GPUHarbor: Testing GPU Memory Consistency at Large (Experience Paper)

Reese Levine
UC Santa Cruz
USA

Mingun Cho
UC Davis
USA

Devon McKee
UC Santa Cruz
USA

Andrew Quinn
UC Santa Cruz
USA

Tyler Sorensen
UC Santa Cruz
USA

ABSTRACT

Memory consistency specifications (MCSs) are a difficult, yet critical, part of a concurrent programming framework. Existing MCS testing tools are not immediately accessible, and thus, have only been applied to a limited number of devices. However, in the post-Dennard scaling landscape, there has been an explosion of new architectures and frameworks. Studying the shared memory behaviors of these new platforms is important to understand their behavior and ensure conformance to framework specifications.

In this paper, we present GPUHarbor, a widescale GPU MCS testing tool with a web interface and an Android app. Using GPUHarbor, we deployed a testing campaign that checks conformance and characterizes weak behaviors. We advertised GPUHarbor on forums and social media, allowing us to collect testing data from 106 devices, spanning seven vendors. In terms of devices tested, this constitutes the largest study on weak memory behaviors by at least 10×, and our conformance tests identified two new bugs on embedded Arm and NVIDIA devices. Analyzing our characterization data yields many insights, including quantifying and comparing weak behavior occurrence rates (e.g., AMD GPUs show 25.3× more weak behaviors on average than Intel). We conclude with a discussion of the impact our results have on software development for these performance-critical devices.

CCS CONCEPTS

• **Software and its engineering** → **Empirical software validation**; • **Computing methodologies** → **Parallel programming languages**; **Graphics processors**.

KEYWORDS

memory consistency, GPUs, mutation testing

ACM Reference Format:
Reese Levine, Mingun Cho, Devon McKee, Andrew Quinn, and Tyler Sorensen. 2023. GPUHarbor: Testing GPU Memory Consistency at Large (Experience Paper). In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598095

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07.
https://doi.org/10.1145/3597926.3598095

Table 1: The GPU vendors and devices included in the study. Overall, we ran almost 400 billion iterations of weak memory tests on 106 devices, of which 58 we have confirmed to be unique models. We observed over 35 million weak behaviors, with the rates per device and vendor characterized in Sec. 4.

Framework	Vendor	Devices (Unique)	Tests	Weak Behaviors
WebGPU	Intel	26 (17)	105.3b	0.2m
	Apple	26 (6)	104.4b	9.7m
	NVIDIA	31 (18)	125.3b	10.8m
	AMD	15 (9)	60.4b	14.7m
Vulkan	Arm	2 (2)	51.6m	18.2k
	Qualcomm	4 (4)	17.6m	27.2k
	Imagination	1 (1)	6.1m	0
	NVIDIA	1 (1)	49.6m	454
		Total: 106 (58)	**395.5b**	**35.4m**

1 INTRODUCTION

The end of Dennard Scaling has brought about an explosion of multi-core architectures that improve application performance through large-scale parallelism. Graphics Processing Units (GPUs) exemplify this trend and are now integral components of many systems, from smartphones to large HPC supercomputers. While GPUs were previously primarily used for graphics applications, they now have applications in a variety of areas including machine learning [42] and particle simulations used in drug development [38]. GPUs are even being used for security and safety-critical applications such as encryption [37] and self-driving cars [13], making safety and correctness an increasing concern on these devices.

Because GPUs are produced by several vendors (NVIDIA, AMD, Intel, etc.) and evolve rapidly, many different devices are currently deployed. These devices vary both in their performance, as well as their functional behavior. To account for this, the community has developed portable GPU programming frameworks, such as Vulkan [22] and WebGPU [51], as unified abstractions to target these diverse devices.

Memory consistency specifications (MCSs), which define the semantics of shared memory operations, are an important part of these abstractions. While MCSs provide many guarantees, such as atomicity and coherence, they often allow an architecture to implement *weak* memory behaviors to improve efficiency [34]. For example, x86's relaxed MCS [44] allows store buffering behaviors, in which a processor may buffer the stored values before flushing them to a shared memory location; as a result, another processor may observe the buffered store occurring out-of-order.

Because relaxed MCSs can be complex and nuanced, there is a history of platforms (compilers and architectures) containing MCS conformance bugs [1, 4, 25, 30, 31]. That is, the MCS provides a guarantee that the implementation does not honor. Due to the non-determinism of concurrency, MCS bugs may occur extremely rarely, or only when provoked, e.g., by side-channel stress [46]. Apart from conformance, a device's weak behavior *profile*, i.e., the frequency at which allowed weak behaviors occur and how system stress influences this frequency, is also a useful metric. For example, this can be useful in developing conformance testing strategies [28] and enables developers to reason about tradeoffs between accuracy and performance in approximate computing that judiciously elides synchronization [35, 41, 43].

Unfortunately, previous GPU testing work had limited scope, testing only a small number of devices [25, 28], with the largest study testing eight devices [1]. These approaches did not scale due to the difficulty of portable GPU application development and deployment, e.g., while frameworks like OpenCL [21] are portable in theory, there are many difficulties in practice [47]. Consequently, little is known about the MCS conformance and weak behavior profiles *at large*. This is especially problematic as portable GPU frameworks depend upon many layers and environments (e.g., architectures, compilers, runtimes, operating systems, etc.); it is difficult to extrapolate insights from a small number of platforms tested in controlled environments to the diverse universe of deployed GPUs.

GPUHarbor. In this paper, we present a large-scale study of GPU MCS testing, which, to the best of our knowledge, tests 10× more devices than previous studies. Figure 1 summarizes our study, including the number of GPUs that we tested (106), broken down by two frameworks (WebGPU and Vulkan) and seven vendors (Intel, Apple, NVIDIA, AMD, Arm, Qualcomm, and Imagination). This scale is empowered by GPUHarbor, a new cross-platform GPU MCS testing tool suite. GPUHarbor includes two front-ends, a browser web app (using WebGPU) and an Android app (using Vulkan). We advertised our web app on campus forums and social media to obtain a significant number of WebGPU results. We test much fewer Vulkan devices as our Android app is not yet widely accessible on the Google Play Store, but in Sec. 7 we discuss how we will enable larger mobile studies on both Android and iOS.

GPUHarbor uses *litmus tests*, small concurrent programs that check for load/store reordering corresponding to weak memory behaviors. Current GPU MCS testing tools execute litmus tests many times in succession to check conformance and characterize devices [1, 25, 46]. However, these prior approaches have several shortcomings: (1) they are implemented in vendor-specific languages, e.g., CUDA; (2) they require expert users to build, configure, and execute tests on each device, e.g., as is the case for OpenCL; or (3) litmus tests were embedded in vendor-specific stress testing environments and thus, would not execute efficiently on other devices. This cumbersome litmus testing workflow made it infeasible to perform a large-scale study. In contrast, GPUHarbor defines litmus tests using a neutral configuration (written in JSON), which it compiles to a portable shading language (WGSL [50] or SPIR-V [20]). The resulting litmus testing application then tunes the testing stress automatically. The net result is a fully automated and easy-to-use tool for GPU MCS testing at large. Table 1 shows how many weak memory litmus test iterations were run and how many weak behaviors were observed in our study.

We perform the following two investigations on our data set: (1) we examine the results of MCS conformance tests and find two new bugs in mobile device GPUs from Arm and NVIDIA, and (2) we characterize weak memory behavior profiles, e.g., the rates at which allowed weak behaviors occur and their sensitivity to system stress. Additionally, we provide several analyses on the weak memory profiles. First, we comment on how per-vendor average profiles compare; for example, AMD shows an average percentage of 1.5% weak behaviors, while Intel shows only .06%. We then cluster different GPUs and find that, surprisingly, cross-vendor devices often have similar profiles, while devices from the *same* vendor sometimes have vastly different profiles. Finally, we discuss how the wide range of different profiles we observed can impact testing strategies and the implementation of synchronization algorithms.

Contributions. In summary, our contributions are:

(1) **Tooling**: We introduce GPUHarbor, a new cross-platform GPU MCS testing tool with accessible web and Android interfaces (Sec. 3).
(2) **GPU MCS Weak Behavior Characterization**: We conduct a large GPU weak memory characterization and conformance testing study, collecting data from 106 GPUs (Sec. 4).
(3) **Conformance Testing and Analysis**:
  (a) We discover two unreported bugs in Arm and NVIDIA devices (Sec. 5.1).
  (b) We analyze statistical similarities across GPUs and describe the impact on testing strategies and device fingerprinting (Sec. 5.2).
  (c) We discuss how weak behavior profiles impact the development and testing of synchronization algorithms on GPUs (Sec. 5.3).
(4) **Lessons Learned:** We detail the lessons learned while designing and running this study, providing a guide to other researchers seeking to implement similar large experimental explorations (Sec. 6).

All of the data we collected as part of our study, and the tools used to do so, are available as part of our artifact [27]. In addition, GPUHarbor's web interface is hosted by UC Santa Cruz and can be found at https://gpuharbor.ucsc.edu/webgpu-mem-testing/.

2 BACKGROUND

In this section we provide an overview of memory consistency specifications (Sec. 2.1), define the litmus tests we run and how they allow reasoning about relaxed memory models (Sec. 2.2), and introduce GPU programming concepts from the WebGPU and Vulkan GPU frameworks, including descriptions of their MCSs (Sec. 2.3).

2.1 Memory Consistency Specifications

Today, the memory consistency specifications for architectures, e.g., x86 [44], and languages, e.g., C++ [7], are formalized using mathematical logic. This formalism represents shared memory program executions as a set of memory operations, e.g., reads, writes, and read-modify-writes, and relations between these events,

e.g., happens-before (*hb*). Allowed executions constrain define constraints on some of these relations, e.g., *hb* is required to be acyclic. The strongest MCS is sequential consistency (SC) [26], which states that concurrent program executions must correspond to a total *hb* order such that the order respects the per-thread program order, allowing events from multiple threads to be interleaved. In relaxed MCSs, the *hb* relation is a partial order, allowing various weak behaviors (i.e. executions that are *not* SC) if shared memory operations on multiple threads are not synchronized.

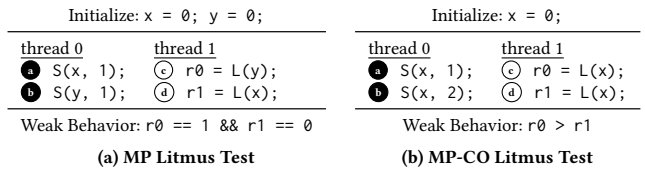
There is a large body of work focused on formalizing MCSs, including a model for Vulkan's [19]. WebGPU generally follows the Vulkan MCS, with prior work [28] formalizing portions of its MCS necessary for reasoning about simple litmus tests. However, for this work it is not necessary to understand the full formalization of the WebGPU and Vulkan MCSs, so we describe the necessary subset of the specification briefly and informally. In addition, we follow prior work on MCS testing [25, 28] and consider only *trivially data-race-free* programs where all operations are atomic, as our intention is not to test the behavior of programs with undefined semantics (caused by data races).

Our Target MCS. Because Vulkan is one of several backends to WebGPU, the MCS for WebGPU is a subset of the MCS for Vulkan. In order to provide a unified study across both frameworks, we target only the WebGPU MCS, which we then map to its Vulkan counterpart. The WebGPU MCS provides very little inter-workgroup synchronization due to the diversity of backends it targets, with the weakest backend being Apple's Metal [6], which provides only `relaxed` atomic operations. These operations, which come from the C++ memory model [7], compile to plain loads/stores at the architectural level, but at the language level provide few synchronization guarantees between threads.

The one inter-workgroup MCS property provided by WebGPU atomics is *coherence*, which states that memory accesses to a single location must respect sequential consistency; sometimes called SC-per-loc [4]. However, memory accesses to disjoint addresses are allowed to be reordered. Mapping these WebGPU atomics to Vulkan is straightforward; all WebGPU atomic accesses are simply mapped to SPIR-V atomic accesses with a `relaxed` memory order. While our testing campaign considers only `relaxed` memory accesses, Vulkan allows additional memory orders; specifically, `acquire` and `release`. While the precise semantics of these memory orders is complex, especially when combined with other `relaxed` atomics, we note that they are required to implement the required synchronization in many common concurrency constructs, such as a mutex. [1] The `lock()` method needs to execute an `acquire` atomic operation when the mutex is obtained and the `unlock()` method requires executing a `release` atomic operation. If a mutex is implemented without these memory orders, it is possible to violate mutual exclusion, as we show in Sec. 5.3.

2.2 Litmus Tests

Litmus tests are small concurrent programs that illustrate [45], compare [29, 48], and empirically test [1, 3, 25] MCSs. These tests contain a condition on the final state of local variables and memory values that checks for weak behaviors. For example, the program in Fig. 1a is known as the message passing (**MP**) litmus test, in which one thread writes to a memory location x (ⓐ) followed by a write to y (ⓑ), while a second thread reads from y (ⓒ) and then x (ⓓ).



(a) MP Litmus Test **(b) MP-CO Litmus Test**

Figure 1: The weak behavior in the MP litmus test is allowed by many relaxed MCSs. On the other hand, the weak behavior in the closely-related MP-CO litmus test violates coherency and is disallowed by every major MCS that we know of. Prior work [28] has shown that tuning system stress for the allowed MP test can be used to design effective tests for finding bugs related to the MP-CO test.

As mentioned earlier, in this work, we assume that all of the memory operations in a litmus test are *atomic*, which in languages that follow the C11 style MCS [7] ensures that the semantics of shared memory operations are well-defined. Additionally, unless explicitly noted otherwise, we consider these atomic operations to have a `relaxed` memory order, which allows compilers and hardware to aggressively optimize their execution.

The condition underneath the test shows an outcome that only occurs in relaxed executions. In this case, the behavior corresponds to an execution where the read of y returns 1 but the read of x returns 0. While some relaxed MCSs do not allow this behavior, e.g., the x86 MCS [44], many other relaxed MCSs, especially ones for languages like C++ [7], do allow the behavior. As mentioned earlier, our target WebGPU MCS does not provide any guarantees outside of coherence, and thus the two memory accesses per thread (which target disjoint addresses) are allowed to be reordered. In cases where the weak behavior is allowed (both by the MCS and the implementation), the rate at which this behavior is observed on real systems is highly dependent on *system stress*. Early GPU application development work did not observe any weak behaviors, despite specifications allowing them [12]. However, later work added specialized system stress around the test execution and revealed many cases of surprising weak behaviors [1, 46].

Executing litmus tests on deployed systems can be used for two purposes, which we will illustrate using a litmus test L that can exhibit a weak behavior execution e, and an MCS S.

(1) **Conformance testing**: if e is *disallowed* in S then we can check implementations of S. That is, if a platform p claims to implement S, then we can execute L many times on p, checking for e. The observation of e would indicate a bug.
(2) **Profiling weak behaviors**: if e is *allowed* on S, and a platform p claims to implement S, then we can execute L many times on p to understand the extent to which that platform allows e. In some cases, p might not show e empirically, or

[1] WebGPU does not provide inter-workgroup `acquire` and `release` memory orders, so it is not currently possible to implement a well-specified mutex in WebGPU.

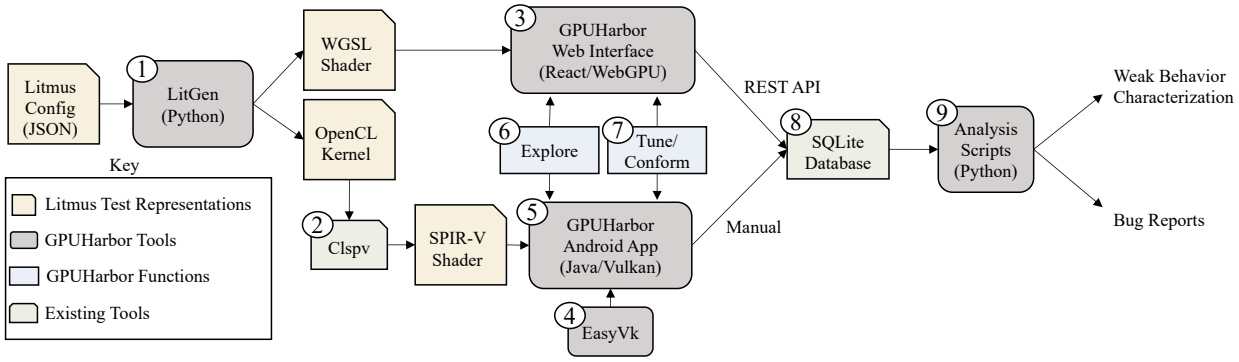


Figure 2: An overview of the tooling flow for GPUHarbor, with WebGPU and Vulkan MCS targets.

maybe *e* appears more frequently under a certain configuration of system stress. A collection of this type of data creates a weak memory profile for *p*.

Prior work [28] has utilized weak memory profiles in highly tuned conformance testing. In that work, it was shown that allowed **MP** executions could be used to tune system stress for disallowed behaviors in associated conformance tests. For example, the **MP-CO** litmus test, shown in Fig. 1b, is similar to **MP**, except that every memory access targets the same memory address and different values are stored (required to identify a weak behavior). Given that there is only one address used in **MP-CO**, the weak behavior in this test is disallowed under coherence, and thus in the WebGPU MCS. If certain system stress reveals weak behaviors in the allowed **MP** litmus test, then, in the case where a platform contains a bug, it is likely to reveal the buggy behavior in the **MP-CO** conformance test. In Sec. 3.1 we show the litmus tests used in our experimental campaign, and in Sec. 5.1 we illustrate the effectiveness of the approach of prior work [28] by describing two new bugs.

2.3 GPU Programming

This study targets two cross-platform GPU frameworks, Vulkan and WebGPU. Vulkan is a modern graphics and compute API that can be run on many Linux, Android, and Windows devices, and can target Apple devices through the MoltenVK [23] portability layer. WebGPU is designed to run in browser environments and is compiled to different backends depending on the operating system of the device (Direct3D [33] on Windows, Vulkan on Linux/Android, and Metal [6] on Apple devices).

Both Vulkan and WebGPU define their own programming languages, called SPIR-V and WGSL respectively. Programs written in these languages are called *shaders* and run on the GPU, while the APIs used to allocate memory on the GPU and dispatch shaders are written in the language of the host device, commonly C++ for Vulkan and JavaScript for WebGPU. In this work, we discuss the complexities of writing tools that must be implemented in different languages and how future development (Sec. 7) could ease the difficulty of cross-platform GPU MCS testing.

GPU Execution Model. GPUs run thousands of concurrent threads (*invocations* in Vulkan and WebGPU) organized hierarchically and executed in a single-instruction, multiple-thread (SIMT) format. To support this execution model, in WGSL and SPIR-V threads are partitioned into discrete *workgroups*, with built-in identifiers used to query a thread's workgroup id. Workgroups are limited in size (e.g. 1024 in CUDA, with limits varying depending on the device in WGSL/SPIR-V) and have access to an efficient shared memory region. A group of threads organized into workgroups and running on the device is called a *grid*, with the number of threads per workgroup and the number of workgroups specified at dispatch time. All threads in the same dispatch have access to a global memory region.

While our target MCS was discussed in the previous section, we note that GPU atomic operations can be annotated with a *memory scope*. Two common scopes in Vulkan and WebGPU are *workgroup*, which specifies that synchronization occurs only between threads in the same workgroup, and *device*, which specifies that synchronization occurs across all threads executing on the device.

Threads within workgroups generally have access to efficient primitive barrier operations, e.g., `workgroupBarrier` in WebGPU. However, highly optimized implementations of important parallel routines (e.g. inter-workgroup prefix scans [32]) rely on fine-grained inter-workgroup communication. Thus, like prior work [25, 28], we see a more imminent need for testing MCS properties at the inter-workgroup level; which we keep as our sole scope for this work. Similarly, GPU programs have several different memory types, e.g. whether it is shared-workgroup memory or device-wide memory. Given that we consider only inter-workgroup interactions, we only consider device-wide memory.

3 SYSTEM OVERVIEW

Building on approaches in prior work [28], we discuss our testing campaign (Sec. 3.1) and the development of our MCS testing tools that are easily accessible on a wide range of devices, summarized in Fig. 2. We overview each stage of the tooling, starting with litmus test generation (Sec. 3.2), moving on to the design of GPUHarbor's web interface and Android app (Sec. 3.3). We end the section by describing our data collection process (Sec. 3.4).

3.1 Litmus Test Selection

The tests we utilize in our study build off of the MCS mutation testing strategy used in [28]. We use 32 *mutants*, out of which 24

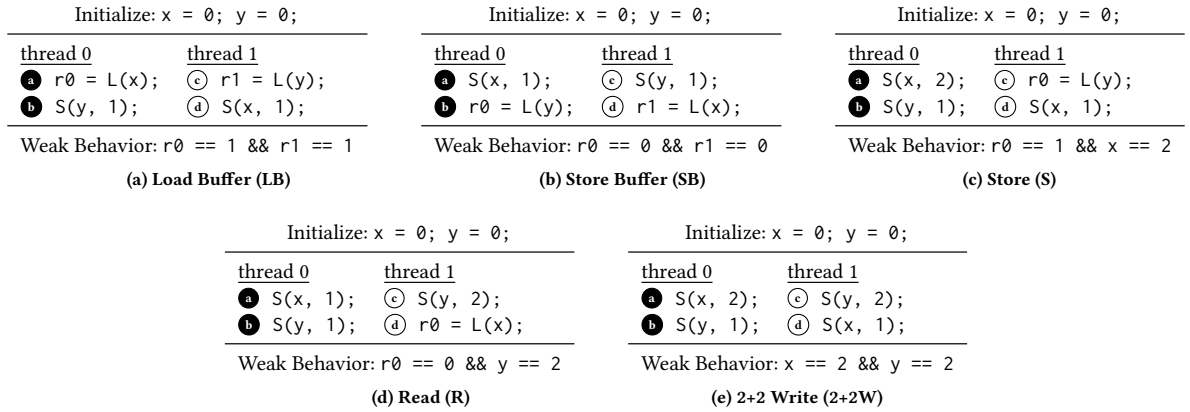


Figure 3: These litmus tests, along with MP from Fig. 1a, represent six classic weak behaviors allowed by relaxed MCSs. `S` and `L` signify a `relaxed` atomic store and load, respectively.

are litmus tests with weak behaviors allowed by the WebGPU MCS. The mutants are used to find effective system stress to then run the conformance tests. Our results analysis focuses on characterizing the rates of weak behaviors of six of the mutants, one of which is **MP** (Fig. 1a), with the other five shown in Fig. 3. These tests enumerate all the combinations of four instructions on two threads that can lead to weak behaviors. Thus, they capture testing for all pair-wise memory reorderings. For example, the **SB** test checks for store-load reorderings, while the **LB** test checks for load-store reorderings. Additionally, these tests capture synchronization patterns used in common concurrency algorithms like a compare-and-swap spinlock. Because of this, prior work has also focused on these tests and has shown their utility in finding bugs in both applications and MCS implementations [25, 46].

Once the mutants are run, we use the weak behavior profile of a device to determine an effective system stress configuration to run conformance tests under. We utilize the 20 conformance tests from [28]. As a concrete illustration using one mutant and conformance test, we would run the **MP** test under many different system stress configurations to build a weak behavior profile. We then use the most effective configuration at revealing **MP** weak behaviors to run a closely related conformance test, e.g., **MP-CO** (Fig. 1b). This approach was shown to be effective at finding bugs in prior work [28] and we further show its effectiveness by discovering two new bugs: a violation of **MP-CO** on Arm devices and a violation of **MP-CO** on an NVIDIA device (see Sec. 5.1).

3.2 Litmus Test Generation

We now discuss our tooling that generates and runs our testing and characterization campaign. Litmus test behaviors are non-deterministic and sensitive to system stress. Due to this, the shaders that run the litmus tests contain not only the actual litmus test instructions, like those in Fig. 3, but take in a number of parameters and provide functions that are used to construct system stress.

To provide a standardized interface for defining litmus tests in different GPU languages, we built a tool, Litmus Generator (LitGen, ① in Fig. 2), which is similar to previous litmus testing tools [3] but is specifically targeted to create GPU programs with system stress, as was shown is necessary for testing GPU MCSs [1, 25, 28]. LitGen takes litmus tests that are written in an abstract format, currently JSON, that specify the actions of the test (e.g. loads and stores) and the possible behaviors of the test, with a special designation being given to weak behaviors. The tests used in this work were all manually specified, as they are relatively small, but LitGen could be integrated with other tools that use formal models to generate litmus tests, e.g., [2, 48], which would provide more automation and account for more complicated tests and MCSs. LitGen outputs a *test shader*, which runs the test alongside system stress developed in prior work [25, 28], and a *result shader*, which aggregates the observed behaviors of the test.

The result shader is generated separately from the test shader for several reasons:

(1) Some tests, like **2+2W** (Fig. 3e), examine memory locations for weak behaviors after all threads have finished executing the test. To avoid relying on synchronization features (some of which we are trying to test), we instead pass the test memory buffer into a new result aggregation shader, which executes after the test shader.
(2) LitGen implements parallel testing, described in [28], which runs thousands of *instances* of each litmus test concurrently. Thus, it is natural to leverage the inherent parallelism of the GPU to also aggregate the many results, which otherwise may be time-consuming to do on the CPU, especially since it requires copying memory from the GPU to the CPU.

Currently, two backends exist for LitGen. The tool outputs WGSL shaders directly, as WGSL is a text-based language. SPIR-V, on the other hand, is a low-level representation similar to LLVM, increasing its flexibility but making code generation more complex. Therefore, for Vulkan backends LitGen first outputs OpenCL, a compute-focused GPU language, which is similar in syntax to C++. Then, it utilizes Clspv [15] (②), a prototype compiler from OpenCL to SPIR-V, to generate the shader used in the Android app.

As WebGPU is primarily browser-based while Vulkan runs on native devices, we currently maintain litmus testing driver programs

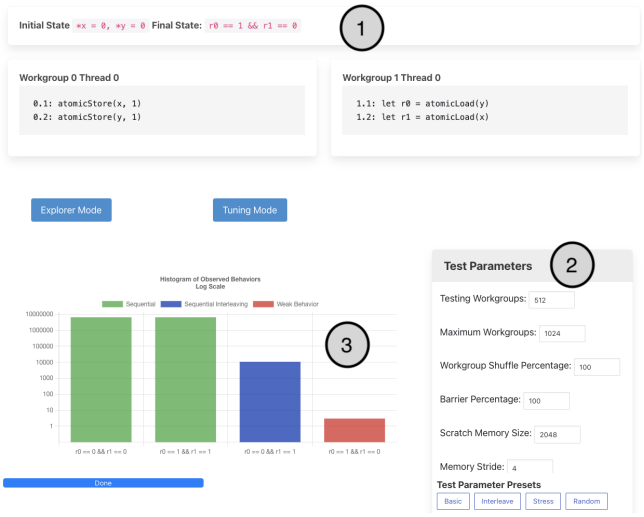


Figure 4: A screenshot of GPUHarbor's web interface showing the "Explore" page for the MP litmus test.

in two languages. WebGPU exposes a relatively simple JavaScript API, on which we build our web interface (③). Vulkan's native C/C++ API is more complex, so to simplify this process, we have built a Vulkan compute wrapper which we call EasyVk (④). This library exposes a simple interface where buffers are defined and shaders dispatched to the GPU in a few lines of code. EasyVk is integrated into the Android app and included as part of our artifact.

3.3 GPUHarbor Design

Previous GPU MCS studies have been limited in reach due to the difficulty in deploying cross-platform GPU applications [47]. One of the reasons for this is the fractured landscape of GPU development. NVIDIA's popular CUDA framework [36] is used for many data science applications but is only supported on NVIDIA GPUs. OpenCL [21] was introduced in 2009 by Apple as a cross-platform standard; however, today Apple no longer supports OpenCL and instead requires developers to use their proprietary Metal API [6].

Another issue with previous GPU MCS testing approaches has been a reliance on expert users to run testing campaigns. For example, using OpenCL would require users to install the required drivers, build the application under a specific environment, etc. To collect data from the diversity of devices necessary to gain confidence in cross-platform GPU frameworks, tools must minimize the friction in setting up and running tests. The tools we introduce here are easily distributed applications with a user-friendly interface on top of new GPU frameworks so that even non-technical users can collect data on their devices and submit results for analysis. To this end, we introduce GPUHarbor: a GPU MCS testing tool with two widely supported and accessible frontends, a web interface and an Android app (③ and ⑤ in Fig. 2).

GPUHarbor's web interface and Android app have a common design with two functions: exploring (⑥) and tuning/conforming (⑦). "Explore" pages run specific litmus tests, display histograms of results, and provide the ability to adjust various parameters that control system stress. When tuning and conforming, a set of tests are chosen to run with multiple random system stress configurations, searching for configurations that maximize the rate of weak behaviors and uncover bugs in MCS implementations. While this study includes the largest collection of data on mobile GPU MCS behaviors, in Sec. 7 we discuss future work that could increase the reach of mobile GPU MCS testing even further.

Exploring. Figure 4 shows a screenshot of GPUHarbor's web interface explore page for the **MP** litmus test after the test has been run with relatively high systems stress on a MacBook Pro with an integrated Intel Iris GPU. The top of the page (①) includes a description of the test and pseudocode showing the test instructions. The right-hand side (②) includes an editable list of the parameters that define system stress, along with several presets. When the test is running, the histogram (③) updates in real-time with the number of times each behavior is observed. The progress bar gives an estimate of how much longer is left to run, based on the speed of previous iterations.

The green bars correspond to sequential behaviors, where one thread runs entirely before the other. The blue bar corresponds to interleaved behaviors, where actions from each thread are interleaved (e.g. leading to the behavior `r0 == 0 && r1 == 1` in the **MP** litmus test). The red bar corresponds to weak behaviors; in this run, three **MP** weak behaviors were observed out of over 13 million test instances, so the histogram shows behaviors (using a log scale, as weak behaviors are relatively rare).

Tuning and Conforming. Both the web interface and the Android app can be used to tune system stress, as in [28]. When tuning, a set of tests can be selected, with presets available for weak memory tests (e.g. those in Fig. 3) and conformance tests, e.g., to test coherence. Testing options like the number of configurations, the maximum number of workgroups, and other parameter overrides can be modified to run different experiments and check specific tests without redeploying any code.

To collect data from volunteer users across a diverse set of devices, we strive to minimize the options users have to configure. This reduces the chances of errors and provides us with a standardized dataset to analyze. The web interface's tuning page, therefore, includes a tab that exposes no configuration options, but instead shows only a few buttons: one button that starts a combined tuning/conformance run with default parameters: and another button that pulls up a submission form, which submits the results along with some (optional) contact information. Our results are all anonymized; contact details were only collected if users wanted to be informed about the outcome of the study. Before submitting, users agreed that their anonymized results could be aggregated, reported on, and released as part of this study.

3.4 Data Collection

To submit test data, the web interface communicates with a backend service that exposes an API for submitting results and inserting them into an SQLite database (⑧ in Fig. 2). The data is then analyzed using Python scripts (⑨). The Android app is not yet available on the app store nor is it integrated with the SQLite backend, so results are manually copied off of the device for analysis. In Sec. 7 we

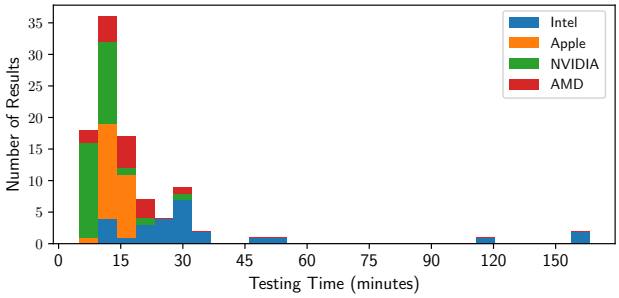


Figure 5: Histogram showing the time spent testing WebGPU's MCS on each device using GPUHarbor's web interface. Each bin is broken down by vendor.

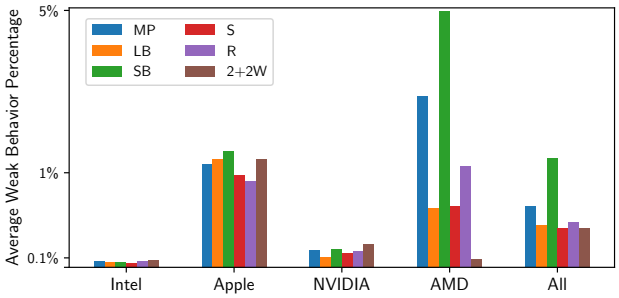


Figure 6: Data showing the average rate of weak behaviors across vendors when running litmus tests using WebGPU.

discuss how we can reduce the friction for submitting mobile app results, and thus, increase the reach of future studies. Nevertheless, our study of eight devices is the largest testing campaign of mobile GPU MCS behaviors of which we are aware.

While system stress configurations are generated randomly, we would like to ensure that the configurations run on different devices are the same for data analysis purposes. That is, if different GPUs are tested with the same stress configurations, we can compare how the different devices behaved under the same stress. We ensure this by integrating a seedable Park-Miller random number generator [39] into both the web interface and the Android app and using the same seed when running all of our tuning experiments.

By default, browsers only expose limited information about the user's GPU without turning on vendor-specific development flags due to privacy and security concerns around fingerprinting [52]. In order to have as much information as possible about our data, we included instructions asking users to temporarily enable flags so we could collect detailed GPU information. Of the 98 results we collected, 67 included the exact GPU tested. The other 31 results did not specify the exact GPU, but included only the vendor and a string describing the GPU architecture, such as "intel gen-9" or "nvidia ampere". All Apple devices reported an architecture of "common-3", making it impossible to immediately distinguish M1's vs M2's. However, we show in Sec. 5.2 that our data can be used to infer device information, hindering the ability of browsers to hide the specifics of a user's GPU.

4 INITIAL RESULTS: WEAK BEHAVIOR CHARACTERIZATION

To collect data from as many sources as possible, we disseminated the link to GPUHarbor's web interface to the general public, utilizing campus forums and social media, and ran the Android app on eight devices that we could physically access. As shown in Tab. 1, we collected data from millions of tests; each test used a randomly generated system stress configuration (we used 50 configurations on the web interface and 150 on the Android app). In each configuration, tests were run millions of times based on a randomly generated number of workgroups and threads per workgroup.

To ensure data integrity, we implemented a checksum algorithm that verified we saw the expected number of overall behaviors based on the system stress configuration. The testing duration was also recorded, however, we ran into one issue here. Some computers went to sleep in the middle of the tests, suspending the browser's process and leading to extremely long recorded test times. To overcome this, we recorded testing time on a per test/configuration basis; we then filtered the results so as to not include any test/configuration durations over one minute. We note that each individual test runs quickly (e.g. in less than 5 seconds), thus, runs that were over one minute were most likely when the computer went to sleep. To approximate the length of the test that was suspended, we used a neighboring test's time.

One consideration for collecting data from the wider public is that we cannot afford to run tests for hours at a time. Previous work targeted only a few devices, running tests on one device for a minimum of 36 hours [25] or 2 hours [28]. However, asking volunteer users to leave their browsers and computer open for that long is impractical and would certainly decrease the number of submissions. Therefore, we heuristically chose the number of test environments and iterations per environment, aiming for the tests to finish in 10-20 minutes.

Figure 5 shows the distribution of testing time on our web interface, broken down by vendor. The results show that NVIDIA devices were the fastest on average, mostly running all tests in under 15 minutes. On the other hand, Intel devices ran slower, with two older Intel GPUs taking over an hour and a half to complete.

In the rest of this section, we analyze our WebGPU and Vulkan data to characterize the rates at which weak behaviors occur on devices from different vendors. These initial results motivate three research questions, which are explored in depth in Sec. 5:

(1) Do MCS bugs exist in the wild, especially in GPUs which are relatively untested (Sec. 5.1)?
(2) Can our characterization data be used to identify similarities between GPUs (Sec. 5.2)? If so, then our data can be used to develop new testing strategies or to expose potential new browser fingerprinting vulnerabilities.
(3) How can a weak behavior characterization study be used in programming guides for implementing synchronization constructs, e.g. mutexes (Sec. 5.3)?

4.1 Weak Behaviors in WebGPU

Figure 6 shows the average rates of observed weak behaviors for the six litmus tests of Fig. 3 (plus **MP**) in the test environment that maximizes the rate on each device broken down by test and vendor. As described in Fig. 1, we have data from at least 15 devices from each vendor. The overall testing time across all 98 devices was 31.1 hours, an average of 19 minutes per device.

Devices from all vendors showed weak behaviors on each litmus test. In all but two cases, observing weak behaviors was all or nothing; if a device revealed weak behaviors on one litmus test, it revealed weak behaviors on all of them. In contrast, on a device implementing x86's TSO MCS, we would expect to only see store buffering behaviors. However, unlike x86, GPU devices do not provide low-level details, such as the hardware-level MCS, thus it was not clear what types of weak behaviors we would observe. These results show that many GPUs implement very relaxed memory models, in contrast to stronger CPU architectures like x86 TSO.

Intel devices tended to have the lowest rate of weak behaviors, with just over half of them (15/26) revealing weak behaviors on each test. The median rate of weak behaviors on Intel devices was even lower than their average, around .02% for each test. No Intel device showed a rate of weak behaviors above 1% on any test.

NVIDIA devices revealed weak behaviors at a relatively low rate. Our results include results from NVIDIA's Kepler (2012), Maxwell (2014), Pascal (2016), Turing (2018), and Ampere (2020) architectures, with a majority being the more recent Ampere. Older devices generally showed fewer weak behaviors, with the minimum on each of the six tests being Kepler and Maxwell devices. However, one outlier is that the maximum rate of **SB** behaviors (.73%) was seen on a Kepler device. Interestingly, that device was also the only device not to observe any weak behaviors on **S**, **LB**, and **2+2W**. The only other device not to reveal weak behaviors on a test was a Quadro K620 with a Maxwell architecture, on **MP**, **R**, and **SB**.

Apple devices were consistently weak, revealing weak behaviors on every device and test, generally at a higher rate on all tests than NVIDIA devices but with less variation than AMD devices. Apple GPUs have only been recently built into non-mobile devices, so these results represent the first comprehensive evaluation of the weak behaviors on Apple GPUs. We don't have the specific name of every Apple device, but we were able to collect enough information to show we had results from Apple M1 (basic, Pro, Max) and Apple M2 (basic, Pro) devices.

AMD devices were also very weak, with 100% of devices showing weak behaviors on every test. The clear highest average rate occurs on the **SB** litmus test on AMD GPUs. Most of the AMD devices show a high rate of weak behaviors on **SB**, approaching 10% and higher, but devices with AMD's Graphics Core Next 5 micro-architecture all showed rates under 1%. This means that even from a single vendor, the behaviors of different architectures can vary widely and past results from one vendor cannot be counted on to predict future behaviors.

4.2 Weak Behaviors in Vulkan

The data in Tab. 2 shows the percentage of weak behaviors in the test environment that maximizes the rate at which they occur for our Android devices. In contrast to our web GPUs, in the mobile setting, weak behaviors were observed in every test on only one device, the NVIDIA Tegra X1, but the rates on this device were very low, beneath 0.1%. The most difficult test to observe in general was **R**, which checks whether a store is reordered with a following load on one thread. We did not observe any weak behaviors on the Imagination GPU; because testing is fundamentally incomplete, this could mean that the device implements a strong MCS, or that our testing approach was not effective. Interestingly, ARM only showed weak behaviors in the **MP** test.

Table 2: Data showing the average rate of weak behaviors across vendors when running litmus tests using Vulkan.

		Litmus Test					
Vendor	**Device**	**MP**	**LB**	**SB**	**S**	**R**	**2+2W**
Qualcomm	Adreno 610	0%	0%	0%	0%	0%	0%
	Adreno 640	0%	2.04%	1.45%	1.65%	0%	2.21%
	Adreno 642L	0.04%	5.75%	5.81%	3.81%	0%	6.38%
	Adreno 660	0.12%	8.5%	14.37%	5.69%	0%	11.5%
Arm	Mali-G71	0.04%	0%	0%	0%	0%	0%
	Mali-G78	1.56%	0%	0%	0%	0%	0%
Imagination	PowerVR GE8320	0%	0%	0%	0%	0%	0%
NVIDIA	Tegra X1	0.01%	0.05%	0.02%	0.01%	0.01%	0.05%

We observe that, in general, the rates of weak behaviors increase as devices become more powerful. This is especially apparent from the four Qualcomm devices we test, as the rate of weak behaviors increases from 0% on the Adreno 610 (which has 96 *shading units*, analogous to NVIDIA's CUDA cores) up to a maximum of 14.37% in **SB** on the Adreno 660 (with 512 shading units). One intuitive explanation for this might be that smaller GPUs lack the ability to schedule as many threads at once, naturally reducing the rates of weak behaviors despite architectures that might allow them. We see a similar trend on the Arm GPUs, where the smaller Mali-G71 (32 shading units) showed a lower rate of weak behaviors than the larger Mali-G78 (384 shading units).

5 INSIGHTS AND IMPACTS

We now set out to answer the three questions posed in Sec. 4 using our data and characterization of weak behavior rates.

5.1 MCS Bugs

Our conformance testing campaigns discovered bugs on several vendors' devices when running under the Vulkan and WebGPU frameworks.

(1) **Arm**: We observed coherency violations of the **MP-CO** litmus test when using the Vulkan framework on two Arm GPUs, a Mali-G71 and a Mali-G78. These bugs were reported to and confirmed by Arm, leading to a compiler fix to insert a missing memory fence. Arm has also added regression tests based on the pattern of the violation we reported.
(2) **NVIDIA**: We also observed violations of the **MP-CO** test when run using the Vulkan framework on an NVIDIA Tegra X1. Additionally, our WebGPU conformance test results revealed violations of a different coherence test, **RR**, on an NVIDIA Quadro P620 running on a Linux desktop (therefore using Vulkan as the native framework). The combined

Table 3: Each row shows the cosine similarity statistics between all pairs of devices from that vendor. The last row shows the similarity statistics across all pairs of devices.

Vendor	Avg	Median	Min	Max
Intel	0.870	0.891	0.683	0.985
Apple	0.903	0.913	0.699	0.993
NVIDIA	0.903	0.931	0.670	0.996
AMD	0.904	0.927	0.661	0.989
All	0.840	0.862	0.477	0.996

Table 4: Device clustering shows that choosing one device from each vendor is not an optimal way to test applications that utilize the MCS.

	Cluster					
Device	**A**	**B**	**C**	**D**	**E**	**F**
Intel	3	0	0	5	18	0
Apple	12	4	0	10	0	0
NVIDIA	1	0	19	8	2	1
AMD	13	1	0	1	0	0

report of the bug on the Tegra X1 and Quadro P620 helped NVIDIA find and fix a bug in their Vulkan compiler. NVIDIA also noted that this bug affected the Vulkan compiler in all pre-Volta architecture GPUs.

(3) **Apple**: Our WebGPU conformance tests reveal coherence violations in **RR** from eight other devices, all running on Apple machines. Five of these devices were Intel GPUs, two were NVIDIA GPUs, and one was an AMD GPU. MCMutants observed the same issue on an Intel integrated device on a MacBook and reported the issue to Apple [28]; the bug has not been confirmed nor fixed. On Intel, it is likely that these bugs are instances of the same issue, but our results are the first time the bug has been observed on non-Intel GPUs.

We found all of the bugs by running conformance litmus tests under tuned system stress. We choose the system stress using the methodology described in prior work [28], i.e., by selecting a system stress configuration effective at revealing weak behaviors in an associated weak memory litmus test. For the **MP-CO** bugs on the Arm and NVIDIA devices, we perform a correlation analysis to empirically validate the methodology.

We run both the **MP** and **MP-CO** litmus tests in 150 randomly generated system stress configurations on each of the Android devices showing the bug, recording the rate of weak behaviors in **MP** and the rate of buggy (i.e. non-coherent) behaviors in **MP-CO**. Our results show that the Pearson Correlation Coefficient (PCC) between the rate of weak and buggy behaviors is 0.732 on the Arm Mali-G71, 0.759 on the Arm Mali-G78, and 0.832 on the NVIDIA Tegra X1. Since these behaviors are recorded from 150 samples (i.e. system stress configurations), we have 148 degrees of freedom, and running a Student's t-test leads to a p-value less than $10^{-5}\%$ on each device. This shows that the PCC between weak behaviors and bugs is certainly not due to random chance, further validating that configurations tuned using weak behaviors are effective at revealing bugs in conformance tests.

5.2 GPU Similarity

All of the data was collected by running the tests with pseudo-randomly generated system stress configurations, but as mentioned in Sec. 3.4 the generator is seeded with a known value. Thus, we can compare how different GPUs behave under the same stress parameters. To do this, each testing run is represented as a vector of the non-sequential (i.e. where one thread runs entirely before another one) behaviors of every test in each configuration. Ignoring the sequential behaviors gives the data a degree of freedom, which is necessary for calculating a valid similarity measure.

For our similarity metric, we choose *cosine similarity*, which measures the cosine of the angle between two vectors and ranges from -1 to 1. We chose cosine similarity because it is a relative metric, not an absolute like Euclidean distance, meaning that devices that show different absolute rates of behaviors, but at similar relativity, are classified as more closely related.

Device Identification. Table 3 shows a summary of the similarity between devices in our study. All similarities are positive, with the minimum being 0.477 between an Intel and AMD device. This is not surprising, since effective system stress is likely to reveal weak behaviors on many devices. However, the average and median similarities between devices from each vendor are higher than the overall average and median, showing that in general devices from the same vendor tend to have more similar MCS behaviors.

For Apple and NVIDIA, we confirmed that the maximum similarity occurs between identical GPUs: two Apple M1 Max's and two NVIDIA GeForce RTX 3080s. For AMD, we observe a maximum similarity of 0.989 between two devices, one of which is a Radeon Pro 5500M (**A**) while the other device (**B**) did not report a model and instead only indicated that it was from the same architectural generation as **A**. However, we observed a high similarity (0.985) between **A** and another Radeon Pro 5500M (**C**), as well as a similarity of 0.984 between **B** and **C**, so it seems likely that **A**, **B**, and **C** are all the same device. We do a similar analysis with Intel to determine that an unknown device is most likely an Intel Iris Xe Graphics. While we are most interested in using this data to help choose conformance test strategies, as shown next, we also note that GPU MCS behavior data like this exposes a fingerprinting vulnerability, despite the specification trying to hide specific device information for security reasons.

Clustering Based Testing Strategies. K-means clustering attempts to minimize the *distortion*, or the sum of squared distances between vectors and a centroid. Applying k-means clustering to GPU MCS behavior has implications for testing strategies; when developing cross-platform GPU applications that rely on shared memory operations, testing these applications on a number of devices can increase confidence in the correctness of the implementation. A naive strategy might be to choose one device from each major vendor, but our results show that this is not necessarily an optimal strategy.

Table 4 shows the result of running k-means with six clusters on the similarity data from Tab. 3. The "elbow-method" heuristic

Table 5: Analysis of three locking algorithms, Test-and-Set (TAS), Test and Test-and-Set (TTAS), and Compare-and-Swap (CAS), showing in how many test runs (out of 1000) we observed failures of unfenced (UF) and fenced (F) lock implementations to protect a critical section. The total time to run all tests on each device is also recorded.

		TAS		TTAS		CAS	
Device	**Time (min)**	**UF**	**F**	**UF**	**F**	**UF**	**F**
Adreno 610	3.5	0	0	0	0	0	0
Mali-G78	67.9	18	0	11	0	7	0

showed that the rate of decrease in distortion leveled off at 6 clusters on our data. The clustering data shows that devices from the same vendor are generally placed into the same cluster, but there are outliers in each case. The only NVIDIA Kepler device in our study was dissimilar enough from other devices that it was placed in its own cluster. Kepler is also the oldest NVIDIA architecture in our study, showing that special testing attention might be needed when supporting older devices in cross-platform GPU frameworks.

When selecting which devices to test a shared memory GPU application on, the strategy should be to first choose a number of clusters based on the rate of decrease in distortion, and then select at least one device from each cluster. Using our data, this might mean selecting an AMD device from **A**, Apple devices from clusters **B** and **D**, NVIDIA devices from clusters **C** and **F** (the only device in that cluster), and an Intel device from cluster **E**. Despite choosing more Apple and NVIDIA devices than AMD and Intel, the similarity data ensures the tests maximally cover devices with different behavior profiles.

5.3 Implementing Synchronization Algorithms

We now discuss a use case of how the diversity of weak memory profiles across these different GPUs can impact software development. Locking algorithms are implemented using atomic operations to synchronize access to critical sections. Implementation of locks depends on careful placement of memory fences to avoid compilers and hardware from reordering memory accesses, which can cause critical section failures. In this section, we implemented three common spin-locks: test-and-set (TAS), test-and-test-and-set (TTAS), and compare-and-swap (CAS). Each of these locks specifically needs to disallow **MP** behaviors using acquire/release memory fences. However, our results in Sec. 4 show that on some mobile devices **MP** weak behaviors never occur, meaning that if the locks are tested on these devices, they may run correctly despite being incorrectly implemented (according to the specification).

To investigate this, we tested our three locks on two Android devices, an Arm Mali-G78 and a Qualcomm Adreno 610. The locks were implemented both with and without appropriate acquire/release memory fences. In these tests, threads from different workgroups acquire the lock 10k times and increment a non-atomic memory location in the critical section. We ran this test for 1k iterations and recorded the number of critical section violations we observed for each device and each lock.

On the Arm Mali-G78, a larger GPU which exhibits a relatively high rate of **MP** behaviors, we observed critical section failures in unfenced versions of all three locks; in every failure case except one the value was 189,999 instead of 190,000, meaning that just one of the increments was not reflected. In the remaining failure case, the value was 189,998. On the Qualcomm Adreno 610, which exhibited no **MP** behaviors in our study, we saw no failures. Both devices exhibited no failures when locks were run with correct fences.

Therefore, when writing applications that require synchronization, care must be taken to ensure the application is tested on devices where incorrect implementations will lead to failures, highlighting the importance of collecting and characterizing MCS behavior data.

6 LESSONS LEARNED

In this section we discuss important lessons learned while developing and running our study.

Ease of Use. Technical studies of low-level details like memory consistency specifications [3, 25, 46] have been run by expert practitioners and involve installing special software (e.g. OpenCL/CUDA drivers) and running experiments from command line interfaces. However, experiments that solicit non-technical users require accessible and frictionless interfaces in order to collect many results. For example, we initially had users download their results and email them to us directly, but found that many users would not take this seemingly small step. Thus, we implemented a way to submit results by simply clicking a button. This required substantial engineering effort, both to set up a client/server infrastructure and to distribute the tools in a non-technical way (e.g. through web browsers/app stores). Once implemented this workflow also had the benefit of making our experiments standardized; instead of relying on users to configure their system and choose the right options, all of this was baked in so that users only had to click a few buttons to run and submit results.

Testing Time. Previous studies ran tests for hours or days, but it is unrealistic for volunteer users to run experiments that long on their devices. Therefore, we explored the trade-off space between experiment time and behavior coverage. Through trial and error, we determined parameters that allowed us to collect high-quality, standardized data in a short time frame, utilizing testing techniques from prior work that increased testing speed and provided statistical measures of reproducibility [28].

Enabling New Research Questions. Important research questions on memory consistency, including the three from Sec. 4, require performing a large-scale study. For example, previous studies [25] have attempted to create portable testing strategies, but could only provide limited guidance on choosing representative sets of devices to test on due to the small number of devices in their evaluation. On the other hand, our data shows that GPUs from different vendors can behave similarly under stress, and thus portability may not be vendor-specific. Therefore, increasing the scale of evaluation through faster and more accessible testing should be an important factor when developing new testing strategies for a diverse (and ever growing) set of devices.

Extensibility. When we first designed our LitGen tool, it only generated SPIR-V shaders. However, as we started focusing on testing WebGPU's MCS, LitGen's neutral configuration language (JSON) allowed us to easily write a backend generator for WGSL shaders. Ensuring our tools are extensible means that they might also be useful for researchers testing other areas of GPU specifications, e.g., floating point operation accuracy. In the same vein, our initial app only targets Android devices running Vulkan, but as we seek to expand the scope of our testing, we plan on developing an app that will work on both Android and iOS devices.

7 FUTURE WORK

This work has spent significant engineering effort enabling the testing of many different GPUs. However, given the difficulty of cross-platform GPU programming, we were still unable to test mobile Apple GPUs, which appear in some of the most widely used mobile devices. Additionally, our web interface and Android app contain distinct user interfaces and GPU setup code, causing duplicate efforts and maintenance. In this section, we outline a path forward, with Flutter as a fitting match for these goals.

Flutter [17] is an open-source software development kit developed by Google that provides deployment options to desktop platforms (such as Windows, macOS, and Linux), mobile platforms (Android, iOS), and even web deployment from a single frontend codebase. With a unified codebase for the MCS testing front end, development work can be focused on designing backend implementations specific to those platforms. Underlying Flutter is Dart [16], a language also developed by Google for cross-platform app development. For each supported platform, Flutter provides an interface to backend code native to the specific platform. On the Android end, GPU access is provided through Dart's foreign function interface (FFI) library to load a dynamically linked C library, compiled against the version of Vulkan provided by Android's Native Development Kit (NDK) [14]. The Dart FFI library can be used similarly on all supported platforms except for the web, for which GPU access will involve calls to JavaScript code utilizing WebGPU.

Vulkan, while well-supported on Windows, Linux, and Android devices, is not officially supported by macOS and iOS clients. For these platforms, there are two possible options. For a more native-friendly option, Vulkan backend code could be instead rewritten to depend on Apple's Metal [6] API, with SPIR-V shaders transpiled to the Metal Shading Language (MSL) using SPIRV-Cross [24], a tool developed by Khronos Group. However, to reduce development time and duplicate code across multiple platforms, Vulkan backend code can be passed through MoltenVK [23], a Khronos Group implementation of a large subset of Vulkan 1.2 on top of Metal. This provides a portability layer with which to run Vulkan applications on iOS and macOS platforms.

We also plan on integrating our new tools with the current server backend, allowing us to collect data from devices we do not have physical access to using a simple API interface. With a single source for interface design, GPU setup, and data collection, it is expected that future work will be able to deploy MCS testing at a wider scale and collect results from GPU hardware previously inaccessible in related work.

8 RELATED WORK

Testing MCSs. Work on testing MCS dates back to tools like ARCHTEST [49] and TSOTool [18], which each generated test programs containing sequences of loads and stores and then looked for violations of sequential consistency. With the introduction of formal MCSs, researchers developed tools like litmus [3], which runs litmus tests generated from formal models directly on ISAs (namely x86, Power, and Arm) and includes stress parameters that make weak behaviors more likely.

Techniques for CPU MCS testing have been extended to GPUs [1, 25]. Weak behaviors on GPUs are notoriously difficult to reveal, leading to work that statistically analyzed tuning techniques and reproducibility of results when running litmus tests on GPUs [25]. To better evaluate the efficacy of test environments and provide confidence in MCS implementations, [28] introduced a methodology based on black-box mutation testing [8], finding bugs in several WebGPU MCS implementations.

Previous studies have been limited in the number of devices they were able to test. In contrast, this study introduces tooling that allows us to conduct the largest ever GPU MCS testing campaign, running tests across 2 frameworks, 7 vendors, and 106 devices.

Testing at Scale. Other studies have tested large numbers of devices, searching for bugs in compilers and hardware. In [11], 17 GPU and driver combinations were tested for compiler bugs. Our approach, distributing the GPU MCS testing experiment using a web interface, is a form of *volunteer computing*, where the general public volunteers their computing resources for research studies. Volunteer computing has been used for many compute-intensive tasks, including searching for extraterrestrial life [5], training neural networks [10], sequencing genomes [40], and climate modeling [9].

9 CONCLUSION

We introduce GPUHarbor, a tool suite with a web interface and Android app for accessible cross-platform GPU MCS testing. We utilize GPUHarbor to perform a large-scale study on weak behaviors in 106 GPUs from seven vendors and find two bugs in GPUs running on mobile devices. Our results show the importance of scaling previous MCS testing strategies in order to characterize the behavior of different devices, perform conformance testing, and design application testing strategies.

ACKNOWLEDGMENTS

We thank the reviewers whose feedback helped strengthen the paper and motivated the lessons learned section. We thank Jan-Harald Frederiksen from Arm for working with us to confirm the bug on Arm's devices, and Jeff Bolz from NVIDIA for finding and confirming the bug in NVIDIA's compiler. We thank David Neto and Alan Baker from Google for feedback on the description of the WebGPU memory model and our results analysis. We thank everyone who submitted anonymous data for this study, including friends and family. This work was supported by a gift from Google.

REFERENCES

[1] 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 *International*

Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (ASPLOS '15). Association for Computing Machinery, 577–591. https://doi.org/10.1145/2694344.2694391

[2] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2010. Fences in weak memory models. In *Computer Aided Verification*, Tayssir Touili, Byron Cook, and Paul Jackson (Eds.). Springer Berlin Heidelberg, 258–272.

[3] 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)*, Vol. 6605. 41–44. https://doi.org/10.1007/978-3-642-19835-9_5

[4] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *Trans. Program. Lang. Syst. (TOPLAS)* 36, 2, Article 7 (July 2014), 74 pages. https://doi.org/10.1145/2627752

[5] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. 2002. SETI@home: An experiment in public-resource computing. *Commun. ACM* 45, 11 (nov 2002), 56–61. https://doi.org/10.1145/581571.581573

[6] Apple. 2023. Metal. https://developer.apple.com/documentation/metal/. Retrieved February 2023.

[7] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *Symposium on Principles of Programming Languages (POPL) (POPL '11)*. Association for Computing Machinery, 55–66. https://doi.org/10.1145/1926385.1926394

[8] Timothy A. Budd and Ajei S. Gopal. 1985. Program testing by specification mutation. *Computer Languages* 10, 1 (1985), 63–73. https://doi.org/10.1016/0096-0551(85)90011-6

[9] C. Christensen, T. Aina, and D. Stainforth. 2005. The challenge of volunteer computing with lengthy climate model simulations. In *First International Conference on e-Science and Grid Computing (e-Science'05)*. 8 pp.–15. https://doi.org/10.1109/E-SCIENCE.2005.76

[10] Travis Desell. 2017. Large scale evolution of convolutional neural networks using volunteer computing. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '17)*. Association for Computing Machinery, 127–128. https://doi.org/10.1145/3067695.3076002

[11] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated testing of graphics shader compilers. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 93 (oct 2017), 29 pages. https://doi.org/10.1145/3133917

[12] Wu-chun Feng and Shucai Xiao. 2010. To GPU synchronize or not GPU synchronize?. In *2010 IEEE International Symposium on Circuits and Systems (ISCAS)*. 3801–3804. https://doi.org/10.1109/ISCAS.2010.5537722

[13] Esther Francis. 2014. Autonomous cars: no longer just science fiction. (2014).

[14] Google. 2023. Android NDK. https://developer.android.com/ndk.

[15] Google. 2023. Clspv. https://github.com/google/clspv.

[16] Google. 2023. Dart. https://dart.dev/.

[17] Google. 2023. Flutter. https://flutter.dev/.

[18] S. Hangal, D. Vahia, C. Manovit, J.-Y.J. Lu, and S. Narayanan. 2004. TSOtool: A program for verifying memory systems using the memory consistency model. In *International Symposium on Computer Architecture (ISCA), 2004.* 114–123. https://doi.org/10.1109/ISCA.2004.1310768

[19] Jeff Bolz. 2022. Vulkan memory model. https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html#memory-model.

[20] Khronos Group. 2021. SPIR-V specification version 1.6, revision 1. https://www.khronos.org/registry/SPIR-V/specs/unified1/SPIRV.html.

[21] Khronos Group. 2022. The OpenCL C Specification. https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_C.html.

[22] Khronos Group. 2022. Vulkan 1.3 Core API.

[23] Khronos Group. 2023. MoltenVK. https://github.com/KhronosGroup/MoltenVK.

[24] Khronos Group. 2023. SPIRV-Cross. https://github.com/KhronosGroup/SPIRV-Cross.

[25] Jake Kirkham, Tyler Sorensen, Esin Tureci, and Margaret Martonosi. 2020. Foundations of empirical memory consistency testing. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 226 (Nov. 2020), 29 pages. https://doi.org/10.1145/3428294

[26] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

[27] Reese Levine, Mingun Cho, Devon McKee, Andrew Quinn, and Tyler Sorensen. 2023. *GPUHarbor: Testing GPU Memory Consistency At Large (Experience Paper): Artifact.* https://doi.org/10.5281/zenodo.7922486

[28] Reese Levine, Tianhao Guo, Mingun Cho, Alan Baker, Raph Levien, David Neto, Andrew Quinn, and Tyler Sorensen. 2023. MC mutants: Evaluating and improving testing for memory consistency specifications. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS 2023)*. Association for Computing Machinery, 473–488. https://doi.org/10.1145/3575693.3575750

[29] Sela Mador-Haim, Rajeev Alur, and Milo M K. Martin. 2010. Generating litmus tests for contrasting memory consistency models. In *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV'10)*. Springer-Verlag, 273–287. https://doi.org/10.1007/978-3-642-14295-6_26

[30] Yatin A. Manerkar, Daniel Lustig, Margaret Martonosi, and Michael Pellauer. 2017. RTLcheck: Verifying the memory consistency of RTL designs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17)*. Association for Computing Machinery, 463–476. https://doi.org/10.1145/3123939.3124536

[31] 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.

[32] Duane Merrill and Michael Garland. 2016. Single-pass parallel prefix scan with decoupled lookback. https://research.nvidia.com/publication/2016-03_single-pass-parallel-prefix-scan-decoupled-look-back

[33] Microsoft. 2020. Programming guide for Direct3D 11. https://docs.microsoft.com/en-us/windows/win32/direct3d11/dx-graphics-overviews.

[34] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, David A. Wood, and Natalie Enright Jerger. 2020. *A primer on memory consistency and cache coherence* (2nd ed.). Morgan & Claypool Publishers.

[35] Feng Niu, Benjamin Recht, Christopher Re, and Stephen J. Wright. 2011. HOGWILD! A lock-free approach to parallelizing stochastic gradient descent. In *Proceedings of the 24th International Conference on Neural Information Processing Systems (NIPS'11)*. Curran Associates Inc., 693–701.

[36] NVIDIA. 2023. CUDA C++ programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[37] Özgün Özerk, Can Elgezen, Ahmet Can Mert, Erdinç Öztürk, and Erkay Savaş. 2022. Efficient number theoretic transform implementation on GPU for homomorphic encryption. *J. Supercomput.* 78, 2 (feb 2022), 2840–2872. https://doi.org/10.1007/s11227-021-03980-5

[38] Mohit Pandey, Michael Fernandez, Francesco Gentile, Olexandr Isayev, Alexander Tropsha, Abraham C Stern, and Artem Cherkasov. 2022. The transformational role of GPU computing and deep learning in drug discovery. *Nature Machine Intelligence* 4, 3 (2022), 211–221.

[39] S. K. Park and K. W. Miller. 1988. Random number generators: Good ones are hard to find. *Commun. ACM* 31, 10 (oct 1988), 1192–1201. https://doi.org/10.1145/63039.63042

[40] S. Pellicer, N. Ahmed, Yi Pan, and Yao Zheng. 2005. Gene sequence alignment on a public computing platform. In *2005 International Conference on Parallel Processing Workshops (ICPPW'05)*. 95–102. https://doi.org/10.1109/ICPPW.2005.35

[41] Lakshminarayanan Renganarayana, Vijayalakshmi Srinivasan, Ravi Nair, and Daniel Prener. 2012. Programming with relaxed synchronization. In *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability (RACES '12)*. Association for Computing Machinery, 41–50. https://doi.org/10.1145/2414729.2414737

[42] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. 2019. Survey and benchmarking of machine learning accelerators. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–9. https://doi.org/10.1109/HPEC.2019.8916327

[43] Mehrzad Samadi, Janghaeng Lee, D. Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. 2013. SAGE: Self-tuning approximation for graphics engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. Association for Computing Machinery, 13–24. https://doi.org/10.1145/2540708.2540711

[44] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. 2009. The semantics of x86-CC multiprocessor machine code. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*. Association for Computing Machinery, 379–391. https://doi.org/10.1145/1480881.1480929

[45] Dennis Shasha and Marc Snir. 1988. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.* 10, 2 (April 1988), 282–312. https://doi.org/10.1145/42190.42277

[46] Tyler Sorensen and Alastair F. Donaldson. 2016. Exposing errors related to weak memory in GPU applications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. Association for Computing Machinery, 100–113. https://doi.org/10.1145/2908080.2908114

[47] Tyler Sorensen and Alastair F. Donaldson. 2016. The hitchhiker's guide to cross-platform OpenCL application development. In *Proceedings of the 4th International Workshop on OpenCL (IWOCL '16)*. Association for Computing Machinery, Article 2, 12 pages. https://doi.org/10.1145/2909437.2909440

[48] John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. 2017. Automatically comparing memory consistency models. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '17)*. Association for Computing Machinery, 190–204. https://doi.org/10.1145/3009837.3009838

[49] William W. Collier. 1994. ARCHTEST. http://www.mpdiag.com/archtest.html.

[50] World Wide Web Consortium (W3C). 2022. WebGPU shading language: Editor's draft. https://gpuweb.github.io/gpuweb/wgsl/.

[51] World Wide Web Consortium (W3C). 2023. WebGPU: W3C working draft. https://www.w3.org/TR/webgpu/.

[52] World Wide Web Consortium (W3C). 2023. WebGPU: W3C working draft: Privacy considerations. https://www.w3.org/TR/webgpu/#privacy-considerations.

Received 2023-02-16; accepted 2023-05-03