

ECE408/CS483/CSE408 Spring 2025

Applied Parallel Programming

Lecture 5:
Locality and Tiled Matrix
Multiplication

Course Reminders

- Lab 2 is due this week on Friday
 - This lab and all future labs will count towards your final grade.
- Reminder about Late Submission for Labs and Project:
 - **All assignments are due by Friday at 8pm.**
 - However, recognizing potential issues such as Delta system maintenance, interviews, or illness, we provide a **three-day** grace period. You can submit an assignment by Monday at 8pm and still receive full credit without penalty. You do not need to notify us if you intend to use this default 3-day grace period.
 - To extend your grace period beyond 3 days, you must notify us by Friday 8pm via email to the TA(s) responsible for the assignment you are asking about. Extensions to the grace period are given out **for extenuating circumstances at our discretion**. Requests after this time will not be considered.
 - **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;

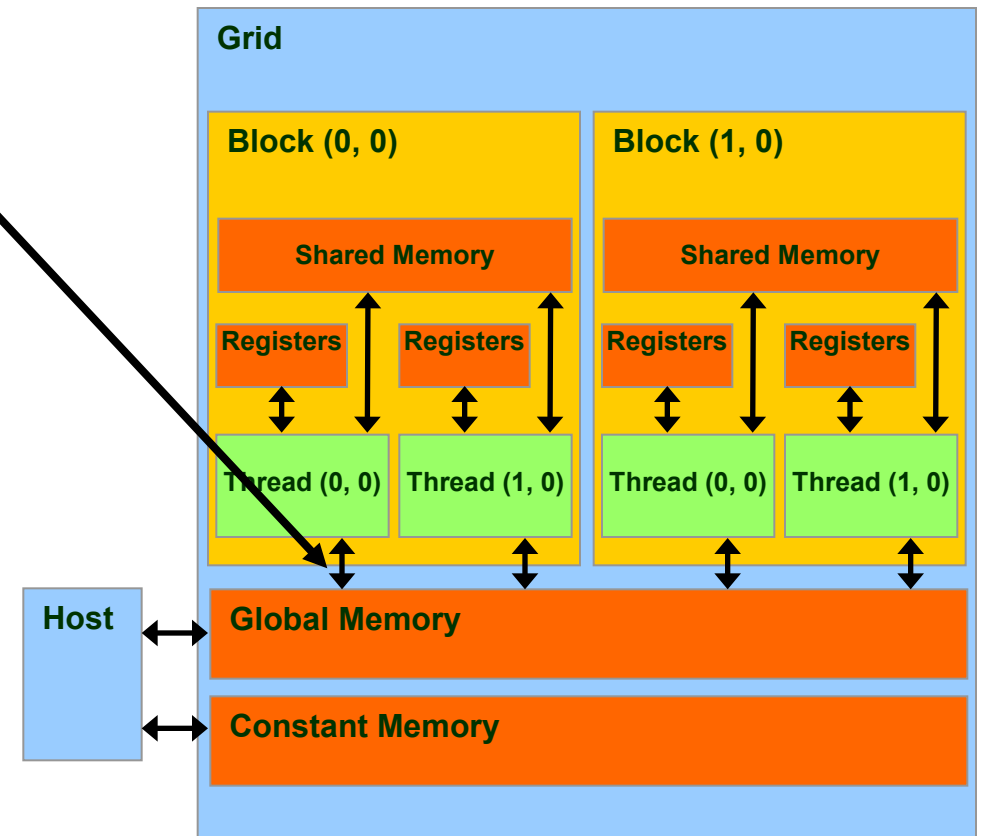
    if ((Row < Width) && (Col < Width)) {
        float Pvalue = 0;
        // each thread computes one element of the block sub-matrix
        for (int k = 0; k < Width; ++k)
            Pvalue += d_M[Row*Width+k] * d_N[k*Width+Col];
        // accumulated dot product is stored in d_P[Row][Col]
       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

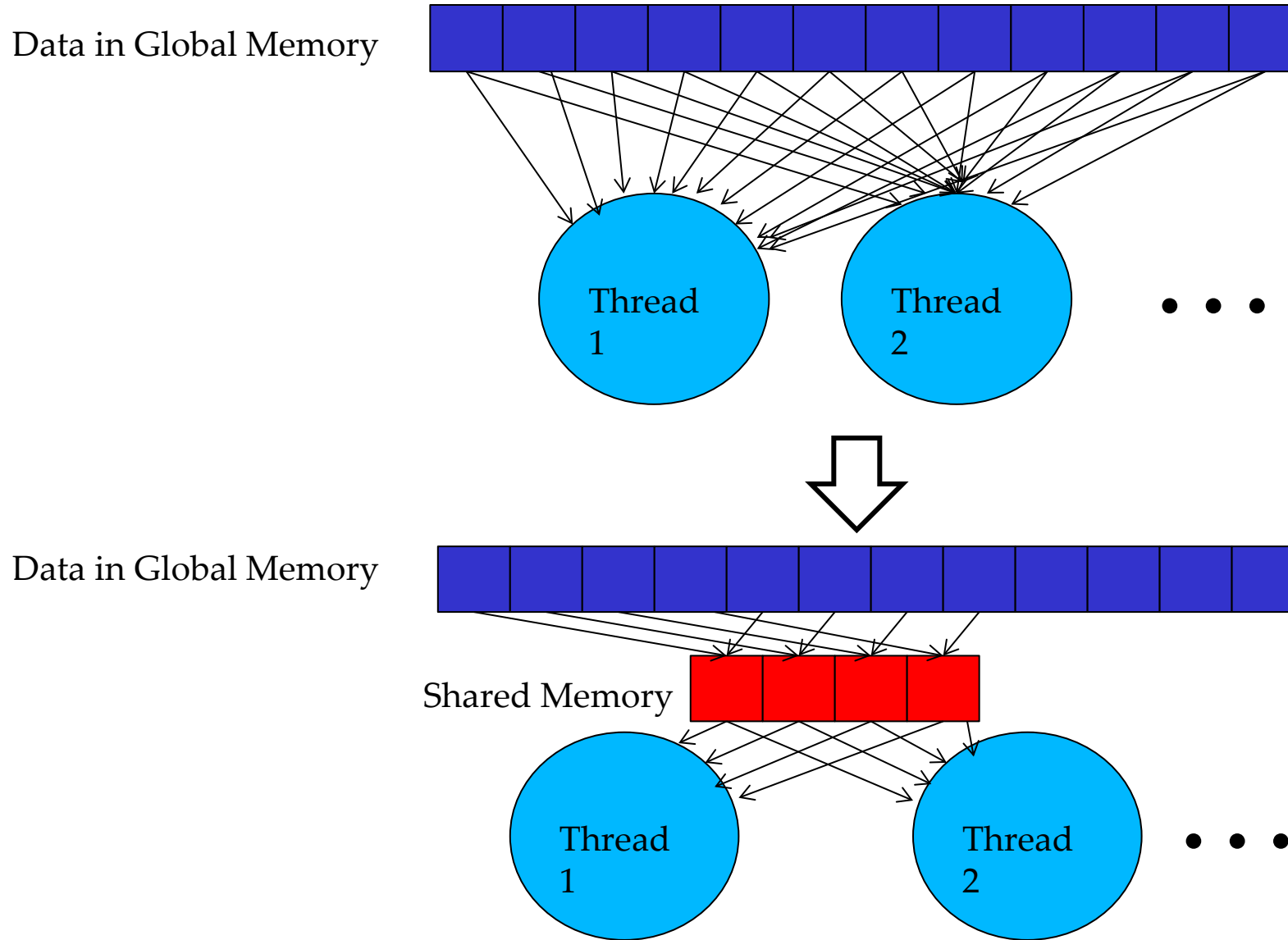
Declaring Shared Memory Arrays

```
__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

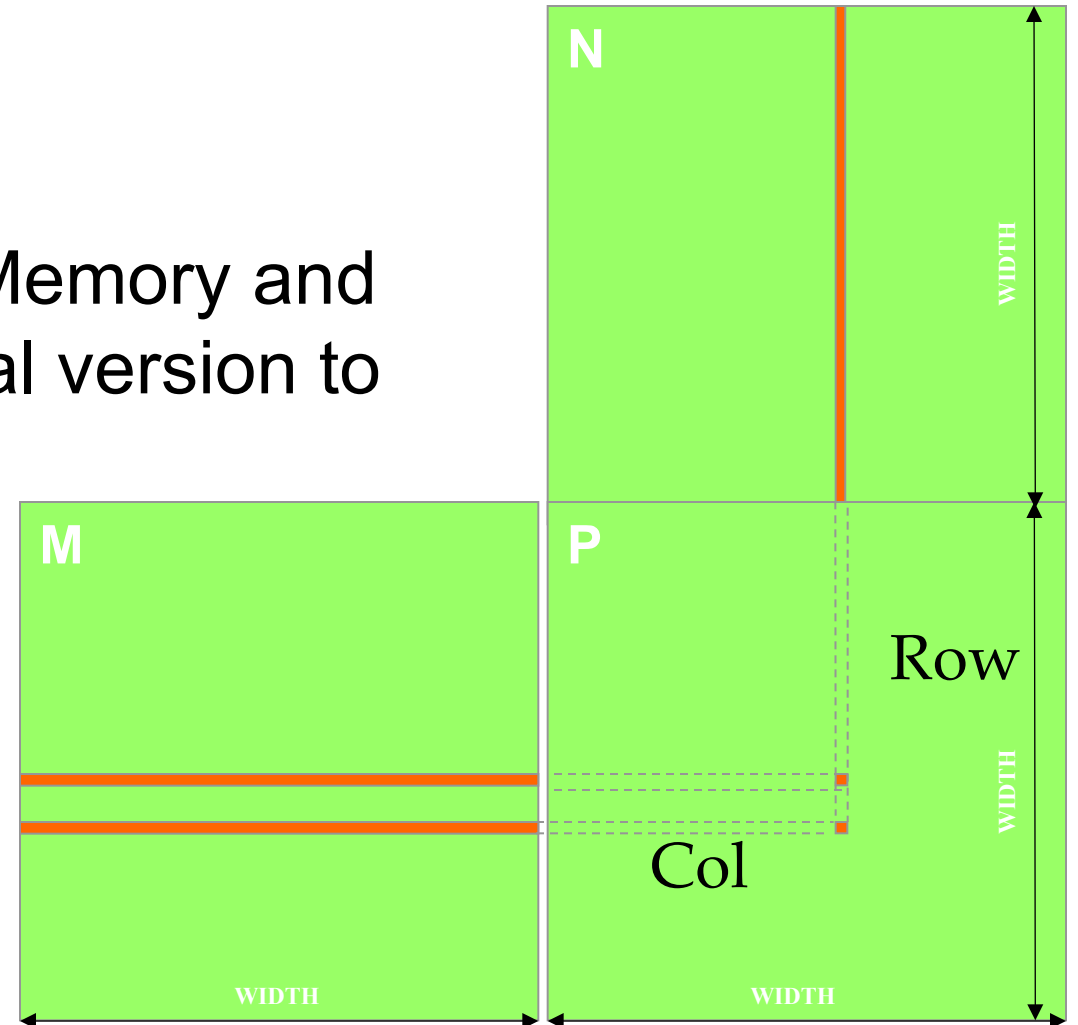


Outline of Technique

- 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

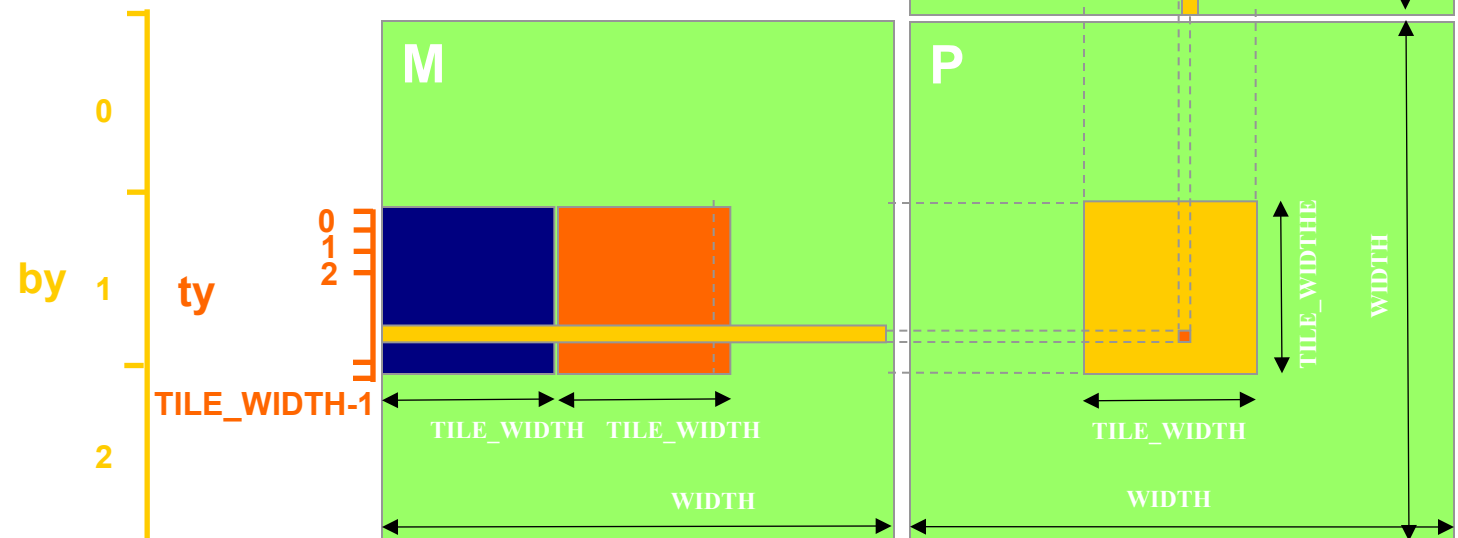
Use Shared Memory for data that will be reused

- 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

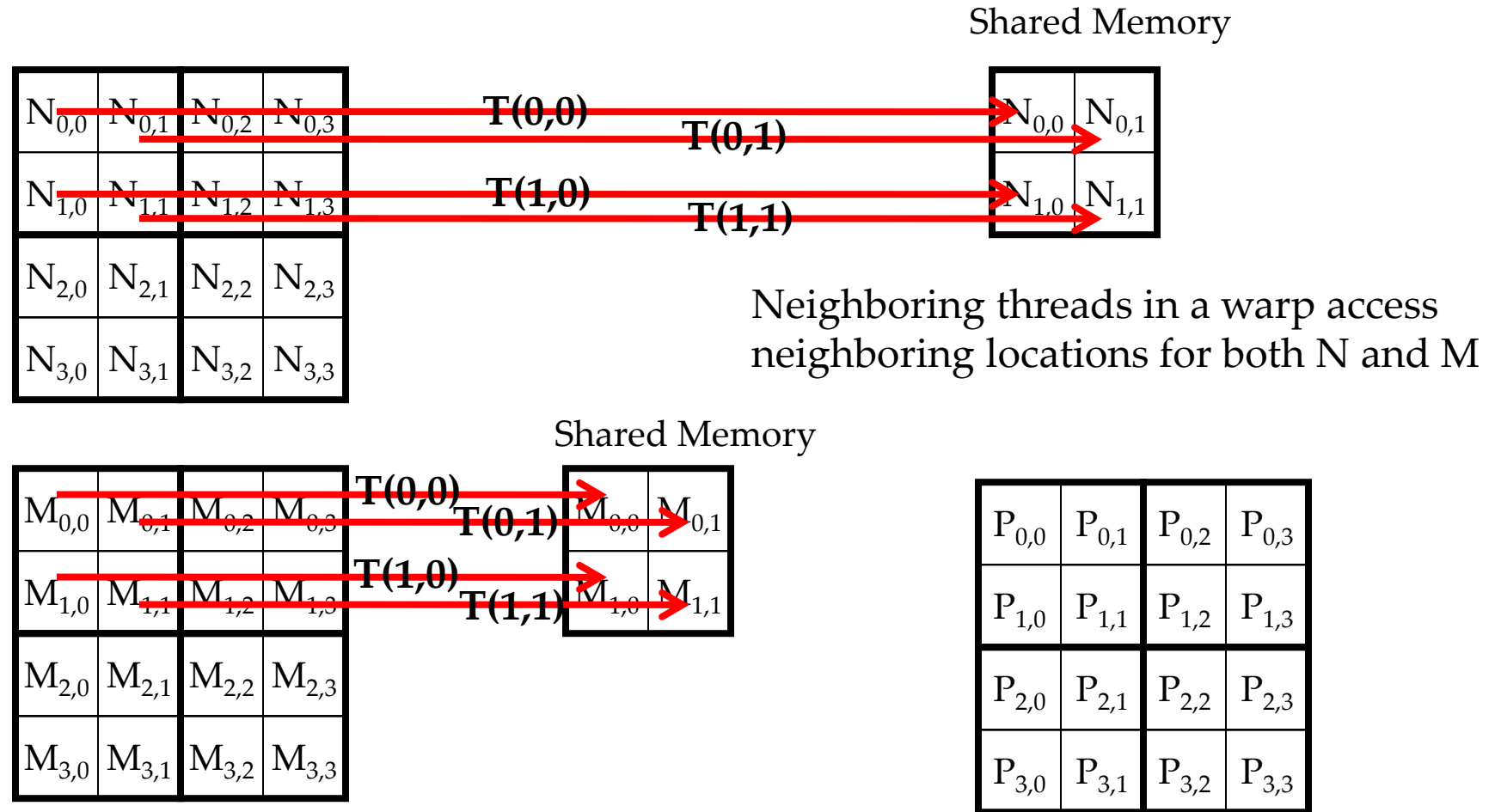
- Break up the execution of the kernel into phases so that the data accesses in each phase are focused on one tile of M and N
- For each tile:
 - Phase 1: Load tiles of M & N into share memory
 - Phase 2: Calculate partial dot product for tile of P



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)

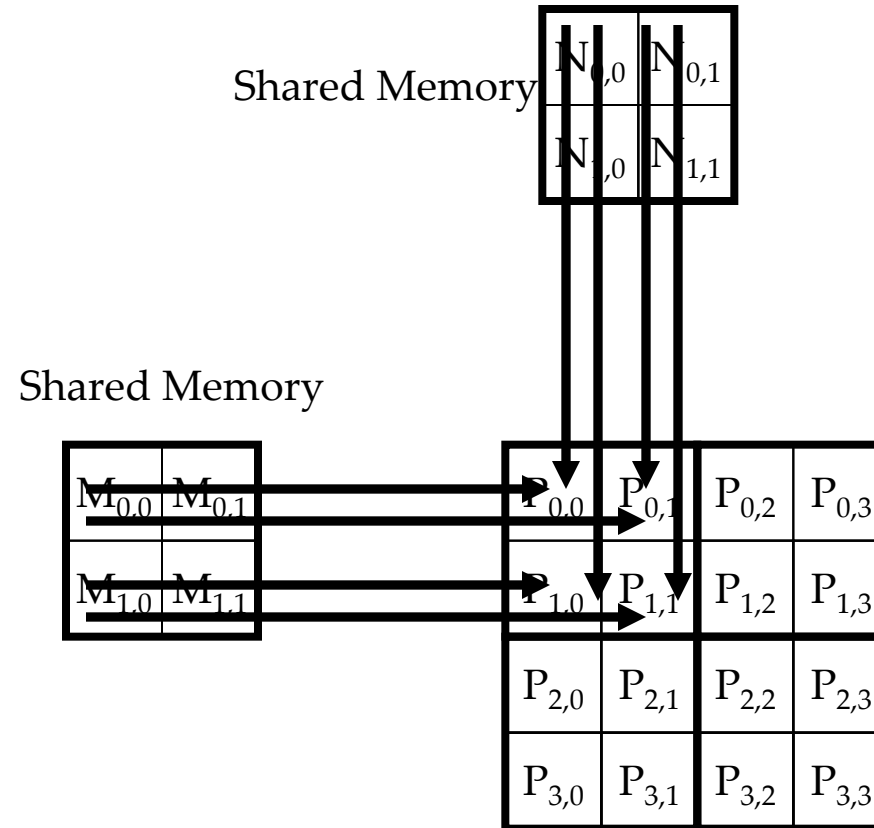


Work for Block (0,0)

Step 1

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	$N_{0,3}$
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	$N_{1,3}$
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	$N_{2,3}$
$N_{3,0}$	$N_{3,1}$	$N_{3,2}$	$N_{3,3}$

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

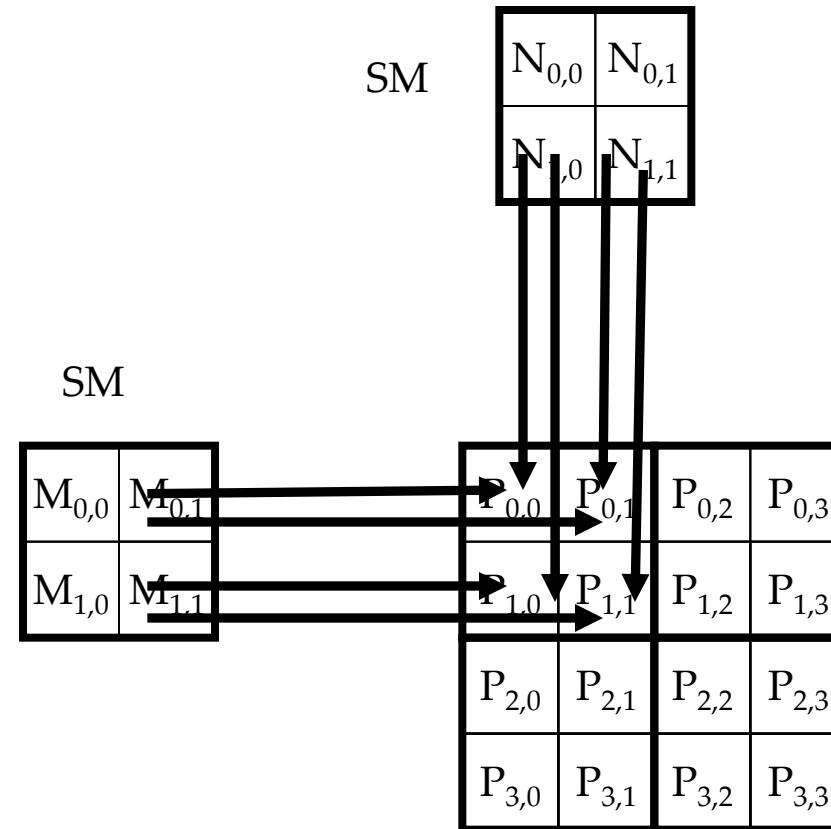


Work for Block (0,0)

Step 2

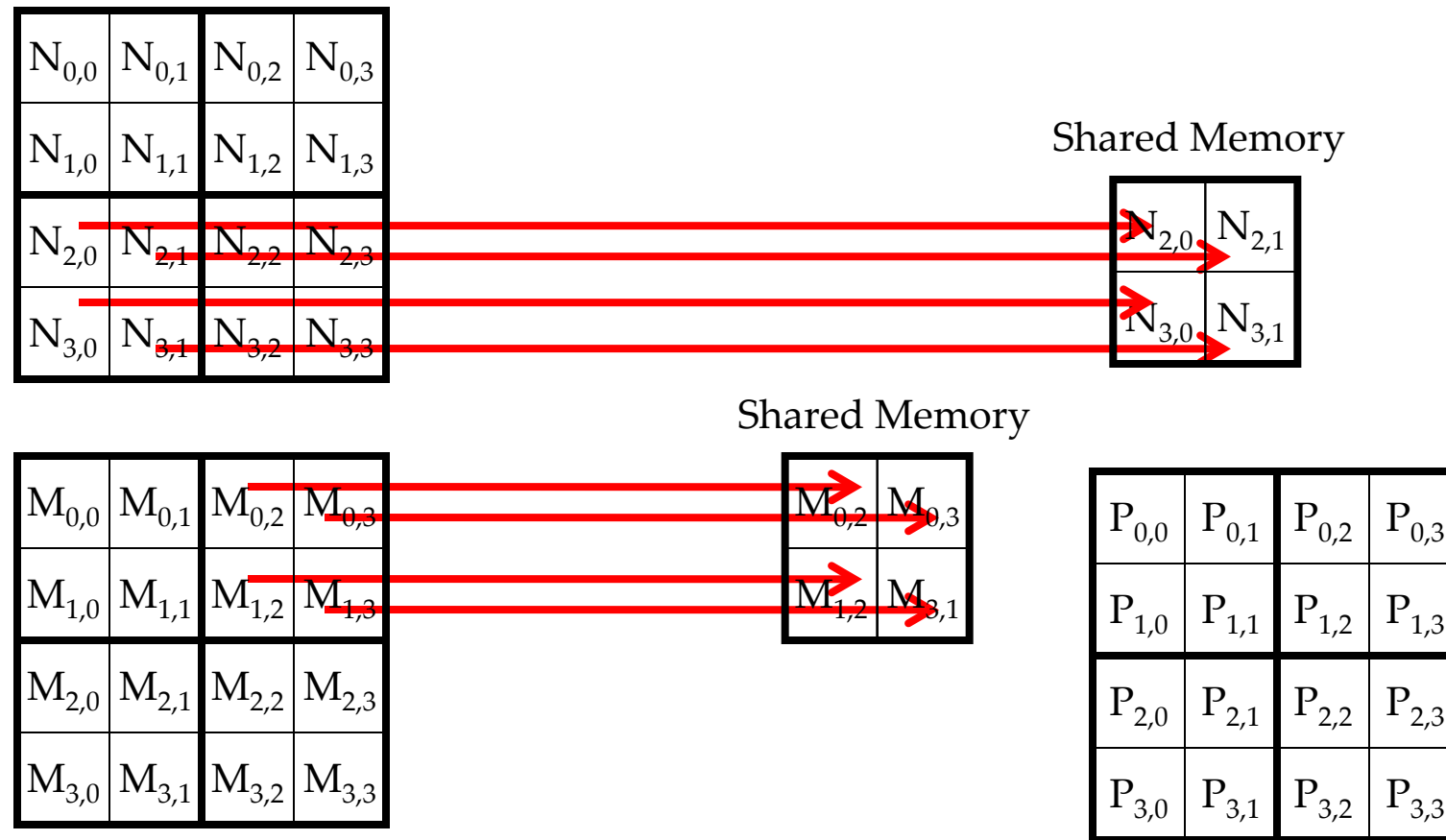
$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	$N_{0,3}$
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	$N_{1,3}$
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	$N_{2,3}$
$N_{3,0}$	$N_{3,1}$	$N_{3,2}$	$N_{3,3}$

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$



Work for Block (0,0)

Step 3

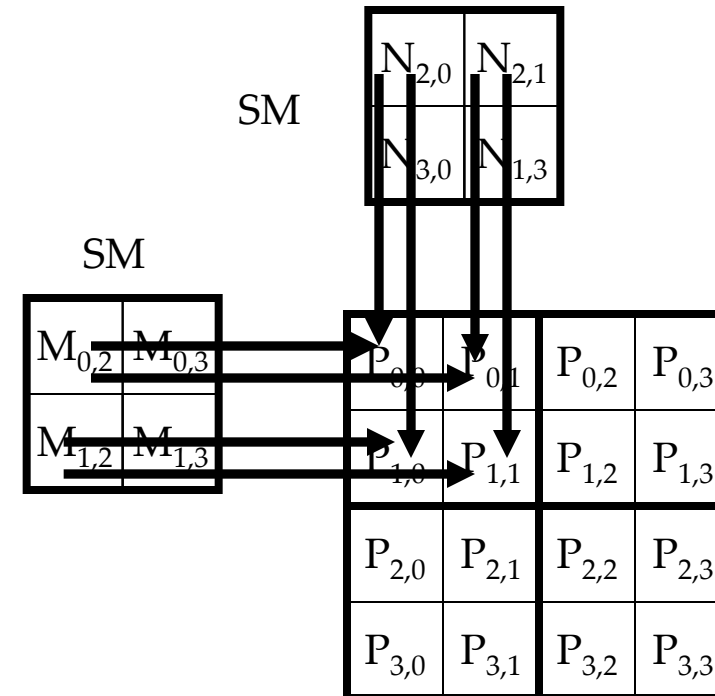


Work for Block (0,0)

Step 4

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	$N_{0,3}$
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	$N_{1,3}$
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	$N_{2,3}$
$N_{3,0}$	$N_{3,1}$	$N_{3,2}$	$N_{3,3}$

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

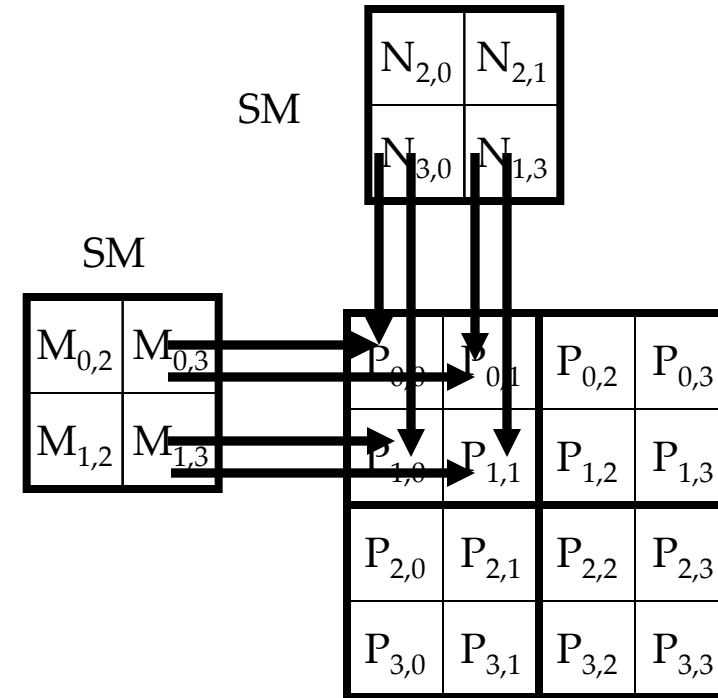


Work for Block (0,0)

Step 5

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	$N_{0,3}$
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	$N_{1,3}$
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	$N_{2,3}$
$N_{3,0}$	$N_{3,1}$	$N_{3,2}$	$N_{3,3}$

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$



Phase 1: Loading a Tile

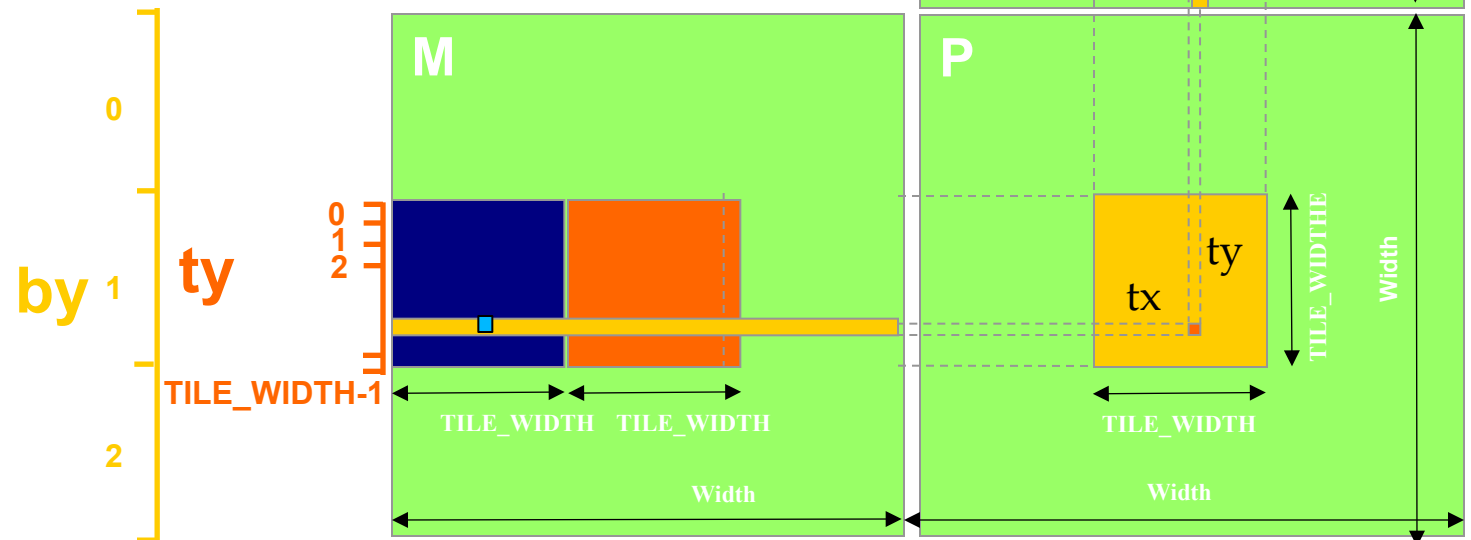
- All threads in a block participate
 - Each thread loads one M element and one N element in basic tiling code
- Assign the loaded element to each thread such that the accesses within each warp is coalesced (more later).

Loading an Input Tile 0

2D indexing for Tile 0

$M[\text{Row}][\text{tx}]$

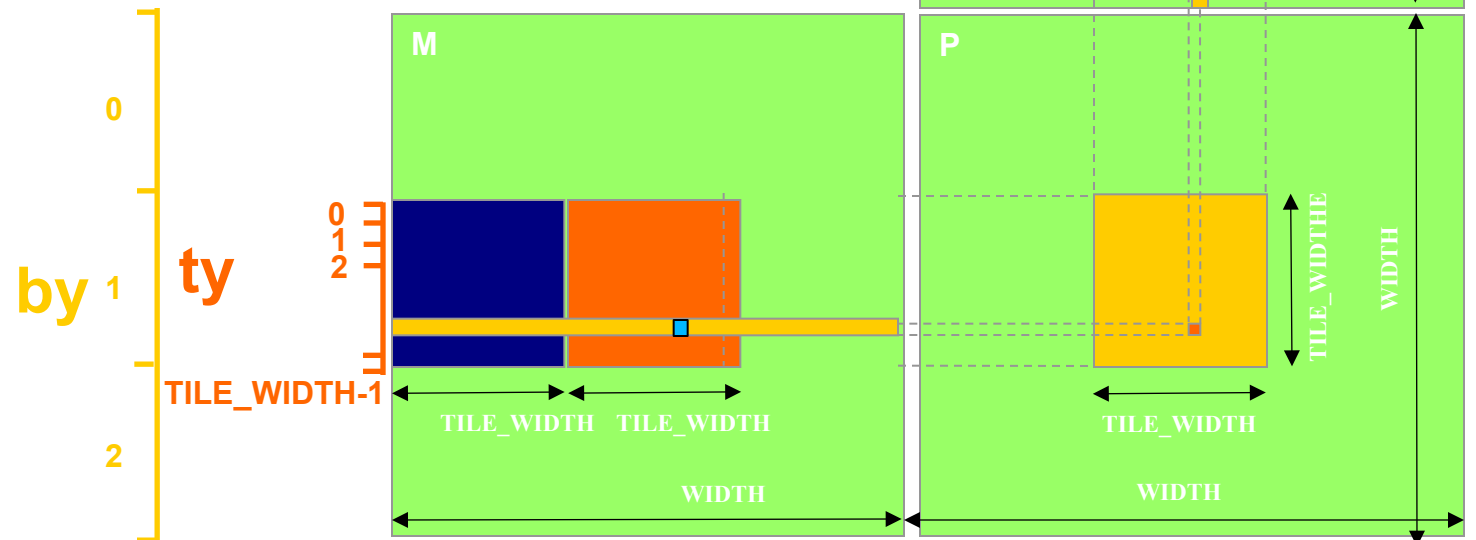
$N[\text{ty}][\text{Col}]$



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

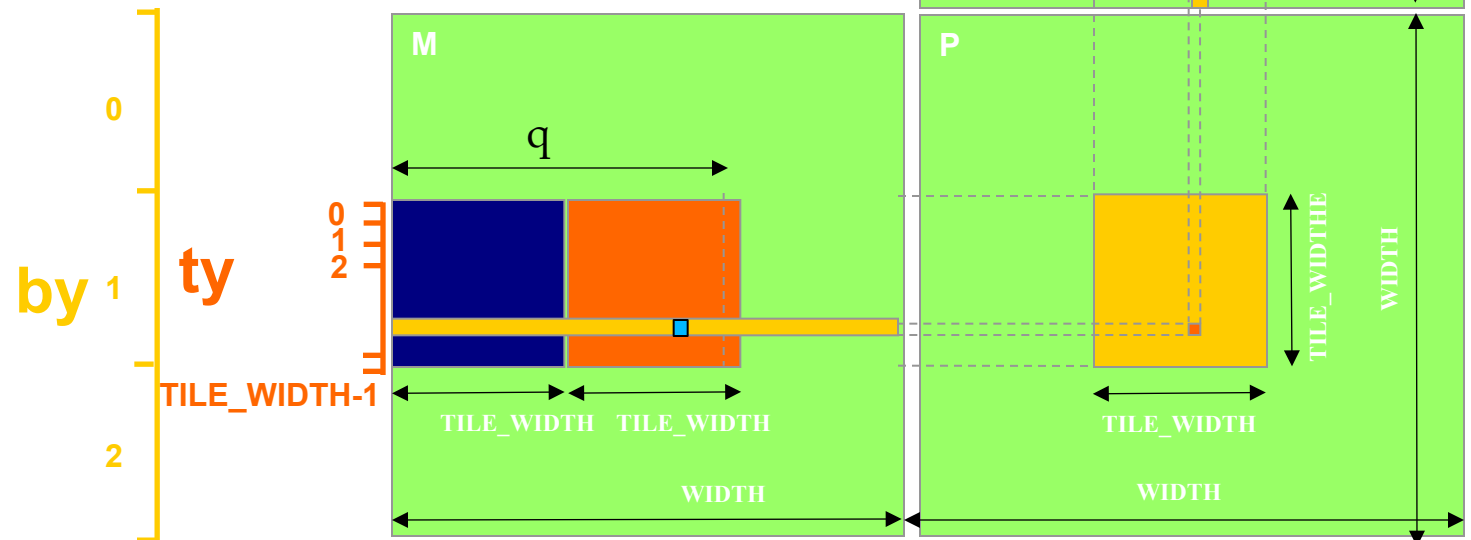
However, recall that M and N are dynamically allocated and can only use 1D indexing:

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

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

`N[q*TILE_WIDTH+ty][Col]`

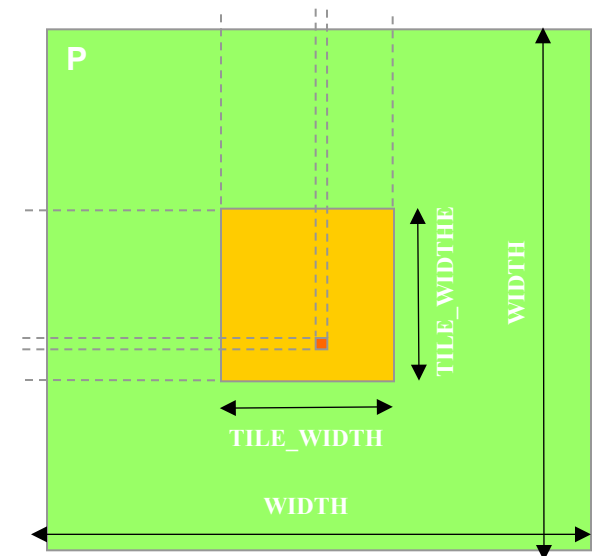
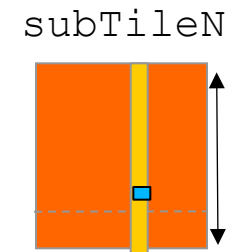
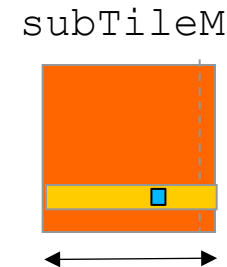
`N[(q*TILE_WIDTH+ty) * Width + Col]`



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

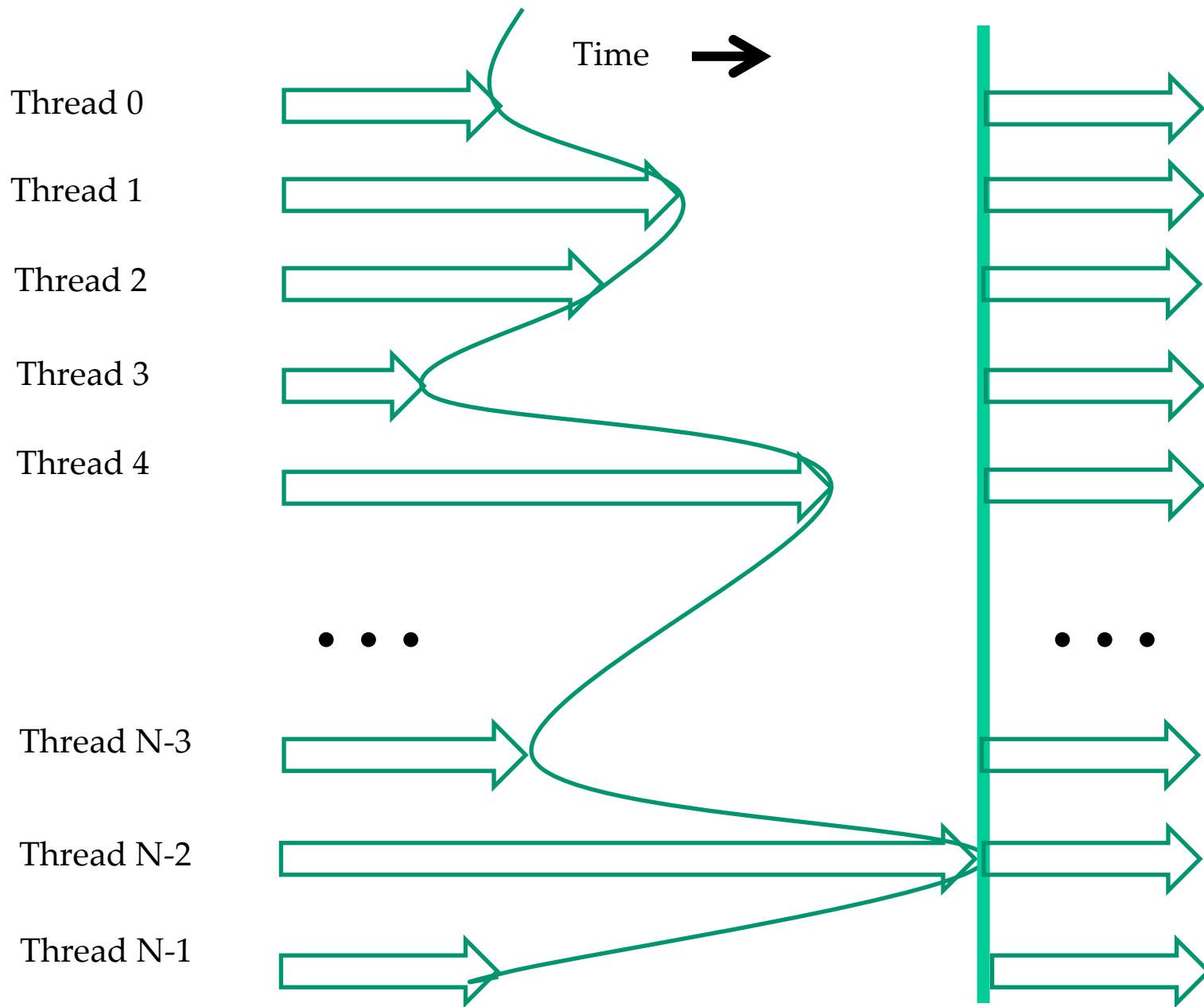
- **Bulk synchronous execution:**
threads execute roughly in unison
 1. Do some work
 2. Wait for others to catch up
 3. Repeat
- **Much easier programming model**
 - Threads only parallel within a section
 - Debug lots of little programs
 - Instead of one large one.
- **Dominates high-performance applications**

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.



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)
{
1.  __shared__ float subTileM[TILE_WIDTH][TILE_WIDTH];
2.  __shared__ float subTileN[TILE_WIDTH][TILE_WIDTH];

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

    // Identify the row and column of the P element to work on
5.  int Row = by * TILE_WIDTH + ty;
6.  int Col = bx * TILE_WIDTH + tx;
7.  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!
8.  for (int q = 0; q < Width/TILE_WIDTH; ++q) {
    // Collaborative loading of M and N tiles into shared memory
9.      subTileM[ty][tx] = M[Row*Width + q*TILE_WIDTH+tx];
10.     subTileN[ty][tx] = N[(q*TILE_WIDTH+ty)*Width+Col];
11.     __syncthreads();
12.     for (int k = 0; k < TILE_WIDTH; ++k)
13.         Pvalue += subTileM[ty][k] * subTileN[k][tx];
14.     __syncthreads();
15. }
16. 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
 $(150/4)*16 = \mathbf{600\ GFLOPS!}$
- **32x32 tiles** use each operand for 32 operations
 - **reduce global** memory **accesses by** a factor of **32**
 - **150 GB/s** bandwidth supports
 $(150/4)*32 = \mathbf{1,200\ GFLOPS!}$
 - **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 $2 \times 1024 \times 4\text{B} = 8\text{kB}$ of shared memory,
 - which limits active blocks to 8;
 - max. of 2,048 threads per SM,
 - which limits blocks to 2.
 - Thus, up to $2 \times 2,048 = \mathbf{4,096}$ **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
 - ...

Two vertical lines, one blue and one orange, are positioned on the left side of the slide.

**ANY MORE QUESTIONS?
READ CHAPTER 4!**