Chapter 5

Cache-aware software optimization

(Slides prepared by Luiz Santos to complement the textbook)

How to optimize array access?

Favor spatial locality

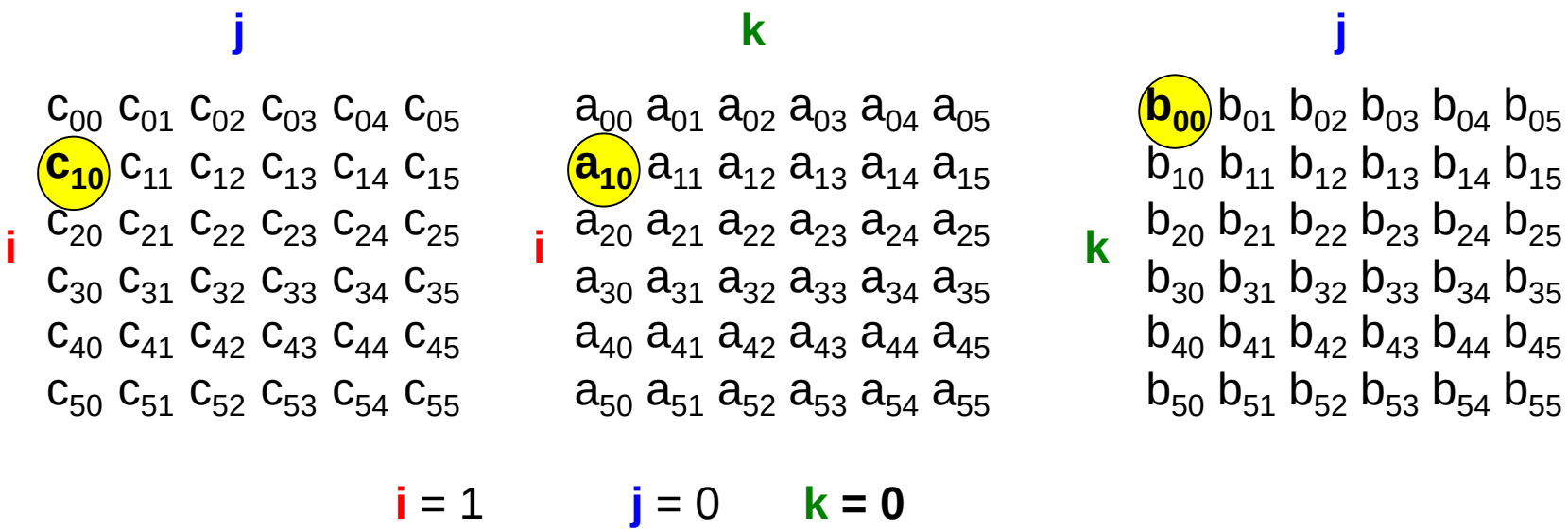
- Unidimensional arrays
  - Favor sequential access (trivial in practice)
- **One** multi-dimensional array
  - Favor sequential accesses according to row-major or column-major (language dependent)
- **Multiple** multi-dimensional arrays
  - Not all arrays may be acessed the same way
    - Not all row-major, not all column-major
    - Example: matrix multiplication

How to optimize array access?

Favor temporal locality

- If all arrays fit entirely in cache
  - All the temporal locality is exploited*
  - Example: dgemm: C = C + A x B
    - A, B, C: 32 x 32, double-precision elements
    - 3 x 2^5 x 2^5 x 2^3 bytes = 24KiB < 32KiB (i7)
- If arrays don´t fit, some locality unexploited
  - Divide them in **smaller blocks** that fit in cache
  - So as keep repeated accesses in cache
    - As much as possible

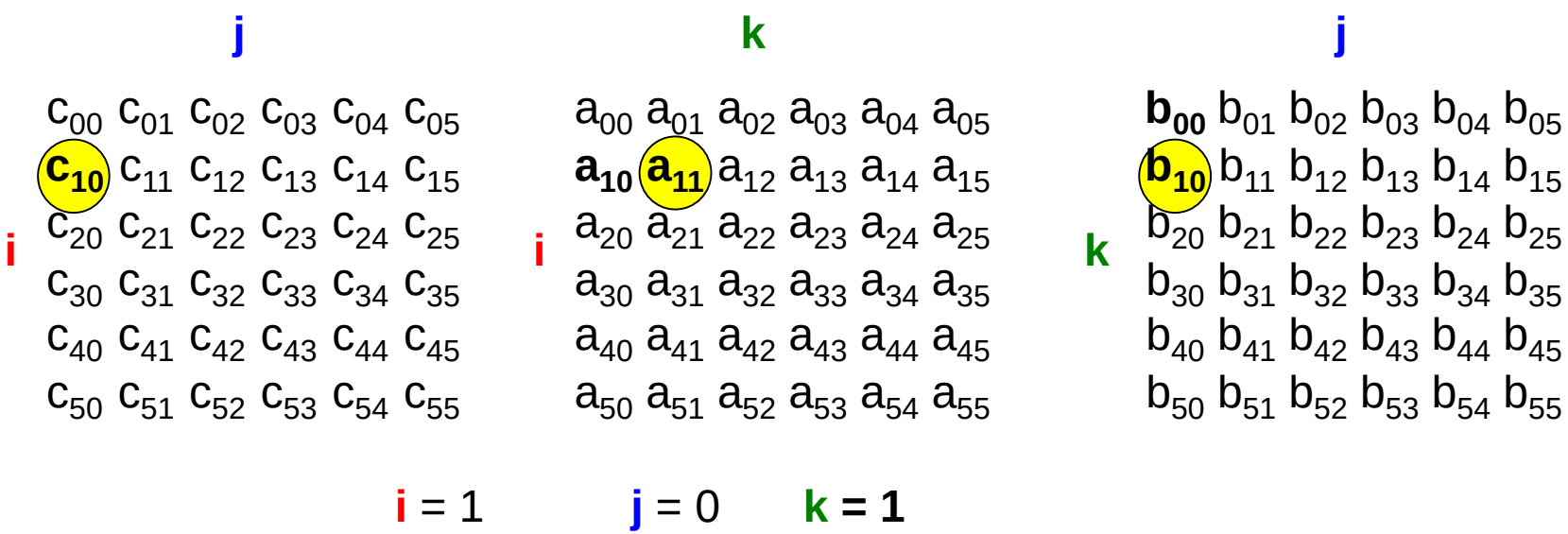
Original traversal



i = 1 **j** = 0 **k = 0**

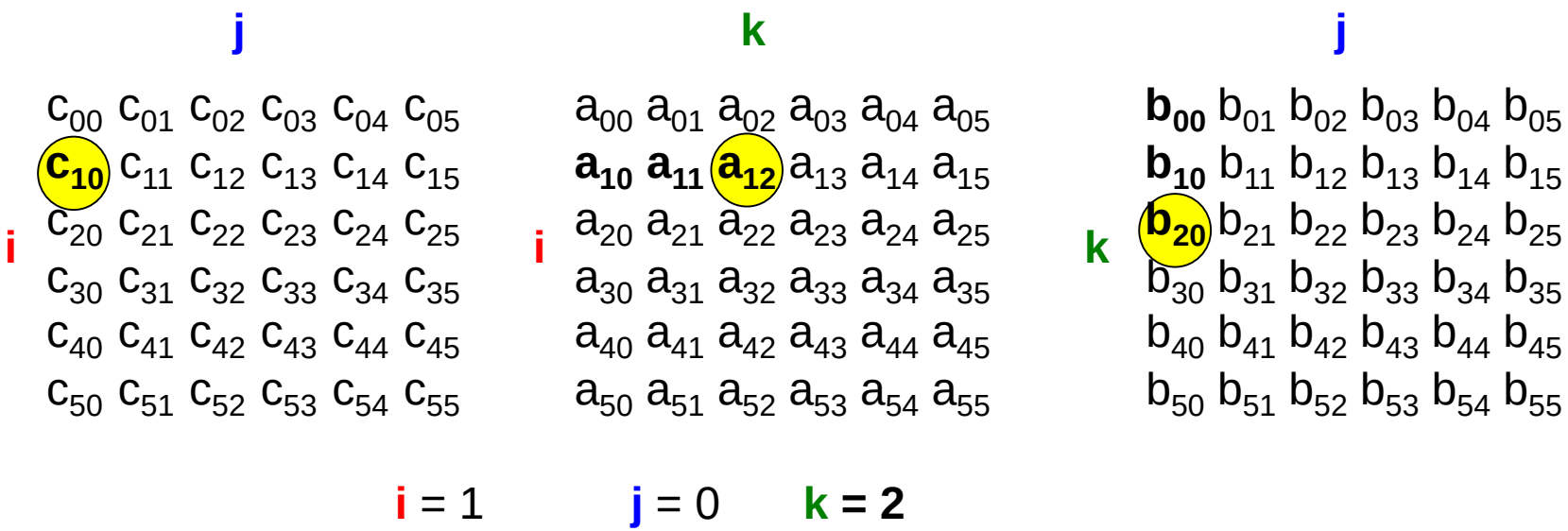
```
4.     for (int j = 0; j < n; ++j)
5.     {
6.      double cij = C[i+j*n]; /* cij = C[i][j] */
7.      for(int k = 0; k < n; k++ )
8.       cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
9.      C[i+j*n] = cij; /* C[i][j] = cij */
```

Original traversal




```
4.     for (int j = 0; j < n; ++j)
5.     {
6.      double cij = C[i+j*n]; /* cij = C[i][j] */
7.      for(int k = 0; k < n; k++ )
8.       cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
9.      C[i+j*n] = cij; /* C[i][j] = cij */
```

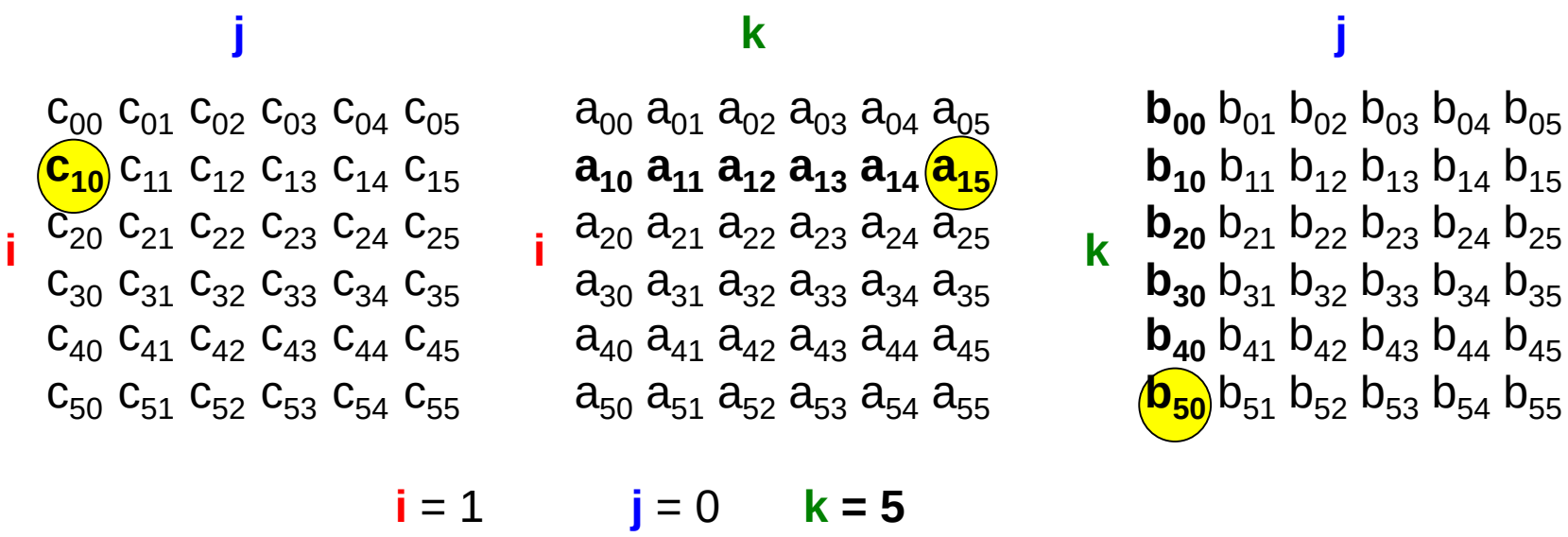
Original traversal



i = 1 **j** = 0 **k = 2**

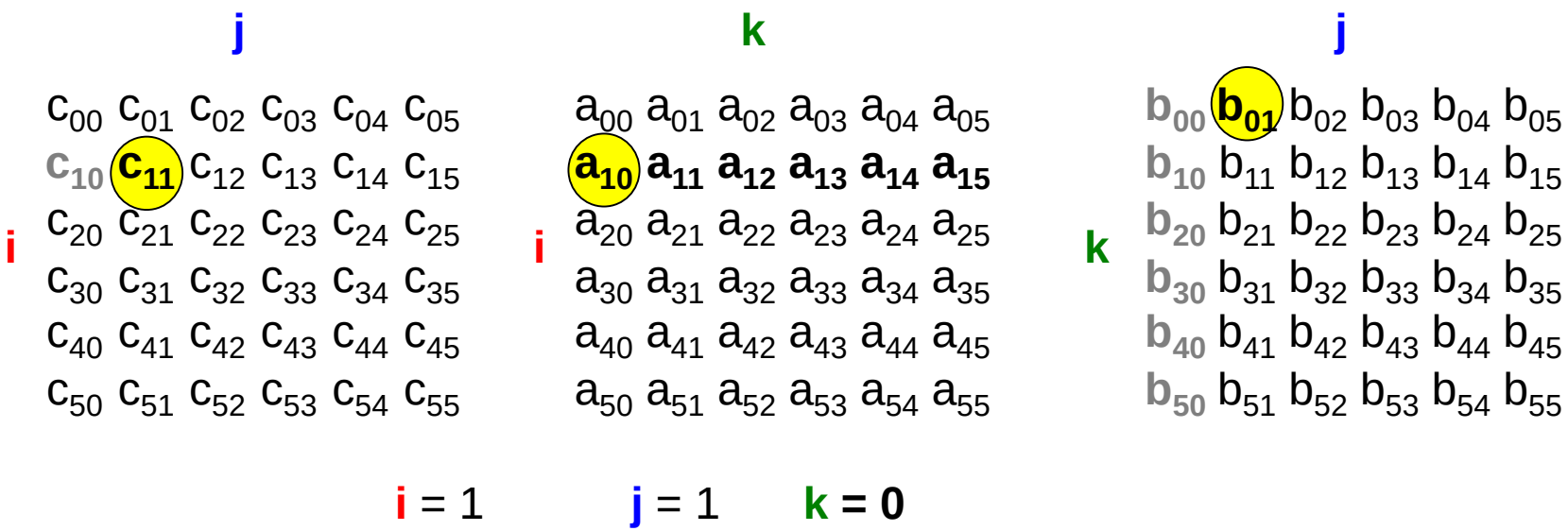
```
4.     for (int j = 0; j < n; ++j)
5.     {
6.      double cij = C[i+j*n]; /* cij = C[i][j] */
7.      for(int k = 0; k < n; k++ )
8.       cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
9.      C[i+j*n] = cij; /* C[i][j] = cij */
```

Original traversal



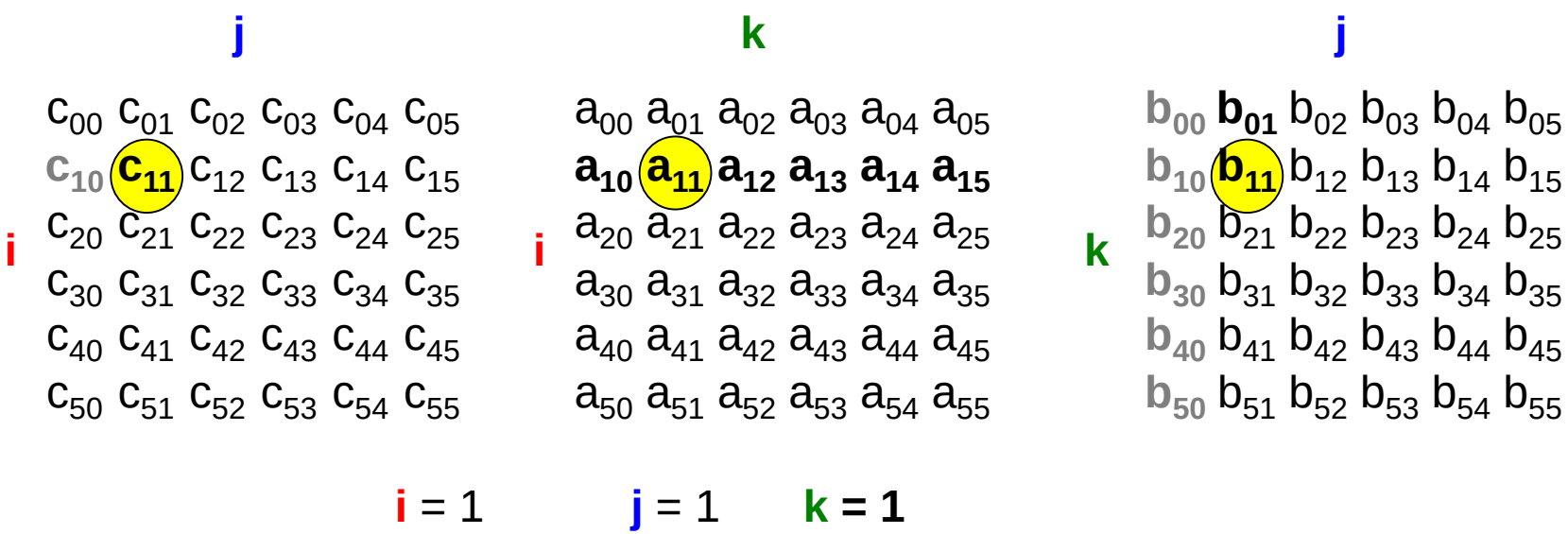

```
4.     for (int j = 0; j < n; ++j)
5.     {
6.      double cij = C[i+j*n]; /* cij = C[i][j] */
7.      for(int k = 0; k < n; k++ )
8.       cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
9.      C[i+j*n] = cij; /* C[i][j] = cij */
```

Original traversal



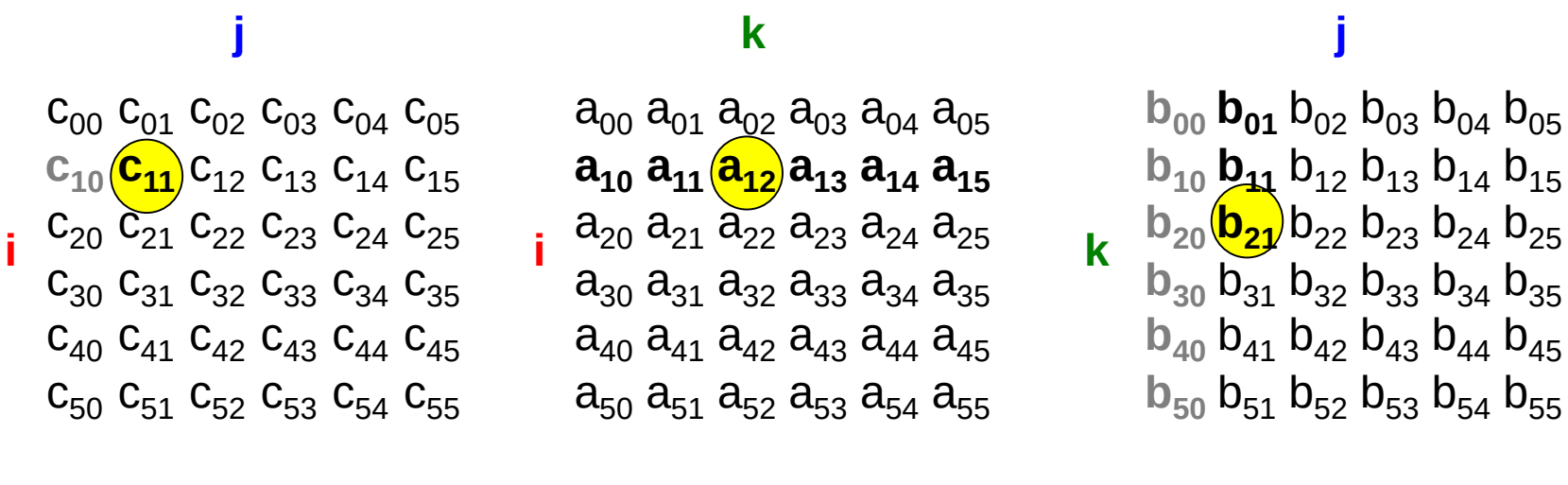

```
4.     for (int j = 0; j < n; ++j)
5.     {
6.      double cij = C[i+j*n]; /* cij = C[i][j] */
7.      for(int k = 0; k < n; k++ )
8.       cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
9.      C[i+j*n] = cij; /* C[i][j] = cij */
```

Original traversal



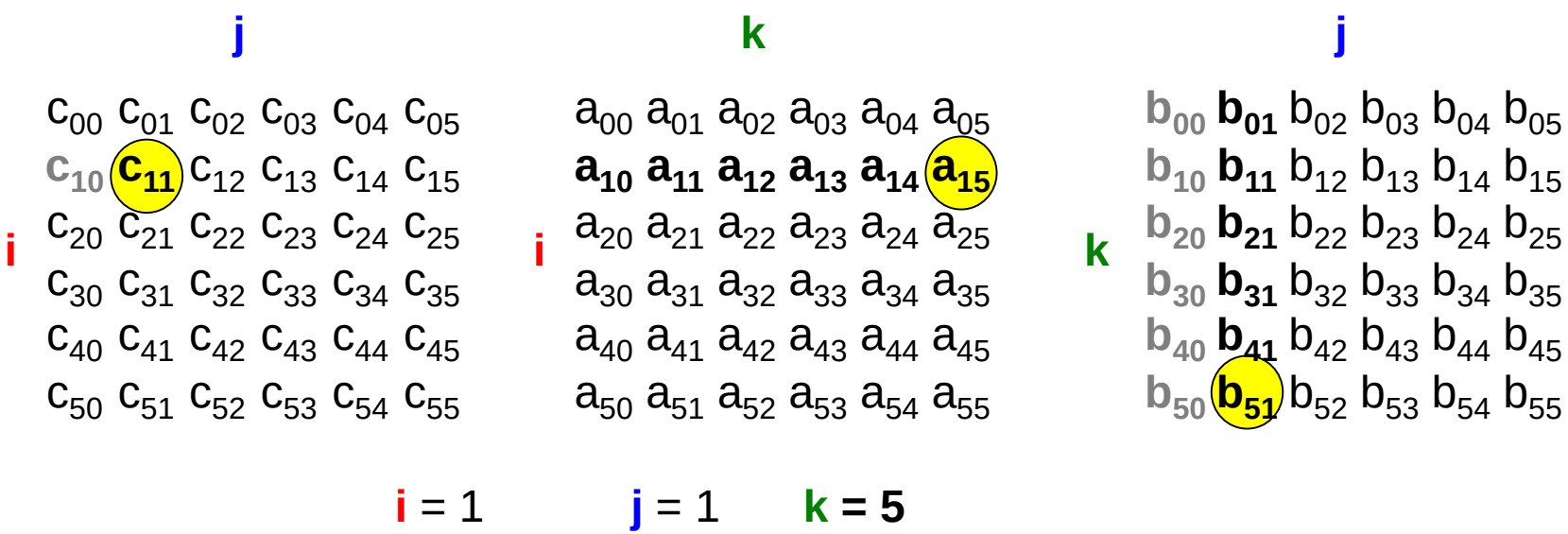

```
4.     for (int j = 0; j < n; ++j)
5.     {
6.      double cij = C[i+j*n]; /* cij = C[i][j] */
7.      for(int k = 0; k < n; k++ )
8.       cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
9.      C[i+j*n] = cij; /* C[i][j] = cij */
```

Original traversal




```
4.     for (int j = 0; j < n; ++j)
5.     {
6.      double cij = C[i+j*n]; /* cij = C[i][j] */
7.      for(int k = 0; k < n; k++ )
8.       cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
9.      C[i+j*n] = cij; /* C[i][j] = cij */
```

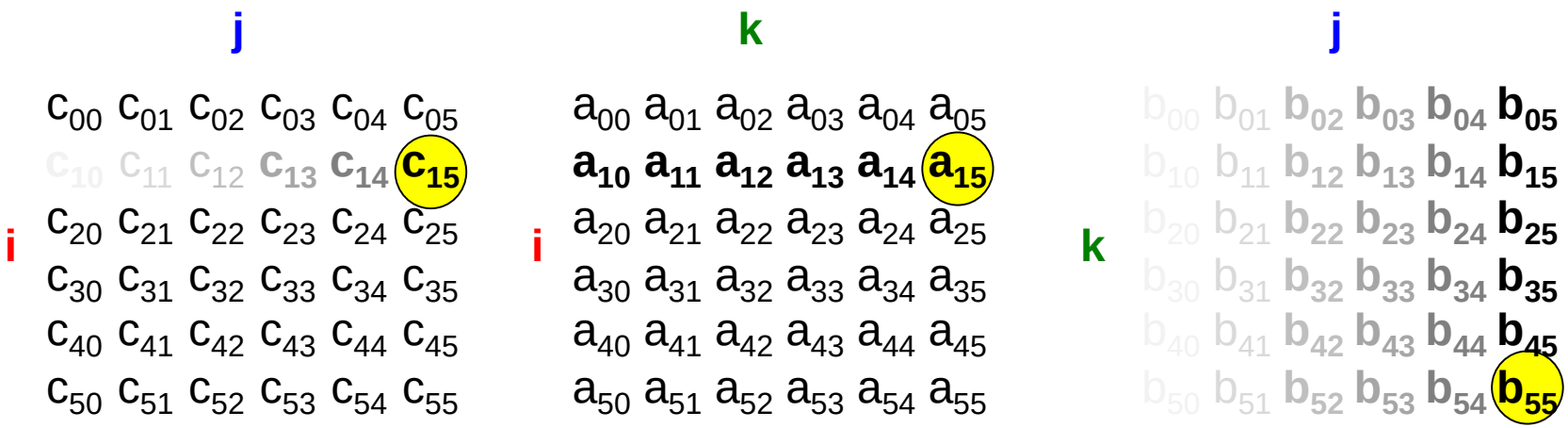
Original traversal




```
4.     for (int j = 0; j < n; ++j)
5.     {
6.      double cij = C[i+j*n]; /* cij = C[i][j] */
7.      for(int k = 0; k < n; k++ )
8.       cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
9.      C[i+j*n] = cij; /* C[i][j] = cij */
```

Original traversal

To determine one row of C,
one row of A is visited n times;
all n columns of B are visited once



To determine the next row of C,
all columns of B will be revisited.

What if B doesn´t entirely fit in cache?

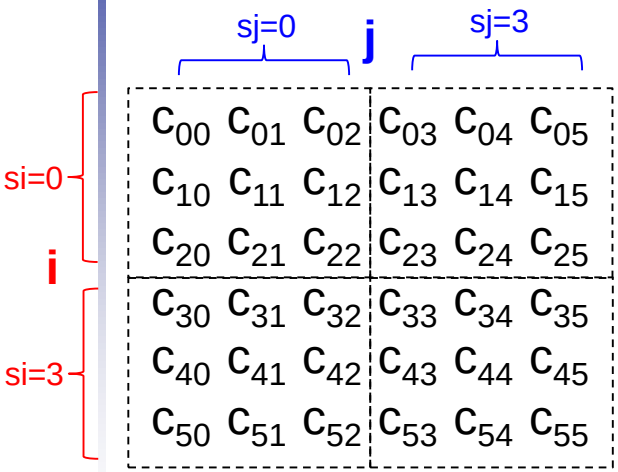
```
4.     for (int j = 0; j < n; ++j)
5.     {
6.      double cij = C[i+j*n]; /* cij = C[i][j] */
7.      for(int k = 0; k < n; k++ )
8.       cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
9.      C[i+j*n] = cij; /* C[i][j] = cij */
```

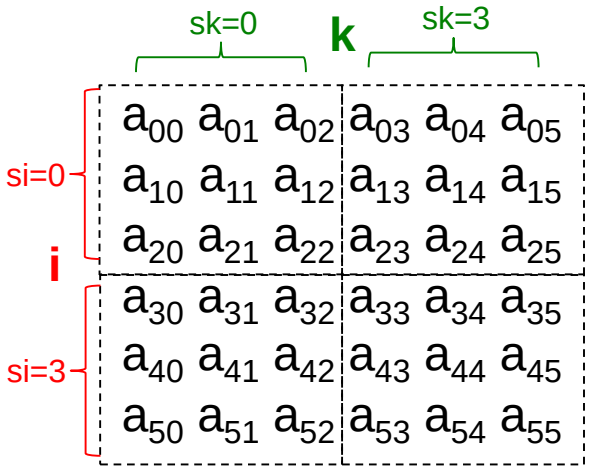
Software optimization via blocking

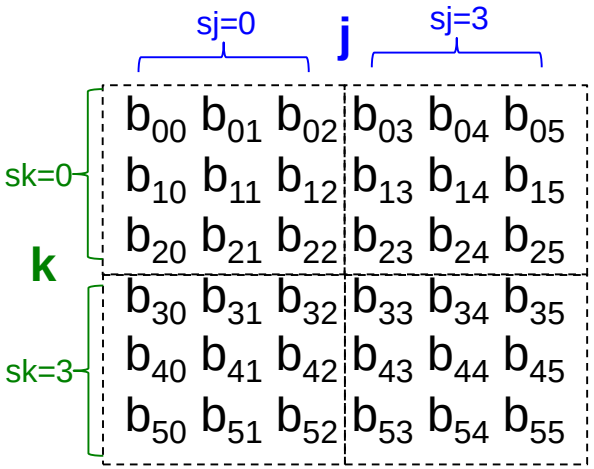
```
void dgemm (int n, double *A, double *B, double *C)
{
    for (int sj = 0; sj < n; sj+=BLOCKSIZE )
        for (int si = 0; si < n; si+=BLOCKSIZE )
            for (int sk = 0; sk < n; sk+=BLOCKSIZE )
                do_block(n, si, sj, sk, A, B, C)
}

void do_block (int n, int si, int sj, int sk, double *A, double *B)
for (int i = si; i < si + BLOCKSIZE; ++i)
    for (int j = sj; j < sj + BLOCKSIZE; ++j)
      {
        double cij = C[i+j*n]; /* cij = C[i][j] */
        for(int k = sk; k < sk + BLOCKSIZE; k++ )
           cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
        C[i+j*n] = cij; /* C[i][j] = cij */
      }
}
```

Matrix traversal with blocking



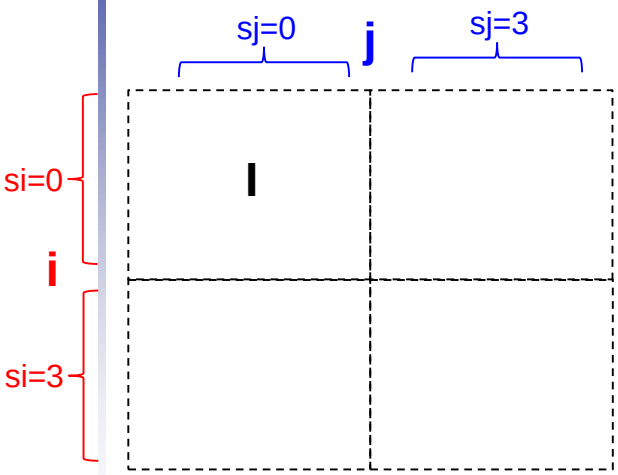


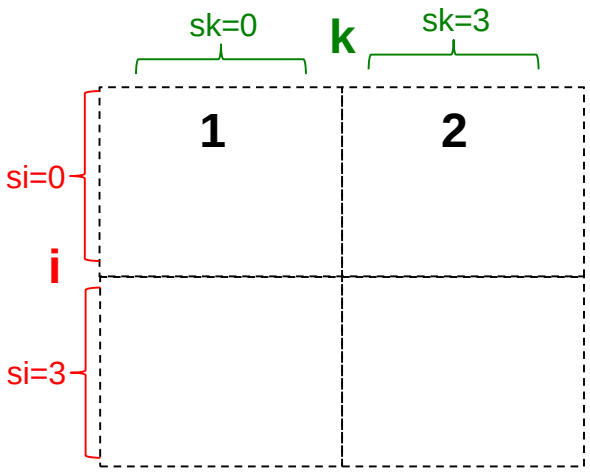


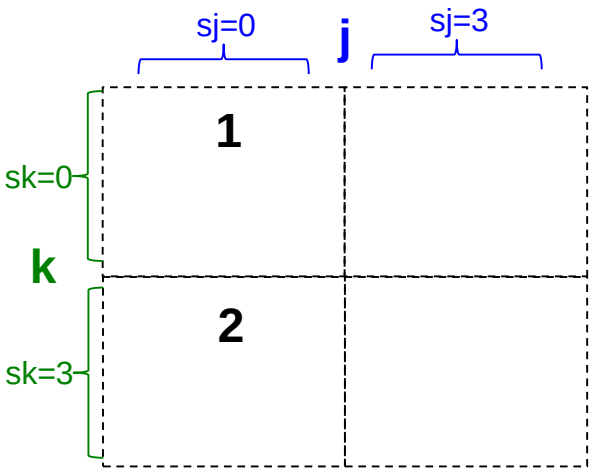
sj=0
- si=0
  - sk=0
  - sk=3
- si=3
  - sk=0
  - sk=3

sj=3
- si=0
  - sk=0
  - sk=3
- si=3
  - sk=0
  - sk=3

Matrix traversal with blocking







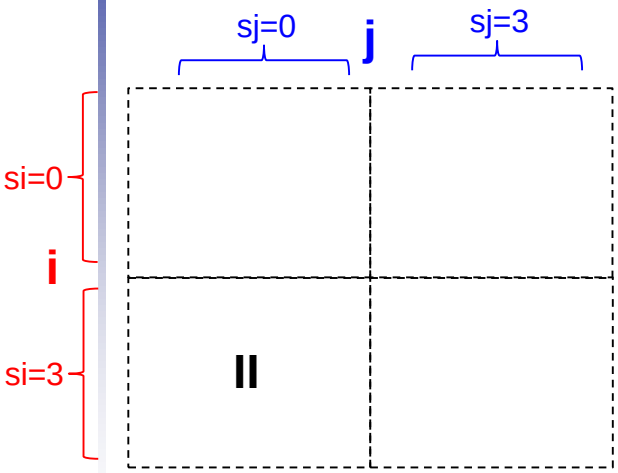
sj=0

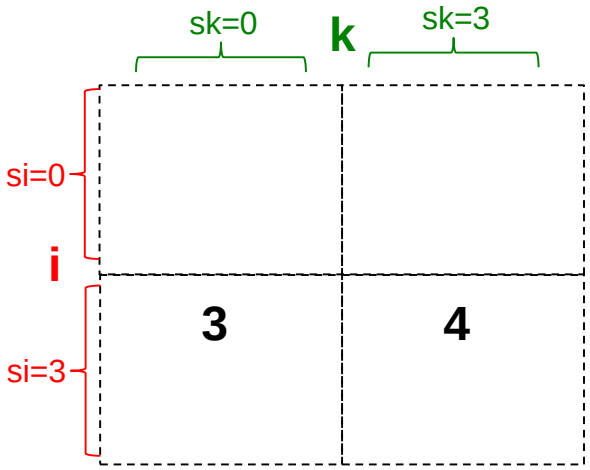
si=0 **(I)**

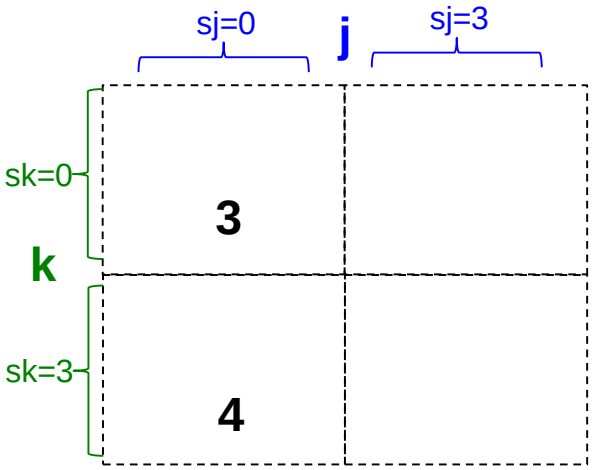
sk=0 **(1)**

sk=3 **(2)**

Matrix traversal with blocking



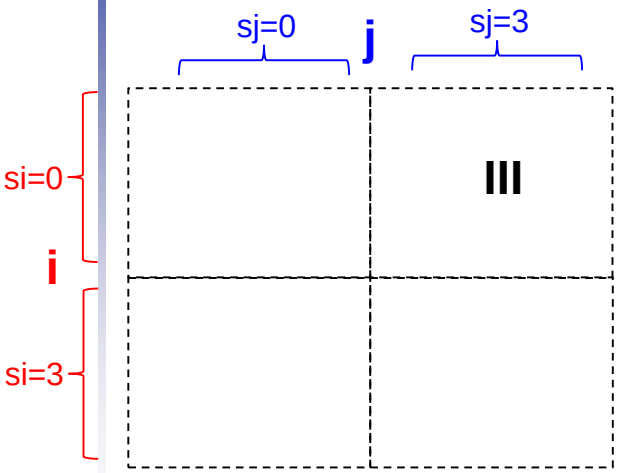


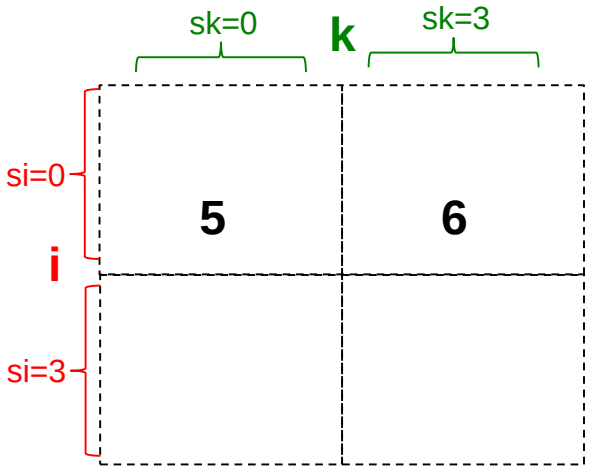


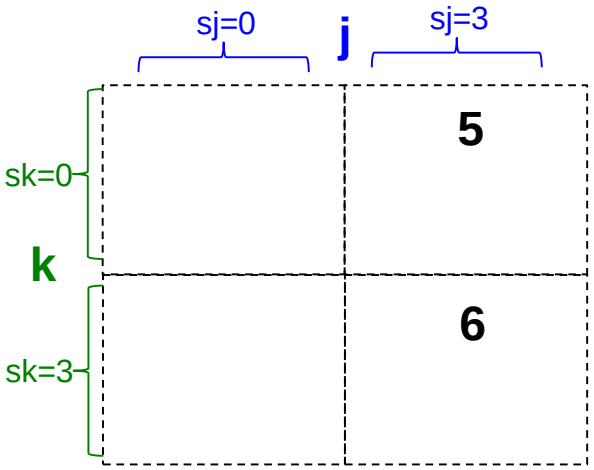
sj=0

- si=0 **(I)**
  - sk=0 **(1)**
  - sk=3 **(2)**
- si=3 **(II)**
  - sk=0 **(3)**
  - sk=3 **(4)**

Matrix traversal with blocking







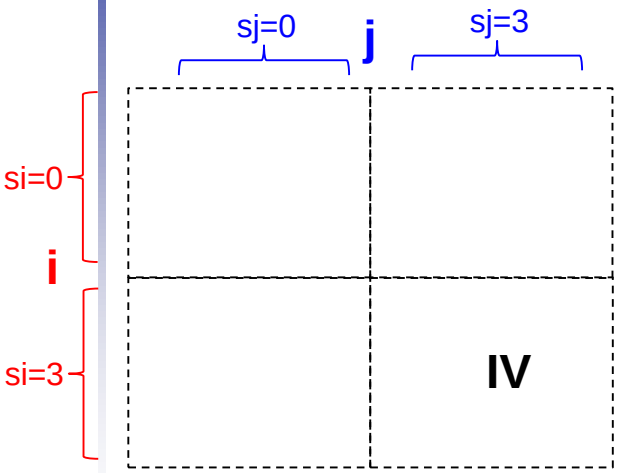
sj=0

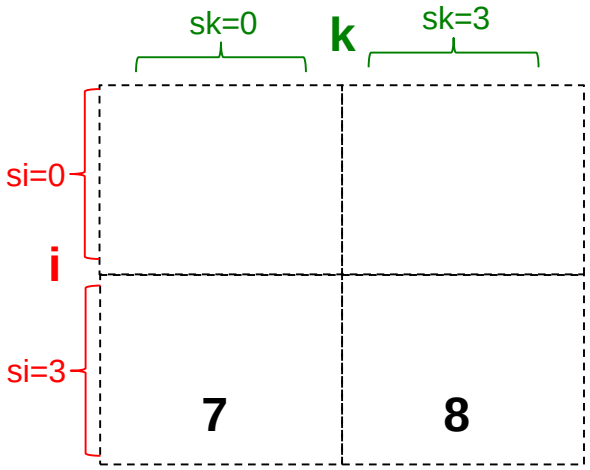
- si=0 **(I)**
  - sk=0 **(1)**
  - sk=3 **(2)**
- si=3 **(II)**
  - sk=0 **(3)**
  - sk=3 **(4)**

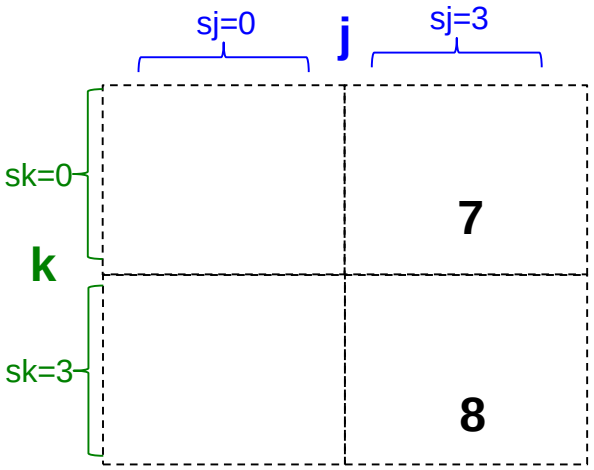
sj=3

- si=0 **(III)**
  - sk=0 **(5)**
  - sk=3 **(6)**

Matrix traversal with blocking







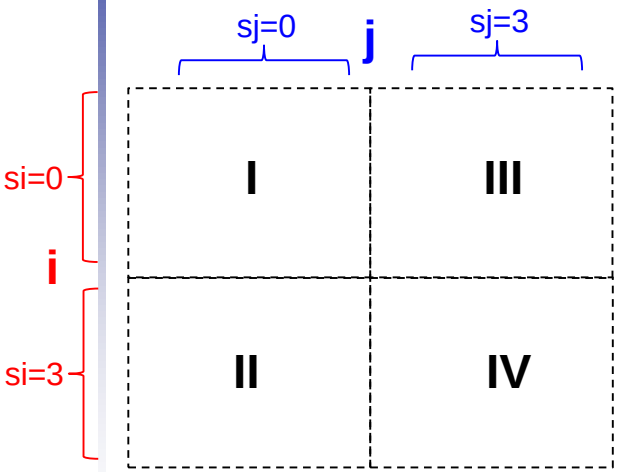
sj=0

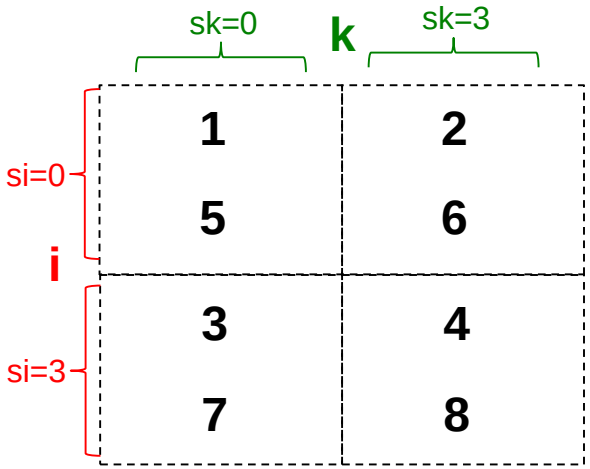
- si=0 **(I)**
  - sk=0 **(1)**
  - sk=3 **(2)**
- si=3 **(II)**
  - sk=0 **(3)**
  - sk=3 **(4)**

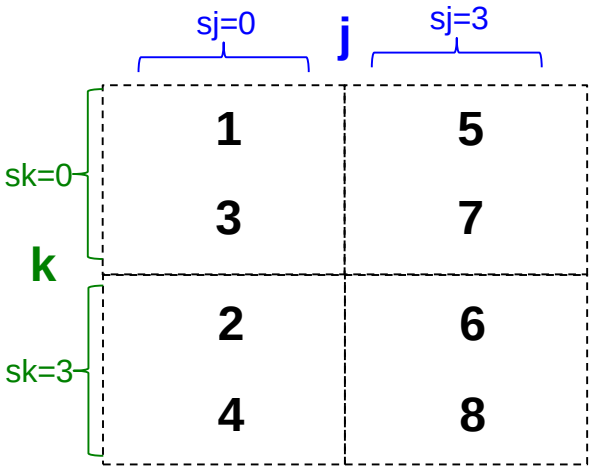
sj=3

- si=0 **(III)**
  - sk=0 **(5)**
  - sk=3 **(6)**
- si=3 **(IV)**
  - sk=0 **(7)**
  - sk=3 **(8)**

Matrix traversal with blocking







sj=0

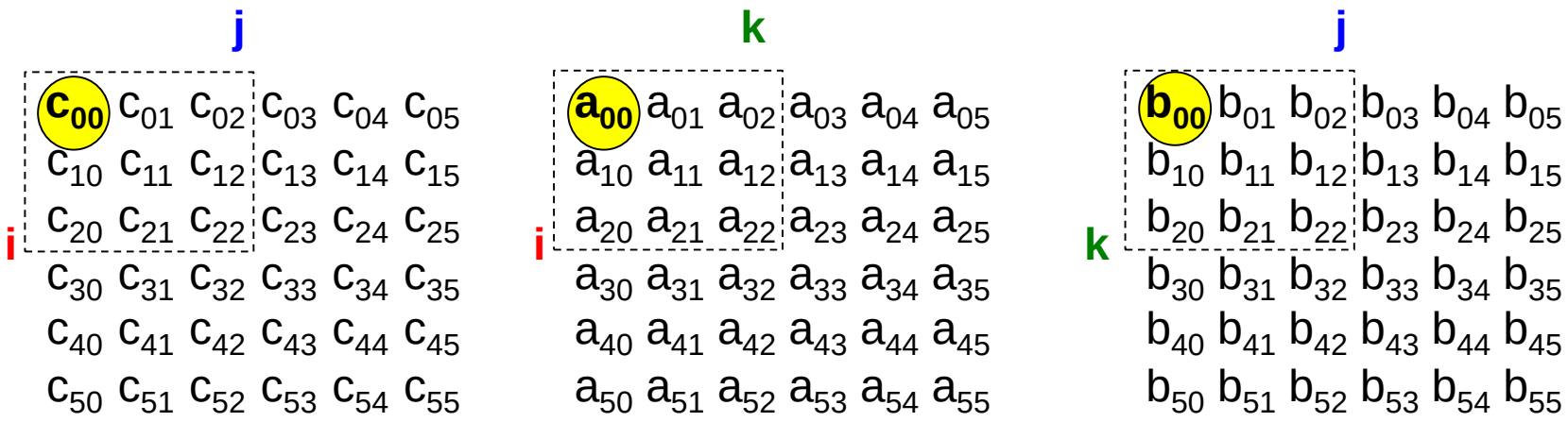
- si=0 **(I)**
  - sk=0 **(1)**
  - sk=3 **(2)**
- si=3 **(II)**
  - sk=0 **(3)**
  - sk=3 **(4)**

sj=3

- si=0 **(III)**
  - sk=0 **(5)**
  - sk=3 **(6)**
- si=3 **(IV)**
  - sk=0 **(7)**
  - sk=3 **(8)**

All blocks of **every** matrix are visited **twice**. In the first visit, **compulsory** misses; in the second, no misses **only when** capacity* enough to keep visited block in cache

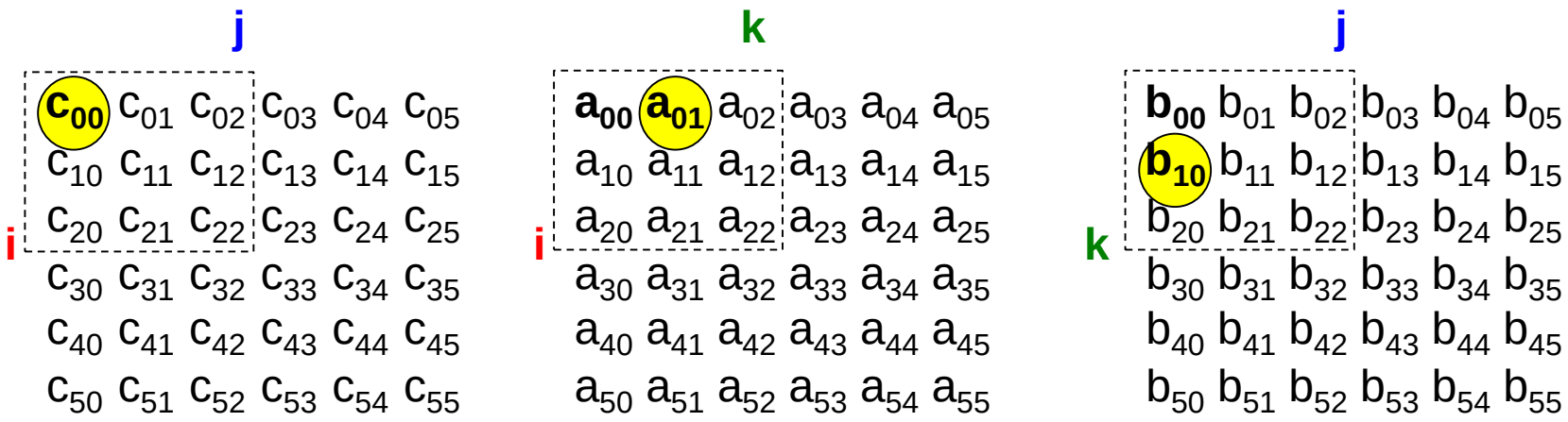
Block traversal (si=0, sj=0, sk=0)



i = 0 **j** = 0 **k = 0** **BLOCKSIZE = 3**

```
for (int i = si; i < si + BLOCKSIZE; ++i)
     for (int j = sj; j < sj + BLOCKSIZE; ++j)
       {
         double cij = C[i+j*n]; /* cij = C[i][j] */
         for(int k = sk; k < sk + BLOCKSIZE; k++ )
            cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
         C[i+j*n] = cij; /* C[i][j] = cij */
       }
}
```

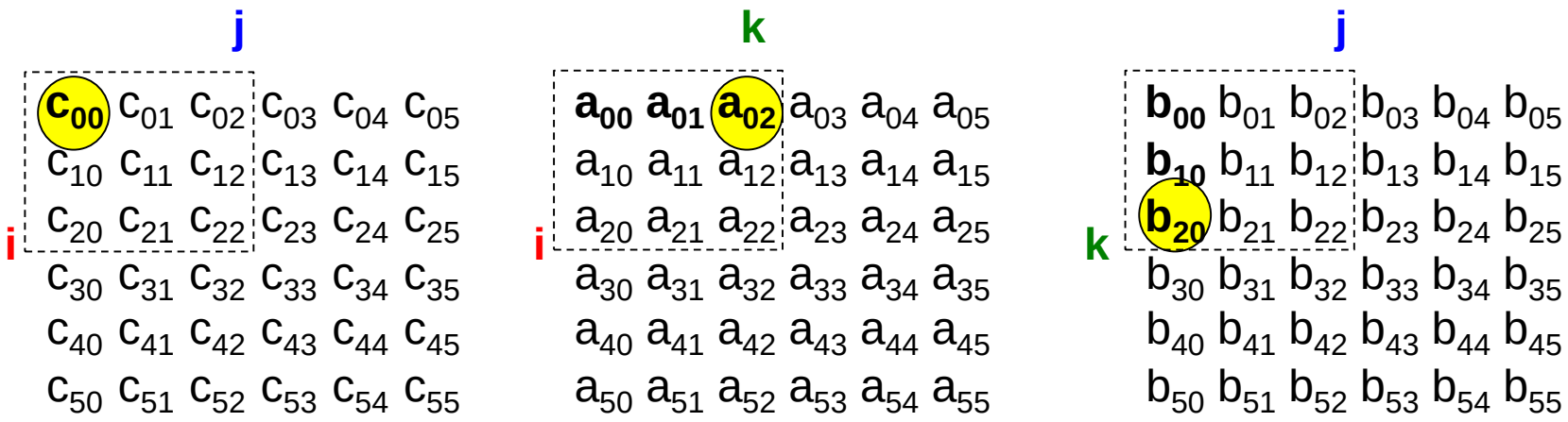
Block traversal (si=0, sj=0, sk=0)



i = 0 **j** = 0 **k = 1** **BLOCKSIZE = 3**

```
for (int i = si; i < si + BLOCKSIZE; ++i)
     for (int j = sj; j < sj + BLOCKSIZE; ++j)
       {
         double cij = C[i+j*n]; /* cij = C[i][j] */
         for(int k = sk; k < sk + BLOCKSIZE; k++ )
            cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
         C[i+j*n] = cij; /* C[i][j] = cij */
       }
}
```

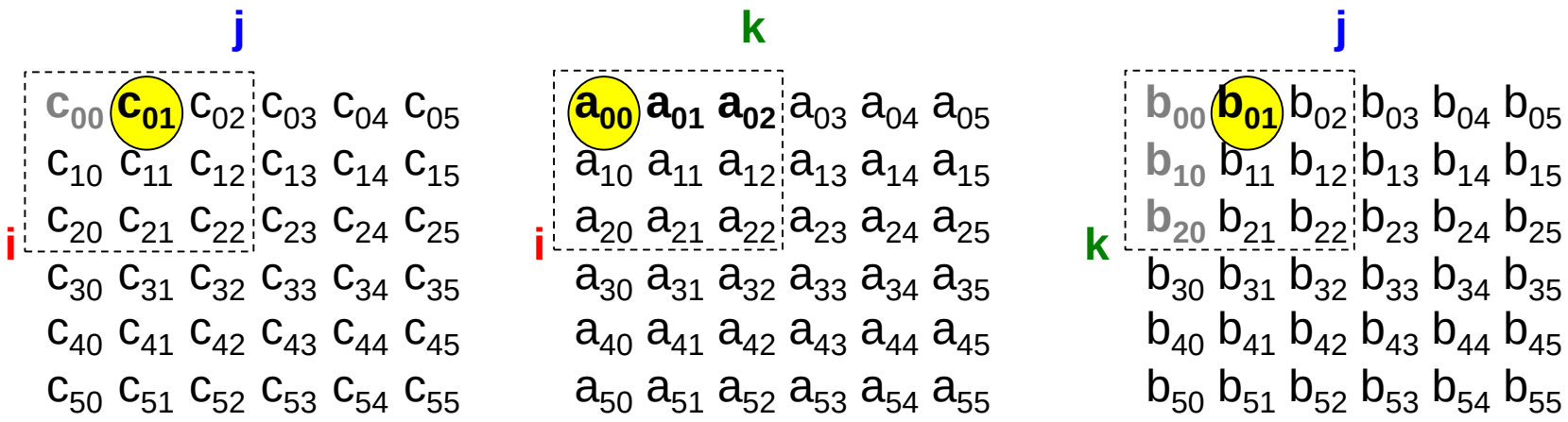
Block traversal (si=0, sj=0, sk=0)



i = 0 **j** = 0 **k = 2** **BLOCKSIZE = 3**

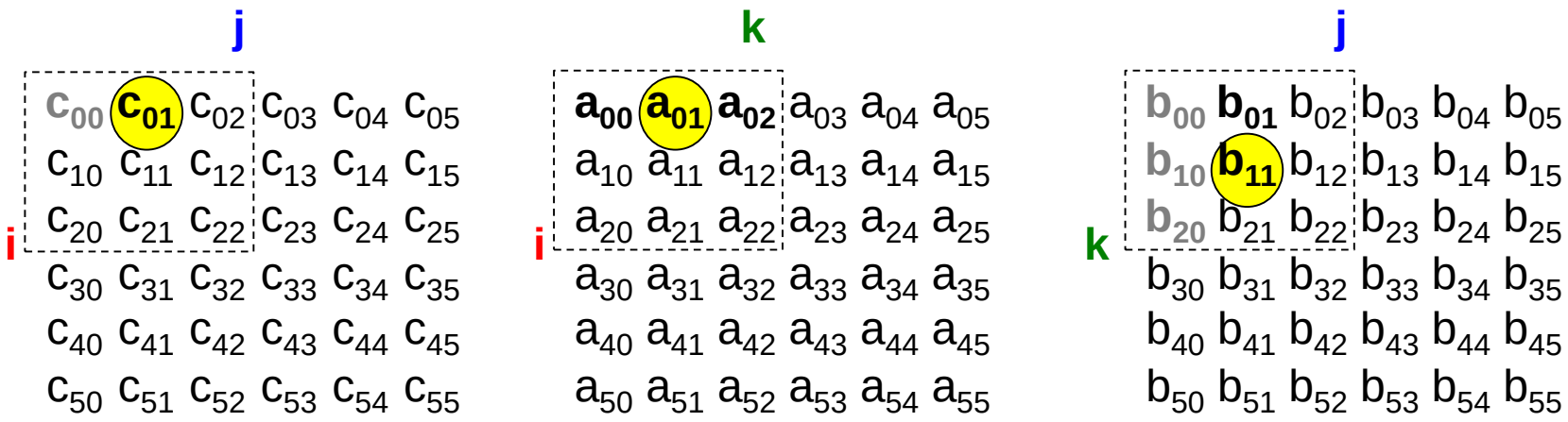
```
for (int i = si; i < si + BLOCKSIZE; ++i)
     for (int j = sj; j < sj + BLOCKSIZE; ++j)
       {
         double cij = C[i+j*n]; /* cij = C[i][j] */
         for(int k = sk; k < sk + BLOCKSIZE; k++ )
            cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
         C[i+j*n] = cij; /* C[i][j] = cij */
       }
}
```

Block traversal (si=0, sj=0, sk=0)



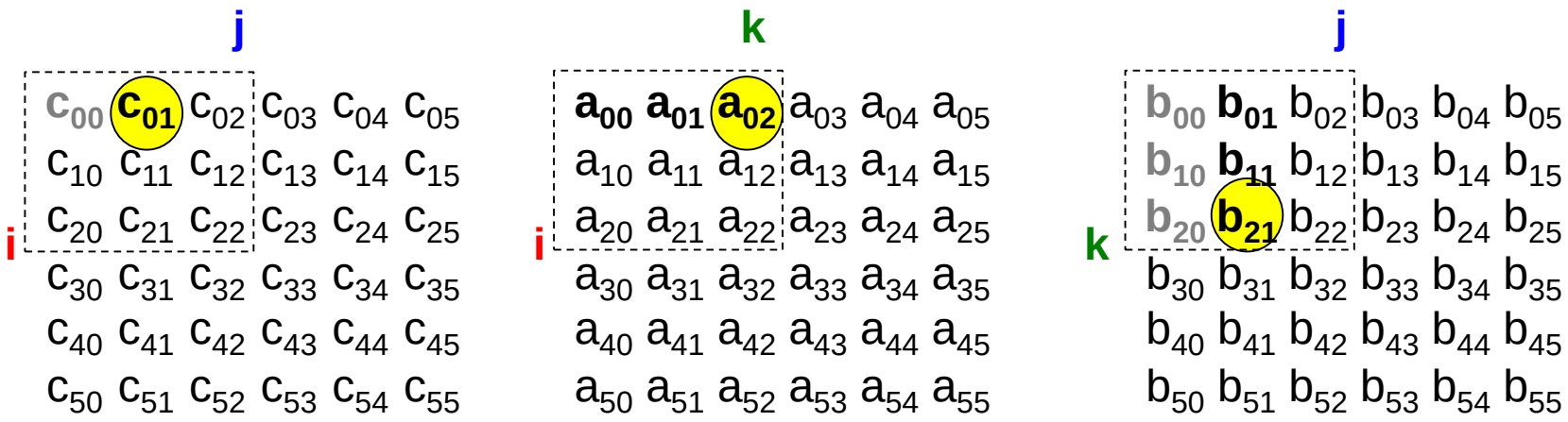

```
for (int i = si; i < si + BLOCKSIZE; ++i)
     for (int j = sj; j < sj + BLOCKSIZE; ++j)
       {
         double cij = C[i+j*n]; /* cij = C[i][j] */
         for(int k = sk; k < sk + BLOCKSIZE; k++ )
            cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
         C[i+j*n] = cij; /* C[i][j] = cij */
       }
}
```

Block traversal (si=0, sj=0, sk=0)



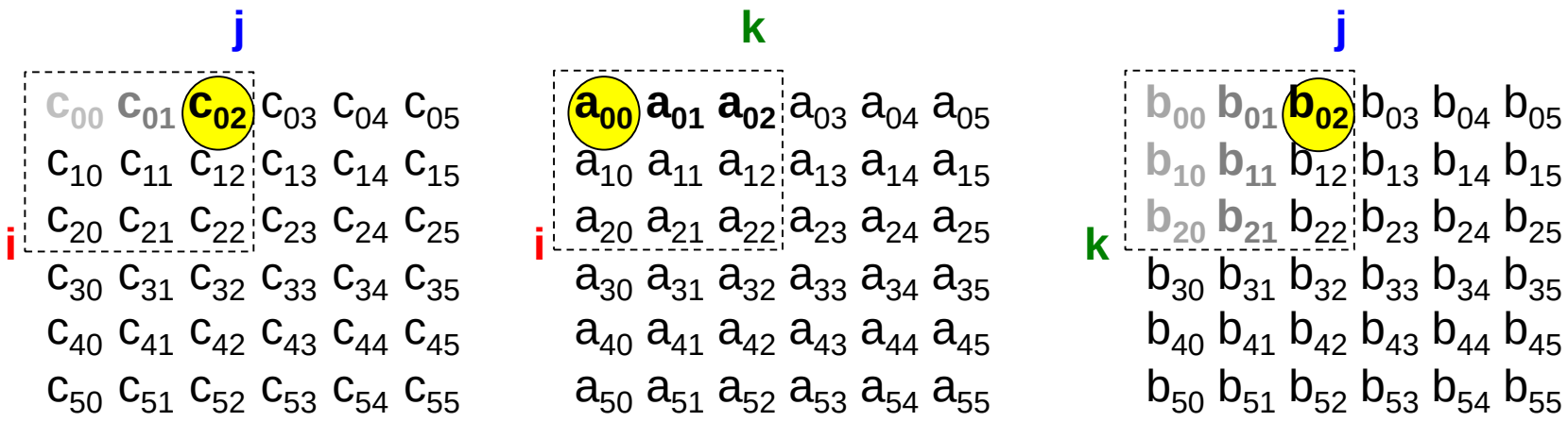

```
for (int i = si; i < si + BLOCKSIZE; ++i)
     for (int j = sj; j < sj + BLOCKSIZE; ++j)
       {
         double cij = C[i+j*n]; /* cij = C[i][j] */
         for(int k = sk; k < sk + BLOCKSIZE; k++ )
            cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
         C[i+j*n] = cij; /* C[i][j] = cij */
       }
}
```

Block traversal (si=0, sj=0, sk=0)




```
for (int i = si; i < si + BLOCKSIZE; ++i)
     for (int j = sj; j < sj + BLOCKSIZE; ++j)
       {
         double cij = C[i+j*n]; /* cij = C[i][j] */
         for(int k = sk; k < sk + BLOCKSIZE; k++ )
            cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
         C[i+j*n] = cij; /* C[i][j] = cij */
       }
}
```

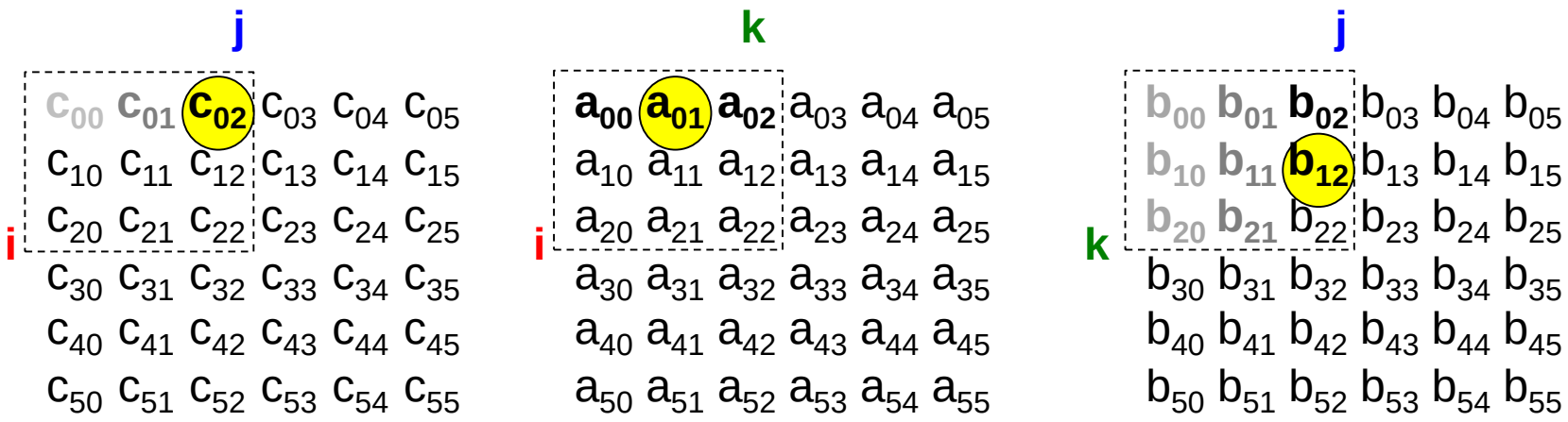
Block traversal (si=0, sj=0, sk=0)



i = 0 **j** = 2 **k** = 0 **BLOCKSIZE = 3**

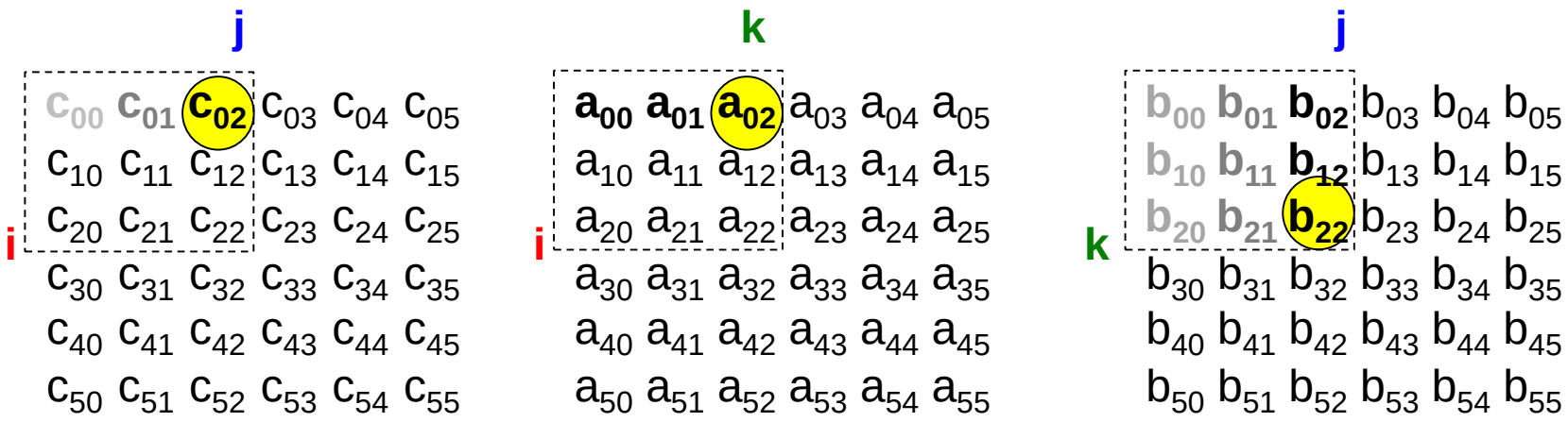
```
for (int i = si; i < si + BLOCKSIZE; ++i)
     for (int j = sj; j < sj + BLOCKSIZE; ++j)
       {
         double cij = C[i+j*n]; /* cij = C[i][j] */
         for(int k = sk; k < sk + BLOCKSIZE; k++ )
            cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
         C[i+j*n] = cij; /* C[i][j] = cij */
       }
}
```

Block traversal (si=0, sj=0, sk=0)




```
for (int i = si; i < si + BLOCKSIZE; ++i)
     for (int j = sj; j < sj + BLOCKSIZE; ++j)
       {
         double cij = C[i+j*n]; /* cij = C[i][j] */
         for(int k = sk; k < sk + BLOCKSIZE; k++ )
            cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
         C[i+j*n] = cij; /* C[i][j] = cij */
       }
}
```

Block traversal (si=0, sj=0, sk=0)

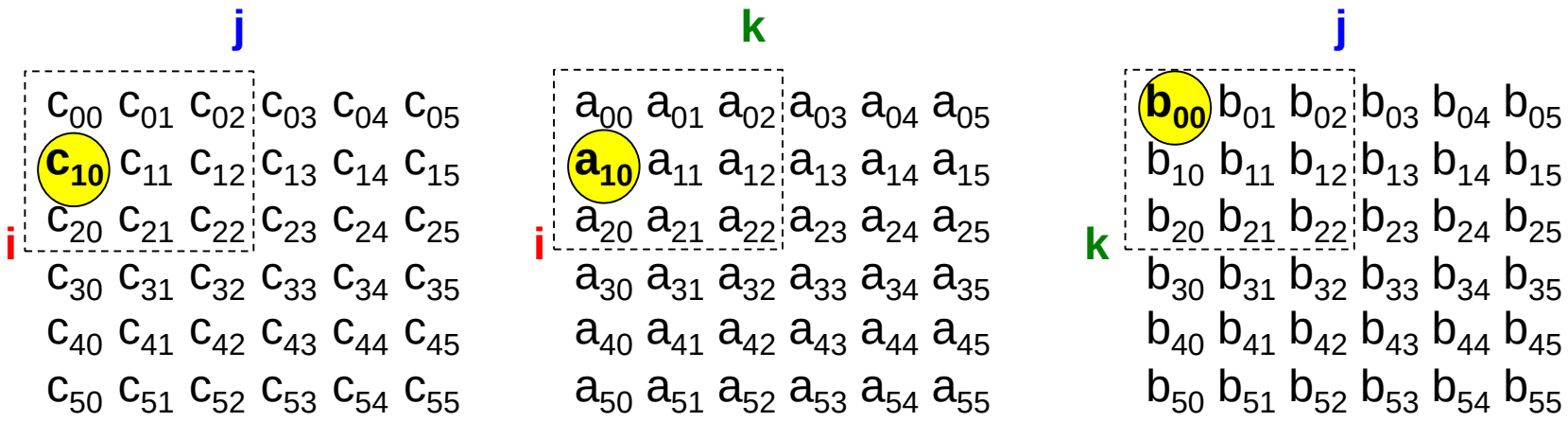


i = 0 **j** = 2 **k = 2** **BLOCKSIZE = 3**

```
for (int i = si; i < si + BLOCKSIZE; ++i)
     for (int j = sj; j < sj + BLOCKSIZE; ++j)
       {
         double cij = C[i+j*n]; /* cij = C[i][j] */
         for(int k = sk; k < sk + BLOCKSIZE; k++ )
            cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
         C[i+j*n] = cij; /* C[i][j] = cij */
       }
}
```

Since the B-block (being smaller) can be kept in cache, the next rows of the C-block can be computed without capacity misses in B

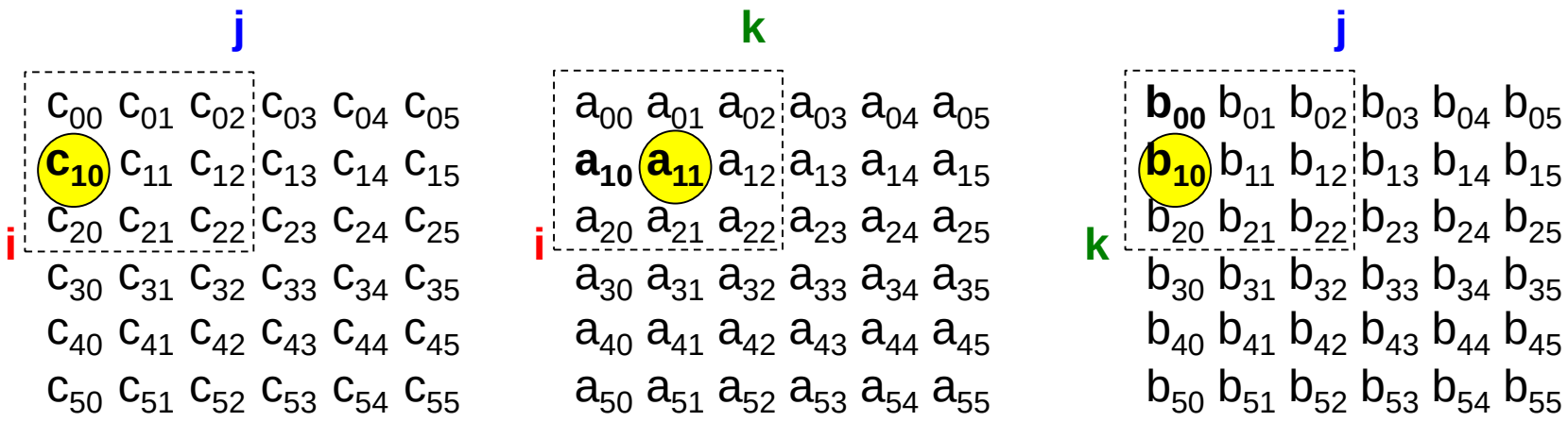
Block traversal (si=0, sj=0, sk=0)



i = 1 **j** = 0 **k = 0** **BLOCKSIZE = 3**

```
for (int i = si; i < si + BLOCKSIZE; ++i)
     for (int j = sj; j < sj + BLOCKSIZE; ++j)
       {
         double cij = C[i+j*n]; /* cij = C[i][j] */
         for(int k = sk; k < sk + BLOCKSIZE; k++ )
            cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
         C[i+j*n] = cij; /* C[i][j] = cij */
       }
}
```

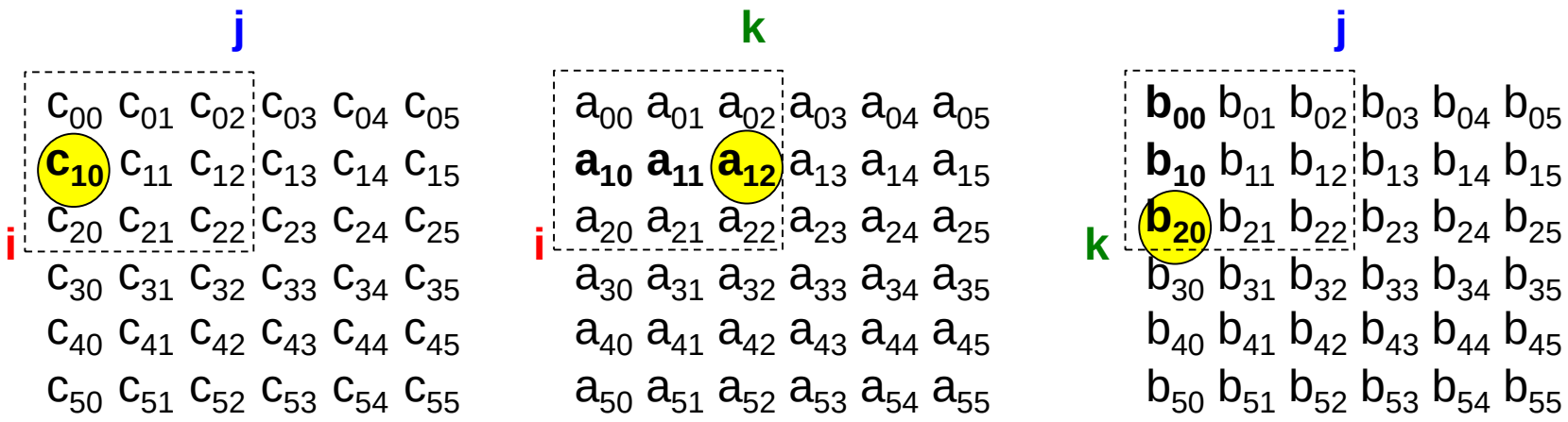
Block traversal (si=0, sj=0, sk=0)



i = 1 **j** = 0 **k = 1** **BLOCKSIZE = 3**

```
for (int i = si; i < si + BLOCKSIZE; ++i)
     for (int j = sj; j < sj + BLOCKSIZE; ++j)
       {
         double cij = C[i+j*n]; /* cij = C[i][j] */
         for(int k = sk; k < sk + BLOCKSIZE; k++ )
            cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
         C[i+j*n] = cij; /* C[i][j] = cij */
       }
}
```

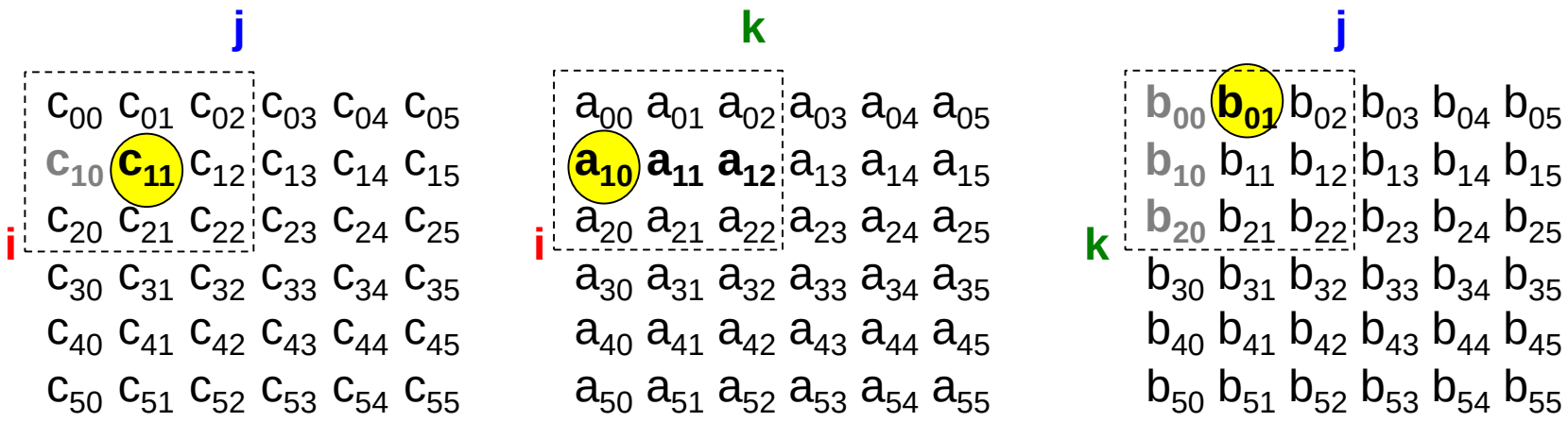
Block traversal (si=0, sj=0, sk=0)



i = 1 **j** = 0 **k = 2** **BLOCKSIZE = 3**

```
for (int i = si; i < si + BLOCKSIZE; ++i)
     for (int j = sj; j < sj + BLOCKSIZE; ++j)
       {
         double cij = C[i+j*n]; /* cij = C[i][j] */
         for(int k = sk; k < sk + BLOCKSIZE; k++ )
            cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
         C[i+j*n] = cij; /* C[i][j] = cij */
       }
}
```

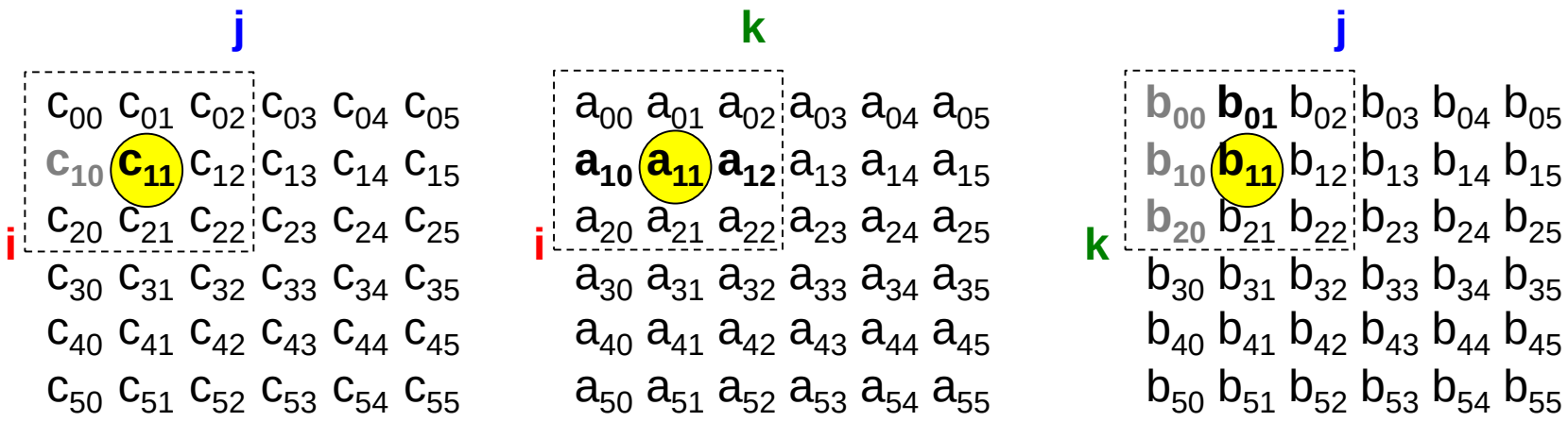
Block traversal (si=0, sj=0, sk=0)



i = 1 **j** = 1 **k** = 0 **BLOCKSIZE = 3**

```
for (int i = si; i < si + BLOCKSIZE; ++i)
     for (int j = sj; j < sj + BLOCKSIZE; ++j)
       {
         double cij = C[i+j*n]; /* cij = C[i][j] */
         for(int k = sk; k < sk + BLOCKSIZE; k++ )
            cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
         C[i+j*n] = cij; /* C[i][j] = cij */
       }
}
```

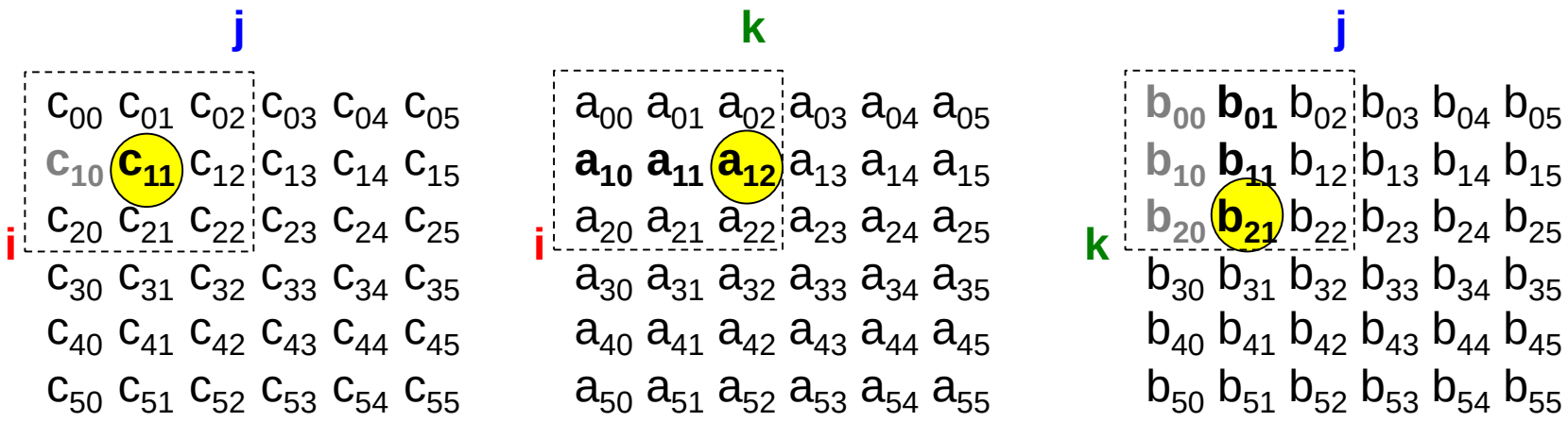
Block traversal (si=0, sj=0, sk=0)



i = 1 **j** = 1 **k** = 1 **BLOCKSIZE = 3**

```
for (int i = si; i < si + BLOCKSIZE; ++i)
     for (int j = sj; j < sj + BLOCKSIZE; ++j)
       {
         double cij = C[i+j*n]; /* cij = C[i][j] */
         for(int k = sk; k < sk + BLOCKSIZE; k++ )
            cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
         C[i+j*n] = cij; /* C[i][j] = cij */
       }
}
```

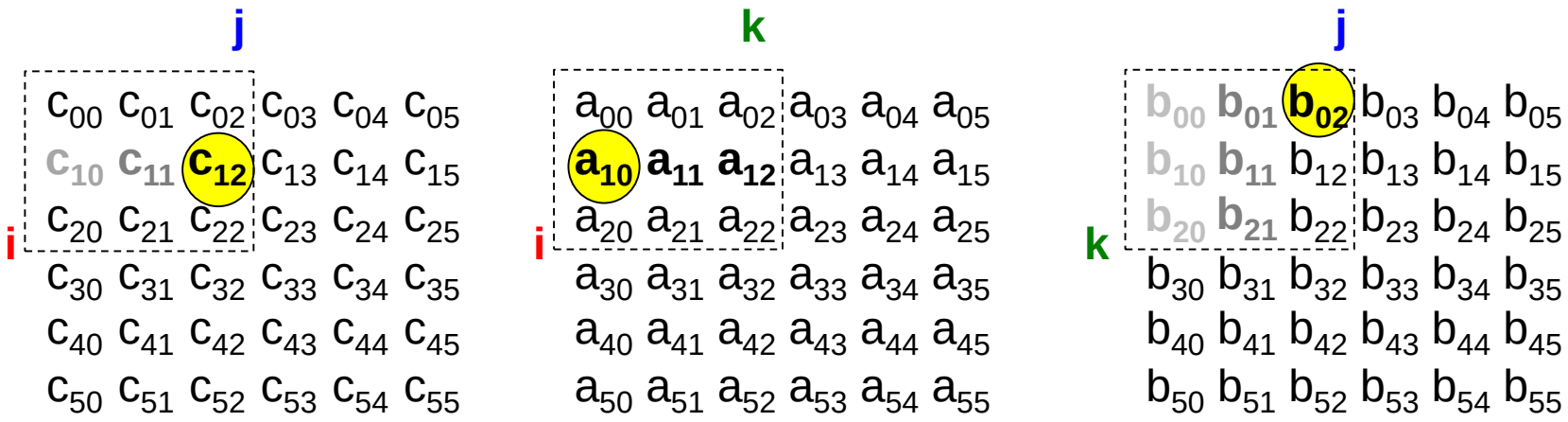
Block traversal (si=0, sj=0, sk=0)



i = 1 **j** = 1 **k** = 2 **BLOCKSIZE = 3**

```
for (int i = si; i < si + BLOCKSIZE; ++i)
     for (int j = sj; j < sj + BLOCKSIZE; ++j)
       {
         double cij = C[i+j*n]; /* cij = C[i][j] */
         for(int k = sk; k < sk + BLOCKSIZE; k++ )
            cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
         C[i+j*n] = cij; /* C[i][j] = cij */
       }
}
```

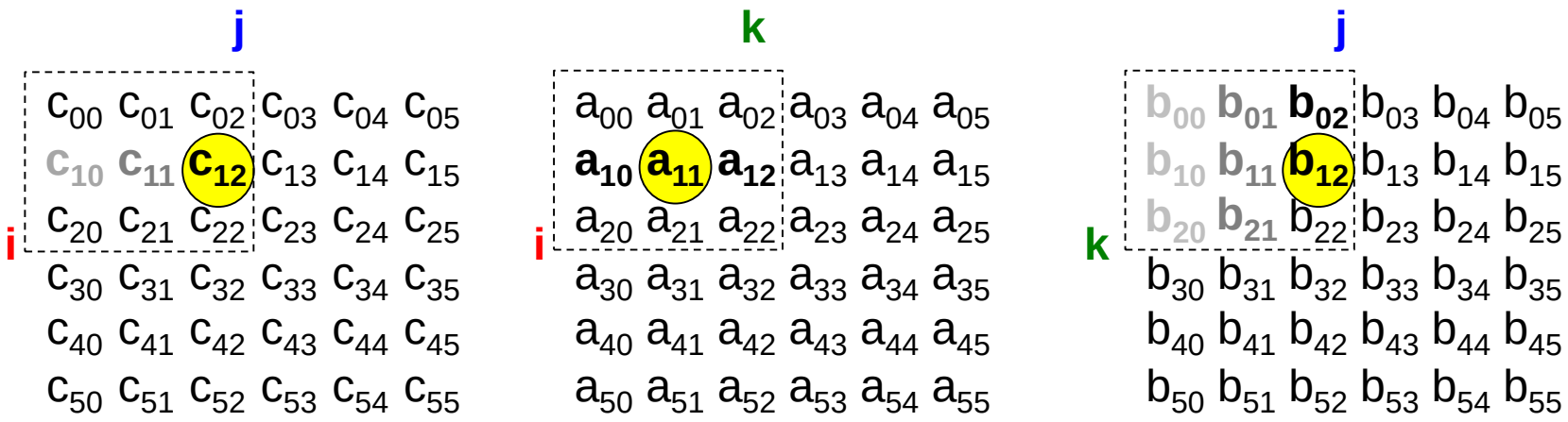
Block traversal (si=0, sj=0, sk=0)



i = 1 **j** = 2 **k** = 0 **BLOCKSIZE = 3**

```
for (int i = si; i < si + BLOCKSIZE; ++i)
     for (int j = sj; j < sj + BLOCKSIZE; ++j)
       {
         double cij = C[i+j*n]; /* cij = C[i][j] */
         for(int k = sk; k < sk + BLOCKSIZE; k++ )
            cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
         C[i+j*n] = cij; /* C[i][j] = cij */
       }
}
```

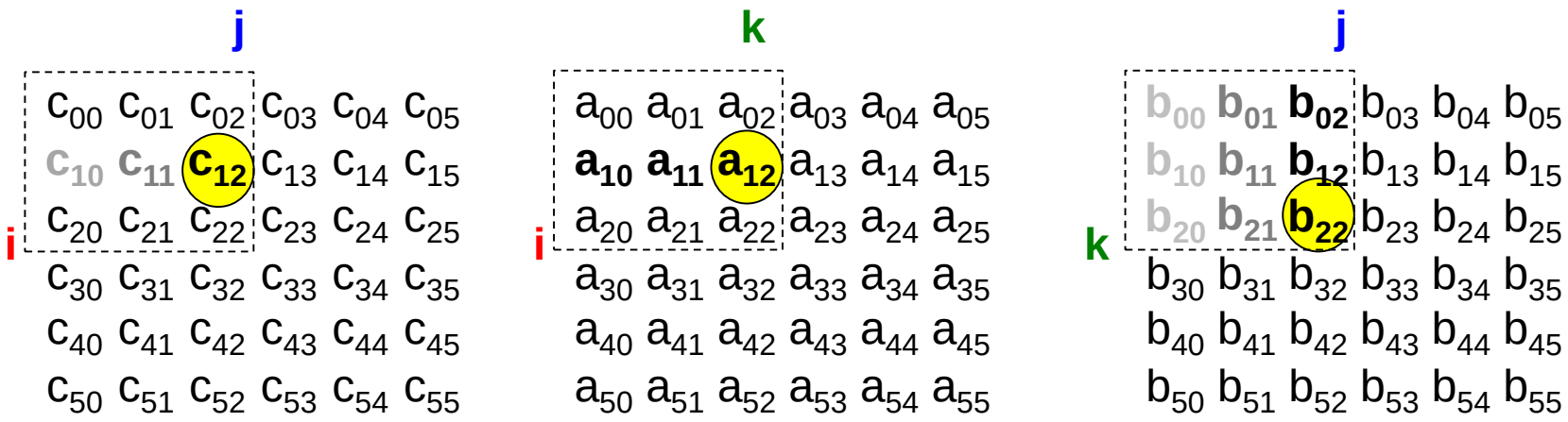
Block traversal (si=0, sj=0, sk=0)



i = 1 **j** = 2 **k** = 1 **BLOCKSIZE = 3**

```
for (int i = si; i < si + BLOCKSIZE; ++i)
     for (int j = sj; j < sj + BLOCKSIZE; ++j)
       {
         double cij = C[i+j*n]; /* cij = C[i][j] */
         for(int k = sk; k < sk + BLOCKSIZE; k++ )
            cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
         C[i+j*n] = cij; /* C[i][j] = cij */
       }
}
```

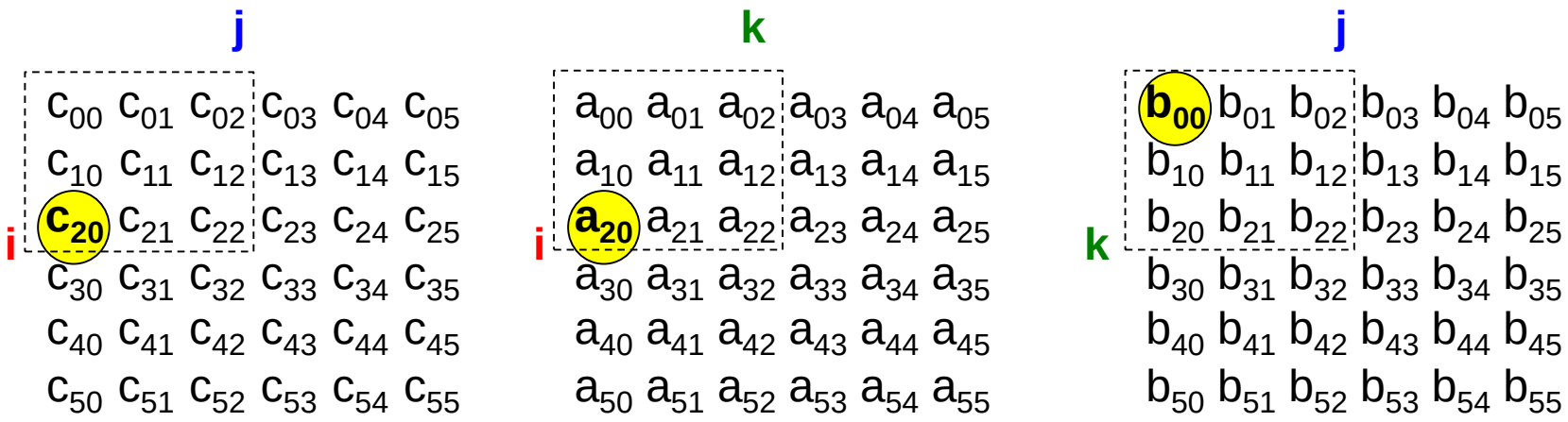
Block traversal (si=0, sj=0, sk=0)



i = 1 **j** = 2 **k = 2** **BLOCKSIZE = 3**

```
for (int i = si; i < si + BLOCKSIZE; ++i)
     for (int j = sj; j < sj + BLOCKSIZE; ++j)
       {
         double cij = C[i+j*n]; /* cij = C[i][j] */
         for(int k = sk; k < sk + BLOCKSIZE; k++ )
            cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
         C[i+j*n] = cij; /* C[i][j] = cij */
       }
}
```

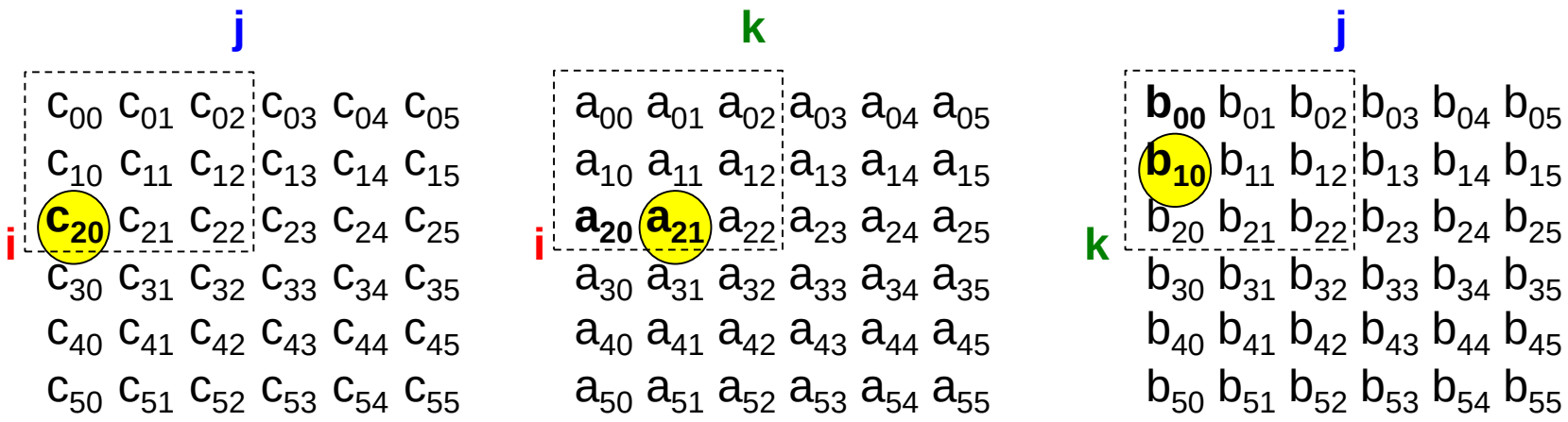
Block traversal (si=0, sj=0, sk=0)



i = 2 **j** = 0 **k = 0** **BLOCKSIZE = 3**

```
for (int i = si; i < si + BLOCKSIZE; ++i)
     for (int j = sj; j < sj + BLOCKSIZE; ++j)
       {
         double cij = C[i+j*n]; /* cij = C[i][j] */
         for(int k = sk; k < sk + BLOCKSIZE; k++ )
            cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
         C[i+j*n] = cij; /* C[i][j] = cij */
       }
}
```

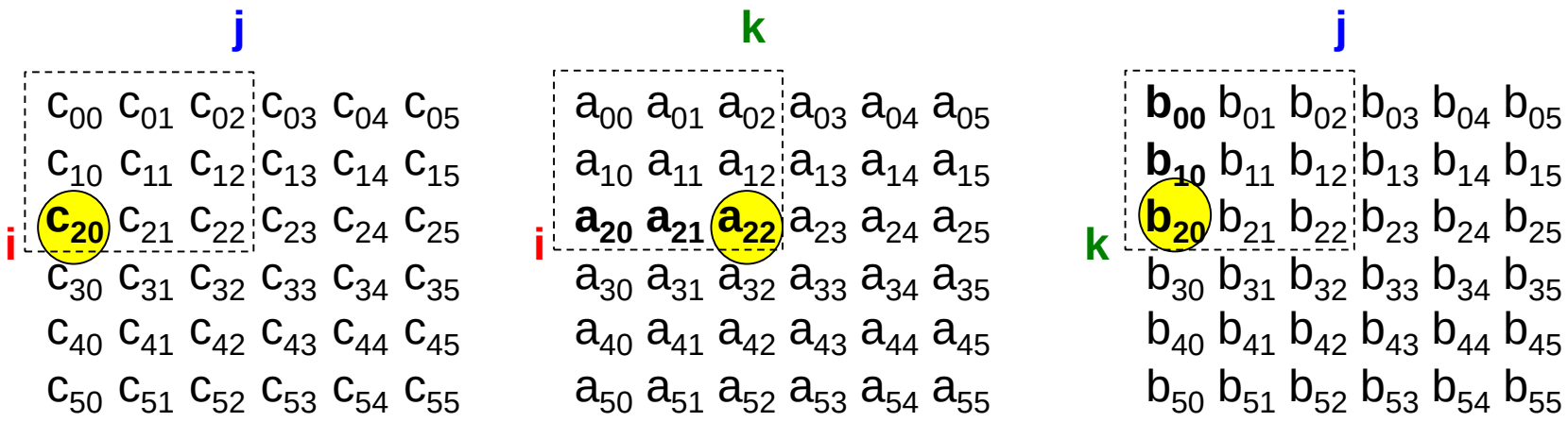
Block traversal (si=0, sj=0, sk=0)



i = 2 **j** = 0 **k = 1** **BLOCKSIZE = 3**

```
for (int i = si; i < si + BLOCKSIZE; ++i)
     for (int j = sj; j < sj + BLOCKSIZE; ++j)
       {
         double cij = C[i+j*n]; /* cij = C[i][j] */
         for(int k = sk; k < sk + BLOCKSIZE; k++ )
            cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
         C[i+j*n] = cij; /* C[i][j] = cij */
       }
}
```

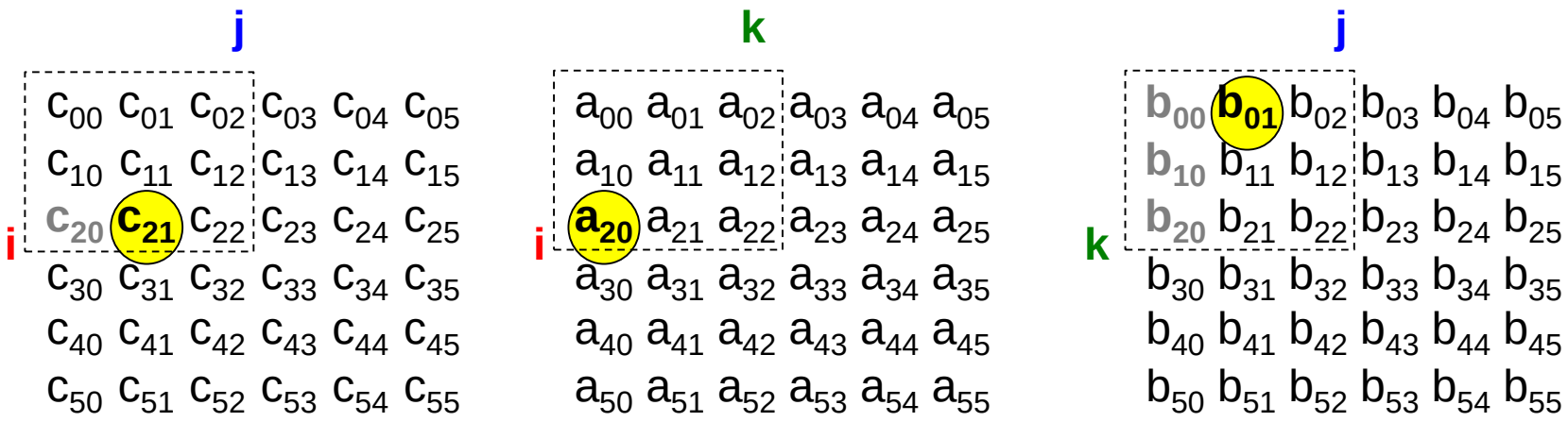
Block traversal (si=0, sj=0, sk=0)



i = 2 **j** = 0 **k = 2** **BLOCKSIZE = 3**

```
for (int i = si; i < si + BLOCKSIZE; ++i)
     for (int j = sj; j < sj + BLOCKSIZE; ++j)
       {
         double cij = C[i+j*n]; /* cij = C[i][j] */
         for(int k = sk; k < sk + BLOCKSIZE; k++ )
            cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
         C[i+j*n] = cij; /* C[i][j] = cij */
       }
}
```

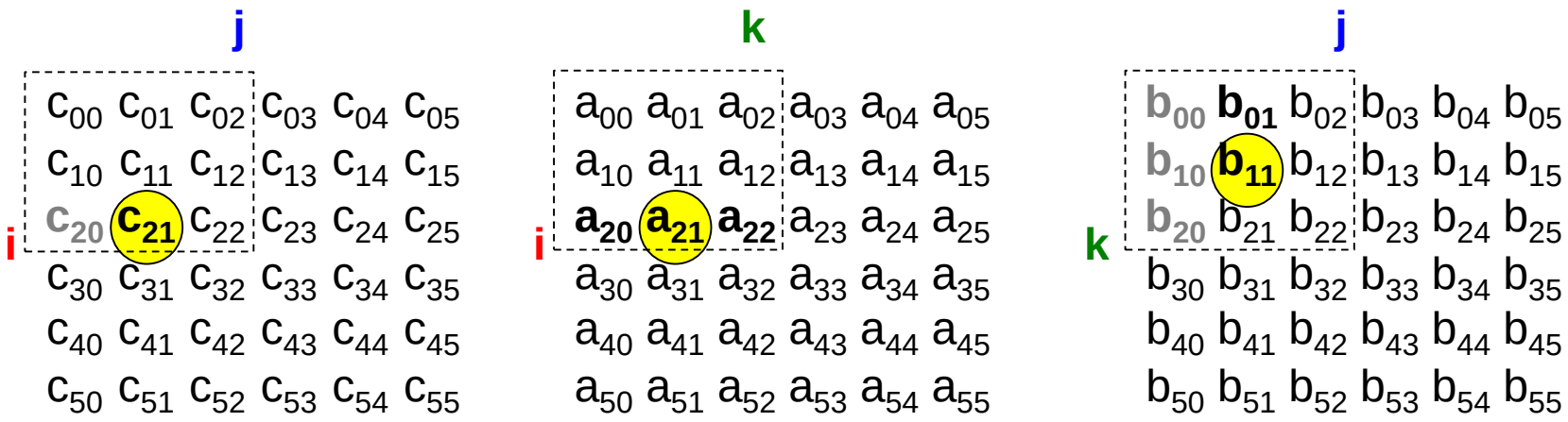
Block traversal (si=0, sj=0, sk=0)



i = 2 **j** = 1 **k** = 0 **BLOCKSIZE = 3**

```
for (int i = si; i < si + BLOCKSIZE; ++i)
     for (int j = sj; j < sj + BLOCKSIZE; ++j)
       {
         double cij = C[i+j*n]; /* cij = C[i][j] */
         for(int k = sk; k < sk + BLOCKSIZE; k++ )
            cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
         C[i+j*n] = cij; /* C[i][j] = cij */
       }
}
```

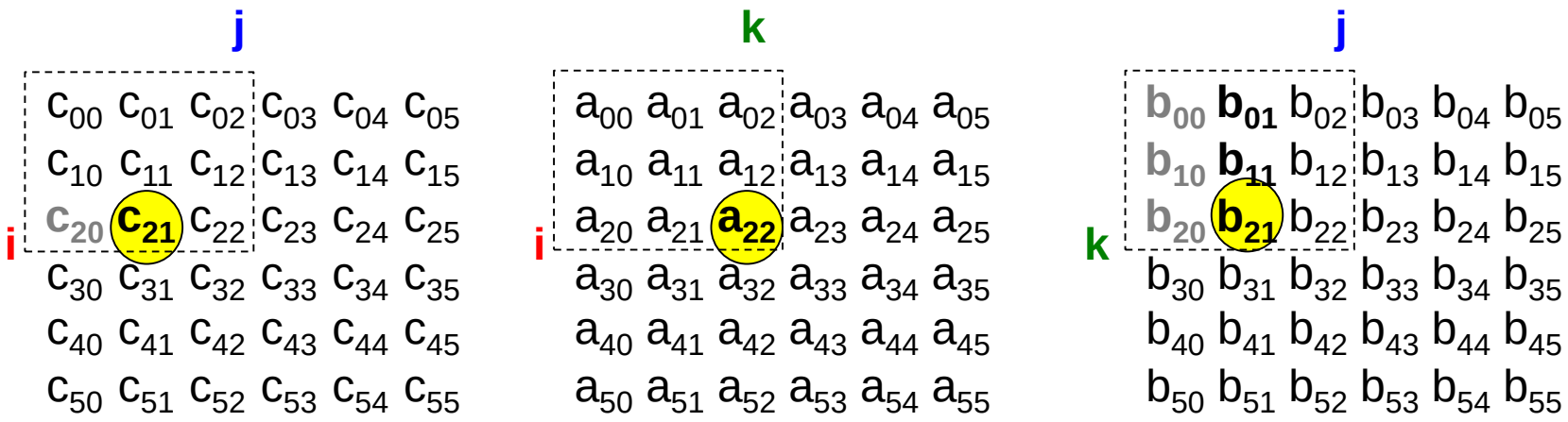
Block traversal (si=0, sj=0, sk=0)



i = 2 **j** = 1 **k** = 1 **BLOCKSIZE = 3**

```
for (int i = si; i < si + BLOCKSIZE; ++i)
     for (int j = sj; j < sj + BLOCKSIZE; ++j)
       {
         double cij = C[i+j*n]; /* cij = C[i][j] */
         for(int k = sk; k < sk + BLOCKSIZE; k++ )
            cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
         C[i+j*n] = cij; /* C[i][j] = cij */
       }
}
```

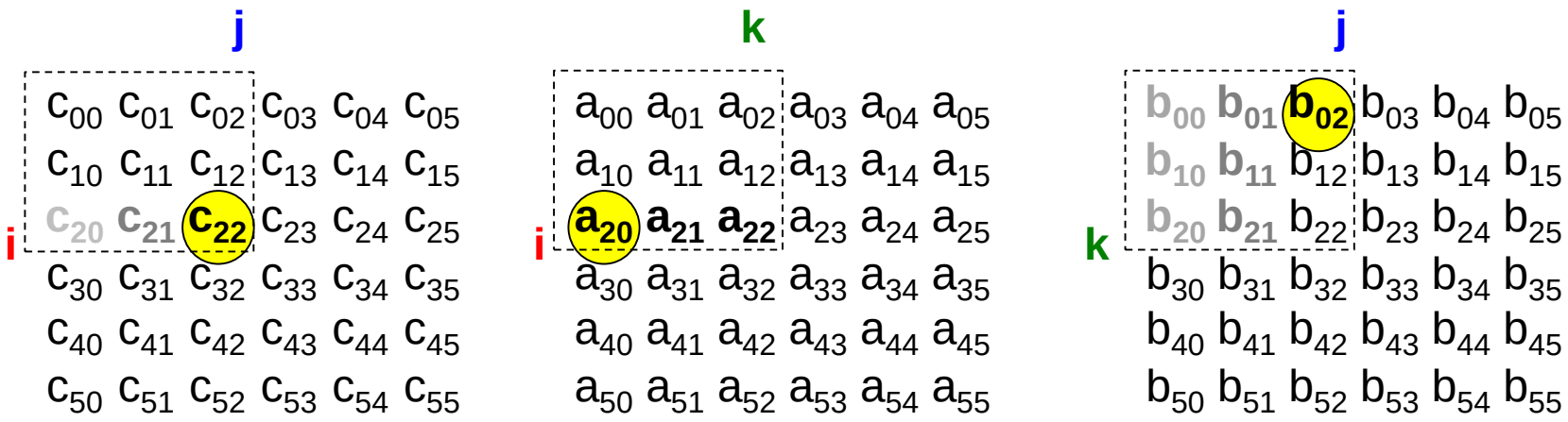
Block traversal (si=0, sj=0, sk=0)



i = 2 **j** = 1 **k** = 2 **BLOCKSIZE = 3**

```
for (int i = si; i < si + BLOCKSIZE; ++i)
     for (int j = sj; j < sj + BLOCKSIZE; ++j)
       {
         double cij = C[i+j*n]; /* cij = C[i][j] */
         for(int k = sk; k < sk + BLOCKSIZE; k++ )
            cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
         C[i+j*n] = cij; /* C[i][j] = cij */
       }
}
```

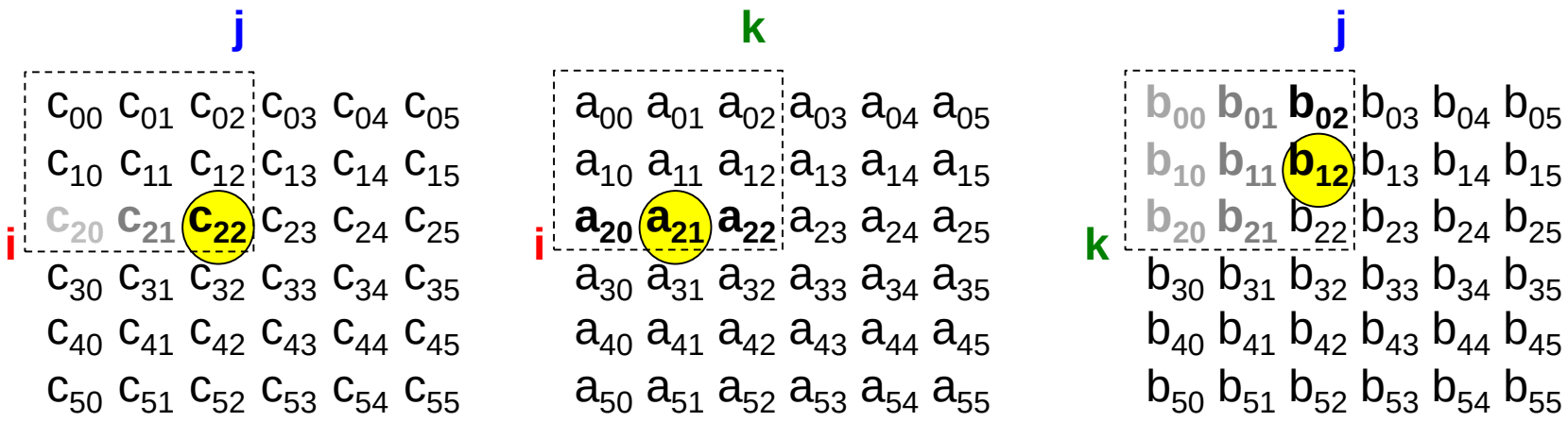
Block traversal (si=0, sj=0, sk=0)



i = 2 **j** = 2 **k = 0** **BLOCKSIZE = 3**

```
for (int i = si; i < si + BLOCKSIZE; ++i)
     for (int j = sj; j < sj + BLOCKSIZE; ++j)
       {
         double cij = C[i+j*n]; /* cij = C[i][j] */
         for(int k = sk; k < sk + BLOCKSIZE; k++ )
            cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
         C[i+j*n] = cij; /* C[i][j] = cij */
       }
}
```

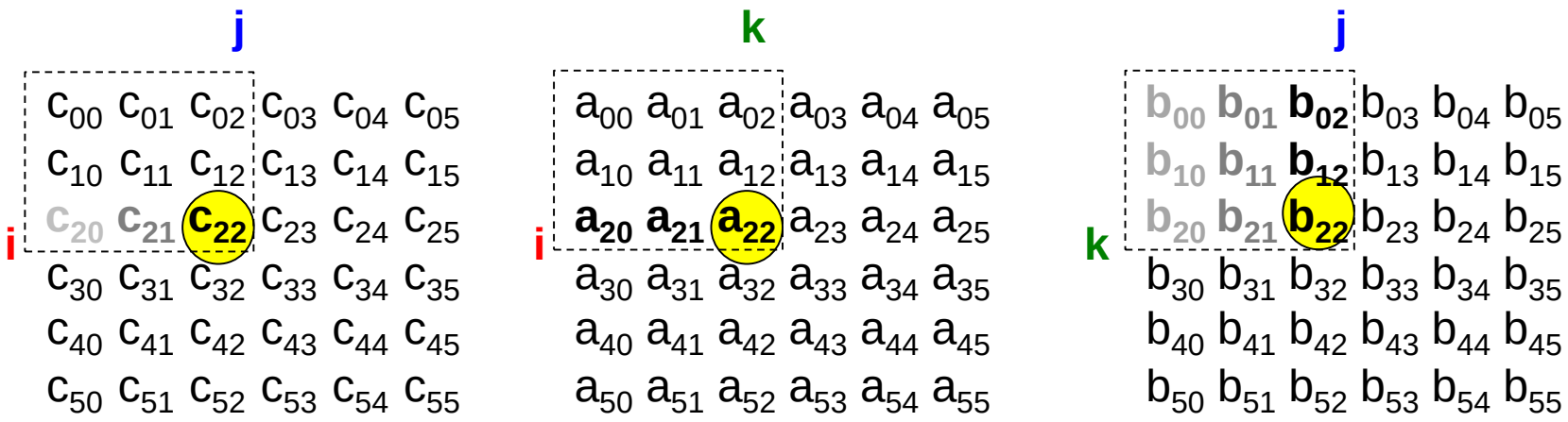
Block traversal (si=0, sj=0, sk=0)



i = 2 **j** = 2 **k** = 1 **BLOCKSIZE = 3**

```
for (int i = si; i < si + BLOCKSIZE; ++i)
     for (int j = sj; j < sj + BLOCKSIZE; ++j)
       {
         double cij = C[i+j*n]; /* cij = C[i][j] */
         for(int k = sk; k < sk + BLOCKSIZE; k++ )
            cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
         C[i+j*n] = cij; /* C[i][j] = cij */
       }
}
```

Block traversal (si=0, sj=0, sk=0)

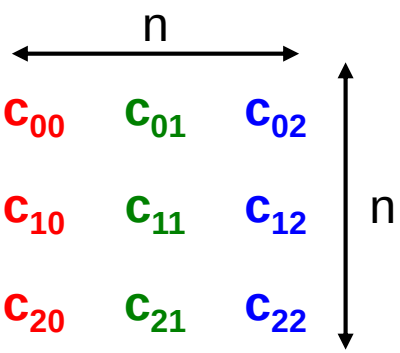


i = 2 **j** = 2 **k** = 2 **BLOCKSIZE = 3**

```
for (int i = si; i < si + BLOCKSIZE; ++i)
     for (int j = sj; j < sj + BLOCKSIZE; ++j)
       {
         double cij = C[i+j*n]; /* cij = C[i][j] */
         for(int k = sk; k < sk + BLOCKSIZE; k++ )
            cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
         C[i+j*n] = cij; /* C[i][j] = cij */
       }
}
```

Matrix: vectorial representation

- For higher performance:
  - Single-dimensional representation of a matrix
  - Column-major transformation



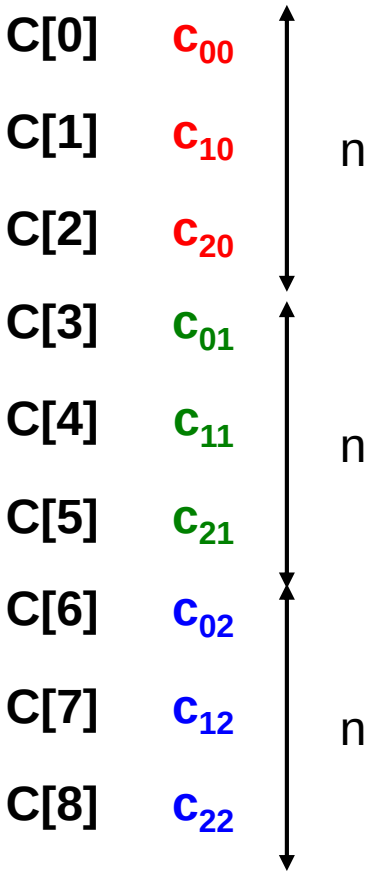
Address arithmetic

$c_{ij} = C[i+j*n]$

i, j in [0,n-1]

$c_{ij} = C[i+j*3]$

i, j in [0,2]

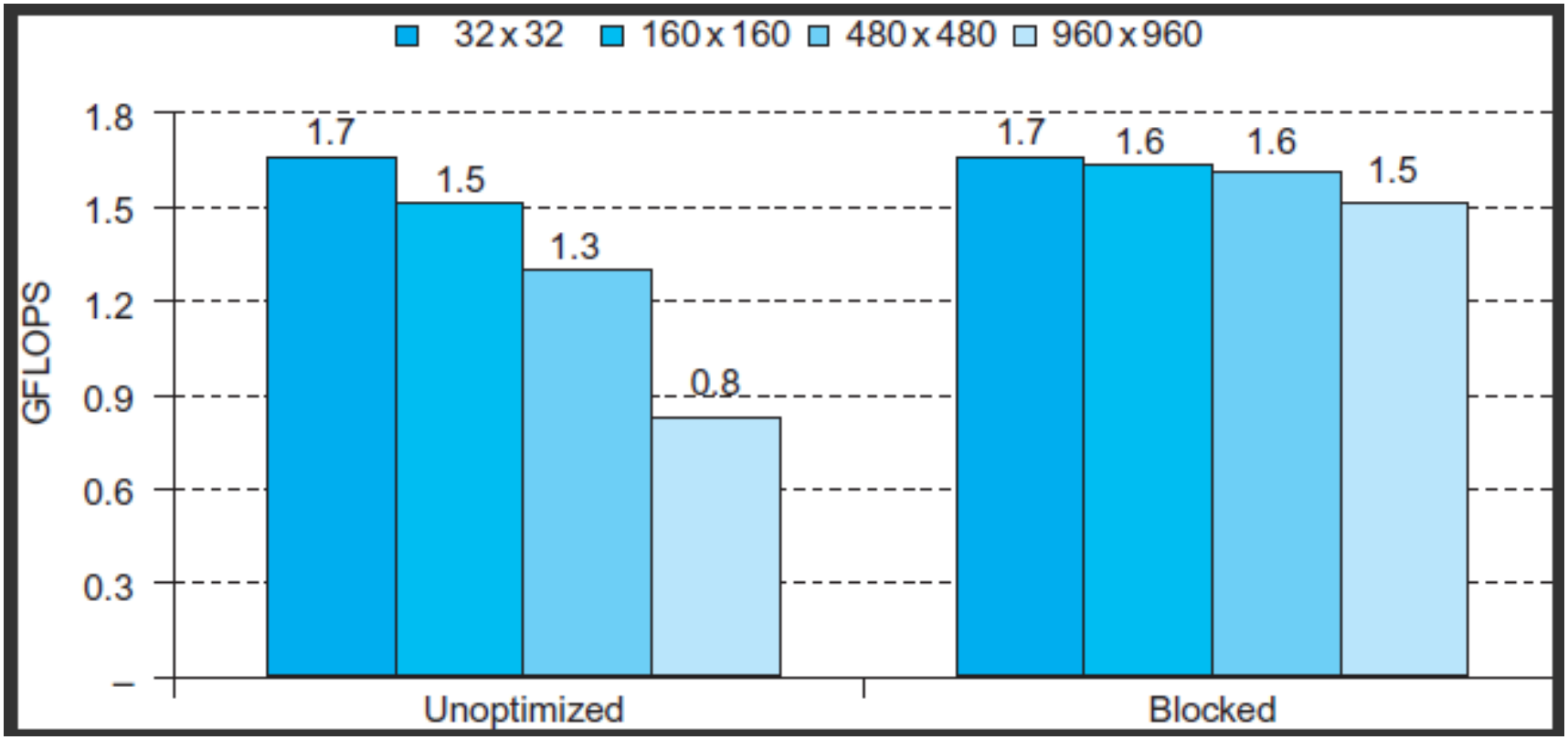


Example: c_{12} = C[1+2*3] = C[7]

Locality induced by variable k

- Column-major favor spatial locality when
  - Elements of a column sequentially accessed
- B is visited by column
  - Spatial locality is also exploited
- A is visited by row
  - Temporal locality is mostly exploited
- Blocking exploits a combination of
  - Spatial and temporal locality

Impact of blocking on DGEMM



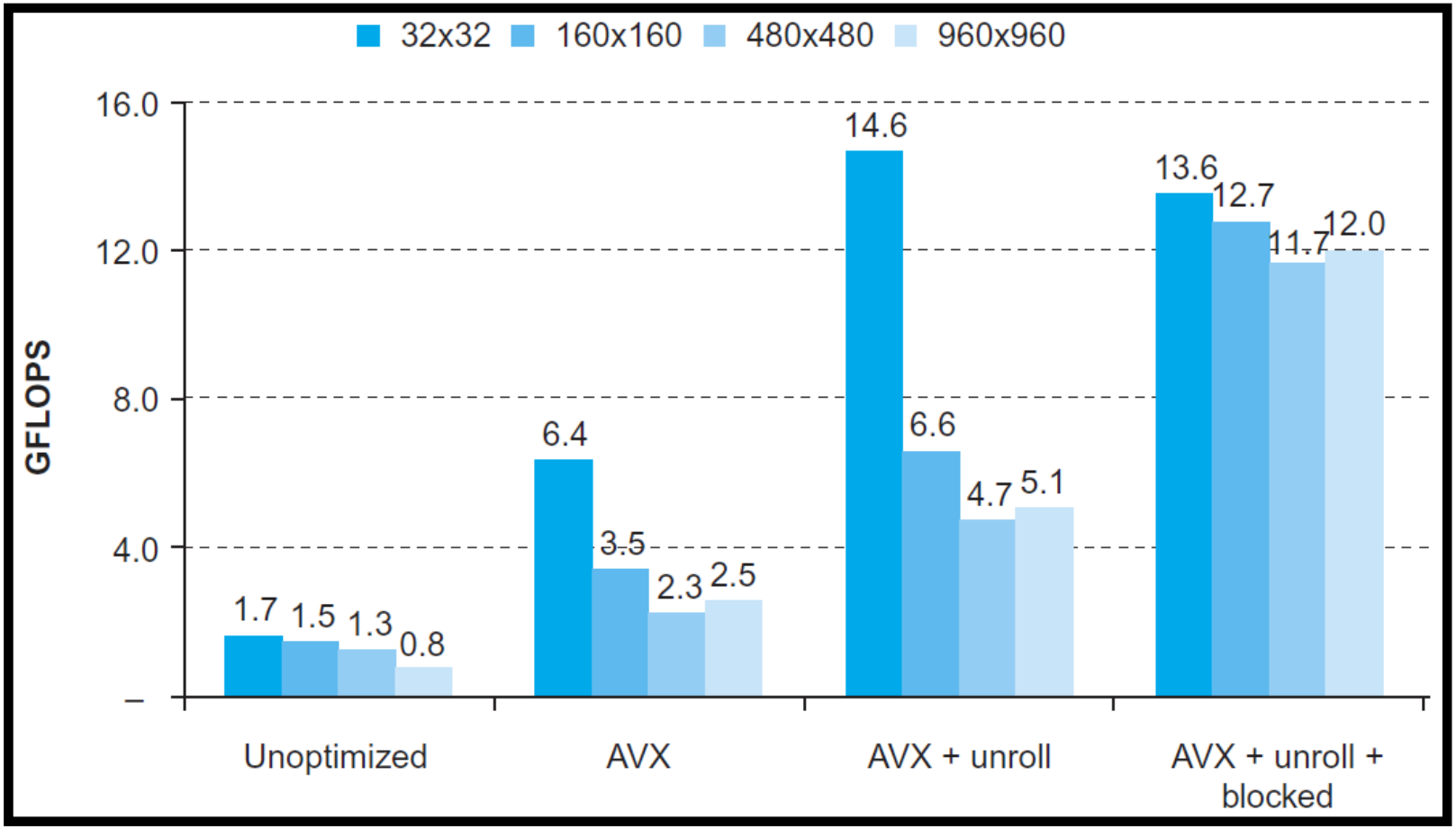
Unoptimized: Throughput for the largest matrix reduced by 50%.

Blocked: Throughput for the largest matrix reduced by 10%.

For 960x960 matrices, the use of blocking doubles the performance!

Slides by Luiz Santos (CCO/UFSC) with a few figures from MKP´s textbook

Blocking combined with parallelism



Parallelism only: Throughput for the largest matrix reduced by 65%.

Combination: Throughput for the largest matrix reduced by only 10%.

Blocking makes exploitation of parallelism less sensitive to cache misses!

Slides by Luiz Santos (CCO/UFSC) with a few figures from MKP´s textbook

Chapter 5

Cache-aware software optimization

(Slides prepared by Luiz Santos to complement the textbook)