CS 33

Exploiting Caches

2-Way Set-Associative Cache Simulation

t=2	s=1	b=1
xx	x	x

M=16 byte addresses, B=2 bytes/block,
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

0	**$[00\underline{0}0_2]$,**	miss
1	**$[00\underline{0}1_2]$,**	hit
7	**$[01\underline{1}1_2]$,**	miss
8	**$[10\underline{0}0_2]$,**	miss
0	**$[00\underline{0}0_2]$**	hit

	v	**Tag**	**Block**
Set 0	1	00	M[0-1]
	1	10	M[8-9]
Set 1	1	01	M[6-7]
	0		

A Higher-Level Example

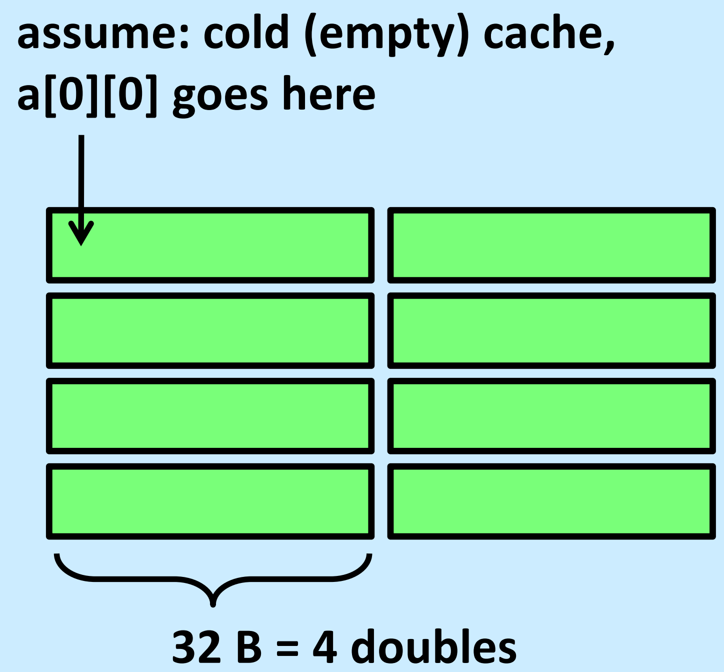
Ignore the variables sum, i, j

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; i++)
        for (i = 0; i < 16; j++)
            sum += a[i][j];
    return sum;
}
```



A Higher-Level Example

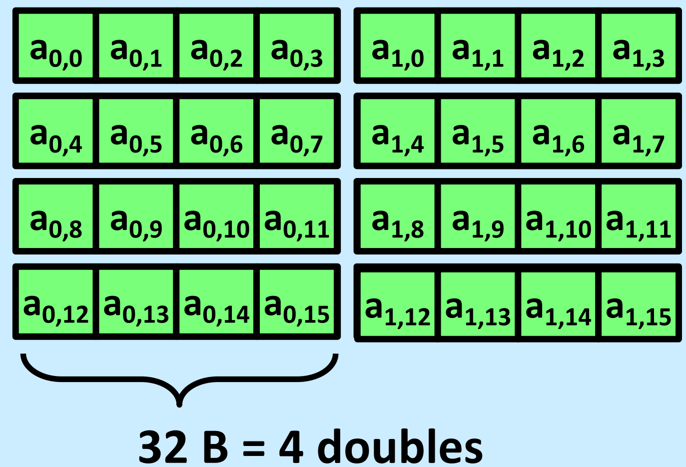
Ignore the variables sum, i, j

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; i < 16; i++)
        for (i = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```



A Higher-Level Example

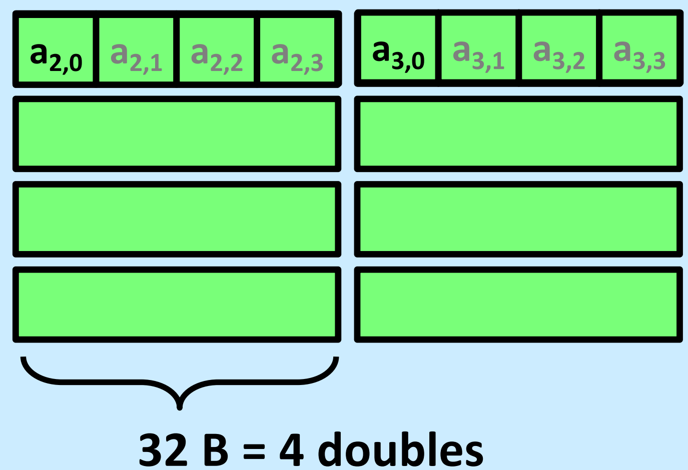
Ignore the variables sum, i, j

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; i < 16; i++)
        for (i = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

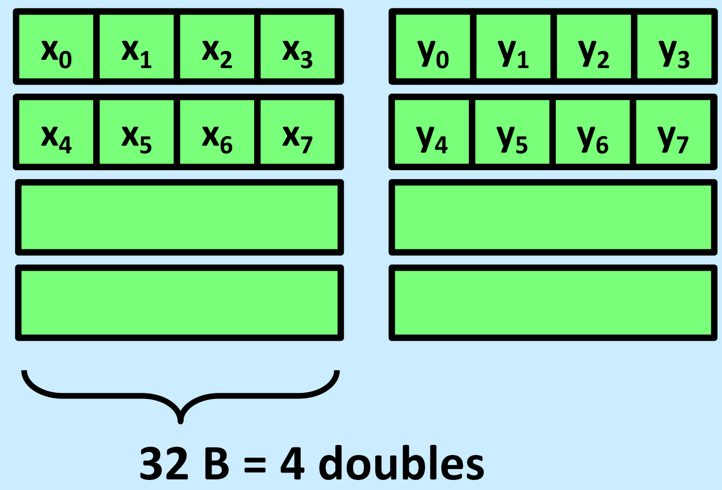


Conflict Misses

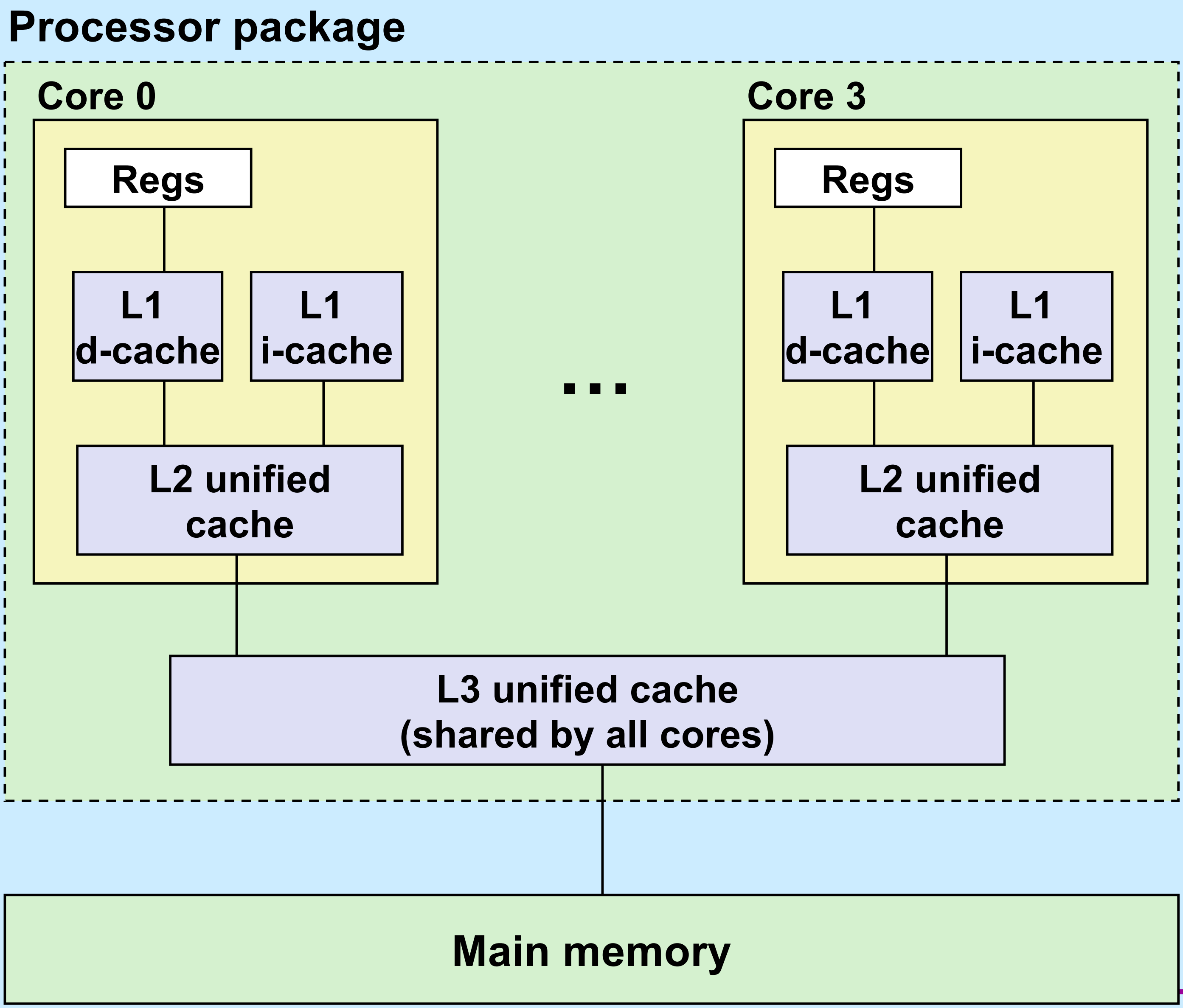
```
double dotprod(double x[8], double y[8]) {
  double sum = 0.0;
  int i;

  for (i=0; i<8; i++)
    sum += x[i] * y[i];

  return sum;
}
```



Intel Core i5 and i7 Cache Hierarchy



L1 i-cache and d-cache:
32 KB, 8-way,
Access: 4 cycles

L2 unified cache:
256 KB, 8-way,
Access: 11 cycles

L3 unified cache:
8 MB, 16-way,
Access: 30-40 cycles

Block size: 64 bytes for all caches

What About Writes?

- Multiple copies of data exist:
  - L1, L2, main memory, disk
- What to do on a write-hit?
  - write-through (write immediately to memory)
  - write-back (defer write to memory until replacement of line)
    - » need a dirty bit (line different from memory or not)
- What to do on a write-miss?
  - write-allocate (load into cache, update line in cache)
    - » good if more writes to the location follow
  - no-write-allocate (writes immediately to memory)
- Typical
  - write-through + no-write-allocate
  - write-back + write-allocate

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
- **99% hit rate is twice as good as 97%!**
  - consider:
    cache hit time of 1 cycle
    miss penalty of 100 cycles
  - average access time:

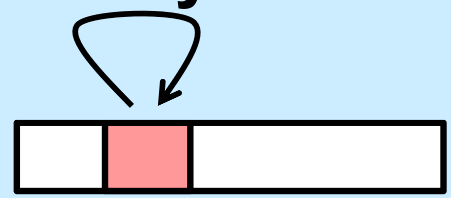
    97% hits: .97 * 1 cycle + 0.03 * 100 cycles ≈ 4 cycles

    99% hits: .99 * 1 cycle + 0.01 * 100 cycles ≈ 2 cycles

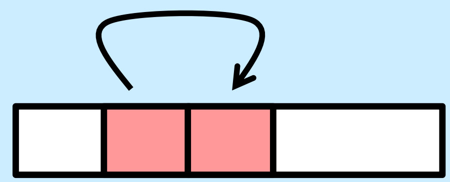
- **This is why “miss rate” is used instead of “hit rate”**

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. — Spatial locality
  - cycle through loop repeatedly — Temporal locality

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)

Matrix Multiplication Example

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

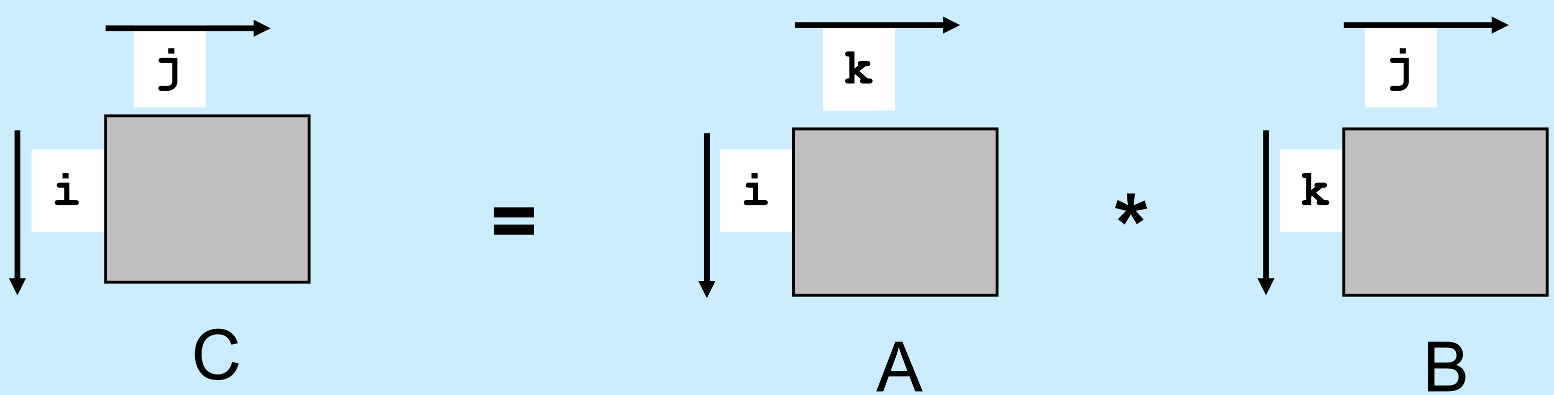
Variable sum held in register

```
/* 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;
  }
}
```

```
/* 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 = 64B (big enough for eight 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



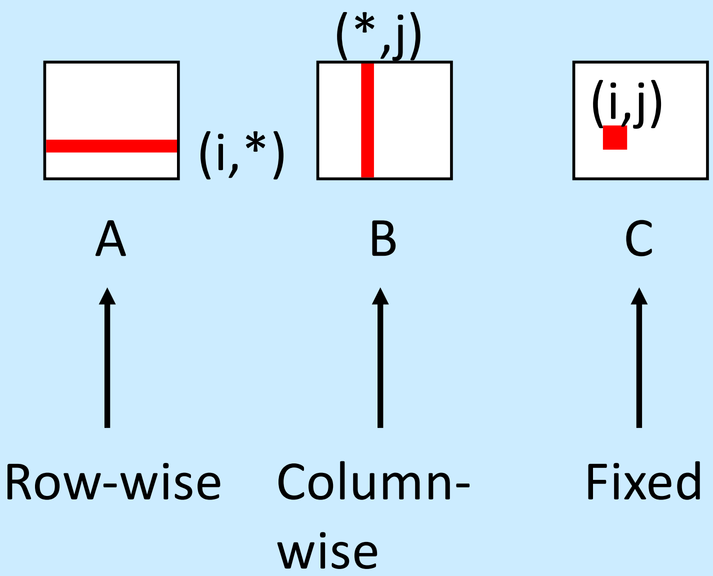
Layout of C Arrays in Memory (review)

- **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) > 8 bytes, exploit spatial locality**
    - » **compulsory miss rate = 8 bytes / Block**
- **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:



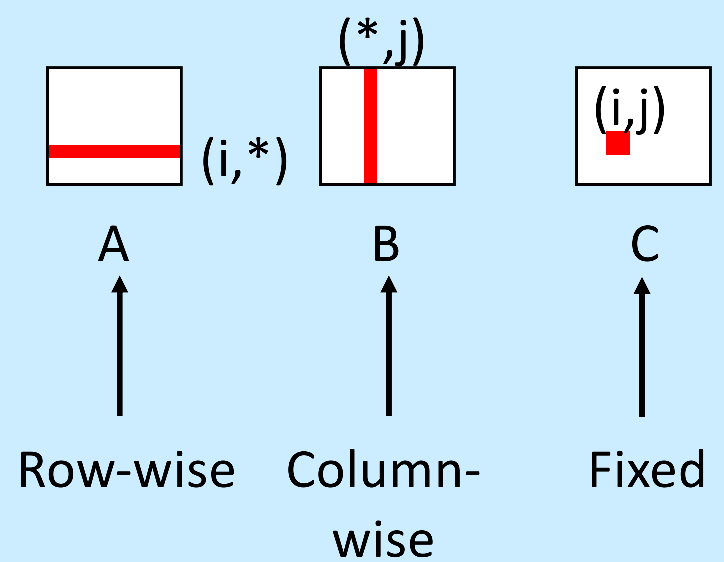
Misses per inner loop iteration:

A	B	C
0.125	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:

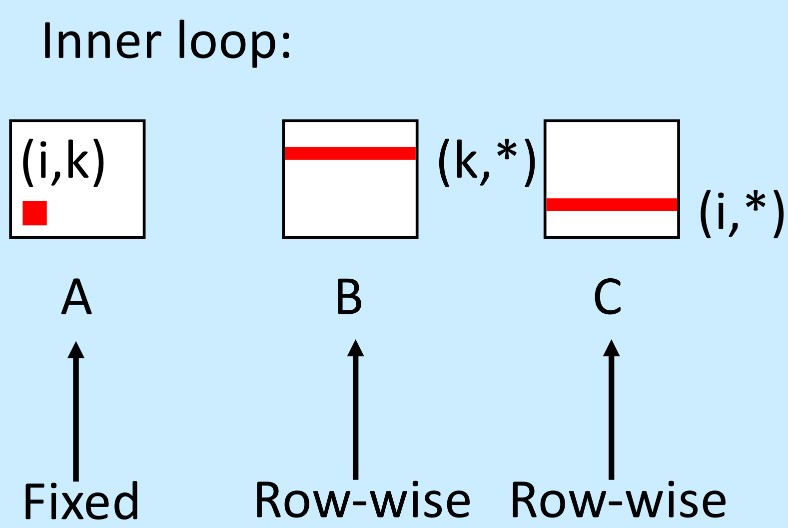


Misses per inner loop iteration:

A	B	C
0.125	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];
  }
}
```



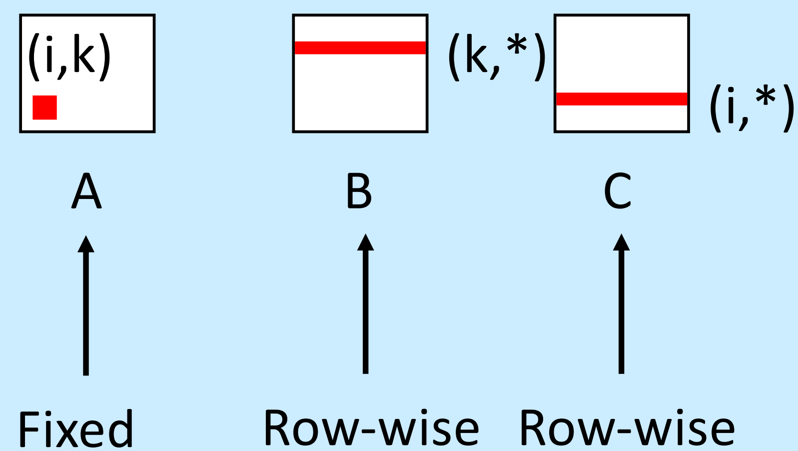
Misses per inner loop iteration:

A	B	C
0.0	0.125	0.125

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:



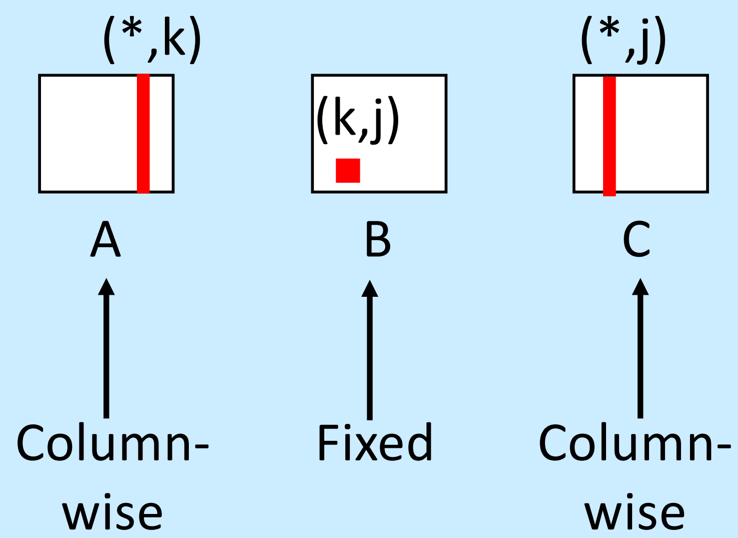
Misses per inner loop iteration:

A	B	C
0.0	0.125	0.125

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:



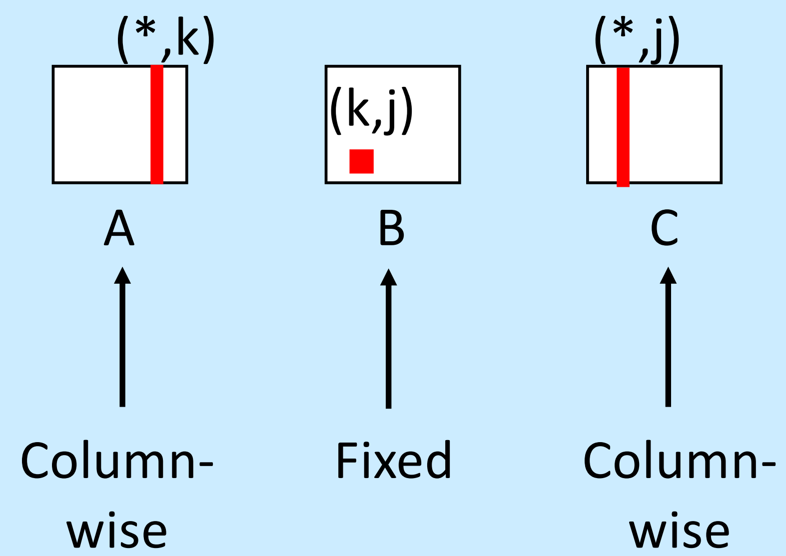
Misses per inner loop iteration:

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:



Misses per inner loop iteration:

A	B	C
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 = **1.125**

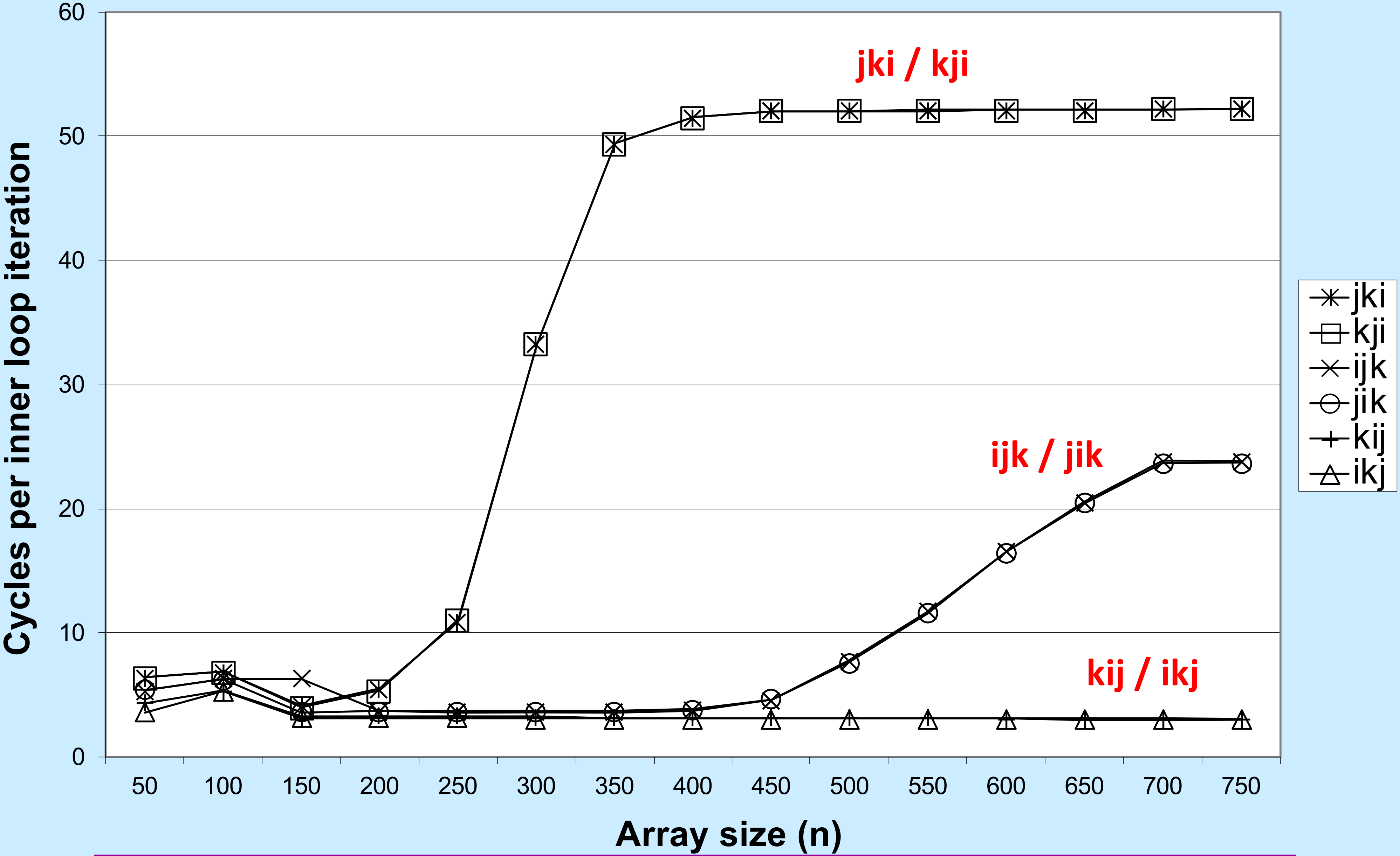
```
for (k=0; k<n; k++)
 for (i=0; i<n; i++) {
  r = a[i][k];
  for (j=0; j<n; j++)
   c[i][j] += r * b[k][j];
  }
```

kij (& ikj):
- 2 loads, 1 store
- misses/iter = **0.25**

```
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;
 }
```

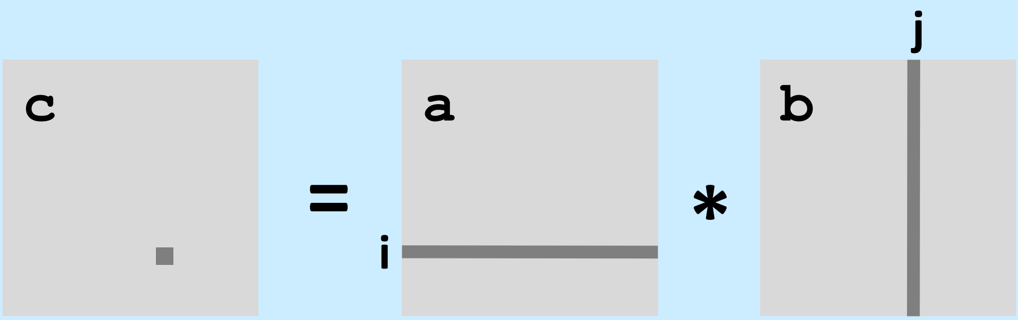
jki (& kji):
- 2 loads, 1 store
- misses/iter = **2.0**

Core i7 Matrix Multiply Performance



Matrix Multiplication: More Analysis

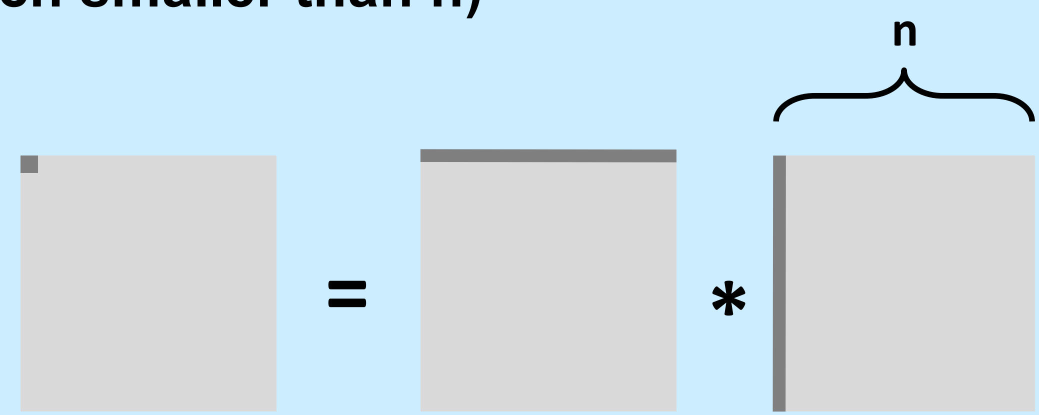
```
/* Multiply n x n matrices a and b  */
void mmm(int n, double a[n][n], double b[n][n], double c[n][n]) {
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
              for (k = 0; k < n; k++)
                c[i][j] += a[i][k]*b[k][j];
}
```



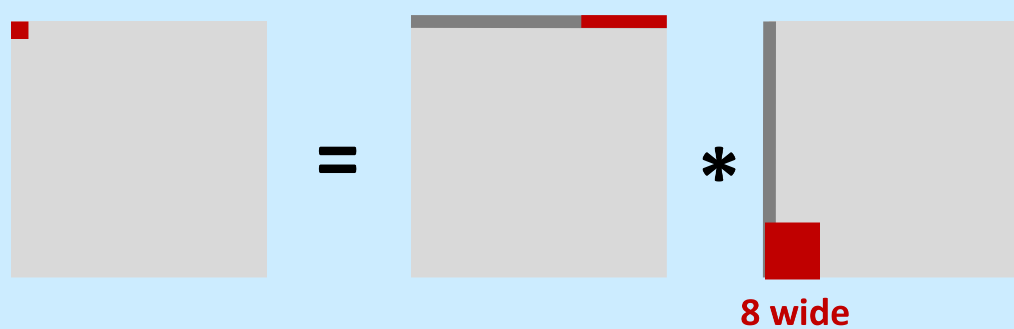
Cache-Miss Analysis

- **Assume:**
  - matrix elements are doubles
  - cache block = 8 doubles
  - cache size C << n (much smaller than n)

- **First iteration:**
  - n/8 + n = 9n/8 misses

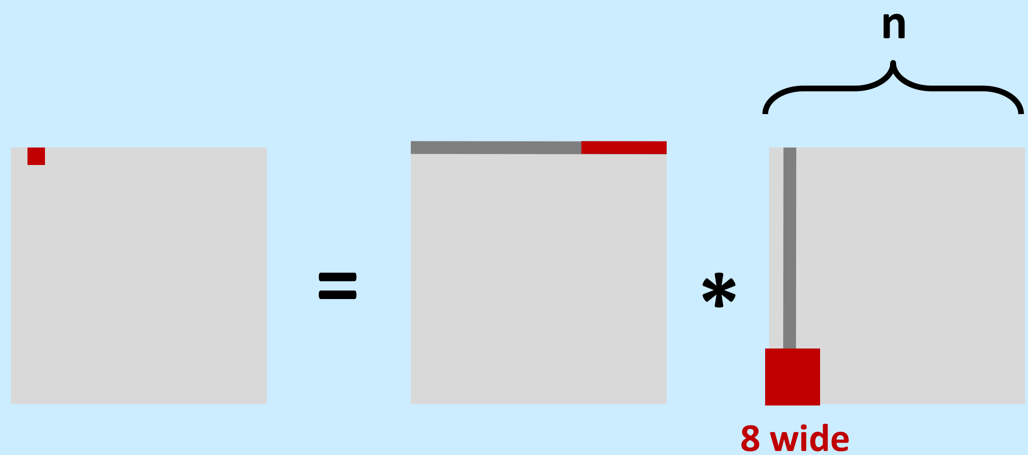


  - afterwards in cache: (schematic)



Cache-Miss Analysis

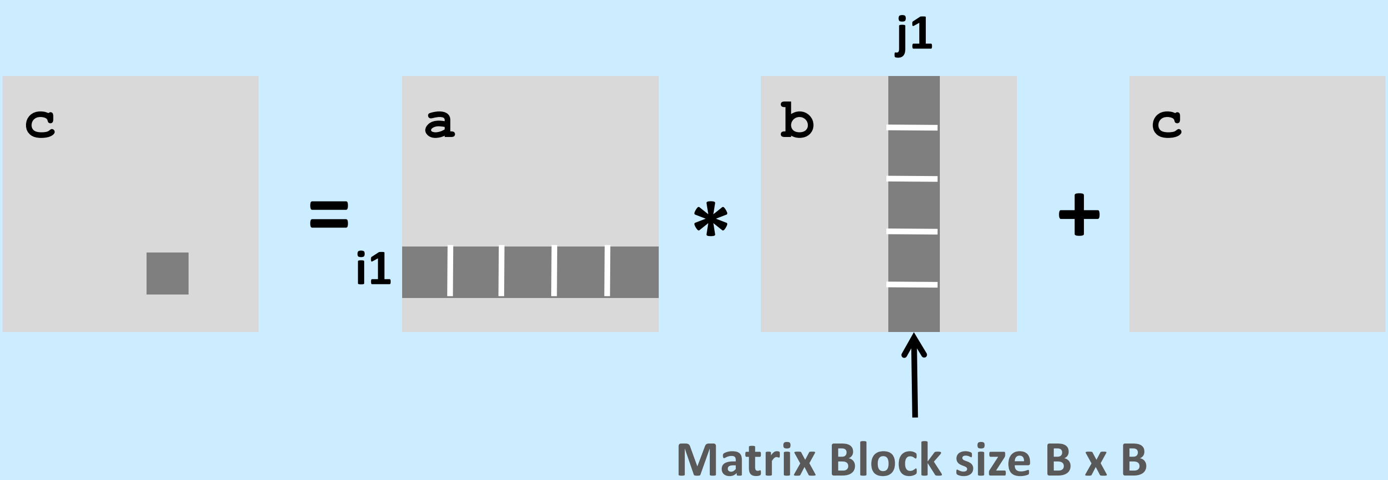
- **Assume:**
  - matrix elements are doubles
  - cache block = 8 doubles
  - cache size C << n (much smaller than n)
- **Second iteration:**
  - again:
    $n/8 + n = 9n/8$ misses



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

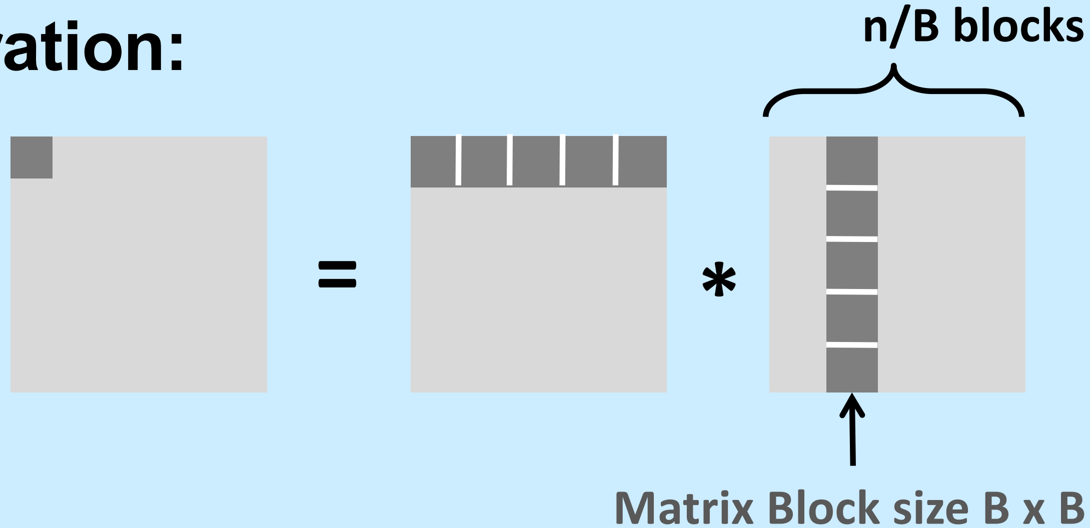
Blocked Matrix Multiplication

```
/* 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];
}
```

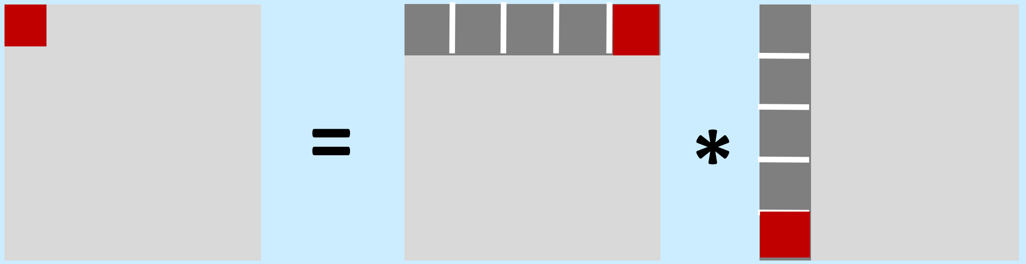


Cache-Miss Analysis

- **Assume:**
  - cache block = 8 doubles
  - cache size C << 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)



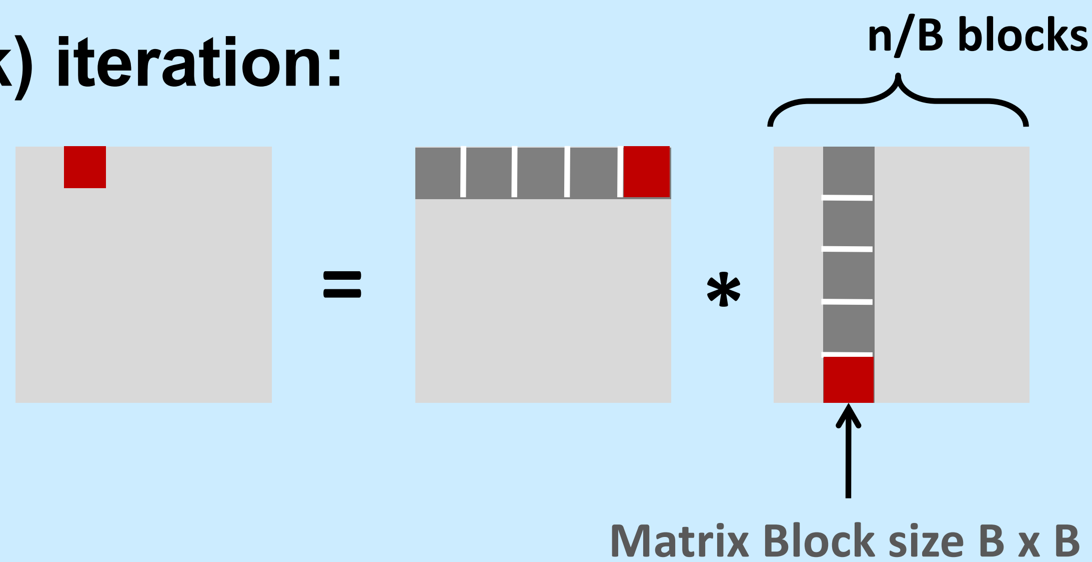
  - afterwards in cache (schematic)



Cache-Miss Analysis

- **Assume:**
  - cache block = 8 doubles
  - cache size C << 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

Our analysis assumes a cache line of 64 bytes. What is the smallest value of B (in 8-byte doubles) for which the cache-miss analysis works? (Hint: each fetch of memory retrieves a complete cache line.)

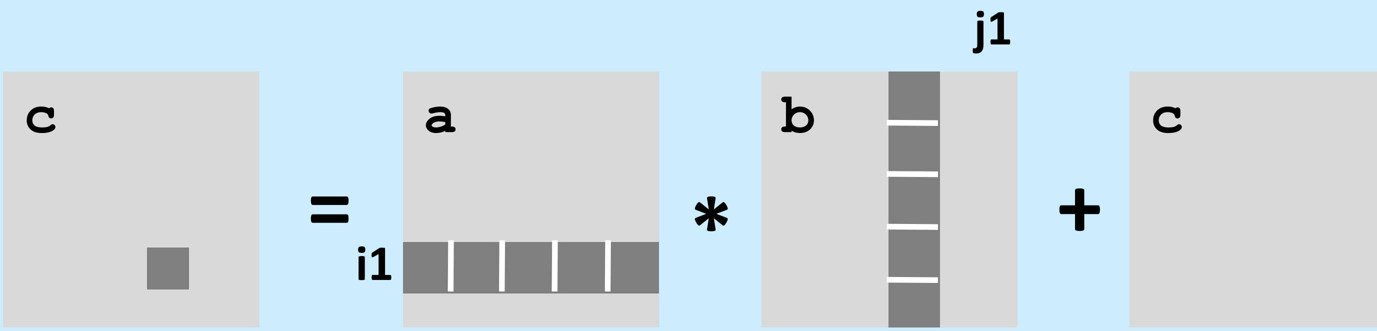
a) 1

b) 2

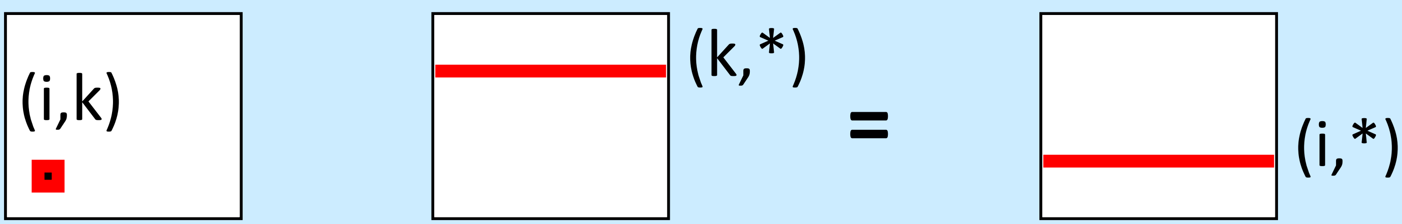
c) 4

d) 8

Blocking vs. ikj



$n^3/(4*8)$ misses



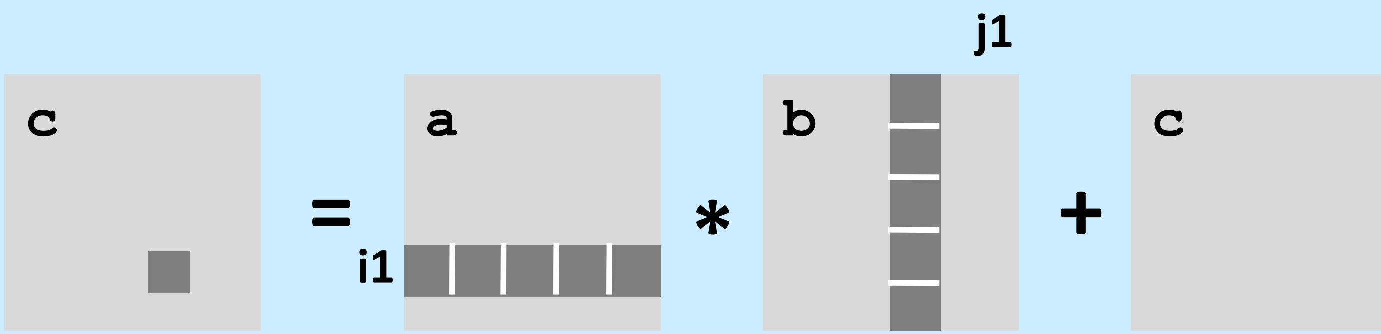
$2*n^3/8$ misses

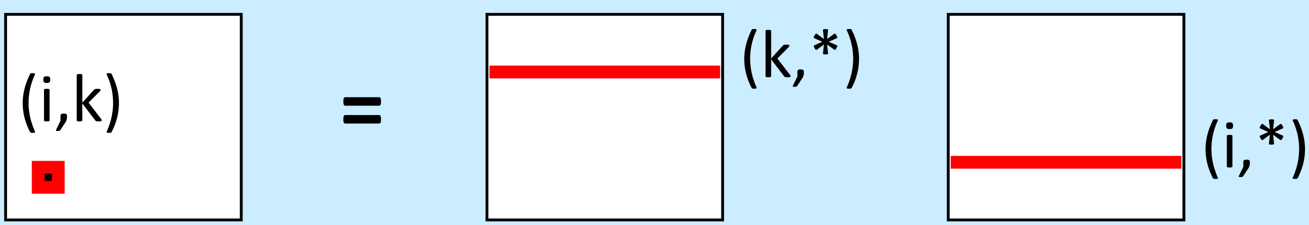
Blocking vs. ikj

```
$ ./matmult_Blocked
Blocked: 1.154 secs
$ ./matmult_ikj
ikj: .699 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)