A Formal Analysis of the NVIDIA PTX Memory Consistency Model

Daniel Lustig
dlustig@nvidia.com
NVIDIA

Sameer Sahasrabuddhe
NVIDIA[1]

Olivier Giroux
ogiroux@nvidia.com
NVIDIA

Abstract

This paper presents the first formal analysis of the official memory consistency model for the NVIDIA PTX virtual ISA. Like other GPU memory models, the PTX memory model is weakly ordered but provides scoped synchronization primitives that enable GPU program threads to communicate through memory. However, unlike some competing GPU memory models, PTX does not require data race freedom, and this results in PTX using a fundamentally different (and more complicated) set of rules in its memory model. As such, PTX has a clear need for a rigorous and reliable memory model testing and analysis infrastructure.

We break our formal analysis of the PTX memory model into multiple steps that collectively demonstrate its rigor and validity. First, we adapt the English language specification from the public PTX documentation into a formal axiomatic model. Second, we derive an up-to-date presentation of an OpenCL-like scoped C++ model and develop a mapping from the synchronization primitives of that scoped C++ model onto PTX. Third, we use the Alloy relational modeling tool to empirically test the correctness of the mapping. Finally, we compile the model and mapping into Coq and build a full machine-checked proof that the mapping is sound for programs of any size. Our analysis demonstrates that in spite of issues in previous generations, the new NVIDIA PTX memory model is suitable as a sound compilation target for GPU programming languages such as CUDA.

CCS Concepts • **Hardware → Theorem proving and SAT solving**; • **Software and its engineering → Consistency**.

Keywords Memory Consistency Models; GPUs; Theorem Proving; Model Finding; SAT Solving

ASPLOS '19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304043

ACM Reference Format:
Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. 2019. A Formal Analysis of the NVIDIA PTX Memory Consistency Model. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3297858.3304043

1 Introduction

A memory consistency model determines the values that can be legally returned by loads from memory. Currently there is a broad spectrum of CPU memory models, from relatively strong models such as x86-TSO [44] to relatively weak models such as IBM Power [9, 46, 48]. Today's general-purpose software (e.g., C/C++, Java, OpenCL), on the other hand, generally uses some form of data race-free (DRF) model layered on top of the hardware model [24, 28, 29, 42]. DRF models distinguish between non-synchronizing and synchronizing memory accesses. During compilation, non-synchronizing accesses may be freely optimized, but each software synchronization primitive must be mapped onto a particular set of assembly instructions according to a recipe specific to each software-hardware combination.

Deriving and proving correct a recipe for compiling software synchronization primitives onto a particular set of instructions for a given hardware instruction set architecture (ISA) is an important and widely-studied problem. Proving these recipes correct is challenging for a number of reasons:

- Industry memory models are typically complex, as they often favor performance, power, area, and architectural flexibility over simplicity.
- Empirical testing often runs into tractability limits and is inherently incomplete [2, 35].
- Memory models change regularly, either intentionally [12] or because bugs are found [11, 32, 51].
- Writing proofs can be hard and/or tedious, especially because the use of rigorous but pedantic theorem provers such as Coq [4] or HOL [5] is the accepted standard.
- Fundamental assumptions made in the proof may later turn out to be invalid [21, 36].

In spite of these challenges, previous work has successfully completed proofs of correctness for the mappings of C/C++ and Java synchronization primitives onto x86, ARMv7, ARMv8, and IBM Power [9, 44, 45, 48].

[1]Work performed while Sahasrabuddhe was at NVIDIA

GPUs originally targeted embarrassingly-parallel code and disallowed any communication among the thread blocks of active kernels. However, in a push towards generality, GPUs have begun to allow more general-purpose inter-thread communication within kernels. In turn, this requires a memory model to be defined. Without a sound memory model, the results of reading memory may be simply unpredictable. The HSA and Heterogeneous-Race-Free (HRF) memory models have provided formal GPU-specific memory model specifications [7, 27]. With the release of the Volta architecture and version 6.0 of the PTX virtual instruction set, NVIDIA has also provided a detailed natural language description of the PTX memory model [40]. The major contribution of this paper is a formalization and rigorous analysis of this model.

To ensure that the memory model does not impose undue burden on hardware architects, the memory models for GPUs (and hence for GPU-targeted software models such as CUDA and OpenCL) frequently employ some notion of *scope*, adding yet another dimension to the complexity of their memory model. The PTX ISA for NVIDIA GPUs uses a weakly-ordered and scoped memory model, but in contrast to the HRF and HSA memory models [7, 27], PTX does not declare racy programs to be illegal. This makes it an unique design point in the evolution of weak GPU memory models.

To provide insight into all of the issues listed above, we present the first formal analysis of the NVIDIA PTX 6.0 memory consistency model. Validating this model requires several distinct steps. First, we derive a formal axiomatic model from the online English language PTX specification [40]. Second, we encode the model in the Alloy relational modeling tool [30] following established techniques [35, 58]. Third, we build a compiler from Alloy to Coq [4], an interactive theorem prover used to machine-check mathematical proofs [8–10, 38, 56, 57]. Fourth, we empirically test the correctness of the mapping using Alloy. Finally, we formally prove the correctness of a mapping from an OpenCL-like scoped C++ memory model onto PTX using Coq.

Our multi-layered approach combines empirical testing (in Alloy) with formal verification (in Coq) into one unified flow and eliminates gaps between the natural language model, the model used for testing, and the model used for formal proofs. This workflow is also flexible enough to easily adapt to changes in model(s) and/or compilation recipes, thanks to the user-friendly interface Alloy provides. A full line-by-line derivation of our Alloy model from PTX documentation, our Alloy models for PTX and our variant of scoped C++ (later described in Section 4.1), our Alloy-to-Coq toolchain, and our full Coq proofs of PTX compliance with this scoped C++ model are available online [34].

2 Background

Memory consistency models have proven over the decades to be a notoriously challenging topic. Work in the eighties and nineties developed much of the theoretical backing for weak memory models, while recent years have seen significant effort spent to formalize widely used hardware and software models [1, 2, 9, 15, 33, 35, 44, 45, 47, 48, 55, 58]. These formalization efforts continue to uncover new bugs and surprises [18, 21, 57], and many of the important memory models in use today are still evolving [12, 19].

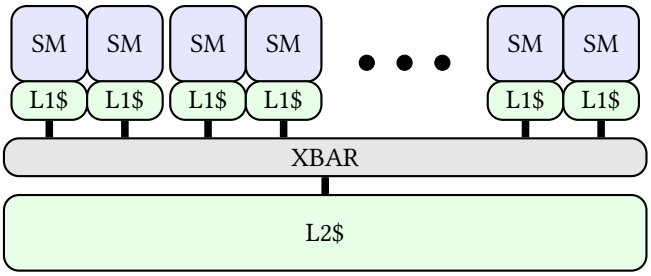


Figure 1. A typical NVIDIA GPU architecture

2.1 GPU Programming and Memory Models

Architecturally, GPUs are organized into a hierarchy of execution units, as shown in Figure 1. From a software perspective, using NVIDIA terminology, GPU execution kernels are subdivided into cooperative thread arrays (CTAs), also known as thread blocks, and CTAs are mapped onto symmetric multiprocessors (SMs) by the GPU scheduler. Each CTA runs within a single SM, but the various CTAs of a kernel are distributed across SMs in order to maximize utilization.

GPUs have traditionally supported a bulk-synchronous programming model. Threads in the same CTA were able to synchronize via execution barriers, but global memory communication between different CTAs in the same kernel was disallowed, in part due to the lack of a rigorous memory model. Communication between kernels was permitted by completing all CTAs in the producer kernel(s) before launching any CTAs of the consumer kernel(s), as the kernel boundary implicitly provided the necessary memory synchronization semantics.

More recently, NVIDIA GPUs have shifted away from a bulk-synchronous model towards a more traditional shared memory programming model. NVIDIA GPUs have exposed a Unified Memory (i.e., unified virtual address space) abstraction since the Kepler generation of architectures [25]. The Volta generation pushes this abstraction even farther: it has *Independent Thread Scheduling*, each thread is treated as an independent unit of execution, as well as explicit forward progress guarantees that explicitly enable programmers to write starvation-free algorithms [41]. Even warp-synchronous programming, the assumption that threads executing as part of the same warp (SIMT grouping of threads) are implicitly synchronized at every instruction, has been obsoleted as of Volta. Instead, threads must now synchronize

using memory synchronization primitives, execution barriers, and/or the new Cooperative Groups API [26], where the latter two have implicit memory synchronization semantics as well. Because of this increase in communication flexibility, the memory model is taking on a more critical role in new GPU architectures.

A feature of many GPU-specific memory models, but not all [49], is the notion of *scope*. For example, at the programming language layer, NVIDIA CUDA and PTX define three levels of scope: CTA, GPU, and System [39, 40]. Scopes capture the idea that programmers will often organize their inter-thread communication patterns such that a thread will only need to communicate with its nearest neighbors (e.g., within the same CTA). Hardware can take advantage of this knowledge by only propagating synchronization mechanisms to the boundaries of a user-specified scope, thereby improving communication throughput and/or latency.

NVIDIA GPUs since Kepler have occasionally had memory model issues [6, 51, 58]. The analysis in this paper aims to place NVIDIA GPU architectures starting from Volta and the PTX ISA from version 6.0 on a solid and more reliable theoretical foundation.

2.2 Axiomatic Memory Models

Two techniques are predominant in the study of memory models today: the axiomatic approach and the operational approach. In the former, candidate executions are described in terms of relations between the primitive events (e.g., reads, writes, fences) in the program. A candidate execution is legal only if it satisfies all of the *axioms* in the memory model. In the latter, legal outcomes are those which can be produced by executions of an abstracted golden architectural model. Ideally, the various ways of expressing any given model will be proven equivalent, leaving the preference up to the user. Such equivalence proofs have been developed for several important memory models in use today [9, 44]. In this work, we choose to use an axiomatic approach, primarily because it matches the PTX documentation [40].

The literature on axiomatic models has converged on a relatively standard set of common relations. Many models add one-or-more model-specific relations as well. To help build intuition, we explain the common notation using the well-known total store ordering (TSO) memory model [53]. A formal axiomatic specification of TSO is presented in Figure 2. It has two axioms: a sequential consistency per location axiom and a causality axiom.

The TSO SC-per-Location axiom states that all accesses to any individual memory address will always appear to settle into some total order at runtime, regardless of any synchronization that is or is not present. Three relations (*rf*, *co*, and *fr*) in this axiom describe communication. The reads-from (*rf*) relation relates every write to the set of reads that return the value stored by that write. In other words, a $\xrightarrow{rf}$ b if and only if *b* returns the value written by *a*. The coherence order (*co*) relation is a total order over the writes to each address. This is similar to, but more general than, a single-writer/multiple-readers requirement often used to describe cache coherence protocols [52]. From-reads (*fr*) relates each read to the coherence successors of the write it reads from; i.e., $fr := rf^{-1}; co$. Here, ";" is a relational join: if b $\xrightarrow{rf}$ a and b $\xrightarrow{co}$ c, then a $\xrightarrow{rf^{-1};co}$ c. The last relation in the SC-per-Location axiom is the "program order, same location" relation (*po_loc*). Program order (*po*) is the order in which instructions are originally laid out in a program's binary. By convention, this does not consider a control flow graph with loops, back-edges, etc., but instead considers only the fully unrolled straight-line execution of a program. *po_loc* is the restriction of *po* to memory accesses with the same address.

TSO definition of *ppo*

	to Load	to Store
from Load	Y	Y
from Store	-	Y

"Y" = always enforced

Axiom	Definition
SC-per-Loc.	acyclic($rf \cup co \cup fr \cup po\_loc$)
Causality	acyclic($rfe \cup co \cup fr \cup ppo \cup fence$)

Figure 2. An axiomatic definition of TSO

The TSO Causality axiom describes memory ordering rules in the presence of per-thread store buffers. First of all, store buffers can cause loads to be reordered before earlier stores from the same thread. Preserved program order (*ppo*) for TSO captures the subset of *po* that excludes store-to-load ordering. The reads-from external (*rfe*) relation is the subset of *rf* that relates events from different threads. Store buffer forwarding results in a load acquiring its value before the store it forwards from becomes visible to other threads. This implies that other threads observe the load to have occurred before the store, meaning that the intra-thread subset of *rf* cannot be used to enforce memory ordering. Finally, the *fence* relation relates pairs of events in the same thread if they are separated by a fence instruction or if at least one is an atomic read-modify-write operation.

Weak memory models are generally more complex to formulate than the TSO formalization in Figure 2. Although many of the relations described above (*po*, *po_loc*, *rf*, *rfe*, *co*, *fr*) now have standard definitions across a wide range of models, they may take on non-standard and/or architecture-specific meaning in some cases. For example, our formalization of the PTX memory model uses *po*, *po_loc*, *rf*, and *fr* exactly as defined above for TSO, it adds a PTX-specific twist to *co* (described later in Section 3.5.1), and it does not use *ppo* at all.

```
ld{.weak}{.ss}{.cop}{.vec}.type d, [a];
ld.volatile{.ss}{.vec}.type d, [a];
ld.relaxed.scope{.ss}{.vec}.type d, [a];
ld.acquire.scope{.ss}{.vec}.type d, [a];

.ss = {.const, .global, .local, .param, .shared};
.cop = {.ca, .cg, .cs, .lu, .cv};
.scope = {.cta, .gpu, .sys};
.vec = {.v2, .v4};
.type = {.b8, .b16, .b32, .b64, .u8, .u16, .u32,
         .u64, .s8, .s16, .s32, .s64, .f32, .f64};
```

(a) ld instruction

```
st{.weak}{.ss}{.cop}{.vec}.type [a], b;
st.volatile{.ss}{.vec}.type [a], b;
st.relaxed.scope{.ss}{.vec}.scope.type [a], b;
st.release.scope{.ss}{.vec}.scope.type [a], b;

.ss = {.global, .local, .param, .shared};
.cop = {.wb, .cg, .cs, .wt};
.scope = {.cta, .gpu, .sys};
.vec = {.v2, .v4};
.type = {.b8, .b16, .b32, .b64, .u8, .u16, .u32,
         .u64, .s8, .s16, .s32, .s64, .f32, .f64};
```

(b) st instruction

```
fence{.sem}.scope;

.sem = {.sc, .acq_rel};
.scope = {.cta, .gpu, .sys};
```

(c) fence instruction. membar is a synonym for fence.sc.

Figure 3. PTX memory instructions. We explicitly model the highlighted portions. The memory model is agnostic to the non-highlighted portions other than .type.

3 Formalizing the PTX Memory Model

This section presents our derivation of a formal axiomatic memory model specification from PTX documentation [40]. The PTX specification itself is written only in plain English, but we attempt to follow the documentation as written as closely as possible. Where appropriate, we quote the relevant text from Sections 8 and 9 of the PTX specification. A full line-by-line derivation of this formalization is available in our supplemental material [34].

3.1 PTX Instruction Set

Figures 3a and 3b show the definition of the ld and st instructions from §9.7.8.7 of the PTX documentation. Each is qualified as .weak, .relaxed, .acquire, or .release. All but .weak also have a .scope qualifier of .cta, .gpu, or .sys, as defined in Table 1. The atom (i.e., atomic read-modify-write) and red (reduction, i.e., an atom that does not return a value) instructions are defined similarly, but these can also take .acq_rel as a qualifier. Figure 3c presents the fence instruction, which similarly contains .sem semantic ordering and .scope qualifiers. Formalizing these ordering and scope semantics are the primary focus of this paper. We omit .type, .vec, .ss, .cop, and .volatile; see Section 3.6.

Scope	Threads included
.cta	Threads in the same cooperative thread array
.gpu	Threads on the same compute device, "includ[ing] other kernel grids invoked [...] on the same compute device"
.sys	All threads in the current program, "including all kernel grids invoked by the host program on all compute devices, and all threads constituting the host program itself."

Table 1. Scopes; abbreviated presentation of Table 18 from the PTX documentation [40]

3.2 Overlapping Memory Accesses

According to the model, "[t]wo memory locations are said to overlap when the starting address of one location is within the range of bytes constituting the other location. Two memory operations are said to overlap when the corresponding memory locations overlap" (§8.2.1). This terminology is intended to account for memory accesses having different widths, but like most other hardware memory models today, the behavior of mixed-width programs is not fully described (see Section 3.6). As such, in this paper, "overlapping" can be considered a synonym for "accessing the same address".

3.3 Scope, Strength, and Moral Strength

PTX captures the effect of .scope qualifiers as follows. A *strong* operation is "[a] fence operation, or a memory operation with a .relaxed, .acquire, .release, [or] .acq_rel [...] qualifier" (§8.4). Then, "[t]wo operations are said to be *morally strong* relative to each other if the operations are related in program order [...] or each operation is strong and specifies a scope that includes the thread executing the other operation, and if both are memory operations then they overlap" (§8.6). Broadly speaking, only morally strong pairs of operations may be used to synchronize between threads.

The use of scope in PTX is broadly similar to its use in HSA [7, 22] and HRF-indirect [27]. For comparison, the HRF-Direct model requires the two scopes in question to be identical, not simply inclusive. The authors of DeNovo have suggested that GPU memory models do not need scopes at all [49]. However, DeNovo requires a (lightweight) coherence protocol in the L2 cache, and it also depends fundamentally on a data race freedom assumption.

Crucially, unlike all of the existing models described above, PTX gives well-defined semantics to racy programs. In PTX, "[t]wo overlapping memory operations [...] conflict when at least one of them is a write. Two conflicting memory operations are [...] in a data-race if they are not related in causality

$$
\begin{aligned}
pattern_{rel} &:= ([W^{\geq REL}]; po\_loc^{?}; [W]) \cup ([F^{REL}]; po; [W]) \\
obs &:= (morally\_strong \cap rf) \cup (obs; rmw; obs) \\
pattern_{acq} &:= ([R]; po\_loc^{?}; [R^{\geq ACQ}]) \cup ([R]; po; [F^{ACQ}]) \\
sw &:= (morally\_strong \cap (pattern_{rel}; obs; pattern_{acq})) \\
&\quad \cup sync_{barrier} \cup sc \\
cause_{base} &:= (po^{?}; sw; po^{?})^{+} \\
cause &:= cause_{base} \cup (obs; (cause_{base} \cup po\_loc))
\end{aligned}
$$

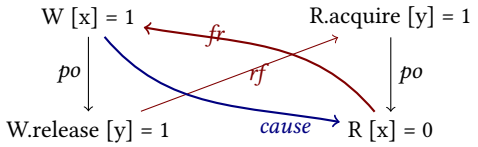
Figure 4. PTX Memory Model Relations

Kernel 0

CTA 0, Thread 0	CTA 1, Thread 1
`st.weak [x], 1`	`ld.acquire.gpu %r1, [y]`
`st.release.gpu %r1, [y]`	`ld.weak %r2, [x]`

Forbidden Final Outcome:
`%r1==1, %r2==0`

(a) Test assembly



(b) Axiomatic analysis

Figure 5. Message passing (MP) using acquire/release synchronization. In all litmus tests in this paper, all of memory is assumed to be initialized to zero at the start of the program.

order and they are not morally strong" (§8.6.1). Properly synchronized and "[c]onflicting morally strong operations are performed with single-copy atomicity" (§8.9.3), i.e., each operation is performed either entirely before or entirely after the other. Racy operations carry no such guarantee. For example, in the presence of a race, one operation may see another in a partially-completed state. Informally, proper synchronization and moral strength rules remain in effect where they are applicable, but as models with mixed-size accesses remain an open area of research (see Section 3.6), the such behaviors remain formally undefined.

3.4 Ordering and Synchronization Relations

The PTX model is not multi-copy atomic. Informally speaking, this means store values may become visible to some threads (e.g., to threads sharing the same L1 cache as the issuing thread) before others (e.g., to threads not sharing the L1 cache). More formally, it means that *rfe*, *co*, and *fr* alone cannot be used for synchronization, as they could under the multi-copy atomic TSO model (Section 2.2). Instead, PTX requires programmers to insert one of the three types of synchronization shown in Figure 4: via release/acquire pairs, via CTA execution barriers, or via `fence.sc` ordering.

3.4.1 Release Consistency

The first type of synchronization, a form of release consistency, is provided by the synchronizes-with (*sw*) relation. It is broken into three components: a release pattern ($pattern_{rel}$), an observation (*obs*), and an acquire pattern ($pattern_{acq}$). A *release pattern* on a location M consists of "[a] release operation on M, or a release operation on M followed by a strong write on M in program order, or a fence followed by a strong write on M in program order" (§8.7). In other words, intuitively, the pattern allows the release operation itself to be optionally decoupled from the write used to communicate that synchronization to other threads. An *acquire pattern* on a location M is defined similarly, but for the consumer: it "consists of an acquire operation on M, or a strong read on M followed by an acquire operation on M in program order, or a strong read on M followed by a fence in program order" (§8.7). In between is an *observation* sequence: "a write W precedes a read R in observation order (*obs*) if R and W are morally strong and R reads the value written by W, or for some atomic operation Z, W precedes Z and Z precedes R in observation order" (§8.8.2). Observation indicates that the consumer received the signal from the producer. The optional inclusion of read-modify-write operations ensures the implementability of C/C++ release sequences [28]. Figure 5 shows a typical use of release-acquire synchronization.

3.4.2 Barrier Synchronization

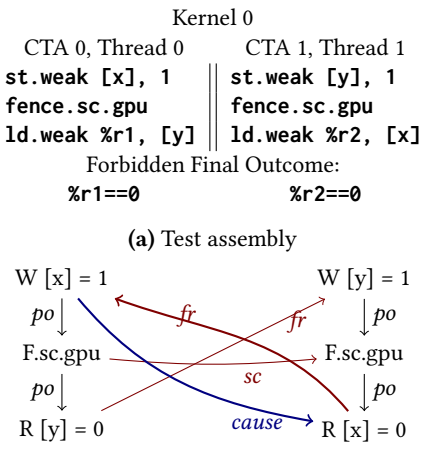
The second form of synchronization is via CTA execution barriers: "a `bar.sync` or `bar.red` or `bar.arrive` operation synchronizes with a `bar.sync` or `bar.red` operation executed on the same barrier" (§8.8.4). Barrier synchronization has the same effect as release-acquire synchronization performed at `.cta` scope.

3.4.3 Fence-SC Order

The third form of synchronization is Fence-SC order (*sc*), defined as "an acyclic partial order, determined at runtime, that relates every pair of morally strong `fence.sc` operations" (§8.8.3). Morally weak fences may be related by *sc* order due to transitivity, but they may also be unrelated in *sc*. The Fence-SC order can prevent weak memory ordering behaviors that acquire/release alone cannot prevent, such as the well-known store buffering (SB) pattern of Figure 6. The introduction of `fence.sc` in the newest generation of PTX corrects the weak SB behavior seen with `membar` in previous NVIDIA GPU architectures [51] (§9.7.12.3).

3.4.4 Causality Order

Synchronization order pairs with program order to form the transitive *base causality* relation ($cause_{base}$): "[a]n operation X precedes an operation Y in base causality order if X synchronizes with Y, or for some operation Z,



(a) Test assembly

(b) Axiomatic analysis

Figure 6. Preventing the non-sequentially-consistent outcome of a store buffering (SB) pattern requires a `fence.sc` to be placed between the memory operations in each thread. PTX also requires the two fences to be morally strong.

1. X precedes Z in program order and Z precedes Y in base causality order, or
2. X precedes Z in base causality order and Z precedes Y in program order, or
3. X precedes Z in base causality order and Z precedes Y in base causality order" (§8.8.5).

The recursion ensures that synchronization composes transitively, e.g., from producer to intermediate hop to consumer. It also enforces the *cumulativity* property that is similarly used to define transitive synchronization on other architectures [9]. The *causality* relation (*cause*) extends base causality ordering to certain same-address relationships occurring before or after: "[a]n operation X precedes an operation Y in causality order if X precedes Y in base causality order, or for some operation Z, X precedes Z in observation order, and

1. Z precedes Y in base causality order, or
2. Z precedes Y in program order, and Z and Y overlap" (§8.8.5)

3.5 Top-Level Memory Model Axioms

Having discussed the relations within the model, we now describe the axioms that use those relations to determine whether a candidate execution is legal. Figure 7 presents our formalization of the six main PTX memory model axioms.

3.5.1 Coherence

Although CPU memory models (including but not limited to TSO) define coherence order as a total order over all stores to each address, in PTX the coherence order (*co*) is defined as "a partial transitive order that relates overlapping write operations, determined at runtime," such that "[t]wo overlapping write operations are related in coherence order if

Axiom 1 (Coherence).
$[W]; cause; [W] \subseteq co$

Axiom 2 (FenceSC).
$irreflexive(sc; cause)$

Axiom 3 (Atomicity).
$empty(((morally\_strong \cap fr);(morally\_strong \cap co)) \cap rmw)$

Axiom 4 (No-Thin-Air).
$acyclic(rf \cup dep)$

Axiom 5 (SC-per-Location).
$acyclic((morally\_strong \cap (rf \cup co \cup fr)) \cup po\_loc)$

Axiom 6 (Causality).
$irreflexive((rf \cup fr); cause)$

Figure 7. PTX Memory Model Axioms

they are morally strong or if they are related in causality order" (§8.8.6). The PTX Coherence axiom covers the latter: "if a write W precedes an overlapping write W' in causality order, then W must precede W' in coherence order" (§8.9.1). In Figure 7, the notation "$[s]$" means $(s \times s) \cap iden$, where *iden* is the identity function relating every operation to itself. This has the effect of restricting relation chains to those passing through operations in the set s at the specified spot.

As a corollary, overlapping morally weak stores are not related by *co* if they are not causally related. Such a scenario would be considered a data race. By not forcing racy stores into coherence order, the model frees the implementation from the burden of keeping memory accesses coherent with those outside the scopes of the operations being performed.

3.5.2 Fence-SC

Although the Fence-SC order (*sc*) is only determined dynamically at runtime, it can be constrained into certain orders via other synchronization in the program, as is done in Figure 6. The Fence-SC Axiom simply states that "Fence-SC order cannot contradict causality order" (§8.9.2).

3.5.3 Atomicity

The Atomicity Axiom defines the atomicity of read-modify-write instructions: "when an atomic operation A and a write W overlap and are morally strong, then the following two communications cannot both exist in the same execution:

- A reads any byte from a write W' that precedes W in coherence order.
- A follows W in coherence order" (§8.9.3)

The first bullet translates to an *fr* relation, and the second translates to *co*. In our model (which for modeling purposes splits `atom` into separate read and write components [32]),

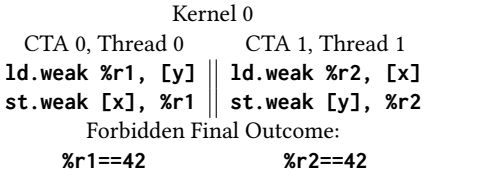


Figure 8. Without a No-Thin-Air axiom, nothing would prevent each loads from speculating a return value of 42 and then using the other's speculation to justify its own, even though the value 42 would never be produced otherwise. The value 42 would have appeared "out of thin air".

that combination is not permitted to intersect the *rmw* relation that connects the two split parts of an `atom`.

Just as with the Coherence axiom, atomic read-modify-write operations need not actually be kept atomic with respect to accesses with which they are morally weak.

3.5.4 No-Thin-Air

The No-Thin-Air Axiom prevents values from appearing "out of thin air" [18]: "an execution cannot speculatively produce a value in such a way that the speculation becomes self-satisfying through chains of instruction dependencies and inter-thread communication" (§8.9.4) as in Figure 8.

3.5.5 SC-per-Location

The SC-per-Location Axiom states that "morally strong [...] communication order cannot contradict program order" (§8.9.5). As seen for TSO in Section 2.2, this is a standard axiom enforcing sane behavior for single-threaded programs and for coherence litmus tests such as those in Figure 9, with the added caveat that such enforcement again only applies to morally strong operations.

3.5.6 Causality

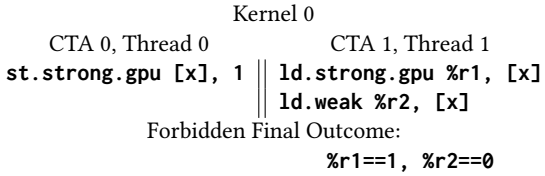
The Causality Axiom states that "relations in communication order cannot contradict causality order [...]:

1. If a read R precedes an overlapping write W in causality order, then R cannot read from W.
2. If a write W precedes an overlapping read R in causality order, then for any byte accessed by both R and W, R cannot read from any write W' that precedes W in coherence order" (§8.9.6)

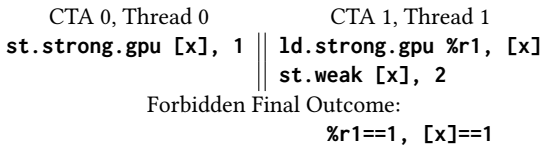
This axiom ensures that communication via memory respects user-inserted acquire/release, barrier, and/or `fence.sc` synchronization, as was used in Figure 5.

3.6 Omitted Qualifiers

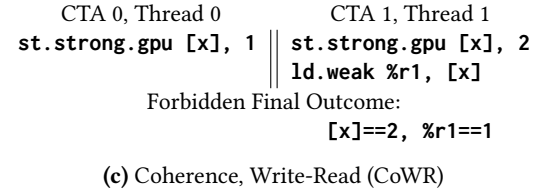
Per PTX documentation, the non-highlighted portions of Figure 3 do not affect the memory model rules, and so we do not explicitly model them.



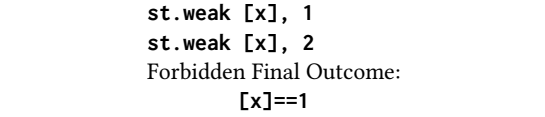
(a) Coherence, Read-Read (CoRR)



(b) Coherence, Read-Write (CoRW)



(c) Coherence, Write-Read (CoWR)



(d) Coherence, Write-Write (CoWW)

Figure 9. Standard coherence litmus tests

Vector accesses (`.vec`) "are modelled as a set of equivalent memory operations with a scalar data-type, executed in an unspecified order on the elements in the vector" (§8.2.2). In prior PTX generations, cache operators (`.cop`) such as `.ca` (cache at all levels) or `.cg` (cache in the L2 cache and beyond) were microarchitecture-specific methods of enforcing consistency [51]. In PTX 6.0, cache operators "are treated as performance hints only. The use of a cache operator [...] does not change the memory consistency behavior of the program" (§9.7.8.1), showing that PTX 6.0 has adopted a more rigorous and modern model. A `.volatile` "operation is always performed" and "has the same memory synchronization semantics as `ld.relaxed.sys`" (§9.7.8.7).

The `.ss` qualifier describes the *state space* being accessed: constant memory, global memory, local memory, parameter memory, or the "shared memory" scratchpad. OpenCL considers synchronization through local memory and through global memory to be two independent relations [31], but this was shown to be problematic [13]. PTX avoids this problem by stating that "the relations defined in the memory consistency model are independent of state spaces" (§8.3).

Event	Legal `memory_order` arguments					
	NA	RLX	ACQ	REL	ACQREL	SC
Read	X	X	X			X
Write	X	X		X		X
Fence			X	X	X	X

(a) Basic Primitives. The `memory_order` set is ordered from left to right, except that `ACQ` and `REL` are not comparable.

$$
\begin{array}{rcl}
sb & := & \text{partial order analog of program order} \\
sb|_{loc} & := & sb \cap (\text{accessing same address}) \\
sb|_{\neq loc} & := & sb - sb|_{loc} \\
mo & := & \text{total order over atomic writes to each address} \\
\\
rb & := & rf^{-1}; mo \\
eco & := & (rf \cup mo \cup rb)^{+} \\
rs & := & [\mathrm{W}]; sb|_{loc}{}^{?}; [\mathrm{W}^{\geq RLX}]; \\
 & & ((\boxed{incl} \cap rf); rmw)^{*} \\
sw & := & [\textsc{Event}^{\geq REL}]; ([\mathrm{F}]; sb)^{?}; \\
 & & rs; (\boxed{incl} \cap rf); [\mathrm{R}^{\geq RLX}]; \\
 & & (sb; [\mathrm{F}])^{?}; [\textsc{Event}^{\geq ACQ}] \\
hb & := & (sb \cup (\boxed{incl} \cap sw))^{+} \\
\\
scb & := & sb \cup (sb|_{\neq loc}; hb; sb|_{\neq loc}) \cup hb|_{loc} \cup mo \cup rb \\
psc_{base} & := & ([\textsc{Event}^{SC}] \cup [\mathrm{F}^{SC}]; hb^{?}); scb; \\
 & & ([\textsc{Event}^{SC}] \cup hb^{?}; [\mathrm{F}^{SC}]) \\
psc_{F} & := & [\mathrm{F}^{SC}]; (hb \cup hb; eco; hb); [\mathrm{F}^{SC}] \\
psc & := & psc_{base} \cup psc_{F}
\end{array}
$$

(b) Important Relations

Axiom	Definition
Coherence	irreflexive(hb; $eco^{?}$)
Atomicity	no $rmw \cap (rb; mo)$
SC	acyclic($\boxed{incl} \cap psc$)
No-Thin-Air	acyclic($sb \cup rf$)

(c) RC11 Axioms

Figure 10. RC11 [32], modified to account for scoping [58]. The $\boxed{incl}$ relation applies only in the scoped model: it relates pairs of events with mutually inclusive scopes.

Finally, the `.type` qualifier specifies the type and width of the memory access. Mixed-width models are new and not yet well-understood [21]. Since "[t]he axioms in the memory consistency model do not apply if a PTX program contains one or more mixed-size data-races" (§8.6.2), we do not attempt to solve the mixed-size problem in this paper.

4 Mapping "Scoped C++" onto PTX

One important requirement for any memory model is the ability to reliably serve as a target for higher-level memory models (e.g., from CUDA or OpenCL) that will be compiled onto it. One major benefit of having a formal memory model specification for PTX is that it enables formal verification of such a proposed mapping.

Because we are modeling PTX, it would be a natural fit to use NVIDIA's CUDA as a source programming language. However, CUDA does not yet have an officially-sanctioned formal memory consistency model. Unfortunately, the OpenCL 2.2 standard also has known issues and is derived from a version of C/C++ that is unsound with respect to canonical compiler mappings onto many architectures [13, 31, 32, 36]. The recent OpenCL formalization by Wickerson et al. [58] is derived from a version of C++ which is slightly more up-to-date than the OpenCL standard, but it is not up to date with developments that have occurred since that paper was published [32, 36]. Furthermore, Wickerson et al. only consider a subset of the full OpenCL scope hierarchy.

In light of these limitations, we choose the "Repaired C11" (RC11) model formalized by Lavav et al. [32] as a starting point to define a new OpenCL-like scoped C++ memory model to map onto PTX. To our knowledge, the RC11 model is the most up-to-date formalization of the C++ model and is the basis for a future revision that is expected to be incorporated into ISO C++ [19]. In this section, we first derive a new scoped C++ memory model from RC11. We then present a mapping of synchronization primitives in this scoped C++ model onto the PTX model. Later, we empirically test and then formally prove the correctness of this mapping.

4.1 A Scope-Extended RC11 Memory Model

Like Wickerson et al. [58], we convert RC11 into a reasonable OpenCL-like scoped C++ model by requiring any inter-thread communication that is used for synchronization to be done via *scope-inclusive* (*incl*) accesses. In this sense, it serves broadly the same purpose as moral strength in PTX.

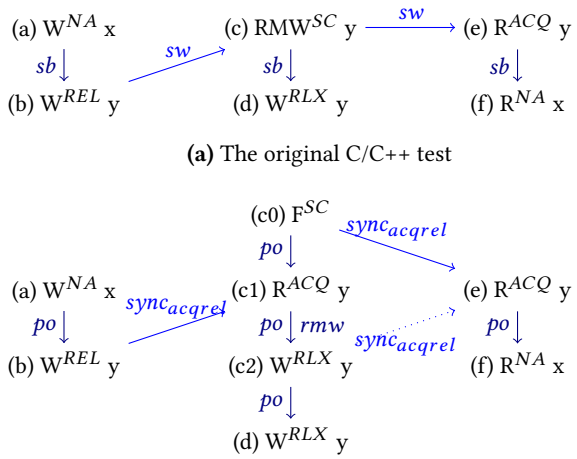
Figure 10 summarizes the RC11 memory model axioms and primitives. For space reasons, we omit a full explanation and instead refer interested readers to the original source for details [32]. We make only two changes to RC11. First, to introduce scopes, we add *incl* as shown in Figure 10. Second, we exclude the RC11 No-Thin-Air axiom, as its proposal to forbid all load-to-store ordering for atomic operations remains controversial, and because it contradicts current GPU behavior. Meanwhile, the out-of-thin-air problem remains an active area of research [43, 50].

4.2 A Mapping from Scoped C++ onto PTX

Our mapping from scoped C++ to PTX turns out to be relatively straightforward, as shown in Figure 11. We also assume straightforward mappings of *sb*, memory locations, and scopes in C++ onto *po*, memory addresses, and scopes in PTX, respectively. The full details are available in our supplemental material [34].

RC11 construct	PTX mapping
$R^{NA,sco}$	`ld.weak`
$R^{RLX,sco}$	`ld.relaxed.`*<sco>*
$R^{ACQ,sco}$	`ld.acquire.`*<sco>*
$R^{SC,sco}$	`fence.sc.`*<sco>*`; ld.acquire.`*<sco>*
$W^{NA,sco}$	`st.weak`
$W^{RLX,sco}$	`st.relaxed`*<sco>*
$W^{REL,sco}$	`st.release.`*<sco>*
$W^{SC,sco}$	`fence.sc.`*<sco>*`; st.release.`*<sco>*
$RMW^{RLX,sco}$	`atom.`*<sco>*
$RMW^{ACQ,sco}$	`atom.acquire.`*<sco>*
$RMW^{REL,sco}$	`atom.release.`*<sco>*
$RMW^{ACQREL,sco}$	`atom.acq_rel.`*<sco>*
$RMW^{SC,sco}$	`fence.sc.`*<sco>*`; atom.acq_rel.`*<sco>*
$F^{ACQ,sco}$	`fence.acquire.`*<sco>*
$F^{REL,sco}$	`fence.release.`*<sco>*
$F^{ACQREL,sco}$	`fence.acq_rel.`*<sco>*
$F^{SC,sco}$	`fence.sc.`*<sco>*

Figure 11. Our mapping from C/C++ to PTX



(a) The original C/C++ test

(b) The test compiled for PTX, without the release annotation on the RMW^{SC}. Note the gap between the $sync_{acqrel}$ edges.

Figure 12. A variant of the ISA2 litmus test, used here to analyze the mapping of RMW(`memory_order_seq_cst`).

Notably, although PTX 6.0 has native acquire and release operations, it does not have native sequentially-consistent read and write operations. The benefit of native SC operations would become apparent if the underlying hardware ISA were to support a fine-grained mechanism for keeping such operations ordered with each other. Lacking this, we simply use a standard leading-fence mapping for those operations.

One particular mapping required extra attention: `.release` annotations are not redundant with a leading `fence.sc`, even though they may seem to be. Figure 12 provides an example: RC11 considers (d) to be part of the release sequence (*rs*) headed by (c), so (c) synchronizes with (e) even when (e) reads from (d) rather than from (c). Combining this with happens-before relationships from (a) to (c) and from (e) to (f), we conclude that (a) happens before (f), which implies that (f) must return the value written by (a).

Now, suppose we compile (c) to the sequence `fence.sc.sco; atom.acquire.sco`, i.e., eliding the `.release`. This produces the execution in Figure 12b. Here, there is a gap between the $sync_{acqrel}$ edge ending at (c1) and the $sync_{acqrel}$ edge starting at (c0). This breaks the expected *cause* relationship between (a) and (f), leading to a failure to maintain RC11 requirements. If, on the other hand, the RMW^{SC} mapping keeps the `.release`, then the dotted $sync_{acqrel}$ edge starting at (c2) will also be enforced, closing the gap.

Notably, this example pushes the limits of what can be tested empirically (see Section 6.1). We caught this corner case only with Coq, not with Alloy. This anecdote highlights an important benefit of combining empirical testing with formal verification in a single workflow.

5 Analyzing PTX Using Alloy and Coq

In this section, we describe the Alloy domain-specific language (DSL) and its use in describing axiomatic memory models. The use of Alloy allows us to empirically test our scoped C++ mapping as well as other properties of the model up to a certain user-defined instance size bound (usually in the single digits [35, 58]). Results of this empirical testing are provided later in Section 6.1. To enable this testing, we present a compiler that converts Alloy models into Coq, an interactive theorem prover, and then we use that tool to build a machine-checked proof that our mapping from scoped C++ onto PTX is sound for programs of *any* size.

5.1 The Alloy Relational Modeling Language

Alloy is a language for describing relational models [30]. The underlying logic for Alloy is a flavor of first-order logic built around the notion of a *relation*: a set of *n*-tuples of primitive "atoms" in the logical universe. Using Kodkod [54], the Alloy tool converts models into SAT formulas and passes them to an off-the-shelf SAT solver. Any instances or counterexamples are translated back into their representation in the Alloy model and presented to the user.

In the Alloy DSL, a **`sig`** ("type Signature") is a collection of atoms. Each **`sig`** may be decomposed into one or more disjoint subsigs which partly or entirely comprise the parent **`sig`**'s atoms. For example, we define **`Event`** as a primitive **`sig`**, **`Fence`** and **`MemoryEvent`** as subsigs of **`Event`**, and **`Read`** and **`Write`** as subsigs of **`MemoryEvent`**. Relations are defined in terms of a domain **`sig`** and a range relation. For example, we define **`po`** as a relation from **`Event`** to **`Event`**, and **`rf`** as a relation from **`Write`** to **`Read`**.

Definition (Causality order).
$cause := cause_{base} \cup (obs; (cause_{base} \cup po\_loc))$

Axiom (Causality). irreflexive$((rf \cup fr); cause)$

(a) Mathematical formulation

```
fun cause : Event->Event {
  cause_base + observation.(cause_base + po_loc) }
pred irreflexive[r: Event->Event] { no iden & r }
pred causality { irreflexive[(rf + fr).cause] }
```

(b) Alloy formulation

Figure 13. Using Alloy to encode axiomatic memory models

```
// the scope relation forms a tree
sig Scope { subscope: set Scope }
fact { subscope.~subscope in iden }
fact { acyclic[subscope] }

// there is exactly one root Scope, called System
fun System : Scope { Scope - Scope.subscope }
fact { one System }

// the leaf Scopes are called Threads
sig Thread extends Scope { start: one Event }
fact { no Thread.subscope }

// every Event has one associated Scope
abstract sig Event { po: lone Event, scope: one
    Scope }

// every Event must be contained within its own
    scope
fact { scope in *~po.~start.*~subscope }
```

Figure 14. Using Alloy to encode a generic *scope* tree

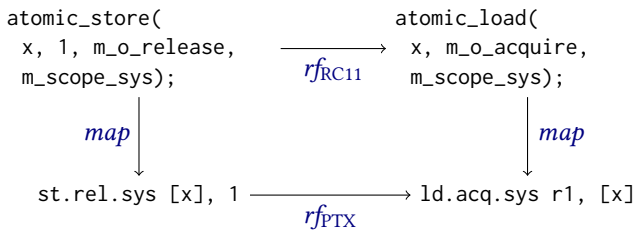


Figure 15. Using the *map* relation to describe mappings from scoped C++ events to PTX events

5.2 Formalizing the PTX Memory Model in Alloy

Alloy's flexible DSL make it very easy to analyze axiomatic memory models [35, 58]. Figure 13 provides an example of how the definitions and axioms from Section 3 can be converted into Alloy in a straightforward mechanical way.

In order to empirically test our scoped C++ to PTX mapping, we also model the mapping itself in Alloy, in conjunction with the respective Alloy models for scoped C++ and PTX. The key element in this combined Alloy model is a new *map* relation describing the mapping from each scoped C++ event onto corresponding PTX event(s). For convenience, we say that a scoped C++ relation r *lowers* to a PTX relation r' if $r \in map; r'; map^{-1}$. This notation is shown in Figure 15. The statement of correctness for the mapping is as follows:

Theorem. Given a valid race-free scoped C/C++ program p, suppose p is compiled onto PTX program p' according to the rules of Section 4, and suppose e' is some legal execution of p'. If we interpret e' as an execution e of the original C/C++ program, then e is a legal execution of p.

Here, e' is interpreted as an execution e of the original RC11 program by assuming that $rf_{\text{PTX}} \subseteq map^{-1}; rf_{\text{RC11}}; map$. In other words, we lift the return values of PTX loads (i.e., rf_{PTX}) up to their scoped C++ equivalents, as this captures the essence of interepreting the PTX execution as an RC11 execution, and follows standard practice for this type of proof [14]. We also assume that $co \subseteq map^{-1}; mo; map$ and $fr \subseteq map^{-1}; rb; map$ This choice is not necessary but turns out to be sufficient to justify the correctness of the mapping.

5.3 Analyzing the PTX Memory Model Using Coq

Here, we describe our Alloy-to-Coq compiler `alloqc`. By compiling our Alloy models into Coq, we can prove formally that our scoped C++ to PTX mapping is valid for programs of any size. These proofs are presented in Section 6.2.

We generated our Coq model by developing an Alloy-to-Coq compiler that we call `alloqc`. The basic principles are similar to prior theoretical work targeting provers such as PVS, KeY, or Athena [10, 38, 56], but Coq has a number of benefits over other solvers [20]. The compiler itself aims to support any general Alloy model; it is not specific to memory models. However, our primary use case in this paper is verification of our mapping from scoped C++ onto PTX. Figure 16 provides a cartoon example of our compiler in action. `alloqc` takes as input any Alloy model (Figure 16a) and produces as output a Coq file containing a translation of each construct in the Alloy model (Figure 16b). The user can then fill in the Coq proofs (Figure 16c).

In an effort to keep components modular, we decouple our Coq implementation of Alloy's relational logic from our generation of models built on top of that logic. To do this, we encode an implementation of Alloy's logic into a stand-alone Coq library `alloy.v`. All Coq files emitted by `alloqc` are built on top of the `alloy.v` library, as shown in Figure 16b. `alloy.v` includes only one external library: the Coq standard library `Eqdep_dec`, which proves the equality of all identity proofs over decidable types (such as `Tuple n`). We use this as part of a convenience layer to spare users from getting

```
check my_lemma { iden in univ->univ } for 3
```

(a) my_model.als (written by user).

```
Require Import alloy.

Definition my_lemma_statement : Prop :=
  forall _i, (inside
    (arrow (m:=0) (n:=0) univ univ) iden).
```

(b) my_model.v (automatically generated by `alloqc`)

```
Require Import alloy.
Require Import alloy_util.
Require Import my_model.

Theorem my_lemma : my_lemma_statement.
Proof.
  intros _i [x y] H.

  (* if (x, y) is in iden, then x = y *)
  unfold_ident. (* a library tactic *)

  (* need to show (y, y) is in univ->univ *)
  apply arrow_split; (* a library lemma *)
  unfold univ; auto. (* ∀y, univ y is True  *)
Qed.
```

(c) proofs.v (written by user). These apply to instances of any size.

Figure 16. Overview of the `alloqc` flow. An assertion written in Alloy can be empirically tested (a), then compiled into a Coq lemma (b) for which the user can fill in a proof (c).

caught in Alloy's strict type system. For example, we include functions to cast between **`Tuple m+n`** and **`Tuple n+m`**.

Most of our Alloy-to-Coq mappings are straightforward; however, the transitive closure is a notable exception. In Kodkod, the transitive closure (which cannot in general be calculated using finite first-order logic) is calculated by iterating $r = r \cup r.r$ enough times to cover the upper bound on the size of the relation. For `alloy.v`, we encode transitive closure as an inductive relation in order ensure generality.

6 Results of Testing and Verifying the Mapping from "Scoped C++" to PTX

We now describe our empirical testing of the Section 4 mapping from scoped C++ to PTX using the toolflow described in Section 5. The goal is to prove that that there are no counterexamples that would render our mapping from scoped C++ onto PTX incorrect, as well as to determine what the upper bound is on the set of behaviors that we can practically analyze using Alloy and its SAT solver backend.

6.1 Empirical Testing Results

We considered two versions of our scoped C++ to PTX mapping. First, we performed the analysis as described in Section 4. Second, for comparison, we also analyzed a "de-scoped" version of the mapping, from RC11 onto a version of PTX with no `.cta` or `.gpu` scope. This comparison allows us to see the verification overhead that the inclusion of scopes incurs. We studied each axiom individually, using the largest event count bound that did not time out after 48 hours. We analyze each of the axioms individually.

The total CPU time taken to perform each check on an Intel Xeon server CPU is shown in Figure 17. The time taken to analyze each axiom varies by orders of magnitude, with the scoped C++ coherence axiom being by far the most expensive. Comparing Figure 17a and 17b clearly shows that the addition of scopes to the model rendered the analysis more expensive by an order of magnitude. This is in line with observations from previous work that memory model analysis time can be superexponential with respect to the model size bound being used for testing [35]. Bounds of five or six events are large enough to cover some of the most important memory consistency model litmus tests for GPUs [51], but unfortunately they do not cover every litmus test of interest. We expect that these bounds could be improved with more aggressive modeling and/or solver techniques.

6.2 Formal Correctness Proofs

Empirical testing is clearly useful, but as we have shown, it is also often limited in its ability to scale to larger problem sizes. As such, we now present an abridged summary of our machine-checked Coq proof that our scoped C++ mapping onto PTX is sound. Although the proof takes additional manual effort to derive, it provides a more comprehensive guarantee that the scoped C++ to PTX mapping is indeed sound for all code. Combined with the empirical testing of the previous section, our analysis demonstrates that scoped C/C++ (and by loose analogy CUDA and OpenCL) can be safely made to target PTX and NVIDIA GPUs. Our complete proof development is available with our supplemental material [34]. The proof itself is approximately 3100 lines of Coq code and checks in approximately 15 seconds.

Theorem 1. RC11 Coherence is satisfied.

Proof. Suppose we have an $hb; eco^?$ cycle. hb lowers either to po or $cause_{base}$, so hb alone cannot be cyclic, because it would violate the PTX Causality and/or SC-per-Location axiom. $hb; eco$ lowers to $(po \cup cause)$; $(rf \cup mo \cup rb)$; $(rf \cap incl)$, and this also violates either SC-per-Location or Causality. □

Theorem 2. RC11 Atomicity is satisfied.

RC11 Axiom	Runtime (Bound = 4 Events)	Runtime (Bound = 5 Events)
Coherence	41s	6.4 hr
Atomicity	4s	5s
SC	10s	15 min

(a) Full models (with scopes)

RC11 Axiom	Runtime (Bound = 5 Events)	Runtime (Bound = 6 Events)
Coherence	1.8 min	3.1 hr
Atomicity	4s	4s
SC	21s	26s

(b) "De-scoped" models (for comparison)

Figure 17. Runtimes to empirically check our scoped C++ to PTX mapping correctness in Alloy.

Proof. Let x and y be the read and write part, respectively, of an RMW, and let m be an intervening store. If m is scope-inclusive with x and y, then PTX Atomicity is violated. If not, then $x \xrightarrow{hb} m \xrightarrow{hb} y$. hb ending at a write must be $hb^?; sb$, giving us $x \xrightarrow{hb} m \xrightarrow{hb^?; sb} y$. Nothing can appear in sb between x and y, so we end up with $x \xrightarrow{hb; hb^?; sb^?} x$, or simply $x \xrightarrow{hb} x$, which leads to a contradiction. □

Theorem 3. RC11 SC is satisfied.

Proof. Following Lahav et al. [32], we pre-convert `m_o_seq_cst` memory events into `m_o_acquire` or `m_o_release` events preceded by `m_o_seq_cst` fences; this has no effect on the mappings of Section 4, which already uses leading fences. With that, both psc_F and psc_{base} are included in the relation $[F^{SC}]; hb^?; (hb \cup eco); hb^?; [F^{SC}]$. The two F^{SC} events are assumed to be scope-inclusive, so they map onto two PTX fences related by sc into an order consistent with psc. However, because psc is assumed cyclic, sc also becomes cyclic, which is a contradiction. □

7 Related Work

A number of GPU models have been proposed in the literature. The HSA specification uses the heterogeneous race-free (HRF) memory model [22, 27], which extends the concept of DRF to programs with explicitly-scoped instructions. OpenCL also extends scopes to the software level, but has had issues in its attempts to do so [13, 31]. The memory model for NVIDIA GPUs has been unofficially formalized by academics in the past [6, 51, 58], but NVIDIA has recently put out an official description of its memory model [40].

A number of practical and popular tools have been developed to complement the theory of modern memory models. `cppmem` provides a bespoke axiomatic C++ memory model [1]. The `diy` suite includes `herd`, a tool for defining and analyzing generic axiomatic models, `litmus`, a tool for black-box testing of hardware, and `diy`, a tool for generating litmus tests [2, 9]. Alloy has also been used for similar purposes [35, 58]. Operational models often come with interactive tools such as `ppcmem` or `rmem` as well [15, 47, 48].

A number of formal verification flows have been developed for memory models over the years. Many use Coq [4] or HOL [5] or a framework such as Lem [3]. Unfortunately, no single tool provides the domain-specific knowledge, the specification flexibility, the ability to perform empirical testing, and the ability to perform rigorous theorem proving all in one package. A key contribution of this paper is the ability of `alloqc` to fill this gap for memory models in particular.

There has been work to port Alloy to interactive theorem provers such as PVS [38], KeY [56], and Athena [10]. Our `alloqc` tool develops a similar flow but targeting the Coq theorem prover with a pragmatic approach aimed primarily at supporting memory model analysis. Previous work has explored mapping Alloy to SMT rather than to SAT [23, 37], but such analyzers are slower for some queries and/or unable to directly analyze all queries. There has also been work to integrate KodKod with theorem proving tools directly [16, 17]. Such integrations, while helpful in mitigating some of the tedium of manual proofs, do not provide the user-friendliness that a front end like Alloy's can provide.

8 Conclusion

Although earlier versions of the NVIDIA GPU memory model were incomplete, our analysis of the newly released PTX 6.0 memory model specification shows that the known issues have been resolved. To confirm the validity of the model, we derive and then perform the first formal axiomatic analysis of this PTX memory model. We also confirm the suitability of PTX as a target for GPU-targeted programming languages by both empirically testing and formally proving correct a mapping from an OpenCL-like scoped C++ memory model onto PTX. With this complete, our analysis builds a solid foundation on top of which future research and development efforts will be able to build. To aid in such efforts, we have publicly released our entire infrastructure: our line-by-line derivation of the PTX model from NVIDIA documentation, Alloy models, general-purpose Alloy-to-Coq compiler, and Coq proofs of PTX compliance with scoped C++.

Acknowledgments

This research was developed, in part, with funding from the United States Department of Energy. The views, opinions, and/or findings contained in this article are those of the authors and should not be interpreted as representing the official views or policies of the Department of Energy or the U.S. Government. We also thank the reviewers and our shepherd Joe Devietti for their helpful feedback.

References

[1] 2017. CppMem: Interactive C/C++ Memory Model. http://svr-pes20-cppmem.cl.cam.ac.uk/cppmem.

[2] 2017. diy, Release Seven. http://diy.inria.fr/.

[3] 2017. Lem, a Tool for Lightweight Executable Mathematics. http://www.cl.cam.ac.uk/~pes20/lem.

[4] 2017. The Coq Proof Assistant. https://coq.inria.fr.

[5] 2017. The HOL Interactive Theorem Prover. https://hol-theorem-prover.org.

[6] Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. 2015. GPU Concurrency: Weak Behaviours and Programming Assumptions. In *20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[7] Jade Alglave and Luc Maranget. 2016. *Towards a Formalization of the HSA Memory Model in the cat Language*. Technical Report. HSA Foundation Specification Version 1.1. URL: http://www.hsafoundation.com/?ddownload=5381.

[8] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2010. Fences in Weak Memory Models. In *22nd International Conference on Computer Aided Verification (CAV)*.

[9] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Transanctions on Programming Languages and Systems (TOPLAS)* 36, 2 (July 2014), 7:1–7:74.

[10] Konstantine Arkoudas, Sarfraz Khurshid, Darko Marinov, and Martin Rinard. 2003. Integrating Model Checking and Theorem Proving for Relational Reasoning. *International Conference on Relational Methods in Computer Science (RelMiCS)*.

[11] ARM. 2011. *Cortex-A9 MPCore™, Programmer Advice Notice, Read-after-Read Hazards*. Technical Report. URL: http://infocenter.arm.com/help/topic/com.arm.doc.uan0004a/UAN0004A_a9_read_read.pdf.

[12] ARM Holdings. 2016. *ARM Architecture Reference Manuals*. Technical Report. URL: http://infocenter.arm.com/help/topic/com.arm.doc.set.architecture.

[13] Mark Batty, Alastair F. Donaldson, and John Wickerson. 2016. Overhauling SC Atomics in C11 and OpenCL. In *43rd Annual Symposium on Principles of Programming Languages (POPL)*.

[14] Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. 2012. Clarifying and Compiling C/C++ Concurrency: from C++11 to POWER. *39th Symposium on Principles of Programming Languages (POPL)* (2012).

[15] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. *38th Symposium on Principles of Programming Languages (POPL)* (2011).

[16] Jasmin Christian Blanchette and Tobias Nipkow. 2010. Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder. *23rd International Conference on Interactive Theorem Proving (ITP)*.

[17] Jasmin Christian Blanchette, Tjark Weber, Mark Batty, Scott Owens, and Susmit Sarkar. 2011. Nitpicking C++ Concurrency. *13th International Symposium on Principles and Practice of Declarative Programming (PPDP)*.

[18] Hans-J. Boehm and Brian Demsky. 2014. Outlawing Ghosts: Avoiding Out-of-thin-air Results. In *Workshop on Memory Systems Performance and Correctness (MSPC)*.

[19] Hans-J. Boehm, Olivier Giroux, Viktor Vafeiadis, and with input from Will Deacon, Doug Lea, Daniel Lustig, Paul McKenney and others [sic]. 2018. P0668R3: Revising the C++ memory model. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0668r3.html. *ISO/JTC1/SC22/WG21* (2018).

[20] Adam Chlipala. 2013. *Certified Programming with Dependent Types: a Pragmatic Introduction to the Coq Proof Assistant*. MIT Press.

[21] Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E Gray, Ali Sezgin, Mark Batty, and Peter Sewell. 2017. Mixed-size Concurrency: ARM, POWER, C/C++11, and SC. (2017).

[22] HSA Foundation. 2017. Heterogeneous System Architecture. http://www.hsafoundation.com/standards.

[23] Aboubakr Achraf El Ghazi and Mana Taghdiri. 2011. Relational Reasoning via SMT Solving. *International Symposium on Formal Methods (FM)*.

[24] Khronos Group. 2017. OpenCL 2.2. https://www.khronos.org/opencl.

[25] Mark Harris. 2017. Unified Memory for CUDA Beginners. *NVIDIA Developer Blog* (2017). https://devblogs.nvidia.com/unified-memory-cuda-beginners.

[26] Mark Harris and Kyrylo Perelygin. 2017. Cooperative Groups: Flexible CUDA Thread Programming. *NVIDIA Developer Blog* (2017). https://devblogs.nvidia.com/cooperative-groups.

[27] Derek R. Hower, Blake A. Hechtman, Bradford M. Beckmann, Benedict R. Gaster, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. 2014. Heterogeneous-race-free Memory Models. *19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[28] International Organization for Standardization (ISO). 2011. *Information technology – Programming languages – C++, ISO/IEC 14882:2011*. Technical Report.

[29] International Organization for Standardization (ISO). 2011. *Information technology – Programming languages – C, ISO/IEC 9899:2011*. Technical Report.

[30] Daniel Jackson. 2002. Alloy: A Lightweight Object Modelling Notation. In *ACM Transactions on Software Engineering and Methodology (TOSEM)*, Vol. 11. URL: http://alloy.mit.edu.

[31] Khronos Group. 2015. *The OpenCL Specification, Version 2.2*. Technical Report. URL: https://www.khronos.org/registry/cl/specs/opencl-2.2.pdf.

[32] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++11. (2017).

[33] Daniel Lustig, Michael Pellauer, and Margaret Martonosi. 2014. PipeCheck: Specifying and Verifying Microarchitectural Enforcement of Memory Consistency Models. *47th Annual International Symposium on Microarchitecture (MICRO)* (2014).

[34] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. 2017. Supplemental material. https://github.com/nvlabs/ptxmemorymodel.

[35] Daniel Lustig, Andrew Wright, Alexandros Papakonstantinou, and Olivier Giroux. 2017. Automated Synthesis of Comprehensive Memory Model Litmus Test Suites. *22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2017).

[36] Yatin A. Manerkar, Caroline Trippel, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. 2016. Counterexamples and Proof Loophole for the C/C++ to POWER and ARMv7 Trailing-sync Compiler Mappings. *arXiv* 1611.01507v2 (Nov 2016).

[37] Baoluo Meng, Andrew Reynolds, Cesare Tinelli, and Clark Barrett. 2017. Relational Constraint Solving in SMT. *26th International Conference on Automated Deduction (CADE)*.

[38] Mariano M. Moscato, Carlos G. Lopez Pombo, and Marcelo F. Frias. 2014. Dynamite: a Tool for the Verification of Alloy Models based on PVS. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23. Issue 2.

[39] NVIDIA. 2017. CUDA C Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide.

[40] NVIDIA. 2017. Parallel Thread Execution ISA Version 6.0. http://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[41] NVIDIA. 2018. *NVIDIA Tesla V100 GPU Architecture: The World's Most Advanced Data Center GPU*. Technical Report.

[42] Oracle. 2017. Java Language and Virtual Machine Specifications. https://docs.oracle.com/javase/specs/.

[43] Peizhao Ou and Brian Demsky. 2018. Towards Understanding the Costs of Avoiding Out-of-Thin-Air Results. *Object-Oriented Programming, Systems, Languages & Applications Conference (OOPSLA)* (2018).

[44] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: x86-TSO. In *22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*.

[45] Gustavo Petri, Jan Vitek, and Suresh Jagannathan. 2015. Cooking the Books: Formalizing JMM Implementation Recipes. *29th European Conference on Object-Oriented Programming (ECOOP)* (2015).

[46] Power.org. 2013. *Power ISA™Version 2.07*. Technical Report. URL: https://www.power.org/wp-content/uploads/2013/05/PowerISA_V2.07_PUBLIC.pdf.

[47] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM Concurrency: Multicopy-Atomic Axiomatic and Operational Models for ARMv8. *45th Symposium on Principles of Programming Languages (POPL)*.

[48] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER Multiprocessors. In *32nd Conference on Programming Language Design and Implementation (PLDI)*.

[49] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. 2015. Efficient GPU Synchronization without Scopes: Saying No to Complex Consistency Models. In *48th Annual International Symposium on Microarchitecture (MICRO)*.

[50] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. 2017. Chasing Away RAts: Semantics and Evaluation for Relaxed Atomics on Heterogeneous Systems. *44th Annual International Symposium on Computer Architecture (ISCA)* (2017).

[51] Tyler Sorensen and Alastair F. Donaldson. 2016. Exposing Errors Related to Weak Memory in GPU Applications. In *37th Conference on Programming Language Design and Implementation (PLDI)*.

[52] Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2011. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers.

[53] SPARC International. 1993. *The SPARC Architecture Manual, Version 9*. Technical Report.

[54] Emina Torlak and Daniel Jackson. 2007. Kodkod: A Relational Model Finder. In *13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

[55] Caroline Trippel, Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. 2017. TriCheck: Memory Model Verification at the Trisection of Software, Hardware, and ISA. *22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2017).

[56] Mattias Ulbrich, Ulrich Geilmann, Aboubakr Achraf El Ghazi, and Mana Taghdiri. 2012. A Proof Assistant for Alloy Specifications. *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

[57] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. Common Compiler Optimisations Are Invalid in the C11 Memory Model and What We Can Do About It. In *42nd Symposium on Principles of Programming Languages (POPL)*.

[58] John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. 2017. Automatically Comparing Memory Consistency Models. *44th Symposium on Principles of Programming Languages (POPL)* (2017).