



ECE408/CS483/CSE408

Applied Parallel Programming

Lecture 4: CUDA Memory Model

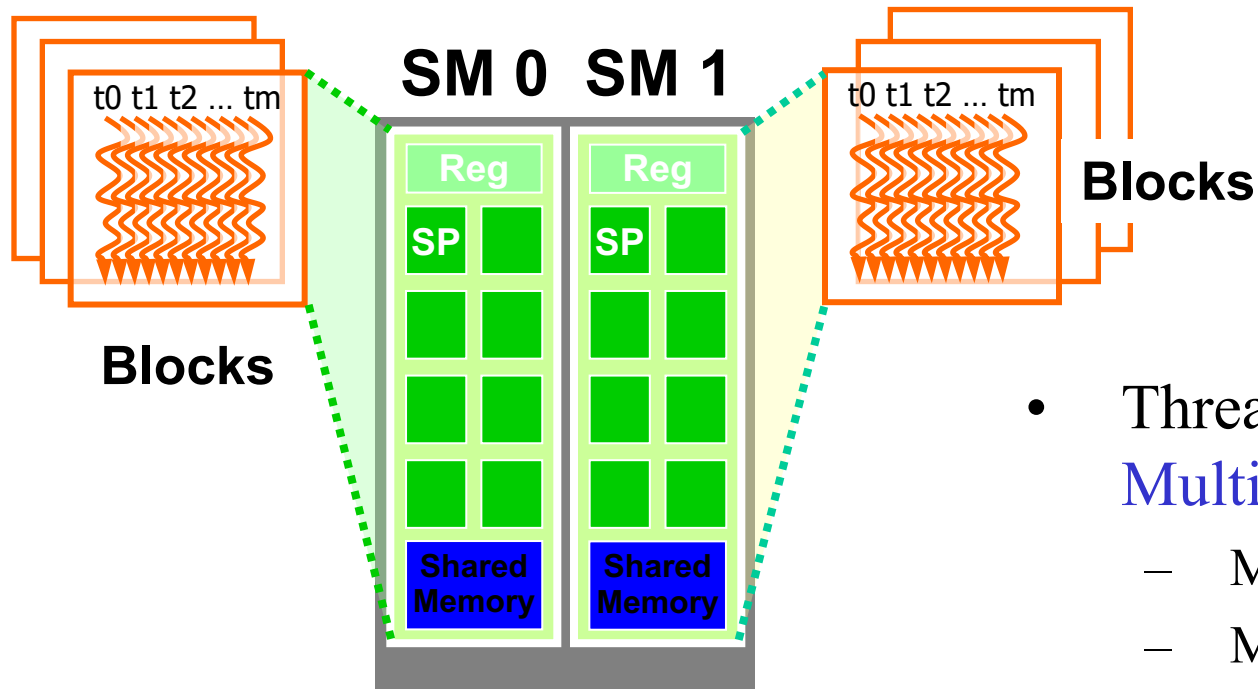
Course Reminders

- Lab 0 is due Friday at 8pm CT
- Lab 1 will be released this Friday
 - **Unlike Lab 0, this lab counts and must be submitted on time!**
- Email Andy Schuh (aschuh@illinois.edu) to get GitHub and rai access if you just signed up for the course
- Make sure you have Canvas access

Objective

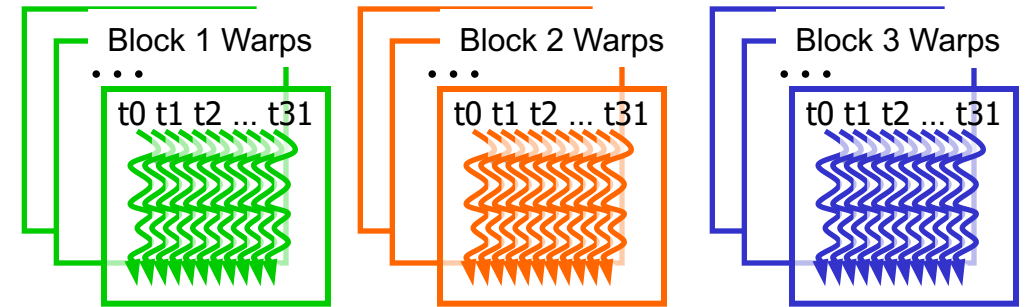
- To learn the basic features of the memories accessible by CUDA threads
- To prepare for Lab 2 - basic matrix multiplication
- To learn to evaluate the performance implications of global memory accesses

Executing Thread Blocks

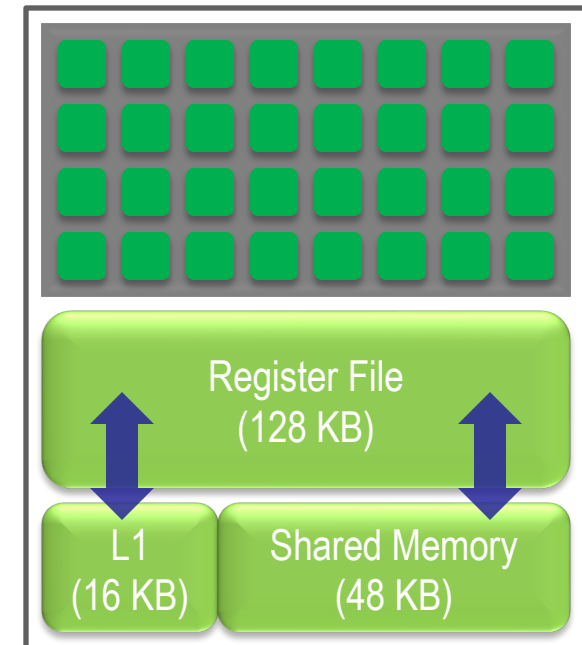


- Threads are assigned to **Streaming Multiprocessors (SM)** in block granularity
 - Max number of blocks per SM
 - Max number of threads per SM
 - Max number of threads per block
 - All are dependent on generation of GPU
- Threads run concurrently, as warps
 - SM maintains thread/block id #s
 - SM manages/schedules thread execution

Thread Scheduling (1/2)

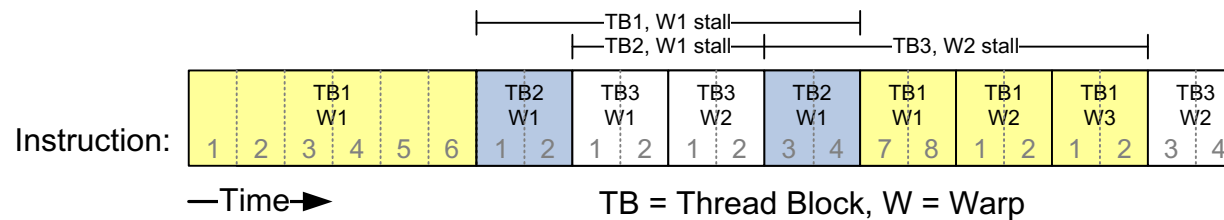


- Each block is executed as 32-thread warps
 - An implementation decision, not part of the CUDA programming model
 - Warps are divided based on their linearized thread index
 - Threads 0-31: warp 0
 - Threads 32-63: warp 1, etc.
 - X-dimension first, then Y, then Z
 - Warps are scheduling units in SM
- If 3 blocks are assigned to an SM and each block has 256 threads, how many warps are there in an SM?
 - Each block is divided into $256/32 = 8$ warps
 - $8 \text{ warps/blk} * 3 \text{ blks} = 24 \text{ warps}$



Thread Scheduling (2/2)

- SM implements zero-overhead warp scheduling
 - Warps whose next instruction has its operands ready for consumption are eligible for execution
 - Eligible warps are selected for execution on a prioritized scheduling policy
 - **All threads in a warp execute the same instruction when selected**



Example execution timing of an SM

Control (branch) Divergence

- Main performance concern with branching is divergence
 - Threads within a single warp take different paths
 - Different execution paths are serialized in current GPUs
- A common case: divergence when a branch condition is a function of thread ID
 - `if (threadIdx.x % 2) { }`
 - This creates two different control paths for threads in a warp
 - Has divergence (50% of threads do nothing)
 - `if ((threadIdx.x / WARP_SIZE) % 2) { }`
 - Also creates two different control paths, but...
 - Branch granularity is a whole multiple of warp size;
 - All threads in any given warp follow the same path
 - No divergence

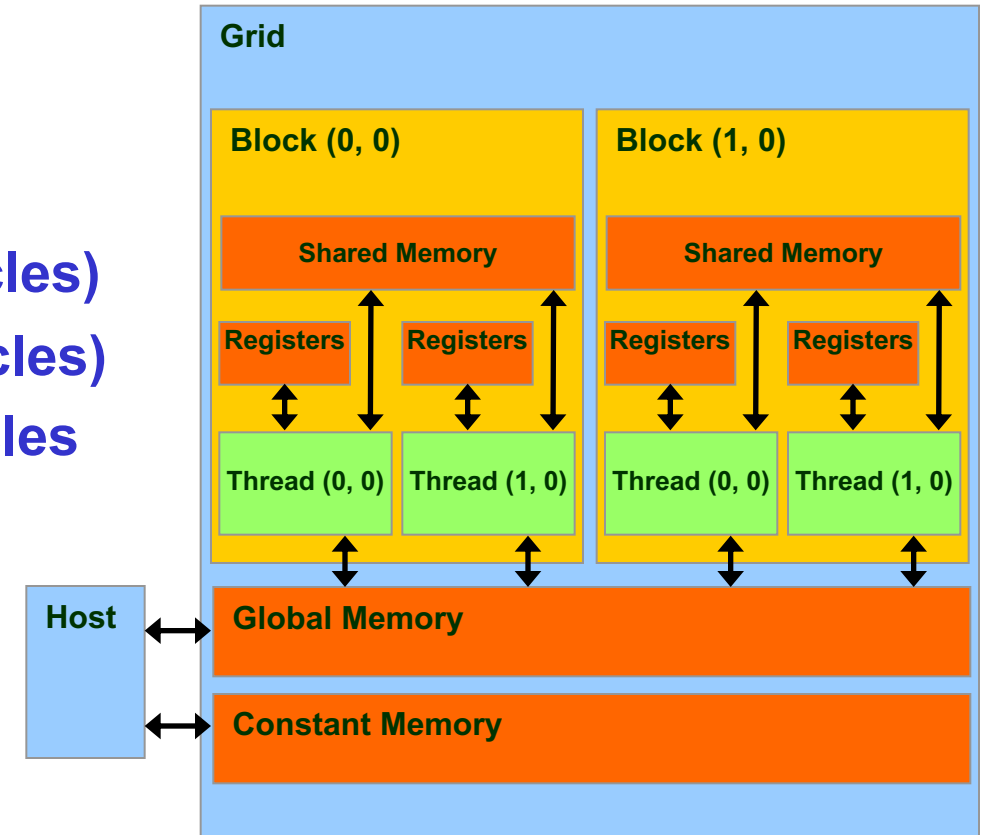
Block Granularity Considerations

- For RGBToGrayscale, should one use 8X8, 16X16 or 32X32 blocks? Assume that in the GPU used, each SM can take up to 1536 threads and up to 8 blocks.
 - For 8X8, we have 64 threads per block. Each SM can take up to 1536 threads, which is 24 blocks. But each SM can only take up to 8 Blocks, only 512 threads (16 warps) will go into each SM!
 - For 16X16, we have 256 threads per block. Since each SM can take up to 1536 threads (48 warps), which is 6 blocks (within the 8 block limit). Thus we use the full thread capacity of an SM.
 - For 32X32, we would have 1024 threads per Block. Only one block can fit into an SM, using only 2/3 of the thread capacity of an SM.

Programmer View of CUDA Memories

Each thread 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**)



CUDA Variable Type Qualifiers

Variable declaration	Memory	Scope	Lifetime
<code>int LocalVar;</code>	register	thread	thread
<code>__device__ __shared__ int SharedVar;</code>	shared	block	block
<code>__device__ int GlobalVar;</code>	global	app.	application
<code>__device__ __constant__ int ConstantVar;</code>	constant	app.	application

- `__device__`
 - optional with `__shared__` or `__constant__`
 - not allowed by itself within functions
- Automatic variables with no qualifiers
 - in registers for primitive types and structures
 - in global memory for per-thread arrays

Next Application: Matrix Multiplication

- Given two square matrices, M and N, dimensions Width × Width
 - we can multiply M by N
 - to compute a third Width × Width matrix, P:
 - $P = MN$

In terms of the elements of P, matrix multiplication implies computing...

$$P_{ij} = \sum_{k=1}^{Width} M_{ik} N_{kj}$$

Matrix Multiplication-- Simple CPU Version

```
// Matrix multiplication on the (CPU) host in single precision
```

```
void MatrixMul(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];
```

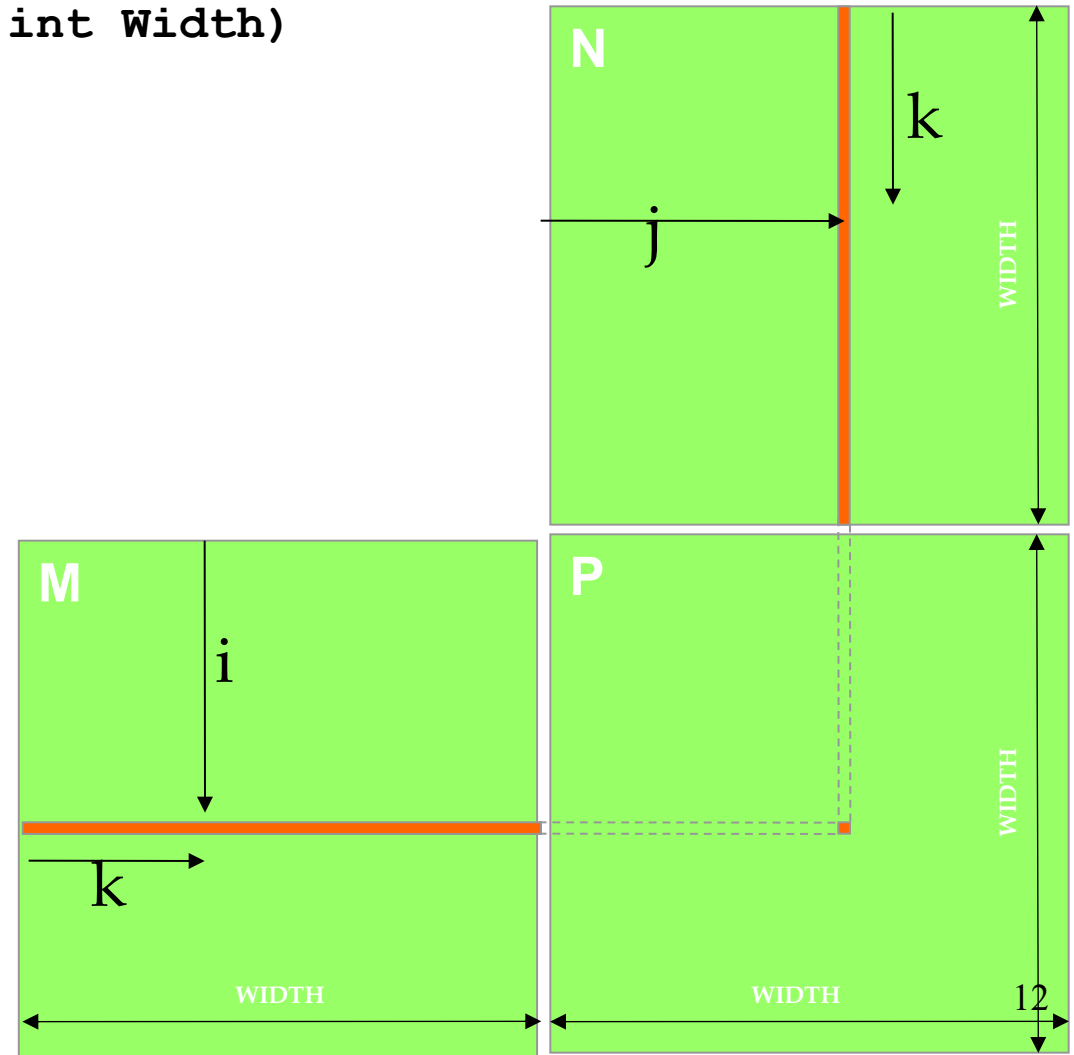
```
                sum += a * b;
```

```
            }
```

```
            P[i * Width + j] = sum;
```

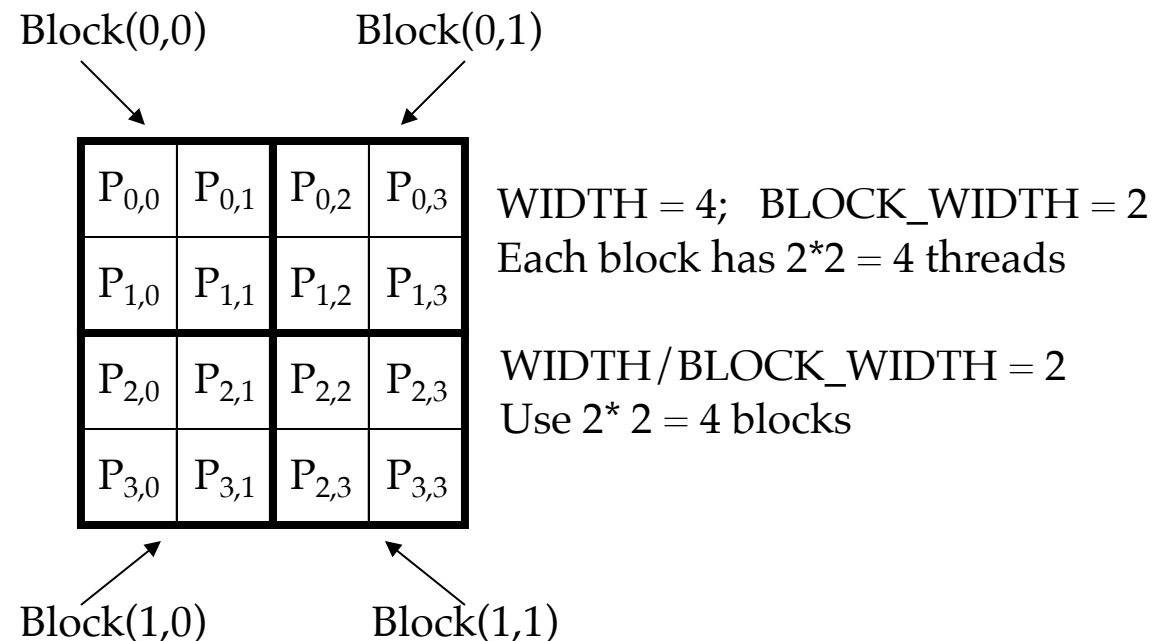
```
        }
```

```
    }
```



Kernel Function - A Small Example

- Have each 2D thread block to compute a $(\text{BLOCK_WIDTH})^2$ sub-matrix of the result matrix
 - Each block has $(\text{BLOCK_WIDTH})^2$ threads
- Generate a 2D Grid of $(\text{WIDTH}/\text{BLOCK_WIDTH})^2$ blocks
- This concept is called **tiling**. Each block represents a **tile**.



A Slightly Bigger Example

(BLOCK_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; BLOCK_WIDTH = 2
Each block has $2 \times 2 = 4$ threads

WIDTH/BLOCK_WIDTH = 4
Use $4 \times 4 = 16$ blocks

A Slightly Bigger Example (cont.)

(BLOCK_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; BLOCK_WIDTH = 4
Each block has 4*4 = 16 threads

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

Kernel Invocation (Host-side Code)

```
// Setup the execution configuration
// BLOCK_WIDTH is a #define constant
dim3 dimGrid(ceil((1.0*Width)/BLOCK_WIDTH),
             ceil((1.0*Width)/BLOCK_WIDTH), 1);

dim3 dimBlock(BLOCK_WIDTH, BLOCK_WIDTH, 1);

// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

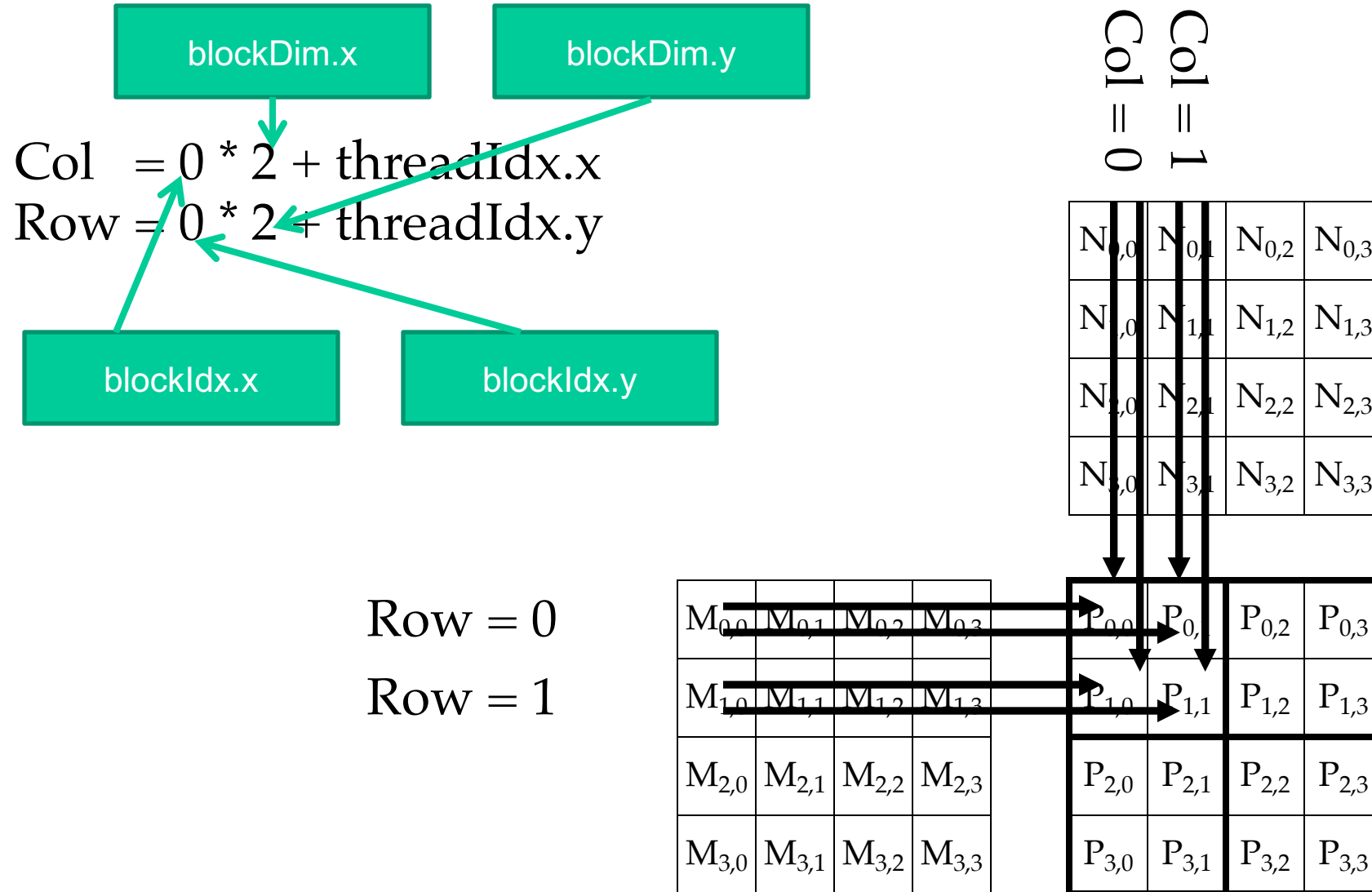

Kernel Function

```
// Matrix multiplication kernel - per thread code

__global__
void MatrixMulKernel(float *d_M, float *d_N, float *d_P, int Width)
{

    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;
```

Work for Block (0,0) for TILE_WIDTH = 2



Work for Block (0,1)

$$\text{Col} = 1 * 2 + \text{threadIdx.x}$$

$$\text{Row} = 0 * 2 + \text{threadIdx.y}$$

blockIdx.x

blockIdx.y

Col = 2

Col = 3

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}

Row = 0

Row = 1

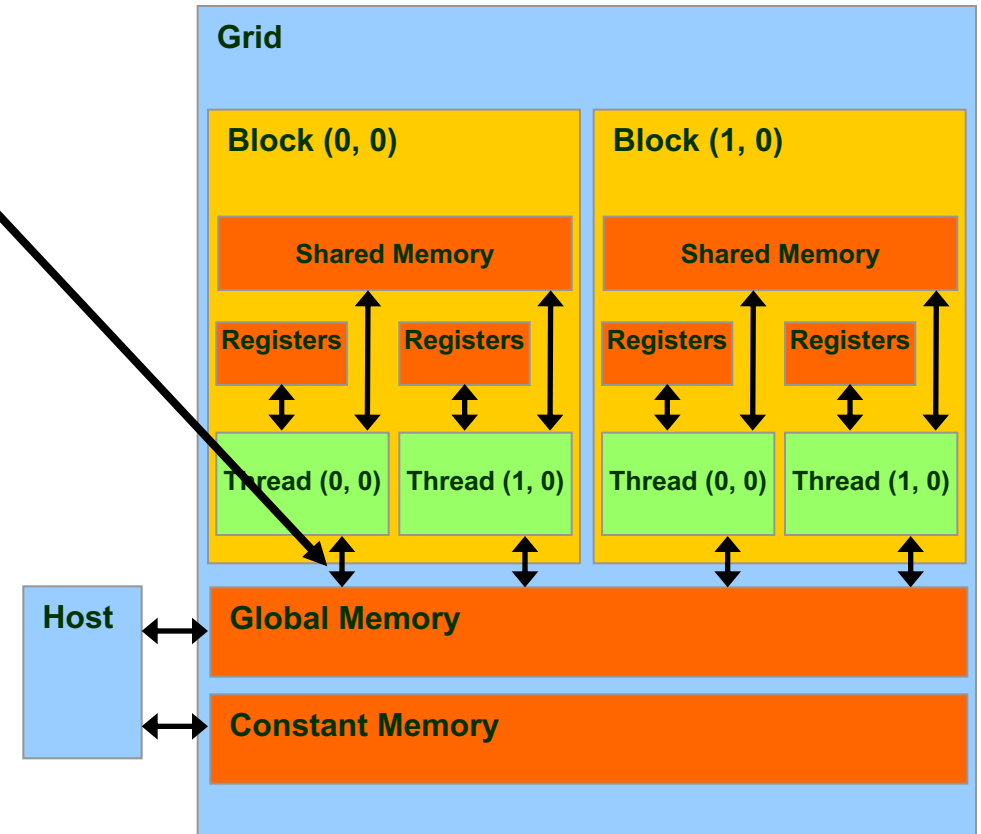
M _{0,0}	M _{0,1}	M _{0,2}	M _{0,3}	P _{0,0}	P _{0,1}	P _{0,2}	P _{0,3}
M _{1,0}	M _{1,1}	M _{1,2}	M _{1,3}	P _{1,0}	P _{1,1}	P _{1,2}	P _{1,3}
M _{2,0}	M _{2,1}	M _{2,2}	M _{2,3}	P _{2,0}	P _{2,1}	P _{2,2}	P _{2,3}
M _{3,0}	M _{3,1}	M _{3,2}	M _{3,3}	P _{3,0}	P _{3,1}	P _{3,2}	P _{3,3}

A Simple Matrix Multiplication Kernel

```
__global__  
void MatrixMulKernel(float *d_M, float *d_N, float *d_P, int Width)  
{  
    // Calculate the column index of d_P and d_N  
    int Col = blockIdx.x * blockDim.x + threadIdx.x;  
  
    // Calculate the row index of d_P and d_M  
    int Row = blockIdx.y * blockDim.y + threadIdx.y;  
  
    if ((Row < Width) && (Col < Width)) {  
        float Pvalue = 0;  
        // each thread computes one element of d_P  
        for (int k = 0; k < Width; ++k)  
            Pvalue += d_M[Row*Width+k] * d_N[k*Width+Col];  
        d_P[Row*Width+Col] = Pvalue;  
    }  
}
```

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 (2 fp ops)
 - 4B of memory for each FLOP
 - 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



Two vertical lines, one blue and one orange, are positioned on the left side of the slide.

**ANY MORE QUESTIONS?
READ CHAPTER 4!**