Characterizing CPU-GPU Memory Sharing over Unified Memory Architecture

Jinrong Yang, Zimeng Wang, Rong Chen*, Haibo Chen
Institute of Parallel and Distributed Systems, School of Computer Science, Shanghai Jiao Tong University

Abstract

Unified Memory Architecture (UMA) enables the CPU and GPU to share a single physical memory. However, its complexity and opacity often drive applications back to inefficient copying or to incorrect direct access that sacrifices performance or even correctness. This paper presents a comprehensive study of CPU-GPU memory sharing on UMA, covering correctness and performance across data-plane transmission and control-plane coordination on four platforms. It yields four findings that guide `libumsh`, a portable library that delivers correct and efficient memory sharing across platforms and extends to emerging ones (e.g., NVIDIA DGX Spark). Applied to nine real-world systems, `libumsh` reduces data-plane time by 16–75% and control-plane overhead by 7–90%.

CCS Concepts: • **Computer systems organization** → **Heterogeneous (hybrid) systems**; • **Software and its engineering** → **Memory management**.

Keywords: Unified memory architecture, CPU-GPU memory sharing, Heterogeneous computing

ACM Reference Format:
Jinrong Yang, Zimeng Wang, Rong Chen, and Haibo Chen. 2027. Characterizing CPU-GPU Memory Sharing over Unified Memory Architecture. In *22nd European Conference on Computer Systems (EuroSys '27), April 19–23, 2027, Rabat, Morocco.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3842654.3848531

1 Introduction

Heterogeneous SoCs that couple a CPU with one or more accelerators—GPUs, NPUs, or ASICs—have become the dominant substrate for model inference, graphics rendering, scientific computing, and generative AI. Across edge and client devices, including embedded systems [8, 38, 82, 102], mobile phones [9, 76, 101], laptops [11, 45, 100], and emerging mini AI supercomputers [10, 15, 111], vendors increasingly integrate these accelerators with the CPU under a Unified Memory Architecture (UMA), in which both processors share a single physical memory. In contrast to the traditional Discrete Memory Architecture, which keeps CPU and GPU memory physically separate and connects them over PCIe or NVLink C2C [81], UMA offers a simpler programming model, a larger memory footprint available to each processor, and the elimination of explicit cross-device copies.

This paper studies CPU-GPU communication via memory sharing on UMA platforms—a basic building block that determines both the correctness and the performance of heterogeneous computing. Such communication serves two qualitatively distinct purposes. Data-plane transmission moves bulk data into or out of GPU computation, including model parameters, inference inputs/outputs, textures, and point clouds, typically at task boundaries; representative scenarios include model loading [31, 49, 98], image preprocessing [90, 119], output sampling [91, 122], and frame composition [59, 115]. Control-plane coordination exchanges small but frequent state—flags, counters, and locks—within and across kernels to sequence work or arbitrate shared resources between the CPU and GPU, as exemplified by hybrid inference [20, 33, 126, 127], model offloading [19, 39, 66], and GPU filesystem operations [67, 99, 110].

Although UMA platforms natively support *direct access* to shared memory, mainstream applications routinely fail to exploit this capability correctly or efficiently. In practice, most applications either treat the platform as if it were discrete and incur unnecessary copy overhead [25, 31, 49, 83, 89, 98, 114, 120], or apply UMA features incorrectly and suffer data inconsistency or crashes [60, 61, 87, 88]. For example, widely used deep learning frameworks such as PyTorch [98] and OpenVINO [49], as well as LLM serving systems such as vLLM [120] and llama.cpp [31], still rely on redundant copies for weight loading and I/O processing on UMA platforms, wasting both time and memory. Rendering engines such as MoltenVK [115] exhibit incorrect usage patterns that cause data inconsistency between the CPU and GPU.

These failures are not matters of engineering preference; they arise from the inherent complexity and opacity of UMA platforms along three aspects. (i) *Hardware*: platforms differ in interconnect type (Ring vs. NoC), in the presence of memory-side cache (MSC) in front of DRAM, and in support for GPU-to-CPU coherence, and each combination alters which access patterns are correct and which are efficient. (ii) *Software*: critical configurations such as page size, cache

*Rong Chen is the corresponding author (rongchen@sjtu.edu.cn).

This work is licensed under a Creative Commons Attribution 4.0 International License.
EuroSys '27, Rabat, Morocco
© 2027 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2971-3/2027/04
https://doi.org/10.1145/3842654.3848531

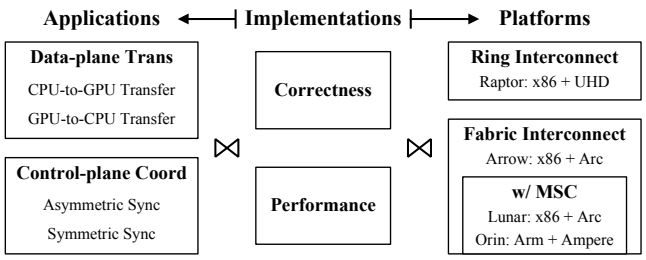

Figure 1. *Our analysis methodology.*

policy, and coherence direction are buried within closed-source drivers and proprietary runtimes, while the OS abstractions above them silently fall back under load [65, 94]. (iii) *Semantics*: data-plane and control-plane traffic impose opposing objectives—bandwidth on large transfers versus low-contention atomics on small flags—so no single memory-sharing recipe is universally optimal.

In response, we present the most comprehensive study of CPU-GPU memory sharing over UMA to date. We pursue completeness along three dimensions (Figure 1) that mirror the source of complexity above: applications (semantics), implementations (software), and platforms (hardware).

- **Applications:** We classify the purposes of CPU-GPU memory sharing into two planes: *data-plane transmission*, which moves data for GPU computation and covers both *CPU-to-GPU* and *GPU-to-CPU* transfer; and *control-plane coordination*, which exchanges state information between the CPU and GPU and covers both *asymmetric* and *symmetric* synchronization.
- **Implementations:** We decompose every sharing operation into an independently configurable *sending* and *receiving* stage, each tunable along three attributes (page size, cache policy, coherence) and realizable as Write-Alone, Write-then-Flush, Write-Bypass, or their read counterparts. This allows us to enumerate every correct combination per platform and identify the fastest for the data and control planes.
- **Platforms:** We evaluate four x86/ARM platforms with Intel/NVIDIA GPUs, representing UMA designs used from cost-sensitive devices to high-performance systems. They contrast Ring and Fabric NoC interconnects, memory-side caches (MSC), and one-way cache coherence (CC). Spark extends the study to two-way CC (§5.2).

Findings. Our analysis covers what we believe are the most representative CPU-GPU memory sharing applications and UMA platforms in use today, yielding correct implementations for each platform and four performance findings (§4).

1. *Page size dominates setup cost and propagates into access cost.* For large regions, 4 KB pages inflate setup time by nearly 100× over 2 MB pages and degrade subsequent bandwidth through TLB pressure. For small regions, forcing 2 MB allocations wastes memory and is up to 5× slower on Lunar and Orin. Transparent Huge Pages (THP) [71] selects an appropriate size automatically; we therefore recommend enabling THP and prefaulting the region at setup.

2. *Uncached memory is simple and platform-agnostic, but its cost is substantial.* Uncached (UC) memory makes direct access correct without code changes—as with Orin's default GPU policy for registered host-memory sharing [79]—yet it inflates single-threaded CPU latency by up to 300×. GPU parallelism can hide UC overhead in microbenchmarks, but real workloads still incur a measurable penalty: an L2-optimized GEMM on Orin loses 16% of its GFLOPS. UC should therefore be reserved for cases in which neither code modification nor coherence support is feasible.

3. *One-way cache coherence simplifies implementation at near-zero cost on MSC-enabled platforms.* GPU-to-CPU cache coherence (CC), supported on most UMA platforms, allows the CPU to correctly access write-back (WB) memory without any application-level changes. CC introduces overhead on GPU accesses, but this cost is negligible when a memory-side cache (MSC) is present. On Arrow, which lacks an MSC, GPU write time with CC is 2.07× that without CC.

4. *Under high concurrency, control-plane coordination is bottlenecked by contention rather than access latency.* For small shared regions, the choice among correct implementations is nearly irrelevant; what matters is the number of GPU threads that touch the flag concurrently. Block-level barriers handle up to 1,000 threads with minimal code complexity. Beyond this threshold, grid-level barriers eliminate contention at the cost of deadlock-avoidance complexity, and a single-thread kernel remains a safe fallback.

Library and extensibility. `libumsh` automatically selects correct, efficient implementations through high-level sender-receiver operations, based on the platform and application sharing requirements (Table 2, §5). An AI-assisted three-step porting workflow (§5.1) extends it to NVIDIA DGX Spark [111], validating extensibility to two-way coherence.

Case study. We applied `libumsh` to nine real-world systems that span data-plane and control-plane patterns across the four target UMA platforms. On the data plane, `libumsh` eliminates redundant copies in model loading (PyTorch), image preprocessing (OpenCV), output sampling (OpenVINO GenAI), and point-cloud processing, reducing transmission time by 16–75%. On the control plane, `libumsh` replaces heavyweight runtime events and CPU-side delegation with lightweight memory-based signaling and atomics in hybrid inference (EdgeNN), model offloading (Accelerate, Demand Layering), GPU filesystem operations (GPUfs), and cross-device file locking (GoFS), reducing coordination overhead by 7–90%.

Contributions. This paper makes three contributions:

- We develop a systematic methodology spanning application semantics, memory-sharing implementations, and UMA properties (Figure 1).
- We characterize four UMA platforms and reveal how page size, cache policy, coherence, MSC, and contention jointly shape data-plane and control-plane performance.
- We build `libumsh`, to our knowledge the first library whose high-level CPU–GPU sharing APIs automatically select correct, efficient implementations by platform and application sharing requirements. The library, correctness tests, and benchmarks are available at https://github.com/xpuos/libumsh.

2 Background and Context

As heterogeneous computing continues to evolve and the demand for high-performance applications increases, various accelerators such as GPUs and NPUs are increasingly being deployed alongside CPUs. This paper focuses on the most representative CPU-GPU communication mechanisms.

2.1 Unified Memory Architecture

Unified memory architecture (UMA) enables the CPU and GPU to share a unified physical memory [107]. UMA is widely deployed in edge devices, including embedded systems [8, 38, 82, 102], mobile phones [9, 76, 101], laptops [11, 45, 100], and emerging mini AI supercomputers [10, 15, 111], due to its simplified programming abstraction and increased available memory. In contrast, discrete memory architecture, the traditional mainstream approach, uses physically separate CPU and GPU memory spaces connected by buses such as PCIe or NVLink C2C [81]. Figure 2 illustrates these two memory design approaches. With discrete memory, CPU-GPU communication requires explicit data copying (I/O operations) [16, 69]; in UMA, all processors can directly access the same data without additional data movement. However, this approach requires a deep understanding of UMA hardware features and careful coordination between CPU and GPU to ensure both correctness and optimal performance.

UMA hardware features. UMA is deeply integrated into SoC internals, so interconnect designs vary across vendors and even chip models (Figure 2).

CPU-GPU Interconnection. The studied UMA platforms use a Ring or network-on-chip (NoC) fabric interconnect. For Ring interconnects, CPU cores and the GPU share a distributed last-level cache (LLC) accessed over the ring [44, 92]. This requires minimal SoC design changes but scales poorly. NoC connects them at the cluster level, scaling better to additional components [3]. In our NoC platforms, CPU caches are not directly accessible to the GPU as a shared cache tier. Some NoC platforms introduce a shared system-level cache above main memory (memory side cache, MSC) [17, 22]. These cache-sharing and coherence properties depend on platform design, rather than interconnect topology alone.

Data Coherence. Cache coherence (CC) between the CPU and GPU provides a consistent shared-memory view and simplifies UMA programming. On Ring-interconnect platforms, the GPU, CPU cores, and the LLC all function as agents on a shared ring bus [42]. This enables them to participate in a unified cache coherence protocol that naturally extends the CPU's own inter-core coherence mechanism, such as MESIF, although GPU private caches generally remain non-coherent with the CPU [30]. This coherence is managed automatically and is not exposed to the driver for configuration [70].

The studied NoC platforms use I/O coherency [17, 74], allowing the GPU to issue coherency transactions to the CPU subsystem. Coherent GPU reads can retrieve the latest data from CPU caches; coherent GPU writes invalidate stale CPU cache copies as required [12, 79]. All four target platforms provide only *one-way coherence*: GPU-initiated accesses can maintain consistency with CPU caches, but the CPU cannot reciprocally snoop or invalidate GPU caches. The cost of GPU-to-CPU coherence depends on the coherence implementation and cache organization. Raptor uses the ring's shared-LLC coherence protocol, while the studied NoC platforms use I/O-coherency transactions with the CPU subsystem that can add memory-access overhead (§4.2).

2.2 CPU-GPU Communication

CPU-GPU communication is a key building block for heterogeneous computing. It spans two purposes: *data-plane transmission* and *control-plane coordination.*[1]

Data-plane transmission. This covers producer-consumer transfers between CPU and GPU, including model parameters, inputs, and outputs. For example, during model inference, data is transferred to and from the GPU at the start and end of task execution. Similar transfers occur in GPU rendering, 3D image processing, scientific computing, and autonomous driving. Data-plane transmission has two types: *CPU-to-GPU* and *GPU-to-CPU* transfer. The volume of transferred data varies considerably—ranging from a few integers for simple inputs to gigabyte-scale model weights. The communication pattern is characterized by low frequency, as data transfers generally occur only at GPU task boundaries.

Control-plane coordination. This involves the exchange of control information between the CPU and GPU, including synchronization signals and status updates in irregular or CPU–GPU collaborative workloads. For example, in model offloading scenarios, the CPU iteratively prepares data for the GPU—such as loading per-layer parameters or a KV Cache—before launching the corresponding kernel. This unidirectional synchronization pattern is referred to as *asymmetric synchronization.* It can be effectively managed through a wait-based mechanism for task dependencies. Furthermore,

[1] Our communication microbenchmarks measure CPU–GPU memory communication without modeling application-level caching or reuse, such as hot-data locality within a matrix-multiplication kernel.

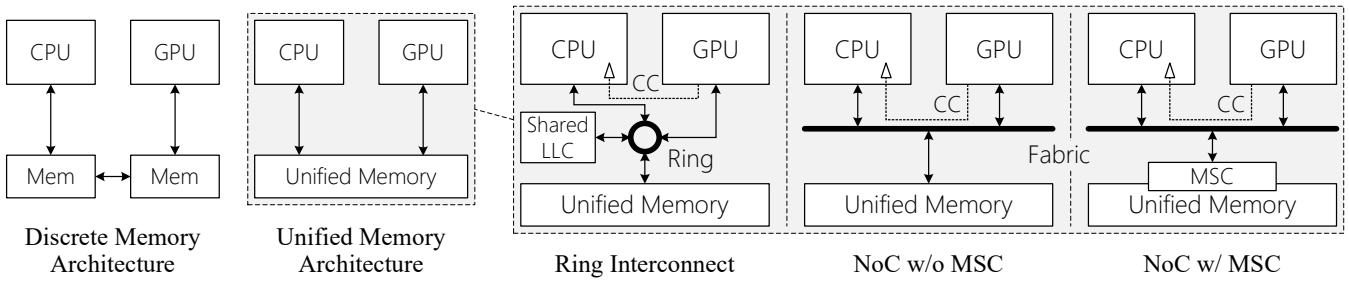

Figure 2. *Discrete memory architecture, unified memory architecture (UMA), and UMA with different hardware features. All interconnect panels are logical views; e.g., the shared LLC on the ring is physically sliced across ring stops co-located with cores.*

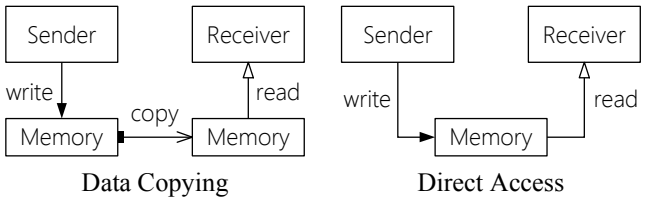

Figure 3. *Two approaches for CPU-GPU communication.*

some heterogeneous workloads (e.g., GPU filesystem) require more complex coordination like cross-device contention control, where both the CPU and GPU must wait for each other at specific execution points to ensure correct operation sequencing. This requires advanced synchronization primitives (barriers or locks) to enable CPU-GPU coordination. This pattern is referred to as *symmetric synchronization*. To maintain tight coordination between the CPU and GPU during task execution, control-plane coordination involves small payloads (typically just control messages or flags) yet occurs much more frequently within GPU tasks.

2.3 Memory Sharing

CPU-GPU communication can be modeled as one processor (the *sender*) transmitting a unit of data to another processor (the *receiver*), where either the CPU or GPU can act as the sender or receiver. Memory sharing between them enables CPU-GPU communication. Two primary approaches exist for memory sharing: *data copying* and *direct access*.

Data copying. Analogous to *pass-by-value* in programming, this approach implements memory sharing through explicit copy operations between two individual memory regions. As shown in Figure 3, the sender first writes data to a designated memory region, and then the receiver copies this data into its own memory region and reads it. This approach imposes minimal demands on underlying hardware capabilities, making it broadly applicable to both discrete and unified memory architectures. However, this approach incurs overhead from additional memory copy operations and increases memory footprint by maintaining duplicate data copies.

Direct access. This approach, akin to *pass-by-reference*, utilizes a physically shared memory region accessible to both sender and receiver, as shown in Figure 3. It eliminates copy operations and extra memory consumption. However, it requires direct CPU and GPU access to the same physical memory region, a capability provided by UMA platforms.

In the wild. Despite native support for direct access on UMA platforms, many existing systems fail to fully leverage this capability. Our analysis of popular open-source LLM inference frameworks [95], including llama.cpp [31], Ollama [89], vLLM [120], and DeepSpeed [25], as well as general-purpose model inference frameworks, including PyTorch [98], TensorFlow [114], TensorRT [83], and OpenVINO [49], reveals that these frameworks predominantly rely on data copying for weight loading and input/output processing.

GPU runtimes provide APIs for direct access, and developers of rendering and computation frameworks use them to optimize performance on UMA platforms. However, direct access is easy to implement incorrectly: missing cache flushes and memory barriers often result in data inconsistency between CPU and GPU, or application crashes [60, 61, 87, 88]. Even with correct implementation, suboptimal approaches can greatly impact performance. For example, inadequate understanding of memory access characteristics may cause direct access to underperform data copying [86]. The model loading optimization for Apple UMA platforms in llama.cpp does not effectively bypass the copy operation. Similarly, improper synchronization granularity often introduces additional overhead in control-plane coordination [35, 67].

Root causes and challenges. These problems stem from two factors. First, these systems were initially designed for discrete memory architectures. To reduce development effort, existing data copying implementations are directly adapted for UMA platforms. Second, substantial variations in hardware architecture, software interfaces, and performance across UMA platforms [108] mean that developers often lack the experience needed to efficiently implement memory sharing via direct access [14, 36, 74]. Achieving correctness and optimal performance on UMA platforms requires precise control over data visibility and consistency—a significant challenge in CPU-GPU heterogeneous environments.

3 Target Platforms

Our core study covers four platforms with one-way CC. The x86 Intel subset—Raptor, Arrow, and Lunar—spans Ring, NoC without MSC, and NoC with MSC, enabling comparisons within one driver/runtime ecosystem. Orin adds an ARM-based NVIDIA platform with NoC and MSC.

3.1 Ring Interconnect: Raptor

The Raptor platform is built around an Intel Core i5-14500 (Raptor Lake) processor with a Ring interconnect linking the CPU cores, GPU, and memory controller. Its 24 MB LLC is distributed across ring-connected slices and shared by CPU cores and the GPU [44]. The system has dual-channel DDR4-3200 memory and an integrated Intel UHD 770 GPU. Both the GPU and memory controller are integrated as agents on the ring bus. Memory requests leaving the GPU traverse the ring nodes hosting the LLC's coherence agents, enabling coherent access to the shared LLC and memory. However, this coherence is one-way, as the GPU's private caches remain non-coherent with the CPU [30]. The ring organization remains relevant in low-power, cost-sensitive products such as Intel Processor N-series (e.g., N150 and N250) [46, 51].

3.2 NoC Interconnect without MSC: Arrow

The Arrow platform uses an Intel Core Ultra 285H (Arrow Lake) processor with a scalable fabric NoC interconnect. The CPU, integrated Intel Arc 140T GPU, and other agents connect at the cluster level. This architecture offers better scalability. The CPU's LLC is shared within the compute tile and is not directly accessible to the GPU as a shared cache tier [47]. The system includes dual-channel DDR5-5600 memory. Coherence transactions across the fabric maintain data consistency. This is also one-way: GPU-initiated accesses can maintain consistency with CPU caches, but not vice versa. The absence of an MSC makes this platform essential for evaluating how a system-level shared cache impacts performance. The NoC-without-MSC organization also appears in newer embedded devices, such as AMD Versal Prime Series Gen 2 models 2VM3104 and 2VM3654 [1].

3.3 NoC Interconnect with MSC: Lunar and Orin

Lunar and Orin platforms feature a NoC interconnect and include an MSC, with one-way coherence. Apple SoCs [22, 125] and AMD APUs [121] also adopt this organization, but some coherence differences will be discussed in §7.

Lunar. The Lunar platform, powered by an Intel Core Ultra 258V (Lunar Lake) processor, targets energy-efficient mobile computing with graphics and AI acceleration. The system includes an Intel Arc 140V GPU and LPDDR5-8533 memory. Its key feature is an 8 MB MSC—a cache logically located between the processors and main memory [17]. The MSC reduces main memory access delays.

Orin. The Orin platform is a Jetson AGX Orin embedded system, built on an ARM Cortex A78AE CPU with an integrated NVIDIA Ampere GPU, and demonstrates these architectural trends in the ARM ecosystem. It also uses a scalable NoC interconnect and includes a 4 MB MSC to improve memory performance. By including it, we conduct a broader analysis of UMA behaviors beyond x86 architecture, providing a more comprehensive understanding of the underlying principles.

4 Communication via Memory Sharing

Analyzing the correctness and performance of CPU-GPU communication over UMA presents significant challenges due to two key factors. First, it requires cooperation between heterogeneous processors (CPU and GPU) with distinct hardware characteristics, across UMA platforms that vary in their interconnects and configurations. Second, CPU-GPU communication serves different purposes, such as data-plane transmission and control-plane coordination, each with unique behavioral patterns and requirements. To facilitate analysis, we decompose the execution flow of CPU-GPU communication via memory sharing into two stages: *sending* and *receiving*, with unified memory serving as the boundary. These stages can be implemented independently and combined flexibly, enabling systematic analysis of correctness and performance.

4.1 Correctness of Memory Sharing

Communication via memory sharing involves four operations:

- **Bind** establishes a shared memory region and configures its attributes separately for the sender and receiver.
- **Write** populates the memory with data from the sender.
- **Read** retrieves data from the memory for the receiver.
- **Unbind** revokes access permissions and frees memory.

Methodology. We model memory sharing as two stages: sending (Bind-Write-Unbind) and receiving (Bind-Read-Unbind). Each stage involves a single processor (CPU or GPU) and can be configured and implemented independently (e.g., memory attributes). We empirically test candidate sending and receiving combinations in both CPU–GPU directions under selected memory configurations, keeping control state separate from the tested data. A single-thread persistent GPU kernel participates in each test. After reading the previous payload, the receiver checks every word of a new deterministic payload written by the sender. The verdict is fixed before kernel exit. An exact match records a successful transfer; stable failure requires reproducible mismatches after bounded retries. Timeouts and infrastructure errors are inconclusive, while unsupported configurations are skipped.

These tests evaluate data visibility between a single sender and receiver. Write-write contention is outside these tests; atomic synchronization and contended locks are examined separately (§4.1.4). We assume that the driver and hardware apply the requested page size, cache policy, and coherence settings. The results provide empirical evidence for the tested platforms, configurations, and sharing model, without formal

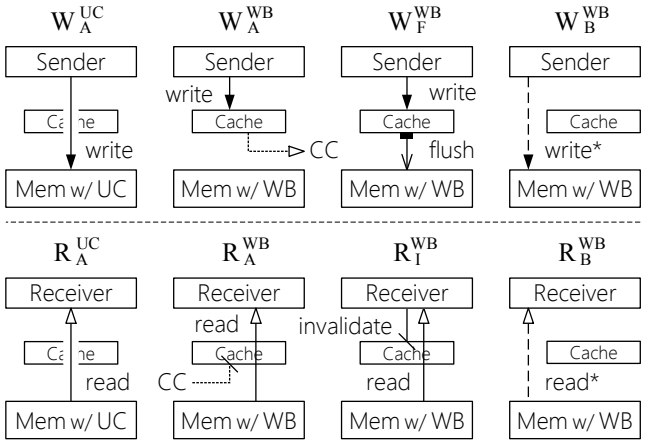

Figure 4. *Different implementations of Write and Read.*

guarantees for general CPU–GPU shared-memory interactions. The testing tool is included in the `libumsh` artifact.

4.1.1 Bind and Unbind

Bind allocates a shared memory region from unified physical memory and configures its memory attributes by setting page tables of both the CPU and GPU, which may act as the sender and receiver. Three critical attributes collectively determine sharing semantics and performance characteristics.

Page Size. Both CPU and GPU support multiple page sizes. The CPU typically uses a small page size (4 KB) by default. In contrast, the GPU's page size depends on the kernel-space driver and user-space runtime implementation. While most GPUs support common page sizes, they typically prefer a large page size (2 MB) for better streaming memory access performance [29, 55]. When CPU and GPU share memory, the shared region must satisfy the page-alignment requirements of both the CPU and GPU [43].

Cache Policy. Write-back (WB) and uncached (UC) are two primary cache policies for memory access. UC ensures direct access to shared memory, while WB access data through caches. The page table entry (PTE) controls this attribute, affecting only memory accesses initiated by the associated processor. For the CPU, the kernel offers a unified interface to control the cache policy [93]. For the GPU, configuring the cache policy requires case-by-case analysis of the driver's page table configuration code—no general method is available [103, 104]. GPU accesses on Raptor, Arrow, and Lunar use WB in our evaluation. On Orin, we use registered host memory, excluding CUDA managed memory. Registered host memory is uncached in GPU L2 by default [79][2].

Data Coherence. Enabling cache coherence (CC) ensures that read operations always retrieve the latest version of data and write operations always invalidate stale versions. This simplifies implementation on the opposite side and may enhance performance for memory sharing. On our target platforms, only the GPU supports controlling CC and enables it by default for memory pages. The kernel configures this feature by modifying PTE bits. Intel's user-space runtimes indirectly expose configurable options that allow applications to specify CC strategies. Some Intel platforms (e.g., Arrow and Lunar) even allow applications to manipulate the relevant PTE bits, providing fine-grained control.

Page size selection affects only performance (see §4.2.1 for details). In contrast, cache policy and data coherence determine the correctness of memory sharing—specifically, which write and read implementations are needed to ensure data is properly written to and read from memory.

Unbind revokes access permissions for both sender and receiver to the shared memory region and releases associated resources. This process reverses the actions performed during Bind. Note that unbinding the same memory region twice will cause an application crash.

[2]On Orin, registered host memory is uncached in GPU L2 but can be cached in GPU L1. We bypass GPU L1 for shared-memory loads using the compiler option "-Xptxas -dlcm=cg".

Table 1. *Correct combinations of Write/Read and memory attributes on different platforms for CPU-GPU memory sharing.*

Bind	Write	Platform
UC	Write-Alone	CPU (All) GPU (Orin)
WB	Write-Alone	CPU (with GPU CC)
	Write-then-Flush	CPU (All) GPU (All)
	Write-Bypass	CPU (DMA) GPU (Arrow, Lunar)

Bind	Read	Platform
UC	Read-Alone	CPU (All) GPU (Orin)
WB	Read-Alone	CPU (with GPU CC)
	Read-after-Inval	CPU (All) GPU (All)
	Read-Bypass	CPU (DMA) GPU (Arrow, Lunar)

4.1.2 Write

To ensure data is properly written to memory and visible to the receiver, three Write implementations are available. Figure 4 illustrates these implementations, and Table 1 lists the correct combinations of them and memory attributes.

Write-Alone ($\mathtt{W}_\mathtt{A}$). A write instruction (e.g., `mov` or `store`) can correctly write data to memory only when configured as UC. However, if the receiver has CC enabled, Write-Alone also functions correctly with memory configured as WB.

Write-then-Flush ($\mathtt{W}_\mathtt{F}$). After writes, flushing instructions ensures data is written back to shared memory, even with WB caching. Two approaches can evict cached data: 1) device-specific flush instructions, and 2) external flush commands. Flush instructions are widely supported on CPUs (e.g., `cl flushopt` on x86 and `dc` on ARM), although GPU support varies across platforms. Arrow and Lunar provide `lsc_fence` [41], while Raptor and Orin do not include such instructions in official documents or compilers. On our target

platforms, the CPU driver can issue external flush commands to evict cached data to memory for the GPU via MMIO, while the reverse is not supported. On all three Intel GPUs, the flush command is enqueued as a task in the hardware queue and processed in order. Upon completion of the current kernel, data written by that kernel is flushed from the GPU's internal cache to memory. In contrast, on Orin, a flush command is triggered by writing to a special register (BAR), which suggests this process operates independently of kernel execution.

Write-Bypass (W_B). Certain specialized instructions enable direct memory writes that bypass the private cache hierarchy, ensuring correct operation on memory configured as WB. For CPUs, non-temporal store instructions (e.g., `movntdq`) write directly to memory and reduce cache pollution. However, these instructions serve as hints rather than strict guarantees. Furthermore, I/O devices (e.g., SSD) can bypass the cache and write to memory directly via DMA. For GPUs, store instructions can incorporate cache-bypass modifiers to implement Write-Bypass behavior. Arrow and Lunar support write instructions that bypass the L2 cache, whereas such support was not found on the Raptor and Orin platforms.

4.1.3 Read

To ensure data from the sender is properly read, three Read implementations are available to retrieve data from memory. Figure 4 illustrates these implementations, and Table 1 lists the correct combinations of them and memory attributes.

Read-Alone (R_A). A read instruction (e.g., `mov` or `load`) can correctly read data from memory only when configured as UC. However, if the sender has CC enabled, Read-Alone also functions correctly with memory configured as WB.

Read-after-Inval (R_I). This implementation first invalidates the private cache, then performs read instructions on memory configured as WB. Similar to Write-then-Flush, two approaches can invalidate cached data: (1) device-specific invalidation instructions, and (2) external invalidation commands. Due to data integrity concerns, most platforms do not provide dedicated invalidation instructions (or commands). Instead, they rely on flush instructions (or commands) to implement invalidation, even though no actual writeback occurs. Among our target platforms, only the ARM CPU provides the `dc ivac` instruction, which invalidates cache lines within a specified region without writeback.

Read-Bypass (R_B). This implementation mirrors Write-Bypass, its counterpart for direct memory writes. I/O devices use DMA to bypass the cache and read memory directly, while GPUs use load instructions with cache-bypass modifiers.

4.1.4 Acquire and Release

To enable sophisticated CPU-GPU coordination (e.g., locks and barriers), control-plane symmetric synchronization relies on two atomic operations built on memory sharing:

- **Acquire** reads the current state to check availability and conditionally writes to establish ownership.
- **Release** writes to the state to relinquish ownership, making it available once more.

Traditional cross-device implementations of these operations rely on a *delegation* mechanism [67], where the CPU serves as an arbiter to handle acquisition and release requests from the GPU, or vice versa. For PCIe-connected discrete GPUs, delegation is useful when host memory lacks CPU–GPU read-modify-write (RMW) atomicity. System-scope RMW atomicity also depends on platform capabilities [84]. In our tests on NVIDIA discrete GPUs, `atomicAdd_system` executed but lost updates under concurrent CPU increments on mapped host memory. It produced exact counts without concurrent CPU writers, or with them on managed memory.

UMA platforms provide opportunities to optimize symmetric synchronization by enabling direct contention on shared memory instead of relying on delegation to establish a global serialization point. Through the unified memory hierarchy, both Acquire and Release can be implemented using atomic instructions (e.g., CAS) directly on shared memory, eliminating the overhead and complexity associated with delegation. Our target platforms support such CPU–GPU atomic operations on the shared memory used by our lock implementations, with CPU mappings configured as WB.

4.2 Performance of Memory Sharing

4.2.1 Memory Binding

Memory binding time represents substantial overhead (about 50%) in scenarios like model loading, where each CPU-GPU communication requires one memory binding. Binding time consists of three parts: memory allocation and page table configuration for both the CPU and GPU. Memory allocation time depends heavily on memory utilization, while the page table configuration process and performance characteristics remain consistent across the CPU and GPU. Therefore, we focus on examining page table configuration time for the GPU. As shown in Figure 5, page size primarily influences memory binding performance and varies across UMA platforms. Cache policy and data coherence settings have negligible impact since these configurations are applied per PTE.

For large memory regions exceeding 32 KB, the default 4 KB page size incurs significant time overhead, dominated by PTE population. Hugepages (2 MB) show substantial advantages. For small memory regions fitting within a single physical page, performance varies across platforms. The Intel drivers on Raptor and Arrow show comparable times for 4 KB and 2 MB pages, while 4 KB pages significantly outperform 2 MB pages on Lunar and Orin (reaching about 5× on Orin). This happens because the driver scans the input memory range before modifying page table entries to determine page size and PTE count. Registering large pages forces the

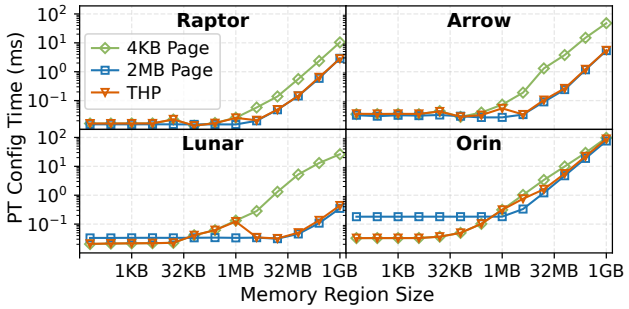

Figure 5. *A comparison of page table configuration time for the GPU using different page sizes on our target platforms.*

driver to process a full 2 MB-aligned region even for small memory regions, introducing unnecessary overhead.

Since 4 KB pages can incur nearly 100× higher bind overhead and degrade access performance due to TLB pressure [7, 52, 129], GPU runtimes typically enforce a 2 MB minimum allocation. However, this granularity wastes memory for small control-plane messages. Transparent Huge Pages (THP) [71] minimizes memory waste while automatically promoting memory to hugepages before page size becomes a performance bottleneck for memory access. As shown in Figure 5, THP provides optimal performance in most scenarios.

Implications. Page size significantly impacts both setup latency and access performance. While 2 MB allocation optimizes streaming patterns, it proves inefficient for small regions. We recommend enabling THP and initializing the entire region via `memset` to fully populate memory upfront. This approach balances these benefits against THP drawbacks such as compaction latency spikes [75, 94] (Finding 1).

4.2.2 From CPU to GPU (CPU⇒GPU)

Communication from CPU to GPU over UMA involves a CPU-side sending stage (CPU Write) followed by a GPU-side receiving stage (GPU Read) on a shared memory region. To achieve optimal performance, we select the most efficient implementation for each stage and combine them.

CPU Write. Two primary Write implementations are available: Write-Alone (W_A) and Write-then-Flush (W_F). W_A is applicable when memory is configured as UC or as WB with GPU-side CC enabled. As shown in Figure 6, using UC memory (W_A/UC) forces each write to bypass the cache, increasing write time by up to 300×. Although hardware write buffers provide partial write asynchronicity [27], their capacity is limited. In contrast, WB memory (W_A/WB) leverages the CPU private cache to enable write combining, achieving the lowest write time across all four platforms.

W_F performance depends on the overhead of the mandatory flush operation. On the ARM-based Orin, which is limited to fine-grained flushing W_F/Fine (`dc civac`), performance matches W_A/WB for regions under 16 MB. Beyond this size, flush overhead dominates, increasing latency by up to 50%. In contrast, x86 platforms support both user-space fine-grained (`clflushopt`) and privileged global flushing (`wbinvd`). W_F/Coarse is inefficient for small memory regions due to a high fixed overhead (~2 ms) from kernel transitions and global flushing. Conversely, W_F/Fine is effective up to 256 MB. Beyond this threshold, the linear cost of per-line flushing causes approximately 20% slowdown. For sufficiently large transfers, the fixed overhead of W_F/Coarse is amortized, allowing performance to approach W_A/WB.

We further study Write-Bypass (W_B) performance when data originates from an SSD on Arrow (PCIe 5.0) and Orin (PCIe 4.0). As shown in Figure 7, compared to the mmap-and-copy approach, our W_B implementation eliminates overheads of redundant copying, page cache management, and coherence maintenance. In mmap-and-copy, mmap stages data in CPU memory before an explicit CPU-to-GPU copy, leaving its zero-copy and on-demand semantics orthogonal to the SSD-to-GPU transfer evaluated here and safe to bypass with our DMA path. This reduces time by 70% on average and up to 98% for 512-byte transfers on Arrow.

GPU Read. GPU-side implementation performance varies significantly across platforms. Therefore, we discuss three groups: Intel UHD GPUs (Raptor), Intel Arc GPUs (Arrow and Lunar), and NVIDIA GPUs (Orin).

Intel UHD GPUs. Raptor lacks `LSC`-like instructions for memory and cache control. It only supports driver-initiated Read-after-Inval (R_I/Dri), which executes at inter-kernel boundaries to perform invalidation. Raptor cannot perform intra-kernel invalidation, preventing control-plane coordination for applications. Furthermore, as shown in Figure 6, enabling CC on Raptor has no discernible impact on GPU Read performance (R_I/Dri+CC vs. R_I/Dri). We hypothesize that GPU requests traverse the LLC coherence agents on Raptor's ring and use its coherence protocol to access the shared LLC.

Intel Arc GPUs. Arrow and Lunar feature high-performance Intel Arc GPUs [17, 50]. They support fine-grained control over the caching of read/write instructions and explicit cache management [48]. This offers the most flexible hardware for memory performance evaluation, supporting Read-after-Inval with both driver-triggered (R_I/Dri) and instruction-triggered (R_I/Inst), as well as Read-Bypass (R_B).

R_I/Dri delivers optimal performance in most cases, although its use is restricted to inter-kernel boundaries. Enabling CC introduces a marginal read overhead on these platforms due to additional snooping operations required: the system must query the snoop filter to locate the latest data and broadcast snoop requests to upper-level caches. This overhead becomes measurable, ranging from 1% to 6%, when the memory region size reaches the GB scale.

R_I/Inst supports both inter-kernel and intra-kernel boundaries. We implemented R_I/Inst by performing one flush per block with intra-block synchronization. This approach balances performance and generality: it reduces invalidation frequency compared to per-thread invalidation, while

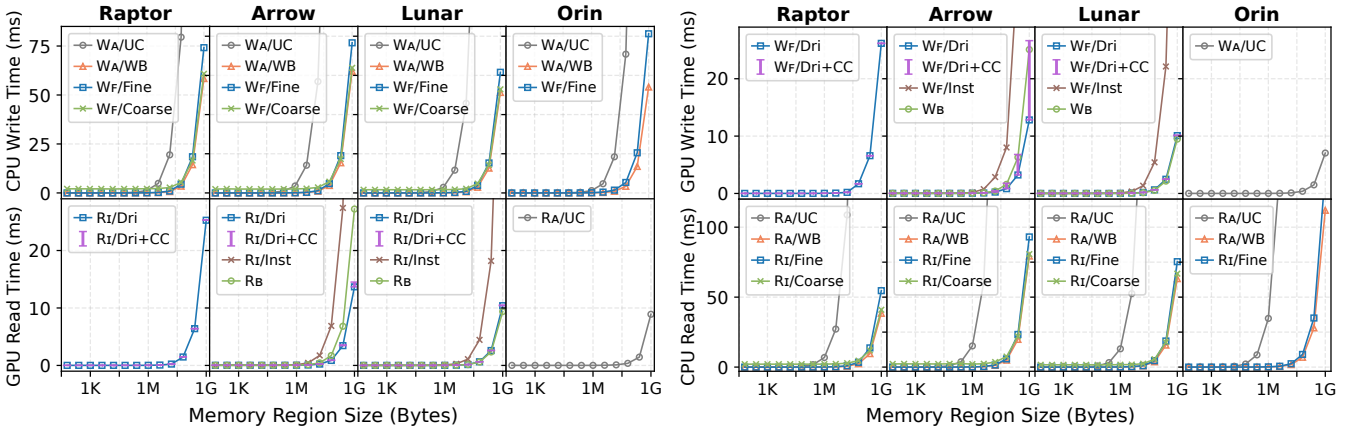

Figure 6. *Performance of WRITE and READ implementations for memory sharing from CPU to GPU (left) and from GPU to CPU (right). Vertical bars mark the GPU time overhead from enabling CC on Raptor, Arrow, and Lunar.*

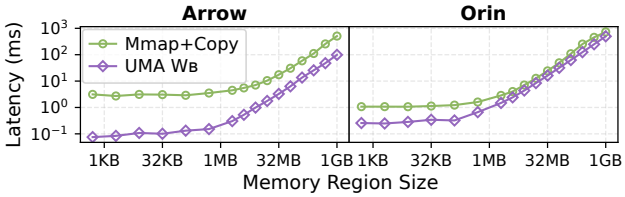

Figure 7. *Performance of CPU WRITE from SSD to memory.*

eliminating the need for implementations to avoid deadlocks in global synchronization scenarios. However, even with these optimizations, R_I/`Inst` is still the slowest option on both Arrow and Lunar.

R_B performance depends heavily on MSC. We hypothesize that the MSC coalesces fine-grained R_B requests, reducing pressure on the memory controller. On Lunar (with MSC), R_B slightly outperforms R_I/`Dri` for large memory regions by skipping invalidation operations. On Arrow (without MSC), R_B read time nearly doubles for regions exceeding 16 MB compared to R_I/`Dri`. Additionally, R_B works on both inter-kernel and intra-kernel boundaries.

NVIDIA GPUs. Orin uses UC for registered host memory [79], and its runtime does not expose a way to disable CC. Consequently, Orin only supports Read-Alone (R_A) on UC memory. However, in our streaming microbenchmarks, its performance remains comparable to that on WB memory; we conjecture that CC overhead is negligible here. The GPU hides the latency of uncacheable accesses through extensive thread-level parallelism—a possible rationale behind NVIDIA's design to use UC policy and enable CC for registered host-memory access on Orin. Nevertheless, while GPU workloads generally feature streaming access patterns with limited temporal locality [58, 116], real-world applications exhibit this locality at the intra-kernel level. Therefore, bypassing the cache incurs a notable performance penalty. For example, when the working set fits within the L2 cache, UC access experiences a 4× latency increase compared to cached access. An L2-optimized GEMM workload on Orin degrades by 16% (from 2,680 to 2,250 GFLOPS) on UC memory.

Implications. UC memory simplifies sharing but increases CPU write latency by up to 300× and penalizes GPU workloads with temporal locality. It should be reserved for cases where neither CC nor access-code modification is feasible (Finding 2). Enabling GPU CC permits CPU writes to WB memory without explicit flushing. Its small GPU-read overhead on Lunar is consistent with low-cost coherence on MSC-enabled platforms. Arrow also incurs 1–6% read overhead without MSC; the larger MSC-related difference arises in GPU writes (§4.2.3) (Finding 3). For I/O-sourced transfers, DMA into the shared region avoids intermediate CPU buffering, redundant copies, page-cache management, and explicit coherence maintenance. This confirms established practice described in NVIDIA's GPUDirect Storage guidance [85].

We summarize these conclusions as CPU-to-GPU implementation choices in Table 2. For CPU-generated data, use Write-Alone on WB memory (W_A^{WB}) with GPU CC enabled; for I/O data, use DMA-based Write-Bypass (W_B^{WB}). On Intel GPUs, use driver-triggered Read-after-Inval (R_I^{WB}) between kernels and Read-Bypass (R_B^{WB}) within kernels on Arrow and Lunar. Raptor does not support intra-kernel communication. On Orin, Read-Alone on UC memory (R_A^{UC}) is the only option for registered host memory.

4.2.3 From GPU to CPU (GPU⇒CPU)

Communication from GPU to CPU over UMA involves a GPU-side sending stage (GPU WRITE) followed by a CPU-side receiving stage (CPU READ) on shared memory. We follow the same approach to achieve optimal performance.

GPU WRITE. As shown in Figure 6, on Raptor, Lunar, and Orin, the performance characteristics of GPU WRITE implementations align with those of GPU READ. Both write and read operations saturate memory bandwidth, and overheads from flushing or bypassing remain comparable. However, on Arrow, GPU write time with CC is 2.07× that without CC

Table 2. *Default CPU-GPU memory sharing implementations selected by* `libumsh` *for our target platforms. Note that* W_A^{WB} *and* R_A^{WB} *require the opposite GPU to have CC enabled.*

CPU⇒GPU	Raptor	Arrow	Lunar	Orin
I/O Device	$W_B^{WB} \Rightarrow R_I^{WB}$			$W_B^{WB} \Rightarrow R_A^{UC}$
Inter-Kernel	$W_A^{WB} \Rightarrow R_I^{WB}$			$W_A^{WB} \Rightarrow R_A^{UC}$
Intra-Kernel	N/A	$W_A^{WB} \Rightarrow R_B^{WB}$		
GPU⇒CPU	**Raptor**	**Arrow**	**Lunar**	**Orin**
Inter-Kernel	$W_F^{WB} \Rightarrow R_A^{WB}$	$W_F^{WB} \Rightarrow R_I^{WB}$	$W_F^{WB} \Rightarrow R_A^{WB}$	$W_A^{UC} \Rightarrow R_A^{WB}$
Intra-Kernel	N/A	$W_B^{WB} \Rightarrow R_I^{WB}$	$W_B^{WB} \Rightarrow R_A^{WB}$	

(W_F/Dri+CC vs. W_F/Dri). The smaller overhead on Lunar is consistent with its MSC reducing DRAM traffic.

CPU Read. On the CPU side, read and write performance remains comparable across all four platforms. Note that on the Arm CPU, the invalidation instruction (`dc ivac`) is much slower than the flush instruction (`dc civac`), even though it skips writeback. Therefore, on Orin, R_I/Fine uses the flush instruction, consistent with W_F/Fine.

Implications. CC simplifies CPU reads with negligible GPU-write overhead on Lunar, which has an MSC. On Arrow, without an MSC, GPU write time with CC is 2.07× that without CC, making coherence a significant cost (Finding 3).

Table 2 summarizes GPU-to-CPU choices. For inter-kernel communication on Intel platforms, use driver-triggered Write-then-Flush (W_F^{WB}). Raptor and Lunar retain GPU CC with CPU Read-Alone (R_A^{WB}); Arrow disables GPU CC to avoid coherent-write overhead and uses CPU Read-after-Inval (R_I^{WB}) to invalidate stale cache lines before reading. For intra-kernel communication, use GPU Write-Bypass (W_B^{WB}): Lunar retains CC with CPU Read-Alone, whereas Arrow disables CC and uses CPU Read-after-Inval (R_I^{WB}) to avoid coherent-write overhead. Raptor does not support intra-kernel communication. On Orin, use GPU Write-Alone on UC memory (W_A^{UC}) and CPU Read-Alone on WB memory (R_A^{WB}).

4.2.4 High Contention

Control-plane coordination suffers from performance degradation due to massive contention on small shared data, such as concurrent reads on the GPU side. As shown in Figure 8 (left), on all platforms, Read-Bypass (R_B) performance degrades sharply as thread count increases. We evaluated three approaches to limit concurrency on Orin (see Figure 8 right):

- **Baseline** uses no barriers—all threads read concurrently.
- **Block** enables block-level barriers—only one thread in each block reads the shared data. It offers a simple implementation that scales well for thread counts up to 1,000.
- **Grid** enables grid-level barriers—only one thread across all blocks reads the shared data. It requires support for Cooperative Groups [78] or atomic operations with deadlock avoidance techniques [32]. While incurring a fixed overhead, this approach offers stable performance and outperforms Block when exceeding 1,000 threads.
- **Kernel** uses a separate kernel with one thread to read shared data. Its performance is consistently lower than Grid due to kernel launch overhead, but it avoids deadlock risks and requires no special instructions.

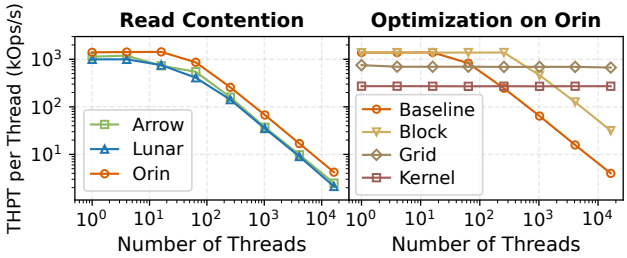

Figure 8. *GPU R_B performance as the thread count increases. The number of threads per block is up to 256.*

Implications. High GPU concurrency severely degrades performance in control-plane coordination, making contention management essential. We recommend *block-level barriers* to mitigate contention when thread counts are below 1,000. For higher concurrency, *grid-level barriers* are preferred to eliminate contention entirely, though they require more complex implementations. The kernel-level approach should be used only when source code modification is infeasible (Finding 4).

5 libumsh and Platform Extension

Drawing on these findings, we built *libumsh*, a library that delivers correct and efficient CPU-GPU memory sharing across UMA platforms. Developers compose sending and receiving processes from six core operations: `bind`, `unbind`, `write`, `read`, `acquire`, and `release`. On top of these, *libumsh* provides high-level helpers for control-plane coordination: `wait` and `set` (built on `read` and `write`) and `lock` and `unlock` (built on `acquire` and `release`) for asymmetric and symmetric synchronization, respectively.

Applications specify the transfer direction, I/O origin, and inter-kernel or intra-kernel boundary. Given these requirements and the compile-time target platform, *libumsh* automatically selects default Read and Write implementations and cache and coherence settings (Table 2). The backend handles platform-specific binding and cache maintenance.

We deploy *libumsh* on all four UMA platforms (see §3). It builds on the stock Intel GPU runtime on Raptor and the CUDA runtime on Orin; Arrow and Lunar additionally require custom runtime modifications to expose the fine-grained CC control APIs. The core logic spans about 1,000 lines of code (LoC). *libumsh* also ships with a companion suite of correctness tests and performance benchmarks for characterizing hardware behavior on new platforms. With compile-time defaults, template-based backend dispatch, and inlined functions, *libumsh* adds less than 1% overhead over manually applying the same implementation choices.

5.1 Porting Workflow

Extending `libumsh` to a new UMA platform requires two tasks: identifying which platform properties behind our findings still apply on the target, and learning how to configure each through the APIs exposed by its driver and runtime. We use the correctness and performance suite from §4 to validate behavior, and rely on AI-assisted reading of code and documentation to narrow the search. We distill the procedure into a repeatable three-step workflow:

- **Locate APIs.** AI-assisted scanning of runtimes, drivers, kernel modules, and documentation surfaces the APIs that configure page size, cacheability, and coherence, and flags internal behaviors with no public API.
- **Validate behavior.** With AI assistance, we adapt our tests and benchmarks to the identified controls, retaining the sending and receiving model and verdict criteria in §4.1.
- **Adapt the platform backend.** We map the validated controls to `libumsh`'s per-platform backend, patching the driver only when a missing control must be exposed.

5.2 DGX Spark Extension

This extension tests whether our page-size, cache-policy, and coherence abstractions support a two-way-coherent UMA platform while preserving `libumsh`'s high-level APIs.

Platform details. We applied this workflow as a post-study extension to NVIDIA DGX Spark [111]. Its GB10 Grace Blackwell Superchip integrates a 20-core Arm CPU (10 Cortex-X925 and 10 Cortex-A725 cores), a Blackwell GPU, and 128 GB of LPDDR5x unified system memory. The 256-bit memory interface provides 273 GB/s bandwidth [80]. The CPU and GPU communicate over NVLink-C2C with AMBA CHI interoperability, and a 16 MB MSC serves both CPU and GPU accesses. Spark shares Orin's NoC-with-MSC organization, but supports two-way CPU–GPU coherence, extending beyond the core study's one-way coherence.

Library extension. Following §5.1, AI-assisted inspection of NVIDIA's open-source GPU kernel modules, UVM driver, CUDA runtime behavior, and documentation located page-size, cacheability, and two-way CC controls. Analysis of about 50 K lines of driver code identified APIs and internal controls; a roughly 200-line kernel patch exposed controls without public interfaces. Following our suite, AI generated about 16 K lines of correctness tests and performance microbenchmarks. The `libumsh` backend required about 100 lines to map library operations to the validated controls. Measurements and case studies use the same interfaces and helpers, with no Spark-specific application changes. The backend independently selects GPU cacheability (WB or UC) and coherence (one-way or two-way).

Performance of memory sharing. Spark preserves THP as a practical default for shared-memory binding. The CPU-side recommendations also carry over: Write-Alone on WB memory (W_A^{WB}) for CPU-generated data and Write-Bypass (W_B^{WB}) for I/O-sourced data. Our results favor UC for large streaming GPU writes (W_A^{UC}), while coherent WB remains practical for GPU reads (R_A^{WB}) and preferable for small GPU-generated control flags ($W_A^{WB} \Rightarrow R_A^{WB}$). For the largest streaming transfers, GPU Read-Alone and Write-Alone on WB memory with two-way CC lose about 8% and up to 27% throughput, respectively, relative to their UC counterparts. In our GPU polling benchmark, *kernel-level* throttling has a lower fixed overhead than on Orin and is preferred under high contention; *block-level barriers* remain an alternative when polling must stay in the original kernel. Appendix B provides detailed measurements, implementations, and case-study results.

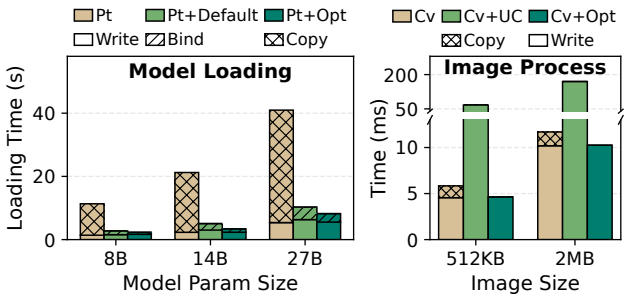

Figure 9. *Data-plane optimization from CPU to GPU on Arrow.*

6 Case Study

To demonstrate our findings and the library, this section presents case studies that optimize nine real-world applications on our four target UMA platforms. The application integrations use `libumsh`'s default platform implementations. Due to space constraints, we show only representative results from these platforms. Optimized systems are marked as "+Opt". Intermediate variants isolate the effects of individual implementation choices.

6.1 Data-plane Transmission

Data-plane transmission typically occurs across kernel boundaries. Some scenarios are sensitive to Bind latency while others prioritize Read and Write latency.

6.1.1 CPU-to-GPU Transfer

CPU-to-GPU transfer is essential for tasks where the CPU prepares data for GPU consumption. Common scenarios include model weight loading, inference input preparation, texture uploading, and software video decoding.

Model loading. PyTorch conventionally loads models by loading (Write) and copying them to GPU (Copy). We optimize this by loading directly into unified memory using W_B, eliminating the redundant copy. Implemented on Arrow (~300 LoC), we evaluated this across various LLMs using both default 4 KB pages (Pt+Default) and THP (Pt+Opt) to assess page-size effects on Bind (see Finding 1). Notably, the baseline copy metric includes lazy Bind overhead. As shown in Figure 9 (left), by bypassing the copy phase, our default approach reduces loading time by 75% and halves memory

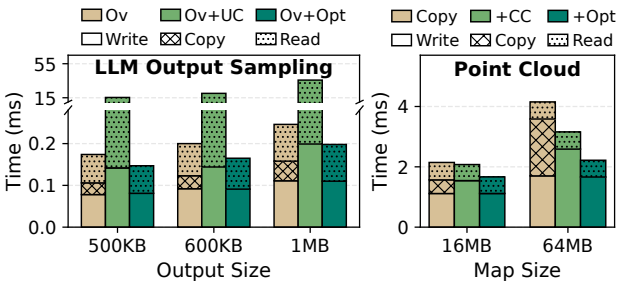

Figure 10. *Data-plane optimization from GPU to CPU on Arrow.*

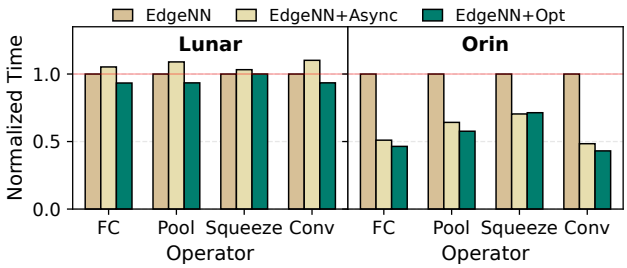

Figure 11. *Control-plane optimization for hybrid inference on Lunar (left) and Orin (right).*

consumption. THP yields an additional 5% time reduction by minimizing Bind overhead.

Image preprocessing. Image preprocessing is a critical stage in multimodal inference. We use Arrow to compare CPU UC/WB access costs with GPU-side CC (see Finding 2). We optimized OpenCV [90] by intercepting its memory allocator to employ unified memory, bypassing the copy. With *libumsh*, the overall modification requires only a few LoC. We evaluated the time of CPU processing (Write) and GPU transfer (Copy). As shown in Figure 9 (right), while direct access eliminates copy overhead, employing naive UC memory (Cv+UC) increases the write time by 20×. In contrast, selecting WB memory with GPU-side CC (Cv+Opt) bypasses the copy step without degrading write performance, reducing total time by 16%.

6.1.2 GPU-to-CPU Transfer

GPU-to-CPU transfer is required for CPU consumption of GPU outputs. Key examples include getting inference results and handling screen captures for recording or streaming.

LLM output sampling. During LLM inference with OpenVINO GenAI [91], model logits must be transferred to the CPU for sampling after each iteration. We use Arrow to examine CPU-read cacheability (see Finding 2). We optimize this by substituting the default output tensor with a pre-allocated buffer via Bind, bypassing data copying. This integration requires only ~50 LoC. As shown in Figure 10 (left), using naive UC memory (Ov+UC) incurs a high overhead for Read. In contrast, Arrow's default configuration (Ov+Opt) disables GPU CC and pairs $\mathtt{W}_{\mathtt{F}}^{\mathtt{WB}}$ with $\mathtt{R}_{\mathtt{I}}^{\mathtt{WB}}$: the CPU invalidates its cache before reading WB memory. This eliminates copying without degrading Read performance. It results in an 18% reduction in end-to-end time.

Sparse point cloud processing. In 3D spatial data processing, converting sparse point clouds to dense images on the GPU for CPU-side processing [117, 123] often incurs significant transmission costs. On Arrow, an initial optimization using $\mathtt{W}_{\mathtt{F}}$ and $\mathtt{R}_{\mathtt{A}}$ with CC (+CC) skipped the copy phase, but as shown in Figure 10 (right), it incurred a 1.5× Write time penalty because Arrow lacks MSC and the *high-intensity GPU writes* were bottlenecked by CC (see Finding 3). For +Opt, *libumsh* uses Arrow's default inter-kernel configuration in Table 2, with GPU-side CC disabled and CPU cache invalidation before reading. This retains zero-copy benefits while mitigating Write overhead, achieving a 34% reduction in data transfer latency compared to the copy baseline.

6.2 Control-plane Coordination

Control-plane coordination employs small memory regions and prioritizes throughput for the Read and Write operations. There is no significant difference in latency among the various Read and Write implementations.

6.2.1 Asymmetric Synchronization

Asymmetric synchronization is a basic pattern for sequencing cross-device operations and enforcing dependencies. Common scenarios include the CPU waiting for GPU RPC calls or the GPU awaiting host data preparation. Traditional approaches rely on high-overhead mechanisms, such as explicit kernel launches or runtime events. UMA platforms allow us to leverage *direct access* for lightweight control. This mechanism employs a one-way signal that repeatedly reads a flag and writes it once, using `wait/set` in *libumsh*.

Hybrid inference. To address the synchronization overhead in CPU-GPU hybrid inference, we leveraged EdgeNN [127] to evaluate a control flow optimization based on direct access signaling. We use Lunar and Orin to compare synchronization across two vendors' NoC-with-MSC platforms. By default, EdgeNN employs a serialized execution model: launching the GPU kernel, performing CPU computation, and then blocking until GPU completion. To mitigate the high control overhead associated with frequent kernel launches, we initially decoupled the launch and compute phases using native runtime *events* (EdgeNN+Async), with ~100 LoC. However, as shown in Figure 11, although this approach prevents the kernel launch from blocking CPU execution, the runtime overhead proved detrimental on the Lunar platform, increasing latency to as much as 1.1× the baseline. We replaced the *events* with our `wait/set` operations (~50 LoC), enabling UMA-based lightweight signaling (EdgeNN+Opt). This optimization finally reduced operator execution time by 4.9% on Lunar and 45.4% on Orin on average compared to EdgeNN.

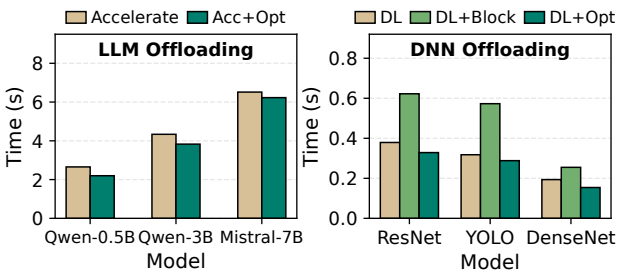

Figure 12. *Offloading inference: Accelerate on Arrow (left) and Demand Layering on Orin (right).*

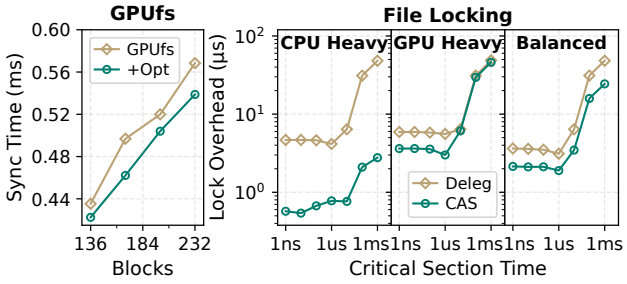

Figure 13. *Control-plane optimization for GPUfs and file locking on Orin. Deleg denotes Delegation on the right figure.*

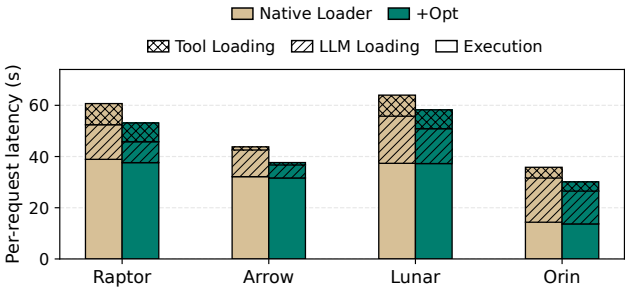

Figure 14. *Estimated per-request agent-serving latency on TaskBench traces, including three tool calls.*

Model offloading. We address the synchronization overhead between weight loading and kernel execution. First, we targeted Accelerate [39] on Arrow to demonstrate a black-box scenario where JIT-generated operators preclude source code modification. We decoupled the weight-loading and kernel-launch phases with a *kernel-level approach* to control the subsequent kernel execution, managing dependencies via `libumsh` (~200 LoC, Acc+Opt). As shown in Figure 12 (left), evaluation on a 10-token prefill workload demonstrates a TTFT reduction of 11%.

To apply Orin's contention-control techniques (Figure 8), we optimized Demand Layering [53] on Orin, where operator source code can be modified directly. Modifications to individual operators required only a few LoC, while the remaining integration involved ~200 LoC. Given that concurrent R_B operations heavily impact read throughput, we evaluated the use of both *block-* and *grid-level barriers* to throttle concurrency. As illustrated in Figure 12 (right), results indicate that *block-level barriers* (DL+Block) backfired, increasing execution time to 1.8× the baseline. Conversely, while *grid-level barriers* (DL+Opt) introduced complexity due to deadlock avoidance, this effectively managed contention, reducing execution time by 14%.

GPU file access. Beyond computation and memory, invoking system services constitutes a critical workload. We use Orin to compare block- and grid-level coordination under the polling contention identified in Figure 8. GPUfs [110] facilitates this by delegating GPU file requests to the CPU via memory-based synchronization with *block-level barriers* (GPUfs). However, our analysis reveals that for kernel-wide file operations, this approach still suffers from contention. At high block counts, *grid-level barriers* (+Opt) reduce synchronization time by about 7% (Figure 13, left).

6.2.2 Symmetric Synchronization

Symmetric synchronization is characterized by the CPU and GPU contending for shared state, necessitating atomic operations to prevent race conditions. Examples include mediating concurrent writes in a shared filesystem or updating arrival counters during barrier synchronization. We select the cross-device lock as the representative case for analysis.

Cross-device file locking. GoFS [67] manages shared file consistency between the CPU and GPU via a GPU-resident delegation server. To reduce the overhead of this centralized approach, we implemented a direct-access *CAS lock* (via `lock` and `unlock` in `libumsh`) on Orin, which supports CPU–GPU atomics on shared host memory, to replace the default *delegation lock*. We evaluated the lock overhead, defined as the time per lock operation minus the critical-section time, under low contention where one CPU thread and one GPU thread compete for the same lock. As shown in Figure 13 (right), in the CPU-heavy scenario, for file operations whose critical-section time is on the microsecond scale, the *CAS lock* reduces lock overhead by about 90% compared with the *delegation lock*. This advantage arises because *CAS* operates directly on shared memory, avoiding the RPC overhead.

6.3 End-to-End Evaluation

Our agent-serving evaluation uses TaskBench traces [109], in which a GLM-4.7-Flash orchestrator invokes ViT, SD-Turbo, and DETR sequentially. The LLM runs in `llama.cpp` and the tools in PyTorch; runtimes remain warm while models reload on demand. Across four platforms, we compare `libumsh` with native loaders using estimated per-request latency: the sum of the separately measured means of orchestration time and the three tools' combined execution time, both including model loading.

Figure 14 shows that `libumsh` reduces estimated per-request latency by 9.0–15.9%, with model loading accounting for most of the savings. On Arrow, latency decreases by 14.0% (from 43.78 s to 37.65 s). LLM loading before prefill falls by

51.3% (from 10.40 s to 5.06 s), accounting for 5.34 s of the 6.13 s total saving. These results show how the model-loading benefits in Figure 9 translate into lower serving latency.

7 Discussion

Limited visibility and control through OS abstractions. OS abstractions simplify development but hide critical parameters (page size, device CC status, and cache policy) from user space. For example, THP treats hugepage allocation as a best-effort hint and silently falls back to 4 KB pages under fragmentation, causing unpredictable performance drops [65, 94]. `libumsh` mitigates the worst cases by prefaulting THP regions (§4.2.1) and patching the driver to expose hidden parameters (§5), but the underlying lack of visibility remains a systemic obstacle to correct and efficient CPU-GPU memory sharing.

The evolution and caveats of hardware coherence. Hardware CC reduces the software burden of memory sharing, but using it correctly still demands platform-specific analysis. The shift from one-way to two-way coherence appears to simplify development, yet in practice it complicates analysis and optimization. AMD's two-way coherence omits the GPU L0/L1 caches and still requires Write-Bypass instructions for correctness [73]; NVIDIA explicitly recommends benchmarking because of potential overheads [79]. The DGX Spark extension in §5.2 illustrates why a wider coherence domain still requires validation of cache coverage and performance.

Beyond CPU-GPU memory sharing. Our methodology applies to other heterogeneous components in UMA platforms, such as NPUs and ASICs. While the logic for analyzing correctness and performance is universal, most ISAs and runtimes remain closed and undocumented [21, 37, 57]. The AI-assisted porting workflow (§5.1) narrows the search for controls, as the DGX Spark extension demonstrates. Applying it to other accelerator classes still requires access to their hardware controls and validation of cache visibility and synchronization behavior.

8 Related Work

Memory performance of UMA platforms. Prior research on UMA memory performance can be categorized into three categories. First, many studies examine isolated architectural features, such as interconnects [26, 105, 112], memory access [13, 64, 96, 125], and cache coherence [2, 5, 6, 77, 97, 130]. Second, most are designed for specific platforms, limiting their applicability across different architectures [24, 30, 34, 40, 54, 121]. Third, while some cross-platform analyses exist [18, 74, 113, 118, 128], they either focus on performance data without addressing correctness, or they target narrow application scenarios. Our analysis provides a comprehensive methodology that spans both correctness and performance across diverse platforms and use cases.

CPU-GPU communication. Prior research has extensively investigated and optimized CPU-GPU communication. For data-plane transmission, researchers have used unified memory to achieve transparent data transfer between the CPU and GPU [4, 28, 56], or optimize transmission for specific scenarios [72, 106]. For control-plane coordination, optimization efforts have mainly focused on hybrid computing [20, 62, 68, 124, 126, 127] and task scheduling [23, 108]. Cross-platform libraries such as Kokkos [63] abstract memory spaces, while CUDA [79] provides vendor-specific allocation and registration APIs. However, these efforts are largely conducted in isolation. Our work provides a comprehensive analysis of the application-level requirements governing CPU-GPU communication and a library that automatically selects implementations across platforms.

9 Conclusion

This paper systematically characterizes the correctness and performance of CPU-GPU memory sharing on unified memory architectures. By decomposing sharing into memory binding and paired write and read operations, our methodology exposes how page size, cache policy, coherence, MSC, and contention interact across four representative platforms. Our correctness tests and four performance findings show why direct access alone does not ensure correct and efficient sharing: implementation choices must account for both the platform's memory hierarchy and the visibility and synchronization requirements of each sharing pattern.

Guided by this, we developed `libumsh`, which encapsulates platform-specific implementations behind high-level sender-receiver operations and automatically selects defaults according to application requirements. Its application to nine real-world systems demonstrates benefits for both data-plane transmission and control-plane coordination. The NVIDIA DGX Spark extension applies the same methodology and high-level APIs to a platform with two-way coherence. Together, the methodology, validation suite, and library provide a reusable approach to supporting new UMA platforms through correctness testing, performance measurement, and adaptation of platform-specific implementations.

Acknowledgments

We sincerely thank our shepherd and the anonymous reviewers for their constructive feedback and suggestions. This work was supported in part by the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (JYB2025XDXM122) and the National Natural Science Foundation of China (No. 62672297, 62432010).

References

[1] Advanced Micro Devices. 2026. Versal Architecture and Product Data Sheet: Overview. https://docs.amd.com/api/khub/documents/BY0FMkLH05y85Nwu4GmWsw/content. DS950, version 2.11.

[2] Neha Agarwal, David Nellans, Eiman Ebrahimi, Thomas F. Wenisch, John Danskin, and Stephen W. Keckler. 2016. Selective GPU caches to eliminate CPU-GPU HW cache coherence. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 494–506. doi:10.1109/HPCA.2016.7446089

[3] Itir Akgun, Jia Zhan, Yuangang Wang, and Yuan Xie. 2016. Scalable memory fabric for silicon interposer-based multi-core systems. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*. 33–40. doi:10.1109/ICCD.2016.7753258

[4] Tyler Allen and Rong Ge. 2021. In-depth analyses of unified virtual memory system for GPU accelerated computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 64, 15 pages. doi:10.1145/3458817.3480855

[5] Johnathan Alsop, Weon Taek Na, Matthew D. Sinclair, Samuel Grayson, and Sarita Adve. 2022. A Case for Fine-grain Coherence Specialization in Heterogeneous Systems. *ACM Trans. Archit. Code Optim.* 19, 3, Article 41 (Aug. 2022), 26 pages. doi:10.1145/3530819

[6] Johnathan Alsop, Matthew Sinclair, and Sarita Adve. 2018. Spandex: A Flexible Interface for Efficient Heterogeneous Coherence. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 261–274. doi:10.1109/ISCA.2018.00031

[7] Nadav Amit, Muli Ben-Yehuda, and Ben-Ami Yassour. 2010. IOMMU: strategies for mitigating the IOTLB bottleneck. In *Proceedings of the 2010 International Conference on Computer Architecture* (Saint-Malo, France) *(ISCA'10)*. Springer-Verlag, Berlin, Heidelberg, 256–274. doi:10.1007/978-3-642-24322-6_22

[8] Apple Inc. 2025. Apple Watch. https://www.apple.com/apple-watch-series-11/specs/.

[9] Apple Inc. 2025. iPhone. https://support.apple.com/en-us/125090.

[10] Apple Inc. 2025. Mac Studio. https://support.apple.com/en-us/122211.

[11] Apple Inc. 2025. MacBook Pro. https://support.apple.com/en-us/125405.

[12] Arm. 2025. AMBA CHI Architecture Specification. https://developer.arm.com/documentation/ihi0050/latest/.

[13] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J. Rossbach, and Onur Mutlu. 2017. Mosaic: a GPU memory manager with application-transparent support for multiple page sizes. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, Massachusetts) *(MICRO-50 '17)*. Association for Computing Machinery, New York, NY, USA, 136–150. doi:10.1145/3123939.3123975

[14] Joshua Bakita and James H. Anderson. 2022. Enabling GPU Memory Oversubscription via Transparent Paging to an NVMe SSD. In *2022 IEEE Real-Time Systems Symposium (RTSS)*. 370–382. doi:10.1109/RTSS55097.2022.00039

[15] Beelink. 2025. Beelink GTR9 Pro AMD Ryzen AI Max+ 395 Processor. https://www.bee-link.com/products/beelink-gtr9-pro-amd-ryzen-ai-max-395.

[16] Tarun Beri, Sorav Bansal, and Subodh Kumar. 2017. The Unicorn Runtime: Efficient Distributed Shared Memory Programming for Hybrid CPU-GPU Clusters. *IEEE Transactions on Parallel and Distributed Systems* 28, 5 (2017), 1518–1534. doi:10.1109/TPDS.2016.2616314

[17] Nadav Bonen, Arik Gihon, Leon Polishuk, Yoni Aizik, Yulia Okunev, Tsvika Kurts, and Nithiyanandan Bashyam. 2025. Lunar Lake an Intel Mobile Processor: SoC Architecture Overview (2024). *IEEE Micro* 45, 3 (2025), 15–21. doi:10.1109/MM.2025.3558407

[18] Roberto Cavicchioli, Nicola Capodieci, and Marko Bertogna. 2017. Memory interference characterization between CPU cores and integrated GPUs in mixed-criticality platforms. In *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. 1–10. doi:10.1109/ETFA.2017.8247615

[19] Hongtao Chen, Weiyu Xie, Boxin Zhang, Jingqi Tang, Jiahao Wang, Jianwei Dong, Shaoyuan Chen, Ziwei Yuan, Chen Lin, Chengyu Qiu, Yuening Zhu, Qingliang Ou, Jiaqi Liao, Xianglin Chen, Zhiyuan Ai, Yongwei Wu, and Mingxing Zhang. 2025. KTransformers: Unleashing the Full Potential of CPU/GPU Hybrid Inference for MoE Models. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles* (Lotte Hotel World, Seoul, Republic of Korea) *(SOSP '25)*. Association for Computing Machinery, New York, NY, USA, 1014–1029. doi:10.1145/3731569.3764843

[20] Le Chen, Dahu Feng, Erhu Feng, Yingrui Wang, Rong Zhao, Yubin Xia, Pinjie Xu, and Haibo Chen. 2025. Characterizing Mobile SoC for Accelerating Heterogeneous LLM Inference. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles* (Lotte Hotel World, Seoul, Republic of Korea) *(SOSP '25)*. Association for Computing Machinery, New York, NY, USA, 359–374. doi:10.1145/3731569.3764808

[21] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[22] Patrick Cronin, Xing Gao, Haining Wang, and Chase Cotton. 2021. An Exploration of ARM System-Level Cache and GPU Side Channels. In *Proceedings of the 37th Annual Computer Security Applications Conference* (Virtual Event, USA) *(ACSAC '21)*. Association for Computing Machinery, New York, NY, USA, 784–795. doi:10.1145/3485832.3485902

[23] Ismet Dagli and Mehmet E. Belviranli. 2024. Shared Memory-contention-aware Concurrent DNN Execution for Diversely Heterogeneous System-on-Chips. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Edinburgh, United Kingdom) *(PPoPP '24)*. Association for Computing Machinery, New York, NY, USA, 243–256. doi:10.1145/3627535.3638502

[24] Marvin Damschen, Frank Mueller, and Jörg Henkel. 2018. Co-Scheduling on Fused CPU-GPU Architectures With Shared Last Level Caches. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2337–2347. doi:10.1109/TCAD.2018.2857042

[25] deepspeedai. 2025. DeepSpeed. https://www.deepspeed.ai/.

[26] Sankha Baran Dutta, Hoda Naghibijouybari, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. 2021. Leaky Buddies: Cross-Component Covert Channels on Integrated CPU-GPU Systems. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 972–984. doi:10.1109/ISCA52012.2021.00080

[27] Agner Fog. 2025-09-20. The microarchitecture of Intel, AMD and VIA CPUs. https://www.agner.org/optimize/microarchitecture.pdf.

[28] Juan Fumero, Florin Blanaru, Athanasios Stratikopoulos, Steve Dohrmann, Sandhya Viswanathan, and Christos Kotselidis. 2023. Unified Shared Memory: Friend or Foe? Understanding the Implications of Unified Memory on Managed Heaps. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (Cascais, Portugal) *(MPLR 2023)*. Association for Computing Machinery, New York, NY, USA, 143–157. doi:10.1145/3617651.3622984

[29] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. 2020. Adaptive Page Migration for Irregular Data-intensive Applications under GPU Memory Oversubscription. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 451–461. doi:10.1109/IPDPS47924.2020.00054

[30] Prasun Gera, Hyojong Kim, Hyesoon Kim, Sunpyo Hong, Vinod George, and Chi-Keung Luk. 2018. Performance Characterisation and Simulation of Intel's Integrated GPU Architecture. In *2018 IEEE*

International Symposium on Performance Analysis of Systems and Software (ISPASS). 139–148. doi:10.1109/ISPASS.2018.00027

[31] ggml-org. 2025. Llama.Cpp. https://github.com/ggml-org/llama.cpp.

[32] Kshitij Gupta, Jeff A. Stuart, and John D. Owens. 2012. A study of Persistent Threads style GPU programming for GPGPU workloads. In *2012 Innovative Parallel Computing (InPar)*. 1–14. doi:10.1109/InPar.2012.6339596

[33] Juan Gómez-Luna, Izzat El Hajj, Li-Wen Chang, Víctor García-Floreszx, Simon Garcia de Gonzalo, Thomas B. Jablin, Antonio J. Peña, and Wen-mei Hwu. 2017. Chai: Collaborative heterogeneous applications for integrated-architectures. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 43–54. doi:10.1109/ISPASS.2017.7975269

[34] Hassan Halawa, Hazem A. Abdelhafez, Andrew Boktor, and Matei Ripeanu. 2017. NVIDIA Jetson Platform Characterization. In *Euro-Par 2017: Parallel Processing*, Francisco F. Rivera, Tomás F. Pena, and José C. Cabaleiro (Eds.). Springer International Publishing, Cham, 92–105.

[35] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale Preemption for Concurrent GPU-accelerated DNN Inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 539–558. https://www.usenix.org/conference/osdi22/presentation/han

[36] Glenn Henry, Parviz Palangpour, Michael Thomson, J Scott Gardner, Bryce Arden, Jim Donahue, Kimble Houck, Jonathan Johnson, Kyle O'Brien, Scott Petersen, Benjamin Seroussi, and Tyler Walker. 2020. High-Performance Deep-Learning Coprocessor Integrated into X86 SoC with Server-Class CPUs Industrial Product. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 15–26. doi:10.1109/ISCA45697.2020.00013

[37] Han Hu, Yujin Huang, Qiuyuan Chen, Terry Yue Zhuo, and Chunyang Chen. 2023. A First Look at On-device Models in iOS Apps. *ACM Trans. Softw. Eng. Methodol.* 33, 1, Article 26 (Nov. 2023), 30 pages. doi:10.1145/3617177

[38] Huawei. 2025. Huawei MDC 610. https://support.huawei.com/enterprise/en/smart-driving/mdc-610-pid-251277587.

[39] Huggingface. 2025. Accelerate. https://github.com/huggingface/accelerate.

[40] Paul Hübner, Andong Hu, Ivy Peng, and Stefano Markidis. 2025. Apple vs. Oranges: Evaluating the Apple Silicon M-Series SoCs for HPC Performance and Efficiency. In *2025 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 45–54. doi:10.1109/IPDPSW66978.2025.00013

[41] Intel. 2025. Instruction Set for Intel Graphics Compiler. https://github.com/intel/intel-graphics-compiler/blob/master/documentation/visa/6_instructions.md.

[42] Intel Corporation. 2023. Intel® Processor Graphics Gen11 Architecture. https://www.intel.com/content/www/us/en/content-details/686065/intel-processor-graphics-gen11-architecture.html.

[43] Intel Corporation. 2023. Optimizing Memory Movement Between Host and Accelerator. https://www.intel.com/content/www/us/en/docs/oneapi/optimization-guide-gpu/2023-2/optimizing-memory-movement-between-host-and.html.

[44] Intel Corporation. 2024. Intel Smart Cache Technology — 13th and 14th Generation Core Processors Datasheet. https://edc.intel.com/content/www/us/en/design/products/platforms/details/raptor-lake-s/13th-generation-core-processors-datasheet-volume-1-of-2/009/011/intel-smart-cache-technology/.

[45] Intel Corporation. 2025. Intel Core Ultra Processors. https://www.intel.com/content/www/us/en/products/details/processors/core-ultra.html.

[46] Intel Corporation. 2025. Intel Processor and Intel Core i3 N-Series Datasheet, Volume 1: Ring Interconnect. https://edc.intel.com/content/www/us/en/design/products/platforms/processor-and-core-i3-n-series-datasheet-volume-1-of-2/002/ring-interconnect/. Document 759603, revision 002.

[47] Intel Corporation. 2025. Intel Smart Cache Technology — Core Ultra 200H and 200U Series Processors Datasheet. https://edc.intel.com/content/www/us/en/design/products-and-solutions/processors-and-chipsets/core-ultra-200h-and-200u-series-processors-datasheet-volume-1-of-2/intel-smart-cache-technology/.

[48] Intel Corporation. 2025. Intel Xe GPU Architecture. https://www.intel.com/content/www/us/en/docs/oneapi/optimization-guide-gpu/2025-2/intel-xe-gpu-architecture.html.

[49] Intel Corporation. 2025. OpenVINO Documentation. https://docs.openvino.ai.

[50] Intel Corporation. 2026. Intel Core Ultra 9 Processor 285H: Specifications. https://www.intel.com/content/www/us/en/products/sku/241747/intel-core-ultra-9-processor-285h-24m-cache-up-to-5-40-ghz/specifications.html.

[51] Intel Corporation. 2026. Intel Processor and Intel Core 3 N-Series Processors. https://cdrdv2-public.intel.com/763908/n-series-processors-product-brief.pdf. Product Brief.

[52] Aamer Jaleel, Eiman Ebrahimi, and Sam Duncan. 2019. DUCATI: High-performance Address Translation by Extending TLB Reach of GPU-accelerated Systems. *ACM Trans. Archit. Code Optim.* 16, 1, Article 6 (March 2019), 24 pages. doi:10.1145/3309710

[53] Mingoo Ji, Saehanseul Yi, Changjin Koo, Sol Ahn, Dongjoo Seo, Nikil Dutt, and Jong-Chan Kim. 2022. Demand Layering for Real-Time DNN Inference with Minimized Memory Usage. In *2022 IEEE Real-Time Systems Symposium (RTSS)*. 291–304. doi:10.1109/RTSS55097.2022.00033

[54] Zheming Jin and Hal Finkel. 2020. Population Count on Intel® CPU, GPU and FPGA. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 432–439. doi:10.1109/IPDPSW50202.2020.00081

[55] John Linford. 2024. Developing for NVIDIA Superchips. https://www.stonybrook.edu/commcms/ookami/_pdf/20240523_Developing_GH_SW_Public.pdf.

[56] Jaehoon Jung, Daeyoung Park, Youngdong Do, Jungho Park, and Jaejin Lee. 2020. Overlapping host-to-device copy and computation using hidden unified memory. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California) *(PPoPP '20)*. Association for Computing Machinery, New York, NY, USA, 321–335. doi:10.1145/3332466.3374531

[57] Sarunas Kalade and Graham Schelle. 2025. NPUEval: Optimizing NPU Kernels with LLMs and Open Source Compilers. *arXiv preprint arXiv:2507.14403* (2025).

[58] Mahmoud Khairy, Mohamed Zahran, and Amr G. Wassal. 2015. Efficient utilization of GPGPU cache hierarchy. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs* (San Francisco, CA, USA) *(GPGPU-8)*. Association for Computing Machinery, New York, NY, USA, 36–47. doi:10.1145/2716282.2716291

[59] Khronos Group Inc. 2025. Vulkan. https://vulkan.org/.

[60] KhronosGroup/MoltenVK. 2018. Support Permanently Open vkMapMemory() on Coherent Image Memory · Issue #175. https://github.com/KhronosGroup/MoltenVK/issues/175.

[61] KhronosGroup/MoltenVK. 2019. Some Linear Images Can't Be Mapped to Memory with VK_MEMORY_PROPERTY_HOST_COHERENT_BIT · Issue #687. https://github.com/KhronosGroup/MoltenVK/issues/687.

[62] Youngsok Kim, Joonsung Kim, Dongju Chae, Daehyun Kim, and Jangwoo Kim. 2019. μLayer: Low Latency On-Device Inference Using Cooperative Single-Layer Acceleration and Processor-Friendly Quantization. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 45, 15 pages. doi:10.1145/3302424.3303950

[63] Kokkos Team. 2026. Memory Spaces — Kokkos Documentation. https://kokkos.org/kokkos-core-wiki/API/core/memory_spaces.html.

[64] Zili Kou, Sharad Sinha, Wenjian He, and Wei Zhang. 2022. Attack Directories on ARM big.LITTLE Processors. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design* (San Diego, California) *(ICCAD '22)*. Association for Computing Machinery, New York, NY, USA, Article 62, 9 pages. doi:10.1145/3508352.3549340

[65] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2017. Ingens: Huge Page Support for the OS and Hypervisor. *SIGOPS Oper. Syst. Rev.* 51, 1 (Sept. 2017), 83–93. doi:10.1145/3139645.3139659

[66] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 155–172. https://www.usenix.org/conference/osdi24/presentation/lee

[67] Shaobo Li, Yirui Eric Zhou, Yuqi Xue, Yuan Xu, and Jian Huang. 2025. Managing Scalable Direct Storage Accesses for GPUs with GoFS. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles* (Lotte Hotel World, Seoul, Republic of Korea) *(SOSP '25)*. Association for Computing Machinery, New York, NY, USA, 979–995. doi:10.1145/3731569.3764857

[68] Yiwei Li and Mingyu Gao. 2024. Hydrogen: Contention-Aware Hybrid Memory for Heterogeneous CPU-GPU Architectures. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15. doi:10.1109/SC41406.2024.00017

[69] Gangmuk Lim, Jeongseob Ahn, Wencong Xiao, Youngjin Kwon, and Myeongjae Jeon. 2021. Zico: Efficient GPU Memory Sharing for Concurrent DNN Training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 161–175. https://www.usenix.org/conference/atc21/presentation/lim

[70] Linux. 2025. I915_gem_object_types.h - Drivers/Gpu/Drm/I915/Gem/I915_gem_object_types.h - Linux Source Code v6.17.7 - Bootlin Elixir Cross Referencer. https://elixir.bootlin.com/linux/v6.17.7/source/drivers/gpu/drm/i915/gem/i915_gem_object_types.h#L429-L430.

[71] Linux Kernel Development Community. 2025. Transparent Hugepage Support. https://docs.kernel.org/admin-guide/mm/transhuge.html.

[72] Jiesong Liu, Feng Zhang, Hourun Li, Dalin Wang, Weitao Wan, Xiaokun Fang, Jidong Zhai, and Xiaoyong Du. 2022. Exploring Query Processing on CPU-GPU Integrated Edge Device. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (2022), 4057–4070. doi:10.1109/TPDS.2022.3177811

[73] LLVM. 2025. User Guide for AMDGPU Backend. https://rocm.docs.amd.com/projects/llvm-project/en/latest/LLVM/llvm/html/AMDGPUUsage.html#memory-model-gfx10-gfx11.

[74] Francesco Lumpp, Hiren D. Patel, and Nicola Bombieri. 2021. A Framework for Optimizing CPU-iGPU Communication on Embedded Platforms. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 685–690. doi:10.1109/DAC18074.2021.9586304

[75] Mark Mansi and Michael M. Swift. 2020. ∅sim: Preparing System Software for a World with Terabyte-scale Memories. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 267–282. doi:10.1145/3373376.3378451

[76] MediaTek Inc. 2025. MediaTek Dimensity Smartphones. https://www.mediatek.com/products/smartphones/dimensity-5g.

[77] Anoop Mysore Nataraja, Ricardo Fernández-Pascual, and Alberto Ros. 2024. Enhanced System-Level Coherence for Heterogeneous Unified Memory Architectures. In *2024 IEEE International Symposium on Workload Characterization (IISWC)*. 273–283. doi:10.1109/IISWC63097.2024.00032

[78] NVIDIA Corporation. 2017. Cooperative Groups: Flexible CUDA Thread Programming. https://developer.nvidia.com/blog/cooperative-groups/.

[79] NVIDIA Corporation. 2025. CUDA for Tegra. https://docs.nvidia.com/cuda/cuda-for-tegra-appnote/index.html#memory-management.

[80] NVIDIA Corporation. 2025. NVIDIA DGX Spark. https://www.nvidia.com/en-us/products/workstations/dgx-spark/.

[81] NVIDIA Corporation. 2025. NVIDIA Grace Hopper Superchip. https://www.nvidia.com/en-us/data-center/grace-hopper-superchip/.

[82] NVIDIA Corporation. 2025. NVIDIA Jetson AGX Orin. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/.

[83] NVIDIA Corporation. 2025. TensorRT. https://developer.nvidia.com/tensorrt.

[84] NVIDIA Corporation. 2026. Atomicity — CUDA Core Compute Libraries. https://nvidia.github.io/cccl/unstable/libcudacxx/extended_api/memory_model.html#atomicity.

[85] NVIDIA Corporation. 2026. GPUDirect Storage Overview Guide. https://docs.nvidia.com/gpudirect-storage/overview-guide/index.html.

[86] NVIDIA Developer Forums. 2016. Zero-Copy and Managed memory on Jetson - Jetson & Embedded Systems / Jetson TX1. https://forums.developer.nvidia.com/t/zero-copy-and-managed-memory-on-jetson/42504.

[87] NVIDIA Developer Forums. 2020. Dual problems with unified memory - Jetson & Embedded Systems / Jetson Nano. https://forums.developer.nvidia.com/t/dual-problems-with-unified-memory/112637.

[88] NVIDIA Developer Forums. 2025. Unified Memory Concurrency - Jetson & Embedded Systems / Jetson AGX Orin. https://forums.developer.nvidia.com/t/unified-memory-concurrency/352261.

[89] Ollama. 2025. Ollama. https://ollama.com.

[90] OpenCV team. 2025. Open Computer Vision Library. https://opencv.org/.

[91] OpenVINOToolkit. 2025. OpenVINO GenAI. https://github.com/openvinotoolkit/openvino.genai.

[92] Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher. 2021. Lord of the Ring(s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 645–662. https://www.usenix.org/conference/usenixsecurity21/presentation/paccagnella

[93] Venkatesh Pallipadi and Suresh Siddha. 2008. PATting linux. In *2008 Linux Symposium*.

[94] Ashish Panwar, Aravinda Prasad, and K. Gopinath. 2018. Making Huge Pages Actually Useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 679–692. doi:10.1145/3173162.3173203

[95] Sihyeong Park, Sungryeol Jeon, Chaelyn Lee, Seokhun Jeon, Byung-Soo Kim, and Jemin Lee. 2025. A Survey on Inference Engines for Large Language Models: Perspectives on Optimization and Efficiency. *arXiv preprint arXiv:2505.01658* (2025).

[96] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. 2014. Architectural support for address translation on GPUs: designing memory management units for CPU/GPUs with unified address spaces. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 743–758. doi:10.1145/2541940.2541942

[97] Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M. Beckmann, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. 2013. Heterogeneous system coherence for integrated CPU-GPU systems. In *Proceedings of the 46th Annual IEEE/ACM International*

Symposium on Microarchitecture (Davis, California) *(MICRO-46)*. Association for Computing Machinery, New York, NY, USA, 457–467. doi:10.1145/2540708.2540747

[98] PyTorch Foundation. 2025. PyTorch. https://pytorch.org.

[99] Shi Qiu, Weinan Liu, Yifan Hu, Jianqin Yan, Zhirong Shen, Xin Yao, Renhai Chen, Gong Zhang, and Yiming Zhang. 2025. GeminiFS: A Companion File System for GPUs. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. USENIX Association, Santa Clara, CA, 221–236. https://www.usenix.org/conference/fast25/presentation/qiu

[100] Qualcomm. 2025. Laptops Powered by Snapdragon. https://www.qualcomm.com/snapdragon/laptops-and-tablets.

[101] Qualcomm. 2025. Snapdragon Smartphones. https://www.qualcomm.com/snapdragon/smartphones.

[102] Raspberry Pi. 2025. Raspberry Pi 5. https://www.raspberrypi.com/products/raspberry-pi-5/.

[103] 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)*. 582–595. doi:10.1109/HPCA47549.2020.00054

[104] Christopher J. Rossbach, Jon Currey, and Emmett Witchel. 2011. Operating systems must support GPU abstractions. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems* (Napa, California) *(HotOS'13)*. USENIX Association, USA, 32.

[105] Anastasiia Ruzhanskaia, Pengcheng Xu, David Cock, and Timothy Roscoe. 2024. Rethinking Programmed I/O for Fast Devices, Cheap Cores, and Coherent Interconnects. *arXiv preprint arXiv:2409.08141* (2024).

[106] Larry Seiler, Daqi Lin, and Cem Yuksel. 2020. Automatic GPU Data Compression and Address Swizzling for CPUs via Modified Virtual Address Translation. In *Symposium on Interactive 3D Graphics and Games* (San Francisco, CA, USA) *(I3D '20)*. Association for Computing Machinery, New York, NY, USA, Article 12, 10 pages. doi:10.1145/3384382.3384533

[107] Seong Hoon Seo, Junghoon Kim, Donghyun Lee, Seonah Yoo, Seokwon Moon, Yeonhong Park, and Jae W. Lee. 2025. FACIL: Flexible DRAM Address Mapping for SoC-PIM Cooperative On-device LLM Inference. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 1720–1733. doi:10.1109/HPCA61900.2025.00127

[108] Weihang Shen, Mingcong Han, Jialong Liu, Rong Chen, and Haibo Chen. 2025. XSched: Preemptive Scheduling for Diverse XPUs. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. USENIX Association, Boston, MA, 671–692. https://www.usenix.org/conference/osdi25/presentation/shen-weihang

[109] Yongliang Shen, Kaitao Song, Xu Tan, Wenqi Zhang, Kan Ren, Siyu Yuan, Weiming Lu, Dongsheng Li, and Yueting Zhuang. 2024. TaskBench: Benchmarking Large Language Models for Task Automation. In *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (Eds.), Vol. 37. Curran Associates, Inc., 4540–4574. doi:10.52202/079017-0148

[110] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. 2014. GPUfs: Integrating a file system with GPUs. *ACM Transactions on Computer Systems (TOCS)* 32, 1 (2014), 1–31.

[111] A. Skende, G. Rosseel, and N. Pinckney. 2026. 17.1 NVIDIA GB10: SoC Built for AI Acceleration. In *2026 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 69. 298–300. doi:10.1109/ISSCC49663.2026.11409351

[112] Alan Smith, Gabriel H. Loh, Michael J. Schulte, Mike Ignatowski, Samuel Naffziger, Mike Mantor, Mark Fowler Nathan Kalyanasundharam, Vamsi Alla, Nicholas Malaya, Joseph L. Greathouse, Eric Chapman, and Raja Swaminathan. 2024. Realizing the AMD Exascale Heterogeneous Processor Vision : Industry Product. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 876–889. doi:10.1109/ISCA59077.2024.00068

[113] Tuan Ta, David Troendle, Xiaoqi Hu, and Byunghyun Jang. 2017. Understanding the Impact of Fine-Grained Data Sharing and Thread Communication on Heterogeneous Workload Development. In *2017 16th International Symposium on Parallel and Distributed Computing (ISPDC)*. 132–139. doi:10.1109/ISPDC.2017.16

[114] TensorFlow. 2015. TensorFlow. https://www.tensorflow.org/. doi:10.5281/zenodo.4724125

[115] The Khronos Group. 2025. MoltenVK. https://github.com/KhronosGroup/MoltenVK.

[116] Yingying Tian, Sooraj Puthoor, Joseph L. Greathouse, Bradford M. Beckmann, and Daniel A. Jiménez. 2015. Adaptive GPU cache bypassing. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs* (San Francisco, CA, USA) *(GPGPU-8)*. Association for Computing Machinery, New York, NY, USA, 25–35. doi:10.1145/2716282.2716283

[117] Chenxi Tu, Eijiro Takeuchi, Alexander Carballo, and Kazuya Takeda. 2019. Real-Time Streaming Point Cloud Compression for 3D LiDAR Sensor Using U-Net. *IEEE Access* 7 (2019), 113616–113625. doi:10.1109/ACCESS.2019.2935253

[118] Shivakumar Udkar, Vikas Gupta, Manjunath Reddy, and Muthukumaran Vaithianathan. 2025. Uniform Memory Architecture for GPU-CPU Resource Sharing in Edge AI Systems. In *2025 5th International Conference on Expert Clouds and Applications (ICOECA)*. 426–431. doi:10.1109/ICOECA66273.2025.00079

[119] Ultralytics Inc. 2025. Ultralytics. https://github.com/ultralytics/ultralytics.

[120] vllm-project. 2025. vLLM. https://github.com/vllm-project/vllm.

[121] Jacob Wahlgren, Gabin Schieffer, Ruimin Shi, Edgar A. León, Roger Pearce, Maya Gokhale, and Ivy Peng. 2025. Dissecting CPU-GPU Unified Physical Memory on AMD MI300A APUs. In *2025 IEEE International Symposium on Workload Characterization (IISWC)*. 368–380. doi:10.1109/IISWC66894.2025.00038

[122] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Online, 38–45. https://www.aclweb.org/anthology/2020.emnlp-demos.6

[123] Bichen Wu, Alvin Wan, Xiangyu Yue, and Kurt Keutzer. 2018. SqueezeSeg: Convolutional Neural Nets with Recurrent CRF for Real-Time Road-Object Segmentation from 3D LiDAR Point Cloud. In *2018 IEEE International Conference on Robotics and Automation (ICRA)* (Brisbane, Australia). IEEE Press, 1887–1893. doi:10.1109/ICRA.2018.8462926

[124] Daliang Xu, Hao Zhang, Liming Yang, Ruiqi Liu, Gang Huang, Mengwei Xu, and Xuanzhe Liu. 2025. Fast On-device LLM Inference with NPUs. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 445–462. doi:10.1145/3669940.3707239

[125] Tianhong Xu, Aidong Adam Ding, and Yunsi Fei. 2025. EXAM: Exploiting Exclusive System-Level Cache in Apple M-Series SoCs for Enhanced Cache Occupancy Attacks. In *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security (ASIA CCS '25)*. Association for Computing Machinery, New York, NY, USA, 1294–1308. doi:10.1145/3708821.3710844

[126] Zhenliang Xue, Yixin Song, Zeyu Mi, Xinrui Zheng, Yubin Xia, and Haibo Chen. 2024. Powerinfer-2: Fast large language model inference

on a smartphone. *arXiv preprint arXiv:2406.06282* (2024).

[127] Chenyang Zhang, Feng Zhang, Kuangyu Chen, Mingjun Chen, Bingsheng He, and Xiaoyong Du. 2023. EdgeNN: Efficient Neural Network Inference for CPU-GPU Integrated Edge Devices. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 1193–1207. doi:10.1109/ICDE55515.2023.00096

[128] Chenyang Zhang, Feng Zhang, Xiaoguang Guo, Bingsheng He, Xiao Zhang, and Xiaoyong Du. 2021. iMLBench: A Machine Learning Benchmark Suite for CPU-GPU Integrated Architectures. *IEEE Transactions on Parallel and Distributed Systems* 32, 7 (2021), 1740–1752. doi:10.1109/TPDS.2020.3046870

[129] Yufeng Zhou, Alan L. Cox, Sandhya Dwarkadas, and Xiaowan Dong. 2023. The Impact of Page Size and Microarchitecture on Instruction Address Translation Overhead. *ACM Trans. Archit. Code Optim.* 20, 3, Article 38 (July 2023), 25 pages. doi:10.1145/3600089

[130] Joseph Zuckerman, Davide Giri, Jihye Kwon, Paolo Mantovani, and Luca P. Carloni. 2021. Cohmeleon: Learning-Based Orchestration of Accelerator Coherence in Heterogeneous SoCs. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 350–365. doi:10.1145/3466752.3480065

Appendix A Artifact Appendix

This artifact provides the source code of `libumsh`, correctness tests, and microbenchmarks for the EuroSys 2027 paper "Characterizing CPU-GPU Memory Sharing over Unified Memory Architecture" by J. Yang, Z. Wang, R. Chen, and H. Chen. `libumsh` is a library that enables CPU-GPU memory sharing on unified memory architectures through its data-plane and control-plane implementations. We provide instructions for building the library and running the tests and microbenchmarks, along with reference measurements and plotting scripts for the performance characterization in §4. Our artifact received the "Artifacts Available" and "Artifacts Evaluated (Functional)" badges from the EuroSys 2027 Artifact Evaluation process. The DOI of our artifact is https://doi.org/10.5281/zenodo.22909405.

Artifact repository. The `libumsh` source code and instructions for building and evaluating it on supported Intel and NVIDIA UMA platforms are available in the following Git repository: https://github.com/XpuOS/libumsh-artifacts.

Appendix B NVIDIA DGX Spark Extension

This appendix reports memory-sharing performance measurements and case studies for NVIDIA DGX Spark. The platform details, library extension, and main conclusions are presented in §5.2. All measurements below exercise the Spark backend added to `libumsh`.

Appendix B.1 Performance of Memory Sharing

This subsection extends the memory-sharing performance study in §4.2 to Spark. We run the corresponding correctness tests and performance benchmarks in `libumsh` to characterize Spark's hardware behavior. We analyze the same implementation choices for binding, CPU-to-GPU transfer, GPU-to-CPU transfer, and GPU-side contention.

Bind. Figure 1 shows the time to set up a memory region for sharing, which includes page table configuration and any associated cache management. Small regions favor 4 KB pages or THP because 2 MB mappings impose a fixed setup cost, while large streaming regions amortize the larger mapping setup. Thus, on Spark, THP remains a practical default: it avoids the fixed setup cost of explicit 2 MB mappings for small regions while not introducing a penalty for large regions.

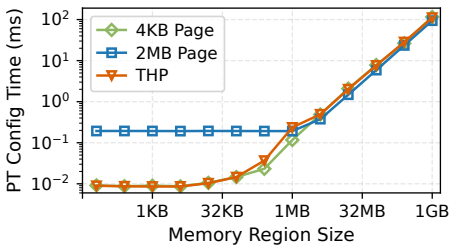

Figure 1. *Page table configuration time for the GPU using different page sizes on Spark.*

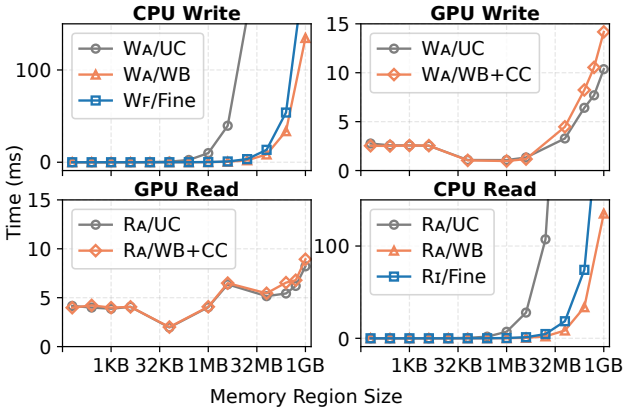

Figure 2. *Performance of Write and Read implementations for memory sharing on Spark, broken down by CPU and GPU access paths. CC denotes two-way coherence.*

Write and Read. Figure 2 reports the performance of the Spark implementations of Write and Read. On the CPU side, Spark exposes the standard CPU access paths used in the main study: W_A/WB for CPU-generated data, W_F/Fine when explicit cache flushing is required, R_A/WB for cached reads, and R_I/Fine when invalidation is needed before reading. The GPU side exposes a richer set of choices than Orin because Spark's GPU driver is open-source: our backend extension can expose additional APIs to `libumsh`, including APIs for selecting WB memory and toggling two-way CC support.

The CPU-side results preserve the main-study choices: WB memory keeps W_A efficient for CPU-generated data, while explicit fine-grained flushing or invalidation is only needed when software must maintain visibility without relying on coherence. The GPU-side results are different: hardware-managed two-way CC does not universally replace software-managed accesses. In our streaming benchmarks, once the shared buffer reaches the MB scale, enabling two-way CC on WB memory can be slower than the UC paths. For the largest transfers, R_A/WB+CC loses about 8% GPU Read throughput relative to R_A/UC, while W_A/WB+CC loses up to 27% GPU Write throughput relative to W_A/UC. Thus, Spark's two-way CC is useful for coherent GPU reads and small control flags, but MB-scale streaming GPU writes should prefer the UC path.

Figure 3 presents the performance of Write-Bypass (W_B) on Spark. For SSD-backed CPU-to-GPU data-plane transmission, W_B avoids page-cache staging and an extra memory copy, so it outperforms the traditional mmap-and-copy path. Overall, Spark does not change the high-level implementation guidance: use bypass-style CPU Write for I/O-sourced data.

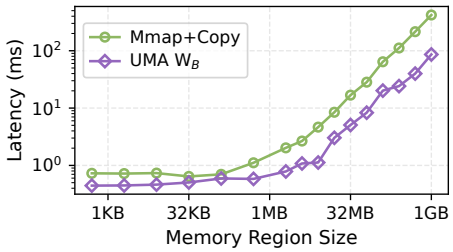

Figure 3. *Performance of CPU Write from SSD to memory on Spark. W_B reduces end-to-end loading time relative to the mmap-and-copy path by avoiding extra staging and copy overhead.*

Table 1. *Recommended CPU-GPU memory sharing implementations on Spark. Note that W_A^{WB} and R_A^{WB} rely on CC support.*

Stage	Case	Recommended path
Bind	General	THP
CPU⇒GPU	I/O Device	$W_B^{WB} \Rightarrow R_A^{WB}$
	Inter-Kernel	$W_A^{WB} \Rightarrow R_A^{WB}$
	Intra-Kernel	
GPU⇒CPU	Inter-Kernel	$W_A^{UC} \Rightarrow R_A^{WB}$
	Intra-Kernel	$W_A^{WB} \Rightarrow R_A^{WB}$

Implications. Table 1 summarizes the recommended CPU-GPU memory sharing implementations on Spark. Spark mostly preserves the main study's recommendations for binding and CPU-side accesses, but changes the GPU-side recommendation: coherent WB memory remains practical for GPU Read and for small GPU-generated flags, whereas GPU Write should prefer the non-coherent UC path for MB-scale streaming transfers. The UC recommendation applies to streaming transfers; applications with intra-kernel reuse also need to consider cache locality (§4.2.2).

For Bind, THP is recommended for general shared regions because it avoids the fixed setup cost of explicit 2 MB mappings for small regions while preserving large-region performance. For CPU-to-GPU communication, DMA-based Write-Bypass (W_B^{WB}) followed by GPU Read-Alone on WB memory (R_A^{WB}) is recommended for data from I/O devices (e.g., SSD), whereas Write-Alone on WB memory (W_A^{WB}) followed by the same GPU read path is appropriate for CPU-generated data. In both cases, Spark should keep the GPU consumer on coherent WB reads rather than switching to UC memory. For GPU-to-CPU communication, Spark should use the non-coherent UC GPU write path (W_A^{UC}) for inter-kernel streaming transfers because it avoids the two-way CC overhead observed for large WB GPU writes. For intra-kernel flag passing, the coherent WB path ($W_A^{WB} \Rightarrow R_A^{WB}$) is preferable because the performance cost of two-way CC is negligible at this scale, and the CPU can read the flag directly.

Two-way coherence. To isolate the source of Spark's two-way CC behavior, we further tested GPU access performance

Figure 4. *GPU Read and Write performance on Spark under different cache and CC policies. 1-CC denotes one-way coherence, and 2-CC denotes two-way coherence.*

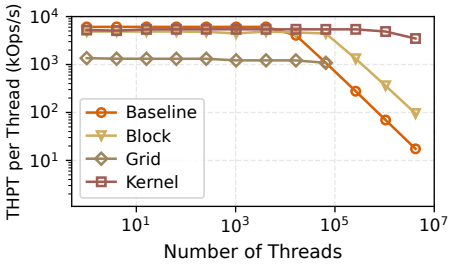

Figure 5. *GPU R_B performance as thread count increases under different concurrency-control approaches on Spark.*

under a 2×2 matrix of memory configurations. Spark allows the backend to configure both cache policy (WB or UC) and CC policy (one-way or two-way), so Figure 4 compares GPU Read and Write performance under the four combinations. Two observations stand out. First, in this streaming test, GPU Read is largely insensitive to both policies, whereas GPU Write varies visibly across configurations. Second, the cost of two-way CC is paid only when the GPU L2 actually caches the data: under WB, two-way CC becomes noticeably slower than one-way CC once the buffer exceeds 32 MB, matching the trend in Figure 2; under UC, the two CC policies are indistinguishable.

We hypothesize that this pattern is a direct consequence of implementing two-way CC using the AMBA CHI protocol. In CHI terminology, one-way CC corresponds to attaching the GPU as an IO-coherent request node (RN-I, equivalent in role to an ACE-Lite agent): the GPU can issue snoopable requests that invalidate CPU caches, but its own cache is not visible to the fabric and, per the CHI specification, does *not* receive snoops. Two-way CC instead attaches the GPU as a fully coherent request node (RN-F), which by definition holds a hardware-coherent cache, must respond to all snoop transactions, and is registered in the Home Node's directory. Under WB, every cache line that enters the GPU L2 therefore incurs directory registration on fill, ownership acquisition before a store can retire, and de-registration on eviction—traffic that one-way CC avoids entirely because the GPU L2 is not a participant in the coherence domain. Under UC, the GPU L2 is bypassed, so no cache-line transitions occur and neither CC policy pays this cost.

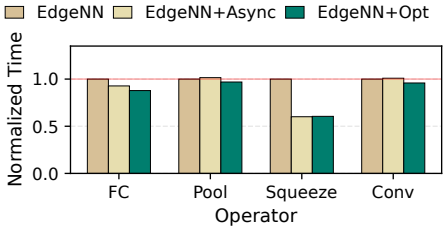

Figure 6. *Control-plane coordination for hybrid inference on Spark.*

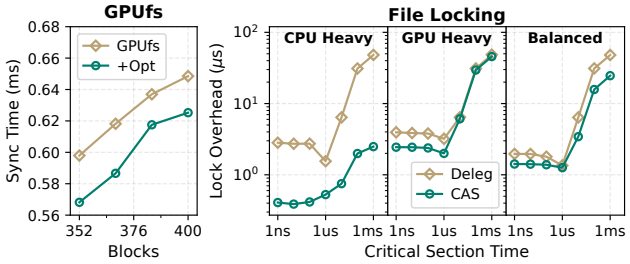

Figure 7. *Control-plane coordination for GPUfs and file locking on Spark. Deleg denotes the delegation lock in the right-hand plot.*

GPU contention. Figure 5 shows how Spark handles GPU-side contention when many GPU threads poll the same shared location through R_B. The *baseline* without barriers remains stable at small and moderate thread counts, but eventually collapses as concurrent requests concentrate on one memory location. Barrier-based throttling is therefore still necessary: *block-level barriers* reduce thread-level contention with low fixed overhead, while *grid-level barriers* remove almost all concurrent polling at the cost of stronger synchronization requirements. On Spark, however, the *kernel-level approach* has a much smaller fixed overhead than on Orin, so it becomes the preferred implementation for high-contention polling. *Block-level barriers* remain the next-best choice when launching a separate polling kernel is undesirable or the polling logic must remain inside the original kernel.

Appendix B.2 Case Study

We reran the Orin-based case studies on Spark through the same platform-independent `libumsh` interfaces. No Spark-specific application code is needed once the backend exposes the required operations.

Hybrid inference. Figure 6 shows that the Orin optimization for hybrid inference also applies to Spark: replacing EdgeNN's runtime-based coordination with `libumsh`'s direct-access `wait/set` signaling reduces normalized operator time by 14.7% on average relative to EdgeNN. The gain is smaller than on Orin because Spark's lower kernel launch overhead leaves less control-plane overhead for the signaling optimization to remove.

GPUfs and file locking. Figure 7 shows that the Orin control-plane optimizations for GPU file access also carry over to Spark. In GPUfs, kernel-wide file operations still create GPU-side polling contention, so replacing *block-level* throttling with *grid-level barriers* lowers synchronization time by up to 5.1% at high block counts. For file locking, replacing the GPU-resident *delegation lock* with a direct-access *CAS lock* lets the CPU and GPU coordinate through shared-memory atomics instead of RPC. This reduces lock overhead by about 85%, 37%, and 26% in the CPU-heavy, GPU-heavy, and balanced configurations, respectively.