Efficiently Enforcing Strong Memory Ordering in GPUs

Abhayendra Singh*, Shaizeen Aga, Satish Narayanasamy

Google, University of Michigan, Ann Arbor

Dec 9, 2015

*author performed the work at the University of Michigan, Ann Arbor
Increasing communication between threads in GPGPU applications

More irregular applications run on GPUs
  data-dependent,
  higher communication

TreeBuilding kernel in barneshut
(Burtscher et al., IISWC’12)
Heterogeneous systems will have more fine-grained communication

Fine-grain communication between CPU and GPU
- Unified virtual memory
- Cache coherence [Power et al., MICRO’13]
Heterogeneous systems will have more fine-grained communication

Fine-grain communication between CPU and GPU
  Unified virtual memory
  Cache coherence [Power et al., MICRO’13]

OpenCL supports fine-grain sharing

More irregularity in applications
Memory Consistency Model

Defines rules that a programmer can use to reason about a parallel execution
Memory Consistency Model

Defines rules that a programmer can use to reason about a parallel execution

Sequential Consistency (SC)

“program-order”

```
ptr = NULL; done = false
```

Producer

```
a: ptr = alloc()
b: done = true
```

Consumer

```
c: if (done)
d: rl = ptr->x
```
Memory Consistency Model

Defines rules that a programmer can use to reason about a parallel execution.

Sequential Consistency (SC)

“program-order” +

“atomic memory”

\[
\text{ptr} = \text{NULL}; \quad \text{done} = \text{false}
\]

Producer

\[\begin{align*}
\text{a: } & \text{ptr} = \text{alloc()} \\
\text{b: } & \text{done} = \text{true}
\end{align*}\]

Consumer

\[\begin{align*}
\text{c: } & \text{if (done)} \\
\text{d: } & \text{r1 = ptr->x}
\end{align*}\]
Data-race-free-0 (DRF-0) Memory Model

C++, Java
OpenCL, CUDA
Heterogeneous-race-free (HRF) (Hower et al., ASPLOS’14)
Data-race-free-0 (DRF-0) Memory Model

C++, Java

OpenCL, CUDA
  Heterogeneous-race-free (HRF) (Hower et al., ASPLOS’14)

SC if data-race-free
  Programmers annotate synchronization variables

\[\text{ptr} = \text{NULL}; \quad \text{atomic} \quad \text{done} = \text{false}\]

**Producer**
- a: \text{ptr} = \text{alloc()} \\
- b: \text{done} = \text{true}

**Consumer**
- c: if (\text{done}) \\
- d: \text{r1} = \text{ptr} - > x
Data-race-free-0 (DRF-0) Memory Model

C++, Java
OpenCL, CUDA
Heterogeneous-race-free (HRF) (Hower et al. ASPLOS’14)

SC if data-race-free
Programmers annotate synchronization variables
Compiler and runtime guarantee total order on synchronization operations

```c
ptr = NULL; atomic done = false
```

Producer
a: `ptr = alloc()`
b: `done = true`

Consumer
c: if (done)
d: `rl = ptr->x`

Data-race-free-0 (DRF-0) Memory Model

C++, Java

OpenCL, CUDA

Heterogeneous-race-free (HRF) (Hower et al., ASPLOS’14)

SC if data-race-free

Programmers annotate synchronization variables

Compiler and runtime guarantee total order on synchronization operations

```
ptr = NULL; done = false
```

Producer

\begin{align*}
a: & \quad \text{ptr} = \text{alloc}() \\
b: & \quad \text{done} = \text{true}
\end{align*}

Consumer

\begin{align*}
c: & \quad \text{if} \ (\text{done}) \\
d: & \quad r1 = \text{ptr} \rightarrow x
\end{align*}

reordering could lead to ptr being NULL
Data-race-free-0 (DRF-0) Memory Model

C++, Java

OpenCL, CUDA
  Heterogeneous-race-free (HRF) (Hower et al., ASPLOS’14)

SC if data-race-free
  Programmers annotate synchronization variables
  Compiler and runtime guarantee total order on synchronization operations

Undefined semantics for programs with a data-race
Documented data-races in GPGPU programs

Image source: [Alglave et al., ASPLOS 2015]

Bug: a data-race in code for dynamic load balancing
[Tyler Sorensen, MS thesis, 2014]

Other data-races:

N-body simulation [Betts et al., OOPSLA 2012]
RadixSort [Li et al., PPoPP 2012]
Efficient Synchronization Primitives for GPUs [Tyler Sorensen, MS thesis, 2014]
Is there a motivation for DRF-0 over SC?

Performance of DRF-0 better than SC?

*Very little* for CPUs

IEEE Computer’98, PACT’02, ISCA’12

Is there a performance justification for DRF-0 (or TSO) over SC in GPUs?
Goals

Identify sources of SC violation in GPUs

Understand overhead of various memory ordering constraints in GPUs
  DRF-0, TSO, SC

Bridge the gap between SC and DRF-0
  Access-type aware GPU architecture
How can GPU violate SC?

Instructions are executed in-order
How can GPU violate SC?

Instructions are executed in-order

But, can complete out-of-order

- Caching at L1
- Reordering in interconnect
- Partitioned address space
How can GPU violate SC?

Instructions are executed in-order

But, can complete out-of-order

- Caching at L1
- Reordering in interconnect
- Partitioned address space

<table>
<thead>
<tr>
<th>producer</th>
<th>consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>① ptr = alloc()</td>
<td>③ if (done)</td>
</tr>
<tr>
<td>② done = true</td>
<td>④ r1 = ptr-&gt;x</td>
</tr>
</tbody>
</table>
How can GPU violate SC?

- Instructions are executed in-order

But, can complete out-of-order
  - Caching at L1
  - Reordering in interconnect
  - Partitioned address space

<table>
<thead>
<tr>
<th>producer</th>
<th>consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>ptr = alloc()</td>
<td>if (done)</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>done = true</td>
<td>r1 = ptr-&gt;x</td>
</tr>
</tbody>
</table>

- Red: cache miss
- Green: cache hit
How can GPU violate SC?

Instructions are executed in-order

But, can complete out-of-order
  – Caching at L1
  – Reordering in interconnect
  – Partitioned address space

Producer
1. \( \text{ptr} = \text{alloc()} \)
2. \( \text{done} = \text{true} \)

Consumer
3. if (done)
4. \( \text{r1} = \text{ptr} \rightarrow x \)

- cache miss
- cache hit
How can GPU violate SC?

Instructions are executed in-order

But, can complete out-of-order
– Caching at L1
– Reordering in interconnect
– Partitioned address space

producer

1. ptr = alloc()
2. done = true

consumer

3. if (done)
4. r1 = ptr->x

cache miss

cache hit
**How can GPU violate SC?**

Instructions are executed in-order

But, can complete out-of-order
- Caching at L1
- Reordering in interconnect
- Partitioned address space

<table>
<thead>
<tr>
<th></th>
<th>producer</th>
<th>consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>ptr = alloc()</code></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td><code>done = true</code></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td><code>if (done)</code></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td><code>r1 = ptr-&gt;x</code></td>
</tr>
</tbody>
</table>

- **cache miss**
- **cache hit**
How can GPU violate SC?

Instructions are executed in-order

But, can complete out-of-order
- Caching at L1
- Reordering in interconnect
- Partitioned address space

⇒ Can violate SC

```
ptr = alloc()
done = true
r1 = ptr->x
```

If (done)

- cache miss
- cache hit

SC violation
Roadmap

Identify sources of SC violation

Understand overhead of various memory ordering constraints in GPUs
   DRF-0, TSO, SC

Bridge the gap between SC and DRF-0
   Access-type aware GPU architecture
Fences for various memory models

DRF-0

fences only for synchronization

SC

any shared or global access behaves like a fence
Naïvely Enforcing Fence Constraints

Delay a warp till non-local memory accesses preceding a fence are complete
Naïvely Enforcing Fence Constraints

Delay a warp till non-local memory accesses preceding a fence are complete

GPU extension:

Two counters per warp track its pending global loads and stores

No need to track pending shared memory accesses

<table>
<thead>
<tr>
<th>warp id</th>
<th>pending loads</th>
<th>pending stores</th>
</tr>
</thead>
<tbody>
<tr>
<td>w0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Experimental Methodology

**Simulator:** GPGPU-sim v3.2.1
- extended with Ruby memory hierarchy
- 16 SMs, crossbar interconnect

**L1 Cache Coherence protocol**
- MESI for write-back
- Valid/Invalid for write-through

**Benchmarks**
- applications from Rodinia, Polybench benchmark suite
- Applications used in GPU coherence [Singh et al., HPCA’13]
18 out of 22 applications incur insignificant SC overhead
Warp-level-parallelism (WLP) masks SC overhead

Warp-0

- Cache miss
- Cache Hit
Warp-level-parallelism (WLP) masks SC overhead
Warp-level-parallelism (WLP) masks SC overhead

SC can exploit inter-warp MLP

Cache miss  
Cache Hit
Warp-level-parallelism (WLP) masks SC overhead

SC can exploit inter-warp MLP

Adequate WLP => Low SC overhead
Warp-level-parallelism (WLP) masks SC overhead

Execution time normalized to DRF-0
(benchmark: guassian)

SC can exploit inter-warp MLP

Adequate WLP => Low SC overhead
Higher SC overhead in apps where intra-warp MLP is important

Need for intra-warp MLP
App has fewer warps
Want fewer warps to avoid cache thrashing
Higher SC overhead in apps where intra-warp MLP is important

Need for intra-warp MLP
App has fewer warps
Want fewer warps to avoid cache thrashing

Cache miss  Cache Hit
Higher SC overhead in apps where intra-warp MLP is important

Need for intra-warp MLP
App has fewer warps
Want fewer warps to avoid cache thrashing

In-order execution limits the ability to exploit intra-warp MLP in DRF-0
Higher SC overhead in apps where intra-warp MLP is important

Unlike DRF-0, SC cannot exploit intra-warp MLP

Need for intra-warp MLP
   App has fewer warps
   Want fewer warps to avoid cache thrashing

In-order execution limits the ability to exploit intra-warp MLP in DRF-0
4 out of 22 applications exhibit significant SC overhead

Execution time normalized to DRF-0

Reason: Unlike DRF-0, SC cannot exploit intra-warp MLP
TSO is not suitable for GPUs

Applications with small performance overhead

Execution time normalized to baseline (DRF-0)

Applications with significant performance overhead

TSO does not offer much performance or programmability advantage over SC
Roadmap

How GPU optimizations can violate memory ordering constraints?

Understand overhead of various memory ordering constraints in GPUs
  DRF-0, TSO, SC

Bridge the gap between SC and DRF-0
  Access-type aware GPU architecture
Access-type Aware Optimization for GPU

Relax ordering constraint for safe accesses
Accesses to thread-private or read-only location are safe
(Shasha & Snir, TOPLAS’88, Singh et al., ISCA’12)

Thread-level classification is prohibitively expensive
Classify accesses as unsafe or SM-safe
Type-aware SC Extensions to Baseline GPU

1. Classify memory accesses
1. Classify memory accesses
Type-aware SC Extensions to Baseline GPU

1. Classify memory accesses
Type-aware SC Extensions to Baseline GPU

1. Classify memory accesses
Type-aware SC Extensions to Baseline GPU

1. Classify memory accesses
2. Relax ordering constraints for SM-safe accesses
1. Classify memory accesses

2. Relax ordering constraints for SM-safe accesses

Problem: Ensuring ordering among conflicting accesses to an SM-safe location within an SM

✓ See details in the paper
Type-aware SC incurs only small performance overhead

Applications with significant performance overhead

3mm  |  fdtd-2d  |  gemm  |  gramschm
---   |          |       |       
3     | 2         | 3     | 1     

Proposed design is able to exploit intra-warp MLP for SM-safe accesses
Future research directions

Build SC-preserving GPU compiler

Overhead SC-preserving LLVM compiler for C++: ~2%  [Marino et al., PLDI’11]

Language level memory model

- GPU Compiler
- GPU hardware
Conclusion

Quantified performance overhead of various memory models in GPUs

TSO is unattractive for GPUs: No performance or programmability benefits over SC

Performance gap between SC and DRF-0 is insignificant for most applications

Access type aware optimization bridges the gap in remaining applications
Questions?