CS 380 - GPU and GPGPU Programming
Lecture 19: CUDA Memory Access 3

Markus Hadwiger, KAUST
Reading Assignment #9 (until Apr. 14)

Read (required):

• Programming Massively Parallel Processors book, Chapter 5 (*CUDA Memories*)

• **CUDA C Programming Guide 5.0**
  Appendix F: Compute Capabilities

  Study the different memory access requirements for different compute capabilities
Assignment #3:

- Image Processing with (a) GLSL, and (b) CUDA  
  due Apr 7

Assignment #4:

- Conjugate Gradient Linear Systems Solver (CUDA)  
  due Apr 28
2. Shared Memory Accesses

- Banked memory access / bank conflicts
Parallel Memory Architecture

- In a parallel machine, many threads access memory
  - Therefore, 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
Bank Addressing Examples

- No Bank Conflicts
  - Linear addressing
    stride == 1
  
- Random 1:1 Permutation

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE 498AL, University of Illinois, Urbana-Champaign
Bank Addressing Examples

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

- 8-way Bank Conflicts
  - Linear addressing
  - stride == 8
How addresses map to banks on G80

- 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-warsps, only within a single half-warp

Fermi has 32 banks, considers full warps instead of half warps!
Shared Memory Bank Conflicts

- 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

full warps instead of half warps on Fermi!
Linear Addressing

- **Given:**

  \[
  \_\_\_\_\_\text{shared}\_\_\_\_\_\text{ float shared}[256];
  \]

  \[
  \text{float foo =}
  \]

  \[
  \text{shared[baseIndex + s * threadIdx.x]};
  \]

- **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**
Data Types and Bank Conflicts

- This has no conflicts if type of shared is 32-bits:
  
  ```
  foo = shared[baseIndex + threadIdx.x]
  ```

- But not if the data type is smaller
  - 4-way bank conflicts:
    ```
    __shared__ char shared[];
    foo = shared[baseIndex + threadIdx.x];
    ```
    not true on Fermi, because of multi-cast!

  - 2-way bank conflicts:
    ```
    __shared__ short shared[];
    foo = shared[baseIndex + threadIdx.x];
    ```
    not true on Fermi, because of multi-cast!
Structs and Bank Conflicts

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

```
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)

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

- **This has 2-way bank conflicts for myType;**
  (each bank will be accessed by 2 threads simultaneously)

```
struct myType m = myTypes[baseIndex + threadIdx.x];
```
Broadcast on Shared Memory

- Each thread loads the same element – no bank conflict
  \[ x = \text{shared}[0]; \]
- Will be resolved implicitly

multi-cast on Fermi!
Common Array Bank Conflict Patterns

1D

- Each thread loads 2 elements into shared mem:
  - 2-way-interleaved loads result in 2-way bank conflicts:

```
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.

```
shared[tid] = global[tid];
shared[tid + blockDim.x] = global[tid + blockDim.x];
```
Texture Memory
Texture Memory

- **Cached**, potentially exhibiting higher bandwidth if there is locality in the texture fetches;
- They are not subject to the constraints on memory access patterns that global or constant memory reads must respect to get good performance;
- The latency of addressing calculations is hidden better, possibly improving performance for applications that perform random accesses to the data;
- No penalty when accessing float4;
- Optional
  - 8-bit and 16-bit integer input data may be optionally converted to 32-bit floatingpoint
  - Packed data may be broadcast to separate variables in a single operation;
  - values in the range $[0.0, 1.0]$ or $[-1.0, 1.0]$
  - texture filtering
  - address modes, e.g. wrapping / texture borders
Native Memory Layout – Data Locality

CPU
- 1D input
- 1D output
- Other dimensions with offsets

GPU
- 2D input
- 2D output
- Other dimensions with offsets

Color coded locality
red (near), blue (far)

Input

Output
1D Access

• Access to linear Cuda memory
  ```c
  float4* pos; cudaMalloc( (void**)&pos, x*sizeof(float4) );
  ```

• Texture reference
  – type
  – access/filtering mode
  ```c
  // global texture reference
  texture< float4, 1, cudaReadModeElementType> texPos;
  ```

• Bind to linear array
  ```c
  cudaBindTexture(0, texPos, pos, x*sizeof(float4));
  cudaUnbindTexture(texPos);
  ```

• Within kernel
  ```c
  float4 pal = tex1Dfetch( texPos, threadIdx.x);
  ```

• Writing to a texture that is currently read by some threads is undefined!!!
2D Access

- Optimized for 2D / 3D locality

```c
texture< float4, 2, cudaReadModeElementType> texImg;
```

- Requires binding to special Array memory – special memory layout

```c
cudaChannelFormatDesc floatTex =
cudaCreateChannelDesc<float4>();
float4* src;
cudaArray* img;
cudaMallocArray( &img, &floatTex, w, h);
cudaMemcpyToArray(img, 0, 0, src, w*h*sizeof(float4),
cudaMemcpyHostToDevice);
cudaBindTextureToArray( texImg, img, floatTex );
cudaUnbindTexture(texImg);
```
2D Access

- **Within kernel**
  
  ```
  float4 r = tex2D( teximg, x +xoff, y+yoff);
  ```

- **Pros**
  - optimized for 2D locality (optimized memory layout / spacefilling curve)

- **Cons**
  - If the result of some kernel should be used as 2D texture, `cudaMemcpyToArray` is required
  - You cannot write to a texture which is currently read from
Thank you.

- Hendrik Lensch, Robert Strzodka