GPU-Initiated Networking for NCCL

Khaled Hamidouche, John Bachan, Pak Markthub, Peter-Jan Gootzen, Elena Agostini,
Sylvain Jeaugey, Aamir Shafi, Georgios Theodorakis, Manjunath Gorentla Venkata

{*khamidouche, jbachan, pmarkthub, pgootzen, eagostini, sjeaugey, ashafi, gtheodorakis, manjunath*}*@nvidia.com*
NVIDIA Corporation

***Abstract*—Modern AI workloads, especially Mixture-of-Experts (MoE) architectures, increasingly demand low-latency, fine-grained GPU-to-GPU communication with device-side control. Traditional GPU communication follows a host-initiated model, where the CPU orchestrates all communication operations—a characteristic of the CUDA runtime. Although robust for collective operations, applications requiring tight integration of computation and communication can benefit from device-initiated communication that eliminates CPU coordination overhead.**

NCCL 2.28 introduces the Device API with three operation modes: Load/Store Accessible (LSA) for NVLink/PCIe, Multimem for NVLink SHARP, and GPU-Initiated Networking (GIN) for network RDMA. This paper presents the GIN architecture, design, semantics, and highlights its impact on MoE communication. GIN builds on a three-layer architecture: i) NCCL Core host-side APIs for device communicator setup and collective memory window registration; ii) Device-side APIs for remote memory operations callable from CUDA kernels; and iii) A network plugin architecture with dual semantics (GPUDirect Async Kernel-Initiated and Proxy) for broad hardware support. The GPUDirect Async Kernel-Initiated backend leverages DOCA GPUNetIO for direct GPU-to-NIC communication, while the Proxy backend provides equivalent functionality via lock-free GPU-to-CPU queues over standard RDMA networks. We demonstrate GIN's practicality through integration with DeepEP, an MoE communication library. Comprehensive benchmarking shows that GIN provides device-initiated communication within NCCL's unified runtime, combining low-latency operations with NCCL's collective algorithms and production infrastructure.

***Index Terms*—NCCL Device API, GPU-Initiated Networking, Load/Store Accessible, Multimem, RDMA, One-Sided Communication, Device-Initiated Communication, DeepEP**

I. Introduction

The rapid development of large language models (LLMs) has introduced new performance demands for GPU communication libraries. Modern AI workloads require more than traditional collective communication: they require low-latency point-to-point operations for inference token generation [1], [2], custom communication patterns for Mixture-of-Experts (MoE) architectures [3], and close integration of computation and communication in compiler-generated kernels [4], [5]. These workloads benefit from *GPUs directly initiating and controlling network communication without CPU involvement*.

NCCL [6] has established itself as the de facto communication runtime for GPU-based machine learning, providing optimized collective algorithms and robust infrastructure for distributed training and inference. Traditional GPU communication follows a host-initiated model, where the CPU orchestrates all communication operations. This approach requires explicit host-device synchronization—a characteristic of the CUDA runtime model—and separate kernel launches for each communication call. While this model has proven robust for large-scale collective operations, applications requiring tight integration of computation and communication—such as dynamic token routing in MoE inference [3], compiler-generated communication in JAX [5] and Triton [4] kernels—require device-initiated communication that eliminates the CPU coordination overhead stemming from CUDA-based host-side synchronization. On the other hand, the NVSHMEM library [7], [8] has successfully demonstrated the viability and performance impact of GPUDirect Async Kernel-Initiated (GDAKI) capabilities, providing device primitives that enable communication and computation fusion for AI workloads.

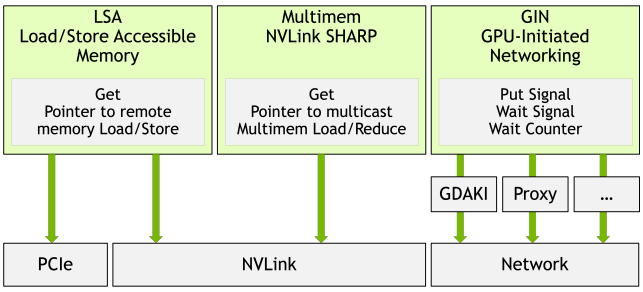

Fig. 1. NCCL Device API architecture showing three operation modes and their underlying interconnect technologies. Load/Store Accessible (LSA) uses PCIe and NVLink for intra-node memory operations, Multimem leverages NVLink SHARP for hardware multicast, and GPU-Initiated Networking (GIN) provides dual backend implementations (GDAKI and Proxy) for network-based communication over InfiniBand and RoCE.

To satisfy the tight integration of computation and communication—required by modern AI workloads—NCCL 2.28 introduces the **Device API** [9], enabling GPUs to initiate communication operations directly from within kernels. The Device API supports three operation modes for device-initiated communication (Figure 1): **Load/Store Accessible (LSA)** for intra-node communication over NVLink and PCIe using memory load/store operations, **Multimem** for hardware multicast via NVLink SHARP, and **GPU-Initiated Networking (GIN)** for inter-node communication over InfiniBand and RoCE networks. This paper focuses on **GIN**, which enables GPUs to initiate network operations directly within a GPU kernel. GIN

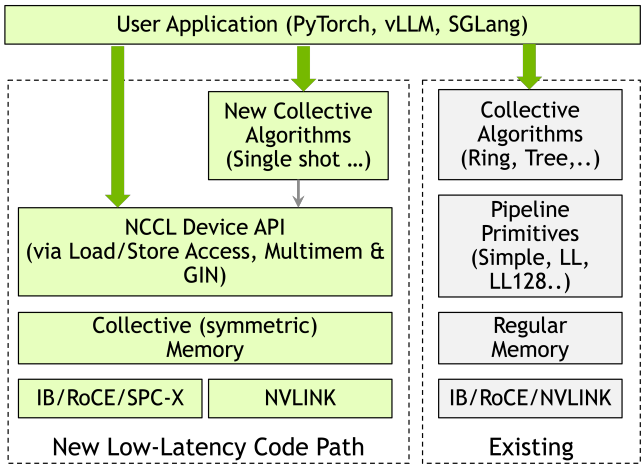

Fig. 2. High-level architecture comparison: NCCL Device API (left) with three operation modes (Load/Store Accessible for NVLink/PCIe, Multimem for NVLink SHARP, GIN for Network RDMA) enables single-shot collective algorithms with collective symmetric memory, while traditional NCCL (right) uses host-initiated algorithms with pipeline primitives over regular memory.

provides device-side APIs for one-sided operations, allowing CUDA kernels to perform remote memory operations, point-to-point synchronizations, and poll for completion entirely from device code. NCCL's Device API with its GIN capabilities provides AI workloads with low-latency primitives, fusion, and customization opportunities while enabling these applications to leverage NCCL's existing infrastructure, such as hierarchical communicators, elasticity, and fault-tolerance mechanisms for large-scale production deployments.

Figure 2 compares the NCCL Device API architecture with traditional host-initiated NCCL. Applications (PyTorch [10], TRT-LLM [1], vLLM [2], SGLang [11]) can either use NCCL-provided single-shot collective algorithms implemented with the Device API or directly invoke Device API primitives to implement custom communication patterns in GPU kernels. The Device API operates over collective symmetric memory, contrasting with traditional NCCL's host-initiated collectives that use pipeline primitives (Simple, LL, LL128) over regular memory.

GIN achieves this through a three-layer architecture: i) a host-side API extending NCCL Core for communicator initialization, GIN resource management, and collective memory window registration; ii) a device-side API that exposes `put`/`signal` primitives for remote memory operations directly callable from CUDA kernels with flexible completion semantics; and iii) a pluggable network backend architecture supporting both direct GPU-to-NIC communication via DOCA GPUNetIO (GDAKI backend) and CPU-assisted operation via lock-free queues (Proxy backend). By providing both hardware-direct and CPU-assisted plugin interfaces, GIN offers functionality across diverse deployment scenarios while maintaining compatibility with NCCL's existing ecosystem. Unlike typical layered architectures, GIN introduces minimal overhead through compile-time optimizations and direct hardware access.

A. Key GIN Design Elements

The design and implementation of GIN incorporate several key technical elements:

i) **Device API Design.** GIN provides a device-side API that supports flexible cooperation models (thread-level and warp-level collective operations) with optimized primitives for small inline values, window-based memory addressing with byte offsets, and flexible local (flush and counter) and remote completion (signal) mechanisms. This allows applications to express custom communication patterns and fuse communication with computation within kernels.

ii) **Dual Plugin Architecture.** NCCL GIN implements two plugin architectures: the GDAKI interface that leverages the DOCA GPUNetIO backend to enable GPU threads to directly program InfiniBand/RoCE NICs through device verbs, while the Proxy interface uses lock-free GPU-to-CPU queues with 64-byte descriptors to enable GIN functionality on any RDMA-capable NIC.

iii) **Synchronization Mechanisms.** GIN provides local and remote completion notification primitives: signals (for remote notifications) and counters (local completion tracking). These integrate with CUDA's memory model through automatic fence insertion based on thread scope annotations.

iv) **Memory Management.** GIN uses collective window registration (`ncclCommWindowRegister`), where each rank contributes a local buffer and receives handles containing remote keys for all peers, enabling zero-copy one-sided operations.

B. Contributions

In summary, the main contributions of this work are:

i) Design and implementation of GIN in NCCL, including unified host and device APIs, modular synchronization primitives for asynchronous completion (signals and counters), and two interchangeable backend architectures: GDAKI for direct GPU-to-NIC communication using DOCA GPUNetIO, and Proxy for CPU-assisted operation over standard RDMA.

ii) Integration with DeepEP, a specialized MoE communication library, demonstrating GIN's practical applicability and compatibility with existing NVSHMEM-based device-initiated communication.

iii) Comprehensive performance evaluation of GIN using microbenchmarks and application-level experiments with DeepEP kernels, with comparative analysis to establish performance characteristics.

The remainder of this paper is organized as follows. Section II provides the background on GPU communication and motivates the need for GIN. Section III presents the architecture, design principles, device-side API semantics, and implementation of GIN—including the three-layer design and both GDAKI and Proxy backends, along with synchronization primitives and memory management. Section IV describes

the integration of GIN with DeepEP, demonstrating practical use for dynamic MoE workloads. Section V evaluates GIN's performance through microbenchmarks and application-level benchmarks in DeepEP across multi-node GPU clusters. Section VI reviews related work in GPU-direct communication and one-sided programming models. Section VII concludes with lessons learned and future directions for GIN.

II. Background

Traditional communication libraries like OpenSHMEM provide one-sided primitives (`put`, `get`, atomics) over symmetric memory regions, enabling asynchronous data movement without sender-receiver coordination [12]. However, these specifications assume CPU-centric execution where all communication primitives are invoked from host code. GPU integration into HPC systems exposed inefficiencies in this model: fine-grained GPU-to-GPU communication incurred kernel launch overhead, PCIe transfers to stage data through host memory, and CPU scheduling latency [13]. These bottlenecks motivated GPU-aware extensions that enable device-initiated communication directly from CUDA kernels [13], [14].

A. GPUDirect Technologies

GPUDirect RDMA (2013) [15], [16] enables RDMA-capable network interfaces to directly access GPU memory via PCIe Base Address Register (BAR) mappings, eliminating CPU and host memory from the data path for inter-node transfers. The NIC's DMA engine performs PCIe peer-to-peer transactions to GPU BARs, accessing memory regions registered through the `nvidia_p2p` kernel module. However, GPUDirect RDMA provides *consistency guarantees only at kernel boundaries*. GPU memory model semantics (relaxed ordering, write-back caching) prevent safe concurrent access to RDMA-registered memory from executing kernels, forcing applications to separate computation and communication [17].

GPUDirect Async (2016) [18] introduced partial control-path offload: GPU threads trigger pre-configured network operations writing to NIC doorbell registers that are memory-mapped into the GPU address space. However, the CPU must pre-construct communication descriptors, limiting operations to those pre-configured by the host and preventing fully autonomous device-driven networking.

B. Device-Initiated Communication Primitives

Fully device-initiated networking requires implementing network programming interfaces directly in GPU code. Early prototypes like **GPUrdma** [19] and **GIO** [17] exposed InfiniBand verbs as device-callable functions but faced GPU-NIC memory consistency challenges.

NVSHMEM [8] extended OpenSHMEM semantics to GPU clusters with device-callable one-sided operations (`put`, `get`, atomics) invokable from CUDA kernels. This enables the device code to interleave computation and communication without kernel launch overhead, using transport backends including IBGDA (InfiniBand with GPUDirect Async) for inter-node transfers and symmetric memory mechanisms for intra-node communication.

DOCA GPUNetIO provides GPU-side RDMA APIs for InfiniBand and RoCE networks (IBGDA), exposing device functions that enable GPU kernels to directly program NICs [20]. Specifically, it implements GPUDirect RDMA (direct GPU data movement) and GPUDirect Async Kernel-Initiated (GPU controlling network communications) technologies. It forms the foundation for GIN's GDAKI backend, enabling direct GPU-to-NIC communication through hardware-supported device verbs.

C. Networking Hardware for GPU-Initiated Communication

GPU-initiated networking requires network interface cards supporting direct device access through one of several RDMA technologies: InfiniBand, RoCE, or iWARP [21].

InfiniBand provides native RDMA support with credit-based flow control, achieving port-to-port latencies of approximately 130 ns and supporting tens of thousands of nodes per subnet [22]. InfiniBand adapters expose memory-mapped queue pairs, completion queues, and doorbell registers through PCIe BARs, enabling direct GPU access when combined with GPUDirect RDMA [13], [15].

RoCE implements RDMA over standard Ethernet, offering lower cost and broader compatibility with existing data center infrastructure [21], [22]. RoCEv2, the prevalent variant, encapsulates InfiniBand transport over UDP/IP, achieving port-to-port latencies around 400 ns—higher than native InfiniBand but sufficient for many workloads [22]. RoCE requires lossless Ethernet configurations using Priority Flow Control (PFC) and Explicit Congestion Notification (ECN) to prevent packet loss, which can complicate deployment in multi-tenant environments [22]. Both InfiniBand and RoCE share the same user-space verbs API, enabling interconnect portability [21].

For GPU-initiated communication, the hardware requirement is NIC support for device-accessible control structures. NVIDIA ConnectX series adapters (ConnectX-6 Dx and later) and BlueField DPUs provide this capability through DOCA GPUNetIO [20]. Systems lacking such hardware support cannot enable direct GPU-NIC communication and must fall back to CPU-mediated mechanisms. In GIN's architecture, this constraint motivates the dual-backend design: the GDAKI backend leverages DOCA GPUNetIO for direct device communication on supported hardware, while the Proxy backend provides functionally equivalent semantics through lock-free GPU-to-CPU queues and CPU-driven network operations on arbitrary RDMA-capable NICs.

Optimal performance requires co-location of GPUs and NICs on the same PCIe root complex to minimize peer-to-peer latency and maximize bandwidth [15], [23]. Multi-socket systems with distributed PCIe topologies may incur inter-socket traversal penalties, reducing GPUDirect RDMA efficiency. Additionally, GPU-initiated networking requires the `nv_peer_mem` kernel module (for GPUDirect RDMA) and appropriate driver stacks (OFED for InfiniBand, MOFED for

Mellanox adapters) to establish memory mappings between GPU and NIC address spaces [23].

D. NCCL Architecture and Network Plugins

NCCL is the standard collective communication runtime for multi-GPU machine learning, providing topology-aware implementations of allreduce, allgather, reduce-scatter, and broadcast operations [**?**]. NCCL's architecture uses CPU proxy threads to orchestrate network operations, where GPU kernels enqueue communication descriptors into host-visible queues and CPU threads execute them via network plugins. While this design has proven robust for collective operations at scale, NCCL 2.28's Device API [9] extends this architecture with device-side primitives that enable applications to implement custom communication patterns directly from GPU code, integrate communication within compute kernels, and achieve fine-grained computation-communication overlap for emerging workloads.

The widespread adoption of NCCL in production frameworks motivated this integration of device-initiated communication capabilities, preserving ecosystem compatibility while enabling new use cases.

NCCL Network Plugin architecture provides an abstraction layer that decouples the core library from specific network implementations. NCCL supports both internal plugins (e.g., Socket and InfiniBand) built directly into the library, and external plugins that implement the NCCL network API as shared libraries (`libnccl-net.so`). This design allows network vendors and hardware providers to extend NCCL with specialized transport implementations without modifying the NCCL core. External plugins are dynamically loaded at runtime, selected via the `NCCL_NET_PLUGIN` environment variable, enabling seamless integration of diverse networking technologies while maintaining version compatibility through a versioned API interface.

E. Specialized MoE Communication Libraries

MoE architectures in LLMs require dynamic, load-balanced all-to-all token routing with unpredictable message sizes, creating irregular communication patterns that traditional collectives [24] are not well suited for. Specialized libraries like **DeepEP** [25] and **Perplexity's pplx-kernels** [26] target these workloads with CUDA-optimized, GPU-initiated primitives for low-latency all-to-all transfers. These efforts demonstrate the value of GPU-centric communication for MoE workloads but remain separate from NCCL's ecosystem.

GIN brings GPU-initiated RDMA operations into NCCL, enabling applications to leverage both host-optimized collectives and device-driven point-to-point communication within a unified runtime.

Building on these device-initiated networking technologies and NCCL's plugin architecture, the next section presents GIN's design and implementation.

III. NCCL GIN: GPU-Initiated Networking

This section presents the design and implementation of GIN for NCCL. As established in Section II, extending NCCL with device-side primitives enables modern AI workloads to achieve tight coupling between computation and communication. GIN provides this capability by integrating device-initiated one-sided primitives into NCCL, allowing GPU threads to initiate network operations directly from CUDA kernels without CPU involvement.

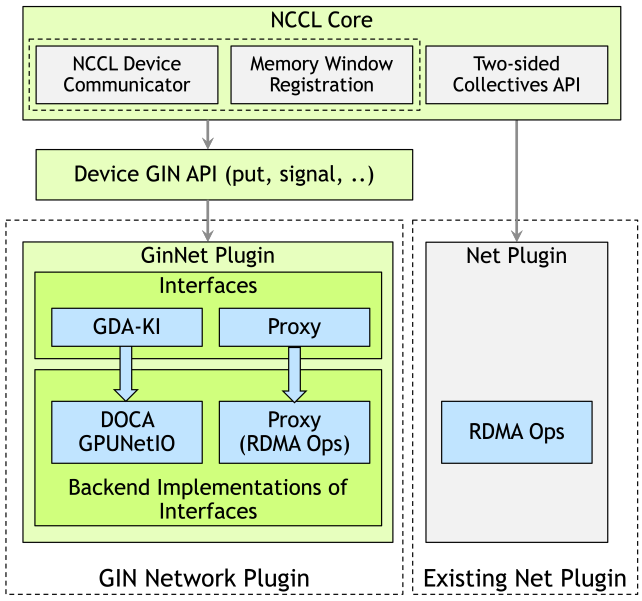

Fig. 3. GIN Architecture showing the interactions between NCCL Core, Plugin Layer, and Device-Side API.

The design preserves NCCL's established programming model and ecosystem integration while adding a parallel low-latency path for device-driven communication. This enables production systems like TensorRT-LLM, vLLM, and SGLang, as well as communication libraries like DeepEP, to implement custom collective algorithms and kernel fusion patterns previously unattainable with conventional NCCL.

We organize the section in three parts: Section III-A introduces the three-layer architecture and core design principles; Section III-B presents the device-side API and demonstrates its usage through a practical example; and Section III-C analyzes the two plugin interfaces and their backend implementations, GDAKI and Proxy, explaining their design rationale and performance characteristics.

A. Core Principles and Architecture

NCCL GIN's architecture is built on one-sided communication semantics, which enable two critical performance objectives: *maximum communication-computation overlap* and *minimum end-to-end operation latency*. By enabling GPU threads to initiate RDMA operations directly—reading from and writing to remote memory without receiver coordination—GIN eliminates host-device synchronization overhead and two-sided handshaking delays, allowing asynchronous data transfers to proceed concurrently with computation.

The architecture comprises three cooperating layers for GIN—shown in the green path (left side) of Figure 3—

designed to balance high performance with broad vendor support: *NCCL Core* (host-side APIs), *Device GIN API* (GPU-callable primitives), and *GIN Network Plugin* (pluggable network backends). The grey path (right side) of Figure 3 shows the existing two-sided collectives API over the built-in network plugin. We discuss the three GIN layers further: **i) NCCL Core**: the host-side NCCL functionality manages memory-window registration, resource allocation, and communicator initialization, providing the foundation for GIN resource management and extending NCCL's existing infrastructure with device-initiated communication capabilities; **ii) Device GIN API**: the device-side API exposes a unified interface to GPU kernels, enabling applications to invoke one-sided communication operations directly from CUDA kernels. It dispatches to either NCCL-provided (Proxy) or plugin-provided (GDAKI) implementations based on the underlying network backend; **iii) GIN Network Plugin**: the plugin layer provides an extensibility mechanism that defines remote data-movement operations and supports dual semantics—GDAKI and Proxy—to maximize network coverage. NCCL's InfiniBand transport implements both, while external vendors may supply their own. Under the Proxy interface, NCCL Core owns control structures, device-side queuing logic, and device API implementations, while plugins provide only CPU-based `put`, `signal`, `test`, and `regMr` operations, enabling networks without GPU-direct capabilities and lowering the barrier to GIN adoption. Under GDAKI semantics, plugins own both the control path and device API: they create GPU contexts via `createContext` and supply device code that directly programs NICs using kernel-initiated APIs such as DOCA GPUNetIO, with NCCL Core coordinating the structure exchange between host and device components.

These components enable device-initiated one-sided communication through several key design elements: *one-sided semantics* for unilateral data movement, *symmetric memory windows* for zero-copy remote access, and *asynchronous completion tracking* with flexible ordering semantics.

One-Sided Communication Semantics. GIN exposes one-sided RDMA primitives—`put` for remote writes and `put` with `signal` for writes with remote notification—enabling GPU threads to access remote memory without any receiver coordination. This one-sided model eliminates the overhead of handshaking protocols and the need for receiver participation, allowing initiators to issue transfers unilaterally and independently control when to verify completion. The one-sided model proves particularly effective for irregular communication patterns in MoE workloads, where dynamic token routing creates unpredictable traffic patterns, and for single-shot collective implementations that benefit from parallel, non-blocking peer communication.

Windows-based *(A)*Symmetric Memory. Communication buffers must be collectively registered across all ranks, establishing memory windows that are symmetric in addressability, following the MPI RMA window model [27]. All processes can access registered memory, analogous to NVSHMEM's symmetric heap. GIN windows are designed to support asymmetry in capacity: each rank may register different buffer sizes. This flexibility proves essential for disaggregated serving architectures where prefill ranks require larger buffers than decode ranks. Note that the current implementation in NCCL 2.28 enforces symmetric sizes, though this constraint will be lifted in future releases. Furthermore, the memory allocation is not coupled with the registration: NCCL-GIN memory windows allow users to create a window from an existing allocation. Each registration produces window handles encapsulating remote access metadata. The window handles provide backend-specific optimization opportunities. Backends construct RDMA descriptors directly from window metadata and target addresses using rank-relative offsets.

GIN Contexts for Network Parallelism. GIN contexts serve as the primary abstraction for expressing network parallelism. Each context abstracts a channel between the GPU and NIC and encapsulates network resources and connections (queue pairs (QPs)). Multiple contexts per communicator enable applications to exploit network-level parallelism across multiple NICs, ports, and QPs, allowing independent concurrent communication streams. A single context can address every rank associated with the communicator. As such, a context can issue multiple concurrent operations to different peers.

Asynchronous Completion Tracking. All device-initiated operations execute asynchronously and return immediately to enable other work to proceed in parallel, such as computation or intra-node communication through NVLink. Applications track operation completion through two distinct mechanisms using per-context resources. *Counters* are local objects that track completion on the sender side, indicating when source buffers can be safely reused. Unlike the *flush* operation that tracks the completion of all operations posted to the context, *Counters* are a powerful concept to track local completions per operation, allowing users to describe pipeline algorithms efficiently. Each data movement operation can optionally report local completion to a counter provided by the user (via counterID).

On the other hand, *Signals*, which are symmetric objects, provide remote completion tracking, confirming data arrival and visibility at the destination. Unlike OpenSHMEM's address-based synchronization, GIN uses ID-based addressing: each signal (and counter) is identified by an integer ID rather than a memory address. This ID-based design simplifies resource management and enables efficient hardware implementation of completion notifications.

Ordering Semantics. To maximize network efficiency and throughput, GIN operations are unordered by default. However, GIN provides ordering guarantees only between `put` and `signal` operations in the same context to the same peer. When a `signal` operation (either standalone `signal` or `put` with `signal` action `ncclGin_SignalInc`/`ncclGin_SignalAdd`) completes at the destination, it guarantees that all preceding `put` operations to that peer on the same context have completed and are visible to remote GPU threads. This provides lightweight ordering without requiring explicit fence operations: applica-

tions can batch multiple puts and attach a signal to the final operation, ensuring ordered remote visibility of the entire sequence. In contrast, the flush operation ensures only local completion—all pending operations have been consumed and source buffers can be safely reused—but makes no guarantees about remote visibility. GIN's signal-based ordering aims for maximum performance by providing ordering guarantees selectively through signals rather than globally. It is worth noting that GIN does not assume or guarantee any ordering between GPU threads. It is the responsibility of the user to synchronize the threads using CUDA synchronization primitives.

B. Device-Side API and Programming Model

The device-side API provides GPU kernels with direct control over network operations, exposing methods invokable from CUDA device code without CPU intervention. The programming model centers on the ncclGin object, which encapsulates networking resources (contexts, operation queues, and peer connection state) and provides methods for data movement, completion tracking, and synchronization. Backend selection (DOCA GPUNetIO or Proxy) occurs transparently at communicator initialization based on hardware capabilities and user configuration, presenting an identical device-facing interface regardless of the underlying implementation.

API Organization. The interface organizes operations into four logical categories that reflect the communication workflow. *Data movement* operations (put, putValue, signal) initiate one-sided transfers or notifications to remote peers, posting RDMA operations that execute asynchronously. *Completion tracking* operations distinguish between local completion (flush, readCounter, waitCounter)—which signals that source buffers can be safely reused—and remote completion (readSignal, waitSignal)—which confirms data arrival at the destination and visibility to remote GPU threads. *Barrier synchronization* (ncclGinBarrierSession) coordinates all ranks within a team before communication phases, ensuring global consistency through network-wide synchronization. *State management* operations (resetCounter, resetSignal) reset completion state for reuse across multiple communication rounds. This separation of concerns enables fine-grained control over communication-computation overlap and supports diverse synchronization patterns.

Furthermore, GIN device API has a tight interaction with the GPU memory model and takes hints from users to optimize performance-critical memory ordering and consistency tasks. For instance, put takes two hints from the user on the provided visibility scope of the data as well as the expected user visibility after the operation completes.

Usage Workflow. Applications interact with GIN through a three-phase workflow (Listing 1). During initialization, applications enable GIN support when creating the NCCL Device communicator (ncclDevCommCreate with appropriate configuration flags), which creates the GIN contexts. Applications then collectively register memory buffers as windows using ncclCommWindowRegister, which returns window handles for use in device code. During kernel execution, device threads instantiate an ncclGin object specifying the desired context index (typically selected based on target peer or load balancing requirements) and issue data movement operations, optionally attaching completion actions such as remote signal increments or local counter updates. Finally, kernels synchronize by waiting on signals or counters before reusing buffers or advancing to subsequent computation stages, ensuring proper ordering of communication and computation phases. Listing 1 presents a simplified view of the NCCL GIN interface, highlighting the essential operations. The actual API includes additional template parameters and options for advanced use cases (e.g., cooperative thread groups, inline data transfers), but the core abstraction remains consistent.

```
class ncclGin {
  // Constructor: initialize with device
  // communicator and context ID
  ncclGin(ncclDevComm comm, int contextIndex);

  // Data Movement Operations
  void put(team, peer, dstWindow, dstOffset,
           srcWindow, srcOffset, bytes, ...);
  void putValue(team, peer, dstWindow, dstOffset,
    value, ...);
  void signal(team, peer, signalId);

  // Local Completion Tracking
  void flush(coop); // Block until ops complete
  uint64_t readCounter(counterId); // Poll counter
  void waitCounter(coop, counterId, expectedValue);
  void resetCounter(counterId); // Reset for reuse

  // Remote Completion Tracking
  uint64_t readSignal(signalId); // Poll signal
  void waitSignal(coop, signalId, expectedValue);
  void resetSignal(signalId); // Reset for reuse
};

// Network Barrier for cross-rank synchronization
class ncclGinBarrierSession {
  ncclGinBarrierSession(coop, gin, team,
    barrierHandle, index);
  void sync(coop); // Global barrier sync
};

// Optional: Attach completion actions to data
// Remote signal
put(..., ncclGin_SignalInc{signalId});
// Local counter
put(..., ncclGin_CounterInc{counterId});
```

Listing 1. Simplified NCCL GIN device API.

Usage Example. Listing 2 demonstrates a unidirectional ring exchange pattern using GIN primitives. In this kernel, each rank sends data to its successor in a ring topology (myRank + 1), with data flowing in one direction around the ring, implementing a common pattern in pipelined communication algorithms. The put operation (lines 13–16) transfers data from the local sendWin to the peer's recvWin at a computed offset and atomically increments remote signal 0 upon completion, providing remote notification of data arrival. The sending rank then waits for its own signal 0 to be incremented

by its predecessor (line 19), ensuring that received data has arrived and is visible to local GPU threads before proceeding with computation. Finally, the signal is reset for subsequent communication rounds (line 21). This pattern illustrates how GIN's asynchronous operations, flexible completion semantics, and explicit synchronization primitives enable efficient overlapped point-to-point communication from device code.

```
1  __global__ void ringExchange(
2    ncclDevComm devComm,
3    ncclWindow_t sendWin,
4    ncclWindow_t recvWin,
5    size_t dataSize, int myRank)
6  {
7    // Initialize ctx 0
8    ncclGin gin(devComm, 0);
9    int peer = (myRank + 1) % devComm.nRanks;
10
11   // Send data to peer and signal completion
12   // by incrementing peer's signal
13   gin.put(ncclTeamWorld(devComm), peer,
14             recvWin, myRank * dataSize,
15             sendWin, peer * dataSize, dataSize,
16             ncclGin_SignalInc{0} );
17
18   // Wait for predecessor
19   gin.waitSignal(ncclCoopCta(), 0, 1);
20   // Reset for next round
21   gin.resetSignal(0);
22 }
```

Listing 2. Unidirectional ring exchange using NCCL GIN.

C. Backend (GDAKI and Proxy) Implementations

The device API abstracts two distinct backend implementations that realize the dual plugin semantics described earlier. The GDAKI backend implements the GDAKI semantics through direct GPU-to-NIC communication via DOCA GPUNetIO, with the plugin providing both device API and control path. The Proxy backend implements Proxy semantics through CPU-mediated transfers, with NCCL Core providing the device API and the plugin providing only CPU-based data path operations. Both backends expose an identical device-facing interface, enabling transparent backend selection at runtime without application code changes.

GDAKI Backend: Direct GPU-to-NIC Communication. The GDAKI backend implements device-initiated networking in its purest form by leveraging DOCA GPUNetIO to enable GPU threads to directly program network interface cards without CPU involvement. When a kernel invokes `put`, GPU threads construct RDMA work queue entries (WQEs) in device memory, populate them with source/destination addresses and transfer metadata, and directly write to the NIC's doorbell registers to trigger DMA transfers. The NIC hardware manages operation progress autonomously: it polls GPU memory for new WQEs, executes RDMA transactions over InfiniBand or RoCE, and updates completion queue entries in GPU-visible memory. This direct GPU-NIC path eliminates PCIe round-trips to the CPU and achieves low-latency for small messages. However, the approach requires modern hardware and software: ConnectX-6 Dx or newer NICs with GPU-accessible control structures and required CUDA 12.x version (as detailed in [20]). Additionally, proper system configuration—including GPUDirect RDMA kernel modules (`nv_peer_mem` or `dmabuf`) and co-located GPU-NIC PCIe topology—is essential for correct operation and optimal performance.

Proxy Backend: CPU-Assisted Communication. The Proxy backend trades peak performance for hardware portability by introducing the CPU into the communication path as an intermediary between the GPU and NIC. GPU threads enqueue operation descriptors—64-bytes containing source/destination window handles, potential source inline value, offsets, sizes, and completion actions—into lock-free queues allocated in CPU memory using fire-and-forget stores. A dedicated CPU proxy thread per communicator, pinned to NUMA nodes near the local rank's GPU and NIC, continuously polls these queues. Upon detecting new descriptors, the proxy thread extracts the fields and posts the network operation through the network plugin's `iput`/`iput_signal` interface, which maps to standard InfiniBand verbs or other network APIs. The plugin is responsible for executing the signal and ensuring visibility of all prior *put* operations. Completion notifications follow the reverse path: the proxy thread polls on completions using the network plugin's *test* interface, matches completed operations to their associated GIN counters, and updates completion state in GPU-visible memory (GPU or CPU-resident based on GDRCopy availability). While this CPU involvement introduces additional latency compared to GDAKI, the Proxy backend supports arbitrary CUDA versions, any GPUDirect RDMA-capable NIC (InfiniBand, RoCE, iWARP), and Volta-or-newer GPUs. Additionally, CPU involvement simplifies debugging through host-side instrumentation and enables graceful performance degradation on systems where direct GPU-NIC communication is unavailable.

Backend Selection and Portability. Table I summarizes the architectural differences between GDAKI and Proxy backends. High-performance production systems with modern NVIDIA end-to-end networking infrastructure (ConnectX-6 Dx or newer NICs, recent CUDA versions, properly configured GPUDirect RDMA) favor GDAKI for minimal latency and zero CPU overhead. Development environments, legacy hardware deployments, multi-vendor network fabrics, or systems with misconfigured GPUDirect support rely on Proxy for functional correctness and operational flexibility. The runtime automatically detects available backends during communicator initialization (`ncclCommInitRank`), probing for DOCA GPUNetIO support through capability queries and falling back to Proxy when necessary. Applications can override this selection through environment variables (`NCCL_GIN_BACKEND`) for debugging or performance tuning. This design ensures portability across diverse deployment scenarios while preserving the performance benefits of direct GPU-NIC communication where hardware and software infrastructure permit.

Characteristic	GDAKI	Proxy
Comm. Path	Direct GPU↔NIC	GPU→CPU↔NIC
CPU Involvement	Zero (fully device-driven)	Required (dedicated thread per comm)
Progress Model	NIC hardware autonomously polls GPU memory	CPU thread polls queues and posts to NIC
Operation Posting	GPU directly rings NIC doorbell	GPU writes descriptor; CPU extracts and posts
Hardware Requirements	ConnectX-6 Dx+ NIC CUDA 12.2+	Any RDMA NIC Any CUDA GPU Any CUDA version
Implementation Base	DOCA GPUNetIO device verbs	Plugin `iput`/`test` API
Debugging Support	Device-side tools only	Host-side inspection and tracing
Portability	Requires GPU-NIC direct access (reference impl: ConnectX)	Universal (all vendors)
Use Case	Production HPC/AI clusters	Development, legacy, multi-vendor

TABLE I
ARCHITECTURAL COMPARISON OF NCCL GIN BACKEND IMPLEMENTATIONS

IV. DEEPEP INTEGRATION

This section presents NCCL GIN integration into DeepEP to validate its effectiveness for workloads requiring computation-communication fusion and low-latency. DeepEP is a specialized MoE communication library that implements device-initiated sparse all-to-all communication—dispatch and combine primitives—using NVSHMEM and IBGDA. The library provides two flavors of dispatch and combine primitives: high-throughput (HT) and low-latency (LL) kernels used in training/inference-prefill and inference-decode phases, respectively. This integration demonstrates how DeepEP's device-initiated communication patterns can be implemented using GIN APIs while preserving performance characteristics and maintaining coexistence with the existing NVSHMEM communication backend.

A. Integration Requirements

DeepEP's communication patterns impose several requirements: **i) High QP Parallelism**—HT kernels require 24 QPs, while LL kernels require 8-16 QPs matching local expert count; **ii) Heterogeneous Topology Support**—HT uses symmetric rank-to-rank RDMA with NVLink forwarding, while LL uses full all-to-all RDMA mesh; **iii) Fine-Grained Synchronization**—atomic updates to head/tail pointers for circular buffer flow control; **iv) Backend Coexistence**—NVSHMEM IBGDA and GIN backends must coexist to match user preference according to the execution environment.

B. Backend Integration Strategy

The integration employs a minimal abstraction layer for lifecycle management (initialization, memory allocation, barriers) while allowing performance-critical operations to use backend-specific device APIs directly in kernels via conditional compilation. This design accommodates fundamental semantic differences: IBGDA uses pointer-based addressing with memory atomics, while NCCL GIN uses window-based addressing with signal atomics.

The integration addresses four key translation challenges. **First**, multi-communicator mapping: since NCCL GIN provides 4 contexts per communicator, meeting DeepEP's QP requirements requires $\lceil \text{QPs}/4 \rceil$ communicators, with work distributed via deterministic selection (`comm_id = id / 4`, `ctx_id = id % 4`). **Second**, memory management: the backend registers allocated buffers with all communicators and stores device-accessible window handles in GPU memory, enabling kernels to translate pointer arithmetic to (window, offset) pairs. **Third**, synchronization: pre-allocated structured signal layouts map memory-based atomics to signal primitives (HT: two signals per channel for head/tail; LL: one signal per expert). **Fourth**, semantic preservation: zero-byte `puts` with atomic signals emulate release-acquire semantics, ensuring visibility of prior transfers before signaling completion.

C. Operation Semantic Mapping

The migration from NVSHMEM to NCCL GIN requires translating between distinct programming models while preserving communication semantics. NVSHMEM provides a PGAS (Partitioned Global Address Space) abstraction with pointer-based addressing and memory-based synchronization primitives, while NCCL GIN follows a window-based one-sided model with signal-based completion tracking (reviewed earlier in Section III-B). Table II maps DeepEP's communication patterns to the corresponding NVSHMEM and GIN primitives, highlighting the key semantic transformations required for integration. For data transfers, kernels compute window-relative offsets on-the-fly and select communicators deterministically based on channel or expert ID, load balancing across QPs. For synchronization, the signal-based design decouples data movement from completion notification: bulk transfers use `put()` without immediate signaling, followed by explicit `signal()` operations that atomically update remote counters only after all prior operations complete. This pattern implements release-acquire semantics through network primitives rather than memory ordering, ensuring data visibility at the completion of signal arrival.

D. High-Throughput Kernel Integration

HT kernels optimize for large batches (4096 tokens) using hierarchical communication. GPUs send data to remote nodes over symmetric RDMA connections, which then forward tokens via NVLink to destination GPUs. This minimizes inter-node traffic while maximizing intra-node bandwidth. The RDMA buffer contains multiple channels functioning as QPs, each with send/receive buffers. Head and tail pointers track buffer occupancy, providing circular-buffer flow control.

The dispatch kernel assigns specialized roles to SMs. Odd-numbered SMs act as Senders (transmitting tokens to re-

Aspect	DeepEP-Custom NVSHMEM/IBGDA Layer	NCCL GIN
Memory Model	*PGAS*: Partitioned Global Address Space with symmetric heap; direct pointer-based addressing across PEs	*Window-based*: Explicit registration of memory windows; offset-based addressing within windows
Data Transfer API	`put_nbi(dst_ptr, src_ptr, count, pe)` — pointer-based non-blocking puts; warp-collective execution; asynchronous one-sided	`put(team, peer, dstWin, dstOff, srcWin, srcOff, bytes)` — window/offset pairs; thread-based; asynchronous one-sided
Synchronization Primitives	*Memory atomics*: Direct atomic operations (`atomic_add`, `atomic_fetch`) on remote memory locations	*Signal atomics*: Dedicated signal infrastructure; `signal(peer, id)` for atomic updates, `readSignal(id)` for polling
Completion Model	*Local flush*: `quiet()` per QP; blocks until all operations complete on a particular QP for a unique remote PE	*Per-context flush*: `flush()` per context enables parallel completion checking across multiple queues
Barrier Operations	Team-based barriers (`barrier(team)`, `barrier_all()`) with opaque team handles; synchronizes all team members	`ncclGinBarrierSession` with team tags and session IDs; enables symmetric subset synchronization without full team coordination

TABLE II
SELECTIVE DEEPEP-CUSTOM NVSHMEM/IBGDA AND NCCL GIN APIS USED BY DEEPEP COMMUNICATION KERNELS.
NOTE: THIS TABLE IS NOT A COMPARISON OF NVSHMEM/IBGDA AND NCCL GIN APIS—IT FOCUSES ON CORRESPONDING NVSHMEM AND NCCL APIS USED IN DEEPEP LIBRARY.

mote ranks) and NVLink Receivers (final destinations), while even-numbered SMs act as Forwarders (receiving RDMA tokens and forwarding them via NVLink). This specialization enables concurrent, bidirectional communication. To reduce contention, separate channels are used for data/tail updates and for head-pointer updates, distributing work across distinct communicators.

Following the operation mappings (Table II), each SM role uses signal-based atomics for pointer management and window-based `put()` for data transfers. Remote tail signals are incremented using `gin.signal(SignalAdd, 1)`, while head-pointer flow control relies on polling local head signals via `gin.readSignal(signal_id)`. Data transfers use single-threaded NCCL GIN `put()` followed by `__syncwarp()` to preserve warp-collective semantics. The notify dispatch kernel uses a coordinator SM to flush all writes (`gin.flush()`), perform a barrier across symmetric RDMA ranks, reset head/tail signals, and exchange metadata before starting the main dispatch.

The combine kernel mirrors the dispatch kernel's specialization, using 25 warps per SM. Even-numbered SMs serve as NVLink Senders (distributing input tokens to local buffers), RDMA Receivers (integrating remote tokens with bias terms), and Coordinators monitoring receiver progress. Odd-numbered SMs act as NVLink and RDMA Forwarders (merging local tokens and forwarding them to remote ranks), with corresponding Coordinators monitoring forwarders. The operation mapping parallels dispatch: Forwarder warps use single-threaded `put()` with `__syncwarp()` for data transfer, `readSignal()` for head-pointer polling, and `signal()` for tail updates. Receiver warps use `readSignal()` for tail-pointer monitoring, while Coordinator warps update head pointers via `signal()`.

E. Low-Latency Kernel Integration

LL kernels optimize for small batches (1-128 tokens) using full all-to-all RDMA mesh connectivity, enabling direct GPU-to-GPU communication. Token streaming embeds routing metadata without separate notify phases, minimizing dispatch-combine cycle time. Per-expert signal allocation provides direct coordination between any expert pair across the cluster.

SM allocation distributes experts using $G = \lceil N/S \rceil$ warp groups per SM, where N is total experts and S is available SMs. Each SM is assigned experts via `expert_idx = sm_id * G + warp_group_id`. Within each warp group, most warps handle FP8 quantization and token sending, while a counting warp manages expert counts and metadata for all assigned experts. This organization enables efficient parallelization across hundreds of experts (e.g., 288 experts across 132 SMs with 3 warp groups per SM).

LL kernels leverage hybrid NVLink-RDMA communication. For each token transfer, kernels check NVLink availability via custom function `nccl_get_p2p_ptr`. If available, tokens are copied directly using warp-level memory operations; otherwise, NCCL GIN's `put()` performs RDMA transfers (Table II). Tokens are first copied from PyTorch tensors into RDMA send buffers, with optional FP8 quantization applied during this stage. After completing token transfers to a destination, a counting warp sends the per-expert token count using zero-byte `put()` with `SignalAdd`, ensuring all prior data transfers have completed and become visible before the count is delivered—this implements the release-acquire semantics described in the Backend Integration Strategy. Receivers poll using `gin.readSignal(signal_id)` until tokens arrive.

The combine kernel routes expert outputs back to source ranks with weighted reduction. It uses the same hybrid NVLink-RDMA approach with optional LogFMT compression to reduce data volume. After transmitting expert outputs,

Specification	DGXH100 Node
Number of Nodes	576
GPU Model	H100 80GB HBM3
GPUs per Node	8 (640 GB total)
GPU Memory BW	3.2 TB/s
NVLink Generation	4th Generation
NVLink BW	900 GB/s bidirectional
NVLink per GPU	18 links
CPU Model	Intel Xeon Platinum 8480CL
CPU Sockets	2
CPU Cores	112 (56 per socket)
CPU Clock	2.0 GHz base, 3.8 GHz boost
System Memory	2 TB
InfiniBand	8×400 Gbit/s (compute) 2×400 Gbit/s (storage)

TABLE III
HARDWARE SPECIFICATION OF EOS DGXH100 COMPUTE NODES.

flag signals notify destinations using zero-byte `put()` combined with `SignalAdd`, ensuring completion before receivers begin accumulation. Receivers employ TMA load warps to fetch expert outputs into shared memory, then apply top-k weights using reduction warps in FP32 before converting to BF16 output.

V. Performance Evaluation

This section evaluates NCCL GIN and NVSHMEM through two complementary approaches. We begin with point-to-point microbenchmarks (Section V-A) to isolate protocol-level performance characteristics, then assess integration with DeepEP version 1.2.1, a production MoE communication library. The DeepEP evaluation[1] covers High-Throughput (HT) kernels for training and inference-prefill (Section V-B), and Low-Latency (LL) kernels for inference-decode (Section V-C) under both hybrid RDMA+NVLink and pure RDMA configurations.

All experiments run on NVIDIA's EOS cluster with H100 GPUs (Table III), using NVSHMEM version 3.4.5 and NCCL version 2.28. DeepEP benchmarks allocate 24 SMs per GPU, with communication distributed across NVLink and RDMA based on automatically selected channel configurations.

A. Point-to-Point Microbenchmarks

To establish baseline performance, we measure `put` with signal latency using ping-pong tests across message sizes from 4 bytes to 4 MB between two H100 GPUs. Figure 4 presents the performance of NCCL GIN's dual backends (GDAKI and Proxy) alongside NVSHMEM IBGDA and IBRC transports.

For small messages (4–128 bytes), NCCL GIN GDAKI achieves 16.7 μs round-trip latency, comparable to NVSHMEM IBRC at 16.0 μs, while NVSHMEM IBGDA achieves 24.3 μs. The GDAKI backend's direct GPU-to-NIC path eliminates CPU proxy overhead, while the Proxy backend achieves 18.0 μs despite GPU-to-CPU queue traversal. At larger message sizes, bandwidth limitations dominate and all

[1]While NVSHMEM supports both IBGDA and IBRC transports, DeepEP's implementation is tightly coupled with IBGDA.

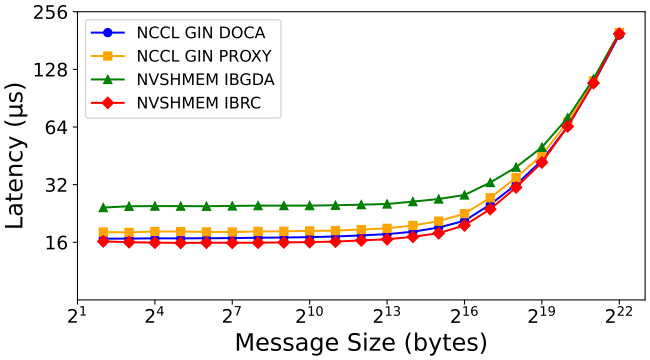

Fig. 4. Point-to-point latency: NVSHMEM IBGDA/IBRC and NCCL GIN GDAKI/Proxy backends.

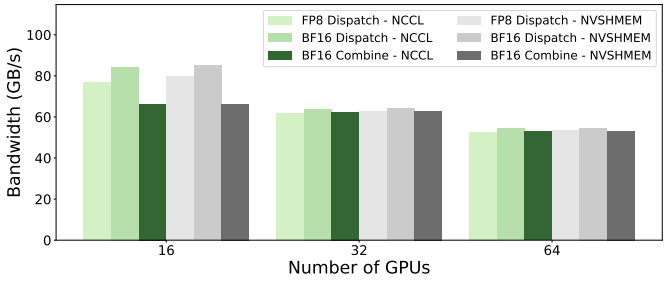

Fig. 5. HT kernel bandwidth for NCCL GIN and NVSHMEM.

implementations converge, validating NCCL GIN's fundamental performance characteristics for application integration.

B. High-Throughput Kernels

HT kernels optimize for MoE training and inference-prefill with large token batches (4096 tokens), using hierarchical communication where specialized SM roles (Senders, Forwarders, NVLink Receivers) minimize inter-node RDMA traffic while maximizing intra-node NVLink bandwidth. Figure 5 presents dispatch and combine bandwidth for FP8 and BF16 precision across 2, 4, and 8 nodes, reporting separate RDMA and NVLink metrics.

Both implementations deliver comparable performance across all configurations. At 2 nodes (16 GPUs) with BF16 precision, dispatch operations achieve 84.36 GB/s RDMA bandwidth with NCCL GIN and 84.97 GB/s with NVSHMEM. At 8 nodes (64 GPUs), both implementations sustain approximately 53–54 GB/s RDMA bandwidth for dispatch operations. The results remain within 1–2% across scales, precision modes, and operation types, demonstrating that NCCL GIN preserves HT throughput while enabling standardization on NCCL infrastructure.

C. Low-Latency Kernels

LL kernels (Section IV) optimize for MoE inference-decode with small token batches (1–128 tokens), using full all-to-all RDMA mesh connectivity with per-expert signals and hybrid NVLink-RDMA paths. With BF16 precision and hidden dimension 7168, dispatch operations transfer 14,352-byte mes-

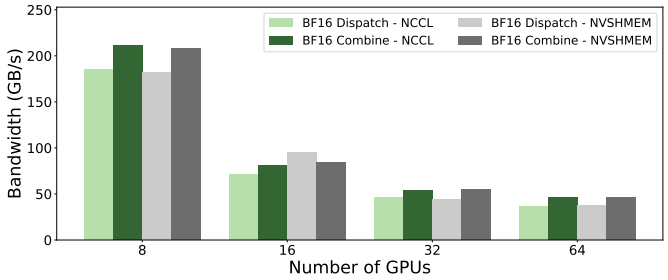

Fig. 6. LL kernel bandwidth with NVLink enabled for NCCL GIN and NVSHMEM.

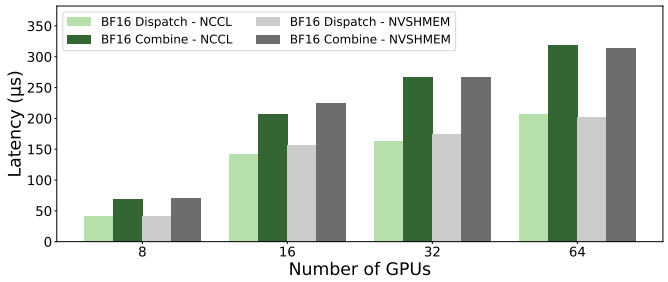

Fig. 7. LL kernel latency with NVLink enabled for NCCL GIN and NVSHMEM.

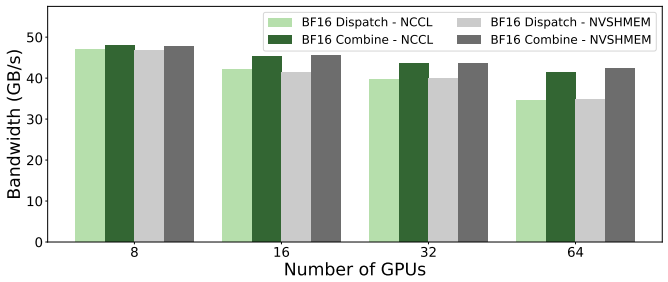

Fig. 8. LL kernel bandwidth with NVLink disabled (pure RDMA) for NCCL GIN and NVSHMEM.

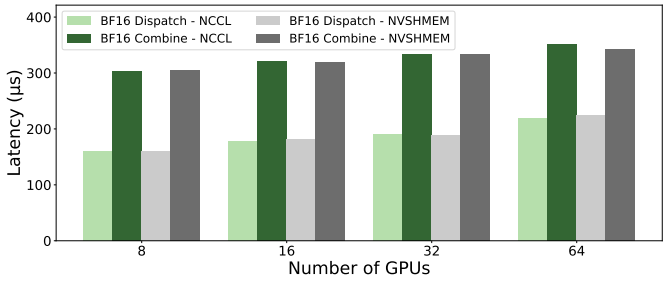

Fig. 9. LL kernel latency with NVLink disabled (pure RDMA) for NCCL GIN and NVSHMEM.

sages (14,336 bytes of token data plus 16 bytes of metadata for token source indexing), while combine operations transfer 14,336-byte messages. Zero-byte `put` with signal operations implement release-acquire semantics for data visibility. We evaluate under hybrid RDMA+NVLink and pure RDMA configurations to assess performance across deployment scenarios.

LL Kernels with NVLink Enabled (RDMA+NVLink). This configuration represents typical deployments with intra-node NVLink and inter-node RDMA. Figures 6 and 7 show bandwidth and latency comparisons. At 1 node (8 GPUs), NCCL GIN slightly outperforms NVSHMEM: dispatch achieves 185.28 GB/s and 40.62 μs compared to 182.15 GB/s and 41.43 μs, while combine operations are virtually identical (211 GB/s, 69 μs).

At multi-node scales, both implementations show comparable performance with minor variations. NCCL GIN consistently delivers lower latency across scales (e.g., 9% lower at 2 nodes: 142.51 μs compared to 157.00 μs). Combine operations remain within 1–3% across all scales.

LL Kernels with NVLink Disabled (Pure RDMA). With NVLink disabled, all communication routes through RDMA—testing scenarios like cross-switch topologies or systems without NVLink. Figures 8 and 9 present measurements showing both implementations maintain comparable performance across scales, with most metrics within 1–2%. At 1 node (8 GPUs), dispatch operations achieve 47.00 GB/s bandwidth and 160.82 μs latency with NCCL GIN, while NVSHMEM achieves 46.79 GB/s and 160.67 μs. At 8 nodes (64 GPUs), both implementations sustain approximately 34–35 GB/s bandwidth and 219–225 μs latency.

D. Discussion

The evaluation results demonstrate that NCCL GIN provides device-initiated communication capabilities with similar performance characteristics as NVSHMEM. Across microbenchmarks and application workloads (HT and LL kernels), GIN integrates device-initiated primitives with NCCL's topology-aware collectives within a single runtime, combining the flexibility of device-side APIs with NCCL's production infrastructure. The GIN implementation remains under active development, with planned optimizations including batching work queue entries and amortizing doorbell costs across multiple operations to further improve performance.

VI. Related Work

Device-Initiated Communication Libraries. OpenSHMEM [12], [28] established PGAS semantics for symmetric memory and one-sided operations, but early GPU extensions remained CPU-mediated [14]. NVSHMEM [8], [13] enabled device-callable operations from CUDA kernels, achieving 60–75% speedups by eliminating kernel launch overhead. However, NVSHMEM operates as a standalone runtime separate from existing collective communication frameworks. Early GPU-initiated RDMA efforts like GPUrdma [19] and GIO [17] faced GPU-NIC memory consistency challenges, achieving 44% improvements for irregular applications through kernel driver extensions. DOCA GPUNetIO [20], [29] provides production-grade device-side RDMA APIs for InfiniBand and RoCE, forming the foundation for GIN's GDAKI backend.

Collective Communication Runtimes. NCCL [6], [30] provides topology-aware collective algorithms and production

infrastructure for distributed training, traditionally using host-initiated communication. MPI RMA [31] operations remain host-initiated, while UCX [32] and UCC [33] provide unified communication frameworks without device-callable primitives. NCCL 2.28 extends this with the Device API, integrating device-initiated capabilities (LSA, Multimem, GIN) into NCCL's existing infrastructure.

MoE Communication Libraries. MoE architectures require irregular all-to-all routing with unpredictable message sizes [24]. DeepSpeed-MoE [34] employs hierarchical parallelism, while FasterMoE [35] and Tutel [36] optimize expert scheduling. DeepEP [25] and pplx-kernels [26] provide low-latency GPU-initiated primitives but operate independently from collective communication frameworks.

GIN's Positioning. GIN uniquely integrates device-initiated network primitives into NCCL's production infrastructure through dual backends: GDAKI for direct GPU-to-NIC communication and Proxy for CPU-assisted operation on commodity hardware. This integration preserves NCCL's ecosystem compatibility while enabling device-driven communication for emerging workloads like MoE inference and kernel fusion patterns.

VII. Conclusions and Future Work

Modern AI workloads—including MoE inference and compiler-generated fusion kernels—require direct GPU control over network operations, capabilities that extend beyond NCCL's traditional host-initiated model. This work introduces **GIN**, part of NCCL 2.28's Device API [9], which enables GPU threads to issue one-sided RDMA operations directly from CUDA kernels. GIN provides a unified three-layer architecture (host APIs, device APIs, and pluggable network backends with dual semantics) and supports both direct GPU-to-NIC communication via GDAKI and CPU-assisted operation on standard RDMA hardware. Our evaluation validates GIN's practical viability: the GDAKI backend achieves 16.7 μs round-trip latency for small messages, and DeepEP integration demonstrates competitive performance across DeepEP's High-Throughput and Low-Latency kernels with minimal code changes.

Importantly, GIN's value extends beyond raw performance to ecosystem unification, delivering scalability and extensibility through integration with NCCL's production infrastructure. Applications gain access to a unified set of communication abstractions—Load/Store Accessible (LSA) for NVLink/PCIe, Multimem for NVLink SHARP, and GIN for network RDMA—enabling the right primitive for each pattern. Critically, this integration preserves NCCL's production-ready features: hierarchical communicators for multi-dimensional parallelism (expert-parallel, tensor-parallel, pipeline-parallel), fault tolerance and elasticity for resilient large-scale training, and topology-aware optimization. These capabilities eliminate the operational complexity of deploying multiple communication runtimes while providing the flexibility needed for emerging workloads like MoE inference and compiler-generated fusion kernels.

Future work will focus on the broader adoption of GIN in production applications such as PyTorch distributed training, TensorRT-LLM inference serving, vLLM and SGLang for LLM inference, and JAX/Triton compiler-generated kernels. We also plan to extend GIN's API with additional one-sided primitives to support emerging communication patterns and distributed algorithm requirements.

Acknowledgments

The authors thank the NCCL and NVSHMEM teams for their contributions. The authors also thank Jeff Hammond and Matthew Nicely for their careful review of the manuscript and for their insightful feedback.

The authors acknowledge the use of Cursor AI to assist in the writing and editing of this manuscript. The authors reviewed and approved all content for accuracy and originality.

References

[1] NVIDIA Corporation, "TensorRT-LLM: A Framework for Efficient Inference of Large Language Models," https://github.com/NVIDIA/TensorRT-LLM, 2023, accessed: 2025.

[2] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "vLLM: Easy, Fast, and Cheap LLM Serving with PagedAttention," arXiv:2309.06180, 2023, https://arxiv.org/abs/2309.06180.

[3] DeepSeek-AI, "DeepSeek-V3 Technical Report," *arXiv preprint arXiv:2412.19437*, 2024. [Online]. Available: https://arxiv.org/abs/2412.19437

[4] P. Tillet, H. T. Kung, and D. Cox, "Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations," MAPL@PLDI 2019, 2019, https://github.com/openai/triton.

[5] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: Composable Transformations of Python+NumPy Programs," http://github.com/google/jax, 2018, version 0.3.13.

[6] NVIDIA Corporation, "NCCL: Optimized Primitives for Collective Multi-GPU Communication," https://developer.nvidia.com/nccl, 2023, accessed: 2025.

[7] ——, "NVSHMEM: Scalable Communication Library for NVIDIA GPU Clusters," https://developer.nvidia.com/nvshmem, 2020, accessed: 2025.

[8] ——, "NVSHMEM 3.0 Programming Guide," https://docs.nvidia.com/nvshmem/, 2023, accessed: 2025.

[9] S. Jeaugey, J. Bachan, P. Markthub, Z. He, S. Das, and F. Ghodsian, "Fusing Communication and Compute with New Device API and Copy Engine Collectives in NVIDIA NCCL 2.28," https://developer.nvidia.com/blog/fusing-communication-and-compute-with-new-device-api-and-copy-engine-collectives-in-nvidia-nccl-2-28/, November 2025, accessed: 2025.

[10] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[11] L. Zheng, L. Yin, Z. Xie, J. Huang, C. Sun, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez, C. Barrett, and Y. Sheng, "SGLang: Fast Serving Framework for Large Language Models and Vision Language Models," 2024.

[12] OpenSHMEM Specification Committee, "OpenSHMEM Application Programming Interface, Version 1.0," OpenSHMEM, Tech. Rep., 2012, http://openshmem.org/.

[13] S. Potluri, A. Goswami, D. Rossetti, C. J. Newburn, M. G. Venkata, and N. Imam, "GPU-Centric Communication on NVIDIA GPU Clusters with InfiniBand: A Case Study with OpenSHMEM," in *24th IEEE International Conference on High Performance Computing, HiPC 2017, Jaipur, India, December 18-21, 2017*, 2017, pp. 253–262. [Online]. Available: https://doi.org/10.1109/HiPC.2017.00037

[14] M. G. Venkata, N. Imam, S. Pophale, and T. M. Mintz, "Exploring OpenSHMEM Model to Program GPU-Based Extreme-Scale Systems," in *OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools*. Springer, 2015, pp. 18–35.
[15] NVIDIA Corporation, "GPUDirect RDMA," https://docs.nvidia.com/cuda/gpudirect-rdma/, 2013, introduced in CUDA 5.0.
[16] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. K. Panda, "Efficient Inter-Node MPI Communication using GPUDirect RDMA for InfiniBand Clusters with NVIDIA GPUs," in *2013 42nd International Conference on Parallel Processing*. IEEE, 2013, pp. 80–89.
[17] K. Hamidouche, T. L. Falch, K. Beatty, T. M. Mintz, M. G. Venkata, S. Pophale, A. C. Elster, and N. Imam, "GPU Initiated OpenSHMEM: Correct and Efficient Intra-Kernel Networking for dGPUs," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 488–498.
[18] E. Agostini, D. Rossetti, and S. Potluri, "GPUDirect Async: Exploring GPU Synchronous Communication Techniques for InfiniBand Clusters," in *Journal of Parallel and Distributed Computing*, vol. 114. Elsevier, 2018, pp. 28–45. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731517303386
[19] A. Daoud, M. Suntinger, G. Kestor, R. Gerstenberger, T. L. Falch, T. Hoefler, and A. C. Elster, "GPUrdma: GPU-Side Library for High Performance Networking from GPU Kernels," in *Proceedings of the ACM International Conference on Computing Frontiers*. ACM, 2016, pp. 398–403.
[20] NVIDIA Corporation, "DOCA GPUNetIO Programming Guide," https://docs.nvidia.com/doca/sdk/doca+gpunetio/index.html, accessed: 2025.
[21] ——, "RDMA Technologies Comparison: InfiniBand, RoCE, iWARP," https://network.nvidia.com/, 2022, accessed: 2025.
[22] CloudSwitch, "InfiniBand vs RoCE: A Comprehensive Guide," https://www.cloudswitch.com/, 2025, accessed: 2025.
[23] NVIDIA Corporation, "GPUDirect RDMA Requirements and Recommendations," https://docs.nvidia.com/cuda/gpudirect-rdma/, 2021, accessed: 2025.
[24] D. Ren, C. Qin, K. Liu, H. Zhang, R. Guo, Y. Zhang, Z. Sun, Z. Zhang, B. Chen, Z. Hou *et al.*, "DeepSeekMoE: Towards Ultimate Expert Specialization in Mixture-of-Experts Language Models," *arXiv preprint arXiv:2401.06066*, 2024.
[25] DeepSeek-AI, "DeepEP: Efficient Mixture-of-Experts Communication Library," https://github.com/deepseek-ai/DeepEP, 2025, accessed: 2025.
[26] Perplexity AI, "pplx-kernels: Perplexity GPU Kernels for MoE Communication," https://github.com/perplexityai/pplx-kernels, 2024.
[27] T. Hoefler, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, and K. Underwood, "Remote memory access programming in mpi-3," *ACM Trans. Parallel Comput.*, vol. 2, no. 2, Jun. 2015. [Online]. Available: https://doi.org/10.1145/2780584
[28] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, "Introducing OpenSHMEM: SHMEM for the PGAS Community," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*. ACM, 2010, pp. 1–3.
[29] NVIDIA Corporation, "DOCA: Data Center Infrastructure on a Chip Architecture," https://developer.nvidia.com/networking/doca, 2023, accessed: 2025.
[30] Z. Hu, S. Shen, T. Bonato, S. Jeaugey, C. Alexander, E. Spada, J. Dinan, J. Hammond, and T. Hoefler, " Demystifying NCCL: An In-Depth Analysis of GPU Communication Protocols and Algorithms ," in *2025 IEEE Symposium on High-Performance Interconnects (HOTI)*. Los Alamitos, CA, USA: IEEE Computer Society, Aug. 2025, pp. 48–59. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/HOTI66940.2025.00024
[31] MPI Forum, *MPI: A Message-Passing Interface Standard, Version 4.0*. University of Tennessee, 2021, https://www.mpi-forum.org/docs/.
[32] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. R. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss, Y. Shahar, S. Potluri, D. Rossetti, D. Becker, D. Poole, C. Lamb, S. Kumar, C. Stunkel, G. Bosilca, and A. Bouteiller, "UCX: An Open Source Framework for HPC Network APIs and Beyond," in *23rd IEEE Annual Symposium on High-Performance Interconnects, HOTI 2015, Santa Clara, CA, USA, August 26-28, 2015*, 2015, pp. 40–43. [Online]. Available: https://doi.org/10.1109/HOTI.2015.13
[33] "Unified Collective Communication (UCC) Library," https://github.com/openucx/ucc.
[34] S. Rajbhandari, C. Li, Z. Yao, M. Zhang, R. Y. Aminabadi, A. A. Awan, J. Rasley, and Y. He, "DeepSpeed-MoE: Advancing Mixture-of-Experts Inference and Training to Power Next-Generation AI Scale," in *Proceedings of the 39th International Conference on Machine Learning (ICML)*, 2022, pp. 18 332–18 346.
[35] J. He, J. Zhai, T. Antunes, H. Wang, F. Luo, S. Shi, and Q. Li, "FasterMoE: Modeling and Optimizing Training of Large-Scale Dynamic Pre-Trained Models," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2022, pp. 120–134.
[36] C. Hwang, W. Cui, Y. Xiong, Z. Yang, Z. Liu, H. Hu, Z. Wang, R. Salas, J. Jose, P. Ram, J. McKenzie, M. Kamani, S. Venkatasubramanian, C. Zhang *et al.*, "Tutel: Adaptive Mixture-of-Experts at Scale," *Proceedings of Machine Learning and Systems (MLSys)*, vol. 5, 2023.