Iris: First-Class Multi-GPU Programming Experience in Triton

Muhammad Awad
muhaawad@amd.com
Advanced Micro Devices, Inc.
Santa Clara, CA, USA

Muhammad Osama
muhammad.osama@amd.com
Advanced Micro Devices, Inc.
Santa Clara, CA, USA

Brandon Potter
brandon.potter@amd.com
Advanced Micro Devices, Inc.
Austin, TX, USA

Abstract

Multi-GPU programming traditionally requires developers to navigate complex trade-offs between performance and programmability. High-performance implementations typically rely on low-level HIP/CUDA communication libraries that demand substantial engineering effort for even basic overlap patterns, while simpler abstractions often sacrifice performance. We present Iris, a multi-GPU communication library implemented entirely in Python and Triton that eliminates this trade-off. Iris provides tile-based symmetric memory abstractions that naturally align with Triton's programming model, enabling developers to write single-source kernels that seamlessly interleave computation and communication. We demonstrate a taxonomy of compute-communication overlap patterns—from bulk-synchronous to fine-grained workgroup specialization—that can be implemented with minimal code changes in Iris, often requiring just a few additional lines within the same Triton kernel. Our evaluation shows that Iris achieves near-optimal bandwidth utilization in microbenchmarks and delivers up to 1.79× speedup over PyTorch and RCCL for GEMM+All-Scatter workloads, demonstrating that high-level implementations can match or exceed heavily-optimized libraries while dramatically simplifying multi-GPU programming.

Keywords

Distributed Computing, GPU, Fused Kernels, Triton, Wavefront-specialization

1 Introduction

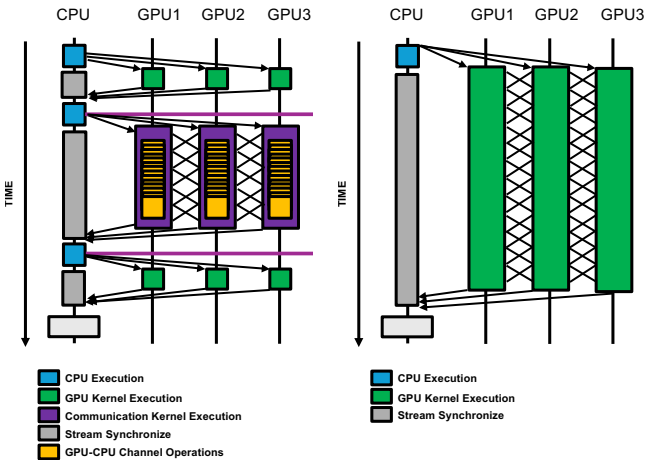
Modern AI workloads demand near-peak performance to extract the full efficiency of AI systems. Teams of specialists with deep understanding of both model characteristics and hardware architecture are required to craft highly optimized training and inference kernels for these AI workloads. Even for seasoned engineers, this process requires iterative refinement, hardware-specific tuning, and extensive experimentation. This challenge arises because contemporary AI models comprise numerous operators, each with multiple potential optimization strategies. Determining the appropriate optimizations depends on a wide range of factors—including model structure, configuration parameters, hardware vendor and generation, system topology, and supporting software ecosystems. Moreover, end-to-end performance is strongly influenced by distributed-parallel execution primitives—such as all-reduce and all-to-all—that orchestrate computation across devices.

Distributed parallelism further amplifies the complexity. Practitioners routinely combine data, tensor, pipeline, and expert parallelism strategies and expect these hybrid approaches to perform consistently across heterogeneous hardware platforms. For system and library developers, this presents a significant challenge: kernel efficiency is tightly coupled to network characteristics and system architecture, both of which vary substantially in real-world deployments. As a result, collective communication libraries must deliver high performance across diverse and evolving environments.

Given this complex landscape, higher-level abstractions are essential to simplify development without sacrificing performance. However, modern accelerators also introduce specialized units and mechanisms—such as tensor cores and asynchronous memory-movement engines (SDMA, TMA, and asynchronous copy instructions)—that demand fine-grained, tile-based programming to fully exploit their capabilities for overlapping communication and computation. A number of efforts have advanced the state of the art, including compiler-driven approaches (e.g., XLA [18], TVM [7], and Triton [26]) as well as template-based libraries (e.g., CUTLASS, CuTe [10], ThunderKittens [24]). Among these, Triton has shown sustained maturity, performance portability, and broad adoption for computation. However, a critical gap remains: while these abstractions have successfully tackled local operators and compute kernels, they have largely overlooked communication as an equally important concern. As distributed training and inference scales to hundreds or thousands of GPUs, communication becomes the dominant performance bottleneck.

The Consensus: Fine-Grained Overlap is Essential. The research and production communities have converged on a clear solution: overlap communication with computation at fine granularity. Rather than executing communication and computation in rigid, sequential bulk-synchronous phases (Figure 1a), modern workloads demand fine-grained overlap where data is communicated as soon as it is produced, at tile granularity (Figure 1b), so that computation can proceed on other tiles without waiting. This fine-grained overlap hides communication latency behind useful work, eliminating the idle "bubbles" that plague bulk-synchronous execution. Recent systems such as TorchTitan's AsyncTP [11] and production LLM training pipelines [8, 23, 27] have demonstrated the necessity of this approach.

Communication: The Missing Abstraction. Fine-grained overlap requires a communication abstraction that operates at the same tile granularity as Triton's computational model. However, Triton provides no such abstraction. While Triton's tile-based programming model has revolutionized how developers write optimized compute kernels—automatically handling memory coalescing, shared memory management, and intra-kernel scheduling–communication remains an afterthought. Developers must rely on external libraries (RCCL [4], NCCL [13]) that operate at coarse kernel boundaries, or hand-craft device-to-device data movement without compiler support. This approach forces communication to remain outside the compiler's purview, preventing the tile-level interleaving that the consensus approach demands. The result is that fine-grained



(a) Bulk-Synchronous **(b) Fine-Grained Overlap**

Figure 1: Execution model comparison: (a) bulk-synchronous execution with rigid sequential phases and CPU-initiated kernel launches versus (b) fine-grained overlap enabling dynamic communication at tile granularity with GPU-initiated execution.

overlap, despite being widely recognized as essential, remains inaccessible to Triton developers.

The Challenge: Architectural Limitations of Existing Approaches. Recognizing this gap, several efforts have attempted to bring communication into Triton. However, these attempts face fundamental architectural constraints. Current efforts typically wrap vendor libraries (rocSHMEM [3], NVSHMEM [16]) as opaque bytecode, linking them into Triton kernels. While pragmatic, this approach introduces several limitations. First, these libraries inherit design constraints from the OpenSHMEM specification and carry technical debt from APIs originally designed for CPU-based distributed computing, which do not align naturally with Triton's tile-centric programming model and introduce anti-patterns that work against compiler optimizations. Second, and more critically, wrapping external libraries as opaque binaries prevents the compiler from seeing communication operations, precluding co-optimization of computation and communication, intelligent scheduling, and unified fusing of kernels across their boundaries. Communication remains a second-class citizen—linked as binary blobs rather than first-class Triton code. Similarly, traditional collective communication libraries (CCLs) such as RCCL and NCCL impose bulk-synchronous semantics with CPU-initiated kernel launches, adding coordination overhead, kernel teardown costs, and redundant memory transfers at kernel boundaries. These architectural constraints limit the ability to achieve the native, compiler-visible, tile-granular primitives required for true fine-grained overlap.

Iris: Native Communication for Tile-Based Programming. We present Iris [5][1], the first multi-GPU library architected from the ground up for Triton's tile-based programming model to full enable the fine-grained computation and communication overlap within AI workloads (Figure 1b). Unlike existing approaches that wrap legacy SHMEM libraries as opaque bytecode, Iris is implemented entirely in Python and Triton, giving the compiler full visibility into both computation and communication. Iris provides native symmetric memory abstractions that enable developers to write concise single-source kernels that seamlessly interleave computation and communication at tile granularity with no external dependencies, no opaque library calls, no bulk-synchronous phases. Iris also offers an experimental Gluon backend using Triton's `@gluon.jit` and `@aggregate` decorators for improved ergonomics, but this paper focuses on the standard Triton API for clarity and broader compatibility. Our contributions are as follows:

- **Native Triton implementation:** The first multi-GPU communication library implemented entirely in Python and Triton, providing full compiler visibility and enabling co-optimization of computation and communication
- **Tile-based symmetric memory API:** Pythonic abstractions that align naturally with Triton's tile-centric programming model, supporting both value-based and pointer-based communication primitives
- **Taxonomy of fused patterns:** A comprehensive classification of compute-communication overlap strategies, including bulk-synchronous, producer-consumer, and workgroup-specialized approaches
- **Performance validation:** Experimental evaluation demonstrating up to 1.79× speedup over PyTorch and RCCL for GEMM+All-Scatter workloads across multiple problem sizes
- **Open-source release:** Fully open-source implementation enabling reproducibility and community adoption

2 Background and Related Work

Iris builds on established GPU communication mechanisms—symmetric memory, direct inter-GPU interconnects, and well-defined memory consistency models—while introducing a novel programming abstraction that distinguishes it from prior work. We first review the underlying hardware and runtime infrastructure, then survey related efforts to integrate communication into GPU programming frameworks.

2.1 Background

We briefly review the key mechanisms that enable multi-GPU communication on AMD platforms. This includes the physical interconnect topology, the runtime interfaces for establishing symmetric memory across processes, and the memory consistency model that provides correctness guarantees. Together, these components form the execution substrate on which Iris is built.

2.1.1 Interconnect Topology. Iris targets scale-up environments where multiple GPUs within a single node are connected via a high-bandwidth interconnect. The AMD Instinct MI300X and MI325X platforms [2] use seven high-bandwidth, low-latency AMD Infinity Fabric links per GPU to form a fully connected 8-GPU system. Each GPU is also connected to the host CPU via a x16 PCIe Gen 5 link. This fully connected mesh topology provides direct peer-to-peer access between any GPU pair without traversing the host CPU, enabling the low-latency, high-bandwidth communication that Iris leverages for efficient multi-GPU operations. Iris exploits

[1]Iris is open-source and available at https://github.com/ROCm/iris

this topology to implement efficient collective operations and point-to-point communication patterns directly in Triton kernels.

2.1.2 Symmetric Memory via IPC. Iris establishes symmetric memory across GPUs using HIP's inter-process communication (IPC) mechanism [1]. Each process allocates device memory using standard `hipMalloc`, then exports handles via `hipIpcGetMemHandle` and imports peer handles via `hipIpcOpenMemHandle`. This enables direct memory access across GPU boundaries: each GPU can read from and write to any peer GPU's memory using simple pointer arithmetic. Iris uses coarse-grained memory semantics for simplicity and portability, relying on the memory consistency model (described below) to ensure correctness of cross-GPU operations. By leveraging IPC, Iris exposes a clean symmetric memory abstraction to programmers, enabling Triton kernels to perform remote memory operations as naturally as local ones.

2.1.3 Memory Model and Synchronization. Iris relies on AMD's memory model to provide formal correctness guarantees in multi-GPU execution. This model has been described in publicly available literature [9] and is implemented concretely in AMDGPU LLVM [12]. The AMD memory model is Sequentially Consistent Heterogeneous Race Free (SC-HRF), analogous to the C++ model but extended with GPU-specific memory scopes. The model supports standard C++ memory orderings (`acquire`, `release`, `acq_rel`, and `seq_cst`) with familiar semantics: acquire operations prevent subsequent loads and stores from being reordered before them, while release operations prevent preceding loads and stores from being reordered after them. Critically, the model introduces hierarchical memory scopes—`wavefront` (warp), `workgroup` (block), `agent` (device), and `system`—that define visibility domains for synchronization operations. Triton already exposes these memory orderings and scopes through its atomic operations API, and Iris leverages Triton's implementation to use hardware-level synchronization primitives directly. For multi-GPU communication, Iris uses `agent`-scoped atomics to synchronize between GPUs within a node, and `system`-scoped operations when broader visibility is required.

Iris adopts this memory model because it is well-established, widely-adopted across CPU and GPU programming, and provides programmers with familiar, intuitive synchronization primitives (acquire/release, memory scopes) that are straightforward to reason about and use, while still delivering provable correctness guarantees for multi-GPU coordination.

2.1.4 Symmetric Memory Programming Models. The symmetric memory programming model, established by the OpenSHMEM specification and implemented by hardware vendors as xSHMEM variants (rocSHMEM [3] and NVSHMEM [16]), provides a foundational abstraction where each process allocates memory in a symmetric heap that is directly accessible by all peers. This model enables one-sided communication primitives—such as remote puts, gets, and atomic operations—that can be issued without requiring explicit receiver-side coordination, making it well-suited for GPU programming where fine-grained producer-consumer patterns are common.

Iris adopts this symmetric heap abstraction as its core memory model, recognizing its proven utility for multi-GPU communication. However, rather than wrapping existing xSHMEM implementations, Iris reimagines the programming interface with a modern, Pythonic API built natively in Triton. This approach preserves the conceptual benefits of symmetric memory while eliminating the legacy constraints of C-style interfaces and explicit thread-ID management that were inherited from CPU-era HPC programming models.

Table 1: Comparison of multi-GPU communication approaches.

Aspect	Built for Triton	Wrapper-Based
Libraries	Iris	Triton-Dist., PyTorch
Approach	Native Triton	Wraps xSHMEM
Programming Model	Triton-native	OpenSHMEM-based
Compiler Visibility	Full, co-optimization	Opaque, limited
API Style	Pythonic, tile-based	C-style, Python-wrapped
Language	Python + Triton	Python + xSHMEM
Memory Model	C++/HIP model	Ill-defined
Synchronization	Acquire/release semantics	`quiet/wait_until`

2.2 Related Work

A number of systems provide abstractions for multi-GPU communication, symmetric memory, and fused compute-communication execution models. Table 1 contrasts Iris with existing approaches, highlighting two fundamental implementation strategies: native implementations architected for the target programming model versus wrapper-based approaches that integrate external libraries. Below, we discuss the most closely related efforts in detail.

2.2.1 HIP/CUDA/C++-Based Approaches.

xSHMEM Libraries. As discussed in Section 2.1.4, xSHMEM libraries (NVSHMEM, rocSHMEM) implement the OpenSHMEM specification [17] and establish the symmetric memory abstraction that Iris builds upon. However, their APIs do not align well with Triton's tile-centric programming paradigm: they require explicit thread-ID management and rely on low-level C-style interfaces. These abstractions were originally built for HPC-CPU environments and later ported to GPU, which limits their effectiveness for modern GPU programming models. Iris retains the symmetric heap abstraction but provides a modern, Pythonic API that integrates naturally with Triton's tile-based execution model.

Flux. Chang et al. [6] introduced Flux, which targets symmetric memory communication but is implemented directly in CUDA and CUTLASS. While offering highly optimized kernels, this approach requires longer development cycles and relies heavily on C++ template metaprogramming. Iris maintains a simpler Python- and Triton-based design that enables faster prototyping, easier debugging, and full compiler visibility.

2.2.2 Triton-Based Wrappers.

Triton-Distributed. Zheng et al. [28] introduced Triton-Distributed, which wraps existing xSHMEM libraries behind Triton-specific Python APIs, introducing proof-of-concept fused kernels for GEMM and AllGather and MoE-style patterns. While similar in motivation to Iris, it inherits the limitations of its underlying SHMEM

implementation: the communication layer remains a thin wrapper around vendor libraries, preventing compiler-level visibility. In contrast, Iris implements remote memory operations directly in Triton with no external dependencies, enabling full compiler visibility for fine-grained fused operations.

PyTorch Symmetric Memory and TorchTitan AsyncTP. Recent PyTorch efforts [20] introduce symmetric memory support at the framework level, enabling asynchronous tensor-parallel communication via decomposed point-to-point operations (`put`/`get`). TorchTitan leverages this for fused GEMM-AllGather and MoE patterns optimized via `torch.compile`. However, similar to Triton-Distributed, these systems rely on vendor-supplied communication backends, limiting compiler optimization opportunities. Iris draws inspiration from the decomposed pattern approach but exposes primitives directly within Triton kernels, enabling tile-level fused kernels and eliminating kernel-switching overhead while avoiding reliance on external communication libraries.

3 Iris

Iris is a multi-GPU library built from scratch for scaling with minimal dependencies (only Triton, PyTorch, and HIP runtime). The library provides intuitive and simple APIs for developers without requiring knowledge of distributed systems architecture, enabling Python and Triton developers to write multi-GPU code leveraging high-level language abstractions. First we will discuss the design decisions that influenced Iris's design and successfully resulted in an abstraction that allows Triton developers to be productive and use familiar abstractions.

3.1 Design Philosophy

Iris adopts the symmetric heap abstraction from SHMEM (as discussed in Section 2.1.4) but modernizes the programming model for GPU computing. Rather than porting legacy SHMEM APIs, we provide Pythonic and Triton-native interfaces that respect modern programming paradigms. As summarized in Table 1, this distinguishes Iris from wrapper-based approaches that rely on existing xSHMEM implementations.

3.1.1 Adoption of Symmetric Heap Abstraction. Iris implements a Symmetric Heap abstraction. While Iris deviates from SHMEM-like library APIs, it implements one of the core ideas adopted in the OpenSHMEM specification: the symmetric heap. Symmetric heaps are simple to understand, implement, and use. The symmetric heap design provides predictable memory layouts across all GPUs, enabling efficient pointer translation with minimal overhead since each rank's memory has an identical structure at corresponding offsets.

3.1.2 Familiar Memory Model and Pythonic APIs. Rather than introducing new synchronization primitives or memory semantics, Iris adopts the well-established C++/HIP/CUDA memory model with acquire/release ordering (as detailed in Section 2.1.3). This design choice is intentional: GPU programmers already reason about memory consistency, ordering, and synchronization in their single-GPU kernels. By reusing these familiar semantics for multi-GPU communication, Iris eliminates the need to learn a new memory model.

Both HIP/HSA [9] and CUDA [15] programming models define clear semantics for atomic operations with configurable memory ordering (relaxed, acquire, release, acquire-release) and synchronization scopes (block, GPU, system). These well-defined scopes allow developers to precisely control the visibility and ordering of memory operations across different granularities—from thread block synchronization to system-wide coherence across GPUs. This familiarity extends to both host-side APIs (PyTorch-compatible tensor operations) and device-side APIs (Triton-native operations), which we detail in subsequent sections.

3.1.3 Pure Python and Triton Implementation. A key distinguishing feature of Iris is its implementation: the entire framework is built from scratch in Python and Triton without requiring external communication libraries or custom runtime dependencies. Unlike wrapper-based approaches such as Triton-Distributed (discussed in Section 2.2) that rely on rocSHMEM bytecode or other low-level communication primitives, Iris leverages only the existing PyTorch ecosystem (for host-side operations) and HIP runtime APIs (for GPU IPC, as described in Section 2.1.2). This design choice provides several advantages: (1) portability across different GPU vendors without vendor-specific communication libraries, (2) ease of debugging and modification since the entire codebase is in high-level Python and Triton, (3) simplified deployment with no additional system dependencies beyond PyTorch and the standard HIP/CUDA runtime, and (4) compiler visibility—the Triton compiler has full visibility into the entire codebase, enabling optimizations across computation and communication boundaries rather than treating communication primitives as opaque binary blobs linked into the final executable.

3.1.4 Value- and Pointer-based APIs. Existing SHMEM-based APIs treat the source and destination arguments to a SHMEM function as buffers that are pointed to by a pointer, along with their respective buffer sizes. CPU threads are more *heavyweight* and typically work on buffers of data, which likely influenced this design choice. However, Iris targets GPUs where hundreds of thousands of threads are actively doing work. Massively-parallel GPUs require both value-based and pointer-based operations. Value-based data movement copies data directly from registers to other GPUs. In contrast, pointer-based data movement acts as a data copy between the main memory of local GPUs and that of another GPU.

The rise of tile-based programming frameworks such as Triton, ThunderKittens [25], and CuTe DSL [14] demonstrates the importance of value-based APIs that directly operate on tensors rather than raw bytes. These frameworks prioritize high-level tensor abstractions because they align with how developers reason about computation and data movement in modern GPU programming. Iris's value-based APIs enable developers to express operations at the granularity of computational tiles, moving partial results directly from registers to remote memory without intermediate buffering. This approach is particularly effective for fine-grained computation-communication overlap patterns where data becomes available incrementally during computation. We will provide examples that leverage both value-based and pointer-based APIs in Section 4.

3.2 Host-Side APIs

Iris provides Pythonic PyTorch-like host APIs organized into several categories: initialization, memory management, rank/world queries, host-side communication, and tensor construction. Iris implements a full symmetric heap, where each allocation returns a PyTorch tensor that wraps the allocated virtual memory address range. We organize the discussion into three main areas: core infrastructure setup, distributed operations, and tensor management. Table 2 at the end provides a quick reference summary of all host-side APIs.

3.2.1 *Core Infrastructure.*

Constructor and Initialization. The initialization process follows several key steps: (1) PyTorch Distributed initialization and rank assignment, (2) GPU device selection based on rank, (3) symmetric heap initialization on the selected device, (4) IPC handle creation and exchange across all ranks using PyTorch Distributed all-gather operations, (5) opening of remote IPC handles to establish cross-GPU memory access, and (6) creation of a tensor containing all heap base addresses for device-side translation. This setup enables seamless remote memory access through the translate function, which converts local pointers to remote addresses by computing offsets and applying them to destination heap bases. Iris sets the GPU device and treats each single GPU as its own rank in the distributed communicator[2].

3.2.2 *Distributed Operations.*

Rank and Device Queries. Iris provides several query functions for distributed computing context. The `get_rank()` method returns the current process's rank ID in the distributed communicator, while `get_num_ranks()` returns the total number of ranks (world size). The `get_heap_bases()` method returns a tensor containing symmetric heap base addresses for all ranks, which is essential for device-side pointer translation.

Host-side Communication. Iris provides two primary communication primitives. The `barrier()` function synchronizes all ranks across the entire system by first calling `torch.cuda.synchronize()` (or `stream.synchronize()` if a stream is specified) to ensure the local GPU has finished all queued work, then performing a global distributed barrier so all ranks reach the same point before proceeding. The `broadcast()` function broadcasts a value from one rank to all others, automatically detecting the value type and using the appropriate mechanism: for tensors and arrays it uses efficient PyTorch distributed tensor collectives, while for scalars and other objects it uses object broadcast. This intelligent type detection makes broadcasting seamless across different data types.

3.2.3 *Tensor Management.*

Tensor Construction. Since remote memory operations require symmetric heap allocation, Iris provides PyTorch-compatible functions for tensor creation and initialization. The library supports three categories of functions (see Table 2 for the complete list):

- *Creation functions*: `zeros`, `ones`, `empty`, `full`, `zeros_like` for basic tensor initialization

[2]To simplify distributed programming, typically, each GPU is treated as its own rank rather than a single compute node.

```python
@triton.jit
def load(pointer, to_rank, from_rank, heap_bases, mask=None):
    translated_ptr = __translate(pointer, to_rank, from_rank,
    ↪  heap_bases)
    result = tl.load(translated_ptr, mask=mask)
    return result

@triton.jit
def __translate(ptr, from_rank, to_rank, heap_bases):
    from_base = tl.load(heap_bases + from_rank)
    to_base = tl.load(heap_bases + to_rank)
    ptr_int = tl.cast(ptr, tl.uint64)
    offset = ptr_int - from_base
    to_base_byte = tl.cast(to_base, tl.pointer_type(tl.int8))
    translated_ptr_byte = to_base_byte + offset
    translated_ptr = tl.cast(translated_ptr_byte, ptr.dtype)
    return translated_ptr
```

Listing 1: Iris load and pointer translation implementation.

- *Range functions*: `arange` and `linspace` for generating sequences
- *Random functions*: `rand`, `randn`, `randint`, `uniform` for sampling from various distributions

All functions support standard PyTorch parameters (`dtype`, `device`, `requires_grad`), providing drop-in compatibility with existing PyTorch code. The key innovation is that all allocated tensors reside in the symmetric heap, enabling direct remote GPU access through Iris's device-side APIs.

Table 2: Iris Host-Side API Summary.

Category	Function	Description
Core	`__init__()`	Initialize Iris runtime, setup symmetric heap, exchange IPC handles
	`barrier()`	Synchronize all ranks (GPU sync + distributed barrier)
	`broadcast()`	Broadcast tensor or scalar from one rank to all others
	`get_rank()`	Return current process rank ID
	`get_num_ranks()`	Return total number of ranks (world size)
	`get_heap_bases()`	Return tensor containing all symmetric heap base addresses
Tensor Creation	`zeros()`	Create tensor filled with zeros in symmetric heap
	`ones()`	Create tensor filled with ones in symmetric heap
	`empty()`	Create uninitialized tensor in symmetric heap
	`full()`	Create tensor filled with specified value
	`zeros_like()`	Create zeros tensor matching input tensor's shape
	`rand()`	Create tensor with uniform random values [0, 1)
	`randn()`	Create tensor with normal distribution (mean=0, std=1)
	`randint()`	Create tensor with random integers in range
	`uniform()`	Create tensor with uniform random values in [low, high)
Sequences	`arange()`	Create 1D tensor with evenly spaced values
	`linspace()`	Create 1D tensor with linearly spaced values

Table 3: Iris Device-Side API Summary

Category	Function	Description
Memory Operations	`load()`	Load value from remote rank's memory (value-based)
	`store()`	Store value to remote rank's memory (value-based)
	`get()`	Copy from remote memory to local memory (pointer-based)
	`put()`	Copy from local memory to remote memory (pointer-based)
	`copy()`	Copy between any two ranks (pointer-based)
Atomics	`atomic_add()`	Atomically add value to remote memory location
	`atomic_xchg()`	Atomically swap value with remote memory location
	`atomic_cas()`	Atomically compare and conditionally swap if values match
	`atomic_and()`	Atomically perform bitwise AND on remote memory
	`atomic_or()`	Atomically perform bitwise OR on remote memory
	`atomic_xor()`	Atomically perform bitwise XOR on remote memory
	`atomic_min()`	Atomically compute minimum with remote memory
	`atomic_max()`	Atomically compute maximum with remote memory
Translation	`__translate()`	Internal function for pointer translation

Note: All functions require `heap_bases` parameter. Atomics support `sem` (relaxed/acquire/release/acq_rel) and `scope` (block/gpu/sys).

3.3 Device-Side APIs

Iris provides Pythonic Triton-style device APIs for remote memory access and atomic operations. Since Triton doesn't support object-oriented programming, all functions require passing the symmetric heap pointer obtained via the `get_heap_bases` API. All device-side operations follow a consistent two-step pattern: (1) pointer translation from local to remote address space, and (2) memory operation on the translated pointer. Table 3 summarizes all device-side APIs.

3.3.1 Pointer Translation Mechanism. The core of Iris's remote memory access is the `__translate` function, which enables seamless access to remote memory without requiring explicit memory management from the programmer. The translation process (illustrated in Figure 2) works as follows:

(1) Compute the offset of the pointer within the local rank's symmetric heap
(2) Add this offset to the target rank's heap base address
(3) Cast the result back to the appropriate pointer type

In the current version of Iris supporting intra-node communication via IPC, the remote operation can be directly performed on the computed remote pointer using standard memory operations (e.g., `tl.load`). While the implementation loads the `heap_bases` on each call, experimental results show that these loads have no overhead, likely because the heap bases array (64 bytes) remains cached at the L1 level. Listing 1 shows the implementation of `load` and `__translate`, illustrating the two-step translation-then-operation pattern.

3.3.2 Memory Operations. Iris provides both value-based and pointer-based memory operations (see Table 3):

- *Value-based operations*: `load()` and `store()` move data directly between registers and remote memory, enabling fine-grained register-to-memory transfers
- *Pointer-based operations*: `get()`, `put()`, and `copy()` perform bulk transfers between memory regions, operating on buffer-to-buffer copies

All memory operations are non-blocking and use relaxed memory ordering by default. The choice between value-based and pointer-based operations depends on the communication pattern and data granularity requirements (discussed further in Section 4).

3.3.3 Atomic Operations and Memory Model. Iris provides the complete HIP/CUDA memory model semantics for atomic operations. The library supports three synchronization scopes:

- `block`: synchronization visible within a thread block (CTA)
- `gpu`: synchronization visible across the entire GPU
- `sys`: synchronization visible system-wide across all GPUs

Memory ordering options include `relaxed`, `acquire`, `release`, and `acq_rel`, enabling fine-grained control over synchronization semantics. The atomic operations include arithmetic (`add`), bitwise (`and`, `or`, `xor`), comparison (`min`, `max`), and exchange (`xchg`, `cas`) operations. All atomics follow the same pattern: translate the pointer, then perform the atomic operation with specified memory ordering and scope.

3.3.4 Gluon Backend. Iris also provides a Gluon backend that uses Triton's `@gluon.jit` decorator and `@aggregate` to encapsulate backend state, eliminating the need to pass `heap_bases` manually. This backend offers improved ergonomics by encapsulating the Iris device context in an aggregate type. For example, instead of passing `heap_bases` explicitly (standard API: `iris.load(buffer, to_rank, from_rank, heap_bases)`), the Gluon API encapsulates this in a context object (`ctx.load(buffer, from_rank=1)`). However, we focus this paper on the standard Triton API for clarity and broader compatibility.

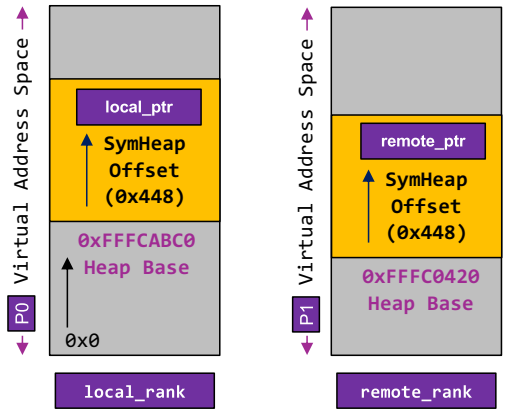


Figure 2: Pointer translation mechanism in Iris showing how local pointers are converted to remote addresses through offset computation.

3.4 Iris Features and More

Tile-based APIs and programming model for communication. Iris provides a tile-based communication model where operations are organized into tiles (e.g., `BLOCK_SIZE_M`, `BLOCK_SIZE_N`) that naturally fit within cache hierarchies. This tile-based approach enables building larger tiles within L1, L2, and LLC caches on chiplet-based architectures like MI300X and MI350X. Communication operations work at tile granularity, allowing fine-grained overlap where tiles can be communicated as soon as they are produced, rather than waiting for entire computation phases to complete. The tile-based model seamlessly integrates with Triton's blocked tensor operations, providing a unified programming model for both local computation and remote communication.

Ease of instrumentation and profiling granularity within the kernel and library. Since Iris is implemented entirely in Triton, profiling and instrumentation (e.g., through Triton's official profiler Proton) can be performed at fine-grained granularity both within user kernels and inside the Iris library itself. Developers can instrument specific communication operations, measure overlap efficiency, and analyze performance at the workgroup or even instruction level—not just at the boundaries of library calls, but deep within the library's implementation. This enables debugging and performance analysis of the pointer translation mechanism, remote memory operations, and synchronization primitives. This contrasts sharply with wrapped libraries where only opaque function call boundaries are visible, making it impossible to understand the performance characteristics of individual communication operations within a fused kernel or to diagnose issues inside the library itself.

L1-, L2-, LLC-cache aware programming using swizzling and cache modifiers. Iris provides explicit control over cache behavior through two complementary mechanisms. First, **cache modifiers** on load/store operations (e.g., `cache_modifier=".wt"` for write-through) allow direct control over how data is written to memory hierarchy, enabling optimization for chiplet architectures where cache coherence across XCDs (Accelerator Complex Die) is critical. Second, **swizzling** is implemented at multiple levels: (1) across XCDs using `chiplet_swizzle` to map work-groups to specific XCDs grouping tiles together for better Last-Level Cache (LLC) locality; and (2) spatial swizzling using `GROUP_SIZE_M` for L2-cache locality within tiles. This multi-level swizzling strategy, combined with cache modifiers, allows building larger tiles that efficiently utilize the entire cache hierarchy, from L1 caches within individual compute units to the LLC shared across XCDs.

4 Building Complex Multi-GPU Patterns

Iris enables sophisticated distributed algorithms through a simple yet powerful API. To illustrate this, we present a taxonomy of fused and unfused compute-communication patterns, using General Matrix Multiplication (GEMM) and all-scatter as a case study.

GEMM is the foundational building block of modern GPU workloads, from deep learning to scientific simulation, responsible for most of the floating-point operations in large models. All-scatter, on the other hand, is a collective communication primitive where each GPU (or rank) distributes distinct portions of its data to all other GPUs. Together, they represent a common and challenging pattern: local compute producing partial results (via GEMM) that must be rapidly exchanged across devices (via all-scatter) to form the complete global output.

```python
@triton.jit
def gemm_loop(A, B, C, ...):
    # Tile coordinate calculation removed for brevity
    # Memory layout setup
    rm = (pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)) % M
    rn = (pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)) % N
    rk = tl.arange(0, BLOCK_SIZE_K)
    A_BASE = A + rm[:, None] * stride_am + rk[None, :] * stride_ak
    B_BASE = B + rk[:, None] * stride_bk + rn[None, :] * stride_bn

    # Initialize accumulator registers
    acc = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)

    # GEMM's Main loop
    for k in range( tl.cdiv(K, BLOCK_SIZE_K)):
        a = tl.load(tl.multiple_of(A_BASE, (1, 16)))
        b = tl.load(tl.multiple_of(B_BASE, (16, 1)))
        acc += tl.dot(a, b)
        A_BASE += BLOCK_SIZE_K * stride_ak
        B_BASE += BLOCK_SIZE_K * stride_bk

    # Non-even K handling removed for brevity
    ...

    # Accumulator registers with C results
    return acc.to(C.type.element_ty)
```

Listing 2: A Triton GEMM main-loop routine, repurposed for several algorithms explained in the paper.

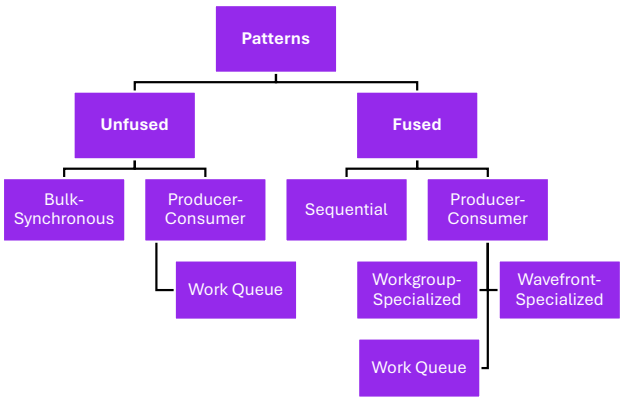


Figure 3: Taxonomy of unfused and fused computation and communication overlap patterns.

With Iris, these patterns can be expressed naturally within Triton, developers can write kernels that overlap GEMM computation with communication, eliminating execution "bubbles"[3]. This overlap is notoriously difficult to achieve in practice, yet Iris makes it straightforward through intuitive APIs that integrate remote memory operations directly into the Triton programming model.

In contrast to the traditional complexity of building fused kernels, Iris enables developers to construct multi-GPU pipelines that are both efficient and maintainable. As we show next, this provides a

[3]Bubbles refer to idle pipeline stages where no useful work occurs, often due to kernel launch overhead or synchronization delays.

practical taxonomy of compute-communication overlap strategies captured in Figure 3.

We organize the patterns into two main categories: unfused patterns where computation and communication execute in separate kernels, and fused patterns where both operations are combined within a single kernel. Each category offers different trade-offs between implementation complexity, resource utilization, and performance characteristics.

4.1 Unfused Patterns

Unfused patterns separate computation and communication into distinct kernels, providing clear boundaries between operations. We begin with the simplest approach and progress to more sophisticated producer-consumer strategies.

4.1.1 Bulk-Synchronous. The simplest approach to coordinating compute and communication is the bulk-synchronous pattern, where operations execute sequentially with explicit synchronization barriers between kernels. In this pattern, the GEMM kernel first completes all computation and stores results to local GPU memory, and only after the entire GEMM kernel finishes and synchronizes does the all-scatter kernel begin, reading from local memory and distributing data to remote GPUs using `iris.put`. Listing 3 demonstrates this pattern: two separate kernels are launched sequentially on the same stream, establishing a strict data dependency that ensures the GEMM kernel fully completes before any communication begins.

This approach offers the benefit of simplicity and clear separation of concerns—each kernel has a single, well-defined responsibility. However, it introduces significant execution "bubbles" as shown in Figure 4: the GPU must wait for all GEMM work to complete and all workgroups to synchronize before any communication can proceed, leaving computational resources idle during the synchronization barrier. The pattern also requires intermediate writes to global memory, as the GEMM results must be stored before the all-scatter kernel can read them, adding memory bandwidth overhead that could be avoided with more sophisticated overlap strategies.

```python
@triton.jit()
def gemm(
    A, B, C, ...
):
    pid = tl.program_id(0)
    for tile_id in range(pid, total_tiles, NUM_SMS):
        c = gemm_loop(A, B, C)
        ...
        # Store to local GPU's memory
        tl.store(C + offset, c, mask=mask, cache_modifier=".wt")

@triton.jit()
def all_scatter(C, ...):
    pid = tl.program_id(0)
    for tile_id in range(pid, total_tiles, NUM_SMS):

        # Begin: See the if segment for explanation:
        rm = (pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)) % M
        rn = (pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)) % N
        rm = tl.max_contiguous(tl.multiple_of(rm, BLOCK_SIZE_M),
        ↪  BLOCK_SIZE_M)
        rn = tl.max_contiguous(tl.multiple_of(rn, BLOCK_SIZE_N),
        ↪  BLOCK_SIZE_N)
        mask = (rm[:, None] < M) & (rn[None, :] < N)
        offset = rm[:, None] * stride_cm_global + (rn[None, :] +
        ↪  cur_rank * N) * stride_cn_global
        # End: masks/offset calculations.

        # Store from local to all other GPU's memory
        for remote_rank in range(world_size):
            if remote_rank != cur_rank:
                iris.put(C + offset, C + offset,
                cur_rank, remote_rank, heap_bases, mask=mask)

# On a single stream launch both kernels,
# establishing dependency of the two operations.
with torch.cuda.stream(main_stream):
    gemm[(num_sms,)](A, B, C, ...)
with torch.cuda.stream(main_stream):
    all_scatter[(num_sms,)](C, ...)
```

Listing 3: Iris: Unfused, Bulk Synchronous – illustrates the use of `iris.put` in a separate kernel after the GEMM kernel concludes and synchronizes.

4.1.2 Producer-Consumer (Stream Concurrency). Building on the bulk-synchronous pattern, the unfused producer-consumer approach achieves overlap by launching two separate kernels on asynchronous streams with explicit resource partitioning. Unlike bulk-synchronous execution where each kernel uses the entire GPU sequentially, this pattern limits the number of compute units (CUs) or streaming multiprocessors (SMs) allocated to each kernel. One kernel (the producer) uses a subset of CUs to perform GEMM computation, while another kernel (the consumer) uses the remaining CUs to perform communication. Dependencies between the kernels are managed through atomic-based synchronization primitives (similar to Listing 5), but instead of using an if/else statement within a single fused kernel, the two operations execute as separate kernels on different streams. This approach enables concurrent execution of computation and communication while maintaining explicit control over resource allocation.

For example, on an MI300X with 304 compute units, the producer kernel might be launched with 256 SMs to handle GEMM tiles, while the consumer kernel runs concurrently with the remaining 48 SMs to perform all-scatter communication. The producer kernel writes completed tiles to local memory and signals their availability using atomic operations (e.g., `tl.atomic_cas` with release semantics). The consumer kernel, executing concurrently on a separate stream, spins on these atomic locks (with acquire semantics) and immediately begins scattering tiles to remote GPUs as they become available. This pattern offers the modularity benefits of separate kernels while achieving overlap through hardware concurrency, though it requires careful tuning of the CU partition to balance computation and communication workloads.

4.2 Fused Patterns

While unfused patterns provide simplicity and modularity, fused patterns offer superior performance by eliminating kernel launch overhead and enabling fine-grained computation-communication overlap within a single kernel. These patterns leverage Iris's native Triton implementation to seamlessly interleave operations at tile granularity.

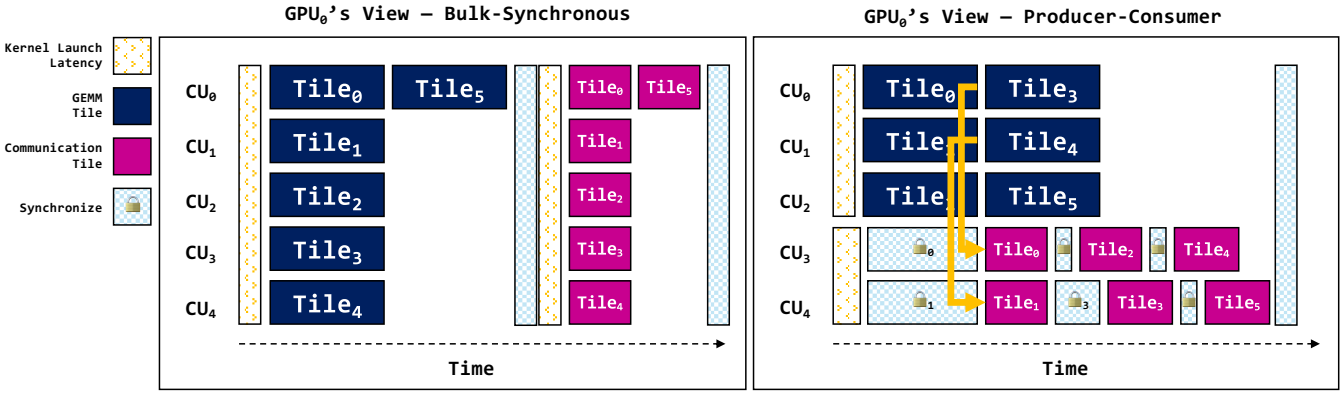


Figure 4: Timeline: Illustrates a single GPU's view of the taxonomy of unfused computation and communication patterns. (left) bulk-synchronous highlights the hard synchronization barriers that exist after each kernel, and (right) a multi-kernel producer-consumer pattern shows how overlap can be achieved by moving the synchronization at a finer granularity and partitioning the compute units (CUs) between computation and communication workers.

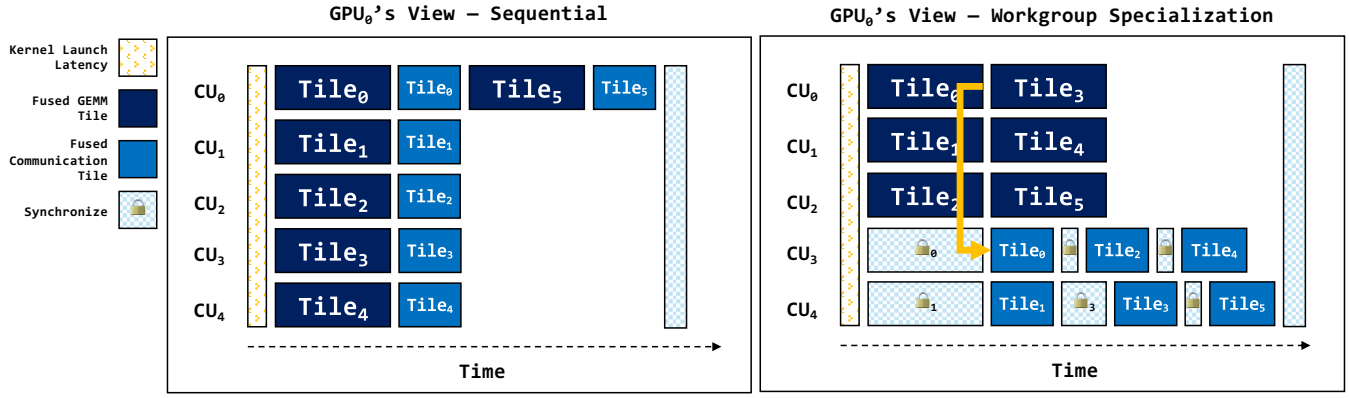


Figure 5: Timeline: Illustrates a single GPU's view of the taxonomy of fused GEMM and All-Scatter patterns.

4.2.1 Sequential. To bring more control of scheduling the work in the hands of developers, we can fuse multiple operations together in a mega or uber kernel [8, 23] and reduce the overhead of tearing down and recreating the kernels. This approach significantly reduces the "bubbles" in the total workload by moving to a fine-grained synchronization approach at a tile granularity. One such way fused kernels are implemented is following the data-dependencies that inherently exist within those operators, for example, we insert the all-scatter operator sequentially after the computation of each output GEMM tile. Iris enables this pattern through its device-side APIs that operate directly within Triton kernels. Listing 4 illustrates how developers may use `iris.store` to immediately scatter the GEMM tile produced to all remote GPUs without the need for a bulk-synchronous kernel-level barrier (see Listing 3).

Such pattern has the benefit of operating on the data as soon as it is ready (such as all scatter's store on the accumulator registers) with no intermediate writes to global memory required. Fused operators, however, still retain the sequential dependencies of executing one operator, waiting for example, the GEMM to complete and issuing the next operator in the same kernel. The impact of tail latency (tail occupancy inefficiency) worsens, because now the last "wave" of work needs to process GEMM and all scatter before the kernel completes.

4.2.2 Producer-Consumer (Workgroup Specialization). One such way of avoiding the sequential issuance of the two operator is by using specialization techniques over the available compute resources. We can implement a persistent-style kernels (see the for-loop over all tiles in Listings 4) and specialize the type of computation each compute resource (e.g., workgroups) does by using the workgroup index (`pid` in Triton). Listing 5 shows an example of GEMM + All Scatter using Iris, where each workgroup gets mapped to a compute unit of an AMD GPU (MI300X has 304 compute units); the first 256 (0-255) workgroups are responsible for computing the GEMM output tile and signaling the other 48 workgroups (256-303) responsible for waiting for a tile to be produced (using for example a spin-lock), and then scatter the result to other GPUs. With this method, we can dedicate exact compute resources for various tasks—this is especially useful when workloads like GEMMs do not require the entire

```python
@triton.jit
def fused_gemm_all_scatter(
    A, B, C, ...
):
    pid = tl.program_id(0)
    num_pid_m, num_pid_n = tl.cdiv(M, BLOCK_SIZE_M), tl.cdiv(N,
    ↪  BLOCK_SIZE_N)
    total_tiles = num_pid_m * num_pid_n
    for tile_id in range(pid, total_tiles, NUM_SMS):
        c = gemm_loop(A, B, C)

        rm = (pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)) % M
        rn = (pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)) % N

        # Add compiler hints
        rm = tl.max_contiguous(tl.multiple_of(rm, BLOCK_SIZE_M),
        ↪  BLOCK_SIZE_M)
        rn = tl.max_contiguous(tl.multiple_of(rn, BLOCK_SIZE_N),
        ↪  BLOCK_SIZE_N)

        # Define the C-mask (BLOCK_SIZE_M, 1) x (1, BLOCK_SIZE_N)
        mask = (rm[:, None] < M) & (rn[None, :] < N)

        # Calculate the "global" offset of C based on the rank.
        # Note the N-dimension is being multiplied by current rank.
        # This is because each rank is computing a portion of the
        # N-dimension locally and then scattering it to all other
        # ranks to complete the global N-dimension.
        offset = rm[:, None] * stride_cm_global + (rn[None, :] +
        ↪  cur_rank * N) * stride_cn_global

        # Scatter to all ranks
        for remote_rank in range(world_size):
            iris.store(C + offset, c, cur_rank, remote_rank,
            ↪  heap_bases, mask=mask)
```

Listing 4: Iris: Fused, Sequential – illustrates the use of `iris.store` right after the GEMM tile is produced.

```python
@triton.jit()
def wg_specialized_gemm_all_scatter(
    A, B, C, locks, GEMM_SMS, COMM_SMS, ...
):
    pid = tl.program_id(0)
    if pid < GEMM_SMS:

        # Process all gemm tiles using GEMM_SMS number of
        # workgroups in a persistent fashion.
        for tile_id in range(pid, total_tiles, GEMM_SMS):
            c = gemm_loop(A, B, C)
            ...
            # Store to local GPU's memory
            tl.store(C + offset, c, mask=mask, cache_modifier=".wt")
            tl.atomic_cas(locks + tile_id, 0, 1, sem="release",
            ↪  scope="gpu")

    else:  # pid >= GEMM_SMS
        COMM_SMS = NUM_SMS - GEMM_SMS
        pid = pid - GEMM_SMS

        # Process all comm tiles using COMM_SMS number of
        # workgroups in a persistent fashion.
        for tile_id in range(pid, total_tiles, COMM_SMS):

            # Wait for the tile to be ready.
            while tl.atomic_cas(locks + tile_id, 1, 0, sem="acquire",
            ↪  scope="gpu") == 0:
                pass

            # Store from local to all other GPU's memory
            for remote_rank in range(world_size):
                if remote_rank != cur_rank:
                    iris.put(C + offset, C + offset,
                    cur_rank, remote_rank, heap_bases, mask=mask)

# Launch code:
with torch.cuda.stream(main_stream):
    wg_specialized_gemm_all_scatter[(num_sms,)](
        A, B, C, locks, GEMM_SMS, COMM_SMS, ...)
```

Listing 5: Iris: Fused, Workgroup Specialization – illustrates how a single fused kernel can be split into components where dedicated workgroups perform either communication or computation operations.

device to achieve peak performance. Fused workgroup specialization, however, just like all other fused kernels, requires worst-case resource allocation (i.e., an operation such as all-scatter is forced to occupy more resources than needed because it is fused with a more resource-intensive operation such as GEMM).

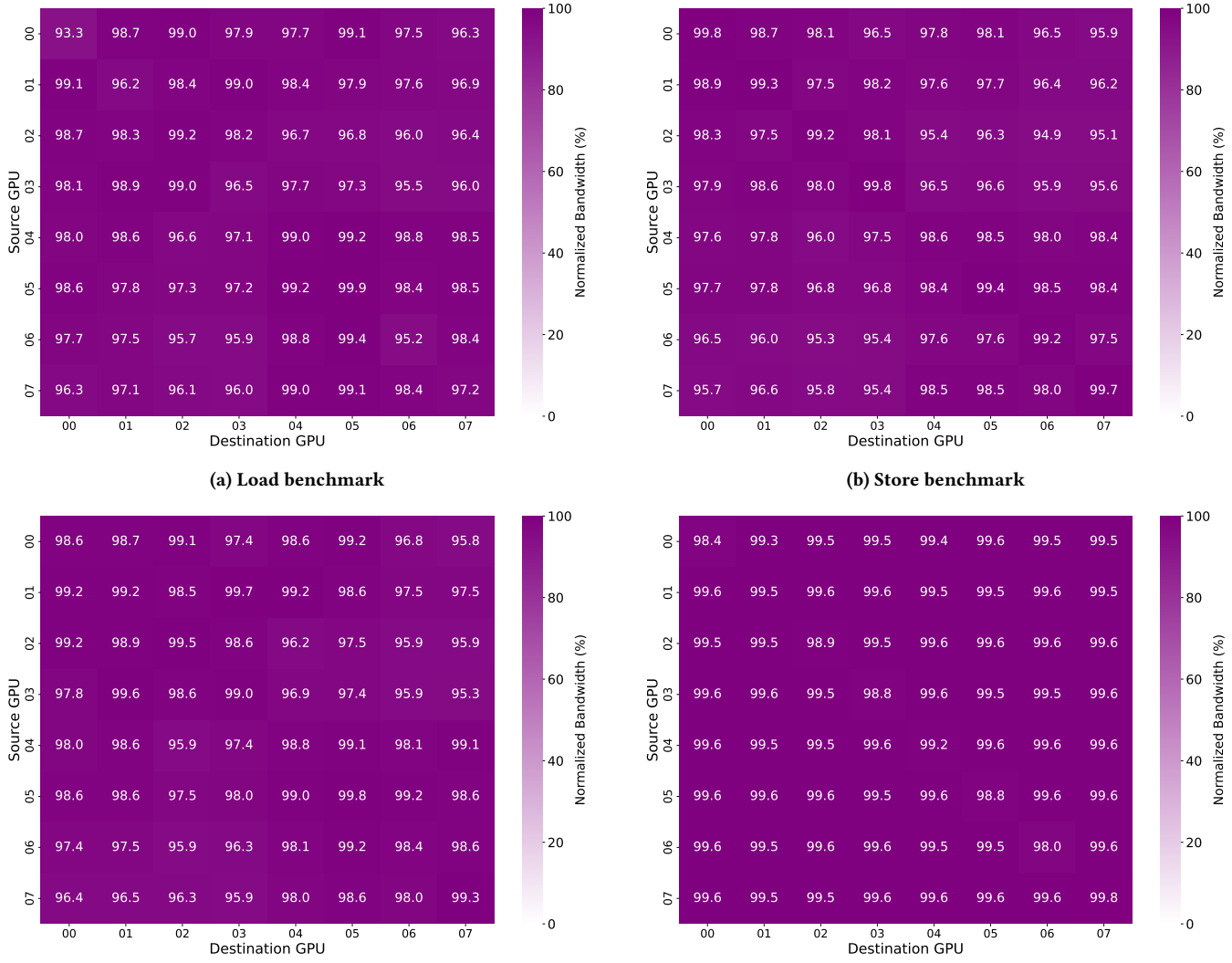
4.2.3 Producer-Consumer (Wave Specialization) and Work Queue. Similar to workgroup specialization, we can also split the work at a finer granularity of a wavefront (AMD GPU) or a warp (NVIDIA GPU), where 64 or 32 threads in a lockstep fashion work on issuing communication or processing compute. However, without using Gluon, this pattern is not typically suited for a more workgroup-centric language like Triton. Work queue on the other hand extends these patterns and moves the management of the work in a separate queue-like data structure, due to the synchronization cost of inserting and removing "work" (communication or computation tile), queues are also not well suited for a GPU architecture (or Triton language.) In this paper, we focus on all other patterns described in the previous subsections.

5 Results

We evaluate Iris on a system with 8x AMD Instinct™ MI300X GPUs configured under NPS1/SPX memory and compute partition modes [19] and ROCm 6.3.1. Our evaluation consists of two main components: microbenchmarks that characterize Iris's fundamental performance across point-to-point and collective communication primitives, and real-world application studies using the GEMM+All-Scatter fused patterns from Section 4. The microbenchmarks demonstrate that Iris achieves near-optimal bandwidth utilization across all operations, validating the efficiency of its native Triton implementation. For real-world workloads, Iris's fine-grained overlap capabilities enable significant performance improvements over the bulk-synchronous baseline, with speedups ranging from 0.93× to 1.79× (average 1.21×) compared to PyTorch and RCCL. These results highlight both the low overhead of Iris's abstraction and the substantial benefits of tile-granularity computation-communication overlap enabled by its design.

5.1 Microbenchmarks

Figure 6 presents performance benchmarks for point-to-point load, store, and atomic operations. All benchmarks are normalized relative to the achievable bandwidth [21, 22] on the system, with the heatmaps showing bandwidth percentages where darker shades indicate better (higher) performance. The results demonstrate Iris's efficiency in handling different types of remote memory access



(a) Load benchmark

(b) Store benchmark

(c) Atomic add

(d) Atomic xchg

Figure 6: Performance benchmarks for load, store, and atomic operations. The results demonstrate Iris's efficiency in handling different types of remote memory access patterns.

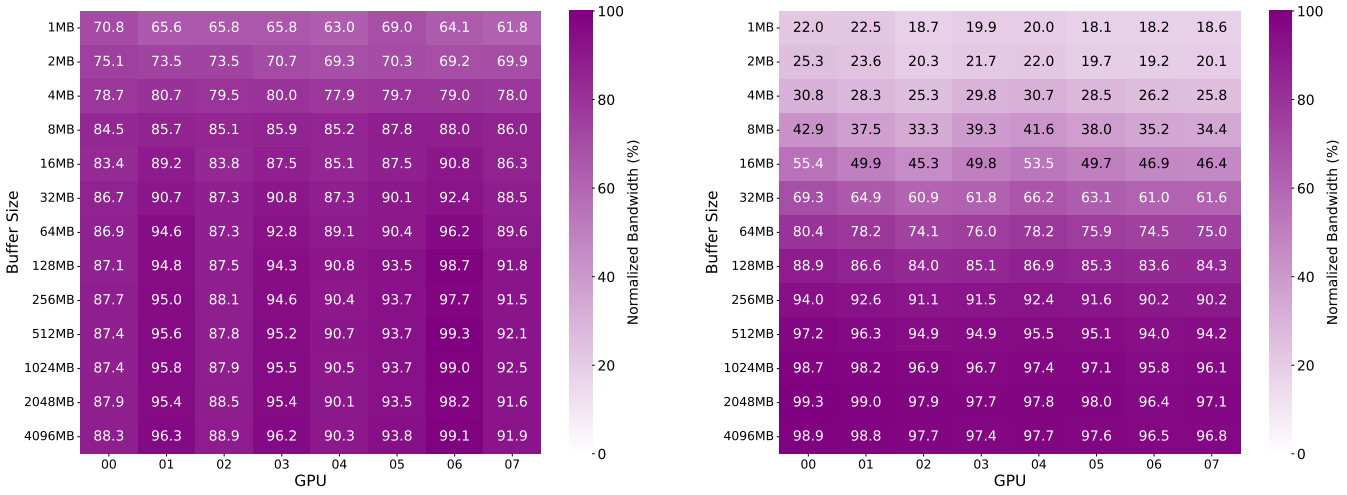
patterns, with consistent performance across operations, achieving near-optimal bandwidth utilization.

Figure 7 provides the all-load and all-store mircobenchmark results where all GPUs participate in the load and store operations across all links. In the benchmark, different buffer sizes are moved across all ranks at the same time. The heatmaps show the normalized bandwidth relative to achievable bandwidth where darker shades represent superior performance. As buffer size increases, performance improves significantly (as expected), reaching near-optimal achievable bandwidth utilization showcasing the efficiency of the Iris simple-yet-effective implementation and abstraction.

5.2 Evaluating Fused, Unfused Patterns Taxonomy

To evaluate Iris' in a real-world application, we continue our case-study from Section 4. We implemented many of the fused and unfused patterns described in the Figure 3 and Listings 3, 4 and 5 to capture the versatility of the abstraction and APIs. In this section, we cover a deep-dive of these patterns using PyTorch's `torch.matmul` and RCCL's All-Gather as a functionally equivalent baseline. Figure 8 shows the complete performance landscape across six different problem shapes and sizes with varying N and K-dimensions (and M=8192) for different number of GPUs (world size).

We first establish Iris' baseline using the "Unfused, Bulk-synchronous" schedule, which as a schedule is equivalent



(a) All-Load benchmark **(b) All-Store benchmark**

Figure 7: All load/all store benchmark results where all GPUs participate in all load/store operations across all links. As buffer size increases, performance approaches peak bandwidth utilization as expected.

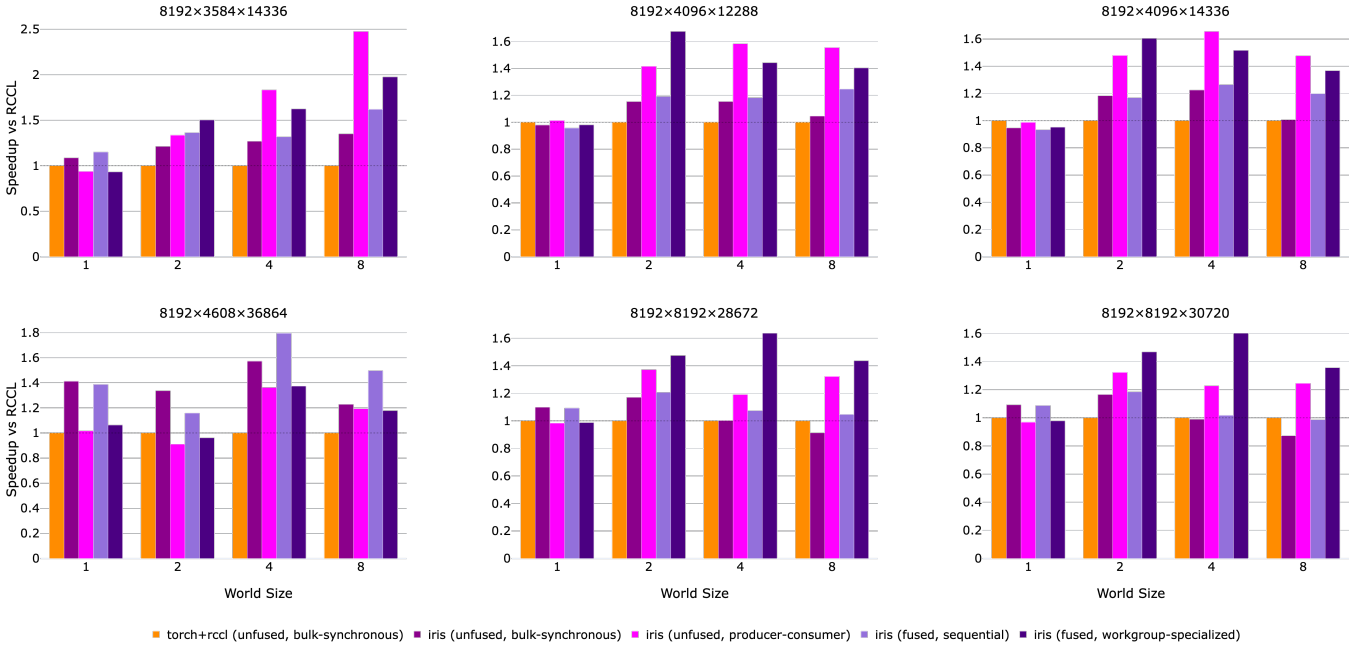


Figure 8: Complete performance comparison between Iris fused GEMM + All-Scatter and RCCL GEMM + AllGather kernels across different problem sizes and world sizes.

PyTorch and RCCL. We observe that Iris competes with state-of-the-art GEMM and All-Gather implementation — this further validates that Iris' abstraction isn't resulting in any discernible overheads. In some cases, such as $8192 \times 4608 \times 36864$, Iris is 20% faster using 8 GPUs. This is largely attributed to difference in heuristics for the PyTorch and RCCL's heuristics selecting a suboptimal configuration (see Figure 9).

Iris also allows to break the rigid bulk-synchronous programming model by using the device-side APIs and moving the synchronization at a fine grained tile-level granularity. "Unfused, producer-consumer", "fused, workgroup-specialization" and "fused, sequential" all follow this model. Unfused, producer-consumer approach gives up-to 2.5× speedup for problem shape of $8192 \times 3584 \times 14336$

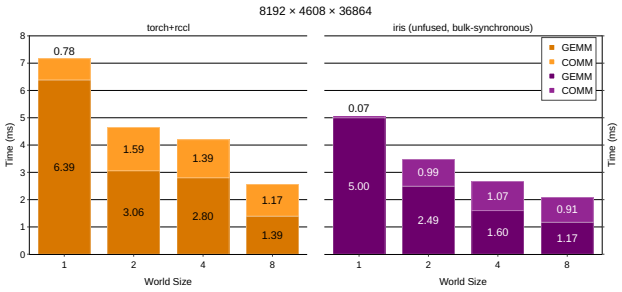


Figure 9: Deep-dive: Shows breakdown of GEMM (darker region) and Communication (lighter region) for Iris compared to PyTorch and RCCL for $8192 \times 4608 \times 36864$ matrix size. Note the slight speedups in both GEMM and communication due to Iris' flexibility to be able to cater to a specific problem shape and size. Tile-based abstraction allows for users to simply adjust the needed tile-size at compile-time per a problem/kernel granularity.

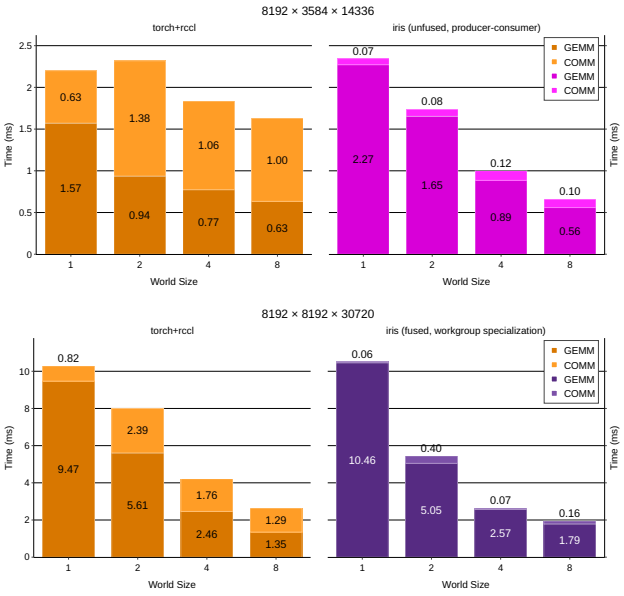


Figure 10: Deep-dive: Shows breakdown of GEMM (darker region) and Communication (lighter region) for Iris compared to PyTorch and RCCL for $8192 \times 3584 \times 14336$ matrix size. These two problems are specifically of interest for producer-consumer and workgroup specialization based schedules as they show the approach mostly hides the overhead of communication behind the GEMM (darker region) by splitting the available GPU's compute units between GEMM and communication.

on 8 GPUs. The nature of the problem (small-N after being split 8-ways, and large-K) allows producer-consumer-style model (unfused or fused using workgroup-specialization) to completely hide the communication operation behind the GEMM operation, this is illustrated in Figure 10. The difference between the fused and unfused variants of producer-consumer approach is that using unfused two kernels (one producer and one consumer), we avoid worst-case resource allocation[4] (typically bounded by GEMMs) and promotes better occupancy at the cost of kernel launch latency and less control over the scheduling of operations. Whereas in a fused variant, we only launch one kernel and do not have to pay an additional cost to launch the kernel, it allows for more scheduling control and re-purposing/reusing resources and data when relevant, however, requires worst-case resource allocation and limits the occupancy for one of the operator.

A fused, sequential schedule is the simplest of them all — essentially appending the communication operation at the end of the main-loop of the GEMM operation. This required a few lines of code changes as shown in Listing 4, and works well for problems that need more resources for GEMM (such as small N and massive large-K.) However, as the name suggest, this sort of schedule creates a sequential dependency between the GEMM operation and All-Scatter operation, and increases the tail latency of the entire problem. Potentially creating large bubbles in the last timestep of the problem (also illustrated in Figure 5.) With this schedule, Iris outperforms the baseline PyTorch and RCCL implementation by 1.8× for 4 GPUs and 1.5× for 8 GPUs on $8192 \times 4608 \times 36864$ problem size (see Figure 11).

Across all tested configurations, Iris achieves an average speedup of 1.21× over PyTorch and RCCL, with speedups ranging from 0.93× to 1.79×, highlighting the consistent performance benefits of Iris's design. These speedups are largely attributed to the flexibility of Iris' design to be able to implement a fused kernel with tile-granularity synchronization (and the resultant compute and communication

[4]Worst-case resource allocation: Size of the allocated shared-memory, number of VGPRs or number of threads launched is not bounded by the worst-case operation.

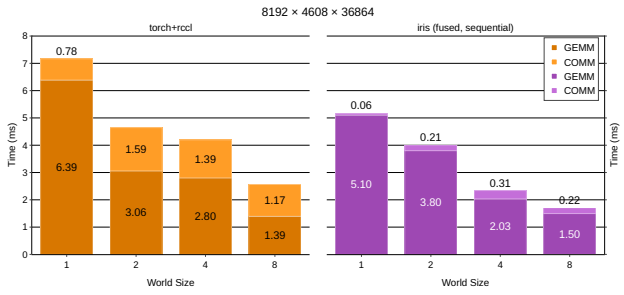


Figure 11: Deep-dive: Shows breakdown of GEMM (darker region) and Communication (lighter region) for Iris compared to PyTorch and RCCL for $8192 \times 4608 \times 36864$ matrix size. This shape illustrates when fused sequential approach benefits when the added communication is an overall small overhead (smaller output tile and really large K) and more resources can simply be allocated to processing GEMM.

overlap) versus a rigid, bulk-synchronous programming model of PyTorch's GEMM and RCCL's AllGather kernels.

6 Conclusion and Future Work

Iris's ability to match or exceed the performance of heavily-optimized HIP/CUDA-based libraries like RCCL despite its pure Triton implementation demonstrates that low-level native implementations are not a fundamental requirement for multi-GPU programming. The 1.79× peak speedup reflects a qualitatively different programming model where synchronization granularity shifts from kernel boundaries to tiles. Perhaps more significant is what Iris makes tractable: our taxonomy demonstrates that diverse overlap patterns—from bulk-synchronous to fused sequential to producer-consumer to workgroup specialization—can be implemented with minimal code changes, often requiring only a few additional lines within the same Triton kernel. Patterns that would demand substantial engineering effort with traditional CCLs (separate kernel implementations, complex host-side coordination, manual resource partitioning) emerge naturally in Iris using the same primitives Triton developers already use for single-GPU work.

This suggests that the real barrier to fine-grained overlap has been abstraction mismatch, not hardware capability. When communication primitives live in the same semantic space as computation (tile-based Triton), overlap patterns become straightforward extensions rather than heroic engineering efforts. Future work will focus on extending Iris to multi-node settings with RDMA, exploring additional fused patterns such as wave-specialization in Gluon and work queues, and investigating opportunities to offload Iris operations to the compiler itself, leveraging Triton's ability to optimize across the entire computation-communication pipeline.

Acknowledgments

This work was supported in part by Advanced Micro Devices, Inc. under the AMD AI & HPC Cluster Program. The authors would like to thank Karl Schulz, Lei Zhang, Ziyad AlBanoby, Octavian-Alexandru Trifan, David Sidler, Xiaohu Guo, Karthik Sangaiah, Lixun Zhang, Vinayak Gokhale, Panagiotis Mylonas, Eric Eaton, Aditya Nandakumar, Ahmed Eltantawy, Dimple Prajapati, Mike Chu, Mike Schulte, Ganesh Dasika, Brad Beckmann, Ralph Wittig, and Peng Sun for their continuous feedback, support and suggestions. AMD, the AMD Arrow logo, AMD CDNA™, AMD Instinct™, AMD ROCm™, AMD Infinity Cache™, AMD Infinity Fabric™, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

References

[1] Advanced Micro Devices, Inc. 2025. HIP Documentation (ROCm Software Future Release). https://rocm.docs.amd.com/projects/HIP/en/docs-develop/index.html. [Online; accessed 6-August-2025].

[2] Advanced Micro Devices, Inc. 2025. *Introducing AMD CDNA™ 3 Architecture.* Technical Report. Advanced Micro Devices, Inc. [Online; accessed 6-August-2025].

[3] Advanced Micro Devices, Inc. 2025. ROCm OpenSHMEM (rocSHMEM). https://github.com/ROCm/rocSHMEM. [Online; accessed 6-August-2025].

[4] AMD. 2025. *RCCL: ROCm Communication Collectives Library.* https://github.com/ROCm/rccl [Online; accessed 27-October-2025].

[5] Muhammad Awad, Muhammad Osama, and Brandon Potter. 2025. *Iris: First-Class Multi-GPU Programming Experience in Triton.* https://doi.org/10.5281/zenodo.17382307

[6] Li-Wen Chang, Wenlei Bao, Qi Hou, Chengquan Jiang, Ningxin Zheng, Yinmin Zhong, Xuanrun Zhang, Zuquan Song, Chengji Yao, Ziheng Jiang, Haibin Lin, Xin Jin, and Xin Liu. 2024. FLUX: Fast Software-Based Communication Overlap on GPUs Through Kernel Fusion. https://doi.org/10.48550/arXiv.2406.06858 arXiv:2406.06858 [cs.LG]

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. https://doi.org/10.48550/arXiv.1802.04799 arXiv:1802.04799 [cs.LG]

[8] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. 2025. DeepSeek-V3 Technical Report. https://doi.org/10.48550/arXiv.2412.19437 arXiv:2412.19437 [cs.CL]

[9] HSA Foundation. 2018. HSA Platform System Architecture Specification. https://hsafoundation.com/wp-content/uploads/2021/02/HSA-SysArch-1.2.pdf. [Online; accessed 2-November-2025].

[10] Andrew Kerr, Duane Merrill, Julien Demouth, and John Tran. 2017. CUTLASS: Fast Linear Algebra in CUDA C++. https://devblogs.nvidia.com/cutlass-linear-algebra-cuda/

[11] Wanchao Liang, Tianyu Liu, Less Wright, Will Constable, Andrew Gu, Chien-Chin Huang, Iris Zhang, Wei Feng, Howard Huang, Junjie Wang, Sanket Purandare, Gokul Nadathur, and Stratos Idreos. 2025. TorchTitan: One-stop PyTorch Native Solution for Production Ready LLM Pre-training. https://doi.org/10.48550/arXiv.2410.06511 arXiv:2410.06511 [cs.CL]

[12] LLVM Project. 2025. User Guide for the AMDGPU Backend: Memory Model. https://llvm.org/docs/AMDGPUUsage.html#memory-model. [Online; accessed 6-August-2025].

[13] NVIDIA. 2025. *NCCL: Optimized Primitives for Collective Multi-GPU Communication.* https://github.com/NVIDIA/nccl [Online; accessed 27-October-2025].

[14] NVIDIA Corporation. 2024. CUTLASS: CUDA Templates and Python DSLs for High-Performance Linear Algebra. https://github.com/NVIDIA/cutlass. Version 4.3.0. [Online; accessed 2-November-2025].

[15] NVIDIA Corporation. 2025. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/. [Online; accessed 7-August-2025].

[16] NVIDIA Corporation. 2025. NVSHMEM: OpenSHMEM-Based Parallel Programming Interface for NVIDIA GPUs. https://github.com/NVIDIA/nvshmem. [Online; accessed 6-November-2025].

[17] OpenSHMEM. 2012. OpenSHMEM Specification. http://openshmem.org/site/specification/. [Online; accessed 6-August-2025].

[18] OpenXLA Project. 2025. XLA: Accelerated Linear Algebra Compiler for GPUs, CPUs, and ML Accelerators. https://github.com/openxla/xla. [Online; accessed 6-November-2025].

[19] Muhammad Osama, Robert Swann, Krishnan Sangaiah, Saurabh Singh, Ganesh Dasika, and Rakesh Bhardwaj. 2025. Deep Dive into the MI300 Compute and Memory Partition Modes. https://rocm.blogs.amd.com/software-tools-optimization/compute-memory-modes/README.html. Accessed: 2025-08-10.

[20] PyTorch Foundation. 2025. PyTorch 2.9 Release Blog. https://pytorch.org/blog/pytorch-2.9-release/. [Online; accessed 6-November-2025].

[21] Ben Sander. 2025. Understanding Peak, Max-Achievable & Delivered FLOPs, Part 1. https://rocm.blogs.amd.com/software-tools-optimization/Understanding_Peak_and_Max-Achievable_FLOPS/README.html.

[22] Ben Sander, Evan Masters, Babak Poursartip, and Henry Ho. 2025. Measuring Max-Achievable FLOPs – Part 2. https://rocm.blogs.amd.com/software-tools-optimization/measuring-max-achievable-flops-part2/README.html.

[23] Benjamin Spector, Jordan Juravsky, Stuart Sul, Owen Dugan, Dylan Lim, Dan Fu, Simran Arora, and Chris Ré. 2025. Look Ma, No Bubbles! Designing a Low-Latency Megakernel for Llama-1B. https://hazyresearch.stanford.edu/blog/2025-05-27-no-bubbles. [Online; accessed 10-August-2025].

[24] Benjamin F. Spector, Simran Arora, Aaryan Singhal, Daniel Y. Fu, and Christopher Ré. 2024. ThunderKittens: Simple, Fast, and Adorable AI Kernels. https://doi.org/10.48550/arXiv.2410.20399 arXiv:2410.20399 [cs.LG]

[25] Benjamin F. Spector, Simran Arora, Aaryan Singhal, Daniel Y. Fu, and Christopher Ré. 2024. ThunderKittens: Simple, Fast, and Adorable AI Kernels. arXiv:2410.20399 [cs.LG] https://arxiv.org/abs/2410.20399

[26] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL '19) (MAPL 2019)*. 1–10. https://doi.org/10.1145/3315508.3329973

[27] Octavian Alexandru Trifan, Karthik Sangaiah, Muhammad Awad, Muhammad Osama, Sumanth Gudaparthi, Alexandru Nicolau, Alexander Veidenbaum, and Ganesh Dasika. 2025. Eliminating Multi-GPU Performance Taxes: A Systems Approach to Efficient Distributed LLMs. https://doi.org/10.48550/arXiv.2511.02168 arXiv:2511.02168 [cs.DC]

[28] Size Zheng, Wenlei Bao, Qi Hou, Xuegui Zheng, Jin Fang, Chenhui Huang, Tianqi Li, Haojie Duanmu, Renze Chen, Ruifan Xu, Yifan Guo, Ningxin Zheng, Ziheng Jiang, Xinyi Di, Dongyang Wang, Jianxi Ye, Haibin Lin, Li-Wen Chang, Liqiang Lu, Yun Liang, Jidong Zhai, and Xin Liu. 2025. Triton-Distributed: Programming Overlapping Kernels on Distributed AI Systems with the Triton Compiler. https://doi.org/10.48550/arXiv.2504.19442 arXiv:2504.19442 [cs.DC]