1 Introduction

Motivation We are researchers interested in what kinds of behaviours are allowed on GPU architectures with respect to shared memory consistency. Our motivations for this are:

- to provide GPU developers with clear guidelines on the behaviours they should expect on a chip, thus allowing them to design robust and efficient algorithms,
- to help designers of GPU programming models (such as CUDA and OpenCL) ensure that it is possible to implement language-level memory models accurately and efficiently on modern hardware, and
- to lay firm foundations for formal and semi-formal analysis tools for GPU software.

Contributions of this document In previous work, we have developed a GPU extension to the litmus and diy tools (see http://diy.inria.fr) that allows us to test the memory behaviour of Nvidia GPUs, with an emphasis on inter-thread interactions.

In this document, we test the GPU spin-lock mutex given in the book CUDA by Example by Jason Sanders and Edward Kandrot [2]. A similar implementation is given by Jeff Stuart and John Owens in their paper Efficient Synchronization Primitives for GPUs [3]. We show that these implementations do not satisfy what is generally considered to be the correct specification for a mutex.

Specifically, we show that a critical section may read data that is stale w.r.t. the previous critical section for inter-block interactions. The problem is caused by weak memory orderings at the hardware level. We then show that the addition of memory fences ensures the expected behaviour. We are currently unsure whether the weak behaviour seen without the fences (which we discuss in Section 6.1 and 6.2) is intended
by NVIDIA hardware or not, especially as one of the examples comes from an NVIDIA publication (*CUDA by Example*).

We document this behaviour in terms of short litmus tests and the results of running the tests using our GPU extensions to litmus and diy. The tests are written in PTX assembly. We have verified, using cuobjdump, that the PTX assembler does not reorder or otherwise optimize the tests. We ran the tests on three GPUs, a Tesla C2075 (Fermi), a GeForce GTX 660 (Kepler), and a Tesla K20c (Kepler). The versions of the CUDA compilation tools used were release 5.5 V5.5.0 for the Tesla chips and release 5.0 V0.2.1221 for the GTX (as given by `nvcc --version`).

## 2 Notation and Terminology

This section contains a detailed description of the notation and terminology we use.

**Test specifications** A *test specification*, such as the one shown in Figure 1, consists of several columns, each headed by a global thread ID. Each thread scheduled on the GPU has a unique global thread ID. In practice, a global thread ID can be computed using a combination of the built-in CUDA values `blockIdx.x`, `blockdim.x`, and `threadIdx.x`. However, in our examples, we use symbolic global thread IDs, such as `P0` and `P1`, for ease of presentation.

Under each global thread ID is a *program*: the sequence of instructions executed by that thread. Ordinarily, every thread executes the same program, but we can arrange each thread to execute a different program by writing conditional code, having threads branch to different parts of the program based on their global thread IDs.

```
P0: st.s32.cg [x], 1 | ld.s32.cg r1, [y] | st.s32.cg [y], 1 | ld.s32.cg r2, [x] |

exists (1:r1=1 \ 1:r2=0)
```

*Figure 1: A test specification for MP (message passing)*

Consider the example in Figure 1, which implements a message passing idiom. (We do not analyse this test in this section; it is given for explanatory reasons only.) Here, the global thread IDs are `P0` and `P1`.

We assume each kernel is launched with a sufficient number of blocks and threads such that each program in the test will eventually be executed on the GPU. In the example, the two store instructions will be executed by thread `P0`, and the two load instructions by thread `P1`. 
Notation for registers and memory accesses  We deviate from concrete PTX assembly in that we allow direct stores of immediate values to memory (e.g. \texttt{st.s32.cg [x],1} as seen in \texttt{P0}'s program in Figure 1), instead of moving the value first to a register (via \texttt{mov}), and then storing the register contents to memory. Thread local registers are denoted by \texttt{rn} where \texttt{n} is a non-negative integer. Locations are given by single lower-case letters, e.g. \texttt{x}, \texttt{y}, \texttt{z}. In Figure 1, we have two memory locations \texttt{x} and \texttt{y}, and thread \texttt{P1} writes to registers \texttt{r1} and \texttt{r2}.

Initialization of memory locations  In each test, the initial value of each memory location will be explicitly provided.

Placement of threads  The placement of threads in the GPU thread hierarchy will be explicitly mentioned when we describe each test. Regarding Figure 1, there may be different considerations depending on whether \texttt{P0} and \texttt{P1} are in the same CTA or in different CTAs. We shall also describe which memory space each location belongs to (e.g. global memory, shared memory, etc.). At this point we are only interested in intra-kernel interactions.

Test outcomes  Questions about the executions of a test are given as a constraint on test outcomes: the final values of registers or memory locations. In Figure 1, we write:

\[
\text{exists (1:r1=1 \land 1:r2=0)}
\]

to ask if it is possible to observe \texttt{P1}'s private registers \texttt{r1} and \texttt{r2} at values \texttt{1} and \texttt{0} respectively in the final state of the GPU after having executed all testing threads. In general, registers in the final constraint are denoted \texttt{n:reg}, where \texttt{reg} is a register and \texttt{n} is the ID of the thread the register belongs to.

3 Thread Hierarchies and Memory Operations Used During Testing

In this document, we only consider two simple thread hierarchy configurations. We refer to them as \texttt{same-cta} and \texttt{diff-cta} and they are defined as follows:

- \texttt{same-cta}: all programs in the test are mapped to threads in different warps, but in the same CTA.
- \texttt{diff-cta}: all programs in the test are mapped to different CTAs but the same kernel.
Furthermore, all memory access instructions use the same cache operator that accesses the L2 cache (.cg) and target global memory. The PTX ISA describes L1 cache operators (.ca, .wb) as being incoherent across blocks [1, pp. 118–120]. Furthermore, our ongoing work shows\(^1\) that the L1 cache operators cannot be used reliably for inter-CTA interactions. We emphasise that the issues described in this document are not a symptom of the use of incoherent cache operators.

4 CUDA to PTX Mappings

We briefly describe the relevant instruction mappings from CUDA to PTX for these examples. We have verified these instruction mappings on CUDA release 5.5 V5.5.0. We have taken precautions to ensure loads and stores are compiled with the L2 cache operator (see Section 3). This can be accomplished by setting the compiler flags: `-Xptxas -dlcm= cg -Xptxas -dscm=cg`.

<table>
<thead>
<tr>
<th>CUDA Instruction</th>
<th>PTX Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomicCAS</td>
<td>atom.cas.b32</td>
</tr>
<tr>
<td>atomicExch</td>
<td>atom.exch.b32</td>
</tr>
<tr>
<td><code>__threadfence</code></td>
<td>membar.gl</td>
</tr>
<tr>
<td><code>__threadfence_block</code></td>
<td>membar.cta</td>
</tr>
<tr>
<td>store global int</td>
<td>st.cg.u32</td>
</tr>
<tr>
<td>load global int</td>
<td>ld.cg.u32</td>
</tr>
<tr>
<td>control flow (e.g. while, if)</td>
<td>setp with predicate (e.g. @r1)</td>
</tr>
</tbody>
</table>

The focus of this document is to show the behaviour of these examples at the hardware level; as such, the effects of potential compiler optimisations are outside the scope of this document.

5 GPU Mutexes

Here we briefly describe the GPU mutex implementations that inspired this testing.

5.1 CUDA by Example

_CUDA by Example_ [2] presents a mutex implementation for combining block-local partial sums. The mutex implementation is a simple CAS spin lock with an atomic exchange release. We reproduce a simplified version of the lock and unlock functions

\(^1\)Write-up in progress
here for reference. Note that we have repaired the `atomicExch` with the correct value as given in the official errata for the book.\(^2\)

```c
__device__ int mutex;
__device__ void lock( void ) {
    while( atomicCAS( mutex, 0, 1 ) != 0 );
}
__device__ void unlock( void ) {
    atomicExch( mutex, 0 );
}
```

The locks are then used to update a global value `c` with the block-local partial sums located in `cacheIndex[0]`. Only one thread per block executes this code, thus removing the potential for warp-synchronous spin lock deadlocks.

```c
//cacheIndex is equal to threadIdx.x
if (cacheIndex == 0) {
    lock.lock();
    *c += cache[0];
    lock.unlock();
}
```

The book does not explicitly mention memory consistency issues, but the following paragraph suggests that the behaviour typically expected from a lock can be obtained by only using atomic operations. For context, it is explaining why `unlock` must be an atomic exchange rather than simply a store.

```
Atomic transactions and generic global memory operations follow different paths through the GPU. Using both atomics and standard global memory operations could therefore lead to an `unlock()` seeming out of sync with a subsequent attempt to `lock()` the mutex. The behavior would still be functionally correct, but to ensure consistently intuitive behavior from the application’s perspective, it’s best to use the same pathway for all accesses to the mutex. Because we’re required to use an atomic to lock the resource, we have chosen to also use an atomic to unlock the resource.
```

\(^2\)https://developer.nvidia.com/cuda-example-errata-page
5.2 Efficient Synchronization Primitives for GPUs

In their paper, Jeff Stuart and John Owens provide synchronization primitives for GPUs [3]. They include a simple spin lock that is similar to the one presented by CUDA by Example, except that they use atomic exchange instead of CAS for the locking function. They continue to discuss how to optimize the mutex functions by reducing contention for a resource using a method they refer to as a backoff strategy. This method does not introduce any additional memory ordering operations (e.g., memory fences).

Stuart and Owens explicitly make the assumption that an atomic exchange has the semantics of a fence by stating:

Also, we use atomicExch() instead of a volatile store and threadfence() because the atomic queue has predictable behavior; threadfence() does not (i.e. it can vary greatly in execution time if other memory operations are pending).

5.3 PTX ISA

We were unable to locate unambiguous justifications for the above assumptions in any NVIDIA documentation (CUDA or PTX). The following paragraph from the PTX ISA [1, pp. 166–167] may be related, but seems to be restricted to atomicity and single address interactions and does not account for memory accesses inside the critical section:

Atomic operations on shared memory locations do not guarantee atomicity with respect to normal store instructions to the same address. It is the programmer’s responsibility to guarantee correctness of programs that use shared memory atomic instructions, e.g., by inserting barriers between normal stores and atomic operations to a common address, or by using atom.exch to store to locations accessed by other atomic operations.

6 Testing Results

Now we describe the tests and their outcomes; we show a summary of the results shown in two tables, one for each thread hierarchy configuration. For each test, the table states how often the tested behaviour (specified by the final condition) occurred when the test was executed 100,000 times.

We have developed heuristics which stress the GPU in various ways to encourage weak memory behaviours to appear during the testing. For instance, we create noise making threads that stress the GPU memory system by constantly reading and writing
to non-testing memory locations. Another heuristic is the randomization of testing parameters, such as which warp or CTA a thread is in. New random parameters are picked for each of the 100,000 runs of each test.

Naturally, there is no guarantee that our heuristics are sufficient to make all behaviours show up. Also, the frequency of a certain outcome may change when new, different heuristics are used during testing. The numbers in all of our tables should be considered with these points in mind.
6.1 CAS Spin Lock

Test specification (CAS-SL)

Initial values: \( y = 1, x = 0; \)

\[
P0 \quad | \quad P1 \\
\text{st.u32.cg} [x], 1 \quad | \quad \text{atom.cas.b32} \ r0,[y],0,1 \\
\text{atom.exch.b32} \ r0,[y],0 \quad | \quad \text{setp.eq.u32} \ r1, r0, 0 \\
\quad | \quad \text{membar.gl} \\
\quad | \quad \text{@r1 ld.u32.cg} \ r2,[x] \\
\text{exists} \ (1:r0=0 \ \land \ 1:r2=0)
\]

Description  This test (named CAS-SL) describes two threads interacting via a CAS spin lock. The \( y \) memory location is the mutex and \( x \) is global data accessed in the critical section. The test begins in a state where \( P0 \) has the mutex. \( P0 \) stores a value to \( x \) and then releases the mutex via an atomic exchange. \( P1 \) attempts to acquire the lock via a CAS instruction, then checks to see if the lock was acquired successfully via the setp command. If the lock was acquired, i.e. \( r0 == 0 \), \( P1 \) attempts to read the global data in \( x \). This is enforced using PTX predicated execution; that is, instructions annotated with \( \text{@r1} \) will only execute if the setp command was satisfied. The final constraint describes an execution where \( P1 \) acquires the lock \((1:r0 = 0)\) and does not see the updated value in \( x \)(1:r2 = 0).

Our results show that in inter-block interactions, we can indeed see stale values. However, when we place the appropriate fence (membar.gl) between the accesses, we consistently observe the updated (i.e. intended) value. The modified test is named CAS-SL+membar.gls and its specification is given below.

Test specification (CAS-SL+membar.gls)

Initial values: \( y = 1, x = 0; \)

\[
P0 \quad | \quad P1 \\
\text{st.u32.cg} [x], 1 \quad | \quad \text{atom.cas.b32} \ r0,[y],0,1 \\
\text{membar.gl} \quad | \quad \text{setp.eq.u32} \ r1, r0, 0 \\
\text{atom.exch.b32} \ r0,[y],0 \quad | \quad \text{@r1 membar.gl} \\
\quad | \quad \text{@r1 ld.u32.cg} \ r2,[x] \\
\text{exists} \ (1:r0=0 \ \land \ 1:r2=0)
\]
**Our observations**  The table shows the frequency of test outcomes for variants of the CAS-SL test for three different chips. All tests were run 100,000 times. The weak behaviour was not observed for the thread hierarchy configuration **same-cta**.

<table>
<thead>
<tr>
<th>Thread hierarchy configuration</th>
<th>Test name</th>
<th>GTX 660</th>
<th>Tesla K20c</th>
<th>Tesla C2075</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>same-cta</strong></td>
<td>CAS-SL</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>diff-cta</strong></td>
<td>CAS-SL</td>
<td>25</td>
<td>1607</td>
<td>86</td>
</tr>
<tr>
<td></td>
<td>CAS-SL+membar.gls</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Relation to concrete example**  The tests in this section distill the locking behaviour in *CUDA by Example* discussed in Section 5.1 to a simple message passing idiom. If $P_1$ is able to see a stale value, then the total sum could be computed without considering $P_0$’s contribution; this will lead to an incorrect dot product result. The implementation in *CUDA by Example* has inter-block interactions and is lacking fence instructions which leaves the code vulnerable to this error.
6.2 Atomic Exchange Spin Lock

Test specification (EXCH-SL)

Initial values: \( y = 1, x = 0; \)

\[
P0 | P1
st.u32.cg [x], 1 | atom.exch.b32 r0,[y],1 ;
atom.exch.b32 r0,[y],0 | setp.eq.u32 r1, r0, 0 ;
| @r1 ld.u32.cg r2,[x] ;
\]

\[
\exists (1:r0=0 /\ 1:r2=0)
\]

Description This test (named EXCH-SL) describes two threads interacting via an atomic exchange spin lock. The description is very similar to the CAS-SL test described in Section 6.1. The final constraint describes an execution where \( P1 \) acquires the lock \((1:r0 = 0)\) and does not see the updated value in \( x (1:r2 = 0)\).

Our results show that in inter-block interactions, we can indeed see stale values. However, when we place the appropriate fence \( \text{membar.gl} \) between the accesses we consistently observe the updated (i.e. intended) value. We name the test with added \( \text{membar.gls} \) EXCH-SL+membar.gls and it’s test specification is given below:

EXCH-SL+membar.gls

Initial values: \( y = 1, x = 0; \)

\[
P0 | P1
st.u32.cg [x], 1 | atom.exch.b32 r0,[y],1 ;
membar.gl | setp.eq.u32 r1, r0, 0 ;
atom.exch.b32 r0,[y],0 | @r1 membar.gl ;
| @r1 ld.u32.cg r2,[x] ;
\]

\[
\exists (1:r0=0 /\ 1:r2=0)
\]

Our observations The table shows the frequency of test outcomes for variants of the EXCH-SL test for three different chips. All tests were run 100,000 times. The weak
behaviour was not observed for the thread hierarchy configuration **same-cta**.

<table>
<thead>
<tr>
<th>Thread hierarchy configuration</th>
<th>Test name</th>
<th>GTX 660</th>
<th>Kepler Tesla K20c</th>
<th>Fermi Tesla C2075</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>same-cta</strong></td>
<td>EXCH-SL</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>diff-cta</strong></td>
<td>EXCH-SL</td>
<td>33</td>
<td>1468</td>
<td>98</td>
</tr>
<tr>
<td></td>
<td>EXCH-SL+membar.gls</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Relation to concrete example** While the paper *Efficient Synchronization Primitives for GPUs* (discussed in Section 5.2) doesn’t provide concrete examples using the locking mechanisms, this test distills a simple locking message passing idiom one might implement. Traditionally, lock implementations have provided sufficient synchronization to ensure that critical sections observe the most recent values computed in previous critical sections; formally this property has been called *sequential consistency for data race free programs*. As seen in our results above, this is not the case in the inter-block use case. Although the paper makes no claims about formal synchronization properties, we feel that it may not have been intentional to allow such behaviours.

### 7 Summary and Additional Testing

We have shown empirically that GPU mutexes must use memory fences to ensure consistency between critical sections. We chose these two examples because of their availability and clarity, but we suspect that any GPU code that depends on inter-CTA communication without using fences will exhibit similar behaviours. The testing code we acquired these results with is available on demand. We would be interested in any feedback, comments, or additional resources that can be provided.

### References

