PASI Summer School

Advanced Algorithmic Techniques for GPUs

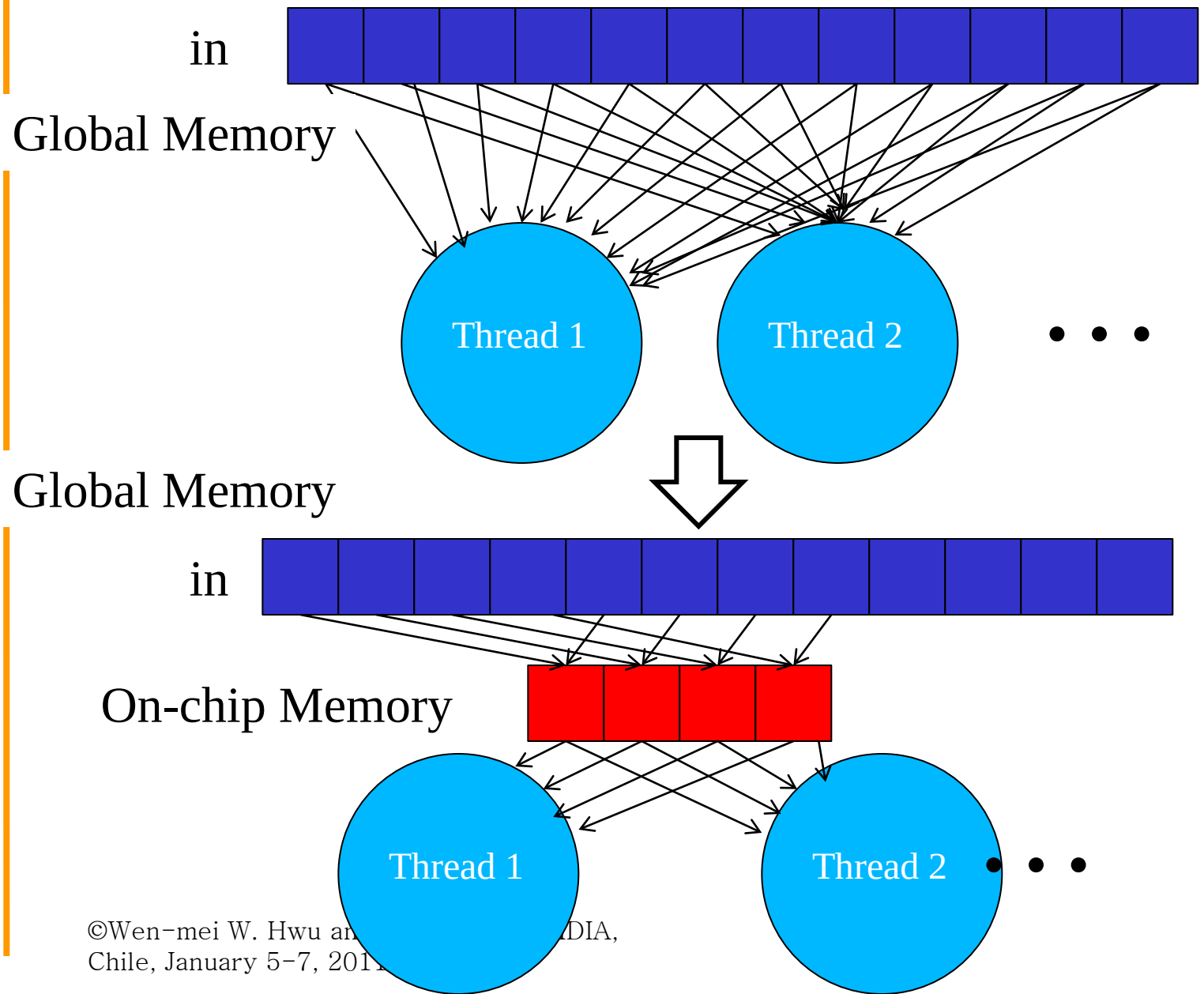
Lecture 3: Blocking/Tiling for Locality

Objective

- Reuse each data accessed from the global memory multiple times
  - Across threads – shared memory blocking
  - Within a thread - register tiling

- Register tiling is also often used to re-use computation results for increased efficiency.

Shared Memory Blocking Basic Idea



Basic Concept of Blocking/Tiling

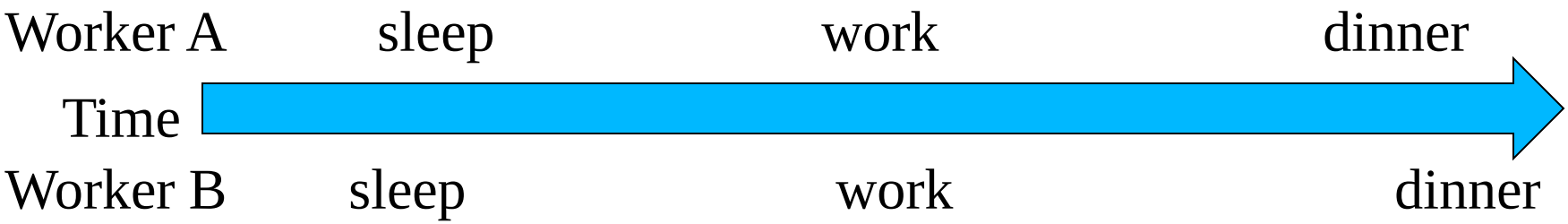
- In a congested traffic system, significant reduction of vehicles can greatly improve the delay seen by all vehicles
  - Carpooling for commuters
  - Blocking/Tiling for global memory accesses
    - drivers = threads,
    - cars = data

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

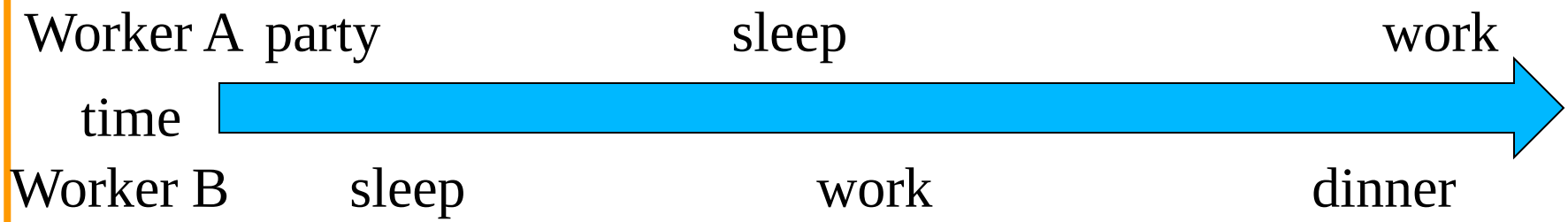
- Some carpools may be easier than others
  - More efficient if neighbors are also classmates or co-workers
  - Some vehicles may be more suitable for carpooling
- Similar variations exist in blocking/tiling

Carpools need synchronization.

- Good – when people have similar schedule

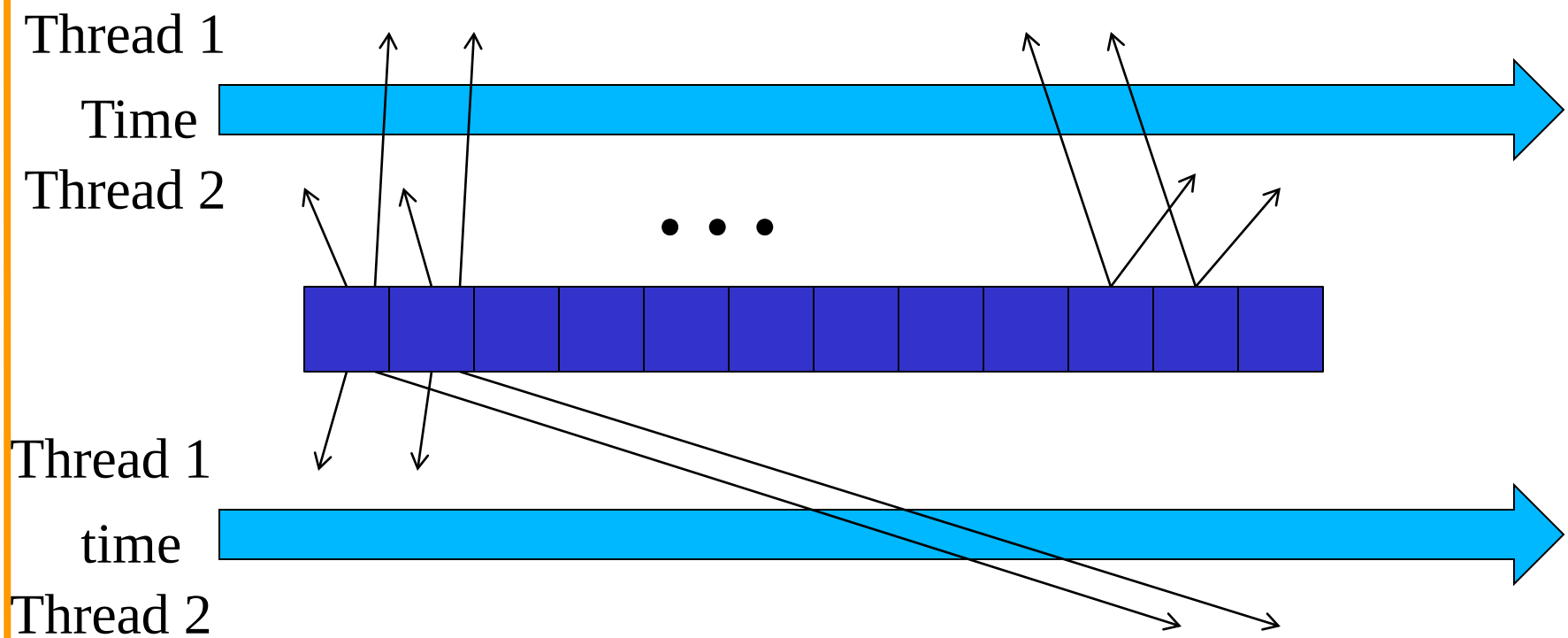


- Bad – when people have very different schedule



Same with Blocking/Tiling

- Good – when threads have similar access timing



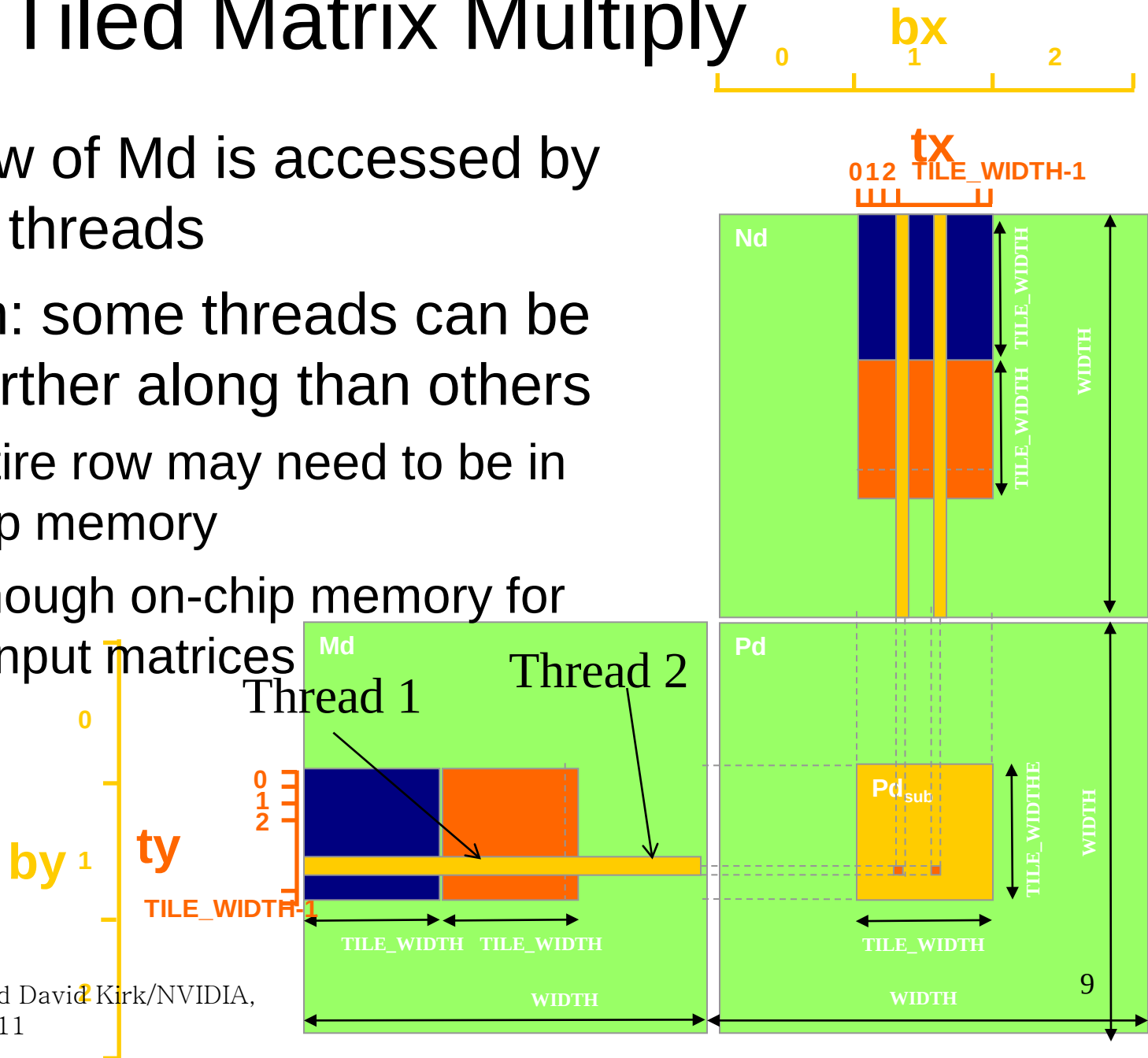
- Bad – when threads have very different timing

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

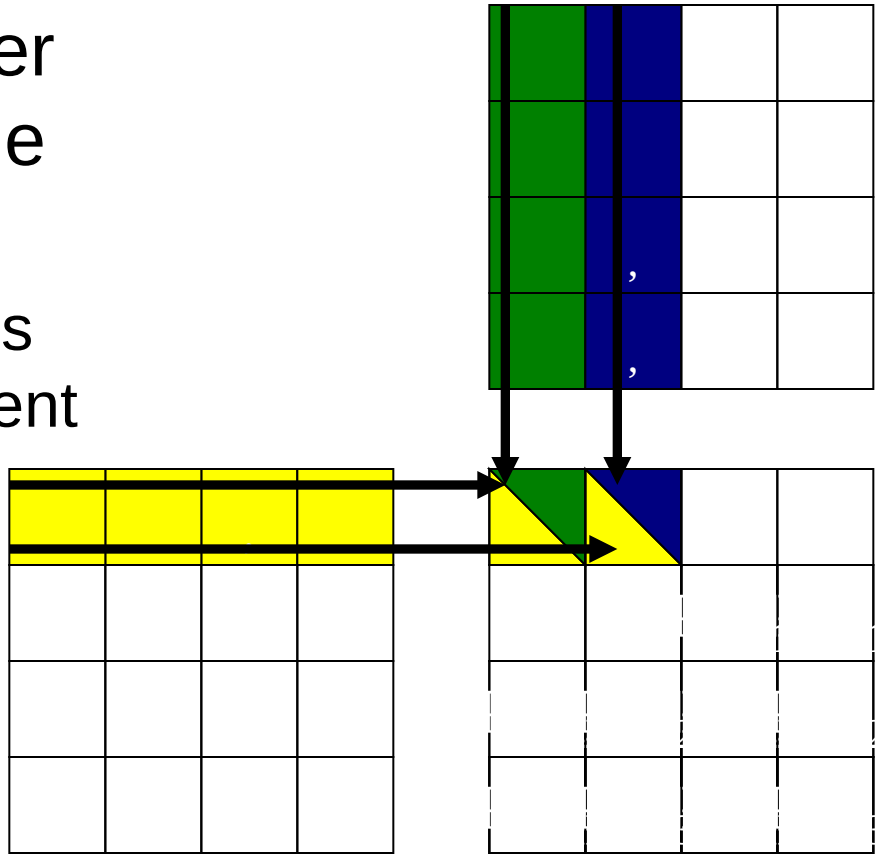
Tiled Matrix Multiply

- Each row of Md is accessed by multiple threads
- Problem: some threads can be much further along than others
  - An entire row may need to be in on-chip memory
  - Not enough on-chip memory for large input matrices



A Small Example

- Can we use two on-chip memory locations to reduce the number of M accesses by the two threads?
  - Not if the two threads can have very different timing!

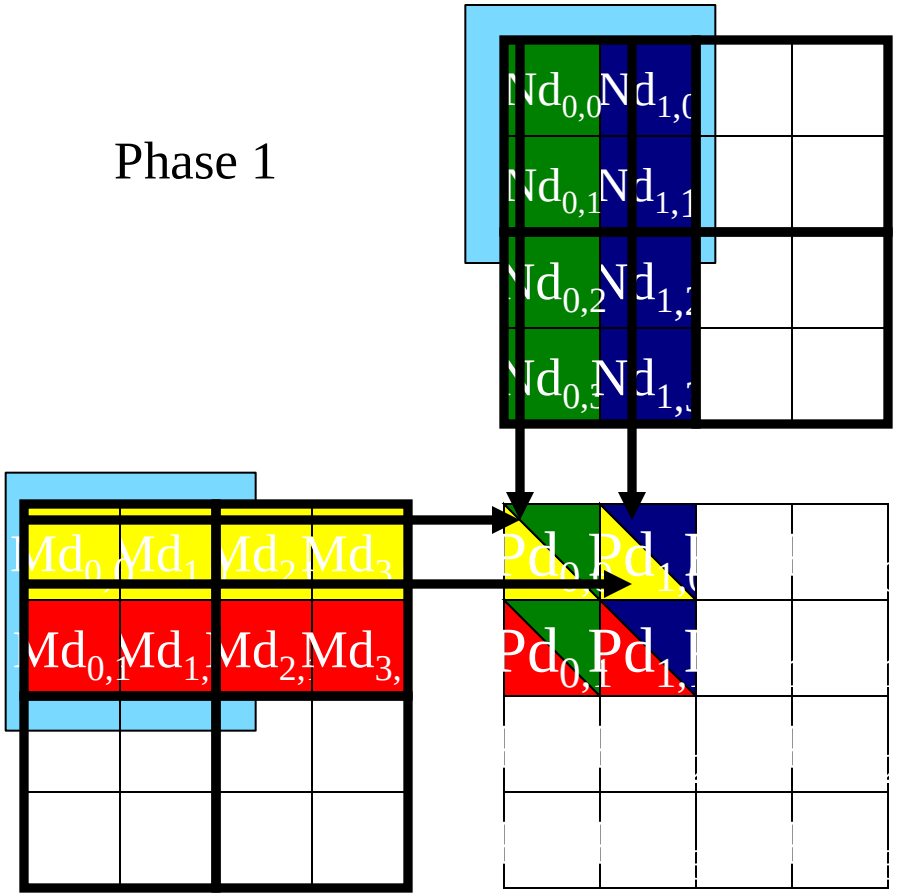


Every M and N Element is used exactly twice in generating a 2X2 tile of P

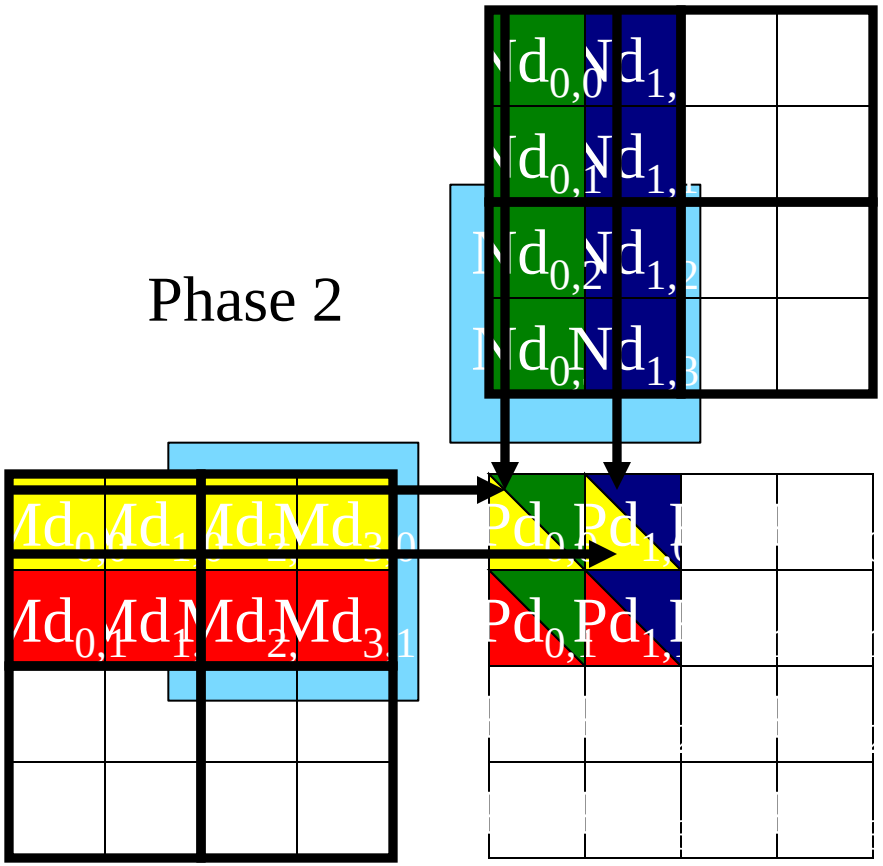
Access order

$P_{0,0}$ $thread_{0,0}$	$P_{1,0}$ $thread_{1,0}$	$P_{0,1}$ $thread_{0,1}$	$P_{1,1}$ $thread_{1,1}$
$M_{0,0} * N_{0,0}$	$M_{0,0} * N_{1,0}$	$M_{0,1} * N_{0,0}$	$M_{0,1} * N_{1,0}$
$M_{1,0} * N_{0,1}$	$M_{1,0} * N_{1,1}$	$M_{1,1} * N_{0,1}$	$M_{1,1} * N_{1,1}$
$M_{2,0} * N_{0,2}$	$M_{2,0} * N_{1,2}$	$M_{2,1} * N_{0,2}$	$M_{2,1} * N_{1,2}$
$M_{3,0} * N_{0,3}$	$M_{3,0} * N_{1,3}$	$M_{3,1} * N_{0,3}$	$M_{3,1} * N_{1,3}$

Breaking Md and Nd into Tiles



Breaking Md and Nd into Tiles (cont.)



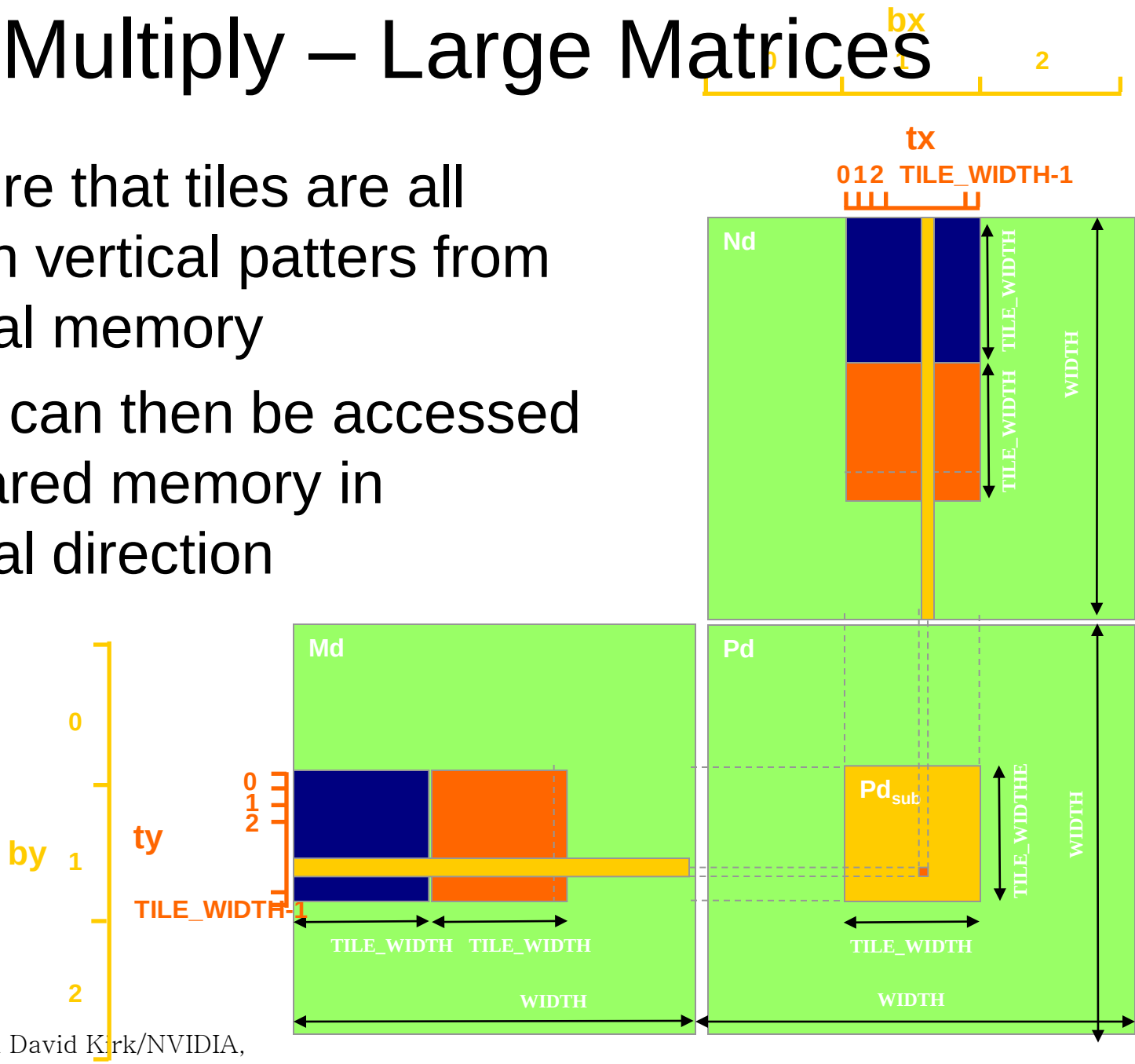
Each phase uses one tile from Md and one from Nd

	Phase 1			Phase 2		
$T_{0,0}$	**$Md_{0,0}$** ↓ $Mds_{0,0}$	**$Nd_{0,0}$** ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{1,0}$*$Nds_{0,1}$	**$Md_{2,0}$** ↓ $Mds_{0,0}$	**$Nd_{0,2}$** ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{1,0}$*$Nds_{0,1}$
$T_{1,0}$	**$Md_{1,0}$** ↓ $Mds_{1,0}$	**$Nd_{1,0}$** ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{0,0}$*$Nds_{1,0}$ + $Mds_{1,0}$*$Nds_{1,1}$	**$Md_{3,0}$** ↓ $Mds_{1,0}$	**$Nd_{1,2}$** ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{0,0}$*$Nds_{1,0}$ + $Mds_{1,0}$*$Nds_{1,1}$
$T_{0,1}$	**$Md_{0,1}$** ↓ $Mds_{0,1}$	**$Nd_{0,1}$** ↓ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{0,1}$	**$Md_{2,1}$** ↓ $Mds_{0,1}$	**$Nd_{0,3}$** ↓ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{0,1}$
$T_{1,1}$	**$Md_{1,1}$** ↓ $Mds_{1,1}$	**$Nd_{1,1}$** ↓ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1,1}$*$Nds_{1,1}$	**$Md_{3,1}$** ↓ $Mds_{1,1}$	**$Nd_{1,3}$** ↓ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1,1}$*$Nds_{1,1}$

time

Tiled Multiply – Large Matrices

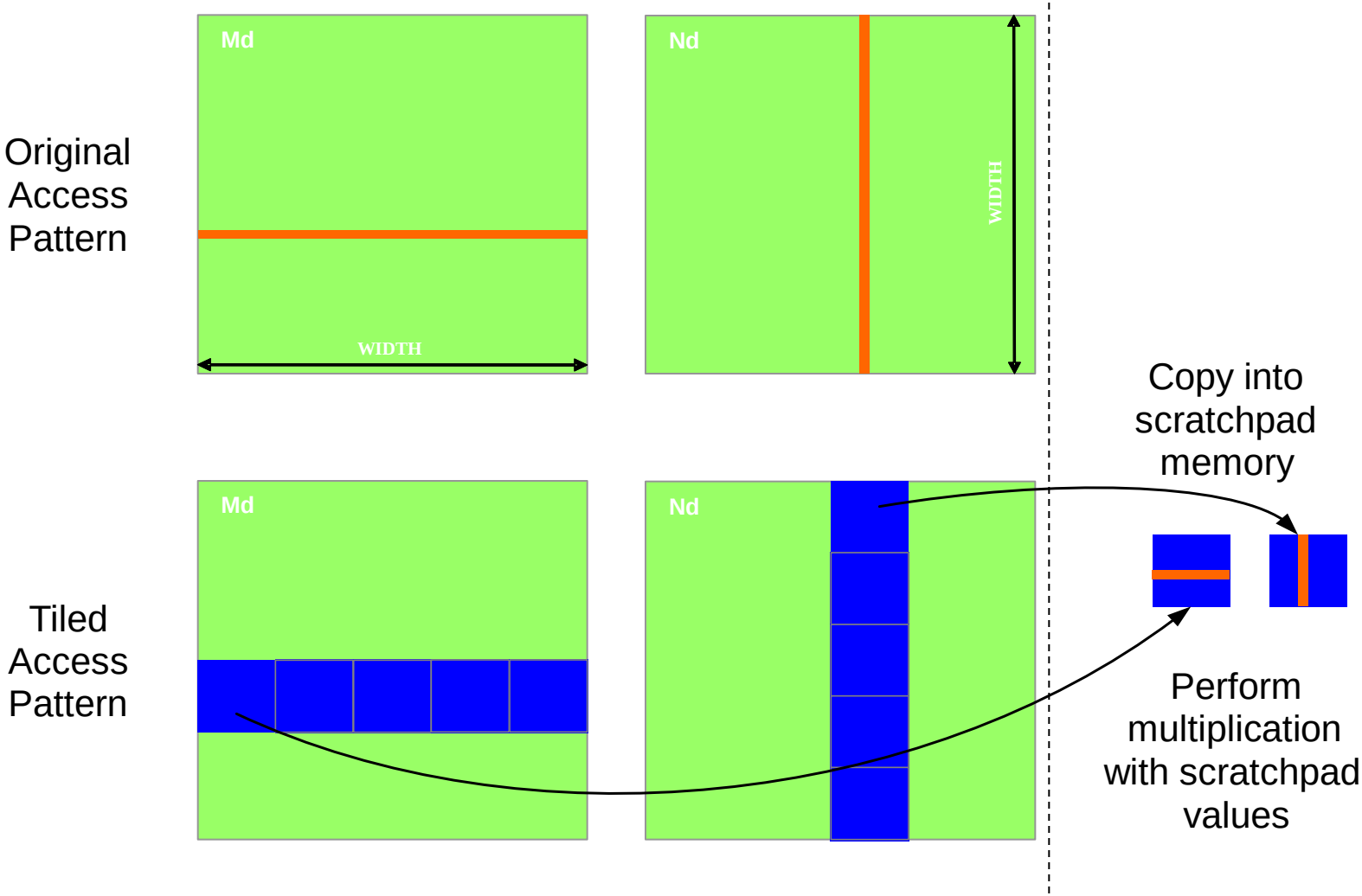
- Make sure that tiles are all loaded in vertical patters from the global memory
- Md data can then be accessed from shared memory in horizontal direction



First-order Size Considerations

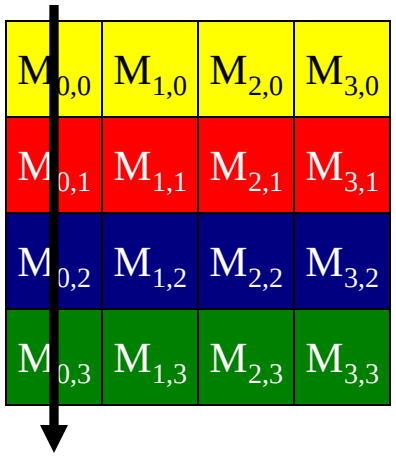
- Assume
  - TILE_WIDTH of 16 gives 16*16 = 256 threads
  - A 1024*1024 Pd gives 64*64 = 4096 Thread Blocks
- Each thread block perform 2*256 = 512 float loads from global memory for 256 * (2*16) = 8,192 mul/add operations.
  - Memory bandwidth no longer a limiting factor
  - Could use thread coarsening to further reduce traffic
- Each thread block can have up to 1024 threads
  - Can use 32*32 tiles to further reduce traffic

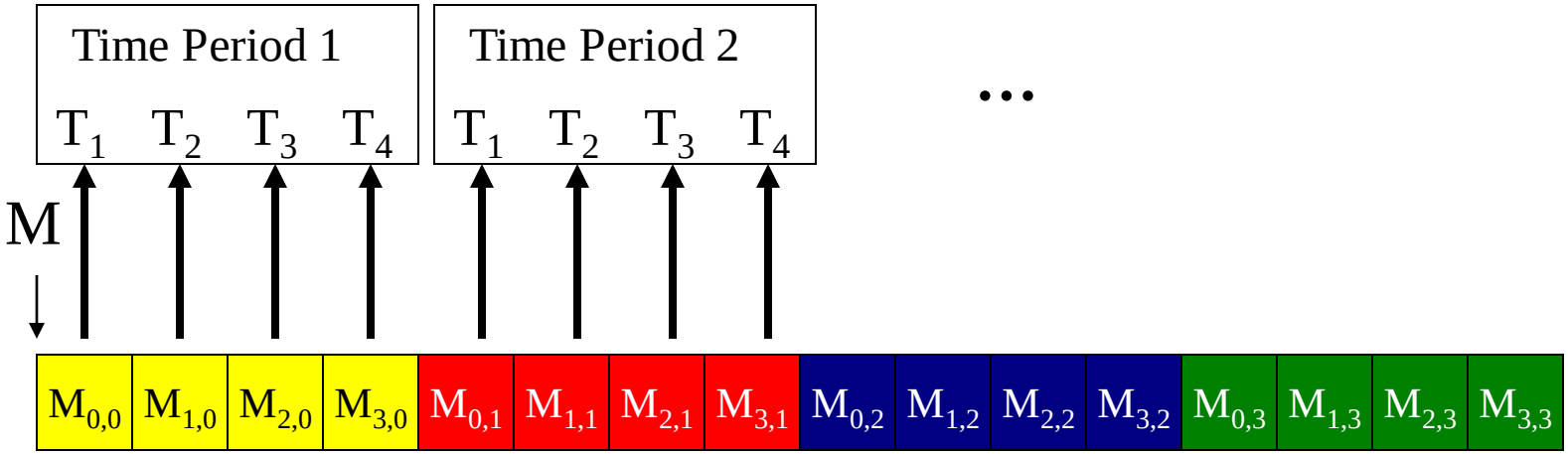
Memory Access Pattern (Corner Turning)



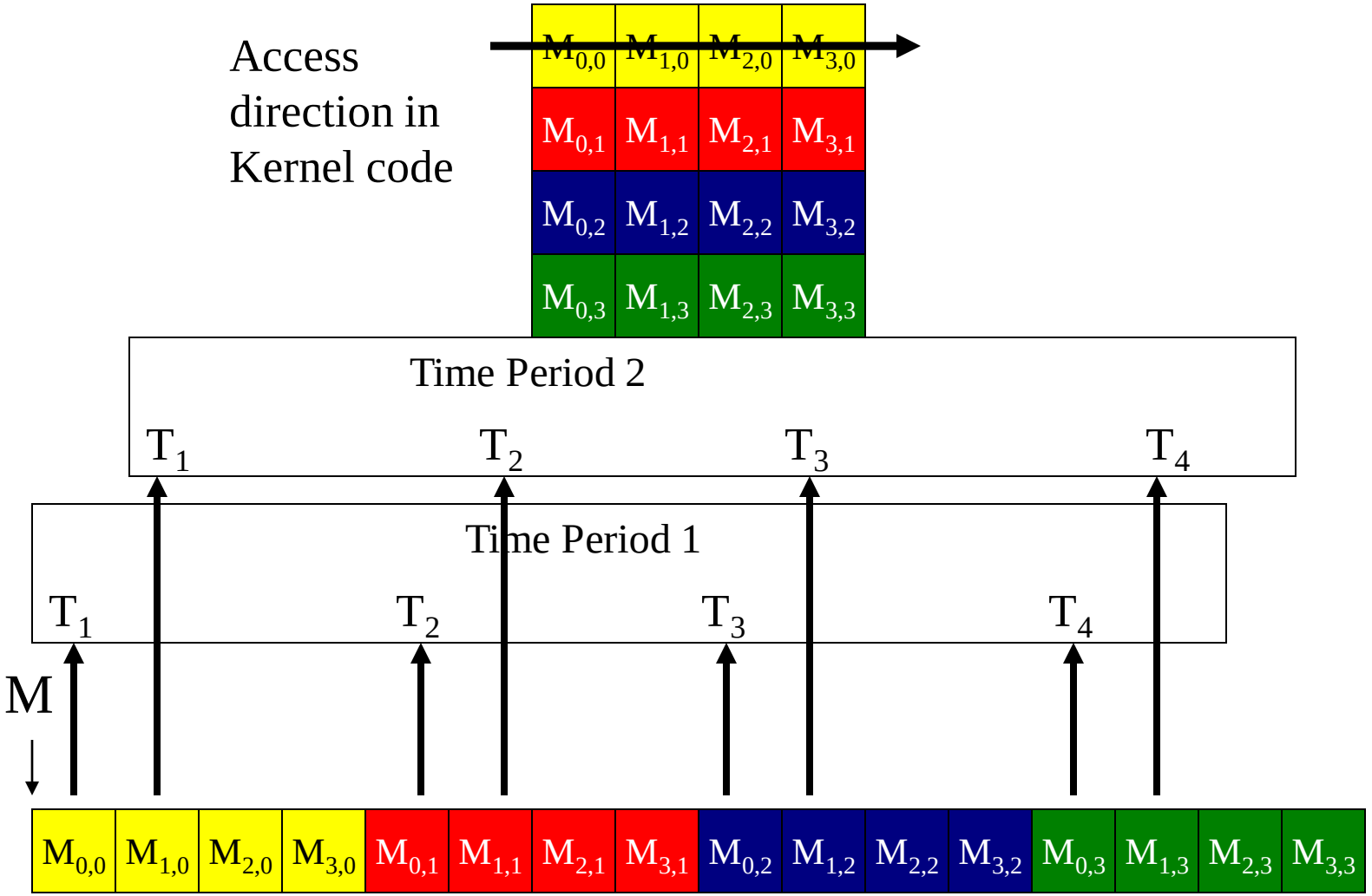
Memory Layout of a Matrix in C

Access direction in Kernel code





Memory Layout of a Matrix in C



Loading a Tile

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

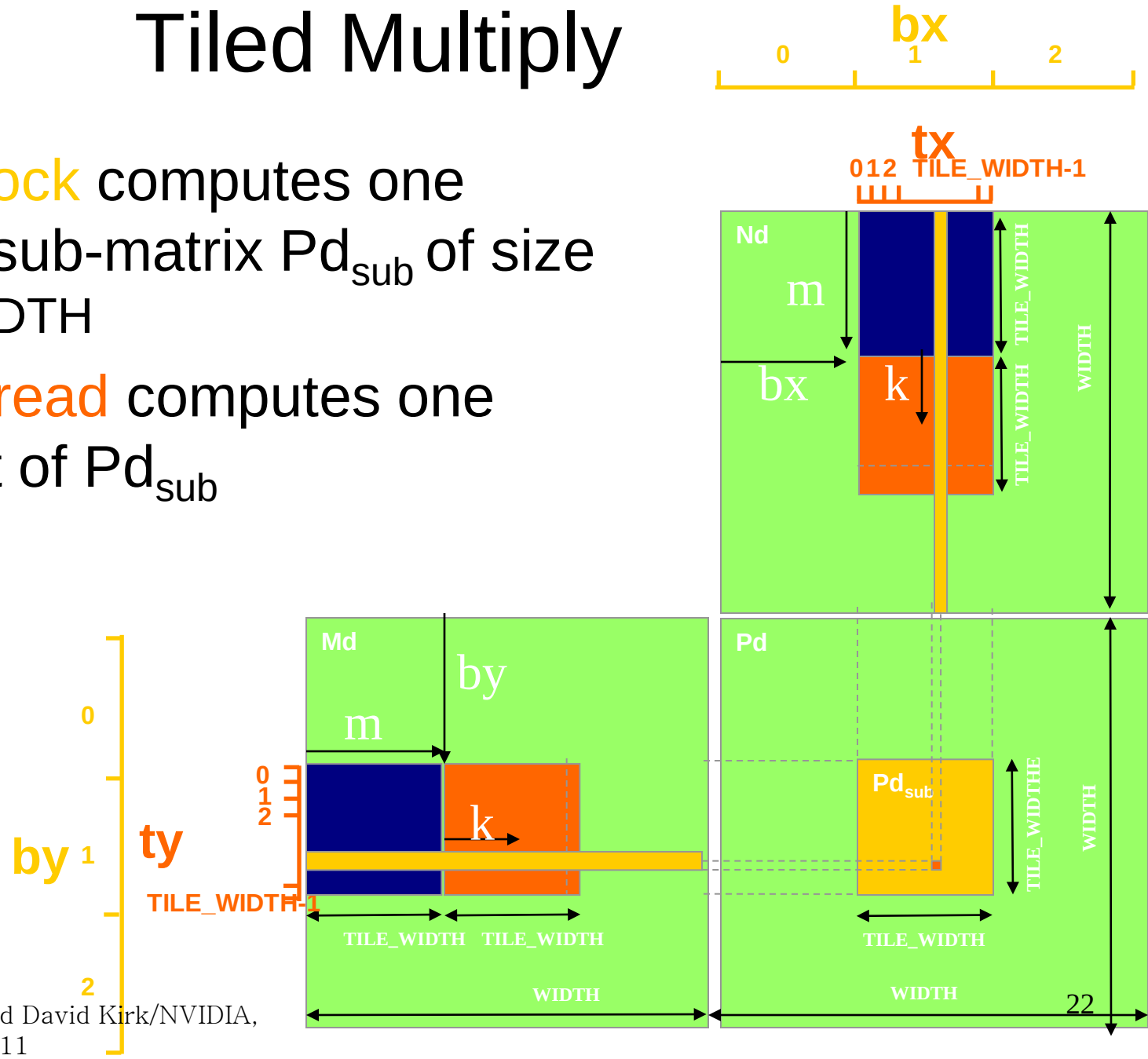
- Assign the loaded element to each thread such that the accesses within each warp is coalesced

CUDA Code – Kernel Execution Configuration

```
// Setup the execution configuration
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);
dim3 dimGrid(Width  / TILE_WIDTH,
             Width /  TILE_WIDTH);
```

Tiled Multiply

- Each block computes one square sub-matrix Pd_{sub} of size TILE_WIDTH
- Each thread computes one element of Pd_{sub}



Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
1.  __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
2.  __shared__ float Nds[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.     Mds[tx][ty] = Md[Row*Width + m*TILE_WIDTH + tx];
10.    Nds[tx][ty] = Nd[(m*TILE_WIDTH + ty) * Width + Col)];
11.    __syncthreads();
12.   for (int k = 0; k < TILE_WIDTH; ++k)
13.      Pvalue += Mds[tx][k] * Nds[k][ty];
14.    __syncthreads();
15.}
16.   Pd[Row*Width+Col] = Pvalue;
}
```

Shared Memory and Threading

- Each SM in Fermi has 64KB on-chip SRAM, partitioned into 48KB L1 cache and 16KB shared memory, or vice versa
  - SM shared memory 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 (Problem with earlier GPUs!)
- Using 16x16 tiling, we reduce the accesses to the global memory by a factor of 16
  - A 150GB/s bandwidth can now support (150/4)*16 = 600 GFLOPS!

ANY MORE QUESTIONS?