Last Class

1, Two ways to define shared memory

2, Common Cuda Programming Strategy

3, Memory Access Efficiency, CGMA

Two ways to Define Shared Mem

- Two ways of defining shared memory

Dynamic (second kind)

```
// when the size of the array isn't known at compile time...
__global__ void adj_diff(int *result, int *input)
{
  // use extern to indicate a __shared__ array will be
  // allocated dynamically at kernel launch time
  extern __shared__ int s_data[]; // This has to be one dimensional. If you use 2D tile, you have to
                                  // flatten the 2D tile into 1D shared memory.

  ...
}
```

```
// pass the size of the per-block array, in bytes, as the third
// argument to the triple chevrons,
// used with deviceQuery functions.
adj_diff<<<num_blocks, block_size, block_size * sizeof(int)>>>(r,i);
```

Today

1, Strategy to reduce global memory traffic

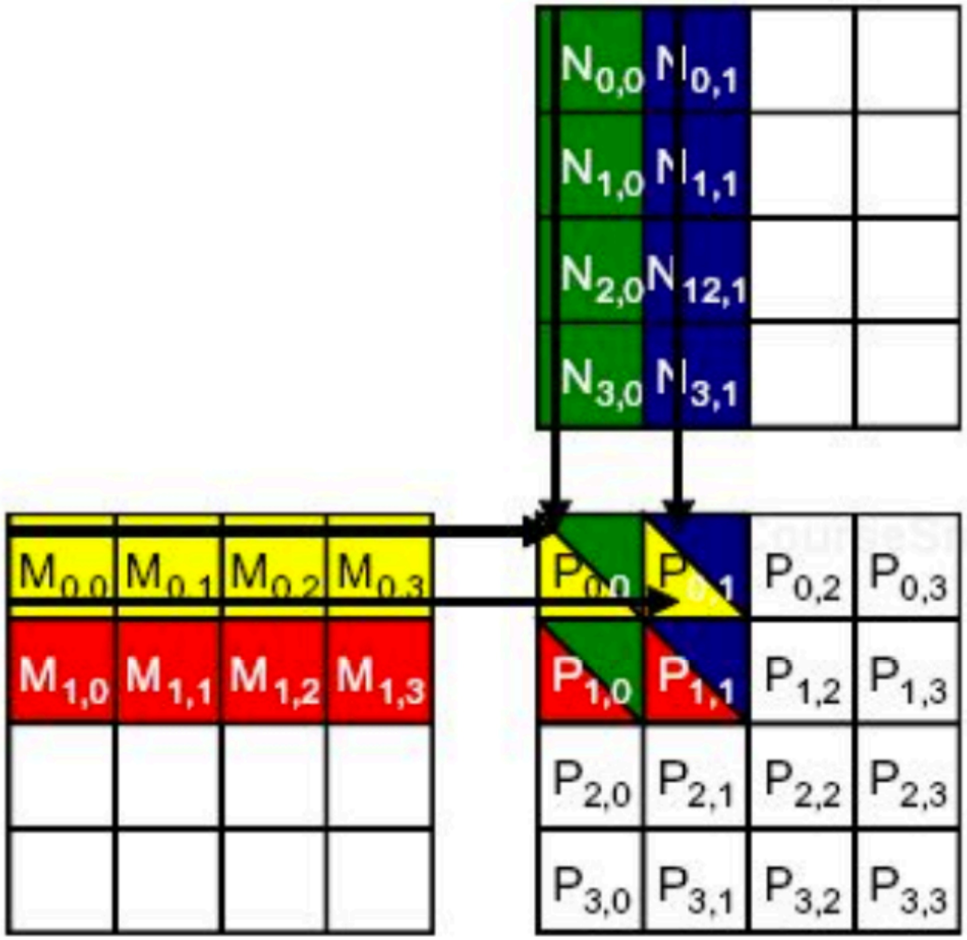
2, Tiled Matrix-matrix multiplication kernel

3, Analyzing how much global memory access is reduced for the new kernel?

Small Shared Mem

- Global Memory is big, but very slow.
- Shared memory is 100X faster, but small.
- Strategy to use Shared Memory
    - Partition data into tile
    - Each tile can fit into shared memory.
    - Tiles should be independent to each other when they are being processed.
    - Reduce Global Memory Traffic

Strategy to Reduce Global Mem Traffic

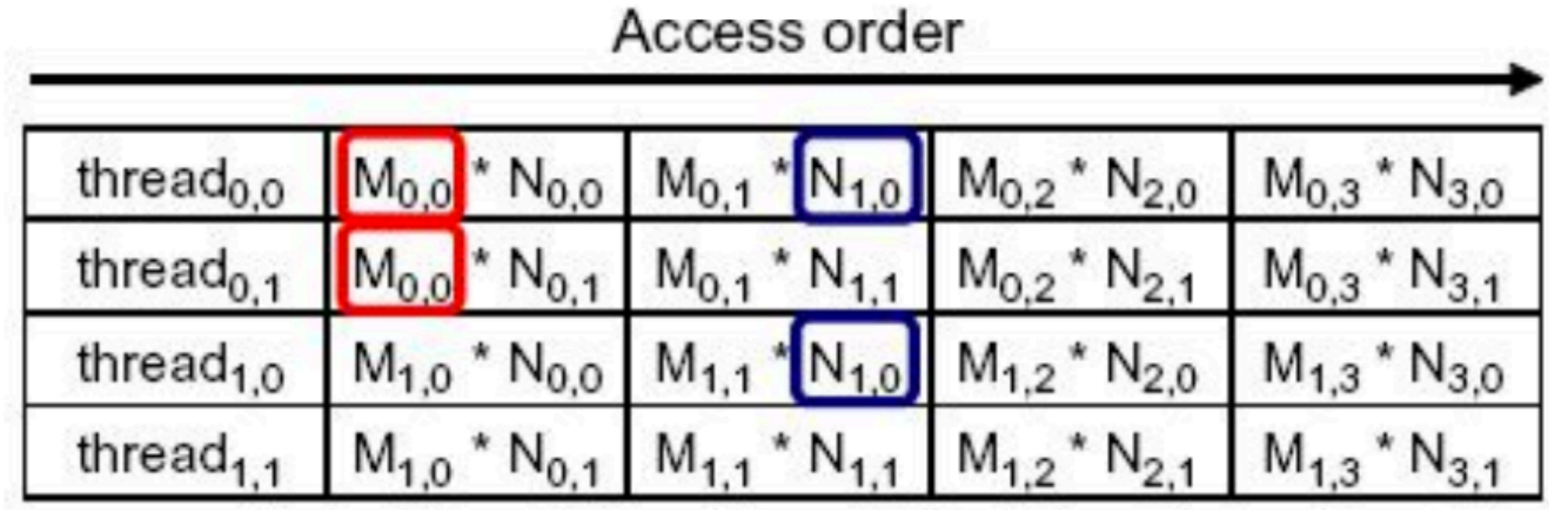


$N_{0,0}$ $N_{0,1}$
$N_{1,0}$ $N_{1,1}$
$N_{2,0}$ $N_{12,1}$
$N_{3,0}$ $N_{3,1}$
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$P_{0,0}$ $P_{0,1}$ $P_{0,2}$ $P_{0,3}$
$P_{1,0}$ $P_{1,1}$ $P_{1,2}$ $P_{1,3}$
$P_{2,0}$ $P_{2,1}$ $P_{2,2}$ $P_{2,3}$
$P_{3,0}$ $P_{3,1}$ $P_{3,2}$ $P_{3,3}$

FIGURE 5.5

A small example of matrix multiplication. For brevity, We show d_M [y*Width + x], d_N[y*Width + x], d_P[y*Width + x] as $M_{y,x}$, $N_{y,x}$, $P_{y,x}$, respectively.

Strategy to Reduce Global Mem Traffic

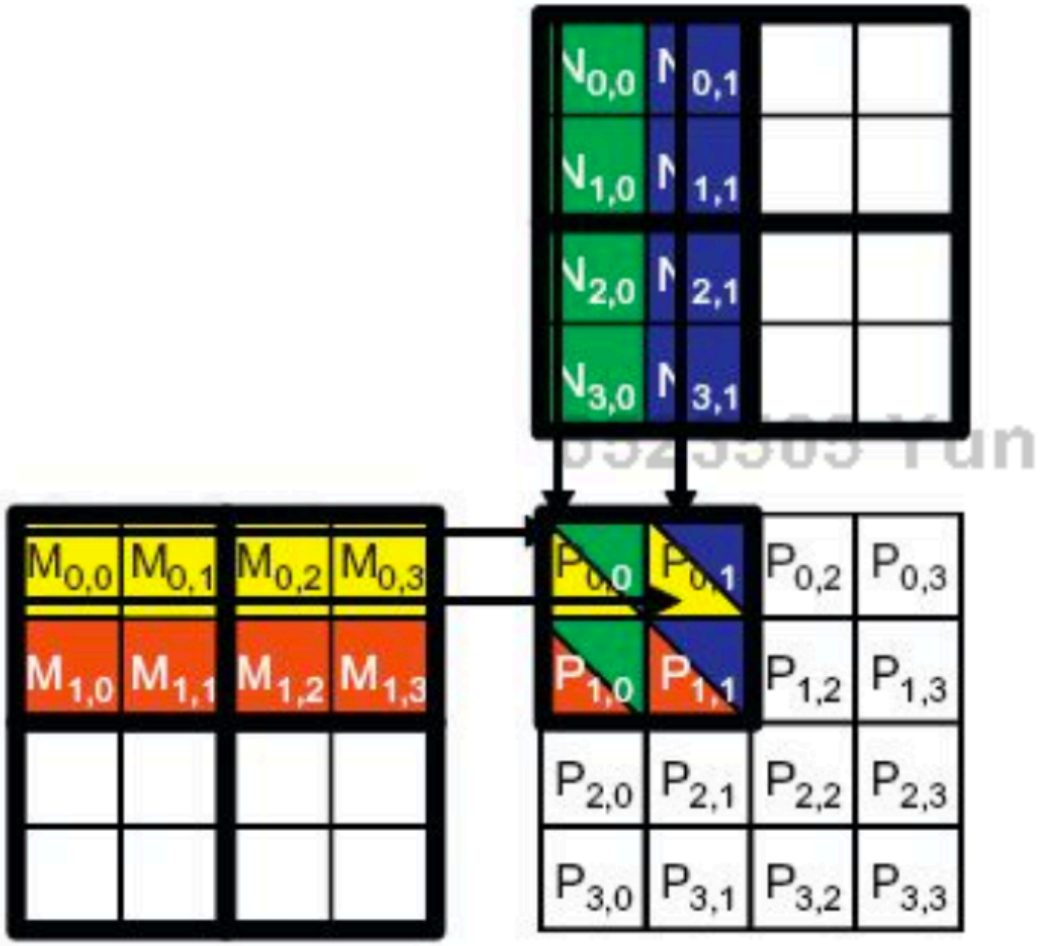

Access order

$\text{thread}_{0,0}$	$\boxed{M_{0,0}} * N_{0,0}$	$M_{0,1} * \boxed{N_{1,0}}$	$M_{0,2} * N_{2,0}$	$M_{0,3} * N_{3,0}$
$\text{thread}_{0,1}$	$\boxed{M_{0,0}} * N_{0,1}$	$M_{0,1} * N_{1,1}$	$M_{0,2} * N_{2,1}$	$M_{0,3} * N_{3,1}$
$\text{thread}_{1,0}$	$M_{1,0} * N_{0,0}$	$M_{1,1} * \boxed{N_{1,0}}$	$M_{1,2} * N_{2,0}$	$M_{1,3} * N_{3,0}$
$\text{thread}_{1,1}$	$M_{1,0} * N_{0,1}$	$M_{1,1} * N_{1,1}$	$M_{1,2} * N_{2,1}$	$M_{1,3} * N_{3,1}$

URE 5.6

bal memory accesses performed by threads in $\text{block}_{0,0}$.

Tiled Matrix-Matrix Multiplication

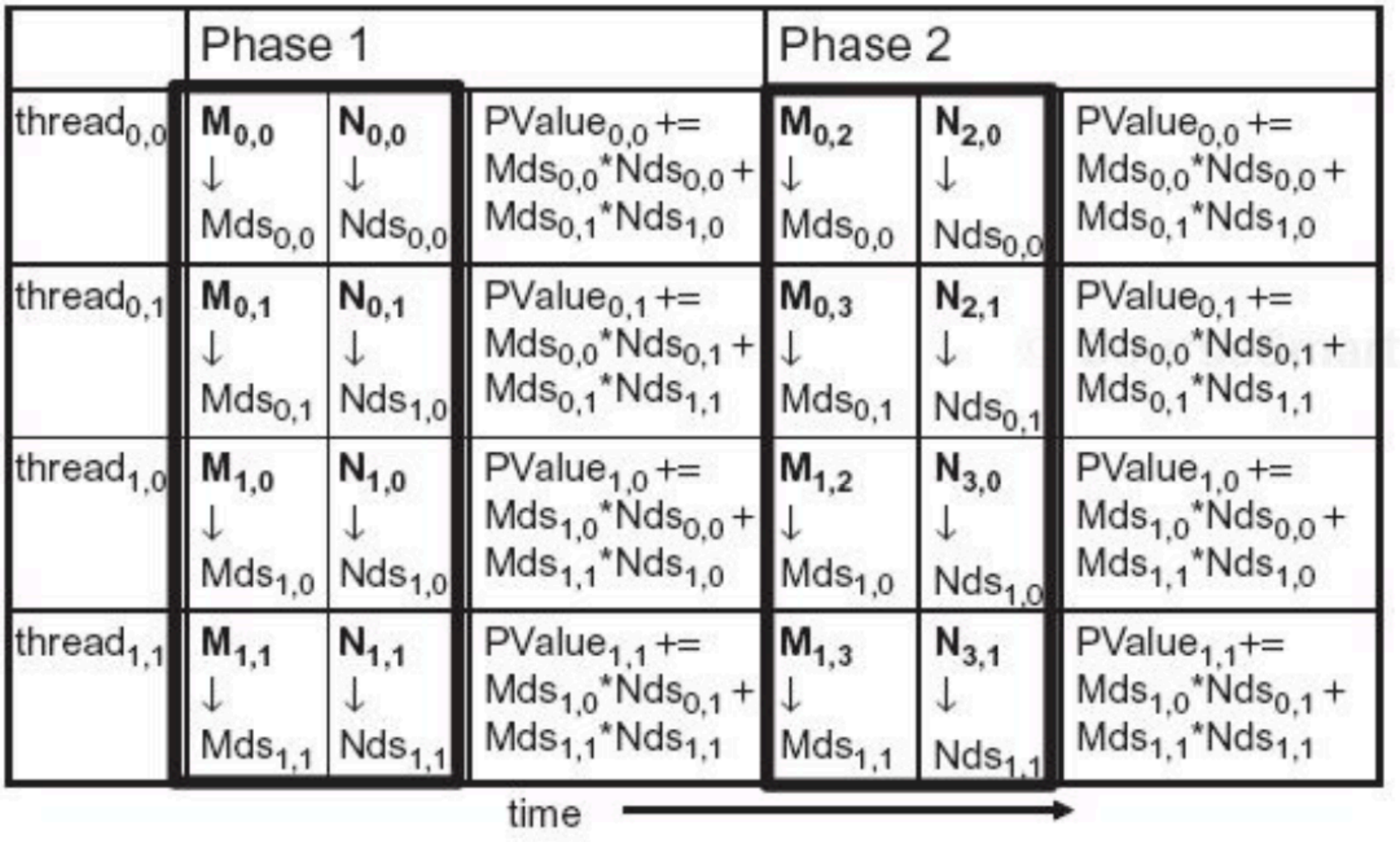


$N_{0,0}$ $N_{0,1}$
$N_{1,0}$ $N_{1,1}$
$N_{2,0}$ $N_{2,1}$
$N_{3,0}$ $N_{3,1}$
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$P_{0,0}$ $P_{0,1}$ $P_{0,2}$ $P_{0,3}$
$P_{1,0}$ $P_{1,1}$ $P_{1,2}$ $P_{1,3}$
$P_{2,0}$ $P_{2,1}$ $P_{2,2}$ $P_{2,3}$
$P_{3,0}$ $P_{3,1}$ $P_{3,2}$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

FIGURE 5.10

Tiling M and N matrices to utilize shared memory.

Tiled Matrix-Matrix Multiplication



	Phase 1			Phase 2		
$thread_{0,0}$	$\mathbf{M}_{0,0}$ ↓ $Mds_{0,0}$	$\mathbf{N}_{0,0}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$	$\mathbf{M}_{0,2}$ ↓ $Mds_{0,0}$	$\mathbf{N}_{2,0}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$
$thread_{0,1}$	$\mathbf{M}_{0,1}$ ↓ $Mds_{0,1}$	$\mathbf{N}_{0,1}$ ↓ $Nds_{1,0}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$	$\mathbf{M}_{0,3}$ ↓ $Mds_{0,1}$	$\mathbf{N}_{2,1}$ ↓ $Nds_{0,1}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$
$thread_{1,0}$	$\mathbf{M}_{1,0}$ ↓ $Mds_{1,0}$	$\mathbf{N}_{1,0}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$	$\mathbf{M}_{1,2}$ ↓ $Mds_{1,0}$	$\mathbf{N}_{3,0}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$
$thread_{1,1}$	$\mathbf{M}_{1,1}$ ↓ $Mds_{1,1}$	$\mathbf{N}_{1,1}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$	$\mathbf{M}_{1,3}$ ↓ $Mds_{1,1}$	$\mathbf{N}_{3,1}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$

time →

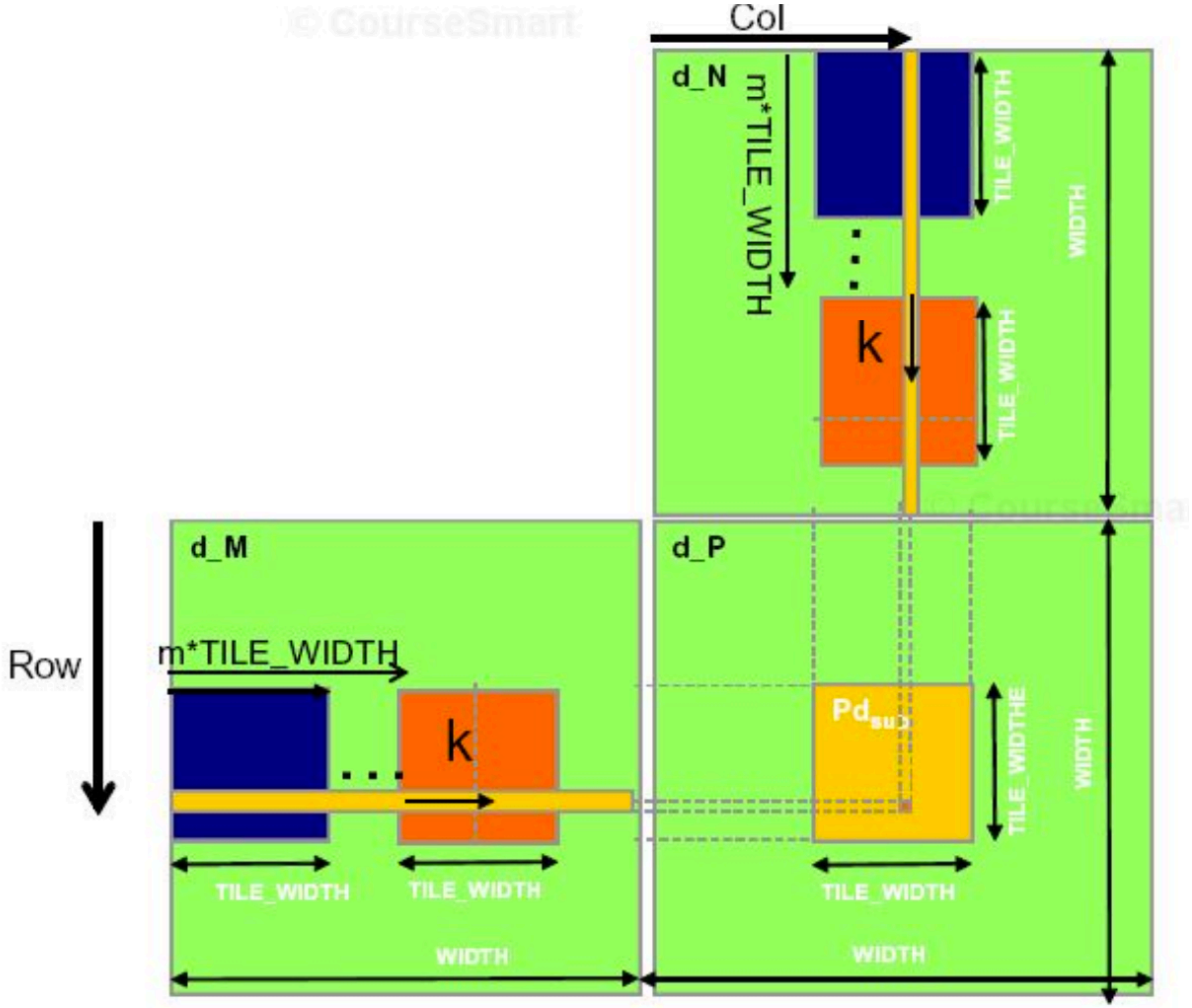
FIGURE 5.11

Execution phases of a tiled matrix multiplication.

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

      __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
      __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

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

      // Identify the row and column of the d_P element to work on
      int Row = by * TILE_WIDTH + ty;
      int Col = bx * TILE_WIDTH + tx;

      float Pvalue = 0;
      // Loop over the d_M and d_N tiles required to compute d_P element
      for (int m = 0; m < Width/TILE_WIDTH; ++m) {

        // Coolaborative loading of d_M and d_N tiles into shared memory
        Mds[ty][tx] = d_M[Row*Width + m*TILE_WIDTH + tx];
        Nds[ty][tx] = d_N[(m*TILE_WIDTH + ty)*Width + Col];
        __syncthreads();

        for (int k = 0; k < TILE_WIDTH; ++k) {
          Pvalue += Mds[ty][k] * Nds[k][tx];
        }
        __syncthreads();
      }
      d_P[Row*Width + Col] = Pvalue;
    }
```

FIGURE 5.12

Tiled matrix multiplication kernel using shared memory.


Col
d_N
m*TILE_WIDTH
TILE_WIDTH
WIDTH
k
TILE_WIDTH
d_M
d_P
Row
m*TILE_WIDTH
k
Pd_sub
TILE_WIDTHE
WIDTH
TILE_WIDTH
TILE_WIDTH
TILE_WIDTH
WIDTH
WIDTH

Analysis

- If we use TILE_WIDTH=16, also the block size is 16 X 16.
- We can reduce the global memory accesses by a factor of 16.
  - Increase the CGMA from 1 to 16.
  - Improve throughput of the Cuda device.

Analysis

- If we use TILE_WIDTH=16, also the block size is 16 X 16.
- We can reduce the global memory accesses by a factor of 16. How to calculate?
  - In figure 5.13, focus on the tile Pd_{sub} in d_P,
  - For simple Matrix-Matrix Multiplication Kernel, total global access C1= (WIDTH * TILE_WIDTH) * TILE_WIDTH *2
  - For tiled Matrxi-Matrix Multipliction Kernel, total global mem access C2 = (WIDTH * TILE_WIDTH) * 2
  - C1 / C2 = TILE_WIDTH

Analysis

- If our device has a 200GB/s global memory bandwidth, with CGMA = 16,
- Now it supports (200 / 4) * 16 = 800 GFLOPS.
  - Close to its peak performance.

Wrap up

1, Strategy to reduce global memory traffic

2, Tiled Matrix-matrix multiplication kernel

3, Analyzing how much global memory access is reduced for the new kernel?

Next Class

1, Memory as a limiting factor to parallelism

2, Start chapter 6, Warp and Thread Execution