Towards Unified Analysis of GPU Consistency

Haining Tong
University of Helsinki
Finland
haining.tong@helsinki.fi

Natalia Gavrilenko
Huawei Dresden Research Center
Germany
natalia.gavrilenko@huawei.com

Hernán Ponce de León
Huawei Dresden Research Center
Germany
hernanl.leon@huawei.com

Keijo Heljanko
University of Helsinki
Helsinki Institute for Information Technology
Finland
keijo.heljanko@helsinki.fi

ABSTRACT

After more than 30 years of research, there is a solid understanding of the consistency guarantees given by CPU systems. Unfortunately, the same is not yet true for GPUs. The growing popularity of general purpose GPU programming has been a call for action which industry players like Nvidia and Khronos have answered by formalizing their Ptx and Vulkan consistency models. These models give precise answers to questions about program's correctness. However, interpreting them still requires a level of expertise that escapes most developers, and the current tool support is insufficient.

To remedy this, we translated and integrated the Ptx and Vulkan models into the Dartagnan verification tool. This makes Dartagnan the first analysis tool for multiple GPU consistency models that can analyze real GPU code. During the validation of the translated models, we discovered two bugs in the original Ptx and Vulkan consistency models.

ACM Reference Format:
Haining Tong, Natalia Gavrilenko, Hernán Ponce de León, and Keijo Heljanko. 2024. Towards Unified Analysis of GPU Consistency. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3622781.3674174

1 INTRODUCTION

Since the early 2000s, GPUs have become popular for accelerating compute-intensive applications due to the massive parallelism they provide. To achieve high performance, hardware architects have opted for weak consistency models [14, 33, 37, 48, 49]. However, only recently, significant progress has been made in formalizing consistency guarantees of GPUs [12, 47, 48], other accelerators [40], and heterogeneous computing systems [35].

While there are frameworks to assist in verifying and optimizing concurrent libraries for CPUs [53, 63, 64], we lack similar tools for the GPU models. There are prototypes, built on top of the Alloy tool [41], allowing to reason about GPU consistency models [3, 12]. However, these tools are limited to a single model, they do not support real GPU programming APIs (just straight line pseudo-assembly with no control flow instructions), they are not able to reason about liveness properties, and cannot be applied to programs with more than a few instructions due to limited scalability.

```
1  __kernel void xf_barrier(global atomic_uint *flag, global uint* in, global uint* out) {
2
3          in[get_global_id(0)] = 1;
4
5          if (get_group_id(0) == 0) {
6              if (get_local_id(0) + 1 < get_num_groups(0)) {
7                  while (atomic_load(&flag[get_local_id(0) + 1]) == 0);
8              }
9              barrier(CLK_GLOBAL_MEM_FENCE);
10             if (get_local_id(0) + 1 < get_num_groups(0)) {
11                 atomic_store(&flag[get_local_id(0) + 1], 0);
12             }
13         } else {
14             barrier(CLK_GLOBAL_MEM_FENCE);
15             if (get_local_id(0) == 0) {
16                 atomic_store(&flag[get_group_id(0)], 1);
17                 while (atomic_load(&flag[get_group_id(0)]) == 1);
18             }
19             barrier(CLK_GLOBAL_MEM_FENCE);
20         }
21
22         for (unsigned int i = 0; i < get_global_size(0); i++) {
23             out[get_global_id(0)] += in[i];
24         }
25
26         // assert(out[get_global_id(0)] == get_global_size(0));
27 }
```

Figure 1: A kernel using the portable version [60] of the XF-barrier [66] to synchronize different workgroups.

Consider the XF inter-workgroup barrier in Figure 1, written in OpenCL and adapted from [60]. The barrier should prevent data races and guarantee that each thread observes (line 23) correct values written (line 3) by all other threads. Threads in workgroup zero act as leaders for the other workgroups. The remaining workgroups contain followers. In each follower workgroup, the thread with local id zero is a representative of the workgroup.

The barrier works as follows. First, each follower thread synchronizes (line 14) with all other followers from the same workgroup. After that, the representative thread sets a flag (line 16) indicating that all workgroup threads have arrived to the barrier. Then, the representative thread spins (line 17) waiting for a flag from its leader. The other followers wait for their representative on a control barrier (line 19).

In the leaders workgroup, each thread spins (line 7) waiting for a flag from the representative of its followers. After receiving the flag, the thread waits for the other leaders on the control barrier (line 9). When all leaders have synchronized, each leader sets a

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0391-1/24/04.
https://doi.org/10.1145/3622781.3674174

flag (on line 11) signaling to its followers that they can proceed. This allows the representative to exit the spinloop and execute the control barrier (on line 19), thus unblocking all other followers.

The original implementation of the algorithm [66] uses plain accesses instead of the atomic ones shown in Figure 1. As noted in [60], this makes the code not portable across different GPUs: the code can hang and contains data races. It was empirically shown in [60] that these problems disappear if release-acquire semantics are used. However, previous to our work, no automatic tool could verify this code under weak consistency. In Section 6 we formally and automatically prove, for the first time, that the code using atomics is correct and that the release-acquire semantics cannot be relaxed without introducing bugs.

In this work, we focus on the consistency models of two popular GPU programming APIs: Nvidia Ptx [47, 48] (the low level ISA used by Cuda) and Khronos Vulkan [12]. We formalize both models in the `.cat` domain specific language [13, 15, 21], extending it with previously unsupported GPU specific features. We integrate these models into Dartagnan, a scalable tool for weak consistency models. Dartagnan helps developers of concurrency primitives and compiler engineers to analyze and optimize their code under different consistency models. It also helps porting concurrency primitives across different GPU architectures, and to uncover bugs in existing algorithm implementations.

Concretely, we make the following contributions:

- Enhance `.cat` with new GPU specific features.
- Provide formal models, written in `.cat`, for Ptx (versions 6.0 and 7.5) and Vulkan.
- Extend the Dartagnan verification tool[1] with support for GPU consistency.
- Add to Dartagnan three new front-ends: two for the pseudo-assembly-like syntax used in our examples (one for Ptx and one for Vulkan), and one for a subset of real Spir-V assembly [9].
- Discuss similarities and differences between Ptx and Vulkan consistency, and how they affect portability of GPU algorithms.

2 BACKGROUND

This section introduces the language we use to formalize consistency models and our approach to automatically analyse programs with respect to these models.

2.1 Consistency Models

We consider consistency models written in the `.cat` language [13, 15, 21], whose core part is given in Figure 2. A consistency model is defined in `.cat` via memory event tags (s), relations over memory events (r), and axioms (emptiness, irreflexivity, and acyclicity) over those relations (axm).

Base event tags (t) are derived from the semantics of program instructions, e.g., writes $\mathbb{W}$, reads $\mathbb{R}$, fences $\mathbb{F}$, control barriers $\mathbb{B}$, and writes populating initial memory values $\mathbb{I}$. Other base event tags can be used to enforce special semantics, such as the memory order of atomic accesses. New event tags (s) can be derived using union ($s \cup s$), intersection ($s \cap s$), and set difference ($s \setminus s$).

[1] https://github.com/hernanponcedeleon/Dat3M

$$
\begin{array}{rcl}
mm & ::= & def \mid axm \mid mm \wedge mm \\
axm & ::= & \textbf{acyclic}(r) \mid \textbf{irreflexive}(r) \mid \textbf{empty}(r) \\
def & ::= & \texttt{let}\ d := r \mid \texttt{let}\ d := s \\
r & ::= & b \mid d \mid r^{-1} \mid r;r \mid r \cap r \mid r \cup r \mid r \setminus r \\
b & ::= & \mathsf{po} \mid \mathsf{rf} \mid \mathsf{co} \mid \mathsf{loc} \mid [t] \mid t \times t \mid \ldots \\
s & ::= & t \mid d \mid s \cap s \mid s \cup s \mid s \setminus s \\
t & ::= & \mathbb{I} \mid \mathbb{W} \mid \mathbb{R} \mid \mathbb{F} \mid \mathbb{B} \mid \ldots
\end{array}
$$

Figure 2: The `.cat` language for consistency models [15].

Base relations (b) describe the static semantics of the program and its dynamic behavior. Program order po is the order of events in a thread. Read-from rf relates a write event with a read event that loads the value stored by the first. Coherence order co represents the order of writes to a memory location. Relation loc matches all accesses to the same shared memory location. New relations (r) can be derived using union ($r \cup r$), intersection ($r \cap r$), difference ($r \setminus r$), relation composition ($r;r$), and relation inverse (r^{-1}). Derived sets and relations can be given a name using the `let` keyword.

2.2 Programs and Behaviours

To have a common program syntax for different architectures, throughout the rest of the paper, we give code snippets in the form of litmus tests. Figure 3 shows a simplified version of the original XF-barrier (which contrary to Figure 1, uses non-atomic accesses). Columns in the litmus syntax represent threads (e.g., `P0`). The first line specifies where in the GPUs hierarchy the thread lives, e.g., which workgroup the thread belongs to (see Section 3.1 for more details). The remaining lines are the sequence of thread instructions, e.g., reading from (`ld`) and writing to (`st`) memory, labels (`LC00`), conditional (`bne`) and unconditional (`goto`) jumps, or control barriers (`cbar`). A litmus test ends with an `exists` or `forall` safety condition.

A weak behavior of a concurrent program is defined by the values that threads write to and read from the shared memory. Formally, a behavior is a tuple $(\mathbb{X}, \mathsf{rf}, \mathsf{co})$ of executed events, the read-from relation, and the coherence order relation. Behaviors must be well-defined: *(i)* events should represent valid control flows, *(ii)* every read must obtain its value from some write, *(iii)* rf and co can only relate events accessing the same location, *(iv)* writes in $\mathbb{I}$ come first in co order, and *(v)* when projected to a single location, co is a total order (with the exception of Ptx, see Section 4.1). A behavior is allowed by a consistency model if it satisfies all axioms of the model, otherwise it is forbidden. The set of allowed behaviors defines the semantics of the program with respect to the consistency model.

A behavior can be visualised as an execution graph. The nodes (called events) are derived from executed instructions (memory accesses, fences and barriers) and the edges are relations over those events. Events are annotated with the thread identifier, the corresponding line of code, the instruction type, and additional data. Additional data can contain the variable name, the stored or observed value, or the barrier identifier. Figure 3 shows an execution graph where $e_4^{\mathtt{P0}}$ reads from the initial write e_{init}. Write $e_3^{\mathtt{P1}}$ overrides the initial value of `f`, which is represented by the co edge.

```
P0@sg 0,wg 0,qf 0   | P1@sg 0,wg 1,qf 0   | P2@sg 0,wg 1,qf 0 ;
st.av.dv.sc0 x, 1   | cbar.wg 1           | cbar.wg 1         ;
LC00:               | st.dv.sc0 f, 1      | cbar.wg 2         ;
ld.dv.sc0 r2, f     | LC10:               |                   ;
bne r2, 0, LC01     | ld.dv.sc0 r2, f     |                   ;
goto LC00           | bne r2, 1, LC11     |                   ;
LC01:               | goto LC10           |                   ;
cbar.wg 1           | LC11:               |                   ;
st.dv.sc0 f, 0      | cbar.wg 2           |                   ;
                    | ld.vis.dv.sc0 r1, x |                   ;
exists
(P1:r2 != 1 /\ P1:r1 == 0)
```

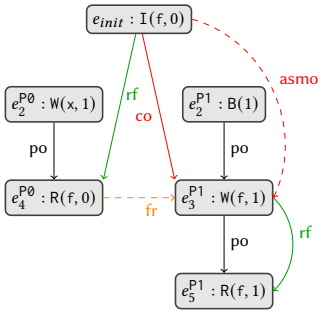

Figure 3: A data race in the original XF-barrier from [66].

The po edges show program order of each thread. This behaviour is consistent according to the Vulkan model in Figure 8, but it contains a data race.

2.3 Bounded Model Checking

It is possible to encode the semantics of a program w.r.t. a `.cat` model as a SAT modulo theories (SMT) formula. Then, Bounded Model Checking (BMC) can be used to find violations of safety, liveness, or data race freedom (DRF) [18, 29]. In its most naive form, the encoding uses one Boolean variable for each combination of a relation and an event pair. If the variable is set, then the pair belongs to the relation. The encoding of base relations must ensure that behaviors are well-defined. For most derived relations, the encoding is straightforward, e.g., union between relations corresponds to a disjunction between variables. Recursively defined relations require to compute fix-points and are the hardest to handle. There are several tricks (which the Dartagnan tool uses) to alleviate the problem [29, 30].

This naive encoding generates a large formula and verification does not scale well. For a more compact encoding, Dartagnan employs static analysis techniques to compute lower and upper bounds for the relations [34]. For example, alias analysis can identify pairs that do not access the same location in any execution, allowing us to remove these pairs from the upper bounds of the rf and co relations. Removing such pairs decreases the number of SMT variables needed to encode these relations. Lower bounds can also simplify the encoding [36]. For example, for events within the same control flow block, the encoding of the po relation can be replaced by the encoding of whether both events have been executed. These techniques are not only applied to derive bounds for base relations, but also to compute bounds for all derived relations.

3 GPU CONSISTENCY

Extensive research has been devoted to clarify and formalize CPU memory consistency [16, 17, 32, 50, 54–57]. However, these approaches do not directly apply to other types of processing units, such as GPUs, due to their distinct architectural characteristics [39, 47, 52]. This section provides an overview of these GPU-specific characteristics. Section 4 shows how they are formalized within the Ptx and Vulkan consistency models.

3.1 Memory Scopes

To achieve high parallelism, modern GPUs utilize numerous processing cores to execute a massive number of threads in parallel. These threads can communicate via the GPU's global memory and through a shared memory segment provided by the host CPU. However, in both cases, the latency is typically high. To mitigate the latency, modern GPUs have a hierarchical memory system. A level of this hierarchy is referred to as a scope. Each scope is accessible by a different set of threads. The lowest scope may be accessible by e.g., those threads inside the same workgroup or the same compute thread array. The global memory of a GPU device forms another scope, accessible by all threads running on the GPU. Intermediate scopes may exist for subsets of threads of different sizes. Finally, the largest scope encompasses the memory of the host shared with one or more accelerators in the heterogeneous system.

The concept of a memory scope has been introduced by the model of Sequential Consistency for Heterogeneous-Race-Free (HRF) [37] and later extended to support relaxed atomics by the HRF-relaxed model [33]. Modern GPU consistency models opted for using scopes. However, there is an alternative model [58], which defines the semantics of heterogeneous systems without the notion of scopes, aiming to reduce the complexity for programmers. Nevertheless, the alternative model has not been widely adopted in practice.

Both Ptx and Vulkan models employ scopes similar to those in the HRF-relaxed model [33]. The Ptx model [3] defines three scopes: *CTA* (compute thread array), *GPU* (global memory of a device), and *SYS* (the memory shared between all processing units in the system). The Vulkan model [12] specifies four scopes: *Subgroup*, *Workgroup*, *Queue family*, and *Device*. Vulkan does not specify a scope for the shared memory of the entire system, but it expresses this concept through the notion of *system-synchronizes-with*.

3.2 Memory Fences and Control Barriers

GPUs offer fine-grained synchronization mechanisms via memory fences and control barriers. Similar to CPU fences, a GPU memory fence can be used to synchronize accesses to shared memory. In both Ptx and Vulkan models, besides a memory ordering, the semantics of a fence includes the scope at which the memory should be synchronized.

While fences synchronize memory accesses, a control barrier synchronizes the execution of threads. In particular, it ensures that all threads within a specified scope reach the barrier before any of them can proceed further. In most architectures, the scope of control barrier synchronization is limited to a workgroup [60]. In Vulkan, a control barrier can have both control and memory ordering semantics.

The litmus format distinguishes between different control barriers via an identifier. Synchronization is only effective for barriers with the same identifier.

3.3 Address Spaces

State-of-art GPUs use specialized accelerators with their dedicated caches to boost performance [3]. Notable examples include Nvidia's acceleration of surface and texture graphics operations [47], AMD's texture filtering logic [39], and Intel's texture sampler unit [43]. Compared to the generic memory paths, such specialized caches often provide weaker coherence guarantees. Moreover, the same memory address can be accessed via the conventional memory path and via a specialized cache. In the following, we outline how specialized caches are represented in GPU consistency models.

Proxies. Ptx introduced the notion of proxies to handle specialized accelerators with non-coherent cache hierarchies [47]. The model distinguishes between four proxies: *GEN* (generic proxy), *TEX* (texture proxy), *SUR* (surface proxy), and *CON* (constant proxy). A generic proxy represents the conventional path to the memory. Texture and surface proxies describe the dedicated caches for graphics workloads. A constant proxy represents a special memory for storing constant values, e.g., kernel launch parameters. Note that constant proxy should be included into a consistency model, because constant memory can be updated by a host during the execution of a kernel [47]. When two instructions access a memory address via the same proxy (and via the same reference), then they are said to be accessing the same virtual memory location. For a generic proxy, it means that the instructions are subject to the consistency guarantees of the core Ptx model.

Additional proxy fences may be required to synchronize memory accesses via different proxies. The Ptx model defines four proxy fences: alias-proxy, texture-proxy, surface-proxy, and constant-proxy. The alias-proxy fence reestablishes the ordering of memory accesses to the same location via different proxies. The texture-proxy, surface-proxy, and constant-proxy fences synchronize the respective specialized cache with the generic proxy.

Storage Classes. Vulkan uses the concept of storage classes to represent memory accesses via different cache types. The formal model does not explicitly list all storage classes, but uses two abstract definitions: `sc0` and `sc1`. All memory operations have an associated storage class, the one of the corresponding memory object reference.

In addition to storage classes, the model uses the notion of storage class semantics. The latter can be specified for atomic memory operations and memory barriers used for synchronization. It defines which storage class the synchronization should be applied to. One instruction can combine multiple storage class semantics. The formal model uses abstract definition of storage class semantics: `semsc0` and `semsc1`.

Similar to Ptx, Vulkan has a concept of two operations accessing the same memory location via the same reference. In the Vulkan model, this concept is referred to as using the same reference. We will use the Ptx terminology of same virtual address for both models.

3.4 Availability and Visibility

In Ptx, the release-acquire semantics is generally strong enough to make sure that one thread observes a value stored by another thread. In Vulkan however, this alone might not be sufficient. To provide more fine-grained synchronisation, Vulkan introduces the concepts of availability and visibility. Availability means making a stored value accessible in a particular cache. Visibility means that a thread observes a value available in a cache. Thus, availability is only relevant for writes, and visibility is only relevant for reads. Atomic memory accesses are available or visible by default. However, non-atomic accesses may require additional availability and visibility flags for proper synchronization.

We explain how availability can be enforced for writes. Enforcing visibility for reads is symmetric. The value of a non-atomic write can be made available using a successive memory fence or atomic write. In Vulkan, memory fences and atomic writes can have an availability semantics flag. If this flag is set, then preceding writes will be made available if they have the same scope and if their storage class matches the storage class semantics of the current instruction. Alternatively, a non-atomic write may use an availability flag to enforce the availability operation only for its own value.

3.5 Data Races

Besides safety, DRF is an important property in the presence of concurrency. Data races can introduce unpredictable and unintended behaviors. In Ptx, two memory operations accessing the same location are considered a data race when they lack a causal relationship and they are not morally strong. However, Ptx does not consider this to be undefined behavior since the consistency model does not require DRF [48]. In Vulkan, a data race occurs whenever two memory accesses (at least one being a write) with overlapping sets of memory locations are neither mutually-ordered atomic operations nor location-ordered. Vulkan considers data races to be undefined behavior, and thus developers must ensure their applications are DRF.

4 FORMAL MODELS FOR GPU CONSISTENCY

Several consistency models for heterogeneous architectures have been formalized in the past. Those include HSA [51], OpenCL [26], Ptx [14, 47, 48, 65], Vulkan [12] and FPGAs [40]. The first models were written in `.cat`. However, those only covered basic features of GPUs and their tool support has been discontinued. The most recent models are written in Alloy [41]. Unfortunately, these models use different representations of the same basic concepts, making it difficult to distinguish between differences in their consistency guarantees. Therefore, we opted to use `.cat`.

We describe the extensions made to the `.cat` grammar from Figure 2 to support all GPU features used by Ptx and Vulkan models. Concretely, we added the base relations described in Table 1, and the event tags listed in Table 2. The rest of this section describes the extensions in detail.

4.1 Ptx Consistency Model

The first formal analysis for a fragment of Ptx was proposed in [14]. However, it was not until ISA v6.0 that Nvidia provided

Relation	Related events and semantics	Model
co	Write to the same location. Must be a partial order.	PTX
vloc	Same virtual address specified in program source code.	PTX, VULKAN
sr	Events within the same scope.	PTX
scta	Events within the same CTA scope.	PTX
ssg, swg, sqf	Events within the same subgroup, workgroup, queue family.	VULKAN
ssw	Events from threads marked as system-synchronizes-with.	VULKAN
syncbar	Barriers with same logical barrier id.	VULKAN
sync_barrier	Barriers with same logical barrier id. Must be a subset of scta.	PTX
sync_fence	Morally strong SC fences. Must be a partial order.	PTX

Table 1: Semantics of new (or modified) base relations.

Set	Contained Instructions	Model
ACQ, REL	Acquire or release atomic memory accesses	PTX, VULKAN
RLX	Relaxed atomic memory accesses	PTX
PRIV	Thread-private memory accesses	VULKAN
SG, WG, QF, DV	Memory accesses to subgroup, workgroup, queue family, or device scope	VULKAN
GEN, SUR, TEX, CON	Memory accesses through generic, surface, texture, or constant memory proxy	PTX
ALIAS	Proxy fences to synchronize across to different generic proxies	PTX
SC0, SC1	Memory accesses to storage class 0 or storage class 1	VULKAN
SEMSC0, SEMSC1	Atomic memory accesses or fences with storage class semantic 0 or semantic 1	VULKAN
AV, VIS	Memory accesses with availability or visibility flag	VULKAN
SEMAV, SEMVIS	Atomic memory accesses or fences with availabiity or visibility semantic	VULKAN
AVDEVICE, VISDEVICE	Forcing availability or visibility to device domain	VULKAN

Table 2: New event tags and the instructions they model.

```
1  (* Proxy *)
2  let sameProx = GEN * GEN | SUR * SUR | TEX * TEX | CON * CON
3  let povloc = po & vloc
4
5  let co = co+
6  let fr = rf^-1; co
7
8  (* Morally-strong *)
9  let ms1 = (po | po^-1) | ([strongOp]; sr; [strongOp])
10 let ms2 = sameProx
11 let ms3 = ((M * M) & vloc) | ((_ * _) \ (M * M))
12 let ms = (ms1 & ms2 & ms3) \ id
13
14 (* Proxy-aware causality ordering *)
15 let pxyFM = [F]; (sameProx & scta); [M]
16 let proxyPreservedCauBase =
17   ( [GEN]; (vloc & cauBase); [GEN])
18   | ([M]; (sameProx & scta & vloc & cauBase); [M])
19   | vloc & (cauBase & (pxyFM^-1); cauBase; [GEN])
20   | vloc & ([GEN]; cauBase; cauBase & pxyFM)
21   | vloc & (cauBase & (pxyFM^-1); cauBase; cauBase & pxyFM)
22   |  loc & ([GEN]; cauBase; [F & ALIAS]; cauBase; [GEN])
23   |  loc & (cauBase & (pxyFM^-1); cauBase; [F & ALIAS]; cauBase; [GEN])
24   |  loc & ([GEN]; cauBase; [F & ALIAS]; cauBase; cauBase & pxyFM)
25   |  loc & (cauBase & (pxyFM^-1); cauBase;
26           [F & ALIAS]; cauBase; cauBase & pxyFM)
27 let cause = observation?; proxyPreservedCauBase
28
29 (* Axioms Coherence *)
30 empty ((([W]; cause; [W]) & loc) \ co)
31 empty (([W]; ms; [W]) & loc) \ (co | co^-1)
32 (* Axiom FenceSC *)
33 empty (([F & SC]; cause; [F & SC]) \ sync_fence)
34 (* Axiom Atomicity *)
35 empty (((ms & fr); (ms & co)) & rmw)
36 (* Axiom No-Thin-Air *)
37 let dep = addr | data | ctrl
38 acyclic (rf | dep)
39 (* Axiom Causality *)
40 irreflexive ((rf | fr); cause)
```

Figure 4: (Part of) NVIDIA PTX consistency model with mixed-proxy extension [47] defined in `.cat`.

a detailed description of its consistency model. This version was formalized in [48] and then extended with proxies from v7.5 [47]. We formalized both v6.0 and v7.5 in the `.cat` language. Our model is given in Figure 4. Due to the lack of space, we present only a part of the model for v7.5.

Events within a reachable scope are related by relation sr from Table 1. Relation scta is the subset of sr relating only events within the same CTA, and where the instructions they model are tagged by `.cta`. To model proxies, we follow [47] and use the new base event tags from Table 2 to define the sameProxy relation on line 2. The core idea (see lines 15-27) is that operations performed over the same proxy give the same guarantees as generic memory operations in PTX-v6.0. Supporting proxies only required adding to `.cat` the tags GEN, SUR, TEX and CON and used them in events modeling respectively `.generic`, `.surface`, `.texture`, and `.constant` instructions.

Figure 5 shows a message passing (MP) pattern using proxies. The prelude defines which address space the variables belong to and which location they alias to. The surface store `sust` followed by the fence in `P0` synchronizes `s` and `x` over the surface space. Thread `P1` first synchronizes `x` and `y` over the generic proxy (using an alias fence) and then `y` and `t` over the texture space using a texture load `tld`. Together with the release-acquire ordering over `flag`, this guarantees that `P1` observes the first write from `P0`, thus, the final ~ `exists` condition holds.

For CPU consistency models, coherence is a total order over all writes to each address. This is part of the well-defined notion introduced in Section 2.2. In PTX, however, coherence is a partial (not necessary total) transitive order that relates writes to the same location[2], such that two events are related if they are morally strong or if they are related in causality order. We relaxed the notion of well-defined in `.cat`, for co not to be total for PTX. Transitivity is guaranteed by line 5. The remaining semantics are enforced by the axioms on lines 30-31. An example showing that coherence might not be total is given in Figure 6. PTX allows threads `P2` and `P3` to observe contradicting values of `x`. If coherence were total, which could be enforced by adding the axiom **empty** `(((W * W) & loc) \ co)`, or the writes would be atomic, the behavior would be forbidden.

The semantics of morally-strong are only informally discussed in [47, 48]. We give a formal definition on lines 9-12. Two distinct operations are *morally-strong* if they satisfy all the following: *(ms_1)* they are in program order or each operation is strong (i.e., a fence

[2]The specification refers to "overlapping operations", but since we do not consider mixed-size concurrency, we use "same location" instead.

```
{
 y @ generic aliases x;
 s @ surface aliases x;
 t @ texture aliases y;
}
 P0@cta 0,gpu 0          |P1@cta 0,gpu 0          ;
 sust.weak s, 1          |ld.acquire.cta r0, flag ;
 fence.proxy.surface     |fence.proxy.alias       ;
 st.release.cta flag, 1 |fence.proxy.texture     ;
                         |tld.weak r1, t          ;
~exists
 (P1:r0 == 1 /\ P1:r1 != 1)
```

Figure 5: Message passing (MP) across different proxies.

```
 P0@cta 0,gpu 0| P1@cta 0,gpu 0| P2@cta 0,gpu 0      | P3@cta 0,gpu 0       ;
 st.weak x, 1  | st.weak x, 2  | ld.acquire.sys r0, x| ld.acquire.sys r2, x ;
               |               | ld.acquire.sys r1, x| ld.acquire.sys r3, x ;
exists
 (P2:r0 = 1 /\ P2:r1 = 2 /\ P3:r2 = 2 /\ P3:r3 = 1)
```

Figure 6: Non-causal weak writes are not ordered by coherence in Ptx.

```
 P0@cta 0,gpu 0  | P1@cta 0,gpu 0 | P2@cta 0,gpu 0 ;
 st.weak x, 1    | st.weak y, 1   | st.weak z, 1   ;
 ld.weak r2, z   | bar.cta.sync 1 |                ;
 bar.cta.sync r2 | ld.weak r1, x  |                ;
 ld.weak r0, y   |                |                ;
forall
 (P0:r0 == 1 \/ P1:r1 == 1)
```

Figure 7: Store buffering (SB) with dynamic control barrier.

or an atomic memory access) and they are related by sr, *(ms_2)* they are in the same proxy, and *(ms_3)* if both are memory operations, then they overlap completely (i.e., they are related by vloc).

The final changes to `.cat` are related to synchronization via fences and barriers. Relation sync_fence is a partial order (enforced at runtime; see the SMT encoding in Section 6) that relates every pair of morally strong `fence.sc` instructions. The axiom on line 33 states that sync_fence cannot contradict causality order. Relation sync_barrier is the subset of scta relating control barriers sharing the same id (whose value might be static or determined at runtime). The store buffering (SB) pattern in Figure 7 shows an example using control barriers. Since the load to `r2` can read the initial value of z, which litmus tests assume to be zero, it is not guaranteed for the barrier in `P0` to synchronize with the barrier in `P1` having id=1, and the `forall` condition can be violated.

4.2 Vulkan Consistency Model

We added to `.cat` new new base relations in Table 1 and event tags in Table 2 for scopes, control barriers, different address spaces, cache control, and system level synchronization. Following the Alloy model from [12], our `.cat` consistency model for Vulkan is given in Figure 8.

While in Ptx there is a single base relation for all scopes, Vulkan defines one base relation for each of the inner scopes: same subgroup (ssg), workgroup (swg) and queue family (sqf). It then derives from them the definition of inscope on lines 11-13. To restrict to those instructions that have a large enough scope tag and can synchronize, the model then intersects, e.g., swg with (DV|QF|WG) ∗ (DV|QF|WG). Restrictions to swg and sqf use similar intersections (see lines 11-13).

```
(* Sets *)
let PRIV = (R | W) \ NONPRIV
let SEMSC01 = SEMSC0 & SEMSC1
let AVSH = (AV | SEMAV) & DV
...
let VIWG = (VIS | SEMVIS) & (DV | QF | WG)
let AVSG = AV | SEMAV
let VISG = VIS | SEMVIS

(* Static Relations *)
let inscope = (DV * DV) | (sqf & ((DV | QF) * (DV | QF)))
 | (swg & ((DV | QF | WG) * (DV | QF | WG)))
 | (ssg & ((DV | QF | WG | SG) * (DV | QF | WG | SG)))
let chains = ((M | F | CBAR | AVDEVICE | VISDEVICE) \ IW) *
 ((M | F | CBAR | AVDEVICE | VISDEVICE) \ IW)
let moa = (loc & vloc & inscope & (A * A)) \ id \\ mutordatom

(* Dynamic Relations *)
let asmo = co & ((A | IW) * A)
let fr = (rf^-1; asmo) | (([IW]; rf)^-1; ((loc;[W]) \ id))
let fre = fr & ext
let rs = [REL & A]; ((asmo \ (asmo; asmo+)); [RMW])*
let hypors = [W & A]; ((asmo \ (asmo; asmo+)); [RMW])*
let sw = inscope & \\ synchronizes-with
 ( ([REL & A]; rs; (rf & moa); [ACQ & A])
...
 | ([REL & F]; posemtosc; [A & W]; hypors; (rf & moa);
    [A & R]; posctosem; [ACQ & F])
 | ([REL & F]; po?; [CBAR]; ((syncbar & inscope) \ id);
    [CBAR]; po?; [ACQ & F]))
let ithbsemsc0 = (ssw | ([SEMSC0]; sw; [SEMSC0])
 | ([SC0 | SEMSC0]; po; [REL & SEMSC0])
 | ([ACQ & SEMSC0]; po; [SC0 | SEMSC0]))+
...
let hb = ithbsemsc0 | ithbsemsc1 | ithbsemsc01 | po
let avwg = (chains & ([AVSG]; (hb & ssg & avinc))?); [AVWG]]
...
let vissh = [VISH]; (chains & (((hb & sqf & avinc); visqf)?;
 ((hb & swg & avinc); viswg)?; ((hb & ssg & avinc); [VISG])?))
let locord = loc & ((hb & int & vloc)
 | ([R \ PRIV]; hb; [(R | W) \ PRIV])
 | ([R]; ssw+; [R | W])
 | (vloc & ([W \ PRIV]; (po? & avinc); [AVSG]; (hb & ssg);
    [W \ PRIV]))
...
 | ([W]; (hb & avinc); [AVDEVICE]; hb; [VISDEVICE];
    (hb & avinc); [R]))

(* Axiom Coherence *)
empty ([A & W]; moa; [A & W]) \ (asmo | asmo^-1)

(* Axiom Consistency Cycle *)
acyclic (locord | rf | fr | asmo)

(* Axiom Consistency Read-from *)
empty (rf; ([R \ A])) & (([W]; locord); ([W]; locord)+)

( * Axiom Atomicity * )
empty rmw & (fre; (asmo & ext))

(* Axiom Location Order Complete *)
let non-rmw-reads = (R \ RMW) * (R \ RMW)
empty ((((locord | locord^-1) | moa) \ non-rmw-reads)
  \ (loc \ id \ non-rmw-reads))

(* Data Races *)
let wm = ((W * W) | (W * R) | (R * W)) \ ((IW * _) | (_ * IW))
let dr = loc & (wm \ moa \ id \ (locord | locord^-1))
flag ~empty dr
```

Figure 8: (Part of) Vulkan consistency model [12] defined in `.cat`.

```
1  P0@sg 0, wg 0, qf 0                  | P1@sg 1,wg 0, qf 0                          | P2@sg 0,wg 1, qf 0                    ;
2  st.av.wg.sc0 x, 1                    | ld.atom.acq.wg.sc1.semsc0.semsc1 r0, y      | ld.atom.acq.dv.sc1.semsc0.semsc1 r1, z ;
3  st.atom.rel.wg.sc1.semsc0.semsc1 y, 1 | st.atom.rel.dv.sc1.semsc0.semsc1.semav z, 1 | ld.vis.dv.sc0 r2, x                    ;
4 filter
5  (P1:r0 == 1 /\ P2:r1 == 1)
6 exists
7  (P2:r2 == 0)
```

Figure 9: Non-transitivity of happens-before and HRF-indirect of Vulkan.

Address spaces and coherence are handled similarly to Ptx. Storage classes correspond to sets which are then used to define locord (which is similar to proxyPreservedCauBase in Ptx). The coherence axiom on line 50 is defined analogously to the one in Ptx. However, it refers to the scoped modification order asmo instead of co. While Ptx allows weak stores to be unordered w.r.t. coherence, Vulkan considers this a data race. The program in Figure 6 (assuming Vulkan's instructions syntax) has a data race and thus its behavior is undefined. This example shows GPU models can have subtle differences developers should be aware of.

We show other peculiarities of Vulkan using Figure 9. The `filter` statement forces one to consider only behaviours where acquire-reads get their values from release-writes. The use of `.sc0`, `.sc1`, `.semsc0`, and `.semsc1` annotations guarantees (see lines 31-35) the following happens-before chain

$$e_2^{P0} \xrightarrow{hb} e_3^{P0} \xrightarrow{hb} e_2^{P1} \xrightarrow{hb} e_3^{P1} \xrightarrow{hb} e_3^{P2} \xrightarrow{hb} e_2^{P2}.$$

Contrary to other consistency models (like C11), hb is not transitive in Vulkan. This means those edges alone are not sufficient to guarantee that e_2^{P2} observes the value written by e_2^{P0}. We need an availability-visibility chain as on lines 36-39 (which requires annotating e_3^{P1} with `.semav`). This example also shows that Vulkan is similar to HRF-indirect models [33, 37] in which two threads can synchronize indirectly through a third party, even if each of these interactions uses a different scope.

We emphasize one difference between our `.cat` model and the Alloy one from [12]. In the later, read-modify-write (RMW) instructions are represented by a single event which is both a read and a write. We model RMWs as a read-write pair. The axiom on line 59 (common to other consistency models) is required to guarantee there are no context switches between the execution of the pair.

The Vulkan model defines data races using the same approach as language models like C11 [26] or Lkmm [19]. It flags as buggy any behaviour where certain relation is not empty. In Figure 8, this relation is defined on line 68. A behaviour having two different accesses to the same location, where at least one is a write, and which are not related by mutordatom or locord, is flagged and considered a bug.

5 UNDERSTANDING THE FORMAL MODELS

This section presents several examples showing the need for formal consistency models and explaining how to use them to decide if a given behavior is allowed or not.

A compiler bug. Consider the example in Figure 10 with `P0` and `P1` belonging to different workgroups (warps in Cuda terminology), but sharing the same queue family zero (the default one in Vulkan for compute). The scope of each instruction is device (`.dv`), which includes workgroups and queue families, meaning both threads can synchronize using atomic instructions (`.atom`) and memory barriers (`membar`).

```
1  P0@sg 0,wg 0, qf 0     | P1@sg 0,wg 1, qf 0          ;
2  st.atom.dv.sc0 data, 1 | LC00:                      ;
3  membar.rel.dv.semsc0   | ld.atom.dv.sc0 r1, flag     ;
4  st.atom.dv.sc0 flag, 1 | membar.acq.dv.semsc0    (-) ;
5                         | bne r1, 0, LC01             ;
6                         | goto LC00                   ;
7                         | LC01:                       ;
8                         | membar.acq.dv.semsc0    (+) ;
9                         | ld.atom.dv.sc0 r2, data     ;
10                        | ld.sc0 r3, 1                ;
11 exists
12  (P1:r3 == 1 /\ P1:r2 != 1)
```

Figure 10: MP with a spinloop.

```
1  P0@sg 0,wg 0, qf 0     | P1@sg 0,wg 1, qf 0      ;
2  st.atom.dv.sc0 data, 1 | membar.acq.dv.semsc0    ;
3  membar.rel.dv.semsc0   | ld.atom.dv.sc0 r2, data ;
4  st.atom.dv.sc0 flag, 1 | ld.sc0 r3, 1            ;
5 exists
6  (P1:r3 == 1 /\ P1:r2 != 1)
```

Figure 11: Incorrect optimization by the NIR compiler [7].

Thread `P0` writes the `data` variable on line 2, does a release memory barrier on line 3, and then sets a `flag` variable on line 4 to inform the `data` is ready for thread `P1` to be read. Thread `P1` spins (lines 2-6) until it observes the `flag` being set. Now first consider a version of the program where the line 8 marked with (+) would have been commented out. The barrier inside the loop on line 4 of `P1` and the corresponding barrier on line 3 of `P0` guarantee proper release-acquire synchronization before `P1` reads `data` on line 9. Finally, the `exists` condition checks if `P1` can observe stale data in those behaviours where `P0` does not spin forever (termination is enforced by `P1:r3 == 1`). According to Vulkan's consistency model [12], this behavior is forbidden due to the proper use of release-acquire barriers (see lines 27-28 in Figure 8).

Since barriers are expensive in terms of performance, an optimizing compiler might want move the acquire barrier outside of the loop. We use (-) and (+) (remove line 4 from `P1` and add line 8) to distinguish the versions before and after this optimization. The optimization is in fact sound according to Vulkan's consistency model and it is used, e.g., by the Boost library [11]. However, the NIR compiler [5], seeing a spinloop on lines 2-6 without any memory barriers, used to incorrectly apply further optimizations, resulting in the unsound code given in Figure 11. The (incorrect) claim was that relaxed reads cannot be expected to be relied upon for synchronization and an infinite loop without any side effects could thus be eliminated. After the (unsound) loop removal optimization, observing stale `data` is possible and the program incorrect. Agreeing on the unsoundness of the optimization was not easy [7]. This was partly because the consistency model was under-specified, but also

```
1  P0@cta 0,gpu 0      | P1@cta 1,gpu 0       ;
2  st.relaxed.gpu d, 1 | ld.weak r0, t        ;
3  fence.sc.gpu        | beq r0, r3, LC00     ;
4  ld.weak r2, t       | fence.sc.gpu         ;
5  add r2, r2, 1       | ld.relaxed.gpu r1, d ;
6  st.weak t, r2       | LC00:                ;
7 exists
8  (P1:r0 == 1 /\ P1:r1 == 0)
```

Figure 12: Snippet of a work-stealing double-ended queue.

```
1  P0@cta 0,gpu 0              | P1@cta 1,gpu 0              ;
2  atom.acq.gpu.add r1, in, 1  | atom.acq.gpu.add r1, in, 1  ;
3  LC00:                       | LC10:                       ;
4  ld.acq.gpu r2, out          | ld.acq.gpu r2, out          ;
5  beq r1, r2, LC01            | beq r1, r2, LC11            ;
6  goto LC00                   | goto LC10                   ;
7  LC01:                       | LC11:                       ;
8  ld.weak r3, x               | ld.weak r3, x               ;
9  st.weak x, 1                | st.weak x, 2                ;
10 atom.rel.gpu.add r4, out, 1 | atom.rel.gpu.add r4, out, 1 ;
11 exists
12 (P0:r1 == P0:r2 /\ P1:r1 == P1:r2 /\ P0:r3 == 0 /\ P1:r3 == 0)
```

Figure 13: Ticket mutex from `libcu++` [2].

due to a misunderstanding by the compiler engineers about what subtle guarantees are given by the Vulkan model.

A consistency bug in a work-stealing deque. Figure 12 (adapted from [14] and written in the Nvidia Ptx assembly syntax of Dartagnan) shows a snippet of the ABP work-stealing deque [24] used in the context of graphics processors [38]. Using Nvidia terminology, each compute thread array (CTA, also sometimes called a thread block) owns a deque to push to and pop work from. If a CTA's deque is empty, it attempts to steal some work from another deque. The deque is implemented as an array with two indexes. The push operation increments the tail index `t`, while pop decrements it. The head index (not shown in code the snippet) is incremented by steal operations.

The original code had none of the fences (line 3 of `P0` and line 4 of `P1` were missing) and it has been empirically shown to be buggy without them [14]. Thread `P0` pushes some work to its deque on line 2 and increments the tail index `t` on lines 4-6. When `P1` tries to steal, it can observe the correct `t` index, but a stale value of the array. The reason for this is that without the fences, the two writes by `P0` are not ordered in the Ptx consistency model. This is again the MP pattern, showing that even well-known weak behaviors can elude the developer understanding of the consistency guarantees. This bug was found before Nvidia released the first official Ptx consistency model [48]. Using the model, together with our Dartagnan tool, we can automatically show that the buggy behavior is indeed allowed when fences are not used, and forbidden when they are.

Unnecessary fences. Formally reasoning about consistency does not only allow one to show the absence of errors, but also to optimize code. Consider the `libcu++` mutex implementation [2] in Figure 13 written in Ptx assembly. Two threads try to acquire the mutex by obtaining a unique ticket (lines 2-7). Uniqueness is guaranteed by an atomic increment to the variable `in` on line 2 in both threads. The thread then spins on lines 3-6 until the currently served ticket `out` matches its own. Once inside the critical section, it reads the shared resource `x` and assigns a new value to it. The mutex is then released by atomically incrementing the next ticket to be served on line 10. The safety condition on line 12 checks if both threads can succeed in acquiring the mutex (i.e., `P0:r1 == P0:r2 /\ P1:r1 == P1:r2`), but still both read the initial value of `x`, clearly violating mutual exclusion.

This implementation guarantees mutual exclusion according to Ptx consistency model, and we can check this by running Dartagnan. However, it also contains an unnecessary barrier which degrades performance. The atomic increments on line 2 can be relaxed from acquire (`.acq`) to relaxed (`.rlx`) without sacrificing correctness. Developers tend to be cautious about manually doing such optimizations due to the complexity of understanding all corner cases of the underlying consistency model. Dartagnan can be used to find such optimization possibilities in the code.

6 UNIFIED ANALYSIS OF GPU CONSISTENCY

Formal models allow building tools to automatically reason about the impact of weak behaviors on program correctness. Below we describe the current limitations of state-of-the-art tools for GPU consistency and how we address these limitations in our framework.

6.1 Tools for GPU Consistency

While there are prototypes for the Alloy models of Ptx [3] and Vulkan [12], they have several limitations.

(1) **Program syntax and constructs:** the Alloy tools only support pseudo-assembly syntax with no control flow instructions (i.e., just straight line code), making it impossible to reason about real code or at least non-trivial examples like those in Figures 10-13.
(2) **Tool maturity:** we found two bugs while using the Ptx v7.5 tool, one wrongly defining the morally strong relation [4], the other allowing out-of-thin-air behaviors even if the model forbids them [6]. Additionally, the Ptx v7.5 tool lacks support for the constant proxy and control barriers.
(3) **Incompatible models:** the Ptx v6.0 model does not have an associated tool. The program parser of [3] is tightly coupled with the Ptx v7.5 model, making it hard to backport it to the previous version. Similarly, we cannot use the same tool for Ptx and Vulkan, despite both models being written in Alloy.
(4) **Scalability:** Alloy-based tools for consistency models do not scale for large number of events events [28, 48, 49, 65].

To address these limitations, we extended Dartagnan [31, 34], an analysis tool for consistency models written in `.cat`, with support for Ptx v6.0, Ptx v7.5, and Vulkan consistency models. In addition to safety conditions, it can verify liveness (as defined in [44, 53] and described in Section 6.4) and properties specified within a `.cat` model. In particular, it can check for DRF according to the definition in the Vulkan model (see lines 66-69 from Figure 8).

Dartagnan encodes the program's semantics defined by the given consistency model into an SMT formula. Then it uses an off-the-shelf solver to find violations of a given correctness specification. To achieve scalability, Dartagnan uses a static analysis technique [34, 36] which derives lower and upper bounds for relations. These bounds drastically reduce the encoding size and speeds up the verification times. To support GPU consistency models, we have implemented (for each new base relation) bounds computation

Relation	Lower Bound	Upper Bound
sr	$\{(e_1,e_2) \in \mathbb{X}\times\mathbb{X} \mid thread(e1) \in scope(e2) \wedge thread(e2) \in scope(e1) \wedge \neg mutExcl(e_1,e_2) \wedge visibleFrom(thread(e_1), thread(e_2), scope(e_1))\}$	Same as Lower Bound
r ∈ {scta, ssg, swg, ssw}	$\{(e_1,e_2) \in \mathbb{X}\times\mathbb{X} \mid \neg mutExcl(e_1,e_2) \wedge visibleFrom(thread(e_1), thread(e_2), \mathrm{r})$	Same as Lower Bound
syncbar	$\{(e_1,e_2) \in \mathbb{B}\times\mathbb{B} \mid id(e_1) = id(e_2) \wedge \neg mutExcl(e_1,e_2)\}$	Same as Lower Bound
sync_barrier	$\{(e_1,e_2) \in \mathbb{B}\times\mathbb{B} \mid id(e_1) = id(e_2) \wedge \neg mutExcl(e_1,e_2) \wedge sameCTA(thread(e_1), thread(e_2))\}$	$\{(e_1,e_2) \in \mathbb{B}\times\mathbb{B} \mid \neg mutExcl(e_1,e_2) \wedge sameCTA(thread(e_1), thread(e_2))\}$
sync_fence	$\emptyset$	$\{(e_1,e_2) \in \mathbb{F}\times\mathbb{F} \mid memOrd(e_1) = SC \wedge memOrd(e_2) = SC \wedge \neg mutExcl(e_1,e_2)\}$
vloc	$\{(e_1,e_2) \in \mathbb{M}\times\mathbb{M} \mid mustAlias(e_1,e_2) \wedge sameVirtual(e_1,e_2) \wedge \neg mutExcl(e_1,e_2)\}$	$\{(e_1,e_2) \in \mathbb{M}\times\mathbb{M} \mid mayAlias(e_1,e_2) \wedge sameVirtual(e_1,e_2) \wedge \neg mutExcl(e_1,e_2)\}$

Table 3: Lower and upper bounds for new base relations.

$\bigwedge_{e_1,e_2\in\mathbb{X}\ \mathrm{r}\in\{\mathrm{sr,scta,ssg,swg,sqf,ssw,syncbar}\}}$	$(\mathrm{exec}_{e_1} \wedge \mathrm{exec}_{e_2}) \Leftrightarrow \mathrm{r}(e_1,e_2)$
$\bigwedge_{e_1,e_2\in\mathbb{X}}$	$(\mathrm{exec}_{e_1} \wedge \mathrm{exec}_{e_2} \wedge id(e_1) = id(e_2)) \Leftrightarrow \mathrm{sync\_barrier}(e_1,e_2)$
$\bigwedge_{(e_1,e_2)\in[\mathrm{F\ \&\ SC}];\mathrm{sr};[\mathrm{F\ \&\ SC}]}$	$((\mathrm{exec}_{e_1} \wedge \mathrm{exec}_{e_2}) \Leftrightarrow (\mathrm{sync\_fence}(e_1,e_2) \vee \mathrm{sync\_fence}(e_2,e_1)) \wedge (\mathrm{sync\_fence}(e_1,e_2) \Rightarrow \mathrm{clk}_{e_1}^{\mathrm{sync\_fence}} < \mathrm{clk}_{e_2}^{\mathrm{sync\_fence}}))$
$\bigwedge_{e_1,e_2\in\mathbb{X}}$	$\mathrm{vloc}(e_1,e_2) \Rightarrow (\mathrm{exec}_{e_1} \wedge \mathrm{exec}_{e_2} \wedge addr(e_1) = addr(e_2))$
$\bigwedge_{e_1,e_2\in\mathbb{X}}$	$\mathrm{co}(e_1,e_2) \Rightarrow (\mathrm{exec}_{e_1} \wedge \mathrm{exec}_{e_2} \wedge addr(e_1) = addr(e_2) \wedge \mathrm{clk}_{e_1}^{\mathrm{co}} < \mathrm{clk}_{e_2}^{\mathrm{co}})$

Table 4: SMT encoding for new base relations. The encoding of co only affects Ptx.

from Table 3 and SMT encoding from Table 4. For the new base tags, the integration was simple. It only required making the program parser to tag events according to the instructions they model.

We have added to Dartagnan three new frontends: two for the litmus syntax used in our examples (one for Ptx and one for Vulkan), and one for a subset of real Spir-V assembly. The former allows us to validate our models w.r.t. the Alloy ones, and the later to apply our tool to real code. To the best of our knowledge, this makes Dartagnan the first tool to analyze the effects of weak GPU consistency on real code.

6.2 Relation Analysis

Relation analysis is an approach to reduce the size of the SMT encoding for consistency models [34, 36]. It computes upper and lower bounds for all relations. To make use of this analysis in the context of GPU consistency, we compute bounds for all new base relations. The analysis uses this information to calculate bounds from every derived relation defined in the `.cat` model. The bounds of the new base relations are given in Table 3. Notice that while we adapted the semantics of co for Ptx, bounds are just approximations of the relation and they remain unchanged from [34, 36].

Relations sr, scta, ssg, swg, sqf, ssw and syncbar are static, meaning they are fully computed from the source code of the program. Lower and upper bounds of static relations coincide. For scope-based relations sr, scta, ssg, swg, sqf, and ssw, the bounds contain all non-mutually exclusive events where the scopes of the instructions match, and the corresponding threads are visible to each other according to that scope. The later is covered by the predicate *visibleFrom*. For syncbar, the bounds contain all pairs of non-mutually exclusive barriers with matching identifiers (in the Vulkan model, identifiers of control barriers are static).

For sync_barrier, the upper bound contains every pair of control barriers in the same CTA (if they are not mutually exclusive). If we can additionally guarantee that their ids coincide, we add the pair to the lower bound. For sync_fence, we know nothing about its lower bound. The upper bounds contains every non mutually exclusive fences having SC as its memory ordering.

The bounds of vloc[3] contain all pairs of memory events that can execute together and access the same virtual addresses. Moreover, the addresses must (lower bound) or may (upper bound) point to the same location. We use a field sensitive version of Andersen alias analysis [22] to decide this.

6.3 SMT Encoding

Each base relation has its own semantics which needs to be encoded into the SMT formula. Table 4 describes the encoding of the new relations introduced in Table 1. We explain this encoding in the following. Note that when encoding a relation, we only consider pairs from the upper bounds computed by the relation analysis.

For the static relations sr, scta, ssg, swg, sqf, ssw and syncbar, the encoding is simple. After parsing a program, we know whether an event pair belongs to a relation (i.e., upper and lower bounds coincide). We also encode vloc as a static relation. This is because, at the moment, we only support static references in GPU code, thus may and must alias analysis information coincide. Thus, we encode that a relation holds for two events if both are executed. For sync_barrier, the encoding is similar, except we additionally require both barriers to have the same identifiers.

The sync_fence relation has more complex semantics. It is a partial order of SC fences, while for each subset of fence events located within the same CTA is a total order. Whether two fence events belong to the same CTA is statically computed by the relation analysis and does not require encoding. To encode the partial and total orders, we follow the approach of [29]. Specifically, we introduce

[3]The Alloy model of Vulkan calls this relation sref. It reflects what descriptor is used to access memory.

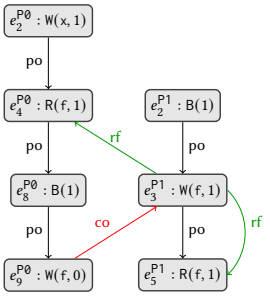

Figure 14: Liveness violation in XF-barrier from [66].

Tool	Safety	Liveness	DRF	#Tests	Time/Test (ms)
Ptx v6.0					
Dartagnan	106	73	0	179	797
Alloy	0	0	0	0	0
Ptx v7.5					
Dartagnan	235	73	0	308	841
Alloy	160	0	0	160	538
Vulkan					
Dartagnan	110	73	106	289	932
Alloy	101	0	82	183	3,464

Table 5: Comparing Dartagnan and Alloy-based tools for safety, liveness and DRF on litmus tests.

a clock variable for each fence event and enforce the ordering of those clock variables.

For the Vulkan model, co is encoded as a total order of writes to the same memory location, similar to the CPU memory models. However, in Ptx co is not total. Therefore, we do not enforce edges between writes to the same address, but use clock variables to enforce partial order.

6.4 Verification of Correctness Properties

While the Alloy-based tools check for safety and data races, Dartagnan also detects liveness violations as described in [44, 53]. The approach works for spinloops (i.e., side-effect-free loops) under the assumption that every thread has started and every enabled instruction is eventually executed. This progress assumption is not necessarily satisfied by GPUs [60] and thus, we cannot always claim the absence of liveness bugs (see Section 8). However, under the assumption that every thread is started, Dartagnan can find real liveness violations.

Finding liveness violations works as follows. For each thread, we start by detecting when a spinloop reads from the maximal event in coherence order, and the read value does not break the looping condition. Since a spinloop has no side-effects, the thread cannot help the progress of other threads. When at least one thread is spinning and all other threads cannot progress (they are either spinning or terminated), a liveness violation has been found. The co-maximality together with the fact other threads cannot progress guarantee no other write could stop the spinning.

Figure 14 shows a behaviour of the XF-Barrier with only two threads (one in each workgroup) and violating liveness. Event e_5^{P1} reads from the co-maximal event e_3^{P1}, but reading value 1 does not break the spinning. The problem is that e_9^{P0} and e_3^{P1} are non-atomic and thus not ordered by asmo (despite being related by co). Making e_4^{P0}, e_9^{P0}, e_3^{P1} and e_5^{P1} relaxed atomic removes the liveness violation. However, e_9^{P0} and e_5^{P1} must have also release-acquire semantics to avoid data races.

Dartagnan (as most model checking techniques) can only verify the correctness of an algorithm with respect to a given test case. For the one given in Figure 3 where *only one* thread writes before the barrier and *only one* reads after this, having e_9^{P0} and e_5^{P1} with release-acquire semantics and the other synchronization instruction as relaxed, is enough for correctness. However, since the barrier is expected to make all threads synchronize, one would expect *every* write before the barrier to be observable by *all* reads after it, as shown in Figure 1. We do not explicitly state the memory order of atomic instructions, but the compiler[4] generates release stores and acquire loads as in [60]. Table 7 shows that relaxing any of these barriers result in a bug.

7 EVALUATION

We performed several experiments to validate our tool and models. All experiments were conducted on an Ubuntu 22.04.4 LTS machine equipped with an 11th Gen Intel(R) Core(TM) i5-1135G7 processor (2.40 GHz, 8 cores) and 16 GB of RAM.

7.1 Model Validation

To validate our models, Table 5 compares Dartagnan and the Alloy-based tools on several litmus test. Our test suite includes all the original tests from [3, 12], papers formalizing Ptx consistency models [48, 49], all the examples in this paper, and other common weak consistency patterns. We additionally ported 73 litmus test[5] from the GPU Harbor project [45] for forward progress testing [62]. We use the latter to check for liveness. In total we collected 179 Ptx tests without proxies, 129 Ptx tests using proxies, and 289 Vulkan tests. Tests in the safety row contains `exists` or `forall` conditions representing interesting final states. From 106 Vulkan tests, we replace the `exists` with a `filter` and check for data races. This matches the way of checking races with the Alloy-based tool. We run Dartagnan on 102 tests since the remaining 4 ones do not have any consistent behaviour satisfying the filtering condition.

Table 5 shows that the tools based on Alloy support around one third of the tests. Ptx v6.0 does not have an associated tool. The missing safety tests in Ptx v7.5 and Vulkan use control flow instructions, control barriers or constant proxies. Neither this, nor liveness, is supported by any of the Alloy-based tools. For tests supported by both tools, all results match, giving us confidence to the correctness of our models. For the liveness tests, Dartagnan does not report any violation which is aligned with the results from [62] under classic progress models (which Dartagnan assumes).

[4]We use clspv to compile from OpenCL to the Spir-V assembly that is then passed to Dartagnan for the analysis.
[5]The remaining tests are unsupported by Dartagnan; see Section 8.

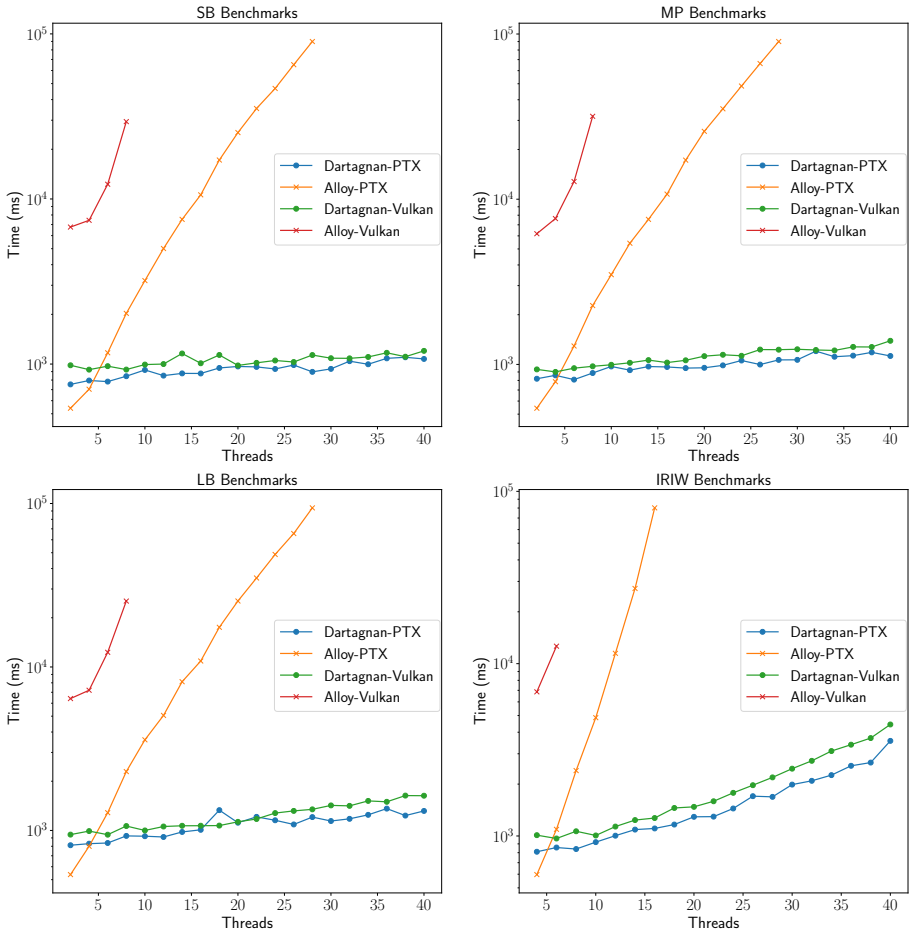

Figure 15: Comparing the scalability of Dartagnan and Alloy-based tools

7.2 Scalability

To show that Dartagnan can scale better than Alloy-based tools, we used four common weak consistency patterns (MP, SB, LB, and IRIW) to generate a series of tests with increasing number of threads. Figure 15 shows the number of threads used in each pattern in the x-axis, and the running time (in logarithm scale) in the y-axis. The running time for Alloy-based tools increases exponentially. Due to out-of-memory errors, we were unable to obtain results for the Alloy-Ptx tool for benchmarks with more than 25 threads (approximately 50 events), and for the Alloy-Vulkan tool when the number of threads exceeded 10 (approximately 20 events). However, the running time of Dartagnan only grows linearly w.r.t. the number of threads. Previous work has shown a similar performance between Dartagnan and herd7 [34]. Finally, as herd7 is also 10x faster than Isla [23], it follows that Dartagnan outperforms all those consistency model aware tools.

7.3 Tool Validation

To further validate our tool, we apply Dartagnan to the test suite of GPUVerify, a static analyser for verifying DRF of GPU kernels [25, 27]. This test suite contains a large number of real-world kernels from public and commercial sources, allowing us to asses how much of a real GPU API is supported by Dartagnan. We tried to compile the 486 OpenCL kernels in the test suite to Spir-V using the clspv compiler [10]. Compilation fails for 225 tests. From the remaining 261 tests, the compiler removes functions or loads in 84 tests due to unused returned or loaded values. These tests are trivially race-free, and thus we removed them from our evaluation. From the remaining 177 tests, Dartagnan can verify 66. The other ones require yet unsupported features like floating point variables.

Table 6 compares the number of tests supported by Dartagnan and GPUVerify and the average time per test. Despite the larger search space for Dartagnan due to weak consistency behaviors, its performance is competitive with GPUVerify. Verification results of both tools agree on 59/66 of the kernels. We manually checked the remaining ones. In 4/7 of the cases, GPUVerify contradicts comments in the OpenCL source code about the expected result, but Dartagnan matches them. For other 2/7 tests, GPUVerify reports failures due to barrier divergence, a correctness property that Dartagnan cannot yet verify. However, we believe the code

```
1  P0@sg 0, wg 0, qf 0          | P1@sg 1, wg 0, qf 0          ;
2  st.av.dv.sc0 x, 1            | cbar.acq_rel.dv.semsc0 0     ;
3  cbar.acq_rel.dv.semsc0 0     | rmw.atom.dv.sc0.add r0, x, 1 ;
4  rmw.atom.dv.sc0.add r0, x, 1 |                              ;
5 exists
6 (P0:r0 == 1 /\ P1:r0 == 1)
```

Figure 16: A program showing a bug in the Vulkan model.

Tool	#Tests	Time/Test (ms)
Vulkan		
Dartagnan	66	1679
OpenCL		
GPUVerify	177	1172

Table 6: Comparing Dartagnan and GPUVerify for DRF on kernels from [25, 27].

is DRF as reported by Dartagnan. In the last 1/7 test Dartagnan reports a race, but GPUVerify does not.

We show a simple variant of the test in Figure 16[6]. Vulkan consistency model allows both RMW-operations to read the same value, clearly violating atomicity. We explain why we believe this to be a bug in the model.

(1) If one makes the store atomic, the behavior becomes forbidden. This suggests the non-atomic store is problematic despite threads having the control barriers to synchronize.
(2) Replacing RMW-operations with atomic loads (while keeping the store as non-atomic) shows that the control barriers are sufficient to enforce the visibility of the store in both threads.
(3) Strengthening both atomic instructions to release-acquire semantics (the strongest semantics since there is no sequentially consistent memory order in Vulkan) is not sufficient to prevent both RMW-operations from reading the same value. Therefore, the problem is not the lack of memory ordering in the test.

We ported this test to Alloy to validate that it also allows this behavior. We reported the issue to the maintainers of the Vulkan model, which confirmed that the behaviour should be forbidden by the model and their tool [8].

7.4 Verification of Real Code

To support our claim that Dartagnan can verify real code, we implemented four synchronization primitives (caslock, ticketlock, ttaslock and the XF-Barrier) and checked for safety, liveness and DRF. These benchmarks use relaxed atomics, a relatively new, but important feature of real GPU APIs. GPUVerify supports strong, but not relaxed, atomics. However, it reports false positives even when using strong accesses. For example, it finds a data race in the critical section of caslock, even if all synchronization accesses are marked as sequentially consistent [1].

[6]The Spir-V test has a race involving indexes of an array. We use assertions on observed values, because Alloy-Vulkan doesn't support arrays.

Benchmark	Grid	|T|	|E|	Correct	Time (ms)
caslock	2.3	6	99	✓	23975
caslock-acq2rx	4.2	8	98	✗	1728
caslock-rel2rx	4.2	8	98	✗	1822
caslock-dv2wg	4.1	4	73	✓	1994
caslock-dv2wg	4.2	8	109	✗	2183
ticketlock	2.3	6	106	✓	33651
ticketlock-acq2rx	4.2	8	123	✗	1984
ticketlock-rel2rx	4.2	8	123	✗	1918
ticketlock-dv2wg	4.1	4	78	✓	1865
ticketlock-dv2wg	4.2	8	134	✗	2083
ttaslock	2.2	4	105	✓	65597
ttaslock-acq2rx	4.2	8	106	✗	2179
ttaslock-rel2rx	4.2	8	106	✗	1998
ttaslock-dv2wg	4.1	4	105	✓	21893
ttaslock-dv2wg	4.2	8	117	✗	2403
xf-barrier	3.3	9	457	✓	13598
xf-barrier-acq2rx-1	2.2	4	192	✗	2803
xf-barrier-acq2rx-2	2.2	4	192	✗	2792
xf-barrier-rel2rx-1	2.2	4	192	✗	2661
xf-barrier-rel2rx-2	2.2	4	192	✗	2777

Table 7: Verification of synchronization primitives (with different barriers and scopes).

Table 7 shows the results of the verification of real kernels. Columns Grid, $|T|$ and $|E|$ report respectively the thread organization (a grid X.Y means *"X threads per workgroup, and Y workgroups per device"*), number of threads and events. Benchmarks with a postfix `acq2rx` (resp. `rel2rx`) mean an acquire (resp. release) barrier was weakened to a relaxed one. Similarly, `dv2wg` means a device scope was reduced to workgroup.

All three lock implementations use release-acquire barriers and are correct. Relaxing any barrier results in a bug. If we keep the correct memory orderings, but their scope is reduced from device to workgroup, and threads belong to different workgroups (e.g., having a 4.2 grid), then the code is also incorrect since the instructions cannot be relied on for synchronization. Moving all threads to the same workgroup restores correctness. The XF-Barrier requires at least two workgroups, otherwise leaders in the first workgroup would blocked waiting for non-existing followers. Dartagnan shows that the code is correct with a 3.3. grid. Finally, relaxing any barrier results in a bug.

In terms of scalability, Dartagnan can find violations in a fraction of a second even if the benchmark has more than 100 events. When the benchmark is correct, the SMT solver needs to explore the whole search space and performance slows down. However, verification time still remains below 5 minutes. This shows that Dartagnan can handle much larger benchmarks that alloy-based tools [28, 48, 49, 65].

8 DISCUSSION

We believe our work is a step forward in helping developers understand better the guarantees given by different GPU consistency models. However, several problems remain open. This section discusses them and explains why they do not invalidate our results.

Progress guarantees. It is difficult to write correct GPU programs where threads rely on each another for making progress [59]. Computations are split into sub-tasks and then assigned to workgroups. Not all workgroups might execute concurrently (e.g., if they exceed the available resources), meaning the scheduler might delay execution of some workgroups until others have finished. If the spinning condition of a thread from an early workgroup depends on a thread from a later, delayed, workgroup, this will lead to starvation and execution will not terminate.

Even if most GPU specifications do not commit to any kind of progress guarantees, it has been empirically shown that most implementations provide some sort of progress between workgroups [60, 61]. Because of this, several progress models have been rigorously described [62]. In this paper we assume a forward progress model where whenever a thread becomes enabled, it will eventually be scheduled. The limitation of this is that Dartagnan might report false negatives about liveness violations, i.e., a *safe* result cannot be trusted.

Liveness definition. While DRF is encoded in the `.cat` model, safety and liveness are directly implemented as part of the SMT encoding phase of Dartagnan. The liveness encoding makes use of the co relation and does not consider scopes. Coherence not being total in Ptx is not a problem. This is because the SMT solver can still find a co-maximal event to falsify the liveness property. Using co instead of asmo in Vulkan is neither a problem. If two writes are related by co, but not asmo, the program has a data race bug that Dartagnan can detect. The example in Figure 14 fits in this category.

The lack of consideration for scopes in the liveness encoding is more problematic. In its current form, a spinloop reading-from a co-maximal event which is not in scope is considered a bug. However, these operations should not be allowed to synchronize. None of the litmus test used in our validation has this problem. In the future, we plan to solve this by moving the definition of liveness to the `.cat` model instead of directly encoding it in the SMT formula. This requires further extensions to `.cat`.

Models and tool validation. Section 7 includes not only all tests previously used to validate both the Ptx [3] and Vulkan [12] models, but also other interesting tests collected from the GPU consistency literature. However, it falls short w.r.t. validation approaches for CPU consistency [16, 20, 21, 42] or early GPU models [14]. Unfortunately, the online companion material for [14] is not accessible anymore. Additionally, herd7 (and its associated tool for systematically generating litmus tests) discontinued the support for Ptx. While there are recent approaches deploying large testing campaigns to check conformance of GPU consistency models [45, 46], they focus on a small set of common weak behaviors (and their mutants) instead of targeting GPU-specific features like address spaces, virtual aliasing and cache control. Extending frameworks to generate litmus tests using GPU-specific features is future work.

Table 5 contains some of, but not all, the tests used to validate progress models [62]. We excluded tests making use of exchange operations in spinloops (however, Dartagnan handles compare-and-swap). The reason is that the exchange of each iteration creates a new write event, making co not prefix-finite, which is a requirement to check for liveness in axiomatic consistency models [44]. Supporting those instructions requires extending the underlying theory of liveness. We see this as an interesting open problem.

9 ACKNOWLEDGEMENTS

We thank the anonymous reviewers and our shepherd Caroline Trippel for their feedback, which greatly improved the paper. We thank Daniel Lustig and Jeff Bolz for clarifications about the Ptx and Vulkan models. We also thank Thomas Haas for his feedback when integrating our changes into Dartagnan. This work was partially supported by the Research Council of Finland (336092 and 351145).

A ARTIFACT APPENDIX

A.1 Abstract

This work is accompanied by an artifact which contains the Dartagnan verification tool, and the cat models for Ptx-v6.0, Ptx-v7.5 and Vulkan. It also contains all benchmarks and the other tools discussed in the paper. Note that the results obtained for the same benchmarks by the open-source version of Dartagnan might differ in the future, as the tool will evolve.

A.2 Artifact Checklist (Meta-information)

- **Algorithm:** Dartagnan, a model-checking algorithm for weak consistency models.
- **Program:** The Dartagnan verification tool. Tests and benchmarks in the format of litmus tests, disassembled Spir-V kernels, and OpenCL kernels.
- **Transformations:** The docker image contains the clspv compiler v19.0.0git to generate Spir-V tests and Spirv-Tools v2024.1-0-g04896c46 to generate Spir-V tests.
- **Run-time environment:** A container engine supporting docker images is required to run the artifact container. The docker image contains openjre-17 to run Dartagnan and the Alloy-based tools.
- **Metrics:** Execution time.
- **Output:** Console, tables, and graphs.
- **Experiments:** The artifact contains ready scripts to reproduce results in the paper.
- **How much disk space required (approximately)?:** About 4 GB to store a ready image and run a container.
- **How much time is needed to complete experiments (approximately)?:** About two hours.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** See the webpage of Dartagnan https://github.com/hernanponcedeleon/Dat3M.
- **Data licenses (if publicly available)?:** All benchmarks to be run with Dartagnan are under MIT license. Litmus tests to be run with the Vulkan Alloy-based tool are under CC 4.0 license. Litmus tests to be run with the Ptx Alloy-based tool are under BSD-3 license. OpenCL tests to be run with GPUVerify are under Microsoft Public License (Ms-PL).
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.12516539.

A.3 Description

A.3.1 How to Access. The artifact (available on Zenodo) consists of a docker image with all tools and benchmarks that are required to validate the claims made in the paper.

A.3.2 Software Dependencies. Docker https://docs.docker.com/get-docker/.

A.3.3 Hardware Dependencies. Depending on your operating system, Docker might impose some extra hardware restrictions.

A.4 Installation

To load the docker image, run

```
> docker load < dat3m-gpu-artifact
```

Start a docker container:

```
> docker run -it --rm dat3m-gpu-artifact
```

A.5 Experiment Workflow and Expected Results

The artifact provides scripts to automatically generate results shown in Table 5, Table 6, Table 7 and Figure 15. The consistency models written in `.cat` (Figure 4 and Figure 8) are located in:

/home/Dat3M/cat/ptx-v6.0.cat
/home/Dat3M/cat/ptx-v7.5.cat
/home/Dat3M/cat/spriv.cat

Experimental Setup: The evaluation was done on an Ubuntu 22.04.4 LTS machine with an 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz (8 cores) and 16GB of RAM.

A.5.1 Model Validation. Table 5 compares the number of supported tests by Dartagnan and the Alloy-based tools. Tests in the litmus format supported by Dartagnan are located in:

/home/Dat3M/litmus/PTX
/home/Dat3M/litmus/VULKAN

For Alloy-Ptx, the tests are located in:

/home/mixedproxy/tests

The tests for Alloy-Vulkan are can be found in:

/home/Vulkan-MemoryModel/alloy/tests

To automatically generate all results in Table 5 run:

```
> python3 /home/scripts/generate-table-5.py
```

This will display the table in the console and generate a `table5.csv` file.

Expected execution time: 20 minutes.

A.5.2 Tool Validation. Table 6 compares the number of supported tests and the verification time of Dartagnan and GPUVerify. Both tools have been used to verify data race freedom of OpenCL programs in the test suite of GPUVerify.

For GPUVerify, we used the latest release of the tool (2018-03-22) along with the most recent OpenCL test suite from their GitHub repository, which can be found in

/home/gpuverify-release/latest_benchmarks/opencl

To execute the same benchmarks with Dartagnan, we compiled the OpenCL kernels into Spir-V using the clspv compiler and then disassembled the compiled files using Spirv-Tools. The artifact includes precompiled and disassembled files located in

/home/Dat3M/dartagnan/src/test/resources/spirv/gpuverify

To automatically generate all results in Table 6 run:

```
> python3 /home/scripts/generate-table-6.py
```

This will display the table in the console and generate a `table6.csv` file.

Expected execution time: 10 minutes.

A.5.3 Verification of Real Code. Table 7 shows verification times of Dartagnan for a set of synchronization primitives. These synchronization primitives had been initially implemented in OpenCL, and then compiled to Spir-V assembly. The artifact includes precompiled and disassembled benchmarks located in

/home/benchmarks/spirv

To automatically generate all results in Table 7 run:

```
> python3 /home/scripts/generate-table-7.py
```

This will display the table in the console and generate a `table7.csv` file.

Expected execution time: 5 minutes.

A.5.4 Scalability. Figure 15 supports our claim that Dartagnan can be an order of magnitude faster than Alloy-based tools. To automatically generate the plots showing this, run:

```
> python3 /home/scripts/generate-plots.py
```

This will generate several data files (`MP.csv`, `SB.csv`, `LB.csv`, `IRIW.csv`) and plot files (`MP.png`, `SB.png`, `LB.png`, `IRIW.png`) in the `/home/scripts/output` directory. The folder `/home/output` can be copied to the host machine using the following command:

```
> docker cp <containerId>:/home/output/ .
```

Expected execution time: 60 minutes.

A.6 Experiment Customization

This section describes how to execute individual tests.

A.6.1 Running a litmus test with Dartagnan. To run an individual test, use the following command:

```
> cd /home/Dat3M && \
  java -jar dartagnan/target/dartagnan.jar \
  <path/to/test.litmus> \
  cat/<ptx-v6.0.cat|ptx-v7.5.cat|spirv.cat> \
  --property=<program_spec|cat_spec|liveness> \
  --target=<ptx|vulkan> \
  --method=assume
```

The litmus tests used in the experiments are located respectively in:

/home/Dat3M/litmus/PTX
/home/Dat3M/litmus/VULKAN
/home/benchmarks/dat3m_ptx
/home/benchmarks/dat3m_vkn

A.6.2 Running a Spir-V test with Dartagnan. To run a single Spir-V test with Dartagnan, use the following command:

```
> cd /home/Dat3M && \
  java -jar dartagnan/target/dartagnan.jar \
  <path/to/test.spv.dis> \
  cat/spirv.cat \
  --property=<program_spec|cat_spec|liveness> \
  --bound=<loop-bound> \
  --target=vulkan \
  --encoding.integers=true \
  --method=assume
```

The tests used in the experiments are located respectively in:

/home/Dat3M/dartagnan/src/test/resources/spirv
/home/benchmarks/spirv

REFERENCES

[1] Data race reported on CAS-lock implementation. https://github.com/mc-imperial/gpuverify/issues/55. Accessed: 04/15/2024.

[2] libcu++: The C++ Standard Library for Your Entire System. https://github.com/NVIDIA/libcudacxx/blob/main/benchmarks/concurrency.cpp. Accessed: 09/26/2023.

[3] Mixed-proxy extensions for the NVIDIA PTX memory consistency model. https://github.com/NVlabs/mixedproxy. Accessed: 09/26/2023.

[4] Morally Strong Definition. https://github.com/NVlabs/mixedproxy/issues/1. Accessed: 09/26/2023.

[5] NIR Intermediate Representation (NIR). https://docs.mesa3d.org/nir/index.html. Accessed: 09/26/2023.

[6] No-Thin-Air axiom testing. https://github.com/NVlabs/mixedproxy/issues/2. Accessed: 09/26/2023.

[7] Relaxed atomic loads in while loops being optimized away. https://gitlab.freedesktop.org/mesa/mesa/-/issues/4475. Accessed: 09/26/2023.

[8] RMW atomicity is broken. https://github.com/KhronosGroup/Vulkan-MemoryModel/issues/36. Accessed: 06/23/2024.

[9] SPIR-V, Extended Instruction Set, and Extension Specifications. https://registry.khronos.org/SPIR-V/. Accessed: 06/12/2024.

[10] The Compiler. https://github.com/google/clspv. Accessed: 04/24/2024.

[11] Thread coordination using Boost.Atomic. https://www.boost.org/doc/libs/1_55_0/doc/html/atomic/thread_coordination.html#atomic.thread_coordination.fences. Accessed: 09/26/2023.

[12] Vulkan-MemoryModel. https://github.com/KhronosGroup/Vulkan-MemoryModel. Accessed: 09/26/2023.

[13] Jade Alglave. A shared memory poetics. *These de doctorat, L'université Paris Denis Diderot*, 2010.

[14] Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. GPU concurrency: Weak behaviours and programming assumptions. In Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas, editors, *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015*, pages 577–591. ACM, 2015.

[15] Jade Alglave, Patrick Cousot, and Luc Maranget. Syntax and semantics of the weak consistency model specification language cat. *CoRR*, abs/1608.07531, 2016.

[16] Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. Armed cats: Formal concurrency modelling at Arm. *ACM Trans. Program. Lang. Syst.*, 43(2):8:1–8:54, 2021.

[17] Jade Alglave, Anthony C. J. Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. The semantics of power and ARM multiprocessor machine code. In Leaf Petersen and Manuel M. T. Chakravarty, editors, *Proceedings of the POPL 2009 Workshop on Declarative Aspects of Multicore Programming, DAMP 2009, Savannah, GA, USA, January 20, 2009*, pages 13–24. ACM, 2009.

[18] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 141–157. Springer, 2013.

[19] Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan S. Stern. Frightening small children and disconcerting grown-ups: Concurrency in the Linux kernel. In Xipeng Shen, James Tuck, Ricardo Bianchini, and Vivek Sarkar, editors, *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, pages 405–418. ACM, 2018.

[20] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Litmus: Running tests against hardware. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6605 of *Lecture Notes in Computer Science*, pages 41–44. Springer, 2011.

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

[22] Lars Ole Andersen and Peter Lee. Program analysis and specialization for the C programming language. 2005.

[23] Alasdair Armstrong, Brian Campbell, Ben Simner, Christopher Pulte, and Peter Sewell. Isla: Integrating full-scale ISA semantics and axiomatic concurrency models. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, volume 12759 of *Lecture Notes in Computer Science*, pages 303–316. Springer, 2021.

[24] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In Gary L. Miller and Phillip B. Gibbons, editors, *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '98, Puerto Vallarta, Mexico, June 28 - July 2, 1998*, pages 119–129. ACM, 1998.

[25] Ethel Bardsley and Alastair F. Donaldson. Warps and atomics: Beyond barrier synchronization in the verification of GPU kernels. In Julia M. Badger and Kristin Yvonne Rozier, editors, *NASA Formal Methods - 6th International Symposium, NFM 2014, Houston, TX, USA, April 29 - May 1, 2014. Proceedings*, volume 8430 of *Lecture Notes in Computer Science*, pages 230–245. Springer, 2014.

[26] Mark Batty, Alastair F. Donaldson, and John Wickerson. Overhauling SC atomics in C11 and OpenCL. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 634–648. ACM, 2016.

[27] Adam Betts, Nathan Chong, Alastair F. Donaldson, Shaz Qadeer, and Paul Thomson. Gpuverify: a verifier for GPU kernels. In Gary T. Leavens and Matthew B. Dwyer, editors, *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 113–132. ACM, 2012.

[28] James Bornholt and Emina Torlak. Synthesizing memory models from framework sketches and Litmus tests. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 467–481. ACM, 2017.

[29] Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. Portability analysis for weak memory models. PORTHOS: One tool for all models. In Francesco Ranzato, editor, *Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings*, volume 10422 of *Lecture Notes in Computer Science*, pages 299–320. Springer, 2017.

[30] Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. BMC with memory models as modules. In Nikolaj S. Bjørner and Arie Gurfinkel, editors, *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, pages 1–9. IEEE, 2018.

[31] Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. Dartagnan: Bounded model checking for weak memory models (Competition contribution). In Armin Biere and David Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II*, volume 12079 of *Lecture Notes in Computer Science*, pages 378–382. Springer, 2020.

[32] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 608–621. ACM, 2016.

[33] Benedict R. Gaster, Derek Hower, and Lee W. Howes. HRF-relaxed: Adapting HRF to the complexities of industrial heterogeneous memory models. *ACM Trans. Archit. Code Optim.*, 12(1):7:1–7:26, 2015.

[34] Natalia Gavrilenko, Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. BMC for weak memory models: Relation analysis for compact SMT encodings. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 355–365. Springer, 2019.

[35] Andrés Goens, Soham Chakraborty, Susmit Sarkar, Sukarn Agarwal, Nicolai Oswald, and Vijay Nagarajan. Compound Memory Models. *Proc. ACM Program. Lang.*, 7(PLDI):1145–1168, 2023.

[36] Thomas Haas, René Maseli, Roland Meyer, and Hernán Ponce de León. Static analysis of memory models for SMT encodings. *Proc. ACM Program. Lang.*, 7(OOPSLA2), oct 2023.

[37] Derek R. Hower, Blake A. Hechtman, Bradford M. Beckmann, Benedict R. Gaster, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. Heterogeneous-race-free memory models. In Rajeev Balasubramonian, Al Davis, and Sarita V. Adve, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2014, Salt Lake City, UT, USA, March 1-5, 2014*, pages 427–440. ACM, 2014.

[38] Wen-mei W. Hwu. *GPU Computing Gems Jade Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

[39] AMD Inc. Introducing RDNA architecture, the all new Radeon gaming architecture powering "Navi", 2019.

[40] Dan Iorga, Alastair F. Donaldson, Tyler Sorensen, and John Wickerson. The semantics of shared memory in Intel CPU/FPGA systems. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–28, 2021.

[41] Daniel Jackson. Alloy: A language and tool for exploring software designs. *Commun. ACM*, 62(9):66–76, 2019.

[42] Jake Kirkham, Tyler Sorensen, Esin Tureci, and Margaret Martonosi. Foundations of empirical memory consistency testing. *Proc. ACM Program. Lang.*,

4(OOPSLA):226:1–226:29, 2020.

[43] Filip Kruzel and Mateusz Nytko. Intel Iris Xe-LP as a platform for scientific computing. In Maria Ganzha, Leszek A. Maciaszek, Marcin Paprzycki, and Dominik Slezak, editors, *Communication Papers of the 17th Conference on Computer Science and Intelligence Systems, FedCSIS 2022, Sofia, Bulgaria, September 4-7, 2022*, volume 32 of *Annals of Computer Science and Information Systems*, pages 121–128, 2022.

[44] Ori Lahav, Egor Namakonov, Jonas Oberhauser, Anton Podkopaev, and Viktor Vafeiadis. Making weak memory models fair. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–27, 2021.

[45] Reese Levine, Mingun Cho, Devon McKee, Andrew Quinn, and Tyler Sorensen. GPUHarbor: Testing GPU memory consistency at large (Experience paper). In René Just and Gordon Fraser, editors, *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, pages 779–791. ACM, 2023.

[46] Reese Levine, Tianhao Guo, Mingun Cho, Alan Baker, Raph Levien, David Neto, Andrew Quinn, and Tyler Sorensen. MC mutants: Evaluating and improving testing for memory consistency specification. In Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift, editors, *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pages 473–488. ACM, 2023.

[47] Daniel Lustig, Simon Cooksey, and Olivier Giroux. Mixed-proxy extensions for the NVIDIA PTX memory consistency model: Industrial product. In Valentina Salapura, Mohamed Zahran, Fred Chong, and Lingjia Tang, editors, *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, pages 1058–1070. ACM, 2022.

[48] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. A formal analysis of the NVIDIA PTX memory consistency model. In Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck, editors, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 257–270. ACM, 2019.

[49] Daniel Lustig, Andrew Wright, Alexandros Papakonstantinou, and Olivier Giroux. Automated synthesis of comprehensive memory model litmus test suites. In Yunji Chen, Olivier Temam, and John Carter, editors, *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, pages 661–675. ACM, 2017.

[50] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. An axiomatic memory model for POWER multiprocessors. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 495–512. Springer, 2012.

[51] Luc Maranget and Jade Alglave. Towards a formalization of the HSA memory model in the cat language, 2015.

[52] Chris McClanahan. History and evolution of GPU architecture. *A Survey Paper*, 9:1–7, 2010.

[53] Jonas Oberhauser, Rafael Lourenco de Lima Chehab, Diogo Behrens, Ming Fu, Antonio Paolillo, Lilith Oberhauser, Koustubha Bhat, Yuzhong Wen, Haibo Chen, Jaeho Kim, and Viktor Vafeiadis. VSync: Push-button verification and optimization for synchronization primitives on weak memory models. In *ASPLOS*, pages 530–545. ACM, 2021.

[54] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 391–407. Springer, 2009.

[55] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 175–186. ACM, 2011.

[56] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. The semantics of x86-CC multiprocessor machine code. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 379–391. ACM, 2009.

[57] Ben Simner, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell. Relaxed virtual memory in Armv8-A. In Ilya Sergey, editor, *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*, volume 13240 of *Lecture Notes in Computer Science*, pages 143–173. Springer, 2022.

[58] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. Efficient GPU synchronization without scopes: Saying no to complex consistency models. In Milos Prvulovic, editor, *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*, pages 647–659. ACM, 2015.

[59] Tyler Sorensen and Alastair F. Donaldson. The hitchhiker's guide to cross-platform OpenCL application development. In *Proceedings of the 4th International Workshop on OpenCL, IWOCL 2016, Vienna, Austria, April 19-21, 2016*, pages 2:1–2:12. ACM, 2016.

[60] Tyler Sorensen, Alastair F. Donaldson, Mark Batty, Ganesh Gopalakrishnan, and Zvonimir Rakamaric. Portable inter-workgroup barrier synchronisation for GPUs. In Eelco Visser and Yannis Smaragdakis, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 39–58. ACM, 2016.

[61] Tyler Sorensen, Sreepathi Pai, and Alastair F. Donaldson. One size doesn't fit all: Quantifying performance portability of graph applications on GPUs. In *IEEE International Symposium on Workload Characterization, IISWC 2019, Orlando, FL, USA, November 3-5, 2019*, pages 155–166. IEEE, 2019.

[62] Tyler Sorensen, Lucas F. Salvador, Harmit Raval, Hugues Evrard, John Wickerson, Margaret Martonosi, and Alastair F. Donaldson. Specifying and testing GPU workgroup progress models. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–30, 2021.

[63] Jiawei Wang, Diogo Behrens, Ming Fu, Lilith Oberhauser, Jonas Oberhauser, Jitang Lei, Geng Chen, Hermann Härtig, and Haibo Chen. BBQ: A block-based bounded queue for exchanging data and profiling. In Jiri Schindler and Noa Zilberman, editors, *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 249–262. USENIX Association, 2022.

[64] Jiawei Wang, Bohdan Trach, Ming Fu, Diogo Behrens, Jonathan Schwender, Yutao Liu, Jitang Lei, Viktor Vafeiadis, Hermann Härtig, and Haibo Chen. BWoS: Formally verified block-based work stealing for parallel processing. In Roxana Geambasu and Ed Nightingale, editors, *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, pages 833–850. USENIX Association, 2023.

[65] John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. Automatically comparing memory consistency models. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 190–204. ACM, 2017.

[66] Shucai Xiao and Wu-chun Feng. Inter-block GPU communication via fast barrier synchronization. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*, pages 1–12. IEEE, 2010.