CSCI-GA.3033-004

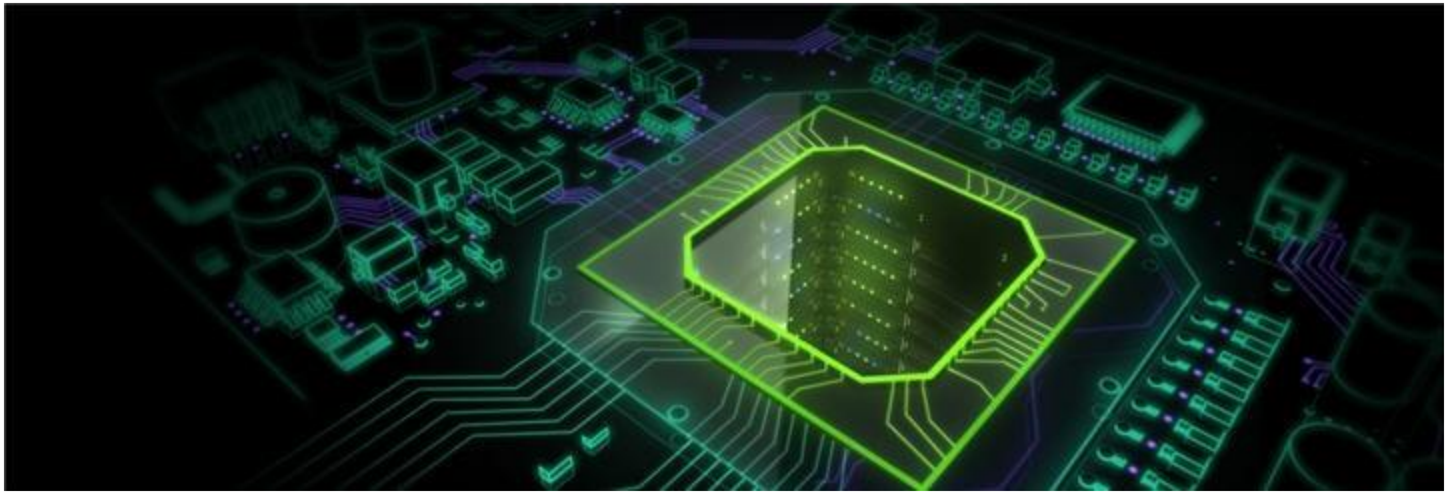
Graphics Processing Units (GPUs): Architecture and Programming

Lecture 4: CUDA Threads & Memories

Mohamed Zahran (aka Z)

mzahran@cs.nyu.edu

http://www.mzahran.com



Software <-> Hardware

- From a programmer's perspective:
  - Blocks
  - Kernel
  - Threads
  - Grid
- Hardware Implementation:
  - SMs
  - SPs (per SM)
  - Warps

Some Restrictions First

- All threads in a grid execute the same kernel function
- A grid is organized as a 2D or 3D array of blocks (gridDim.x, gridDim.y, and gridDim.z)
- Each block is organized as 3D array of threads (blockDim.x, blockDim.y, and blockDim.z)
- Once a kernel is launched, its dimensions cannot change.
- All blocks in a grid have the same dimension
- The total size of a block has an upper bound
- Once assigned to an SM, the block must execute in its entirety by the SM

Compute Capability

- A standard way to expose hardware resources to applications.
- CUDA compute capability starts with 1.0 and latest one is 7.x (as of today)
- API: cudaGetDeviceProperties()

```
cudaError_t cudaGetDeviceProperties(
        struct cudaDeviceProp * prop,
        int device)
```

```
struct cudaDeviceProp {
        char name[256];
        size_t  totalGlobalMem;  /* in bytes */
        size_t sharedMemPerBlock;  /* in bytes */
        int  regsPerBlock;
        int warpSize;
        int maxThreadsPerBlock;
        int maxThreadsDim[3];
        int maxGridSize[3];
        int clockRate;  /* in KHz */
        size_t totalConstMem;
        int major;    int minor;
        int multiProcessorCount;
        int concurrentKernels;
        int unifiedAddressing;
        int memoryClockRate;
        int memoryBusWidth;
        int l2CacheSize;
        int maxThreadsPerMultiProcessor;
        … and a lot of other stuff}
```

```
cudaError_t
    cudaGetDeviceCount(
            int * count )
```

Compute Capability Example

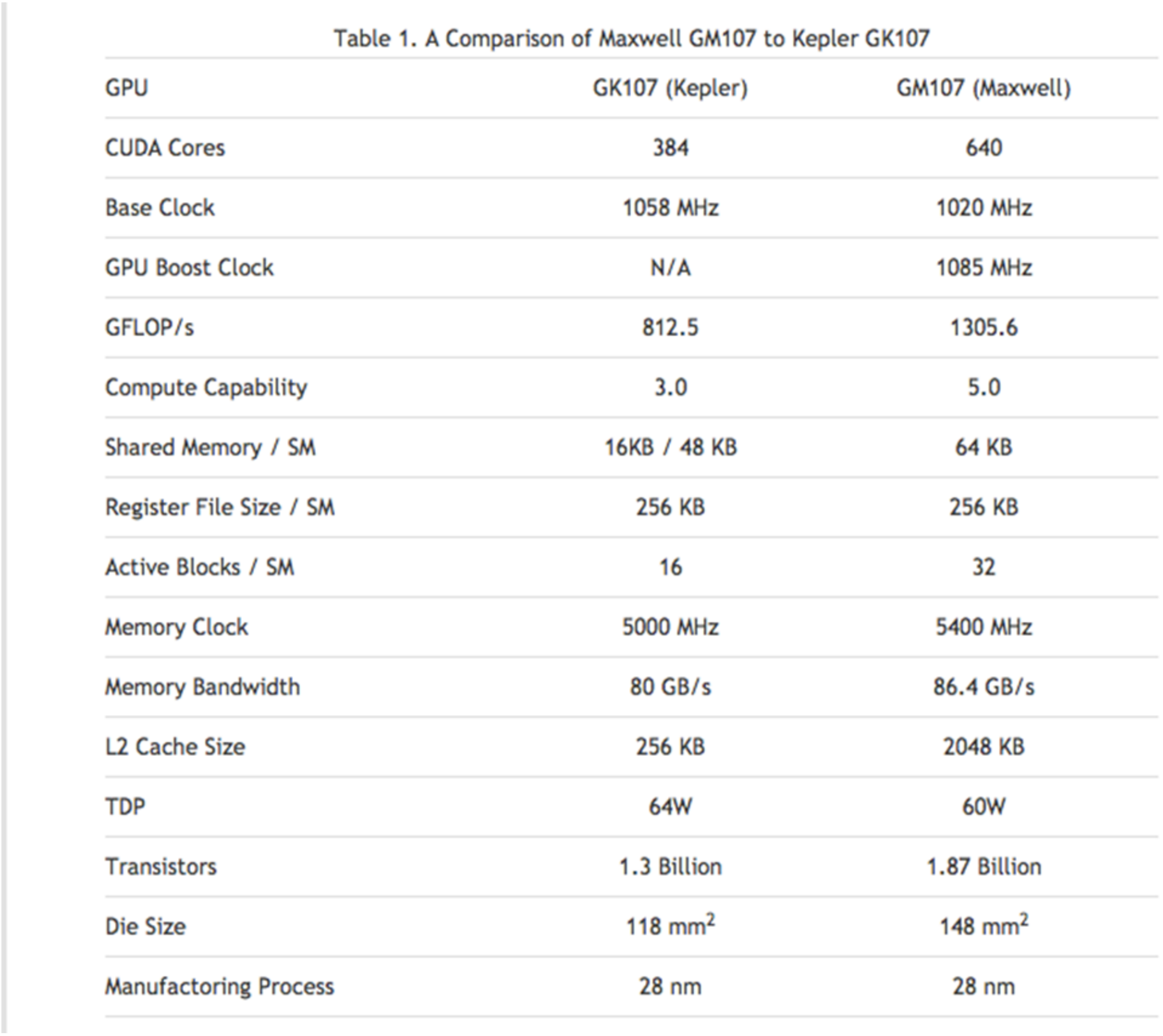
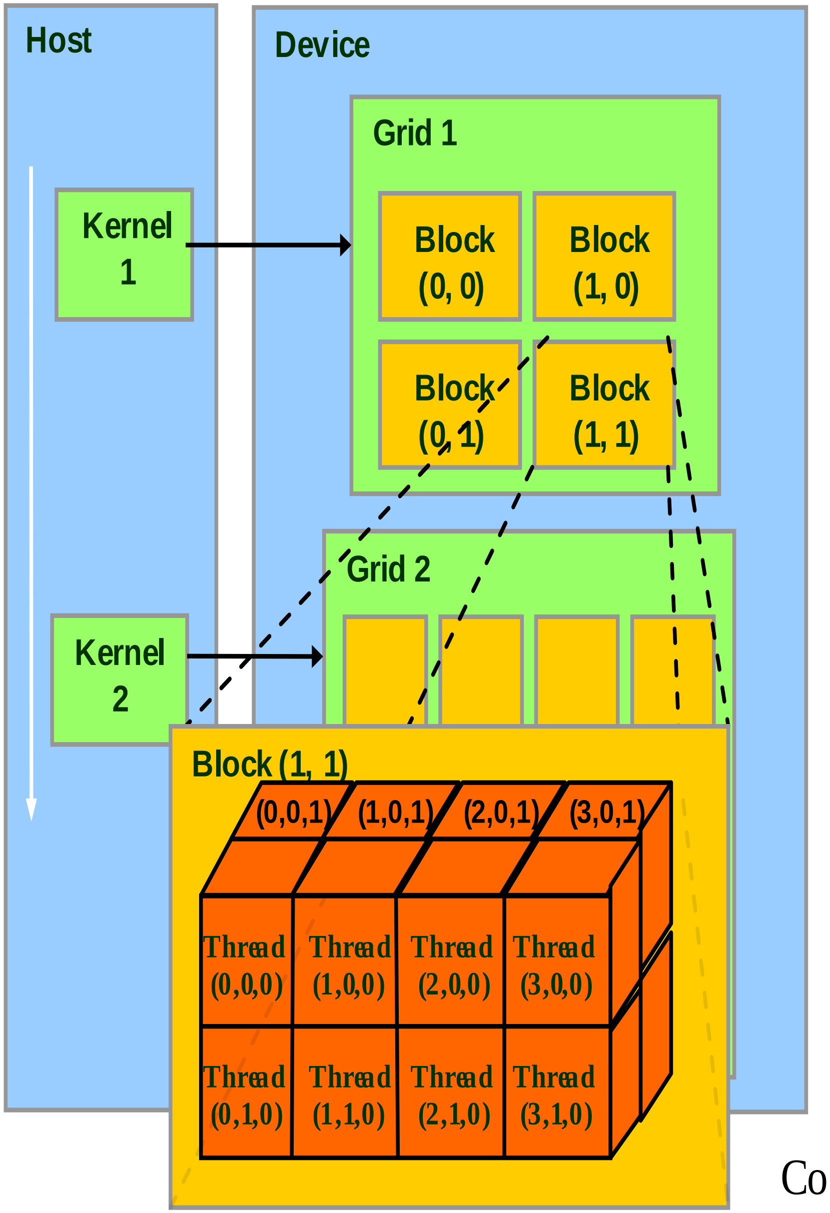


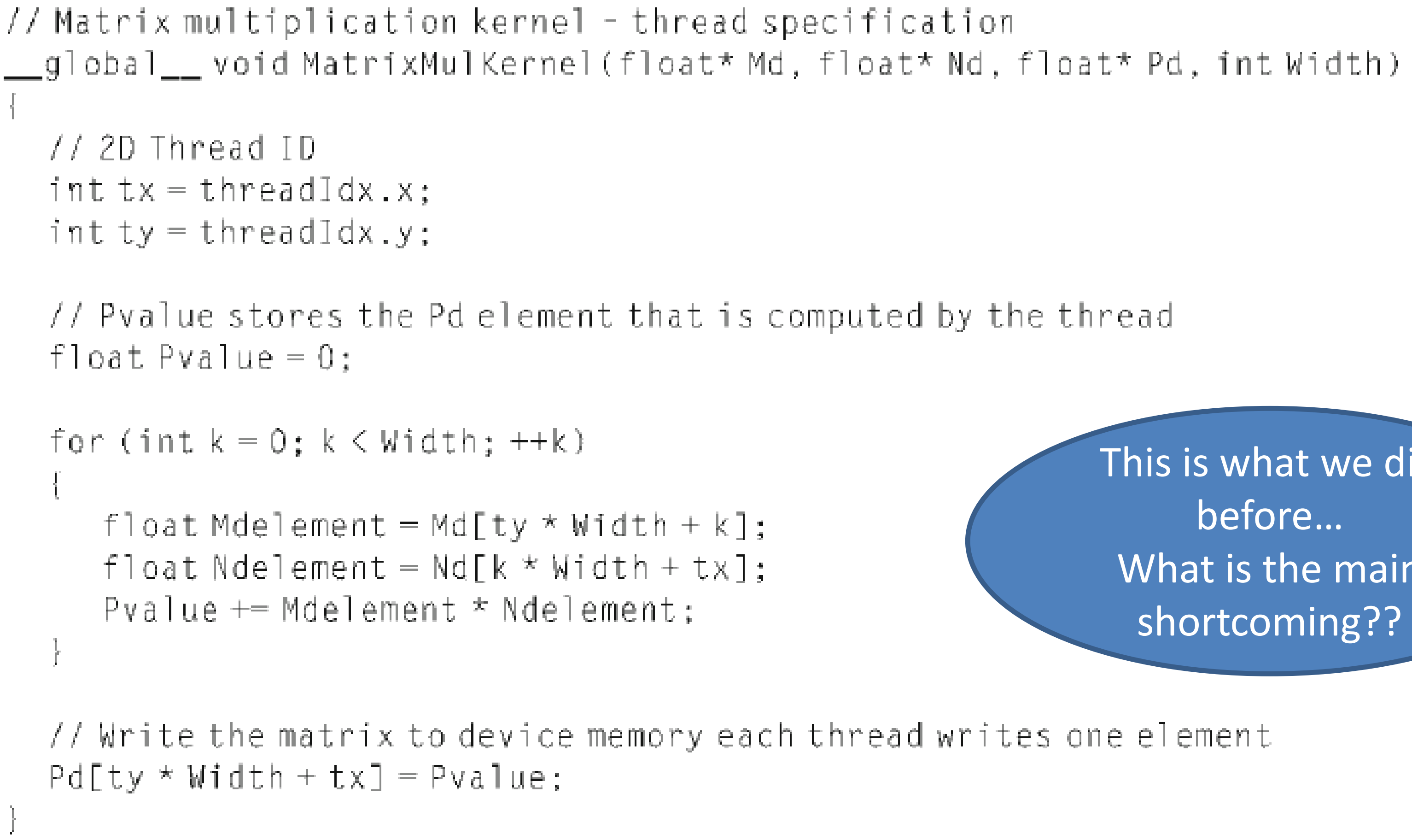
Table 1. A Comparison of Maxwell GM107 to Kepler GK107

GPU	GK107 (Kepler)	GM107 (Maxwell)
CUDA Cores	384	640
Base Clock	1058 MHz	1020 MHz
GPU Boost Clock	N/A	1085 MHz
GFLOP/s	812.5	1305.6
Compute Capability	3.0	5.0
Shared Memory / SM	16KB / 48 KB	64 KB
Register File Size / SM	256 KB	256 KB
Active Blocks / SM	16	32
Memory Clock	5000 MHz	5400 MHz
Memory Bandwidth	80 GB/s	86.4 GB/s
L2 Cache Size	256 KB	2048 KB
TDP	64W	60W
Transistors	1.3 Billion	1.87 Billion
Die Size	118 mm^2	148 mm^2
Manufactoring Process	28 nm	28 nm



Courtesy: NDVIA

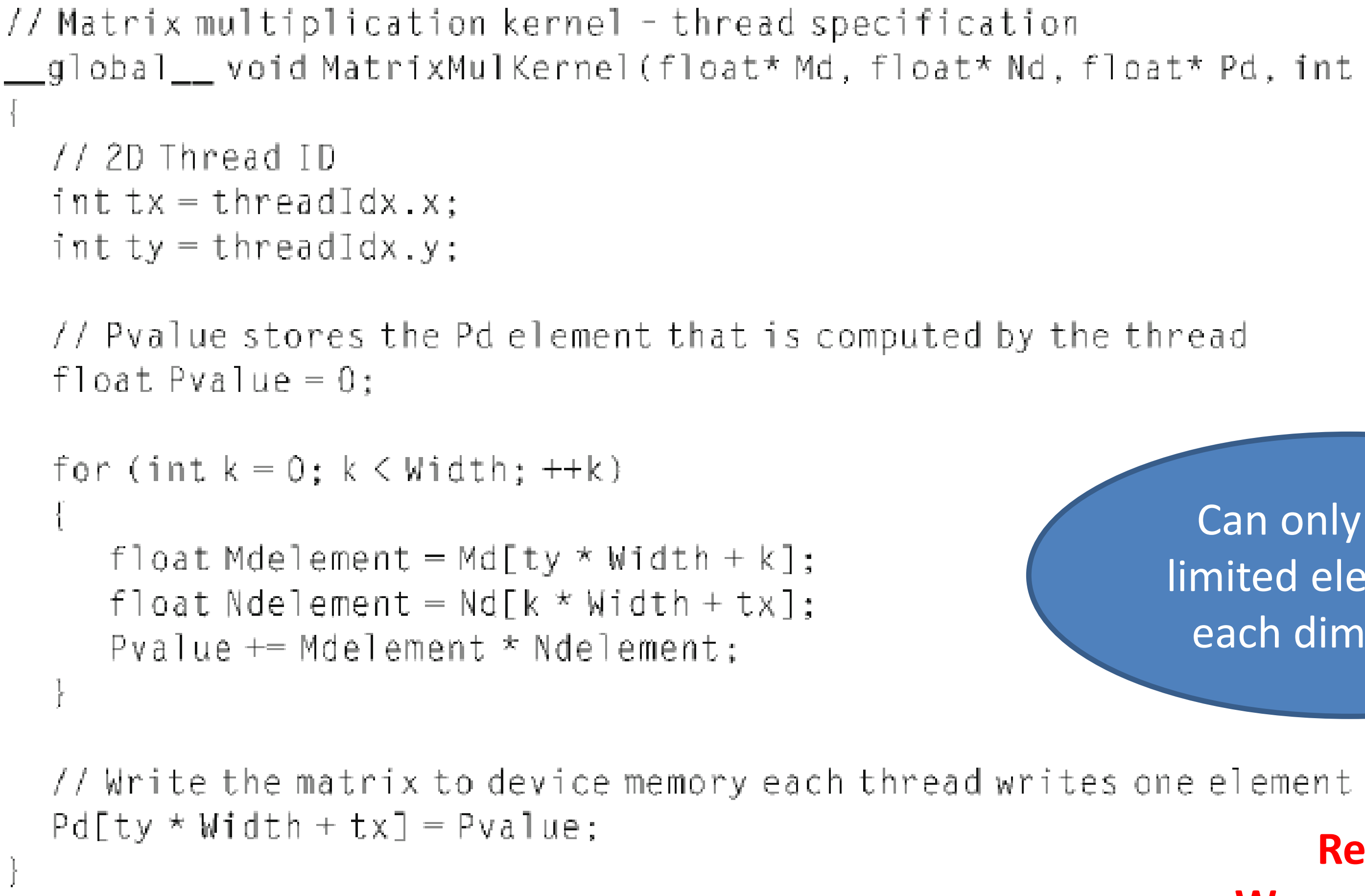
Revisiting Matrix Multiplication



```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
   // 2D Thread ID
   int tx = threadIdx.x;
   int ty = threadIdx.y;

   // Pvalue stores the Pd element that is computed by the thread
   float Pvalue = 0;

   for (int k = 0; k < Width; ++k)
   {
      float Mdelement = Md[ty * Width + k];
      float Ndelement = Nd[k * Width + tx];
      Pvalue += Mdelement * Ndelement;
   }

   // Write the matrix to device memory each thread writes one element
   Pd[ty * Width + tx] = Pvalue;
}
```

This is what we did before…
What is the main shortcoming??

Revisiting Matrix Multiplication



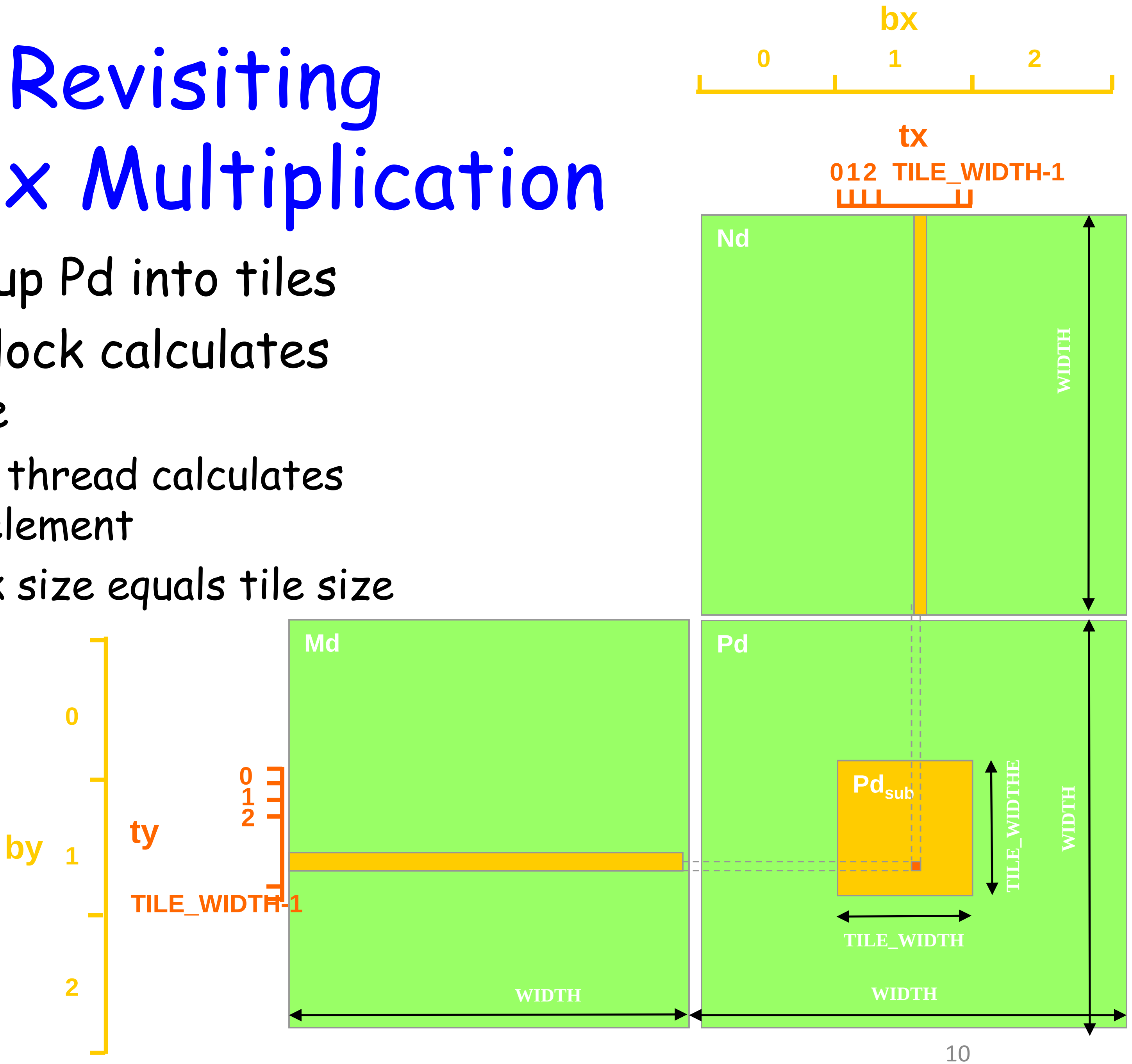
```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
  // 2D Thread ID
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  // Pvalue stores the Pd element that is computed by the thread
  float Pvalue = 0;

  for (int k = 0; k < Width; ++k)
  {
     float Mdelement = Md[ty * Width + k];
     float Ndelement = Nd[k * Width + tx];
     Pvalue += Mdelement * Ndelement;
  }

  // Write the matrix to device memory each thread writes one element
  Pd[ty * Width + tx] = Pvalue;
}
```

Can only handle limited elements in each dimension!

Reason:
We used 1 block,
and a block is limited to X threads
(X depends on GPU type)

Revisiting Matrix Multiplication

- Break-up Pd into tiles
- Each block calculates one tile
  - Each thread calculates one element
  - Block size equals tile size



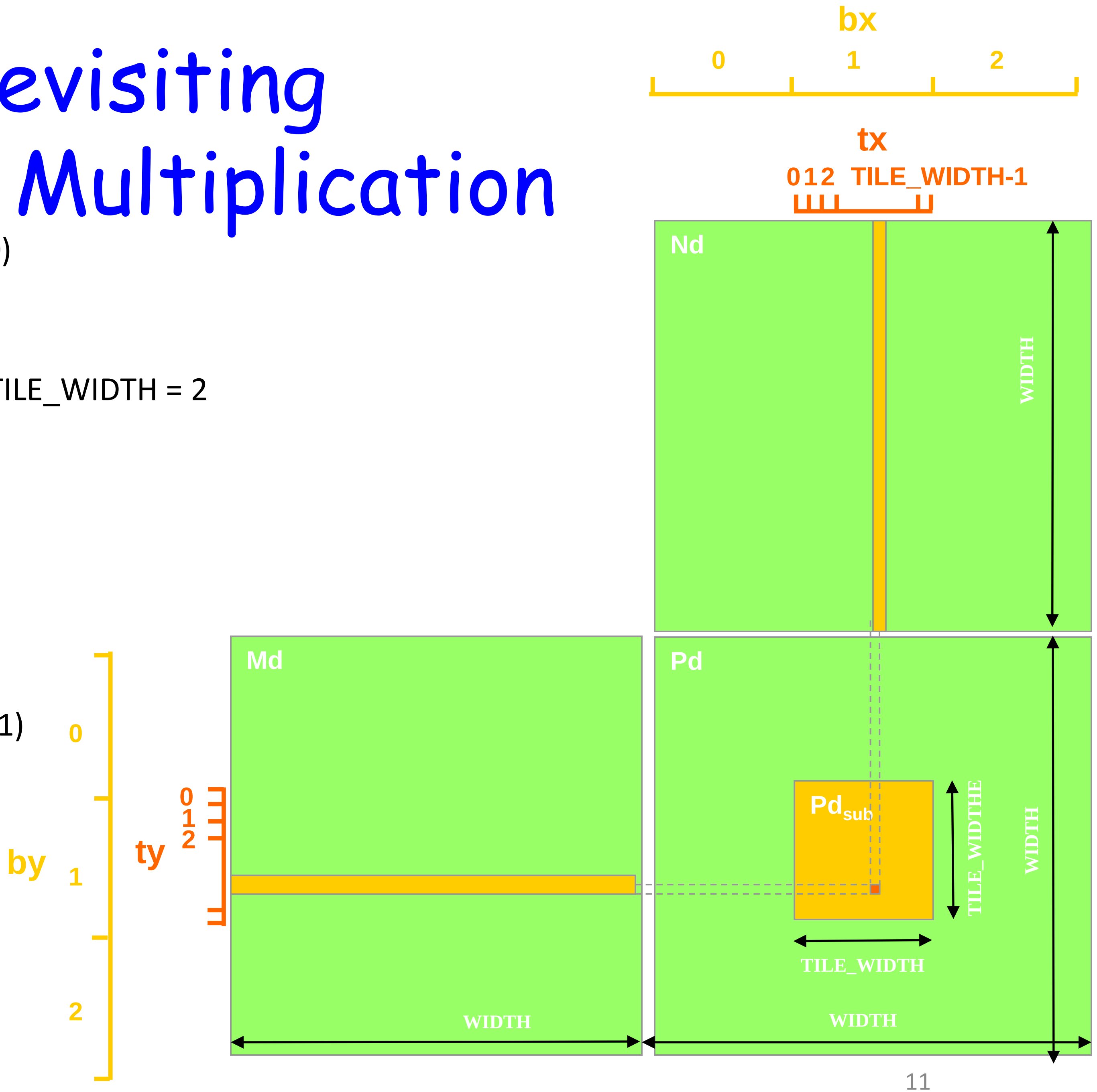
Revisiting Matrix Multiplication

Block(0,0) Block(1,0)

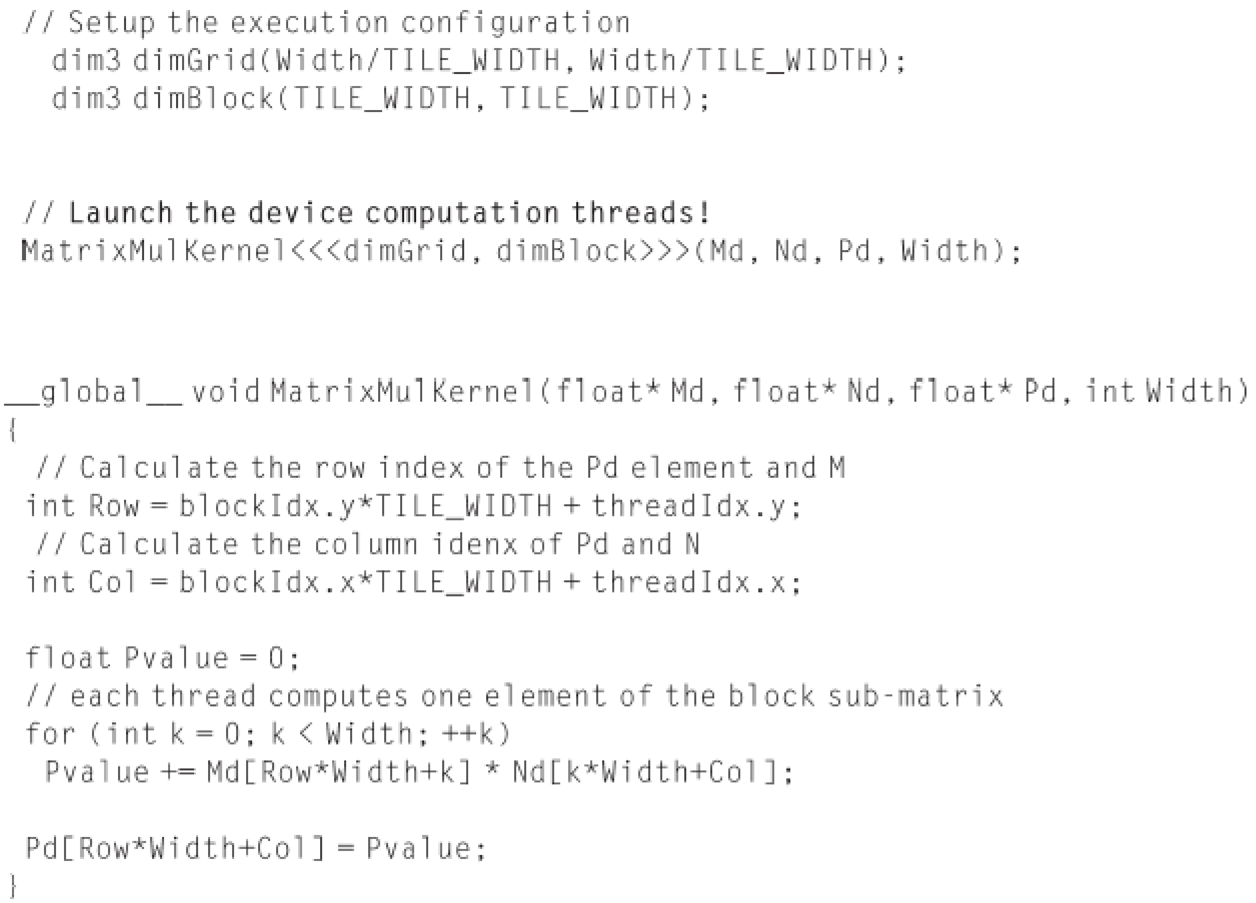
$P_{0,0}$	$P_{1,0}$	$P_{2,0}$	$P_{3,0}$
$P_{0,1}$	$P_{1,1}$	$P_{2,1}$	$P_{3,1}$
$P_{0,2}$	$P_{1,2}$	$P_{2,2}$	$P_{3,2}$
$P_{0,3}$	$P_{1,3}$	$P_{2,3}$	$P_{3,3}$

TILE_WIDTH = 2

Block(0,1) Block(1,1)



Revisiting Matrix Multiplication

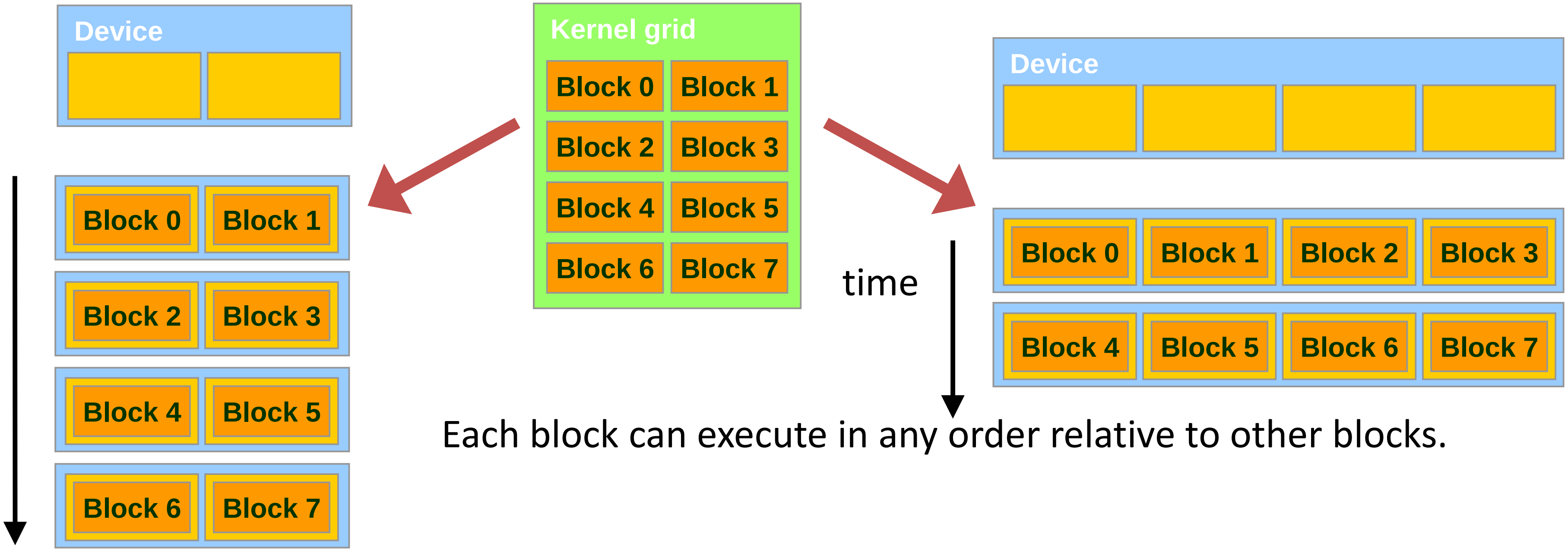


```
// Setup the execution configuration
  dim3 dimGrid(Width/TILE_WIDTH, Width/TILE_WIDTH);
  dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);


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


__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;
}
```

Synchronization

__syncthreads()

- called by a kernel function
- The thread that makes the call will be held at the calling location until every thread in the block reaches the location
- Beware of if-then-else
- Threads in different blocks cannot synchronize -> CUDA runtime system can execute blocks in any order

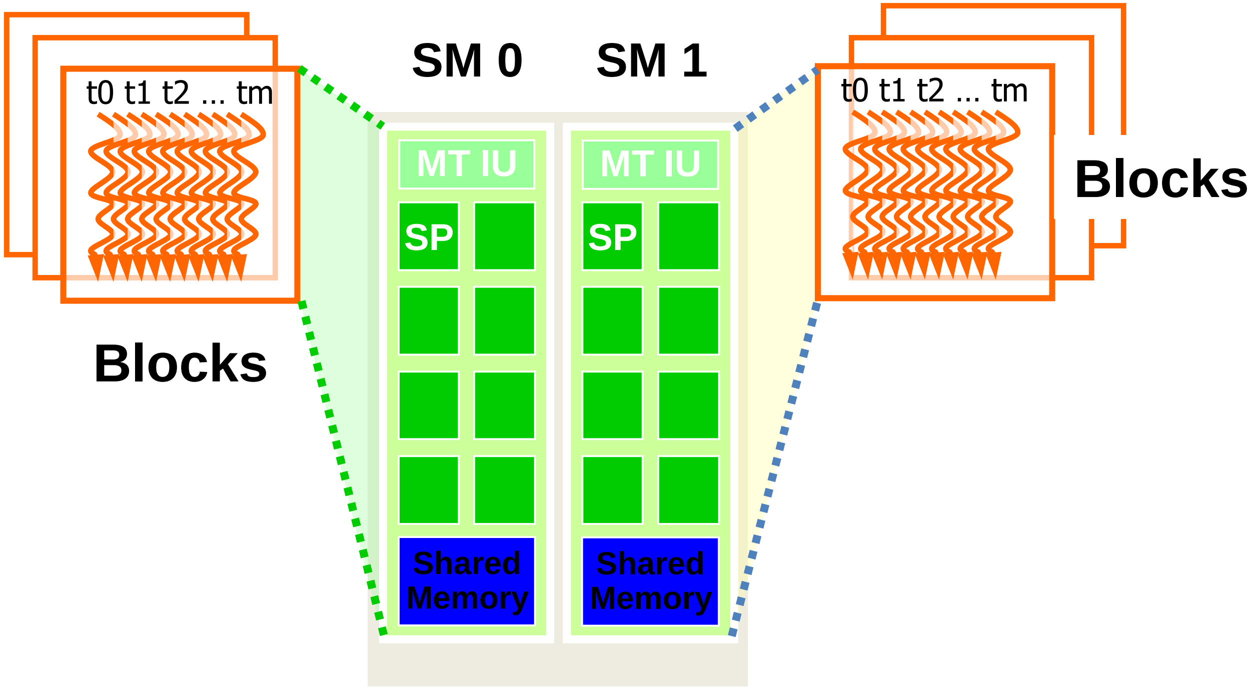


Each block can execute in any order relative to other blocks.

The ability to execute the same application code on hardware with different number of execution resources is called **transparent scalability**

Thread Assignment

- Threads assigned to execution resources on a block-by-block basis.
- CUDA runtime automatically reduces number of blocks assigned to each SM until resource usage is under limit.
- Runtime system:
  - maintains a list of blocks that need to execute
  - assigns new blocks to SM as they compute previously assigned blocks
- Example of SM resources
  - computational units
  - number of threads that can be simultaneously tracked and scheduled.
  - registers
  - shared memory



GT200 can accommodate 8 blocks/SM and up to 1024 threads can be assigned to an SM.

What are our choices for number of blocks and number of threads/block?

Thread scheduling is an implementation concept.

Warps

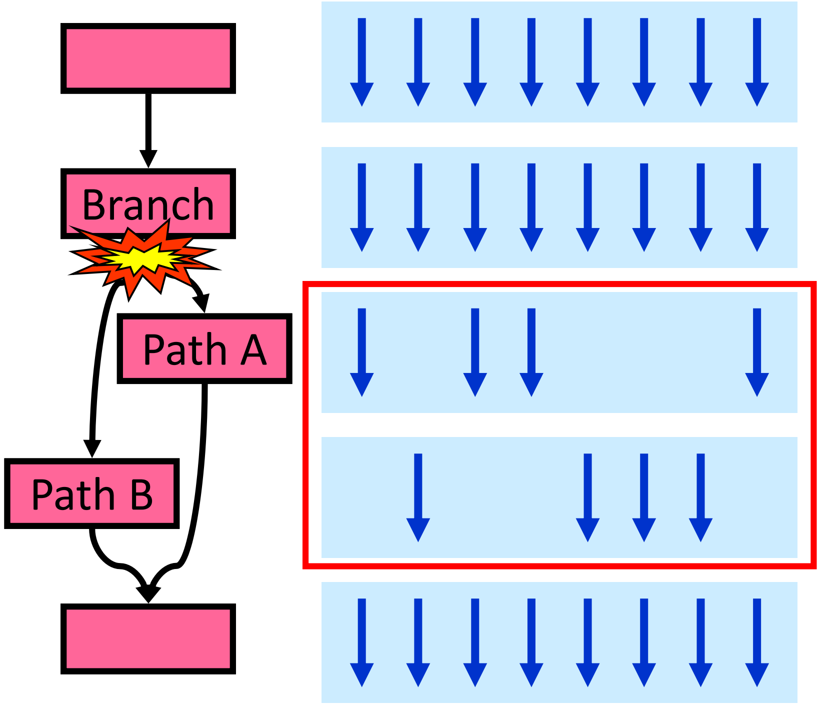
- Once a block is assigned to an SM, it is divided into units called warps.
  - Thread IDs within a warp are consecutive and increasing
  - Warp 0 starts with Thread ID 0
  - Partitioning is always the same
- Warp size is implementation specific.
- Warp is unit of thread scheduling in SMs

Warps

- DO NOT rely on any execution order among warps
  - That is, we cannot tell which warp will finish first.
- Each warp is executed in a SIMD fashion (i.e. all threads within a warp must execute the same instruction at any given time).
  - Problem: branch divergence

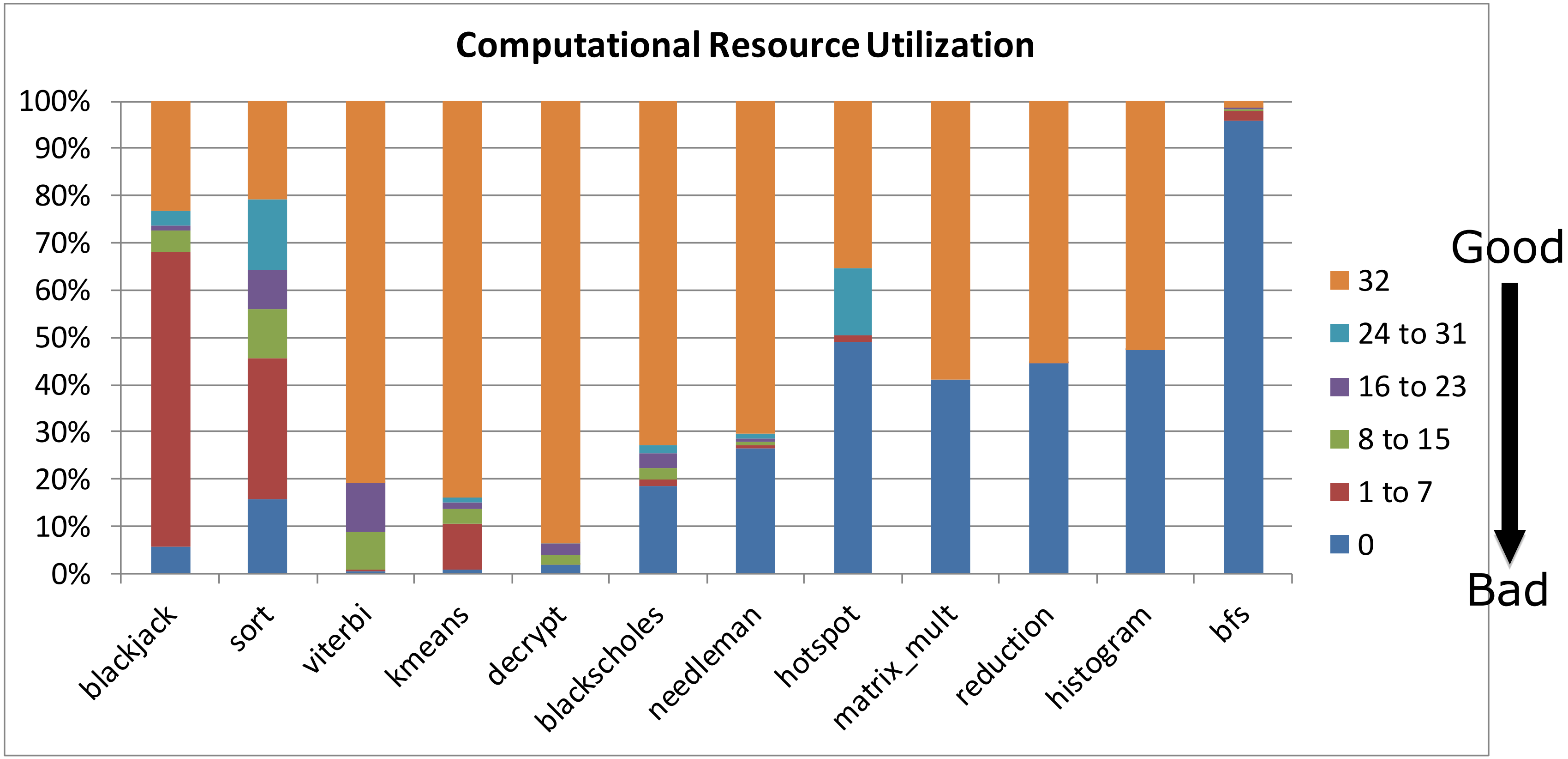
Branch Divergence in Warps

- occurs when threads inside warps branches to different execution paths.



50% performance loss

Example of underutilization



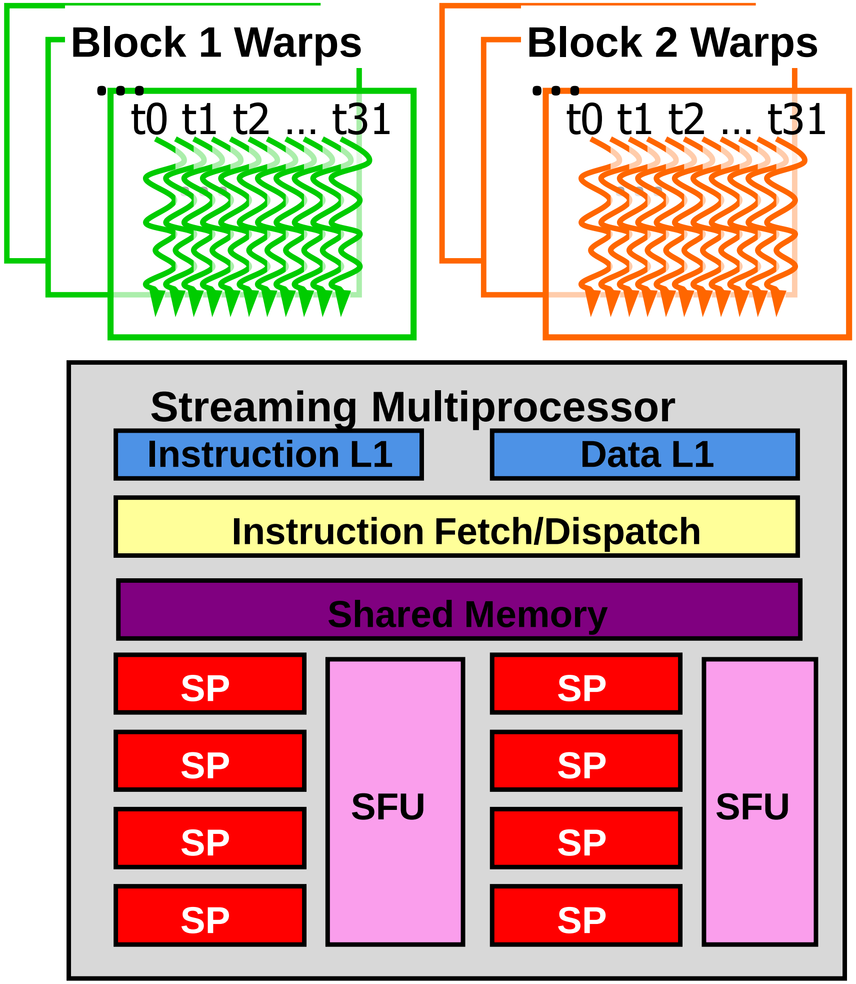
32 warps, 32 threads per warp, round-robin scheduling

Dealing With Branch Divergence

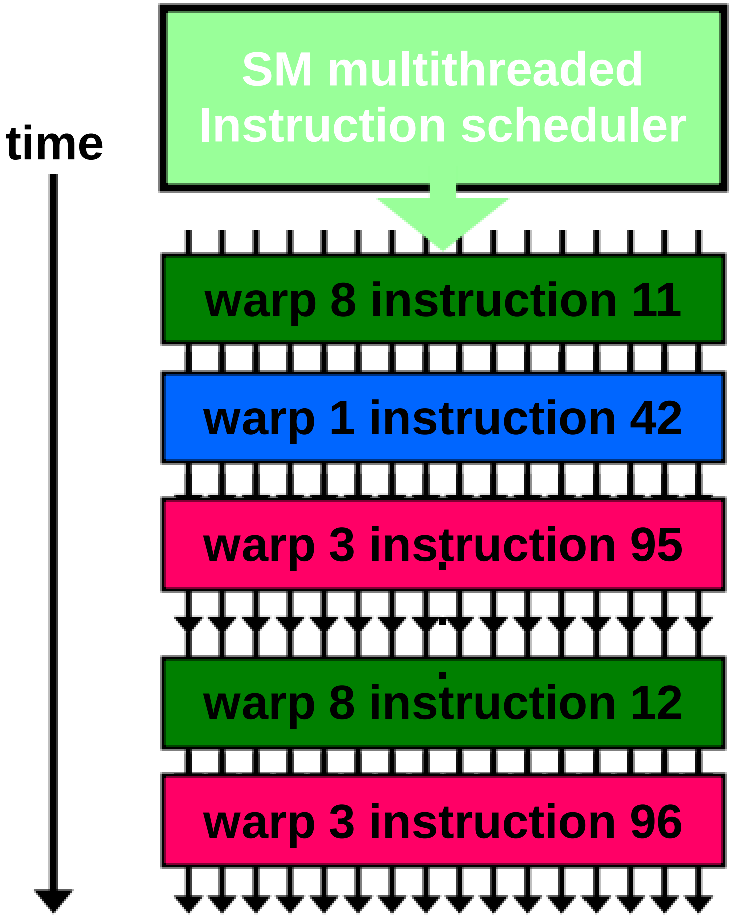
- A common case: avoid divergence when branch condition is a function of thread ID
  - Example with divergence:
    - `If (threadIdx.x > 2) { }`
    - This creates two different control paths for threads in a block
  - Example without divergence:
    - `If (threadIdx.x / WARP_SIZE > 2) { }`
    - Also creates two different control paths for threads in a block
    - Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path
- There is a big body of research for dealing with branch divergence

Latency Tolerance

- When an instruction executed by the threads in a warp must wait for the result of a previously initiated long-latency operation, the warp is not selected for execution -> latency hiding
- Priority mechanism used to schedule ready warps
- Scheduling does not introduce idle time -> zero-overhead thread scheduling
- Scheduling is used for tolerating long-latency operations, such as:
  - pipelined floating-point arithmetic
  - branch instructions



This ability of tolerating long-latency operation is the main reason why GPUs do not dedicate as much chip area to cache memory and branch prediction mechanisms as traditional CPUs.



Exercise: Suppose 4 clock cycles are needed to dispatch the same instruction for all threads in a Warp in G80. If there is one global memory access every 4 instructions, how many warps are needed to fully tolerate 200-cycle memory latency?

Exercise

The GT200 has the following specs (maximum numbers):

- 512 threads/block
- 1024 threads/SM
- 8 blocks/SM
- 32 threads/warp

What is the best configuration for thread blocks to implement matrix multiplications

8x8, 16x16, or 32x32?

Exercise

If a CUDA device's SM can take up to 1,536 threads and up to 4 blocks, which of the following block configs would result in the most number of threads in the SM?

- 128 threads/blk
- 256 threads/blk
- 512 threads/blk   512*3 = 1536 threads in SM
- 1,024 threads/blk

Exercise

- For a vector addition, assume that the vector length is 2,000, each thread calculates one output element, and the thread block size 512 threads. How many threads will be in the grid?
- Given the above, how many warps do you expect to have divergence due to the boundary check on the vector length?

Exercise

A CUDA programmer says that if they launch a kernel with only 32 threads in each block, they can leave out the __syncthreads() instruction wherever barrier synchronization is needed. Do you think this is a good idea? Explain.

Motivational Example

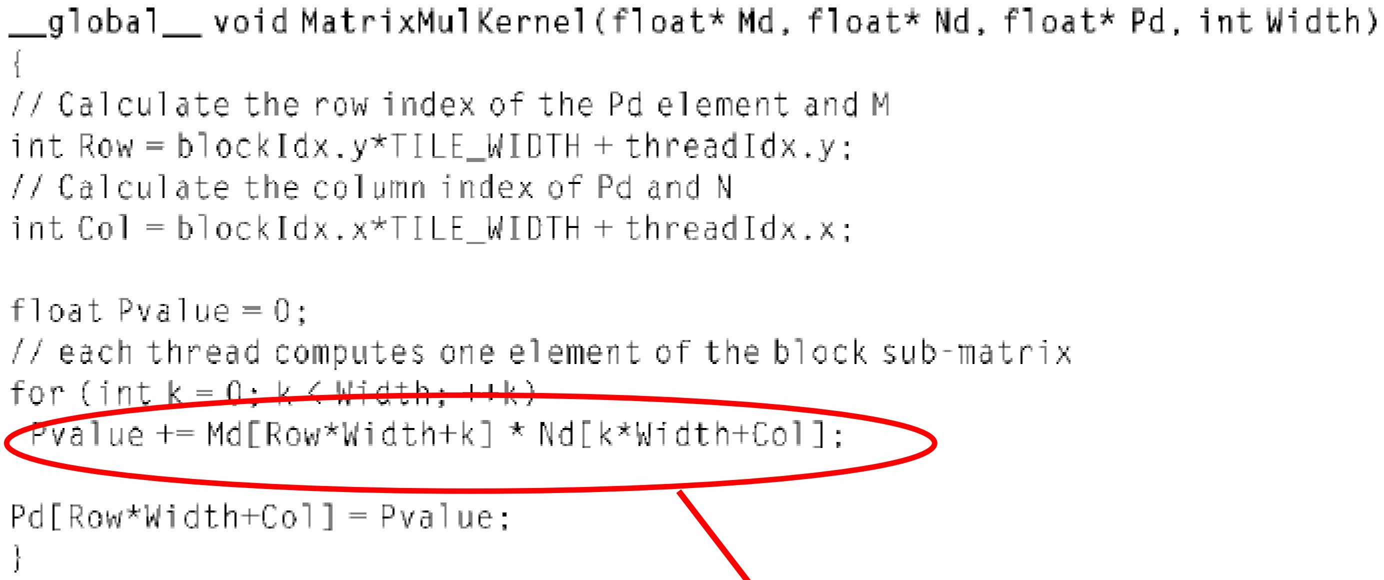
- G80 supports 86.4 GB/s of global memory access
- Single precision floating point = 4 bytes
- Then we cannot load more than 86.4/4 = 21.6 giga single precision data per second
- Theoretical peak performance of G80 is 367gigaflops!

Let's talk about memory …

Computation vs Memory Access

- Compute to global memory access (CGMA) ratio
- Definition: The number of FP calculations performed for each access to the global memory within a region in a CUDA program.

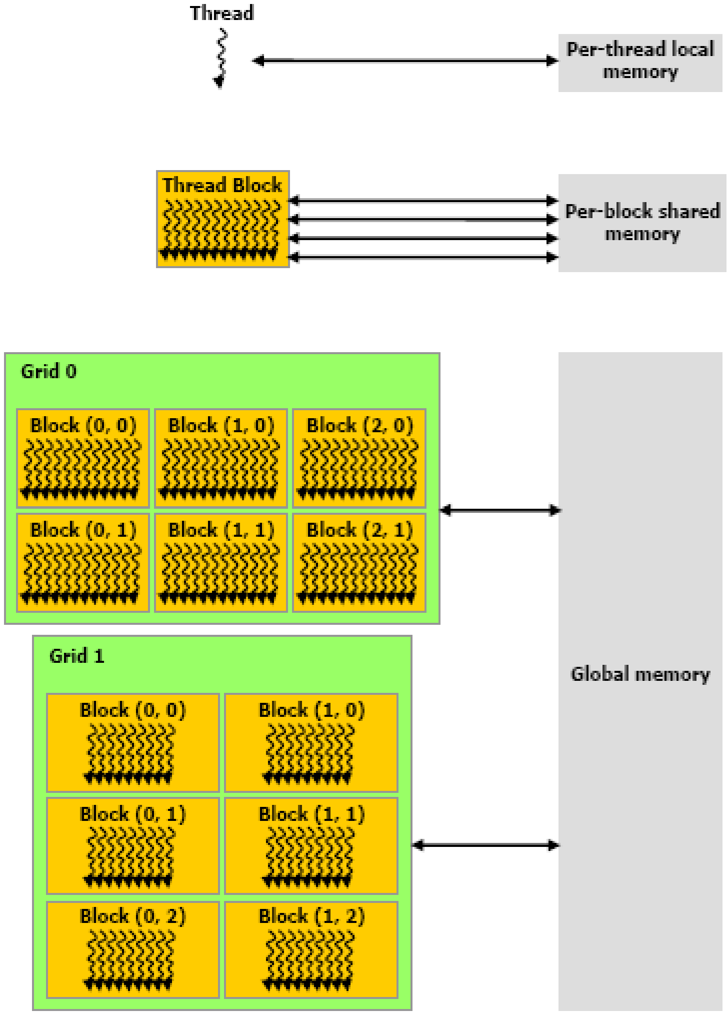
Computation vs Memory Access



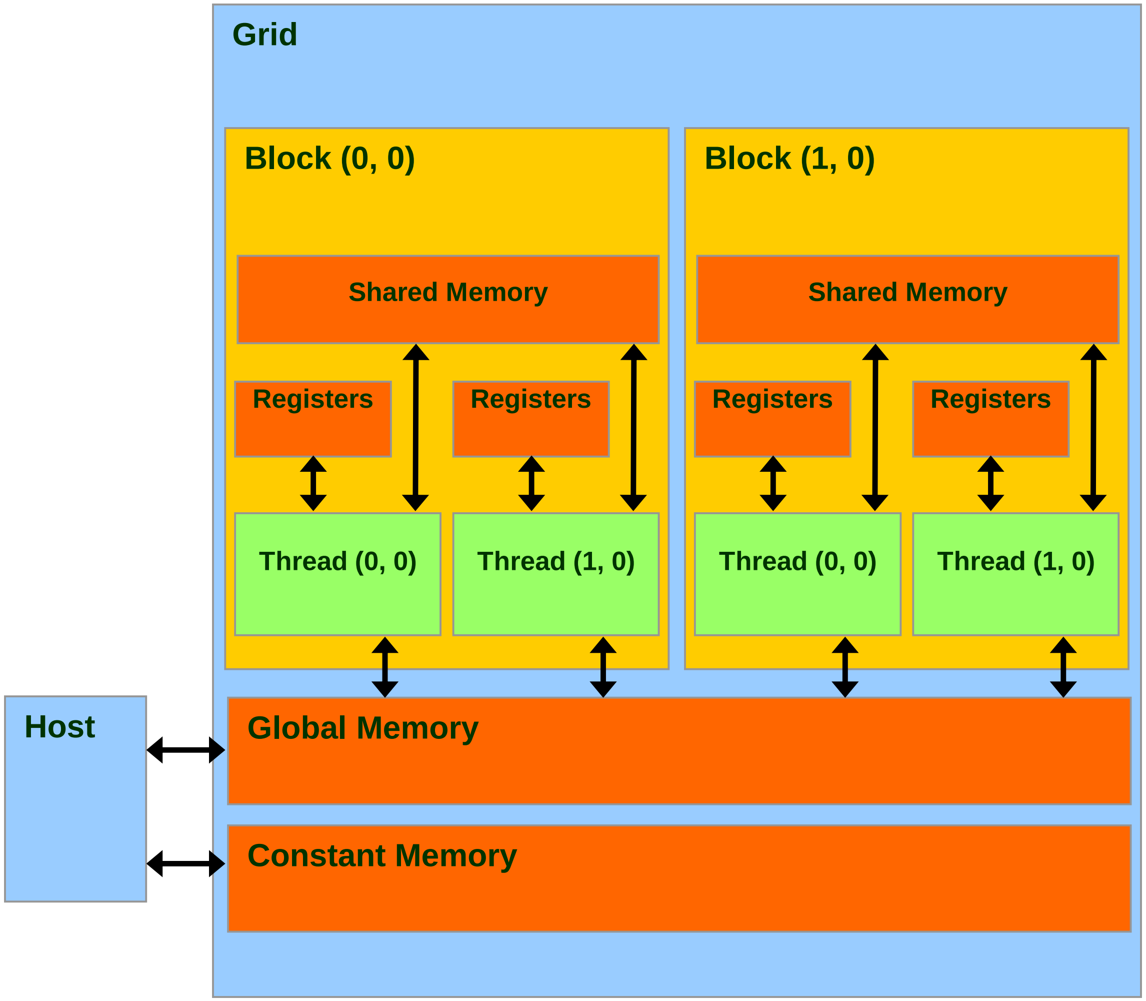
```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
// Calculate the row index of the Pd element and M
int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
// Calculate the column index of Pd and N
int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

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;
}
```

2 memory accesses
1 FP multiplication
1 FP addition
so CGMA = 1

Main Goals



- How to make the best use of the GPU memory system?
- How to deal with hardware limitation?



Grid

Block (0, 0)

Shared Memory

Registers

Registers

Thread (0, 0)

Thread (1, 0)

Block (1, 0)

Shared Memory

Registers

Registers

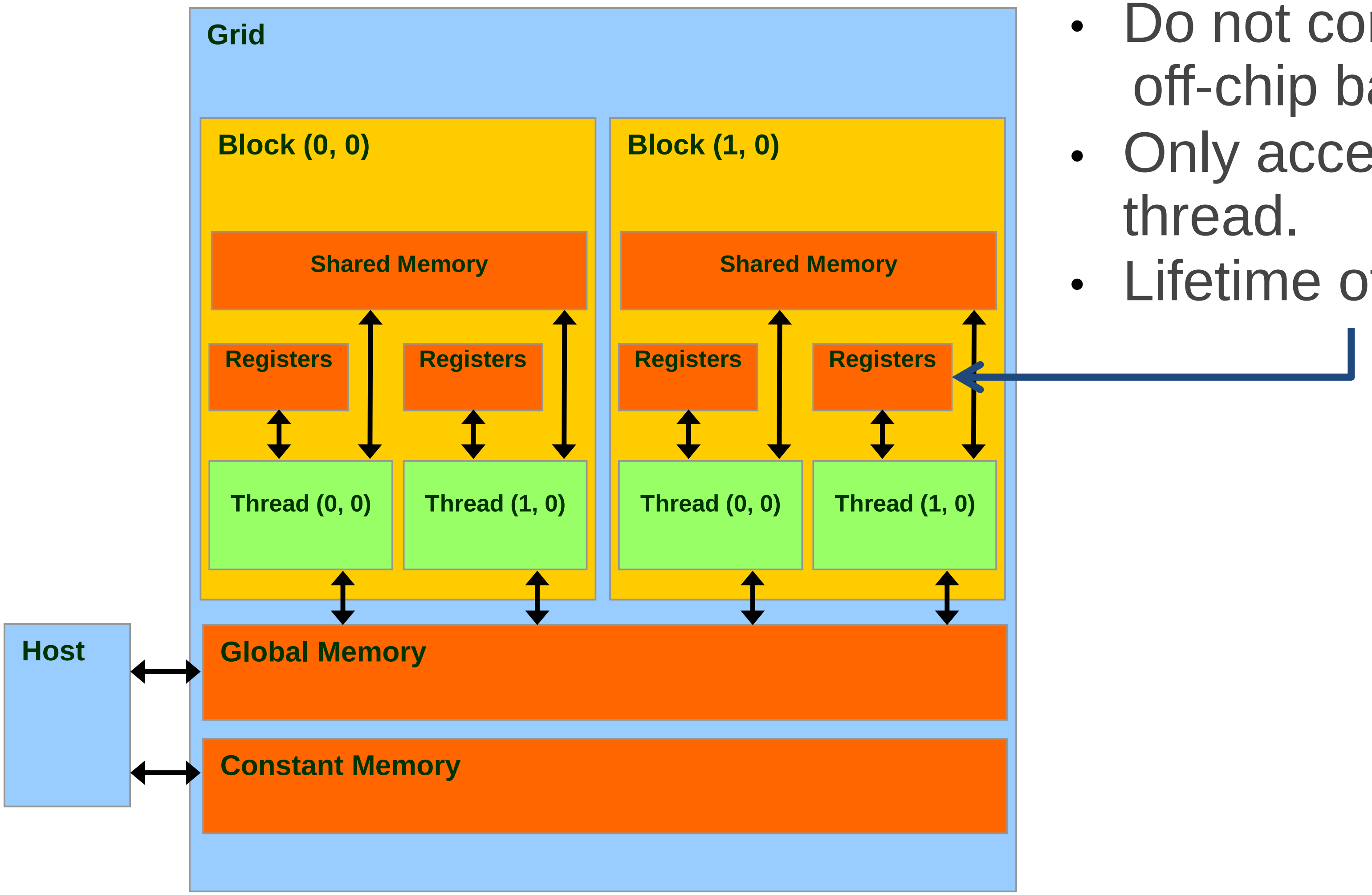
Thread (0, 0)

Thread (1, 0)

Host

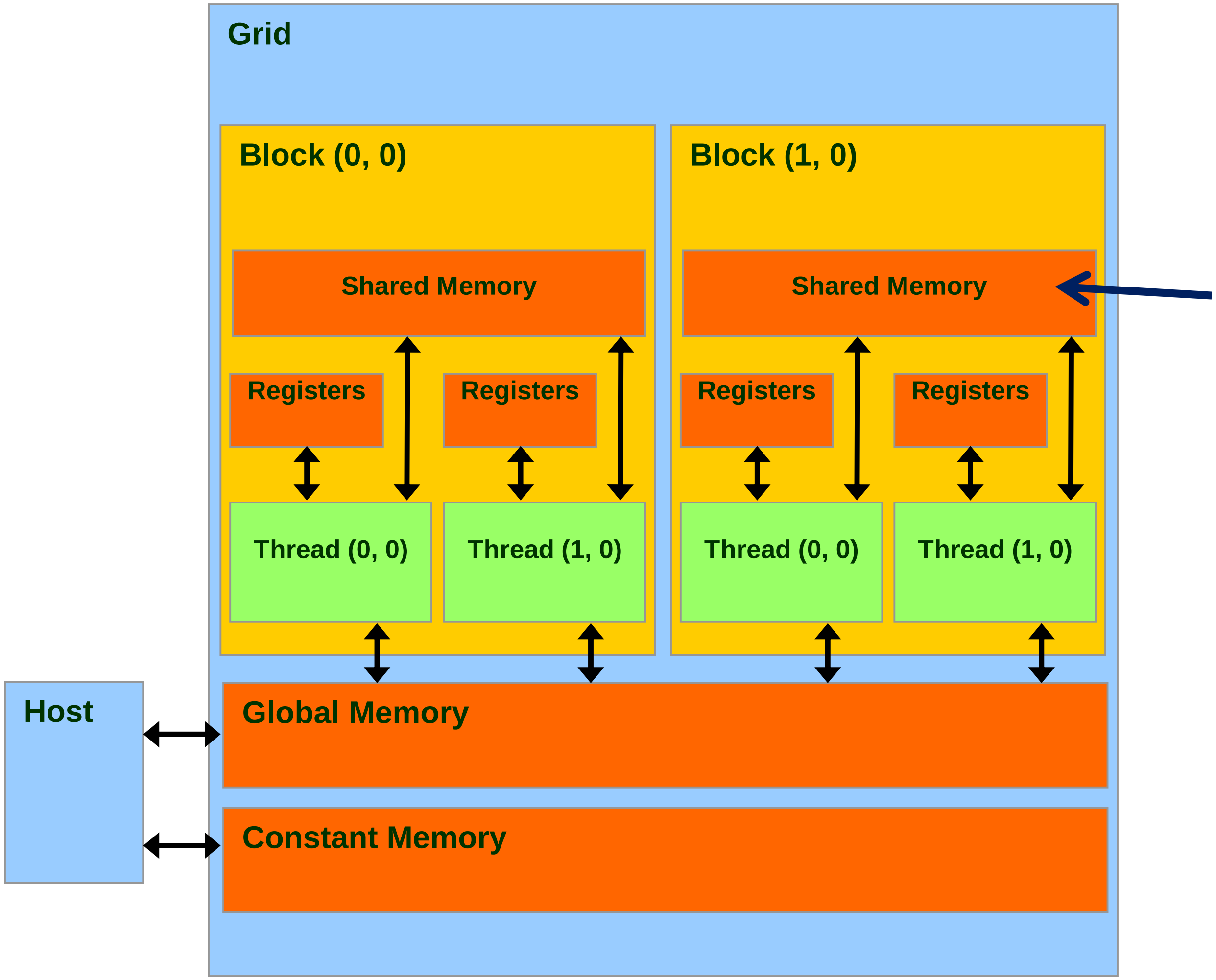
Global Memory

Constant Memory



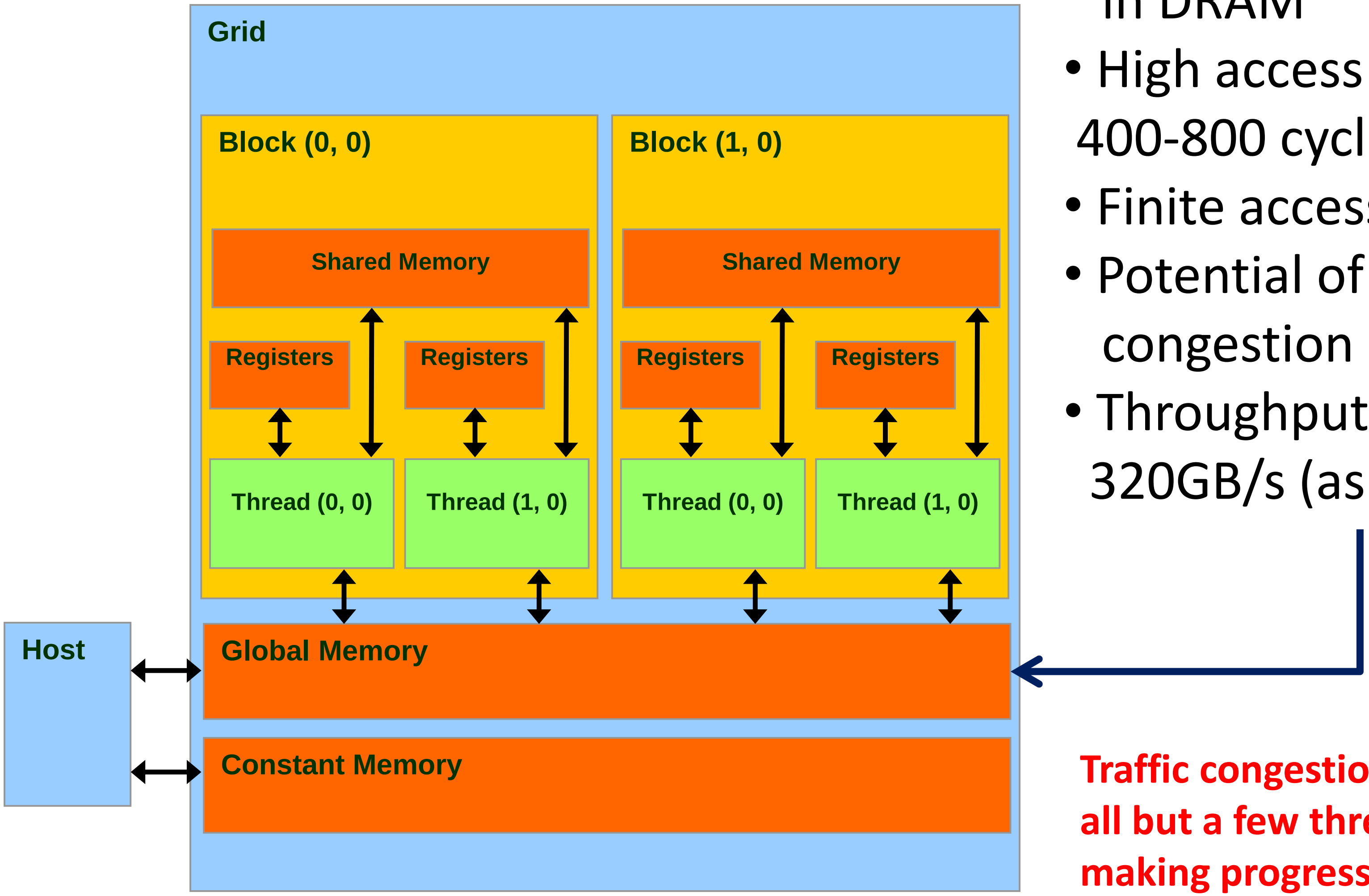
Registers

- Fastest.
- Do not consume off-chip bandwidth.
- Only accessible by a thread.
- Lifetime of a thread



Shared Memory

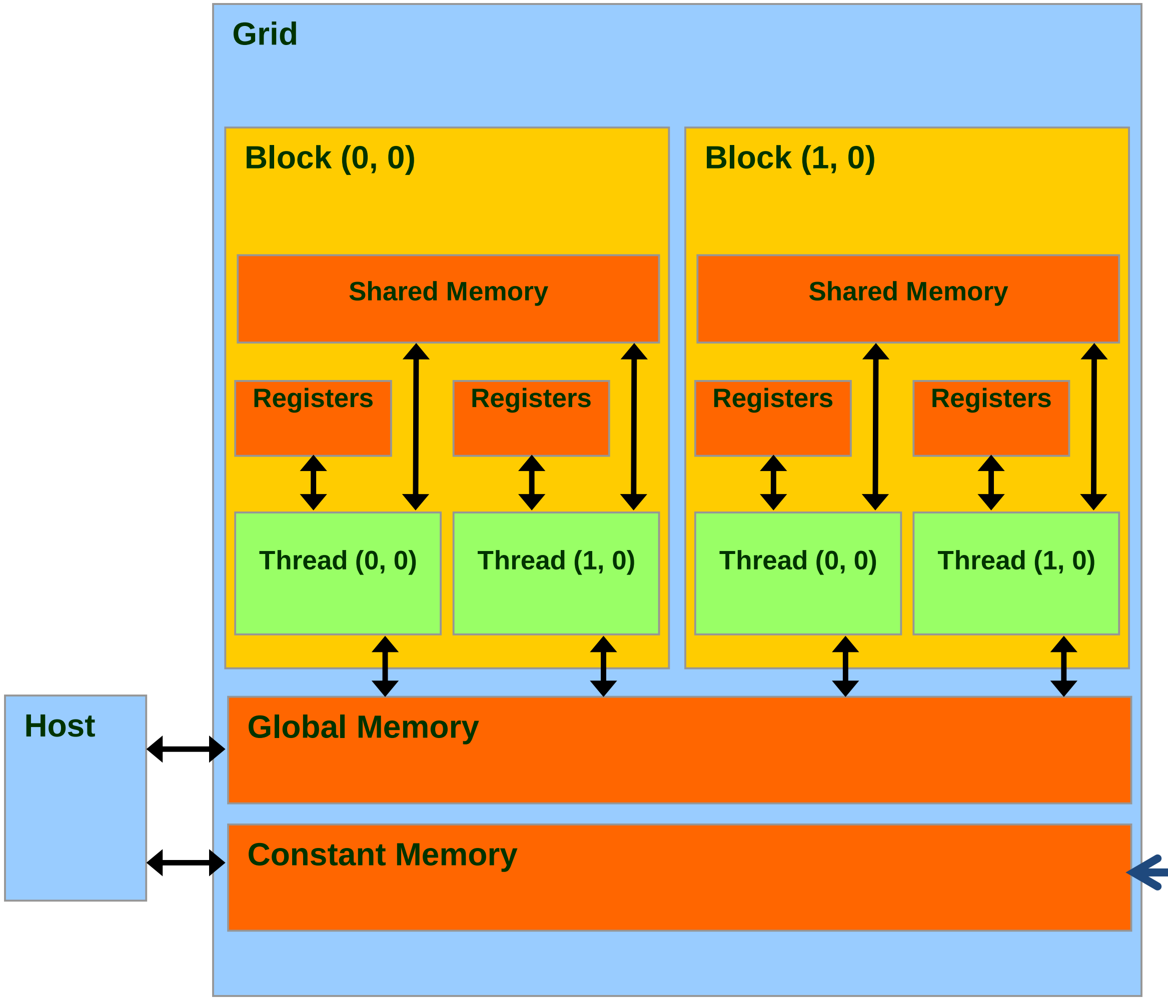
- Extremely fast
- Highly parallel
- Restricted to a block
- Example: Fermi's shared/L1 is 1+TB/s aggregate



Global Memory

- Typically implemented in DRAM
- High access latency: 400-800 cycles
- Finite access bandwidth
- Potential of traffic congestion
- Throughput of about 320GB/s (as of 2017)

Traffic congestion prevents all but a few threads from making progress.

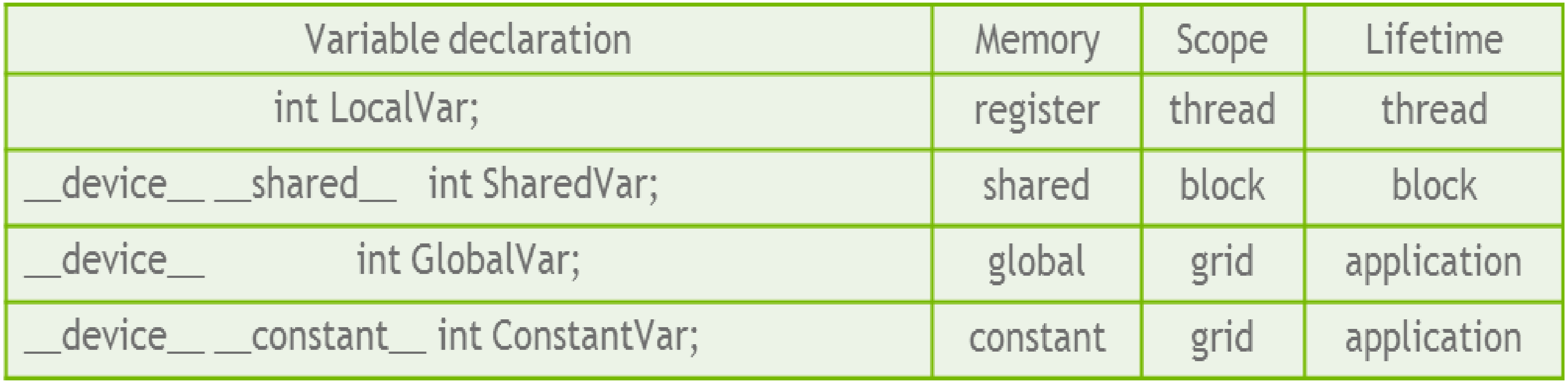


Constant Memory

- Read only
- Short latency and high bandwidth when all threads access the same location

Important!

- Each access to registers involves *fewer* instructions than global memory.
- Aggregate register files bandwidth = ~two orders of magnitude that of the global memory!
- Energy consumed for accessing a value from the register file =~ at least an order of magnitude lower than accessing global memory!
- Shared memory is part of the address space → accessing it requires load/store instructions.

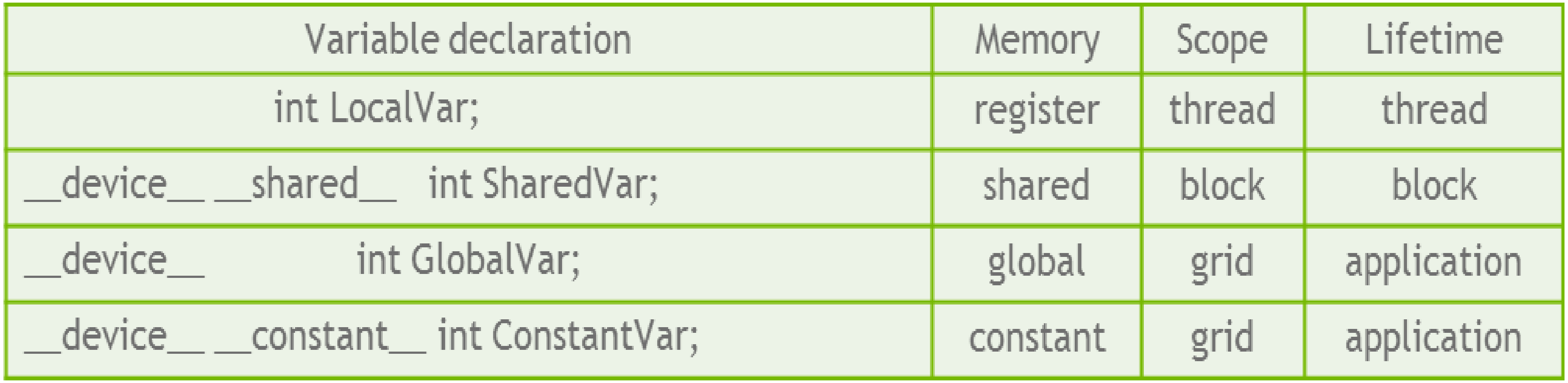


Variable declaration	Memory	Scope	Lifetime
int LocalVar;	register	thread	thread
__device__ __shared__ int SharedVar;	shared	block	block
__device__ int GlobalVar;	global	grid	application
__device__ __constant__ int ConstantVar;	constant	grid	application

Scope: the range of threads that can access a variable
Lifetime: the portion of the program's execution
when the variable is available for use.

__device__ is optional when used with __shared__, or __constant__

Automatic variables reside in a register



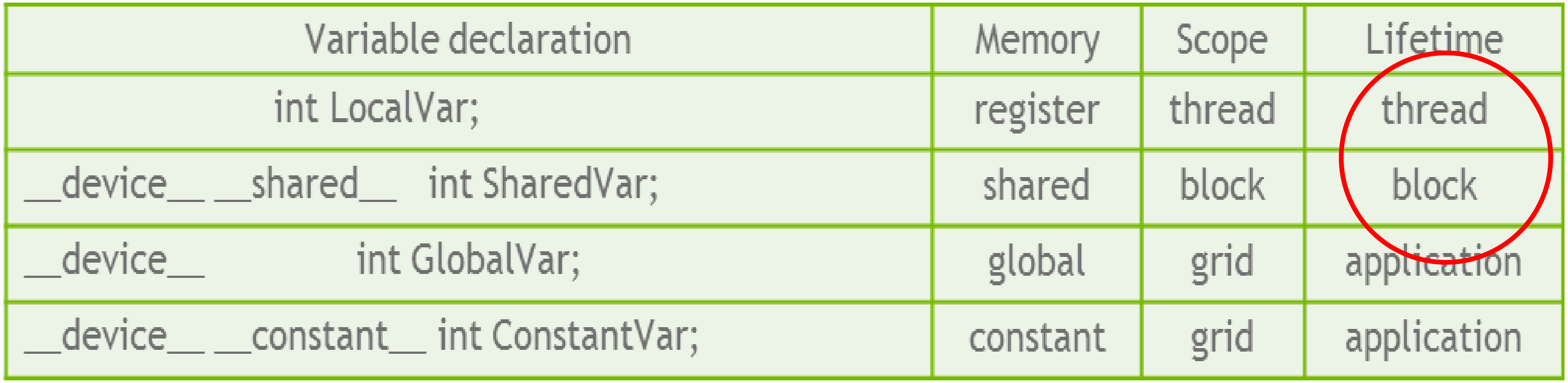
Variable declaration	Memory	Scope	Lifetime
int LocalVar;	register	thread	thread
__device__ __shared__ int SharedVar;	shared	block	block
__device__ int GlobalVar;	global	grid	application
__device__ __constant__ int ConstantVar;	constant	grid	application

Automatic array variables local to a thread reside in **local memory.**

local memory

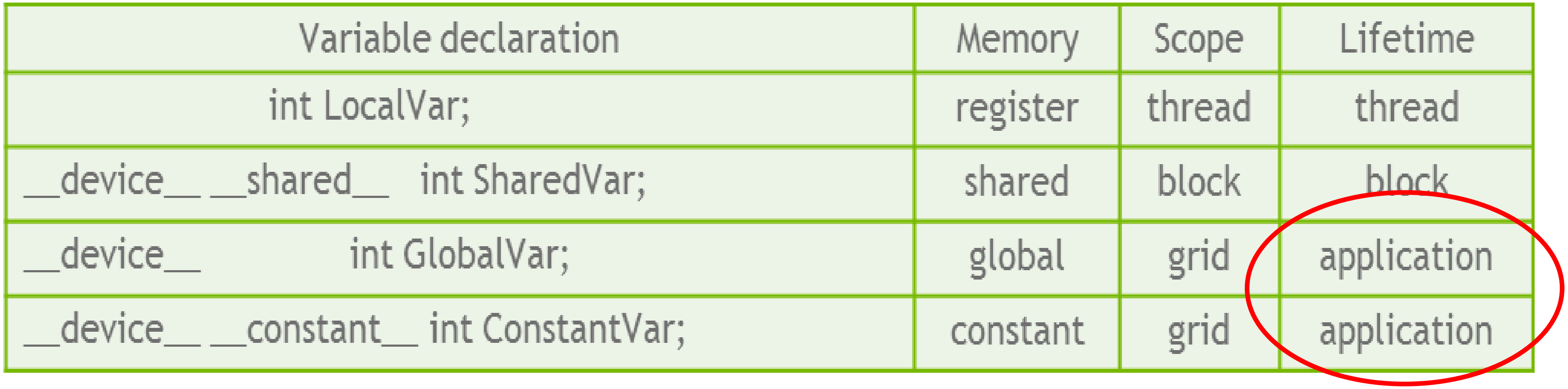
↓

Does not physically exist. It is an abstraction to the local scope of a thread. Actually put in global memory by the compiler.



Variable declaration	Memory	Scope	Lifetime
int LocalVar;	register	thread	thread
__device__ __shared__ int SharedVar;	shared	block	block
__device__ int GlobalVar;	global	grid	application
__device__ __constant__ int ConstantVar;	constant	grid	application

The variable must be declared within the kernel function body; and will be available only within the kernel code.



Variable declaration	Memory	Scope	Lifetime
int LocalVar;	register	thread	thread
__device__ __shared__ int SharedVar;	shared	block	block
__device__ int GlobalVar;	global	grid	application
__device__ __constant__ int ConstantVar;	constant	grid	application

The variable must be declared outside of any function.

- Declaration of constant variables must be outside any function body.
- Currently total size of constant variables in an application is limited to 64KB.

By declaring a CUDA variable in one of the CUDA memory types, a CUDA programmer dictates the visibility and access speed of the variable.

A Note About Local Memory

- Local for each thread
- Check local memory usage: nvcc –Xptxas –v
  - will print number of lmem bytes for each kernel, if the kernel uses local memory.
- Used for:
  - Arrays not accessed by constant indices
  - Large structures and arrays that do not fit into registers
  - If thread uses a lot of registers the runtime uses register spilling into local memory to increase concurrency in the SM.
- Internally, the hardware has a limit in the number of registers per thread.
- As programmer you can specify the maximum registers a thread can use
  - nvcc –maxrregcount num …
  - If num > hardware limit, then register spilling happens.

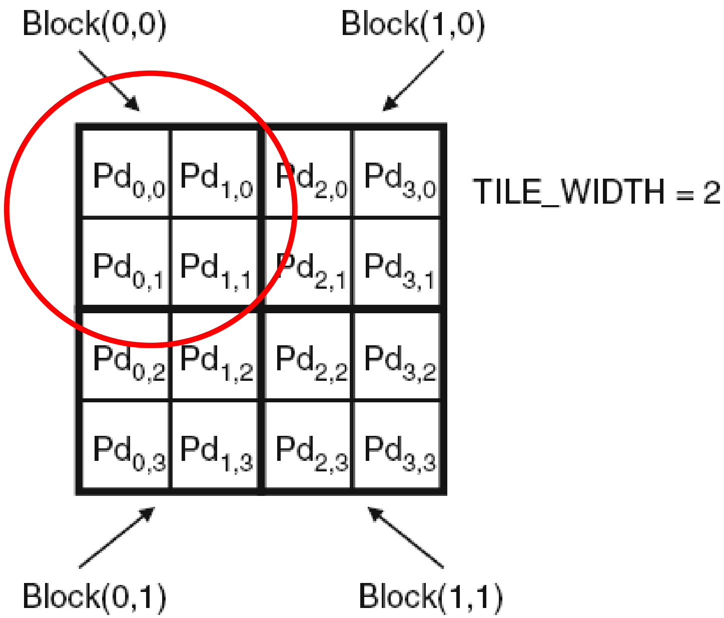
Reducing Global Memory Traffic

- Global memory access is performance bottleneck.
- The lower CGMA the lower the performance
- Reducing global memory access enhances performance.
- A common strategy is **tiling**: partition the data into subsets called tiles, such that each tile fits into the shared memory. We saw this earlier.

Outline of Tiling

- Identify a tile of global memory contents that are accessed by multiple threads
- Load the tile from global memory into on-chip memory (e.g. shared memory)
- Use barrier synchronization to make sure that all threads are ready to start the phase
- Have the multiple threads to access their data from the on-chip memory
- Use barrier synchronization to make sure that all threads have completed the current phase
- Move on to the next tile

Back to Matrix Multiplication

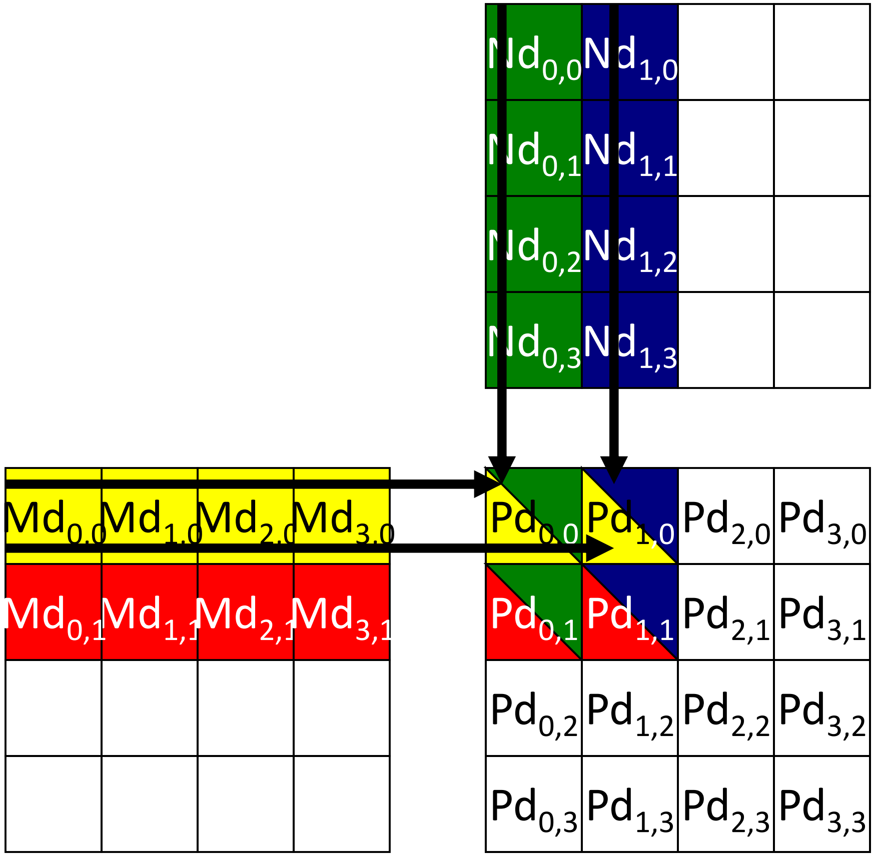


Block(0,0) Block(1,0)

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

TILE_WIDTH = 2

Block(0,1) Block(1,1)



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

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

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

Back to Matrix Multiplication

Access order

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

Back to Matrix Multiplication

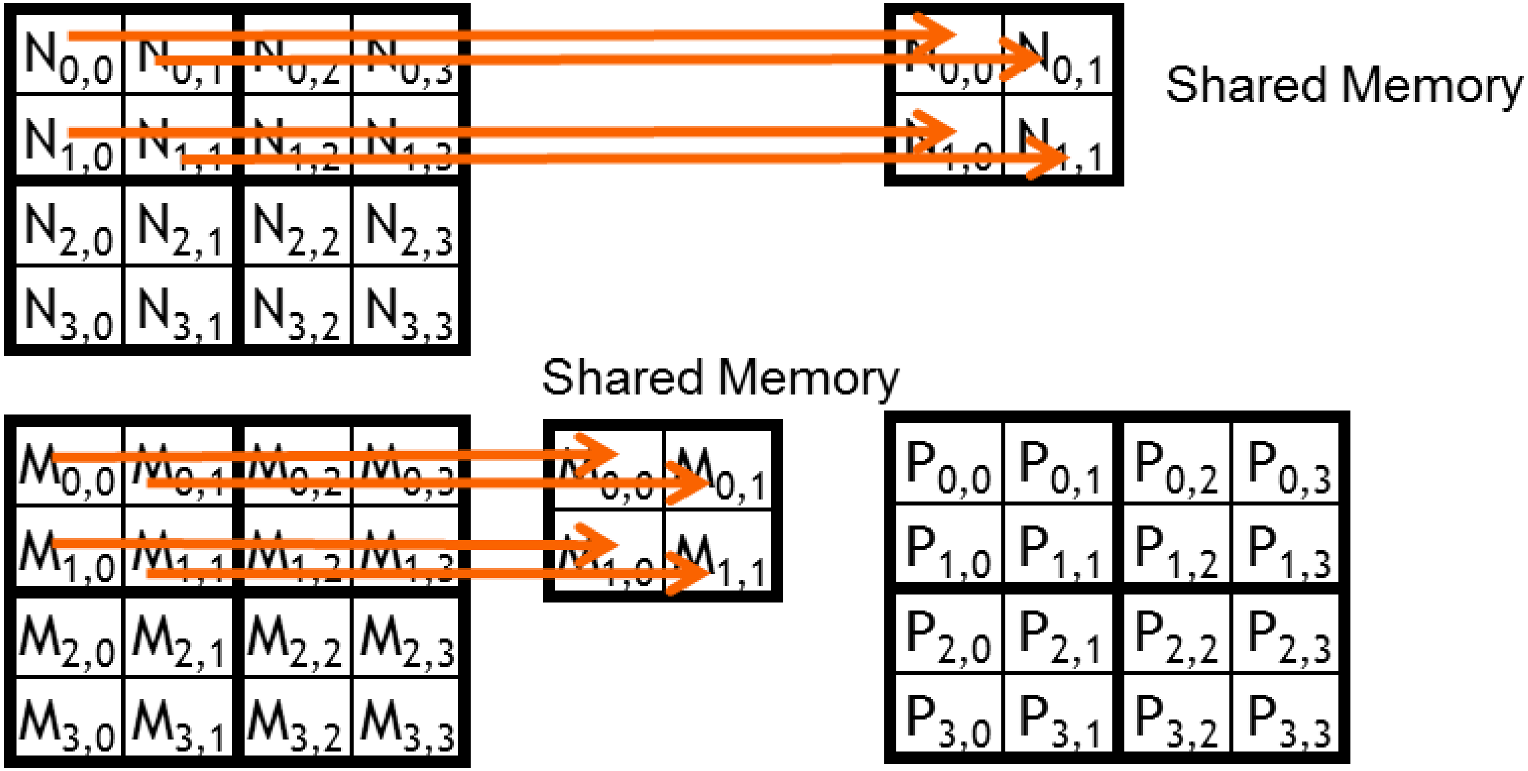
- The basic idea is to make threads that use common elements collaborate.
- Each thread can load different elements into the shared memory before calculations.
- These elements will be used by the thread that loaded them and other threads that share them.

Back to Matrix Multiplication

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

Time →

Back to Matrix Multiplication



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

$N_{0,0}$	$N_{0,1}$
$N_{1,0}$	$N_{1,1}$

Shared Memory

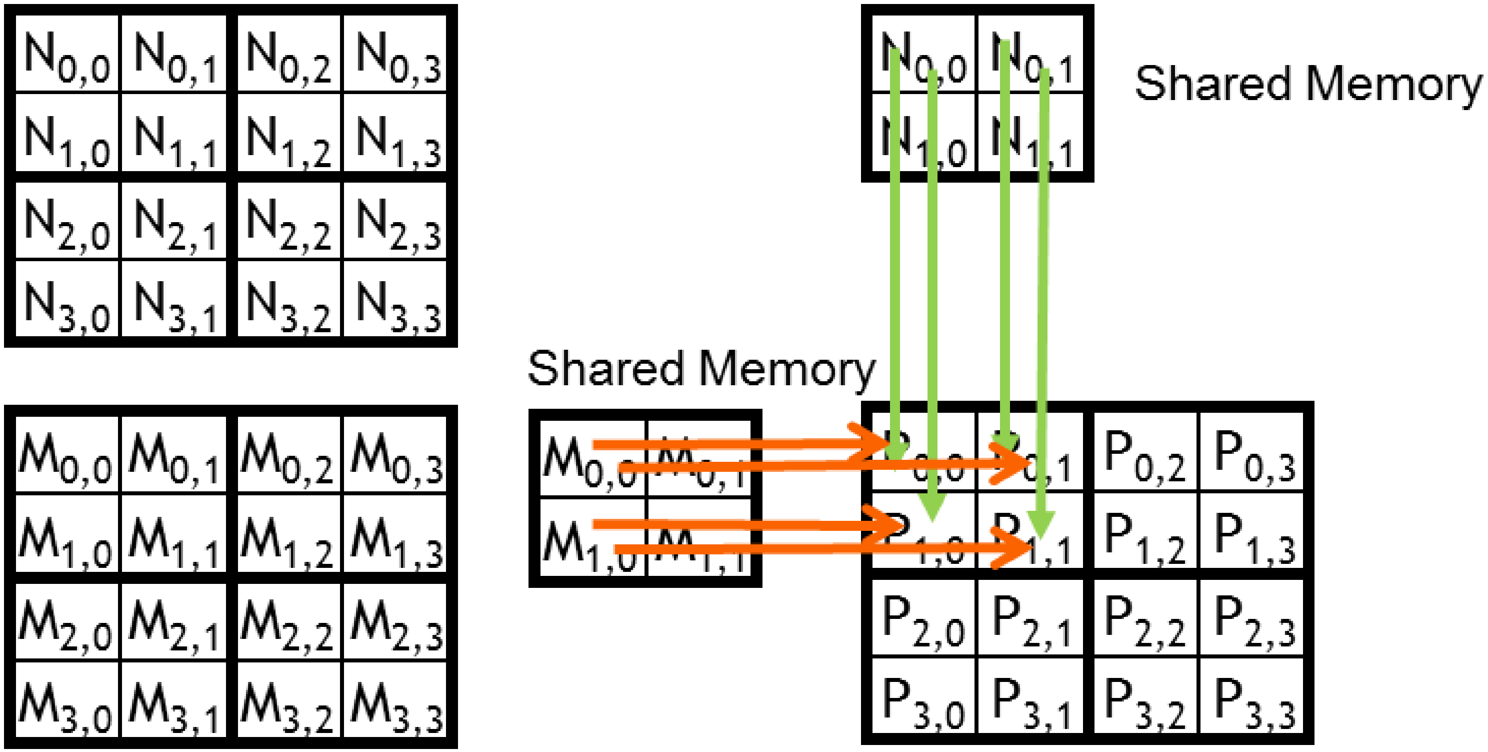
Shared Memory

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

$M_{0,0}$	$M_{0,1}$
$M_{1,0}$	$M_{1,1}$

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

Back to Matrix Multiplication



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

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

Shared Memory

$N_{0,0}$	$N_{0,1}$
$N_{1,0}$	$N_{1,1}$

Shared Memory

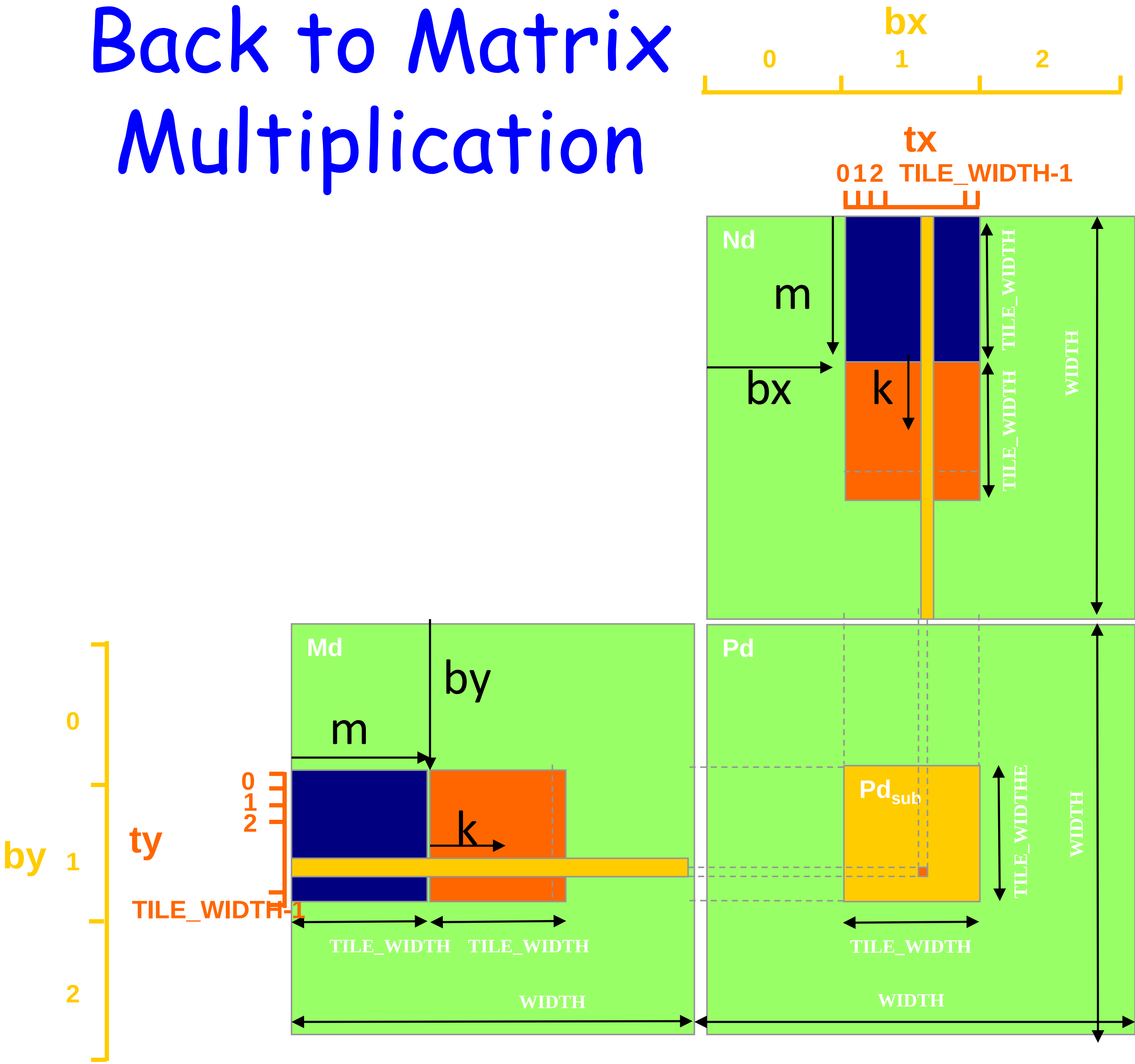
$M_{0,0}$	$M_{0,1}$
$M_{1,0}$	$M_{1,1}$

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

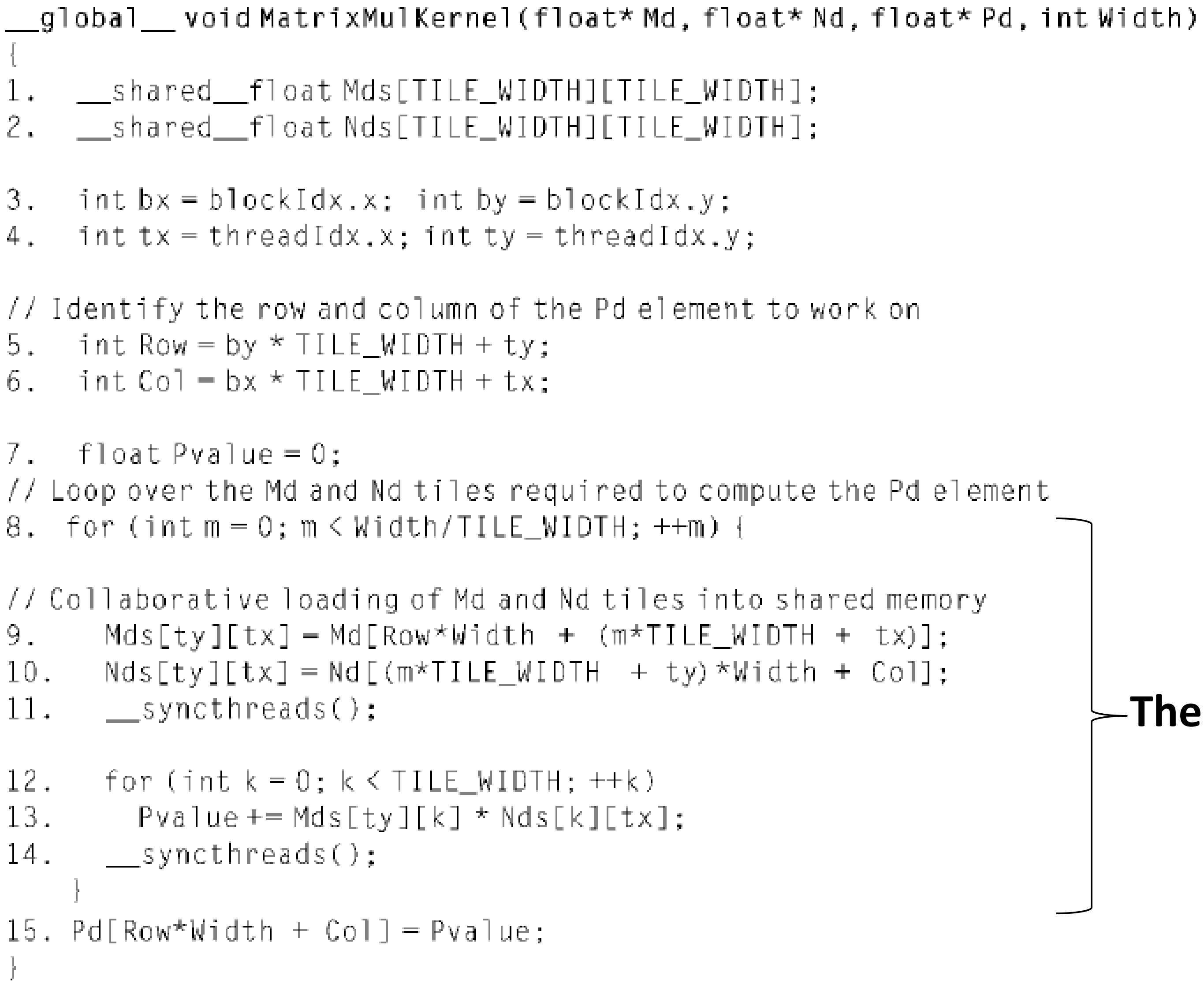
Back to Matrix Multiplication

- Potential reduction in global memory traffic in matrix multiplication example is proportional to the dimension of the blocks used.
  - With NxN blocks the potential reduction would be N
- If an input matrix is of dimension M and the tile size is TILE_WIDTH, the dot product will be performed in M/TILE_WIDTH phases.

Back to Matrix Multiplication



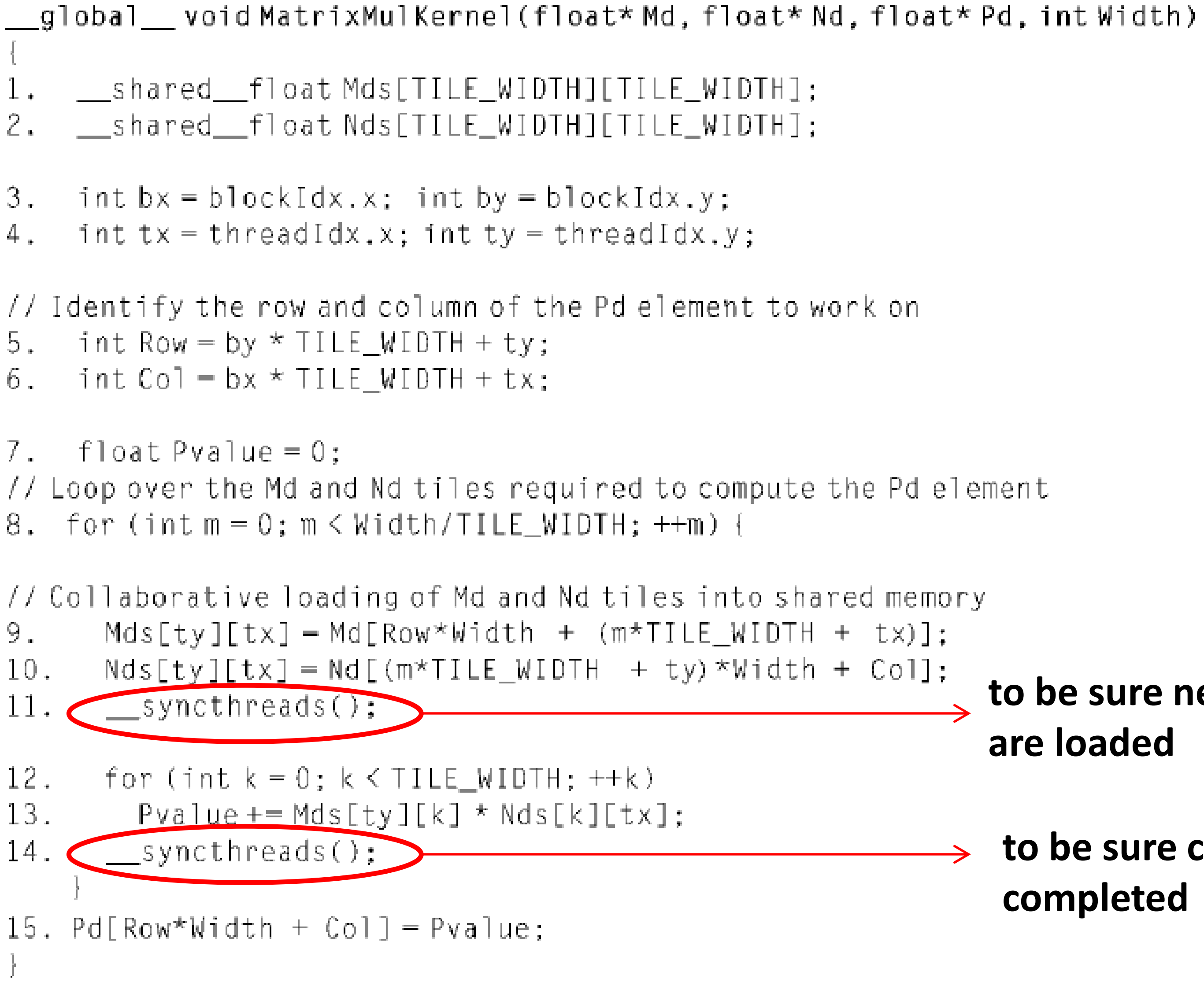
Back to Matrix Multiplication



```
__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) {

// Collaborative loading of Md and Nd tiles into shared memory
      Mds[ty][tx] = Md[Row*Width  +  (m*TILE_WIDTH  +  tx)];
      Nds[ty][tx] = Nd[(m*TILE_WIDTH   + ty)*Width  +  Col];
      __syncthreads();

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

The Phases

Back to Matrix Multiplication



```
__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) {

// Collaborative loading of Md and Nd tiles into shared memory
      Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
      Nds[ty][tx] = Nd[(m*TILE_WIDTH + ty)*Width + Col];
      __syncthreads();

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

to be sure needed elements are loaded

to be sure calculations are completed

Exercise

Can we use shared memory to reduce global memory bandwidth for matrix addition?

Do you Remember the G80 example?

- 86.4 GB/s global memory bandwidth
- In matrix multiplication if we use 16x16 blocks -> reduction in memory traffic by a factor of 16
- Global memory can now support [(86.4/4) x 16] = 345.6 gigaflops -> very close to the peak (367gigaglops).

Memory As Limiting Factor to Parallelism

- Limited shared memory limits the number of threads that can execute simultaneously in SM for a given application
  - The more memory locations each thread requires, the fewer the number of threads per SM
  - Same applies to registers

Memory As Limiting Factor to Parallelism

- Example: **Shared memory**
  - G80 has 16KB of shared memory per SM
  - SM accommodates up to 8 blocks
  - To reach this maximum each block must not exceed 16KB/8 = 2KB of memory.
  - e.g. if each block uses 5KB -> no more than 3 blocks can be assigned to each SM

Registers As Limiting Factor to Parallelism

- Example: **Registers**
  - G80 has 8K registers per SM -> 128K registers for entire processor.
  - G80 can accommodate up to 768 threads per SM
  - To fill this capacity each thread can use only 8K/768 = 10 registers.
  - If each thread uses 11 registers -> threads per SM are reduced -> per block granularity
  - e.g. if block contains 256 threads the number of threads will be reduced by 256 -> lowering the number of threads/SM from 768 to 512 (i.e. 1/3 reduction of threads!)

Utilization Analysis Example

- Lets assume that:
  - We have a kernel that requires 48 registers per thread
  - Assume GTX 580 GPU (CC 2.0, 16SMs, 32k registers/SM)
  - Execution configuration is a grid of 4x5x3 blocks, each 100 threads
- Each block requires 100*48=4800 registers
- The grid is made of 4*5*3 = 60 blocks that need to be distributed to the 16 SMs of the card.There will be 12 SMs that will receive 4 blocks and 4 SMs that will receive 3 blocks → inefficient
- Additionally, each of the 100-thread blocks would be split into ceil[100/warp_size] = ceil[100/32]
  - warps. The first three warps would have 32 threads and the last would have 4 threads ! So during the execution of the last warp of each block most of the SPs will be idle → inefficient

Source: ***Multicore and GPU Programming: An Integrated Approach*** *by G. Barlas*

Myths About CUDA

- GPUs have very wide (1000s) SIMD machines
  - No, a CUDA Warp is only 32 threads
- Branching is not possible on GPUs
  - Incorrect.
- GPUs are power-inefficient
  - Nope, performance per watt is quite good
- CUDA is only for C or C++ programmers
  - Not true, there are third party wrappers for Java, Python, and more

Conclusions

- Using memory effectively will likely require the redesign of the algorithm.
- The only safe way to synchronize threads in different blocks is to terminate the kernel and start a new kernel for the activities after the synchronization point
- The ability to reason about hardware limitations when developing an application is a key concept of computational thinking.