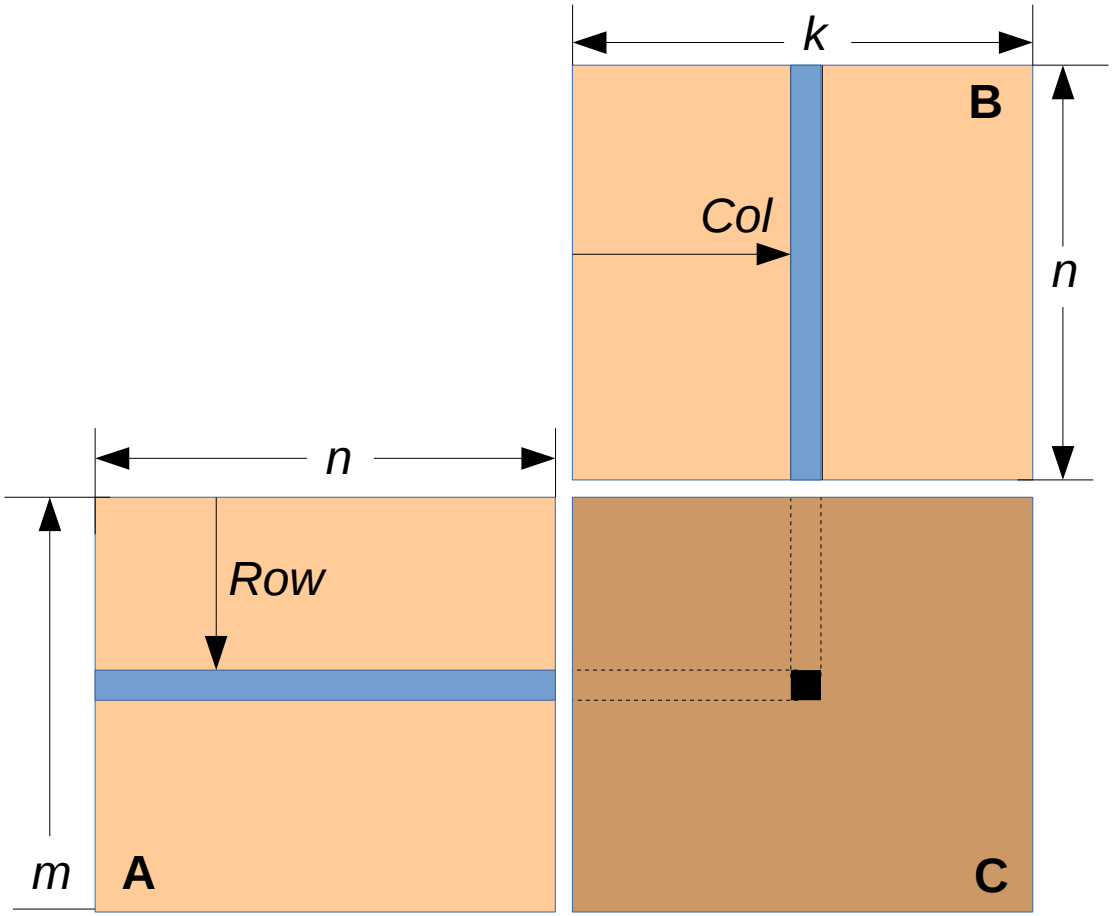
Matrix Multiplication

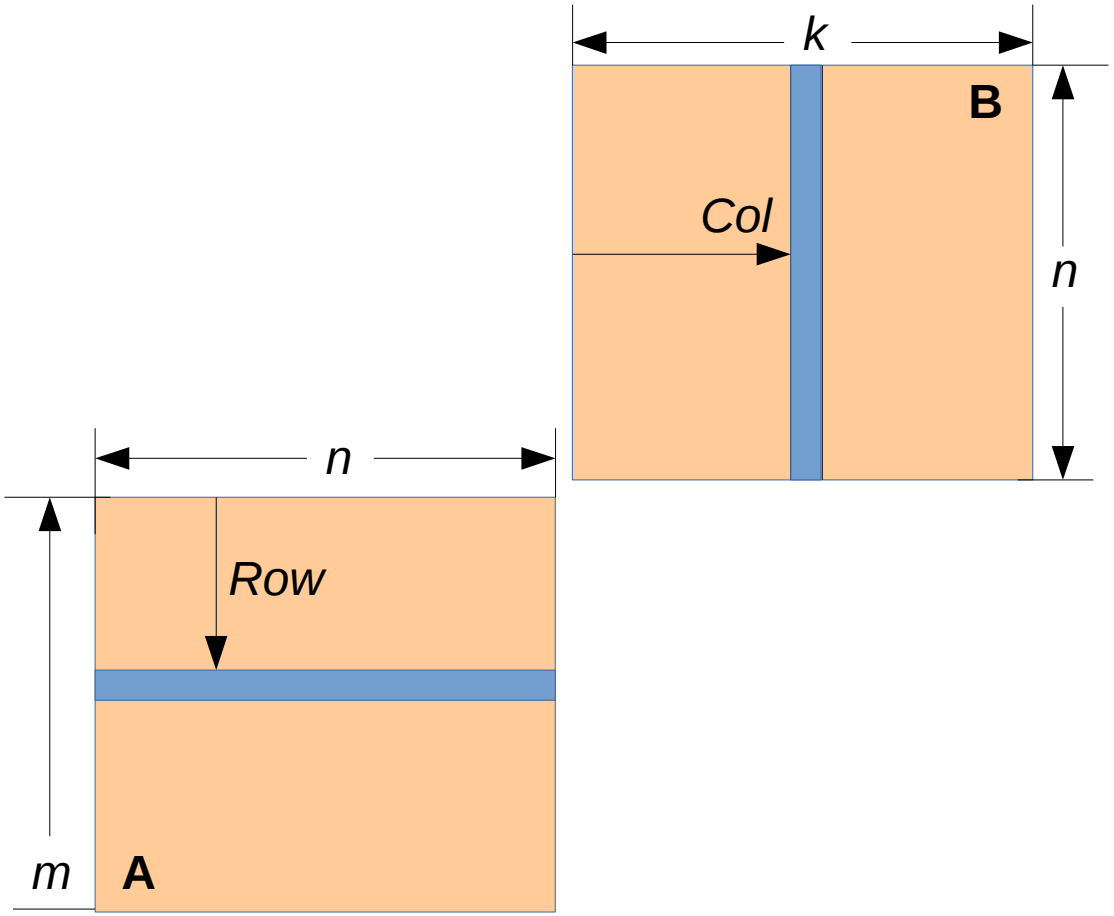
Matrix Multiplication – Outline

- C Implementation
- Basic CUDA MM kernel
- Work in each block

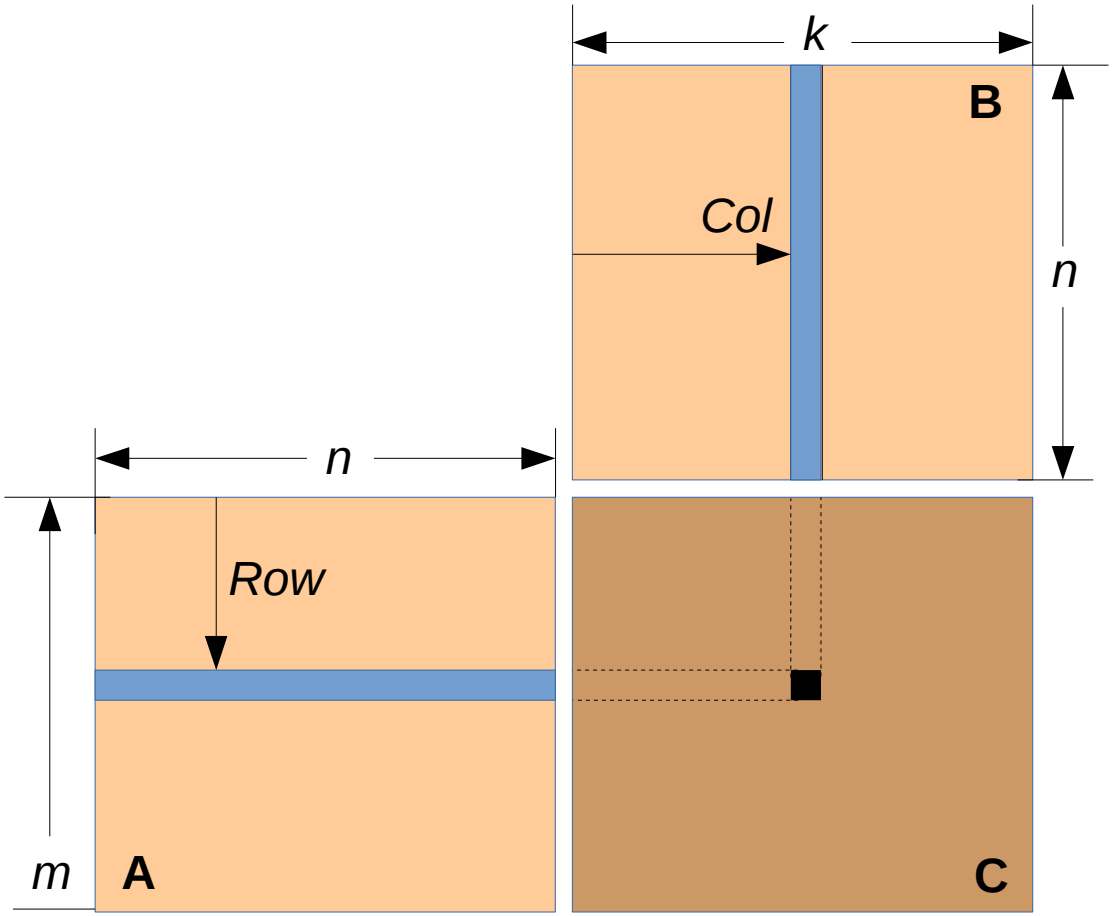
Matrix Multiplication



Matrix Multiplication

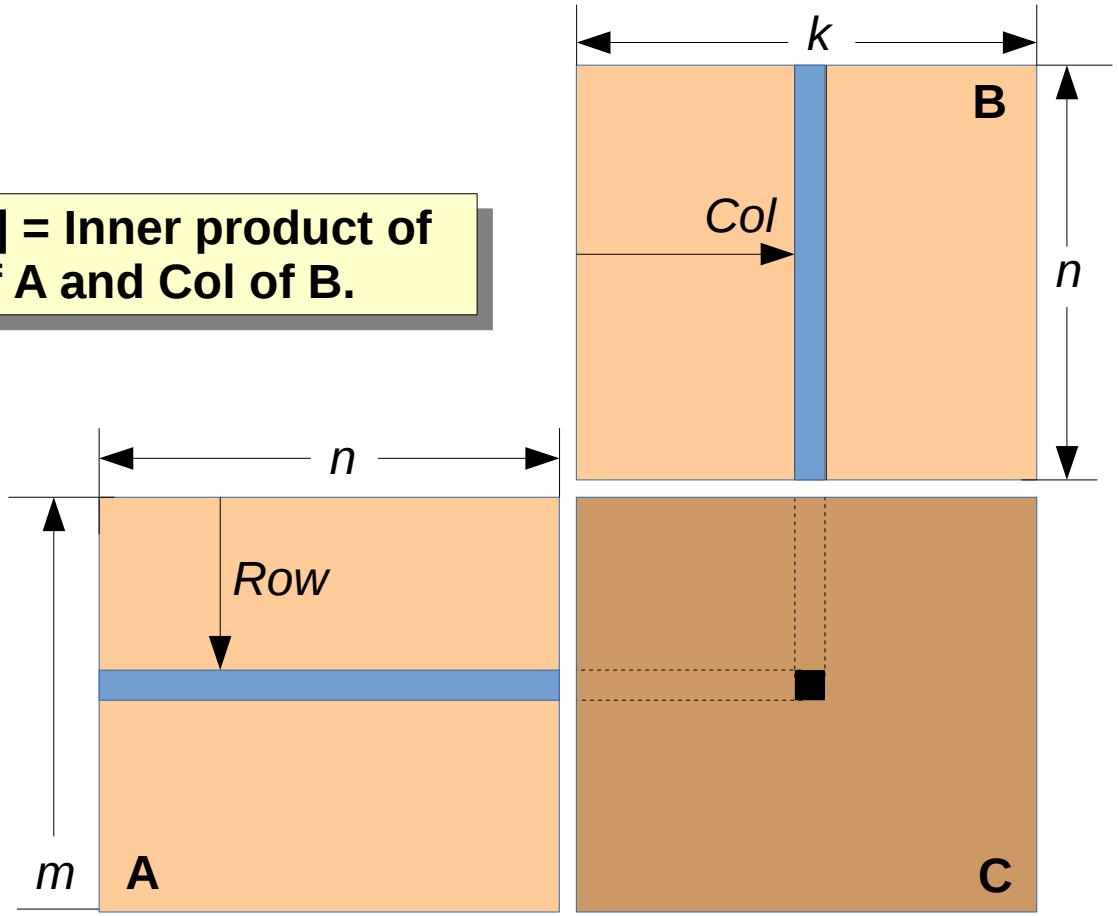


Matrix Multiplication



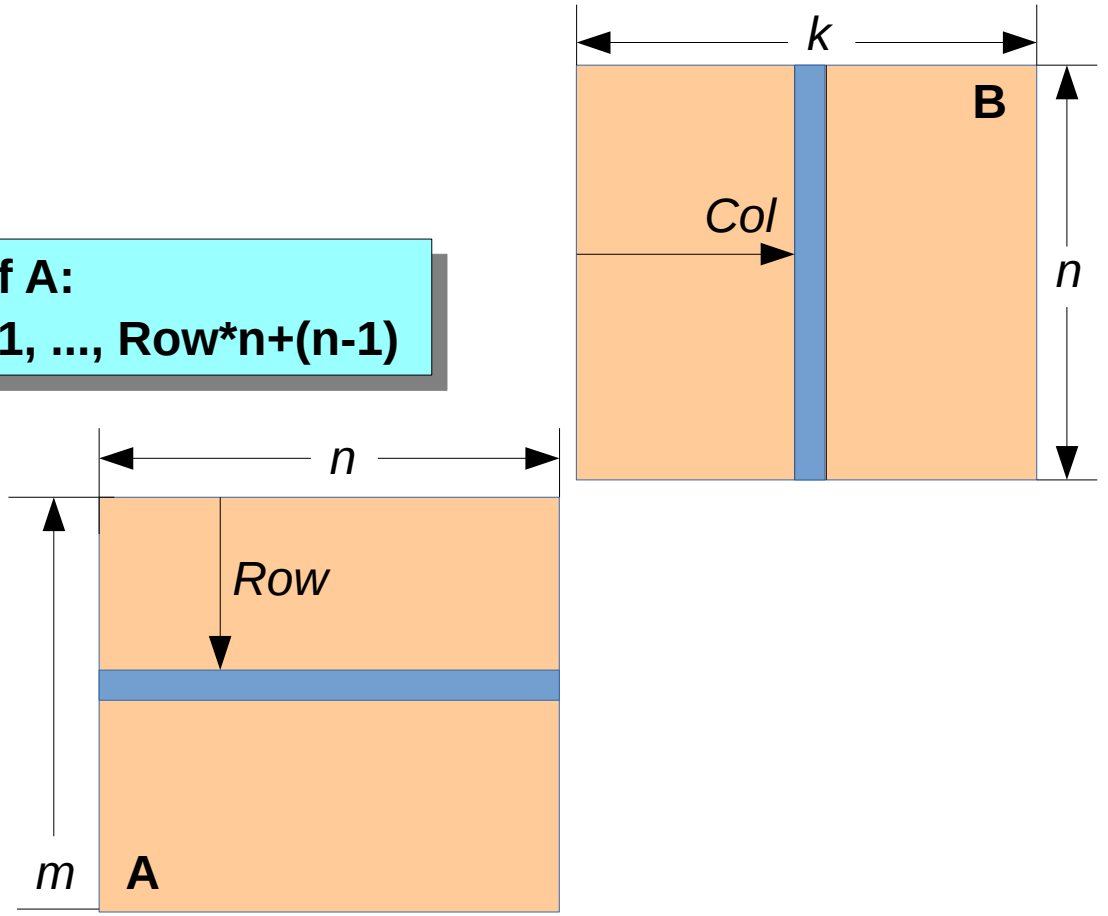
Matrix Multiplication

C[Row,Col] = Inner product of Row of A and Col of B.



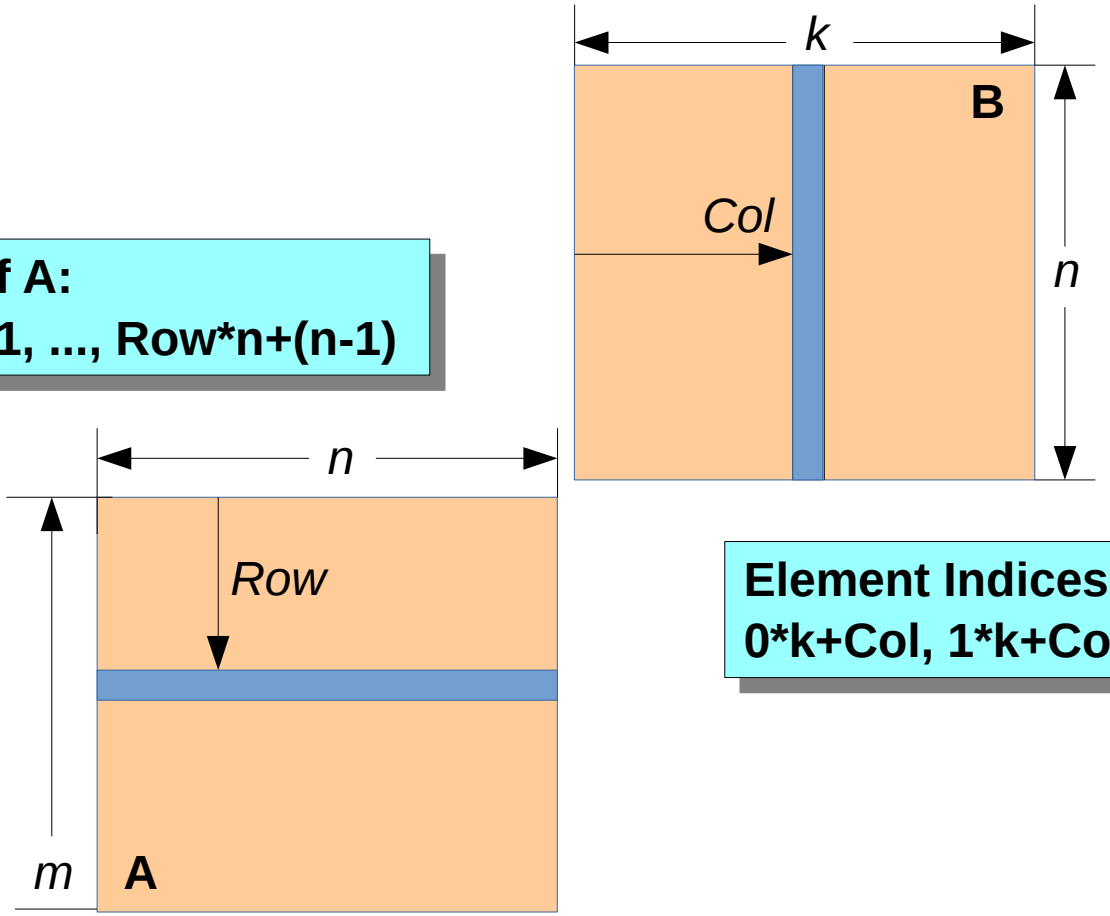
Matrix Multiplication

Element Indices of A:
Row*n+0, Row*n+1, ..., Row*n+(n-1)



Matrix Multiplication

Element Indices of A:
Row*n+0, Row*n+1, ..., Row*n+(n-1)



Element Indices of B:
0*k+Col, 1*k+Col, ..., (n-1)*k+Col

MM – Sequential Code in C

```
void MatrixMulOnHost(int m, int n, int k, float* A, float* B, float* C) {

    for (int Row = 0; Row < m; ++Row)
    for (int Col = 0; Col < k; ++Col) {
        float sum = 0;
        for (int i = 0; i < n; ++i) {
            float a = A[Row*n + i];
            float b = B[Col + i*k];
            sum += a * b;
        }
        C[Row*k + Col] = sum;
    }
}
```

MM – Work Distribution per Thread

MM – Work Distribution per Thread

- Each thread creates one element in the product matrix, *C*.

MM – Work Distribution per Thread

- Each thread creates one element in the product matrix, *C*.
- Every thread reads the corresponding *Row* and *Col* elements from input matrices *A* and *B*.

MM – Work Distribution per Thread

- Each thread creates one element in the product matrix, *C*.
- Every thread reads the corresponding *Row* and *Col* elements from input matrices *A* and *B*.
- A thread block could be square or rectangular in size
  - Depends on the output matrix size

Kernel Function – Example

$C_{0,0}$	$C_{0,1}$	$C_{0,2}$
$C_{1,0}$	$C_{1,1}$	$C_{1,2}$
$C_{2,0}$	$C_{2,1}$	$C_{2,2}$
$C_{3,0}$	$C_{3,1}$	$C_{3,2}$

Product Matrix

m=4, n=4, k=3

Kernel Function – Example

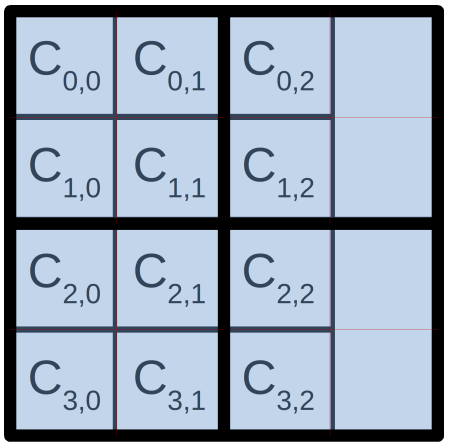
$C_{0,0}$	$C_{0,1}$	$C_{0,2}$
$C_{1,0}$	$C_{1,1}$	$C_{1,2}$
$C_{2,0}$	$C_{2,1}$	$C_{2,2}$
$C_{3,0}$	$C_{3,1}$	$C_{3,2}$

Product Matrix

m=4, n=4, k=3

TILE_WIDTH = 2
Each block = 2x2 = 4 threads

Kernel Function – Example

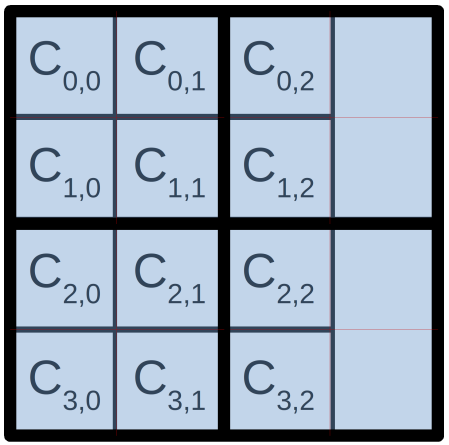


Product Matrix

m=4, n=4, k=3

TILE_WIDTH = 2
Each block = 2x2 = 4 threads

Kernel Function – Example



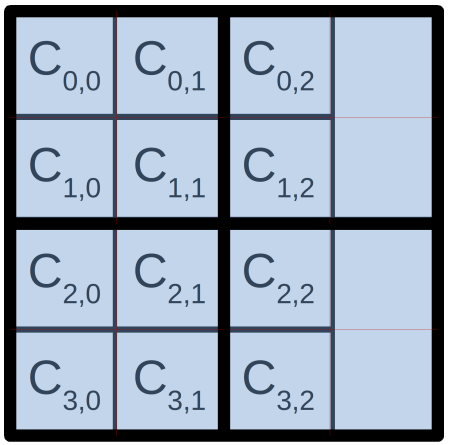
Product Matrix

m=4, n=4, k=3

TILE_WIDTH = 2
Each block = 2x2 = 4 threads

What is the grid size for a *m* x *k* product matrix?

Kernel Function – Example



Product Matrix

m=4, n=4, k=3

TILE_WIDTH = 2
Each block = 2x2 = 4 threads

What is the grid size for a *m* x *k* product matrix?

$$\left(\frac{(k-1)}{TILEWIDTH}+1, \frac{(m-1)}{TILEWIDTH}+1, 1\right)$$

Kernel Invocation

```
// Setup the execution configuration
// TILE_WIDTH is a #define constant

dim3 dimGrid((k-1)/TILE_WIDTH+1, (m-1)/TILE_WIDTH+1, 1);
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);

// Launch the device computation threads!

MatrixMulKernel<<<dimGrid, dimBlock>>>(m, n, k, d_A, d_B, d_C);
```

A Simple Matrix Multiplication Kernel

```
__global__
void MatrixMulKernel(int m, int n, int k, float* A, float* B,
float* C)
{
  ...
}
```

A Simple Matrix Multiplication Kernel

```
__global__
void MatrixMulKernel(int m, int n, int k, float* A, float* B,
float* C)
{
  // Row and Col variables for this thread?

  // If Row and Column are valid
    // perform the inner product of A[Row] elements
    // with B[Col] elements
}
```

A Simple Matrix Multiplication Kernel

```
__global__
void MatrixMulKernel(int m, int n, int k, float* A, float* B,
float* C)
{
    int Row = blockIdx.y*blockDim.y+threadIdx.y;
    int Col = blockIdx.x*blockDim.x+threadIdx.x;

  // If Row and Column are valid
    // perform the inner product of A[Row] elements
    // with B[Col] elements

}
```

A Simple Matrix Multiplication Kernel

```
__global__
void MatrixMulKernel(int m, int n, int k, float* A, float* B,
float* C)
{
    int Row = blockIdx.y*blockDim.y+threadIdx.y;
    int Col = blockIdx.x*blockDim.x+threadIdx.x;

    if ((Row < m) && (Col < k)) {

     // perform the inner product of A[Row] elements
     // with B[Col] elements
    }
}
```

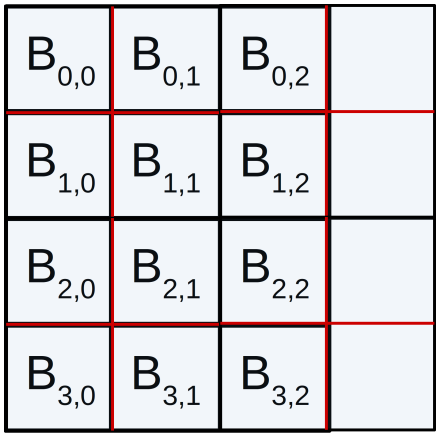
A Simple Matrix Multiplication Kernel

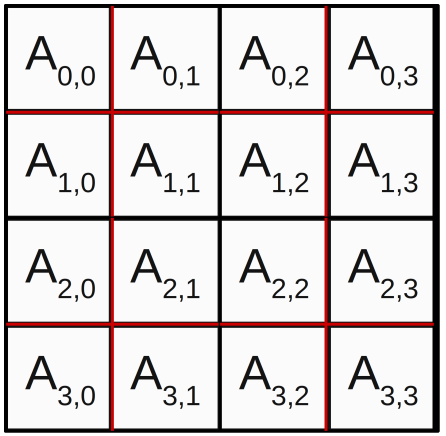
```
__global__
void MatrixMulKernel(int m, int n, int k, float* A, float* B,
float* C)
{
    int Row = blockIdx.y*blockDim.y+threadIdx.y;
    int Col = blockIdx.x*blockDim.x+threadIdx.x;
    if ((Row < m) && (Col < k)) {
        float Cvalue = 0.0;
        for (int i = 0; i < n; ++i)
            Cvalue += A[Row*n+i] * B[Col+i*k];
        C[Row*k+Col] = Cvalue;
    }
}
```

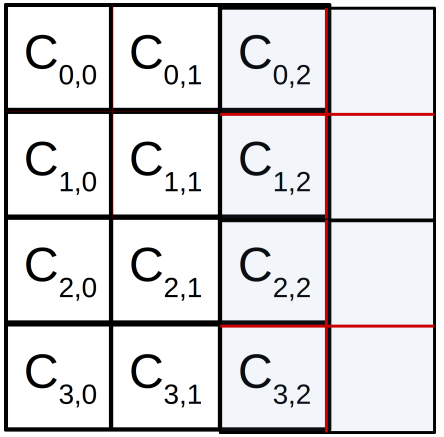
Work in Block(0,0)

- Row =
- Col =

m=4, n=4, k=3



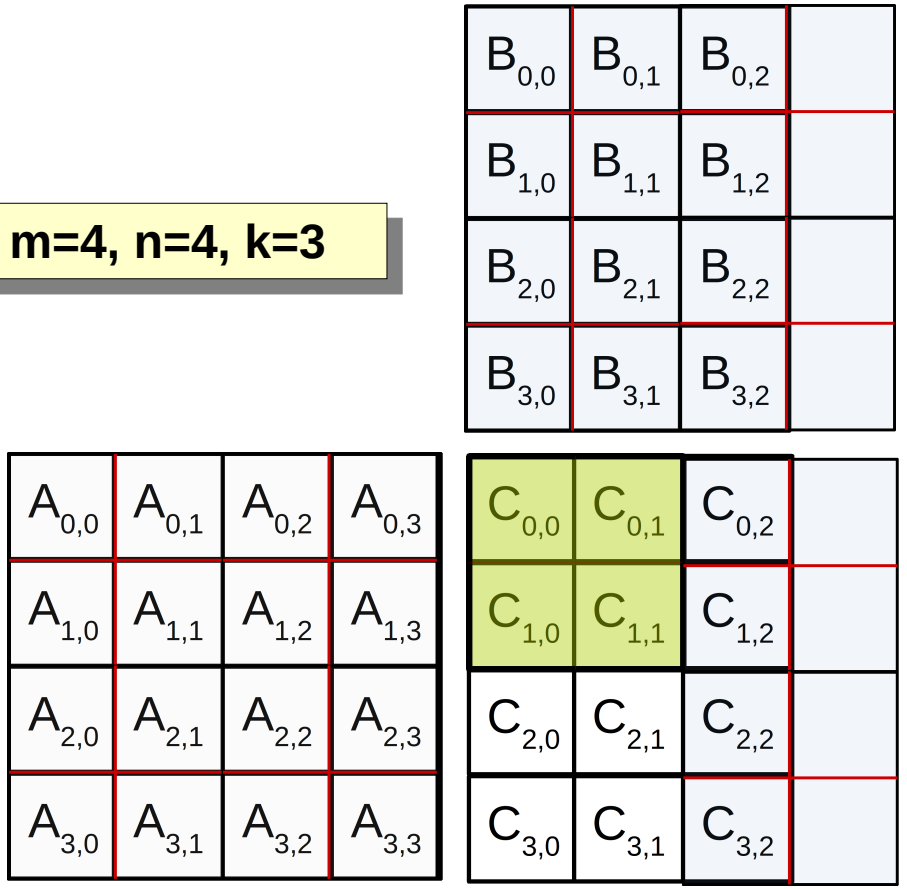




Work in Block(0,0)

- Row =
- Col =

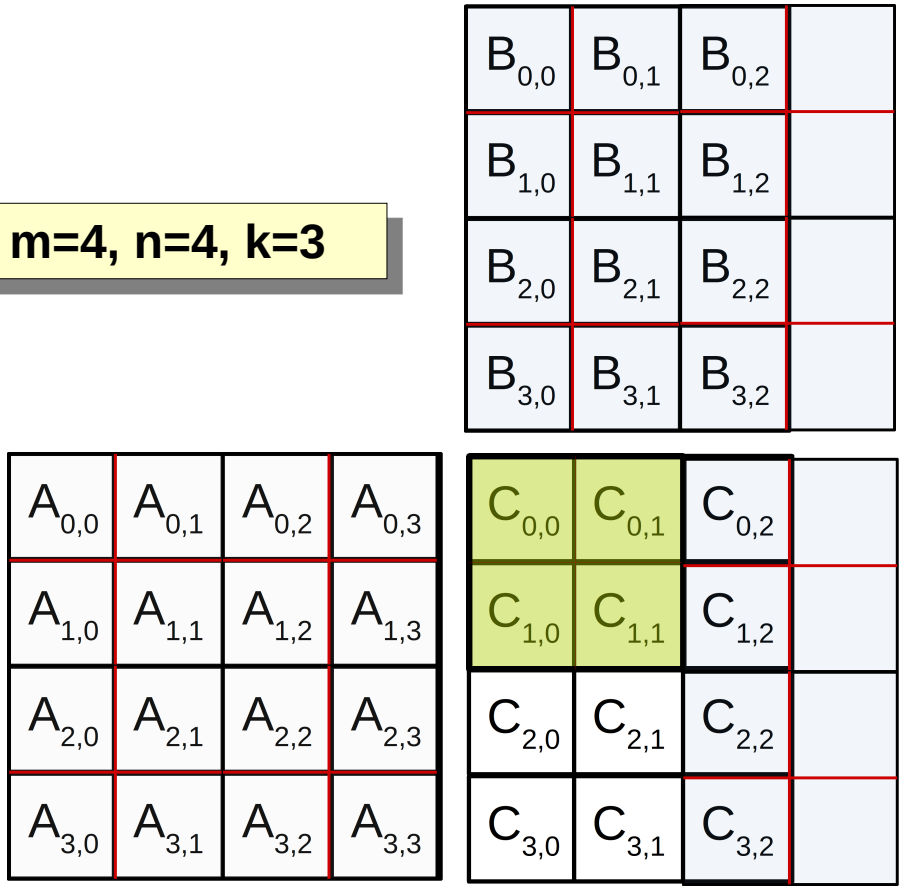
m=4, n=4, k=3



Work in Block(0,0)

- Row = 0 * 2 + threadIdx.y
- Col = 0 * 2 + threadIdx.x

m=4, n=4, k=3

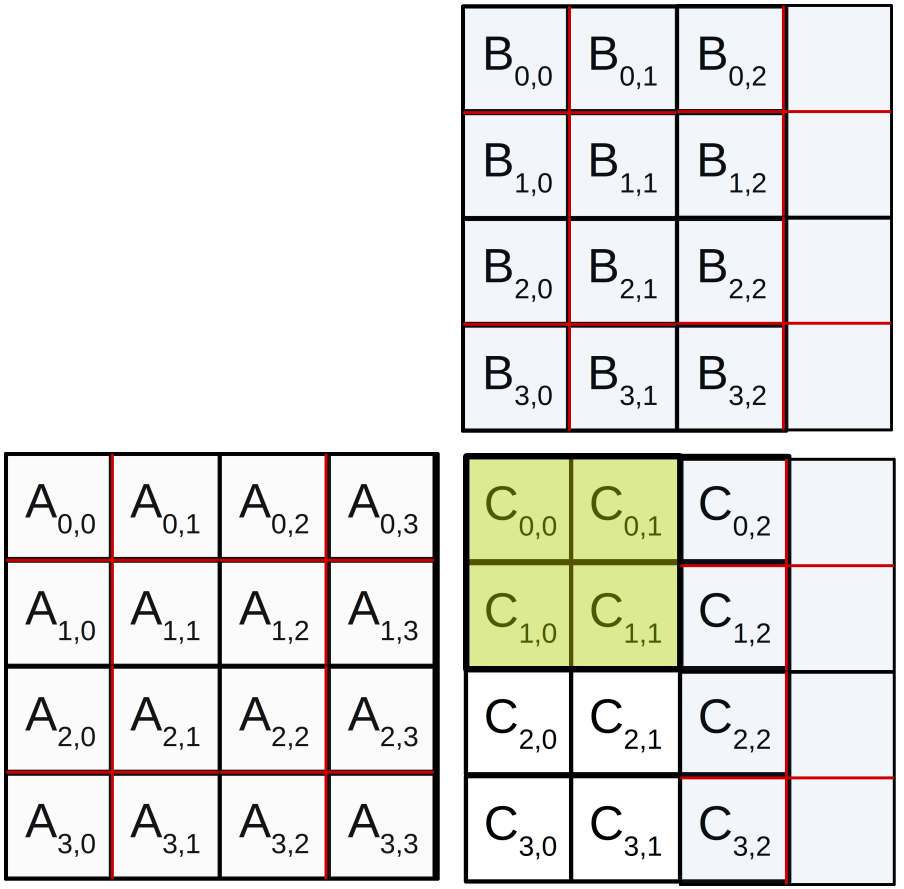


Work in Block(0,0)

- Row = 0 * 2 + threadIdx.y
- Col = 0 * 2 + threadIdx.x

threadIdx.x = {0,1}

threadIdx.y = {0,1}

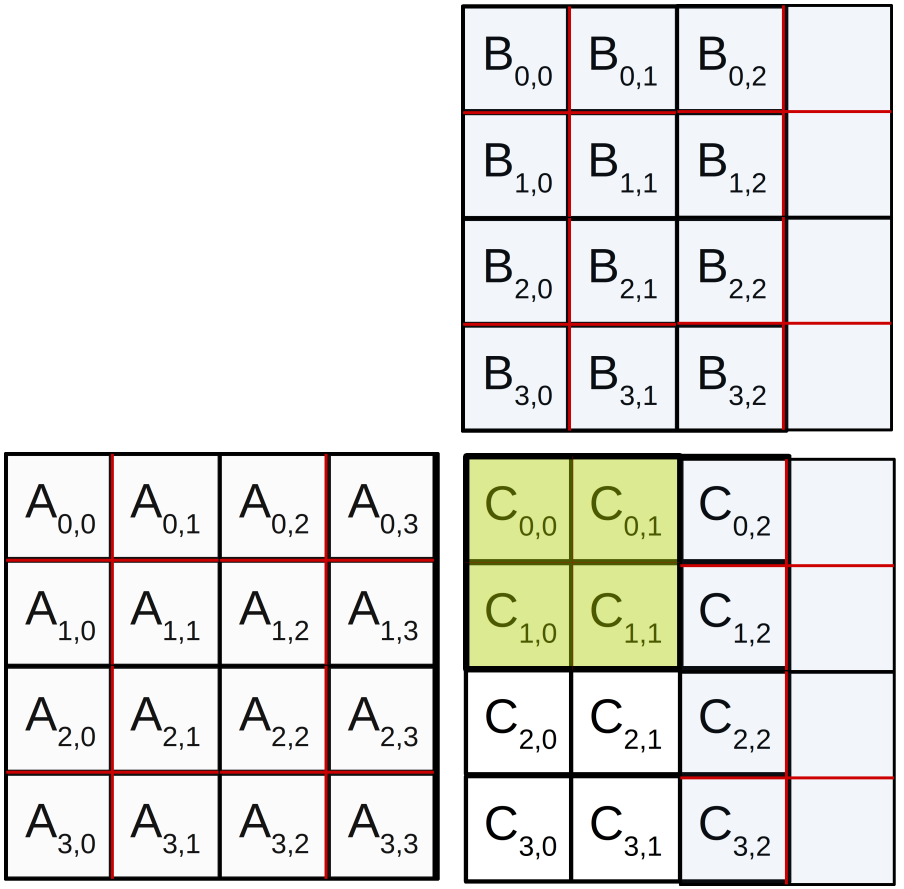


Work in Block(0,0)

- Row = 0 * 2 + threadIdx.y
- Col = 0 * 2 + threadIdx.x

threadIdx.x = 0

threadIdx.y = 0



Work in Block(0,0)

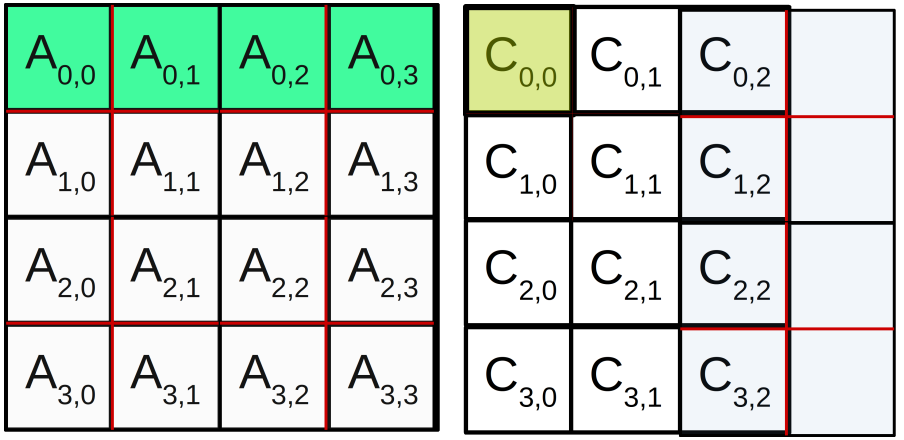
- Row = 0 * 2 + threadIdx.y
- Col = 0 * 2 + threadIdx.x

threadIdx.x = 0

threadIdx.y = 0

Row = 0

Col = 0

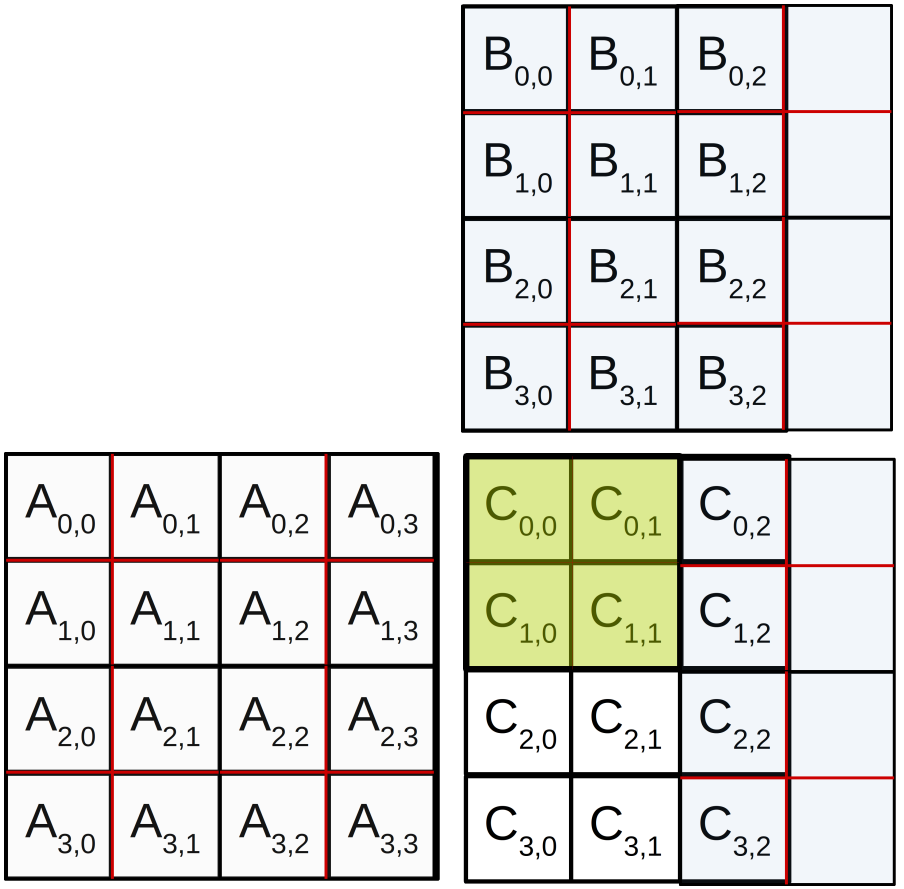


Work in Block(0,0)

- Row = 0 * 2 + threadIdx.y
- Col = 0 * 2 + threadIdx.x

threadIdx.x = 1

threadIdx.y = 0



Work in Block(0,0)

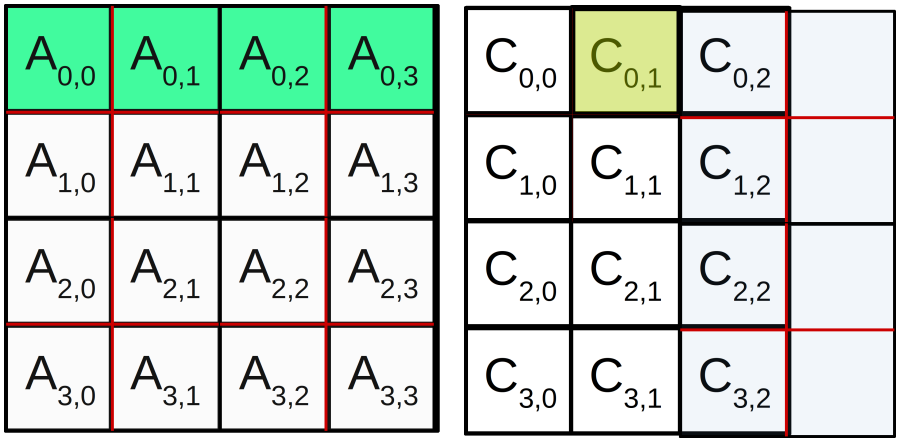
- Row = 0 * 2 + threadIdx.y
- Col = 0 * 2 + threadIdx.x

threadIdx.x = 1

threadIdx.y = 0

Row = 0

Col = 1

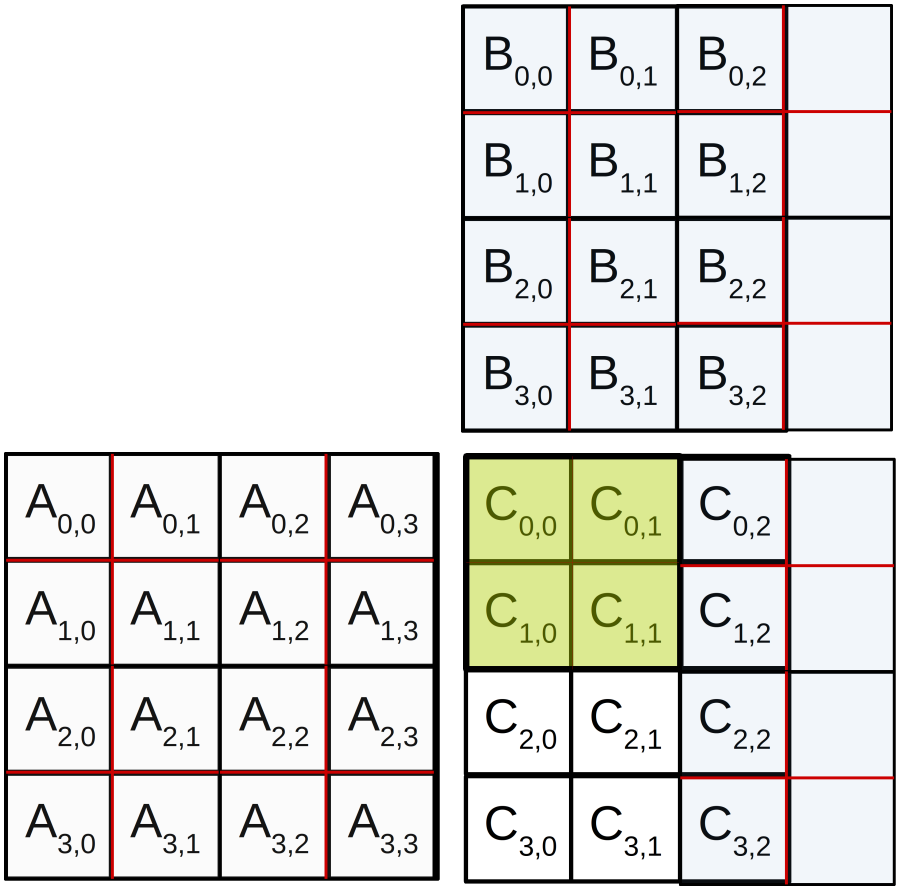


Work in Block(0,0)

- Row = 0 * 2 + threadIdx.y
- Col = 0 * 2 + threadIdx.x

threadIdx.x = 1

threadIdx.y = 1



Work in Block(0,0)

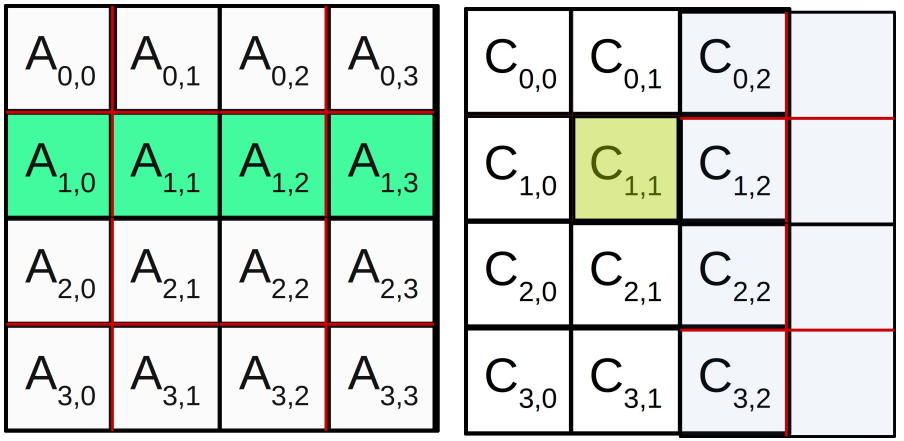
- Row = 0 * 2 + threadIdx.y
- Col = 0 * 2 + threadIdx.x

threadIdx.x = 1

threadIdx.y = 0

Col = 1

Row = 1



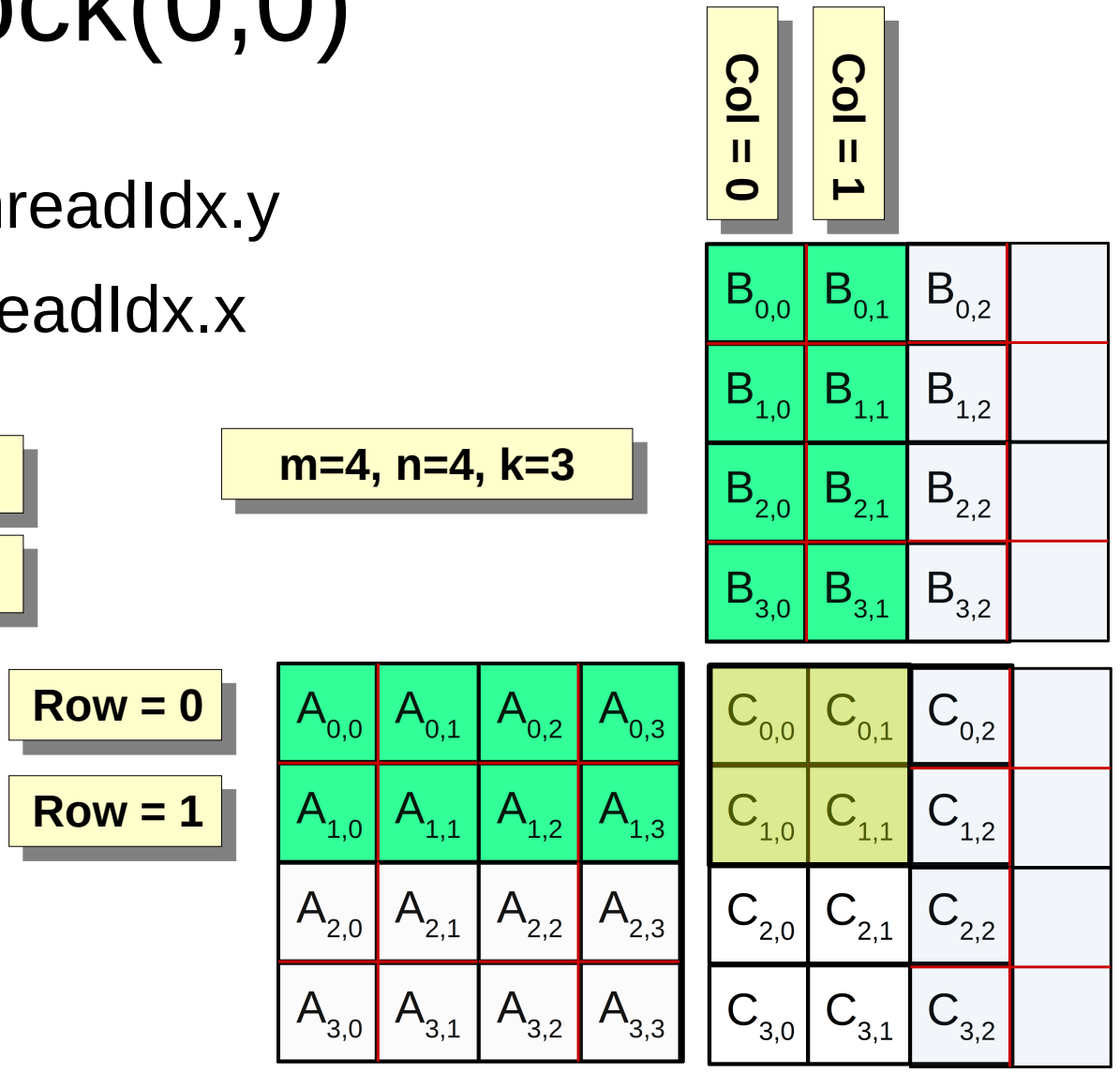
Work in Block(0,0)

- Row = 0 * 2 + threadIdx.y
- Col = 0 * 2 + threadIdx.x

threadIdx.x = {0,1}

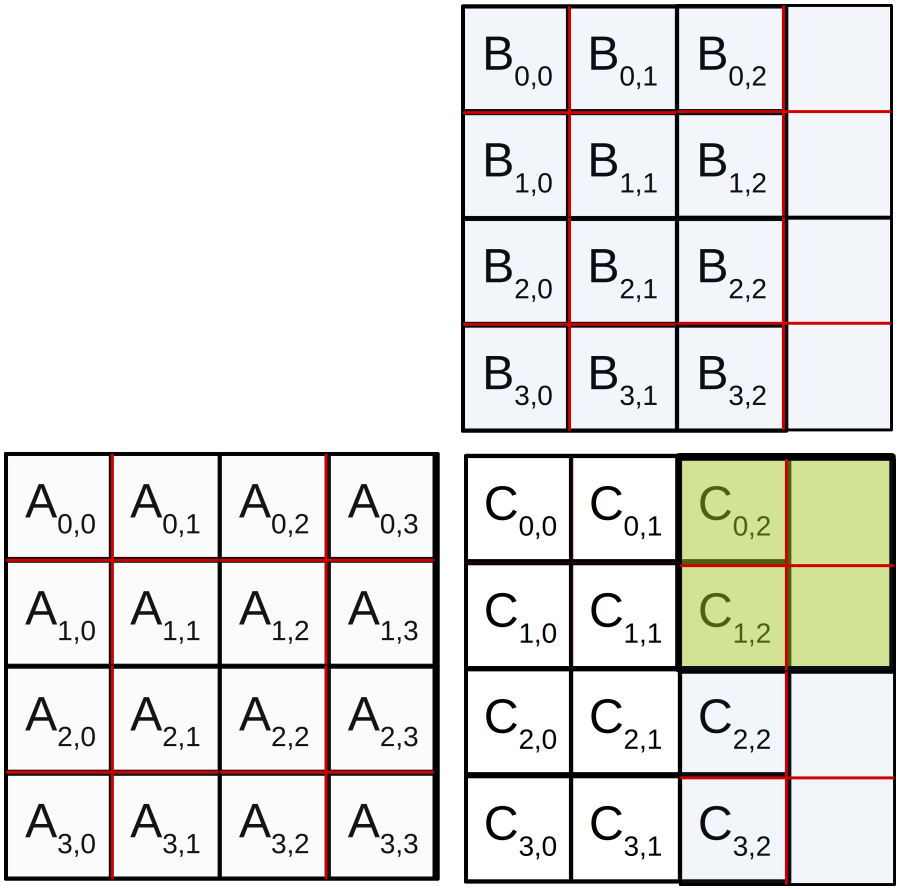
threadIdx.y = {0,1}

m=4, n=4, k=3



Work in Block(0,1)

- Row = 0 * 2 + threadIdx.y
- Col = 1 * 2 + threadIdx.x

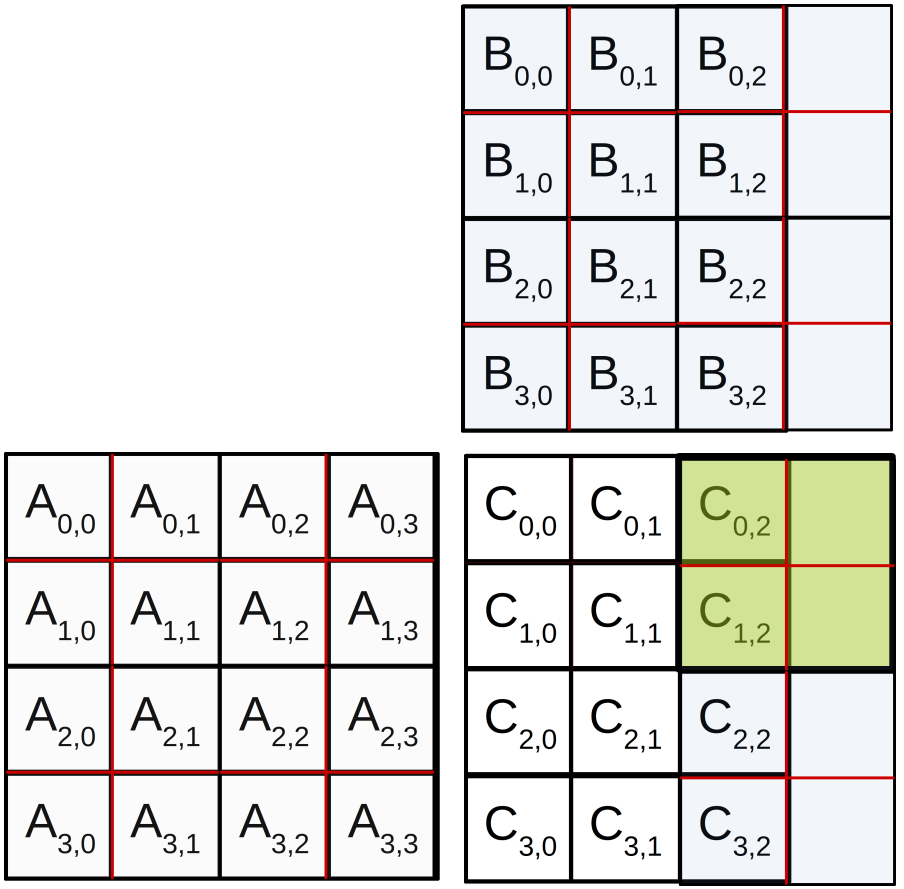


Work in Block(0,1)

- Row = 0 * 2 + threadIdx.y
- Col = 1 * 2 + threadIdx.x

threadIdx.x = {0,1}

threadIdx.y = {0,1}



Work in Block(0,1)

- Row = 0 * 2 + threadIdx.y
- Col = 1 * 2 + threadIdx.x