Global Memory Usage and Strategy

GPU Computing Webinar 7/12/2011

Dr. Justin Luitjens, NVIDIA Corporation
Why Focus on Global Memory Accesses?

- GPU’s have many processing cores (upwards of 500)
  - Achieving high throughput depends on keeping these cores fed with data
- Most applications tend to be bandwidth bound
- Most data access begins in global memory
- Maximizing global memory bandwidth is a fundamental optimization
  - If you don’t get this correct other optimizations will likely be insignificant
Launch Configuration

Global memory Instructions
- Instructions are issued in order
- A thread stalls when one of the operands isn’t ready
- Latency is hidden by switching warps (32 threads)
  - GMEM latency: 400-800 cycles
  - Need enough threads to hide latency

How many threads/threadblocks to launch?
- Number of threads needed depends on the access pattern and word size
- Need enough memory transactions in flight to saturate the bus
- Increase transactions by having
  - Independent loads and stores from the same thread
  - Loads and stores from different threads (more threads)
  - Larger word sizes (float2 is twice the transactions of float, for example)
Maximizing Memory Throughput

- Increment of an array of 64M elements
  - Two accesses per thread (load then store)
  - The two accesses are dependent, so really 1 access per thread at a time
- Tesla C2050, ECC on, theoretical bandwidth: ~120 GB/s

Several independent smaller accesses have the same effect as one larger one.
For example:
Four 32-bit $\approx$ one 128-bit
Launch Configuration: Summary

- Need enough total threads to keep GPU busy
  - Typically, you’d like 512+ threads per SM
  - More if processing one fp32 element per thread
  - Of course, exceptions exist

Threadblock configuration
- Threads per block should be a multiple of warp size (32)
- SM can concurrently execute up to 8 threadblocks
  - Really small threadblocks prevent achieving good occupancy
  - Really large threadblocks are less flexible
  - I generally use 128-256 threads/block, but use whatever is best for the application

For more details:
- Vasily Volkov’s GTC2010 talk “Better Performance at Lower Occupancy”
Global Memory Access Patterns
Fermi Memory Hierarchy

- **Local storage (on-chip)**
  - Each thread has own local storage
  - Mostly registers (managed by the compiler)

- **Shared memory / L1 (on-chip)**
  - Program configurable: 16KB shared / 48 KB L1 OR 48KB shared / 16KB L1
  - Shared memory is accessible by the threads in the same threadblock
  - Very low latency
  - Very high throughput: 1+ TB/s aggregate

- **L2 (off-chip)**
  - All accesses to global memory go through L2, including copies to/from CPU host

- **Global memory (off-chip)**
  - Accessible by all threads
  - Higher latency (400-800 cycles)
  - Throughput: up to 177 GB/s
Fermi Memory Hierarchy

SM-0
- Registers
- L1
- SMEM

SM-1
- Registers
- L1
- SMEM

SM-N
- Registers
- L1
- SMEM

L2

Global Memory
Load Operations

Memory operations are issued per warp (32 threads)
- Just like all other instructions
- Prior to Fermi, memory issues were per half-warp

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

- Addresses from a warp ("thread-vector") are converted into line requests
- Line sizes: 32B and 128B
- Goal is to maximally utilize the bytes in these lines

addresses from a warp are within cache line

32  64  96  128  160  192  224  256  288  320  352  384
2D Array Access Pattern (row major)

float A[N][32];
...
A[threadIdx.x][0]=...;
A[threadIdx.x][1]=...;
...

1 thread per row

Uncoalesced access pattern
- Elements read in on first SIMT access: 0, 32, 64, ...
- Elements read in on second SIMT access: 1, 33, 65, ...
- Extra data will be transferred in order to fill the cache line size

Generally the most natural access pattern for a port of a C/C++ code!
Transposed 2D Array Access Pattern

```c
float A[32][N];
...
A[0][threadIdx.x]=...;
A[1][threadIdx.x]=...;
...
```

Elements read in on first SIMT access: 0, 1, 2, ..., 31
Elements read in on second SIMT access: 32, 33, ..., 63

Minimizes transactions and total bytes transferred
Array of Structures vs Structure of Arrays

An array of structures behaves like row major accesses
- struct Point { double x; double y; double z; double w; } A[N];
- ...
- A[threadIdx.x].x = …

A structure of arrays behaves like column major
- struct PointList{double *x; double *y; double *z; double *w;} A;
- ...
- A.x[threadIdx.x] = …
Fermi GMEM Operations

Two types of loads:

- **Caching**
  - Default mode (can also compile \texttt{--Xptxas --dlcm=ca} option to nvcc)
  - Attempts to hit in L1, then L2, then GMEM
  - Load granularity is \textbf{128-bytes}

- **Non-caching**
  - Compile with \texttt{--Xptxas --dlcm=cg} option to nvcc
  - Skip L1, Attempts to hit in L2, then GMEM
    - Do not hit in L1, invalidate the line if it’s in L1 already
  - Load granularity is \textbf{32-bytes}

Stores:

- Invalidate L1, write-back for L2
Load Caching and L1 Size

Non-caching loads can improve perf when:
- Loading scattered words or only a part of a warp issues a load
  - Benefit: transaction is smaller, so useful payload is a larger percentage
- Loading halos, for example
- Spilling registers (reduce line fighting with spillage)

Large L1 can improve perf when:
- Spilling registers (more lines so fewer evictions)
- Some misaligned, strided access patterns
- 16-KB L1 / 48-KB smem OR 48-KB L1 / 16-KB smem
  - cudaFuncSetCacheConfig(MyKernel, cudaFuncCachePreferShared);
  - cudaFuncSetCacheConfig(MyKernel, cudaFuncCachePreferL1);

How to use:
- Just try a 2x2 experiment matrix: \{CA,CG\} x \{48-L1, 16-L1\}
- Keep the best combination - same as you would with any HW managed cache, including CPUs
Caching Load

- Warp requests 32 aligned, consecutive 4-byte words
- Addresses fall within 1 cache-line
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: **100%**
  - Transactions: **1**

addresses from a warp

Memory addresses
Non-caching Load

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

addresses from a warp

Memory addresses

0 32 64 96 128 160 192 224 256 288 320 352 384 416 44
Caching Load

- Warp requests 32 aligned, permuted 4-byte words
- Addresses fall within 1 cache-line
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 100%
  - Transactions: 1

Addresses from a warp...

Memory addresses
Non-caching Load

- Warp requests 32 aligned, permuted 4-byte words
- Addresses fall within 4 segments
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 100%
  - Transactions: 4

addresses from a warp

<table>
<thead>
<tr>
<th>0</th>
<th>32</th>
<th>64</th>
<th>96</th>
<th>128</th>
<th>160</th>
<th>192</th>
<th>224</th>
<th>256</th>
<th>288</th>
<th>320</th>
<th>352</th>
<th>384</th>
<th>416</th>
<th>44</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory addresses</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Warp requests 32 misaligned, consecutive 4-byte words
Addresses fall within 2 cache-lines
- Warp needs 128 bytes
- 256 bytes move across the bus on misses
- Bus utilization: 50%
- Transactions: 2

addresses from a warp

Memory addresses
Non-caching Load

- 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

...
All threads in a warp request the same 4-byte word
Addresses fall within a single cache-line
  - Warp needs 4 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 3.125%
Non-caching Load

- 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 on a miss
  - Bus utilization: 12.5%
Caching Load

- Warp requests 32 scattered 4-byte words
- Addresses fall within $N$ cache-lines
  - Warp needs 128 bytes
  - $N \times 128$ bytes move across the bus on a miss
- Bus utilization: $\frac{128}{N \times 128}$
Non-caching Load

- Warp requests 32 scattered 4-byte words
- Addresses fall within $N$ segments
  - Warp needs 128 bytes
  - $N \times 32$ bytes move across the bus on a miss
  - Bus utilization: $\frac{128}{(N \times 32)}$

Memory addresses:
- Addresses from a warp
GMEM Optimization Guidelines

- Strive for perfect coalescing per warp
  - Align starting address (may require padding)
  - A warp should access within a contiguous region
  - Data structure, Data structure, Data structure
    - Using transpose your data so that it is a structure of arrays
- Have enough concurrent accesses to saturate the bus
  - Launch enough threads to maximize throughput
    - Latency is hidden by switching warps (32 threads)
  - Process several elements per thread
    - Multiple loads get pipelined
- Try L1 and caching configurations to see which one works best
  - Caching vs non-caching loads (compiler option)
  - 16KB vs 48KB L1 (CUDA call)
Questions?