Consider the C array declared by

```
double A[5][4];
```

Since memory is linear, the array must be unpacked in some systematic way.

C does it in row-major order.

a	a + 8	a + 16	a + 24
A[0][0]	A[0][1]	A[0][2]	A[0][3]
a + 32	a + 40	a + 48	a + 56
A[1][0]	A[1][1]	A[1][2]	A[1][3]
a + 64	a + 72	a + 80	a + 88
A[2][0]	A[2][1]	A[2][2]	A[2][3]
a + 96	a + 104	a + 112	a + 120
A[3][0]	A[3][1]	A[3][2]	A[3][3]
a + 128	a + 136	a + 144	a + 152
A[4][0]	A[4][1]	A[4][2]	A[4][3]

Locality Example

Claim: Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer.

Question: Does this function have good locality?

```c
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
}
```

Locality Example

Question: Does this function have good locality?

```
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
}
```

STORAGE OF FORTRAN ARRAYS

Consider the Fortran array

```
double precision A(4,5)
```

It is stored in column-major order.

a	a + 40	a + 80	a + 120
A(1,1)	A(1,2)	A(1,3)	A(1,4)
a + 8	a + 48	a + 88	a + 128
A(2,1)	A(2,2)	A(2,3)	A(2,4)
a + 16	a + 56	a + 96	a + 136
A(3,1)	A(3,2)	A(3,3)	A(3,4)
a + 24	a + 64	a + 104	a + 144
A(4,1)	A(4,2)	A(4,3)	A(4,4)
a + 32	a + 72	a + 112	a + 152
A(5,1)	A(5,2)	A(5,3)	A(5,4)

If we have to access all the elements of an array, we should access them with unit stride.

Thus in C we should access the array by rows:

```
A[0][0] A[0][1] A[0][2] A[0][3] A[1][0] A[1][1] ...
```

In Fortran we should access the array by rows.

```
A(1,1) A(2,1) A(3,1) A(4,1) A(5,1) A(1,2) A(2,2)  ...
```

This means that we must code algorithms differently in C and Fortran.

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;
}
```

Miss rate = 1/4 = 25%

```
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 = 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.

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 */
}
```

Memory Mountain Main Routine

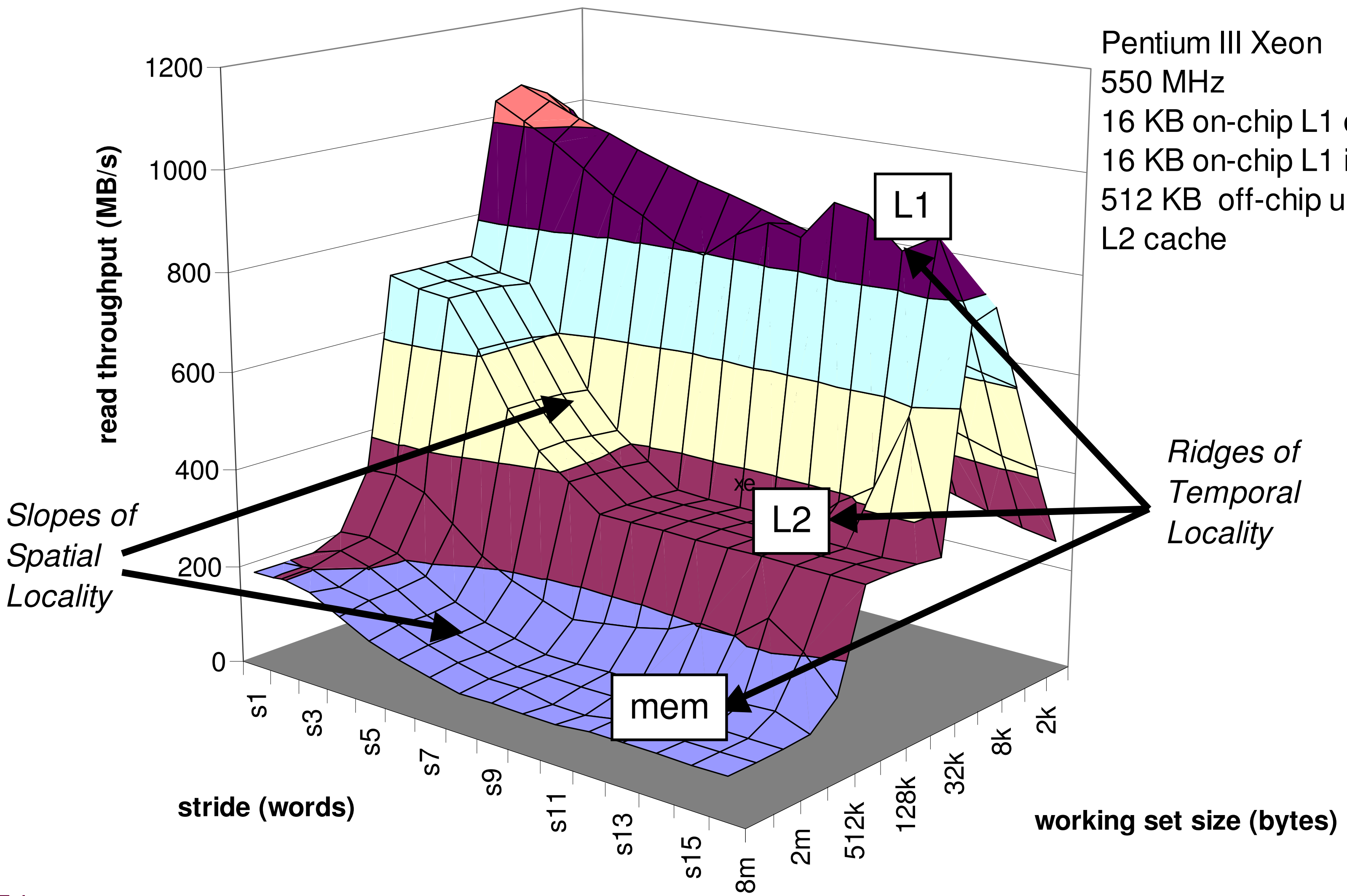
```
/* mountain.c - Generate the memory mountain. */
#define MINBYTES (1 << 10)  /* Working set size ranges from 1 KB */
#define MAXBYTES (1 << 23)  /* ... up to 8 MB */
#define MAXSTRIDE 16        /* Strides range from 1 to 16 */
#define MAXELEMS MAXBYTES/sizeof(int)

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

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

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

The Memory Mountain

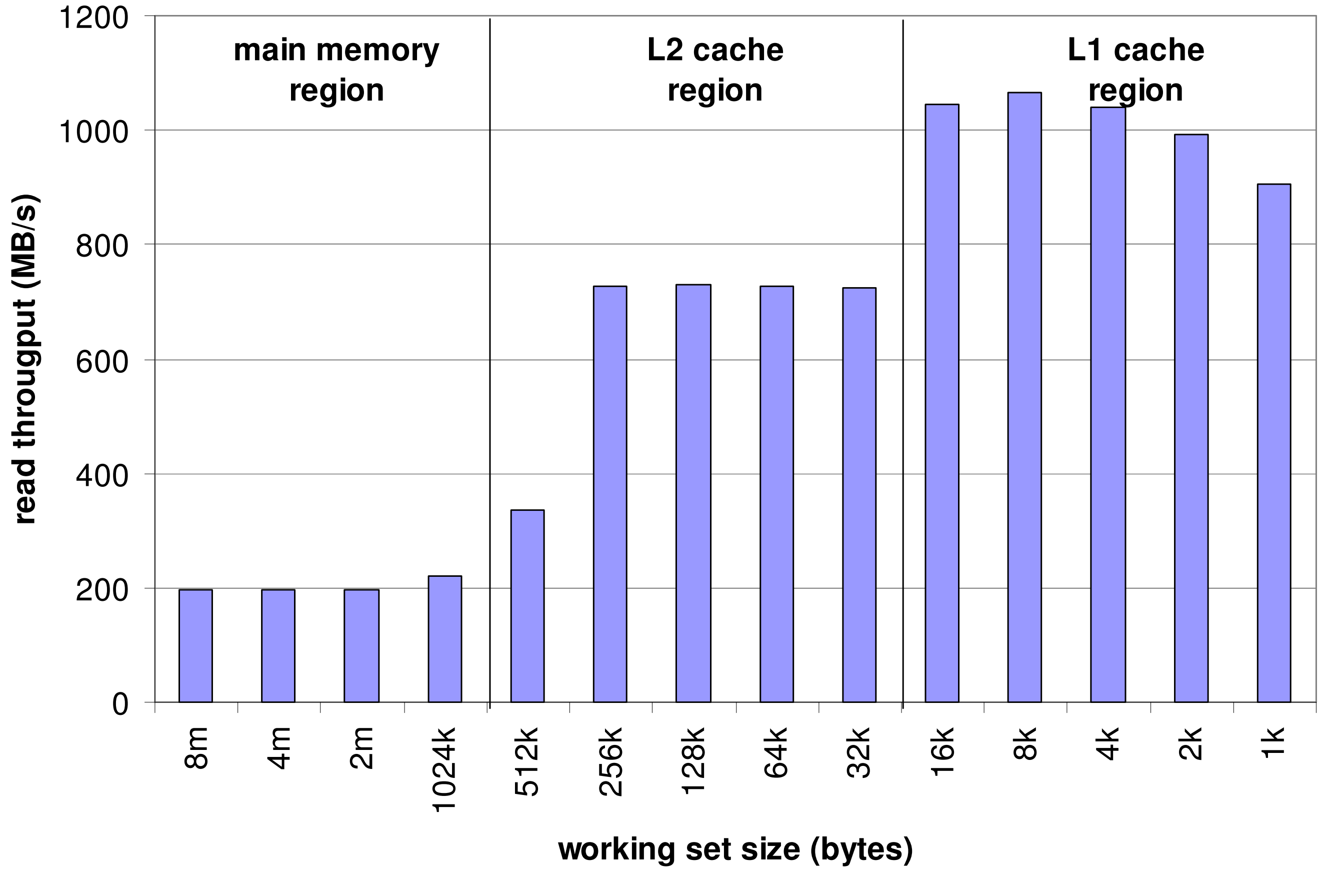


Pentium III Xeon
550 MHz
16 KB on-chip L1 d-cache
16 KB on-chip L1 i-cache
512 KB off-chip unified
L2 cache

Ridges of Temporal Locality

Slice through the memory mountain with stride=1

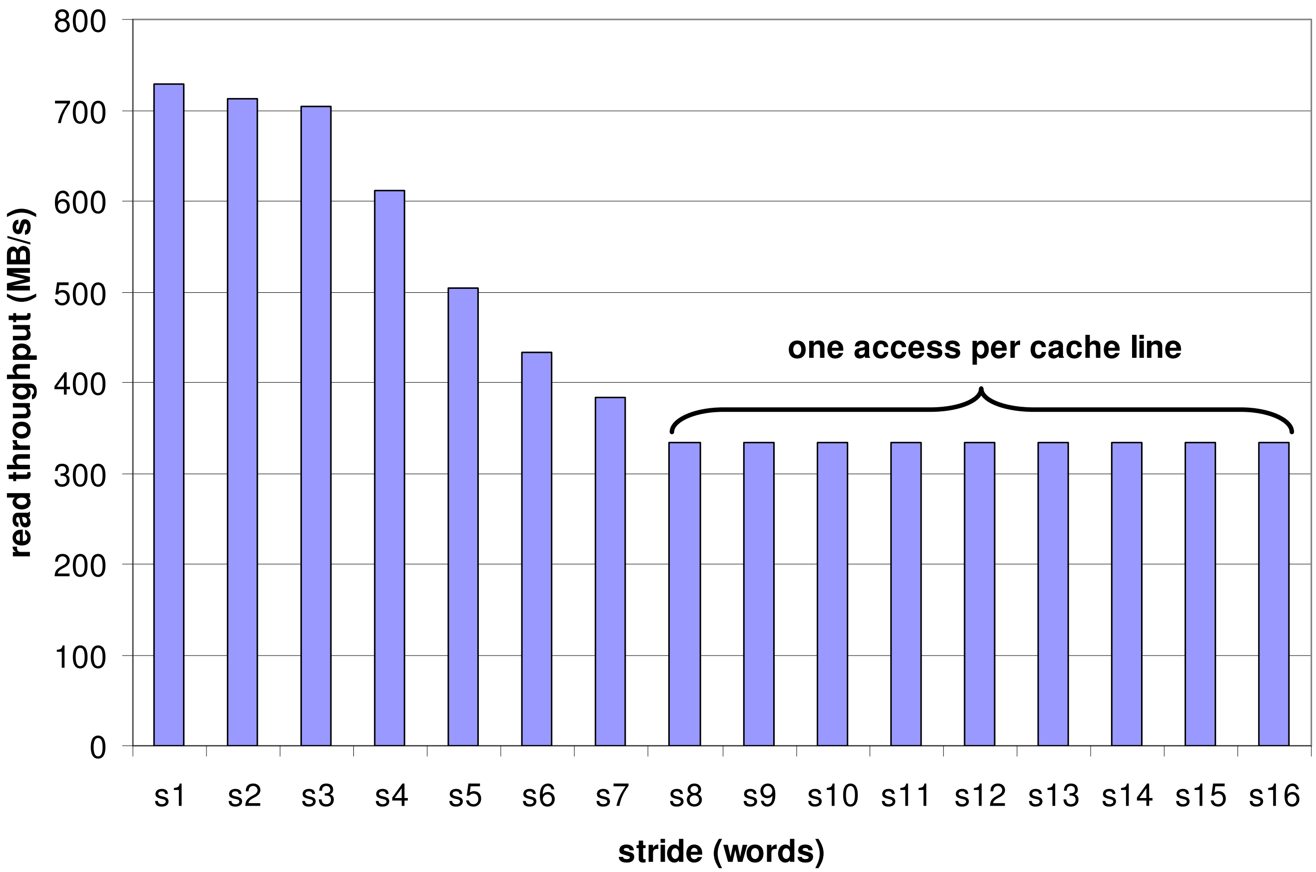
- **illuminates read throughputs of different caches and memory**



A Slope of Spatial Locality

Slice through memory mountain with size=256KB

- shows cache block size.



Consider the problem of computing Ax, where A is $n \times n$.

Partition A by columns:

$$y = (a_1 \; a_2 \; \cdots \; a_n) \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = x_1 a_1 + x_2 a_2 + \cdots + a_n x_n.$$

This gives the algorithm (in Matlab)

```
y = 0;
for j = 1:n
   y = y + x(j)*A(:,j);  % This is an AXPY.
end
```

In scalar form

```
for i=1:n
  y(i) = 0;
end
for j=1:n
   for i=1:n
      y(i) = y(i) + x(j)*A(i,j);
   end
end
```

Partition A in the form

$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} a_1^{\mathrm{T}} \\ a_2^{\mathrm{T}} \\ \vdots \\ a_n^{\mathrm{T}} \end{pmatrix} x = \begin{pmatrix} a_1^{\mathrm{T}} x \\ a_2^{\mathrm{T}} x \\ \vdots \\ a_n^{\mathrm{T}} x \end{pmatrix}$$

This gives the program

```
for i=1:n
   y(i) = A(i,:)*x;   % This is a DOT.
end
```

In scalar form

```
for i=1:n
   y(i) = 0;
   for j=1:n
      y(i) = y(i) + A(i,j)*x(j);
    end
end
```

MATRIX-MATRIX MULTIPLICATION (COLUMN ORIENTED)

Consider the product $C = AB$ and partition C and B by columns.

$$(c_1 \ c_2 \ \cdots \ c_n) = A(b_1 \ b_2 \ \cdots \ b_n) = (Ab_1 \ Ab_2 \ \cdots \ Ab_n).$$

We can then use AXPY's to compute the products Ab_k.

```
for k=1:n
   C(:,k) = 0;
   for j=1:n
      for i=1:n
         C(i,k) = C(i,k) + A(i,j)*B(j,k);
      end
   end
end
```

```
for k=1:n
   C(:,k) = 0;
   for j=1:n
      for i=1:n
         C(i,k) = C(i,k) + A(i,j)*B(j,k);
      end
   end
end
```

Locality of reference comes in two types.

Spatial locality means that when a memory reference to an address is made, it is surrounded by references to nearby addresses.

The above algorithm has good spatial locality (in a column oriented language) because it references its arrays by columns.

Temporal locality means that when a memory reference to an address is made, the address is reused frequently before it must be swapped out.

The above algorithm does not have good temporal locality because it makes n passes over the array A.

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(N3) total operations
- Accesses
  - N reads per source element
  - N values summed per destination
    - » but may be able to hold in register

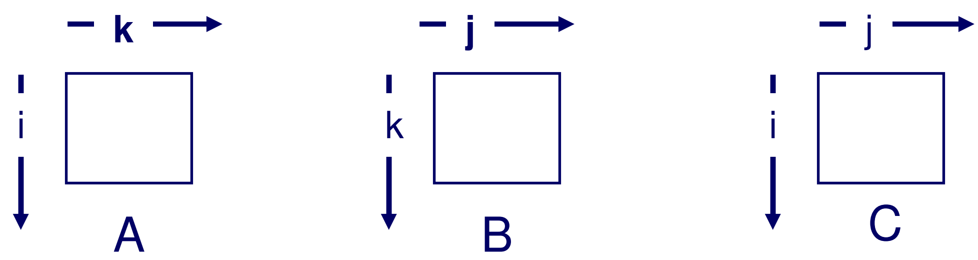
Miss Rate Analysis for Matrix Multiply

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 (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
- if block size (B) > 4 bytes, exploit spatial locality
  - compulsory miss rate = 4 bytes / B

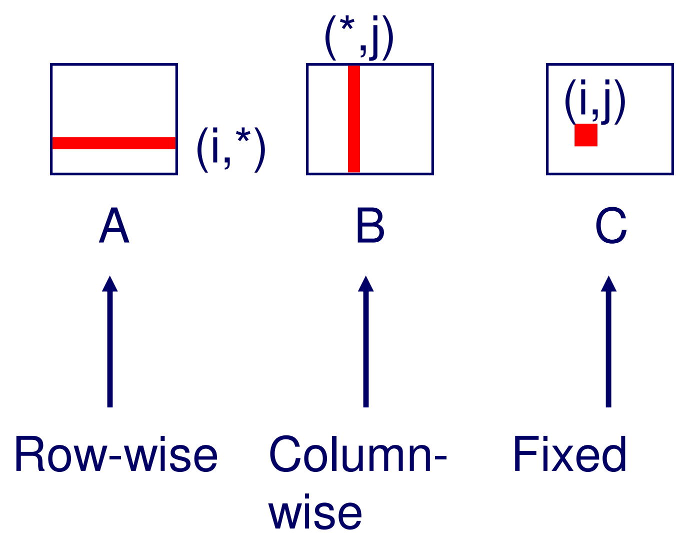
Stepping through rows in one column:

- ```
  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)

```
/* 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:



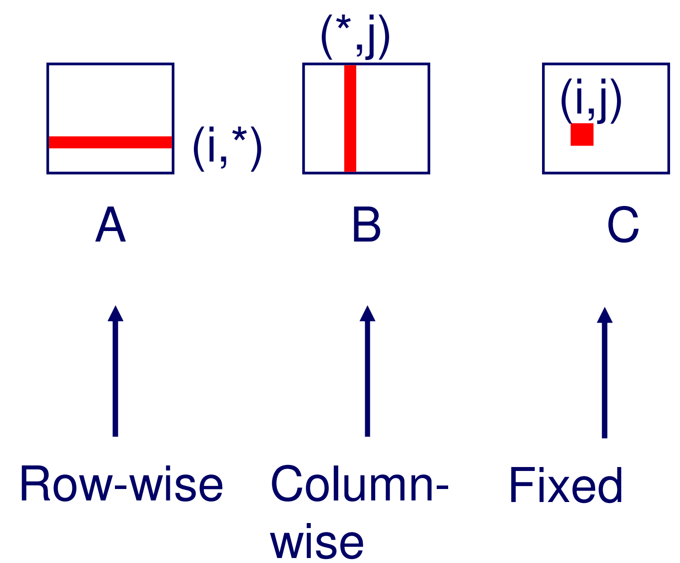
Misses per Inner Loop Iteration:

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:

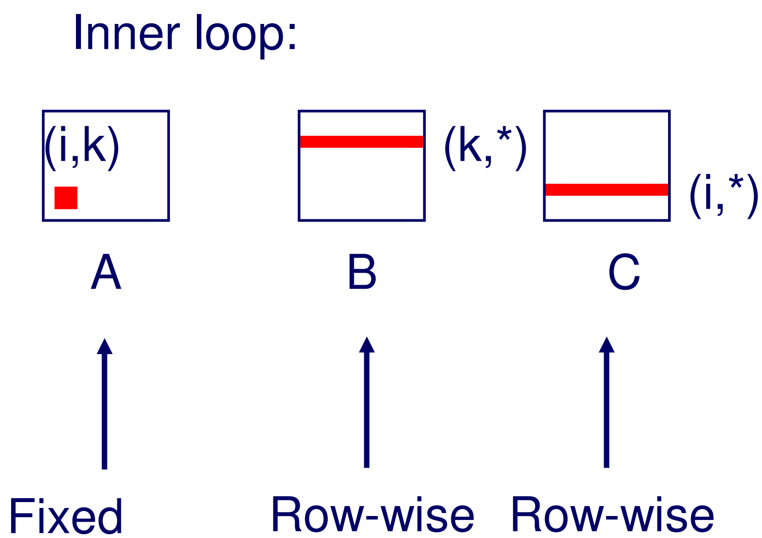


Misses per Inner Loop Iteration:

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];
  }
}
```

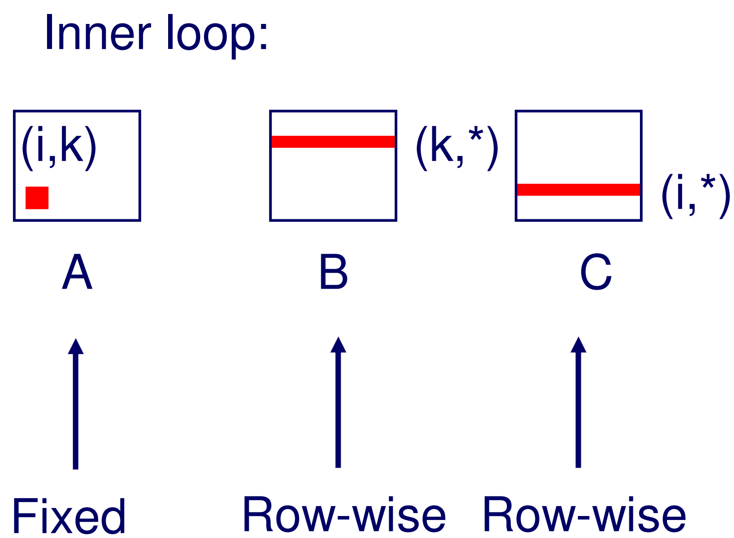


Misses per Inner Loop Iteration:

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];
  }
}
```

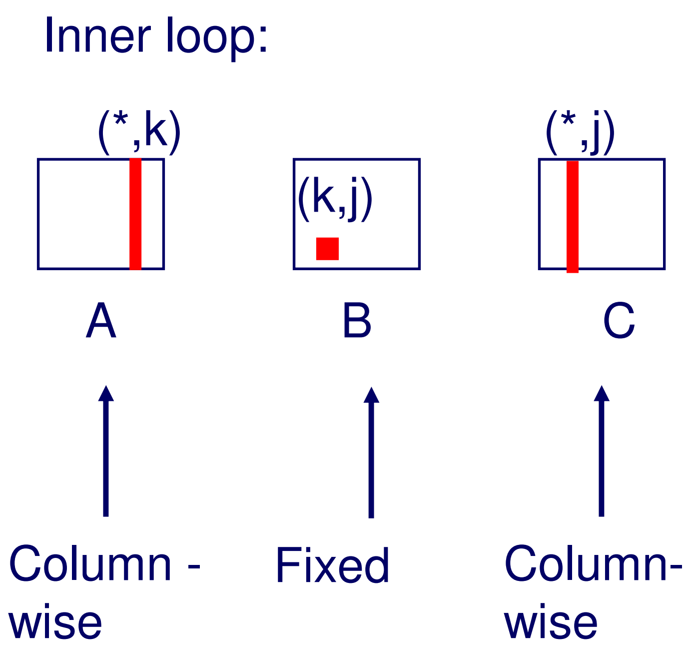


Misses per Inner Loop Iteration:

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;
  }
}
```



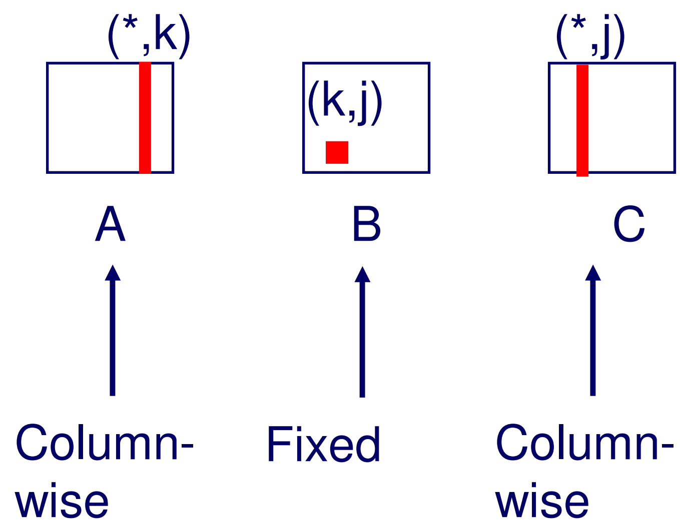
Misses per Inner Loop Iteration:

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:



Misses per Inner Loop Iteration:

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];
   }
}
```

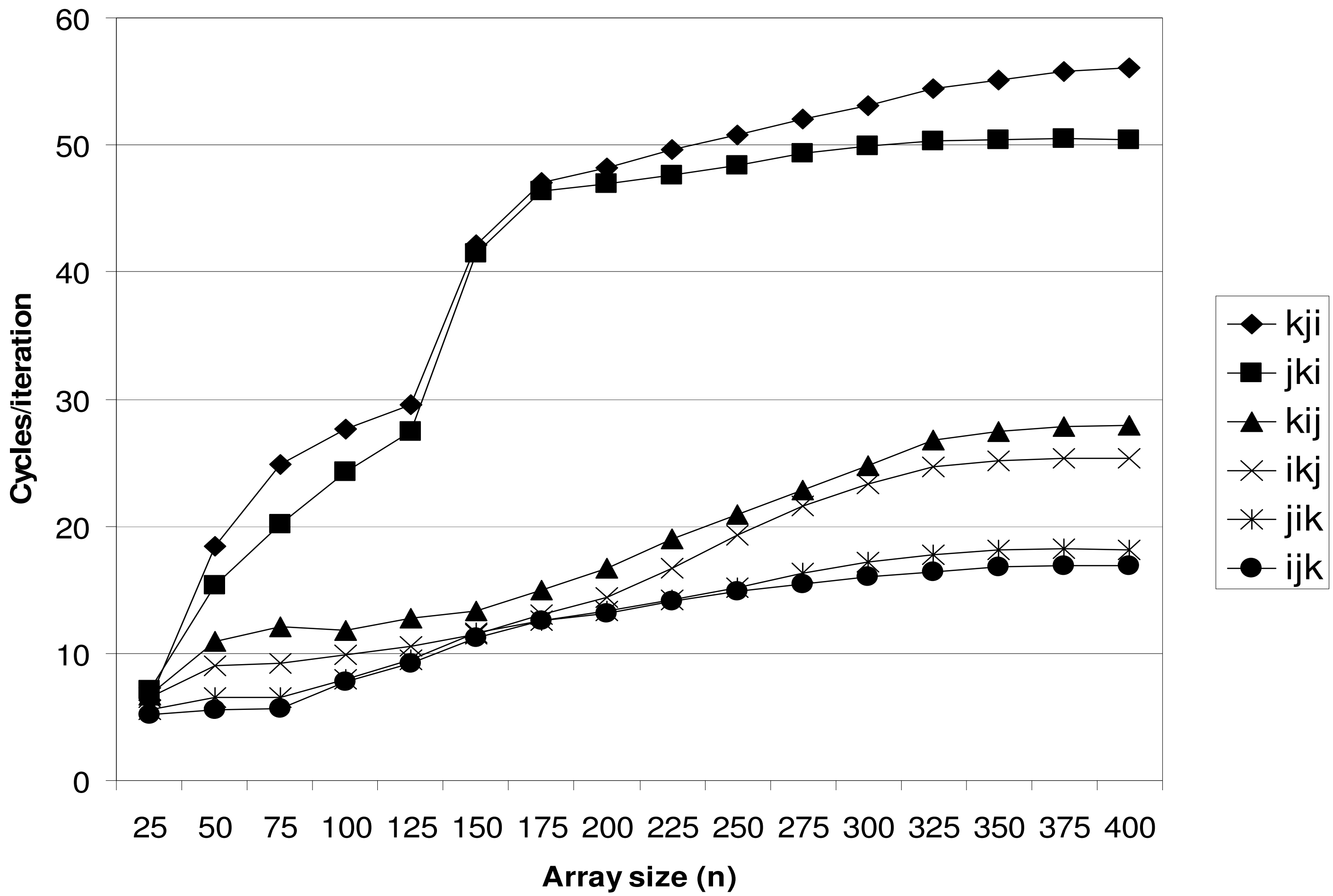
jki (& kji):
- 2 loads, 1 store
- misses/iter = **2.0**

```
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;
   }
}
```

Pentium Matrix Multiply Performance

Miss rates are helpful but not perfect predictors.

- Code scheduling matters, too.



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., $\mathbf{A_{xy}}$) can be treated just like scalars.

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

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

BLOCKING

We can increase temporal locality by blocking. Partition $C = AB$ in the form

$$\begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1\ell} \\ C_{21} & C_{22} & \cdots & C_{2\ell} \\ \vdots & \vdots & & \vdots \\ C_{\ell 1} & C_{\ell 2} & \cdots & C_{\ell\ell} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1\ell} \\ A_{21} & A_{22} & \cdots & A_{2\ell} \\ \vdots & \vdots & & \vdots \\ A_{\ell 1} & A_{\ell 2} & \cdots & A_{\ell\ell} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1\ell} \\ B_{21} & B_{22} & \cdots & B_{2\ell} \\ \vdots & \vdots & & \vdots \\ B_{\ell 1} & B_{\ell 2} & \cdots & B_{\ell\ell} \end{pmatrix},$$

where all the blocks are $m \times m$.

The our blocked algorithm is

```
for k=1:m:n
   kk = k+m-1;
   C(:,k:kk) = 0;
   for j=1:m:n
      jj = j+m-1;
      for i=1:m:n
         ii = i+m-1;
         C(i:ii,k:kk) = C(i:ii,k:kk)
                                 + A(i:ii,j:jj)*B(j:jj,k:kk);
      end
   end
end
```

BLOCKING CONTINUED

```
for k=1:m:n
   kk = k+m-1;
   C(:,k:kk) = 0;
   for j=1:m:n
      jj = j+m-1;
      for i=1:m:n
         ii = i+m-1;
         C(i:ii,k:kk) = C(i:ii,k:kk)
                      + A(i:ii,j:jj)*B(j:jj,k:kk);
      end
   end
end
```

If the blocks `A(i:ii,j:jj)`, `B(j:jj,k:kk)`, an `C(i:ii,k:kk)` all fit into cache, then it does not matter how they are multiplied.

We now make only n/m passes over A.

Blocked Matrix Multiply (bijk)

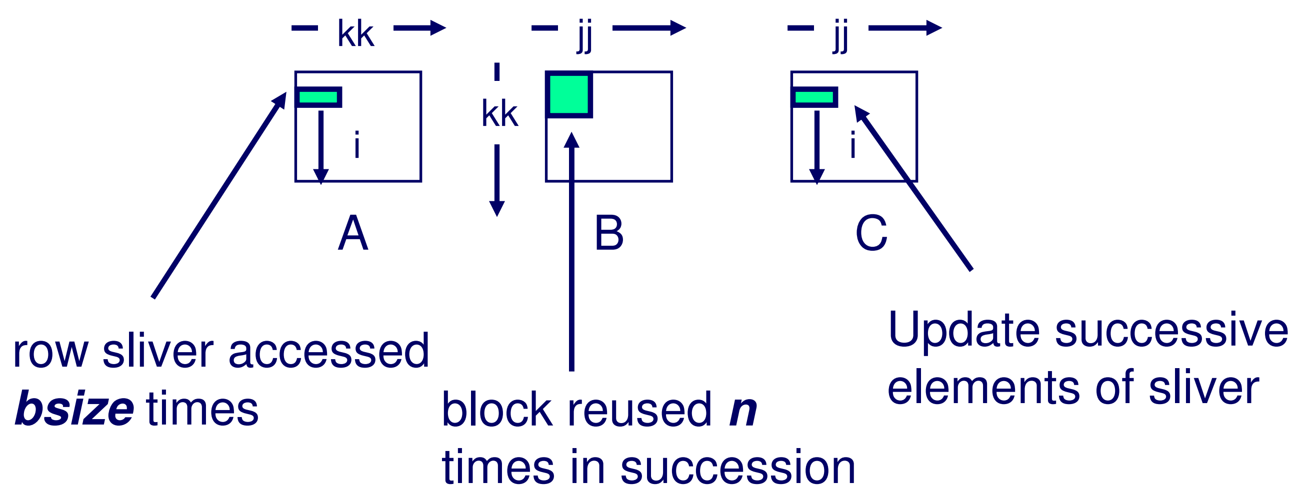
```
for (jj=0; jj<n; jj+=bsize) {
  for (i=0; i<n; i++)
    for (j=jj; j < min(jj+bsize,n); j++)
      c[i][j] = 0.0;
  for (kk=0; kk<n; kk+=bsize) {
    for (i=0; i<n; i++) {
      for (j=jj; j < min(jj+bsize,n); j++) {
        sum = 0.0
        for (k=kk; k < min(kk+bsize,n); k++) {
          sum +=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

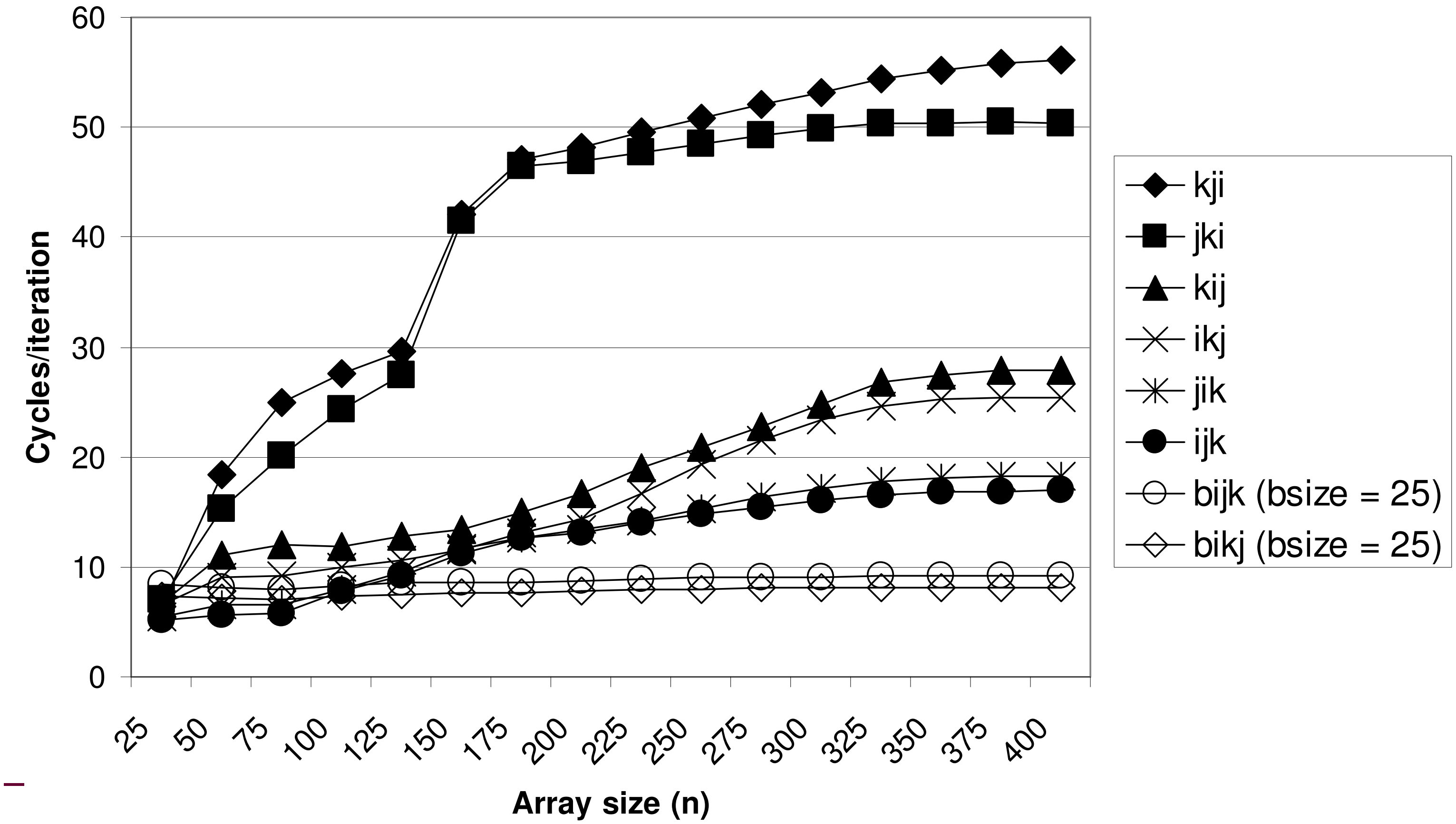
Innermost Loop Pair



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)