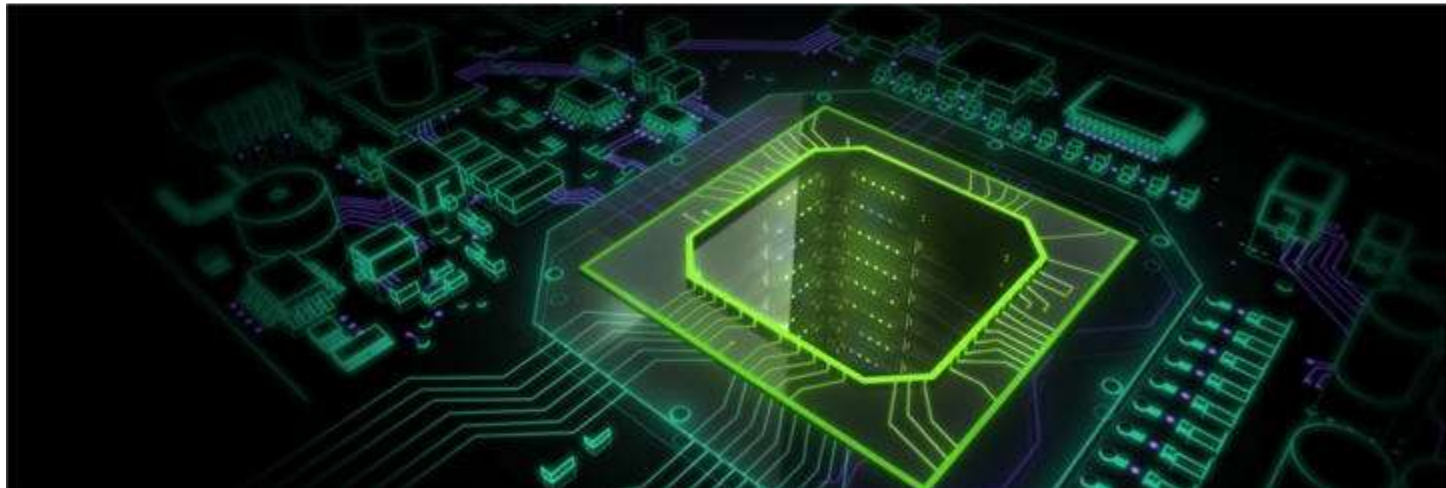
CSCI-GA.3033-012

Graphics Processing Units (GPUs): Architecture and Programming

Lecture 6: CUDA Memories

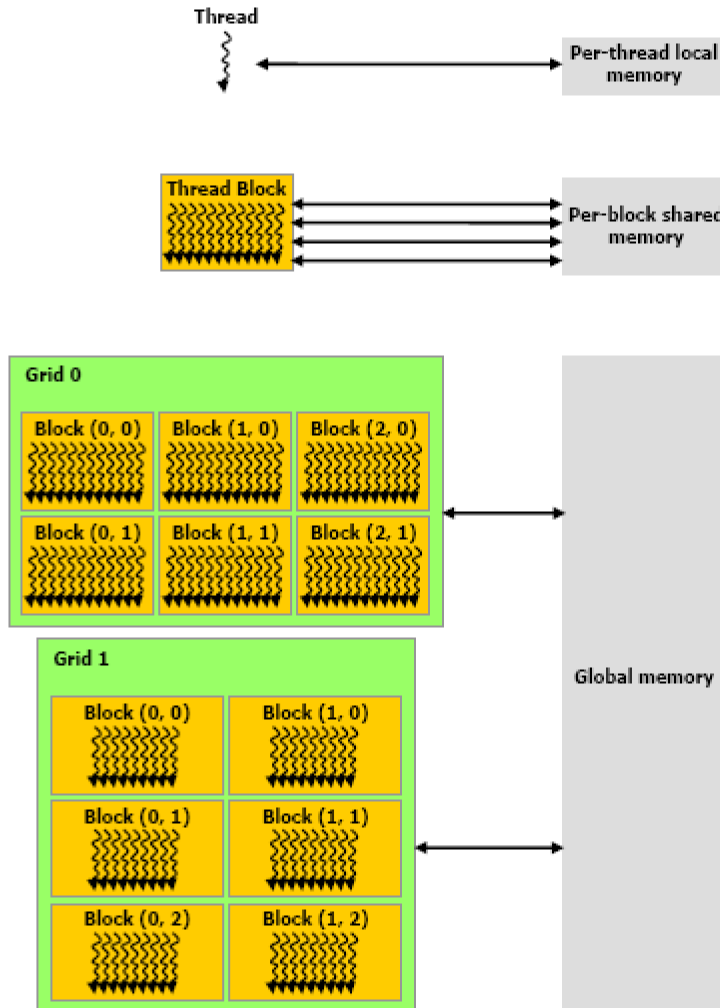
Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com



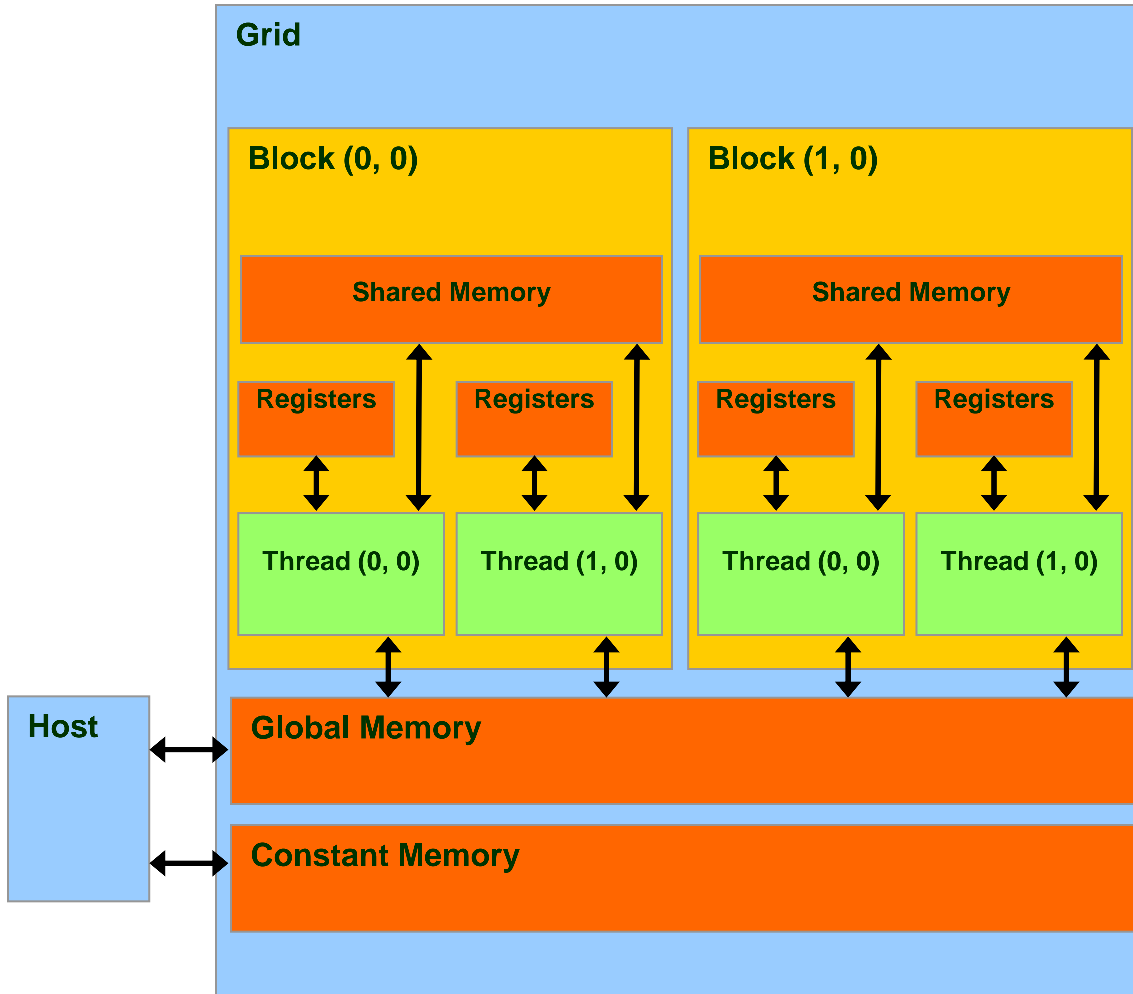
Let's Start With An Example

- 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 367gigaglops!

Main Goals for This Lecture



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


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

Grid

Block (0, 0)

Shared Memory

Registers

Registers

Thread (0, 0)

Thread (1, 0)

Block (1, 0)

Shared Memory

Registers

Registers

Thread (0, 0)

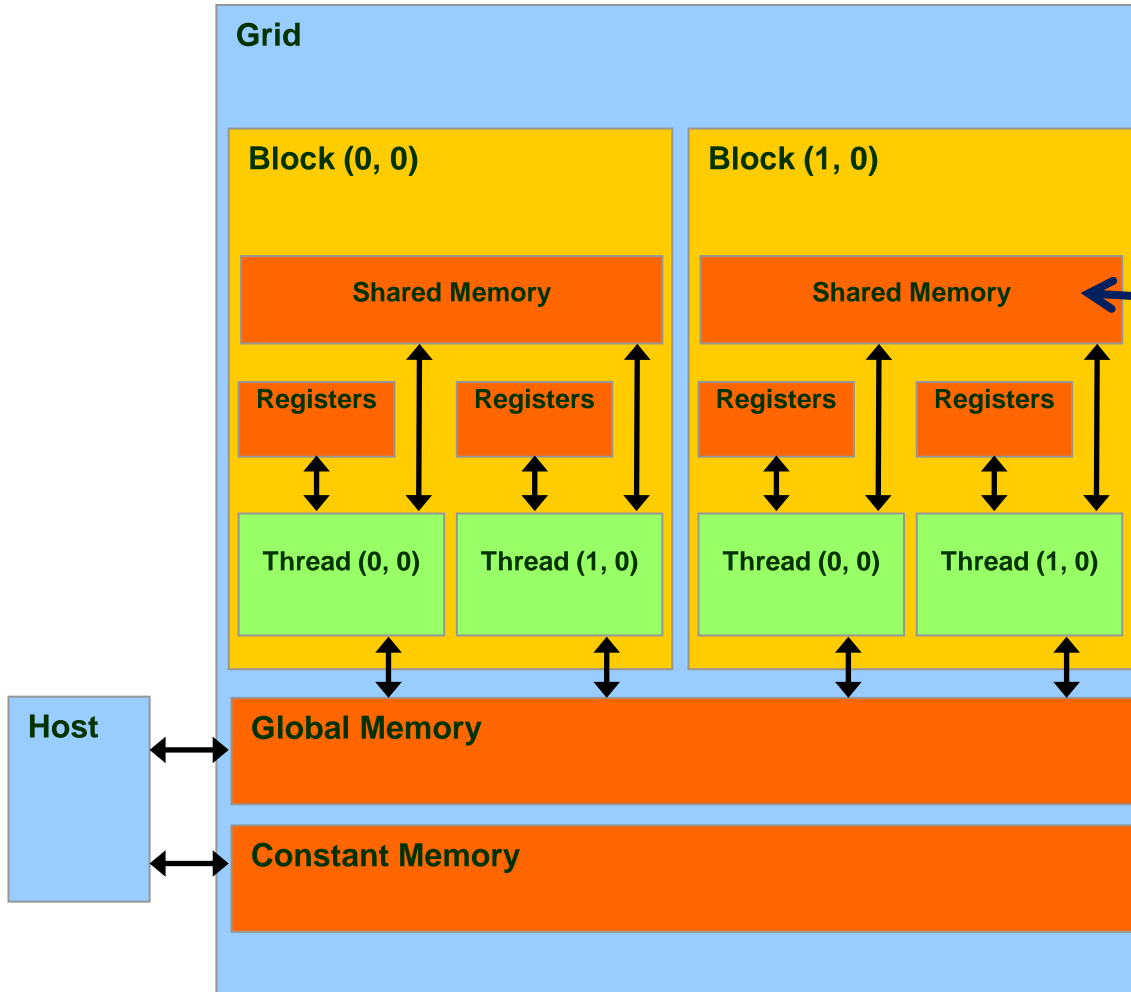
Thread (1, 0)

Host

Global Memory

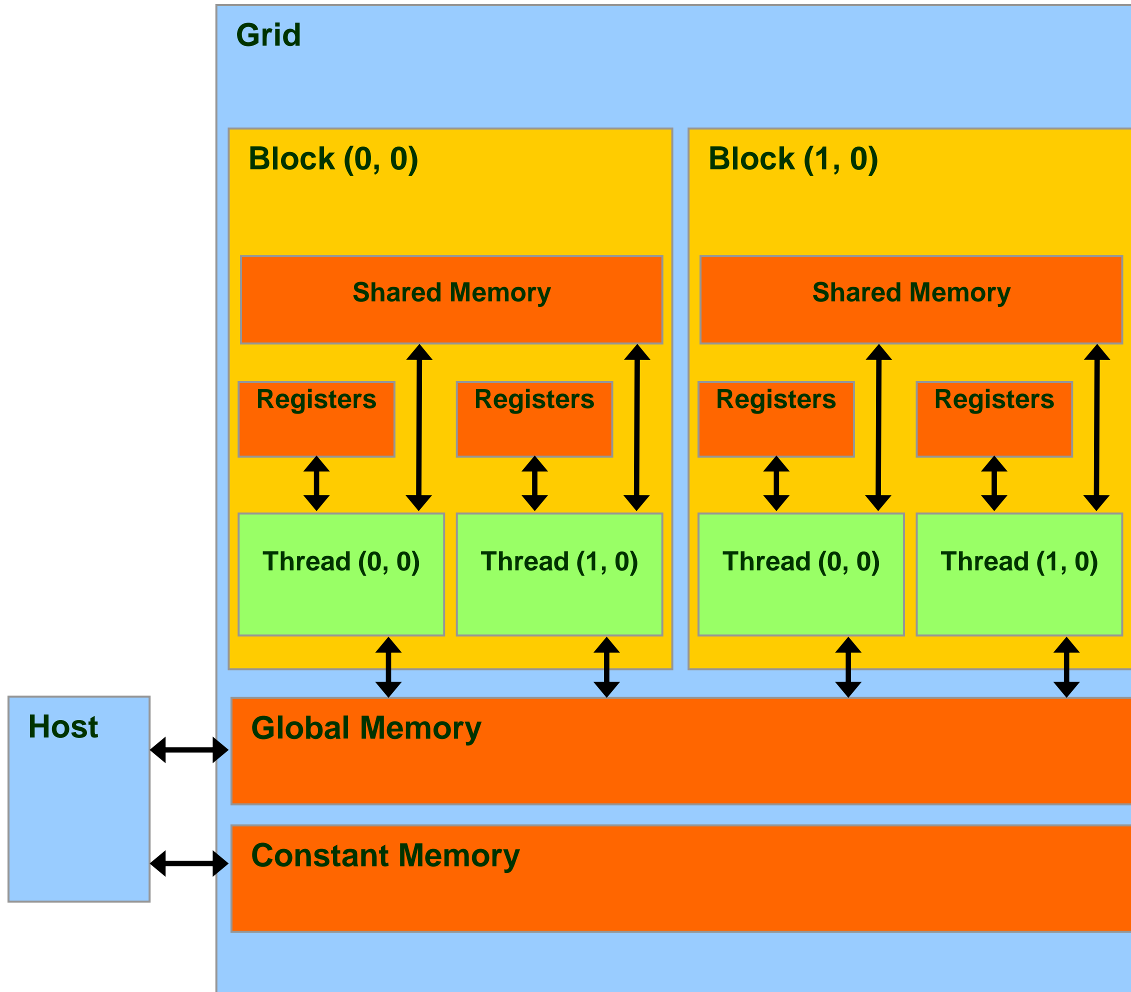
Constant Memory

- Fastest.
- Only accessible by a thread.
- Lifetime of a thread



Shared Memory

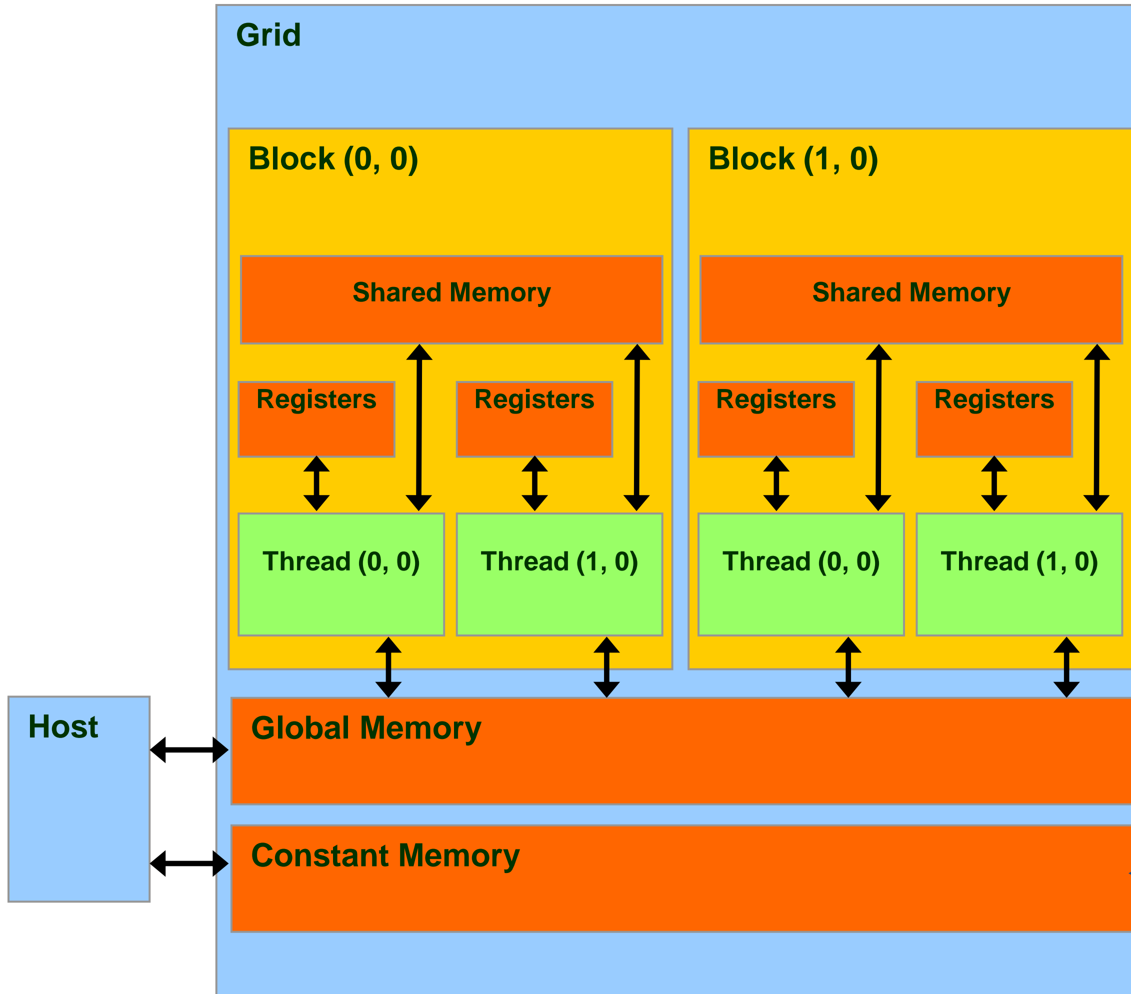
- Extremely fast
- Highly parallel
- Restricted to a block
- Example: Fermi's shared/L1 is 1+TB/s aggregate



Global Memory

- Typically implemented in DRAM
- High access latency: 400-800 cycles
- Finite access bandwidth
- Potential of traffic congestion
- Throughput up to 177GB/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

Variable Declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
`__device__, __shared__, int SharedVar;`	Shared	Block	Kernel
`__device__, int GlobalVar;`	Global	Grid	Application
`__device__, __constant__, int ConstVar;`	Constant	Grid	Application

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
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
`__device__, __shared__, int SharedVar;`	Shared	Block	Kernel
`__device__, int GlobalVar;`	Global	Grid	Application
`__device__, __constant__, int ConstVar;`	Constant	Grid	Application

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

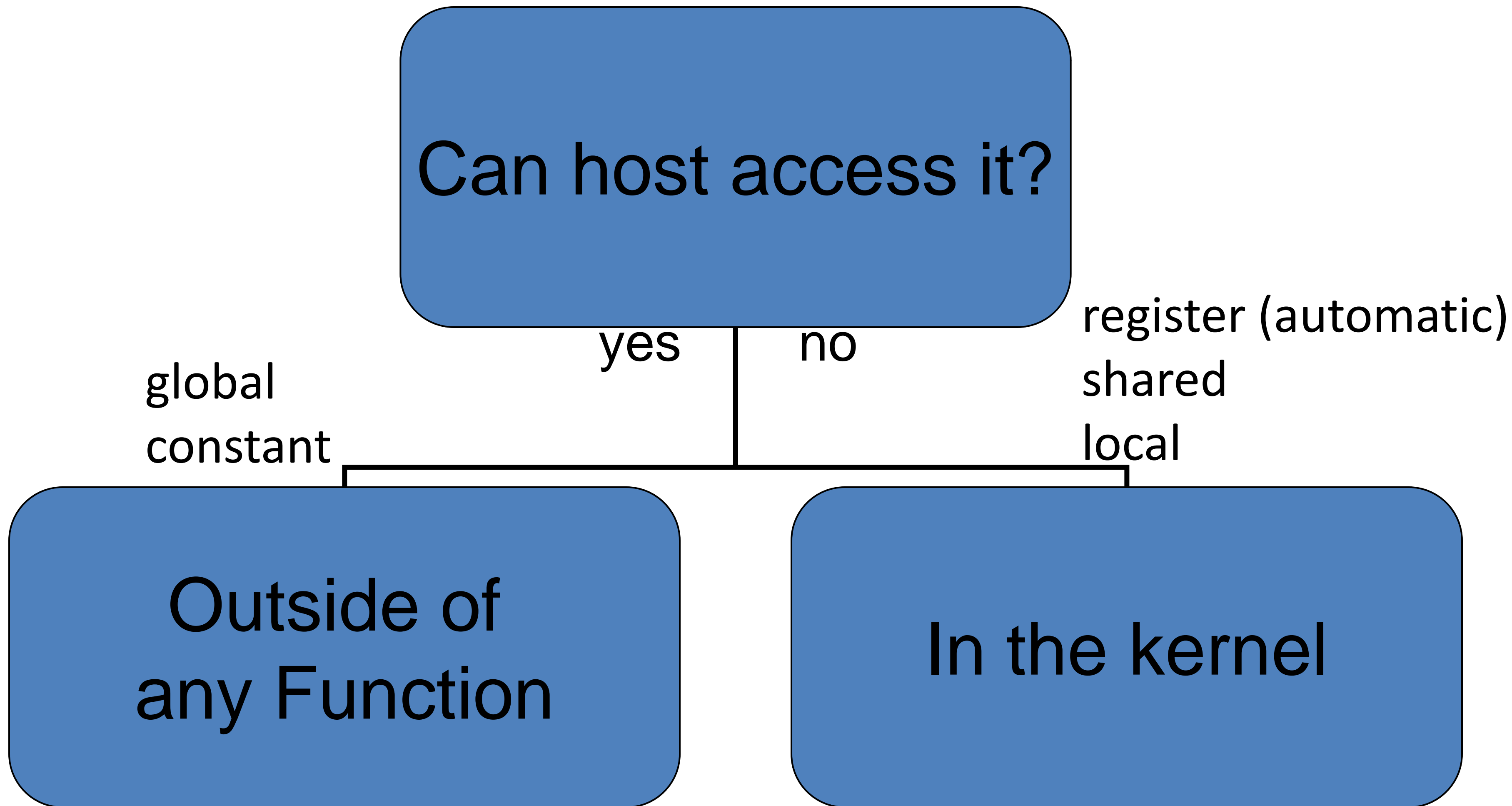
Variable Declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
`__device__, __shared__, int SharedVar;`	Shared	Block	Kernel
`__device__, int GlobalVar;`	Global	Grid	Application
`__device__, __constant__, int ConstVar;`	Constant	Grid	Application

The variable must be declared outside of any function.

Variable Declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
`__device__, __shared__, int SharedVar;`	Shared	Block	Kernel
`__device__, int GlobalVar;`	Global	Grid	Application
`__device__, __constant__, int ConstVar;`	Constant	Grid	Application

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

Where to Declare Variables?

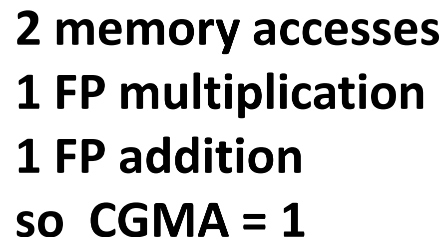


Computation vs Memory Access

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

Computation vs Memory Access

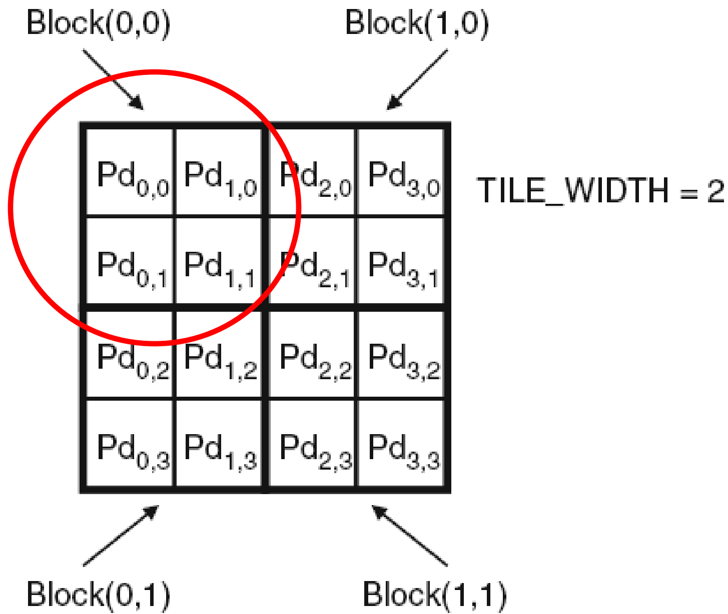
```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
// Calculate the row index of the Pd element and M
int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
// Calculate the column index of Pd and N
int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

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;
}
```

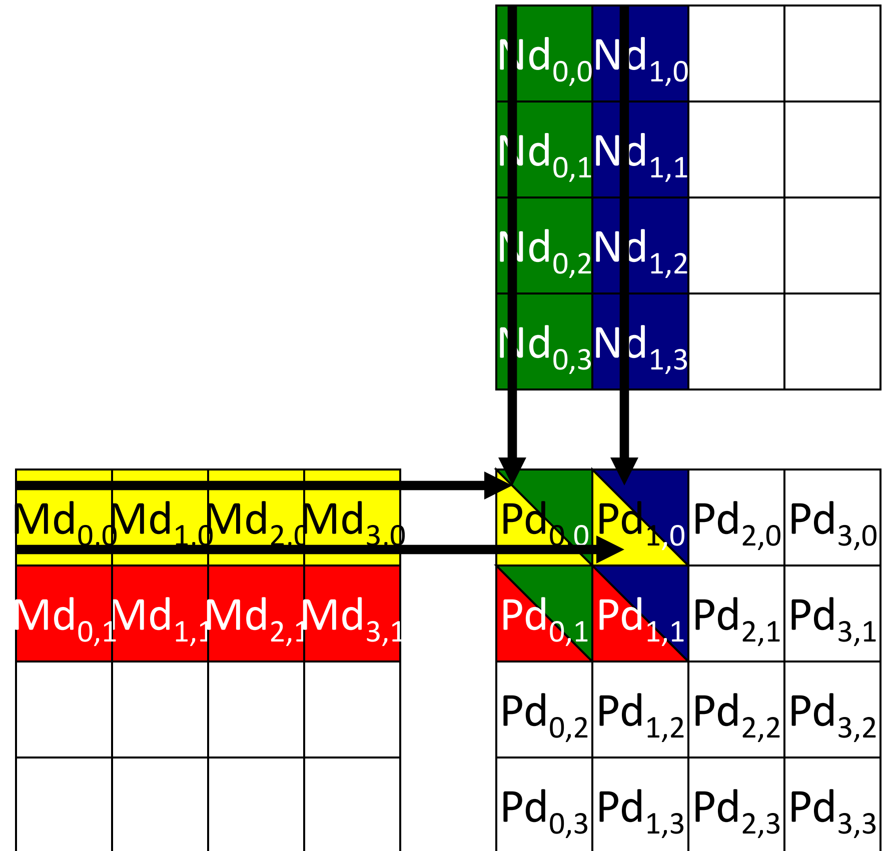


Reducing Global Memory Traffic

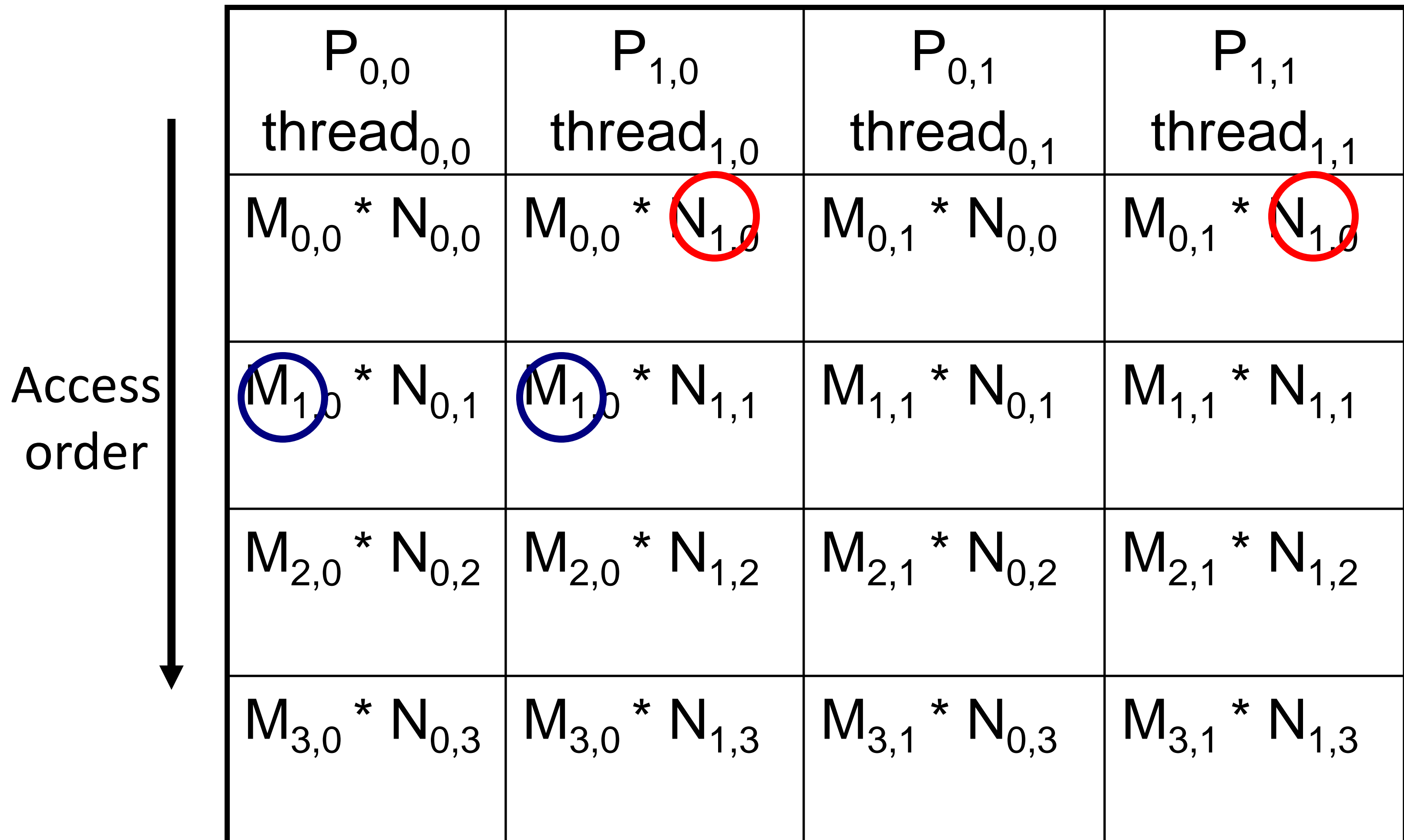
- Global memory access is performance bottleneck.
- The lower CGMA the lower the performance
- Reducing global memory access enhances performance.
- A common strategy is **tiling**: partition the data into subsets called tiles, such that each tile fits into the shared memory.

Back to Matrix Multiplication





Back to Matrix Multiplication



$P_{0,0}$ 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

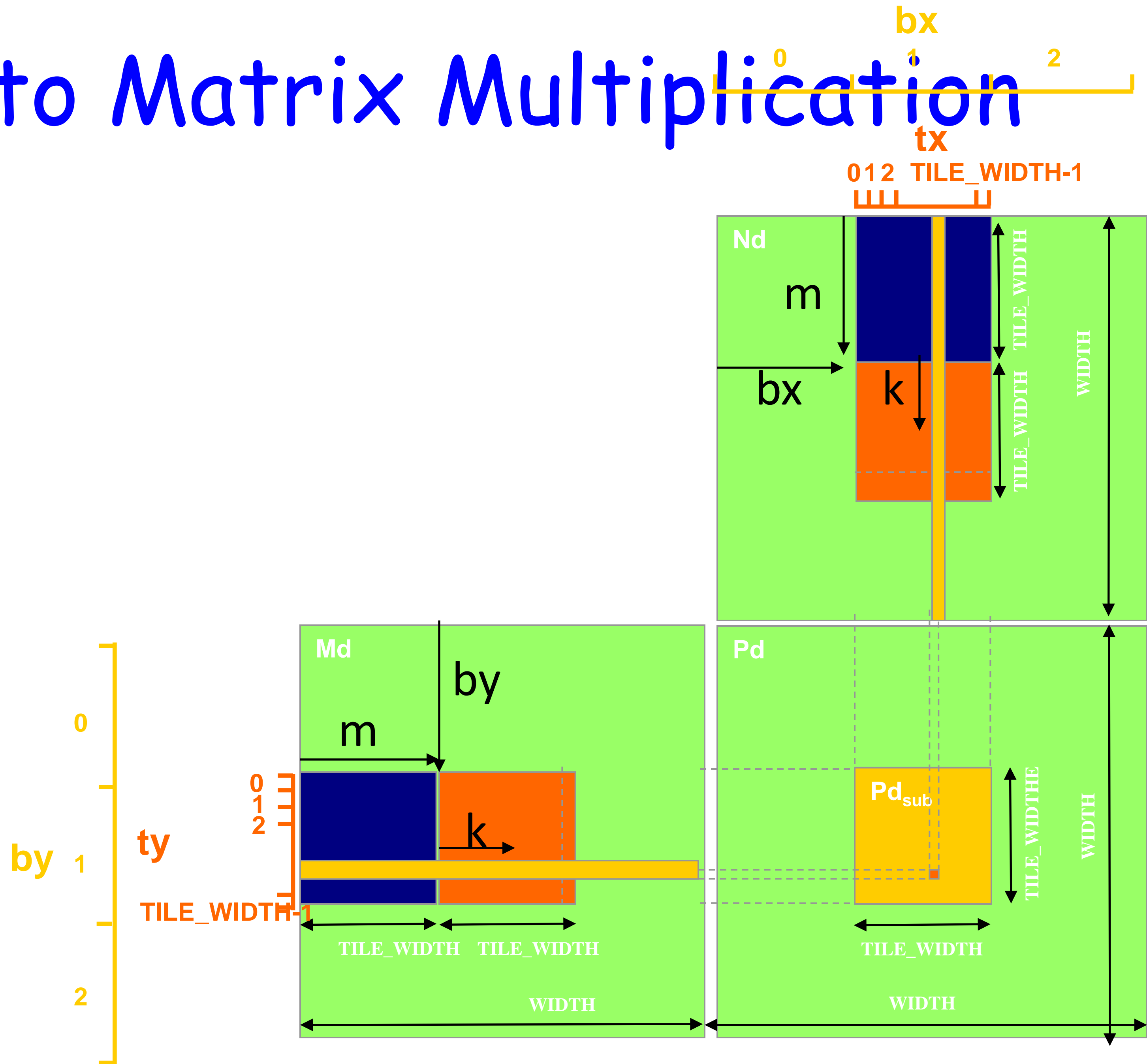
	Phase 1			Phase 2		
$T_{0,0}$	$\mathbf{Md_{0,0}}$ ↓ $Mds_{0,0}$	$\mathbf{Nd_{0,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}*Nds_{0,0}$ + $Mds_{1,0}*Nds_{0,1}$	$\mathbf{Md_{2,0}}$ ↓ $Mds_{0,0}$	$\mathbf{Nd_{0,2}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}*Nds_{0,0}$ + $Mds_{1,0}*Nds_{0,1}$
$T_{1,0}$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

Time

Back to Matrix Multiplication

- 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)
{
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[(m*TILE_WIDTH + ty)*Width + Col];
11.    __syncthreads();

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

Back to Matrix Multiplication

```
__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[(m*TILE_WIDTH + ty)*Width + Col];
11.    __syncthreads();

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

to be sure needed elements are loaded (line 11)

to be sure calculations are completed (line 14)

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 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 CUDA memory limits the number of threads that can execute simultaneously in SM for a given application
  - The more memory location each thread requires, the fewer the number of threads per SM

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

GPU *Compute Capability*

- A way to express hardware resources in a standardized form
- Starts with *compute 1.0*
- Higher level compute capability defines a superset of features of those at lower levels

GPU *Compute Capability*

Compute 1.0

Features	Compute 1.0
Number of stream processors per SM	8
Maximum number of threads per block	512
Maximum grid dimension (*x*, *y*)	65,535, 65,535
Maximum block dimension (*x*, *y*, *z*)	512, 512, 64
Threads in a warp	32
Registers per SM	8192 (8 K)
Shared memory per SM	16,384 (16 K)
Banks in shared memory	16
Total constant memory	65,536 (64 K)
Cache working set for constants per SM	8192 (8 K)
Local memory per thread	16,384 (16 K)
Cache working set for texture per SM	6 to 8 kB
Maximum number of active blocks per SM	8
Maximum active warps per SM	24
Maximum active threads per SM	768
1D texture bound to CUDA array—maximum width	2^{13}
1D texture bound to linear memory–maximum width	2^{27}
2D texture bound to linear memory or CUDA array—maximum dimension (*x*, *y*)	2^{16}, 2^{15}
3D texture bound to a CUDA array—maximum dimension (*x*, *y*, *z*)	2^{11}, 2^{11}, 2^{11}
Maximum kernel size	2 million microcode instructions

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.
- We are done with chp 6.