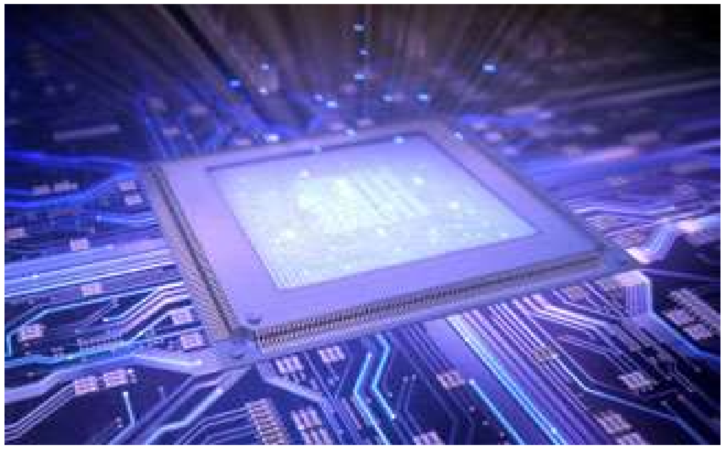
Parallel Computing

CUDA III

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com



Quick Exercises

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
- 1,024 threads/blk

Quick Exercises

- 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?

Quick Exercises

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.

A 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!

How come??

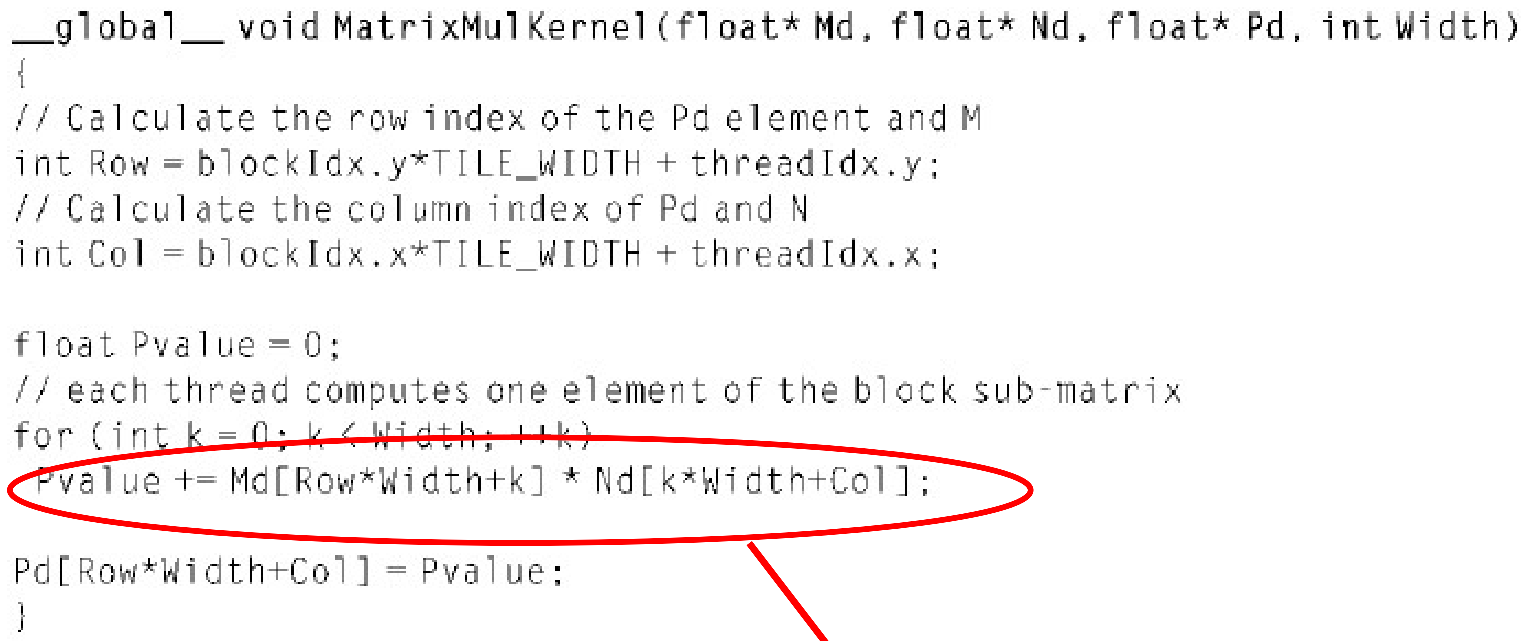
Computation vs Memory Access

- 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.

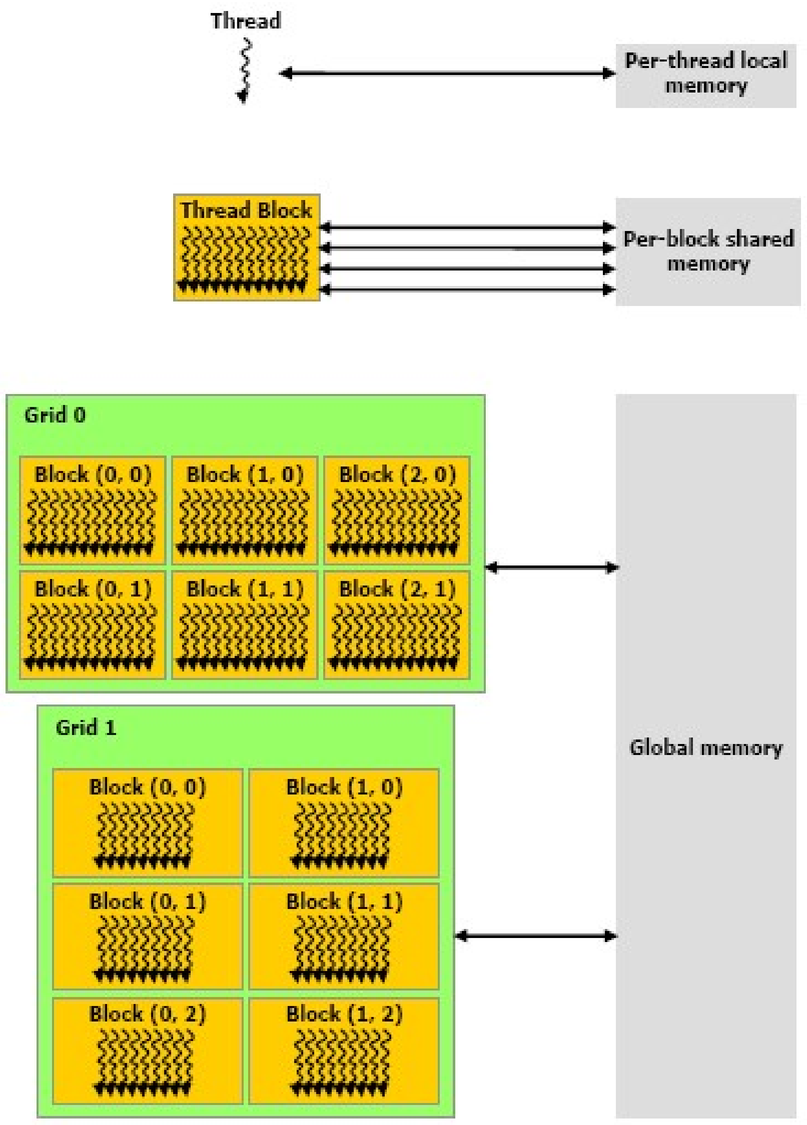
Computation vs Memory Access



```
__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;
}
```

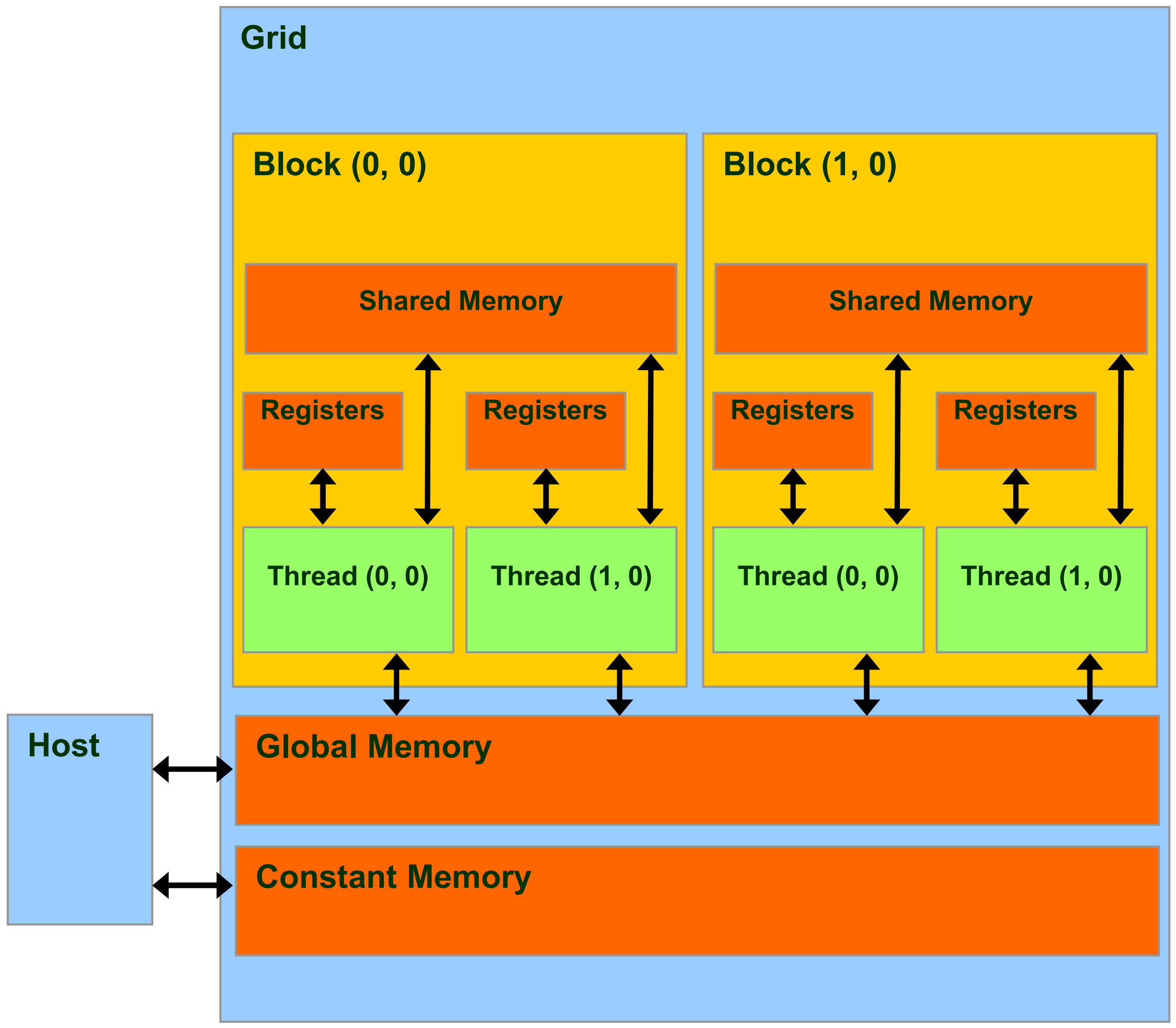
2 memory accesses
1 FP multiplication
1 FP addition
so CGMA = 1

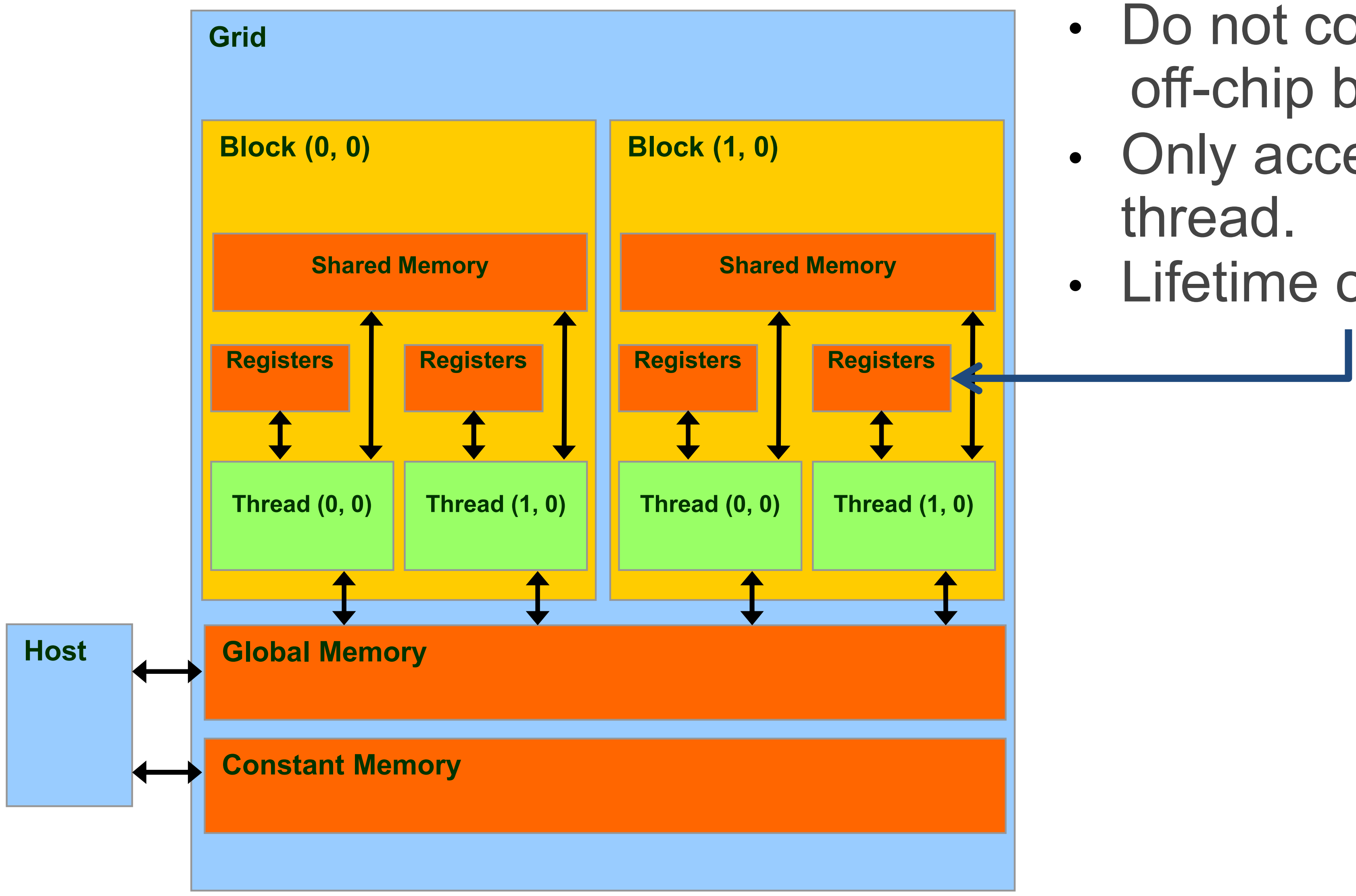
Main Goals for This Lecture



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

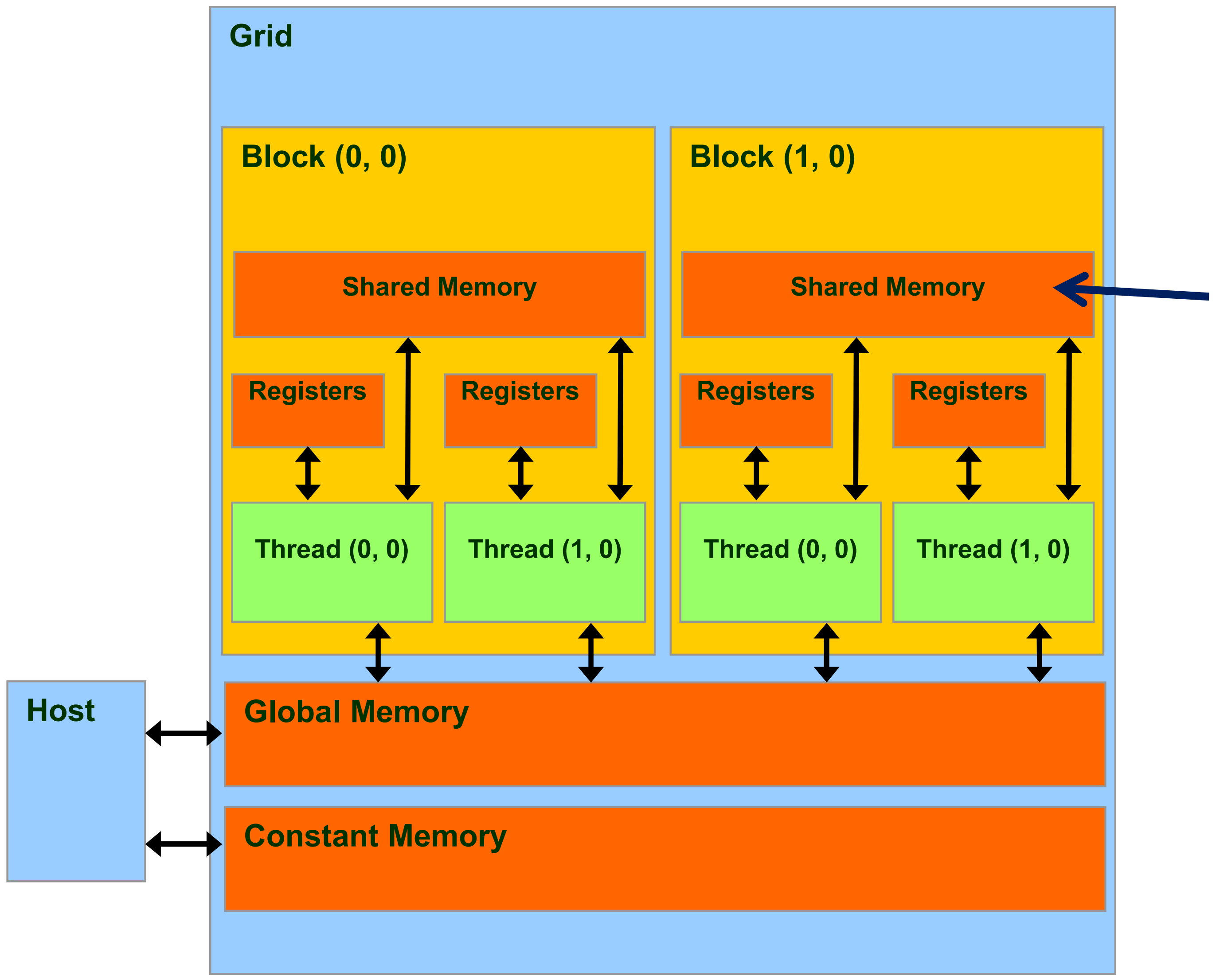
Measure of success: higher CGMA


Grid
Block (0, 0)
Block (1, 0)
Shared Memory
Shared Memory
Registers
Registers
Registers
Registers
Thread (0, 0)
Thread (1, 0)
Thread (0, 0)
Thread (1, 0)
Host
Global Memory
Constant Memory



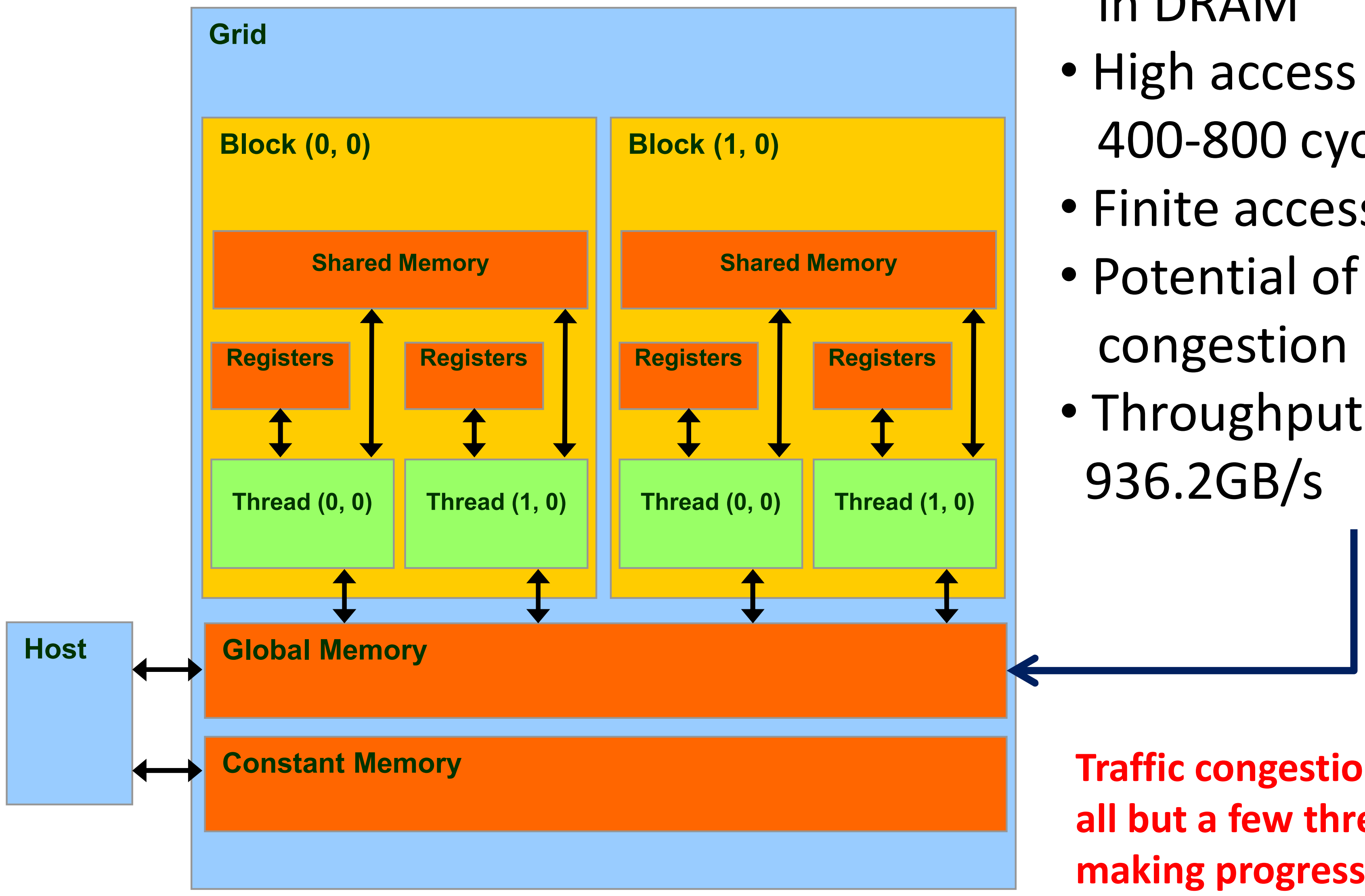
Registers

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



Shared Memory

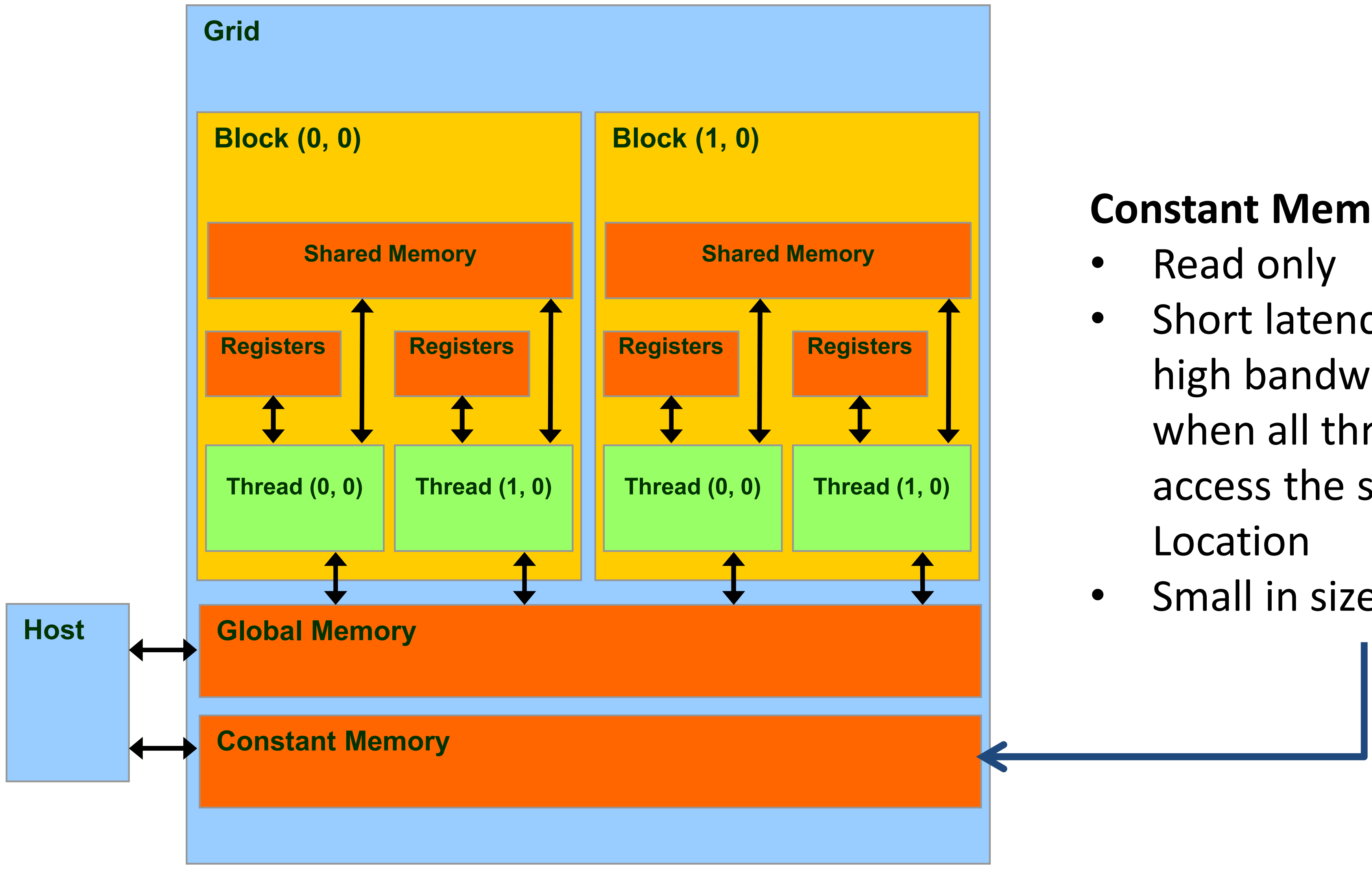
- Extremely fast
- Highly parallel
- Restricted to a block
- Example: Fermi's shared/L1 is 1+TB/s aggregate



Global Memory

- Typically implemented in DRAM
- High access latency: 400-800 cycles
- Finite access bandwidth
- Potential of traffic congestion
- Throughput up to 936.2GB/s

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

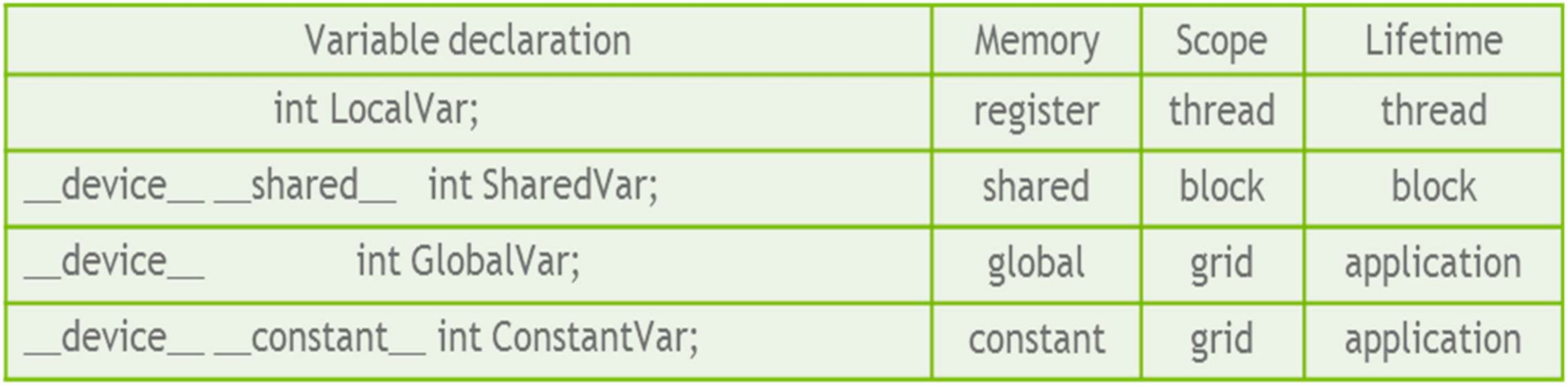


Constant Memory

- Read only
- Short latency and high bandwidth when all threads access the same Location
- Small in size ~64KB

Important!

- Each access to registers involves *fewer machine-level* 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.

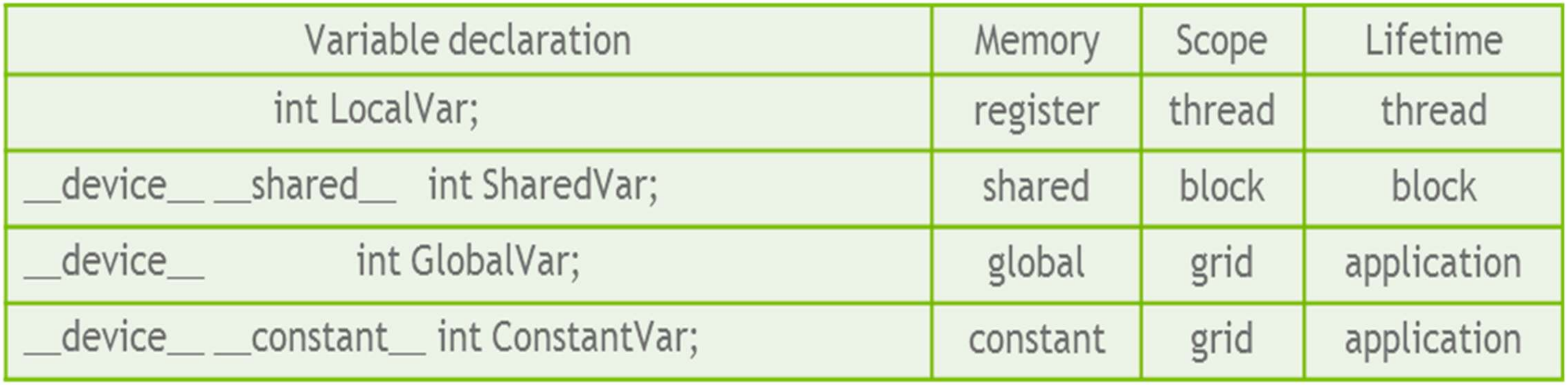


Variable declaration	Memory	Scope	Lifetime
int LocalVar;	register	thread	thread
__device__ __shared__ int SharedVar;	shared	block	block
__device__ int GlobalVar;	global	grid	application
__device__ __constant__ int ConstantVar;	constant	grid	application

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

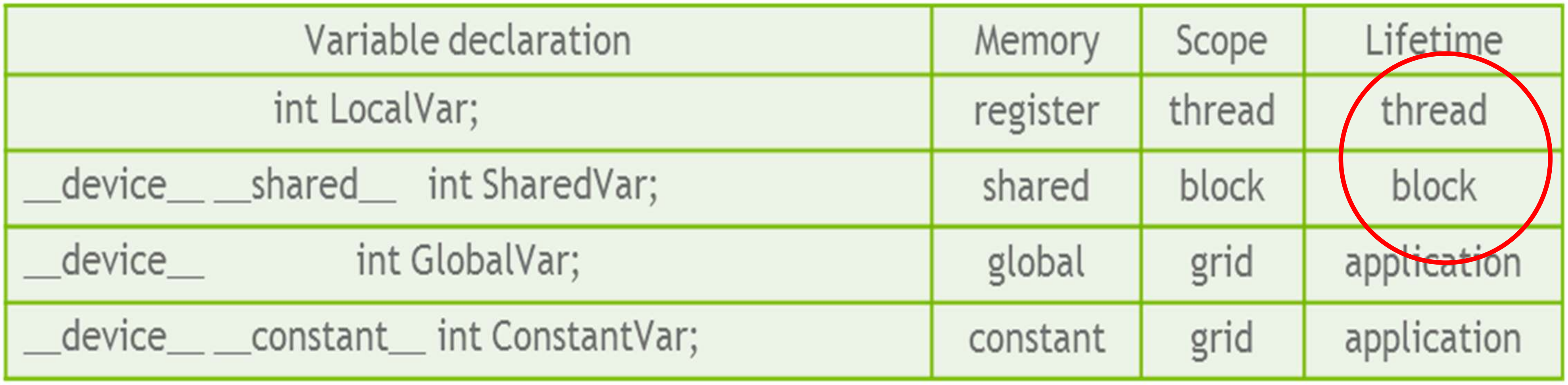


Variable declaration	Memory	Scope	Lifetime
int LocalVar;	register	thread	thread
__device__ __shared__ int SharedVar;	shared	block	block
__device__ int GlobalVar;	global	grid	application
__device__ __constant__ int ConstantVar;	constant	grid	application

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.

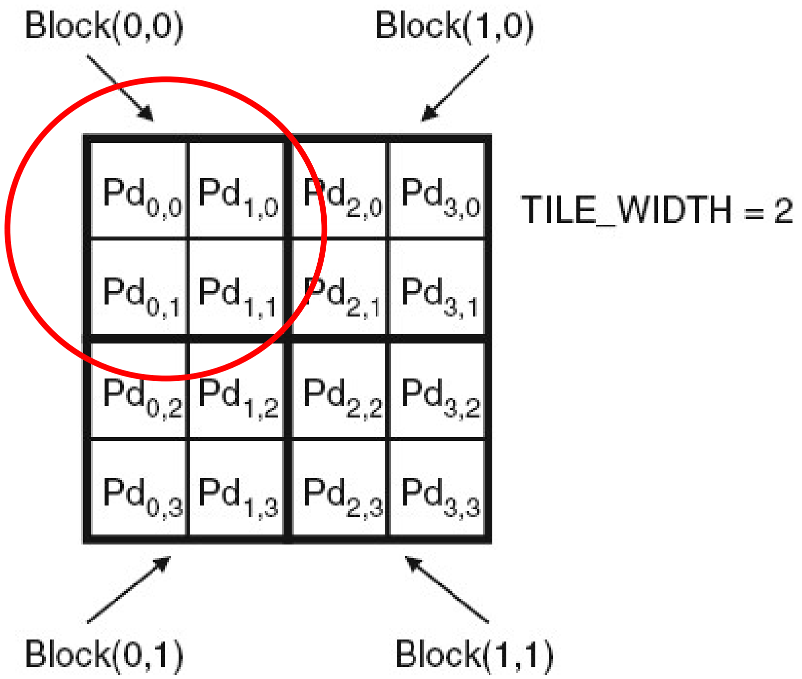
- Declaration of constant variables must be outside any function body.
- Currently total size of constant variables in an application is limited to 64KB.

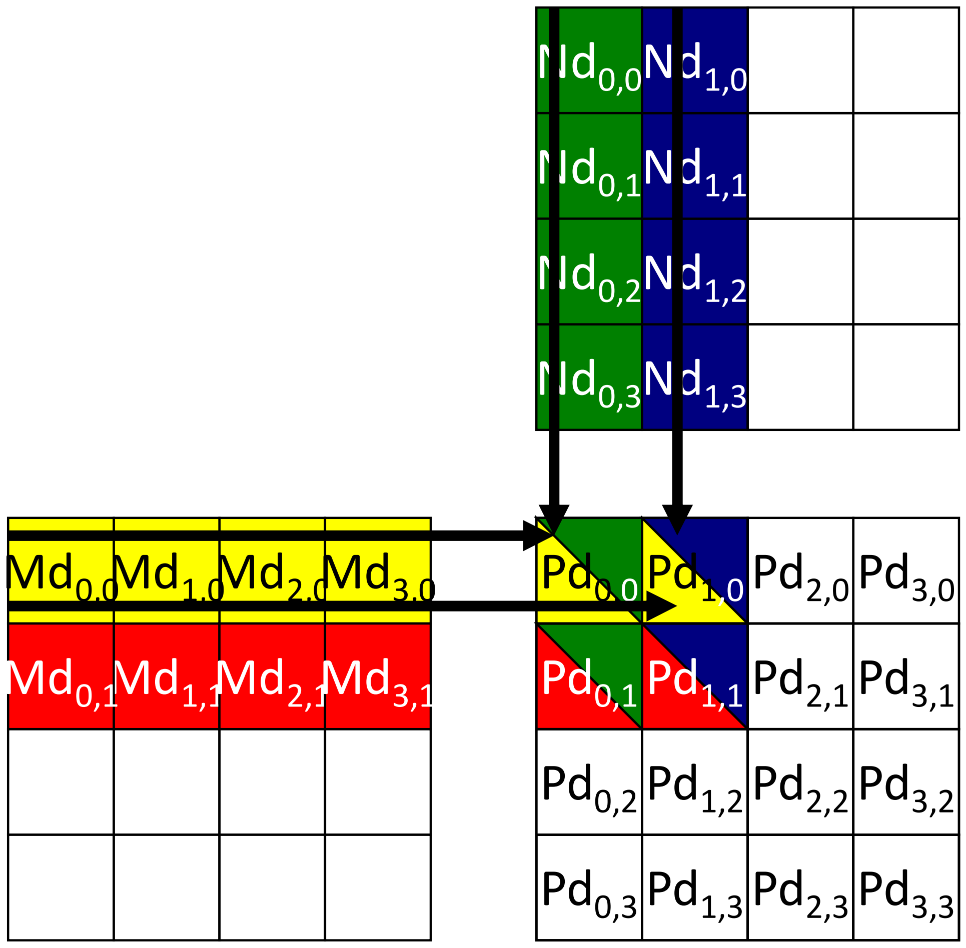
By declaring a CUDA variable in one of the CUDA memory types, a CUDA programmer dictates the visibility and access speed of the variable.

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.

Back to Matrix Multiplication





Back to Matrix Multiplication

Access order

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

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

Time $\longrightarrow$

Back to Matrix Multiplication

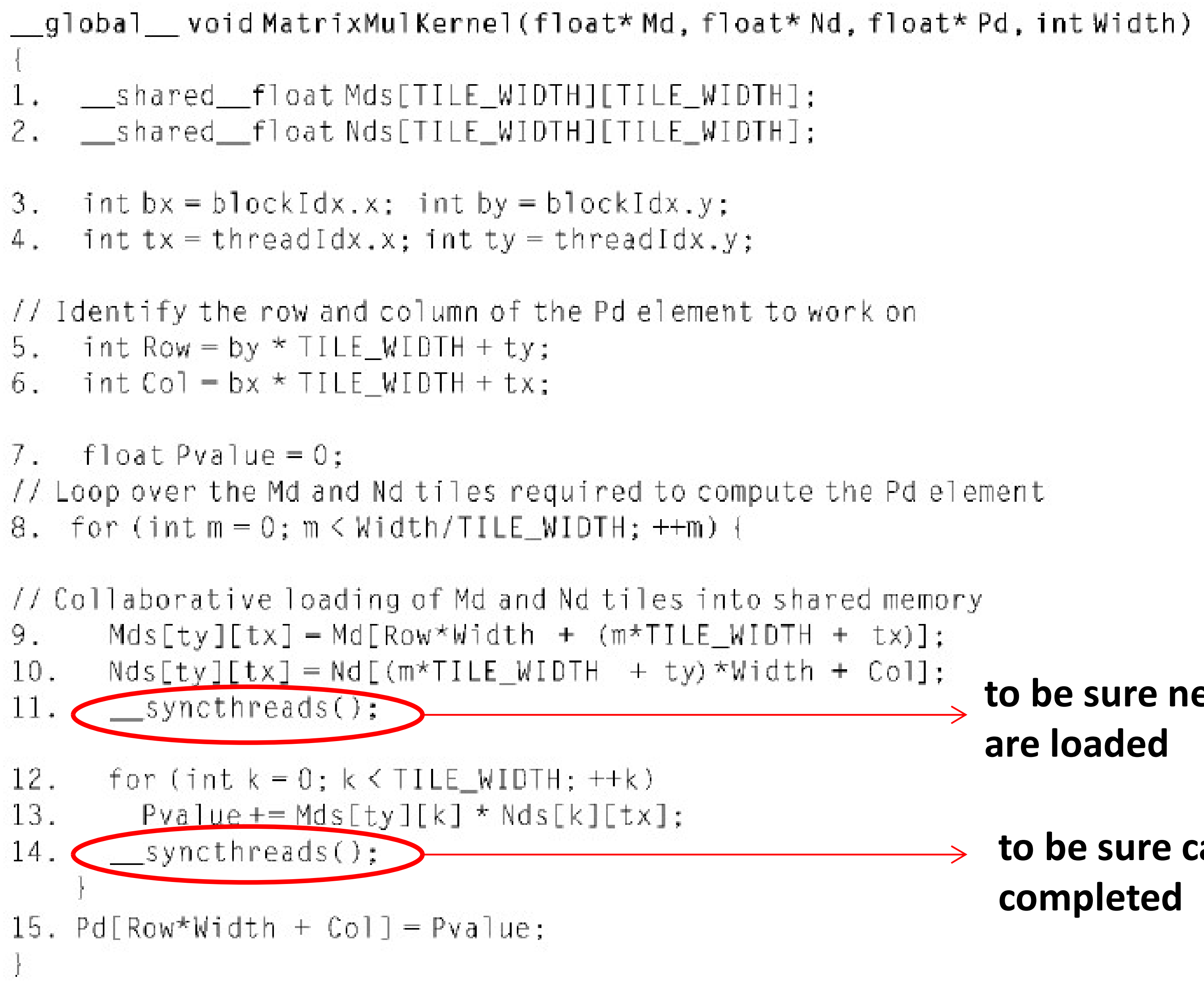
- Potential reduction in global memory traffic in matrix multiplication example is proportional to the dimension of the blocks used.
  - With NxN blocks the potential reduction would be N
- If an input matrix is of dimension M and the tile size is TILE_WIDTH, the dot product will be performed in M/TILE_WIDTH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

Back to Matrix Multiplication

```
__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

How 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 tiles -> 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: **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!)

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

Conclusions

- Using memory effectively will likely require the redesign of the algorithm.
- The ability to reason about hardware limitations when developing an application is a key concept of computational thinking.