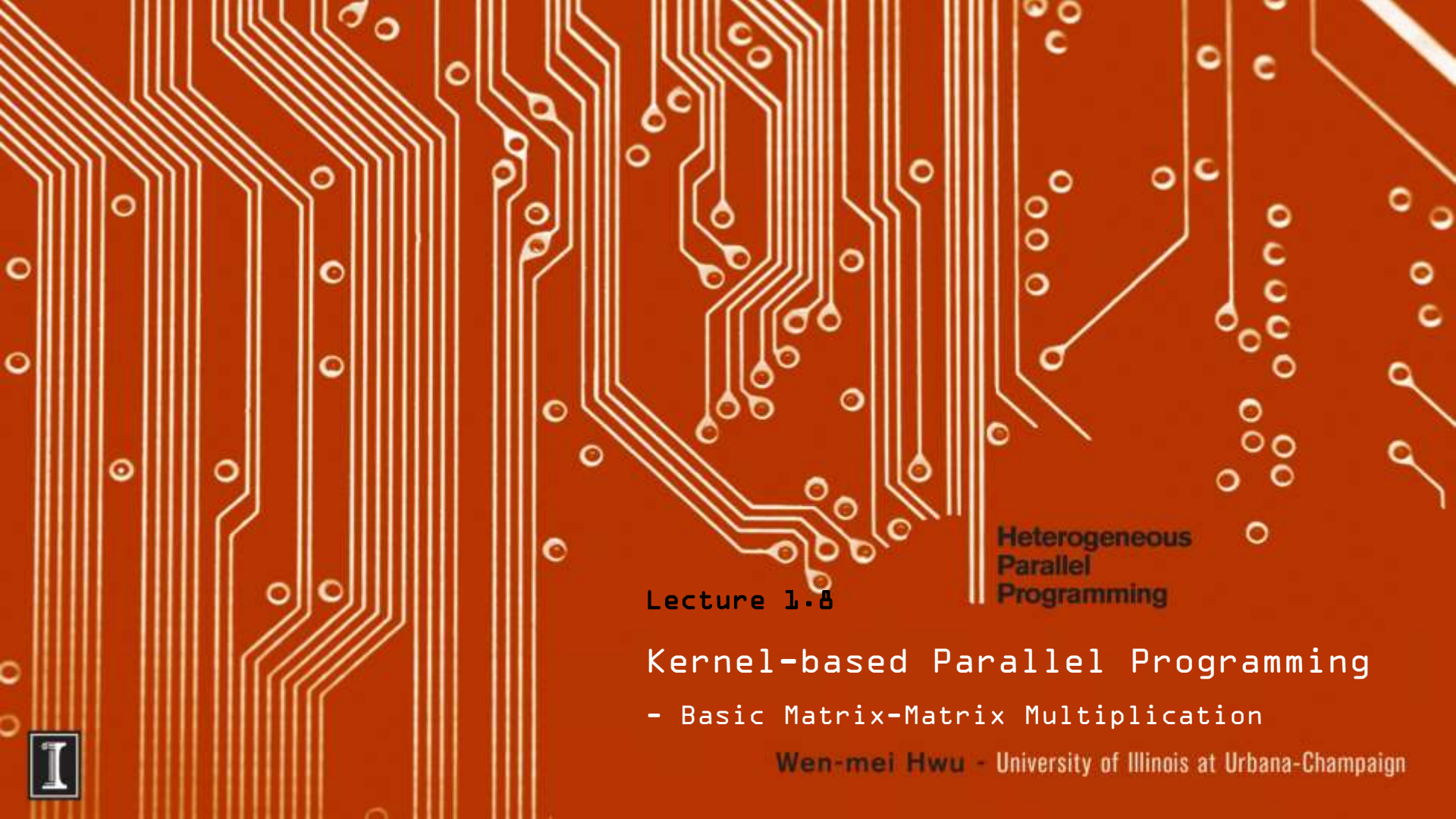


Lecture 1.8

Kernel-based Parallel Programming

- Basic Matrix-Matrix Multiplication

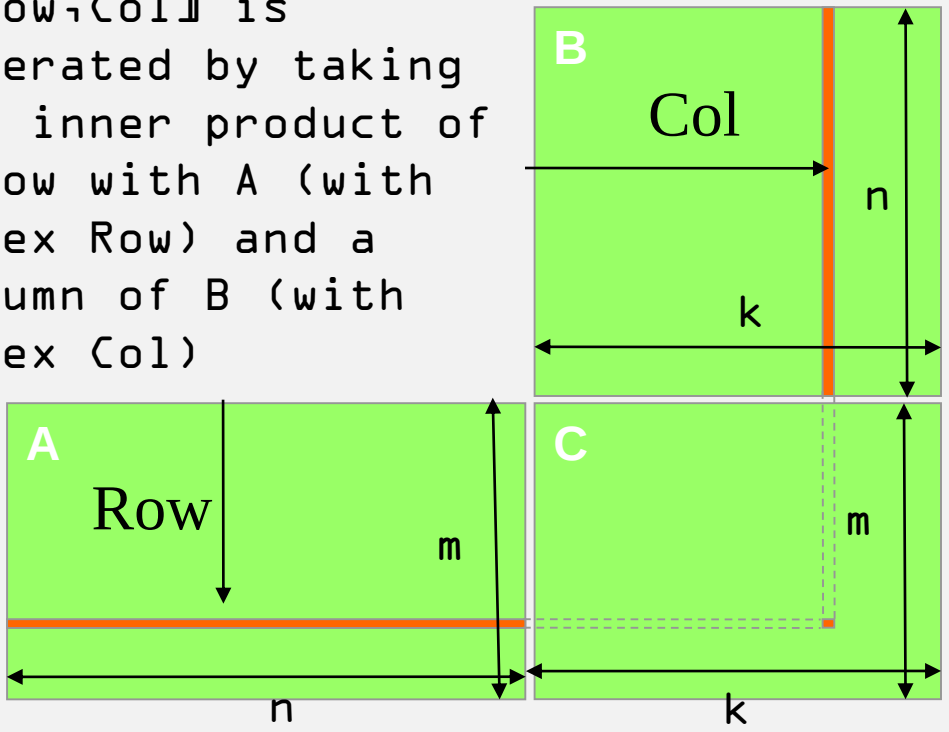
Wen-mei Hwu - University of Illinois at Urbana-Champaign

Objective

- To prepare for the lab assignment of matrix-matrix multiplication
  - Quick review of computation
  - Block/Thread index to data index mapping
  - Loop control flow in kernels

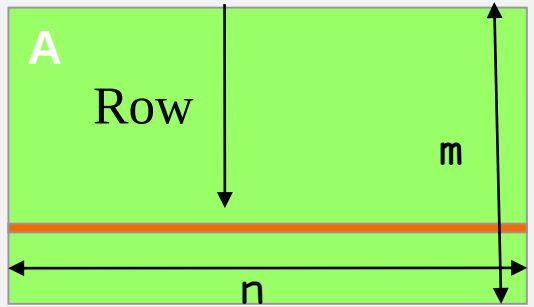
Matrix(-Matrix) Multiplication
A Simple Sequential Code in C

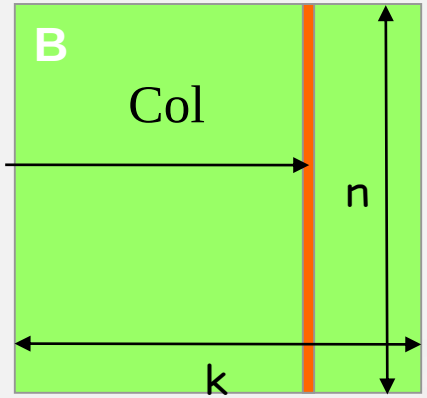
C[Row,Col] is generated by taking the inner product of a row with A (with index Row) and a column of B (with index Col)

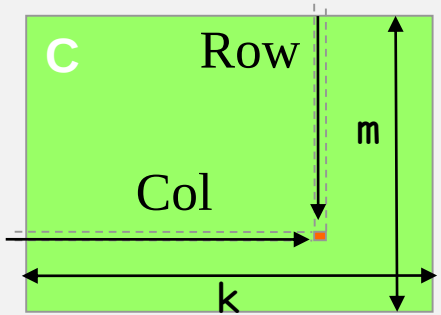


Matrix Multiplication
A Simple Sequential Code in C

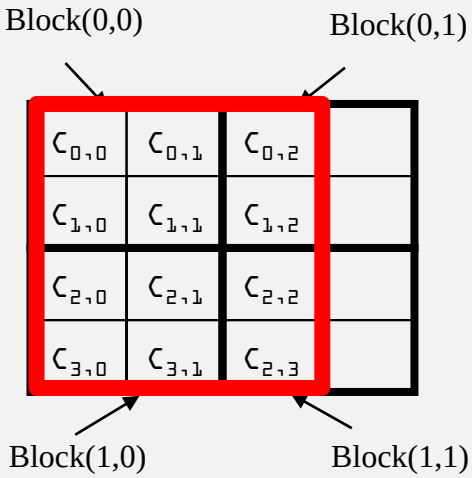
```
void MatrixMulOnHost(int m, int n, int k, float* A, float* B, float* C)
{
    for (int Row = 0; Row < m; ++Row)
        for (int Col = 0; Col < k; ++Col) {
            float sum = 0;
            for (int i = 0; i < n; ++i) {
                float a = A[Row * n + i];
                float b = B[Col+i*k];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







Kernel Function - A Small Example



TILE_WIDTH = 2
Each block has 2*2 = 4 threads

- Have each 2D thread block to compute a (TILE_WIDTH)^2 sub-matrix (tile) of the result matrix
  - Each has (TILE_WIDTH)^2 threads
  - We use square tiles for simplicity
- Generate a 2D Grid of blocks

m = 4; n = 4; k = 3
(m-1)/TILE_WIDTH +1= 2
(k-1)/TILE_WIDTH+1 = 2
Use 2* 2 = 4 blocks

Kernel Invocation in Host Code

```
// 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

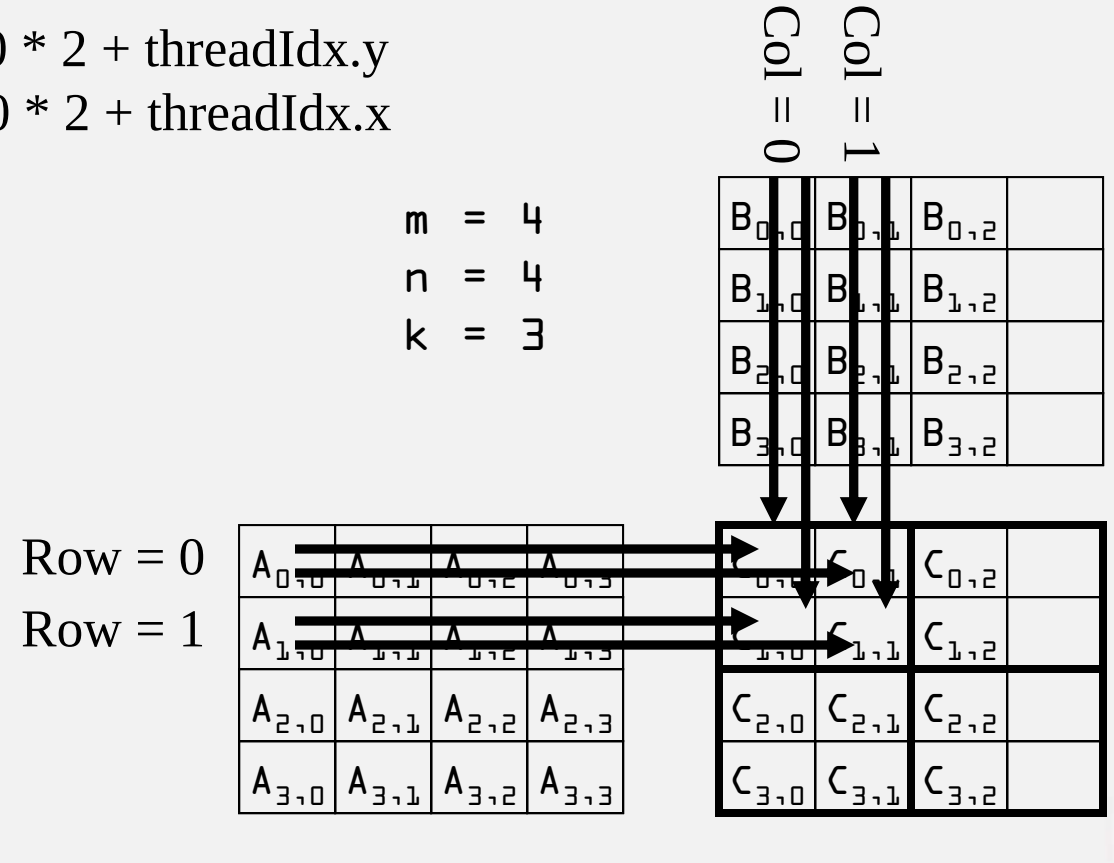
```
__global__ void MatrixMulKernel(int m, int
    n, int k, float* A, float* B, float* C)
{
    int Row =
    blockIdx.y*blockDim.y+threadIdx.y;
    int Col =
    blockIdx.x*blockDim.x+threadIdx.x;

    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 for Block (0,0)in a TILE_WIDTH = 2 Configuration

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

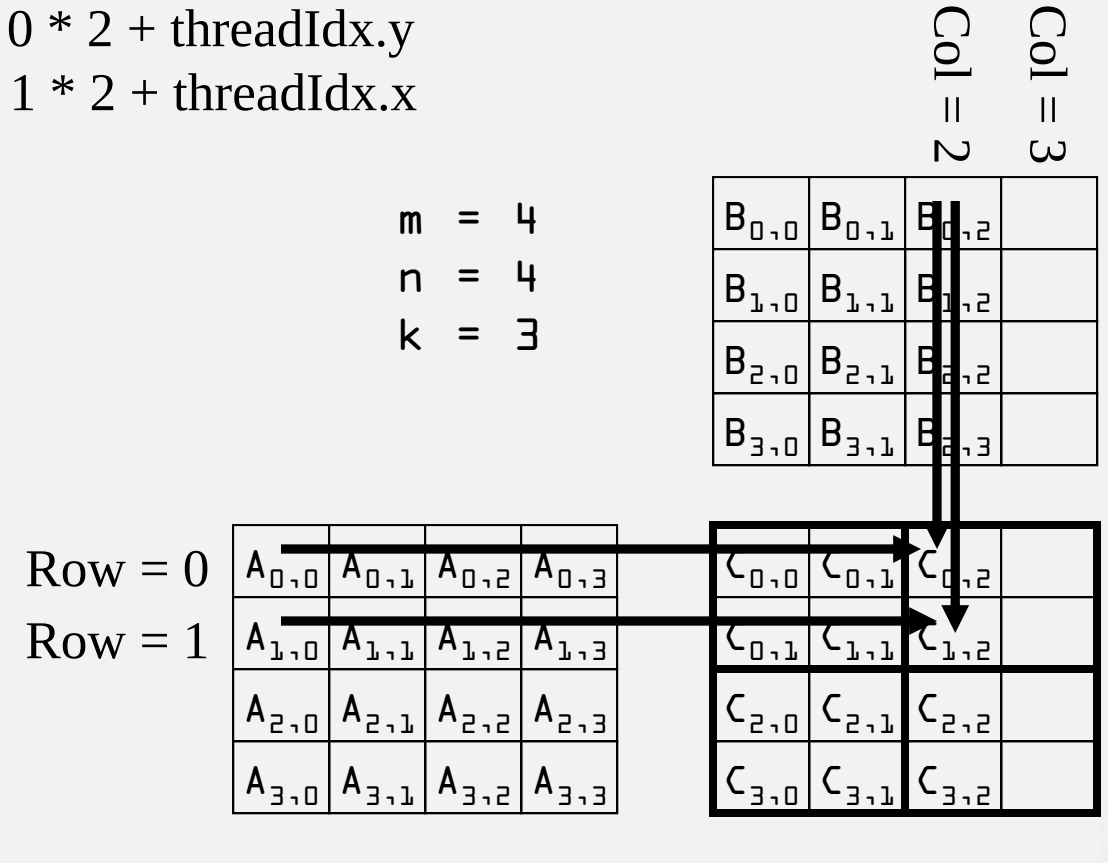
m = 4
n = 4
k = 3

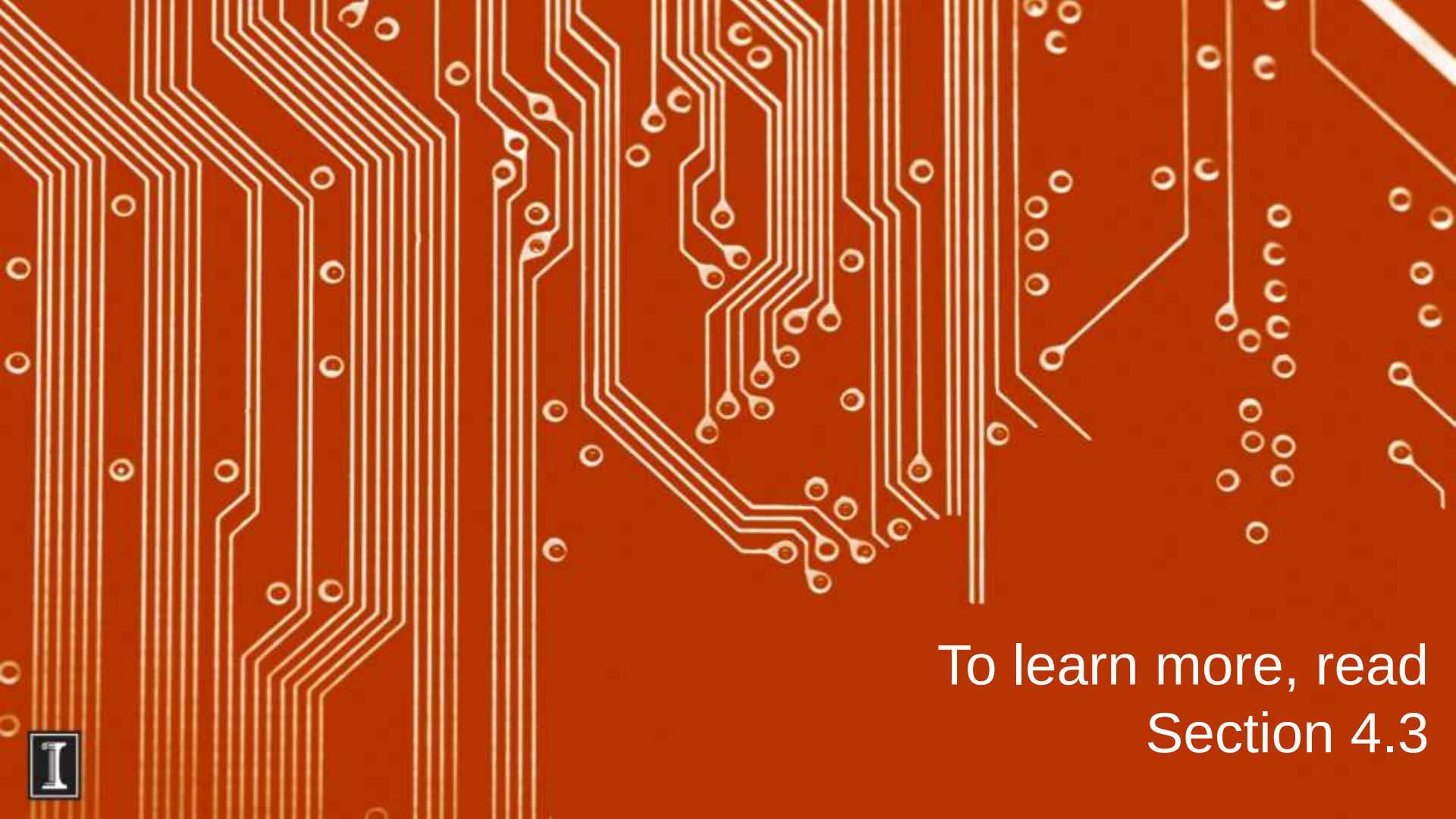


Work for Block (0,1)

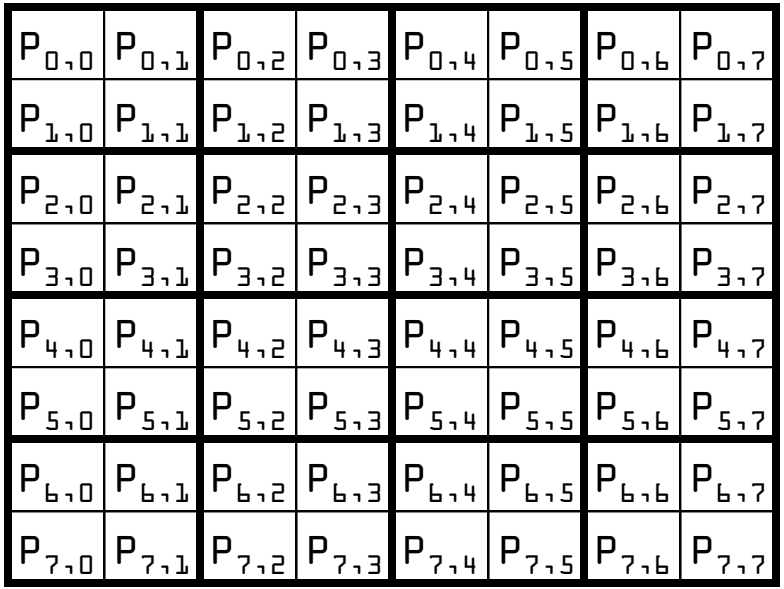
Row = 0 * 2 + threadIdx.y
Col = 1 * 2 + threadIdx.x

m = 4
n = 4
k = 3




To learn more, read
Section 4.3

A Slightly Bigger Example



WIDTH = 8; TILE_WIDTH = 2
Each block has 2*2 = 4 threads

WIDTH/TILE_WIDTH = 4
Use 4* 4 = 16 blocks

A Slightly Bigger Example (cont.)

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$	$P_{0,4}$	$P_{0,5}$	$P_{0,6}$	$P_{0,7}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$	$P_{1,4}$	$P_{1,5}$	$P_{1,6}$	$P_{1,7}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$	$P_{2,4}$	$P_{2,5}$	$P_{2,6}$	$P_{2,7}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$	$P_{3,4}$	$P_{3,5}$	$P_{3,6}$	$P_{3,7}$
$P_{4,0}$	$P_{4,1}$	$P_{4,2}$	$P_{4,3}$	$P_{4,4}$	$P_{4,5}$	$P_{4,6}$	$P_{4,7}$
$P_{5,0}$	$P_{5,1}$	$P_{5,2}$	$P_{5,3}$	$P_{5,4}$	$P_{5,5}$	$P_{5,6}$	$P_{5,7}$
$P_{6,0}$	$P_{6,1}$	$P_{6,2}$	$P_{6,3}$	$P_{6,4}$	$P_{6,5}$	$P_{6,6}$	$P_{6,7}$
$P_{7,0}$	$P_{7,1}$	$P_{7,2}$	$P_{7,3}$	$P_{7,4}$	$P_{7,5}$	$P_{7,6}$	$P_{7,7}$

WIDTH = 8; TILE_WIDTH = 4
Each block has 4*4 =16 threads

WIDTH/TILE_WIDTH = 2
Use 2* 2 = 4 blocks