

# **CUDA**

**CSCI 4830/7000**

**Advanced Computer Graphics**

**Spring 2010**



# CUDA

- “Compute Unified Device Architecture”
- General purpose programming model
  - User kicks off batches of threads on the GPU
  - GPU = dedicated super-threaded, massively data parallel co-processor
- Targeted software stack
  - Compute oriented drivers, language, and tools
- Driver for loading computation programs into GPU
  - Standalone Driver - Optimized for computation
  - Interface designed for compute –graphics-free API
  - Data sharing with OpenGL buffer objects
  - Guaranteed maximum download & readback speeds
  - Explicit GPU memory management



# Parallel Computing on a GPU

- 8-series GPUs deliver 25 to 200+ GFLOPS on compiled parallel C applications
  - Available in laptops, desktops, and clusters
- GPU parallelism is doubling every year
- Programming model scales transparently
- Programmable in C with CUDA tools
- Multithreaded SPMD model uses application data parallelism and thread parallelism



**GeForce 8800**



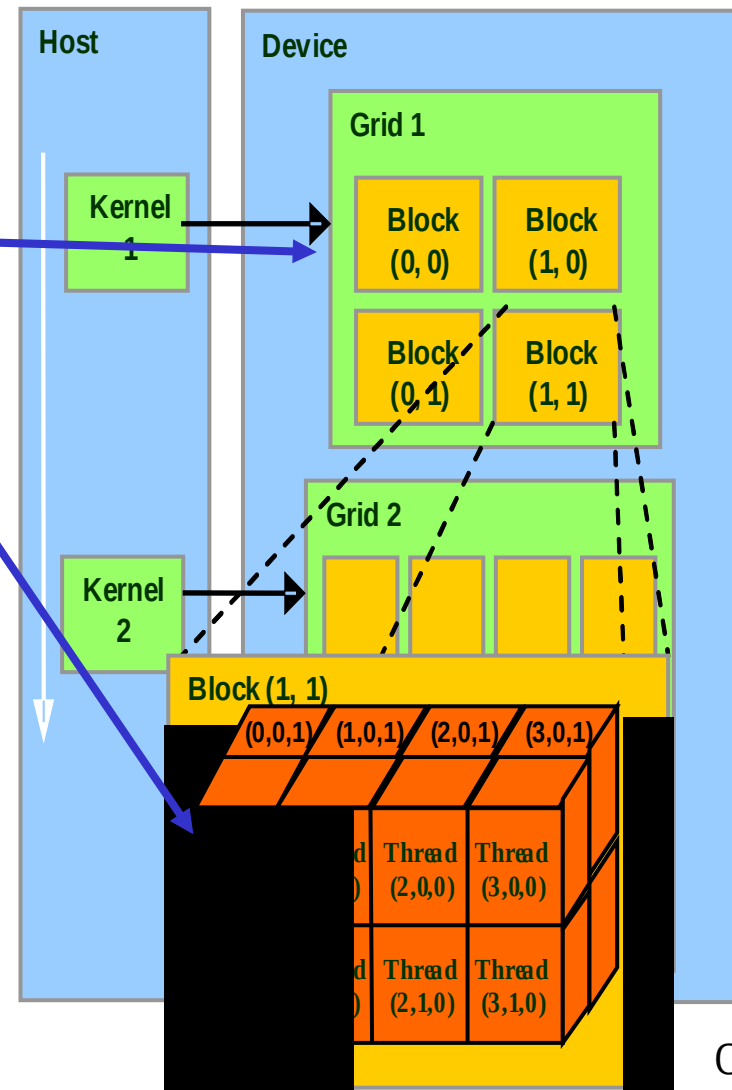
**Tesla D870**



**Tesla S870**

# Block IDs and Thread IDs

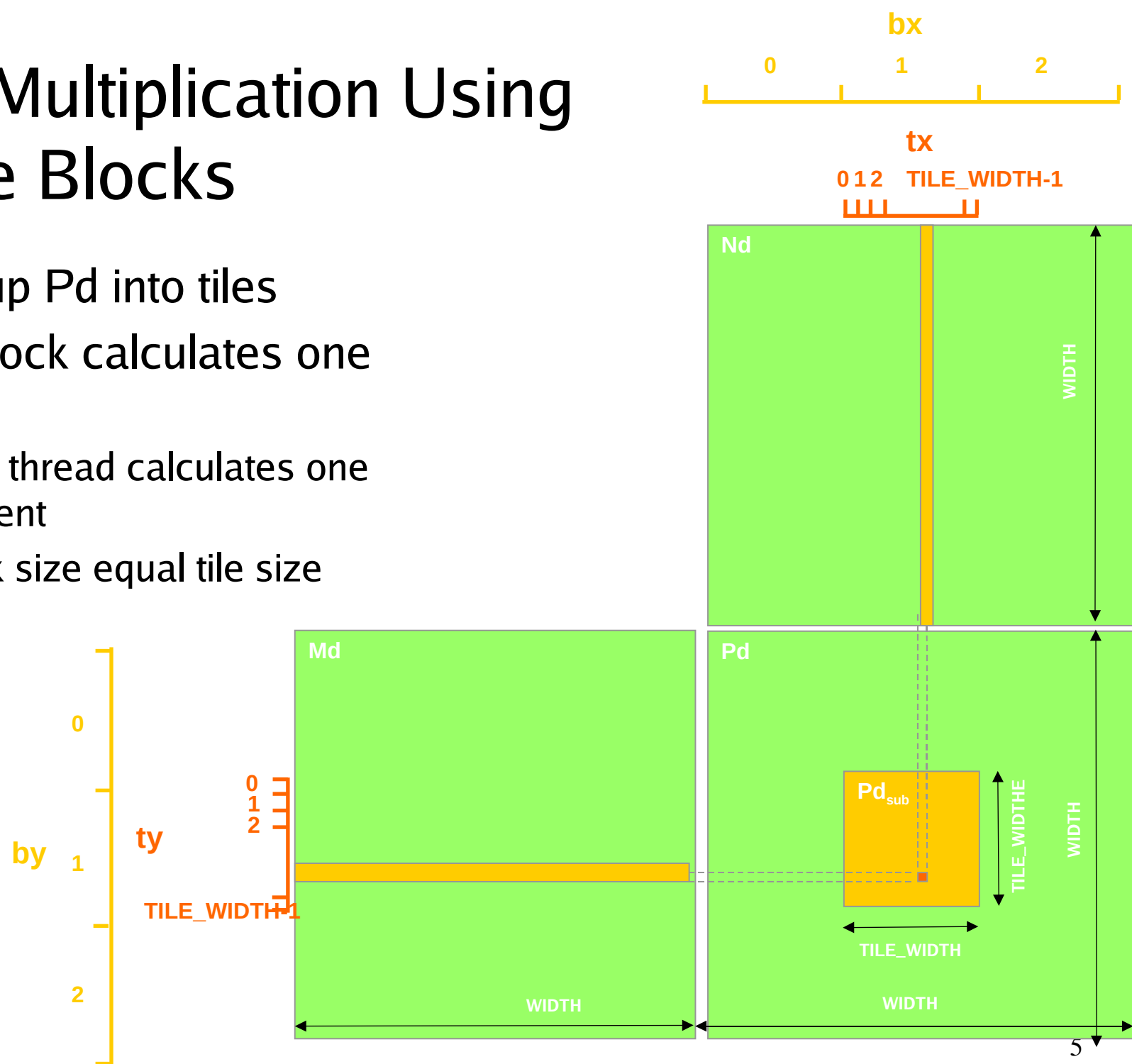
- Each thread uses IDs to decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - ...



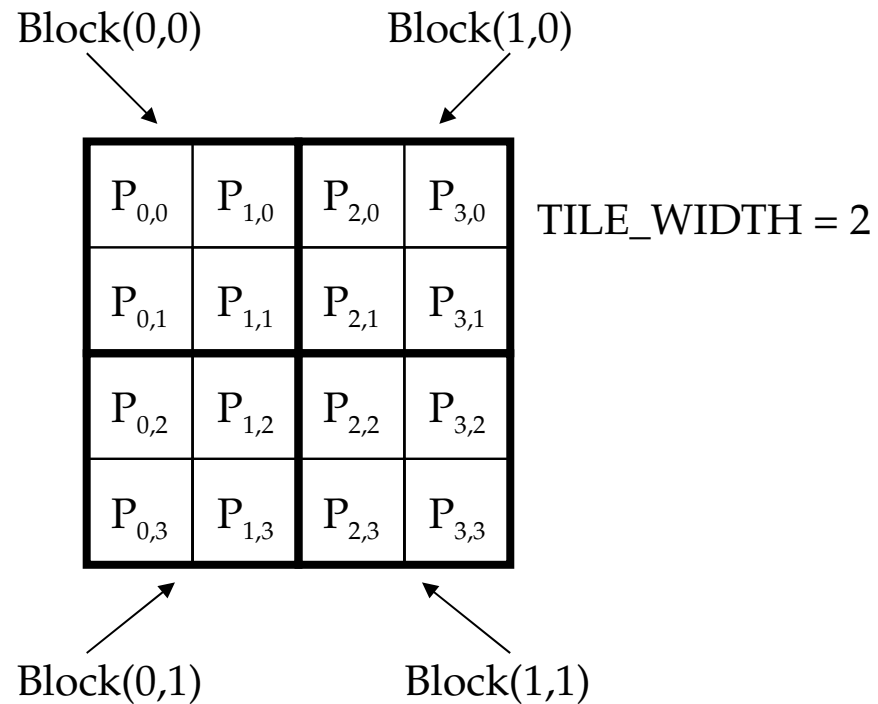
Courtesy: NDVIA

# Matrix Multiplication Using Multiple Blocks

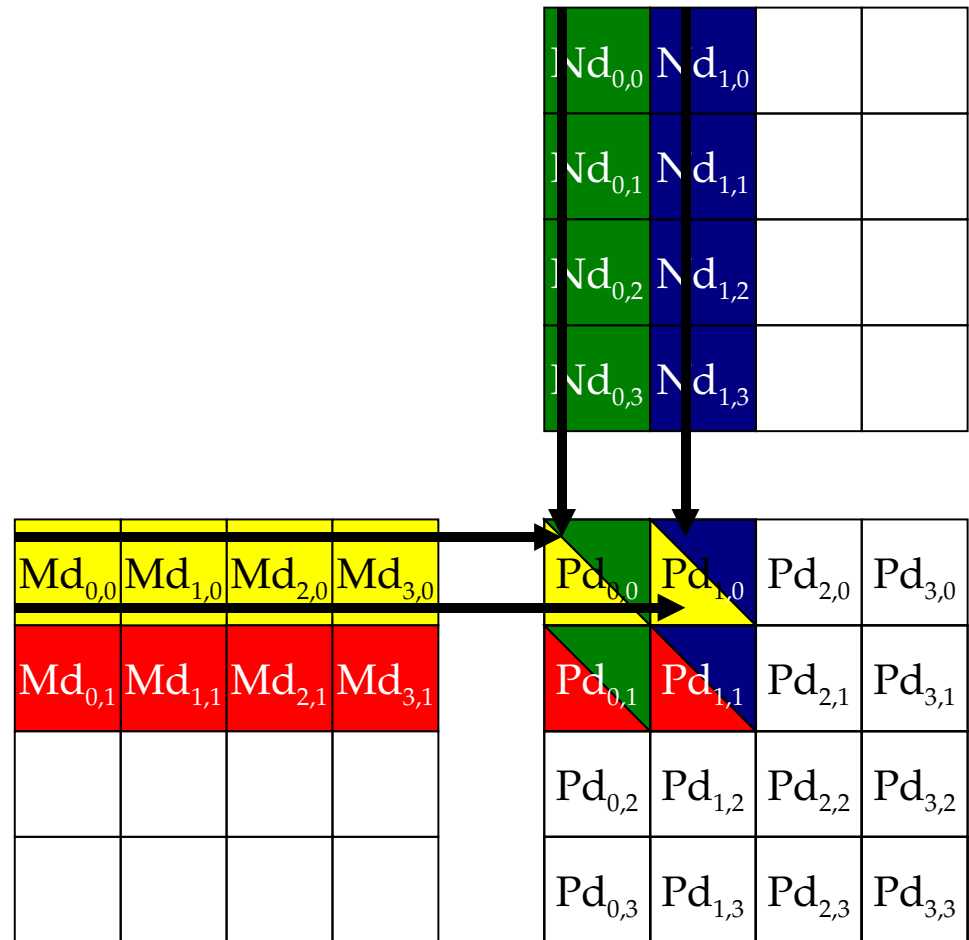
- Break-up  $P_d$  into tiles
- Each block calculates one tile
  - Each thread calculates one element
  - Block size equal tile size



# A Small Example



# A Small Example: Multiplication



# Revised 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 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;
}
```



# Revised Step 5: Kernel Invocation (Host-side Code)

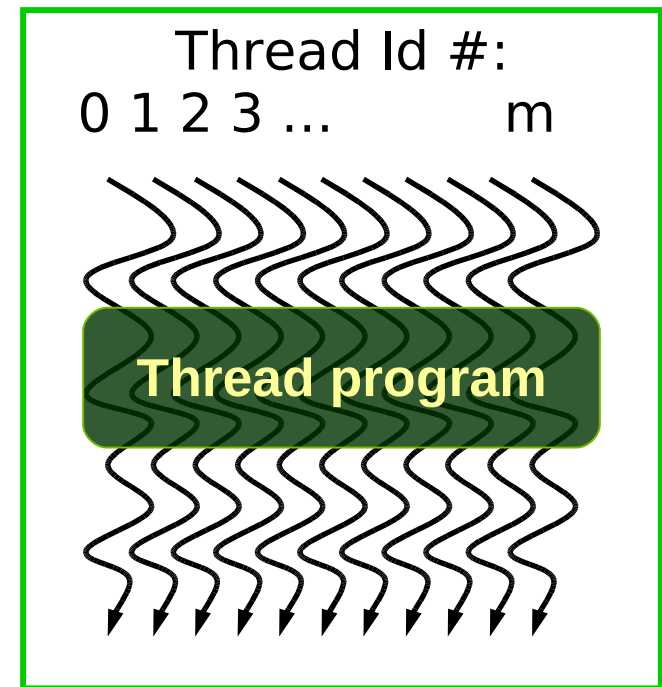
```
// Setup the execution configuration
dim3 dimGrid(Width/TILE_WIDTH, Width/TILE_WIDTH);
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);

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

# CUDA Thread Block

- All threads in a block execute the same kernel program (SPMD)
- Programmer declares block:
  - Block size 1 to 512 concurrent threads
  - Block shape 1D, 2D, or 3D
  - Block dimensions in threads
- Threads have **thread id** numbers within block
  - Thread program uses **thread id** to select work and address shared data
- Threads in the same block share data and synchronize while doing their share of the work
- Threads in different blocks cannot cooperate
  - Each block can execute in any order relative to other blocs!

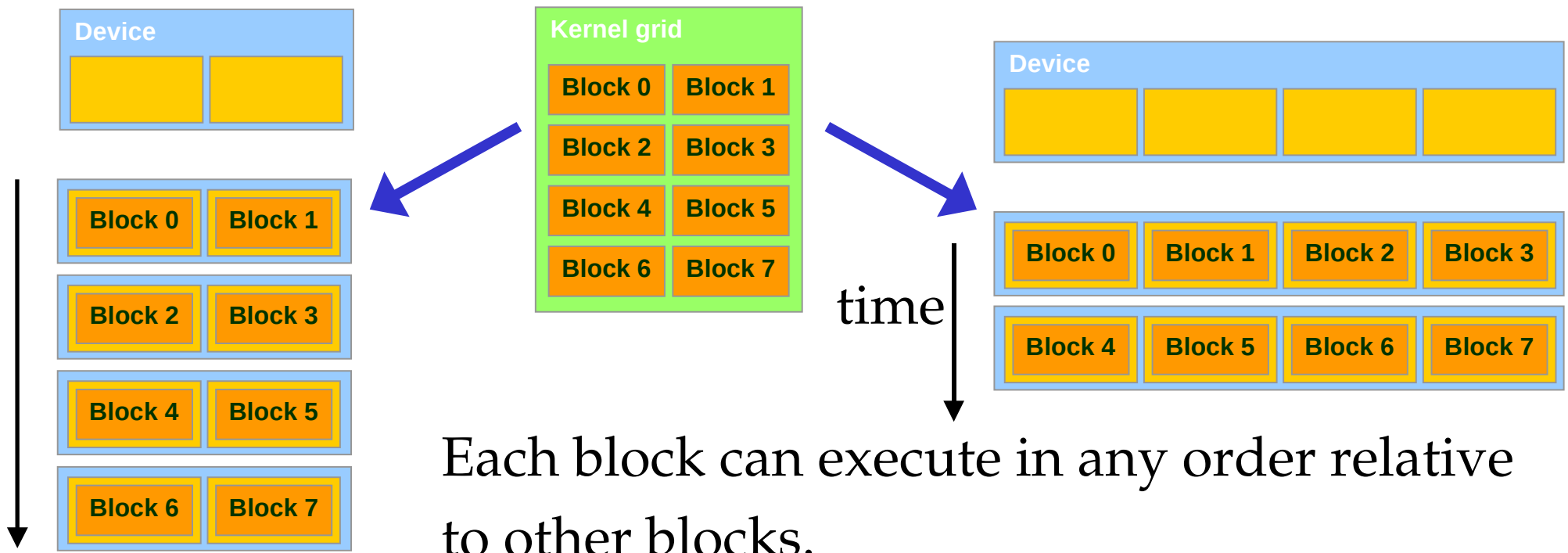
## CUDA Thread Block



Courtesy: John Nickolls, NVIDIA

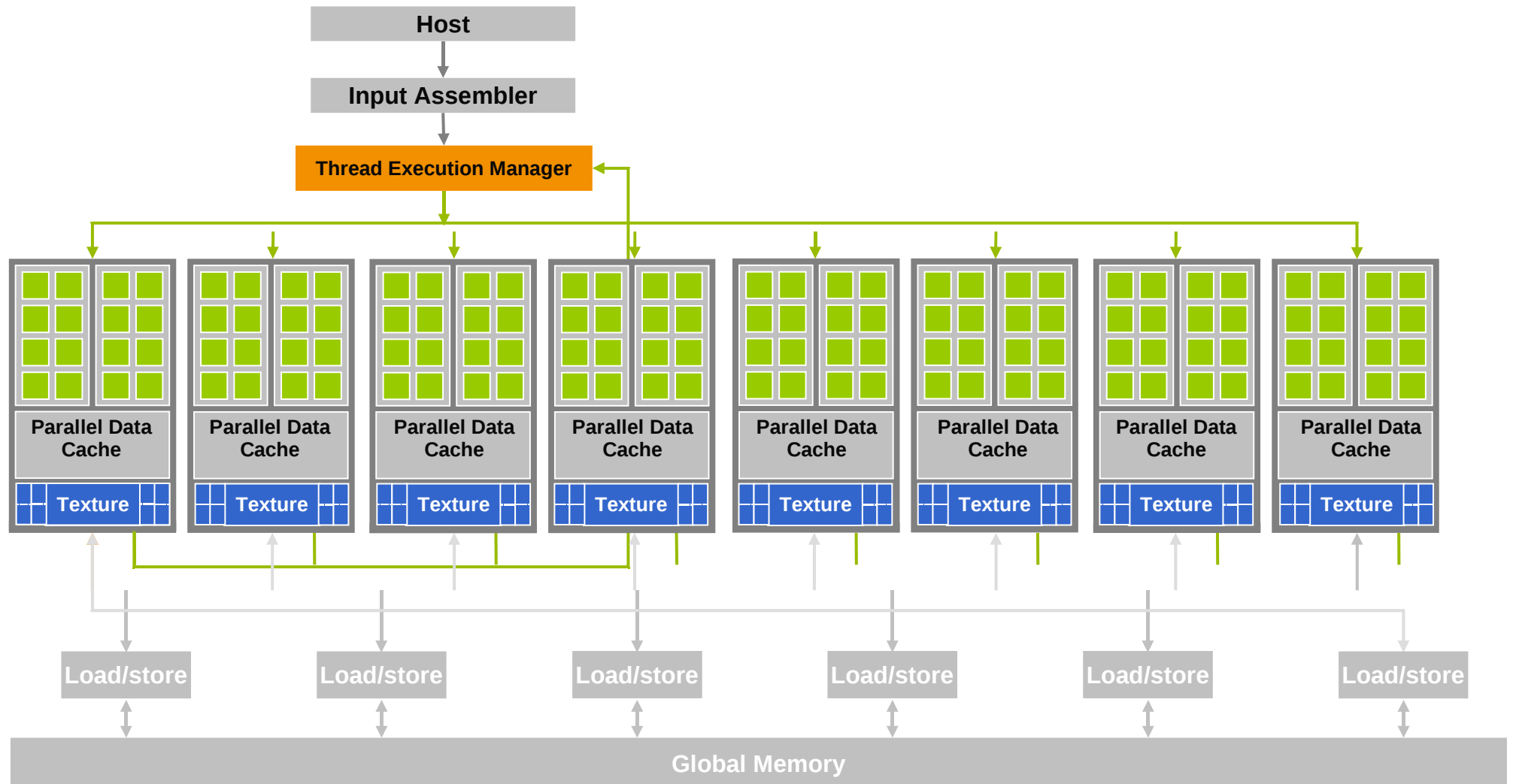
# Transparent Scalability

- Hardware is free to assign blocks to any processor at any time
  - A kernel scales across any number of parallel processors

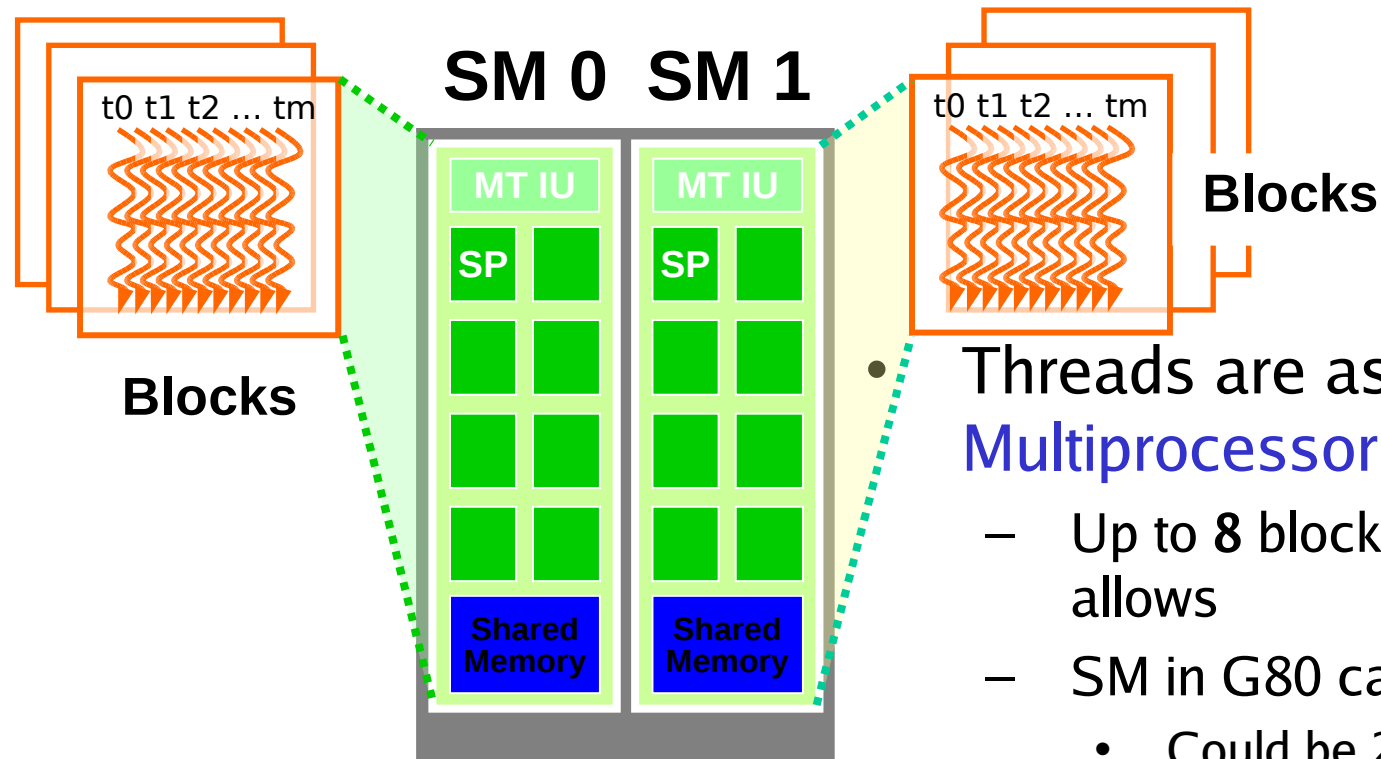


# G80 CUDA mode –A Review

- Processors execute computing threads
- New operating mode/HW interface for computing



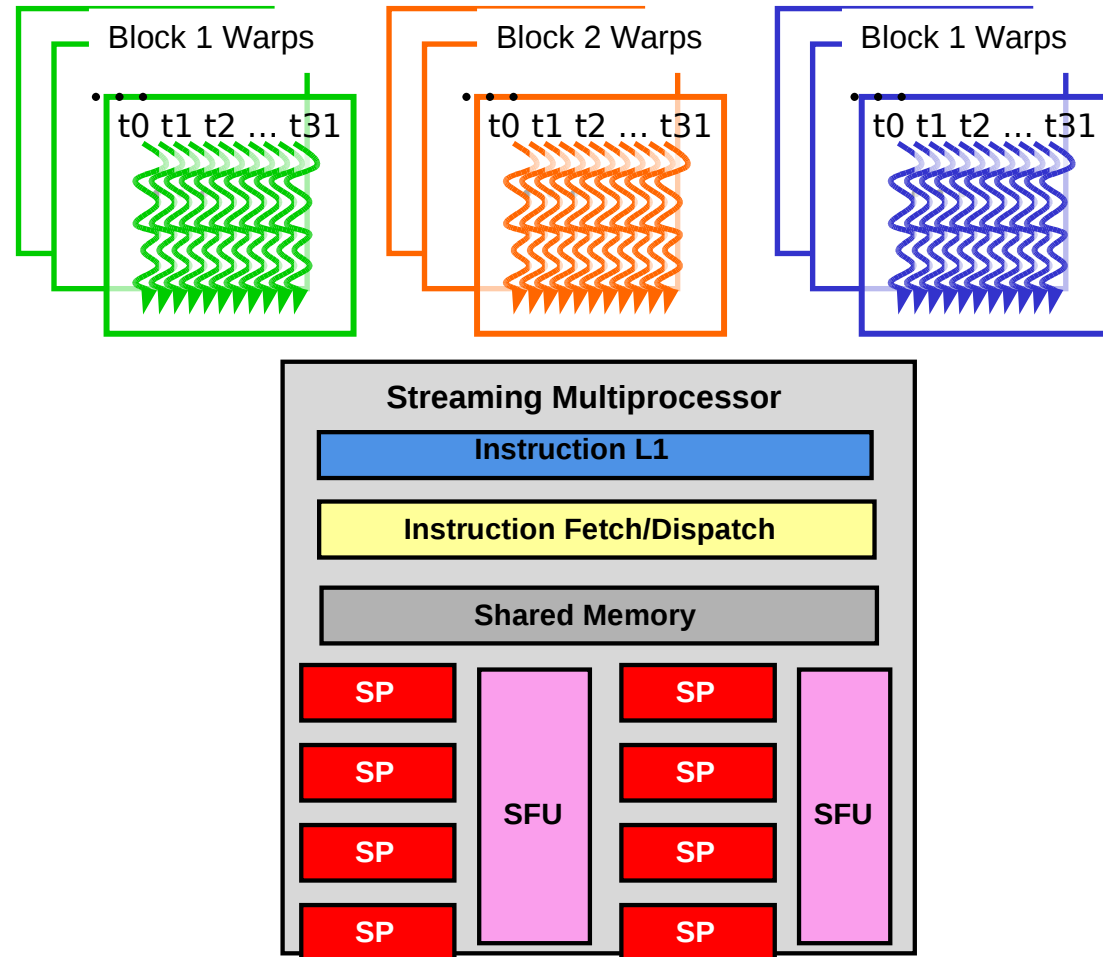
# G80 Example: Executing Thread Blocks



- Threads are assigned to **Streaming Multiprocessors** in block granularity
  - Up to **8** blocks to each SM as resource allows
  - SM in G80 can take up to **768** threads
    - Could be 256 (threads/block) \* 3 blocks
    - Or 128 (threads/block) \* 6 blocks, etc.
- Threads run concurrently
  - SM maintains thread/block id #s
  - SM manages/schedules thread execution

# G80 Example: Thread Scheduling

- Each Block is executed as 32-thread Warps
  - An implementation decision, not part of the CUDA programming model
  - Warps are scheduling units in SM
- If 3 blocks are assigned to an SM and each block has 256 threads, how many Warps are there in an SM?
  - Each Block is divided into  $256/32 = 8$  Warps
  - There are  $8 * 3 = 24$  Warps



# G80 Example: Thread Scheduling (Cont.)

- SM implements zero-overhead warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a warp execute the same instruction when selected

# G80 Block Granularity Considerations

- 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!

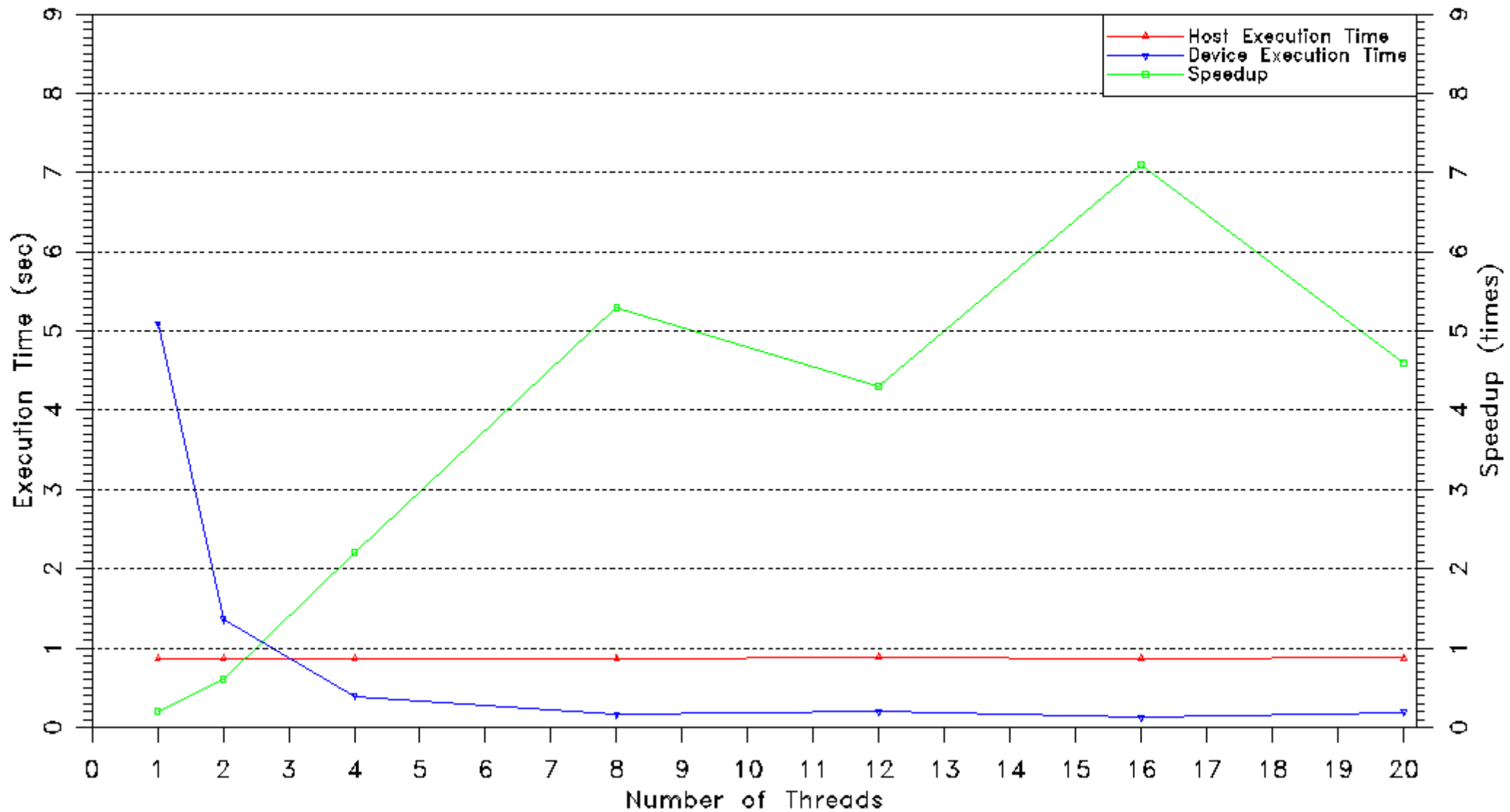


# Example 47: Matrix Multiply

- Download CUDA from  
[http://developer.nvidia.com/object/cuda\\_3\\_0\\_downloads.html](http://developer.nvidia.com/object/cuda_3_0_downloads.html)
- Default install is in `/usr/local/cuda`
- Usage: `ex47 <Bw> <Bn>`
  - $B_w * B_w$  threads per block
  - $B_n * B_n$  blocks
  - Matrix size  $B_w * B_n \times B_w * B_n$
- Order  $n^3$  problem

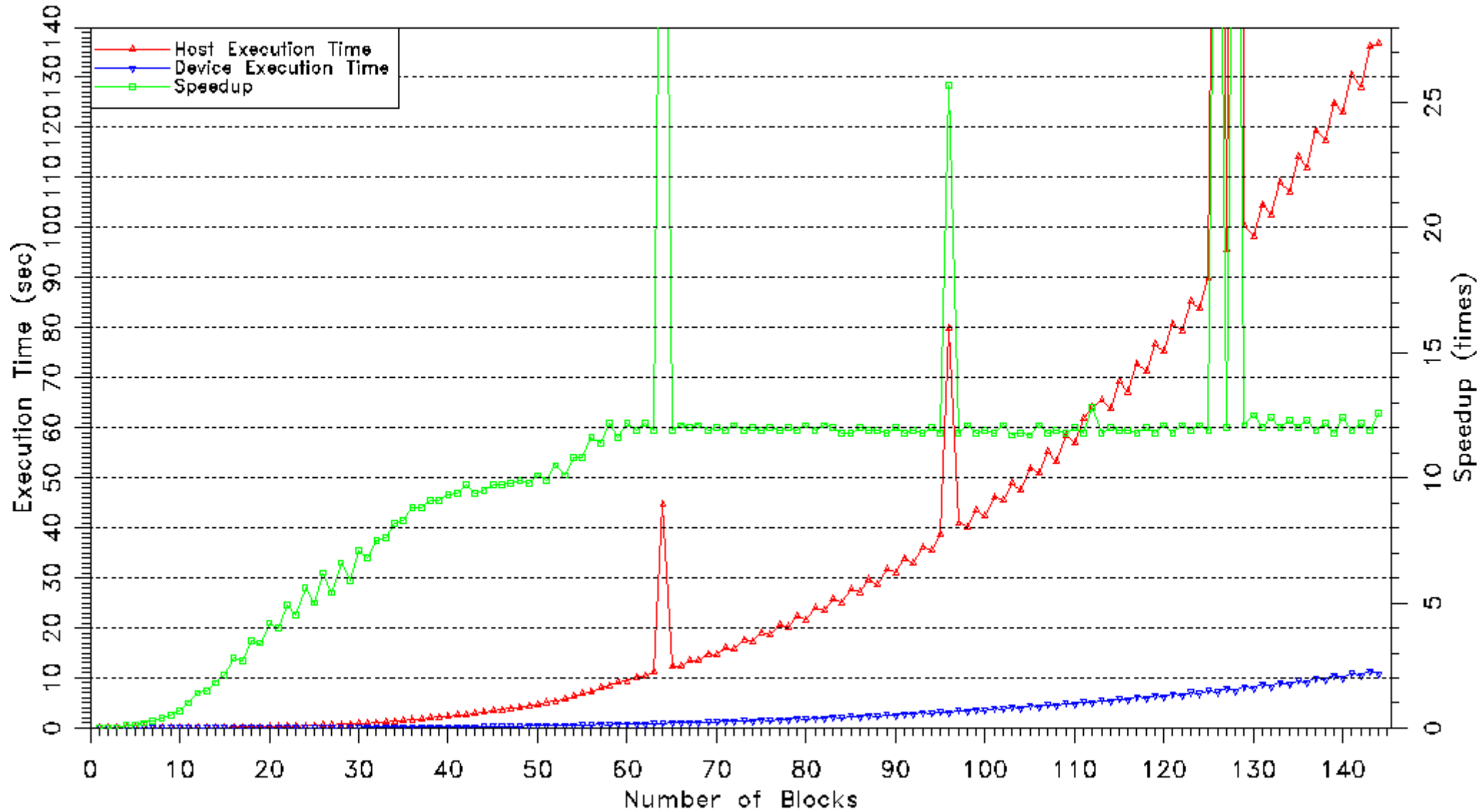
# Matrix Multiply Threads $\times$ Blocks = 480

CUDA Performance: GeForce 9400M vs. Core 2 Duo 2.53GHz



# Matrix Multiply 16x16 Threads per Block

CUDA Performance: GeForce 9400M vs. Core 2 Duo 2.53GHz



# Application Programming Interface

- The API is an **extension to the C programming language**
- It consists of:
  - **Language extensions**
    - To target portions of the code for execution on the device
  - **A runtime library split into:**
    - A **common component** providing built-in vector types and a subset of the C runtime library in both host and device codes
    - A **host component** to control and access one or more devices from the host
    - A **device component** providing device-specific functions

# Language Extensions: Built-in Variables

- **dim3 gridDim;**
  - Dimensions of the grid in blocks (**gridDim.z** unused)
- **dim3 blockDim;**
  - Dimensions of the block in threads
- **dim3 blockIdx;**
  - Block index within the grid
- **dim3 threadIdx;**
  - Thread index within the block

# Common Runtime Component: Mathematical Functions

- `pow`, `sqrt`, `cbrt`, `hypot`
- `exp`, `exp2`, `expm1`
- `log`, `log2`, `log10`, `log1p`
- `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`
- `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`
- `ceil`, `floor`, `trunc`, `round`
- Etc.
  - When executed on the host, a given function uses the C runtime implementation if available
  - These functions are only supported for scalar types, not vector types

# Device Runtime Component: Mathematical Functions

- Some mathematical functions (e.g. **sin(x)**) have a less accurate, but faster device-only version (e.g. **\_\_sin(x)**)
  - **\_\_pow**
  - **\_\_log, \_\_log2, \_\_log10**
  - **\_\_exp**
  - **\_\_sin, \_\_cos, \_\_tan**

# Host Runtime Component

- Provides functions to deal with:
  - **Device** management (including multi-device systems)
  - **Memory** management
  - **Error** handling
- Initializes the first time a runtime function is called
- A host thread can invoke device code on only one device
  - Multiple host threads required to run on multiple devices

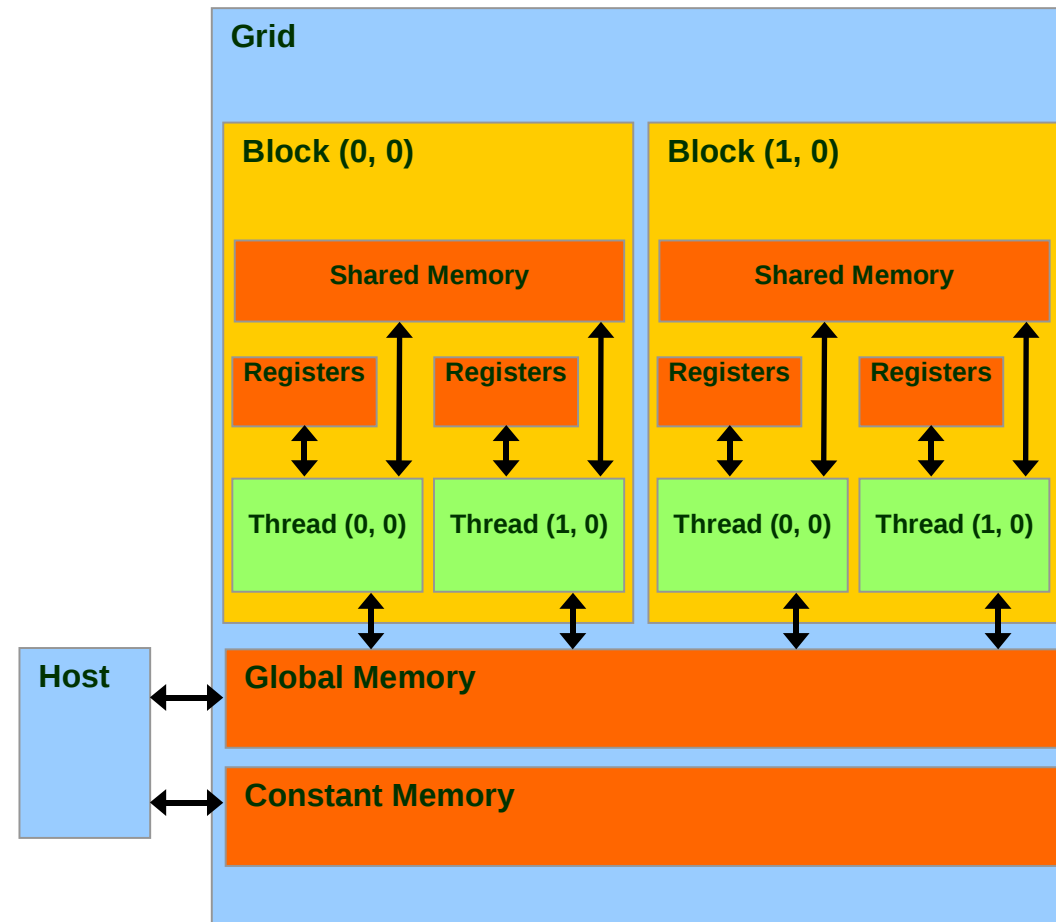


# Device Runtime Component: Synchronization Function

- `void __syncthreads();`
- Synchronizes all threads in a block
- Once all threads have reached this point, execution resumes normally
- Used to avoid RAW / WAR / WAW hazards when accessing shared or global memory
- Allowed in conditional constructs only if the conditional is uniform across the entire thread block

# 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**

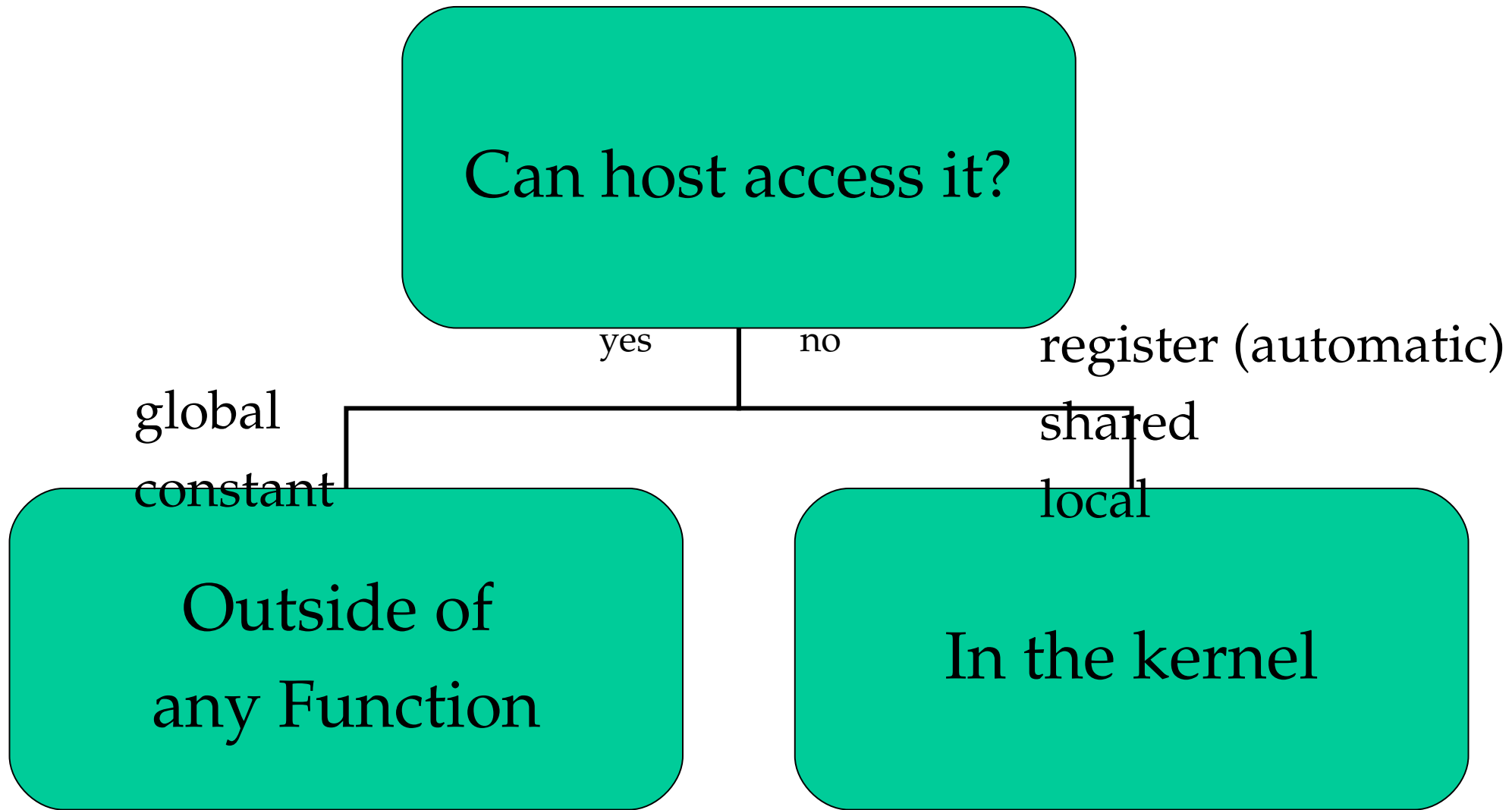


# CUDA Variable Type Qualifiers

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
  - add, sub, min, max, ...
  - and, or, xor
  - Increment, decrement
  - Exchange, compare and swap
- Requires hardware with compute capability 1.1 and above.



# Review: Matrix Multiplication Kernel using Multiple Blocks

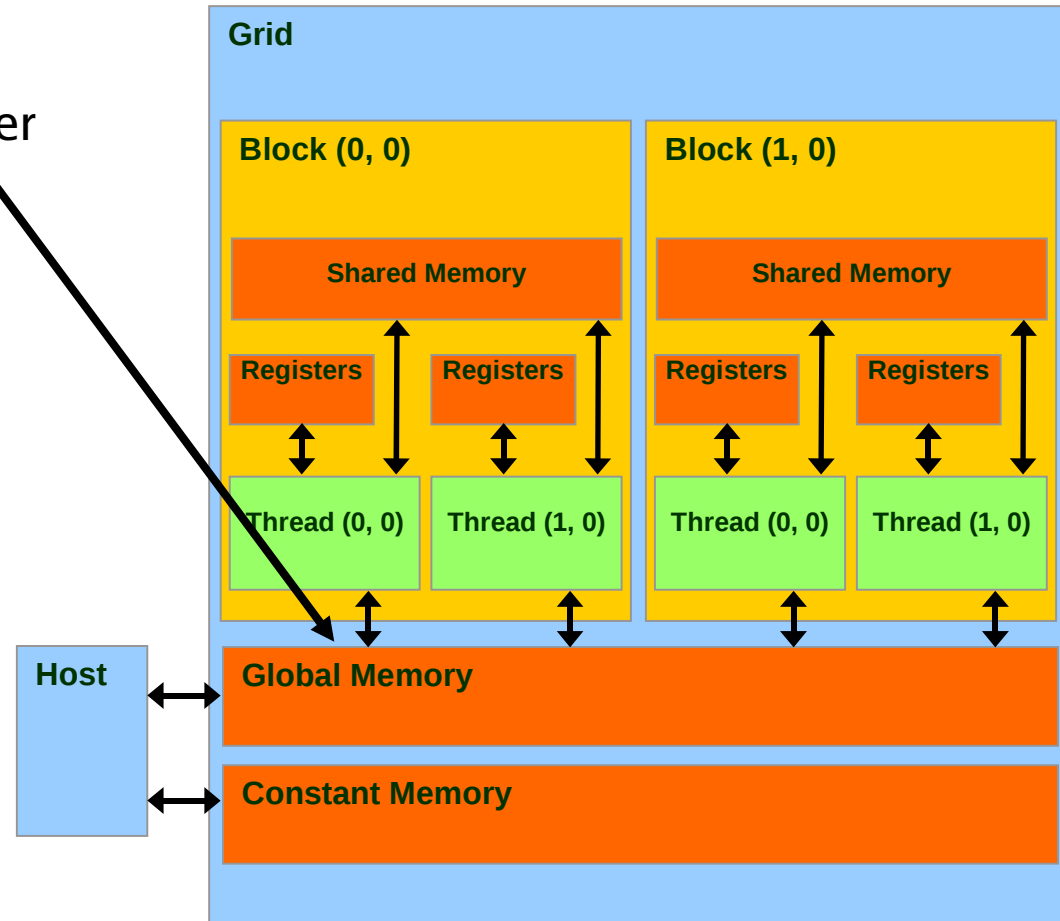
```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // Calculate the row index of the Pd element and M
    int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
    // Calculate the column idenx of Pd and N
    int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

    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;
}
```

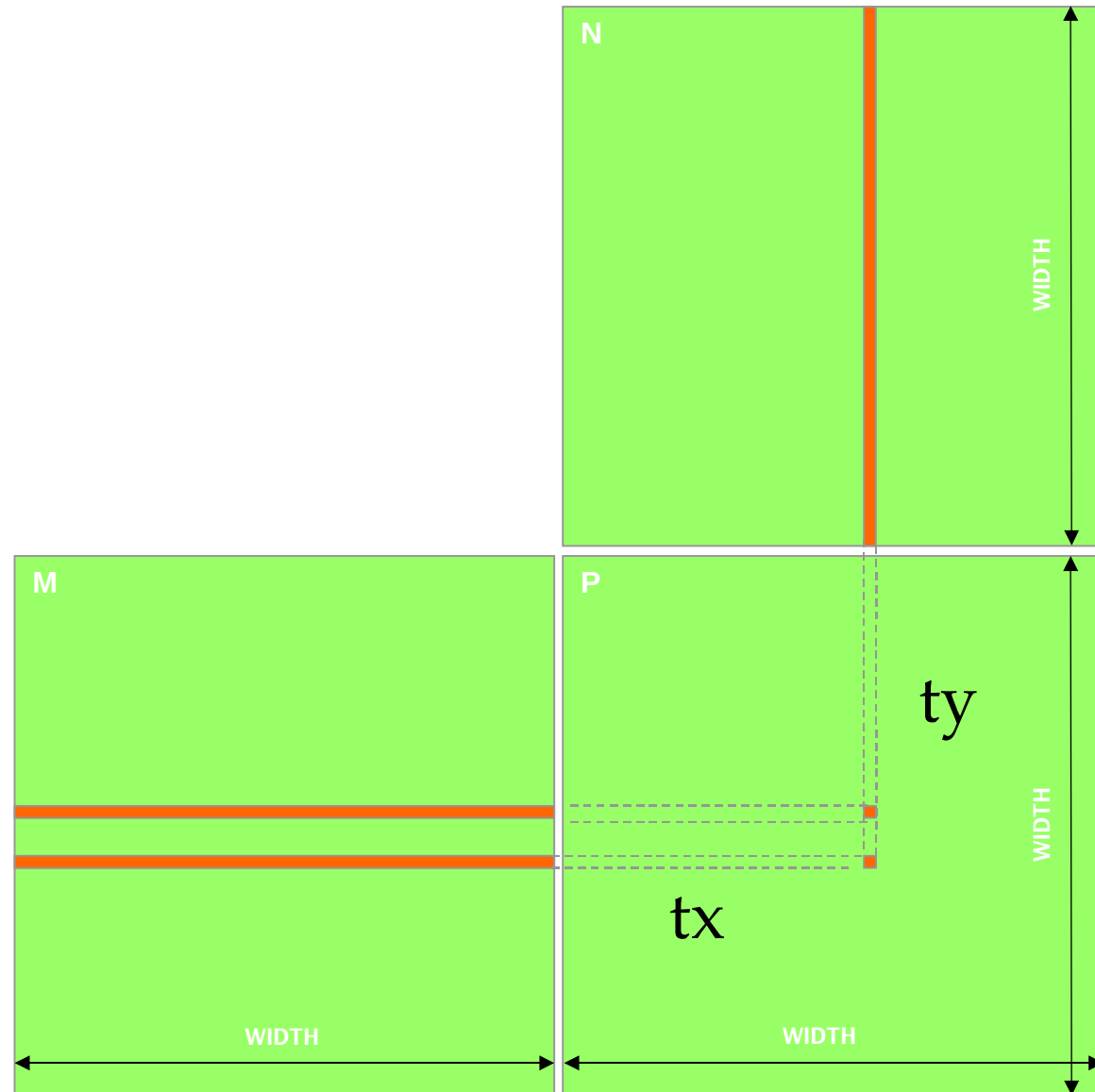
# How about performance on G80?

- 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
- The actual code runs at about 15 GFLOPS
- Need to drastically cut down memory accesses to get closer to the peak 346.5 GFLOPS



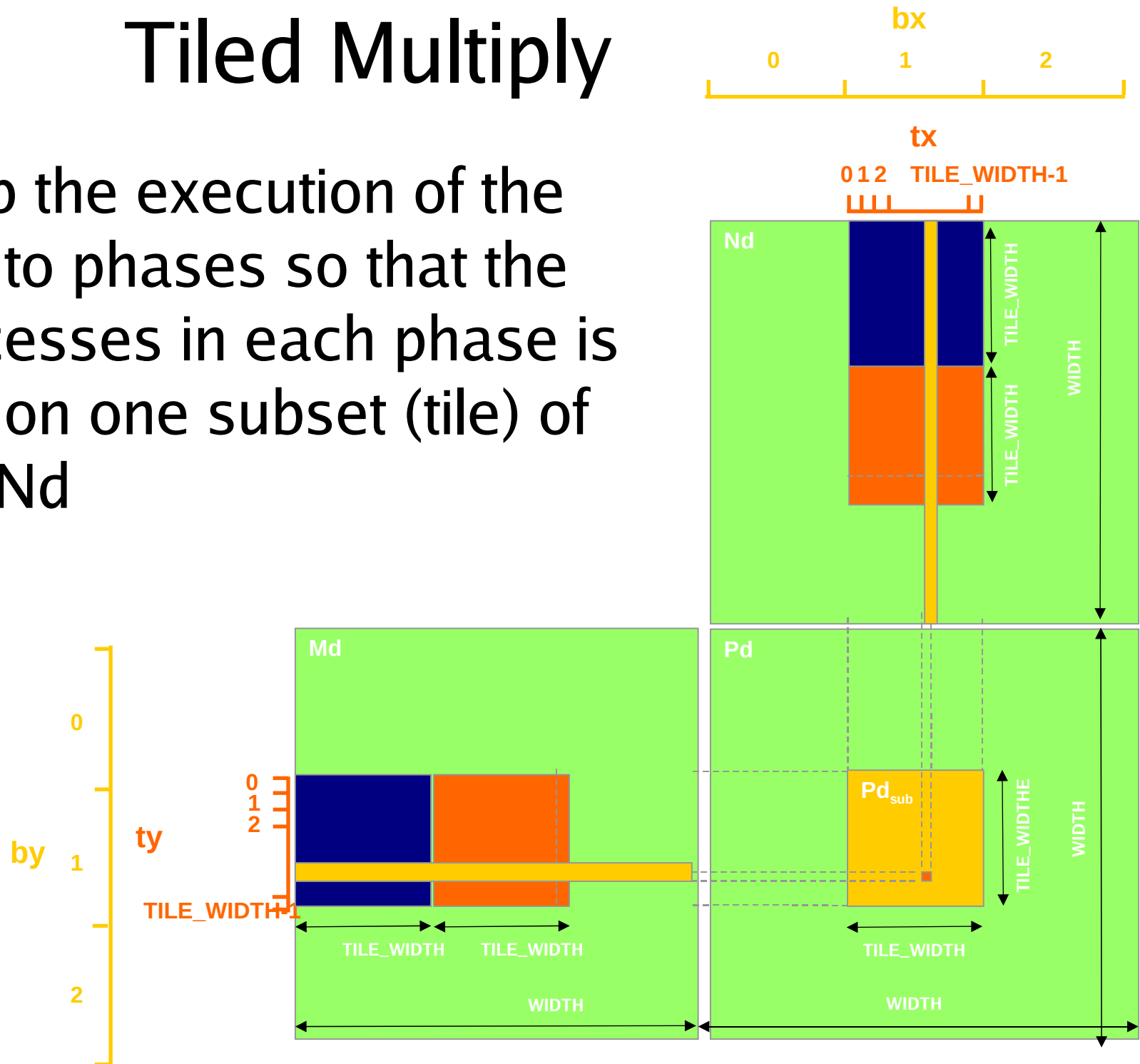
# Idea: Use Shared Memory to reuse global memory data

- 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

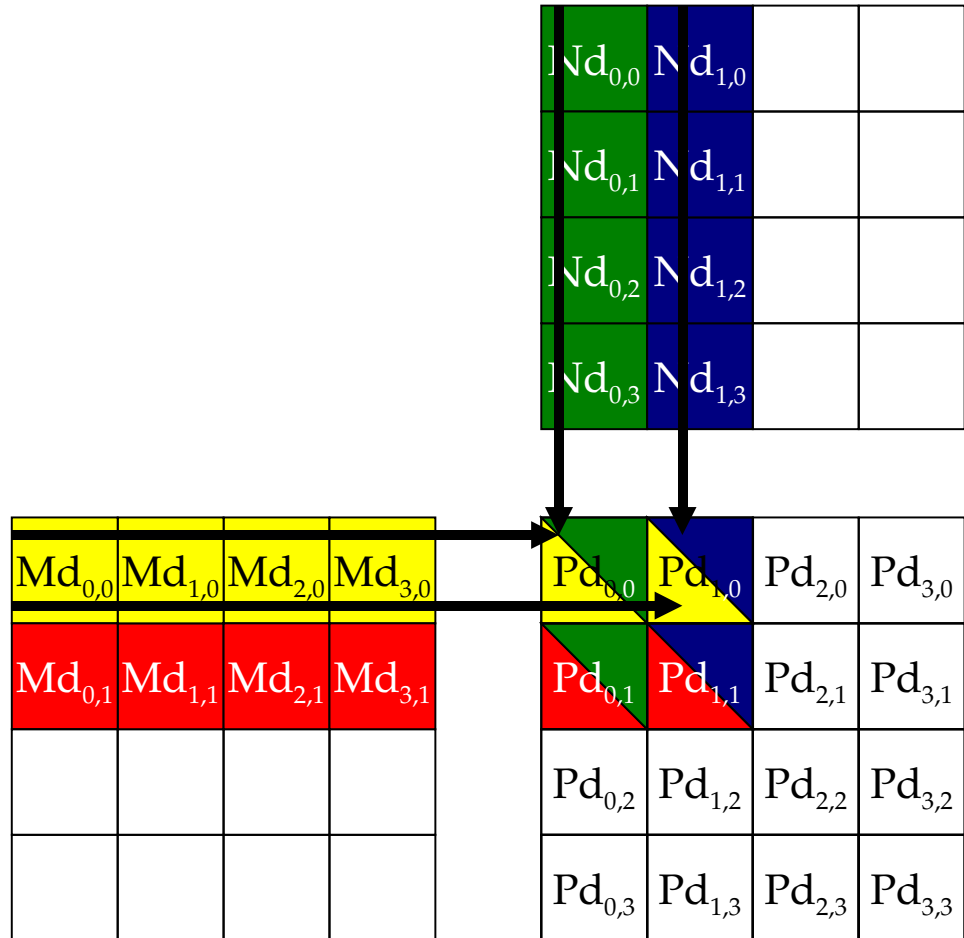


# Tiled Multiply

- 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



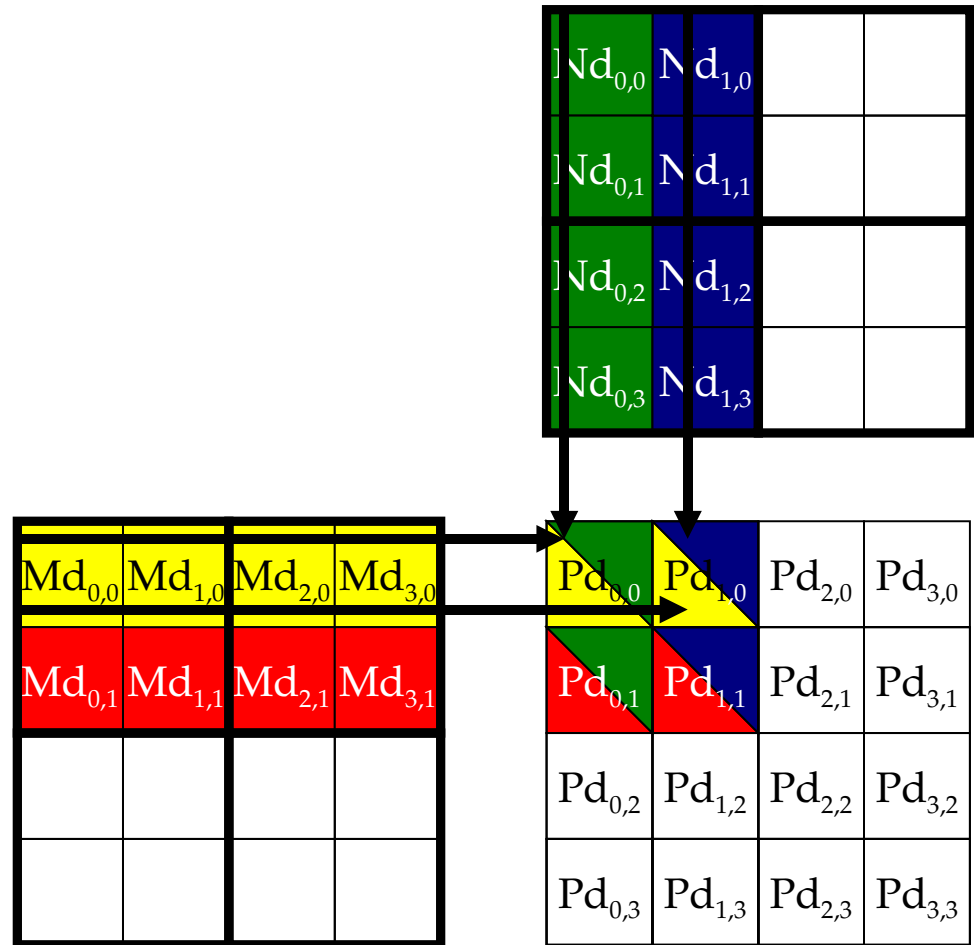
# Every $M_d$ and $N_d$ Element is used exactly twice in generating a $2 \times 2$ tile of $P$

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>
$M_{0,0} * N_{0,0}$	$M_{0,0} * N_{1,0}$	$M_{0,1} * N_{0,0}$	$M_{0,1} * N_{1,0}$
$M_{1,0} * N_{0,1}$	$M_{1,0} * N_{1,1}$	$M_{1,1} * N_{0,1}$	$M_{1,1} * N_{1,1}$
$M_{2,0} * N_{0,2}$	$M_{2,0} * N_{1,2}$	$M_{2,1} * N_{0,2}$	$M_{2,1} * N_{1,2}$
$M_{3,0} * N_{0,3}$	$M_{3,0} * N_{1,3}$	$M_{3,1} * N_{0,3}$	$M_{3,1} * N_{1,3}$

# 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		
$T_{0,0}$	<b>Md</b> <sub>0,0</sub> ↓ Mds <sub>0,0</sub>	<b>Nd</b> <sub>0,0</sub> ↓ Nds <sub>0,0</sub>	PValue <sub>0,0</sub> += Mds <sub>0,0</sub> *Nds <sub>0,0</sub> + Mds <sub>1,0</sub> *Nds <sub>0,1</sub>	<b>Md</b> <sub>2,0</sub> ↓ Mds <sub>0,0</sub>	<b>Nd</b> <sub>0,2</sub> ↓ Nds <sub>0,0</sub>	PValue <sub>0,0</sub> += Mds <sub>0,0</sub> *Nds <sub>0,0</sub> + Mds <sub>1,0</sub> *Nds <sub>0,1</sub>
$T_{1,0}$	<b>Md</b> <sub>1,0</sub> ↓ Mds <sub>1,0</sub>	<b>Nd</b> <sub>1,0</sub> ↓ Nds <sub>1,0</sub>	PValue <sub>1,0</sub> += Mds <sub>0,0</sub> *Nds <sub>1,0</sub> + Mds <sub>1,0</sub> *Nds <sub>1,1</sub>	<b>Md</b> <sub>3,0</sub> ↓ Mds <sub>1,0</sub>	<b>Nd</b> <sub>1,2</sub> ↓ Nds <sub>1,0</sub>	PValue <sub>1,0</sub> += Mds <sub>0,0</sub> *Nds <sub>1,0</sub> + Mds <sub>1,0</sub> *Nds <sub>1,1</sub>
$T_{0,1}$	<b>Md</b> <sub>0,1</sub> ↓ Mds <sub>0,1</sub>	<b>Nd</b> <sub>0,1</sub> ↓ Nds <sub>0,1</sub>	PdValue <sub>0,1</sub> += Mds <sub>0,1</sub> *Nds <sub>0,0</sub> + Mds <sub>1,1</sub> *Nds <sub>0,1</sub>	<b>Md</b> <sub>2,1</sub> ↓ Mds <sub>0,1</sub>	<b>Nd</b> <sub>0,3</sub> ↓ Nds <sub>0,1</sub>	PdValue <sub>0,1</sub> += Mds <sub>0,1</sub> *Nds <sub>0,0</sub> + Mds <sub>1,1</sub> *Nds <sub>0,1</sub>
$T_{1,1}$	<b>Md</b> <sub>1,1</sub> ↓ Mds <sub>1,1</sub>	<b>Nd</b> <sub>1,1</sub> ↓ Nds <sub>1,1</sub>	PdValue <sub>1,1</sub> += Mds <sub>0,1</sub> *Nds <sub>1,0</sub> + Mds <sub>1,1</sub> *Nds <sub>1,1</sub>	<b>Md</b> <sub>3,1</sub> ↓ Mds <sub>1,1</sub>	<b>Nd</b> <sub>1,3</sub> ↓ Nds <sub>1,1</sub>	PdValue <sub>1,1</sub> += Mds <sub>0,1</sub> *Nds <sub>1,0</sub> + Mds <sub>1,1</sub> *Nds <sub>1,1</sub>

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)];
•      Nds[ty][tx] = Nd[(m*TILE_WIDTH + ty)*Width + Col];
•      __syncthreads();

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

# CUDA Code –Kernel Execution Configuration

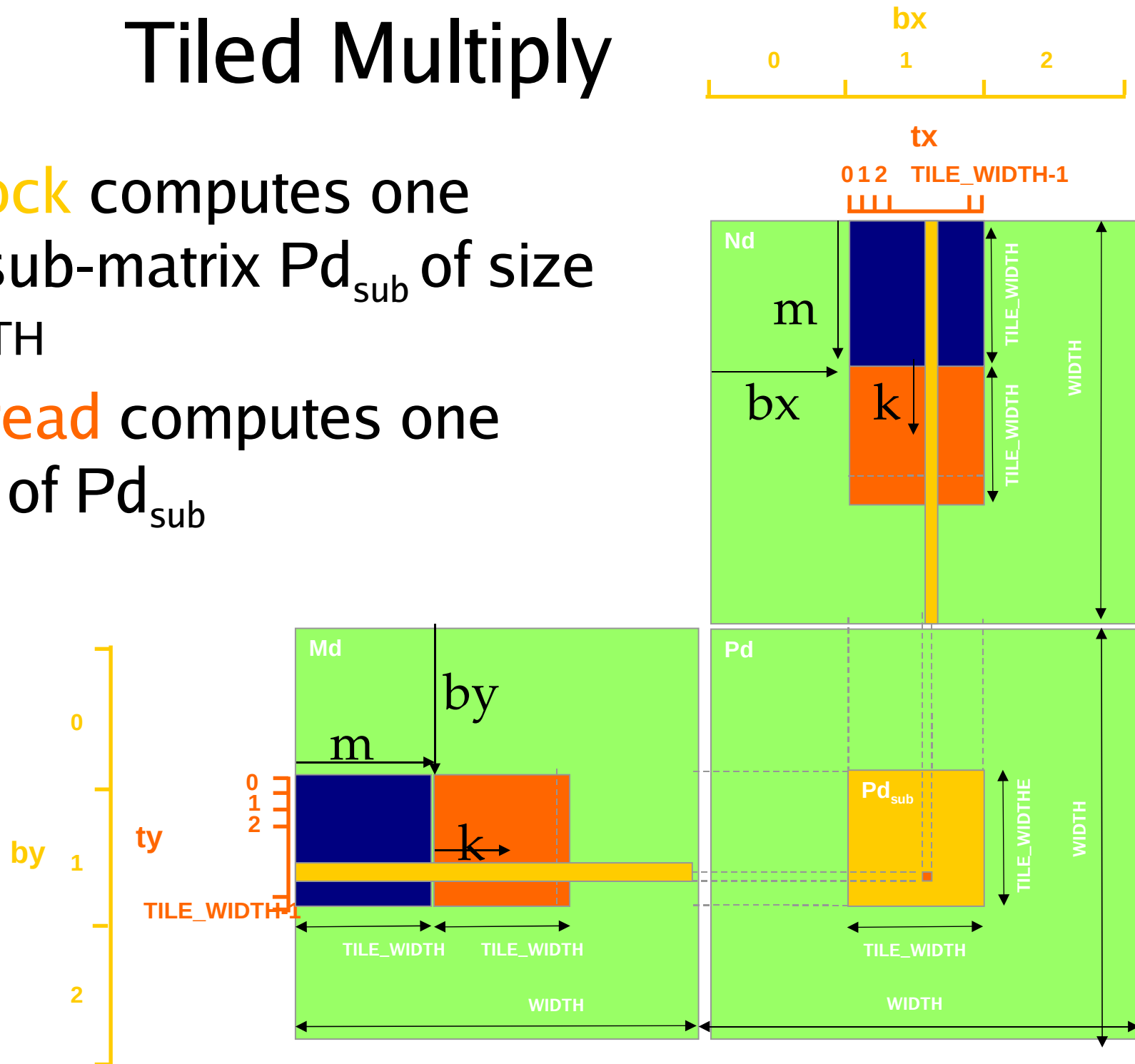
```
// Setup the execution configuration
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);
dim3 dimGrid(Width / TILE_WIDTH,
             Width / TILE_WIDTH);
```

# 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  $2*256*4B = 2KB$  of shared memory.
  - The 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 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  $2*32*32*4B = 8KB$  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!

# 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;

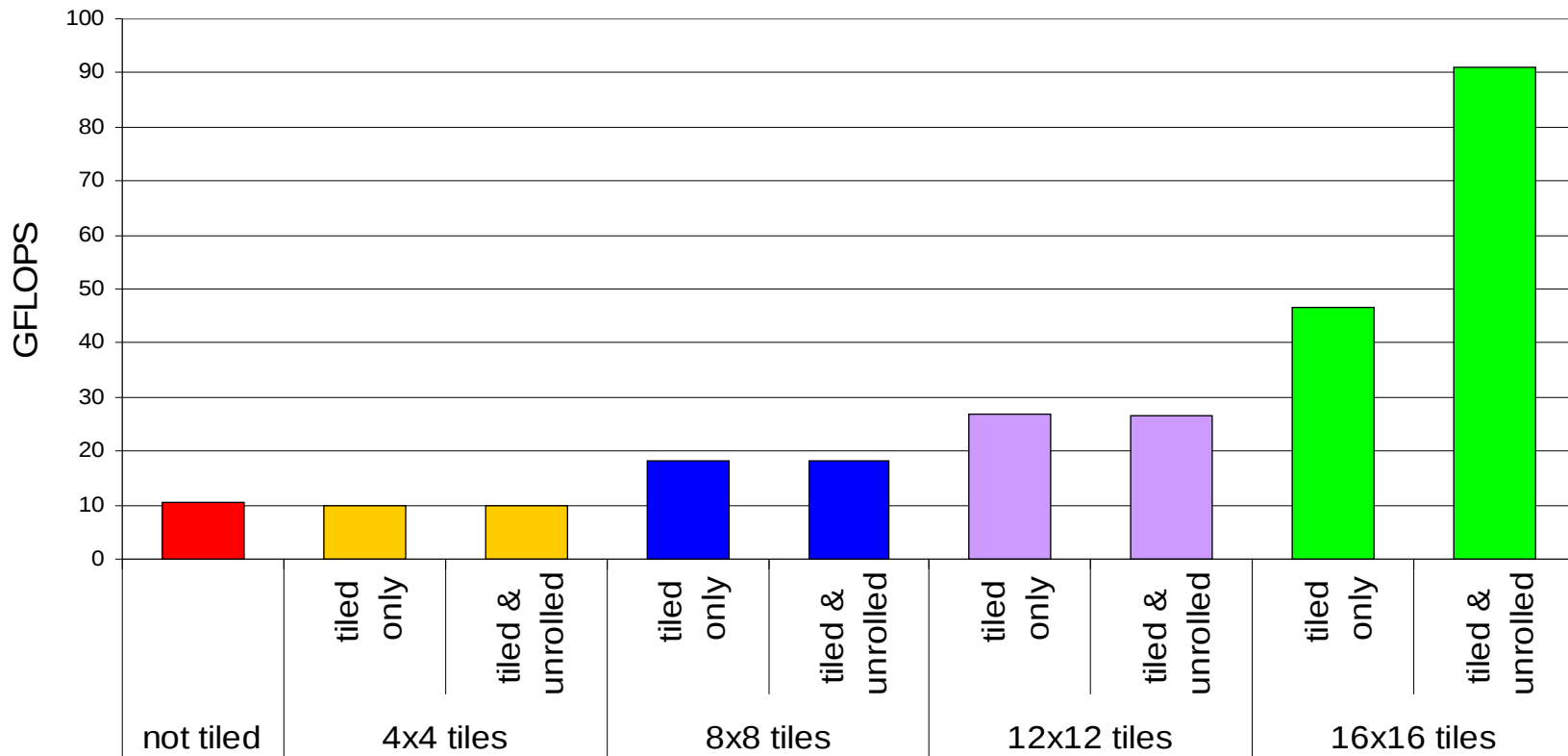
// Identify the row and column of the Pd element to work on
5.  int Row = by * TILE_WIDTH + ty;
6.  int Col = bx * TILE_WIDTH + tx;

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)];
•      Nds[ty][tx] = Nd[(m*TILE_WIDTH + ty)*Width + Col];
•      __syncthreads();


12.     for (int k = 0; k < TILE_WIDTH; ++k)
•         Pvalue += Mds[ty][k] * Nds[k][tx];
14.     __syncthreads();
}
15. 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_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