

CS 33

Caches

Cache Performance Metrics

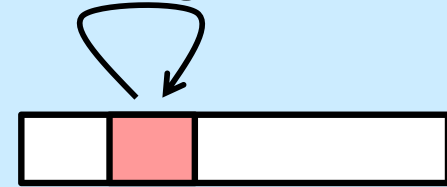
- **Miss rate**
 - fraction of memory references not found in cache (misses / accesses)
= 1 – hit rate
 - typical numbers (in percentages):
 - » 3-10% for L1
 - » can be quite small (e.g., < 1%) for L2, depending on size, etc.
 - **Hit time**
 - time to deliver a line in the cache to the processor
 - » includes time to determine whether the line is in the cache
 - typical numbers:
 - » 1-2 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 (trend: increasing!)
-

Let's Think About Those Numbers

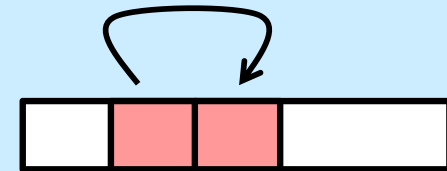
- Huge difference between a hit and a miss
 - could be 100x, if just L1 and main memory
- Would you believe 99% hit rate is twice as good as 97%?
 - consider:
 - cache hit time of 1 cycle
 - miss penalty of 100 cycles
 - average access time:
 - 97% hits: $.97 * 1 \text{ cycle} + 0.03 * 100 \text{ cycles} \approx 4 \text{ cycles}$
 - 99% hits: $.99 * 1 \text{ cycle} + 0.01 * 100 \text{ cycles} \approx 2 \text{ cycles}$
- This is why “miss rate” is used instead of “hit rate”

Locality

- **Principle of Locality:** programs tend to use data and instructions with addresses near or equal to those they have used recently



- **Temporal locality:**
 - recently referenced items are likely to be referenced again in the near future



- **Spatial locality:**
 - items with nearby addresses tend to be referenced close together in time

Locality Example

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

- **Data references**

- reference array elements in succession (stride-1 reference pattern)

Spatial locality

- reference variable `sum` each iteration

Temporal locality

- **Instruction references**

- reference instructions in sequence.
- cycle through loop repeatedly

Spatial locality

Temporal locality

Qualitative Estimates of Locality

- **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 with respect to array a?

```
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;  
}
```

Quiz 1

Does this function have good locality with respect to array a?

- a) yes
- b) no

```
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;  
}
```

Writing Cache-Friendly Code

- **Make the common case go fast**
 - focus on the inner loops of the core functions
- **Minimize the misses in the inner loops**
 - repeated references to variables are good (**temporal locality**)
 - stride-1 reference patterns are good (**spatial locality**)

Key idea: our qualitative notion of locality is quantified through our understanding of cache memories

Matrix Multiplication Example

- **Description:**
 - multiply $N \times N$ matrices
 - $O(N^3)$ total operations
 - N reads per source element
 - N values summed per destination
 - » but may be able to hold in register

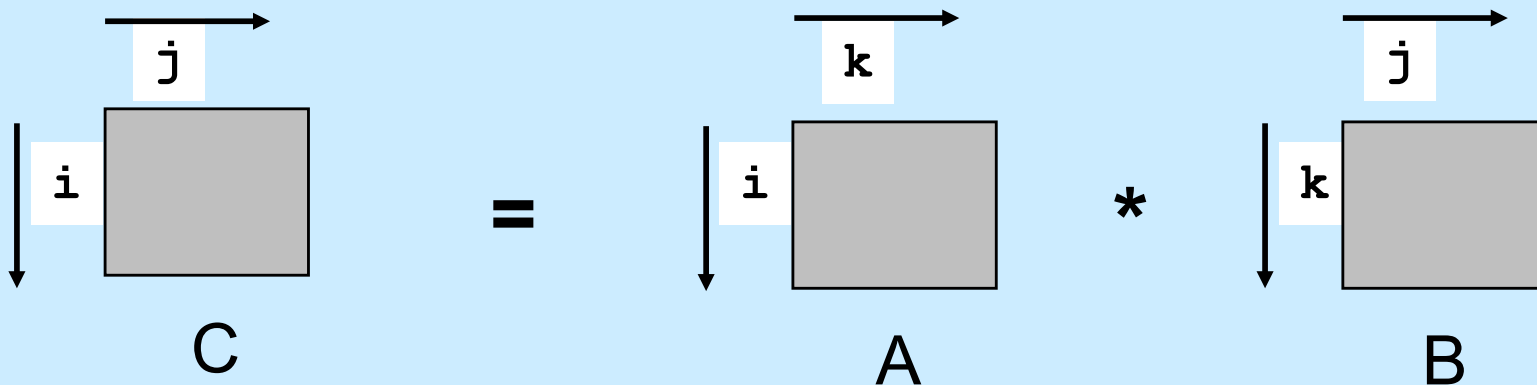
```
/* ijk */
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

Variable sum held in register

```
/* 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];
    }
}
```

Miss-Rate Analysis for Matrix Multiply

- **Assume:**
 - Block size = 32B (big enough for four 64-bit words)
 - matrix dimension (N) is very large
 - » approximate $1/N$ as 0.0
 - cache is not big enough to hold multiple rows
- **Analysis method:**
 - look at access pattern of inner loop



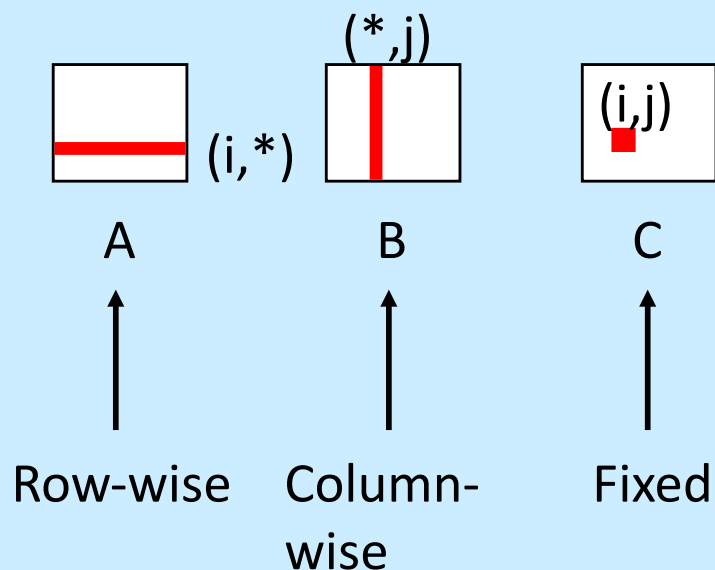
Layout of C Arrays in Memory (review)

- **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:



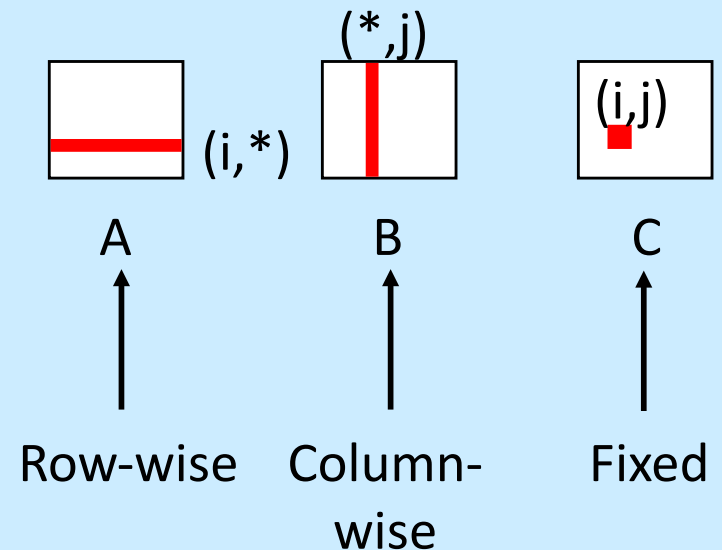
Misses per inner loop iteration:

| <u>A</u> | <u>B</u> | <u>C</u> |
|----------|----------|----------|
| 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:



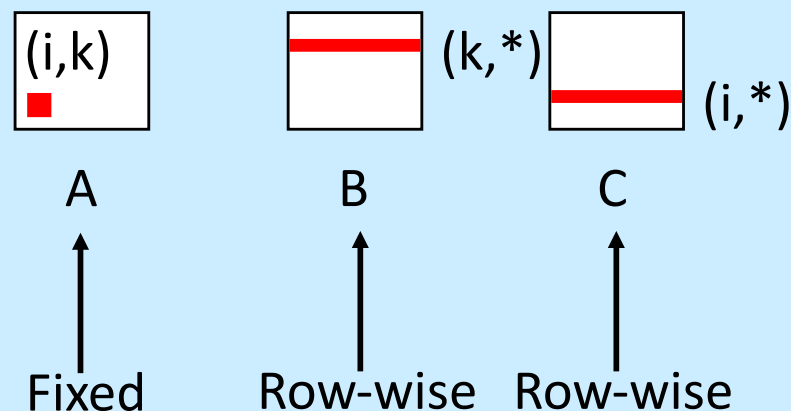
Misses per inner loop iteration:

| <u>A</u> | <u>B</u> | <u>C</u> |
|----------|----------|----------|
| 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:

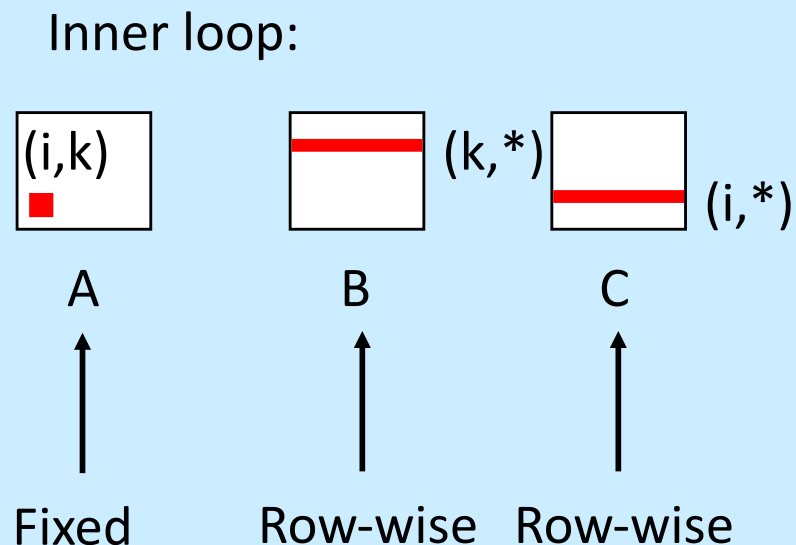
A
0.0

B
0.25

C
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];  
    }  
}
```



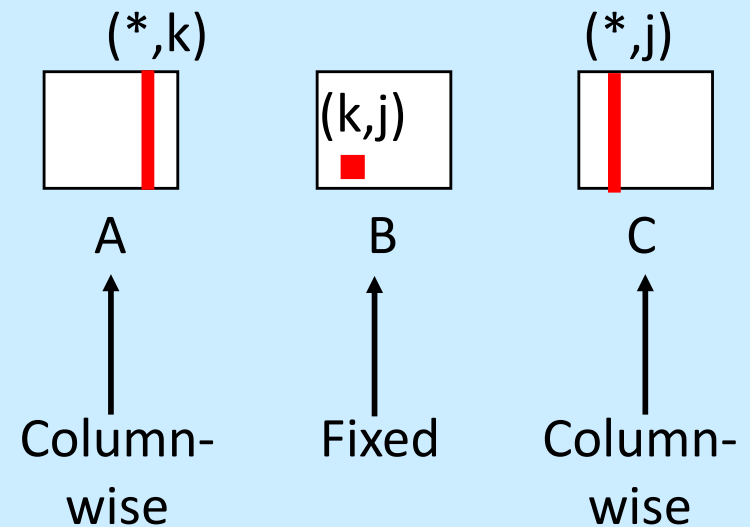
Misses per inner loop iteration:

| <u>A</u> | <u>B</u> | <u>C</u> |
|----------|----------|----------|
| 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:

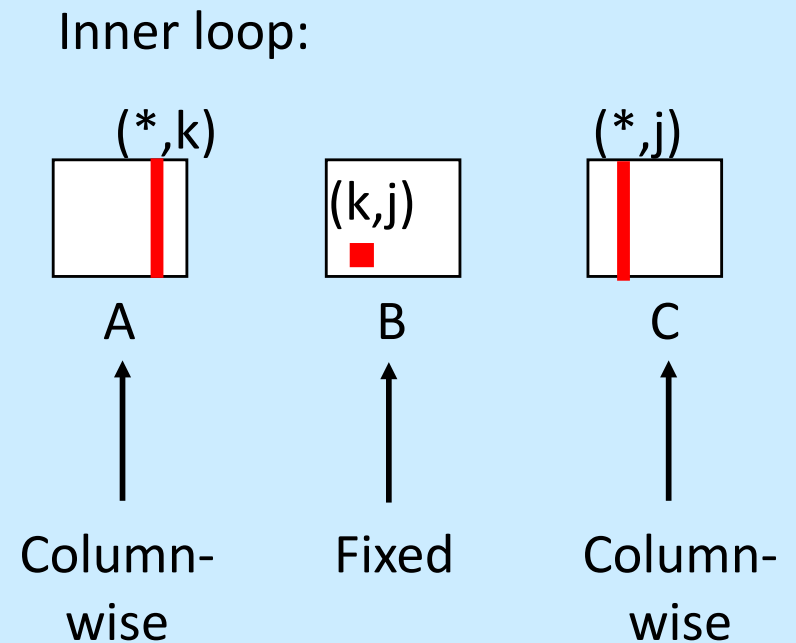
A
1.0

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;  
    }  
}
```



Misses per inner loop iteration:

| <u>A</u> | <u>B</u> | <u>C</u> |
|----------|----------|----------|
| 1.0 | 0.0 | 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 (& 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 (& ikj):

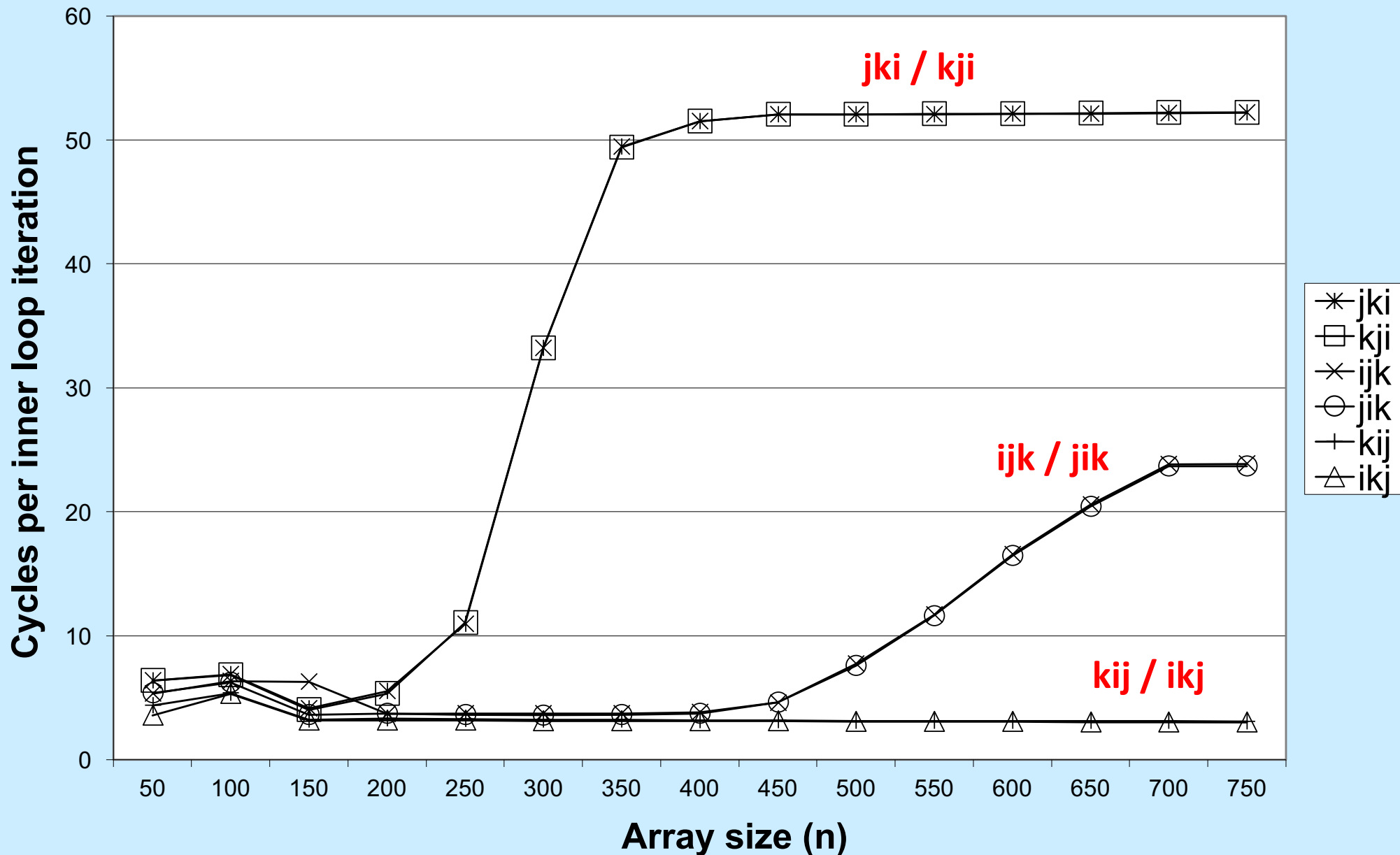
- 2 loads, 1 store
- misses/iter = **0.5**

```
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 (& kji):

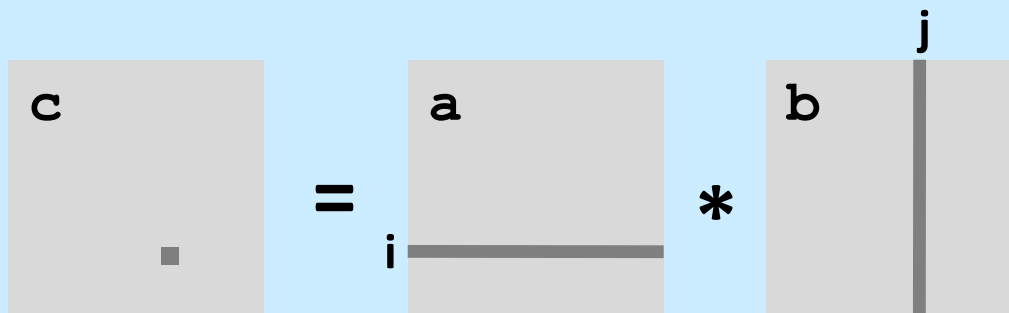
- 2 loads, 1 store
- misses/iter = **2.0**

Core i7 Matrix Multiply Performance



Matrix Multiplication: More Analysis

```
/* Multiply n x n matrices a and b */  
void mmm(int n, double a[n][n], double b[n][n], double c[n][n]) {  
    int i, j, k;  
    for (i = 0; i < n; i++)  
        for (j = 0; j < n; j++)  
            for (k = 0; k < n; k++)  
                c[i][j] += a[i][k]*b[k][j];  
}
```

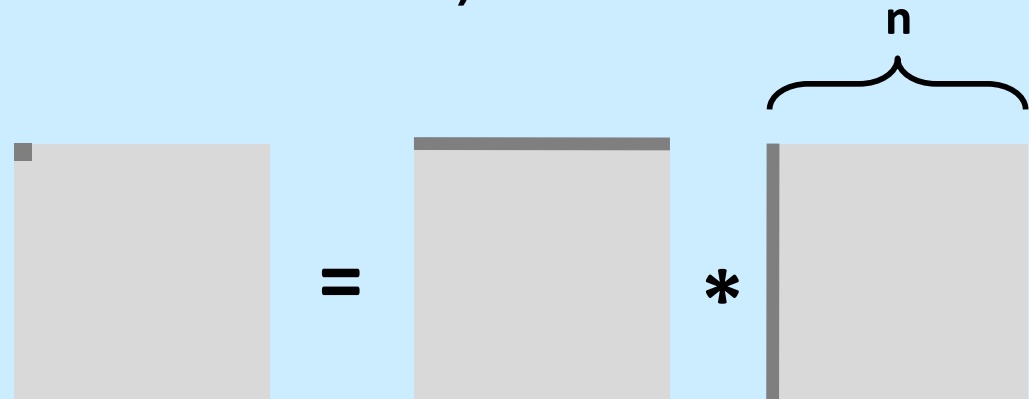


Cache-Miss Analysis

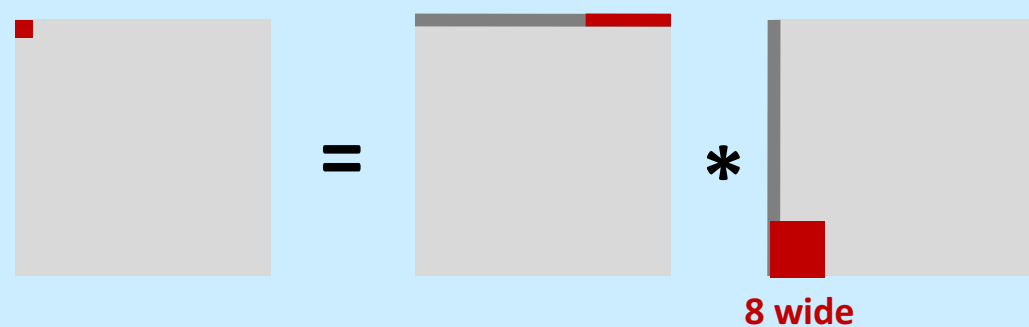
- Assume:
 - matrix elements are doubles
 - cache block = 8 doubles
 - cache size $C \ll n$ (much smaller than n)

- First iteration:

- $n/8 + n = 9n/8$ misses



- afterwards **in cache:**
(schematic)

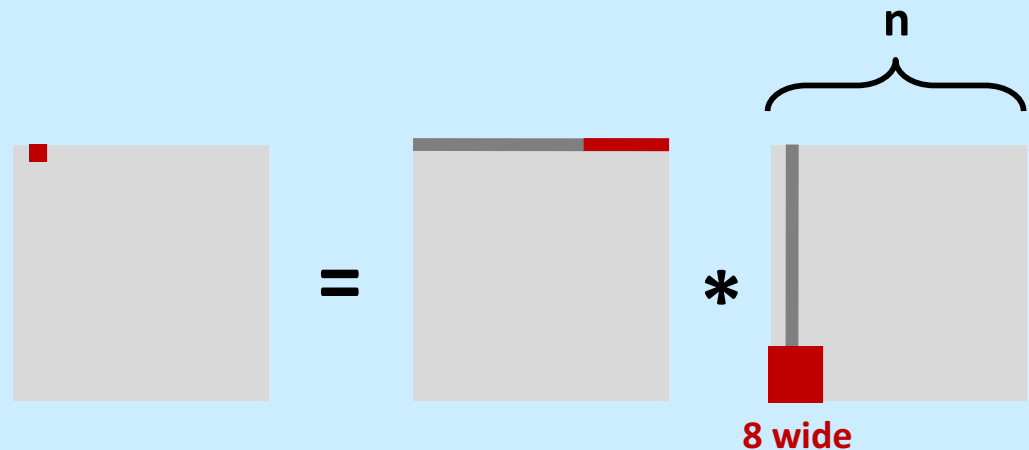


Cache-Miss Analysis

- Assume:
 - matrix elements are doubles
 - cache block = 8 doubles
 - cache size $C \ll n$ (much smaller than n)

- Second iteration:

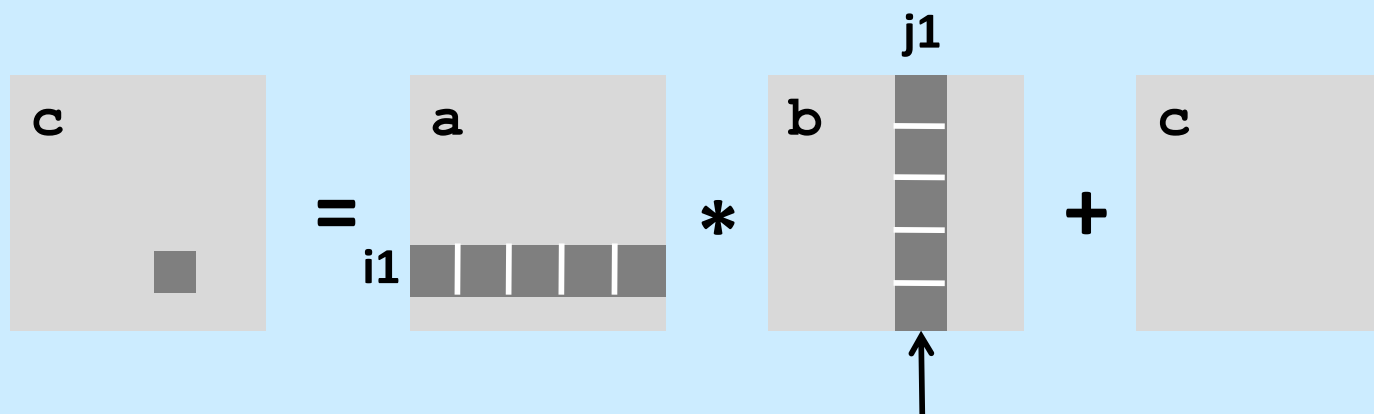
- again:
 $n/8 + n = 9n/8$ misses



- Total misses:
 - $9n/8 * n^2 = (9/8) * n^3$

Blocked Matrix Multiplication


```
/* Multiply n x n matrices a and b */
void mmm(int n, double a[n][n], double b[n][n], double c[n][n]) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                /* B x B mini matrix multiplications */
                for (i1 = i; i1 < i+B; i1++)
                    for (j1 = j; j1 < j+B; j1++)
                        for (k1 = k; k1 < k+B; k1++)
                            c[i1][j1] += a[i1][k1]*b[k1][j1];
}
```



Matrix Block size B x B

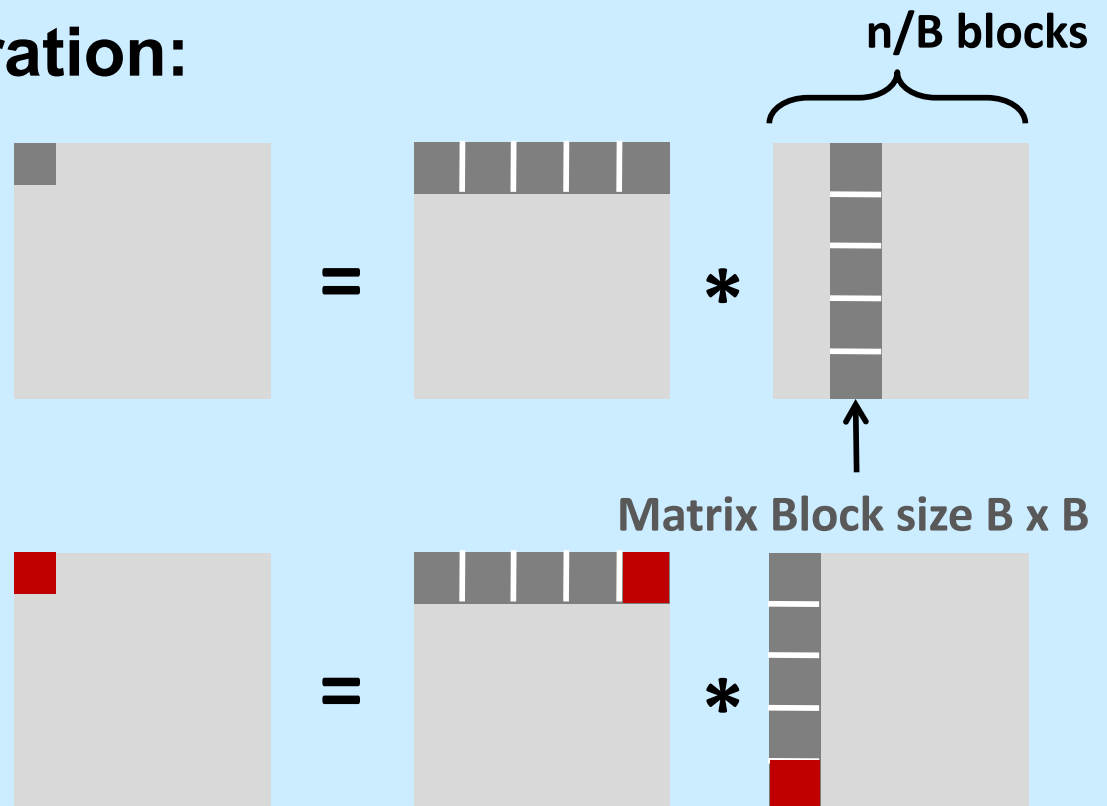
Cache-Miss Analysis

- Assume:

- cache block = 8 doubles
- cache size $C \ll n$ (much smaller than n)
- three matrix blocks  fit into cache: $3B^2 < C$


- First (matrix block) iteration:

- $B^2/8$ misses for each block
- $2n/B * B^2/8 = nB/4$ (omitting matrix c)



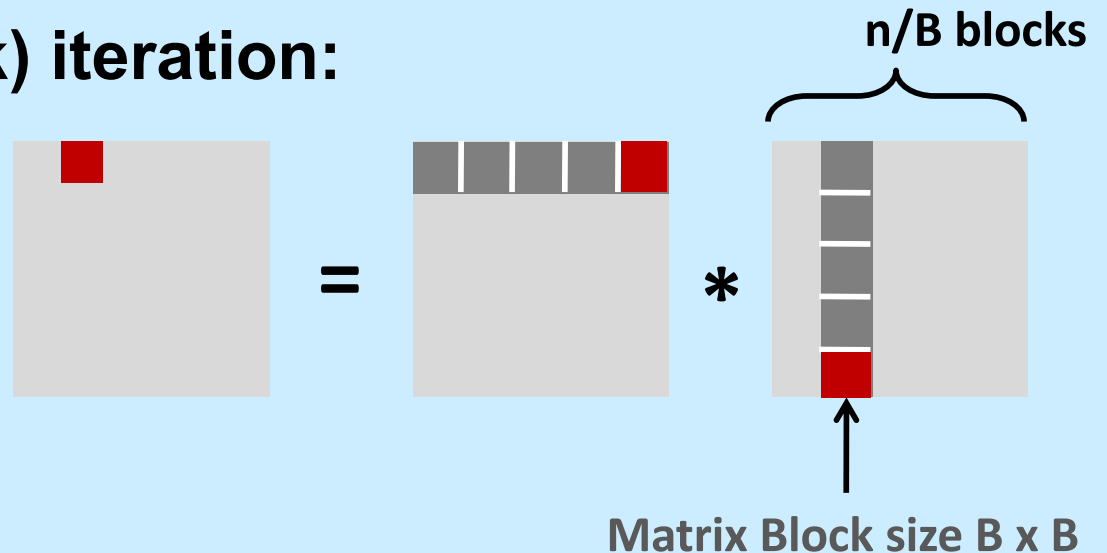
Cache-Miss Analysis

- **Assume:**

- cache block = 8 doubles
- cache size $C \ll n$ (much smaller than n)
- three matrix blocks  fit into cache: $3B^2 < C$

- **Second (matrix block) iteration:**

- same as first iteration
- $2n/B * B^2/8 = nB/4$



- **Total misses:**

- $nB/4 * (n/B)^2 = n^3/(4B)$

Summary

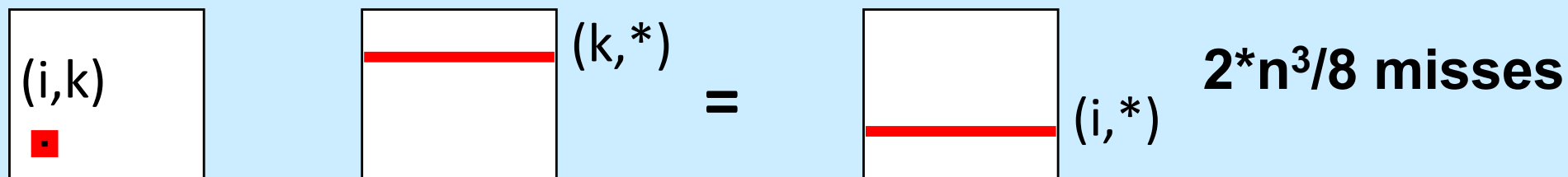
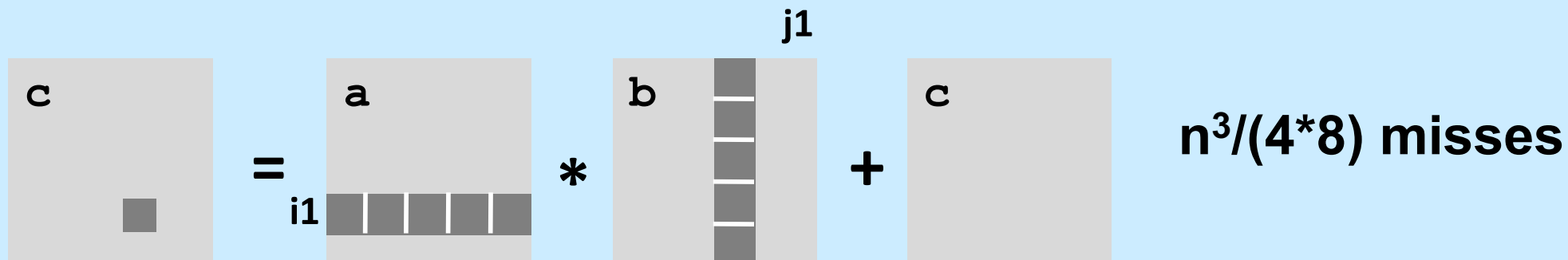
- **No blocking: $(9/8) * n^3$**
- **Blocking: $1/(4B) * n^3$**
- **Suggest largest possible block size B, but limit $3B^2 < C$!**
- **Reason for dramatic difference:**
 - **matrix multiplication has inherent temporal locality:**
 - » **input data: $3n^2$, computation $2n^3$**
 - » **every array element used $O(n)$ times!**
 - **but program has to be written properly**

Quiz 2

What is the smallest value of B (in 8-byte doubles) for which the cache-miss analysis works?

- a) 1
- b) 2
- c) 4
- d) 8

Blocking vs. ikj



Blocking vs. ikj

```
$ ./matmult_ikj
```

```
ikj: .608 secs
```

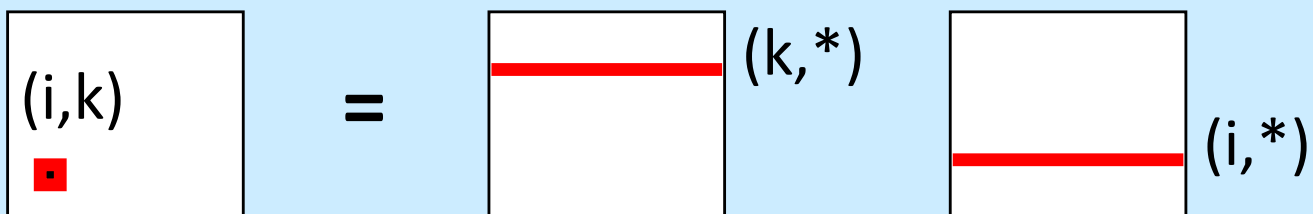
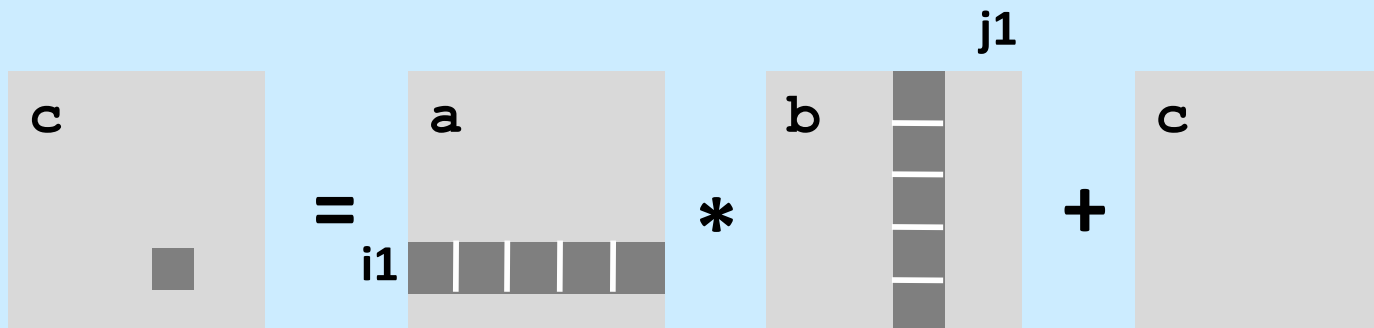
```
$ ./matmult_Blocked
```

```
Blocked: .880 secs
```

Why is ikj Faster?

- Prefetching

- the processor detects sequential (stride-1) accesses to memory and issues loads before they are needed



Concluding Observations

- **Programmer can optimize for cache performance**
 - organize data structures appropriately
 - take care in how data structures are accesses
 - » 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)