ECE408/CS483/CSE408 Fall 2024

Applied Parallel Programming

Lecture 5:
Locality and Tiled Matrix Multiplication

Course Reminders

- Lab 1 is due this Friday – this is the first REAL lab.
  - Do not wait until last minute, we do not accept late submissions, see below.
- Lab 2 will be out on Friday; it is due next week.
- Lowest lab grade will be dropped from the final grade.
- Update on Late Submission for Labs and Project:
  - We allow 2 48-hour deadline extensions, combined across labs and project milestones. If such an extension is needed, please fill in the form https://forms.gle/Tj5p2SS2EN27fY68A before noon of the day of the corresponding deadline.
    - For example, to request a 48-hour deadline extension for Lab 1, which is due at 8pm on Friday, September 13th, please fill the form before 12:00 PM (noon) on 9/13.
  - If you have already used this extension twice, you no longer can use it.
  - **Because of this extension policy and because a lab with the lowest score will be dropped from the final grade, no late submissions will be allowed for any reasons for both labs and projects.**

Objective

- To learn to evaluate the performance implications of global memory accesses
- To prepare for Lab 3: tiled matrix multiplication
- To learn to assess the benefit of tiling

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);
```

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 index 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];
      // accumulated dot product is stored in d_P[Row][Col]
      d_P[Row*Width+Col] = Pvalue;
   }
}
```

Review: 4B of Data per FLOP

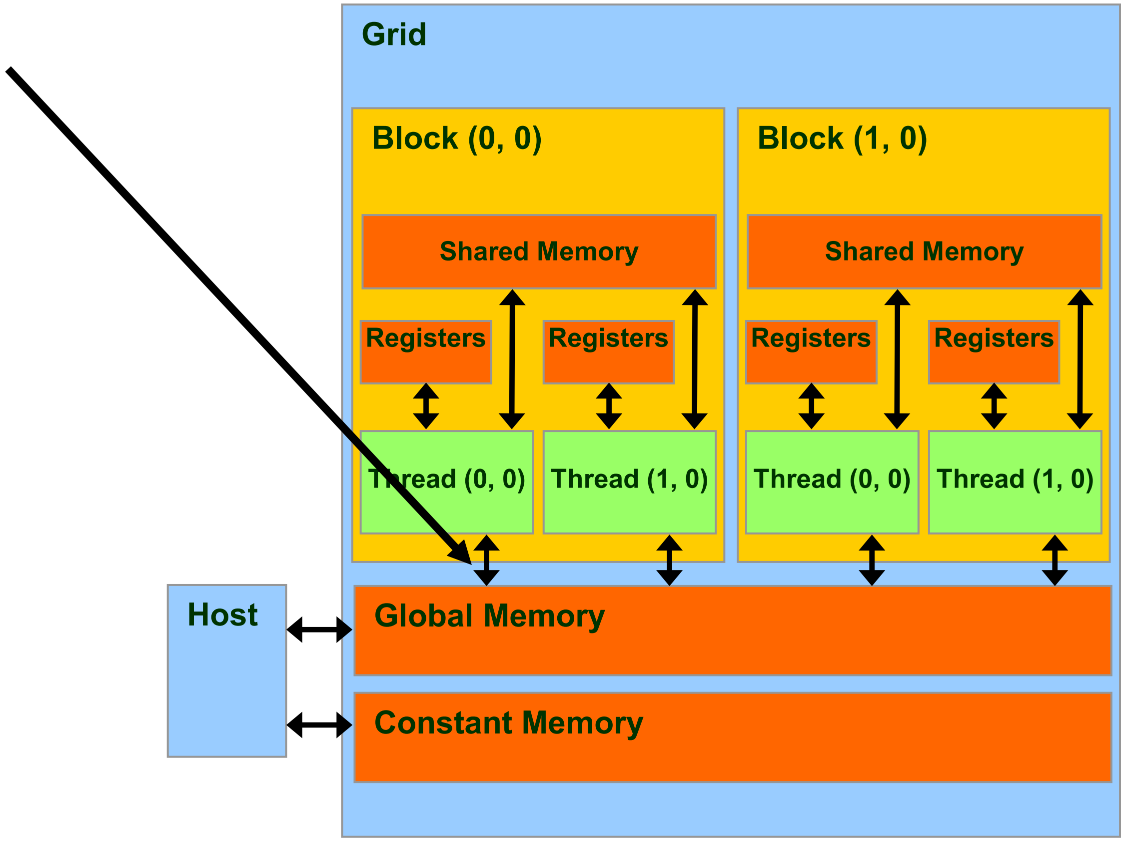
- Each threads access global memory
  - for elements of **M** and **N**:
  - **4B each**, or **8B per pair**.
  - (And once TOTAL to **P** per thread—ignore it.)
- With each pair of elements,
  - a thread does a single multiply-add,
  - **2 FLOP**—floating-point operations.
- So for every FLOP,
  - **a thread needs** 4B from memory:
  - **4B / FLOP**.

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



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

A Common Programming Strategy

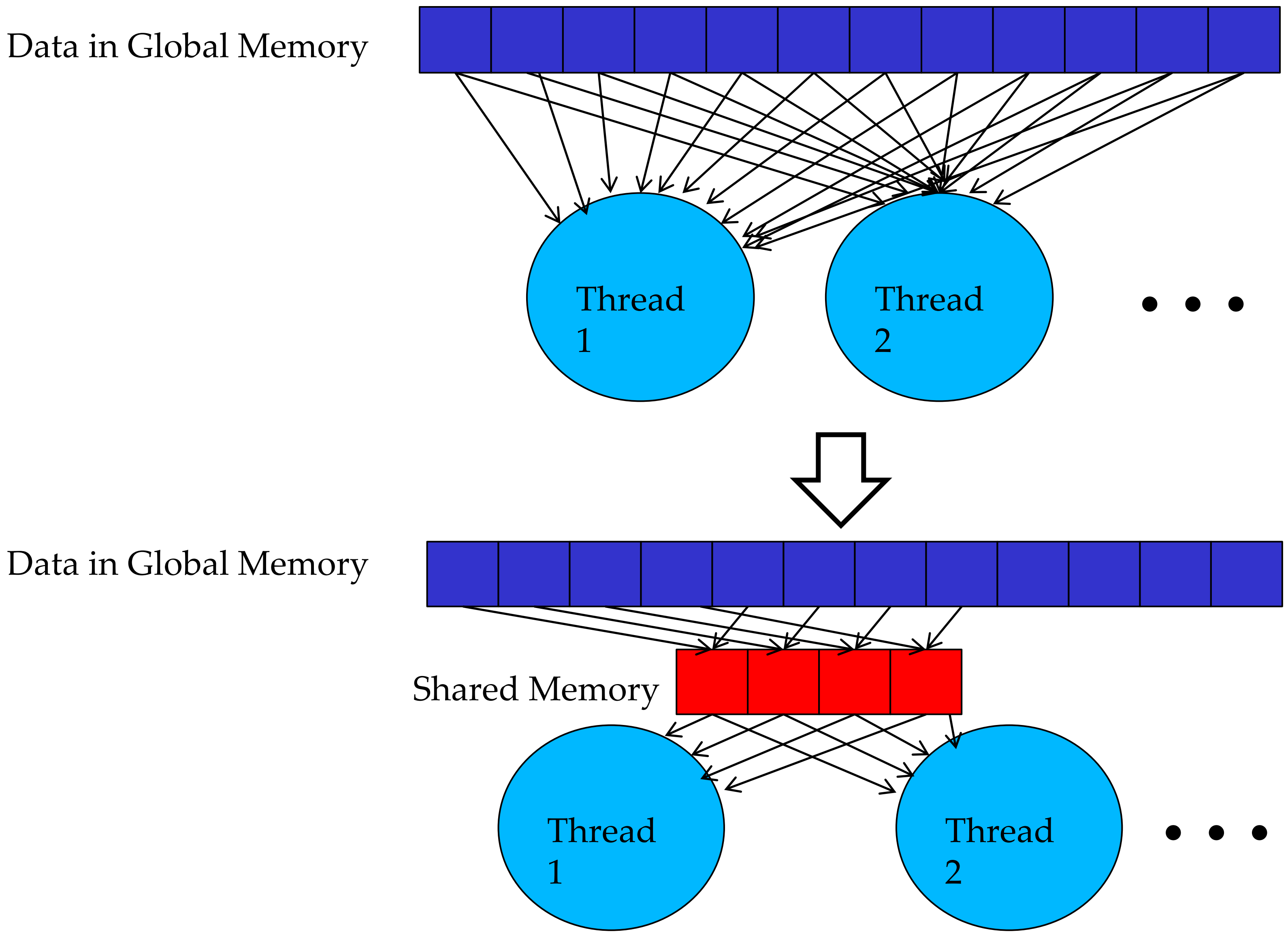
- In a GPU, **only threads in a block can** use **shared** memory.
- Thus, each **block** operates on **separate tiles**:
  - **Read tile(s)** into shared memory **using multiple threads** to exploit memory-level parallelism.
  - **Compute** based on shared memory tiles.
  - **Repeat**.
  - **Write results** back **to global memory**.

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];
```

Common across all threads in a block

Shared Memory Tiling Basic Idea



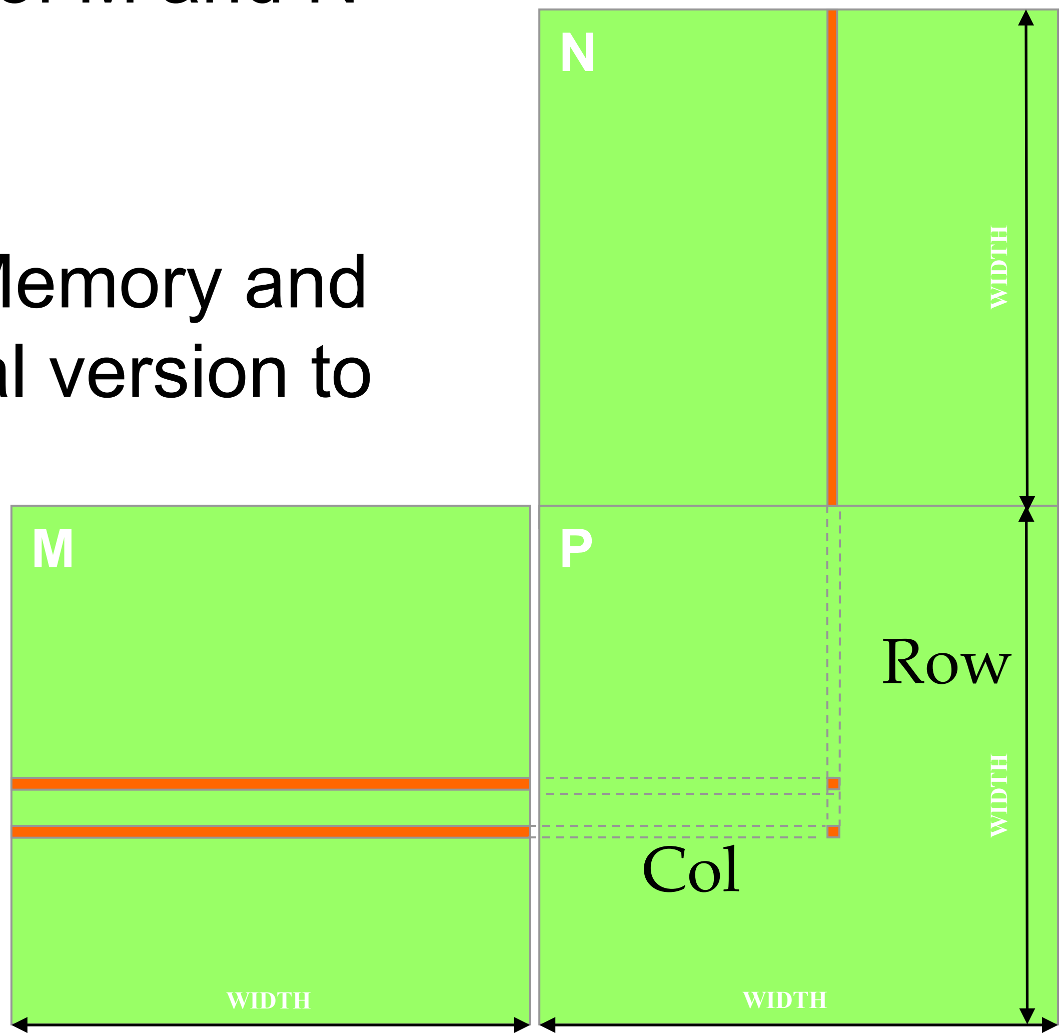
Outline of Technique

- 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

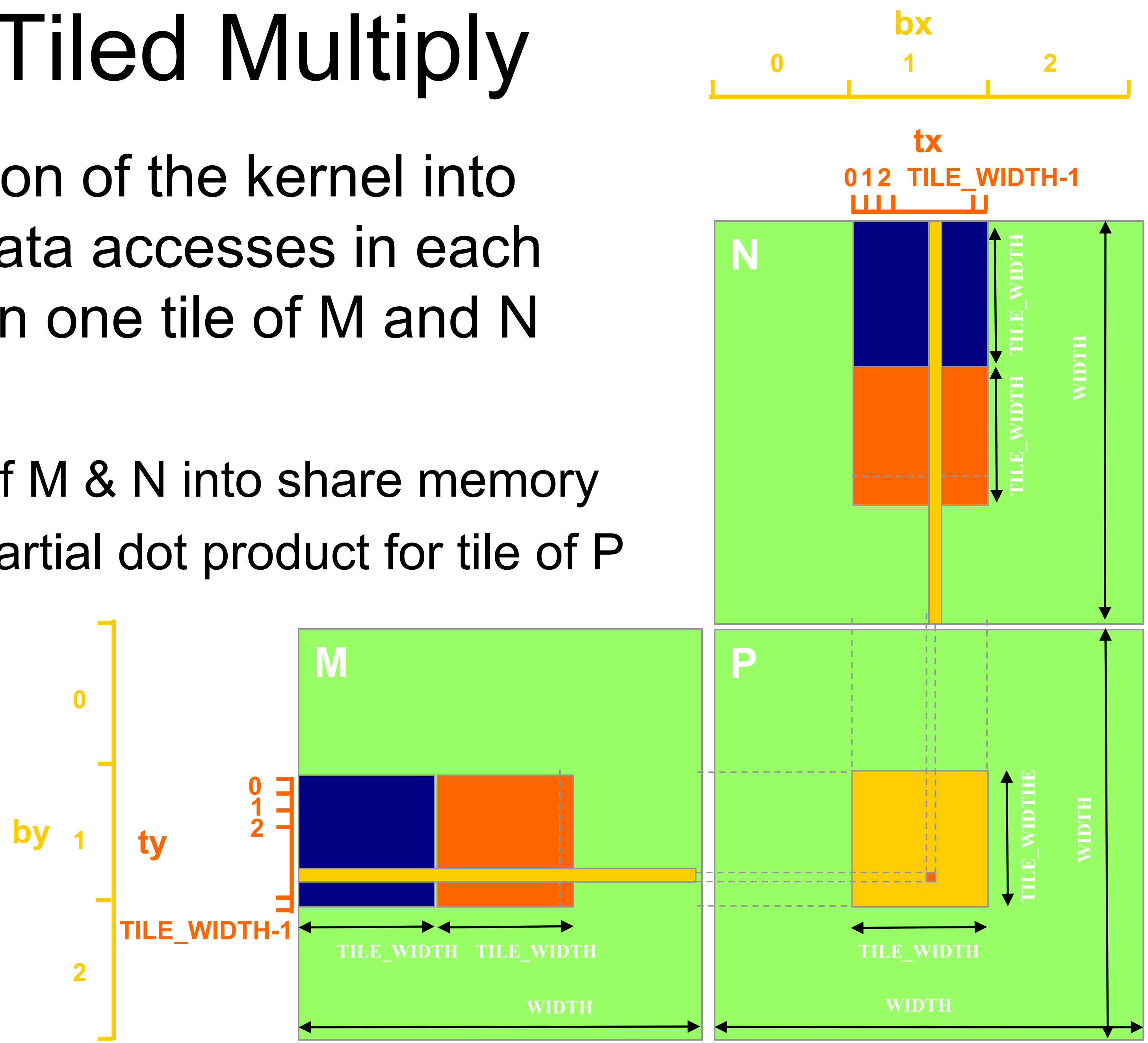
- Observe that each input element of M and N is used WIDTH times

- 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

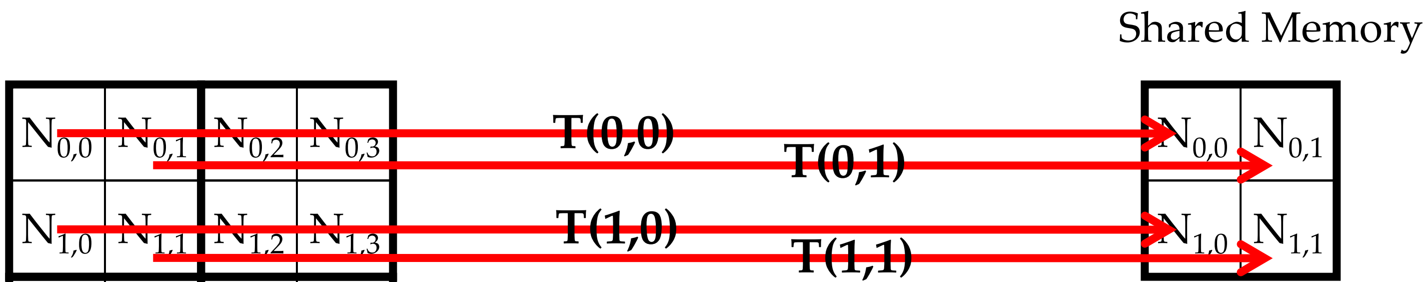


Loading a Tile

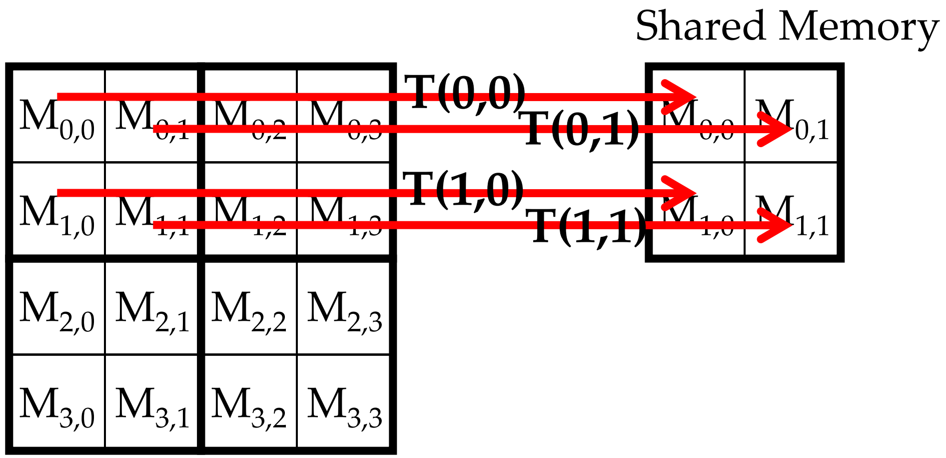
- All threads in a block participate
  - Each thread loads
    - one **M** element and
    - one **N** element
  - in basic tiling code.

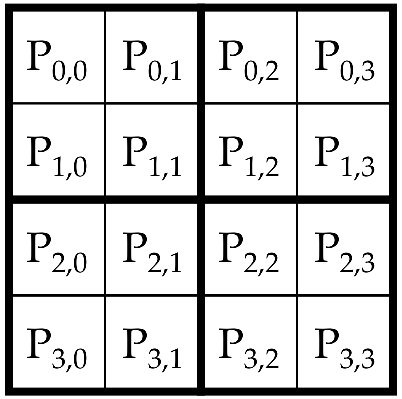
- Assign the loaded element to each thread such that the accesses within each warp is coalesced (more later).

Loading Tiles for Block (0,0)

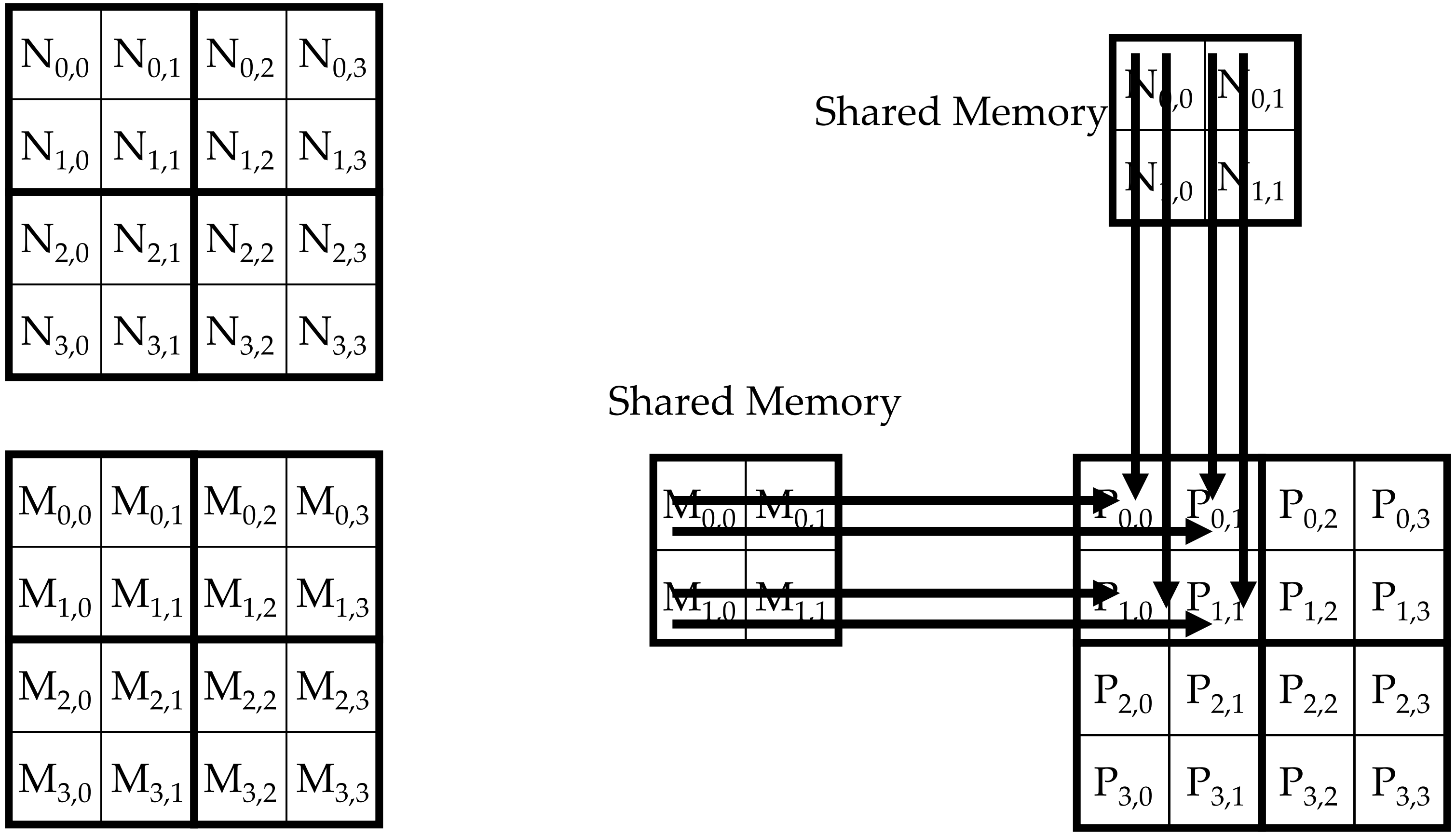


Neighboring threads in a warp access neighboring locations for both N and M



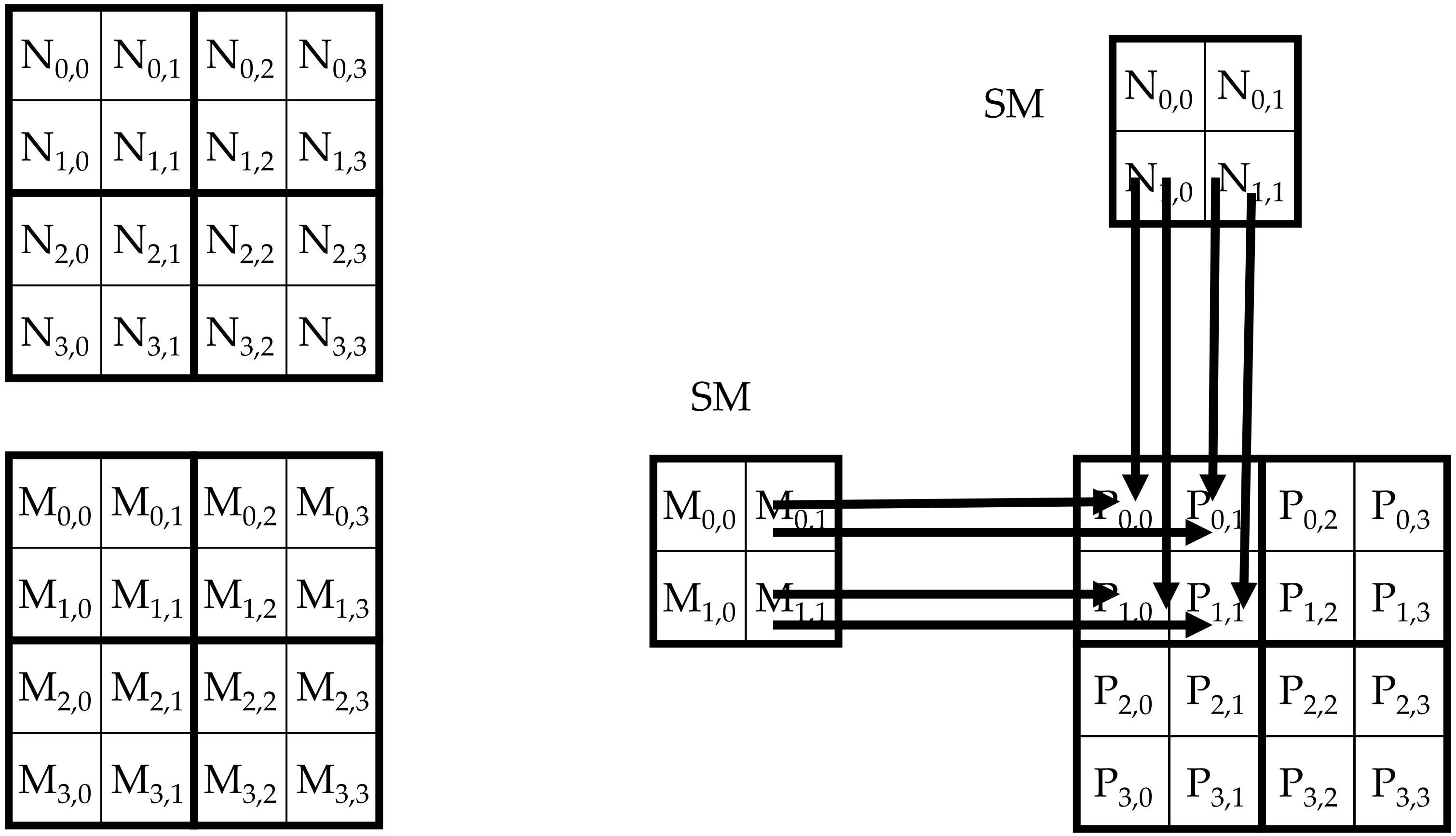


Work for Block (0,0)
Step 1

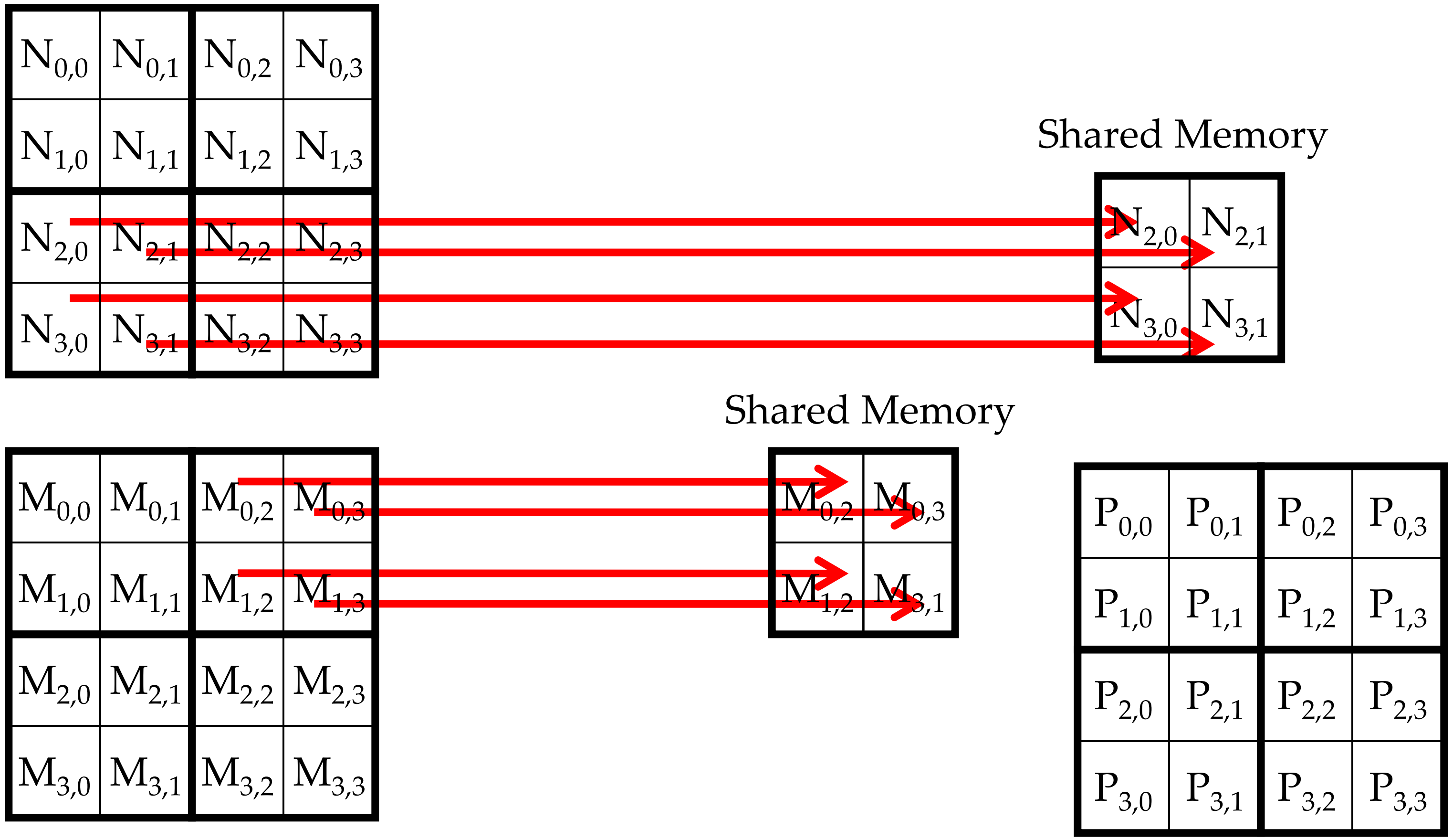


Work for Block (0,0)

Step 2

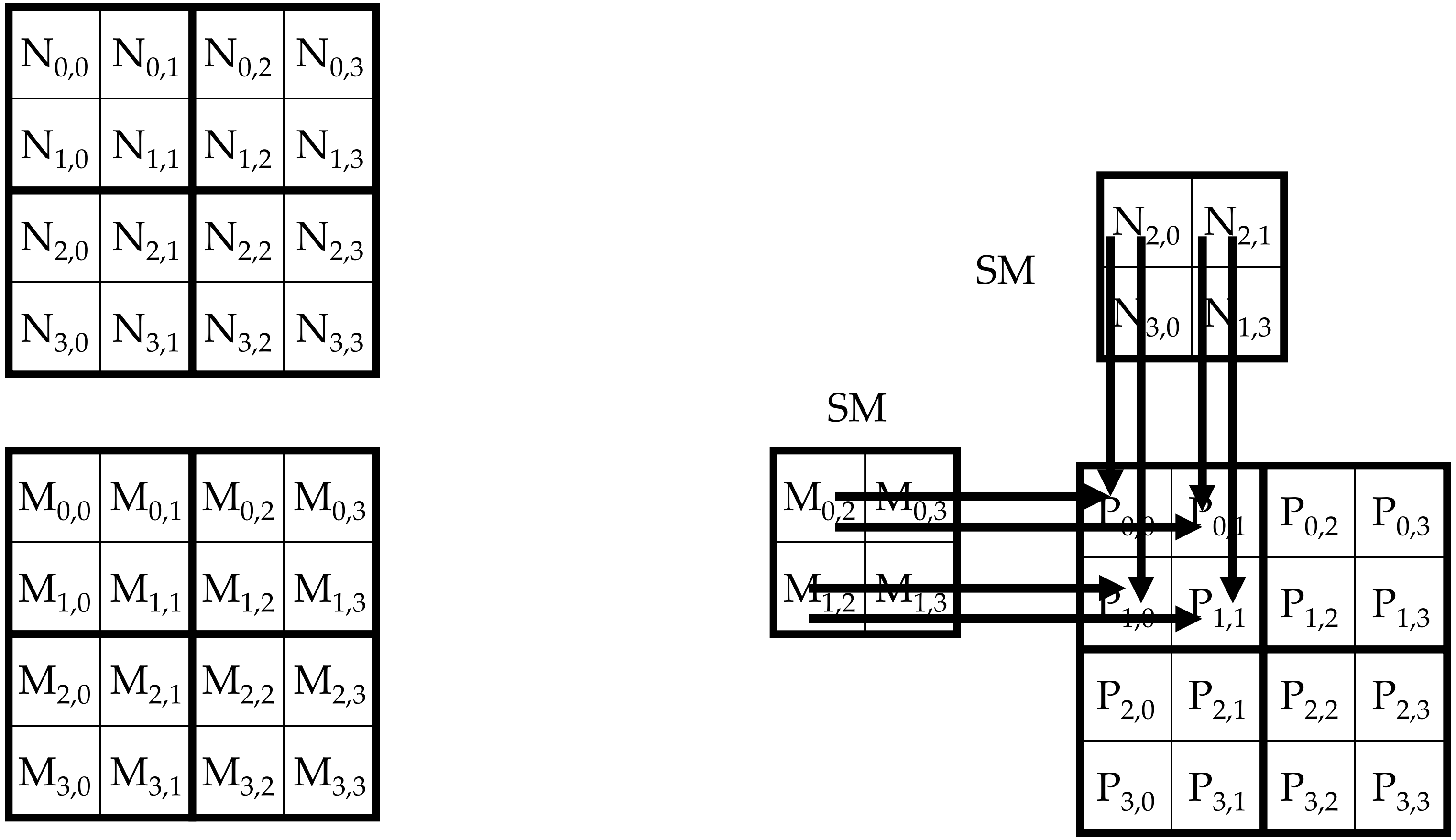


Work for Block (0,0)
Step 3



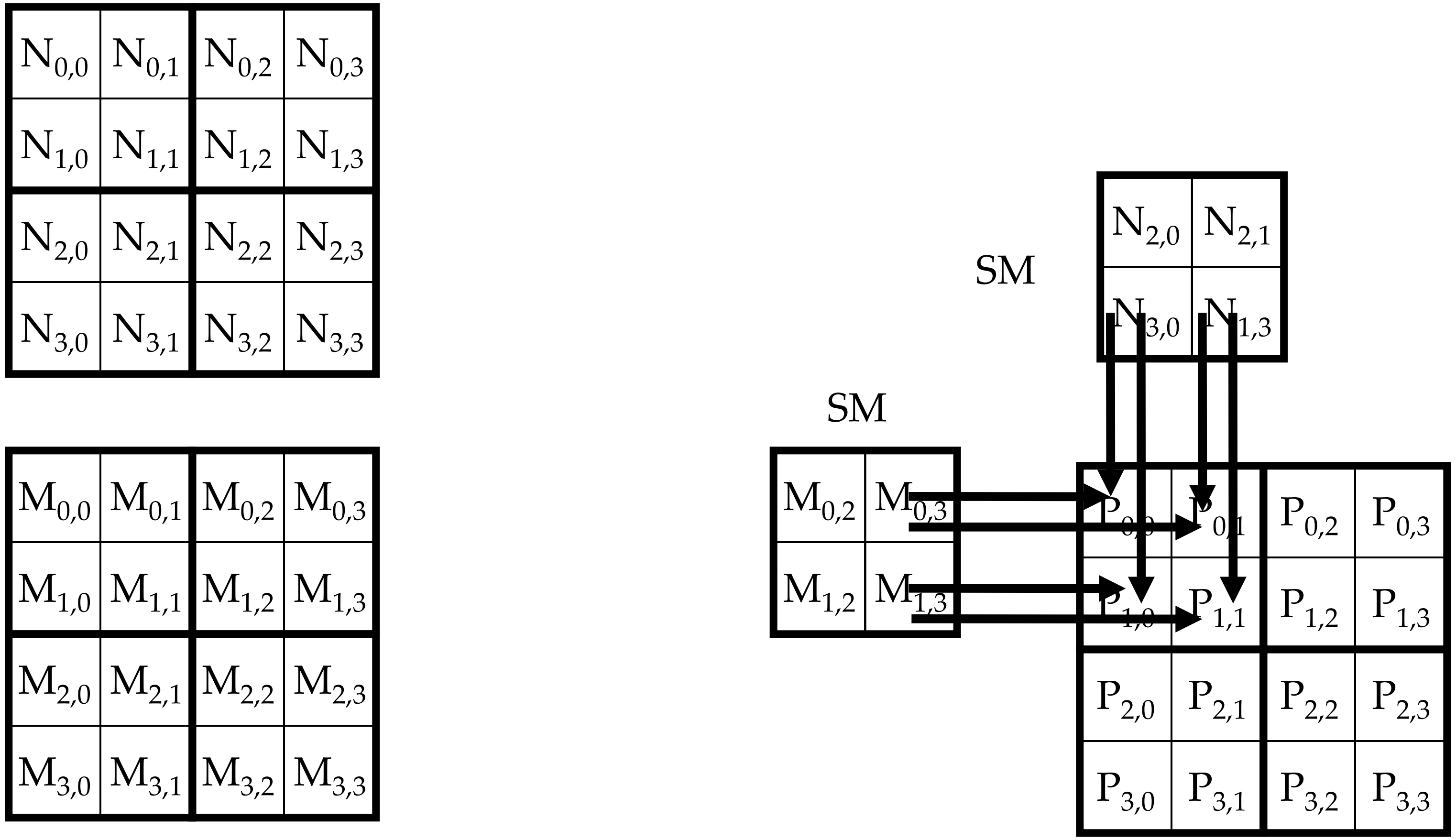
Work for Block (0,0)

Step 4



Work for Block (0,0)

Step 5



Phase 1: Loading a Tile

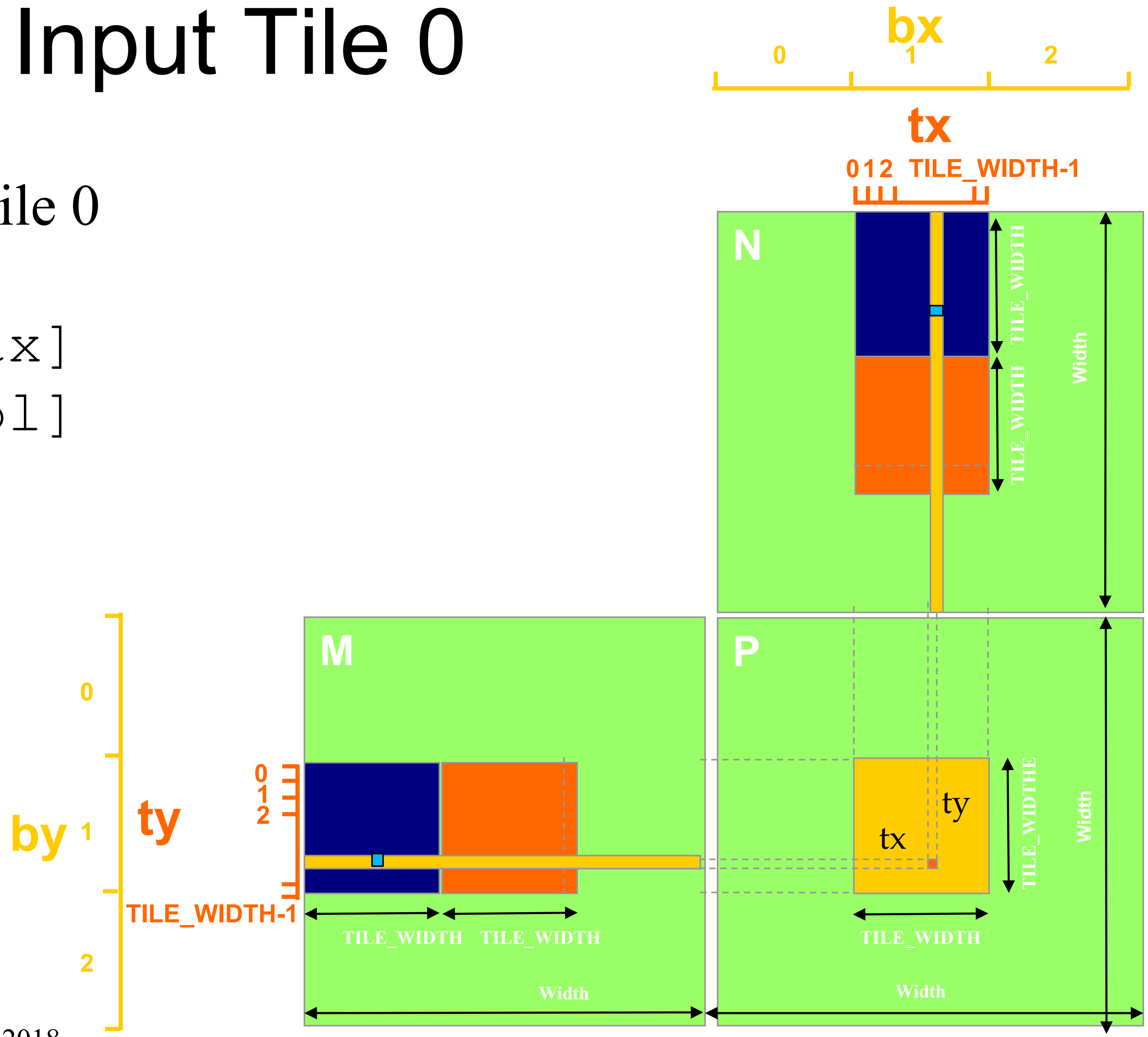
- All threads in a block participate
  - Each thread loads one M element and one N element in basic tiling code

- 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

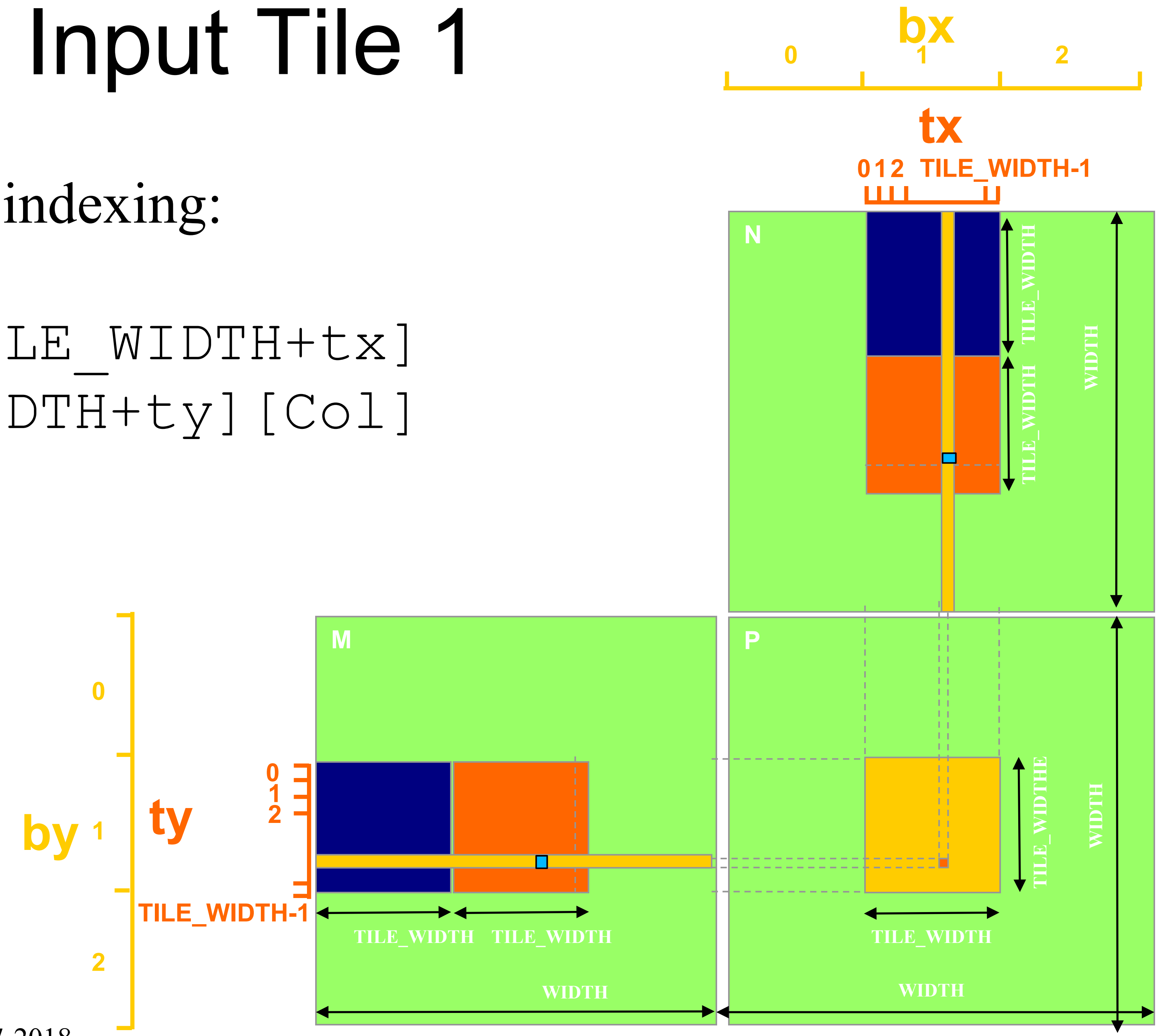
```
M[Row][tx]
N[ty][Col]
```



Loading an Input Tile 1

Accessing tile 1 in 2D indexing:

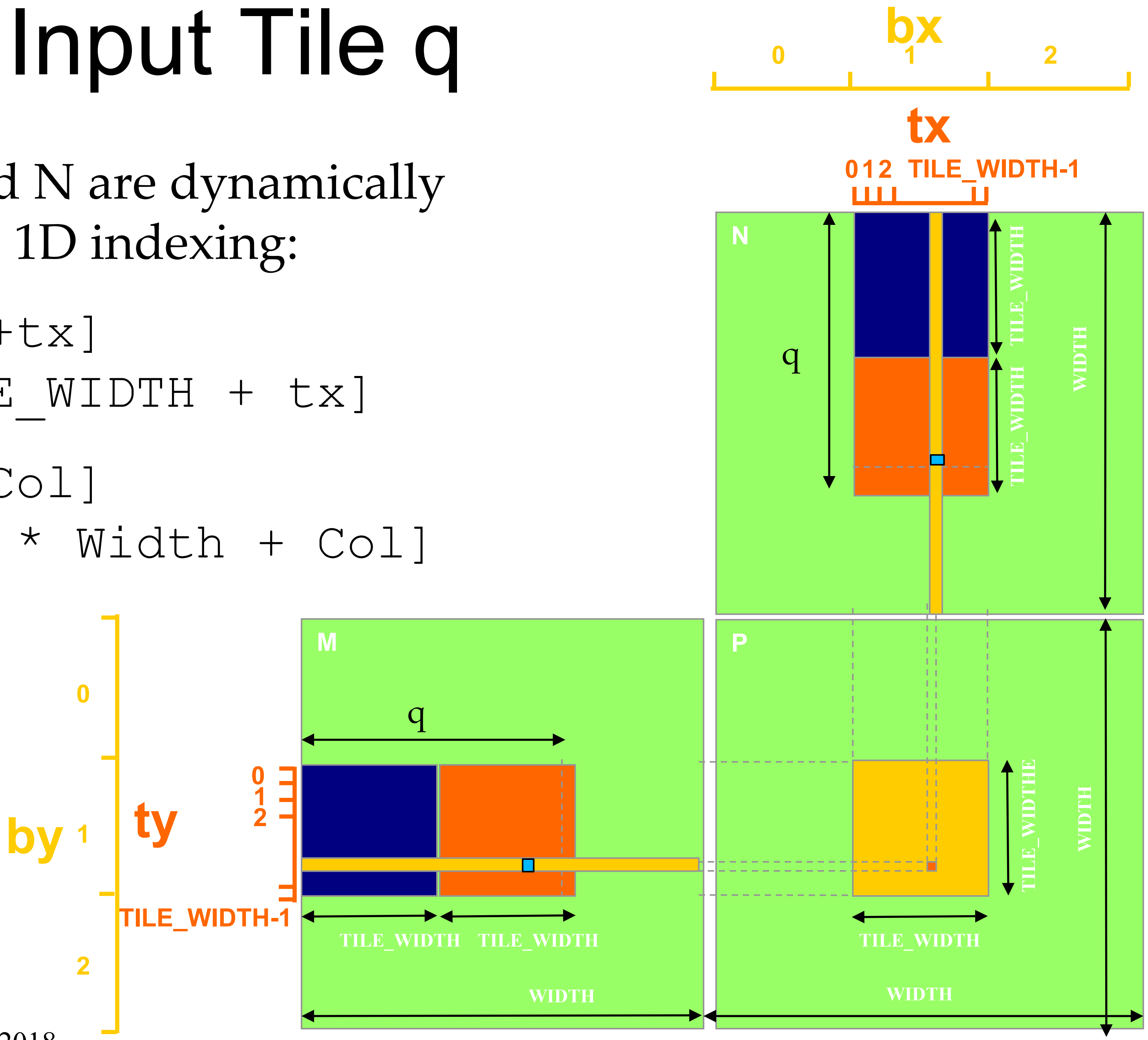
```
M[Row][1*TILE_WIDTH+tx]
N[1*TILE_WIDTH+ty][Col]
```



Loading an Input Tile q

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

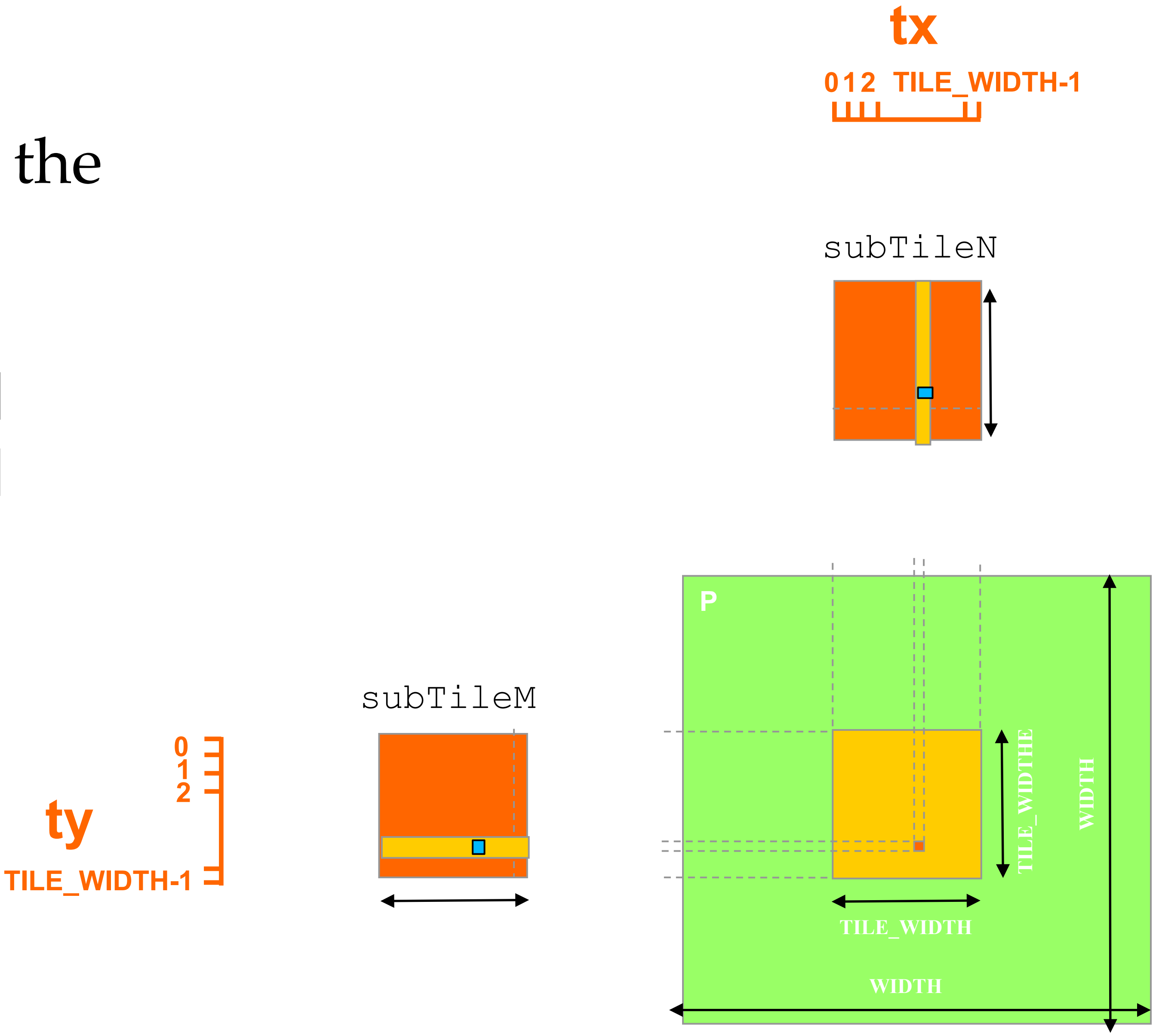
```
M[Row][q*TILE_WIDTH+tx]
M[Row*Width + q*TILE_WIDTH + tx]

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



Phase 2: Compute partial product

To perform the k^{th} step of the product within the tile:

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



We're Not There Yet!

- But …
- **How can a thread know** …
  - **That another thread has finished** its part of the tile?
  - Or that another thread has finished using the previous tile?

We need to synchronize!

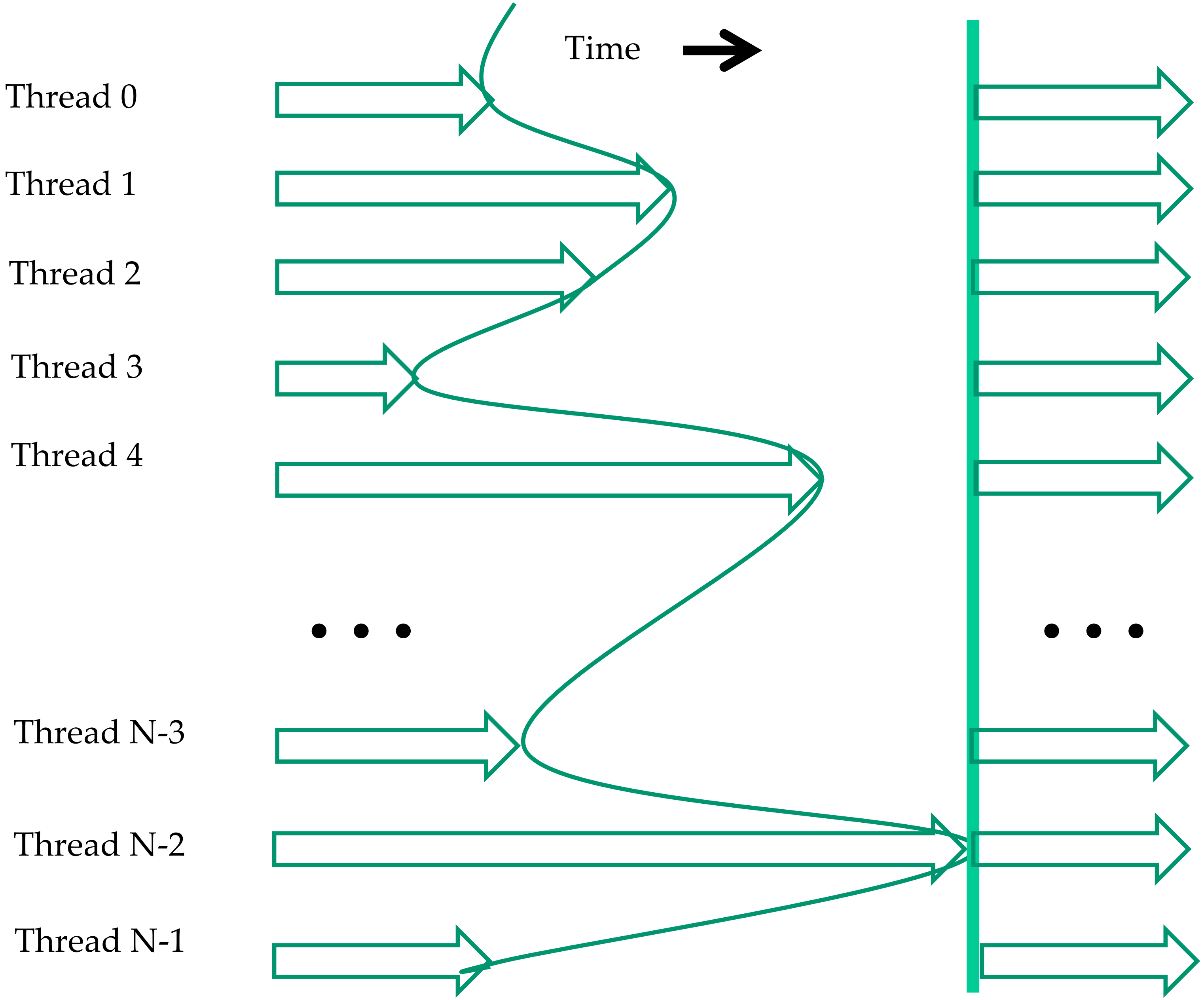
Leveraging Parallel Strategies

- **Bulk synchronous execution**:
  threads execute roughly in unison
  1. Do some work
  2. Wait for others to catch up
  3. Repeat
- **Much easier programming model**
  – Threads only parallel within a section
  – Debug lots of little programs
  – Instead of one large one.
- **Dominates high-performance applications**

Bulk Synchronous Steps Based on Barriers

- **How does it work?**
  Use a barrier to wait for thread to ‘catch up.’

- A barrier is a synchronization point:
  - **each thread calls a function** to enter barrier;
  - **threads block** (sleep) in barrier function
    until all threads have called;
  - **after last thread calls** function,
    all threads continue past the barrier.


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

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;
  }
}
```

Use of Large Tiles Shifts Bottleneck

- Recall our example GPU: **1,000 GFLOP/s**, **150 GB/s**
- **16x16 tiles** use each operand for 16 operations
  - **reduce global** memory **accesses by** a factor of **16**
  - **150GB/s** bandwidth supports (150/4)*16 = **600 GFLOPS**!
- **32x32 tiles** use each operand for 32 operations
  - **reduce global** memory **accesses by** a factor of **32**
  - **150 GB/s** bandwidth supports (150/4)*32 = **1,200 GFLOPS**!
  - **Memory bandwidth is no longer the bottleneck!**

Also Need Parallel Accesses to Memory

- Shared memory size
  - implementation dependent
  - **64kB** per SM in Maxwell (48kB max per block)
- Given **TILE_WIDTH of 16** (256 threads / block),
  - each thread block uses
    2*256*4B = 2kB of shared memory,
  - which limits active blocks to 32;
  - max. of 2048 threads per SM,
  - which limits blocks to 8.
  - Thus, up to 8*512 = **4,096 pending loads**
    (2 per thread, 256 threads per block)

Another Good Choice: 32x32 Tiles

- Given **TILE_WIDTH of 32** (1,024 threads / block),
  - each thread block uses
    2*1024*4B = 8kB of shared memory,
  - which limits active blocks to 8;
  - max. of 2,048 threads per SM,
  - which limits blocks to 2.
  - Thus, up to 2*2,048 = **4,096 pending loads**
    (2 per thread, 1,024 threads per block)

(same memory parallelism exposed)

Current GPU? Use Device Query

- Number of devices in the system

```
int dev_count;
cudaGetDeviceCount( &dev_count);
```

- Capability of devices

```
cudaDeviceProp   dev_prop;
for (i = 0; i < dev_count; i++) {
        cudaGetDeviceProperties( &dev_prop, i);
  // decide if device has sufficient resources and capabilities
}
```

- cudaDeviceProp is a built-in C structure type
  – dev_prop.maxThreadsPerBlock
  – dev_prop.sharedMemoryPerBlock
  – …

ANY MORE QUESTIONS? READ CHAPTER 4!