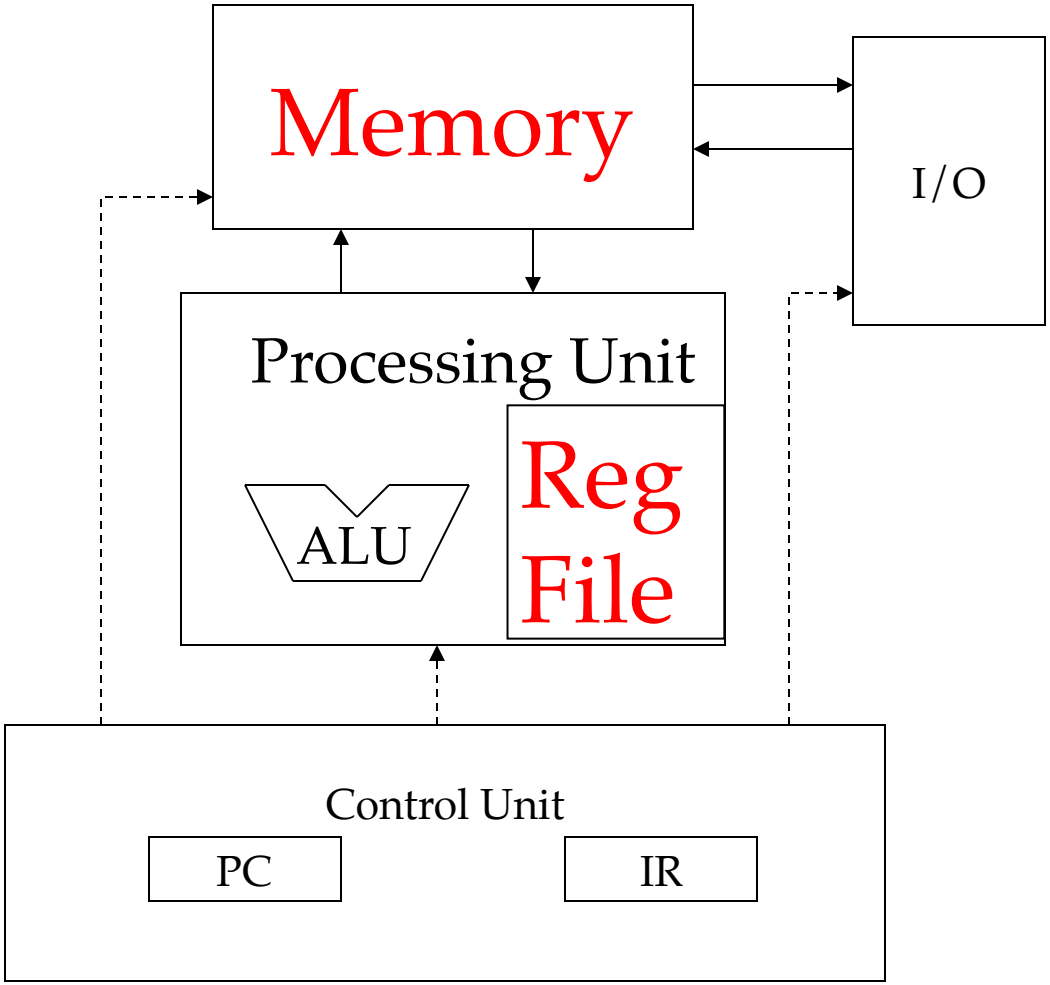
CS/EE 217
GPU Architecture and Parallel Programming

Lectures 4 and 5:
Memory Model and Locality

Objective

- To learn to efficiently use the important levels of the CUDA memory hierarchy
  - Registers, shared memory, global memory
  - Tiled algorithms and barrier synchronization

The Von-Neumann Model



Going back to the program

- Every instruction needs to be fetched from memory, decoded, then executed.
  - The decode stage typically accesses register file
- Instructions come in three flavors: Operate, Data transfer, and Program Control Flow.
- An example instruction cycle is the following:

Fetch | Decode | Execute | Memory

Operate Instructions

- Example of an operate instruction:

  ADD R1, R2, R3

- Instruction cycle for an operate instruction:

  Fetch | Decode | Execute | Memory

Memory Access Instructions

- Examples of memory access instruction:

  LDR R1, R2, #2

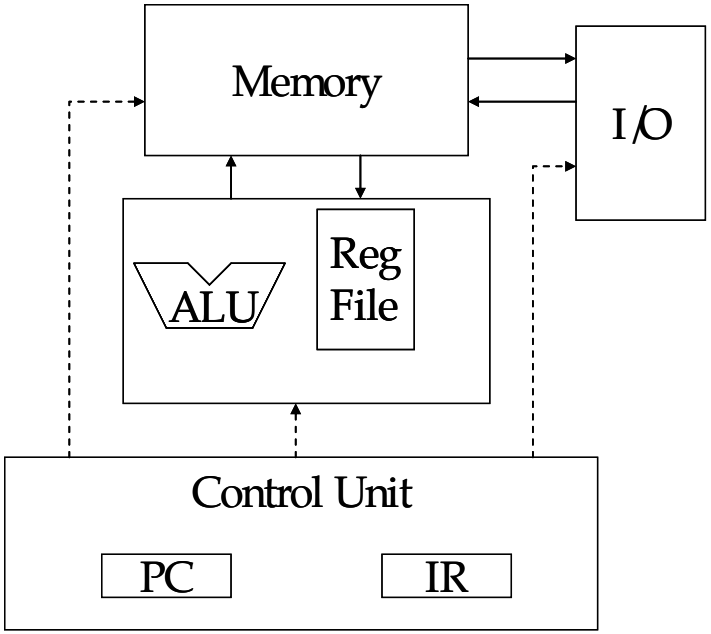
  STR R1, R2, #2

- Instruction cycle for an operate instruction:

  Fetch | Decode | Execute | Memory

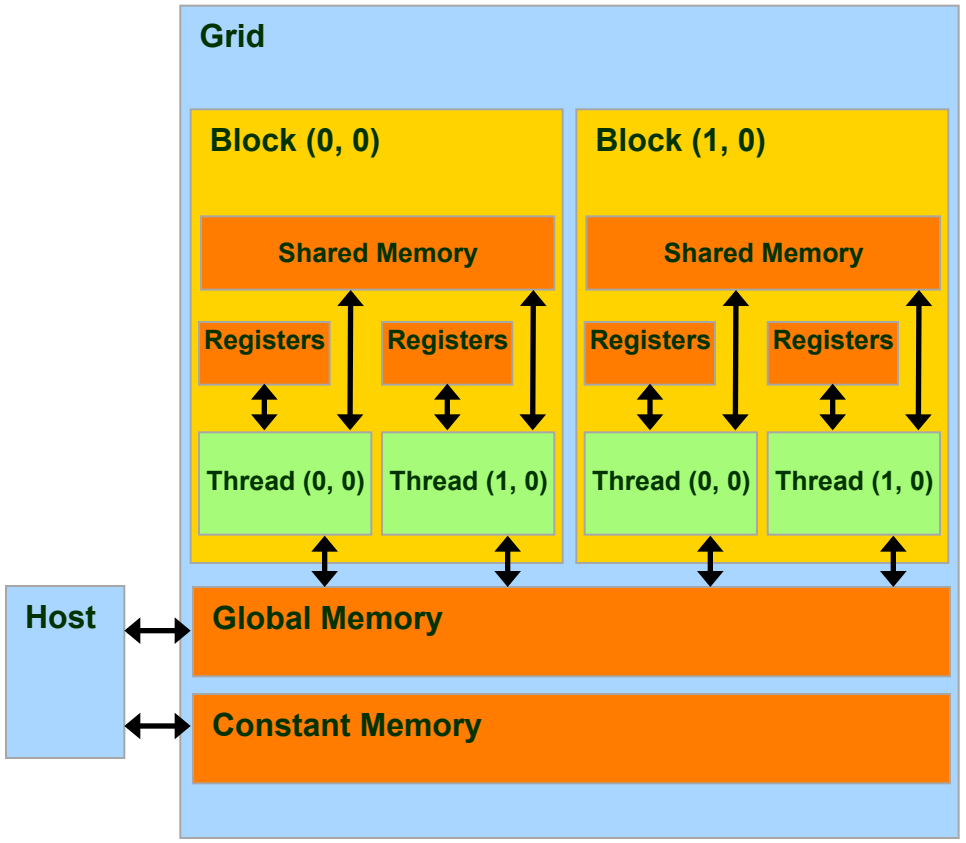
Registers vs Memory

- Registers are “free”
  - No additional memory access instruction
  - Very fast to use, however, there are very few of them
- Memory is expensive (slow), but very large



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)**



Shared Memory in CUDA

- A special type of memory whose contents are explicitly declared and used in the source code
  - Located in the processor
  - Accessed at much higher speed (in both latency and throughput)
  - Still accessed by memory access instructions
  - Commonly referred to as scratchpad memory in computer architecture

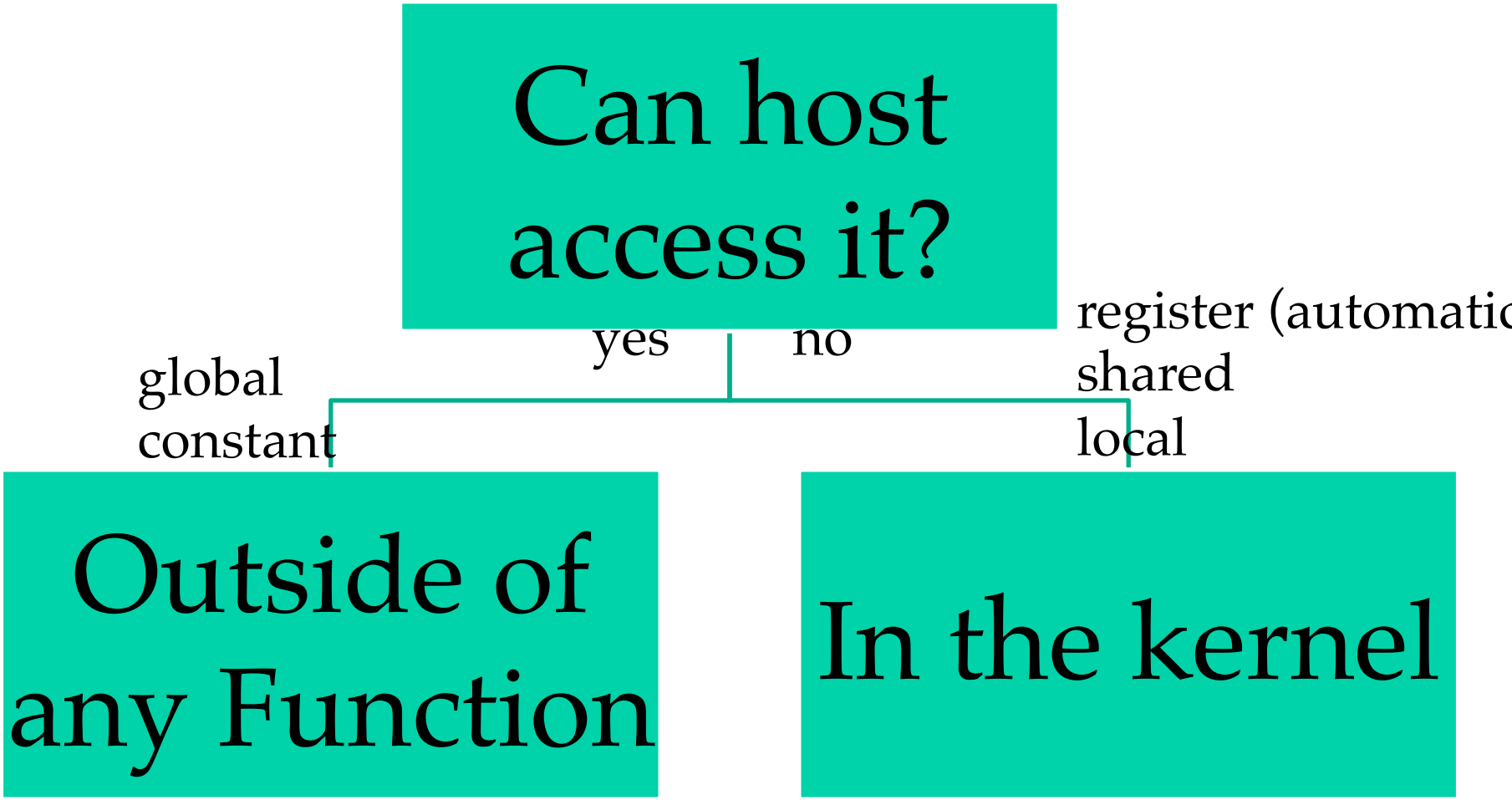
CUDA Variable Type Qualifiers

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

Where to Declare Variables?



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

```
1.   __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
2.   __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];
```

A Common Programming Strategy

- Global memory resides in device memory (DRAM) - slow access
- So, a profitable way of performing computation on the device is to tile input data to take advantage of fast shared memory:
  - Partition data into subsets that fit into 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 multi-pass over any data element
    - Copying results from shared memory to global memory

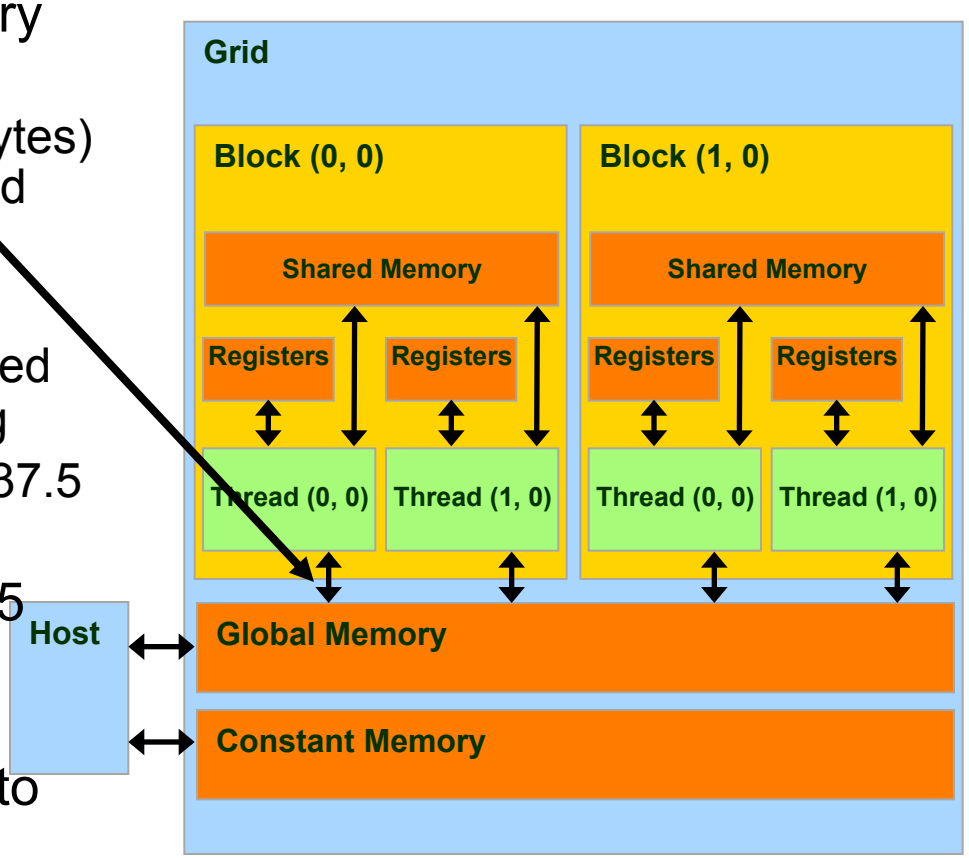
Matrix-Matrix Multiplication using Shared Memory

Base Matrix Multiplication Kernel

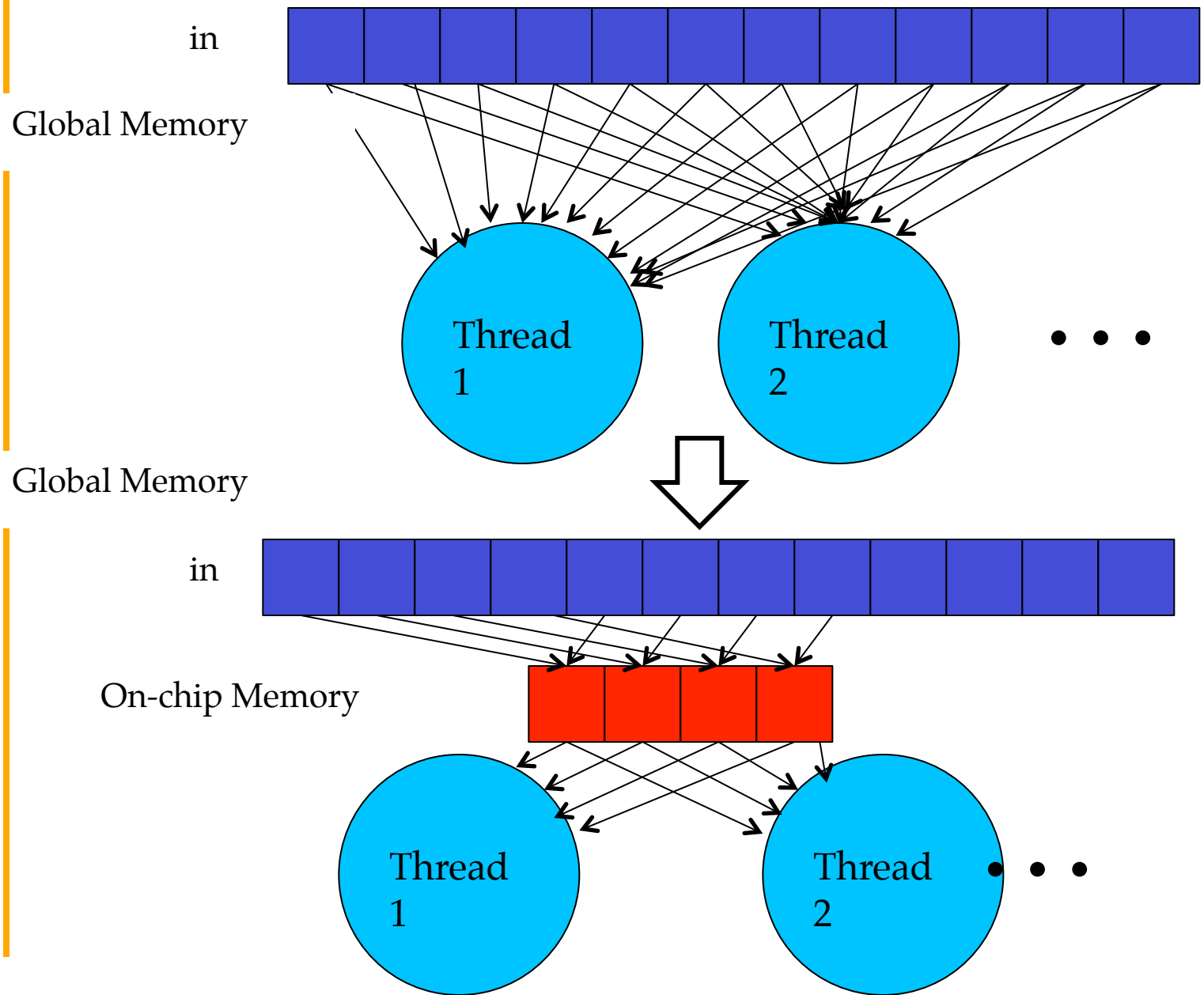
```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width)
{
// Calculate the row index of the Pd element and M
int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
// Calculate the column idenx of Pd and N
int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

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 Fermi?

- All threads access global memory for their input matrix elements
  - Two memory accesses (8 bytes) per floating point multiply-add
  - 4B/s of memory bandwidth/ FLOPS
  - 4*1,000 = 4,000 GB/s required to achieve peak FLOP rating
  - 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 1,000 GFLOPS



Shared Memory Blocking Basic Idea



Basic Concept of Blocking/Tiling

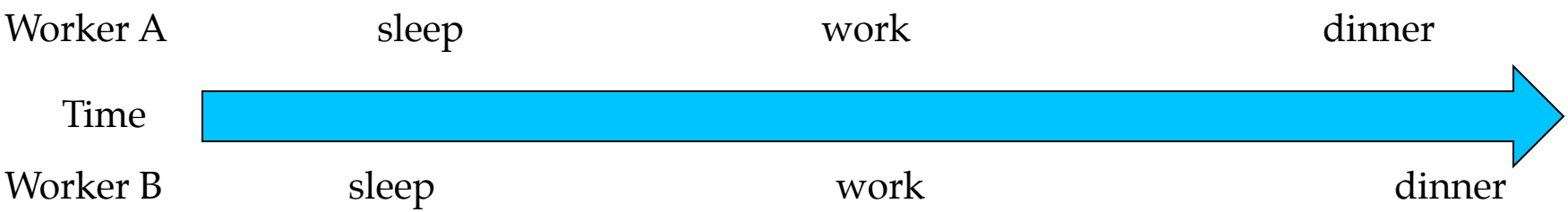
- In a congested traffic system, significant reduction of vehicles can greatly improve the delay seen by all vehicles
  - Carpooling for commuters
  - Blocking/Tiling for global memory accesses
    - drivers = threads,
    - cars = data

Some computations are more challenging to block/tile than others.

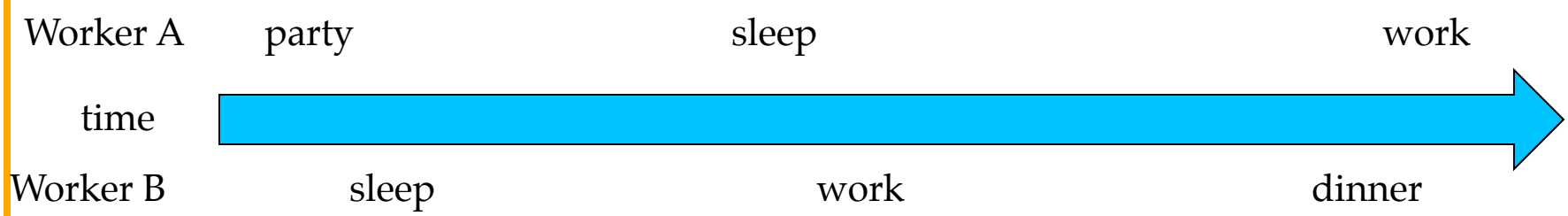
- Some carpools may be easier than others
  - More efficient if neighbors are also classmates or co-workers
  - Some vehicles may be more suitable for carpooling
- Similar variations exist in blocking/tiling

Carpools need synchronization.

- Good – when people have similar schedule

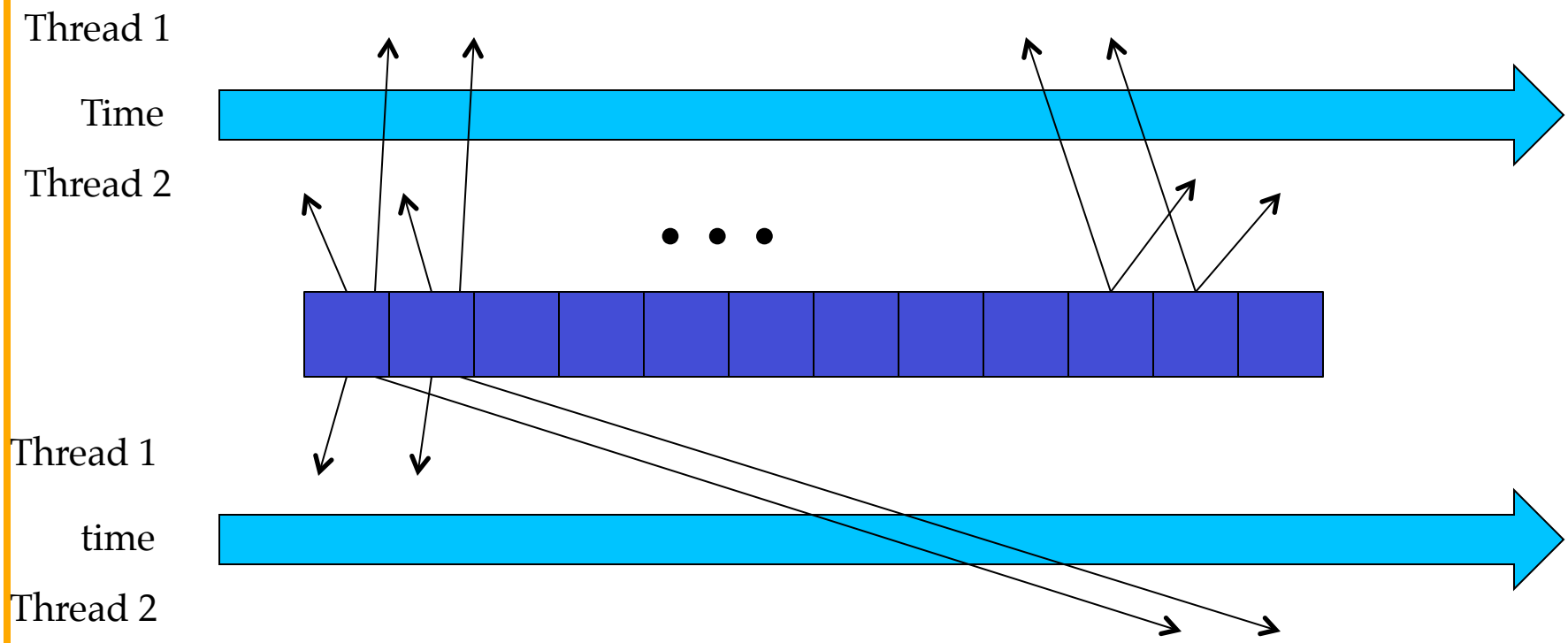


- Bad – when people have very different schedule



Same with Blocking/Tiling

- Good – when threads have similar access timing



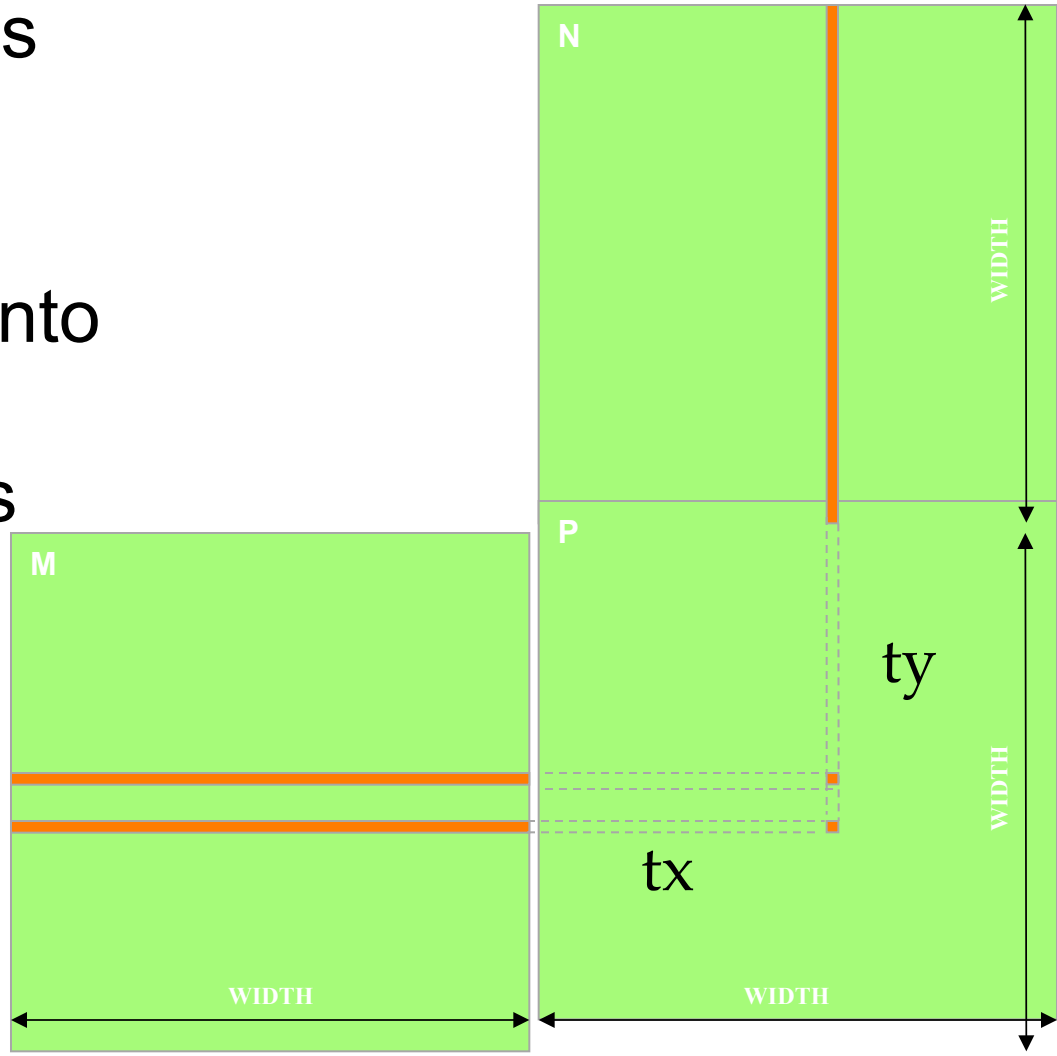
- Bad – when threads have very different timing

Outline of Technique

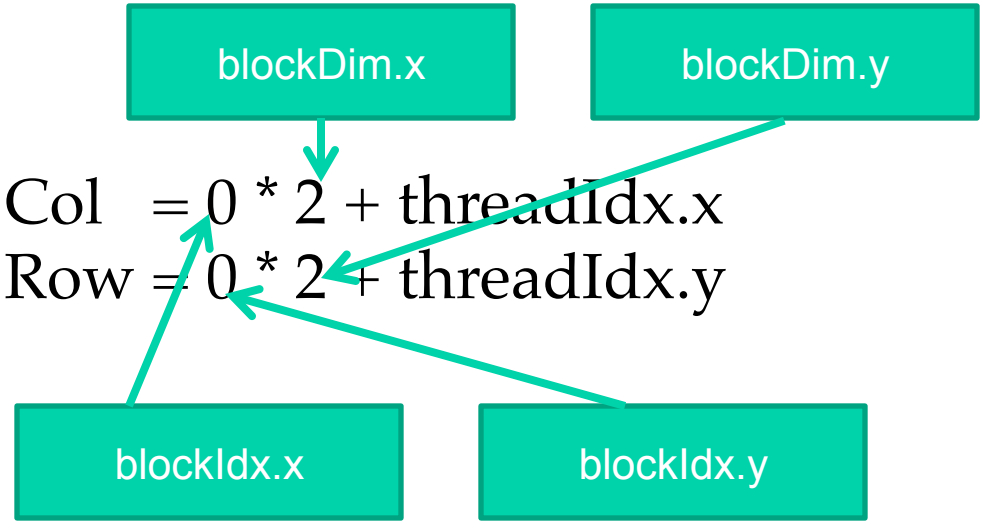
- Identify a block/tile of global memory content that are accessed by multiple threads
- Load the block/tile from global memory into on-chip memory
- Have the multiple threads to access their data from the on-chip memory
- Move on to the next block/tile

Idea: Use Shared Memory to reuse global memory data

- Each input element is read by WIDTH threads.
- Load each element into Shared Memory and have several threads use the local version reduce the memory bandwidth
  - Tiled algorithms



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

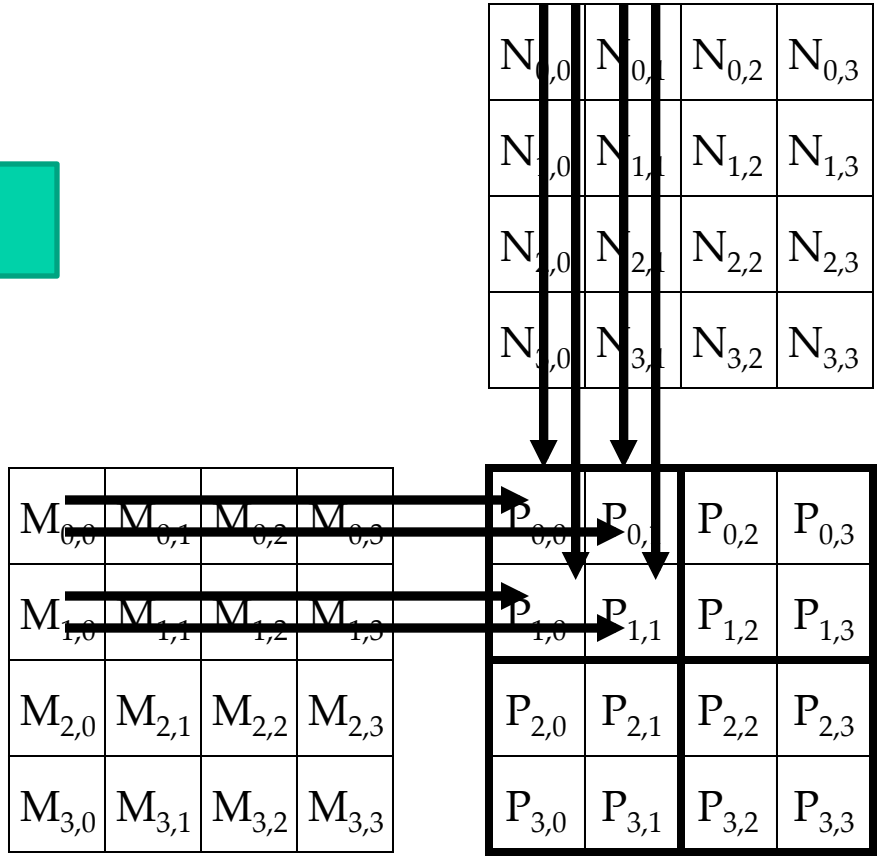


Col = 0

Col = 1

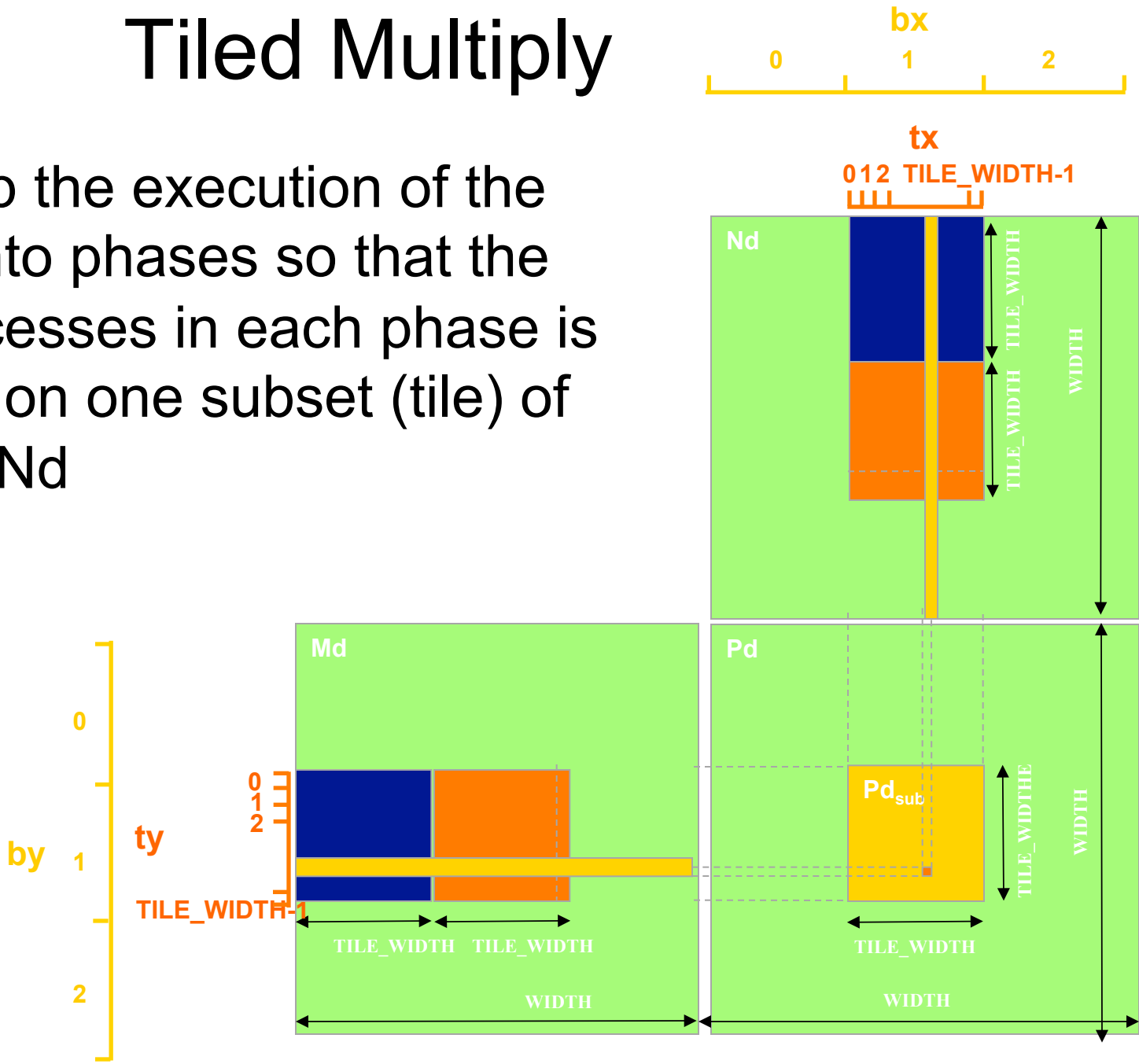
Row = 0

Row = 1



Tiled Multiply

- Break up the execution of the kernel into phases so that the data accesses in each phase is focused on one subset (tile) of Md and Nd

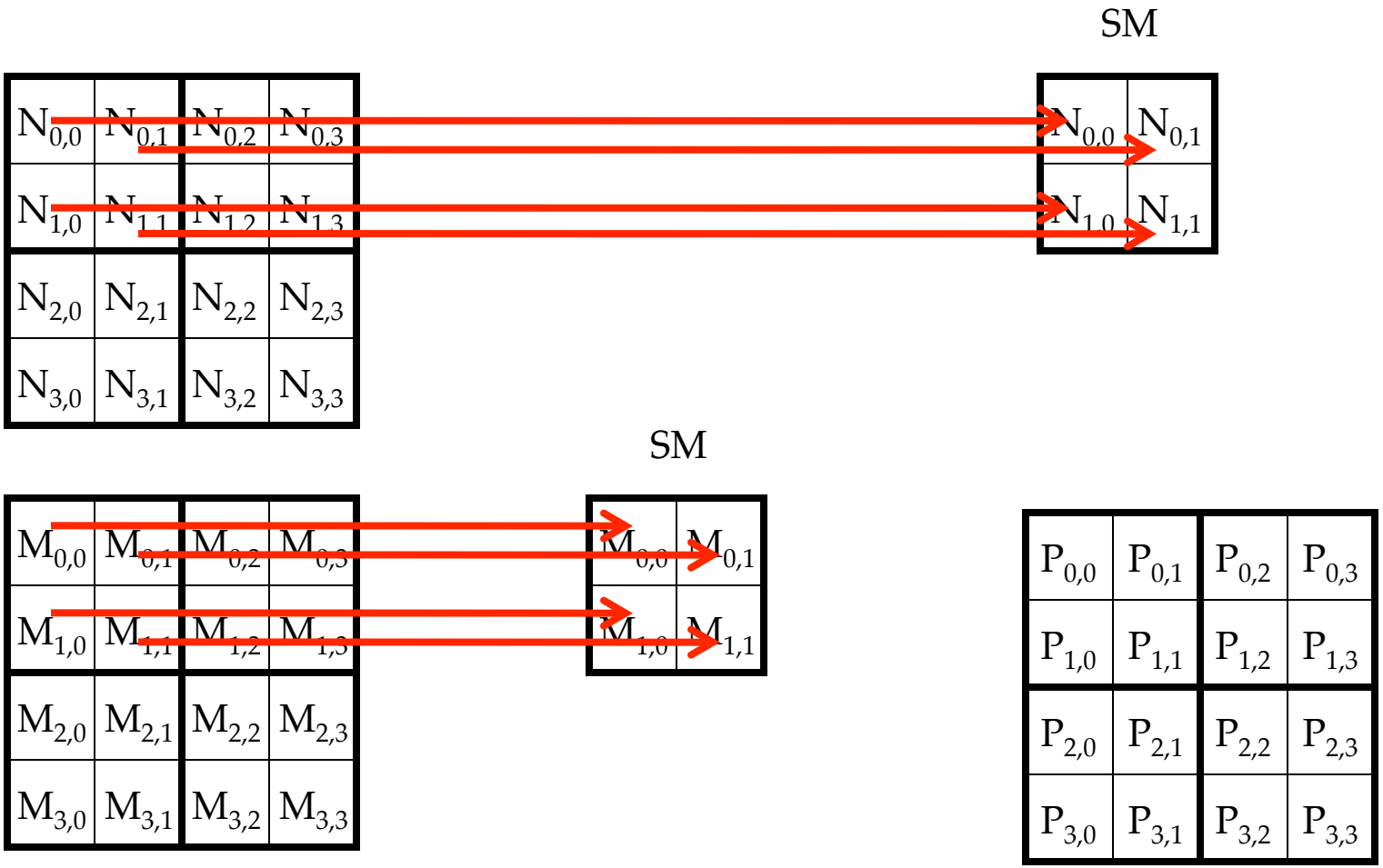


Loading a Tile

- All threads in a block participate
  - Each thread loads one Md element and one Nd element in based tiled code

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

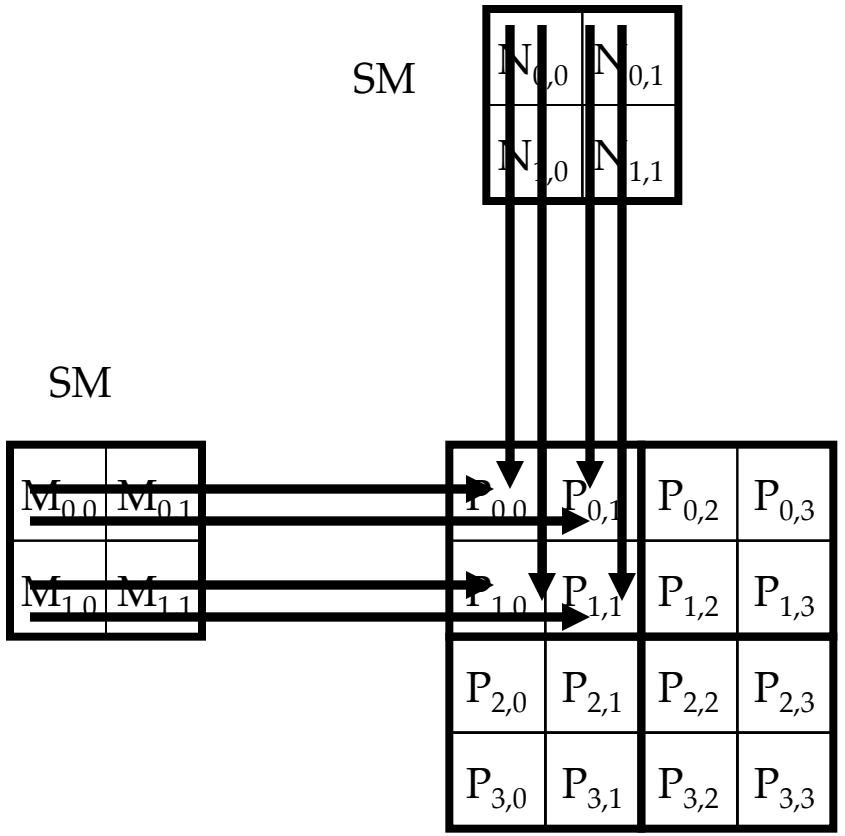
Work for Block (0,0)



Work for Block (0,0)

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

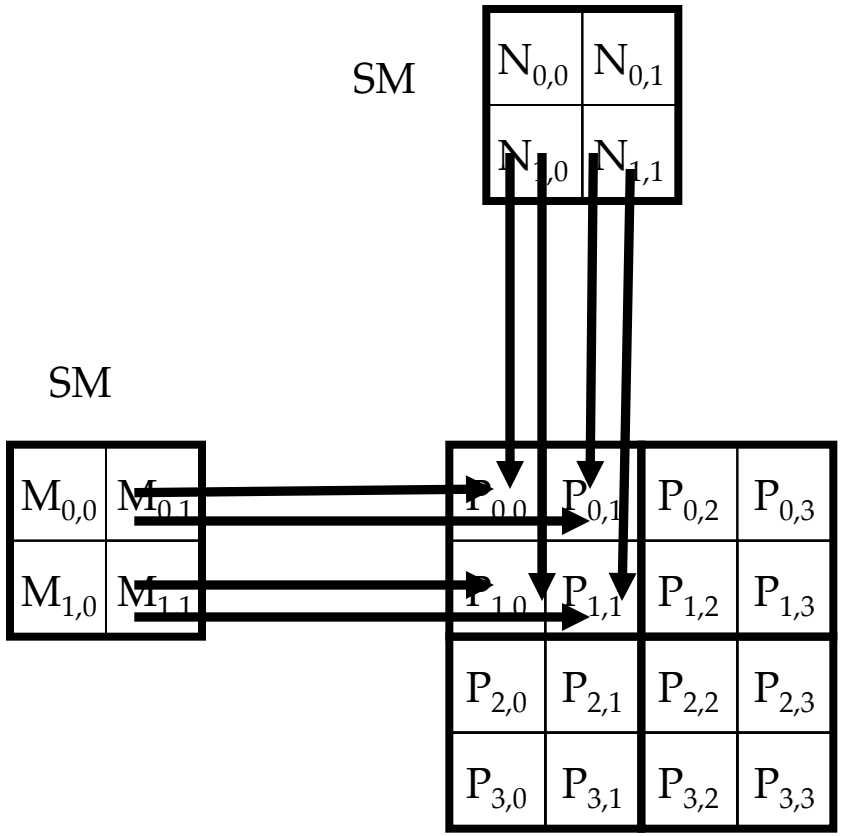
$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$



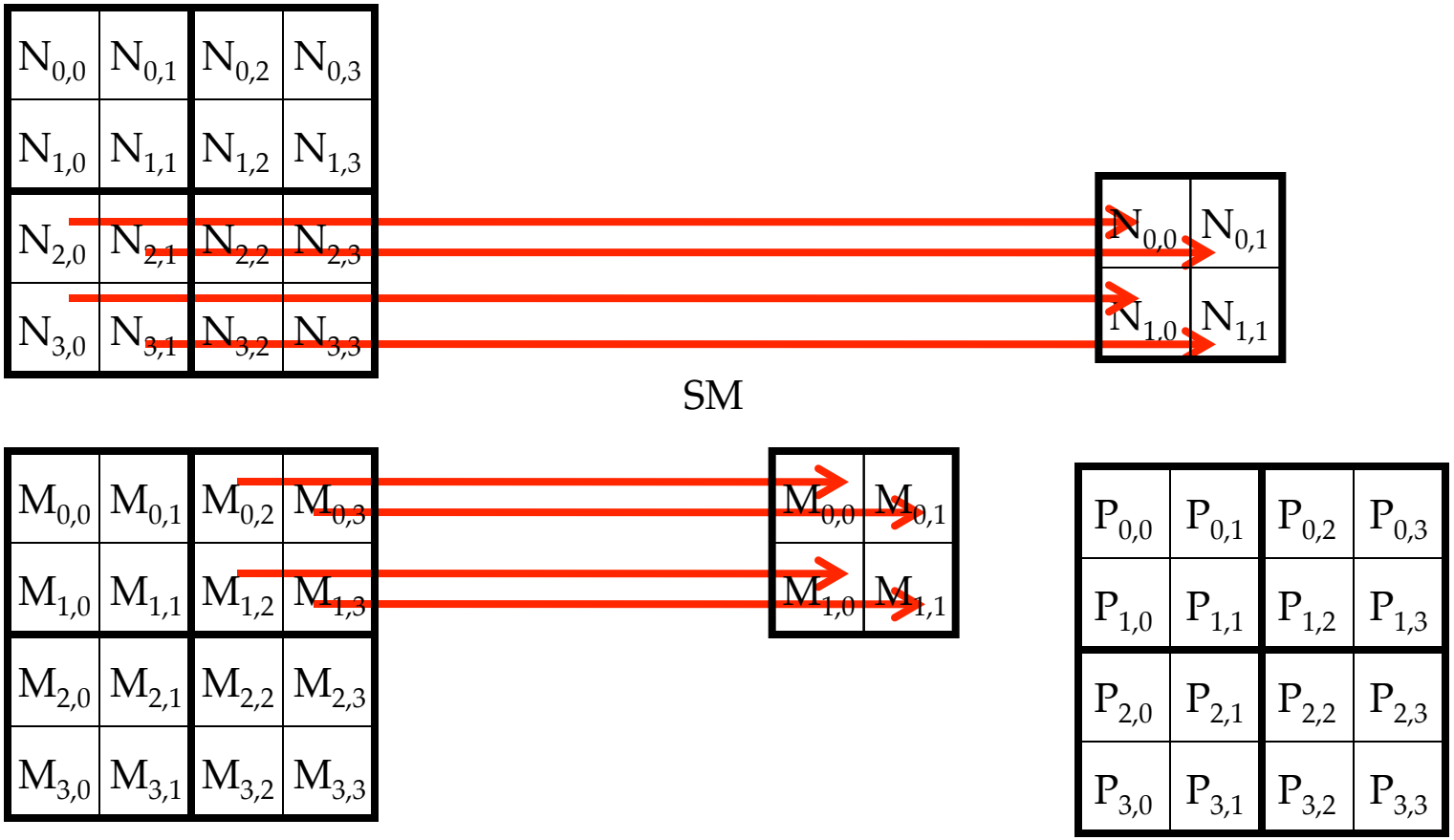
Work for Block (0,0)

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

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



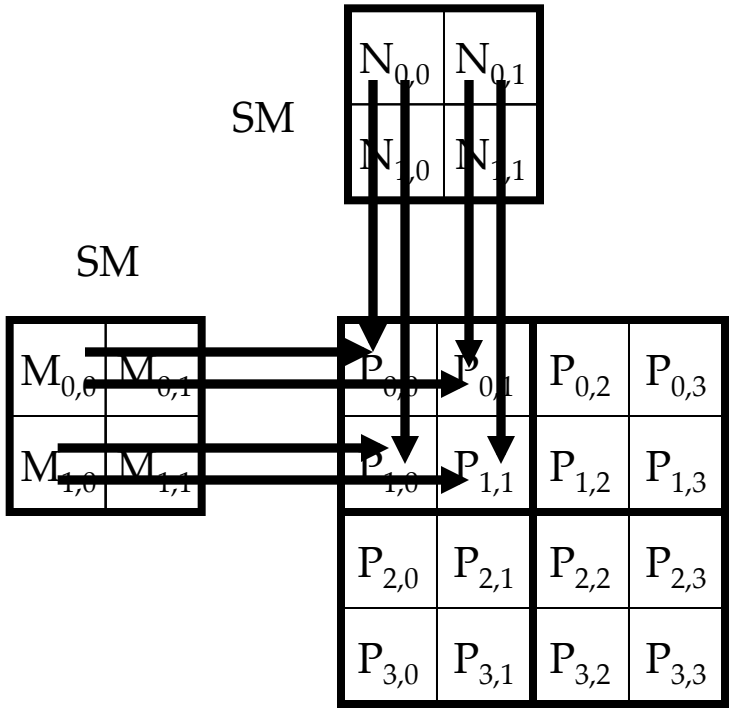
Work for Block (0,0)



Work for Block (0,0)

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

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



Barrier Synchronization

- An API function call in CUDA
  - __syncthreads()
- All threads in the same block must reach the __synctrheads() before any can move on
- Best used to coordinate tiled algorithms
  - To ensure that all elements of a tile are loaded
  - To ensure that all elements of a tile are consumed

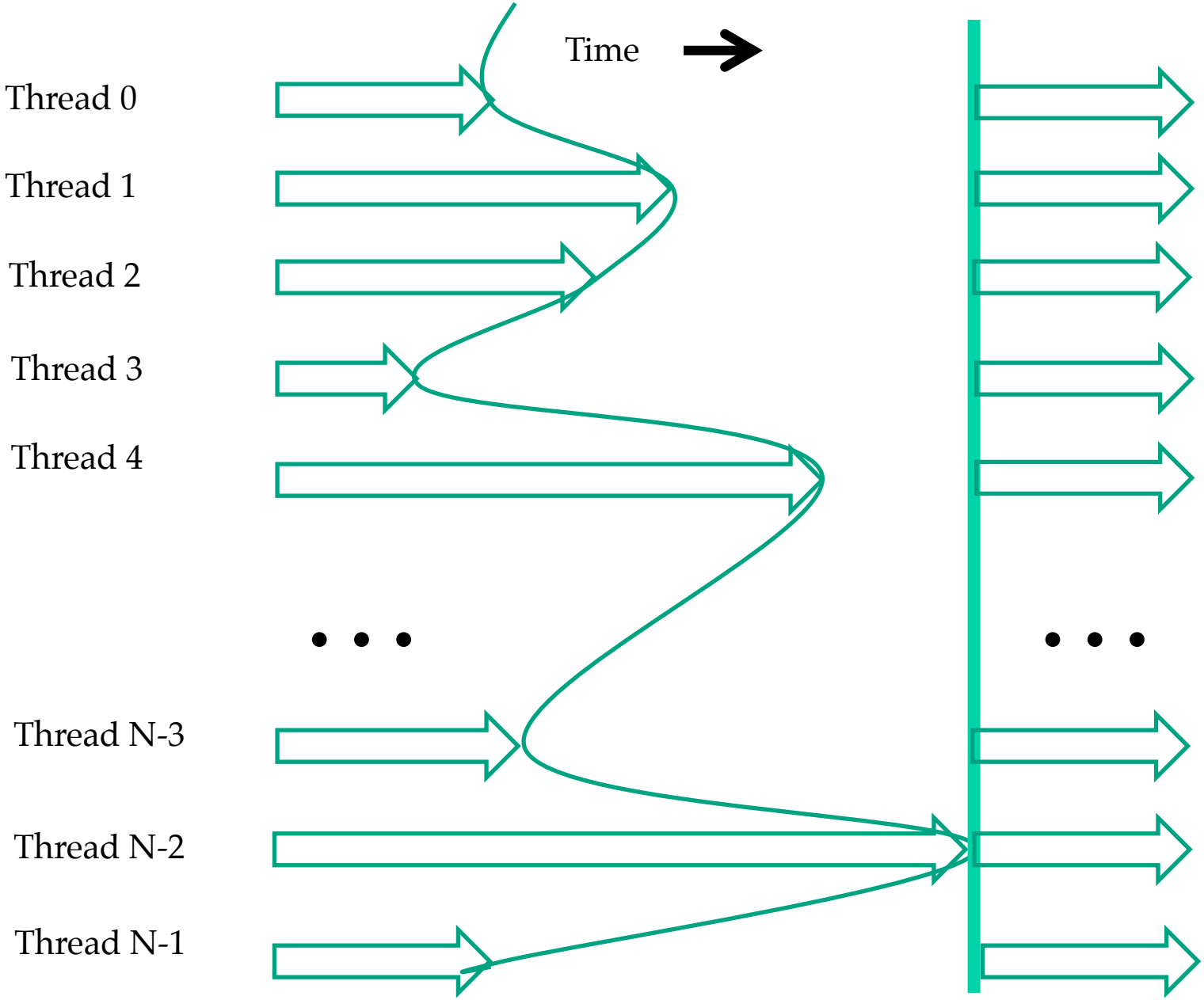


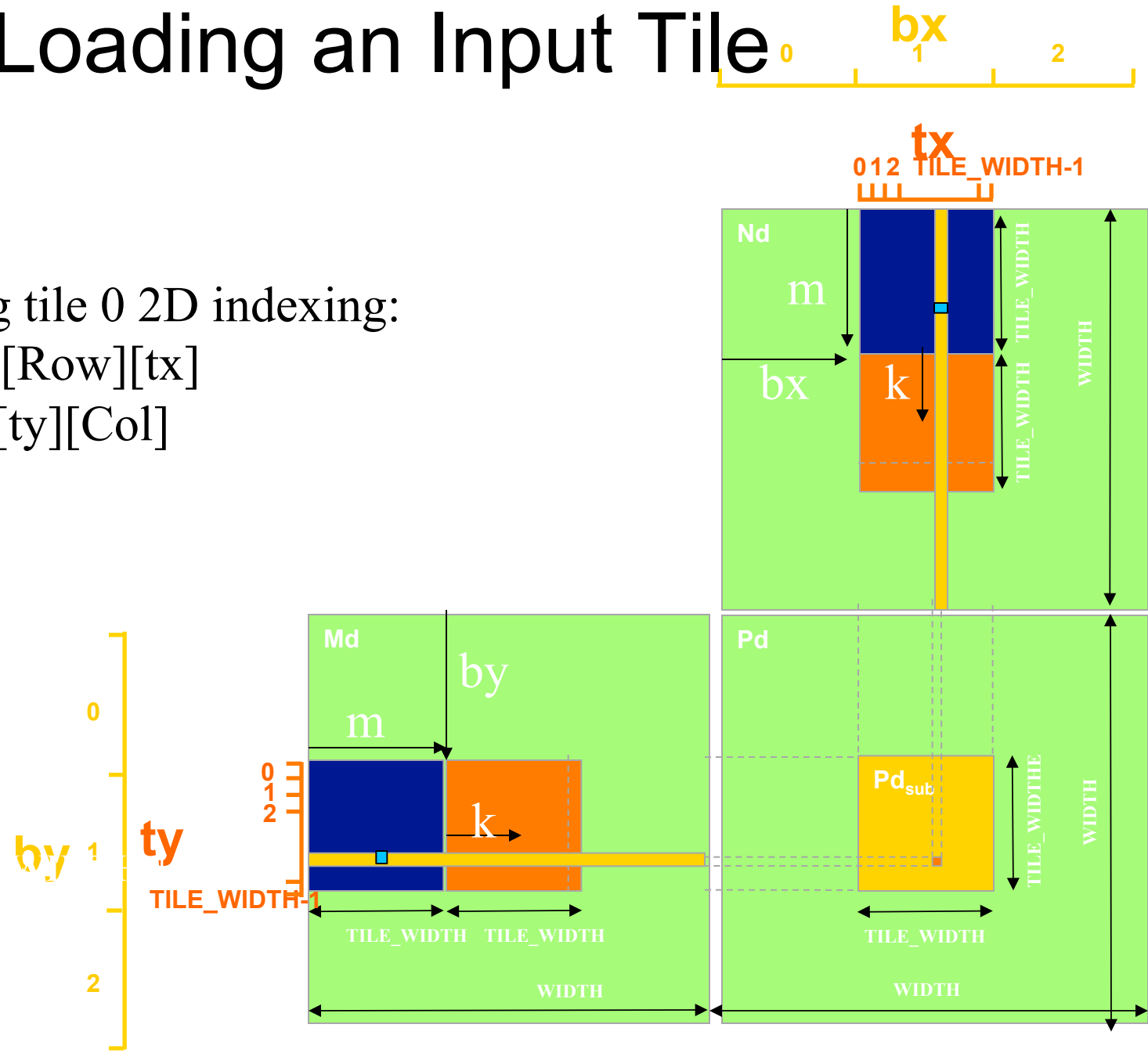
Figure 4.11 An example execution timing of barrier synchronization.

Loading an Input Tile

Accessing tile 0 2D indexing:

M[Row][tx]

N[ty][Col]

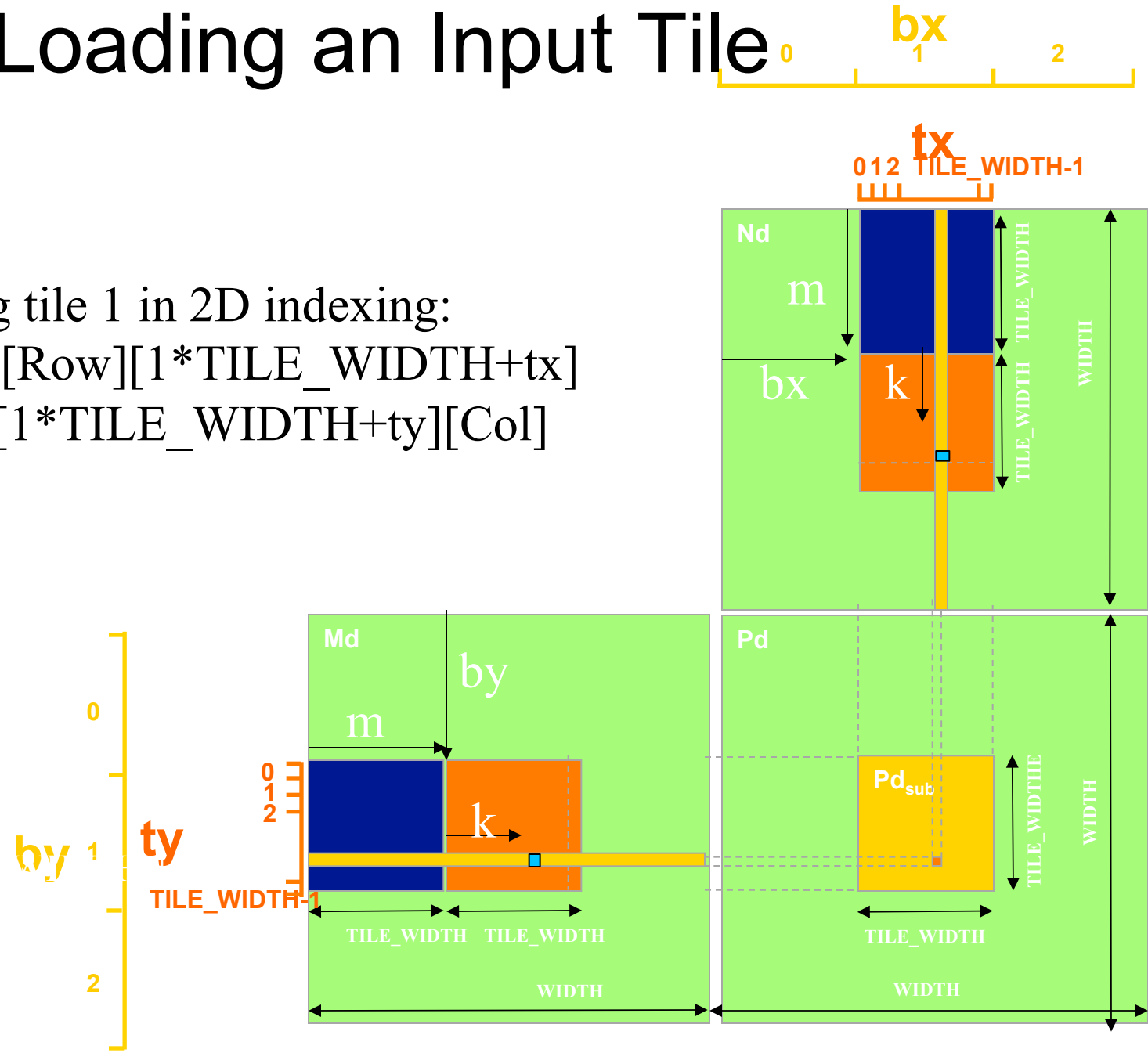


Loading an Input Tile

Accessing tile 1 in 2D indexing:

M[Row][1*TILE_WIDTH+tx]

N[1*TILE_WIDTH+ty][Col]

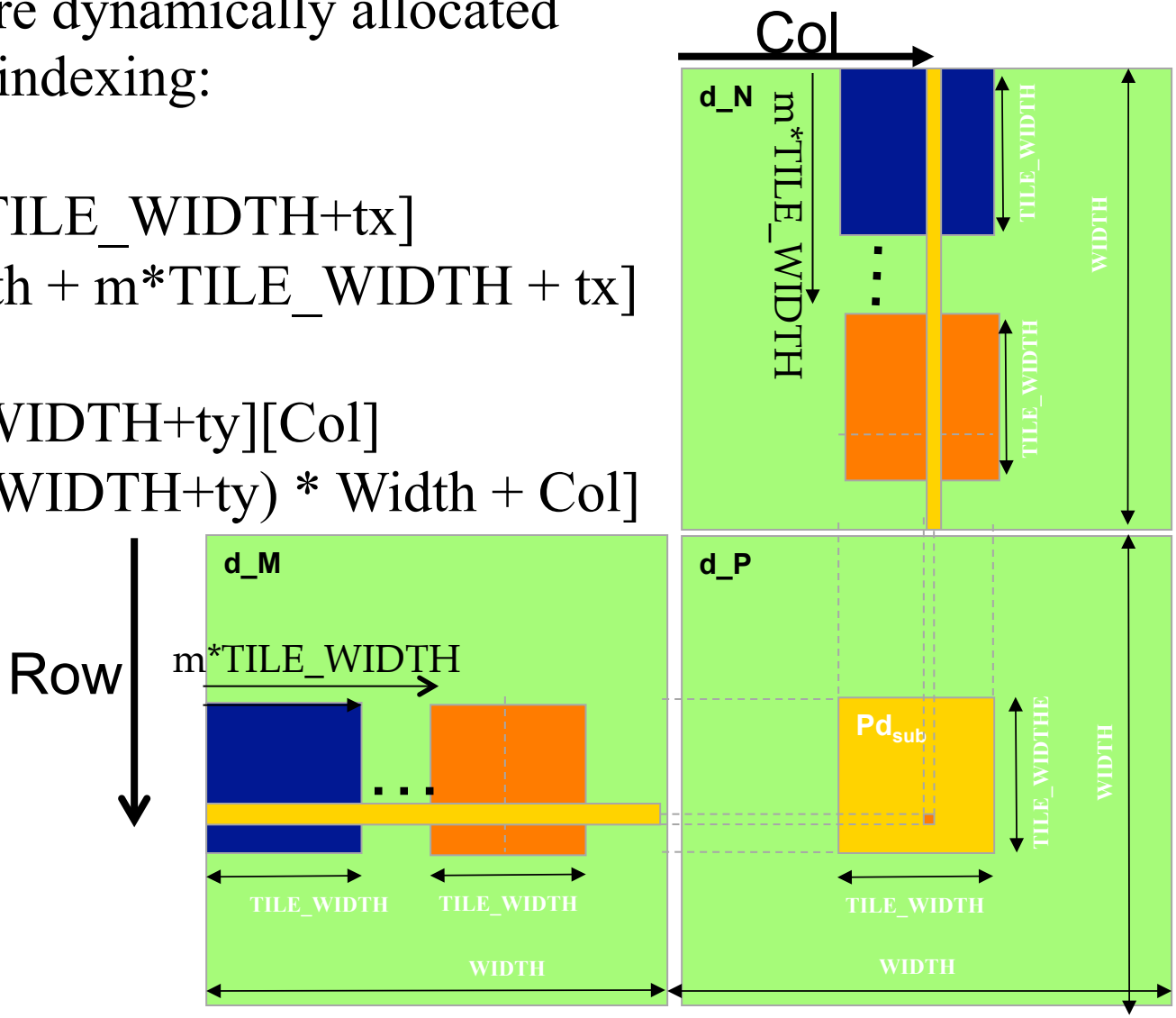


Loading Input Tile m

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

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

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



Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width)
{
1.  __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
2.  __shared__ float ds_N[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 Pd 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 Md and Nd tiles required to compute the Pd element
8.   for (int m = 0; m < Width/TILE_WIDTH; ++m) {
// Coolaborative loading of Md and Nd tiles into shared memory
9.    ds_M[ty][tx] = d_M[Row*Width + m*TILE_WIDTH+tx];
10.   ds_N[ty][tx] = d_N[Col+(m*TILE_WIDTH+ty)*Width];
11.   __syncthreads();
12.  for (int k = 0; k < TILE_WIDTH; ++k)
13.     Pvalue += ds_M[ty][k] * ds_N[k][tx];
14.   __synchthreads();
15.}
16.  d_P[Row*Width+Col] = Pvalue;
}
```

Compare with the Base Kernel

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

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;
}
```

First-order 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, each block performs 2*256 = 512 float loads from global memory for 256 * (2*16) = 8,192 mul/add operations.

- For 32, each block performs 2*1024 = 2048 float loads from global memory for 1024 * (2*32) = 65,536 mul/add operations

Shared Memory and Threading

- Each SM in Fermi has 16KB or 48KB shared memory*
  - SM size is implementation dependent!
  - For TILE_WIDTH = 16, each thread block uses 2*256*4B = 2KB of shared memory.
  - Can potentially have up to 8 Thread Blocks actively executing
    - This allows up to 8*512 = 4,096 pending loads. (2 per thread, 256 threads per block)
  - The next TILE_WIDTH 32 would lead to 2*32*32*4B= 8KB shared memory usage per thread block, allowing 2 or 6 thread blocks active at the same time
- Using 16x16 tiling, we reduce the accesses to the global memory by a factor of 16
  - The 150GB/s bandwidth can now support (150/4)*16 = 600GFLOPS!

*Configurable vs L1, total 64KB

Boundary conditions

- What to do if the matrix size is not a multiple of width?
  - Tricky problem, lets work through an example
- Too many boundary checks can cause control divergence and overhead
- Something that you have to work through for lab 2
  - I'll start a piazza discussion on the topic

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
  - …

Summary- Typical Structure of a CUDA Program

- Global variables declaration
  - __host__
  - __device__... __global__, __constant__, __texture__
- Function prototypes
  - __global__ void kernelOne(…)
  - float handyFunction(…)
- Main ()
  - allocate memory space on the device – cudaMalloc(&d_GlblVarPtr, bytes)
  - transfer data from host to device – cudaMemCpy(d_GlblVarPtr, h_Gl…)
  - execution configuration setup
  - kernel call – kernelOne<<<execution configuration>>>(args…);
  - transfer results from device to host – cudaMemCpy(h_GlblVarPtr,…)
  - optional: compare against golden (host computed) solution
- Kernel – void kernelOne(type args,…)
  - variables declaration - auto, __shared__
    - automatic variables transparently assigned to registers
  - syncthreads()…
- Other functions
  - float handyFunction(int inVar…);

repeat as needed

ANY MORE QUESTIONS?
READ CHAPTER 5!