

Lecture 14:

Cache Performance Measurement and Optimization

Prof. Matt Welsh
October 20, 2009



Topics for today

- Cache performance metrics
- Discovering your cache's size and performance
- The “Memory Mountain”
- Matrix multiply, six ways
- Blocked matrix multiplication

Cache Performance Metrics

Miss Rate

- Fraction of memory references not found in cache ($\# \text{ misses} / \# \text{ references}$)
- Typical numbers:
 - *3-10% for L1*
 - *Can be quite small (e.g., $< 1\%$) for L2, depending on size and locality.*

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 cycles for L1*
 - *5-20 clock cycles for L2*

Miss Penalty

- Additional time required because of a miss
 - *Typically 50-200 cycles for main memory*

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 sum_array_rows(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;
}
```

Miss rate = 1/4 = 25%

```
int sum_array_cols(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;
}
```

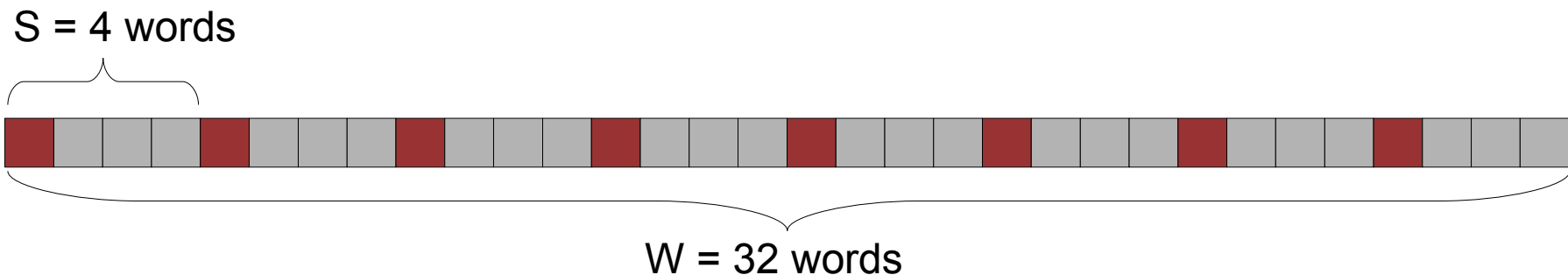
Miss rate = 100%

Determining cache characteristics

- Say I gave you a machine and didn't tell you anything about its cache size or speeds.
- How would you figure these values out?

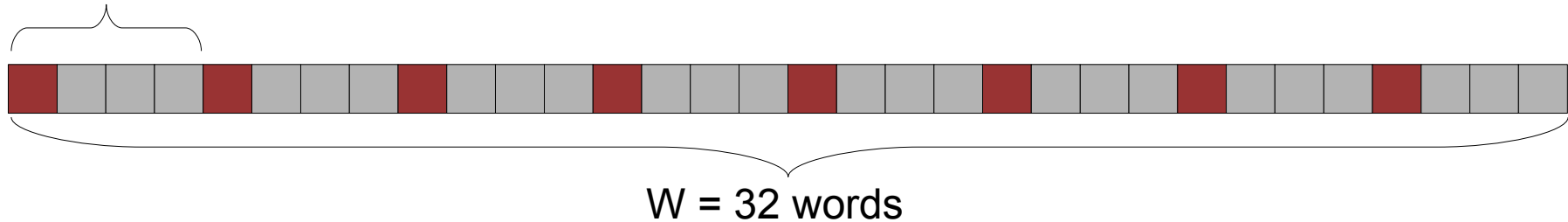
Determining cache characteristics

- Say I gave you a machine and didn't tell you anything about its cache size or speeds.
- How would you figure these values out?
- Idea: Write a program to measure the cache's behavior and performance.
 - Program needs to perform memory accesses with different locality patterns.
- Simple approach:
 - Allocate array of size “W” words
 - Loop over the array with stride index “S” and measure speed of memory accesses
 - Vary W and S to estimate cache characteristics



Determining cache characteristics

$S = 4$ words

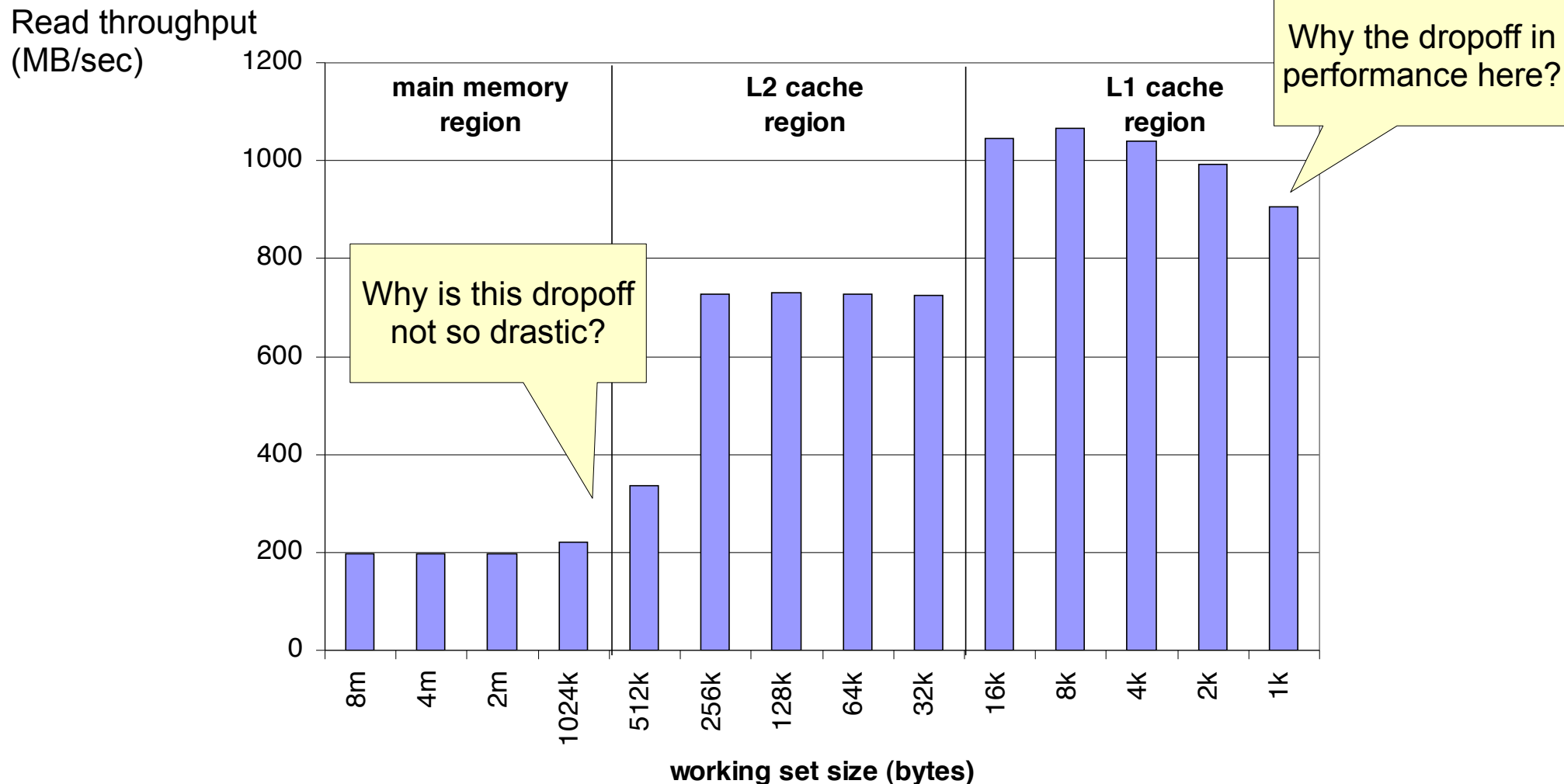


- What happens as you vary W and S ?
- Changing W varies the total amount of memory accessed by the program.
 - As W gets larger than one level of the cache, performance of the program will drop.
- Changing S varies the spatial locality of each access.
 - If S is less than the size of a cache line, sequential accesses will be fast.
 - If S is greater than the size of a cache line, sequential accesses will be slower.
- See end of lecture notes for example C program to do this.

Varying Working Set

Keep stride constant at $S = 1$, and vary W from 1KB to 8MB

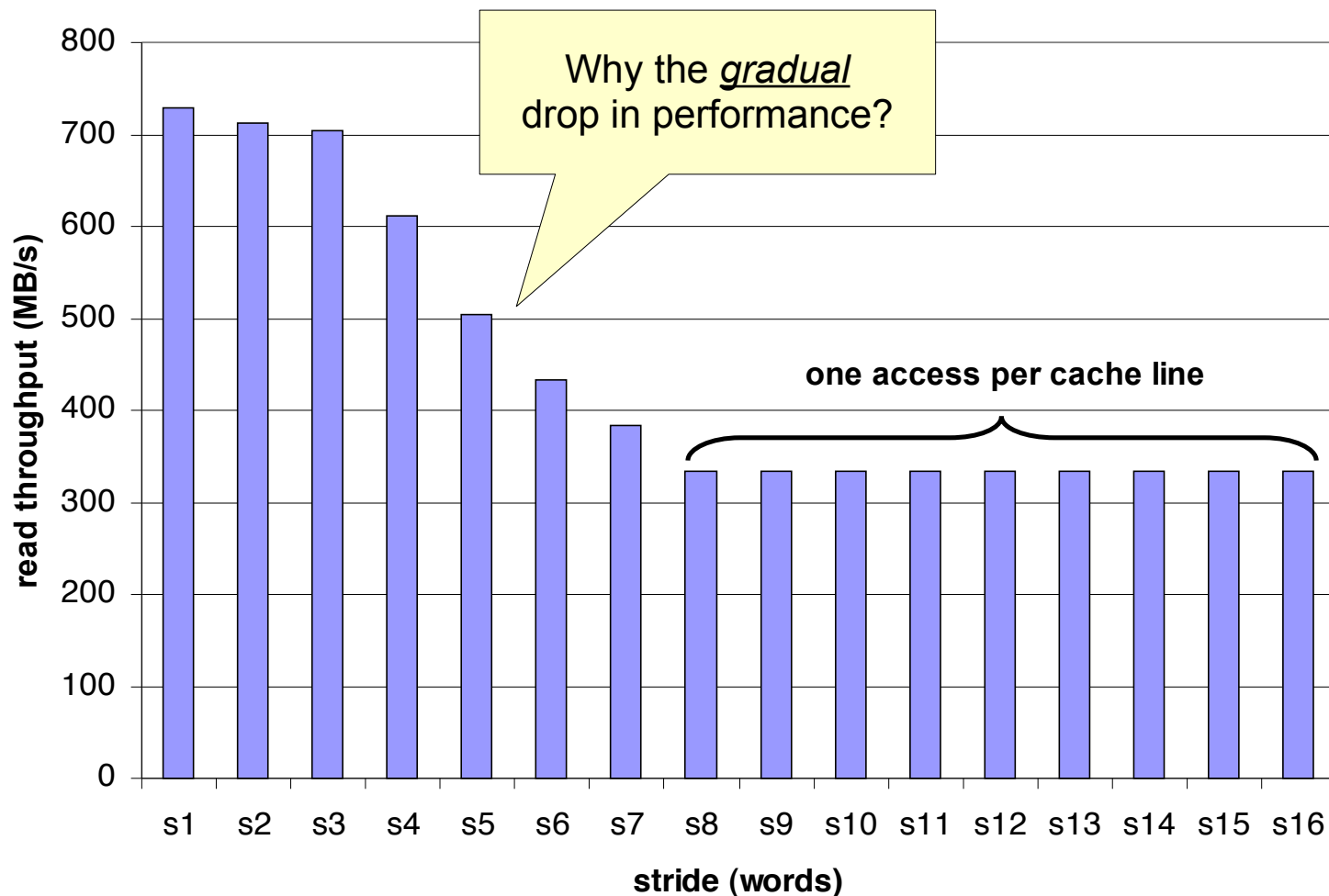
- Shows size and read throughputs of different cache levels and memory



Varying stride

Keep working set constant at $W = 256$ KB, vary stride from 1-16

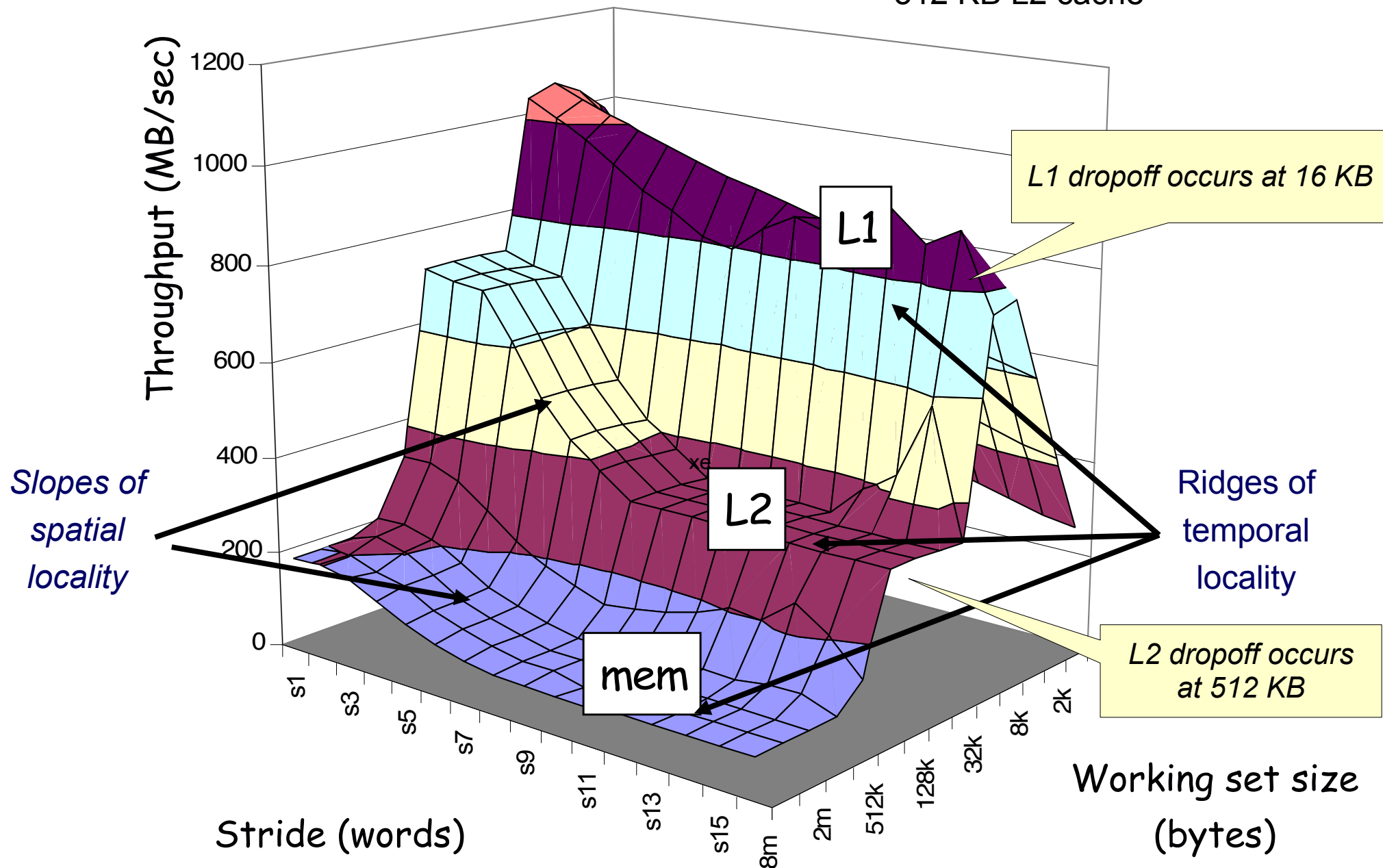
- Shows the cache block size.



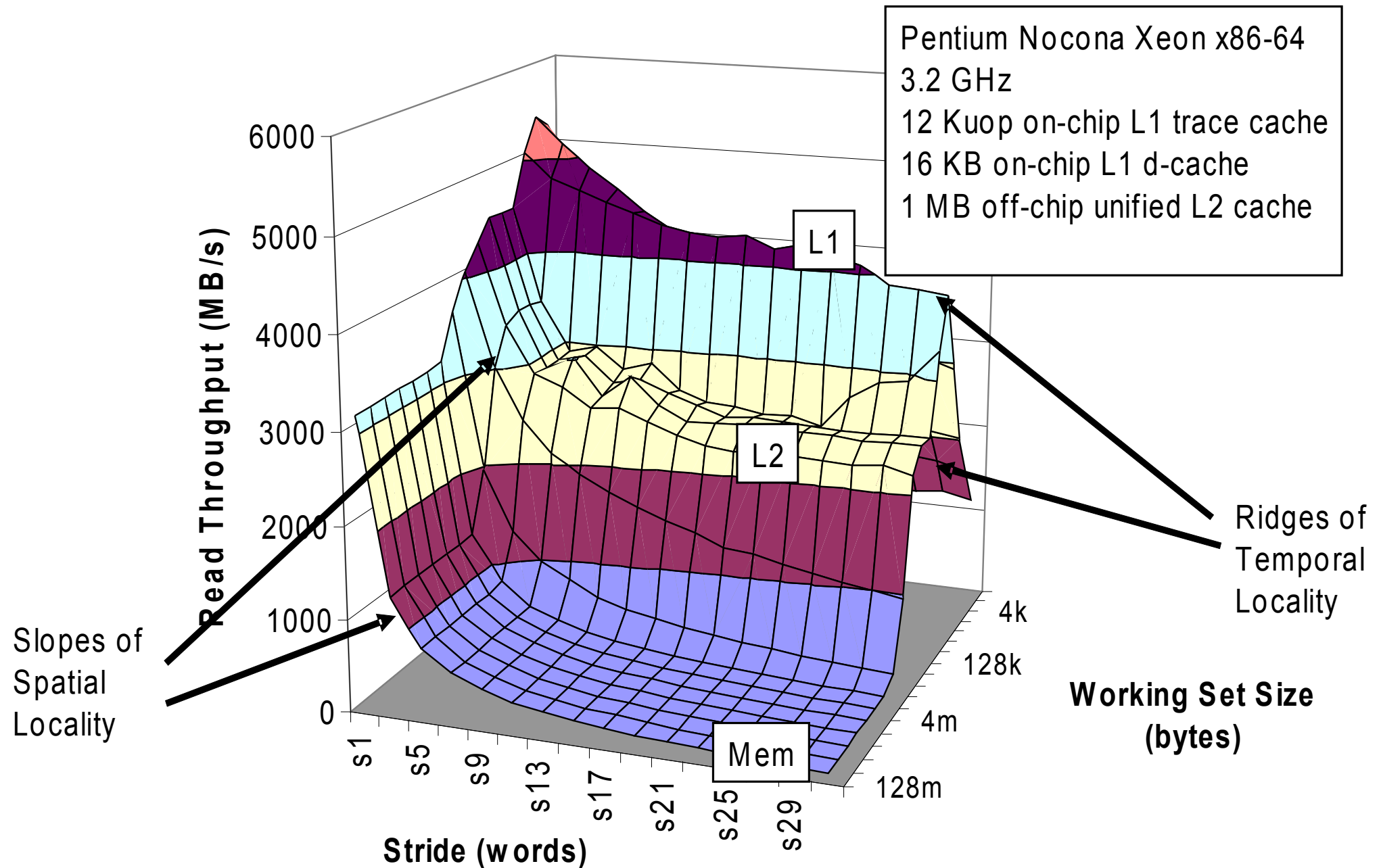
The Memory Mountain

Pentium III – 550 Mhz

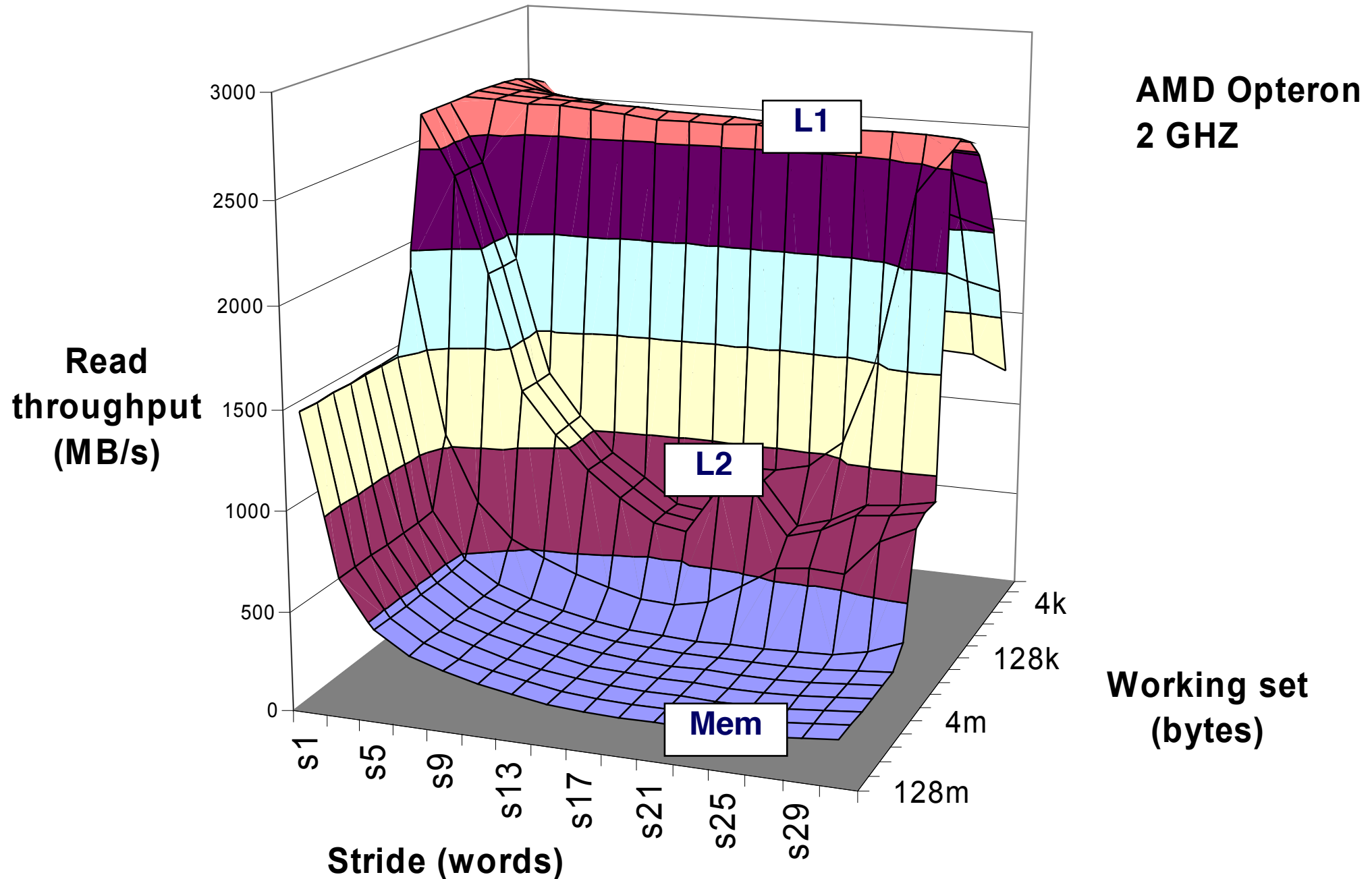
16 KB L1 cache
512 KB L2 cache



X86-64 Memory Mountain



Opteron Memory Mountain

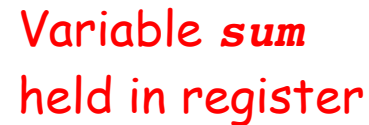


Matrix Multiplication Example

- Matrix multiplication is heavily used in numeric and scientific applications.
 - It's also a nice example of a program that is highly sensitive to cache effects.
- Multiply two $N \times N$ matrices
 - $O(N^3)$ total operations
 - Read N values for each source element
 - Sum up N values for each destination

```
/* 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



Matrix Multiplication Example

```
/* 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;  
    }  
}
```

$$4*3 + 2*2 + 7*5 = 51$$

4	2	7	x	3	0	1	=	51		
1	8	2		2	4	5				
6	0	1		5	9	1				

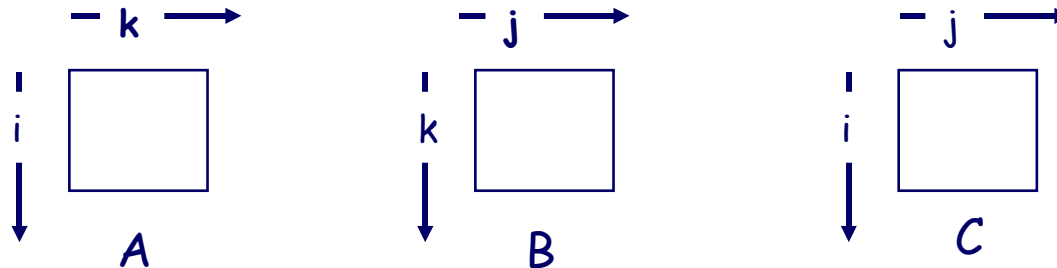
Miss Rate Analysis for Matrix Multiply

Assume:

- Line size = 32B (big enough for four 64-bit “double” values)
- Matrix dimension (N) is very large
- 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:

- ```
for (i = 0; i < N; i++)
 sum += a[0][i];
```
- Accesses successive elements
- Compulsory miss rate: **(8 bytes per double) / (block size of cache)**

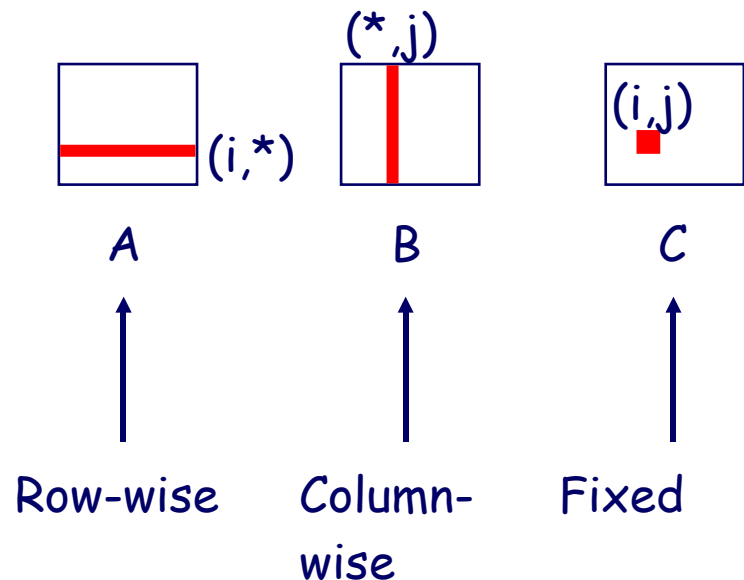
Stepping through rows in one column:

- ```
for (i = 0; i < n; i++)  
    sum += a[i][0];
```
- Accesses distant elements -- no spatial locality!
- Compulsory miss rate = **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:

<u>A</u>	<u>B</u>	<u>C</u>
0.25	1.0	0.0

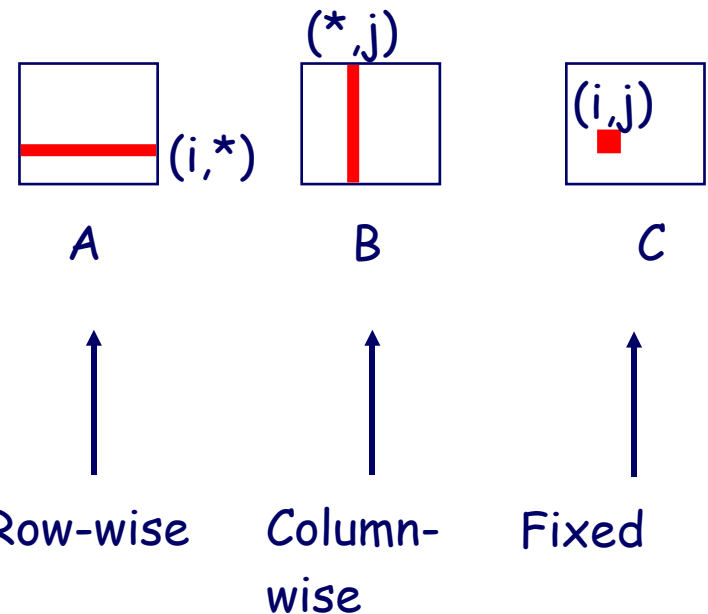
Assume cache line size of 32 bytes.

Compulsory miss rate = 8 bytes per double / 32 bytes = $\frac{1}{4}$ = 0.25

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:



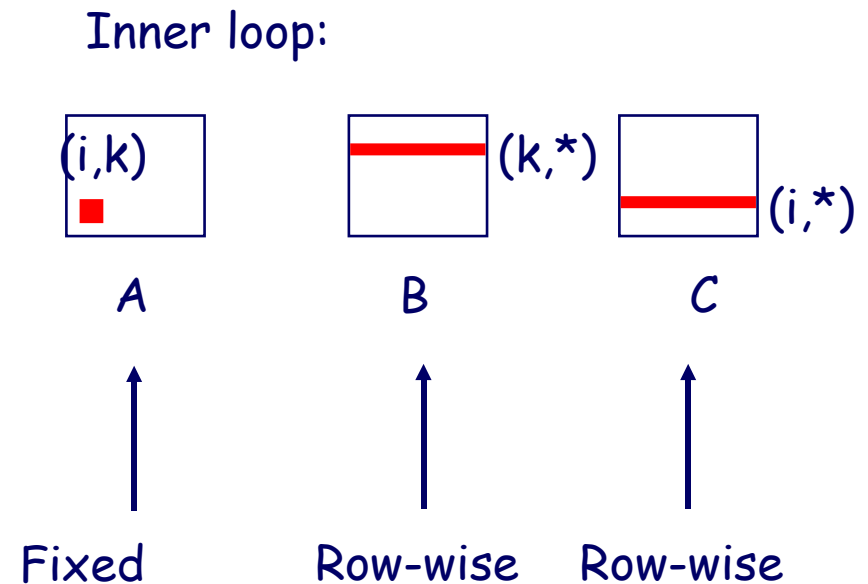
Misses per Inner Loop Iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.25	1.0	0.0

Same as ijk. Just swapped i and j.

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
0.0

B
0.25

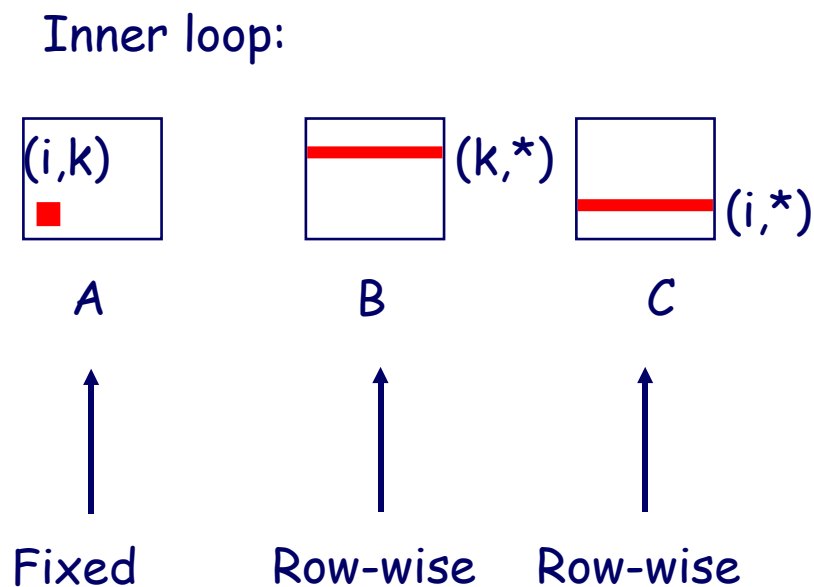
C
0.25

Now we suffer 0.25 compulsory misses per iteration for “B” and “C” accesses.

Also need to store back “temporary” result $c[i][j]$ on each innermost loop iteration!

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
0.0

B
0.25

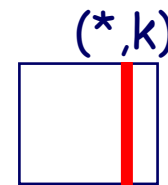
C
0.25

Same as kij.

Matrix Multiplication (jki)

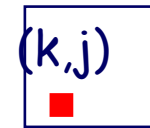
```
/* jki */
for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
        r = b[k][j];
        for (i=0; i<n; i++)
            c[i][j] += a[i][k] * r;
    }
}
```

Inner loop:



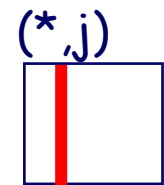
A

Column -
wise



B

Fixed



C

Column-
wise

Misses per Inner Loop Iteration:

A
1.0

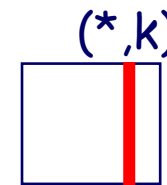
B
0.0

C
1.0

Matrix Multiplication (kji)

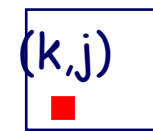
```
/* kji */
for (k=0; k<n; k++) {
  for (j=0; j<n; j++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

Inner loop:



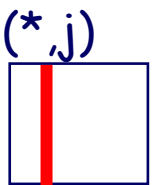
A

Column-
wise



B

Fixed



C

Column-
wise

Misses per Inner Loop Iteration:

A
1.0

B
0.0

C
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 or 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 or ikj:

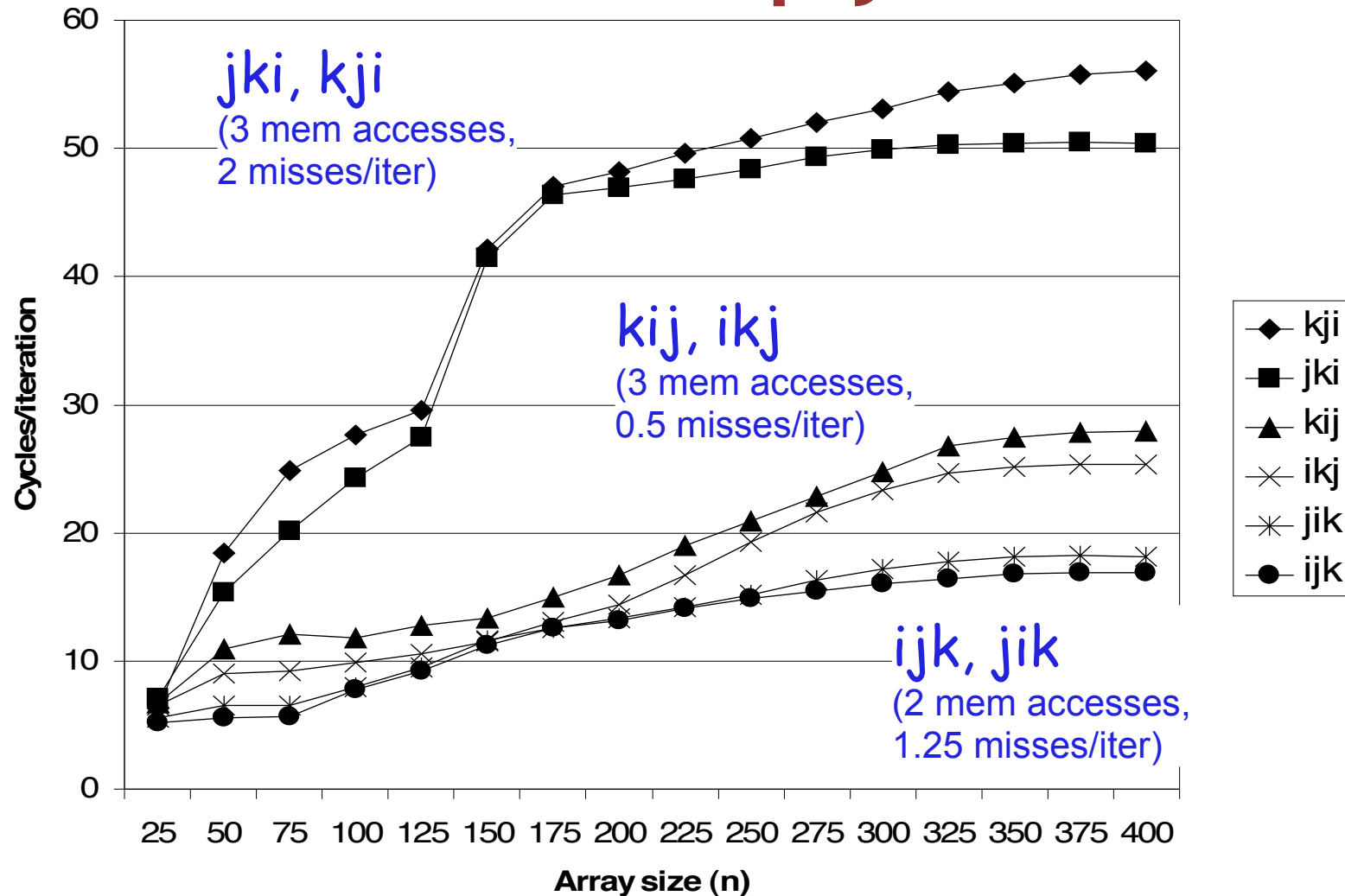
- 2 loads, 1 store
- misses/iter = **0.5**

```
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 or kji:

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

Pentium Matrix Multiply Performance



- Versions with same number of mem accesses and miss rate perform about the same.
- Lower misses/iter tends to do better
- ijk and jik version fastest, although higher miss rate than kij and ikj versions

Using blocking to improve locality

Blocked matrix multiplication

- Break matrix into smaller blocks and perform independent multiplications on each block.
- Improves locality by operating on one block at a time.
- Best if each block can fit in the cache!

Example: Break each matrix into four sub-blocks

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

Key idea: Sub-blocks (i.e., A_{xy}) can be treated just like scalars.

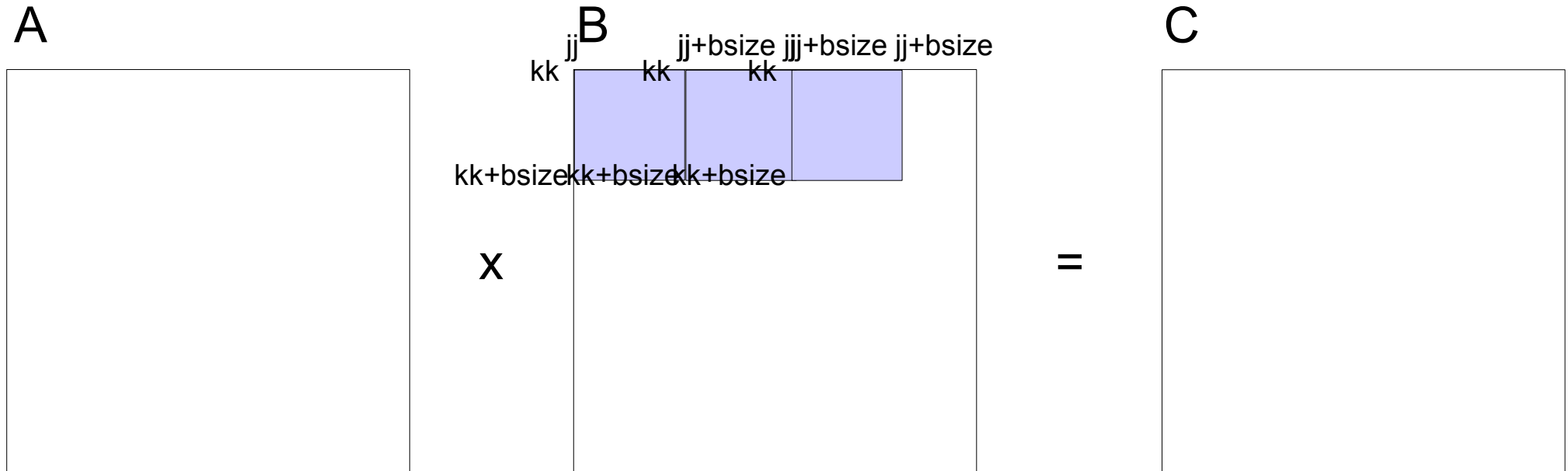
$$C_{11} = A_{11}B_{11} + A_{12}B_{21} \quad C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

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

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;  
            }  
        }  
    }  
}
```

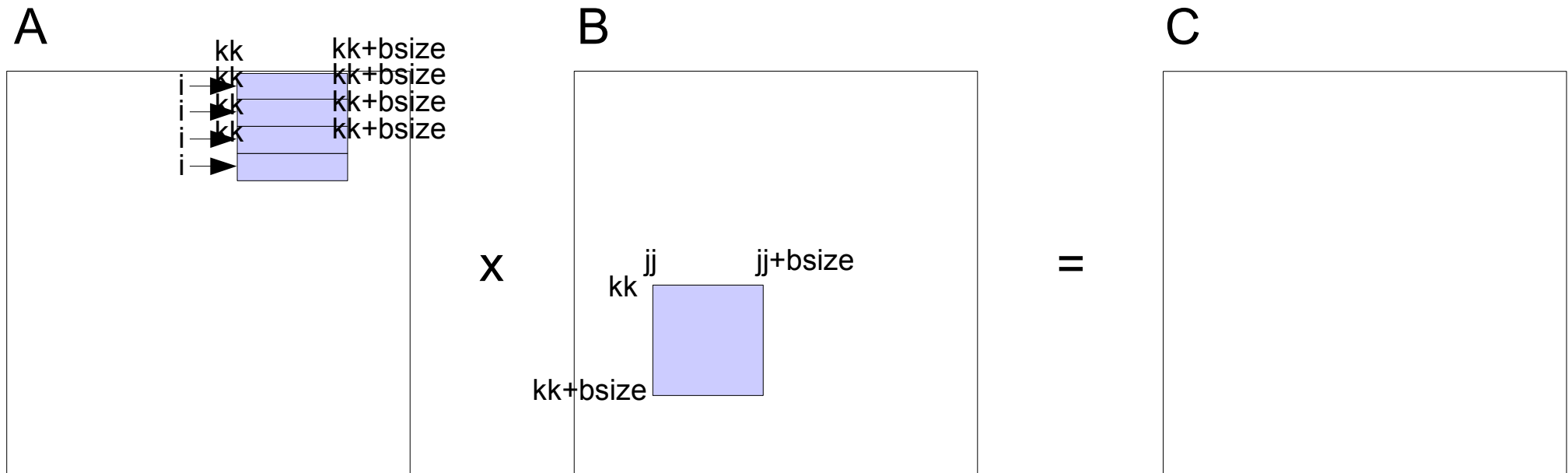
Blocked matrix multiply operation



Step 1: Pick location of block in matrix B

- Block slides across matrix B left-to-right, top-to-bottom, by “bsize” units at a time

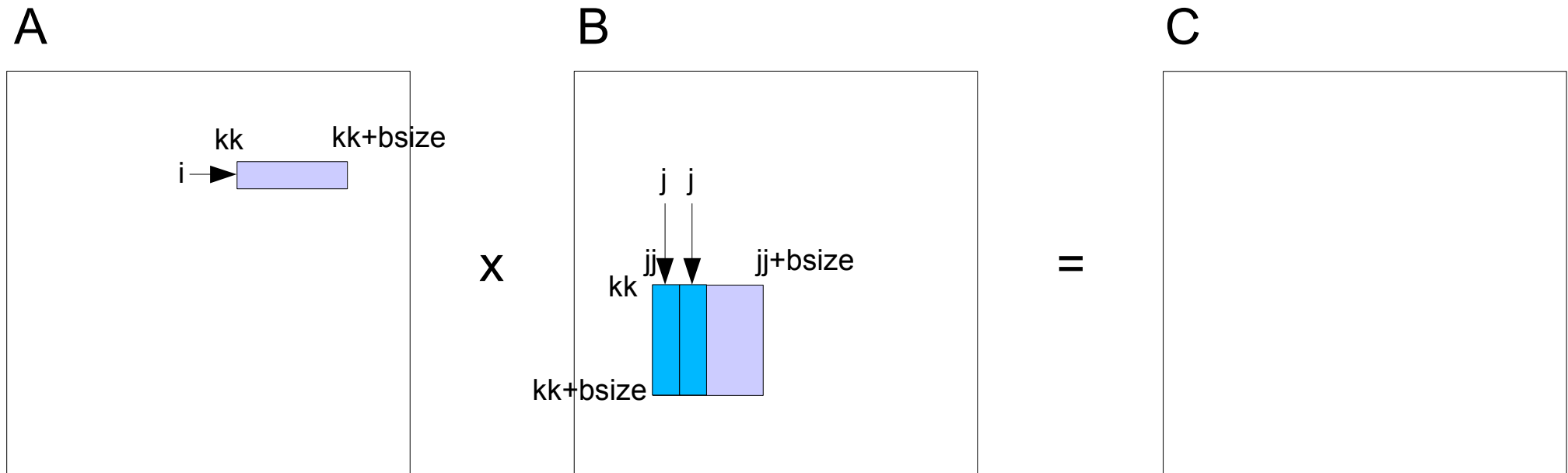
Blocked matrix multiply operation



Step 2: Slide row sliver across matrix A

- Hold block in matrix B fixed.
- Row sliver slides from top to bottom across matrix A
- Row sliver spans columns [$kk \dots kk+bsize$]

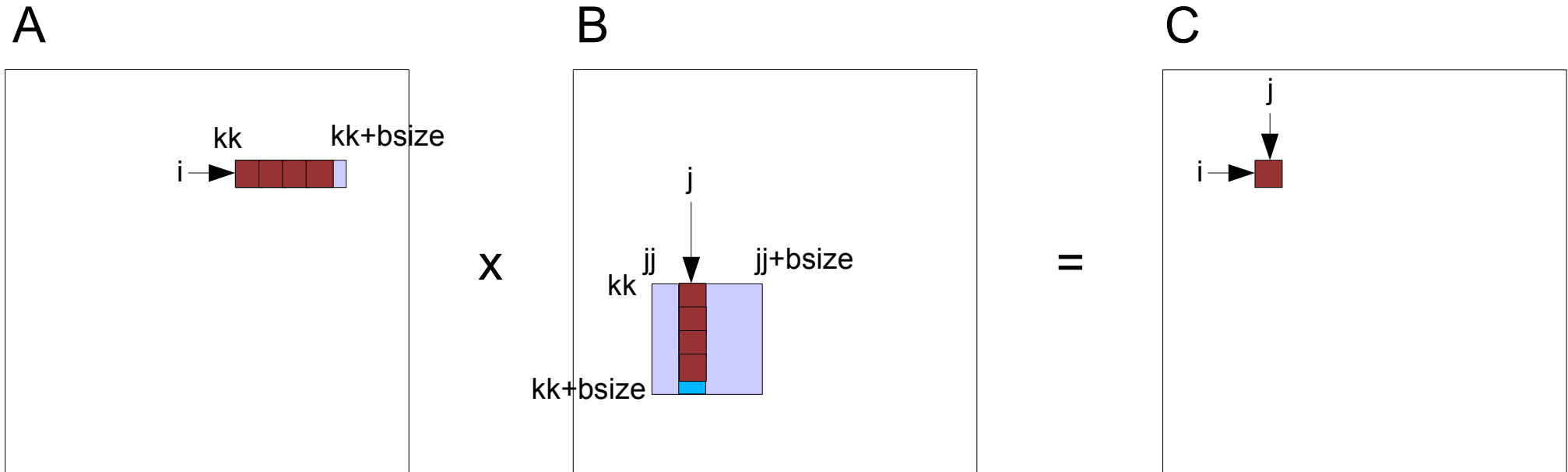
Blocked matrix multiply operation



Step 3: Slide column sliver across block in matrix B

- Row sliver in matrix A stays fixed.
- Column sliver slides from left to right across the block
- Column sliver spans rows $[kk \dots kk+bsize]$ in matrix B

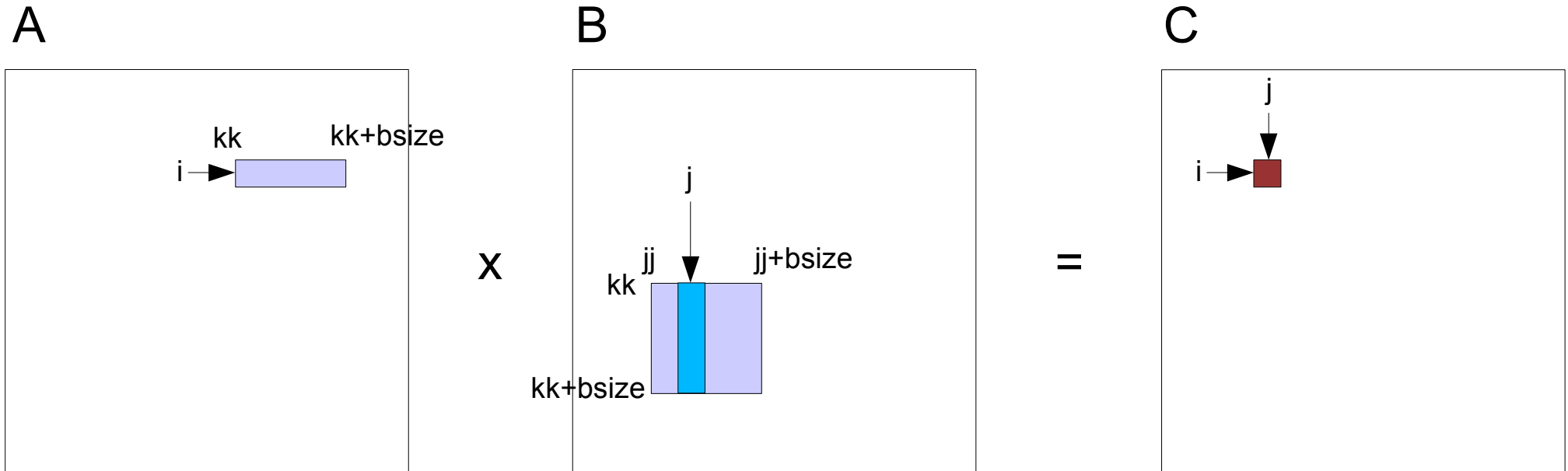
Blocked matrix multiply operation



Step 4: Iterate over row and column slivers together

- Compute dot product of both vectors of length “bsize”
- Dot product is **added to** contents of cell (i,j) in matrix C

Locality properties



What is the locality of this algorithm?

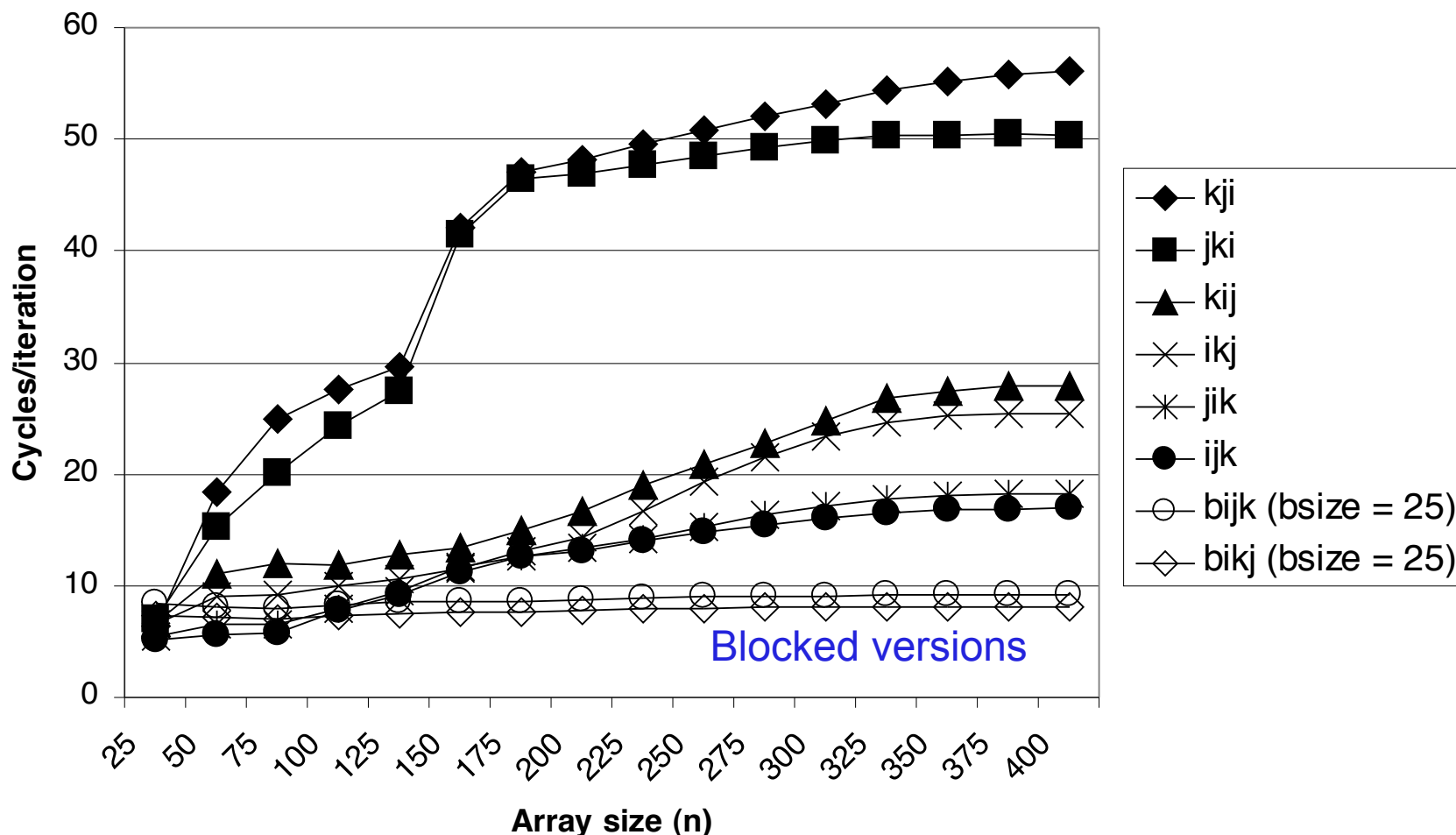
- Iterate over all elements of the block N times (once for each row sliver in A)
- Row sliver in matrix A accessed “ $bsize$ ” times (once for each column sliver in B)

If block and row slivers fit in the cache, performance should rock!

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.



Cache performance test program

```
/* 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)
{
    uint64_t start_cycles, end_cycles, diff;
    int elems = size / sizeof(int);

    test(elems, stride); /* warm up the cache */
    start_cycles = get_cpu_cycle_counter(); /* Read CPU cycle counter */
    test(elems, stride); /* Run test */
    end_cycles = get_cpu_cycle_counter(); /* Read CPU cycle counter again */
    diff = end_cycles - start_cycles; /* Compute time */
    return (size / stride) / (diff / CPU_MHZ); /* convert cycles to MB/s */
}
```

Cache performance main routine

```
#define CPU_MHZ 2.8 * 1024.0 * 1024.0; /* e.g., 2.8 GHz */
#define MINBYTES (1 << 10) /* Working set size ranges from 1 KB */
#define MAXBYTES (1 << 23) /* ... up to 8 MB */
#define MAXSTRIDE 16 /* Strides range from 1 to 16 */
#define MAXELEMS MAXBYTES/sizeof(int)

int data[MAXELEMS]; /* The array we'll be traversing */

int main()
{
    int size; /* Working set size (in bytes) */
    int stride; /* Stride (in array elements) */

    init_data(data, MAXELEMS); /* Initialize each element in data to 1 */
    for (size = MAXBYTES; size >= MINBYTES; size >>= 1) {
        for (stride = 1; stride <= MAXSTRIDE; stride++)
            printf("%.1f\t", run(size, stride));
        printf("\n");
    }
    exit(0);
}
```