GPU Memory II

— Memory Hardware and Bank Conflict
CUDA Device Memory Space: Review

- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory

- The host can R/W global, constant, and texture memories
Parallel Memory Sharing

- **Local Memory**: per-thread
  - Private per thread
  - Auto variables, register spill

- **Shared Memory**: per-Block
  - Shared by threads of the same block
  - Inter-thread communication

- **Global Memory**: per-application
  - Shared by all threads
  - Inter-Grid communication

Copyright © 2010 by Yong Cao, Referencing UIUC ECE498AL Course Notes
Hardware Overview

Streaming Processor Array

Thread Processor Cluster

Streaming Multiprocessor

Instruction Fetch/Dispatch

Shared Memory

Instruction L1

Data L1

Special Function Unit (SFU)

Copyright © 2010 by Yong Cao, Referencing UIUC ECE498AL Course Notes
Register File

- Register File (RF)
  - 32 KB
  - Provides 4 operands/clock
- Texture pipe can also read/write RF
  - 2 SMs share 1 TEX
- Load/Store pipe can also read/write RF
There are 8192 registers in each SM in G80

- Registers are dynamically partitioned across all Blocks assigned to the SM
- Once assigned to a Block, the register is NOT accessible by threads in other Blocks
- Each thread in the same Block only access registers assigned to itself
If each Block has 16X16 threads and each thread uses 10 registers, how many thread can run on each SM?

- Each Block requires 10*256 = 2560 registers
- 8192 = 3 * 2560 + change
- So, three blocks can run on an SM as far as registers are concerned

How about if each thread increases the use of registers by 1?

- Each Block now requires 11*256 = 2816 registers
- 8192 < 2816 *3
- Only two Blocks can run on an SM, 1/3 reduction of parallelism!!!
More on Dynamic Partitioning

- Dynamic partitioning gives more flexibility to compilers/programmers
  - One can run a smaller number of threads that require many registers each or a large number of threads that require few registers each
    - This allows for finer grain threading than traditional CPU threading models.
  - The compiler can tradeoff between instruction-level parallelism and thread level parallelism
Assume that a kernel has 256-thread Blocks, 4 independent instructions for each global memory load in the thread program, and each thread uses 10 registers, global loads have 200 cycles

- 3 Blocks can run on each SM

If a Compiler can use one more register to change the dependence pattern so that 8 independent instructions exist for each global memory load

- Only two can run on each SM
- However, one only needs 200/(8*4) = 7 Warps to tolerate the memory latency
- Two Blocks have 16 Warps. The performance can be actually higher!
Constant

- Immediate address constants
- Indexed address constants
- Constants stored in DRAM, and cached on chip
  - L1 per SM
- A constant value can be broadcast to all threads in a Warp
  - Extremely efficient way of accessing a value that is common for all threads in a Block!
Shared Memory

- Each Multi-processor has 16 KB of Shared Memory
  - 16 banks of 32bit words
  - Will discuss about accessing pattern later
- Visible to all threads in a thread block
  - read and write access
Explore Tile-based implementation with Shared Memory.

Question:
- How is shared memory organized?
- What are the issues when accessing shared memory?
Tile Based Multiplication

- One block computes one square sub-matrix $P_{sub}$ of size $\text{BLOCK\_SIZE}$
- One thread computes one element of $P_{sub}$
- Assume that the dimensions of $M$ and $N$ are multiples of $\text{BLOCK\_SIZE}$ and square shape
Tiled Matrix Multiplication Kernel -- Review

```c
__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;

  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;

  float Pvalue = 0;

  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[Col + (m*TILE_WIDTH + ty)*Width];
    __syncthreads();

    for (int k = 0; k < TILE_WIDTH; ++k) {
      Pvalue += Mds[ty][k] * Nds[k][tx];
    }
    __syncthreads();
  } 
  Pd[Row*Width+Col] = Pvalue;
}
```
Each Block requires $2^2 \times BLOCK\_SIZE^2 \times 4$ bytes of shared memory storage

- For $BLOCK\_SIZE = 16$, each BLOCK requires 2KB, up to 8 Blocks can fit into the Shared Memory of an SM
- Since each SM can only take 768 threads, each SM can only take 3 Blocks of 256 threads each
- Occupancy is not limited by Shared memory
Shared Memory Organization

- Parallel Memory Architecture:
  - Memory is divided into banks
  - Essential to achieve high bandwidth

- Each bank can service one address per cycle:
  - A memory can service as many simultaneous accesses as it has banks

- Multiple simultaneous accesses to a bank result in a bank conflict:
  - Conflicting accesses are serialized
Share Memory Access Issue

- No Bank Conflicts
  - Linear addressing
    stride == 1

- No Bank Conflicts
  - Random 1:1 Permutation

Copyright © 2010 by Yong Cao, Referencing UIUC ECE498AL Course Notes
Share Memory Access Issue

- **2-way Bank Conflicts**
  - Linear addressing
  - stride == 2

- **8-way Bank Conflicts**
  - Linear addressing
  - stride == 8

Copyright © 2010 by Yong Cao, Referencing UIUC ECE498AL Course Notes
How addresses map to banks in CUDA

- Each bank has a bandwidth of 32 bits per clock cycle
- Successive 32-bit words are assigned to successive banks
- G80 has 16 banks
  - So bank = address % 16
  - Same as the size of a half-warp
    - No bank conflicts between different half-warps, only within a single half-warp
Share Memory Performance

- Shared memory is as fast as registers if there are no bank conflicts.

  - The fast case:
    - If all threads of a half-warp access different banks, there is no bank conflict.
    - If all threads of a half-warp access the identical address, there is no bank conflict (broadcast).

  - The slow case:
    - Bank Conflict: multiple threads in the same half-warp access the same bank.
    - Must serialize the accesses.
    - Cost = max # of simultaneous accesses to a single bank.
Linear Addressing (1D)

- **Given:**
  ```
  __shared__ float shared[256];
  float foo = shared[baseIndex + s * threadIdx.x];
  ```

- **This is only bank-conflict-free if** \( s \) **shares no common factors with the number of banks**
  - 16 on G80, so \( s \) **must be odd**
This has no conflicts if type of `shared` is 32-bits:

```c
foo = shared[baseIndex + threadIdx.x]
```

But not if the data type is smaller

- 4-way bank conflicts:
  ```c
  __shared__ char shared[];
  foo = shared[baseIndex + threadIdx.x];
  ```

- 2-way bank conflicts:
  ```c
  __shared__ short shared[];
  foo = shared[baseIndex + threadIdx.x];
  ```
Structs and Bank Conflicts

- Struct assignments compile into as many memory accesses as there are struct members:

  ```c
  struct vector { float x, y, z; }
  struct myType {
    float f;
    int c;
  };
  __shared__ struct vector vectors[64];
  __shared__ struct myType myTypes[64];
  ```

- This has no bank conflicts for vector; struct size is 3 words
  - 3 accesses per thread, contiguous banks (no common factor with 16)

  ```c
  struct vector v = vectors[baseIndex + threadIdx.x];
  ```

- This has 2-way bank conflicts for my Type; (2 accesses per thread)

  ```c
  struct myType m = myTypes[baseIndex + threadIdx.x];
  ```
Each thread loads 2 elements into shared memory:
  2-way-interleaved loads result in 2-way bank conflicts:

```c
int tid = threadIdx.x;
shared[2*tid] = global[2*tid];
shared[2*tid+1] = global[2*tid+1];
```

This makes sense for traditional CPU threads, locality in cache line usage and reduced sharing traffic.
  Not in shared memory usage where there is no cache line effects but banking effects
A Better Array Access Pattern

- Each thread loads one element in every consecutive group of blockDim elements.

\[
\text{shared}[\text{tid}] = \text{global}[\text{tid}]; \\
\text{shared}[\text{tid} + \text{blockDim.x}] = \text{global}[\text{tid} + \text{blockDim.x}];
\]
Common Bank Conflict Patterns (2D)

- Operating on 2D array of floats in shared memory
  - e.g. image processing

- Example: 16x16 block
  - Each thread processes a row
  - So threads in a block access the elements in each column simultaneously (example: row 1 in purple)
  - 16-way bank conflicts: rows all start at bank 0

- Solution 1) pad the rows
  - Add one float to the end of each row

- Solution 2) transpose before processing
  - Suffer bank conflicts during transpose

Bank Indices without Padding:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>...</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>...</td>
<td>15</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>...</td>
<td>15</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>...</td>
<td>15</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>...</td>
<td>15</td>
</tr>
</tbody>
</table>

Bank Indices with Padding:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>...</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>...</td>
<td>15</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>...</td>
<td>15</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>...</td>
<td>15</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>...</td>
<td>15</td>
</tr>
</tbody>
</table>

Copyright © 2010 by Yong Cao, Referencing UIUC ECE498AL Course Notes
Does Matrix Multiplication Incur Shared Memory Bank Conflicts?

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

Copyright © 2010 by Yong Cao, Referencing UIUC ECE498AL Course Notes
Does Matrix Multiplication Incur Shared Memory Bank Conflicts?

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

Mds[ty*TILE_WIDTH + k]  Nds[k*TILE_WIDTH + tx]
Does Matrix Multiplication Incur Shared Memory Bank Conflicts?

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

For TILE_WIDTH = 16
- The whole half-warp is accessing the same shared memory location.
- Conflict. But, GPU support broadcasting.
Does Matrix Multiplication Incur Shared Memory Bank Conflicts?

12. for (int k = 0; k < TILE_WIDTH; ++k) {
13.     pvalue += Mds[ty][k] * Nds[k][tx];
14.     syncthreads();
15. }

For TILE_WIDTH = 8
- The first half-warp and the second half-warp are accessing two different shared memory location.
- 8-way bank conflict.
Does Matrix Multiplication Incur Shared Memory Bank Conflicts?

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

For TILE_WIDTH = 4
- 4-way bank conflict.
Does Matrix Multiplication Incur Shared Memory Bank Conflicts?

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

For **TILE_WIDTH** = 16
- Each thread in a half-warp is accessing different shared memory location.
- No conflict.
Does Matrix Multiplication Incur Shared Memory Bank Conflicts?

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

For TILE_WIDTH = 8
- Since the memory storage organization is row-major for 2D array, so it’s the same with TILE_WIDTH = 16.
- No conflict.