CSE251: System Programming

14. Cache (3)

Seongil Wi

Department of Computer Science and Engineering

Important Principle: Locality (지역성)

The tendency to access the same set of memory locations *repetitively* over a short period of time

1. **Temporal locality** (locality in time)
   - If an item is referenced, *the same item* will tend to be referenced again soon

2. **Spatial locality** (locality in space)
   - If an item is referenced, *nearby items* will tend to be referenced soon

Motivation – Semantically Equivalent Code

Would it be the same in terms of performance?

```
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;
}
```

```
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;
}
```

(Demo) Let's check about it!

Motivation – Semantically Equivalent Code

Would it be the same in terms of performance?

```
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;
}
```

```
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;
}
```

★ Get a qualitative sense of its locality
is a key skill for a professional programmer!

Writing Cache-Friendly Code

Qualitative Estimates of Locality

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

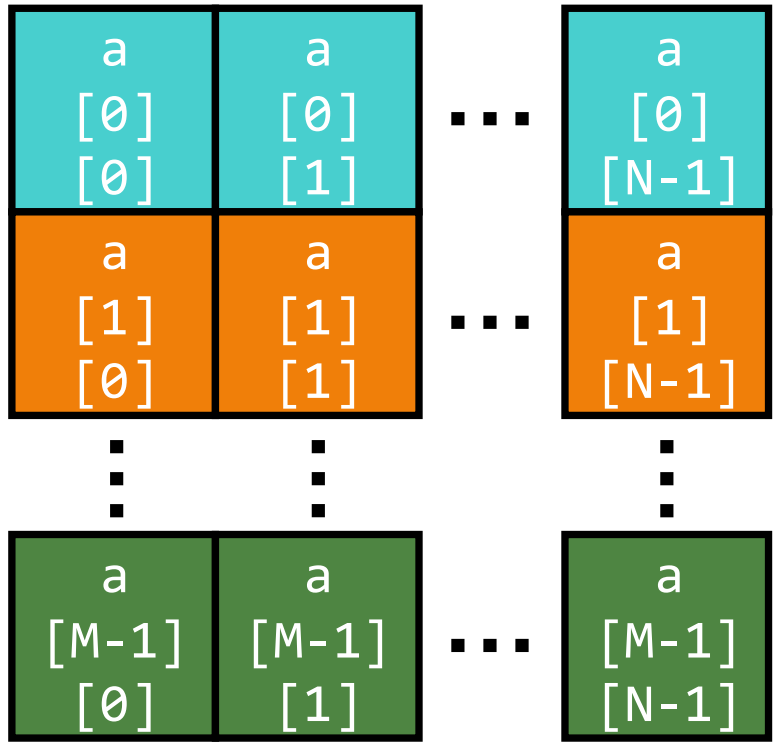
```
int sum_array_rows(int a[M][N]) {
    int i, j, sum = 0;

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

Qualitative Estimates of Locality

- 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;
}
```

Hint: Look at the memory layout of variable a!

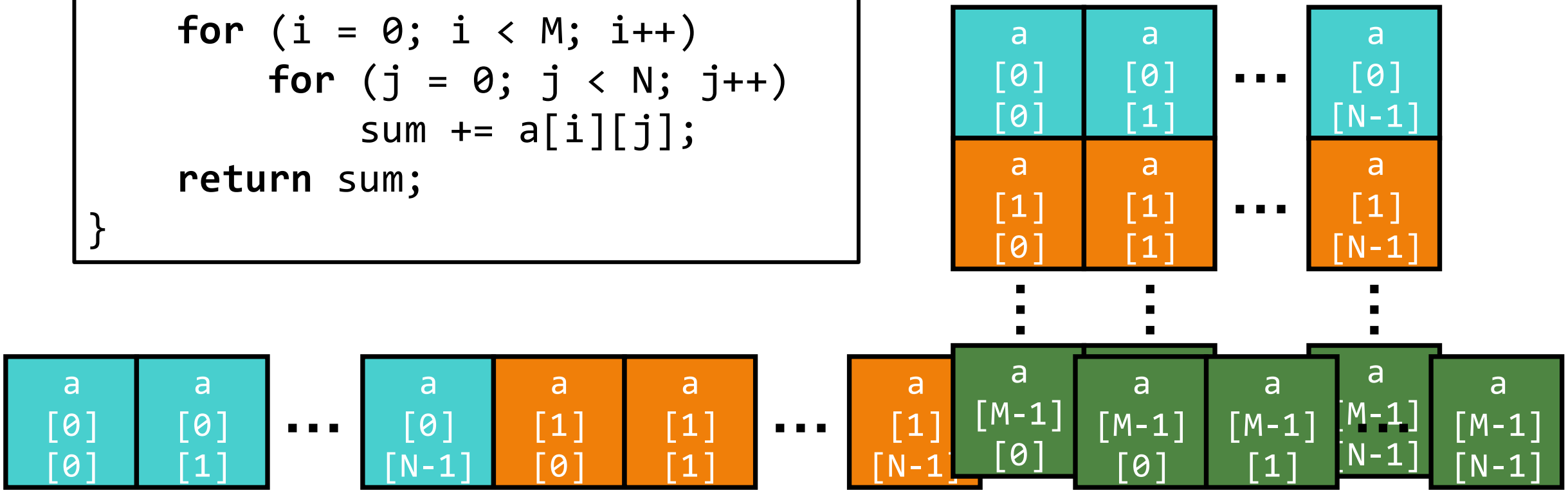


Qualitative Estimates of Locality

- 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;
}
```

Hint: Look at the memory layout of variable a!



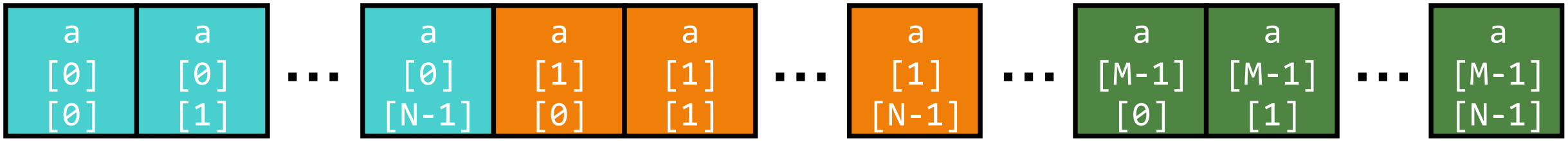
Qualitative Estimates of Locality

- 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;
}
```

Hint: Look at the memory layout of variable a!

Array layout is row-major order



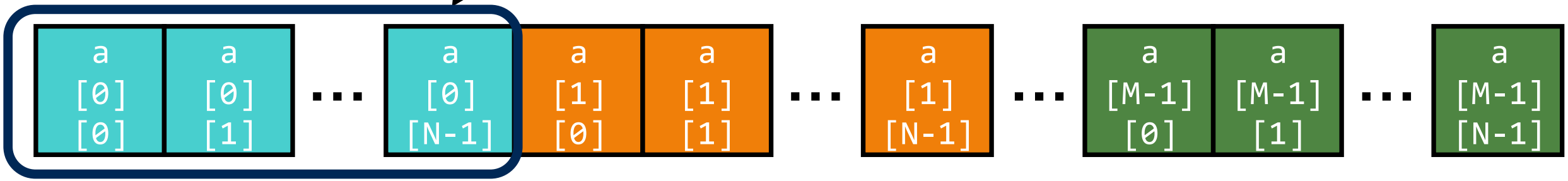
Qualitative Estimates of Locality

- 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;
}
```

Hint: Look at the memory layout of variable a!

Locality!



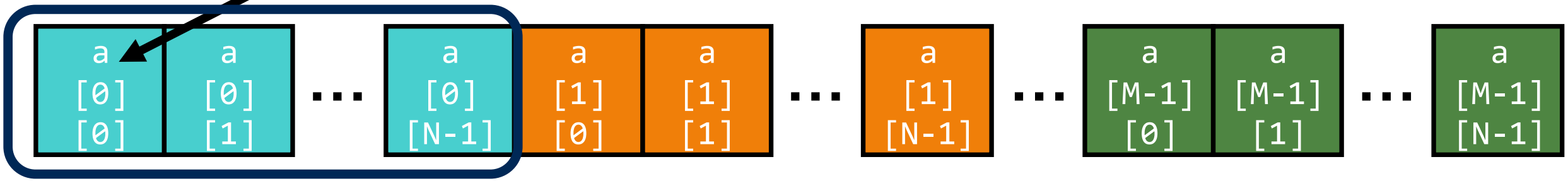
Qualitative Estimates of Locality

- 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;
}
```

Hint: Look at the memory layout of variable a!

Miss

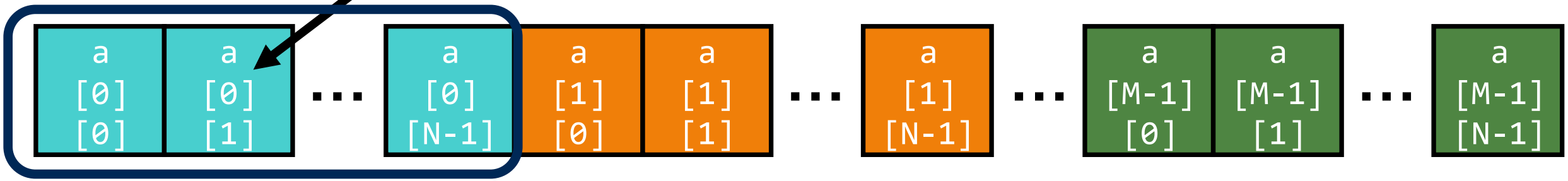


Qualitative Estimates of Locality

- 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;
}
```

Hint: Look at the memory layout of variable a!

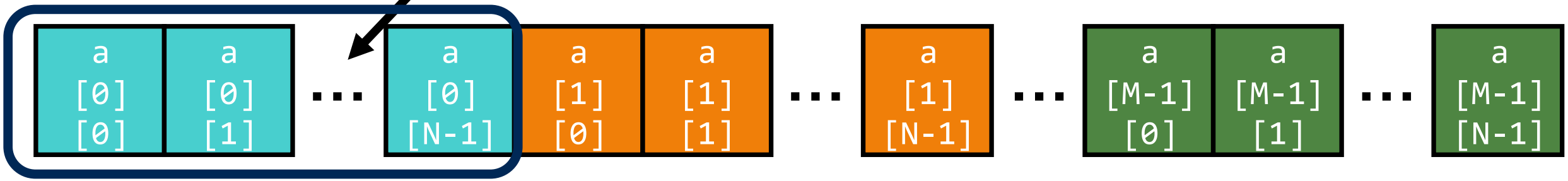


Qualitative Estimates of Locality

- 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;
}
```

Hint: Look at the memory layout of variable a!

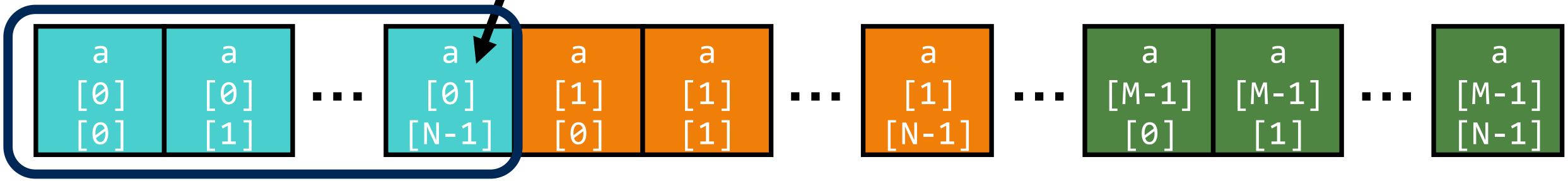


Qualitative Estimates of Locality

- 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;
}
```

Hint: Look at the memory layout of variable a!



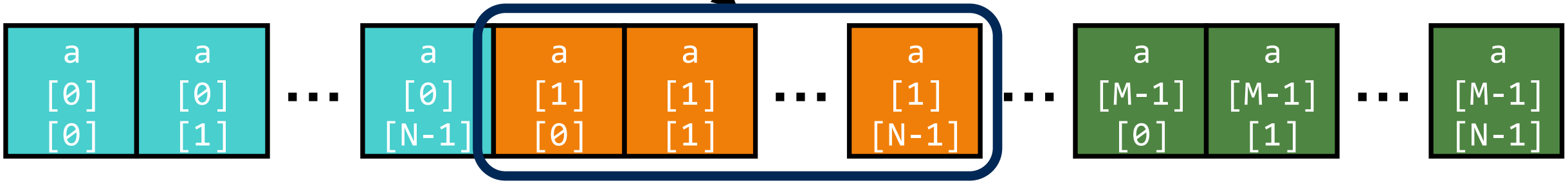
Qualitative Estimates of Locality

- 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;
}
```

Hint: Look at the memory layout of variable a!

Locality!



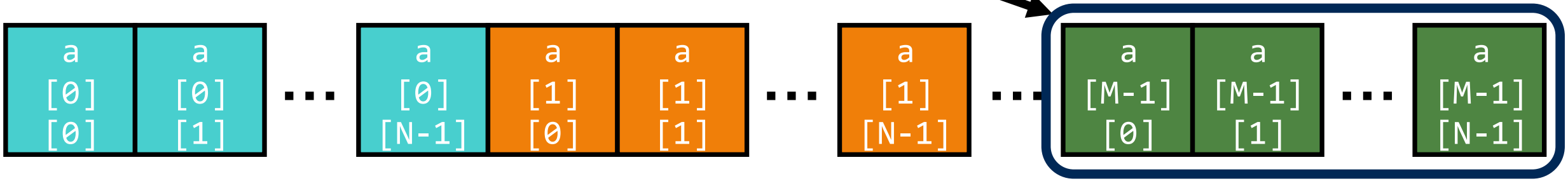
Qualitative Estimates of Locality

- 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;
}
```

Hint: Look at the memory layout of variable a!

Locality!



Qualitative Estimates of Locality

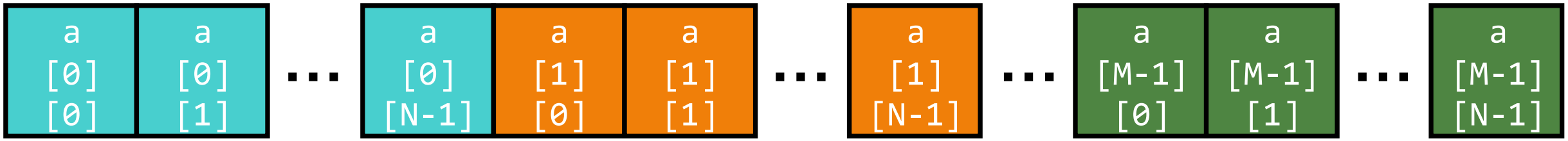
- Does this function have good locality with respect to array a?

```
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;
}
```

Hint: Look at the memory layout of variable a!

Answer: Yes ☺

Stride-1 reference pattern

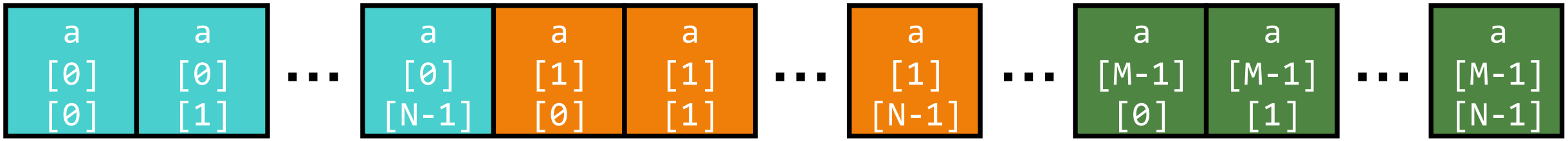


How About This One?

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

```
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;
}
```

Hint: Look at the memory layout of variable a!

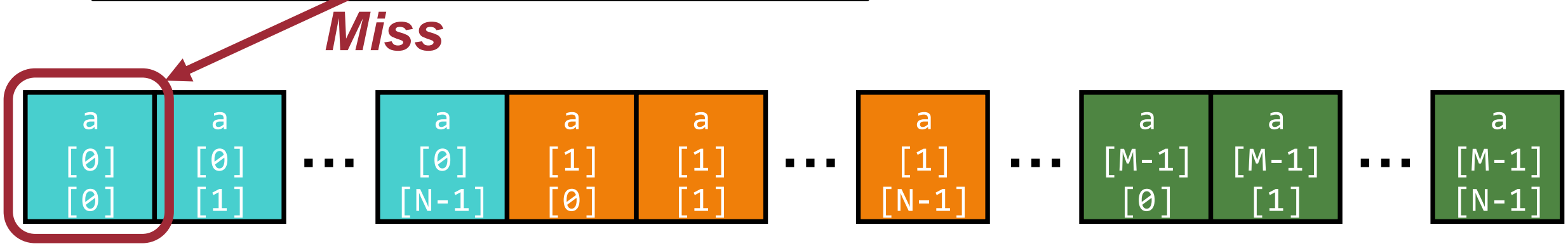


How About This One?

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

```
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;
}
```

Hint: Look at the memory layout of variable a!

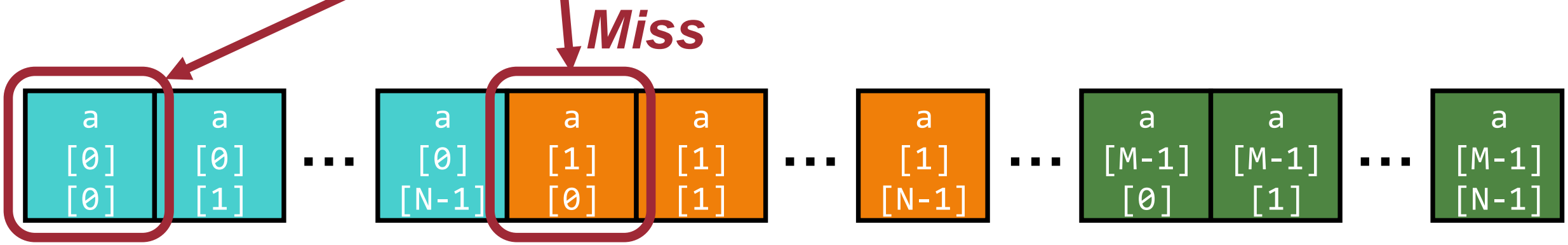


How About This One?

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

```
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;
}
```

Hint: Look at the memory layout of variable a!



How About This One?

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

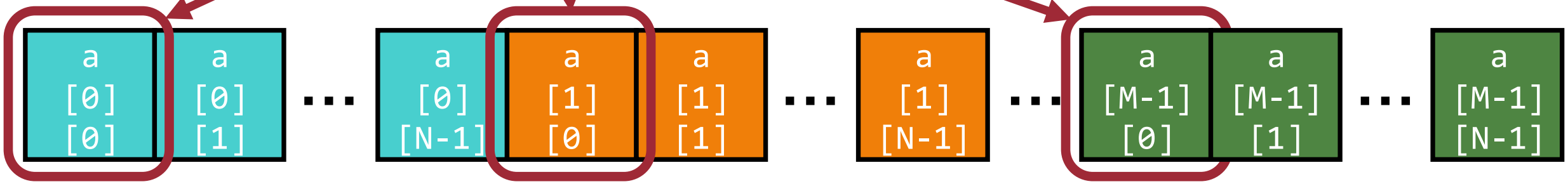
```
int sum_array_cols(int a[M][N]) {
    int i, j, sum = 0;

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

Hint: Look at the memory layout of variable a!

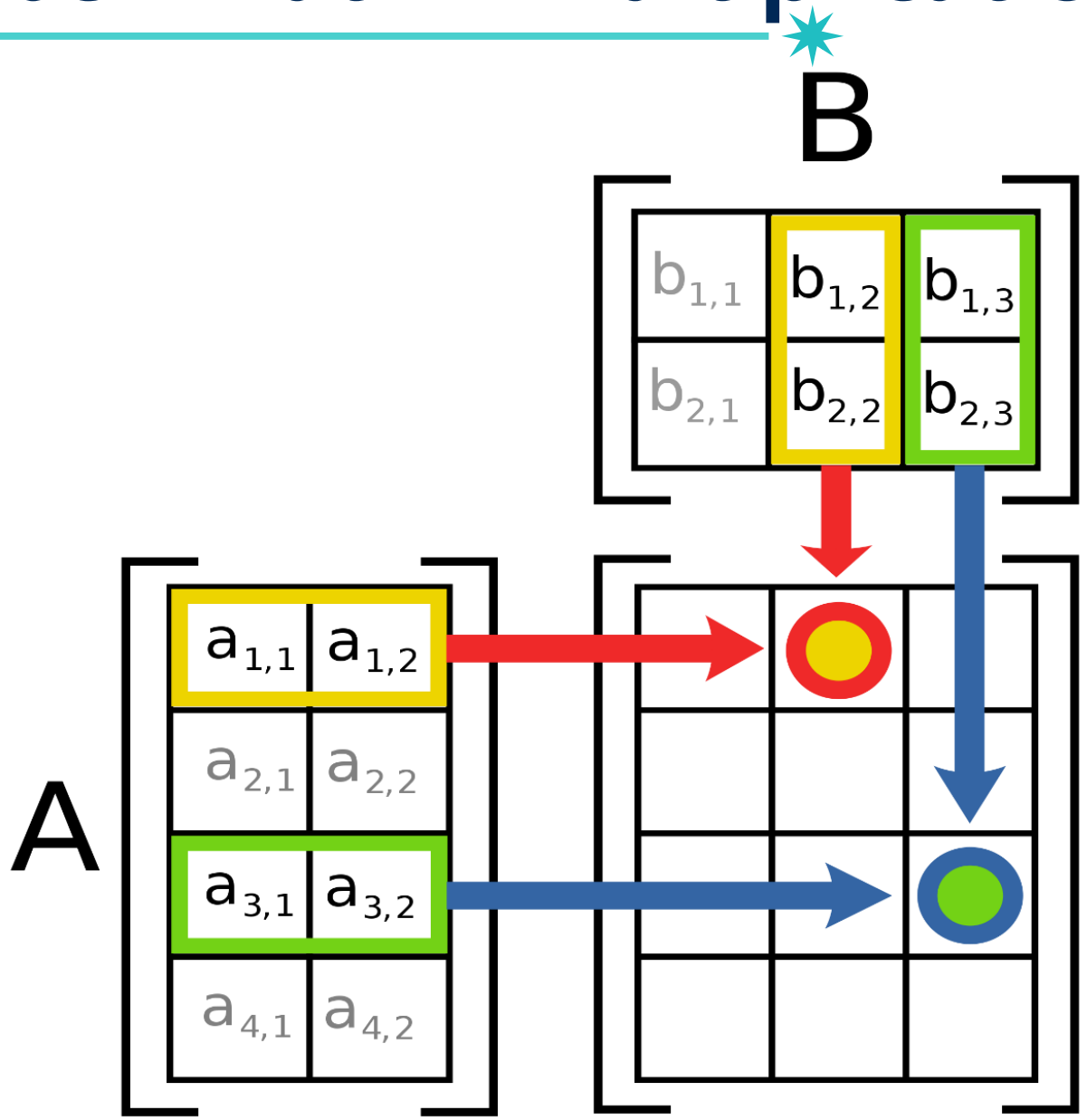
Answer: No ☹

Stride-N reference pattern

Miss



Remember Matrix Multiplication



Matrix Multiplication Example

Multiplying a pair of *n* x *n* matrices: *C* = *AB* (*n*=2)

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \times \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

```
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;
  }
}
```

Matrix Multiplication Example

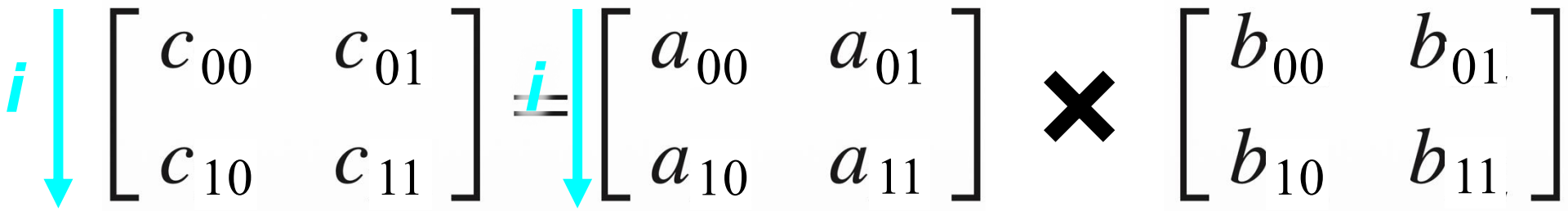
Multiplying a pair of *n* x *n* matrices: *C* = *AB* (*n*=2)

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \times \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$c_{00} = a_{00}b_{00} + a_{01}b_{10}$$

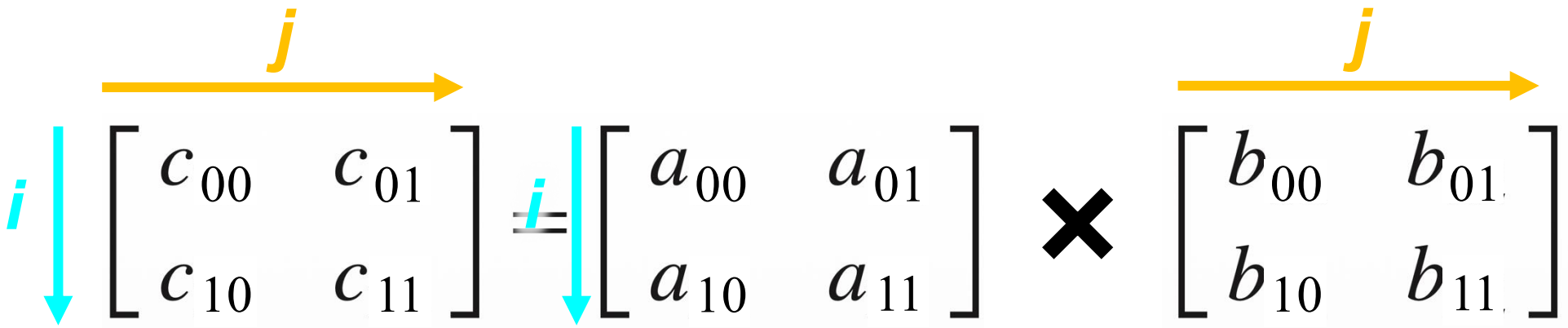
```
for (i = 0; i < n; i++)  {
    for (j = 0; j < n; j++) {
        sum = 0.0;
        for (k = 0; k < n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
  }
}
```

Three Indexes: *i, j, k*



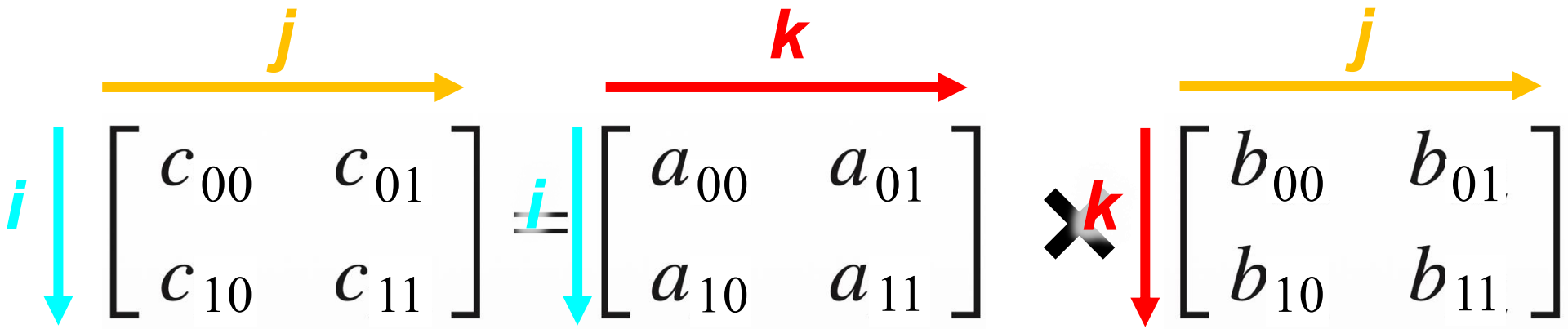

```
for (i = 0; i < n; i++)  {
    for (j = 0; j < n; j++) {
        sum = 0.0;
        for (k = 0; k < n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
  }
}
```

Three Indexes: *i, j, k*



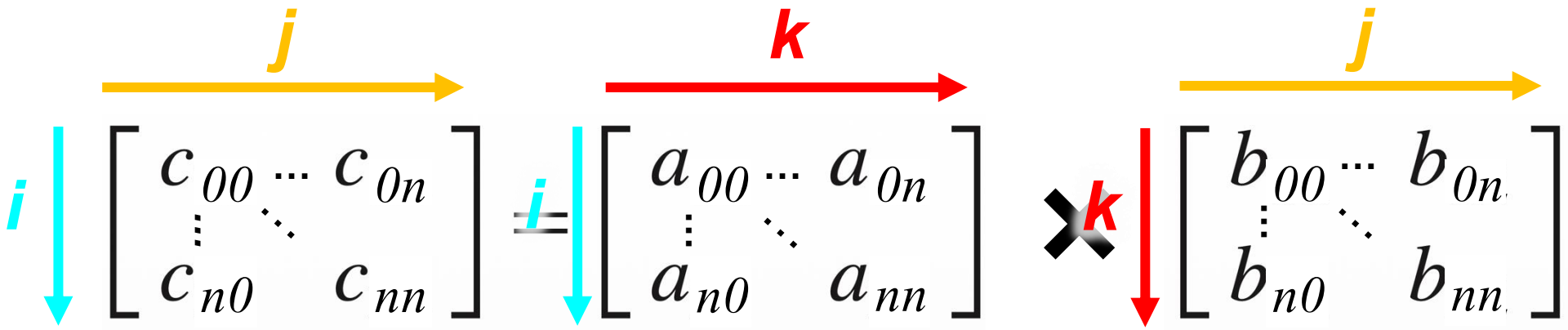

```
for (i = 0; i < n; i++)  {
    for (j = 0; j < n; j++) {
        sum = 0.0;
        for (k = 0; k < n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
  }
}
```

Three Indexes: *i, j, k*




```
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 Multiply



Assumption:

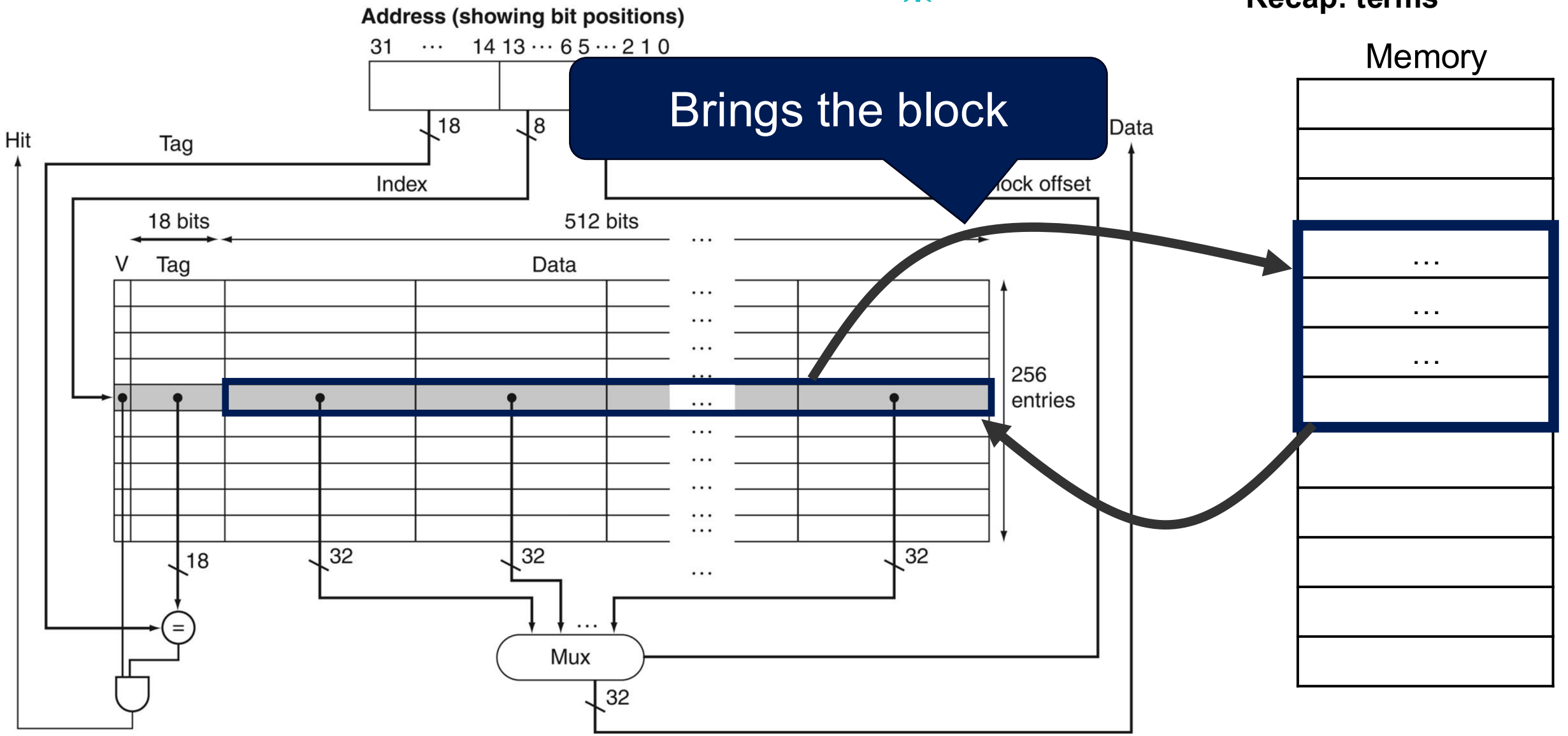
- Multiply *n* x *n* matrices
  - Matrix dimension (*n*) is very large
- Matrix elements are doubles (8 bytes)
- Block size = 32 bytes (4 doubles)

```
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;
  }
}
```

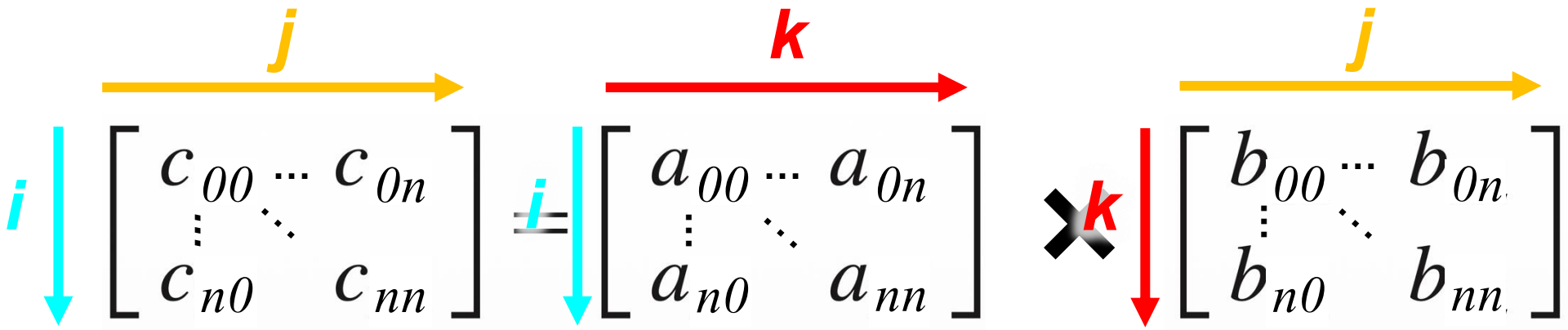
Recap: Block

- Block (a.k.a., line): unit of copying
  - Several words in cache memory

Recap: terms



Miss Rate Analysis for Matrix Multiply

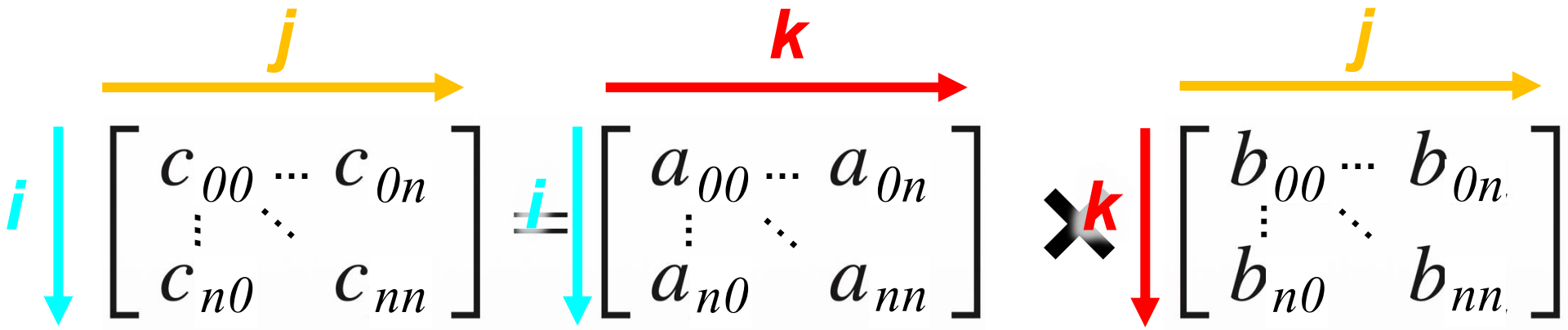


Assumption:

- Multiply *n* x *n* matrices
  - Matrix dimension (*n*) is very large
- Matrix elements are doubles (8 bytes)
- Block size = 32 bytes (4 doubles)
- Cache is not even big enough to hold multiple rows

```
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;
  }
}
```

Three Variants: *ijk, jki, kij*



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;
  }
}
```

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;
  }
}
```

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];
  }
}
```

Three Variants: *ijk, jki, kij*

They are semantically equivalent.
In terms of the performance, which one is better?

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;
  }
}
```

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;
  }
}
```

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];
  }
}
```

Three Variants: *ijk, jki, kij*

They are semantically equivalent.
In terms of the performance, which one is better?

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;
  }
}
```

jki

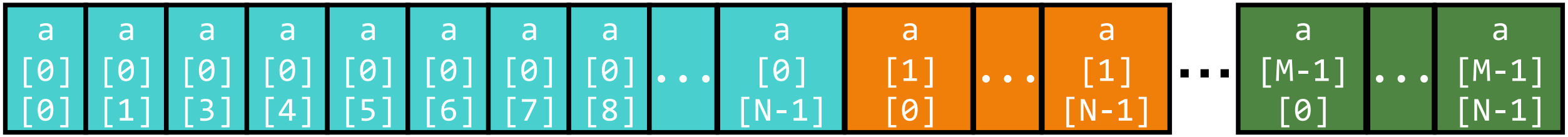
```
for (j = 0; j < n; j++) {
  for (k = 0; k < n; k++) {
    r = b[k][j];
    for (i = 0; i < n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

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];
  }
}
```

We need to analyze the miss rate for inner loop iterations!

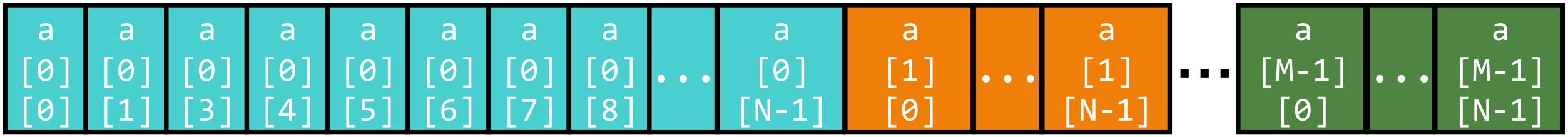
Layout of C Array in Memory (Review)



- **Stepping through columns in one row:**

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

Layout of C Array in Memory



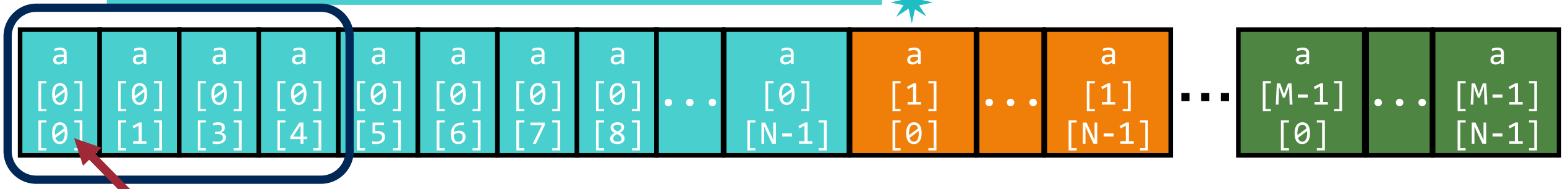
- **Stepping through columns in one row:**

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

- **Hit**: data requested is in the upper level
  - Hit ratio: hits/accesses
- **Miss**: data requested is not in the upper level
  - Block copied from lower level
  - Miss penalty: time taken to resolve miss
  - Miss ratio: misses/accesses
    = 1 – hit ratio

Terms

Layout of C Array in Memory



1 miss

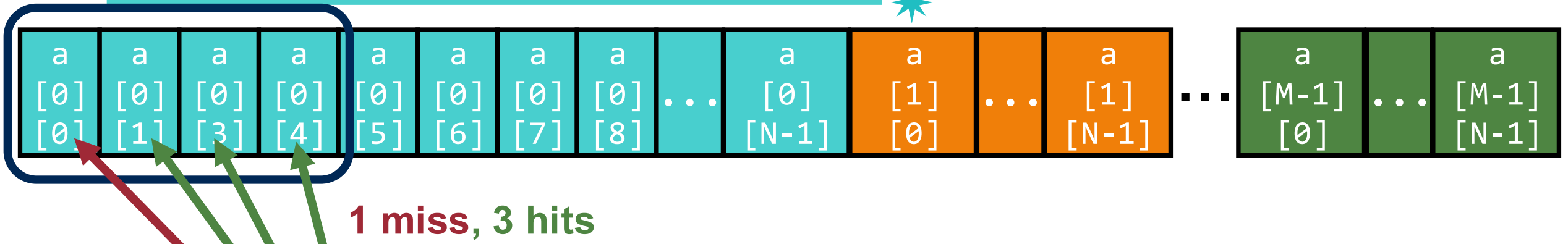
- **Stepping through columns in one row:**

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

- **Hit**: data requested is in the upper level
  - Hit ratio: hits/accesses
- **Miss**: data requested is not in the upper level
  - Block copied from lower level
  - Miss penalty: time taken to resolve miss
  - Miss ratio: misses/accesses
    = 1 – hit ratio

Terms

Layout of C Array in Memory



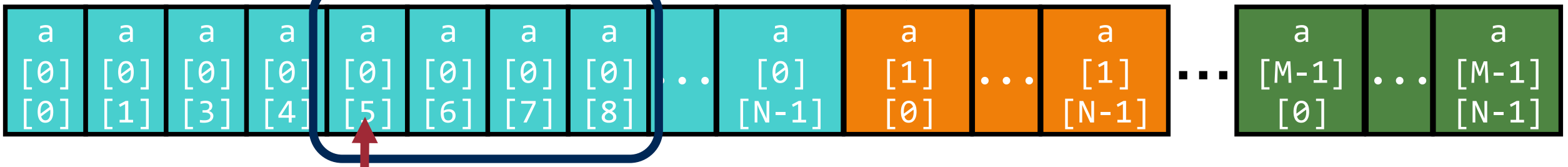
- **Stepping through columns in one row:**

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

- **Hit**: data requested is in the upper level
  - Hit ratio: hits/accesses
- **Miss**: data requested is not in the upper level
  - Block copied from lower level
  - Miss penalty: time taken to resolve miss
  - Miss ratio: misses/accesses
    = 1 – hit ratio

Terms

Layout of C Array in Memory



1 miss

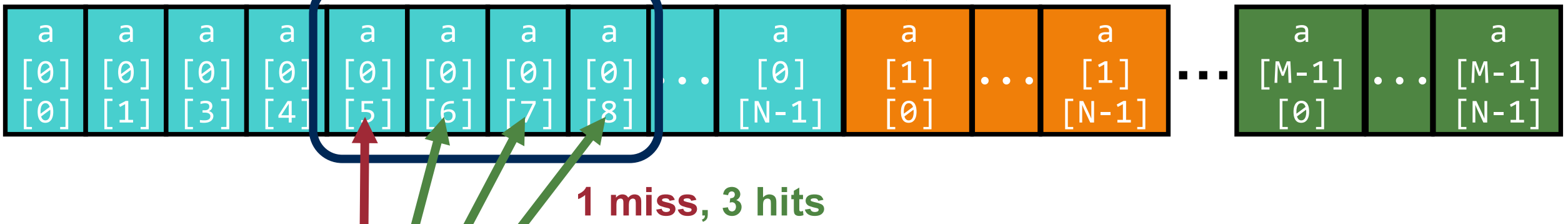
- **Stepping through columns in one row:**

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

- **Hit**: data requested is in the upper level
  - Hit ratio: hits/accesses
- **Miss**: data requested is not in the upper level
  - Block copied from lower level
  - Miss penalty: time taken to resolve miss
  - Miss ratio: misses/accesses
    = 1 – hit ratio

Terms

Layout of C Array in Memory



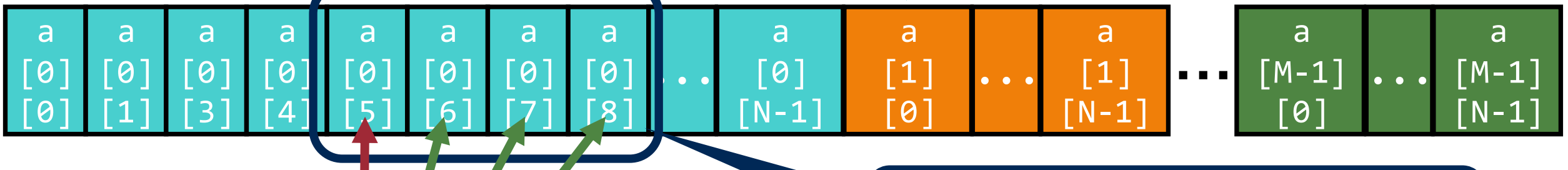
- **Stepping through columns in one row:**

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

- **Hit**: data requested is in the upper level
  - Hit ratio: hits/accesses
- **Miss**: data requested is not in the upper level
  - Block copied from lower level
  - Miss penalty: time taken to resolve miss
  - Miss ratio: misses/accesses
    = 1 – hit ratio

Terms

Layout of C Array in Memory



- Block size = 32 bytes
- sizeof(a_{ij}) = 8 bytes

- **Stepping through columns in one row:**

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

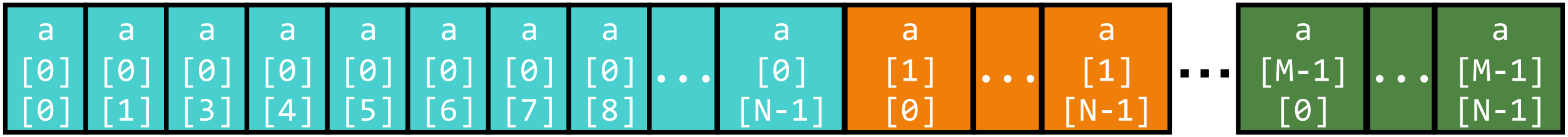
If block size > sizeof(a_{ij}) bytes, exploit spatial locality ☺

miss rate = sizeof(a_{ij}) / block size
= 1/4 = 0.25

- **Hit**: data requested is in the upper level
  - Hit ratio: hits/accesses
- **Miss**: data requested is not in the upper level
  - Block copied from lower level
  - Miss penalty: time taken to resolve miss
  - Miss ratio: misses/accesses
    = 1 – hit ratio

Terms

Layout of C Array in Memory



- **Stepping through columns in one row:**

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

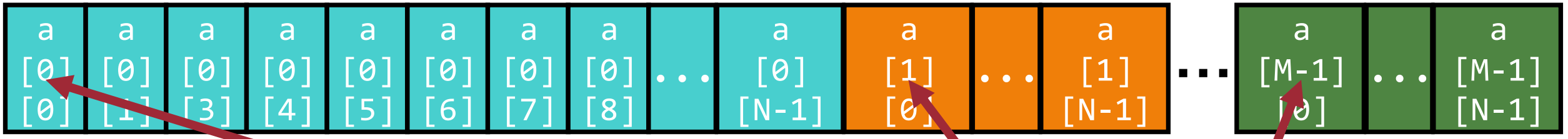
If block size > sizeof(a_{ij}) bytes, exploit spatial locality ☺

miss rate = sizeof(a_{ij}) / block size
= 1/4 = 0.25

- **Stepping through rows in one column:**

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

Layout of C Array in Memory



- **Stepping through columns in one row:**

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

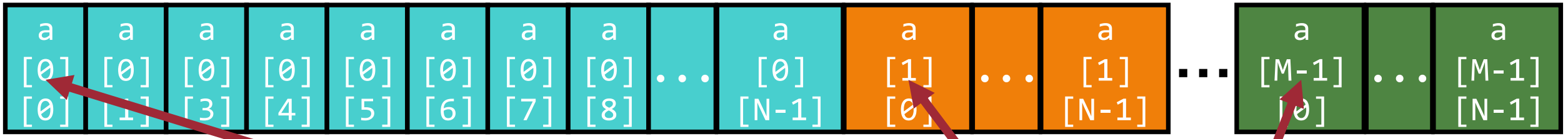
If block size > sizeof(a_{ij}) bytes, exploit spatial locality ☺

miss rate = sizeof(a_{ij}) / block size
= 1/4 = 0.25

- **Stepping through rows in one column:**

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

Layout of C Array in Memory



- **Stepping through columns in one row:**

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

If block size > sizeof(a_{ij}) bytes, exploit spatial locality ☺

miss rate = sizeof(a_{ij}) / block size
= 1/4 = 0.25

- **Stepping through rows in one column:**

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

Accesses distant elements: no spatial locality ☹

miss rate = 1 (i.e. 100%)

Three Variants: *ijk, jki, kij*

They are semantically equivalent.
In terms of the performance, which one is better?

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;
  }
}
```

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;
  }
}
```

kij

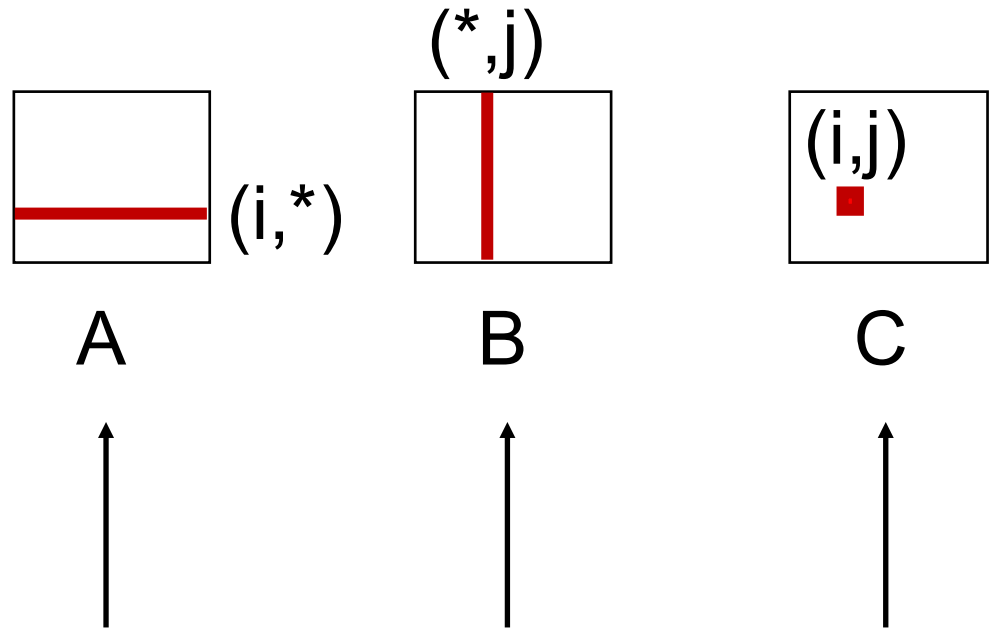
```
for (k = 0; k < n; k++) {
  for (i = 0; i < n; i++) {
    r = a[i][k];
    for (j = 0; j < n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

We need to analyze the miss rate for inner loop iterations!

Matrix Multiplication (*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:

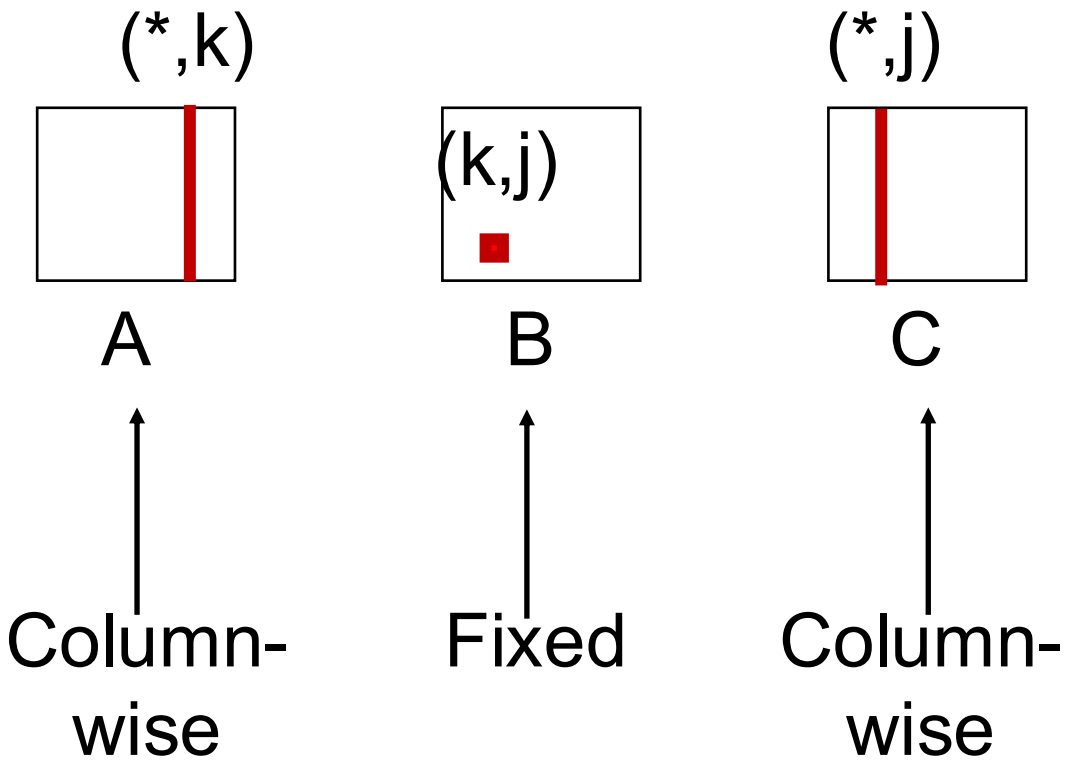


- Miss rate for inner loop iterations:
  - A: 0.25
  - B: 1.0
  - C: 0.0

Matrix Multiplication (*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:

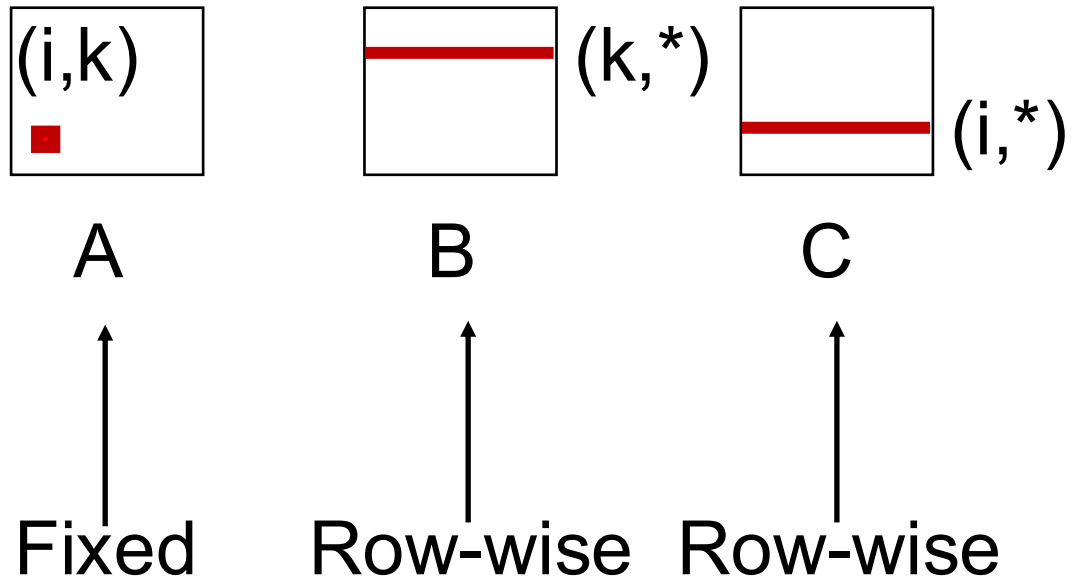


- Miss rate for inner loop iterations:
  - A: 1.0
  - B: 0.0
  - C: 1.0

Matrix Multiplication (*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:

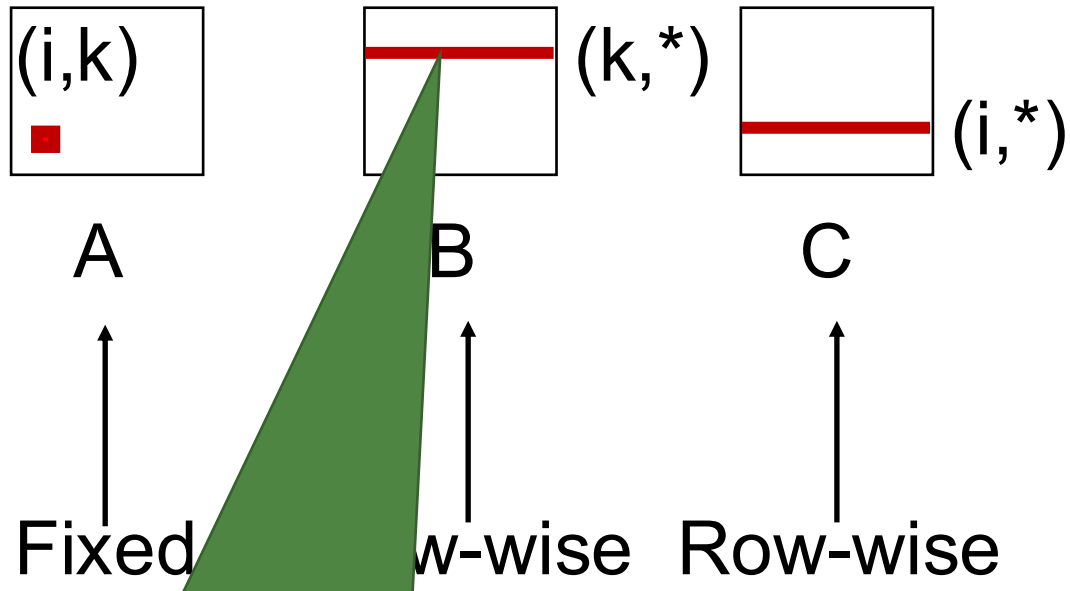


- Miss rate for inner loop iterations:
  - A: 0.0
  - B: 0.25
  - C: 0.25

Matrix Multiplication (*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:



- Miss rate for inner loop iterations:
  - A: 0.0
  - B: 0.25
  - C: 0.25

To maximize spatial locality, **access data with stride 1!**

Summary of Matrix Multiplication

ijk (& ***jik***):
- 2 loads, 0 stores
- avg misses/iter = 1.25

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

kij (& ***ikj***):
- 2 loads, 1 store
- avg misses/iter = 0.5

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;
  }
}
```

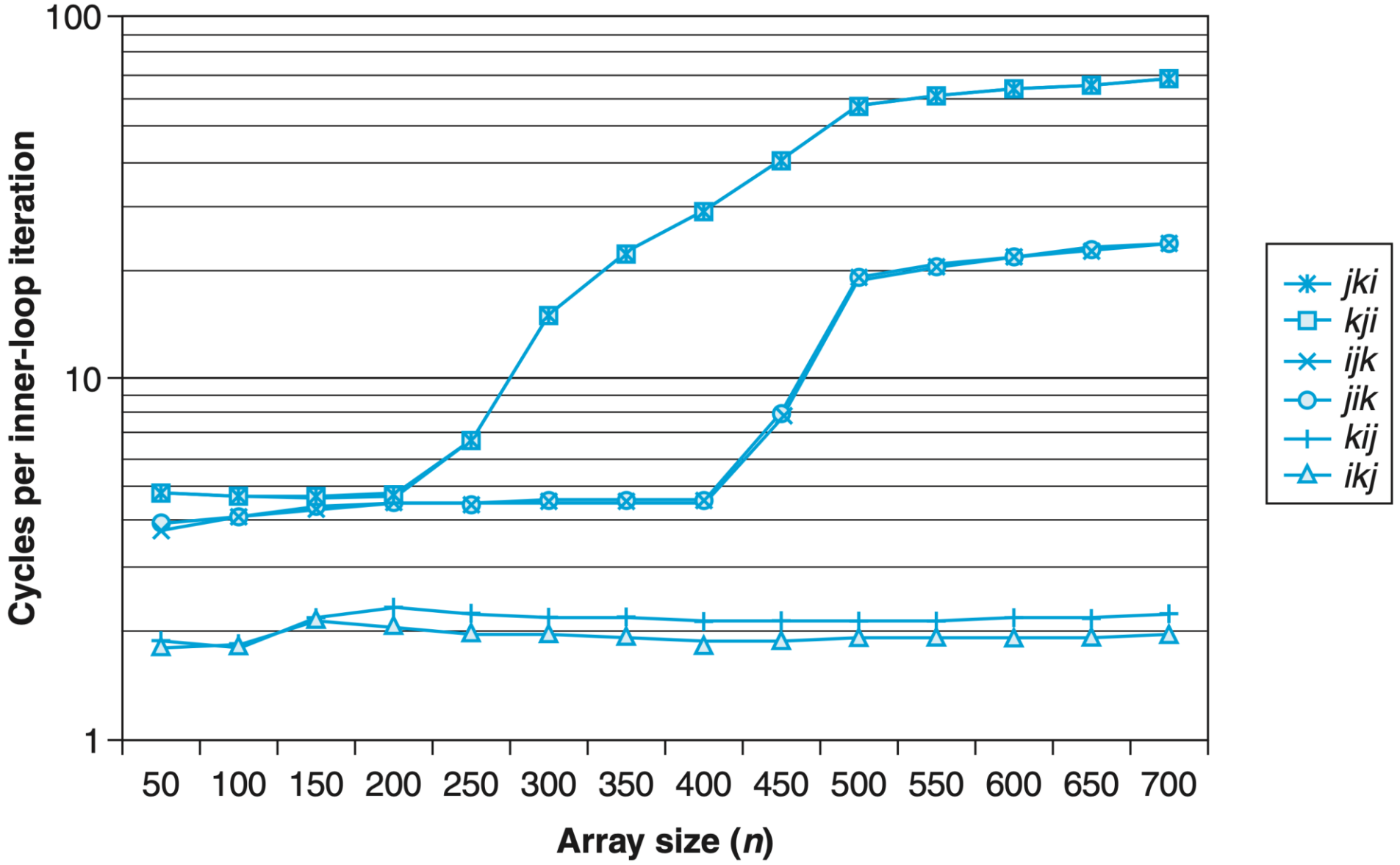
jki

```
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;
  }
}
```

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];
  }
}
```

Core i7 Matrix Multiply Performance



Cache Summary

- Cache memories can have significant performance impact

- You can write your programs to exploit this!
  - **Focus on the inner loops**, where bulk of computations and memory accesses occur
  - **Try to maximize spatial locality** by reading data objects sequentially with stride 1
  - **Try to maximize temporal locality** by using a data object as often as possible once it's read from memory

Question?