

Advanced Threads

In This Lecture

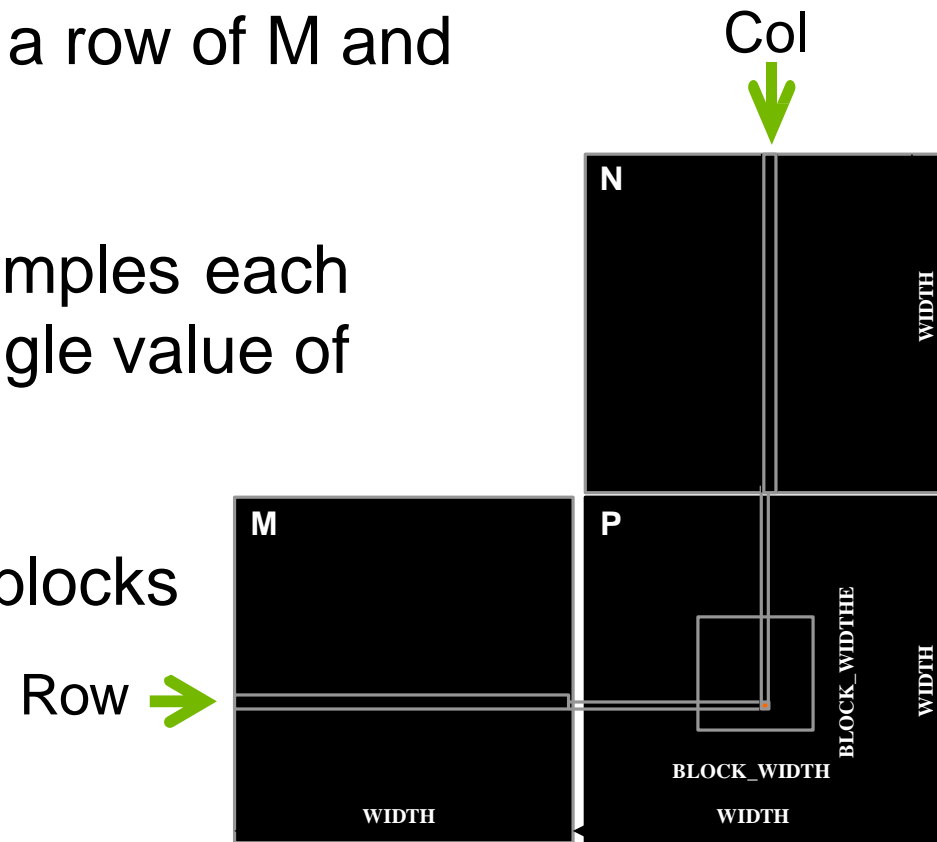
- Matrix Multiplication Review
- Reduce memory traffic: tiling
- Tiling Matrix Multiplication
- Memory Coalescing
- Boundary Checks

Matrix Multiplication Review



Basic problem: matrix multiplication

- When performing a matrix multiplication, each element of the output matrix P is an inner (dot) 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 P
- We organize threads as blocks

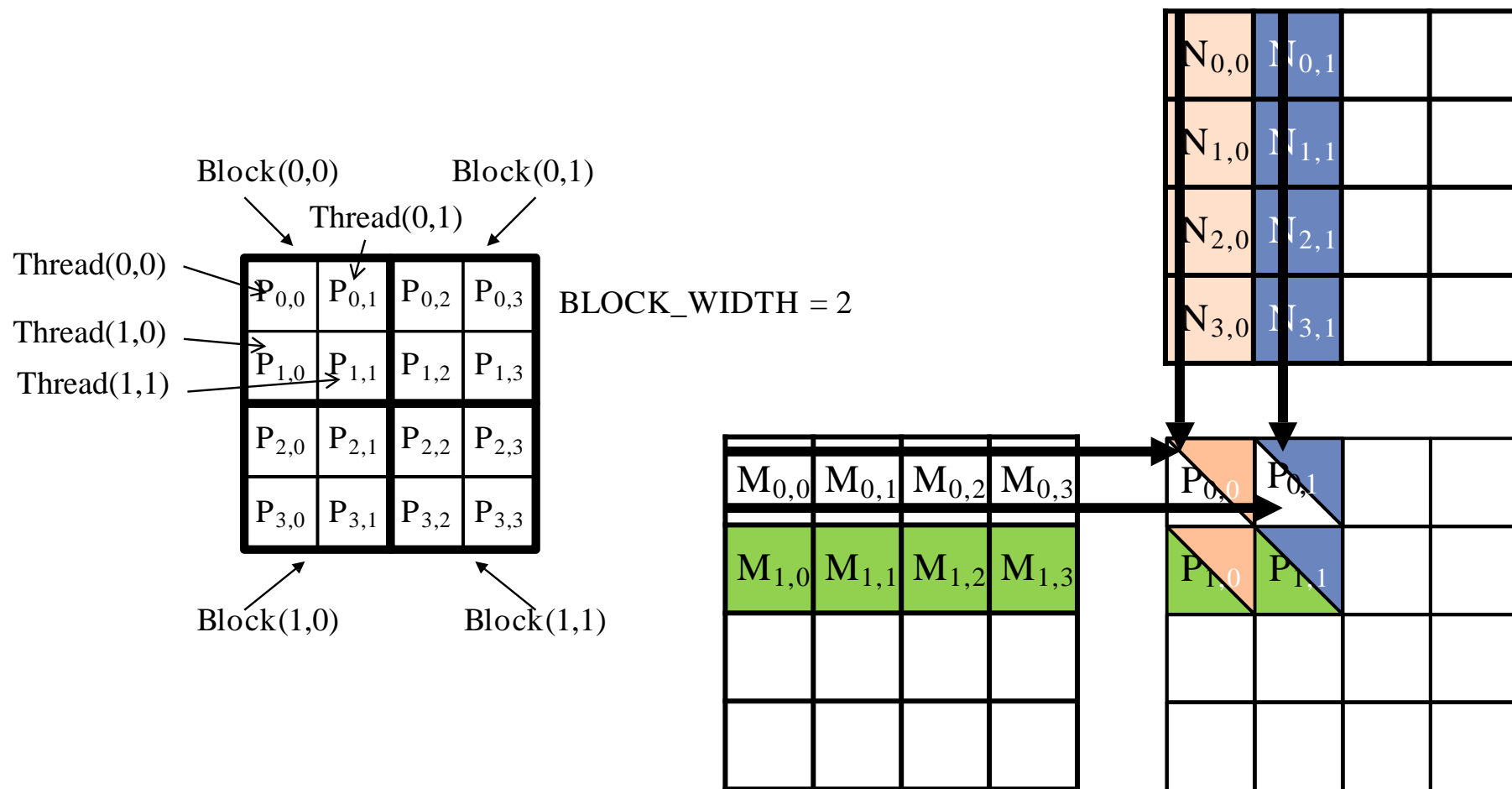


Basic Matrix Multiplication

```
__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;
    }
}
```

Toy example visualization



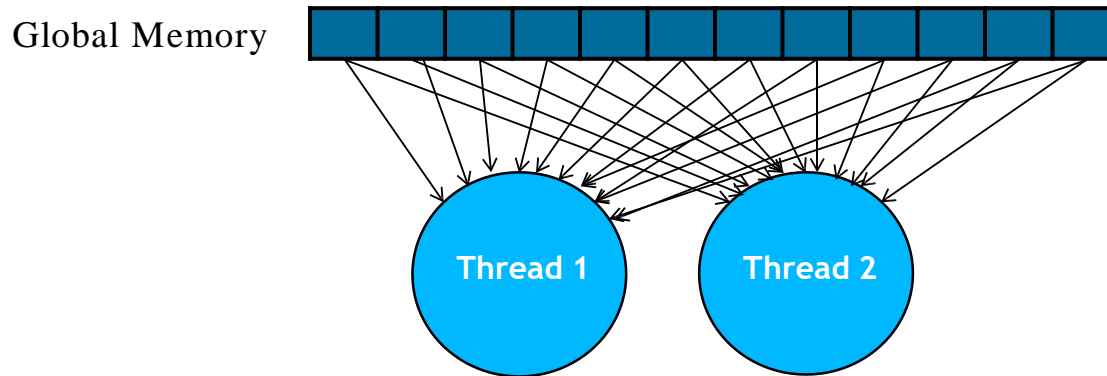
Reduce memory traffic: **Tiling**



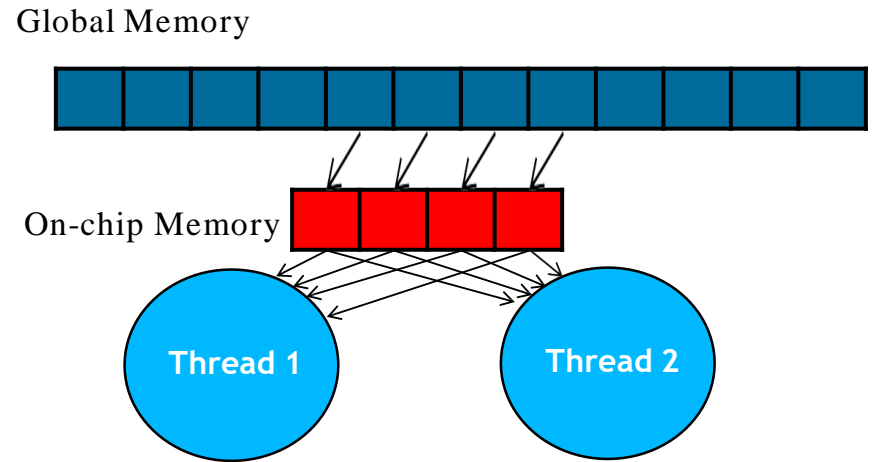
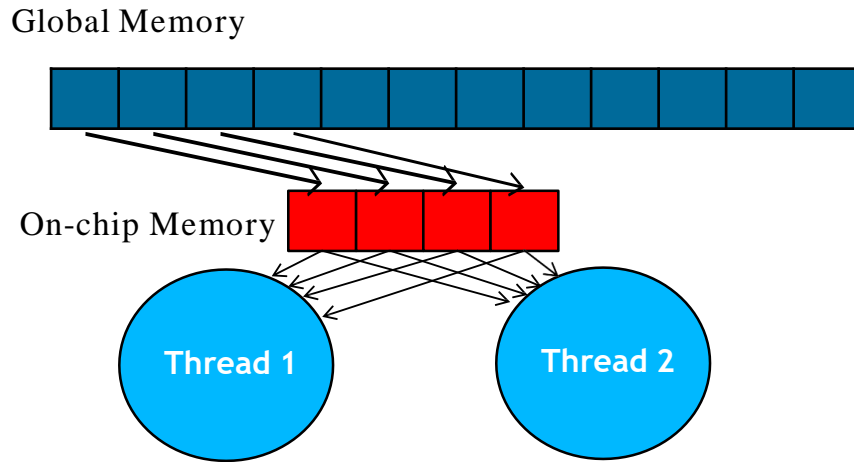
Tiling and CUDA memories

- Remind the tradeoffs of CUDA memories:
 - the global memory is **large but slow**;
 - the shared memory is **small but fast**. (100X low latency)
- 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

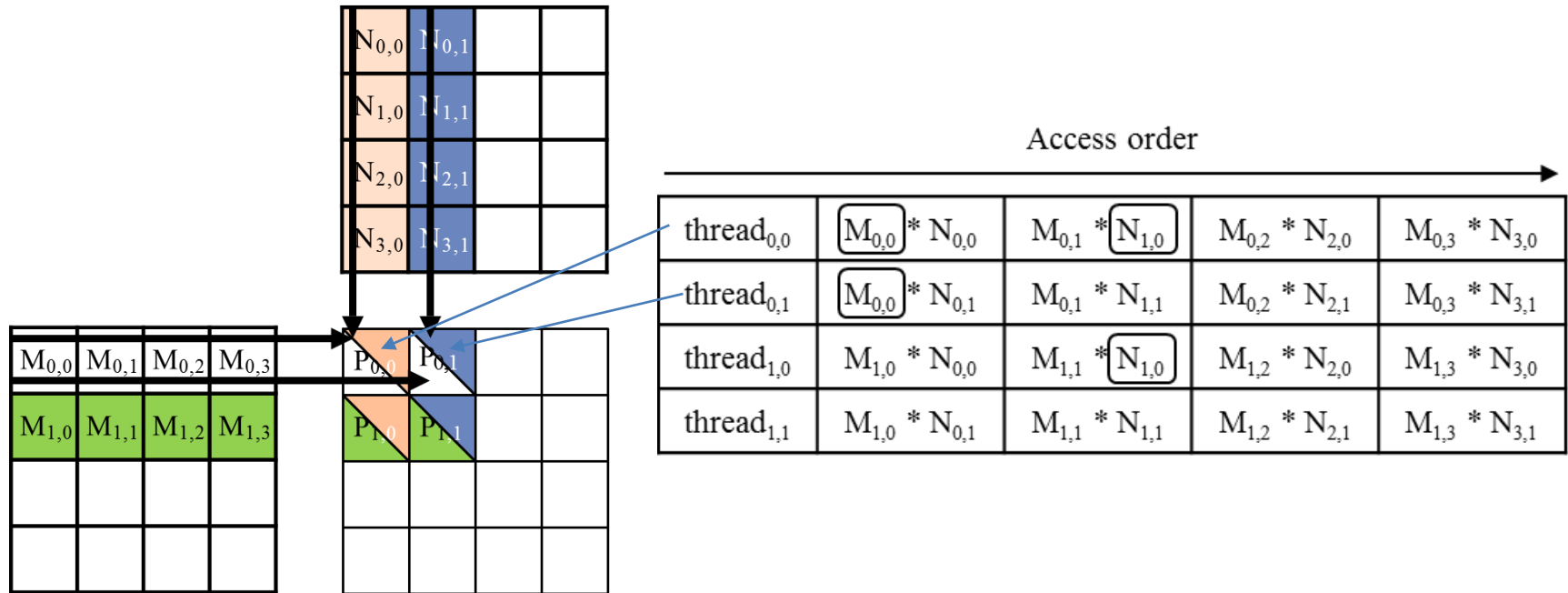


Bad



Good

Repeated Global Memory Access

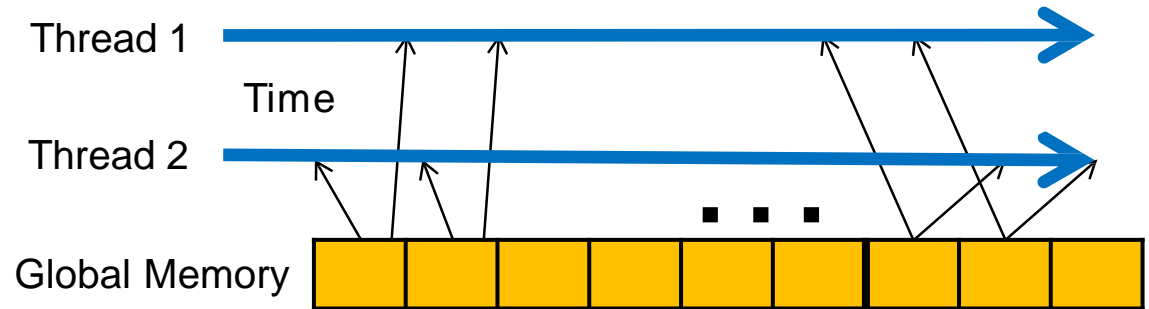


- Each thread accesses four elements of M and four elements of N during its execution. Among the four threads highlighted, there is a significant overlap in the M and N elements they access. For example, **thread_{0,0} and thread_{0,1} both access M_{0,0}** as well as the rest of row 0 of M.
- If we can somehow manage to have thread_{0,0} and thread_{0,1} to collaborate so that these M elements are only **loaded once from global memory**, 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}. (If block size is 4x4, then 4 accesses on a SM)

Memory access patterns

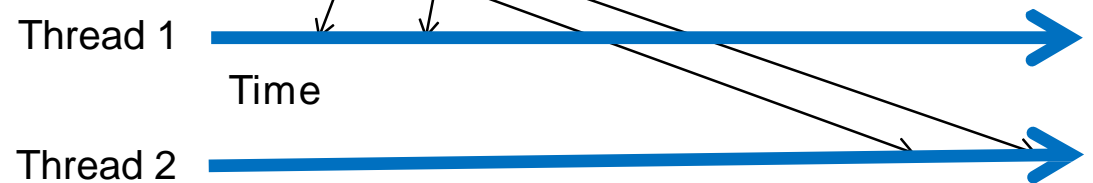
Good:

when threads have similar
access timing



Bad:

when threads have very
different timing



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

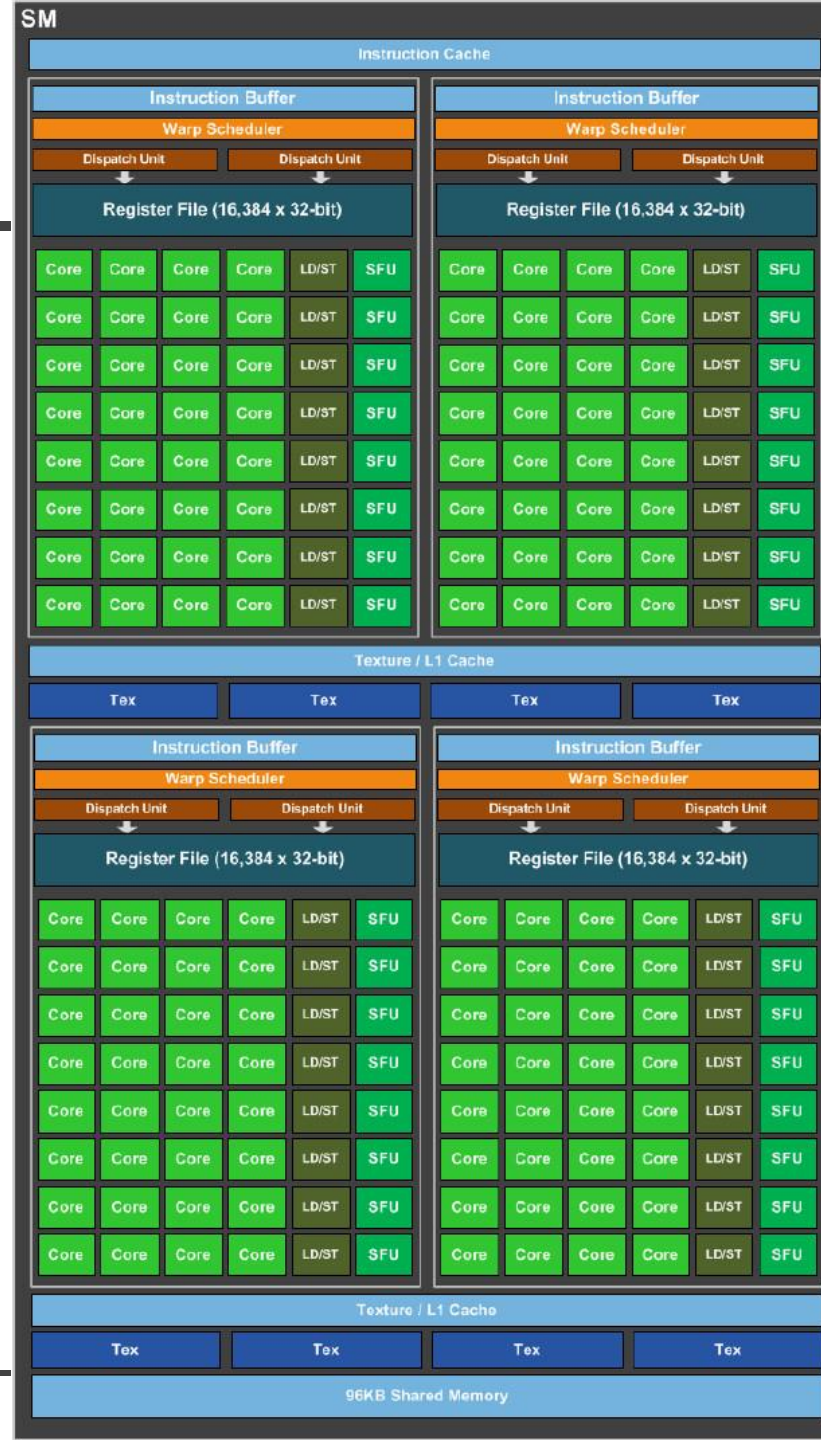
Long access sequences of each thread are broken into phases, using barriers to synchronize access times to each section by the threads.

Tiling in General: how ?

- Identify a tile of global memory contents that are accessed by multiple threads
- Load the tile from global memory into on-chip memory
- 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 (tile)
- Move on to the next tile

On-Chip Memory in SM

- A SM of NVIDIA GTX 1080
 - Registers
 - Shared Memory (96Kb)

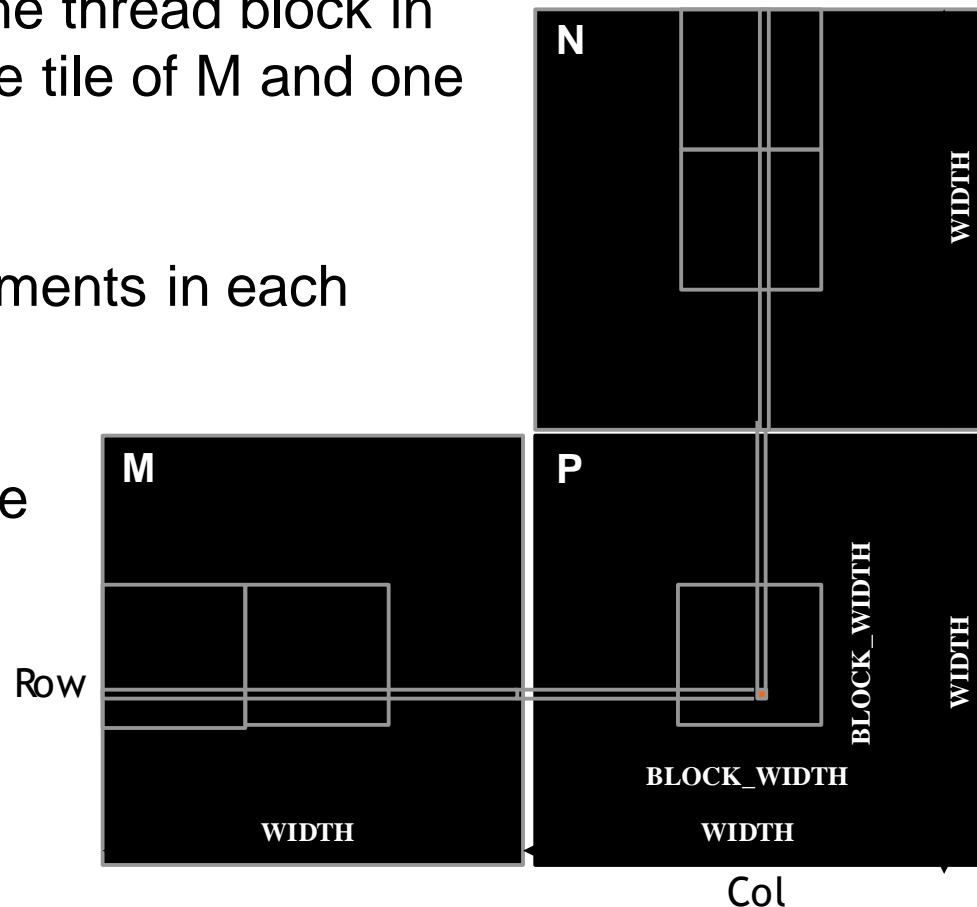


Tiling Matrix Multiplication



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

Shared Memory Preparation

- In reality **M**, **N**, **P** are 1D arrays in Global Memory
- Shared Memories for **M** and **N** are allocated as 2D arrays

N

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

Shared Memory

P

M

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

Shared Memory

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

Tiling phases

Phase 0 Load for Block (0,0)

→ Copy data to Shared Memory

N

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

Shared Memory

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

P

M

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

Shared Memory

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

Tiling phases

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

→ Calculate dot product

M

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

Shared Memory

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

Shared Memory

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

N

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

P

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

Tiling phases

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

→ Calculate dot product

M

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

Shared Memory

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

N

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

Shared Memory

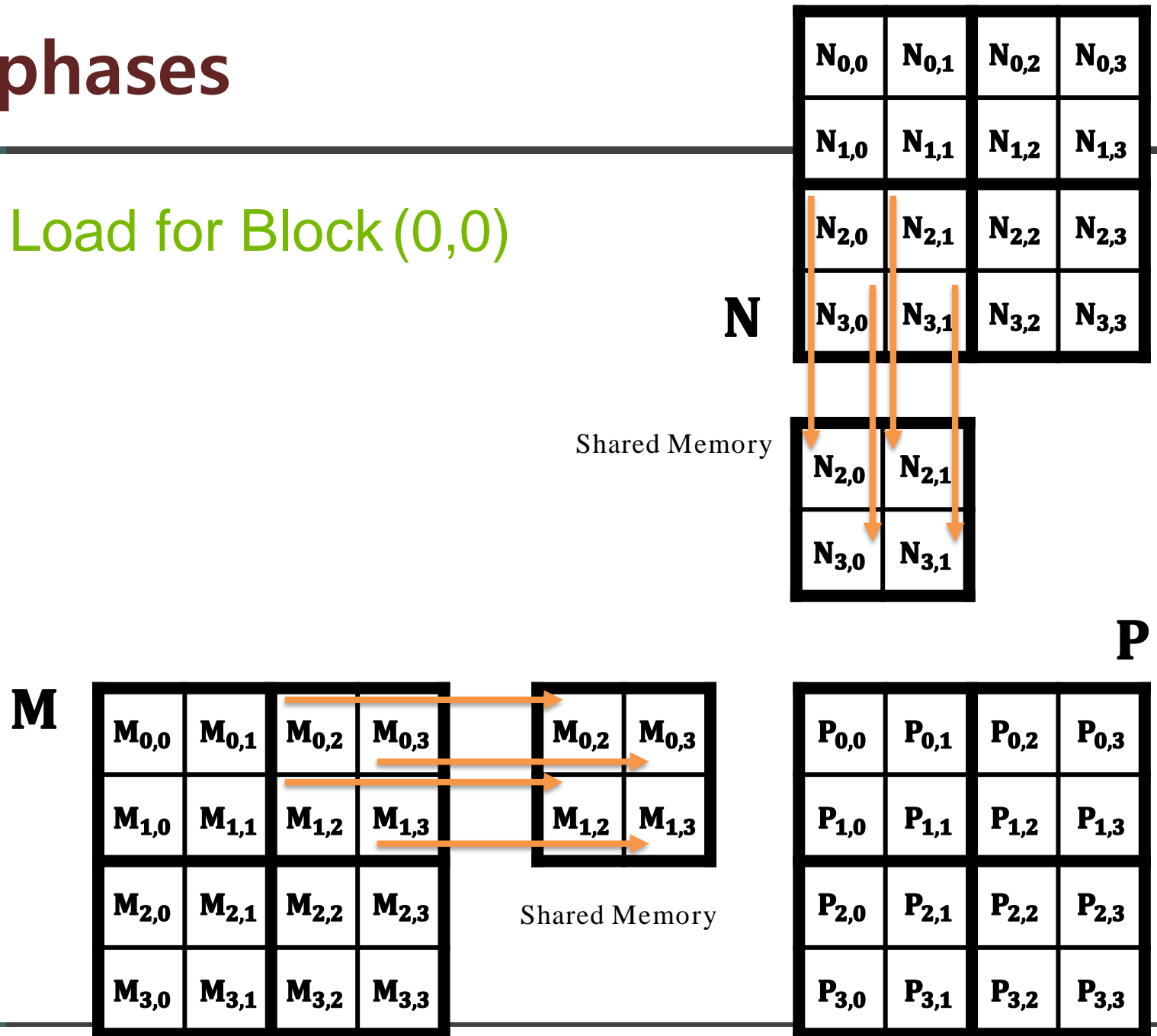
$N_{0,0}$	$N_{0,1}$
$N_{1,0}$	$N_{1,1}$

P

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

Tiling phases

Phase 1 Load for Block (0,0)



Tiling phases

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

M

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

Shared Memory

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

Shared Memory

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

N

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

P

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

Tiling phases

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

→ Result for Block (0,0) is finished

M

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

Shared Memory

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

N

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

Shared Memory

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


P

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

Execution Phases of Toy Example

Time 

	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

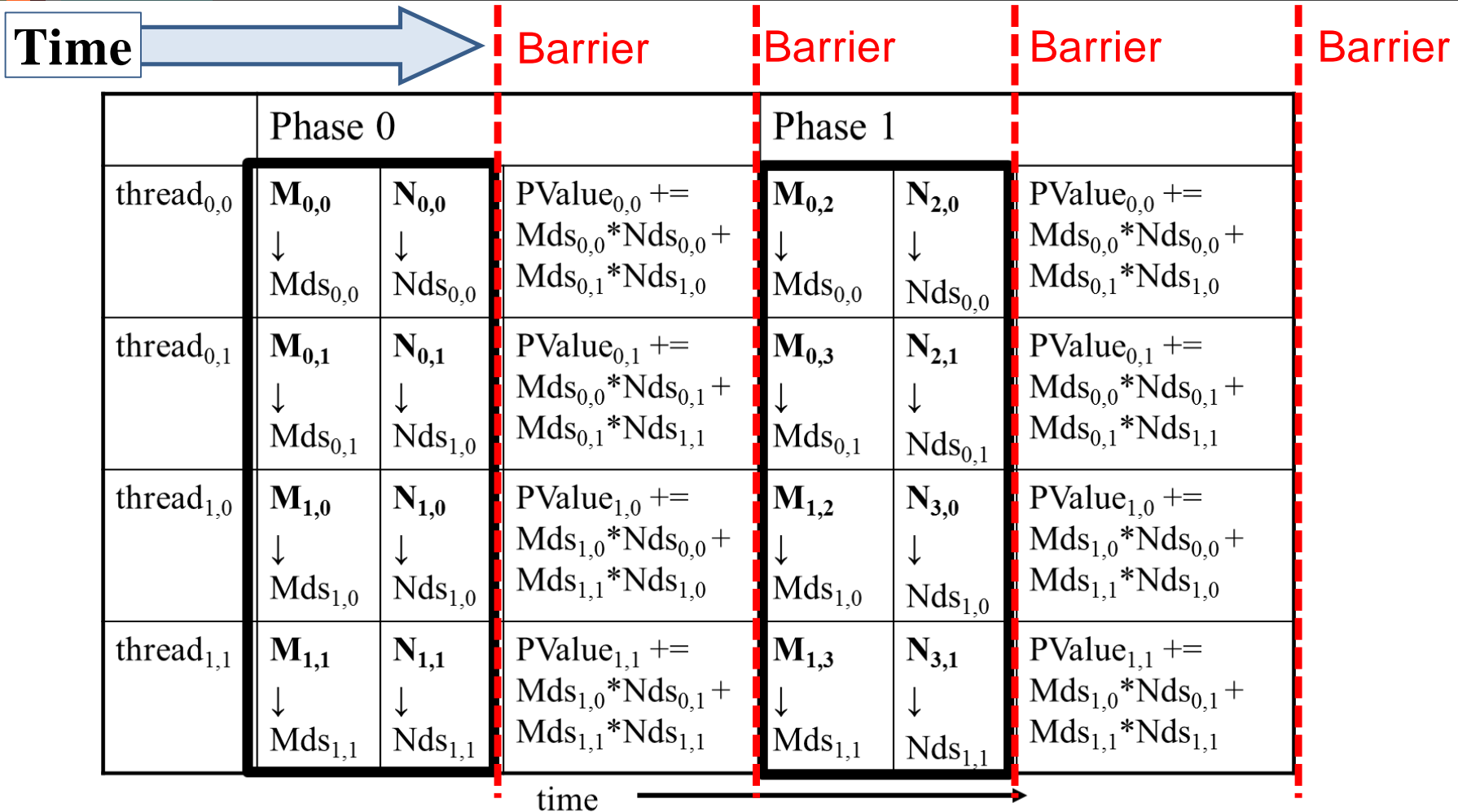
Execution Phases of Toy Example

Time 

	Phase 0			Phase 1		
thread _{0,0}	$\mathbf{M}_{0,0}$ ↓ $\mathbf{Mds}_{0,0}$	$\mathbf{N}_{0,0}$ ↓ $\mathbf{Nds}_{0,0}$	$\mathbf{PValue}_{0,0} +=$ $\mathbf{Mds}_{0,0} * \mathbf{Nds}_{0,0} +$ $\mathbf{Mds}_{0,1} * \mathbf{Nds}_{1,0}$	$\mathbf{M}_{0,2}$ ↓ $\mathbf{Mds}_{0,0}$	$\mathbf{N}_{2,0}$ ↓ $\mathbf{Nds}_{0,0}$	$\mathbf{PValue}_{0,0} +=$ $\mathbf{Mds}_{0,0} * \mathbf{Nds}_{0,0} +$ $\mathbf{Mds}_{0,1} * \mathbf{Nds}_{1,0}$
thread _{0,1}	$\mathbf{M}_{0,1}$ ↓ $\mathbf{Mds}_{0,1}$	$\mathbf{N}_{0,1}$ ↓ $\mathbf{Nds}_{1,0}$	$\mathbf{PValue}_{0,1} +=$ $\mathbf{Mds}_{0,0} * \mathbf{Nds}_{0,1} +$ $\mathbf{Mds}_{0,1} * \mathbf{Nds}_{1,1}$	$\mathbf{M}_{0,3}$ ↓ $\mathbf{Mds}_{0,1}$	$\mathbf{N}_{2,1}$ ↓ $\mathbf{Nds}_{0,1}$	$\mathbf{PValue}_{0,1} +=$ $\mathbf{Mds}_{0,0} * \mathbf{Nds}_{0,1} +$ $\mathbf{Mds}_{0,1} * \mathbf{Nds}_{1,1}$
thread _{1,0}	$\mathbf{M}_{1,0}$ ↓ $\mathbf{Mds}_{1,0}$	$\mathbf{N}_{1,0}$ ↓ $\mathbf{Nds}_{0,0}$	$\mathbf{PValue}_{1,0} +=$ $\mathbf{Mds}_{1,0} * \mathbf{Nds}_{0,0} +$	$\mathbf{M}_{1,2}$ ↓ $\mathbf{Mds}_{1,0}$	$\mathbf{N}_{3,0}$ ↓ $\mathbf{Nds}_{0,0}$	$\mathbf{PValue}_{1,0} +=$ $\mathbf{Mds}_{1,0} * \mathbf{Nds}_{0,0} +$

- In general, if an input matrix is of dimension Width and the tile size is TILE_WIDTH, the dot product would be performed in Width/TILE_WIDTH phases (num of blocks to process).
- 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 the amount of shared memory.

Execution Phases of Toy Example



- Both the data loading and calculation have to be Synchronized

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 for 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)

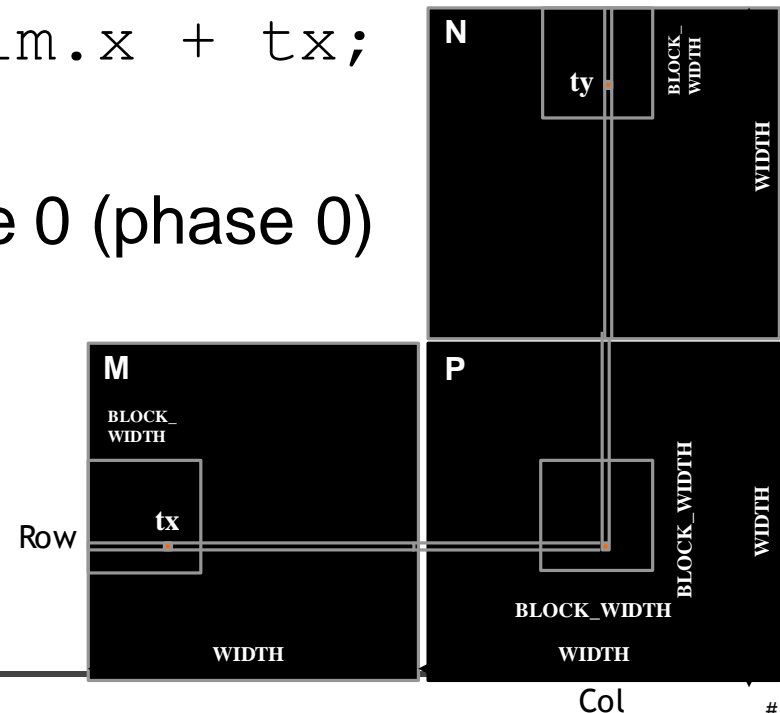
- Have each thread load an M element and an N element at the same relative position as its P element.

- `int Row = by * blockDim.y + ty;`
- `int Col = bx * blockDim.x + tx;`

- 2D indexing for accessing Tile 0 (phase 0)

`M[Row][tx]`

`N[ty][Col]`

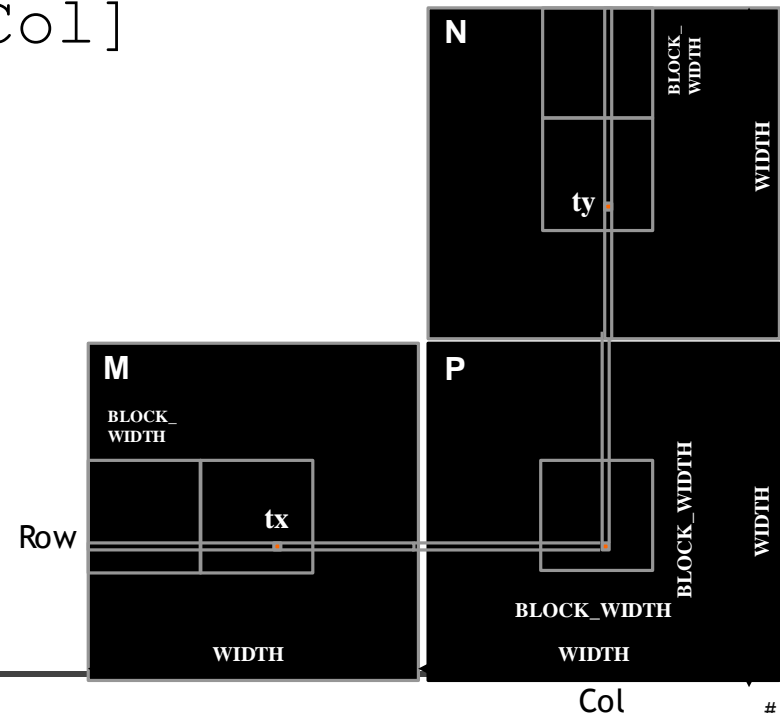


Loading Input Tile 1 of M (Phase 1)

- 2D indexing for accessing Tile 1 (phase 1)

`M[Row][1*TILE_WIDTH + tx]`

`N[1*TILE_WIDTH + ty][Col]`



Allocating M and N

- M and N can be allocated dynamically, using 1D indexing
 - $M[\text{Row}][\text{ph} * \text{TILE_WIDTH} + \text{tx}]$
 - $M[\text{Row} * \text{Width} + \text{ph} * \text{TILE_WIDTH} + \text{tx}]$
 - $N[\text{ph} * \text{TILE_WIDTH} + \text{ty}][\text{Col}]$
 - $N[(\text{ph} * \text{TILE_WIDTH} + \text{ty}) * \text{Width} + \text{Col}]$
- 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[(t*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];
```

Shared variables: block scope

```
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;
```

Automatic scalar variables: registers

```
// Loop over the M and N tiles required to compute the P element
```

iterates over phases

```
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[(t*TILE_WIDTH+ty)*Width + Col];
```

load into shared memory

```
__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;
```

```
}
```

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 40 gives $40 \times 40 = 1600$ threads
- For 16, in each phase, each block performs $2 \times 256 = 512$ float loads from global memory for $256 * (2 \times 16) = 8,192$ mul/add operations. (16 floating-point operations for each memory load)
- For 40, in each phase, each block performs $2 \times 1600 = 3,200$ float loads from global memory for $1600 * (2 \times 40) = 1,280,000$ mul/add operations. (40 floating-point operation for each memory load)

Shared Memory and Threading

- For an SM with 96 KB shared memory (GTX 1080)
 - Shared memory size is implementation dependent!
 - For `TILE_WIDTH = 16`, each thread block uses $2 \times 256 \times 4B = 2KB$ of shared memory.
 - For 96KB shared memory, one can potentially have up to 48 thread blocks executing on a SM (fully occupied)
 - This allows up to $48 \times 256 \times 2 = 24,576$ pending loads.
(2 per thread, 256 threads per block)
 - In case of `TILE_WIDTH 40` would lead to $2 \times 40 \times 40 \times 4 \text{ Byte} = 12.5KB$ shared memory usage per thread block, allowing 7 thread blocks ($1600 \times 7 = 11200$ threads) active at the same time
 - However, the thread count limitation of 2048 threads per SM in current GPUs will reduce the number of blocks per SM to one ! ($2048 - 1600 = 448$ threads unoccupied)
- Each `__syncthreads()` can reduce the number of active threads for a block

Memory Coalescing



Memory bandwidth

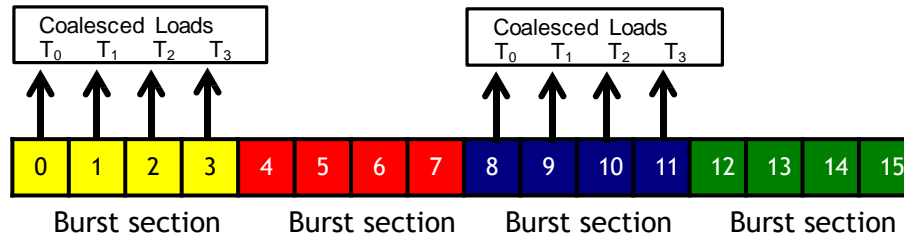
- One of the most important factors of CUDA kernel performance is accessing data in the global memory (a DRAM memory).
- The process of reading a the status of a single bit takes tenth of nanoseconds in modern DRAM chips. This is in sharp contrast with the sub-nanosecond clock cycle time of modern computing devices.
- Modern DRAMs use parallelism to increase their rate of data access: Each time a DRAM location is accessed, a range of consecutive locations that include the requested location are actually accessed (DRAM bursts).
- When all threads in a warp execute a load instruction, the hardware detects whether they access consecutive global memory locations. In this case, the hardware combines, or **coalesces**, all these accesses into a consolidated access to consecutive DRAM locations.
- Modern DRAM systems are designed to always be accessed in burst mode. Burst bytes are transferred to the processor but discarded when accesses are not to sequential locations.

DRAM Burst – A System View

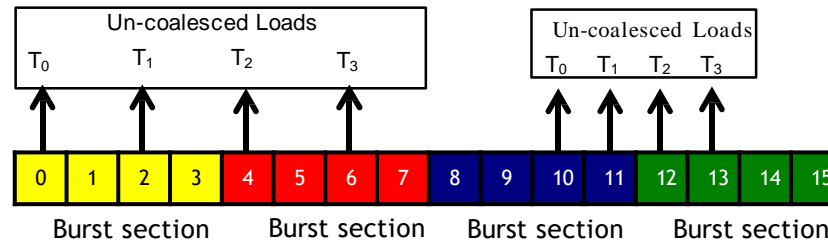


- Each address space is partitioned into burst sections
 - Whenever a location is accessed, all other locations in the same section are also delivered to the processor
- Basic example: a 16-byte address space, 4-byte burst sections
 - In practice, we have at least 4GB address space, burst section sizes of 128-bytes or more

Coalesced access

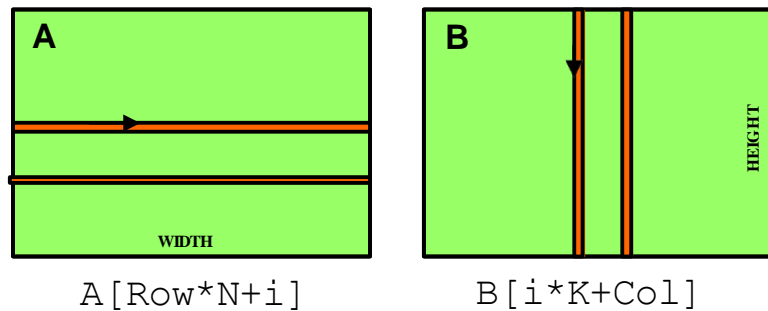


- When all threads of a warp execute a load instruction, if all accessed locations fall into the same burst section, only one DRAM request will be made and the access is fully coalesced.



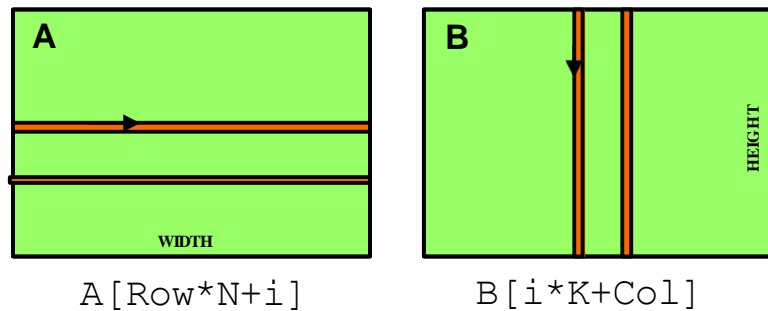
- When the accessed locations spread across burst section boundaries:
 - Coalescing fails
 - Multiple DRAM requests are made
 - The access is not fully coalesced.
- Some of the bytes accessed and transferred are not used by the threads

Two Access Patterns of Basic Matrix Multiplication



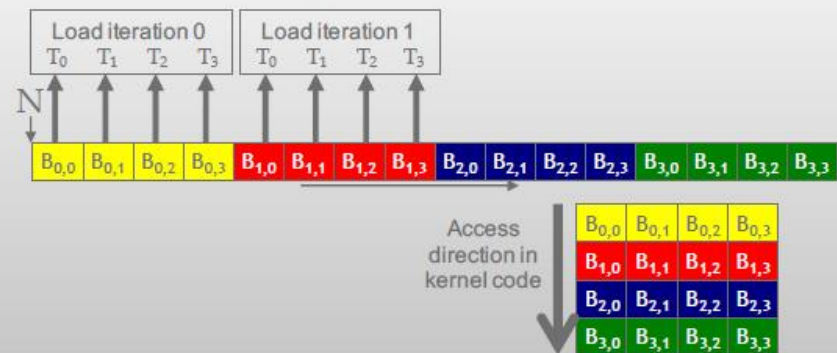
- i is the loop counter in the inner product loop of the kernel code
- A is $M \times N$, B is $N \times K$
- $\text{Col} = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}$

Two Access Patterns of Basic Matrix Multiplication

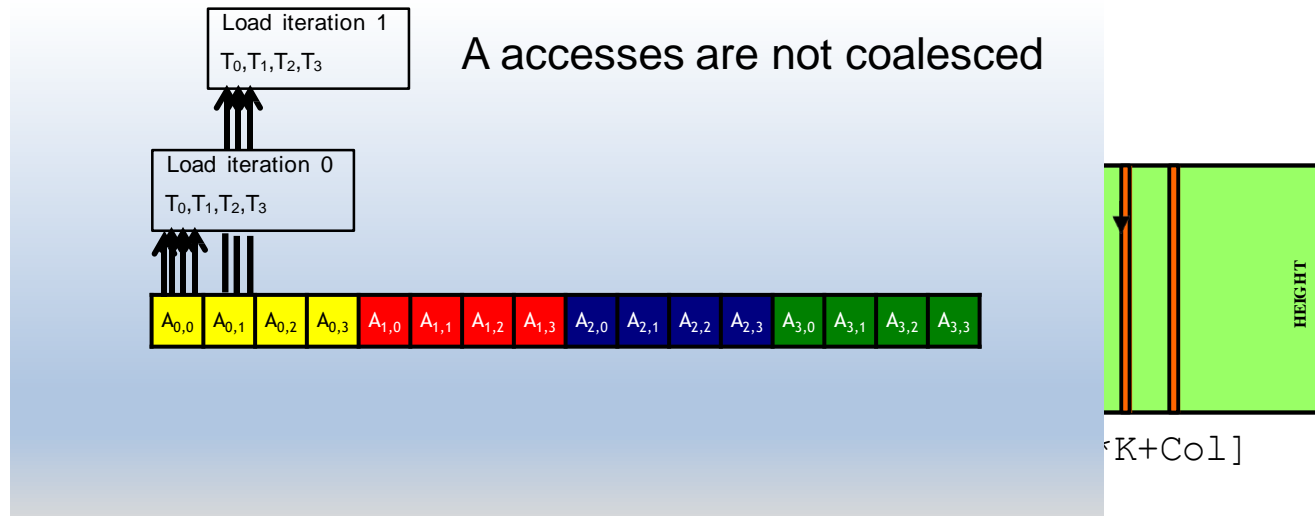


- i is the loop counter in the inner loop
- A is $M \times N$, B is $N \times K$
- $Col = blockIdx.x * blockDim.x + threadIdx.x$

B accesses are coalesced

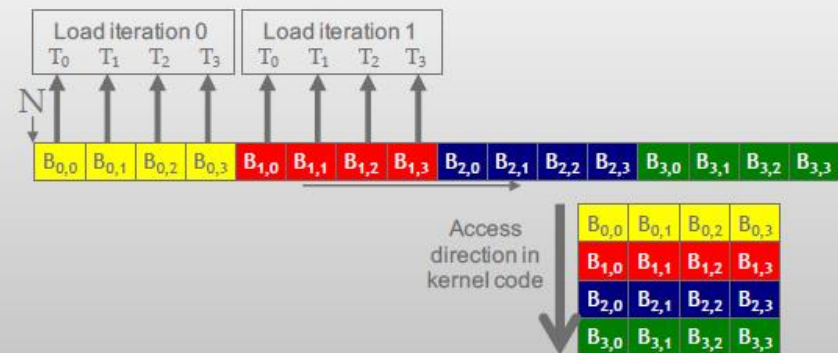


Two Access Patterns of Basic Matrix Multiplication



- i is the loop counter in the inner loop
- A is $M \times N$, B is $N \times K$
- Col = blockIdx.x * blockDim.x

B accesses are coalesced



Non coalesced read

- If an algorithm intrinsically requires a kernel code to iterate through data along the row direction, one can use the shared memory to enable memory coalescing
 -
 - The technique is called **corner turning**
 - A tiled algorithm can be used to enable coalescing
 - Once the data is in shared memory, they can be accessed either on a row basis or a column basis

Loading Data from Global Memory

```
int Row = by * blockDim.y + ty;
int Col = bx * blockDim.x + tx;

for (int ph = 0; ph < Width/TILE_WIDTH; ++ph)
{
    Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
    Nds[ty][tx] = N[(ph*TILE_WIDTH+ty)*Width + Col];

    ...
}
```

For `Mds`, adjacent `threadIdx.x` threads will access adjacent `M` elements.
also,
For `Nds`, adjacent `threadIdx.x` threads will access adjacent `N` elements.

→ Both data loads are all coalesced,
hence extra benefit for using Shared memory.

Boundary Checks



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 square matrices with an arbitrary width.

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

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

N

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

Shared Memory

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

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

M

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

Shared Memory

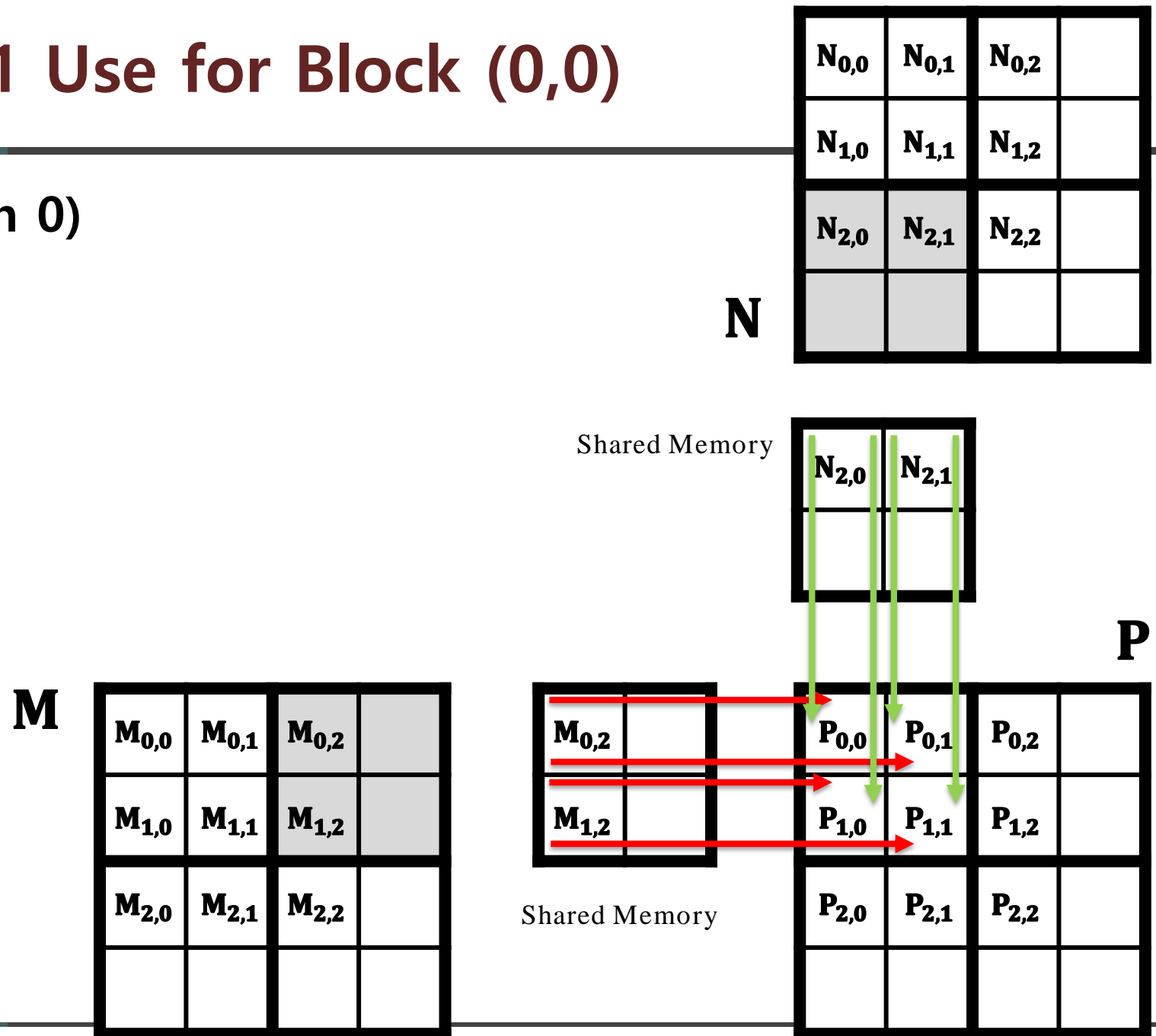
$M_{0,2}$	
$M_{1,2}$	

P

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

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.

M

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

N

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

Shared Memory

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

P

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

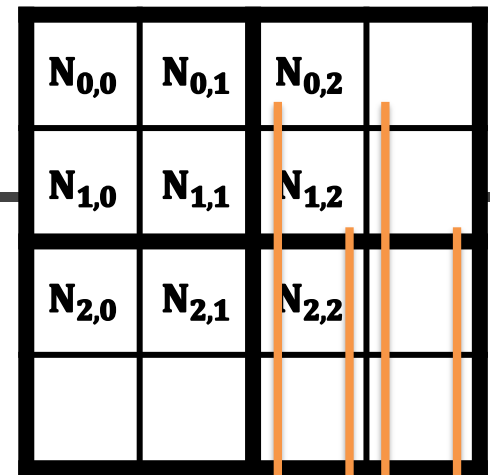
Shared Memory

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

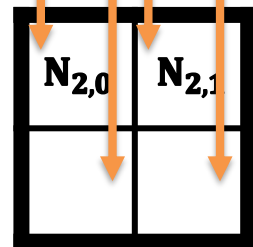
Phase 0 Loads for Block (1,1) for a 3x3

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

N



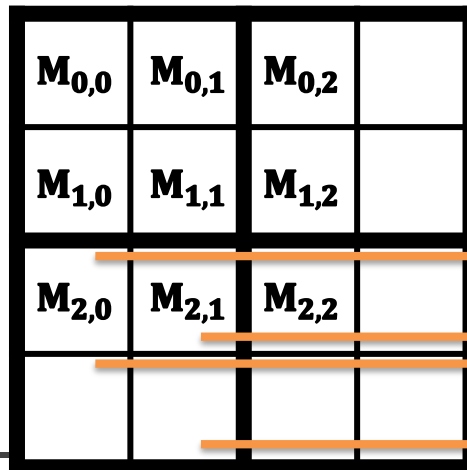
Shared Memory



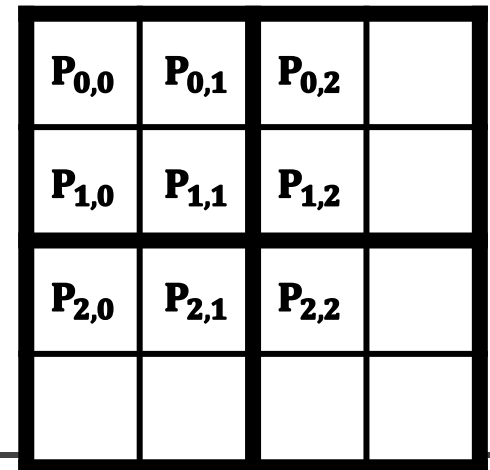
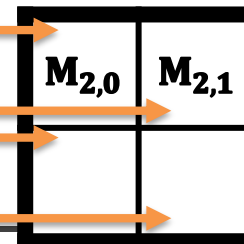
P

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

M



Shared Memory



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

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

N

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

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

M

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

Shared Memory

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

P

Shared Memory

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

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

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 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)

M

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

N

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

Shared Memory

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

P

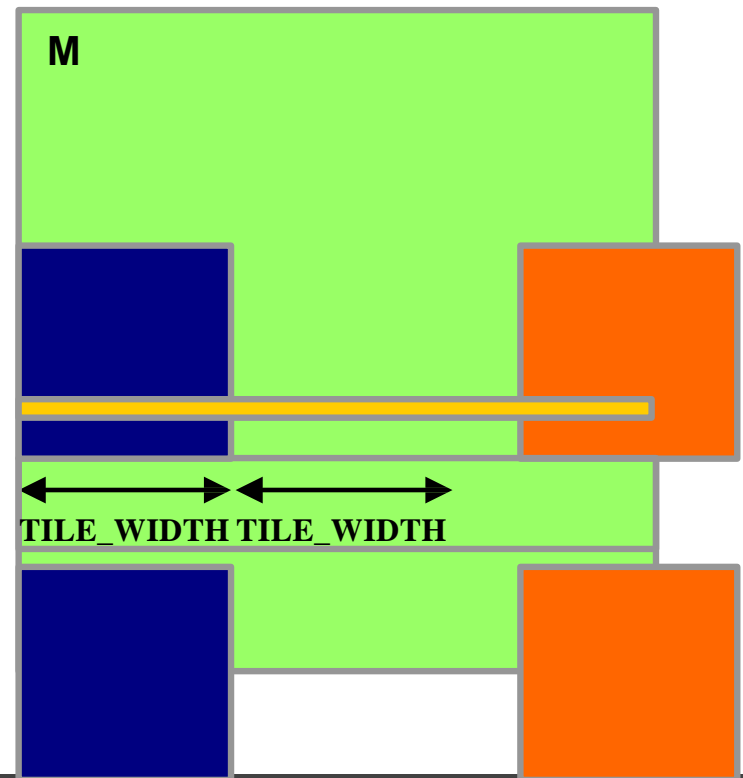
$M_{0,2}$	0
$M_{1,2}$	0

Shared Memory

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

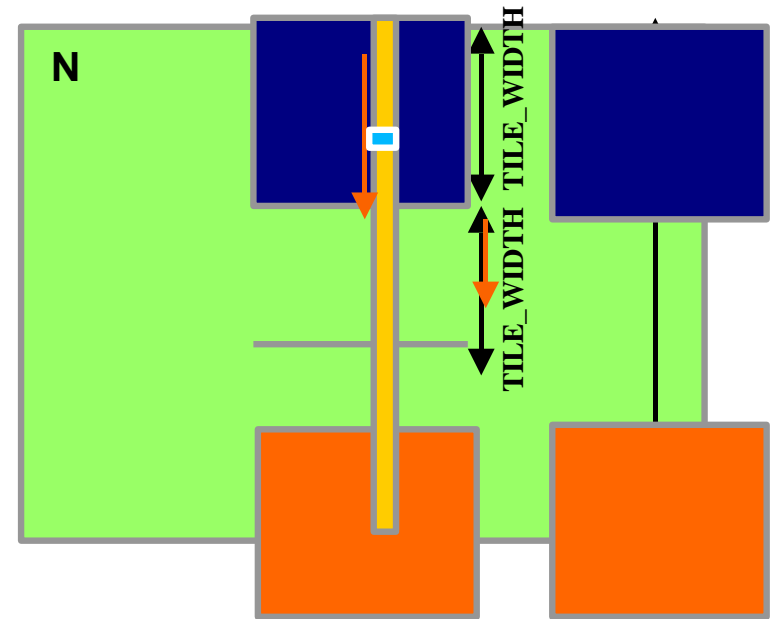
Boundary Condition for Input M Tile

- Each thread loads
 - $M[\text{Row}][\text{ph} * \text{TILE_WIDTH} + \text{tx}]$
 - $M[\text{Row} * \text{Width} + \text{ph} * \text{TILE_WIDTH} + \text{tx}]$
- Need to test
 - $(\text{Row} < \text{Width}) \ \&\& \ (\text{ph} * \text{TILE_WIDTH} + \text{tx} < \text{Width})$
 - If true, load M element
 - Else , load 0



Boundary Condition for Input N Tile

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



Practice: Tiled Matrix Multiplication

- **Copy Sample Skeleton Code**
 - `cp -r /home/share/18_Shared ./[FolderName]`
 - `cd [FolderName]`
- **Notepad: MatrixMulti.cu 코드 Kernel function 완성**
- **Compile & run program**
 - `make`
 - `./EXE`

Matrix Multiplication – with boundary check

```
//Continued from Tiled Matrix Multiplication Kernel
for (int ph = 0; ph < NUM_PHASE; ++ph) {
    if(Row < Width && ph*TILE_WIDTH + tx < Width) {
        Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
    }else{
        Mds[ty][tx] = 0.0;
    }
    if(ph*TILE_WIDTH + ty < Width && Col < Width){
        Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*Width + Col];
    }else{
        Nds[ty][tx] = 0.0;
    }
    __syncthreads();

    if(Row < Width && Col < Width) {
        for (int i = 0; i < TILE_WIDTH; ++i) {
            Pvalue += Mds[ty][i] * Nds[i][tx];
        }
    }
    __syncthreads();
}
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 for P
- 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

Assignment #4

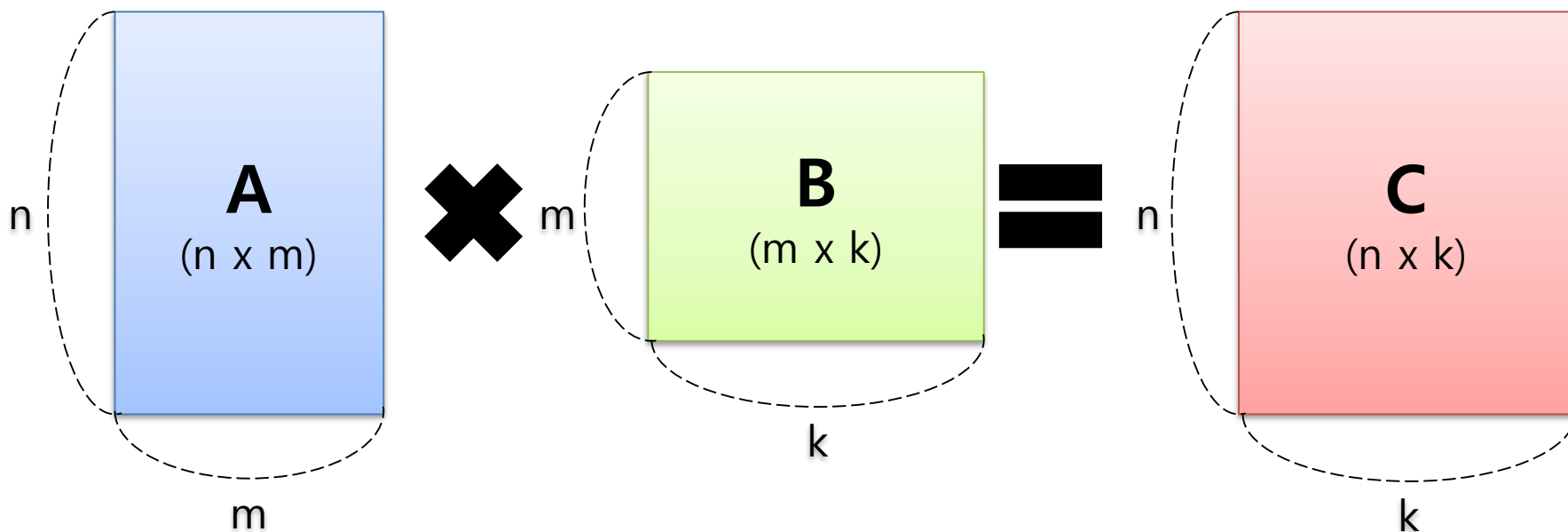


Purpose of Assignment

- **Purpose**

- Programming “Matrix Multiplication Program” on the GPU
 - Skeleton Code is given.
 - Skeleton Code is in the GPU Server.
Path: /home/share/Assignment4/
 - Requires Arbitrary Sized Matrices
 - Requires Matrix Tiling
 - Compile and Run on the GPU Server
 - Print Results to Text File

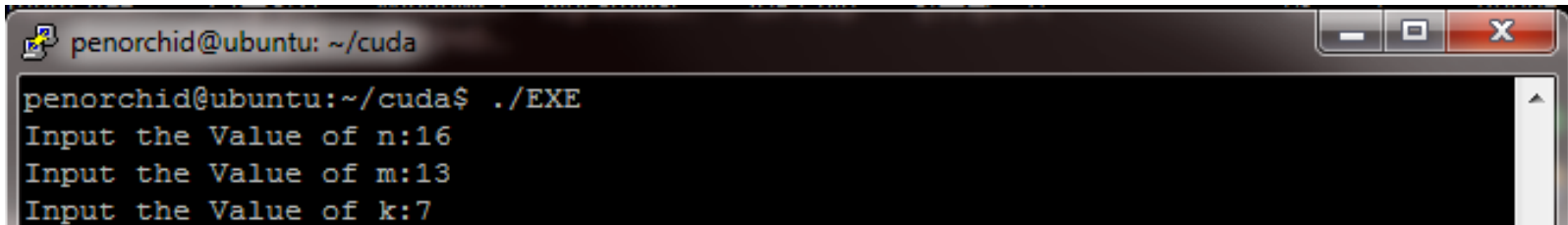
Arbitrary size Matrix Multiplication



- **Implement routine to define Matrix Size**
 - n , m and k
- **Input Elements of A and B are Random Variables**
 - Integers in a range $(-10, 10)$

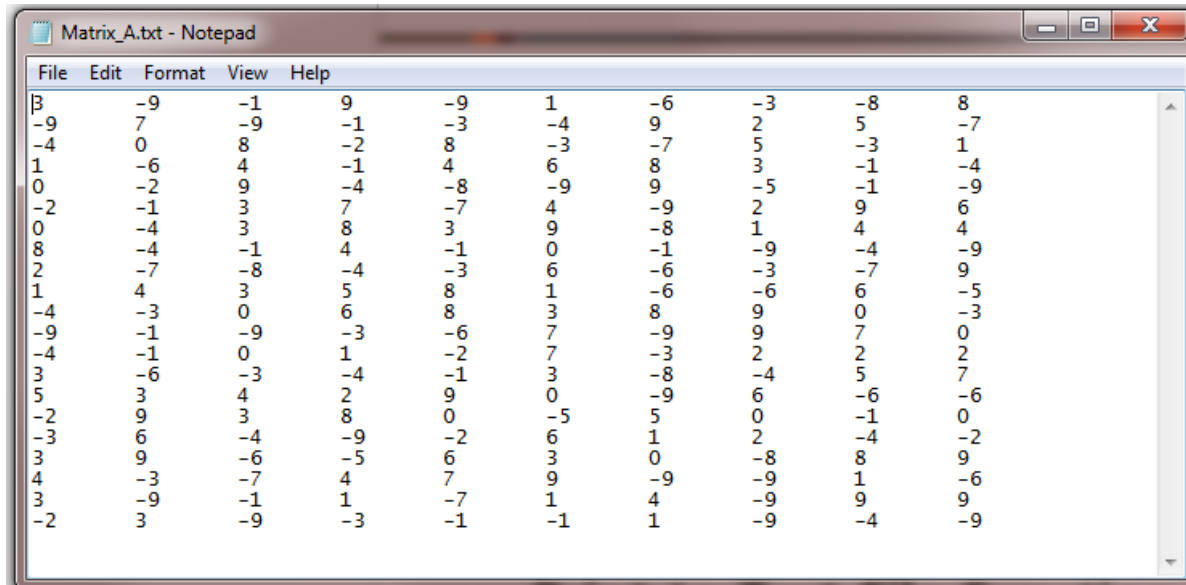
Parameters input

- Input Parameters with Console Commands
 - Matrix Size



```
penorchid@ubuntu: ~/cuda
penorchid@ubuntu:~/cuda$ ./EXE
Input the Value of n:16
Input the Value of m:13
Input the Value of k:7
```

Results Print



The screenshot shows a Notepad window with the title 'Matrix_A.txt - Notepad'. The menu bar includes 'File', 'Edit', 'Format', 'View', and 'Help'. The text area contains a 10x10 matrix of integers, with each row starting with a letter from 'B' to 'S' in the first column. The matrix is as follows:

	B	C	D	E	F	G	H	I	J
B	-9	-1	9	-9	1	-6	-3	-8	8
C	-9	7	-9	-1	-3	-4	9	2	5
D	-4	0	8	-2	8	-3	-7	5	-3
E	1	-6	4	-1	4	6	8	3	-1
F	0	-2	9	-4	-8	-9	9	-5	-1
G	-2	-1	3	7	-7	4	-9	2	9
H	0	-4	3	8	3	9	-8	1	4
I	8	-4	-1	4	-1	0	-1	-9	-4
J	2	-7	-8	-4	-3	6	-6	-3	-7
K	1	4	3	5	8	1	-6	-6	6
L	-4	-3	0	6	8	3	8	9	0
M	-9	-1	-9	-3	-6	7	-9	9	7
N	-4	-1	0	1	-2	7	-3	2	2
O	3	-6	-3	-4	-1	3	-8	-4	5
P	5	3	4	2	9	0	-9	6	-6
Q	-2	9	3	8	0	-5	5	0	-1
R	-3	6	-4	-9	-2	6	1	2	-4
S	3	9	-6	-5	6	3	0	-8	8
T	4	-3	-7	4	7	9	-9	-9	1
U	3	-9	-1	1	-7	1	4	-9	9
V	-2	3	-9	-3	-1	-1	1	-9	-4

- Routine for File I/O is given.
- Write Matrices A,B and C.
 - 1 text file per Matrix

Submit the Assignment

- **Submit the zip file @ Blackboard**
 - File name must be "Assignment4_StudentID_Name.zip"
 - Ex. Assignment4_2015000000_박지혁.zip
 - Zip file must include
 - Src files
 - Result running Image file
 - Result printed text files

History of Matrix Multiplication



Previous Matrix Multiplication @ CPU

C Function

```
void matrixMult (int a[N][N], int b[N][N], int c[N][N], int width)
{
    for (int i = 0; i < width; i++) {
        for (int j = 0; j < width; j++) {
            int sum = 0;
            for (int k = 0; k < width; k++) {
                int m = a[i][k];
                int n = b[k][j];
                sum += m * n;
            }
            c[i][j] = sum;
        }
    }
}
```

N

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

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}

P

P_{0,0}	P_{0,1}	P_{0,2}	P_{0,3}
P_{1,0}	P_{1,1}	P_{1,2}	P_{1,3}
P_{2,0}	P_{2,1}	P_{2,2}	P_{2,3}
P_{3,0}	P_{3,1}	P_{3,2}	P_{3,3}

Previous Matrix Multiplication @ GPU

CUDA Kernel

```
__global__ void matrixMult (int *a, int *b, int *c, int width) {  
    int k, sum = 0;  
  
    int col = threadIdx.x + blockDim.x * blockIdx.x;  
    int row = threadIdx.y + blockDim.y * blockIdx.y;  
  
    if(col < width && row < width) {  
        for (k = 0; k < width; k++)  
            sum += a[row * width + k] * b[k * width + col];  
        c[row * width + col] = sum;  
    }  
}
```

M

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

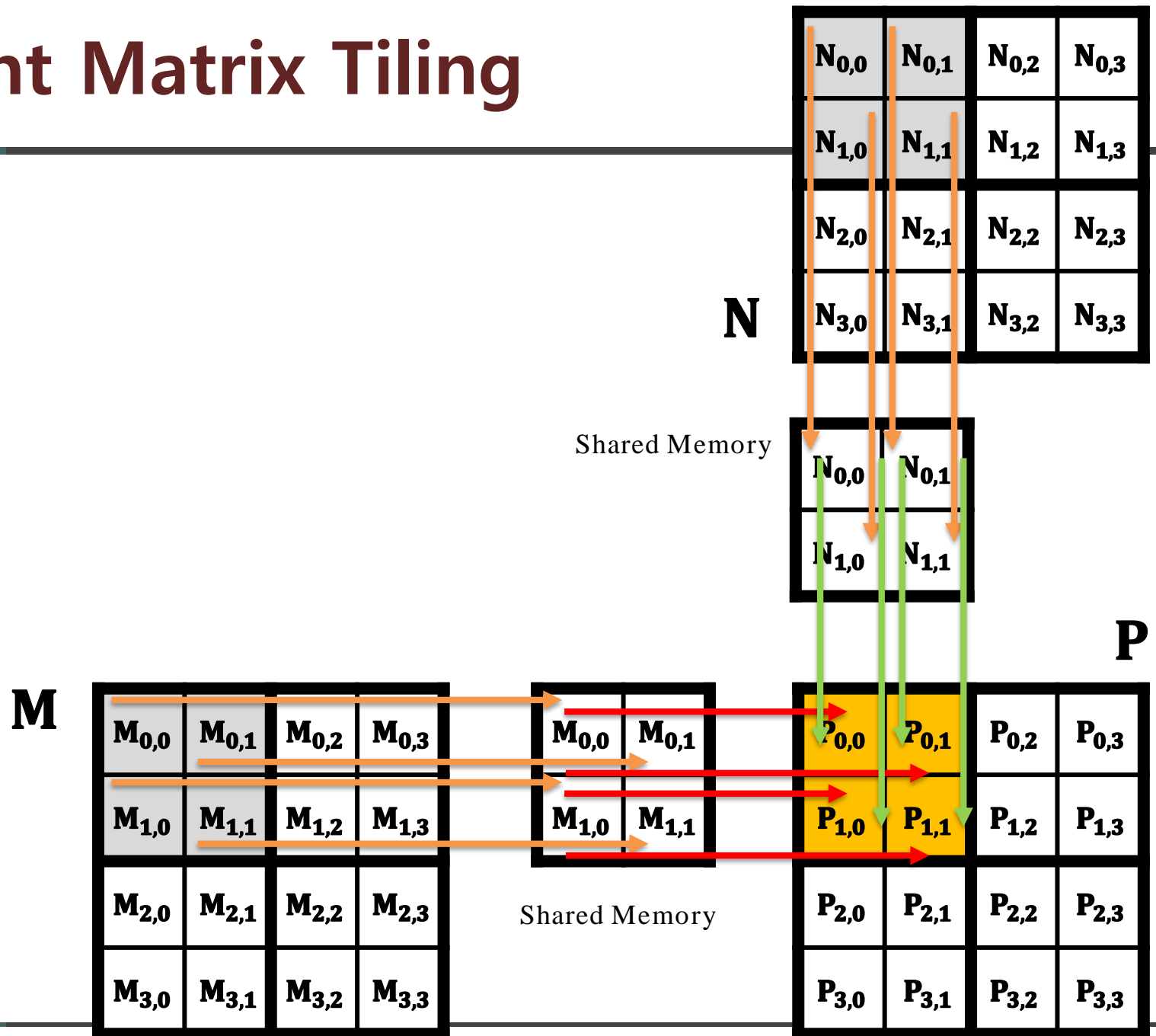
N

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

P

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

Current Matrix Tiling

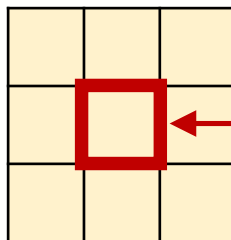


Tiling Animation



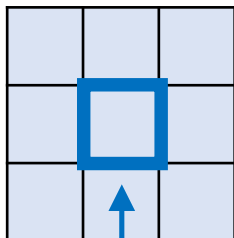
Phase #1

Nds

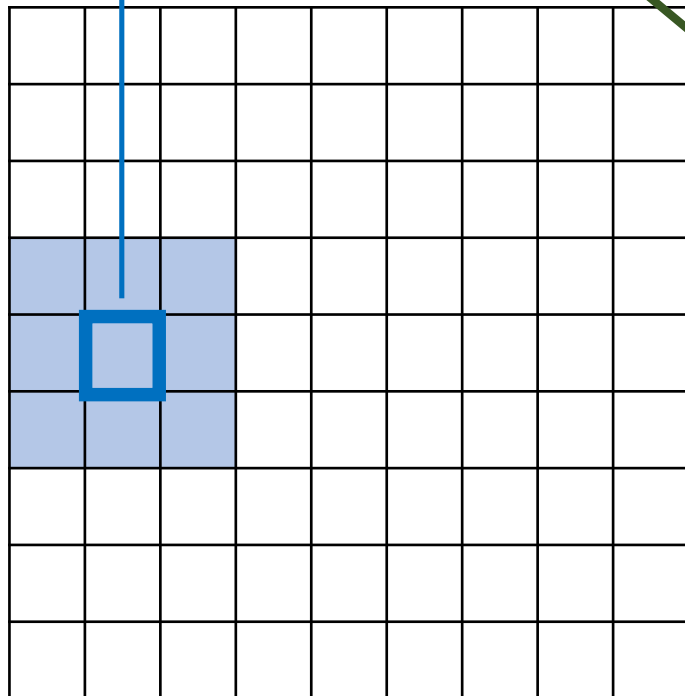


```
row = by * blockDim.y + ty;  
col = bx * blockDim.x + tx;
```

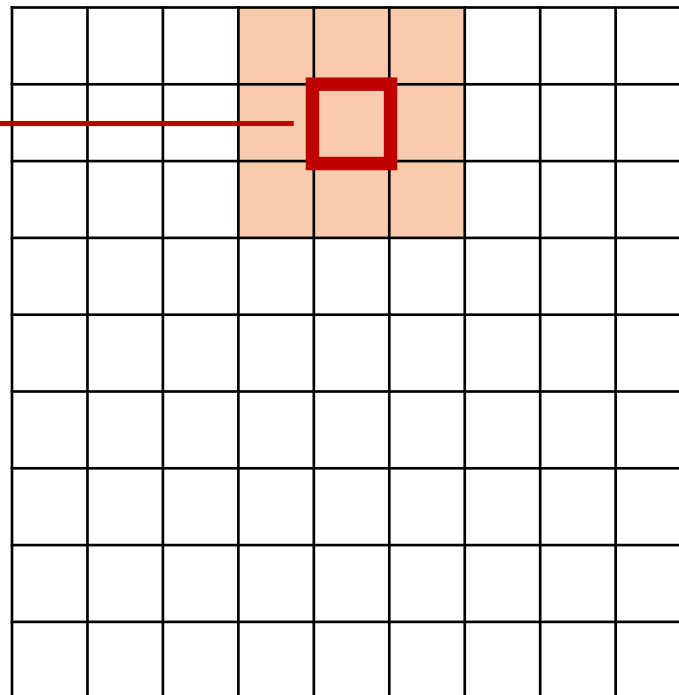
Mds



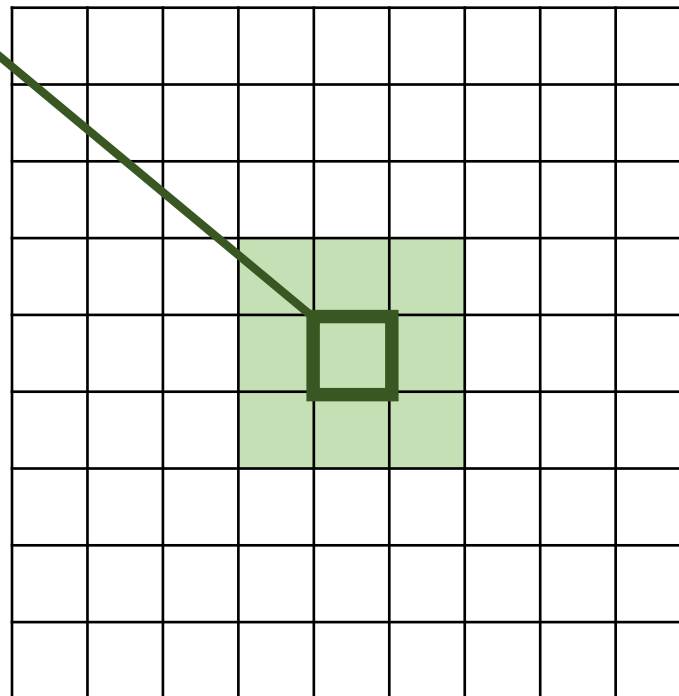
M



N

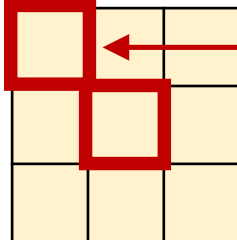


P

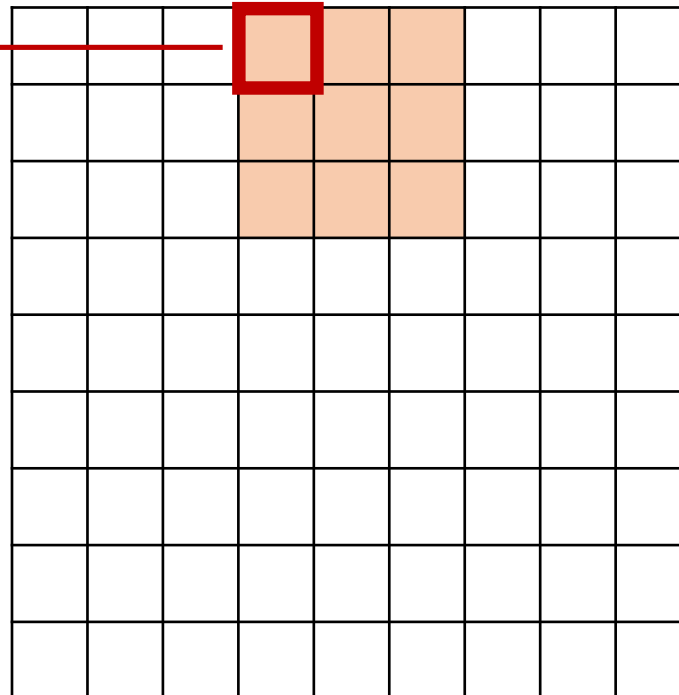


Phase #1

Nds

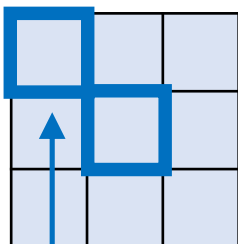


```
row = by * blockDim.y + ty;  
col = bx * blockDim.x + tx;
```

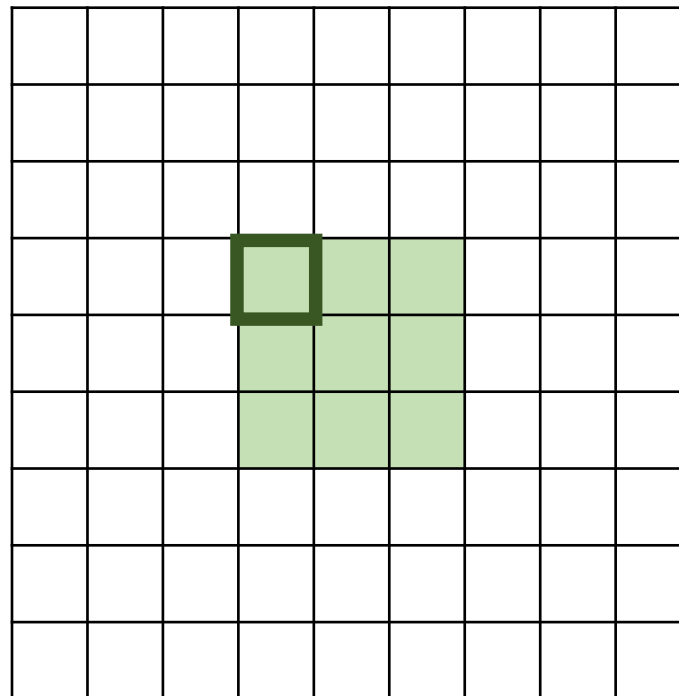
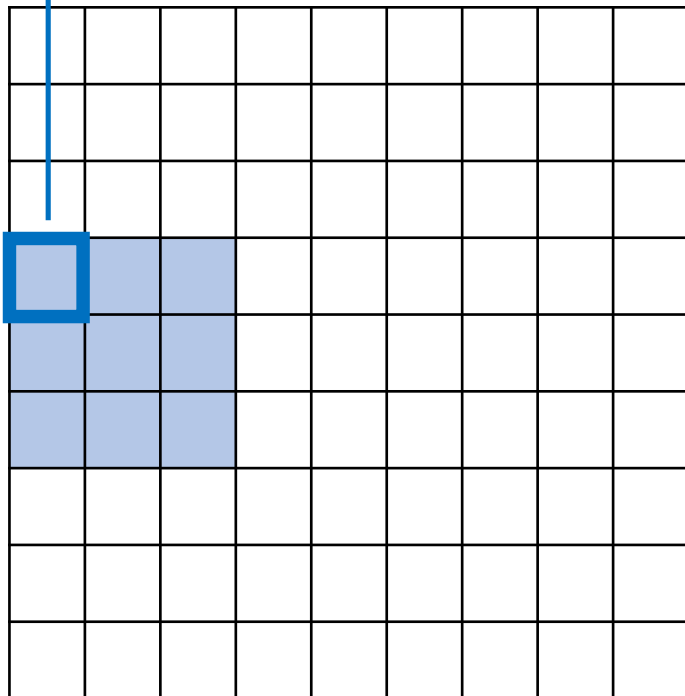


N

Mds



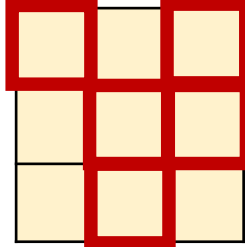
M



P

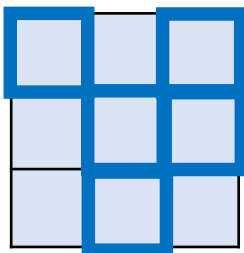
Phase #1

Nds

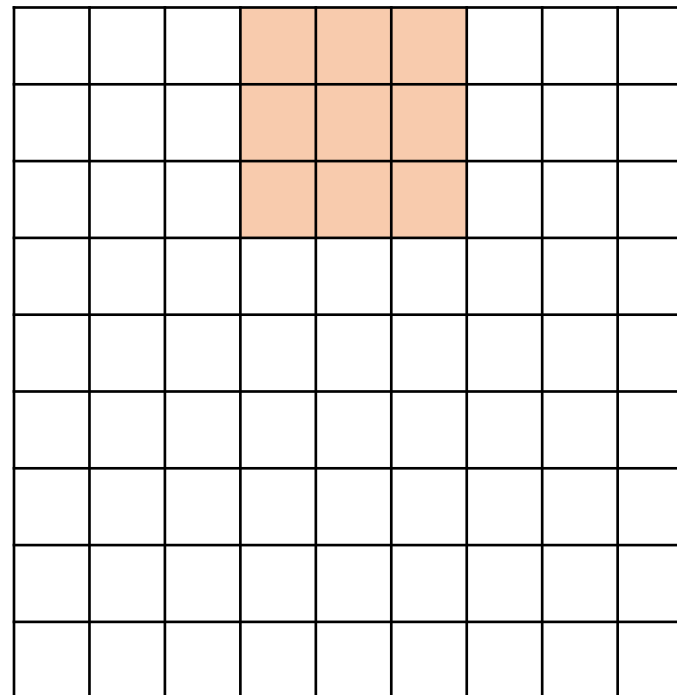
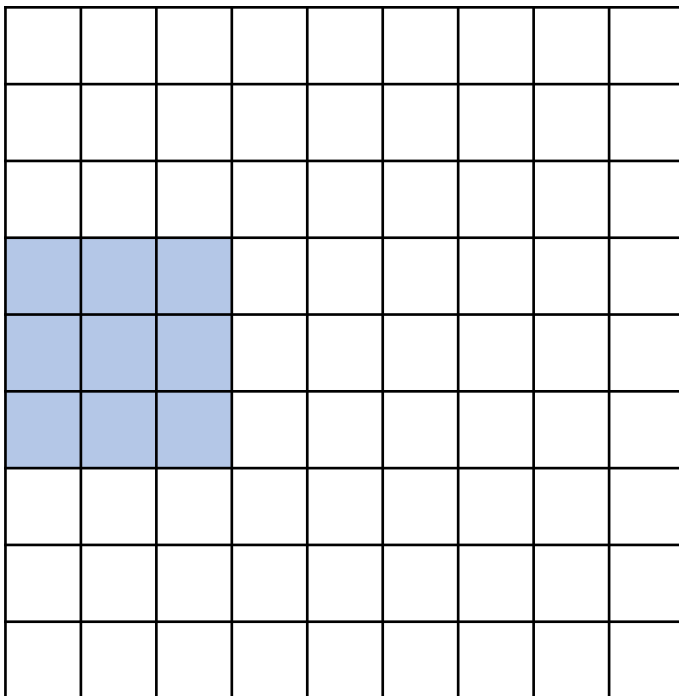


```
row = by * blockDim.y + ty;  
col = bx * blockDim.x + tx;
```

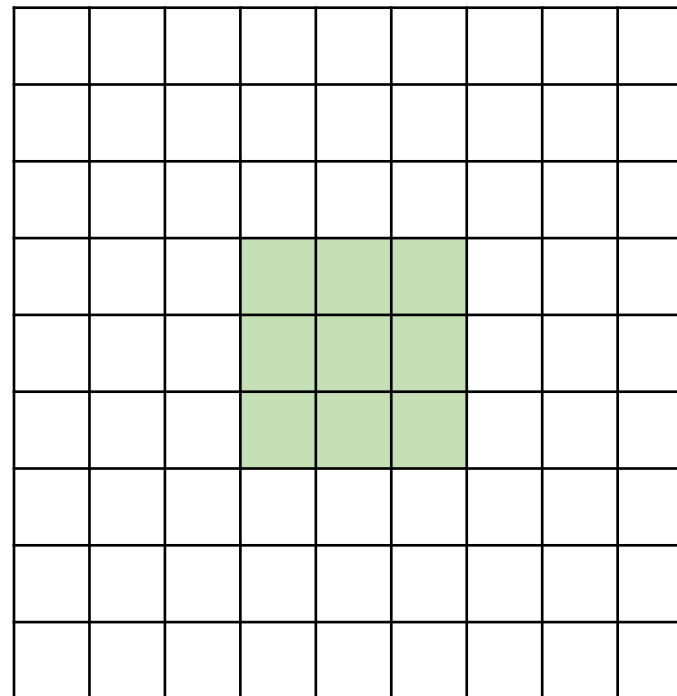
Mds



M



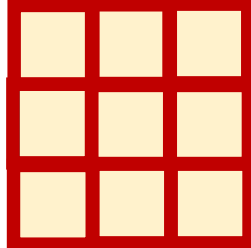
N



P

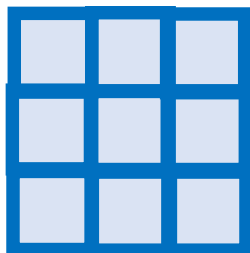
Phase #1

Nds

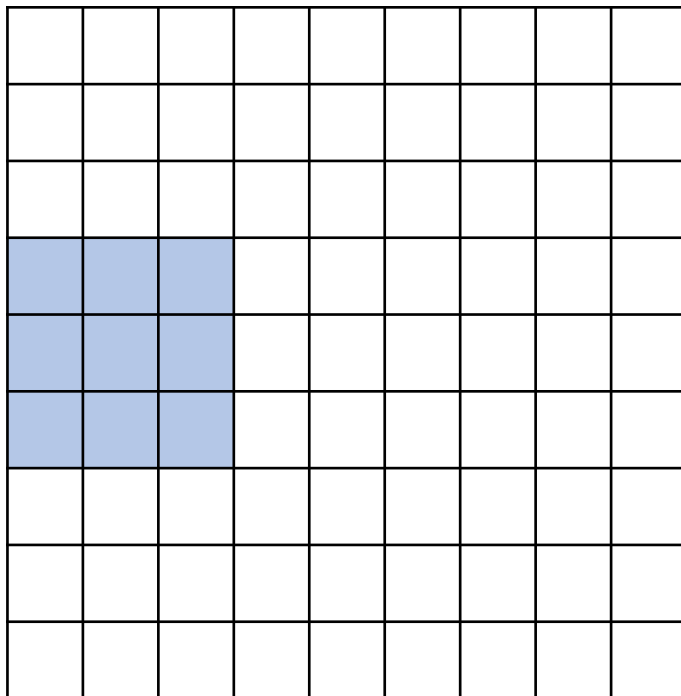


```
row = by * blockDim.y + ty;  
col = bx * blockDim.x + tx;
```

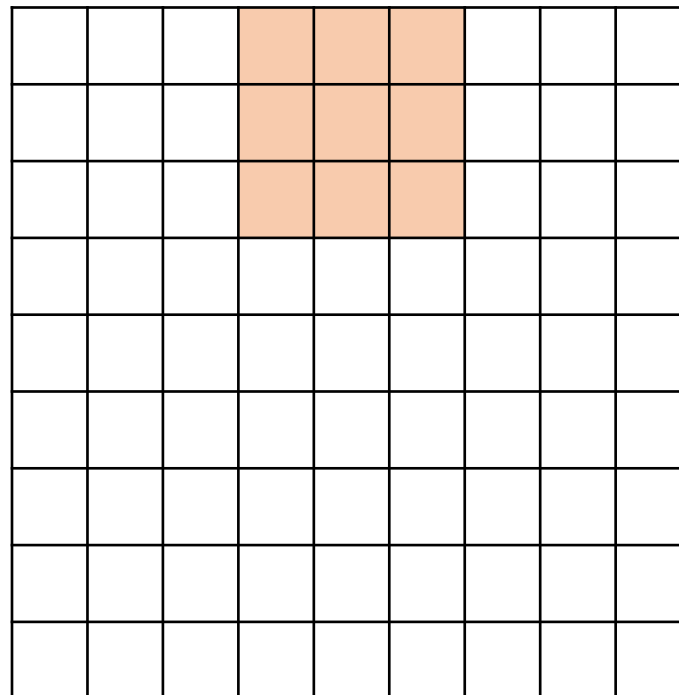
Mds



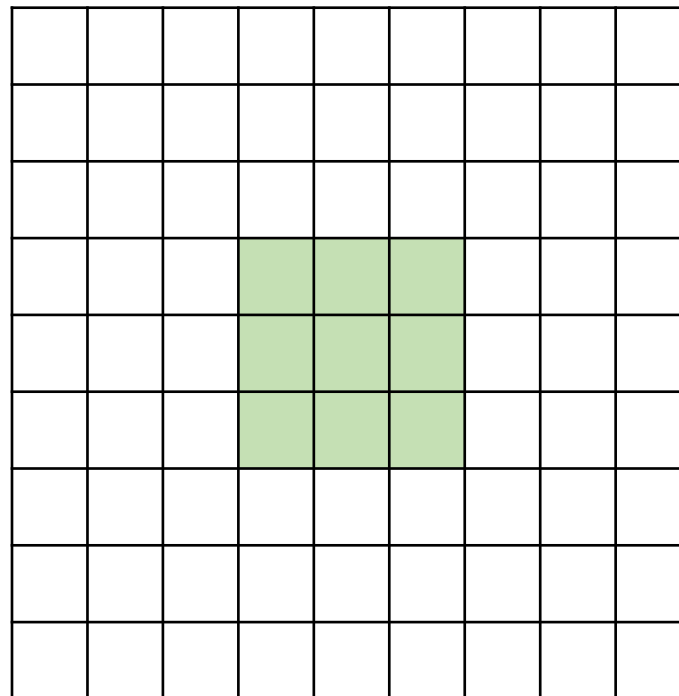
M



N



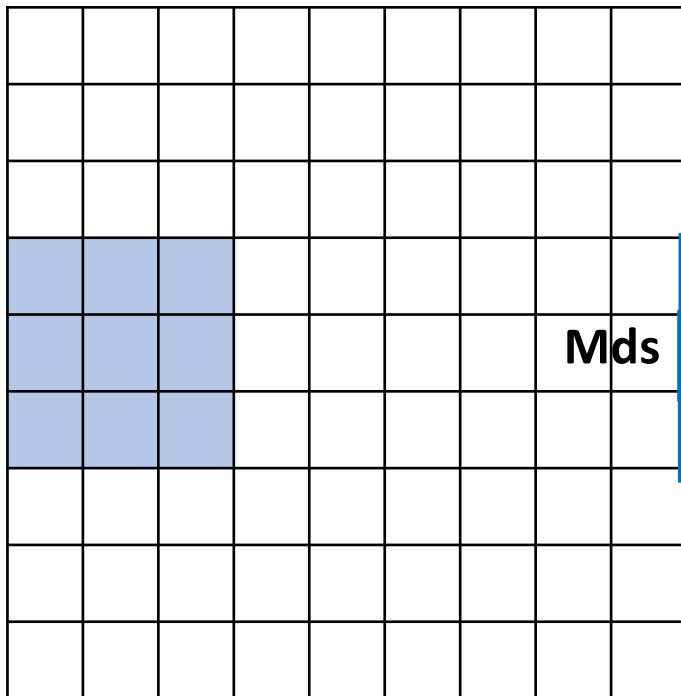
P



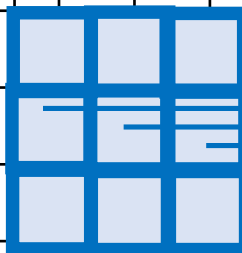
Phase #1

```
row = by * blockDim.y + ty;  
col = bx * blockDim.x + tx;
```

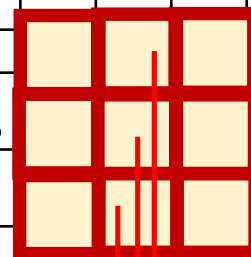
M



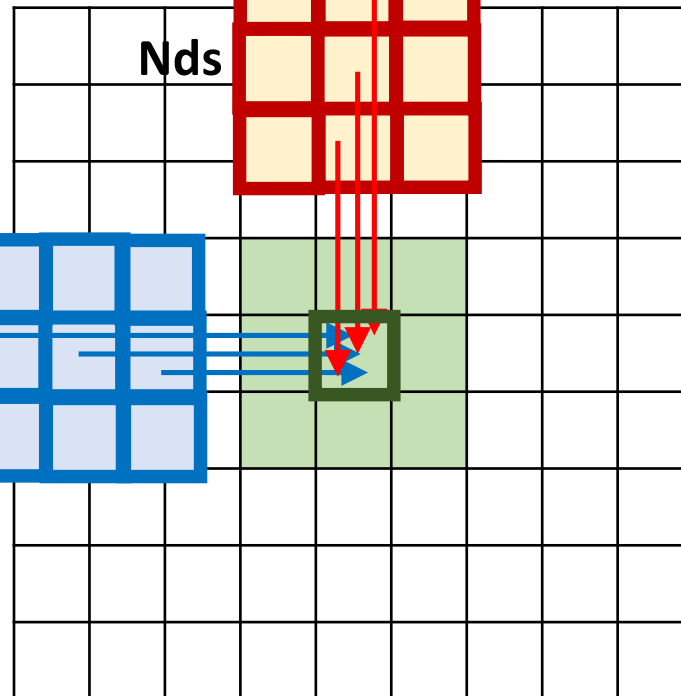
Mds



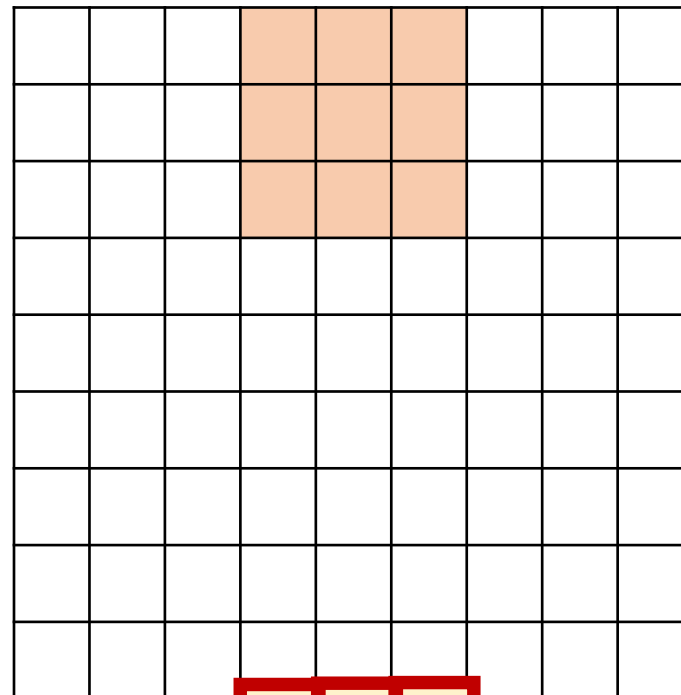
Nds



P



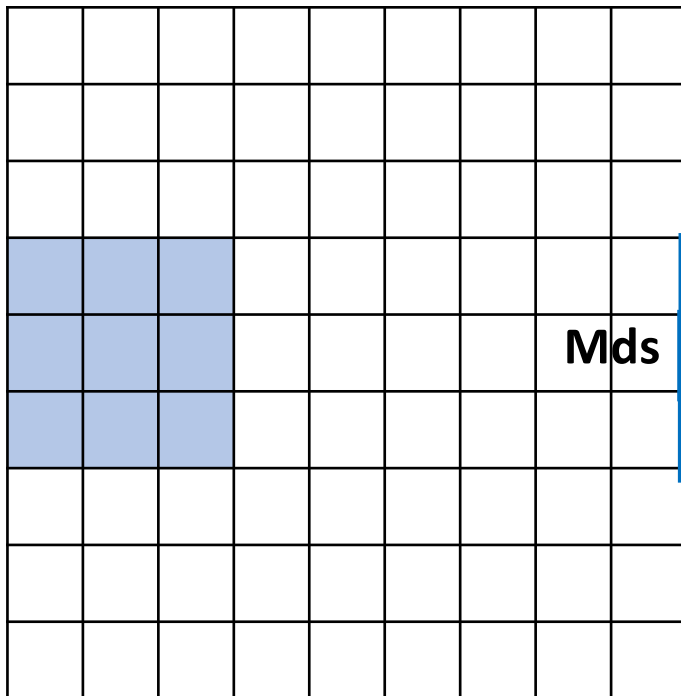
N



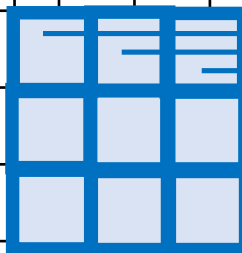
Phase #1

```
row = by * blockDim.y + ty;  
col = bx * blockDim.x + tx;
```

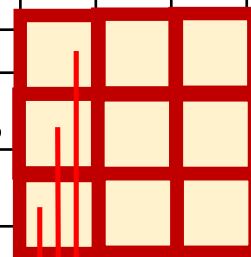
M



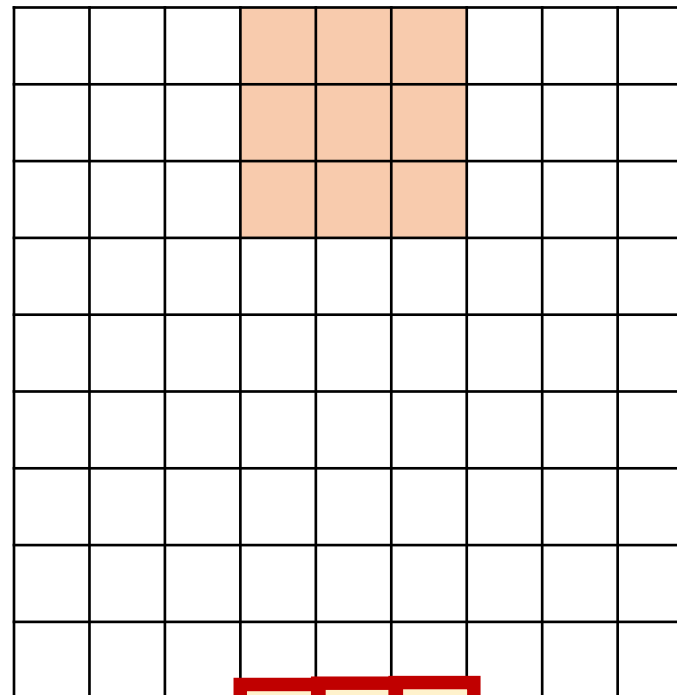
Mds



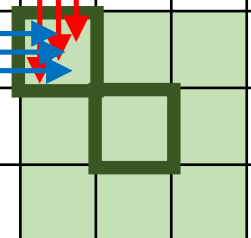
Nds



N



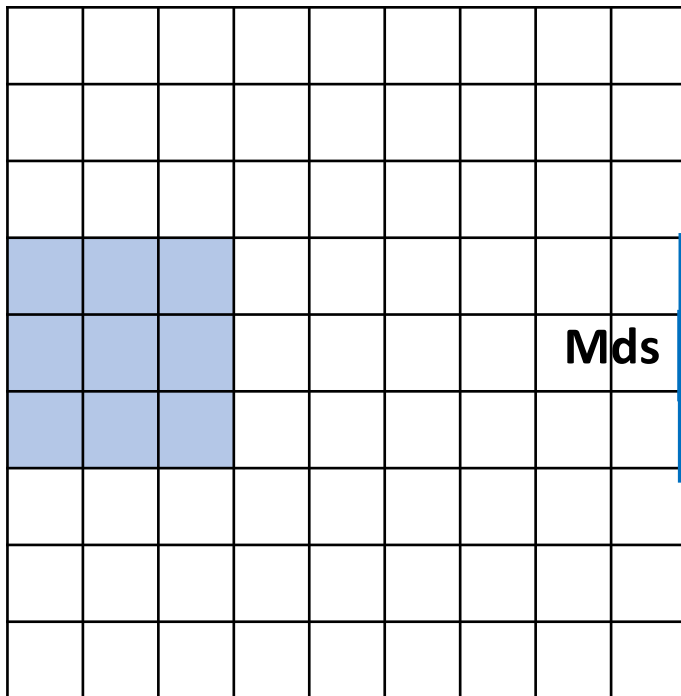
P



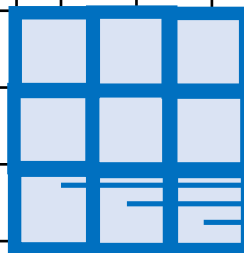
Phase #1

```
row = by * blockDim.y + ty;  
col = bx * blockDim.x + tx;
```

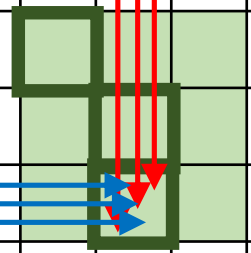
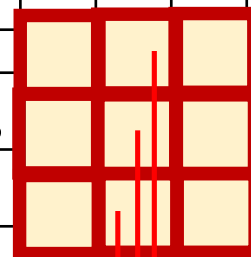
M



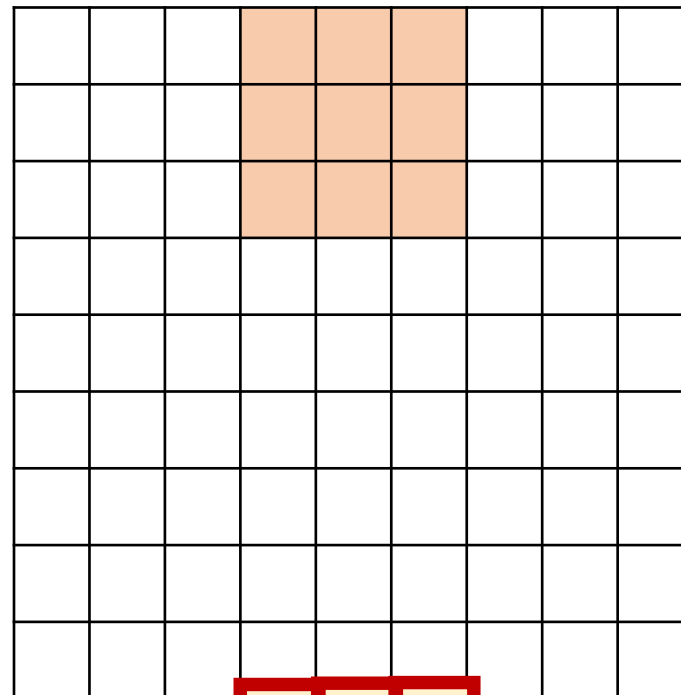
Mds



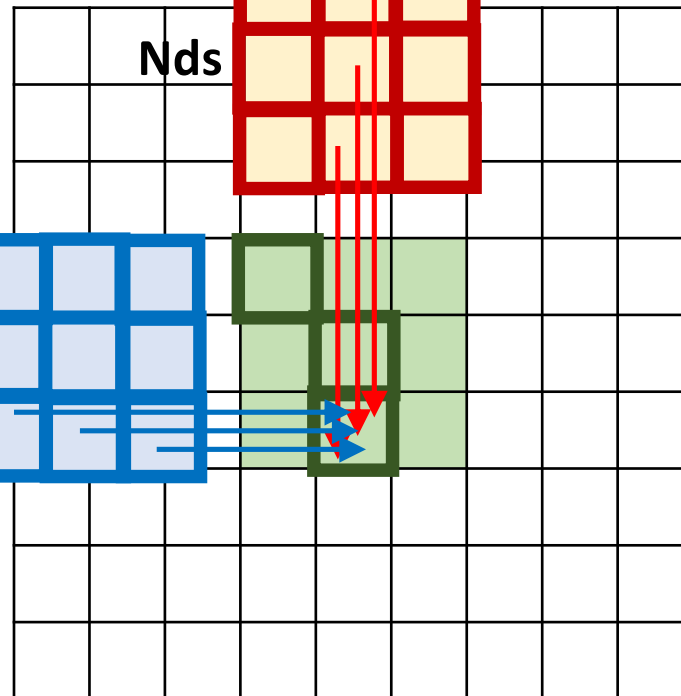
Nds



N



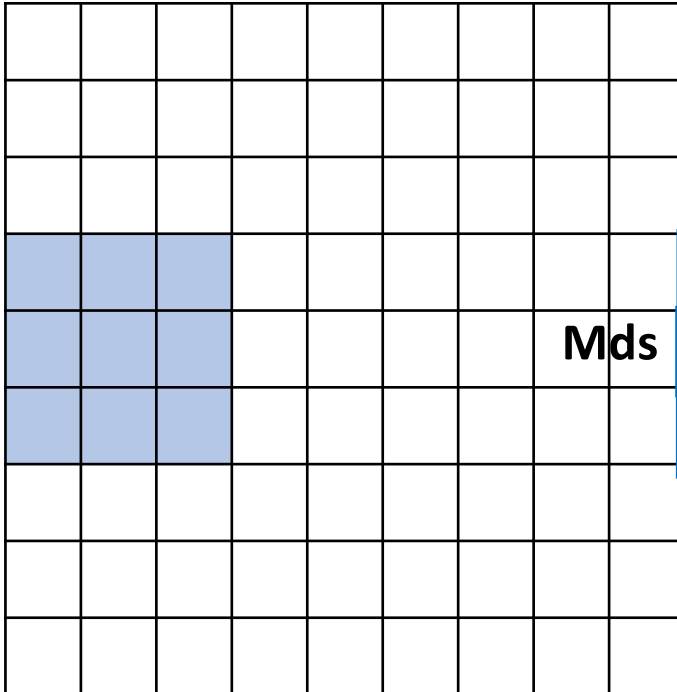
P



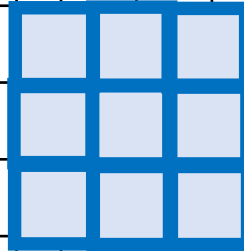
Phase #1

```
row = by * blockDim.y + ty;  
col = bx * blockDim.x + tx;
```

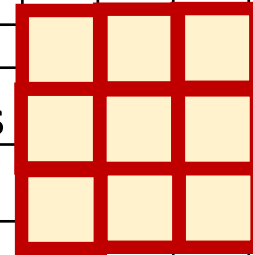
M



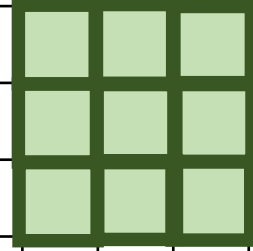
Mds



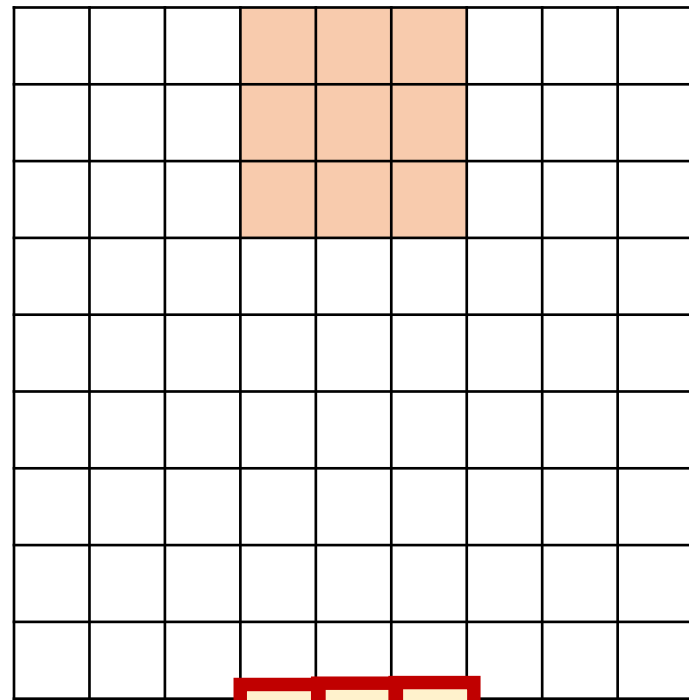
Nds



P

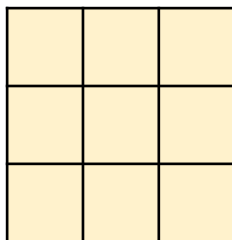


N



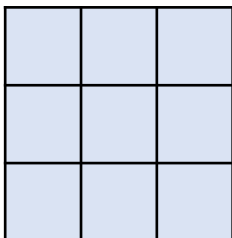
Phase #1

Nds

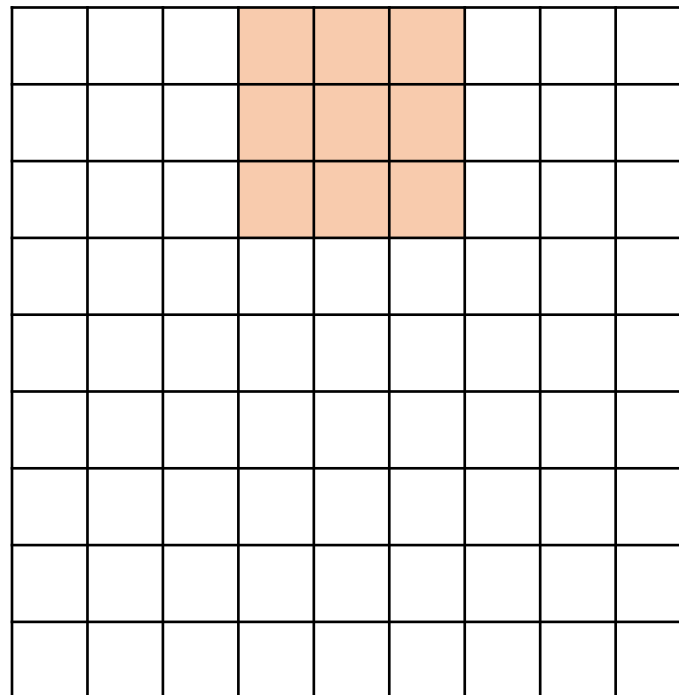
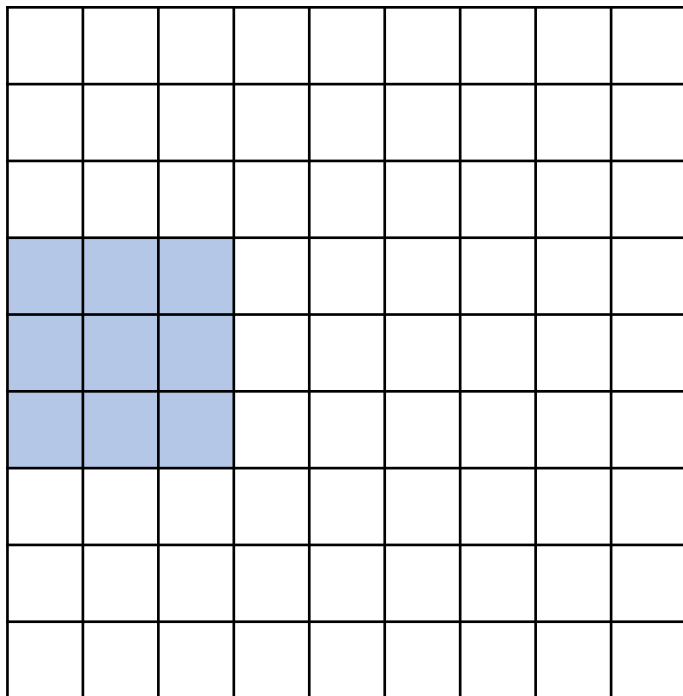


```
row = by * blockDim.y + ty;  
col = bx * blockDim.x + tx;
```

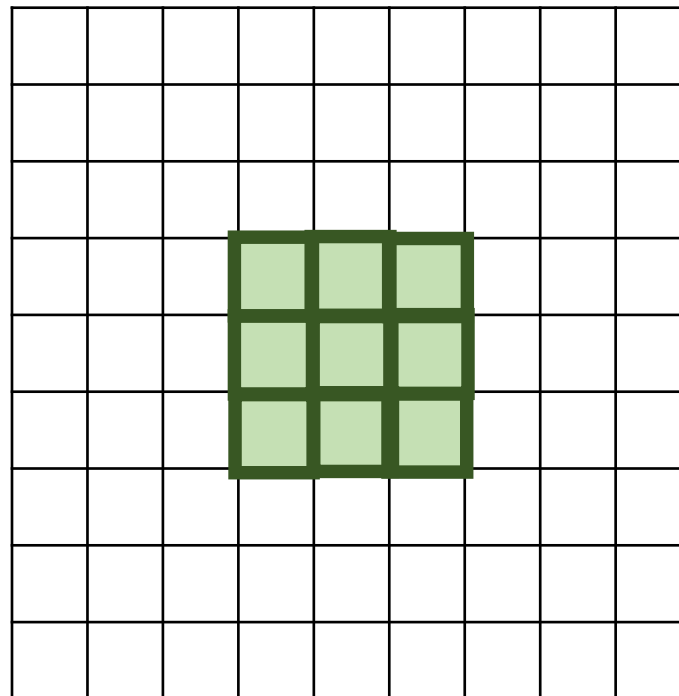
Mds



M



N

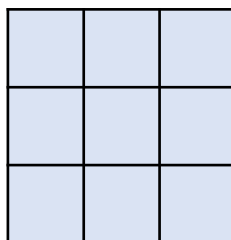


P

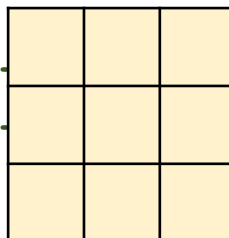
Phase #2

```
row = by * blockDim.y  
col = bx * blockDim.x
```

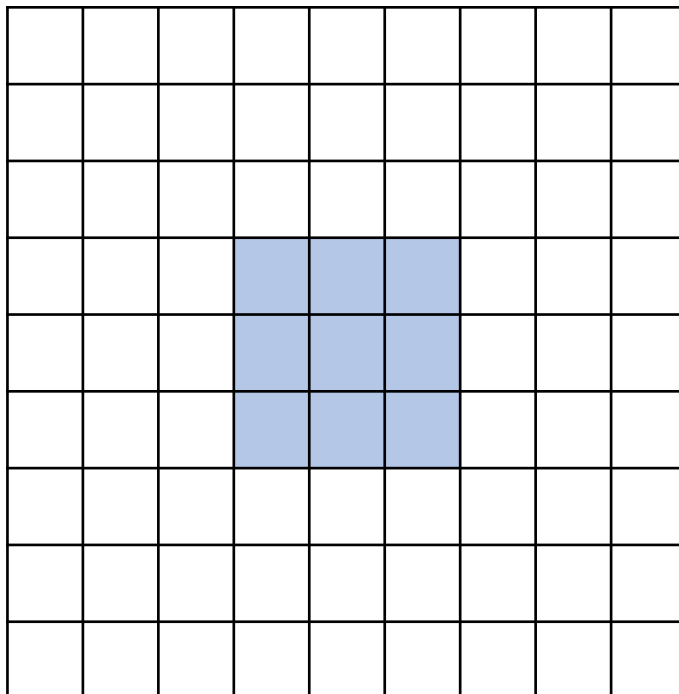
Mds



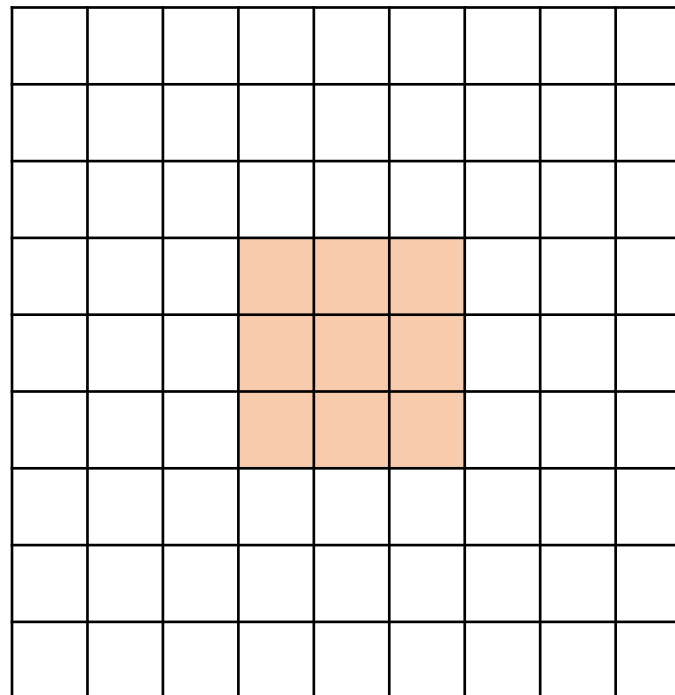
Nds



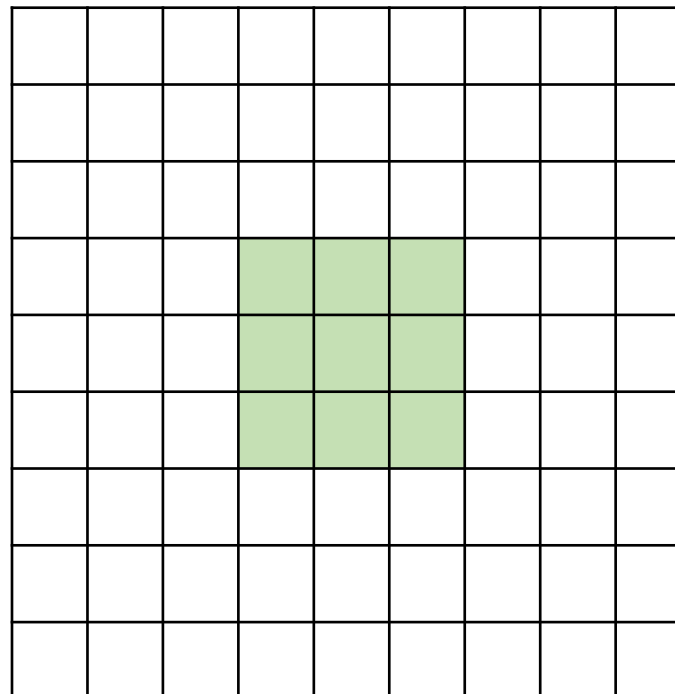
M



N

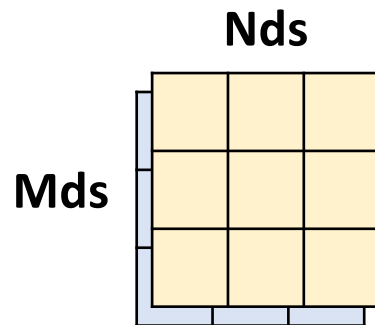


P

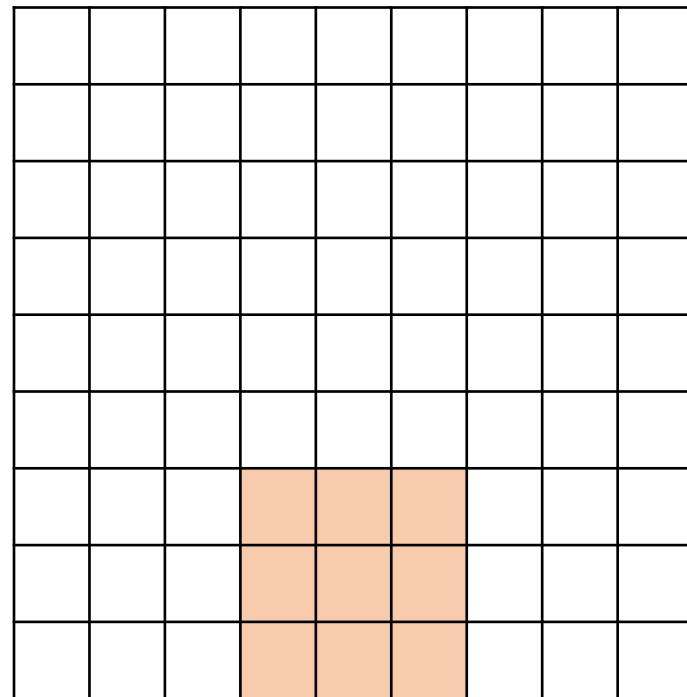
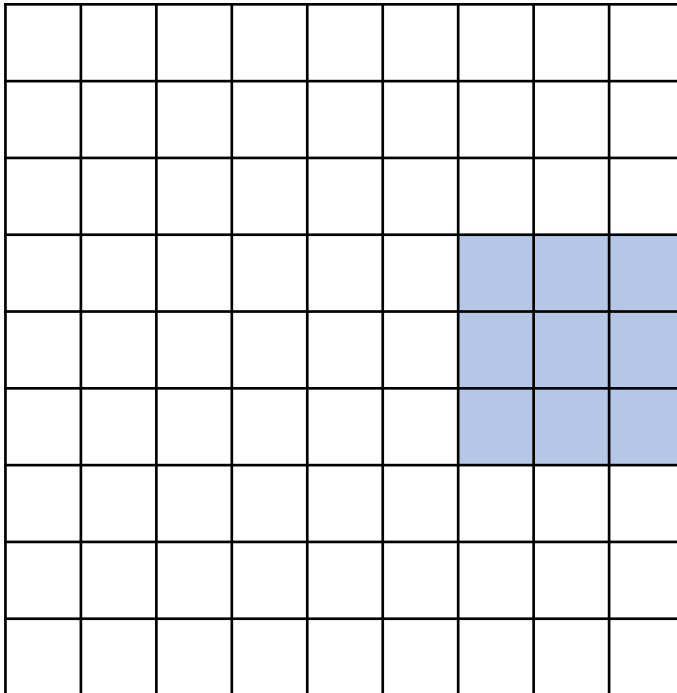


Phase #3

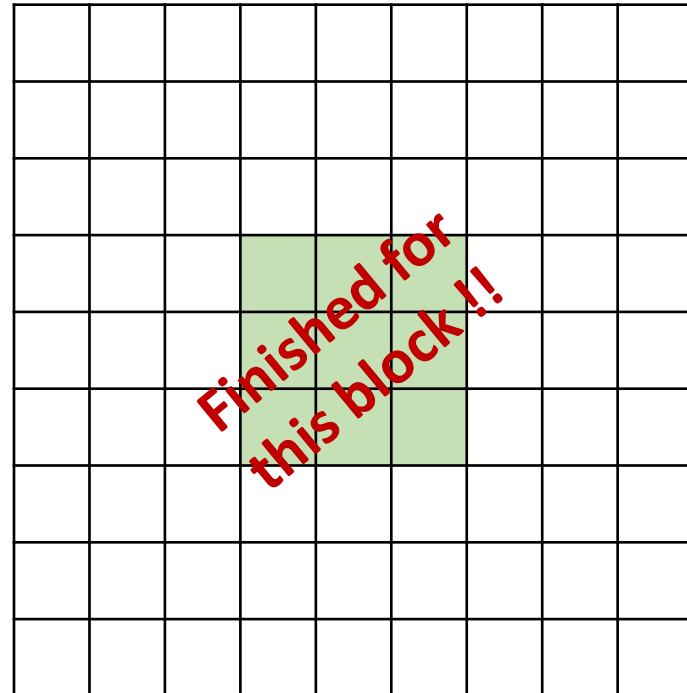
```
row = by * blockDim.y + ty;  
col = bx * blockDim.x + tx;
```



M



N



P