

LECTURE 3 — MEMORY AND DATA LOCALITY



Memory access efficiency

Tiled Matrix Multiplication

Tiled Matrix Multiplication Kernel

Handling Boundary Conditions in Tiling

Tiled Kernel for Arbitrary Matrix Dimensions

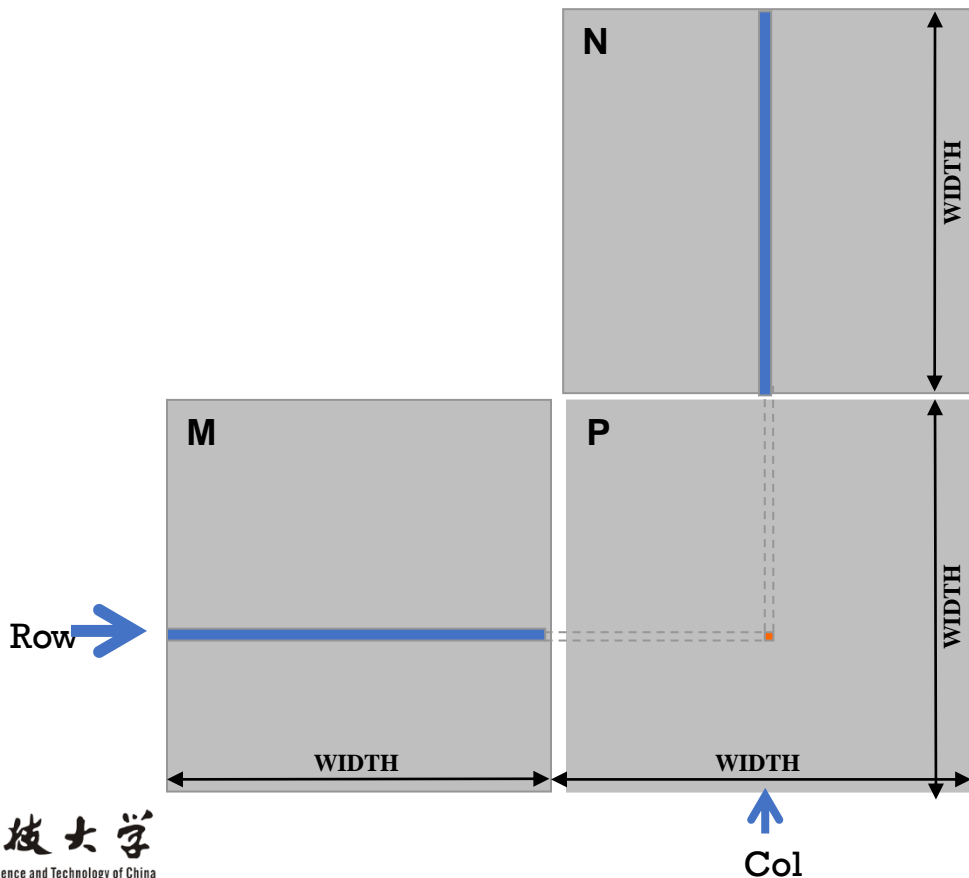


OBJECTIVE

- **To learn to effectively use the CUDA memory types in a parallel program**
 - Importance of memory access efficiency
 - Registers, shared memory, global memory
 - Scope and lifetime



EXAMPLE – MATRIX MULTIPLICATION



A BASIC MATRIX MULTIPLICATION

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
```

```
    // Calculate the row index of the P element and M
    int Row = blockIdx.y*blockDim.y+threadIdx.y;
```

```
    // Calculate the column index of P and N
    int Col = blockIdx.x*blockDim.x+threadIdx.x;
```

```
    if ((Row < Width) && (Col < Width)) {
        float Pvalue = 0;
        // each thread computes one element of the block sub-matrix
        for (int k = 0; k < Width; ++k) {
            Pvalue += M[Row*Width+k]*N[k*Width+Col];
        }
        P[Row*Width+Col] = Pvalue;
    }
}
```



HOW ABOUT PERFORMANCE ON A GPU

- **CGMA ratio** (**Compute to Global Memory Access ratio**) :The number of floating-point calculations performed for each access to the global memory within a region of a CUDA kernel.
- **The bigger, the better.**

What is the CGMA of Basic Matrix Multiplication?



CGMA OF BASIC MATRIX MULTIPLICATION

- **All threads access global memory for their input matrix elements**
 - One memory accesses (4 bytes) per floating-point addition
 - CGMA ratio =1
 - 4B/s of memory bandwidth/FLOPS
- **Assume a GPU with**
 - **Peak floating-point rate 1,500 GFLOPS with 200 GB/s DRAM bandwidth**
 - $4 \times 1,500 = 6,000$ GB/s required to achieve peak FLOPS rating
 - The 200 GB/s memory bandwidth limits the execution at 50 GFLOPS
- **This limits the execution rate to 3.3% (50/1500) of the peak floating-point execution rate of the device!**
- **Need to drastically cut down memory accesses to get close to the 1,500 GFLOPS**
- **To achieve peak 1,500 GFLOPS, we need CGMA=30**

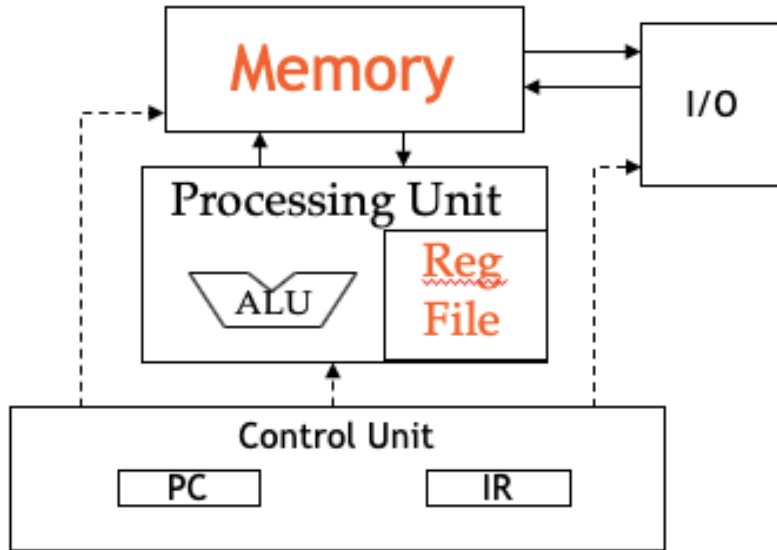


QUESTION

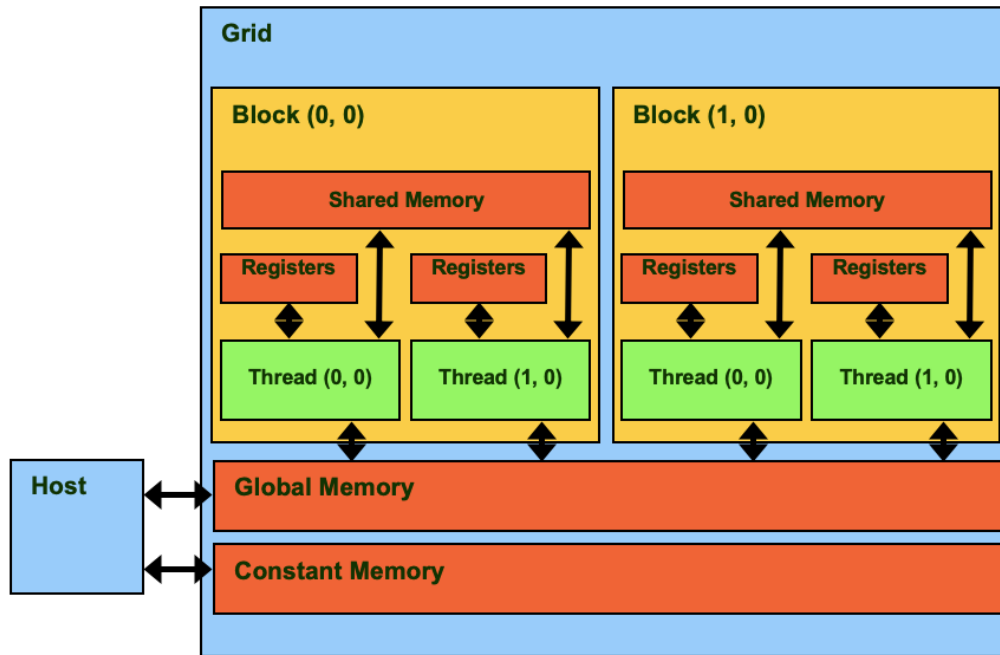
- **How to improve memory access efficiency?**
 - To increase calculation
 - To improve utilization
 - To utilize the memory hierarchy
 - Global memory
 - Shared memory
 - Register file



MEMORY AND REGISTERS IN THE VON-NEUMANN MODEL



PROGRAMMER VIEW OF CUDA MEMORIES



DECLARING CUDA VARIABLES

Variable declaration	Memory	Scope	Lifetime
int LocalVar;	register	thread	thread
__device__ __shared__ int SharedVar;	shared	block	block
__device__ int GlobalVar;	global	grid	application
__device__ __constant__ int ConstantVar;	constant	grid	application

- **__device__** is optional when used with **__shared__**, or **__constant__**
- **Automatic variables** reside in a **register**
 - **Except per-thread arrays** that reside in global memory



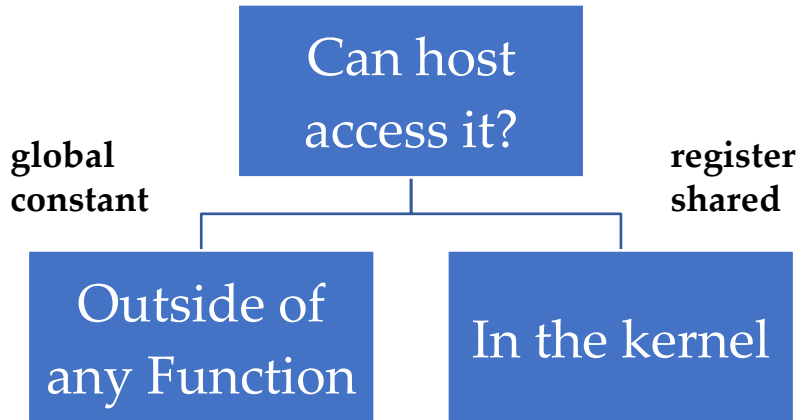
EXAMPLE:

SHARED MEMORY VARIABLE DECLARATION

```
void blurKernel(unsigned char * in, unsigned char * out, int w, int h)
{
    __shared__ float ds_in[TILE_WIDTH][TILE_WIDTH];
    ...
}
```



WHERE TO DECLARE VARIABLES?

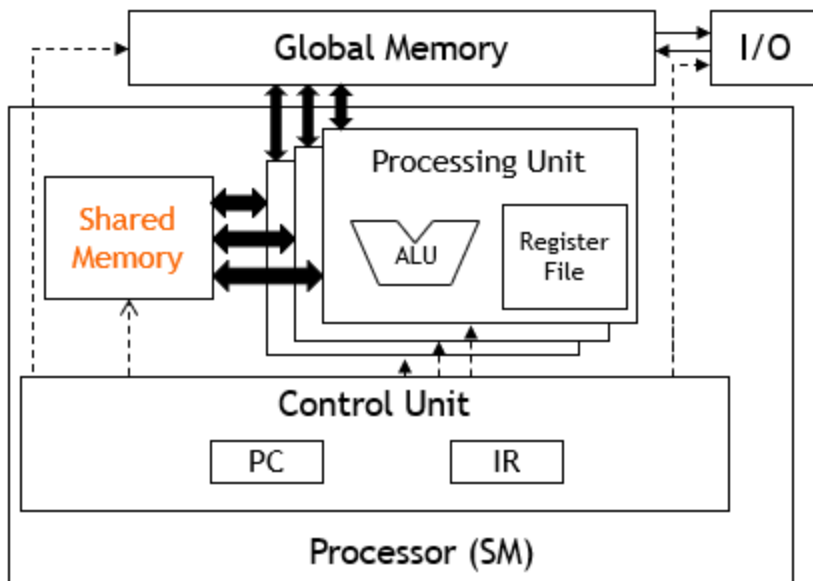


SHARED MEMORY IN CUDA

- **A special type of memory whose contents are explicitly defined and used in the kernel source code**
 - **One in each SM**
 - **Accessed at much higher speed (in both latency and throughput) than global memory**
 - **Scope of access and sharing**
 - **Lifetime – thread block, contents will disappear after the corresponding thread finishes terminates execution**
 - **Accessed by memory load/store instructions**
 - **A form of **scratchpad memory (SPRAM)****



HARDWARE VIEW OF CUDA MEMORIES



CUDA Memories

Tiled Matrix Multiplication

Tiled Matrix Multiplication Kernel

Handling Boundary Conditions in Tiling

Tiled Kernel for Arbitrary Matrix Dimensions



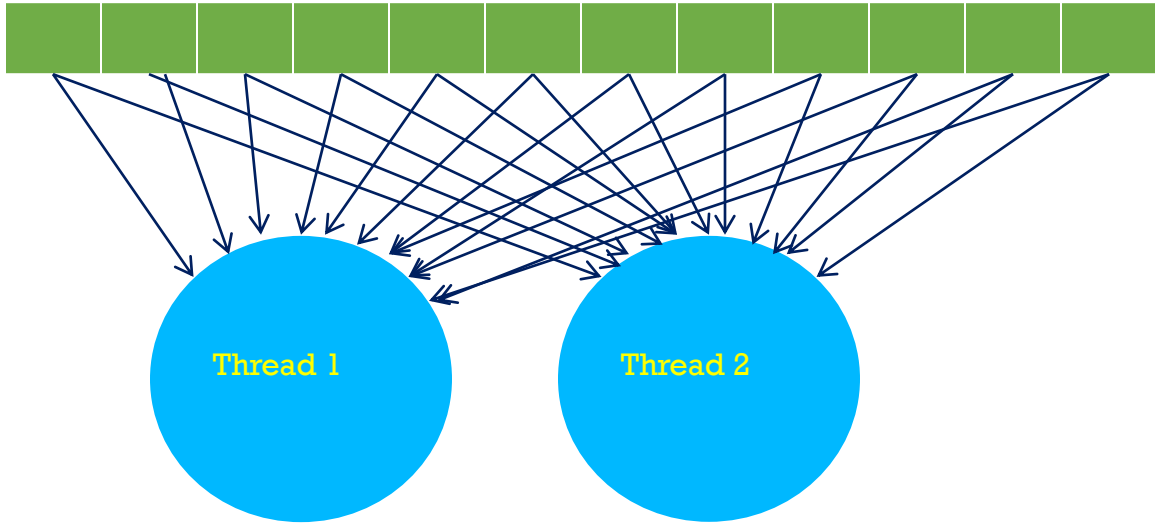
OBJECTIVE

- **To understand the motivation and ideas for tiled parallel algorithms**
 - **Reducing the limiting effect of memory bandwidth on parallel kernel performance**
 - **Tiled algorithms and barrier synchronization**

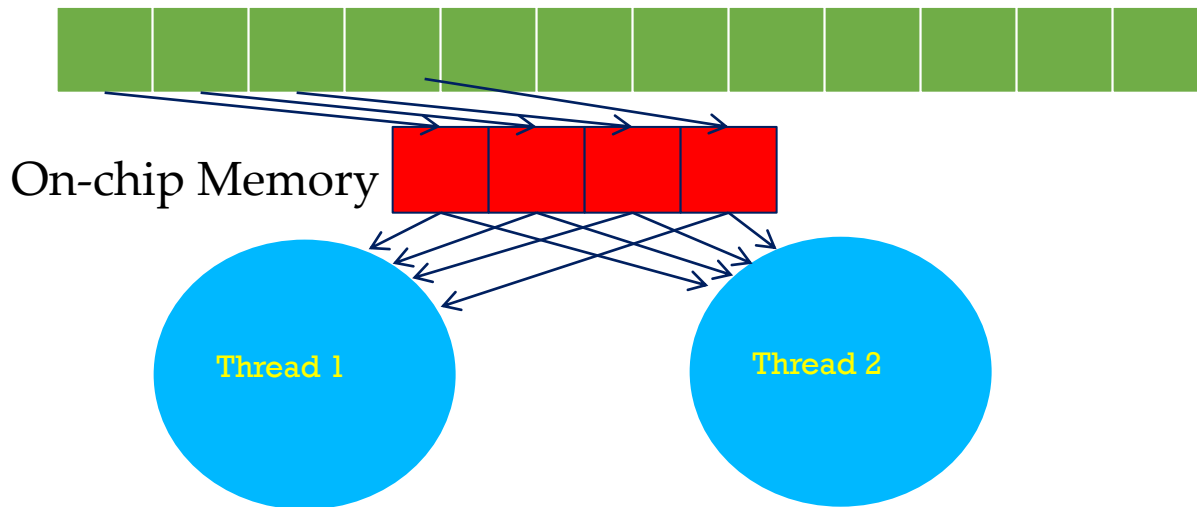


GLOBAL MEMORY ACCESS PATTERN OF THE BASIC MATRIX MULTIPLICATION KERNEL

Global Memory



TILING/BLOCKING - BASIC IDEA

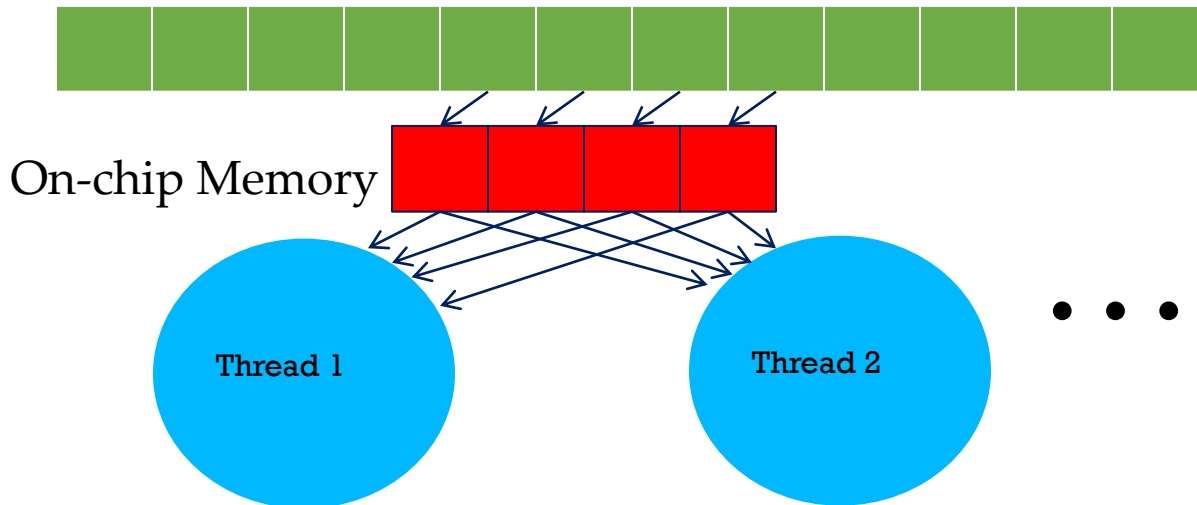


Divide the global memory content into tiles

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



TILING/BLOCKING - BASIC IDEA



BASIC CONCEPT OF TILING

- **In a congested traffic system, significant reduction of vehicles can greatly improve the delay seen by all vehicles**
 - Carpooling for commuters
 - Tiling for global memory accesses
 - drivers = threads accessing their memory data operands
 - cars = memory access requests



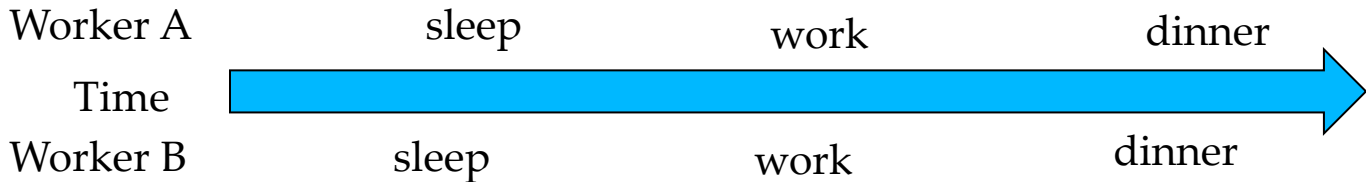
SOME COMPUTATIONS ARE MORE CHALLENGING TO TILE

- **Some carpools may be easier than others**
 - Car pool participants need to have similar work schedule
 - Some vehicles may be more suitable for carpooling
- **Similar challenges exist in tiling**



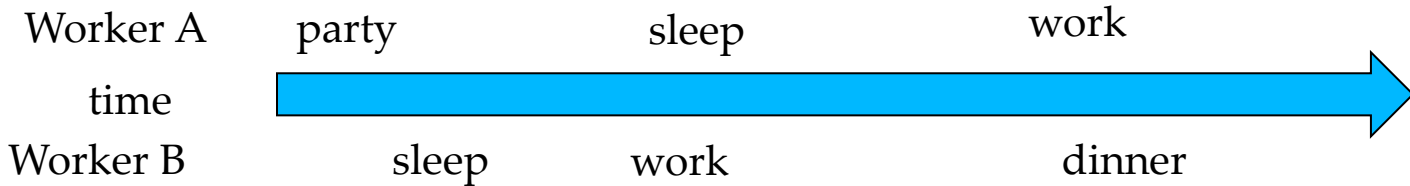
CARPOOLS NEED SYNCHRONIZATION.

- **Good: when people have similar schedule**



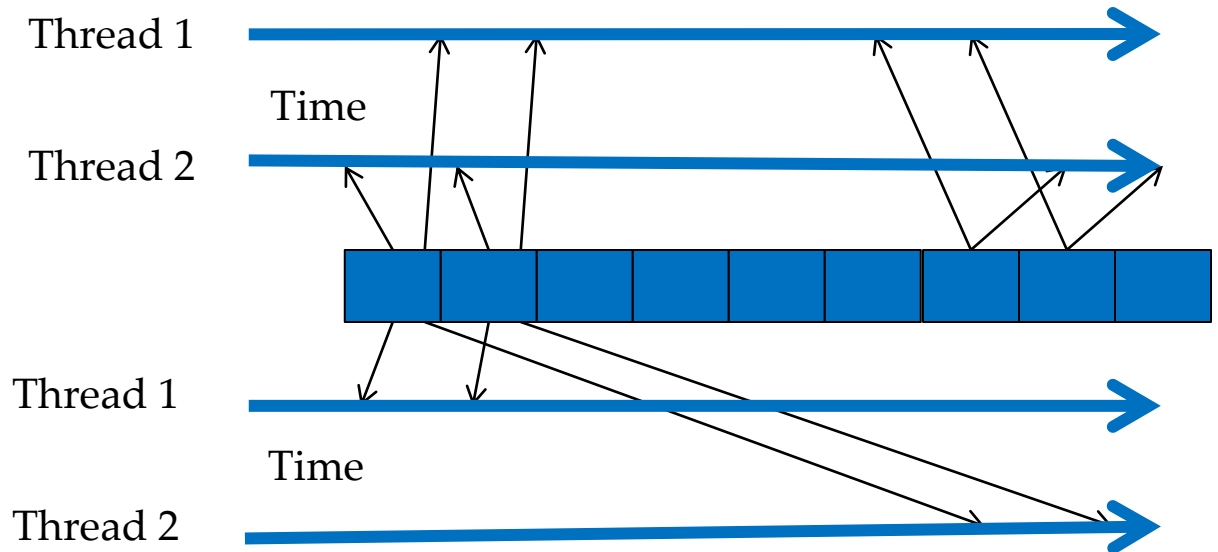
CARPOOLS NEED SYNCHRONIZATION.

- **Bad: when people have very different schedule**



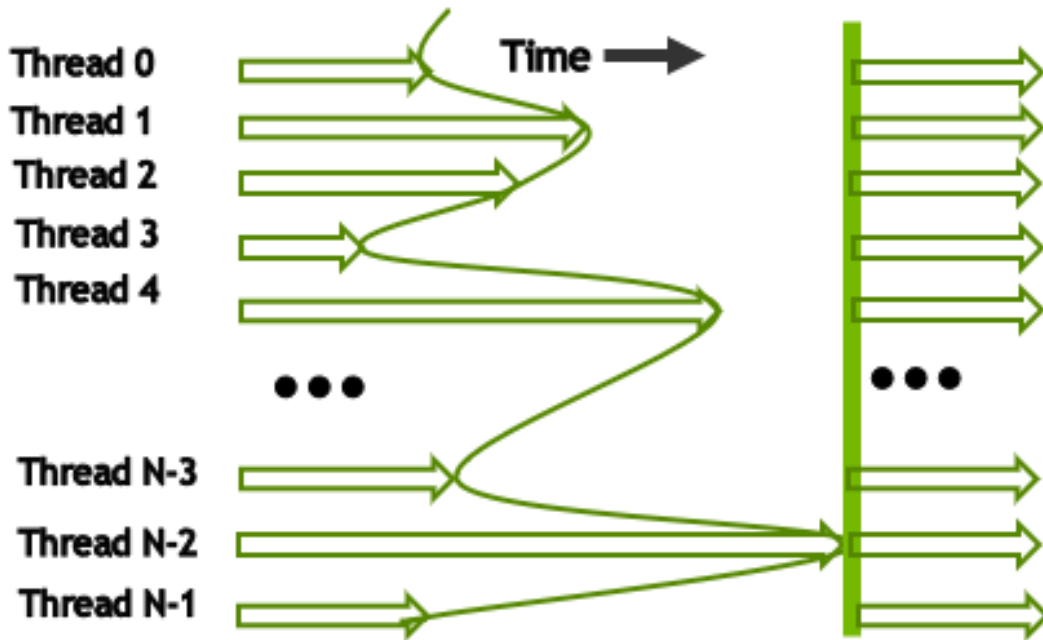
SAME WITH TILING

- **Good:** when threads have similar access timing



- **Bad:** when threads have very different timing

BARRIER SYNCHRONIZATION FOR TILING



OUTLINE OF TILING TECHNIQUE

- 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



CUDA Memories

Tiled Matrix Multiplication

Tiled Matrix Multiplication Kernel

Handling Boundary Conditions in Tiling

Tiled Kernel for Arbitrary Matrix Dimensions



OBJECTIVE

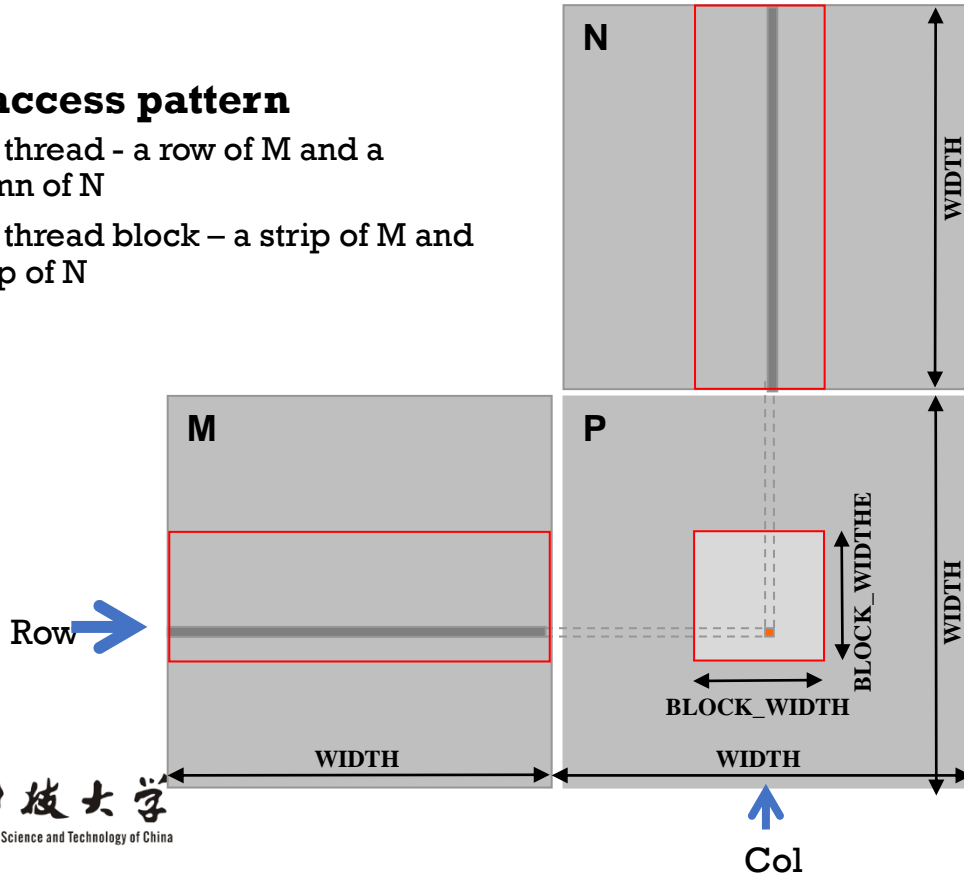
- **To understand the design of a tiled parallel algorithm for matrix multiplication**
 - Loading a tile
 - Phased execution
 - Barrier Synchronization



MATRIX MULTIPLICATION

■ Data access pattern

- Each thread - a row of M and a column of N
- Each thread block – a strip of M and a strip of N



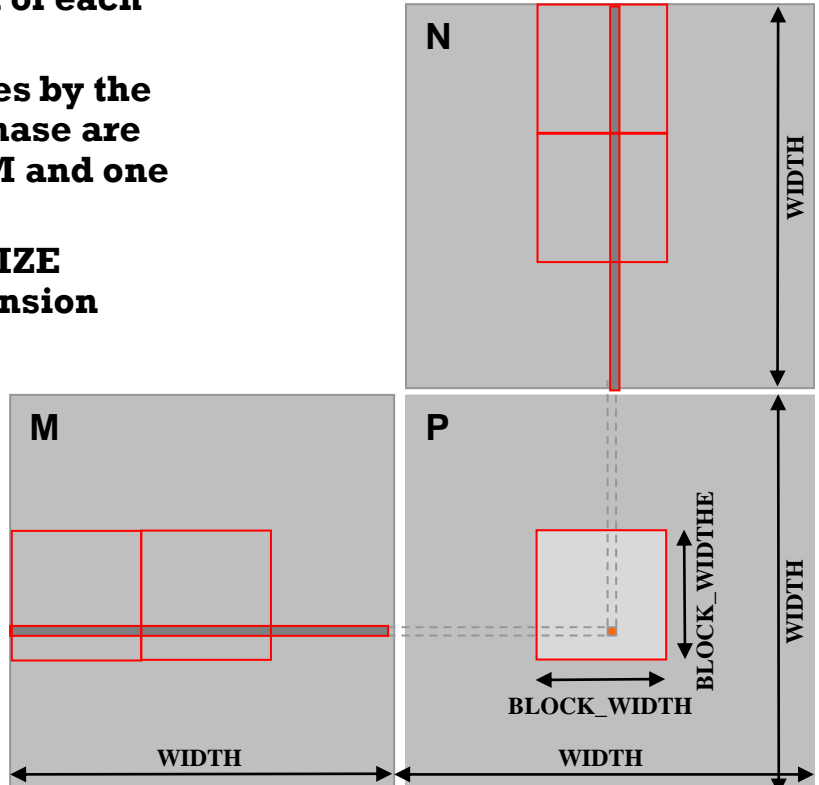
TILED MATRIX MULTIPLICATION

- 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

$$A = \begin{pmatrix} A_{11} & \cdots & A_{1r} \\ \vdots & \cdots & \vdots \\ A_{s1} & \cdots & A_{sr} \end{pmatrix} B = \begin{pmatrix} B_{11} & \cdots & B_{1r} \\ \vdots & \cdots & \vdots \\ B_{t1} & \cdots & B_{tr} \end{pmatrix}$$

$$AB = \begin{pmatrix} C_{11} & \cdots & C_{1r} \\ \vdots & \cdots & \vdots \\ C_{s1} & \cdots & C_{sr} \end{pmatrix}$$

$$C_{ij} = \sum_{k=1}^t A_{ik} B_{kj} \quad (i = 1, \dots, s; j = 1, \dots, r)$$

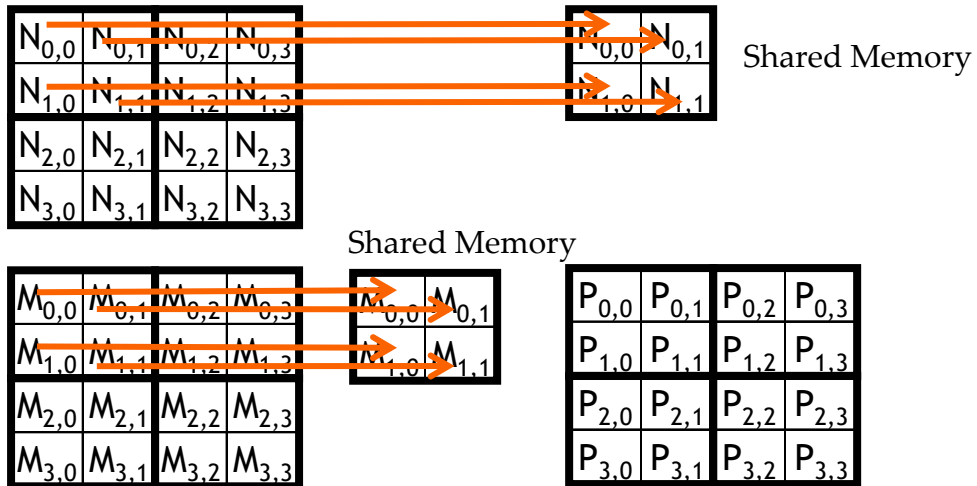


LOADING A TILE

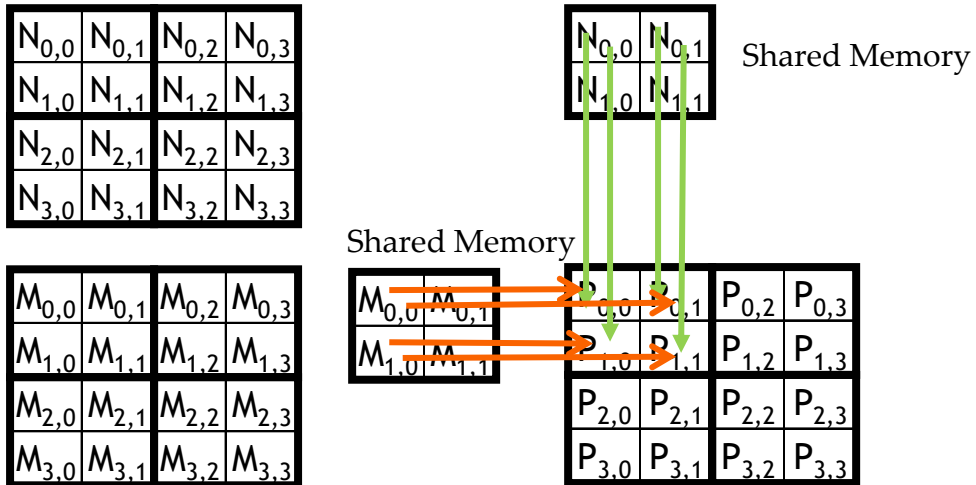
- **All threads in a block participate**
 - Each thread loads one M element and one N element in tiled code



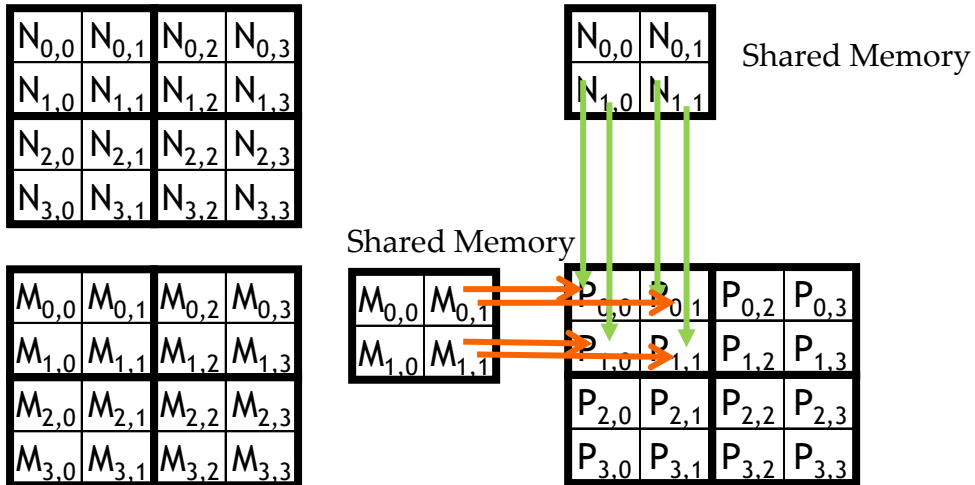
Phase 0 Load for Block (0,0)



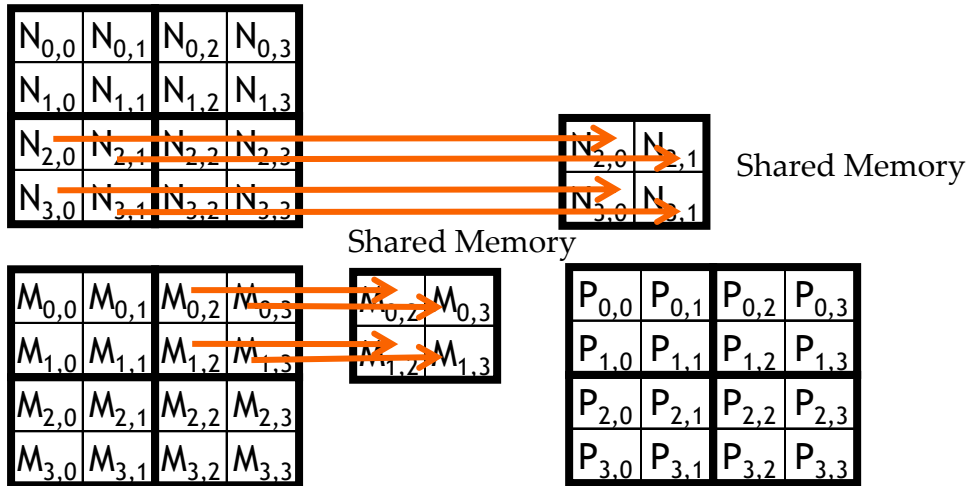
Phase 0 Use for Block (0,0) (iteration 0)



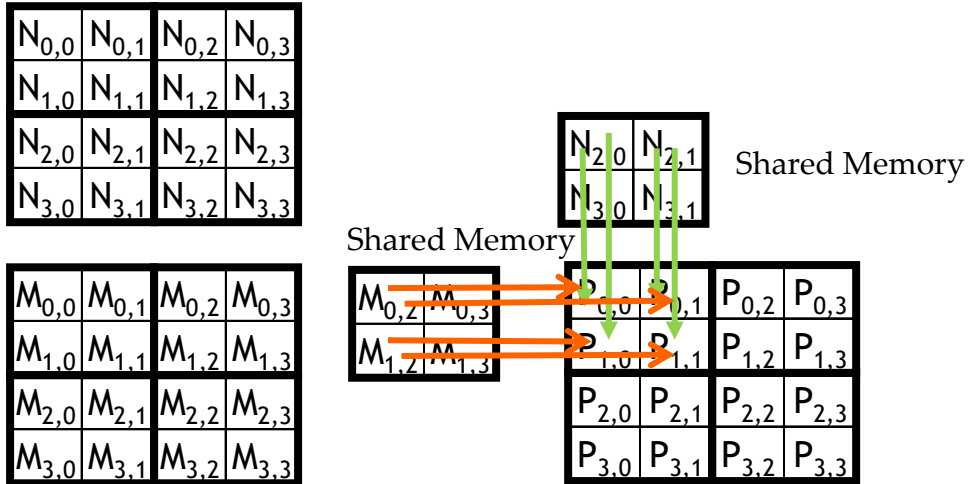
Phase 0 Use for Block (0,0) (iteration 1)



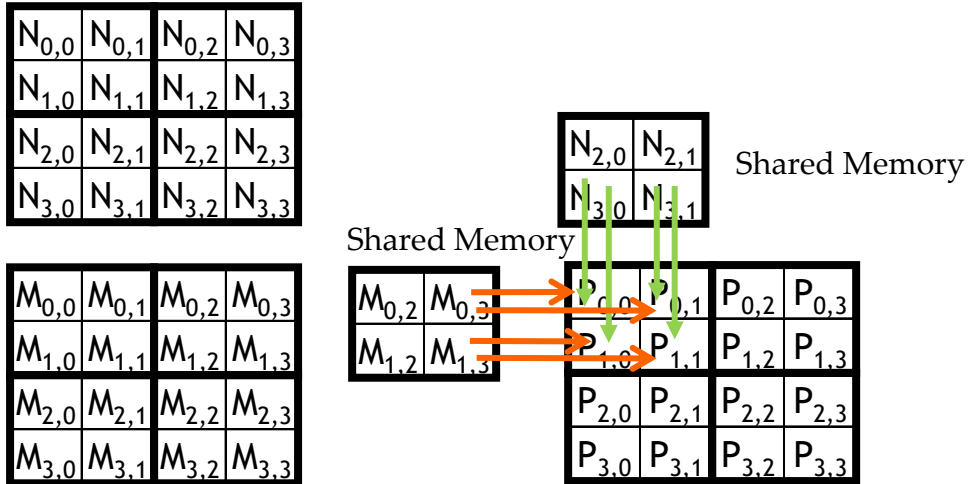
Phase 1 Load for Block (0,0)



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




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



Execution Phases of Toy Example

	Phase 0			Phase 1		
thread _{0,0}	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 

Execution Phases of Toy Example (cont.)

	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

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**



CUDA Memories

Tiled Matrix Multiplication

Tiled Matrix Multiplication Kernel

Handling Boundary Conditions in Tiling

Tiled Kernel for Arbitrary Matrix Dimensions



OBJECTIVE

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



LOADING INPUT TILE 0 OF M (PHASE 0)

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

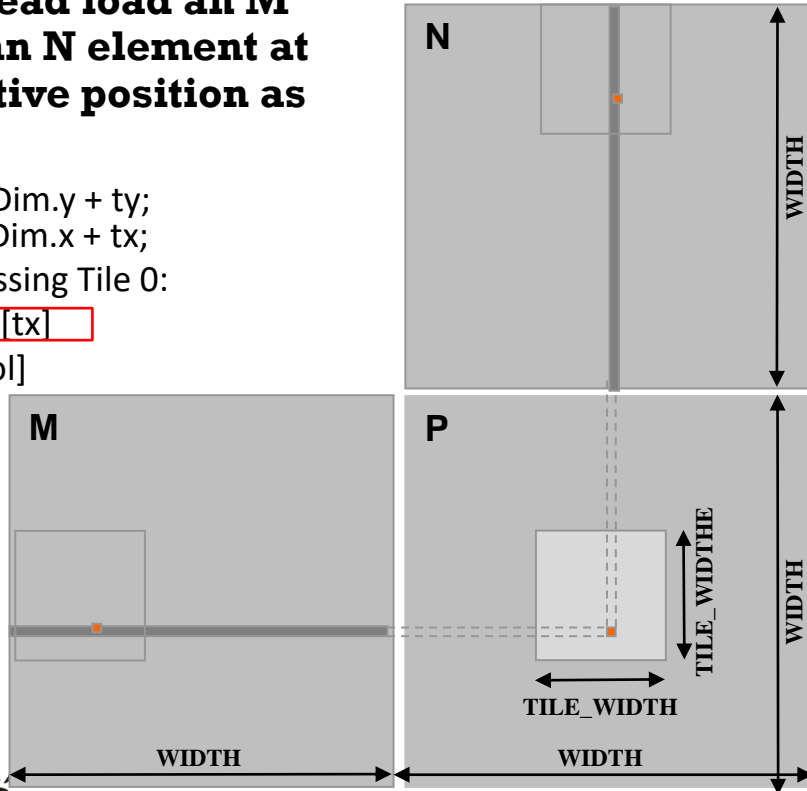
```
int Row = by * blockDim.y + ty;
```

```
int Col = bx * blockDim.x + tx;
```

2D indexing for accessing Tile 0:

$M[Row][tx]$

$N[ty][Col]$



LOADING INPUT TILE 0 OF N (PHASE 0)

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

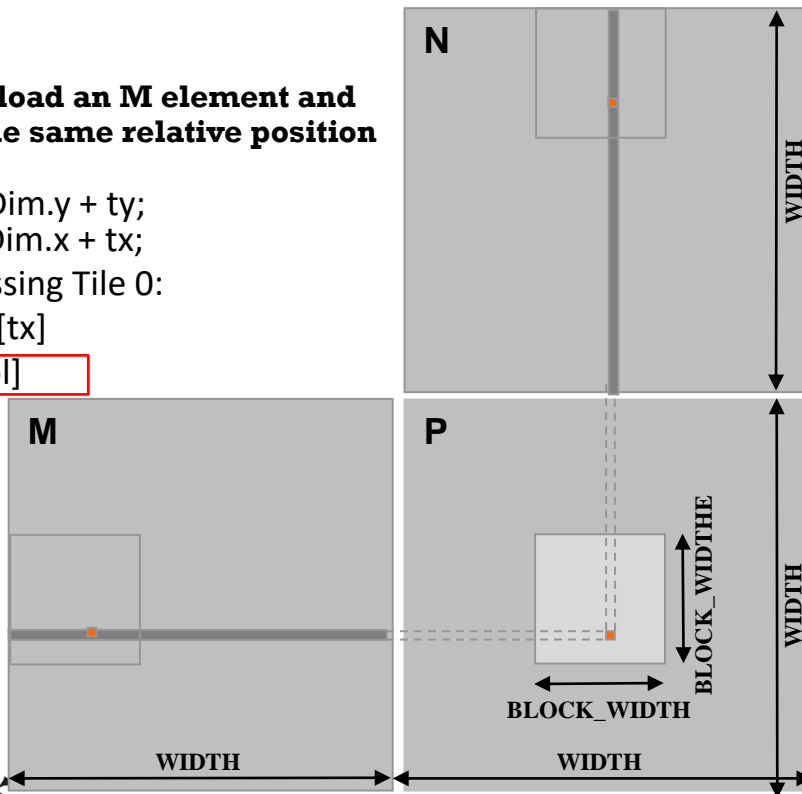
```
int Row = by * blockDim.y + ty;
```

```
int Col = bx * blockDim.x + tx;
```

2D indexing for accessing Tile 0:

$M[\text{Row}][\text{tx}]$

$N[\text{ty}][\text{Col}]$

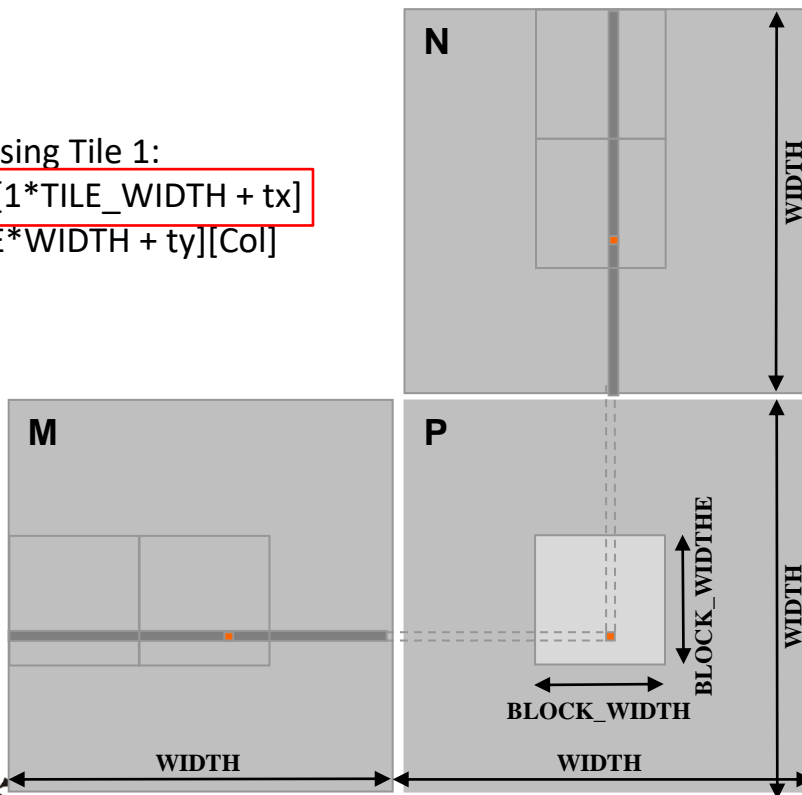


LOADING INPUT TILE 1 OF M (PHASE 1)

2D indexing for accessing Tile 1:

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

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

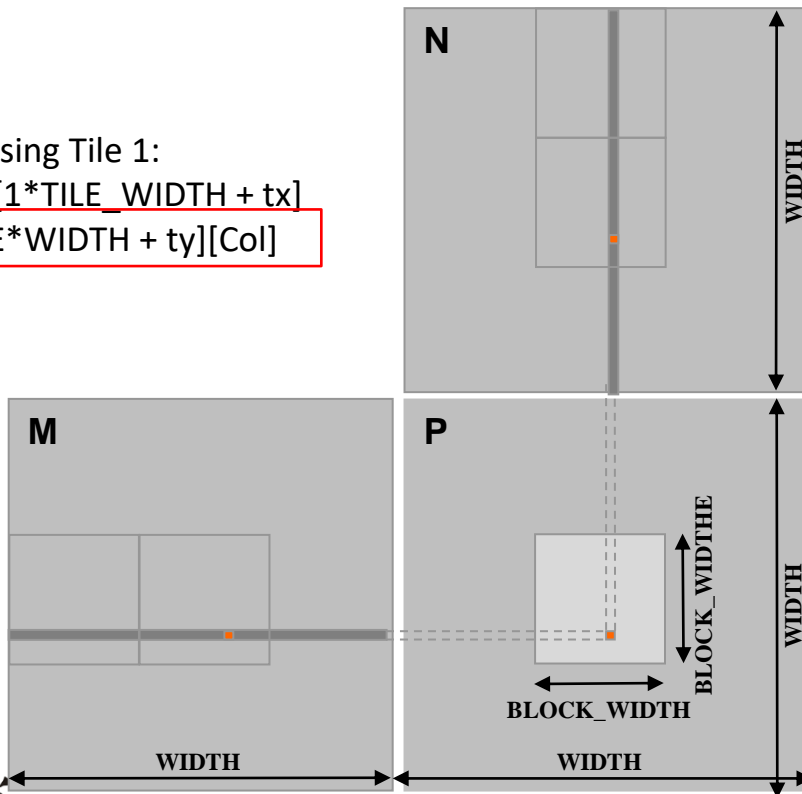


LOADING INPUT TILE 1 OF N (PHASE 1)

2D indexing for accessing Tile 1:

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

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



Tiled Matrix Multiplication Kernel

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

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

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

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

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


Tiled Matrix Multiplication Kernel

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

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

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

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

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

Tiled Matrix Multiplication Kernel

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

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

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

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

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

TILE (THREAD BLOCK) SIZE CONSIDERATIONS

- Each **thread block** should have many threads
 - **TILE_WIDTH** of 16 gives $16*16 = 256$ threads
 - **TILE_WIDTH** of 32 gives $32*32 = 1024$ threads
- For 16, in each phase, each block performs $2*256 = 512$ float loads from global memory for $256 * (2*16) = 8,192$ mul/add operations. (16 floating-point operations for each memory load)
- **CGMA=?**
- For 32, in each phase, each block performs $2*1024 = 2048$ float loads from global memory for $1024 * (2*32) = 65,536$ mul/add operations. (32 floating-point operation for each memory load)
- **CGMA=?**



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 $2 \times 256 \times 4\text{B} = 2\text{KB}$ of shared memory.
 - For 16KB shared memory, one can potentially have up to 8 thread blocks executing
 - This allows up to $8 \times 512 = 4,096$ pending loads. (2 per thread, 256 threads per block)
 - **The next `TILE_WIDTH 32` would lead to $2 \times 32 \times 32 \times 4 \text{ Byte} = 8\text{K Byte}$ 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**



CUDA Memories

Tiled Matrix Multiplication

Tiled Matrix Multiplication Kernel

Handling Boundary Conditions in Tiling

Tiled Kernel for Arbitrary Matrix Dimensions



OBJECTIVE

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

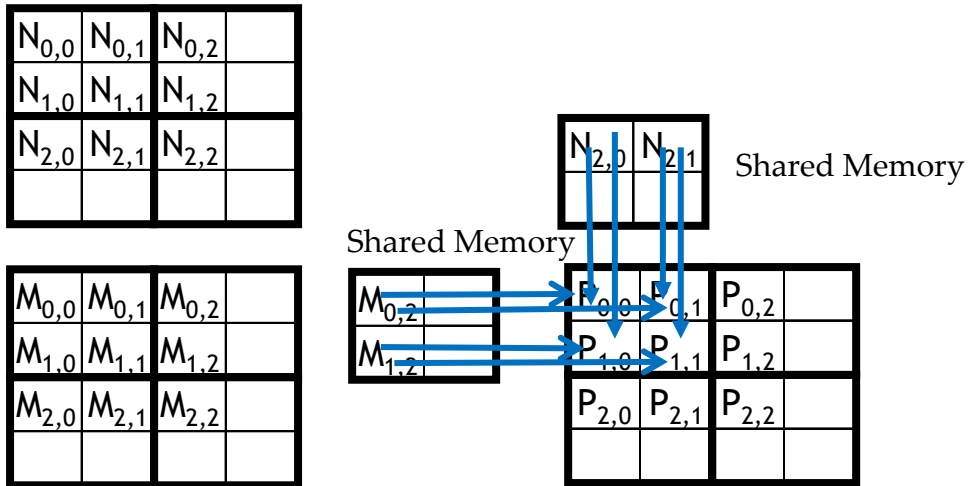


HANDLING MATRIX OF ARBITRARY SIZE

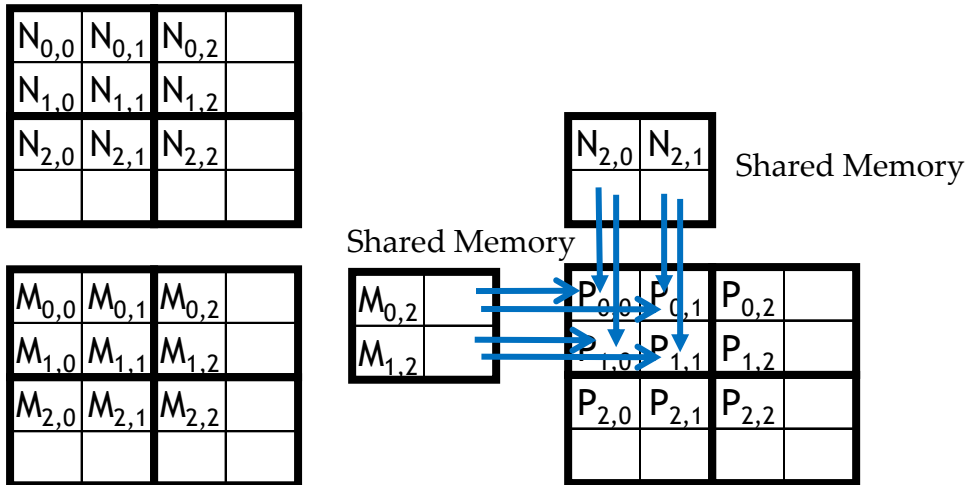
- **The tiled matrix multiplication kernel we presented so far can handle only square matrices whose dimensions (Width) are multiples of the tile width (TILE_WIDTH)**
 - **However, real applications need to handle arbitrary sized matrices.**
 - **One could pad (add elements to) the rows and columns into multiples of the tile size, but would have significant space and data transfer time overhead.**
- **We will take a different approach.**



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



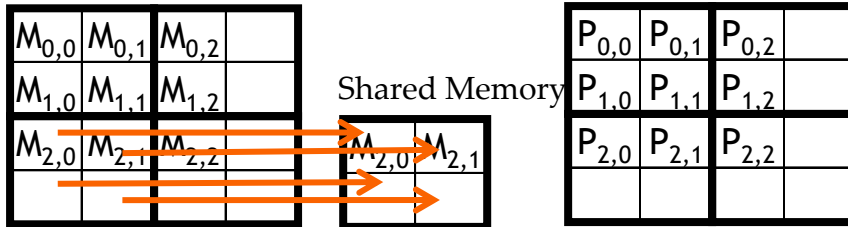
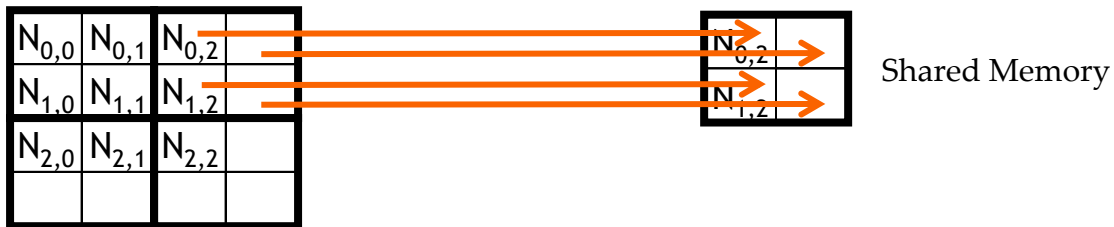
Phase 1 Use for Block (0,0) (iteration 1)



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

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

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



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

Major Cases in Toy Example

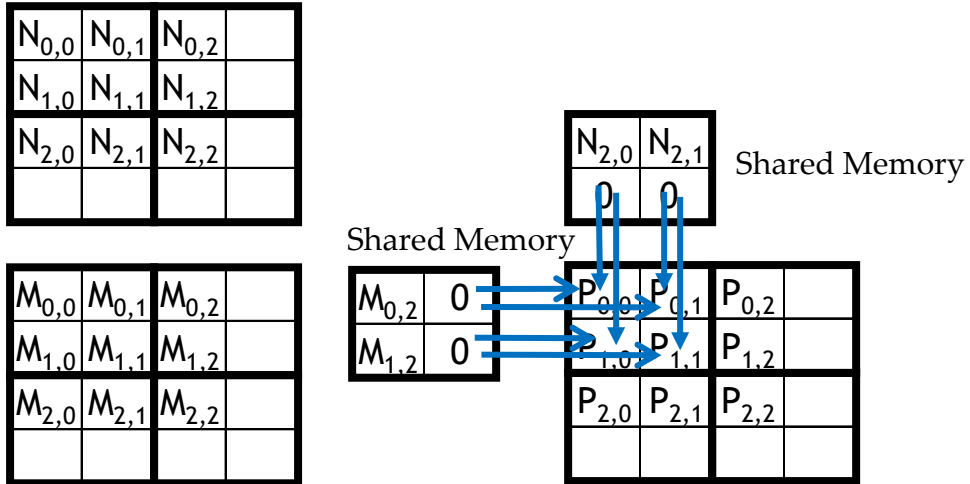
- Threads that do not calculate valid P elements but still need to participate in loading the input tiles
 - Phase 0 of Block(1,1), Thread(1,0), assigned to calculate non-existent $P[3,2]$ but need to participate in loading tile element $N[1,2]$
- Threads that calculate valid P elements may attempt to load non-existing input elements when loading input tiles
 - Phase 0 of Block(0,0), Thread(1,0), assigned to calculate valid $P[1,0]$ but attempts to load non-existing $N[3,0]$

A “SIMPLE” SOLUTION

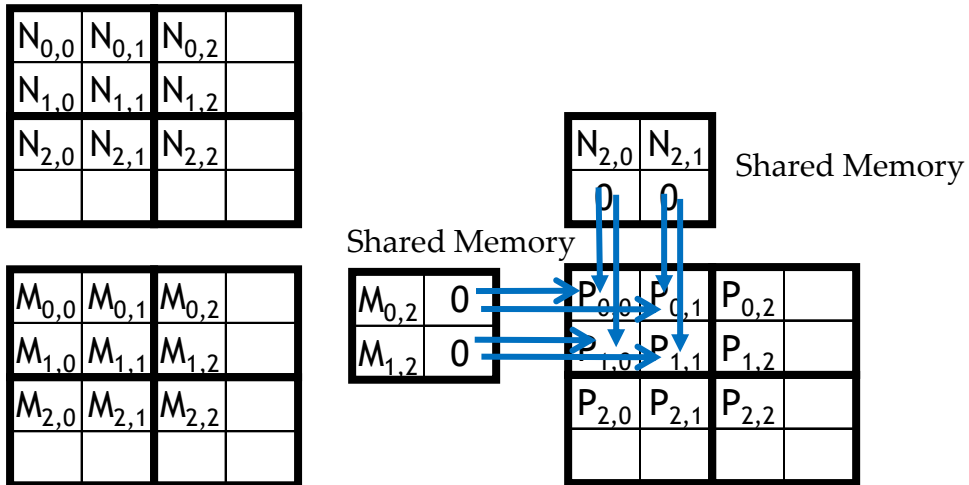
- **When a thread is to load any input element, test if it is in the valid index range**
 - If valid, proceed to load
 - Else, do not load, just write a 0
- **Rationale: a 0 value will ensure that that the multiply-add step does not affect the final value of the output element**
- **The condition tested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

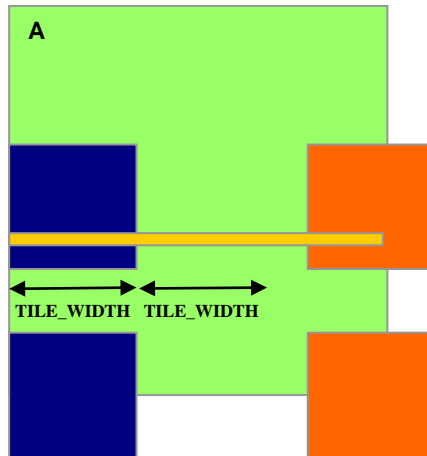


Phase 1 Use for Block (0,0) (iteration 1)



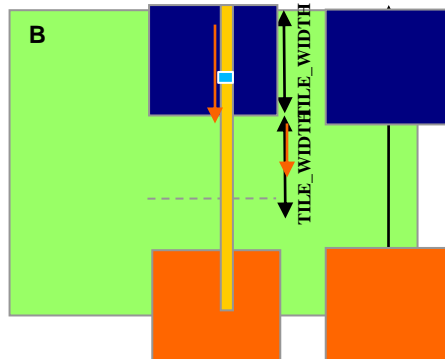
Boundary Condition for Input M Tile

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



Boundary Condition for Input N Tile

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



LOADING ELEMENTS — WITH BOUNDARY CHECK

```

8   for (int p = 0; p < (Width-1) / TILE_WIDTH + 1; ++p) {
    ++   if (Row < Width && t * TILE_WIDTH + tx < Width) {
9       ds_M[ty][tx] = M[Row * Width + p * TILE_WIDTH + tx];
    ++   } else {
    ++       ds_M[ty][tx] = 0.0;
    ++   }
    ++   if (p * TILE_WIDTH + ty < Width && Col < Width) {
10      ds_N[ty][tx] = N[(p * TILE_WIDTH + ty) * Width + Col];
    ++   } else {
    ++       ds_N[ty][tx] = 0.0;
    ++   }
11   __syncthreads();

```



INNER PRODUCT – BEFORE AND AFTER

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



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
- **We have presented square matrix multiplication, a special case**
- **The kernel function needs to be generalized to handle general rectangular matrices**
 - The Width argument is replaced by three arguments: j, k, l
 - When Width is used to refer to the height of M or height of P , replace it with j
 - When Width is used to refer to the width of M or height of N , replace it with k
 - When Width is used to refer to the width of N or width of P , replace it with l

