

Basic Techniques to Account for Memory

Outline

1. Preliminaries: Mapping data structures into memory
2. Simple Example: 2D Array Reduction
3. Real Example: Matrix-Matrix Multiply
 - 3 loop baseline code
 - Loop interchange
 - Blocking

Basic Techniques to Account for Memory

Q: How often do you think about your memory hierarchy when writing code?

A: Didn't think so!

Thought problem: Let's say you were creating code to exercise your memory hierarchy --

1. ***How good can you make your code?***



2. ***How bad can you make your code?***



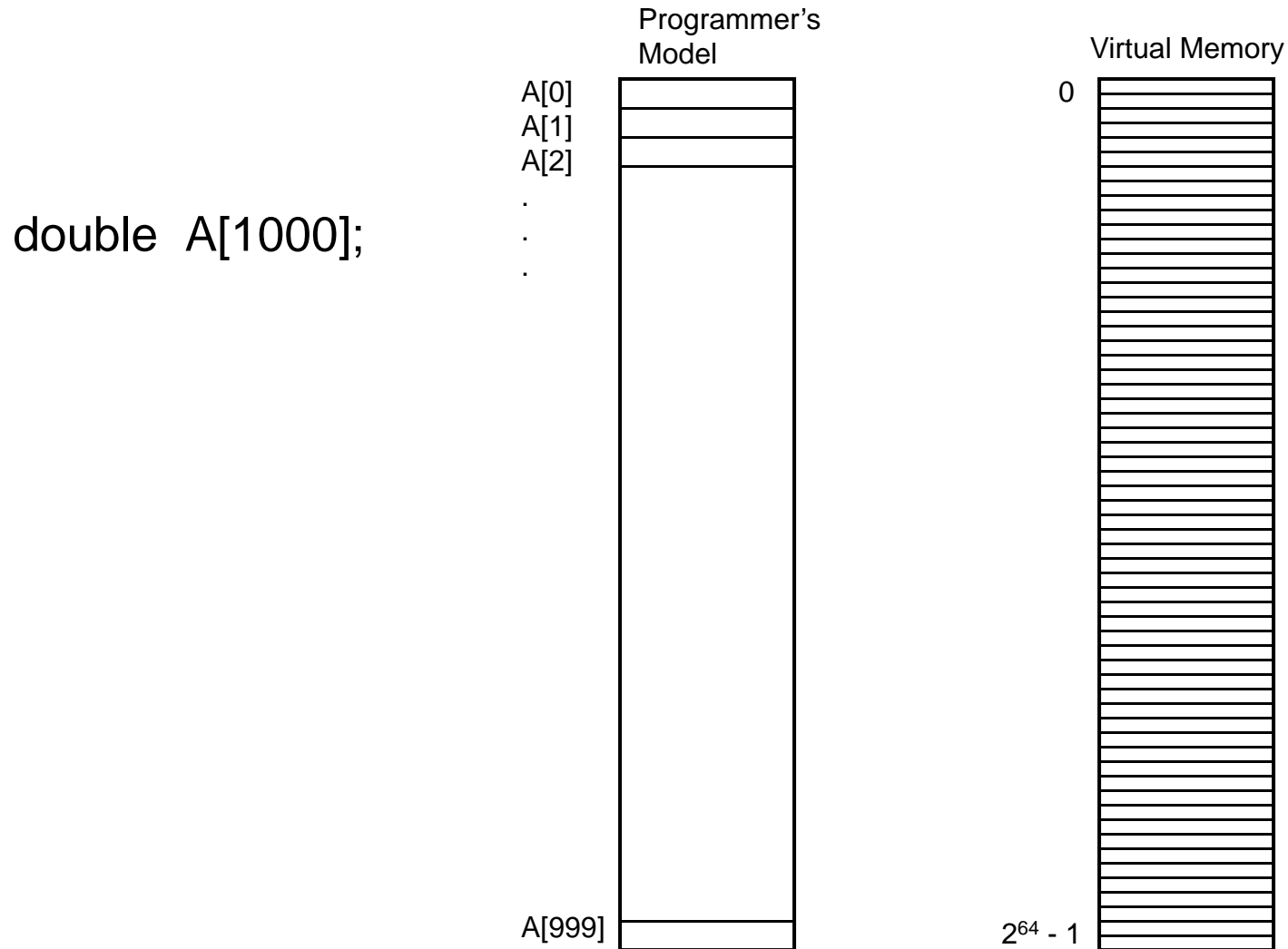
Clearly, code that always hits L1 cache will get excellent performance 😊, while code that misses all levels of cache frequently will have extremely poor performance ☹.

Memory hierarchies are designed to take advantage of spatial and temporal locality. But most computationally intensive codes initially do not have as much locality as it is possible to extract from the application.

Our job: *help out the memory hierarchy.*

Mapping Arrays to Memory – 1D

For 1D arrays, it's easy – we progress sequentially: element 1 goes into word one, element 2 into word two of memory, etc.



Mapping Arrays to Memory – 2D

2D case appears to be easy as well. Rows are just like 1-D arrays and each row we append sequentially onto the end of the previous.

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

Programmer's Model

A	0	1	2	3	4
0					
1					
2					
3					

Virtual Memory

[illegible]

Or is it so easy? Most languages have row major ordering, but FORTRAN has just the opposite!!!

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

Programmer's Model

A	0	1	2	3
0				
1				
2				
3				
4				

Writing Cache Friendly Code

But this is only part of the story. What's more important is how layout interacts with access.

→ *If the data structure fits in L1 cache and reuse is high, then access pattern does not matter.*

Otherwise: What is the access pattern?

→ Is there reuse? Over what time?

temporal locality

→ Are adjacent elements accessed in sequence?

spatial locality

Some general guidelines:

- 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.

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%)

Example 1: Accumulate a 2D Array

Task: Add up all the elements in a 2D array.

Computational Intensity = 1.

- Each datum used once, so no temporal locality, so no reuse.
- Work on Cold Misses, maximize spatial locality

Dependencies: None. >> If there are any, this is a problem with the code

- Nothing prevents access being done in any order
- Code should not prevent arbitrary reordering of fetches

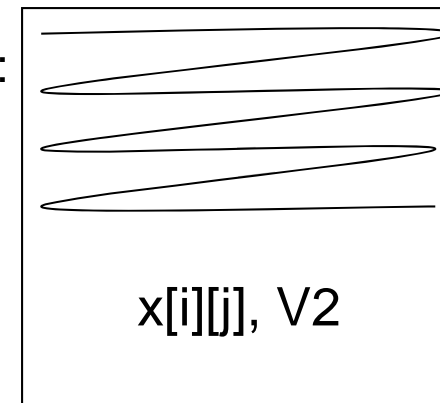
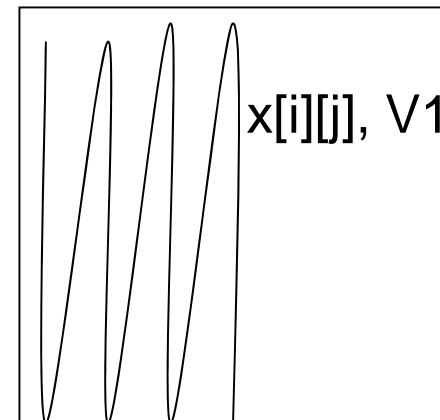
Clearly → Memory Bound

Let's assume row-major ordering. It's perfectly reasonable (with no knowledge of memory lay out and caching) to write code like this (V1):

```
/* version 1 */
A = 0;
for (j = 0; j < 1000; j++)
    for (i = 0; i < 1000; i++) A += x[i][j];
```

It's just as reasonable to write the code like this (V2):

```
/* version 2 */
A = 0;
for (i = 0; i < 1000; i++)
    for (j = 0; j < 1000; j++) A += x[i][j];
```



Ignore the variables sum, i, j

```
int sum_array_rows_v1(double a[4][4])
{
    int i, j;
    double sum = 0;

    for (j = 0; i < 4; i++)
        for (i = 0; j < 4; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_rows_v2(double a[4][4])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
            sum += a[i][j];
    return sum;
}
```

assume: cold (empty) cache,
a[0][0] goes here



32 B = 4 doubles

Ignore the variables sum, i, j

```
int sum_array_rows_v1(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;
}
```

```
int sum_array_rows_v2(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;
}
```

assume: cold (empty) cache,
a[0][0] goes here



32 B = 4 doubles

Another view – more realistic memory

Compare memory hierarchy performance, assuming a cold cache – **1st try:**

Assume: L1 = 16KB, L2 = 256KB, L3 = 8MB, word size = 4B, block size = 8B

V1:

First iteration → miss on all of the first 1000 accesses

- references are 1000 words apart in memory and references will not benefit from any free prefetching due to being part of the same cache block

Second iteration → on next 1000 accesses, we get L1 hits

Iteration 3+ → same pattern (all misses on odd, all L1 hits on even) continues

V2:

Each miss will be followed by one L1 hit.

V1 vs. V2: here, performance is roughly the same.

Comment: This is called ***loop interchange***.

Why is it allowed? No dependencies: Independent instructions can be reordered at will.

Compare memory hierarchy performance, assuming a cold cache – **2nd try**:

Assume: L1 = **4KB**, L2 = 256KB, L3 = 8MB, word size = 4B, block size = **32B**
(note smaller L1 cache)

V1:

First iteration → miss on all of the first 1000 accesses

- references are 1000 words apart in memory and references will not benefit from any free prefetching due to being part of the same cache block

Iterations 2-7 → all miss L1 cache, but all hit L2 cache.

V2:

Each miss will be followed by 7 L1 hits.

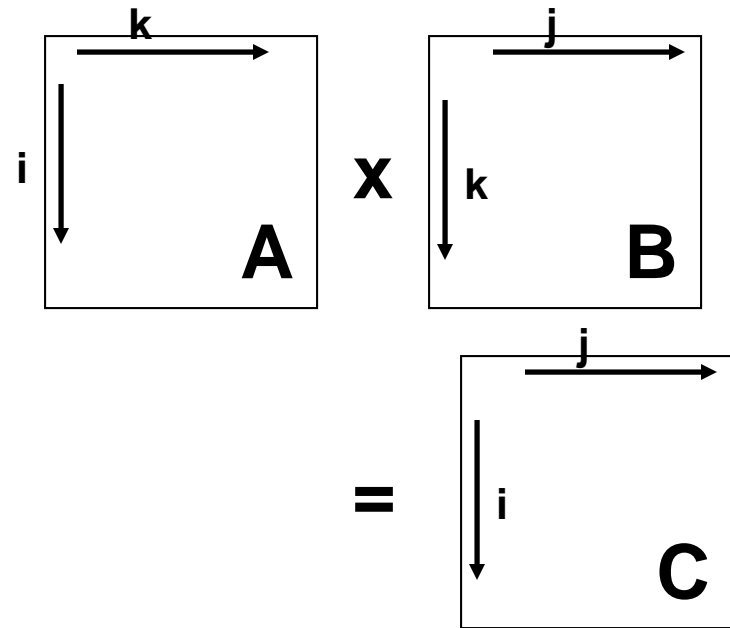
V1 vs. V2: V1 has an L1 miss rate of 100%, V2 has an L1 miss rate of 12.5%.

Both have L2 miss rates of 12.5%.

Example 2: Matrix-Matrix Multiply

Code with no reuse is neither very interesting, nor very common. Here's a critical, everyday example: matrix multiply $\rightarrow A \times B = C$.

```
for (i = 0; i < N; i++)  
  for (j = 0; j < N; j++) {  
    r = 0;  
    for (k = 0; k < N; k++)  
      r = r + y[i][k] * z[k][j];  
    x[i][j] = r;  }
```



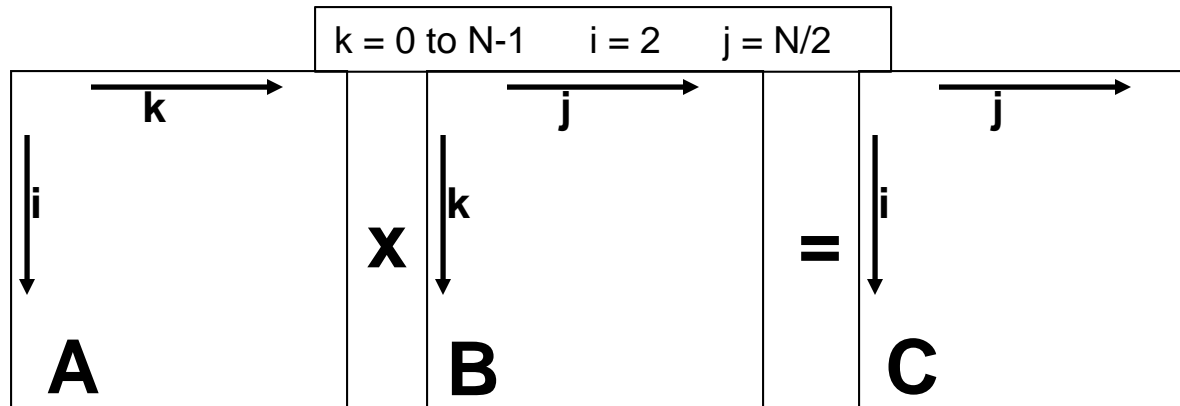
Compute cost for NxN matrices

- N^3 multiplications
- N^3 additions
- $= 2N^3$ floating point operations
- Computational intensity = $N \rightarrow$ ***Should be CPU Bound***

Dependencies

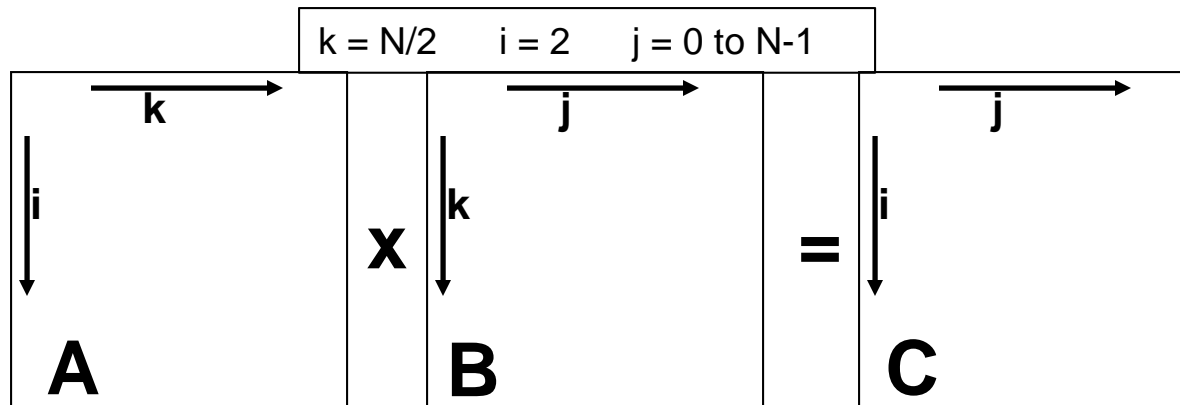
- There are no inherent data or control dependencies
- Nothing in the code should prevent operations from being reordered arbitrarily
- However, pairs of elements are used together systematically

How are elements coupled?

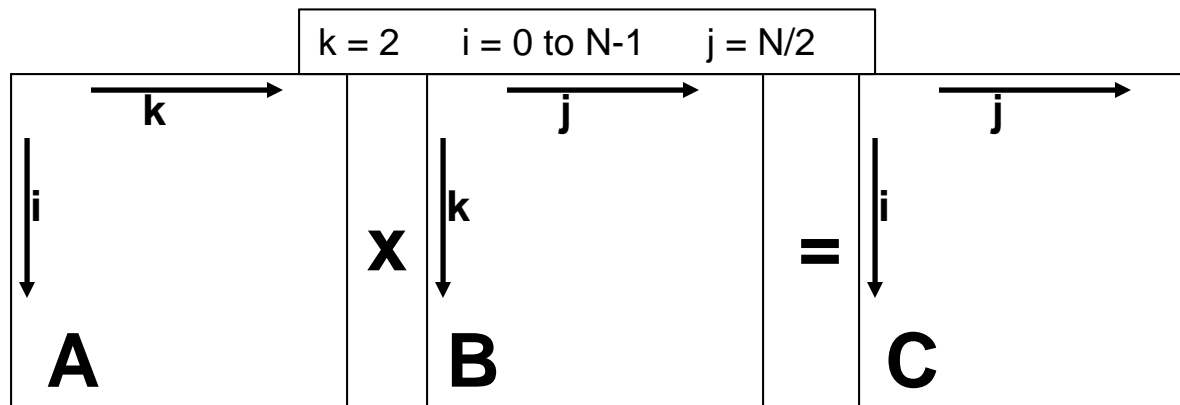


```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++) {
    r = 0;
    for (k = 0; k < N; k++)
      r = r + y[i][k] * z[k][j];
    x[i][j] = r;
  }
```

For any element of **C**, which elements of **A** and **B** affect it?



For any element of **A**, which elements of **B** is it coupled with and which elements of **C** does it affect?



For any element of **B**, which elements of **A** is it coupled with and which elements of **C** does it affect?

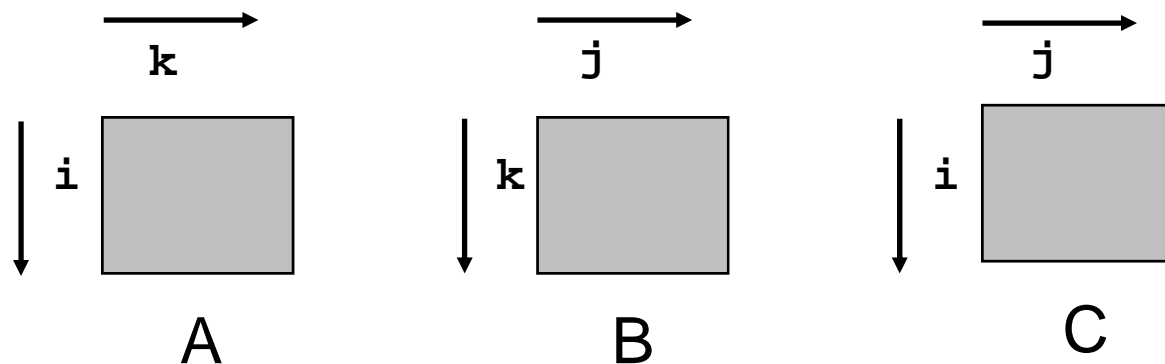
Miss Rate Analysis for Matrix Multiply

■ Assume:

- Line size = 32B (big enough for four 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



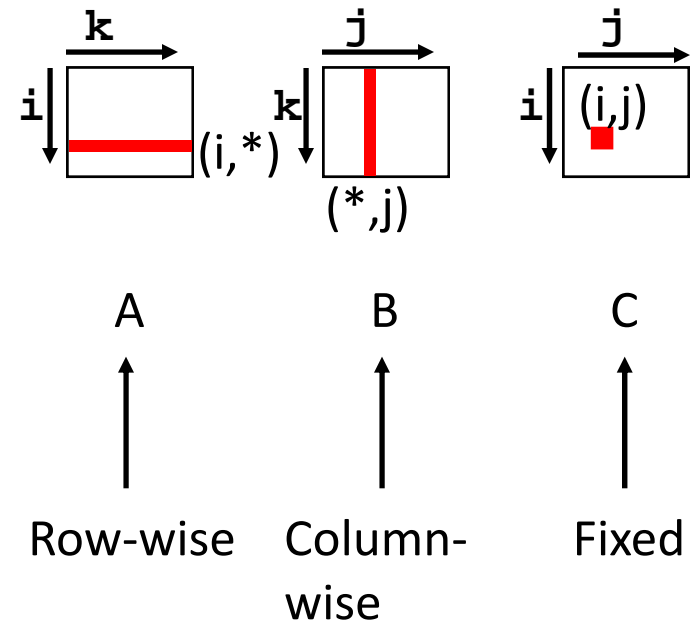
Matrix Multiplication – ijk (jik similar)

```

/* 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

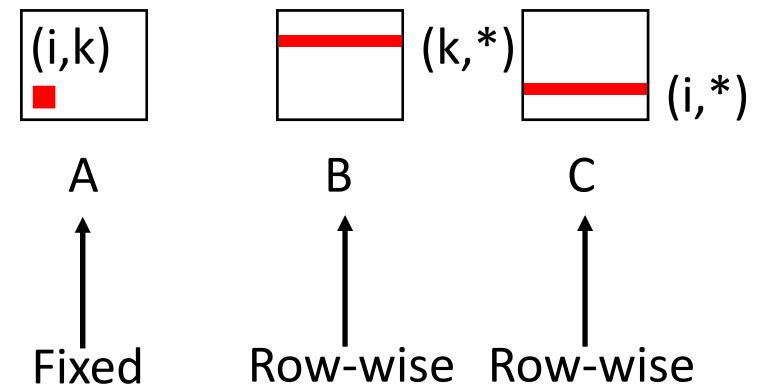
Matrix Multiplication – kij (ikj similar)

```

/* 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:

<u>A</u>	<u>B</u>	<u>C</u>
0.0	0.25	0.25

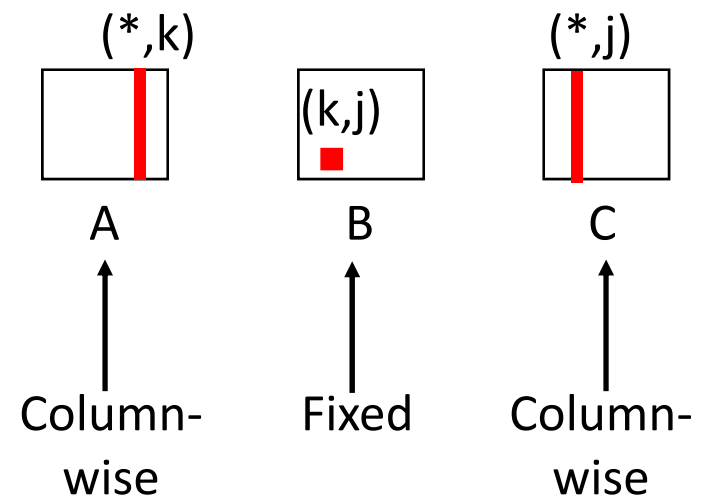
Matrix Multiplication – jki (kji similar)

```

/* 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:

<u>A</u>	<u>B</u>	<u>C</u>
1.0	0.0	1.0

Summary of Matrix Multiplication – loop interchange

```
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):

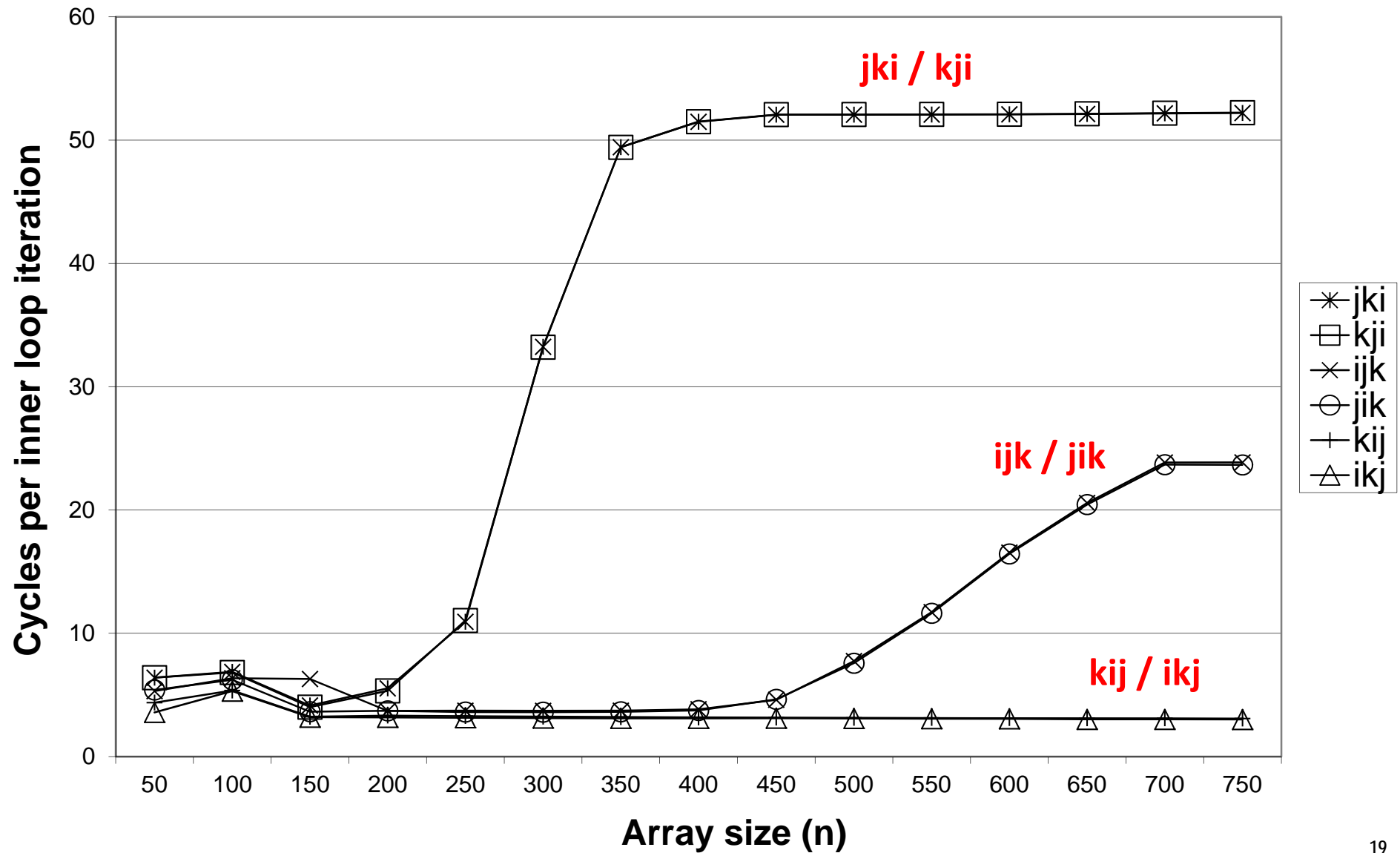
- 2 loads, 1 store
- misses/iter = **0.5**

```
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



Cache Miss Analysis

```
c = (double *) calloc(sizeof(double), n*n);

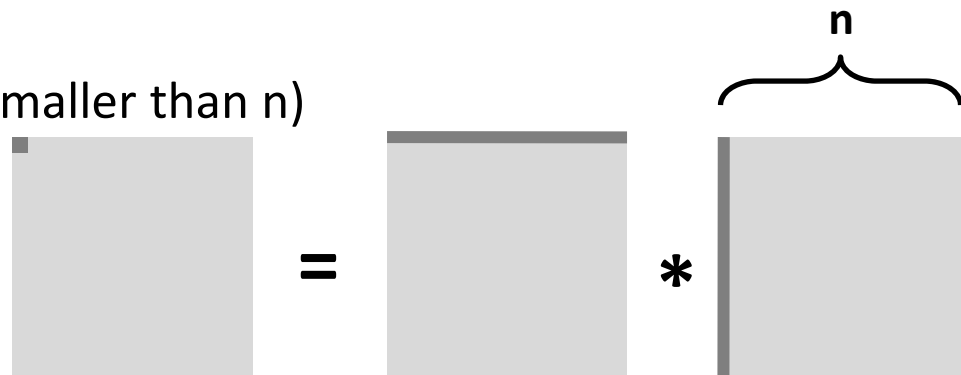
/* Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n+j] += a[i*n + k]*b[k*n + j];
}
```

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)



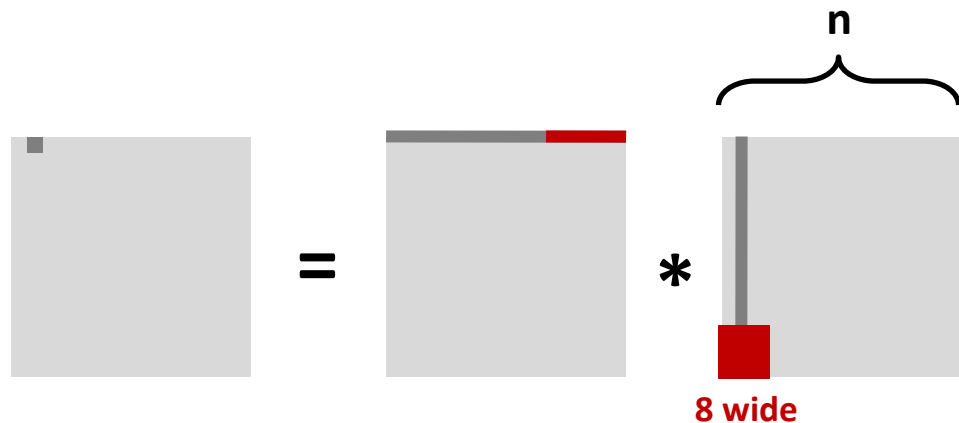
Cache Miss Analysis

■ 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:

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

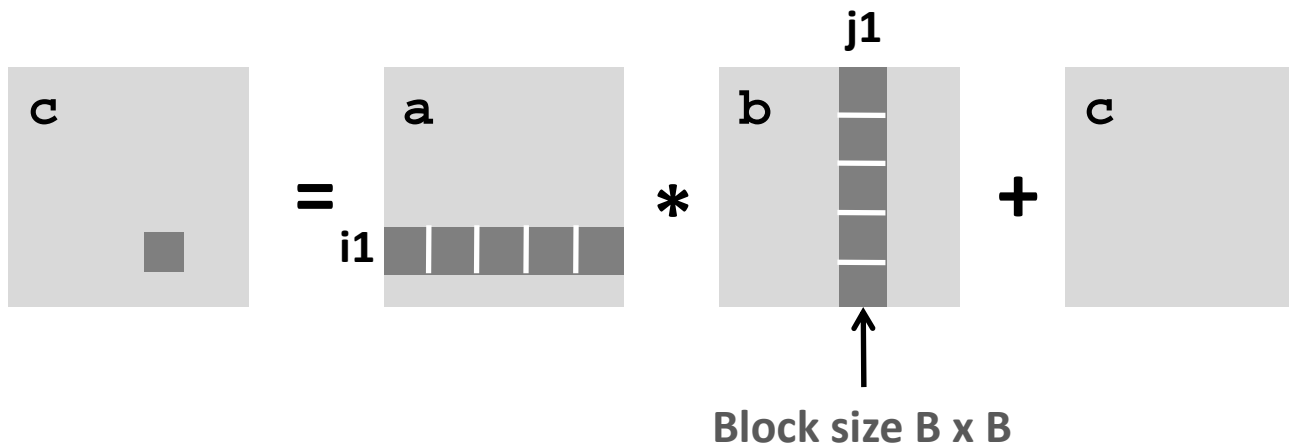
Blocked Matrix Multiplication

```

c = (double *) calloc(sizeof(double), n*n);


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

```



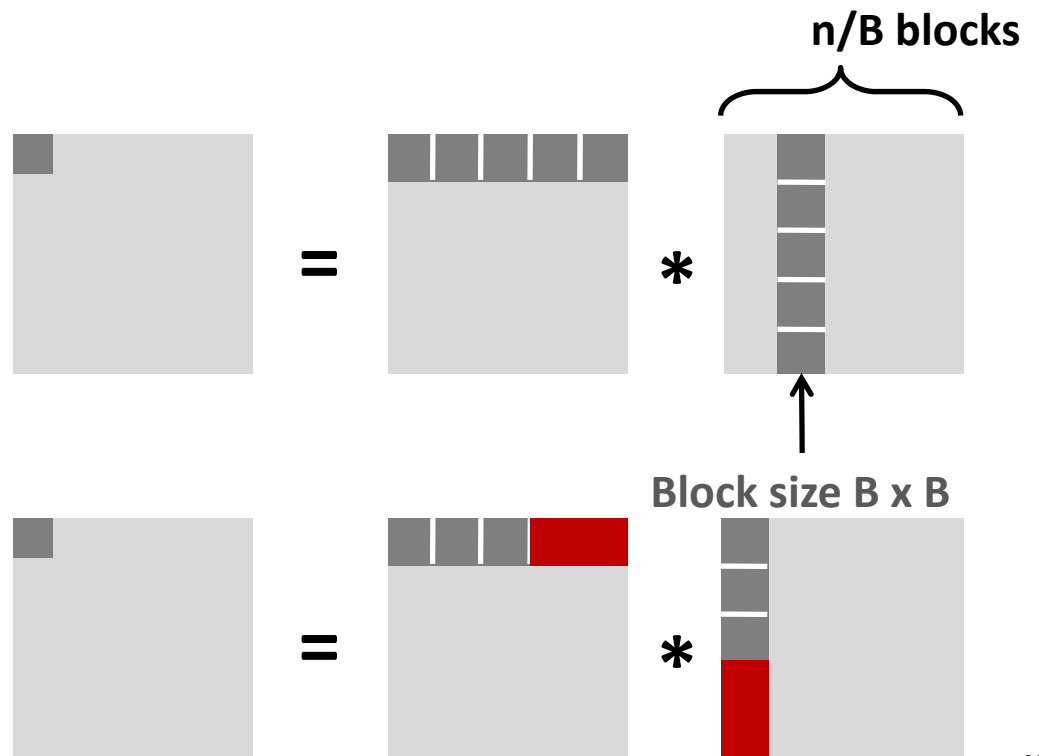
Cache Miss Analysis

■ Assume:

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


■ First (block) iteration:

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



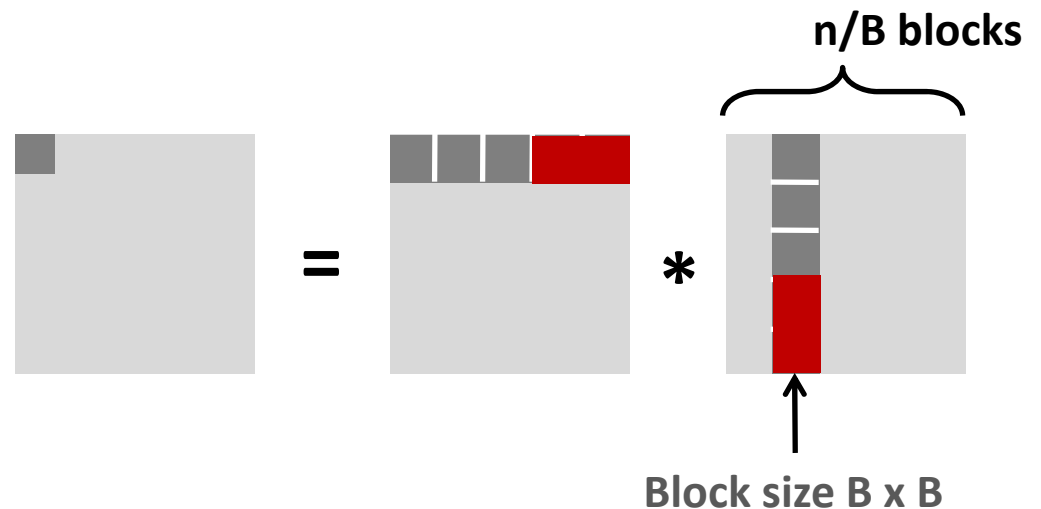
Cache Miss Analysis

■ Assume:

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

■ Second (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 elements used $O(n)$ times!
 - But program has to be written properly

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)