Heterogeneous Parallel Programming COMP4901D

CUDA Example: Matrix Multiplication

`

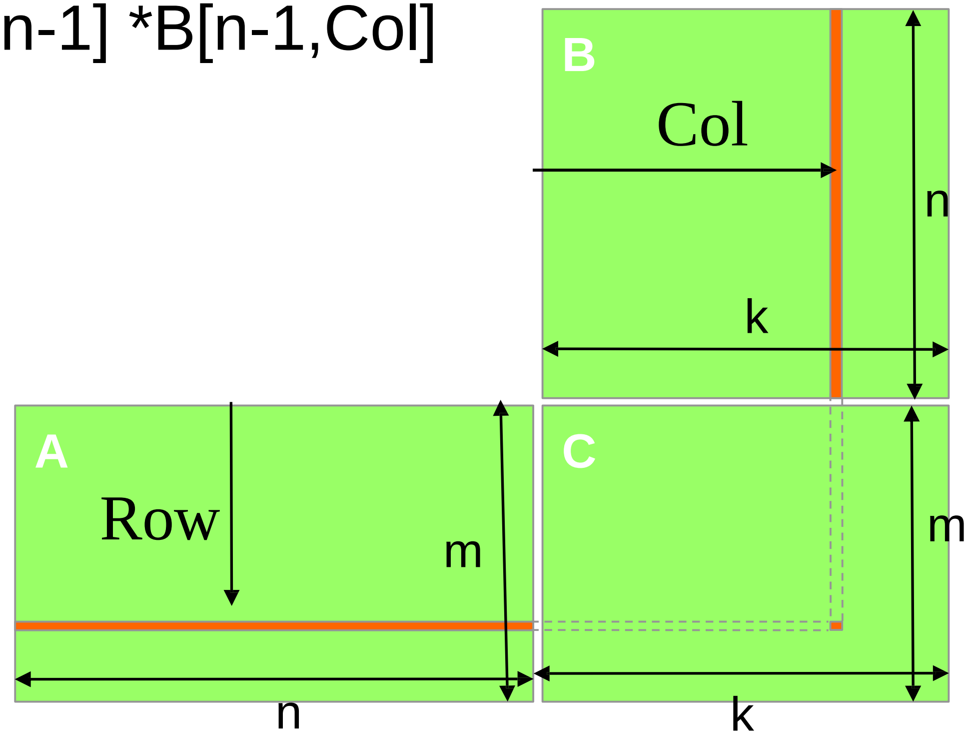
Overview

- Matrix multiplication as an example in CUDA
  - Math operation review
  - Baseline implementation
  - Tiling for shared memory/blocking

Math Review: Matrix Multiplication

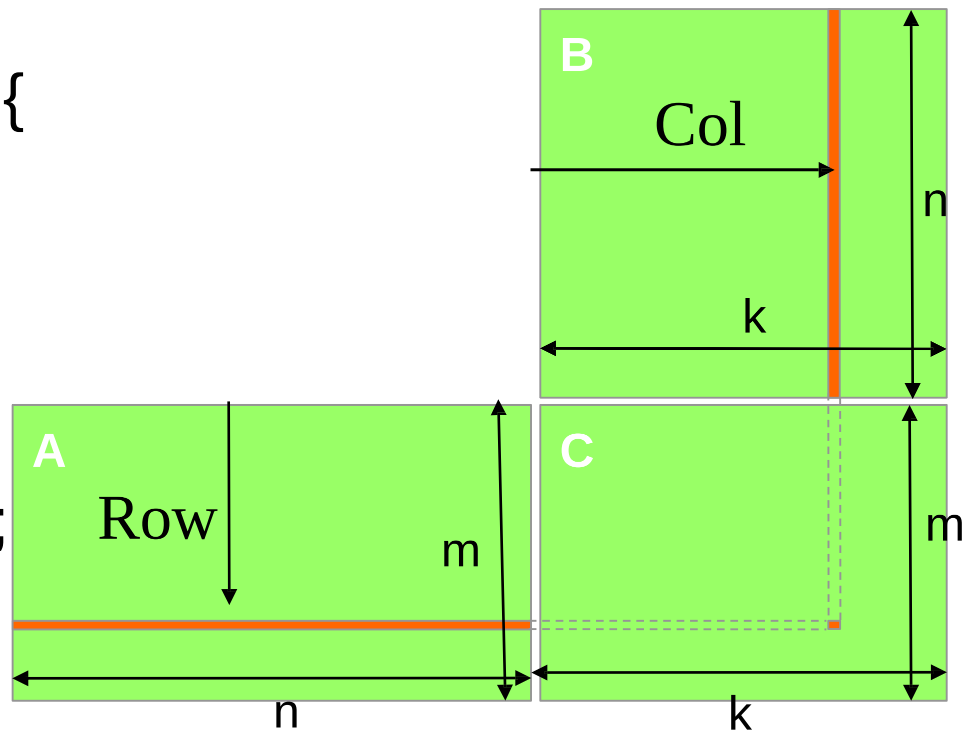
A_{mxn} X B_{nxk} = C_{mxk}

C[Row,Col] = A's row at Row· B's column at Col
= A[Row,0] * B[0, Col] + A[Row,1]*B[1,Col} + …
+A[Row,n-1] *B[n-1,Col]



Sequential C code

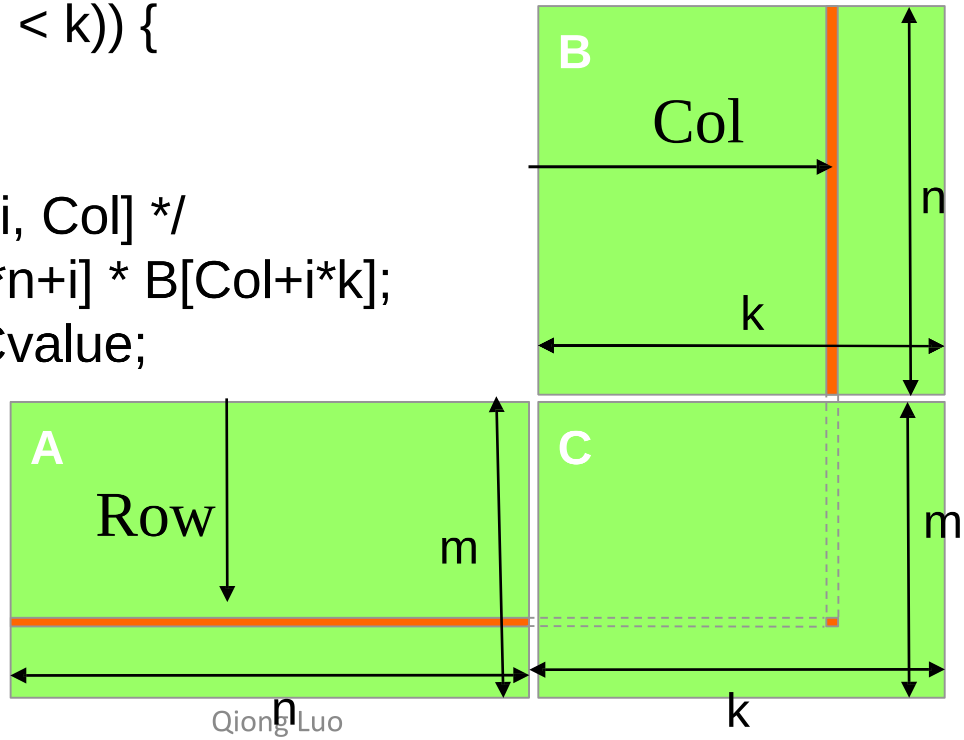
```
void MatrixMulOnHost(int m, int n, int k, float* A, float* B, float* C)
{
 for (int Row = 0; Row < m; ++Row)
   for (int Col = 0; Col < k; ++Col) {
     float sum = 0;
     for (int i = 0; i < n; ++i) {
     float a = A[Row*n + i];
     float b = B[Col + i*k];
     sum += a *b;
     }
     C[Row*k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



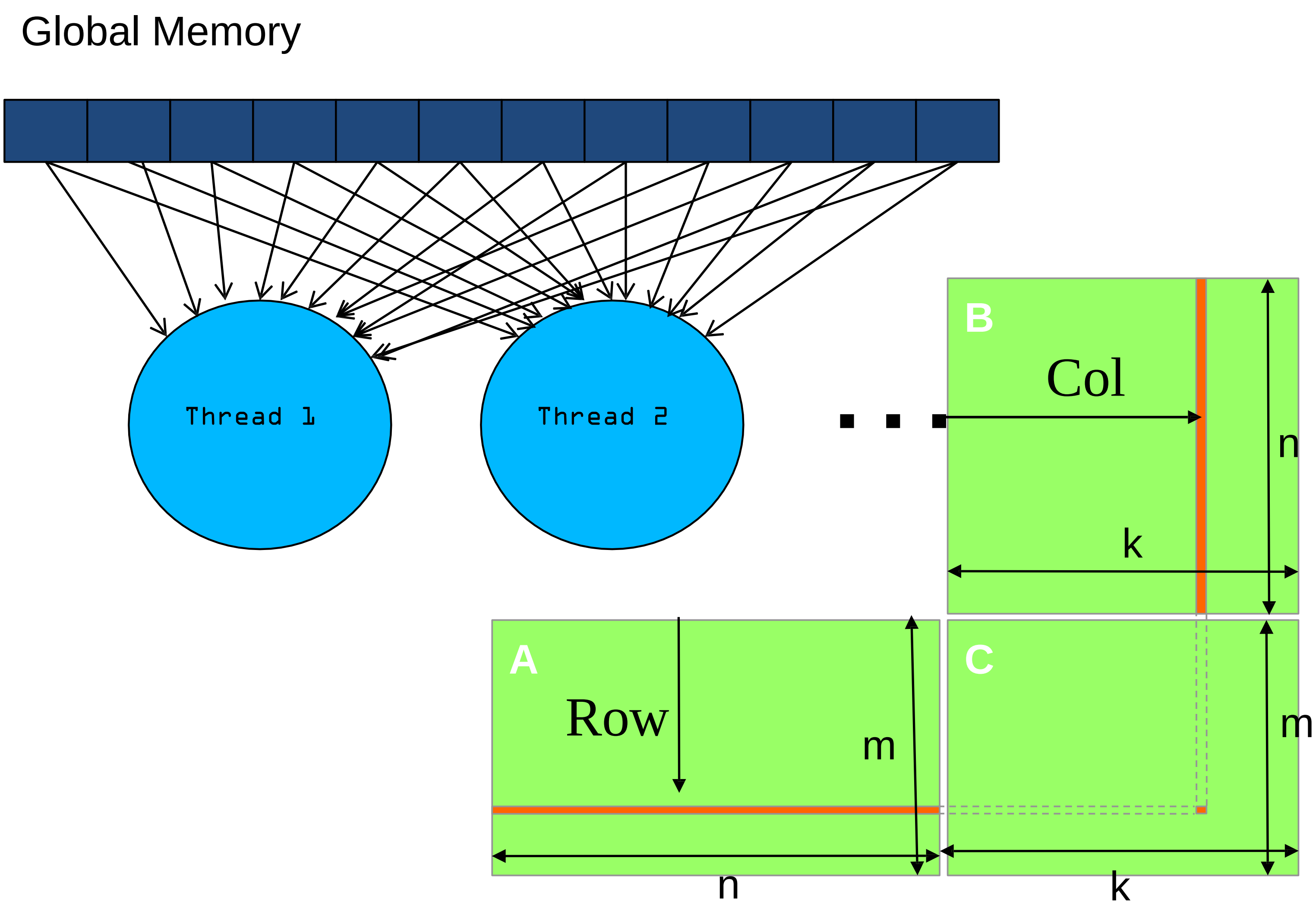
Baseline Kernel

```
_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)
        /* A[Row, i] and B[i, Col] */
        Cvalue += A[Row*n+i] * B[Col+i*k];
        C[Row*k+Col] = Cvalue;
    } }
```



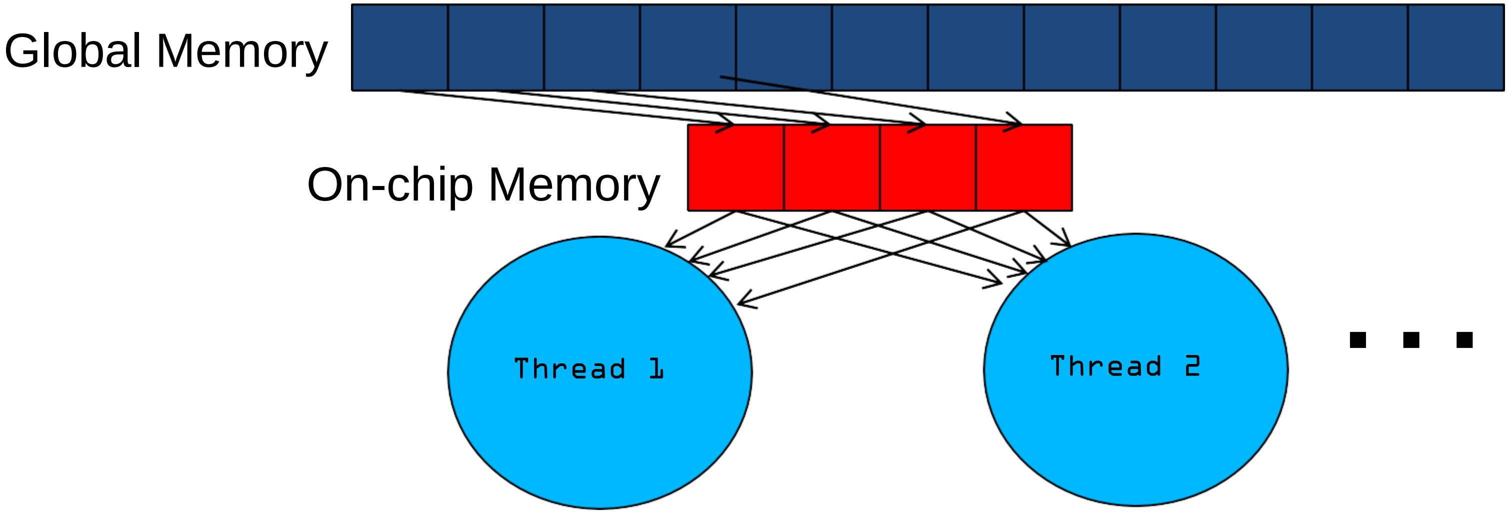
Memory Access Pattern



Review: Tiling with Faster Memory

- Identify a tile of global memory content that are accessed by multiple threads
- Load the tile from global memory into on-chip memory
- Have the multiple threads to access their data from the on-chip memory
- Move on to the next tile

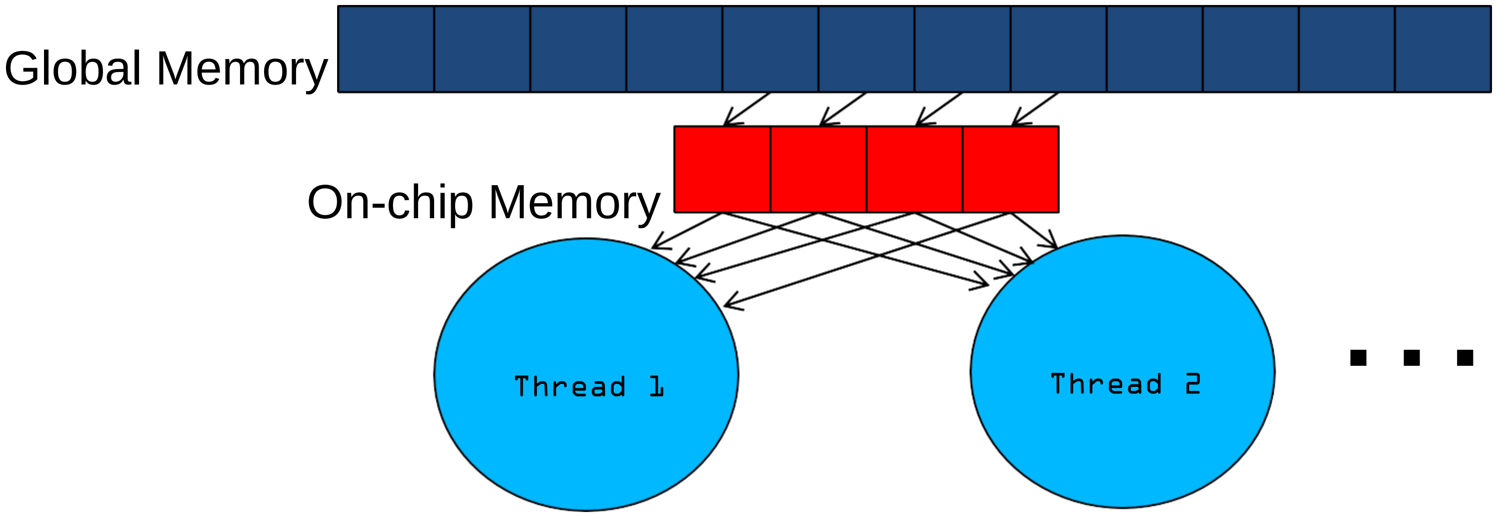
Shared Memory Tiling/Blocking



Divide the global memory content into tiles

Focus the computation of threads on one or a small number of tiles at each point in time

Shared Memory Tiling/Blocking

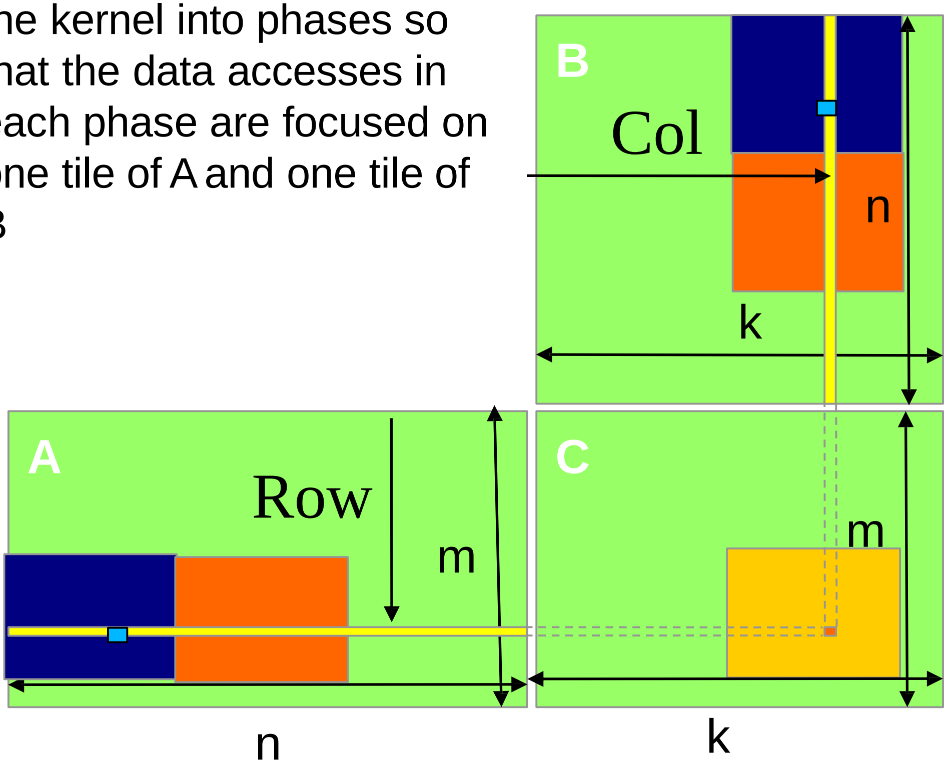


Divide the global memory content into tiles

Focus the computation of threads on one or a small number of tiles at each point in time

Matrix Multiplication Tiled

- Break up the execution of the kernel into phases so that the data accesses in each phase are focused on one tile of A and one tile of B

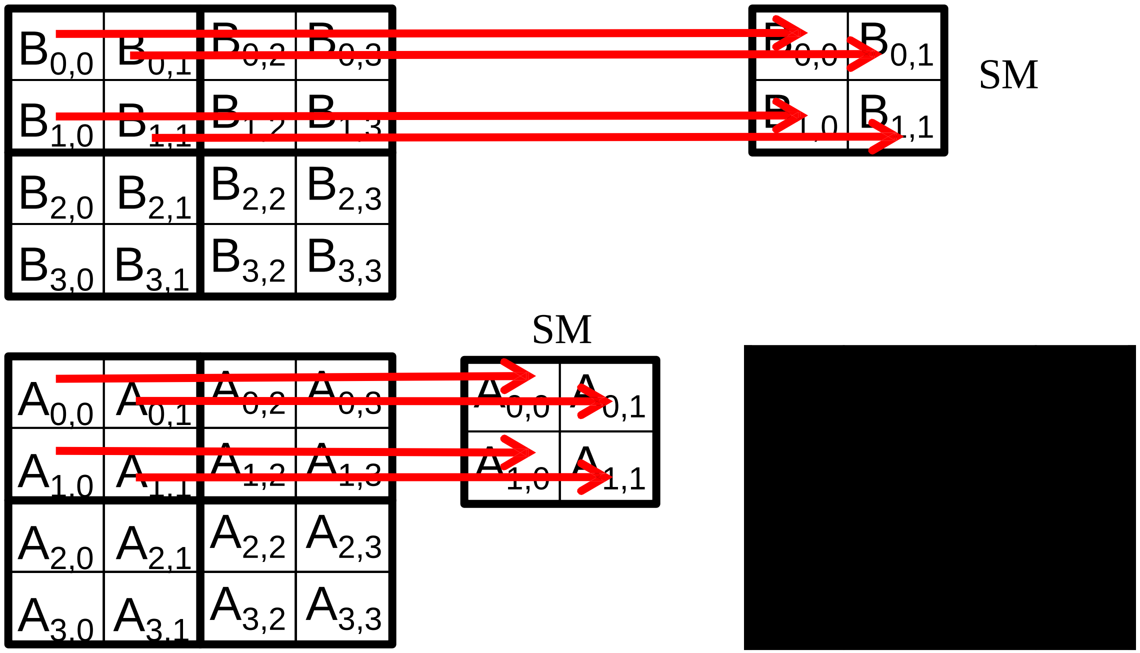


Loading a Tile

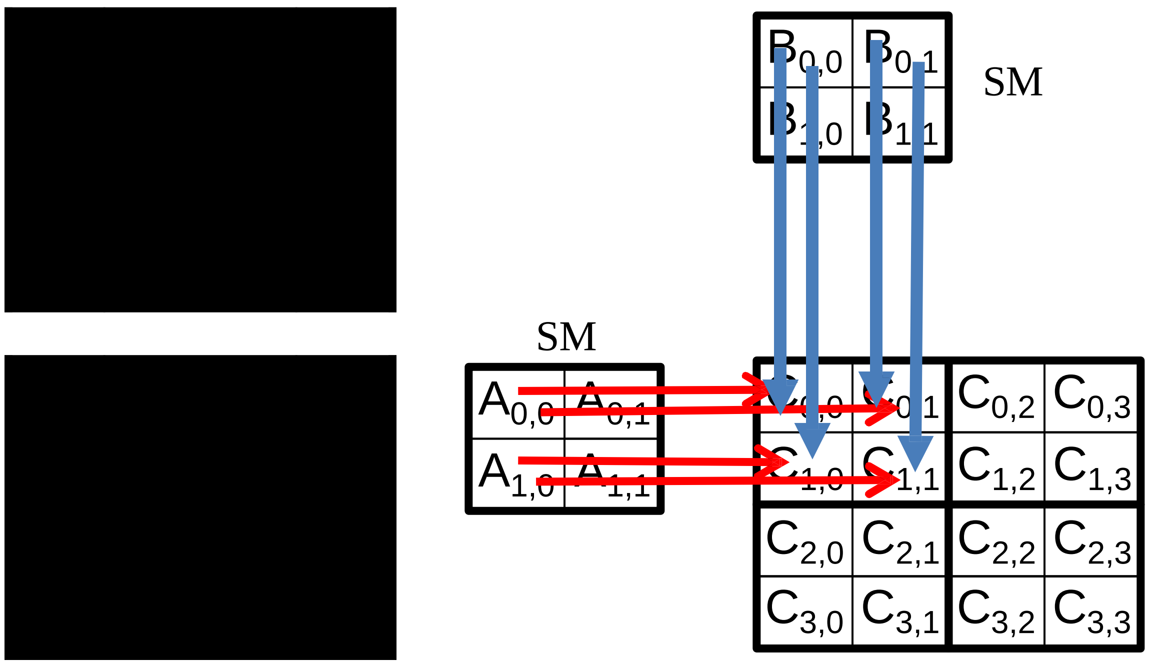
- All threads in a block participate
  - Each thread loads one A element and one B element in tiled code

- Assign the loaded element to each thread such that the accesses within each warp are coalesced

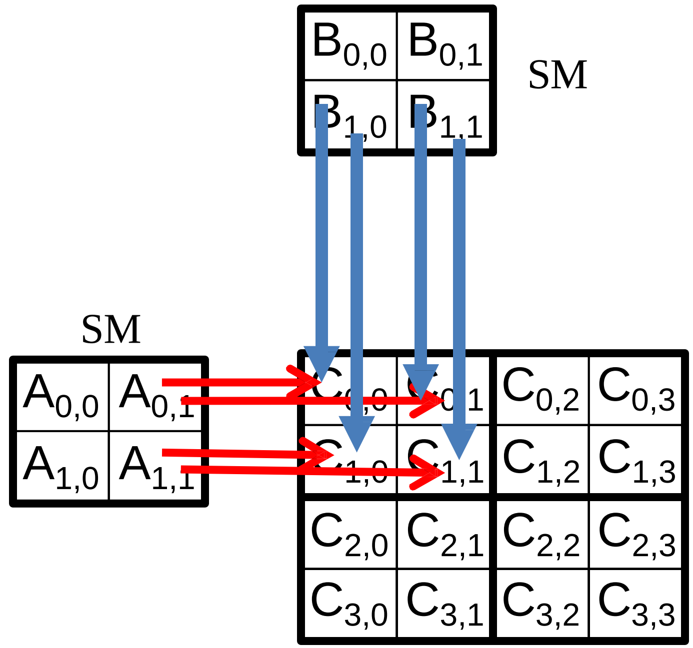
Phase 0: Load for Block (0,0) of C



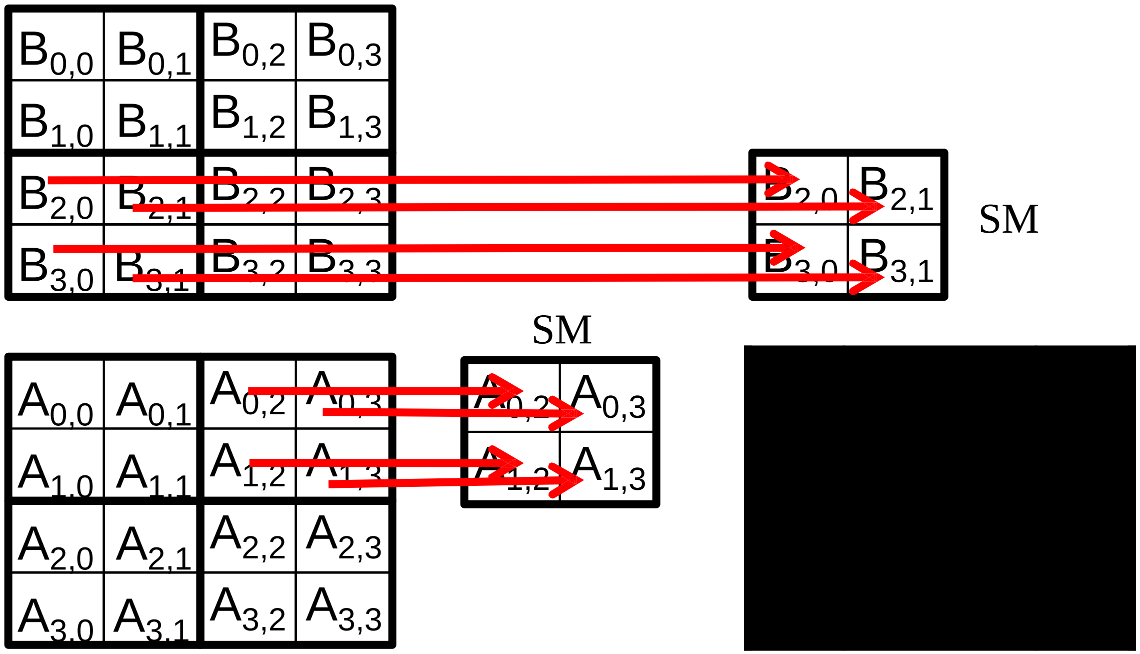
Phase 0: Compute Block (0,0) Iteration 0



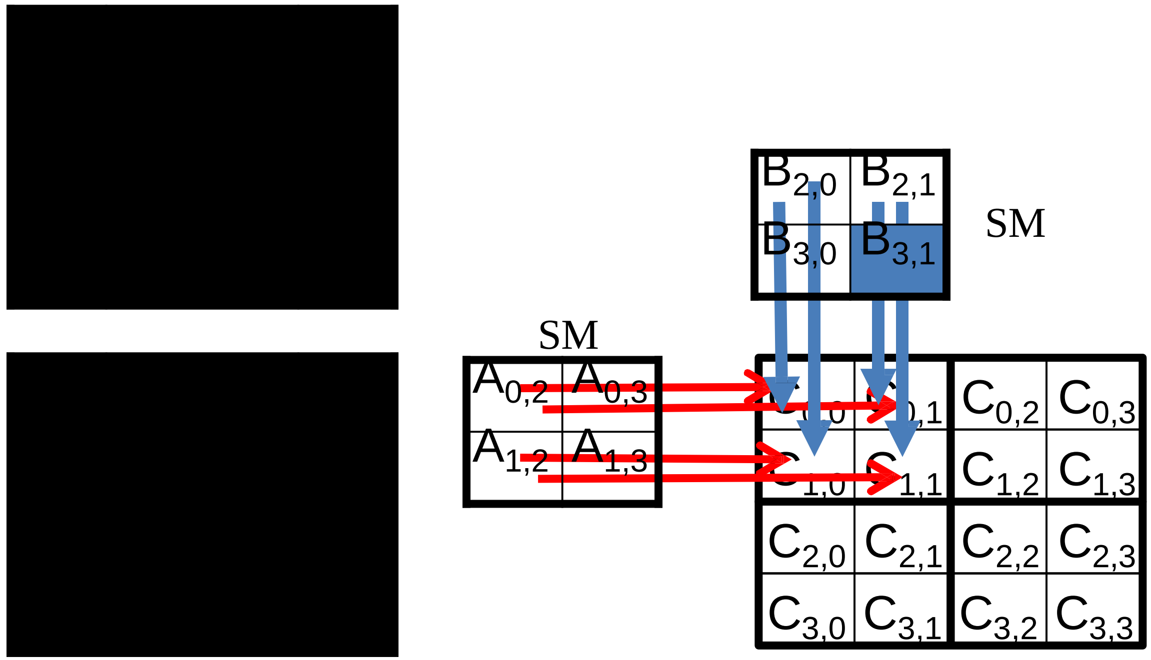
Phase 0: Compute Block (0,0) Iteration 1



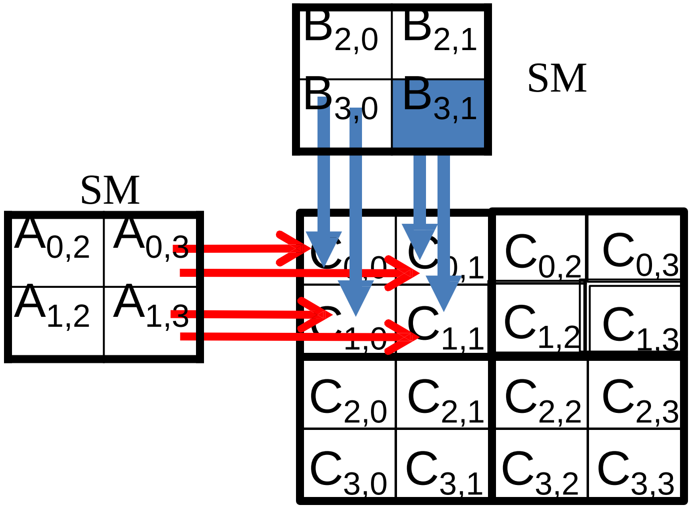
Phase 1: Load for Block (0,0) of C



Phase 1: Compute Block (0,0) Iteration 0



Phase 1: Compute Block (0,0) Iteration 1



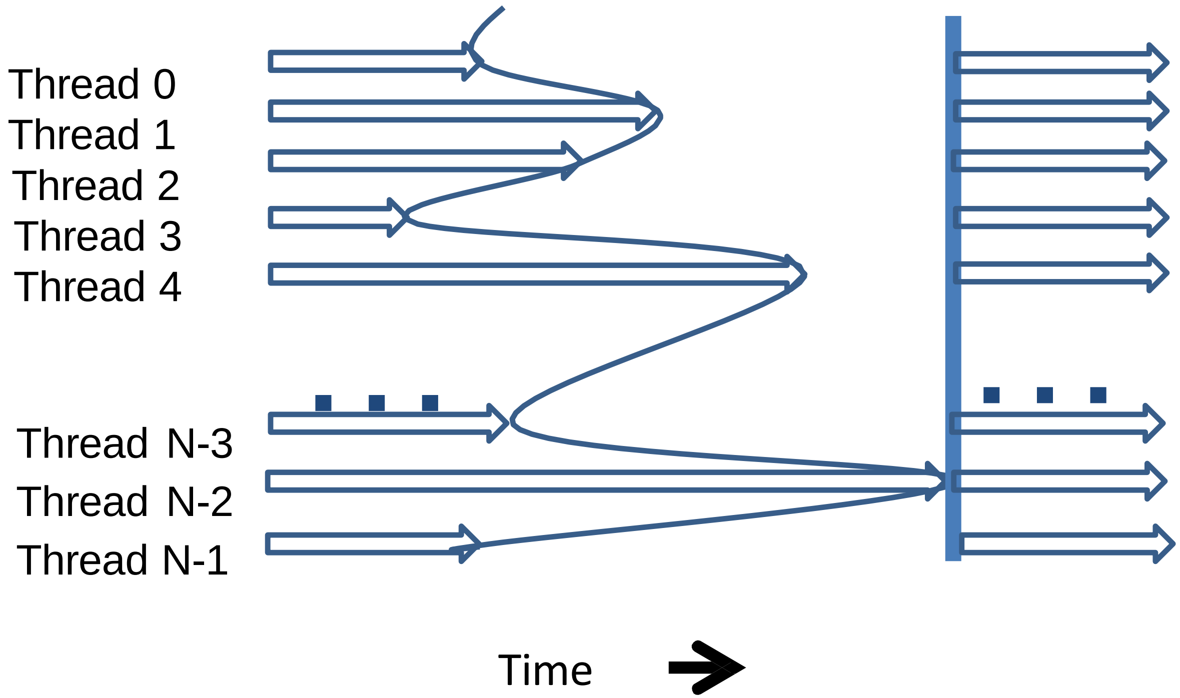
Barrier Synchronization

- An API function call in CUDA
  - `__syncthreads()`

- All threads in the same block must reach the __syncthreads() before any can move on

- Best used to coordinate tiled algorithms
  - `To ensure that all elements of a tile are loaded`
  - `To ensure that all elements of a tile are consumed`

Barrier Synchronization Timing

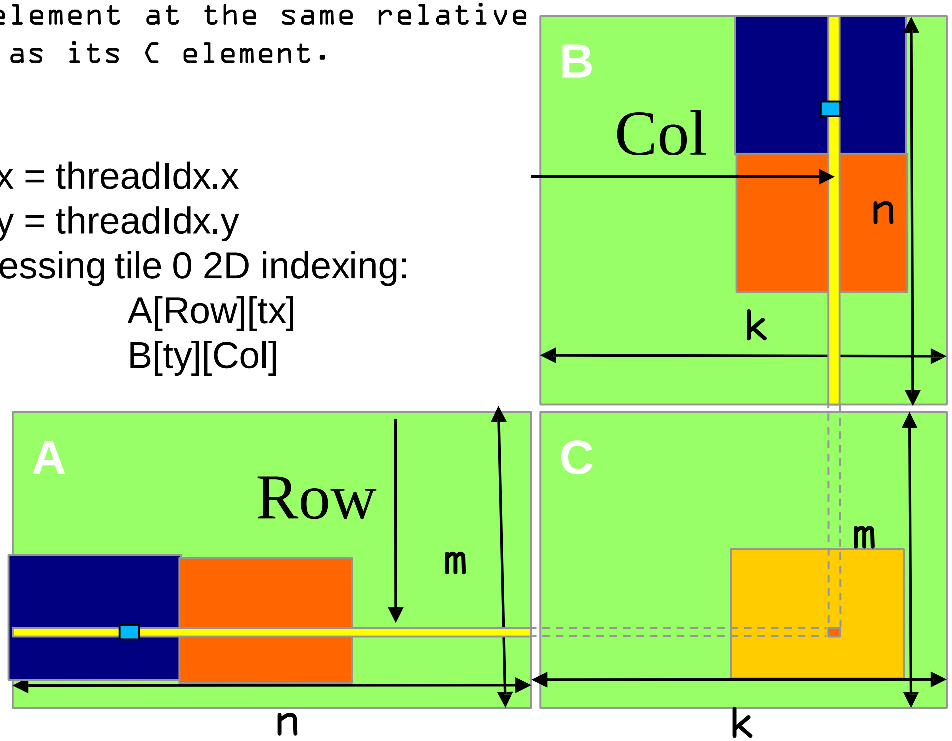


Caution: Syncthreads() can significantly reduce active threads in a block.

Loading a Tile: Element Index

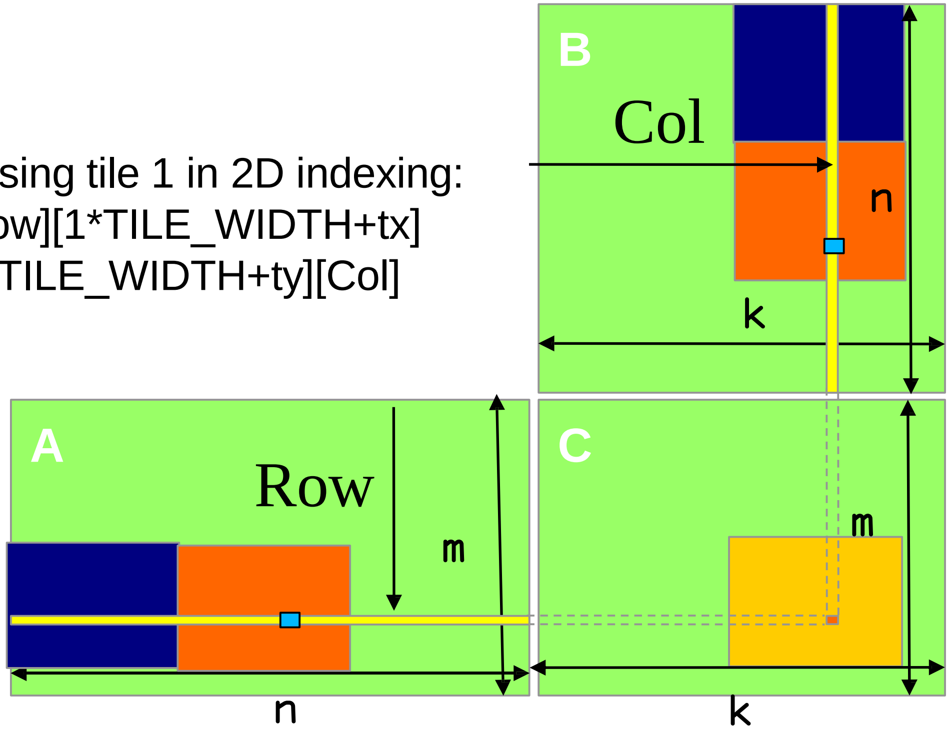
```
Have each thread to load an A element
and a B element at the same relative
position as its C element.
```

int tx = threadIdx.x
int ty = threadIdx.y
Accessing tile 0 2D indexing:
A[Row][tx]
B[ty][Col]



Loading a Tile: Element Index (cont.)

Accessing tile 1 in 2D indexing:
A[Row][1*TILE_WIDTH+tx]
B[1*TILE_WIDTH+ty][Col]



Loading a Tile: Element Index (cont.)

A[Row][t*TILE_WIDTH+tx]
⇨ A[Row*n + t*TILE_WIDTH + tx]

B[t*TILE_WIDTH+ty][Col]
⇨ B[(t*TILE_WIDTH+ty)*k + Col]

where t is the tile sequence number of the current phase

A and B are using 1D indexing

Tiled Matrix Multiplication Kernel

```
__global__void MatrixMulKernel(int m, int n, int k, float* A,
     float* B, float* C)
{
1.   __shared__ float ds_A[TILE_WIDTH][TILE_WIDTH];
2.   __shared__ float ds_B[TILE_WIDTH][TILE_WIDTH];

3.   int bx = blockIdx.x;  int by = blockIdx.y;
4.   int tx = threadIdx.x; int ty = threadIdx.y;

5.   int Row = by * blockDim.y + ty;
6.   int Col = bx * blockDim.x + tx;
7.   float Cvalue = 0;
```

Tiled Matrix Multiplication Kernel (cont.)

```
    // Loop over the A and B tiles required to compute the C element
    for (int t = 0; t < n/TILE_WIDTH; ++t) {
    // Collaborative loading of A and B tiles into shared memory
      ds_A[ty][tx] = A[Row*n + t*TILE_WIDTH+tx];
      ds_B[ty][tx] = B[(t*TILE_WIDTH+ty)*k + Col];
      __syncthreads();

      for (int i = 0; i < TILE_WIDTH; ++i)
         Cvalue += ds_A[ty][i] * ds_B[i][tx];
    __synchthreads();

    }
     C[Row*k+Col] = Cvalue;
}
```

Block Size Consideration

- Each thread block should have many threads
  - TILE_WIDTH of 16 gives 16*16 = 256 threads
  - TILE_WIDTH of 32 gives 32*32 = 1024 threads

- For 16, each block performs 2*256 = 512 float loads from global memory for 256 * (2*16) = 8,192 mul/add operations. (memory traffic reduced by a factor of 16)

- For 32, each block performs 2*1024 = 2048 float loads from global memory for 1024 * (2*32) = 65,536 mul/add operations. (memory traffic reduced by a factor of 32)

Shared Memory Size Consideration

- Shared memory size is implementation dependent!
- For TILE_WIDTH = 16, each thread block uses 2*256*4B = 2KB of shared memory.
- For 16K shared memory, one can potentially have up to 8 thread blocks executing
    - This allows up to 8*512 = 4,096 pending loads. (2 per thread, 256 threads per block)

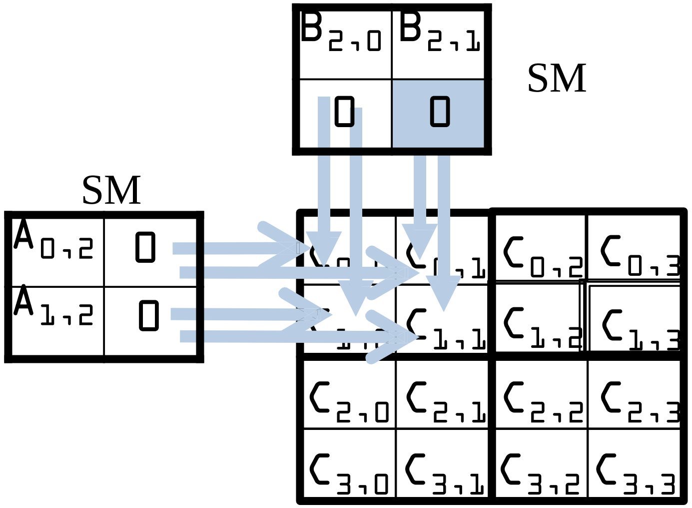
What If Tiles Exceed Matrix Boundaries

- When a thread is to load any input element, test if it is in the valid index range
  - If valid, proceed to load
  - Else, do not load, just write a 0
- Rationale: a 0 value will ensure that that the multiply-add step does not affect the final value of the output element

Compute Elements Exceeding Boundaries

- If a thread does not calculate a valid C element
  - Can still perform multiply-add into its register
  - As long as it is not allowed to write to the global memory at the end of the kernel
  - This way, the thread does not need to be turned off by an if-statement like in the basic kernel; it can participate in the tile loading process

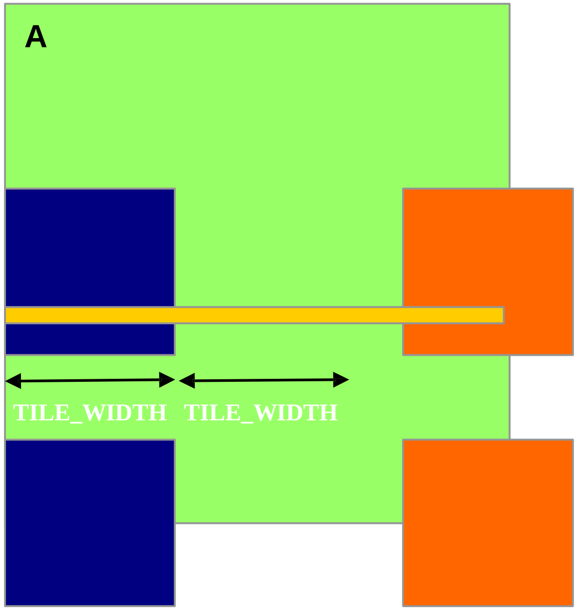
Illustration



The multiply-add will not affect the output due to 0's.

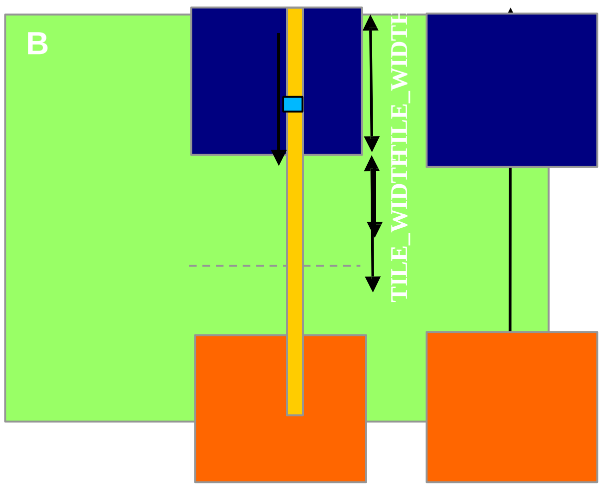
Testing Boundary Condition on A

- Each thread loads
  - A[Row][t*TILE_WIDTH+tx]
  - A[Row*Width + t*TILE_WIDTH+tx]
- Need to test
  - (Row < m) && (t*TILE_WIDTH+tx < n)
  - If true, load A element
  - Else , load 0



Testing Boundary Condition on B

- Each thread loads
  - B[t*TILE_WIDTH+ty][Col]
  - B[(t*TILE_WIDTH+ty)*k+ Col]
- Need to test
  - (t*TILE_WIDTH+ty < n) && (Col< k)
  - If true, load B element
  - Else , load 0



Code: Loading A and B Tiles with Boundary Checks

```
for (int t = 0; t < (n-1)/TILE_WIDTH + 1; ++t) {
++       if(Row < m && t*TILE_WIDTH+tx < n) {
                ds_A[ty][tx] = A[Row*n + t*TILE_WIDTH + tx];
++       } else {
++              ds_A[ty][tx] = 0.0;
++       }
++       if (t*TILE_WIDTH+ty < n && Col < k) {
                ds_B[ty][tx] = B[(t*TILE_WIDTH + ty)*k + Col];
++       } else {
++              ds_B[ty][tx] = 0.0;
++       }
        __syncthreads();
```

Code: Calculate C Values and Store

```
      for (int i = 0; i < TILE_WIDTH; ++i) {
              Cvalue += ds_A[ty][i] * ds_B[i][tx];
      }
      __syncthreads();
    } /* end of outer for loop */
    if (Row < m && Col < k)
          P[Row*k + Col] = Cvalue;
 } /* end of kernel */
```

Summary

- Matrix multiplication is a common computation task in many applications.
- Its parallelization in CUDA can be optimized by tiling and use of shared memory.
- When tiles exceed matrix boundaries, loading the input and storing the result needs to check the boundary conditions.