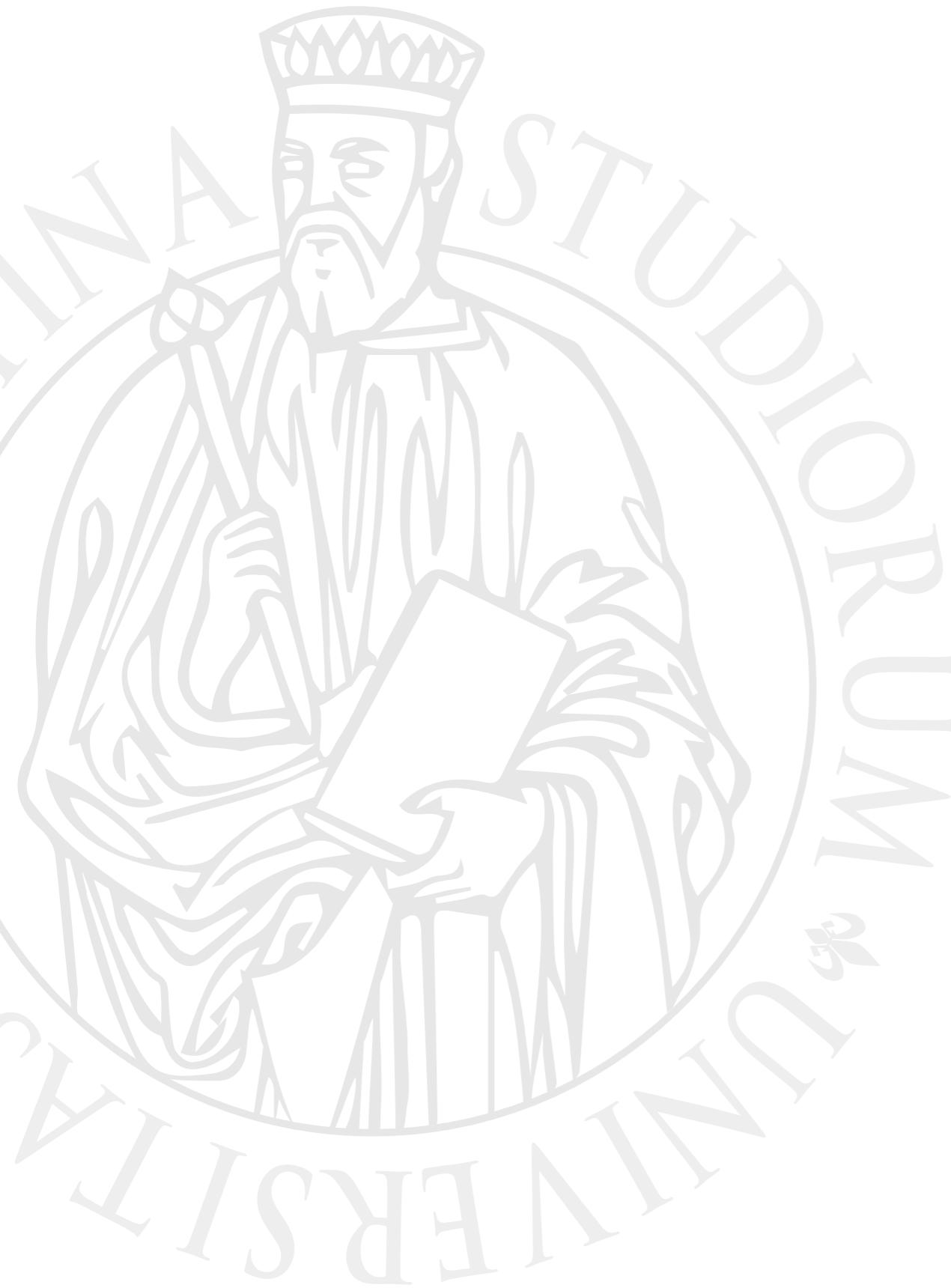




UNIVERSITÀ
DEGLI STUDI
FIRENZE

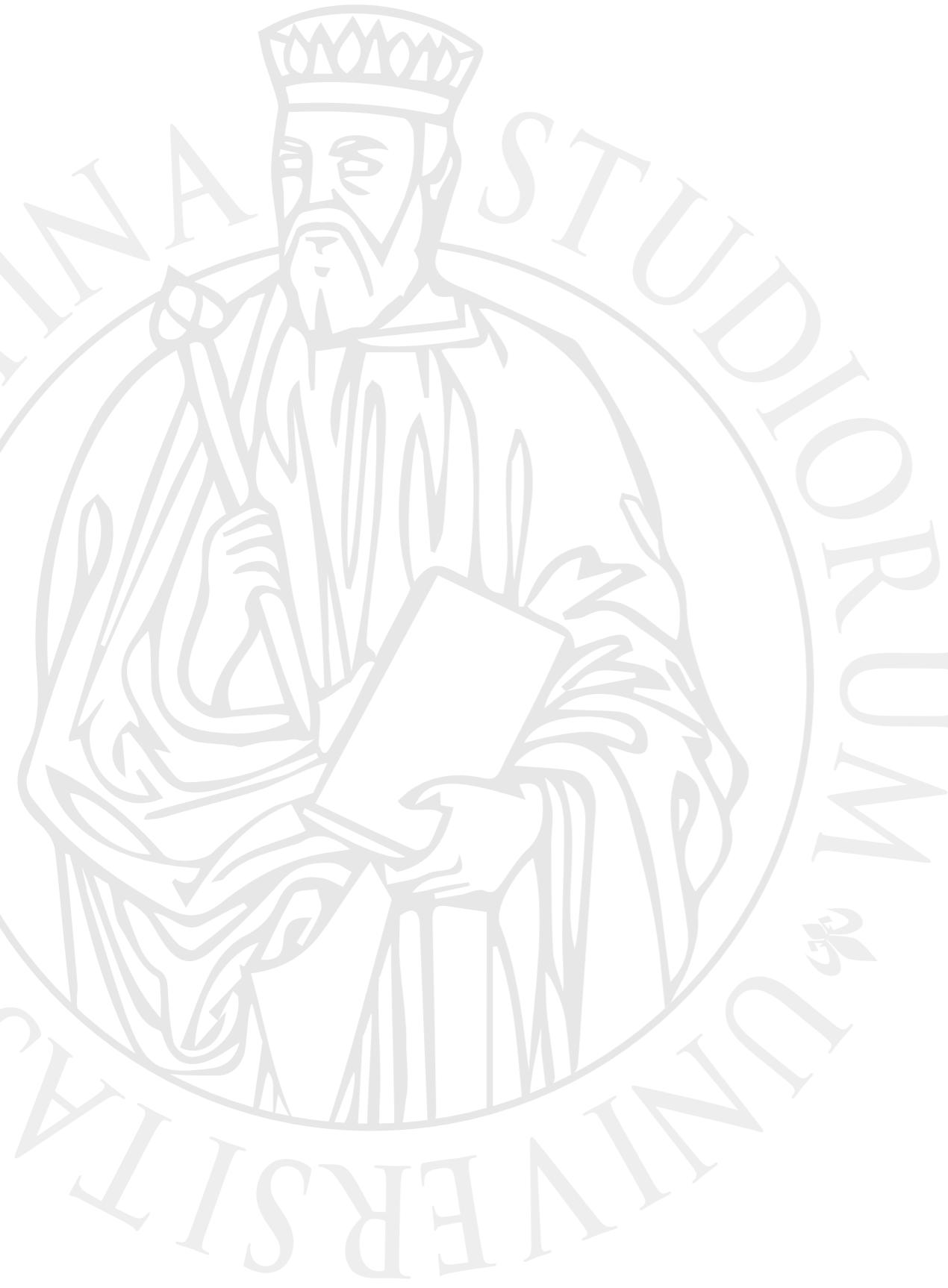


Parallel Programming

Prof. Marco Bertini



UNIVERSITÀ
DEGLI STUDI
FIRENZE



Data parallelism: GPU computing



UNIVERSITÀ
DEGLI STUDI
FIRENZE

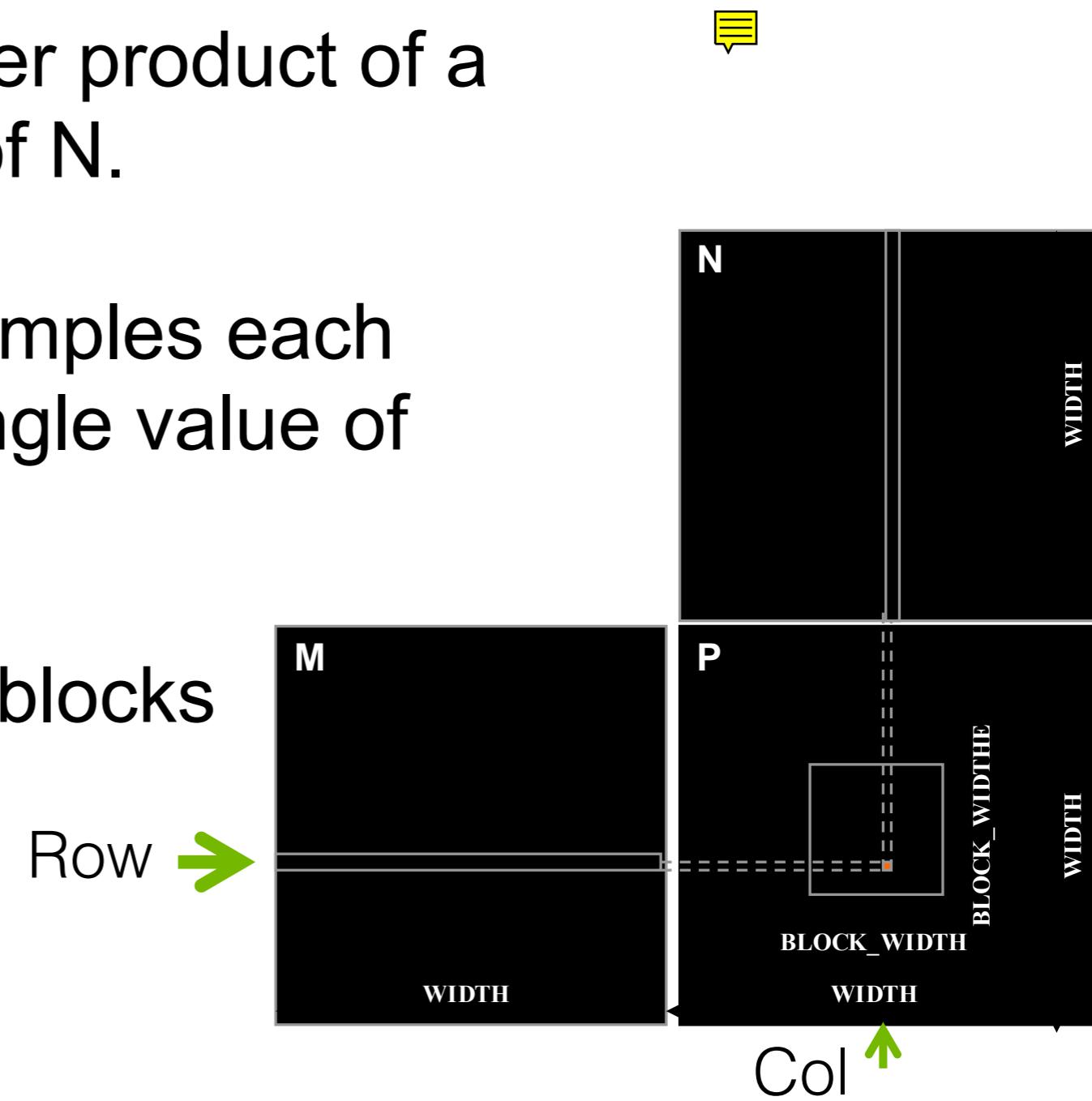


CUDA: **efficient** **programming**



Basic problem: matrix multiplication

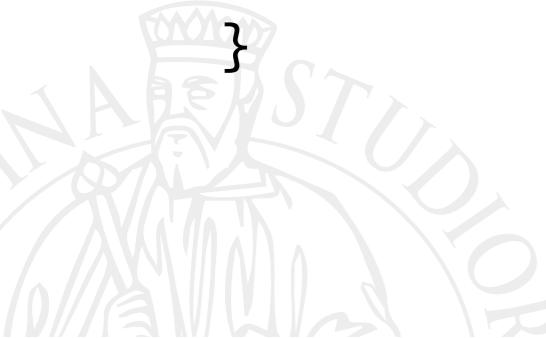
- When performing a matrix multiplication, each element of the output matrix P is an inner product of a row of M and a column of N.
- Similarly to previous examples each thread will compute a single value of the output matrix
- We organize threads as blocks



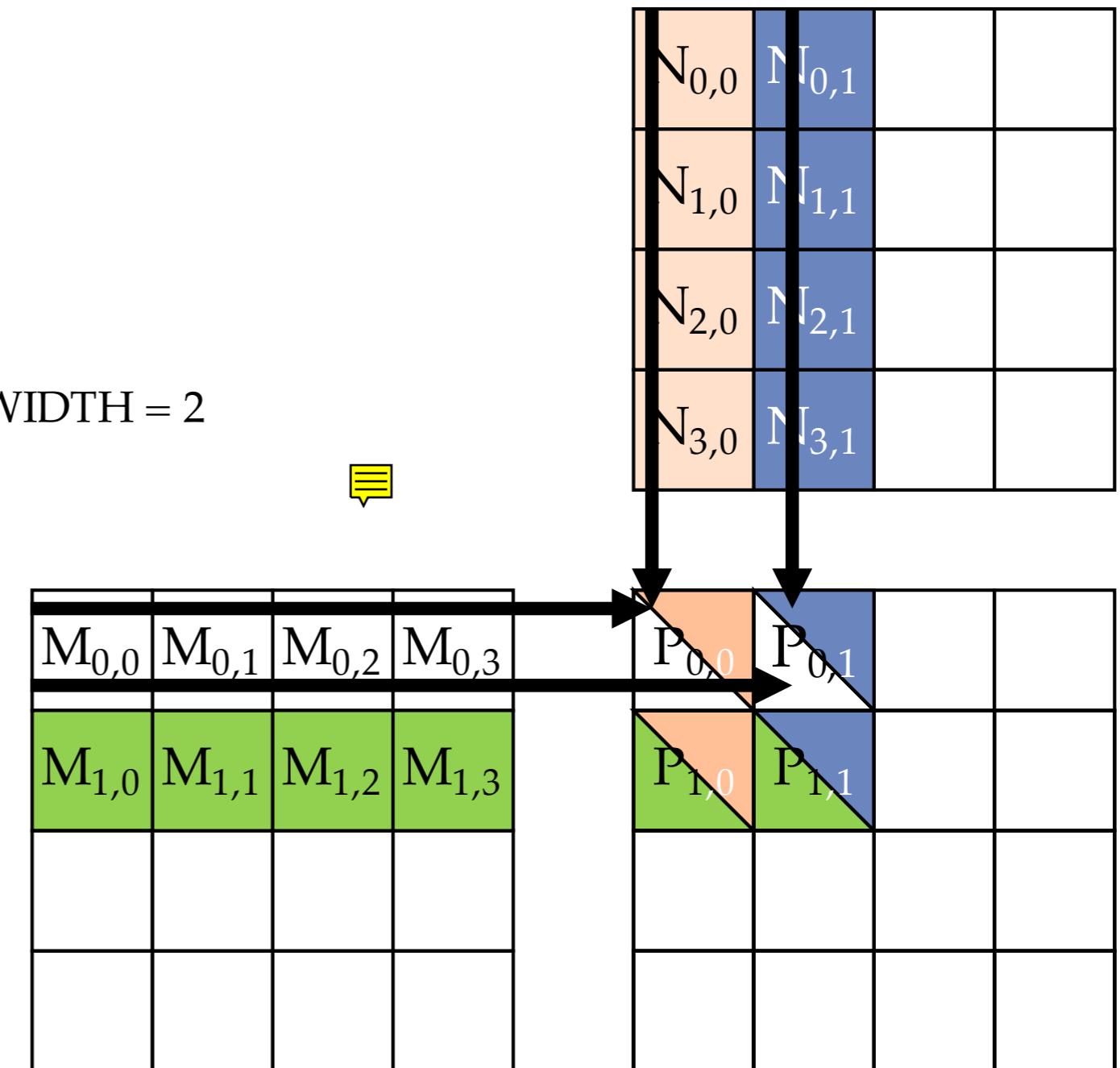
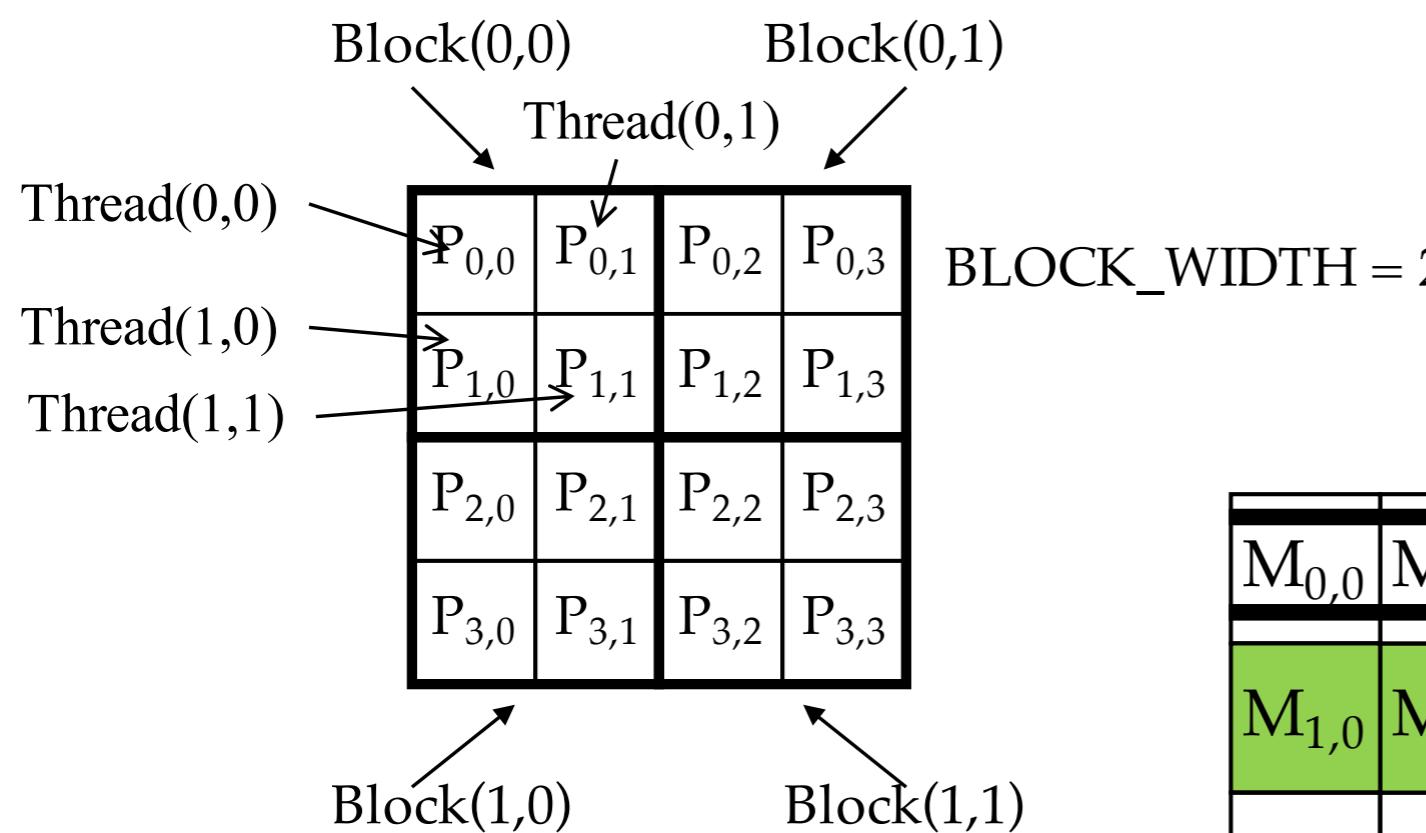


A solution

```
__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;  
    } 
```

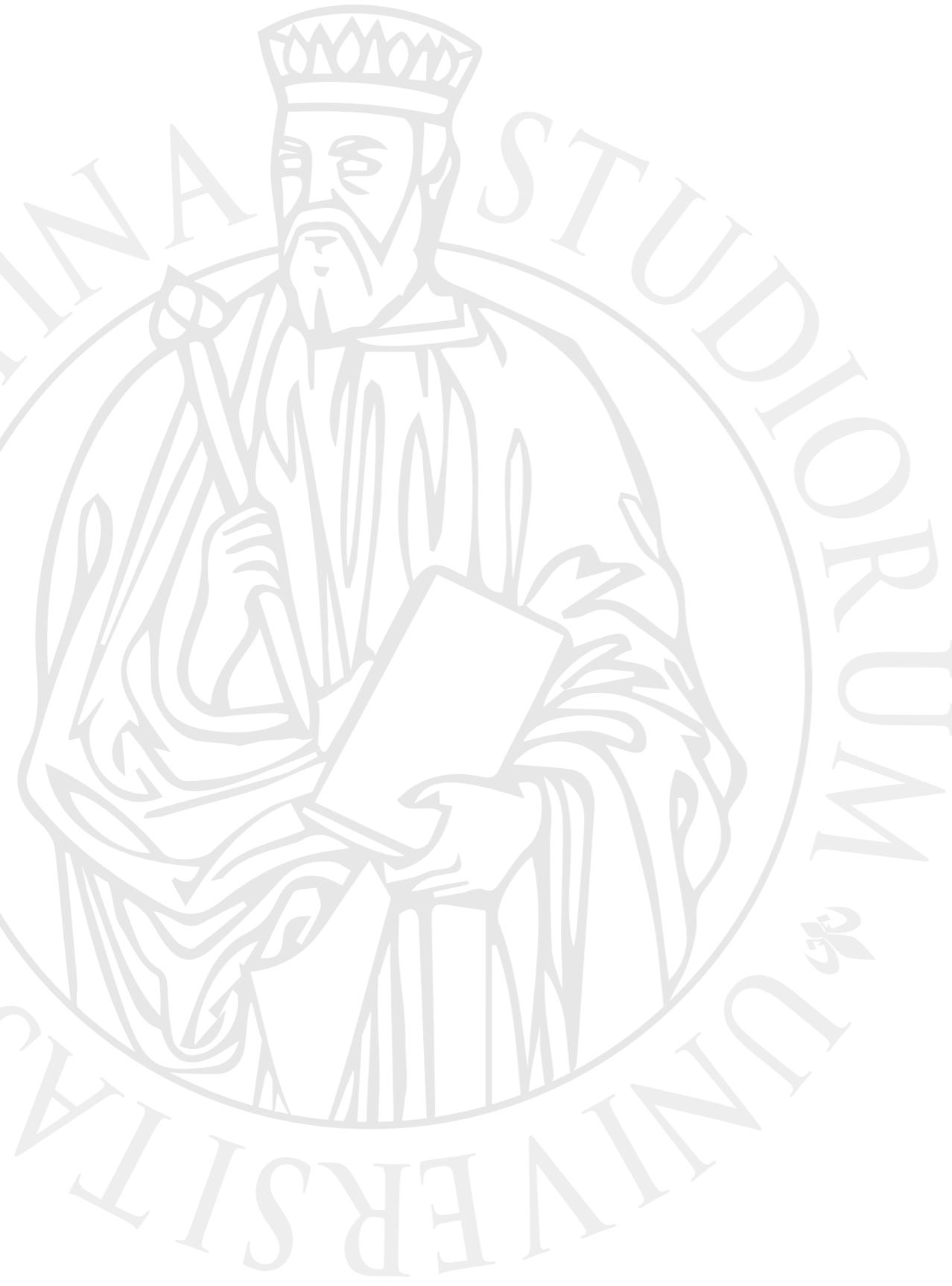


Toy example visualization





UNIVERSITÀ
DEGLI STUDI
FIRENZE



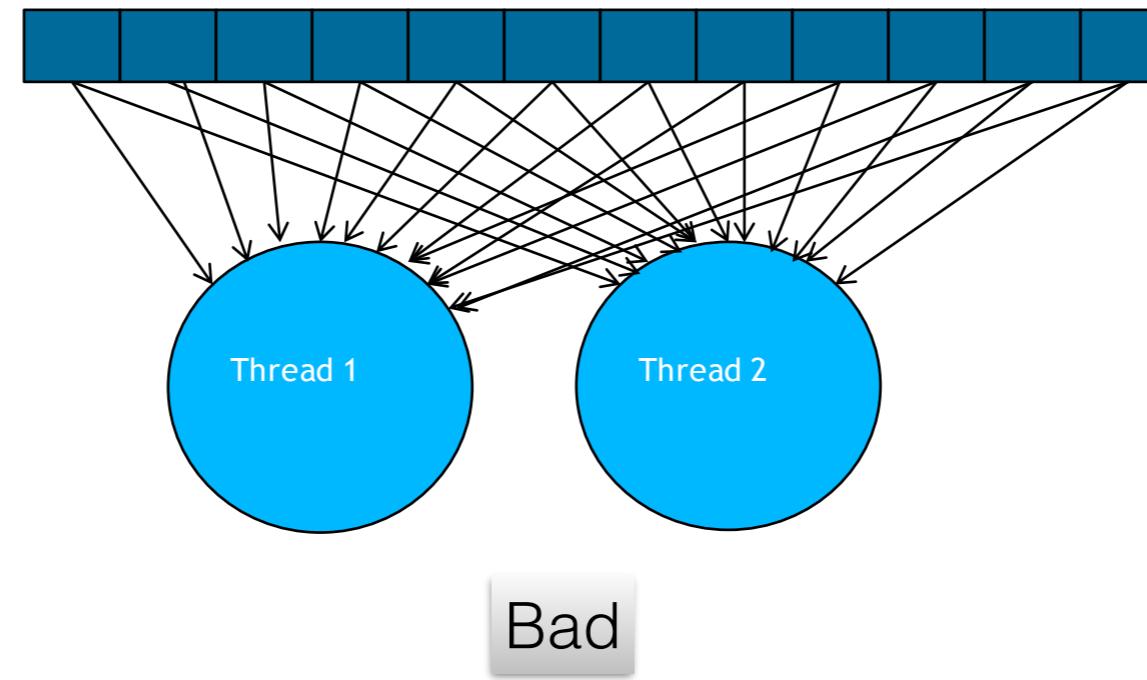
**Reduce
memory traffic:
tiling**

Tiling and CUDA memories

- Remind the tradeoffs of CUDA memories:
 - the global memory is large but slow;
 - the shared memory is **small** but fast.
- A common strategy is to partition the data into subsets called **tiles** so that each tile fits into the shared memory.
An important criterion is that the kernel computation on these tiles can be done **independently** of each other.
 - Note that **not all data structures** can be partitioned into tiles given an arbitrary kernel function.

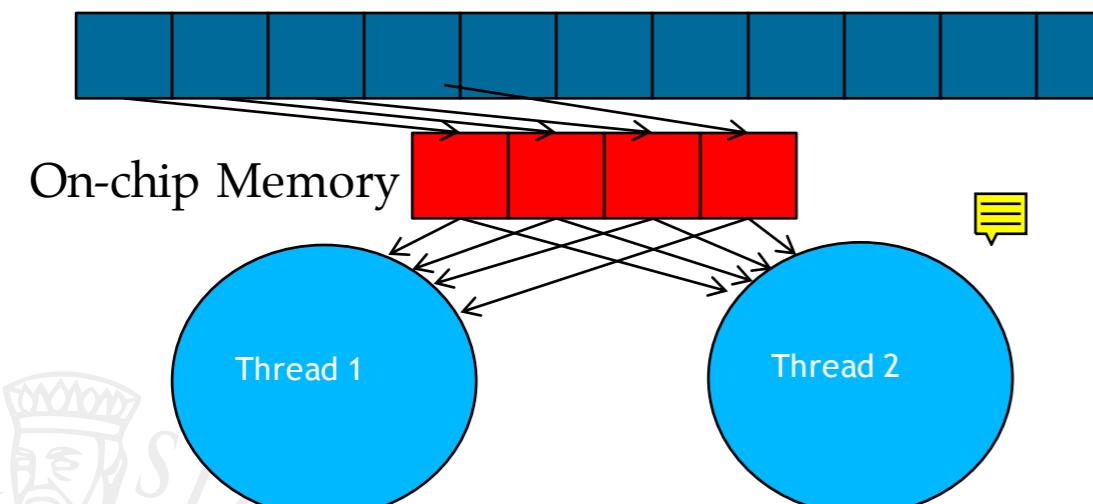
Global vs. shared memory access

Global Memory



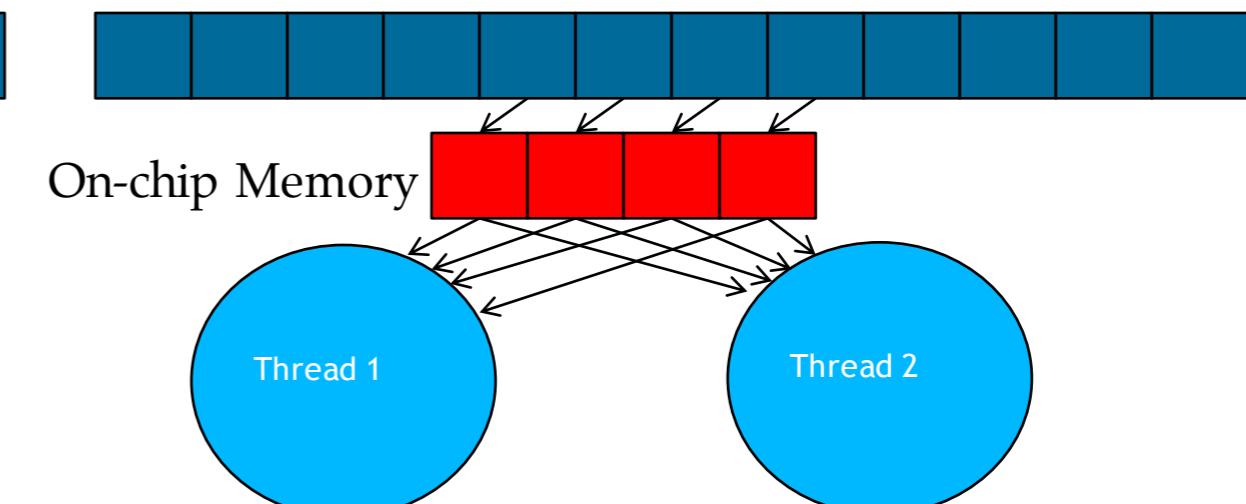
Bad

Global Memory

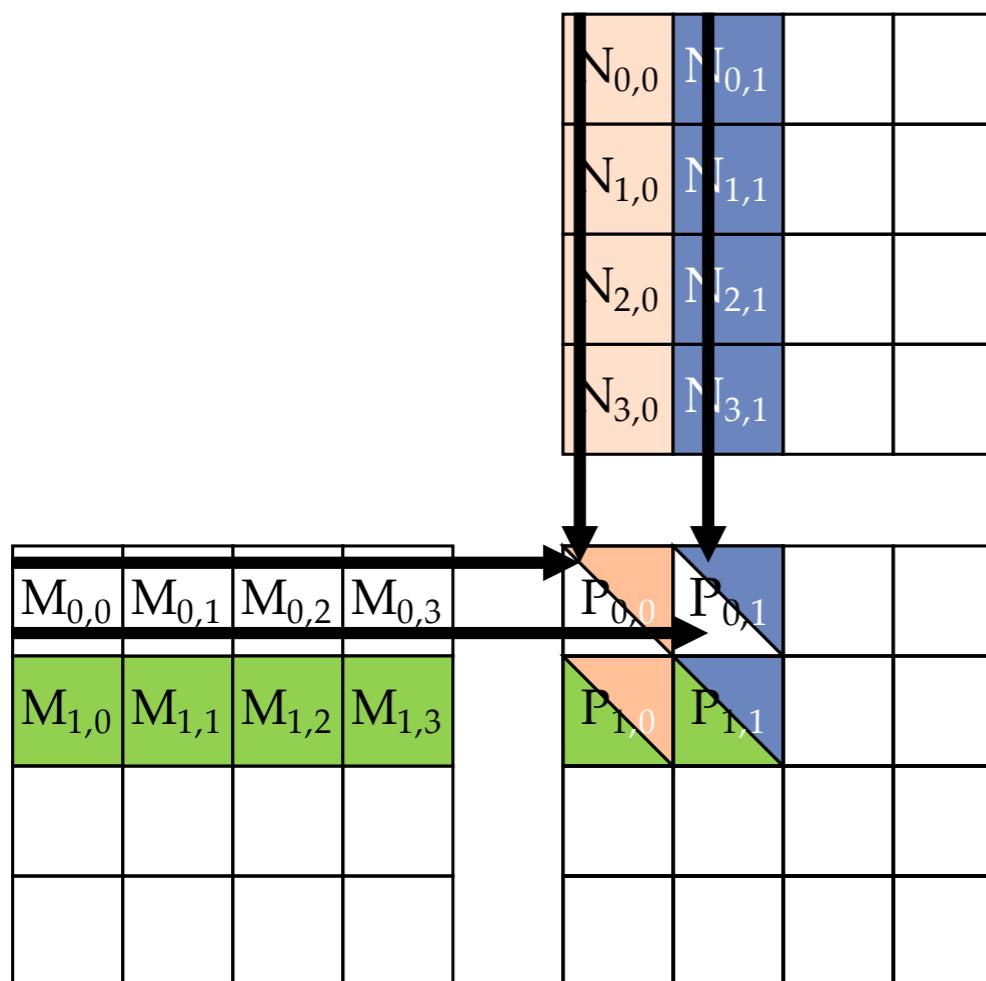


Good

Global Memory



Memory access in matrix multiplication



Access order

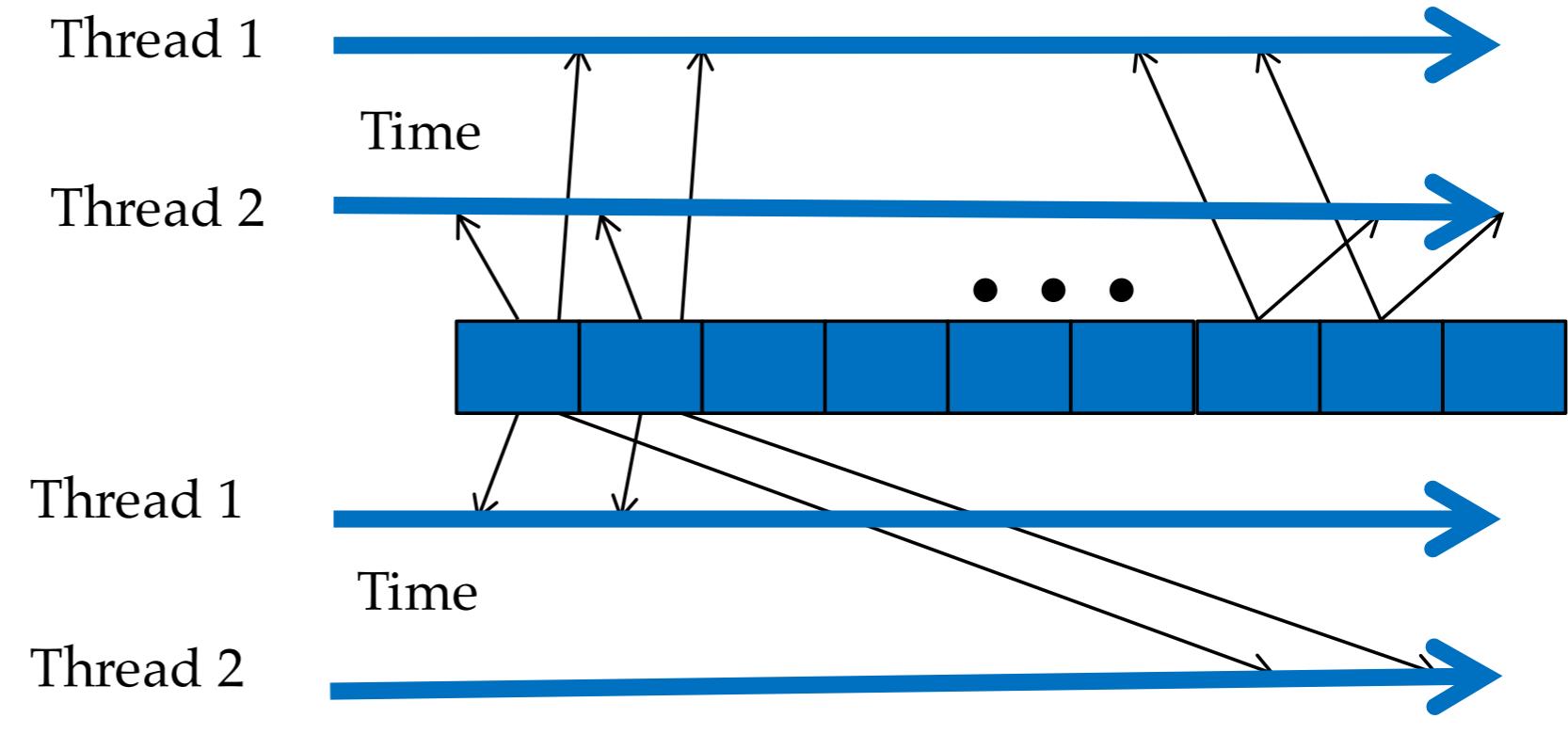
thread _{0,0}	$M_{0,0} * N_{0,0}$	$M_{0,1} * N_{1,0}$	$M_{0,2} * N_{2,0}$	$M_{0,3} * N_{3,0}$
thread _{0,1}	$M_{0,0} * N_{0,1}$	$M_{0,1} * N_{1,1}$	$M_{0,2} * N_{2,1}$	$M_{0,3} * N_{3,1}$
thread _{1,0}	$M_{1,0} * N_{0,0}$	$M_{1,1} * N_{1,0}$	$M_{1,2} * N_{2,0}$	$M_{1,3} * N_{3,0}$
thread _{1,1}	$M_{1,0} * N_{0,1}$	$M_{1,1} * N_{1,1}$	$M_{1,2} * N_{2,1}$	$M_{1,3} * N_{3,1}$



- Each thread accesses four elements of M and four elements of N during its execution. Among the four threads highlighted, there is a significant overlap in the M and N elements they access. For example, thread_{0,0} and thread_{0,1} both access M_{0,0} as well as the rest of row 0 of M.
- If we can somehow manage to have thread_{0,0} and thread_{0,1} to collaborate so that these M elements are only loaded from global memory once, we can reduce the total number of accesses to the global memory by half.
- In fact, we can see that every M and N element is accessed exactly twice during the execution of block_{0,0}.
- Larger block width leads to greater global memory traffic reduction.

Memory access patterns

Good: when
threads have similar
access timing



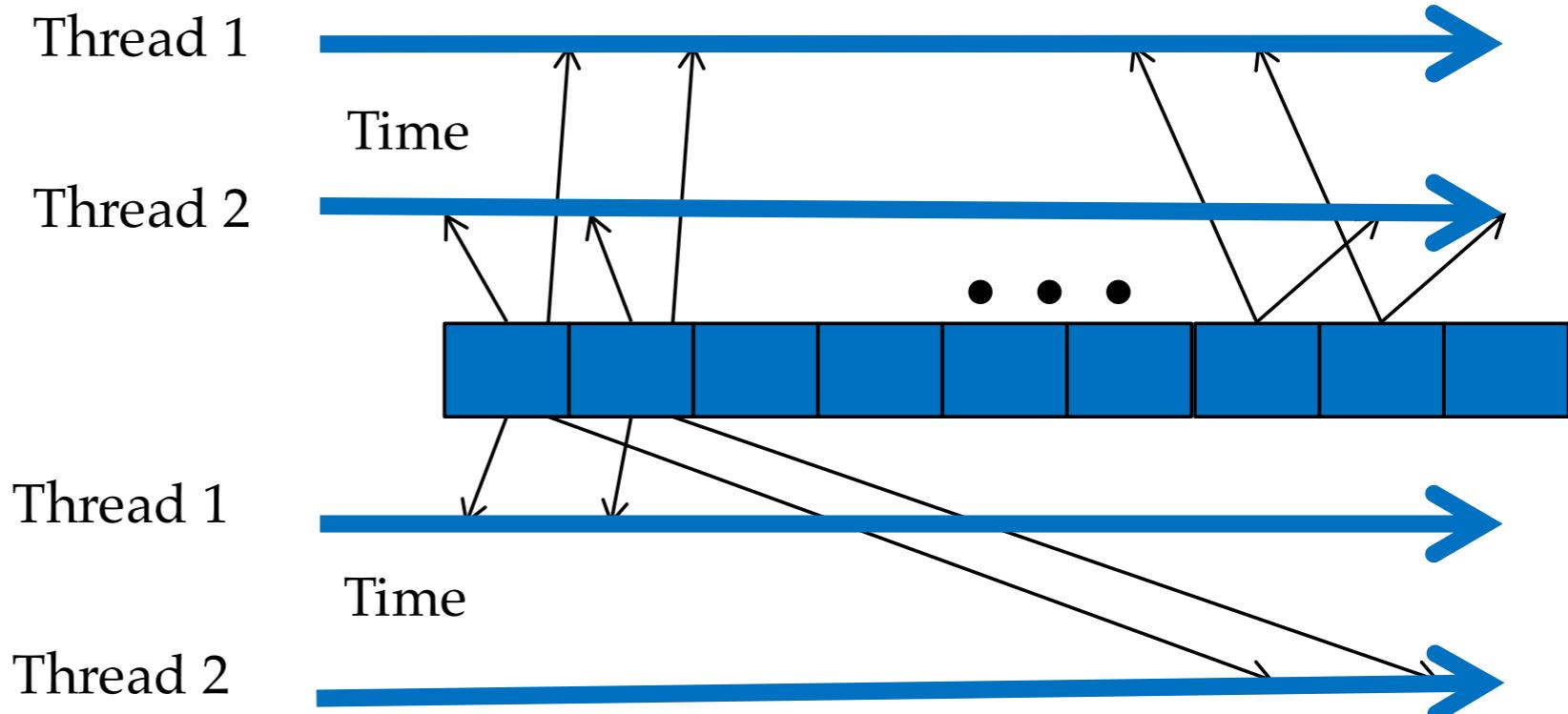
Bad: when threads
have very different
timing

Memory access patterns

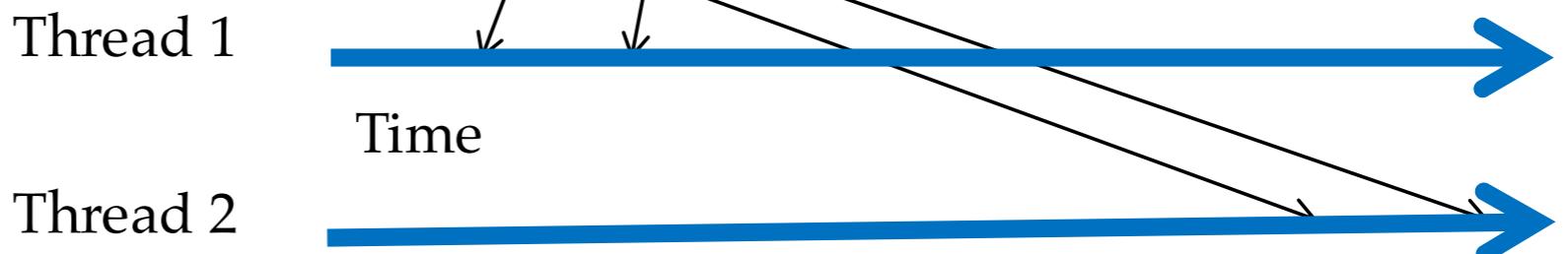
With tiling we are going to transform a program to localize memory locations accessed among threads and timing of their access.

Long access sequences of each thread are broken into phases, using barriers to synchronize access times to each section by the threads.

Good: when
threads have similar
access timing



Bad: when threads
have very different
timing

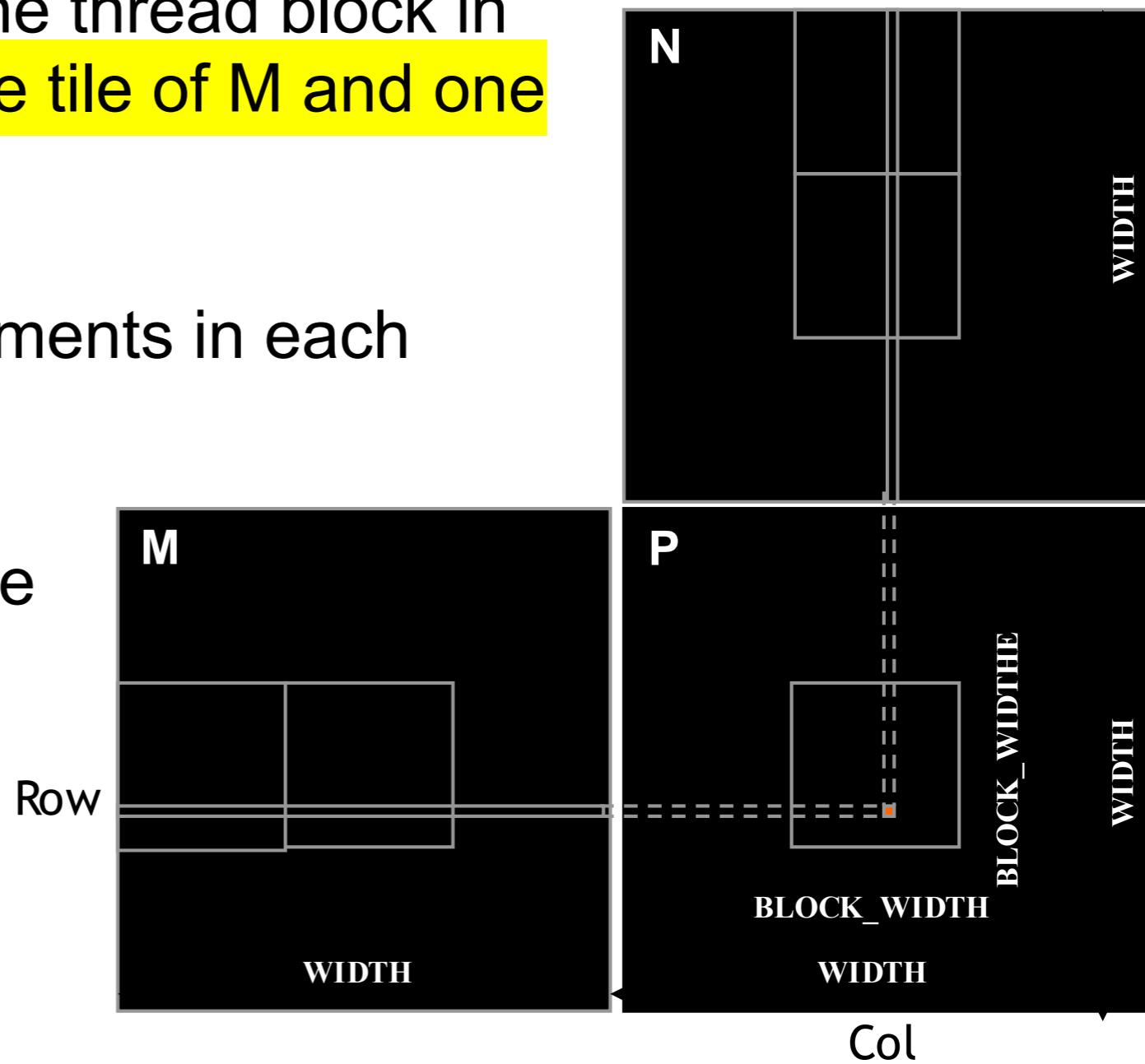


Tiling: how ?

- 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

Tiling matrix multiplication

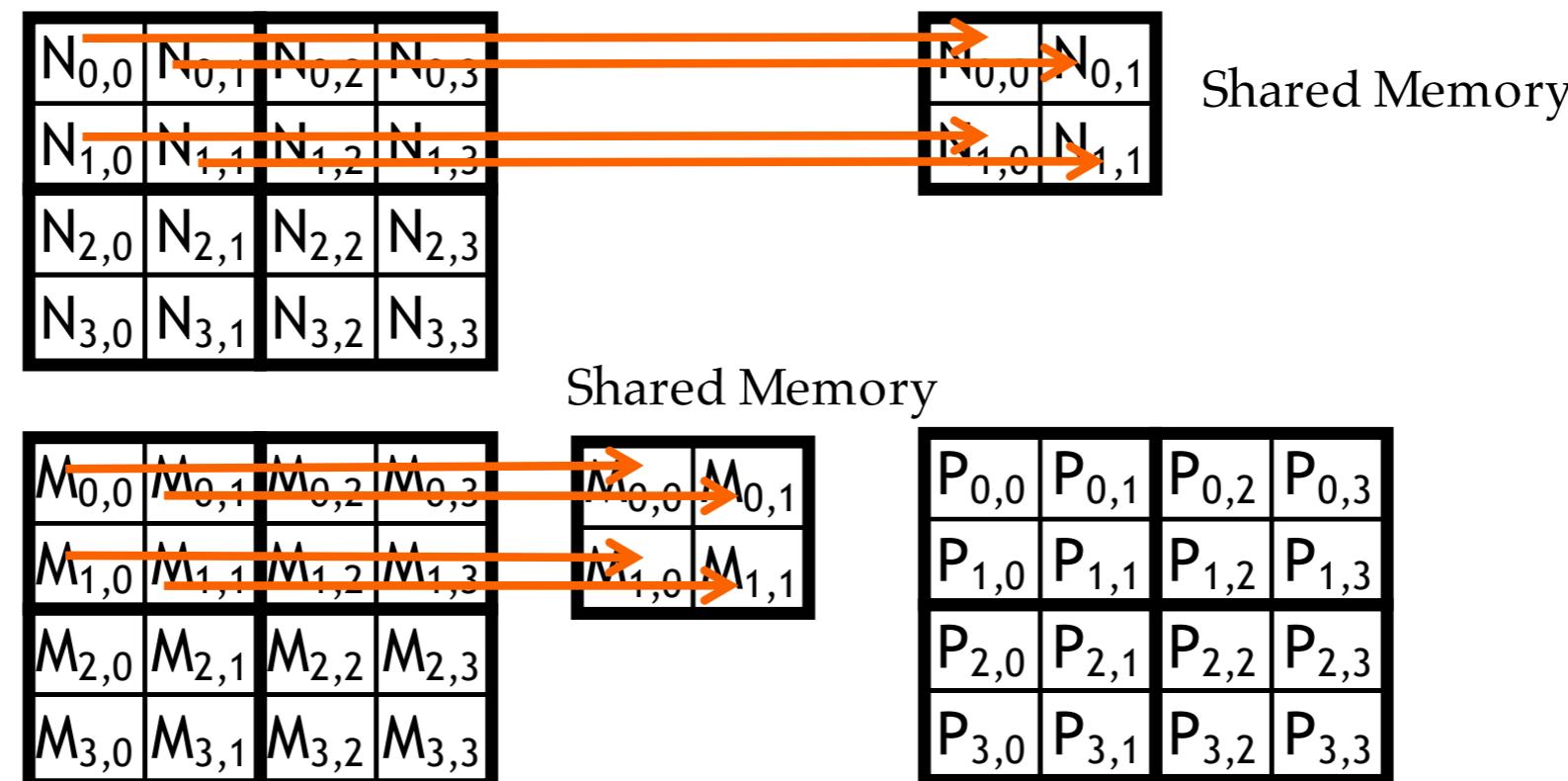
- Break up the execution of each thread into phases
- so that the data accesses by the thread block in each phase are focused **on one tile of M and one tile of N**
- The tile is of `BLOCK_SIZE` elements in each dimension
- All threads in a block participate
- Each thread loads one M element and one N element in tiled code





Tiling phases

Phase 0 Load for Block (0,0)



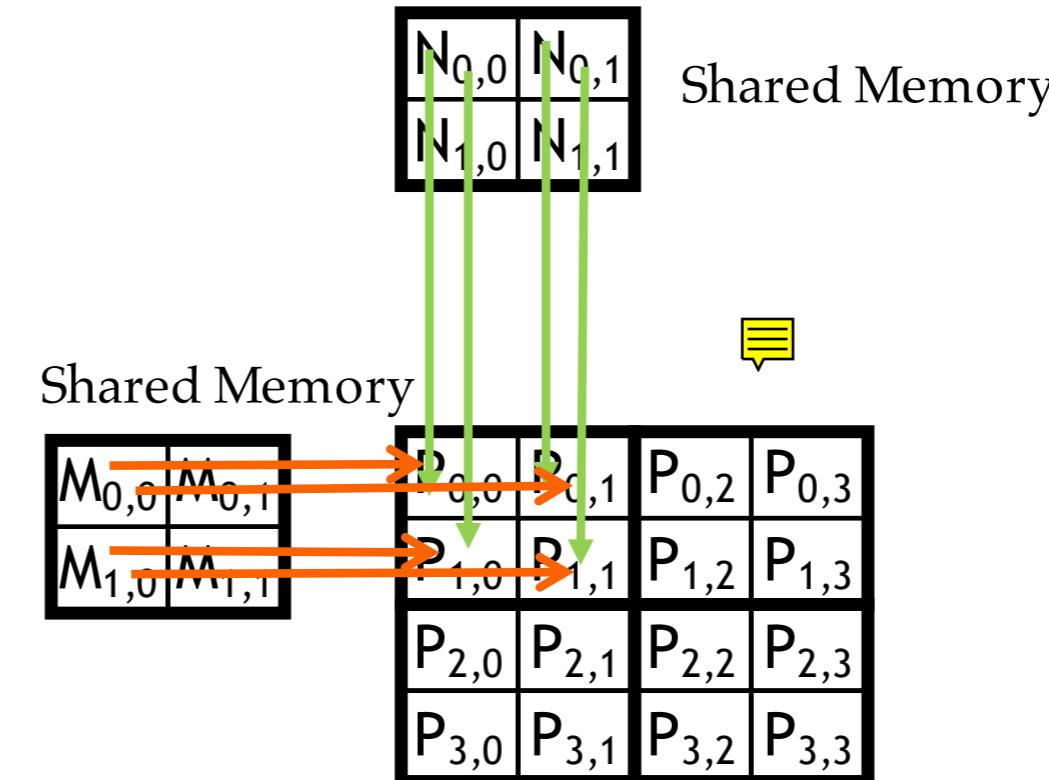


Tiling phases

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



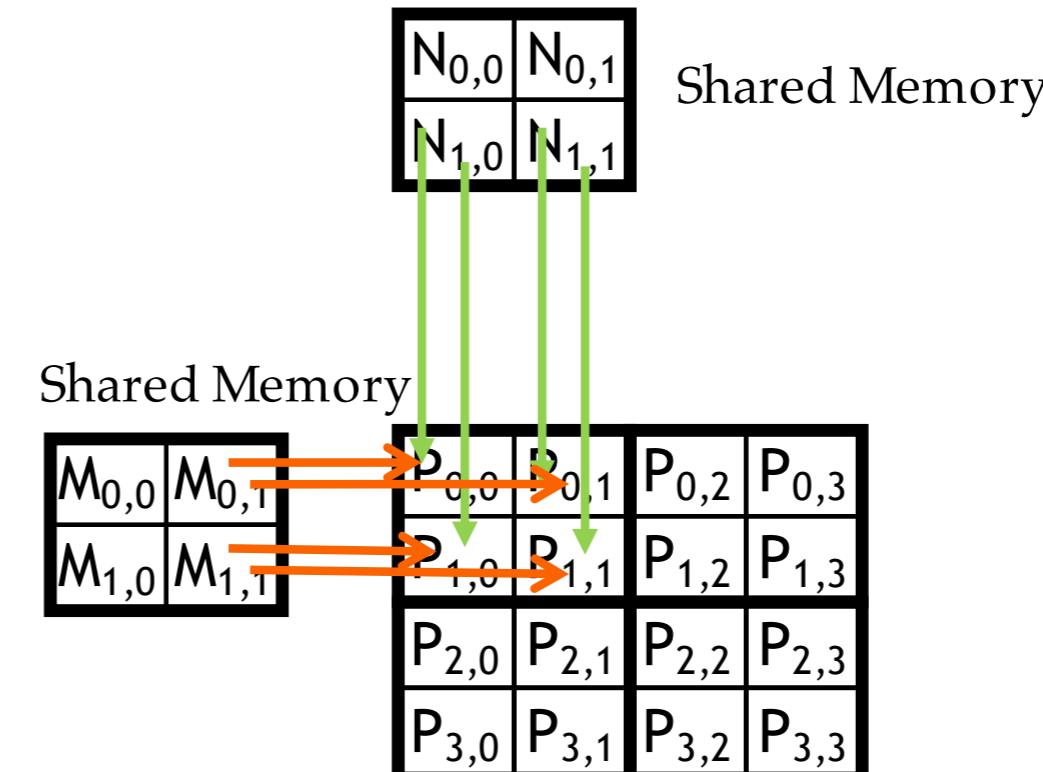


Tiling phases

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

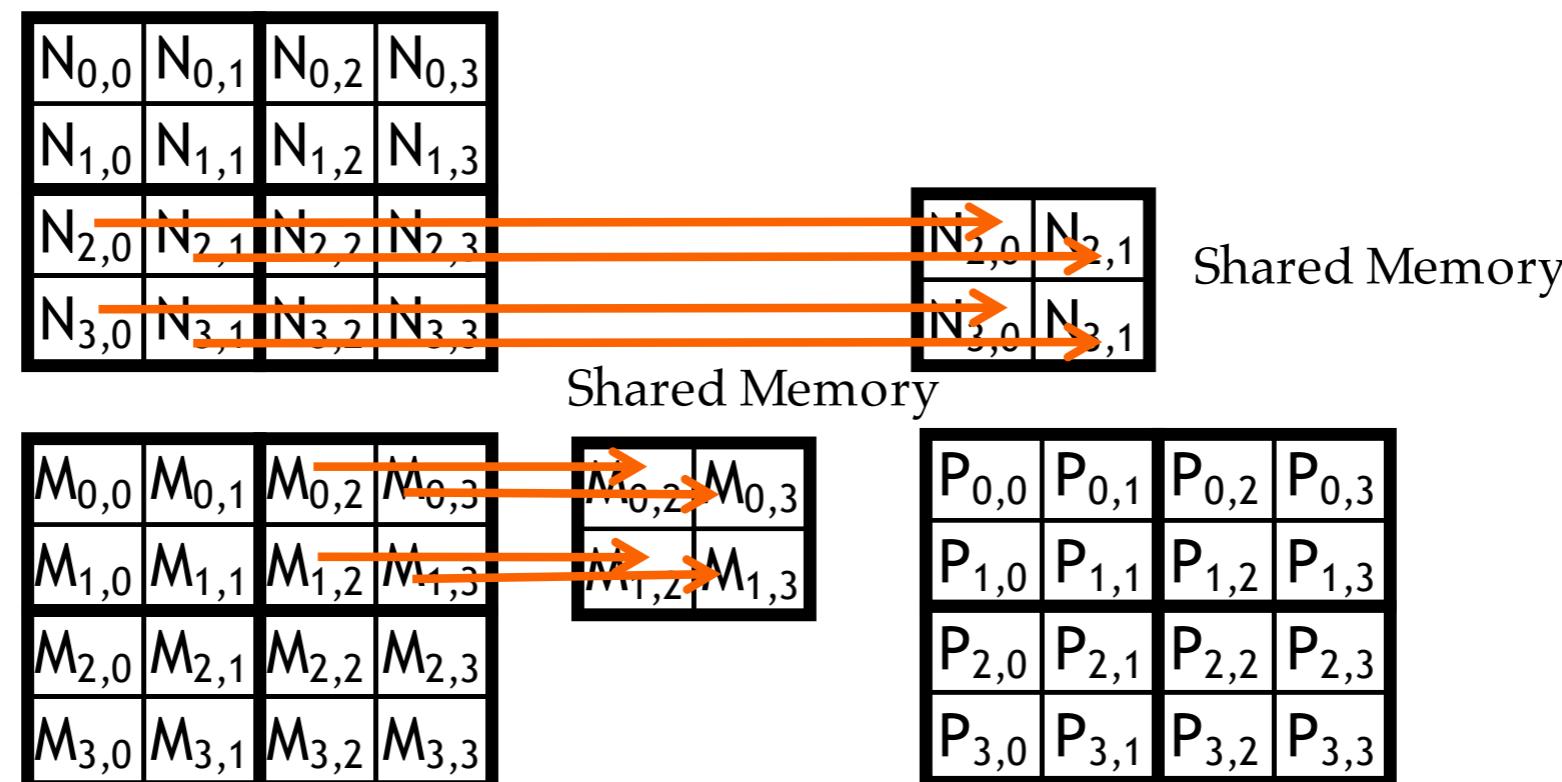
N _{0,0}	N _{0,1}	N _{0,2}	N _{0,3}
N _{1,0}	N _{1,1}	N _{1,2}	N _{1,3}
N _{2,0}	N _{2,1}	N _{2,2}	N _{2,3}
N _{3,0}	N _{3,1}	N _{3,2}	N _{3,3}

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}



Tiling phases

Phase 1 Load for Block (0,0)



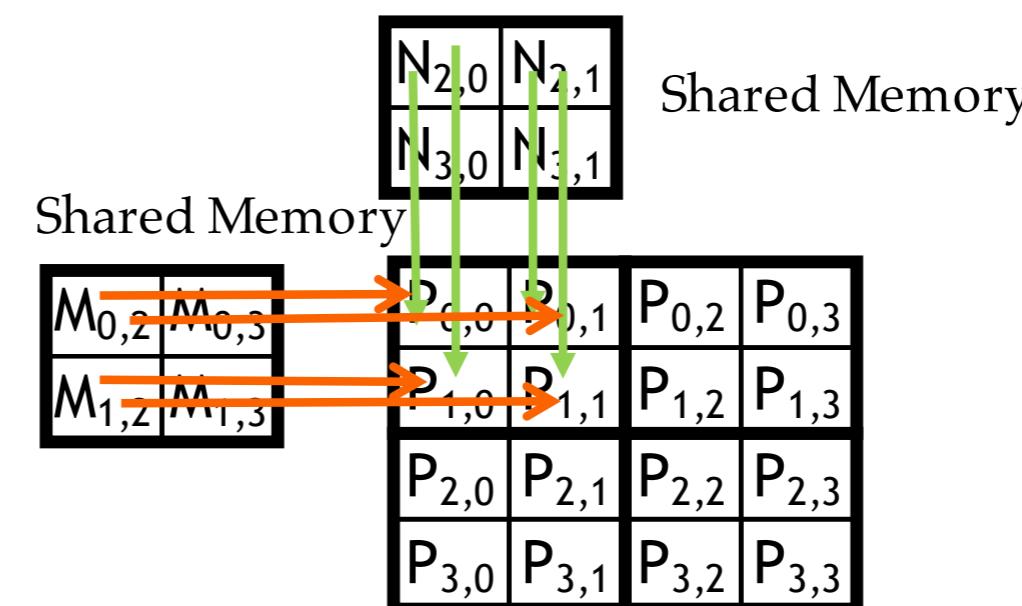


Tiling phases

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

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

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



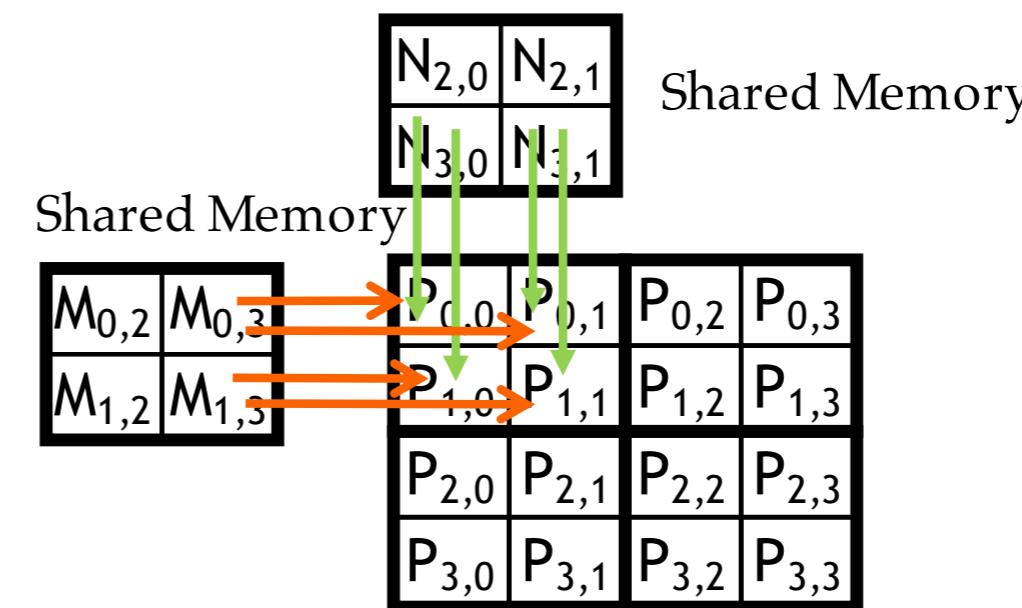


Tiling phases

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

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

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



Execution Phases of Toy Example

	Phase 0			Phase 1		
thread _{0,0}	$M_{0,0}$ ↓ $Mds_{0,0}$	$N_{0,0}$ ↓ $Nds_{0,0}$	$PValue_{0,0} += Mds_{0,0}*Nds_{0,0} + Mds_{0,1}*Nds_{1,0}$	$M_{0,2}$ ↓ $Mds_{0,0}$	$N_{2,0}$ ↓ $Nds_{0,0}$	$PValue_{0,0} += Mds_{0,0}*Nds_{0,0} + Mds_{0,1}*Nds_{1,0}$
thread _{0,1}	$M_{0,1}$ ↓ $Mds_{0,1}$	$N_{0,1}$ ↓ $Nds_{1,0}$	$PValue_{0,1} += Mds_{0,0}*Nds_{0,1} + Mds_{0,1}*Nds_{1,1}$	$M_{0,3}$ ↓ $Mds_{0,1}$	$N_{2,1}$ ↓ $Nds_{0,1}$	$PValue_{0,1} += Mds_{0,0}*Nds_{0,1} + Mds_{0,1}*Nds_{1,1}$
thread _{1,0}	$M_{1,0}$ ↓ $Mds_{1,0}$	$N_{1,0}$ ↓ $Nds_{1,0}$	$PValue_{1,0} += Mds_{1,0}*Nds_{0,0} + Mds_{1,1}*Nds_{1,0}$	$M_{1,2}$ ↓ $Mds_{1,0}$	$N_{3,0}$ ↓ $Nds_{1,0}$	$PValue_{1,0} += Mds_{1,0}*Nds_{0,0} + Mds_{1,1}*Nds_{1,0}$
thread _{1,1}	$M_{1,1}$ ↓ $Mds_{1,1}$	$N_{1,1}$ ↓ $Nds_{1,1}$	$PValue_{1,1} += Mds_{1,0}*Nds_{0,1} + Mds_{1,1}*Nds_{1,1}$	$M_{1,3}$ ↓ $Mds_{1,1}$	$N_{3,1}$ ↓ $Nds_{1,1}$	$PValue_{1,1} += Mds_{1,0}*Nds_{0,1} + Mds_{1,1}*Nds_{1,1}$

time →

- Shared memory allows each value to be accessed by multiple threads

Execution Phases of Toy Example

	Phase 0			Phase 1		
thread _{0,0}	$M_{0,0}$ ↓ $Mds_{0,0}$	$N_{0,0}$ ↓ $Nds_{0,0}$	$PValue_{0,0} +=$ $Mds_{0,0} * Nds_{0,0} +$ $Mds_{0,1} * Nds_{1,0}$	$M_{0,2}$ ↓ $Mds_{0,0}$	$N_{2,0}$ ↓ $Nds_{0,0}$	$PValue_{0,0} +=$ $Mds_{0,0} * Nds_{0,0} +$ $Mds_{0,1} * Nds_{1,0}$
thread _{0,1}	$M_{0,1}$ ↓ $Mds_{0,1}$	$N_{0,1}$ ↓ $Nds_{1,0}$	$PValue_{0,1} +=$ $Mds_{0,0} * Nds_{0,1} +$ $Mds_{0,1} * Nds_{1,1}$	$M_{0,3}$ ↓ $Mds_{0,1}$	$N_{2,1}$ ↓ $Nds_{0,1}$	$PValue_{0,1} +=$ $Mds_{0,0} * Nds_{0,1} +$ $Mds_{0,1} * Nds_{1,1}$
thread _{1,0}	$M_{1,0}$ ↓ $Mds_{1,0}$	$N_{1,0}$ ↓ $Nds_{1,0}$	$PValue_{1,0} +=$ $Mds_{1,0} * Nds_{0,0} +$ $Mds_{1,1} * Nds_{1,0}$	$M_{1,2}$ ↓ $Mds_{1,0}$	$N_{3,0}$ ↓ $Nds_{1,0}$	$PValue_{1,0} +=$ $Mds_{1,0} * Nds_{0,0} +$ $Mds_{1,1} * Nds_{1,0}$
thread _{1,1}	$M_{1,1}$ ↓ $Mds_{1,1}$	$N_{1,1}$ ↓ $Nds_{1,1}$	$PValue_{1,1} +=$ $Mds_{1,0} * Nds_{0,1} +$ $Mds_{1,1} * Nds_{1,1}$	$M_{1,3}$ ↓ $Mds_{1,1}$	$N_{3,1}$ ↓ $Nds_{1,1}$	$PValue_{1,1} +=$ $Mds_{1,0} * Nds_{0,1} +$ $Mds_{1,1} * Nds_{1,1}$

time →

- Shared memory allows each value to be accessed by multiple threads

Execution Phases of Toy Example

	Phase 0			Phase 1		
thread _{i,j}	M _{i,0} ↓ Mds _{i,0}	N _{i,0} ↓ Nds _{i,0}	PValue _{i,0} += Mds _{i,0} *Nds _{i,0} + Mds _{i,1} *Nds _{i,1}	M _{i,2} ↓ Mds _{i,0}	N _{i,2} ↓ Nds _{i,0}	PValue _{i,0} += Mds _{i,0} *Nds _{i,0} + Mds _{i,1} *Nds _{i,1}
thread _{0,0}	M _{0,0} ↓ Mds _{0,0}	N _{0,0} ↓ Nds _{0,0}	PValue _{0,0} += Mds _{0,0} *Nds _{0,0} + Mds _{0,1} *Nds _{1,0}	M _{0,2} ↓ Mds _{0,0}	N _{2,0} ↓ Nds _{0,0}	PValue _{0,0} += Mds _{0,0} *Nds _{0,0} + Mds _{0,1} *Nds _{1,0}
thread _{0,1}	M _{0,1} ↓ Mds _{0,1}	N _{0,1} ↓ Nds _{1,0}	PValue _{0,1} += Mds _{0,0} *Nds _{0,1} + Mds _{0,1} *Nds _{1,1}	M _{0,3} ↓ Mds _{0,1}	N _{2,1} ↓ Nds _{0,1}	PValue _{0,1} += Mds _{0,0} *Nds _{0,1} + Mds _{0,1} *Nds _{1,1}
thread _{1,0}	M _{1,0} ↓ Mds _{1,0}	N _{1,0} ↓ Nds _{1,0}	PValue _{1,0} += Mds _{1,0} *Nds _{0,0} + Mds _{1,1} *Nds _{1,0}	M _{1,2} ↓ Mds _{1,0}	N _{3,0} ↓ Nds _{1,0}	PValue _{1,0} += Mds _{1,0} *Nds _{0,0} + Mds _{1,1} *Nds _{1,0}
thread _{1,1}	M _{1,1} ↓ Mds _{1,1}	N _{1,1} ↓ Nds _{1,1}	PValue _{1,1} += Mds _{1,0} *Nds _{0,1} + Mds _{1,1} *Nds _{1,1}	M _{1,3} ↓ Mds _{1,1}	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

Execution Phases of Toy Example

	Phase 0			Phase 1		
thread _{0,0}	$M_{0,0}$ ↓ $Mds_{0,0}$	$N_{0,0}$ ↓ $Nds_{0,0}$	$PValue_{0,0} +=$ $Mds_{0,0} * Nds_{0,0} +$ $Mds_{0,1} * Nds_{1,0}$	$M_{0,2}$ ↓ $Mds_{0,0}$	$N_{2,0}$ ↓ $Nds_{0,0}$	$PValue_{0,0} +=$ $Mds_{0,0} * Nds_{0,0} +$ $Mds_{0,1} * Nds_{1,0}$
thread _{0,1}	$M_{0,1}$ ↓ $Mds_{0,1}$	$N_{0,1}$ ↓ $Nds_{1,0}$	$PValue_{0,1} +=$ $Mds_{0,0} * Nds_{0,1} +$ $Mds_{0,1} * Nds_{1,1}$	$M_{0,3}$ ↓ $Mds_{0,1}$	$N_{2,1}$ ↓ $Nds_{0,1}$	$PValue_{0,1} +=$ $Mds_{0,0} * Nds_{0,1} +$ $Mds_{0,1} * Nds_{1,1}$
thread _{1,0}	$M_{1,0}$ ↓ $Mds_{1,0}$	$N_{1,0}$ ↓ $Nds_{1,0}$	$PValue_{1,0} +=$ $Mds_{1,0} * Nds_{0,0} +$ $Mds_{1,1} * Nds_{1,0}$	$M_{1,2}$ ↓ $Mds_{1,0}$	$N_{3,0}$ ↓ $Nds_{1,0}$	$PValue_{1,0} +=$ $Mds_{1,0} * Nds_{0,0} +$ $Mds_{1,1} * Nds_{1,0}$

- In general, if an input matrix is of dimension Width and the tile size is TILE_WIDTH, the dot product would be performed in $\lceil \text{Width}/\text{TILE_WIDTH} \rceil$ phases.
- The creation of these phases is key to the reduction of accesses to the global memory.
- With each phase focusing on a small subset of the input matrix values, the threads can collaboratively load the subset into the shared memory and use the values in the shared memory to satisfy their overlapping input needs in the phase.
- Mds and Nds are re-used to hold the input values in different phases: reducing need of shared memory.

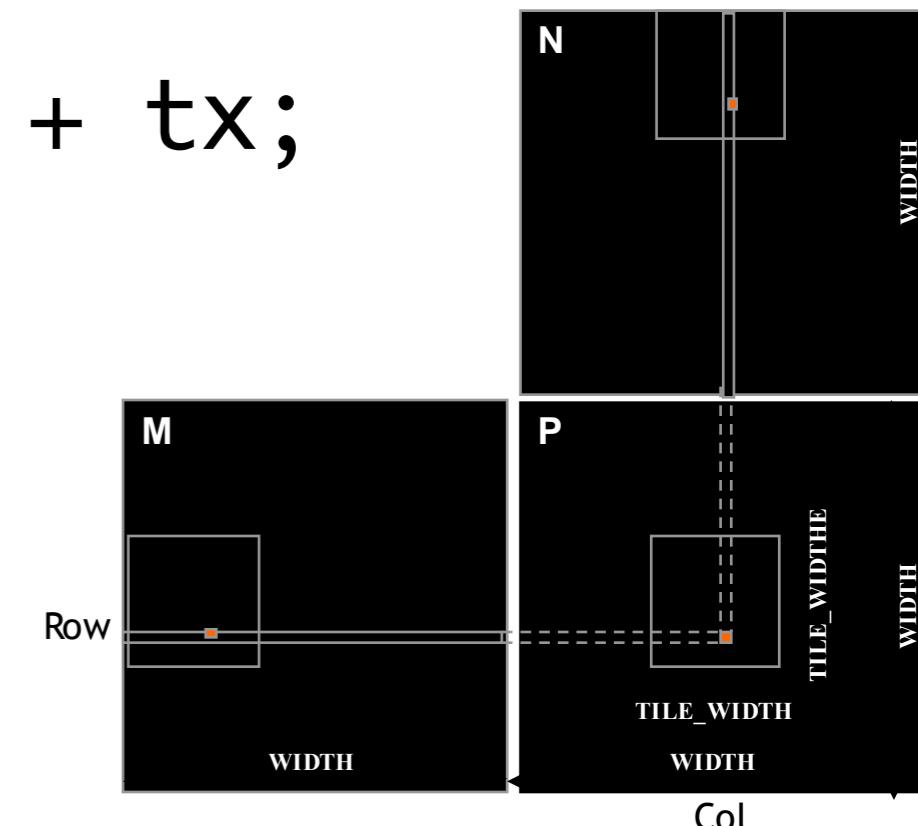
Barrier Synchronization

- 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)

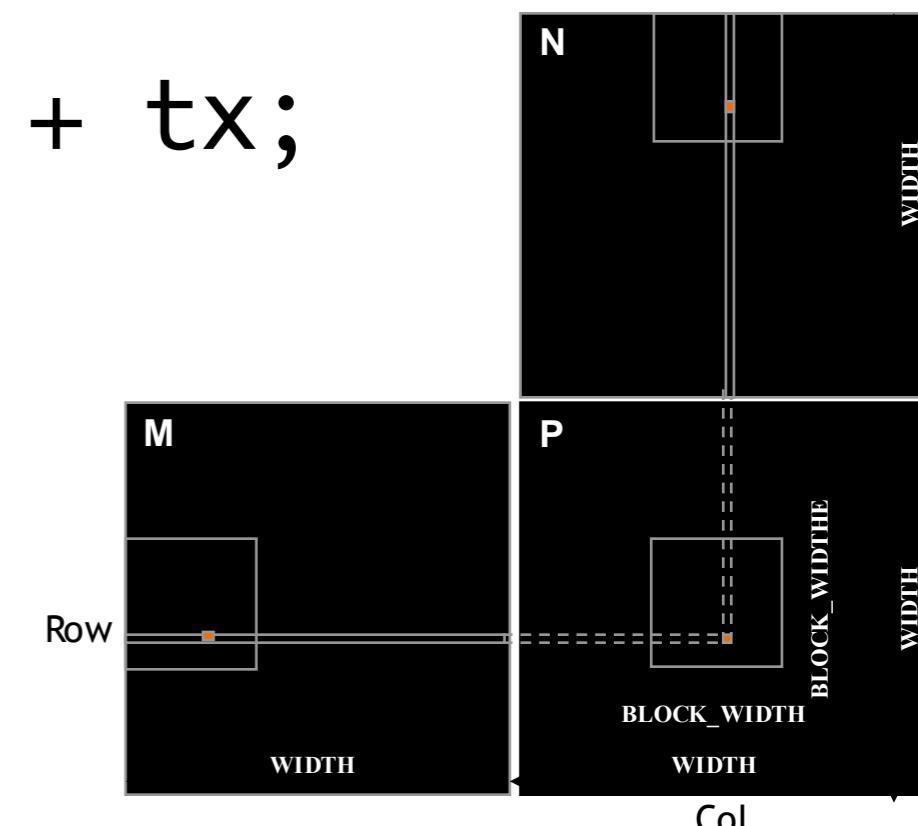
- Have each thread load an M element and an N element at the same relative position as its P element.
- `int Row = by * blockDim.y + ty;`
- `int Col = bx * blockDim.x + tx;`
- 2D indexing for accessing Tile 0:

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:
 $M[Row][tx]$
 $N[ty][Col]$

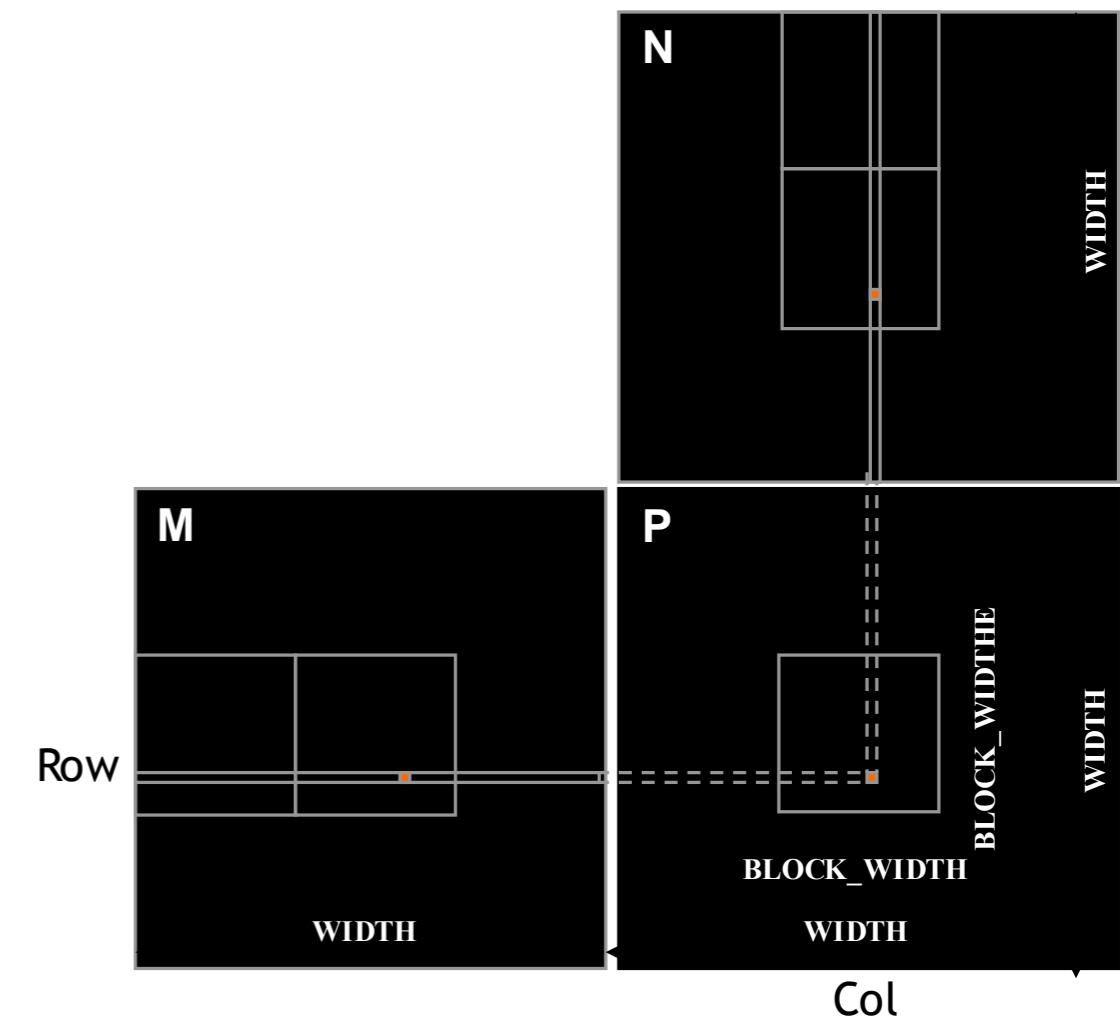




Loading Input Tile 1 of M (Phase 1)

- 2D indexing for accessing Tile 1:

$M[\text{Row}][1 * \text{TILE_WIDTH} + tx]$
 $N[1 * \text{TILE_WIDTH} + ty][\text{Col}]$

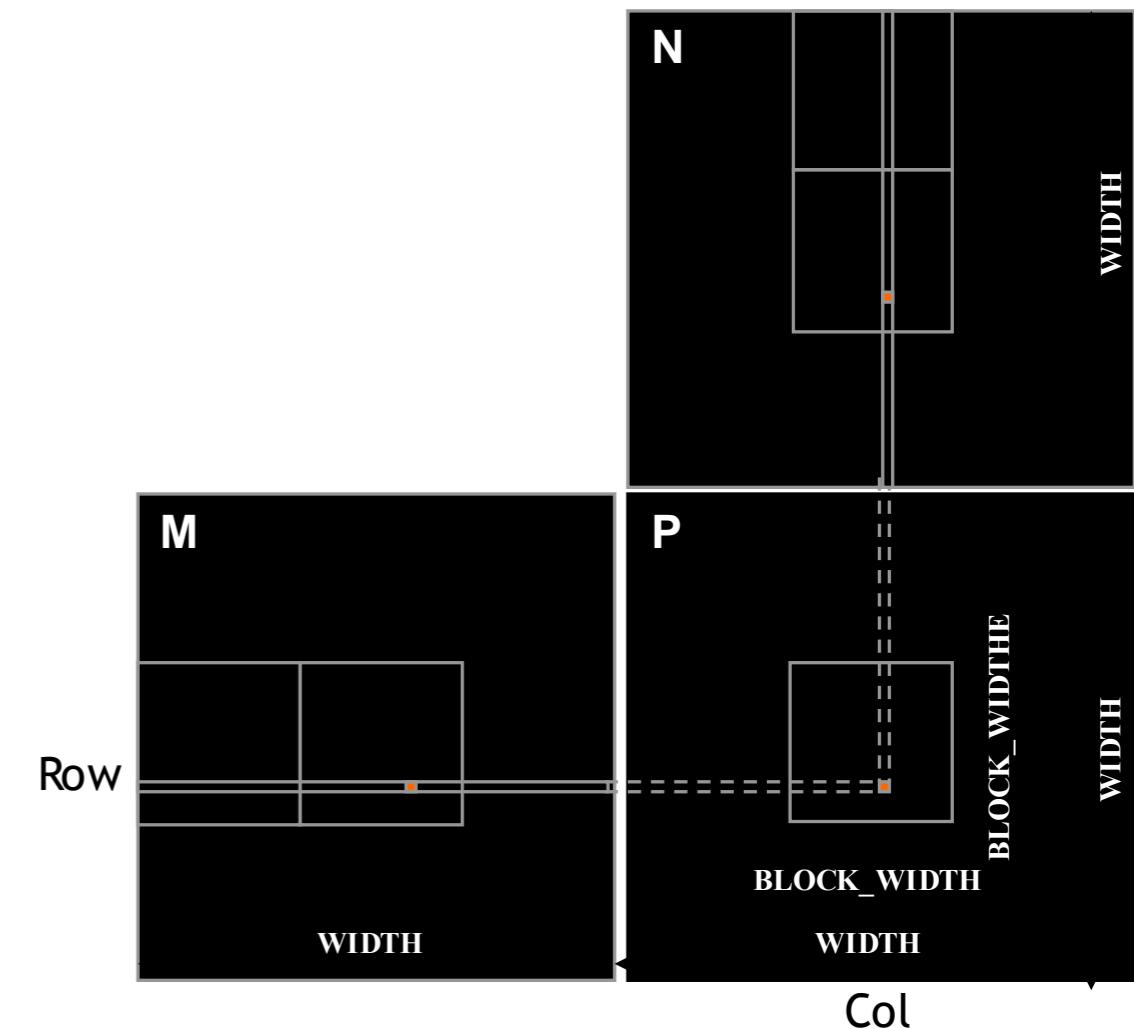




Loading Input Tile 1 of N (Phase 1)

- 2D indexing for accessing Tile 1:

$$M[\text{Row}][1 * \text{TILE_WIDTH} + tx]$$

$$N[1 * \text{TILE_WIDTH} + ty][\text{Col}]$$




Allocating M and N

- M and N can be allocated dynamically, using 1D indexing
- $M[Row][ph*TILE_WIDTH+tx]$
- $M[Row*Width + ph*TILE_WIDTH + tx]$ 
- $N[ph*TILE_WIDTH+ty][Col]$
- $N[(ph*TILE_WIDTH+ty)*Width + Col]$
- where ph 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 Mds[TILE_WIDTH][TILE_WIDTH];  
    __shared__ float Nds[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 ph = 0; ph < Width/TILE_WIDTH; ++ph) {  
        // Collaborative loading of M and N tiles into shared memory  
        Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH+tx];  
        Nds[ty][tx] = N[(ph*TILE_WIDTH+ty)*Width + Col];  ⚡  
        __syncthreads();  
  
        for (int i = 0; i < TILE_WIDTH; ++i)  
            Pvalue += Mds[ty][i] * Nds[i][tx];  
        __syncthreads();  ⚡  
    }  
    P[Row*Width+Col] = Pvalue;  
}
```

Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, Int Width) {  
  
    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];  
    __shared__ float Nds[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 ph = 0; ph < Width/TILE_WIDTH; ++ph) {  
        // Collaborative loading of M and N tiles into shared memory  
        Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH+tx];  
        Nds[ty][tx] = N[(ph*TILE_WIDTH+ty)*Width + Col];  
        __syncthreads();  
  
        for (int i = 0; i < TILE_WIDTH; ++i)  
            Pvalue += Mds[ty][i] * Nds[i][tx];  
        __syncthreads();  
    }  
  
    P[Row*Width+Col] = Pvalue;  
}
```

Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, Int Width) {  
  
    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];  
    __shared__ float Nds[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 ph = 0; ph < Width/TILE_WIDTH; ++ph) {  
        // Collaborative loading of M and N tiles into shared memory  
        Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH+tx];  
        Nds[ty][tx] = N[(ph*TILE_WIDTH+ty)*Width + Col];  
        __syncthreads();  
  
        for (int i = 0; i < TILE_WIDTH; ++i)  
            Pvalue += Mds[ty][i] * Nds[i][tx];  
        __syncthreads();  
    }  
  
    P[Row*Width+Col] = Pvalue;  
}
```

Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, Int Width) {  
  
    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];  
    __shared__ float Nds[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 ph = 0; ph < Width/TILE_WIDTH; ++ph) {  
        // Collaborative loading of M and N tiles into shared memory  
        Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH+tx];  
        Nds[ty][tx] = N[(ph*TILE_WIDTH+ty)*Width + Col];  
        __syncthreads();  
  
        for (int i = 0; i < TILE_WIDTH; ++i)  
            Pvalue += Mds[ty][i] * Nds[i][tx];  
        __syncthreads();  
    }  
  
    P[Row*Width+Col] = Pvalue;  
}
```

Tiled Matrix Multiplication Kernel

```

__global__ void MatrixMulKernel(float* M, float* N, float* P, Int Width) {
    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Nds[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 ph = 0; ph < Width/TILE_WIDTH; ++ph) {
        // Collaborative loading of M and N tiles into shared memory
        Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH+tx];
        Nds[ty][tx] = N[(ph*TILE_WIDTH+ty)*Width + Col];
        __syncthreads();

        for (int i = 0; i < TILE_WIDTH; ++i)
            Pvalue += Mds[ty][i] * Nds[i][tx];
        __syncthreads();
    }

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

```

Shared variables: block scope



Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, Int Width) {  
    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];  
    __shared__ float Nds[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 ph = 0; ph < Width/TILE_WIDTH; ++ph) {  
        // Collaborative loading of M and N tiles into shared memory  
        Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH+tx];  
        Nds[ty][tx] = N[(ph*TILE_WIDTH+ty)*Width + Col];  
        __syncthreads();  
  
        for (int i = 0; i < TILE_WIDTH; ++i)  
            Pvalue += Mds[ty][i] * Nds[i][tx];  
        __syncthreads();  
    }  
  
    P[Row*Width+Col] = Pvalue;  
}
```

Shared variables: block scope

Automatic scalar variables: registers

Tiled Matrix Multiplication Kernel

```

__global__ void MatrixMulKernel(float* M, float* N, float* P, Int Width) {
    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Nds[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 ph = 0; ph < Width/TILE_WIDTH; ++ph) {
        // Collaborative loading of M and N tiles into shared memory
        Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH+tx];
        Nds[ty][tx] = N[(ph*TILE_WIDTH+ty)*Width + Col];
        __syncthreads();

        for (int i = 0; i < TILE_WIDTH; ++i)
            Pvalue += Mds[ty][i] * Nds[i][tx];
        __syncthreads();
    }

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

```

Shared variables: block scope

Automatic scalar variables: registers

iterates over phases

Tiled Matrix Multiplication Kernel

```

__global__ void MatrixMulKernel(float* M, float* N, float* P, Int Width) {
    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Nds[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 ph = 0; ph < Width/TILE_WIDTH; ++ph) {
        // Collaborative loading of M and N tiles into shared memory
        Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH+tx];
        Nds[ty][tx] = N[(ph*TILE_WIDTH+ty)*Width + Col];
        __syncthreads();

        for (int i = 0; i < TILE_WIDTH; ++i)
            Pvalue += Mds[ty][i] * Nds[i][tx];
        __syncthreads();
    }

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

```

Shared variables: block scope

Automatic scalar variables: registers

iterates over phases

load into shared memory

Tiled Matrix Multiplication Kernel

```

__global__ void MatrixMulKernel(float* M, float* N, float* P, Int Width) {
    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Nds[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 ph = 0; ph < Width/TILE_WIDTH; ++ph) {
        // Collaborative loading of M and N tiles into shared memory
        Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH+tx];
        Nds[ty][tx] = N[(ph*TILE_WIDTH+ty)*Width + Col];
        __syncthreads();  assures that all threads have loaded the data into shared mem.

        for (int i = 0; i < TILE_WIDTH; ++i)
            Pvalue += Mds[ty][i] * Nds[i][tx];
        __syncthreads();

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

```

Shared variables: block scope

Automatic scalar variables: registers

iterates over phases

load into shared memory

assures that all threads have loaded the data into shared mem.

Tiled Matrix Multiplication Kernel

```

__global__ void MatrixMulKernel(float* M, float* N, float* P, Int Width) {
    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Nds[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 ph = 0; ph < Width/TILE_WIDTH; ++ph) {
        // Collaborative loading of M and N tiles into shared memory
        Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH+tx];
        Nds[ty][tx] = N[(ph*TILE_WIDTH+ty)*Width + Col];
        __syncthreads();  assures that all threads have loaded the data into shared mem.

        for (int i = 0; i < TILE_WIDTH; ++i)
            Pvalue += Mds[ty][i] * Nds[i][tx];
        __syncthreads();  assures that all threads have finished using the data into
                           shared mem.

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

```

Shared variables: block scope

Automatic scalar variables: registers

iterates over phases

load into shared memory

assures that all threads have loaded the data into shared mem.

assures that all threads have finished using the data into shared mem.

Tile (Thread Block) Size Considerations

- Each thread block should have many threads
 - TILE_WIDTH of 16 gives $16 \times 16 = 256$ threads
 - TILE_WIDTH of 32 gives $32 \times 32 = 1024$ threads
- For 16, in each phase, each block performs $2 \times 256 = 512$ float loads from global memory for $256 \times (2 \times 16) = 8,192$ mul/add operations. (16 floating-point operations for each memory load)
- For 32, in each phase, each block performs $2 \times 1024 = 2048$ float loads from global memory for $1024 \times (2 \times 32) = 65,536$ 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 $2 \times 256 \times 4B = 2KB$ of shared memory.
 - For 16KB shared memory, one can potentially have up to **8 thread blocks 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 $2 \times 32 \times 32 \times 4 \text{ Byte} = 8K \text{ Byte}$ shared memory usage per thread block, allowing **2 thread blocks active** at the same time
 - However, the thread count limitation of 1536 threads per SM in current generation GPUs will reduce the number of blocks per SM to one!
 - Each `__syncthread()` can reduce the number of active threads for a block
 - More thread blocks can be advantageous 



Boundary checks

- The previous code assumes that the matrix has a size (a width) that is an exact multiple of `TILE_WIDTH`.
- Let's extend the code to handle matrices with arbitrary width...
 - ...without using padding, that would waste space and time in copying data.

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

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

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

Shared Memory

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

Shared Memory



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

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

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	

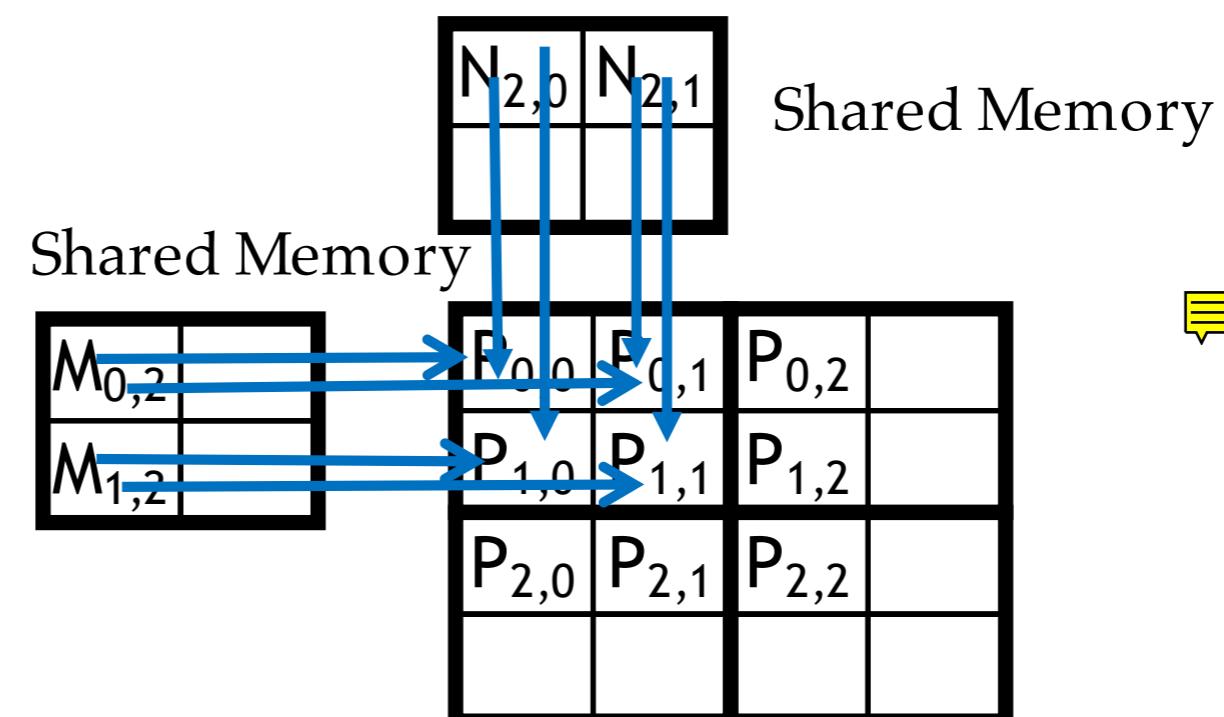
Threads (0,1) and (1,1) need special treatment in loading M tile



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

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

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

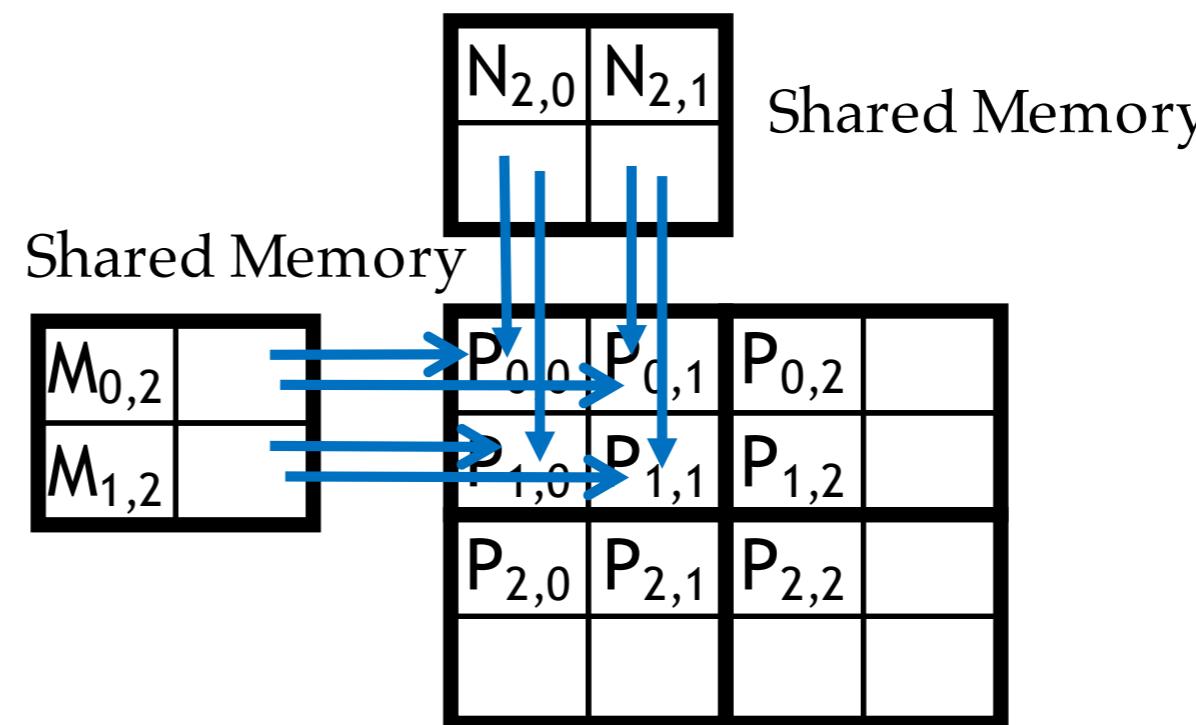




Phase 1 Use for Block (0,0) (iteration 1)

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

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

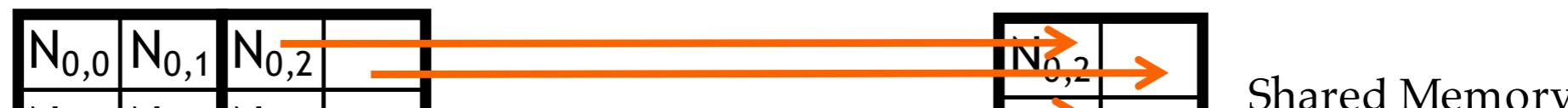


All Threads need special treatment. None of them should introduce invalidate contributions to their P elements.

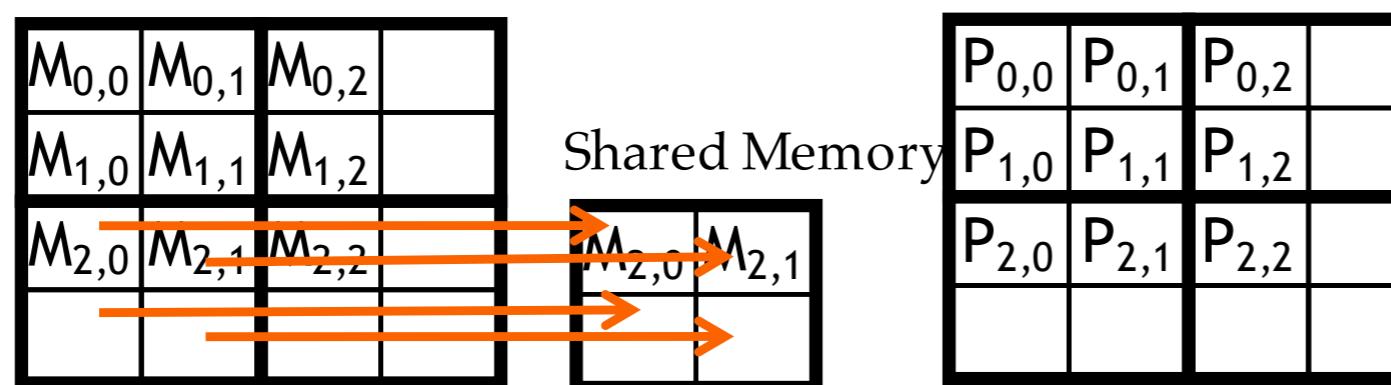


Phase 0 Loads for Block (1,1) for a 3x3 Example

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



Shared Memory



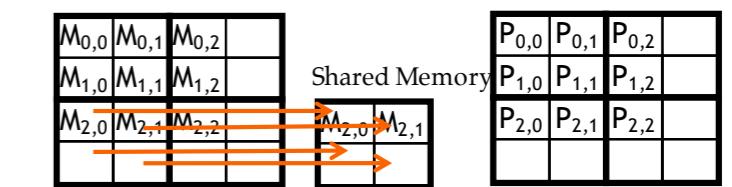
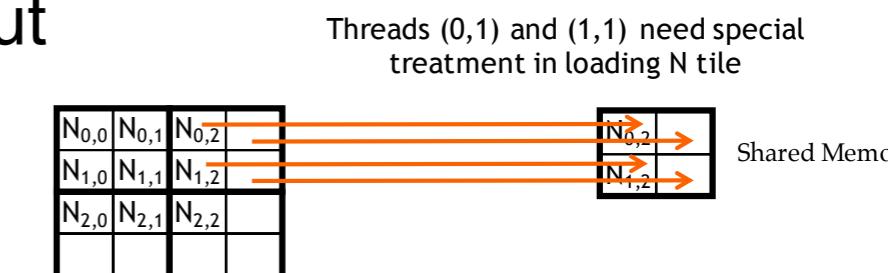
Shared Memory

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

Main cases in the example

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

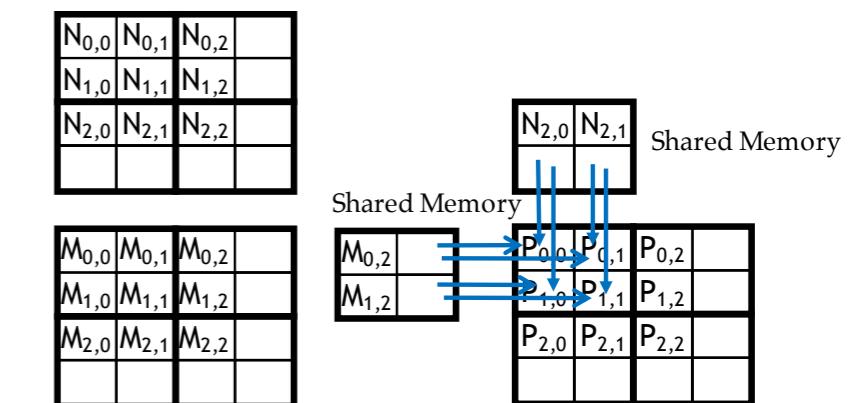
- Phase 0 of Block(1,1), Thread(0,1), assigned to calculate non-existent P[3,2] but need to participate in loading tile element N[1,2]



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

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

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



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

A solution

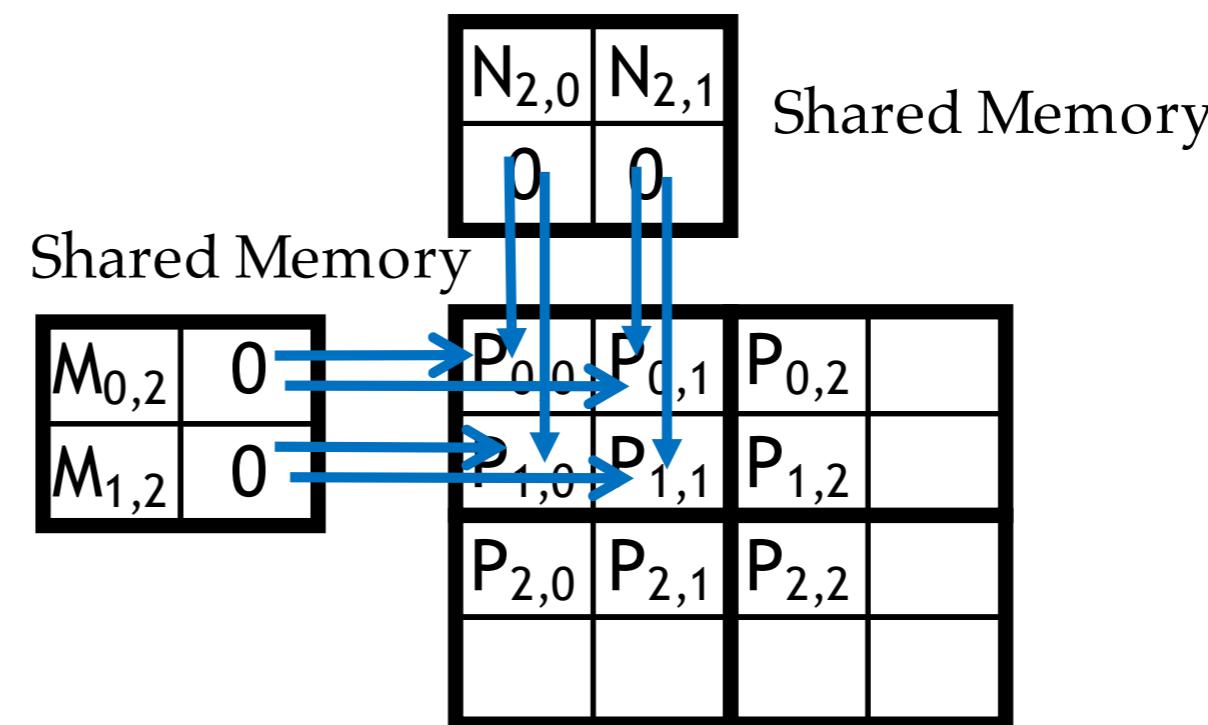
- When a thread is to load any input element, test if it is in the valid index range
 - If valid, proceed to load
 - Else, do not load, just write a 0
- Rationale: a 0 value will ensure that that the multiply-add step does not affect the final value of the output element
- The condition tested for loading 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)

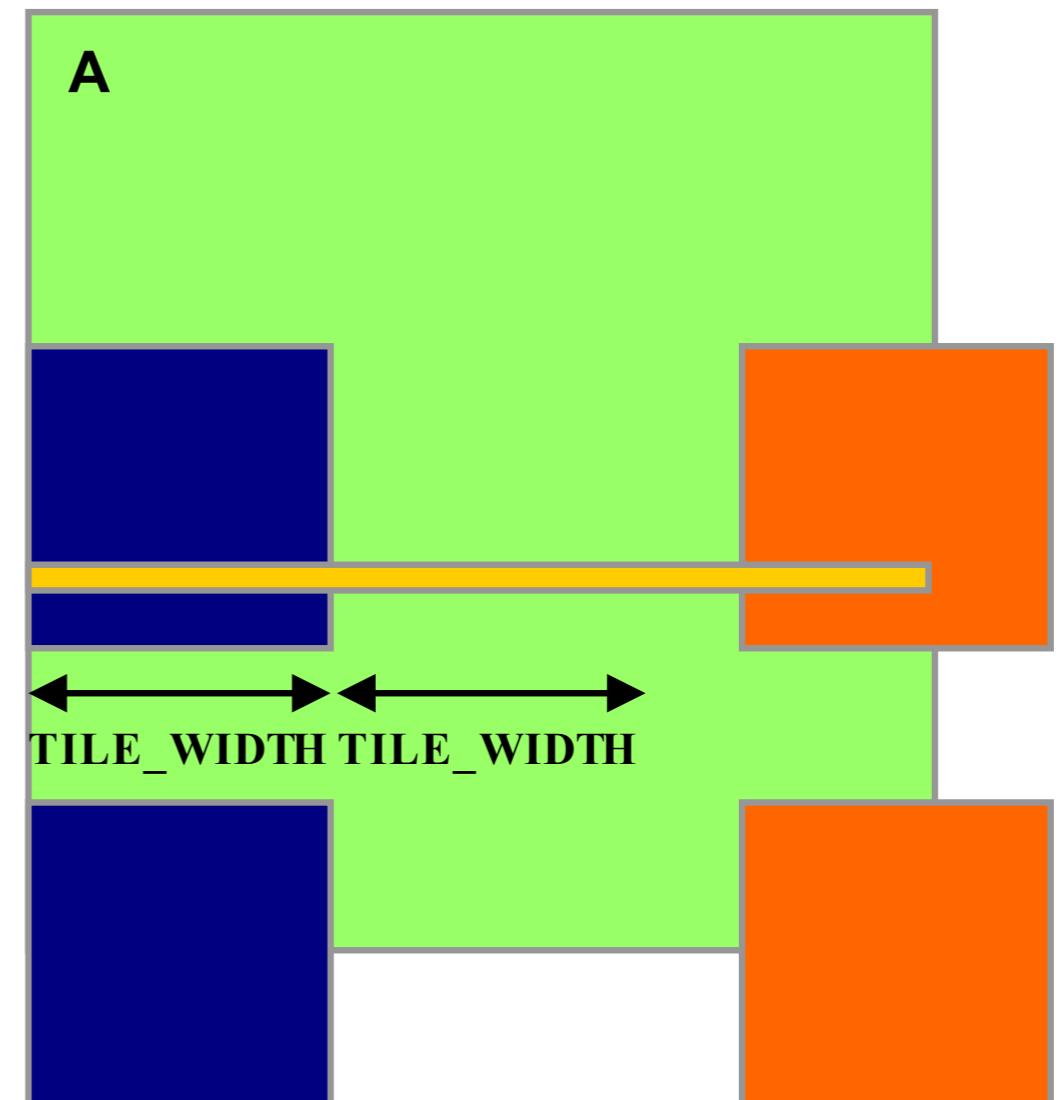
$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	

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



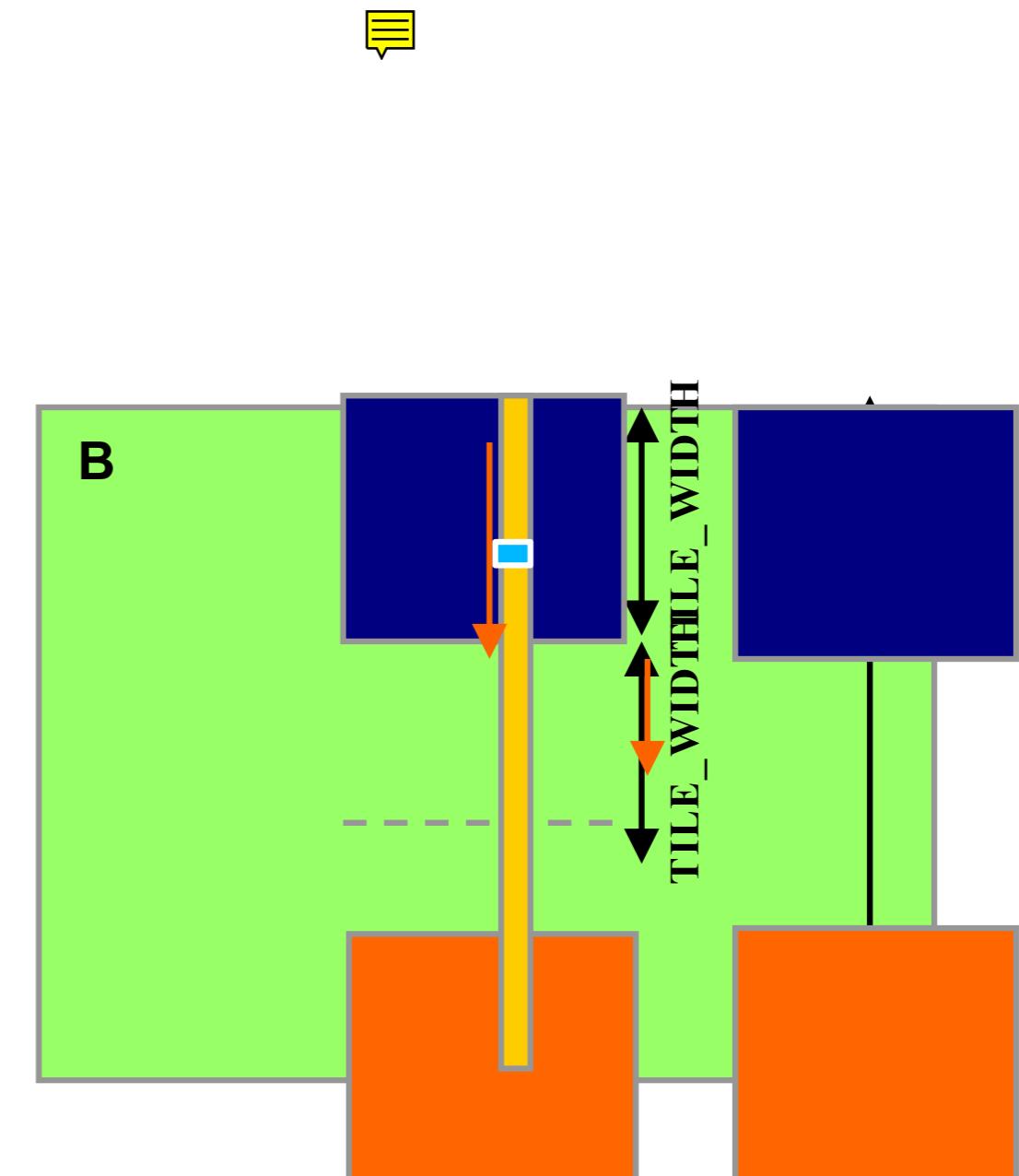
Boundary Condition for Input M Tile

- Each thread loads
 - $M[\text{Row}][p * \text{TILE_WIDTH} + tx]$
 - $M[\text{Row} * \text{Width} + p * \text{TILE_WIDTH} + tx]$
- Need to test
 - $(\text{Row} < \text{Width}) \&\& (p * \text{TILE_WIDTH} + tx < \text{Width})$
 - 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

```
for (int p = 0; p < (Width-1) / TILE_WIDTH + 1; ++p) {  
  
    if(Row < Width && p * TILE_WIDTH + tx < Width) {  
  
        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) {  
  
        ds_N[ty][tx] = N[(p * TILE_WIDTH + ty) * Width + Col];  
  
    } else {  
  
        ds_N[ty][tx] = 0.0;  
  
    }  
  
    __syncthreads();  
}
```

Inner Product – Before and After

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

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
 - A $j \times k$ M matrix multiplied with a $k \times l$ N matrix results in a $j \times l$ P matrix
 - So far we have seen 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

Credits

- These slides report material from:
 - NVIDIA GPU Teaching Kit



Books

- Programming Massively Parallel Processors: A Hands-on Approach, D. B. Kirk and W-M. W. Hwu, Morgan Kaufman - 2nd edition - Chapt. 4-6

or

Programming Massively Parallel Processors: A Hands-on Approach, D. B. Kirk and W-M. W. Hwu, Morgan Kaufman - 3rd edition - Chapt. 3-5