L8: Memory Hierarchy Optimization, Bandwidth

Administrative

- Homework #2
  - Due 5PM, TODAY
  - Use handin program to submit
- Project proposals
  - Due 5PM, Friday, March 13 (hard deadline)
- MPM Sequential code and information posted on website
- Class cancelled on Wednesday, Feb. 25
- Questions?

Targets of Memory Hierarchy Optimizations

- Reduce memory latency
  - The latency of a memory access is the time (usually in cycles) between a memory request and its completion
- Maximize memory bandwidth
  - Bandwidth is the amount of useful data that can be retrieved over a time interval
- Manage overhead
  - Cost of performing optimization (e.g., copying) should be less than anticipated gain

Optimizing the Memory Hierarchy on GPUs

- Device memory access times non-uniform so data placement significantly affects performance.
  - But controlling data placement may require additional copying, so consider overhead.
- Optimizations to increase memory bandwidth.
  - Align data structures to address boundaries
  - Coalesce global memory accesses
  - Avoid memory bank conflicts to increase memory access parallelism
Outline

- Bandwidth Optimizations
  - Parallel memory accesses in shared memory
  - Maximize utility of every memory operation
    - Load/store USEFUL data
- Architecture Issues
  - Shared memory bank conflicts
  - Global memory coalescing
  - Alignment

Global Memory Accesses

- Each thread issues memory accesses to data types of varying sizes, perhaps as small as 1 byte entities
- Given an address to load or store, memory returns/updates "segments" of either 32 bytes, 64 bytes or 128 bytes
- Maximizing bandwidth:
  - Operate on an entire 128 byte segment for each memory transfer

Understanding Global Memory Accesses

Memory protocol for compute capability 1.2* (CUDA Manual 5.1.2.1)

- Start with memory request by smallest numbered thread. Find the memory segment that contains the address (32, 64 or 128 byte segment, depending on data type)
- Find other active threads requesting addresses within that segment and coalesce
- Reduce transaction size if possible
- Access memory and mark threads as "inactive"
- Repeat until all threads in half-warp are serviced

*Includes Tesla and GTX platforms

Protocol for most systems (including lab machines) even more restrictive

- For compute capability 1.0 and 1.1
  - Threads must access the words in a segment in sequence
  - The kth thread must access the kth word
Impact of Global Memory Coalescing (Compute capability 1.1 and below example)

Consider the following CUDA kernel that reverses the elements of an array*:

```c
__global__ void reverseArrayBlock(int *d_out, int *d_in) {
    int inOffset = blockDim.x * blockIdx.x;
    int in = inOffset + threadIdx.x;
    int out = inOffset + blockDim.x - 1 - threadIdx.x;
    d_out[out] = d_in[in];
}
```


---

Shared Memory Version of Reverse Array

```c
__global__ void reverseArrayBlock(int *d_out, int *d_in) {
    extern __shared__ int s_data[];
    int inOffset = blockDim.x * blockIdx.x;
    int in = inOffset + threadIdx.x;
    // Load one element per thread from device memory and store it
    // *in reversed order* into temporary shared memory
    s_data[blockDim.x - 1 - threadIdx.x] = d_in[in];
    __syncthreads();
    // write the data from shared memory in forward order,
    // but to the reversed block offset as before
    d_out[out] = s_data[threadIdx.x];
}
```

What Happened?

- The first version is about 50% slower!
- On examination, the same amount of data is transferred to/from global memory
- Let’s look at the access patterns
  - More examples in CUDA programming guide

Alignment

- Addresses accessed within a half-warp may need to be aligned to the beginning of a segment to enable coalescing
  - An aligned memory address is a multiple of the memory segment size
  - In compute 1.0 and 1.1 devices, address accessed by lowest numbered thread must be aligned to beginning of segment for coalescing
  - In future systems, sometimes alignment can reduce number of accesses

More on Alignment

- Objects allocated statically or by cudaMalloc begin at aligned addresses
  - But still need to think about index expressions
- May want to align structures
  ```c
  struct __align__(8) {
    float a;
    float b;
  };
  ```

What Can You Do to Improve Bandwidth to Global Memory?

- Think about spatial reuse and access patterns across threads
  - May need a different computation & data partitioning
  - May want to rearrange data in shared memory, even if no temporal reuse
  - Similar issues, but much better in future hardware generations
Bandwidth to Shared Memory: Parallel Memory Accesses

• Consider each thread accessing a different location in shared memory
• Bandwidth maximized if each one is able to proceed in parallel
• Hardware to support this
  – Banked memory: each bank can support an access on every memory cycle
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.

Linear Addressing

- 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.

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];
    ```
  - 2-way bank conflicts:
    ```
    __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:
  ```
  struct vector { float x, y, z; };
  struct myType {
    float f;
    int c;
  };
  ```

- This has no bank conflicts for `vector`, struct size is 3 words:
  ```
  __shared__ struct vector vectors[64];
  __shared__ struct myType myTypes[64];
  ```

- This has 2-way bank conflicts for `myType`, (2 accesses per thread):
  ```
  __shared__ struct myType m = myTypes[baseIndex + threadIdx.x];
  ```
Common Bank Conflict Patterns, 1D Array

- 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, exploits spatial locality in cache line and reduces 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];
  ```

What Can You Do to Improve Bandwidth to Shared Memory?

- Think about memory access patterns across threads
  - May need a different computation & data partitioning
  - Sometimes "padding" can be used on a dimension to align accesses

Summary of Lecture

- Maximize Memory Bandwidth!
  - Make each memory access count
- Exploit spatial locality in global memory accesses
- The opposite goal in shared memory
  - Each thread accesses independent memory banks