15-213

Memory System Performance

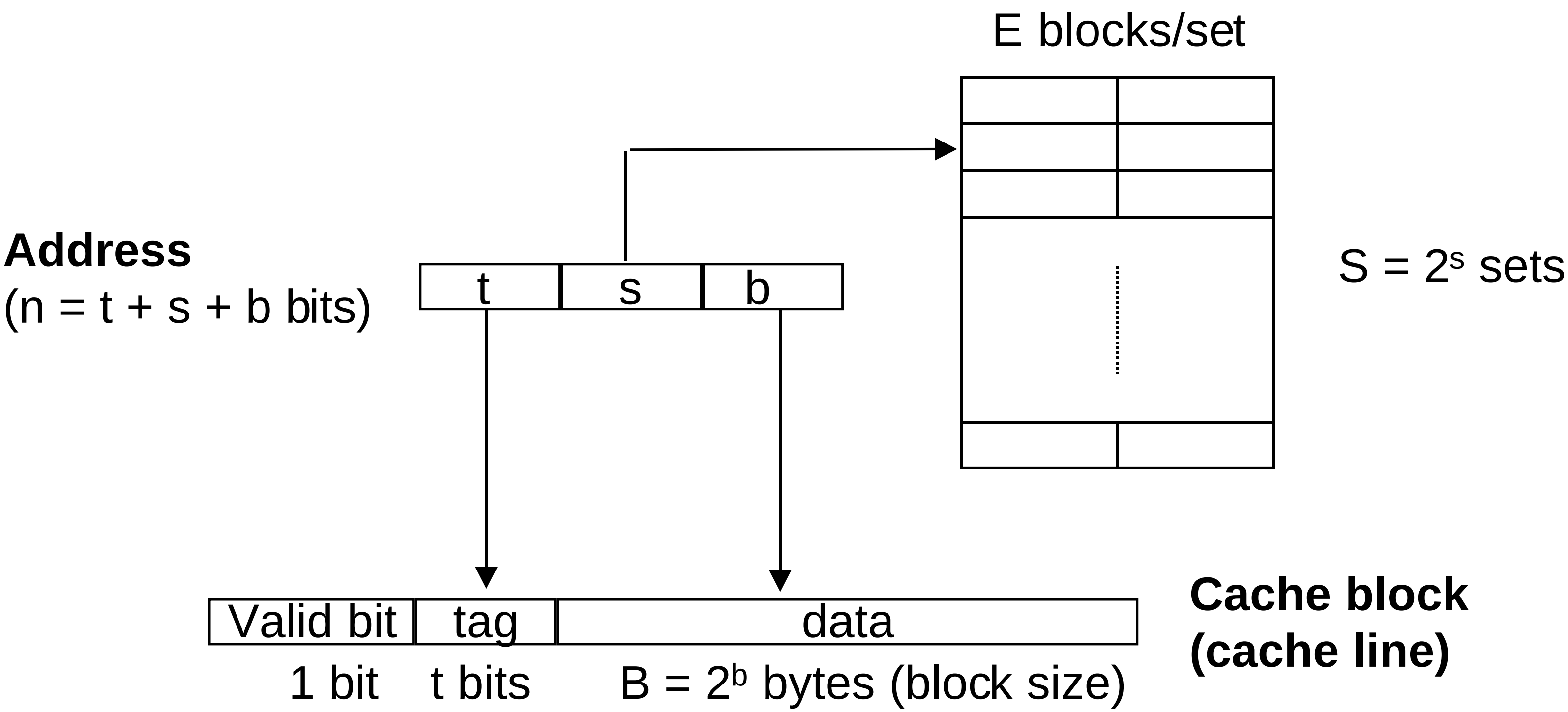
October 29, 1998

Topics

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

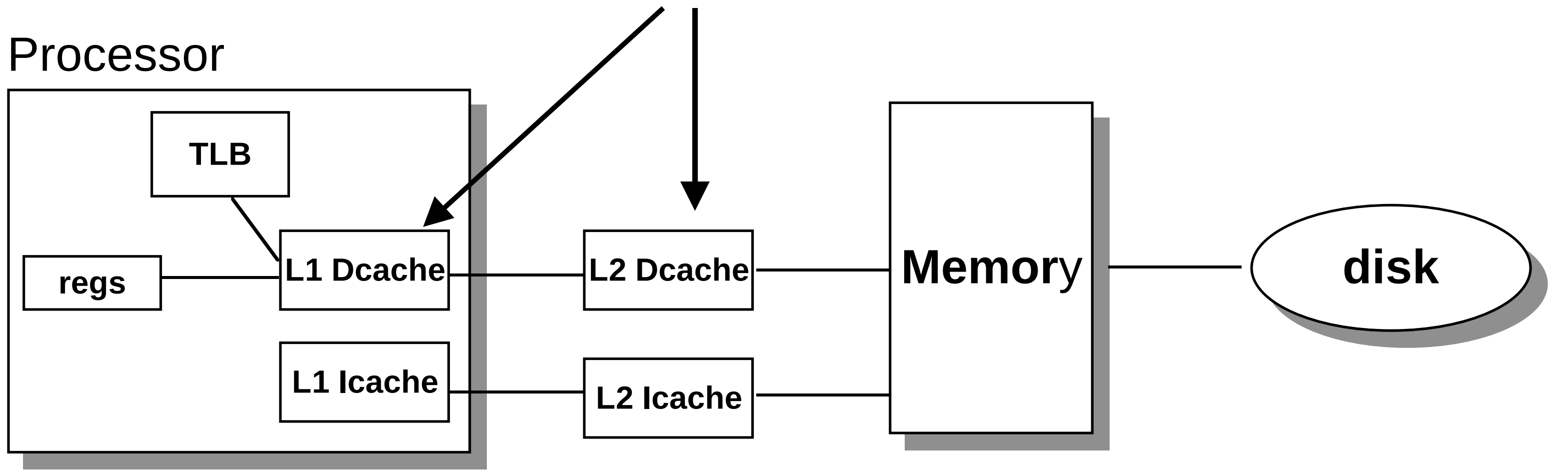
Basic Cache Organization

Address space ($N = 2^n$ bytes) **Cache (C = S x E x B bytes)**



Multi-Level Caches

Can have separate Icache and Dcache or *unified* Icache/Dcache



size:	200 B	8 KB	1M SRAM	128 MB DRAM	10 GB
speed:	5 ns	5 ns	6 ns	70 ns	10 ms
$/Mbyte:			$200/MB	$1.50/MB	$0.06/MB
block size:	4 B	16 B	32 B	4 KB	

larger, slower, cheaper

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

Cache Performance Metrics

Miss Rate

- **fraction of memory references not found in cache (misses/references)**
- **Typical numbers:**

  5-10% for L1

  1-2% for L2

Hit Time

- **time to deliver a block in the cache to the processor (includes time to determine whether the block 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 10-30 cycles for main memory

Impact of Cache and Block Size

Cache Size

- **Effect on miss rate**
  - Larger is better
- **Effect on hit time**
  - Smaller is faster

Block Size

- **Effect on miss rate**
  - Big blocks help exploit spatial locality
  - For given cache size, can hold fewer big blocks than little ones, though
- **Effect on miss penalty**
  - Longer transfer time

Impact of Associativity

- **Direct-mapped, set associative, or fully associative?**

Total Cache Size (tags+data)

- **Higher associativity requires more tag bits, LRU state machine bits**
- **Additional read/write logic, multiplexors**

Miss rate

- **Higher associativity decreases miss rate**

Hit time

- **Higher associativity increases hit time**
  - Direct mapped allows test and data transfer at the same time for read hits.

Miss Penalty

- **Higher associativity requires additional delays to select victim**

Impact of Write Strategy

- **Write-through or write-back?**

Advantages of Write Through

- **Read misses are cheaper. Why?**
- **Simpler to implement.**
- **Requires a write buffer to pipeline writes**

Advantages of Write Back

- **Reduced traffic to memory**
  - Especially if bus used to connect multiple processors or I/O devices
- **Individual writes performed at the processor rate**

Qualitative Cache Performance Model

Compulsory Misses

- First access to line not in cache
- Also called “Cold start” misses

Capacity Misses

- Active portion of memory exceeds cache size

Conflict Misses

- Active portion of address space fits in cache, but too many lines map to same cache entry
- Direct mapped and set associative placement only

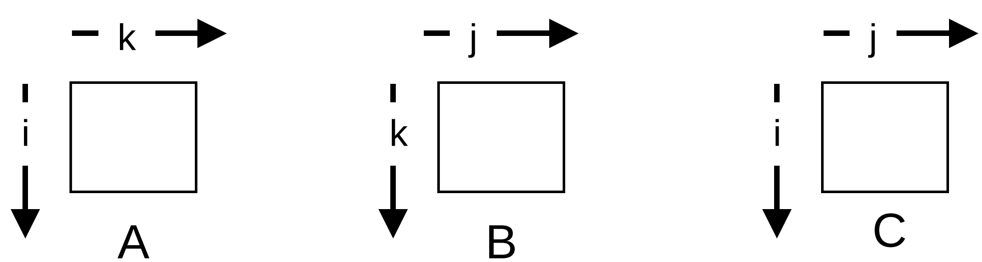
Miss Rate Analysis

Assume

- **Block size = 32B (big enough for 4 32-bit words)**
- **n is very large**
  - Approximate 1/n as 0.0
- **Cache not even big enough to hold multiple rows**

Analysis Method

- **Look at access pattern by inner loop**



Interactions Between Program & Cache

Major Cache Effects to Consider

- **Total cache size**
  - Try to keep heavily used data in highest level cache
- **Block size (sometimes referred to “line size”)**
  - Exploit spatial locality

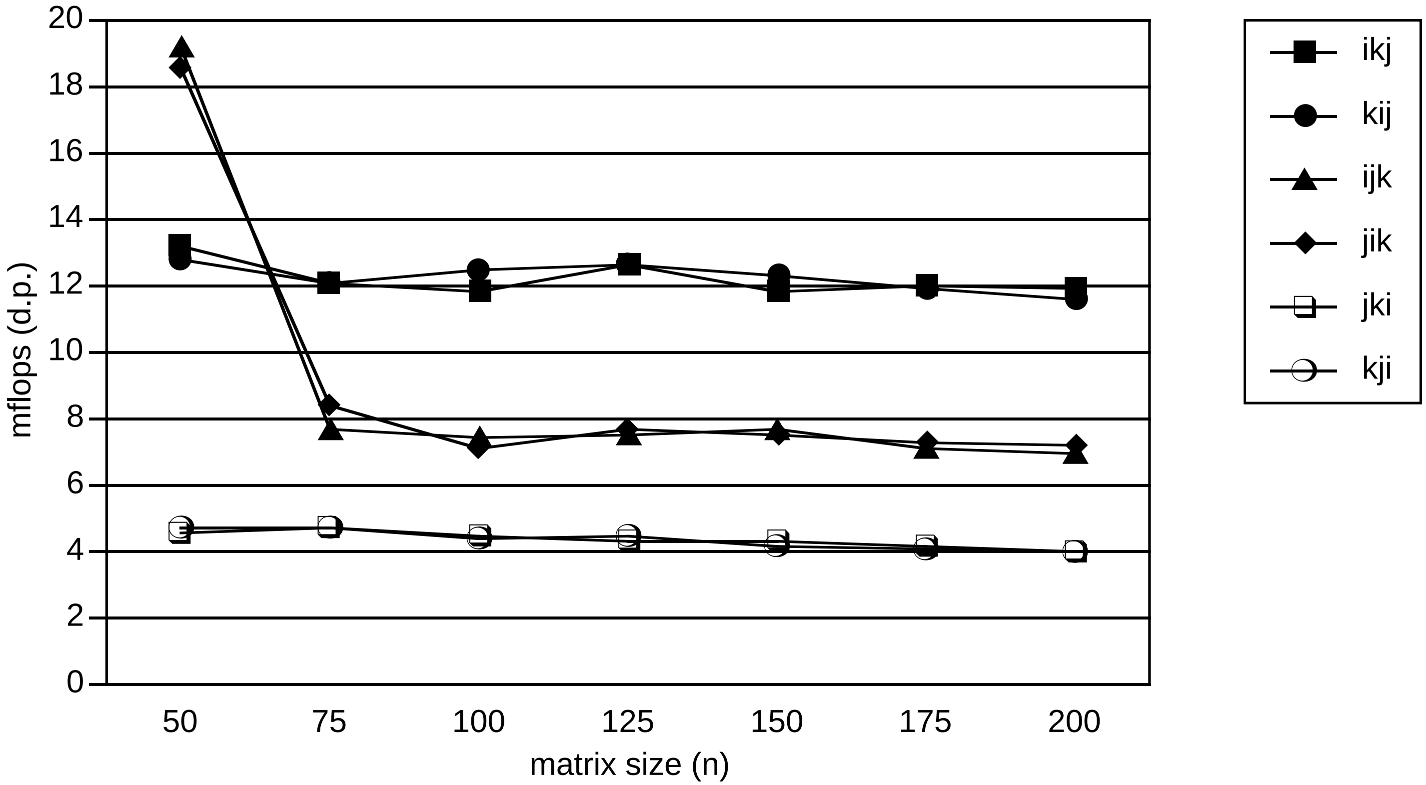
Variable `sum` held in register

```
/* 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

Matmult Performance (Sparc20)



- **As matrices grow in size, exceed cache capacity**
- **Different loop orderings give different performance**
  - Cache effects
  - Whether or not can accumulate in register

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**
- **For block size > 8, get spatial locality**
  - – Cold Start Miss Rate = 8/B

Stepping Through Rows in One Column

```
for (i = 0; i < n; i++)
   sum += a[i][0];
```

- **Accesses distant elements**
- **No spatial locality**
  - – Cold Start Miss rate = 1

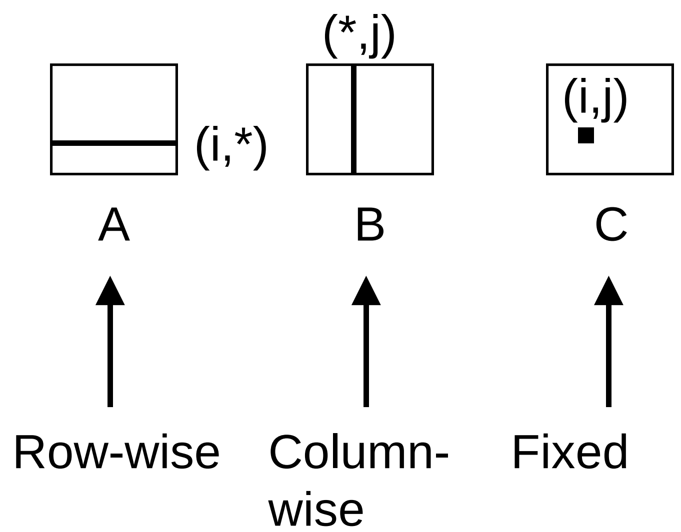
Memory Layout

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:



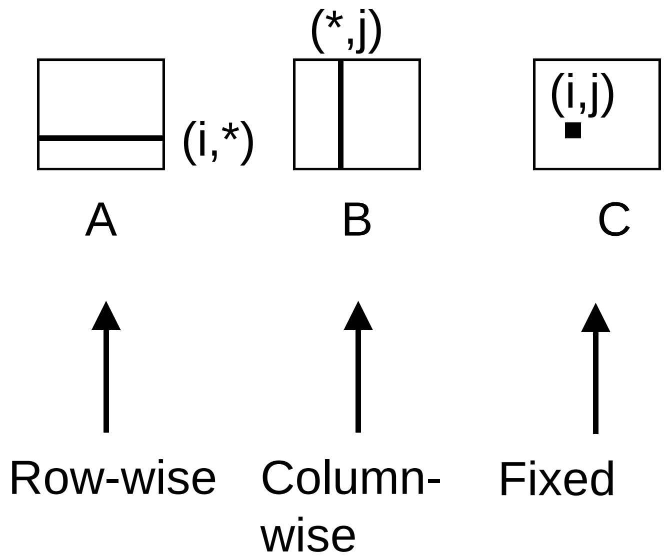
Approx. Miss Rates

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:



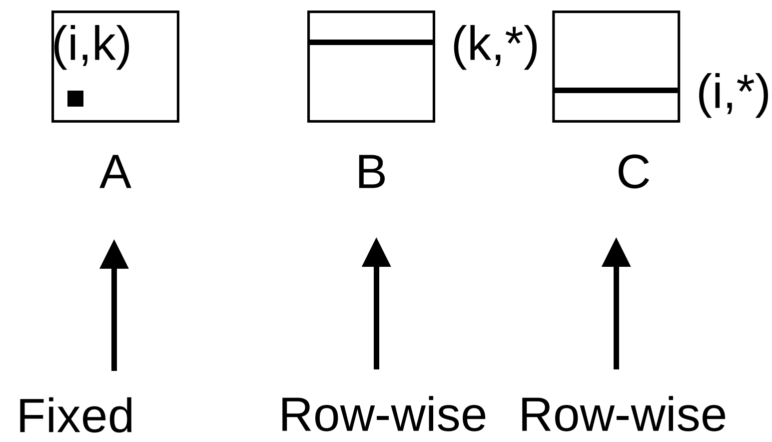
Approx. Miss Rates

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:



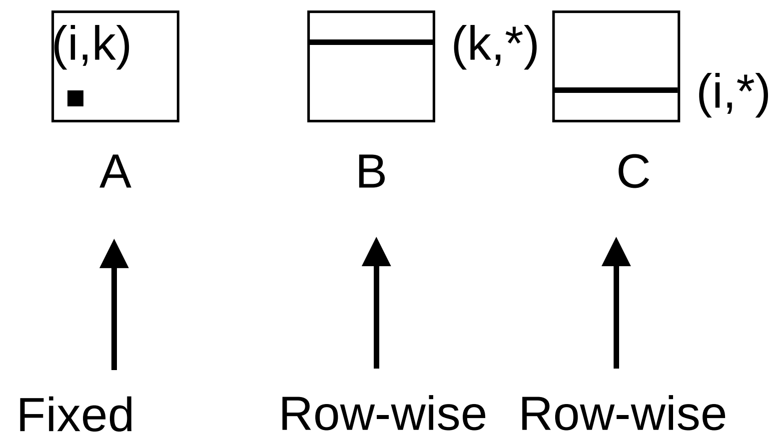
Approx. Miss Rates

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:



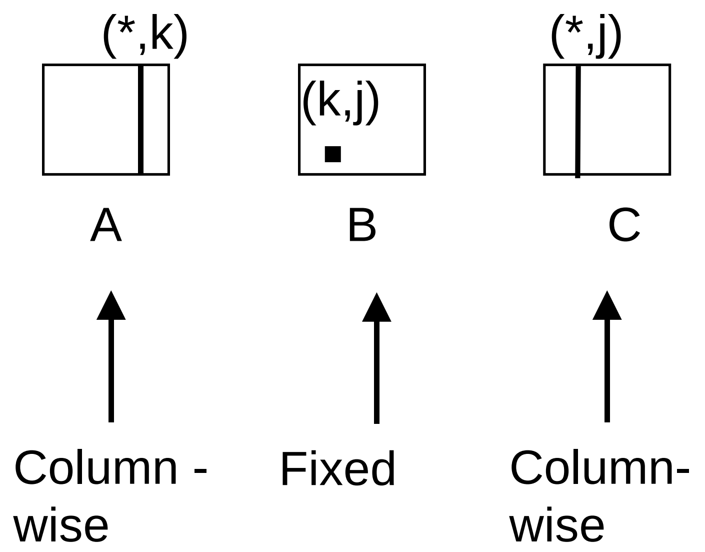
Approx. Miss Rates

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:

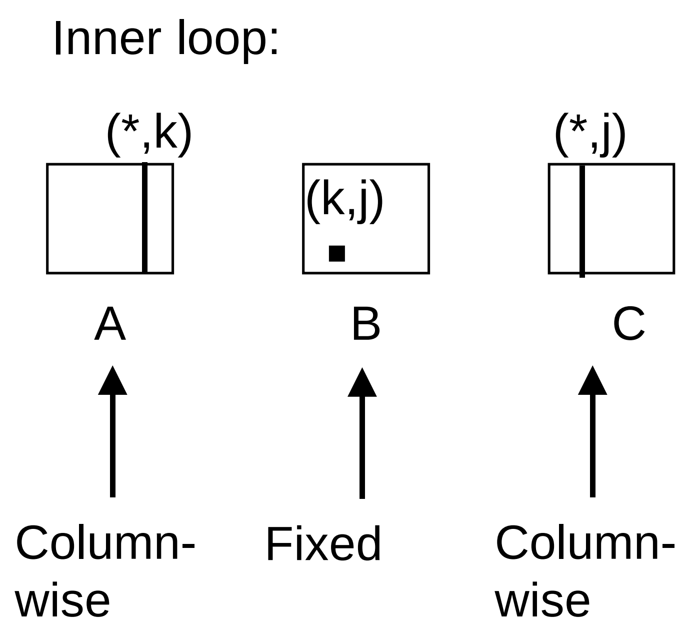


Approx. Miss Rates

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;
  }
}
```



Approx. Miss Rates

a	b	c
1.0	0.0	1.0

Summary of Matrix Multiplication

ijk (L=2, S=0, MR=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;
  }
}
```

jik (L=2, S=0, MR=1.25)

```
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
  }
}
```

kij (L=2, S=1, MR=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];
  }
}
```

ikj (L=2, S=1, MR=0.5)

```
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];
  }
}
```

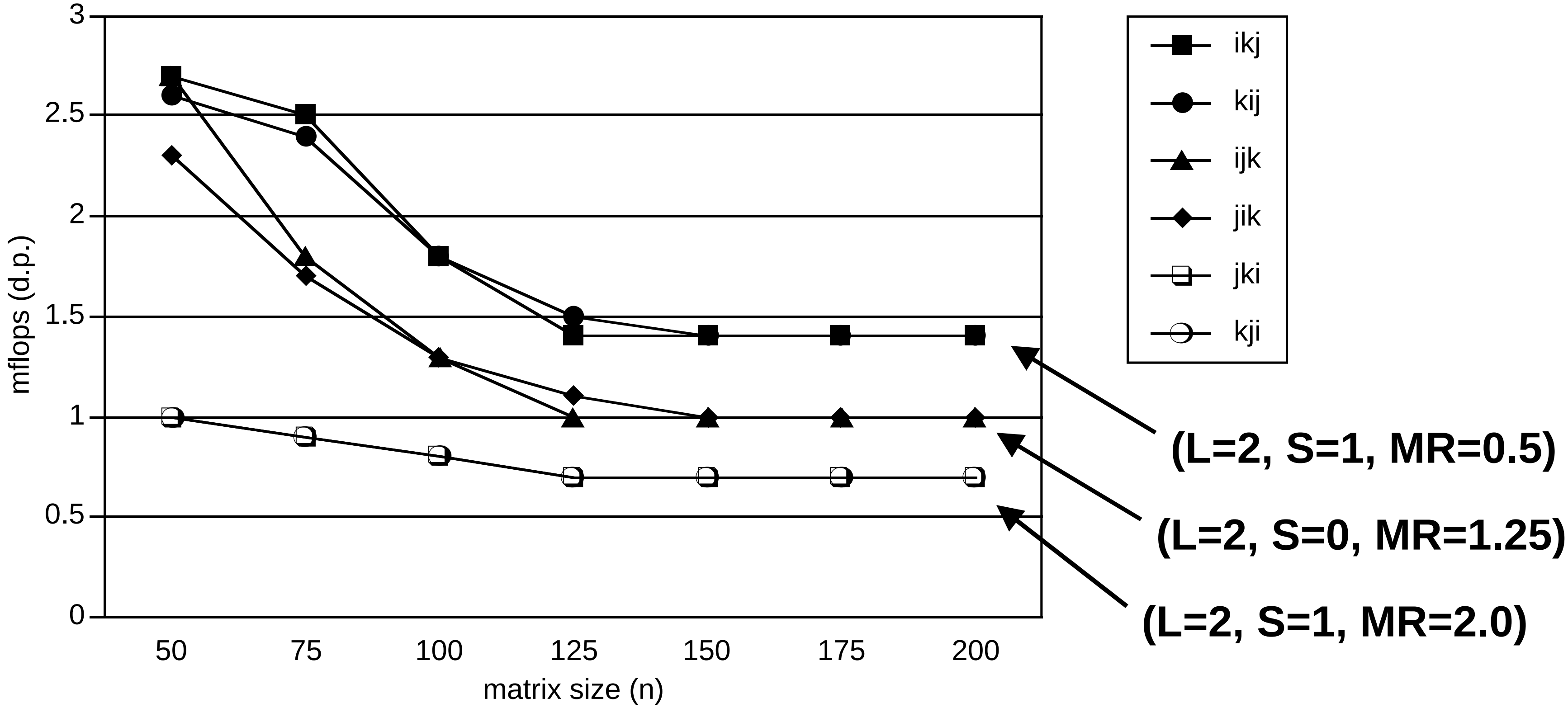
jki (L=2, S=1, MR=2.0)

```
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;
  }
}
```

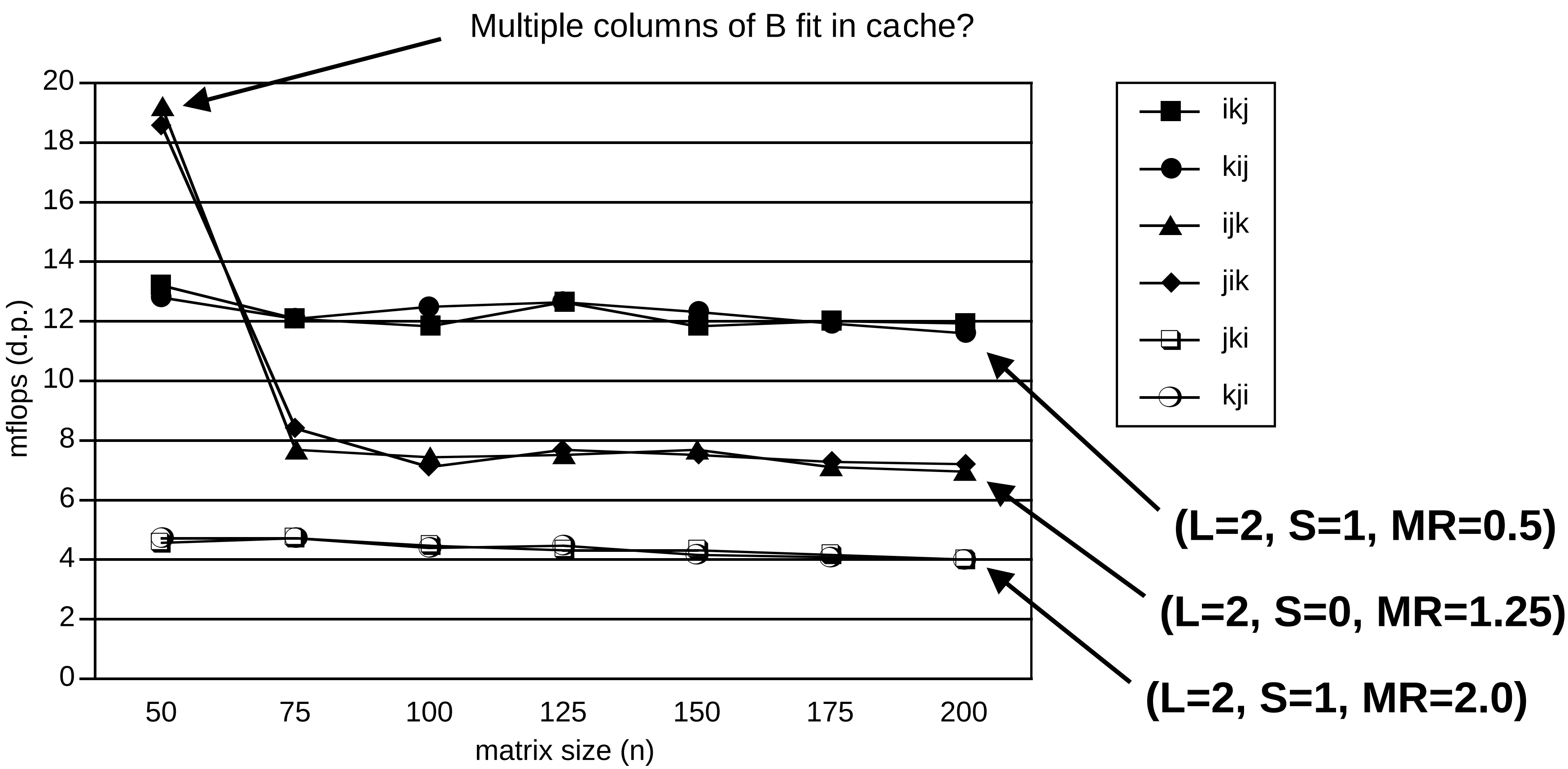
kji (L=2, S=1, MR=2.0)

```
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;
  }
}
```

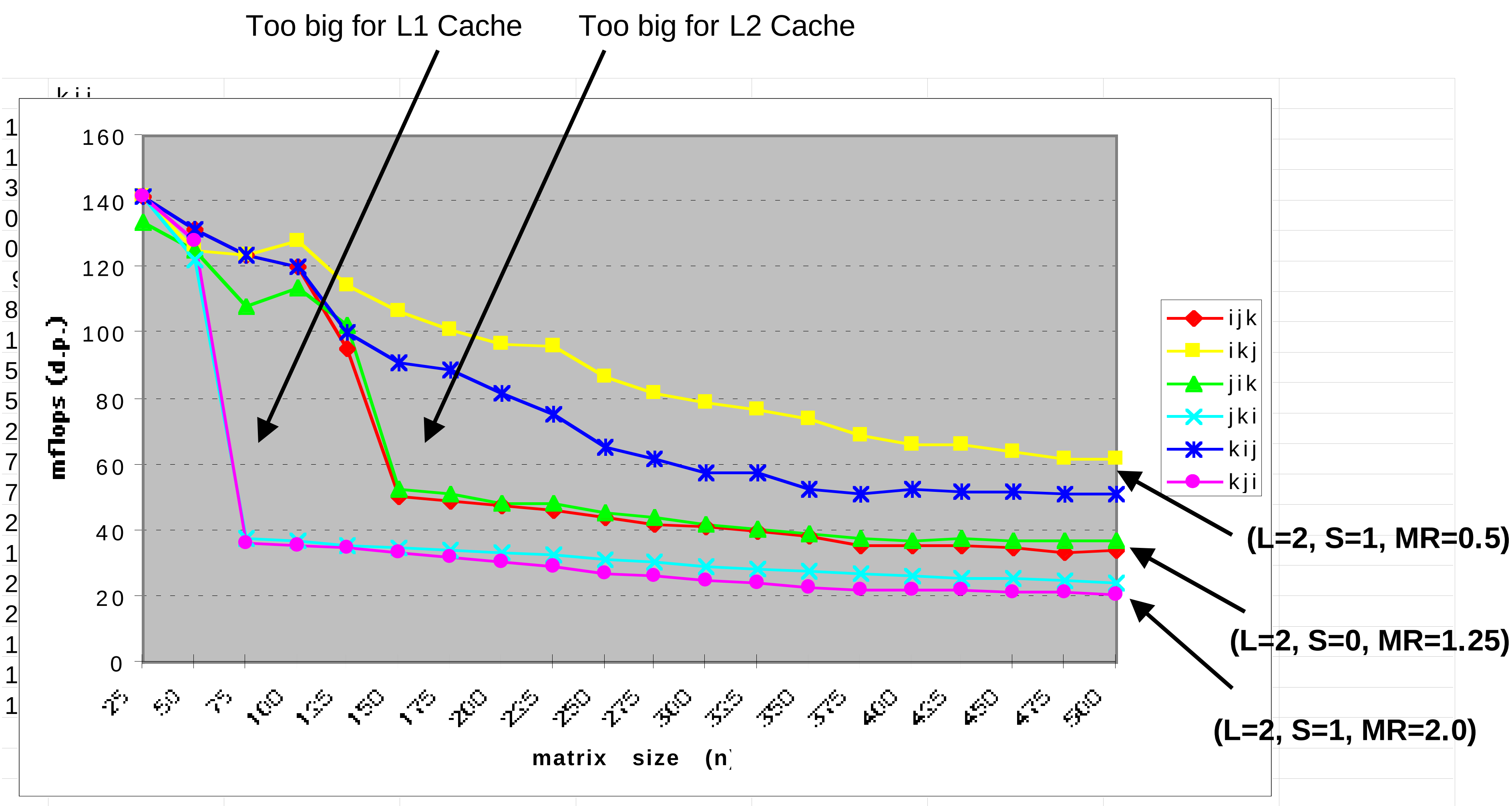
Matmult performance (DEC5000)



Matmult Performance (Sparc20)



Matmult Performance (Alpha 21164)



Block Matrix Multiplication

Example n=8, B = 4:

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

Key idea: Sub-blocks (i.e., A_{ij}) 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)

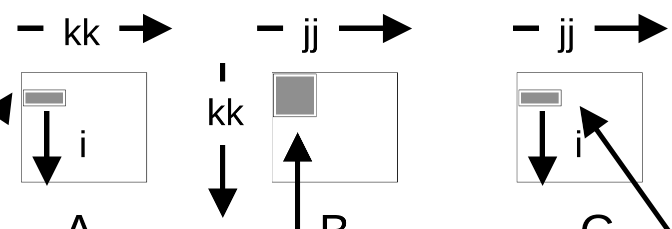
```
for (jj=0; jj<n; jj+=bsize) {
  for (i=0; i<n; i++)
    for (j=jj; j < min(jj+bsize,n); j++)
      c[i][j] = 0.0;
  for (kk=0; kk<n; kk+=bsize) {
    for (i=0; i<n; i++) {
      for (j=jj; j < min(jj+bsize,n); j++) {
        sum = 0.0
        for (k=kk; k < min(kk+bsize,n); k++) {
          sum +=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

Blocked Matrix Multiply Analysis

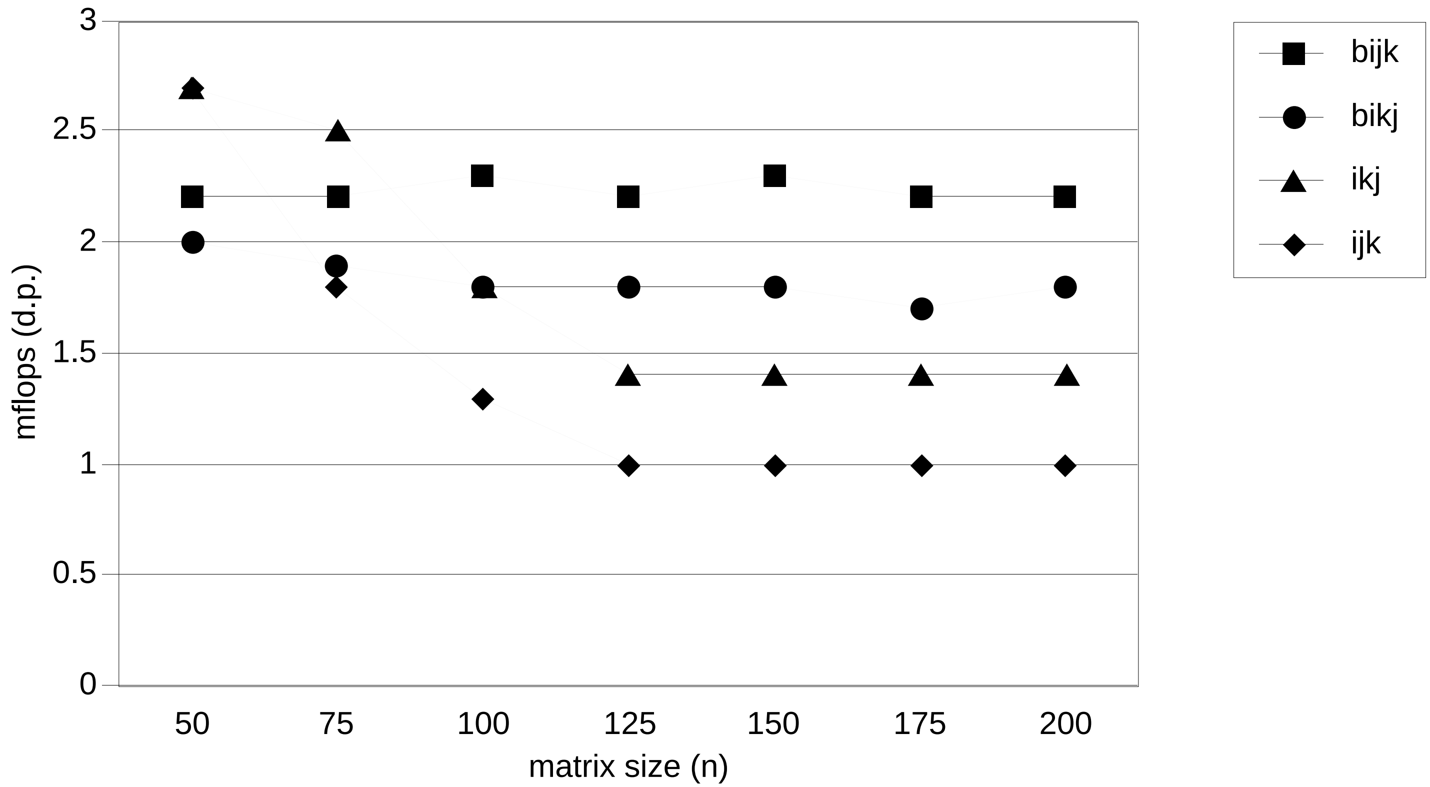
- Innermost loop pair multiplies 1 X bsize sliver of A times bsize X bsize block of B and accumulates into 1 X bsize sliver of C
- Loop over i steps through n row slivers of A & C, using same B

```
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;
      }
```

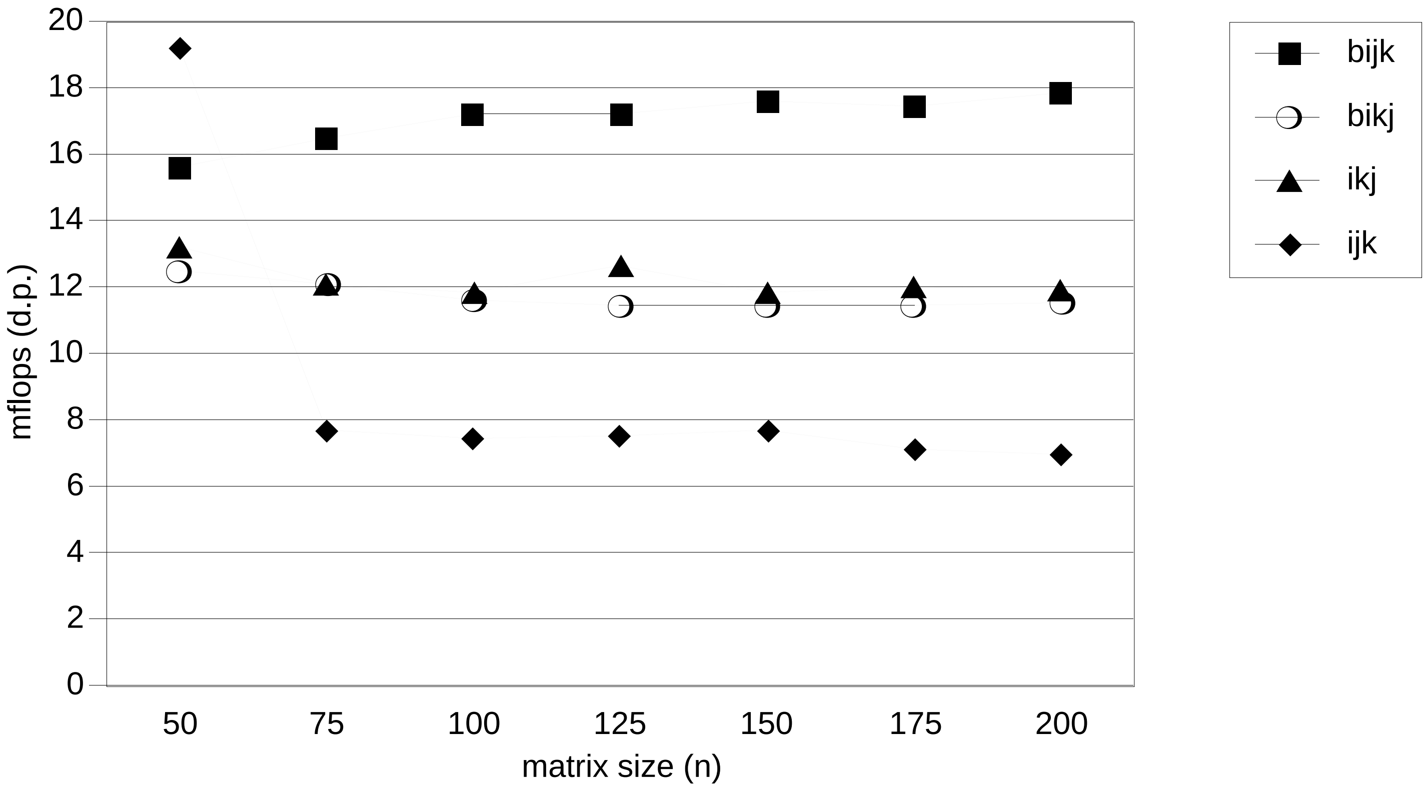
Innermost Loop Pair



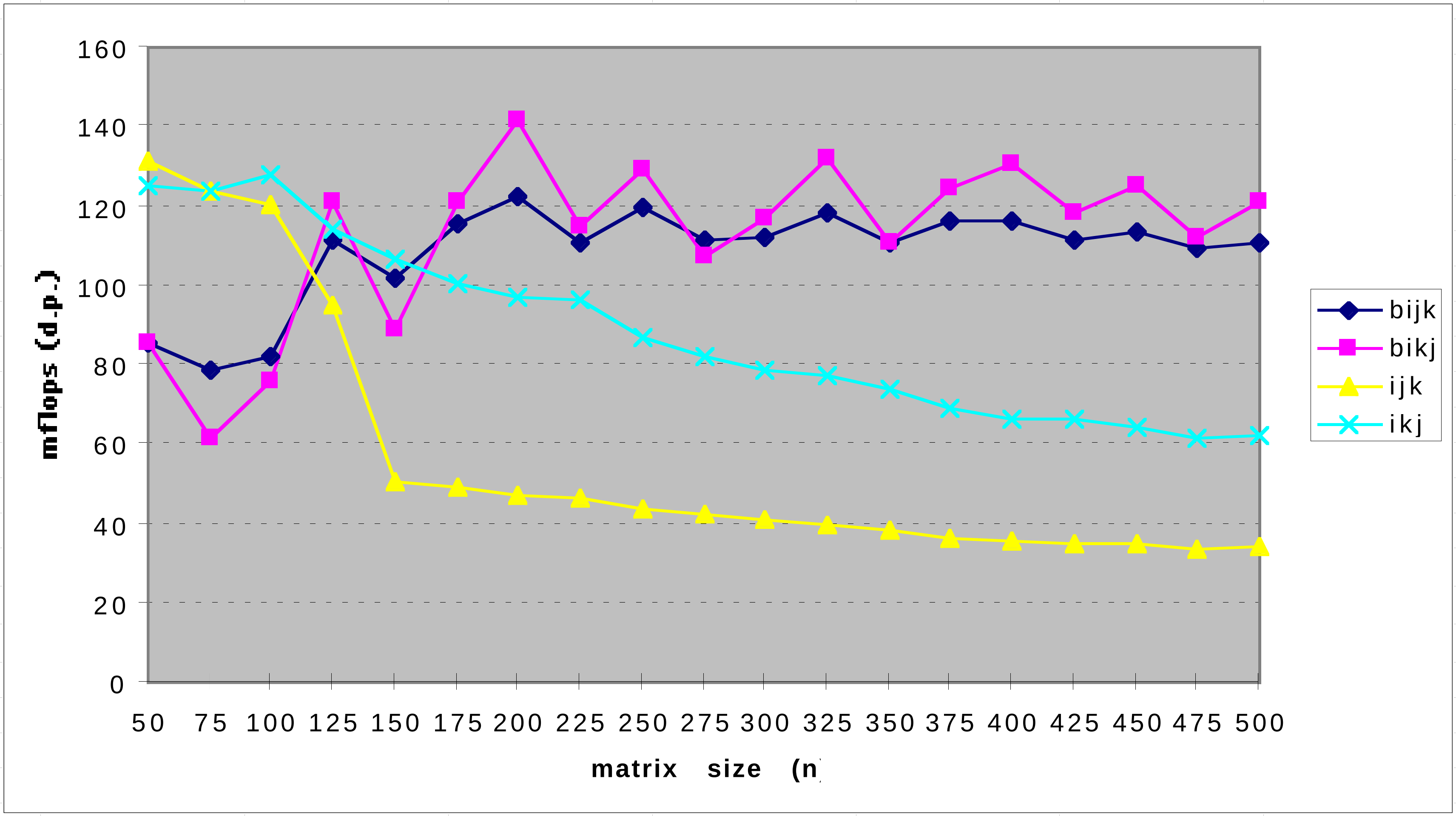
Blocked matmult perf (DEC5000)



Blocked matmult perf (Sparc20)



Blocked matmult perf (Alpha 21164)



Matrix transpose

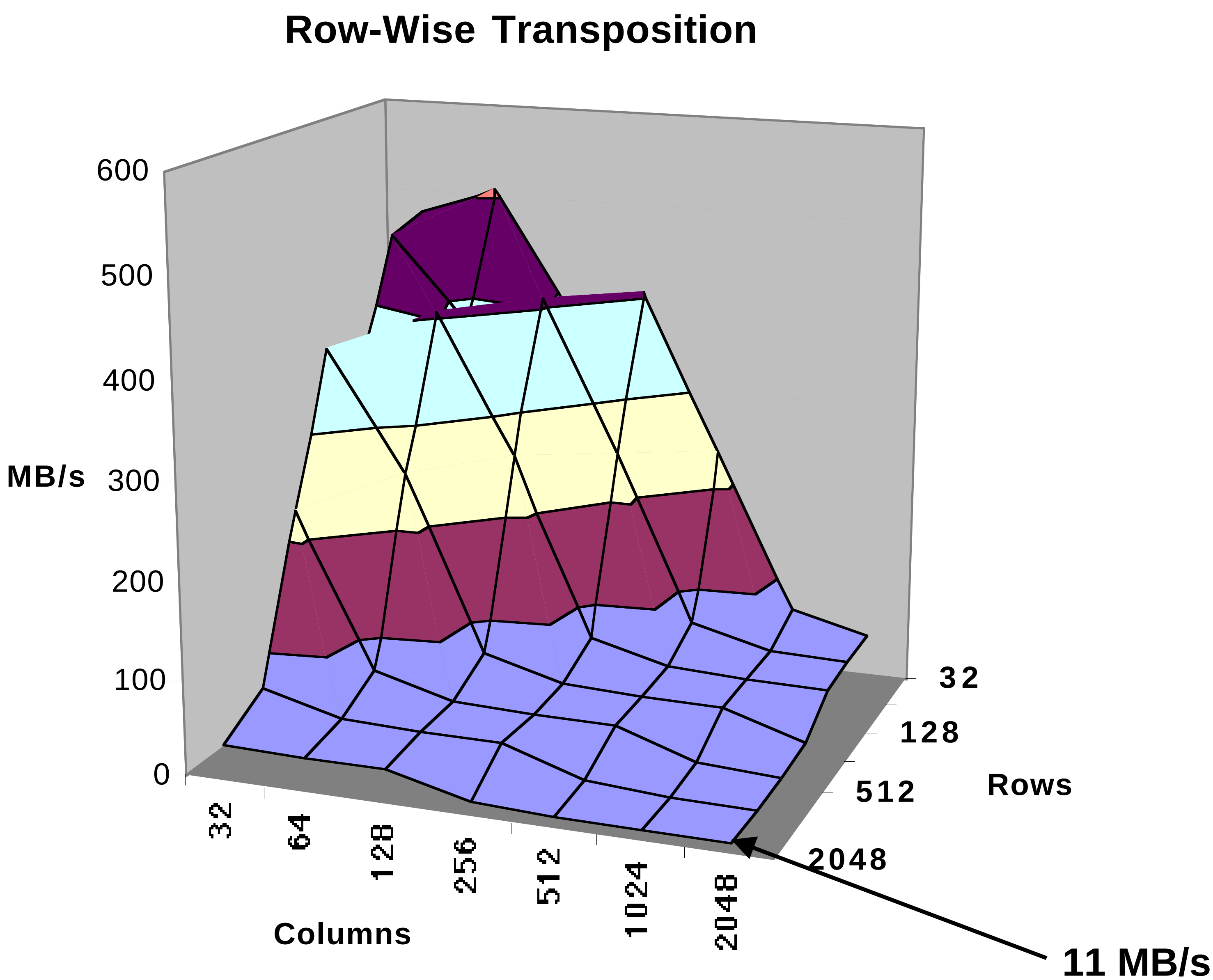
$$\begin{array}{cc} & \text{M cols} \\ \text{N rows} & \begin{bmatrix}1&2&3&4\\5&6&7&8\end{bmatrix}\end{array} \xrightarrow{T} \begin{array}{cc} \text{N cols} & \\ \begin{bmatrix}1&5\\2&6\\3&7\\4&8\end{bmatrix} & \text{M rows}\end{array}$$

Row-wise transpose:

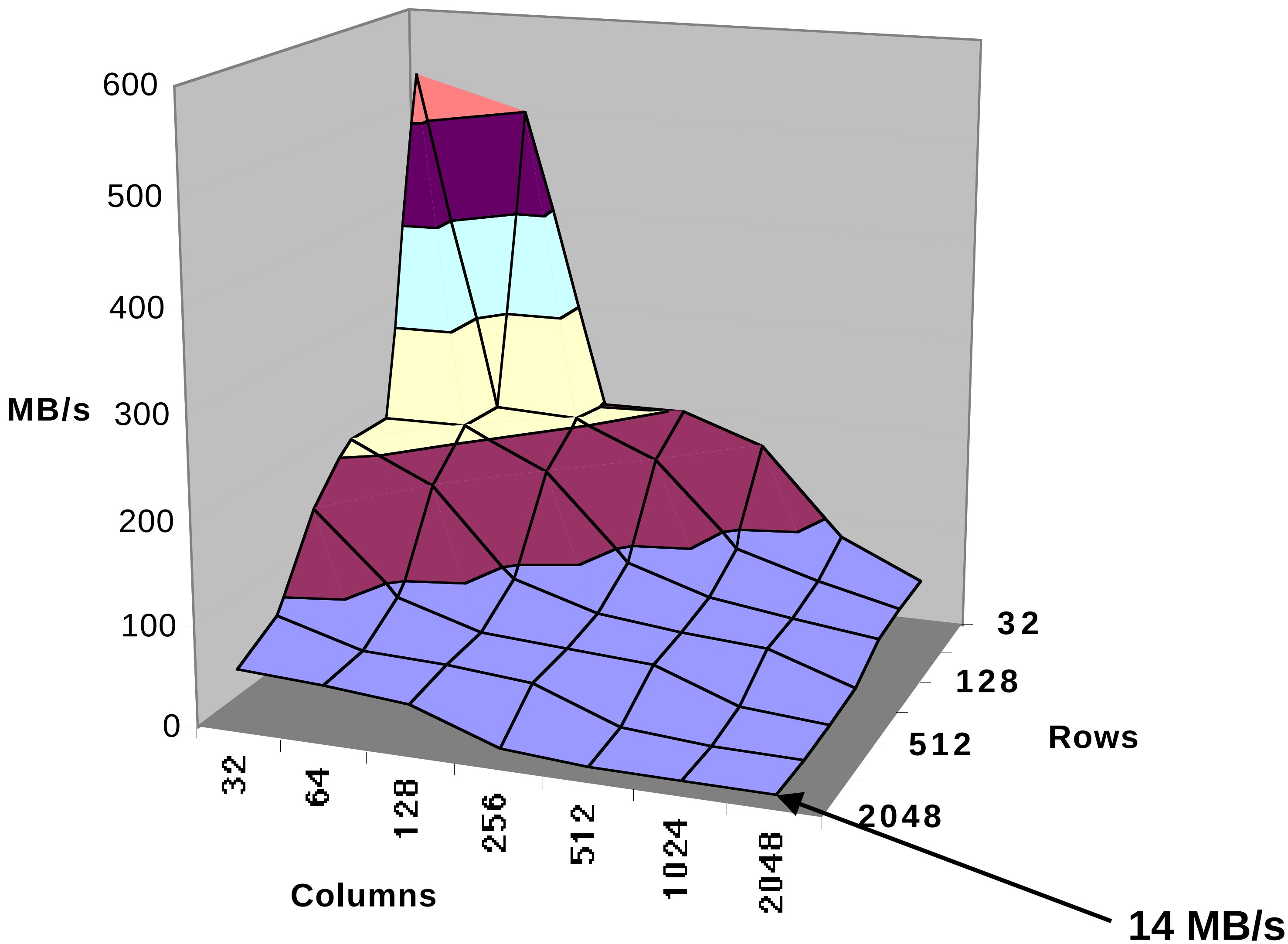
```
for (i=0; i < N; i++)
   for (j=0; j < M; j++)
      dst[j][i] = src[i][j]
```

Column-wise transpose:

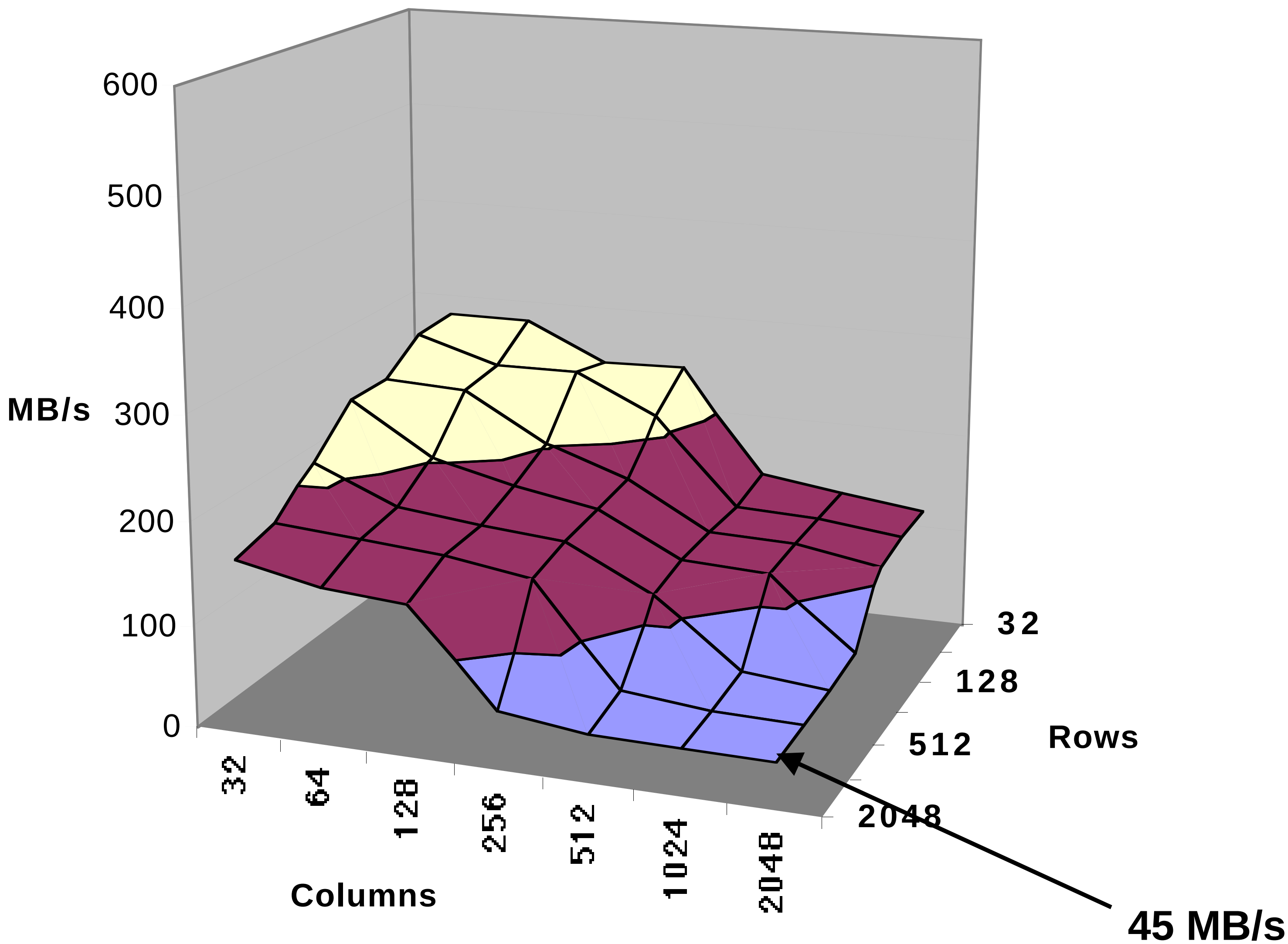
```
for (j=0; j < M; j++)
   for (i=0; i < N; i++)
      dst[j][i] = src[i][j]
```


Row-Wise Transposition
600
500
400
300
200
100
0
MB/s
32
64
128
256
512
1024
2048
Columns
32
128
512
2048
Rows
11 MB/s

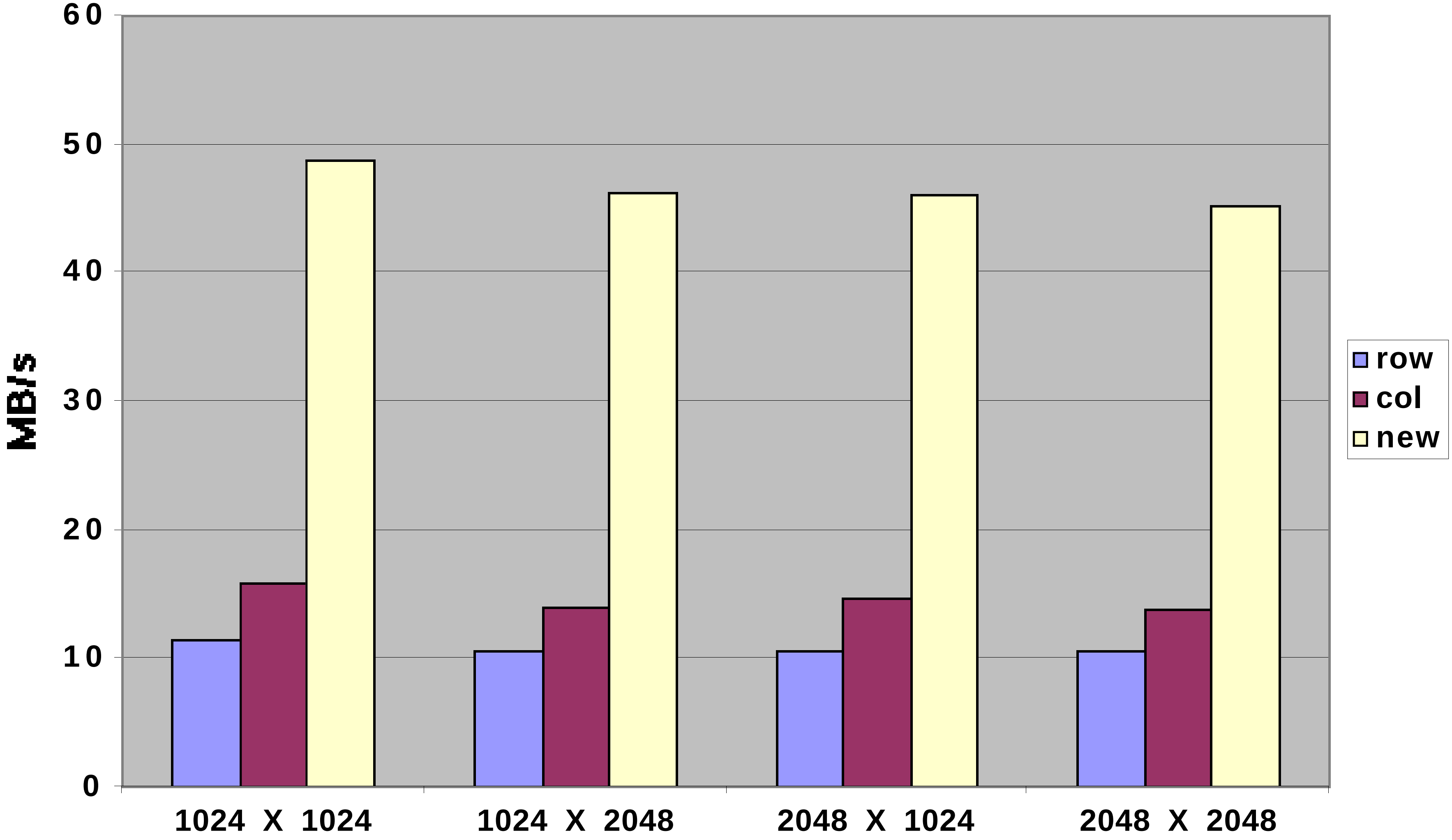
Column-Wise Transposition



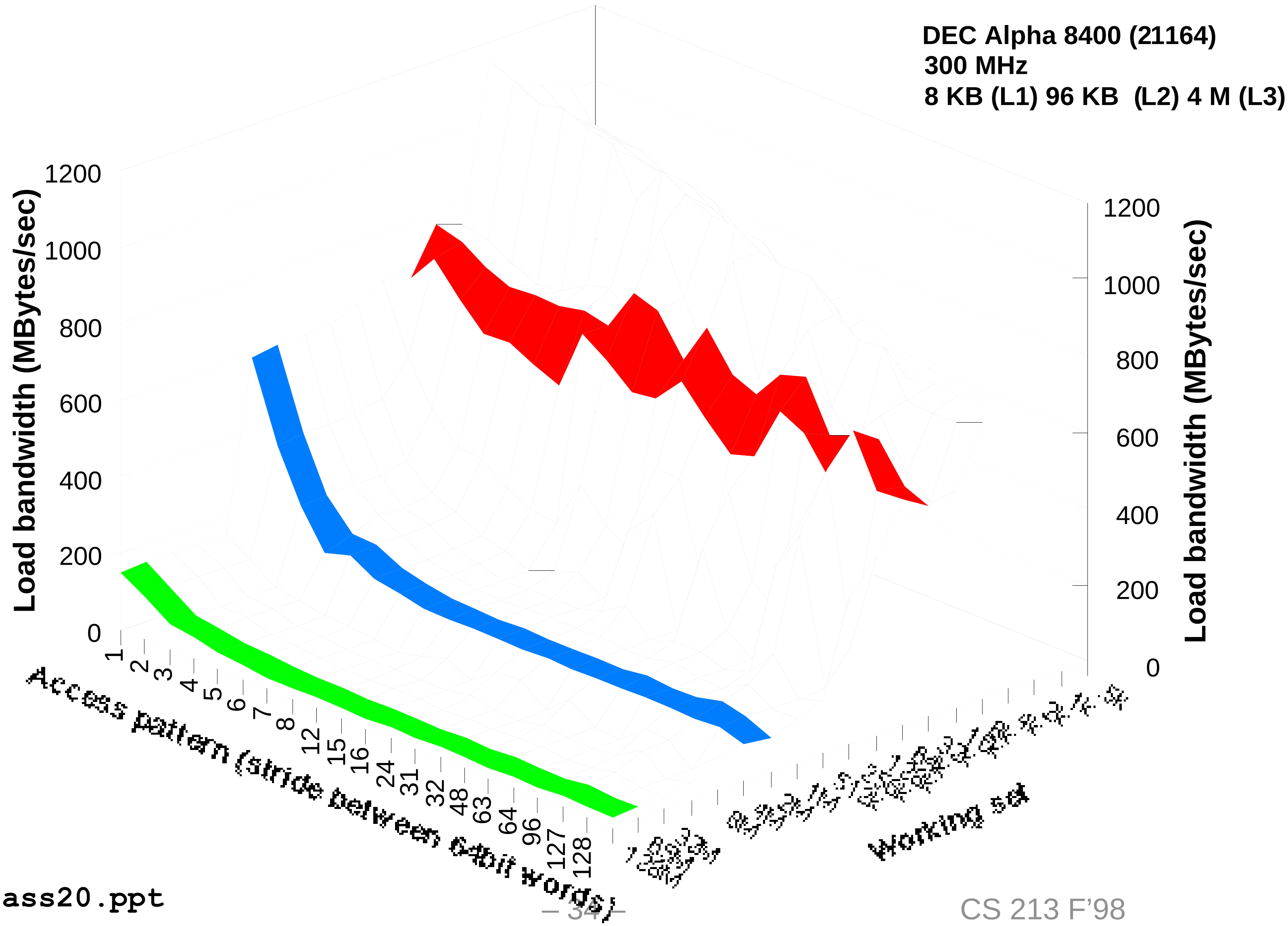
Improved Transposition



Large Matrix Transposition Throughputs



The Memory Mountain Range



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 block