



# Kent Academic Repository

Lustig, Daniel, Cooksey, Simon and Giroux, Olivier (2022) *Mixed-proxy extensions for the NVIDIA PTX memory consistency model*. In: ISCA '22: Proceedings of the 49th Annual International Symposium on Computer Architecture. . pp. 1058-1070. Association for Computing Machinery ISBN 978-1-4503-8610-4.

## Downloaded from

<https://kar.kent.ac.uk/96841/> The University of Kent's Academic Repository KAR

## The version of record is available from

<https://doi.org/10.1145/3470496.3533045>

## This document version

Publisher pdf

## DOI for this version

## Licence for this version

UNSPECIFIED

## Additional information

## Versions of research works

### Versions of Record

If this version is the version of record, it is the same as the published version available on the publisher's web site. Cite as the published version.

### Author Accepted Manuscripts

If this document is identified as the Author Accepted Manuscript it is the version after peer review but before type setting, copy editing or publisher branding. Cite as Surname, Initial. (Year) 'Title of article'. To be published in *Title of Journal*, Volume and issue numbers [peer-reviewed accepted version]. Available at: DOI or URL (Accessed: date).

## Enquiries

If you have questions about this document contact [ResearchSupport@kent.ac.uk](mailto:ResearchSupport@kent.ac.uk). Please include the URL of the record in KAR. If you believe that your, or a third party's rights have been compromised through this document please see our [Take Down policy](https://www.kent.ac.uk/guides/kar-the-kent-academic-repository#policies) (available from <https://www.kent.ac.uk/guides/kar-the-kent-academic-repository#policies>).

# Mixed-Proxy Extensions for the NVIDIA PTX Memory Consistency Model

Industrial Product\*

Daniel Lustig  
dlustig@nvidia.com  
NVIDIA  
Santa Clara, CA, USA

Simon Cooksey  
simon@graymalk.in  
University of Kent  
Canterbury, UK

Olivier Giroux†  
ogiroux@apple.com  
NVIDIA  
Santa Clara, CA, USA

## ABSTRACT

In recent years, there has been a trend towards the use of accelerators and architectural specialization to continue scaling performance in spite of a slowing of Moore’s Law. GPUs have always relied on dedicated hardware for graphics workloads, but modern GPUs now also incorporate compute-domain accelerators such as NVIDIA’s Tensor Cores for machine learning. For these accelerators to be successfully integrated into a general-purpose programming language such as C++ or CUDA, there must be a forward- and backward-compatible API for the functionality they provide. To the extent that all of these accelerators interact with program threads through memory, they should be incorporated into the GPU’s memory consistency model. Unfortunately, the use of accelerators and/or special non-coherent paths into memory produces non-standard memory behavior that existing GPU memory models cannot capture.

In this work, we describe the “proxy” extensions added to version 7.5 of NVIDIA’s PTX ISA for GPUs. A proxy is an extra tag abstractly applied to every memory or fence operation. Proxies generalize the notion of address translation and specialized non-coherent cache hierarchies into an abstraction that cleanly describes the resulting non-standard behavior. The goal of proxies is to facilitate integration of these specialized memory accesses into the general-purpose PTX programming model in a fully composable manner. This paper characterizes the behaviors that proxies can capture, the microarchitectural intuition behind them, the necessary updates to the formal memory model, and the tooling that we built in order to ensure that the resulting model both is sound and meets the needs of business-critical workloads that they are designed to support.

## CCS CONCEPTS

• **Computer systems organization** → **Single instruction, multiple data; Special purpose systems**; • **Theory of computation** → **Parallel computing models**; • **Computing methodologies** → **Modeling methodologies**.

\*This paper is part of the Industry Track of ISCA 2022’s program.

†Author is now at Apple, but work was performed while at NVIDIA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

ISCA ’22, June 18–22, 2022, New York, NY, USA

© 2022 Association for Computing Machinery.

ACM ISBN 978-1-4503-8610-4/22/06...\$15.00

<https://doi.org/10.1145/3470496.3533045>

## KEYWORDS

memory consistency, memory ordering, GPU, synchronization

### ACM Reference Format:

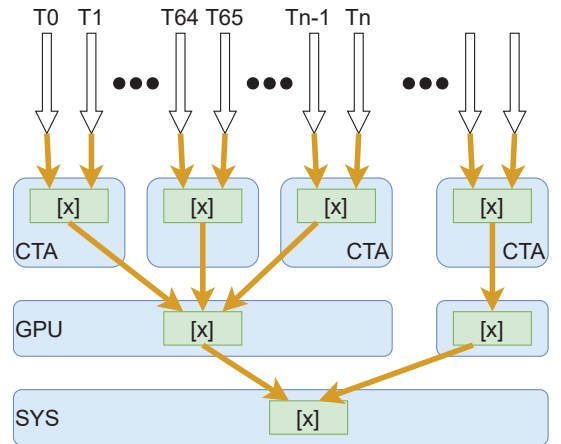
Daniel Lustig, Simon Cooksey, and Olivier Giroux. 2022. Mixed-Proxy Extensions for the NVIDIA PTX Memory Consistency Model **Industrial Product**. In *The 49th Annual International Symposium on Computer Architecture (ISCA ’22)*, June 18–22, 2022, New York, NY, USA. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3470496.3533045>

## 1 INTRODUCTION

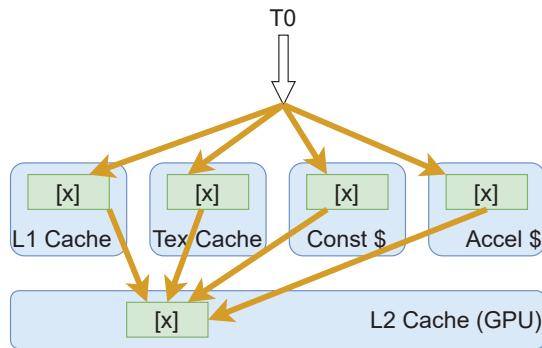
In shared memory programming models, threads’ legal interactions through memory are governed by the memory consistency model. Much ink has been spilled debating trade-offs in this space: strict vs. relaxed memory models, scoped vs. non-scoped models for GPUs, and approaches providing sequential consistency for data race-free programs. In spite of ongoing research, many topics in the field remain less than well understood, even by experts. Nevertheless, the memory model is a crucial part of a system’s overall programming model. Vendors that wish to provide forward-compatible programming models must continually strive for a sound and complete memory consistency model for their architectures.

One of the major challenges in defining a memory model for modern GPUs is the increasing number of special-purpose architectural features and accelerators. Across the entire industry, vendors are turning towards the use of special-purpose acceleration as a means of delivering continued performance scaling where scaling of general-purpose cores has slowed. NVIDIA GPUs have a number of such accelerators: Tensor Cores, Ray Tracing (RT) Cores, acceleration of surface and texture graphics operations, and ISA-defined asynchronous memory copy instructions, to name a few. Growing in number with each GPU generation, these features and accelerators range across a spectrum from being very tightly coupled to the CUDA Core’s pipeline to being very loosely coupled and interacting asynchronously with the launching thread. In addition, for various reasons (later described in Section 4), these architectural components may use non-standard non-coherent paths to the memory system.

The NVIDIA PTX memory consistency model, formalized with version 6.0 of the PTX ISA, is a scoped model [27, 37] but as Figure 1 shows, scope memory models cannot capture the architecturally visible behavior of the features described above. Instead, in existing GPU memory models, hardware features such as surface, texture, and constant memory are merely described as “incoherent”, and the programming guide simply declares that such memory should not be modified during the execution of a CUDA grid.



(a) Intuitively, a GPU’s scope hierarchy forms a tree, with a thread at each leaf and caching at each node. There is one path from each thread to the root node.



(b) Proxies address a different scenario in which there is a lattice pattern with multiple non-coherent paths in hardware from any given thread to the root node.

**Figure 1: The growing number of accelerators and special-purpose caches dictates that the GPU memory model needs to be extended beyond what the baseline scoped memory model is able to describe.**

While the approach of “just don’t do that” sufficed in the past, it is quickly becoming inadequate as tight integration of accelerators is an increasingly fundamental part of the CUDA programming model. Integration of accelerators and special-purpose non-coherent memory paths has become a primary goal for the PTX memory consistency model. The performance enabled by GPUs’ new hardware features are critical to their success in highly competitive markets such as machine learning. However, NVIDIA also promises to deliver a forward-compatible programming model at the PTX layer. It is not sufficient to provide forward compatibility only at the CUDA or software framework layers. The programming model must also be available for general-purpose use, both for customers and for benchmarks<sup>1</sup>.

In this paper, we present NVIDIA’s recent *proxy* extensions to the PTX memory consistency model. A proxy is a forward-compatible

programming model abstraction describing a particular path that an instruction or accelerator’s memory operations can take through the memory system. Proxies provide a well-defined programming model for the types of high-performance microarchitectures that NVIDIA designs to meet business needs, while also allowing PTX software optimized for current architectures to continue running correctly and efficiently on future architectures. The proxy memory model has recently been deployed in PTX 7.5 [37] and may be further extended in the future as the architecture evolves.

Overall, the contributions of this paper are as follows:

- (1) A summary of why NVIDIA made a conscious choice to expose an extremely relaxed memory model in PTX.
- (2) A description of a generally applicable model for how to capture the behavior of special-purpose accelerators and caches within a state-of-the-art general-purpose memory consistency model.
- (3) A formal specification of NVIDIA’s new proxy-aware PTX memory consistency model.
- (4) A set of publicly-available artifacts providing further details and verification of the proxy extensions, at <https://github.com/nvmlabs/mixedproxy>

## 2 BACKGROUND

In this section, we revisit some necessary background on GPU architecture and the role of memory models in modern CPU and GPU architectures.

### 2.1 NVIDIA GPU Architecture

From the compute perspective, NVIDIA GPUs currently consist of as many as 128 *streaming multiprocessors* (SMs) executing hundreds of thousands of concurrent threads communicating via a shared address space. Each SM consists of Single Instruction, Multiple Thread (SIMT) cores executing program code. Each SM also contains an L1 cache, a texture cache, a constant cache, and other specialized blocks (see Section 3.1). There is a single logical L2 cache physically distributed across the GPU.

NVIDIA’s hierarchical programming model largely reflects the GPU’s architecture. The user launches *grids* of threads, each of which runs a separate copy of the same code. Each grid is divided into *thread blocks*, also known as *cooperative thread arrays* (CTAs), comprising a set of threads guaranteed to run concurrently on a single SM. Each CTA is divided microarchitecturally into *warps* whose size matches the width of the SIMT execution engine. Recently, warp-synchronous programming has been deprecated in favor of a model in which each thread is an independent scheduling entity with starvation-freedom guarantees [35, 36].

All threads in a process share a *global* virtual address space, even across multiple GPUs. Each thread also has its own *local* memory space and each CTA has its own *shared* memory space implemented as a scratchpad either within or adjacent to the L1 cache. For composability and convenience when writing code for the GPU, all three spaces are mapped into a single *generic* memory space. Before the introduction of proxies in version 7.5, the PTX memory model applied only to the accesses performed to this generic memory space.

<sup>1</sup>e.g., MLPerf’s “Available” category requires both hardware and software to be publicly available [34].

Thread 0	Thread 1	Thread 2	Thread 3
st [x], 1	ld r1, [x]	ld r3, [y]	st [y], 1
	ld r2, [y]	ld r4, [x]	
Proposed final outcome:			
r1 == r3 == 1; r2 == r4 == 0			

**Figure 2: The “independent reads of independent writes” (IRIW) litmus test [10]. Prior to efforts to formalize the x86 memory model, Intel and AMD architecture specifications disagreed on whether the proposed outcome is legal [42]. At question is whether threads 1 and 2 can observe the updates to x and to y occurring in different orders.**

## 2.2 Formalization of Memory Consistency Models

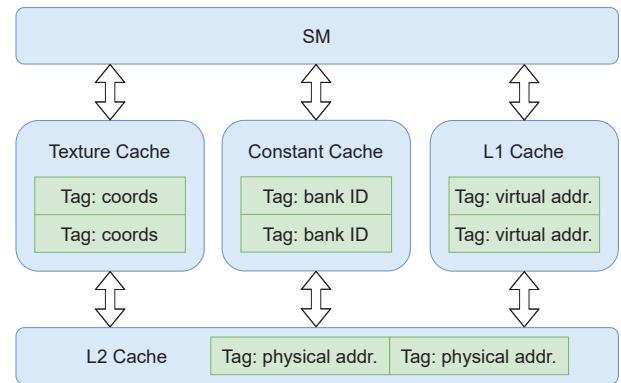
The overall goal of an architecture specification is to define a common programming model abstraction compatible with many different implementations. A *memory consistency model*, or simply *memory model*, defines the values that can be legally returned by loads from memory. The memory model forms an important component of the overall definition of an instruction set architecture (ISA). The role of an architecture-level memory model is to define portability guarantees about the way in which threads can interact with each other via memory.

Historically, industry architecture specifications have been written using informal natural language rather than mathematical formalism, and memory models have been no exception. However, as developers of high-performance data structures continue to aggressively optimize their code to take advantage of any available performance improvements, an increasing number of gaps have been identified in natural language memory model specifications. These gaps fall roughly into two categories: situations where the specification was too vague to give a clear adjudication of which behaviors were valid (e.g., Figure 2) and situations where the rules stated that certain behaviors are forbidden despite being empirically observable on some implementations [5].

These gaps have driven industrial adoption of mathematical formalism for memory model specification. The memory models for nearly all mainstream architectures are now formally specified, via officially sanctioned corporate documents [6, 17, 37] and/or via collaboration between academia and industry [3, 29, 42]. Relaxed memory models from industry can be notoriously complicated, and the large number of obscure corner cases makes them very difficult for humans to successfully reason about. The use of formal models allows architects to develop machine-checked correctness proofs that provide a more complete and more reliable means of verifying the validity of these models. The formal models also help programmers: designers of high performance concurrent data structures can verify correctness properties of their algorithms, and they can use automated tooling to reason about the behavior of their code.

## 2.3 GPU Memory Consistency Models

GPU memory model specifications have followed the trend towards increased formalism. The *heterogeneous race-free* memory model



**Figure 3: A high-level block diagram representing a generic NVIDIA GPU. Many PTX ISA features live outside the standard memory mode due their use of features such as non-coherent texture or constant caches.**

introduced the notion of scope to GPU memory models [16]. A *scope* is defined as the set of threads with respect to which a particular synchronization operation is applicable. GPUs commonly provide scoped synchronization across a CTA, across a single GPU, and across the entire system. Expert programmers can manually specify the scope of any particular synchronization operation, thereby accepting increased program complexity in exchange for increased performance. While there remains opposition to the continued use of scopes [45], scoped memory models remain the standard approach followed by industry today.

NVIDIA first adopted a formal memory model with the release of the PTX 6.0 specification and the Volta architecture generation [37]. PTX is NVIDIA’s virtual instruction set architecture is the lowest layer of the stack at which NVIDIA makes backward and forward compatibility guarantees. PTX provides a scoped memory model designed to accommodate GPU microarchitectures demonstrating very relaxed memory behavior, while nevertheless remaining compatible with important high-level programming languages such as C++ and CUDA. The rules of the PTX memory model were formally analyzed and proven compatible with the C++ memory model specification in 2019 [27]; this analysis was considered to be an important prerequisite to the official release of the model. The proxy memory model described in this paper is a new extension to the PTX 6.0 memory model.

## 3 MOTIVATION: MODERN GPU ACCELERATED COMPUTING ARCHITECTURES

In this section, we describe the architecture features that the proxy memory model targets and the architectural consequences of their inclusion.

### 3.1 Architecture Features Targeted by the Proxy Memory Model

Many accelerated computing features in the PTX ISA and planned for the future remain outside the PTX 6.0 memory model. Here, we

enumerate some of the new features supported by the introduction of proxies into the memory model.

**3.1.1 Surface and Texture Memory.** CUDA and PTX provide general-purpose compute code with the ability to manipulate the texture and surface primitives used by graphics workloads. At a high level, textures and surfaces are similar graphics-targeted data structures living in memory; surfaces have the advantage of being read/write, while textures are read-only but support operations such as interpolation or filtering while surfaces do not. A GPU’s texture cache is a dedicated structure designed to accelerate operations such as interpolation and to cache data using a tagging scheme that promotes locality across multiple spatial dimensions.

The behavior of the texture cache is abstractly depicted in Figure 3. As the current CUDA specification states, “within the same kernel call, the [texture] cache is not kept coherent with respect to global memory writes and surface memory writes, so any texture fetch or surface read to an address that has been written to via a global write or a surface write in the same kernel call returns undefined data” [36]. As such, the current PTX specification simply states that “[t]he memory consistency model does not apply to texture [...] and surface accesses” [37].

**3.1.2 Constant Memory.** GPU constant memory is a small, distinct address space reserved to hold fixed values such as some kernel launch parameters, some compile-time constants, user-declared constant memory arrays, and other metadata relevant to grid execution. The GPU’s constant cache hierarchy bypasses the L1 cache to provide CTAs in each grid with optimized low-latency access to constant memory. This behavior is also depicted in Figure 3.

In spite of their name, constants can also be accessed using read/write global memory aliases through CUDA APIs such as `cudaGetSymbolAddress()`. Indeed, this is how constants are updated by the host CPU before passing control to GPU grids. However, constants updated by the host during execution of a GPU grid result in undefined behavior.

**3.1.3 Virtual Aliasing.** The CUDA `cuMemMap()` API provides a way to create multiple general-purpose virtual aliases to any generic piece of memory [36], much like a standard `mmap()` call does on CPUs. A programmer might use this to produce a read-only and a read-write pointer for the same region. This pair of virtually aliased pointers might also use different CUDA L2 access management policies—which influence L2 occupancy—when accessing the same data, or they can be used to manage mappings visible to more than one process.

Although CUDA APIs allow creation of virtual aliases, “writes to one proxy of the allocation are considered inconsistent and incoherent with any other proxy of the same memory” [36] because some levels of the GPU’s cache may be virtually tagged rather than physically tagged. This is once again depicted in Figure 3. This behavior is unlike the analogous situation in CPUs, because CPUs today typically use physically tagged caches or otherwise ensure that virtual aliases are otherwise automatically resolved in a way that ensures same-physical-address ordering.

**3.1.4 Tightly Coupled Accelerators.** SMs also contain various special-purpose accelerators that interact with memory in non-standard ways. Tensor Cores, which accelerate tensor operations such as

warp level matrix multiply-accumulate (`wmma`) instructions, are highly optimized for high bandwidth with respect to access patterns commonly seen in machine learning workloads. Another example of a special purpose accelerator is accelerated asynchronous memory copy instructions (`cp.async`), which do not obey standard intra-thread memory ordering but instead behave as if they fork a new thread to perform the copy. These accelerators and others like them may contain specialized non-coherent caches and paths to memory that do not obey normal memory ordering rules, just as the texture and constant caches do.

## 3.2 Consequences of Non-Standard Memory Access

Each feature in the previous section exists to accelerate operations for important workloads. However, their non-standard memory behavior requires special treatment in the memory consistency model. Most notably, each feature described can violate intra-thread same-address memory ordering rules.

To illustrate one example, suppose a user performs a store to global memory and then performs a load to a constant bank alias of the same physical address, as shown in Figure 4a. Figure 4b shows how this program might be executed on a GPU microarchitecture today. The store will execute first (1), followed by the load (2). From here, either of the following may occur:

- (1) The load may hit on a previously-cached value still present in the constant cache (3a), but this value is stale because a newer write has been performed to the underlying physical address. Nevertheless, the stale line may still be resident because the caches are not kept coherent.
- (2) Alternatively, the store may be delayed while accessing the generic cache (1) if, for example, the cache is blocked servicing other requests. If the load request passes quickly through the constant cache to the L2 cache (3b) before the store passes through the L1 cache to the L2 cache (4), then the load and store will appear at the L2 cache out of order.

In both of these scenarios, the load will return a value written by a previous store, thereby effectively causing the load to have been reordered before the earlier store to the same physical address. The consequence of such same-address ordering violations is that they introduce the possibility of an *intra-thread data race*. Much like a regular data race, the order in which the store and load in the example above appear to execute is non-deterministic<sup>2</sup>.

Making things worse is that no existing synchronization mechanisms in PTX 6.0 suffice to prevent these intra-thread data races. For example, Figure 4a shows a `__threadfence()` API call, which maps to a fence instruction in PTX. However, this fence was designed to synchronize generic memory operations and hence does not synchronize the non-coherent memory paths used in this example. Therefore, to build a complete model, a new synchronization mechanism is needed that both resolves intra-thread data races across non-standard memory paths and integrates cleanly with standard inter-thread synchronization (e.g., fence instructions, or `.release`

<sup>2</sup>In fact, the results may not always be describable in terms of simple interleavings at all. Data races are commonly defined to result in *undefined behavior* rather than simple non-deterministic interleavings.

```

__constant__ int const_array[N];
__global__ void kernel(int *global_ptr, /* ... */) {
    /* Store global */
    global_ptr[0] = 42;

    /* The fence serves no purpose here */
    __threadfence();

    /* Constant load to alias of same address */
    int x = const_array[0];

    /* ... */
}
int main(int argc, char* argv[]) {
    /* ... */

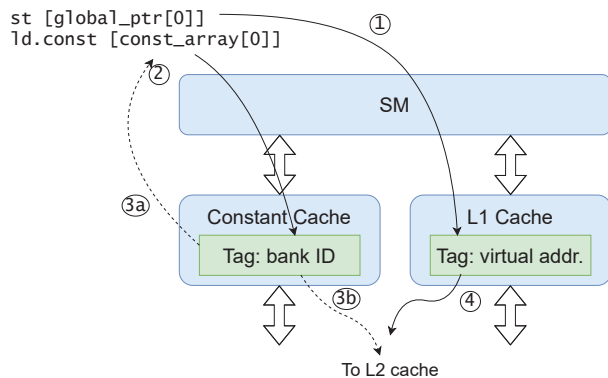
    /* Get global alias for the constant array */
    int *global_pointer;
    cudaGetSymbolAddress(
        (void**)&global_pointer, const_array);

    /* Launch the grid */
    kernel<<<1,1>>>(global_pointer, /* .... */);

    /* ... */
}

```

(a) An example of CUDA code that exposes the weak ordering behavior of mixed-proxy execution.



(b) If (3a) is a hit on a stale cache line or (3b) reaches memory before (4) then the instructions will appear to have been reordered.

**Figure 4: Example showing how specialized caches can produce architecturally visible same-address memory reordering, even within a single thread.**

and .acquire modifiers). This motivates the creation of the new *proxy fence* synchronization primitive that NVIDIA has integrated into the PTX 7.5 memory model.

#### 4 WHY DON'T EXISTING SOLUTIONS SUFFICE?

NVIDIA's GPUs are not the first or only architecture to utilize features that help performance but do not obey standard memory ordering. However, perhaps uniquely among its peers, NVIDIA has consciously chosen to embrace exposure of the highly relaxed memory behaviors introduced by these features into the ISA itself. Here, we discuss why NVIDIA has taken the approach to make the memory model highly relaxed, rather than choosing from available alternatives.

#### 4.1 "Just Don't Do That"

One approach to incorporating the features described in Section 3.1 would be to disallow any code that exposes the lack of coherence by deeming it undefined behavior. In fact, for many years this was the approach taken by NVIDIA to describe surface, texture, and constant memory. However, this approach has become insufficient for two reasons. First, non-standard memory accelerators existed historically as part of the graphics pipeline, with niche use in the compute pipeline. Today, newer features such as asynchronous memory copies and tensor core operations are core parts of the compute pipeline for modern workloads, particularly within the domain of machine learning. As described in Section 3.1, these features make use of special non-coherent paths to memory. In order for them to be exposed in PTX in a forward-compatible manner, there must be a forward-compatible specification of their behavior within the memory model.

Second, disallowing code that exposes the non-coherent paths in the microarchitecture cuts off the exploration of new optimization ideas. For example, a user may wish to write CUDA code to generate surfaces and textures on the fly for their graphics applications. Or, a user performing kernel fusion (i.e., merging two grids into one by manually concatenating the code and performing manual synchronization in between) may wish to overwrite the first grid's constants with the second grids' constants on the fly during the inter-grid transition. The proxy model allows optimizations such as these to be explored and exposed in PTX in a forward-compatible manner.

#### 4.2 "Just Make Everything Coherent"

If "just don't do that" is considered an unacceptable solution, then an alternative solution would be to "just make everything coherent". To achieve this, NVIDIA would need to re-architect the SM and/or memory system to eliminate the presence of multiple non-coherent and/or non-physically tagged caches within the SM. Adding physical tagging with a coherence mechanism to each of the relevant specialized caches would indeed eliminate the problems identified in the previous section.

Unfortunately, there are a number of downsides to the approach of "just make everything coherent". First, adding new coherence mechanisms would impose a cost in performance, power, and/or area. Each of the existing structures has been highly optimized over multiple designs to provide efficient execution for specialized operations. For example, the texture cache is tagged so that it is optimized for locality across multiple coordinate dimensions, and this form of locality does not always correspond well to locality within the one dimensional generic address space. Changing the cache line tag structure to hold only physical addresses would negatively impact texture locality and hence performance. Alternatively, changing the tags to hold both physical addresses and tag coordinates would add area and power to a performance-critical and highly replicated portion of the GPU's microarchitecture. Finally, imposing a requirement that all accesses perform address translation before accessing any caches would itself impose a substantial burden on the L1 cache latency and TLB throughput. None of these points individually is fundamentally intractable, but in all they represent very real costs that to date have led NVIDIA not to pursue them.

Second, academic publications often grossly underestimate the design and verification costs required to add new features to the microarchitecture. In industry, there is a finite budget that can be spent on architectural innovation for each generation. Re-architecting large portions of the SM to become fully coherent would cost a non-trivial portion of the development budget of any given generation. From a design cost perspective, it is also easier for each architectural unit to be designed as a mostly self-contained entity rather than allow tight coupling between units.

### 4.3 Reusing Existing Synchronization Mechanisms

A third option to restore coherence in non-coherent memory paths would be to repurpose existing PTX inter-thread synchronization mechanisms (fence and membar instructions, and .release and .acquire annotations) to synchronize the currently non-coherent paths. However, doing so would add a non-trivial overhead to the cost of performing these operations, particularly for the CTA-scoped variants which are expected by programmers to be very fast. Conceptually this would pessimise the common case for the sake of supporting a smaller set of targeted synchronization scenarios, and this trade-off is not an obvious win.

### 4.4 Summary of Alternatives to Proxies

During the development of the proxy memory model extension and associated GPU architecture, NVIDIA evaluated the trade-offs described above. In the end, *NVIDIA made a conscious choice to develop and release the proxy memory model extension rather than re-architect the cache hierarchy to become fully coherent*. This shifts the verification burden away from individual components and towards a more system-level perspective, but the design and verification of proxies and proxy fence implementations (Section 5) was considered preferable. This also shifts complexity from the architecture to the programming model, thereby making life more complicated for programmers. In contrast to current trends observed in parts of academia, NVIDIA felt that continuing to pursue aggressively weak (but nevertheless fully rigorous) memory models was the approach that provided the best return on investment. Programmers looking for optimal performance on specialized accelerators are (perhaps for lack of better alternatives) willing to do the requisite low-level coding. As such, we now spend the rest of this paper explaining the new proxy programming model, before concluding with Section 7 which explains how both experts and non-experts can program with these proxies in a composable manner.

## 5 PROXY MEMORY MODEL OVERVIEW

In this section we provide an intuitive overview of the proxy model. Section 6 provides a more formal description.

### 5.1 Generalizing a Model Supporting Virtually Tagged Caches

The notion of proxies can be viewed as a generalization of a system employing both virtual address translation and virtually tagged caches. As on any system with virtual memory, all memory accesses are performed using virtual addresses alone, as the programming

model exposes only virtual addresses to the user. On traditional systems with physically tagged caches, all virtual memory addresses are translated into physical memory addresses prior to (or, for virtually indexed caches, in parallel with) checking for cache hits. Therefore, the observable behavior of the memory system (i.e., the memory model) is effectively determined by physical addresses alone. However, on systems with one or more layers of virtually tagged caches, two different virtual address aliases mapping to the same physical address will appear as two unrelated addresses within different cache lines. As such, normal rules such as maintaining physical address read-after-write ordering are violated, in the same way as described in Section 3.2.

For NVIDIA GPUs, rather than writing off architecturally visible virtual caching behaviors as bugs, we have chosen to embrace them due to the architectural flexibility that allowing these behaviors enables. A memory model that can account for these behaviors must satisfy a few properties. First, it must capture the set of mappings from each virtual address to its associated physical address, including tracking the presence of virtual address aliases. Second, it must account for the type of non-deterministic intra-thread behavior described in Section 3.2. Third, it must provide a synchronization mechanism capable of resolving data races. Our proxy memory model captures all of these behaviors, as well as those observed when using texture caches, constant caches, and accelerators accessible from the GPU SM.

### 5.2 The Proxy Abstraction

The foundation for the proxy memory model is the pre-existing PTX memory model described in Section 2.3 [27, 37]. As in most memory models, the basic primitives are memory operations (loads, stores, and atomic read-modify-writes) and memory fences. Memory operations carry an associated address; this was traditionally taken to be the physical address because existing memory models assume that virtual aliases will be resolved automatically by hardware.

We define a *proxy* as an extra tag abstractly applied to each operation when performing memory model analysis. Proxies need not correspond directly to actual bits in hardware; instead, they are merely an abstract conceptual token used to categorize each memory operation. The purpose of proxies is to capture the following key property: operations performed with the same proxy are guaranteed to obey the rules established for generic memory operations in PTX 6.0 (before the introduction of proxies). However, operations performed with different proxies may violate those rules, as intuitively they may be subject to the microarchitectural situations described in Section 3.1.

Figure 5 shows some examples of proxies being applied to memory operations. In the first row, the `ld.global` instruction is, unsurprisingly, modeled as a load operation to some physical address. Since it is a normal load to generic memory (as defined in Section 2.1), it is assigned the *generic proxy* tag. Likewise, the store in the second row is modeled as a store operation to the same address using the same proxy. The store in the third row is again similar, except that it is performed to a virtual alias. In this case, the physical address is the same, but the generic proxy is specialized with a different virtual address. The use of different proxies reflects the fact that the L1 cache may be virtually tagged, and hence the two operations may

PTX Instruction	Operation	(Physical) Address	Scope	Proxy
ld.global.u32 r1, [rd6]	Load	0x4080	Weak	Generic (virtual alias == [rd6])
st.global.sys.u32 [rd6], r4	Store	0x4080	Sys	Generic (virtual alias == [rd6])
st.global.u32 [rd8], r9	Store	0x4080	Weak	Generic (virtual alias == [rd8])
sust.b.1d.vec.b32.clamp [surf, r1], r2	Store	0x9600	Weak	Surface (CTA 4)

Figure 5: As these examples show, a proxy is simply an extra tag conceptually associated with each memory operation based on the operation and virtual address. rd6 and rd8 are assumed to hold two different virtual addresses aliasing the same physical address.

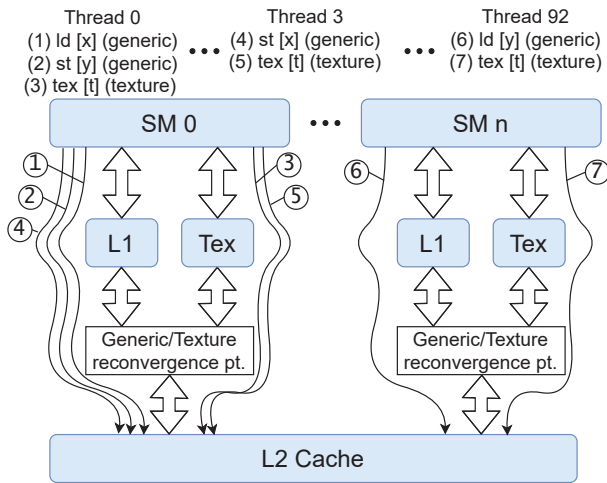


Figure 6: Microarchitectural intuition for the behavior of proxies. Each pictured thread executes a series of memory operations (1)–(7); each operation is labeled using the proxy associated with that operation. Addresses  $x$  and  $t$  are assumed to alias. The operations then map onto microarchitectural packets that take different paths through the pictured implementation.

be subject to an intra-thread data race. Finally, the surface store instruction `sust` in the fourth row also results in a store operation to some physical address. However, in this case it is tagged as being performed using the *surface proxy* rather than the generic proxy, as it will be implemented microarchitecturally as passing through the texture cache rather than the L1 cache, again causing the accesses to form an intra-thread data race.

The PTX memory model rules are then adjusted as follows, using Figure 6 as an intuitive guide:

- Operations from threads in the same CTA to the same address and using the same proxy (e.g., (1) and (4), or (3) and (5)) perform exactly as they would in the original memory model, i.e., intra-thread same-address ordering and standard inter-thread synchronization rules are respected. Intuitively, by virtue of being issued by the same CTA, both accesses would follow identical (in-order) paths through the memory system, and would use matching form(s) of tagging in caches along the way, so no out-of-the-ordinary behavior can be observed.

```
fence{.sem}.scope;
fence.proxy.proxykind;

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

Figure 7: The fence instruction in PTX 7.5 [37]. Only the alias proxy is currently supported.

- Operations to the same address using the same generic proxy but from different CTAs (e.g., (2) and (6), or (3) and (7)) also obey standard inter-CTA synchronization and memory ordering rules. This rule and the prior rule together ensure that pre-existing code written using only generic operations and avoiding virtual aliases continues to work unmodified.
- As described in Section 3.2, unsynchronized operations from the same thread or CTA to the same address but using different proxies (e.g., (1) and (3)) form an intra-thread data race. Thus, synchronization of such operations will now require the proxy fence primitive described in the next section. Note that accesses to aliases of the same physical address are performed via different proxies.
- Operations from threads in different CTAs to the same address but using non-generic proxies (e.g., (5) and (7)) must also be synchronized using proxy fences. For example, even though both operations may be texture load path operations, each CTA will (in general) be executing on a different SM and so the two operations will be performed via two different unsynchronized texture caches.

### 5.3 Proxy Fences

As described in Section 4.3, no pre-existing mechanism suffices to support synchronization of the GPU’s non-coherent paths to memory. Therefore, we introduce a new synchronization primitive called a *proxy fence* with the syntax shown in Figure 7. As described in Section 5.2, proxy fences must be inserted to synchronize across different proxies and/or between non-generic proxies on different CTAs. Here, we describe the operation and usage of proxy fences.

A proxy fence establishes ordering between memory accesses that happen via different proxies. More specifically, a `fence.proxy.alias` instruction synchronizes across to different generic proxies; i.e., it re-establishes ordering between two different virtual aliases. Other proxy fences will be able to synchronize the specified proxy with



the generic proxy. For example, a `fence.proxy.texture` instruction would synchronize a CTA's texture proxy with the generic proxy.

The intuition for a proxy fence is that it flushes prior generic accesses and the specified proxy's prior accesses to the point at which the two microarchitectural paths reconverge, and then it invalidates any possibly-stale cache entries in any cache(s) along those paths. This combination ensures that the prior accesses will appear ordered before any subsequent accesses. For example, a `fence.proxy.texture` in Figure 6 would flush a CTA's outstanding texture loads and generic operations to the pictured reconvergence point. Depending on the proxy and the GPU generation, this will generally be some point in between the L1 and L2 caches. It would also invalidate any possibly stale entries in the constant cache. However, it would not need to invalidate any entries in the L1 cache. This is because there is no texture store instruction in PTX, and hence it is not possible for an L1 cache entry to be stale with respect to a newer store issued via the texture proxy.

## 5.4 Usage Examples

Given the semantics described above, to synchronize across proxies, a programmer must insert a proxy fence at the point in a thread's execution where the inter-proxy synchronization must take place. When synchronizing within a single thread, the proxy fence is simply placed between the instructions in question, as shown in Figures 8a and 8b. When synchronizing across threads, there must already be a chain of causality established by one or more release/acquire pattern pairs. Proxy fences can be inserted anywhere that a normal load or store could be inserted along that chain, i.e., before a release operation or after an acquire operation.

The proxy fence must also be inserted in the same CTA where the non-generic access is taking place. Intuitively, a CTA cannot synchronize a different SM's special-purpose caching. This is shown in Figures 8c through 8e. If multiple distinct proxies are being used, then one proxy fence must be used for each proxy, and the proxy fences must be in the correct order, as shown in Figure 8f. The correct order will in general be to synchronize the first non-generic proxy with the generic proxy, and then to synchronize the generic proxy with the second non-generic proxy.

## 6 FORMALIZATION AND TOOLING

Before deploying NVIDIA's new proxy memory model, we considered it important to make sure that the proposed rules integrate cleanly into the existing formalization of the PTX memory model [27, 37]. Rigorous formalization is a critical tool for ensuring that no obscure corner cases have been missed. Although memory model formalization remains a somewhat opaque topic, NVIDIA considers formal modeling an important prerequisite to the release of our models. As such, here we describe how our formal model was extended to account for proxies and then summarize the tooling and analysis infrastructure built to prove to ourselves that the model is sound.

### 6.1 Baseline: the PTX 6.0 Axiomatic Memory Model

PTX is specified as an axiomatic memory model: a load may return any value that is consistent with the six primary axioms of the model.

- (1) Coherence establishes a consistent ordering among writes to the same address.
- (2) Sequential Consistency per Location handles same-address synchronization rules.
- (3) Causality handles cross-address synchronization rules.
- (4) Fence-SC handles restoration of sequential consistency using `fence.seq_cst` instructions.
- (5) Atomicity handles read-modify-write operations.
- (6) No-Thin-Air handles a thorny theoretical corner case [12].

These axioms operate on *operations* such as reads, writes and fences, which are the primitives in the model, and on *relations* between those primitives. For example, if a read R returns a value written by a write W, then a *reads-from* (`rf`) relation is established between W and R. In Figure 9, if the release (2) and the acquire (3) pair were to establish a *causality* relation between (1) and (4), then the Causality axiom dictates that the `rf` relation must be consistent with the combination of *base causality* (`causebase`) and program order (`po`) into the causality relation (`cause`).

The challenge of formalizing an axiomatic memory model is in specifying precise, sound definitions of relations such as these and the subtle ways in which they may or may not compose with each other in order to establish reliable synchronization patterns. Due to space limitations, and since it was covered in detail in prior work [27], we do not attempt to re-explore every detail of the PTX memory model or its axioms. Instead, we focus only on the changes and their motivation.

### 6.2 Extending the Formal Model

Here, we describe the updates made to the public NVIDIA PTX documentation in order to support proxies. Wording from the PTX specification itself [37] is presented in indented block quotes, and new text within those indented sections is highlighted in green.

**6.2.1 Addresses and Memory Locations.** The first change in the model adds a precise distinction between memory address (virtual) and memory locations (physical):

§8.2: The address operand contains a virtual address that gets converted to a physical address during memory access. The physical address and the size of the data type together define a physical memory location, which is the range of bytes starting from the physical address and extending up to the size of the data type in bytes.

**6.2.2 Moral Strength.** Moral strength is a property of a pair of operations. Informally, it determines whether the accesses are eligible to form some type of synchronization pattern within the model. Moral strength now requires that memory operations be performed via the same proxy. This adjustment is what formally introduces the possibility of intra-thread data races as described earlier in this paper. It also establishes that for inter-thread synchronization patterns to

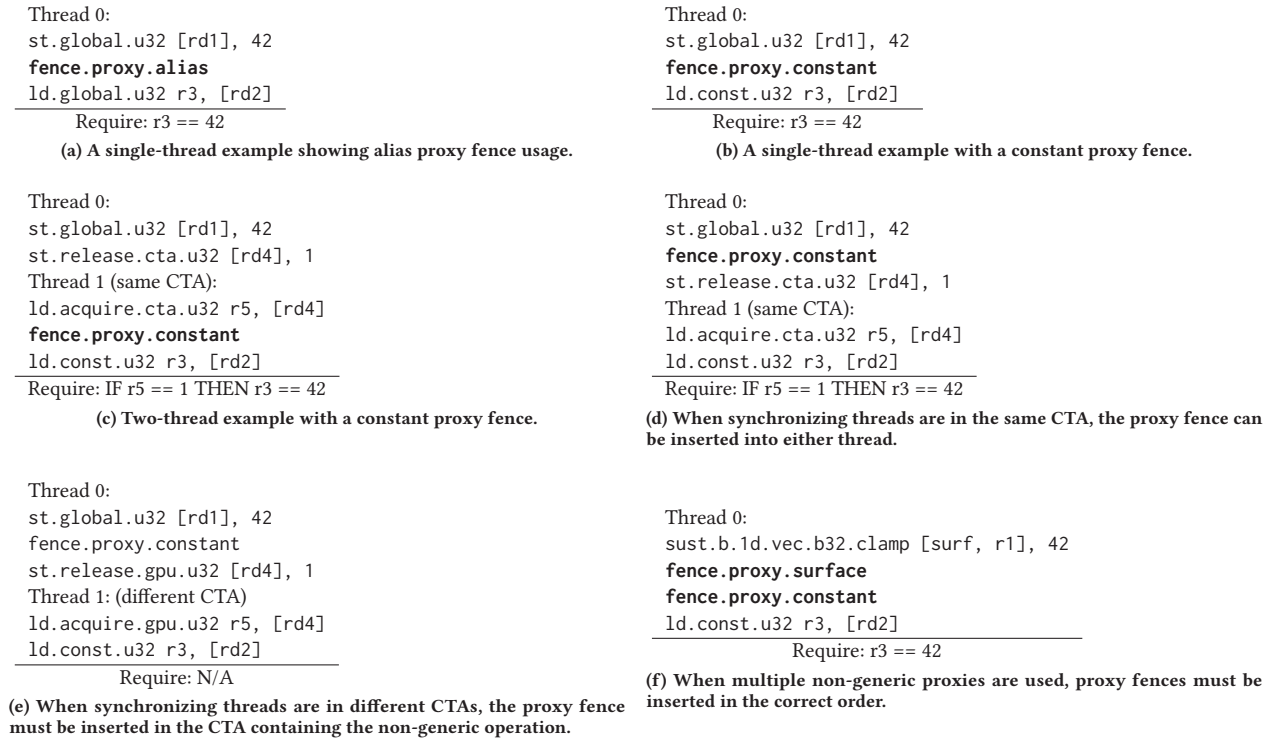


Figure 8: Litmus tests. (Not all proxies shown are present in PTX 7.5.) In all examples, rd1 and rd2 are assumed to alias.

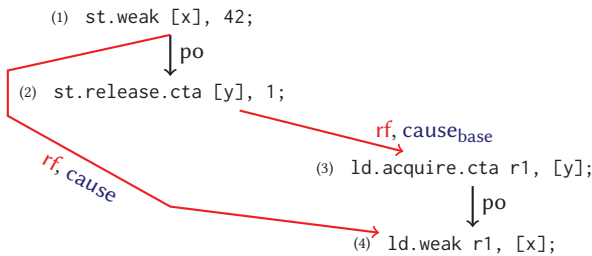


Figure 9: An example of axiomatic memory model analysis

be established by pairing a release-pattern with an acquire-pattern, the release and acquire must also be performed via the same proxy.

§8.7: Two operations are said to be *morally strong* relative to each other if they satisfy **all of** the following conditions:

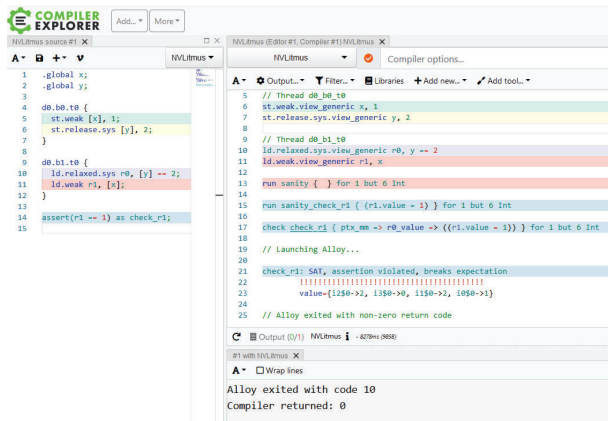
- (1) The operations are related in *program order* (i.e, they are both executed by the same thread), or each operation is *strong* and specifies a *scope* that includes the thread executing the other operation.
- (2) **Both operations are performed via the same proxy.**
- (3) If both are memory operations, then they overlap completely.

6.2.3 *Base Causality Order.* As its name suggests, base causality order establishes a basic individual hop of synchronization. The only change required was to add program order to the list below. This change by itself has no effect in the pre-existing memory model without proxies, but its addition facilitates the modeling of intra-thread data races.

§8.9.5: An operation X precedes an operation Y in *base causality order* if:

- (1) X precedes Y in *program order*, or
- (2) X synchronizes with Y, or
- (3) For some operation Z,
  - (a) X precedes Z in *program order* and Z precedes Y in *base causality order*, or
  - (b) X precedes Z in *base causality order* and Z precedes Y in *program order*, or
  - (c) X precedes Z in *base causality order* and Z precedes Y in *base causality order*.

6.2.4 *Proxy-Preserved Base Causality Order.* Unlike the prior relations that need only minor modifications, this key relation was newly created to account for the impact of proxies on base causality order. It captures the rules laid out in Section 5.2: either matching proxies must be used, or programmers must insert proxy fences in the appropriate locations.



**Figure 10:** A screenshot of the NVLitmus tool developed for this project and integrated into a locally hosted copy of Compiler Explorer [18]

§8.9.5: A memory operation X precedes a memory operation Y in *proxy-preserved base causality order* if X precedes Y in *base causality order*, and:

- (1) X and Y are performed to the same address, using the *generic proxy*, or
- (2) X and Y are performed to the same address, using the same *proxy*, and by the same thread block, or
- (3) X and Y are aliases and there is an *alias proxy fence* along the base causality path from X to Y.

6.2.5 *Causality Order.* Finally, causality order must be adjusted to use the new proxy-aware variant of base causality order.

§8.9.5: *Causality order* combines *base causality order* with some non-transitive relations as follows: An operation X precedes an operation Y in *causality order* if:

- (1) X precedes Y in *proxy-preserved base causality order*, or
- (2) For some operation Z, X precedes Z in observation order, and Z precedes Y in *proxy-preserved base causality order*.

In all, these changes reflect the behaviors motivated microarchitecturally in Section 3.1 in an abstract manner, without resorting to directly exposing microarchitectural details within the programming model.

### 6.3 Tools to Support Verification

Accompanying the memory model changes are a set of tools developed to properly analyze the PTX model extensions. The foundation for these tools is an extension of the Alloy model developed for the original PTX memory model in prior work [27]. Alloy is a relational modeling tool that has been deployed to analyze various memory models [48] as well as in a range of other domains [20]. The proxy extensions follow the same style and methodology, and so for space reasons, we do not further elaborate on them here, but they will be released as artifacts upon paper acceptance.

We also used a variant of the same Alloy model to automatically generate a set of targeted litmus tests, following prior work [28]. When applied to the proxy memory model, we observed similar results to what was seen in prior work: reproducing many standard litmus tests and a number of variants specific to the PTX memory model. A few new tests were generated with the proxy memory model features, revealing a smattering of non-standard patterns that we added to our test suite. This analysis provided evidence that the new proxy memory model behaved “as expected”. Unfortunately, the exponential (or worse) runtime of this technique meant we were unable to generate a fully comprehensive suite of litmus tests. We found that tests with only six instructions were at the practical limit, so we remain on the lookout for ways to improve this methodology moving forward.

Finally, to facilitate the use of the model for analysis by non-experts, we built a front end called NVLitmus for the formal Alloy model and integrated it into a locally hosted copy of Compiler Explorer [18], shown in Figure 10. This allows users to write litmus tests in a stylized plain text representation and then run NVLitmus in the browser, without needing to understand the model or even to install Alloy. NVLitmus has been used successfully by expert programmers within the company when looking to write non-trivial inter-thread synchronization code at the PTX level. Some of these artifacts are publicly available at <https://github.com/nvmlabs/mixedproxy>.

## 7 DISCUSSION

### 7.1 Broader Programming Model Implications

Having a solid memory model foundation is an important step towards making accelerators and special-purpose cache hierarchies usable in a general-purpose manner. However, it is only a first step. Fence insertion is known to be a difficult topic in the world of memory models and many tools have been developed to assist users in deciding where to insert fences [1, 4, 14, 25]. The introduction of proxy fences only makes this topic more challenging. As such, in order for the model to be used effectively, it must come with a set of guidelines for how it can be used by both expert and non-expert programmers.

We do not expect that programmers will mix and match proxies while attempting to write esoteric synchronization patterns. Indeed, the addition of same-proxy requirements to the definition of moral strength (Section 6) was in part intended to discourage such attempts. We expect that the vast majority of use cases will either be single-threaded or one or more iterations of the standard four-step “message passing” idiom shown in Figure 9: 1) write data, 2) write-release flag, 3) read-acquire flag until seeing the updated value, 4) read data. To that end, the litmus tests available with this paper and in the PTX documentation provide clear guidelines on where proxy fences should and should not be inserted.

Another important observation is that careful choice of how to include proxy fences into the definitions of the causality relations means that the proxy model remains fully composable. In memory model terminology, the proxy model continues to respect *cumulativity* [5]. Once a proxy fence has been used to restore ordering across proxies within a CTA, subsequent synchronization

performed with respect to threads outside of that CTA will observe the updated values as well.

## 7.2 Looking Beyond the PTX 7.5 Proxy Memory Model

Perhaps unsurprisingly, in spite of the complexity added by the proxy memory model extensions, there are already calls within NVIDIA to add new features to further accelerate workloads of commercial interest. Although acceleration of these algorithms is clearly an important goal for the company, the addition of such features would add even more complexity to the memory model, thereby presenting another non-obvious trade-off. Adding new memory model features in a way that maintains forward- and backward-compatibility would require once again extending the formal model and then performing the types of analysis described in this paper to convince ourselves that the extensions are sound.

Two other properties of the proxy memory model are worth noting. First, it is designed around the notion of CTA scope being “special”. This was a design choice rather than a fundamental theoretical requirement. The accelerators and caching features described in Section 3.1 all currently live within each individual SM, and a CTA’s threads run on the same SM. Threads within a single CTA often work on common data ranges in a tightly coordinated manner and have very fast intra-CTA synchronization mechanisms available. However, if accelerators or special caches were added at layers of the memory hierarchy outside the SM, then the proxy model could potentially be extended to permit scoped mixed-proxy synchronization.

Second, the proxy model does not currently provide way for users to synchronize across dynamic changes in virtual-to-physical address mappings. On CPU architectures, this often requires execution of a special dedicated TLB synchronization “recipe”, as normal synchronization mechanisms may not flush stale TLB entries [43]. NVIDIA’s Unified Memory page migration mechanism handles its own TLB synchronization needs, but no other general purpose mechanism for synchronizing changed mappings is provided.

## 8 RELATED WORK

### 8.1 Standard Memory Models

GPU memory models have been an active area of research in recent years. Just as in other architectures, the push to formalize GPU memory models more rigorously is based in part on the identification of shortcomings in earlier, more informally specified versions [2, 47]. The GPU industry has since converged around scoped memory models, following early work on Heterogeneous Race-Free models [16, 19, 27, 32, 37]. However, researchers have identified shortcomings in scope-based memory models, such as the challenges of writing work-stealing runtimes [38, 45]. As such, some recent work has pushed against scope-based models in favor of using hardware-assisted protocols such as DeNovo [45].

The approaches taken to formalize GPU memory models follow directly from a large body of work performed to more rigorously characterize and then mathematically formalize memory models for CPUs and for programming languages. The memory models for x86, ARM, Power, and RISC-V, among others, have been formalized both axiomatically and operationally [3, 17, 29, 41, 42] and in most cases

the two approaches have been proven equivalent. As such, although the complexity of highly relaxed models may remain daunting to a general audience, these models are now reasonably well understood by experts. Likewise, memory models for C, C++, and Java have also been formalized in recent years [8, 10, 26, 31].

The formalism applied to memory models has additionally been backed by more pragmatic techniques based on testing. Issues in the C++11 memory model were resolved after testing revealed an unexpected contradiction [26, 30]. Many of the litmus tests capturing interesting synchronization patterns under memory consistency models have resulted from manual investigation of weak memory consistency models [3, 8, 29, 31, 42]. Others have been discovered through guided automatic generation [5, 13, 28, 49].

### 8.2 Beyond Standard Memory Models

In spite of the substantial gains made in understanding memory models over the past decade, many memory-related behaviors remain under-specified. There are still open questions regarding the behavior of syntactic dependencies between instructions [33], of relaxed atomic operations and so-called “out-of-thin-air” situations [7, 12], and of memory models for graphics APIs [24]. Solutions to the thin-air problem are being proposed [9, 11, 21–23, 39, 40, 46], but none has emerged victorious to date. Other work has found that for non-multi-copy-atomic memory models, in programs with multiple memory access sizes it is not always possible to restore sequential consistency via fences alone [15].

Furthermore, the behavior of many “non-standard” memory operations such as instruction fetches or page table walk accesses is an area that researchers have only recently started trying to formalize [43, 44]. Although most architectures provide standardized sequences for instruction memory synchronization and TLB shoot-downs, to date there is no broad consensus for how to completely formalize such operations in general.

The increasing presence of accelerators tightly coupled to traditional pipelines is adding yet more non-standard memory paths that must be considered in modern memory models. This paper presents one practical specification of how to incorporate accelerators and other forms of special-purpose caching into an otherwise general-purpose forward-compatible GPU programming model.

## 9 CONCLUSION

GPU architectures are evolving quickly to meet the needs of workloads in important domains such as machine learning. This fast evolution has resulted in special-purpose accelerators and memory system behavior becoming increasingly exposed at the architecture level to the programmer. GPU programming models must keep up: there is need for continued innovation in the architecture specifications that dictate the behavior of these workloads in order to ensure that the general-purpose GPU programming model retains forward- and backward-compatibility, while still enabling users to reach peak or near-peak performance.

In this work, we have explained the reasoning behind the development of the *proxy*, an abstraction over various kinds of special-purpose compute and memory acceleration, and we have described its integration into the NVIDIA PTX memory consistency model specification. Proxies fill a gap in the GPU programming model

by allowing architecture features such as texture caches, constant caches, virtual memory aliases, and accelerators to be incorporated into the memory model as first-class citizens. Proxy fences provide a synchronization mechanism that respects the intentionally relaxed behavior of the underlying architecture while nevertheless supporting programmers' need to write sound, composable code. The formalization and analysis tools we developed in support of this new model demonstrate the level of rigor that NVIDIA considers necessary before releasing a memory model publicly. We believe NVIDIA's proxy memory model will open the door for a range of new software interoperability paradigms and hardware acceleration mechanisms for future GPU generations.

## ACKNOWLEDGMENTS

The authors would like to thank David Nellans and the anonymous reviewers for their helpful feedback.

## REFERENCES

- [1] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezzine. 2012. Counter-example Guided Fence Insertion Under TSO. In *Proceedings of the Eighteenth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.
- [2] Jade Alglave, Mark Batty, Alistair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. 2015. GPU Concurrency: Weak Behaviours and Programming Assumptions. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
- [3] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. 2008. The Semantics of Power and ARM Multiprocessor Machine Code. In *Proceedings of the Fourth Workshop on Declarative Aspects of Multicore Programming (DAMP)*.
- [4] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. 2017. Don't Sit on the Fence: A Static Analysis Approach to Automatic Fence Insertion. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 39, Issue 2.
- [5] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data-mining for Weak Memory. In *Proceedings of the Thirty-Fifth ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
- [6] ARM. 2021. *Arm Architecture Reference Manual Armv8, for A-Profile Architecture*. <https://developer.arm.com/documentation/ddi0487/gb>
- [7] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. The Problem of Programming Language Concurrency Semantics. In *Proceedings of the Tenth European Symposium on Programming Languages and Systems (ESOP)*.
- [8] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *Proceedings of the Thirty-Eighth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.
- [9] Hans-J. Boehm. 2013. Prohibiting "Out of Thin Air" Results in C++14. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3786.htm>.
- [10] Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ Concurrency Memory Model. In *Proceedings of the Twenty-Ninth International Conference on Programming Language Design and Implementation (PLDI)*.
- [11] Hans-J. Boehm, Mark Batty, Olivier Giroux, Paul McKenney, Peter Sewell, and Francesco Zappa Nardelli. 2013. Specifying the Absence of "Out of Thin Air" Results. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3710.html>.
- [12] Hans-J. Boehm and Brian Densky. 2014. Outlawing Ghosts: Avoiding Out-of-thin-air Results. In *Proceedings of the Workshop on Memory Systems Performance and Correctness (MSPC)*.
- [13] James Bornholt and Emina Torlak. 2017. Synthesizing Memory Models from Framework Sketches and Litmus Tests. In *Proceedings of the Thirty-Eighth ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
- [14] Xing Fang, Jaejin Lee, and Samuel P Midkiff. 2003. Automatic Fence Insertion for Shared Memory Multiprocessing. In *Proceedings of the Seventeenth Annual International Conference on Supercomputing (SC)*.
- [15] Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell. 2017. Mixed-size Concurrency: ARM, POWER, C/C++11, and SC. In *Proceedings of the Forty-Fourth ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*.
- [16] HSA Foundation. 2015. HSA Platform System Architecture Specification. [http://www.hsafoundation.com/html/Content/PRM/Topics/06\\_Memory/mem\\_model.htm](http://www.hsafoundation.com/html/Content/PRM/Topics/06_Memory/mem_model.htm).
- [17] RISC-V Foundation. 2022. The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA. <https://riscv.org>.
- [18] Matt Godbolt. 2022. *Compiler Explorer*. <https://godbolt.org/>
- [19] 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 Nineteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
- [20] Daniel Jackson. 2002. Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology* 11, 2, 256–290.
- [21] Alan Jeffrey and James Riely. 2016. On Thin Air Reads Towards an Event Structures Model of Relaxed Memory. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*.
- [22] Alan Jeffrey, James Riely, Mark Batty, Simon Cooksey, Ilya Kaysin, and Anton Podkopaev. 2022. The Leaky Semicolon: Compositional Semantic Dependencies for Relaxed-Memory Concurrency. In *Proceedings of the 49th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*.
- [23] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-memory Concurrency. In *Proceedings of the Forty-Fourth ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*.
- [24] Khronos Vulkan Working Group. 2018. Vulkan 1.1.92 - A Specification. <https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html>.
- [25] Michael Kuperstein, Martin Vechev, and Eran Yahav. 2012. Automatic Inference of Memory Fences. In *Proceedings of the Tenth Conference on Formal Methods in Computer Aided Design (FMCAD)*.
- [26] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++11. In *Proceedings of the Thirty-Eighth ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
- [27] Daniel Lustig, Sameer Sahasrabudhe, and Olivier Giroux. 2019. A Formal Analysis of the NVIDIA PTX Memory Consistency Model. In *Proceedings of the Twenty-Third ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
- [28] Daniel Lustig, Andrew Wright, Alexandros Papakonstantinou, and Olivier Giroux. 2017. Automated Synthesis of Comprehensive Memory Model Litmus Test Suites. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
- [29] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. 2012. An Axiomatic Memory Model for POWER Multiprocessors. In *Proceedings of the Twenty-Fourth International Conference on Computer Aided Verification (CAV)*.
- [30] 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. In *ArXiv*, Vol. abs/1611.01507.
- [31] Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java Memory Model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.
- [32] Luc Maranget and Jade Alglave. 2015. Towards a Formalization of the HSA Memory Model in the cat Language. <https://hal.inria.fr/hal-01413251>
- [33] Paul E. McKenny, Alan Jeffrey, Ali Sezgin, and Tony Tye. 2016. P0422: Out-of-Thin-Air Execution is Vacuous. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0422r0.html>.
- [34] MLCCommons. 2021. *General MLPerf Submission Rules v0.3*. <https://docs.nvidia.com/cuda/parallel-thread-execution/index.html>
- [35] NVIDIA. 2017. NVIDIA Tesla V100 GPU Architecture. <http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf>.
- [36] NVIDIA. 2021. *CUDA C++ Programming Guide*. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>
- [37] NVIDIA. 2021. *PTX ISA :: CUDA Toolkit Documentation*. <https://docs.nvidia.com/cuda/parallel-thread-execution/index.html>
- [38] 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 (ASPLOS)*.
- [39] Marco Pavoititi, Simon Cooksey, Anouk Paradis, Daniel Wright, Scott Owens, and Mark Batty. 2020. Modular Relaxed Dependencies in Weak Memory Concurrency. In *Proceedings of the Twenty-Ninth European Symposium on Programming (ESOP)*.
- [40] Jean Pichon-Pharabod and Peter Sewell. 2016. A Concurrency Semantics for Relaxed Atomics That Permits Optimisation and Avoids Thin-air Executions. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.
- [41] 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. In *Proceedings of the Forty-Fifth ACM SIGPLAN*

- Symposium on Principles of Programming Languages (POPL).*
- [42] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. X86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. In *Communications of the ACM*, Vol. 53. Issue 7.
  - [43] Ben Simmer, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell. 2022. Relaxed virtual memory in Armv8-A. In *Proceedings of the Thirty-First European Symposium on Programming (ESOP)*.
  - [44] Ben Simmer, Shaked Flur, Christopher Pulte, Alasdair Armstrong, Jean Pichon-Pharabod, Luc Maranget, and Peter Sewell. 2020. ARMv8-A System Semantics: Instruction Fetch in Relaxed Architectures. In *Proceedings of the Twenty-Ninth European Symposium on Programming (ESOP)*.
  - [45] Matthew D Sinclair, Johnathan Alsop, and Sarita V Adve. 2015. Efficient GPU Synchronization Without Scopes: Saying No to Complex Consistency Models. In *Proceedings of the Forty-Eighth International Symposium on Microarchitecture (MICRO)*.
  - [46] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. 2017. Chasing Away Rats: Semantics and Evaluation for Relaxed Atomics on Heterogeneous Systems. In *Proceedings of the Forty-Fourth Annual International Symposium on Computer Architecture (ISCA)*.
  - [47] Tyler Sorensen and Alastair F. Donaldson. 2016. Exposing Errors Related to Weak Memory in GPU Applications. In *Proceedings of the Thirty-Seventh ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
  - [48] John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. 2017. Automatically Comparing Memory Consistency Models. In *Proceedings of the Forty-Fourth ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*.
  - [49] John Wickerson, Mark Batty, Tyler Sorensen, and George A Constantinides. 2017. Automatically Comparing Memory Consistency Models. In *Proceedings of the Forty-Fourth ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*.