GPU Memory

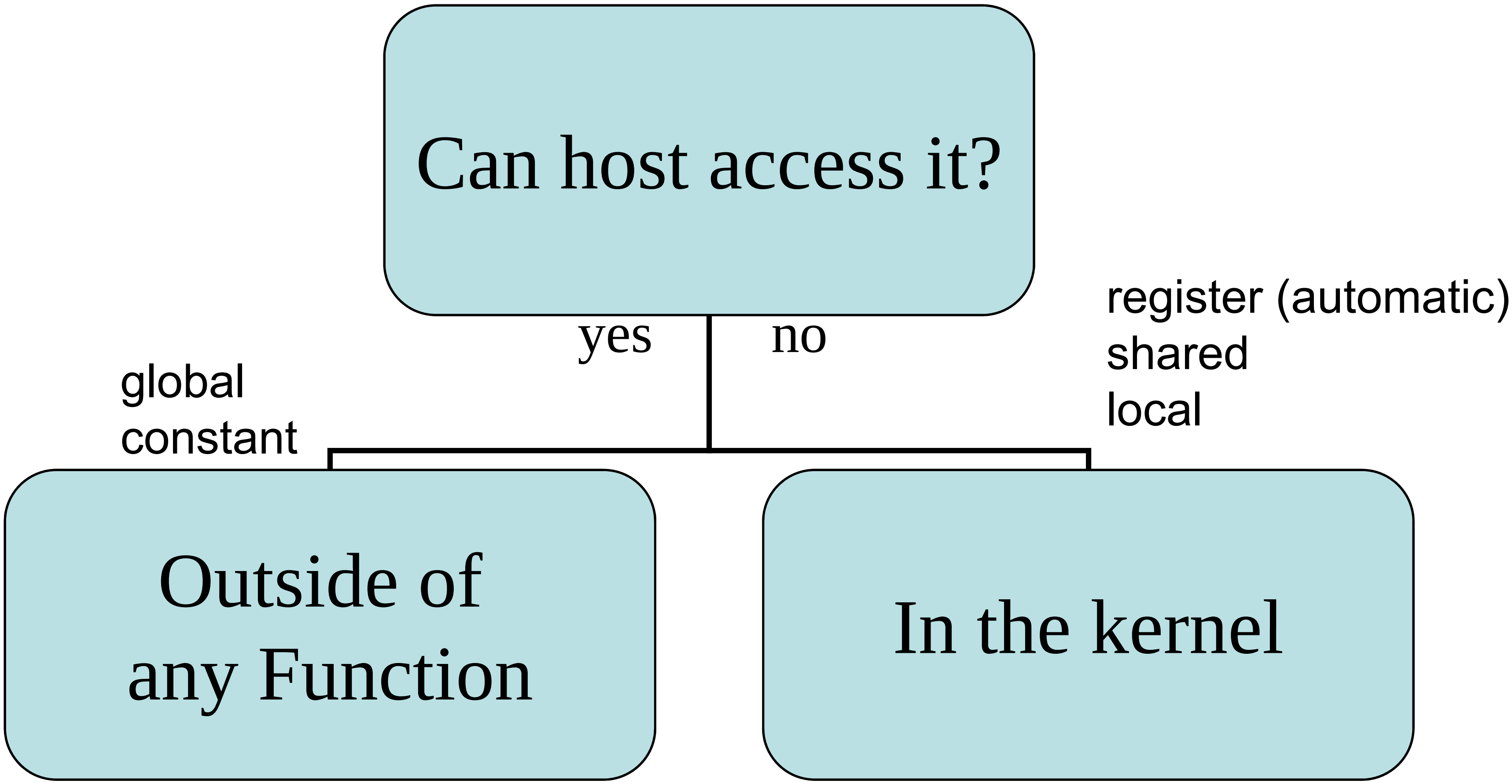
— Memory issue for CUDA programming

CUDA Variable Type Qualifiers

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

- **`__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**

Where to declare variables?



Variable Type Restrictions

- **Pointers can only point to memory allocated or declared in global memory:**
  - **Allocated in the host and passed to the kernel:**

    ```
    __global__ void KernelFunc(float* ptr)
    ```

  - **Obtained as the address of a global variable:**

    ```
    float* ptr = &GlobalVar;
    ```

A Common Programming Strategy

- **Global memory is much slower than shared memory**
- **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**
  - **Handle each data subset with one thread block by:**
    - **Loading the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism**
    - **Performing the computation on the subset from shared memory; each thread can efficiently multi-pass over any data element**
    - **Copying results from shared memory to global memory**

A Common Programming Strategy (Cont.)

- **Constant memory also resides in device memory - much slower access 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)**

For texture memory usage, see NVIDIA document.

GPU Atomic Integer Operations

- **Atomic operations on integers in global memory:**
  - Associative operations on signed/unsigned ints
  - add, sub, min, max, ...
  - and, or, xor
  - Increment, decrement
  - Exchange, compare and swap
- **Requires hardware with compute capability 1.1 and above.**

Shared Memory

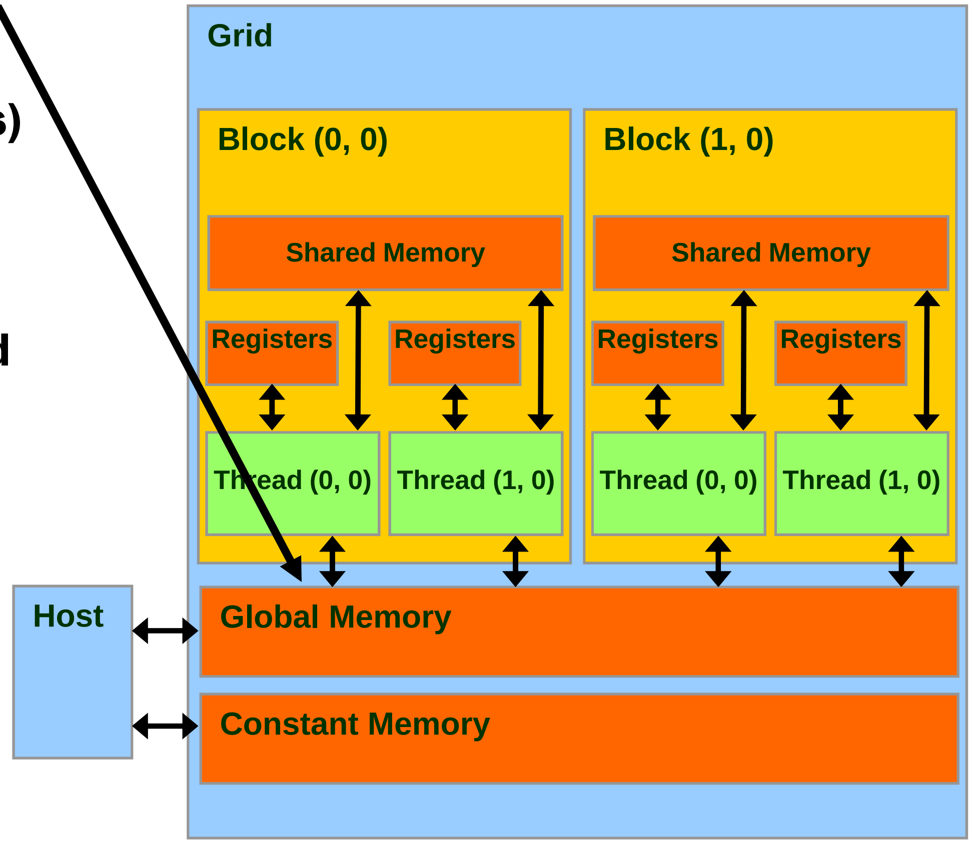
- **Matrix Multiplication as example again.**

Review: Matrix Multiplication 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;
}
```

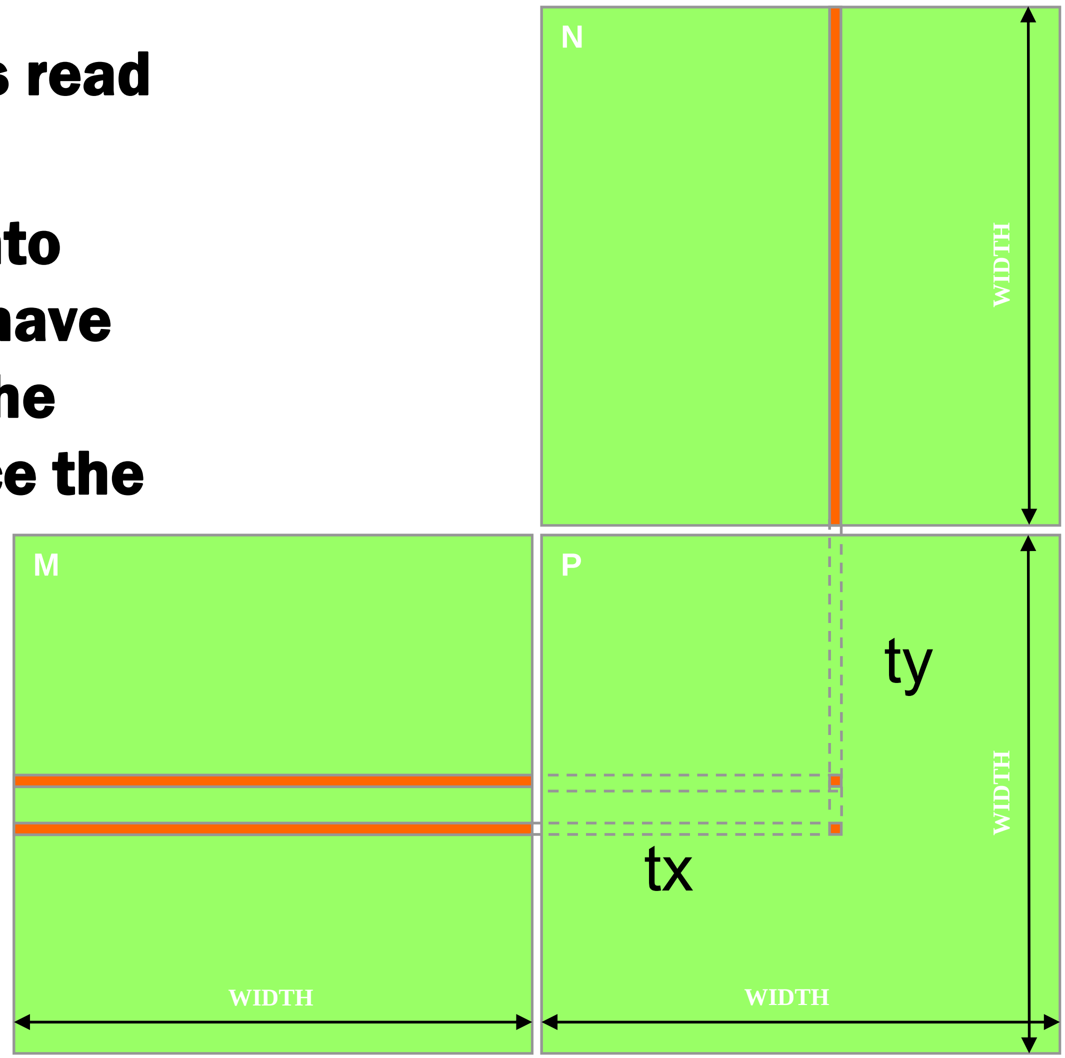
How about performance on G80?

- All threads access global memory for their input matrix elements
  - Two memory accesses (8 bytes) per floating point multiply-add
  - 4B/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
- Need to drastically cut down memory accesses to get closer to the peak 346.5 GFLOPS



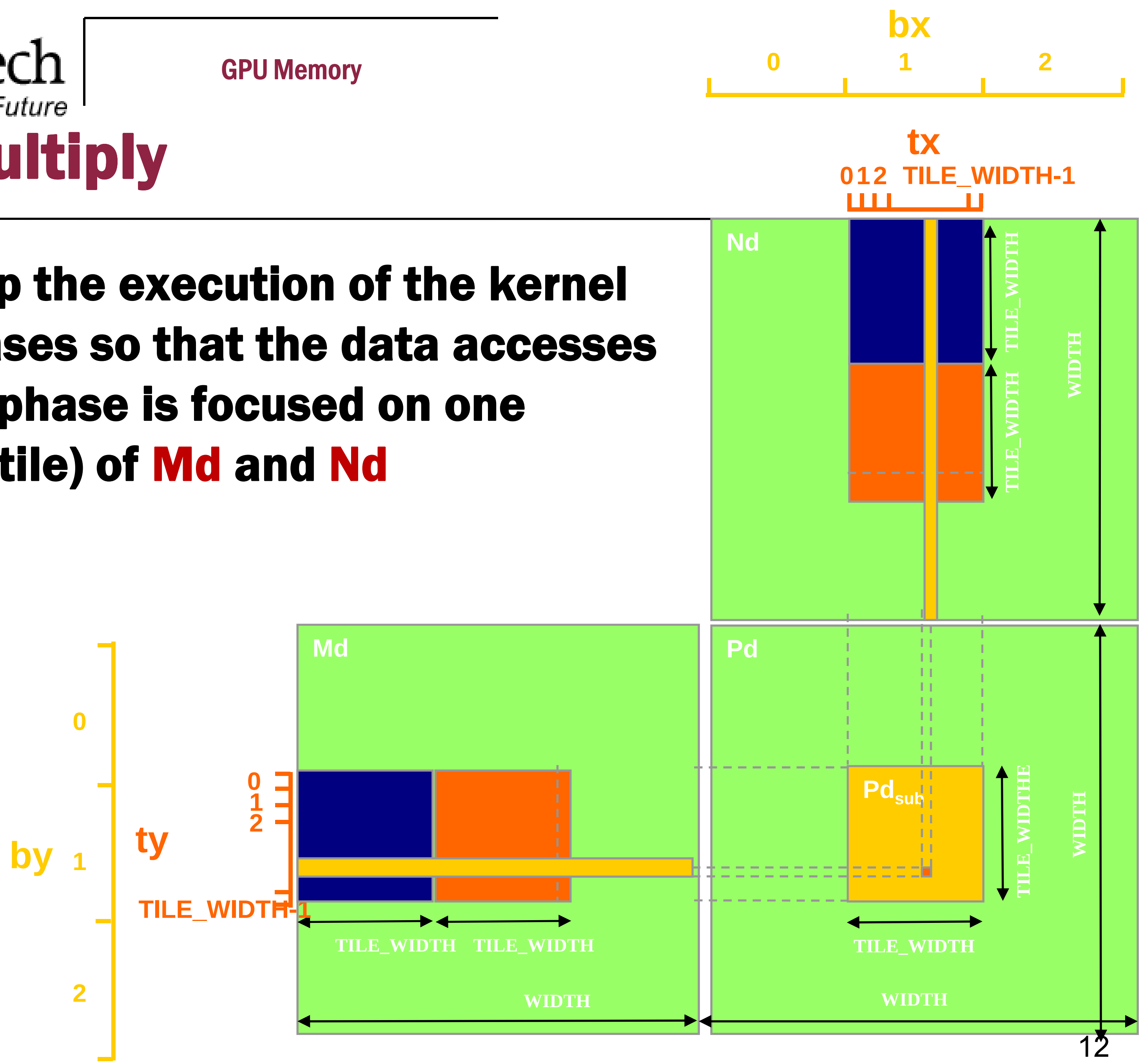
Idea: Use Shared Memory to reuse global memory data

- **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**

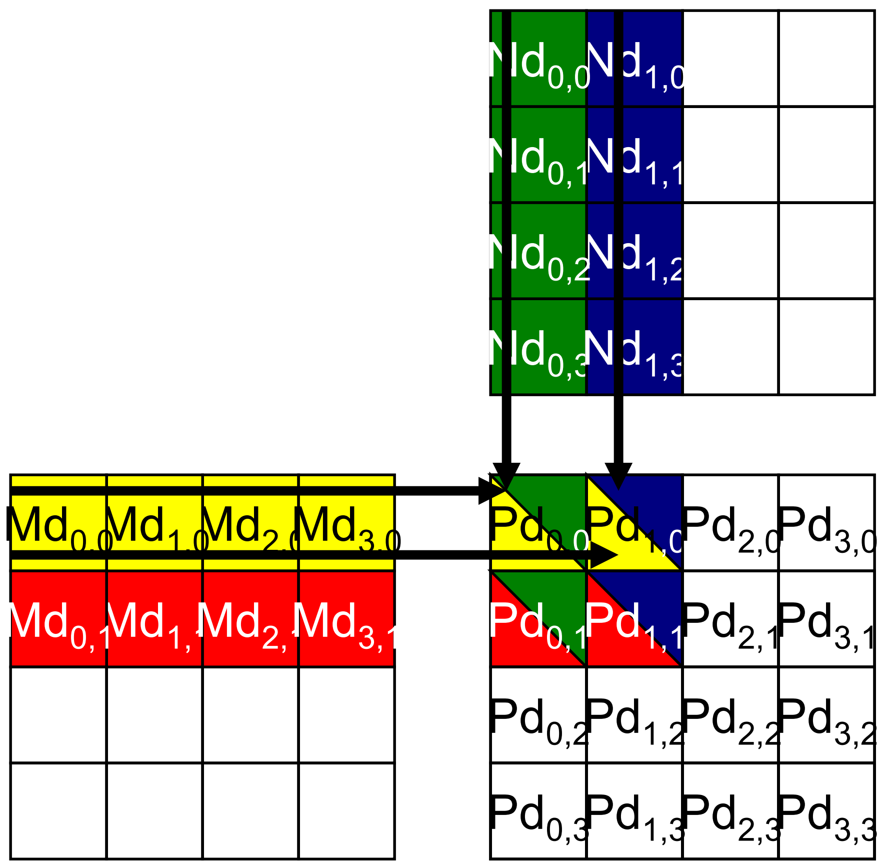


Tiled Multiply

- 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



Example


Nd0,0 Nd1,0
Nd0,1 Nd1,1
Nd0,2 Nd1,2
Nd0,3 Nd1,3
Md0,0 Md1,0 Md2,0 Md3,0
Md0,1 Md1,1 Md2,1 Md3,1
Pd0,0 Pd1,0 Pd2,0 Pd3,0
Pd0,1 Pd1,1 Pd2,1 Pd3,1
Pd0,2 Pd1,2 Pd2,2 Pd3,2
Pd0,3 Pd1,3 Pd2,3 Pd3,3

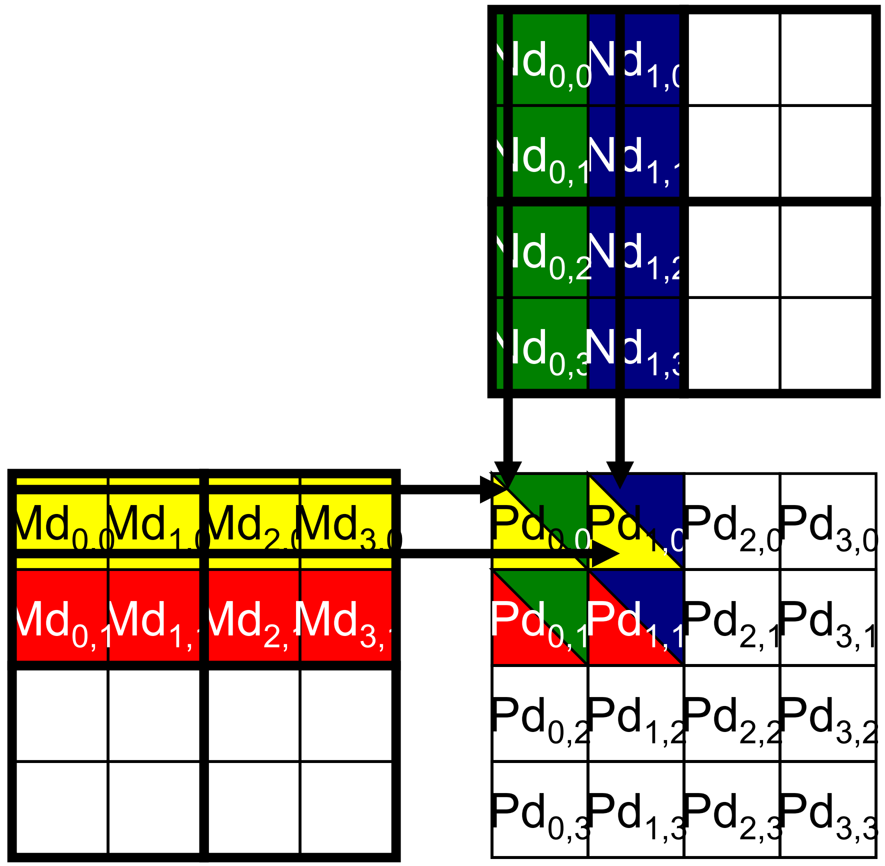
Example (Cont')

- **Every Md and Nd Element is used exactly twice in generating a 2X2 tile of P**

Access order

$P_{0,0}$ thread$_{0,0}$	$P_{1,0}$ thread$_{1,0}$	$P_{0,1}$ thread$_{0,1}$	$P_{1,1}$ thread$_{1,1}$
$M_{0,0} * N_{0,0}$	$M_{0,0} * N_{1,0}$	$M_{0,1} * N_{0,0}$	$M_{0,1} * N_{1,0}$
$M_{1,0} * N_{0,1}$	$M_{1,0} * N_{1,1}$	$M_{1,1} * N_{0,1}$	$M_{1,1} * N_{1,1}$
$M_{2,0} * N_{0,2}$	$M_{2,0} * N_{1,2}$	$M_{2,1} * N_{0,2}$	$M_{2,1} * N_{1,2}$
$M_{3,0} * N_{0,3}$	$M_{3,0} * N_{1,3}$	$M_{3,1} * N_{0,3}$	$M_{3,1} * N_{1,3}$

Breaking Md and Nd into Tiles



Example (2)

- **Each phase of a Thread Block uses one tile from Md and one from Nd**

				Step 4	Step 5	Step 6
$T_{0,0}$	$\mathbf{Md_{0,0}}$ ↓ $Mds_{0,0}$	$\mathbf{Nd_{0,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{1,0}$*$Nds_{0,1}$	$\mathbf{Md_{2,0}}$ ↓ $Mds_{0,0}$	$\mathbf{Nd_{0,2}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{1,0}$*$Nds_{0,1}$
$T_{1,0}$	$\mathbf{Md_{1,0}}$ ↓ $Mds_{1,0}$	$\mathbf{Nd_{1,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{0,0}$*$Nds_{1,0}$ + $Mds_{1,0}$*$Nds_{1,1}$	$\mathbf{Md_{3,0}}$ ↓ $Mds_{1,0}$	$\mathbf{Nd_{1,2}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{0,0}$*$Nds_{1,0}$ + $Mds_{1,0}$*$Nds_{1,1}$
$T_{0,1}$	$\mathbf{Md_{0,1}}$ ↓ $Mds_{0,1}$	$\mathbf{Nd_{0,1}}$ ↓ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{0,1}$	$\mathbf{Md_{2,1}}$ ↓ $Mds_{0,1}$	$\mathbf{Nd_{0,3}}$ ↓ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{0,1}$
$T_{1,1}$	$\mathbf{Md_{1,1}}$ ↓ $Mds_{1,1}$	$\mathbf{Nd_{1,1}}$ ↓ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1,1}$*$Nds_{1,1}$	$\mathbf{Md_{3,1}}$ ↓ $Mds_{1,1}$	$\mathbf{Nd_{1,3}}$ ↓ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1,1}$*$Nds_{1,1}$

time →

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.
  - Memory bandwidth no longer a limiting factor

CUDA Code – Kernel Execution Configuration

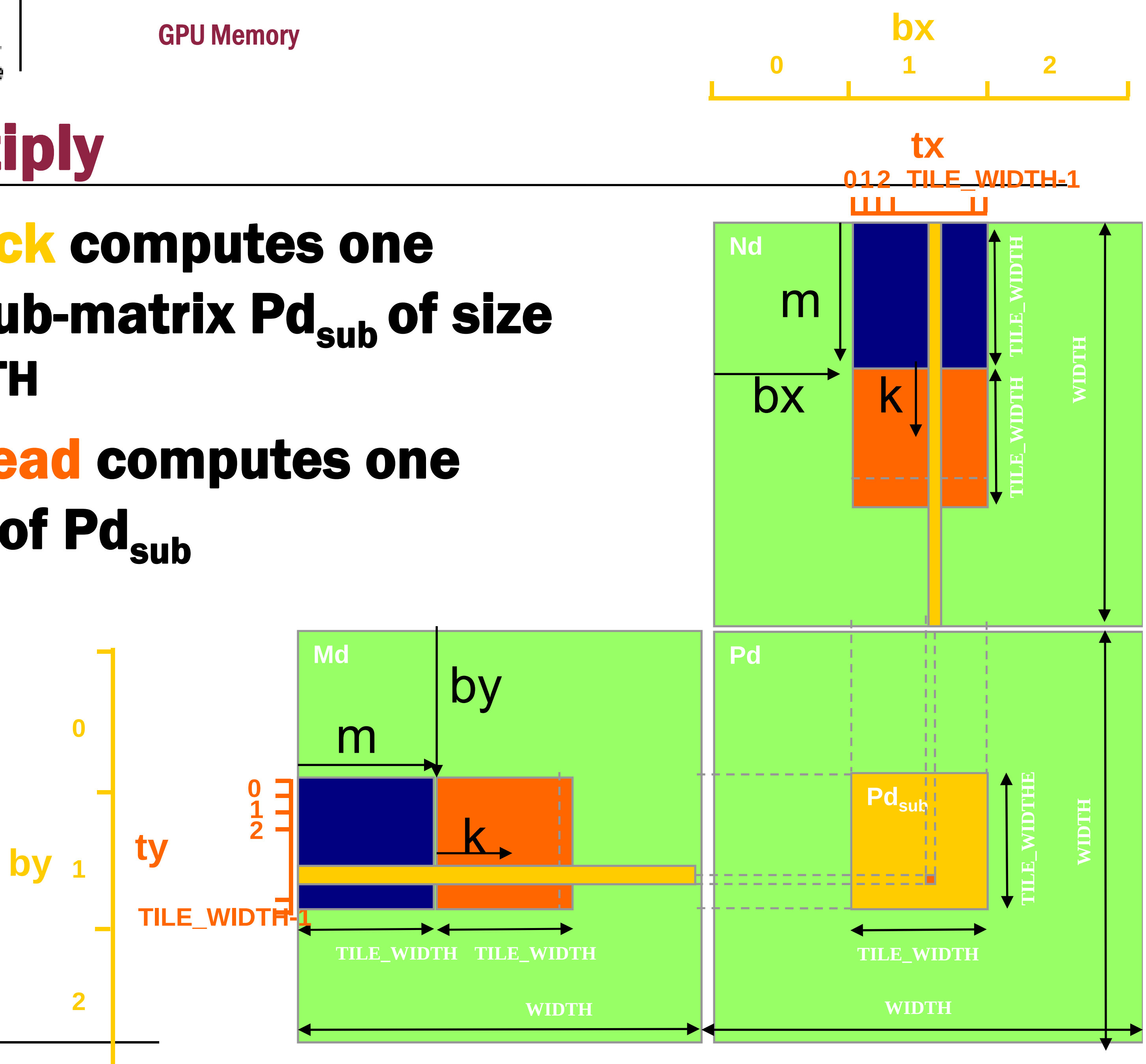
```
// Setup the execution configuration
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);
dim3 dimGrid(Width  / TILE_WIDTH,
             Width /  TILE_WIDTH);
```

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) {
// Coolaborative 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];
     Synchthreads();
    }
    Pd[Row*Width+Col] = Pvalue;
}
```

Tiled Multiply

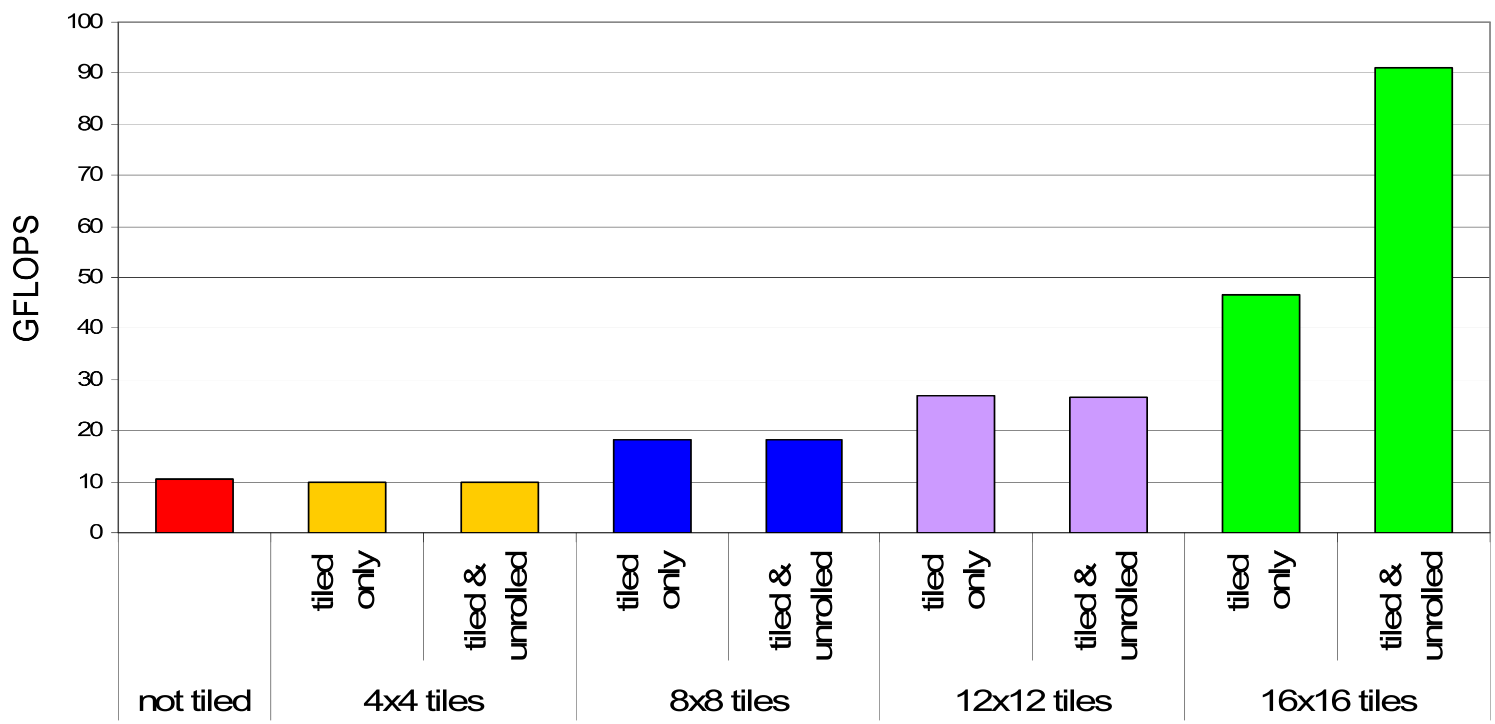
- Each **block** computes one square sub-matrix Pd_{sub} of size TILE_WIDTH
- Each **thread** computes one element of Pd_{sub}



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)**
  - **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 86.4B/s bandwidth can now support (86.4/4)*16 = 347.6 GFLOPS!**

Tiling Size Effects



Summary- Typical Structure of a CUDA Program

- Global variables declaration
  - __host__
  - __device__... __global__, __constant__, __texture__
- Function prototypes
  - __global__ void kernelOne(…)
  - float handyFunction(…)
- Main ()
  - allocate memory space on the device – cudaMalloc(&d_GlblVarPtr, bytes)
  - transfer data from host to device – cudaMemCpy(d_GlblVarPtr, h_Gl…)
  - execution configuration setup
  - kernel call – kernelOne<<<execution configuration>>>(args…);
  - transfer results from device to host – cudaMemCpy(h_GlblVarPtr,…)
  - optional: compare against golden (host computed) solution
- Kernel – void kernelOne(type args,…)
  - variables declaration - __local__, __shared__
    - automatic variables transparently assigned to registers or local memory
  - syncthreads()…
- Other functions
  - float handyFunction(int inVar…);

repeat as needed