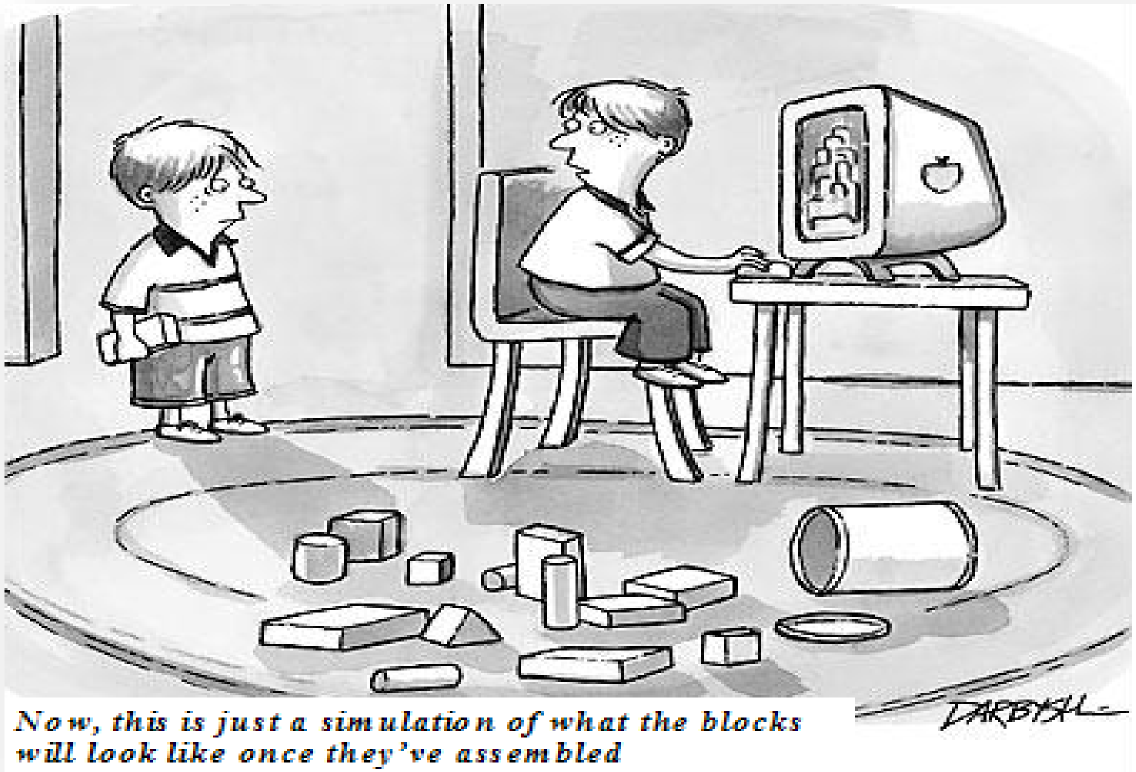
ECE569
Module 24



- Matrix Multiplication Implementation with Tiling

Writing the code for the matrix multiplication

- **To learn to write a tiled matrix-multiplication kernel**
  - Loading and using tiles for matrix multiplication
  - Barrier synchronization, shared memory
  - Resource Considerations
  - Assume that Width is a multiple of tile size for simplicity

Shared Memory and Threading

- For an SM with 48KB shared memory, 2048 threads/SM, 16 Blocks/SM, if the TILE_WIDTH is 16 for a square matrix, what is the maximum number of active thread blocks and which architecture parameter acts as the limiting factor?

- [ ] 16 thread blocks, limitation due to 48KB shared memory
- [ ] 8 thread blocks, limitation due to 16 blocks/SM
- [ ] 8 thread blocks, limitation due to 2048threads/SM
- [ ] 16 thread blocks, limitation due to 2048threads/SM

Barrier Synchronization

- **Synchronize all threads in a block**
  - __syncthreads()
- **All threads in the same block must reach the __syncthreads() before any of the them can move on**
- **Best used to coordinate the phased execution tiled algorithms**
  - To ensure that all elements of a tile are loaded at the beginning of a phase
  - To ensure that all elements of a tile are consumed at the end of a phase

Shared Memory and Threading

- **Each __syncthread() can reduce the number of active threads for a block**
  - More thread blocks can be advantageous
  - Have at least 2 thread blocks!

Loading Input Tile 0 of M (Phase 0)

- Have each thread load an M element and an N element at the same relative position as its P element.

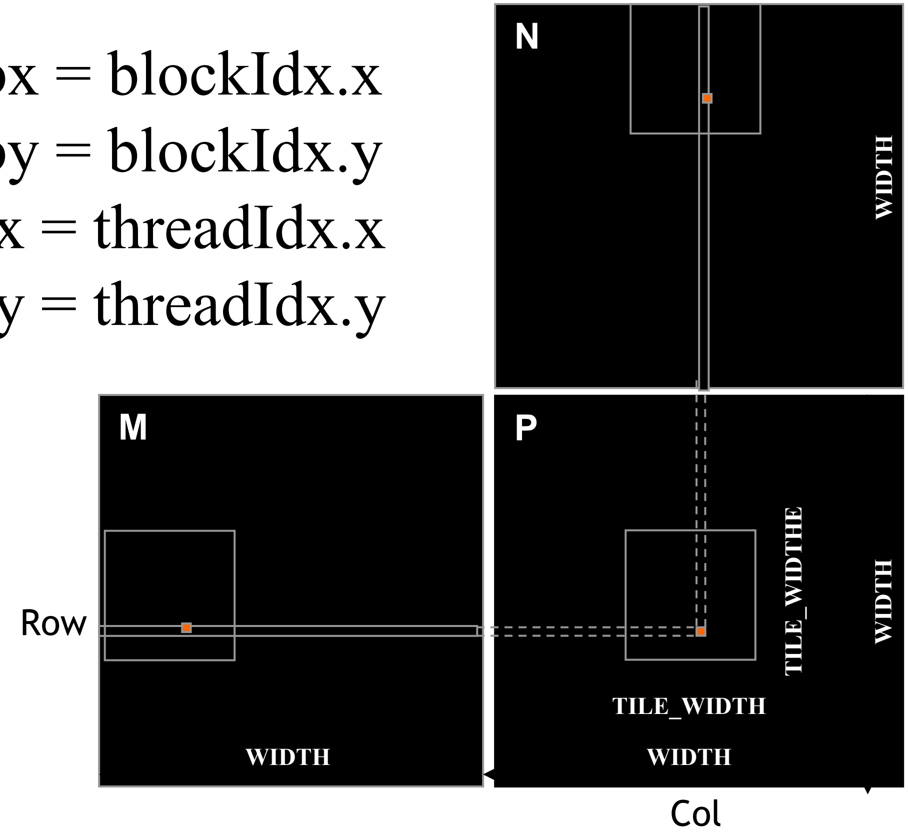
bx = blockIdx.x
by = blockIdx.y
tx = threadIdx.x
ty = threadIdx.y

```
int Row = by * blockDim.y + ty;
int Col =   bx * blockDim.x + tx;
```

2D indexing for accessing Tile 0:

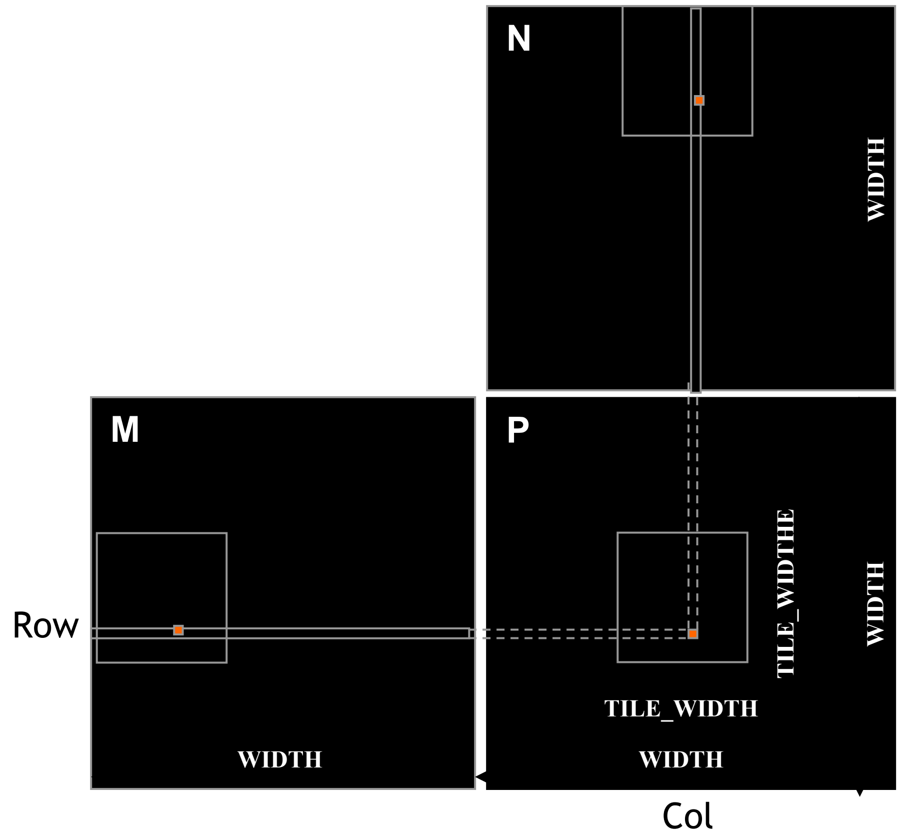
M[???][???]

N[???][???]



Loading Input Tile 0 of M (Phase 0)

- Have each thread load an M element and an N element at the same relative position as its P element.
  - bx = blockIdx.x
  - by = blockIdx.y
  - tx = threadIdx.x
  - ty = threadIdx.y



int Row = by * blockDim.y + ty;
int Col =   bx * blockDim.x + tx;
2D indexing for accessing Tile 0:

M[**Row**][**tx**]

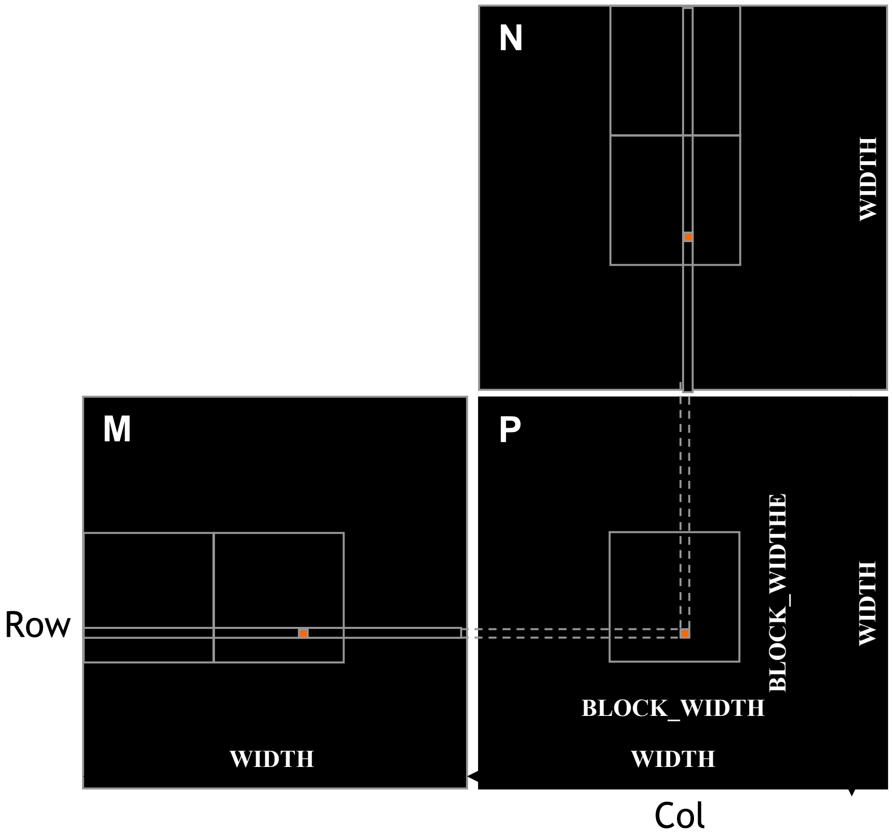
N[**ty**][**Col**]

Loading Input Tile 1 of M (Phase 1)

2D indexing for accessing Tile 1:

M[???][???]

N[???][???]

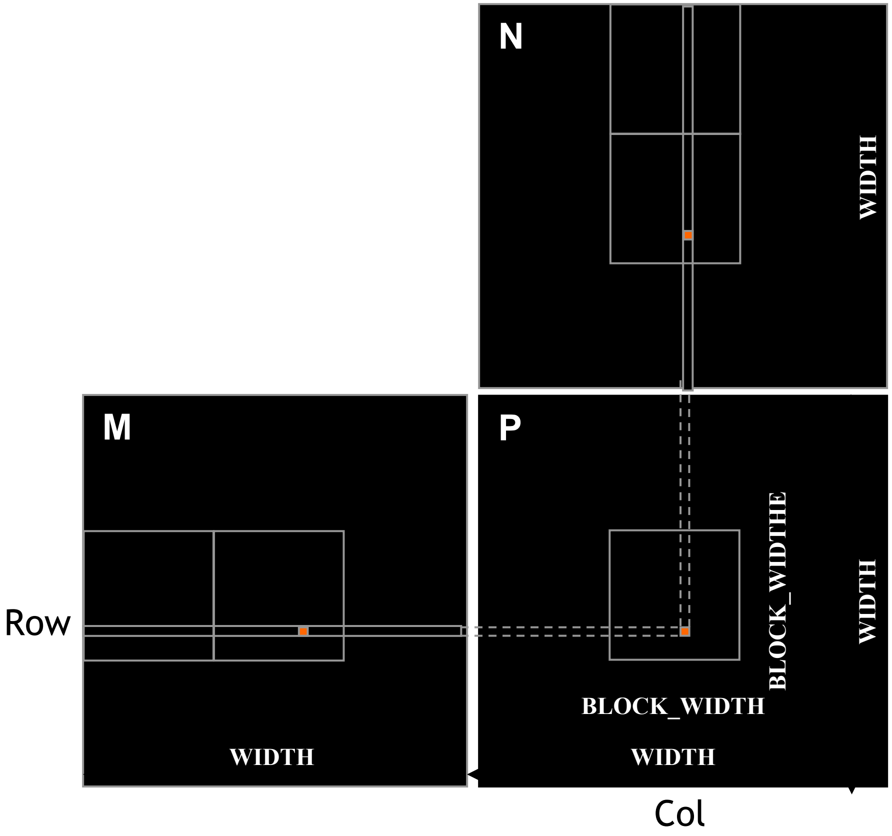


Loading Input Tile 1 of M (Phase 1)

2D indexing for accessing Tile 1:

M[Row][1*TILE_WIDTH + tx]

N[1*TILE_WIDTH + ty][Col]

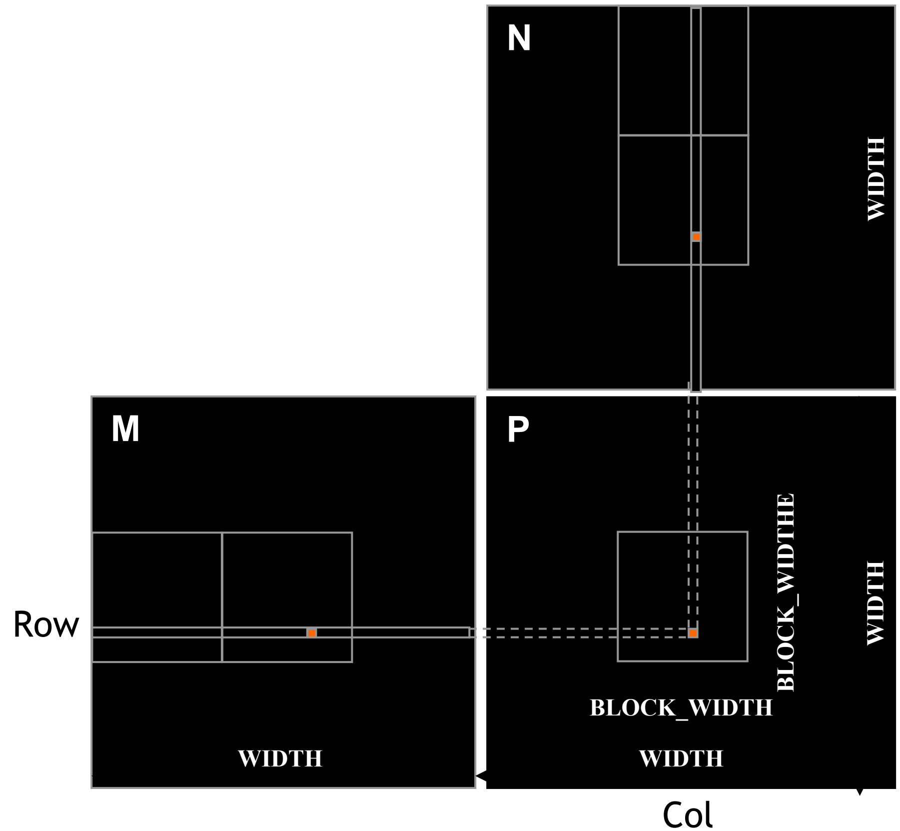


M and N access in phases (p)

2D indexing for accessing Tile 1:

M[Row][1*TILE_WIDTH + tx]
M[???][???]

N[1*TILE_WIDTH + ty][Col]
N[???][???]



M and N access in phases (p)

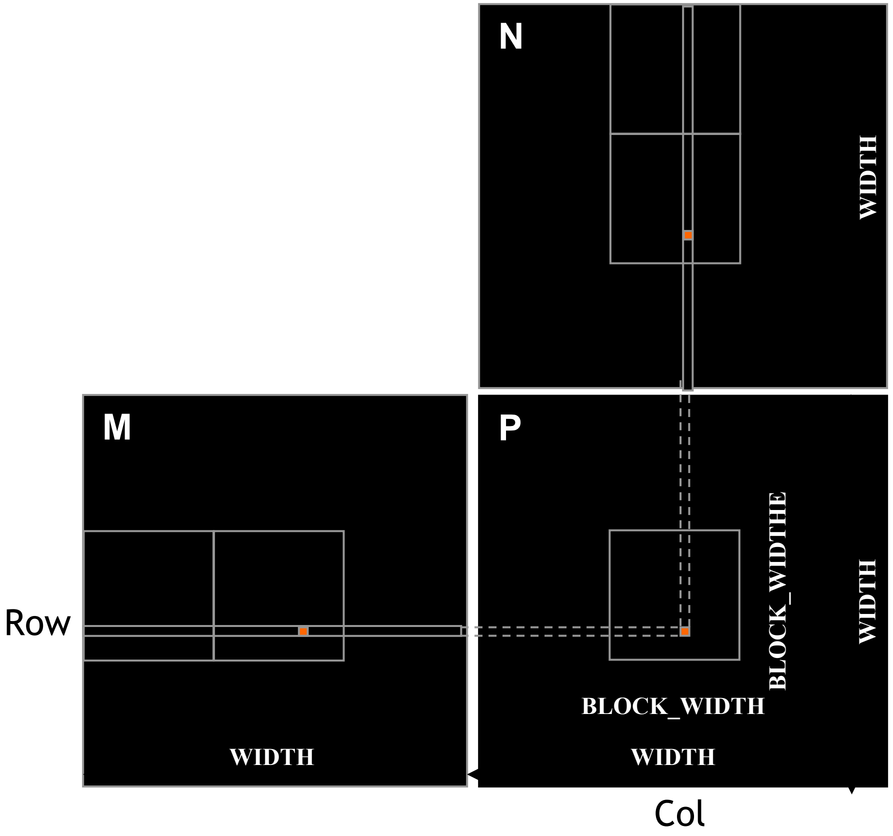
2D indexing for accessing Tile 1:

M[Row][1*TILE_WIDTH + tx]

M[Row][p*TILE_WIDTH+tx]

N[1*TILE_WIDTH + ty][Col]

N[p*TILE_WIDTH + ty][Col]



where p is the sequence number of the current phase

M and N are dynamically allocated - use 1D indexing

M[Row][p*TILE_WIDTH+tx]
M[Row*Width + p*TILE_WIDTH + tx]

N[p*TILE_WIDTH+ty][Col]
N[(p*TILE_WIDTH+ty)*Width + Col]

where p is the sequence number of the current phase

Tiled Matrix Multiplication Kernel (Declarations)

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
```

- **TILE_WIDTH: Known at compile time so we can use 2D notation**

```
  // declare shared memory for a tile of M and N
  // assume TILE_WIDTH known at compile time
  // shared memory can be accessed with 2d notation!

   // declare and assign values for bx, by, tx, ty, Row, Col

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

  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;
```

Tiled Matrix Multiplication Kernel

- **TILE_WIDTH: Known at compile time so we can use 2D notation**

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
  __shared__  float ds_M[TILE_WIDTH][TILE_WIDTH];
  __shared__  float ds_N[TILE_WIDTH][TILE_WIDTH];
  int bx = blockIdx.x;  int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;

  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;
  float Pvalue = 0;

 // Loop over to read a tile of M and N into the
 // shared memory in each iteration

}
```

Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
  __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
  __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];
  int bx = blockIdx.x;  int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;

  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;
  float Pvalue = 0;

 // Loop over the M and N tiles required to compute the P element
 for (int p = 0; p < Width/TILE_WIDTH; ++p) {
    // Collaborative loading of M and N tiles into shared memory
    ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
    ds_N[ty][tx] = N[(p*TILE_WIDTH+ty)*Width + Col];
    // Process on tile of data from shared memory for partial product

}
```

- **TILE_WIDTH: Known at compile time so we can use 2D notation**

Did we miss something?

- **TILE_WIDTH: Known at compile time so we can use 2D notation**

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
  __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
  __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];
  int bx = blockIdx.x;  int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;

  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;
  float Pvalue = 0;

 // Loop over the M and N tiles required to compute the P element
 for (int p = 0; p < n/TILE_WIDTH; ++p) {
    // Collaborative loading of M and N tiles into shared memory
    ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
    ds_N[ty][tx] = N[(p*TILE_WIDTH+ty)*Width + Col];

    for (int i = 0; i < TILE_WIDTH; ++i)
        Pvalue += ds_M[ty][i] * ds_N[i][tx];

  }
  P[Row*Width+Col] = Pvalue;
}
```

Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
  __shared__  float ds_M[TILE_WIDTH][TILE_WIDTH];
  __shared__  float ds_N[TILE_WIDTH][TILE_WIDTH];
  int bx = blockIdx.x;  int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;

  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;
  float Pvalue = 0;

 // Loop over the M and N tiles required to compute the P element
 for (int p = 0; p < n/TILE_WIDTH; ++p) {
    // Collaborative loading of M and N tiles into shared memory
    ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
    ds_N[ty][tx] = N[(p*TILE_WIDTH+ty)*Width + Col];
    __syncthreads();

    for (int i = 0; i < TILE_WIDTH; ++i)
      Pvalue += ds_M[ty][i] * ds_N[i][tx];
    __synchthreads();
  }
  P[Row*Width+Col] = Pvalue;
}
```

ensures that all threads have finished loading the tiles of M and N

ensures that all threads have finished processing the tile in shared memory before loading the next tile into the shared memory

Tile Size Considerations

- **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, in each phase, each block performs**
  - 2*256 = 512 float loads (M,N) from global memory
  - 256 * [(1 mult + 1 add per element)*16] = 8,192 mul/add operations. (16 floating-point operations for each memory load)

- **For 32, in each phase, each block performs**
  - 2*1024 = 2048 float loads from global memory
  - 1024 * (2*32) = 65,536 mul/add operations. (32 floating-point operation for each memory load)

Next: Handling Matrix of Arbitrary Size

- **The tiled matrix multiplication kernel we presented so far can handle only square matrices whose dimensions (Width) are multiples of the tile width (TILE_WIDTH)**
  – However, real applications need to handle arbitrary sized matrices.
  – One could pad (add elements to) the rows and columns into multiples of the tile size, but would have significant space and data transfer time overhead.
- **We will take a different approach.**