

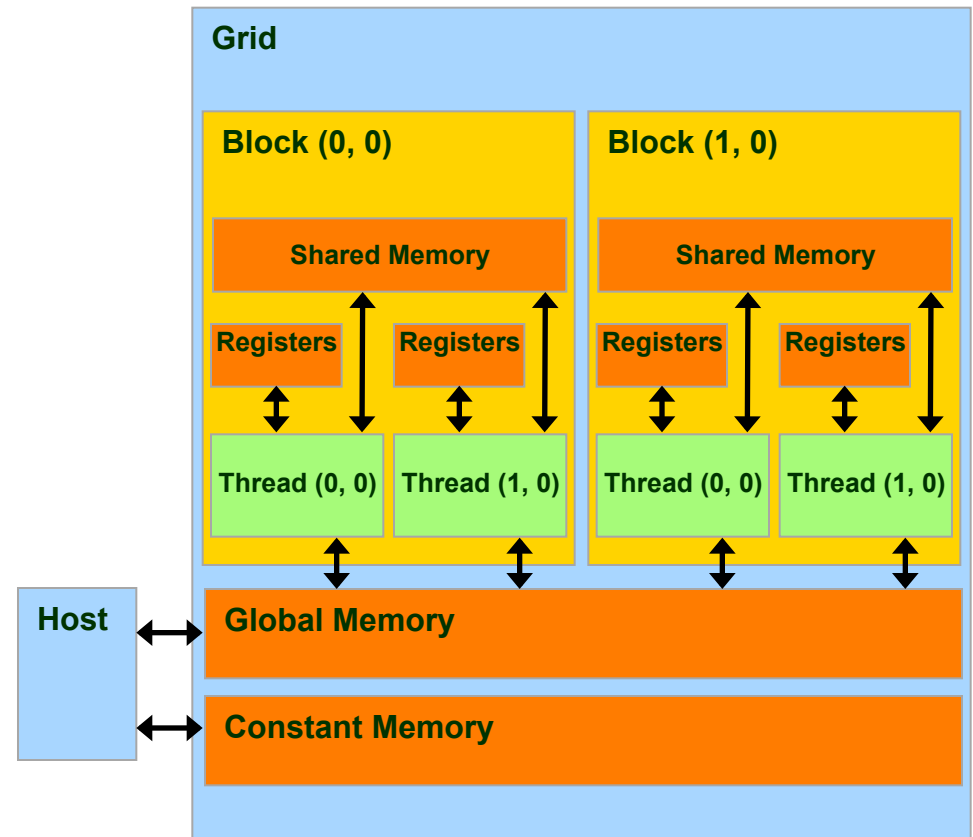
Two vertical lines, one blue and one orange, are positioned on the left side of the slide.

Programming Massively Parallel Processors

CUDA Memories

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**

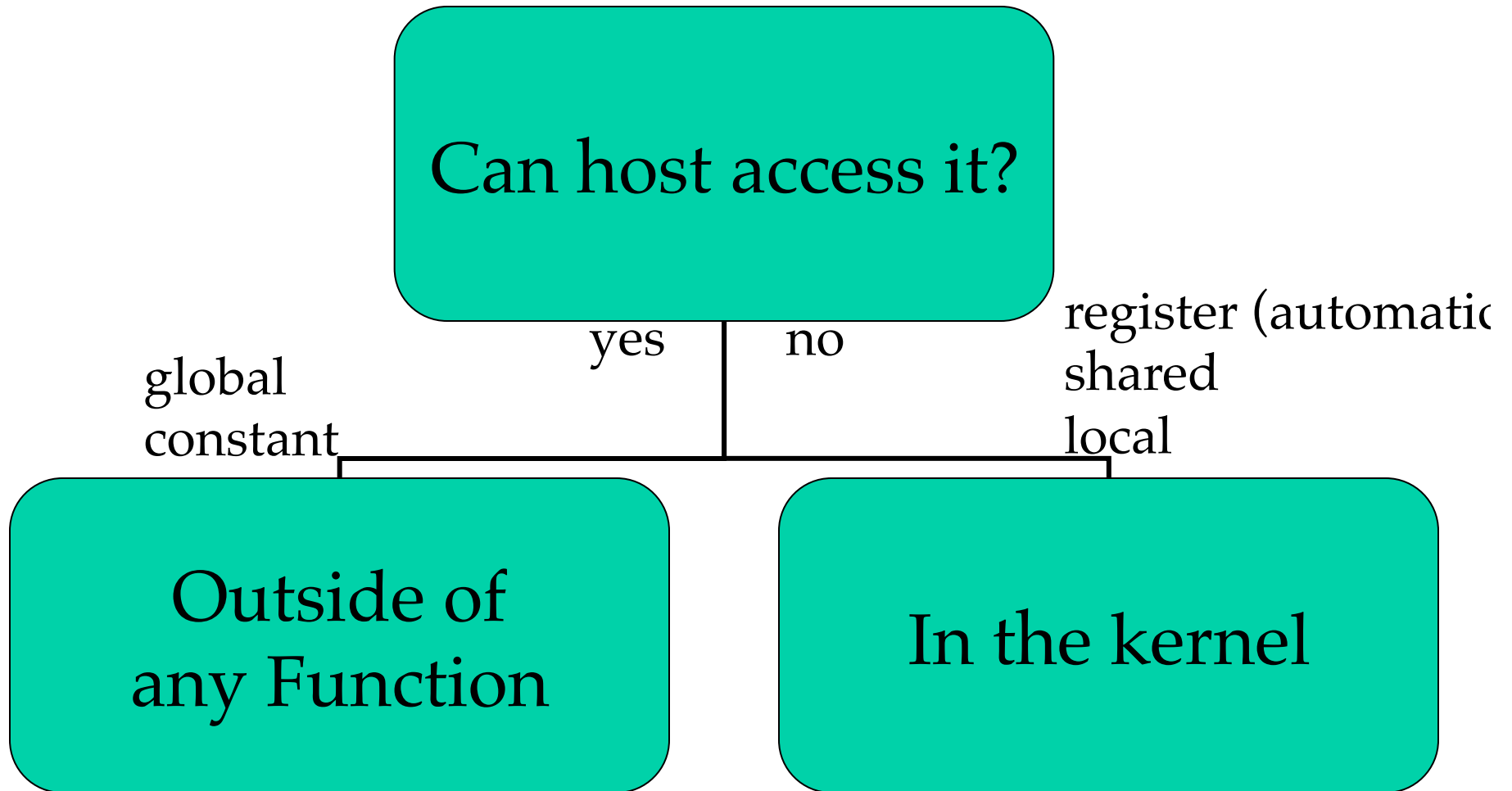


CUDA Variable Type Qualifiers

Variable declaration	Memory	Scope	Lifetime
<code>__device__ __local__ int LocalVar;</code>	local	thread	thread
<code>__device__ __shared__ int SharedVar;</code>	shared	block	block
<code>__device__ int GlobalVar;</code>	global	grid	application
<code>__device__ __constant__ int ConstantVar;</code>	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 resides in device memory (DRAM)
 - much slower access 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 (DRAM) - 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:
atomicAdd(), atomicSub(), atomicExch(), atomicMin(),
atomicMax(), atomicInc(), atomicDec, atomicCAS()
 - Also some operations on bits
- Requires hardware with compute capability 1.1 and above.

Two vertical lines, one blue and one orange, are positioned on the left side of the slide.

Matrix Multiplication using Shared Memory

Review: Matrix Multiplication Kernel using Multiple Blocks

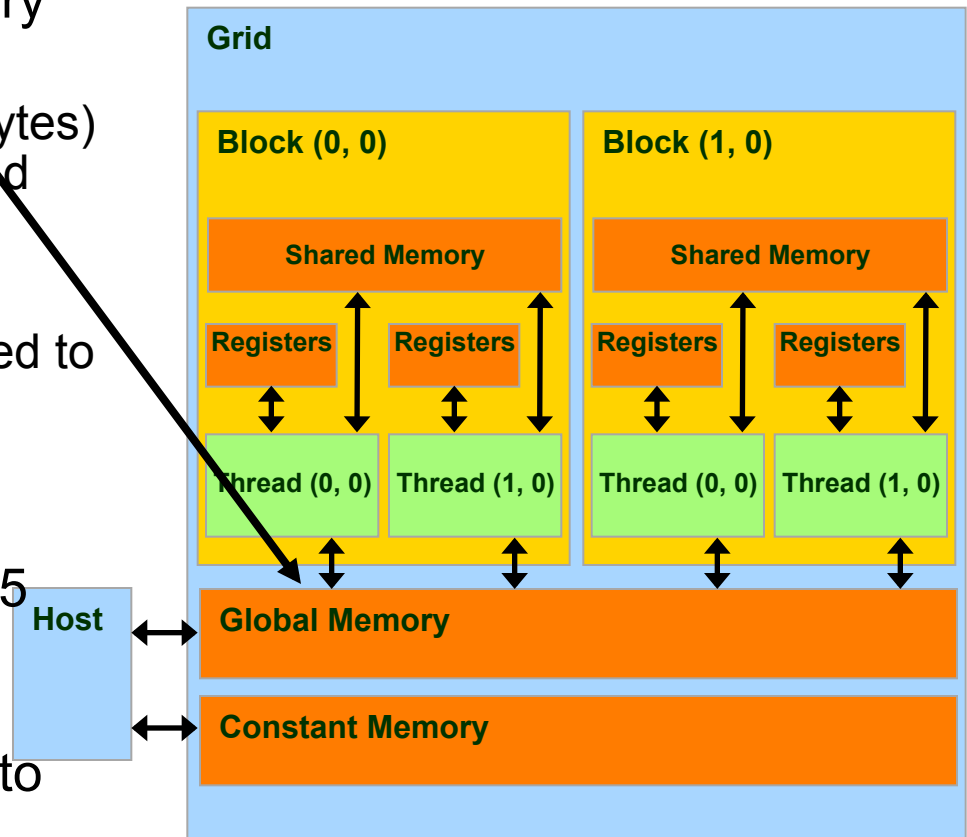
```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // Calculate the row index of the Pd element and M
    int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
    // Calculate the column idenx of Pd and N
    int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

    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;
}
```

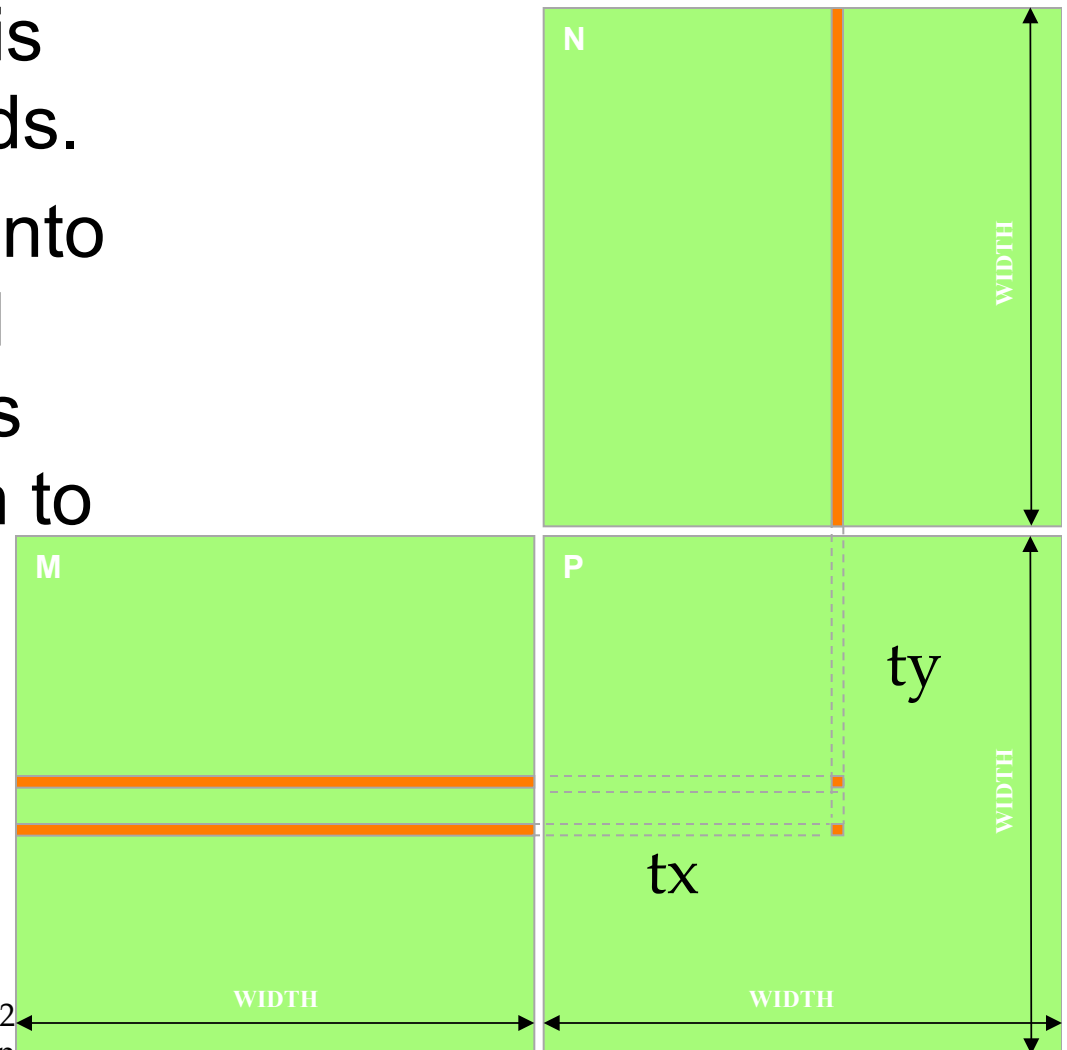
How about performance on G80?

- 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
 - $4 \times 346.5 = 1386$ 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



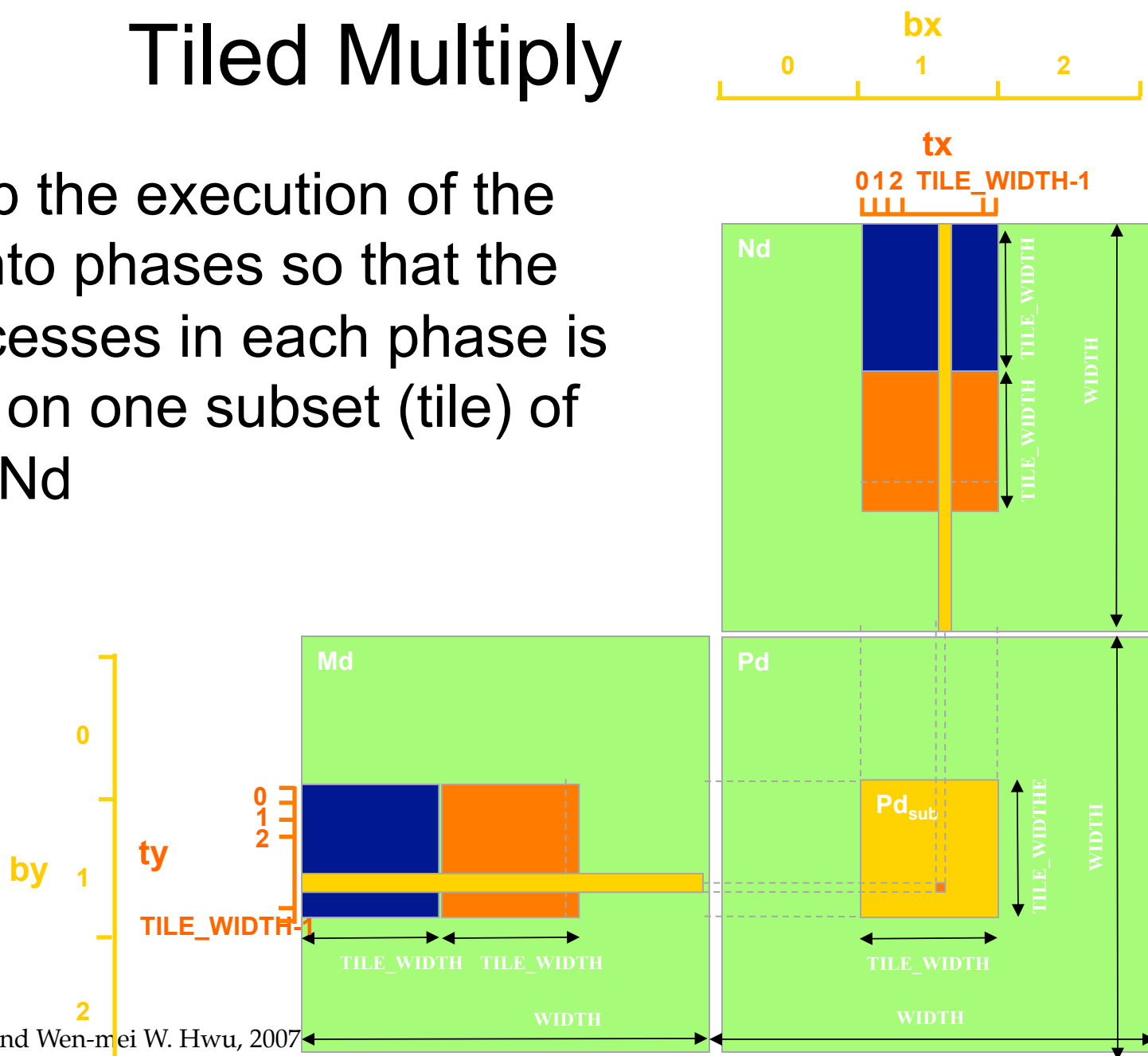
Idea: Use Shared Memory to reuse global memory data

- 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

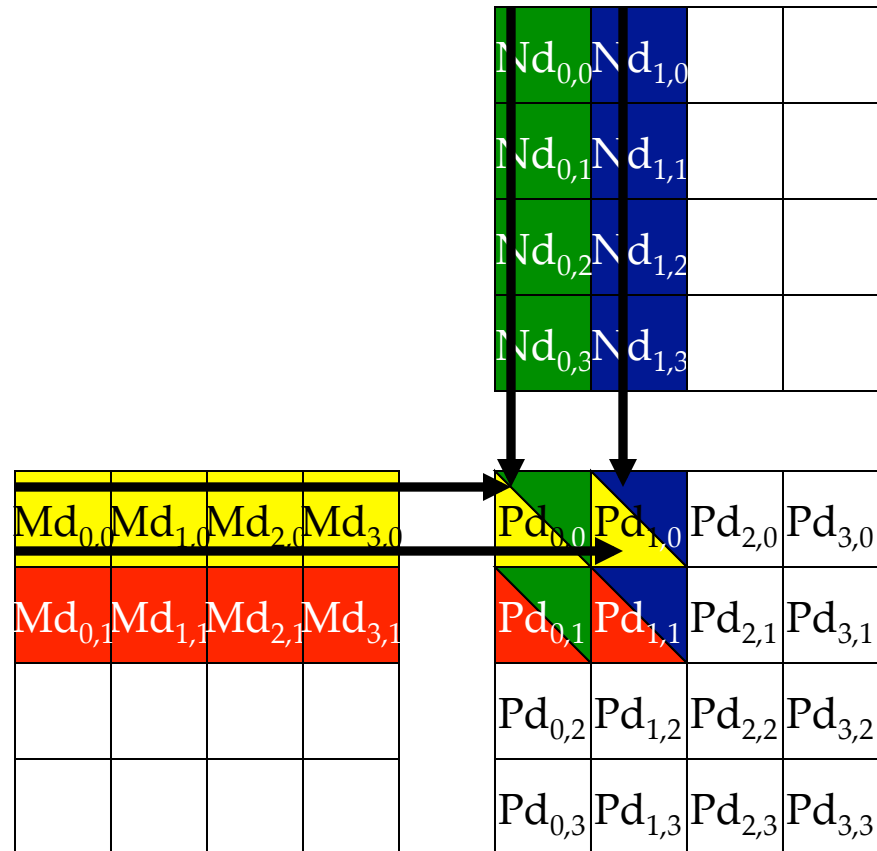


Tiled Multiply

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



A Small Example

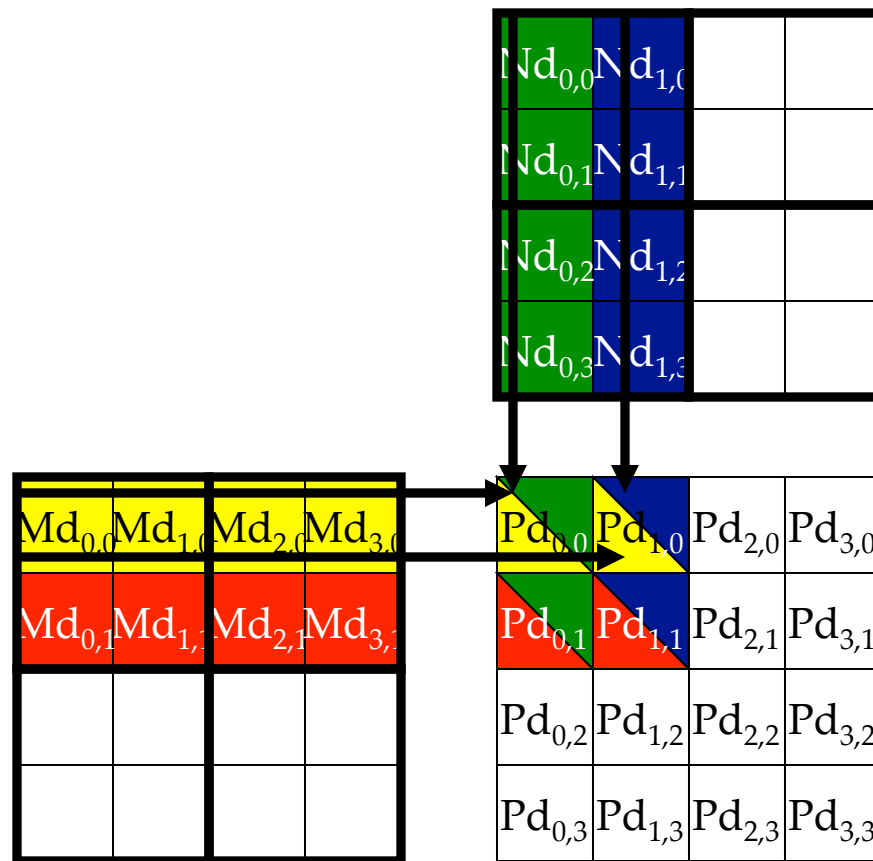


Every M_d and N_d Element is used exactly twice in generating a 2×2 tile of P

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



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

	Phase 1			Phase 2		
$T_{0,0}$	Md_{0,0} ↓ Mds _{0,0}	Nd_{0,0} ↓ Nds _{0,0}	PValue _{0,0} += Mds _{0,0} *Nds _{0,0} + Mds _{1,0} *Nds _{0,1}	Md_{2,0} ↓ Mds _{0,0}	Nd_{0,2} ↓ Nds _{0,0}	PValue _{0,0} += Mds _{0,0} *Nds _{0,0} + Mds _{1,0} *Nds _{0,1}
$T_{1,0}$	Md_{1,0} ↓ Mds _{1,0}	Nd_{1,0} ↓ Nds _{1,0}	PValue _{1,0} += Mds _{0,0} *Nds _{1,0} + Mds _{1,0} *Nds _{1,1}	Md_{3,0} ↓ Mds _{1,0}	Nd_{1,2} ↓ Nds _{1,0}	PValue _{1,0} += Mds _{0,0} *Nds _{1,0} + Mds _{1,0} *Nds _{1,1}
$T_{0,1}$	Md_{0,1} ↓ Mds _{0,1}	Nd_{0,1} ↓ Nds _{0,1}	PdValue _{0,1} += Mds _{0,1} *Nds _{0,0} + Mds _{1,1} *Nds _{0,1}	Md_{2,1} ↓ Mds _{0,1}	Nd_{0,3} ↓ Nds _{0,1}	PdValue _{0,1} += Mds _{0,1} *Nds _{0,0} + Mds _{1,1} *Nds _{0,1}
$T_{1,1}$	Md_{1,1} ↓ Mds _{1,1}	Nd_{1,1} ↓ Nds _{1,1}	PdValue _{1,1} += Mds _{0,1} *Nds _{1,0} + Mds _{1,1} *Nds _{1,1}	Md_{3,1} ↓ Mds _{1,1}	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 $16 \times 16 = 256$ threads
- There should be many thread blocks
 - A 1024×1024 Pd gives $64 \times 64 = 4096$ Thread Blocks
- Each thread block perform $2 \times 256 = 512$ float loads from global memory for $256 * (2 \times 16) = 8,192$ 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)
{
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;

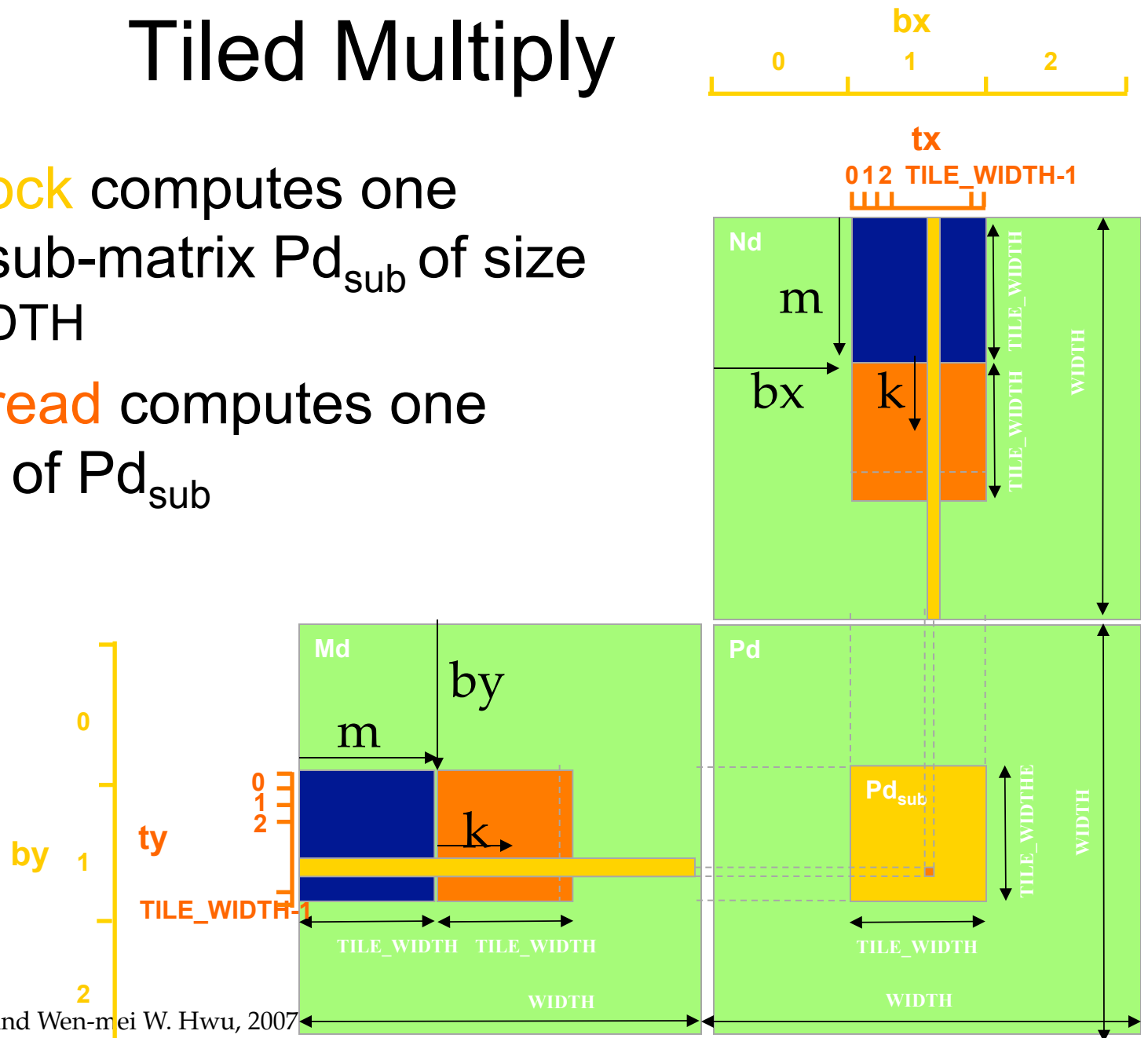
// Identify the row and column of the Pd element to work on
5.  int Row = by * TILE_WIDTH + ty;
6.  int Col = bx * TILE_WIDTH + tx;

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

11.     for (int k = 0; k < TILE_WIDTH; ++k)
12.         Pvalue += Mds[ty][k] * Nds[k][tx];
13.     Syncthreads();
14. }
13. Pd[Row*Width+Col] = Pvalue;
}
```

Tiled Multiply

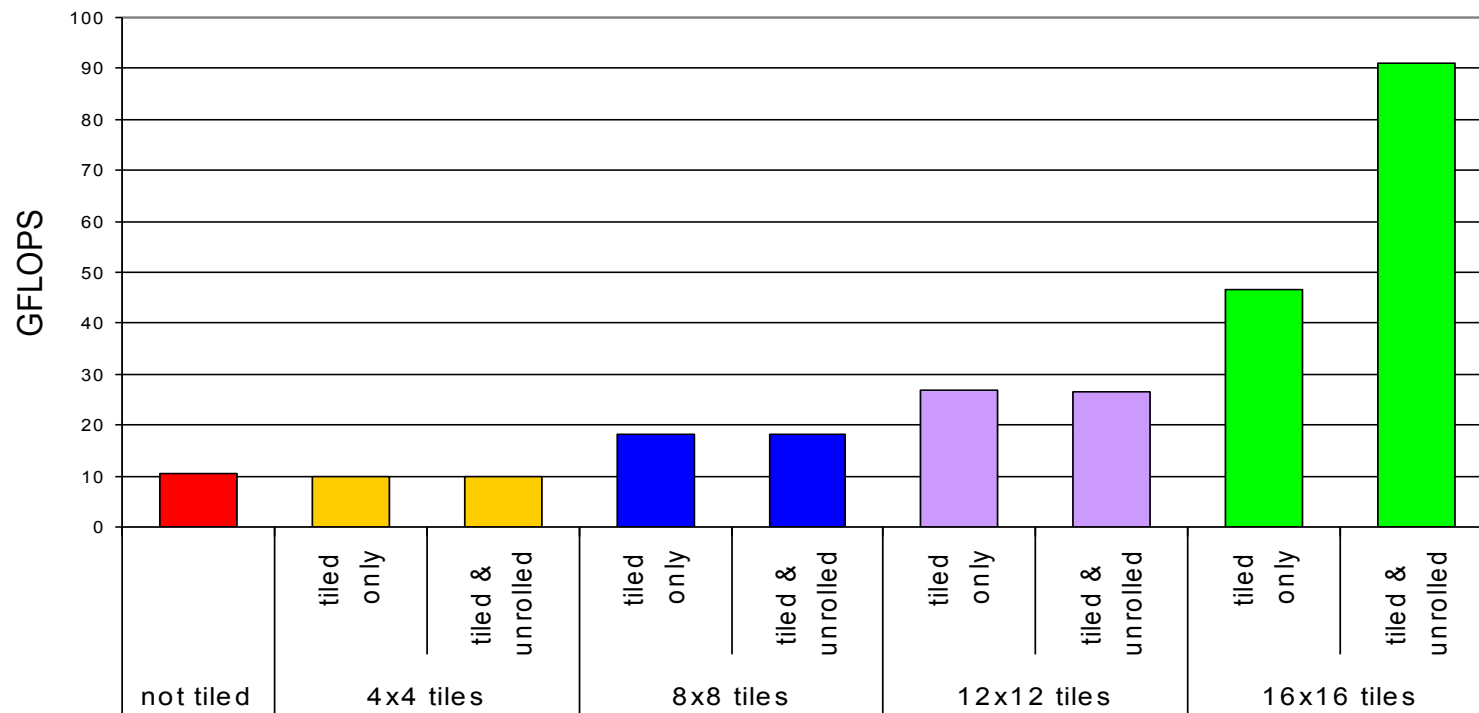
- Each **block** computes one square sub-matrix Pd_{sub} of size $TILE_WIDTH$
- Each **thread** computes one element of Pd_{sub}




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 $2 \times 256 \times 4\text{B} = 2\text{KB}$ of shared memory.
 - Can potentially have up to 8 Thread Blocks actively executing
 - This allows up to $8 \times 512 = 4,096$ pending loads. (2 per thread, 256 threads per block)
 - The next `TILE_WIDTH` 32 would lead to $2 \times 32 \times 32 \times 4\text{B} = 8\text{KB}$ 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 $(86.4/4) \times 16 = 347.6$ 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_GlbIVarPtr, bytes)`
 - transfer data from host to device – `cudaMemcpy(d_GlbIVarPtr, h_Gl...)`
 - execution configuration setup
 - kernel call – `kernelOne<<<execution configuration>>>(args...);`
 - transfer results from device to host – `cudaMemcpy(h_GlbIVarPtr,...)`
 - 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

Two vertical lines, one blue and one orange, are positioned on the left side of the slide.

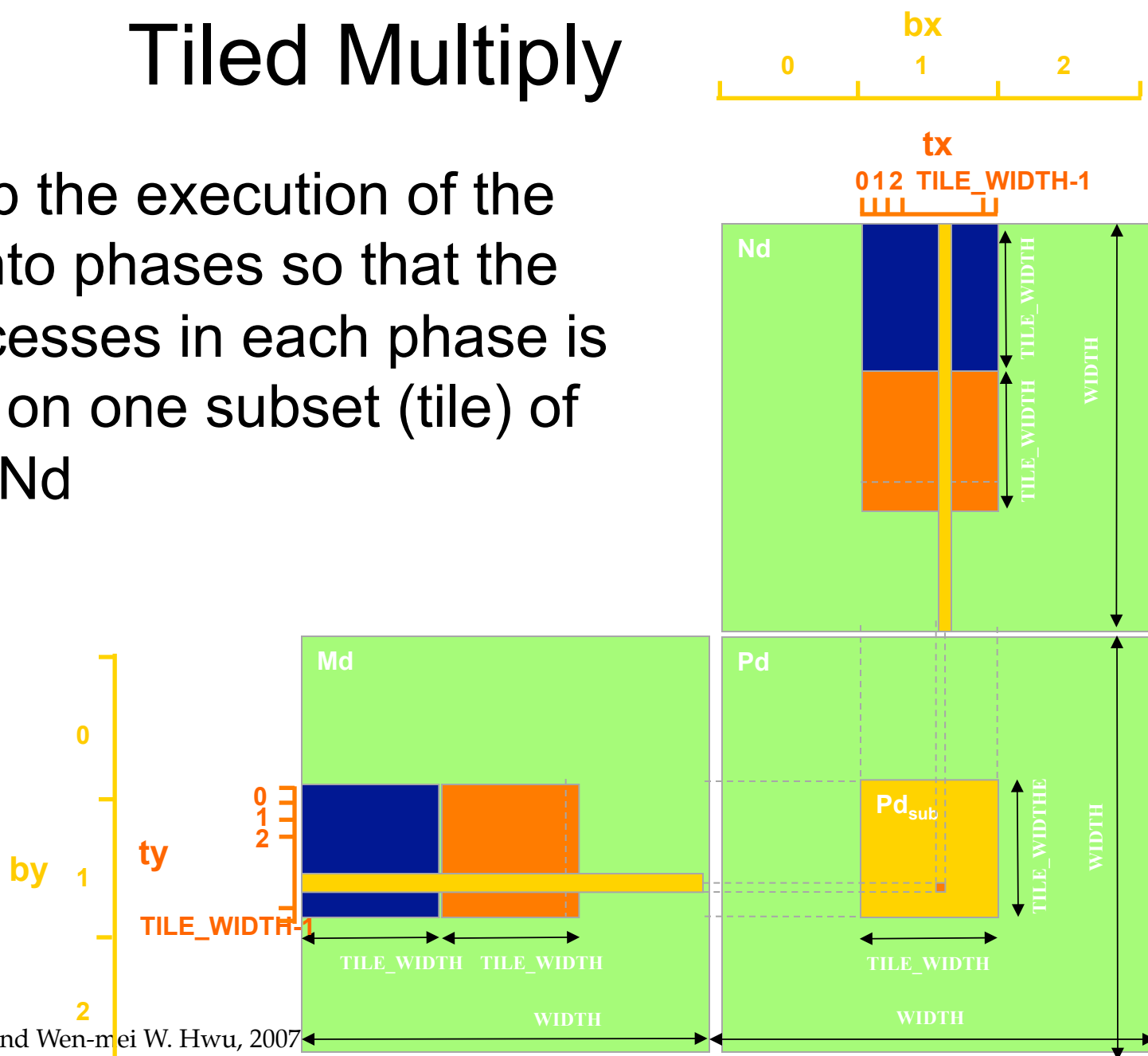
ECE 498AL

Programming Massively Parallel Processors

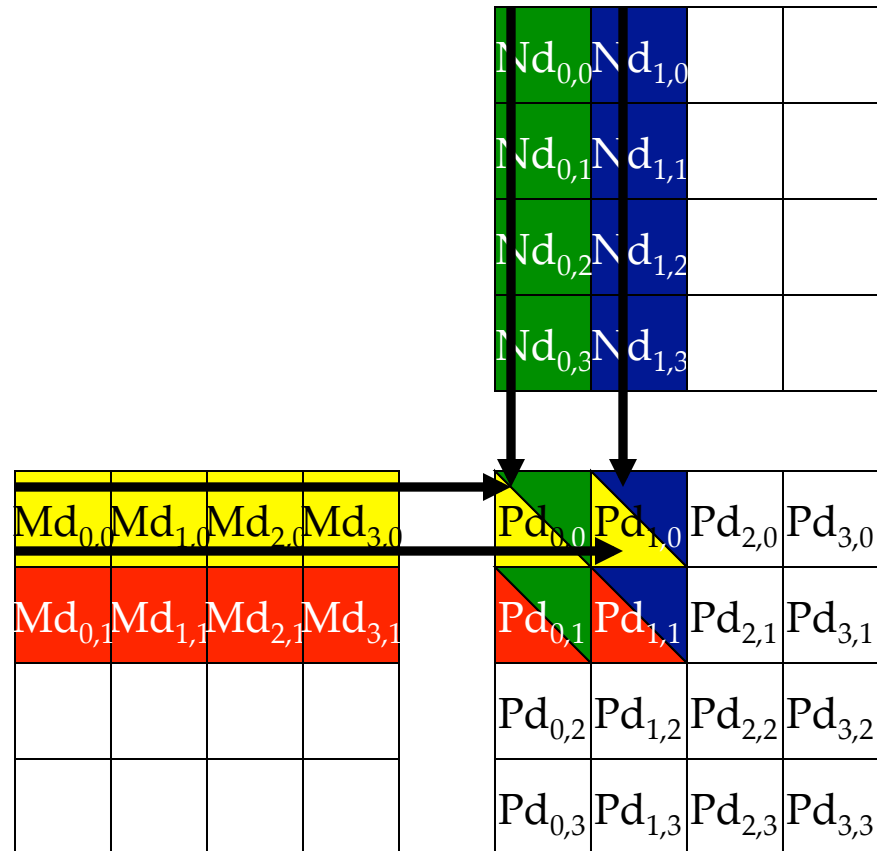
Lecture 6: CUDA Memories Part 2

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 M_d and N_d




A Small Example



Every M_d and N_d Element is used exactly twice in generating a 2×2 tile of P

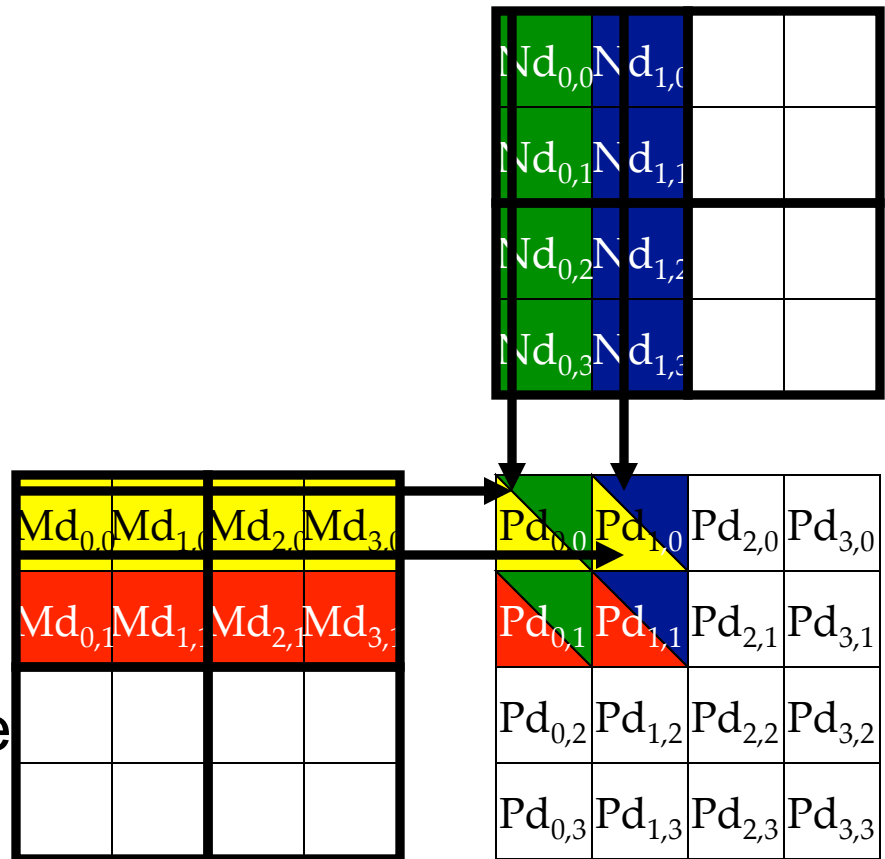
Access
order



$P_{0,0}$ 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

- Break up the inner product loop of each thread into phases
- At the beginning of each phase, load the Md and Nd elements that everyone needs during the phase into shared memory
- Everyone access the Md and Nd elements from the shared memory during the phase



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

	Phase 1			Phase 2		
$T_{0,0}$	Md_{0,0} ↓ Mds _{0,0}	Nd_{0,0} ↓ Nds _{0,0}	PValue _{0,0} += Mds _{0,0} *Nds _{0,0} + Mds _{1,0} *Nds _{0,1}	Md_{2,0} ↓ Mds _{0,0}	Nd_{0,2} ↓ Nds _{0,0}	PValue _{0,0} += Mds _{0,0} *Nds _{0,0} + Mds _{1,0} *Nds _{0,1}
$T_{1,0}$	Md_{1,0} ↓ Mds _{1,0}	Nd_{1,0} ↓ Nds _{1,0}	PValue _{1,0} += Mds _{0,0} *Nds _{1,0} + Mds _{1,0} *Nds _{1,1}	Md_{3,0} ↓ Mds _{1,0}	Nd_{1,2} ↓ Nds _{1,0}	PValue _{1,0} += Mds _{0,0} *Nds _{1,0} + Mds _{1,0} *Nds _{1,1}
$T_{0,1}$	Md_{0,1} ↓ Mds _{0,1}	Nd_{0,1} ↓ Nds _{0,1}	PdValue _{0,1} += Mds _{0,1} *Nds _{0,0} + Mds _{1,1} *Nds _{0,1}	Md_{2,1} ↓ Mds _{0,1}	Nd_{0,3} ↓ Nds _{0,1}	PdValue _{0,1} += Mds _{0,1} *Nds _{0,0} + Mds _{1,1} *Nds _{0,1}
$T_{1,1}$	Md_{1,1} ↓ Mds _{1,1}	Nd_{1,1} ↓ Nds _{1,1}	PdValue _{1,1} += Mds _{0,1} *Nds _{1,0} + Mds _{1,1} *Nds _{1,1}	Md_{3,1} ↓ Mds _{1,1}	Nd_{1,3} ↓ Nds _{1,1}	PdValue _{1,1} += Mds _{0,1} *Nds _{1,0} + Mds _{1,1} *Nds _{1,1}

time →

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;

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

11.     for (int k = 0; k < TILE_WIDTH; ++k)
12.         Pvalue += Mds[ty][k] * Nds[k][tx];
13.     Syncthreads();
14. }
13.  Pd[Row*Width+Col] = Pvalue;
}
```

CUDA Code – Kernel Execution Configuration

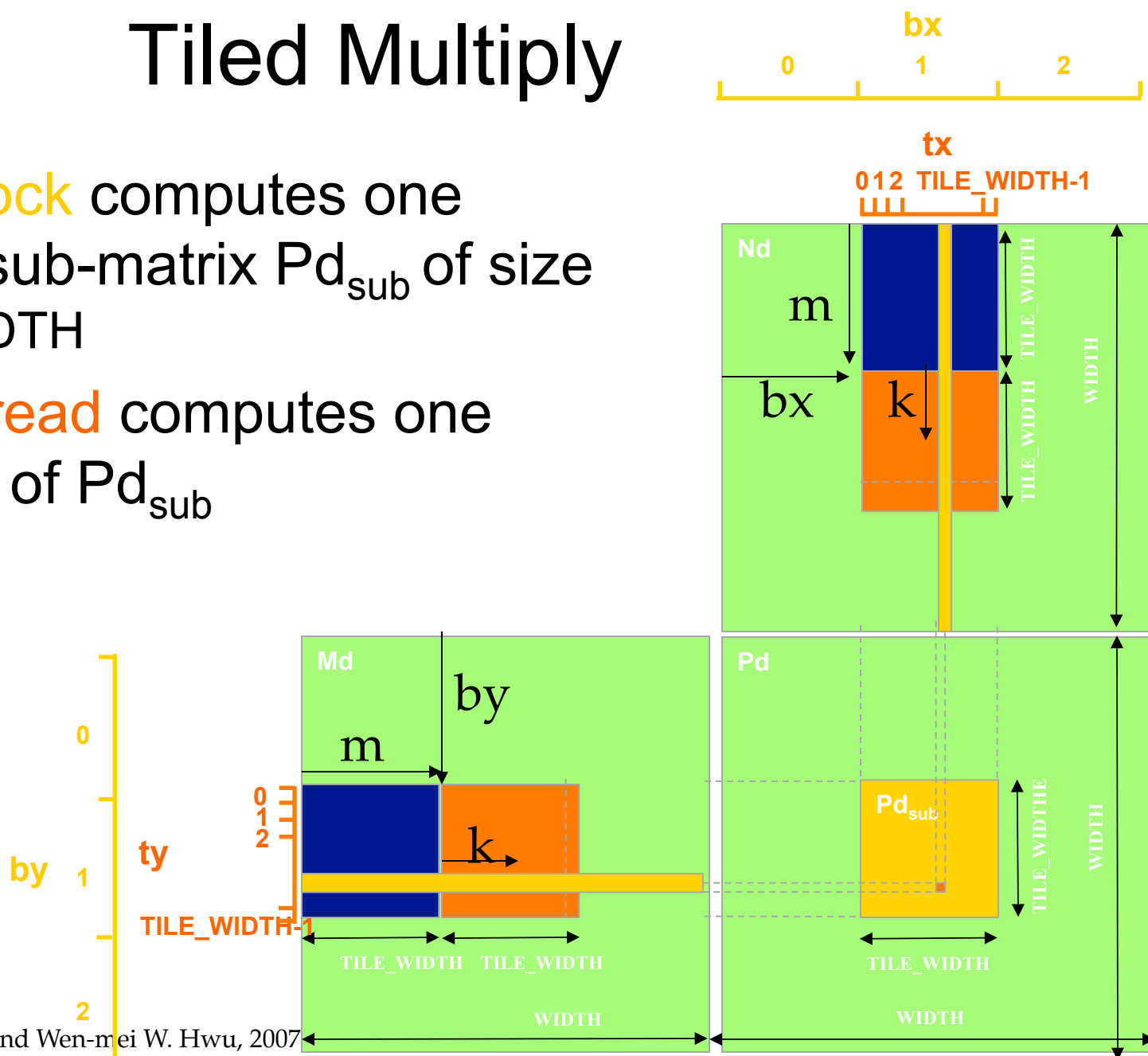
```
// Setup the execution configuration
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);
dim3 dimGrid(Width / TILE_WIDTH,
              Width / TILE_WIDTH);
```


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
 - TILE_WIDTH of 16 gives each SM 3 blocks, 768 threads (full capacity)
- 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

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 $2 \times 256 \times 4\text{B} = 2\text{KB}$ of shared memory.
 - The shared memory can potentially have up to 8 Thread Blocks actively executing
 - This allows up to $8 \times 512 = 4,096$ pending loads. (2 per thread, 256 threads per block)
 - The threading model limits the number of thread blocks to 3 so shared memory is not the limiting factor here
 - The next `TILE_WIDTH` 32 would lead to $2 \times 32 \times 32 \times 4\text{B} = 8\text{KB}$ 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 $(86.4/4) \times 16 = 347.6$ GFLOPS!

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];

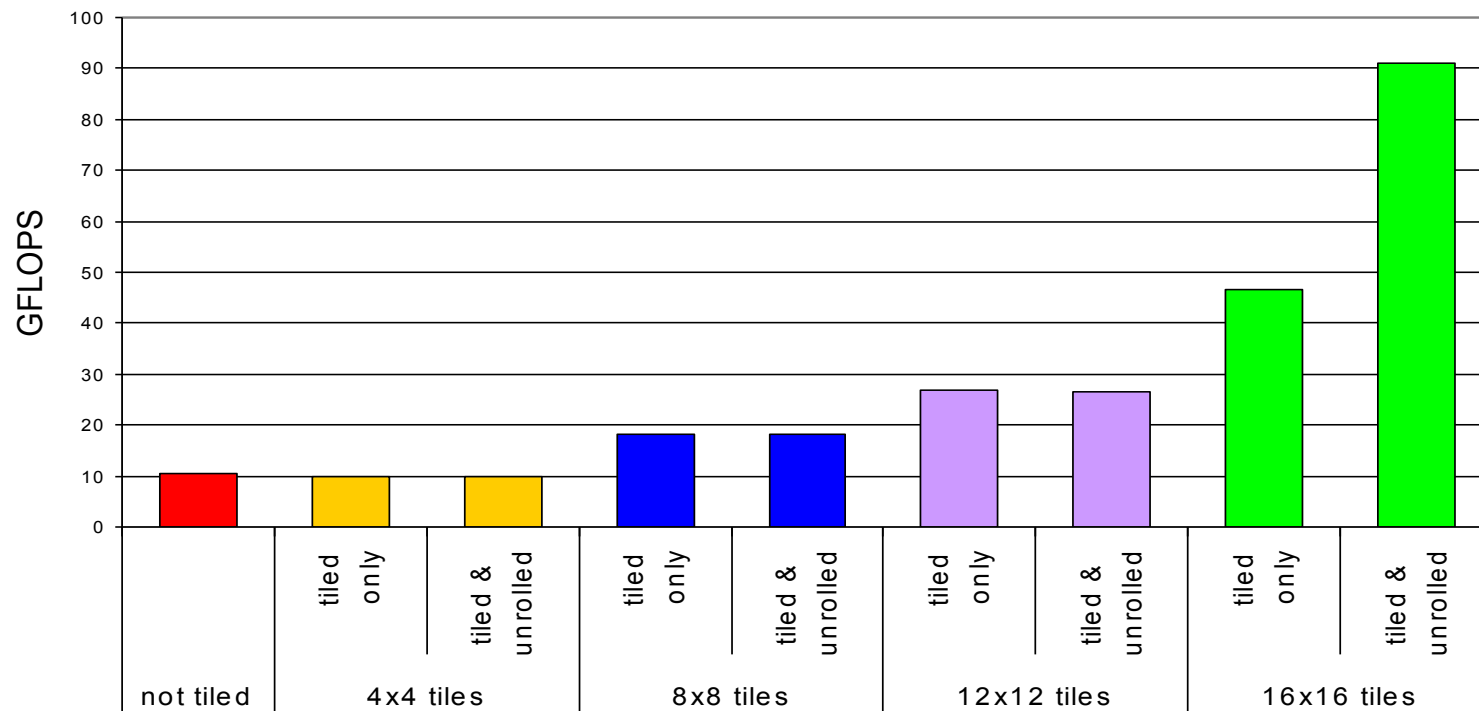
3.  int bx = blockIdx.x;  int by = blockIdx.y;
4.  int tx = threadIdx.x; int ty = threadIdx.y;

// 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;

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


11.    for (int k = 0; k < TILE_WIDTH; ++k)
12.        Pvalue += Mds[ty][k] * Nds[k][tx];
13.    Syncthreads();
14. }
13.    Pd[Row*Width+Col] = Pvalue;
}
```

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_GlbIVarPtr, bytes)`
 - transfer data from host to device – `cudaMemcpy(d_GlbIVarPtr, h_Gl...)`
 - execution configuration setup
 - kernel call – `kernelOne<<<execution configuration>>>(args...);`
 - transfer results from device to host – `cudaMemcpy(h_GlbIVarPtr,...)`
 - 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