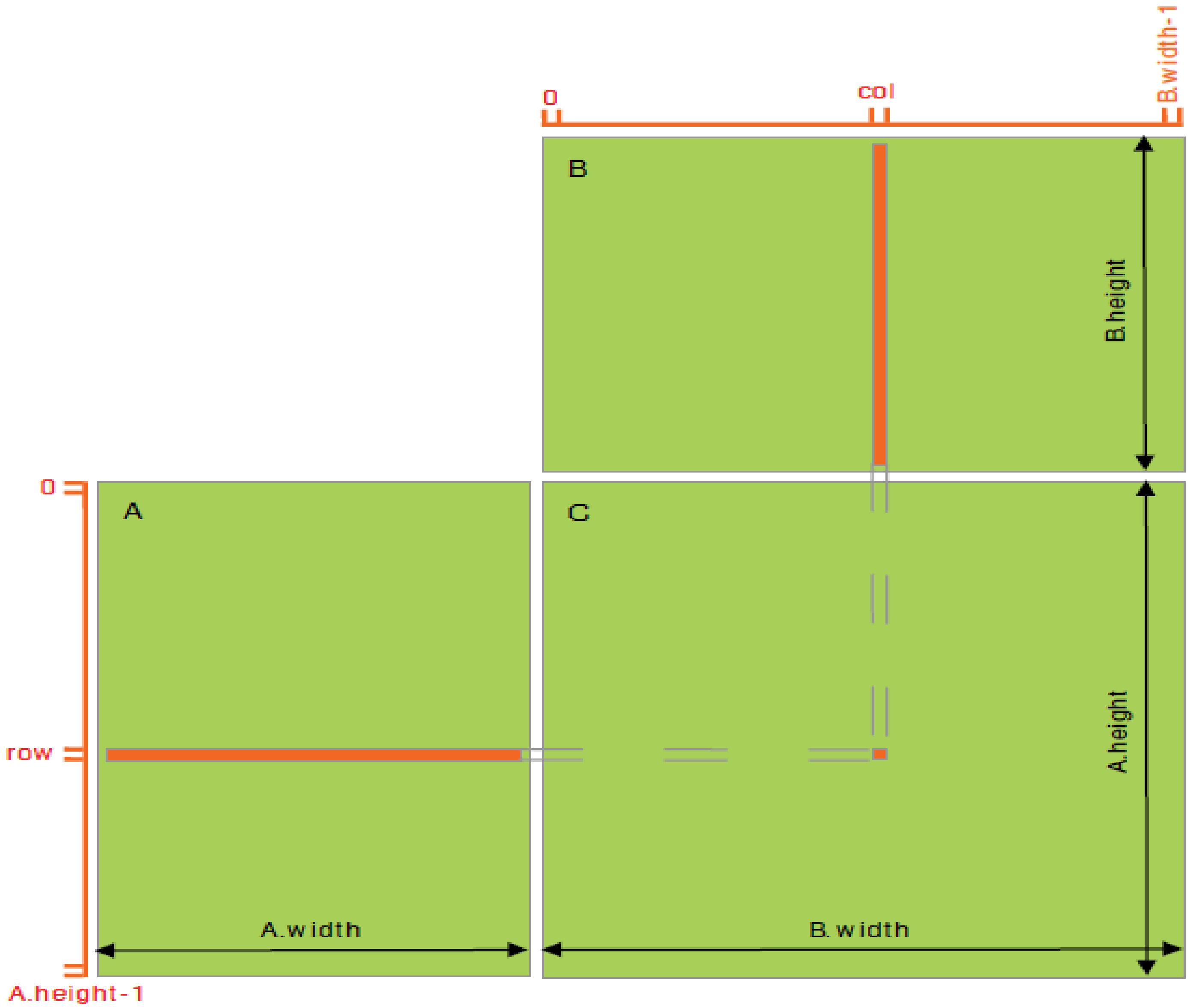
Lecture No.3: Matrix Multiplication

Muhammad Osama Mahmoud, TA

Computer and Systems Eng. Dept. - Faculty of Engineering – University of Minia

Basic Matrix Multiplication



Basic 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 C_Element = 0.0;
                for (int i = 0; i < n; i++)
                        C_Element += A[Row*n + i] * B[Col + i*k];
                C[Row*k + Col] = C_Element;
        }
}
```

Basic Kernel Performance

- All threads access global memory for every matrix input even if reused many times
- Assume two matrices (A, B) have the same number of elements (n x n)
- For the first row in A, the number of multiply-add operations is n
- The number of repeated global memory accesses are n^2
- These repeated global accesses can be reduced from n^2 to just n if we take advantage of the tiling strategy
- Now we ready for the tiled matrix multiplication algorithm

Tiled Matrix Multiplication

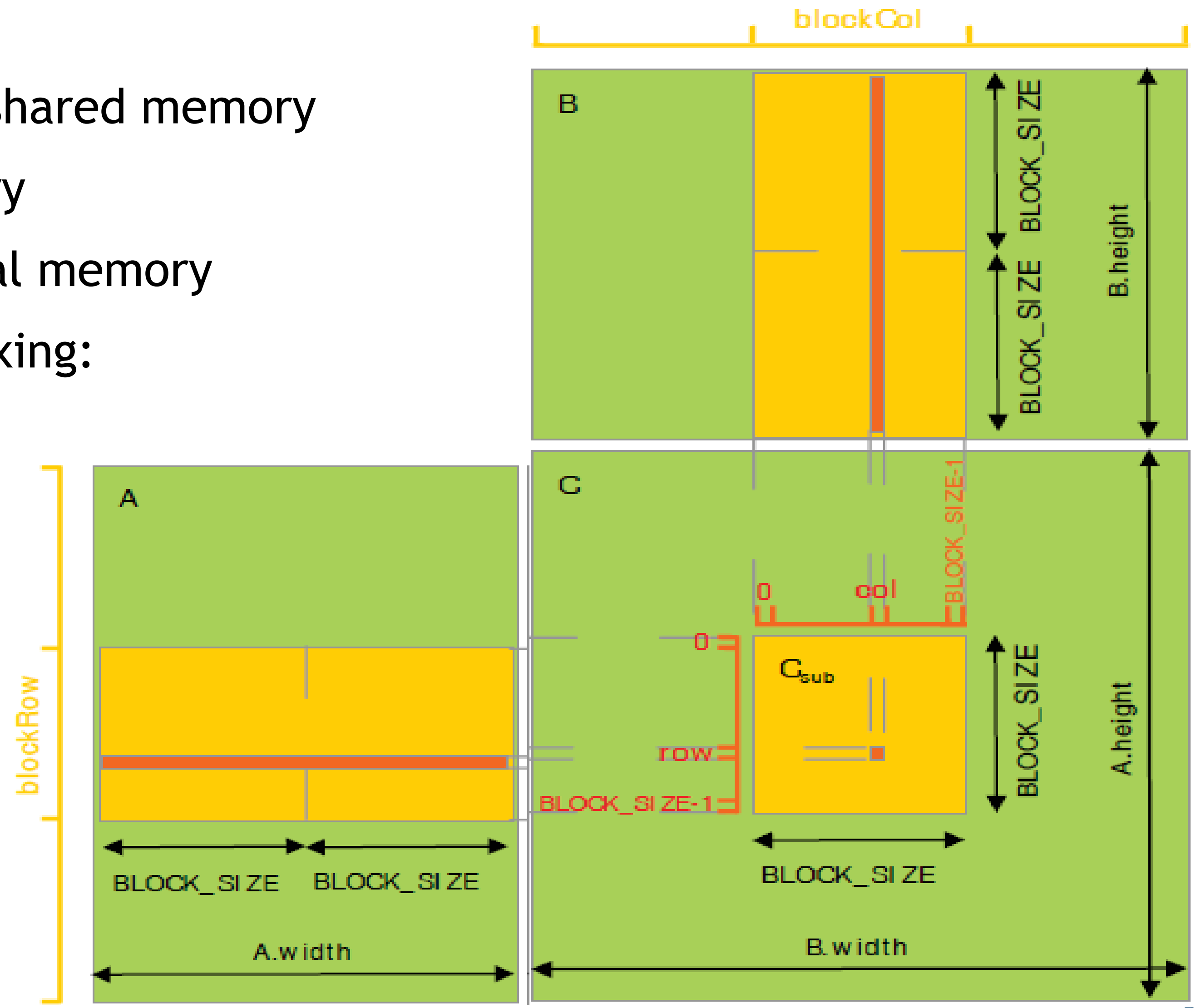
- Load a submatrix from A and B into shared memory
- Process data reside in shared memory
- Write the result submatrix into global memory
- Ex. Accessing t block/tile in 2D indexing:

  A[Row][t * BLOCK_SIZE + tx]

  B[t * BLOCK_SIZE + ty][Col]

  Where tx = threadIdx.x

  ty = threadIdx.y



Tiled Matrix Multiplication (Cont.)

- Mapping from 2D to 1D
  - A[Row][t * BLOCK_SIZE + tx]

↓

A[Row * n + t * BLOCK_SIZE + tx] , where ***n*** is the number of elements in a row of A

  - B[t * BLOCK_SIZE + ty][Col]

↓

B[(t * BLOCK_SIZE + ty) * k + Col] , where ***k*** is the number of elements in a column of B

Tiled Matrix Multiplication Kernel

```
__global__ void TiledMatrixMulKernel(int m, int n, int k, float* A, float* B, float* C) {
        __shared__ float ds_A[TILE_WIDTH][TILE_WIDTH];
        __shared__ float ds_B[TILE_WIDTH][TILE_WIDTH];
        int bx = blockIdx.x; int by = blockIdx.y;
        int tx = threadIdx.x; int ty = threadIdx.y;
        int Row = by * blockDim.y + ty;
        int Col = bx * blockDim.x + tx;
        float C_Element = 0.0;
        // rest of kernel in next slide
```

Tiled Matrix Multiplication Kernel (cont.)

```
    // Loop over the A and B tiles required to compute the C element
    for (int t = 0; t < n/BLOCK_SIZE; t++) {
        // load of A and B tiles into shared memory
        ds_A[ty][tx] = A[Row*n + t*BLOCK_SIZE+tx];
        ds_B[ty][tx] = B[(t*BLOCK_SIZE+ty)*k + Col];
        __syncthreads();
        for (int i = 0; i < BLOCK_SIZE; ++i)
            C_Element += ds_A[ty][i] * ds_B[i][tx];
        __synchthreads();
        }
        C[Row*k+Col] = C_Element;
} // kernel end
```

References

[1] Wen-mei W. Hwu, “Heterogeneous Parallel Programming”. Online course, 2014. Available: https://class.coursera.org/hetero-002

[2] NVIDIA, “CUDA C Programming Guide”, June 2014.