CS 201 System Programming and Architecture

Coding Cache Friendly

Fang Song
Portland State University

Slides adapted from
CS205@PSU(Prof. Li) / CS15-213 @CMU

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).

Recall: Layout of C 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 block size (B) > sizeof(a[i][j]) bytes, exploit spatial locality.
  - Compulsory miss rate = sizeof(a[i][j]) / 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.

Cache Friendly Code Example

Cold cache, 4-byte words, 4-word cache blocks

```
int sumarrayrows(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;
}
```

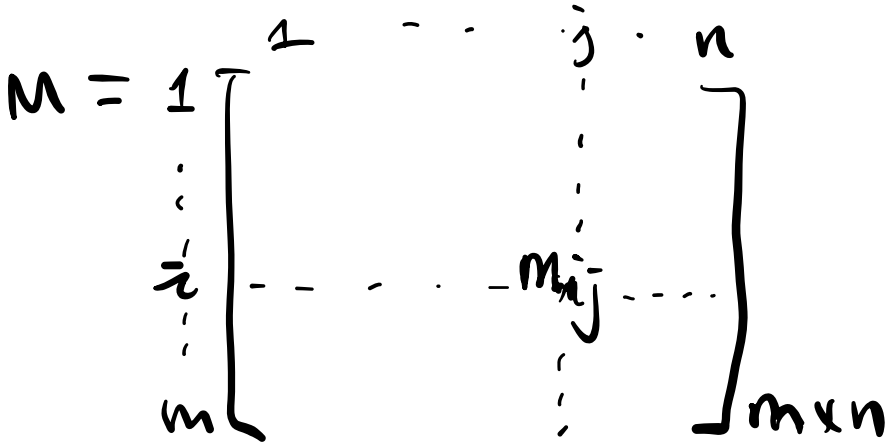
Miss rate = 1/4

```
int sumarraycols(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;
}
```

Miss rate = 1

Matrix Multiplication



- $m = n$: square matrix.
- $\mathbb{1}_n = \begin{bmatrix} 1 & & 0 \\ & \ddots & \\ 0 & & 1 \end{bmatrix}$ (identity)

- m: rows
- n: columns.
- m_{ij}: a number $\in \mathbb{R}$
- row vector: $(m=1)$ $(v_1, \dots, v_n)$ n-dim rowvec.
- column vector: $(n=1)$ $\begin{pmatrix} u_1 \\ \vdots \\ u_m \end{pmatrix}$ m-dim colvec

Matrix Multiplication

Matrix op's:

- addition

$A, B := m \times n$

$C := A + B : m \times n$

$c_{ij} = a_{ij} + b_{ij}$ (entry-wise addition)

$$\begin{matrix} A & & B & & C \\ \begin{bmatrix} 6 & 0 & 3 \\ 2 & 0 & 1 \end{bmatrix}_{2\times 3} & + & \begin{bmatrix} 1 & 6 & 3 \\ -2 & 11 & 9 \end{bmatrix}_{2\times 3} & = & \begin{bmatrix} 7 & 6 & 6 \\ 0 & 11 & 10 \end{bmatrix}_{2\times 3} \end{matrix}$$

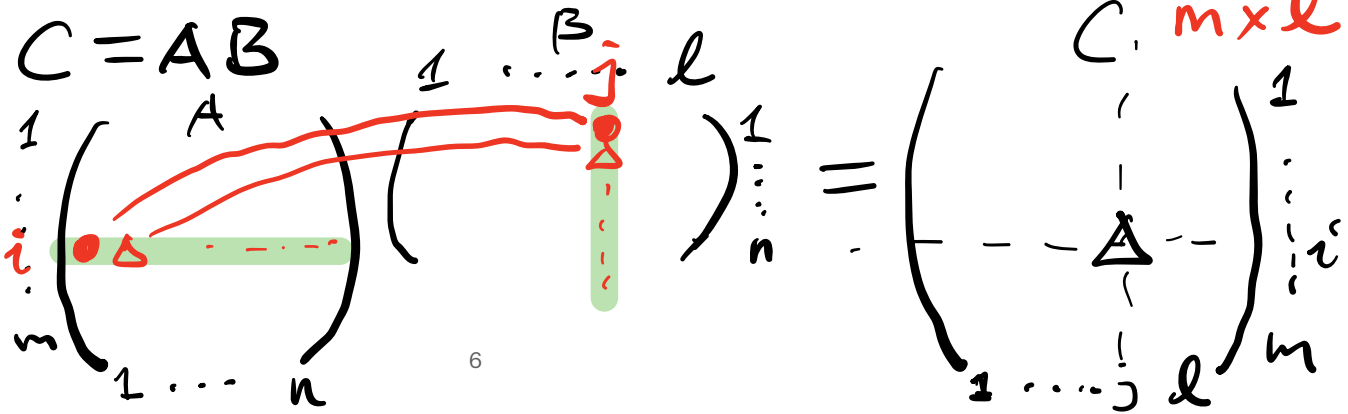
$c_{ij} := a_{i1} \cdot b_{1j} + a_{i2} \cdot b_{2j} + a_{i3} \cdot b_{3j} + \cdots + a_{in} b_{nj}$

- Mult.

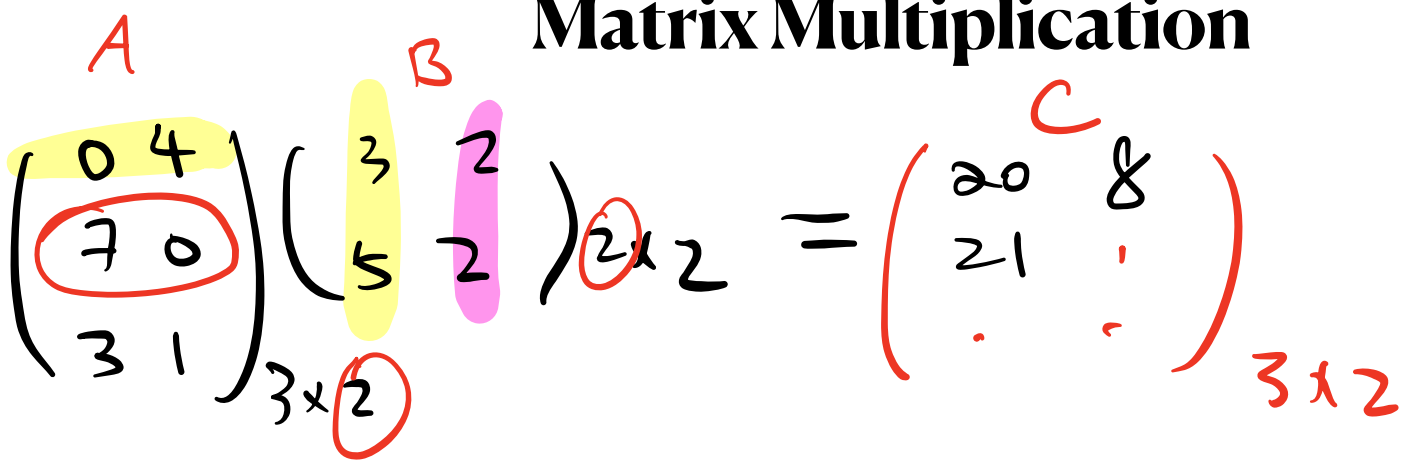
$A : m \times n$

$B : n \times \ell$

$C = AB$



Matrix Multiplication



$c_{11} = 0 \cdot 3 + 4 \cdot 5 = 20$

$c_{12} = 0 \cdot 2 + 4 \cdot 2 = 8$

$c_{21} = 7 \cdot 3 + 0 \cdot 5 = 21$

All square dim: n

Time: (matrix mult.)

$A_{m \times n} \cdot B_{n \times \ell} = C_{m \times \ell}$

In C: $m \times \ell$ entries

every entry n add/mult.

$O(m \times \ell \times n) \rightarrow O(n^3)$

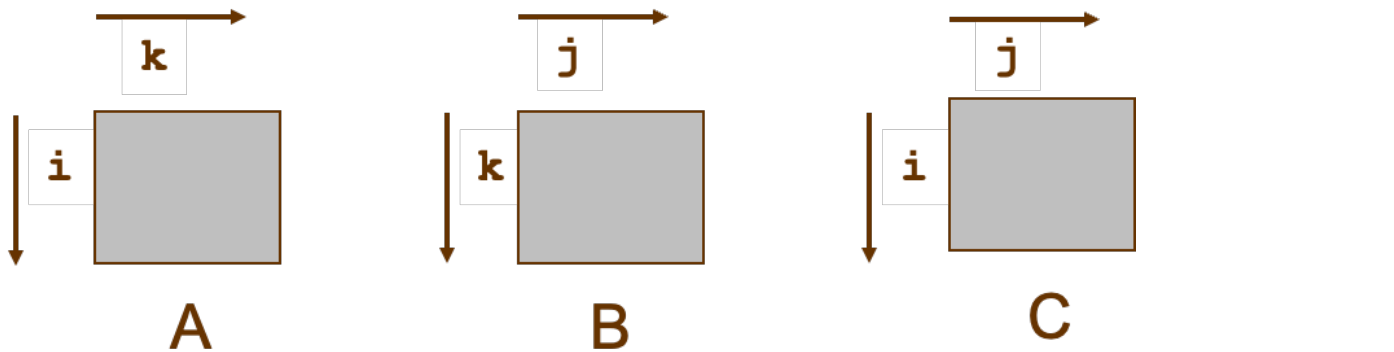
Matrix Multiplication Example

- **Multiply N x 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;
  }
}
```

Miss Rate Analysis for Matrix Mult.

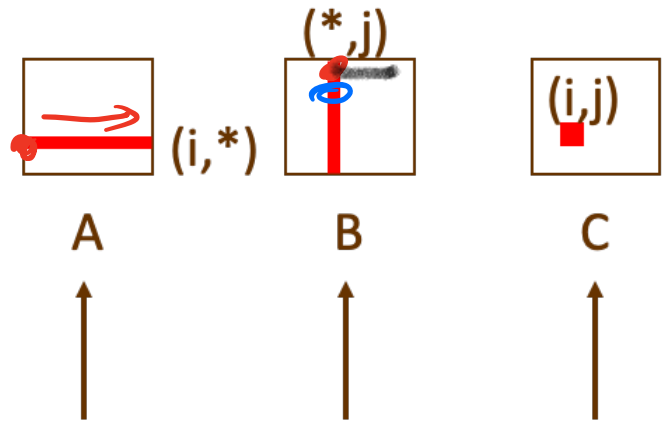
- **Assume:**
  - Line size = 32B (big enough for four 64-bit words).
  - Matrix dimension (N) is very large. (Approximate 1/N as 0.0)
  - Cache is not even big enough to hold multiple rows.
- **Analysis method:** Look at access pattern of inner loop.



Matrix Multiplication (i j k)

```
/* 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:

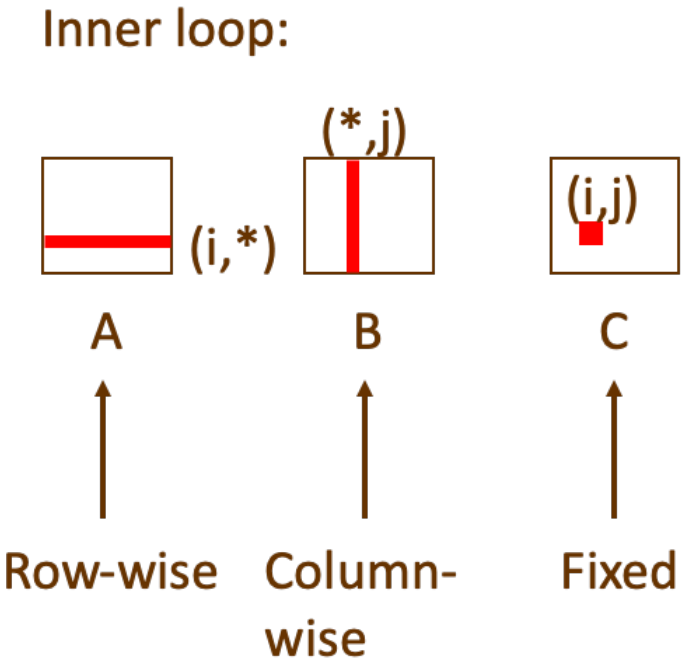
A	B	C
1/4	1	

Matrix Multiplication (i j k)

```
/* 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;
  }
}
```

Misses per inner loop iteration:

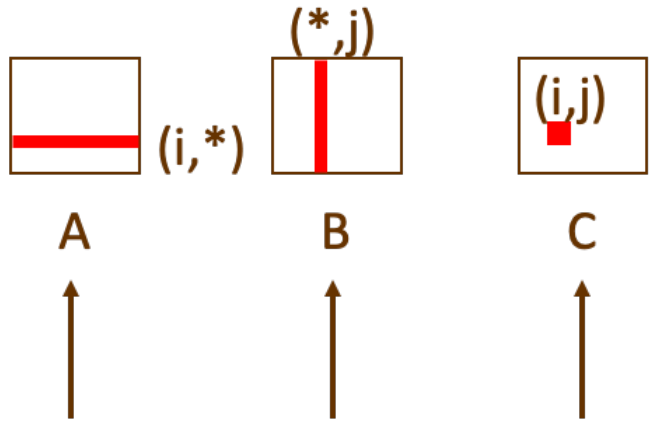
A	B	C
0.25	1.0	0.0



Matrix Multiplication (j i k)

```
/* 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:



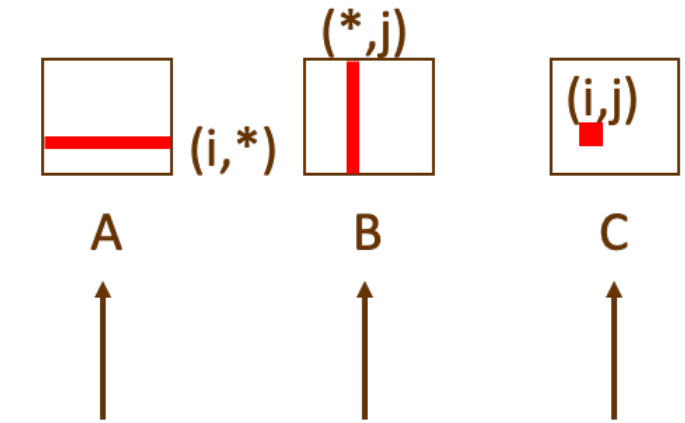
Misses per inner loop iteration:

A	B	C
1/4	1	0

Matrix Multiplication (j i k)

```
/* 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:

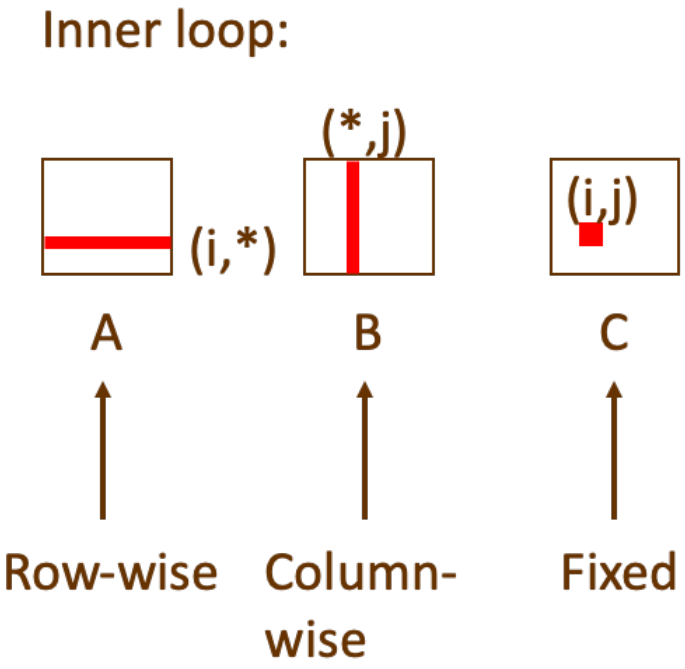
A	B	C

Matrix Multiplication (j i k)

```
/* 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
  }
}
```

Misses per inner loop iteration:

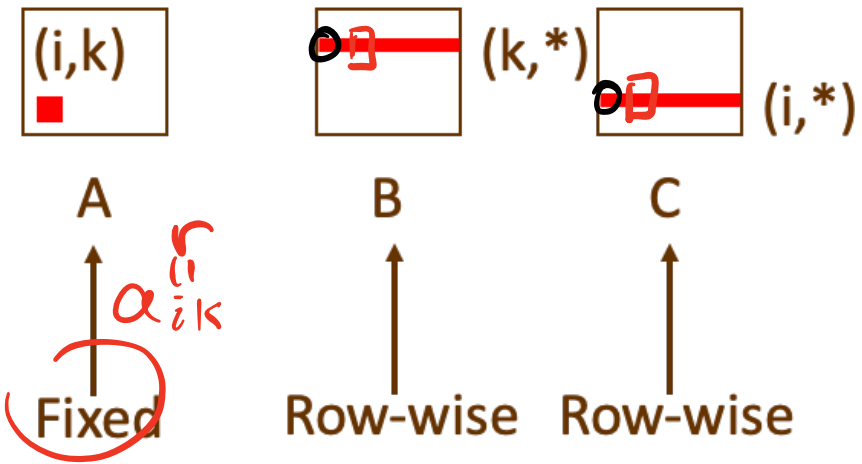
A	B	C
0.25	1.0	0.0



Matrix Multiplication (k i j)

```
/* 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:



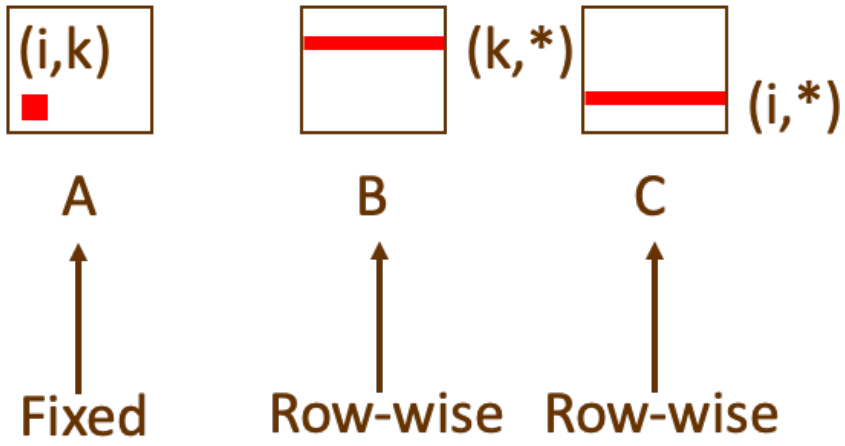
Misses per inner loop iteration:

A	B	C
0	1/4	1/4

Matrix Multiplication (k i j)

```
/* 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

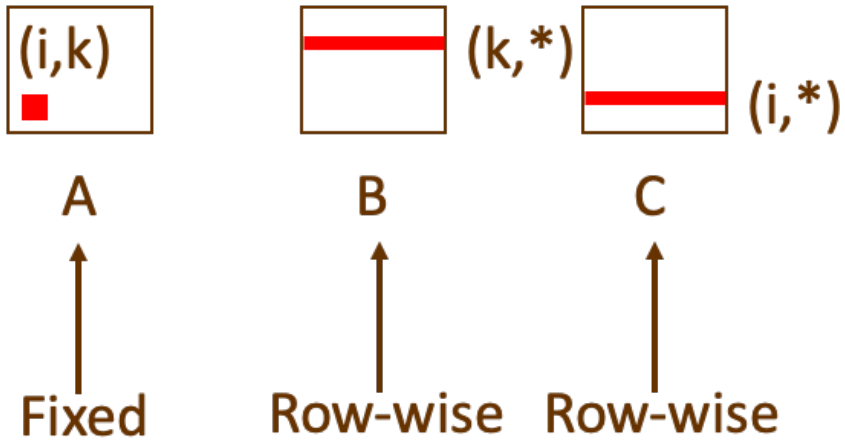
Matrix Multiplication (i k j)

```
/* 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:

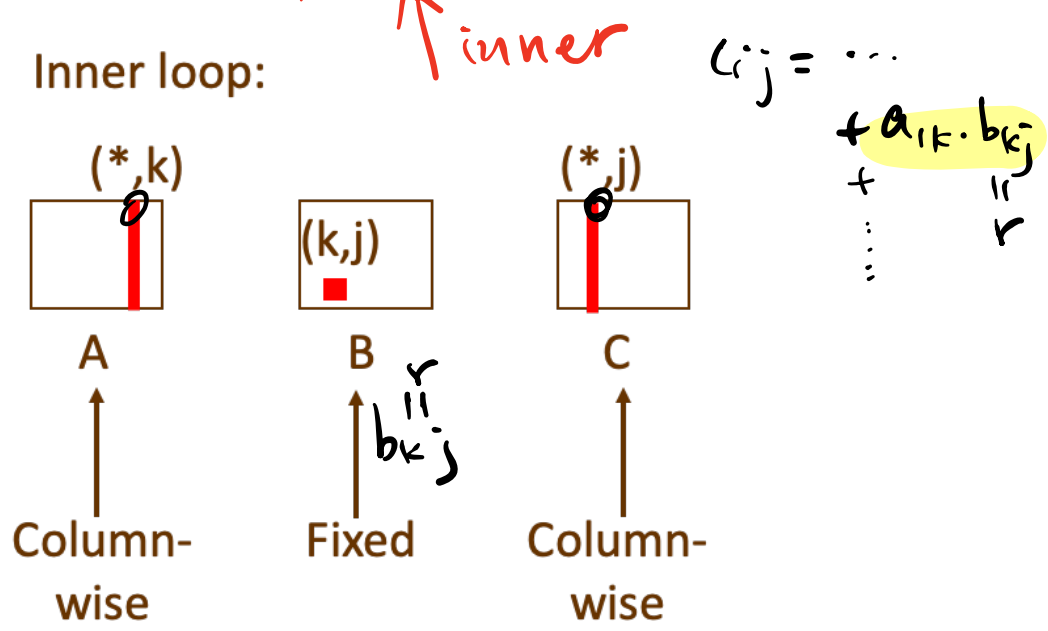
A	B	C
0.0	0.25	0.25

Inner loop:



Matrix Multiplication (j k i)

```
/* 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;
  }
}
```



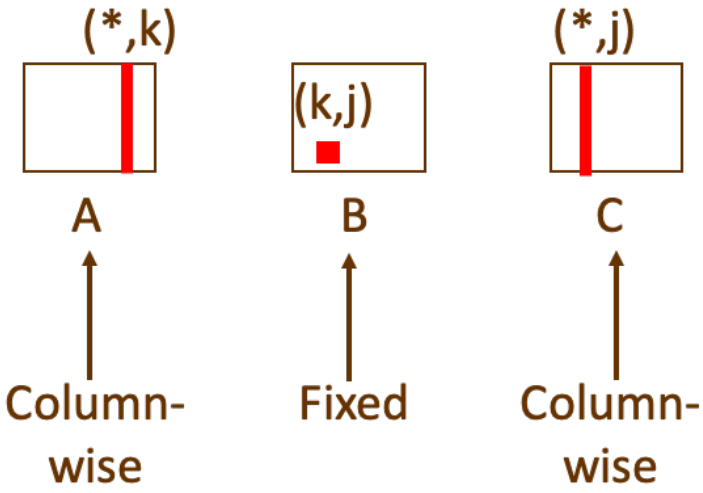
Misses per inner loop iteration:

A	B	C
1	0	1

Matrix Multiplication (j k i)

```
/* 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:



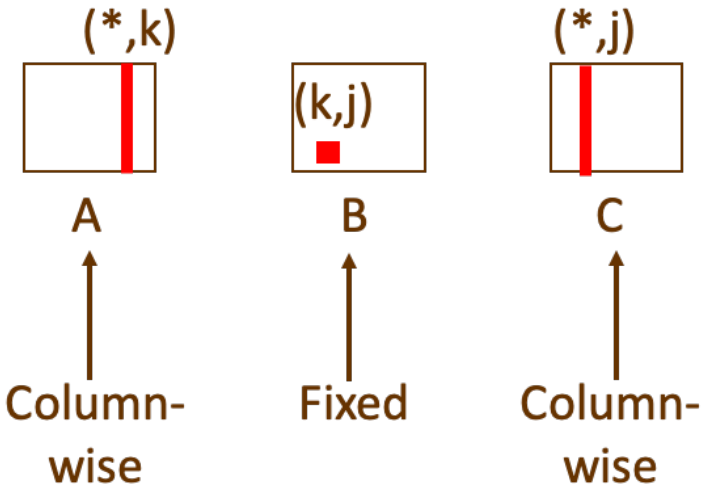
Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Matrix Multiplication (k j i)

```
/* 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 = 1.25

```
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):

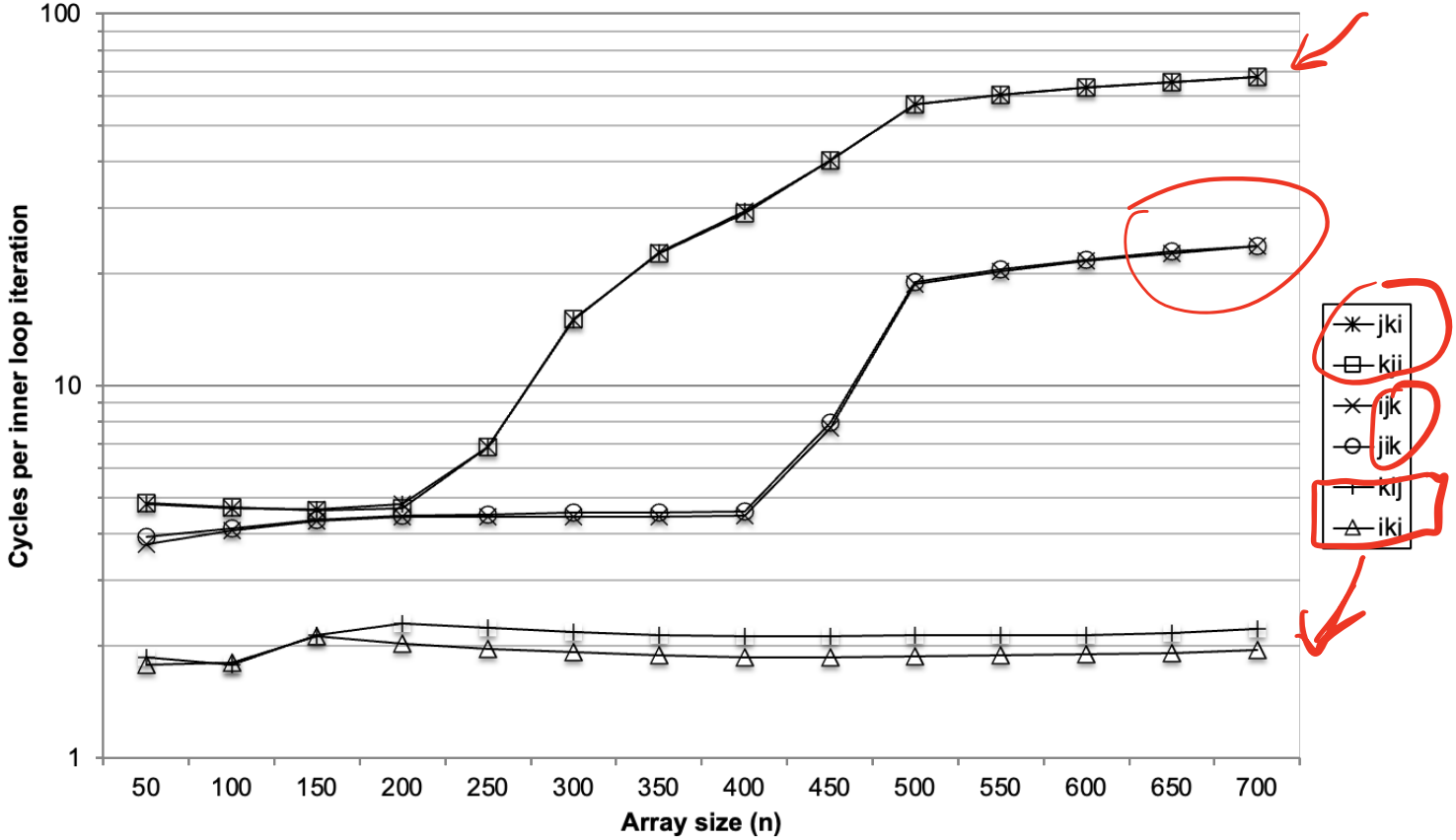
- 2 loads, 1 stores
- misses/iter = 0.5

```
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 stores
- misses/iter = 2

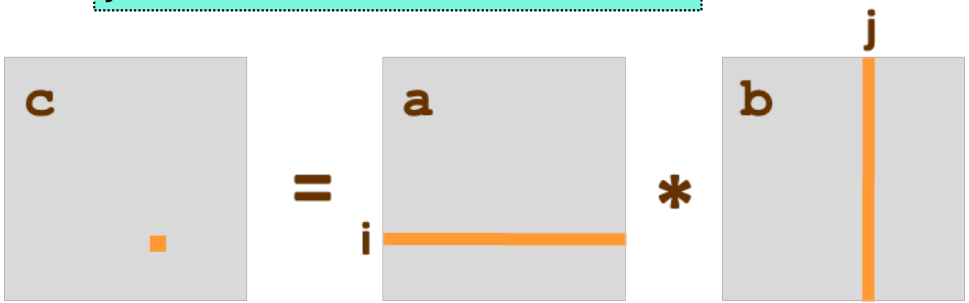
Summary of Matrix Multiplication



Improving Temporal Locality

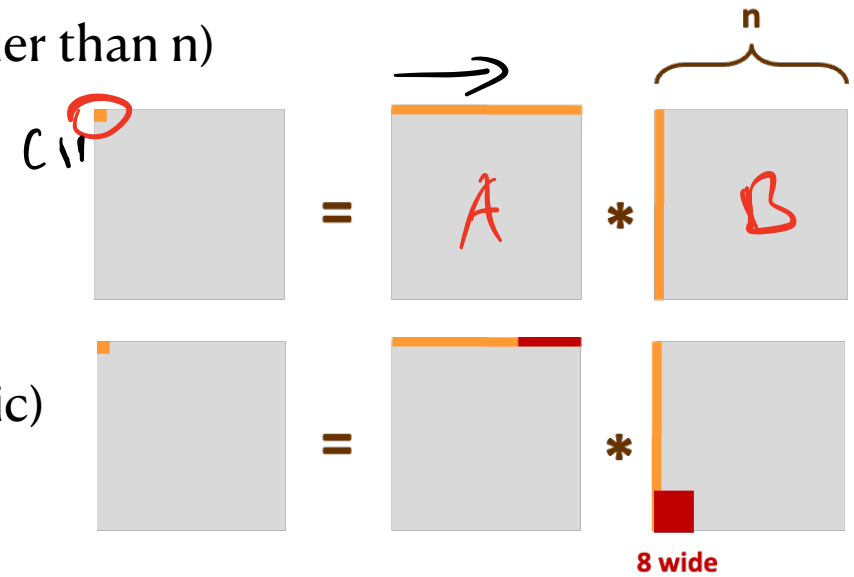
- Example: Matrix multiplication (again)

```
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;
  }
}
```



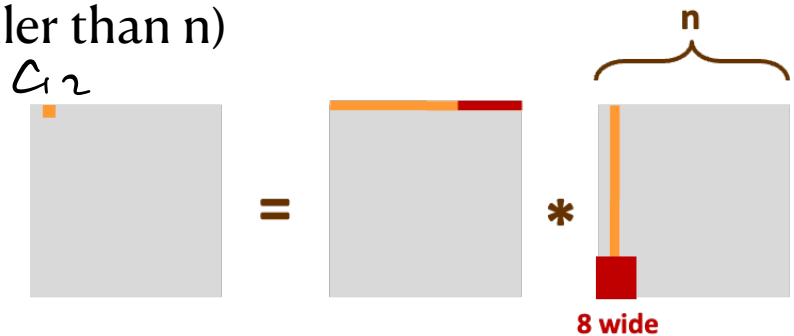
Cache Miss Analysis

- **Assume:**
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
- **1st 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)
- **2nd iteration**
  - Again n/8 + n = 9n/8 misses



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

Matrix Blocking

- Organize matrix-based computation through sub-blocks.

- Example: Blocked matrix multiplication (N = 8; sub-block size = 4)

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

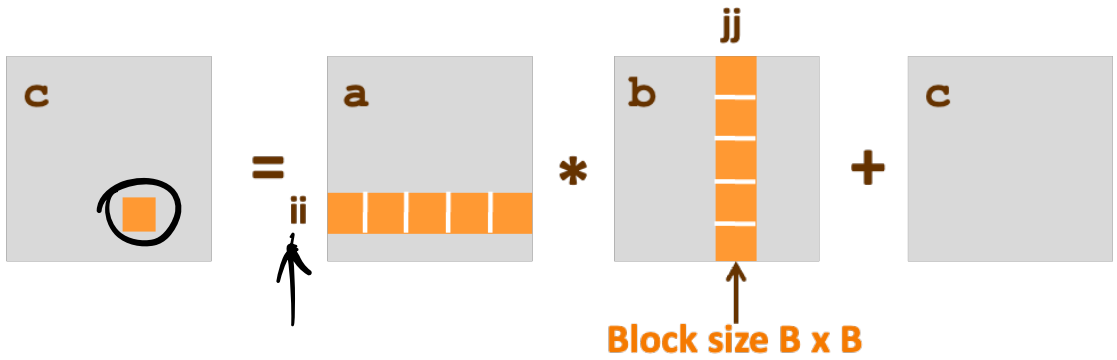
$$C_{11} = A_{11}B_{11} + A_{12}B_{21} \qquad C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

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

- Key idea: Sub-blocks can be treated just like scalars.

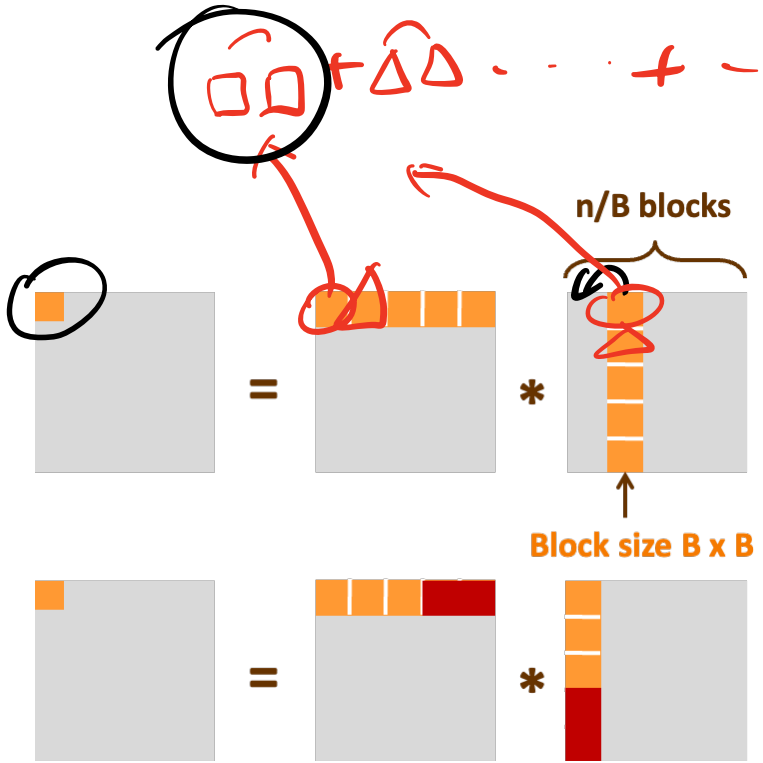
Blocked Matrix Multiplication

```
/* Assume c is initialized to all 0s. */
for (ii = 0; ii < n; ii+=B)
  for (jj = 0; jj < n; jj+=B)
    for (kk = 0; kk < n; kk+=B)
      /* B x B mini matrix multiplications */
      for (i = ii; i < ii+B; i++)
        for (j = jj; j < jj+B; j++)
          for (k = kk; k < kk+B; k++)
            c[i][j] += a[i][k] * b[k][j];
```



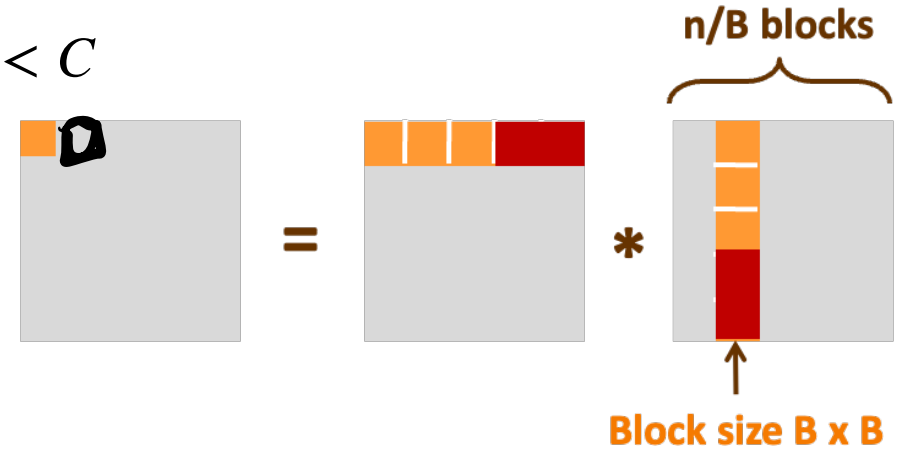
Cache Miss Analysis

- **Assume:**
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks ■ fit into cache: $3B^2 < C$
- **1st (block) iteration:**
  - $B^2/8$ misses for each block
  - $2n/B * B^2/8 = nB/4$ (omitting C)
  - Afterwards in cache (schematic)



Cache Miss Analysis

- **Assume:**
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks ■ fit into cache: $3B^2 < C$
- **2nd (block) iteration:**
  - Same as first iteration
  - $2n/B * B^2/8 = nB/4$



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

Blocking Summary

- **No blocking:** $(9/8) * n^3$
- **With blocking:** $(1/4B) * n^3$

⇒**Suggest largest possible block size B, with 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: Blocking is a general technique

- **All systems favor "cache friendly code"**
  - Getting absolute optimum performance is platform specific.
  - Can get most of the advantage with generic code.
    - Keep working set reasonably small (temporal locality)
    - Use small strides (spatial locality).

PORTLAND
TRAIL BLAZERS

Best of luck!