Exposing Errors Related to Weak Memory in GPU Applications

Tyler Sorensen
Imperial College London, UK
t.sorensen15@imperial.ac.uk

Alastair F. Donaldson
Imperial College London, UK
alastair.donaldson@imperial.ac.uk

Abstract

We present the systematic design of a testing environment that uses stressing and fuzzing to reveal errors in GPU applications that arise due to weak memory effects. We evaluate our approach on seven GPUs spanning three Nvidia architectures, across ten CUDA applications that use fine-grained concurrency. Our results show that applications that rarely or never exhibit errors related to weak memory when executed natively can readily exhibit these errors when executed in our testing environment. Our testing environment also provides a means to help identify the root causes of such errors, and automatically suggests how to insert fences that harden an application against weak memory bugs. To understand the cost of GPU fences, we benchmark applications with fences provided by the hardening strategy as well as a more conservative, sound fencing strategy.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.2.5 [*Software Engineering*]: Testing and Debugging

Keywords GPU, CUDA, Nvidia, weak memory, stress testing, memory fences, synchronisation, concurrency

1. Introduction

General purpose programming languages for graphics processing units (GPUs), e.g. CUDA [38] and OpenCL [24], allow applications from a wide spectrum of domains to take advantage of the computational power and energy efficiency offered by these devices (see [45, pp. 8-10] for an overview).

GPU applications are prone to concurrency bugs, which are notoriously difficult to reproduce and fix due to the sheer number of possible instruction interleavings. Worse, GPUs have been shown to implement *weak memory models*, so that behaviours beyond those obtained from straightforward interleavings are possible [8], making GPU application debugging even more challenging.

It has been argued, through hand analysis, that certain deployed CUDA applications can exhibit weak memory bugs in theory [8], but we are not aware of any existing practical method for exposing witnesses to weak memory bugs in GPU applications, i.e. demonstrating erroneous outcomes from actually running the application on current GPUs. Our focus in this work is on investigating (a) whether GPU weak memory bugs can be provoked when running real-world applications on state-of-the-art hardware, (b) whether such bugs can be provoked into occurring *frequently* (to aid in testing and debugging), and (c) the performance cost of adding fences to harden GPU applications against such bugs.

We make three main contributions:

1. We develop a novel testing environment designed to reveal weak memory behaviours in real-world GPU applications. This environment uses a sophisticated memory stressing strategy, inspired by prior work on GPU litmus testing [8], systematically tuned per chip using results from nearly half a billion micro-benchmark executions. Applying the testing environment requires no prior knowledge of the application under test (Sec. 3).
2. We evaluate our approach on seven GPUs spanning three Nvidia architectures, across ten application case-studies, showing that we can provoke errors related to weak memory and discovering previously unknown weak memory issues in two applications (Sec. 4). Our experiments show that two straightforward methods for memory stressing are *not* effective at exposing weak memory bugs, while in contrast our novel test environment (tuned per chip) is often highly effective.
3. We use our testing environment as a fence placement mechanism, to help understand the root causes of weak memory bugs, and to harden applications against such bugs (Sec. 5); we also benchmark the overhead (in terms of runtime and energy) associated with inserting fences to harden applications (Sec. 6).

Table 1: The seven Nvidia GPUs that we study

chip	architecture	short name	released
GTX 980	Maxwell	980	2014
Quadro K5200	Kepler	K5200	2014
GTX Titan	Kepler	Titan	2013
Tesla K20	Kepler	K20	2013
GTX 770	Kepler	770	2013
Tesla C2075	Fermi	C2075	2011
Tesla C2050	Fermi	C2050	2010

Unlike previous works for checking applications under weak memory (see Sec. 7), our technique does not require a formal memory model description, and requires no modifications to the compiler or scheduler.

We begin with a high-level overview of our approach.

Running example We use the CUDA code of Fig. 1, extracted from the dot product case study from [45, ch. A1.2] which we dub *cbe-dot*, to illustrate our approach. (Background on CUDA is provided in Sec. 2.) The application incorporates a spinlock, and correctness depends on the source code ordering between the atomic operations on the lock (lines 19, 22) and the memory operations in the critical section (line 15) being preserved. Reordering these operations can lead to an incorrect dot product value being computed. However, no erroneous behaviour is observed when conducting 1000 executions of the application on a Tesla K20 GPU. A developer who is not suspicious about weak memory effects might conclude that the application is correct.

Testing environment We present in Sec. 3 a testing environment for provoking weak behaviours in applications across a range of Nvidia GPUs, given in Tab. 1.[1] Under our testing environment, errors (due to weak memory) appear in 102 out of 1000 executions of *cbe-dot* on the K20.

The key part of our testing environment is a memory stressing strategy that targets a completely disjoint region of memory from the application data (called a *scratchpad*) using extra GPU threads disjoint from the threads that execute the application (the extra threads are called *stressing threads*). Because the stressing threads and memory are disjoint from application threads and data, the set of possible behaviours a program can exhibit remains the same.

Using nearly half a billion micro-benchmark executions, the stressing strategy is tuned per GPU to identify *where* to stress within the scratchpad, *which* instructions to apply during stressing, and *how many* memory locations to stress in parallel. Parameters for these values are selected based on how many weak behaviours they expose in the micro-benchmarks. For example, micro-benchmarking shows that for K20 it is effective to apply stress to two memory locations in parallel, each aligned at 32-word boundaries.

[1] We restrict to Nvidia GPUs because the majority of available applications that exhibit fine-grained concurrency are written using Nvidia's CUDA.

```
1  __global__
2  void dot(int *mutex, float *a, float *b, float *c) {
3    int tid = threadIdx.x + blockIdx.x * blockDim.x;
4    int cacheIndex = threadIdx.x;
5    float temp = 0;
6
7    while (tid < N) {
8      temp += a[tid] * b[tid];
9      tid += blockDim.x * gridDim.x; }
10
11   // local computation code omitted
12
13   if (cacheIndex == 0) {
14     lock(mutex);
15     *c += cache[0];
16     unlock(mutex); }        }
17
18 __device__ void lock(int *mutex) {
19   while (atomicCAS(mutex, 0, 1) != 0 ); }
20
21 __device__ void unlock(int *mutex) {
22   atomicExch(mutex, 0); }
```

Figure 1: CUDA code for the *cbe-dot* application

Weak behaviours in applications We evaluate our testing environment on ten GPU application case-studies that implement fine-grained concurrency idioms, e.g. mutexes and concurrent data-structures (Sec. 4). On each of the seven GPUs of Tab. 1, we executed each application repeatedly for one hour under our testing environment. For comparison, we also tested the applications under two more straightforward stressing environments that are not tuned per chip.

Our results show that our testing environment is able to provoke erroneous outcomes in 55 out of 70 chip/application combinations, leading to the discovery of previously unknown weak memory issues in two applications. Because errors are easier to debug when they occur frequently, we also measure the *effectiveness* of the testing environment, i.e., how frequently bugs are provoked. If a bug appears in more than 5% of the executions associated with a chip/application combination, we say the testing framework is *effective*. Out of the 55 of applications we observed errors for, our testing method is effective for 43 of them.

We evaluate our tuned stress against several other straightforward stressing methods. We observe these methods are considerably less able to expose weak behaviours, the most capable method revealing erroneous runs 13 out of 70 chip/application combinations (and effective for only 6).

Hardening applications against weak memory bugs Unlike in the case of e.g. C11, there is currently no agreed formal memory for CUDA. Consequently, it is not possible to provide formal correctness guarantees about CUDA applications that use fences to eliminate weak memory bugs. As a pragmatic alternative, we employ our testing method to suggest a minimal set of memory fences that suffice to suppress weak memory bugs under our aggressive test environment (Sec. 5). This *empirical fence insertion* starts with a fence instruction inserted after memory access and repeatedly attempts to remove fences, using our testing environment to

assess, empirically, whether each removal introduces a bug. The process converges to a minimal set of fences such that removing any single fence exposes erroneous behaviours.

While clearly providing no guarantees, the suggested fences can aid developers in understanding the causes of weak memory defects, and can aid in *hardening* the application against weak memory defects by, at a minimum, making them less likely to occur. In *cbe-dot* for K20, the fence insertion suggested adding a single fence after line 15, suggesting an error in the `unlock` function. Prior work, through hand analysis, identified the same issue and prescribed a fence at the beginning of the `unlock` function; this is logically the same fence identified by our fence insertion.

Evaluating the cost of fences To understand the cost of adding fences to applications, we benchmark both the runtime and energy usage of our application case studies (Sec. 6). We consider three fencing strategies: removing all fences (unsafe), adding a fence after every memory access (safe, but conservative), and adding fences suggested by empirical fence insertion (hardened, but not guaranteed to be safe). This allows us to investigate the overhead associated with hardening applications via empirical fence insertion, providing a lower bound on the cost of eliminating weak memory defects, vs. conservatively guaranteeing absence of weak behaviours by adding fences after all memory accesses, providing an upper bound on the cost. The results of *cbe-dot* for K20 shows that adding fences via empirical fence insertion incurs a small runtime/energy cost (less than 3%), in comparison to full fence insertion, which incurs a 145% runtime and a 173% energy cost.

Related GPU memory model testing work Previous work has applied weak memory testing to GPUs in the context of short idiomatic tests (i.e. litmus tests) with the tool GPU LITMUS [8]. GPU LITMUS implements a memory stress heuristic that inspired this work. However, the method and aims of [8] are fundamentally different to ours. Specifically, the aim in [8] was simply to show the existence of weak behaviours, using carefully crafted tests. In contrast, our aim is to test applications in a black box manner, with no *a priori* knowledge about the application, e.g. the communication idioms used in the application.

Our aim is also bolder: to be able to provoke errors frequently, rather than merely show the existence of errors. Hence optimising for *frequency* of weak behaviours observed, with respect to idioms that have classically caused bugs, is at the heart of our novel stressing strategy.

2. Background

We provide necessary background on memory models and litmus tests, and a brief overview of the CUDA programming model including details of memory fences in CUDA.

Memory models For a given architecture and concurrent program, a *memory model* determines the values that load instructions are allowed to return [46, ch. 1]. The strongest memory model, *sequential consistency*, only allows executions that correspond to an interleaving of thread instructions [25]. Many architectures (e.g. x86, ARM, Nvidia GPUs) provide *weak memory models* [7, 8, 46], whereby executions may not correspond to such an interleaving. We say such executions exhibit *weak behaviour*. Weak behaviours can be disallowed (at a performance cost) by placing *memory fences* between memory accesses [6–8, 46].

Message Passing (MP)		Load Buffering (LB)		Store Buffering (SB)	
init: x = 0, y = 0		init: x = 0, y = 0		init: x = 0, y = 0	
T1	T2	T1	T2	T1	T2
x ← 1;	r1 ← y;	r1 ← x;	r2 ← y;	x ← 1;	y ← 1;
y ← 1;	r2 ← x;	y ← 1;	x ← 1;	r1 ← y;	r2 ← x;
weak behaviour: r1 = 1 ∧ r2 = 0		weak behaviour: r1 = 1 ∧ r2 = 1		weak behaviour: r1 = 0 ∧ r2 = 0	

Figure 2: MP, LB, and SB weak memory litmus tests

Weak behaviours can be illustrated by *litmus tests*: short concurrent programs with a query about the final state. Three well-known litmus tests, discussed throughout the paper, are presented in Fig. 2. The *message passing* (MP) test illustrates a handshake protocol where thread 1 writes data to memory location x and then sets a flag in memory location y, while thread 2 reads the flag value and then reads the data. The test exhibits weak behaviour if thread 2 can observe a set flag ($y = 1$) but see stale data ($x = 0$). The weak behaviour illustrated by the *load buffering* (LB) test checks whether load instructions are allowed to be buffered after store instructions. *Store buffering* (SB) similarly checks whether store instructions are allowed to be buffered after load instructions. These weak behaviours are all allowed on ARM and IBM Power CPUs, and Nvidia GPUs. The SB weak behaviour (but not MP nor LB) is also allowed under the x86 TSO memory model.

We use *communication idiom* to refer to a configuration of threads, locations and instructions that could lead to a weak behaviour, and *communication locations/communicating threads* to refer to the memory locations/threads involved in a communication idiom. For example, the MP test of Fig. 2 describes a communication idiom over communication locations x and y, with two communicating threads.

The CUDA programming model In the CUDA programming model [38], a program consists of *host* code that executes on the CPU of the machine, and *device* code that executes on the GPU. The device code is called a *kernel*, and is executed by many threads in a *single instruction, multiple threads* (SIMT) manner. A thread is a basic unit of computation that executes the kernel. Threads are grouped in disjoint sets of size 32, called *warps*, that execute in lock-step: they synchronously execute the same instruction and share a program counter. Warps are grouped into disjoint sets called *blocks*; the number of threads (and by extension, warps) in

a block is a parameter of the kernel. Collectively, the blocks that execute a kernel form a *grid*; the grid size is also a parameter of the kernel. Threads may query their thread id within a block, their block id within the grid, the number of threads per block and the number of blocks in the grid, using CUDA primitives.

Threads in the same block can synchronise at a *barrier*. Each thread in the block waits at the barrier until every thread in the block has reached the barrier, at which point memory consistency is guaranteed between threads in the block. Execution of a CUDA barrier has undefined behaviour unless *all* threads in a block execute the barrier; violation of this condition is known as *barrier divergence* [38, p. 98].

Threads in the same block can communicate using *shared memory*, and a single *global memory* region is accessible to all threads in the grid.

Memory fences in CUDA In CUDA, weak behaviours for communication idioms where communicating threads are in the same block (resp. different blocks) can be disabled using block level fence (resp. device level fence) instructions [38, ch. B.5]. In this work we focus exclusively on inter-block communication idioms because we did not encounter applications that use communication idioms for which communicating threads are in the same block.

3. The Design of Our Stressing Strategy

Here we describe the systematic development of our memory stressing strategy, inspired by work in [8] and based on results of micro-benchmarks, designed to be effective at revealing weak behaviours. By targeting a *scratchpad* memory region using *stressing threads*, completely disjoint from the memory and threads of the application, our stressing strategy does not modify the possible behaviours of the application. We additionally want our stressing strategy to be agnostic to the communication idioms (including communicating threads and locations) inside the application, thus allowing for black box application testing.

We partition threads into stressing threads and application threads at the block level (rather than allowing a single block to contain both kinds of threads) to avoid introducing barrier divergence (see Sec. 2). We refer to a block of stressing threads as a *stressing block*.

We conducted pilot experiments applying different memory stress to three applications, the *cbe-ht*, *cbe-dot* and *ct-octree* applications presented in Sec. 4, running on two GPUs, Titan and C2075 (see Tab. 1). Our findings indicated that stressing could be effective at provoking erroneous executions due to weak behaviours, but that the effectiveness of memory stress is highly dependent on a number of parameters. Specifically, the added scratchpad memory provides many possible *locations* that can be stressed, stressing can be applied to many different *location combinations*, and there are many choices for the *instruction sequences* that can be used to stress these memory locations. We refer to these as *memory stress parameters*, and refer to the extent to which a set of memory stress parameters is able to provoke weak behaviours running on a particular GPU as *effectiveness*.

When stressing, we do not consider the base address of the scratchpad because GPUs use virtual memory addressing (reported in [31]) and we are unaware of any method for obtaining the physical address from the virtual address. Thus, we cannot control the distance between the physical locations of the scratchpad and the memory used by the application under test. Because of this, we design our stress so as not to rely on this distance.

We detail the micro-benchmark design and results used to obtain effective parameters per chip, guided by insights gained through our pilot experiments. We present full details using precise notation to enable others to reproduce our approach in future work, e.g. for CPU or next-generation GPU application testing.

3.1 Focusing on Litmus Tests

Our overall aim is to provoke weak behaviours in an application without knowledge of the fine-grained idioms that the application might rely on. Weak behaviours of the MP, LB and SB tests (Fig. 2) are known sources of bugs; e.g. MP and LB weak behaviours were shown (via hand analysis) to be problematic in GPU applications [8], and SB weak behaviours cause issues in an implementation of Dekker's algorithm [46, p. 20]. All weak memory bugs we are aware of relate to one of these idioms.

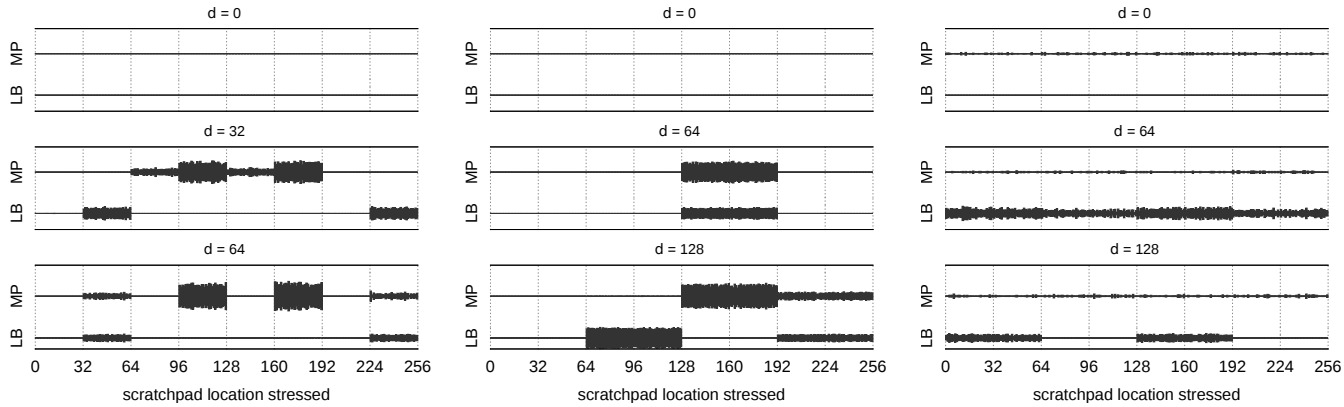
Hypothesising that memory stress parameters tuned according to these litmus tests are likely to be effective in exposing practical weak memory bugs, we assess the fitness of memory stress parameters based on their effectiveness at provoking weak behaviours in these litmus tests. While we focus on the MP, LB and SB tests, our stress may be tuned to any set of litmus tests. If, in the future, weak memory model bugs are discovered that correspond to communication idioms not covered by these three tests, our stress can be re-tuned.

Recall from Fig. 2 that each of the litmus tests involves two communication locations, `x` and `y`. In an application that implicitly uses one of these idioms, the relative addresses of communication locations depends on the data layout of the application. For a litmus test $T \in \{\mathsf{MP}, \mathsf{LB}, \mathsf{SB}\}$, we thus consider a variety of test instances, T_d, where d is a non-negative *distance* indicating the number of memory words separating the communication locations. We seek memory stress parameters capable of provoking weak behaviours in litmus tests for a range of distances, to account for the unknown distance between relevant locations in applications.

Because we consider applications with inter-block communication, tests are configured with `x` and `y` in global memory, and communicating threads in distinct blocks.

3.2 Identifying Effective Locations to Stress

In our pilot experiments, we found that the choice of which scratchpad locations to stress greatly influenced the extent of



(a) GTX Titan shows patch size 32 (b) Tesla C2075 shows patch size 64 (c) GTX 980 shows patch size 64

Figure 3: Patch-finding results for MP and LB

observed weak behaviours. On Titan we found it was roughly equally effective to stress any one of the first "patch" of 32 scratchpad (word-sized) locations, equally effective to stress any one of the next patch of 32 scratchpad locations, and so on, but that the effectiveness varied between patches.

We now explain our empirical method for discovering whether a chip naturally exhibits a patch size, so as to avoid redundantly stressing multiple locations in the same patch.

Let $\langle T_d, l \rangle$ denote test instance T_d with memory stress applied at scratchpad location l. For example, $\langle \mathsf{LB}_0, 5 \rangle$ denotes an instance of LB with contiguous communication locations and stressing applied at scratchpad location 5. Let D and L denote a maximum distance and scratchpad location to be considered, respectively, and let C be an *execution limit*. For each $T \in \{\mathsf{MP}, \mathsf{LB}, \mathsf{SB}\}$, $0 \leq d < D$ and $0 \leq l < L$, we conduct C executions of test $\langle T_d, l \rangle$, recording the number of times that $\langle T_d, l \rangle$ yields weak behaviour.

We chose $D = 256$, $L = 256$ and $C = 1000$, leading to $\sim$196.6M test executions per GPU. Each execution employs a random number of stressing threads such that the total number of threads executing the kernel is 50% to 100% of the maximum threads that can run concurrently on the GPU. Each stressing thread executes a loop where, on every iteration, the thread stores to and then loads from location l.

The plots of Fig. 3a illustrate the results of our experiments for Titan (Kepler architecture). Results for the MP and LB tests are shown; the results for SB are very similar to those for LB and are omitted. We show plots for three values of d: 0, 32 and 64. The x axis is divided into $L = 256$ segments. For each segment position x, a vertical bar is plotted. The height of the bar indicates the number of weak behaviours that were observed during $C = 1000$ executions of test $\langle T_d, x \rangle$ (to avoid cluttering the figure we do not number the y axis for these plots). In many cases no bar is visible, indicating that no weak behaviours were observed. The plots of Figs. 3b and 3c show similar data for C2075 (Fermi) and 980 (Maxwell) chips, respectively, for $d \in \{0, 64, 128\}$. The raw data and graphs for all experimental results can be found at `http://multicore.doc.ic.ac.uk/projects/gpuwmmtesting/`.

The Titan and C2075 results (Figs. 3a and 3b, respectively) exhibit similar characteristics: no weak behaviour is observed when communication locations are contiguous ($d = 0$); our full data shows that this is the case for all $d < 32$ (Titan) and $d < 64$ (C2075). After this, *patches* of weak behaviour emerge: for $d = 32$ and $d = 64$, Fig. 3a shows that the rate of weak behaviours exhibited by Titan is fairly consistent when stressing within 32-word contiguous scratchpad regions, but varies between different regions. For $33 \leq d < 64$, our full test data shows visually similar plots to the $d = 32$ plot of Fig. 3a, with stressing exposing weak behaviour in the same regions. These regions change at $d = 64$, remain constant for $65 \leq d < 96$, change again at $d = 96$, etc. Similar results are found for C2075, (Fig. 3b) but the patch size is 64.

Results clearly indicating a patch size are observed for all chips, with the exception of the recent 980 (Maxwell). Fig. 3c shows that for 980, we consistently observe a small number of MP weak behaviours for all stressing locations, even with $d = 0$; we see LB weak behaviours universally for $64 \leq d < 128$ (the figure shows the $d = 64$ case), and patches of length 64 emerge for LB at $d = 128$. We observe only noise levels of MP weak behaviour for $d \leq 256$, but after running some extra experiments on this chip we found that more significant MP weak behaviours emerge from $d = 256$ (not depicted in the figure), showing a patch size of 64.

The plots of Figs. 3a and 3b provide intuition for the idea that a GPU exhibits a natural *patch size*, which can differ between architectures. The 980 results of Figs. 3c also indicate that a minimum threshold of weak behaviour may need to be considered to identify the patch size for a chip.

We now formalise our method for determining the patch size of a chip empirically. Let $T \in \{\mathsf{MP}, \mathsf{LB}, \mathsf{SB}\}$ be a test,

Table 2: Stressing parameters and time spent tuning

chip	c. patch size	sequence	spread	~time (mins)
980	64	ld^4 st	2	1731
K5200	32	ld^3 st ld	2	3069
Titan	32	ld st^2 ld	2	3115
K20	32	ld st^2 ld	2	4215
770	32	st^2 ld^2	2	1831
C2075	64	ld st	2	2145
C2050	64	ld st	2	1996

and ϵ a non-negative *noise threshold*. A maximal contiguous sequence of P locations, $l_1, \ldots, l_P$, for which each $\langle T_d, l_i \rangle$ yields more than ϵ weak behaviours, is called an *ϵ-patch* of size P. A test may exhibit multiple ϵ-patches for a given P. If all of MP, LB and SB agree on the value of P for which the largest number of ϵ-patches of size P are observed, we call P the *critical patch size* for the GPU (w.r.t. to ϵ).

In our experiments we used a noise threshold of 3. We found that each GPU exhibited a critical patch size of either 32 or 64; specifically, 32 for Kepler chips, and 64 for Fermi chips. For Maxwell (980), critical patches for MP did not appear except in our special additional tests (where $D > 256$). Because of this, we say the critical patch size for 980 is 64, based on the patterns observed in the extra tests and the patch sizes observed for LB. The critical patch sizes for all chips are summarised in Tab. 2.

For brevity, we henceforth omit ϵ when discussing critical patch sizes.

3.3 Identifying Effective Access Sequences

We now turn to deriving an effective sequence of instructions to be issued by stressing threads; our pilot experiments indicated that this could influence exposure of weak behaviour.

Letting ld and st denote *load* and *store* instructions, respectively, we assess the effectiveness of stressing using a variety of access sequences. We do this by instantiating the loop body executed by the stressing threads with each access sequence σ matching the regular expression $(\mathsf{ld}|\mathsf{st})^+$, up to some maximum length N. For a litmus test T, access sequence σ, distance d $(0 \leq d < D)$ and stressing location l $(0 \leq l < L)$, let $\langle T_d, \sigma@l \rangle$ denote the test T instantiated with distance d between communication locations, and with access pattern σ used to apply memory stress at location l.

Suppose P is the critical patch size for the GPU of interest. Because stressing multiple locations in a patch is not worthwhile, we consider stressing each location in the set $\{l : 0 \leq l < L \wedge P|l\}$, i.e. the first location in each critical patch-sized region. For each such location l, we count the number of weak behaviours observed during C executions of $\langle T_d, \sigma@l \rangle$, for each test T, distance d and access sequence σ.

The *total* number of weak behaviours observed for test T with access sequence σ, summed over all distances and stressing locations, allows us to order the effectiveness of the access sequences with respect to T. An access sequence σ is *maximally effective* for a set of GPU test data if no other sequence σ' is observed to be more effective than σ with respect to all three litmus tests; that is, σ is *Pareto optimal* over the litmus tests. In our experiments we occasionally observed two distinct access sequences were maximally effective; we were able to break such ties by selecting the sequence that was most effective for two out of the three litmus tests. After tie-breaking, we have a single *most effective* access sequence for the GPU.

Table 3: Snippet of σs and scores for Titan

	MP			LB			SB	
rank	σ	score	rank	σ	score	rank	σ	score
1	ld^3 st ld	153k	1	st^2 ld^3	98k	1	st^2 ld	138k
2	st ld^2	140k	2	st ld^3 st	96k	2	ld st^2	128k
3	ld st ld	131k	3	ld^3 st^2	93k	3	ld^2 st^3	125k
...	...	...	...	...	...	...	...	
17	ld st^2 ld	104k	17	ld st^2 ld	64k	21	ld st^2 ld	93k
...	...	...	...	...	...	...	...	
61	st	342	61	st^5	126	61	st^3	674
62	st^3	266	62	st^2	108	62	st^4	520
63	st^5	232	63	st^3	90	63	st^5	348

In our experiments we chose $N = 5$ as the maximum access sequence length (leading to $2^{n+1} - 1 = 63$ possible access sequences), and $D = 256$, $L = 256$ and $C = 1000$ as before. With three litmus tests this required running $\sim$387.1M tests for a GPU with critical patch size $P = 32$ ($\sim$193.5M with critical patch size $P = 64$).

Table 2 shows the most effective sequences for our GPUs, based on our experimental findings (where ld^x denotes a sequence of x loads, st^x is similar). The most effective sequences match for both Fermi chips (C2075, C2050). For two of our four Kepler chips (Titan, K20) the most effective sequence is ld st^2 ld, which is equivalent under rotation to st^2 ld^2, the most effective sequence for one of the other Kepler chips (770). We observe that all of the *most effective* sequences involve a combination of loads and stores.

In Tab. 3 we give a snapshot of results for Titan. For the given σ, *score* shows the number of weak behaviours observed for the associated test using σ, over all distances and stressed locations. We show the top- and bottom-three σs for each test (ranked by score). The disparity between high and low scores provides evidence that σ influences the stressing effectiveness. The *most effective* sequence for the chip (shown in the middle) is not especially highly ranked for any one test, but is orders of magnitude more effective than the lowest-ranked σs. For most chips, the lowest ranked σs consist exclusively of stores.

Because access sequences are executed in a loop, we hypothesised that it might be redundant to test two σs that are equivalent under rotation, e.g. (ld st) and (st ld). However, our results show that two σs that are equivalent under rotation can yield remarkably different amounts of weak behaviour. For example, on Titan, $\sigma = (\mathsf{ld\ st})$ gave a score of 79K, while $\sigma = (\mathsf{st\ ld})$ gave a score of 91K. These differences may be

due to interference from loop boundary instructions. Thus, we opted to test all instruction permutations and did not consider rotational equivalence.

3.4 Identifying How Many Locations to Stress

Our patch testing results show that some critical patch-sized regions yield no weak behaviour while others yield a lot of weak behaviour, varying between distances. Because applications may exhibit arbitrary distances between communication locations, it may be sensible to select multiple regions to stress, to increase the probability of stressing a region that is effective for the application.

Our pilot experiments showed that simultaneously stressing a *spread* of 2–8 randomly chosen locations in different regions worked well, varying between chips. We now consider how to systematically derive an effective spread.

Let T be a test, σ an access sequence, d a distance and $\mathcal{L}$ a *set* of scratchpad locations. We use $\langle T_d, \sigma@\mathcal{L}\rangle$ to denote the litmus test T instantiated with distance d between communication locations, and with memory stress applied simultaneously at *each* location in $\mathcal{L}$ with respect to access sequence σ. The number of stressing threads is chosen randomly, as before, but at least $|\mathcal{L}|$ threads are used, and the threads are assigned evenly (modulo rounding) to the locations in $\mathcal{L}$.

To identify an effective spread, we consider a scratchpad of size $P \cdot M$, where P is the critical patch size of the GPU under consideration, and M is a positive *maximum spread*. This yields M distinct critical patch-sized regions to which stressing can be applied. The set $\mathcal{M} = \{l : 0 \leq l < P \cdot M \wedge P|l\}$ provides the first location in each region.

Let D denote the maximum distance between communication locations, as before, and let σ be the most effective access sequence for the GPU under consideration. For each test T and *spread* m $(1 \leq m \leq M)$ we execute C tests of the form $\langle T_d, \sigma@\mathcal{L}_m\rangle$, where for each test $\mathcal{L}_m$ is a randomly selected subset of $\mathcal{M}$ with size m, so that stressing is applied to m distinct critical patch-sized regions.

We use *score* for spread m to refer to the number of weak behaviours observed for test T with spread m, summed over all distances. A spread m is *maximally effective* for a set of GPU test data if it is Pareto optimal with respect to the idioms (i.e., if no other spread m' was observed to have a higher score than m with respect to all three litmus tests). Our experiments identified a single maximally effective m for each GPU, with no tie-breaking required.

In our experiments we chose $M = 64$ as the maximum spread, and $D = 256$ and $C = 1000$ as before. With three litmus tests, this required running $\sim$49.2M tests per chip.

Table 2 shows that 2 is the most effective spread for *all* the GPUs we tested. Figure 4 illustrates spread-finding results in more detail for 980 and K20, plotting spread on the x-axis and score on the y-axis. For 980 we see that 2 is clearly the most effective spread. The U-shape for K20 is less striking, yet the highest scores remain with a spread of 2.

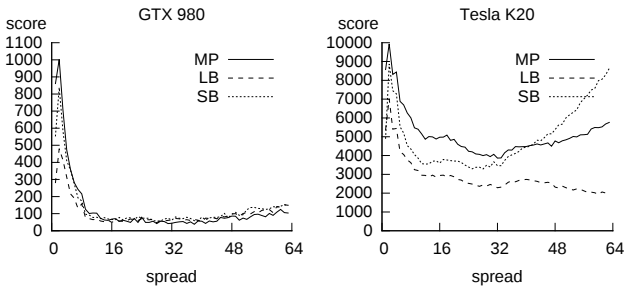


Figure 4: Spread finding for 980 and K20

3.5 Thread Randomisation

To amplify the effectiveness of memory stressing, we apply a straightforward adaption of the *thread randomisation* heuristic, originally presented in [8]. Previously used exclusively in the context of litmus testing, we extend thread randomisation to apply to black box application testing.

In this heuristic, the GPU thread ids are randomised, but constrained to honour the GPU programming model. Namely, randomisation must respect *block membership*: if two threads share a common block before randomisation, they must share a common (but possibly different) block after randomisation. This is vital if the application uses barriers, as placing previously co-located threads into different blocks can induce barrier divergence. Randomisation must also respect *warp membership*, as applications may exploit implicit intra-warp synchronisation to prevent what might otherwise be erroneous interleavings (e.g. [32] exploits this implicit intra-warp synchronisation).

For our case studies, we evaluate the effectiveness of thread randomisation at revealing weak memory bugs when applied in isolation and in conjunction with memory stress.

Summary We used nearly half a billion micro-benchmark executions to identify the critical patch size, an effective access sequence, and the best number of critical patch-sized regions to stress simultaneously, for each GPU. Results of our findings for the GPUs of Tab. 1 are given in Tab. 2, and full experimental data is given in our companion material. While we focused on the MP, LB and SB tests in this work, our stress can be tuned to other litmus tests. For litmus tests that also use two communication locations, our approach applies directly. For litmus tests with additional communication locations, tuning would need to consider multiple distance parameters. While conceptually straightforward, the tuning time for tests with more distance parameters may become expensive.

Combined with thread randomisation, we now evaluate the effectiveness of this test environment at provoking weak behaviours in real-world GPU applications.

4. Provoking Weak Behaviours in the Wild

We evaluate the testing environment of Sec. 3 w.r.t. ten GPU applications that are known to use fine-grained concurrency.

We detail the applications (Sec. 4.1), describe our experimental setup (Sec. 4.2) and present our findings (Sec. 4.3).

4.1 Application Case Studies

We undertook a thorough, best-effort search for CUDA applications that might be subject to weak behaviours, selecting applications that met three criteria: (1) their source code is available, (2) they do not rely on non-portable assumptions, (3) they appear to exhibit fine-grained concurrency.

We omit applications described in [20, 47] due to source code unavailability, and some applications of [12, 37, 49] as they use a non-portable *global barrier* that depends on precise occupancy; adding extra stressing blocks and applying thread randomisation to this construct causes deadlock.

We assess criterion (3) according to whether applications use mutexes or concurrent data-structures (with which weak behaviours are often associated, e.g. [26, 36]), or include fence instructions (an explicit acknowledgment of memory consistency issues). Most CUDA applications *do not* meet this criterion and exhibit no communication between thread blocks, as they use barriers for intra-block communication; thus they are not prone to weak behaviours. In these cases, our approach would provide no benefit. The subset of relevant applications is small, but the applications are important, and as interest in the use of fine-grained concurrency increases we expect the set of relevant applications to grow.

Our approach requires each application to be equipped with a user-supplied functional post-condition, to check whether an execution is erroneous; because applications may exhibit nondeterminism it is not sufficient to check for repeated computation of an identical result. We omitted several candidate applications for which, as outsiders, we could not easily derive suitable post-conditions. For example, GPU hashtables in [2, 35] drop items if collisions exceed a certain threshold; we found it hard to formulate a robust post-condition to capture the intended behaviour. Other such examples were found in [22, 33].

The evaluated applications are summarised in Tab. 4, a total of ten applications derived from seven code bases (with three variants obtained by removing existing fences). We detail the application source, the nature of communication, and the post-condition used to check correctness. All applications except for *ct-octree*, *tpo-tm* and *ls-bh* were provided with a testing harness containing a post-condition. For *ct-octree* and *tpo-tm*, meta-data were gathered during the execution and used to implement post-condition checks. For *ls-bh*, we obtained a reference solution from the conservatively fenced variation of the application (see Sec. 5). The post-condition compares the computed values with the reference.

While our environment was developed to expose bugs due to weak behaviours, other types of bugs were uncovered in our case studies: improper memory initialisation in *ct-octree* and out-of-bounds queue accesses in *ct-octree* and *tpo-tm*. Our experiments are performed on patched versions of these applications, not showing these issues.

The applications *ls-bh-nf*, *cub-scan-nf*, and *sdk-red-nf* are manufactured from *ls-bh*, *cub-scan*, and *sdk-red* respectively. The original applications contained fence instructions which we removed to create the *-nf* (no fence) variants. This allows us to test if the provided fences are (a) experimentally needed to disallow errors and (b) sufficient to disallow weak behaviour bugs in the application.

4.2 Experimental Setup

Testing environments We evaluate the effectiveness of our systematically designed memory stressing strategy, *sys-str*, by considering two straightforward memory stressing strategies to compare against. In the first strategy, *rand-str*, stressing threads repeatedly execute a load or store (at random) to a random location in the scratchpad (using *curand* [42] to generate random numbers in CUDA). The second strategy, *cache-str*, allocates a scratchpad the size of the GPU L2 cache, and each stressing block then repeatedly performs a load and store to each location in the scratchpad. The L2 cache-sized scratchpad is an attempt to exploit a hardware parameter of the chip that we speculate may be relevant for triggering weak behaviours. For example, reading and writing to a scratchpad of this size may cause cache thrashing, which we hypothesised might encourage weak behaviours to appear. We also consider *no-str*: the application is executed natively, without any memory stress.

To evaluate thread randomisation, we experiment with each stressing strategy both with thread randomisation enabled and disabled, indicated by a + (enabled) or - (disabled) at the end of the stressing strategy name. For example, *sys-str+* denotes systematic stressing with thread randomisation enabled. This leads to a total of eight testing environments.

Testing parameters Natively, we find that application executions terminate within 8 seconds across all our GPUs, dominated by initialisation of the CUDA framework with kernel execution itself accounting for a small fraction of total time. To catch timeout errors (due to weak behaviours affecting the termination condition of an application), we set a timeout limit of 30 seconds per application execution. When memory stress is enabled, we randomly set the number of stressing blocks to a value between 15% and 50% of the number of thread blocks launched by the original application. To ensure that stressing is applied for at least as long as a kernel would normally execute, we configure the number of stressing loop iterations on a per application basis so that the stressing threads execute for roughly 10 times as long as the kernel takes to execute. Because kernel execution accounts for only a fraction of total execution time for an application, this has little impact on overall execution time.

For each combination of GPU (Tab. 1), application (Tab. 4) and testing environment, we repeatedly execute the application for one hour and record the number of erroneous runs observed. The number of executions varies between combina-

Table 4: The ten case studies we consider, derived from seven distinct applications

short name	description	communication	post-condition
cbe-ht	Concurrent hashtable given in the book *CUDA by Example* [45, ch. A1.3]	Concurrent hashtable insertion protected by custom mutexes	All elements inserted into the hashtable are in the final hashtable
cbe-dot	Dot product routine given in the book *CUDA by Example* [45, ch. A1.2]	Global final reduction across blocks protected by a custom mutex	GPU result matches a CPU reference result
ct-octree	Octree partitioning routine by Cederman and Tsigas [22, ch. 37]	Concurrent access to non-blocking queues	All original particles are in final octree
tpo-tm	Dynamic task management framework by Tzeng, Patney, and Owens [48]	Concurrent access to queues protected by custom mutexes	Expected number of tasks are executed
sdk-red *	Reduction routine from the CUDA 7 SDK [39]	Last block (via atomic counter) combines block-local results	GPU result matches a CPU reference result
cub-scan *	Prefix scan from the CUB GPU library [37]	Blocks communicate partial results using MP-style handshake	GPU result matches a CPU reference result
ls-bh *	Barnes-Hut N-body simulation from the Lonestar GPU benchmarks [12]	Various instances across three kernels	Final particle positions match results from reference implementation

**These apps. contain fence instructions; we also consider variants without fences, using the names* sdk-red-nf*,* cub-scan-nf*, and* ls-bh-nf

tions, ranging from 160 for Titan/*ct-octree*/*sys-str+* to 10,907 for 980/*cbe-dot*/*no-str*.

4.3 Results

The results of running combinations of environments, chips, and applications are summarised in Tab. 5. For each chip and environment, a/b means that we could observe erroneous runs for b applications, and that in a of these cases we observed errors in more than 5% of the executions, in which case we say that the environment is *effective* in exposing errors for the application and chip. We highlight in bold the most effective strategy for each chip. For example, *sys-str+* is the most effective strategy for K20, observing errors for eight applications, and exceeding the effectiveness threshold for seven.

Observed errors We observed weak behaviour in all applications except *sdk-red* and *cub-scan*. Because we observe weak behaviours in the fenceless versions of these applications (*sdk-red-nf* and *cub-scan-nf*), it appears that the fences included in the original applications do prevent errors. In contrast, we observed errors in both *ls-bh* and *ls-bh-nf*, showing that the fences included in *ls-bh* are insufficient.

We observe errors natively (*no-str-*) only for 3 chip-application combinations: 770-*cbe-ht*, K5200-*cub-scan-nf* and C2075-*ls-bh*. Only Titan, using the *sys-str-* environment was effective at exposing errors in *sdk-red-nf*

Comparing strategies Environments with *sys-str* stress are always more capable (show errors in more applications and are effective in more applications) than any of the other stressing strategies. In many cases, other stressing strategies are only able to reveal weak behaviours in fewer than two applications, and are only effective for one application: *cbe-ht*. We did not find any applications for which *cache-str*, *rand-str* or *no-str* were able to reveal (or effectively reveal) errors not revealed by *sys-str*.

With the exception of Titan, *sys-str+* is the most effective environment for every chip. For Titan, the *sys-str-* strategy is more effective than *sys-str+* for one application. On all chips except 980, *sys-str+* revealed weak behaviours in every application for which any environment could reveal weak behaviours, and is effective for most applications.

Effectiveness of thread randomisation Thread randomisation led to modest increases in effectiveness in most cases; although there were several exceptions, e.g. 980/*rand-str*.

Reported bugs The errors for *ct-octree*, *cbe-dot*, and *cbe-ht* provide empirical witnesses for the bugs reported via hand analysis in [8]. The unreported errors in *ls-bh* and *tpo-tm* have been acknowledged by the authors.

5. Program Hardening

We now turn to the problem of experimentally suppressing weak memory bugs by adding memory fences. While inserting fences to disallow weak memory bugs has been studied for CPUs (Sec. 7), these methods require a formal memory model (e.g. as provided by the GPU languages OpenCL 2.0 [24] and HSA [21]). Though foundations of a model for PTX [44] (the intermediate representation underlying CUDA) have been proposed [8], there is no agreed memory model for CUDA. Thus we have no means of formally validating the necessity of fences. As a pragmatic alternative, we employ our testing to suggest a minimal set of fences that suffice to disallow weak memory bugs under our aggressive testing environment, a process we call *empirical fence insertion*; as discussed in Sec. 1 this can aid in understanding weak memory bugs, and in *hardening* applications against such bugs. In future, the suggested fences could be used as a starting point for a verification effort if a suitable memory model (and accompanying verification technique) become available.

Table 5: Summary of the effectiveness of the testing environments we consider with respect to the GPUs of Tab. 1

chip	*no-str-*	*no-str+*	*sys-str-*	*sys-str+*	*rand-str-*	*rand-str+*	*cache-str-*	*cache-str+*
980	0 / 0	0 / 0	1 / 6	**4 / 7***	0 / 2	0 / 0	0 / 2	0 / 2
K5200	1 / 1	0 / 0	5 / 7*	**7*****/ 8***	1*/ 1*	1*/ 2	1*/ 1*	1*/ 1*
Titan	0 / 0	0 / 0	**8*****/ 8***	7*/ 8*	0 / 2	0 / 2	1*/ 1*	1*/ 2
K20c	0 / 0	0 / 0	7*/ 7*	**7*****/ 8***	1*/ 1*	0 / 1*	1*/ 1*	1*/ 1*
770	1*/ 1*	0 / 1*	5 / 8*	**6 / 8***	1*/ 1*	1*/ 1*	0 / 1*	1*/ 1*
C2075	0 / 1	1 / 2	5 / 8*	**6 / 8***	1*/ 2	1*/ 3	1*/ 2	1*/ 4
C2050	0 / 0	1 / 3	5 / 7*	**6 / 8***	1*/ 2	1*/ 3	1*/ 1*	1*/ 2

Common sets of applications:

1* – The lone application is *cbe-ht*

8* – All applications except *sdk-red* and *cub-scan*

7* – All applications except *sdk-red*, *sdk-red-nf* and *cub-scan*

Algorithm 1 Empirical fence insertion, based on binary and linear reduction procedures

input: a target application A, a set of empirically stable fences F, an iteration count I
output: a minimal set of empirically stable fences

```
1:  procedure EMPERICALFENCEINSERTION(A, F, I)
2:      do
3:          F_b ← BINARYREDUCTION(A, F, I)
4:          F_l ← LINEARREDUCTION(A, F_b, I)
5:          I ← 2 · I
6:      while ¬EmpiricallyStable(A, F_l)
7:      return F_l

8:  procedure LINEARREDUCTION(A, F, I)
9:      for f ∈ F do
10:         if CheckApplication(A, F \ {f}, I) then
11:             F ← F \ {f}
12:     return F

13: procedure BINARYREDUCTION(A, F, I)
14:     while |F| > 1 do
15:         F_1, F_2 = SplitFences(F)
16:         if CheckApplication(A, F_1, I) then
17:             F ← F \ F_1
18:         else if CheckApplication(A, F_2, I) then
19:             F ← F \ F_2
20:         else
21:             return F
22:     return F
```

We comment on the relation between the fences found by our insertion method and fences (a) prescribed by prior hand analysis and (b) already present in the original application.

5.1 Empirical Fence Insertion

We say that an application shows *empirically stable* behaviour if we observe no errors when executing the application for one hour under a testing environment (we use *sys-str+*). Given an application A containing no fences, let $A + F$ denote the application after adding a given set of fences, F. Our goal is to find a set of fences F such that (a) $A + F$ shows empirically stable behaviour, and (b) $A + F'$ does *not* show empirically stable behaviour for any $F' \subset F$, i.e. empirically unnecessary fences are not present in F. The procedure is detailed in Alg. 1 and described below.

Starting with an application A and fence set F such that $A + F$ shows empirically stable behaviour (in practice, we take F to be the set of fences inserted after every memory access), our empirical fence insertion attempts to reduce the size of F using *linear reduction* and *binary reduction*; each reduction takes an integer-valued *iteration* argument I.

Linear reduction (line 8) attempts to remove fences one at a time. For each fence $f \in F$ (line 9), $A + (F \setminus \{f\})$ is executed for I iterations (line 10). This is done through the $\mathsf{CheckApplication}(A, F, I)$ function, which executes application $A + F$ for I iterations and returns $true$ if no errors are observed and $false$ otherwise. If no errors are observed, f is immediately removed from F (line 11). The method returns the updated set of fences F (line 12).

Binary reduction (line 13) iteratively tries to remove half of the remaining fences: while F contains more than one fence, the $\mathsf{SplitFences}(F)$ function is used to split F into halves, F_1 and F_2 (line 15). In our implementation, the fences are sorted based on their location in the code. The first half of the fences go to F_1 and the second half of the fences go to F_2. It may be possible to develop a more informed fence splitting strategy to fully exploit binary reduction; however, we leave such exploration to future work. Application $A + (F \setminus F_1)$ is executed for I iterations. If no errors are observed, F_1 is removed from F, and the process repeats (lines 16 and 17). Otherwise, $A + (F \setminus F_2)$ is executed for I iterations. If no errors are observed, F_2 is removed from F and the process repeats (lines 18 and 19). Otherwise, binary reduction terminates returning F (line 21). In the worst case, binary reduction removes no fences, if A empirically requires multiple fences that are split between F_1 and F_2.

At the top level, empirical fence insertion (line 1) uses binary reduction in an attempt to quickly reduce F, yielding a set of fences F_b (line 3) to which linear reduction is then applied, yielding a set of fences F_l (line 4). If $A + F_l$ is observed to be empirically stable, F_l is returned as a set of empirically required fences (lines 6 and 7). The $\mathsf{EmpiricallyStable}(A, F)$ function checks application $A + F$ for empirically stable behaviour (i.e. whether $A + F$ behaves correctly when repeatedly executed for one hour) and returns

Table 6: Empirical fence insertion results

app.	inserted fences init.	inserted fences red. (Titan)	agreeing chips	red. time (mins.) min	red. time (mins.) med	red. time (mins.) max
cbe-ht	10	1	5	80	106	127
cbe-dot	4	1	5	62	63	65
ct-octree	33	1	5	67	69	735
tpo-tm	28	1	4	63	67	124
sdk-red-nf	6	1	4	63	72	258
cub-scan-nf	51	2	4	90	116	1407
ls-bh-nf	90	4	0	235	343	t.o

$true$ if no errors are observed, and $false$ otherwise. If the empirical stability check fails, then the iteration count I used during reduction was not large enough. In this case, the reduction process restarts with the original fence set F and iteration count $2 \cdot I$ (line 5).

The role of iteration argument I is to attempt to accelerate fence insertion. Each call to CheckApplication could be replaced by a call to EmpiricallyStable, which does not use the iteration count I. Using CheckApplication may lead to faster convergence because it runs for I iterations, whereas EmpiricallyStable always runs for one hour. The call to EmpiricallyStable at line 6 of Alg. 1 ensures that the final set of fences returned by the algorithm is empirically stable; if the candidate fence set is not stable then the insertion procedure essentially restarts with a larger iteration count.

The fences returned by empirical fence insertion are dependent on the effectiveness of the testing environment and the order in which fences are removed. If the fences are considered deterministically (i.e. binary reduction splits the fence set deterministically and linear reduction iterates through the fences set in a strict order), and if CheckApplication is deterministically able to find bugs, then empirical fence insertion will deterministically return the same set of fences if applied multiple times to an application on a given GPU. However, because CheckApplication is based on testing, and is thus non-deterministic, the empirically stable fence set returned by empirical fence insertion may not be deterministic.

5.2 Results

We experiment with the applications that contain no fences (i.e. omitting *sdk-red cub-scan* and *ls-bh*). We use $I = 32$ initially and *sys-str+* as the testing environment (chosen based on its effectiveness in Sec. 4), using a 24h timeout per application.

For each application, Tab. 6 shows how many fences were provided in the initial state, i.e., when inserted after every memory access, and how many fences remained after the reduction methods converged on Titan (which often revealed errors most frequently). We show the number of cases (maximum of six) where fence insertion on other chips found the same fences as on Titan, as different chips may find different fences depending on how often testing reveals errors. The minimum, median, and maximum times (over the results for all GPUs) for the reduction processes are shown. Due to the large amount of time required to run these experiments, we did not perform multiple runs per GPU for this experiment.

In the case of all applications except *cub-scan-nf* and *ls-bh-nf*, insertion yielded a single fence on Titan. In most cases, the reduced fences found on other chips agree with the reduced fences found on Titan, showing that our method yields similar results across chips. The outlier chip is 770, which never found fences that agreed with Titan, often finding fences immediately following (in program order) the fences found on Titan. For example, in the *cbe-dot* application (Fig. 1), empirical fence insertion on the 770 placed the fence in the `lock` function (after line 19). All other chips placed the lock immediately prior to the `unlock` function (line 15). If we consider the `lock` and `unlock` functions inlined, the two fencing solutions are a single memory instruction apart (line 15). The fence solution found by 770 is incorrect based on the memory model proposed in [8], as a fence between the critical section access (line 15) and the unlock operation (line 22) is required to ensure that the next thread entering the critical section observes up-to-date values. We have no hypothesis for why 770 finds such fences and attribute it to a quirk of the chip.

The other chip that found different fences from the majority was 980, which found no fences for *sdk-red-nf* and only one of the two fences for *cub-scan-nf*. The outlier application is *ls-bh-nf*, on which insertion for all chips timed out except on Titan and K20. The K20 solution is a subset of the Titan solution, differing by one fence.

In six of the applications, at least half of the chips found a reduced solution within two hours (median), and the fastest took just over an hour (recall that one hour is required in order to check for empirical stability). However we observed cases where the insertion method was inefficient. The timeouts in *ls-bh-nf* are due to both the large number of initial fences and the location of required fences (found by Titan)—binary reduction was unable to remove fences at a course level of granularity.

Evaluating reduced results Here we discuss the fences found by empirical fence insertion (on Titan) and how they relate to existing hand analysis and fences existing in the original application.

Prior hand analysis prescribed two fences for *cbe-ht* and *cbe-dot* [8]. The inserted fence corresponds to one of these fences; the other prescribed fence is redundant with a dependency and was not found by insertion. The same hand analysis prescribed four fences for *ct-octree*, one of which corresponds to the inserted fence. The other prescribed fences were either redundant with dependencies or involved a sequence of data-structure operations not occurring in the actual application.

The two inserted fences for *cub-scan-nf* correspond exactly to the provided fences in *cub-scan*, giving us high confi-

dence in the empirical solution. The reduced fence for *sdk-red-nf* does not correspond to the provided fence in *sdk-red*; this solution may be consistent with a temporally bounded model [34] where extra instructions in one communicating thread can make up for the lack of fences in the other. The reduced fences for *ls-bh-nf* are a superset of the fences in *ls-bh* (as *ls-bh* showed errors with provided fences).

Because empirical fence insertion only hardens applications, it may give even empirically unsound results; e.g. 770 observes errors for *sdk-red-nf* (Tab. 5), but due to the infrequency of observed errors, empirical fence insertion suggested no fences.

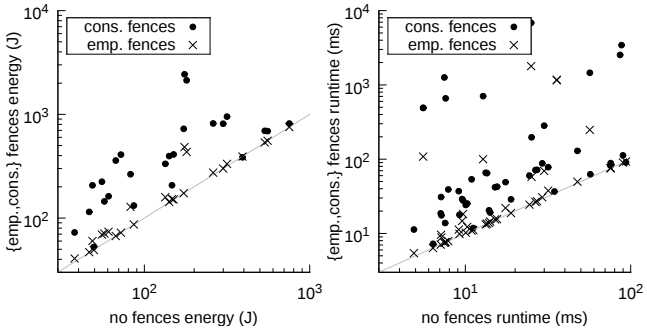


Figure 5: Cost of *reduced* vs. {*no*, *conservative*} fences

6. The Cost of Fences

To better understand the performance cost associated with fences in GPU applications, we benchmark the applications of Sec. 5 when run natively (i.e., without a testing environment) under two fencing configurations. We compare the runtime and energy overhead w.r.t. the application containing no fences. Runtime is measured using CUDA events [41, p. 56]. For energy, NVML [43] is used to query GPU power usage throughout the execution. The average power reading is multiplied by the kernel runtime to estimate energy usage. Only K5200, Titan, K20, and C2075 support power queries. There are known inaccuracies when measuring GPU power this way [13, 16], thus we emphasise that our energy results are estimates. Results are averaged over 100 runs.

The two fencing strategies we consider are: a conservative fence strategy where a fence is placed after every memory access (*cons* fences) and the fences found during empirical fence insertion (*emp* fences). We consider *emp* fences on a per GPU basis, that is, a given GPU uses the fences it found during empirical fence insertion. Thus, for a given application, *emp* fences may be different across GPUs. We compare an application with these fencing strategies applied to the application without fences (*no* fences). We record performance results only if the application passes the post-condition, otherwise the run is discarded and not counted as part of the 100 runs. Because applications rarely exhibit weak behaviours when run naively (see Sec. 4), this was not an issue.

Figure 5 shows two scatter plots (logarithmic scale): the left graph shows energy consumption (in J) and the right graph shows runtime (in ms). Each point on the graph is a chip/application combination. A cross (resp. dot) with coordinates (x, y) indicates that execution consumed an estimated x J (left) or took x ms (right) with no fences, and an estimated y J (left) or y ms (right) with *emp* fences (resp. *cons* fences). The distance between a point (above the diagonal) and the diagonal represents the cost of the fencing strategy. Points close to the diagonal indicate a low cost associated with inserting fences, while points further away indicate that fences carry a higher cost.

There are a total of 93 and 54 data points for runtime and energy respectively. The graphs omit ten outlier points for runtime and three for energy. Unsurprisingly, we see no points below the diagonal, showing fences never decrease cost. We see that generally *cons* fences cost more than *emp* fences, given that dots appear further from the diagonal than the crosses. Runtime costs corresponds closely to energy costs (consistent with findings in, e.g. [50] for CPU systems). We comment on extreme cases for runtime comparisons (energy comparisons are similar) and give the median for both runtime and energy costs.

Comparing the cost of *cons* fences to *no* fences, we observe some dramatic results. The highest chip-application runtime cost is 35,120% for C2075/*cbe-ht*. In fact, for the three oldest chips (770, C2075 and 2050) we observe similarly high costs in several applications. The newer chips show a less dramatic cost, the highest being 843% for K20/*cbe-ht*. The median runtime and energy costs are 174% and 171% respectively.

Comparing the cost of *emp* fences to *no fences*, we observe substantially lower costs, which is to be expected given that *emp* fences are a subset of *cons* fences. The highest cost is 7052% for 770/*cbe-ht*. Like the previous comparisons, the oldest three chips have fairly extreme results; excluding these chips the highest cost is 131% for K20/*cbe-ht*. The median runtime and energy cost of *emp* fences are both very small (less than 3%). This can be seen on the graphs of Fig. 5 as many crosses are very close to the diagonal.

7. Related Work

Memory model testing Testing for weak behaviours on hardware has largely been for litmus tests. The ARCHTEST [15] tool and TSOTOOL [19] ran tests on systems with the TSO memory model (e.g. x86 CPUs). The LITMUS tool [3] runs tests on x86, IBM Power, and ARM chips. Litmus testing has recently been applied to GPUs with the tool GPU LITMUS [8]. Our work is inspired by GPU LITMUS; the relation between this work and GPU LITMUS is detailed at the end of Sec. 1.

Fence insertion Alglave et al. [6] survey static methods for inserting fences to restore sequential consistency in CPU applications (e.g. [27]), evaluating each method based on the number of fences inserted and the associated runtime overhead. They propose a new method based on linear programming. Joshi and Kroening [23] use bounded model checking to insert fences, not to restore sequential consistency but, as in our work, to restore sufficient orderings to satisfy specifications of the application. Using a demonic scheduler that delays memory accesses, Liu et al. consider finding and suppressing weak behaviours-related errors through dynamic analysis and fence insertion [30].

For GPUs, Feng and Shucai [17] benchmark a global barrier implementation with and without fences. They report observing no errors when the fences are omitted and high runtime costs when fences are included.

GPU program analysis Current GPU program analysis tools focus on data-race freedom, barrier properties and memory safety: GKLEE [28] uses concolic execution, while GPUVERIFY [10, 11] is based on verification conditions and invariant generation. Extensions of these methods support atomic operations to a limited extent [9, 14], but neither provides a precise analysis accounting for weak behaviours. The CUDA-MEMCHECK [40] tool, provided with the CUDA SDK, dynamically checks for illegal memory accesses and data-races, but does not account for weak memory effects.

Weak memory program analysis Several methods exist to analyse programs under CPU memory models. The CDSCHECKER tool [36] buffers loads and stores and is configured to simulate the C++11 memory model. The bounded model checker CBMC supports reasoning about weak memory either by transforming code to simulate weak effects, after which an analysis that assumes sequential consistency can be applied [4], or via a partial order relaxation of interleavings [5]. The JUMBLE [18] tool creates an execution environment which intentionally provides stale values (simulating weak behaviours) attempting to crash applications.

8. Conclusions and Future Work

We presented a testing environment, systematically designed through micro-benchmarking, that is effective in exposing weak behaviours in GPU applications. Our testing environment can be used for fence insertion, aiding in understanding and repairing weak memory bugs.

We see several avenues for future investigation: (a) the design of GPU weak memory-aware formal program analysis techniques (to enable verification of our empirically proposed fixes); (b) architectural investigation of the critical patches revealed by our micro-benchmarks, building on the insights into Nvidia GPUs provided by GPGPU-Sim [1]; (c) applying and evaluating our methods to CPU architectures and applications; (d) combining our techniques with recent methods for GPU compiler testing [29] in order to check the correctness of compilation in the presence of fine-grained concurrency; (e) combining our techniques with race detectors to help pinpoint communication idioms in applications and developing targeted testing around these locations.

Acknowledgments

We are grateful to: John Wickerson, Pantazis Deligiannis, Kareem Khazem and Peter Sewell for feedback and insightful discussions; Jade Alglave for discussion and support during the inception of this project; Ganesh Gopalakrishnan (Utah) and the UCL high performance computing lab (in particular Tristan Clark) for providing access to GPU resources; Stanley Tzeng and John Owens for providing the code and discussions around the *tpo-tm* application; Sreepathi Pai and Martin Burtscher for discussions around the *ls-bh* application; the PLDI reviewers for their thorough comments and feedback, and Brandon Lucia for shepherding the paper; GCHQ for funding the purchase of equipment used in this study.

References

[1] T. M. Aamodt and W. W. Fung. GPGPU-Sim 3.x manual, 2015. http://gpgpu-sim.org/manual/index.php/GPGPU-Sim_3.x_Manual.

[2] D. A. F. Alcantara. *Efficient Hash Tables on the GPU*. PhD thesis, University of California, Davis, 2011.

[3] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Litmus: Running tests against hardware. TACAS, pages 41–44. Springer, 2011.

[4] J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig. Software verification for weak memory via program transformation. In *ESOP*, pages 512–532. Springer, 2013.

[5] J. Alglave, D. Kroening, and M. Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *CAV*, pages 141–157. Springer, 2013.

[6] J. Alglave, D. Kroening, V. Nimal, and D. Poetzl. Don't sit on the fence - A static analysis approach to automatic fence insertion. In *CAV*, pages 508–524. Springer, 2014.

[7] J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, 2014.

[8] J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson. GPU concurrency: Weak behaviours and programming assumptions. In *ASPLOS*, pages 577–591. ACM, 2015.

[9] E. Bardsley and A. F. Donaldson. Warps and atomics: Beyond barrier synchronization in the verification of GPU kernels. In *NFM*, pages 230–245. Springer, 2014.

[10] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson. GPUVerify: A verifier for GPU kernels. In *OOPSLA*, pages 113–132. ACM, 2012.

[11] A. Betts, N. Chong, A. F. Donaldson, J. Ketema, S. Qadeer, P. Thomson, and J. Wickerson. The design and implementation of a verification technique for GPU kernels. *ACM Trans. Program. Lang. Syst.*, 37(3):10, 2015.

[12] M. Burtscher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on GPUs. In *IISWC*, pages 141–151. IEEE, 2012.

[13] M. Burtscher, I. Zecena, and Z. Zong. Measuring GPU power with the K20 built-in sensor. GPGPU, pages 28–36. ACM, 2014.

[14] W. Chiang, G. Gopalakrishnan, G. Li, and Z. Rakamaric. Formal analysis of GPU programs with atomics via conflict-directed delay-bounding. In *NFM*, pages 213–228. Springer, 2013.

[15] W. W. Collier. *Reasoning About Parallel Architectures*. Prentice-Hall, Inc., 1992. ISBN 0-13-767187-3.

[16] J. Coplin and M. Burtscher. Power characteristics of irregular GPGPU programs. Workshop on Green Programming, Computing, and Data Processing (GPCDP), 2014.

[17] W. Feng and S. Xiao. To GPU synchronize or not GPU synchronize? In *International Symposium on Circuits and Systems (ISCAS)*, pages 3801–3804, 2010.

[18] C. Flanagan and S. N. Freund. Adversarial memory for detecting destructive races. PLDI, pages 244–254. ACM, 2010.

[19] S. Hangal, D. Vahia, C. Manovit, and J.-Y. J. Lu. Tsotool: A program for verifying memory systems using the memory consistency model. ISCA '04. IEEE Computer Society, 2004.

[20] B. He and J. X. Yu. High-throughput transaction executions on graphics processors. *Proc. VLDB Endow.*, 4(5):314–325, 2011.

[21] V. . HSA Foundation. HSA programmer's reference manual, July 2015. http://www.hsafoundation.com/html/Content/PRM/Topics/PRM_title_page.htm.

[22] W.-m. W. Hwu. *GPU Computing Gems Jade Edition*. Morgan Kaufmann, 2011.

[23] S. Joshi and D. Kroening. Property-driven fence insertion using reorder bounded model checking. In *FM*, pages 291–307. Springer, 2015.

[24] Khronos Group. OpenCL: Open Computing Language. http://www.khronos.org/opencl.

[25] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, pages 690–691, Sept. 1979.

[26] N. M. Lê, A. Pop, A. Cohen, and F. Zappa Nardelli. Correct and efficient work-stealing for weak memory models. PPoPP '13, pages 69–80. ACM, 2013. ISBN 978-1-4503-1922-5.

[27] J. Lee and D. A. Padua. Hiding relaxed memory consistency with a compiler. *IEEE Trans. Comput.*, 50(8):824–833, Aug. 2001.

[28] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan. GKLEE: concolic verification and test generation for GPUs. In *PPoPP*, pages 215–224. ACM, 2012.

[29] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson. Many-core compiler fuzzing. In *PLDI*, pages 65–76, 2015.

[30] F. Liu, N. Nedev, N. Prisadnikov, M. Vechev, and E. Yahav. Dynamic synthesis for relaxed memory models. In *PLDI*, pages 429–440, 2014.

[31] X. Mei and X. Chu. Dissecting GPU memory hierarchy through microbenchmarking. *CoRR*, abs/1509.02308, 2015. URL http://arxiv.org/abs/1509.02308.

[32] D. Merrill, M. Garland, and A. Grimshaw. Scalable GPU graph traversal. In *PPoPP*, pages 117–128. ACM, 2012.

[33] P. Misra and M. Chaudhuri. Performance evaluation of concurrent lock-free data structures on GPUs. ICPADS, pages 53–60. IEEE, 2012.

[34] A. Morrison and Y. Afek. Temporally bounding TSO for fence-free asymmetric synchronization. In *ASPLOS 2015*, pages 45–58. ACM.

[35] M. Nießner, M. Zollhöfer, S. Izadi, and M. Stamminger. Real-time 3D reconstruction at scale using voxel hashing. *ACM Trans. Graph.*, 32(6):169:1–169:11, 2013.

[36] B. Norris and B. Demsky. CDSchecker: Checking concurrent data structures written with C/C++ atomics. In *OOPSLA*, pages 131–150. ACM, 2013.

[37] Nvidia. CUB, April 2015. http://nvlabs.github.io/cub/.

[38] Nvidia. CUDA C programming guide, version 7, March 2015. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[39] Nvidia. CUDA Code Samples, 2015. https://developer.nvidia.com/cuda-code-samples.

[40] Nvidia. CUDA-memcheck, 2015. https://developer.nvidia.com/CUDA-MEMCHECK.

[41] Nvidia. CUDA runtime API, March 2015. http://docs.nvidia.com/cuda/pdf/CUDA_Runtime_API.pdf.

[42] Nvidia. cuRAND, version 7, March 2015. http://docs.nvidia.com/cuda/pdf/CURAND_Library.pdf.

[43] Nvidia. NVML reference manual, March 2015. http://docs.nvidia.com/deploy/pdf/NVML_API_Reference_Guide.pdf.

[44] Nvidia. Parallel thread execution ISA: Version 4.2, March 2015. http://docs.nvidia.com/cuda/pdf/ptx_isa_4.2.pdf.

[45] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010.

[46] D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011.

[47] J. A. Stuart and J. D. Owens. Efficient synchronization primitives for GPUs. *CoRR*, abs/1110.4623, 2011. URL http://arxiv.org/abs/1110.4623.

[48] S. Tzeng, A. Patney, and J. D. Owens. Task management for irregular-parallel workloads on the GPU. In *HPG*, pages 29–37, 2010.

[49] S. Xiao and W. Feng. Inter-block GPU communication via fast barrier synchronization. IPDPS, pages 1–12. IEEE, 2010.

[50] T. Yuki and S. Rajopadhye. Folklore confirmed: Compiling for speed = compiling for energy. In *Languages and Compilers for Parallel Computing*, pages 169–184. Springer, 2014.