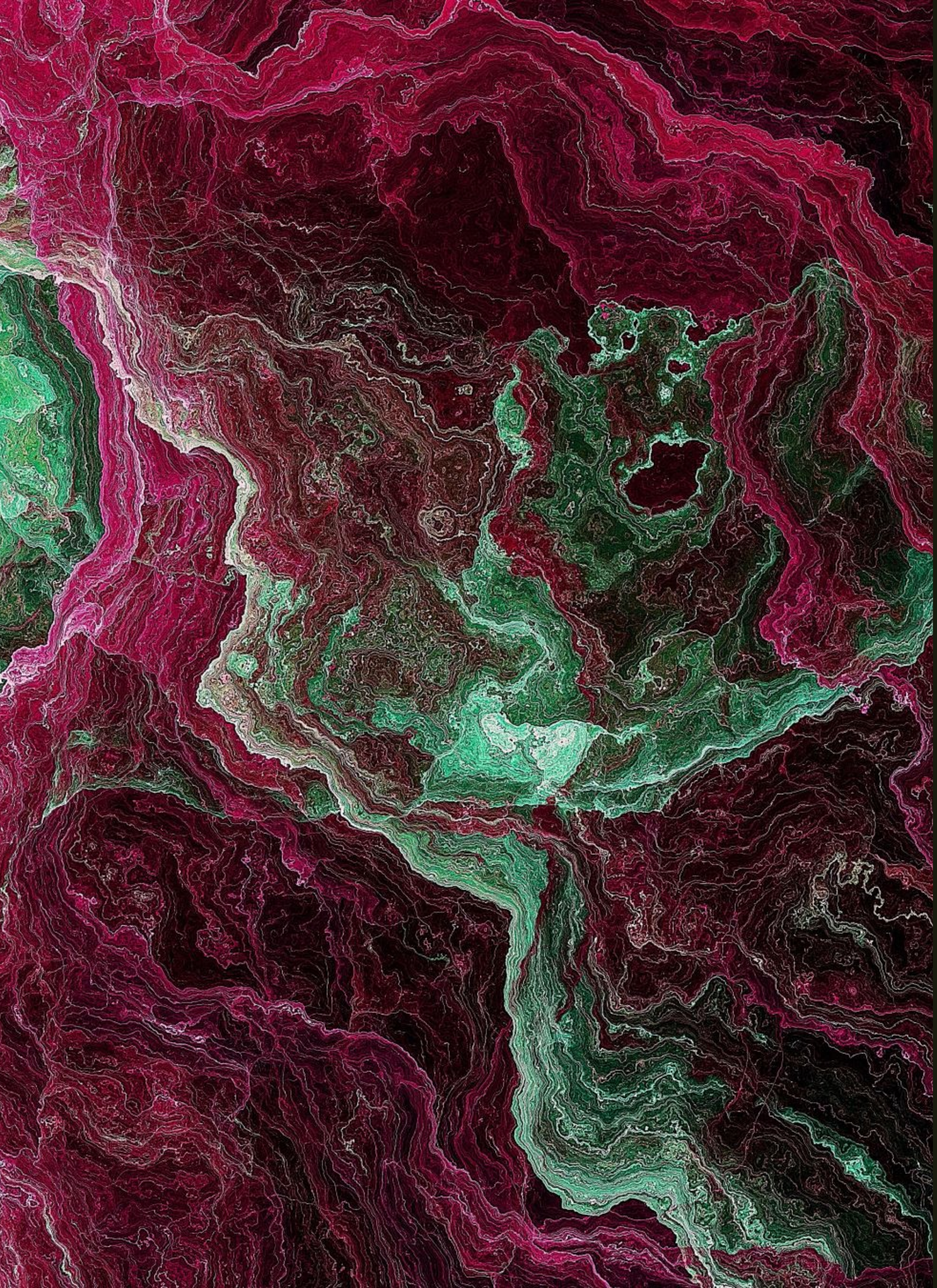


GPU: MEMORY MODEL - TILING

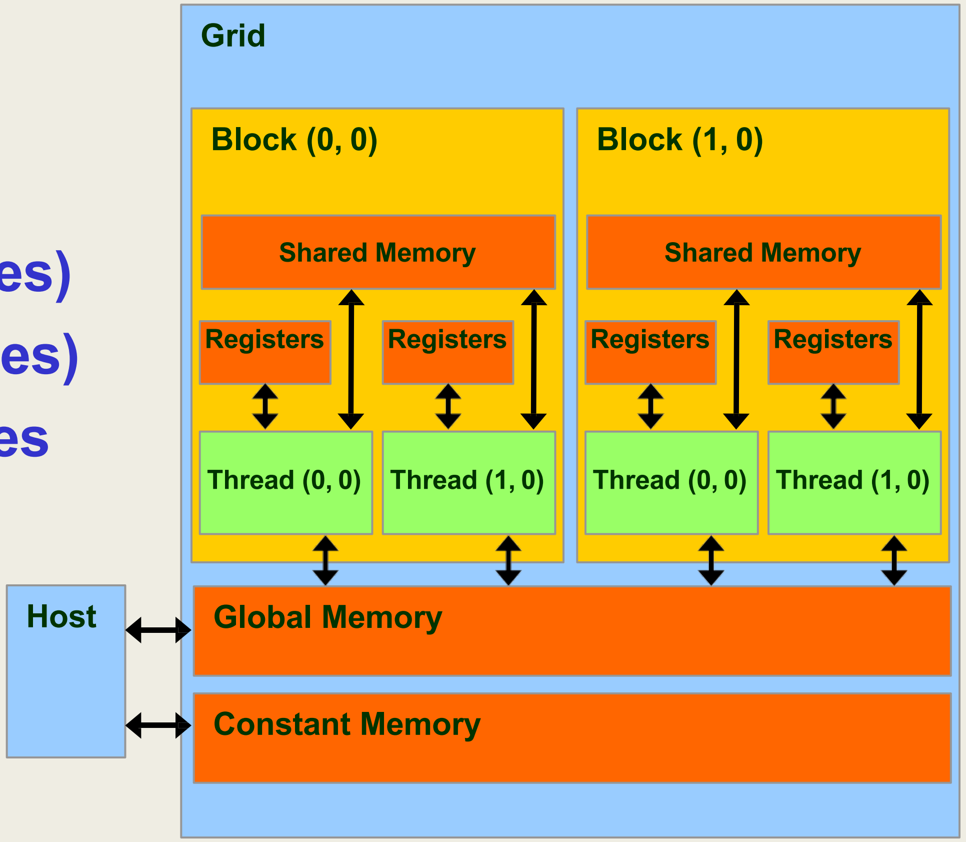
Dr. Steve Petruzza

UtahStateUniversity

Programmer View of CUDA Memories

Each thread can:

- Read/write per-thread **registers (~1 cycle)**
- Read/write per-block **shared memory (~5 cycles)**
- Read/write per-grid **global memory (~500 cycles)**
- Read/only per-grid **constant memory (~5 cycles with caching)**



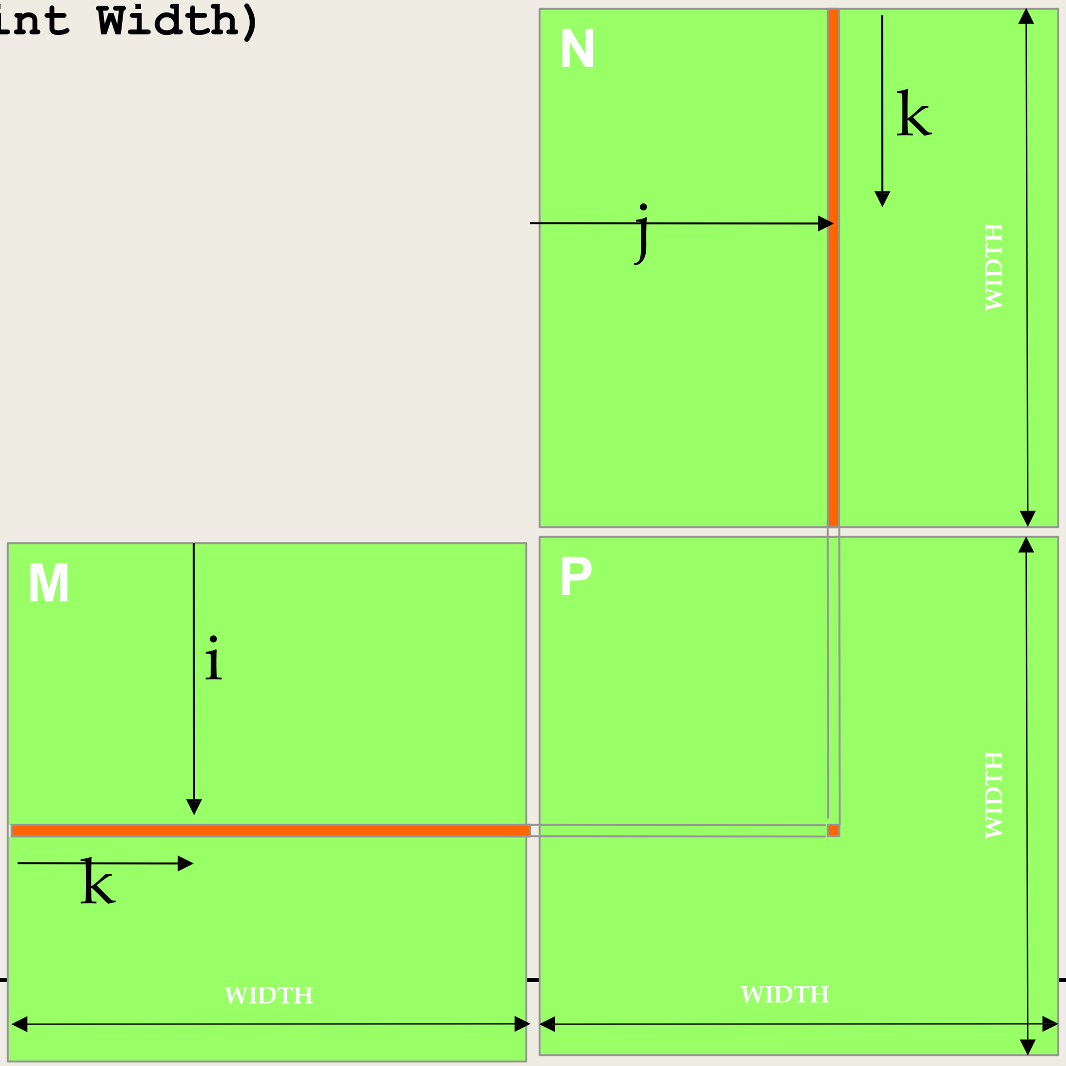
CUDA Variable Type Qualifiers

Variable declaration	Memory	Scope	Lifetime
`int LocalVar;`	register	thread	thread
`__device__ __shared__ int SharedVar;`	shared	block	block
`__device__ int GlobalVar;`	global	grid	application
`__device__ __constant__ int ConstantVar;`	constant	grid	application

- `__device__` is optional when used with `__shared__`, or `__constant__`
- Automatic variables without any qualifier reside in a register
  - Except per-thread arrays that reside in global memory

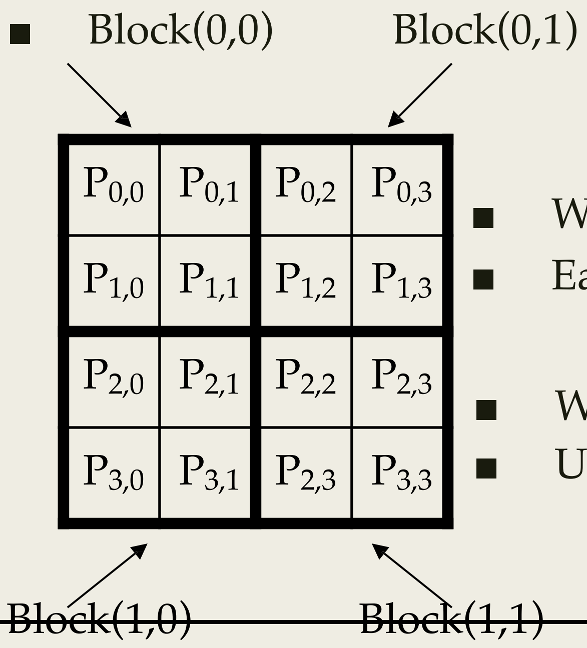
Matrix Multiplication-- Simple CPU Version

```
// Matrix multiplication on the (CPU) host in single precision
void MatrixMul(float *M, float *N, float *P, int Width)
{
    for (int i = 0; i < Width; ++i)
        for (int j = 0; j < Width; ++j) {
            float sum = 0;
            for (int k = 0; k < Width; ++k) {
                float a = M[i * Width + k];
                float b = N[k * Width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
}
```



Kernel Function - A Small Example

- Have each 2D thread block to compute a $(\text{BLOCK\_WIDTH})^2$ sub-matrix of the result matrix
  - Each block has $(\text{BLOCK\_WIDTH})^2$ threads
- Generate a 2D Grid of $(\text{WIDTH/BLOCK\_WIDTH})^2$ blocks
- This concept is called **tiling**. Each block represents a **tile.**

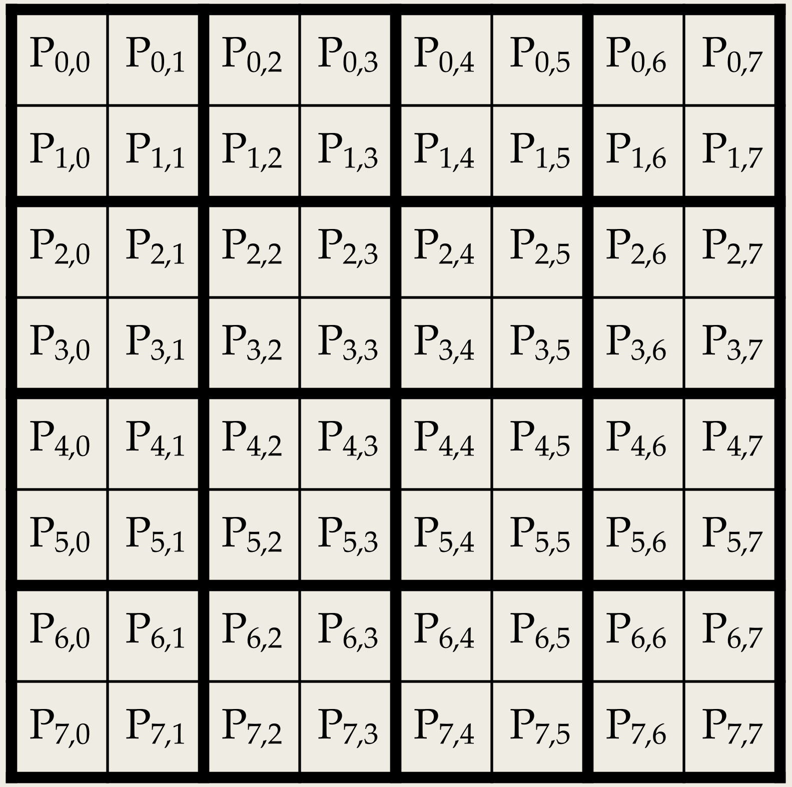


Block(0,0)
Block(0,1)
$P_{0,0}$ $P_{0,1}$ $P_{0,2}$ $P_{0,3}$
$P_{1,0}$ $P_{1,1}$ $P_{1,2}$ $P_{1,3}$
$P_{2,0}$ $P_{2,1}$ $P_{2,2}$ $P_{2,3}$
$P_{3,0}$ $P_{3,1}$ $P_{2,3}$ $P_{3,3}$
Block(1,0)
Block(1,1)

- WIDTH = 4; BLOCK_WIDTH = 2
- Each block has 2*2 = 4 threads

- WIDTH/BLOCK_WIDTH = 2
- Use 2* 2 = 4 blocks

A Slightly Bigger Example (BLOCK_WIDTH =2)



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

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

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

A Slightly Bigger Example (cont.)
(BLOCK_WIDTH = 4)

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

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

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

Kernel Invocation (Host-side Code)

```
// Setup the execution configuration
// BLOCK_WIDTH is a #define constant
dim3 dimGrid(ceil((1.0*Width)/BLOCK_WIDTH),
             ceil((1.0*Width)/BLOCK_WIDTH), 1);

dim3 dimBlock(BLOCK_WIDTH, BLOCK_WIDTH, 1);


// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

Kernel Function

```
// Matrix multiplication kernel – per thread code

__global__
void MatrixMulKernel(float *d_M, float *d_N, float *d_P, int Width)
{

    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;
```

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

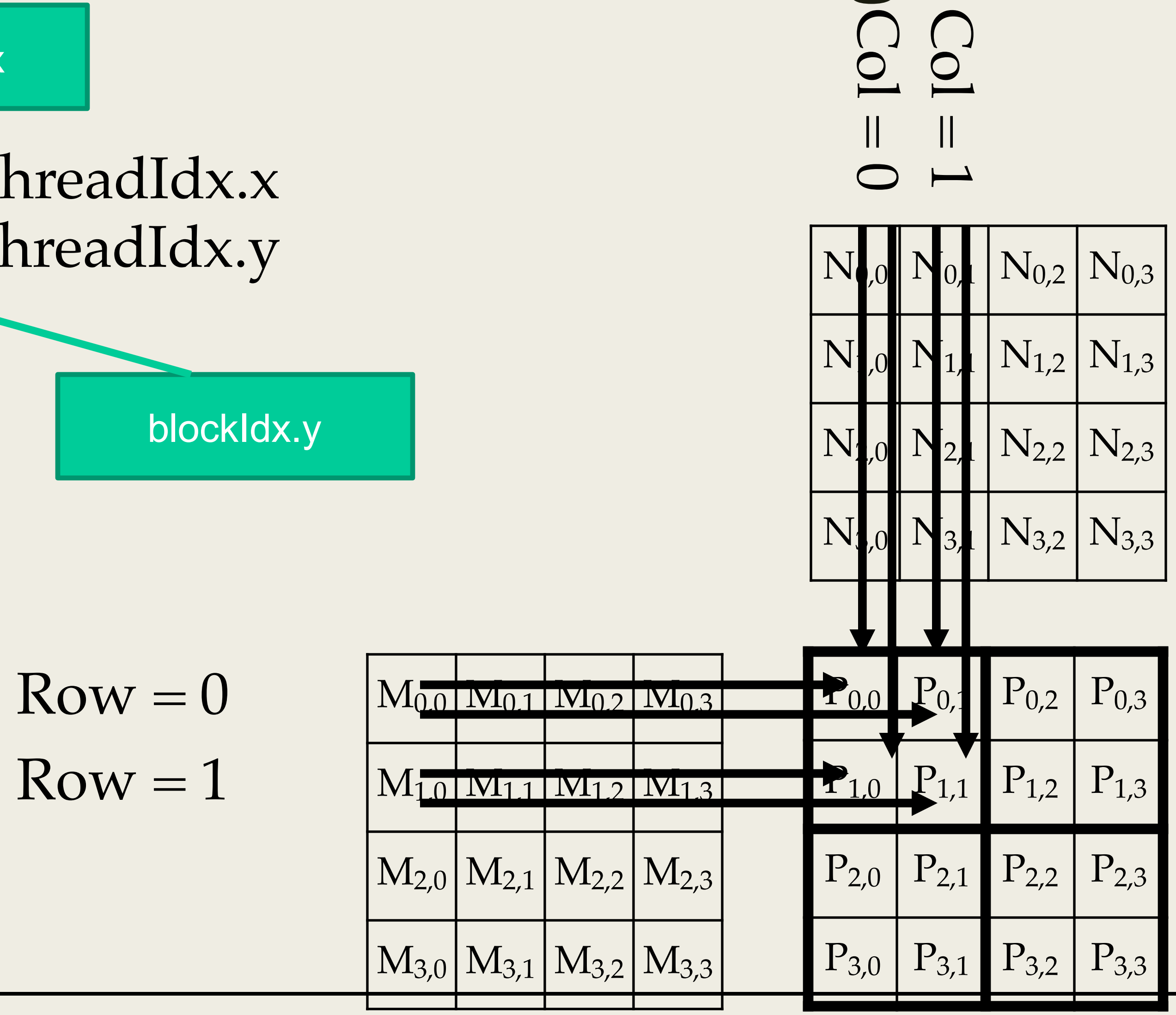
blockIdx.x

blockDim.x

$$\text{Col} = 0 * 2 + \text{threadIdx.x}$$
$$\text{Row} = 0 * 2 + \text{threadIdx.y}$$

blockDim.y

blockIdx.y



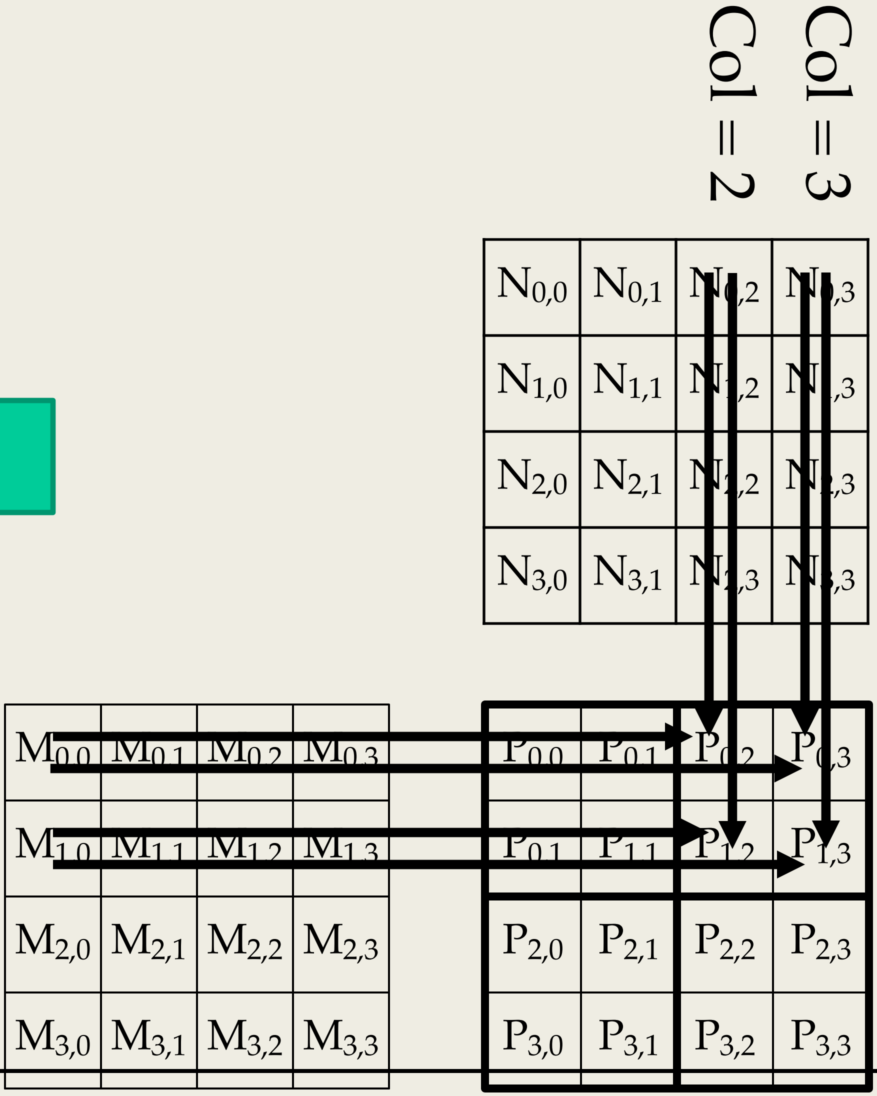
Col = 0
Col = 1
Row = 0
Row = 1
$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$
$N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$
$N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$
$N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
$P_{0,0}$ $P_{0,1}$ $P_{0,2}$ $P_{0,3}$
$P_{1,0}$ $P_{1,1}$ $P_{1,2}$ $P_{1,3}$
$P_{2,0}$ $P_{2,1}$ $P_{2,2}$ $P_{2,3}$
$P_{3,0}$ $P_{3,1}$ $P_{3,2}$ $P_{3,3}$

Work for Block (0,1)

blockIdx.x

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

blockIdx.y

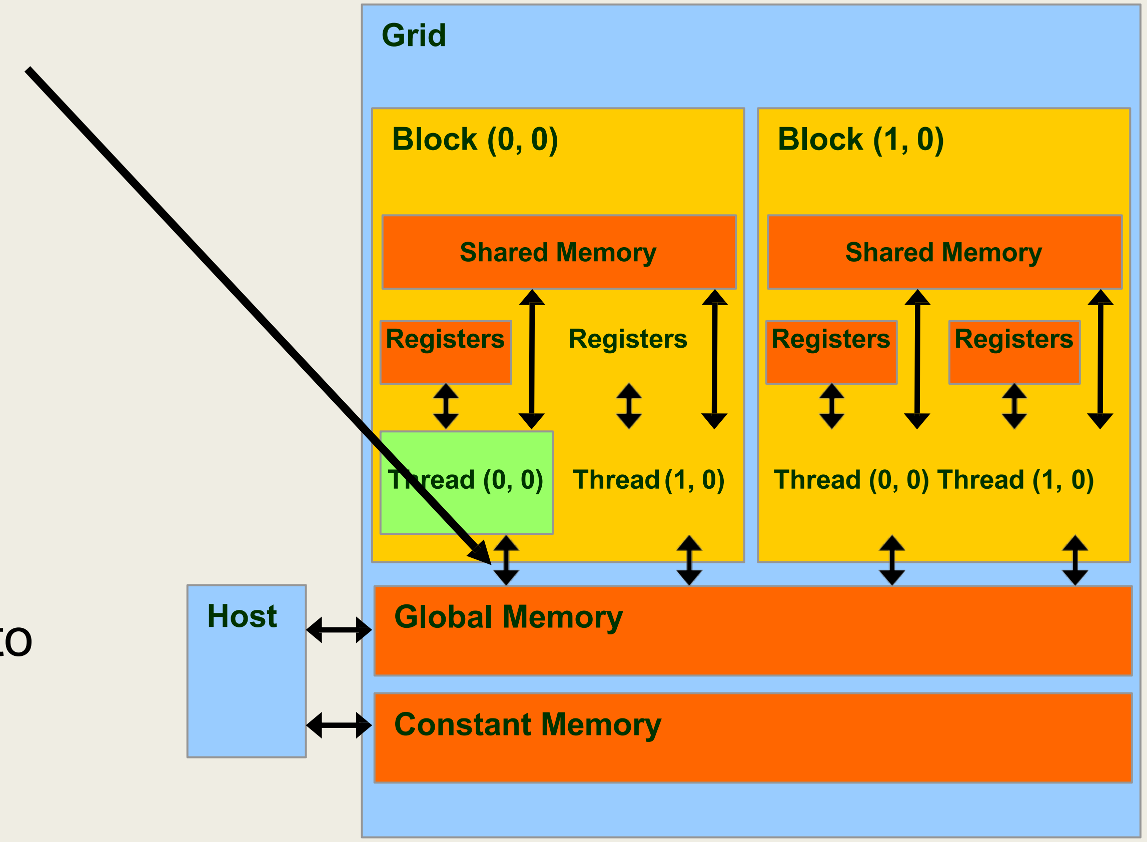


A Simple Matrix Multiplication Kernel

```
__global__
void MatrixMulKernel(float *d_M, float *d_N, float *d_P, int Width)
{
   // Calculate the row index of the d_P element and d_M
   int Row = blockIdx.y*blockDim.y+threadIdx.y;

   // Calculate the column idenx of d_P and d_N
   int Col = blockIdx.x*blockDim.x+threadIdx.x;

   if ((Row < Width) && (Col < Width)) {
       float Pvalue = 0;
      // each thread computes one element of the block sub-matrix
      for (int k = 0; k < Width; ++k)
          Pvalue += d_M[Row*Width+k] * d_N[k*Width+Col];
      d_P[Row*Width+Col] = Pvalue;
   }
}
```

How about performance on a device with 150 GB/s memory bandwidth?

- All threads access global memory for their input matrix elements
  - Two memory accesses (8 bytes) per floating point multiply-add (2 fp ops)
  - 4B/s of memory bandwidth/FLOPS
  - 150 GB/s limits the code at 37.5 GFLOPS
- The actual code runs at about 25 GFLOPS
- Need to drastically cut down memory accesses to get closer to the peak of more than 1,000 GFLOPS



What would happen instead on a CPU?

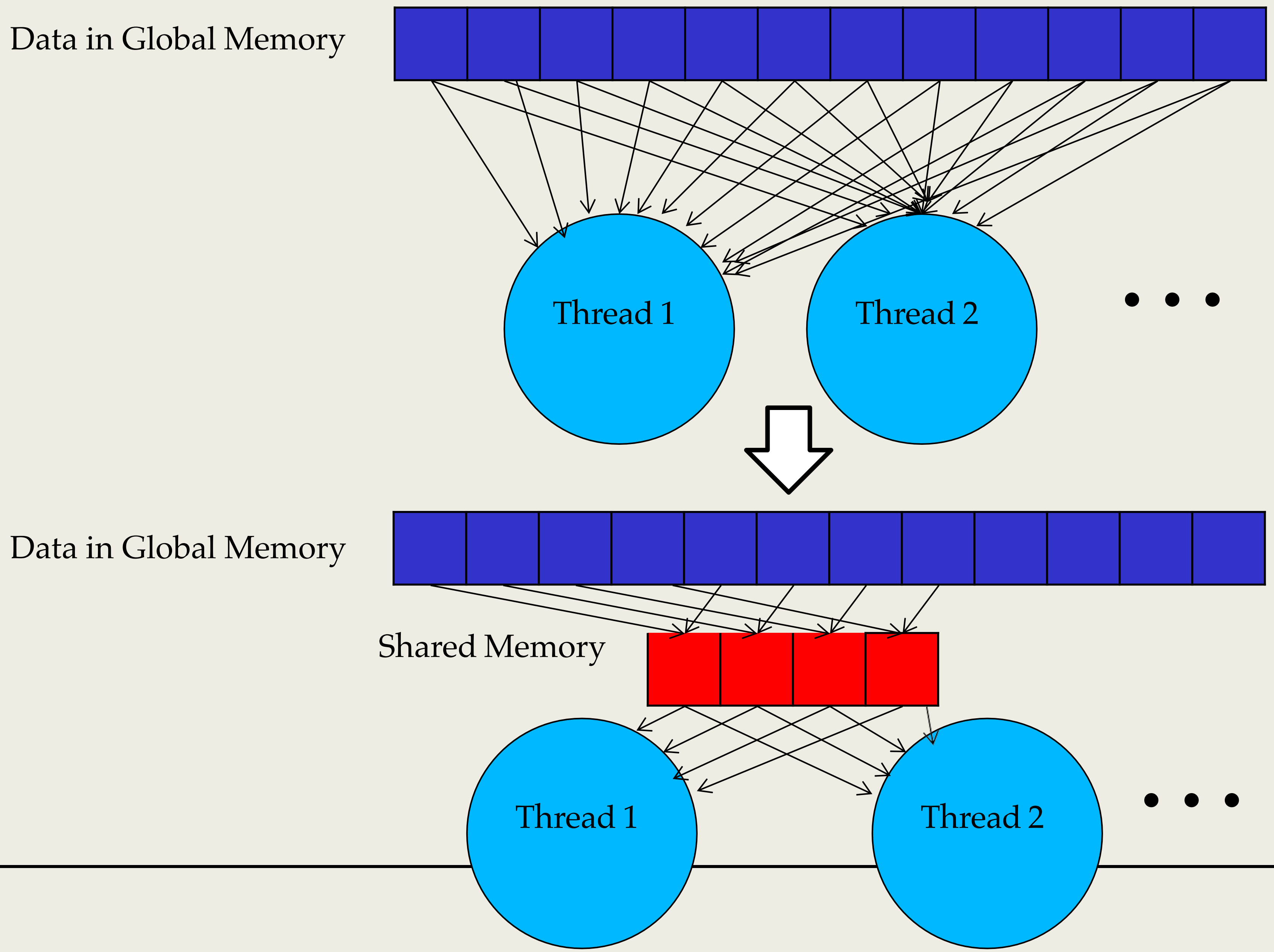
A Common Programming Strategy

- Global memory is implemented with DRAM - slow
- To avoid Global Memory bottleneck, tile the input data to take advantage of Shared Memory:
  - Partition data into subsets (tiles) that fit into the (smaller but faster) shared memory
  - Handle each data subset with one thread block by:
    - Loading the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism
    - Performing the computation on the subset from shared memory; each thread can efficiently access any data element
    - Copying results from shared memory to global memory
  - Tiles are also called blocks in the literature

Declaring Shared Memory Arrays

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
    __shared__ float subTileM[TILE_WIDTH][TILE_WIDTH];
    __shared__ float subTileN[TILE_WIDTH][TILE_WIDTH];
```

Shared Memory Tiling Basic Idea



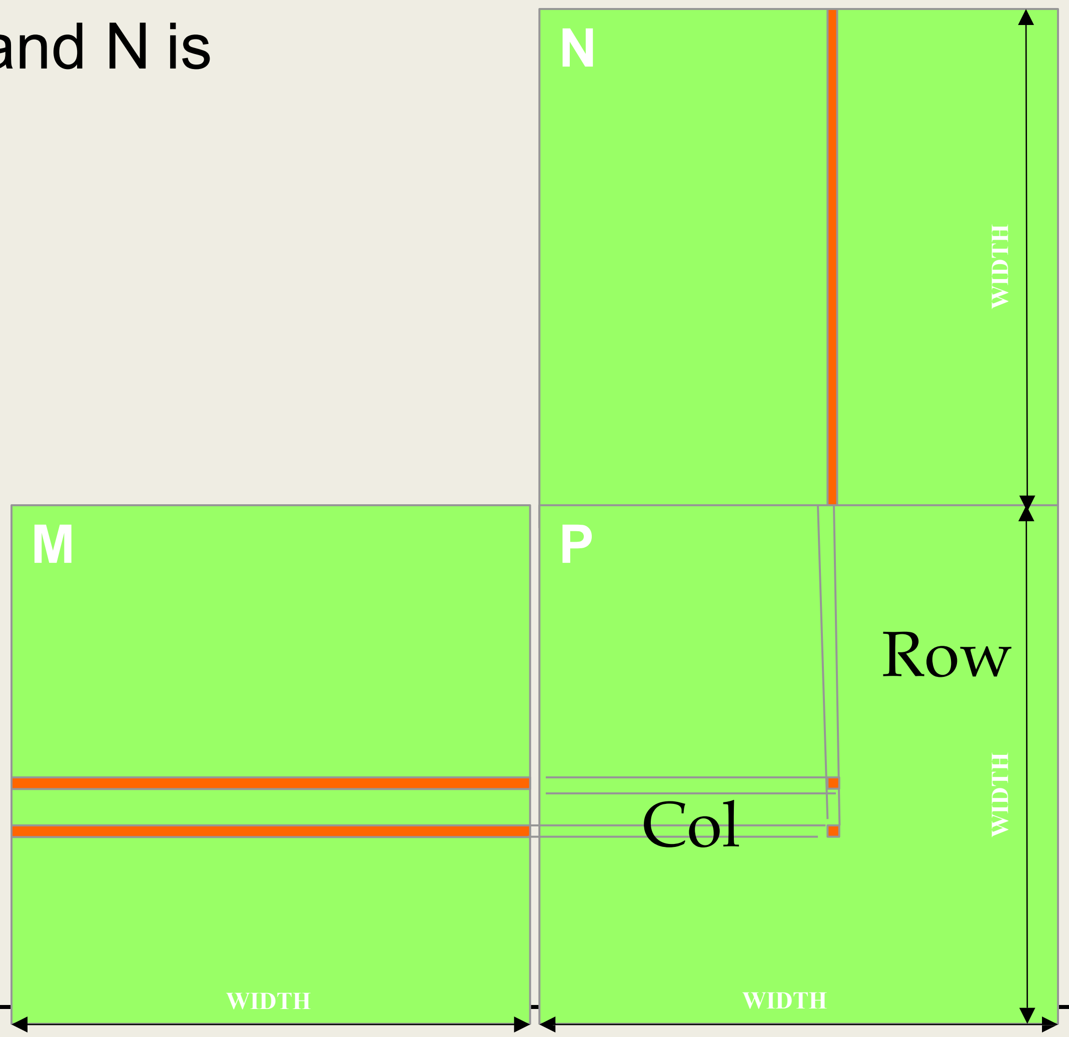
Outline of Technique

- Identify a tile of global data 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 block/tile

Use Shared Memory for data that will be reused

- Observe that each input element of M and N is
  - used WIDTH times

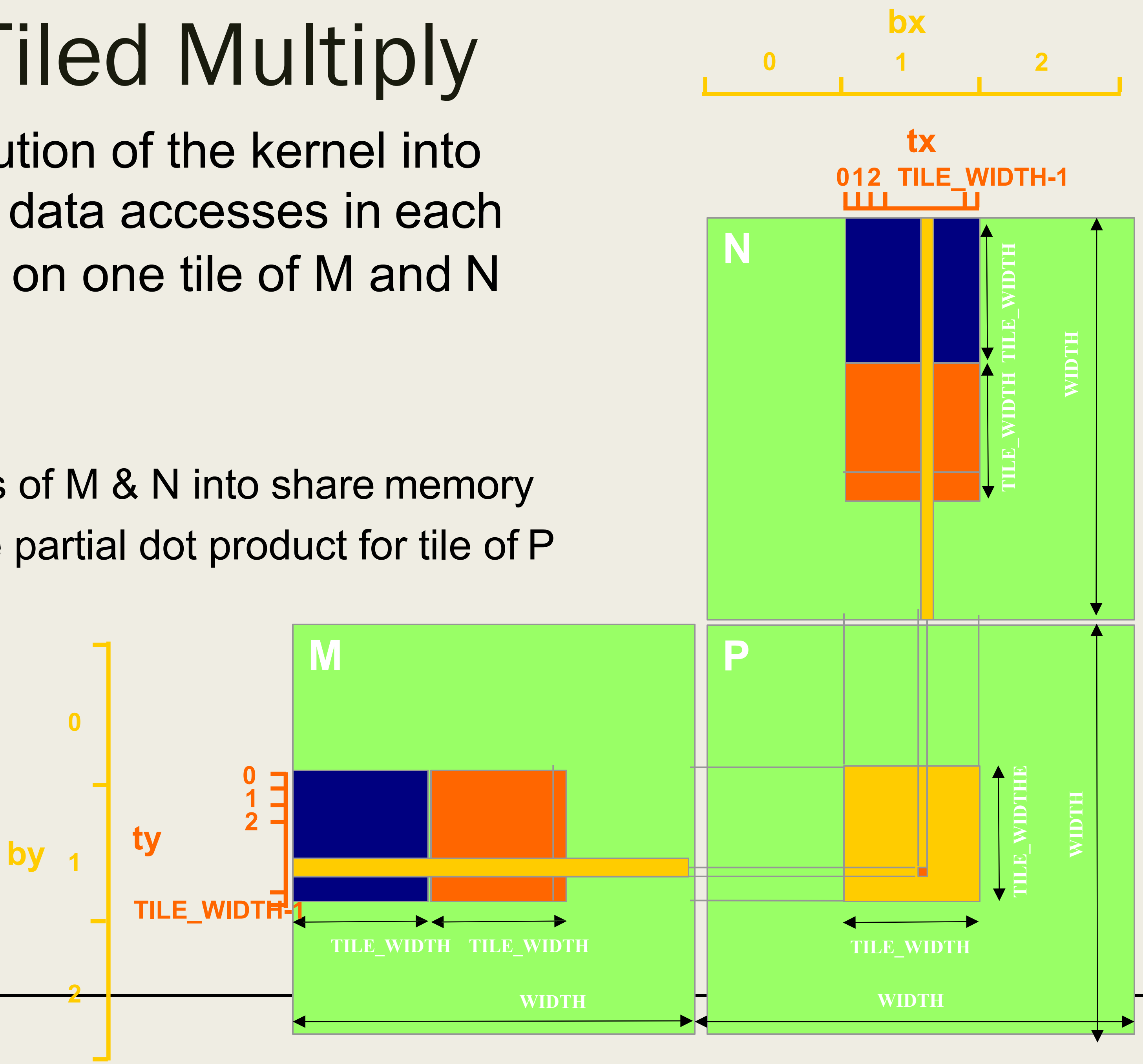
- Load each element into Shared Memory and have several threads use the local version to reduce the memory bandwidth



Tiled Multiply

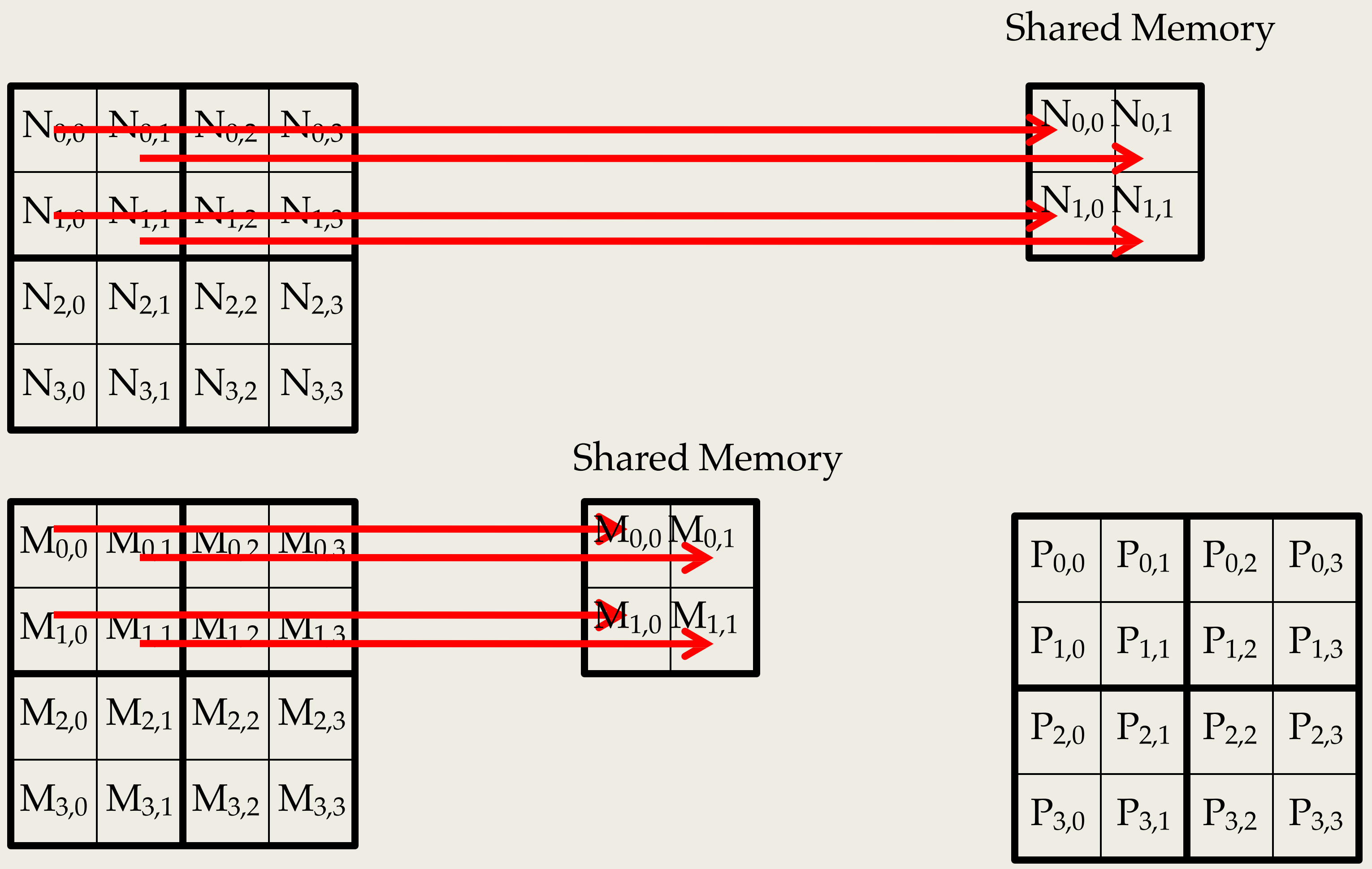
- Break up the execution of the kernel into phases so that the data accesses in each phase are focused on one tile of M and N

- For each tile:
  - Phase 1: Load tiles of M & N into share memory
  - Phase 2: Calculate partial dot product for tile of P



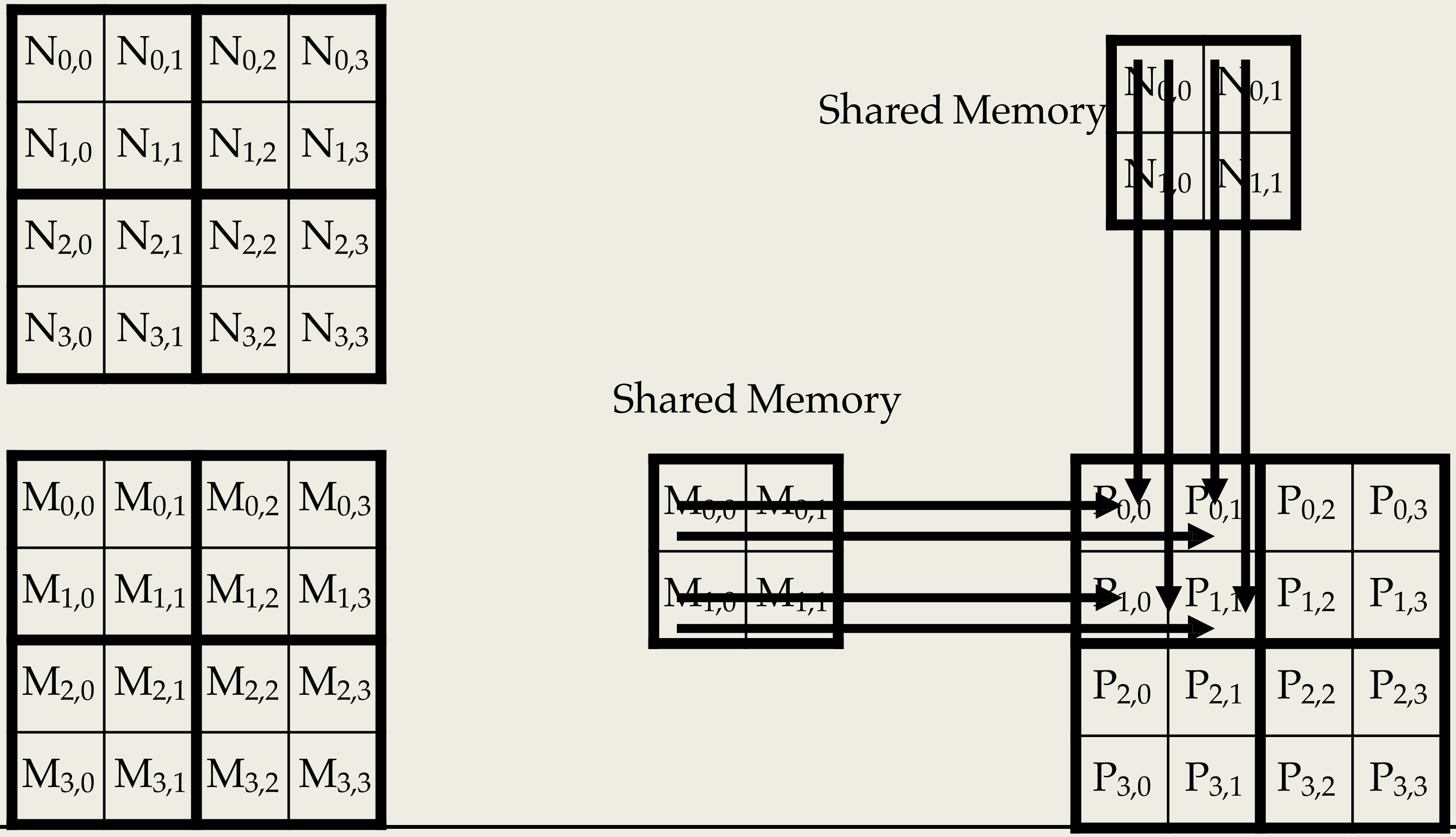
Work for Block (0,0)

Step 0


Shared Memory
$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$
$N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$
$N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$
$N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$
$N_{0,0}$ $N_{0,1}$
$N_{1,0}$ $N_{1,1}$
Shared Memory
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
$M_{0,0}$ $M_{0,1}$
$M_{1,0}$ $M_{1,1}$
$P_{0,0}$ $P_{0,1}$ $P_{0,2}$ $P_{0,3}$
$P_{1,0}$ $P_{1,1}$ $P_{1,2}$ $P_{1,3}$
$P_{2,0}$ $P_{2,1}$ $P_{2,2}$ $P_{2,3}$
$P_{3,0}$ $P_{3,1}$ $P_{3,2}$ $P_{3,3}$

Work for Block (0,0)

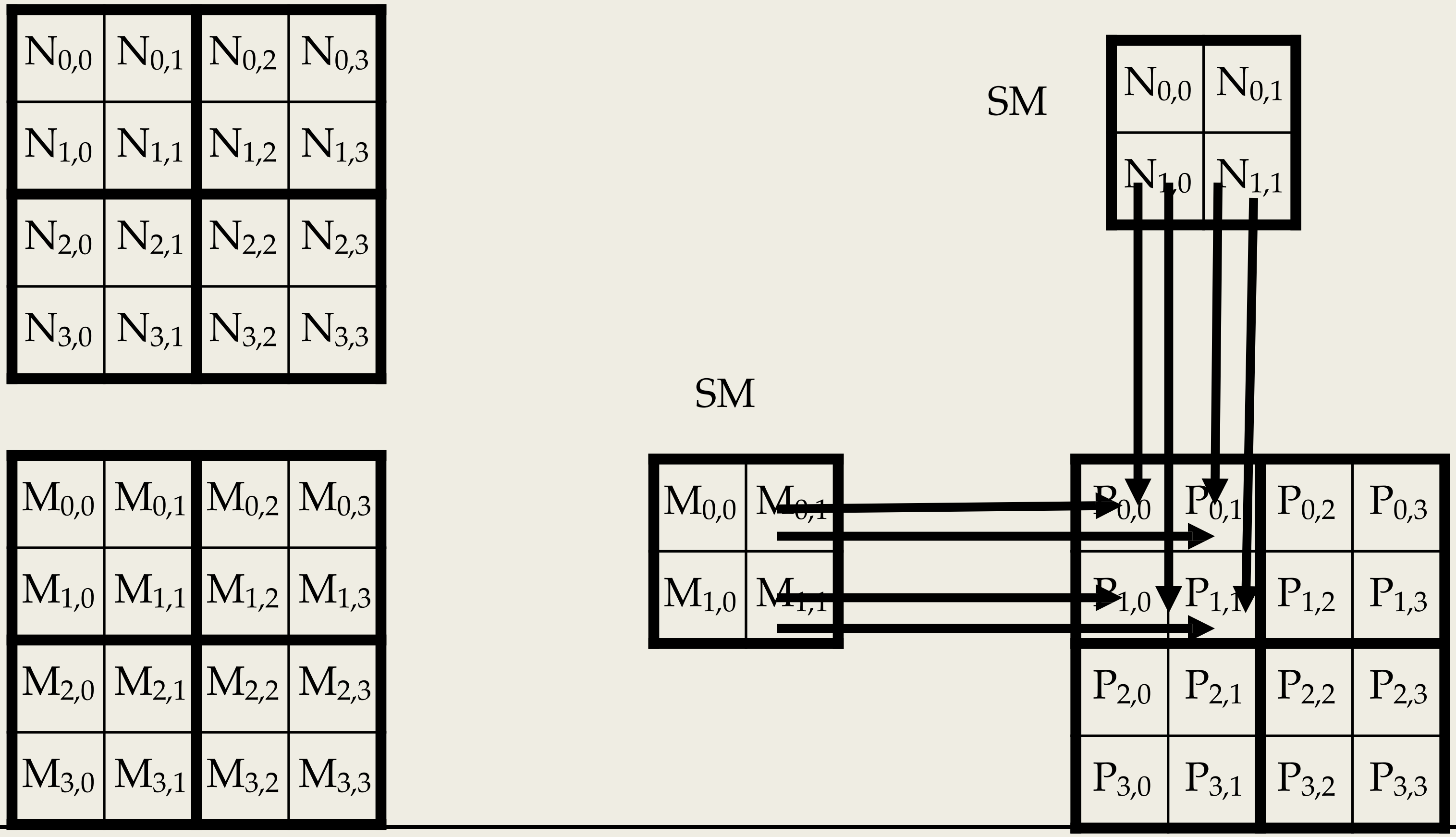
Step 1



$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$
$N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$
$N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$
$N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
Shared Memory
$N_{0,0}$ $N_{0,1}$
$N_{1,0}$ $N_{1,1}$
Shared Memory
$M_{0,0}$ $M_{0,1}$
$M_{1,0}$ $M_{1,1}$
$P_{0,0}$ $P_{0,1}$ $P_{0,2}$ $P_{0,3}$
$P_{1,0}$ $P_{1,1}$ $P_{1,2}$ $P_{1,3}$
$P_{2,0}$ $P_{2,1}$ $P_{2,2}$ $P_{2,3}$
$P_{3,0}$ $P_{3,1}$ $P_{3,2}$ $P_{3,3}$

Work for Block (0,0)

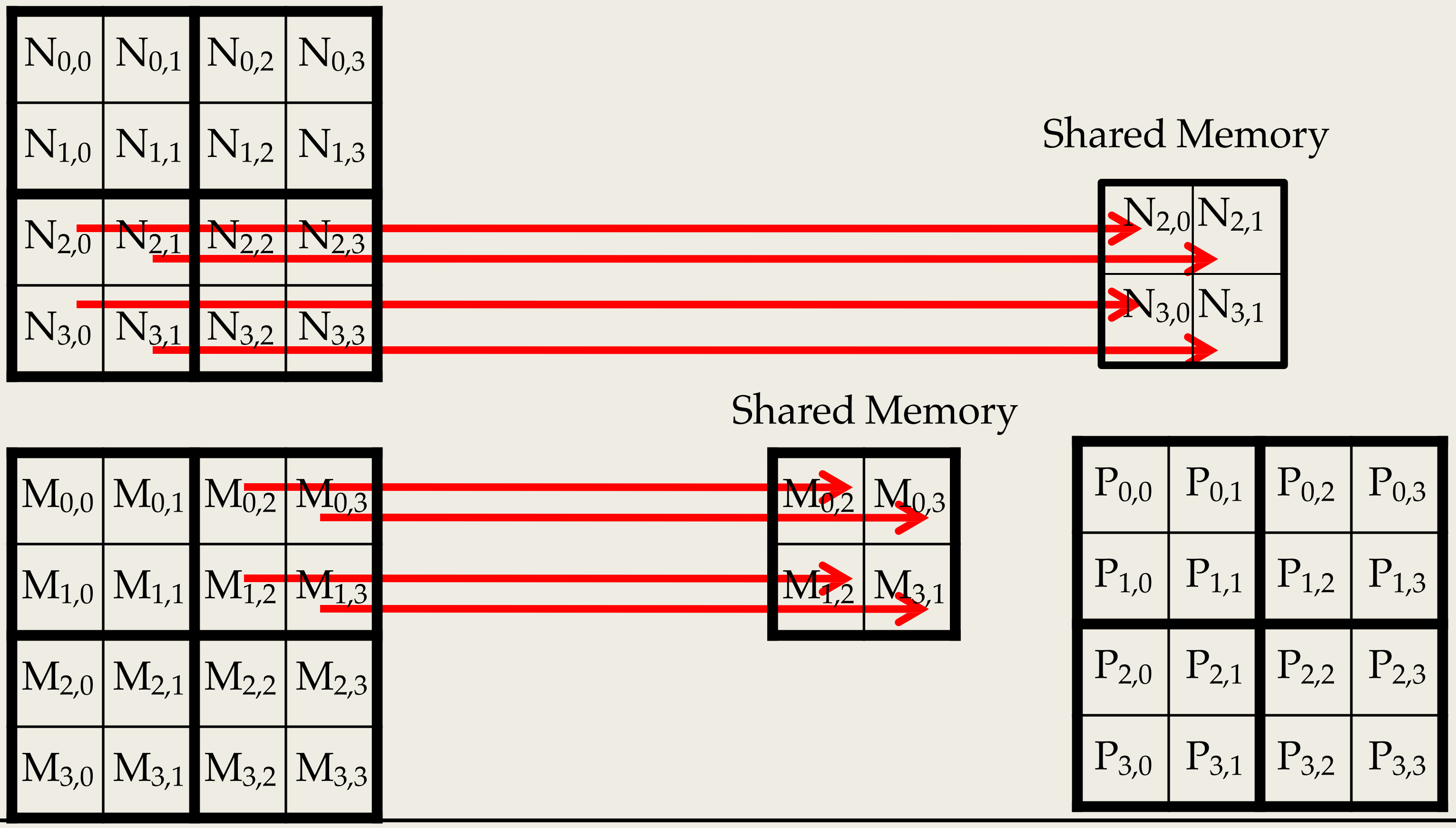
Step 2



$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$
$N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$
$N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$
$N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
SM
$N_{0,0}$ $N_{0,1}$
$N_{1,0}$ $N_{1,1}$
SM
$M_{0,0}$ $M_{0,1}$
$M_{1,0}$ $M_{1,1}$
$P_{0,0}$ $P_{0,1}$ $P_{0,2}$ $P_{0,3}$
$P_{1,0}$ $P_{1,1}$ $P_{1,2}$ $P_{1,3}$
$P_{2,0}$ $P_{2,1}$ $P_{2,2}$ $P_{2,3}$
$P_{3,0}$ $P_{3,1}$ $P_{3,2}$ $P_{3,3}$

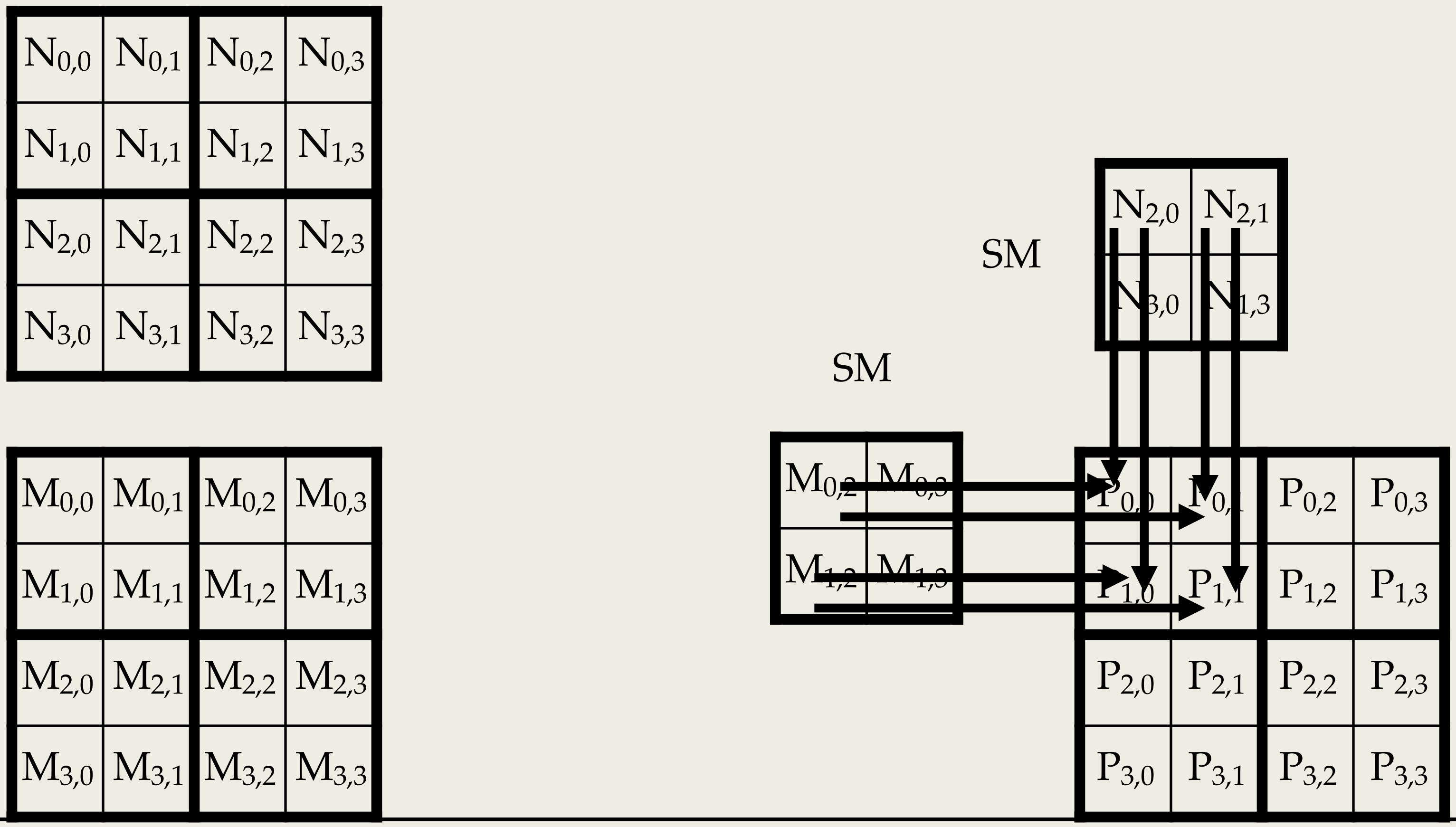
Work for Block (0,0)

Step 3


$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$
$N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$
$N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$
$N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$
Shared Memory
$N_{2,0}$ $N_{2,1}$
$N_{3,0}$ $N_{3,1}$
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
Shared Memory
$M_{0,2}$ $M_{0,3}$
$M_{1,2}$ $M_{3,1}$
$P_{0,0}$ $P_{0,1}$ $P_{0,2}$ $P_{0,3}$
$P_{1,0}$ $P_{1,1}$ $P_{1,2}$ $P_{1,3}$
$P_{2,0}$ $P_{2,1}$ $P_{2,2}$ $P_{2,3}$
$P_{3,0}$ $P_{3,1}$ $P_{3,2}$ $P_{3,3}$

Work for Block (0,0)

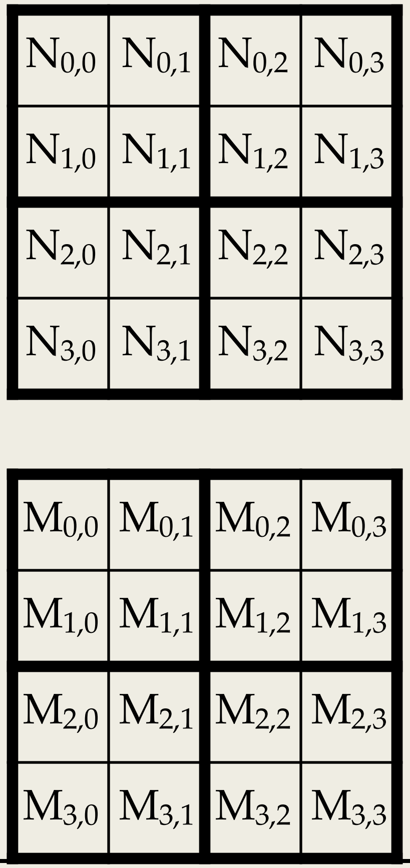
Step 4

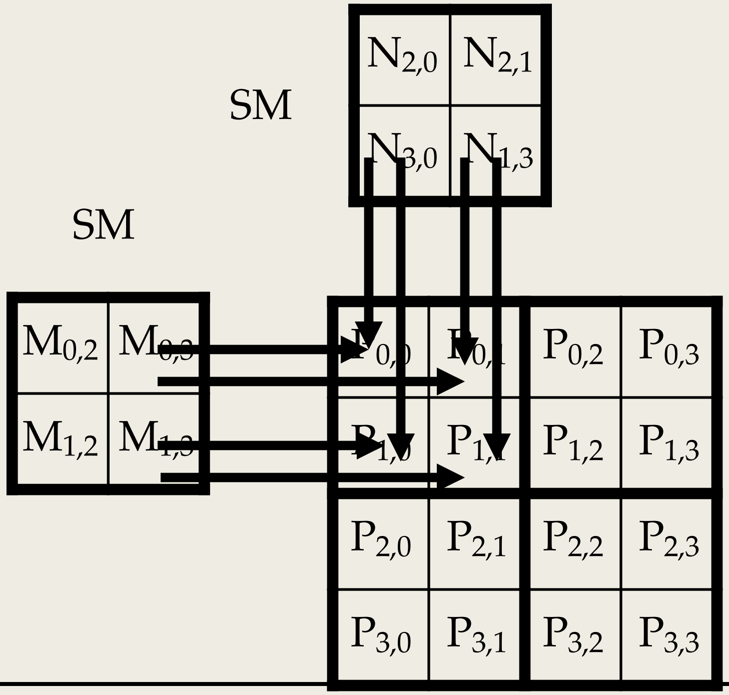


$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$
$N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$
$N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$
$N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
SM
$N_{2,0}$ $N_{2,1}$
$N_{3,0}$ $N_{1,3}$
SM
$M_{0,2}$ $M_{0,3}$
$M_{1,2}$ $M_{1,3}$
$P_{0,0}$ $P_{0,1}$ $P_{0,2}$ $P_{0,3}$
$P_{1,0}$ $P_{1,1}$ $P_{1,2}$ $P_{1,3}$
$P_{2,0}$ $P_{2,1}$ $P_{2,2}$ $P_{2,3}$
$P_{3,0}$ $P_{3,1}$ $P_{3,2}$ $P_{3,3}$

Work for Block (0,0)

Step 5





Phase 1: Loading a Tile

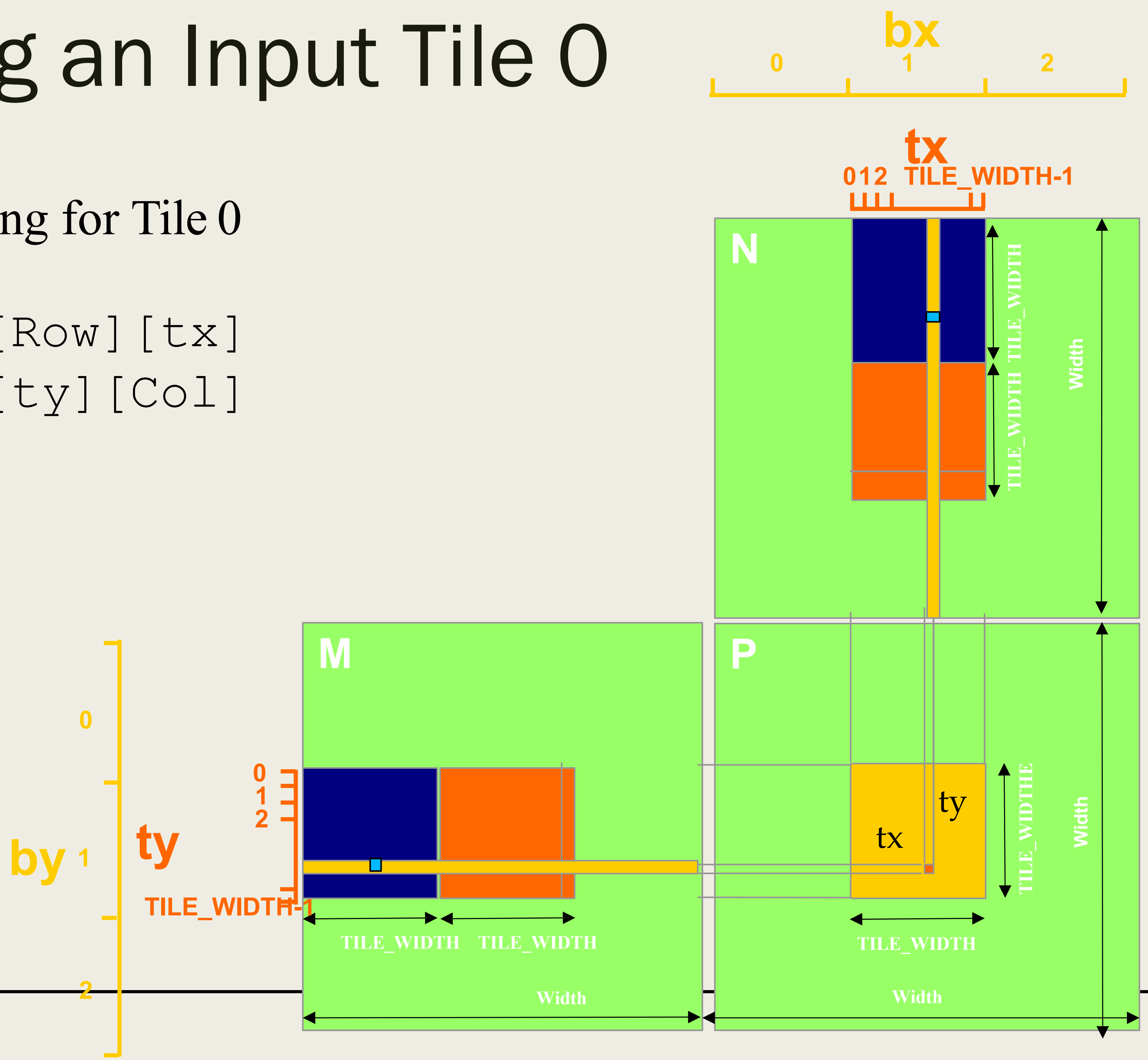
- All threads in a block participate
  - – Each thread loads one M element and one N element in basic tiling code

- Assign the loaded element to each thread such that the accesses within each warp is coalesced (more later).

Loading an Input Tile 0

2D indexing for Tile 0

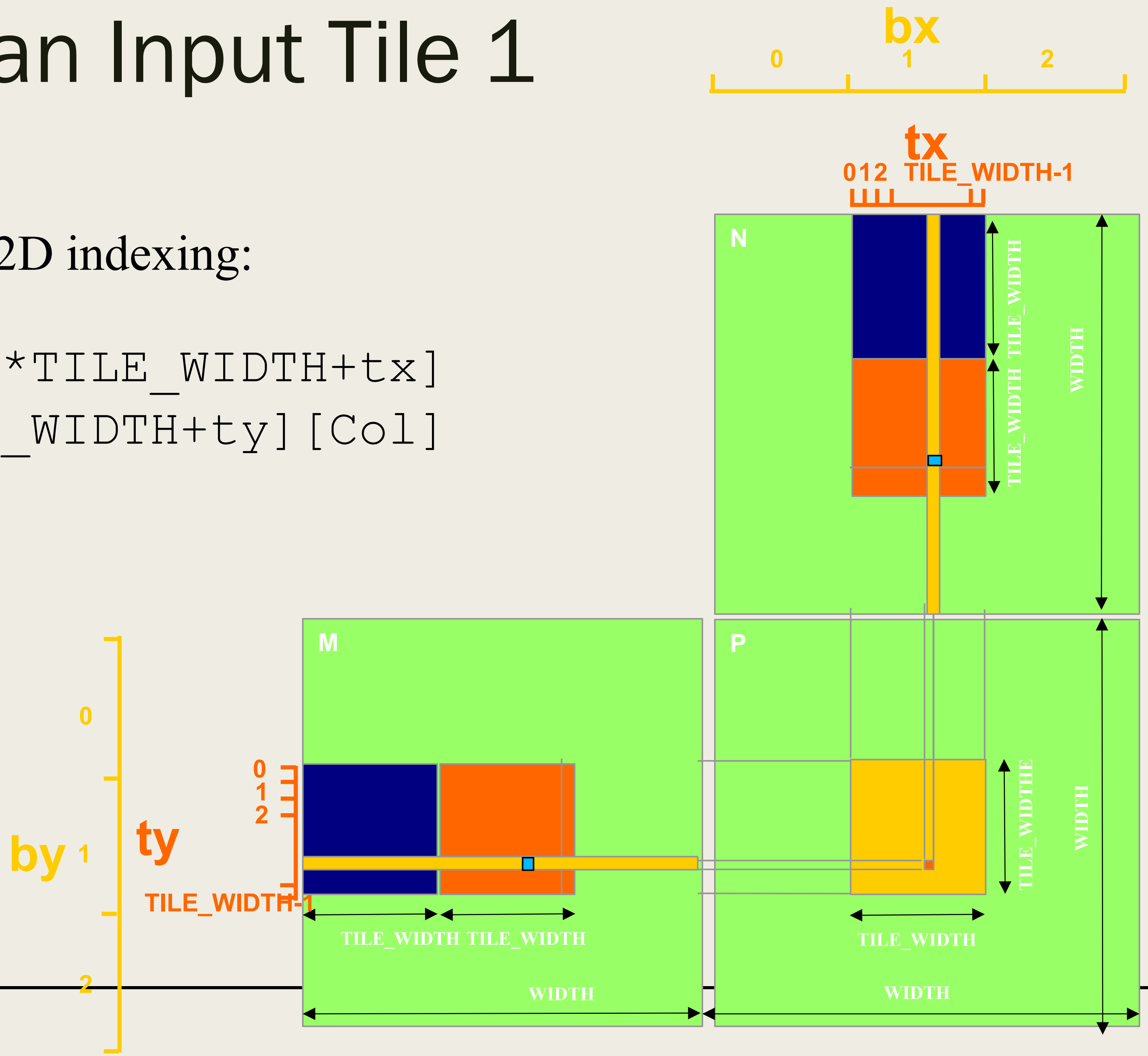
```
M[Row][tx]
N[ty][Col]
```



Loading an Input Tile 1

Accessing tile 1 in 2D indexing:

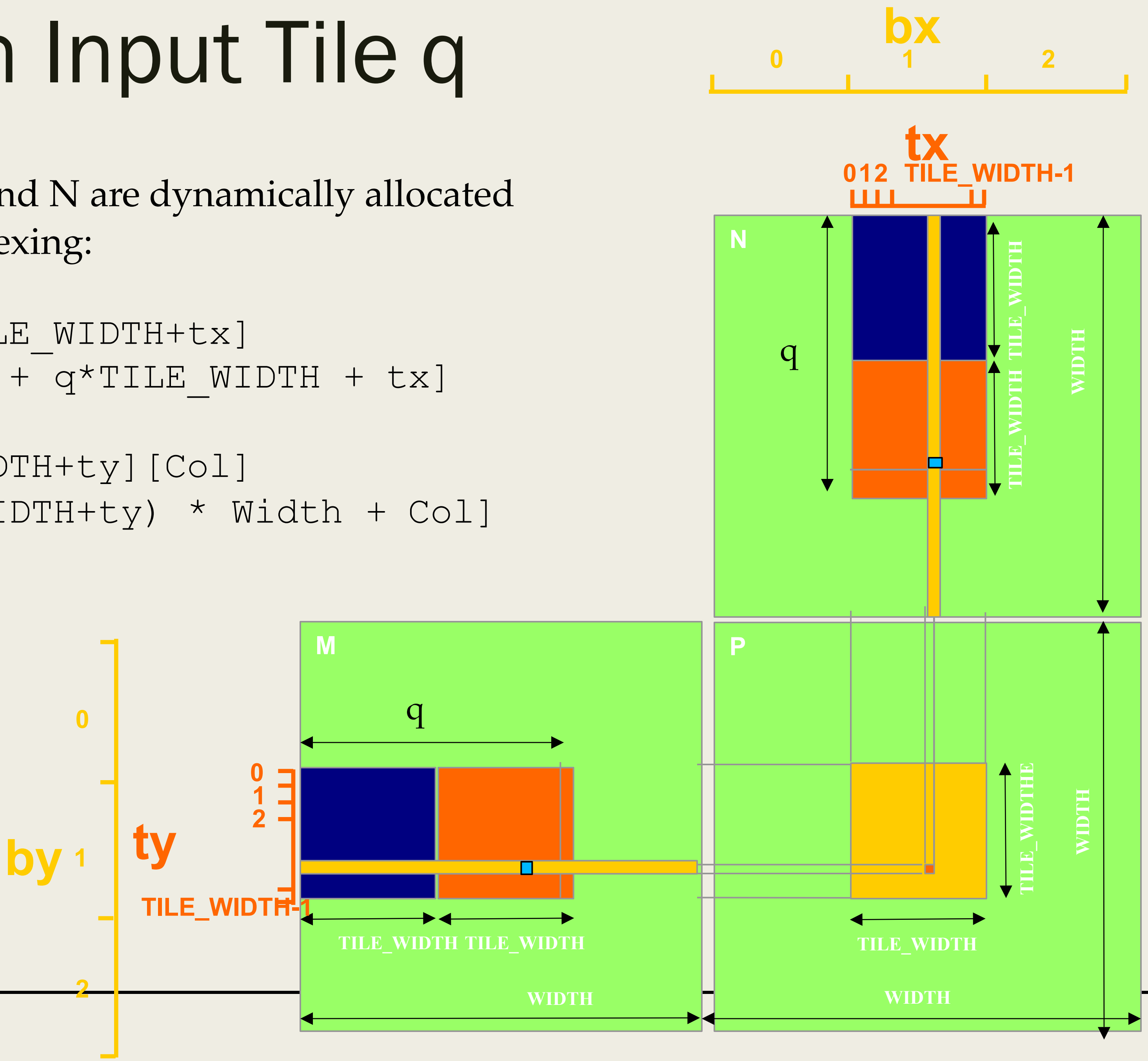
```
M[Row][1*TILE_WIDTH+tx]
N[1*TILE_WIDTH+ty][Col]
```



Loading an Input Tile q

However, recall that M and N are dynamically allocated and can only use 1D indexing:

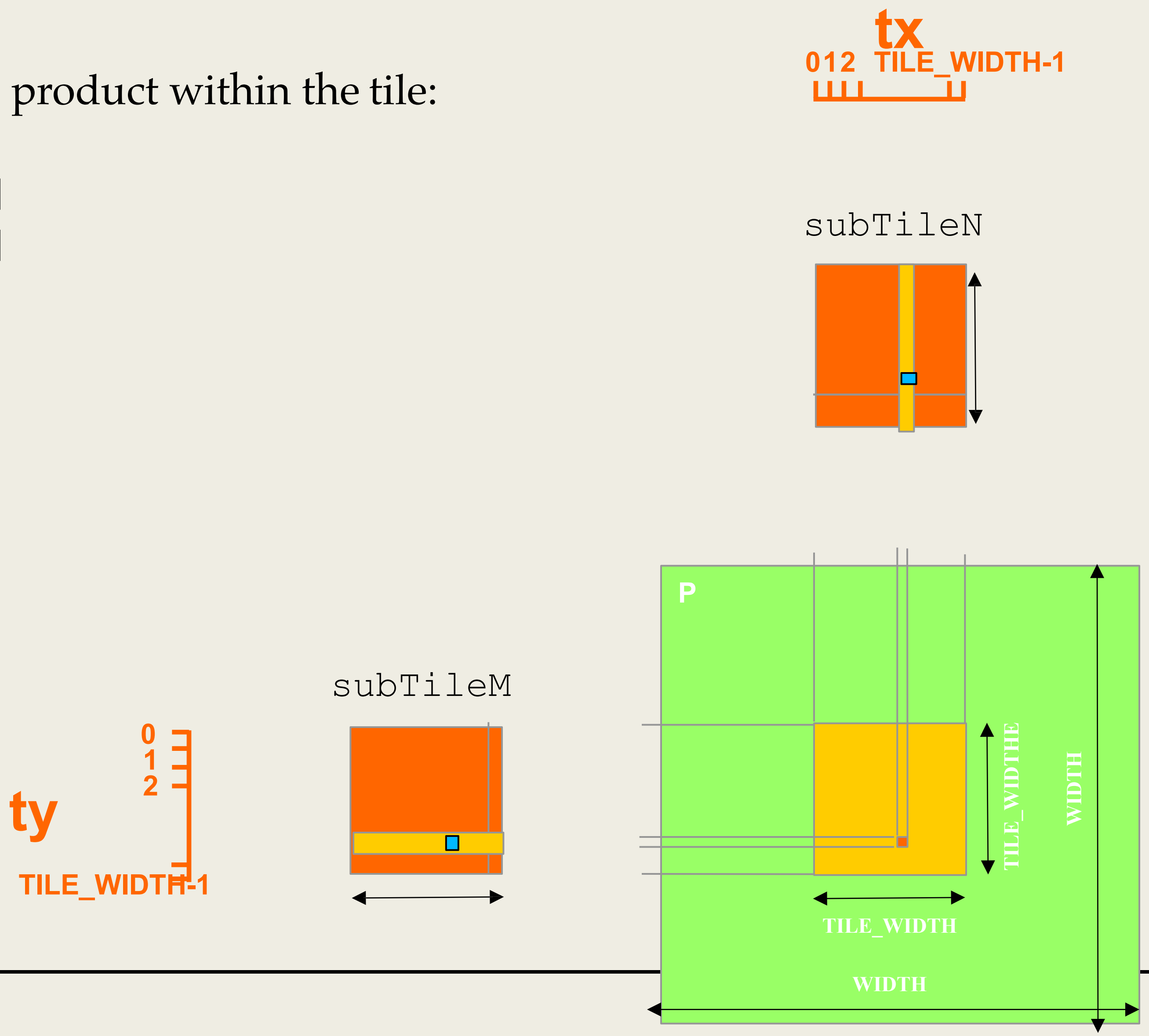
```
M[Row][m*TILE_WIDTH+tx]
M[Row*Width + q*TILE_WIDTH + tx]

N[q*TILE_WIDTH+ty][Col]
N[(q*TILE_WIDTH+ty) * Width + Col]
```



Phase 2: Compute partial product

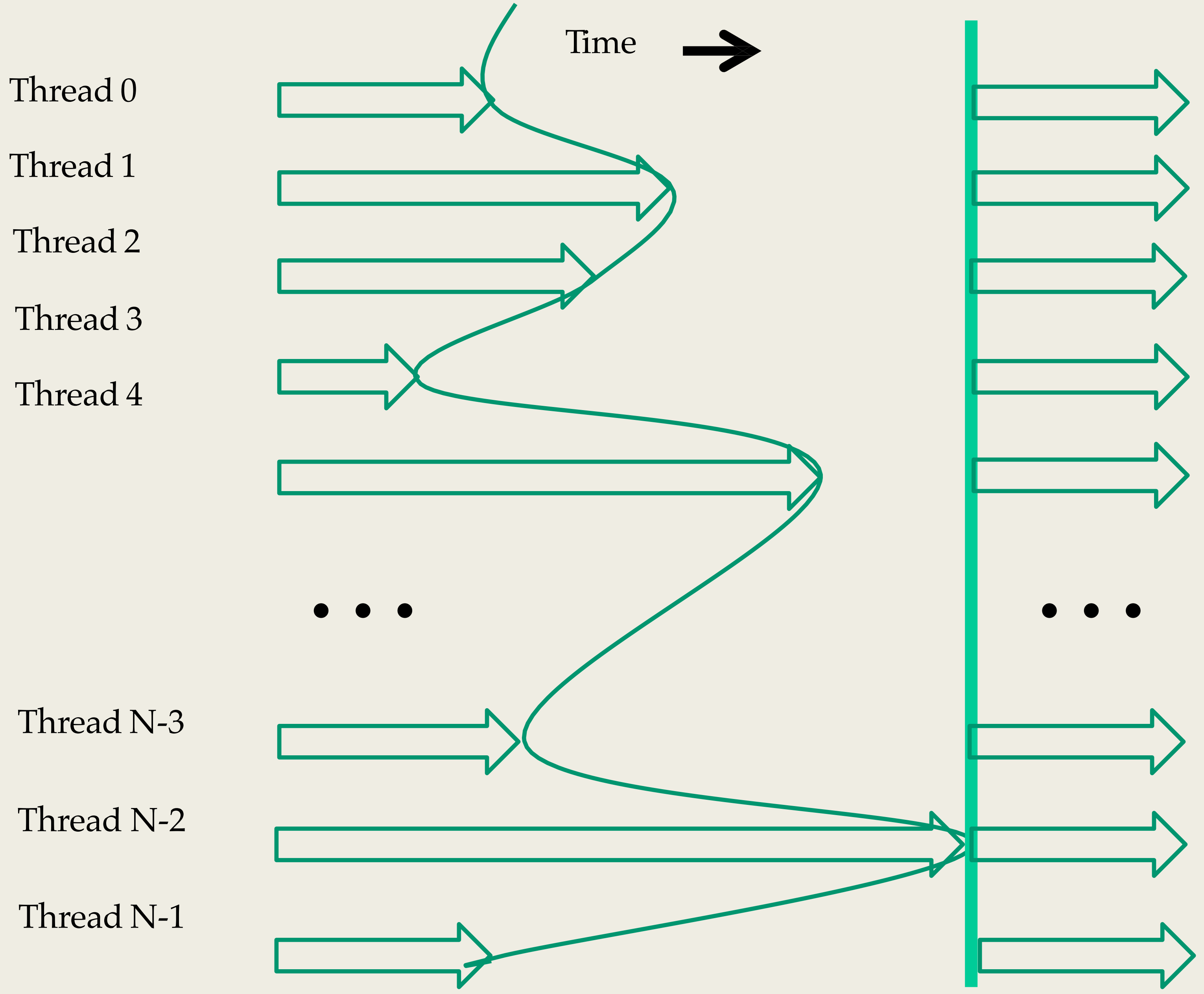
To perform the kth step of the product within the tile:

```
subTileM[ty][k]
subTileN[k][tx]
```



Barrier Synchronization

- An API function call in CUDA `__syncthreads()`
- All threads in the same block must reach the. `__syncthreads()` before any can move on
- Can be used to coordinate tiled algorithms
  - To ensure that all elements of a tile are loaded
  - To ensure that certain computation on elements is complete


Time
Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
• • •
• • •
Thread N-3
Thread N-2
Thread N-1

Tiled Matrix Multiplication Kernel

```
__global void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
    __shared__ float subTileM[TILE_WIDTH][TILE_WIDTH];
    __shared__ float subTileN[TILE_WIDTH][TILE_WIDTH];

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

    // Identify the row and column of the P element to work on
    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
    // The code assumes that the Width is a multiple of TILE_WIDTH!
    for (int q = 0; q < Width/TILE_WIDTH; ++q) {
       // Collaborative loading of M and N tiles into shared memory
       subTileM[ty][tx] = M[Row*Width + q*TILE_WIDTH+tx];
       subTileN[ty][tx] = N[(q*TILE_WIDTH+ty)*Width+Col];
       __syncthreads();
       for (int k = 0; k < TILE_WIDTH; ++k)
           Pvalue += subTileM[ty][k] * subTileN[k][tx];
       __syncthreads();
    }
    P[Row*Width+Col] = Pvalue;
}
```

Compare with Basic MM Kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
 // Calculate the row index of the P element and M
 int Row = blockIdx.y * blockDim.y + threadIdx.y;
 // Calculate the column index of P and N
 int Col = blockIdx.x * blockDim.x + threadIdx.x;

 if ((Row < Width) && (Col < Width)) {
    float Pvalue = 0;

    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; ++k)
      Pvalue += M[Row*Width+k] * N[k*Width+Col];

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

Shared Memory and Threading

- Each SM in Maxwell has 64KB shared memory (48KB max per block)
  - Shared memory size is implementation dependent!

  - For TILE_WIDTH = 16, each thread block uses 2*256*4B = 2KB of shared memory
    - Shared memory can potentially support up to 32 active blocks
    - The threads per SM constraint (2048) will limit the number of blocks to 8
    - This allows up to 8*512 = 4096 pending loads. (2 per thread, 256 threads per block)

  - TILE_WIDTH = 32 would lead to 2*32*32*4B= 8KB shared memory per thread block
    - Shared memory can potentially support up to 8 active blocks
    - The threads per SM constraint (2048) will limit the number of blocks to 2
    - This allows up to 2*2048 = 4096 pending loads (2 per thread, 1024 threads per block)

Memory Bandwidth Consumption

- Using 16x16 tiling, we reduce the global memory by a factor of 16
  - Each float is now used by 16 floating-point operations
  - The 150GB/s bandwidth can now support (150/4)*16 = 600 GFLOPS!
- Using 32x32 tiling, we reduce the global memory accesses by a factor of 32
  - Each float is now used by 32 floating-point operations
  - The 150 GB/s bandwidth can now support (150/4)*32 = 1200 GFLOPS!
  - The memory bandwidth is no longer a limiting factor for performance!