CPSC/ECE 4780/6780

General-Purpose Computation on Graphical Processing Units (GPGPU)

Lecture 5: CUDA Memories

Recaps of Last Lecture

- CUDA threads are organized as a **two-level hierarchy**
- Subdivision of a grid is flexible but with hardware limitations
- Threads are assigned to execution resources on a block-by-block basis
  - Blocks can be executed in any order relative to each other to enable **transparent scalability**
- Within a block, threads have mechanism to communicate and synchronize
  - **Barrier synchronization** ensures all threads in a block have completed a phase of their execution of the kernel before any moves to the next phase
- **Warp** is the unit of thread scheduling in SMs
  - Latency hiding
  - Zero-overhead thread scheduling
  - Warp divergence

Memory Access Efficiency

- **Compute to global memory access (CGMA) ratio**: the number of floating-point calculations performed for each access to the global memory within a region of a CUDA program

```
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];
```

2 global memory access
1 floating-point multiplication
1 floating-point addition
CGMA ratio = 1

- CGMA ratio influence the performance of a CUDA kernel
  - G80 supports 86.4 GB/s of global memory access bandwidth
  - The size of one single-precision global memory datum is 4 bytes
  - We can load no more than 86.4/4 = 21.6 GB single-precision data per second
  - With CGMA ratio = 1, that's only 21.6 gigaflops, a very tiny fraction comparing to the peak performance of G80 (367 gigaflops!)
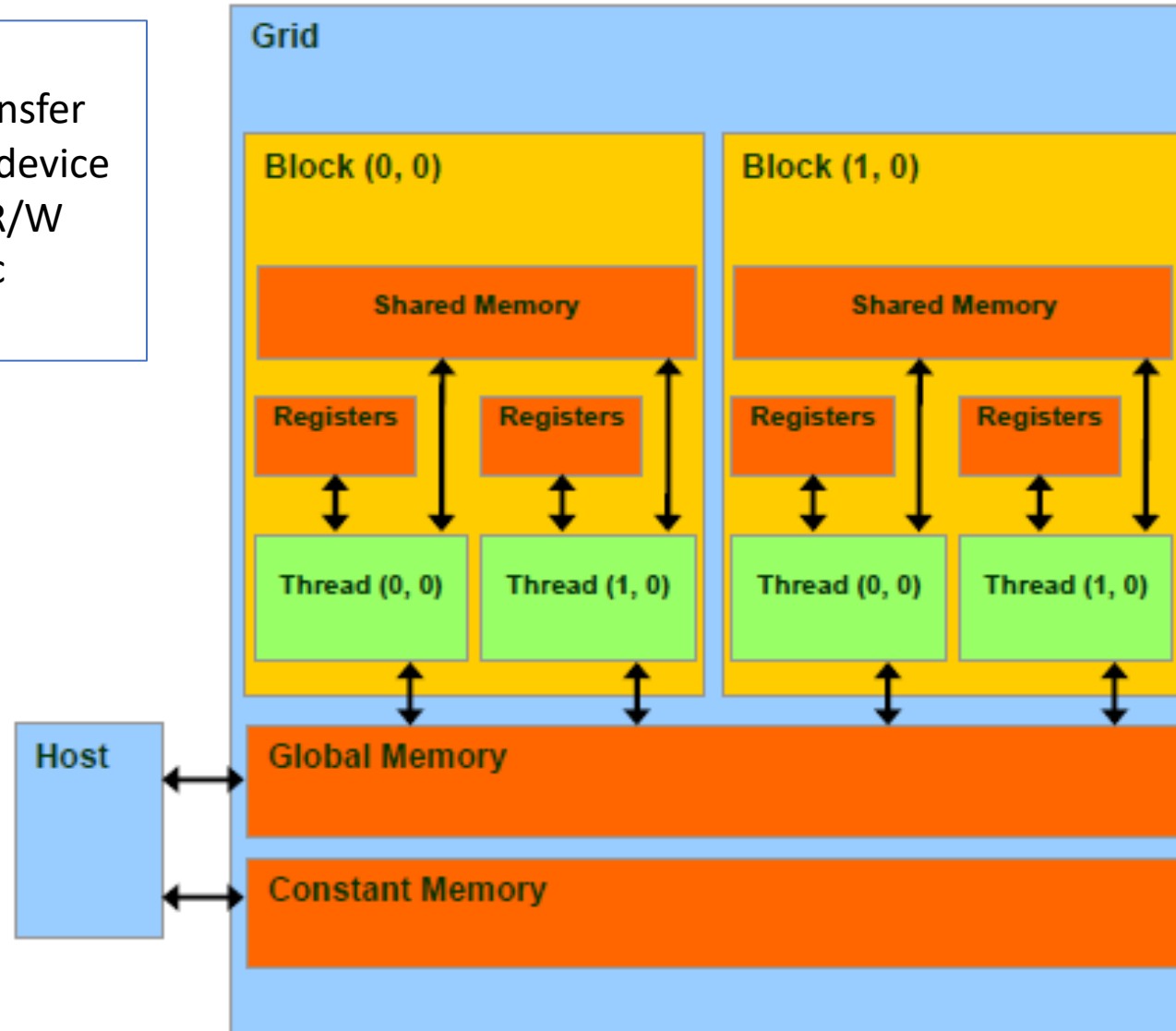- Improve CGMA ratio for higher level of performance for the kernel

CUDA Device Memory Types

Global Memory
- Host code can transfer data to/from the device
- Device code can R/W
- Potential of traffic congestion

Constant Memory
- Host code can transfer data to/from the device
- Device code can only R
- Short latency, high bandwidth when all threads access the same location

Shared memory
- On-chip memory
- Restricted to a block
- Extremely fast
- Highly parallel

Register
- On-chip memory
- Restricted to a thread
- Fastest

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

4

Scope and Lifetime of CUDA Variables

- **Scope**: the range of threads that can access the variable
  - By a single thread only
  - By all threads of a block
  - By all threads of all grids
- **Lifetime**: the portion of the program's execution duration when the variable is available for use
  - Within a kernel's invocation
  - Throughout the entire application

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

CUDA Variable Type Qualifiers

Reducing Global Memory Traffic

- Global memory access is performance bottleneck

- The lower CGMA ratio the lower the performance

- Reducing global memory access enhances performance

- A common strategy is "tiling" – partition the data into subsets (tiles) such that each tile fits into the shared memory
  - Criterion: the kernel computations on these tiles can be done independently of each other

Shared Memory

- On-chip memory

- Extremely fast

- Highly parallel

- Restricted to a block (Block level computation)
  - Allows data to be shared between threads in the same block
  - User configurable cache at the thread block level
  - Still no broader synchronization beyond the level of thread blocks

- Declared by __shared__ keyword

Back to Matrix Multiplication

- Breaks Pd into four 2x2 blocks
- Each block calculates one tile
  - Block size equals tile size
  - Each thread in a block calculates one element
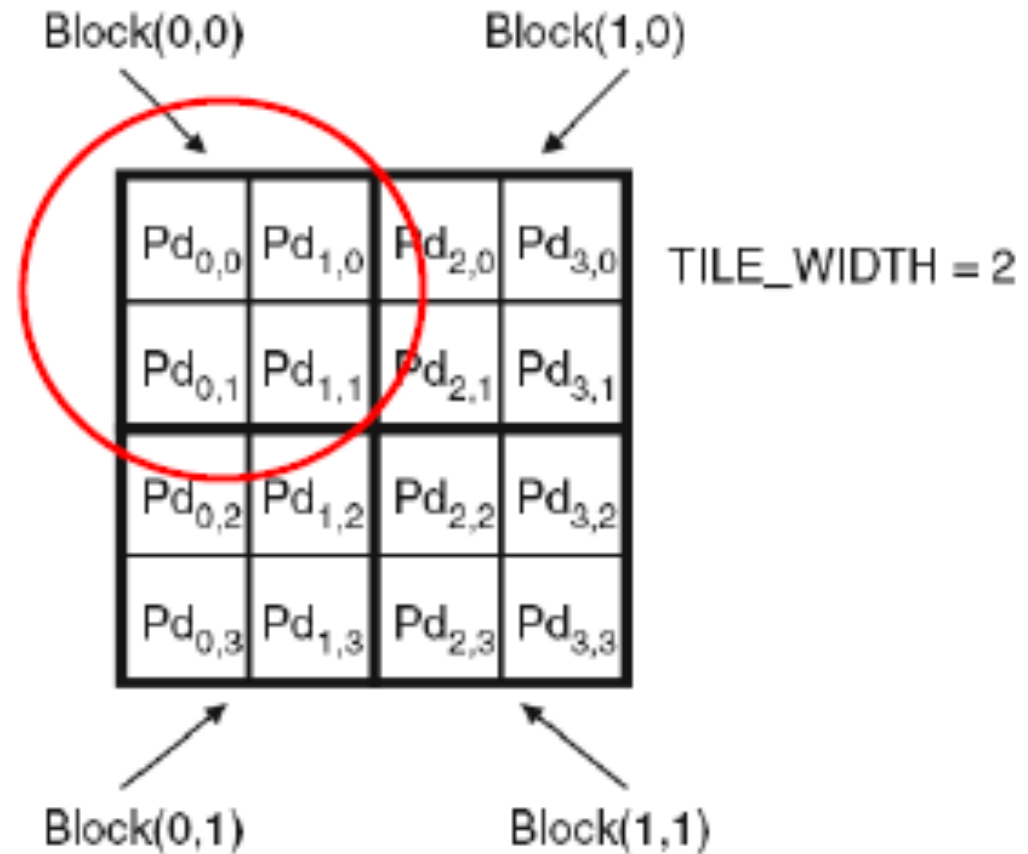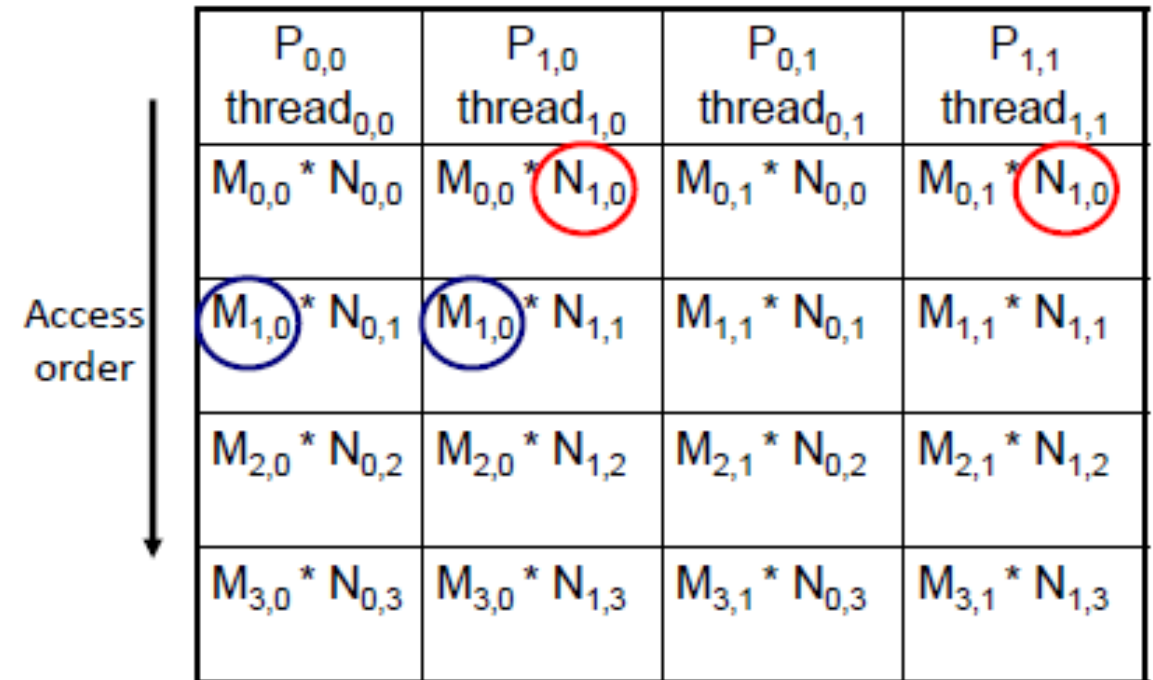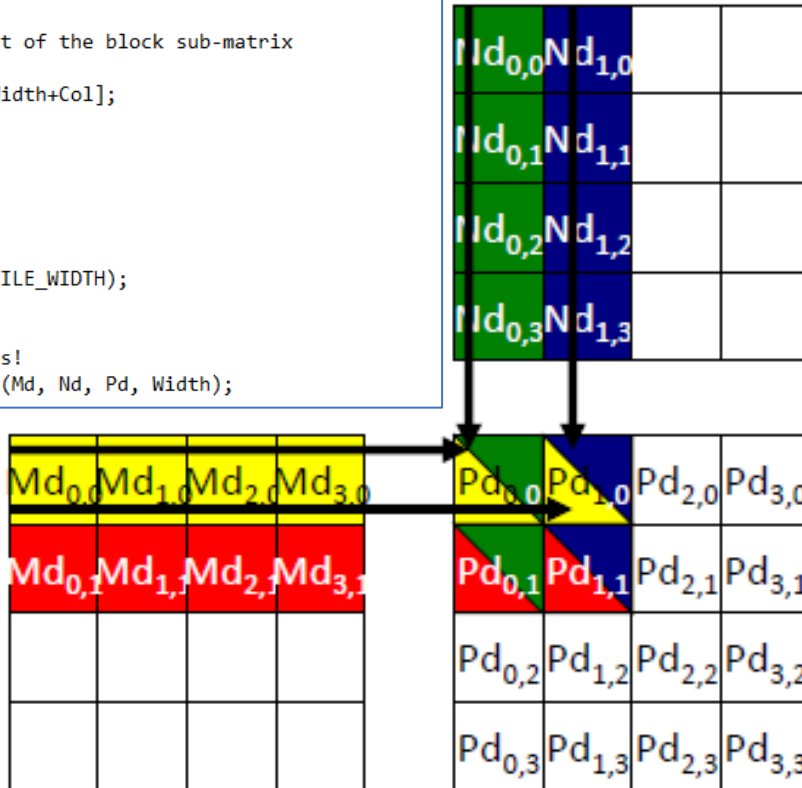
Global Memory Accesses in Block(0,0)

```
__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;
}


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

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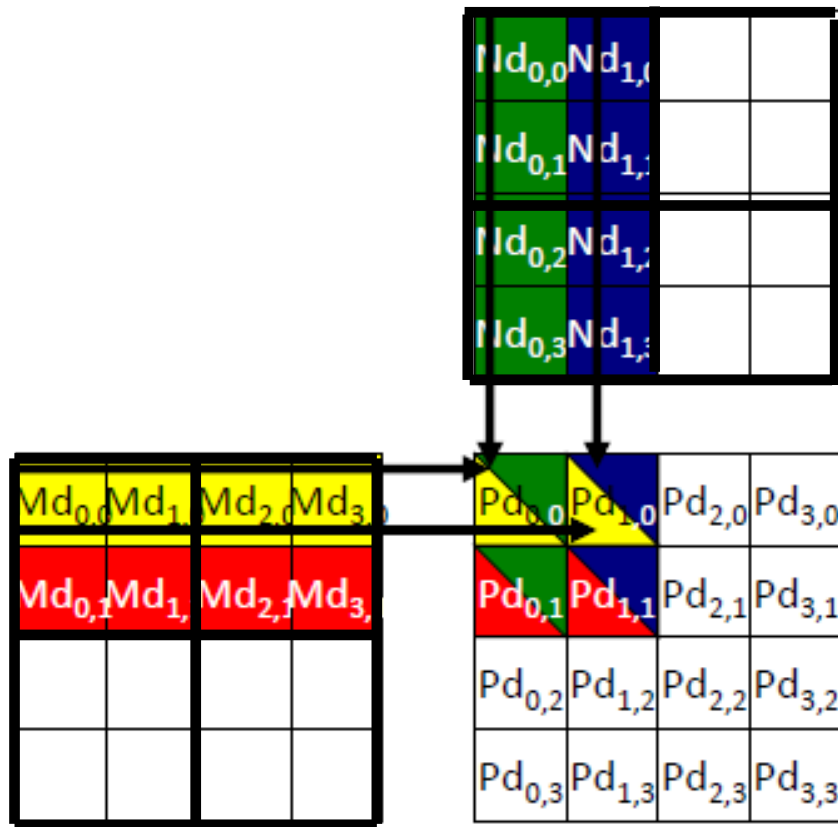

	$P_{0,0}$ thread$_{0,0}$	$P_{1,0}$ thread$_{1,0}$	$P_{0,1}$ thread$_{0,1}$	$P_{1,1}$ thread$_{1,1}$
	$M_{0,0} * N_{0,0}$	$M_{0,0} * N_{1,0}$	$M_{0,1} * N_{0,0}$	$M_{0,1} * N_{1,0}$
Access order	$M_{1,0} * N_{0,1}$	$M_{1,0} * N_{1,1}$	$M_{1,1} * N_{0,1}$	$M_{1,1} * N_{1,1}$
	$M_{2,0} * N_{0,2}$	$M_{2,0}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

The Basic Idea is to Reduce the Traffic

- Make threads that use common elements collaborate
- Each thread loads different elements into the shared memory before calculations
  - Not to exceed the capacity of shared memory
  - Dividing data into smaller tiles
- These elements will be used by the thread that loaded them and other threads that share them within a block

Tiling Md and Nd in Matrix Multiplication

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

Time →

The four threads of block (0,0) collaboratively load a tile of Md and Nd into shared memory

The calculation of each dot product is now performed in two phases. Products of two pairs of the input matrix elements are accumulated

11

Reduction in Matrix Multiplication

- The reduction of global memory access is by a factor of N if the tiles are NxN elements

- If an input matrix is of dimension N and the tile size is TILE_WIDTH, the dot product would be performed in N/TILE_WIDTH phases

- Reusing shared memory variables Mds and Nds allows a much smaller shared memory to serve most of the accesses to global memory. Such focused access behavior is called "locality"

Tiled Kernel Function in 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;

// Determine the row and column index of the Pd element
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[tx.y][tx.x] = Md[Row*Width + (m*TILE_WIDTH + tx.x)];
10.         Nds[tx.y][tx.x] = Nd[(m*TILE_WIDTH + tx.y)*Width + Col];
11.         __syncthreads();

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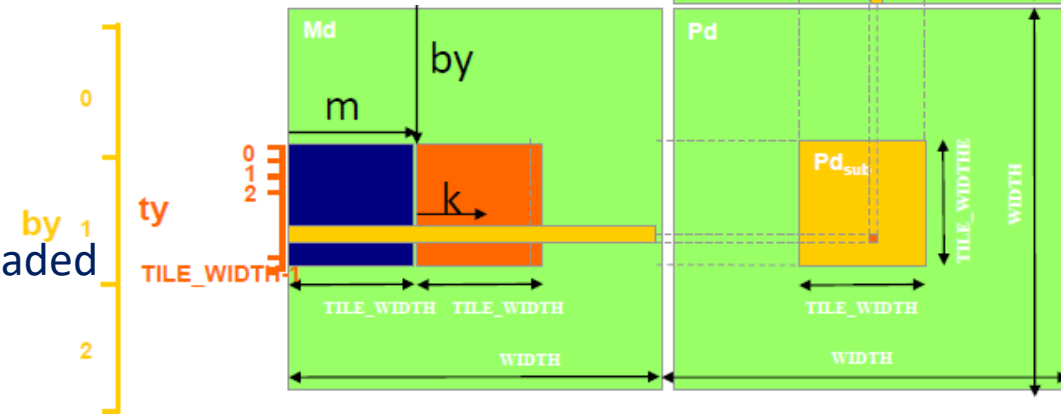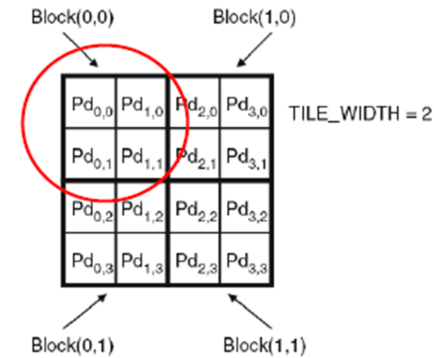
Stored in shared memory

Stored in registers

E.g., thread(1,0) of block(0,1) calculates
Pd(0*2+1,1*2+0) = Pd(1,2)

Make sure needed elements are loaded

Make sure calculations are completed

13

Benefit of Titled Algorithm in Matrix Multiplication on G80

- Before
  - G80 supports 86.4 GB/s of global memory access bandwidth
  - The size of one single-precision global memory datum is 4 bytes
  - We can load no more than 86.4/4 = 21.6 GB single-precision data per second
  - With CGMA ratio = 1, that's only 21.6 gigaflops, a very tiny fraction comparing to the peak performance of G80 (367 gigaflops!)
- After
  - The global memory access are reduced by a factor of TILE_WIDTH
  - If use 16x16 tiles => reduction in the global memory accesses by a factor of 16
  - Global memory can now support (86.4/4)x16= 345.6 gigaflops => very close to the peak!

Shared Memory Banks

- Shared memory is partitioned into **banks**. Any load/store of *n* addresses that spans *n* distinct banks is executed entirely in parallel

- Multiple thread accesses to the same bank (**bank conflict**) are serialized
    - Cost = max # of simultaneous accesses to a single bank
    - Exception: if all threads in a warp access 1 location

- There are typically 32 banks of memory and successive words are assigned to successive banks. If each thread in a warp accesses a successive word in memory, it's all done in parallel

- The bandwidth of shared memory is 32 bits per bank per clock cycle

- 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

Coalesced Memory in CUDA

- Shared memory can be used to avoid **un-coalesced memory** accesses by loading and storing data in a **coalesced pattern** from global memory and then reordering it in shared memory

- A **coalesced memory transaction** is one in which all of the threads in a half-warp access global memory at the same time. The correct way to do it is just having consecutive threads access consecutive memory addresses

  - We could access row after row: 0 1 2 3 4 5 6 7 8 9 10 11
    (row, col) maps to memory (row*4 + col)

  - Suppose we need to access element once with four threads, which threads will be used for which element?

0	1	2	3
4	5	6	7
8	9	10	11

Thread 0	0	1	2
Thread 1	3	4	5
Thread 2	6	7	8
Thread 3	9	10	11

Or

| Thread 0	0	4	8
Thread 1	1	5	9
Thread 2	2	6	10
Thread 3	3	7	11

Memory Bank Conflict

- A bank conflict arises if any of the threads in a half warp access different words (32-bit) in the same bank. When a bank conflict happens, the access to the data is serialized
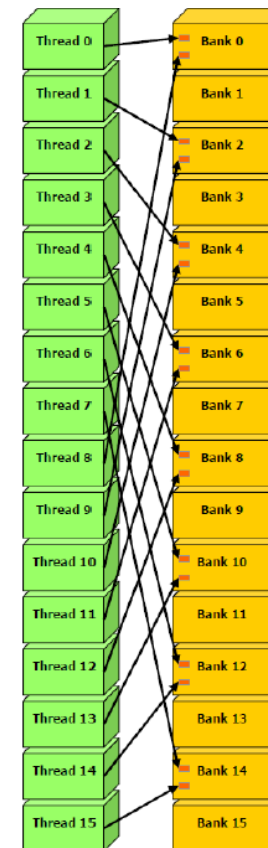
No Bank Conflict Cases

Bank Conflict Cases
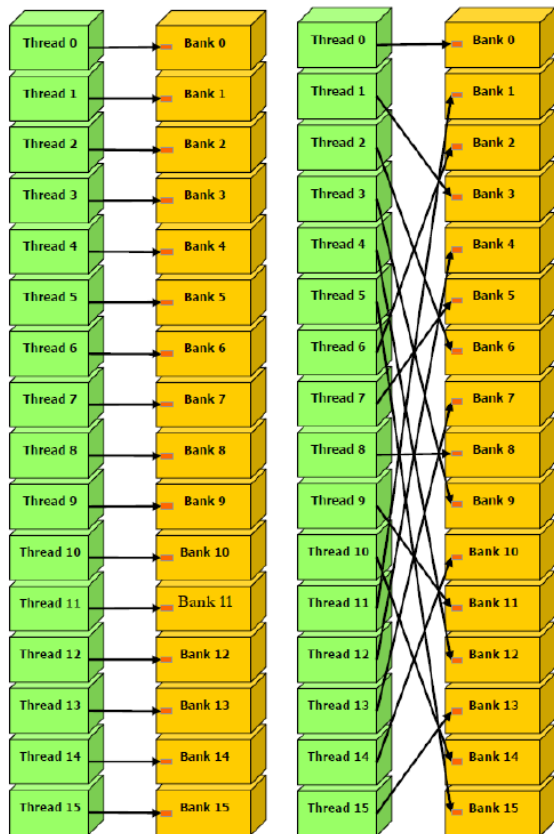
a) Liner addressing
__shared__ float shared[32];
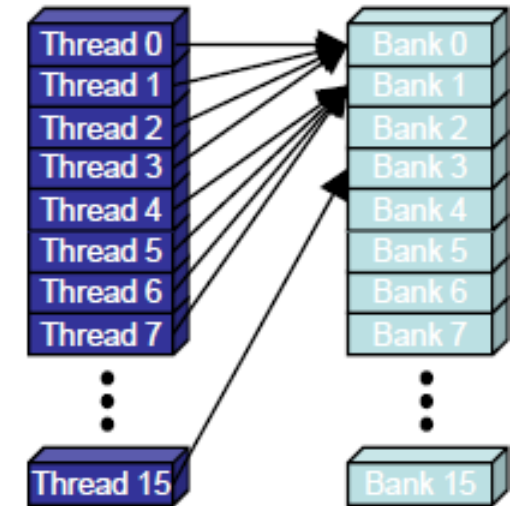float data = shared[BaseIndex + S*tid]; with S=1 and S=3
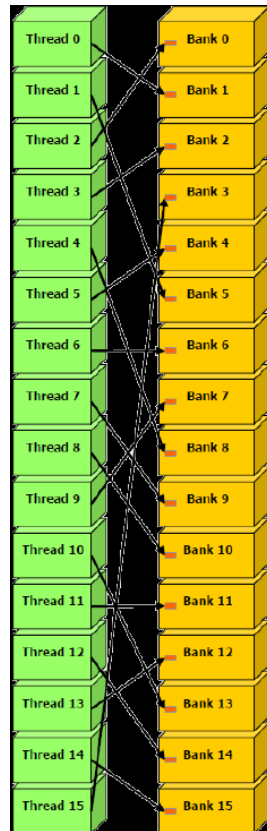
b) Random addressing

c) 2-way bank conflict
__shared__ double shared[32];
double data = shared[BaseIndex + S*tid]; with S=1

d) 4-way bank conflict
__shared__ char shared[32];
char data = shared[BaseIndex + S*tid]; with S=1

Identifying and Avoiding Bank Conflicts

Primitive Data Types

Keyword	Description	Size/Format
(integers)		
byte	Byte-length integer	8-bit two's complement
short	Short integer	16-bit two's complement
int	Integer	32-bit two's complement
long	Long integer	64-bit two's complement
(real numbers)		
float	Single-precision floating point	32-bit IEEE 754
double	Double-precision floating point	64-bit IEEE 754
(other types)		
char	A single character	16-bit Unicode character
boolean	A boolean value (true or false)	true or false

- What if a thread loads 2 consecutive array elements?
  int tid = threadIdx.x;
  shared[2*tid] = global[2*tid];
  shared[2*tid+1] = global[2*tid+1];

- To avoid conflicts
  shared[tid] = global[tid];
  shared[tid+blockDim.x] = global[tid+blockDim.x];

- Consider
  __shared__ float shared[256];
  float data = shared[BaseIndex + S * tid];

- If S has no common factors with the number of banks (16), there are no conflicts (S is odd)

- Memory padding

Constant Memory

- Constant memory is memory used for data that will not change over the course of a kernel execution (read-only) to reduce the required memory bandwidth

- A total of 64KB constant memory and 8KB cache on a device

- Read data through multiprocessor constant cache

- Declared by __constant__ keyword
  - E.g., __constant__ float cst_ptr [size];

- Use cudaMemcpyToSymbol to copy the data from host to the constant memory
  - E.g., cudaMemcpyToSymbol (cst_ptr, host_ptr, data_size);

Performance of Constant Memory

- Benefits
  - Reading from the 64KB of constant memory can save bandwidth over standard reads of global memory
    - **Receives data in a half-warp broadcast**: NVIDIA hardware can broadcast a single memory read to each half-warp -> 1/16 (roughly 6 percent) of memory traffic saving
    - **Retrieves data from the constant memory cache**: No additional memory traffic for consecutive reads of the same address because the constant memory is cached
- Beware that
  - Optimized when warp of threads read same location
  - Serialized when warp of threads read in different locations
  - Very slow when cache miss (read data from global memory)

Constant Memory Broadcast

- When values are broadcast to threads in a half warp (groups of 16 threads), which of the following is good use of constant memory?

```
__constant__ int my_const[16];

__global__ void constant_test() {
int i = blockIdx.x;

int value = my_const[i % 16];
}
```

```
__constant__ int my_const[16];

__global__ void constant_test() {
int i = blockIdx.x * blockDim.x + threadIdx.x;

int value = my_const[i % 16];
}
```

Constant Memory Example

```c
//declare constant memory
__constant__ float cangle[360];

__global__ void test_kernel(float* darray)
{
    int index;

    //calculate each thread global index
    index = blockIdx.x * blockDim.x + threadIdx.x;

    for(int loop=0;loop<360;loop++)
        darray[index]= darray [index]+ cangle[loop];
    return;
}
```

```c
int main(int argc,char** argv)
{
    int size=3200;
    float* darray;
    float hangle[360];

    //allocate device memory
    cudaMalloc((void**)&darray,sizeof(float)*size);

    //initialize allocated memory
    cudaMemset(darray,0,sizeof(float)*size);

    //initialize angle array on host
    for(int loop=0;loop<360;loop++)
        hangle[loop] = acos( -1.0f )* loop/ 180.0f;

    //copy host angle data to constant memory
    cudaMemcpyToSymbol(cangle, hangle, sizeof(float)*360);

    test_kernel<<<size/64,64>>>(darray);

    //free device memory
    cudaFree(darray);
    return 0;
}
```
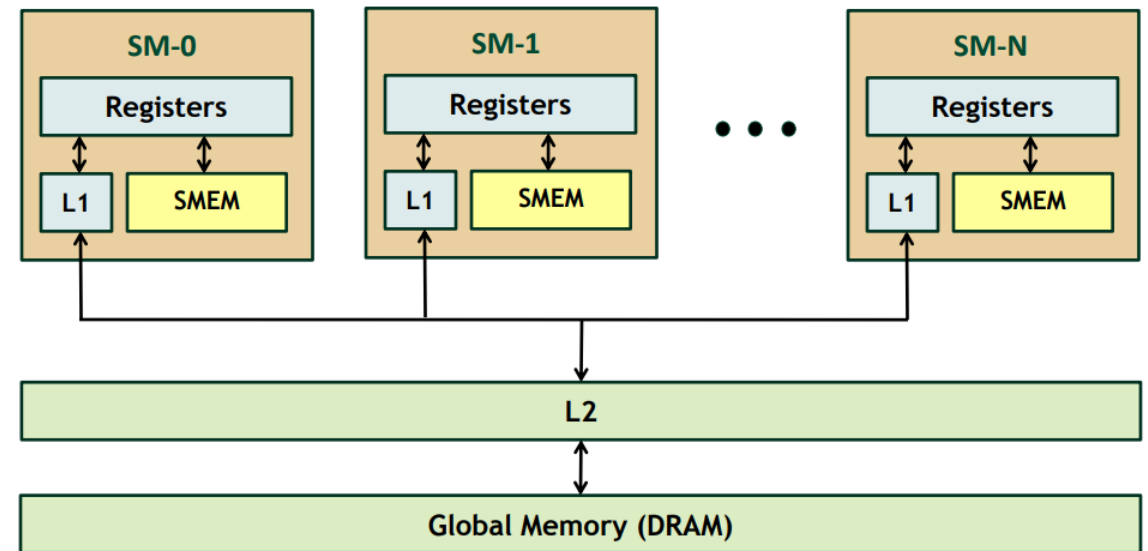
Registers

- Scalar kernel variables are mapped to registers
- If you need more registers than are available, they spill into L1 cache
- If you exceed L1 cache, they spills to global memory => big performance impact


Fermi Memory Hierarchy

Unroll Small Local Arrays to Registers

```
__global__ void my_kernel() {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  float local_array[4];

  for (int k = 0; k < 4; k++){
    local_array[k] = calculate_some_value(k);
  }
  //some more code using local_array

}
```

```
__global__ void my_kernel() {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  float local_array_0 = calculate_some_value(0);
  float local_array_1 = calculate_some_value(1);
  float local_array_2 = calculate_some_value(2);
  float local_array_3 = calculate_some_value(3);
  //some more code using local_array

}
```

24

Texture Memory

- Read-only memory

- Cached on chip

- Texture caches are designed for graphics application where memory access patterns exhibit a great deal of "**spatial locality**" – a thread is likely to read from an address "near" the address that nearby threads read
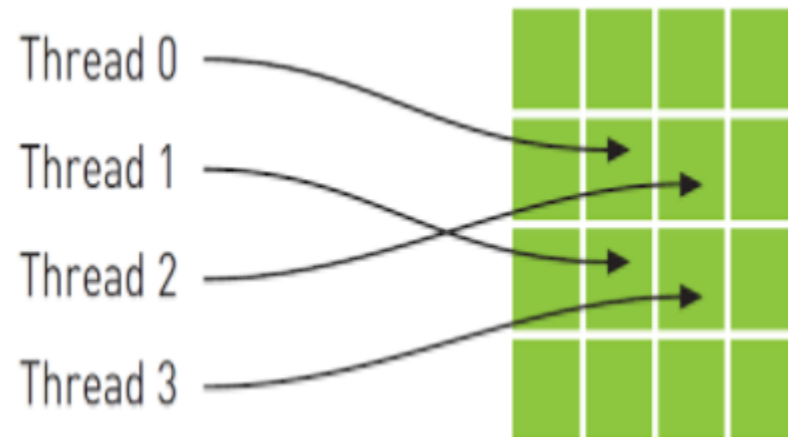
Memory as Limiting Factor to Parallelism

- Each CUDA device offers a limited amount of CUDA memory, which 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

Memory as Limiting Factor to Parallelism

- Example: Registers
  - G80 has 8K registers per SM -> 128K registers for the 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 at the block granularity.
    - E.g., if each block contains 256 threads, the number of threads will be reduced 256 at a time -> lowering the number of threads/SM from 768 to 512, a 1/3 reduction of threads

Memory as Limiting Factor to Parallelism

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

CUDA Memories Summary

- CUDA defined registers, shared memory, and constant memory can be accessed at higher speed and in a more parallel manner than the global memory
  - Requires redesign of the algorithm (tiling, locality of data access)
  - Be aware of the limited sizes of these types of memory (capacities are implementation dependent)
- The ability to reason about hardware limitations when developing an application is a key concept of computational thinking