Consistency and Coherence of the NVIDIA Grace-Hopper Superchip

Soham Bagchi
University of Utah
Salt Lake City, USA
soham.bagchi@utah.edu

Sanya Srivastava
Duke University
Durham, USA
sanya.srivastava@duke.edu

Reese Levine
University of California at Santa Cruz
Santa Cruz, USA
reeselevine@ucsc.edu

Tyler Sorensen
University of California at Santa Cruz
Santa Cruz, USA
tyler.sorensen@ucsc.edu

Ryan Stutsman
University of Utah
Salt Lake City, USA
stutsman@cs.utah.edu

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

Abstract

Modern heterogeneous processors like the NVIDIA Grace-Hopper Superchip tightly integrate CPU and GPU cores across a cache-coherent interconnect, with an implicit assumption that independently compiled CPU and GPU code can safely interact via shared memory. Yet the memory consistency and coherence of such systems remain empirically unvalidated. This paper presents the first systematic study of consistency and coherence on the Grace-Hopper. We empirically validate that the system enforces the Compound Memory Consistency Model (CMCM)—a theoretical prerequisite for correct independent compilation—using a novel heterogeneous litmus testing methodology spanning 1,960 test variants. We further introduce Value Propagation tests to reverse-engineer the underlying coherence mechanisms, revealing that internal GPU coherence relies on write-throughs and self-invalidations rather than classical writer-initiated invalidations, while global CPU-GPU coherence is maintained via directory-based invalidations consistent with an AMBA CHI-like protocol. These results establish the CMCM as a concrete architectural target for heterogeneous systems and provide the first empirical characterization of GPU and CPU-GPU coherence mechanisms in a commercial heterogeneous processor.

CCS Concepts: • **Computer systems organization → Heterogeneous (hybrid) systems**.

Keywords: Memory Consistency, Cache Coherence, Grace Hopper

ISMM '26, Boulder, CO, USA

ACM ISBN 979-8-4007-2720-7/2026/06
https://doi.org/10.1145/3814942.3816134

ACM Reference Format:
Soham Bagchi, Sanya Srivastava, Reese Levine, Tyler Sorensen, Ryan Stutsman, and Vijay Nagarajan. 2026. Consistency and Coherence of the NVIDIA Grace-Hopper Superchip. In *Proceedings of the 2026 ACM SIGPLAN International Symposium on Memory Management (ISMM '26), June 16, 2026, Boulder, CO, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3814942.3816134

1 Introduction

Heterogeneous processors, such as the NVIDIA Grace Hopper Superchip, tightly integrate distinct processing units, like an ARM multi-core CPU and an NVIDIA GPU, across a cache-coherent interconnect. To maximize performance and minimize kernel launch overheads, modern software increasingly relies on fine-grained, cross-device synchronization over this shared memory [37, 43].

However, reasoning about these interactions is complicated by the system's bifurcated software stack. In these heterogeneous setups, CPU codes are compiled by a CPU compiler and GPU codes by a GPU compiler. There is a fundamental, implicit assumption that these independently compiled codes can "play well together" when executing concurrently. Figure 1 illustrates this assumption in practice using a standard producer-consumer synchronization pattern. A CPU producer thread, written in C++, compiles down to ARM instructions utilizing an unscoped release store (`STLR`) to signal via a flag variable that data is ready. Concurrently, a GPU consumer thread, written in CUDA, compiles down to PTX instructions utilizing an explicitly scoped acquire load (`ld.acq.sys`) to read the flag.

Traditionally, the semantics of such concurrent memory operations are governed by a Memory Consistency Model (MCM). However, as Figure 1 highlights, the MCMs governing CPUs and GPUs are fundamentally mismatched. CPU MCMs (like ARM) are typically stronger and apply their ordering guarantees uniformly across all threads. In contrast, GPU MCMs (like PTX) are heavily relaxed and utilize an implicit, scoped thread hierarchy—e.g., Cooperative Thread Array (`cta`) , `gpu`, `system`—to enforce ordering.

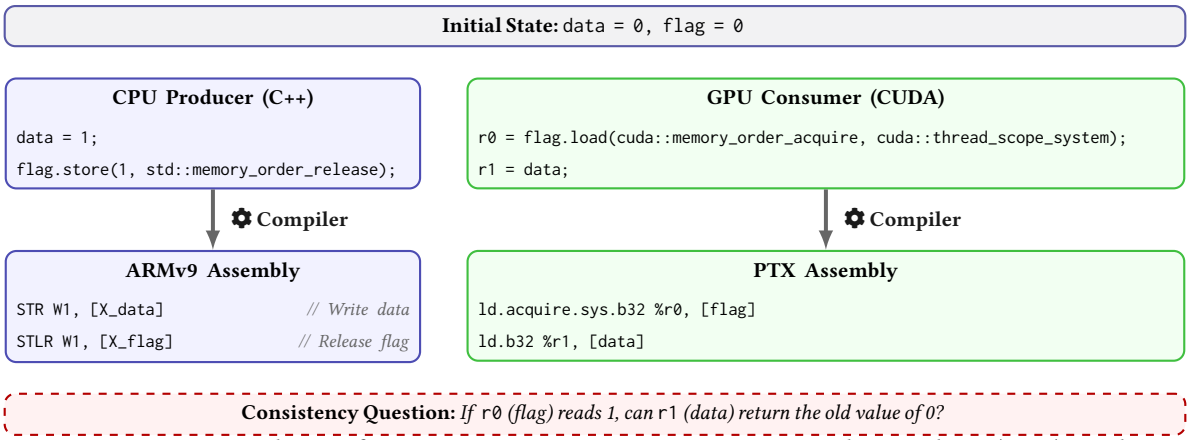

Figure 1. Heterogeneous Compilation of a Message Passing Litmus Test. An unscoped CPU release (STLR) synchronizes with a system-scoped GPU acquire (`ld.acquire.sys.b32`). This semantic mismatch raises a critical question: will the hardware successfully bridge these models to prevent the consumer from reading stale data? The Compound Memory Consistency Model (CMCM) theoretically dictates that it must, but hardware validation is required.

This mismatch creates a critical semantic tension: when the CPU which enforces a stronger MCM interacts with a GPU that enforces a weaker MCM, what is the resulting consistency model of the unified machine? If the hardware enforces the stronger CPU model globally, it would require infeasible internal changes to the GPU architecture: among other things, it would require doing away with the scope hierarchy which is fundamental to GPU architectures. Conversely, if it defaults to the weaker GPU model, the previously compiled CPU code would silently break, violating the requirement that the devices can interact seamlessly.

Recent theoretical work introduced the concept of Compound Memory Consistency Models (CMCMs) to resolve this [15]. The authors argue that a heterogeneous system should act as a compositional amalgamation of its parts: CPU threads continue to adhere strictly to the CPU MCM, while GPU threads adhere to the GPU MCM. They demonstrate that enforcing this CMCM is mathematically necessary to ensure independent compilation; going back to Figure 1, this means the hardware must ensure that the ARM STLR release synchronizes with the PTX `ld.acq.sys` to prevent the GPU consumer from reading stale data. It must do this dynamically, without requiring either instruction to be recompiled to match the other's memory model. But do heterogeneous systems actually enforce the CMCM? There has been no study to date on this. This requires a new approach of heterogeneous litmus testing—systematically running inter-device litmus tests to check for properties. Adherence to CMCM would place the existing practice of independent compilation on a firmer empirical foundation, and vindicate CMCM as a natural target for future heterogeneous machines. On the contrary, discovering violations would alert the community to take steps to resolve this problem.

Validating that the hardware enforces the CMCM answers what the system guarantees. An equally important question is how it enforces these guarantees—i.e., what are the underlying coherence mechanisms? It is a commonly-held belief that GPUs do not employ classical coherence protocols that obtain write permissions on a store by invalidating sharers [24]. It is thought that such protocols eschew writer-initiated invalidations and instead enforce coherence via writebacks and self-invalidations.

Are GPU coherence protocols *actually* different from CPU coherence protocols? Focusing on the implementation of shared memory at the heterogeneous level, the Grace-Hopper Superchip reportedly employs its NVLink-C2C interconnect to provide a cache-coherent memory domain shared between the CPU and the GPU [25]. While it is speculated that the underlying coherence protocol is based on ARM's AMBA Coherent Hub Interface (CHI) protocol [4], this again remains a hypothesis that has yet to be empirically validated through rigorous study. Just as understanding CPU cache coherence spurred software optimizations for multi-core CPUs [9, 28], we expect similar gains for GPU and CPU-GPU programming with deeper coherence insights.

In this paper, we present the first systematic exploration of the consistency and coherence of the Grace-Hopper Superchip. To evaluate the semantics, we introduce a novel heterogeneous litmus testing methodology with tests spanning the CPU and GPU. Strategically, we focus this methodology primarily on Message-Passing (MP) litmus tests, generating a massive suite of 1,960 test variants for the Grace-Hopper Superchip. We prioritize MP patterns because they directly map to the ubiquitous producer-consumer synchronization patterns found in real-world CPU-GPU applications.

While traditional litmus testing is highly effective for verifying consistency compliance (checking "is this outcome allowed?"), it is fundamentally a falsification tool. It relies

on stress-testing to hunt for rare, forbidden behaviors, making it incapable of disambiguating the underlying hardware mechanisms that actually produced the outcome.

To deduce the underlying coherence mechanisms, we propose a novel empirical methodology: Value Propagation (VP) tests. Rather than hunting for rare violations, VP tests are designed to explicitly reverse-engineer coherence protocols. By analyzing the values returned by unsynchronized reads—and correlating the timing of those new values with precise cache latencies—VP tests deduce exactly when a write reaches a specific level of the cache hierarchy. This allows us to map the protocol's behavior, deducing, for example, whether a GPU store instruction propagates its value directly to other L1 caches, or if it merely writes through to the shared L2 cache.

Our investigation yields four primary contributions:

1. Validation of Compound Consistency: We empirically confirm that the Grace-Hopper Superchip, as a unified system, enforces the Compound ARM/PTX Memory Consistency Model. This result validates the safety of independent CPU/GPU compilation, guaranteeing that memory operations interacting across the heterogeneous boundary execute correctly without requiring software recompilation. Furthermore, coupled with its theoretical foundations, this empirical observation establishes the CMCM as a concrete architectural target for the design of future heterogeneous devices.

2. Discovery of Stronger-than-Model Behaviors: In evaluating the system's semantics, we uncovered instances where the hardware enforces orderings significantly stronger than predicted by the PTX specification. Specifically, atomic Read-Modify-Write (`rmw`) operations executed with narrow programmer-specified scopes (e.g., `cta` or gpu) exhibited global visibility when synchronizing both internally within the GPU and externally with the CPU. This finding challenges current programming assumptions and raises questions for future specifications.

3. Value Propagation (VP) Tests: We introduce VP tests to reverse-engineer commercial coherence protocols without documentation. This provides a new tool for the architecture community to distinguish various types of coherence protocols including eager (writer-initiated) and lazy (self-invalidating) coherence protocols.

4. Grace-Hopper Coherence Architecture: We provide the first empirical map of the Grace-Hopper coherence hierarchy. We demonstrate that global coherence across the NVLink-C2C interconnect is maintained via directory-based invalidations consistent with an AMBA CHI [4]), while internal GPU coherence relies on write-throughs and self-invalidations.

2 Background and Related Work

2.1 Memory Consistency Models

A Memory Consistency Model (MCM) is a crucial component of the instruction set architecture, defining the rules that govern the ordering of memory operations. Fundamentally, an

Initial State: X=0 ∧ Y=0

T1: Prod	T2: Cons
m_1: St(X, 1)	m_3: r0 = Ld(Y)
m_2: St(Y, 1)	m_4: r1 = Ld(X)

(a) Weak MP

T1: Prod	T2: Cons
m_1: St(X, 1)	m_3: r0 = Ld_{acq}(Y)
m_2: St_{rel}(Y, 1)	m_4: r1 = Ld(X)

(b) Acq-Rel MP (RC)

T1: Prod	T2: Cons
m_1: St(X, 1)	m_3: r0 = Ld(Y)
DMB	DMB
m_2: St(Y, 1)	m_4: r1 = Ld(X)

(c) Message Passing with ARM DMB Barrier

T1: Prod	T2: Cons
m_1: St(X, 1)	m_3: r0 = Ld(Y)
$fence_{sc}^{gpu}$	$fence_{sc}^{gpu}$
m_2: St(Y, 1)	m_4: r1 = Ld(X)

(d) Message Passing with Scoped Fences

T1: Prod	T2: Cons
m_1: St(X, 1)	m_3: r0 = Ld_{acq}^{cta}(Y)
m_2: St_{rel}^{cta}(Y, 1)	m_4: r1 = Ld(X)

(e) cta-Scoped Acq-Rel MP

T1: Prod	T2: Cons
m_1: St(X, 1)	m_3: r0 = Ld_{acq}^{gpu}(Y)
m_2: St_{rel}^{gpu}(Y, 1)	m_4: r1 = Ld(X)

(f) gpu-Scoped Acq-Rel MP

Allowed Behaviors: X=1 ∧ Y=1 ∨ X=0 ∧ Y=0 ∨ X=1 ∧ Y=0
Outcome: r0=1 ∧ r1=0

Figure 2. Message Passing Litmus Tests

MCM addresses the question: **what value should a read return?** Most major commercial homogeneous processor architectures adhere to precisely defined MCMs [34, 39].

Ordering. Arguably, the defining characteristic of MCMs is the preservation (or lack thereof) of memory ordering. This is effectively illustrated using a *litmus test*: a short, multi-threaded program snippet designed to expose memory ordering behaviors. Consider the classic message passing (mp) litmus test, shown in Figure 2a: thread T1 (the producer) performs two store operations to distinct memory locations, while thread T2 (the consumer) performs two load operations on the same locations. The key question this test probes is: if the load m_3 observes the updated value of 1, is it possible for the subsequent load m_4 to still observe the initial value of 0? This outcome is disallowed by MCMs such as x86-TSO, which enforces the $store \rightarrow store$ ordering and $load \rightarrow load$ orderings: if m_3 sees the update of m_2, the preceding store m_1 must have completed before m_4 forcing m_4 to also see the updated value.

Multi-copy atomicity. Another key aspect of MCMs is how store values propagate. In *multi-copy atomic* (MCA) memory models [24] (enforced by CPUs like x86, ARMv8, and RISC-V), once a store is visible to another processor, all subsequent loads in logical time will observe the updated value. In contrast, GPUs like NVIDIA PTX permit non-MCA behaviors. This work focuses on the ordering aspects of Grace-Hopper, leaving the interaction between ARM's MCA properties and PTX's non-MCA properties for future investigation.

Read-Modify-Writes. CPUs and GPUs define atomic Read-Modify-Write (RMW) instructions (like fetch-and-add) in their MCMs. We are particularly interested in the semantics and the implementation of CPU-GPU RMW interactions.

Language-level Memory Models. High-level languages (e.g., C, CUDA, JavaScript) have memory models with synchronization primitives [20, 40]. A key compilation task is to ensure provably correct mappings of these primitives to the instruction set, matching their memory ordering guarantees at the machine level.

2.2 Coherence Protocols

A processor's MCM is determined by the interplay between how its pipeline may reorder memory operations and the ordering guarantees enforced by its cache coherence protocol. Processor cores interact with the coherence protocol through an interface consisting of reads, writes, and fences [24].

Most coherence protocols, especially CPU protocols, enforce the Single-Writer-Multiple-Reader (SWMR) invariant: there is either a single processor core that holds write permissions or there could be multiple cores that hold read permissions. SWMR is typically enforced by obtaining exclusive permissions on a cache line upon a write by invalidating the cache lines that are shared in other processor cores. Examples include the classic MOESI-style protocols and industrial-strength protocol specifications such as CHI [4] and CXL [7].

Not all protocols enforce SWMR, however. Some protocols eschew writer-initiated invalidations and instead rely on writebacks and self-invalidations. For example, to synchronize, the dirty cache lines in the local cache are flushed down to the lower level, and shared cache lines are self-invalidated. It is thought that GPU protocols follow this paradigm [24].

The observable difference between these two paradigms is the timing of value propagation. Under writer-initiated invalidation, a write by one core immediately invalidates cached copies in other cores; a subsequent read by any core will therefore observe the new value, even without intervening synchronization. Under writeback and self-invalidation, a write does not disturb other cores' caches. The new value becomes visible to other cores only after the writer flushes its cache (typically on a release) and the reader self-invalidates (typically on an acquire). This distinction is the foundation of our Value Propagation methodology

2.3 Related Work

Memory Consistency Validation.

One of the early significant works in this area is TSOtool [18], which automatically generated pseudo-random, multi-threaded programs with races, executed them on multiprocessor systems, and verified the results against the target machine's MCM. Building upon this concept, the approach of litmus testing fueled a resurgence of interest in MCM validation in the 2010s [1, 34]. Unlike the automated generation of TSOtool, litmus tests are typically manually crafted to systematically explore the spectrum of ordering and Multi-Copy Atomicity (MCA) behaviors defined by MCMs. In its basic form, a litmus test, such as the message passing (mp) example discussed earlier, is executed repeatedly (often millions of

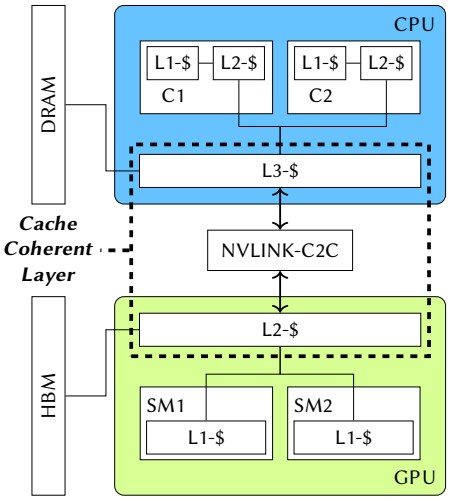

Figure 3. NVIDIA Grace-Hopper System Architecture

times), and the observed output patterns are used to infer the enforced MCM. Subsequent research has focused on enhancing the efficiency of litmus testing [22] and on augmenting test scenarios with stress threads to expose a wider range of behaviors, particularly in the context of GPUs [21].

In this work, we leverage these established best practices to conduct, to the best of our knowledge, the first empirical validation of the individual MCM behaviors of both the Grace CPU and the Hopper GPU. Moreover, we address a significant gap in the existing literature by presenting the first attempt to empirically validate the combined MCM behavior of a heterogeneous processing device.

Coherence Protocol Characterization. In contrast to the extensive body of work on MCM validation, the characterization and reverse engineering of cache coherence protocols have received comparatively less attention. Early work by Molka et al [16, 23]. created specific benchmark programs for characterizing the coherence latencies and bandwidth of NUMA systems such as Intel Nehalem and AMD Shanghai. More recent work by Dai et al. [8] has reverse engineered the mesh interconnect in Intel server processors for possible side channel attacks. To our knowledge, there is no existing work that has attempted to reverse engineer the mechanisms through which GPUs, such as Hopper, enforce consistency.

Grace-Hopper Characterization Studies. Prior studies on the Grace-Hopper Superchip have primarily focused on performance benchmarking of memory access latencies and bandwidth between the CPU and GPU using tools like CXL-Bench [42], as well as performance in multi-chip configurations [14]. Other work has investigated memory allocation behaviors, migration policies, and address translation for HPC applications [5, 33, 35]. To our knowledge, no existing research has empirically studied the memory consistency behaviors or the underlying coherence protocol mechanisms of the Grace-Hopper Superchip, which is the central focus of this work.

3 Unpacking Grace-Hopper's Consistency and Coherence: Knowns and Unknowns

This section outlines what is known and unknown about memory consistency and coherence in the NVIDIA Grace-Hopper architecture. We synthesize vendor documentation to establish a baseline, highlight information gaps and ambiguities, and conclude by stating the open research questions driving our empirical investigation.

3.1 The Grace-Hopper Superchip Architecture

The Grace-Hopper Superchip features a Grace CPU and a Hopper GPU, each with separate memory hierarchies, connected by an NVLink-C2C Interconnect that vendor documentation claims provides hardware cache coherence. Details are in Table 1 and a visual overview is shown in Figure 3.

CPUs and GPUs employ different memory allocators for sharing resulting in different implementation approaches. Historically, NVIDIA GPUs relied on *cudaMalloc()* coupled with explicit *cudaMemcpy()* for CPU-GPU data transfer. Subsequent CUDA versions introduced *cudaMallocHost()* for DMA access to pinned CPU memory and *cudaMallocManaged()* for unified memory with potential page migration. The Grace-Hopper architecture elegantly enables direct CPU and GPU access to memory allocated via standard *malloc()*, enforced via hardware cache coherence. Given our focus on the hardware-level consistency and coherence, this study exclusively examines memory regions allocated using *malloc()*.

Table 1. Grace CPU and Hopper GPU Hardware

Feature	Grace CPU	Hopper GPU
Cores	72 ARMv9 Neoverse V2	132 SMs (128 cores/SM)
L1 Cache	64KB i-cache, 64KB d-cache	256KB per SM
L2 Cache	1MB per-core	51MB
L3 Cache	114MB	×
Mem Type	LPDDR5X	HBM3
Mem Capacity	480GB	96GB
Mem Bandwidth	512GB/s	4TB/s
NVLink-C2C Interconnect		
Mem Bandwidth	900GB/s	

3.2 Grace-Hopper: Consistency

What is Known. The Grace-Hopper Superchip integrates an ARM-based CPU, which adheres to the ARMv9 MCM, and a Hopper GPU, which follows the PTX MCM. Both of these MCMs have been specified in prose and formalized and mechanized using tools such as Alloy, allowing for rigorous analysis of their behaviors. Next, we will briefly outline the key characteristics of the ARMv9 and PTX MCMs, utilizing litmus tests to illustrate their ordering rules.

Grace MCM: ARM v9 enforces a variant of Release Consistency (RC), a relaxed MCM. In essence, ARMv9 does not inherently enforce ordering between ordinary load and store operations. Revisiting the message passing (MP) litmus test illustrated in Figure 2a, the outcome where the consumer observes the write to one memory location but not the write to the other (e.g., r0 = 1, r1 = 0) is permissible under the ARMv9 MCM in the absence of explicit synchronization. To enforce specific orderings, ARMv9 relies on explicit synchronization primitives, such as release and acquire operations, or memory barrier instructions. As demonstrated in Figure 2b, marking the store operation to m_2 as a release and the load operation from m_3 as an acquire prevents the aforementioned outcome. Memory barrier instructions, such as DMB, offer another solution; a producer-side $store \rightarrow store$ `DMB ST` combined with a consumer-side $load \rightarrow load$ `DMB LD` would also disallow this.

Hopper MCM: PTX is also a relaxed MCM, meaning it does not guarantee program order for ordinary memory operations. A key distinction from CPU MCMs is that PTX is *scoped*.

GPUs expose their hierarchical architecture through a thread hierarchy organized into scopes. The Cooperative Thread Array (`cta`) scope encompasses all threads belonging to one CTA; principally, all threads within a CTA are guaranteed to be mapped to the same Streaming Multiprocessor (SM). Synchronization within the `cta` scope is highly efficient because these threads share the L1 cache, allowing synchronizing loads and stores to remain local without needing to propagate to the L2 cache or beyond.[1] The `gpu` scope refers to all threads belonging to a single GPU, which share a common L2 cache. Synchronization at the `gpu` scope necessitates that loads and stores involved in the synchronization propagate to the L2 cache, but not necessarily further. Finally, the `system` scope includes all threads across the entire system, encompassing not only GPU threads but also threads on the CPUs and other GPUs. Synchronization at the `system` scope requires that the involved loads and stores are propagated to main memory to ensure visibility across all participating entities.

In PTX, every memory operation can be annotated not only with a memory ordering label (e.g., release and acquire, similar to ARM) but also with a scope label (specifying `cta`, `gpu`, or `system`). The scope labels attached to the synchronizing memory accesses, combined with the scope of the threads involved in the synchronization, give rise to the notion of *moral strength* in synchronization. **Synchronizing operations are considered *morally strong* if and only if the scope of each operation encompasses the scope of the thread performing the other operation involved in the synchronization.**

Consider the message passing (MP) litmus test shown in Figure 2e, assuming that threads T1 and T2 belong to different CTAs. In this scenario, the release operation on m_2 and the acquire operation on m_3 are both marked with `cta` scope. Consequently, the scope of the operation on m_2 (which is `cta`) does not encompass the scope of the other synchronizing thread, T2 (which resides in a different CTA). Therefore, this release-acquire synchronization is not morally strong, and

[1] PTX also supports `cluster` scope (which is between `cta` and `gpu`) but our evaluation didn't observe any differences between `cta` and `cluster` scopes

the outcome where T2 observes the write to m_2 but not the write to m_1 (resulting in r0 = 1, r1 = 0) is permitted under the PTX memory model.

One way to prevent this outcome would be to ensure that threads T1 and T2 are mapped to the same CTA. Alternatively, if the operations on m_2 and m_3 are marked with gpu scope, as shown in Figure 2f, then the moral strength condition would be satisfied, as the gpu scope includes all CTAs within the GPU, thus again disallowing the outcome.

Like ARM, PTX provides memory fences offering release, acquire, release-acquire, and SC fences, configurable with `cta`, gpu, or `system` scope. To prevent r0=1, r1=0 in the MP test, a gpu-scoped fence with release (or stronger) between T1's stores and a gpu-scoped fence with acquire (or stronger) between T2's loads would suffice. gpu scope guarantees ordering visibility across the entire GPU.

What is Unknown. While the ARMv9 MCM is precisely defined and has been empirically validated on various ARM processors [12, 13, 17, 27, 36], the specific ARMv9 implementation within NVIDIA's Grace CPU remains publicly empirically unvalidated. Similarly, despite the formal specification and documentation of the PTX memory model by NVIDIA [26], there has been a notable lack of comprehensive empirical validation of its behavior on actual NVIDIA GPU hardware. This leads to two questions: **1) Does the Grace CPU indeed strictly adhere to the ARMv9 MCM as documented? 2) Does the Hopper GPU strictly adhere to the PTX MCM as defined?**

More fundamentally, a critical gap exists in our understanding of how the Grace CPU and the Hopper GPU interact in synchronization scenarios. Consider the inter-device message passing (mp) litmus test shown in Figure 4a: if the CPU acts as the producer and the GPU as the consumer, and the store at m_2 is a release while the load at m_3 is a system-scoped acquire, would the outcome r0 = 1, r1 = 0 be disallowed? What if the load at m_3 is only an acquire with gpu scope or `cta` scope—would the outcome still be prevented? What if the store at m_2 and the load at m_3 are not explicitly marked as release and acquire, respectively? These are fundamental questions regarding the combined memory consistency behavior of the heterogeneous Grace-Hopper architecture that remain unanswered by current specifications.

As introduced in Section 1, Goens et al. [15] showed that a heterogeneous system must enforce a Compound Memory Consistency Model (CMCM)—an amalgamation of its component MCMs—to enable independent compilation. A key rule of the CMCM is that CPU unscoped operations are treated as system-scoped when synchronizing with a GPU that employs a scoped model.

Considering the inter-device message passing (mp) litmus test in Figure 4a, if the store at m_2 by the CPU is a release operation (enforcing $m_1 \rightarrow m_2$ under ARM rules) and the load at m_3 by the GPU is an acquire operation with system scope (enforcing $m_3 \rightarrow m_4$ under PTX rules), the synchronization is deemed morally strong. This combination of orderings and scope ensures that the outcome r0 = 1, r1 = 0 is disallowed. Conversely, if the load at m_3 is tagged with GPU scope (as shown in Figure 4b), the inter-device synchronization is not considered morally strong, allowing the outcome r0 = 1, r1 = 0. Similarly, if the release or acquire labels are omitted from the respective memory operations (as shown in Figure 2a), the necessary ordering constraints are absent, again permitting the outcome.

Goens et al. [15] showed heterogeneous devices need to enforce compound memory models (or stronger) to enable independent compilation—the ability to compile CPU code with standard CPU compilers and GPU code with GPU compilers for synchronized programs (like in CUDA). Thus, Grace Hopper's existing CPU/GPU compilation workflow implicitly assumes it enforces a compound ARM/PTX MCM (or stronger). Despite this foundational reliance, no prior work has empirically validated this assumption, leading to the crucial question: **3) Does the Grace Hopper indeed enforce the ARM/PTX compound memory consistency model**?

3.3 Grace-Hopper: Coherence

While ARMv9 and PTX memory models are well-documented, the internal cache coherence protocols of Grace CPU and Hopper GPU are less known. Moreover, the coherence mechanisms across the NVLink-C2C interconnect between them are largely undocumented beyond NVIDIA's claim of 'hardware cache coherence'.

Grace Coherence. The Grace CPU is an ARM-based CPU designed by NVIDIA. While the internal cache coherence protocol is not publicly documented, ARM has published coherence specifications like AMBA CHI to ensure coherence among cores within a chip. Protocols like CHI typically enforce the SWMR invariant by invalidating shared copies upon a write operation. In contrast, academic research has explored alternatives, such as VIPS [31], DeNovo [6], and TSO-CC [11], which aim to enforce relaxed consistency models without relying on writer-initiated invalidations, instead employing writebacks and self-invalidations. The specific coherence protocol implemented within the Grace CPU remains unknown. **4) Does the Grace CPU actually enforce SWMR via writer-initiated invalidations?**

Hopper Coherence. NVIDIA has not disclosed the specific mechanisms its GPUs use to enforce the PTX MCM. Academic research has extensively argued against the efficiency of conventional invalidation-based cache coherence protocols for GPUs [2, 6, 24, 29, 38]. Notably, a paper by NVIDIA [30] proposed a coherence protocol based on writebacks and self-invalidations for GPU L1 caches, although it did not explicitly confirm its deployment in NVIDIA products. Even within those approaches, different variants exist. Ordinary (non-synchronizing) writes could be buffered in the L1 cache and flushed to lower levels upon a release operation. Alternatively, a write-through protocol could be employed, where every

write propagates to the lower-level cache, and a release operation only needs to ensure that all preceding writes have reached that lower level. **5) Does the Hopper GPU utilize a coherence protocol based on writebacks and self-invalidations? 6) If so, does it employ a write-back or a write-through strategy for non-synchronizing writes?**
Grace-Hopper Coherence. NVIDIA states that Grace CPU and Hopper GPU are 'hardware coherent' [25] via NVLink-C2C. It is speculated (without official confirmation) that AMBA CHI is employed for CPU-GPU coherence [14]. However, the detailed mechanisms of CPU-GPU coherence are unknown, particularly if the GPU uses writeback/self invalidation, unlike CHI's typical SWMR protocol. **7) How does ARM internal coherence integrate with GPU internal coherence over NVLink-C2C, and what precisely is 'hardware coherence' in this context?**

4 Validating Memory Consistency

In this section, we detail the methodology and present the results of our validation efforts, aimed at answering the questions posed in the previous section regarding the MCM of the Grace CPU, the Hopper GPU, and their combined behavior within the Grace-Hopper Superchip (questions one through three). For our validation suite, we predominantly employed variants of the message passing (MP) litmus test for reasons outlined in §1.

Recall that this choice was driven by two key factors: first, the MP test effectively captures the producer-consumer communication patterns that are prevalent in typical Grace-Hopper workloads; and second, the existence of numerous well-established variants of the MP test allows for a comprehensive exploration of different memory ordering scenarios.

We now describe our efforts to validate and identify the memory consistency model observed in the NVIDIA Grace-Hopper Superchip. Before attempting to experiment with the heterogeneous testing environment, we validated the consistency of each processor individually.

4.1 Validating Individual MCMs

We first validated the consistency of each processor individually using the MP variants in Table 2a. For the Grace CPU, we generated all seven variants using *diyone7* and executed each 225M times using *litmus7*, and additionally, we varied both ARM fence types (DMB.SY and DMB.ST/LD). For the Hopper GPU, we expanded the base variants with the scope and fence combinations in Table 2b: for each variant, we generated 3 scope variations for Y (ensuring the scope of Y is at least as broad as X) and 10 fence variations (no fence, plus three scopes for each of three fence types), yielding 560 litmus tests. These were executed using the *parallel testing environment* [21] with producer and consumer threads on different CTAs, and all threads concurrently performing *memory stressing* to maximize the likelihood of exposing weak behaviors.

Table 2. Message Passing Variants

	Test	Producer	Consumer
(a) Memory Ordering Variants			
	No-Fence		
1	MP (rlx/rlx)	$St_{rlx}(X)$ $St_{rlx}(Y)$	$Ld_{rlx}(Y)$ $Ld_{rlx}(X)$
2	MP (rel/acq)	$St_{rlx}(X)$ $St_{rel}(Y)$	$Ld_{acq}(Y)$ $Ld_{rlx}(X)$
3	MP (rlx/acq)	$St_{rlx}(X)$ $St_{rlx}(Y)$	$Ld_{acq}(Y)$ $Ld_{rlx}(X)$
4	MP (rel/rlx)	$St_{rlx}(X)$ $St_{rel}(Y)$	$Ld_{rlx}(Y)$ $Ld_{rlx}(X)$
	Fence (f.)		
5	MP (fence pc)	$St_{rlx}(X)$ **f.** $St_{rlx}(Y)$	$Ld_{rlx}(Y)$ **f.** $Ld_{rlx}(X)$
6	MP (fence p)	$St_{rlx}(X)$ **f.** $St_{rlx}(Y)$	$Ld_{rlx}(Y)$ $Ld_{rlx}(X)$
7	MP (fence c)	$St_{rlx}(X)$ $St_{rlx}(Y)$	$Ld_{rlx}(Y)$ **f.** $Ld_{rlx}(X)$
(b) Scope Variants			
	Candidate	**Variants**	
	X	`cta`, `gpu`	
	Y	`cta`, `gpu`, `system`	
	Fence	rlx, acq-rel, sc, none	
	Fence Scopes	`cta`, `gpu`, `system`	

Table 3. Grace Message Passing Weak Behaviors. ✓ = weak behavior observed; × = no weak behavior observed. The subscripts on `fence` denote where the fence appears: p = producer side, c = consumer side, pc = both. Suffixes (DMB.ST, DMB.LD, DMB.SY) specify the fence type.

	Litmus Test	Weak Behaviors
1	rlx/rlx	✓
2	rel/acq	×
3	rlx/acq	✓
4	rel/rlx	✓
5.1	fence $pc_{DMB.ST/LD}$	×
5.2	fence $pc_{DMB.SY}$	×
6.1	fence $p_{DMB.ST}$	✓
6.2	fence $p_{DMB.SY}$	✓
7.1	fence $c_{DMB.LD}$	✓
7.2	fence $c_{DMB.SY}$	✓

Grace CPU Results. As expected, the weak outcome was forbidden only when both sides supplied the required ordering—a release or producer-side fence on the producer *and* an acquire or consumer-side fence on consumer (variants 2, 5). In all other variants (1, 3, 4, 6, 7) at least one side was missing the required ordering, and the weak outcome was observed (Table 3).

Hopper GPU Results. Across all 560 litmus tests, whenever a morally strong release-acquire synchronization was established (i.e., both the release and acquire at gpu scope or higher), no weak behaviors were observed. Conversely, in all other scenarios—including insufficient scope and asymmetric cases where only one side had a release or acquire—weak behaviors were consistently observed. This confirms accurate enforcement of the PTX MCM and demonstrates that the microarchitecture implements scoped ordering aggressively: weak

behaviors appear precisely and only when morally strong synchronization is absent.

4.2 Evaluating Grace-Hopper's Memory Consistency

Does the Grace-Hopper Superchip as a whole adhere to the ARM/PTX Compound Memory Consistency Model (CMCM)? We extended our litmus testing methodology to probe inter-device consistency, building on Srivastava's framework [32]. Starting from the base variants in Table 2, we generated tests in both communication directions (CPU producer/GPU consumer and vice versa). Since the ARM architecture lacks explicit scope labels, CPU operations are unscoped in all variants. This yielded a total of 1,960 MP variants. Each was executed millions of times with memory stressing [21] on both devices.

Expected Behavior. As discussed in §3.2, the ARM/PTX CMCM dictates two rules. First, because CPU operations lack scopes, unscoped CPU operations are treated as system-scope. Second, morally-strong CPU-GPU synchronization requires a release/acquire pair whose scopes each encompass the other thread. We therefore expect weak behaviors precisely when scope or ordering semantics are insufficient.

Initial State: X=0 ∧ Y=0

CPU Prod.	GPU Cons.
m1: St(X, 1)	*m3:* r0 = Ld^{sys}_{acq}(Y)
m2: St_{rel}(Y, 1)	*m4:* r1 = Ld(X)

(a) `system`-scope RC (WB ×)

CPU Prod.	GPU Cons.
m1: St(X, 1)	*m3:* r0 = Ld^{gpu}_{acq}(Y)
m2: St_{rel}(Y, 1)	*m4:* r1 = Ld(X)

(b) gpu-scope RC (WB ✓)

CPU Prod.	GPU Cons.
m1: St(X, 1)	*m3:* r0 = Ld^{cta}_{acq}(Y)
m2: St_{rel}(Y, 1)	*m4:* r1 = Ld(X)

(c) `cta`-scope RC (WB ✓)

CPU Prod.	GPU Cons.
m1: St(X, 1)	*m3:* r0 = Ld^{sys}_{rlx}(Y)
m2: St_{rel}(Y, 1)	*m4:* r1 = Ld(X)

(d) `system`-scope No RC (WB ✓)

GPU Prod.	CPU Cons.
m1: St(X, 1)	*m4:* r0 = Ld_{acq}(Y)
m2: $\text{fence}^{cta}_{acq-rel}$	
m3: St^{gpu}_{rel}(Y, 1)	*m5:* r1 = Ld(X)

(e) gpu-scope RC (WB ✓)

Figure 4. Scoped Message Passing with Release-Acquire. Behavior of interest: $r0 = 1 \wedge r1 = 0$. WB✓ = weak behavior observed; WB× = no weak behavior observed.

Results: A Validation of the Compound Model Table 4 presents the results of our heterogenous litmus tests. Crucially, this table lists **only** the scenarios where weak behaviors were detected. By correlating these observed failures with the theoretical requirements of the CMCM, we derive two key conclusions that validate this model:

Weak Behaviors from Insufficient Scope [Rows 1, 3-5, 8-9, 12-15, 19-22]. The CMCM dictates that for a GPU thread to synchronize with a CPU thread, memory operations must use System scope (Figure 4a). In the tests corresponding to these rows, we purposely restricted the synchronization Y to narrower scopes (CTA or GPU). In every such instance,

Table 4. Heterogeneous Message Passing Weak Behaviors

	X	Y	P. Fence	C. Fence	Producer	Consumer
1	gpu	gpu	Acq-Rel	Acq-Rel^{cta}	CPU_{rlx}	GPU_{rlx}
2	gpu	system	Acq-Rel	Acq-Rel^{cta}	CPU_{rlx}	GPU_{rlx}
3	cta	cta	Acq-Rel	×	CPU_{rlx}	GPU_{rlx}
4	cta	gpu	Acq-Rel	×	CPU_{rlx}	GPU_{rlx}
5	cta	gpu	×	×	CPU_{rlx}	GPU_{rlx}
6	cta	system	×	×	CPU_{rlx}	GPU_{rlx}
7	cta	system	SC	×	CPU_{rlx}	GPU_{rlx}
8	cta	cta	×	SC^{cta}	CPU_{rlx}	GPU_{rlx}
9	cta	gpu	×	SC^{cta}	CPU_{rlx}	GPU_{rlx}
10	cta	system	×	SC^{cta}	CPU_{rlx}	GPU_{rlx}
11	gpu	system	Acq-Rel	×	CPU_{rel}	GPU_{rlx}
12	cta	cta	×	×	CPU_{rel}	GPU_{rlx}
13	cta	gpu	SC	×	CPU_{rel}	GPU_{rlx}
14	cta	gpu	×	×	GPU_{rlx}	CPU_{rlx}
15	cta	cta	SC^{cta}	×	GPU_{rlx}	CPU_{rlx}
16	gpu	system	SC^{cta}	×	GPU_{rlx}	CPU_{rlx}
17	cta	system	×	Acq-Rel	GPU_{rlx}	CPU_{acq}
18	cta	system	×	SC	GPU_{rlx}	CPU_{acq}
19	gpu	gpu	Acq-Rel^{cta}	×	GPU_{rel}	CPU_{rlx}
20	gpu	gpu	×	SC	GPU_{rel}	CPU_{rlx}
21	gpu	gpu	×	Acq-Rel	GPU_{rel}	CPU_{acq}
22	cta	gpu	Acq-Rel^{cta}	Acq-Rel	GPU_{rel}	CPU_{acq}

the test produced weak behaviors, failing to establish critical orderings. This confirms that the hardware respects the hierarchy: `cta` and gpu scoped operations are strictly internal to the device and are not promoted to the global interconnect, validating the independence of device-level consistency domains. [2] For example, in Figure 4e the GPU producer issues a gpu-scoped `release` (m_3) preceded by a `cta`-scoped fence (m_2), while the CPU consumer issues an `acquire` (m_4). Under the CMCM, the unscoped CPU acquire is treated as system-scope, but the GPU-side release scope (gpu) does not encompass the CPU thread. The pair is therefore not morally strong, and the weak outcome is permitted.

Weak Behaviors from Insufficient Ordering [Rows 2, 6-7, 10-11, 16-18]. The CMCM requires a valid `release` on the producer to match a valid `acquire` on the consumer. The highlighted rows represent scenarios where operations were correctly System-scoped, but the ordering semantics were mismatched (e.g., a `release` paired with a `relaxed`, or narrowly-scoped fences). These configurations consistently resulted in weak behaviors. This confirms that the Grace-Hopper interconnect adheres strictly to relaxed ordering rules that require explicit software synchronization. For example, Figure 4d uses a CPU producer with a `release` store, but the GPU only does a `system`-scoped `relaxed` load, the load-store pairing is insufficient and weak behaviours are observed.

Across all 1,960 variants, weak behaviors appeared exactly where the CMCM predicts: when scope or ordering semantics were insufficient for morally strong synchronization. No correctly synchronized system-scope test exhibited weak behaviors. Notably, several patterns that were safe in CPU-only

[2] However, § 4.3 discusses an exception to the rule.

or GPU-only contexts (e.g., gpu-scoped or asymmetric release-acquire pairings) produced weak behaviors in the heterogeneous context—precisely as the CMCM requires. This validates the ARM/PTX Compound Memory Consistency Model on the Grace-Hopper Superchip across our message-passing litmus tests.

4.3 Evaluating Read-Modify-Writes

Both the Grace CPU and the Hopper GPU provide support for read-modify-write (`rmw`) instructions, such as fetch-and-add. These instructions atomically read a value from a memory location, modify it (e.g., increment), and write the updated value back to the same location. Notably, the Hopper GPU supports `rmw` operations with `cta`, gpu, and `system` scopes. Our investigation aimed to determine if concurrent `rmw`s from both the CPU and the GPU to a shared memory location would yield the correct final count. Furthermore, we aimed to identify the necessary scope for GPU-initiated `rmw`s to ensure the integrity of these atomic operations when interacting with the CPU.

To evaluate the atomicity of inter-device `rmw` operations, we designed a fetch-and-add litmus test. In this test, two threads concurrently perform a fetch-and-add operation on a shared counter variable for one billion iterations each. We considered two placement scenarios for these threads: both threads residing on the GPU (in different CTAs) and one thread on the GPU while the other runs on the CPU. For the GPU-based `rmw` operations, we systematically varied the scope, employing `cta`, gpu, and `system` scopes. Upon completion of all iterations, we verified whether the final value of the shared counter accurately reflected the total number of expected increments (two billion), thus indicating the absence of any lost updates.
Expected Behavior. First, considering the scenario where both threads reside on the GPU in different CTAs, the PTX MCM dictates that only `rmw` operations with gpu or `system` scope are guaranteed to produce the correct final count of two billion. If the `rmw` operations are `cta`-scoped, it is permissible for counts to be missed due to the limited visibility of `cta`-scoped operations across different CTAs. For the CPU-GPU case, the ARM/PTX CMCM mandates that the final count will only be guaranteed to be two billion if the `rmw` operation on the GPU is performed with `system` scope. Otherwise, if the GPU `rmw` uses a narrower scope (`cta` or gpu), it is acceptable for the final count to be less than two billion, indicating missed updates.
Results. Surprisingly, across all our fetch-and-add experiments, regardless of whether the two threads were both on the GPU (in different CTAs) or one on the GPU and the other on the CPU, and irrespective of the scope (`cta`, gpu, or `system`) used for the GPU's `rmw` operation, the final count consistently reached two billion. This outcome indicates that the microarchitectural implementation of `rmw` operations on the Grace-Hopper provides stronger atomicity guarantees than those strictly mandated by the PTX MCM and the expected behavior under the CMCM. Specifically, the scope of the `rmw` operation on the GPU did not seem to affect the correctness of the final count in our tests, suggesting that the underlying hardware enforces atomicity across different CTAs within the GPU and between the CPU and GPU, effectively disregarding the specified scope for these atomic operations. We explain this scope-agnostic behavior in § 5.5, where our coherence analysis reveals that RMW operations are executed as far atomics at the home node, bypassing the cache hierarchy entirely. We emphasize that this scope-agnosticism applies only to atomicity: the scope of an `rmw` continues to govern its memory *ordering* guarantees. For example, a `cta`-scoped `rmw` orders only within its CTA, while a `system`-scoped `rmw` establishes ordering across the entire system.

5 Validating Coherence

Fundamentally, the questions we aim to answer are: what mechanisms does the GPU employ to maintain cache consistency (and do these differ from classical coherence protocols that invalidate sharers on a write?), and through what mechanisms are the CPU caches and GPU caches kept consistent with each other? We first explain the core idea behind our "Value Propagation" (VP) litmus tests, which we designed to decode the underlying coherence mechanisms. Subsequently, we detail how we utilized these VP tests to address each of our coherence-related questions.

5.1 Value Propagation Tests

To empirically determine the underlying cache coherence mechanism of the Grace-Hopper Superchip, we introduce the *Value Propagation (VP)* test. Unlike traditional consistency litmus tests § 4, which rely on probabilistic stress-testing to discover disallowed executions, VP tests are designed as **deterministic probes**, using **value outcomes** to distinguish between two fundamental coherence paradigms:
1. Writer-Initiated Invalidation (Eager): A write by Core 1 immediately invalidates shared copies in Core 2. Subsequent reads by Core 2 miss in the L1, and fetch the new updated value from the LLC.
2. Write-Propagation with Explicit Synchronization (Lazy): A write by Core 1 does not immediately invalidate Core 2's local cache. Core 2 continues reading stale L1 data until the following sequence takes place: **first**, Core 1 performs a release operation, updating the LLC via write-through caches; **subsequently** Core 2 performs an acquire operation, causing it to invalidate its local L1; **finally**, a subsequent read by Core 2 misses in the local cache and fetches the updated value from the LLC. While demonstrated on Grace-Hopper, VP tests are platform-agnostic and applicable to any shared-memory system.
The Value Propagation Test. operates as a *philosophical dual* to standard litmus testing, specifically Message-Passing (MP): it checks for *eager* visibility in the absence of synchronization, rather than checking for *disallowed* behaviors.

Initialization: Shared variable X is cached in both Core 1 and Core 2.
Step 1 (Core 1): Write to X, then write to Y.
Step 2 (Core 2): Spin on Y until the flag is observed.
Step 3 (Core 2): Immediately read X (without an intervening fence or `acquire`).

The outcome of Core 2's read of `X` disambiguates the protocol. If Core 2 observes the updated X, the write must have propagated eagerly, invalidating Core 2's cache line before the flag Y was observed. Implying *Writer-Initiated Invalidation*. But, if Core 2 observes the old X, the write to X did not actively invalidate Core 2's private cache. The new value may be resident in the LLC, but invisible to Core 2 until a self-invalidation occurs. Implying *Write-Propagation with Explicit Synchronization*.

A potential concern is that relaxed memory models permit reordering of accesses to X and Y, which could mask the coherence behavior. However, our Section 4.1 results bound this effect: relaxed MP tests exhibited weak behaviors at ~0.2% rates. Therefore, when our VP tests observe stale values in nearly 100% of runs (as in the GPU cross-CTA case, §5.3), reordering alone cannot account for the result—the stale reads are a definitive signature of the absence of writer-initiated invalidation. Conversely, when VP tests observe updated values consistently, this is a high-probability indicator of eager invalidation, given the ~0.2% reordering rate.

5.2 Grace Coherence

Does the Grace CPU employ writer-initiated invalidations? We executed the VP test described above on two different CPU cores within the Grace processor. Our observation was that the read of X by the second core revealed the updated value written by the first core, even in the absence of explicit release-acquire synchronization. This result strongly indicates that the Grace CPU utilizes a writer-initiated invalidation coherence mechanism.

Table 5. Hopper `cta`-Scoped Read Latencies and Behavior

Ordering	Source	L1 Cached	Latency	Status
Relaxed § 5.3	L1	✓	6ns	*Stale*
Relaxed § 5.3	L2	–	145ns	Valid
Acquire § 5.3	L2	✓	145ns	Valid

5.3 Hopper Coherence

Does the Hopper GPU employ writer-initiated invalidations? We conducted a variant of the VP test. In this experiment, a GPU thread T0 wrote to two shared variables, X and Y. The write to X was performed with `cta`-scope, while the write to Y used `gpu`-scope. We then deployed two consumer GPU threads: T1 on the same CTA as producer T0, and T2 located on a different CTA. We ensured that both consumer threads, T1 and T2, **had cached the initial value of X in their local L1 caches before the producer began its writes**. Subsequently, both T1 and T2 performed a `gpu`-scoped read of Y followed by a `cta`-scoped read of X.

Our observations revealed that while thread T1, being on the same CTA as producer T0, successfully read the updated value of X, thread T2, residing on a different CTA, did not observe this updated value of X. The fact that T1 and the producer T0 share the same L1 cache makes T1's observation of the updated X value unsurprising.

However, the crucial finding is that thread T2, despite observing the initial write to Y (eventually, due to `gpu`-scope), did not see the updated value of X when it performed its `cta`-scoped read. This stale data read occured with L1 read latencies (~ 6ns) This strongly suggests that the `cta`-scoped write to X performed by T0 did not invalidate the cached copy of X in T2's local L1 cache. This lack of invalidation across different CTAs provides compelling evidence that the **Hopper GPU does not employ writer-initiated invalidations of sharers in other L1s.**

Does the Hopper GPU employ write-through L1 caches? Having established that the Hopper GPU does not utilize writer-initiated invalidations of L1 caches across different CTAs, and knowing that `cta`-scoped operations primarily interact with the L1 cache while `gpu`-scoped (or higher) operations interact with the L2 cache according to the PTX MCM, we now investigate whether `cta`-scoped writes also propagate to the L2 cache (i.e., do L1 caches write through to the L2?).

We made a small modification to the consumer thread T2 from the previous experiment: we disabled the caching of X and Y during the thread's initialization. When the updated value of Y was eventually visible to T2, the subsequent loads for X would therefore miss in the L1 cache and read from the L2 cache. In the absence of X cached in T2's L1, we observed that T2 did see the updated value of X, and the latencies for these reads were ~ 145ns, matching the L2 hit latency. Because T2 operates on a different CTA (and thus a different L1) than T0 yet observes the updated value at L2-hit latency, the `cta`-scoped write by T0 must have propagated beyond its local L1 and reached the GPU's L2 cache. **This suggests that `cta`-scoped writes on the Hopper GPU are written through to the L2 cache.**

Does the Hopper GPU self-invalidate the L1 on an acquire? We returned to the original VP test setup for Hopper. Producer thread T0 performed a `cta`-scoped write to X, followed by a `gpu`-scoped write to Y. We then modified the consumer thread T2: instead of a `gpu`-scoped read of Y, T2 now performed a `gpu`-scoped `acquire` on Y, followed by a `cta`-scoped read of X.

With this modification, we observed that T2 could see the updated value of X. Furthermore, changing the scope of the `acquire` on Y to `system` also resulted in T2 observing the updated X. However, if the scope of the `acquire` on Y was `cta`, T2 did not observe the updated value of X. **This pattern of**

results strongly suggests that an `acquire` operation on the Hopper GPU with a scope of gpu or higher triggers a self-invalidation of the local L1 cache. This invalidation would then force the subsequent `cta`-scoped read of X to fetch the updated value from the L2, which had previously been written through to the L2 cache (as established in § 5.3). This read from the L2 is substantiated by the fact that the read latencies matched the L2 hit latencies (~ 145ns).

5.4 Grace-Hopper Coherence

Do Grace CPU writes invalidate GPU caches? We employed a modified VP test. In this setup, a CPU thread acted as the producer, first writing to a shared variable X and then performing a regular (non-release) write to another variable Y. The consumer was a GPU thread, and we ensured that the initial value of X was cached in both the L1 and L2 caches of the GPU. The GPU consumer first performed a read of Y, waiting until it observed the updated value written by the CPU. Subsequently, it performed a `cta`-scoped read of X. In this scenario, we observed that the GPU consumer consistently read the original, stale value of X, indicating that the invalidation resulting from the CPU's write to X did not propagate to the GPU's L1 cache.

We also executed a variant of this test where the GPU consumer used the gpu-scope to read X instead of `cta`-scope. In this case, the GPU thread did observe the new value of X written by the CPU. This implies that the invalidation signal originating from the CPU's write to X must have successfully reached the GPU's L2 cache and invalidated the corresponding cached copy of X at that level.

These findings strongly suggest that write operations from the Grace CPU not only invalidate shared cache lines within the CPU's cache hierarchy but also extend to invalidate shared copies residing in the Hopper GPU's L2 cache (via the NVLink-C2C interconnect). However, our results indicate that these invalidations do not appear to directly propagate to the GPU's L1 caches. This is consistent with a hierarchical protocol where the NVLink-C2C interconnect maintains coherence at the L2 level, while GPU L1 caches remain invisible to the global protocol and are managed solely through self-invalidation on acquires (Section 5.3).

Do Hopper GPU writes invalidate CPU caches? In this experiment, we reversed the roles, setting a GPU thread as the producer and a CPU thread as the consumer. We ensured that the CPU consumer cached the initial value of X. In this configuration, we observed that regardless of the scope (`cta`, gpu, or `system`) used by the GPU producer for its writes to X, the CPU consumer consistently read the updated values. **This result indicates that writes originating from the Hopper GPU first propagate to the GPU's L2 (since GPU's L1s are write through, § 5.3) from where they eventually propagate to the coherent memory region shared with the CPU, leading to the invalidation of any cached copies in the CPU's cache hierarchy.** This suggests a mechanism where GPU writes, despite their initial scope, are made globally visible via the NVLink-C2C interconnect and maintain coherence with the CPU's caches.

The latency data related to these patterns of Hopper read operations is described in Table 5.

5.5 NVLink-C2C Interconnect Protocol: AMBA CHI?

We investigate the protocol governing the high-speed NVLink-C2C interconnect between Grace and Hopper, which is speculated to be based on the AMBA CHI protocol specification [10, 33].

A Hierarchical SWMR Protocol. The AMBA CHI is a classical coherence protocol designed to enforce the Single-Writer-Multiple-Reader (SWMR) invariant by invalidating sharers upon a write. Drawing upon the findings from the preceding section, we have gathered evidence indicating that: (1) writes from the GPU invalidate any sharers residing in the CPU's caches; and (2) writes from the CPU invalidate any sharers residing in the GPU's L2 cache (but not the L1). From these observations, we can reasonably infer that the CPU's Last-Level Caches (LLCs) and the GPU's L2 cache are kept coherent using a classical SWMR-based protocol across the NVLink-C2C interconnect. This SWMR protocol appears to be integrated hierarchically with the internal coherence protocols of both the CPUs and the GPUs.

Protocol Flows. We have also established that the CPU's internal coherence protocol enforces SWMR, while the GPU's internal protocol relies on self-invalidation of the L1 and write-through to the L2 on an acquire. Consequently, any write targeting the GPU's L2 cache would trigger invalidation messages to be sent across the NVLink-C2C interconnect to the CPU's LLCs, which would then propagate these invalidations to its internal caches via its own SWMR protocol. Conversely, any write to the CPU's LLC would result in invalidations being sent to any sharers within the GPU's L2 cache, but not to its L1 sharers.

The Evidence for CHI. While we have established a hierarchical SWMR protocol tying the devices together, we have not yet confirmed that it adheres to the CHI specification. One distinguishing characteristic of CHI lies in its specific protocol flows for Read-Modify-Write (RMW) operations. While RMWs can be implemented in various ways—such as acquiring exclusive access to a cache line and locking it [19, 41]—the CHI protocol uniquely specifies an in-memory approach that bypasses local caching, which the specification terms **"far atomics"** [3]. When a processor intends to perform an RMW, it first invalidates any sharers and then executes the RMW operation directly in the "home node's" memory (the physical memory it is present on) , without bringing the affected cache block into its local cache. This characteristic aligns with our earlier observation in the CPU-GPU fetch-and-add test, where the atomic counts were consistent regardless of the

GPU's RMW operation's scope. A plausible explanation for this scope-agnostic behavior is that, because CHI RMWs use far atomics to operate directly in memory, the concept of cache scope becomes irrelevant.

To further investigate this hypothesis, we conducted a ping-pong RMW experiment involving alternating atomic operations between the CPU and the GPU on a shared variable.
Methodology. The CPU sets `X=1`, and the GPU waits until it observes `X==1`, then sets `X=2`. Subsequently, the CPU would wait for `X==2` and atomically set it back to `X=1`. We measure the latency of 10000 such **cycles** (Figure 5).

1. **RMW Ping-Pong:** Each cycle uses RMW operations to utilize AMBA CHI's far atomics.
2. **Load-Store Ping-Pong:** Each cycle uses individual load and store operations. This is used as a control.

For the GPU's RMW operations in this experiment, we varied the scope across `cta`, `gpu`, and `system`.
Expected Behavior. If the interconnect uses standard coherence for RMWs, we expect the latency of the RMW Ping-Pong to be roughly the same as that of the Ld-St Ping-Pong, since both require transferring cacheline ownership. Furthermore, if RMWs are executed locally in the GPU L1, using `cta` scope should result in deadlocks when communicating with the CPU. Conversely, if the interconnect uses CHI's far atomics, the latency of the RMW Ping-Pong cycle should be significantly lower than that of the Ld-St Ping-Pong cycle, as it avoids fetching the cacheline. Since far atomics execute at the Home Node, they should be inherently global and therefore scope-agnostic. *Even `cta`-scoped RMWs should then be visible to the CPU.*
Results. Figure 5 presents the observed latencies across 10000 measurements. The results indicate that the NVLink-C2C interconnect implements RMW operations via far atomics at the home node, a distinguishing feature of the AMBA CHI protocol specification:

- RMW's are scope-agnostic, and fast, achieving a consistently low latency (~800ns) regardless of the scope used, essentially confirming that the hardware propagates RMW operations to the Home Node. This uniformity is consistent with an architectural far-atomic path rather than a scope or placement heuristic.
- Standard Load-Store coherence is slow and scope sensitive, exhibiting a significantly higher latency (~3x), reflecting the cost of transferring cachelines and handling invalidations. We also notice that the Ld-St Ping Pong test deadlocks with the `cta` scope, confirming that local scoping is respected.

These results also explain the "stronger-than-spec" behavior of the RMWs seen in § 4.3.

6 Summary of Insights

For Performance Programmers. `cta`-scoped and `gpu`-scoped read-modify-write operations exhibit system-wide atomicity across the CPU-GPU boundary, stronger than the PTX specification mandates. Performance programmers should nevertheless be careful in exploiting this behavior: it is not guaranteed by the PTX specification and may not hold on future hardware.
For Compiler Writers. Compilation of independent CPU /GPU programs is empirically safe because the Grace-Hopper Superchip adheres to the ARM/PTX Compound Memory Consistency Model—which compilers, in our view, should target exclusively. Exploiting stronger-than-spec hardware behaviors is the domain of brave performance programmers, at their own risk.
For Architects. The reverse-engineered coherence architecture indicates a hierarchical approach to heterogeneous coherence [24, 30]. The inter-node global protocol employs AMBA CHI to enforce SWMR, while the intra-node layer within the GPU uses write-through self-invalidating L1 caches. Additionally, atomic RMW operations execute at the home node via far atomics.

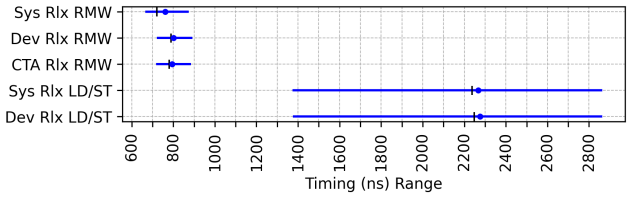

Figure 5. CPU-GPU Ping-Pong Latency in ns across 10K measurements: RMW and Ld/St (Range, Mean, Median)

7 Conclusion

Through 1,960 heterogeneous message-passing litmus tests, we have empirically validated that the Grace-Hopper Superchip enforces the ARM/PTX Compound Memory Consistency Model, placing the widespread practice of independent CPU/GPU compilation on a firm empirical foundation. This result also establishes the CMCM as a concrete architectural target for future heterogeneous systems. Using our novel Value Propagation methodology, we have provided the first empirical map of the Grace-Hopper coherence hierarchy: write-through self-invalidating L1 caches within the GPU, and CHI-like directory-based SWMR across the NVLink-C2C interconnect, with RMW operations executing as far atomics at the home node. We expect these coherence insights—particularly the write-through behavior and far atomics—will enable systems software and performance programming communities to better optimize communication-bound heterogeneous workloads.

Acknowledgments

We thank the reviewers for the helpful feedback. We thank CloudLab for the hardware infrastructure. This work is supported in part by the National Science Foundation under grant CCF-2525271 and ARM.

References

[1] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2011. Litmus: Running Tests Against Hardware. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 41–44.

[2] Johnathan Alsop, Marc S Orr, Bradford M Beckmann, and David A Wood. 2016. Lazy Release Consistency for GPUs. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–14.

[3] ARM 2021. *Atomic transactions in AMBA CHI* (01 ed.). ARM.

[4] ARM 2024. *AMBA CHI Architecture Specification* (g ed.). ARM.

[5] Fabio Banchelli, Joan Vinyals-Ylla-Catala, Josep Pocurull, Marc Clascà, Kilian Peiro, Filippo Spiga, Marta Garcia-Gasulla, and Filippo Mantovani. 2024. NVIDIA Grace Superchip Early Evaluation for HPC Applications. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region Workshops*. 45–54.

[6] Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V Adve, Vikram S Adve, Nicholas P Carter, and Ching-Tsun Chou. 2011. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 155–166.

[7] Compute Express Link Consortium, Inc. 2023. *Compute Express Link™ (CXL™)* (3.1 ed.). Compute Express Link Consortium, Inc.

[8] Miles Dai. 2021. *Reverse Engineering the Intel Cascade Lake Mesh Interconnect*. Ph. D. Dissertation. Massachusetts Institute of Technology.

[9] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2013. Everything you always wanted to know about synchronization but were afraid to ask. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, 33–48. doi:10.1145/2517349.2522714

[10] Cliff Edwards. 2022. NVIDIA Opens NVLink for Custom Silicon Integration. *NVIDIA Newsroom* (March 2022).

[11] Marco Elver and Vijay Nagarajan. 2014. TSO-CC: Consistency Directed Cache Coherence for TSO. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 165–176.

[12] Shaked Flur, Kathryn E Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 608–621.

[13] 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. *ACM SIGPLAN Notices* 52, 1 (2017), 429–442.

[14] Luigi Fusco, Mikhail Khalilov, Marcin Chrapek, Giridhar Chukkapalli, Thomas Schulthess, and Torsten Hoefler. 2024. Understanding Data Movement in Tightly Coupled Heterogeneous Systems: A Case Study with the Grace Hopper Superchip. *arXiv preprint arXiv:2408.11556* (2024).

[15] Andrés Goens, Soham Chakraborty, Susmit Sarkar, Sukarn Agarwal, Nicolai Oswald, and Vijay Nagarajan. 2023. Compound Memory Models. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1145–1168.

[16] Daniel Hackenberg, Daniel Molka, and Wolfgang E Nagel. 2009. Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on microarchitecture*. 413–422.

[17] Angus Hammond, Zongyuan Liu, Thibaut Pérami, Peter Sewell, Lars Birkedal, and Jean Pichon-Pharabod. 2024. An Axiomatic Basis for Computer Programming on the Relaxed Arm-A Architecture: The AxSL Logic. *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 604–637.

[18] Sudheendra Hangal, Durgam Vahia, Chaiyasit Manovit, and Juin-Yeu Joseph Lu. 2004. TSOtool: A Program for Verifying Memory Systems using the Memory Consistency Model. *ACM SIGARCH Computer Architecture News* 32, 2 (2004), 114.

[19] Intel®2025. *Intel®64 and IA-32 Architectures Software Developer's Manual*. Intel®.

[20] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++ 11. *ACM SIGPLAN Notices* 52, 6 (2017), 618–632.

[21] Reese Levine, Tianhao Guo, Mingun Cho, Alan Baker, Raph Levien, David Neto, Andrew Quinn, and Tyler Sorensen. 2023. MC-Mutants: Evaluating and Improving Testing for Memory Consistency Specifications. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 473–488.

[22] Themis Melissaris, Markos Markakis, Kelly Shaw, and Margaret Martonosi. 2020. PerpLE: Improving the Speed and Effectiveness of Memory Consistency Testing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 329–341.

[23] Daniel Molka, Daniel Hackenberg, Robert Schone, and Matthias S Muller. 2009. Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 261–270.

[24] Vijay Nagarajan, Daniel J Sorin, Mark D Hill, and David A Wood. 2020. *A Primer on Memory Consistency and Cache Coherence*. Springer Nature.

[25] NVIDIA Corporation 2024. *NVIDIA GH200 Grace Hopper Superchip Architecture Whitepaper*. NVIDIA Corporation. v1.21.

[26] NVIDIA Corporation 2025. *PTX: Parallel Thread Execution ISA* (8.7 ed.). NVIDIA Corporation.

[27] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2017. Simplifying ARM Concurrency: Multicopy-Atomic Axiomatic and Operational Models for ARMv8. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–29.

[28] Sabela Ramos and Torsten Hoefler. 2016. Cache Line Aware Algorithm Design for Cache-Coherent Architectures. *IEEE Trans. Parallel Distributed Syst.* 27, 10 (2016), 2824–2837. doi:10.1109/TPDS.2016.2516540

[29] Xiaowei Ren and Mieszko Lis. 2017. Efficient Sequential Consistency in GPUs via Relativistic Cache Coherence. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 625–636.

[30] Xiaowei Ren, Daniel Lustig, Evgeny Bolotin, Aamer Jaleel, Oreste Villa, and David Nellans. 2020. HMG: Extending Cache Coherence Protocols across Modern Hierarchical Multi-GPU Systems. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 582–595.

[31] Alberto Ros and Stefanos Kaxiras. 2012. Complexity-Effective Multicore Coherence. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. 241–252.

[32] Srivastava Sanya. 2024. Testing Memory Models of Heterogeneous CPU-GPU Systems. https://escholarship.org/uc/item/5hm7q1jv

[33] Gabin Schieffer, Jacob Wahlgren, Jie Ren, Jennifer Faj, and Ivy Peng. 2024. Harnessing Integrated CPU-GPU System Memory for HPC: A First Look into Grace Hopper. In *Proceedings of the 53rd International Conference on Parallel Processing*. 199–209.

[34] 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. *Commun. ACM* 53, 7 (2010), 89–97.

[35] Nikolay A Simakov, Matthew D Jones, Thomas R Furlani, Eva Siegmann, and Robert J Harrison. 2024. First Impressions of the NVIDIA Grace CPU Superchip and NVIDIA Grace Hopper Superchip for Scientific Workloads. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region Workshops*. 36–44.

[36] Ben Simner, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell. 2022. Relaxed virtual memory in Armv8-A. In *European Symposium on Programming*. Springer International Publishing Cham, 143–173.

[37] Benjamin Spector, Jordan Juravsky, Stuart Sul, Owen Dugan, Dylan Lim, Dan Fu, Simran Arora, and Christopher Ré. 2025. Look Ma, No Bubbles! Designing a Low-Latency Megakernel for Llama-1B. https://hazyresearch.stanford.edu/blog/2025-05-27-no-bubbles Hazy Research Blog.

[38] Abdulaziz Tabbakh, Xuehai Qian, and Murali Annavaram. 2018. G-TSC: Timestamp Based Coherence for GPUs. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 403–415.

[39] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovic. 2014. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-54* (2014), 4.

[40] Conrad Watt, Christopher Pulte, Anton Podkopaev, Guillaume Barbier, Stephen Dolan, Shaked Flur, Jean Pichon-Pharabod, and Shu-yu Guo. 2020. Repairing and Mechanising the JavaScript Relaxed Memory Model. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 346–361.

[41] David L. Weaver and Tom Germond. 2000. *The SPARC Architecture Manual*. SPARC International, Inc.

[42] Felix Werner, Marcel Weisgut, and Tilmann Rabl. 2025. Towards Memory Disaggregation via NVLink C2C: Benchmarking CPU-Requested GPU Memory Access. (2025).

[43] Mengdi Wu, Xinhao Cheng, Shengyu Liu, Chunan Shi, Jianan Ji, Man Kit Ao, Praveen Velliengiri, Xupeng Miao, Oded Padon, and Zhihao Jia. 2025. Mirage: a multi-level superoptimizer for tensor programs. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation (OSDI '25)*. USENIX Association, USA.

Received 2026-04-02; accepted 2026-05-04