Computer Organization: A Programmer's Perspective

Optimizing for Cache Performance

Gal A. Kaminka
galk@cs.biu.ac.il

Cache Performance Metrics

Miss Rate

- Fraction (percent) of memory references not found in cache
- Typical numbers (misses/references):
  - 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:
  - 4 clock cycles for L1
  - 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”**

Writing Cache Friendly Code

Repeated references to variables are good (temporal locality)

Stride-1 reference patterns are good (spatial locality)

Examples:

cold cache, 4-byte words, 4-word cache blocks

```
int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;

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

```
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;

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

Miss rate = **1/4 = 25%**

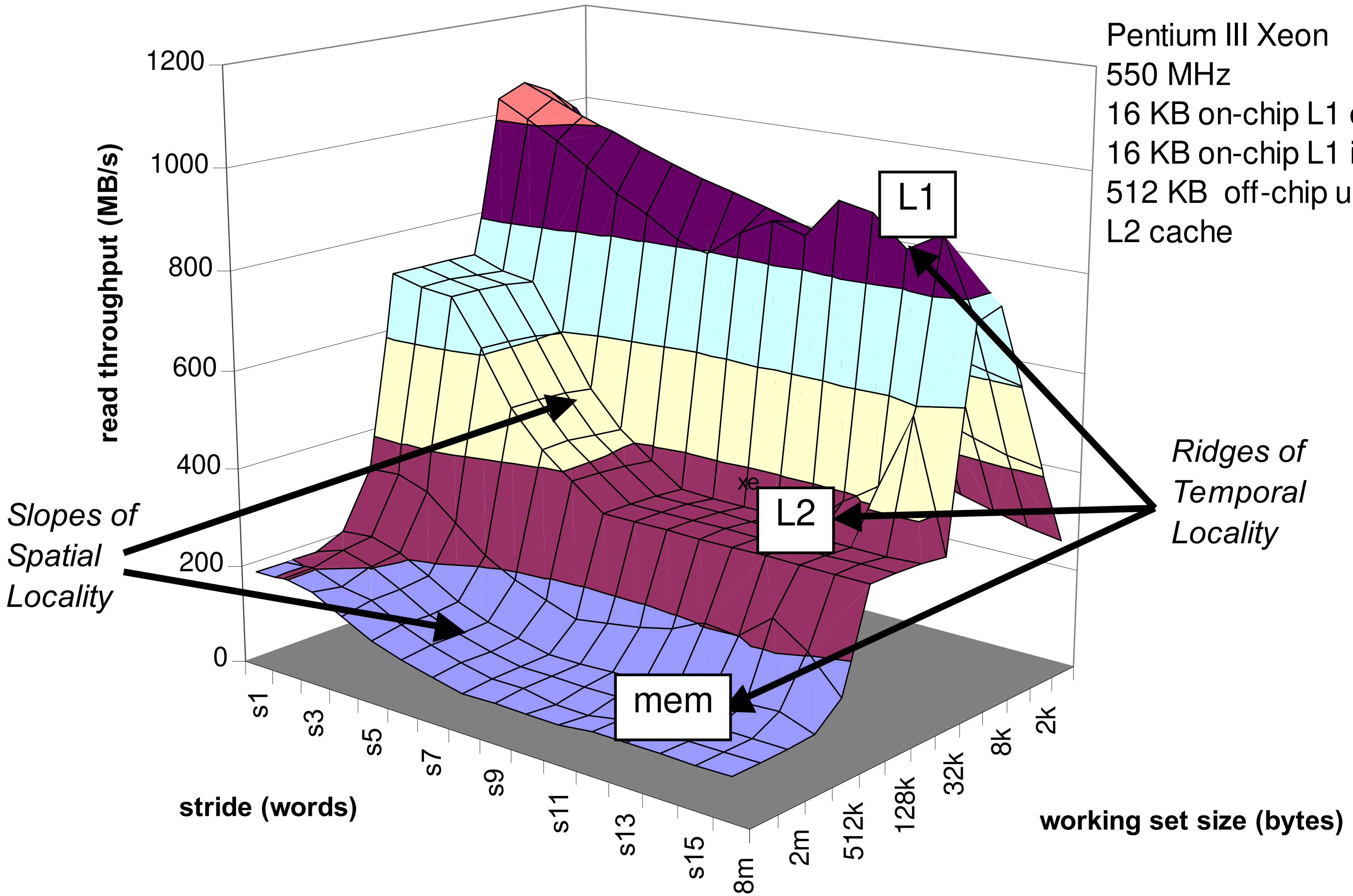
Miss rate = **100%**

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.

The Memory Mountain

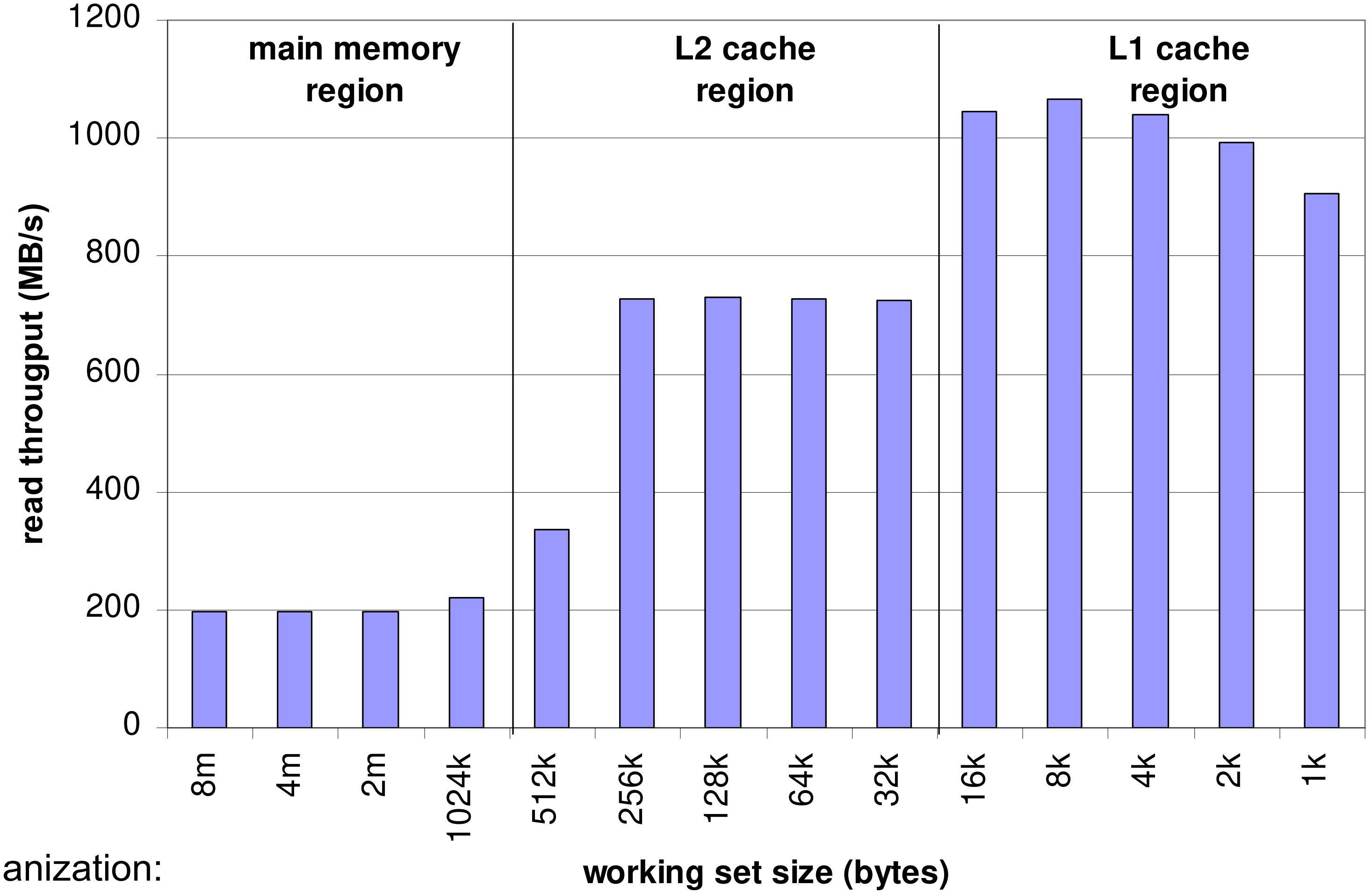
Pentium III Xeon
550 MHz
16 KB on-chip L1 d-cache
16 KB on-chip L1 i-cache
512 KB off-chip unified
L2 cache



Ridges of Temporal Locality

Slice through the memory mountain with stride=1

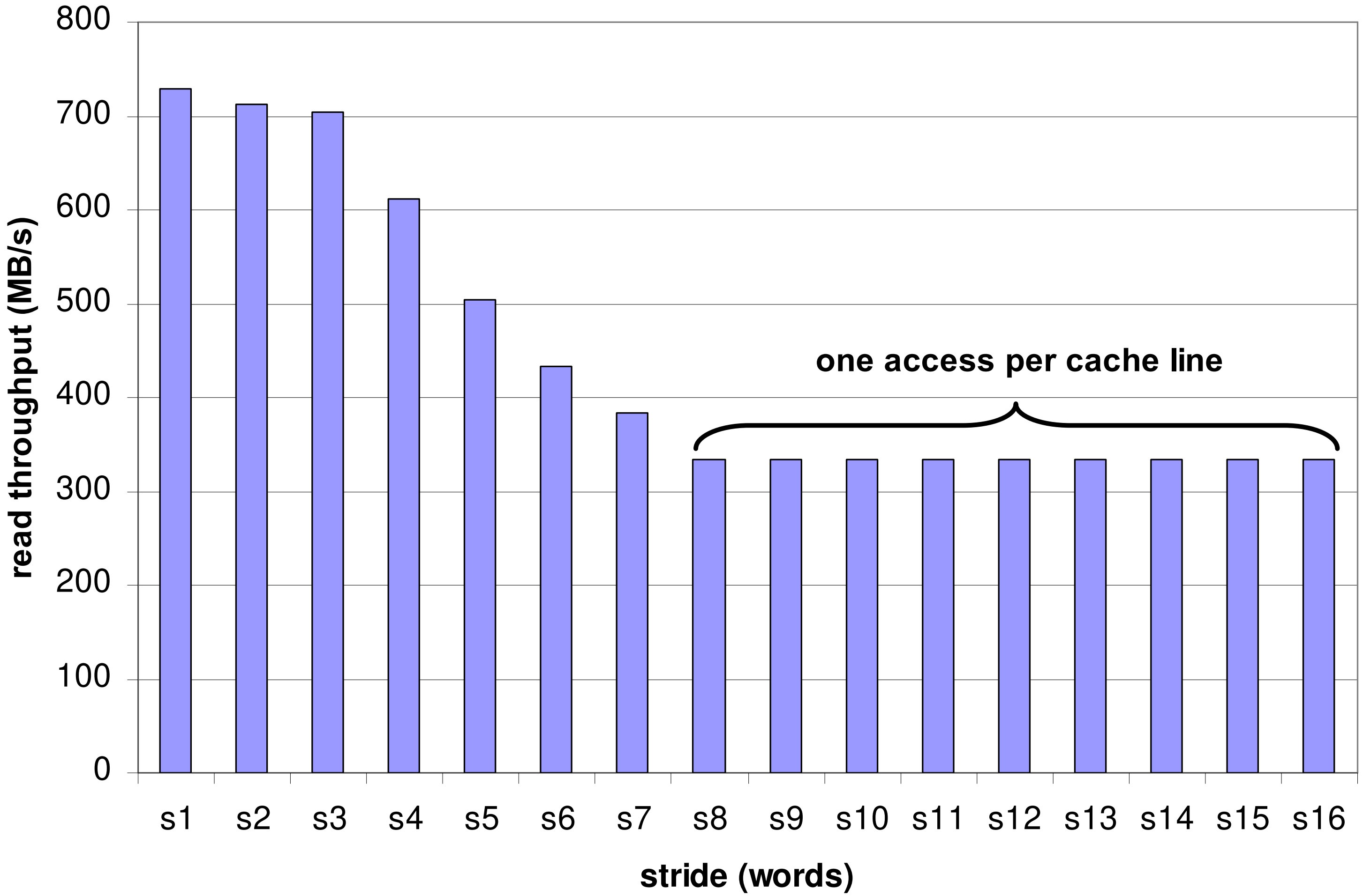
illuminates read throughputs of different caches and memory



A Slope of Spatial Locality

Slice through memory mountain with size=256KB

shows cache block size.



Matrix Multiplication Example

- Major Cache Effects to Consider
  - Total cache size
    - Exploit temporal locality and keep the working set small (e.g., by using blocking)
  - Block size
    - Exploit spatial locality

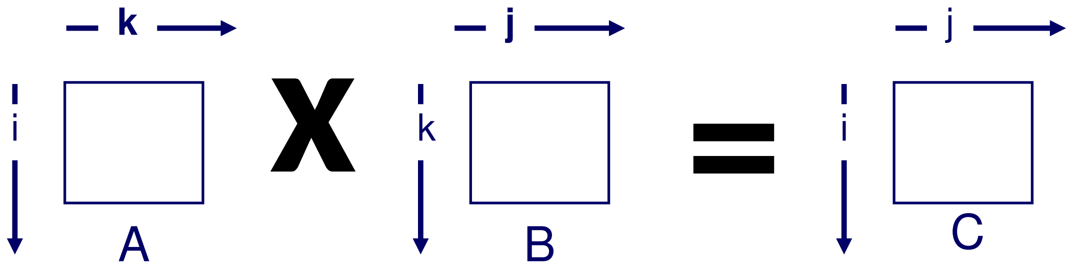
```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Variable ***sum*** *held in register*

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

Miss Rate Analysis for Matrix Multiply

- Assume:
  - Line size = 32B (big enough for 4 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



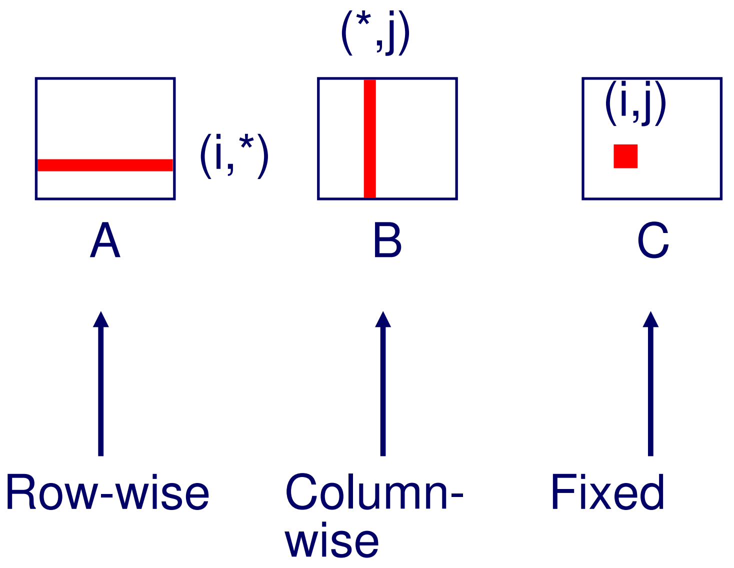
Layout of C Arrays in Memory

- 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) > sizeof(a[i][j]), exploit spatial locality
    - miss rate = sizeof(a[i][j]) / B
- 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%)
      ```

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:



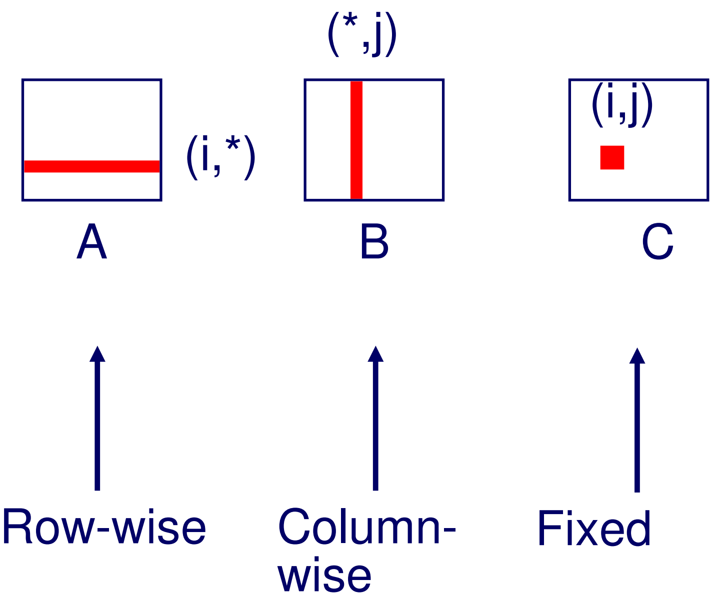
Misses per Inner Loop Iteration:

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:



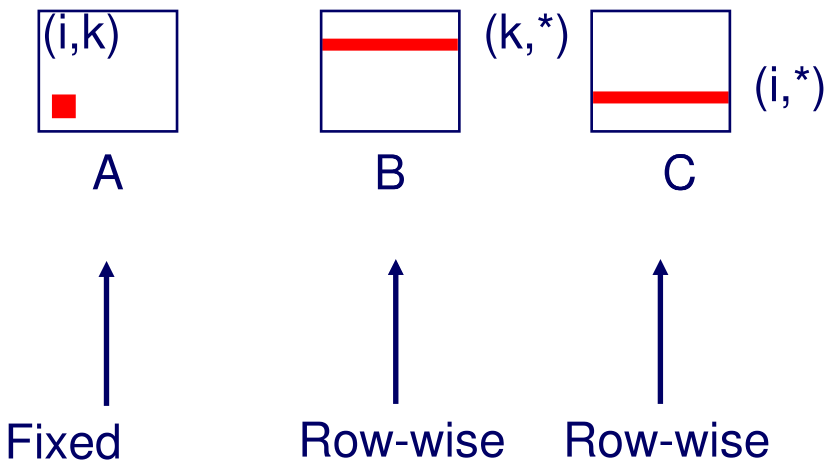
Misses per Inner Loop Iteration:

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];
  }
}
```

Inner loop:



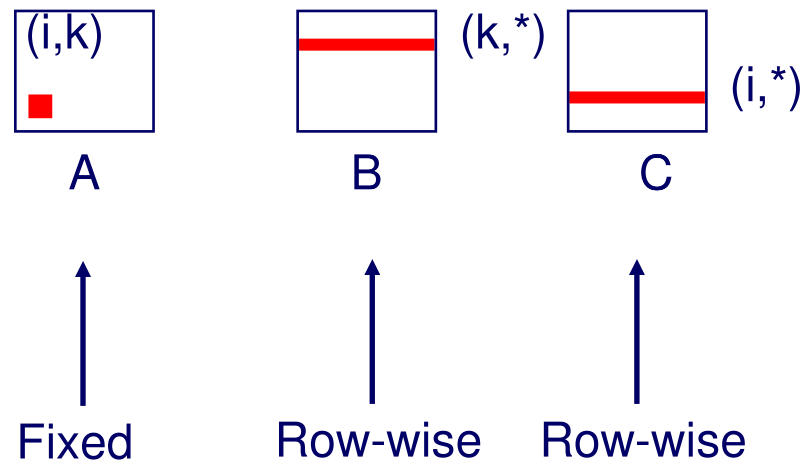
<u>Misses per Inner Loop Iteration:</u>

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];
  }
}
```

Inner loop:



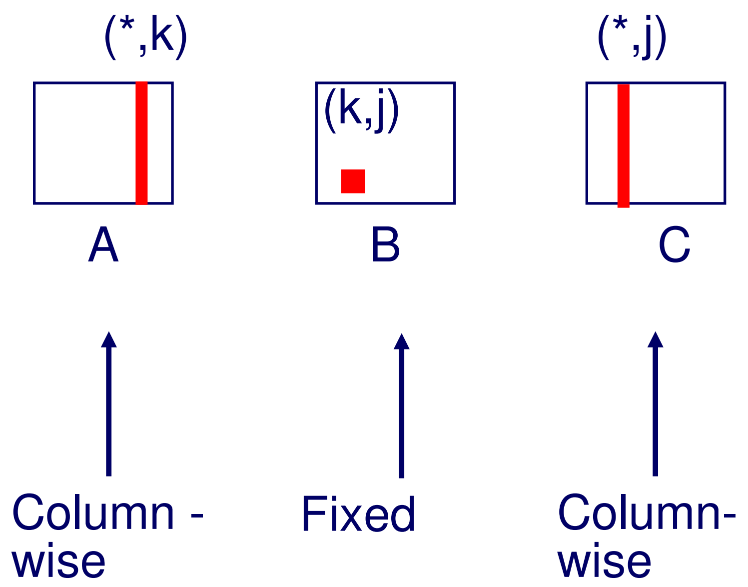
Misses per Inner Loop Iteration:

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:



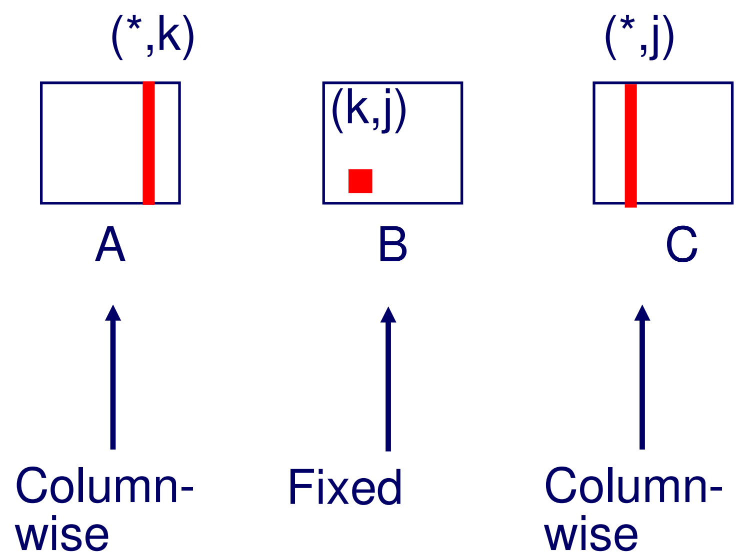
Misses per Inner Loop Iteration:

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;
  }
}
```

Inner loop:



<u>Misses per Inner Loop Iteration:</u>

A	B	C
1.0	0.0	1.0

Summary of Matrix Multiplication

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;
   }
}
```

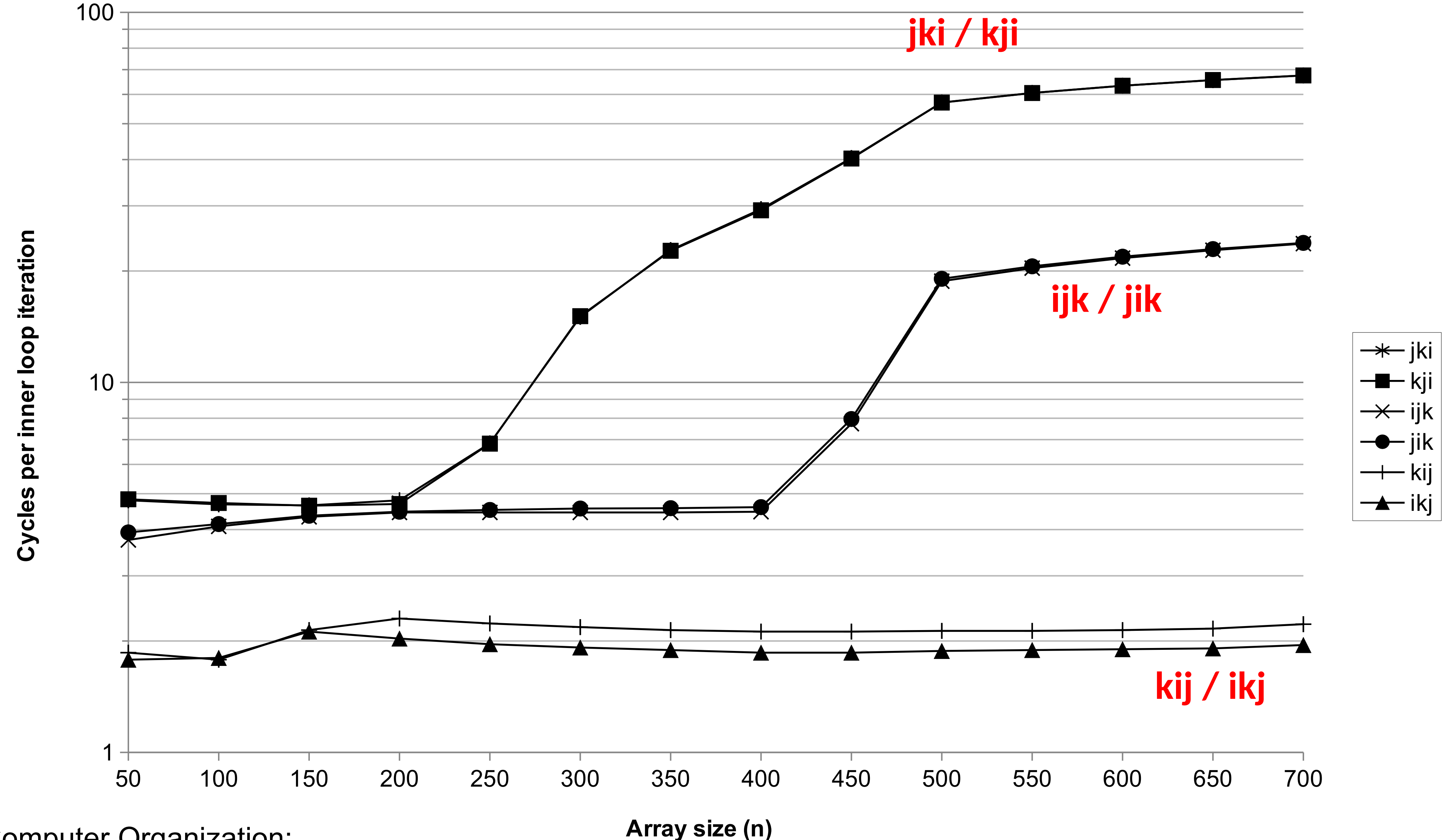
kij (& ikj):
2 loads, 1 store
misses/iter = **0.5**

```
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];
   }
}
```

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;
   }
}
```

Core i7 Matrix Multiply Performance



Improving Temporal Locality by Blocking

Example: Blocked matrix multiplication

"block" (in this context) does not mean "cache block".

Instead, it mean a sub-block within the matrix.

Example: N = 8; sub-block size = 4

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \text{ X } \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Key idea: Sub-blocks (i.e., $\mathbf{A}_{\mathbf{xy}}$) can be treated just like scalars.

$$C_{11} = A_{11}B_{11} + A_{12}B_{21} \qquad C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21} \qquad C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

Blocked Matrix Multiply (bijk)

```
for (jj=0; jj<n; jj+=bsize) {
  for (i=0; i<n; i++)
    for (j=jj; j < min(jj+bsize,n); j++)
      c[i][j] = 0.0;
  for (kk=0; kk<n; kk+=bsize) {
    for (i=0; i<n; i++) {
      for (j=jj; j < min(jj+bsize,n); j++) {
        sum = 0.0
        for (k=kk; k < min(kk+bsize,n); k++) {
          sum += a[i][k] * b[k][j];
        }
        c[i][j] += sum;
      }
    }
  }
}
```

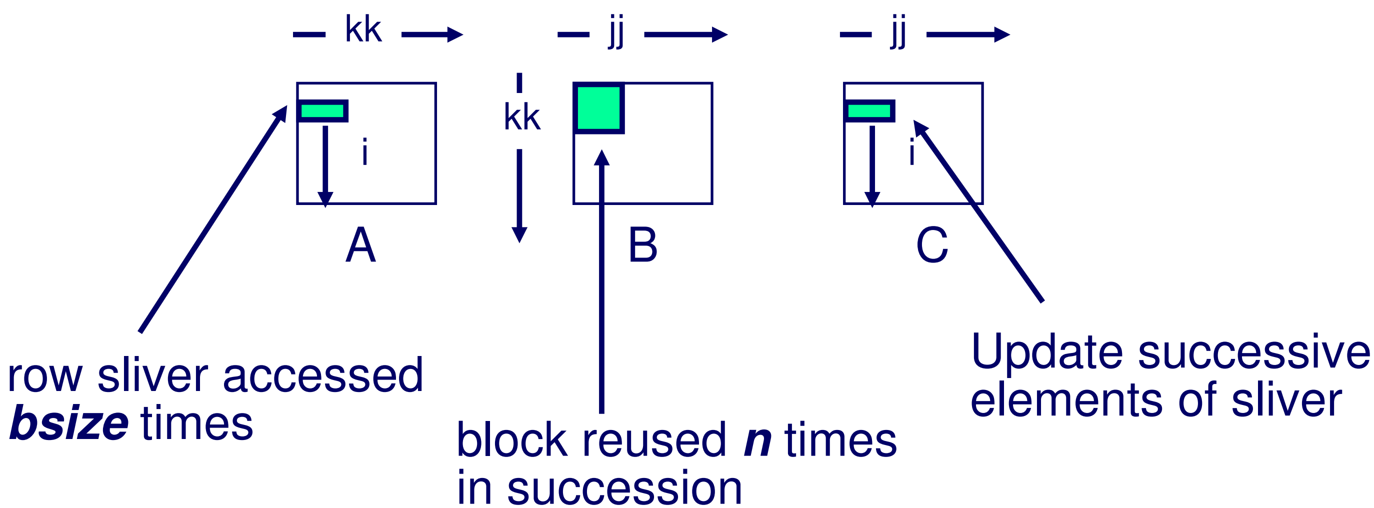
Blocked Matrix Multiply Analysis

Innermost loop pair multiplies a *1 X bsize* sliver of *A* by a *bsize X bsize* block of *B* and accumulates into *1 X bsize* sliver of *C*

Loop over *i* steps through *n* row slivers of *A* & *C*, using same *B*

```
for (i=0; i<n; i++) {
    for (j=jj; j < min(jj+bsize,n); j++) {
      sum = 0.0
      for (k=kk; k < min(kk+bsize,n); k++) {
        sum += a[i][k] * b[k][j];
      }
      c[i][j] += sum;
    }
```

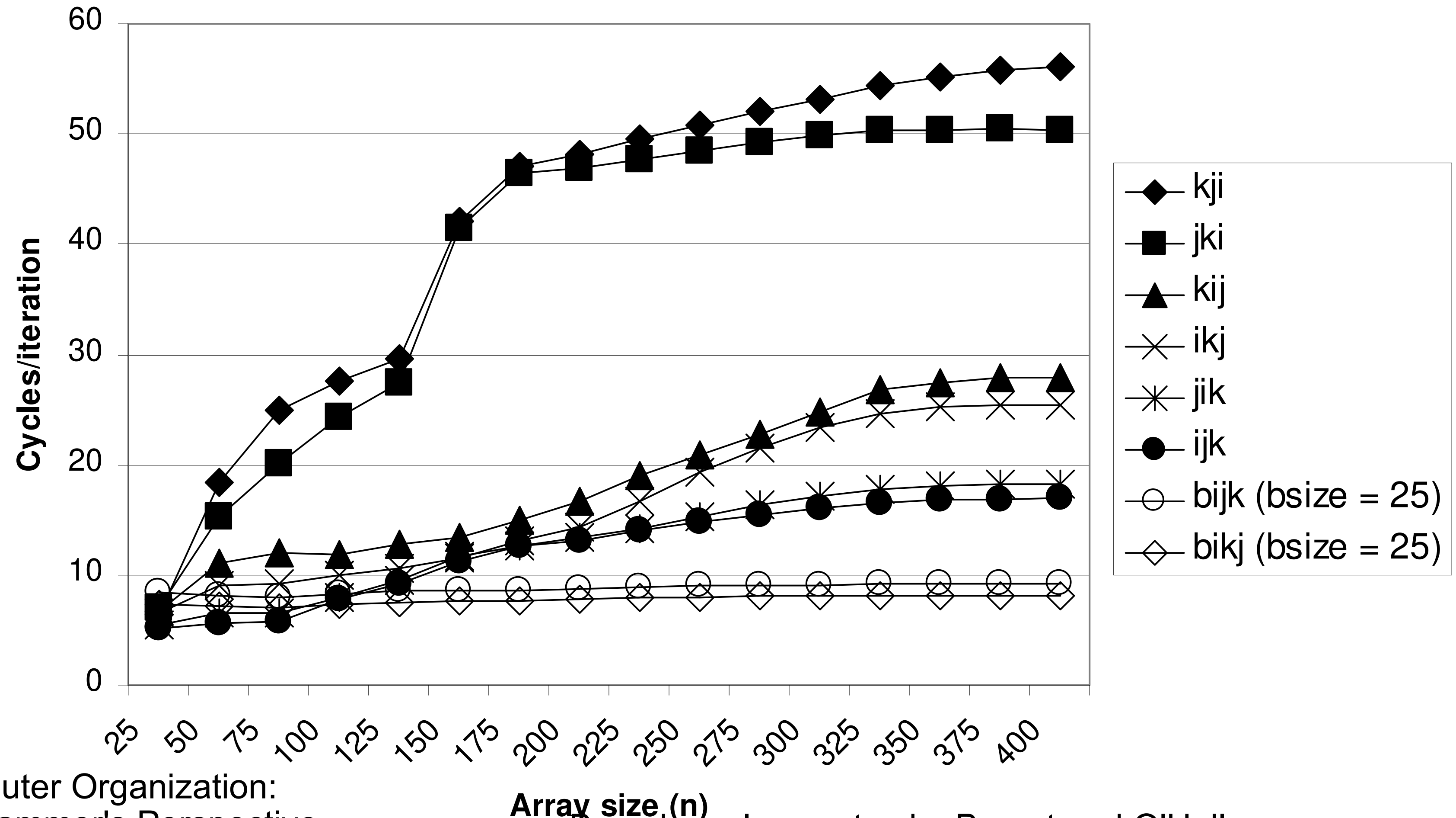
Innermost Loop Pair



Pentium Blocked Matrix Multiply Performance

Blocking (bijk and bikj) improves performance by a factor of two over unblocked versions (ijk and jik)

relatively insensitive to array size.



Concluding Observations

- Programmer can optimize for cache performance
  - How data structures are organized
  - How data are accessed
    - Nested loop structure
    - Blocking is a general technique
- 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)

שאלות?