Beyond the Fence: Sound Compilation for GPU and Heterogeneous CPU+GPU Systems

Dennis Sprokholt
TU Munich
Munich, Germany
dennis.sprokholt@tum.de

Reese Levine
UC Santa Cruz
Santa Cruz, USA
reeselevine@ucsc.edu

Susmit Sarkar
University of St Andrews
St Andrews, United Kingdom
susmit.sarkar@st-andrews.ac.uk

Andrés Goens
TU Darmstadt
Darmstadt, Germany
andres.goens@tu-darmstadt.de

Vijay Nagarajan
University of Utah
Salt Lake City, USA
vijay@cs.utah.edu

Soham Chakraborty
TU Delft
Delft, Netherlands
s.s.chakraborty@tudelft.nl

Abstract

Modern heterogeneous processors like NVIDIA's Grace Hopper tightly integrate CPUs and GPUs with shared coherent memory, relying on independent compilation of CPU and GPU code. We show that the standard compilation path for such heterogeneous CUDA programs is unsound: Independently compiled code can violate the guarantees of the scoped-C/C++ concurrency model. In diagnosing this failure, we uncover a deeper flaw in the PTX memory model itself. The compilation from scoped-C/C++ to PTX—which NVIDIA formally verified and considered "an important prerequisite to the official release of the model"—is unsound; NVIDIA has confirmed our finding. The flaw stems from the model treating full fences specially, unlike other ordering primitives.

We resolve both the heterogeneous and homogeneous flaws with a single, targeted change to the PTX SC ordering. Our modified model preserves non-multicopy-atomic behaviors, respects scope boundaries, and—as we confirm through empirical litmus testing on NVIDIA A100, H100, GH200, and GB10 GPUs—imposes no new requirements beyond what the hardware already enforces. All soundness proofs are mechanized in Agda. Although our model is specific to PTX and ARM-PTX, the lesson is general: Ordering between fences should not be treated specially but should instead be derived from existing primitive relations.

CCS Concepts: • **Theory of computation** → **Program semantics**; • **Computer systems organization** → **Multicore architectures**; • **Software and its engineering** → *Formal methods.*

Keywords: weak memory; compound memory models; GPUs

ASPLOS '27, Heraklion, Greece
© 2027 Copyright held by the owner/author(s).
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of the 32nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '27), April 11–15, 2027, Heraklion, Greece*, https://doi.org/10.1145/3845814.3850083.

ACM Reference Format:
Dennis Sprokholt, Reese Levine, Susmit Sarkar, Andrés Goens, Vijay Nagarajan, and Soham Chakraborty. 2027. Beyond the Fence: Sound Compilation for GPU and Heterogeneous CPU+GPU Systems. In *Proceedings of the 32nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '27), April 11–15, 2027, Heraklion, Greece*. ACM, New York, NY, USA, 24 pages. https://doi.org/10.1145/3845814.3850083

1 Introduction

Can programs for heterogeneous processors with CPUs and GPUs be compiled independently and composed soundly? This question is critical for heterogeneous systems such as NVIDIA's Grace Hopper, which tightly integrates an ARM CPU and a Hopper GPU with coherent shared memory.

In a CUDA program, the CPU parts are compiled by GCC or Clang, and the GPU parts independently by NVCC. The correctness of this composition rests on the underlying memory consistency models (MCMs).

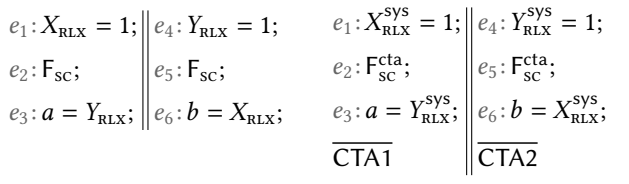
We show that a fundamental conflict between the CPU and the GPU MCMs makes the standard compilation path for heterogeneous CUDA programs unsound. The compilation can violate the guarantees of the scoped-C/C++ model—a scoped extension of the C++ memory model designed by NVIDIA to capture the semantics of GPU programming models like CUDA and OpenCL [25]. The compilation target we consider is the formalization [25] of the PTX model described in prose in the NVIDIA documentation [28, Sec. 8].

The conflict stems from fundamental architectural differences. CPUs, like ARM, typically provide Multi-Copy Atomicity (MCA), ensuring that, logically, writes become visible to all cores simultaneously. In contrast, GPUs often lack MCA (*i.e.* they are non-MCA), meaning writes may become visible to different cores at different times. This asymmetry leads to fundamentally different strategies for compiling C++ sequentially consistent (SC) atomics: ARM's MCA enables an efficient "no-fence" mapping that avoids costly memory fences [6, 14, 34], while non-MCA PTX requires a conservative "leading-fence" mapping that inserts an explicit *full fence* (also known as sc-fence) before each sc operation.

While investigating this heterogeneous incompatibility, we uncovered a more fundamental and pre-existing flaw, not in the composition, but in the soundness argument for compilation to the official PTX model itself. The PTX model was designed to correctly implement the scoped-C/C++ model [25], and its formal compatibility with scoped-C/C++ was considered to be "an important prerequisite to the official release of the model" [24]. This correctness argument, however, hinges on a critical, unproven transformation applied before the mechanization. Following prior work [20], the authors split the SC accesses of scoped-C/C++ into an sc-fence followed by a corresponding release/acquire access, assuming this simplification was sound for the scoped case as well. We demonstrate that this "fence-splitting" transformation is unsound in the presence of scopes: the newly-introduced SC fence does not inherit the same memory ordering from the original SC access across scopes, breaking the ordering guarantees mandated by scoped-C/C++. This transformation is unsound in the scoped-C/C++ model itself and thus for its compilation to NVIDIA PTX. We contacted the NVIDIA authors of the original PTX model [25]. They confirmed the flaw and were eager to learn about our proposed solution.

The definition of fence ordering is also the root cause of the heterogeneity problem. The ARM architecture, with its inherent MCA, provides strong ordering guarantees without using the explicit sc-fences that the PTX model's logic relies on. In both cases—whether composing with scoped-C/C++'s sc accesses or ARM's MCA—the PTX model is blind to critical orderings, leading to a soundness gap.

We resolve both correctness challenges with a single, model-level solution. We propose a modest change to the PTX model's SC ordering mechanism. The existing model mandates this order via sc fences only; we propose an alternate ordering derived from existing relations, including coherence (co) and from-read (fr). We formally prove that this single fix: *(a)* repairs the soundness proof for the existing homogeneous PTX compiler mappings for scoped-C/C++, and *(b)* guarantees the sound composition of these mappings with the "no-fence" mappings used by ARM (which uses synchronizing accesses and inserts no fences). All proofs are mechanized in Agda.



(a) Unscoped **(b)** Scoped, in different CTAs

Figure 1. A *store buffering* program, with unscoped and scoped variants. In (a), the weak behavior $a{=}b{=}0$ is disallowed by fences e_2 and e_5 in *un*scoped-C/C++. In (b), that behavior is allowed in both scoped-C/C++ and PTX as the fences are *not morally strong*: they are cta-scoped but in different CTAs.

This strengthening is surgical. First, it does not impose full MCA on the GPU; non-MCA behaviors like IRIW (Independent Reads of Independent Writes) are still permitted. Second, co and fr participate in the SC ordering only when the events share a sufficient scope—that is, when they are *morally strong*—so cross-scope weak behaviors remain allowed. Third, the hardware already satisfies our strengthened requirements: empirical litmus testing on NVIDIA A100, H100, GH200, and GB10 GPUs confirms every execution our model forbids is also empirically forbidden, imposing no microarchitectural cost.

Contributions

- We are the first to demonstrate that composing standard, optimized compiler mappings for heterogeneous ARM-PTX systems (no-fence vs. leading-fence) is unsound.
- We identify a subtle flaw in the soundness argument for the official scoped-C/C++ to PTX compiler mappings in Figure 5 [25], showing its reliance on an unproven "fence-splitting" transformation is unsound.
- We show these two distinct problems share a common root cause: the PTX model's special treatment of sc-fences fails to capture critical orderings in both the homogeneous scoped-C/C++ and heterogeneous ARM contexts.
- We propose a unified, model-level solution: a solution to the PTX SC ordering definition that fixes the scoped-C/C++-to-PTX miscompilation.
- We propose the ARM-PTX model and prove that existing compilation schemes from scoped-C/C++ to ARM and PTX remain sound. We have mechanized these proofs in Agda.
- We validate our model with empirical litmus testing on NVIDIA hardware, finding no violations of our proposed model and validating our surgical strengthening while still allowing observable non-MCA behaviors.

2 Two Problems, Root Cause, One Solution

To make the claims from the introduction concrete, we motivate the problem with two key litmus tests [3]. The first, a heterogeneous test, exposes a challenge in compiler correctness when composing ARM and PTX. The second, a homogeneous test, reveals a similar issue rooted in a subtle pre-existing error in the compilation from scoped-C/C++ to PTX.

Our goal in this section is to provide a high-level, intuitive overview of these failures. The formal definitions for the models and relations (like happens-before, scopes, and morally strong) that underpin this analysis will be detailed in Section 3.

2.1 Review: Scoped Memory Ordering Intuition

Before exploring those problems, we give an intuition for weak memory semantics of scoped models, which we consider in this paper. We explain by example through two variants of the *store buffering* program in Figure 1, starting with the simpler unscoped variant in Figure 1a.

The program has two threads, separated by the vertical ||. We assume that all variables are initialized to 0 before creating the threads. Operation e_1 writes 1 to X and e_3 reads from Y. As

both accesses are relaxed (RLX) they could independently execute out-of-order, but the intermediate full fence ensures they *cannot* reorder across it. Similarly, the right thread writes to Y and reads from X, with full fence e_5 in-between. Without those fences e_2 and e_5, the memory accesses could reorder, resulting in the weak outcome a=b=0. However, as the fences restrict ordering, that outcome is forbidden in unscoped-C/C++.

Figure 1b is the same program, but now with scope annotations. In particular, the memory accesses e_1, e_3, e_4, and e_6 are sys-scoped, which means their ordering is visible across the entire *system*, including CPUs and GPUs. When two scoped memory operations have scope annotations that include the thread executing the other, then those accesses are *morally strong* [25]. For instance, e_3 and e_4 are morally strong as both have sys-annotations and both threads execute on the same system. In contrast, the fences e_2 and e_5 are cta-scoped, meaning their order is only visible to other threads in the *cooperative thread array* (CTA). As the threads execute in different CTAs, the fences are *not* morally strong, making the weak behavior a=b=0 observable in both scoped-C/C++ and PTX.

We use these scoped weak memory semantics to reason about compilation correctness of scoped-C/C++ to PTX or heterogeneously to ARM-PTX.

2.2 Symptom: Heterogeneous Composition Unsound

To illustrate the challenge of heterogeneous ARM-PTX composition, consider the C++ program in Figure 2a, a test of transitive synchronization forbidden by the C++ standard. A program with at least four threads can have two synchronizations between SC-accesses in two threads each, showing how two fences appear outside the ordering path when compiling to PTX (Figure 2c) and ARM-PTX (Figure 2d). In Figure 2a, the operations in each thread are reads, writes, and fences, with release (REL), acquire (ACQ), or sequentially consistent (SC) memory orders. For example, the first operation in the first thread (T_1), '$e_1 : X_{\text{SC}} = 1$' is an SC write to shared variable X of the value 1. We label this access with e_1.

2.2.1 Correctness on a Homogeneous CPU or GPU.

Figure 2b shows the generated ARM program, where the C++ SC and release writes map to ARM release write (L), while acquire and SC reads translate to ARM acquirePC (Q) and acquire (A) reads, respectively. These release accesses order with predecessors, while acquire accesses (including acquirePC) order with succeeding accesses. The accesses create an ordering chain from e_1 to e_8, which results in a=c=1,b=0, but correctly forbids d to read value 0.

Now consider the generated PTX program in Figure 2c[1]. This strategy implements an SC memory operation by replacing it with two instructions: an SC-fence that leads a memory access with weaker semantics; specifically, an SC-store becomes an SC-fence followed by a REL-store, while an SC-load becomes an SC-fence followed by an ACQ-load. This mapping to PTX correctly disallows the forbidden outcome by establishing a complete ordering chain from the write to X in e_1 to the final read of X in e_8.

- A release-acquire chain orders the first write (e_1) with the fence in T_2 (ρ_4).
- The SC-fence in T_2 (ρ_4) and the SC-fence in T_4 (ρ_8) are part of a total order over morally-strong SC-fences. Here, b (e_4) reads 0 (from the initial value) and not 1 (from e_5) forcing the fence ρ_4 to be ordered before the fence ρ_8 in this total order.
- Finally, the fence in T_4 (ρ_8) is ordered by program order with the final load (e_8). This creates a complete chain forcing the final read (e_8) to see the initial write (e_1) and thus forbidding the weak outcome.

2.2.2 The Puzzle on a Heterogeneous System.

The situation becomes more complex on a heterogeneous machine like Grace Hopper. Consider the case where threads T_1–T_3 run on the GPU (PTX) and T_4 runs on the CPU (ARM), as shown in Figure 2d. The ARM architecture's strong MCA guarantees, combined with the fact that a release-write is ordered with a subsequent (in program order) acquire-read enables the compiler to use the efficient "no-fence" strategy. The C++ ACQ-load of V in T_4 (e_7) and SC-load of X (e_8) are translated directly to load-acquirePC and load-acquire instructions, respectively.

The critical SC-fence in T_4 (Figure 2c, ρ_8) is now absent. The PTX model's SC ordering logic, *which relies only on a*

$$\begin{array}{l|l|l|l} e_1 : X_{\text{SC}} = 1 & e_3 : a = Y_{\text{ACQ}} & e_5 : Z_{\text{SC}} = 1 & e_7 : c = V_{\text{ACQ}} \\ e_2 : Y_{\text{REL}} = 1 & e_4 : b = Z_{\text{SC}} & e_6 : V_{\text{REL}} = 1 & e_8 : d = X_{\text{SC}} \end{array}$$

(a) Source (C/C++)

$$\begin{array}{l|l|l|l} e_1 : X_{\text{L}} = 1 & e_3 : a = Y_{\text{Q}} & e_5 : Z_{\text{L}} = 1 & e_7 : c = V_{\text{Q}} \\ e_2 : Y_{\text{L}} = 1 & e_4 : b = Z_{\text{A}} & e_6 : V_{\text{L}} = 1 & e_8 : d = X_{\text{A}} \end{array}$$

(b) ARM

$$\begin{array}{l|l|l|l} \rho_1 : \text{F}_{\text{SC}} & e_3 : a = Y_{\text{ACQ}} & \rho_5 : \text{F}_{\text{SC}} & e_7 : c = V_{\text{ACQ}} \\ e_1 : X_{\text{REL}} = 1 & \rho_4 : \text{F}_{\text{SC}} & e_5 : Z_{\text{REL}} = 1 & \rho_8 : \text{F}_{\text{SC}} \\ e_2 : Y_{\text{REL}} = 1 & e_4 : b = Z_{\text{ACQ}} & e_6 : V_{\text{REL}} = 1 & e_8 : d = X_{\text{ACQ}} \end{array}$$

(c) PTX

$$\begin{array}{l|l|l|l} \text{(PTX)} & \text{(PTX)} & \text{(PTX)} & \text{(ARM)} \\ \rho_1 : \text{F}_{\text{SC}} & e_3 : a = Y_{\text{ACQ}} & \rho_5 : \text{F}_{\text{SC}} & \\ e_1 : X_{\text{REL}} = 1 & \rho_4 : \text{F}_{\text{SC}} & e_5 : Z_{\text{REL}} = 1 & e_7 : c = V_{\text{Q}} \\ e_2 : Y_{\text{REL}} = 1 & e_4 : b = Z_{\text{ACQ}} & e_6 : V_{\text{REL}} = 1 & e_8 : d = X_{\text{A}} \end{array}$$

(d) ARM-PTX

Figure 2. An example of compilation from C++ to ARM, PTX, and ARM-PTX following the existing mapping schemes [25, 40]. The outcome a=c=1, b=d=0 is forbidden in C++. Hence, the generated programs must forbid the outcome to ensure compiler correctness. (All accesses are sys-scoped)

[1]We assume each of the operations are system scoped, but do not show the scope labels in the litmus test

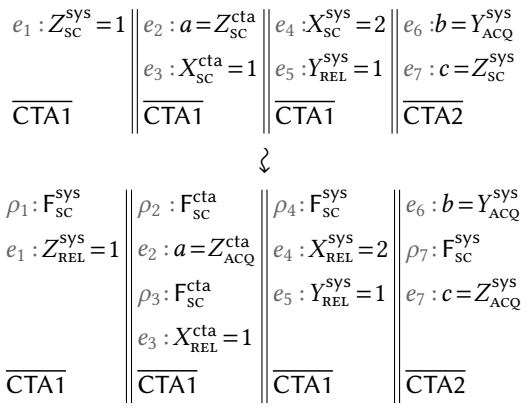


Figure 3. An incorrect compilation from scoped-C/C++ to PTX. The outcome a=b=1, c=0, X=2 is forbidden in the source program but allowed in the generated PTX program.

total order of SC fences, is now broken in Figure 2d. The fence ρ_4 in T_2 has no corresponding fence in T_4 to order against. This raises the key puzzle: *Without that fence, how can e_8 be forced to see the value of e_1? How can correctness be maintained across the architectural boundary, while allowing these efficient, independent mappings?*

2.3 The Problem: A Pre-existing Flaw in PTX

Treating SC-fences as the primary primitive to enforce memory ordering is not just a problem for heterogeneity; it is linked to a fundamental, pre-existing flaw in the soundness argument for the homogeneous PTX model itself. We demonstrate this using the scoped-C/C++ litmus test shown in Figure 3. The test consists of four threads, with the first three threads belonging to one CTA and the last thread belonging to a different CTA. This test is forbidden by the scoped-C/C++ model [25].

Informally, the model establishes a chain of SC ordering edges from e_1 to e_7. For this chain to be valid, each link must be "morally strong" (ms)—a key concept in scoped models. This simply means that for two operations to synchronize, their scopes must be compatible (*e.g.,* one operation's scope must include the other's thread). In this test, the chain is pairwise morally-strong and unbroken, which forces the final read e_7 to see the value from e_1.

However, when compiled to PTX using the leading-fence mapping, this outcome is incorrectly allowed. The PTX model fails because its SC ordering logic is too weak, establishing an SC order only between fences that are morally strong with each other. In this example, the relevant fences ρ_3 and ρ_7 are *not* morally strong, since ρ_3 has cta-scope but ρ_7 is in a different CTA. Crucially, being morally strong is *not* transitive.

2.4 The Common Root Cause and Our Solution

Both the heterogeneous and homogeneous bugs stem from the same root cause: the PTX model treats SC-fences as the primary primitive to enforce ordering across scopes, while ordering of other primitives is not sufficiently considered. This single flaw manifests in two distinct ways:

- In the heterogeneous case, the logic fails because the ARM thread provides no fence for the PTX model to order against.
- In the homogeneous case, the logic fails because, while the individual links in the chain are morally strong, the provided fences are not, and thus establish no ordering.

We propose a unified, model-level solution. The main problem with the existing model is that the source language model orders SC-*accesses*, for which the mapping pushes the leading SC-*fences* outside the ordering path. Figure 2d showed only one SC-fence appearing on the path between reads and writes, which cannot enforce the ordering by itself. Our model must thus include those edges the source model used to forbid the outcome – and nothing more, ensuring observable non-MCA behaviors remain consistent with our model.

We fix that root cause by enhancing the PTX model's SC ordering definition. Instead of relying only on fences, our strengthened model redefines the SC order to also include: coherence order (co), which enforces the order between writes to the same memory location; and from-read (fr), which enforces the order from a read to a subsequent write that overwrites the value just read. This surgical strengthening restores correctness in both cases:

- For the heterogeneous bug (Figure 2d), the fr order from e_4 to e_5 is now included in the SC order. This fr order composes with ARM's global memory order to disallow the forbidden outcome, even without a fence on the ARM side.
- For the homogeneous bug (Figure 3), the co from e_3 to e_4 is now included in the SC order, disallowing the outcome directly without relying on (now-redundant) fence ordering.

This allows both compilers to soundly continue using their optimal, independent strategies (no-fence and leading-fence).

3 Preliminaries: Concurrency Models

We now review the relevant fragments of the axiomatic formalization of two existing memory consistency models: The existing PTX model (Section 3.1 and Figure 4b), defined on the PTX virtual architecture for NVIDIA GPUs [25, 29], and the scoped-C/C++ model (Section 3.2). The standard notations of axiomatic consistency [4, 8] and details of the ARM model are included in Appendix B.

> ► **Notation.** In our examples we omit initialization events (writing 0) and their outgoing po and rf (to events reading 0) edges. All examples consider memory initialized to 0.

3.1 PTX

The existing axiomatic PTX model in Figure 4b is defined on the PTX virtual architecture for NVIDIA GPUs [25, 29].

Events. In the PTX model, the read, write, RMW, and fence accesses have memory orders: weak (WK), relaxed (RLX), acquire (ACQ), release (REL), acquire-release (AQRL), and sequentially-consistent (SC). We will also explain these memory orders upon use, but give them here for completeness. The

ob is irreflexive *where* (External)

$ob \triangleq (rfe \cup coe \cup fre \cup bob \cup \ldots)^+$

$bob \triangleq [W_L];po;[R_A] \cup [R_{(A|Q)}];po \cup po;[W_L] \cup \ldots$

(a) ARM

$(sc;cause)$ is irreflexive *where* (FenceSC)

$sw \triangleq (ms \cap ([W_{REL}];rf;[R_{ACQ}])) \cup sc \cup \ldots$

$cause \triangleq (po^?;sw;po^?)^+ \cup \ldots$

(b) PTX

Figure 4. Relevant fragment of existing ARM and PTX axiomatic memory models [25, 35]. (Fully in Appendix B)

$hb;eco^?$ is irreflexive (Coherence)

$rmw \cap (fr;co) = \emptyset$ (Atomicity)

$(ms \cap psc)$ is acyclic (SC)

where

$rseq \triangleq [W];po_{=loc}^?;[W_{\sqsupseteq RLX}];((ms \cap rf);rmw)^*$

$sw \triangleq [E_{\sqsupseteq REL}];([F];po)^?;rseq;(ms \cap rf);$
$[R_{\sqsupseteq RLX}];(po;[F])^?;[E_{\sqsupseteq ACQ}]$

$hb \triangleq (po \cup (ms \cap sw))^+$

$scb \triangleq po \cup (po_{\neq loc};hb;po_{\neq loc}) \cup hb_{=loc} \cup co \cup fr$

$pscb \triangleq ([E_{sc}] \cup [F_{sc}];hb^?);scb;([E_{sc}] \cup hb^?;[F_{sc}])$

$pscf \triangleq [F_{sc}];(hb \cup hb;eco;hb);[F_{sc}]$

$psc \triangleq pscb \cup pscf$

(a) scoped-C/C++ model [25] (without No-Thin-Air)

C/C++	ARM	PTX
ld^{sco}_{sc}	ld^{sys}_{A}	$F^{sco}_{sc};ld^{sco}_{ACQ}$
st^{sco}_{sc}	st^{sys}_{L}	$F^{sco}_{sc};st^{sco}_{REL}$

C/C++	ARM	PTX
ld^{sco}_{ACQ}	ld^{sys}_{Q}	ld^{sco}_{ACQ}
st^{sco}_{REL}	st^{sys}_{L}	st^{sco}_{REL}

(b) Mapping of SC accesses. (Full mapping in Appendix B.4)

Figure 5. The scoped-C/C++ formal model [25] and its mapping of SC accesses to ARM and PTX [25, 40].

memory order of a read, write, RMW, and fence operation is one of {WK, RLX, ACQ}, {WK, RLX, REL}, {RLX, ACQ, REL, AQRL}, and {ACQ, REL, AQRL, SC}, respectively. The annotations order: WK < RLX < {ACQ, REL} < AQRL < SC.

Non-weak PTX accesses also have an additional parameter: scope. The scope denotes the thread hierarchies within which the event may participate in synchronization – cta for all threads in a cooperative thread array, gpu for all threads on the same GPU, and sys for all threads in all devices, *i.e.* in terms of the hierarchy of scopes, we have cta < gpu < sys.

Relations. On these events, PTX defines two additional primitive relations: A pair of non-weak events is *morally strong* (ms) when both events are not weak and have a scope that includes the thread of each other; and the sc relation orders a pair of morally-strong SC fences.

The PTX model has a synchronization (sw) relation between a pair of morally strong release and an acquire events, which may arise in several ways. For example, if a load-acquire reads-from a store-release and the events are morally strong, then it establishes synchronization. sc also establishes synchronization sw. With on sw and po PTX derives a global ordering cause as defined in Figure 4b.

Axioms. Axiom (FenceSC) in Figure 4b is most relevant for us: it restricts the ordering between F_{sc} events, generated by full fences, along sc and cause. The additional axioms in the original PTX are in Appendix B, which enforce consistency based on cause and co, rf, and fr.

3.2 Scoped-C/C++

The scoped-C/C++ model in Figure 5a [25] provides non-atomic load and store accesses, similar to weak PTX accesses. Conversely, scoped-C/C++ provides atomic read, write, RMW, and fence accesses, with memory orders and scopes, like the strong PTX accesses. Unlike PTX, however, scoped-C/C++ also provides SC memory accesses. Thus, the memory order for the read, write, RMW, and fence accesses is one of {NA, RLX, ACQ, SC}, {NA, RLX, REL, SC}, {RLX, REL, ACQ, AQRL, SC}, and {REL, ACQ, AQRL, SC}. Scoped-C/C++ provides the same memory scopes as PTX and defines the *scope-inclusive* relation, which is the same as the morally-strong relation [25].

Relations. Scoped-C/C++ defines the *synchronizes-with* (sw) relation between release and acquire events. The transitive closure of po and morally strong sw creates the *happens-before* (hb) relation. Scoped-C/C++ adopted psc, pscb, and pscf relations from RC11 [20], extended with scopes, to order sc accesses – unlike the explicit sc order in PTX.

An execution has a heterogeneous race if it contains a pair of concurrent events on same memory locations, at least one of them is a write, and the event pair is not morally-strong. Any program with a data race (like in C/C++) or heterogeneous race has undefined behavior.

Axioms. The (Coherence) axiom ensures that hb is acyclic or a pair of same-location hb-related events and eco relation agree. The (SC) axiom ensures that the morally-strong psc relation remains acyclic. (Atomicity) ensures that a pair of rmw-related events has no intermediate write on the same location. We drop the (No-Thin-Air) axiom from scoped-C/C++ [25] as GPUs exhibit po∪rf cycles [2, 19, 22].

3.3 Mapping from scoped-C/C++ to ARM and PTX

Proposed mapping schemes of the scoped-C/C++ accesses to ARM and PTX are in Figure 5b (extended in Appendix B.4), defined by Sewell [40] and Lustig et al. [25] respectively. A scoped-C/C++ load (ld), store (st), atomic update (U), and

fence (F) accesses are mapped to the respective accesses in ARM and PTX with the respective memory order and scopes.

ARM mapping schemes. For the ARM mapping, all generated accesses are of sys-scope. Notably, C/C++'s load-acquire (ld_{ACQ}) maps to an ARM load-acquirePC (ld_Q^{sys}), while a load-SC (ld_{SC}) maps to an ARM load-acquire (ld_A^{sys}), because the latter orders stronger with preceding store-release (st_L^{sys}) instructions, which are mapped from C++ sc-store (st_{SC}) and release-store (st_{REL}) instructions. An acquire fence (F_{ACQ}) maps to an ARM load fence (F_{RM}^{sys}), while a release fence (F_{REL}) maps to an ARM full fence (F_{MM}^{sys}). The mapping of other instructions is straightforward.

PTX mapping scheme. For all accesses, the memory scope remains the same. The scoped-C/C++ non-sc memory accesses and sc-fences are mapped one-to-one to accesses in PTX. The sc accesses follow a leading scheme – an sc-load, -store, or -update access will generate an acquire-load, release-store, and acquire-release update access respectively, with a preceding (*i.e.* leading) sc-fence.

4 Deriving an ARM-PTX Concurrency Model

We now derive a sound concurrency model for heterogeneous ARM-PTX systems. Our goal is to define a model satisfying two competing requirements: it must be (1) strong enough to ensure correct compilation from the scoped-C/C++ concurrency model defined in Figure 5a, yet (2) weak enough to capture the behaviors of real systems. We identify the required ordering rules in a principled, step-by-step manner, guided not only by the formal requirements of the source model but also by architectural properties of GPU coherence protocols that inform what the hardware naturally enforces.

4.1 Architectural Background

Axiomatic memory models are defined over threads and scopes, abstracting over how scopes are realized in the memory hierarchy. To justify our proposed model, we sketch the architectural intuition behind scoped GPU coherence and its connection to the CPU. While the microarchitectural intuition is based on the NVIDIA GPUs we evaluate (Section 6), our model does not hinge on these exact details, and future microarchitectures may realize the model differently. GPUs expose a scoped thread hierarchy. Threads within a scope traditionally share a common serialization point in the cache hierarchy: CTA-scoped threads share the L1, GPU-scoped threads share the L2, and system-scoped threads are connected via a coherent interconnect such as AMBA CHI [5]. Unlike CPU L1s, GPU L1s are not kept coherent by writer-initiated invalidations—a write to an L1 does not automatically invalidate copies in other L1s. Instead, GPU protocols rely on write-through and self-invalidation [27]. A GPU-scoped release ensures all prior writes have been written back to the L2 before the release completes. A GPU-scoped acquire self-invalidates the L1, forcing subsequent reads to observe the latest value at the L2. A system-scoped release additionally flushes prior writes to the CPU-GPU coherent interconnect, making them visible to CPU threads. sc-fences extend this mechanism: an sc-fence synchronously flushes all prior writes to its scope's serialization point—a CTA-scoped fence to the L1, a GPU-scoped fence to the L2, and a system-scoped fence to the entire system. The flush completes before any subsequent operation proceeds. This scoped, synchronous propagation is the architectural property that underpins our formal model: it naturally preserves coherence-order (co) and from-read (fr) edges across barriers, which is precisely what our strengthening exposes. Note that recent and future GPU microarchitectures may not adhere to the above sketch in every detail. Even on the GPUs we evaluate, the L2 is physically partitioned; in multi-die designs such as NVIDIA's B200, it spans dies and need not be a single serialization point. The programming model, however, is unchanged: NVIDIA GPUs continue to expose scopes, and both the official PTX model and ours are defined at that level.

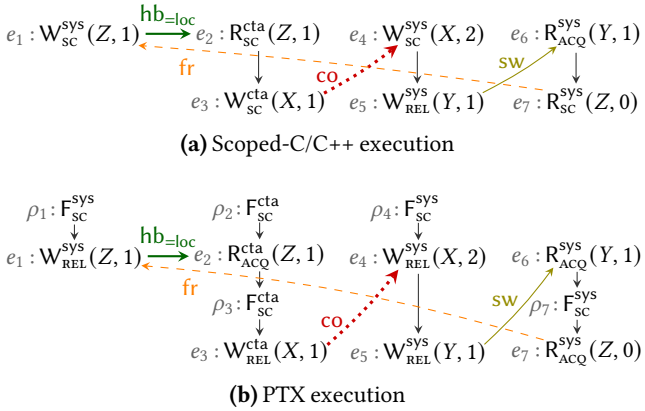


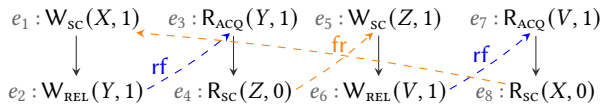
Figure 6. Execution of the source scoped-C/C++ and the generated PTX programs from Figure 3. The source scoped-C/C++ execution forbids the outcome a=b=1, c=0, X=2 but the PTX execution allows it (under the pre-existing PTX model).

4.2 Revisiting the PTX mapping example

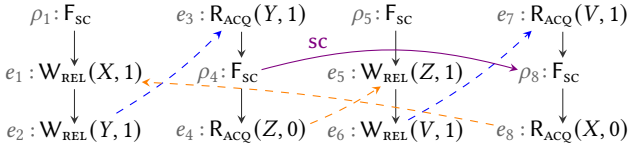
Consider the executions in Figure 6 of the programs in Figure 3. The scoped-C/C++ execution contains a (ms ∩ psc) cycle as follows, where ms holds for all pairs below.

- $\langle e_1,e_2\rangle \in [E_{sc}];hb_{=loc};[E_{sc}] \subseteq [E_{sc}];scb;[E_{sc}] \subseteq psc$
- $\langle e_2,e_3\rangle \in [E_{sc}];po;[E_{sc}] \subseteq [E_{sc}];scb;[E_{sc}] \subseteq psc$
- $\langle e_3,e_4\rangle \in [E_{sc}];co;[E_{sc}] \subseteq psc$
- $\langle e_4,e_7\rangle \in [E_{sc}];po_{\neq loc};hb;po_{\neq loc};[E_{sc}] \subseteq [E_{sc}];scb;[E_{sc}] \subseteq psc$
- $\langle e_7,e_1\rangle \in [E_{sc}];fr;[E_{sc}] \subseteq psc$

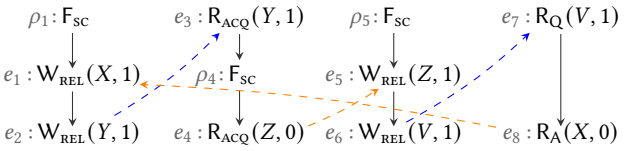
Now, consider the leading-fence mapping and the corresponding PTX execution graph. This execution is (incorrectly) allowed in PTX; Although it contains a cycle $e_1 \xrightarrow{cause} e_3 \xrightarrow{co} e_4 \xrightarrow{cause} e_7 \xrightarrow{fr} e_1$, the existing PTX axioms are not strong enough to forbid it. This is because the ρ_3 and ρ_7 fences are not morally-strong (ms) and cannot establish an sc-relation. Hence, the target program allows the outcome a=1, b=1, c=0, X=2, and the compilation is thus incorrect.



(a) Source (C/C++)



(b) Target (pre-existing PTX)



(c) Target (ARM-PTX) – three PTX threads, right-most on ARM

Figure 7. Executions of the example programs in Figure 2. scoped-C/C++ forbids the outcome a=1, b=0, c=1, d=0. ARM-PTX must thus also forbid the outcome in the target execution to ensure correctness of the existing mapping schemes.

4.3 Revisiting the ARM-PTX mapping example

Now we revisit the example from Figure 2 with executions from PTX and ARM-PTX shown in Figure 7, following the respective formal models.

Scoped-C/C++. The execution in Figure 7a is forbidden in scoped-C/C++ (Figure 5). As all events are sys-scoped, all relations are ms, violating the (SC) axiom with the (ms ∩ psc) cycle: $e_1 \xrightarrow{s} e_4 \xrightarrow{fr} e_5 \xrightarrow{s} e_8 \xrightarrow{fr} e_1$, where $s \triangleq \mathrm{po}_{\neq \mathrm{loc}}; \mathrm{hb}; \mathrm{po}_{\neq \mathrm{loc}} \subseteq \mathrm{scb}$.

PTX. The execution is in Figure 7b. Following the PTX model in Figure 4b, $\langle e_1, e_8 \rangle \in$ cause: $e_1 \xrightarrow{po} e_2 \xrightarrow{sw} e_3 \xrightarrow{po} \rho_4 \xrightarrow{sc \subseteq sw} \rho_8 \xrightarrow{po} e_8$. Finally, with $\langle e_8, e_1 \rangle \in$ fr the (Causality) axiom is violated, thus forbidding the execution.

ARM-PTX. In the ARM-PTX execution in Figure 7c, where the first three threads are on PTX and the last thread on ARM, relations $e_1 \xrightarrow{cause} e_4$ and $e_4 \xrightarrow{fr} e_5$ hold. Also, assuming $\langle e_6, e_7 \rangle$ establishes a global synchronization in ARM-PTX, the execution has a causality relation from e_5 to e_8. The execution has a $e_1 \xrightarrow{cause} e_4 \xrightarrow{fr} e_5 \xrightarrow{cause} e_8 \xrightarrow{fr} e_1$ cycle.

Compared to the PTX execution in Figure 7b, a subtle difference is that the PTX execution has an explicit fence ρ_8 followed by the R_{ACQ} event e_8. However, this ARM-PTX execution has no explicit fence in T_4, instead it has an R_A event in e_8. Consequently, unlike the $\rho_4 \xrightarrow{sc} \rho_8$ edge in the PTX execution, this ARM-PTX execution does not have any sc order from T_2 to T_4. Hence, if we directly apply the PTX axioms, the ARM-PTX execution is allowed. Therefore, to ensure correct mapping from scoped-C/C++ to ARM-PTX, we must derive stronger ordering constraints for ARM-PTX.

This requirement raises an appealing solution: *treat* R_A *as a full fence and create a global* sc *order.* However, this is impractical: it would require the ARM subsystem to receive and interpret GPU fence events over the coherent interconnect, stalling its local pipeline until the remote fence completes. This violates the architectural abstraction boundary—the ARM core's pipeline and coherence protocols should be designed around the ARM ISA, with no awareness of GPU-internal events.

4.4 Insight 1: The Missing Links: co and fr

Following the ARM-PTX execution in Figure 7c, the key insight lies in identifying the relations that are "lost" in the mapping from the source. In the source execution (Figure 7a), the fr relations $\langle e_4, e_5 \rangle$ and $\langle e_8, e_1 \rangle$ are both part of psc, which forbids the execution. However, in the original PTX model, the corresponding fr relation between the non-SC events (*e.g.*, $e_4 \xrightarrow{fr} e_5$) does not imply any causality order. The model relies on the explicit PTX SC fences, ρ_4 and ρ_8, to enforce the ordering.

Hence, our PTX model includes fr relations in the global ordering, for instance, $e_4 \xrightarrow{fr} e_5$, independent of the SC fences. It is also required that the $e_8 \xrightarrow{fr} e_1$ relation between the ARM and PTX accesses is a part of this global ordering. Thus, the inclusion of fr restores the complete cycle in the heterogeneous execution: $e_1 \xrightarrow{cause} \ldots \xrightarrow{fr} e_5 \xrightarrow{cause} \ldots \xrightarrow{fr} e_1$. Thus our new model forbids e_8 to read 0, restoring soundness.

In addition to the fr relation, we also add the co relation as part of the global ordering. To understand the reason, consider a small change in the original C program (Figure 7a): replace the load on Z from T_2 (event e_4) with a store to Z. The same logic applies: this would change the critical $e_4 \xrightarrow{fr} e_5$ edge to a co edge. Ensuring this variant compiles soundly requires also adding co to the global ordering.

Note that the scoped-C/C++ model requires acyclicity of (ms ∩ psc)—only morally-strong edges participate in the SC ordering. Accordingly, our model only requires morally-strong co and fr edges to be ordered. However, as we show in Section 4.5, these edges cannot simply be added to the existing cause relation—doing so would forbid observable non-MCA behaviors such as IRIW. Instead, morally-strong co and fr must participate in a new, weaker ordering, derived from the remaining insights and formalized as prop in Section 5.

This strengthening (morally strong co and fr) is architecturally grounded. As discussed in Section 4.1, GPU writes serialize at a common scope-level serialization point (*e.g.*, the L2 for GPU scope), and the synchronous propagation enforced by barriers ensures that barrier-ordered predecessors are visible before their successors. Morally-strong co and fr edges across barriers are therefore already respected by GPU coherence protocols—our model simply formalizes what the hardware naturally enforces.

4.5 Insight 2: Full co ∪ fr Is Too Strong

While the co and fr relations are needed to forbid the unsound execution in Figure 7c from Insight 1 (Section 4.4),

a naive strengthening—for example, simply forbidding all (cause ∪ co ∪ fr) cycles—is too strong.

We demonstrate this using the IRIW-RA (Independent-Read-Independent-Write-Release-Acquire) example in Figure 8. The execution with outcome $a{=}c{=}1$, $b{=}d{=}0$ is allowed in the non-MCA models, including PTX, and is empirically observable on PTX hardware. However, the shown execution contains a (cause ∪ fr) cycle: $e_1 \xrightarrow{\text{cause}} e_3 \xrightarrow{\text{fr}} e_6 \xrightarrow{\text{cause}} e_5 \xrightarrow{\text{fr}} e_1$, as $e_1 \xrightarrow{\text{sw}} e_2$, implying $e_1 \xrightarrow{\text{cause}} e_3$, and $e_6 \xrightarrow{\text{sw}} e_4$, implying $e_6 \xrightarrow{\text{cause}} e_5$.

Hence, a model that simply forbids all (cause ∪ co ∪ fr) cycles would incorrectly disallow this valid, observable behavior. We must derive a weaker, more nuanced order that allows this execution while *still* preserving the orderings required for ensuring soundness of the existing mappings in Figure 5. More specifically, the PTX and the ARM-PTX model must have enough ordering constraints such that if a scoped-C/C++ model forbids an execution by an ms ∩ psc ordering, then the mapped execution in PTX and ARM-PTX is also forbidden.

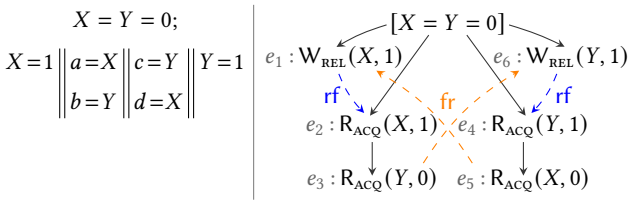


Figure 8. IRIW-RA program (with release-writes and acquire-reads) and its $a{=}c{=}1$, $b{=}d{=}0$ execution. (All sys-scoped)

4.6 Insight 3: A Scope-Aware Ordering for SC Fences

We now address a parallel challenge: how to correctly relate sc-fences (F_{sc}). A naive,"scope-blind" rule—e.g., one that imposes a total order on all F_{sc} fences—is too strong and does not match the hardware. We show this with the IRIW-F (fence-based) example in Figure 9 where the observed behavior on PTX hardware depends entirely on the scope k of the fences:

- If k = sys or k = gpu, the execution is forbidden.
- If k = cta, two scenarios are possible:

(1) If the threads $\langle T_2, T_3 \rangle$ are in the same CTA, the execution is forbidden.
(2) If they are in different CTAs, the execution is allowed.

A naive model that orders all F_{sc} fences would incorrectly forbid case (2). Our model *must* be precise enough to allow this empirically observed behavior.

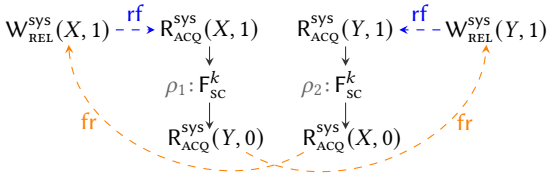


Figure 9. Execution graph of IRIW-F program with outcome $a{=}c{=}1$, $b{=}d{=}0$.

Instead of adopting the primitive sc relation from the original PTX model, we construct a new sound F_{sc}-ordering from first principles. We adopt a weaker variant of pscf from the scoped-C/C++ model (Figure 5) itself; specifically, we consider only the contained morally-strong rf, fr, and co edges – this is sufficient to ensure soundness. By requiring that our final SC ordering includes the (ms ∩ pscf) relation, we precisely capture the hardware's behavior. In the example of Figure 9:

- Case (1) – same CTA: The threads are in the same scope, so the fences ρ_1 and ρ_2 are morally-strong. This creates a (ms∩ pscf) cycle between ρ_1 and ρ_2, forbidden by our (SC) axiom.
- Case (2) – different CTAs: The threads are in different CTAs, so the fences are not morally-strong. So, the execution contains no (ms∩pscf) cycle, correctly allowing the execution.

Thus, adopting (ms∩pscf) is a key part of our model, ensuring soundness, yet weak enough to capture hardware behavior.

4.7 Insight 4: Write Propagation

The ordering rules identified so far—namely, including morally strong co and fr (from Section 4.4), and the scope-aware fence relation (ms ∩ pscf) (from Section 4.6)—do not suffice to preserve the scoped-C/C++ to PTX mapping. Consider the program in Figure 10a. The scoped-C/C++ execution with outcome $a{=}c{=}1$, $b{=}d{=}0$ in scoped-C/C++ has a ms ∩ psc cycle in Figure 10b, and thus is forbidden:

$$e_1 \xrightarrow{\text{hb}_{=\text{loc}}} e_2 \xrightarrow{\text{po}} e_3 \xrightarrow{\text{hb;fr}} e_8 \xrightarrow{\text{hb}_{=\text{loc}}} e_5 \xrightarrow{\text{po}} e_6 \xrightarrow{\text{hb;fr}} e_1$$

To ensure the correct mapping, our PTX and ARM-PTX models must also forbid this execution in the target program. The fences are cta-scoped and are in different CTAs. Hence, the (ms ∩ pscf) relation from Section 4.6 does not apply.

To derive the required ordering, we must formally model this "propagation" of writes, but without naively adding full causality (which Section 4.5 showed is too strong). We follow the architectural intuition: the write e_1 ($W^{sys}_{sc}(X, 1)$) from T_1 propagates to T_2, and is observed before e_3 (F^{cta}_{sc}) in T_2. That fence e_3 flushes this propagated write to all of the threads within the scope. Hence, this write is propagated before the fence e_3's (ms ∩ hb;fr) successor: e_8 ($W^{sys}_{sc}(Y, 1)$). Thus, e_1 in T_1 propagates before e_8 in T_4. By the same argument, e_8 in T_4 propagates before e_1 in T_1. These propagations create a cycle which we must forbid in PTX and ARM-PTX.

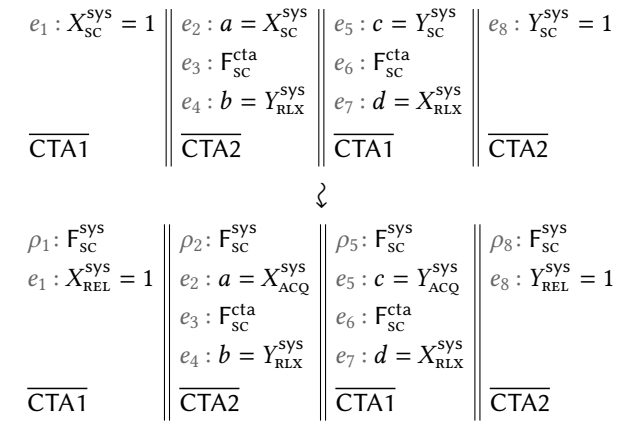
The ordering of writes following the propagation order along hb combined with other relations aligns with architectural intuition, as discussed in Section 4.1.

5 Formalizing the PTX and ARM-PTX Models

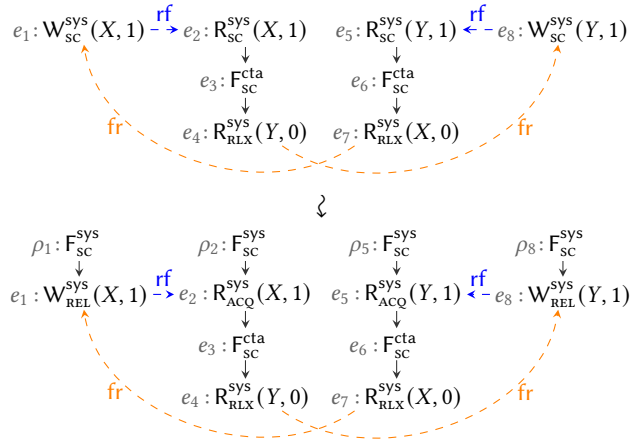
We now assimilate the insights from the previous section to formally define our new PTX model, and then compose it with the existing ARM model to define the sound ARM-PTX model.

5.1 Proposed PTX Model

We define the modified formal model for PTX in Figure 11. We remove the original, flawed sc fence-ordering relation; a key difference from the existing model in Figure 4b [25]. We



(a) Program mapped from scoped-C/C++ to PTX



(b) Corresponding executions for outcome a=c=1, b=d=0.

Figure 10. Program mapped from scoped-C/C++ to PTX with its executions, with F^{cta}_{sc}s in different CTAs.

replace the original (FenceSC) axiom with our new (SC) axiom defined on our unified propagation relation, prop, which is a culmination of the insights from Section 4.

Relations. We define the relations for our new model in Figure 11. First, we have removed the ∪sc component from the sw relations. The core of our new model is the unified propagation relation (prop), which is "home" for all the ordering constraints we derived. The formal definition is composed of several cases that capture the full propagation chain:

- *Case 1a.* (ms ∩ ([F_{sc}];hb);(ms ∩ ca)) links a F_{sc} to a hb-successor that propagates before a W-successor, following a subsequent co or fr edge – the F_{sc} and W being morally strong (Insight 4).
- *Case 1b.* (ms ∩ ca) captures the subtle ordering demanded by morally-strong co and fr edges (Insight 1).
- *Case 2.* (ms ∩ pscf) links the scope-aware sc-fences, covering the case where the source F_{sc} chains to another morally-strong F_{sc} (Insight 3).

Definitions

prel ≜ ([$W_{\sqsupseteq REL}$];$po_{=loc}^{?}$;[W]) ∪ ([$F_{\sqsupseteq REL}$];po;[W])

pacq ≜ ([R];$po_{=loc}^{?}$;[$R_{\sqsupseteq ACQ}$]) ∪ ([R];po;[$F_{\sqsupseteq ACQ}$])

obs ≜ (ms ∩ rf) ∪ (obs;rmw;obs)

sw ≜ (ms ∩ (prel;obs;pacq)) ~~∪ sc~~

eco ≜ (ms ∩ (rf ∪ co ∪ fr))$^{+}$

hb ≜ (po ∪ sw)$^{+}$

ca ≜ co ∪ fr

pscf ≜ [F_{sc}];hb;eco;hb;[F_{sc}]

prop ≜ (ms ∩ ([F_{sc}];hb)$^{?}$;(ms ∩ ca)) ∪ (ms ∩ pscf)
∪ hb;[$E_{\sqsupseteq REL}$] ∪ hb$^{?}$;sw

Axioms

hb;eco$^{?}$ is irreflexive.	~~(Causality)~~ (Coherence)
((ms ∩ fr);(ms ∩ co)) ∩ rmw is empty.	(Atomicity)
($po_{=loc}$ ∪ (ms ∩ (rf ∪ co ∪ fr))) is acyclic.	(SC-per-loc)
(rf ∪ dep) is acyclic.	(No-Thin-Air)
prop is acyclic.	~~(FenceSC)~~ (SC)

Figure 11. New PTX model. Changes from the original are marked with green boxes.

- *Case 3.* (hb;[$E_{\sqsupseteq REL}$]) links events along a hb-chain *leading up to* a flushing event ⊒REL (Insight 4).
- *Case 4.* (hb$^{?}$;sw) captures the propagation *from* that flushing event ⊒REL via a sw edge (Insight 4).

While cases 1a and 2 both start with a fence, pscf also *ends* in one, allowing it to order across weaker intermediate edges (*e.g.*, $W_{RLX} \xrightarrow{rf} R_{RLX}$) on the path. Appendix C.2 further explains why each prop-case is needed for the scoped-C/C++ mapping.

Axioms. We preserve three axioms from the original model. Our key changes are twofold:

- We replace the original (Causality) axiom with an updated (Coherence) axiom, where hb;eco$^{?}$ is irreflexive.
- We replace the original (FenceSC) axiom, defined on the old sc relation, with our new (SC) axiom, where prop must be acyclic. This is the cornerstone of our solution.

5.2 ARM-PTX

We now define the final heterogeneous ARM-PTX model by composing our new PTX model with the standard ARM model. In the ARM-PTX model, an execution contains both ARM and PTX events, and relations between them. This composition, defined in Figure 12, involves "globalizing" the key relations to handle inter-processor synchronization and ordering.

Definitions

$\text{posc}^{\text{ARM}} \triangleq [\text{F}_{\text{MM}}];\text{po} \cup \text{po};[\text{F}_{\text{MM}}] \cup [\text{R}_\text{A} \cup \text{R}_\text{Q}];\text{po} \cup \text{po};[\text{W}_\text{L}] \cup [\text{W}_\text{L}];\text{po};[\text{R}_\text{A}]$

$\text{posc}^{\text{PTX}} \triangleq [\text{E}_{\sqsupseteq\text{REL}}];\text{po} \cup \text{po};[\text{E}_{\sqsupseteq\text{REL}}]$

$\text{gprel} \triangleq [\text{W}_{\sqsupseteq\text{REL}} \cup \text{W}_\text{L}];\text{po}_{=\text{loc}}{}^{?};[\text{W}] \cup [\text{F}_{\sqsupseteq\text{REL}} \cup \text{F}_{\text{MM}}];\text{po};[\text{W}]$

$\text{gpacq} \triangleq [\text{R}];\text{po}_{=\text{loc}}{}^{?};[\text{R}_{\sqsupseteq\text{ACQ}} \cup \text{R}_{\sqsupseteq\text{Q}}] \cup [\text{R}];\text{po};[\text{F}_{\sqsupseteq\text{ACQ}} \cup \text{F}_{\sqsupseteq\text{RM}}]$

$\text{gsw} \triangleq \text{ms} \cap (\text{gprel};\text{obs};\text{gpacq})$

$\text{ghb} \triangleq (\text{gob} \cup \text{posc} \cup \text{po}^{?};\text{gsw};\text{po}^{?})^{+}$
where $\text{gob} \triangleq \text{rfe};\text{ob}$ and $\text{posc} \triangleq \text{posc}^{\text{ARM}} \cup \text{posc}^{\text{PTX}}$

$\text{gprop} \triangleq (\text{ms} \cap ([\text{F}_{\text{SC}} \cup \text{F}_{\text{MM}}];\text{ghb})^{?};(\text{ms} \cap \text{ca})) \cup (\text{ms} \cap \text{gpscf}) \cup \text{ghb};[B] \cup (\text{po} \cup \text{gsw})^{*};\text{gsw}$
where $B \triangleq \text{E}_{\sqsupseteq\text{REL}} \cup \text{W}_\text{L} \cup \text{F}_{\text{MM}} \cup \text{R}_\text{A}$

$\text{gpscf} \triangleq [\text{F}_{\text{SC}} \cup \text{F}_{\text{MM}}];\text{ghb};\text{eco};\text{ghb};[\text{F}_{\text{SC}} \cup \text{F}_{\text{MM}}]$

Axioms

$(\text{ghb};\text{eco}^{?})$ is irreflexive.	(Coherence)
$((\text{ms} \cap \text{fr});(\text{ms} \cap \text{co})) \cap \text{rmw}$ is empty.	(Atomicity)
$(\text{po}_{=\text{loc}} \cup (\text{ms} \cap (\text{rf} \cup \text{co} \cup \text{fr})))$ is acyclic.	(SC-per-loc)
$(\text{rf} \cup \text{dep} \cup \text{dob})$ is acyclic.	(No-Thin-Air)
gprop is acyclic.	(SC)

Figure 12. Our ARM-PTX Memory model.

Definitions. We create global versions of the relations, such as gsw and ghb, which combine the respective relations from both models. Crucially, our final unified propagation ordering, gprop, is built using the same principles we used to derive prop in the previous sections.

We define the global release and acquire patterns, gprel and gpacq, based on the release (REL), acquire (ACQ), or stronger (SC) accesses in PTX, and release (L), acquirePC (Q), acquire (A), or stronger (SC) accesses in ARM. Based on gprel and gpacq relations, we define the global-synchronizes-with (gsw) relation that can synchronize ARM and PTX events.

We combine the orderings in ARM and PTX to define the global-happens-before (ghb) relation. Relation ghb consists of the ordering relations in ARM and PTX, posc^{ARM} and posc^{PTX} respectively, along with their interactions by gsw and global-ob relation (gob). The posc^{ARM} orders a full fence with its po-predecessor and po-successor events, acquire (R_A) and acquirePC (R_Q) events with their po-successors, release writes (W_L) with their po-predecessors, and release-write (W_L) with a po-successor acquire-read event (R_A). The posc^{PTX} orders a release or stronger PTX event with its po-predecessor and po-successor events; note that those po-orders appear only inside ghb, and thus do not enforce those po-orders beyond cases including ghb. The global-ob relation extends the ob relation in ARM with the rfe relation from PTX write events in order to effectively "bridge" to PTX. If event pair $\langle r, a\rangle$ in ARM is in ob relation and event r reads from a PTX write event w, then $w \xrightarrow{\text{gob}} a$ holds.

Our global-pscf relation (gpscf) orders full fences in ARM and PTX. We obtained gpscf from scoped-C/C++, uniquely allowing us to consider the scopes of fences, as gpscf only orders when morally-strong (ms) – no other definition captures this.

Finally, we define the global-prop relation (gprop) to capture propagation-based ordering between ARM and PTX events based on the intermediate events in set B. Relation gprop judiciously combines ghb, gpscf, co, and fr to establish the order for a non-MCA memory model in the presence of B events without forbidding the non-MCA executions observed on real hardware – similar to prop in PTX. The gprop relation preserves the order between the accesses generated from scoped-C/C++ sc accesses, ensuring the mapping schemes remain correct when mapping scoped-C/C++ to ARM-PTX.

Axioms. The ARM-PTX axioms are similar to those of PTX, with global orderings when applicable. For instance, the (Coherence) axiom is defined based on ghb relation. The (Atomicity), (SC-per-loc) axioms remain the same. The (No-Thin-Air) axiom includes dependency ordering restrictions, dep and dob, in PTX and ARM, respectively. The axiom (SC) is defined by the acyclicity of the gprop relation.

5.3 Mapping correctness

Our heterogeneous ARM-PTX model (Figure 12) is sufficiently strong to map from scoped-C/C++, for which we prove the existing mappings stated in Figure 5b (and Appendix B.4). Although those schemes already mapped to ARM, the mapping to the original PTX model was incorrect (Section 4.2). For our new PTX-only model that mapping is now correct and, more importantly, only with our compound model can we correctly combine the mappings to ARM and PTX.

For PTX threads, we split $\text{R}^{\text{sco}}_{\text{SC}}$ and $\text{W}^{\text{sco}}_{\text{SC}}$ events into a $\text{R}^{\text{sco}}_{\text{ACQ}}$ and $\text{W}^{\text{sco}}_{\text{REL}}$ with po_{imm}-preceding F_{SC}, respectively. On ARM threads, those same accesses map to $\text{R}^{\text{sys}}_{\text{A}}$ and $\text{W}^{\text{sys}}_{\text{L}}$. Intuitively, the mapping is correct because it does not introduce any new behavior, formally stated by Theorem 1.

> ▶ **Theorem 1** (Correct Mapping). The mapping scheme (Figure 5b) generates an ARM-PTX program P_t from any given scoped-C/C++ program P_s. That scheme is correct because for every ARM-PTX execution of P_t there exists a scoped-C/C++ execution of P_s with the same behavior.

Proof Overview. Given any ARM-PTX execution of a mapped program, which satisfies its five axioms (Figure 12), we show that any execution of the source scoped-C/C++ execution satisfies its three axioms (Figure 5a). We prove Theorem 1 by contradiction. For each scoped-C/C++ axiom, we show that its violation necessarily violates an ARM-PTX axiom—which

we assumed was satisfied—meaning that the violation is impossible. Here, we give the high-level proof sketch for the most challenging scoped-C/C++ axiom: (SC).

We map scoped-C/C++ 'ms∩psc' to our ARM-PTX gprop, which is disallowed by (SC). We must analyze many cases; notably, the case $[F_{sc}];hb^{?};scb;[R_{sc} \cup W_{sc}] \subseteq psc$ where the final R_{sc} or W_{sc} splits on PTX, is challenging. The order is not preserved when the leading F_{sc} is outside the chain (*e.g.*, as in Figure 7). We must then combine multiple psc edges into a single gprop along the mapping.

Mechanized Proof. We mechanize our proof of Theorem 1 in the Agda proof assistant [1] using the Burrow weak memory proof framework [41], totaling over 4,000 lines.

6 Experimental Evaluation

To understand the implementation of current hardware and to test whether our proposed model is accurate with respect to observed behaviors, we execute 1306 litmus test variants on three NVIDIA GPU architectures and four machines: Ampere (A100), Hopper (H100 and GH200), and Blackwell (GB10). The A100 and H100 are discrete GPUs, while the GH200 and GB10 contain NVIDIA's specialized "Superchip" architecture, which integrates the GPU with an ARM-based CPU.

Our testing campaign focuses exclusively on GPU-only (PTX) executions, even on the heterogeneous GH200 and GB10. This is a deliberate methodological choice. Our formal analysis in Section 2 demonstrated that the soundness failures in both the homogeneous PTX case and the heterogeneous ARM-PTX case stemmed from the exact same root cause: the original PTX model treating sc-fences specially, unlike other primitives. The heterogeneous failure was merely a symptom of this subtle PTX flaw. Therefore, the most critical empirical questions are:

1. Do the existing compilation of the flawed scoped-C/C++ programs show weak behaviors?
2. Does the PTX hardware actually support our proposed strengthening (*i.e.* does it enforce co and fr as part of its SC ordering)?
3. Is the hardware actually non-MCA, making our surgical fix necessary?

We answer these questions directly with the PTX-only testing campaign. To be clear, a full-blown cross-processor litmus testing showing our proposed globalizations of the causality and the SC relations are in fact sound with respect to hardware is a useful research endeavor. However, developing a robust, cross-processor (CPU+GPU) litmus test harness that can reliably expose weak behavior is a significant engineering challenge for future work, and our PTX-only validation is sufficient to confirm the central thesis of our paper. Our experimental evaluation reveals three key insights:

1. None of the litmus tests showing flaws in the existing heterogeneous compiler composition (Figure 2) or homogeneous PTX compilation (Figure 3) exhibit the specified weak behaviors when running all threads on the GPU, meaning that GPU hardware today is likely strong enough to support our sound model.
2. Running litmus tests such as **2+2w**, **3.2w**, and **z6-3** show empirically that hardware does enforce SC-ordering for the co and fr relations, validating the strengthening of the PTX memory model.
3. The more recent NVIDIA Hopper architecture H100 GPU does exhibit non-multi-copy atomic (MCA) behaviors even when memory accesses are as strong as release and acquire, showing that the surgical strengthening of the model is necessary to both restore soundness and allow behaviors visible on hardware.

Table 1. Number of variants generated for each litmus test (total of 1306 variants).

Paper Examples		co / fr Tests		MCA Tests	
Test	**Variants**	**Test**	**Variants**		
Figure 2c	2	**2+2w**	32	**iriw**	240
Figure 3	2	**3.2w**	160	**isa2**	230
		z6-3	160	**rwc**	110
				wrc	110
				wrw+2w	110
				wwc	150

6.1 Testing Setup

Litmus tests are small, concurrent programs that can be used to show empirically MCM behaviors on real hardware, both for CPUs [3] and GPUs [2]. We generate a suite of litmus tests to check whether existing hardware conforms to our model; specifically, our tests are either the examples highlighting deficiencies in existing models, or variants of litmus tests described in prior work [37]. These existing tests are used either to check whether co and fr preserve SC-ordering (**2+2w**, **3.2w**, **z6-3**), or check whether hardware is MCA (**iriw**, **wrc**, **isa2**, **rwc**, **wwc**, **wrw+2w**)[2].

Each litmus test base is a *template* that we instantiate in several ways. First, we systematically generate variants of all six tests by enumerating combinations of memory orders that exhibit certain characteristics of the model proposed in this paper. For tests like **2+2w** or **z6-3**, this means strengthening memory orders to REL in order to show that co and fr are empirically part of SC-ordering. For MCA tests, this means strengthening memory orders up to but not including sc-fences, to determine whether the model still allows non-MCA behaviors under relatively strong REL and ACQ memory orders. As an example, for **iriw** the first read on each thread can be ACQ, or left as RLX with an AQRL fence inserted between the two reads.

Next, for each base variant, we generate a set of scoped variants with a scope of either cta or gpu, with the constraint that all accesses are at the same scope sco_a and all fences are at the same scope sco_b. Additionally, we generate variants for all

[2]We show each of these litmus tests in Figure 18 in the appendix.

possible combinations of cooperative thread array groupings. For example, **iriw**, which consists of four threads, has 15 possible groupings, *e.g.,* all threads in the same CTA, all threads in separate CTAs, etc. Along with the strengthened versions of each test, we include *relaxed* variants which contain no fences and only relaxed memory orders. Overall, our systematic generation leads to 1306 litmus test variants, with the breakdown per test shown in Table 1. To verify the intended behavior, we checked each variant against both the published Alloy PTX model [25] and an Alloy version of our sound PTX model. We find that the sound PTX model disallows more variants (108 vs. 98 in the original PTX model), and no disallowed variants had observable weak behaviors on hardware[3].

Along with these tests, we also compared whether the existing PTX and our proposed model allows weak behaviors for 96 variants of three common litmus tests found in the literature: message passing, load buffer, and store buffer. Previous work has shown that these tests' weak behaviors commonly occur on GPUs [22]. This comparison showed that the models are equivalent with respect to these variants of the load buffer and message passing tests, but our model allows one more variant of store buffer than the original PTX model. This test is shown in Figure 17 and discussed in Appendix C.3.

6.2 Testing Techniques and Efficacy

Naive executions of litmus tests, *e.g.,* simply running one set of threads that execute only the litmus test instructions, are unlikely to show weak behaviors even if they are possible on the hardware. Previous work on GPU litmus testing has explored various techniques that stress the memory system to increase the chances of seeing weak behaviors [19, 23]. We port these stressing techniques and adapt them to support the new types of litmus tests run in this paper, including our three and four-thread SC-ordering and MCA tests which previous GPU work has not tested thoroughly.

Along with stressing techniques, we also adopt a *mutation testing* approach from previous work, which allows us to evaluate the efficacy of the techniques [23]. Under this approach, litmus test variants that allow weak behaviors are run first in order to find effective stress configurations, after which the strengthened variants are run using these configurations. We apply this approach using the 106 relaxed variants of the litmus tests in Table 1. We observed weak behaviors for 70 of these variants on all GPUs except the GB10, where one variant of Figure 2c with all relaxed accesses was not observed. The other 36 relaxed variants where no weak behaviors were observed either run all or certain sets of threads in the same CTA, indicating that the conditions which lead to reordering at this level of the cache hierarchy occur more rarely and are harder to trigger.

[3]A detailed breakdown of allowed and visible tests is shown in Table 2 in the appendix.

6.3 Testing Campaign and Insights

We run our litmus test variants on the A100, H100, and GB10 for 24 hours each, and on the GH200 GPU for 6 hours (due to difficulty in getting access to one), leading to a total of over 10 trillion test executions and over 700 million weak behavior observations. Based on these observed results, we discuss three main insights:

Insight A: Behaviors allowed due to deficiencies in current models are not observable. The litmus test described in Figure 3 shows a behavior that is disallowed in scoped-C/C++ but allowed in the compiled PTX program using the leading-fence mapping. However, our testing campaign revealed no observations of this behavior, indicating that the flaw is likely not in the compilation strategy or in the implementation of the hardware, but rather in the PTX model itself. Therefore, strengthening the PTX model and disallowing this behavior in the program accurately represents real hardware while restoring soundness to scoped-C/C++ to PTX compilation schemes.

Similarly, we test the litmus test shown in Figure 2c with one small difference—we elide the SC fence in T4 to emulate the heterogeneous context (recall that an actual heterogeneous test would lack the SC fence if T4 were an ARM thread.). We observe that the behavior shown is indeed disallowed. This shows that the current NVIDIA hardware already disallows the outcome without the crutch of SC fences, instead relying on co and fr edges, vindicating the core hypothesis of the paper. Next, we independently test whether co and fr relations are empirically part of SC-ordering.

Insight B: co and fr relations are empirically part of SC-ordering. The **2+2w** and **3.2w** litmus tests, with memory orders strengthened to REL and ACQ, check whether writes can be propagated in ways that break the SC-reasoning necessary for defining our sound model. The **z6-3** litmus test checks whether this SC-ordering composes with hb-ordering, which was enforced in an initial version of our model but relaxed when it became clear it was not necessary to support sound compilation. On all of the machines we tested, we did not observe weak behaviors when running strengthened variants disallowed under our model, empirically validating the inclusion of co and fr in SC-ordering in our new PTX model. Importantly, initial versions of the strengthened model disallowed weak behaviors in variants of all three of these tests which *were* observed on the machines we tested, leading the model to be refined so it was weak enough to accurately represent the behavior of real hardware. Similarly, weak behaviors in strengthened **z6-3** variants were not observed despite them being allowed by the model, showing that there is still room to develop refined models that more closely match actual hardware implementations.

Insight C: Non-multi-copy atomic behaviors are observable. For the MCA litmus tests, weak behaviors are extremely common when all memory orders are relaxed. However, unlike in the ARM memory model, where the execution shown in

Figure 8 (with accesses strengthened to REL/ACQ) is disallowed due to the strong MCA requirements, it *is* observed on the H100. These observations are extremely rare; in our test campaign they occurred in less than one in a million executions, even with the effective stress configurations that readily cause weak behaviors with relaxed accesses. This rarity highlights the importance in both carefully defining the PTX memory model and in rigorously testing it, as behaviors occurring this rarely could be difficult to diagnose in an application relying on an incorrectly strengthened memory model.

7 Related Work

Compiler mappings of weak memory primitives. Correct compilation of C/C++ concurrency primitives to architectures like x86, Power, and ARM (Armv7, Armv8 or later) is a widely studied problem [4, 8, 10, 11, 18, 21, 33, 34, 36–40]. A particularly subtle issue is the mapping of C/C++ Sequentially Consistent (SC) accesses. Manerkar et al. [26] identified key miscompilation errors for SC on ARM and Power. This led to foundational work by Lahav et al. [20] on the complexities of SC mappings, which proposed the "leading-fence" and "trailing-fence" schemes and helped evolve the C/C++ language semantics to enable provably correct mappings [7].

For GPU architectures, this body of work was extended by Lustig et al. [25], who applied the "leading-fence" mapping to the NVIDIA PTX model and provided a mechanized proof of its soundness. These fence-splitting mappings form the PTX ABI for C/C++ [30]. We demonstrated the mapping relies on an unproven fence-splitting transformation—converting SC accesses to a fence-plus-weaker-access—which we show is unsound in the presence of scopes.

Heterogeneous memory models. Heterogeneous memory consistency models result from the interaction of multiple concurrency models via shared memory. Oswald et al. [32] propose a heterogeneous memory model that combines MCA memory models. In contrast, we combine an MCA (ARM) and a non-MCA (PTX) model, which is significantly more challenging. Other work has combined C/C++ with x86 non-temporal accesses [12] and x86 with FPGAs [17].

The most closely related work is from Goens et al. [15], who compose x86 and PTX models. They propose both operational and axiomatic heterogeneous models and, based on their axiomatic model, a trailing-fence mapping for scoped-C/C++. However, their x86-PTX model achieves soundness by treating x86 reads as full fences, creating an explicit sc order between PTX fences and x86 reads. While this "sledgehammer" approach achieves soundness, this design choice unnecessarily strengthens the model, as noted in Section 4.3. In contrast, our ARM-PTX model is surgically developed (taking inspiration from Goens et al. [15] operational approach), to restore soundness *without* being overly strong, allowing it to match the observable behaviors of real hardware.

Models alternative to PTX are the *Heterogeneous Race Free* (HRF) lineage [13, 16] and their adoption in HSA [31]. "Morally strong" is NVIDIA's PTX terminology for its scope-matching; scoped-C/C++ and HRF-Relaxed use a *scope-inclusion* relation for the same purpose. The distinguishing feature of PTX is therefore not this predicate but its special treatment of sc-fences. The HRF lineage also sits at a different layer: it is a language model and does not introduce a (virtual-)ISA GPU model, nor does it compile to such an ISA model (GPU-only or CPU-GPU). Instead it reasons about hardware directly enforcing the language-level model – an approach orthogonal to NVIDIA's. The flaw therefore does not surface in the HRF lineage, but the lesson is general: any (virtual-)ISA-level GPU model must be careful how it defines SC ordering.

8 Conclusion

We identified a subtle error in the official soundness argument for compiling scoped-C/C++ to PTX, and showed that the same root cause of treating sc-fences differently from other primitives breaks heterogeneous ARM-PTX compilation. We resolved both problems with a single solution: grounding SC ordering in scope-aware coherence and from-read relations rather than exclusively in fences. Our model preserves non-MCA behaviors, respects scope boundaries, and imposes no requirements beyond what A100, H100, GH200, and GB10 hardware already enforces. As the error appears only in subtle cross-scope communication patterns, most programs are unaffected in practice, which explains why it remained undetected for almost a decade. That also highlights the unique strength of our mechanized proofs to uncover precisely those subtle cases.

While our concrete results target PTX and ARM-PTX, the underlying lesson is general. Defining the SC order exclusively through a total order on sc-fences is fundamentally fragile in two settings that are increasingly prevalent. First, in scoped models, moral strength between events is not transitive; a chain of pairwise morally-strong sc accesses does not imply that the fences guarding them are mutually morally strong. Considering only the sc-fences does not capture such chains. Second, in heterogeneous systems, different ISAs may implement SC guarantees through entirely different mechanisms—access modes rather than fences—making sc-fence ordering insufficient for cross-ISA ordering. These are not incidental properties of PTX; they are inherent limitations of specially treating sc-fences in scoped or heterogeneous context. Our solution—grounding SC ordering in coherence and from-read relations rather than purely a total order on fences—addresses the root cause and provides a blueprint for sound memory model design in future scoped and heterogeneous systems.

Acknowledgements

We thank Daniel Lustig and the other members of the NVIDIA weak memory team for confirming the problem and for their feedback on our early draft. This work is supported in part by the National Science Foundation under grant CCF-2525270, and ARM. Generative AI tools were used to copyedit the text.

References

[1] Agda Team. 2025. Agda Documentation. https://agda.readthedocs.io/en/stable/.

[2] 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 *ASPLOS '15*. doi:10.1145/2694344.2694391

[3] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2011. Litmus: Running Tests Against Hardware. In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Part of the Joint European Conferences on Theory and Practice of Software (TACAS'11/ETAPS'11)*. https://dl.acm.org/doi/10.5555/1987389.1987395

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

[5] ARM. 2021. The AMBA CHI Specification. https://developer.arm.com/architectures/system-architectures/amba/amba-5. Accessed: 5th July 2022.

[6] Arm 2026. C/C++ Atomics Application Binary Interface Standard for the Arm® 64-bit Architecture. https://github.com/ARM-software/abi-aa/blob/main/atomicsabi64/atomicsabi64.rst.

[7] Mark Batty, Alastair F. Donaldson, and John Wickerson. 2016. Overhauling SC Atomics in C11 and OpenCL. In *POPL '16*. ACM, 634–648. doi:10.1145/2837614.2837637

[8] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *POPL'11*. ACM, 55–66. doi:10.1145/1926385.1926394

[9] Hans-J. Boehm and Brian Demsky. 2014. Outlawing Ghosts: avoiding Out-of-thin-air Results. In *MSPC'14* (Edinburgh, United Kingdom). ACM, Article 7, 6 pages. doi:10.1145/2618128.2618134

[10] Soham Chakraborty and Viktor Vafeiadis. 2017. Formalizing the Concurrency Semantics of an LLVM Fragment. In *CGO '17*. IEEE, 100–110. doi:10.1109/CGO.2017.7863732

[11] Soham Chakraborty and Viktor Vafeiadis. 2019. Grounding Thin-Air Reads with Event Structures. 3, POPL (2019). doi:10.1145/3290383

[12] Paulo Emílio de Vilhena, Ori Lahav, Viktor Vafeiadis, and Azalea Raad. 2024. Extending the C/C++ Memory Model with Inline Assembly. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024). doi:10.1145/3689749

[13] Benedict R. Gaster, Derek Hower, and Lee Howes. 2015. HRF-Relaxed: Adapting HRF to the Complexities of Industrial Heterogeneous Memory Models. *ACM Trans. Archit. Code Optim.* 12, 1, Article 7 (April 2015), 26 pages. doi:10.1145/2701618

[14] Luke Geeson, James Brotherston, Wilco Dijkstra, Alastair F. Donaldson, Lee Smith, Tyler Sorensen, and John Wickerson. 2024. Mix Testing: Specifying and Testing ABI Compatibility of C/C++ Atomics Implementations. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 287 (Oct. 2024), 26 pages. doi:10.1145/3689727

[15] Andrés Goens, Soham Chakraborty, Susmit Sarkar, Sukarn Agarwal, Nicolai Oswald, and Vijay Nagarajan. 2023. Compound Memory Models. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1145–1168. doi:10.1145/3591267

[16] Derek R. Hower, Blake A. Hechtman, Bradford M. Beckmann, Benedict R. Gaster, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. 2014. Heterogeneous-race-free Memory Models. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 427–440. doi:10.1145/2541940.2541981

[17] Dan Iorga, Alastair F. Donaldson, Tyler Sorensen, and John Wickerson. 2021. The Semantics of Shared Memory in Intel CPU/FPGA Systems. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–28. doi:10.1145/3485497

[18] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-Memory Concurrency. In *POPL'17* (Paris, France) *(POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 175–189. doi:10.1145/3009837.3009850

[19] Jake Kirkham, Tyler Sorensen, Esin Tureci, and Margaret Martonosi. 2020. Foundations of Empirical Memory Consistency Testing. *Proc. ACM Program. Lang.* (2020). doi:10.1145/3428294

[20] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++11. In *PLDI 2017*. 618–632. doi:10.1145/3062341.3062352 Technical Appendix Available at https://plv.mpi-sws.org/scfix/full.pdf.

[21] Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. 2020. Promising 2.0: Global Optimizations in Relaxed Memory Concurrency. In *PLDI 2020*. 362–376. doi:10.1145/3385412.3386010

[22] Reese Levine, Mingun Cho, Devon McKee, Andi Quinn, and Tyler Sorensen. 2023. GPUHarbor: Testing GPU Memory Consistency at Large (Experience Paper). In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. Association for Computing Machinery. doi:10.1145/3597926.3598095

[23] Reese Levine, Tianhao Guo, Mingun Cho, Alan Baker, Raph Levien, David Neto, Andi Quinn, and Tyler Sorensen. 2023. MC Mutants: Evaluating and Improving Testing for Memory Consistency Specifications. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS 2023)*. doi:10.1145/3575693.3575750

[24] Daniel Lustig, Simon Cooksey, and Olivier Giroux. 2022. Mixed-Proxy Extensions for the NVIDIA PTX Memory Consistency Model: Industrial Product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 1058–1070. doi:10.1145/3470496.3533045

[25] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. 2019. A Formal Analysis of the NVIDIA PTX Memory Consistency Model. In *ASPLOS 2019*. 257–270. doi:10.1145/3297858.3304043

[26] 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* abs/1611.01507 (2016).

[27] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2020. *A Primer on Memory Consistency and Cache Coherence, Second Edition*. Morgan & Claypool Publishers. doi:10.2200/S00962ED2V01Y201910CAC049

[28] NVIDIA. 2018. Parallel Thread Execution ISA Version 6.0. https://docs.nvidia.com/cuda/archive/9.0/parallel-thread-execution/index.html

[29] NVIDIA. 2019. CUDA Toolkit Documentation - PTX ISA. https://docs.nvidia.com/cuda/parallel-thread-execution/

[30] NVIDIA Corporation. 2026. CUDA Toolkit Documentation – Atomics Application Binary Interface. https://docs.nvidia.com/cuda/ptx-writers-guide-to-interoperability/atomic-abi.html.

[31] Marc S. Orr, Shuai Che, Ayse Yilmazer, Bradford M. Beckmann, Mark D. Hill, and David A. Wood. 2015. Synchronization Using Remote-Scope Promotion. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 73–86. doi:10.1145/2694344.2694350

[32] Nicolai Oswald, Vijay Nagarajan, Daniel J. Sorin, Vasilis Gavrielatos, Theo Olausson, and Reece Carr. 2022. HeteroGen: Automatic Synthesis of Heterogeneous Cache Coherence Protocols. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2022, Seoul, South Korea, April 2-6, 2022*. IEEE, 756–771. doi:10.1109/HPCA53966.2022.00061

[33] Gustavo Petri, Jan Vitek, and Suresh Jagannathan. 2015. Cooking the Books: Formalizing JMM Implementation Recipes. In *ECOOP 2015*, Vol. 37. 445–469. doi:10.4230/LIPIcs.ECOOP.2015.445

[34] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2019. Bridging the Gap between Programming Languages and Hardware Weak Memory Models. *Proc. ACM Program. Lang.* 3, POPL (2019). doi:10.1145/3290382

[35] 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. *PACMPL* 2, POPL (2018), 19:1–19:29. doi:10.1145/3158107

[36] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. 2012. Synchronising C/C++ and POWER. In *PLDI'12*. ACM, 311–322. doi:10.1145/2254064.2254102

[37] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER Multiprocessors. In *PLDI '11*. 175–186. doi:10.1145/1993498.1993520

[38] Jaroslav Sevcík. 2011. Safe optimisations for shared-memory concurrent programs. In *PLDI 2011*. 306–316. doi:10.1145/1993498.1993534

[39] Jaroslav Sevcík and David Aspinall. 2008. On Validity of Program Transformations in the Java Memory Model. In *ECOOP 2008*. 27–51. doi:10.1007/978-3-540-70592-5_3

[40] Peter Sewell. 2022. C/C++11 mappings to processors. https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html. Accessed April 5, 2023.

[41] Dennis Sprokholt and Soham Chakraborty. 2026. Burrow: A Proof Framework for Weak Memory. In *Computer Aided Verification (CAV 26)*. Springer Nature Switzerland, 213–227. doi:10.1007/978-3-032-32526-6_10

This is the appendix of the paper "Beyond the Fence: Sound Compilation for GPU and Heterogeneous CPU+GPU Systems".

Table of contents

- Appendix A describes our public artifact with the proofs, Alloy litmus tests, and hardware litmus tests.
- Appendix B provides the existing full formal model of ARM and PTX [25, 35], with mapping schemes from scoped-C/C++.
- Appendix C expands on the translation bug from scoped-C/C++ to PTX. We also give a detailed justification for each of our prop cases.
- Appendix D provides the litmus tests and further details on experimental evaluation.

A Artifact Appendix

A.1 Abstract

Our artifact consists of *three* components, each corresponding to a different result in our paper: ① we claim the compilation of scoped-C/C++ to our ARM-PTX model is correct (Theorem 1 in Section 5.3), for which the artifact contains mechanized proofs in Agda; ② we identify the weak behaviors for a suite of litmus tests, which are contained in the artifact for Alloy; and ③ we claim our model matches NVIDIA GPU behavior (Section 6), for which the artifact contains hardware litmus tests with testing infrastructure.

A.2 Artifact check-list (meta-information)

- **Mechanized Proofs:** Agda proofs for compilation correctness over axiomatic weak memory ①.
- **Program:** A suite with PTX litmus tests for Alloy ② and another suite of kernels for GPU hardware ③.
- **Model:** Our ARM-PTX weak memory model is formalized in Agda for the proofs ① and our PTX model in Alloy ②.
- **Hardware:** Our original hardware experiments ③ were produced on three NVIDIA GPU microarchitectures: Ampere (A100), Hopper (H100 and GH200), and Blackwell (GB10).
- **Output:** Each component has its own output:
  - ① Running Agda on the proofs checks them without errors. Separately, they can be rendered to HTML to inspect them.
  - ② The Alloy litmus tests produce a `.csv` file stating for each litmus test whether it is observable under our PTX model.
  - ③ The hardware experiments produce another `.csv` files with litmus behaviors observed on hardware.
- **How much disk space required (approximately)?:** <400MB
- **How much time is needed to prepare workflow (approximately)?:** ~ 30 minutes
- **How much time is needed to complete experiments (approximately)?:** The Agda proofs check within 5 minutes ①, the Alloy litmus tests take 1-2 hours ②, the reduced hardware tests take around 1.5 hours ③.
- **Publicly available?:** Yes, the artifact repository is online: `github.com/sourcedennis/beyond-the-fence-artifact` From that repository the Docker image can be built to check the Agda proofs ① and execute the Alloy litmus tests ②. It also contains the hardware experiments ③ that must run outside Docker.
- **Code licenses (if publicly available)?:** BSD-3
- **Archived?:** `zenodo.org/records/22545540`

A.3 Description

A.3.1 How to access. Clone our artifact repository from GitHub: `github.com/sourcedennis/beyond-the-fence-artifact`. Note that when building the Docker image (with the contained `Dockerfile`), it pulls in the Agda proofs and its dependencies from a separate repository: `github.com/sourcedennis/ptx-proofs`.

A.3.2 Hardware dependencies. Our original hardware experiments ③ were produced on three NVIDIA GPU microarchitectures: Ampere (A100), Hopper (H100 and GH200), and Blackwell (GB10). A representative subset can be reproduced with any one of those. The mechanized proofs ① and Alloy litmus testing ② require no specialized hardware.

A.3.3 Software dependencies. We provide a `Dockerfile` which requires *Docker* to build and run an image with the needed dependencies to check the mechanized proofs ① and run the Alloy litmus tests ②. *Alternatively, running these outside Docker requires*, respectively:

① The proofs require Agda[4] [1] (v2.7.0.1), its standard library (v2.3), and the proof libraries Burrow[5] [41] and Dodo[6].
② Running the Alloy litmus tests requires Alloy[7] (v6.2.0), which requires a Java JRE (v17).

Running the hardware experiments ③ on a machine with NVIDIA GPU requires: Python 3, an NVIDIA driver and CUDA runtime, NVIDIA CUDA Compiler (`nvcc`), and NVIDIA System Management Interface (`nvidia-smi`), through which the NVIDIA GPU must be visible.

A.3.4 Models. Our ARM-PTX axiomatic weak memory model is formalized in Agda ① and our PTX model in Alloy ②.

A.4 Installation

①+② The Agda proofs ① and Alloy litmus tests ② can be packed together in a Docker image. After cloning the GitHub repository, build that image with:

```
docker build . --tag=beyond-the-fence
```

To check whether it built correctly, start the image interactively (which executes `bash`):

```
docker run -it --rm beyond-the-fence
```

③ Copy the '`hw/`' directory (from our GitHub repository) to the machine with the NVIDIA GPU. Make sure the relevant software dependencies are installed (Section A.3.3, point ③), which can be checked with:

```
./scripts/check_env.sh
```

Smoke test. Execute a smoke test with:

```
./scripts/run_smoke.sh \
  --official-machine gb10 \
  --test-environments 1 \
  --iterations-per-test 100
```

The script writes a log to `output/smoke.txt` and prints a comparison summary against our results for the machine named by `--official-machine` (*i.e.* GB10 above). Look out for:

- The CUDA kernels compile without `nvcc` errors.
- The protected Figure 3 test reports '`weak: 0`'.
- '`Unexpected reviewer seen`' should be `0`.
- At least one relaxed or block-scoped test may report '`weak`' greater than zero. This is a useful sanity check, but short runs can miss weak behaviors.

[4] `agda.readthedocs.io/en/latest/getting-started/`
[5] `github.com/sourcedennis/agda-burrow`
[6] `github.com/sourcedennis/agda-dodo`
[7] `alloytools.org`

A.5 Experiment workflow

① *Agda proofs.* On the machine with the Docker image `beyond-the-fence` (from Section A.4), run the following command to check the Agda proofs:

```
docker run --rm \
  -w /home/proof/ptx-proofs \
  beyond-the-fence \
  agda src/Main.agda --safe
```

This takes around *two* minutes. Note that the `--safe` flag ensures the proofs are logically consistent. Without that flag, Agda accepts proofs with postulates, unsolved meta variables, and various other inconsistencies[8].

Separately, to inspect the proofs, they can be rendered to HTML with the following command (which creates and mounts the local directory '`html/`'):

```
mkdir html
docker run --rm \
  -w /home/proof/ptx-proofs \
  -v "$PWD/html:/home/proof/ptx-proofs/html" \
  beyond-the-fence \
  agda --html --html-dir=html src/Main.agda
```

Notably, the file `Main.agda` (rendered to `Main.html`) links the proofs together. `ProofSplit.agda` (rendered to `ProofSplit.html`) is our key result, showing the leading fence splitting on PTX threads from scoped-C/C++, of which `ProofSplit/Consistent.agda` (rendered to `ProofSplit.Consistent.html`) contains the most interesting proof fragments.

② *Alloy tests.* With that same Docker image, the Alloy litmus tests can be executed (located in '`alloy/litmus/`'). From the directory with our GitHub repository execute the command (which mounts directory '`alloy/`'):

```
docker run --rm \
  -w /home/proof/alloy \
  -v "$PWD/alloy:/home/proof/alloy" \
  beyond-the-fence \
  bash runlitmus.sh
```

This script takes 1-2 hours to execute. It writes the results to '`alloy/results.csv`', capturing whether our PTX model allows the litmus behaviors.

These results can be combined with the hardware behaviors to see whether only observed behaviors are allowed by the formal model. The following command checks the litmus tests against our hardware observations (from '`hw/expected/official-observations.csv`'):

```
python3 tools/combine_results.py \
  --alloy-csv alloy/results.csv \
  --hw-source official \
  --official-machine gb10 \
  --output hw/output/combined-official-gb10.csv
```

The script writes results (for the GB10 hardware behaviors) to '`hw/output/combined-official-gb10.csv`' and prints a mismatch summary. It exits nonzero only if hardware observed a weak behavior that Alloy reports as `UNSAT`, corresponding to a hardware behavior the formal model disallows. In contrast, behaviors allowed by Alloy but not observed on hardware are reported but do not fail.

③ *Hardware tests.* On the machine with NVIDIA GPU, inside the '`hw/`' directory, execute the full hardware testing campaign to reproduce Table 1 with:

```
./scripts/run_full_campaign.sh \
  --test-environments 2 \
  --iterations-per-test 1000
```

Runtime depends on the GPU and iteration count; the above command on a machine with GB10 takes around 1.5 hours. The full campaign compiles and runs the CUDA variants described by '`tuning-files/`' in Table 1, writing the outputs to '`hw/output/`'.

Note that the paper's official campaign ran A100, H100, and GB10 for 24 hours each, and GH200 for 6 hours.

A.6 Evaluation and expected results

① *Agda proofs.* The Agda proofs are expected to type check without errors.

② *Alloy tests.* For the Alloy litmus tests, only the behaviors allowed by the formal models should be observable on hardware. Our script checks that against our reference hardware behaviors.

③ *Hardware tests.* The hardware results in '`hw/output/`' should roughly match our results in '`hw/expected/`' when compared against the Alloy results. In particular the new GB10 traces in '`hw/output/full.txt`' should roughly match those in '`hw/expected/raw/gb10.txt`'. As weak memory behaviors appear at random, specific observations may differ.

These new traces should match our model, for which they can be compared against the Alloy results (Section A.5, point ②); invoke the above `combine_results.py` script with '`--hw-source hw/output/full.txt`'.

Generate Table 1 with:

```
python3 tools/generate_table_summaries.py \
  --view table1 \
  --output-csv hw/output/table1-summary.csv
```

Generate Table 2 with:

```
python3 tools/generate_table_summaries.py \
  --alloy-csv alloy/results.csv \
  --output-csv hw/output/table2-summary.csv
```

A.7 Methodology

Submission, reviewing, and badging methodology:

- www.acm.org/publications/policies/artifact-review-and-badging-current
- cTuning.org/ae

[8] agda.readthedocs.io/en/latest/language/safe-agda.html

B Memory Consistency Models

B.1 Axiomatic Shared-Memory Concurrency

The axiomatic semantics of a program in a model are given as a set of consistent executions [4, 8]. An execution consists of a set of events (shared-memory accesses or fences) and a set of relations defined on those events. As these definitions are entirely standard, we briefly cover basic notations, but refer to existing works for details [4, 8].

Execution. An execution $\mathsf{X} = \langle \mathsf{E}, \mathsf{po}, \mathsf{rf}, \mathsf{co}, \mathsf{rmw} \rangle$ corresponds to a directed graph. E is a set of events, consisting of reads (R), writes (W), and fences (F) – these are the graph nodes. The *program order* po, *reads-from* rf, *coherence order* co, and *read-modify-write* rmw are the sets of relations between events in the execution graph.

Consistency. The weak memory semantics of a program are captured by *a set* of those graphs. An axiomatic memory consistency model consists of several consistency predicates, or axioms, which restrict the set of execution graphs allowed in the model. An execution is consistent if it satisfies all consistency predicates. $[\![\mathbb{P}]\!]_M$ denotes the set of consistent executions of program $\mathbb{P}$ in memory model M.

Consistency is often defined over *derived* relations, which are a notational convenience over the primitive relations. For instance, *from-reads* $\mathsf{fr} \triangleq \mathsf{rf}^{-1};\mathsf{co}$ is a derived from rf and co. We define all ways in which we compose relations here. Given a binary relation B, its reflexive, transitive, reflexive-transitive closures, and inverse relations are denoted by $B^?$, B^+, B^*, B^{-1} respectively. We write $B_1; B_2$ to compose relations B_1 and B_2, *i.e.* $(B_1; B_2)(a, b) \triangleq \exists c\, B_1(a, c) \wedge B_2(c, b)$. The immediate relation B_{imm} is denoted as follows: $B_{\mathsf{imm}}(a, b) \triangleq B(a, b) \wedge \nexists c\, (B(a, c) \wedge B(c, b))$. The identity relation on a set A is denoted by $[A]$ *i.e.*, $[A](a, b) \triangleq a = b \wedge a \in A$. We write B_{loc} to denote the relation B between a pair of events accessing same memory location. We write $\mathsf{irr}(B)$ to denote that relation B is irreflexive and $\mathsf{acy}(B)$ to denote the acyclicity of relation B, that is, B^+ is irreflexive.

Behavior. The behavior of a consistent execution is set of values of all co-maximal write events. More formally, $\mathsf{Behav}(\mathsf{X}) \triangleq \{\langle e.\mathsf{loc}, e.\mathsf{Val} \rangle \mid e \in \mathsf{X}.\mathsf{W} \wedge [\{e\}];\mathsf{X}.\mathsf{co} = \emptyset\}$.

B.2 ARM

We review the existing axiomatic shared-memory concurrency model of ARM [35] (Figure 13a).

Events. ARM has a variety of load (relaxed, acquirePC, acquire), store (relaxed, release), atomic read-modify-write (relaxed, acquire, release, acquire-release), and fence (control, load-memory, store-store, and full) operations.

- A relaxed, acquirePC, or acquire load operation on memory location X generates $\mathsf{R}_{\mathsf{RLX}}(X, v)$, $\mathsf{R}_{\mathsf{Q}}(X, v)$, or $\mathsf{R}_{\mathsf{A}}(X, v)$ respectively, where the read value is v.
- A relaxed or release operation on memory location X generates $\mathsf{W}_{\mathsf{RLX}}(X, v)$, $\mathsf{W}_{\mathsf{L}}(X, v)$ respectively where the written value is v.
- A relaxed, acquire, release, or acquire-release RMW operation on memory location x generates a rmw-related events pair with labels: $\mathsf{R}_{\mathsf{RLX}}(X, v)$ and $\mathsf{W}_{\mathsf{RLX}}(X, v')$, $\mathsf{R}_{\mathsf{A}}(X, v)$ and $\mathsf{W}_{\mathsf{RLX}}(X, v')$, $\mathsf{R}_{\mathsf{RLX}}(X, v)$ and $\mathsf{W}_{\mathsf{REL}}(X, v')$, or $\mathsf{R}_{\mathsf{ACQ}}(X, v)$ and $\mathsf{W}_{\mathsf{REL}}(X, v')$ respectively where v and v' are the read and written values.
- A control, load-memory, store-store, full fence generates isb, F_{RM}, F_{WW}, or F_{MM} events respectively.

Relations. ARM defines a global ordering relation Ordered-before (ob), which is the transitive closure of the atomic-ordered-before (aob), Barrier-ordered-before (bob), dependency-ordered-before (dob), and Observed-by (obs) relations. Among these, the obs relation is an inter-thread, and aob, bob, and dob are intra-thread relations.

The obs relation is established by rfe, fre, and coe relations. The aob relation is based on rmw accesses – events a and b are in aob if $(a, b) \in$ rmw or event a is an rmw write and event b is a po-successor acquirePC or acquire read event, reading from a.

The bob relation in ARM orders a pair of po-related events a and b in the following cases:

- a is an acquire or acquirePC event
- b is a release event
- a is a release and b is an acquire event
- a is a read event, and (a, b) has an intermediate F_{RM} event
- a and b are write events with an intermediate F_{WW} event
- a and b has an intermediate F_{MM} event
- b is a write event and (a, b) has an intermediate release-write event on the same-location as b.

The dob relation in ARM is based on dependencies. It defines dob from a read to its po-successor event(s). A pair of events (a, b) is in dob in the following cases:

- b is addr or data dependent on a.
- b is a write and control dependent on a.
- Let c be an isb event and is control dependent on a or in addr; po from a.
- Event b is a write and there is an intermediate event e which is address dependent on a.
- b is a write and let e be a write on the same location as b and is ctrl or data dependent on a.
- Event b is a read which is satisfied by an intermediate write event e that forwards its written value to b and event e is addr or data dependent on event a.

Axioms. ARM defines the (sc-per-loc) axiom to establish a total order on same-location accesses. The (Atomicity) axiom forbids an intermediate same-location write event between an rmw-related event pair. Finally, the (External) axiom enforces a global order, where ob is acyclic.

B.3 PTX

The existing axiomatic PTX model (Figure 13b) is defined on the virtual PTX architecture for NVIDIA GPUs [25, 29].

$(po_{=loc} \cup co \cup fr \cup rf)$ is acyclic (SC-per-loc)

$rmw \cap (fre;coe) = \emptyset$ (Atomicity)

ob is irreflexive (External)

where

$ob \triangleq (obs \cup aob \cup bob \cup dob)^+$

$obs \triangleq rfe \cup coe \cup fre$

$aob \triangleq rmw \cup [codom(rmw)];rfi;[R_{\sqsupseteq Q}]$

$bob \triangleq po;[F_{MM}];po \cup [W_L];po;[R_A]$
$\cup\ [R];po;[F_{RM}];po \cup [R_{(A|Q)}];po$
$\cup\ [W];po;[F_{WW}];po;[W]$
$\cup\ po;[W_L] \cup po;[W_L];coi$

$dob \triangleq addr \cup data \cup ctrl;[W]$
$\cup\ (ctrl \cup (addr;po));[isb];po;[R]$
$\cup\ addr;po;[W] \cup (ctrl \cup data);coi$
$\cup\ (addr \cup data);rfi$

(a) ARM

$[W];cause_{=loc};[W] \subseteq co$ (Coherence)

$(sc;cause)$ is irreflexive (FenceSC)

$((rf \cup fr);cause)$ is irreflexive (Causality)

$rmw \cap ((ms \cap fr);(ms \cap co))$ is empty (Atomicity)

$(po_{=loc} \cup (ms \cap (rf \cup co \cup fr)))$ is acyclic (SC-per-loc)

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

where

$prel \triangleq ([W_{\sqsupseteq REL}];po_{=loc}{}^{?};[W]) \cup ([F_{\sqsupseteq REL}];po;[W])$

$pacq \triangleq ([R];po_{=loc}{}^{?};[R_{\sqsupseteq ACQ}]) \cup ([R];po;[F_{\sqsupseteq ACQ}])$

$obs \triangleq (ms \cap rf) \cup (obs;rmw;obs)$

$sw \triangleq (ms \cap (prel;obs;pacq)) \cup sc$

$causeb \triangleq (po^{?};sw;po^{?})^+$

$cause \triangleq causeb \cup (obs;(causeb \cup po_{=loc}))$

(b) PTX

Figure 13. Existing ARM and PTX axiomatic memory consistency models [25, 35].

Events. In the PTX model, the read, write, RMW, and fence accesses have memory orders: weak (WK), relaxed (RLX), acquire (ACQ), release (REL), acquire-release (AQRL), and sequentially-consistent (SC). The memory order of a read, write, RMW, and fence operation is one of o_r, o_w, o_u, o_f where $o_r \in \{\text{WK}, \text{RLX}, \text{ACQ}\}$, $o_w \in \{\text{WK}, \text{RLX}, \text{REL}\}$, $o_u \in \{\text{RLX}, \text{ACQ}, \text{REL}, \text{AQRL}\}$, and $o_f \in \{\text{ACQ}, \text{REL}, \text{AQRL}, \text{SC}\}$, respectively. The annotations are ordered by $\text{WK} < \text{RLX} < \{\text{ACQ}, \text{REL}\} < \text{AQRL} < \text{SC}$.

Non-weak PTX accesses also have an additional parameter, scope. The scope denotes the thread hierarchies within which the event may participate in synchronization – cta for all threads on a cooperative thread group, gpu for all threads on the same GPU, and sys for all threads in all devices, *i.e.* in terms of the hierarchy of scopes, we have cta < gpu < sys.

Relations. Based on these events and primitive relations, PTX derives a number of relations. A pair of non-weak events is *morally strong* (ms) when both events are not weak and have a scope that includes the thread of each other.

The sc relation orders a pair of morally-strong SC fences. The PTX model has a synchronization (sw) relation, built in several ways. For example, if an acquire reads from a release write and the events are morally strong, then it establishes synchronization. In general, a pair of morally strong release and an acquire patterns (prel and pacq) may synchronize via morally strong observations (obs). In addition, sc also establishes synchronization. Based on sw, po, $po_{=loc}$ and obs, PTX derives a global ordering cause as defined in Figure 13b. PTX also considers dependency relations to establish an intra-thread order.

Axioms. Based on these relations, the PTX model defines its axioms. The (Coherence), (FenceSC), and (Causality) axioms enforce consistency between the cause relation and the co, sc, rf, and fr relations. (Atomicity) and (SC-per-loc) axioms enforce atomicity and coherence for morally strong accesses. Finally, the (No-Thin-Air) axiom forbids any causality cycle consisting of dep and rf edges and thereby forbids any out-of-thin-air outcome [9].

B.4 Mapping

Figure 14 shows the mappings from scoped-C/C++ to PTX and ARM.

C/C++	ARM	PTX
$ld^{sco}_{\sqsubseteq RLX}$	ld^{sys}_{RLX}	$ld^{sco}_{\sqsubseteq RLX}$
ld^{sco}_{ACQ}	ld^{sys}_{Q}	ld^{sco}_{ACQ}
$st^{sco}_{\sqsubseteq RLX}$	st^{sys}_{RLX}	$st^{sco}_{\sqsubseteq RLX}$
st^{sco}_{REL}	st^{sys}_{L}	st^{sco}_{REL}
$U^{sco}_{o_u}$	$U^{sys}_{o_u}$	$U^{sco}_{o_u}$
F^{sco}_{ACQ}	F^{sys}_{RM}	F^{sco}_{ACQ}

C/C++	ARM	PTX
F^{sco}_{REL}	F^{sys}_{MM}	F^{sco}_{REL}
F^{sco}_{AQRL}	F^{sys}_{MM}	F^{sco}_{AQRL}
F^{sco}_{SC}	F^{sys}_{MM}	F^{sco}_{SC}
ld^{sco}_{SC}	ld^{sys}_{A}	$F_{SC}; ld^{sco}_{ACQ}$
st^{sco}_{SC}	st^{sys}_{L}	$F_{SC}; st^{sco}_{REL}$
U^{sco}_{SC}	U^{sys}_{AL}	$F_{SC}; U^{sco}_{AQRL}$

Figure 14. The mappings from scoped-C/C++ to PTX [25] and ARM [40]. $o_u \in \{\text{RLX}, \text{ACQ/A}, \text{REL/L}, \text{AQRL/AL}\}$

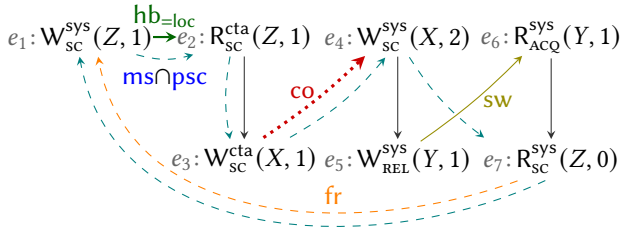
C Expanded Formal Analysis

Here we expand on our analysis of the incorrect compilation from scoped-C/C++ to PTX and we justify each of the components of our new prop definition.

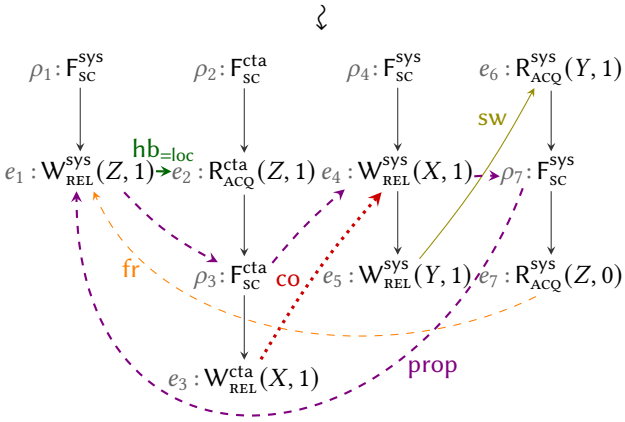
C.1 Mapping bug fixed

Figure 3 showed an example of incorrect compilation from scoped-C/C++ to PTX, which we explained in subsection 2.3, and which we later revisited in subsection 4.2. We show the corresponding scoped-C/C++ execution graph in Figure 15a with ms∩psc cycle, forbidden by the Scoped-C/C++ (Coherence) axiom. In subsection 4.2 we explained why the existing PTX model is insufficient to enforce the ordering; the mapping is incorrect under that model, as it erroneously allows the outcome a=b=1, c=0, X=2.

In Figure 15b we show the prop cycle for that program under our new PTX model. As the execution is disallowed by our (SC) axiom, the leading-F_{sc} mapping is correct for this program under our PTX model.

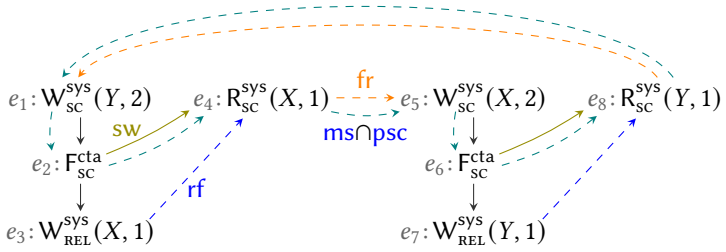


(a) in scoped-C/C++ (ms ∩ psc cycle)

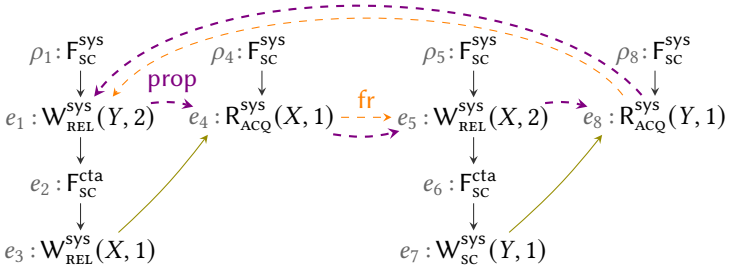


(b) in our new PTX model (prop cycle)

Figure 15. Execution graphs for programs in Figure 3, which maps correctly under our PTX model. The rightmost thread is in a separate CTA from the other threads.



(a) ms∩psc cycle in scoped-C/C++



(b) prop cycle in our PTX model

Figure 16. The execution graphs of a scoped-C/C++ program mapped to PTX. The first two threads are in a different CTA than the last two threads. Both executions are disallowed by a cycle in their respective model. Note that the sw edge to e_4 can start in either e_2 or e_3 (and to e_8 from e_6 or e_7) in both graphs; we draw the variant relevant for each cycle.

C.2 Justification for prop

In Section 5.1 we defined our own prop relation. Each of its cases is necessary for a particular case in the scoped-C/C++-to-ARM mapping proofs, which we explain here.

Case 1a. $(ms \cap ([F_{sc}];hb);(ms \cap ca))$ is needed for the program by the graph in Figure 15b, capturing the edge from $\rho_7 \xrightarrow{prop} e_1$, which no other prop case covers.

Additionally, this case covers $\rho_3 \xrightarrow{prop} e_4$ in our drawing. However, its context could be covered by an alternative path, namely: $e_1 \xrightarrow{prop} e_3 \xrightarrow{prop} e_4$, covered by case 3 and case 1b, respectively.

Case 1b. $(ms \cap ca)$ is needed for the program of the graphs in Figure 16, covering the edges $e_4 \xrightarrow{prop} e_5$ and $e_8 \xrightarrow{prop} e_1$.

Observe that no alternative prop-case covers those edges. Particularly, case 1a does not apply as the F^{cta}_{SC} e_2 is *not* morally-strong (ms) with e_5 (similarly for e_6 and e_1). Also the pscf-case (case 2) does not apply as the two F^{cta}_{SC} fences e_2 and e_6 are not morally-strong with each other.

Case 2. $(ms \cap pscf)$ is needed to link morally-strong fences, as we have explained in Section 4 with Figure 9.

Case 3. $(hb;[E_{\sqsupseteq REL}])$ is need in Figure 15, with a similar argument as we gave for case 1. This case covers $e_1 \xrightarrow{prop} \rho_3$ and $e_4 \xrightarrow{prop} \rho_7$; for the latter there is no alternative path. For the first case, there is an alternative path $e_1 \xrightarrow{prop} e_3 \xrightarrow{prop} e_4$, however, the $e_1 \xrightarrow{prop} e_3$ is covered by this same prop-case.

Case 4. $(hb^?;sw)$ is needed for the program represented by the graphs in Figure 16, where it covers $e_1 \xrightarrow{prop} e_4$ and $e_5 \xrightarrow{prop} e_8$. Observe that no other case covers it; The F_{sc}-case (case 1) does not apply, as the e_2 F^{cta}_{SC} is not strong enough to propagate to e_5 in the other CTA (similarly for e_6 to e_1).

The need for this case is a subtle consequence of the sw definition in scoped-C/C++, which can start either at a $W_{\sqsupseteq REL}$ or a $F_{\sqsupseteq REL}$. Crucially, *the scope of that fence is unrelated to the scope of the write*; if e_3 were also cta-scoped, $e_3 \xrightarrow{co} e_5$ would be a heterogeneous race (similarly for $e_7 \xrightarrow{co} e_1$). After mapping, that results in the situation where the F^{cta}_{SC} e_2 is not strong enough to propagate to e_5 in the other CTA, while the po-subsequent W^{sys}_{REL} e_3 cannot propagate fr by itself. Hence, no other prop-case can address this subtle case of scoped-C/C++.

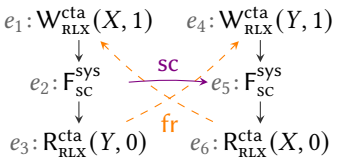


Figure 17. Store-buffering program, with threads in different CTAs. Disallowed by the pre-existing PTX model.

C.3 Relation to pre-existing PTX

Our model differs structurally from the original PTX model [25]. We took care to match its behavior on many known test cases (Section 6). However, in general, our model is *incomparable* to the original, meaning it is both *stronger* and *weaker*, depending on the case:

- *Stronger* – We are stronger to correctly support the leading-F_{sc} mapping. For instance, consider the program in Figure 3, for which the original PTX was not strong enough to disallow, whereas our model correctly disallows it (Figure 15b).
- *Weaker* – Our model is also *weaker* in some cases; never in ways observed on hardware nor does it affect the proofs.

  Consider the store buffering program in Figure 17. It is disallowed by the original PTX model, as it contains the cycle $e_1 \xrightarrow{\mathsf{po;sc;po}} e_6 \xrightarrow{\mathsf{fr}} e_1$. Note that $\mathsf{po;sc;po} \subseteq \mathsf{po;sw;po} \subseteq \mathsf{cause}$. Then $\mathsf{fr;cause_b}$ is disallowed by (Causality) from Figure 13b.

  In our model, however, we do not disallow this program. The fr-edges are not morally strong, as they are cta-scoped and go between different CTAs. Hence, only by the sys-scoped fences could we propagate information between the threads. However, both prop cases that cover fences are too weak to disallow this; Our pscf requires the intermediate fr edges to be ms, which they are not, while our F_{sc}-propagation (case 1a) requires the F_{sc} e_2 to be morally strong with the fr-target e_4 (similarly for e_5 and e_1).

  This program is not affected by the mapping; the program has a heterogeneous race, for which the source program behavior is undefined.

D Experimental Evaluation

Figure 18 shows the litmus tests run during our experimental evaluation. Figures 18a to 18c are used to test whether co and fr are part of SC-ordering. Figures 18d to 18i are MCA litmus tests, and are used to determine whether the hardware is MCA or not. In our experimental evaluation, variants of **iriw** did show non-MCA behaviors even with ACQ accesses and fences, although these were very rare. Table 2 shows the breakdown of allowed and visible tests on the base and strengthened PTX model across the GPUs we tested on.

$$\begin{array}{l|l} X=1 & Y=1 \\ Y=2 & X=2 \end{array}$$

(a) 2+2w

$$\begin{array}{l|l|l} X=1 & Y=1 & Z=1 \\ Y=2 & Z=2 & X=2 \end{array}$$

(b) 3.2w

$$\begin{array}{l|l|l} X=1 & Y=2 & a=Z \\ Y=1 & Z=1 & b=X \end{array}$$

(c) z6-3

$$\begin{array}{l|l|l|l} X=1 & a=X & c=Y & Y=1 \\ & b=Y & d=X & \end{array}$$

(d) iriw

$$\begin{array}{l|l|l} X=1 & a=Y & b=Z \\ Y=1 & Z=1 & c=X \end{array}$$

(e) isa2

$$\begin{array}{l|l|l} X=1 & a=X & Y=1 \\ & b=Y & c=X \end{array}$$

(f) rwc

$$\begin{array}{l|l|l} X=1 & a=X & b=Y \\ & Y=1 & c=X \end{array}$$

(g) wrc

$$\begin{array}{l|l|l} X=2 & a=X & Y=2 \\ & Y=1 & X=1 \end{array}$$

(h) wrw+2w

$$\begin{array}{l|l|l} X=2 & a=X & b=Y \\ & Y=1 & X=1 \end{array}$$

(i) wwc

Figure 18. Templates for the litmus tests run during our experimental evaluation.

Table 2. Breakdown showing the number of allowed tests under the base PTX and strengthened Sound PTX model proposed in this paper. The number of visible tests is very similar across the A100 and H100, although our experiments show that in rare cases the H100 GPU shows weak behaviors for MCA tests that are not seen on the A100. Due to a shorter runtime, the GH200 shows less weak behaviors; it is likely that running experiments for longer time periods would reveal more tests that show weak behaviors. Similarly, the GB10 shows less weak behaviors, likely due to the fact that it is a smaller, less-powerful GPU where the timing conditions necessary for weak behaviors occur more rarely.

Test	Allowed		Visible			
	PTX	Sound PTX	A100	H100	GH200	GB10
All	1208	1198	558	558	412	307
Figure 2c	2	2	1	1	1	0
Figure 3	2	1	0	0	0	0
2+2w	32	29	11	11	11	8
3.2w	160	154	80	79	72	44
z6-3	160	160	75	77	63	38
iriw	240	240	94	93	51	60
isa2	198	198	105	107	75	50
rwc	110	110	51	50	44	41
wrc	77	77	42	41	23	20
wrw+2w	110	110	44	44	37	25
wwc	117	117	55	55	35	21