Foundations of Empirical Memory Consistency Testing

JAKE KIRKHAM, Princeton University, USA
TYLER SORENSEN, UC Santa Cruz, USA
ESIN TURECI, Princeton University, USA
MARGARET MARTONOSI, Princeton University, USA

Modern memory consistency models are complex, and it is difficult to reason about the relaxed behaviors that current systems allow. Programming languages, such as C and OpenCL, offer a memory model interface that developers can use to safely write concurrent applications. This abstraction provides functional portability across any platform that implements the interface, regardless of differences in the underlying systems. This powerful abstraction hinges on the ability of the system to correctly implement the interface. Many techniques for memory consistency model validation use empirical testing, which has been effective at uncovering undocumented behaviors and even finding bugs in trusted compilation schemes. Memory model testing consists of small concurrent unit tests called "litmus tests". In these tests, certain observations, *including potential bugs*, are exceedingly rare, as they may only be triggered by precise interleaving of system steps in a complex processor, which is probabilistic in nature. Thus, each test must be run many times in order to provide a high level of confidence in its coverage.

In this work, we rigorously investigate empirical memory model testing. In particular, we propose methodologies for navigating complex stressing routines and analyzing large numbers of testing observations. Using these insights, we can more efficiently tune stressing parameters, which can lead to higher confidence results at a faster rate. We emphasize the need for such approaches by performing a meta-study of prior work, which reveals results with low reproducibility and inefficient use of testing time.

Our investigation is presented alongside empirical data. We believe that OpenCL targeting GPUs is a pragmatic choice in this domain as there exists a variety of different platforms to test, from large HPC servers to power-efficient edge devices. The tests presented in the work span 3 GPUs from 3 different vendors. We show that our methodologies are applicable across the GPUs, despite significant variances in the results. Concretely, our results show: lossless speedups of more than 5× in tuning using data peeking; a definition of portable stressing parameters which loses only 12% efficiency when generalized across our domain; a priority order of litmus tests for tuning. We stress test a conformance test suite for the OpenCL 2.0 memory model and discover a bug in Intel's compiler. Our methods are evaluated on the other two GPUs using mutation testing. We end with recommendations for official memory model conformance tests.

CCS Concepts: • **Computing methodologies** → *Graphics processors*; • **Software and its engineering** → **Software testing and debugging**.

Additional Key Words and Phrases: memory consistency, autotuning, conformance testing, GPUs, OpenCL

ACM Reference Format:
Jake Kirkham, Tyler Sorensen, Esin Tureci, and Margaret Martonosi. 2020. Foundations of Empirical Memory Consistency Testing. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 226 (November 2020), 29 pages. https://doi.org/10.1145/3428294

Authors' addresses: Jake Kirkham, jak4@alumni.princeton.edu, Princeton University, USA; Tyler Sorensen, tyler.sorensen@ucsc.edu, UC Santa Cruz, USA; Esin Tureci, esin.tureci@princeton.edu, Princeton University, USA; Margaret Martonosi, mrm@cs.princeton.edu, Princeton University, USA.

2475-1421/2020/11-ART226
https://doi.org/10.1145/3428294

1 INTRODUCTION

With the end of Dennard scaling, computational gains are increasingly achieved through parallelism and specialization. However, efficient many-core systems remain difficult to design and implement correctly. A crucial component of such systems is the *memory consistency model*, which specifies memory-ordering constraints in shared memory concurrent programs. These constraints are specified at the programming language and must be honored throughout the various layers of the modern system stack. For example, compilers must take the memory model into account when performing optimizations, and microarchitectural implementations must ensure certain orderings when propagating values through deep and distributed shared memory hierarchies. A bug in any one of these complex systems could be critical: e.g. memory propagating out-of-order is akin to a race-condition, which have been the cause of power failures [United States Department of Energy 2004] and medical equipment malfunctions [Leveson and Turner 1993].

The strongest memory model, i.e. the model that allows the fewest behaviors, is known as *sequential consistency* [Lamport 1979]. It states that a concurrent program execution must correspond to a sequential interleaving of thread instructions. However, most modern systems (languages and architectures) have adopted *relaxed memory models*, in which there are legal program executions that do *not* correspond to an interleaving. Such behaviors, known as *relaxed behaviors*, are allowed to accommodate various optimizations, from compiler transformations to hardware store buffers. Taking into account the nuanced (and sometimes counter-intuitive) nature of relaxed models, there are ample opportunities for bugs to be introduced throughout the system stack. Indeed, prior work has identified memory-model related errors in applications [Alglave et al. 2015], compilers [Manerkar et al. 2016], and hardware implementations [ARM 2011].

Many modern high-performance programming languages, e.g. C/++ and OpenCL, specify a relational relaxed memory model, described using a dynamic, partially ordered *happens-before* relation between annotated memory operations. The key validation task now is providing confidence that programming language implementations, i.e. the compiler and resulting machine code execution, matches the programming language specification. State-of-the-art techniques in this area typically fall into two categories: formal proof and empirical testing.

Formal Proofs. Ideally, validation of an implementation would be through formal proof, which would show that an implementation honors a specification given any legal program. There has been much work on performing such proofs for memory consistency models, however, due to the complexity of the models and implementations, proofs are largely contained to one layer of the system stack and are performed manually, e.g. using interactive theorem provers. For example, the C++ programming language has proven compiler mappings to the following ISAs: x86 [Batty et al. 2011], PowerPC, and ARM [Batty et al. 2012]. Moving down the system stack, prior work has proven hardware implementations, however with manual effort [Choi et al. 2017].

While these proofs and methodologies are important, they are not always practical (or even correct). Compiler proofs must be manually adapted to account for new (or unmodeled) parts of the languages (e.g. see [Chong et al. 2018; Flur et al. 2017]) and must be safely incorporated into massive compiler code bases. Hardware proofs at this point have only been applied to small systems implemented using very specific frameworks. Deployed systems are much larger and implementation details are proprietary. In the worst (though rare) cases, bugs have even been found in "proven-correct" systems [Manerkar et al. 2016; Norris and Demsky 2013; Yang et al. 2011].

Empirical Testing. A more accessible approach to validation is testing, where a large number of test cases are run on an implementation. While testing does not provide formal guarantees, it has been incredibly successful at finding bugs in complex systems, e.g. compilers [Yang et al. 2011]

and microarchitectural features [Ta et al. 2019]. Memory consistency testing has been especially effective over the years, as problematic behaviors can be concisely distilled into small unit tests, called "litmus tests". These test cases can be synthesized by enumerating cycles in happens-before relations [Alglave et al. 2014], which correspond to a program exhibiting a relaxed behavior.

Litmus tests are parallel programs and are non-deterministic. Each test allows several outcomes, only one of which corresponds to a relaxed behavior. To provide coverage for the many interleavings that may occur throughout the system stack, litmus tests are run many times. Testing campaigns in prior work have applied a limited amount of tuning and run for a bespoke constant iteration count, typically between 50K to 10M per litmus test. However, these testing campaigns show a variety of undesirable properties when analyzed under a rigorous statistical lens.

1.1 Deficiencies of Prior Studies

We illustrate the issues with using fixed iteration counts in empirical memory model testing by examining a large set data set from a 2015 study on Nvidia GPUs [Alglave et al. 2015], which we refer to as the AEA (Alglave Et Al.) dataset. This campaign ran 10930 tests across 5 different GPUs; altogether, containing the results of roughly 140 hours of memory model testing. Each test was run for a fixed 50K or 100K iterations on each device. Analyzing this data set, we identify three significant deficiencies:

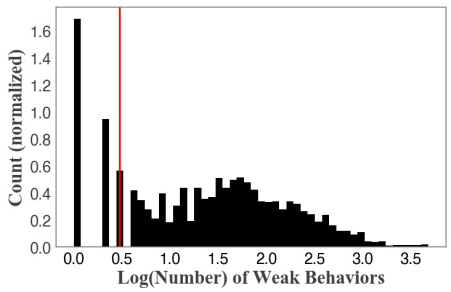


Fig. 1. Distribution of logarithm of number of relaxed behaviors in the AEA dataset. Vertical red line indicates the cutoff point for 95% reproducibility probability. 8% of the tests revealing relaxed behavior are not reproducible at this rate.

Reproducibility. A conformance test suite would be of limited use if different runs passed and failed sporadically. While testing has an inherent amount of uncertainty in it, statistical methods can provide meaningful bounds of confidence on runs, missing in all prior memory consistency testing work.

To illustrate this, we analyze the AEA data set as follows: For each GPU, we compute the probability of reproducibility, P_{rep}, as the probability of observing at least one relaxed behavior for tests that showed any relaxed behaviors when the same number of iterations is repeated. Approximating the true rate of occurrence of relaxed behavior as the observed rate, we can compute this probability as:

$$P_{rep} = 1 - k^N \tag{1}$$

where k is the frequency of not observing $(1 - (n/N))$, n is the number of occurrences of relaxed behavior and N is the number of iterations run. A closer look at this equation shows that $(1 - (n/N))^N$ converges to the Taylor series expansion of e^{-n} for large N:

$$\begin{aligned}(1 - (n/N))^N &= 1 - N\,\frac{n}{N} + \frac{N\,(N-1)}{2!}\,\frac{n^2}{N^2} + \frac{N\,(N-1)\,(N-2)}{3!}\,\frac{n^3}{N^3}\cdots \\ &\approx 1 - n + \frac{1}{2!}\,n^2 + \frac{1}{3!}n^3\cdots \\ &\approx e^{-n}\end{aligned} \tag{2}$$

Since we typically have thousands of iterations, the reproducibility rate can therefore be directly computed as $1 - e^{-n}$ and tied only to the number of times a relaxed behavior is observed: for $n = 1$, P_{rep} is 63.21%, for $n = 2$, it is 86.47%, for $n = 3$, it is 95.02% and so on.

Figure 1 shows the number of relaxed behaviors for all the tests that have at least one observation for all the GPUs in the AEA dataset, where the red vertical line is the number of observations needed for P_{rep} of 95% which is 3. Note that this is an arbitrarily chosen level of reproducibility and can be set to any desired level. In the AEA dataset, 960 (~19.4%) of all tests with relaxed behavior have 1 or 2 observations made (P_{rep} of 63.21% and 86.47% respectively). This, as we will describe below, is also an important indicator for potential false negatives.

Estimated False Negatives: A common critique of testing is that it cannot guarantee the absence of bugs. While this is always going to be true, there are statistical techniques that can be applied to large data-sets to control for the number of false negatives. We cannot compute the true false negative rate without making assumptions about the underlying processes. However, we can derive a lower bound on the potential number of false negatives (F), by undertaking the following exercise: how many of the observed relaxed behaviors would we not observe if we were to repeat the above set of experiments? We can compute this using

$$F = \sum_{i=1}^{max} m_i * (1 - i/N_i)^{N_i} \tag{3}$$

where m_i is number of parameter/GPU sets with i observed relaxed behaviors and $(1 - i/N_i)^{N_i}$ is the probability of not being able to reproduce this observation with N_i iterations. As can be seen, the rate of false negatives is closely related to P_{rep}. We note, however, that this provides a lower bound on the true number of false negatives since we cannot include true false negative cases from the original dataset. Computing this for the AEA dataset, we see that 287/4948 tests would not be reproduced resulting in a false negative rate of 5.8% percent. If all positive cases in the AEA set had 3 or more observations, this number would have been 61 (1.2%).

Just as we probabilistically sampled the AEA dataset, the original dataset and experiments themselves sample the true probabilistic landscape of these systems. Accordingly, it is likely that AEA dataset also had similar rate of false negatives if not more. The above exercise therefore provides an insight into what percentage of such tests might have been missed in the original experiment. We propose that the number of iterations for zero observations be kept at the maximum iteration count that is required for P_{rep} value of at least 95%, for the cases where it was observed.

Redundant Testing Time: The time spent testing a system is increasingly becoming a significant concern, as projects have built up large suites of tests over the years. Furthermore, smaller systems with lower clock frequencies and smaller amounts of memory, e.g. mobile devices, may take longer to execute tests. For example, using the litmus7 testing tool [Alglave et al. 2011], we found that a server-level CPU executes 5M test iterations per second. On the other hand, we found that an Intel integrated GPU (used in laptops) executes only 750 tests per second.

One of our contributions is showing that litmus tests may not need to be run for the full iteration count **if** interesting behaviors are occurring frequently enough. To understand the savings that can occur in a large campaign, we run the following analysis on the AEA dataset. We compute the minimum number of iterations that need to be run in order to have a P_{rep} of $> 95\%$ for each test case where relaxed behavior was observed. We reverse calculate the number of iterations (N) that would be sufficient: $N = log(1 - P_{rep})/(log(1 - p))$, where p is the observation rate. This computation shows that in the AEA dataset over 86 million iterations were unnecessary, which in some scenarios can correspond to several days of unnecessary testing.

Table 1. GPUs used in this study; along with short names used throughout the text. While the Intel OpenCL driver reports a large number of compute units (CUs), prior work has thread-occupancy 8× less than this reported number [Sorensen et al. 2016]

Vendor	Chip	Discrete	# of CUs	Short Name
Nvidia	Quadro P5000	Yes	20	Quadro
AMD	RX Vega 64	Yes	64	Vega
Intel	Iris Pro 650	No	48	Iris

1.2 Foundations of Empirical Memory Consistency Testing

In this work, we explore and evaluate our ideas on Graphics Processing Units (GPUs). These ubiquitous devices are found in systems ranging from the most powerful super computers to tiny edge devices, e.g. smart watches. Clearly, GPUs in these different settings have different use-cases, and the corresponding performance/power trade-offs have resulted in a large architectural diversity, with nearly all major vendors producing their own chip. Despite this chaotic landscape, programming models, such as OpenCL, aim to provide a unified abstraction under which these devices can be programmed. Since version 2.0, OpenCL has officially adopted a language-level memory consistency model, which we will use to guide our experiments.

Contributions. As this paper will show throughout, different GPUs have significantly different behaviors in terms of the type and frequency of observed relaxed behaviors, as well as the stressing parameters that increase relaxed behavior observations. We also show that our methodologies are robust to these differences, and while the specific numbers in observations may change, our proposed techniques are able to increase efficiency and yield statistical insights across all platforms. Our contributions can be summarized as:

(1) **Tooling**: a *portable* GPU memory model testing tool, targeting OpenCL, that can be configured with a range of stressing parameters (Section 3).[1]
(2) **Characterization of Relaxed Memory Observations**: Using our tool, we collect relaxed memory testing observations across 3 GPUs, 6 litmus tests, 1K stress parameter configurations and 10K test iterations each. We characterize: (1) the frequency of observing relaxed behaviors with different stressing parameters; (2) the rates of relaxed behaviors per parameter configuration; (3) the stability of relaxed behavior rates across time (Section 4).
(3) **Efficiently Tuning and Generalizing Stress Parameters**: Our characterizations motivate the need to tune stressing parameters to yield higher rates of relaxed memory observations. To this end, we develop techniques to efficiently tune stress parameters and methods for generalizing parameters to perform well across different GPUs. This includes: (1) a data peeking scheme that yields a lossless 5× increase in tuning speed; and (2) a novel portability metric that enables stressing parameters to be generalized across GPUs, costing only a 12% reduction in the rate of relaxed memory observations (Section 5).
(4) **Conformance Testing of the OpenCL 2.0 Memory Model**: We develop a conformance test suite for the OpenCL 2.0 memory model, which couples litmus tests with tuned stress. Using this, we find a bug in the Intel OpenCL compiler and perform a mutation testing analysis on AMD and Nvidia GPUs. We show that our conformance tests can expose 83% of the mutants, while existing conformance tests find none. We conclude with recommendations for memory model conformance test designs (Section 6).

[1]Tool is available at: https://github.com/tyler-utah/OpenCLLitmus_v_2

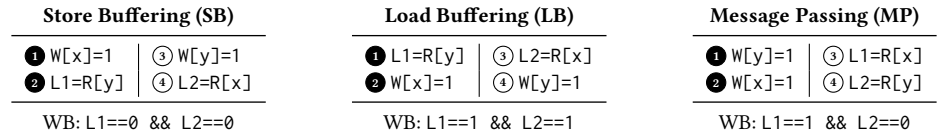


Fig. 2. Three litmus tests: **SB**, **LB**, and **MP**. Each test consists of two concurrent threads: one thread executes the left-hand instructions, (black circles); another thread executes the right-hand instructions (white circles). Instructions are Read (R) and Write (W). Memory locations are initialized to 0. Local variables are prefixed with an L. Each test allows one relaxed behavior, which can be tested by examining values of local variables and memory post mortem, as indicated by the last line in each test (labeled with WB).

Table 2. The 7 possible outcomes of the tests in Figure 2: Sequentialization of the threads (the first two); sequential interleaving (next 4, allowed under sequential consistency); the relaxed behavior (allowed under various relaxed memory models (e.g. OpenCL and C/++) if sufficient synchronization is not used).

Execution	SB	LB	MP
❶ ❷ ③ ④	(L1==0 && L2==1)	(L1==0 && L2==1)	(L1==1 && L2==1)
③ ④ ❶ ❷	(L1==1 && L2==0)	(L1==1 && L2==0)	(L1==0 && L2==0)
❶ ③ ❷ ④	(L1==1 && L2==1)	(L1==0 && L2==0)	(L1==0 && L2==1)
❶ ③ ④ ❷	(L1==1 && L2==1)	(L1==0 && L2==0)	(L1==0 && L2==1)
③ ❶ ④ ❷	(L1==1 && L2==1)	(L1==0 && L2==0)	(L1==0 && L2==1)
③ ❶ ❷ ④	(L1==1 && L2==1)	(L1==0 && L2==0)	(L1==0 && L2==1)
Relaxed Behavior	(L1==0 && L2==0)	(L1==1 && L2==1)	(L1==1 && L2==0)

2 BACKGROUND

Here, we review memory consistency models, illustrating relaxed memory behaviors through small idiomatic tests (Section 2.1). Next, we provide an overview of GPU programming, including the formal memory model adopted in OpenCL 2.0 (Section 2.2).

2.1 Memory Consistency Models

A memory consistency model specifies the values that load operations are allowed to return in shared-memory concurrent programs [Sorin et al. 2011, ch. 1]. The strongest model is sequential consistency [Lamport 1979], which states that any behavior of a concurrent program must correspond to some sequential interleaving of the instructions. However, all modern multi-core architectures, and many low-level languages (e.g. C) provide relaxed memory models, where some allowed program behaviors do not correspond to an interleaving of instructions.

Litmus Tests. We illustrate relaxed memory models using "litmus tests": small concurrent tests with a post-mortem condition that is only allowed under a relaxed memory model. Figure 2 shows three tests that we will discuss frequently throughout: Store Buffering (**SB**), Load Buffering (**LB**) and Message Passing (**MP**). The possible behaviors of each test are enumerated in Table 2: there are six possible sequential interleavings of instructions, leading to three sets of final values for the two local variables (L1 and L2). Each test might also allow a relaxed behavior, allowed by common relaxations in memory consistency models. For example, the **SB** relaxed behavior is allowed in the x86 ISA [Sewell et al. 2010], while **LB** and **MP** are allowed in ISAs like ARMv8 [Pulte et al. 2018].

Memory Model Specifications. Nowadays, memory models are specified through happens-before relational constraints, e.g. the C++ model [Batty et al. 2011]. These models formalize concurrent

program executions as a set of events (e.g. reads and writes), and relations. Executions are disallowed if the model's relational constraints are not satisfied; for example, a model may build a temporal happens-before relation between events, which is intuitively constrained to be acyclic. These models are appealing due to their expressivity, i.e. they can capture more behaviors than thread-local reordering specifications, and their relational semantics are straightforwardly incorporated into formal tools, e.g. test-case synthesis [Wickerson et al. 2017]. However, the formalism of these models is not needed for this work. Thus, while we may refer to relational models and their applications (e.g. test-case synthesis in Section 6), our contributions do not require an in-depth background.

Explicit Orderings and Data races. Memory models describe how to disallow relaxed behaviors, i.e. restoring sequential consistency. Some models provide special fence instructions (e.g. `mfence` in x86) that can be placed between memory operations to provide relative ordering. For example, if an `mfence` instruction is placed between the write and read in the SB test on both threads, then the relaxed behavior would be disallowed on x86. Other models, e.g. ARMv8 and C, provide programming annotations for read and write instructions, e.g. `release` and `acquire` annotations for writes and reads, respectively. In the MP test, if instructions ❷ and ③ are annotated with `release` and `acquire` respectively, then the relaxed behavior is disallowed.

Higher-level languages (e.g. C) can have a notion of a *data-race*: a pair of concurrent memory accesses that are not synchronized (e.g. ordered by a happens-before relation). Programs with races are typically given "catch-fire" semantics, i.e. all program behavior is undefined. In this work, we aim to test a well-defined memory model specification, thus we protect all our concurrent memory accesses with explicit `atomic` types and accesses, provided by the OpenCL language (described in Section 2.2). By definition, atomic accesses never race, thus our tests are all data-race free.

2.2 GPU Programming and OpenCL

GPUs have supported general purpose programming languages for over a decade now, with the two main languages being: CUDA [Nvidia 2019], which exclusively targets Nvidia GPUs; and OpenCL [Khronos Group 2019], which targets GPUs from many different vendors. GPU programming models have evolved rapidly and now contain a range of diverse features. However, given that (1) support for programming features varies across different chips, and (2) we target a range of GPUs in this work (Table 1), our GPU programming subset will be relatively simple. Because we target GPUs from non-Nvidia vendors, we will default to OpenCL terminology throughout.

Programming GPUs. A GPU program consists of two components: the *host* computation, which is executed on the CPU; and the *device* computation (sometimes called the *kernel*), which is executed on the GPU. The host copies memory to the device and then launches the kernel to perform computation on the data. After the kernel finishes, the host can copy the computed data back to the CPU. The kernel is programmed with OpenCL C (based on C99). A kernel is executed in a SIMT (single instruction, multiple threads) manner, where many threads execute the same kernel, but have access to unique thread identifiers, which are commonly used to: (1) access distinct memory locations in a data-parallel program, or (2) conditionally execute certain code blocks.

The OpenCL model exposes an execution hierarchy, which can efficiently be mapped to the spatial layout of massively parallel GPU devices. The base unit of execution is called a thread,[2] which executes one instance of the kernel. Equal-sized contiguous sets of threads are organized into workgroups (or thread-blocks in CUDA), which naturally map onto compute unit (or streaming multiprocessor in CUDA). A GPU provides three memory spaces: *global* memory is available to all threads on the GPU; *local* memory (shared memory in CUDA) is shared only between threads in the

[2]Some OpenCL literature refers to threads as *work-items*

same workgroup; *private* memory is private to a thread. These memory regions can be efficiently implemented on GPUs through large register files (for private memory), non-coherent scratchpads (for local memory) and a traditional, cached memory hierarchy (for global memory).

Synchronization can be achieved at the workgroup level using the `barrier` primitive. There is no official synchronization primitive available at the inter-workgroup scope, although prior works have shown how one can be implemented [Gupta et al. 2012; Xiao and Feng 2010].

OpenCL Memory Consistency Model. OpenCL specifies a relational memory model, defined using happens-before relations [Khronos Group 2015, ch. 3.3]. Instructions executed in program order are defined to be happens-before ordered. Inter-thread orderings can be induced by using *atomic* types (e.g. integers and booleans), which can be accessed (i.e. read from or written to) with primitive functions. These accesses can be annotated with different memory orders, which induce different strengths of synchronization. In order from strongest to weakest, these orders are:

(1) **`seq_cst`**: is short for sequential consistency; if all shared accesses have this annotation, then all relaxed behaviors are disallowed; e.g. if all instructions of the tests of Figure 2 are given this annotation, then the specification disallows all relaxed behaviors.
(2) **`release`**/**`acquire`**: are given to writes and reads, respectively. If a read `acquire` observes a value written by a write `release`, then it induces a happens-before relation; e.g. in **MP**, if ❷ and ③ are `release`/`acquire` annotated, then the relaxed behavior is disallowed.
(3) **`relaxed`**: is the annotation with the weakest constraints. It induces no synchronization; e.g. all relaxed behaviors of the Figure 2 tests are allowed if the accesses are annotated this way.

The OpenCL model extends the C model by providing an additional annotation to atomic memory accesses that specifies the level (or *scope*) of the OpenCL hierarchy at which synchronization is to be induced, e.g. `workgroup`, or `device`. Memory ordering guarantees are only provided if the synchronizing threads belong to the same instance of the hierarchy defined by the memory scope annotation. That is, if two threads are in the same workgroup, they should use the `workgroup` scope annotation. The widest scope we consider in this work is `device`, and all threads executing on the same GPU can synchronize with this scope.

There are many nuances to these models (e.g. mixing annotations, non-atomic accesses, etc.). However, a full description is outside the scope of this paper. These models have been rigorously formalized (for C [Batty et al. 2011] and OpenCL [Batty et al. 2016]) and tools exist to explore litmus test behaviors[3]. This work uses the OpenCL model only at a high-level to reason about simple litmus tests; however, this is sufficient to find bugs in an industrial compiler (Section 6).

3 TUNING PARAMETERS AND EXPLORATORY DATASETS

Relaxed behaviors can be difficult to observe on some architectures, especially GPUs. Prior works have shown that to observe *any* relaxed behaviors on Nvidia GPUs, some additional stress and fuzzing heuristics were required. For example, litmus tests typically consist of a small number of threads; the additional GPU threads can be programmed to repeatedly access a disjoint memory region. This can cause contention in the memory hierarchy, affecting the ordering that memory values propagate to other threads. Other heuristics involve changing the affinity of testing threads, i.e. changing the physical core to which they are mapped. Different cores are at spatially different locations on the chip and may interact through different mechanisms, e.g. a different cache level.

Here we overview tuning parameters we use to influence the rate at which relaxed behaviors are observed in empirical litmus testing. These tuning parameters are inspired by prior works [Alglave et al. 2015, 2011; Sorensen and Donaldson 2016] and have been shown to be effective for CPUs and

[3] see: http://svr-pes20-cppmem.cl.cam.ac.uk/cppmem/

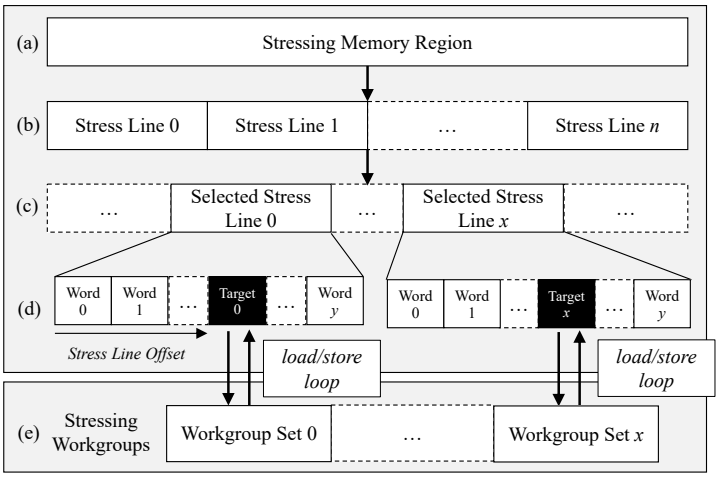


Fig. 3. Overview of memory stress. The memory stressing region is divided into disjoint, equal sized regions called stress lines (a→b). A random set of stress lines is picked at each iteration (c). The memory location that is actually accessed is selected at a random offset in each selected stress line (d). Stressing workgroups, disjoint from testing workgroups, repeatedly access testing targets with load and store operations in a loop.

Nvidia GPUs. Our contribution is to port these parameters to OpenCL in a robust and portable way, which allows us to evaluate their efficacy on a wider variety of devices. Our implementation prioritizes parameter exploration while other implementations either hard code parameter settings, or limit their domain. For example, the study in [Sorensen and Donaldson 2016] (which we will refer to as the SD16 study) used much smaller domains and explored them exhaustively. We increase the domain sizes with the intent of developing efficient tuning strategies.

The parameters are divided into three sets: (1) Memory parameters that dictate the memory locations in the litmus test and how non-testing threads might stress the memory system; (2) Concurrency parameters that aim to temporally align testing threads; and (3) Affinity parameters that aim to map testing threads to a variety of computational resources on the device. Our framework has a 2-level search space. We will call the outer-level the *parameter space*. We will refer to parameter names at this level with the SmallCaps font. Once parameter values are set, there are inner-level parameters, called *iteration parameters*. These parameters are simply randomized at each iteration. That is, iteration parameters are not tuned, they are fuzzed. Each iteration is one execution of a litmus test and will produce exactly one outcome, e.g. in Table 2, one iteration of the SB test will yield one of the behaviors in the **SB** column.

3.1 Memory Parameters: Stressing and Testing Locations

Our memory stressing scheme is inspired by the SD16 study, where threads (at the workgroup granularity) and memory are cleanly partitioned into two disjoint sets: testing and stressing. *Testing workgroups* execute the litmus test on a region of memory allocated for the test, *testing memory*. On the other hand, *stressing workgroups* only access *stressing memory*. This partition eases development, as stressing never conflicts with the actual litmus test, which could result in false positives. Stressing workgroups repeatedly access a small number of memory locations, known as *stressing targets* in the stressing memory. Figure 3 shows an overview of the stressing partition of the testing framework.

Stress Lines. The SD16 study showed that stressing targets had similar efficacy (with respect to the rate at which relaxed behavior were observed) in contiguous 512-byte (or 1024-byte) memory regions. That is, if stressing a memory location l caused a rate r of relaxed behaviors to be observed for a given test and chip, then stressing any memory location from l to $l + 512$ (assuming that l is 512-byte aligned and byte-addressable), would yield a similar rate r for the same chip and test. While there was no concrete hardware justification given for this, it was speculated that these region sizes corresponded to a combination of bus width, cache-line size, etc.

To implement this, we introduce the parameter StressLineSize. This parameter is set to a power-of-two value, say Z. The stressing memory is divided into disjoint, equal sets of size Z, each of which we call a stress line. This is illustrated in (a)→(b) in Figure 3. The stressing targets are then selected to be in different stress lines. The selected stress lines and the stress line offset (see (d) in Figure 3) are iteration parameters, i.e. randomized for each iteration.

Target Number. Many threads concurrently accessing the same location will pressure the coherence protocol, which must enforce single-address consistency. However, accessing a wide variety of locations may cause cache thrashing and timing variability with respect to how memory is propagated through the system. Thus, there is intuition for having both a large and small number of targets. To implement this, we introduce the TargetNumber parameter. This is a number between 1 and 16 (N), which dictates how many stress-lines are concurrently stressed. That is, N disjoint stress-lines are selected (as randomized iteration parameters) and assigned to the stressing workgroups (see stressing memory in (d) and stressing workgroups in (e) of Figure 3). The assignment strategy of mapping workgroups to stressing targets is also a parameter: StressAssignment. This can be set to either round-robin or chunking. In round-robin, stressing workgroups (with id `wig`) are assigned to stressing the target number `(wig % N)`. For chunking, W stressing workgroups are assigned to stressing target `(wig / (W/N))`.

Access Pattern. Having assigned targets to workgroups, now the access pattern to use to stress the target must be determined. Intuitively, load instructions can bring in fresh lines, potentially causing temporal variability from cache thrashing. On the other hand, store instructions can invalidate remote cache lines, increasing bus traffic. Taking inspiration from the SD16 study, all stressing workgroups use the same sequence of instructions to repeatedly access their target location in a loop. The parameter AccessPattern dictates this access sequence: it is two instructions, each of which can be a load or store. This is illustrated in the interaction between (e) and (d) in Figure 3.

Testing Locations. Litmus tests typically consist of a small number of memory locations. The tests we consider in this work contain only two, which we refer to as x and y. Empirically a litmus test may show different behaviors depending on the relative location of x and y. For example, if x and y are on the same cache line, then behavior corresponding to asynchronous cache line propagation may not be observed. Thus, it is important to empirically run tests with testing locations distributed throughout memory. To implement this, the XYStride parameter is provided. This value can be set to a power-of-two, say Q. Testing locations (x and y) are then assigned to be in distinct Q-sized regions of the testing memory. There are two associated randomized iteration parameters: The selected Q-sized region for x and y, and the word-offset of x and y inside their respective region.

Pretest Memory Stress. The hardware units that testing threads execute on may contain components that allow relaxed behaviors if stressed. Because stressing threads are disjoint from testing threads, these testing thread hardware components may not be stressed. Thus, we provide a PretestMemoryStress parameter where testing threads execute several iterations of memory accesses (determined by a corresponding PretestAccessPattern parameter) immediately prior to executing the litmus test. Pilot tests showed that 100 iterations worked well in practice.

3.2 Concurrency Parameters: Barrier and Occupancy

Due to subtle timing differences (e.g. thread launch latency, variable memory access latency), threads can become significantly desynchronized. This is problematic in litmus tests because one thread may execute its test program far enough ahead of the other thread that all memory accesses have had enough time to propagate to global visibility. Thus, the observed outcomes will simply correspond to a sequential execution, e.g. the first two outcomes in Table 2.

Prior work has shown that synchronizing threads immediately before they execute the litmus test can increase the number of relaxed behaviors observed [Alglave et al. 2015, 2011]. To achieve this, we provide the Barrier parameter: when enabled, it adds a software-implemented barrier for the litmus testing threads, occurring immediately before their litmus test instructions. Because GPUs do not provide the necessary forward progress guarantees for barriers [Sorensen et al. 2016], we use a *timeout* feature, where threads can exit the barrier after some time. Thus, if testing threads are not fairly scheduled, they will eventually cleanly exit the barrier unsynchronized; however, this is better than the alternative of deadlock, from which it can be difficult to recover.

Occupancy. GPUs offer a parallel programming model in which kernels can be launched with thousands of threads. However, the system is only able to run a subset of threads concurrently, limited by hardware resources [Sorensen et al. 2016]. In order for testing threads to temporally align, and for stressing threads to execute concurrently, we limit the number of workgroups for the testing kernel. while still aiming to cover a variety of system configurations. To achieve this, we use an iteration parameter that randomizes the number of workgroups each iteration. We select between 2× the number of compute units, and 2, the number of testing threads.

3.3 Affinity Parameters: Id Shuffling and Oversubscription

While GPUs can run many threads concurrently, litmus tests typically consist of only a small number of threads. A thread-to-hardware mapping might deterministically place threads on computational resources spatially related to their thread id. Thus, simply assigning litmus test threads to the first few GPU thread ids may not robustly expose behaviors that occur when threads mapped to different hardware components interact, e.g. due to a shared cache. Prior work has shown that randomly assigning native thread ids to the litmus test threads has been successful at increasing the number of observed relaxed behaviors [Alglave et al. 2015, 2011; Sorensen and Donaldson 2016].

To encourage threads that span various hardware resources to interact, we implement a Thread-Shuffle parameter. When this parameter is enabled, we override the native OpenCL thread ID primitive with a new thread id mapping that our testing framework provides. Concretely, in a kernel launched with a total thread count of T, we allocate a new array ST of size $|T|$, and initialize it with the identity mapping, i.e. $ST[i] = i$. We then shuffle ST and use the values as a new thread id for kernel threads, i.e. a kernel thread can get a new shuffled id with $ST[tid]$. The shuffling is constrained to honor workgroup membership. That is, two threads that share the same workgroup natively, will have shuffled ids that also share the same workgroup.

Similar to the iteration parameter that randomizes the number of workgroups each iteration, we similarly randomize the number of threads per workgroup. Prior work has shown that GPU compute units can be oversubscribed with smaller workgroup sizes. Thus, random configurations can encourage both co-occupancy and disjoint resource occupancy of testing threads, testing even more system configurations. Thus, we supply an iteration parameter for workgroup size.

3.4 Parameter Summary and Datasets

The outer parameter space is summarized in Table 3. There are 10 parameters (the memory stress must be enabled for the shaded ones), leading to a total of 737K possible configurations.

To explore how parameters influence the rate of observed relaxed behaviors, we collect two data sets: the Exploratory Data Set (EDS) and the Stationarity Data Set (SDS). These data sets consist of many samples across many parameter settings, summarized in Table 4. The EDS will be analyzed for the effect that different parameters have on the rate of observed relaxed behaviors; thus, the large number of parameters sampled. On the other hand, the SDS dataset will be analyzed to see if relaxed behavior rates are stable across a given parameter setting, thus there is only one parameter sample and many iterations. The parameter samples and iteration values were chosen as they were worked well in pilot experiments and were feasible under our time and resource constraints. The runtime, per chip, to collect the data is given in Table 5.

To explore a variety of relaxed memory behaviors, we applied the parameters to all of the two-thread, two-memory location litmus tests, as enumerated using critical cycles by prior work [Alglave et al. 2010]. These tests are the three presented earlier in Figure 2: **MP**, **LB**, and **SB**. The three additional tests are **S**, **R** and **2+2W**. Each test is instantiated where each memory access is annotated with relaxed memory order and device memory scope. Thus, the tests are data-race-free; however, the OpenCL memory model allows the relaxed outcome. This is desirable as our initial exploration aims to examine the rates at which relaxed behaviors can be observed when allowed. Our motivation is that if we cannot observe relaxed behaviors when they are allowed, then it is very unlikely we will observe them when they are forbidden (e.g. in conformance tests). Our approach has proven pragmatic, finding a bug in Intel's compiler, as documented in Section 6.1.

Table 3. Summary of outer parameters: Parameters marked with a † have a corresponding pre-stress parameter. Shaded parameters correspond to memory stress and only change the program if MemoryStress is enabled. Upper bounds were obtained from a short run of pilot experiments across our GPUs.

Parameter	Values
ThreadShuffle	{on, off}
Barrier	{on, off}
MemoryStress†	{on, off}
StressLineSize	$2^x, x \in \{1-10\}$
TargetNumber	$\{1-16\}$
StressAssignment	{round-robin, chunking}
AccessPattern†	`A0;A1; A0,A1` $\in$ {LD, ST}
XYStride	$2^x, x \in \{1-9\}$

Table 4. Experimental space for the: Exploratory Data Set (EDS) and Stationarity Data Set (SDS). The one parameter sample for SDS corresponds to the parameter configuration from EDS that revealed the most relaxed behaviors per chip/test combination.

	EDS	SDS
GPUs	3	3
Litmus Tests	6	6
Parameter Samples	1000	1
Iterations	10000	1000000
Total Executions	18000000	18000000

Table 5. Total time to collect data described in Table 4. Larger chips (w.r.t. compute units) take longer to run as the memory stress provides more workgroups to concurrently access memory, significantly slowing down execution time, e.g. see [Iorga et al. 2020].

Chip	Time (h)
Quadro	52
Vega	66
Iris	36
Total	154

Cause of Relaxed Behaviors. Relaxed memory behaviors can occur throughout the computing stack (e.g. compiler or hardware). Our stresses are aimed at finding bugs that deal with insufficient

synchronization after the high-level language is mapped to ISA instructions. Different techniques would likely be employed to test compiler reorderings. While we cannot guarantee that the relaxed behaviors we observe are not due to compiler reorderings, we believe it to be unlikely. The addresses for litmus test locations are passed as kernel parameters, which the compiler cannot tell are distinct, thus a compiler reordering could violate the single-address consistency guarantees from the high-level language. Furthermore, memory consistency bugs have typically been found in the compilation mapping scheme (from JavaScript [Watt et al. 2020], to C++ [Manerkar et al. 2016], as well as the issue we present in Section 6.1). As far as we know, there have never been errors reported in compiler reorderings of atomic instructions; thus we leave testing and tuning compiler reorderings to future work.

4 INITIAL OBSERVATIONS: FREQUENCY, RATES, STABILITY AND EFFECTIVENESS

In order to develop a statistically sound, and effective, tuning methodology, we first need to understand the landscape over which the tuning would be performed. To achieve this, we characterize several dimensions over our entire dataset. In particular, we examine: the number of parameter configurations that yield any relaxed behavior result; the rates at which relaxed behaviors are observed; and the stability of relaxed behavior rates over time. We use these observations to motivate and guide the tuning strategy laid out in the Section 5. Throughout the following results section, we will often color code our images. Red corresponds to results from Vega, yellow corresponds to results from Quadro, and blue corresponds to results from Iris.

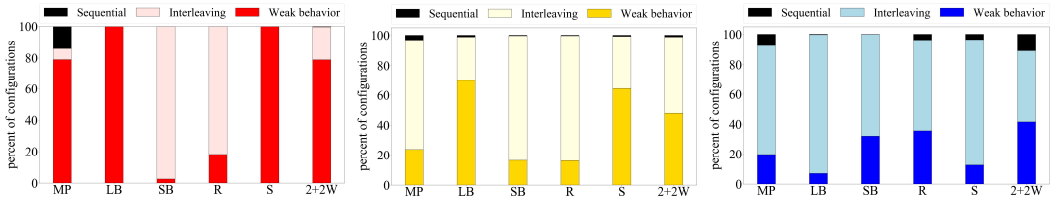
4.1 Sequential, Interleaving, and Relaxed Observations

We first explore how many parameter configurations are able to reveal at least one relaxed behavior across chips and tests. Prior work has postulated that GPU relaxed behaviors are extremely rare and require tuned stress to reveal behaviors [Sorensen and Donaldson 2016]. If indeed, only a small number of parameters are able to reveal relaxed behaviors, it would be strong evidence that memory consistency testing needs to be carefully tuned. Otherwise tuning may not be necessary.

Figure 4 shows stacked bar charts across chips and litmus tests showing the different outcomes observed. Recall that there are 1000 configurations, executed for 10K iterations each. The bar chart shows how many configurations, across all iterations, revealed: (1) only sequential behaviors, i.e. the first two rows in Figure 2; (2) at least one interleaving behavior, i.e. the middle four rows of Figure 2; and (3) at least one relaxed behavior, i.e. the last row of Figure 2. We see that only a small number of parameters result in exclusively sequential behaviors, likely due to the care taken to ensure concurrent execution (see Section 3.2).

Overall, we observed relaxed behaviors for all chips on all tests; thus, we see this as evidence that our stress techniques generalize across GPUs, even if relaxed behavior revealing parameters are rare. We also note that this is further evidence that GPUs implement relatively *weak* memory models, compared to some of their CPU counterparts (e.g. x86 would only show **SB** relaxed behaviors). Finally, this is the first work to show empirical relaxed behaviors on Intel GPUs (Nvidia and AMD were shown in [Alglave et al. 2015]).

The percent of parameters that reveal relaxed behaviors, however, vary significantly across chip and litmus test. Overall, Iris revealed relaxed behaviors for the fewest number of configurations (less than 50% for all tests); thus, this chip likely needs more careful tuning when testing. Interestingly, for Vega both extremes are observed: For example, **LB** and **S** both showed at least one relaxed behavior for *every* parameter configuration. This is the first evidence of relaxed behaviors being so readily observable on GPUs. On the other hand, **SB** on the same chip only appeared in 2.6% configurations, being the ”rarest“ observed relaxed behavior for all chip/test pairs. Across chips, the **SB** relaxed behavior is observed in the fewest number of configurations. The **LB** and **S** tests are



(a) Vega (b) Quadro (c) Iris

Fig. 4. Stacked bar charts per chip (and test) for the percent of outer parameter configurations that revealed: at least one relaxed behavior; at least one interleaving, but no relaxed behaviors; and only sequential behaviors.

observed in the largest numbers of configurations for both Vega and Quadro, but interestingly, in the fewest for Iris.

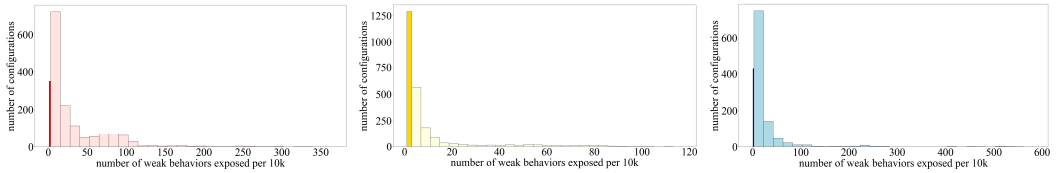
Ultimately, these results show that while some relaxed behaviors are relatively easy to provoke (appearing under a variety of parameters), some remain incredibly rare. Thus, it appears that time allocation for tuning should be adaptable, so that rare tests can be more carefully tuned, while others can be more quickly tuned. We propose such a method in Section 5.1 and show a 5× speedup in tuning across the board.

4.2 Rate Distribution

The percentage of parameter configurations that revealed at least one relaxed behavior does not tell the complete story. Different parameters may cause relaxed behaviors to be observed at higher or lower rates. Low rates are undesirable because they must be run a high number of iterations to establish confidence about the observations, a key example being reproducibility (as we discussed in Section 1.1). On the other hand, a parameter set that yields a high rate of observed relaxed behaviors does not have to be run for as many iterations and higher confidence can be established much quicker. We will use this insight in Section 5.1.

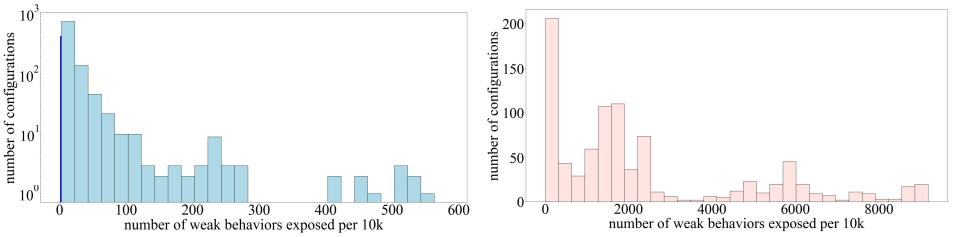
Figure 5 shows for each of the three GPUs, a histogram of the rates at which relaxed behaviors were observed for all configurations which revealed at least one relaxed behavior. Results are aggregated across all tests per GPU, with the tests **R** and **LB** being omitted for Vega. This is because all tests (per GPU) followed a similar trend, except for these two. Specifically, the histogram shows an exponential decay, i.e. the number of parameters that yield higher relaxed behavior observation rates decreases exponentially. Figure 6a shows the same data as Figure 5c, except on a logarithmic scale to illustrate this point. Figure 6b shows one of the outliers, **LB** on Vega which clearly does not follow an exponential decay and exposes massive rates, i.e. **LB** shows a max rate of over 8K (per 10K) compared to the max rate of 350 when these outliers are excluded.

As discussed in Section 1.1 , in practice the mere existence of relaxed behavior is not sufficient for the analysis of a system for correctness: reproducibility of the relaxed behavior in a short amount of time is necessary to make the design, analysis and debugging process efficient. Therefore, we now explore the above results with respect to reproducibility. Revisiting Section 1.1, number of relaxed behavior observations needs to be *at least* 3 for a P_{rep} of 95%. Our graphs show that many of parameter configurations revealed 1 or 2 observations, meaning that they are not reproducible with high probability. This is shown in the first bucket (highlighted) in Figure 5, which has a fixed bucket size of [1-2]. In fact, over half of the parameter configurations for Quadro did not yield a high enough relaxed behavior rate to be reproducible. The flip side of the coin is that we likely have a significant number of false negatives. Using now Equation 3, we can estimate the probability of not being able to reproduce these results with at least one relaxed behavior. For Iris, Quadro and Vega,



(a) Vega (b) Quadro (c) Iris

Fig. 5. Rate of observed relaxed behaviors histograms: All GPUs show an exponential decay, that is parameters that reveal a low rate of relaxed behaviors are much more common than parameters that reveal many relaxed behaviors. The first bucket (shown in a darker color) is a fixed size, containing parameters that had a rate of [1-2] (per 10K) relaxed behaviors. This bucket is highlighted as these runs have low P_{rep} (described in text).



(a) Iris rate histogram on a logarithmic scale (b) Vega rate histogram for the **LB** test

Fig. 6. Two cases of the rate histograms examined further

number of false negatives per number of parameter sets with relaxed behavior is 25/533, 56/876 and 0/998 respectively (lowest number of relaxed behavior occurrence for Vega was 14). Note that this is a lower bound for false negatives since we cannot include the cases that are truly false negative in our original data. We do not explore this further however, as we are more interested in parameter settings that revealed a high rate of relaxed behaviors.

Our key observation here is that there are a small number of parameter configurations that lead to high rates of relaxed behavior observations, which in turn, allow more confident reasoning about results. A relaxed memory testing suite should strive for these high rates to ensure reproducibility, even at lower iteration counts. At the extreme end, the highest observed rate (out of 10K) is 9484 for **LB** on Vega, which shows a relaxed behavior nearly every single execution (we leave further exploration around this extreme result to future work). The lowest rate is **SB** on Vega showing only 3 per 10K; a startling difference. This is further evidence that tuning and testing techniques should be adaptable, i.e. on Vega one would allocate more time to difficult tests like **SB** and less time to tests like **LB** that are readily observable. We also note the curiosity that **SB** has the highest rate on Iris, but lowest on Quadro and Vega. The Intel chip is integrated while the others are discrete, and thus, they likely have different hardware components that cause these differences.

4.3 Stationary Relaxed Behavior Observation

Because litmus tests are run for many iterations, one must make sure that the system is stable, i.e. there are no fluctuations that cannot be accounted for by the mean and variance of the relaxed behavior rate. A stable system provides confidence that there are no unaccounted factors contributing to the relaxed behavior rate. This is especially important when experiments are run consecutively; prolonged GPU usage might result in physical conditions that can affect the system

Table 6. Minimum and maximum rates observed across chips. Quadro has a low max rate (111 out of 10K), meaning all relaxed behaviors are rare on this chip. Vega has an incredibly high rate of almost 95% for **LB**.

Chip	Min.	Rate (per 10k)	Max.	Rate (per 10k)
Quadro	SB	8	S	111
Vega	SB	3	LB	9484
Iris	LB	62	SB	544

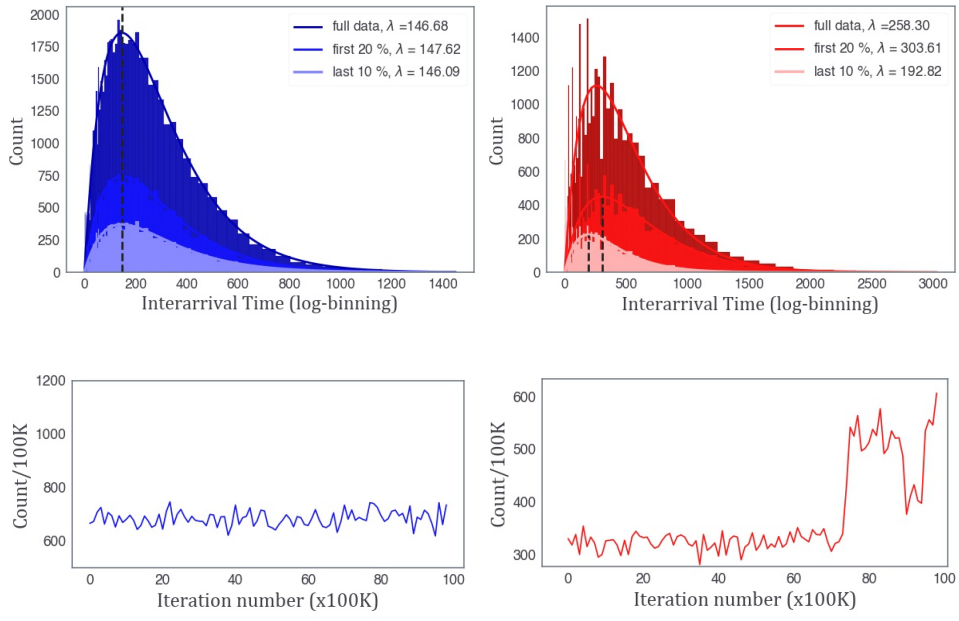
performance. For example, recent work showed that frequency throttling caused large variances in shared-memory interference experiments [Iorga et al. 2020]. In addition, our tuning heuristics (see Section 5) require reliable confidence intervals that can only be derived from stationary processes.

To explore system stability, we analyzed the rate of relaxed behaviors given a fixed set of parameters over ten million iterations, which is the maximum number of consecutive iterations of tests run on any of the GPUs in our study. We used several methods to test whether relaxed behavior observation is stationary. A process is stationary if its mean and variance is preserved across independent variables, e.g. time and iteration count. In the case of a Poisson process (such as observance of a relaxed behavior) these two properties, i.e. mean and variance, are equal.

To ensure that the relaxed behavior rate is stationary for our chip/test combinations, we carried out several analyses: in addition to visually inspecting the rate across time (Figure 7, bottom row), we compared inter-arrival count distributions of the entire run, the initial 20% and the last 10% of the entire run for each chip/test combination to compare relaxed behavior frequencies in these two windows. We fit the data using a Poisson process to obtain the rate of relaxed behavior and compared the observed rates both qualitatively and also using the Kolmogorov-Smirnov (KS) statistical significance test using the raw data. As can be seen in Figure 7, for a stable chip/test combination, the rate is stable throughout the experiment duration. However, for some chip/test combinations this is not the case.

The KS test evaluates whether two datasets are likely coming from different distributions. The rationale for using the KS test is this: our goal is to determine if the relaxed behavior rate is stable across a tuning experiment. This would provide confidence that a change in rate of occurrence is due to change in parameters rather than a mode switch or drift in the system. Thus, we need a test that can show if there are any significant changes in the rate throughout the duration of an entire tuning experiment. The KS test achieves this goal. We run the KS test with the null hypothesis that the distributions are coming from the same distribution. If the null hypothesis is rejected then KS test indicates that with high likelihood (p-value < 0.05) two samples have different distributions, i.e. rates, thus tuning experiments cannot be soundly run. However, if KS test cannot reject the null hypothesis, then even if we cannot guarantee that the rate distributions are not coming from the *same* distribution, the rates of these distributions must be (with high likelihood) very close to each other. Since all we need is for the rate to not change statistically significantly, this means the tuning experiment is sound to run, as our goal is not to prove that there is only one mode but rather that the rate does not change significantly across time/iterations.

Running this test on the first 20% of the iterations and the last 10% of the iterations we can eliminate chip/test combinations that are not stable for further analyses. Table 7 shows the p-values coming from the KS test for all chip/test combinations. We found 4 combinations (out of 18) that are not stable across iterations. Figure 7b shows the rate of relaxed behavior occurrence for one of these combinations. One can observe here that around iteration count 7 million the rate jumps,



(a) Iris (b) Vega

Fig. 7. Top row: Histogram of inter-arrival iteration count for relaxed behavior generated with log-binning. Plotting the rate this way results in histograms with the peak of the curve pointing to the Poisson rate [Sigworth and Sine 1987]. Distribution for the entire 10 million iterations (black) as well as the first 20% (grey) and the last 10% of the iterations (white) show the same Poisson rate as also confirmed with KS statistical test. The fitted lines show curve-fitting of the histogram data to $h(x) = Axp^{x-1}(1-p)$, where A is the scaling factor, x is the number of relaxed behaviors (independent variable, x-axis) and p is the probability of relaxed behavior. Bottom row: Number of Relaxed Behavior per 100K iterations over 10M iterations.

Table 7. Kolmogorov-Smirnov statistical test p-values for all chip and test combinations. These p-values were used to determine the conditions that do not result in the same rate at the beginning and end of the entire experimental run.

Chip	MP	LB	SB	R	S	2+2W
Vega	0.451	0.991	0.436	7.6e-38	0.991	2.7e-66
Quadro	0.073	4.5e-14	0.418	0.857	0.974	3.5e-08
Iris	0.988	0.419	0.126	0.447	0.231	0.546

along with its variance. If the tuning experiment were to be run on this combination the effect of a new parameter set could not be distinguished from such a jump and therefore yield false results.

5 LITMUS TEST TUNING

Due to the large combinatorial space, the parameter configurations that yield high rates of relaxed behaviors are rare and highly variable. Ideally, one would like to be able to quickly tune over this configuration space. Tuning could be as straightforward as collecting a data set similar to our EDS data set (see Table 4) and simply searching for the parameter configurations that are the most

effective. However, the running time of this approach is prohibitive (see Table 5), requiring well over 1 day of just tuning time. Industrial contacts have informed us that such a high running time is not feasible in many cases, e.g. conformance test suites.

This section aims to reduce the overall cost of tuning parameters. First, we propose a *data peeking* technique that can be used to quickly prune parameter settings that do not improve effectiveness. Next, we evaluate portable parameters and various metrics to obtain them. We show that global (i.e. a single) parameter setting can reveal relaxed behaviors in 88% of the cases that hyper-tuned parameters can, but only if the right parameter combining method is used. Finally, we examine cross-test correlations and show that parameters that are effective for one test may also be effective for another. Thus, tuning time can be reduced by only tuning across a smaller number of tests. Between these observations, we show that tuning time can be reduced by orders-of-magnitude, at only a small cost to overall effectiveness, if any.

5.1 Data Peeking on Test Tuning

Our first tuning optimization technique is *data peeking*. That is, in a test comparing two populations A and B, we can stop collecting samples early from A if we can be confident that B is more effective. In some cases, data peeking has been controversial and has led to questionable research practices, e.g. *p-hacking*. For example, an incorrect way to use data peeking is to run experiments until a statistical test reports a small enough p value. Since confidence intervals of stationary processes are monotonically decreasing, this would not be a reliable method. Rather than using p values, we use confidence intervals. Thus, two populations of unequal size can be confidently compared if their confidence intervals do not overlap.

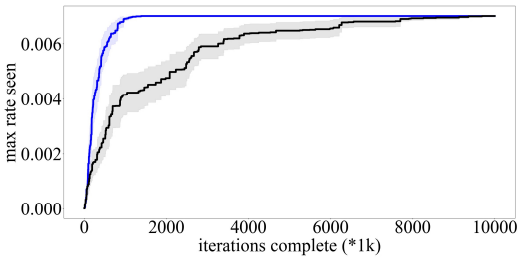


Fig. 8. Time evolution for the number of relaxed behaviors observed while tuning with data peeking. The blue bar is for our data peeking while the black for running all 10K iterations. Each bar has a band of the 95 CI. This example is for **MP** on Iris.

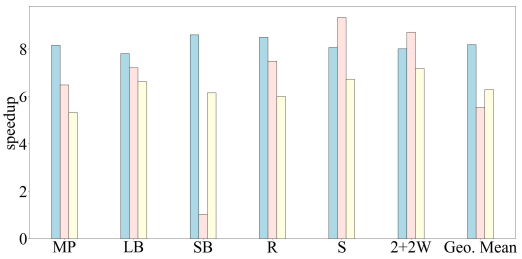


Fig. 9. Speedups obtained through data peeking for each GPU/test combination. Blue bars are for Iris, red for Vega, and yellow for Quadro.

Using this insight, we propose the following random-search tuning strategy, also demonstrated by pseudo-code in Figure 10. Select three values: (1) M, the total number of parameter configurations to explore; (2) N, an upper bound on the number of samples to collect per configuration; and (3) $P < N$, a peeking factor (lines 1-3). Next, randomly select a parameter configuration C and collect N samples (lines 22-23). Determine the rate R of observed relaxed behaviors, along with the confidence interval around R, using the normal distribution as an approximation to binomial distribution with $\mathcal{N}\left(np, \sqrt{np(1-p)}\right)$ and using the $Zscore$ for 95% confidence interval(lines 24-25; 12-18). Next, for M iterations, randomly select a new parameter configuration C^* (lines 28-29). For P iterations, cumulatively collect N/P samples from C^* (lines 33-34). At each iteration, compute R^* and its CI (lines 35-36). For each iteration, determine if the

```
M = input() # total number of parameter configurations to test
N = input() # max iterations for a given parameter configuration
P = input() # peeking factor

RANDOM_CONFIG():
    # returns a random parameter configuration

EXECUTE(CONFIGURATION, I):
    # executes I iterations using CONFIGURATION parameters and returns a list of length I
    # with an outcome for each iteration.

CI(DATA):
    # returns the size of the 95 CI bounds around the rate of relaxed memory observations

    Z = 1.96 # Z-score constant for 95% CI
    W = # num. of relaxed behaviors in DATA
    Q = # num. of observations in DATA
    return Z * sqrt((W * (1 - W)) / Q)

MAIN():
    # the initial parameter configuration is run for the full iteration count
    C = RANDOM_CONFIG()
    DATA = EXECUTE(C, N)
    R = # relaxed behavior rate in DATA
    R_CI = CI(DATA)

    # iterate through M parameter configurations
    for i in range(M):
        C* = RANDOM_CONFIG()
        DATA* = []

        # perform data peeking every N / P iterations
        for j in range(P):
            append EXECUTE(C*, N / P) to DATA*
            R* = # relaxed behavior observation rate in DATA*
            R_CI* = CI(DATA*)

            # if the new parameters show fewer relaxed behaviors than the
            # current best parameters (using 95 CI), then break early
            if R* + R_CI* < R - R_CI:
                break

        # if better, update the current best parameter set
        # (and corresponding rate and confidence interval)
        if R* > R:
            C, R, R_CI = C*, R*, R_CI*

    # return the best found configuration and corresponding rate
    # of relaxed behavior observations
    return C,R
```

Fig. 10. Pseudo code for using data peeking to accelerate tuning time.

confidence interval around R^* is below the confidence interval of R (lines 40-41). If so, the sample collecting from C^* can be stopped early, as we are rigorously confident it has a lower rate of relaxed behavior observation than C. In the case where confidence intervals overlap, then N total samples of C^* will be collected. R and R^* are compared. If R^* is larger, then C is replaced with C^* (and R with R^*) (lines 45-46). After M total iterations, we return C as our tuned parameter configuration.

Evaluation. We evaluate our approach by simulating tuning on our EDS using $M = 1000$ and $N = 10000$. We choose a peeking factor of 10, i.e. we peek every 1000 samples. We include all chip/test combinations in our evaluations; including the 4 combinations that were non-stationary (Section 4.3). We urge caution when tuning non-stationary processes, as observations can be influenced by unaccounted for variables, and thus may not converge to the global maximum. In our evaluation, we verify that all tuning simulations converged to the maximum, but this is not guaranteed for non-stationary processes. For such cases in practice, we recommend additional stability checks (e.g. our approach using the KS test in Section 4.3) during the tuning run; non-stable runs can then be restarted from the point of instability.

We run 32 simulations of tuning, both using data peeking, and also just running full iterations. We show the average (over 32 runs) observed rate of relaxed behaviors in virtual time (with one step being the time to run one iteration) with (blue) and without (black) peeking. Figure 8. The speedup is clear across the runtime.

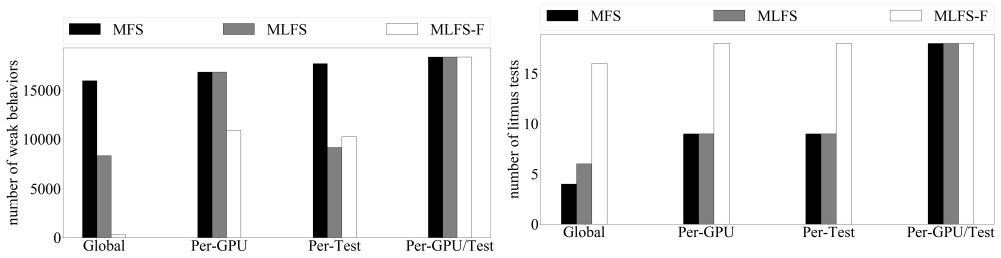
Aggregate speedups are shown in Figure 9. In all cases with early stopping, our data peeking is able to find the actual maximum parameter configuration, indicating our statistical assumptions are sound. Data peeking often yields substantial speedups in tuning, with a geometric mean speedup multiple of 8.16 for Iris, 5.53 for Vega, and 6.32 for Quadro, resulting in a time savings of 32 hours for Iris, 54 hours for Vega, and 44 hours for Quadro, using the timings from Table 5.

While there are other intelligent tuning approaches, e.g. simulated annealing and Bayesian optimization, we leave exploration to future work. Such techniques often require careful selection of meta-parameters. Prior work studying tuning for shared memory interference (in the context of application slowdowns) showed that random-search often performed just as well as more intelligent searches [Iorga et al. 2020], most likely due to non-smooth nature of many of the parameter functions. Because relaxed memory observations likely stress similar hardware components, this is likely the case for our study as well.

5.2 Portable Parameters

While we have explored methods for reducing the cost of tuning, it may be the case that a test-suite prefers *not* to tune at all, and instead, contain parameter values that work *reasonably* well across a variety of chips, even if they are not the most effective on any one chip (or test) in particular. We will refer to this concept as *portable parameters*. The notion of portability in test suites is different than what has been classically studied in performance. That is, the most important aspect is the observation of a behavior, with the magnitude of the effect being a secondary consideration. In this section, we explore various methods of providing portable parameters and evaluate how effective they are across our EDS. There are three dimensions of portability we will consider:

(1) **Global**: This is the widest dimension: it considers all tests across all GPUs and aims to provide a single parameter configuration that works well across the entire domain.
(2) **Per-GPU**: This approach aggregates results from all litmus tests per-GPU to provide N (in our case, 3, the number of GPUs) parameter configurations, specialized for each chip.
(3) **Per-test**: This approach aggregates results from all GPUs per-litmus test to provides M (in our case, 6, the number of litmus tests) parameter configurations, specialized per test.



(a) # of relaxed behaviors

(b) # of tests that revealed at least one relaxed behavior

Fig. 11. Two graphs showing the specialization/portability trade-off and effectiveness of optimization metrics

(4) **Per-GPU/Test**: This is narrowest dimension, simply providing each GPU/test combination the parameter configuration that reveals the highest rate of relaxed behaviors. There are $N * M$ parameter configurations here (18 in our case).

We consider 3 possible optimization metrics when deriving portable parameter configurations across tests or GPUs. Each successive strategy more aggressively aims to reward per-test observation over frequency magnitude. We refer to a GPU/test tuple as a *combo* when discussing these metrics:

(1) **Maximize Frequency Sum (MFS)**: Pick the parameter configuration that maximizes the sum of relaxed behavior observation frequencies across the combos in the domain.
(2) **Maximize Log Frequency Sum (MLFS)**: Pick the parameter configuration that maximizes the sum of *the log of the* relaxed behavior observation frequencies across all combos in the domain. The intuition here is that increasing the rate in test with a low frequency test is more valuable than increasing the rate in a test with an already high frequency. Combos that show 0 relaxed behaviors are ignored.
(3) **MLFS-Floor (MLFS-F)**: This strategy is more involved, and aims to harshly punish configuration that do not reveal any relaxed behaviors in a combo. The base-case is MLFS, with the addition that a rate of 0 provides a score of $-\infty$. Thus, only configurations where relaxed behaviors were observed in all combos (across their domain) will have a meaningful score. If all configurations provide a $-\infty$ score (across the domain), the metric picks the configuration with fewest instances of $-\infty$, and picks the one with the highest MLFS score.

Figure 11 evaluates these three strategies across our three domains, using our EDS. Figure 11a reports the raw number of relaxed behaviors observed across all chips/tests using different portability strategies. However, given that the observation of relaxed behaviors is more important than the rate, Figure 11b shows the number of GPU/test combinations that revealed at least one relaxed behavior with the portable parameter configuration selected. Given that we have 3 GPUs and 6 tests, the max value in Figure 11b graphs is 18. The upper bound in this analysis is the **Per-GPU/Test** dimension, as it will simply pick the parameter configuration that caused the highest rate on each GPU/test pair, regardless of the strategy. The lower bound is **Global**, as it aims to generalize (i.e. provide a single parameter configuration) across all GPUs and tests.

Figure 11a shows that MFS shows close to the same number of total relaxed behaviors regardless of whether it was tuned globally or per-GPU/Test. This is likely due to a few tests showing a large number of relaxed behaviors. On the other hand, MLFS-F shows far fewer relaxed behaviors as the domain becomes more portable, with only a small score at the global domain. However, we see the opposite in Figure 11b. This is bounded by the total number of combinations we have (18). MLFS-F reduces only slightly to 16 when generalized to global. MLFS reveals the same behaviors

Table 8. R^2 Values for Cross-Test Correlations For Each Chip. Sums are normalized to the highest value.

(a) Vega

R^2	MP	LB	SB	R	S	TPTW
MP		0.05	0.09	0	0.01	0.01
LB			0.01	0.19	0.61	0.13
SB				0.03	0.01	0
R					0.02	0.12
S						0.04
SUM	0.16	0.99	0.13	0.36	0.69	0.3

(b) Quadro

R^2	MP	LB	SB	R	S	TPTW
MP		0.034	0.126	0.089	0.041	0.029
LB			0.029	0.033	0.787	0.618
SB				0.149	0.035	0.021
R					0.037	0.027
S						0.636
SUM	0.208	0.977	0.235	0.218	1	0.866

as MFS, only slightly increasing in the global domain. Thus, we see the trade-offs in creating portable parameter sets. If raw relaxed behavior counts are most interesting, then MFS performs well. If per-test observations are prioritized, then MLFS-F performs well. We highlight that the most general domain (global) for MLFS-F keeps 88% (16 out of 18) of its efficiency from the hyper-tuned (Per-GPU/Test) configurations, showing that portable stressing configurations can be effective.

5.3 Cross Test Correlations

Optimizing tuning using data peeking is a *lossless* optimization. Here we aim to optimize tuning further by reducing the number of tests that need to be tuned. The intuition is that certain relaxed behaviors in different tests may be provoked by stressing the same hardware components. For example, both the **SB** and **R** litmus test contain a thread that executes a read-after-write sequence. The relaxed behavior that reorders this sequence could potentially be due to a store-buffer, which buffers the write operation and executes the read out-of-order.

We can explore the relationship of relaxed behavior observation rates between tests by using a correlation analysis, while acknowledging that such a relationship does not necessarily have to be linear. This yields a metric, R^2, which can determine the extent of the correlation. This value is between 0 and 1 and indicates the fraction of variability that can be explained by a common factor where 0 indicates no correlation and 1 indicates perfect correlation. Because we seed the parameter configuration random sampler the same across tests and chips, we can match up the same configurations across tests to check for correlation. Table 8 shows the R^2 values for pairwise correlation between tests for Vega, which has the lowest values, and Quadro, which has the highest levels. Notice that both of these two chips have **S** and **LB** tests highly correlated.

We note that while one could compute the multivariate correlations from a covariance matrix for the entire test suite run under various parameter configurations, one cannot make use of this in a practical sense since the independent factors obtained are not interpretable for decision making and therefore cannot be used for sorting. We thus use the pairwise R^2 metric and sort the litmus tests in order of aggregate R^2 values, providing a priority order when tuning (Table 9). That is, while an unconstrained tuning campaign would simply tune across tests, our test ordering provides a meaningful way to reduce the number of tuned tests while maintaining effectiveness across the entire suite. However, this optimization is not guaranteed to yield the best parameter settings overall, and the fewer tests are tuned, the less effective the parameters will be over all the tests.

6 TESTING CONFORMANCE AND CONSISTENCY PROPERTIES

We now turn our attention to the main use-case: conformance testing. Throughout, we have performed systematic characterizations of the effect of parameters and tuning. In order to rigorously explore conformance testing, we need a formal foundation and a methodology to examine how future bugs might appear.

To do this, we first perform a conformance testing campaign on existing OpenCL 2.0 implementations (Section 6.1).[4] In doing so, we found (and confirmed) an issue in the compiler for Iris. We explore the types of stress required to reveal the bug and how frequently it occurs. Next, we evaluate our stress and conformance tests using *mutation testing* [DeMillo et al. 1978; Jia and Harman 2011], an approach where small errors are introduced, and testing techniques are evaluated based on how well they find the errors (Section 6.2). We discuss our results in relation to industrial conformance test suites, how they compare to ours, and how they might be improved (Section 6.3). Finally, we explore two important memory model properties: SC-per-location and Multi-copy atomicity on our chips (Section 6.4).

Table 9. The litmus test priority order per-chip, and across all chips (Global). Orders are determined using sum of R^2 values. Tests higher in the priority have a higher rate of correlation across the testing set, and thus are candidates to be prioritized higher for both tuning and testing.

Chip	Litmus Test Priority
Vega	**LB** > **S** > **R** > **2+2W** > **MP** > **SB**
Quadro	**S** > **LB** > **2+2W** > **SB** > **R** > **MP**
Iris	**LB** > **MP** > **S** > **R** > **2+2W** > **SB**
Global	**LB** > **S** > **2+2W** > **MP** > **R** > **SB**

6.1 Conformance Testing

First, we explore if bugs exist in current OpenCL implementations. To do this, we create conformance tests from our 6 litmus tests. For tuning, we used the relaxed atomic annotation. For conformance tests, we strengthen the memory annotations to produce the *minimally forbidden* [Wickerson et al. 2017] variant of the test. That is, memory orders are selected such that if any one of them is relaxed, the relaxed behavior would be allowed. Across our 6 tests: **MP**, **LB**, **SB**, **R**, **S**, **2+2W**, this yields 9, 4, and 2 store instructions with memory orders `seq_cst`, `release`, and `relaxed`, respectively. For loads, there are 3, 4 and 2 with memory orders `seq_cst`, `acquire`, and `relaxed`, respectively.

We run the 6 conformance tests across our three GPUs for 100K iterations using the narrowest tuned variants of stress: i.e. the parameters found in the EDT that revealed the highest number of relaxed behaviors for the corresponding relaxed-variant of the test. Our results showed no relaxed behaviors on Vega and Quadro, but we observed 34 relaxed behaviors for **MP** on Iris: an incorrect behavior according to the specification. We have reported the issue to Intel and they have confirmed and pushed an update to the compiler mapping for release and acquire atomic instructions.[5]

Further Exploration. The **MP** test was the only test that revealed erroneous relaxed behaviors on the Iris. Our initial discovery used the least portable parameter configuration, i.e. the one selected using the Per-GPU/Test approach (see Section 5.2). With this parameter configuration, the Iris exposed 34 relaxed behaviors over 100k iterations. To explore the portability of our metrics, we ran the same **MP** conformance test using the parameter configuration selected by MLFS-F under Global portability. Under this parameter configuration, the Iris exposed 23 relaxed behaviors over 100k iterations. We then repeated the same **MP** conformance test with no stress. Under these conditions, the Iris exposed 0 relaxed behaviors over 100k iterations; this result suggests that stress of some sort is a requirement to observe any erroneous behavior for the **MP** test on Iris; however, that stress can be tuned globally and does not have to be specific to the chip or test.

6.2 Mutation Testing

Even without observing bugs on Vega or Quadro, we can still evaluate our methods using *mutation testing*: a methodology to evaluate test suites. Mutation testing begins with an implementation, *i*,

[4]Nvidia does not yet have an implementation of OpenCL 2.0. We use the mapping proposed in [Batty et al. 2016]
[5]See: https://github.com/intel/intel-graphics-compiler/commit/1c6b78c8b02d7383a1b12bc2323c9bf56380a72c

Table 10. Relaxed behavior exposure rates for different parameter configuration optimization approaches over 6 mutations. We began with the most specialized approach, Per-GPU/Test, and eliminated mutations, indicated by –, from further testing if that parameter configuration did not expose any relaxed behaviors.

Chip	i_0	i_1	i_2	i_3	i_4	i_5
Vega Per-GPU/Test	0.47216 **S**	0.00012 **SB**	0.00015 **SB**	0.0	0.01798 **2+2W**	0.00007 **MP**
Vega No Stress	0.17247 **S**	0.0	0.0	–	0.0	0.0
Vega Global MLFS-F	0.00033 **S**	0.0	0.0	–	0.00019 **2+2W**	0.0
Quadro Per-GPU/Test	0.0	0.0	0.0	0.0	0.00108 **2+2W**	0.0
Quadro No Stress	–	–	–	–	0.0	–
Quadro Global MLFS-F	–	–	–	–	0.00016 **2+2W**	–

to be tested. The implementation is then duplicated into a set I; however, each $i' \in I$ contains some modification to i' which makes it incorrect, i.e. allows erroneous behaviors that were not allowed previously. Examples include negating conditionals, changing loop bounds to be off-by-one, etc. Each i' is called a *mutant*. A test suite u can then be run on each mutant; the more mutants that u exposes (i.e. the test suite identifies a bug in i'), the more effective the test suite.

To apply mutation testing to our domain, we start with the vendor-provided OpenCL memory model implementation as i. To create our set of mutants, I, we create a memory model API wrapper that intercepts calls to the OpenCL atomic operations. We re-map the atomic operation memory orders to introduce problematic weakenings, while still being well-defined. An example is weakening a `seq_cst` annotation to a `relaxed` annotation. Considering all pairs of operations (load and store) and all memory orders (`seq_cst`, `release/acquire`, `relaxed`), there are 6 possible re-mappings that introduce weakenings: thus we have 6 mutants (called i_x where x is between 0 and 5).

In our set-up, a test suite u consists of a set of litmus tests *coupled* with stressing parameters. Our litmus test suite consists of the 6 conformance tests discussed in Section 6.1. Recall that these tests are instantiated with *minimally* forbidden memory orders; thus, any memory order weakening would allow a relaxed behavior under the OpenCL specification. We can then evaluate various stressing approaches by creating a set of test suites, U, composed of different test suites, u', which are differentiated only by their stressing parameters. This experiment will allow us to evaluate the stressing parameters based on how many mutants they expose.

Evaluation. The results of testing all mutants under the different stresses are shown in Table 10. Each row is a chip coupled with a stress configuration; each column is a mutant. The table entry is empty (-) if we did not run the mutant: we only ran mutants where we saw an initial witness in the Per-GPU/Test stresses. If the entry is non-empty and non-zero, then it contains a rate (observations divided by 100K) and the litmus test with that rate. The litmus test with the highest rate is selected.

We see that Vega is able to expose 5 out of the 6 mutants (i_0, i_1, i_2, i_4, i_5), while Quadro is only able to reveal 1 (i_4). We postulate that the low values for Quadro could be due to the Nvidia OpenCL mapping being too conservative, e.g. `seq_cst` accesses have fences before and after. Thus, tests like **SB** that have only `seq_cst` accesses require more than one weakening for two accesses not to have a fence between them. We additionally try to account for overfitting by running tests with global parameters. However, we are still only able to observe one mutant on Quadro. Future work will more closely explore how to expose more mutants on more devices.

We also find that *only one* mutant was exposed with no stress. This is compelling evidence that *some* amount of stressing is necessary in memory model conformance testing. Out of 5 observed mutants, only 2 were observable with the global portable stress (i_0, i_4). For both chips, **2+2W** observation rates reduced by at least an order of magnitude going from Per-GPU/Test stress to

Global stress. Thus, more specifically targeted stress is more capable at finding bugs. Throughout this paper, **S** on Vega has been a curiosity. Here we see that Global stress provides a lower rate of relaxed behavior observations than no stress on i_0.

6.3 Discussion on Conformance Testing

The official Khronos Group conformance test suite for Vulkan[6] contains only two litmus tests: **MP** and a coherence test (Write-after-read). It executes no stress with the tests. On the other hand, academic explorations execute massive amounts of tests over many hours. We believe it is time to bridge this gap, i.e. that there can be efficient and robust conformance testing for memory models. The results from this work suggest some small, but meaningful additions to the test suite: (1) Some amount of stress, as the Intel bug and 4 mutants required it to observe; (2) The addition of **S**, **SB**, and **2+2W** to the conformance test suite, as they revealed mutants at the highest rate; (3) a data aggregation effort, where large testing campaigns can be run once, then analyzed in studies like this. Over the last ten years, rigorous formal memory models have been adopted by industry (Vulkan, ARM, C, OpenCL). We believe it is time that their testing received a similar treatment.

6.4 SC-per-Location and Multi-Copy Atomicity

As a final set of experiments, we explore two additional memory consistency properties: SC-per-Location and multi-copy atomicity.

SC-per-Location. Intuitively, SC-per-location means that if a program consists of a single memory location, then it must have sequentially consistent behavior. Most ISAs and languages provide this property, assuming that the program is data-race free (e.g. C++ [Batty et al. 2011], X86 [Sewell et al. 2010], PowerPC and ARM [Batty et al. 2012]). However, previous work has shown that some ARM CPUs erroneously violated this property [ARM 2011], and violations have also been observed on Nvidia GPUs, i.e. in the AEA study [Alglave et al. 2015]. Given the problematic history of SC-per-location, we explore if we are able to: (1) reproduce prior results of SC-per-location violations on older Nvidia GPUs; and (2) test SC-per-location on our three newer GPUs. We use the 5 coherence tests given in [Alglave et al. 2014, fig. 6]: **CoWW**, **CoRW1**, **CoRW2**, **CoWR**, and **CoRR**. We implement the tests in OpenCL by instantiating the memory order for each access to be `relaxed`, which is sufficient (according to the specification) to disallow the relaxed behaviors.

We obtained access to an Nvidia K80 GPU through Amazon Web Services (AWS). This is not the exact same GPU chip used in the AEA study, but it has the same architecture (Kepler). Because this is an Nvidia chip, we first use the Per-GPU Nvidia stressing parameters. We run the **CoRR** test for 10M iterations and observed 15 violations, successfully reproducing the violation of SC-per-location reported in prior works. We also run **CoRR** using the global stressing parameters: under this stress variant we observed 378 relaxed behaviors in 10M iterations. Thus, even though the Per-GPU stress was tuned on an Nvidia GPU, the K80 is different enough from the Quadro that the general stress revealed relaxed behaviors at a much higher rate. We are encouraged that our general stress worked well on an untuned chip and test, however, this result also shows the danger of overfitting: parameters for one chip may not be optimal on another chip, even from the same vendor.

Finally, we run all five SC-per-location tests for 10M on our three GPUs (Vega, Iris, Quadro) using the Per-GPU stress. We do not observe any violations; thus, although older GPUs showed SC-per-location violations, the newer GPUs that we test do not.

Multi-Copy Atomicity. The second property we test is multi-copy atomicity: a property that states that two threads cannot observe memory updates in conflicting order. The idiomatic litmus

[6] see: https://github.com/KhronosGroup/VK-GL-CTS/tree/master/external/vulkancts/modules/vulkan/memory_model

test for this property is known as **IRIW** [Sorin et al. 2011, sec. 5.5.2], in which two threads both write to two different memory locations, x and y. Then two additional threads read both x and y. The behavior of interest is when one of the reading threads sees location x updated before y, and the other reader sees y updated before x.

A difficulty in testing multi-copy atomicity in language models (like OpenCL), is that thread-local reorderings of the loads should be disallowed. In ISA models, this can be enforced with dependencies between the loads [Alglave et al. 2014], however language models do not provide mechanisms to control this. Thus, we test two variants of IRIW: one with relaxed memory order (**IRIW_Relaxed**) and one where all accesses are annotated with release/acquire memory orders (**IRIW_RelAcq**). Both are allowed by the OpenCL specification, but the release/acquire variant might disallow more thread-local reorderings.[7] Observations of the relaxed behavior do not indicate a bug, but rather reveal that the chip may not be multi-copy atomic: a property considered quite weak in traditional memory models. For example, early ARM ISA memory models allowed the behavior, but disallowed the behavior since ARMv8 [Pulte et al. 2018].

The results of running the two **IRIW** variants for 10M iterations each is given in Table 11. For each chip C, we run the test with the per-GPU tuned stress for chip C, as this stress should be specialized per-chip, but portable over tests. Our results show **IRIW** relaxed behaviors on the Iris, but none on the Vega or Quadro. The relaxed variant of the test shows many more relaxed behaviors than the release/acquire variant, potentially because thread-local reorderings are disallowed by the release/acquire annotations. To account for overfitting, we also ran these tests on Vega and Quadro with the global MLFS stress parameters and still did not observe any relaxed behaviors. Thus, we are not able to provide any evidence that these two chips are not multi-copy atomic. We emphasize that our testing does not provide any guarantees: it is possible that our global stress is still overfit to tests with only two threads.

Table 11. Number of relaxed behaviors observed out of 10M iterations of the two **IRIW** tests. Iris shows weak behaviors in both tests, while Vega and Quadro do not show any.

Chip	IRIW_Relaxed	IRIW_RelAcq
Vega	0	0
Quadro	0	0
Iris	188	10

7 RELATED WORK

Testing Memory Models. There have been many prior works on testing memory models, however this work is the first that we know of that statistically analyzes the results of a large amount number of tests, under a variety of stresses, and executed across a wide range of different devices. Early work dates back to the early 90's with the ARCHTEST tool [Collier 1992]. This tool checked for thread-local reorderings (e.g. the Write-Read reorderings allowed by x86) by checking for violations when reading and writing monotonically increasing sequences to different memory locations.

TSOtool [Hangal et al. 2004] was another early testing tool, which generated pseudo random programs that contained memory access sequences that would detect violations of the TSO memory model. Using a similar philosophy, [Ta et al. 2019] randomly generates data-race free programs with manufactured memory access conflicts; these programs can then be tested for relaxed behaviors, which would violate a SC for DRF (sequential consistency for data-race free programs) property. Our approach uses formally synthesized programs, i.e. they are not randomly generated, however our stressing parameters do contain a degree of randomness. We believe this allows us to get

[7] by thread-local reorderings, we do not mean just compiler reorderings; we also mean processor-local reorderings, which can occur in the reorder buffer (RoB) or load-store queue (LSQ) of modern processors. Prior work has referred to thread-local ordering as *preserved program order* [Alglave et al. 2014]

the best of both worlds, i.e. our litmus tests are idiomatic and can come with a notion of formal coverage, while the stress can be tuned per GPU to increase the probability of seeing issues.

The `diy` memory model tool suite combines formal semantic modeling with empirical testing [Alglave et al. 2014]. The `litmus` tool executes litmus tests for a fixed iteration count and reports a histogram of observed results. `litmus` performs testing at the ISA level and has been used to test x86, ARM, and PowerPC memory models. It was later extended to test Nvidia GPUs [Alglave et al. 2015], revealing many unexpected behaviors. More recently, empirical testing has become a standard part of shared memory semantic modeling. For example, works on ARM mixed-sized memory accesses [Flur et al. 2017], JavaScript concurrency [Watt et al. 2020], and transactional memory [Chong et al. 2018] all contain a litmus testing campaigns (albeit with fixed iteration counts and no stressing). Our hope is that the statistical methodologies proposed in this work will influence the now-common empirical validation parts of these works.

GPU Memory Model Specifications. Our testing methodologies require a rigorous specification to test against; such specifications have been developed and analyzed in prior works. Most closely related is a study on how SC annotations can lead to counter-intuitive behaviors when combined with GPU scopes in OpenCL [Batty et al. 2016]. These surprising behaviors could be distilled as litmus tests and examined using our testing methodologies, e.g. to inform specification designers. However, the current scope of our work does not consider different levels of the GPU hierarchy, and thus these surprising behaviors were not considered. Instead, we choose to be exhaustive at a single level of the GPU hierarchy (inter-workgroup), and a fixed number of threads (2).

Other GPU memory model specification work includes an official formal PTX model [Lustig et al. 2019]; this specification would be an interesting target for our testing, but would only apply to Nvidia GPUs. Others have proposed stricter data-race free models for heterogeneous systems, like GPUs [Hower et al. 2014]; these models would need to be tested differently than our proposed methodology: that is, because they do not have any well-defined allowed relaxed behaviors, they cannot be tuned to maximally expose relaxed behaviors.

8 CONCLUSION

In this work, we have rigorously explored the frequency, rates and stability of relaxed behavior observations in relaxed memory model testing. Our tests span 3 GPUs, 6 litmus tests and millions of test samples. We have shown that effective stressing parameters are exponentially rare; however, we can rapidly tune to find them. We show a method for providing portable parameters that remain effective when applied across a wide domain. Our testing results led to a confirmed bug in an industry compiler and concrete suggestions for official conformance testing suites.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback which greatly improved the clarity of the paper. We thank Sreepathi Pai for the use of the GPU Zoo at the University of Rochester, which was our main source of GPUs for this work. We thank Mariusz Merecki and Jacek Jankowski at Intel for their feedback and insights around the the Iris. We thank the Khronos SPIR Memory Model TSG, (especially Rob Simpson, Alan Baker, David Neto, Jeff Bolz, Nicolai Hähnle, Graeme Leese, Brian Sumner) for their support and feedback on this work over the last year. The early-stage equipment for this project was partially funded by the MacCracken Independent Work/Senior Thesis Fund from the School of Engineering and Applied Science (SEAS) at Princeton University. This material is based upon work supported by the National Science Foundation under Grant No. 1739674. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. 2015. GPU Concurrency: Weak Behaviours and Programming Assumptions. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM. https://doi.org/10.1145/2694344.2694391

Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2010. Fences in Weak Memory Models. In *Computer Aided Verification (CAV)*. Springer. https://doi.org/10.1007/978-3-642-14295-6_25

Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2011. Litmus: Running Tests against Hardware. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer. https://doi.org/10.1007/978-3-642-19835-9_5

Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2 (2014), 7:1–7:74. https://doi.org/10.1145/2627752

ARM. 2011. Cortex-A9 MPCore, programmer advice notice, read-after-read hazards.

Mark Batty, Alastair F. Donaldson, and John Wickerson. 2016. Overhauling SC atomics in C11 and OpenCL. In *Principles of Programming Languages (POPL)*. ACM. https://doi.org/10.1145/2837614.2837637

Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. 2012. Clarifying and compiling C/C++ concurrency: from C++11 to POWER. In *Principles of Programming Languages (POPL)*. ACM. https://doi.org/10.1145/1926385.1926394

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *Principles of Programming Languages (POPL)*. ACM. https://doi.org/10.1145/1926385.1926394

Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. 2017. Kami: a platform for high-level parametric hardware specification and its modular verification. *Proc. ACM Program. Lang.* 1, ICFP (2017), 24:1–24:30. https://doi.org/10.1145/3110268

Nathan Chong, Tyler Sorensen, and John Wickerson. 2018. The semantics of transactions and weak memory in x86, Power, ARM, and C++. In *Programming Language Design and Implementation (PLDI)*. ACM. https://doi.org/10.1145/3192366.3192373

William W. Collier. 1992. *Reasoning About Parallel Architectures*. Prentice-Hall. http://www.mpdiag.com/.

R. A. DeMillo, R. J. Lipton, and F. G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (1978), 34–41. https://doi.org/10.1109/C-M.1978.218136

Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell. 2017. Mixed-size concurrency: ARM, POWER, C/C++11, and SC. In *Principles of Programming Languages (POPL)*. ACM. https://doi.org/10.1145/3093333.3009839

Kshitij Gupta, Jeff Stuart, and John D. Owens. 2012. A study of persistent threads style GPU programming for GPGPU workloads. In *Innovative Parallel Computing (InPar)*. IEEE Computer Society. https://doi.org/10.1109/InPar.2012.6339596

Sudheendra Hangal, Durgam Vahia, Chaiyasit Manovit, Juin-Yeu Joseph Lu, and Sridhar Narayanan. 2004. TSOtool: A program for verifying memory systems using the memory consistency model. In *International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society. https://doi.org/10.1145/1028176.1006710

Derek R. Hower, Blake A. Hechtman, Bradford M. Beckmann, Benedict R. Gaster, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. 2014. Heterogeneous-race-free memory models. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 427–440. https://doi.org/10.1145/2541940.2541981

Dan Iorga, Tyler Sorensen, John Wickerson, and Alastair F. Donaldson. 2020. Slow and Steady: Measuring and Tuning Multicore Interference. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. https://doi.org/10.1109/RTAS48715.2020.000-6

Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Software Eng.* 37, 5 (2011), 649–678. https://doi.org/10.1109/TSE.2010.62

Khronos Group. 2015. The OpenCL Specification Version: 2.0 (rev. 29).

Khronos Group. 2019. The OpenCL Specification Version: 2.2 (rev. 2.2-11).

L. Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* 28, 9 (1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

N. G. Leveson and C. S. Turner. 1993. An investigation of the Therac-25 accidents. *IEEE Computer* 26, 7 (1993), 18–41. https://doi.org/10.1109/MC.1993.274940

Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. 2019. A Formal Analysis of the NVIDIA PTX Memory Consistency Model. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS*. ACM. https://doi.org/10.1145/3297858.3304043

Yatin A. Manerkar, Caroline Trippel, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. 2016. Counterexamples and proof loophole for the C/C++ to POWER and ARMv7 trailing-sync compiler mappings. arXiv:1611.01507 2016.

Brian Norris and Brian Demsky. 2013. CDSChecker: Checking concurrent data structures written with C/C++ atomics. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2013*. ACM. https://doi.org/10.1145/2544173.2509514

Nvidia. 2019. CUDA C Programming Guide, Version 10.2.

Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *Proc. ACM Program. Lang.* POPL (2018). https://doi.org/10.1145/3158107

Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM* 53, 7 (2010), 89–97. https://doi.org/10.1145/1785414.1785443

F J Sigworth and S M Sine. 1987. Data transformations for improved display and fitting of single-channel dwell time histograms. *Biophysical journal* 52, 6 (Dec 1987), 1047–54. https://doi.org/10.1016/S0006-3495(87)83298-8

Tyler Sorensen and Alastair F. Donaldson. 2016. Exposing Errors Related to Weak Memory in GPU Applications. In *Programming Language Design and Implementation (PLDI)*. ACM. https://doi.org/10.1145/2908080.2908114

Tyler Sorensen, Alastair F. Donaldson, Mark Batty, Ganesh Gopalakrishnan, and Zvonimir Rakamaric. 2016. Portable inter-workgroup barrier synchronisation for GPUs. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM. https://doi.org/10.1145/2983990.2984032

Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2011. *A Primer on Memory Consistency and Cache Coherence* (1st ed.). Morgan & Claypool Publishers. https://doi.org/10.2200/S00346ED1V01Y201104CAC016

Tuan Ta, Xianwei Zhang, Anthony Gutierrez, and Bradford M. Beckmann. 2019. Autonomous data-race-free GPU testing. In *International Symposium on Workload Characterization (IISWC)*. IEEE Computer Society. https://doi.org/10.1109/IISWC47752.2019.9042019

United States Department of Energy. 2004. Final Report on the August 14, 2003 Blackout in the United States and Canada: Causes and Recommendations.

Conrad Watt, Christopher Pulte, Anton Podkopaev, Guillaume Barbier, Stephen Dolan, Shaked Flur, Jean Pichon-Pharabod, and Shu yu Guo. 2020. Repairing and mechanising the JavaScript relaxed memory model. In *Programming Language Design and Implementation (PLDI)*. ACM. https://doi.org/10.1145/3385412.3385973

John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. 2017. Automatically comparing memory consistency models. In *Principles of Programming Languages (POPL)*. ACM. https://doi.org/10.1145/3009837.3009838

Shucai Xiao and Wu-chun Feng. 2010. Inter-block GPU communication via fast barrier synchronization. In *International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE Computer Society. https://doi.org/10.1109/IPDPS.2010.5470477

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Programming Language Design and Implementation (PLDI)*. ACM. https://doi.org/10.1145/1993316.1993532