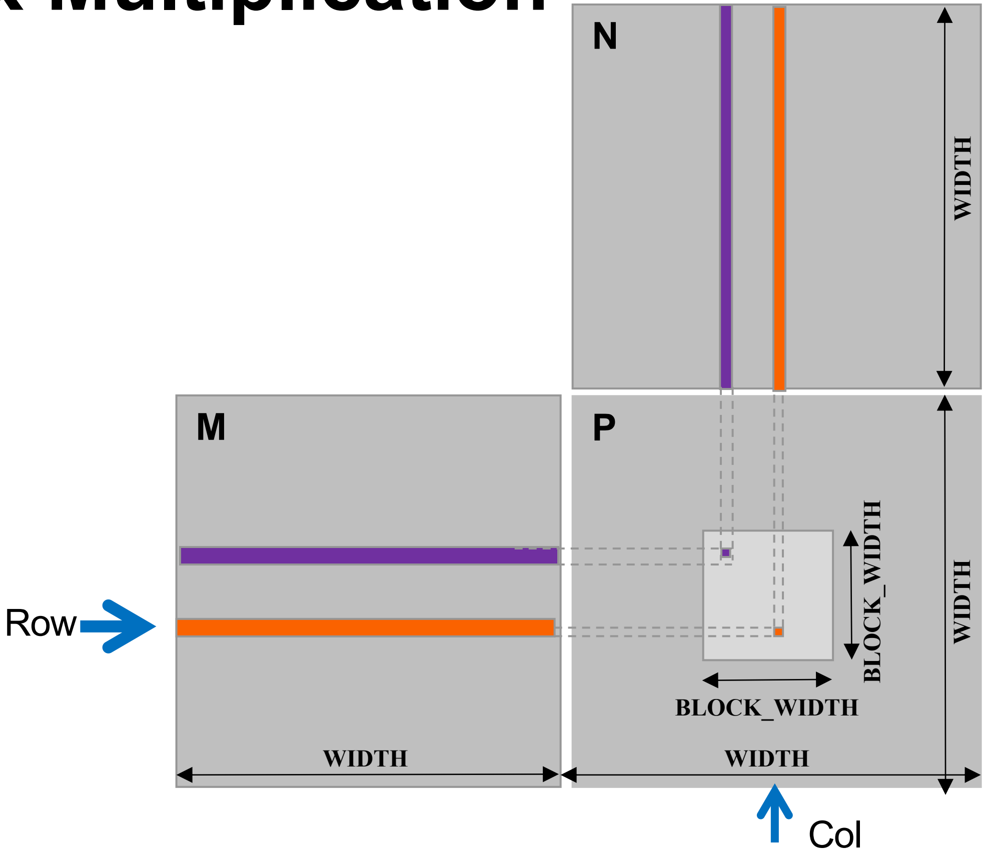
Matrix Multiply (Memory and Data Locality)

UNIVERSITY OF CALIFORNIA, RIVERSIDE

Slide credit: Slides adapted from

Example – Matrix Multiplication



A Basic Matrix Multiplication

```
__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;
  }

}
```

2D Index

Each thread maps to an output element

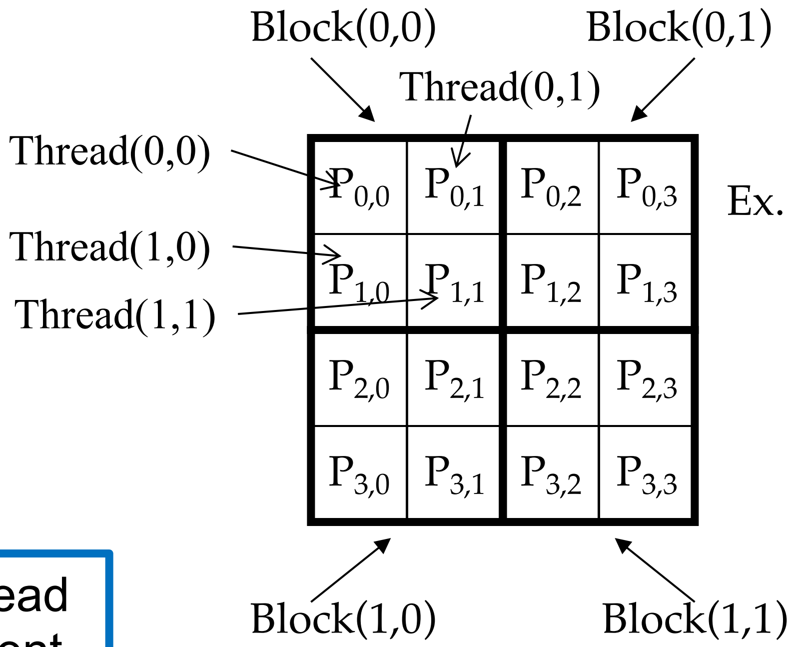
Example – Matrix Multiplication

```
__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;
  }

}
```

Dot product

Each thread maps to an output element

A Toy Example: Thread to Output Data Mapping



Ex. BLOCK_WIDTH = 2

We map each thread to an output element.

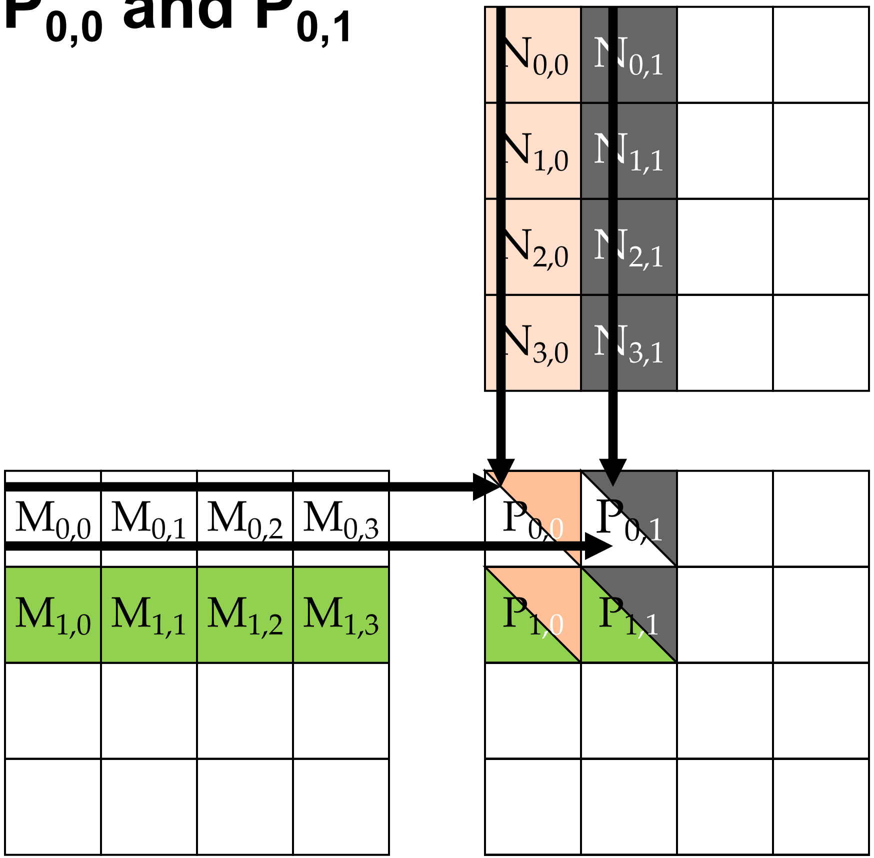
Note:

Thread(Row,Col)

Block(Row,Col)

$P_{Row,Col}$

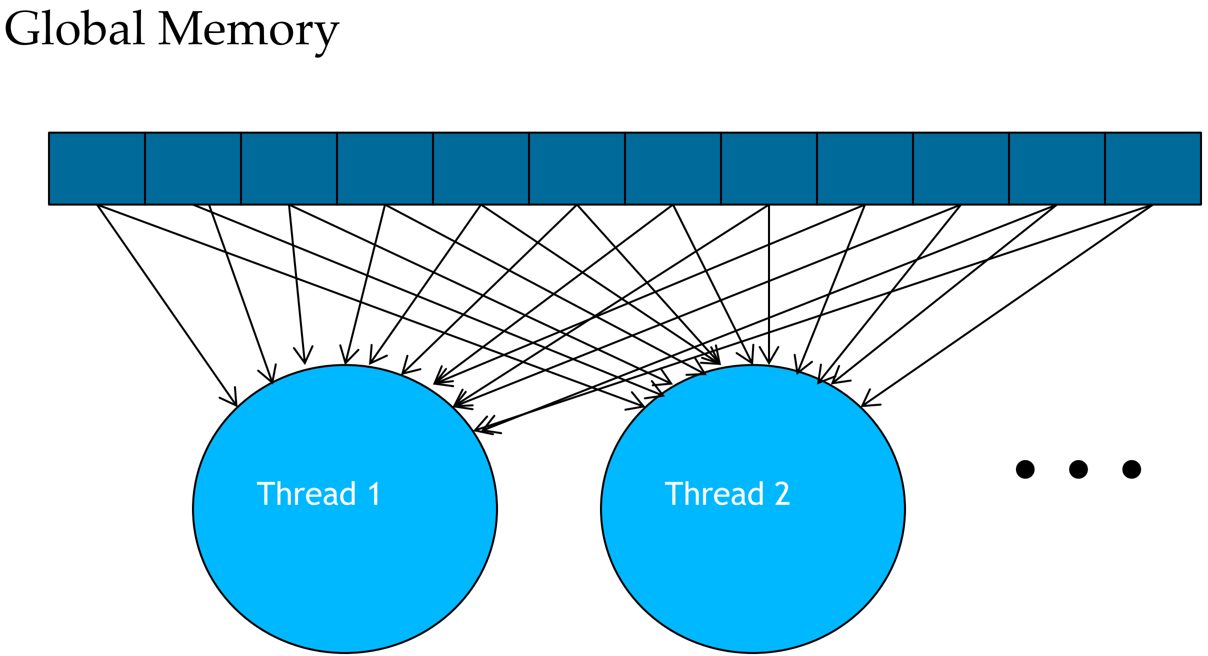
Calculation of $P_{0,0}$ and $P_{0,1}$



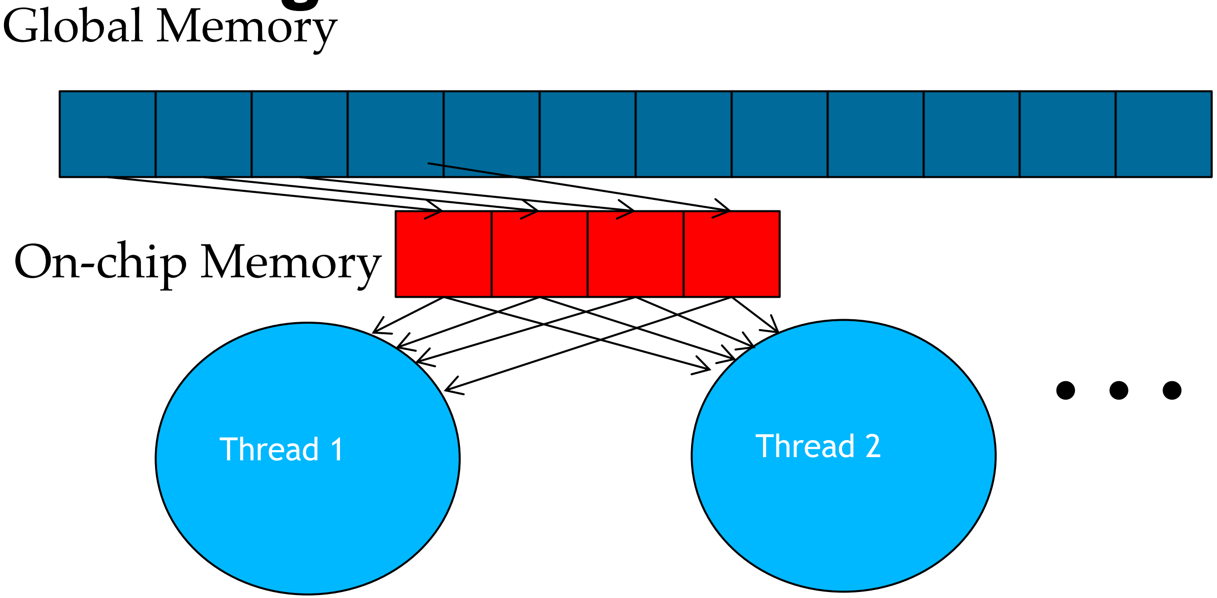
We map each thread to an output element.

TILED PARALLEL ALGORITHMS

Global Memory Access Pattern of the Basic Matrix Multiplication Kernel



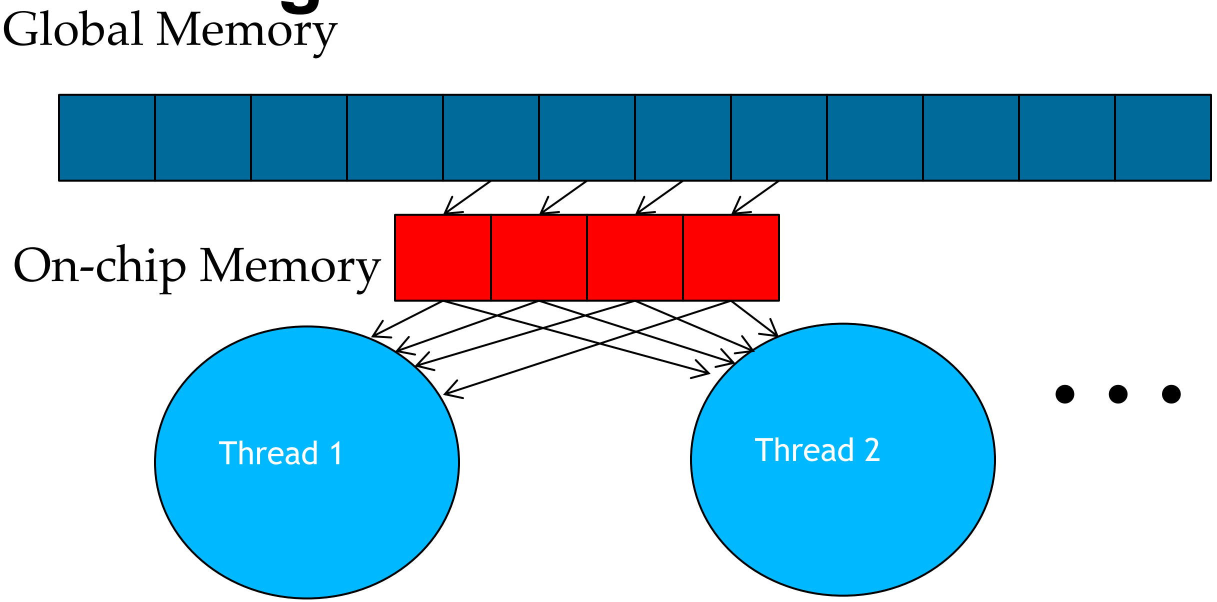
Tiling/Blocking - Basic Idea



Divide the global memory content into tiles

Focus the computation of threads on one or a small number of tiles at each point in time

Tiling/Blocking - Basic Idea



Outline of Tiling Technique

1. Identify a tile of global memory contents that are accessed by multiple threads

2. Load the tile from global memory into on-chip memory

3. Use barrier synchronization to make sure that all threads are ready to start the phase

4. Have the multiple threads to access their data from shared memory

5. Use barrier synchronization to make sure that all threads have completed the current phase

6. Move on to the next tile, repeat step 4.

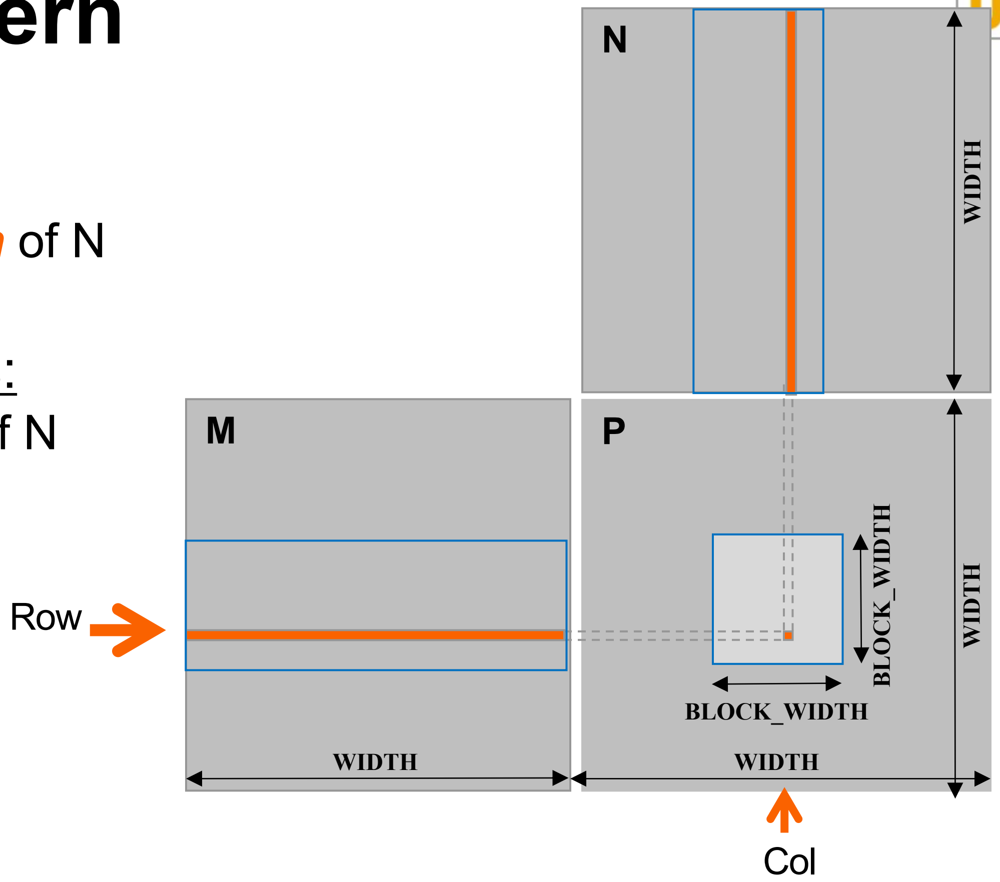
TILED MATRIX MULTIPLICATION

Objective

- To understand the design of a tiled parallel algorithm for matrix multiplication
  - Loading a tile
  - Phased execution
  - Barrier Synchronization

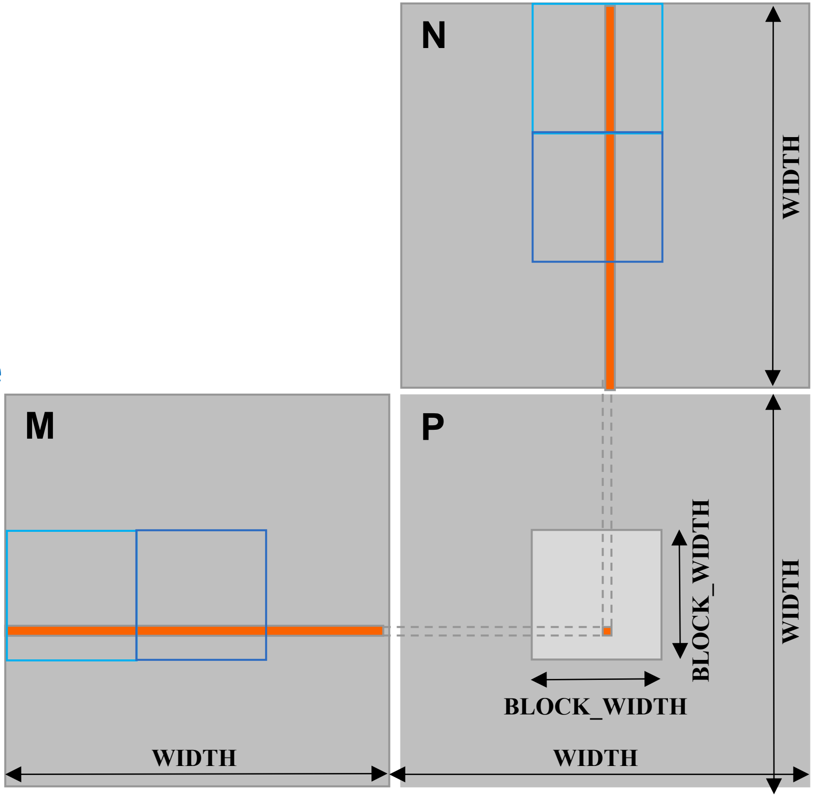
Data access pattern

- Each thread access:
  - a ***row*** of M and a ***column*** of N
- Each thread block access:
  - a ***strip*** of M and a ***strip*** of N



Tiled Matrix Multiplication Phases

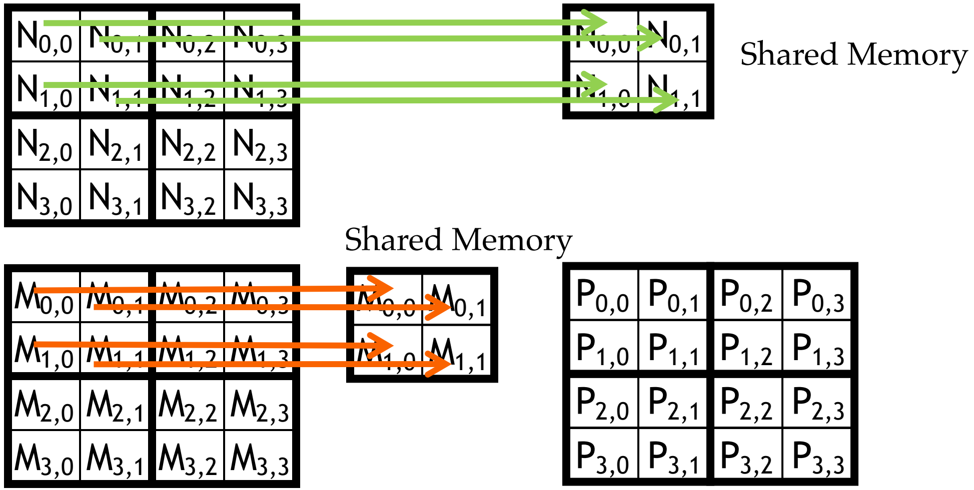
- Break up the execution of each thread into ***phases***
- data accesses by the thread block in each ***phase*** are focused on one tile of M and one tile of N
- The tile is of BLOCK_SIZE elements in each dimension



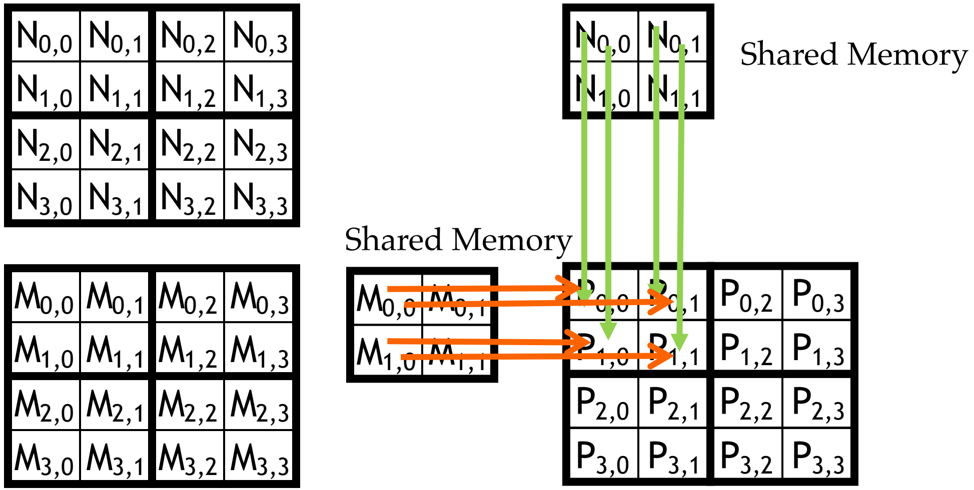
Loading a Tile

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

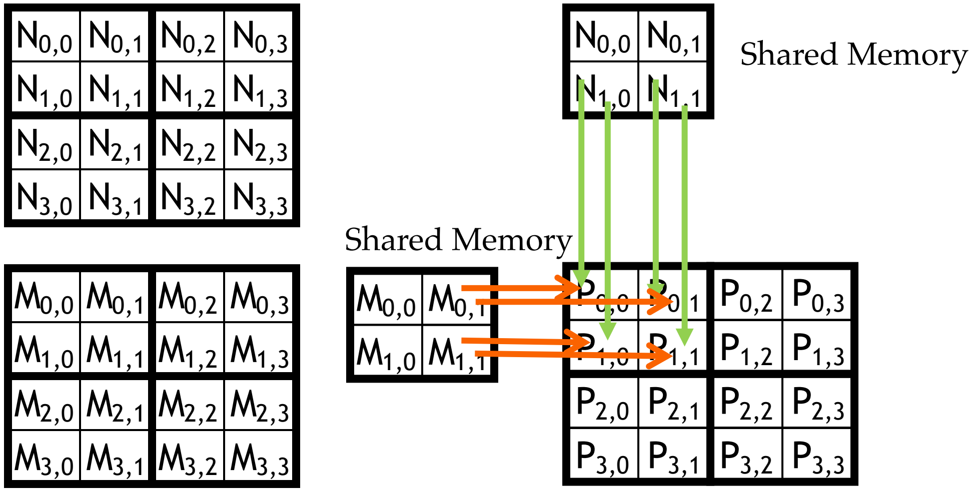
Phase 0 Load for Block (0,0)



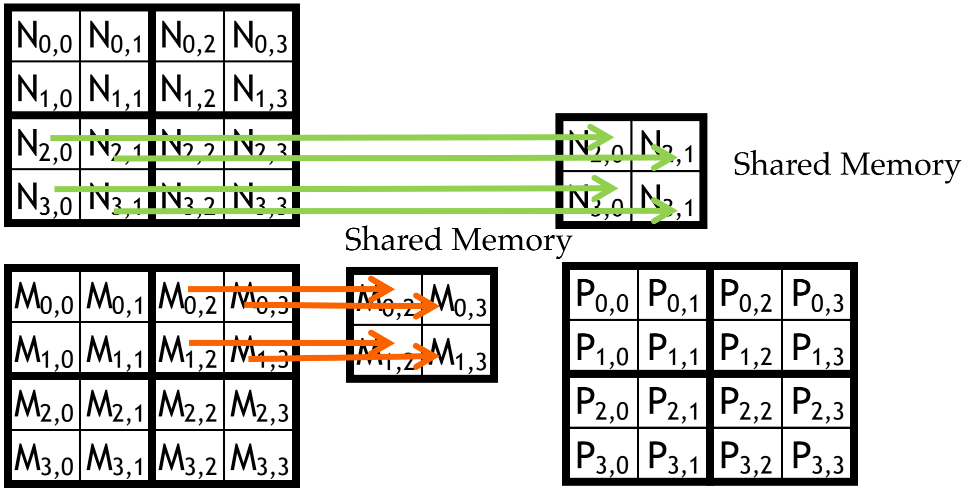
Phase 0 Use for Block (0,0) (iteration 0)



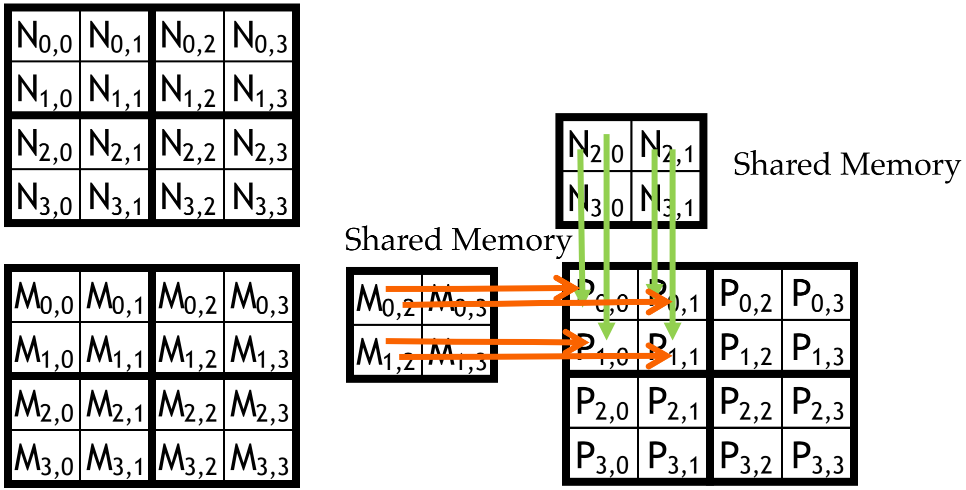
Phase 0 Use for Block (0,0) (iteration 1)



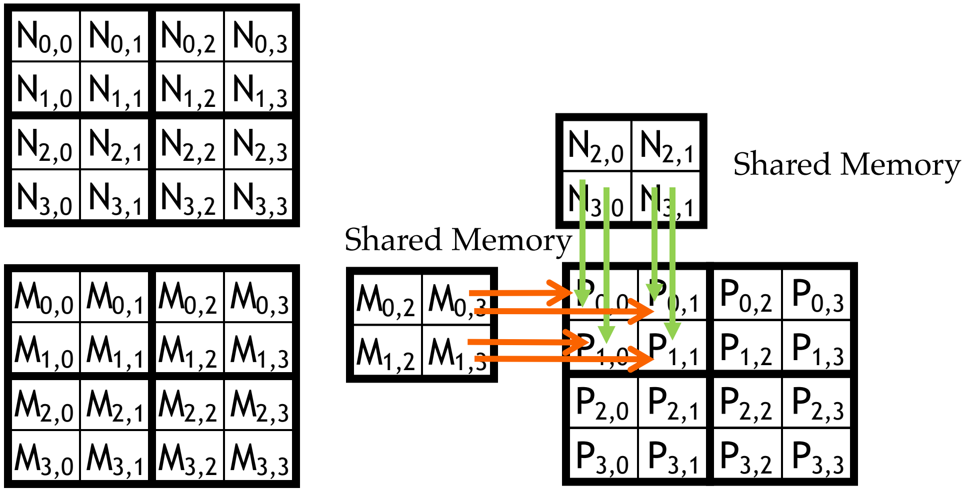
Phase 1 Load for Block (0,0)



Phase 1 Use for Block (0,0) (iteration 0)



Phase 1 Use for Block (0,0) (iteration 1)



Execution Phases of Toy Example

	Phase 0			Phase 1		
$thread_{0,0}$	$\mathbf{M_{0,0}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{0,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}*Nds_{0,0}$ + $Mds_{0,1}*Nds_{1,0}$	$\mathbf{M_{0,2}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{2,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}*Nds_{0,0}$ + $Mds_{0,1}*Nds_{1,0}$
$thread_{0,1}$	$\mathbf{M_{0,1}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{0,1}}$ ↓ $Nds_{1,0}$	$PValue_{0,1}$ += $Mds_{0,0}*Nds_{0,1}$ + $Mds_{0,1}*Nds_{1,1}$	$\mathbf{M_{0,3}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{2,1}}$ ↓ $Nds_{0,1}$	$PValue_{0,1}$ += $Mds_{0,0}*Nds_{0,1}$ + $Mds_{0,1}*Nds_{1,1}$
$thread_{1,0}$	$\mathbf{M_{1,0}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{1,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}*Nds_{0,0}$ + $Mds_{1,1}*Nds_{1,0}$	$\mathbf{M_{1,2}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{3,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}*Nds_{0,0}$ + $Mds_{1,1}*Nds_{1,0}$
$thread_{1,1}$	$\mathbf{M_{1,1}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{1,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}*Nds_{0,1}$ + $Mds_{1,1}*Nds_{1,1}$	$\mathbf{M_{1,3}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{3,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}*Nds_{0,1}$ + $Mds_{1,1}*Nds_{1,1}$

time →

Execution Phases of Toy Example (cont.)

	Phase 0			Phase 1		
$thread_{0,0}$	$\mathbf{M_{0,0}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{0,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$	$\mathbf{M_{0,2}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{2,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$
$thread_{0,1}$	$\mathbf{M_{0,1}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{0,1}}$ ↓ $Nds_{1,0}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$	$\mathbf{M_{0,3}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{2,1}}$ ↓ $Nds_{0,1}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$
$thread_{1,0}$	$\mathbf{M_{1,0}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{1,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$	$\mathbf{M_{1,2}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{3,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$
$thread_{1,1}$	$\mathbf{M_{1,1}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{1,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$	$\mathbf{M_{1,3}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{3,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$

time →

Shared memory allows each value to be accessed by multiple threads

Barrier Synchronization

- Synchronize all threads in a block
  - __syncthreads()
- All threads in the same block must reach the __syncthreads() before any of the them can move on
- Best used to coordinate the phased execution tiled algorithms
  - To ensure that all elements of a tile are loaded at the beginning of a phase
  - To ensure that all elements of a tile are consumed at the end of a phase

TILED MATRIX MULTIPLICATION KERNEL

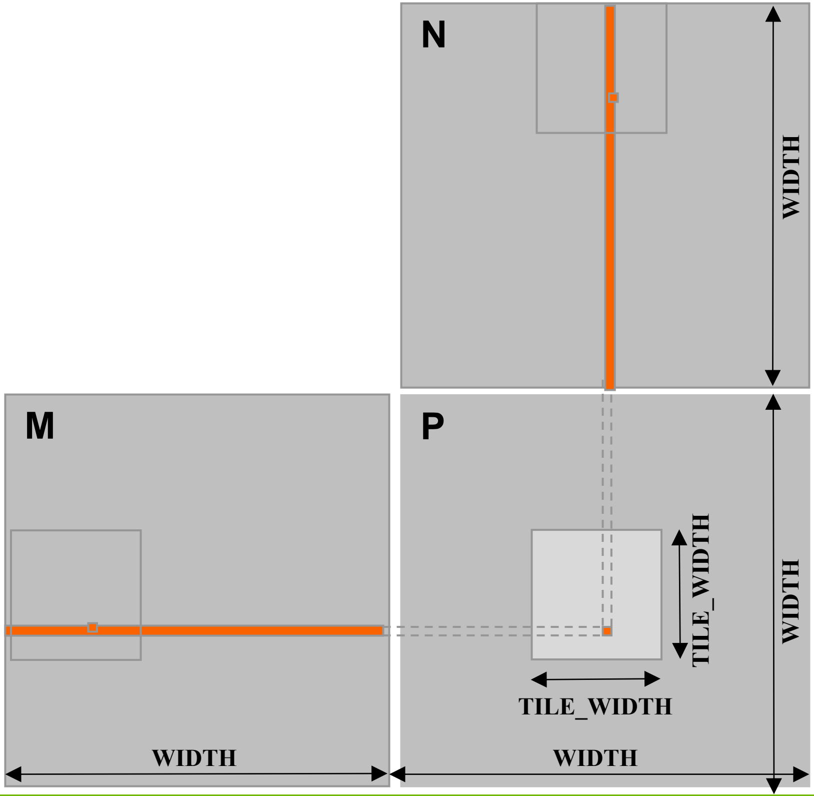
Objective

- To learn to write a tiled matrix-multiplication kernel
  - Loading and using tiles for matrix multiplication
  - Barrier synchronization, shared memory
  - Resource Considerations
  - Assume that Width is a multiple of tile size for simplicity

Loading Input Tile 0 of M (Phase 0)

– Have each thread load an M element and an N element at the same relative position as its P element.

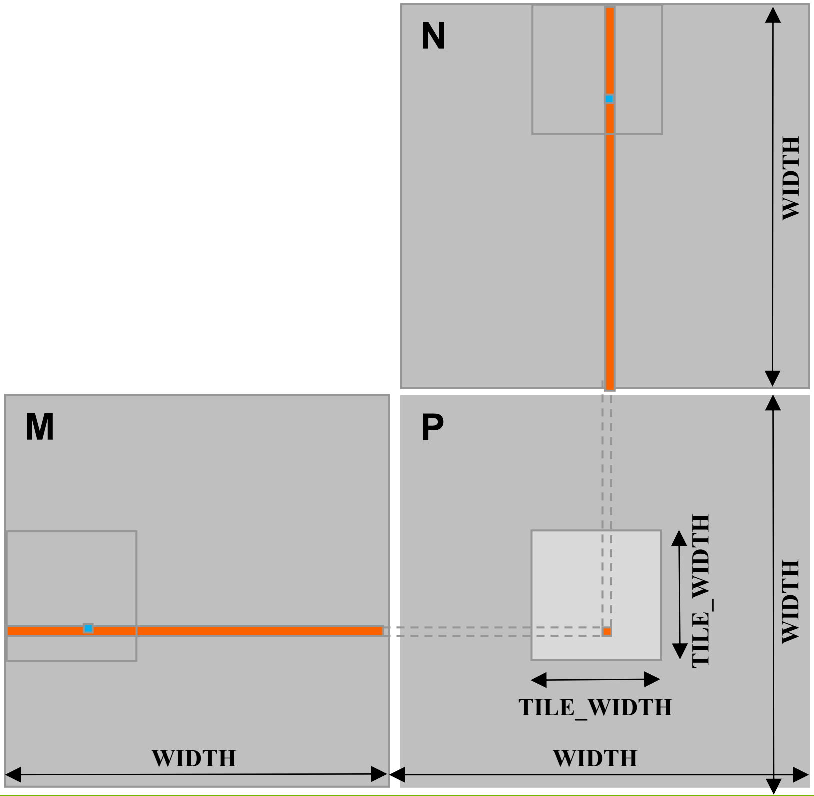
```
int Row = by * blockDim.y + ty;
int Col =   bx * blockDim.x + tx;
2D indexing for accessing Tile 0:
            M[Row][tx]
            N[ty][Col]
```



Loading Input Tile 0 of N (Phase 0)

– Have each thread load
  an M element and
  an N element
  at the same relative position as
  its P element.

```
int Row = by * blockDim.y + ty;
int Col =   bx * blockDim.x + tx;
2D indexing for accessing Tile 0:
            M[Row][tx]
            N[ty][Col]
```

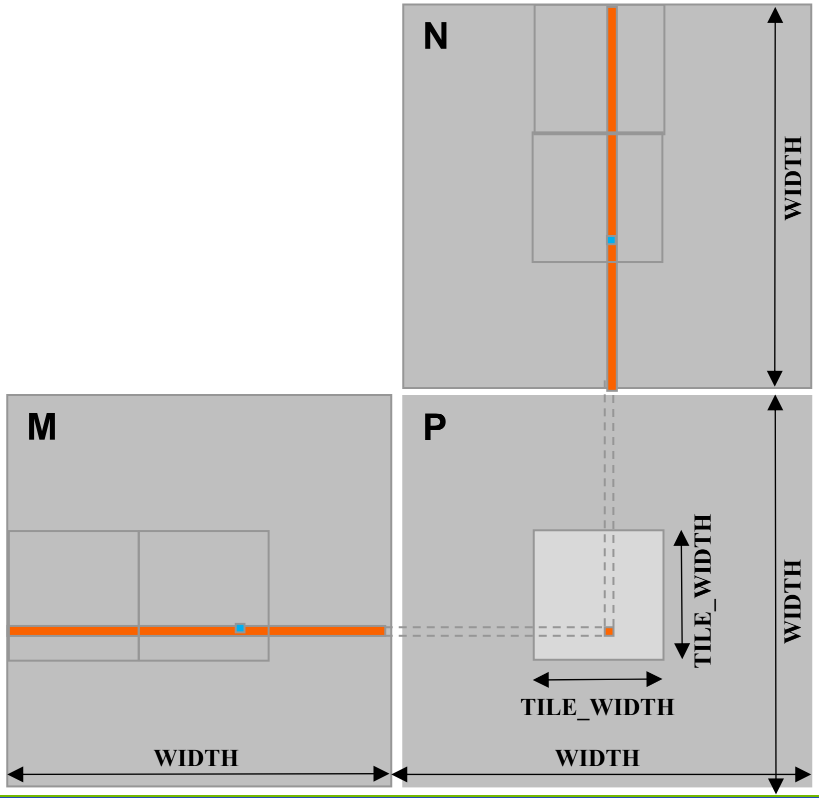


Loading Input Tile 1 of M (Phase 1)

2D indexing for accessing Tile 1:

$\boxed{\text{M[Row][1*TILE\_WIDTH + tx]}}$

N[1*TILE*WIDTH + ty][Col]

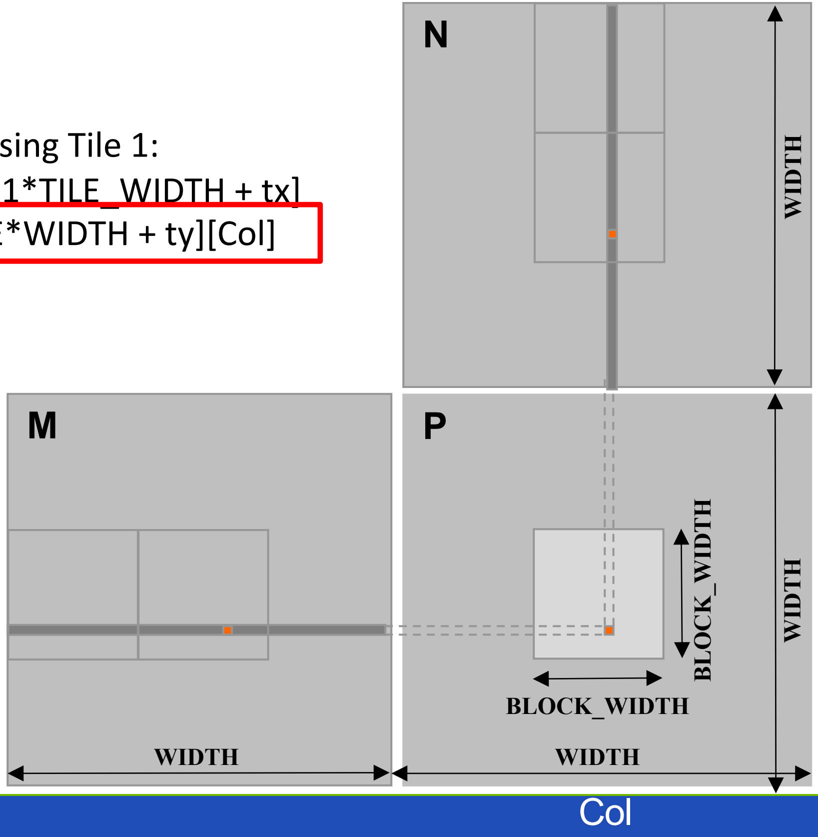


Loading Input Tile 1 of N (Phase 1)

2D indexing for accessing Tile 1:

M[Row][1*TILE_WIDTH + tx]

N[1*TILE*WIDTH + ty][Col]



M and N are dynamically allocated - use 1D indexing

M[Row][p*TILE_WIDTH+tx]
⇨ M[Row*Width + p*TILE_WIDTH + tx]

N[p*TILE_WIDTH+ty][Col]
⇨ N[(p*TILE_WIDTH+ty)*Width + Col]

where p is the sequence number of the current phase

Tile (Thread Block) 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 TILE_WIDTH 16, in each phase, how many loads from global memory are performed?

- For TILE_WIDTH 16, in each phase, how many mul/add operations are performed?

- For 16, in each phase, each block performs 2*256 = 512 float loads from global memory for 256 * (2*16) = 8,192 mul/add operations. (16 **floating-point operations for each memory load**)

Tile (Thread Block) 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 TILE_WIDTH 32, in each phase, how many loads from global memory are performed?

- For TILE_WIDTH 32, in each phase, how many mul/add operations are performed?

- For 32, in each phase, each block performs 2*1024 = 2048 float loads from global memory for 1024 * (2*32) = 65,536 mul/add operations. (32 **floating-point operation for each memory load**)

Shared Memory and Threading

- For an SM with 16KB shared memory
  - Shared memory size is implementation dependent!
- For TILE_WIDTH = 16,
  each thread block uses 2*16*16*4 Byte = 2KB of shared memory.
  - For 16KB shared memory, one can potentially have up to 8 thread blocks executing
- For TILE_WIDTH = 32,
  each thread block uses 2*32*32*4 Byte= 8KB of shared memory.
  - Allows 2 thread blocks active at the same time
- Each __syncthread() can reduce the number of active threads for a block
  - More thread blocks can be advantageous

HANDLING ARBITRARY MATRIX SIZES IN TILED ALGORITHMS

Objective

- To learn to handle arbitrary matrix sizes in tiled matrix multiplication
  - Boundary condition checking
  - Regularizing tile contents
  - Rectangular matrices

Handling Matrix of Arbitrary Size

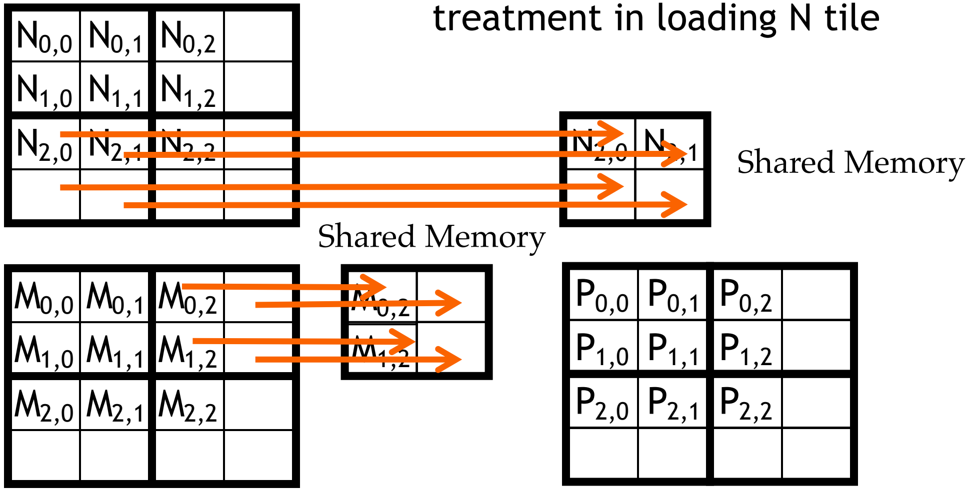
- The tiled matrix multiplication kernel we presented so far can handle only square matrices whose dimensions (Width) are multiples of the tile width (TILE_WIDTH)
  - However, real applications need to handle *arbitrary sized matrices*.

- One could pad (add elements to) the rows and columns into multiples of the tile size, but would have significant space and data transfer time overhead.

- We will take a different approach.

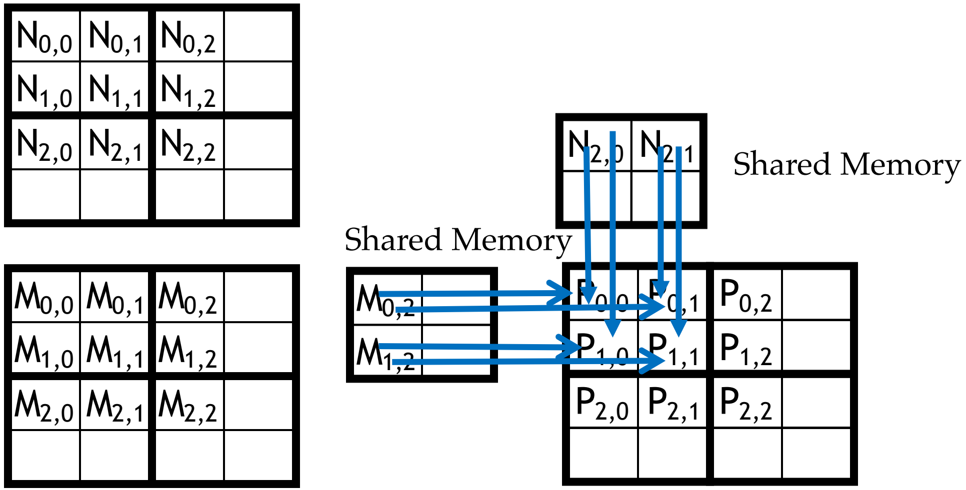
Phase 1 Loads for Block (0,0) for a 3x3 Example

Threads (1,0) and (1,1) need special treatment in loading N tile

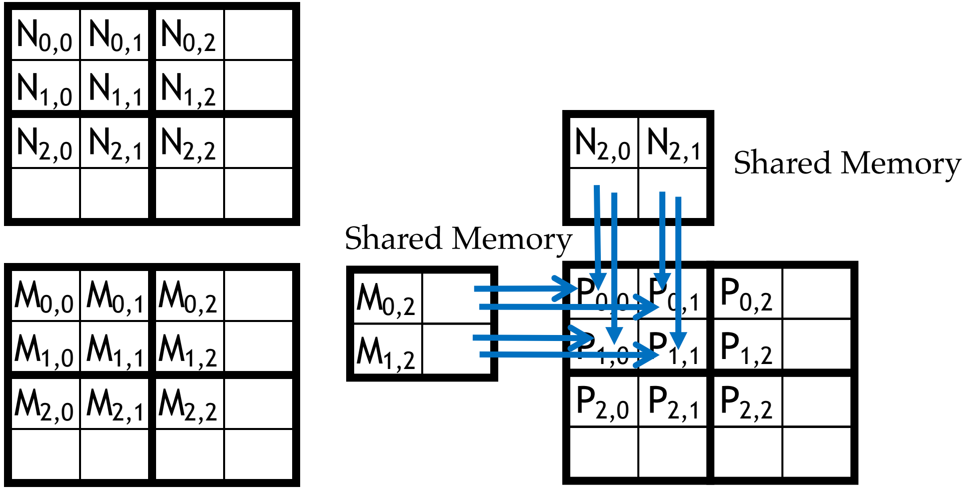


Threads (0,1) and (1,1) need special treatment in loading M tile

Phase 1 Use for Block (0,0) (iteration 0)



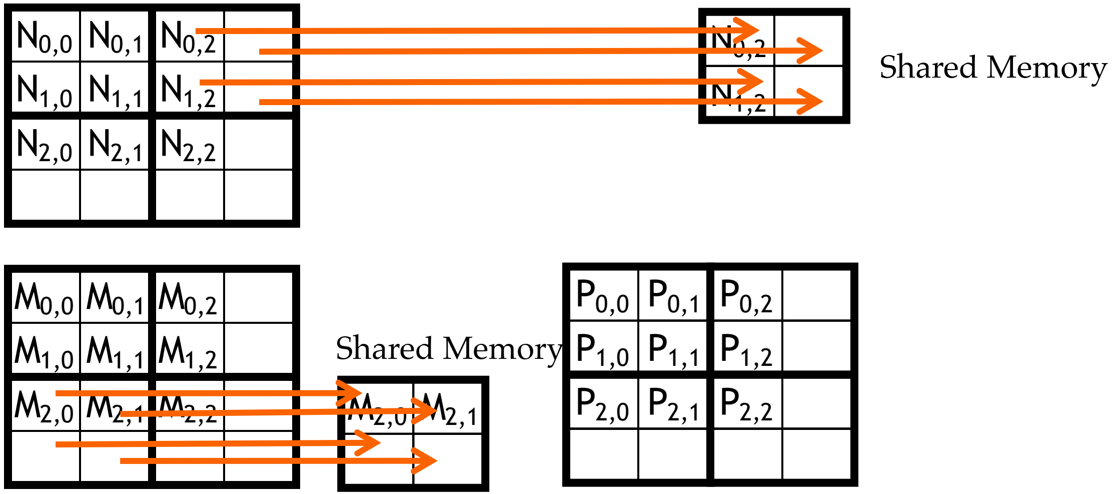
Phase 1 Use for Block (0,0) (iteration 1)



All Threads need special treatment. None of them should introduce invalidate contributions to their P elements.

Phase 0 Loads for Block (1,1) for a 3x3 Example

Threads (0,1) and (1,1) need special treatment in loading N tile



Threads (1,0) and (1,1) need special treatment in loading M tile

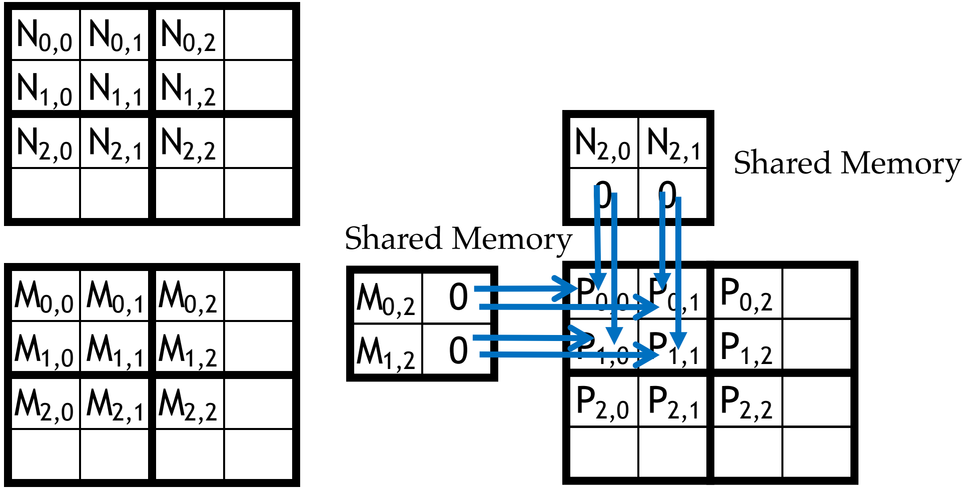
A “Simple” Solution

- When a thread is to load any input element, test if it is in the valid index range
  - If valid, proceed to load
  - Else, do not load, just write a 0

- Rationale: a 0 value will ensure that that the multiply-add step does not affect the final value of the output element

- The condition tested for loading input elements is different from the test for calculating output P element
  - A thread that does not calculate valid P element can still participate in loading input tile elements

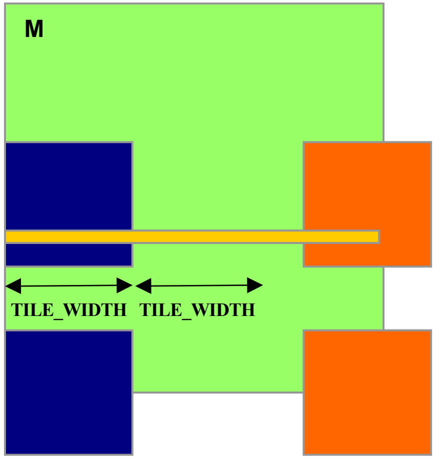
Phase 1 Use for Block (0,0) (iteration 1)



Boundary Condition for Input M Tile

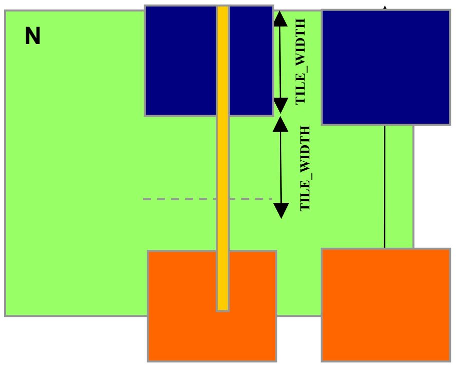
- Each thread loads
  - M[Row][p*TILE_WIDTH+tx]
  - M[Row*Width + p*TILE_WIDTH+tx]

- Need to test
  - (Row < Width) && (p*TILE_WIDTH+tx < Width)
  - If true, load M element
  - Else , load 0



Boundary Condition for Input N Tile

- Each thread loads
  - N[p*TILE_WIDTH+ty][Col]
  - N[(p*TILE_WIDTH+ty)*Width+ Col]
- Need to test
  - (p*TILE_WIDTH+ty < Width) && (Col< Width)
  - If true, load N element
  - Else , load 0



Some Important Points

- For each thread the conditions are different for
  - Loading M element
  - Loading N element
  - Calculating and storing output elements

- The effect of control divergence should be small for large matrices

Handling General Rectangular Matrices

- In general, the matrix multiplication is defined in terms of rectangular matrices
  - j x k **M** matrix multiplied with a k x l **N** matrix results in a j x l **P** matrix

- We have presented square matrix multiplication, a special case

- The kernel function needs to be generalized to handle general rectangular matrices
  - The Width argument is replaced by three arguments: j, k, l
  - When Width is used to refer to the height of M or height of P, replace it with j
  - When Width is used to refer to the width of M or height of N, replace it with k
  - When Width is used to refer to the width of N or width of P, replace it with l