CS6068 Parallel Computing Lect 4 – Sept 22, 2015

Topics:

Udacity Unit 5

Optimization - Achieving Terascale

CUDA Memory Restrictions

Using Shared Memory

Running Matrix Mult Example

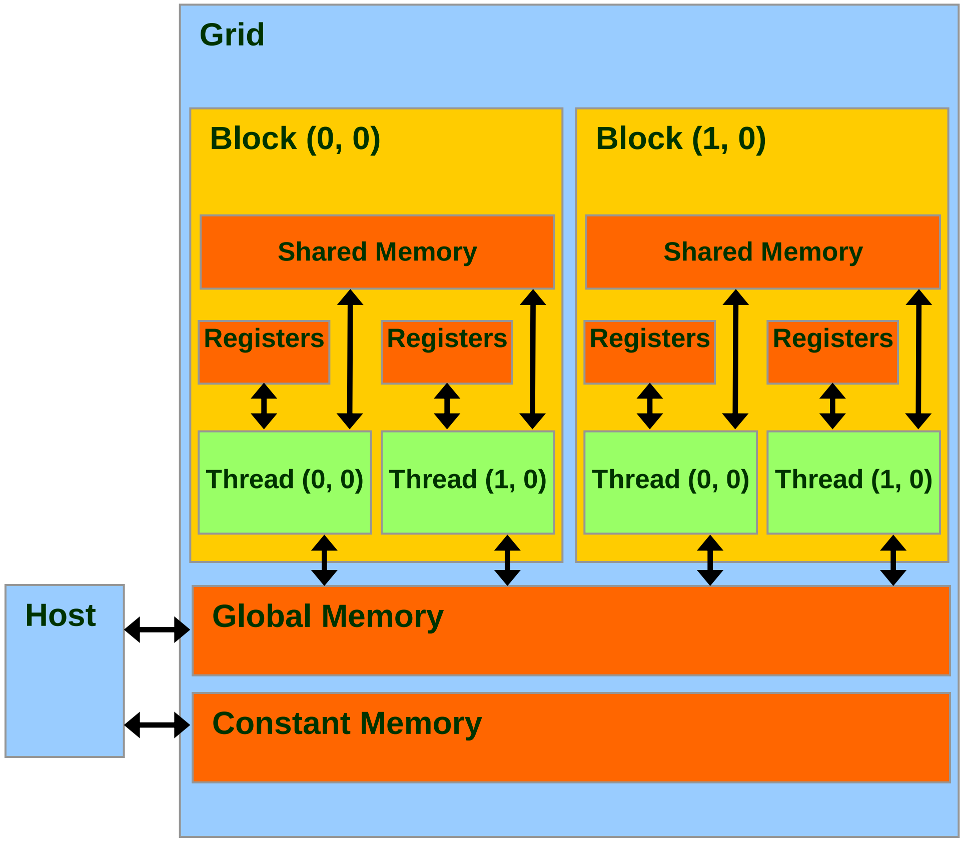
Performance Evaluation

Important Notions for Optimization

- First improve algorithm! REVIEW
- Second Look at Memory
- Most tuned GPU codes are Memory Limited.
- Device Query: Memory Clock Rate, Memory Bus Width → Theo Peak Bandwidth
- May use Profiler (like NVPP) to dig into potential Bottlenecks
- Memory Hierarchy, Coalescing, Ninja

Variety of CUDA Memories

- Each thread can:
  - Read/write per-thread registers
  - Read/write per-thread local memory
  - Read/write per-block shared memory
  - Read/write per-grid global memory
  - Read/only per-grid constant memory
  - Read/only per-grid texture memory



A Common Optimization: Tiling Data for Shared Memory

- Global memory resides in device memory (DRAM) – results in much slower access than shared memory
- Tiling data takes advantage of faster shared memory:
  - Partition data into subsets that fit into shared memory
  - Handle each data subset with one thread block by:
    - Load from global memory to shared memory,
    - Perform several computations
    - Copy results from shared memory to global memory

Another Common Optimization: Constant Memory

- Constant memory also resides in device DRAM - much slower than shared memory But…cached!
  - Highly efficient access for read-only data
- Carefully divide data according to access patterns
  - R/Only → constant memory (very fast if in cache)
  - R/W shared within Block → shared memory (very fast)
  - R/W within each thread → registers (very fast)
  - R/W inputs/results → global memory (very slow)

Optimization of Dot Products

Start design with relevant parameters

```
#define NumSMPs 32    #Machine dependent
#define N 33 * 8192       #problem size
#define ThreadsperBlock   256
#define BlockspergGrid NumSMPs
```

```
__global__ void dot( int*a, int*b, int*c ) {
// Shared memory for results of multiplication

__shared__ int temp[N];
temp[threadIdx.x] = a[threadIdx.x]*b[threadIdx.x];

// Thread 0 sums the pairwise products

if( 0 == threadIdx.x ) {
    intsum = 0;
    for( int i= 0; i< N; i++ )
        sum += temp[i];
    *c = sum;
}}
```

Let’s Deal with case where N is bigger than allowable number of threads (ThreadsperBlock * BlockspergGrid) = blockDim.x * gridDim.x

Our attempt begins with shared cache and initializing tid as offset into the N-item array.

```
__global__ void dot( float *a, float *b, float *c ) {
__shared__ float cache[ThreadsPerBlock];
int tid = threadIdx.x + blockIdx.x * blockDim.x;
int cacheIdx = threadIdx.x;
float temp = 0;
while (tid < N) {
   temp += a[tid] * b[tid];
   tid += blockDim.x * gridDim.x;
 }
cache[cacheIdx] = temp; }
```

Still need to reduce each block, and reduce across blocks

```
//add pairs of values offset by 2-powers
// recursive halving

__shared__ float cache[ThreadsPerBlock];
int cacheIdx = threadIdx.x;
int offset = blockDim.x/2;
while (offset != 0) {
     if (cacheIdx < offset){
          cache[cacheIdx] += cache[cacheIdx +
offset];}
    __syncthreads();
    offset /= 2; }
if (cacheIdx == 0)
c[blockIdx.x] = cache[0];
}
```

Matrix Multiplication Optimization

GOAL: ACHIEVING TERASCALE for Matrix Multiply

Assume that all threads access global memory for their input matrix elements

– Two memory accesses (8 bytes)
per floating point multiply-add
– 4B/s of memory bandwidth/
FLOPS
– Tesla has 1030 GFLOP peak
– 4*1030 = 4120 GB/s required to achieve peak FLOP rating
– 148 GB/s on global memory limits
the code at 37 GFLOPS
• Need to drastically cut down global memory accesses to get closer to the terascale peak 1030 GFLOPS

```
// Matrix multiply on host in double
precision
void MatrixMulOnHost(float* M, float* N,
float* P, int Width)
{
for (int i = 0; i < Width; ++i)
   for (int j = 0; j < Width; ++j) {
      double sum = 0;
      for (int k = 0; k < Width; ++k) {
        double a = M[i * width + k];
        double b = N[k * width + j];
         sum += a * b;
}
P[i * Width + j] = sum;
}
```

```
void MatrixMulOnDevice(float* M, float* N, float* P, int
Width)
{
int size = Width * Width * sizeof(float);
float* Md, Nd, Pd;
// Allocate and Load M, N to device memory
cudaMalloc(&Md, size);
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
cudaMalloc(&Nd, size);
cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);
// Allocate P on the device
cudaMalloc(&Pd, size);

// Kernel invocation code – to be shown later
…
// Read P from the device
cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
// Free device matrices
cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
}
```

MM: Kernel Invocation with 1 Block

```
   // Setup the execution configuration
   #define WIDTH  8192
   dim3 dimGrid(1, 1);
   dim3 dimBlock(WIDTH, WIDTH);
   // Launch the device computation threads!
   // 1x1 block with 8192x8192 threads
   MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, 
Pd,WIDTH);
--- MP Information for device 4 ---
Multiprocessor count: 30
Shared mem per mp: 16384
Registers per mp: 16384
Threads in warp: 32
Max threads per block: 512
Max thread dimensions: (512, 512, 64)
Max grid dimensions: (65535, 65535, 1)
```

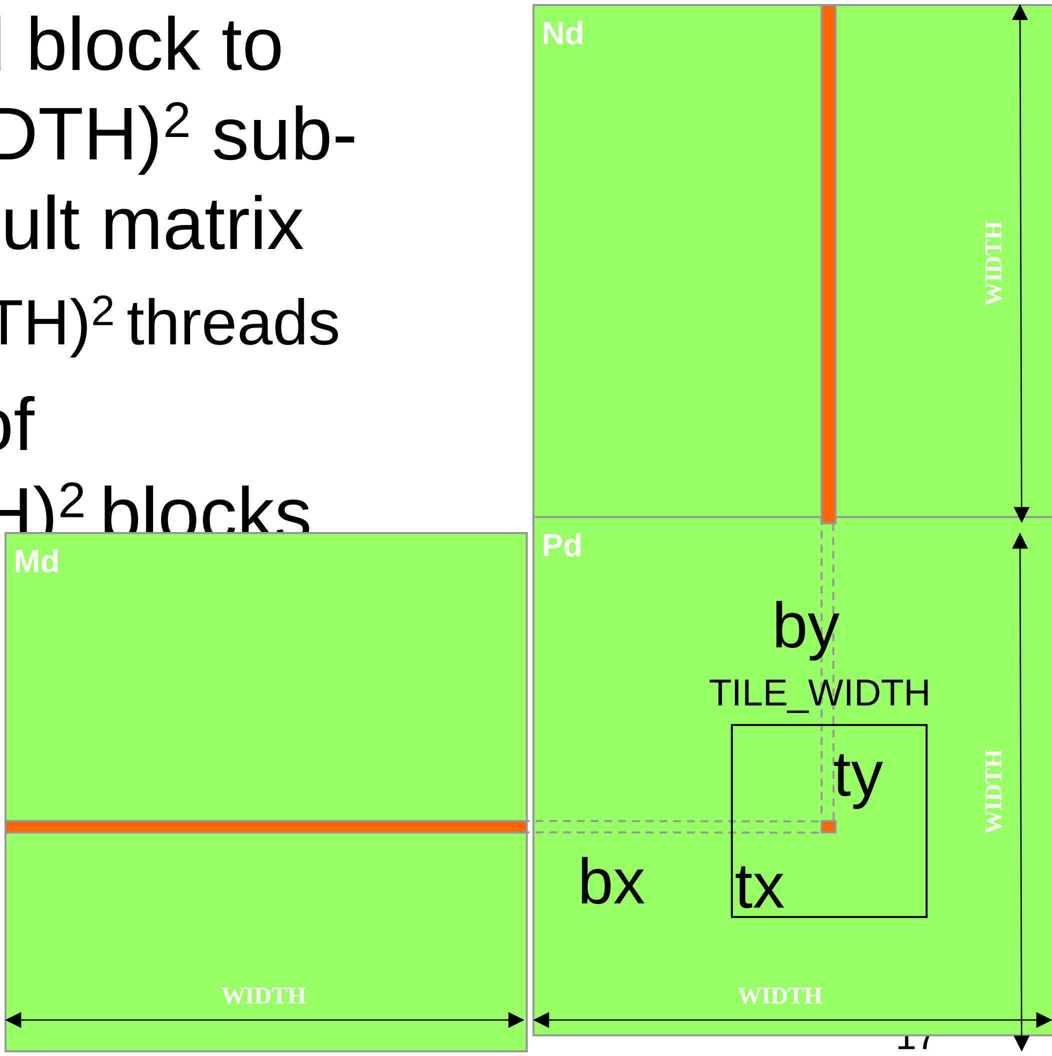
MM: Kernel Invocation

```
   // Setup the execution configuration
   #define WIDTH  8192
   #define TILE_WIDTH 16

   dim3 dimGrid(WIDTH/TILE_WIDTH, WIDTH/TILE_WIDTH);
   dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);

   // Launch the device computation threads!
   // 512x512 blocks with 16x16 threads per block
   MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd,
Pd,WIDTH);
```

Extend to Arbitrary Sized Square Matrices

- Have each 2D thread block to compute a $(TILE\_WIDTH)^2$ sub-matrix (tile) of the result matrix
  - Each has $(TILE\_WIDTH)^2$ threads
- Generate a 2D Grid of $(WIDTH/TILE\_WIDTH)^2$ blocks

You may still need to put a loop around the kernel call for cases where WIDTH/TILE_WIDTH is greater than max grid size (64K)!

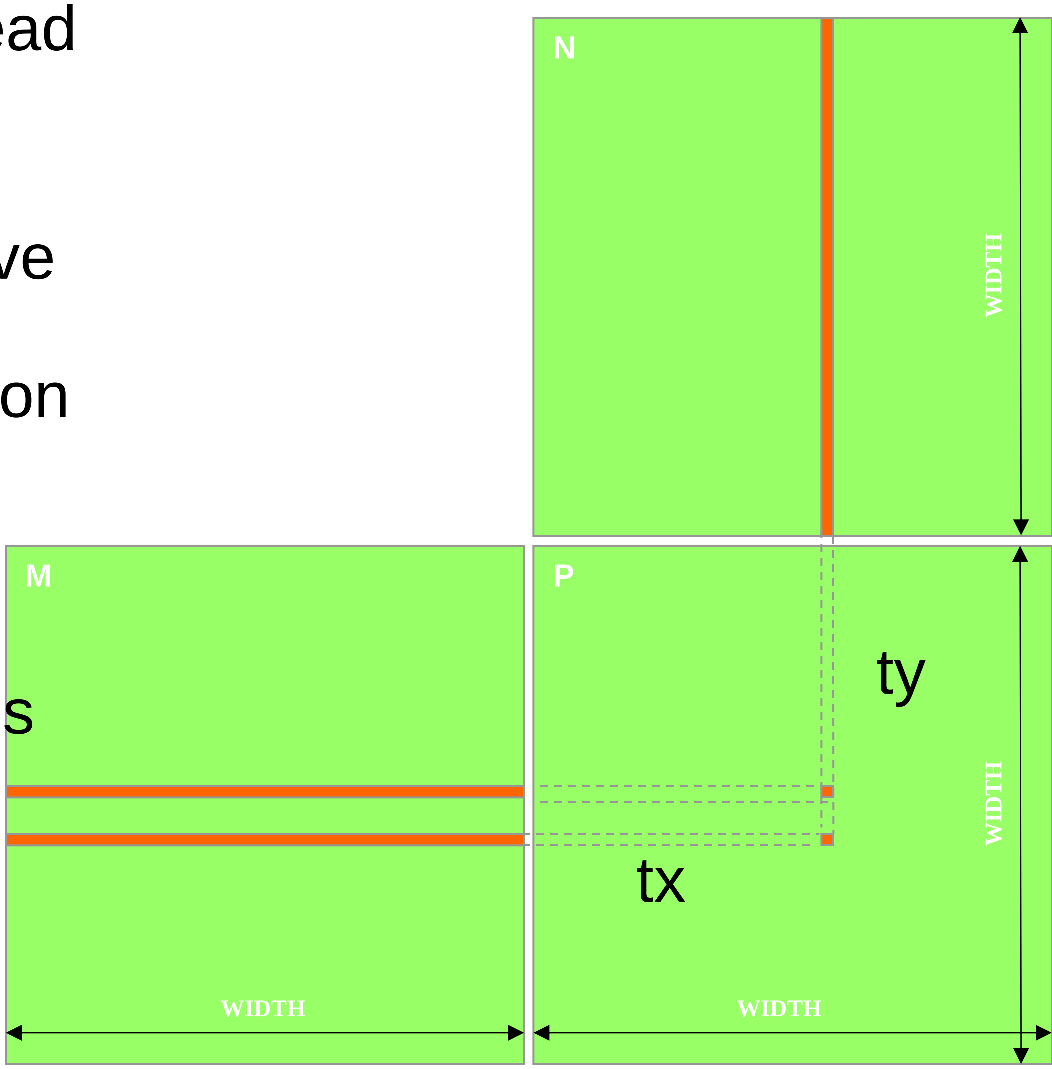


Matrix Multiplication Kernel using Multiple Blocks - Tiling

```
__global__ void MatrixMulKernel
           (float* Md, float* Nd, float* Pd, int Width)
{
// Calculate the row index of the Pd element and M
int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
// Calculate the column index of Pd and N
int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

float Pvalue = 0;
// each thread computes one element of the block sub-matrix
for (int k = 0; k < Width; ++k)
  Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];

Pd[Row*Width+Col] = Pvalue;
```

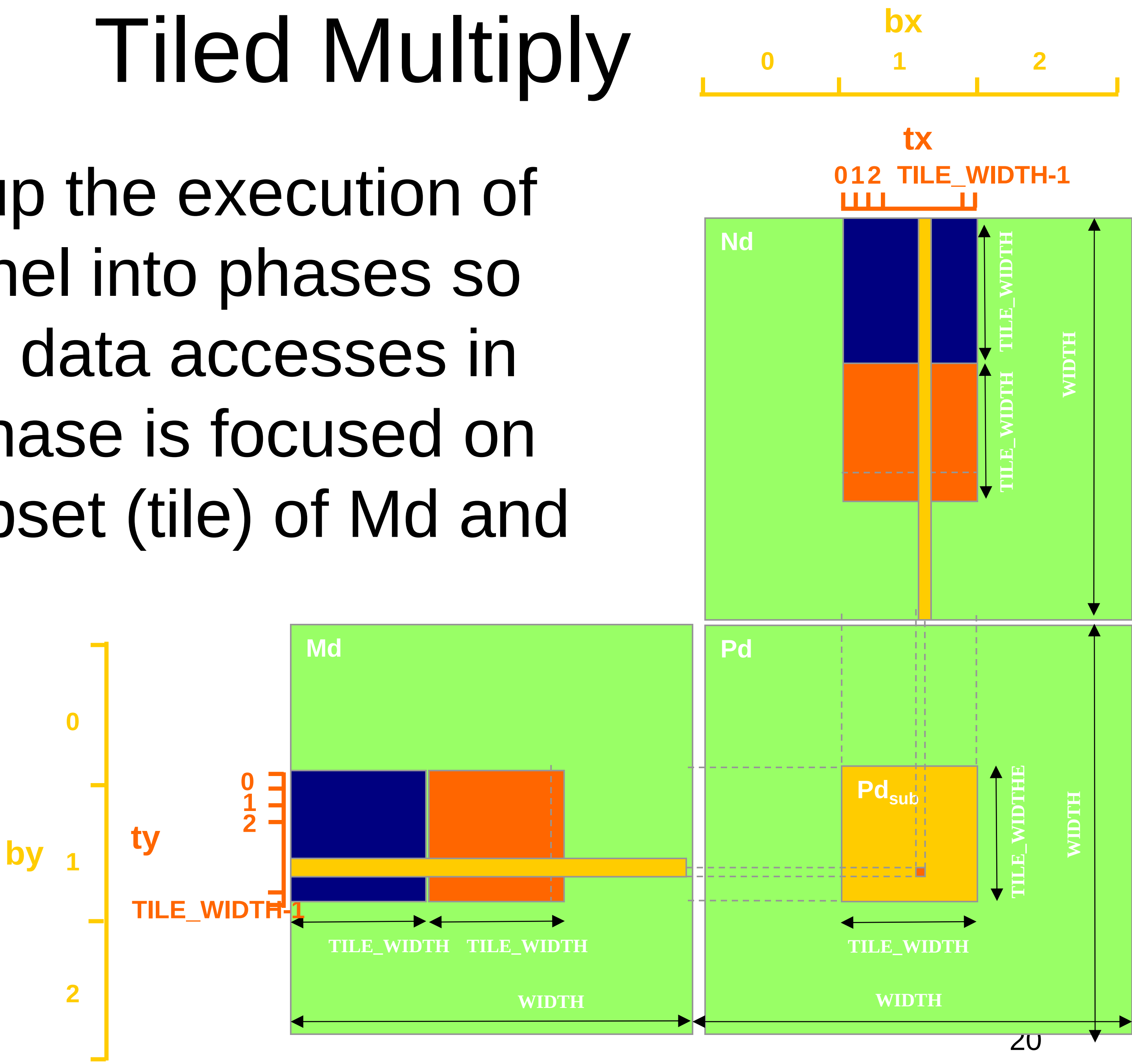
Idea: Reuse limited shared memory

- Each input element is read by Width threads.
- Load each element into Shared Memory and have several threads in same block use the local version to reduce the memory bandwidth
  - Tiled algorithms
- Try for one global access for each tile. All other access shared.

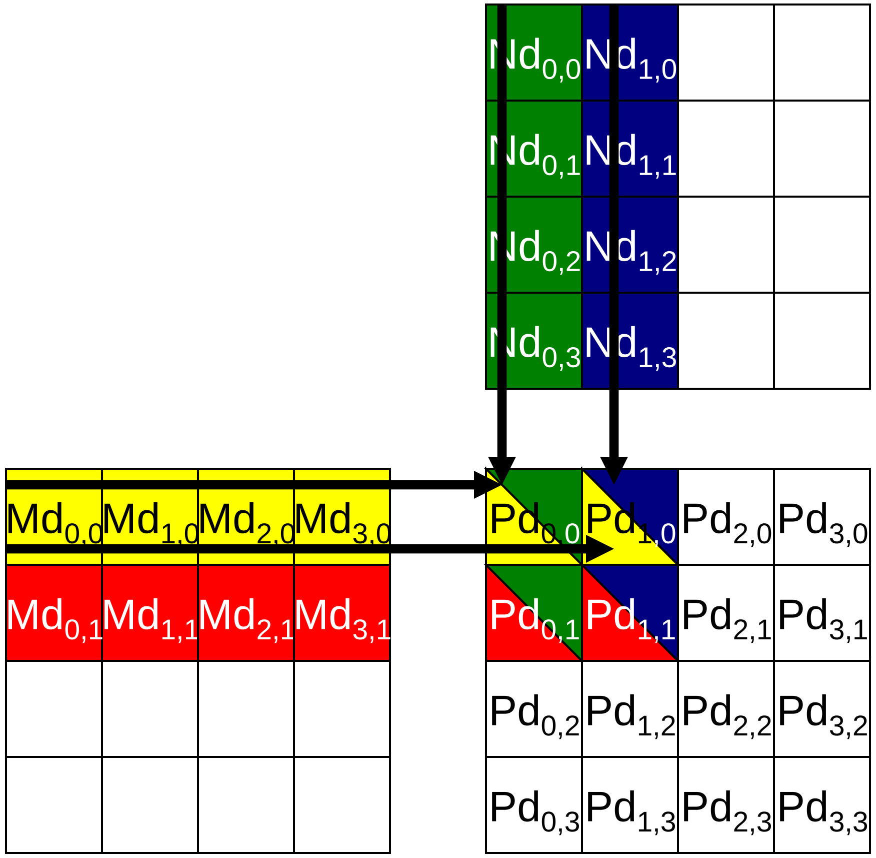


Tiled Multiply

- Break up the execution of the kernel into phases so that the data accesses in each phase is focused on one subset (tile) of Md and Nd

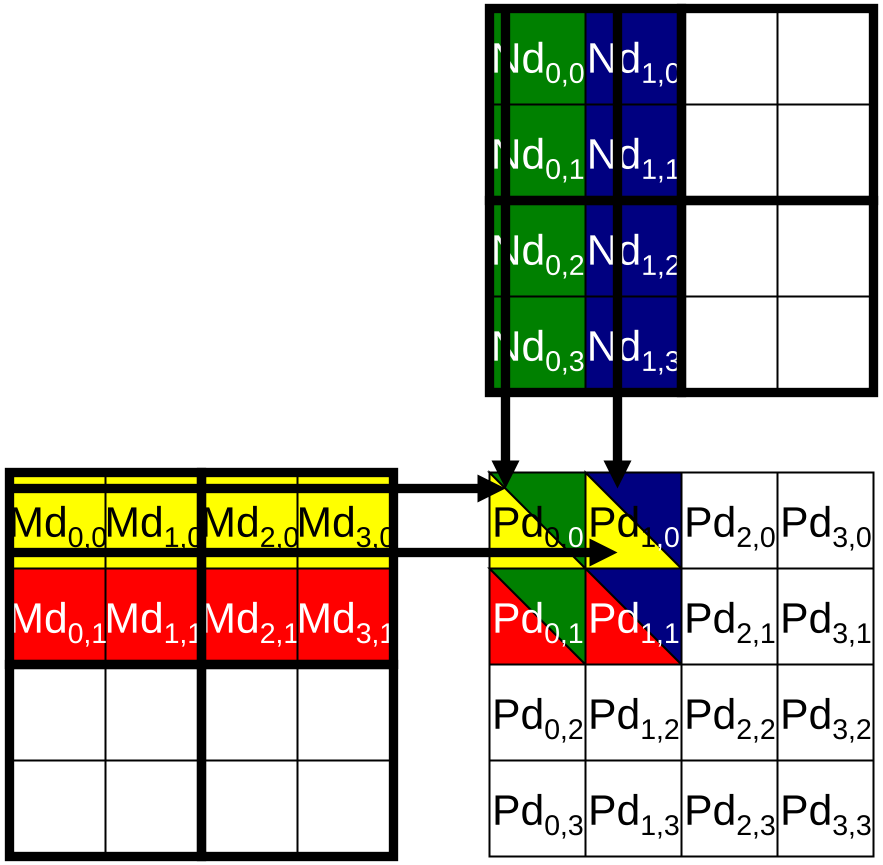


Illustrate Idea with Small Example- 4 blocks of 4 threads each



$Nd_{0,0}$ $Nd_{1,0}$
$Nd_{0,1}$ $Nd_{1,1}$
$Nd_{0,2}$ $Nd_{1,2}$
$Nd_{0,3}$ $Nd_{1,3}$
$Md_{0,0}$ $Md_{1,0}$ $Md_{2,0}$ $Md_{3,0}$
$Md_{0,1}$ $Md_{1,1}$ $Md_{2,1}$ $Md_{3,1}$
$Pd_{0,0}$ $Pd_{1,0}$ $Pd_{2,0}$ $Pd_{3,0}$
$Pd_{0,1}$ $Pd_{1,1}$ $Pd_{2,1}$ $Pd_{3,1}$
$Pd_{0,2}$ $Pd_{1,2}$ $Pd_{2,2}$ $Pd_{3,2}$
$Pd_{0,3}$ $Pd_{1,3}$ $Pd_{2,3}$ $Pd_{3,3}$

Breaking Md and Nd into Tiles



$Nd_{0,0}$ $Nd_{1,0}$
$Nd_{0,1}$ $Nd_{1,1}$
$Nd_{0,2}$ $Nd_{1,2}$
$Nd_{0,3}$ $Nd_{1,3}$
$Md_{0,0}$ $Md_{1,0}$ $Md_{2,0}$ $Md_{3,0}$
$Md_{0,1}$ $Md_{1,1}$ $Md_{2,1}$ $Md_{3,1}$
$Pd_{0,0}$ $Pd_{1,0}$ $Pd_{2,0}$ $Pd_{3,0}$
$Pd_{0,1}$ $Pd_{1,1}$ $Pd_{2,1}$ $Pd_{3,1}$
$Pd_{0,2}$ $Pd_{1,2}$ $Pd_{2,2}$ $Pd_{3,2}$
$Pd_{0,3}$ $Pd_{1,3}$ $Pd_{2,3}$ $Pd_{3,3}$

22

Size Considerations in CUDA

- Each thread block should have many threads
  - TILE_WIDTH of 16 gives 16*16 = 256 threads
- There should be many thread blocks
  - A 1024*1024 Pd gives 64*64 = 4096 Thread Blocks
- Each thread block perform 2*256 = 512 float loads from global memory for 256 * (2*16) = 8,192 mul/add operations.
  - 16 Flops per load
  - Global Memory bandwidth no longer a limiting factor

Reference Code: Tiled Matrix Multiplication Kernel

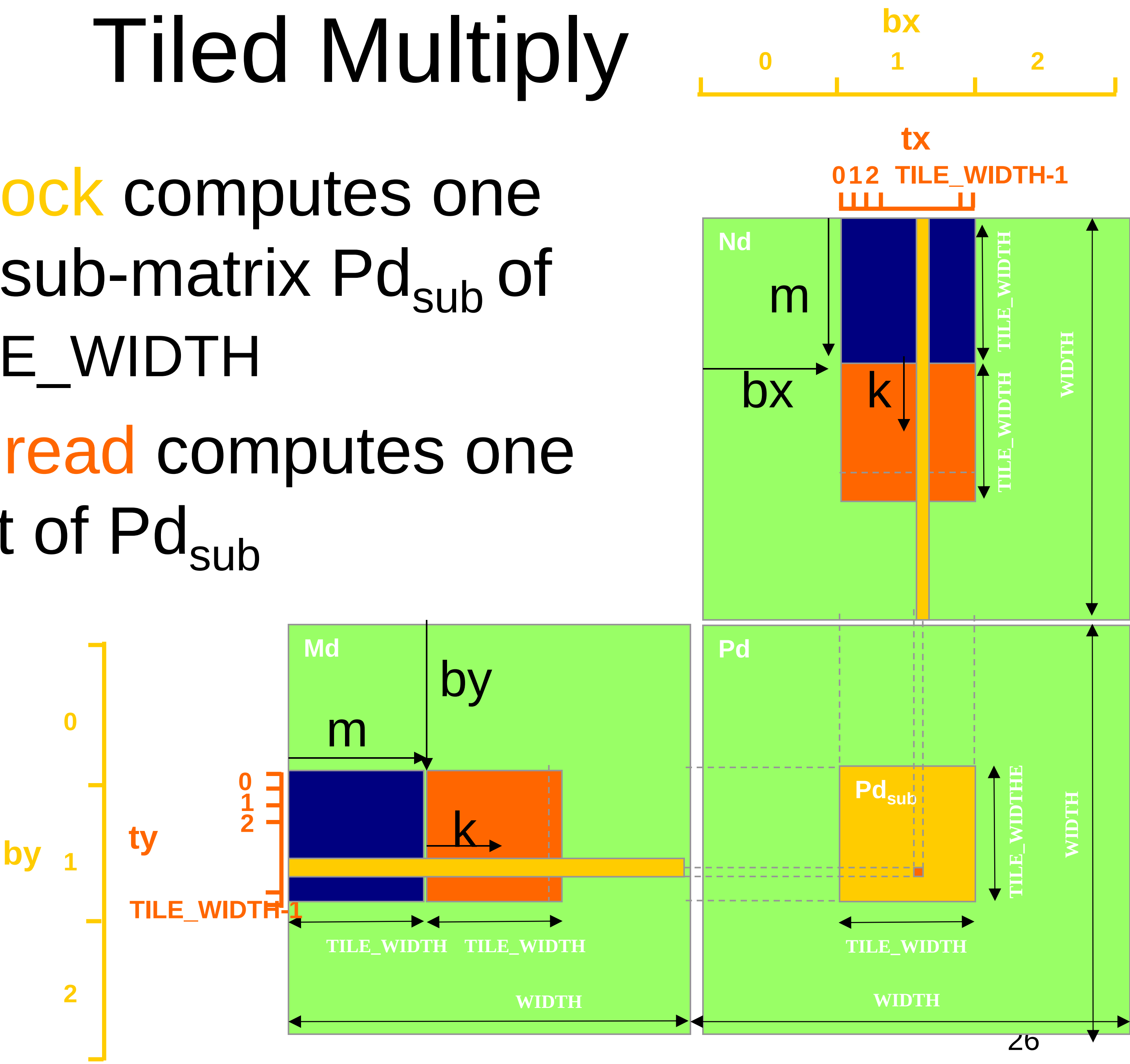
```
__global__ void MatrixMulKernel(float* Md, float* Nd,
    float* Pd, int Width)
{
1.   __shared__float Mds[TILE_WIDTH][TILE_WIDTH];
2.   __shared__float Nds[TILE_WIDTH][TILE_WIDTH];

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 Pd element to
    work on
5.   int Row = by * TILE_WIDTH + ty;
6.   int Col = bx * TILE_WIDTH + tx;
```

Reference Code: Tiled Matrix Multiplication Kernel

```
float Pvalue = 0;
// Loop over the Md and Nd tiles required to compute
    the Pd element
7.    for (int m = 0; m < Width/TILE_WIDTH; ++m) {
// Collaborative loading of Md and Nd tiles into shared
    memory
8. Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
   Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
   __syncthreads();

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

Tiled Multiply

- Each block computes one square sub-matrix Pd_{sub} of size TILE_WIDTH
- Each thread computes one element of Pd_{sub}



CUDA Shared Memory and Latency Hiding

- Each SM has at least 16KB shared memory
  - SM size is implementation dependent!
  - For TILE_WIDTH = 16, each thread block uses 2*256*4B = 2KB of shared memory.
  - Can potentially have at least 8 Thread Blocks actively 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 shared memory usage per thread block, allowing only up to two thread blocks active at the same time
- Using 16x16 tiling, we reduce the accesses to the global memory by a factor of 16
  - The 148 B/s bandwidth can now support (148/4)*16 = 592 GFLOPS!

Ninja Optimizations:

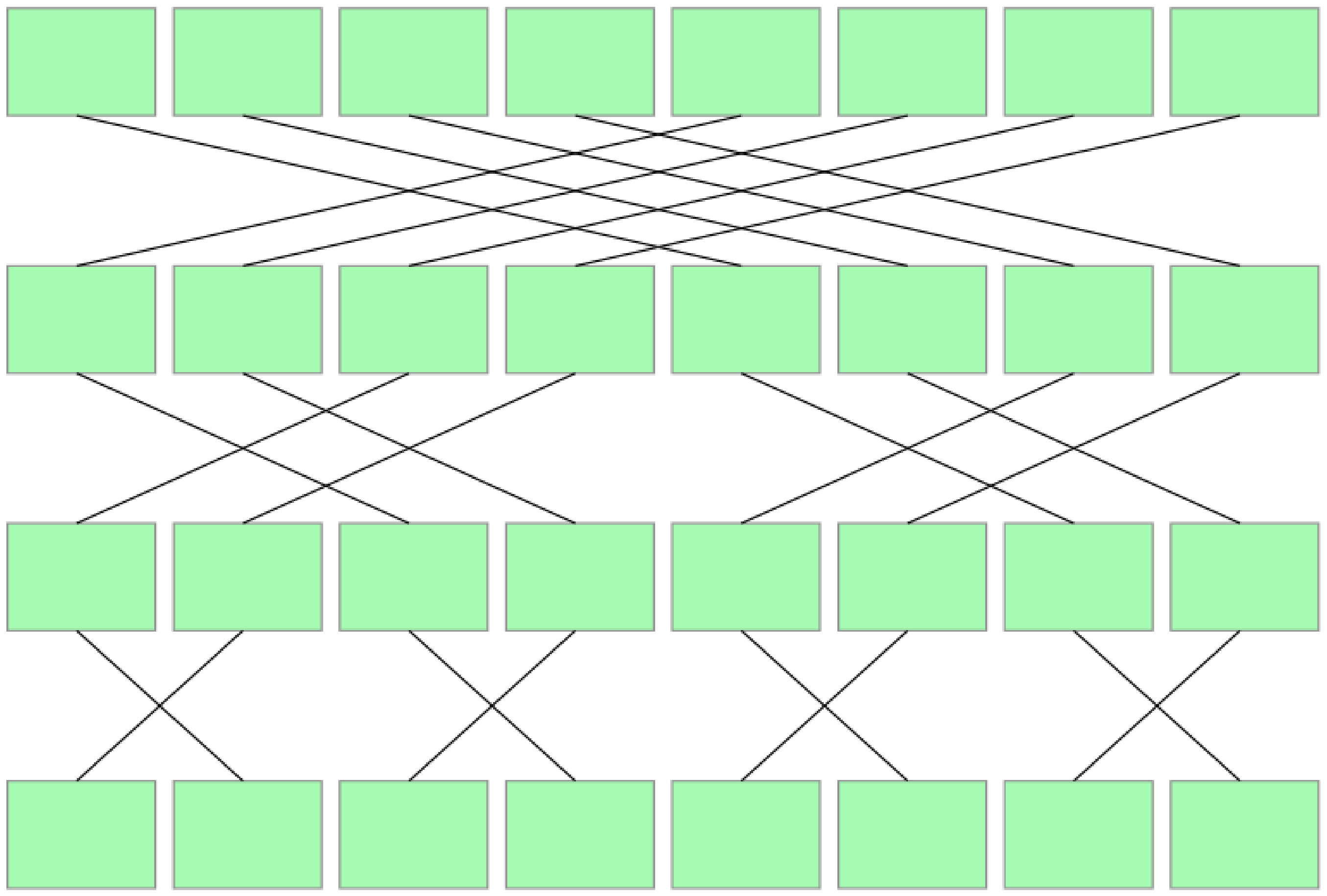
- WARP Control Flow and Divergence
- Bank Conflicts

Important:

CUDA runs smoother when consecutively ordered threads are doing same operation

- Consider the problem of getting all threads in a block to the same state, such as summing all items in cache array. Doing a reduction followed by broadcast creates divergence.

- Each level of the butterfly links indices that differ in single *bit* position



```
__shared__ int bfly[ThreadsperBlock];
int cacheIdx = threadIdx.x;

bfly[cacheIdx] = cache[cacheIdx] ;
__syncthreads();
for(int bit=blocksize/2; bit>0; bit/=2)
{
 int tmp = bfly[cacheIdx]+ bfly[cacheIdx^bit];
  __syncthreads();
 bfly[cacheIdx]=tmp;
  __syncthreads();
}
// Every bfly item now holds sum of all cache
```