ECE408/CS483/CSE408 Fall 2022

Applied Parallel Programming

Lecture 5: Locality and Tiled Matrix Multiply

Course Reminders

- Lab 2 is out, it is due this Friday
- Lowest lab grade will be dropped from the final grade
  - Thus no late submissions are allowed for labs

Objective

- To learn to evaluate the performance implications of global memory accesses
- To prepare for MP3: tiled matrix multiplication
- To learn to assess the benefit of tiling

Parallel Memory Updates

```
__global__
void ThreadIncrement(int *x)
{
   // Each thread increments x
   x++;
}
```

load x from global memory to register

add x = x+1

store x to global memory

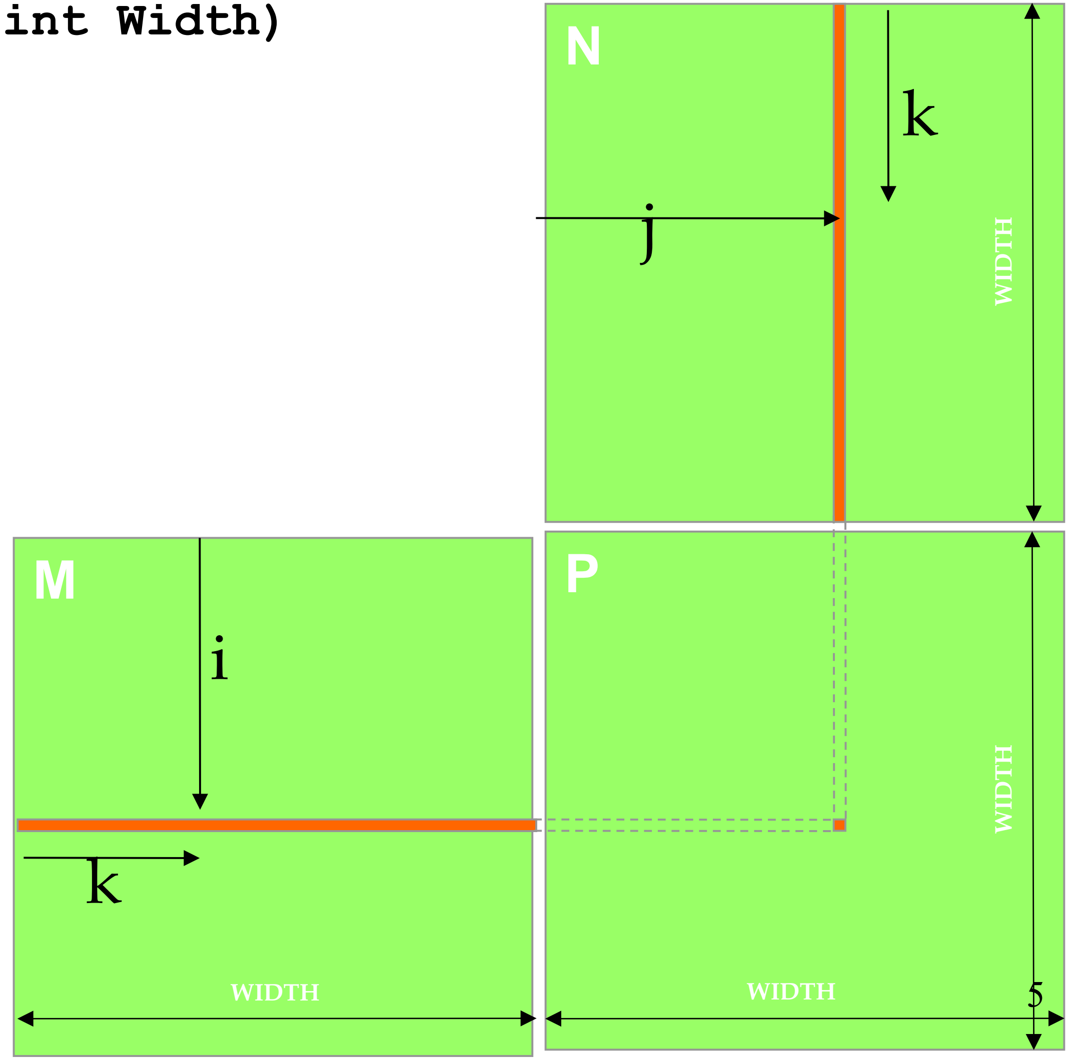
atomic Add(x,1)
unit or aromically

What is the value of x if we have 128 threads in execution?

data race
1-128

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;
        }
}
```



MatMult Kernel: Width = 8, BLOCK_WIDTH = 4

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

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 d_P 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 d_P
      for (int k = 0; k < Width; ++k)
          Pvalue += d_M[Row * Width + k] * d_N[k * Width + Col];
      d_P[Row * Width + Col] = Pvalue;
   }
}
```

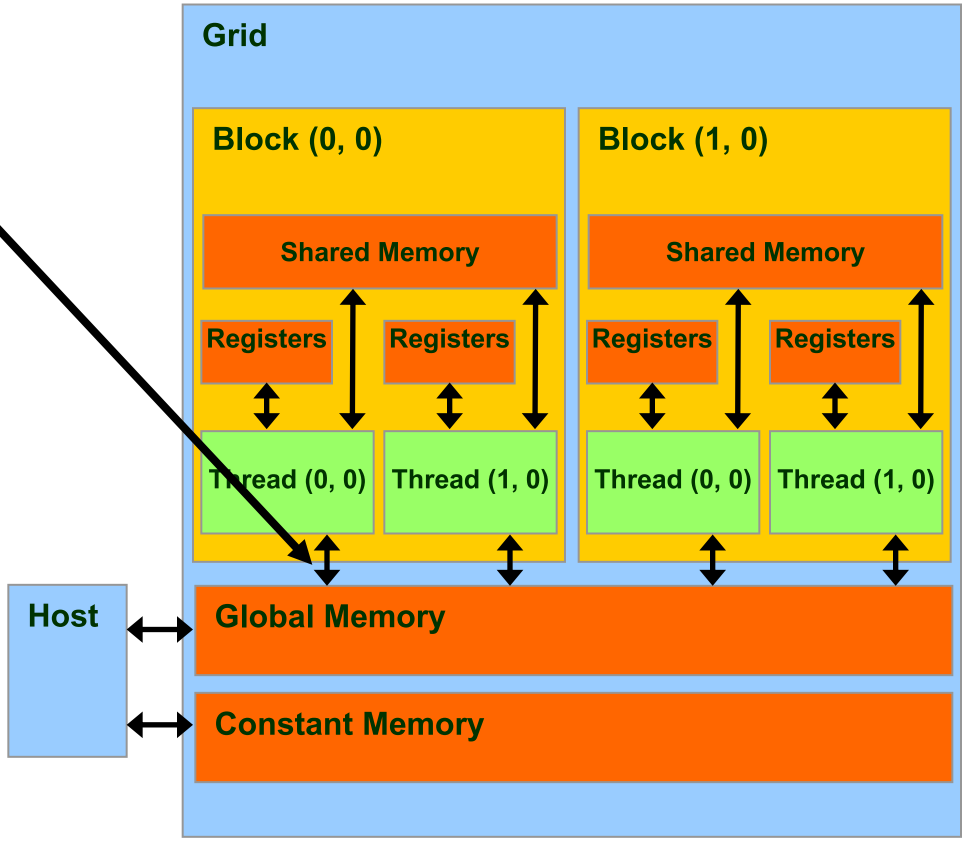
limited by memory bandwidth

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 of memory for each FLOP
  - 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



Avoid the BW Bottleneck by exploiting Reuse

- Each element of M and N is used Width times in calculating P

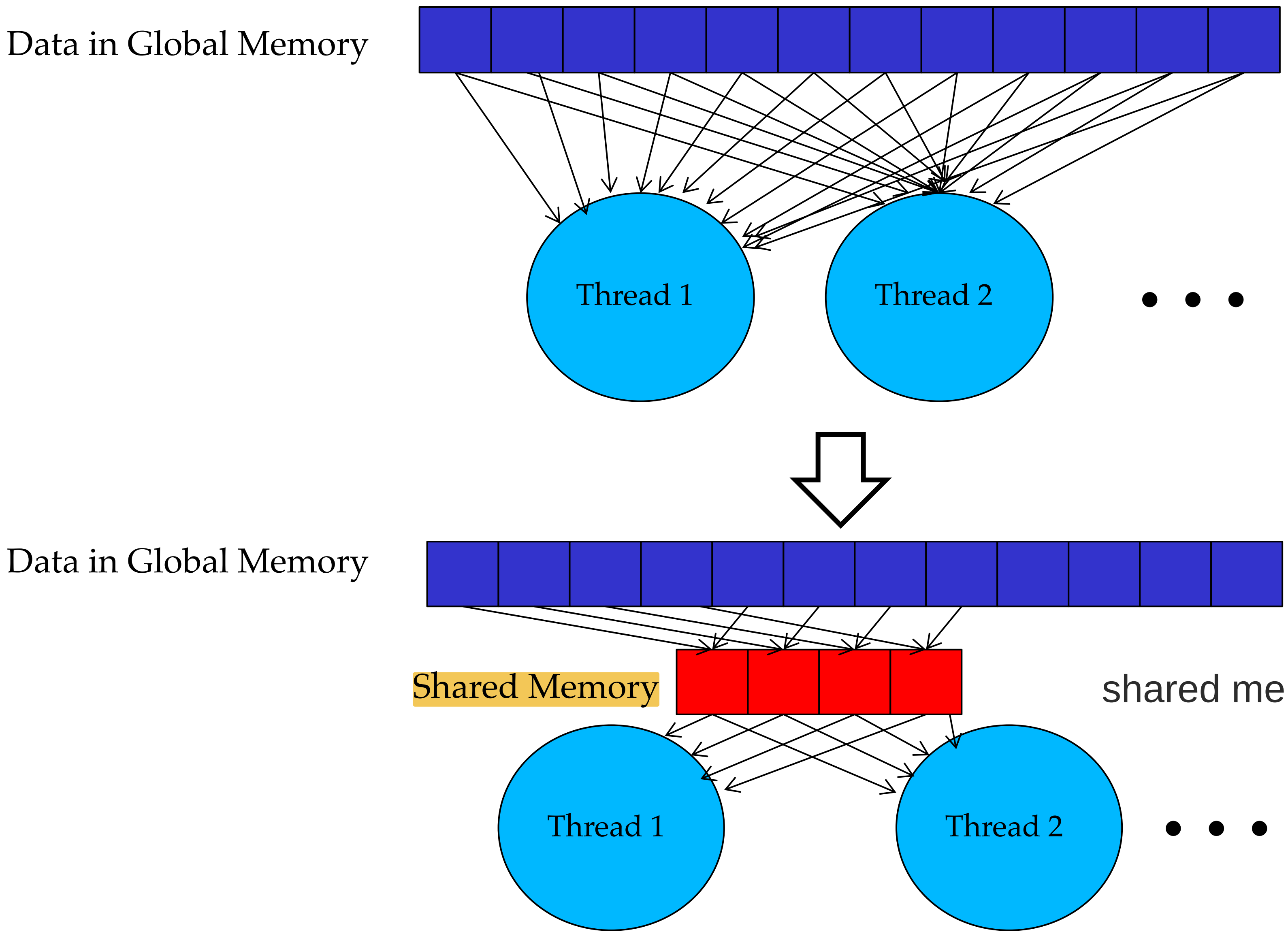
- To avoid BW limitations, exploit data reuse by leveraging the per SM shared memory

software managed shared memory

- Partition data into tiles that fit into the shared memory

- Each thread block uses the following strategy:
  - Load the tile from global memory to shared memory
  - Perform the computation on the tile from shared memory; each thread can efficiently access any data element
  - Copy the result from shared memory to global memory

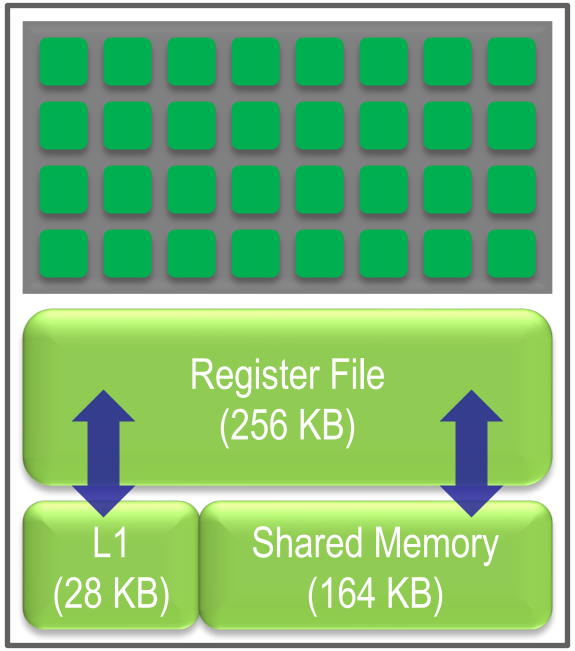
Shared Memory Tiling Basic Idea



shared memory 更快

SM Memory Architecture

Execution Hardware



SM Memories
(Shown for an Ampere-class GPU)

- registers (~1 cycle)
- shared memory (~5 cycles)
- cache/constant memory (~5 cycles)
- global memory (~500 cycles)

Declaring Shared Memory Arrays

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

    __shared__ float subTileM[TILE_WIDTH][TILE_WIDTH];
    __shared__ float subTileN[TILE_WIDTH][TILE_WIDTH];
```

declare variable in shared memory
shared by all the thread in the same block

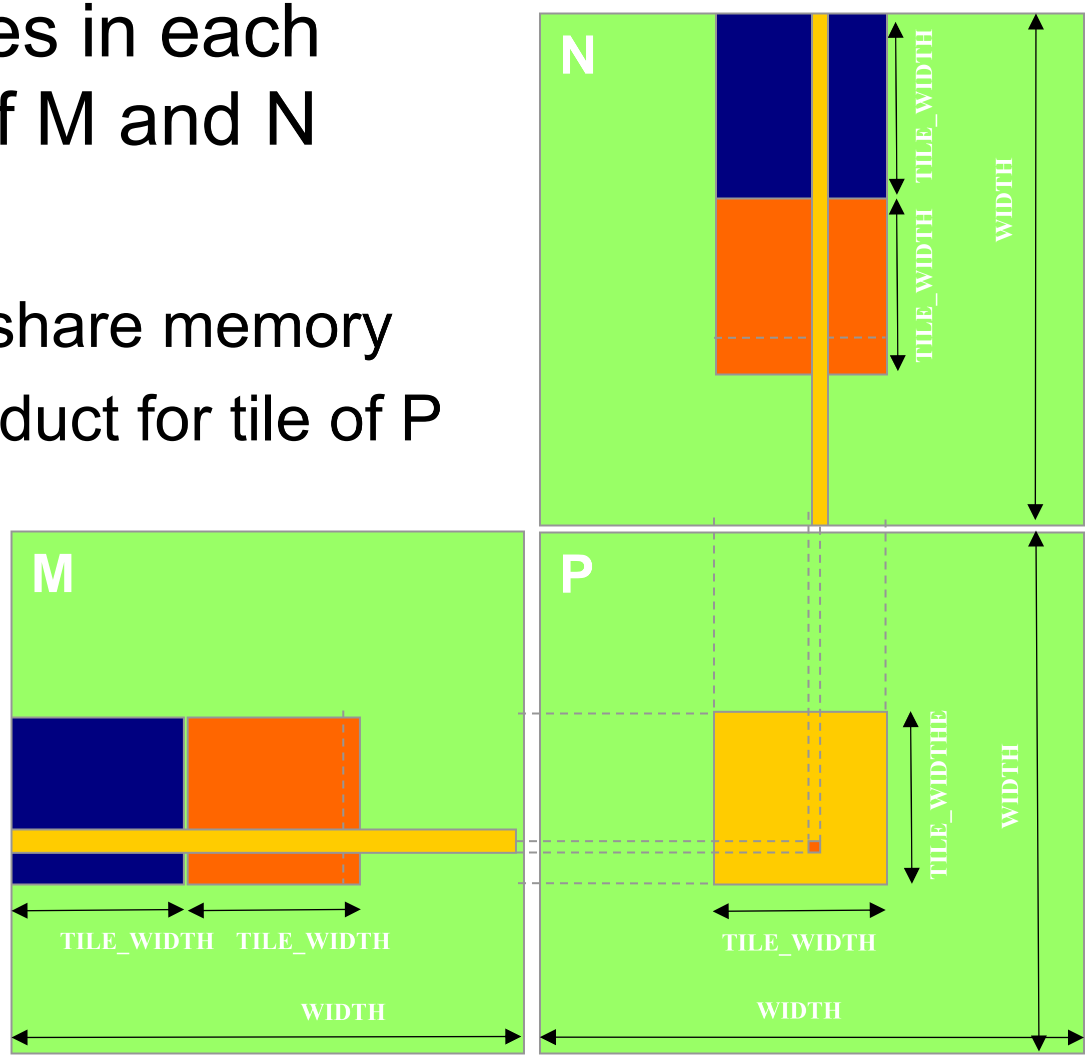
Common across all threads in a block

Outline of Technique

- Identify a tile of global data with high data reuse
- Load the tile from global memory into shared memory
- Threads in the block access their data from shared memory
- Move on to the next block/tile

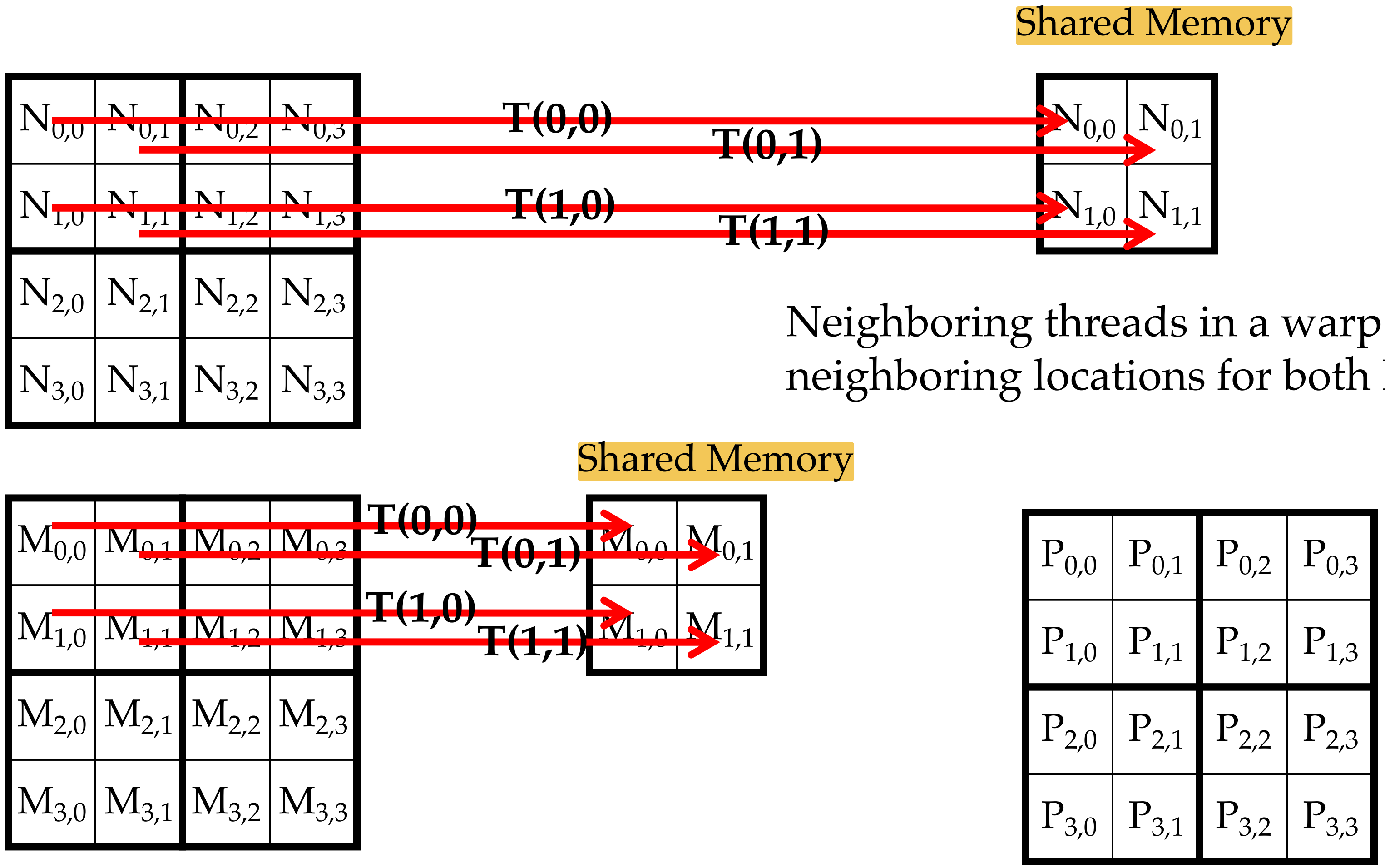
Tiled Multiply

- 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 Tiles for Block (0,0)

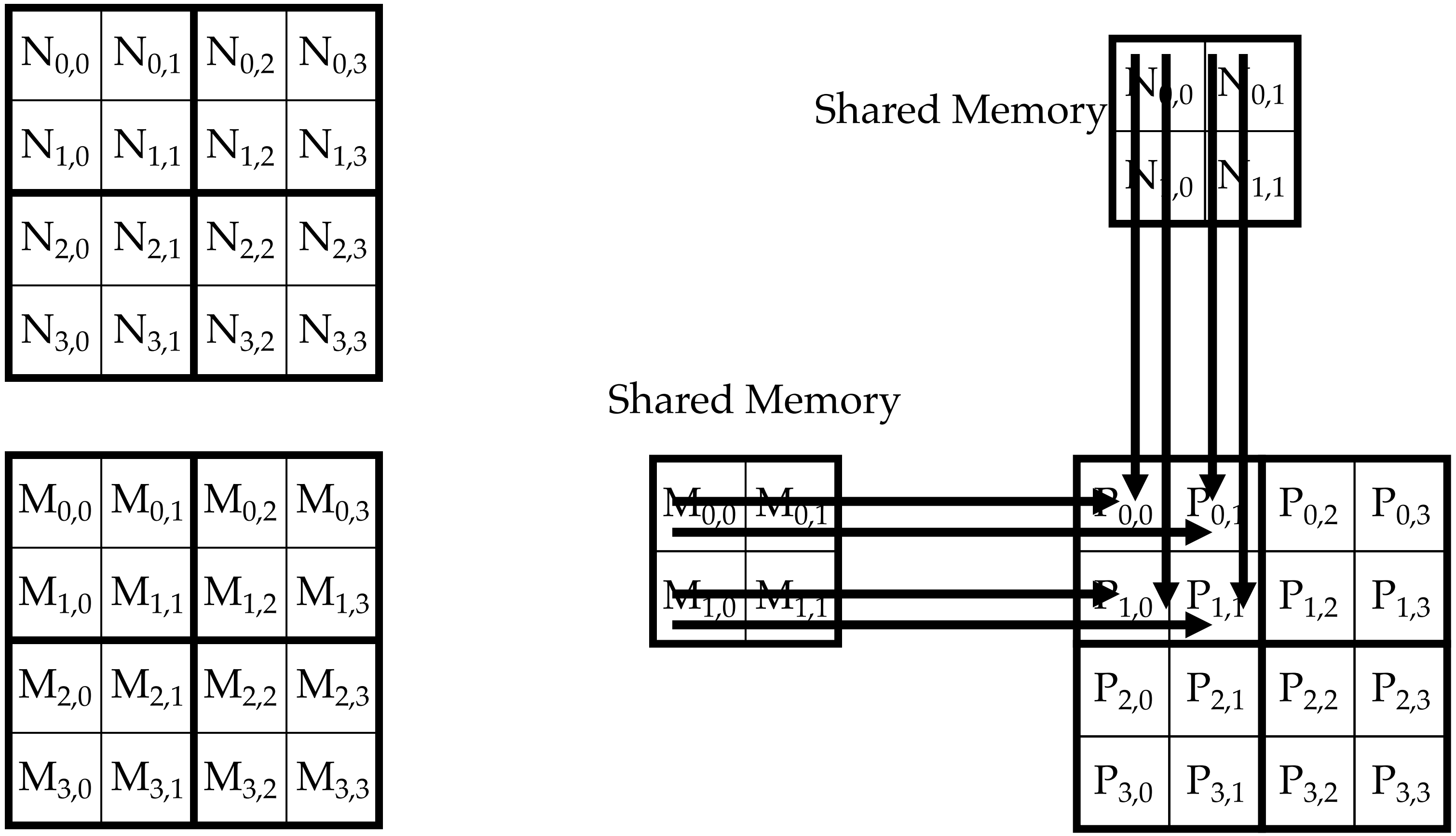
将Tile从global memory中load进入shared memory



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

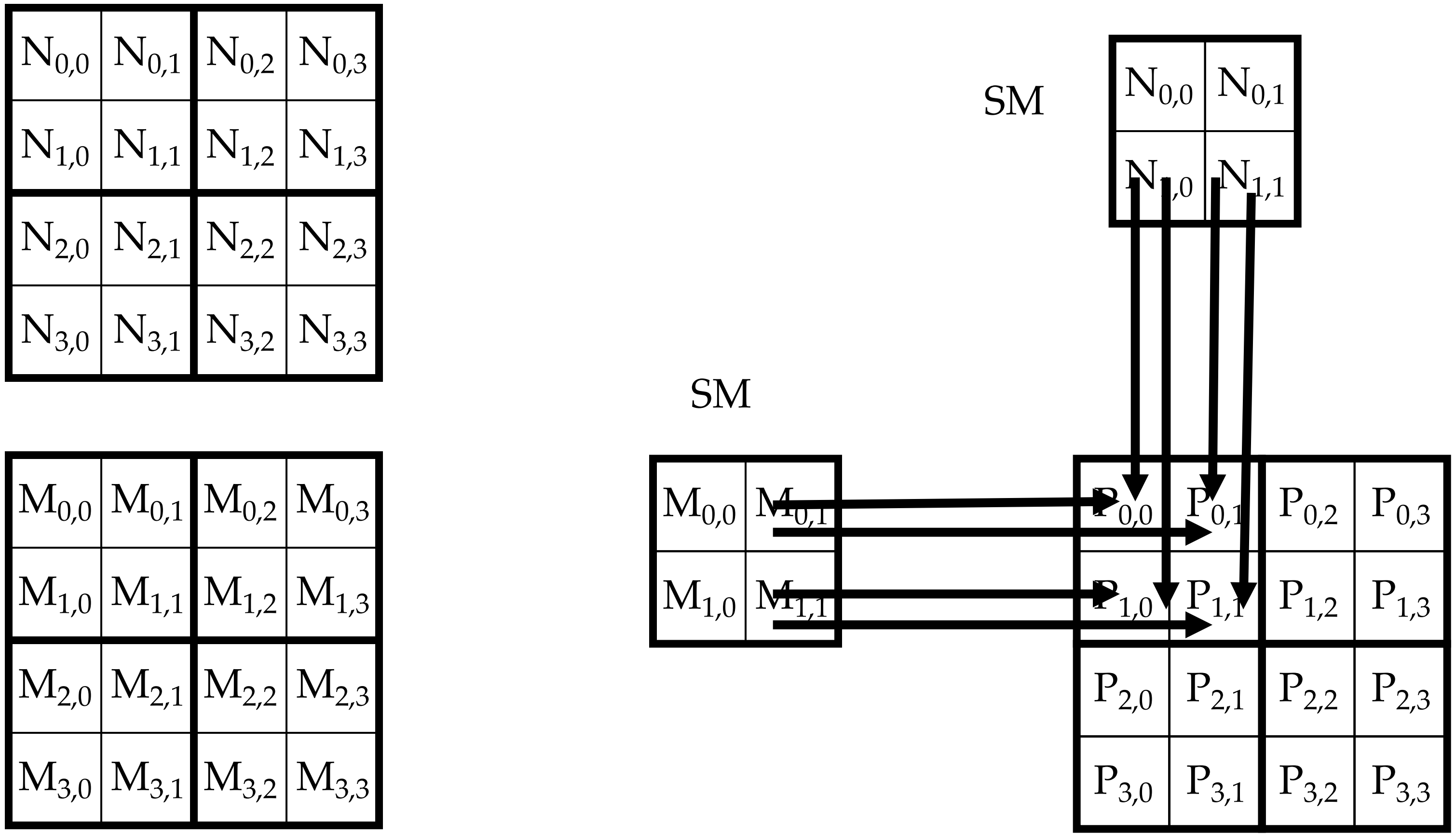
Work for Block (0,0)

Step 1



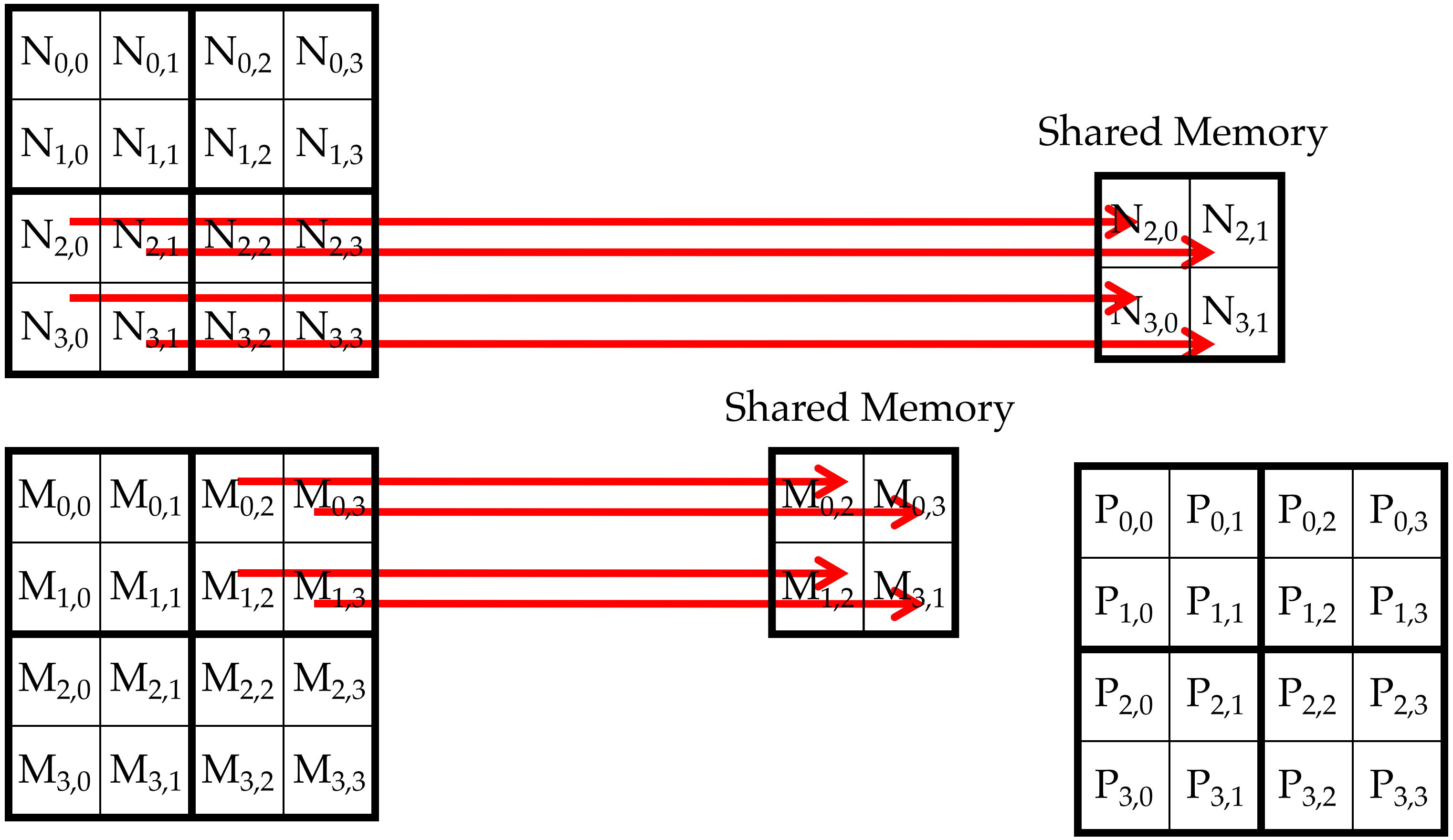
Work for Block (0,0)

Step 2



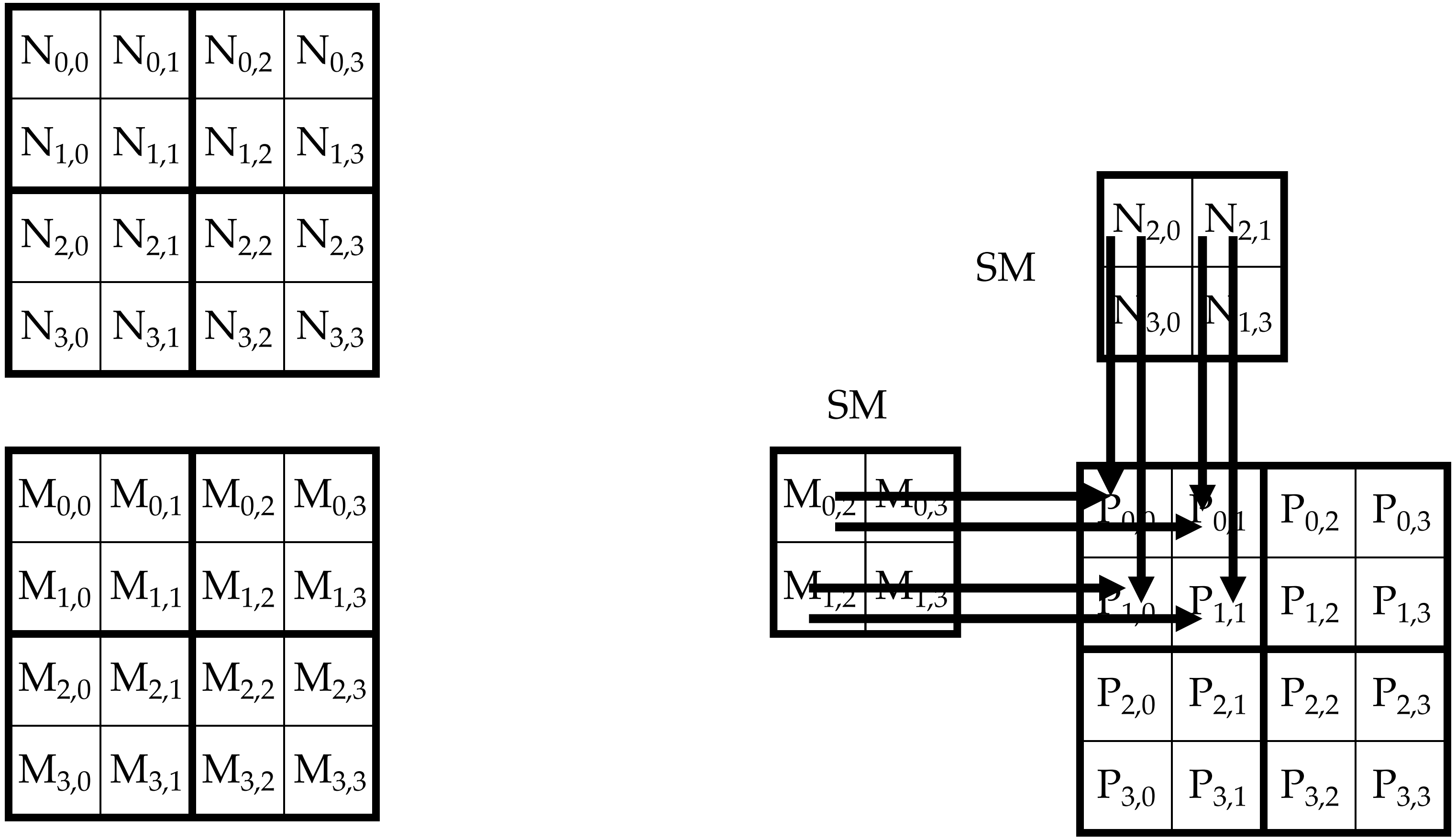
Work for Block (0,0)

Step 3



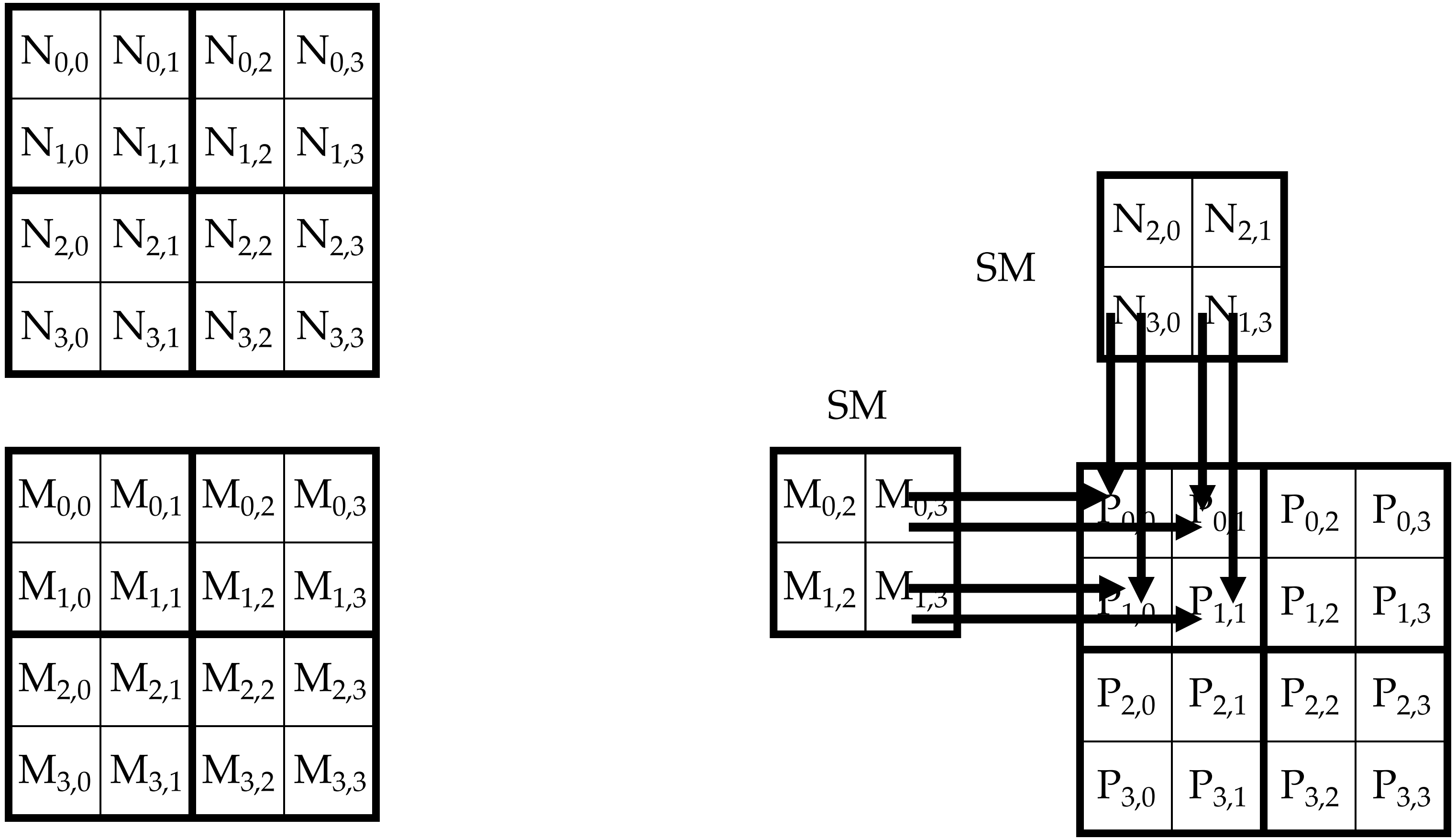
Work for Block (0,0)

Step 4



Work for Block (0,0)

Step 5



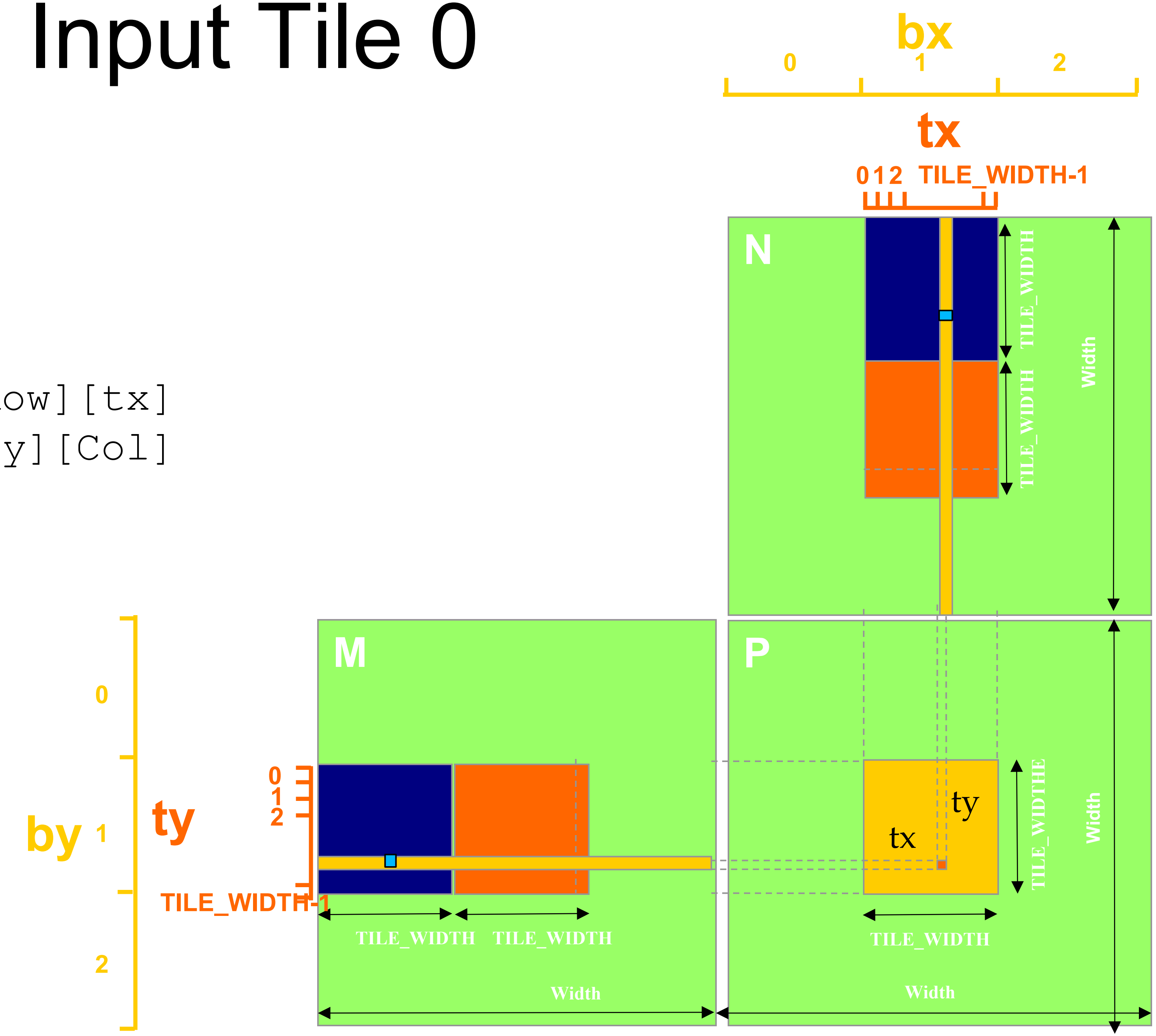
Phase 1: Loading a Tile

- All threads in the block loads one M element and one N element into shared memory

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

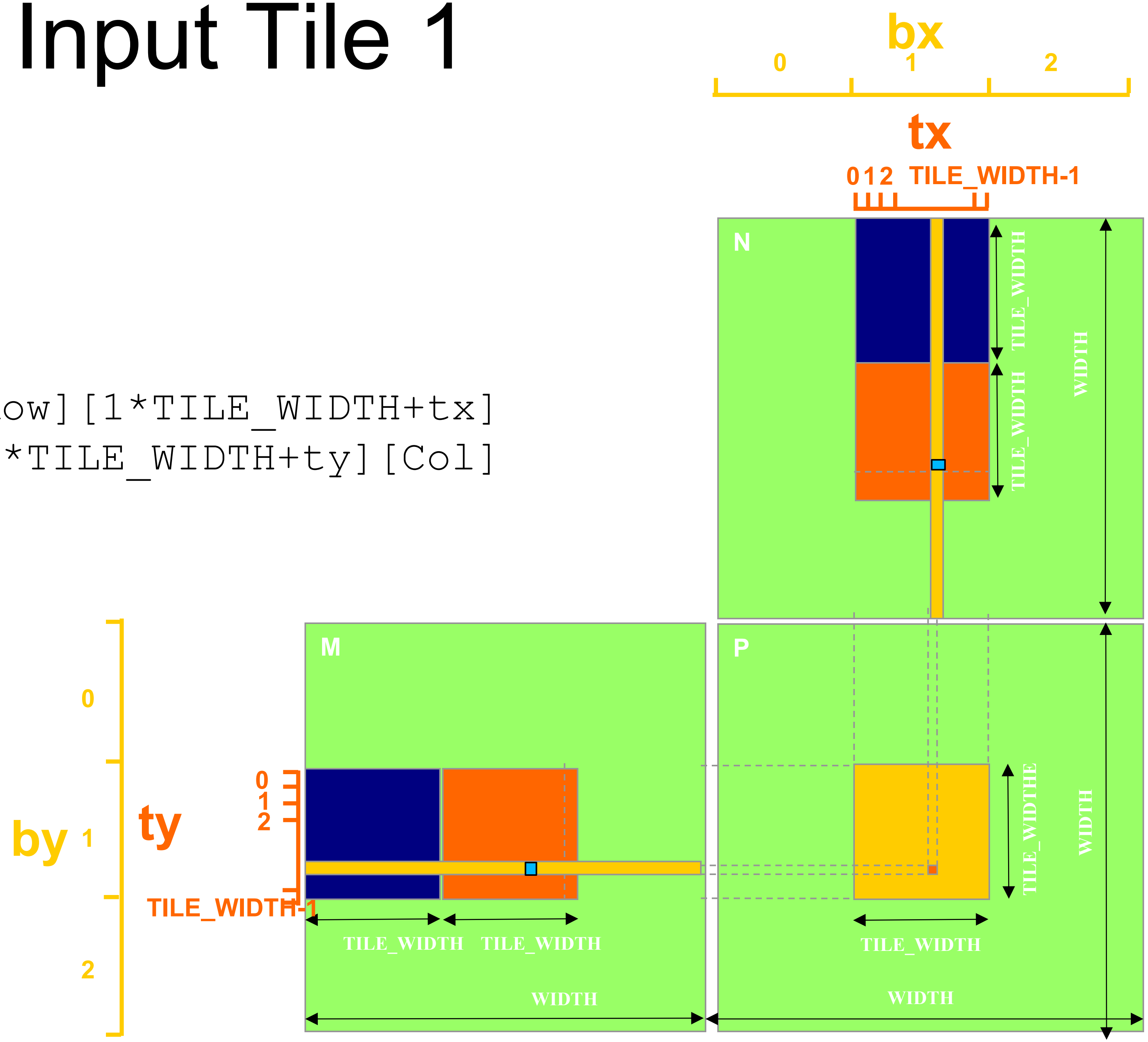
Loading an Input Tile 0

```
tx = threadIdx.x;
ty = threadIdx.y;

subTileM[ty][tx] = M[Row][tx]
subTileN[ty][tx] = N[ty][Col]
```



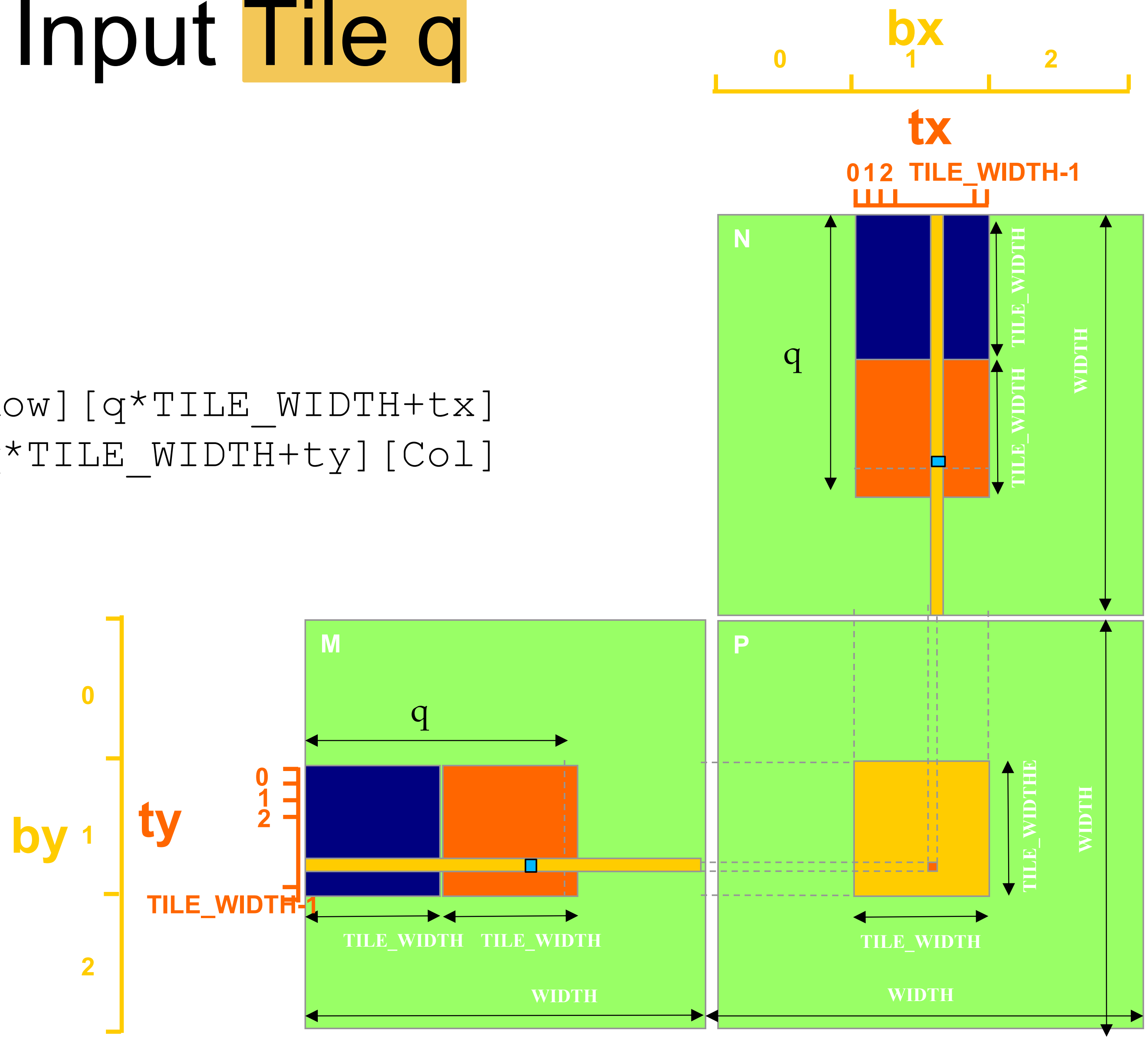
Loading an Input Tile 1

```
tx = threadIdx.x;
ty = threadIdx.y;

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



Loading an Input Tile q

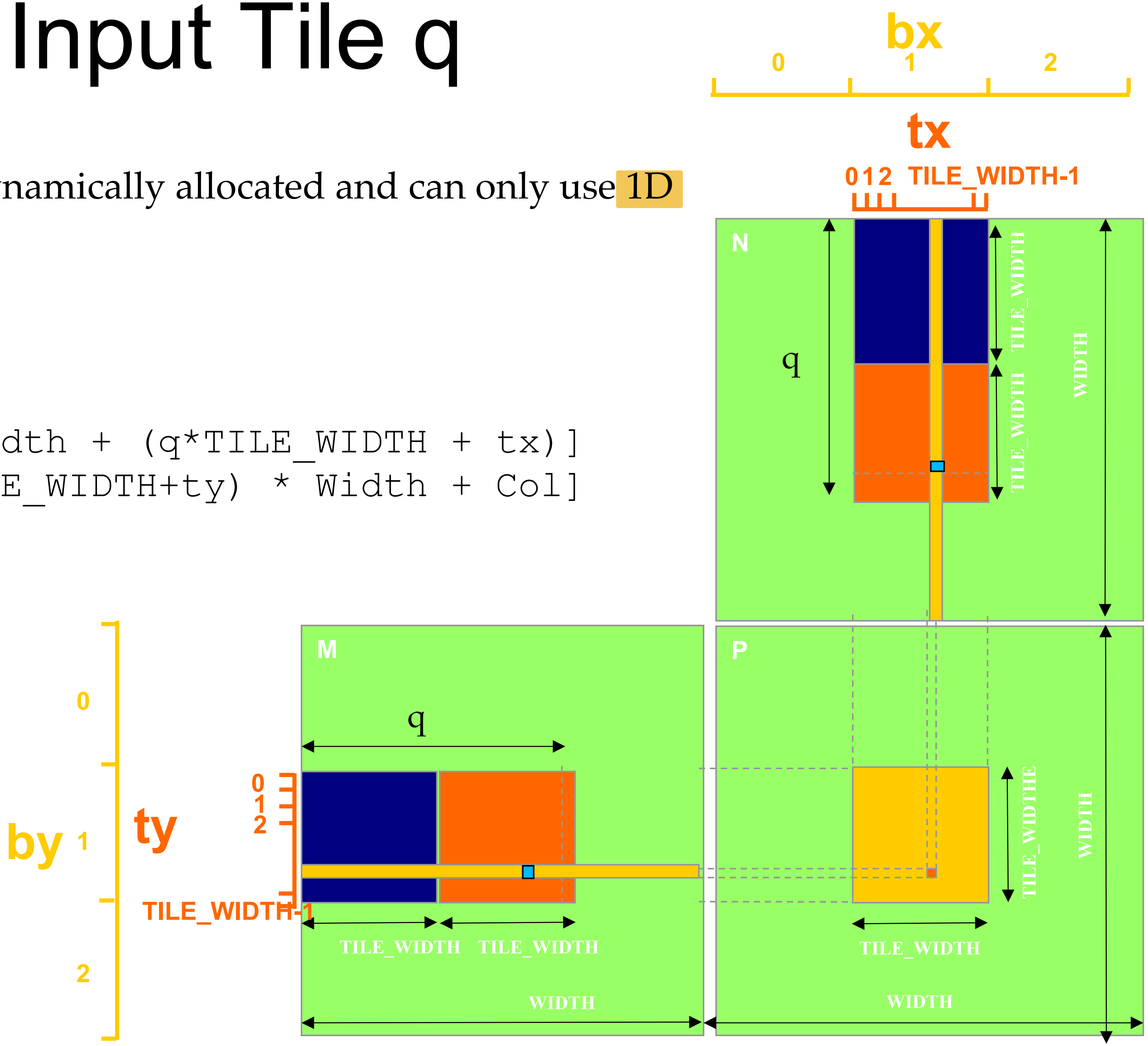
```
tx = threadIdx.x;
ty = threadIdx.y;

subTileM[ty][tx] = M[Row][q*TILE_WIDTH+tx]
subTileN[ty][tx] = N[q*TILE_WIDTH+ty][Col]
```



Loading an Input Tile q

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

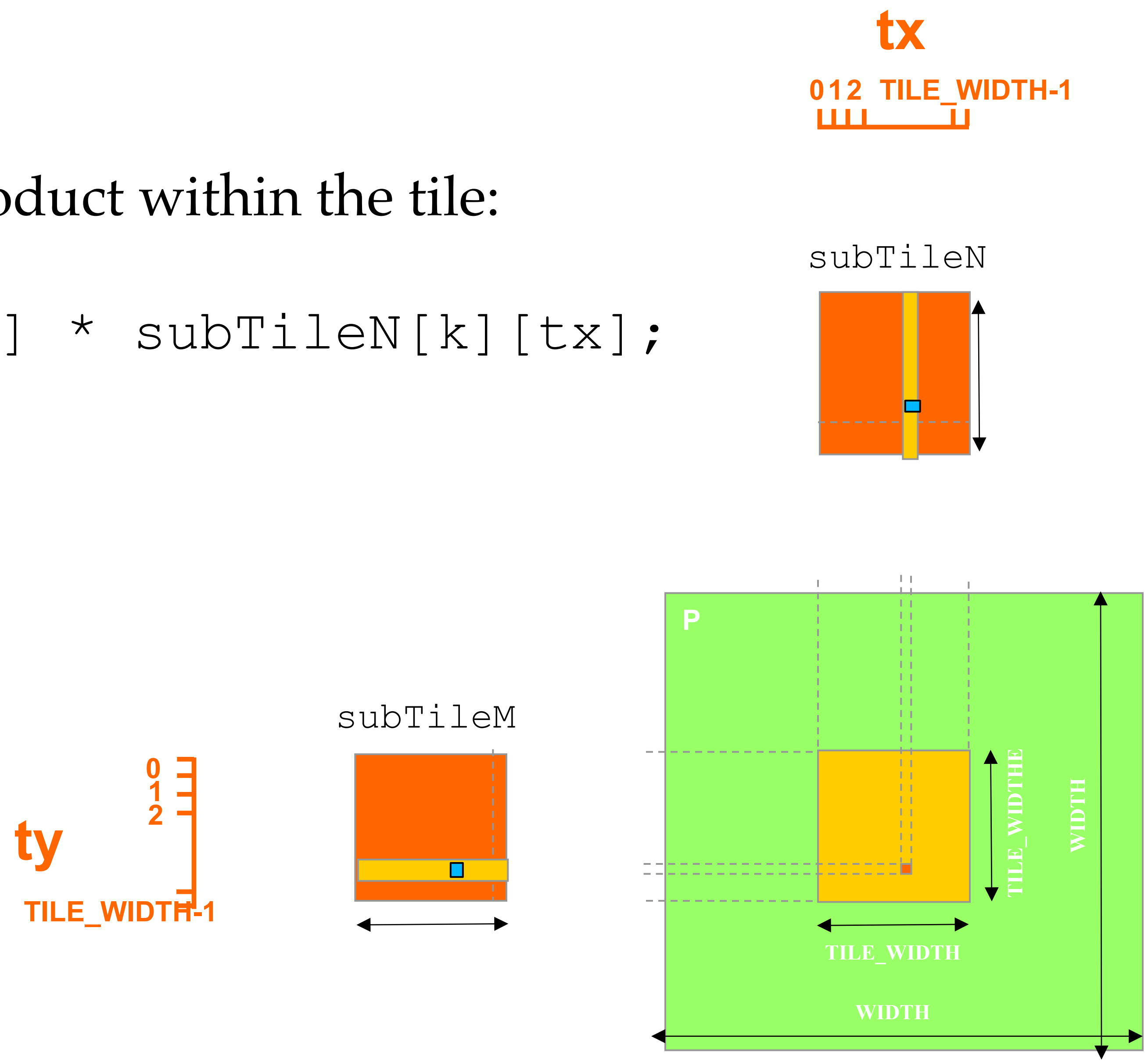
```
tx = threadIdx.x;
ty = threadIdx.y;

subTileM[ty][tx] = M[Row*Width + (q*TILE_WIDTH + tx)]
subTileN[ty][tx] = N[(q*TILE_WIDTH+ty) * Width + Col]
```



Phase 2: Compute partial product

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

```
PValue += subTileM[ty][k] * subTileN[k][tx];
```



(Incorrect) Tiled Matrix Multiplication Kernel

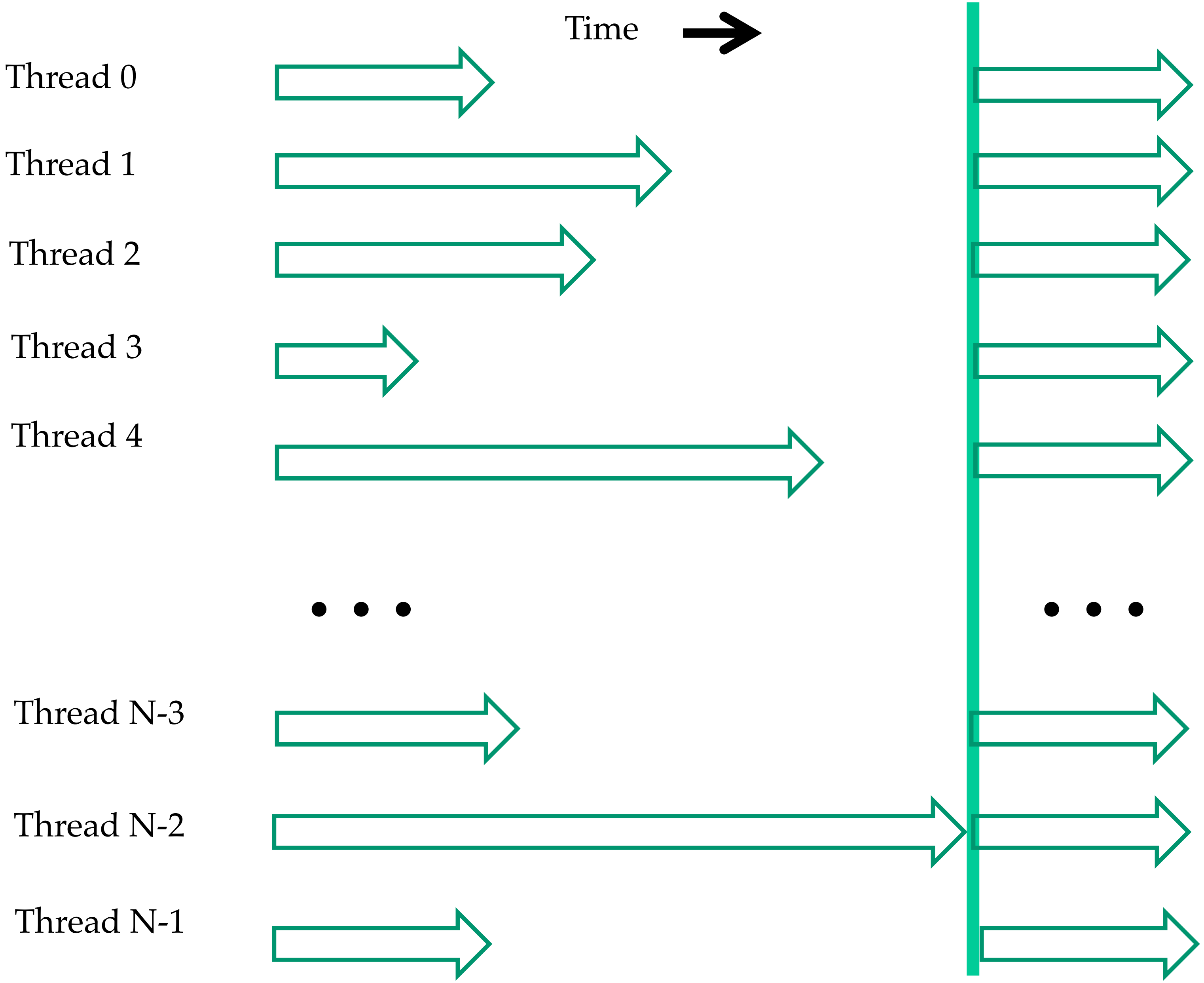
```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
  __shared__ float subTileM[TILE_WIDTH][TILE_WIDTH];
  __shared__ float subTileN[TILE_WIDTH][TILE_WIDTH];

  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];

     for (int k = 0; k < TILE_WIDTH; ++k)
        Pvalue += subTileM[ty][k] * subTileN[k][tx];

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

load subtile
every thread
load two elements

Bulk Synchronous Steps Based on Barriers

- **Sometimes we need all threads to catch up to a certain point in our code before any thread proceeds.**

- 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 it**
  - **after last thread calls** the barrier function, **all threads continue** past the barrier.

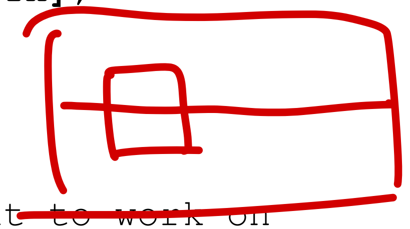

Time
Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread N-3
Thread N-2
Thread N-1

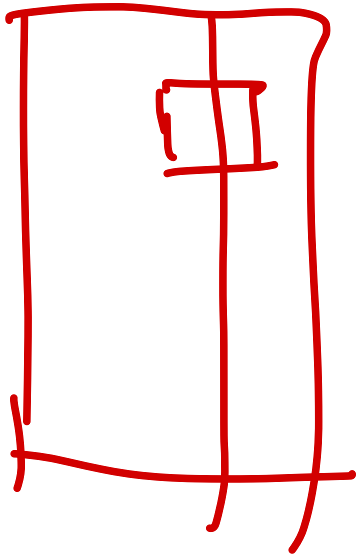
Barrier Synchronization

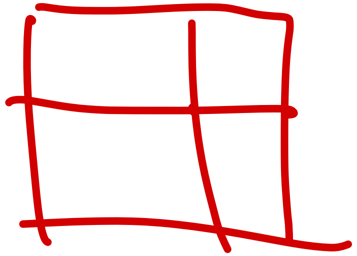
- An API function call in CUDA `__syncthreads()`
- All threads in the same block must reach the `__syncthreads()` before any can move on
- Can be used to coordinate tiled algorithms
  - To ensure that all elements of a tile are loaded
  - To ensure that certain computation on elements is complete

Tiled Matrix Multiplication Kernel

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

```
__global__
void MatrixMulKernel(float *d_M, float *d_N, float *d_P, int Width)
{
   // Calculate the row index of d_P 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 d_P
      for (int k = 0; k < Width; ++k)
          Pvalue += d_M[Row*Width+k] * d_N[k*Width+Col];
      d_P[Row*Width+Col] = Pvalue;
   }
}
```

Use of Large Tiles Shifts Bottleneck

- Recall our example GPU: **1,000 GFLOP/s**, **150 GB/s mem BW**
- **16x16 tiles** reuse each operand 16 times
  - **reduce global** memory **accesses by** a factor of **16**
  - **150GB/s** bandwidth supports (150/4)*16 = **600 GFLOPS**!
- **32x32 tiles** reuse each operand for 32 times
  - **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!**

SM constraints also play a factor

- 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.

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.

shared memory 大小的限制 ⇒ 只能 partial reuse

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.dev_prop.maxThreadsPerBlock`
  - `dev_prop.sharedMemoryPerBlock`
  - …

ANY MORE QUESTIONS? READ CHAPTER 4!