www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Memory and Data Locality

Antonio J. Peña

Based on material from NVIDIA's GPU Teaching Kit

Barcelona, July 4-6 2016

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

CUDA MEMORIES

Review: Image Blur Kernel.

```
// Get the average of the surrounding 2xBLUR_SIZE x 2xBLUR_SIZE box
for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow) {
   for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol) {
      int curRow = Row + blurRow;
      int curCol = Col + blurCol;
      // Verify we have a valid image pixel
      if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
         pixVal += in[curRow * w + curCol];
         pixels++; // Keep track of number of pixels in the accumulated total
      }
   }
}

// Write our new pixel value out
out[Row * w + Col] = (unsigned char)(pixVal / pixels);
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

3

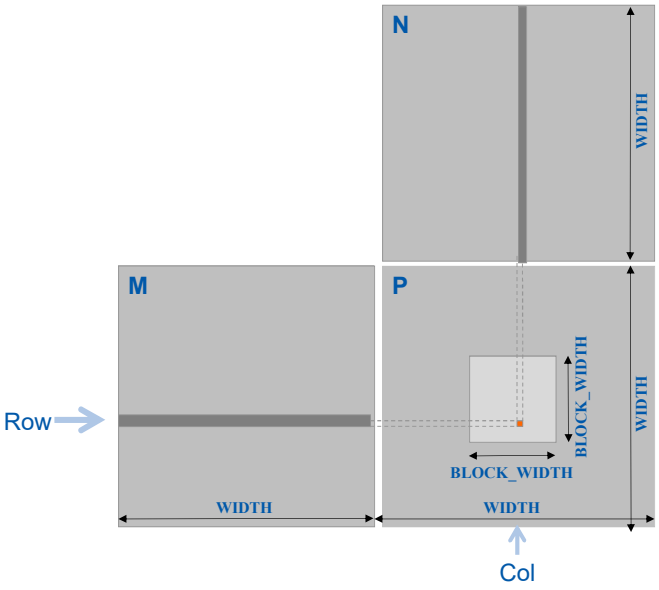
How about performance on a GPU

- All threads access global memory for their input matrix elements
  - One memory access (4 bytes) per floating-point addition
  - 4B/s of memory bandwidth/FLOPS
- Assume a GPU with
  - Peak floating-point rate 1,500 GFLOPS with 200 GB/s DRAM bandwidth
  - 4*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 the1,500 GFLOPS

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

4

Example – Matrix Multiplication



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

5

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;
  }

}
```

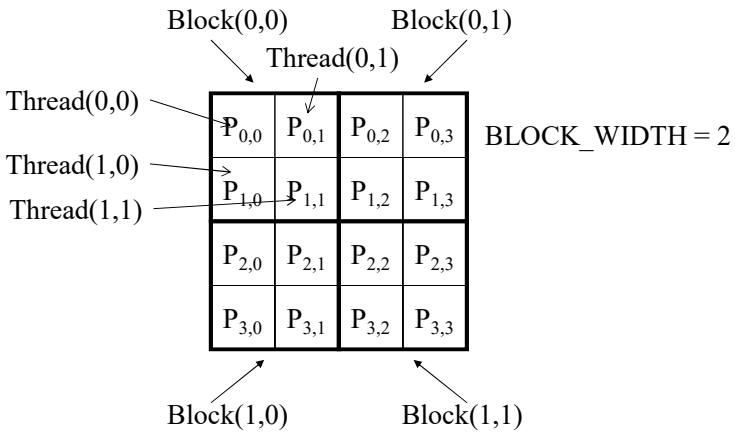
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

6

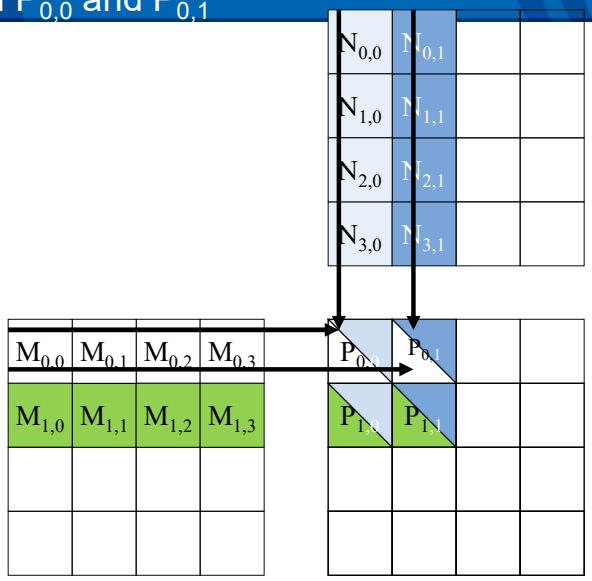
Example – Matrix Multiplication

```
__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;
  }

}
```

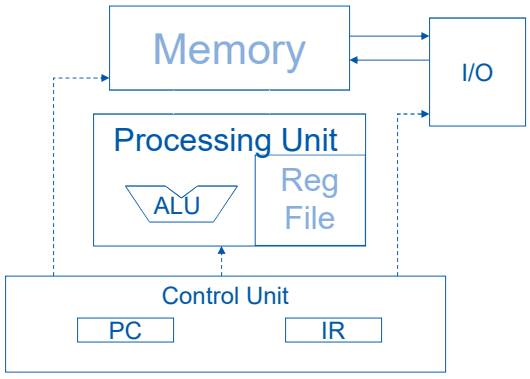

A Toy Example: Thread to P Data Mapping



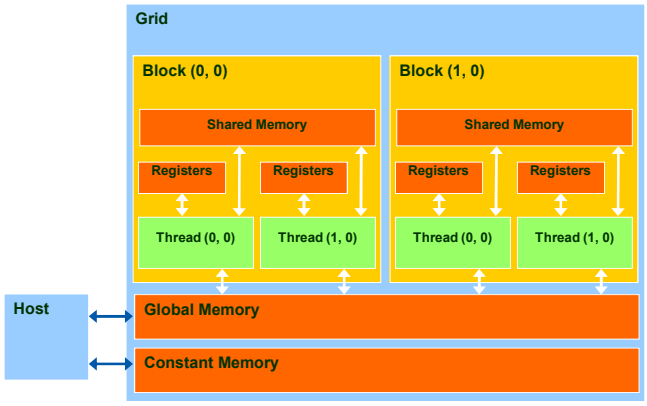
Calculation of $P_{0,0}$ and $P_{0,1}$



Memory and Registers in the Von-Neumann Model



Programmer View of CUDA Memories



Declaring CUDA Variables

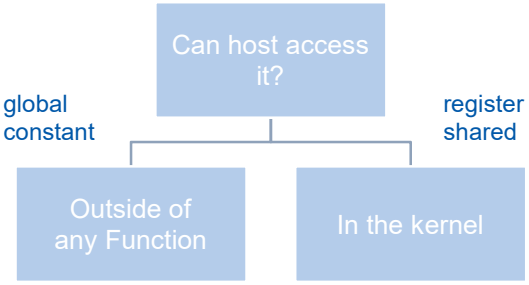
Variable declaration	Memory	Scope	Lifetime
int LocalVar;	register	thread	thread
__device__ __shared__ int SharedVar;	shared	block	block
__device__ int GlobalVar;	global	grid	application
__device__ __constant__ int ConstantVar;	constant	grid	application

- `__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];

  …
}
```

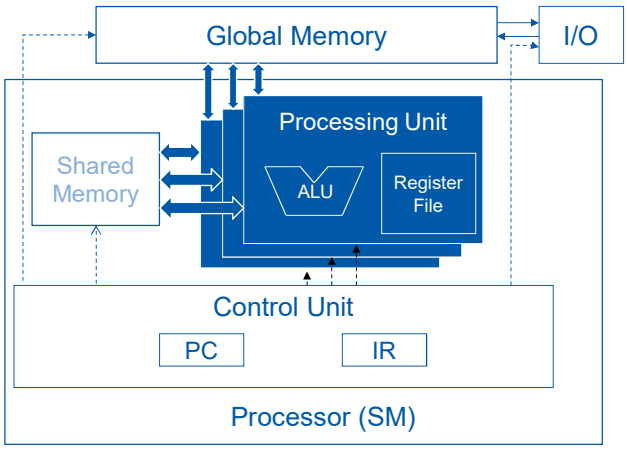

Where to Declare Variables?

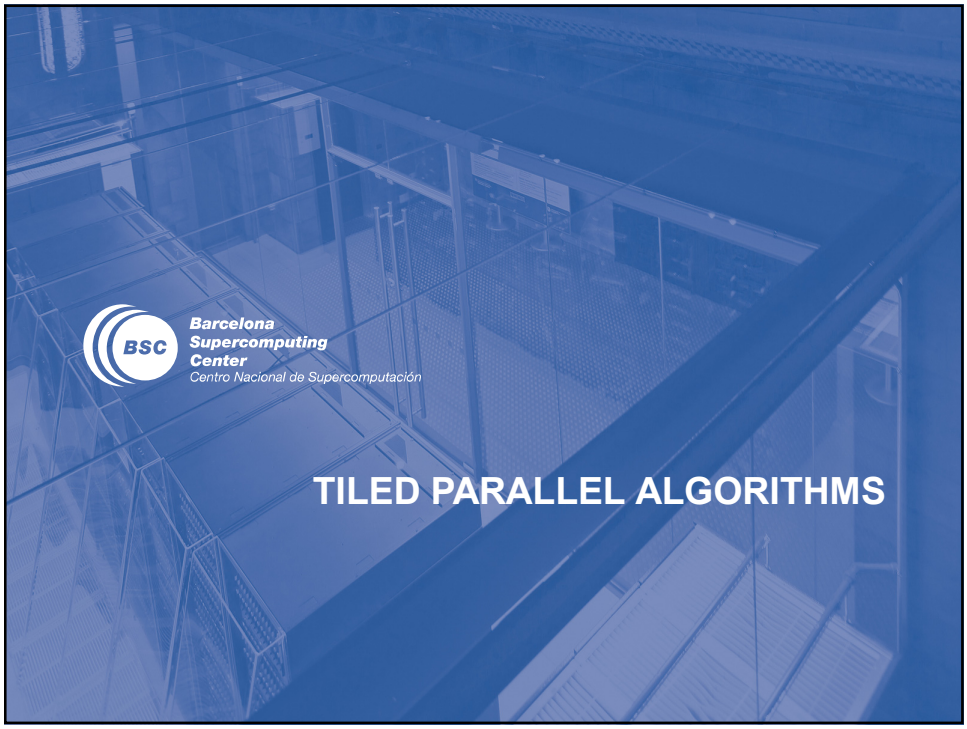


Shared Memory in CUDA

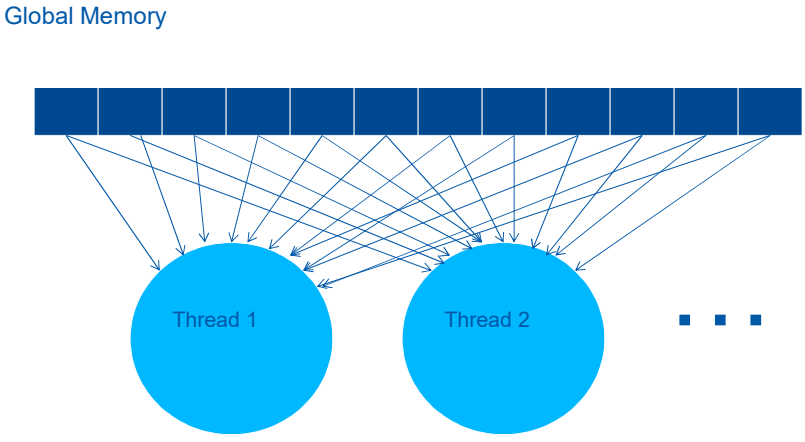
- A special type of memory whose contents are explicitly defined and used in the kernel source code
  - One in each SM
  - Accessed at much higher speed (in both latency and throughput) than global memory
  - Scope of access and sharing - thread blocks
  - Lifetime – thread block, contents will disappear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

Hardware View of CUDA Memories



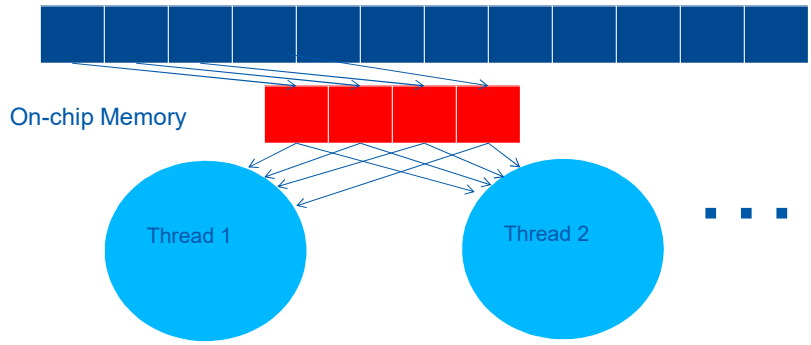


Global Memory Access Pattern of the Basic Matrix Multiplication Kernel



Tiling/Blocking - Basic Idea

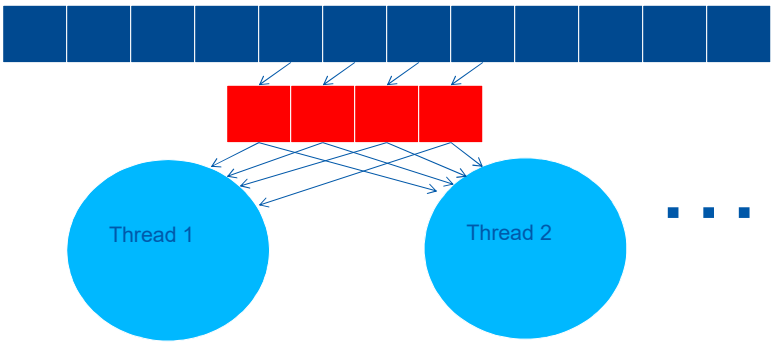
Global Memory



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

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

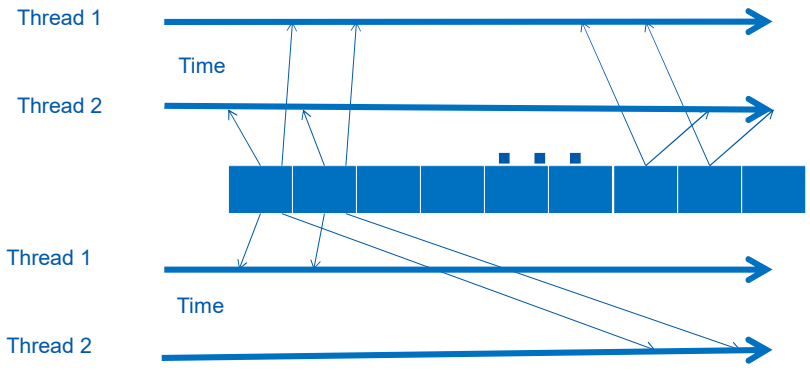
Carpools Need Synchronization

– Bad: when people have very different schedule

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

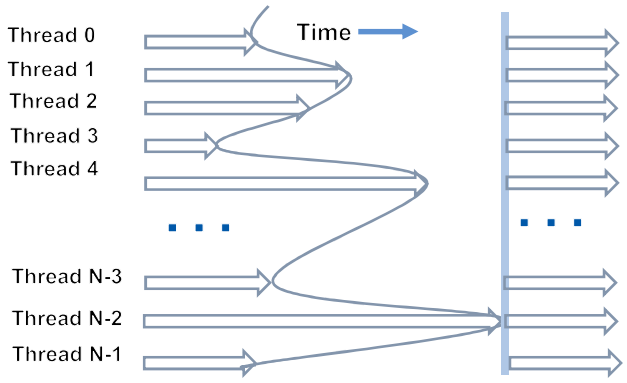
Same with Tiling

- Good: when threads have similar access timing



- Bad: when threads have very different timing

Barrier Synchronization for Tiling

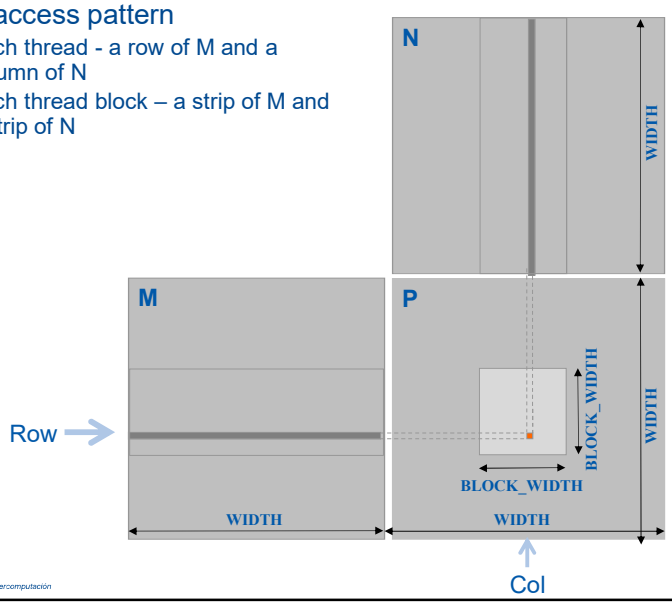


Outline of Tiling Technique

- 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

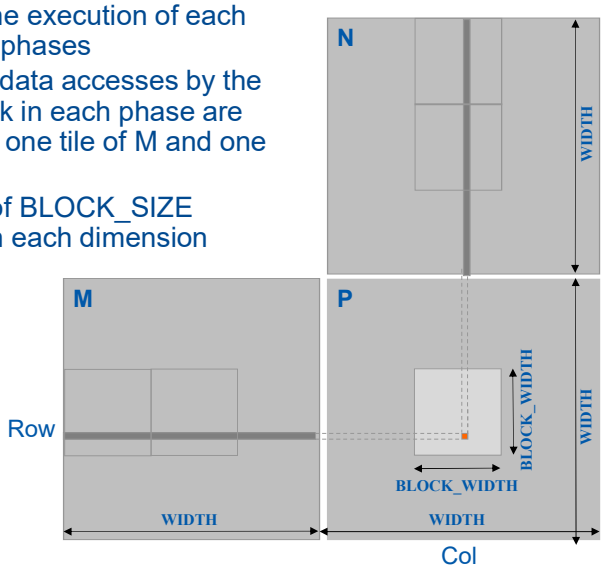
Matrix Multiplication

- 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

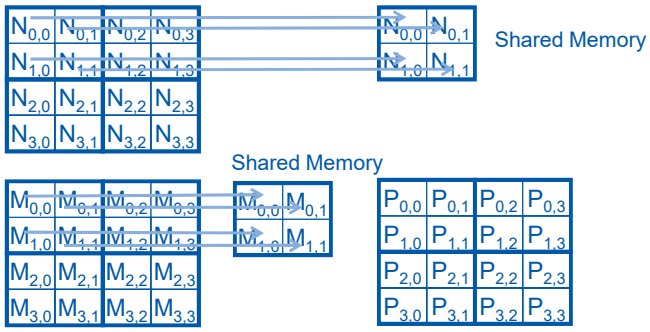
- Break up the execution of each thread into phases
- so that the data accesses by the thread block in each phase are focused on one tile of M and one tile of N
- The tile is of BLOCK_SIZE elements in each dimension



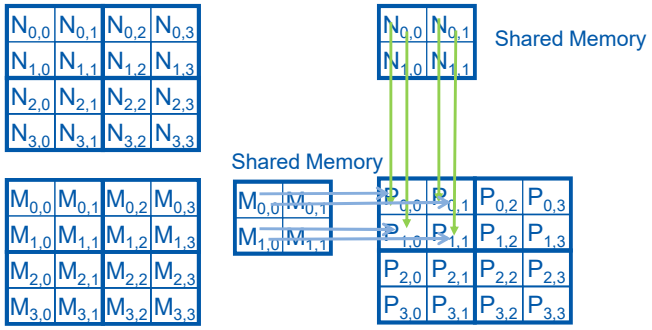
Loading a Tile

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

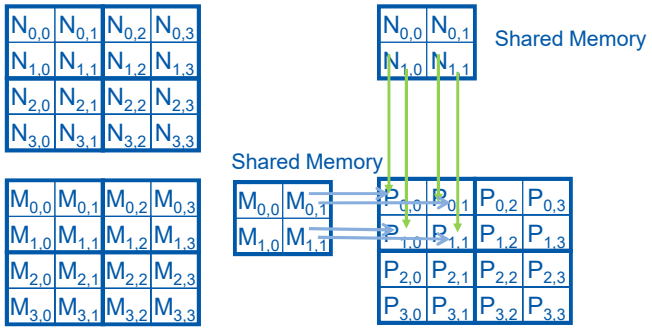
Phase 0 Load for Block (0,0)



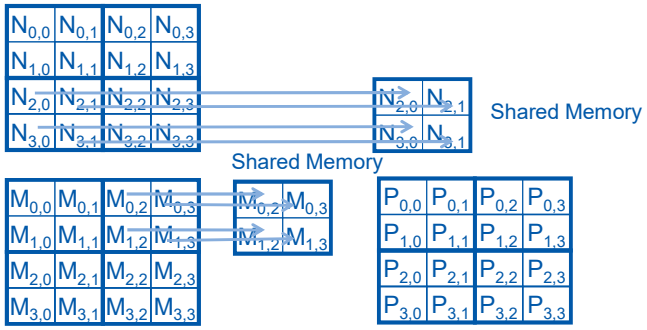
Phase 0 Use for Block (0,0) (iteration 0)



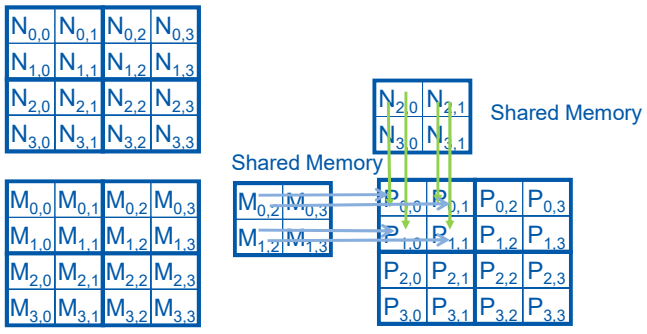
Phase 0 Use for Block (0,0) (iteration 1)



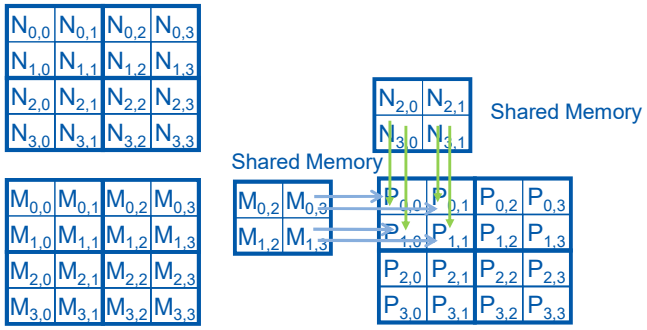
Phase 1 Load for Block (0,0)



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

time →

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

Loading Input Tile 0 of M (Phase 0)

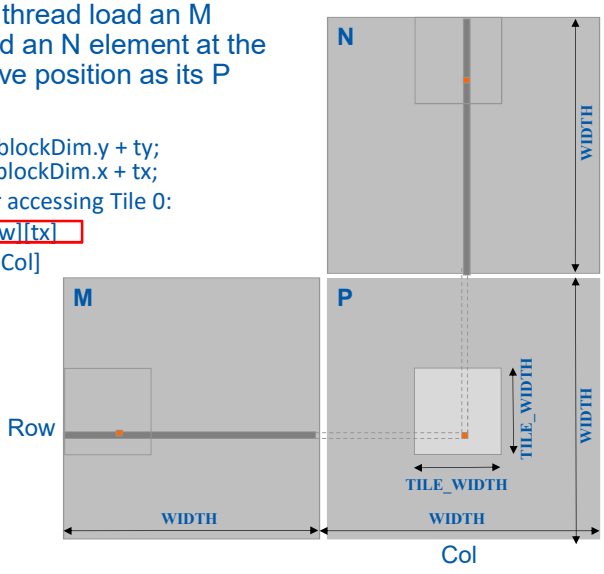
- Have each thread load an M element and an N element at the same relative position as its P element.

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

2D indexing for accessing Tile 0:

M[Row][tx]

N[ty][Col]



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

42

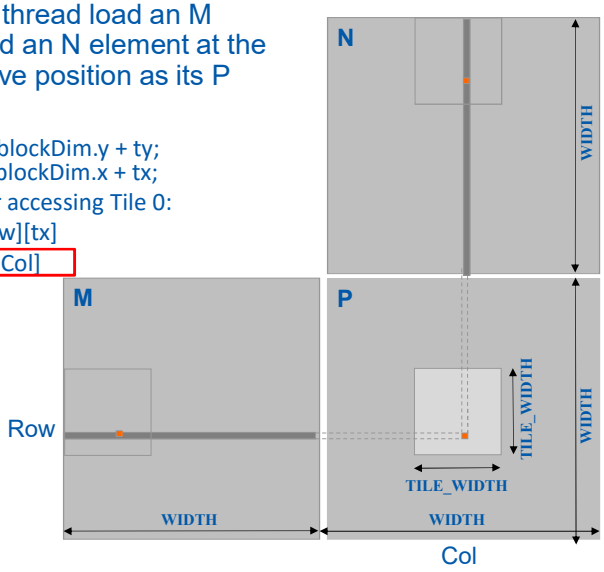
Loading Input Tile 0 of N (Phase 0)

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

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

2D indexing for accessing Tile 0:

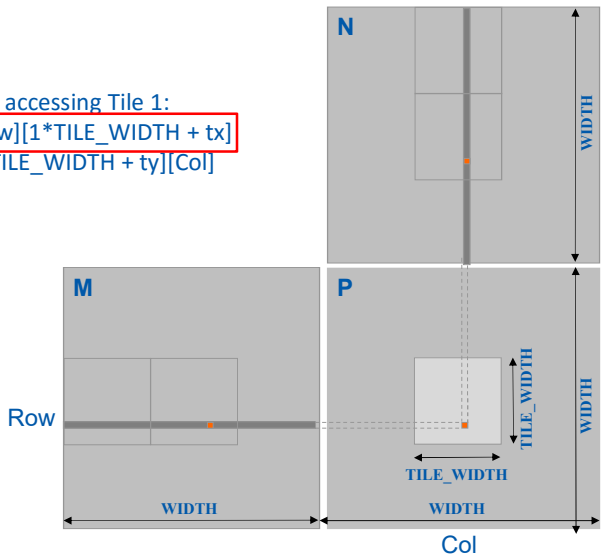
```
M[Row][tx]
N[ty][Col]
```



Loading Input Tile 1 of M (Phase 1)

2D indexing for accessing Tile 1:

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

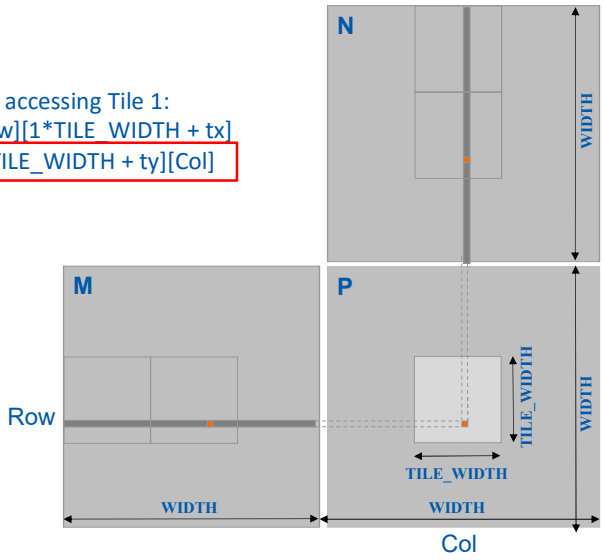


Loading Input Tile 1 of N (Phase 1)

2D indexing for accessing Tile 1:

M[Row][1*TILE_WIDTH + tx]

N[1*TILE_WIDTH + ty][Col]



M and N are dynamically allocated - use 1D indexing

M[Row][p*TILE_WIDTH+tx]

⇨ M[Row*Width + p*TILE_WIDTH + tx]

N[p*TILE_WIDTH+ty][Col]

⇨ N[(p*TILE_WIDTH+ty)*Width + Col]

where p is the sequence number of the current phase

Tiled Matrix Multiplication Kernel

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

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

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

 // Loop over the M and N tiles required to compute the P element
 for (int p = 0; p < Width/TILE_WIDTH; ++p) {  // Phases
    // 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[(t*TILE_WIDTH+ty)*Width + Col];
    __syncthreads();

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


Tiled Matrix Multiplication Kernel

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

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

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

 // Loop over the M and N tiles required to compute the P element
 for (int p = 0; p < Width/TILE_WIDTH; ++p) {  // Phases
    // 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[(t*TILE_WIDTH+ty)*Width + Col];
    __syncthreads();

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

Tiled Matrix Multiplication Kernel

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

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

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

 // Loop over the M and N tiles required to compute the P element
 for (int p = 0; p < Width/TILE_WIDTH; ++p) {  // Phases
    // 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[(t*TILE_WIDTH+ty)*Width + Col];
    __syncthreads();

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


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)
- For 32, in each phase, each block performs 2*1,024 = 2,048 float loads from global memory for 1,024 * (2*32) = 65,536 mul/add operations. (32 floating-point operation for each memory load)

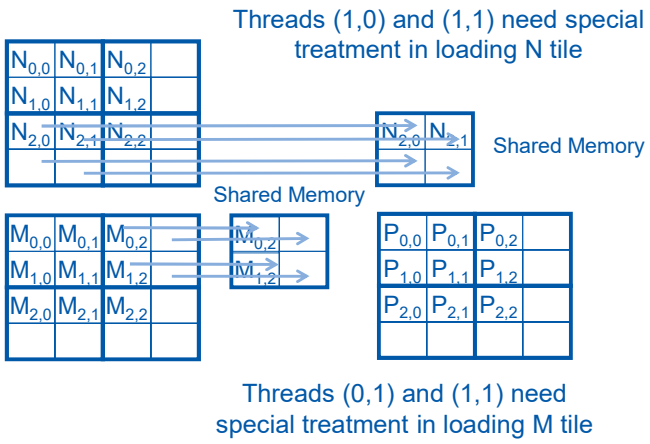
Shared Memory and Threading

- For an SM with 16KB shared memory
  - Shared memory size is implementation dependent!
  - For TILE_WIDTH = 16, each thread block uses 2*256*4B = 2KB of shared memory.
  - For 16KB shared memory, one can potentially have up to 8 thread blocks executing
    - This allows up to 8*512 = 4,096 pending loads. (2 per thread, 256 threads per block)
  - The next TILE_WIDTH 32 would lead to 2*32*32*4B = 8KB of 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

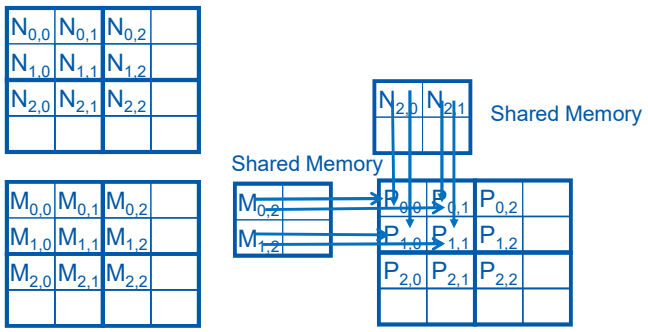
Handling Matrix of Arbitrary Size

- 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.

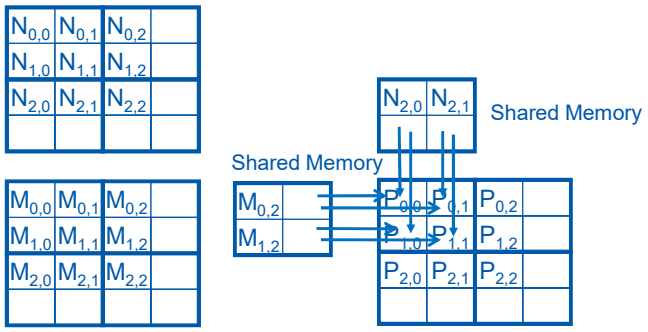
Phase 1 Loads for Block (0,0) for a 3x3 Example



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

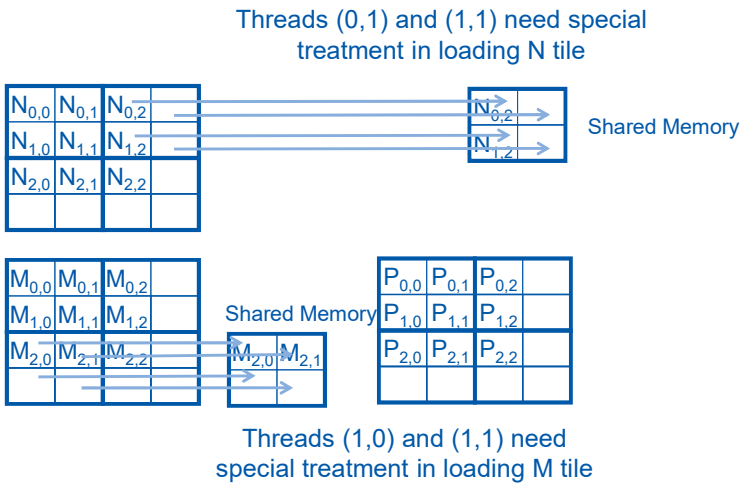


Phase 1 Use for Block (0,0) (iteration 1)



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

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



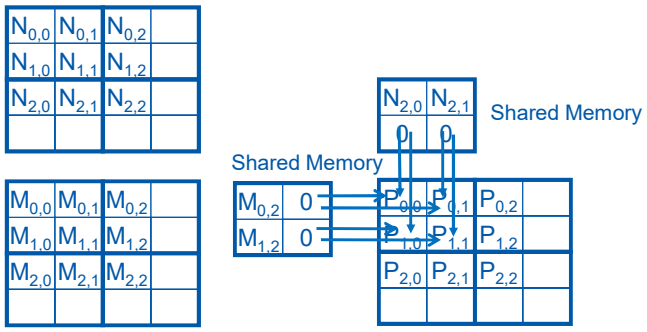
Major Cases in Toy Example

- 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

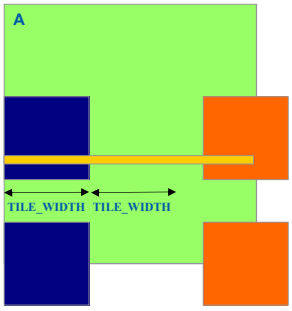
- When a thread is to load any input element, test if it is in the valid index range
  - If valid, proceed to load
  - Else, do not load, just write a 0
- Rationale: a 0 value will ensure that that the multiply-add step does not affect the final value of the output element
- The condition tested for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



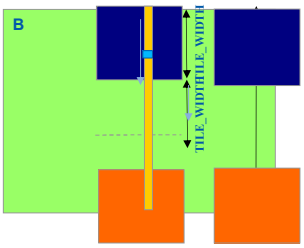
Boundary Condition for Input M Tile

- Each thread loads
  - M[Row][p*TILE_WIDTH+tx]
  - M[Row*Width + p*TILE_WIDTH+tx]
- Need to test
  - (Row < Width) && (p*TILE_WIDTH+tx < Width)
  - If true, load M element
  - Else , load 0



Boundary Condition for Input N Tile

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



Loading Elements – with boundary check

```
–  8    for (int p = 0; p < (Width-1) / TILE_WIDTH + 1; ++p) {
–
–  ++      if(Row < Width && 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 x k M matrix multiplied with a k x l N matrix results in a j x 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

Question 1

- Assume that a kernel is launched with 1,000 thread blocks each of which has 512 threads. If a variable is declared as a shared memory variable, how many versions of the variable will be created through the lifetime of the execution of the kernel?

a) 1
b) 1,000
c) 512
d) 512,000

Question 1 - Answer

- Assume that a kernel is launched with 1,000 thread blocks each of which has 512 threads. If a variable is declared as a shared memory variable, how many versions of the variable will be created through the lifetime of the execution of the kernel?

a) 1
b) 1,000
c) 512
d) 512,000

Explanation: Shared memory variables are allocated to thread blocks. So, the # of versions is the number of th. blocks: 1,000.

Question 2

- For our tiled matrix-matrix multiplication kernel, if we use a 32x32 tile, what is the reduction of memory bandwidth usage for input matrices M and N?

a) 1/8 of the original usage
b) 1/16 of the original usage
c) 1/32 of the original usage
d) 1/64 of the original usage

Question 2 - Answer

- For our tiled matrix-matrix multiplication kernel, if we use a 32x32 tile, what is the reduction of memory bandwidth usage for input matrices M and N?

a) 1/8 of the original usage
b) 1/16 of the original usage
c) 1/32 of the original usage
d) 1/64 of the original usage

Explanation: Each element in the tile is used 32 times

Question 3

- For the tiled single-precision matrix multiplication kernel, assume that the tile size is 32x32 and the system has a DRAM burst size of 128 bytes. How many DRAM bursts will be delivered to the processor as a result of loading one M-matrix tile by a thread block?

a) 16
b) 32
c) 64
d) 128

Question 3 - Answer

- For the tiled single-precision matrix multiplication kernel, assume that the tile size is 32x32 and the system has a DRAM burst size of 128 bytes. How many DRAM bursts will be delivered to the processor as a result of loading one M-matrix tile by a thread block?

a) 16
b) 32
c) 64
d) 128

Explanation: For a 32x32 M-tile, each row in the tile consists of 32 consecutive words (128 bytes). The total amount of data in the row is just a single burst. We have 32 rows in a tile so there will be 32 bursts delivered to the processor.

Question 4

- Assume a tiled matrix multiplication that handles boundary conditions. Assume that we use 32x32 tiles to process square matrices of 1,000x1,000. Within EACH thread block, what is the maximum number of warps that will have control divergence due to handling boundary conditions for loading M tiles throughout the kernel execution?

a) 32
b) 24
c) 16
d) 8

Question 4 - Answer

Assume a tiled matrix multiplication that handles boundary conditions. Assume that we use 32x32 tiles to process square matrices of 1,000x1,000. Within EACH thread block, what is the maximum number of warps that will have control divergence due to handling boundary conditions for loading M tiles throughout the kernel execution?

a) **32**
b) 24
c) 16
d) 8

Explanation: Control divergence happens due to the handling of the right edge. For thread blocks processing tiles that are totally within the valid range in the y-dimension, all 32 warps in a block will experience divergence at the right boundary. For the thread blocks that process the bottom M tiles on the right edge, only 8 warps will experience control divergence because all threads in the lower 24 warps will fail the boundary test.

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Thank you!

For further information please contact
antonio.pena@bsc.es