CENG443
Heterogeneous Parallel Programming

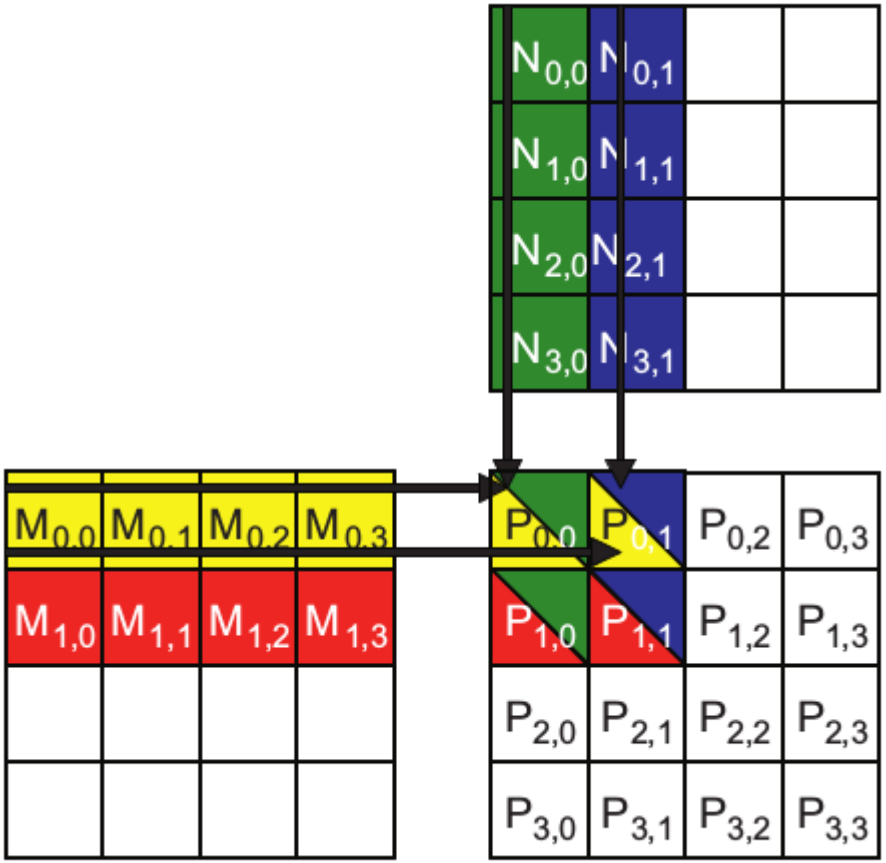
Tiling

Tiling for Reduced Memory Traffic

The global memory is large but slow, whereas the shared memory is small but fast

Partition the data into subsets called tiles so that each tile fits into the shared memory

Matrix Multiplication



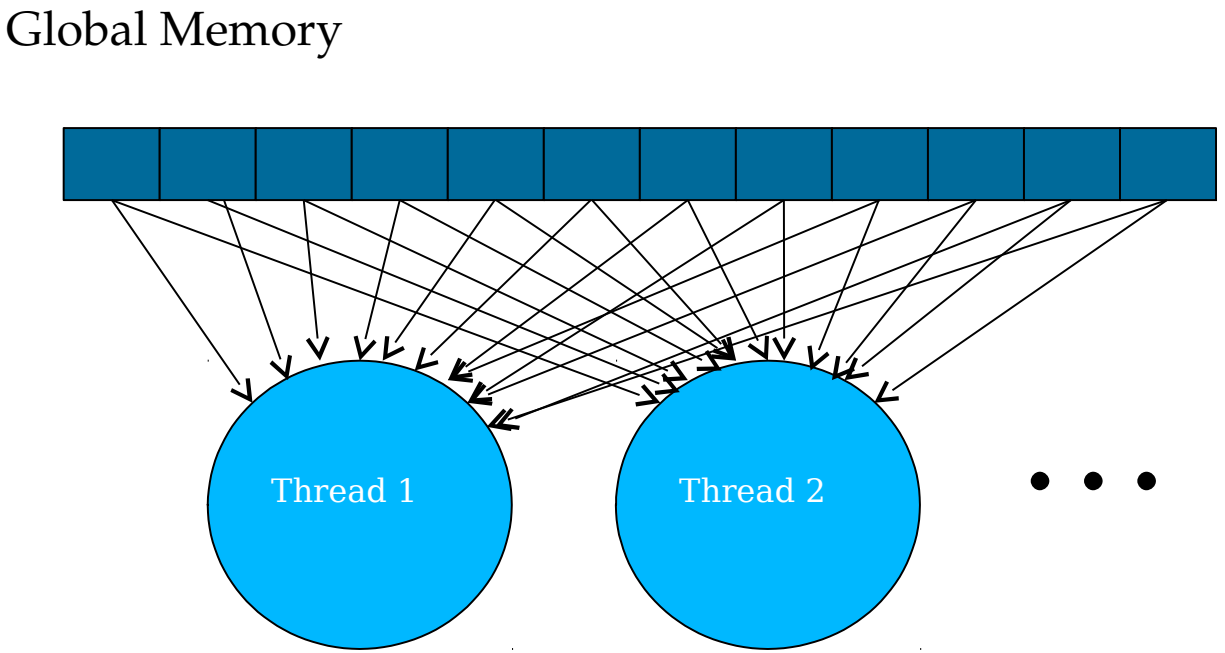
Global memory accesses performed by threads in block$_{0,0}$

If threads can collaborate, M elements are only loaded from the global memory once, the total number of accesses to the global memory can be reduced by half

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

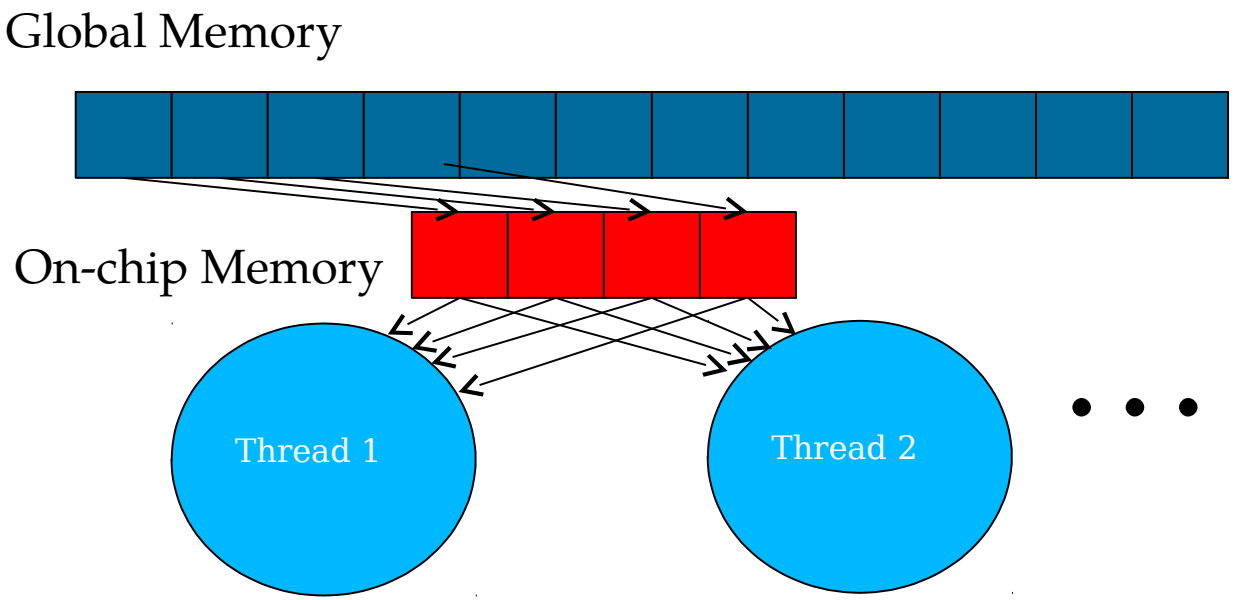
With Width × Width blocks, the potential reduction of global memory traffic would be Width

Global Memory Access Pattern

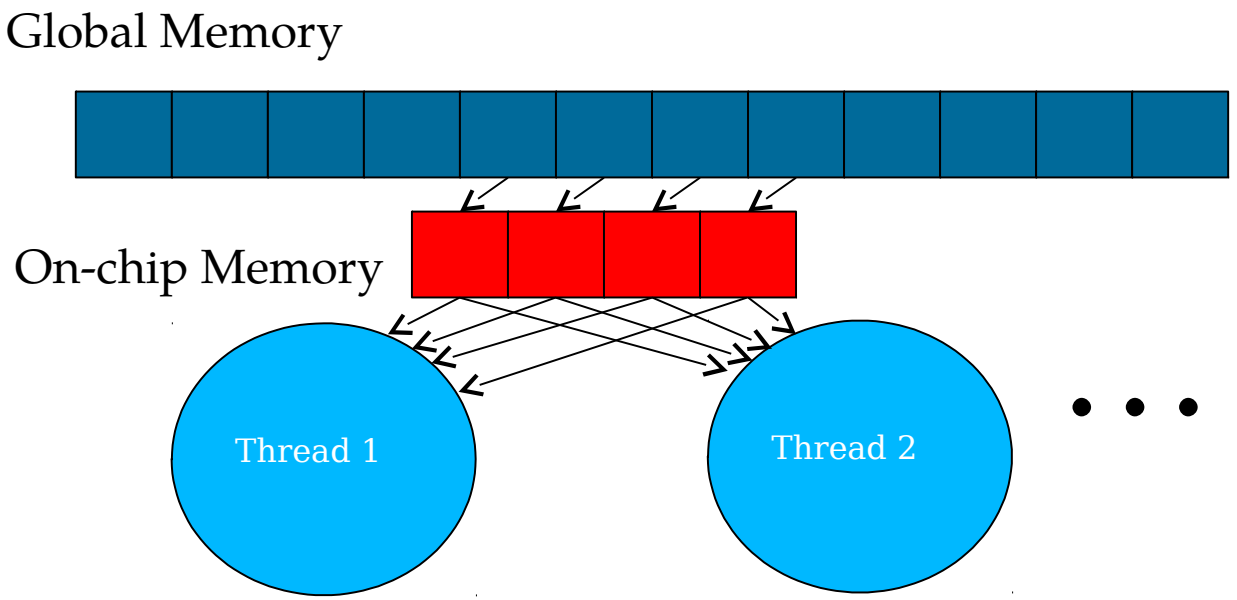


Tiling

Divide the global memory content into tiles



Tiling



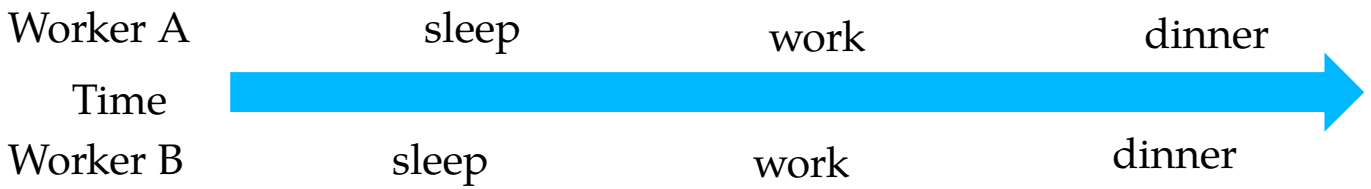
Basic Concept of Tiling

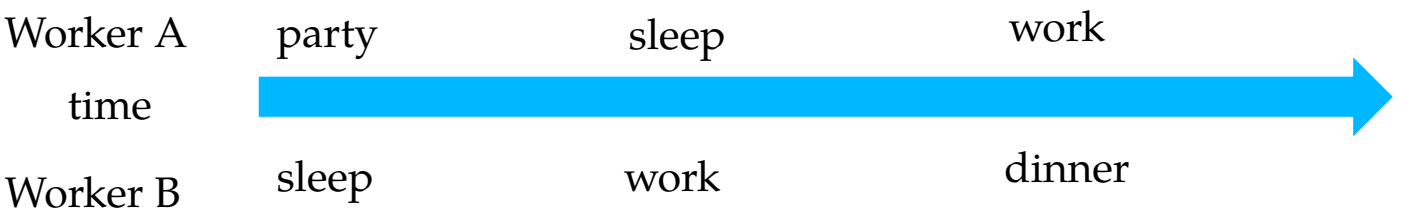
In a congested traffic system, significant reduction of vehicles can greatly improve the delay seen by all vehicles

- Carpooling for commuters (only cars with more than two or three people are allowed to use these lanes)
- Tiling for global memory accesses
  - drivers = threads accessing their memory data operands
  - cars = memory access requests

Carpools Need Synchronization

Good when people have similar schedule

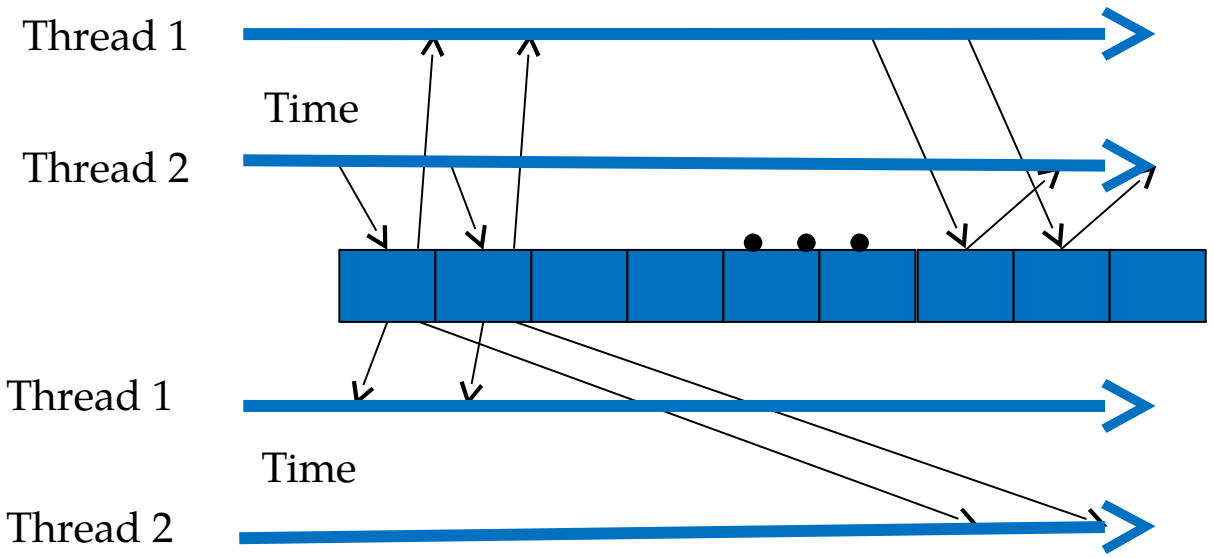




Bad when people have very different schedule

Tiling Requires Synchronization Among Threads

Good when threads have similar access timing



Bad when threads have very different timing

Tiling

Localizes the memory locations accessed among threads and the timing of their accesses

Divides the long access sequences of each thread into phases and uses barrier synchronization to keep the timing of accesses to each section at close intervals

Controls the amount of on-chip memory required by localizing the accesses both in time and in space

Outline of Tiling

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

Load the tile from global memory into on-chip memory

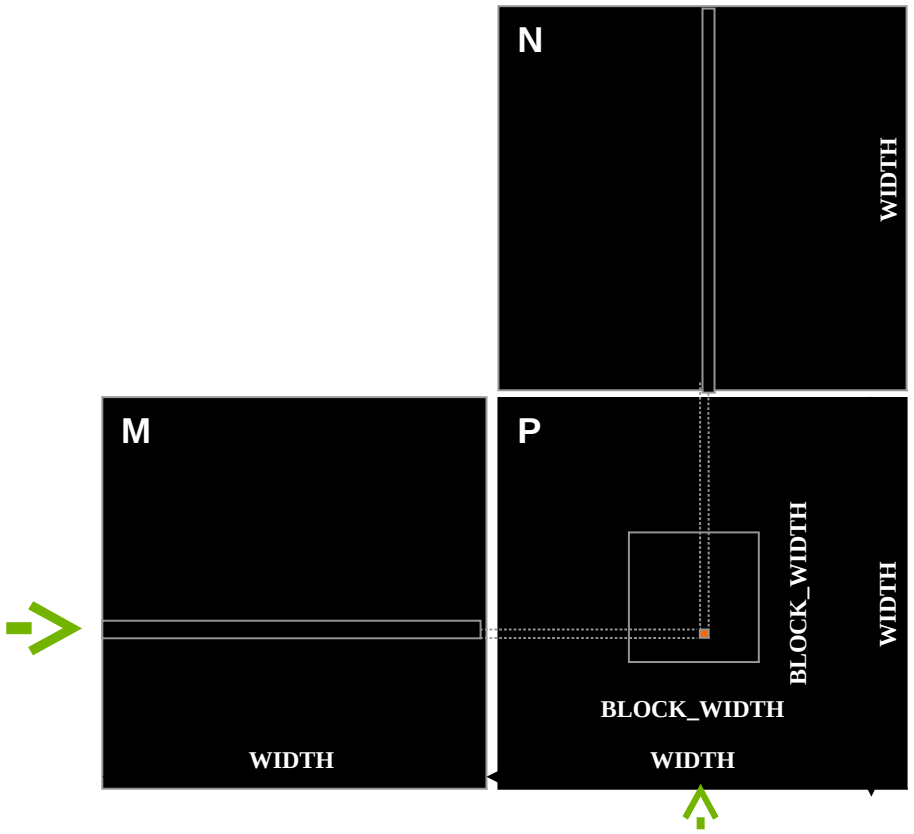
Use barrier synchronization to make sure that all threads are ready to start the phase

Have the multiple threads to access their data from the on-chip memory

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

Move on to the next tile

Example: Matrix Multiplication



A Basic Matrix Multiplication 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;
  }

}
```

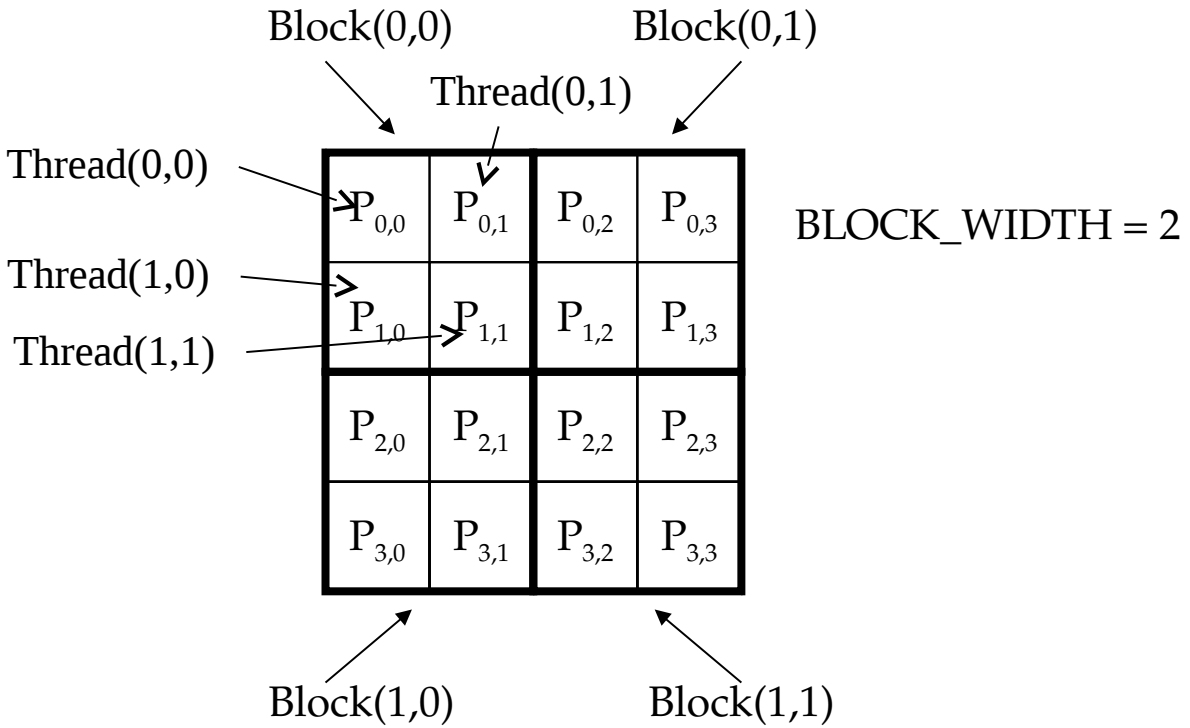
A Basic Matrix Multiplication Kernel

```
__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;
  }

}
```

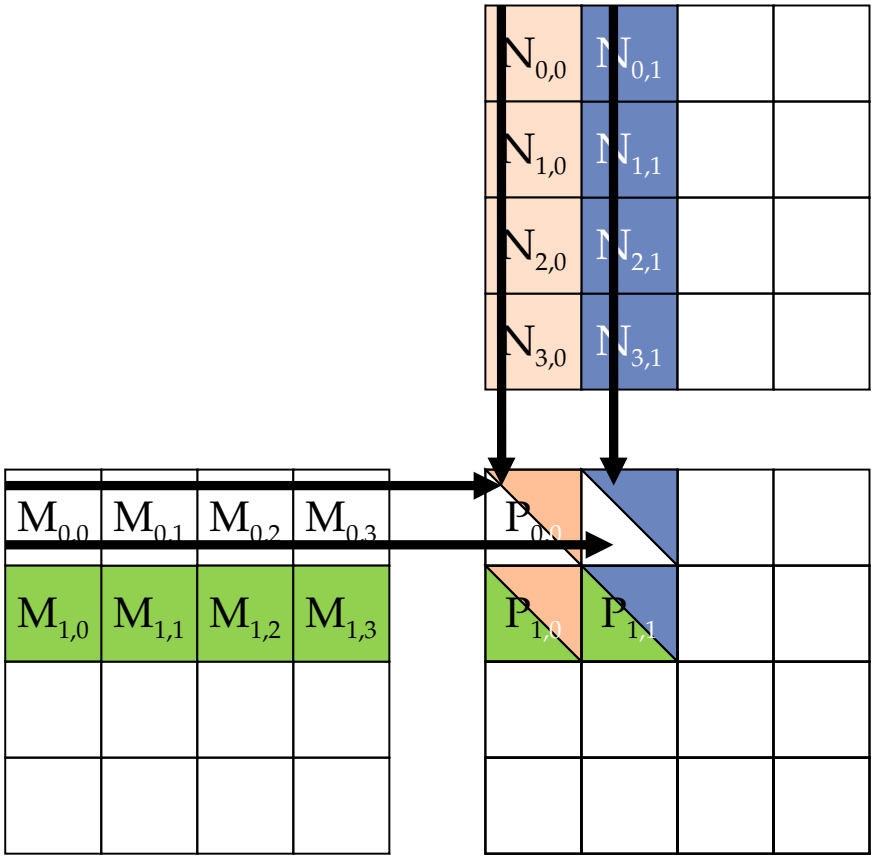
4x4 P: Thread to Data Mapping

P matrix divided into 4 parts

Each block (2x2 threads) calculates 1 part



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

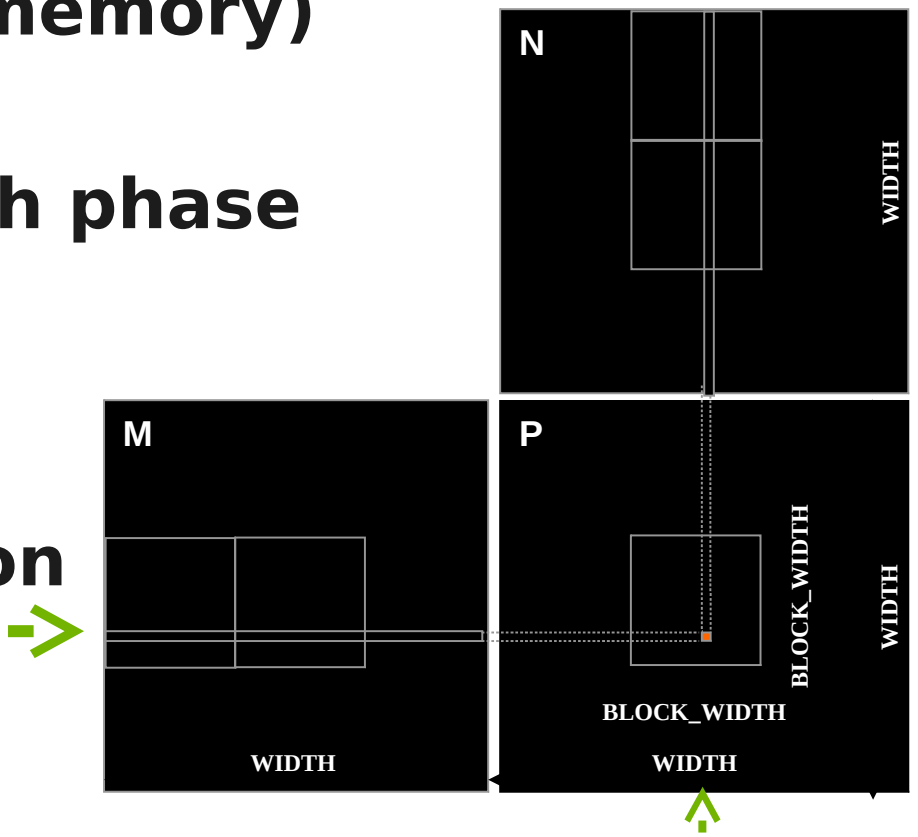


Tiled Matrix Multiplication

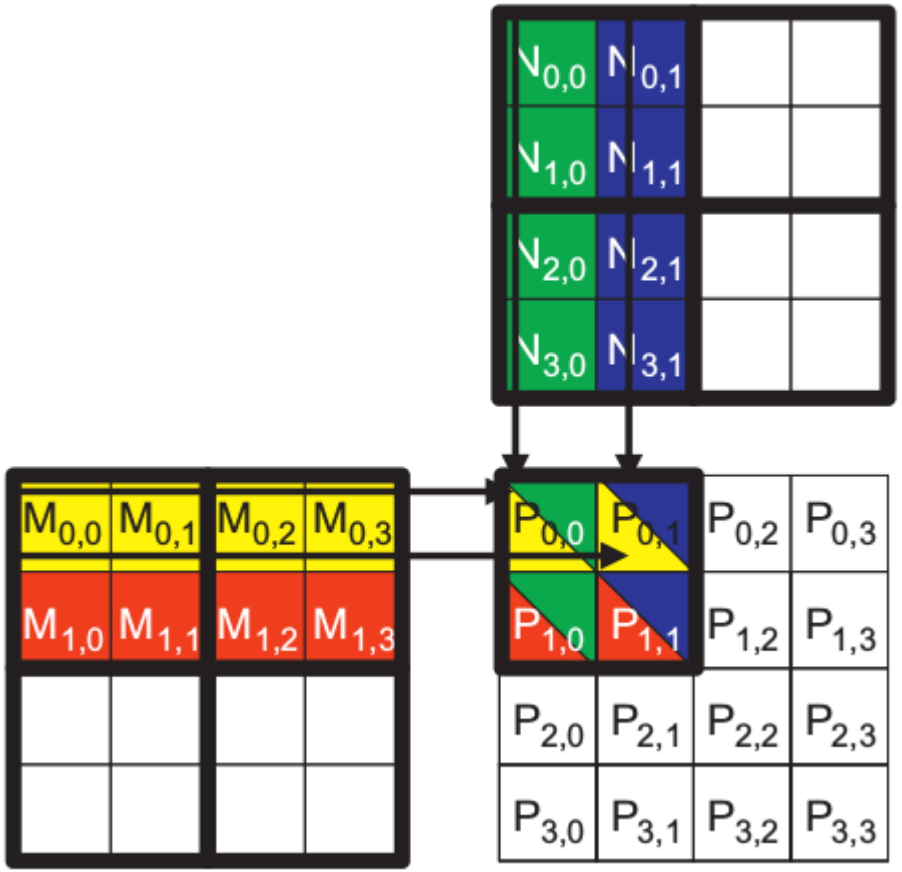
Break up the execution of each thread into phases (to utilize shared memory)

so that the 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

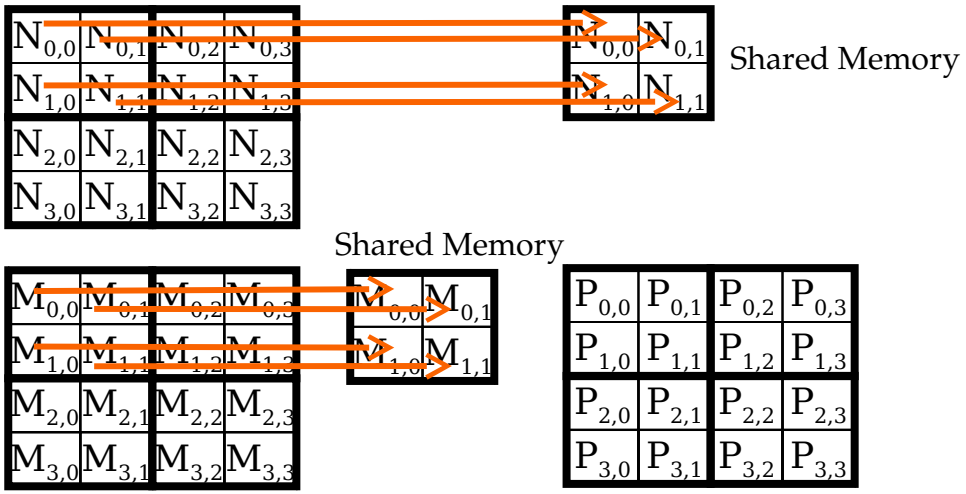


2x2 Tiles

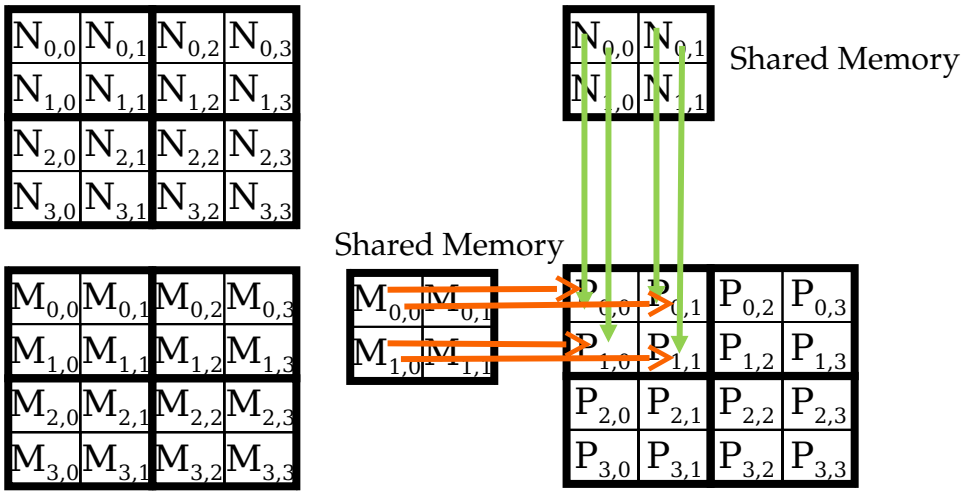

N0,0
N0,1
N1,0
N1,1
N2,0
N2,1
N3,0
N3,1
M0,0
M0,1
M0,2
M0,3
M1,0
M1,1
M1,2
M1,3
P0,0
P0,1
P0,2
P0,3
P1,0
P1,1
P1,2
P1,3
P2,0
P2,1
P2,2
P2,3
P3,0
P3,1
P3,2
P3,3

Phase 0 Load for Block (0,0)

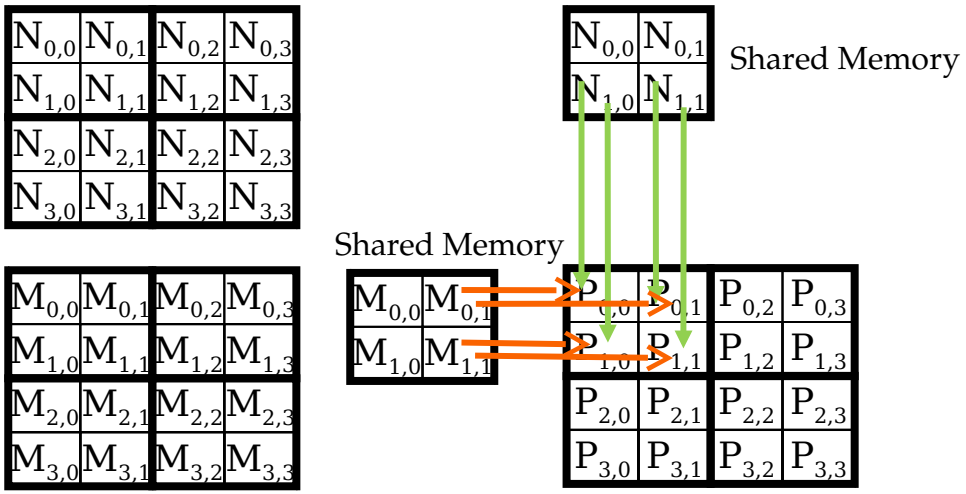
Each thread loads one M element and one N element in tiled code



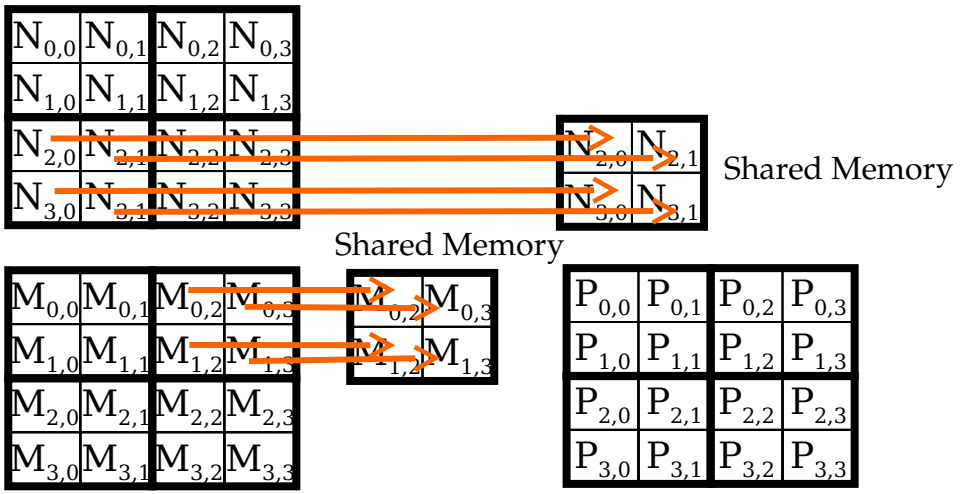
Phase 0 Use for Block (0,0) (iteration 0)



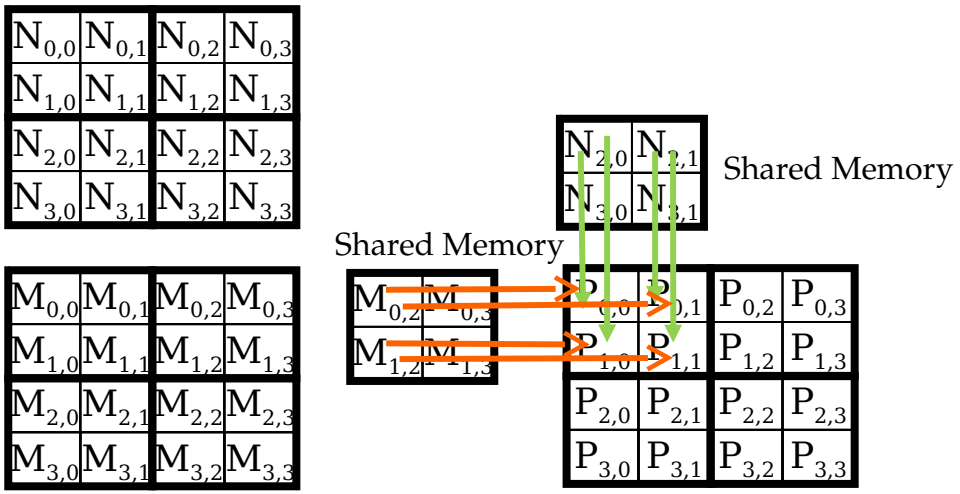
Phase 0 Use for Block (0,0) (iteration 1)



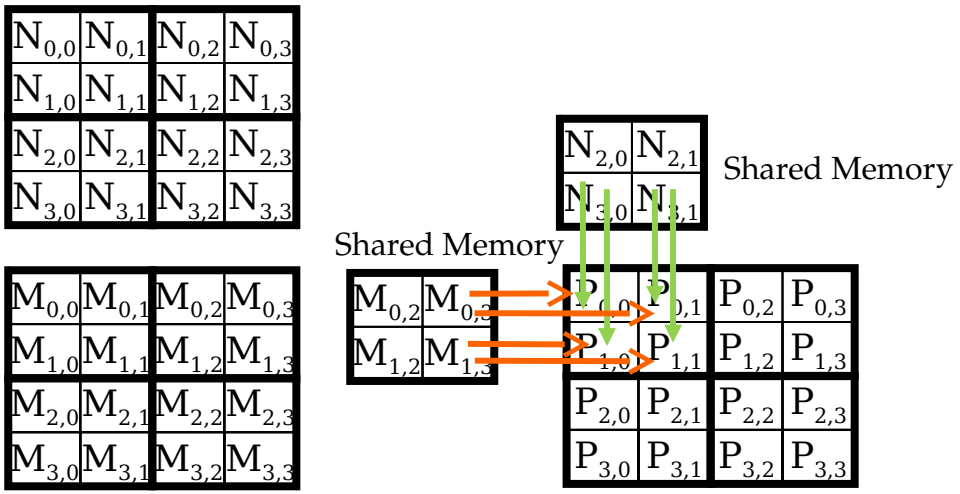
Phase 1 Load for Block (0,0)



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



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



Execution Phases

	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

Shared memory allows each value to be accessed by multiple threads

	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 →

Data in Shared Memory

Mds and Nds: shared memory arrays for M and N elements

They are reused to hold input values, allowing a much smaller shared memory to serve most of the accesses to global memory

Each phase focuses on a small subset of the input matrix elements: locality

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

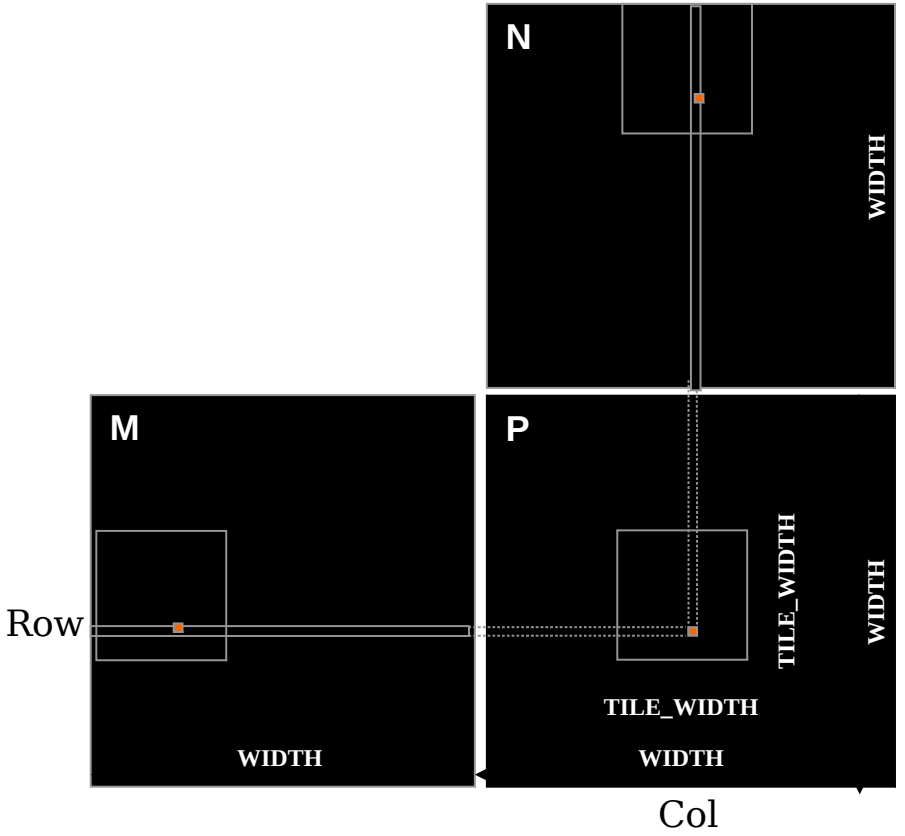
To ensure that all elements of a tile are loaded at the beginning of a phase

To ensure that all elements of a tile are consumed at the end of a phase

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]
```



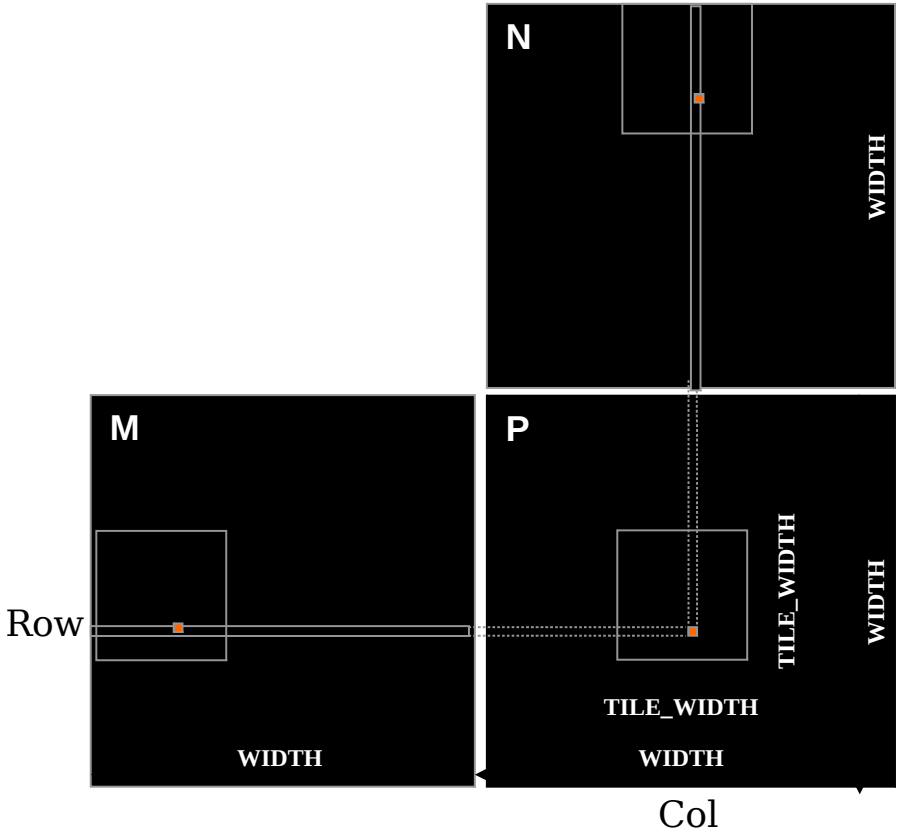
Loading Input Tile 0 of N (Phase 0)

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]
```

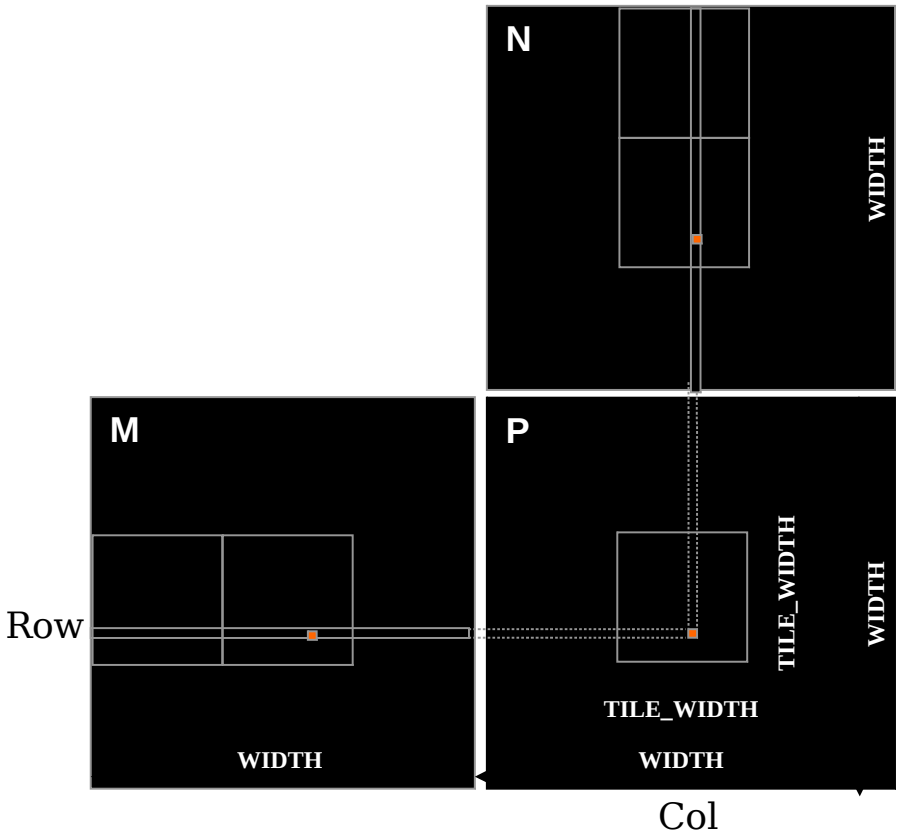


Loading Input Tile 1 of M (Phase 1)

2D indexing for accessing Tile 1:

M[Row][1*TILE_WIDTH + tx]

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

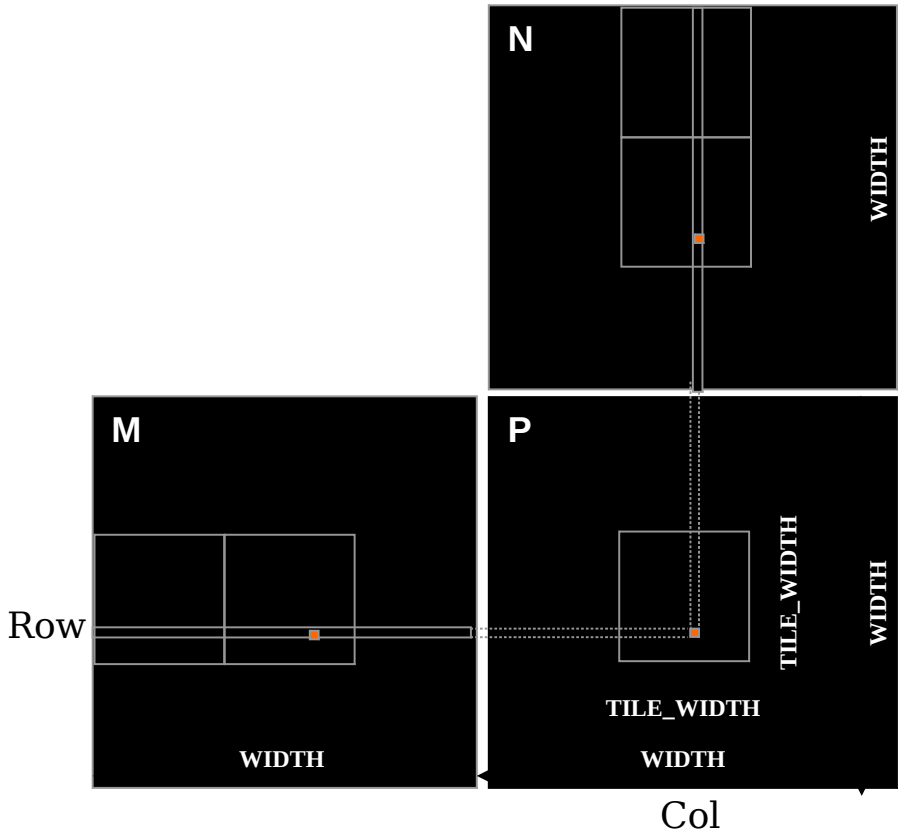


Loading Input Tile 1 of N (Phase 1)

2D indexing for accessing Tile 1:

M[Row][1*TILE_WIDTH + tx]

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



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

Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, Int Width)
{
  __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
  __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];

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

  int Row = by * blockDim.y + ty;
  int Col = bx * blockDim.x + tx;
  float Pvalue = 0;

 // Loop over the M and N tiles required to compute the P element
 for (int p = 0; p < Width/TILE_WIDTH; ++p) {
    // Collaborative loading of M and N tiles into shared memory
    ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
    ds_N[ty][tx] = N[(p*TILE_WIDTH+ty)*Width + Col];
    __syncthreads();

    for (int i = 0; i < TILE_WIDTH; ++i)Pvalue += ds_M[ty][i] * ds_N[i][tx];
    __synchthreads();
  }
  P[Row*Width+Col] = Pvalue;
}
```

Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, Int Width)
{
  __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
  __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];

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

  int Row = by * blockDim.y + ty;
  int Col = bx * blockDim.x + tx;
  float Pvalue = 0;

 // Loop over the M and N tiles required to compute the P element
 for (int p = 0; p < Width/TILE_WIDTH; ++p) {
    // Collaborative loading of M and N tiles into shared memory
    ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
    ds_N[ty][tx] = N[(p*TILE_WIDTH+ty)*Width + Col];
    __syncthreads();

    for (int i = 0; i < TILE_WIDTH; ++i)Pvalue += ds_M[ty][i] * ds_N[i][tx];
    __synchthreads();
  }
  P[Row*Width+Col] = Pvalue;
}
```

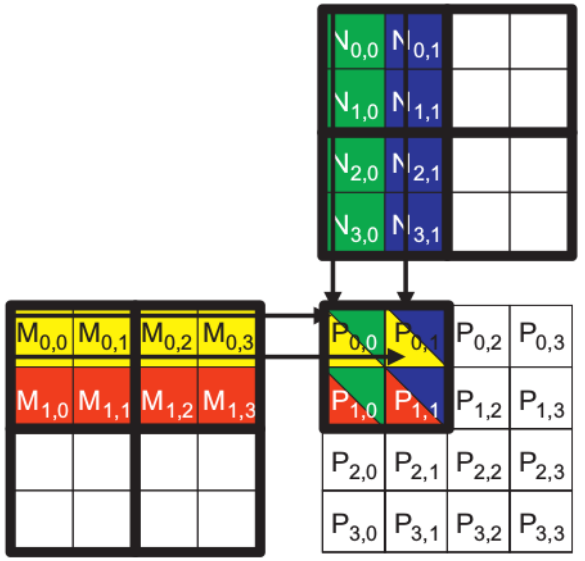
Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, Int Width)
{
  __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
  __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];

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

  int Row = by * blockDim.y + ty;
  int Col = bx * blockDim.x + tx;
  float Pvalue = 0;

 // Loop over the M and N tiles required to compute the P element
 for (int p = 0; p < Width/TILE_WIDTH; ++p) {
    // Collaborative loading of M and N tiles into shared memory
    ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
    ds_N[ty][tx] = N[(p*TILE_WIDTH+ty)*Width + Col];
    __syncthreads();

    for (int i = 0; i < TILE_WIDTH; ++i)Pvalue += ds_M[ty][i] * ds_N[i][tx];
    __synchthreads();
  }
  P[Row*Width+Col] = Pvalue;
}
```

Before-After

```
for (int k = 0; k < Width; ++k) {

      Pvalue += M[Row*Width+k]*N[k*Width+Col];

    }
```

```
for (int p = 0; p < Width/TILE_WIDTH; ++p) {
    ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
    ds_N[ty][tx] = N[(p*TILE_WIDTH+ty)*Width + Col];
    __syncthreads();

    for (int i = 0; i < TILE_WIDTH; ++i)
        Pvalue += ds_M[ty][i] * ds_N[i][tx];
    __synchthreads();
  }
```



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

(reduce global memory access by a factor of TILE_WIDTH)

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)

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

For an SM with 16KB shared memory

- Shared memory size is implementation dependent!
- For TILE_WIDTH = 16, 256 threads, each thread block uses 2*256*4B = 2KB shared memory per block, up to 8 thread blocks
  - This allows up to 8*512 = 4,096 pending loads. (2 per thread, 256 threads per block)
- For TILE_WIDTH = 32, 1024 threads, 2*1024*4B= 8KB shared memory usage per block, allowing 2 thread blocks active at the same time
  - However, in a GPU where the thread count is limited to 1536 threads per SM, the number of blocks per SM is reduced to one!

Each __syncthreads() can reduce the number of active threads for a block

- More thread blocks can be advantageous

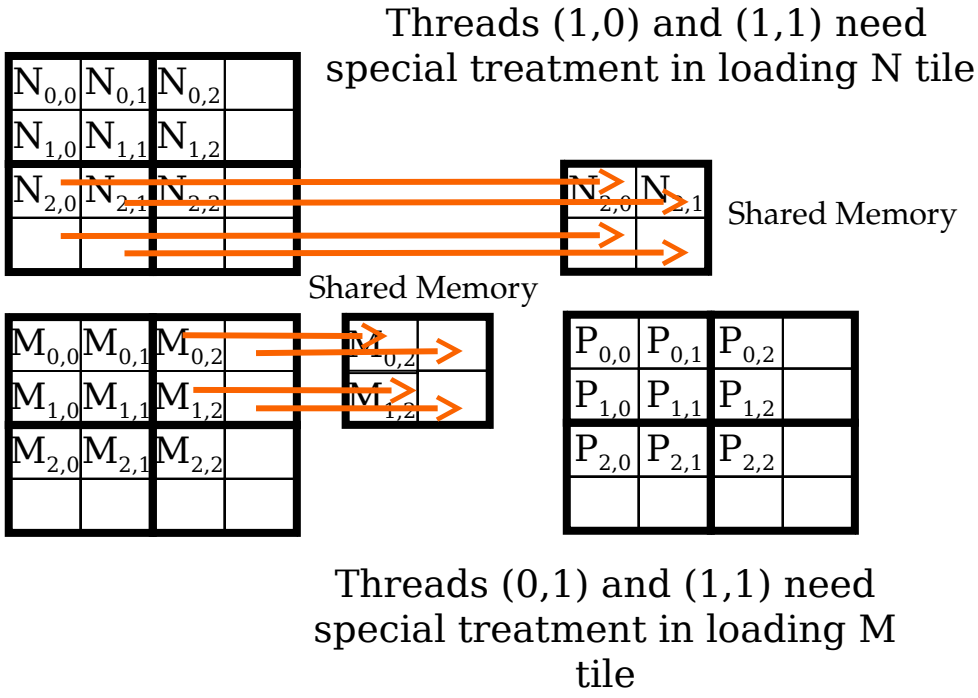
Handling Matrix of Arbitrary Size

Only square matrices whose dimensions (Width) are multiples of the tile width (TILE_WIDTH)

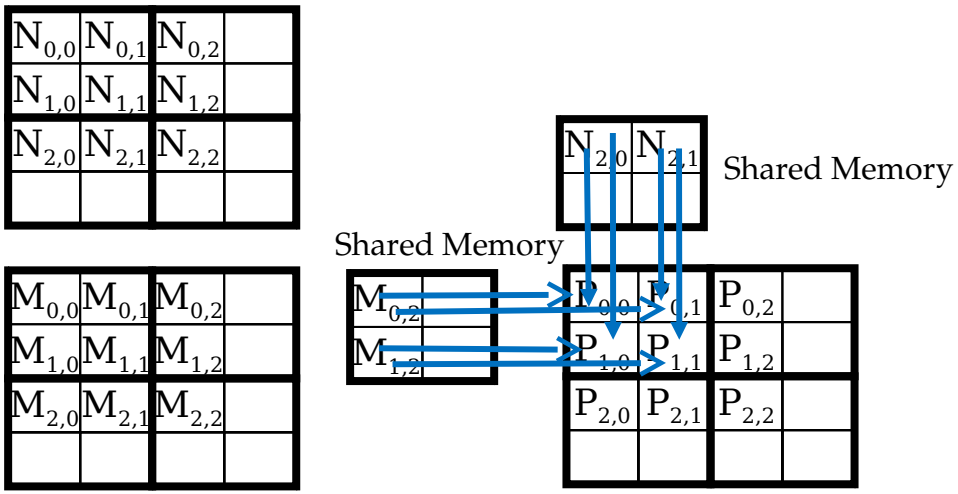
Real applications need to handle arbitrary sized matrices

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

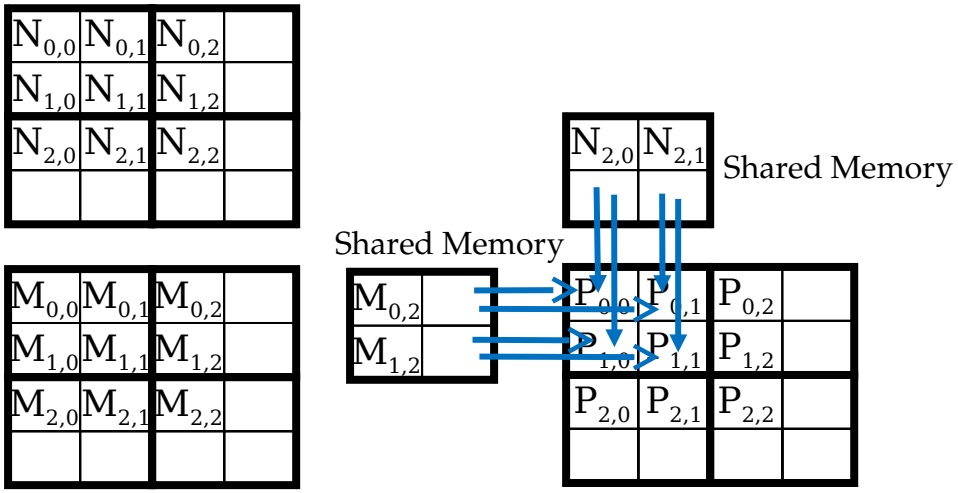
Phase 1 Loads for Block (0,0) for a 3x3 Example



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

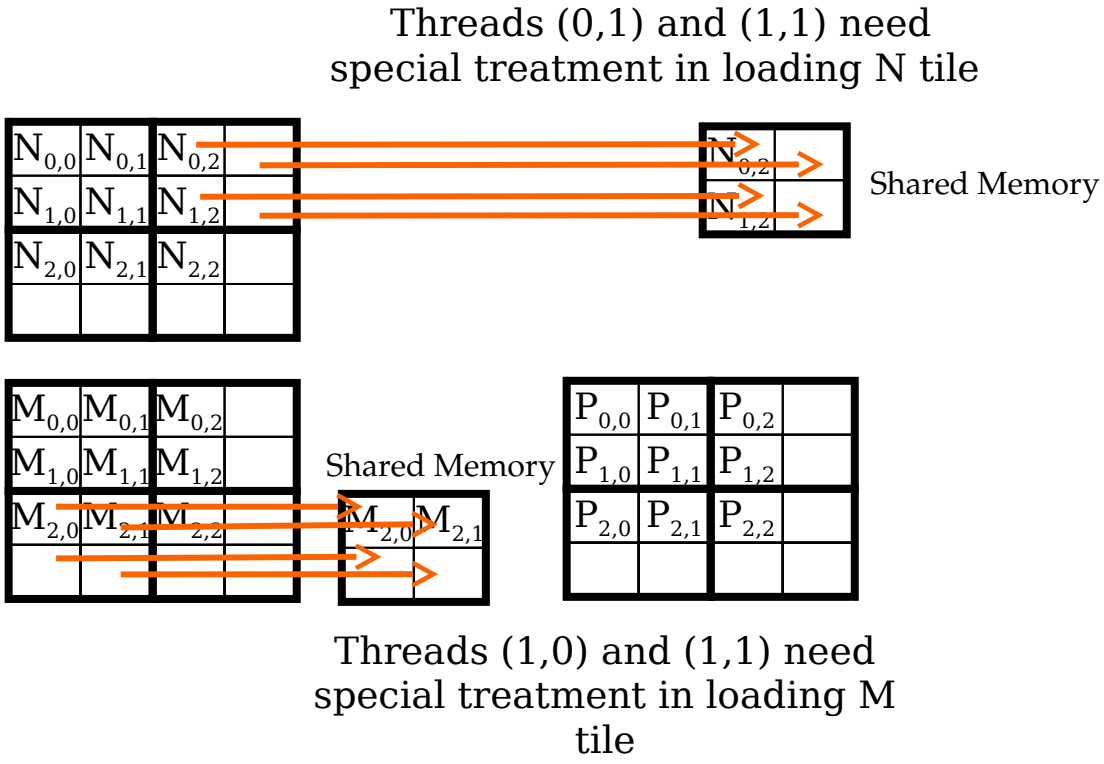


Phase 1 Use for Block (0,0) (iteration 1)



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



Some Cases

Threads that do not calculate valid P elements but still need to participate in loading the input tiles

Phase 0 of Block(1,1), Thread(1,0), assigned to calculate non-existent P[3,2] but need to participate in loading tile element N[1,2]

Threads that calculate valid P elements may attempt to load non-existing input elements when loading input tiles

Phase 1 of Block(0,0), Thread(1,0), assigned to calculate valid P[1,0] but attempts to load non-existing N[3,0]

A Simple Solution

When a thread is to load any input element, test if it is in the valid index range

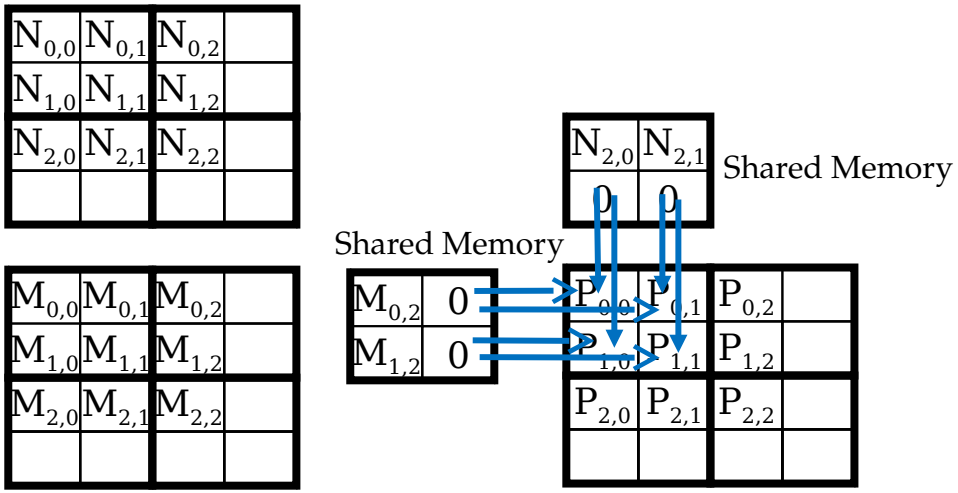
- If valid, proceed to load
- Else, do not load, just write a 0

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

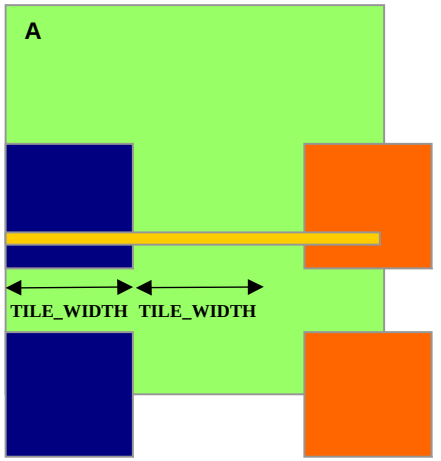
The condition tested for loading input elements is different from the test for calculating output P element

- 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]

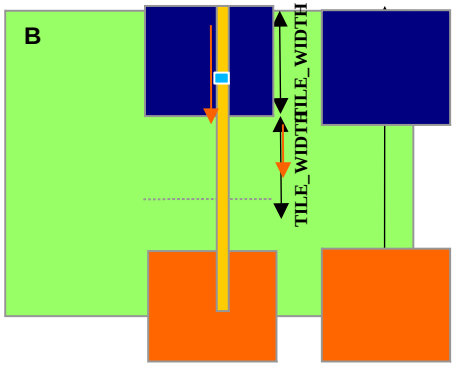
N[(p*TILE_WIDTH+ty)*Width+ Col]

Need to test

(p*TILE_WIDTH+ty < Width) && (Col< Width)

If true, load N element

Else , load 0



Loading Elements – with boundary check

```
8     for (int p = 0; p < (Width-1) / TILE_WIDTH + 1; ++p) {

++        if(Row < Width && p * TILE_WIDTH+tx < Width) {
9               ds_M[ty][tx] = M[Row * Width + p * TILE_WIDTH + tx];
++        } else {
++              ds_M[ty][tx] = 0.0;
++        }
++        if (p*TILE_WIDTH+ty < Width && Col < Width) {
10              ds_N[ty][tx] = N[(p*TILE_WIDTH + ty) * Width + Col];
++        } else {
++              ds_N[ty][tx] = 0.0;
++        }
11       __syncthreads();
```

Inner Product – Before and After

```
++    if(Row < Width && Col < Width) {
12        for (int i = 0; i < TILE_WIDTH; ++i) {
13                Pvalue += ds_M[ty][i] * ds_N[i][tx];
          }
14     __syncthreads();
15   } /* end of outer for loop */
++   if (Row < Width && Col < Width)
16         P[Row*Width + Col] = Pvalue;
  } /* end of kernel */
```