Matrix Multiplication in CUDA

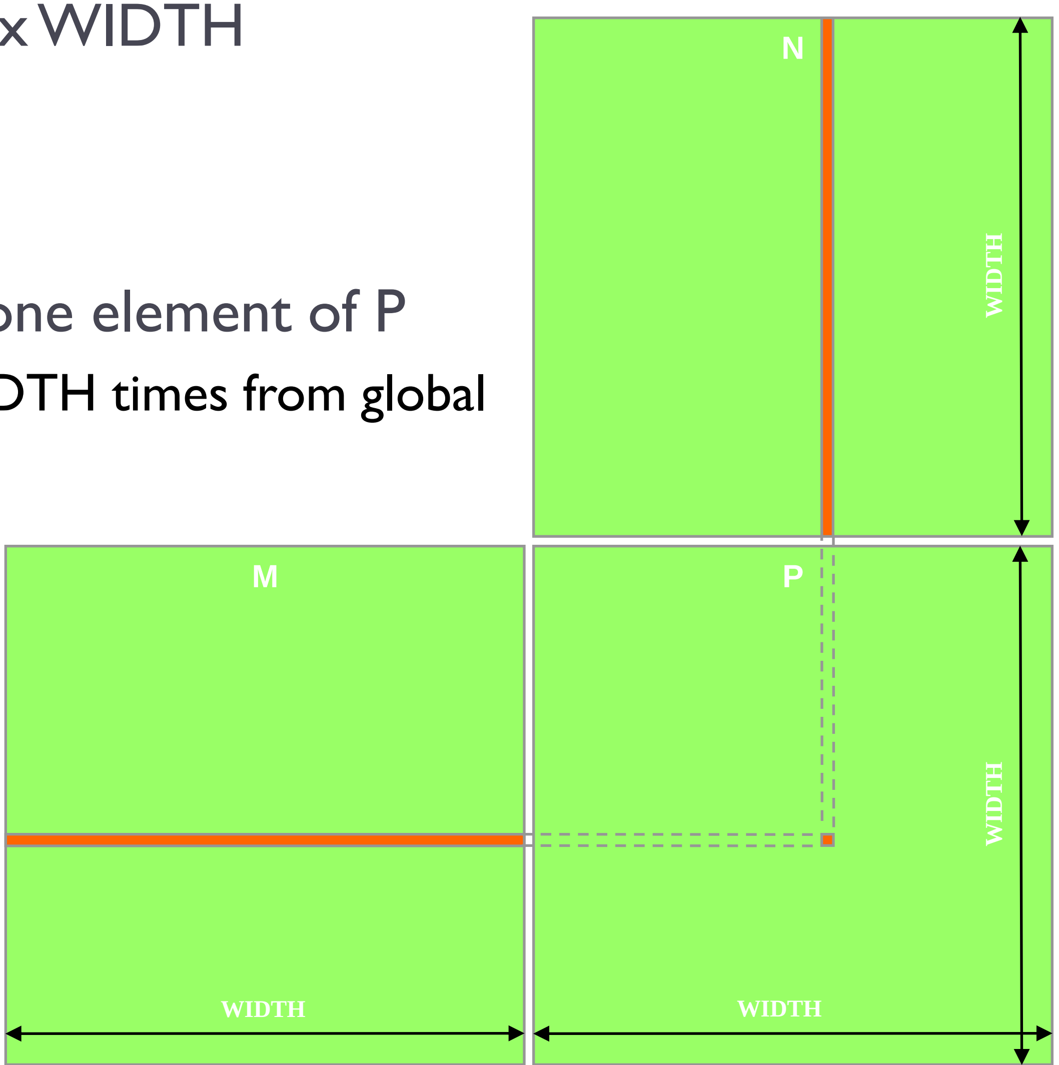
A case study

Matrix Multiplication: A Case Study

- Matrix multiplication illustrates many of the basic features of memory and thread management in CUDA
  - Usage of thread/block IDs
  - Memory data transfer between host and device
  - Motivates some performance issues:
    - shared memory usage
    - register usage
  - Assumptions:
    - Basic unoptimized sgemm
    - Matrices are square (for simplicity)

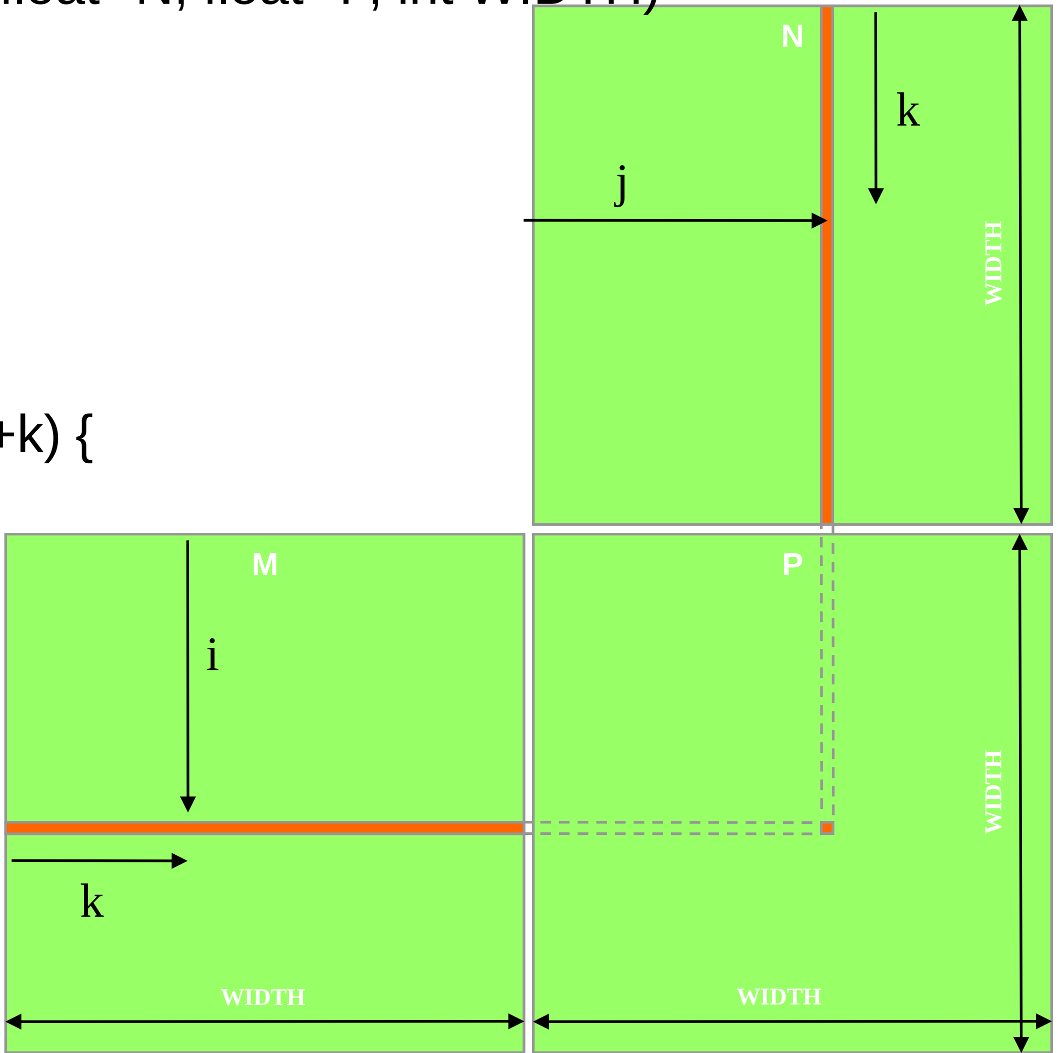
Programming Model: Square Matrix Multiplication Example

- P = M * N
  - Each is of size WIDTH x WIDTH
- Basic Idea:
  - One thread calculates one element of P
    - M and N are loaded WIDTH times from global memory



Step 1: Matrix Multiplication
A Simple Host Version in C

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



Step 2: Input Matrix Data Transfer (Host-side Code)

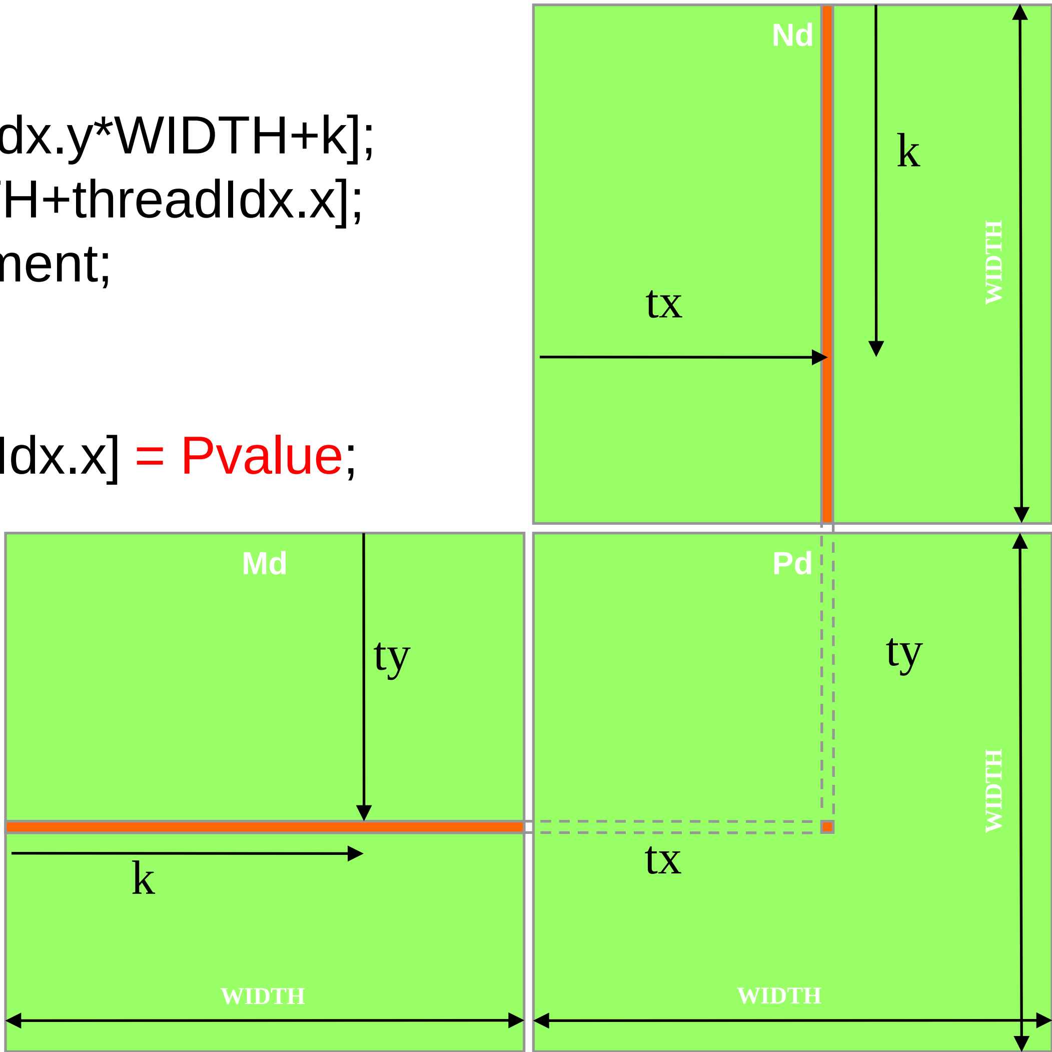
```
void MatrixMulOnDevice(float* M, float* N, float* P, int WIDTH)
{
    int size = WIDTH * WIDTH * sizeof(float);
    float* Md, Nd, Pd;
   …
    // 1. Allocate and Load M, N to device memory
    cudaMalloc(&Md, size);
    cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);

     cudaMalloc(&Nd, size);
     cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

     // Allocate P on the device
    cudaMalloc(&Pd, size);
```

Step 3: Output Matrix Data Transfer (Host-side Code)

```
 // 2. Kernel invocation code – to be shown later
…

 // 3. Read P from the device
 cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);

  // Free device matrices
 cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
}
```

Step 4: Kernel Function

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int WIDTH)
{
    float Pvalue = 0;

    for (int k = 0; k < WIDTH; ++k) {
        float Melement = Md[threadIdx.y*WIDTH+k];
        float Nelement = Nd[k*WIDTH+threadIdx.x];
        Pvalue += Melement * Nelement;
    }

    Pd[threadIdx.y*WIDTH+threadIdx.x] = Pvalue;
}
```



Step 5: Kernel Invocation (Host-side Code)

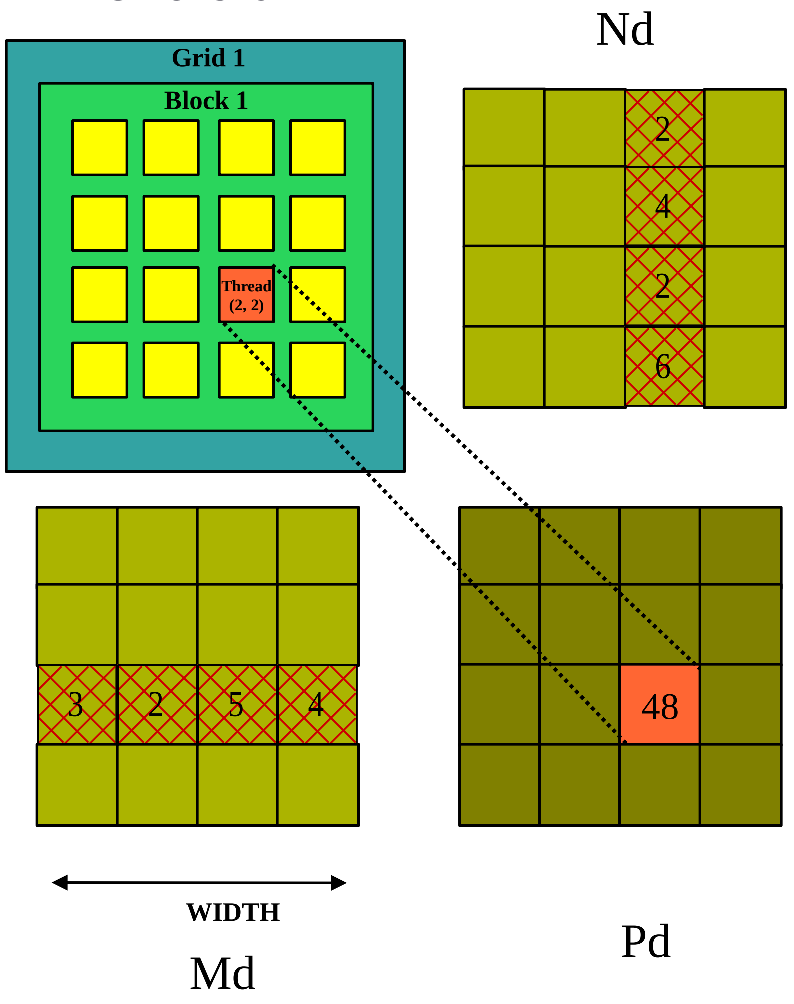
```
// Setup the execution configuration
dim3 dimGrid(1, 1);
dim3 dimBlock(WIDTH, WIDTH);

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

Only One Thread Block Used

- One Block of threads compute the matrix Pd
  - Each thread computes one element of the matrix Pd
- Each thread
  - Loads a row of matrix Md
  - Loads a column of matrix Nd
  - Perform one multiply and addition for each pair of Md and Nd elements

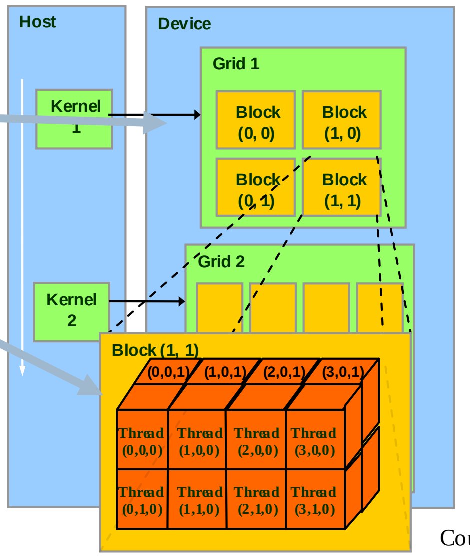
- Compute to off-chip memory access ratio close to 1:1 (not very good)

- Size of matrix limited by the number of threads allowed in a thread block (512)



Block IDs and Thread IDs

- Each thread uses IDs to decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …

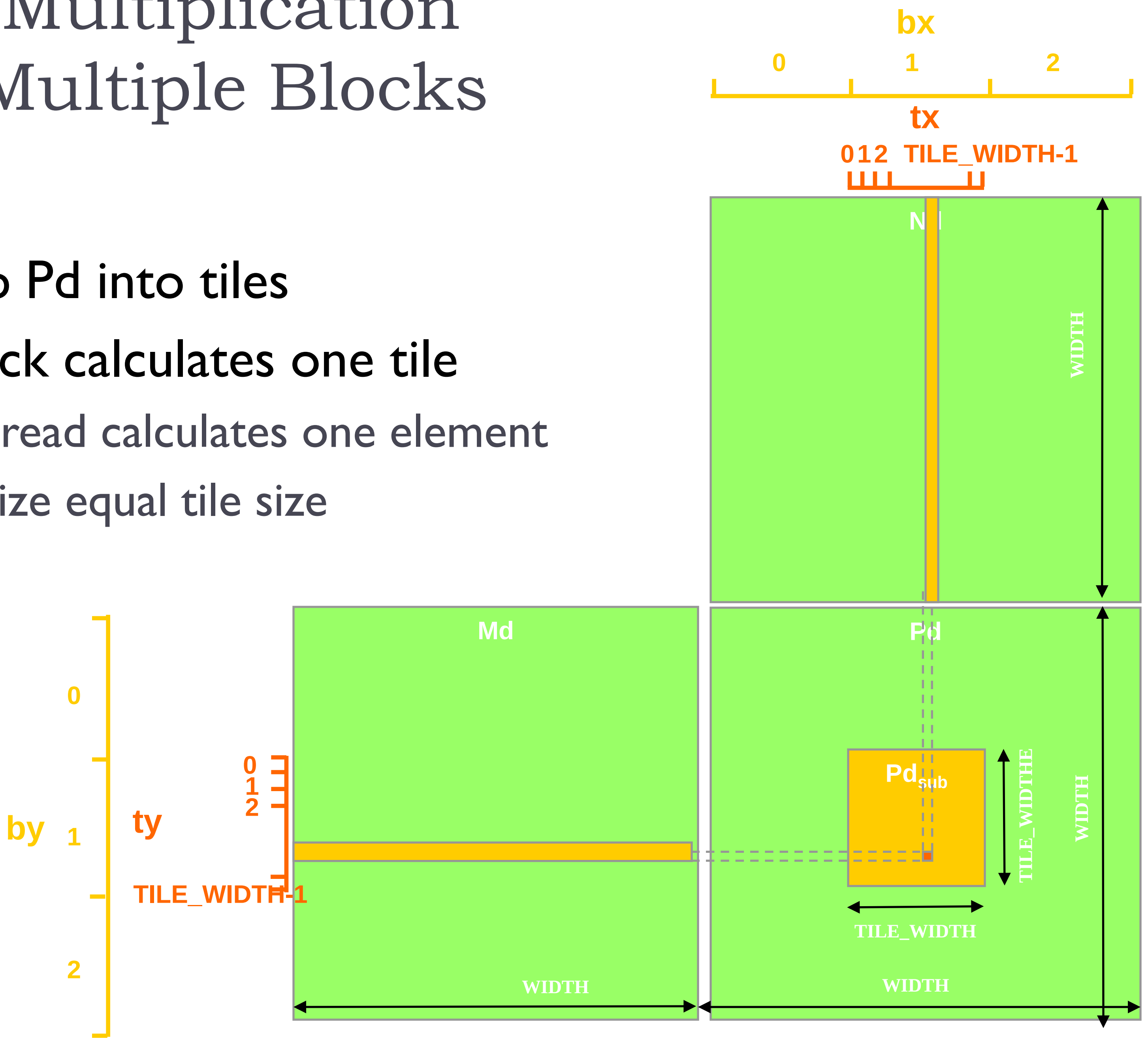


Courtesy: NDVIA

Figure 3.2. An Example of CUDA Thread Organization.

Matrix Multiplication Using Multiple Blocks

- Break-up Pd into tiles
- Each block calculates one tile
  - Each thread calculates one element
  - Block size equal tile size



Revised mmult Kernel using Multiple Blocks

```
__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 idenx 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;
}
```

G80 Block Granularity Considerations

Q: For Matrix Multiplication using multiple blocks, should I use 8x8, 16x16 or 32x32 blocks?

- For 8x8, we have 64 threads per Block. Since each SM can take up to 768 threads, there are 12 Blocks. However, each SM can only take up to 8 Blocks, only 512 threads will go into each SM!
- For 16x16, we have 256 threads per Block. Since each SM can take up to 768 threads, it can take up to 3 Blocks and achieve full capacity unless other resource considerations overrule.
- For 32x32, we have 1024 threads per Block. Not even one can fit into an SM!

Taking CUDA to Ludicrous Speed

Getting Righteous Performance from your GPU

Performance: How Much Is Enough? (CPU Edition)

- Could I be getting better performance?
  - Probably a little bit. Most of the performance is handled in HW
- How much better?
  - If you compile –O3, you can get faster (maybe 2x)
  - If you are careful about tiling your memory, you can get faster on codes that benefit from that (maybe 2-3x)
- Is that much performance worth the work?
  - Compiling with optimizations is a no-brainer (and yet…)
  - Tiling is useful, but takes an investment

Performance: How Much Is Enough? (GPGPU Edition)

- Could I be getting better performance?
  - Am I getting near peak GFLOP performance?
- How much better?
  - Brandon's particle code, using several different code modifications
    - 148ms per time step → 4ms per time step
- Is that much worth the work?
  - How much work would you do for 30-40x?
  - Most of the modifications are fairly straightforward
    - You just need to know how the hardware works a bit more

What's Limiting My Code?

- Am I bandwidth bound? (How do I tell?)
  - Make sure I have high thread occupancy to tolerate latencies (lots of threads)
    - These threads can get some work done while we wait for memory
  - Move re-used values to closer memories
    - Shared
    - Constant/Texture

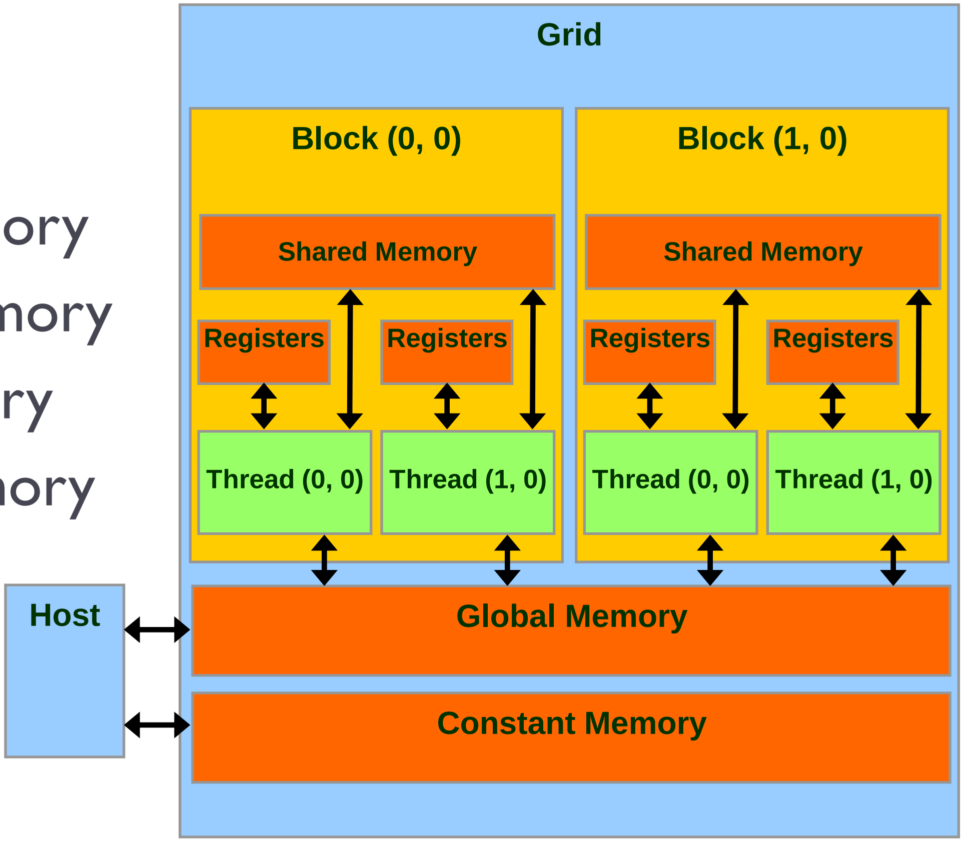
- Am I not bandwidth bound – what is now my limit?
  - Take a closer look at the instruction stream
    - Unroll loops
    - Minimize branch divergence

CUDA Memories

Locality Matters!

G80 Implementation 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



CUDA Variable Type Qualifiers

- __device__ is optional when used with __local__, __shared__, or __constant__

- Automatic variables without any qualifier reside in a register
  - Except arrays that reside in local memory

Variable declaration		Memory	Scope	Lifetime
`__device__ __local__`	`int LocalVar;`	local	thread	thread
`__device__ __shared__`	`int SharedVar;`	shared	block	block
`__device__`	`int GlobalVar;`	global	grid	application
`__device__ __constant__`	`int ConstantVar;`	constant	grid	application

A Common Programming Strategy

- Global memory resides in device memory (DRAM)
  - much slower access than shared memory (200x!)
  - …but also much larger
- So, a profitable way of performing computation on the device is to tile data to take advantage of fast shared memory:
  - Partition data into subsets that fit into shared memory
  - Each block will then:
    - Load its subset from global memory to shared memory
      - using multiple threads to exploit memory-level parallelism
    - Perform the computation on the subset from shared memory
      - each thread can efficiently multi-pass over any data element
    - Copy results from shared memory back to global memory

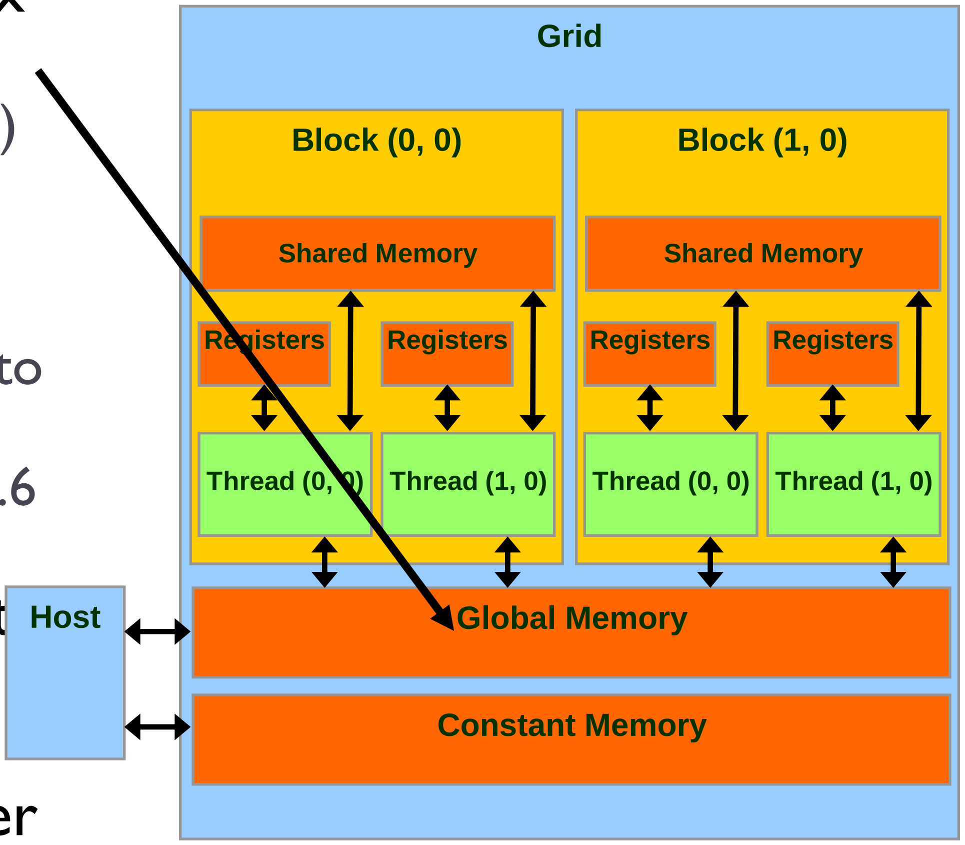
Matrix Multiplication using Shared Memory

Review

```
__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 idenx 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;
}
```

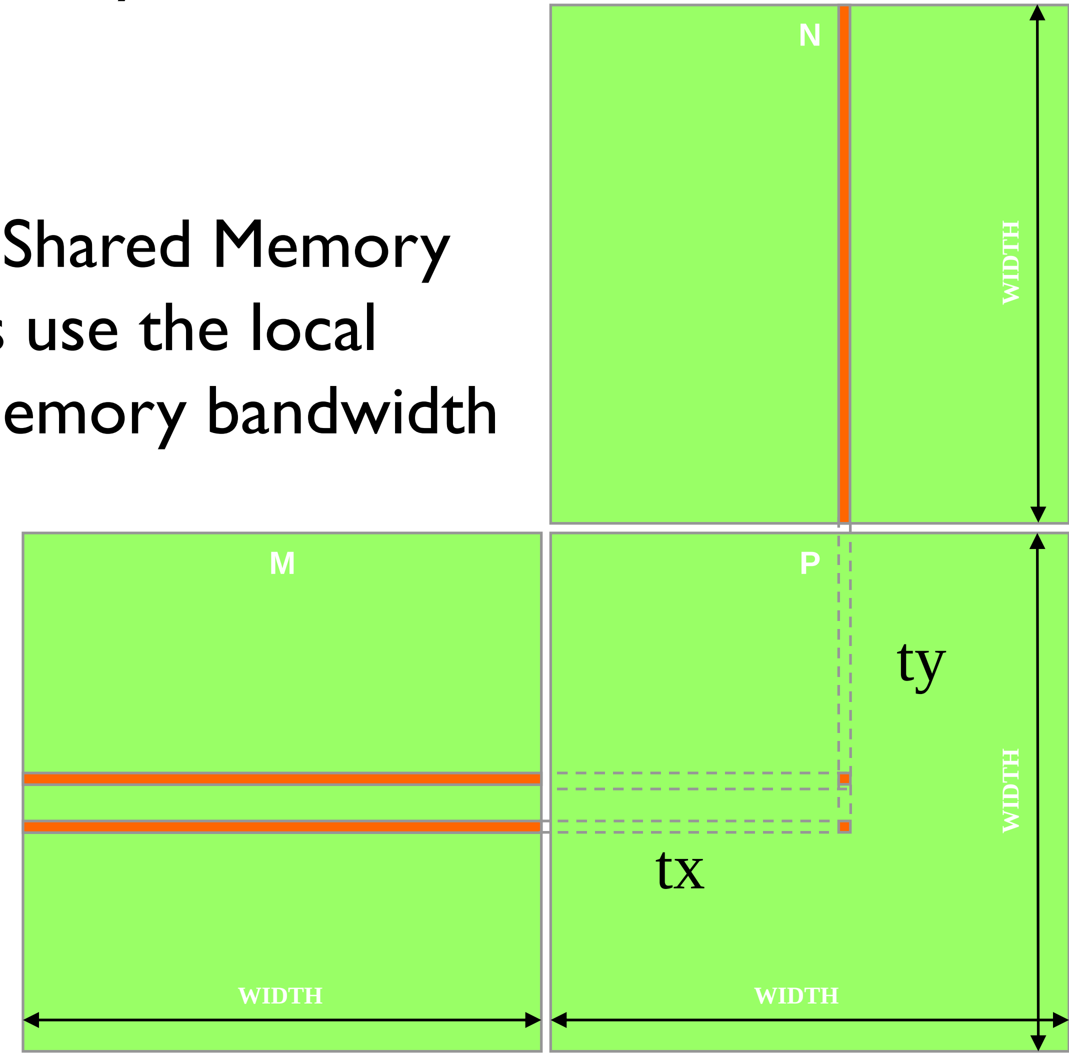
How about performance on G80?

- All threads access global memory for their input matrix elements
  - Two memory accesses (8 bytes) per floating point multiply-add
  - 4 B/s of memory bandwidth/FLOPS
  - 4*346.5 = 1386 GB/s required to achieve peak FLOP rating
  - 86.4 GB/s limits the code at 21.6 GFLOPS
- The actual code runs at about 15 GFLOPS
- Need to drastically cut down memory accesses to get closer to the peak 346.5 GFLOPS



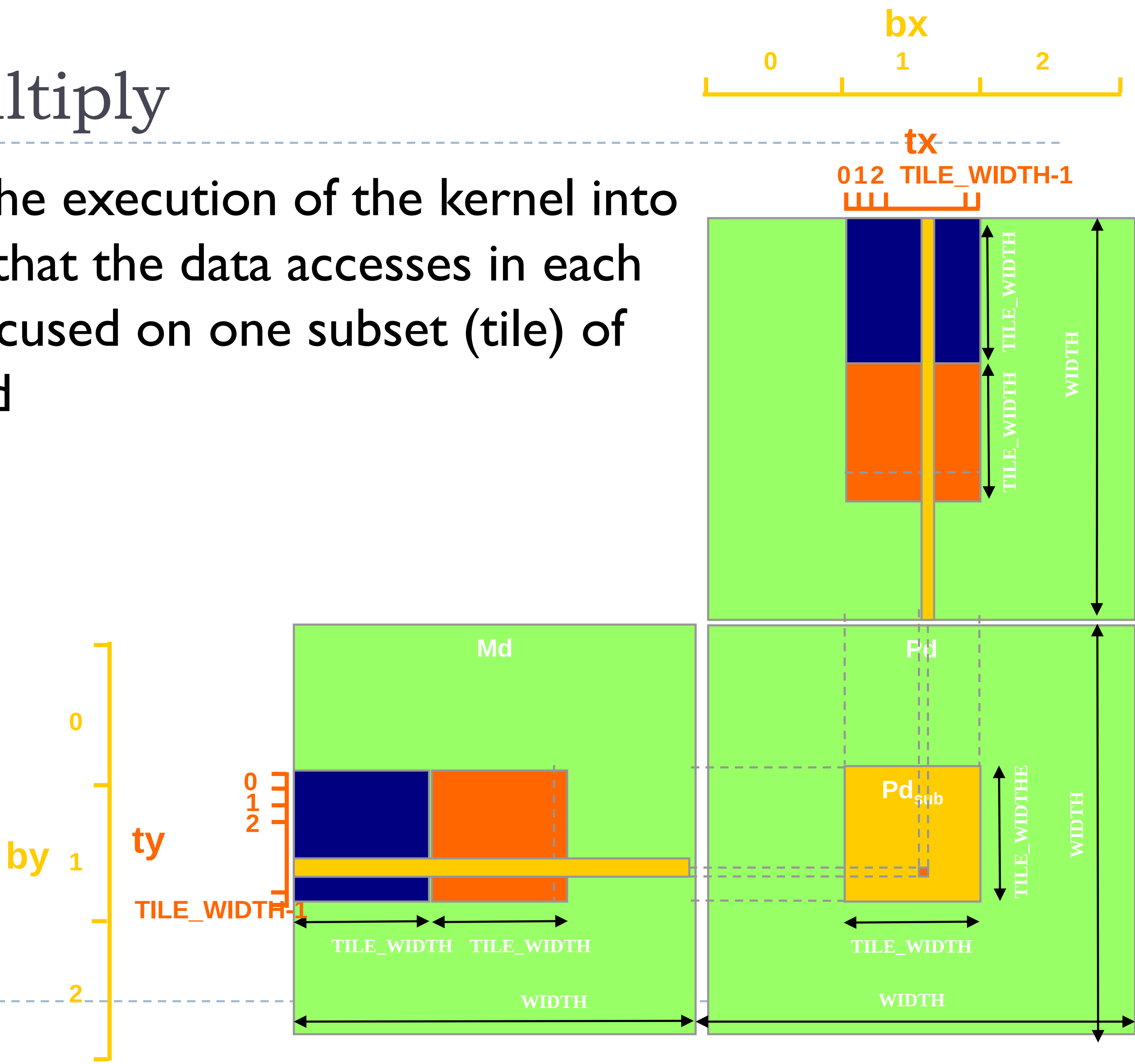
Idea: Use Shared Memory to reuse global memory data

- Each input element is read by WIDTH threads.
- Load each element into Shared Memory and have several threads use the local version to reduce the memory bandwidth
  - Tiled algorithms



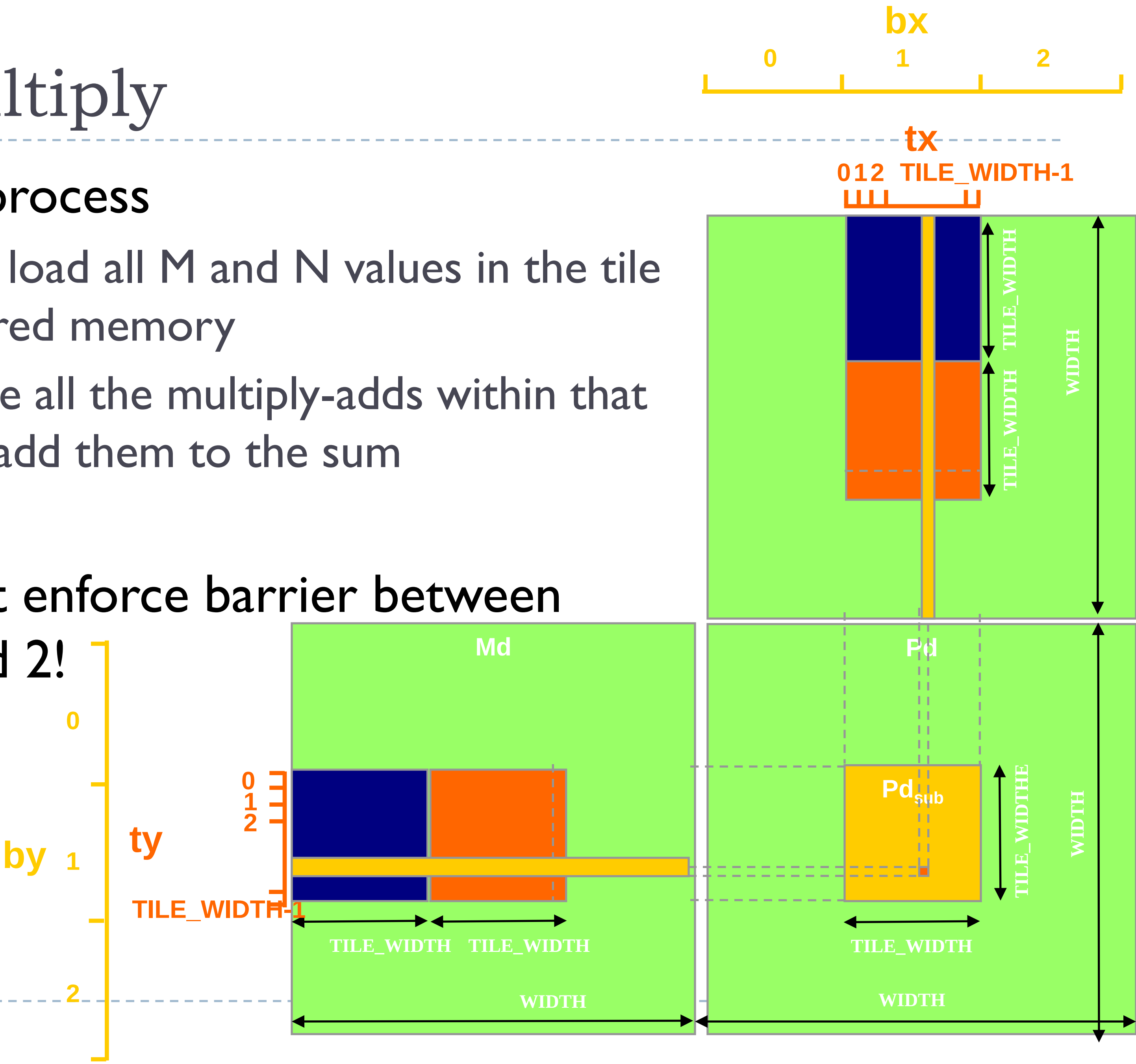
Tiled Multiply

- 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



Tiled Multiply

- Two Step process
  1. Threads load all M and N values in the tile into shared memory
  2. Compute all the multiply-adds within that tile and add them to the sum
- Note: must enforce barrier between steps 1 and 2!



Device Runtime Component: Synchronization Function

void __syncthreads();

- Synchronizes all threads in a block (similar: MPI_Barrier)
  - Once all threads have reached this point, execution resumes normally
- Used to avoid race conditions when accessing shared or global memory
- Allowed in conditional constructs only if the conditional is uniform across the entire thread block

First-order Size Considerations in G80

- 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.
  - Compute to memory ratio is now 16:1 !!
  - Memory bandwidth no longer a limiting factor

CUDA Code:
Kernel Execution Configuration

```
// Setup the execution configuration
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);
dim3 dimGrid(Width / TILE_WIDTH,
                    Width / TILE_WIDTH);
```

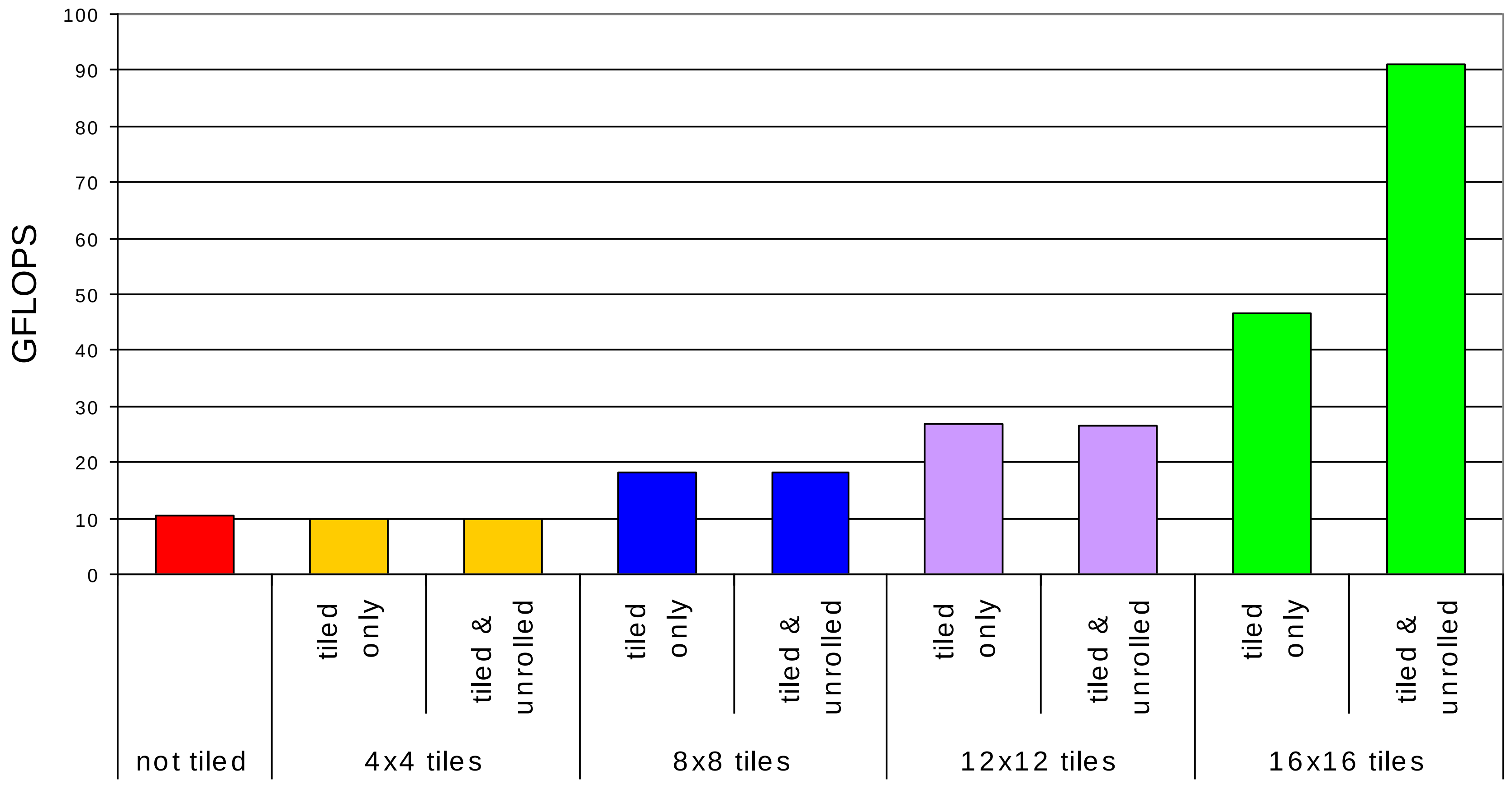
Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, 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 Pd element to work on
      int Row = by * TILE_WIDTH + ty;
      int Col = bx * TILE_WIDTH + tx;
      float Pvalue = 0;

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

          for (int k = 0; k < TILE_WIDTH; ++k)
                   Pvalue += Mds[ty][k] * Nds[k][tx];

          __syncthreads();
      }

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

G80 Shared Memory and Threading

- Each SM in G80 has 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 up to 8 Thread Blocks actively executing
    - This allows up to 8*512 = 4,096 pending loads. (2 per thread, 256 threads per block)
  - 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 per SM
- Using 16x16 tiling, we reduce the accesses to the global memory by a factor of 16
  - The 86.4B/s bandwidth can now support (86.4/4)*16 = 347.6 GFLOPS!

Tiling Size Effects



What’s Limiting My Code?

- Am I bandwidth bound? (How do I tell?)
  - Make sure I have high thread occupancy to tolerate latencies (lots of threads)
    - These threads can get some work done while we wait for memory
  - Move re-used values to closer memories
    - Shared
    - Constant/Texture

- Am I not bandwidth bound – what is now my limit?
  - Take a closer look at the instruction stream
    - Unroll loops
    - Minimize branch divergence

Exercise: Particles (n-Body)

```
cp -r ~ernstdj/NCSI2010 .

go to "particles" directory.

less README.txt
```

(we give you the basic kernel – now make it fast!)