中国科学院大学
University of Chinese Academy of Sciences

Introduction to CUDA

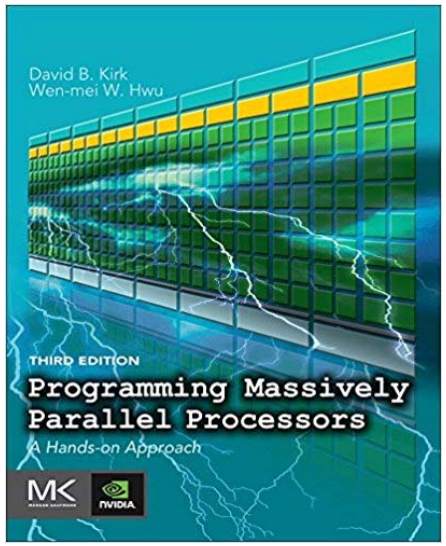
(4) Memory And Data Locality

Reference

- **CUDA C Programming Guide,**
  - **https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html**

- **Programming Massively Parallel Processors,**
  - **A Hands-on Approach**
  - **Third Edition**

  - **Chapter 4**



Content

- Importance of Memory Access Efficiency
  - --Matrix Multiplication
- Memory Types Overview
- Tiling for reduced memory traffic
  - --Matrix Multiplication using Shared Memory

Importance of Memory Access Efficiency

```
        for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow) {
4.        for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol) {

5.          int curRow = Row + blurRow;
6.          int curCol = Col + blurCol;
            // Verify we have a valid image pixel
7.          if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
8.            pixVal += in[curRow * w + curCol];
9.            pixels++; // Keep track of number of pixels in the avg
            }
          }
        }
```

The most executed part of the image blurring kernel.

How to evaluate memory access efficiency:

Compute-to-global-memory-access Ratio

Importance of Memory Access Efficiency

```
        for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow) {
4.        for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol) {

5.          int curRow = Row + blurRow;
6.          int curCol = Col + blurCol;
            // Verify we have a valid image pixel
7.          if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
8.            pixVal += in[curRow * w + curCol];
9.            pixels++; // Keep track of number of pixels in the avg
            }
          }
        }
```

The most executed part of the image blurring kernel.

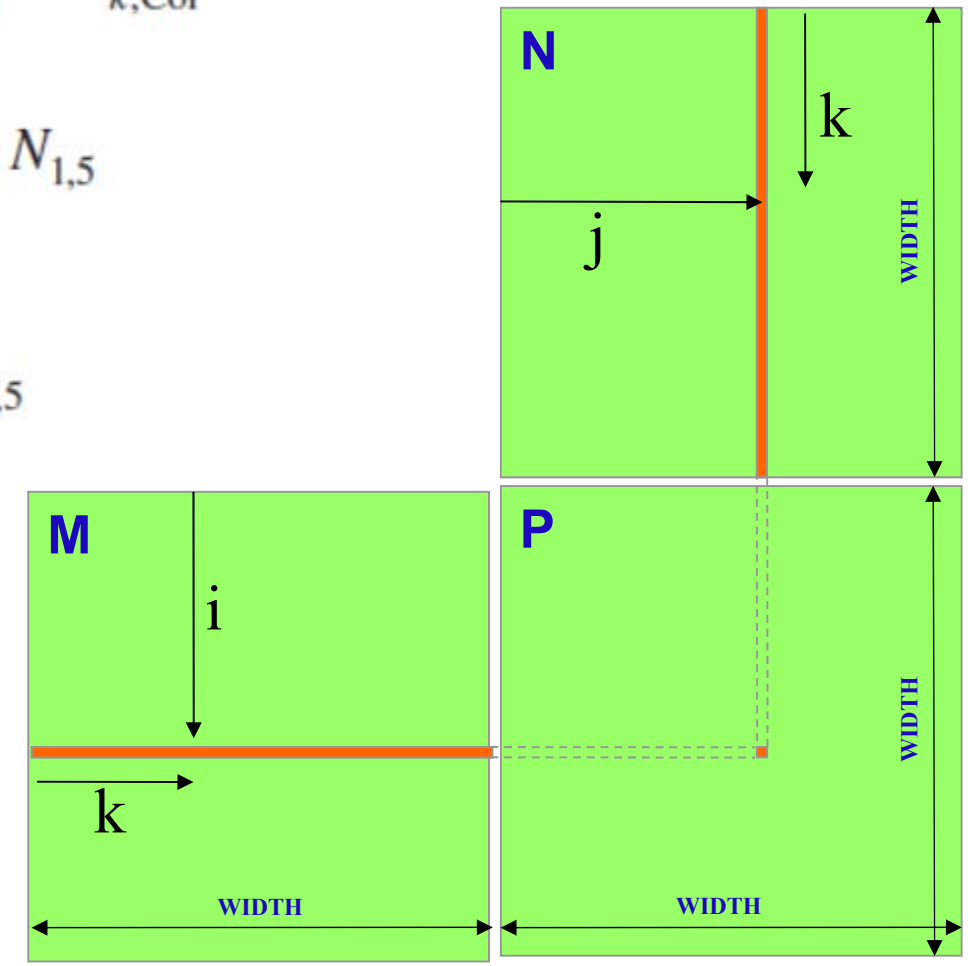
Compute-to-global-memory-access Ratio:

- Global memory bandwidth is 1TB/s, with 4 bytes in each single precision float value,
- No more than 1000/4 = 250 G single-precision ops can be loaded in one second.
- If Compute-to-global-memory-access ratio is 1, no more than 250GFLOPS can be achieved, only 2% of 12TFLOPS of recent device.
- The ratio need to be improved to 48 or higher.

Matrix Multiplication Example

$$P_{\text{Row,Col}} = \sum M_{\text{Row},k} * N_{k,\text{Col}}$$

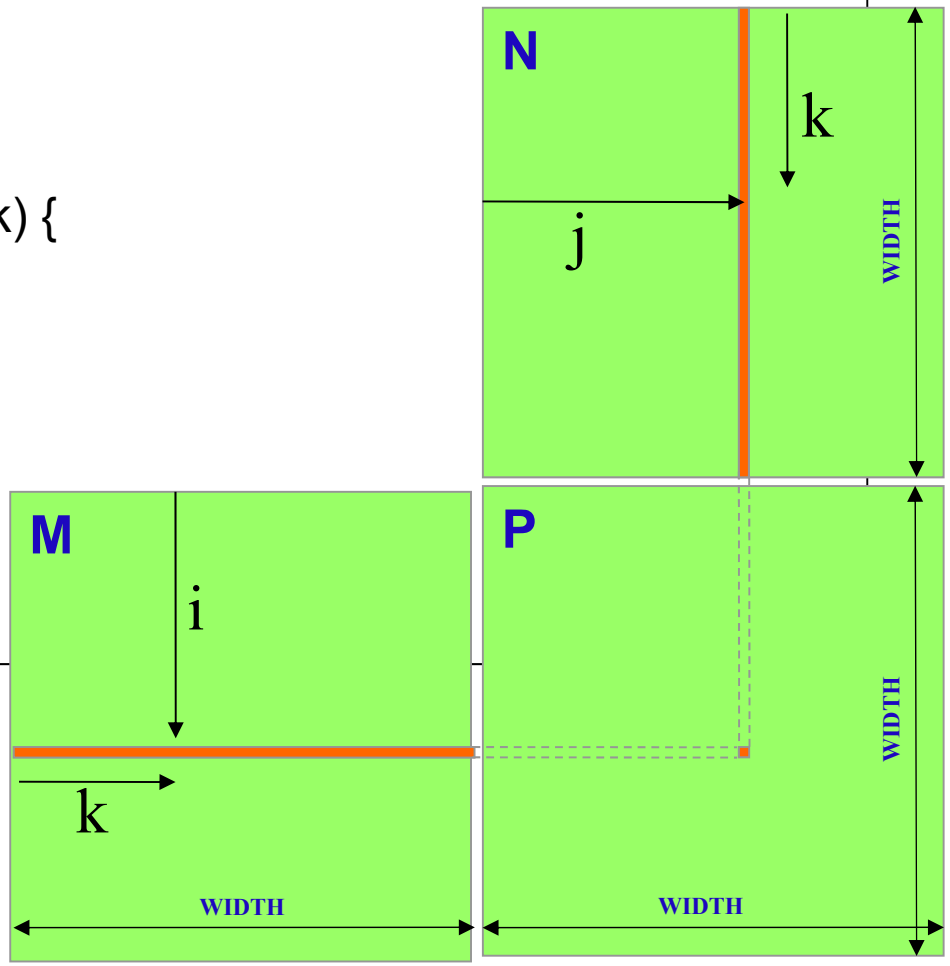
$$P_{1,5} = M_{1,0} * N_{0,5} + M_{1,1} * N_{1,5} + M_{1,2} * N_{2,5} + \cdots + M_{1,\text{Width}-1} * N_{\text{Width}-1,5}$$



Matrix Multiplication Example

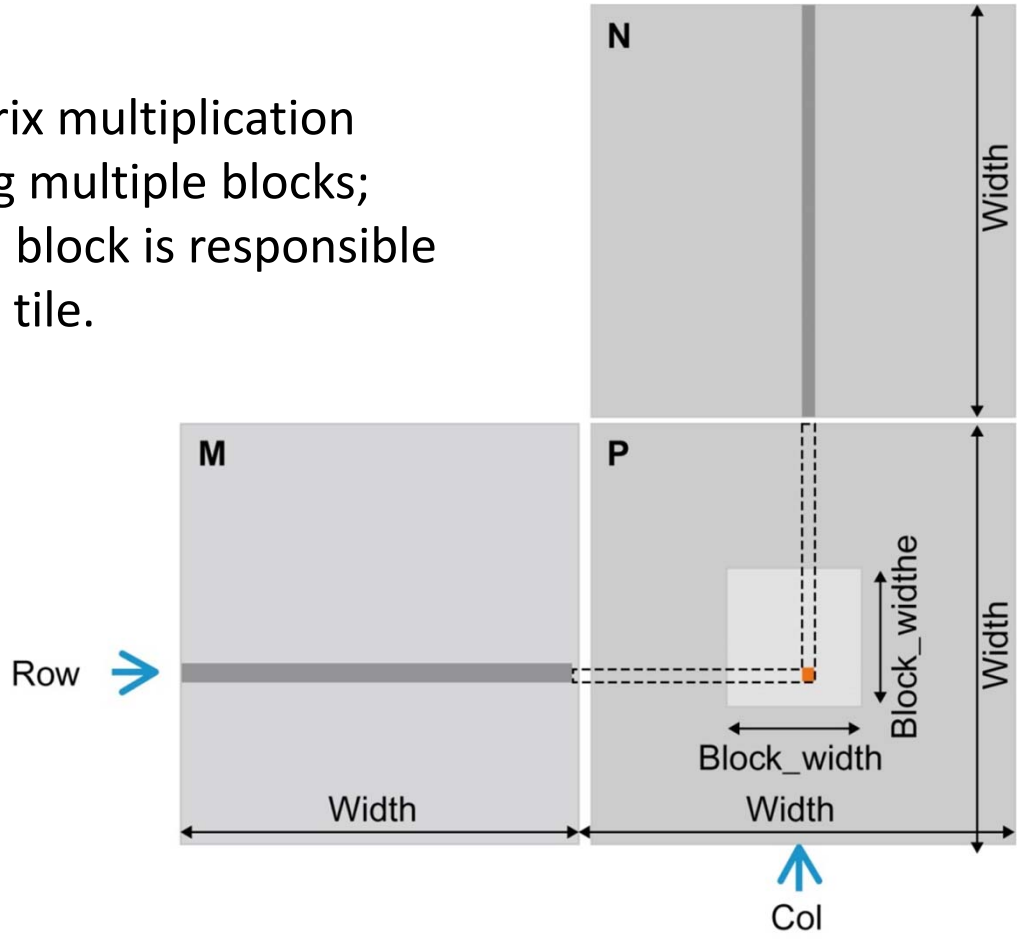
```
// Matrix multiplication on the (CPU) host in single precision
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
    for (int i = 0; i < Width; ++i)
        for (int j = 0; j < Width; ++j) {
            float sum = 0;
            for (int k = 0; k < Width; ++k) {
                float a = M[i * Width + k];
                float b = N[k * Width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
}
```

A Simple Host Version in C



Initial matrix multiplication version

- Matrix multiplication using multiple blocks;
- Each block is responsible for a tile.



Initial matrix multiplication version

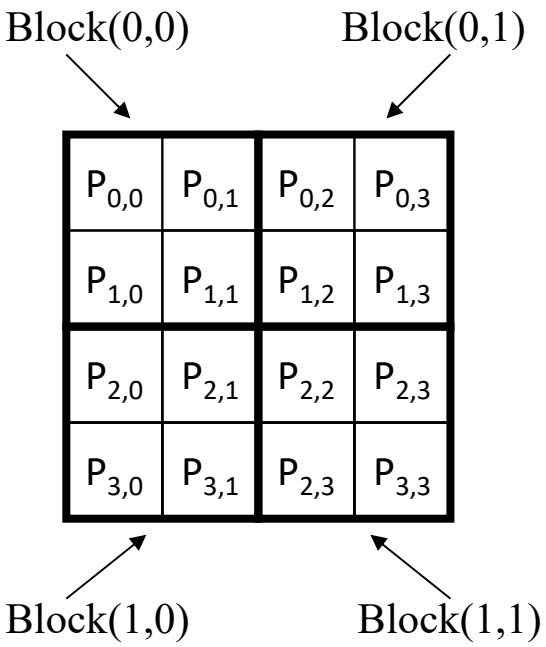
```
__global__ void MatrixMulKernel(float* M, float* N, float* P,
  int Width) {
  // Calculate the row index of the P element and M
  int Row = blockIdx.y*blockDim.y+threadIdx.y;
  // Calculate the column index of P and N
  int Col = blockIdx.x*blockDim.x+threadIdx.x;
  if ((Row < Width) && (Col < Width)) {
    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 simple matrix multiplication kernel:

-- using one thread to compute one P element.

Kernel Function - A Small Example

- Have each 2D thread block to compute a $(TILE\_WIDTH)^2$ sub-matrix (tile) of the result matrix
  - Each has $(TILE\_WIDTH)^2$ threads
- Generate a 2D Grid of $(WIDTH/TILE\_WIDTH)^2$ blocks



WIDTH = 4; TILE_WIDTH = 2
Each block has 2*2 = 4 threads

WIDTH/TILE_WIDTH = 2
Use 2* 2 = 4 blocks

A Slightly Bigger Example

(TILE_WIDTH =2)

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$	$P_{0,4}$	$P_{0,5}$	$P_{0,6}$	$P_{0,7}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$	$P_{1,4}$	$P_{1,5}$	$P_{1,6}$	$P_{1,7}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$	$P_{2,4}$	$P_{2,5}$	$P_{2,6}$	$P_{2,7}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$	$P_{3,4}$	$P_{3,5}$	$P_{3,6}$	$P_{3,7}$
$P_{4,0}$	$P_{4,1}$	$P_{4,2}$	$P_{4,3}$	$P_{4,4}$	$P_{4,5}$	$P_{4,6}$	$P_{4,7}$
$P_{5,0}$	$P_{5,1}$	$P_{5,2}$	$P_{5,3}$	$P_{5,4}$	$P_{5,5}$	$P_{5,6}$	$P_{5,7}$
$P_{6,0}$	$P_{6,1}$	$P_{6,2}$	$P_{6,3}$	$P_{6,4}$	$P_{6,5}$	$P_{6,6}$	$P_{6,7}$
$P_{7,0}$	$P_{7,1}$	$P_{7,2}$	$P_{7,3}$	$P_{7,4}$	$P_{7,5}$	$P_{7,6}$	$P_{7,7}$

WIDTH = 8; TILE_WIDTH = 2
Each block has 2*2 = 4 threads

WIDTH/TILE_WIDTH = 4
Use 4* 4 = 16 blocks

A Slightly Bigger Example (cont.)

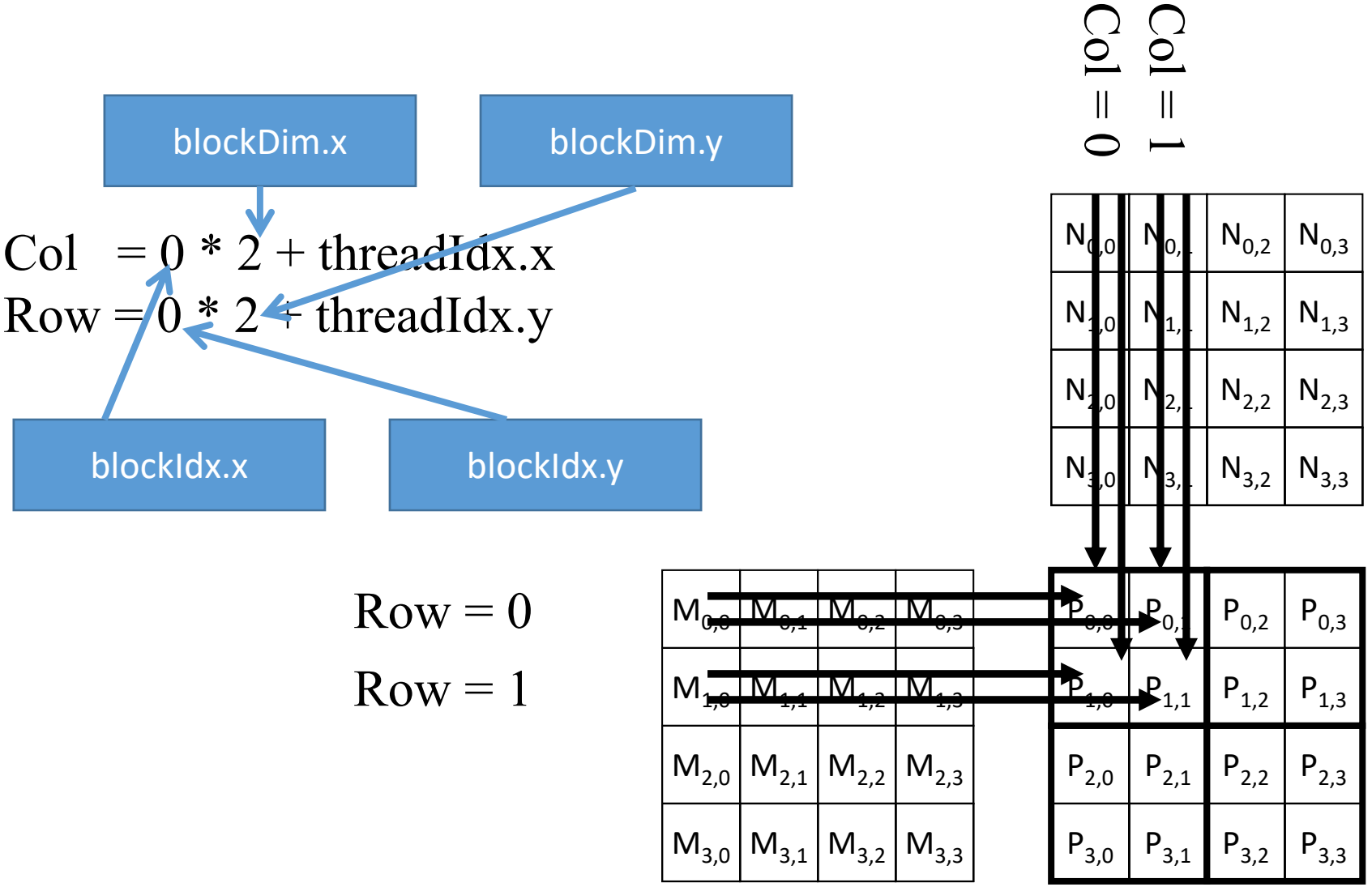
(TILE_WIDTH = 4)

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$	$P_{0,4}$	$P_{0,5}$	$P_{0,6}$	$P_{0,7}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$	$P_{1,4}$	$P_{1,5}$	$P_{1,6}$	$P_{1,7}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$	$P_{2,4}$	$P_{2,5}$	$P_{2,6}$	$P_{2,7}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$	$P_{3,4}$	$P_{3,5}$	$P_{3,6}$	$P_{3,7}$
$P_{4,0}$	$P_{4,1}$	$P_{4,2}$	$P_{4,3}$	$P_{4,4}$	$P_{4,5}$	$P_{4,6}$	$P_{4,7}$
$P_{5,0}$	$P_{5,1}$	$P_{5,2}$	$P_{5,3}$	$P_{5,4}$	$P_{5,5}$	$P_{5,6}$	$P_{5,7}$
$P_{6,0}$	$P_{6,1}$	$P_{6,2}$	$P_{6,3}$	$P_{6,4}$	$P_{6,5}$	$P_{6,6}$	$P_{6,7}$
$P_{7,0}$	$P_{7,1}$	$P_{7,2}$	$P_{7,3}$	$P_{7,4}$	$P_{7,5}$	$P_{7,6}$	$P_{7,7}$

WIDTH = 8; TILE_WIDTH = 4
Each block has 4*4 =16 threads

WIDTH/TILE_WIDTH = 2
Use 2* 2 = 4 blocks

Work for Block (0,0)



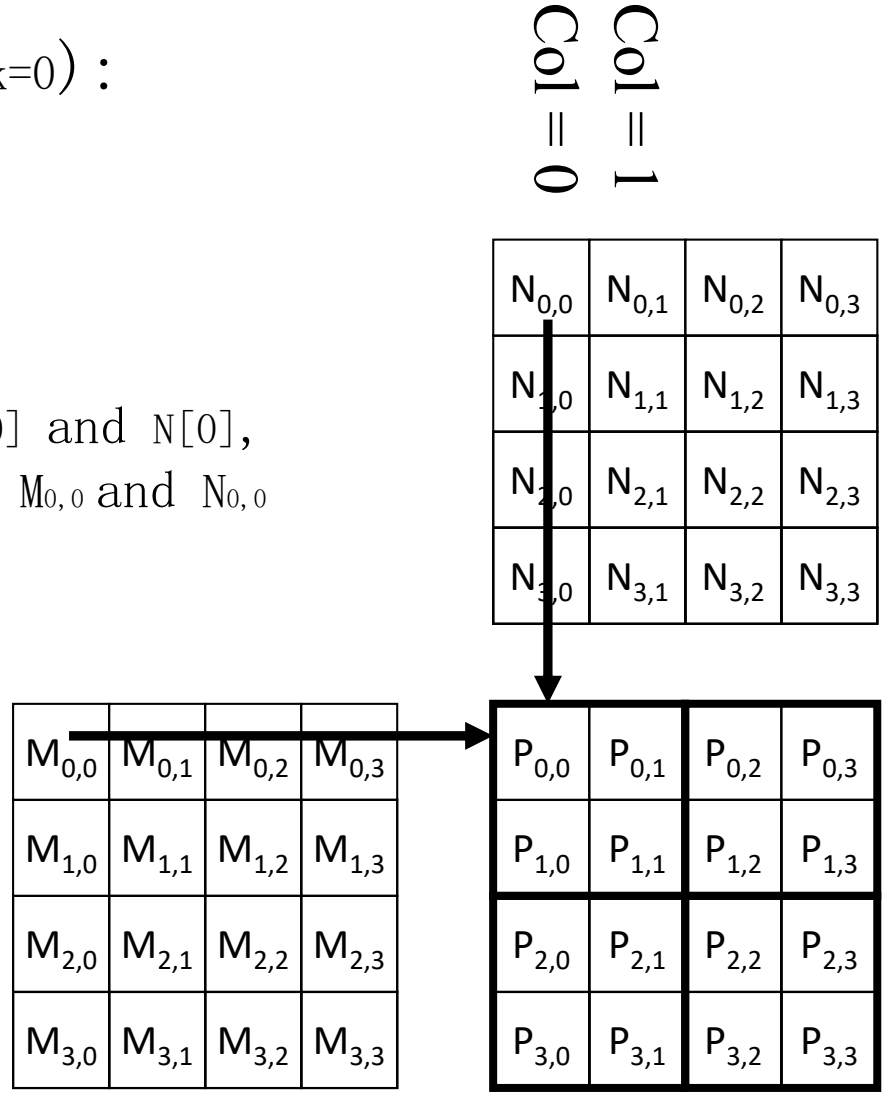
Work for Block (0,0) Thread(0,0)

During the 0^{th} iteration (k=0):

Row*Width+k = 0*4 + 0 = 0

k*Width+Col = 0*4 + 0 = 0

Therefore, we are accessing M[0] and N[0], which are the 1D equivalent of $M_{0,0}$ and $N_{0,0}$



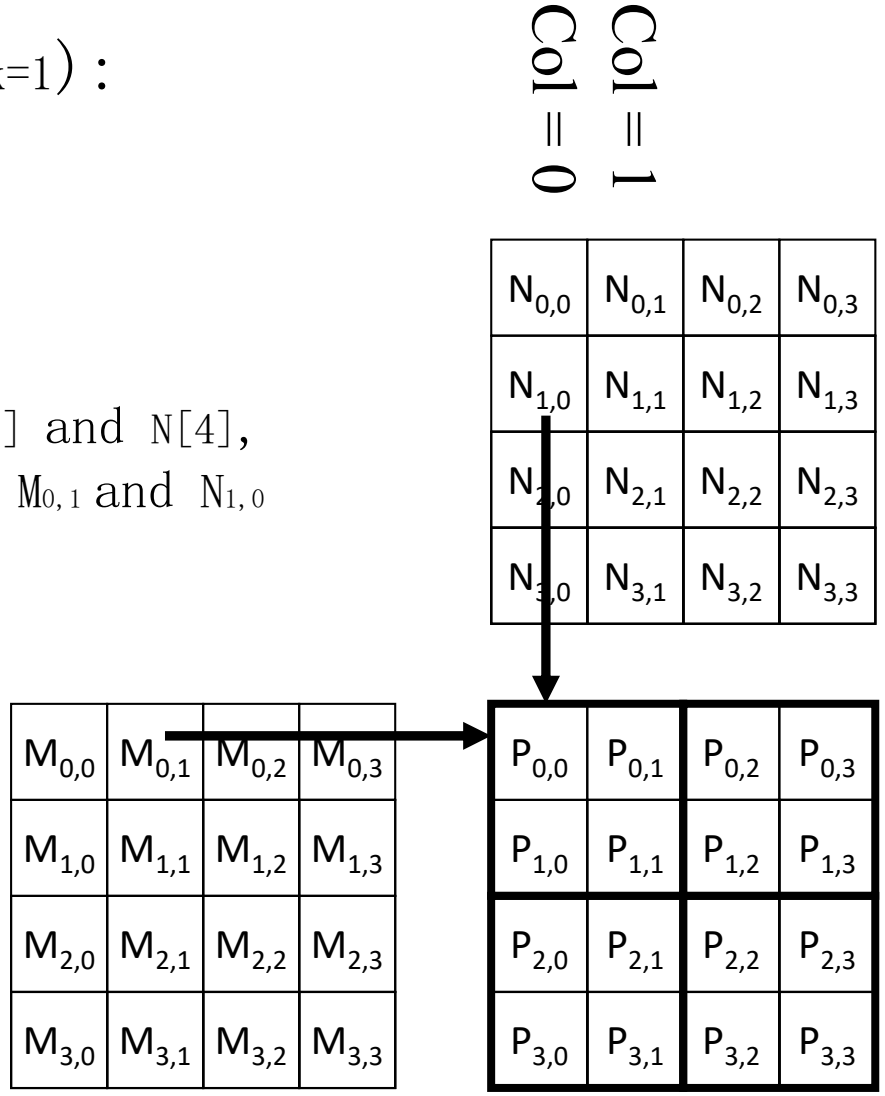
Work for Block (0,0) Thread(0,0)

During the 1st iteration (k=1):

Row*Width+k = 0*4 + 1 = 1

k*Width+Col = 1*4 + 0 = 4

Therefore, we are accessing M[1] and N[4], which are the 1D equivalent of $M_{0,1}$ and $N_{1,0}$



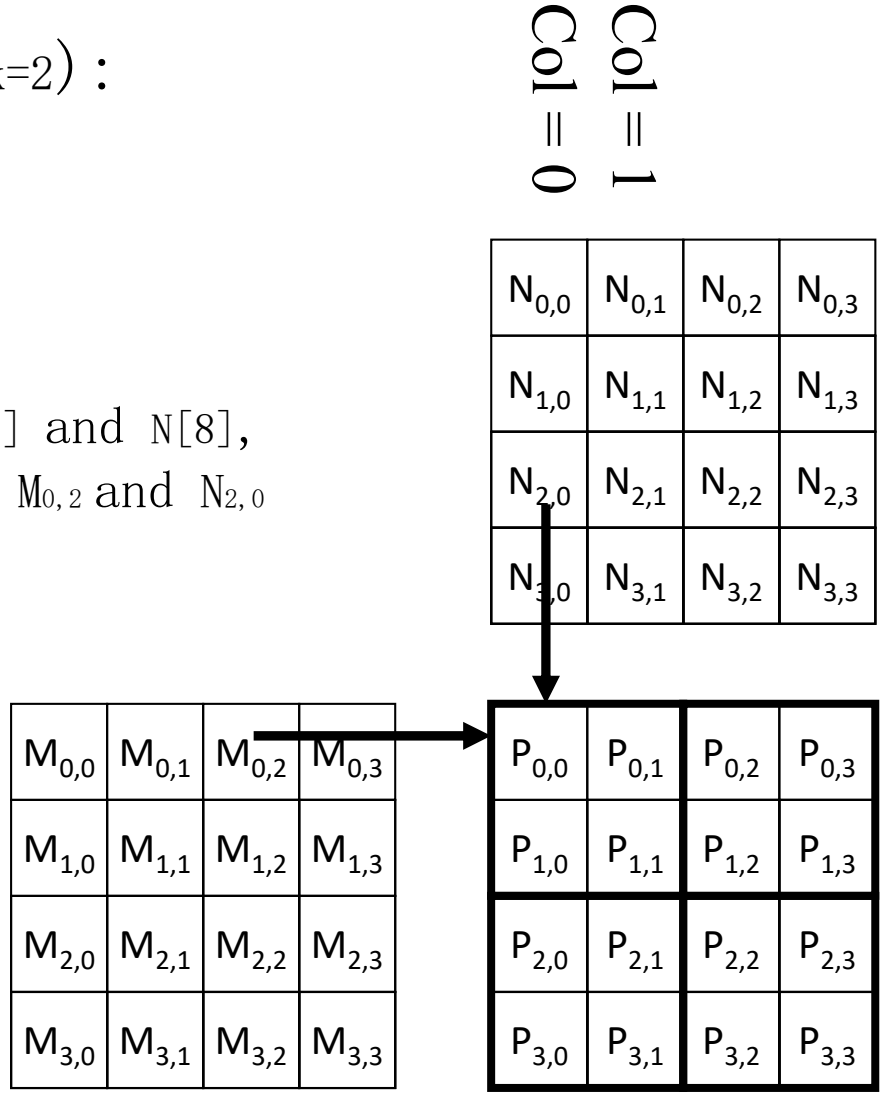
Work for Block (0,0) Thread(0,0)

During the 2nd iteration (k=2):

Row*Width+k = 0*4 + 2 = 2

k*Width+Col = 2*4 + 0 = 8

Therefore, we are accessing M[2] and N[8], which are the 1D equivalent of $M_{0,2}$ and $N_{2,0}$



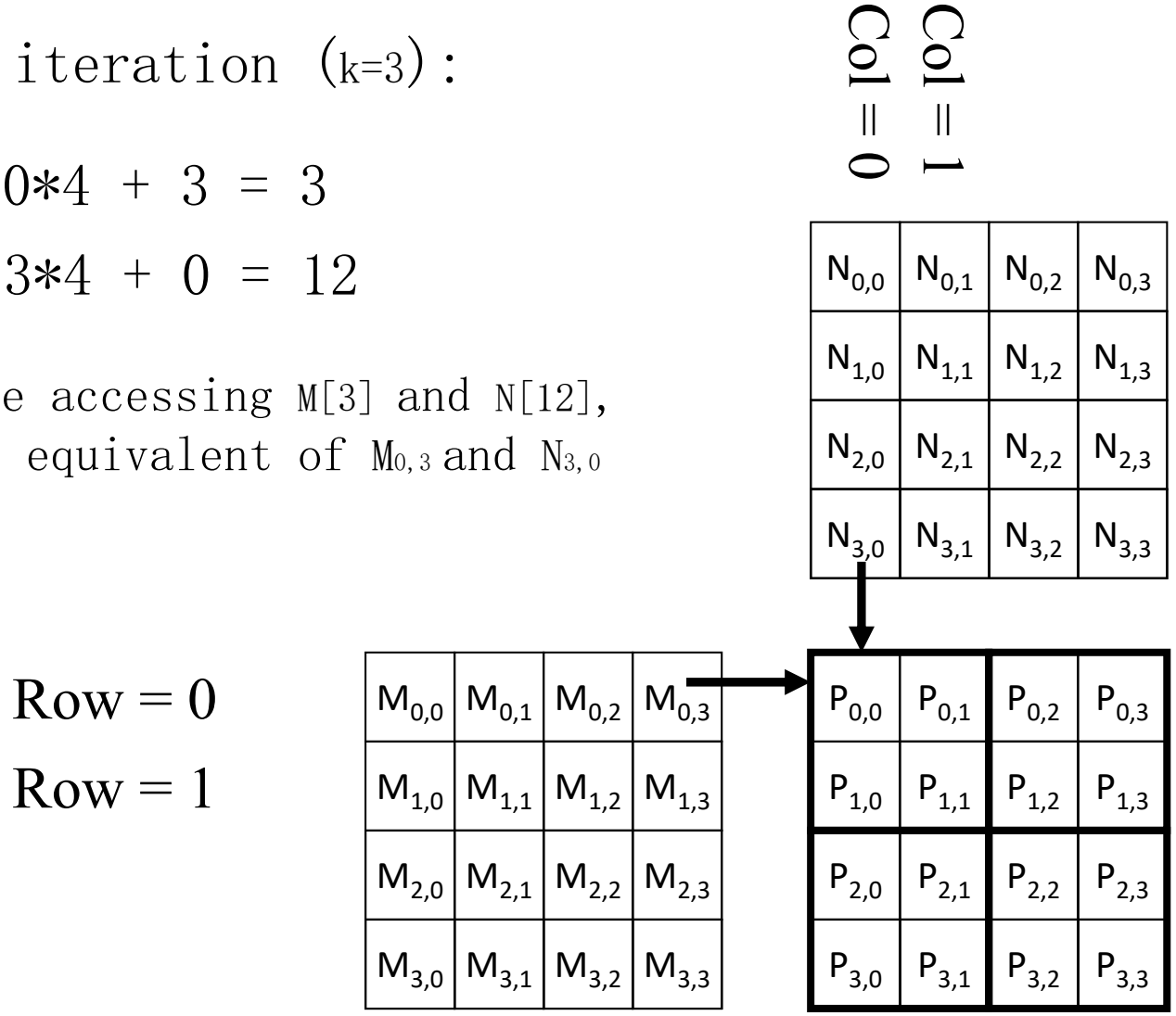
Work for Block (0,0) Thread(0,0)

During the 3rd iteration (k=3):

Row*Width+k = 0*4 + 3 = 3

k*Width+Col = 3*4 + 0 = 12

Therefore, we are accessing M[3] and N[12], which are the 1D equivalent of $M_{0,3}$ and $N_{3,0}$

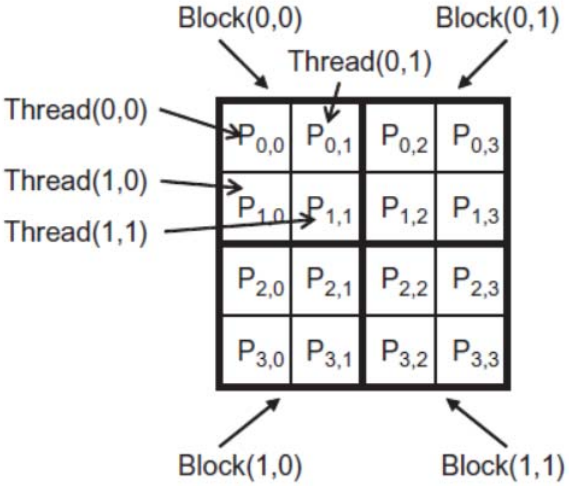


Work for Block (0,0) Thread(0,0)

At last:

Thread(0,0) in block(0,0) successfully calculated the inner product between the 0th row of M and the 0th column of N and deposited the result in $P_{0,0}$.

After the loop, the thread writes P[Row*Width+Col], which is P[0], the 1D equivalent of $P_{0,0}$.



Col = 1

Col = 0

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

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

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

Work for Block (0,1)

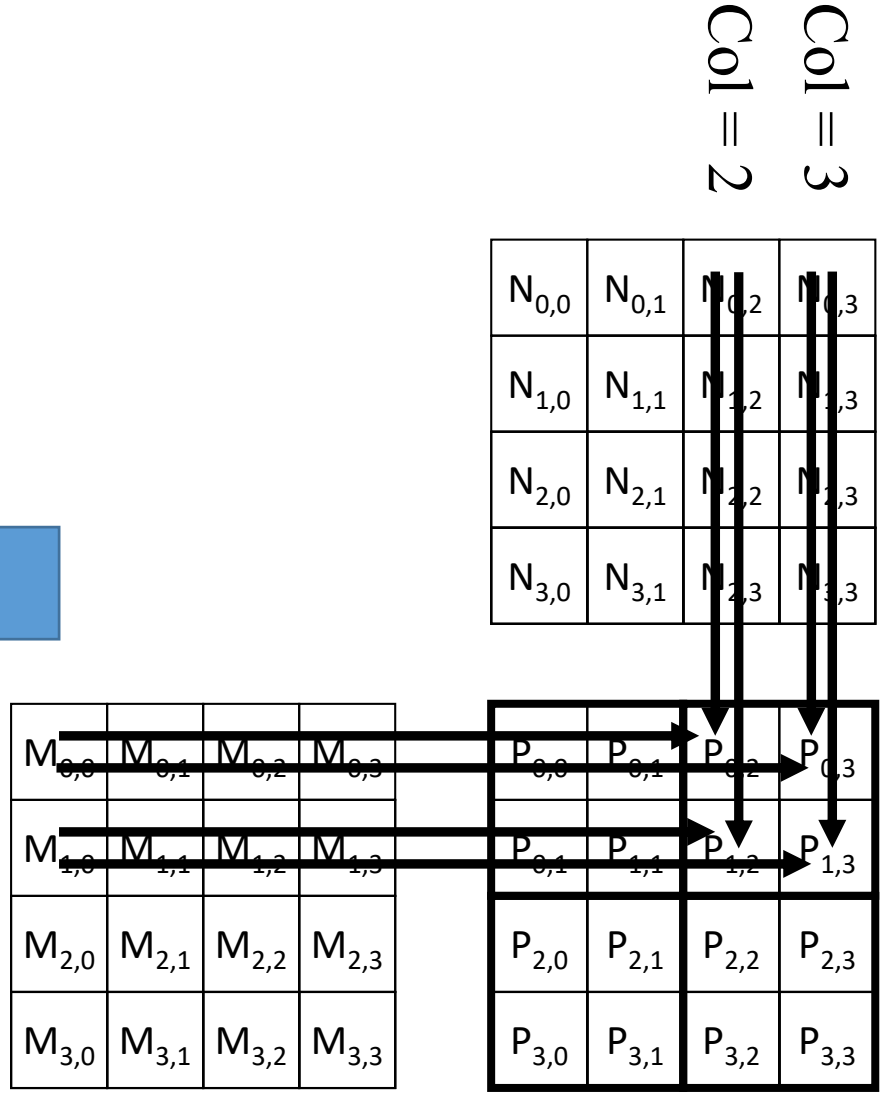
Col = 1 * 2 + threadIdx.x
Row = 0 * 2 + threadIdx.y

blockIdx.x

blockIdx.y

Row = 0

Row = 1



Walk through the for loop

```
for (int k = 0; k < Width; ++k)
    Pvalue += d_M[Row*Width+k] * d_N[k*Width+Col];
```

2 global memory access for:
One floating point multiplication
One floating point addition

Compute-to-global-memory-access ratio is 1

Result in less than 2% utilization

of the peak execution speed

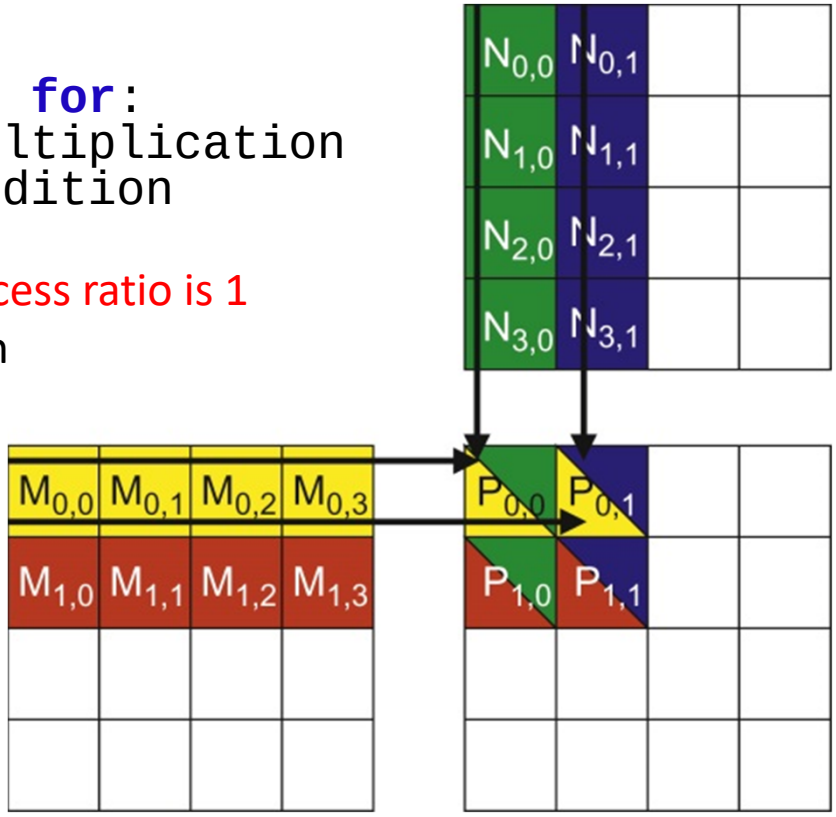
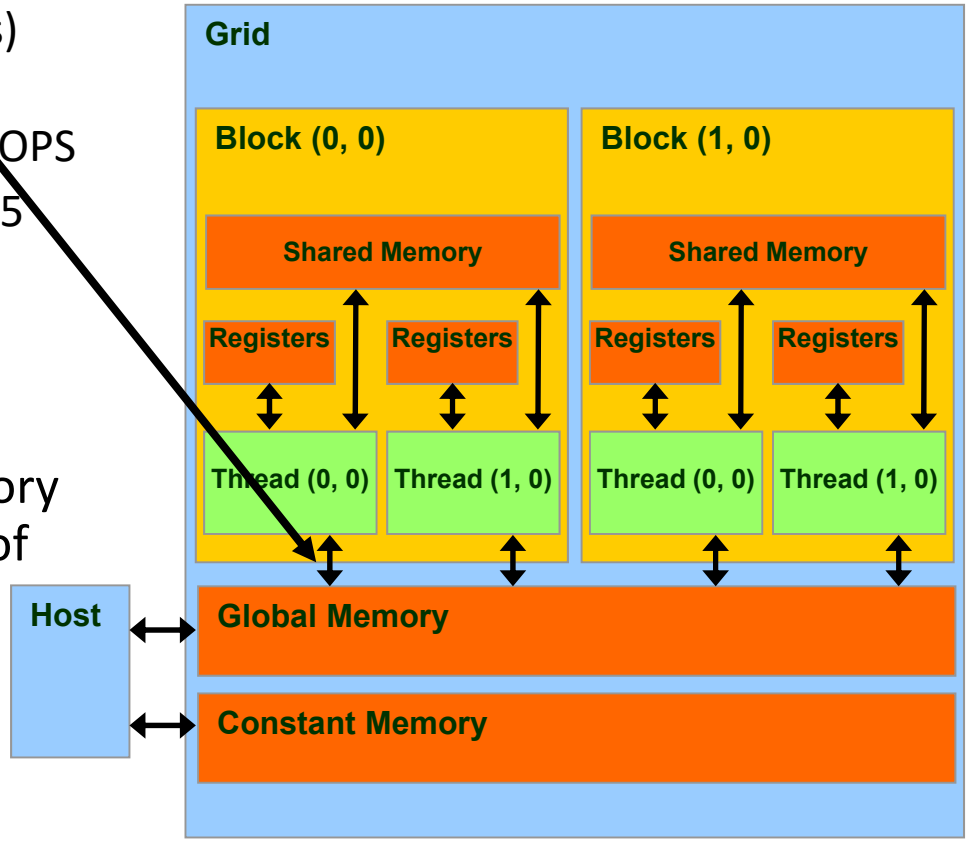


FIGURE 4.5: Matrix multiplication actions of one thread block.

How about performance on a device with 150 GB/s memory bandwidth?

- All threads access global memory for their input matrix elements
  - Two memory accesses (8 bytes) per floating point multiply-add
  - 4B/s of memory bandwidth/FLOPS
  - 150 GB/s limits the code at 37.5 GFLOPS
- The actual code runs at about 25 GFLOPS
- Need to drastically cut down memory accesses to get closer to the peak of more than 1,000 GFLOPS

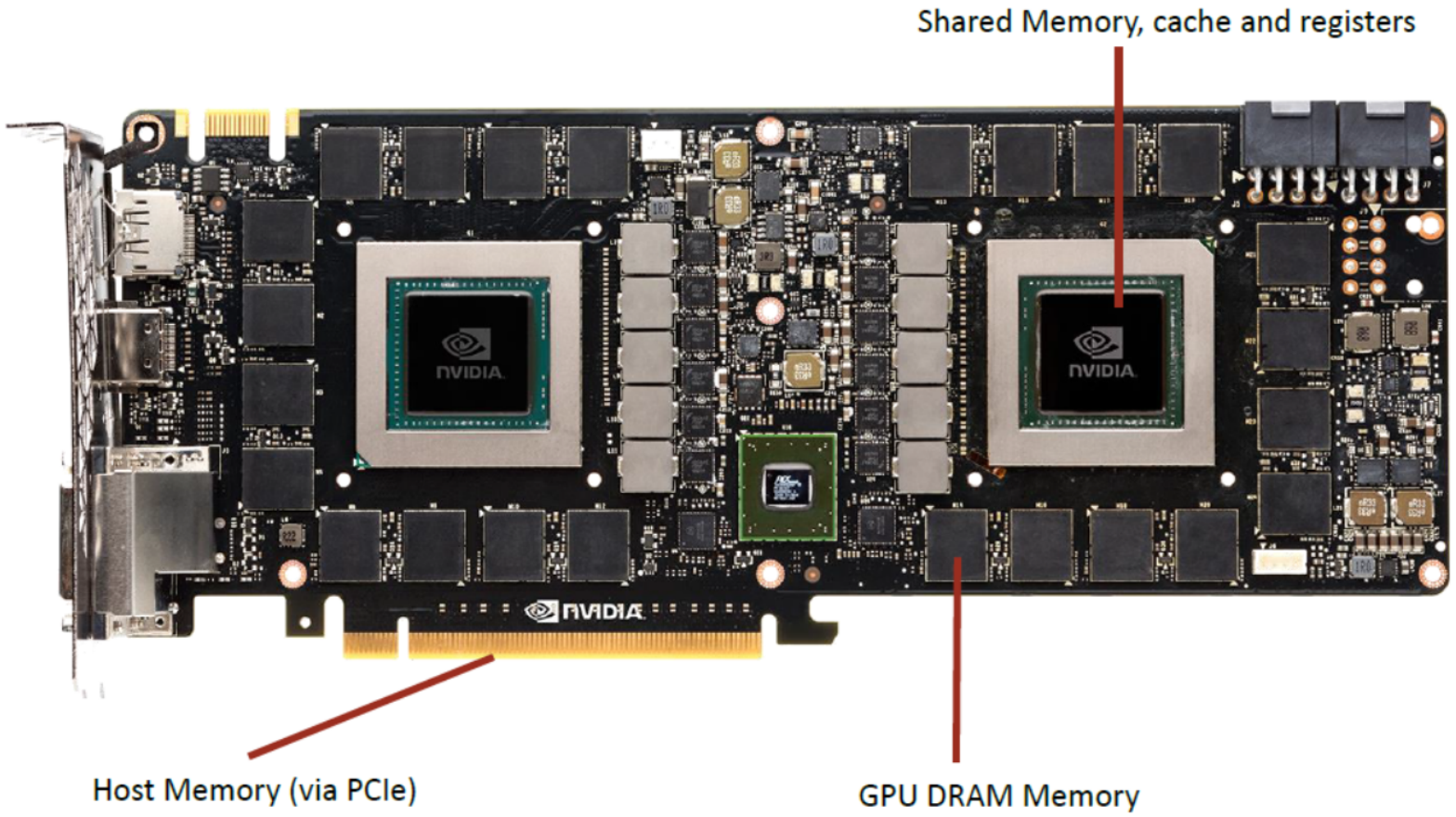


Content

- Importance of Memory Access Efficiency
  - --Matrix Multiplication
- Memory Types Overview
- Tiling for reduced memory traffic
  - --Matrix Multiplication using Shared Memory

CUDA Memories

GTX Titan Z



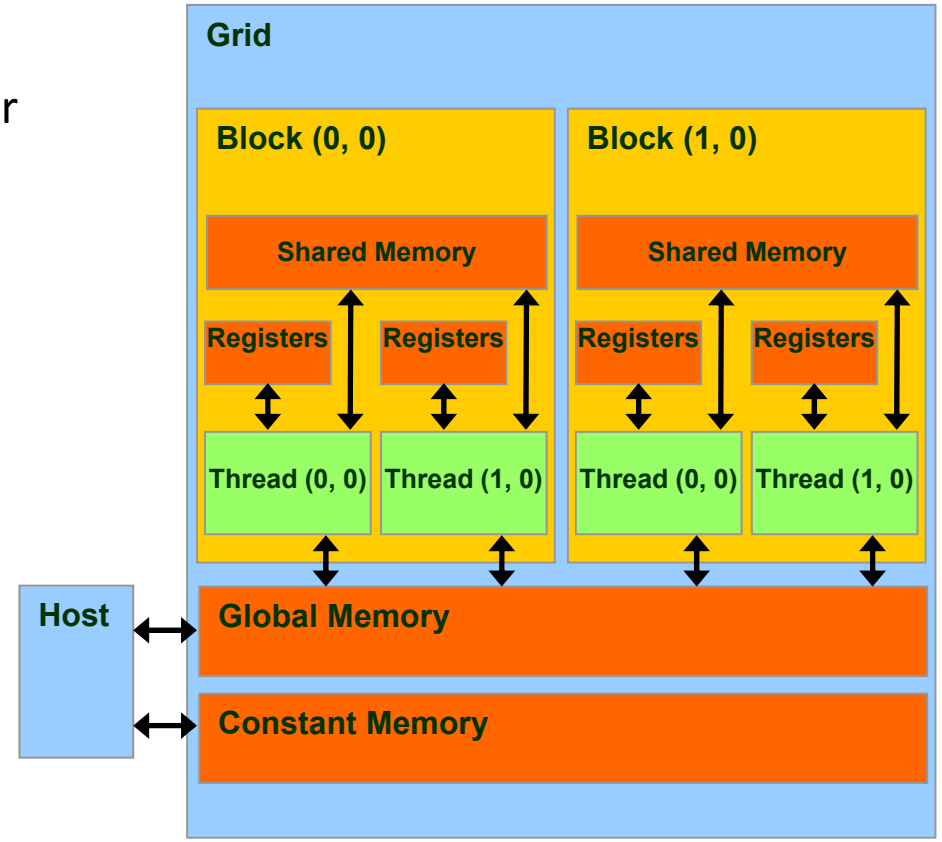
Programmer View of CUDA Memories

Host code can:

- Transfer data to/from per grid **global and constant** memories

Device code can:

- Read/write per-thread
  - **registers**
  - **(~1 cycle)**
- Read/write per-block
  - **shared memory**
  - **(~5 cycles)**
- Read/write per-grid
  - **global memory**
  - **(~500 cycles)**
- Read/only per-grid
  - **constant memory**
  - **(~5 cycles with caching)**



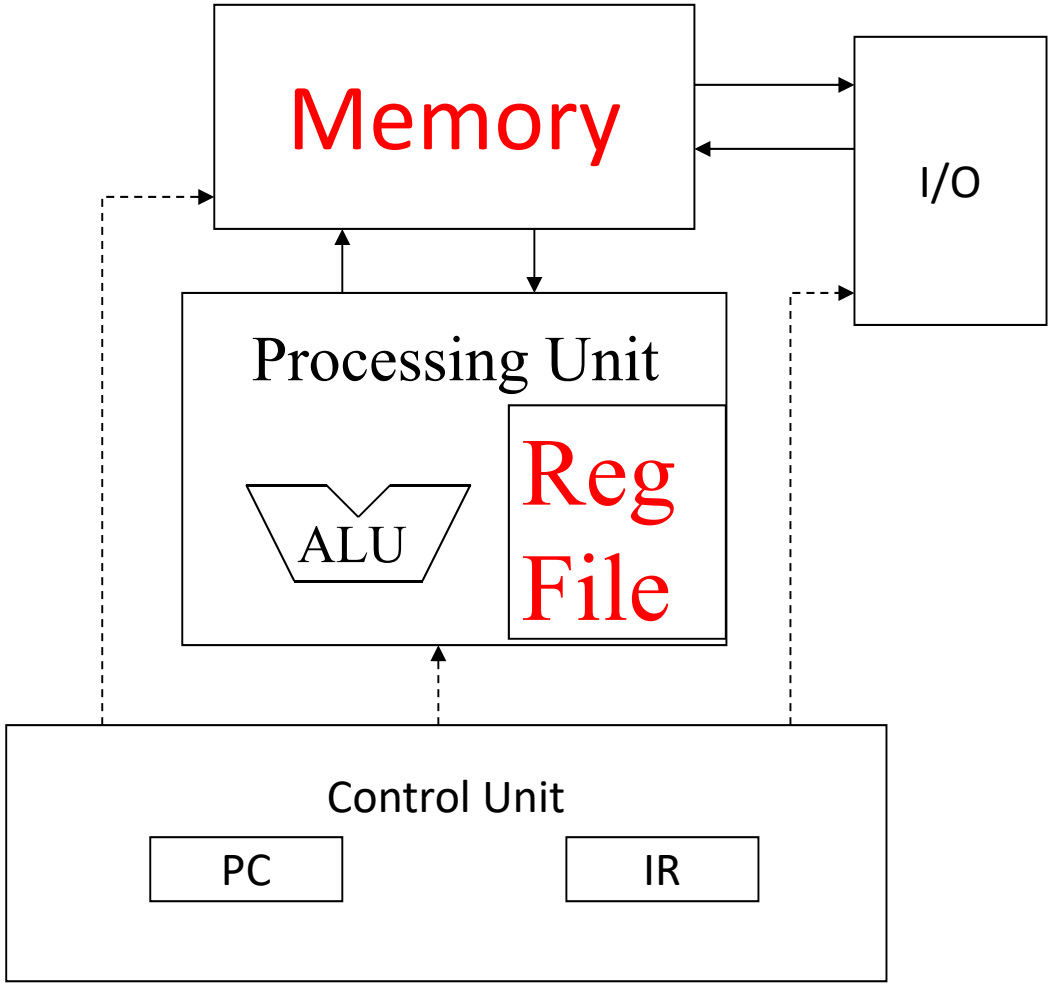
The Von-Neumann Model

Global Memory:

- Off-chip
- implies long access latencies and relatively low access bandwidth.

Register File：

- on processor chip
- very short access latency and drastically higher access bandwidth



Going back to the program

- Every instruction needs to be fetched from memory, decoded, then executed.
  - The decode stage typically accesses register file
- Instructions come in three flavors:
  - Operate, Data transfer, and Program Control Flow.

- An example instruction cycle is the following:

  Fetch | Decode | Execute | Memory

Operate Instructions

- Arithmetic instructions in most modern processors have “built-in” register operands.
- Example of an operate instruction:

```
fadd r1, r2, r3
```

Register numbers

- when an operand of an arithmetic instruction is in a register, no additional instruction is required to make the operand value available to ALU.

Memory Access Instructions

- if an operand value is in the global memory:

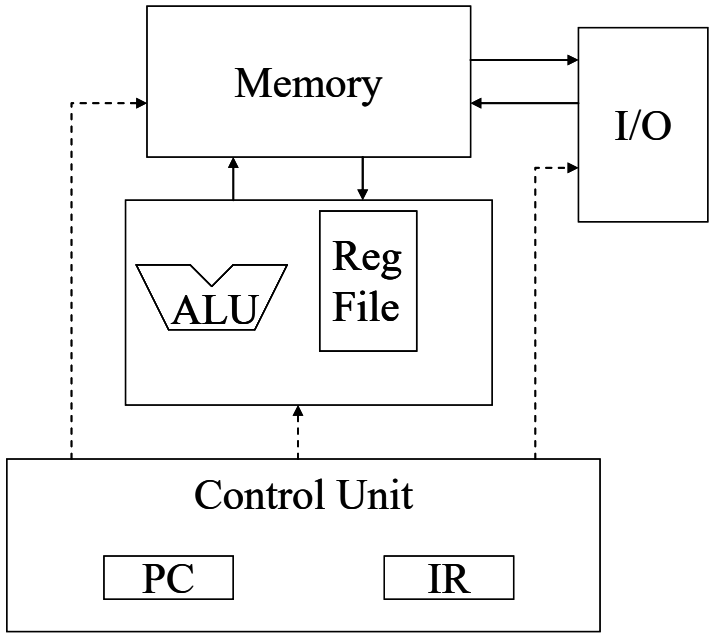
  load r2, r4, offset

  fadd r1, r2, r3

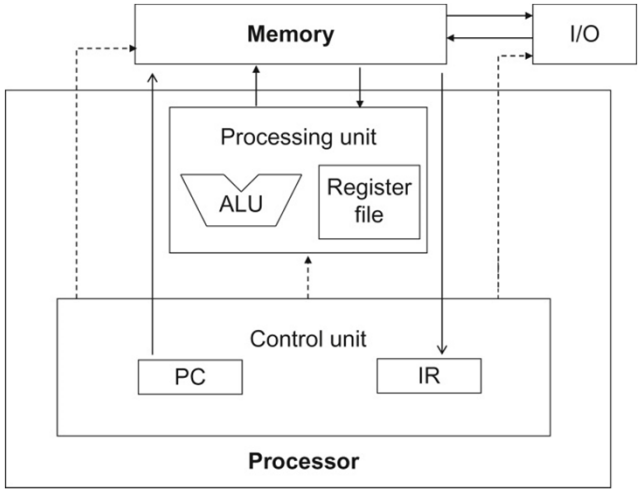
- It takes much more cost to load data from global memory than registers:
  - More clock cycles;
  - More energy;

Registers vs Memory

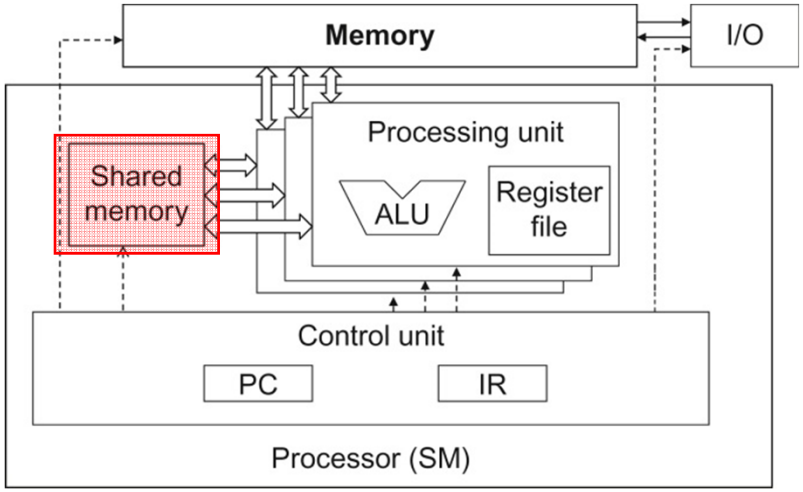
- Registers are “free”
  - No additional memory access instruction
  - Low energy comsumption
  - Very fast to use, however, there are very few of them
- Memory is expensive (slow), but very large



Registers vs Memory



von Neumann model



CUDA device SM

Differences among registers, shared memory, global memory:

- Location
- Latency
- Bandwidth
- Access scope

CUDA Variable Type Qualifiers

Variable declaration		Memory	Scope	Lifetime
Automatic variables	**int LocalVar;**	Register*	thread	kernel
__device__ __shared__	**int SharedVar;**	shared	block	Kernel
__device__	**int GlobalVar;**	global	grid	application
__device__ __constant__	**int ConstantVar;**	constant	grid	application

- **__device__** is optional when used with **__shared__**, or **__constant__**

- Automatic variables
  - without any qualifier reside in a register
  - Private copy for each thread (do not exceed limit)
    - *Except per-thread arrays that reside in global memory

CUDA Variable Type Qualifiers

Variable declaration		Memory	Scope	Lifetime
Automatic variables	`int LocalVar;`	Register*	thread	kernel
`__device__ __shared__`	`int SharedVar;`	shared	block	Kernel
`__device__`	`int GlobalVar;`	global	grid	application
`__device__ __constant__`	`int ConstantVar;`	constant	grid	application

- Shared variables
  - All threads in a block see the same version
  - Private copy for each block
  - Efficient means for threads in block to collaborate
  - Hold portion of global memory data that heavily used in kernel execution phase.

CUDA Variable Type Qualifiers

Variable declaration		Memory	Scope	Lifetime
Automatic variables	`int LocalVar;`	Register*	thread	kernel
`__device__ __shared__`	`int SharedVar;`	shared	block	Kernel
`__device__`	`int GlobalVar;`	global	grid	application
`__device__ __constant__`	`int ConstantVar;`	constant	grid	application

- Global variables
  - Slow, but recent devices use cache to improve latency and throughput
  - Can be used for threads to collaborate across blocks,
  - Or pass information from one kernel invocation to another kernel invocation.

CUDA Variable Type Qualifiers

Variable declaration		Memory	Scope	Lifetime
Automatic variables	`int LocalVar;`	Register*	thread	kernel
`__device__ __shared__`	`int SharedVar;`	shared	block	Kernel
`__device__`	`int GlobalVar;`	global	grid	application
`__device__ __constant__`	`int ConstantVar;`	constant	grid	application

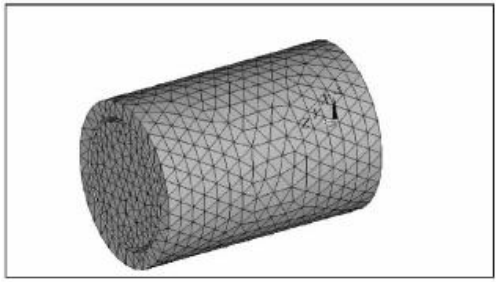
- Constant variables
  - In global memory but cached for efficient access.
  - Must outside any function body
  - All threads in all grids see the same version
  - Often used for variables that provide input values to kernel functions.

Content

- Importance of Memory Access Efficiency
  - --Matrix Multiplication
- Memory Types Overview
- Tiling for reduced memory traffic
  - --Matrix Multiplication using Shared Memory

A Common Programming Strategy

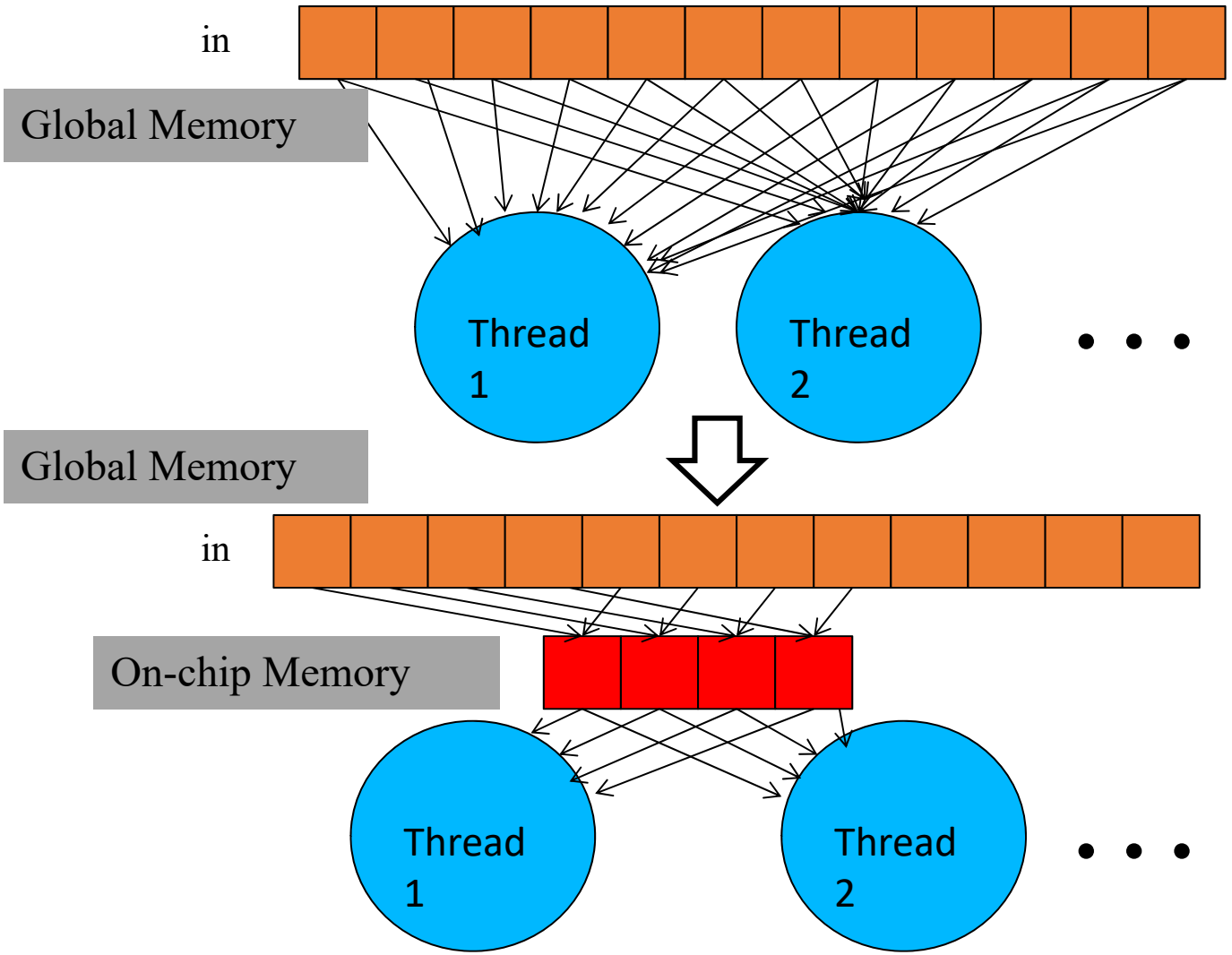
- Tradeoff of device memories in CUDA:
  - the global memory is large but slow,
  - whereas the shared memory is small but fast.
- A common strategy:
  - partition the data into subsets called *tiles*
  - so that each tile fits into the shared memory.



A Common Programming Strategy

- Tile the input data to take advantage of fast shared memory.
- Handle each data subset with one thread block by:
  - Loading the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism
  - Performing the computation on the subset from shared memory; each thread can efficiently multi-pass over any data element
  - Copying results from shared memory to global memory

Shared Memory Blocking Basic Idea



Outline of Technique

- Identify a tile of global data 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
- Move on to the next block/tile

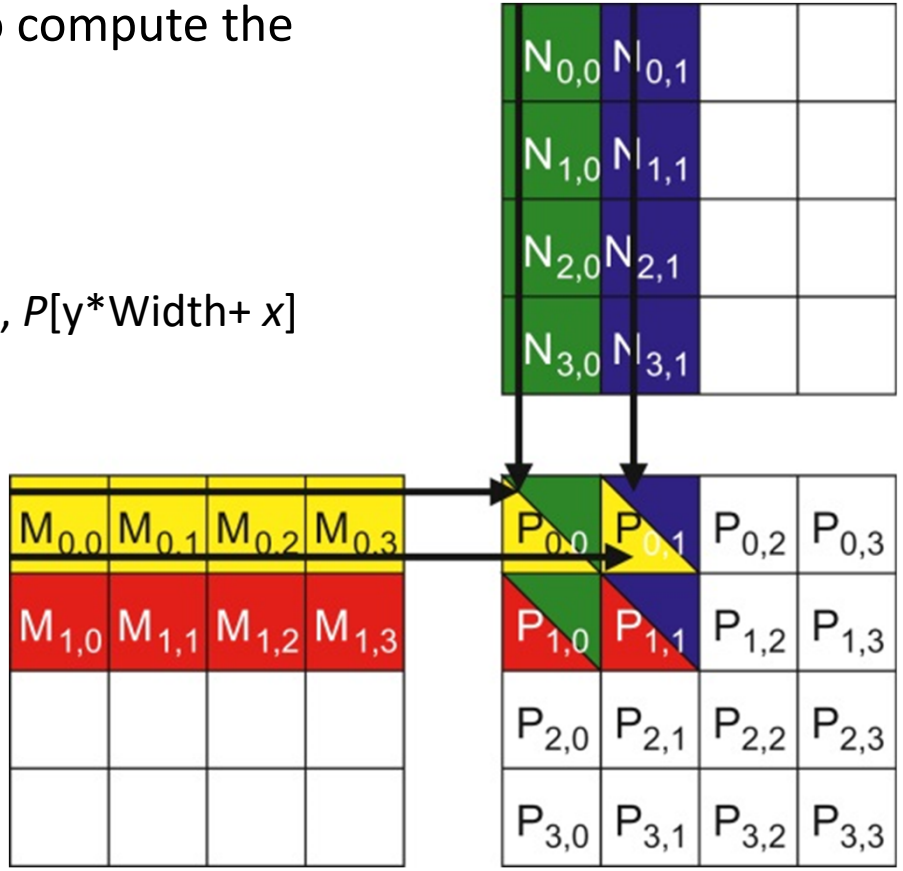
The small example

Assumes four 2× 2 blocks to compute the P matrix.

For brevity:

M[*y**Width+ *x*], *N*[*y**Width + *x*], *P*[y*Width+ *x*] as

My,x *Ny,x* *Py,x*.



4 threads of block(0,0) compute for $P_{0,0}$, $P_{0,1}$, $P_{1,0}$, and $P_{1,1}$.

Thread(0,0) reads $M_{0,0}$ and $N_{0,0}$, followed by $M_{0,1}$ and $N_{1,0}$, …

The small example

Memory access of Block(0,0)

Access order

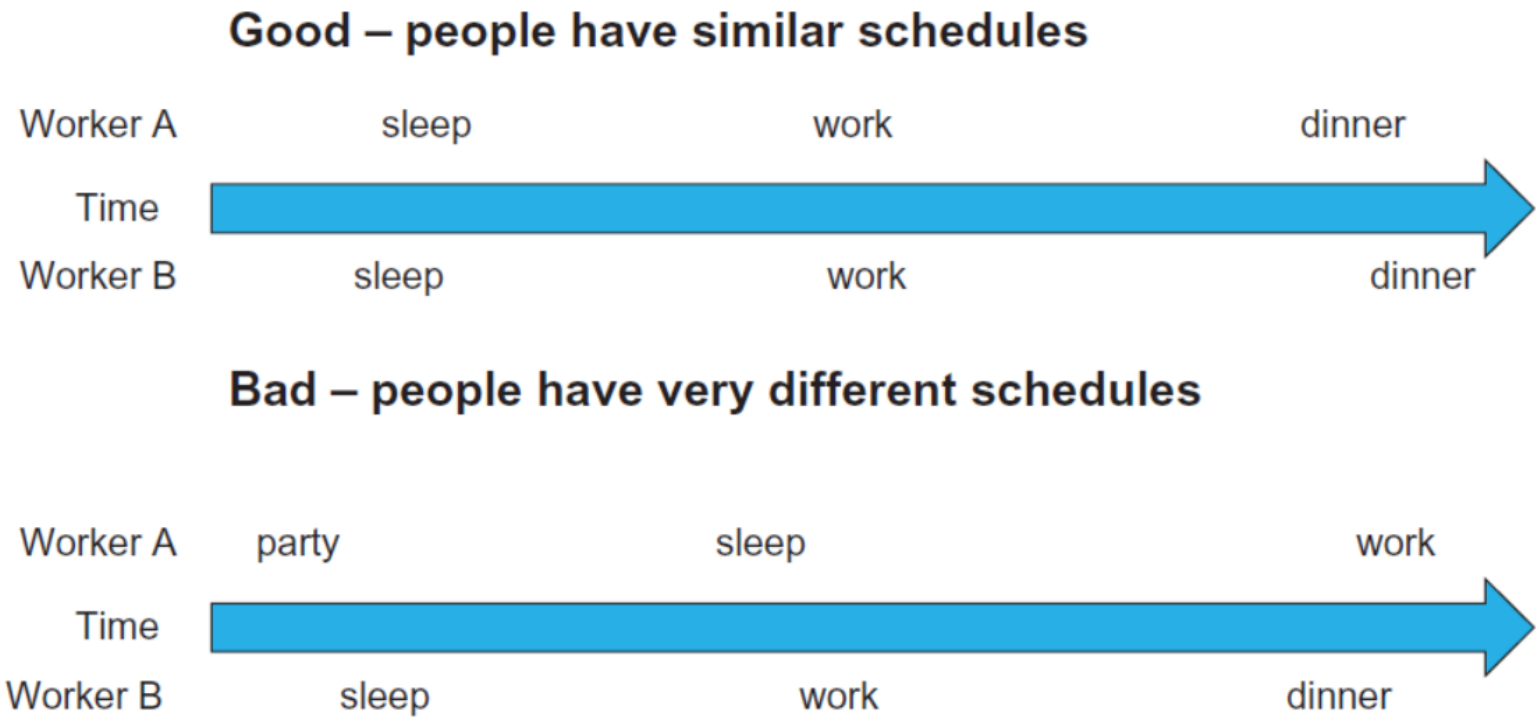
$thread_{0,0}$	$M_{0,0} * N_{0,0}$	$M_{0,1} * N_{1,0}$	$M_{0,2} * N_{2,0}$	$M_{0,3} * N_{3,0}$
$thread_{0,1}$	$M_{0,0} * N_{0,1}$	$M_{0,1} * N_{1,1}$	$M_{0,2} * N_{2,1}$	$M_{0,3} * N_{3,1}$
$thread_{1,0}$	$M_{1,0} * N_{0,0}$	$M_{1,1} * N_{1,0}$	$M_{1,2} * N_{2,0}$	$M_{1,3} * N_{3,0}$
$thread_{1,1}$	$M_{1,0}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

Every M and N element is accessed twice.
The potential reduction in global memory traffic is proportional to the dimension of blocks used.
If we use 16x16 blocks, traffic will be reduced to _____ ?

Compute-to-global-memory-access Ratio increased from 1 to 16,
peak performance approach 600GFLOPS.

Tiled algorithm are highly similar to carpooling arrangement.
With extra effort.

Carpooling requires synchronization



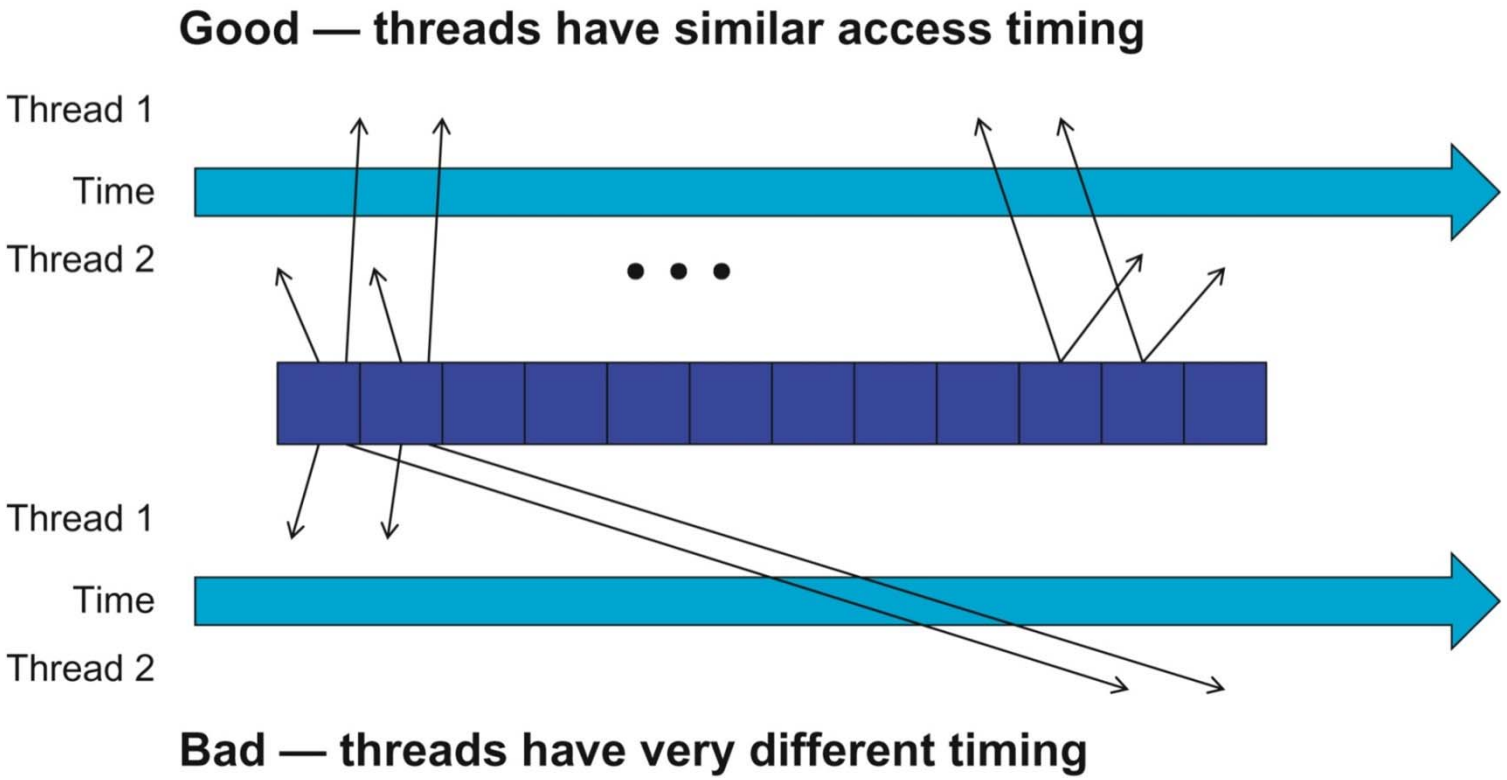


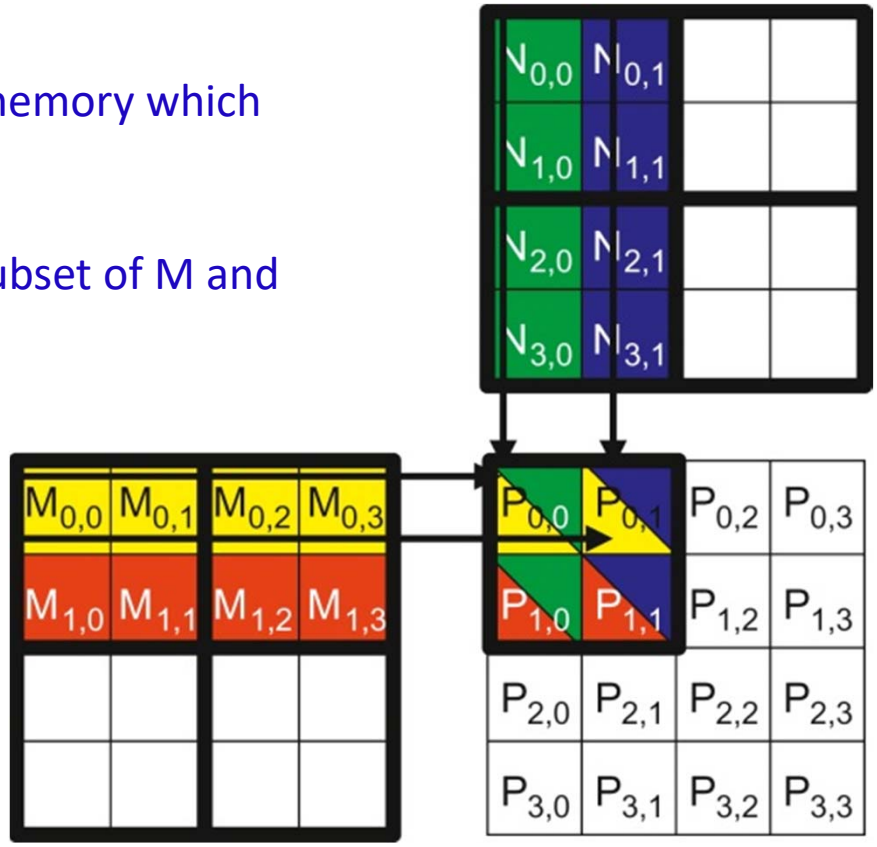
FIGURE 4.13: Tiled Algorithms require synchronization among threads.

The small example

Tiling *M* and *N* to utilize shared memory which is fast but small.

All threads collaboratively load subset of M and N before dot product.

Divide M and N into 2 × 2 tiles:
dot product by each thread are now divided into phases.



In each phase, all threads in a block collaborate to load a tile of M and a tile of N into the shared memory.
Every thread in a block load one M element and one N element.

The small example

Execution phases of a tiled matrix multiplication.

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

time →

The shared memory array for the M elements is called Mds

Each value in the shared memory is used twice; e. g. $M_{1,1}$

locality

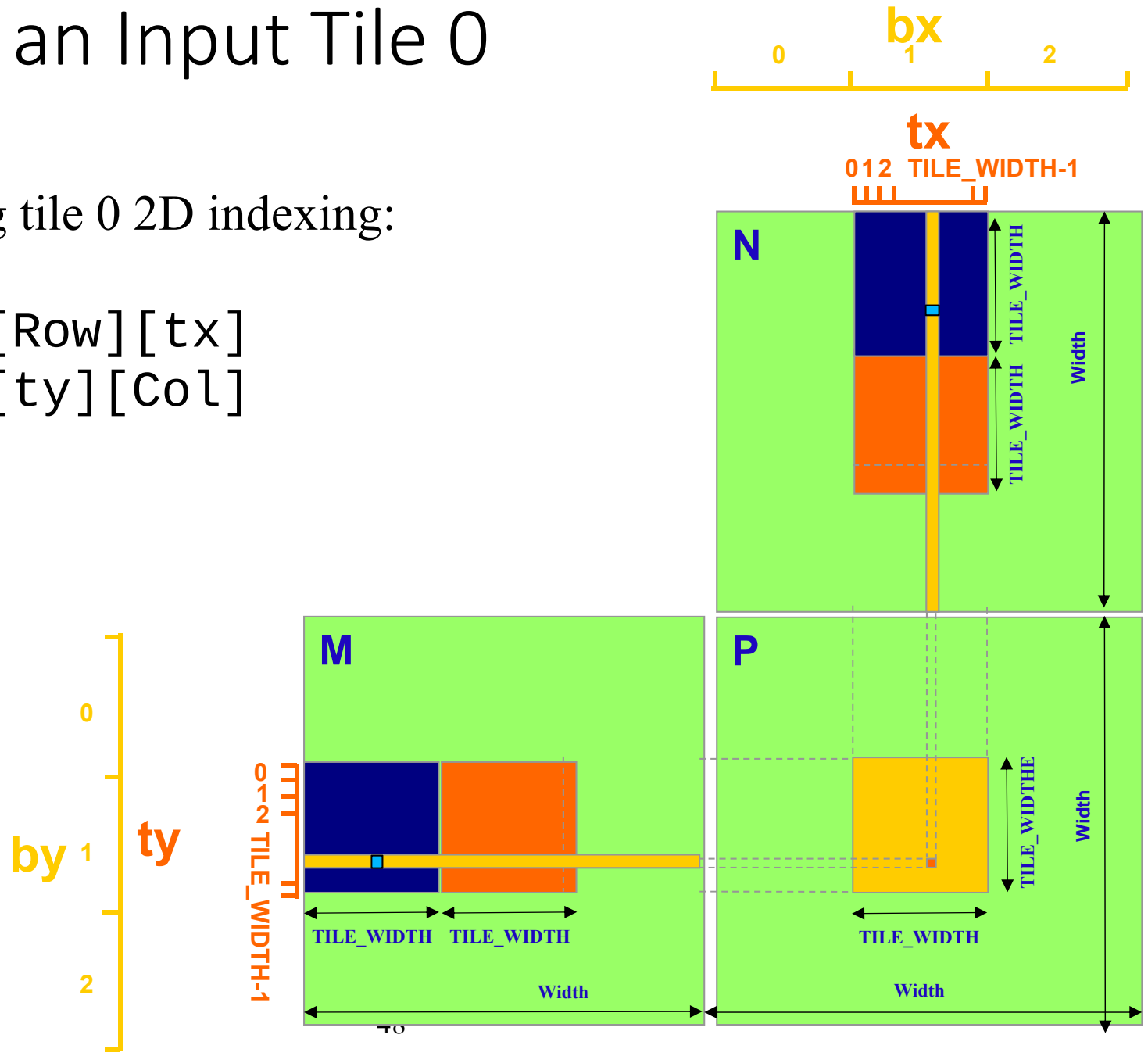
Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width){
1.  __shared__ float subTileM[TILE_WIDTH][TILE_WIDTH];
2.  __shared__ float subTileN[TILE_WIDTH][TILE_WIDTH];
3.  int bx = blockIdx.x;  int by = blockIdx.y;
4.  int tx = threadIdx.x; int ty = threadIdx.y;
    // Identify the row and column of the P element to work on
5.  int Row = by * TILE_WIDTH + ty;
6.  int Col = bx * TILE_WIDTH + tx;
7.  float Pvalue = 0;
    // Loop over the M and N tiles required to compute the P element
8.  for (int m = 0; m < Width/TILE_WIDTH; ++m) {
       // Collaborative loading of M and N tiles into shared memory
9.      subTileM[ty][tx] = M[Row*Width + m*TILE_WIDTH+tx];
10.     subTileN[ty][tx] = N[(m*TILE_WIDTH+ty)*Width+Col];
11.     __syncthreads();
12.     for (int k = 0; k < TILE_WIDTH; ++k)
13.         Pvalue += subTileM[ty][k] * subTileN[k][tx];
14.     __syncthreads();
15. }
16. P[Row*Width+Col] = Pvalue;
}
```

Loading an Input Tile 0

Accessing tile 0 2D indexing:

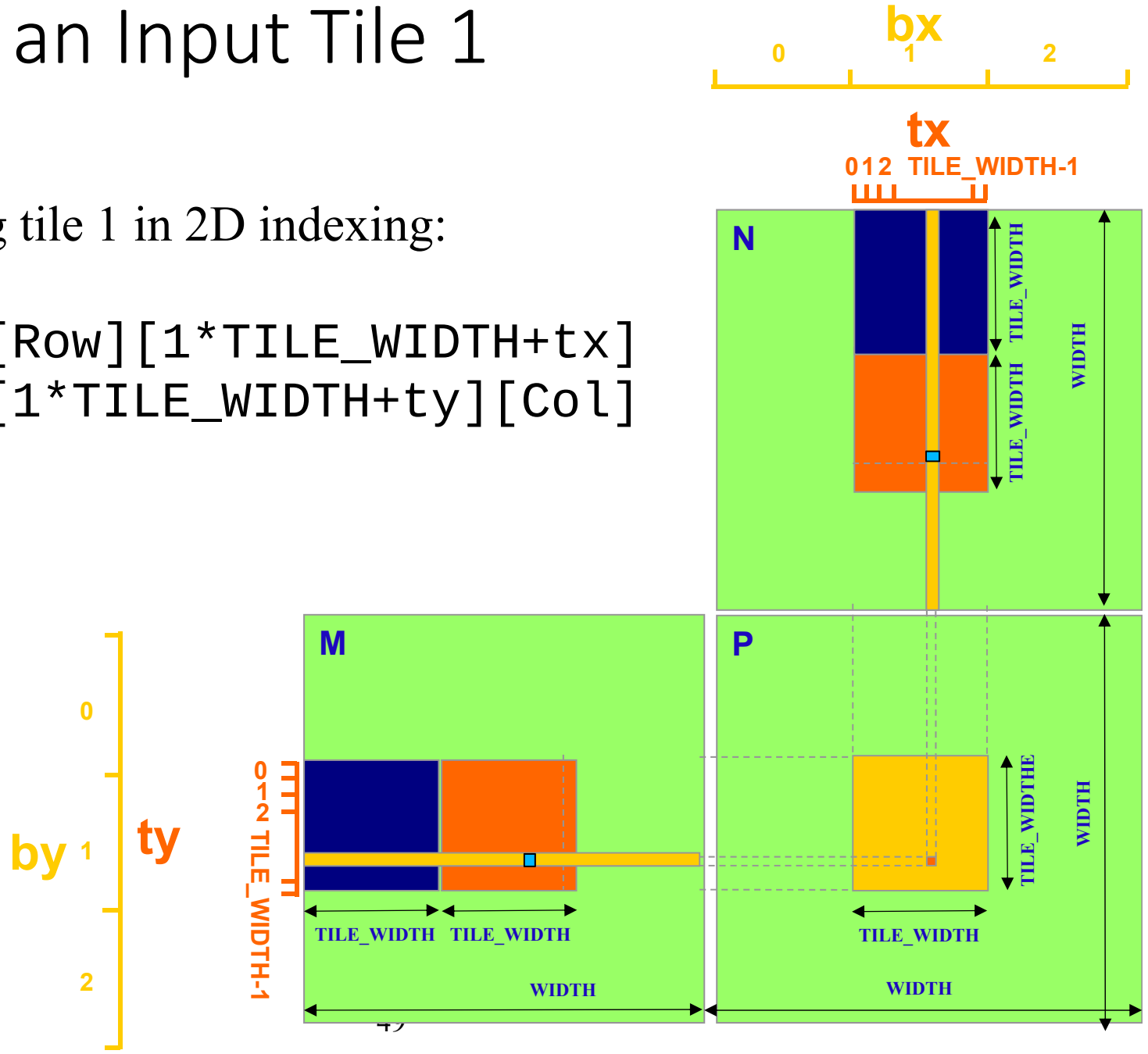
```
M[Row][tx]
N[ty][Col]
```



Loading an Input Tile 1

Accessing tile 1 in 2D indexing:

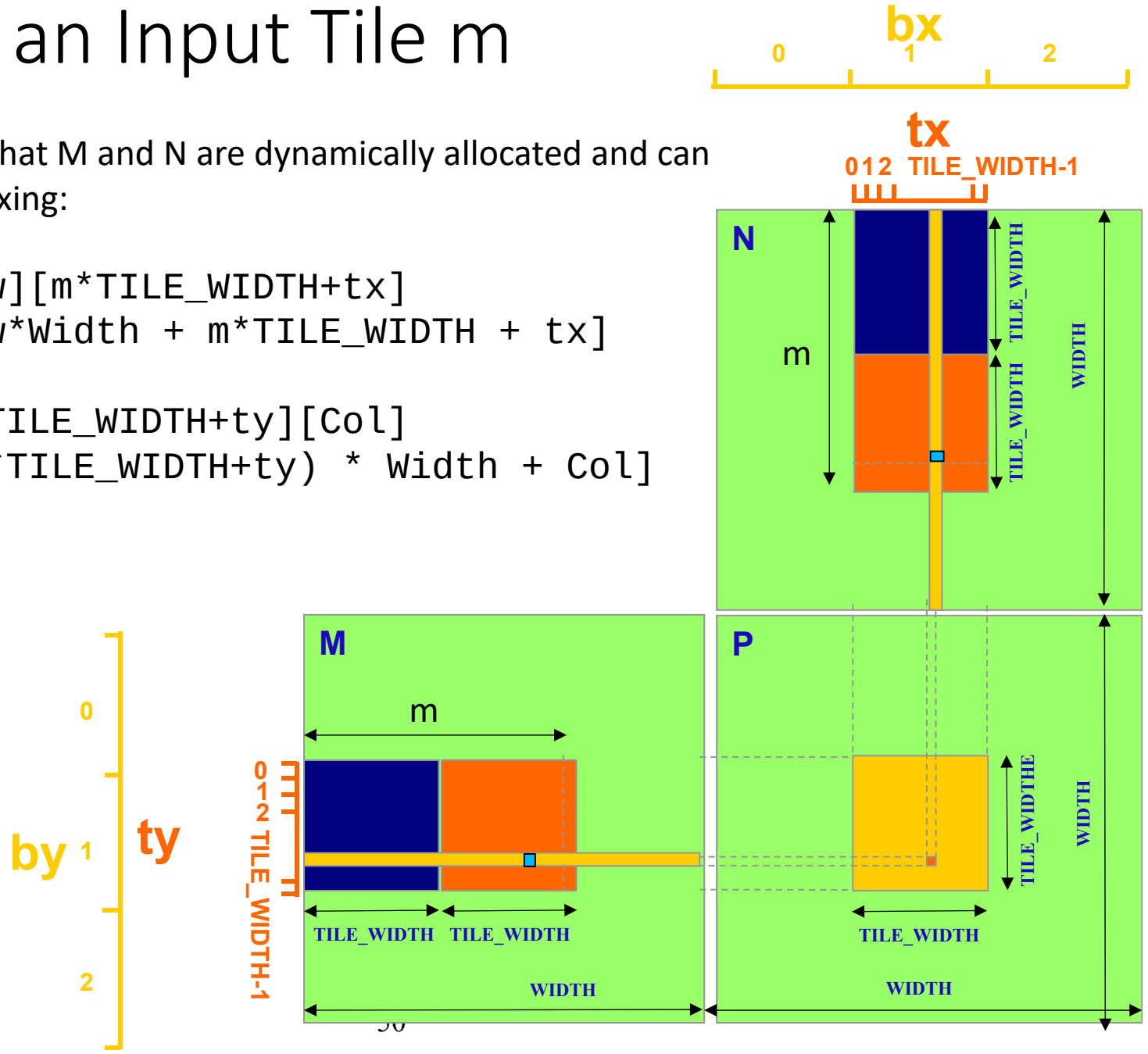
```
M[Row][1*TILE_WIDTH+tx]
N[1*TILE_WIDTH+ty][Col]
```



Loading an Input Tile m

However, recall that M and N are dynamically allocated and can only use 1D indexing:

```
M[Row][m*TILE_WIDTH+tx]
M[Row*Width + m*TILE_WIDTH + tx]

N[m*TILE_WIDTH+ty][Col]
N[(m*TILE_WIDTH+ty) * Width + Col]
```



Barrier Synchronization

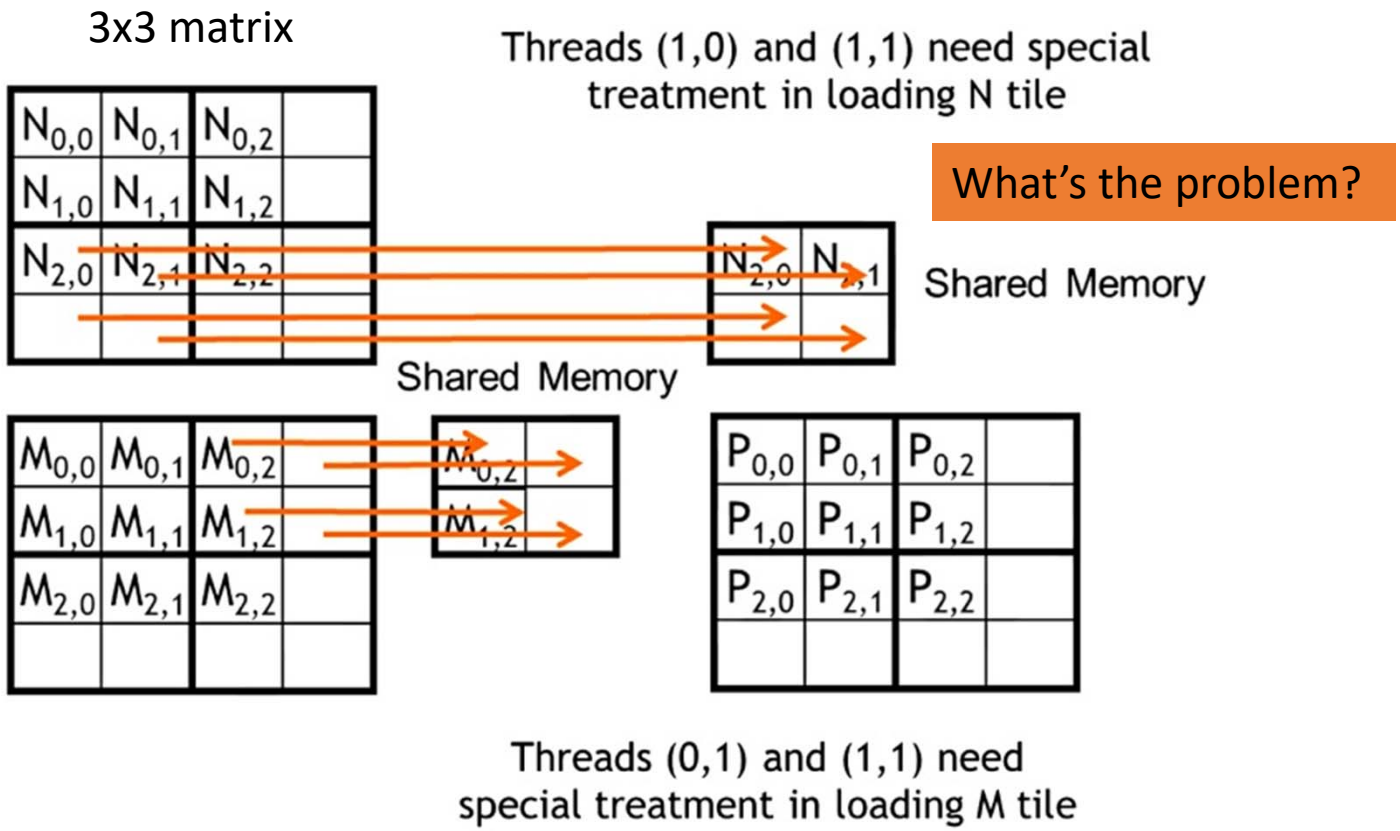
- An API function call in CUDA

__syncthreads()

- All threads in the same block must reach the __syncthreads() before any can move on

- Best used to coordinate tiled algorithms
  - To ensure that all elements of a tile are loaded
  - To ensure that all elements of a tile are consumed

Boundary Checks



Loading input matrix elements that are close to the edge– **phase 1 of Block0,0**.

Boundary Checks

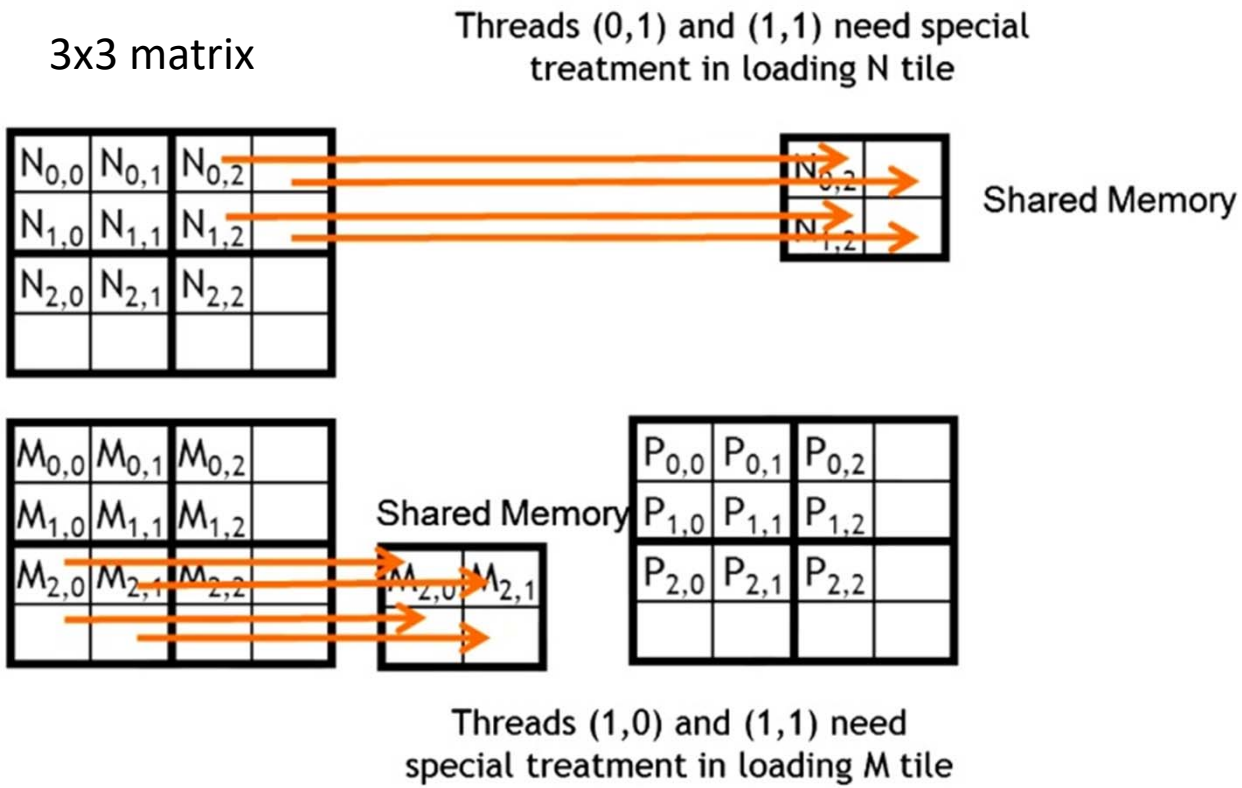


FIGURE 4.19: Loading input elements during **phase 0 of block1,0**.

Boundary Checks

```
// Loop over the M and N tiles required to compute P element
8. for (int ph = 0; ph < ceil(Width/(float)TILE_WIDTH); ++ph) {
   // Collaborative loading of M and N tiles into shared memory
9.  if ((Row< Width) && (ph*TILE_WIDTH+tx)< Width)
      Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
10. if ((ph*TILE_WIDTH+ty)<Width && Col<Width)
      Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*Width + Col];
11. __syncthreads();
12. for (int k = 0; k < TILE_WIDTH; ++k) {
13.   Pvalue += Mds[ty][k] * Nds[k][tx];
    }
14. __syncthreads();
   }
15. if ((Row<Width) && (Col<Width)) P[Row*Width + Col] = Pvalue;
```

Tiled matrix multiplication kernel with boundary condition checks.

Summary

- Importance of Memory Access Efficiency
  - --Matrix Multiplication
- Memory Types Overview
- Tiling for reduced memory traffic
  - --Matrix Multiplication using Shared Memory