Reading Assignment #11 (until Apr. 20)

Read (required):

- Programming Massively Parallel Processors book, Chapter 6 (Performance Considerations)
- CUDA 7.0 C Programming Guide, Appendix G (Compute Capabilities)
2. Shared Memory Accesses

- Banked memory access / bank conflicts
OPTIMIZE

Kernel Optimizations: *Shared Memory Accesses*
Shared Memory

- Accessible by all threads in a block
- Fast compared to global memory
  - Low access latency
  - High bandwidth
- Common uses:
  - Software managed cache
  - Data layout conversion

Global Memory (DRAM)
Shared Memory/L1 Sizing

- Shared memory and L1 use the same 64KB
  - Program-configurable split:
    - Fermi: 48:16, 16:48
    - Kepler: 48:16, 16:48, 32:32
  - CUDA API: `cudaDeviceSetCacheConfig()`, `cudaFuncSetCacheConfig()`

- Large L1 can improve performance when:
  - Spilling registers (more lines in the cache -> fewer evictions)

- Large SMEM can improve performance when:
  - Occupancy is limited by SMEM
Shared Memory

Uses:
- Inter-thread communication within a block
- Cache data to reduce redundant global memory accesses
- Use it to improve global memory access patterns

Organization:
- 32 banks, 4-byte (or 8-byte) banks
- Successive words accessed through different banks
Memory Banks

Fermi/Kepler/Maxwell:
32 banks

default: 4B / bank

Kepler: configurable to 8B / bank
Shared Memory

**Uses:**
- Inter-thread communication within a block
- Cache data to reduce redundant global memory accesses
- Use it to improve global memory access patterns

**Performance:**
- smem accesses are issued per warp
- Throughput is 4 (or 8) bytes per bank per clock per multiprocessor
- **serialization:** if $N$ threads of 32 access different words in the same bank, $N$ accesses are executed serially
- **multicast:** $N$ threads access the same word in one fetch
  - Could be different bytes within the same word
Shared Memory Organization

- Organized in 32 independent banks
- Optimal access: no two words from same bank
  - Separate banks per thread
  - Banks can multicast
- Multiple words from same bank serialize

Any 1:1 or multicast pattern
Bank Addressing Examples

- **No Bank Conflicts**
  - Thread 0 → Bank 0
  - Thread 1 → Bank 1
  - Thread 2 → Bank 2
  - Thread 3 → Bank 3
  - Thread 4 → Bank 4
  - Thread 5 → Bank 5
  - Thread 6 → Bank 6
  - Thread 7 → Bank 7
  - Thread 31 → Bank 31

- **No Bank Conflicts**
  - Thread 0 → Bank 0
  - Thread 1 → Bank 1
  - Thread 2 → Bank 2
  - Thread 3 → Bank 3
  - Thread 4 → Bank 4
  - Thread 5 → Bank 5
  - Thread 6 → Bank 6
  - Thread 7 → Bank 7
  - Thread 31 → Bank 31
Bank Addressing Examples

- **2-way Bank Conflicts**
  - Thread 0
  - Thread 1
  - Thread 2
  - Thread 3
  - Thread 4
  - Thread 28
  - Thread 29
  - Thread 30
  - Thread 31
  - Bank 0
  - Bank 1
  - Bank 2
  - Bank 3
  - Bank 4
  - Bank 5
  - Bank 6
  - Bank 7
  - Bank 31

- **8-way Bank Conflicts**
  - Thread 0
  - Thread 1
  - Thread 2
  - Thread 3
  - Thread 4
  - Thread 5
  - Thread 6
  - Thread 7
  - Thread 31
  - Bank 0
  - Bank 1
  - Bank 2
  - Bank 7
  - Bank 8
  - Bank 9
  - Bank 31
Case Study: Matrix Transpose

- Coalesced read
- Scattered write (stride N)

⇒ Process matrix tile, not single row/column, per block

⇒ Transpose matrix tile within block
Case Study: Matrix Transpose

- Coalesced read
- Scattered write (stride N)
- Transpose matrix tile within block

⇒ Need threads in a block to cooperate: use shared memory
Transpose with coalesced read/write

```c
__global__ transpose(float in[], float out[])
{
  __shared__ float tile[TILE][TILE];

  int glob_in = xIndex + (yIndex)*N;
  int glob_out = xIndex + (yIndex)*N;

  tile[threadIdx.y][threadIdx.x] = in[glob_in];

  __syncthreads();

  out[glob_out] = tile[threadIdx.x][threadIdx.y];
}
```

Fixed GMEM coalescing, but introduced SMEM bank conflicts

```c
threads <<<grid, threads>>>(in, out);
```
Shared Memory: Avoiding Bank Conflicts

- Example: 32x32 SMEM array
- Warp accesses a column:
  - 32-way bank conflicts (threads in a warp access the same bank)

```
<table>
<thead>
<tr>
<th>Bank 0</th>
<th>Bank 1</th>
<th>...</th>
<th>Bank 31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 31</td>
<td>0 1 2 31</td>
<td></td>
<td>0 1 2 31</td>
</tr>
<tr>
<td>0 1 2 31</td>
<td>0 1 2 31</td>
<td></td>
<td>0 1 2 31</td>
</tr>
<tr>
<td>0 1 2 31</td>
<td>0 1 2 31</td>
<td></td>
<td>0 1 2 31</td>
</tr>
<tr>
<td>0 1 2 31</td>
<td>0 1 2 31</td>
<td></td>
<td>0 1 2 31</td>
</tr>
</tbody>
</table>
```
Shared Memory: Avoiding Bank Conflicts

- Add a column for padding:
  - 32x33 SMEM array
- Warp accesses a column:
  - 32 different banks, no bank conflicts

```plaintext
<table>
<thead>
<tr>
<th>Bank 0</th>
<th>Bank 1</th>
<th>...</th>
<th>Bank 31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
```

```plaintext
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>31</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>
```
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

- No Bank Conflicts
  - Random 1:1 Permutation
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-warps, 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:**

  ```c
  __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:
  ```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];
    ```
    not true on Fermi, because of multi-cast!

  - 2-way bank conflicts:
    ```c
    __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:

```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 myType;
  (each bank will be accessed by 2 threads simultaneously)

```c
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!
Each thread loads 2 elements into shared mem:
- 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.

\[
\text{shared[tid]} = \text{global[tid]}; \\
\text{shared[tid + blockDim.x]} = \text{global[tid + blockDim.x]};
\]
OPTIMIZE

Kernel Optimizations: *Global Memory Throughput*
Kepler Memory Hierarchy

SM-0
- Registers
- L1
- SMEM
- Read only

SM-1
- Registers
- L1
- SMEM
- Read only

SM-N
- Registers
- L1
- SMEM
- Read only

...
Kepler Memory Hierarchy

- **Registers**
  - Storage local to each thread
  - Compiler-managed

- **Shared memory / L1 cache**
  - 64 KB, program-configurable into shared:L1
  - Program-managed
  - Accessible by all threads in the same thread block
  - Low latency, high bandwidth: \( \sim 2.5 \text{ TB/s} \)

- **Read-only cache**
  - Up to 48 KB per Kepler SM
  - Hardware-managed (also used by texture units)
  - Used for read-only GMEM accesses (not coherent with writes)
Texture performance

Texture:
- Provides hardware accelerated filtered sampling of data (1D, 2D, 3D)
- Read-only data cache holds fetched samples
- Backed up by the L2 cache

SMX vs Fermi SM:
- 4x filter ops per clock
- 4x cache capacity
Texture Cache Unlocked

- Added a new path for compute
  - Avoids the texture unit
  - Allows a global address to be fetched and cached
  - Eliminates texture setup
- Why use it?
  - Separate pipeline from shared/L1
  - Highest miss bandwidth
  - Flexible, e.g. unaligned accesses
- Managed automatically by compiler
  - “const __restrict” indicates eligibility
Kepler Memory Hierarchy

- **L2**
  - 1.5 MB
  - Hardware-managed: all accesses to global memory go through L2, including CPU and peer GPU

- **Global memory**
  - 6 GB, accessible by all threads, host (CPU), other GPUs in the same system
  - Higher latency (400-800 cycles)
  - 250 GB/s
Load Operation

- Memory operations are issued **per warp** (32 threads)
  - Just like all other instructions

**Operation:**
- Threads in a warp provide memory addresses
- Determine which lines/segments are needed
- Request the needed lines/segments
Memory Throughput Analysis

Two perspectives on the throughput:
- Application’s point of view:
  - count only bytes requested by application
- HW point of view:
  - count all bytes moved by hardware

The two views can be different:
- Memory is accessed at 32 byte granularity
  - Scattered/offset pattern: application doesn’t use all the hw transaction bytes
  - Broadcast: the same small transaction serves many threads in a warp

Two aspects to inspect for performance impact:
- Address pattern
- Number of concurrent accesses in flight
Global Memory Operation

Memory operations are executed per warp
- 32 threads in a warp provide memory addresses
- Hardware determines into which lines those addresses fall
  - Memory transaction granularity is 32 bytes
  - There are benefits to a warp accessing a contiguous aligned region of 128 or 256 bytes

Access word size
- Natively supported sizes (per thread): 1, 2, 4, 8, 16 bytes
  - Assumes that each thread’s address is aligned on the word size boundary
- If you are accessing a data type that’s of non-native size, compiler will generate several load or store instructions with native sizes
Access Patterns vs. Memory Throughput

- **Scenario:**
  - Warp requests 32 aligned, consecutive 4-byte words
  - **Addresses fall within 4 segments**
    - Warp needs 128 bytes
    - 128 bytes move across the bus
    - Bus utilization: 100%

addresses from a warp

Memory addresses

© NVIDIA 2013
Access Patterns vs. Memory Throughput

Scenario:
- Warp requests 32 aligned, permuted 4-byte words
- Addresses fall within 4 segments
  - Warp needs 128 bytes
  - 128 bytes move across the bus
  - Bus utilization: 100%
Access Patterns vs. Memory Throughput

- **Scenario:**
  - Warp requests 32 misaligned, consecutive 4-byte words

- **Addresses fall within at most 5 segments**
  - Warp needs 128 bytes
  - At most 160 bytes move across the bus
  - Bus utilization: at least 80%
    - Some misaligned patterns will fall within 4 segments, so 100% utilization

addresses from a warp

Memory addresses

© NVIDIA 2013
Access Patterns vs. Memory Throughput

**Scenario:**
- All threads in a warp request the same 4-byte word
- Addresses fall within a single segment
  - Warp needs 4 bytes
  - 32 bytes move across the bus
  - Bus utilization: 12.5%

addresses from a warp

Memory addresses

© NVIDIA 2013
Access Patterns vs. Memory Throughput

Scenario:
- Warp requests 32 scattered 4-byte words
- Addresses fall within $N$ segments
  - Warp needs 128 bytes
  - $N\times32$ bytes move across the bus
  - Bus utilization: $128 / (N\times32)$

addresses from a warp
Structures of Non-Native Size

Say we are reading a 12-byte structure per thread

```c
struct Position {
    float x, y, z;
};
...
__global__ void kernel( Position *data, ... )
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    Position temp = data[idx];
    ...
}
```
Structure of Non-Native Size

Compiler converts \( \text{temp} = \text{data}[\text{idx}] \) into 3 loads:
- Each loads 4 bytes
- Can’t do an 8 and a 4 byte load: 12 bytes per element means that every other element wouldn’t align the 8-byte load on 8-byte boundary

Addresses per warp for each of the loads:
- Successive threads read 4 bytes at 12-byte stride
First Load Instruction

addresses from a warp
Second Load Instruction

addresses from a warp
Third Load Instruction

addresses from a warp

© NVIDIA 2013
Performance and Solutions

Because of the address pattern, we end up moving 3x more bytes than application requests

- We waste a lot of bandwidth, leaving performance on the table

Potential solutions:

- Change data layout from array of structures to structure of arrays
  - In this case: 3 separate arrays of floats
  - The most reliable approach (also ideal for both CPUs and GPUs)
- Use loads via read-only cache
  - As long as lines survive in the cache, performance will be nearly optimal
- Stage loads via shared memory
Global Memory Access Patterns

- **SoA vs AoS:**
  - **Good:** \( \text{point.x[i]} \)
  - **Not so good:** \( \text{point[i].x} \)

- **Strided array access:**
  - **~OK:** \( x[i] = a[i+1] - a[i] \)
  - **Slower:** \( x[i] = a[64*i] - a[i] \)

- **Random array access:**
  - **Slower:** \( a[\text{rand}(i)] \)
Summary: GMEM Optimization

- Strive for perfect address coalescing per warp
  - Align starting address (may require padding)
  - A warp will ideally access within a contiguous region
  - Avoid scattered address patterns or patterns with large strides between threads

- Analyze and optimize address patterns:
  - Use profiling tools (included with CUDA toolkit download)
  - Compare the transactions per request to the ideal ratio
  - Choose appropriate data layout (prefer SoA)
  - If needed, try read-only loads, staging accesses via SMEM
A note about caches

- **L1 and L2 caches**
  - Ignore in software design
  - Thousands of concurrent threads – cache blocking difficult at best

- **Read-only Data Cache**
  - Shared with texture pipeline
  - Useful for uncoalesced reads
  - Handled by compiler when `const __restrict__` is used, or use `_ldg()` primitive

---

© NVIDIA 2013
Read-only Data Cache

- Go through the read-only cache
  - Not coherent with writes
  - Thus, addresses must not be written by the same kernel

- Two ways to enable:
  - Decorating pointer arguments as hints to compiler:
    - Pointer of interest: `const __restrict__`
    - All other pointer arguments: `__restrict__`
      - Conveys to compiler that no aliasing will occur
  - Using `__ldg()` intrinsic
    - Requires no pointer decoration
Read-only Data Cache

Go through the read-only cache
- Not coherent with writes
- Thus, addresses must not be written by the same kernel

Two ways to enable:
- Decorating pointer arguments
  - Pointer of interest: `const` marked
  - All other pointer arguments:
    - Conveys to compiler that pointers are not modified
- Using `_ldg()` intrinsic
  - Requires no pointer decoration

```c
__global__ void kernel(
    int* __restrict__ output,
    const int* __restrict__ input )
{
    ...
    output[idx] = input[idx];
}
```
Read-only Data Cache

- Go through the read-only cache
  - Not coherent with writes
  - Thus, addresses must not be written by the same kernel

Two ways to enable:

- Decorating pointer arguments
  - Pointer of interest: `const`
  - All other pointer arguments
    - Conveys to compiler that pointer is read-only
- Using `__ldg()` intrinsic
  - Requires no pointer decoration

```c
__global__ void kernel( int *output,
                        int *input )
{
    ...
    output[idx] = __ldg( &input[idx] );
}
```
Blocking for L1, Read-only, L2 Caches

- Short answer: DON’T
- GPU caches are not intended for the same use as CPU caches
  - Smaller size (especially per thread), so not aimed at temporal reuse
  - Intended to smooth out some access patterns, help with spilled registers, etc.
- Usually not worth trying to cache-block like you would on CPU
  - 100s to 1,000s of run-time scheduled threads competing for the cache
  - If it is possible to block for L1 then it’s possible block for SMEM
    - Same size
    - Same or higher bandwidth
    - Guaranteed locality: hw will not evict behind your back
Maximize Byte Use

• Two things to keep in mind:
  – Memory accesses are per warp
  – Memory is accessed in discrete chunks
    • lines/segments
    • want to make sure that bytes that travel from DRAM to SMs get used
      – For that we should understand how memory system works

• Note: not that different from CPUs
  – x86 needs SSE/AVX memory instructions to maximize performance
GPU Memory System

- All data lives in DRAM
  - Global memory
  - Local memory
  - Textures
  - Constants
GPU Memory System

- All DRAM accesses go through L2
- Including copies:
  - P2P
  - CPU-GPU
GPU Memory System

- Once in an SM, data goes into one of 3 caches/buffers
- Programmer’s choice
  - L1 is the “default”
  - Read-only, Const require explicit code
Access Path

• **L1 path**
  – Global memory
    • Memory allocated with cudaMemcpy()
    • Mapped CPU memory, peer GPU memory
    • Globally-scoped arrays qualified with __global__
  – Local memory
    • allocation/access managed by compiler so we’ll ignore

• **Read-only/TEX path**
  – Data in texture objects, CUDA arrays
  – CC 3.5 and higher:
    • Global memory accessed via intrinsics (or specially qualified kernel arguments)

• **Constant path**
  – Globally-scoped arrays qualified with __constant__
Access Via L1

- **Natively supported word sizes per thread:**
  - 1B, 2B, 4B, 8B, 16B
    - Addresses must be aligned on word-size boundary
    - Accessing types of other sizes will require multiple instructions

- **Accesses are processed per warp**
  - Threads in a warp provide 32 addresses
    - Fewer if some threads are inactive
  - HW converts addresses into memory transactions
    - Address pattern may require multiple transactions for an instruction
    - If \( N \) transactions are needed, there will be \((N-1)\) replays of the instruction
GMEM Writes

- Not cached in the SM
  - Invalidate the line in L1, go to L2
- Access is at **32 B** segment granularity
- Transaction to memory: **1, 2, or 4** segments
  - Only the required segments will be sent
- **If multiple threads in a warp write to the same address**
  - One of the threads will "win"
  - Which one is not defined
Some Store Pattern Examples

addresses from a warp  one 4-segment transaction

Memory addresses
Some Store Pattern Examples

addresses from a warp

three 1-segment transactions

Memory addresses
Some Store Pattern Examples

addresses from a warp

one 2-segment transaction

Memory addresses
Some Store Pattern Examples

addresses from a warp

2 1-segment transactions

Memory addresses
**GMEM Reads**

- **Attempt to hit in L1 depends on programmer choice and compute capability**
- **HW ability to hit in L1:**
  - CC 1.x: no L1
  - CC 2.x: can hit in L1
  - CC 3.0, 3.5: cannot hit in L1
    - L1 is used to cache LMEM (register spills, etc.), buffer reads
- **Read instruction types**
  - Caching:
    - Compiler option: `-Xptxas -dlcm=ca`
    - On L1 miss go to L2, on L2 miss go to DRAM
    - Transaction: 128 B line
  - Non-caching:
    - Compiler option: `-Xptxas -dlcm=cg`
    - Go directly to L2 (invalidate line in L1), on L2 miss go to DRAM
    - Transaction: 1, 2, 4 segments, segment = 32 B (same as for writes)
Caching Load

- **Scenario:**
  - Warp requests 32 aligned, consecutive 4-byte words
- **Addresses fall within 1 cache-line**
  - No replays
  - Bus utilization: 100%
    - Warp needs 128 bytes
    - 128 bytes move across the bus on a miss
Non-caching Load

- **Scenario:**
  - Warp requests 32 aligned, consecutive 4-byte words
- **Addresses fall within 4 segments**
  - No replays
  - Bus utilization: 100%
    - Warp needs 128 bytes
    - 128 bytes move across the bus on a miss

addresses from a warp

0 32 64 96 128 160 192 224 256 288 320 352 384 416 448
Memory addresses
Caching Load

- **Scenario:**
  - Warp requests 32 aligned, permuted 4-byte words
- **Addresses fall within 1 cache-line**
  - No replays
  - Bus utilization: 100%
    - Warp needs 128 bytes
    - 128 bytes move across the bus on a miss

![Diagram showing addresses from a warp on Memory addresses 0 to 448]
Non-caching Load

- **Scenario:**
  - Warp requests 32 aligned, permuted 4-byte words
- **Addresses fall within 4 segments**
  - No replays
  - Bus utilization: 100%
    - Warp needs 128 bytes
    - 128 bytes move across the bus on a miss

addresses from a warp
Caching Load

- **Scenario:**
  - Warp requests 32 consecutive 4-byte words, offset from perfect alignment
- **Addresses fall within 2 cache-lines**
  - 1 replay (2 transactions)
  - Bus utilization: 50%
    - Warp needs 128 bytes
    - 256 bytes move across the bus on misses
Non-caching Load

- **Scenario:**
  - Warp requests 32 consecutive 4-byte words, offset from perfect alignment
- **Addresses fall within at most 5 segments**
  - 1 replay (2 transactions)
  - Bus utilization: at least 80%
    - Warp needs 128 bytes
    - At most 160 bytes move across the bus
    - Some misaligned patterns will fall within 4 segments, so 100% utilization
Caching Load

- **Scenario:**
  - All threads in a warp request the same 4-byte word
- **Addresses fall within a single cache-line**
  - No replays
  - Bus utilization: 3.125%
    - Warp needs 4 bytes
    - 128 bytes move across the bus on a miss
Non-caching Load

- **Scenario:**
  - All threads in a warp request the same 4-byte word
- **Addresses fall within a single segment**
  - No replays
  - Bus utilization: 12.5%
    - Warp needs 4 bytes
    - 32 bytes move across the bus on a miss
Caching Load

- **Scenario:**
  - Warp requests 32 scattered 4-byte words
- **Addresses fall within $N$ cache-lines**
  - $(N-1)$ replays ($N$ transactions)
  - Bus utilization: $32 \times 4B / (N \times 128B)$
    - Warp needs 128 bytes
    - $N \times 128$ bytes move across the bus on a miss
Non-caching Load

- Scenario:
  - Warp requests 32 scattered 4-byte words
- Addresses fall within $N$ segments
  - $(N-1)$ replays $N$ transactions
    - Could be lower some segments can be arranged into a single transaction
  - Bus utilization: $128 / (N*32)$ (4x higher than caching loads)
    - Warp needs 128 bytes
    - $N*32$ bytes move across the bus on a miss

addresses from a warp...
Caching vs Non-caching Loads

- **Compute capabilities that can hit in L1 (CC 2.x)**
  - Caching loads are better if you count on hits
  - Non-caching loads are better if:
    - Warp address pattern is scattered
    - When kernel uses lots of LMEM (register spilling)

- **Compute capabilities that cannot hit in L1 (CC 1.x, 3.0, 3.5)**
  - Does not matter, all loads behave like non-caching

- **In general, don’t rely on GPU caches like you would on CPUs:**
  - 100s of threads sharing the same L1
  - 1000s of threads sharing the same L2
L1 Sizing

- Fermi and Kepler GPUs split 64 KB RAM between L1 and SMEM
  - Fermi GPUs (CC 2.x): 16:48, 48:16
  - Kepler GPUs (CC 3.x): 16:48, 48:16, 32:32

- **Programmer can choose the split:**
  - Default: 16 KB L1, 48 KB SMEM
  - Run-time API functions:
    - cudaDeviceSetCacheConfig(), cudaFuncSetCacheConfig()
  - Kernels that require different L1:SMEM sizing cannot run concurrently

- **Making the choice:**
  - Large L1 can help when using lots of LMEM (spilling registers)
  - Large SMEM can help if occupancy is limited by shared memory
Read-Only Cache

• **An alternative to L1 when accessing DRAM**
  – Also known as *texture* cache: all texture accesses use this cache
  – CC 3.5 and higher also enable global memory accesses
    • Should not be used if a kernel reads and writes to the same addresses

• **Comparing to L1:**
  – Generally better for scattered reads than L1
    • Caching is at 32 B granularity (L1, when caching operates at 128 B granularity)
    • Does not require replay for multiple transactions (L1 does)
  – Higher latency than L1 reads, also tends to increase register use

• **Aggregate 48 KB per SM: 4 12-KB caches**
  – One 12-KB cache per scheduler
    • Warps assigned to a scheduler refer to only that cache
  – Caches are not coherent – data replication is possible
Read-Only Cache Operation

- Always attempts to hit
- Transaction size: 32 B queries
- Warp addresses are converted to queries 4 threads at a time
  - Thus a minimum of 8 queries per warp
  - If data within a 32-B segment is needed by multiple threads in a warp, segment misses at most once
- Additional functionality for texture objects
  - Interpolation, clamping, type conversion
Read-Only Cache Operation

addresses from a warp

1st Query
Read-Only Cache Operation

1st Query

addresses from a warp

2nd Query

addresses from a warp
Read-Only Cache Operation

addresses from a warp

1st Query

0  32  64  96  128  160  192  224  256  288  320  352  384  416  448
Read-Only Cache Operation

addresses from a warp

1st Query

addresses from a warp

2nd and 3rd Queries
Read-Only Cache Operation

1st Query

addresses from a warp

2nd and 3rd Queries

addresses from a warp

Note this segment was already requested in the 1st query: cache hit, no redundant requests to L2
Accessing GMEM via Read-Only Cache

- Compiler must know that addresses read are not also written by the same kernel
- Two ways to achieve this
  - Intrinsic: `__ldg()`
  - Qualify the pointers to the kernel
    - All pointers: `__restrict__`
    - Pointers you’d like to dereference via read-only cache: `const __restrict__`
    - May not be sufficient if kernel passes these pointers to functions
Accessing GMEM via Read-Only Cache

- Compiler must know that addresses read are not also written by the same kernel
- Two ways to achieve this
  - Intrinsic: \texttt{__ldg()}  
  - Qualify the pointers to the data
    - All pointers: \texttt{__restrict__}
    - Pointers you’d like to access: \texttt{__restrict__}
    - May not be sufficient

```
__global__ void kernel( int *output, int *input )
{
    ...
    output[idx] = ... + __ldg( &input[idx] );
}
```
Accessing GMEM via Read-Only Cache

- Compiler must know that addresses read are not also written by the same kernel

- Two ways to achieve this
  - Intrinsic: `__ldg()`
  - Qualify the pointers to the pointers you’re targeting:
    - All pointers: `__restrict__`
    - Pointers you’d like to restrict: `__restrict__`
    - May not be sufficient

```c
__global__ void kernel( int* __restrict__ output, 
                        const int* __restrict__ input ) 
{
    ... 
    output[idx] = ... + input[idx];
}
```
Thank you.

- Hendrik Lensch, Robert Strzodka
- NVIDIA:
  Paulius Micikevicius, Cliff Woolley, Lars Nyland, Justin Luitjens