Thread mapping in 2D Grids & 2D Blocks

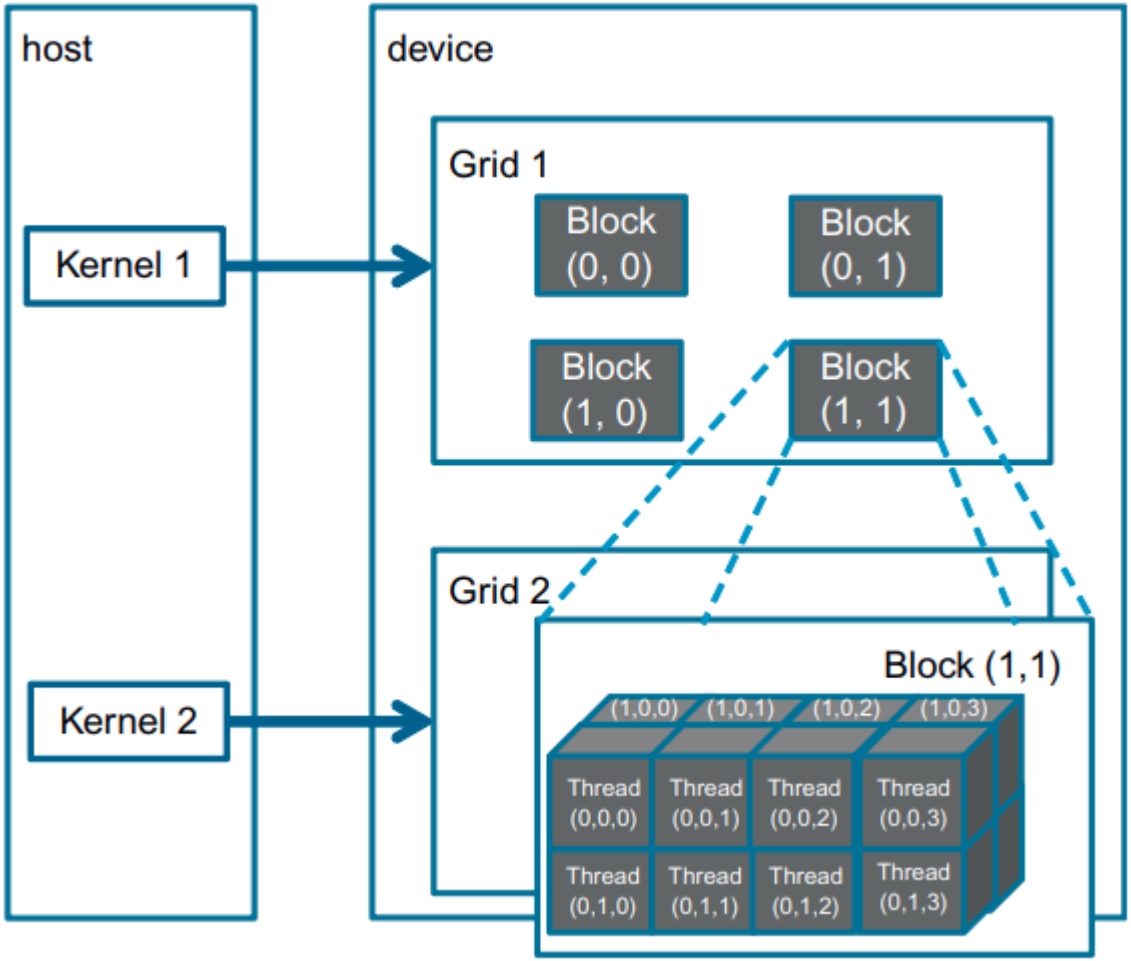
K Hari Babu
Department of Computer Science & Information Systems

BITS Pilani
Pilani Campus

CUDA Thread Organization

BITS Pilani

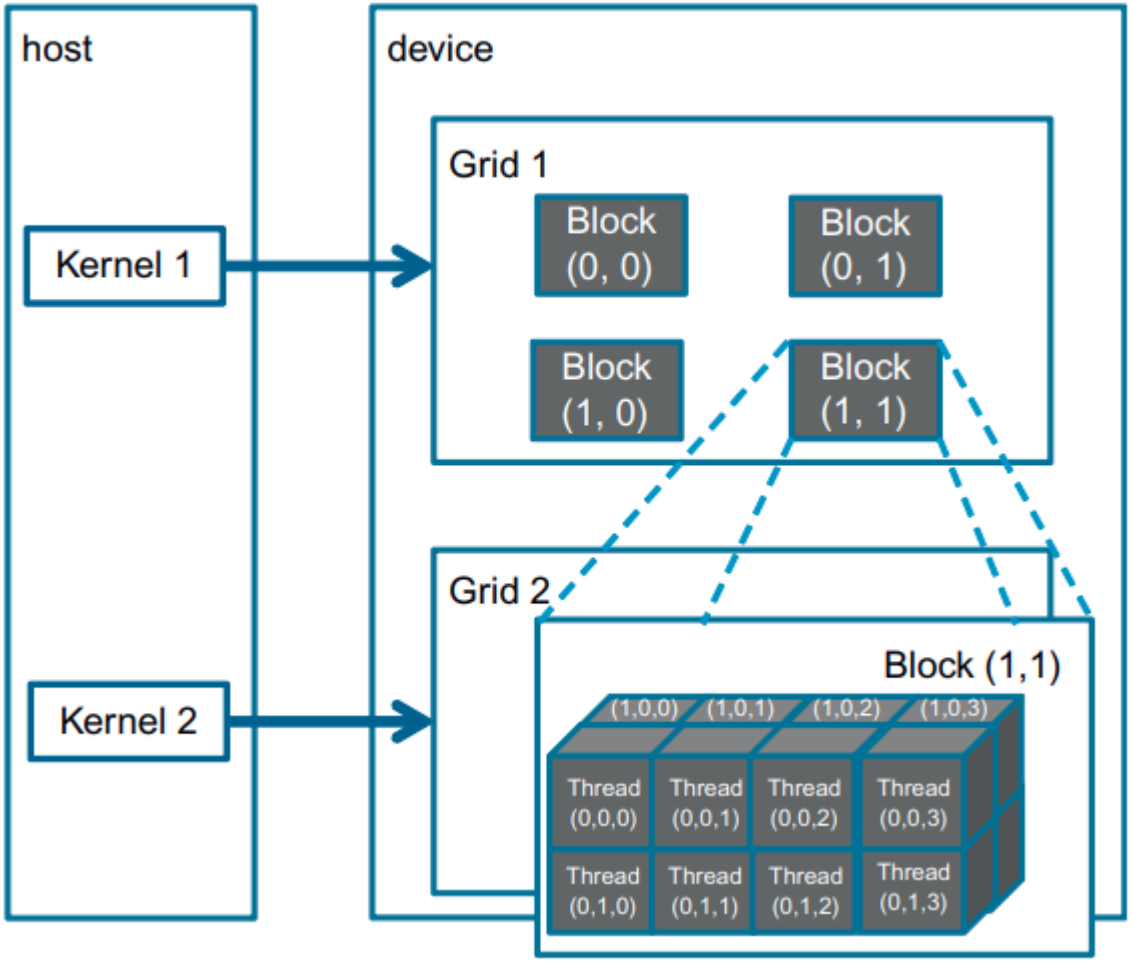
- A grid is a 3D array of blocks and each block is a 3D array of threads
- Threads are organized into a two-level hierarchy: a grid consists of one or more blocks and each block in turn consists of one or more threads



CUDA Thread Organization

BITS Pilani

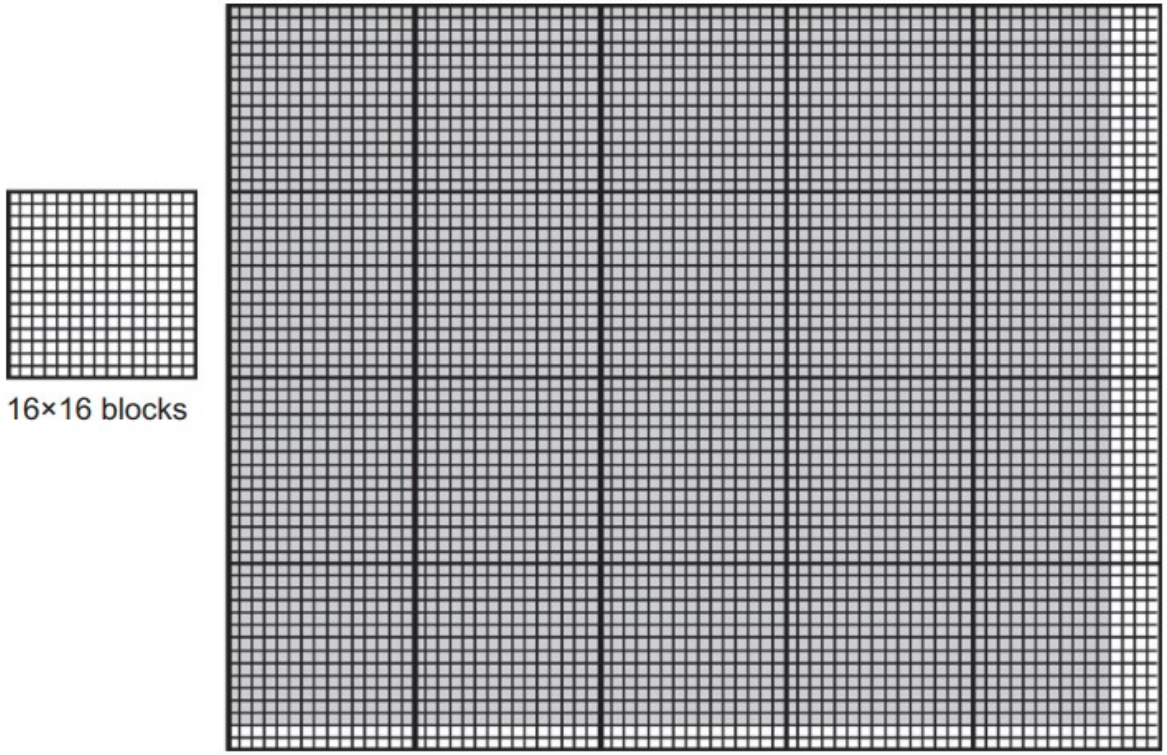
- All threads in a block share the same block index, accessed as the blockIdx
- Each thread also has a thread index, which can be accessed as the threadIdx variable in a kernel
- Dimensions are available as gridDim and blockDim in kernel functions



Mapping Threads to Multidimensional Data

BITS Pilani

- The choice of 1D, 2D, or 3D thread organizations is usually based on the nature of the data
  - For example, pictures are a 2D array of pixels. It is convenient to use a 2D grid that consists of 2D blocks to process the pixels in a picture.
  - 76 x 62 picture

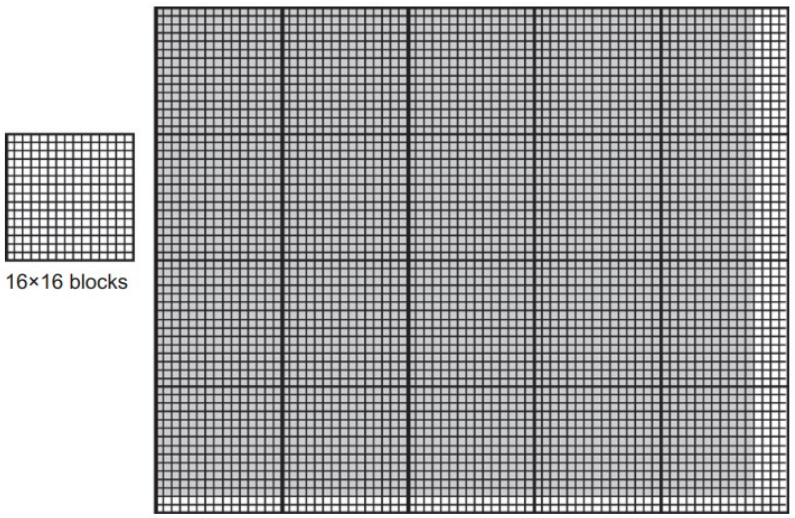


Mapping Threads to Multidimensional Data

BITS Pilani

- 76 x 62 picture
  - 76 pixels in the horizontal or x direction and 62 pixels in the vertical or y direction = 4712 pixels
- Assume that we decided to use a 16 x 16 block, with 16 threads in the x direction and 16 threads in the y direction
  - Need five blocks in the x direction and four blocks in the y direction, which results in 5 x 4 = 20 blocks

- The shaded area depicts the threads that cover pixels. Note that we have four extra threads in the x direction and two extra threads in the y direction. That is, we will generate 80 x 64 threads to process 76 x 62 pixels.
- Should have if statements to test whether the thread indices threadIdx.x and threadIdx.y fall within the valid range of pixels



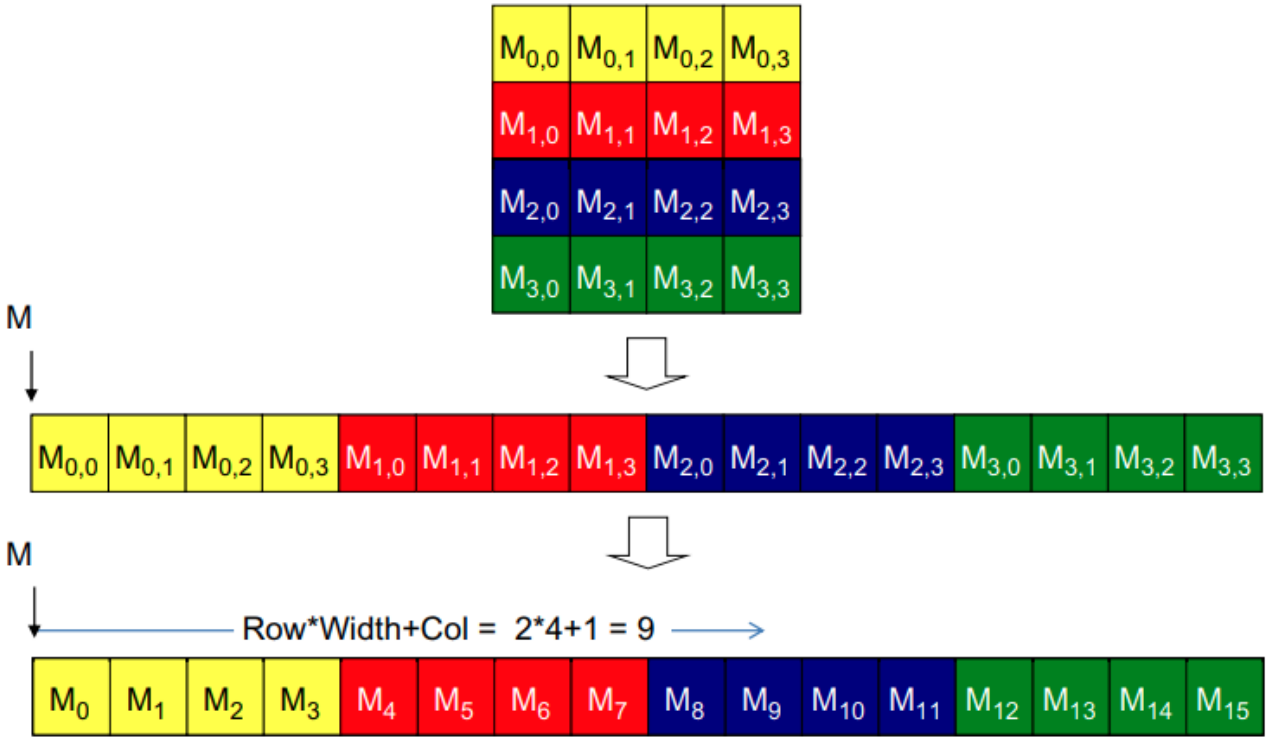
Memory Layout

- There are at least two ways one can linearize a 2D array
- One is to place all elements of the same row into consecutive locations. The rows are then placed one after another into the memory space
  - Row-major layout. Used in C/C++ language compilers
- Another way to linearize a 2D array is to place all elements of the same column into consecutive locations. The columns are then placed one after another into the memory space.
  - This arrangement, called column major layout, is used by FORTRAN compilers
- In CUDA, due to dynamic-size of arrays, compiler leaves the work of such translation to the programmers due to lack of dimensional information

Row-major Layout For a 2D C Array

BITS Pilani

- The result is an equivalent 1D array accessed by an index expression Row*Width+Col for an element that is in the Rowth row and Colth column of an array of Width elements in each row.



Picture Kernel

BITS Pilani

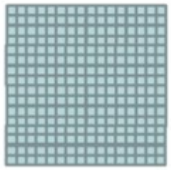
```
1   __global__ void PictureKernell(float* d_Pin, float* d_Pout, int n, int m) {
2       // Calculate the row # of the d_Pin and d_Pout element to process
3       int Row = blockIdx.y*blockDim.y + threadIdx.y;
4       // Calculate the column # of the d_Pin and d_Pout element to process
5       int Col = blockIdx.x*blockDim.x + threadIdx.x;
6       // each thread computes one element of d_Pout if in range
7       if ((Row < m) && (Col < n)) {
8           d_Pout[Row*n+Col] = 2*d_Pin[Row*n+Col];
9   }
10  }
```

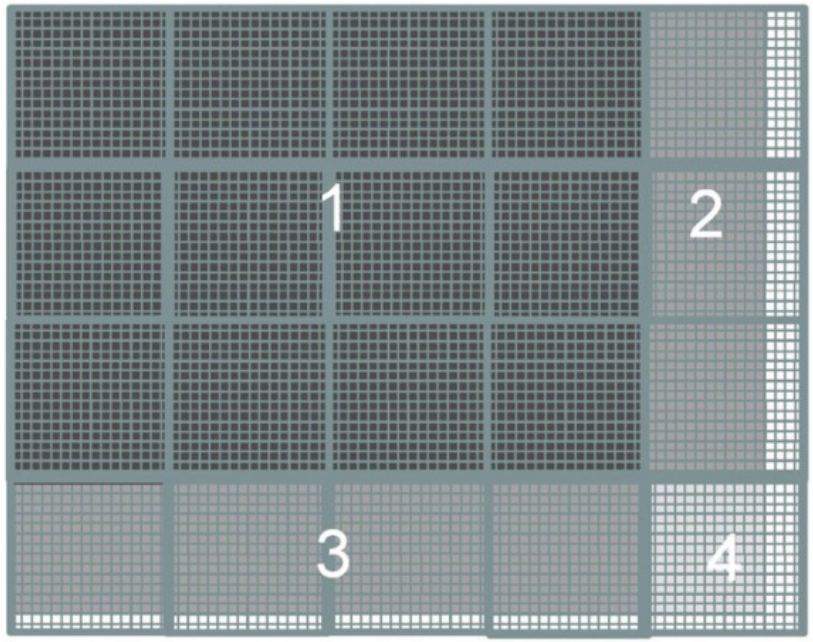
- Let's assume that the kernel will scale every pixel value in the picture by a factor of 2.0
- There are a total of blockDim.x * gridDim.x threads in the horizontal direction.
- Expression Col = blockIdx.x * blockDim.x + threadIdx.x generates every integer value from 0 to blockDim.x*gridDim.x-1

Mapping pixels

BITS Pilani

- Case1: Both Col and Row values of these threads are within range
- Case2: Although the Row values of these threads are always within range, the Col values of some of them exceed the n value
- Case3: Although the Col values of these threads are always within range, the Row values of some of them exceed the m value (62)
- Case4: Some row values and some col values will be outsode range.


16x16 block

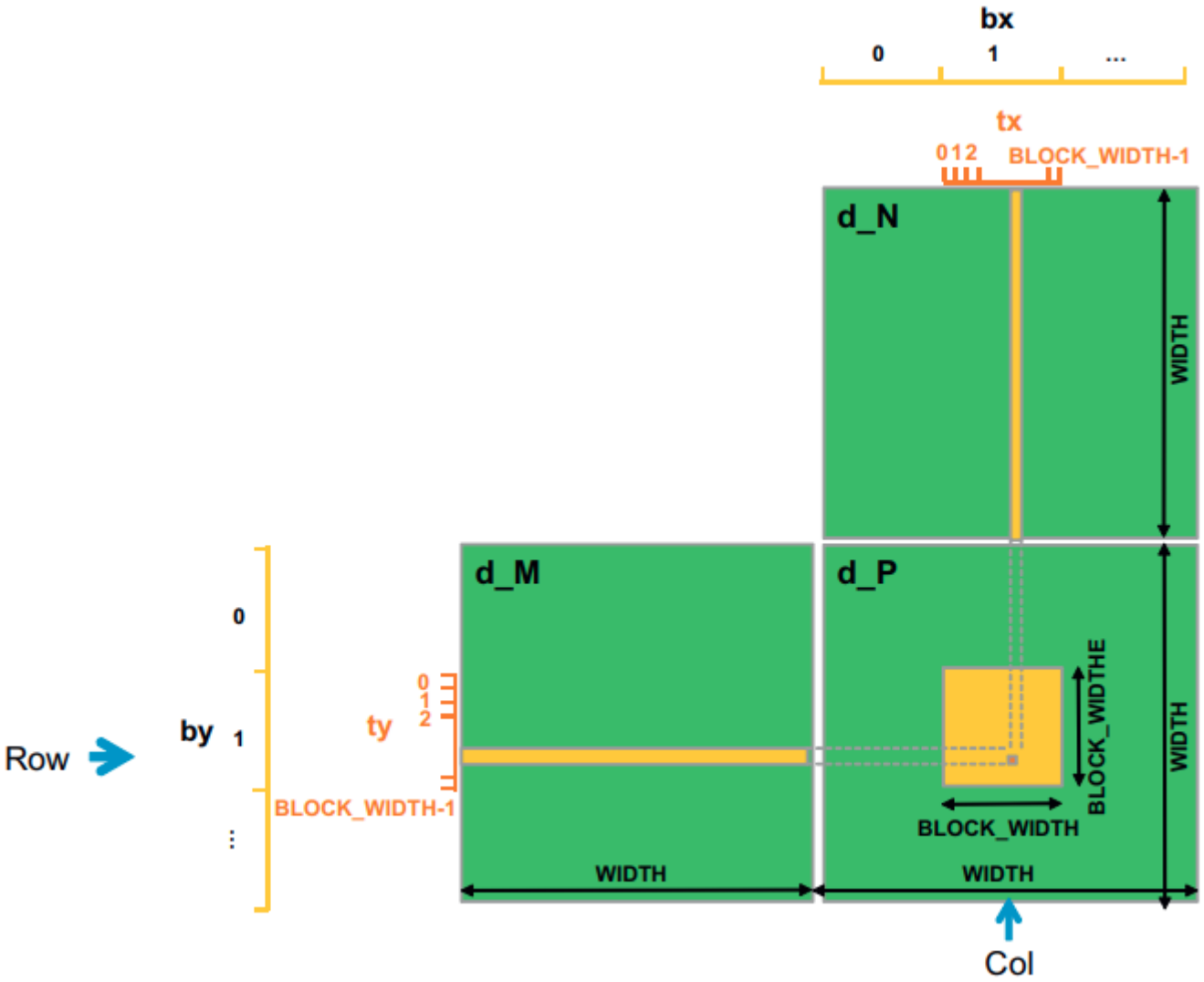


Matrix Multiplication

- A simple illustration of the basic features of memory and thread management in CUDA programs
  - Thread index usage
  - Memory layout
  - Register usage
  - Assume square matrix for simplicity
  - Leave shared memory usage until later

Matrix Multiplication

- We map threads to d_P elements with the same approach as what we used for pictureKernel(). That is, each thread is responsible for calculating one d_P element.



Matrix Multiplication Kernel

BITS Pilani

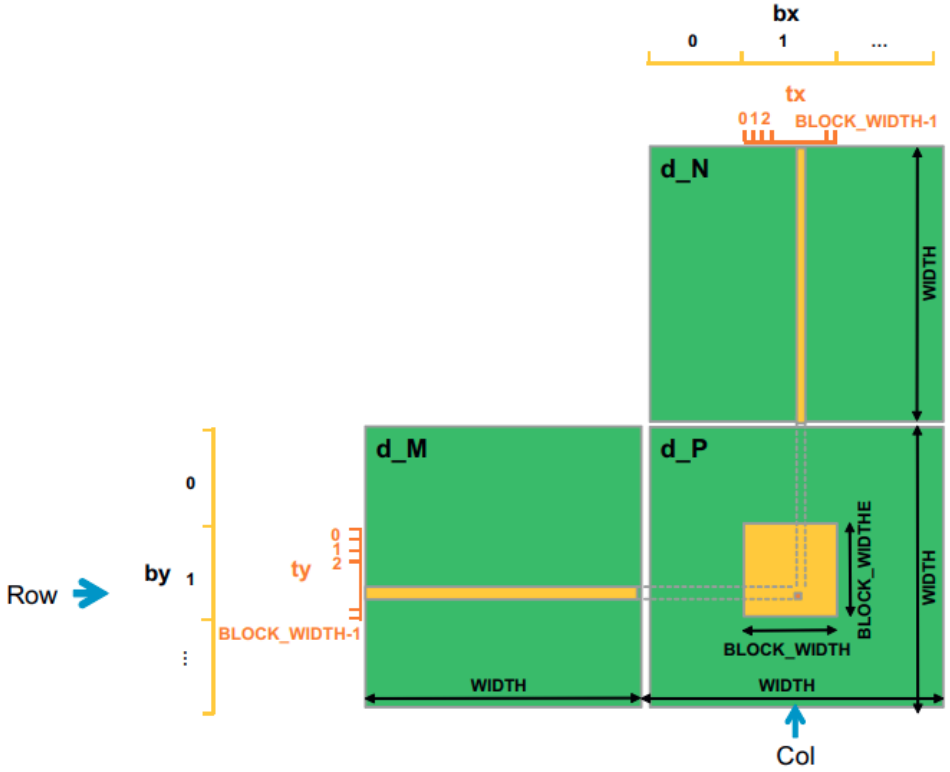
```
1  __global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width) {
2      // Calculate the row index of the d_Pelement and d_M
3      int Row = blockIdx.y*blockDim.y+threadIdx.y;
4      // Calculate the column index of d_P and d_N
5      int Col = blockIdx.x*blockDim.x+threadIdx.x;
6      if ((Row < Width) && (Col < Width)) {
7      float Pvalue = 0;
8      // each thread computes one element of the block sub-matrix
9      for (intk = 0; k < Width; ++k) {
10         Pvalue += d_M[Row*Width+k]*d_N[k*Width+Col];
11     }
12     d_P[Row*Width+Col] = Pvalue;
13 }
14 }
```

- We are assuming square matrices for the matrixMulKernel() so we replace both n and m with Width.

Matrix Multiplication – thread to data

BITS Pilani

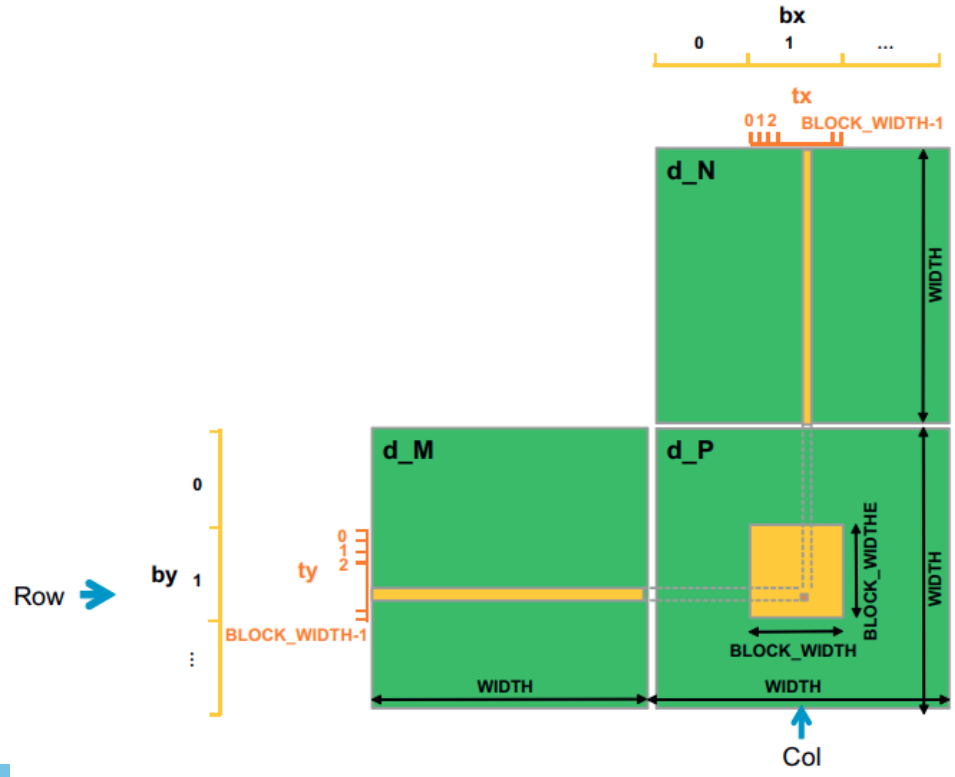
- With our thread-to-data mapping, we effectively divide d_P into square tiles
- Programmers often want to keep the block dimensions as an easily adjustable value in the host code



Matrix Multiplication – thread to data

BITS Pilani

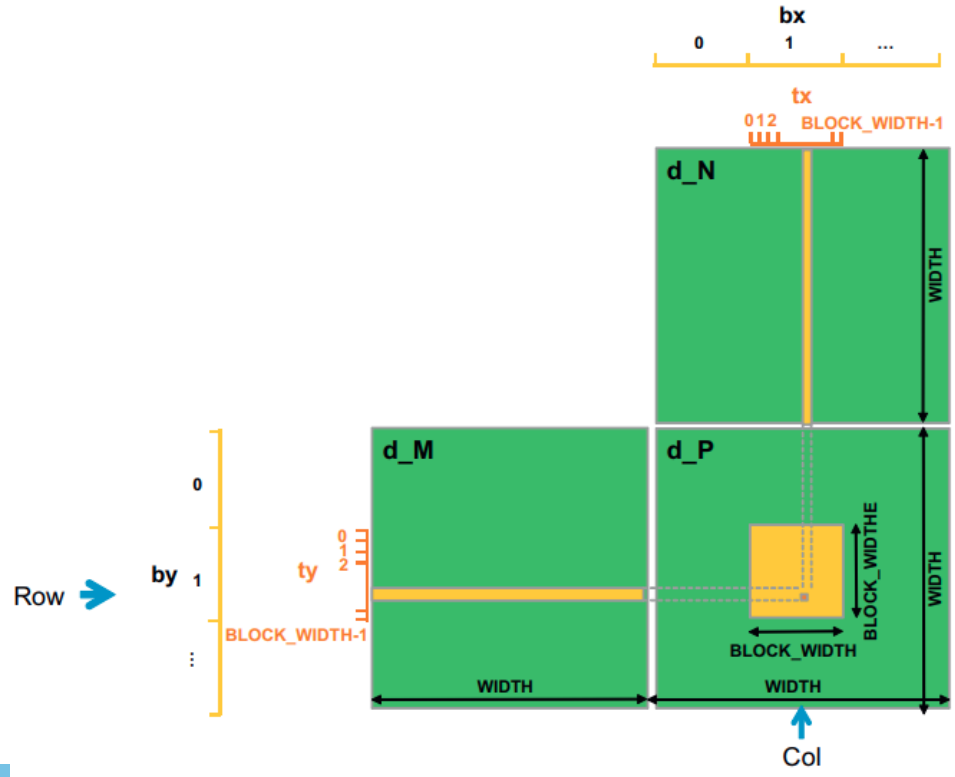
- Assume that we have a Width value of 1,000: 1000 x 1000 matrices need to be multiplied
- For a BLOCK_WIDTH value of 16, we will generate 16x16 blocks: 256 threads per block.
- There will be 64x64 blocks in the grid to cover all d_P elements



Matrix Multiplication – thread to data

BITS Pilani

- #define BLOCK_WIDTH 32
- We will generate 32x32 blocks: 1024 thread per block.
- There will be 32x32 blocks in the grid: 1,048,576 threads.

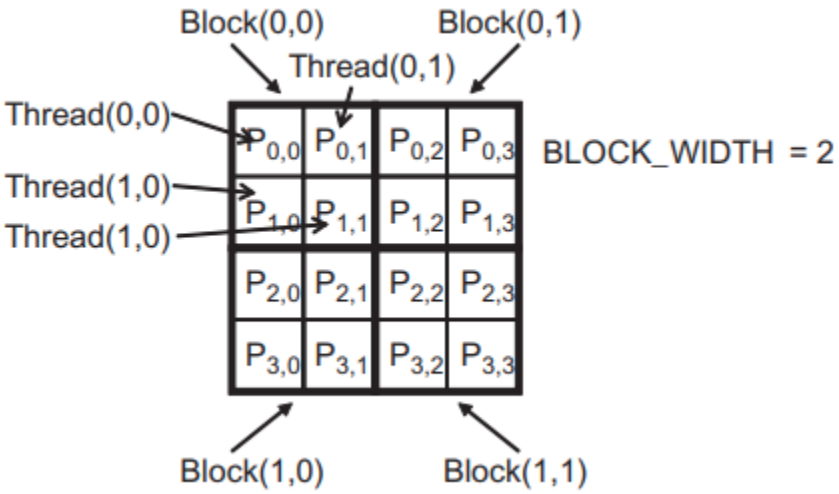


Host code


```
1  define BLOCK_WIDTH 16
2  // Setup the execution configuration
3  int NumBlocks = Width/BLOCK_WIDTH;
4  if (Width % BLOCK_WIDTH) NumBlocks++;
5  dim3 dimGrid(NumBlocks, NumbBlocks);
6  dim3 dimBlock(BLOCK_WIDTH, BLOCK_WIDTH);
7  // Launch the device computation threads!
8  matrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

- Host code for launching the matrixMulKernel() using a compile-time constant BLOCK_WIDTH to set up its configuration parameters

Small Example



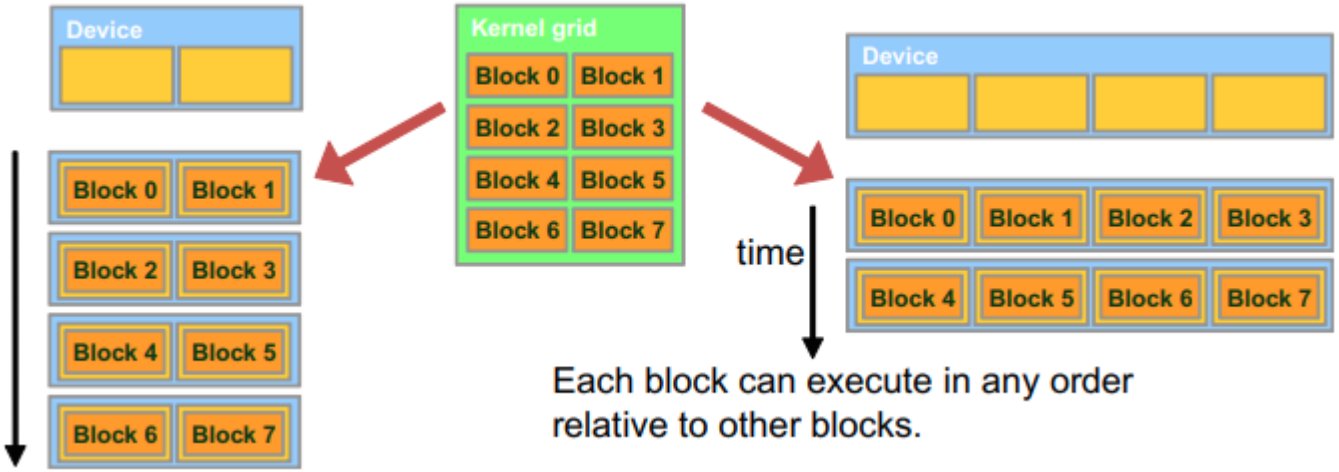
- 4x4 d_P with BLOCK_WIDTH=2
- The d_P matrix is now divided into four tiles and each block calculates one tile
- Thread(0,0) of block(0,0) calculates P0,0, whereas thread(0,0) of block(1,0) calculates P2,0.

$$P_{\text{blockIdx.y*blockDim.y + threadIdx.y,blockIdx.x*BLOCK\_WIDTH + threadIdx.x}} = P_{1*2+0,0*2+0} = P_{2,0}$$

Synchronization & transparent scalability

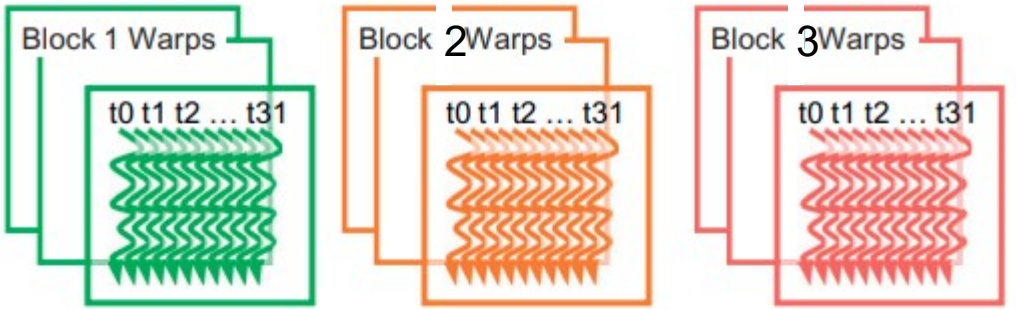
BITS Pilani

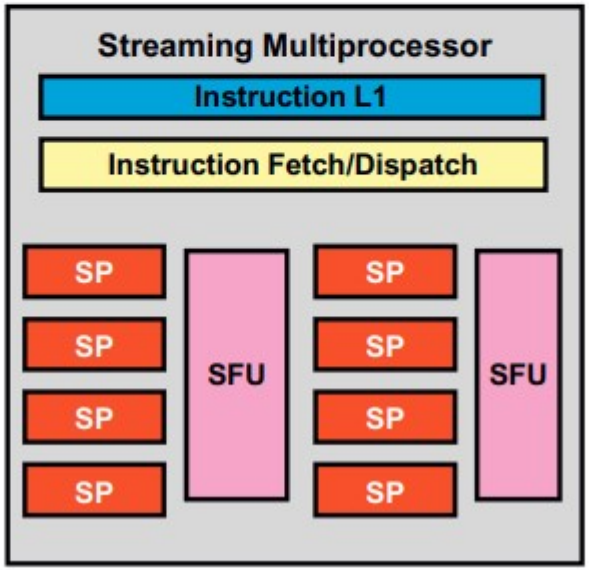
- When a kernel function calls *__syncthreads()*, all threads in a block will be held at the calling location until every thread in the block reaches the location
- Lack of synchronization constraints between blocks enables transparent scalability for CUDA programs



Thread Scheduling and Latency Tolerance

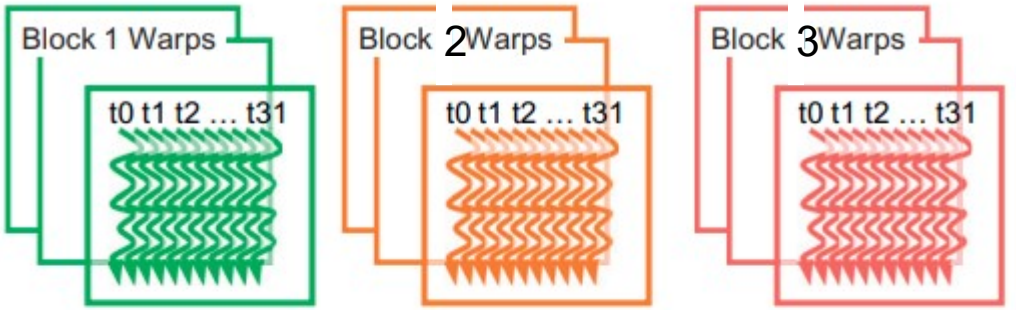
- The warp is the unit of thread scheduling in SMs
- Each warp consists of 32 threads of consecutive threadIdx values: threads 0-31 form the first warp, 32-63 the second warp, and so on
- if each block has 256 threads, we can determine that each block has 256432 or 8 warps. With three blocks in each SM, we have 8x3=24 warps in each SM

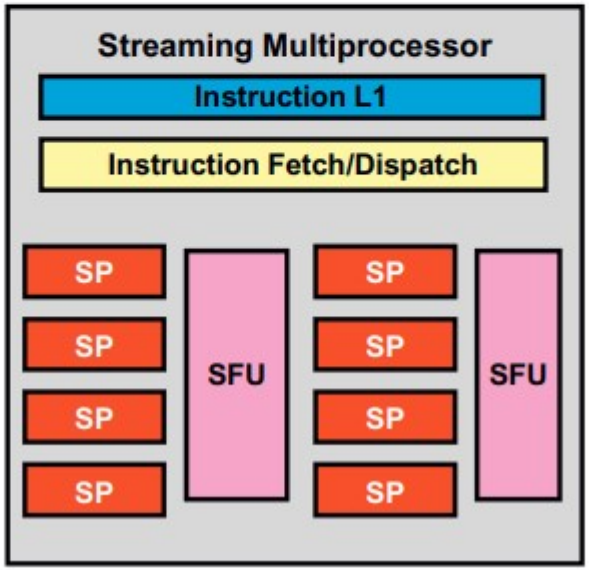




Thread Scheduling and Latency Tolerance

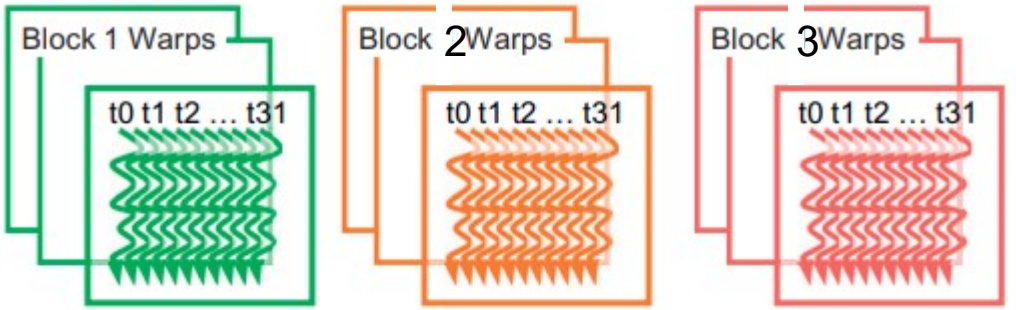
- An SM is designed to execute all threads in a warp following the single instruction, multiple data (SIMD) model
- At any instant in time, one instruction is fetched and executed for all threads in the warp
- All threads in a warp will always have the same execution timing
- Hardware streaming processors (SPs) actually execute instructions

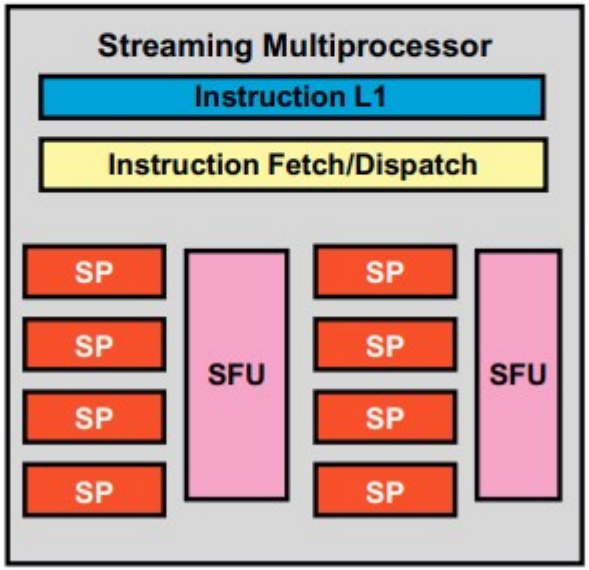




Thread Scheduling and Latency Tolerance

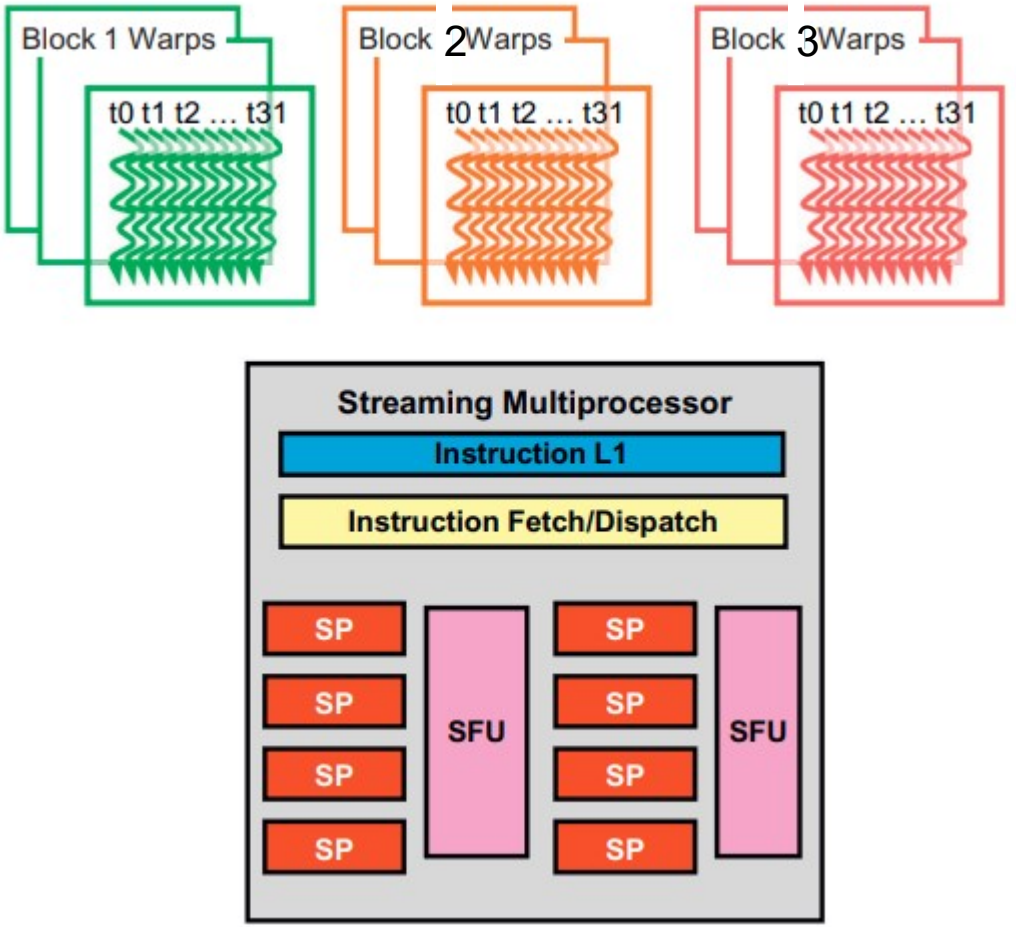
- Hardware can execute instructions for a small subset of all warps in the SM
- why do we need to have so many warps in an SM if it can only execute a small subset of them at any instant?
- The answer is that this is how CUDA processors efficiently execute long-latency operations such as global memory accesses.





Thread Scheduling and Latency Tolerance

- When an instruction executed by the threads in a warp needs to wait for the result of a previously initiated long-latency operation, the warp is not selected for execution
- Another resident warp that is no longer waiting for results will be selected for execution. If more than one warp is ready for execution, a priority mechanism is used to select one for execution.
- This mechanism of filling the latency time of operations with work from other threads called latency tolerance or latency hiding



Thread Scheduling and Latency Tolerance

- Warp scheduling is also used for tolerating other types of operation latencies such as pipelined floating-point arithmetic and branch instructions
- With enough warps around, the hardware will likely find a warp to execute at any point in time, thus making full use of the execution hardware in spite of these long-latency operations
- The selection of ready warps for execution does not introduce any idle time into the execution timeline, which is referred to as zero-overhead thread scheduling
- This ability to tolerate long operation latencies is the main reason why GPUs do not dedicate nearly as much chip area to cache memories and branch prediction mechanisms as CPUs

Example

- Assume that a CUDA device allows up to 8 blocks and 1,024 threads per SM. It allows up to 512 threads in each block. For matrix-matrix multiplication, should we use 8 x 8, 16 x 16, or 32 x 32 thread blocks?
- Case 8x8:
  - If we use 8 x 8 blocks, each block would have only 64 threads. We will need 1,024/64 = 12 blocks to fully occupy an SM. However, since there is a limitation of up to 8 blocks in each SM, we will end up with only 64 x 8 = 512 threads in each SM. This means that the SM execution resources will likely be underutilized because there will be fewer warps to schedule around long-latency operations.
- Case 16x16:
  - The 16 3 16 blocks give 256 threads per block. This means that each SM can take 1,0244256 5 4 blocks. This is within the 8-block limitation.

Example

- Assume that a CUDA device allows up to 8 blocks and 1,024 threads per SM. It allows up to 512 threads in each block. For matrix-matrix multiplication, should we use 8 x 8, 16 x 16, or 32 x 32 thread blocks?
- Case 16x16:
  - The 16 3 16 blocks give 256 threads per block. This means that each SM can take 1,0244256 5 4 blocks. This is within the 8-block limitation. This is a good configuration since we will have full thread capacity in each SM and a maximal number of warps for scheduling around the long latency operations.
- The 32 x 32 blocks would give 1,024 threads in each block, exceeding the limit of 512 threads per block for this device.
- Note that this is an oversimplified exercise. The usage of other resources, such as registers and shared memory, must also be considered when determining the most appropriate block dimensions.

Importance Of Memory Access Efficiency

BITS Pilani

```
11  __global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width) {
12      // Calculate the row index of the d_Pelement and d_M
13      int Row = blockIdx.y*blockDim.y+threadIdx.y;
14      // Calculate the column index of d_P and d_N
15      int Col = blockIdx.x*blockDim.x+threadIdx.x;
16      if ((Row < Width) && (Col < Width)) {
17      float Pvalue = 0;
18      // each thread computes one element of the block sub-matrix
19      for (intk = 0; k < Width; ++k) {
20          Pvalue += d_M[Row*Width+k]*d_N[k*Width+Col];
21      }
22      d_P[Row*Width+Col] = Pvalue;
23  }
24  }
```

- In every iteration of this loop, two global memory accesses are performed for one floating-point multiplication and one floating-point addition.
- Compute to global memory access (CGMA) ratio: 1:1 or 1
- CGMA has major implications on the performance of a CUDA kernel.

Importance Of Memory Access Efficiency

BITS Pilani

```
11  __global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width) {
12      // Calculate the row index of the d_Pelement and d_M
13      int Row = blockIdx.y*blockDim.y+threadIdx.y;
14      // Calculate the column index of d_P and d_N
15      int Col = blockIdx.x*blockDim.x+threadIdx.x;
16      if ((Row < Width) && (Col < Width)) {
17      float Pvalue = 0;
18      // each thread computes one element of the block sub-matrix
19      for (intk = 0; k < Width; ++k) {
20          Pvalue += d_M[Row*Width+k]*d_N[k*Width+Col];
21      }
22      d_P[Row*Width+Col] = Pvalue;
23  }
24  }
```

- In a high-end device today, the global memory bandwidth is around 200 GB/s. With 4 bytes in each single-precision floating-point value, one can expect to load no more than 50 (200/4) giga single-precision operands per second.
  - Speed is limited to 50 GFLOPS where as peak performance is 1500 GFLOPS
  - need a CGMA value of 30

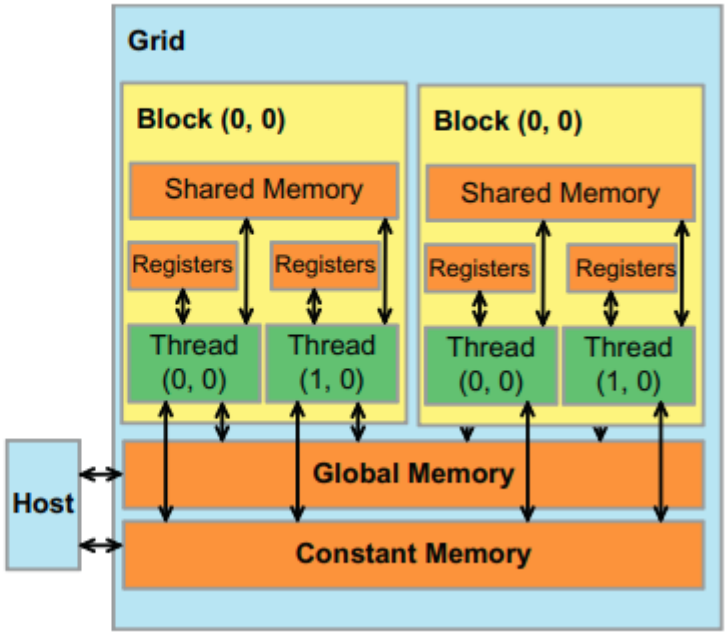
CUDA Variable Type Qualifiers

Device code can:

- R/W per-thread registers
- R/W per-thread local memory
- R/W per-block shared memory
- R/W per-grid global memory
- Read only per-grid constant memory

Host code can

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



Variable Declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
`__device__ __shared__ int SharedVar;`	Shared	Block	Kernel
`__device__ int GlobalVar;`	Global	Grid	Application
`__device__ __constant__ int ConstVar;`	Constant	Grid	Application

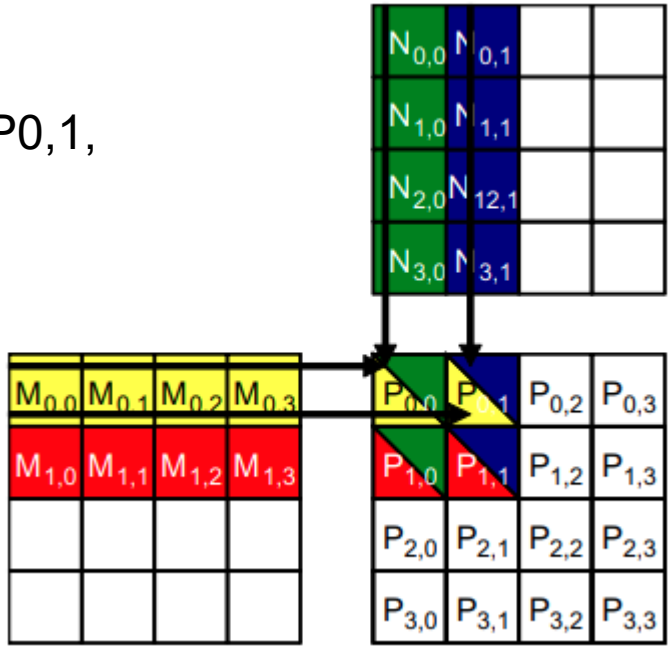
A Strategy for Reducing Global Memory Traffic

- There is an intrinsic trade-off in the use of device memories in CUDA:
  - global memory is large but slow, whereas the shared memory is small but fast
  - A common strategy is partition the data into subsets called tiles so that each tile fits into the shared memory
  - An important criterion is that the kernel computation on these tiles can be done independently of each other. Note that not all data structures can be partitioned into tiles given an arbitrary kernel function.

Matrix Multiplication

BITS Pilani

We use four 2 x 2 blocks to compute the P matrix. These four threads compute P0,0, P0,1, P1,0, and P1,1.



- Global memory accesses performed by threads in block0.
  - significant overlap in terms of the M and N elements they access

Access order

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

Tiling

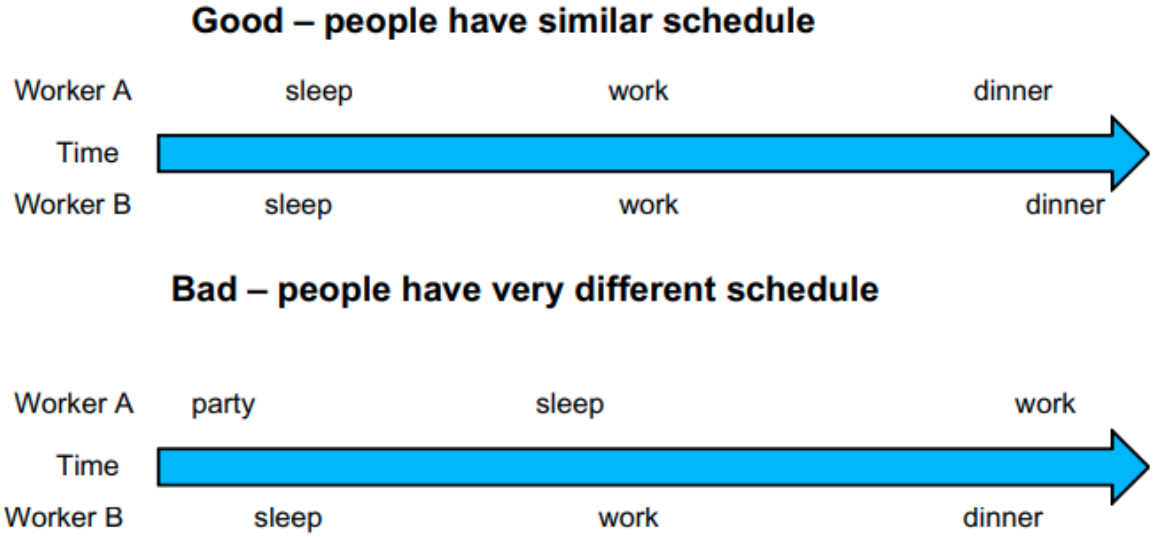
- Both thread0,0 and thread0,1 access row 0 elements of M from the global memory.
- If we can somehow manage to have thread0,0 and thread1,0 to collaborate so that these M elements are only loaded from global memory once, we can reduce the total number of accesses to the global memory by half

- With N x N blocks, the potential reduction of global memory traffic would be N. That is, if we use 16 x 16 blocks, one can potentially reduce the global memory traffic to 1/16 through collaboration between threads.

Tiled Algorithms

BITS Pilani

- Tiled algorithms are very similar to carpooling arrangements



- Carpooling requires synchronization among people

Tiled Matrix-matrix Multiplication Kernel


```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P,
int Width) {
1.  __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
2.  __shared__ float Nds[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 d_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 d_M and d_N tiles required to compute d_P element
8.  for (int m = 0; m < Width/TILE_WIDTH; ++m) {
// Coolaborative loading of d_M and d_N tiles into shared memory
9.  Mds[ty][tx] = d_M[Row*Width + m*TILE_WIDTH + tx];
10.  Nds[ty][tx] = d_N[(m*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.  d_P[Row*Width + Col] = Pvalue;
}
```

Execution phases of a tiled matrix multiplication.

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

time →

Q&A

innovate achieve lead

BITS Pilani
Pilani Campus

Thank You