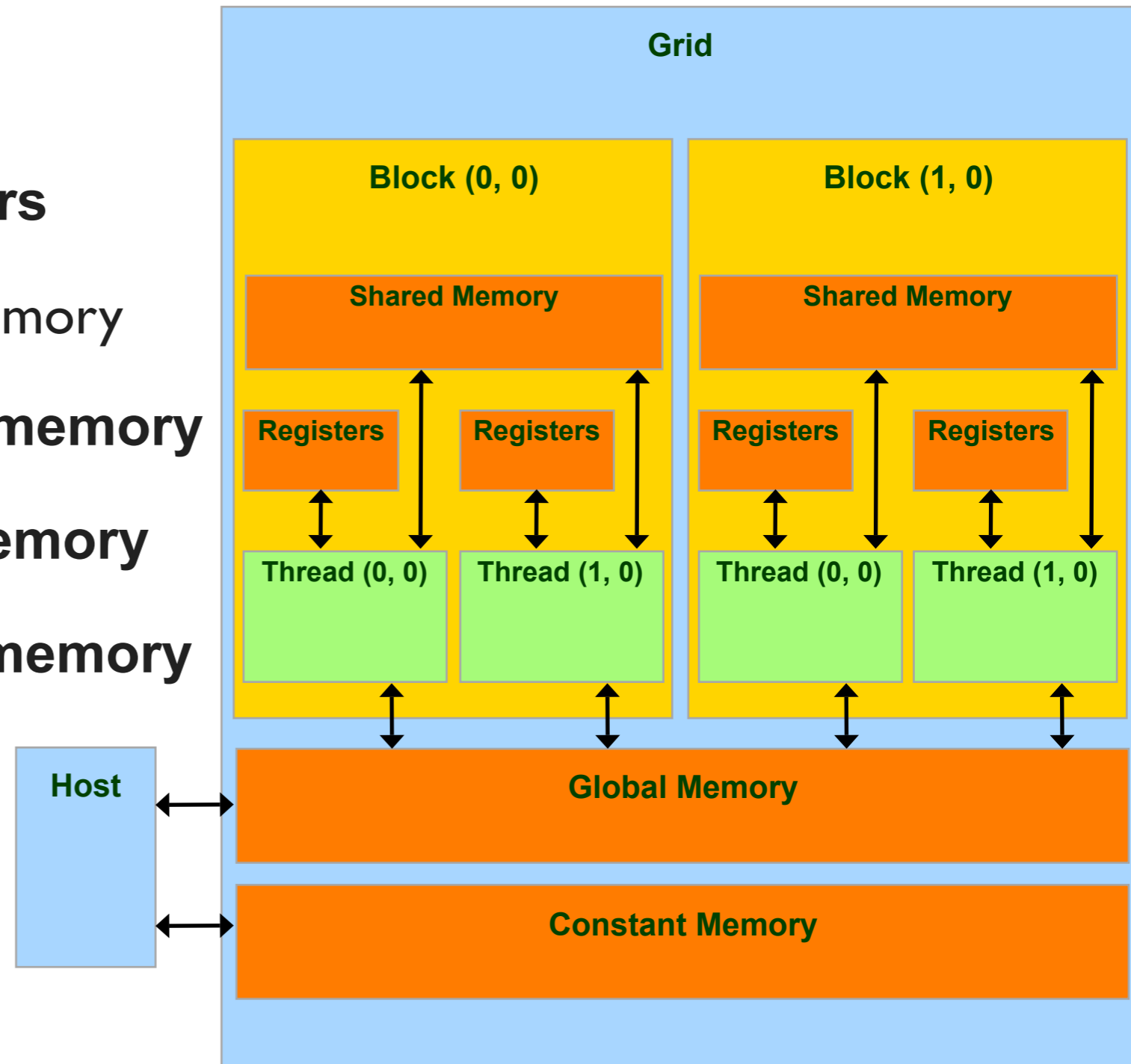


# CUDA Memory Model

Some material © David Kirk, NVIDIA and Wen-mei W. Hwu, 2007-2009 (used with permission)

# 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

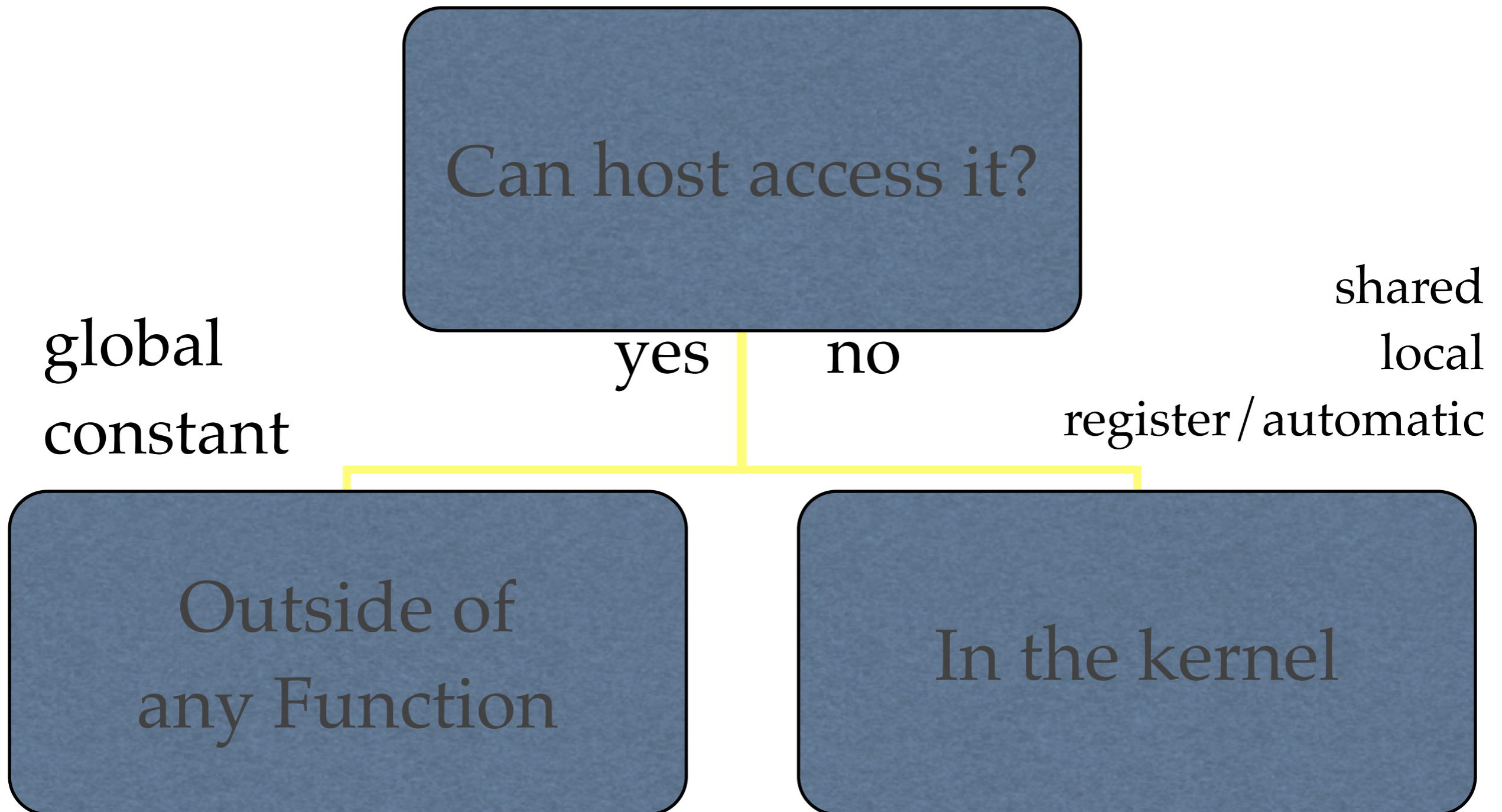
Variable declaration	Memory	Scope	Lifetime
<code>__device__ __local__ int LocalVar;</code>	local	thread	thread
<code>__device__ __shared__ int SharedVar;</code>	shared	block	block
<code>__device__ int GlobalVar;</code>	global	grid	application
<code>__device__ __constant__ int ConstantVar;</code>	constant	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 KernelFun (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
- 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)

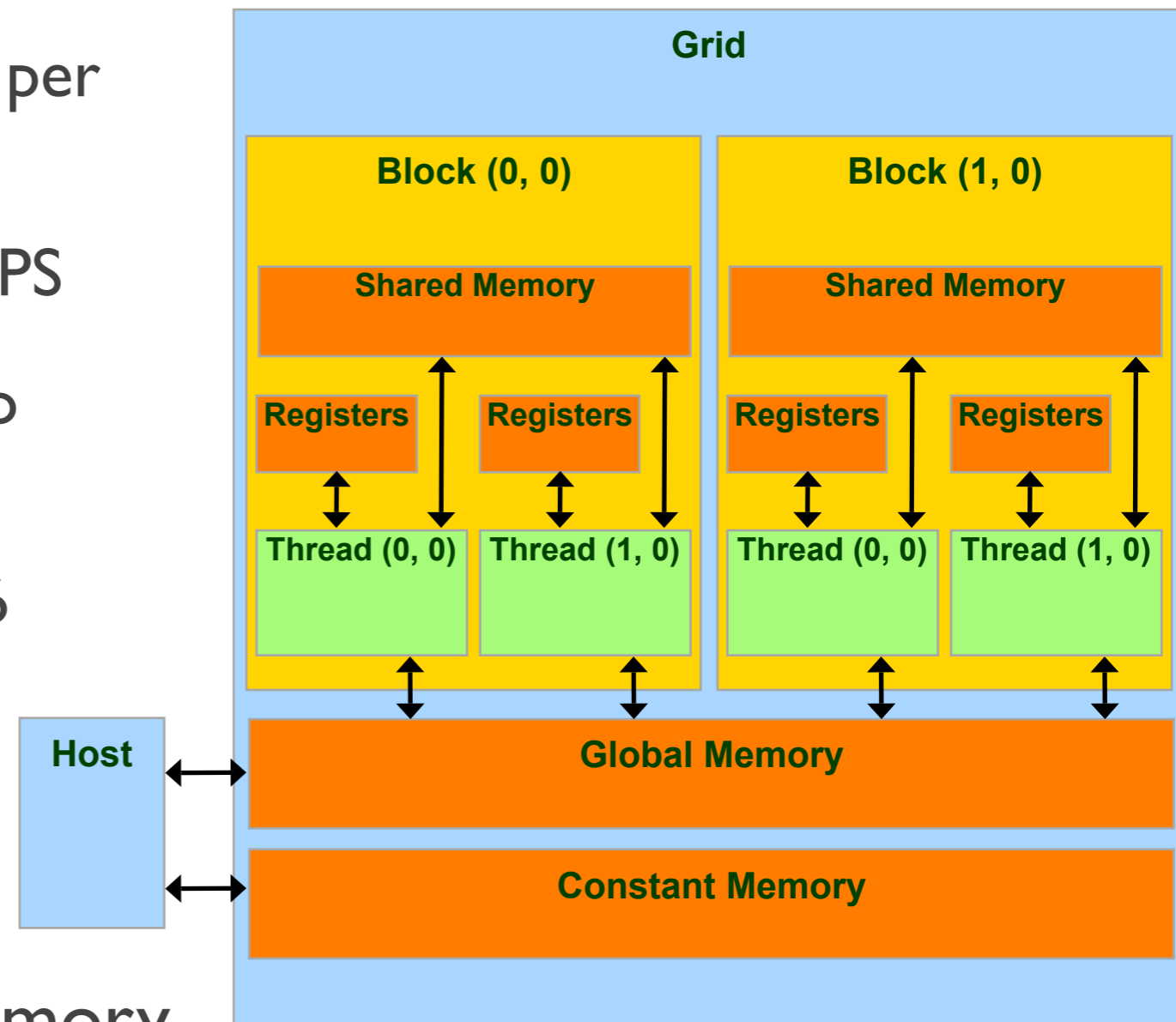
# Matrix Multiplication using Shared Memory

# Matrix Multiplication Kernel

```
__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;
}
```

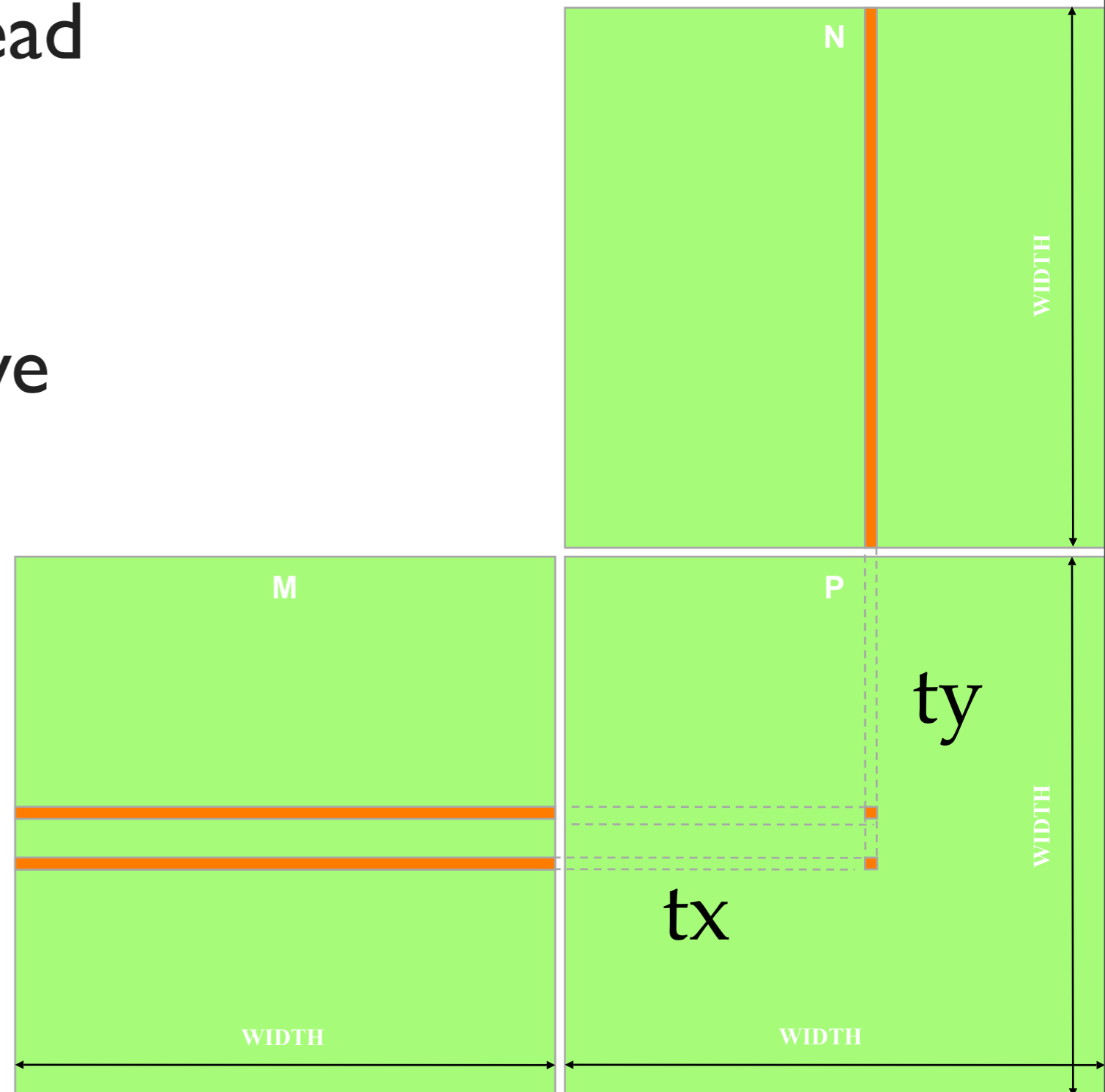
# How about performance on G80?

- 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
- 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 memory bandwidth



# Tiled matrix

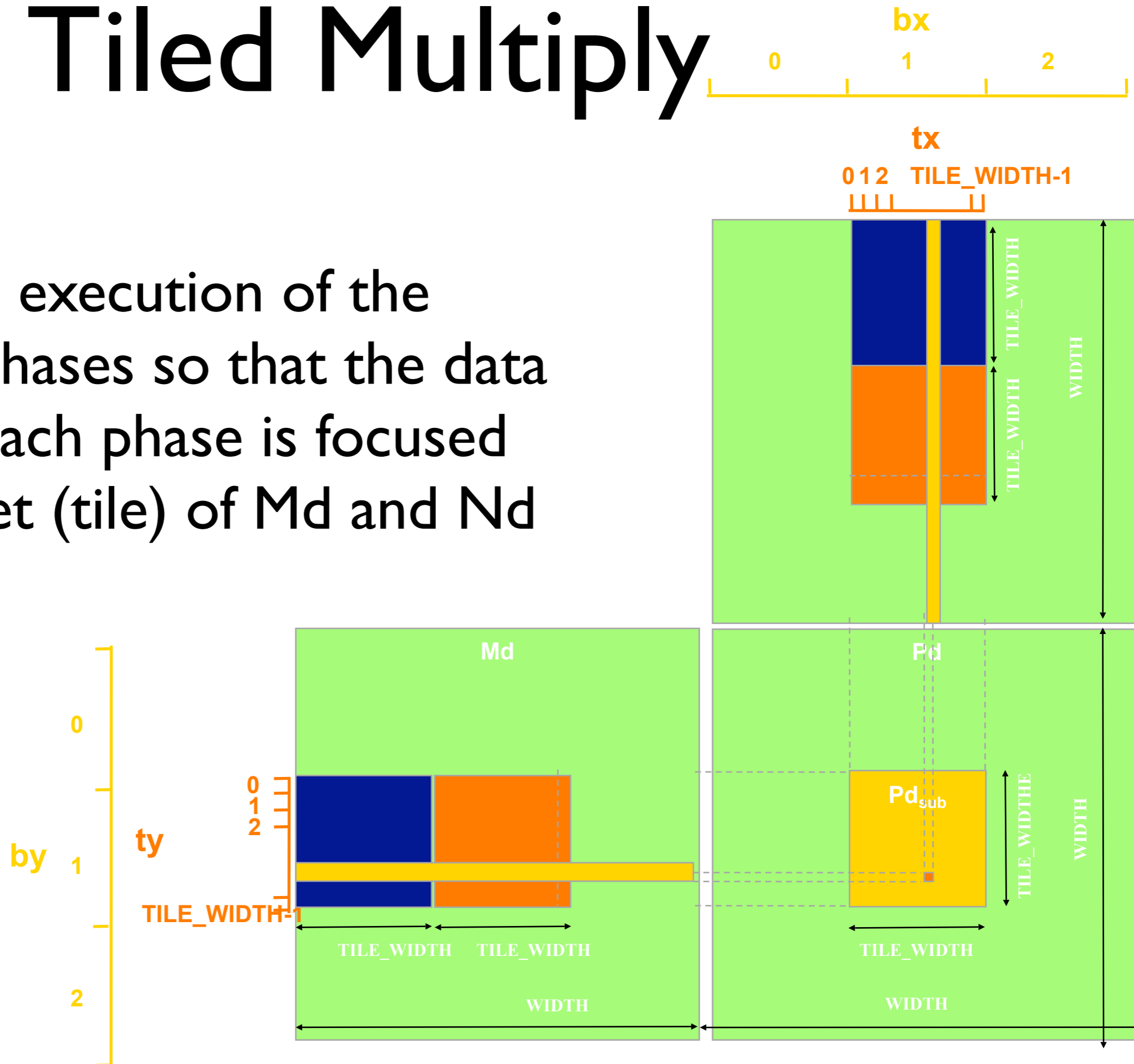
$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots \\ a_{2,1} & a_{2,2} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix} & \cdots \\ \vdots & \ddots \end{pmatrix} = \begin{pmatrix} A_{1,1} & \cdots \\ \vdots & \ddots \end{pmatrix}$$

$$(ab)_{i,j} = \sum_{k=1}^p a_{i,k} b_{k,j}$$

$$(AB)_{i,j} = \sum_{k=1}^{p/m} A_{i,k} B_{k,j}$$

# 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  $N_d$



# Tiled multiply

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

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

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

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

Access  
order



$P_{0,0}$ thread <sub>0,0</sub>	$P_{1,0}$ thread <sub>1,0</sub>	$P_{0,1}$ thread <sub>0,1</sub>	$P_{1,1}$ thread <sub>1,1</sub>
$M_{0,0} * N_{0,0}$	$M_{0,0} * N_{1,0}$	$M_{0,1} * N_{0,0}$	$M_{0,1} * N_{1,0}$
$M_{1,0} * N_{0,1}$	$M_{1,0} * N_{1,1}$	$M_{1,1} * N_{0,1}$	$M_{1,1} * N_{1,1}$
$M_{2,0} * N_{0,2}$	$M_{2,0} * N_{1,2}$	$M_{2,1} * N_{0,2}$	$M_{2,1} * N_{1,2}$
$M_{3,0} * N_{0,3}$	$M_{3,0} * N_{1,3}$	$M_{3,1} * N_{0,3}$	$M_{3,1} * N_{1,3}$

# 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  $16 * 16 = 256$  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 row and column of Pd to work on
    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;
    float Pvalue = 0;
```

```

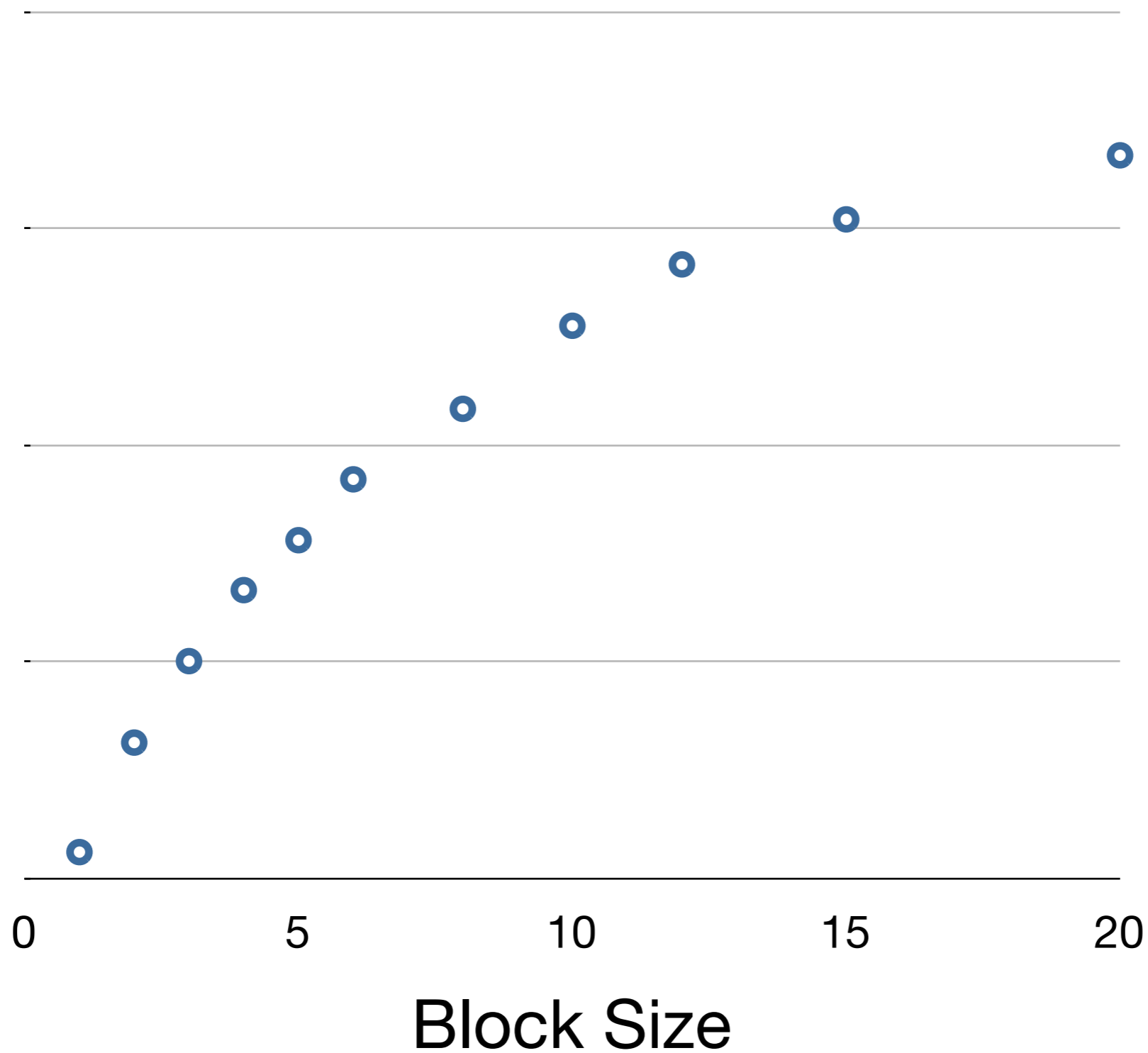
// Loop over Md and Nd tiles required to compute Pd
for (int m = 0; m < Width/TILE_WIDTH; ++m) {
// Collaborative loading of Md and Nd tiles into
  shared memory
  Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
  Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
  __syncthreads();
  for (int k =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

```

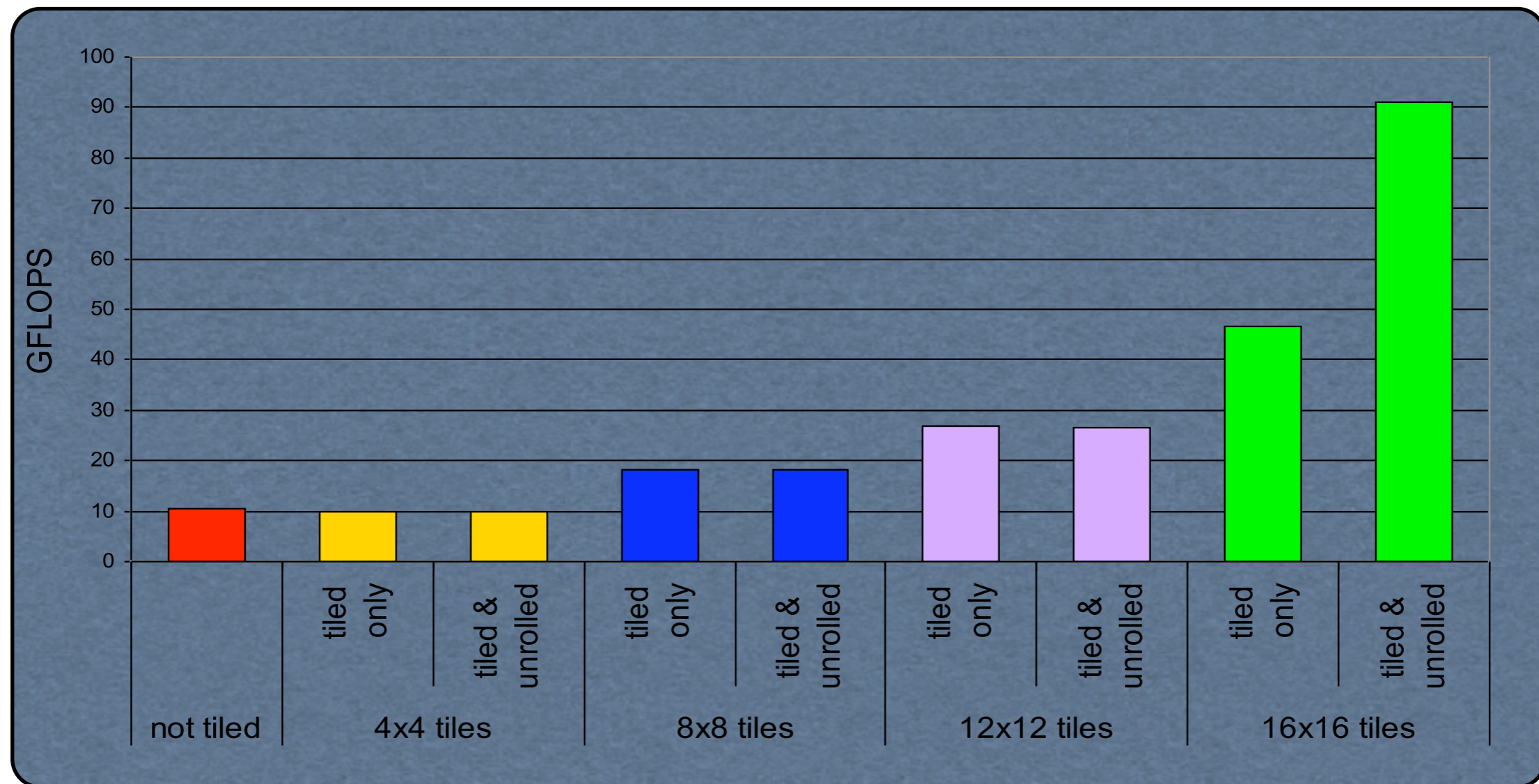
# G80 Shared Memory and Threading

- Each SM in G80 has 16KB shared memory
  - shared memory 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!

# Block size effect on throughput



# Loop unrolling effects



# CUDA Memory Model

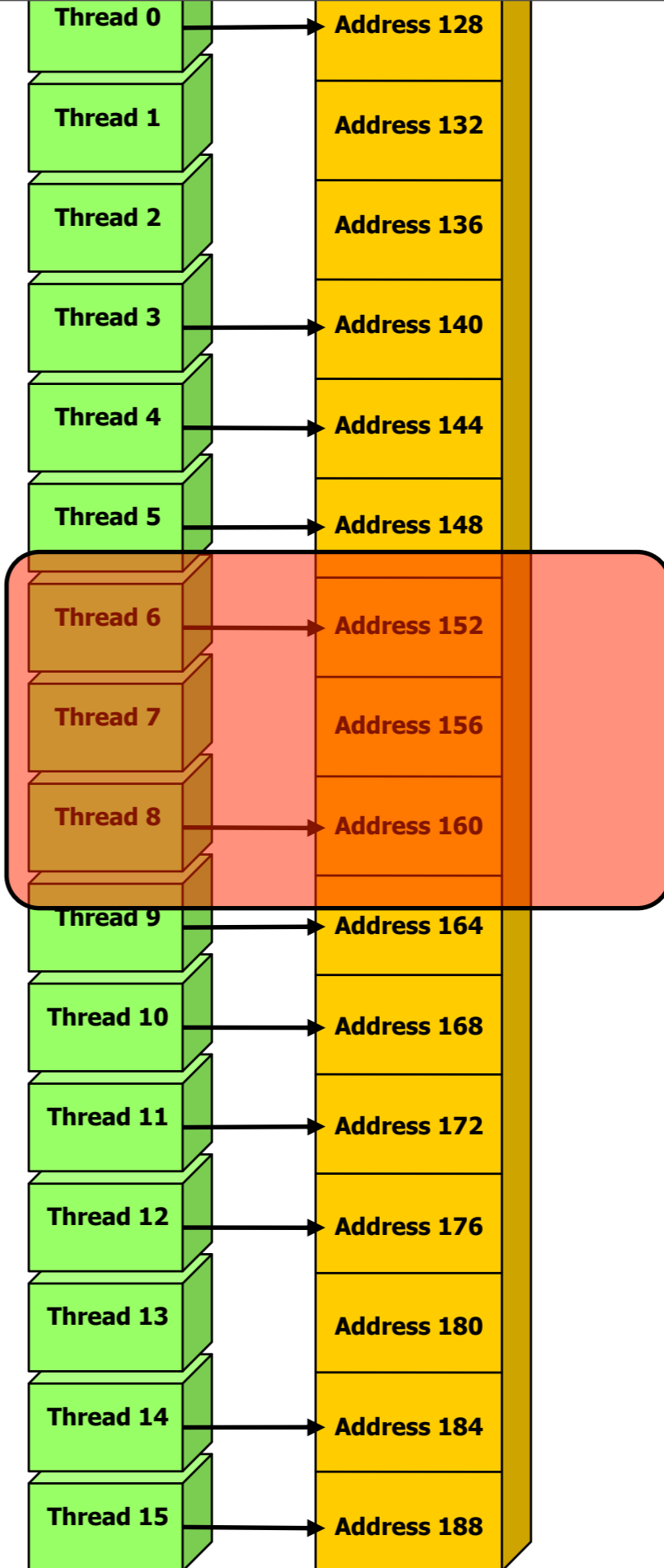
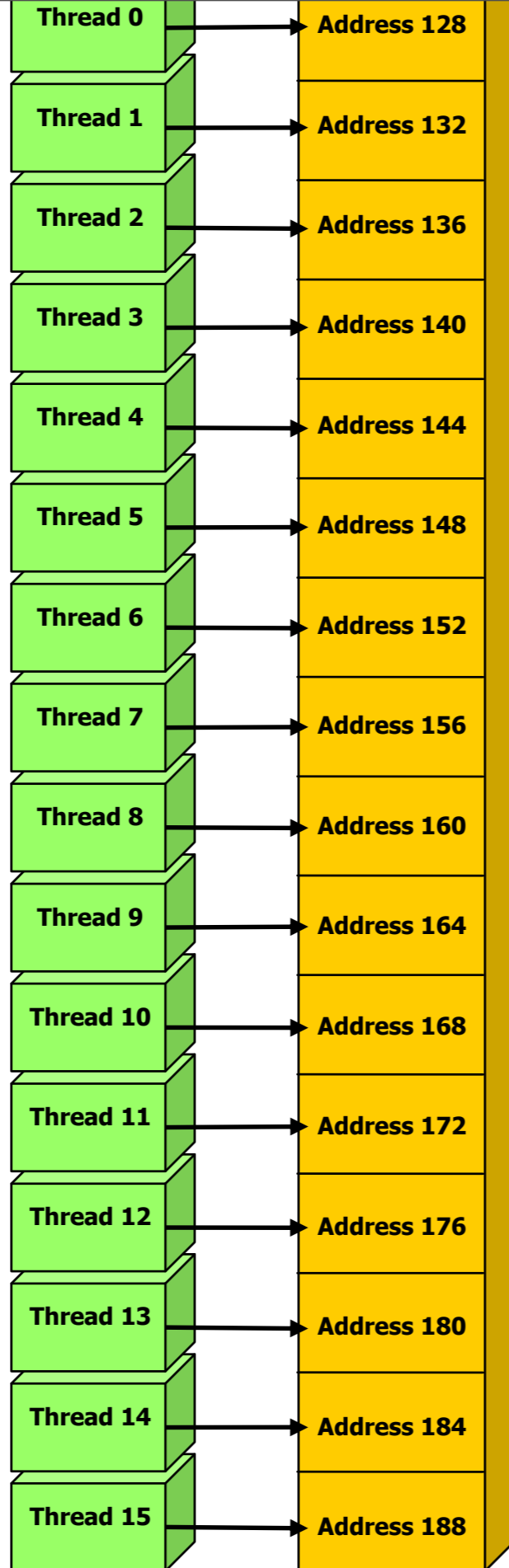
# Global Memory

- memory not cached so using the *right* access pattern is crucial
- memory access (to global memory) is very costly
- variables must be *aligned* for efficient access (usually are)
- simultaneous memory accesses in *half-warps* can be *coalesced* in one transaction
- coalescing depends on device compute capabilities

## Coalescing on Devices with Compute Capability 1.0 and 1.1

The global memory access by all threads of a half-warp is coalesced into one or two memory transactions if it satisfies the following three conditions:

- ❑ Threads must access
  - ❑ Either 32-bit words, resulting in one 64-byte memory transaction,
  - ❑ Or 64-bit words, resulting in one 128-byte memory transaction,
  - ❑ Or 128-bit words, resulting in two 128-byte memory transactions;
- ❑ All 16 words must lie in the same segment of size equal to the memory transaction size (or twice the memory transaction size when accessing 128-bit words);
- ❑ Threads must access the words in sequence: The  $k^{\text{th}}$  thread in the half-warp must access the  $k^{\text{th}}$  word.



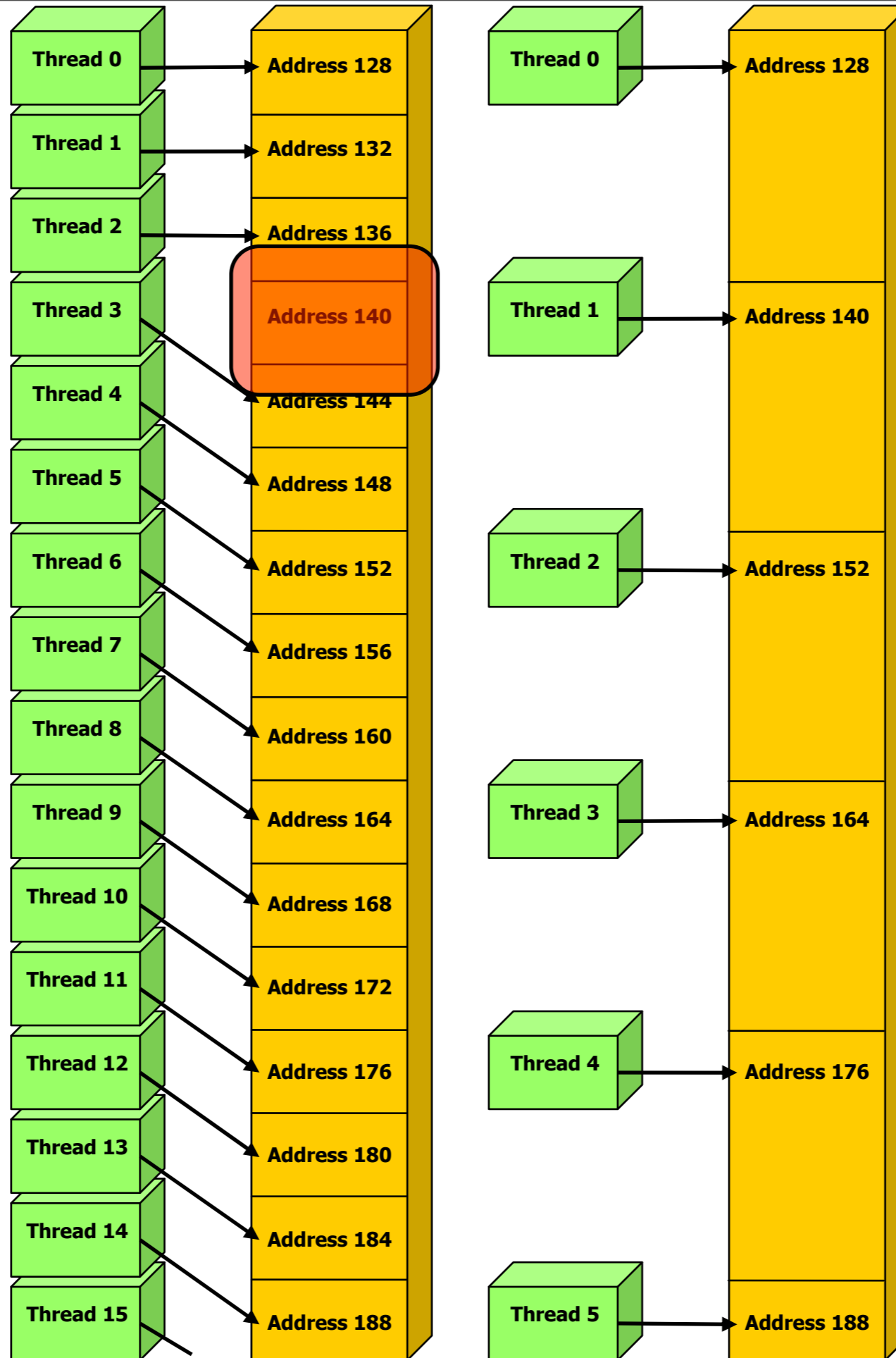
Left: coalesced float memory access, resulting in a single memory transaction.

Right: coalesced float memory access (divergent warp), resulting in a single memory transaction.



Left: non-sequential `float` memory access, resulting in 16 memory transactions.

Right: access with a misaligned starting address, resulting in 16 memory transactions.



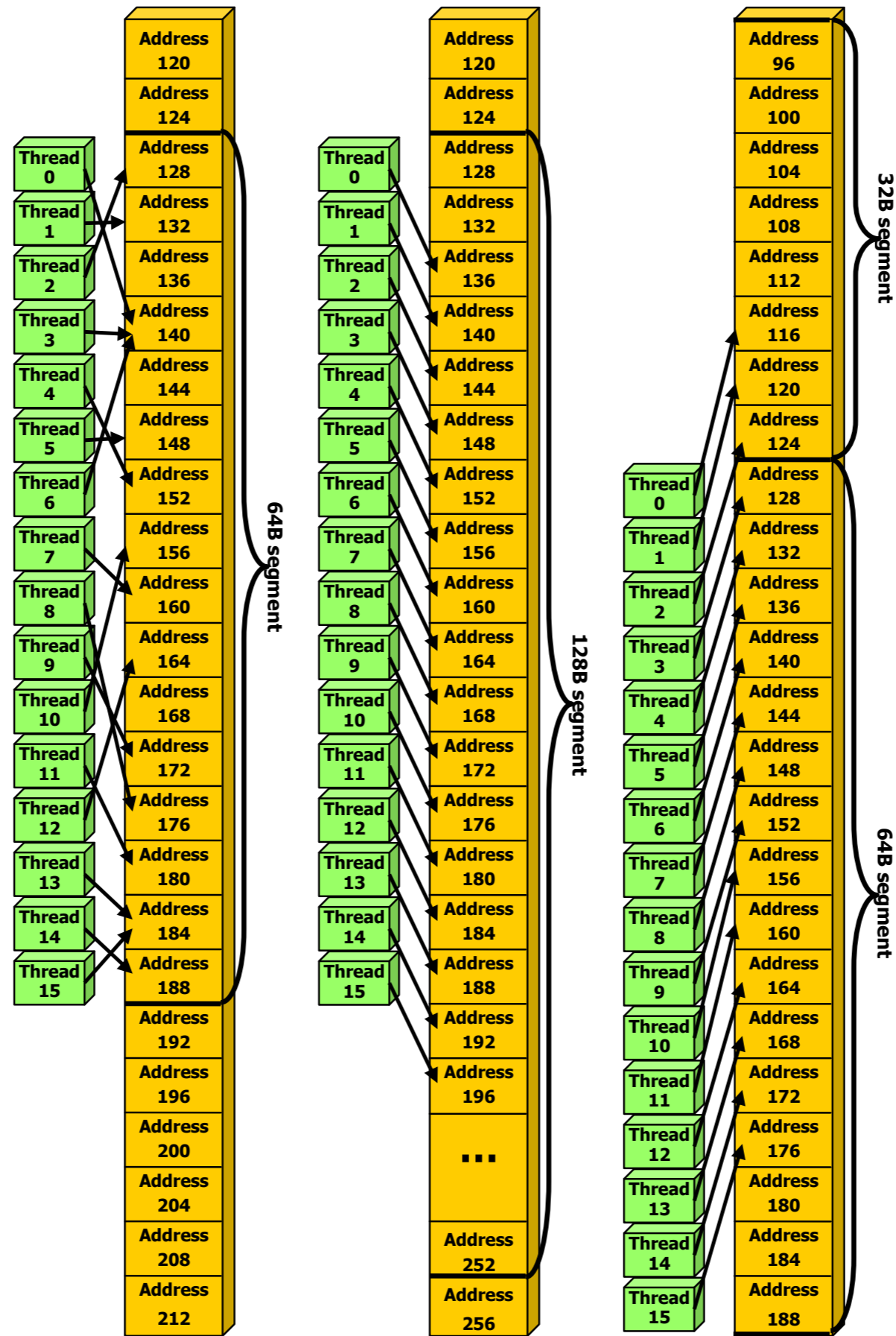
Left: non-contiguous `float` memory access, resulting in 16 memory transactions.

Right: non-coalesced `float3` memory access, resulting in 16 memory transactions.

## Coalescing on Devices with Compute Capability 1.2 and Higher

The global memory access by all threads of a half-warp is coalesced into a single memory transaction as soon as the words accessed by all threads lie in the same segment of size equal to:

- ❑ 32 bytes if all threads access 8-bit words,
- ❑ 64 bytes if all threads access 16-bit words,
- ❑ 128 bytes if all threads access 32-bit or 64-bit words.



Left: random `float` memory access within a 64B segment, resulting in one memory transaction.  
Center: misaligned `float` memory access, resulting in one transaction.  
Right: misaligned `float` memory access, resulting in two transactions.

# Shared memory

- as fast as register access
- if there are no bank conflicts
  - 16 equally-sized modules in 1.x
- $n$  addresses in  $n$  banks = one transaction
  - $n$  addresses in one bank =  $n$  transactions!
- bank conflict can be avoided on read-only access
- successive 32-bit words are assigned to successive banks
- each bank has a bandwidth of 32 bits per two clock cycles.

# More on bank conflicts

- if warp size is 32 and number of banks 16 a shared memory request for a warp is split into one request for the first half of the warp and one request for the second half of the warp
- so: no bank conflict between threads belonging to the different halves of the same warp!

# Typical case

```
__shared__ float shared[32];  
float data = shared[BaseIndex + s * tid];
```

- tid vs. tid+n : bank conflicts?
- $s*m \% 16 \neq s*n \% 16$ 
  - for any  $m \neq n$ ,  $|m-n| < 16$
- $s=1$  (the only safe stride!)

# Beware of data size!

Char is too small

```
__shared__ char shared[32];  
char data = shared[BaseIndex + tid];
```

because **shared[0]**, **shared[1]**, **shared[2]**, and **shared[3]**, for example, belong to the same bank. There are no bank conflicts however, if the same array is accessed the following way:

```
char data = shared[BaseIndex + 4 * tid];
```