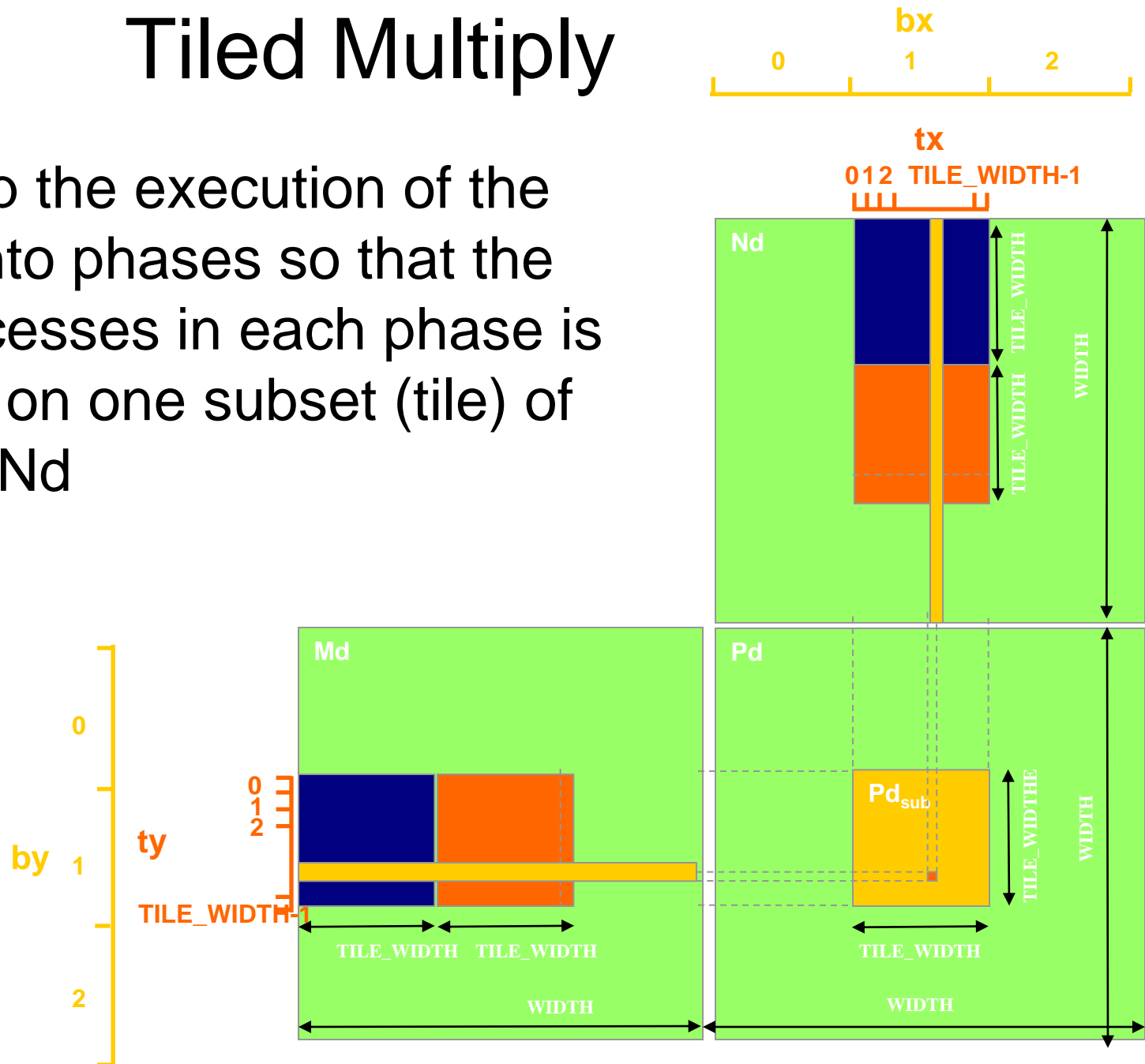
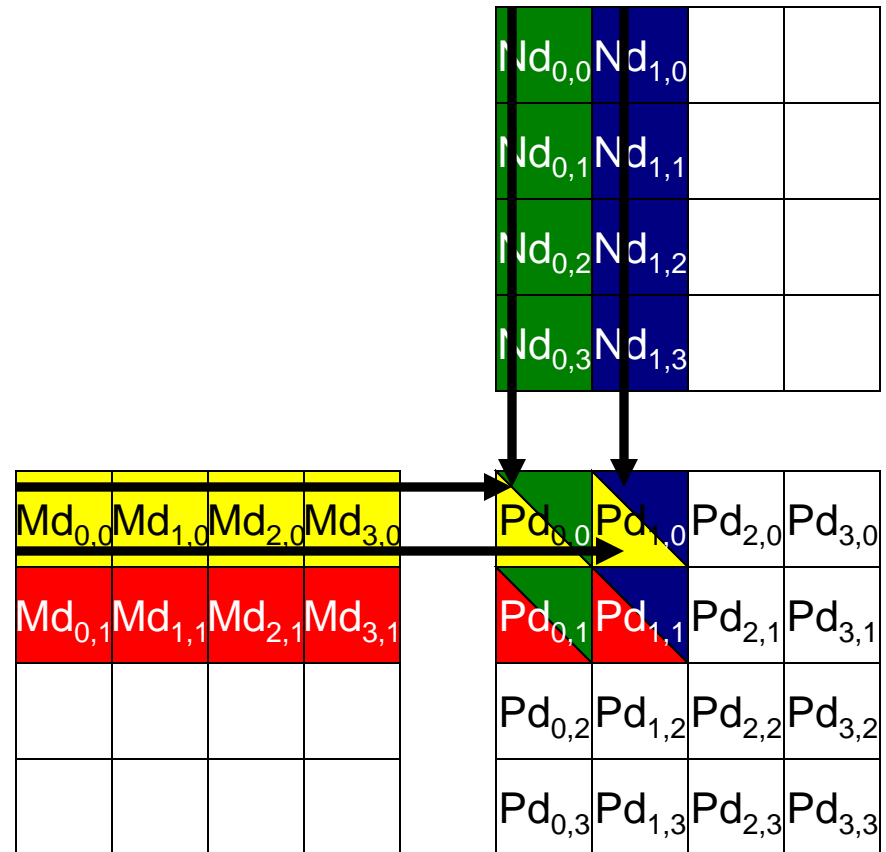


Tiled Multiply

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

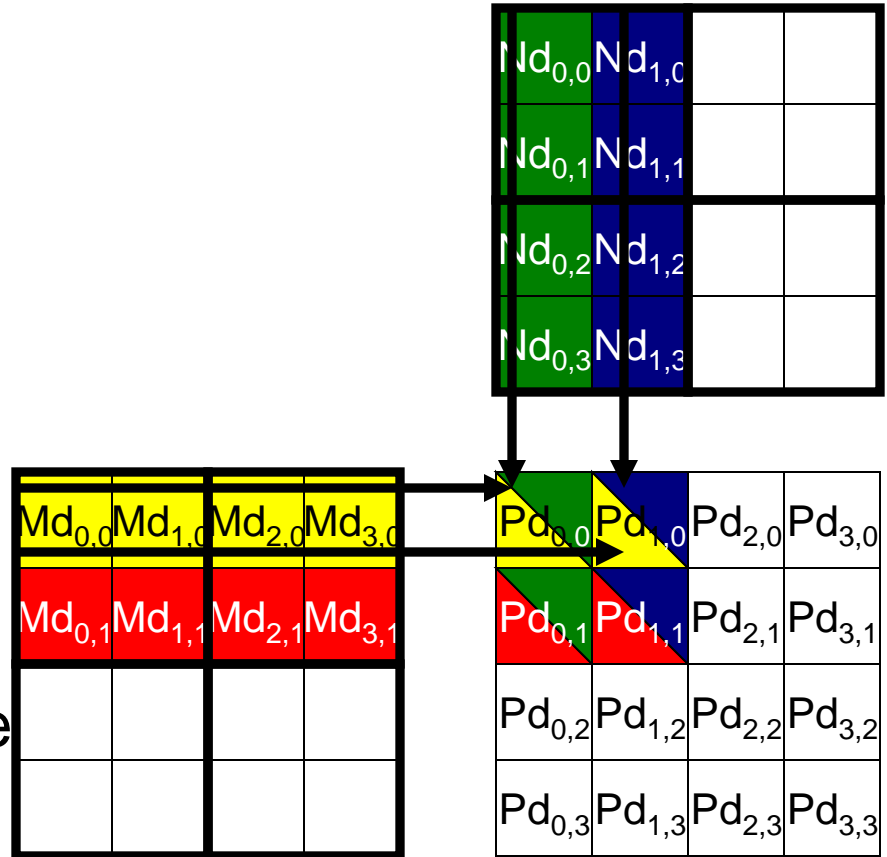
Access
order



$P_{0,0}$ 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

- 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();

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

CUDA Code – Kernel Execution Configuration

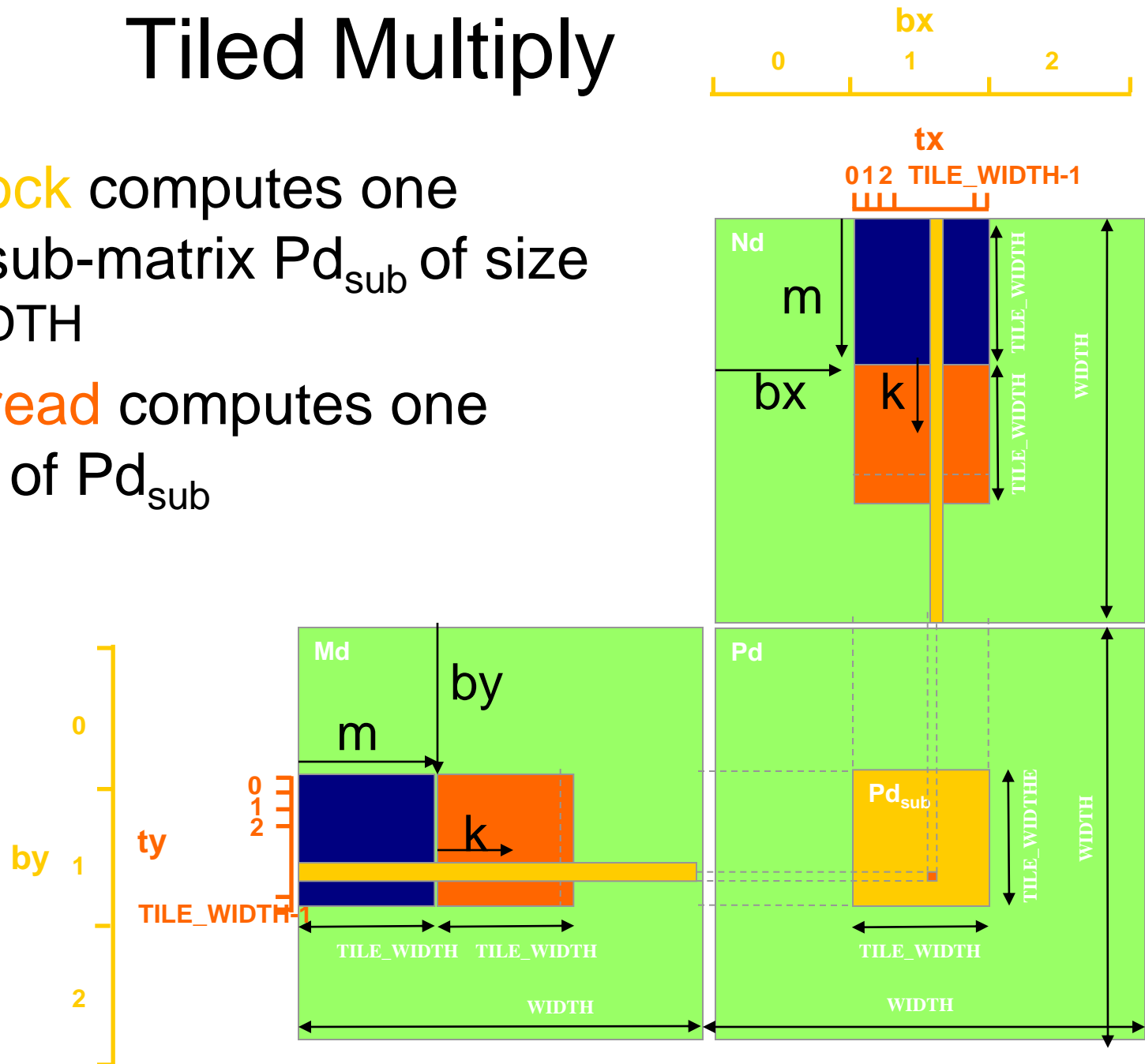
```
// Setup the execution configuration
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);
dim3 dimGrid(Width / TILE_WIDTH,
              Width / TILE_WIDTH);
```

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

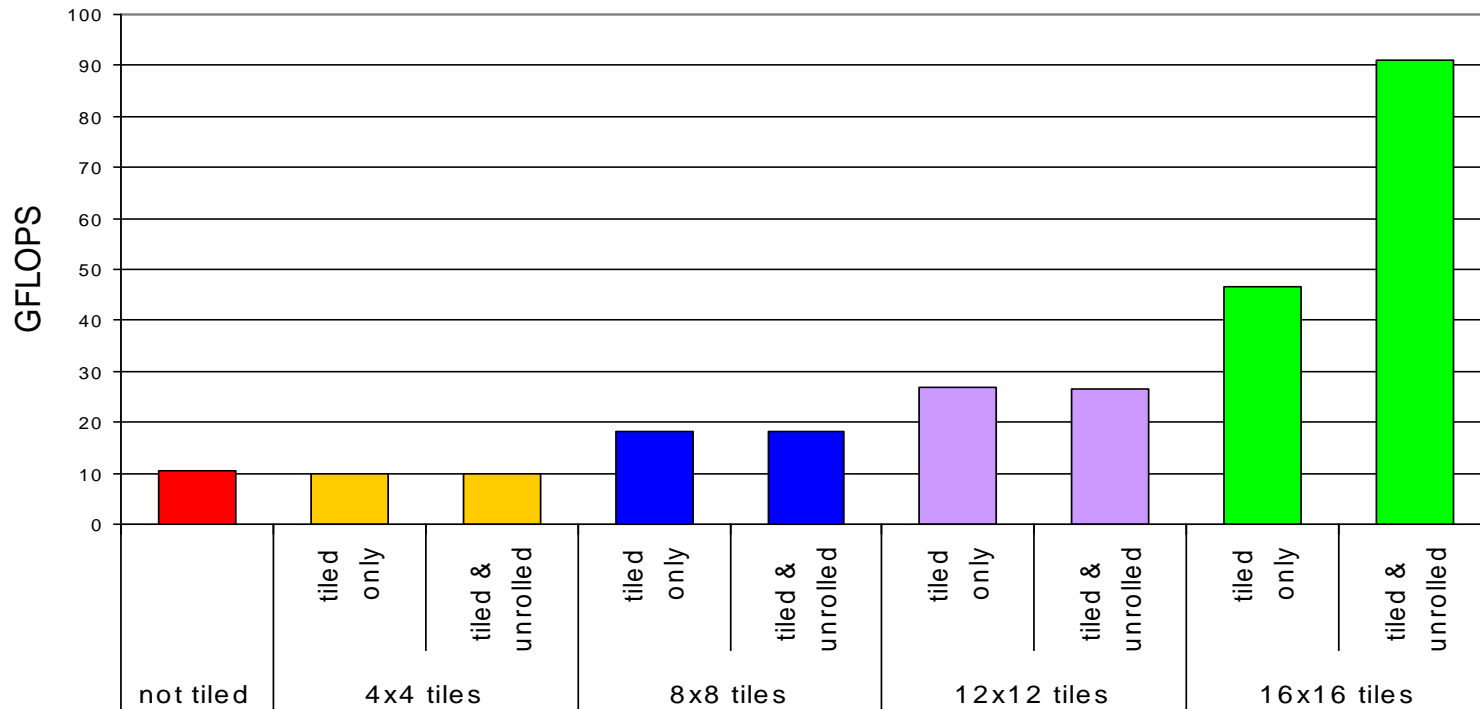
- Each **block** computes one square sub-matrix Pd_{sub} of size $TILE_WIDTH$
- Each **thread** computes one element of Pd_{sub}




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 $2 \times 256 \times 4B = 2KB$ 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 $2 \times 32 \times 32 \times 4B = 8KB$ 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 $(86.4/4) \times 16 = 347.6$ 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