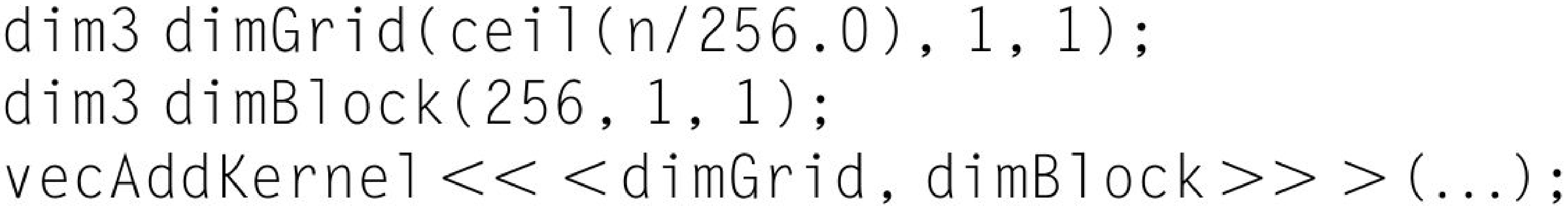
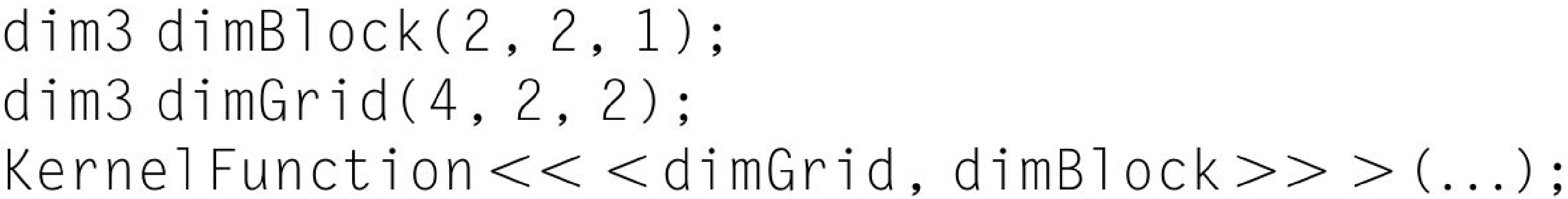
Data Parallel Execution Model

John H. Osorio Ríos

Cuda Thread Organization (1/2)

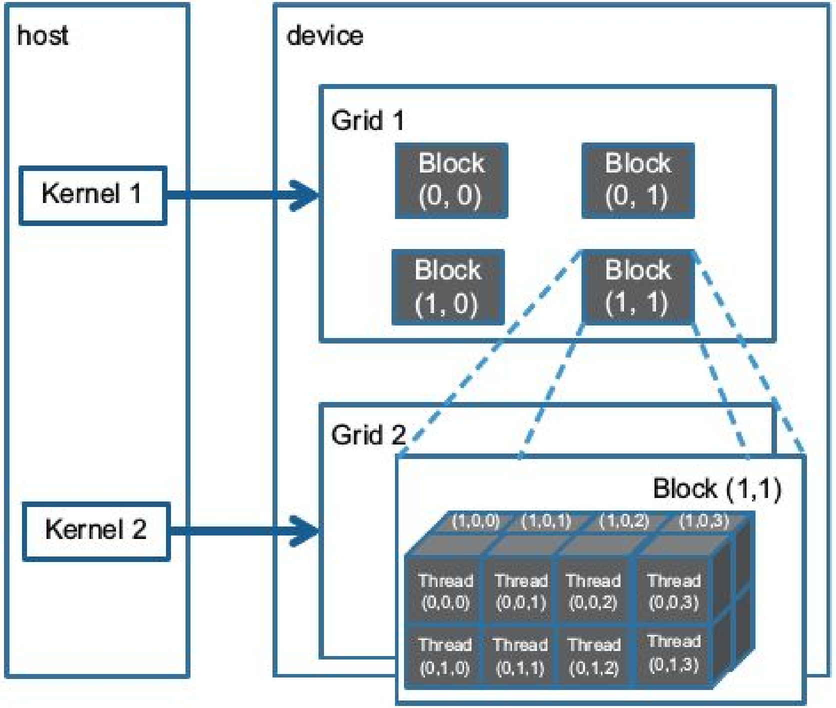


```
dim3 dimGrid(ceil(n/256.0), 1, 1);
dim3 dimBlock(256, 1, 1);
vecAddKernel<<<dimGrid, dimBlock>>>(...);
```



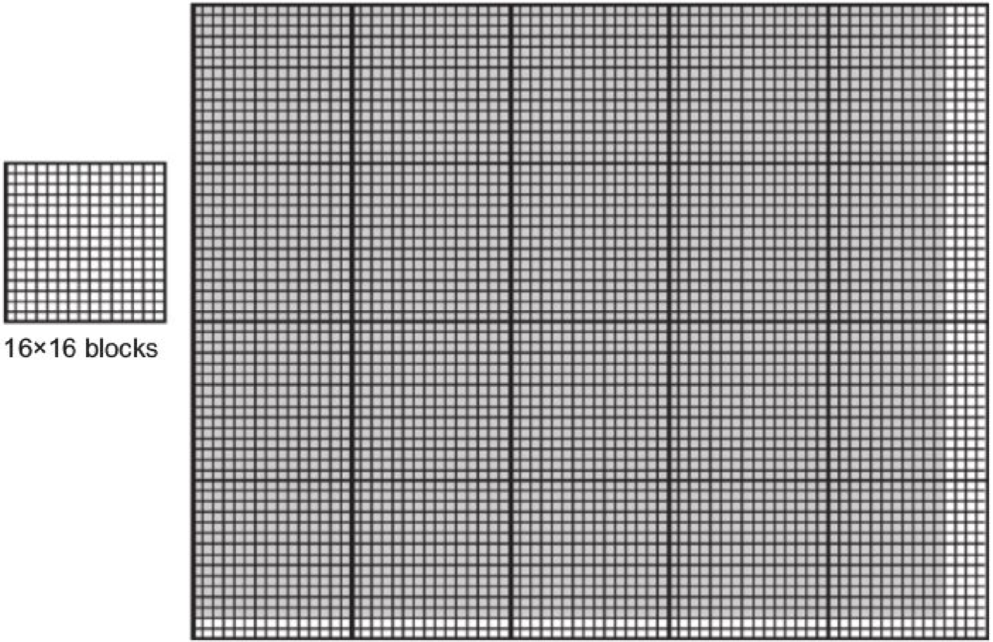
```
dim3 dimBlock(2, 2, 1);
dim3 dimGrid(4, 2, 2);
KernelFunction<<<dimGrid, dimBlock>>>(...);
```

Cuda Thread Organization (2/2)



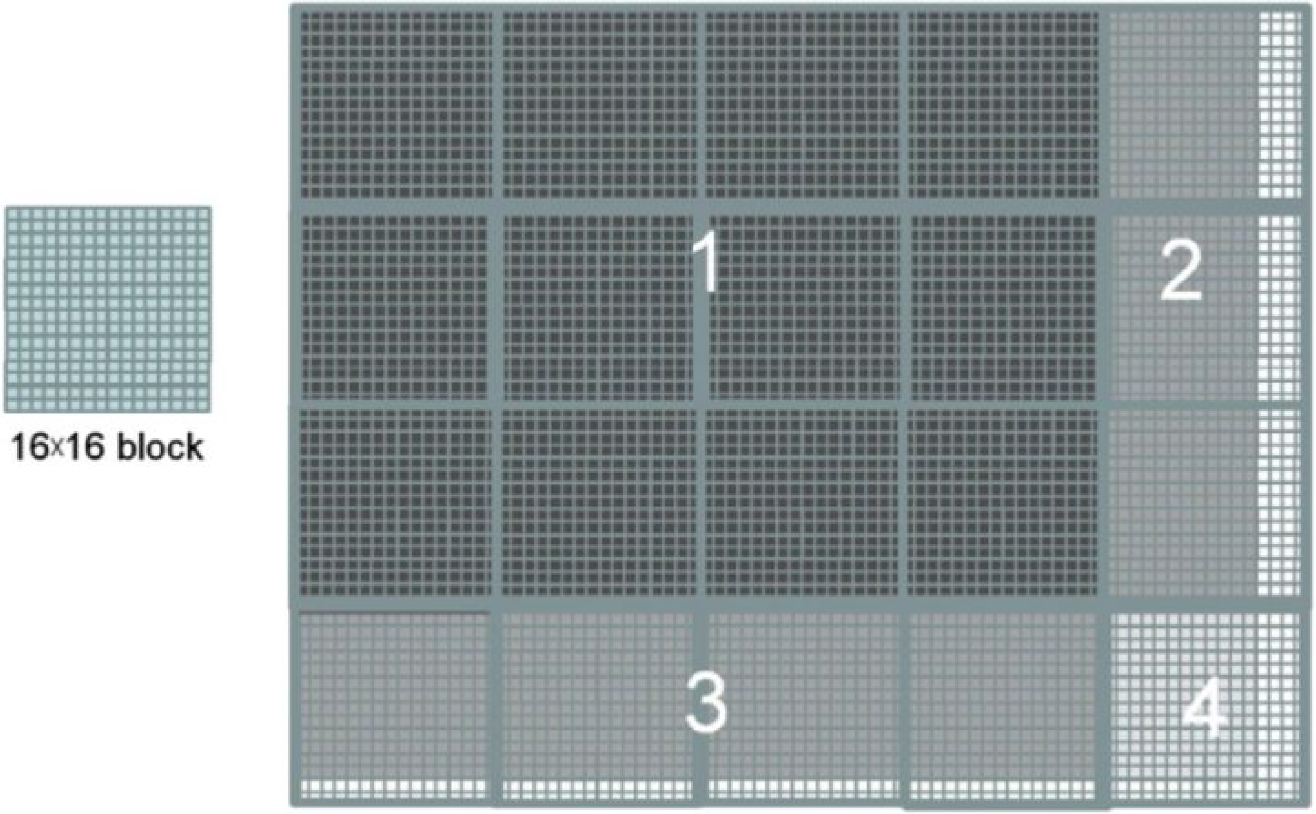
- blockDim
- threadIdx
- blockIdx
- gridDim

Mapping Threads to Multidimensional Data (1/4)

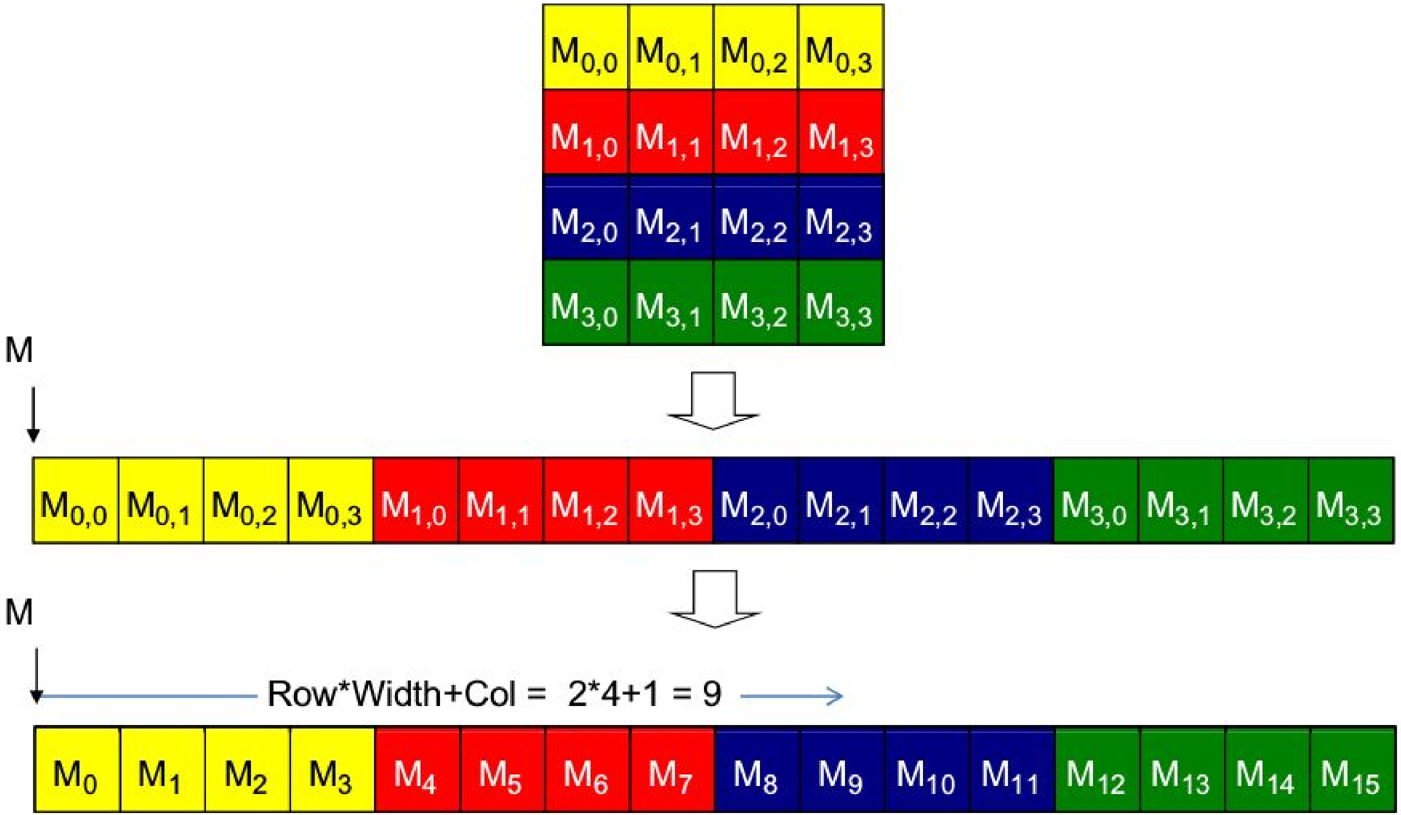


- 76 x 62 Pixels
- 5 x blocks
- 4 y blocks
- 16 x 16 threads in a block

Mapping Threads to Multidimensional Data (2/4)



Mapping Threads to Multidimensional Data (3/4)



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

M

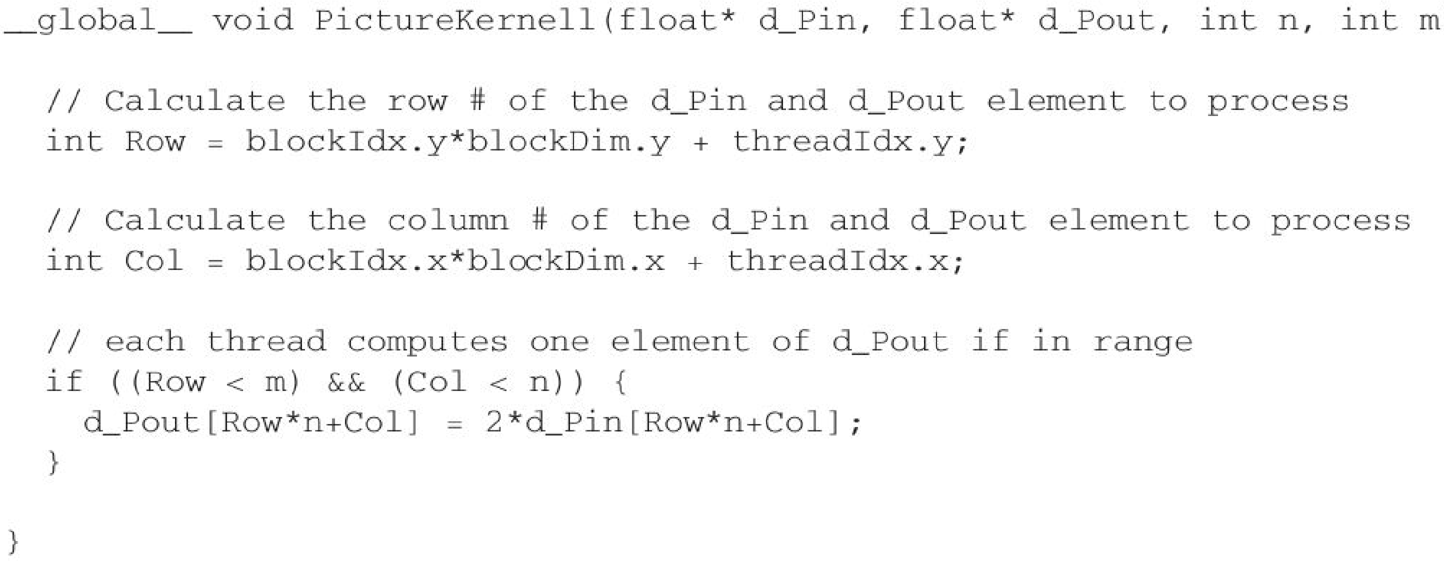
$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$	$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$	$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$	$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

M

Row*Width+Col = 2*4+1 = 9

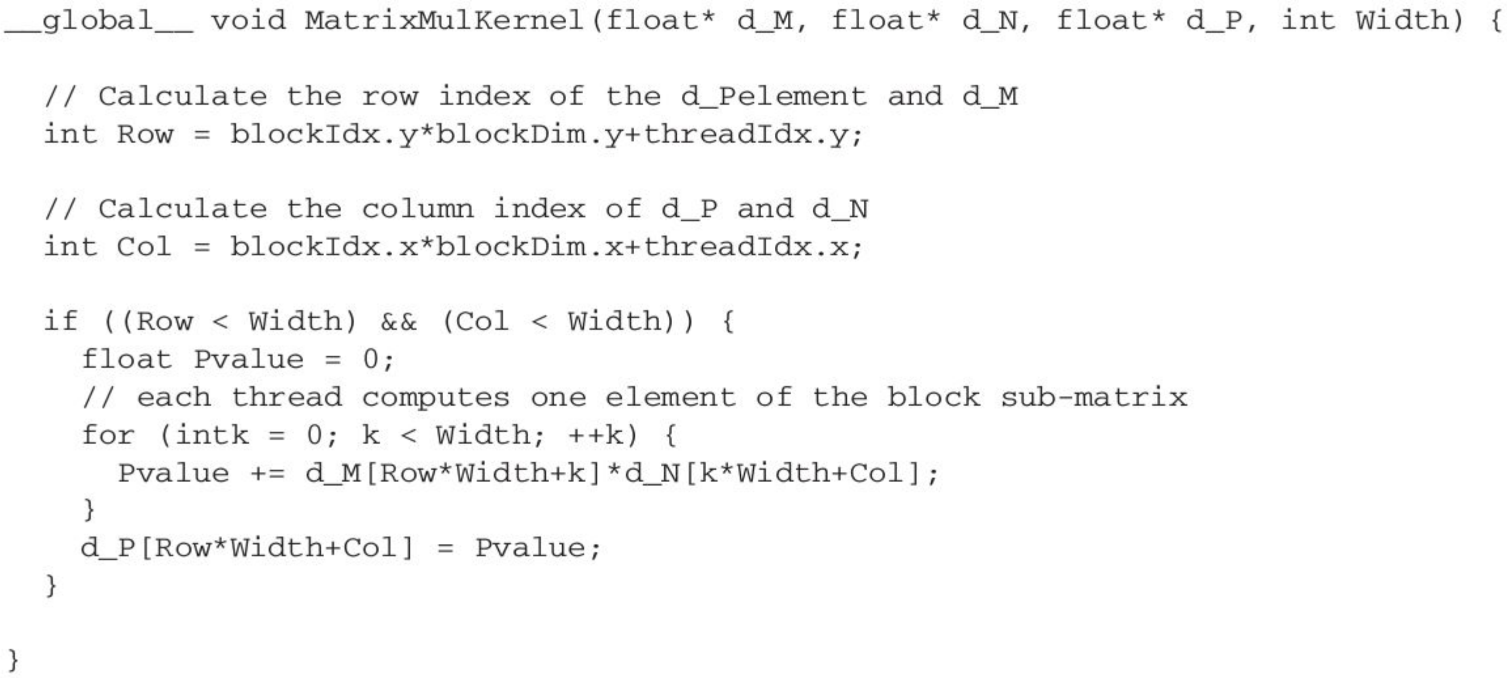
M_0	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8	M_9	M_{10}	M_{11}	M_{12}	M_{13}	M_{14}	M_{15}

Mapping Threads to Multidimensional Data (4/4)



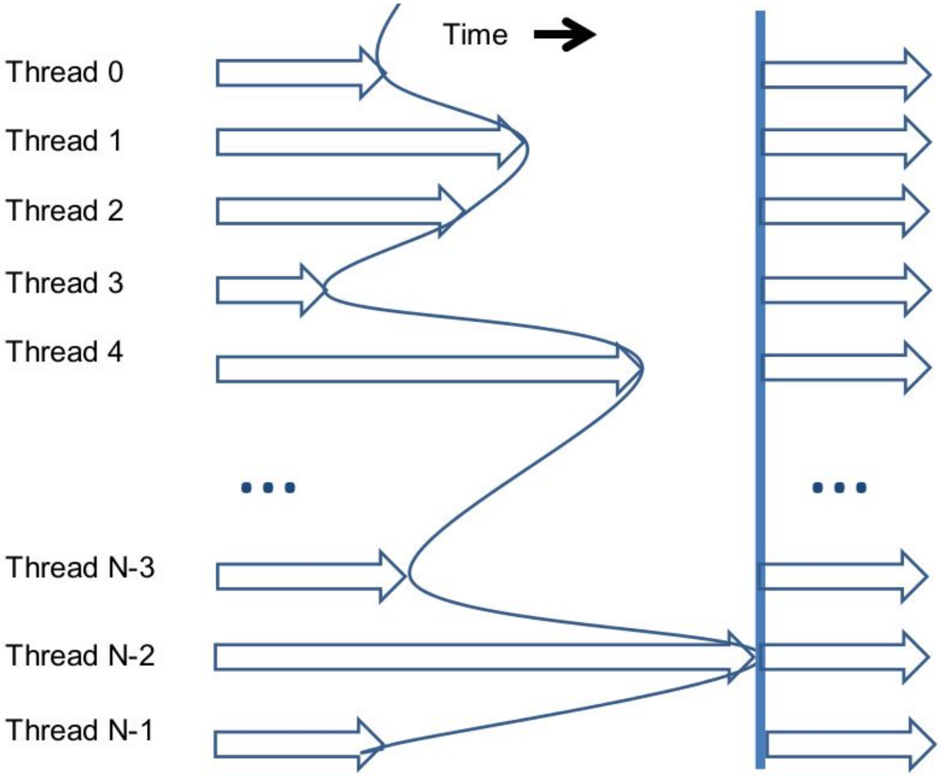
```
__global__ void PictureKernell(float* d_Pin, float* d_Pout, int n, int m) {

  // Calculate the row # of the d_Pin and d_Pout element to process
  int Row = blockIdx.y*blockDim.y + threadIdx.y;

  // Calculate the column # of the d_Pin and d_Pout element to process
  int Col = blockIdx.x*blockDim.x + threadIdx.x;

  // each thread computes one element of d_Pout if in range
  if ((Row < m) && (Col < n)) {
    d_Pout[Row*n+Col] = 2*d_Pin[Row*n+Col];
  }

}
```

Matrix-Matrix Multiplication



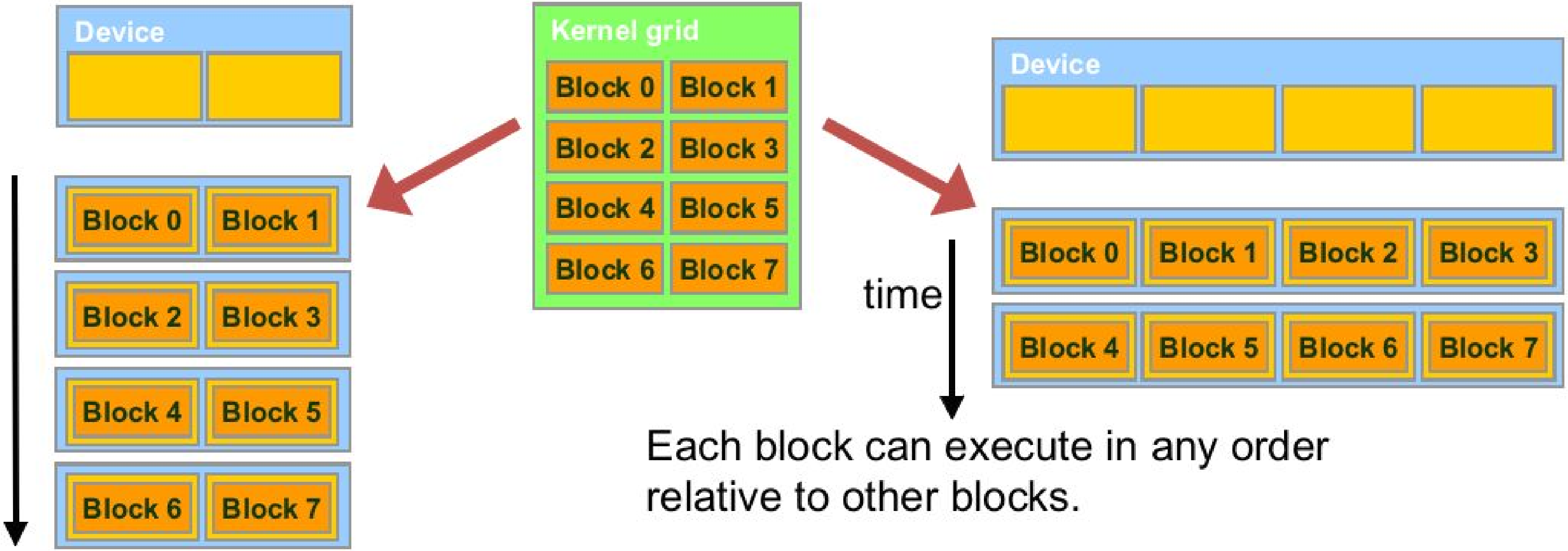
```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width) {

  // Calculate the row index of the d_Pelement and d_M
  int Row = blockIdx.y*blockDim.y+threadIdx.y;

  // Calculate the column index of d_P and d_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 (intk = 0; k < Width; ++k) {
      Pvalue += d_M[Row*Width+k]*d_N[k*Width+Col];
    }
    d_P[Row*Width+Col] = Pvalue;
  }

}
```

Synchronization and Transparent Scalability (1/2)



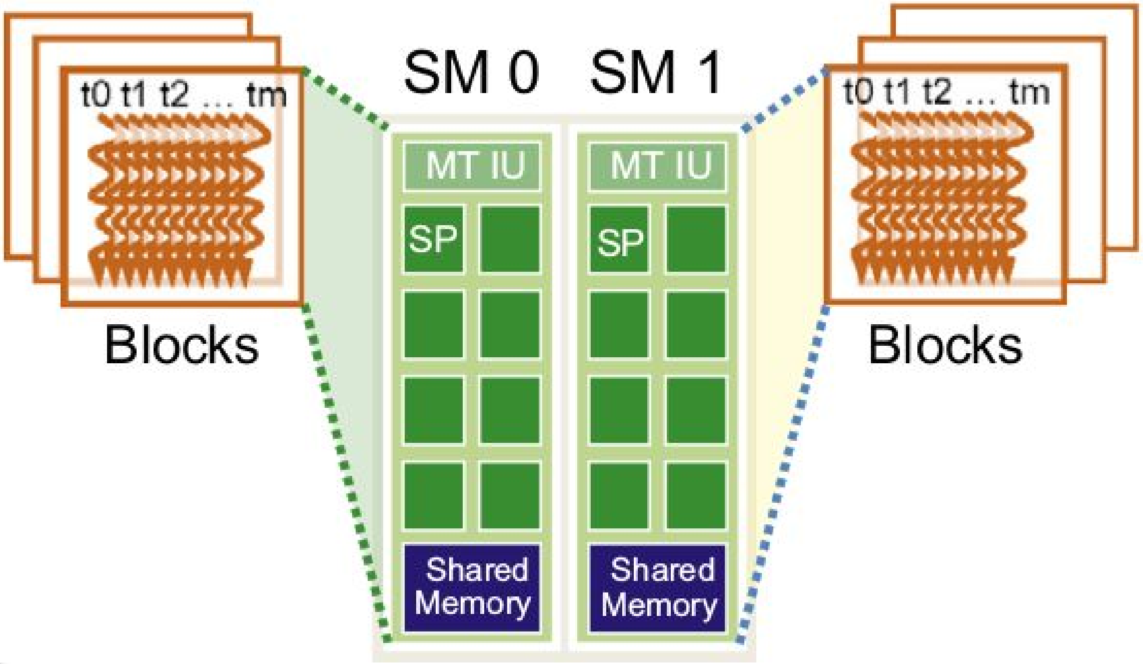
- __syncthreads()

Synchronization and Transparent Scalability (2/2)



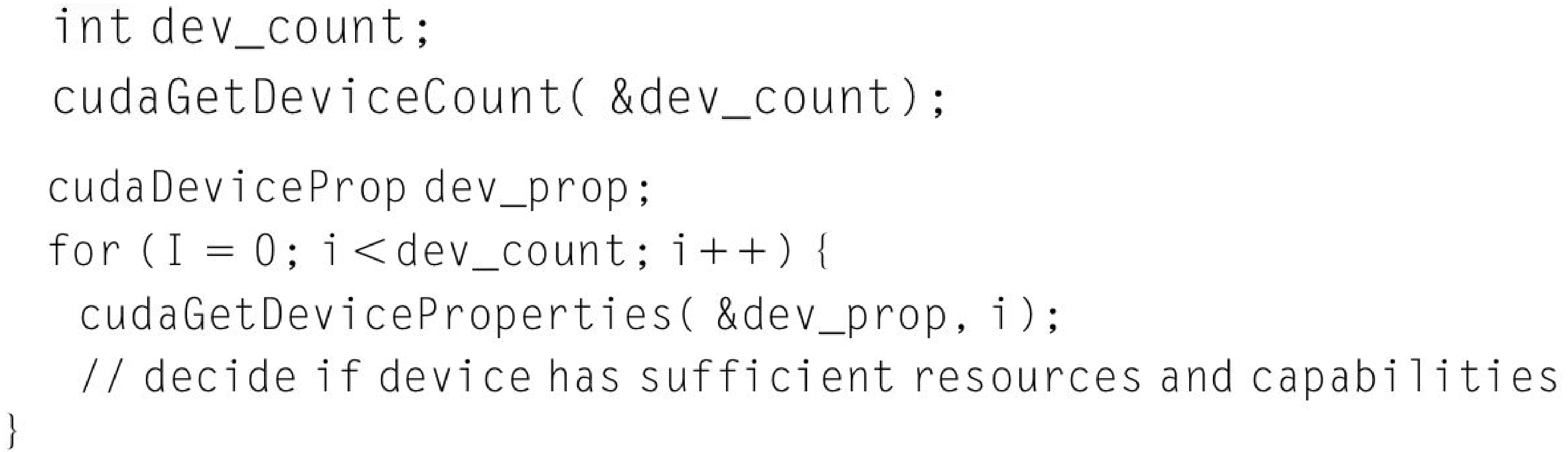
Each block can execute in any order relative to other blocks.

Assigning Resources to Blocks



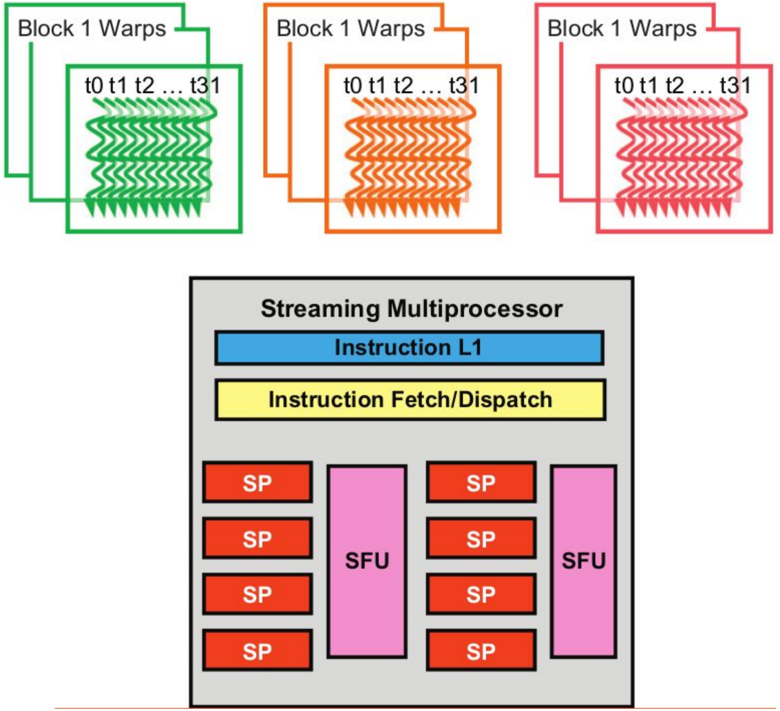
- 8 block to each SM
- 1536 threads to each SM
- 6 blocks of 256 threads
- 3 blocks of 512
- **12 blocks of 128 threads**

Querying Device Properties



```
int dev_count;
cudaGetDeviceCount( &dev_count);
cudaDeviceProp dev_prop;
for (I = 0; i<dev_count; i++) {
  cudaGetDeviceProperties( &dev_prop, i);
  // decide if device has sufficient resources and capabilities
}
```

Thread Scheduling and Latency Tolerance



- **Assume a CUDA device:**
  - 8 blocks per SM
  - 1024 threads per SM
  - 512 threads per block
- **For Matrix Multiplication:**
  - 8x8 thread blocks ?
  - 16x16 thread blocks ?
  - 32x32 thread blocks ?

THANKS

john@sirius.utp.edu.co