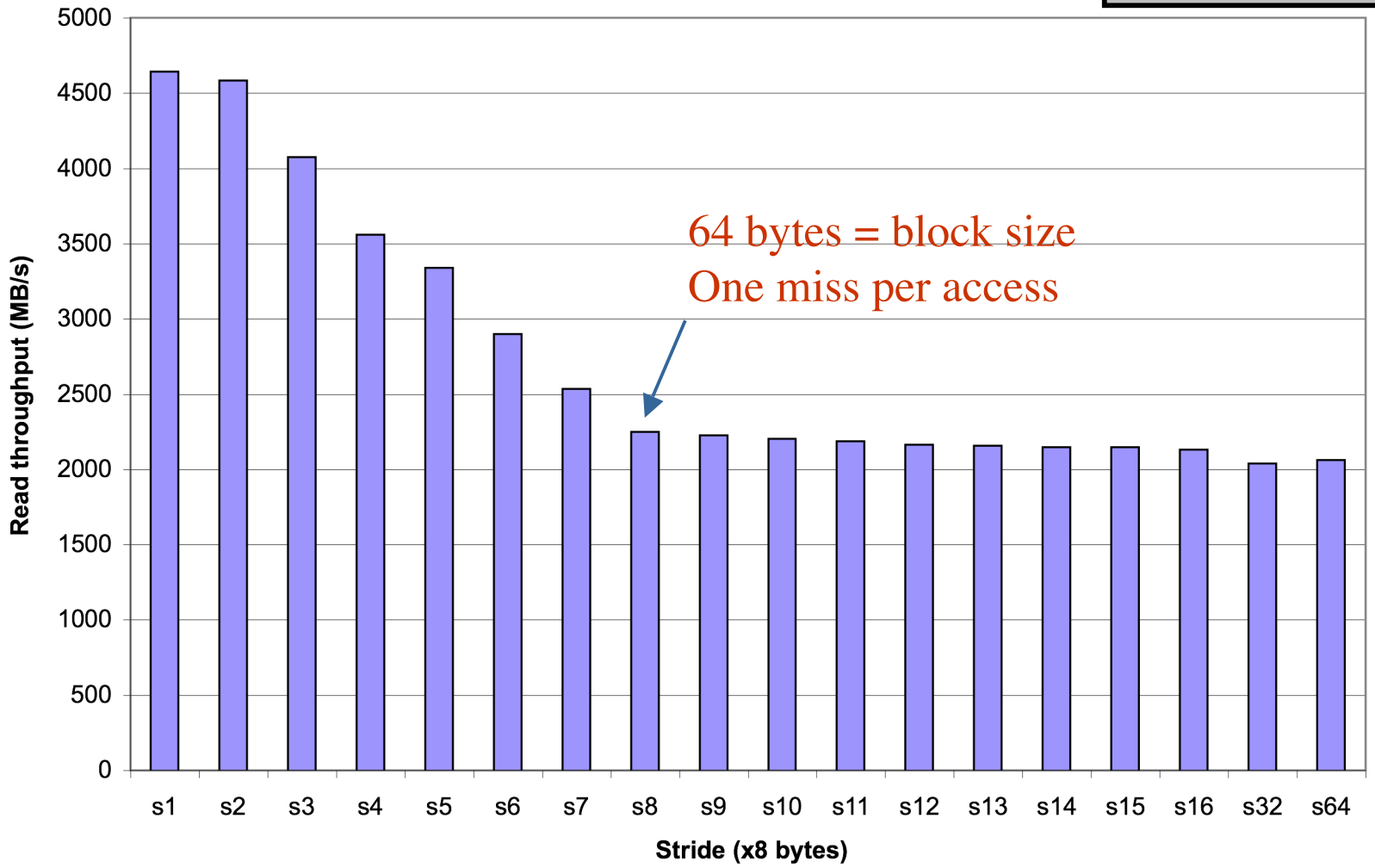


Slope of spatial locality

Intel Core i7
32 KB L1 i-cache
32 KB L1 d-cache
256 KB unified L2 cache
8M unified L3 cache

All caches on-chip

Working set fixed at 4 MB (cut along L3 ridge)



Writing cache-friendly code: ingredients and examples

– Rearranging loops to improve spatial locality

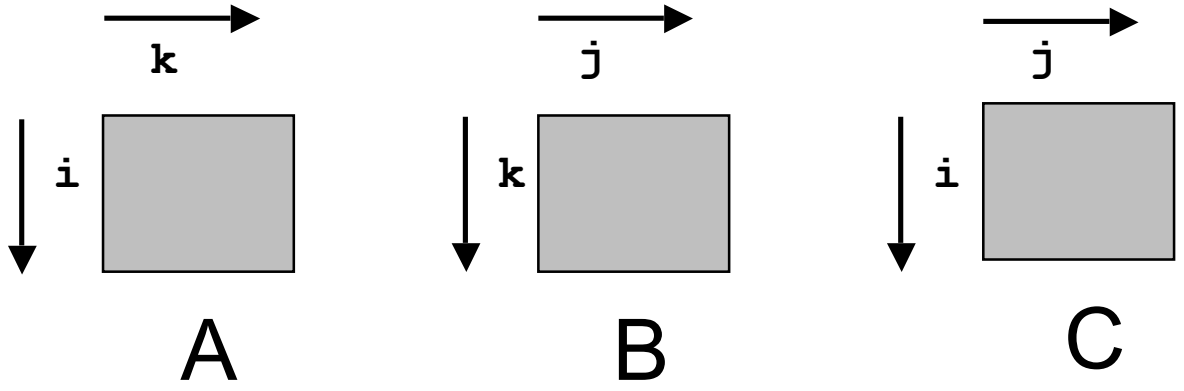
– Using blocking to improve temporal locality

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 (i.e., sequential) reference patterns are good (**spatial locality**)

Miss rate analysis for matrix multiply

- Assume:
  - Line size = 32B (big enough for four 64-bit words)
  - 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



Matrix multiplication example

- Description:
  - Multiply N x N matrices
  - $O(N^3)$ total operations
  - N reads per source element
  - N values summed per destination
    - but may be able to hold in register

Variable sum held in register

```
/* 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;
  }
}
```

Memory layout of arrays in C

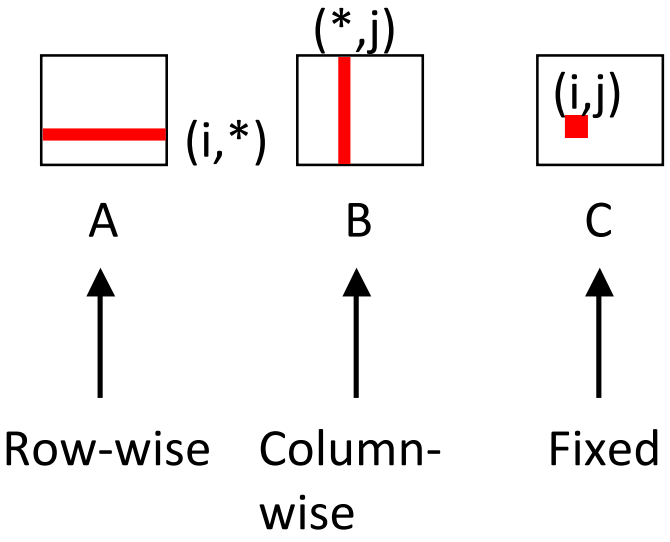
- C arrays allocated in row-major order
  - each row in contiguous memory locations
- Stepping through columns in one row:
  - `for (i = 0; i < N; i++) sum += a[0][i];`
  - accesses successive elements
  - if block size (B) > 4 bytes, exploit spatial locality
    - compulsory miss rate = 4 bytes / B
- Stepping through rows in one column:
  - `for (i = 0; i < n; i++) sum += a[i][0];`
  - accesses distant elements
  - no spatial locality!
    - compulsory miss rate = 1 (i.e. 100%)

Matrix multiplication (ijk)

B=32 bytes, sizeof(double)=8

```
/* 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:



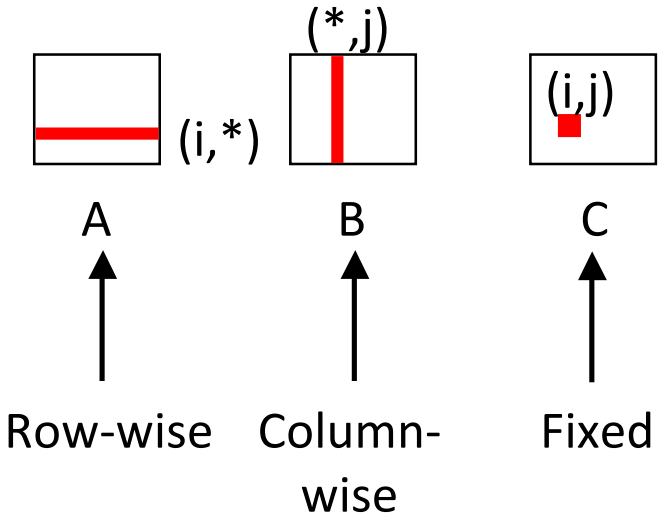
Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0

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
  }
}
```

Inner loop:

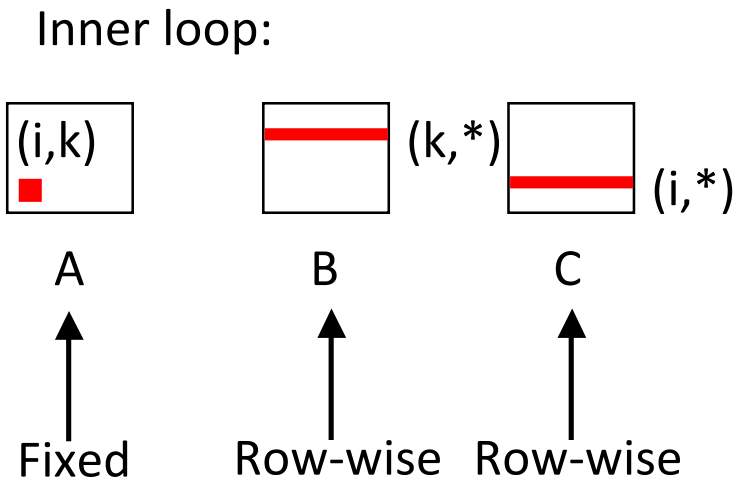


Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0

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];
  }
}
```

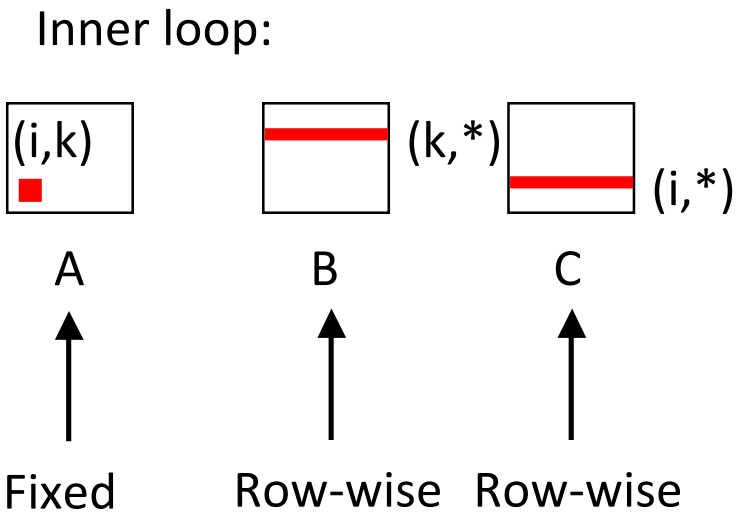


Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

Matrix multiplication (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];
  }
}
```



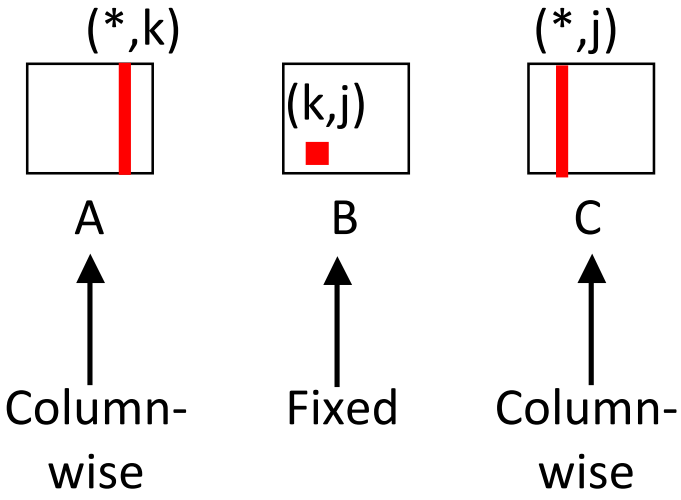
Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

Matrix multiplication (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:

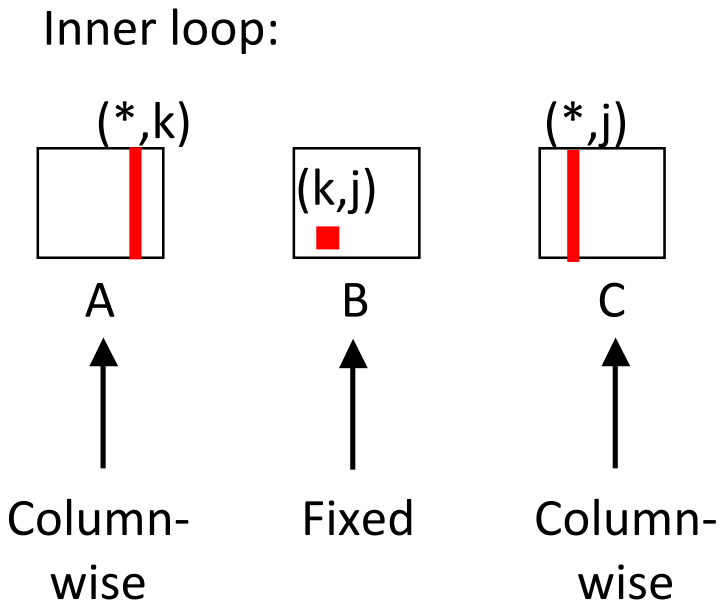


Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

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

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
- 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];
 }
}
```

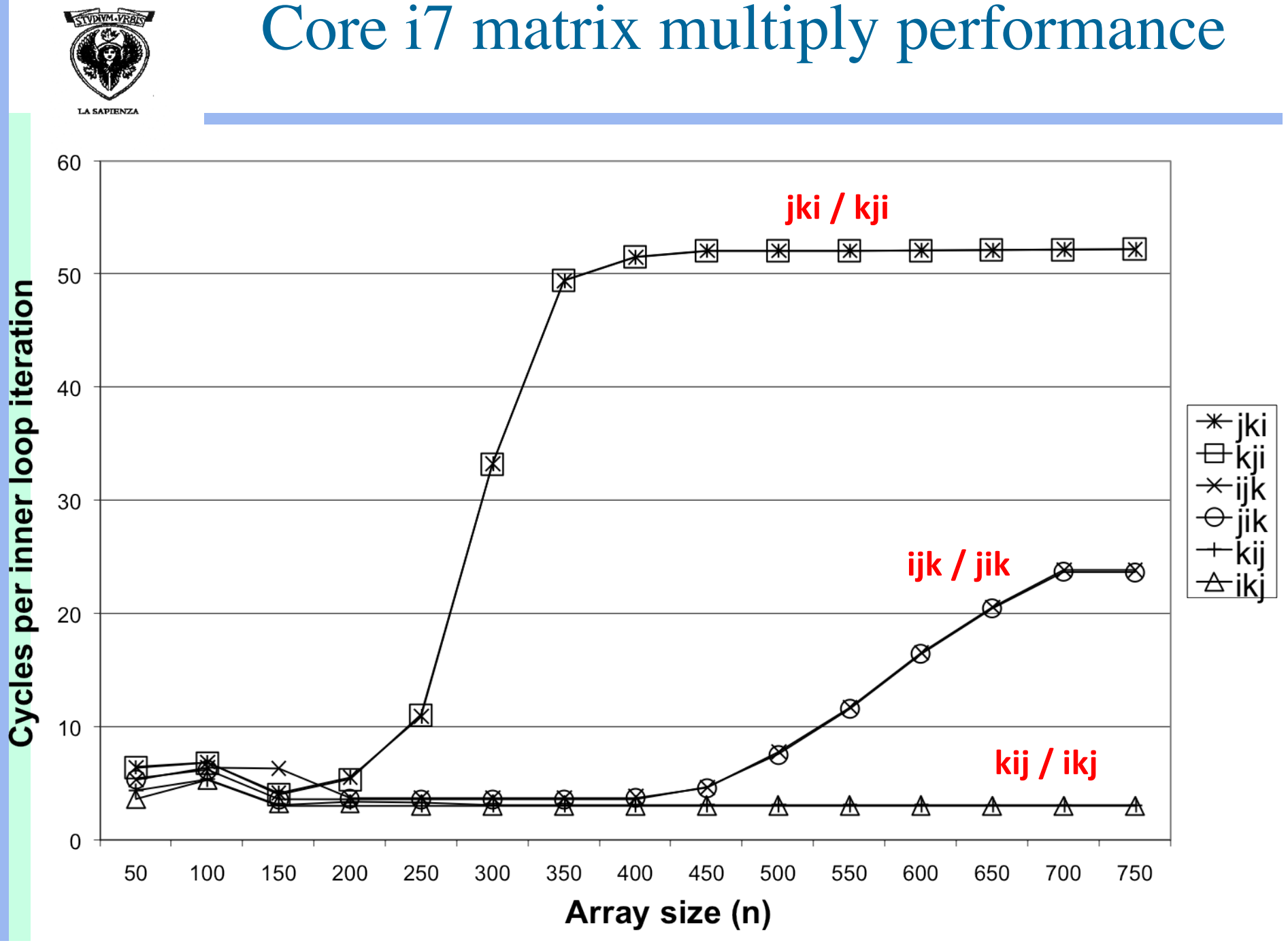
kij (& ikj):

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


Core i7 matrix multiply performance
LA SAPIENZA
Cycles per inner loop iteration
60
50
40
30
20
10
0
jki / kji
ijk / jik
kij / ikj
50
100
150
200
250
300
350
400
450
500
550
600
650
700
750
Array size (n)
jki
kji
ijk
jik
kij
ikj

Blocking

- Improves temporal locality
- Same idea as seen in external memory, but with different parameters
- Parameter choice is an issue, due to multiple cache levels: hierarchical blocking?
- Very difficult in practice tuning the code for a specific memory hierarchy
- Simpler idea: using recursion (we'll see later)

Concluding remarks

- Programmers can optimize for cache performance:
  - How data structures are organized
  - How data are accessed
    - Nested loop structure
    - Blocking
- All systems favor “cache friendly code”
  - Getting absolute optimum performance is very platform specific
    - Cache sizes, line sizes, associativities, etc.
  - Can get most of the advantage with generic code
    - Keep working set reasonably small (temporal locality)
    - Use small strides (spatial locality)

Credits

Computer systems: a programmer's perspective

Randal Bryant - David O'Hallaron

Pearson, second edition, 2010