GPU Concurrency:
Weak Behaviours and Programming Assumptions

Jade Alglave\textsuperscript{1,2}  Mark Batty\textsuperscript{3}  Alastair F. Donaldson\textsuperscript{4}  Ganesh Gopalakrishnan\textsuperscript{5}  Jeroen Ketema\textsuperscript{4}  Daniel Poetzl\textsuperscript{6}  Tyler Sorensen\textsuperscript{1,5}  John Wickerson\textsuperscript{4}

\textsuperscript{1} University College London  \textsuperscript{2} Microsoft Research  \textsuperscript{3} University of Cambridge  \textsuperscript{4} Imperial College London  \textsuperscript{5} University of Utah  \textsuperscript{6} University of Oxford

Abstract
Concurrency is pervasive and perplexing, particularly on graphics processing units (GPUs). Current specifications of languages and hardware are inconclusive; thus programmers often rely on folklore assumptions when writing software.

To remedy this state of affairs, we conducted a large empirical study of the concurrent behaviour of deployed GPUs. Armed with litmus tests (i.e. short concurrent programs), we questioned the assumptions in programming guides and vendor documentation about the guarantees provided by hardware. We developed a tool to generate thousands of litmus tests and run them under stressful workloads. We observed a litany of previously elusive weak behaviours, and exposed folklore beliefs about GPU programming—often supported by official tutorials—as false.

As a way forward, we propose a model of Nvidia GPU hardware, which correctly models every behaviour witnessed in our experiments. The model is a variant of SPARC Relaxed Memory Order (RMO), structured following the GPU concurrency hierarchy.

Categories and Subject Descriptors: B.3.0 [Memory structures]: General

Keywords: memory consistency, GPU, Nvidia PTX, OpenCL, litmus testing, test generation, formal model

1. Introduction
GPUs have cemented their position in computer systems: no longer restricted to graphics, they appear in critical applications, e.g. [29]. Thus programming them correctly is crucial.

Yet GPU concurrency is poorly specified. The vendors’ documentation and programming guides suffer from significant omissions and ambiguities, which force programmers to rely on folklore assumptions when writing software.

To distinguish assumptions from ground truth, we questioned the hardware guarantees and the assumptions made in programming guides. Thus we conducted a large empirical study of deployed Nvidia and AMD GPUs (see Tab. 1).

<table>
<thead>
<tr>
<th>vendor</th>
<th>architecture</th>
<th>chip</th>
<th>short name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nvidia</td>
<td>Tesla</td>
<td>GTX 280</td>
<td>GTX 280</td>
<td>2008</td>
</tr>
<tr>
<td></td>
<td>Fermi</td>
<td>GTX 540m</td>
<td>GTX 5</td>
<td>2011</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tesla C2075</td>
<td>TesC</td>
<td>2011</td>
</tr>
<tr>
<td></td>
<td>Kepler</td>
<td>GTX 660</td>
<td>GTX 6</td>
<td>2012</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GTX Titan</td>
<td>Titan</td>
<td>2013</td>
</tr>
<tr>
<td></td>
<td>Maxwell</td>
<td>GTX 750</td>
<td>GTX 7</td>
<td>2014</td>
</tr>
<tr>
<td>AMD</td>
<td>TeraScale 2</td>
<td>Radeon HD 6570</td>
<td>HD6570</td>
<td>2011</td>
</tr>
<tr>
<td></td>
<td>Graphics Core Next (GCN) 1.0</td>
<td>Radeon HD 7970</td>
<td>HD7970</td>
<td>2012</td>
</tr>
</tbody>
</table>

Table 1: The GPU chips we tested

Our methodology relies on executing short programs (litmus tests), probing specific hardware behaviours [6, 7, 14, 17]. Central to the success of our method is a test harness: we run each test thousands of times under stressful workloads, to provoke the behaviour that the test characterises.

Our litmus tests uncovered weak GPU behaviours, similar to those of CPUs (e.g. IBM Power [6, 7]), which “no existing literature has been able to show how to trigger” and have been dismissed as “infinitesimally unlikely” to occur [19]. We observed weak behaviours on all the chips listed in Tab. 1 except the GTX 280; we henceforth omit this particular chip from our results tables. Moreover, our tests exposed as false several programming assumptions made in academic works [22, 42] and literature endorsed by vendors [26, 36, 38]. We summarise our findings in Tab. 2 and detail them in Sec. 3; we illustrate two key findings below.

7 In fairness to the authors of [19], we were unable to observe weak behaviours using our method on the Nvidia GTX 280 chip they used.
Weak behaviours The litmus test of Fig. 1 (written in Nvidia’s low level language PTX) tests for read-read coherence coRR violations. The left thread stores \( x \) into the location \( x \), which is in global memory and initialised to 0, and the right thread, which is in the same CTA (see Sec. 2.1), loads twice from \( x \). Read-read coherence violations occur for executions ending with register \( r1 \) holding 1 and register \( r2 \) holding 0. This behaviour seems to spark debate for CPUs: it is allowed by SPARC Relaxed Memory Order (RMO) [43, Chap. D.4], but is considered a bug on some ARM chips [12].

Yet on several Nvidia GPUs, we observed coRR violations (i.e. Fig. 2, 6, 10). For the CUDA programs, we extracted a snippet that was susceptible to weak memory behaviours and translated it to PTX by using the mapping in Tab. 5. We then compiled the PTX litmus test to machine code, and checked that the PTX assembler did not reorder or remove memory accesses (see Sec. 4.4). Executing the litmus test on a GPU thus reveals the hardware behaviour.

As a way forward, we propose a model of Nvidia GPU hardware. Our model is based on SPARC RMO, and is stratified according to the thread hierarchy found on GPUs. We validated it against 10930 litmus tests on the Nvidia chips of Tab. 1, each executed 100k times, to confirm that it accounts for every observed behaviour.

![Table 2: Summary of the issues revealed by our study](image)

<table>
<thead>
<tr>
<th>Affected</th>
<th>Litmus Tests</th>
<th>Comment</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nvidia</td>
<td>coRR</td>
<td>sparks debate for CPUs</td>
<td>3.1.1</td>
</tr>
<tr>
<td>Fermi/Kepler architectures</td>
<td>coRR</td>
<td>fences do not restore</td>
<td>3.1.2</td>
</tr>
<tr>
<td>PTX ISA [36]</td>
<td>coRR-L1-L2-L1</td>
<td>volatile documentation disagrees with testing</td>
<td>3.1.2</td>
</tr>
<tr>
<td>Computing Gems [26]</td>
<td>dbf-lb, dbf-lb</td>
<td>fenceless queue allows</td>
<td>3.2.1</td>
</tr>
<tr>
<td>Cuda by Example [38]</td>
<td>cas-sl</td>
<td>fenceless queue allows</td>
<td>3.2.2</td>
</tr>
<tr>
<td>Stuart-Owens lock [42]</td>
<td>exch-sl</td>
<td>fenceless queue allows</td>
<td>3.2.2</td>
</tr>
<tr>
<td>He-Yu lock [22]</td>
<td>sl-future</td>
<td>lock allows future values to be read</td>
<td>3.2.3</td>
</tr>
<tr>
<td>Cuda 5.5 [32]</td>
<td>coRR</td>
<td>compiler reorders volatile loads</td>
<td>4.4</td>
</tr>
<tr>
<td>GCN 1.0</td>
<td>mp</td>
<td>compiler removes fences between loads</td>
<td>3.1.2</td>
</tr>
<tr>
<td>TeraScale 2</td>
<td>dbf-lb</td>
<td>compiler reorders load and CAS</td>
<td>3.2.1</td>
</tr>
</tbody>
</table>

Contributions In essence, we present:

1. a framework for generating and running litmus tests to question memory consistency on GPU chips (see Sec. 4);
2. a set of incantations: heuristics for provoking weak behaviour during testing (see Sec. 4);
3. an extensive empirical evaluation across seven GPUs from Nvidia and AMD (see Tab. 1, Sec. 3 and Sec. 5);
4. details of ten correctness issues in GPU hardware, compilers and public software (see Tab. 2 and Sec. 3); and
5. a formal model of Nvidia GPUs, informed by our evaluation, providing a foundation on which to build more reliable chips, compilers and applications (see Sec. 5).

Online material We give our complete experimental reports online [1], along with extra examples and explanations.

2. Background on GPUs

A GPU (graphics processing unit) features streaming multiprocessors (SMs; compute units on AMD), each with multiple cores [36, Chap. 2–3] [34, App. G] [11, Chap. 1].
2.1 Execution hierarchy

Programs map to hardware in a hierarchical way. A thread (work-item in OpenCL) executes instructions on a core. A warp (wavefront on AMD) is a group of 32 threads (64 on AMD), which execute following the “single instruction multiple threads” model (SIMT). Thus threads in a warp execute in lock step, i.e. run the same code and share a program counter. A cooperative thread array (CTA; block in CUDA and work-group in OpenCL) consists of a configurable number of warps, all executing on the same SM. A grid (NDRange in OpenCL) can consist of millions of CTAs. A kernel refers to a GPU program executed by a grid.

We focus on thread interactions either in the same CTA but different warps, or in the same grid but different CTAs. We do not test inter-grid or inter-GPU interactions as we did not find any example using these features in the literature.

Additionally we do not test intra-warp interactions; this would require threads in the same warp to execute different instructions; several of our incantations (see Sec. 4) require that all threads in a warp execute the same instructions.

2.2 Memory hierarchy

Global memory is shared between all threads in a grid, and may be cached in L1 or L2 caches. The SMs each have their own L1, and share an L2. There is also one region of shared memory per SM, shared only by threads in the same CTA.

GPUs also provide read-only regions (e.g. CUDA constant and texture memory [34, Chap. 3.2.11]). We ignore these as they are uninteresting from a weak memory perspective: reads from a constant location all yield the same result.

2.3 Parallel Thread Execution (PTX) and OpenCL

To test hardware, we run assembly litmus tests. Nvidia’s assembly, SASS, is largely undocumented, except for a list of instructions [35, Chap. 4] which does not describe their semantics. Moreover, there is no openly available assembler from SASS to binary. The AMD TeraScale 2 and GCN 1.0 architectures use the Evergreen [9] and Southern Islands [10] instruction set architectures (ISAs), respectively. These ISAs are documented but assemblers are not openly available. Below we explain how we circumvent these challenges.

Nvidia: PTX For Nvidia chips, we write our tests in Nvidia’s Parallel Thread Execution (PTX) low-level intermediate language [36]. PTX abstracts over the ISAs of Nvidia GPUs. Sec. 4.4 explains how we relate our PTX tests to the hardware behaviours that we observe, using our optcheck tool based on Nvidia’s cuobjdump [35, Chap. 2]: we inspect the SASS code and check that it has not introduced reorderings w.r.t. the initial PTX code that would alter the intention of our tests.

Our formal model of PTX (see Sec. 5) includes the following instructions: loads (ld), stores (st), ALU operations (add, and), fences (membar), unconditional jumps (bra), setting a predicate register if two operands are equal (setp.eq), and predicated instructions that only execute if a predicate register is set (#p1 ...) or unset (#p1 ...). Fences are parameterised by a scope: membar.cta (resp. gl or sys) provides ordering within a CTA (resp. within the GPU or with the host). Other instructions bear a cache operator: for example, load instructions may be annotated with the cache operator .ca (resp. .cg) which specify that the load targets the L1 (resp. L2) cache. Several instructions bear a type specifier indicating their bit width and signedness [36, Chap. 5.2]. For brevity, we omit the type specifier in our examples and use the signed single word size (i.e. .s32) for all instructions.

Some of our examples use compare-and-swap (atom.cas), exchange (atom.exch), and volatile instructions (which inform the compiler that the value in memory “can be changed or used at any time by another thread” [34, p. 170] in CUDA, and “inhibit optimization” [36, p. 131] in PTX), but these instructions are not included in our model.

AMD: OpenCL AMD intermediate language (AMD IL) [8] is analogous to Nvidia PTX: but AMD does not provide compilation tools for it, so we cannot use the same approach as for Nvidia. To test AMD chips we write our tests in OpenCL, relying on the AMD OpenCL compiler to translate them into Evergreen and Southern Islands online [1]. To test AMD chips we write our tests in OpenCL, relying on the AMD OpenCL compiler to translate them into Evergreen and Southern Islands online [1]. To test AMD chips we write our tests in OpenCL, relying on the AMD OpenCL compiler to translate them into Evergreen and Southern Islands online [1]. To test AMD chips we write our tests in OpenCL, relying on the AMD OpenCL compiler to translate them into Evergreen and Southern Islands online [1]. To test AMD chips we write our tests in OpenCL, relying on the AMD OpenCL compiler to translate them into Evergreen and Southern Islands online [1]. To test AMD chips we write our tests in OpenCL, relying on the AMD OpenCL compiler to translate them into Evergreen and Southern Islands online [1].

3. A plea for rigour

Our testing uncovered weak behaviours, and exposed several programming assumptions as false. Tab. 2 summarises our findings; we detail them below, and discuss their implications. In essence, this litany of examples is a plea for more rigour in vendor documentation and programming guides. Otherwise, we are bound to find issues in our hardware, compilers and software, such as the ones that we present below.

The behaviours that we expose correspond to classic litmus idioms, gathered in Tab. 3, together with a brief description and the figures where the idiom appears.

<table>
<thead>
<tr>
<th>name</th>
<th>description</th>
<th>figures</th>
</tr>
</thead>
<tbody>
<tr>
<td>coRR</td>
<td>coherence of read-read pairs</td>
<td>1, 4</td>
</tr>
<tr>
<td>mp</td>
<td>message passing (viz. handshake)</td>
<td>3, 5, 7, 9</td>
</tr>
<tr>
<td>lb</td>
<td>load buffering</td>
<td>8, 11</td>
</tr>
<tr>
<td>sb</td>
<td>store buffering</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 3: Glossary of idioms

Experimental setup For each test, we give the memory region and initial value of each location (see init in Fig. 3)
and the placement of threads in the execution hierarchy (threads), and we report the number of times the final condition (final) is observed (obs) on our chips during 100k executions of the test using the most effective incantations (Sec. 4.3). The complete histogram of results for each test can be found in the online material [1]. We conducted our Nvidia experiments on four machines running Ubuntu 12.04, and our AMD experiments on a single machine running Windows 7 SP1. In the Nvidia case, Tab. 4 lists the CUDA SDK and driver versions we used, and gives the PTX architecture specification, i.e. the argument of the -arch compiler option. In the AMD case, Tab. 4 lists the AMD Accelerated Parallel Processing SDK and Catalyst driver versions. The SDKs include the compilation tools for the respective platforms.

<table>
<thead>
<tr>
<th></th>
<th>GTX5</th>
<th>TesC</th>
<th>GTX6</th>
<th>Titan</th>
<th>TX7</th>
<th>AMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDK driver</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2.9</td>
</tr>
<tr>
<td>options</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>14.4</td>
</tr>
</tbody>
</table>

Table 4: Compilers and drivers used

### 3.1 Weak behaviours

#### 3.1.1 Sequential Consistency (SC) per location

This principle ensures that the values taken by a memory location are the same as if on SC [28]. Nearly all CPU models guarantee this [7], except Sparc RMO [43, Chap. D.4], which allows the weak behaviour of coRR (Fig. 1). As discussed in Sec. 1, this behaviour seems to spark debate for CPUs: indeed, it has been deemed a bug on some ARM chips [12]. Fig. 1 shows that we observed coRR on Nvidia Fermi and Kepler. We did not observe coRR on AMD TeraScale 2 or GCN 1.0 chips.

#### 3.1.2 Cache operators

**Message passing mp** On Nvidia we test mp with the loads bearing the cache operator which targets the L1 cache, i.e. .ca, (mp-L1, see Fig. 3) and all threads in different CTAs. The stores bear the cache operator .cg because our reading of the PTX manual implies that there is no cache operator for stores that target the L1 cache [36, p. 122]. We instantiate the fence at different PTX levels [36, p. 169]: cta, gl, and sys, and also report our observations when the fence is removed.

We observe the weak behaviour on the Tesla C2075, no matter how strong the fences are. Note that .ca is the default cache operator for loads in the CUDA compiler. [36, p. 121].

**Thus no fence (i.e. membar or CUDA equivalent in Tab. 5) is sufficient under default CUDA compilation schemes (i.e. loads targeting the L1 with the .ca cache operator) to compile mp correctly for Nvidia Tesla C2075.**

We experimentally fix this issue by setting cache operators to .cg (using the CUDA compiler flags -Xptxas -dlcm=cg -Xptxas -dsam=cg) and using membar .gl fences (see test mp+membar.gls online [1]).

---

On AMD we cannot directly test mp-L1, because we do not have direct access to the caches when working with OpenCL (as explained in Sec. 2.3). Instead, we revert to the classic mp test, with threads in distinct OpenCL work-groups, all variables in global memory, and OpenCL global fences (mem_fence(CLK_GLOBAL_MEM_FENCE)) between the loads and between the stores. Without the fences, we observe mp on AMD GCN 1.0 (obs: 9327) and TeraScale 2 (obs: 9327). With the fences we do not observe mp on TeraScale 2. On GCN 1.0 we still observe mp when fences are inserted; inspection of the Southern Islands ISA generated by the compiler shows that the fence between load instructions is removed. It is not clear from the OpenCL specification whether this is a legitimate compiler transformation. On the other hand the specification states that “loads and stores preceding the mem_fence will be committed to memory before any loads and stores following the mem_fence” [27, p. 277]; on the other hand it states that “There is no mechanism for synchronization between work-groups” [27, p. 30]. We have reported this issue to AMD.

**Coherent reads coRR** We tested whether using different cache operators within the coRR test can restore SC. The PTX manual states that after an L2 load (i.e. .cg) “existing cache lines that match the requested address in L1 will be evicted” [36, p. 121]. This seems to suggest that a read from the L2 cache can affect the L1 cache.

Let us revisit coRR (see Fig. 1). We run a variant that we call coRR-L2-L1 (see Fig. 4), where we first read from the L2 cache via the .cg operator and then from the L1 cache via the .ca operator. Thus the load 1.3 in Fig. 1 now holds the .ca operator, all the others being the same.

Fig. 4 shows that on the Tesla C2075, no fence guarantees that updated values can be read reliably from the L1 cache even when first reading an updated value from the L2 cache.

This issue does not apply to AMD chips for which, as discussed in Sec. 3.1.1, we did not observe coRR.

**Volatile accesses** PTX accesses can be marked .volatile, which supposedly [36, p. 131 for loads; p. 136 for stores] “may be used […] to enforce sequential consistency between threads accessing shared memory”. We test whether .volatile restores SC with shared memory with the test mp-

---

Figure 3: PTX mp w/ L1 cache operators (mp-L1)

![Figure 3: PTX mp w/ L1 cache operators (mp-L1)](image-url)
volatile (Fig. 5), a variant of mp where all accesses bear the .volatile annotation and locations are in the shared memory region and threads are in the same CTA (but different warps, see Sec. 2.1). W e observe violations on Fermi and Kepler; thus, contrarily to the PTX manual, the .volatile annotation does not restore SC for shared memory.

Because our framework for testing Nvidia chips tests PTX code, we must translate CUDA to PTX. We use the mapping summarised in Tab. 5, which we discovered by examining code generated by the CUDA compiler, release 5.5 (with the compiler flags -Xptxas -dlcm=cfg -Xptxas -dscm=cfg to set cache operators to .cg, to guard against the behaviour shown in Sec. 3.1.2).

For the examples in Sec. 3.2.1 and 3.2.2 we have also written OpenCL litmus tests for evaluation on AMD GPUs; this was not possible for the examples in Sec. 3.2.3 because, as discussed in Sec. 2.3, we were unable to avoid automatic placement of fences by the AMD OpenCL compiler.

### Table 5: CUDA to PTX mapping (for CUDA 5.5)

<table>
<thead>
<tr>
<th>CUDA</th>
<th>PTX</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomicCAS</td>
<td>atom.cas</td>
</tr>
<tr>
<td>atomicExch</td>
<td>atom.exch</td>
</tr>
<tr>
<td>__threadfence</td>
<td>membar.cta</td>
</tr>
<tr>
<td>__threadfence_block</td>
<td>membar.gl</td>
</tr>
<tr>
<td>atomicAdd(...)</td>
<td>atom.inc</td>
</tr>
<tr>
<td>store to global int</td>
<td>st.cg</td>
</tr>
<tr>
<td>load from global int</td>
<td>ld.cg</td>
</tr>
<tr>
<td>store to volatile int</td>
<td>st.volatile</td>
</tr>
<tr>
<td>load from volatile int</td>
<td>ld.volatile</td>
</tr>
<tr>
<td>control flow (while, if)</td>
<td>jumps &amp; predicated instructions</td>
</tr>
</tbody>
</table>

### 3.2.1 “GPUs exhibit no weak memory behaviours”

Several sources (e.g. [15, 26, 45]) simply omit memory model considerations. For example, Cederman and Tsigas [26, Chap. 35] describe a concurrent work-stealing double-ended queue (deque), adapting the queue of Arora et al. [13] to GPUs. The implementation seems to assume the absence of weak behaviour: it does not use fences. **Our testing shows that two bugs result from the absence of fences.**

```c
volatile int head, tail;

void push(task){
    int oldHead = head;
    if (tail <= oldHead.index) return EMPTY;
    atomicCAS(&head, oldHead, newHead)

    tail++;
}

Task steal(){
    int oldHead = head;
    if (tail <= oldHead.index) return EMPTY;
    __threadfence();
    task = tasks[oldHead.index];
    __threadfence();
    newHead = oldHead; newHead.index++;
    if (CAS(&head,oldHead,newHead)) return task;
    return FAILED; }

Task pop(){
    ... tail--; 
    ... head = newHead;
    return FAILED; }
```

Figure 6: CUDA code for queue of [26, p. 490-491]

In the implementation of [26, Chap. 35], each CTA owns a deque that it can push to and pop from. If a CTA's deque is empty then it attempts to steal a task from another CTA. Each deque is implemented as an array with two indices: tail is incremented by push and decremented by pop, and head is incremented by steal; tail and head are declared as volatile. Fig. 6 gives part of the implementation.
Message passing The first bug arises when executing two threads \( T_0 \) and \( T_1 \) in different CTAs. \( T_0 \) pushes to its deque, writes the tasks array (Fig. 6, line 3) and then increments tail (line 5). Assume that \( T_1 \) steals from \( T_0 \), sees the increment made by \( T_0 \) (line 8), and reads the tasks array at index head (line 10). Without fences, \( T_1 \) can see a stale value of the tasks array, rather than the write of \( T_0 \).

![Figure 7: PTX mp from load-balancing (dbl-mp)](image)

We distilled this execution into the dynamic-load-balancing test \( \text{dbl-mp} \) (Fig. 7) by applying the mapping of Tab. 5 to Cederman and Tsigas’ implementation [16]. Each instruction in Fig. 7 is cross-referenced to the corresponding line in Fig. 6. Without fences, the load 1.1 can read 1 and the load 1.4 can read 0, as observed on Fermi (Tesla C2075) and Kepler (GTX 660, GTX Titan). This means reading a stale value from the task array, and results in the deque losing a task. Adding the lines prefixed with \((+)\) forbids this behaviour. We did not observe the weak behaviour on Maxwell or AMD.

Load buffering The second bug arises again when executing \( T_1 \) and \( T_2 \) in different CTAs. \( T_0 \) pushes to its deque, \( T_1 \) steals, reads the tasks array (Fig. 6, line 10) and increments head (line 13). \( T_0 \) pops, reads the incremented head with a compare-and-swap (CAS) instruction, resets tail and returns empty. Then \( T_0 \) pushes a new task \( t \), writing to tasks at the original index (line 3). The implementation allows \( T_1 \)’s steal to read the 0.1, the second value pushed to the deque.

![Figure 8: PTX lb from load-balancing (dbl-lb)](image)

We distilled this execution into the dynamic-load-balancing test \( \text{dbl-lb} \) (Fig. 8), again following Tab. 5 and Cederman and Tsigas’ code [16]. Without fences, the load 1.1 can read from the store 0.4, and the CAS 0.1 can read from the CAS 1.3, as observed on Fermi (Tesla C2075) and Kepler (GTX 660, GTX Titan). This corresponds to the steal reading from the later pop, and hence the deque losing a task. Adding the lines prefixed with \((+)\) forbids this behaviour.

On AMD TeraScale 2 we find that the OpenCL compiler reorders \( T_1 \)’s load and CAS. We regard this as a miscompilation: it invalidates code that uses a CAS to synchronise between threads, even if the threads are in the same workgroup. Therefore we do not present the number of weak behaviours for HD6570 in Fig. 8 and write “n/a” instead. We reported this issue to AMD. On AMD GCN 1.0, we observe the weak behaviour of an OpenCL version of \( \text{dbl-lb} \).

Adding fences (see lines prefixed with \((+)\) in Fig. 6) forbids the behaviours of Fig. 7 and 8 in our experiments, on all Nvidia chips and on AMD GCN 1.0. As we explain in Sec. 3.2.3, pop’s store to head requires an atomic exchange.

3.2.2 “Atomic operations provide synchronisation” Several sources assume that read-modify-writes (RMW) provide synchronisation across CTAs (e.g. [30, 38, 42]). For example, Stuart and Owens “use \( \text{atomicExch()} \) instead of a volatile store and \( \text{threadfence()} \) because the atomic queue has predictable behavior, \( \text{threadfence()} \) does not (i.e. it can vary greatly in execution time if other memory operations are pending)” [42, p. 3]. Communication with the authors confirms that the weak behaviour is unintentional.

Nvidia’s CUDA by Example [38, App. 1] makes similar assumptions. Fig. 2 shows the lock and unlock from [38, App. 1]. For now we ignore the lines prefixed with \((+)\), which we added. Stuart and Owens’ implementation [42, p. 3] is similar, but uses atomic exchange (unconditional RMW) instead of CAS. The lock and unlock of Fig. 2 are used in a dot product [38, App. 1.2] (a linear algebra routine), where each CTA adds a local sum to a global sum, using locks to provide mutual exclusion. The absence of synchronisation in the lock permits stale values of the local sums to be read, leading to a wrong dot product calculation.
now. Lines 0.1 and 1.4 correspond to a store and a load inside a critical section; the other lines cross-reference Fig. 2.

Location m holds the mutex, which is initially locked (i.e. m = 1), and x is the data accessed in the critical section. The left thread stores to x and then releases the mutex with an atomic exchange. The right thread attempts to acquire the lock with a CAS instruction (1.1), and if the lock was acquired successfully (1.2), loads from x (1.4). The final constraint checks whether the lock is successfully acquired (i.e. r1 = 0), yet a stale value of x is read (i.e. r3 = 0).

Fig. 9 gives the outcome for threads in different CTAs using global memory. On Fermi and Kepler we observed stale values, violating the lock specification of [42], and showing the implementation from [38, App. 1] is wrong.

Our reading of the PTX manual implies that the .gl fences (prefixed with a (+) in Fig. 9) forbid the weak behaviour [36, Chap. 8.7.10.2], and with them, we no longer observe it during testing. As pointed out in the introduction, our findings prompted Nvidia to publish an erratum [33] confirming the false programming assumptions of [38, App. 1].

On AMD TeraScale 2 and GCN 1.0, we observe stale values for an OpenCL version of cas-sl (see [1]). Thus replacing CUDA atomics with their OpenCL counterparts in the dot product of [38, App. 1] would result in an incorrect implementation. This weak behaviour is not observed experimentally by inserting OpenCL global memory fences.

3.2.3 “Only unlocks need fences”

He and Yu [22] describe how to execute transactions for databases stored in global memory. They aim to guarantee the isolation property [21], i.e. the database state resulting from a concurrent execution of transactions should match some serial execution of the transactions. We distill litmus tests to experimentally validate the locks used by the database operations.

Spin lock Fig. 10 shows the CUDA spin lock of [22, p. 322]. For now, we ignore the lines marked (+). The locking is handled by the CAS on line 3, the critical section is on line 7, and the write on line 10 implements the unlock.

```
1 bool leaveLoop = false;
2 while(!leaveLoop) {
3   int lockValue = atomicCAS(lockAddr,0,1);
4   if(lockValue == 0) {
5     leaveLoop = true;
6 (+)   __threadfence();
7   // critical section
8 (+)   __threadfence();
9 (+)   atomicExch(lockAddr,0);
10 (-)  *lockAddr = 0;
11 (-)  __threadfence();}
```

Figure 10: CUDA spin lock implementation of [22, p. 322]

To investigate the correctness of the lock, we distilled the sl-future test, given in Fig. 11, from the CUDA code of

```
init:  \begin{pmatrix}
global x=0 \\
global m=1 
\end{pmatrix}
final: r0=1 ∧ r2=0 
threads: inter-CTA
```

<table>
<thead>
<tr>
<th>init</th>
<th>final</th>
<th>threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>ld.cg r0, [x]</td>
<td>7.1</td>
</tr>
<tr>
<td>0.2(+)</td>
<td>membar.gl</td>
<td>8.12</td>
</tr>
<tr>
<td>0.3(+)</td>
<td>__atomic.exch r1, [m], 0</td>
<td>9.13</td>
</tr>
<tr>
<td>0.4(-)</td>
<td>st.cg [m], 0</td>
<td>10.14(+)</td>
</tr>
<tr>
<td>0.5(-)</td>
<td>membar.gl</td>
<td>11.15</td>
</tr>
</tbody>
</table>

Figure 11: PTX spin lock future value test (sl-future)

Fig. 10. We assume that the threads are in different CTAs. Again, we first ignore the lines marked (+). The test checks whether a thread in the critical section can read a value from the future, i.e. written by the next critical section. The left thread reads a value within a critical section (line 0.1) then releases the lock (line 0.4). The right thread attempts to acquire the lock (line 1.1), and if successful, writes 1 to x in another critical section (line 1.5). The final condition checks whether the left thread can read the value written by the right thread when the right thread acquires the lock. Fig. 11 shows that this behaviour can be observed. This effect can lead to a violation of the isolation property described above.

The bugs arise because the CAS at the entry of the critical section (Fig. 10, line 3) does not provide any ordering nor does the release of the lock (line 10). As is, the __threadfence() does not help, because it appears after the release of the lock: this does not prevent the lock release (line 10) from being reordered with the accesses in the critical section (line 7). The fence would need to be placed before the release of the lock.

A possible fix for Fig. 10 is to remove the lines prefixed with (-), and add the lines prefixed with (+). The corrected version has fences both at the entry and exit points of the critical section. The spin lock uses CAS before entering the critical section in an attempt to provide mutual exclusion, but PTX annuls the guarantees afforded to atomic operations if other stores access the same location [36, p. 170], so we replace the normal store that releases the lock (the only other access to lockAddr) with an atomic exchange operation. We applied the equivalent transformations to the distilled test in Fig. 11, and did not observe the weak behaviour anymore.

4. Our testing methodology

Our testing tool takes a litmus test (as given in the previous sections) and produces a CUDA or OpenCL executable that runs the test many times while stressing the memory system, and produces a histogram of all observed outcomes.

4.1 Writing and generating litmus tests

Fig. 12 illustrates the GPU litmus format. Parts of it come from CPU litmus tests [5, 6]; others are specific to GPUs. We focus on the PTX case, the AMD case being similar.
memory stress

<p>| | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>coRR (intra-CTA)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1235</td>
<td>0</td>
<td>9774</td>
<td>161</td>
<td>118</td>
<td>847</td>
<td>362</td>
<td>632</td>
<td>3384</td>
<td>3993</td>
</tr>
<tr>
<td>sb (inter-CTA)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>181</td>
<td>1067</td>
<td>1555</td>
<td>2247</td>
<td>4</td>
<td>37</td>
<td>83</td>
<td>486</td>
</tr>
<tr>
<td>mp (inter-CTA)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>621</td>
<td>0</td>
<td>2921</td>
<td>315</td>
<td>1128</td>
<td>2372</td>
<td>4347</td>
<td>7</td>
<td>94</td>
<td>442</td>
</tr>
<tr>
<td>Nvidia GTX Titan</td>
<td>coRR (intra-CTA)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>462</td>
<td>1403</td>
<td>3308</td>
<td>6673</td>
<td>3</td>
<td>50</td>
</tr>
<tr>
<td>sb (inter-CTA)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>AMD Radeon HD 7970</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>
<td></td>
</tr>
</tbody>
</table>

Table 6: Observations out of 100k executions for combinations of incantations (all tests target global memory)

1. GPU_PTX_Sb
2. {0:.reg .s32 r0; 0:.reg .s32 r2;}
3. 0:.reg .b64 r1 = x; 0:.reg .b64 r3 = y;
4. 1:.reg .s32 r0; 1:.reg .s32 r2;
5. 1:.reg .b64 r1 = y; 1:.reg .b64 r3 = x;
6. T0 | T1
7. mov.s32 r0,1 | mov.s32 r0,1;
8. st.cg.s32 [r1],r0 | st.cg.s32 [r1],r0;
9. ld.cg.s32 r2,[r3] | ld.cg.s32 r2,[r3];
10. ScopeTree(grid(cta(warp T0) (warp T1)))
11. x: shared; y: global
12. exists (0:r2=0 / 1:r2=0)

Figure 12: GPU PTX litmus test sb

Line 1 states the architecture (here GPU_PTX) and test (here SB for “store buffering”, the typical x86-LSB scenario [37]). Lines 2–5 declare and initialise registers; note that PTX registers are typed (see [36, Chap. 5.2]).

Lines 6–9 list the test program with each column describing the sequential program to be executed by a thread. Each sequential program starts with an identifier (e.g. T0), followed by a sequence of PTX instructions. The list of supported instructions is described in Sec. 2.3.

The test ends with an assertion about the final state of registers or memory. In Fig. 12, line 12 asks if T0’s register r2 and T1’s register r2 can both hold 0 at the end.

Execution hierarchy A test specifies the location of its threads in the concurrency hierarchy (see Sec. 2.1) through a scope tree (borrowing the term scope from [24, 25]). In Fig. 12, we declare the scope tree on line 10: T0 and T1 are in the same CTA but different warps.

Memory hierarchy A test specifies a region for each location (viz. shared or global, see Sec. 2.2) in a memory map, immediately after the scope tree: e.g. line 11 in Fig. 12 specifies that x is in shared memory and y is in global memory.

Automatic test generation We extended diy—a tool for systematically generating CPU litmus tests (see [6] and http://diy.inria.fr)—to generate GPU tests. The diy tool assumes an axiomatic modelling style (see Sec. 5.1), where non-SC executions are encoded as cyclic graphs. It takes as input a set of edges, enumerates the possible cycles that can be formed with those edges, and generates a litmus test from each cycle. The main challenge in extending diy from CPUs to GPUs was the need for a much larger set of edges, to accommodate for GPU features such as scope trees and memory maps. Additionally, because we write our tests in an intermediate language, registers must be declared before use (see lines 2–5, Fig. 12), and dependencies must be protected against compiler optimisations (see Sec. 4.5).

4.2 Running litmus tests

Our tool generates code that is split into two parts: the CPU code and the GPU kernel code.

Testing locations The tests’ memory locations (viz. testing locations) are either in the global or shared memory region. Global testing locations are allocated and freed by the CPU while shared testing locations are statically allocated. For incantations (see Sec. 4.3), we allocate an array of global memory, distinct from the testing locations.

Testing threads In GPU programming, threads have access to their CTA id, CTA size and thread id (within the CTA) [34, p. 92]. These values can be combined to give each thread a unique global id within the grid. These ids differ from CPU affinity since they are part of the programming model, e.g. the semantics of CUDA’s __syncthreads() and OpenCL’s barrier() differs for threads in the same or distinct CTAs.

The kernel function, executed by all threads, switches on a set of testing threads runs the test and records register values into a global array that the CPU can copy and record. Unused threads either exit the kernel or participate in incantations (see Sec. 4.3).

Scope tree Our tool computes global ids of the testing threads matching the scope tree specified in the litmus test: if the scope tree requires T0 and T1 to be in different CTAs, we compute T0’s and T1’s global id so that their CTA ids differ. Unless the thread randomisation incantation (Sec. 4.3) is enabled, global ids are assigned in ascending order.
4.3 Incantations

The setup of Sec. 4.2 only witnessed weak behaviours in combination with incantations on Nvidia chips; these incantations also influenced the incidence of weak behaviours on AMD chips. We benchmarked them on a subset of our litmus tests (see complete results online [1]). Tab. 6 gives a selection of results for the GTX Titan and Radeon HD 7970, highlighting for each test the column (i.e. combination of incantations) with the greatest incidence of weak behaviours. We write intra-CTA (resp. inter-CTA) for tests with threads warp as a testing thread. The non-testing threads perform when this incantation is enabled.

Bank conflicts which can handle only one access at a time.

Hypothesis Varying the layout, e.g. the thread ids of testing threads and the number of threads per kernel, of a test in the execution hierarchy, in a way that is consistent with the scope tree of the test, might exercise different components and paths through the hardware and hence, increase the likelihood of weak behaviours.

Implementation We randomly select the ids of testing threads and the number of non-testing threads, while respecting the scope tree, on each test execution.

4.3.2 General bank conflicts

Hypothesis GPUs access shared memory through banks, which can handle only one access at a time. Bank conflicts occur when multiple threads in a warp seek simultaneous access to locations in the same bank. Hardware might handle accesses out of order to hide the latency of bank conflicts.

Implementation Bank conflicts apply only within a warp, so this incantation is performed only by threads in the same warp as a testing thread. The non-testing threads perform the same actions as the testing thread, but on locations that are offset from the testing locations. These offsets can be calculated either to produce bank conflicts or to avoid them, and we randomly oscillate between these on each iteration of the test. For warps that do not contain a testing thread, the threads either exit as in the basic testing setup (see Sec. 4.2), or perform the memory stress incantation (see Sec. 4.3.1).

Efficacy Tab. 6 shows that for intra-CTA tests (coRR), this incantation combined with all others (column 15) provokes the most weak behaviours on Titan. However, general bank conflicts alone do not expose any weak behaviours (see column 5), and even consistently reduce the number of inter-CTA weak behaviours when combined with memory stress: comparing columns 12 and 16 (which differ only by general bank conflicts), the number of weak behaviours for lb decreased from 2247 to 486. On HD7970 we only observed sb when bank conflicts were enabled, but this weak behaviour is still notably infrequent; we observe mp consistently more often when the incantation is enabled.

4.3.3 Thread randomisation

Hypothesis Varying the layout, e.g. the thread ids of testing threads and the number of threads per kernel, of a test in the execution hierarchy, in a way that is consistent with the scope tree of the test, might exercise different components and paths through the hardware and hence, increase the likelihood of weak behaviours.

Implementation We randomly select the ids of testing threads and the number of non-testing threads, while respecting the scope tree, on each test execution.

4.3.4 Thread synchronisation

Hypothesis Synchronising testing threads immediately before running the test promotes interactions while values are actively moving through the memory system, which might increase the likelihood of weak behaviours.

Implementation Testing threads synchronise immediately before running the test by atomically incrementing a counter and busy-waiting until the counter reaches the number of threads participating in the test. Compared with a similar incantation used in CPU testing [5] we had to take care to avoid deadlock due to the lack of progress guarantees across CTAs [34, p. 12] and within warps [20].

Efficacy Tab. 6 records the most weak behaviours on Titan when thread synchronisation is enabled. In inter-CTA tests (lb, mp, and sb) thread synchronisation increases the number of weak behaviours dramatically: comparing columns 15 and 16 (which differ only by thread synchronisation), the number of weak behaviours for coRR increased from 3993 to 9985. On HD7970, thread synchronisation consistently decreases the extent to which we observe mp, but consistently increases observations of lb when combined with memory stress.

4.4 Checking for optimisations

We now discuss how we guard against unwanted compiler optimisations in the case of Nvidia and AMD.
For Nvidia, recall from Sec. 2.3 that we write our tests in PTX. We compile this to SASS machine-level assembly with the ptxas assembler, which optimises the code for efficiency. If we invoke the assembler with minimal optimisations (-00), we find that although each PTX load or store has a corresponding SASS load or store, instructions that were adjacent in the PTX code are separated by several instructions in the SASS code. This is undesirable for testing: it can make the difference between observing weak behaviours or not.

If we invoke the assembler with maximal optimisations (-O3), most intermediate instructions are optimised away. However, we found that on rare occasions some instructions were reordered. For example, testing coRR on Maxwell uncovered cases where the CUDA 5.5 compiler reordered volatile loads to the same address; we did not observe this for CUDA 6.0. This is again harmful for testing, as we could attribute weak behaviours to the hardware, when in fact they were introduced by the compiler. In fact, such optimisations can occur at any optimisation level, in principle even at -00 (which does not fully disable optimisations).

To overcome these challenges, we developed the optcheck tool that detects whether SASS code has been optimised. To do this, we first add instructions to the PTX code of a litmus test that specify certain properties of the test, such as the order of instructions within a thread. The compiled code thus contains both the litmus test code and the specification. Our optcheck tool takes a binary, obtains the corresponding SASS code using cuobjdump [35, Chap. 2], and then checks whether the SASS code and the specification are consistent.

A specification (in PTX) consists of a sequence of xor instructions, placed at the end of each thread, for example:

```
register used xor.b32 r2, rb, 0x07f3a001
instruction type constant position
```

Each xor instruction corresponds to exactly one memory access instruction. The integer literal of an xor instruction (last operand) specifies several properties of the corresponding access: which register it uses, what type of instruction it is (e.g. 00 for a load with cache operator .cg), and its position in the order of memory access instructions. The constant serves to distinguish these specification instructions from any xor instructions that appear in the code. In the litmus tests we generate, the accesses within a thread use different registers, so we can always create a one-to-one correspondence between memory accesses and xor instructions.

Our optcheck tool was essential in checking the data which informs our model of PTX (Sec. 5); this data comes from running 10930 tests on the Nvidia chips of Tab. 1. Our AMD testing is for now more modest: 12 distinct litmus tests to assess weak behaviours and programming assumptions in Sec. 3 and 14 tests to evaluate the incantations of Sec. 4.3.

For all these tests we checked the generated Evergreen (for TeraScale 2) and Southern Islands (for GCN 1.0) ISA files by hand to guard against unwanted compiler optimisations. We observed that multiple loads from the same location (e.g in Fig. 1) get optimised into a single load. We explain online [1] how to suppress this optimisation. We also explain how to check whether the order of loads and stores is consistent with the original litmus test.

4.5 Manufacturing dependencies

We also want to test whether dependencies between memory accesses have an effect on memory consistency. For CPUs, such litmus tests use false dependencies [6]: ones that have no effect on the computed values. For example, in the PTX code snippet in Fig. 1a, there is an address dependency between the load in line 1 and the load in line 5, since the result of the first load is used to compute the address of the memory location accessed by the second load. The dependency is a false dependency as the result of the xor is always 0, so the subsequent add never changes the value of the address register r4.

```
1 ld.s32 r1, [r0] 2 ld.s32 r1, [r0]
2 xor.b32 r2, r1, r1 2 and.b32 r2, r1, 0x80000000
3 cvt.u64.u32 r3, r2 3 cvt.u64.u32 r3, r2
4 add.u64 r4, r4, r3 4 add.u64 r4, r4, r3
5 ld.s32 r5, [r4] 5 ld.s32 r5, [r4]
(a) Optimised by ptxas (-O3) (b) Not optimised by ptxas (-O3)
```

Figure 13: Load-load address dependencies

Since we compile our litmus tests with the highest optimisation settings (cf. Sec. 4.4), the PTX assembler would recognise that the result of the xor is always 0, and hence remove lines 2–4, thereby removing the dependency. Therefore, we use a different scheme for testing dependencies, exemplified in Fig. 13b. It is based on and-ing with a constant that has just the high bit set. The result of this operation will always be 0, since in our litmus tests all memory locations are initialised to 0 and the store instructions only write small positive values (with the high bit being 0). However, determining that the result is 0 would require an inter-thread analysis (which the PTX assembler does not perform). Thus, the dependency is left intact.

5. A model of Nvidia GPUs

Sec. 3 illustrates some difficulties faced by GPU programmers. One crucial issue is to reliably predict the possible behaviours of concurrent GPU programs. As a step forward, we present a formal model for a fragment of PTX. We also propose a simulation tool that determines the allowed behaviours of PTX litmus tests w.r.t. our formal model.

5.1 Axiomatic models

Our model is axiomatic (see e.g. [6, 7]), thus discriminates, for a given program, its candidate executions. Given a PTX program we build a set of candidate executions which our
There is one relation per strength of fence, system (sys), the second thread issues two loads from which consists of a set of instructions. The program order relation (po) between all events. 

The scope relations come directly from the scope tree. Once yielded a candidate execution graph.

5.1.1 Candidate executions

Informally, a candidate execution is a graph (see e.g. Fig. 14), which consists of a set of memory events for each thread, and relations over these events. These relations describe the program order within a thread, the communications between threads, and specifically for GPUs, the scopes of threads along the memory hierarchy.

Memory events give a semantics to instructions (we omit the formal instruction semantics for brevity). Essentially, loads give rise to reads, and stores to writes.

For example in the test of Fig. 14, the first thread issues two stores, the first one to memory location x and the second one to location y, separated by a fence (member. cta). In the execution graph of Fig. 14, we have two corresponding write events, bearing the same cache operator (cg), and mentioning the same locations and values as the store instructions. The second thread issues two loads from y and x, separated by a fence (member. gl). In the execution graph, we have two corresponding read events, bearing the same cache operator (cg), and mentioning the same locations as the load instructions. The values of the reads are given by the final state of the litmus test.

Scope relations link events from threads in the same CTA (cta), same grid (gl) and anywhere in the system (sys). Note that the sys relation is simply the universal relation between all events.

The program order relation (po) totally orders events in a thread, and does not relate events from different threads.

The dependency relation dp, included in po, relates events in program order whose instructions are separated by an address (addr), data (data) or control (ctrl) dependency.

Similarly, the member fence relations, included in po, relate events whose instructions are separated by a fence. There is one relation per strength of fence, sys, gl and cta. In Fig. 14 the fence on the first thread corresponds to the member. cta relation between the writes a and b.

Communication relations The read-from relation (rf) associates every read r with a unique corresponding write that agrees with r on variable and value components. In Fig. 14, the load of y on the second thread reads from the store of y on the first thread, as indicated by the final state (r0=1). Thus we have a read-from between the two corresponding events b and c. The load of x on the second thread reads from the initial state (since r2=0 in the final state), which is depicted as a rf arrow with no source pointing to the read d.

Writes to a single location are totally ordered by coherence co, i.e. the order in which they hit the memory.

5.1.2 From a PTX litmus to its candidate executions

Recall that a PTX litmus test (see Sec. 4.1 and Fig. 12) specifies the shared variables, with initial values, the sequence of instructions for each thread, and a scope tree describing how the threads are organised into warps and CTAs.

We can enumerate the candidate execution graphs of a litmus test by unwinding the body of each thread: this gives us the program order po for each thread, as well as the dependency and fence relations, which are included in po. The scope relations come directly from the scope tree. Once these relations are established, any choice for the read-from and coherence relations respecting the above definitions yields a candidate execution graph.

5.2 Defining our model

Given a candidate execution graph, originating from a PTX litmus test, we seek to answer the question of whether the execution is allowed or not. As mentioned earlier, we achieve this through an axiomatic model. Essentially, an axiomatic model lists a set of constraints over execution graphs, built from the primitive relations described above, such that an execution is allowed if and only if it satisfies the constraints.

5.2.1 Derived relations over events

The following derived relations are useful in defining the constraints of our model.

The relation po-loc is the program order po restricted to events having the same memory location.

The relation ref is the rf relation restricted to external events, i.e. events coming from different threads. For example in Fig. 14 the read-from relation between b and c is in fact an ref relation, as b and c belong to distinct threads.

The from-read relation fr relates a read r to all the writes overwriting the value r reads from. Formally, (r, w) relates by fr when r reads from a write w′ (i.e. (w′, r) is in rf) such that w′ hits the memory before w (i.e. (w, w) is in co).

In Fig. 14, the read of x on the second thread reads from the initial state. By convention the initial state for a given location hits the memory before any update to this location; thus the read d of x is in fr with the update a of x.
5.2.2 The .cat format illustrated on Sparc RMO

The .cat format of [7] uses a small language that allows the user to describe an axiomatic model in a succinct way. A .cat file, together with a litmus test, can be given to the herd tool (see [7] and http://diy.inria.fr/herd). Given an instruction semantics module (i.e. a way to translate a program into a set of candidate executions) for the language under scrutiny (in our case PTX), the tool takes a .cat file (e.g. the one in Fig. 16) to produce a simulator that enumerates all the valid executions of a litmus test.

**Syntax of .cat files** In Fig. 15 and 16, we use several syntactic constructs that we list here. One declares new relations with let. The union of relations is written |, and their intersection is &. One can obtain a subrelation of a relation r using various filters: for example WW(r) returns only the pairs of write events related by r; RW(r) returns the read-write pairs related by r. One can enforce the acyclicity of a relation r by declaring the check acyclic r. One can give a name to such a check with the keyword as; for example acyclic (po | com) as sc declares a new check sc, that enforces the acyclicity of the union of program order and communication relations.

Our model resembles Sparc’s Relaxed Memory Order (RMO) [43], factoring in the GPU concurrency hierarchy. As an introduction to the .cat syntax, we present here the .cat transcription of Sparc RMO as formalised in [3].

**Intuitively, RMO** allows any pair of memory accesses to different locations to be reordered, unless separated by a dependency or a fence. For example, RMO allows the non-SC behaviour of mp (see Fig. 14). To forbid this behaviour, one can use a fence between instructions 0.1 and 0.3 and a dependency between instructions 1.1 and 1.3. Additionally, RMO allows the test coRR of Fig. 1.

**Formally, RMO** relies on three principles, detailed below.

**SC PER LOCATION WITH LOAD-LOAD HAZARD** Most CPU hardware guarantees what we call SC PER LOCATION, explained in Sec. 3.1.1. RMO relaxes this constraint, as it allows coRR (Fig. 1). As shown in Fig. 1, Nvidia chips exhibit this behaviour; thus our model allows it.

Formally, following [3, 4, 7], this corresponds to the constraint sc-per-loc-llh on line 4 of Fig. 15, which builds on the definitions on lines 1 and 3. More precisely, line 1 defines the relation com (for communication) as the union of rf, co and fr. Line 3 defines po-loc-llh: program order over single locations without read-read pairs. We require on line 4 that communications do not contradict po-loc-llh.

The weak behaviour of coRR is allowed by our model, because we excluded the read-read pairs from the sc-per-loc-llh check at line 3.

**NO THIN AIR** prevents causal loops: where the dependency and reads-from, that intuitively suggest causation, form a cycle. Load buffering tests, e.g. db-lb (Fig. 8), check for violations of this principle. Formally, following [3, 4, 7], this corresponds to lines 5-6. Line 5 defines the relation dp for (dependencies), made of the union of address, data, and control dependencies. Line 6 declares the check no-thin-air, which requires that the union of dp and rf is acyclic.

**The rmo relation** declared at line 7 collects the orderings due to dependencies dp, inter-thread communication rf, co and fr, and fences fence, where the behaviour of fences is left parametric. Constraints over rmo can be used to forbid the weak behaviour of idioms such as message passing mp or store buffering sb, when using the appropriate ordering, e.g. fences between writes and dependencies between reads. Such constraints are at the heart of our PTX model.

5.3 Our PTX model

Our model is the concatenation of Fig. 15 and 16, and implements RMO per scope. In contrast to RMO for CPUs, for which Fig. 15 suffices, our PTX model duplicates the rmo relation at each scope (see lines 11, 12 and 13).

More precisely, lines 8–10 declare the relations sys-fence, gl-fence and cta-fence, which provide ordering within the named scopes. Lines 11–13 then instantiate the generic rmo relation (see Fig. 15, line 7) for each scope of fence, using the intersection operator (&) to restrict to the appropriate scope. Lines 14–16 enforce the acyclicity of the three rmo relations; this implements RMO at each scope.

In Fig. 14, the execution of mp exhibits a cycle in the union of membar.cta, rfe, fr and membar.gl, i.e. a cycle in rmo-cta. Our model forbids this execution by the constraint cta-constraint at line 14.

5.4 Validating our model

We developed a PTX simulator as part of the herd tool [7]: it enumerates, for a litmus test, its candidate executions
(see Sec. 5.1.1), then discriminates them following our PTX model (see Fig. 15 and 16). We automatically generated 10930 tests with our extension of the dry tool (see Sec. 4.1).

We supplied all our tests to herd, and our PTX .cat model: our model is experimentally sound w.r.t. our 10930 tests for the Nvidia chips of Tab. 1. This means that whenever the hardware exhibits a behaviour, our model allows it. We provide all experimental data for all chips online [1].

5.5 Limitations of our model

Our model reflects the hardware behaviour of a PTX program, compiled in the setup given in Tab. 1, in which accesses of shared data have not been reordered or optimised, as checked by our optcheck tool (see Sec. 4.4). The limitations of our model are as follows: we only handle the instructions listed in Sec. 2.3, and we assume that all accesses use the .cg cache operator (which targets the L2 cache).

The reason for choosing .cg is that our observations on Fermi (see 3.1.2) show that it is not possible to restore ordering between accesses marked .ca (targeting the L1).

6. Related work

Testing and modelling  Our method follows the work of Alglave et al. [4–7] for CPUs, which follows the steps of Collier [17]. More precisely, in [17] Collier presents the ARCHTEST tool for CPUs, which runs a small number of fixed tests to check for discrepancies with Lamport’s Sequential Consistency [28], e.g. coRR (see Fig. 1). Using few handwritten tests has limitations, as rich sets of litmus tests were required to inform the formalisation of weak architectures such as IBM Power [6, 7, 39]. Alglave et al. [6] developed a method to automatically generate litmus tests for CPUs based on the axiomatic framework of [4, 6], and implemented their approach in the dry toolsuite (see [5–7] and http://diy.inria.fr). The toolsuite generates and runs systematic families of litmus tests, and collects their outcomes. As detailed in Sec. 4, we implemented several novel extensions to make these tools suitable for GPUs.

Microbenchmarking is loosely related to our approach. While we are concerned with semantics, microbenchmarking gathers performance data. The GPUBench [2] suite gathers statistics such as memory bandwidth and instruction throughput of AMD and Nvidia GPUs. Wong et al. [44] developed a test suite to reveal microarchitectural aspects of Nvidia GeForce GT200 and GTX280 GPUs: they draw conclusions about the latency of memory accesses, or the structure of the caches. Feng and Xiao [19] analyse the overhead of barrier synchronisation.

Checking for optimisations  Our checking whether a litmus test has been optimised (see Sec. 4.4) is related to testing of compiler optimisations for concurrent programs.

Eide and Regehr check whether accesses to C volatile variables are compiled correctly [18]. They compile a test case both with and without optimisations (e.g. -O3 and -00), then run both versions with the same input while logging the accesses to volatile variables. If the traces of the two versions differ, an invalid optimisation has been detected. Morisset et al. extend this work to a subset of C++11 [31].

Our approach differs from these in that we do not make use of an unoptimised version of the code, but instead embed a specification of the expected instruction sequence into the optimised version. Moreover, we statically check whether the compiled code conforms to the specification. Finally, the methods have different aims: our aim is not to find compiler bugs but to detect unwanted reorderings due to compilation.

GPU models  Hower et al. proposed several models for GPUs [24, 25]. All of these models are “SC-for-DRF” models, i.e. only concern data race free programs, and ensure that such programs have an SC semantics. Somewhat relatedly, Hechtman and Sorin show that weak memory has negligible performance benefits on their set of benchmarks, thus argue that SC is an attractive model for GPUs [23]. By contrast, and since we are concerned with hardware, we give semantics to race free and racy programs alike.

Sorensen et al. [40, 41] proposed an operational model of Nvidia hardware, based on reading the Nvidia documentation and communication with Nvidia representatives; they provide intuition about their model using GPU litmus tests similar to the ones we present (e.g. Fig. 1). However, this model is unsound w.r.t. hardware: the inter-CTA lb-membar.ctas test, i.e. a variant of dbb-lb (Fig. 8) without atomics and with membar .ctas fences between all accesses, is forbidden by the model, but observed 586 times on GTX Titan and 19 times on GTX 660 out of 100k iterations (see [1]).

7. Perspectives

The present work uncovered weak behaviours, and exposed several programming assumptions as false, summarised in Tab. 2. We use these examples to plead for clarity and rigour in vendor documentations. We believe that formal models, such as the one we propose in Sec. 5, can help remedy this situation, providing a rigorous basis on which to build our systems. Further steps towards that goal include building language level models (e.g. for OpenCL), and sound compilation mappings from language to hardware.

Acknowledgments  We thank Luc Maranget for feedback on extending litmus and diy, Mary Hall for lending us her group’s machines, Tom Stellard for feedback on setting up our AMD testbed, Matt Arsenault and Brad Beckmann for clarification on the AMD OpenCL compiler behaviour, and Benedict Gaster, Peter Sewell, Tatiana Shepeisman and our reviewers for their feedback.

Support  EPSRC grants EP/H005633, H008373, K008528, K011499, K039431), EU FP7 project CARP (287767), NSF CCF Awards 1346756 and 1302449, and SRC project 2269.002.

NSF CCF Awards 1346756 and 1302449, and SRC project 2269.002.
References


[34] Nvidia. CUDA C programming guide, version 6, July 2014.


