15-213

“The course that gives CMU its Zip!”

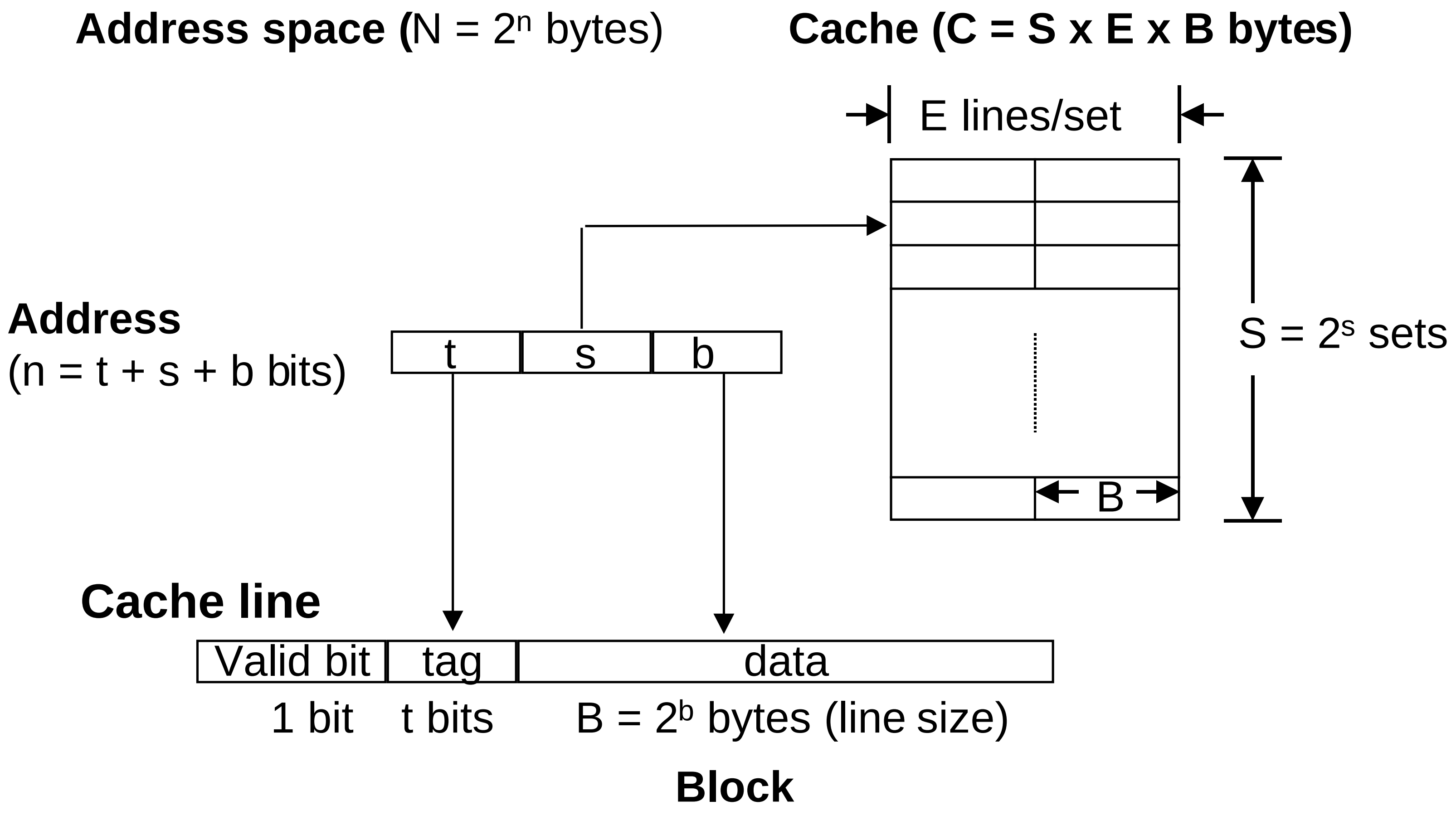
Memory System Performance

October 17, 2000

Topics

- **Impact of cache parameters**
- **Impact of memory reference patterns**
  - – memory mountain range
  - – matrix multiply

Basic Cache Organization



Address space (N = 2^n bytes)

Cache (C = S x E x B bytes)

E lines/set

S = 2^s sets

B

Address

(n = t + s + b bits)

t s b

Cache line

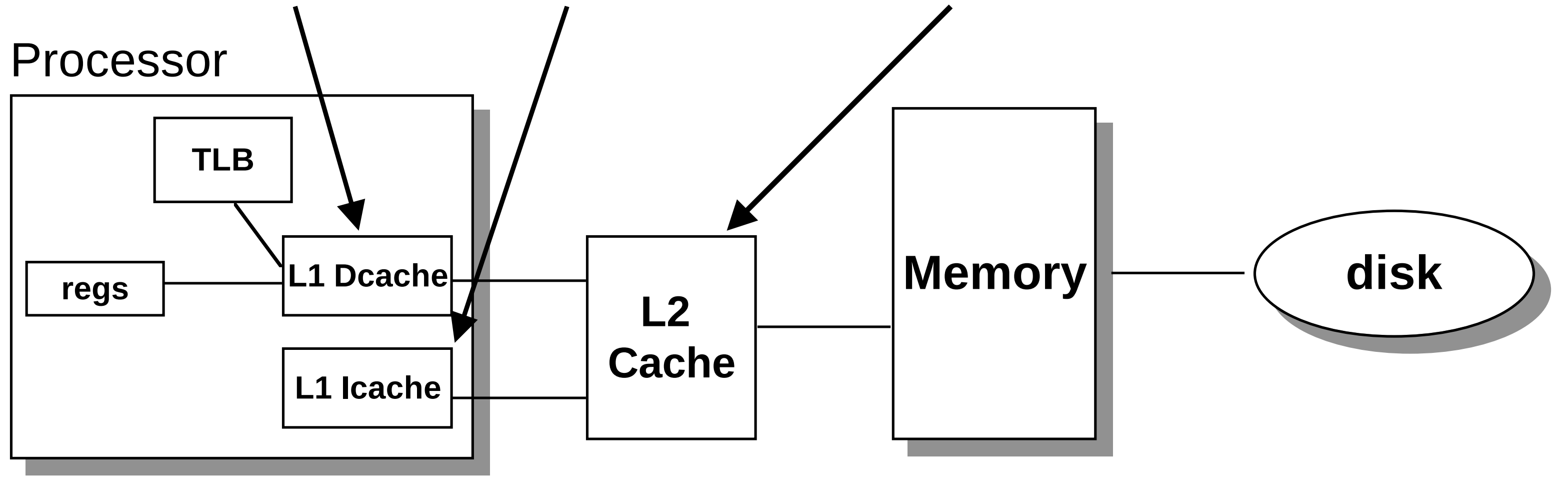
Valid bit | tag | data

1 bit t bits B = 2^b bytes (line size)

Block

Multi-Level Caches

Options: ***separate*** data and instruction caches, or a ***unified*** cache



size:	200 B	8-64 KB	1-4MB SRAM	128 MB DRAM	30 GB
speed:	3 ns	3 ns	4 ns	60 ns	8 ms
$/Mbyte:			$100/MB	$1.50/MB	$0.05/MB
line size:	8 B	32 B	32 B	8 KB	

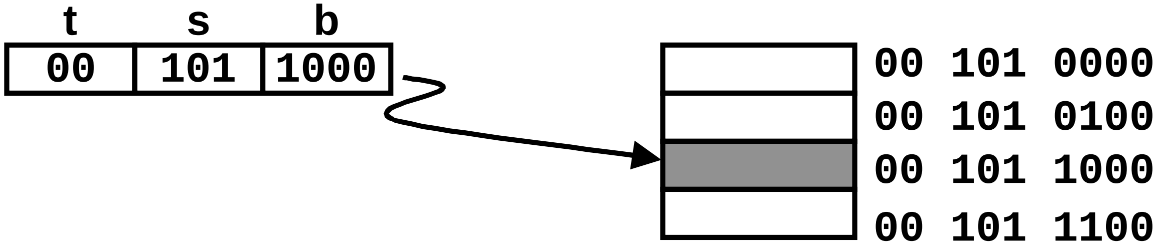
larger, slower, cheaper →

larger line size, higher associativity, more likely to write back

Key Features of Caches

Accessing Word Causes Adjacent Words to be Cached

- B bytes having same bit pattern for upper n–b address bits



- In anticipation that will want to reference these words due to spatial locality in program

Loading Block into Cache Causes Existing Block to be Evicted

- One that maps to same set
- If E > 1, then generally choose least recently used

Cache Performance Metrics

Miss Rate

- **Fraction of memory references not found in cache**
  - misses/references
- **Typical numbers:**

  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 clock cycle for L1

  3-8 clock cycles for L2

Miss Penalty

- **Additional time required because of a miss**
  - Typically 25-100 cycles for main memory

Categorizing Cache Misses

Compulsory ("Cold") Misses:

- First ever access to a memory line
  - since lines are only brought into the cache *on demand*, this is guaranteed to be a cache miss
- **Programmer/system cannot reduce these**

Capacity Misses:

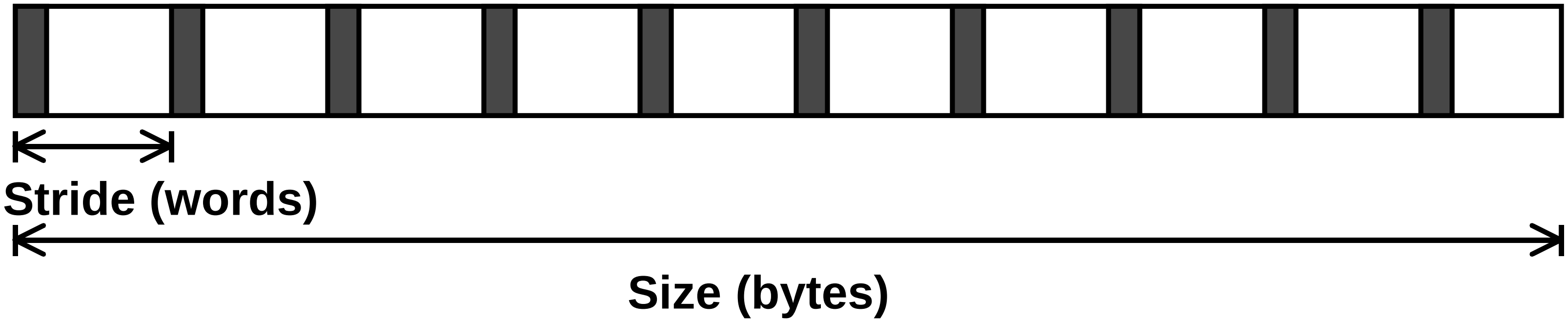
- Active portion of memory exceeds the cache size
- Programmer can reduce by rearranging data & access patterns

Conflict Misses:

- Active portion of address space fits in cache, but too many lines map to the same cache entry
- Programmer can reduce by changing data structure sizes
  - **Avoid powers of 2**

Measuring Memory Bandwidth

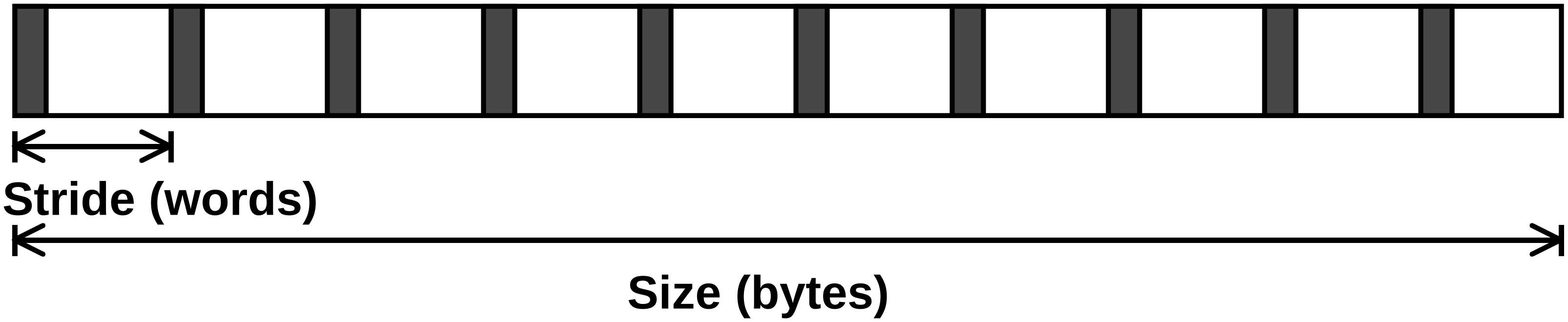
```
int data[MAXSIZE];
int test(int size, int stride)
{
  int result = 0;
  int wsize = size/sizeof(int);
  for (i = 0; i < wsize; i+= stride)
    result += data[i];
  return result;
}
```



Stride (words)

Size (bytes)

Measuring Memory Bandwidth (cont.)



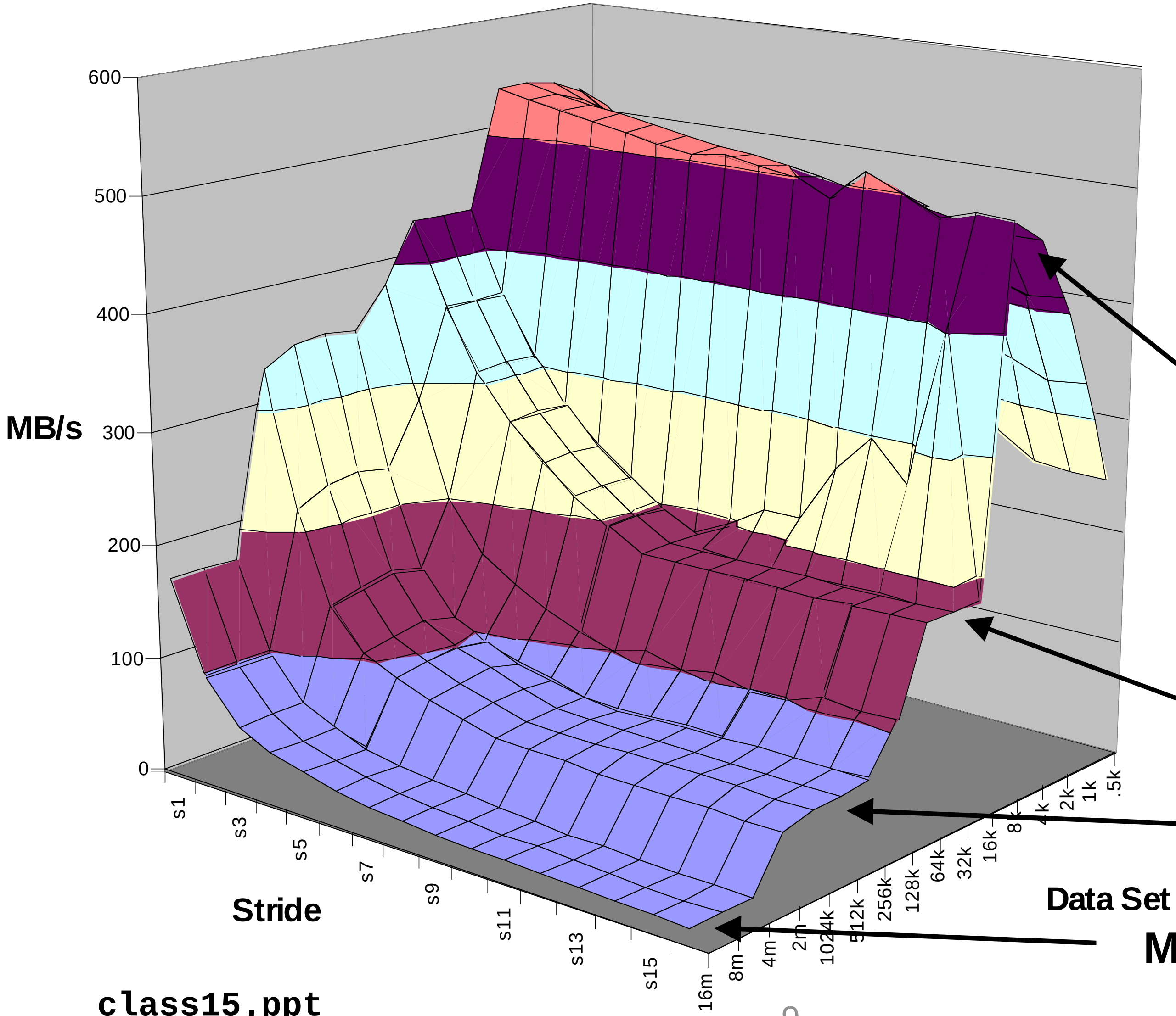
Measurement

- **Time repeated calls to `test`**
  - – If size sufficiently small, then can hold array in cache

Characteristics of Computation

- **Stresses read bandwidth of system**
- **Increasing stride yields decreased spatial locality**
  - – On average will get stride*4/B accesses / cache block
- **Increasing size increases size of “working set”**

Alpha Memory Mountain Range



DEC Alpha 21164
466 MHz
8 KB (L1)
96 KB (L2)
2 M (L3)

L1 Resident

L2 Resident

L3 Resident

Main Memory Resident

Effects Seen in Mountain Range

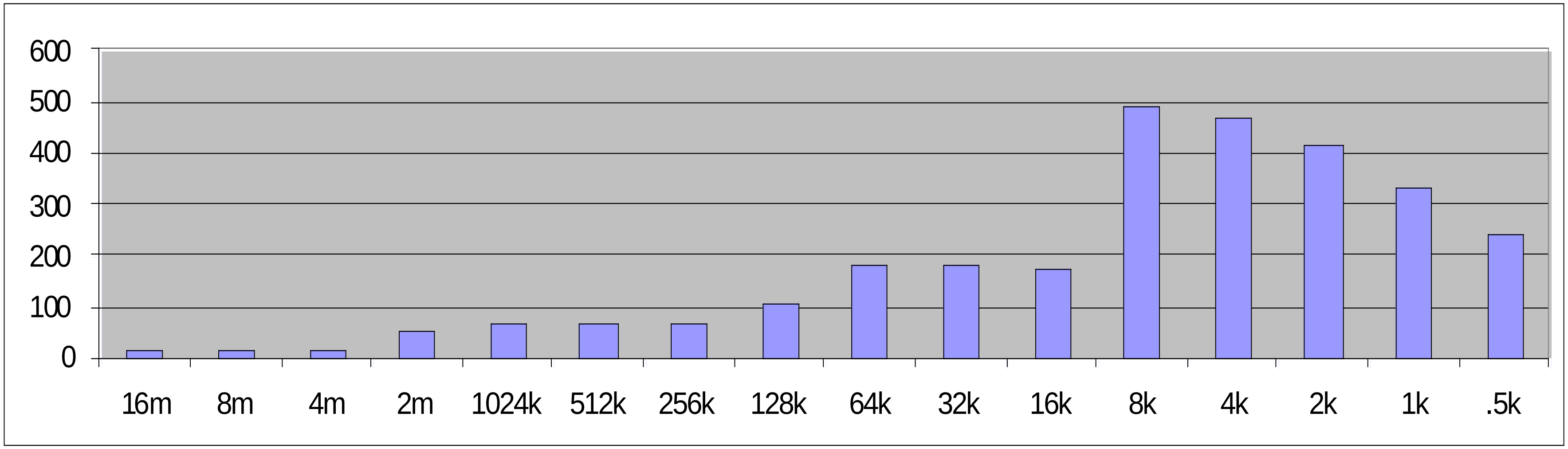
Cache Capacity

- **See sudden drops as increase working set size**

Cache Block Effects

- **Performance degrades as increase stride**
  - Less spatial locality
- **Levels off**
  - When reach single access per line

Alpha Cache Sizes

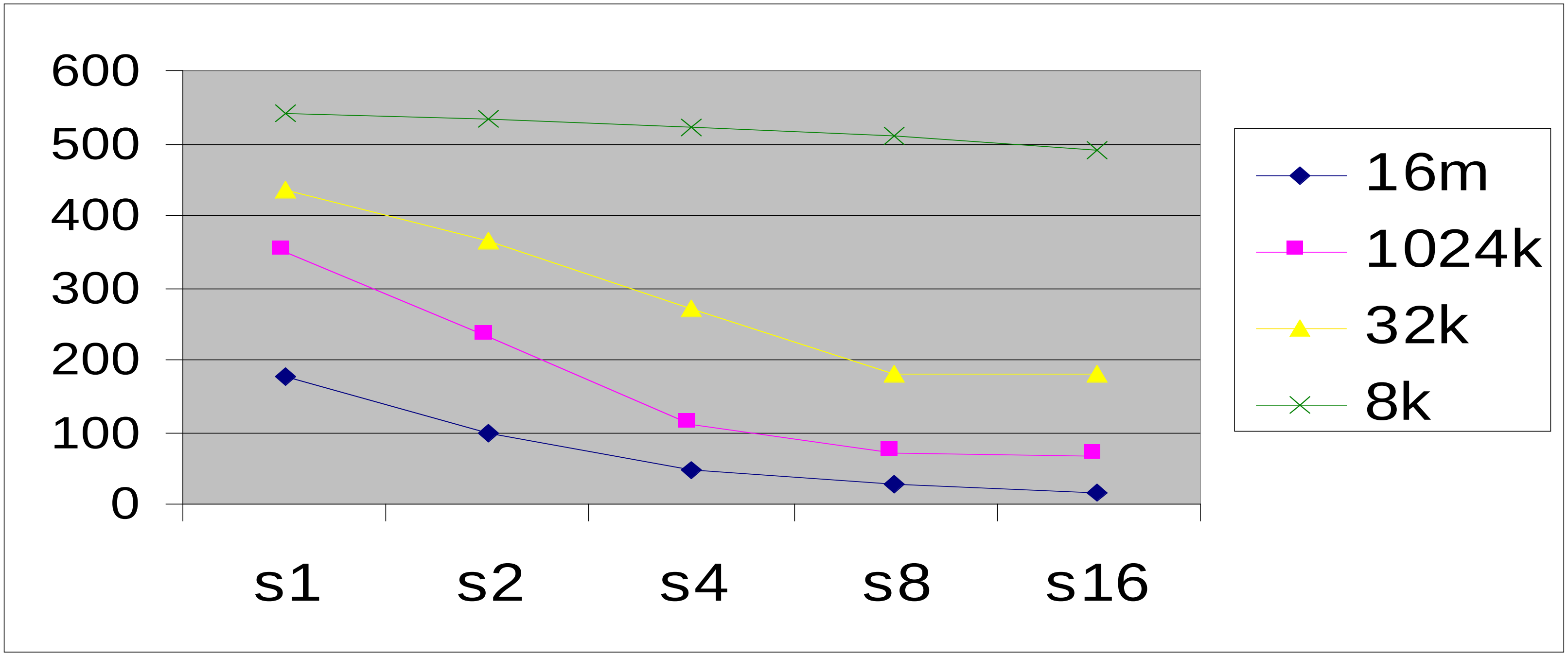


- **MB/s for stride = 16**

Ranges

.5k – 8k	**Running in L1 (High overhead for small data set)**
16k – 64k	**Running in L2.**
128k	**Indistinct cutoff (Since cache is 96KB)**
256k – 2m	**Running in L3.**
4m – 16m	**Running in main memory**

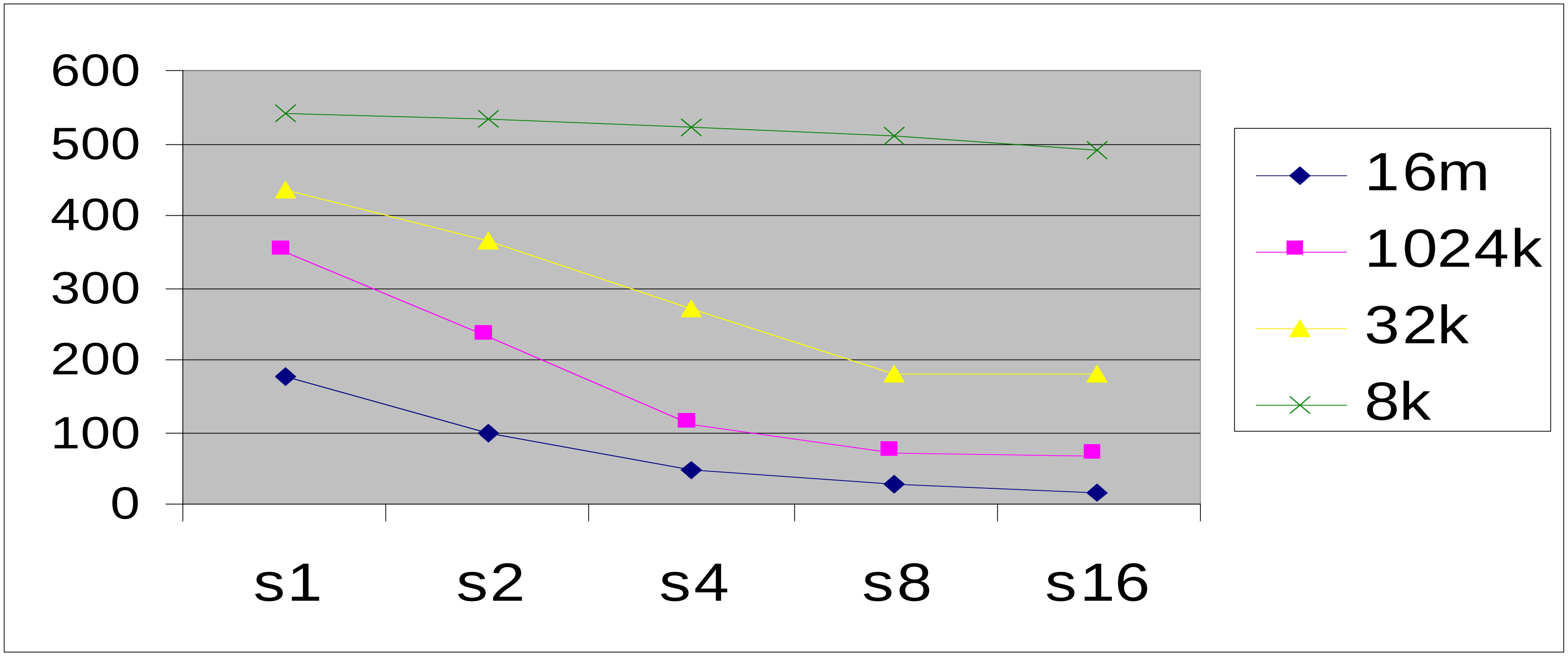
Alpha Line Size Effects



Observed Phenomenon

- **As double stride, decrease accesses/block by 2**
- **Until reaches point where just 1 access / block**
- **Line size at transition from downward slope to horizontal line**
  - Sometimes indistinct

Alpha Line Sizes



Measurements

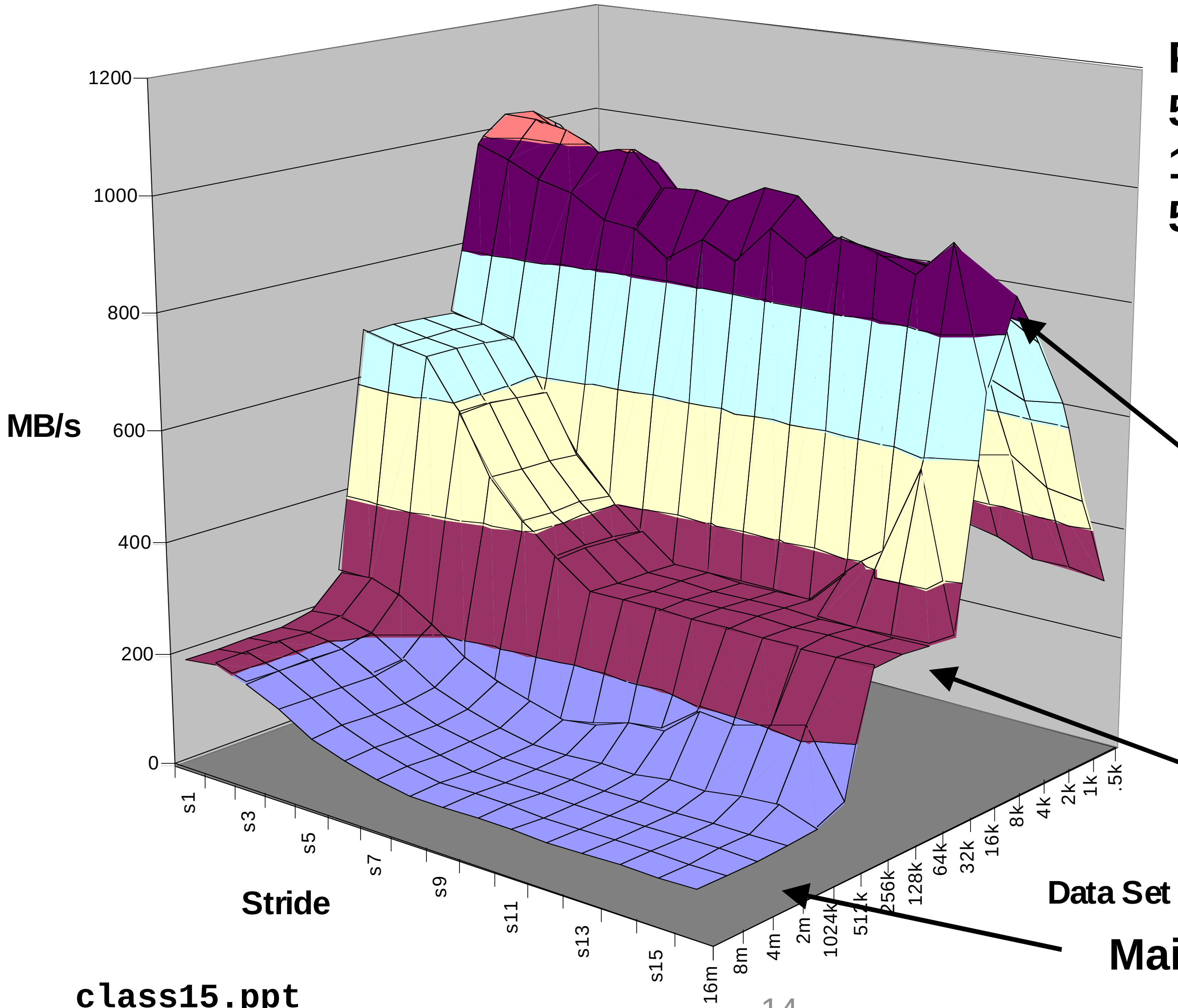
8k Entire array L1 resident. Effectively flat (except for overhead)

32k Shows that L1 line size = 32B

1024k Shows that L2 line size = 32B

16m L3 line size = 64?

Xeon Memory Mountain Range



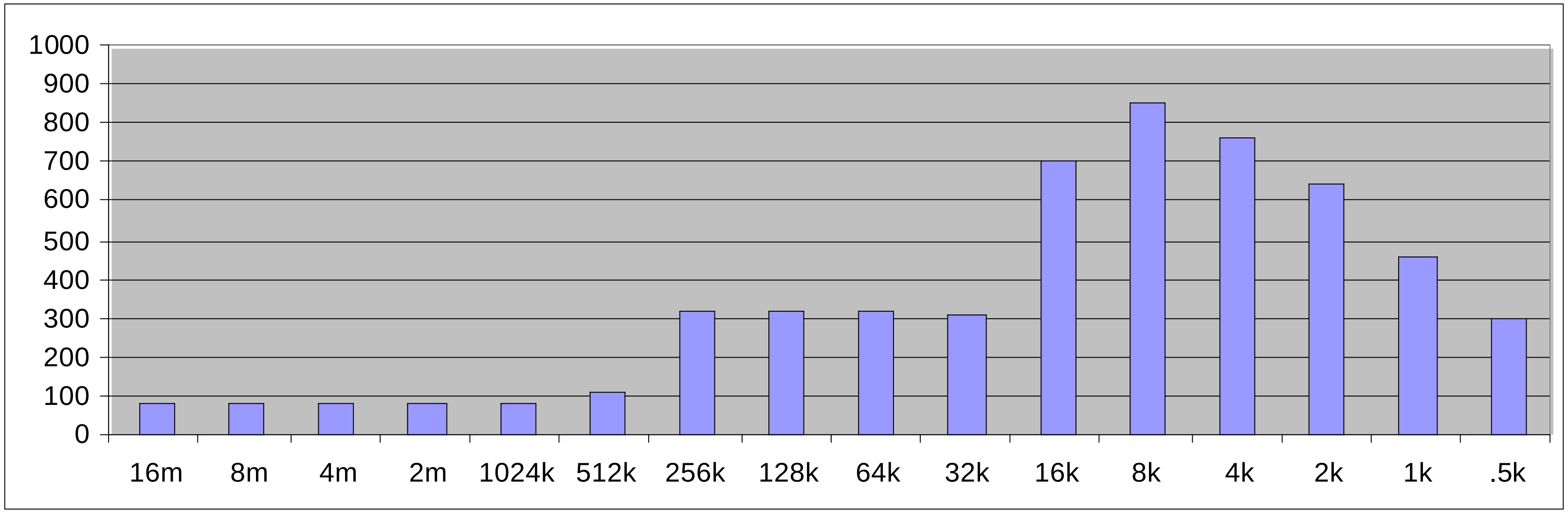
Pentium III Xeon
550 MHz
16 KB (L1)
512 KB (L2)

L1 Resident

L2 Resident

Main Memory Resident

Xeon Cache Sizes

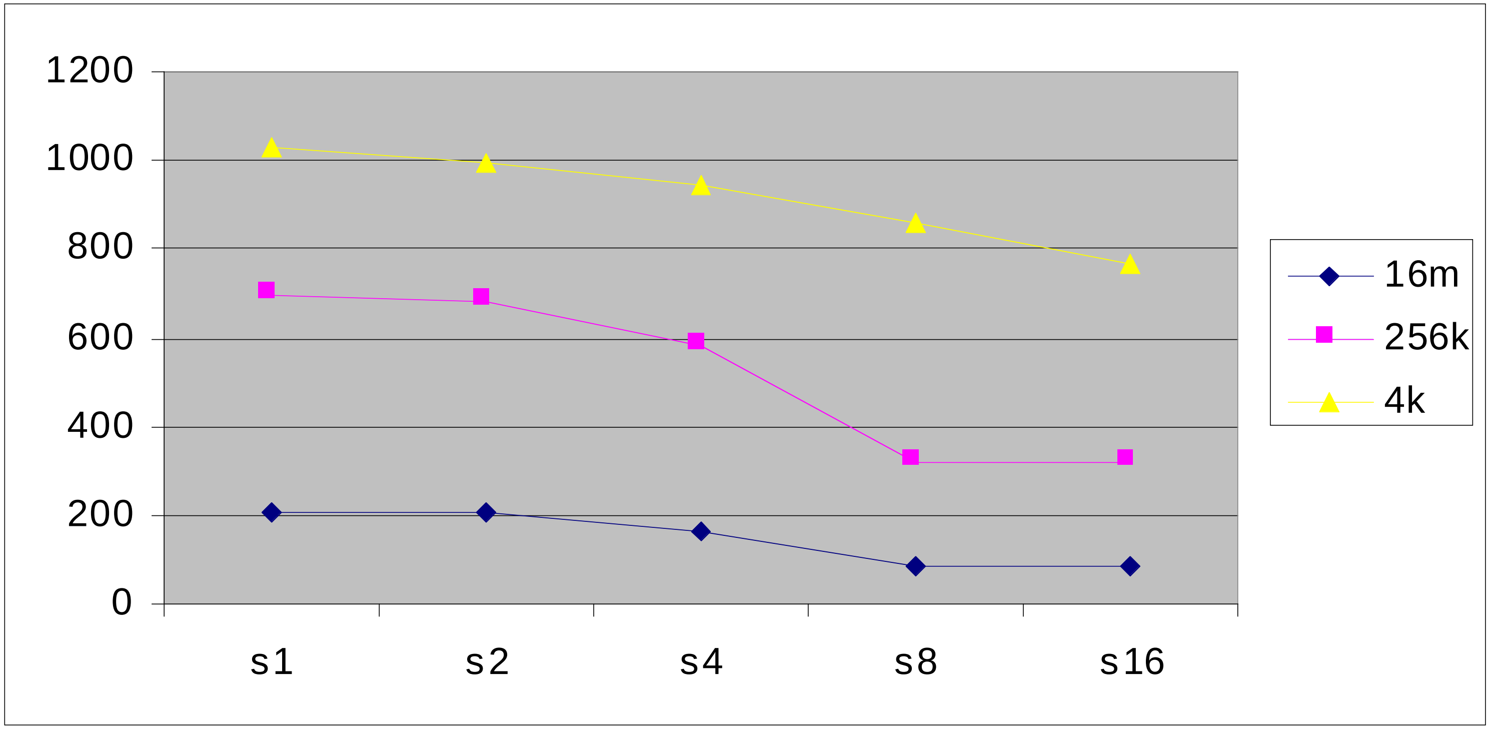


- **MB/s for stride = 16**

Ranges

.5k – 16k	**Running in L1. (Overhead at high end)**
32k – 256k	**Running in L2.**
512k	**Running in main memory (but L2 supposed to be 512K!)**
1m – 16m	**Running in main memory**

Xeon Line Sizes



Measurements

4k	**Entire array L1 resident. Effectively flat (except for overhead)**
256k	**Shows that L1 line size = 32B**
16m	**Shows that L2 line size = 32B**

Interactions Between Program & Cache

Major Cache Effects to Consider

- **Total cache size**
  - – Try to keep heavily used data in cache closest to processor
- **Line size**
  - – Exploit spatial locality

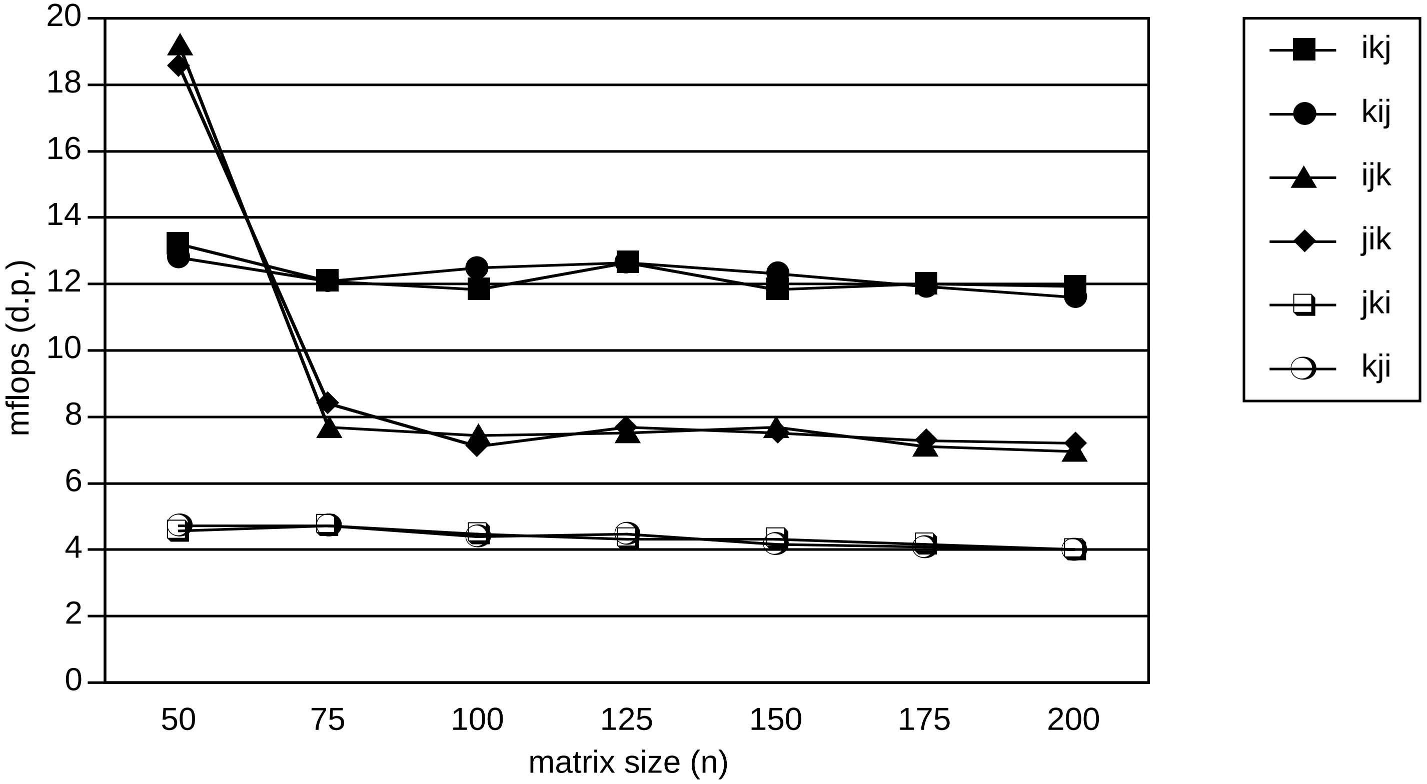
Variable ***sum***
held in register

```
/* 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;
  }
}
```

Example Application

- **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

Matrix Mult. Performance: Sparc20



- **As matrices grow in size, they eventually exceed cache capacity**
- **Different loop orderings give different performance**
  - – cache effects
  - – whether or not we can accumulate partial sums in registers

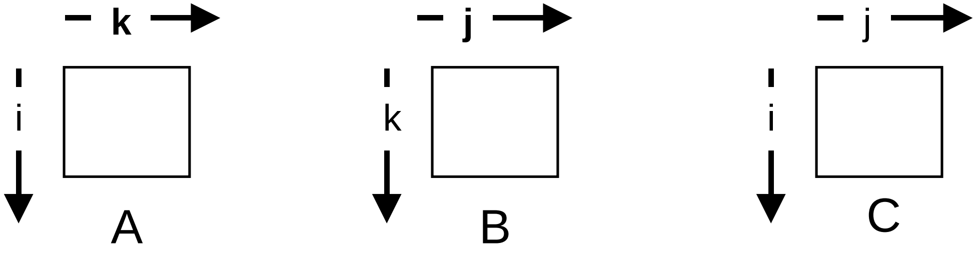
Miss Rate Analysis for Matrix Multiply

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 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];
```

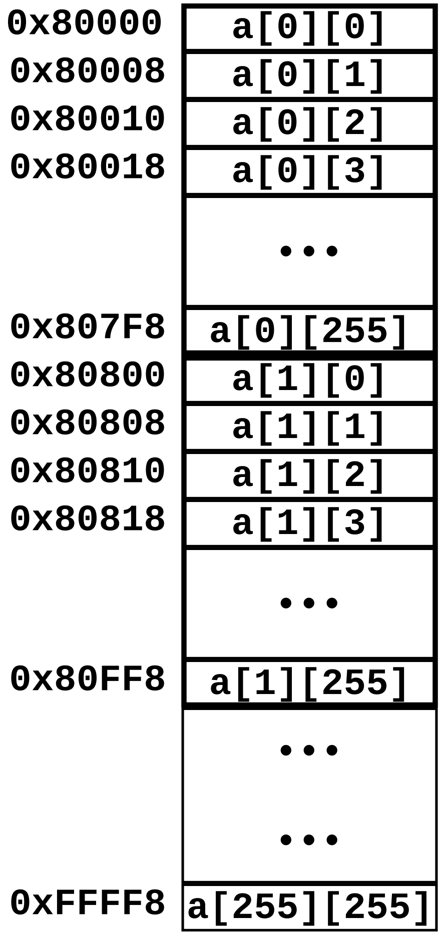
- accesses successive elements
- if line size (B) > 8 bytes, exploit spatial locality
  - compulsory miss rate = 8 bytes / B

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%)

Memory Layout

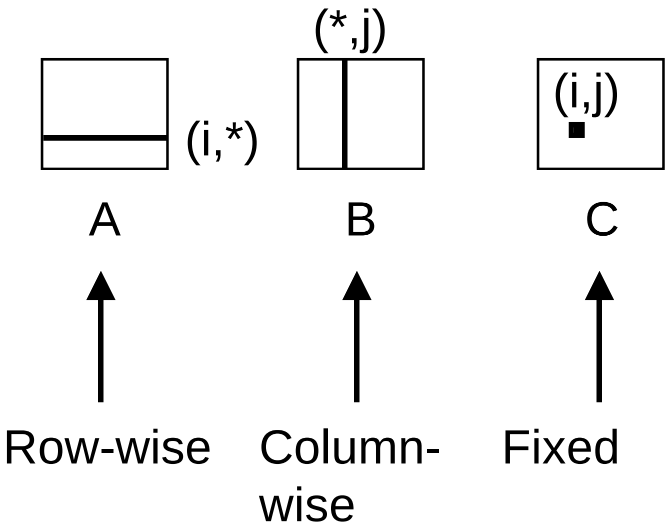


0x80000	a[0][0]
0x80008	a[0][1]
0x80010	a[0][2]
0x80018	a[0][3]
	• • •
0x807F8	a[0][255]
0x80800	a[1][0]
0x80808	a[1][1]
0x80810	a[1][2]
0x80818	a[1][3]
	• • •
0x80FF8	a[1][255]
	• • • • • •
0xFFFF8	a[255][255]

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:



(*,j)

(i,*)

(i,j)

A

B

C

Row-wise

Column-wise

Fixed

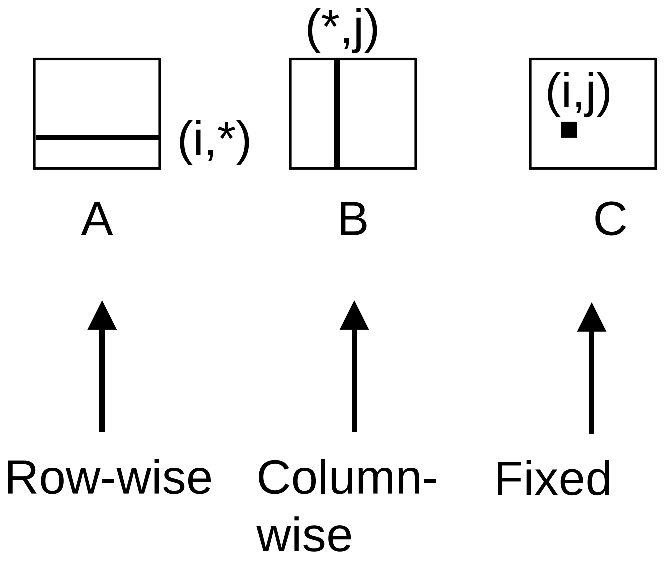
Misses per Inner Loop Iteration:

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:



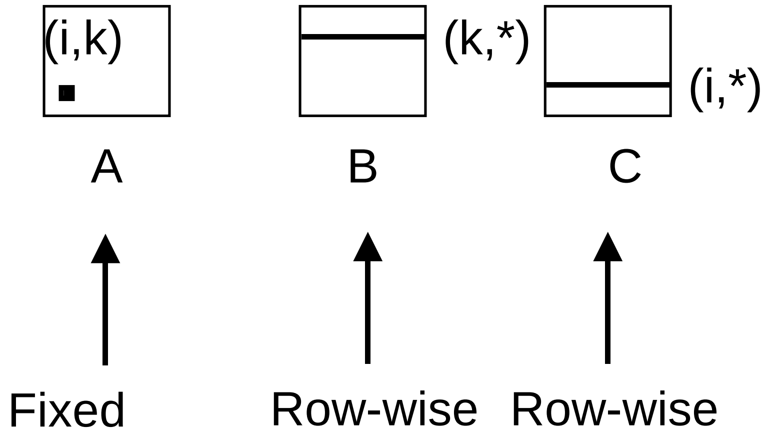
Misses per Inner Loop Iteration:

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:



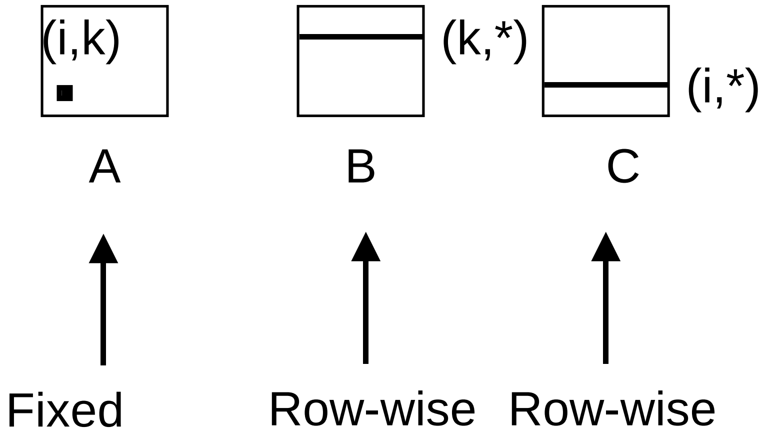
Misses per Inner Loop Iteration:

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];
  }
}
```

Inner loop:



(i,k)

(k,*)

(i,*)

A

B

C

Fixed

Row-wise

Row-wise

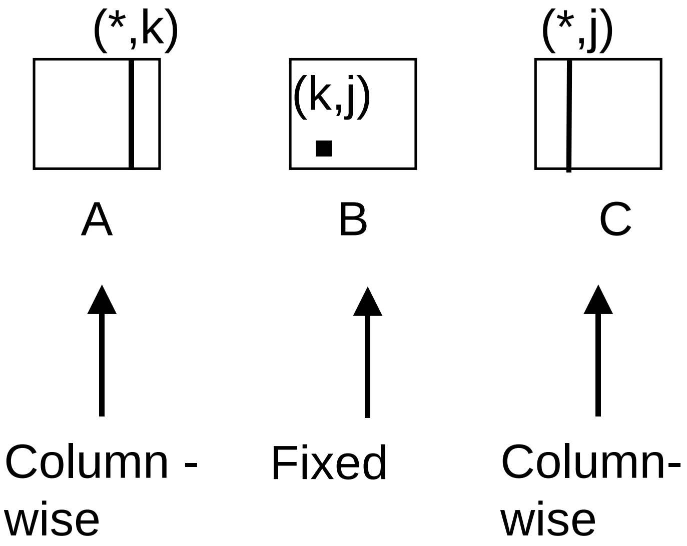
Misses per Inner Loop Iteration:

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:



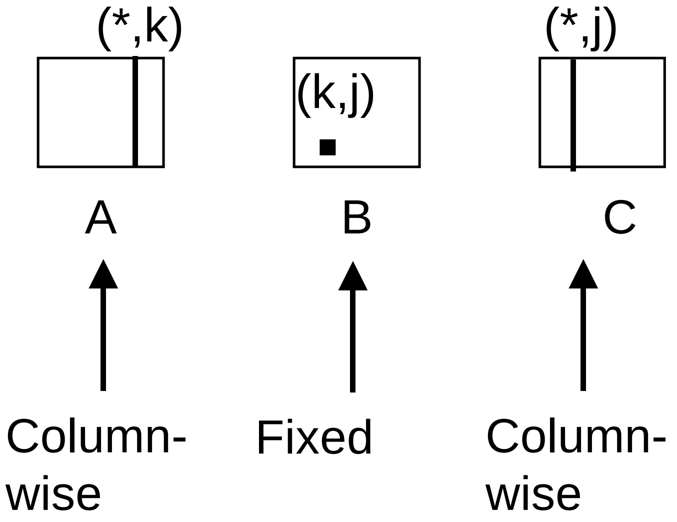
Misses per Inner Loop Iteration:

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:



(*,k)

(k,j)

(*,j)

A

B

C

Column-wise

Fixed

Column-wise

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**

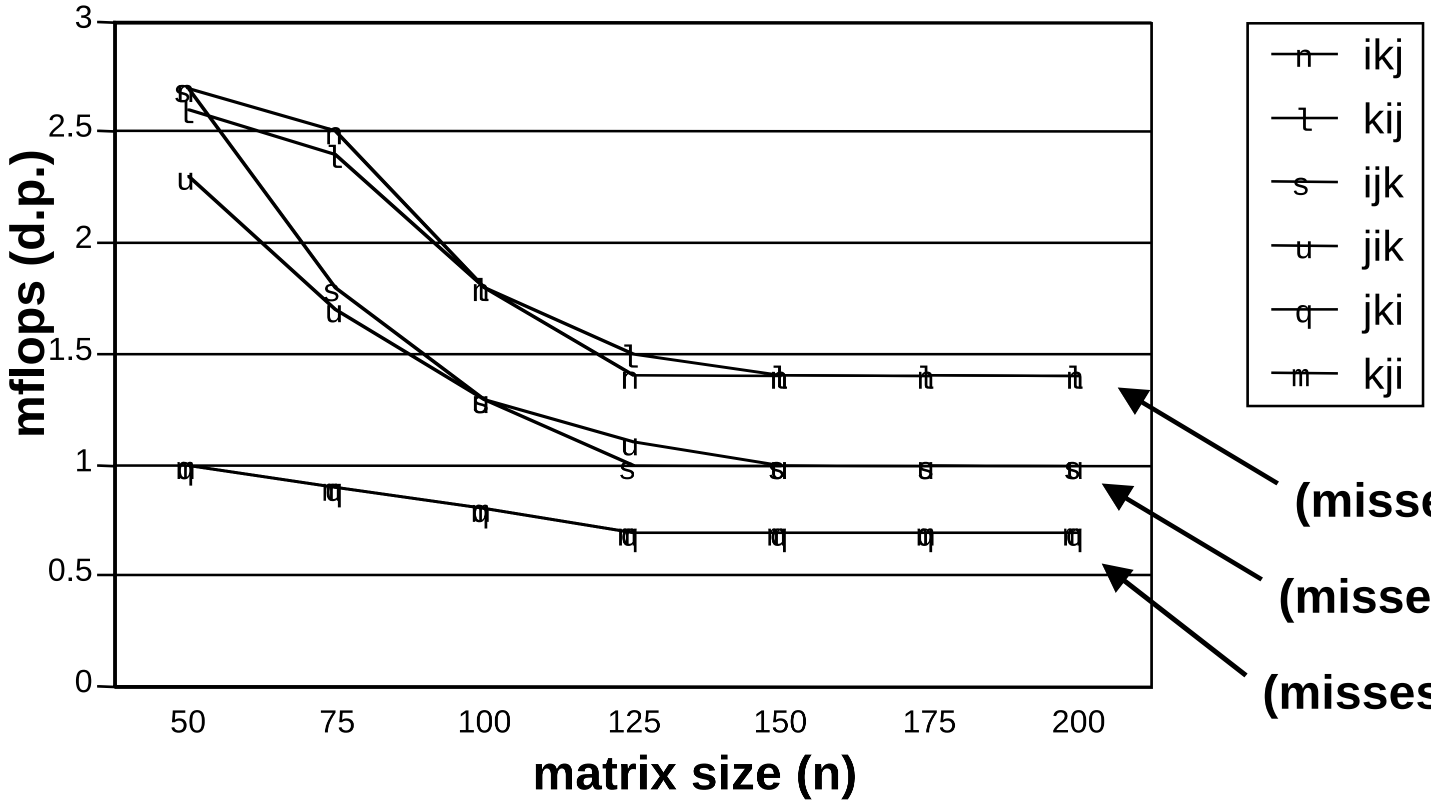
```
for (k=0; k<n; k++) {
   for (i=0; i<n; i++) {
      r = a[i][k];
      for (j=0; j<n; j++)
         c[i][j] += r * b[k][j];
   }
}
```

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;
   }
}
```

Matrix Mult. Performance: DEC5000

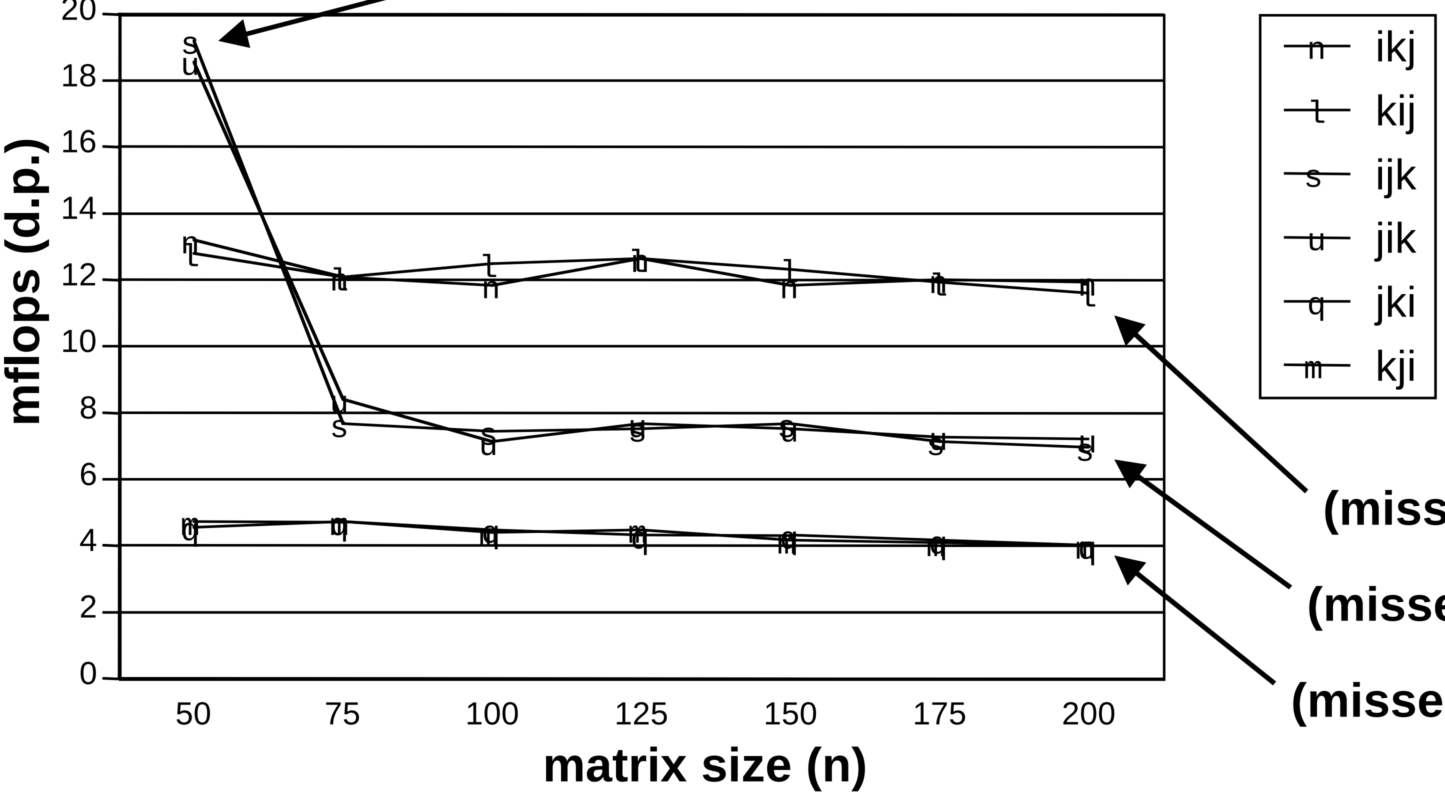


(misses/iter = 0.5)

(misses/iter = 1.25)

(misses/iter = 2.0)

Matrix Mult. Performance: Sparc20



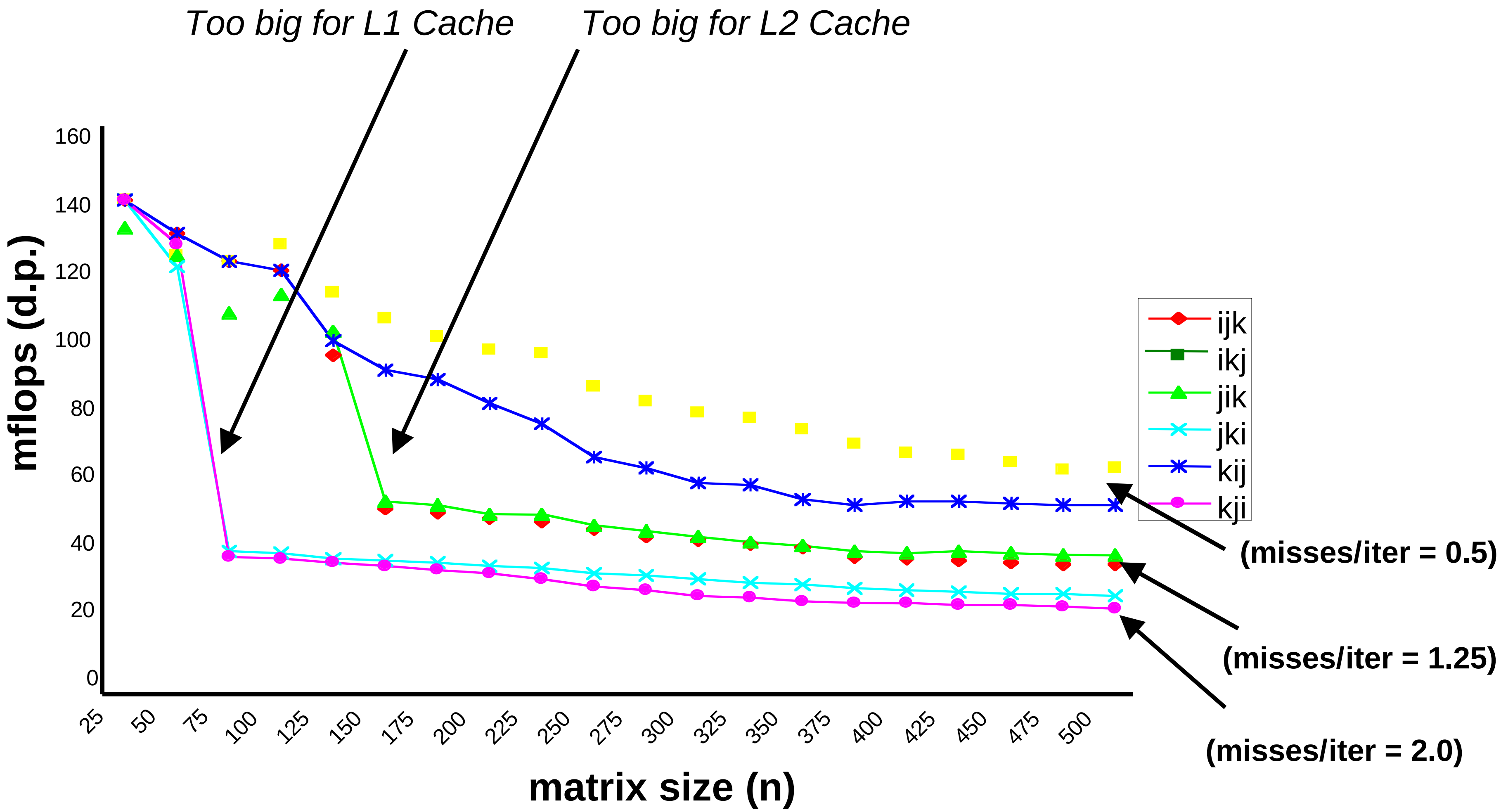
Multiple columns of B fit in cache

(misses/iter = 0.5)

(misses/iter = 1.25)

(misses/iter = 2.0)

Matrix Mult. Performance: Alpha 21164



Too big for L1 Cache

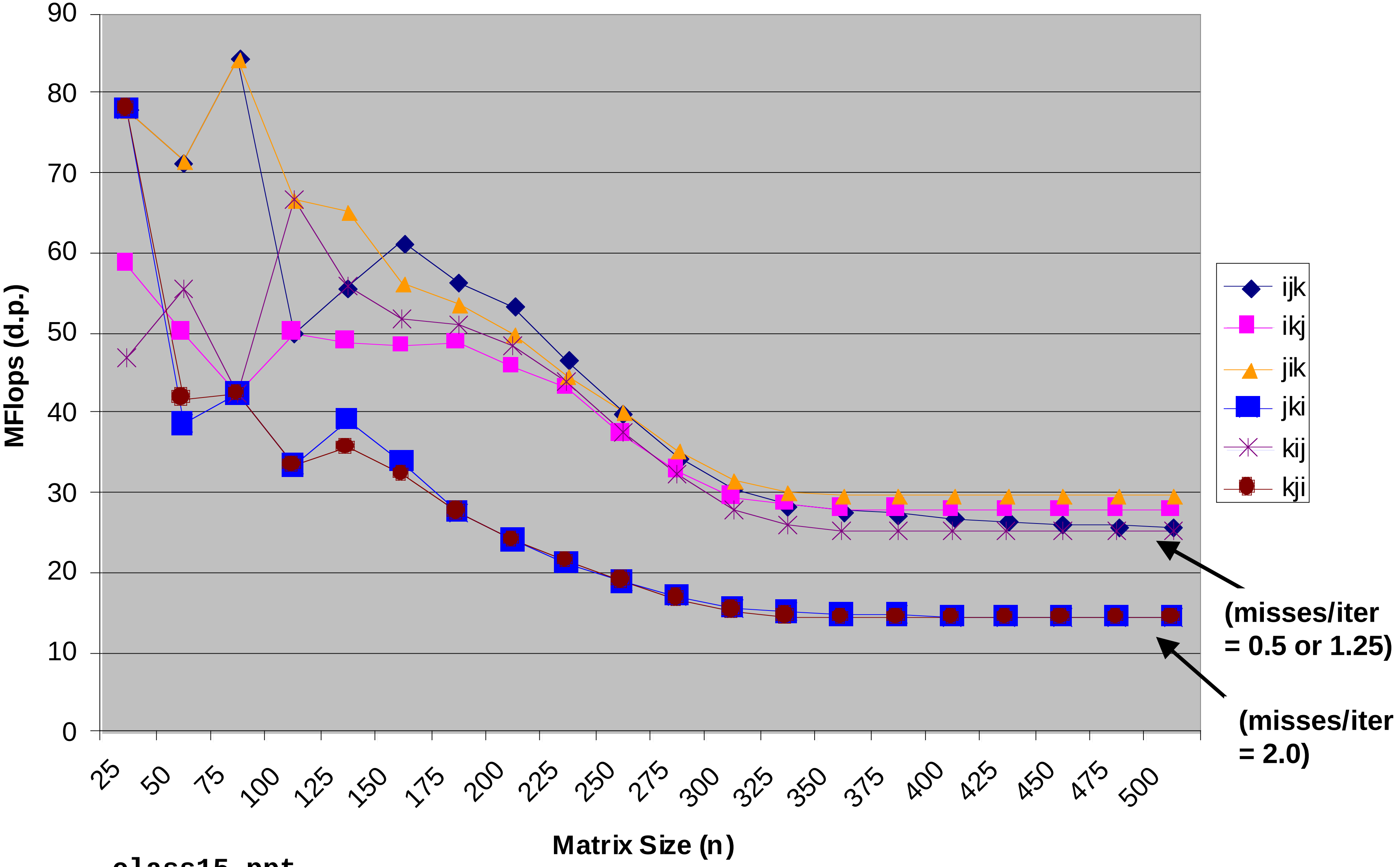
Too big for L2 Cache

(misses/iter = 0.5)

(misses/iter = 1.25)

(misses/iter = 2.0)

Matrix Mult.: Pentium III Xeon



(misses/iter = 0.5 or 1.25)

(misses/iter = 2.0)

class15.ppt

Blocked Matrix Multiplication

- **“Block” (in this context) does not mean “cache block”**
  - instead, it means a sub-block within the matrix

<u>Example:</u> N = 8; *sub-block size* = 4

$$\begin{bmatrix} \text{A11} & \text{A12} \\ \text{A21} & \text{A22} \end{bmatrix} \text{X} \begin{bmatrix} \text{B11} & \text{B12} \\ \text{B21} & \text{B22} \end{bmatrix} = \begin{bmatrix} \text{C11} & \text{C12} \\ \text{C21} & \text{C22} \end{bmatrix}$$

<u>Key idea:</u> Sub-blocks (i.e., $\mathbf{A_{xy}}$) can be treated just like scalars.

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

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

Blocked Matrix Multiply (bijk)

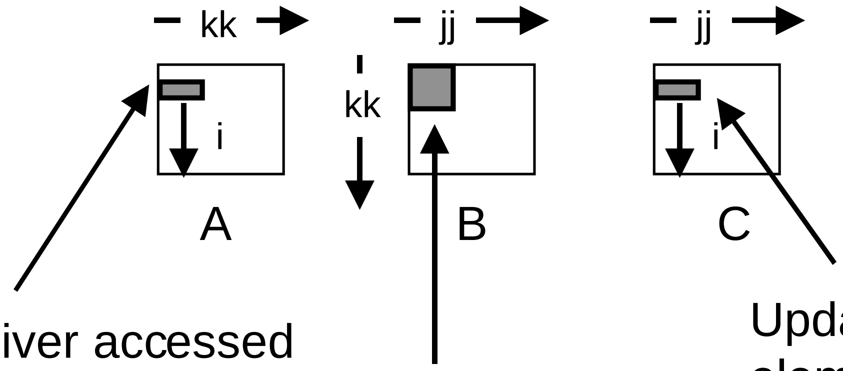
```
for (jj=0; jj<n; jj+=bsize) {
  for (i=0; i<n; i++)
    for (j=jj; j < min(jj+bsize,n); j++)
      c[i][j] = 0.0;
  for (kk=0; kk<n; kk+=bsize) {
    for (i=0; i<n; i++) {
      for (j=jj; j < min(jj+bsize,n); j++) {
        sum = 0.0
        for (k=kk; k < min(kk+bsize,n); k++) {
          sum +=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

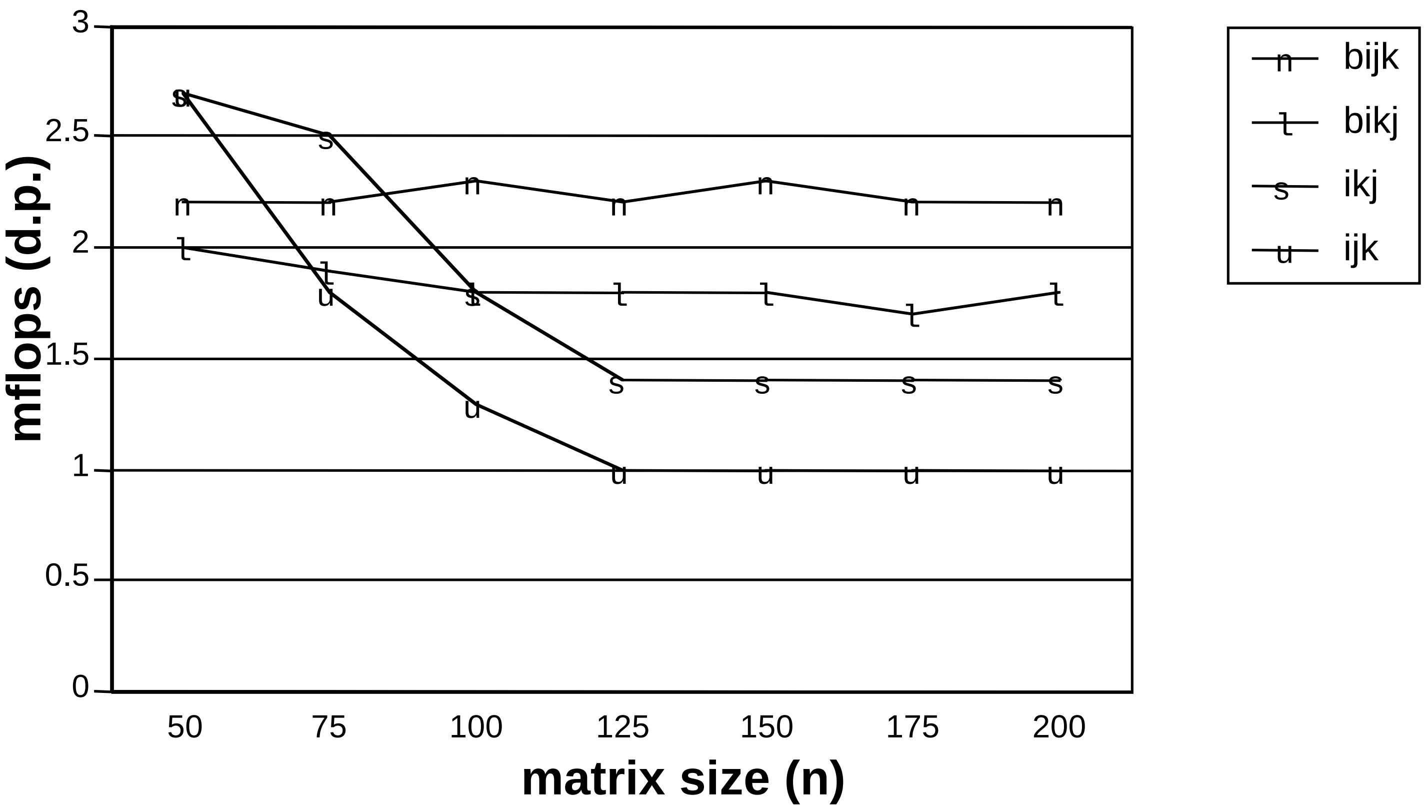


row sliver accessed *bsize* times

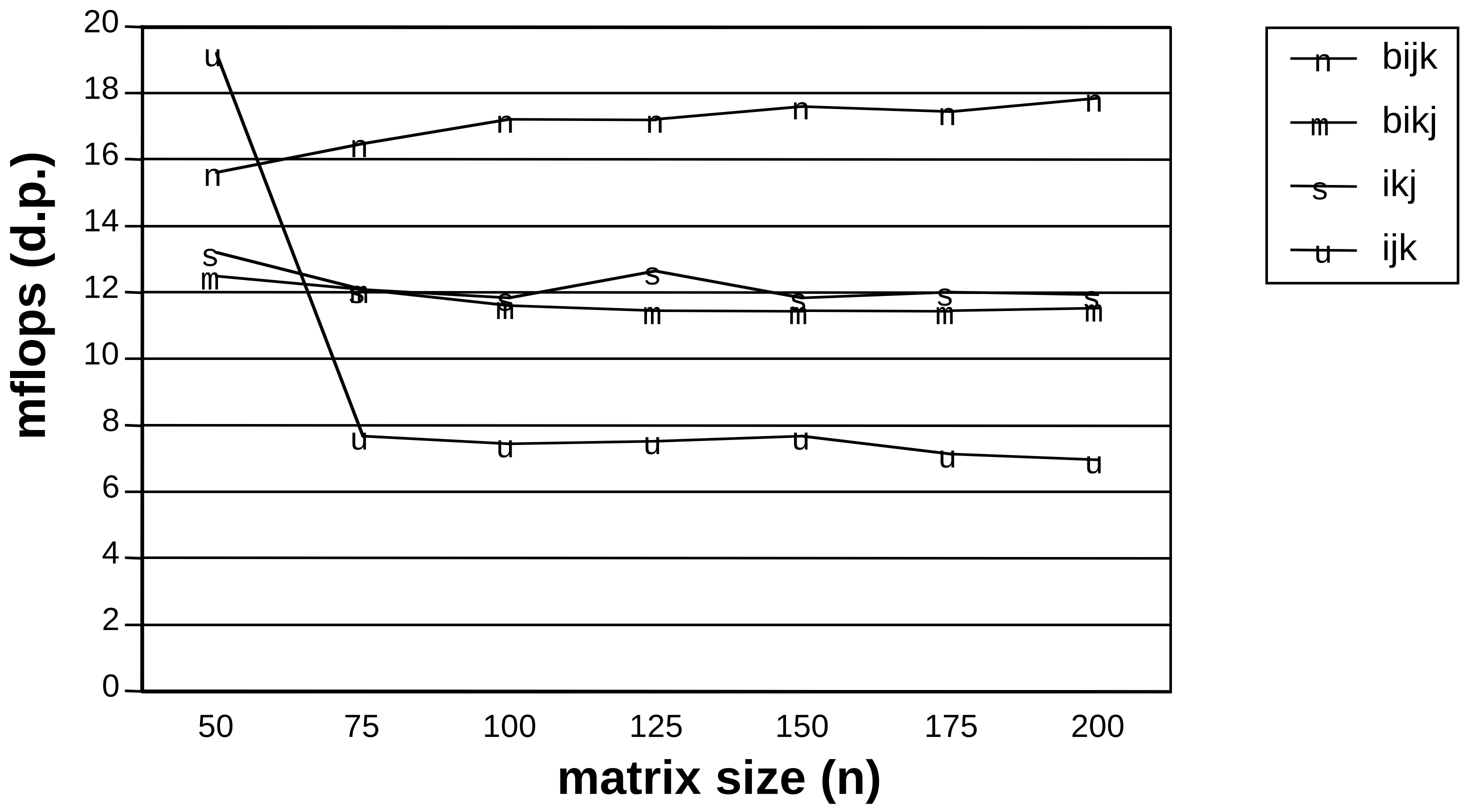
block reused *n* times in succession

Update successive elements of sliver

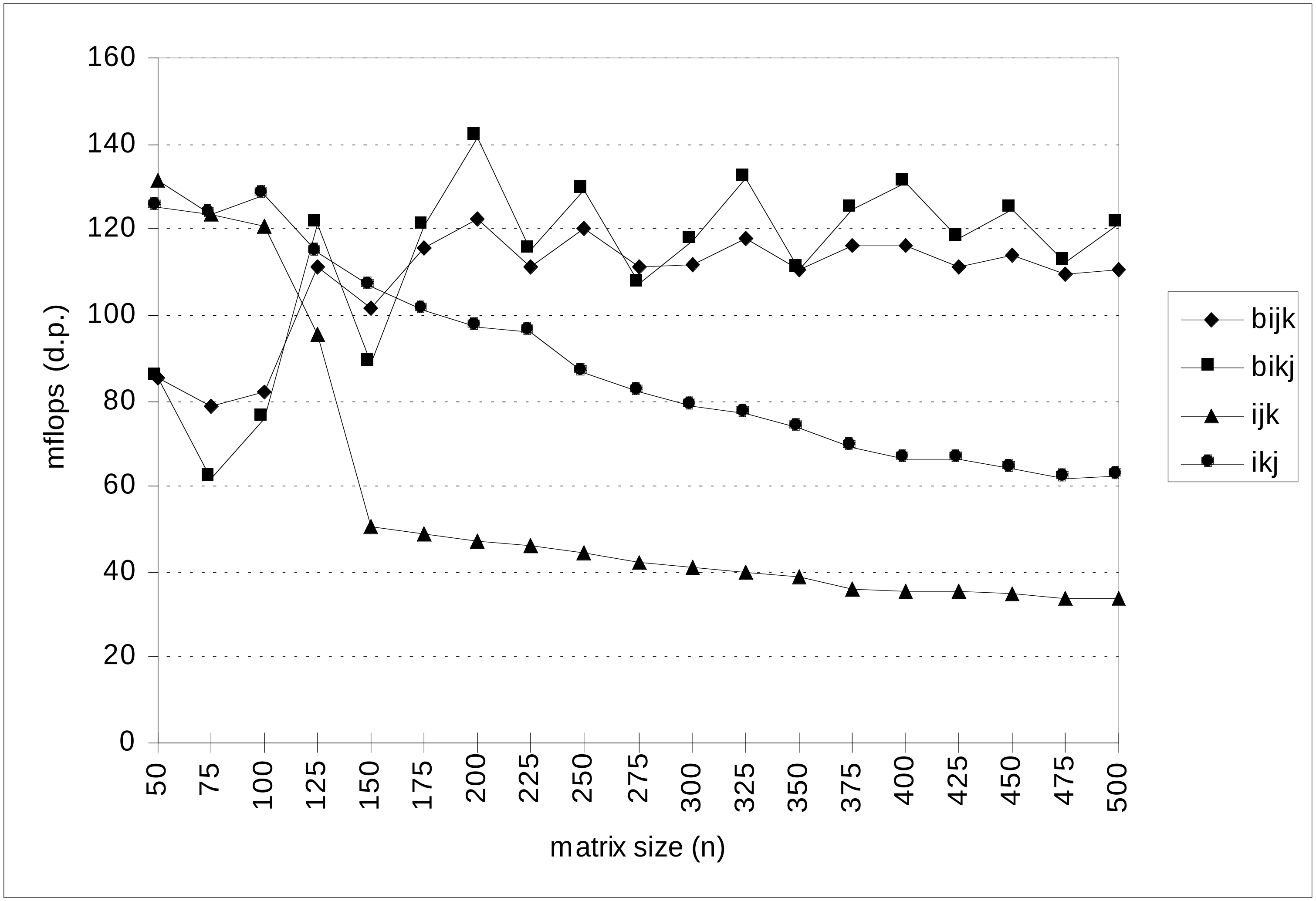
Blocked Matrix Mult. Perf: DEC5000



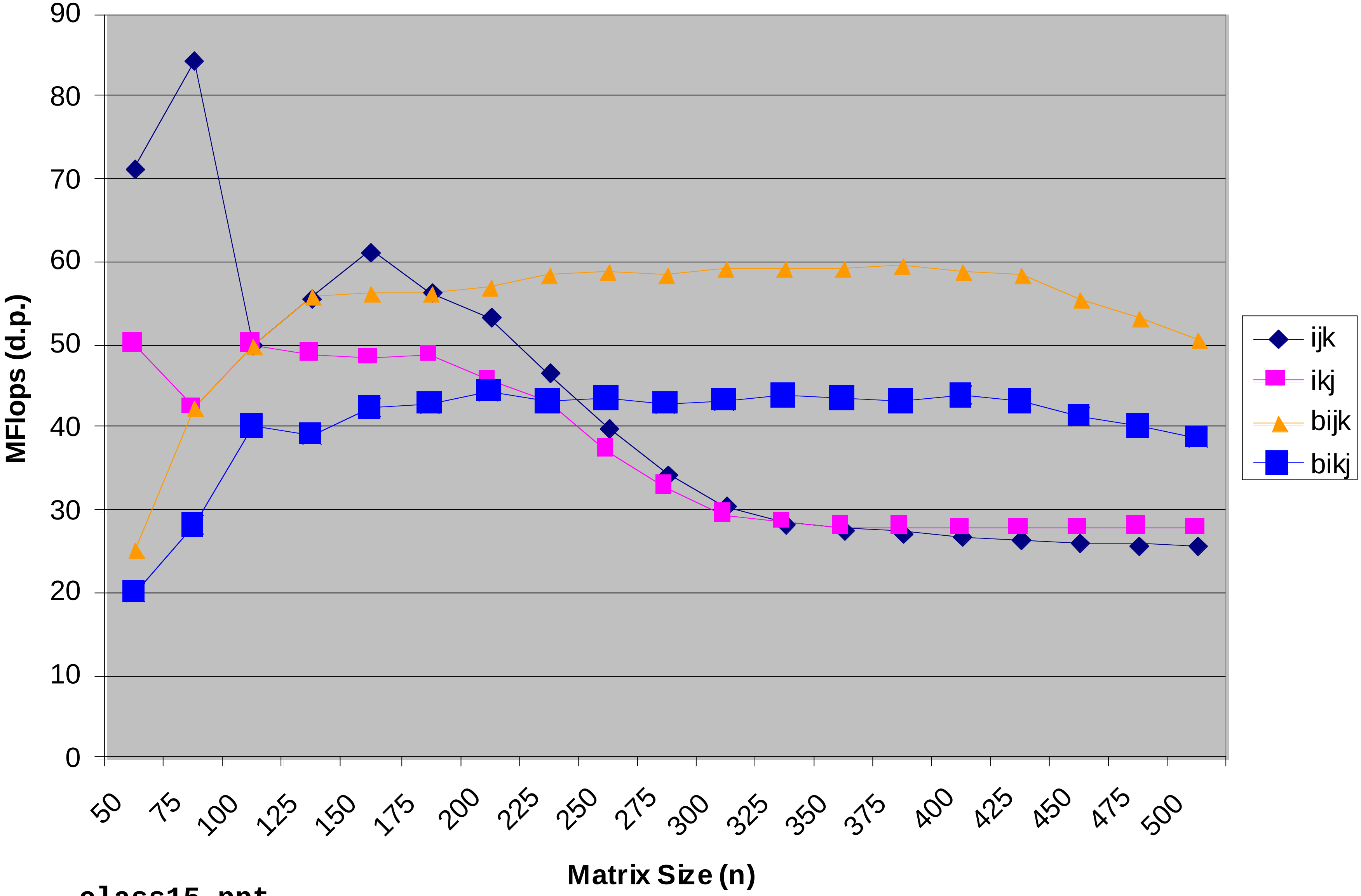
Blocked Matrix Mult. Perf: Sparc20



Blocked Matrix Mult. Perf: Alpha 21164



Blocked Matrix Mult. : Xeon



Observations

Programmer Can Optimize for Cache Performance

- **How data structures are organized**
- **How data accessed**
  - Nested loop structure
  - Blocking is general technique

All Machines Like “Cache Friendly Code”

- **Getting absolute optimum performance very platform specific**
  - Cache sizes, line sizes, associatitivities, etc.
- **Can get most of the advantage with generic code**
  - Keep working set reasonably small