CXL-CCL: Inter-Node Collective GPU-Communication Using a CXL Shared Memory Pool

Dong Xu
University of California, Merced
Merced, California, USA
dxu17@ucmerced.edu

Han Meng
University of California, Merced
Merced, California, USA
hanmeng@ucmerced.edu

Xinyu Chen
Zhejiang University
Hangzhou, China
xy.chen@zju.edu.cn

Dengcheng Zhu
ByteDance
San Jose, California, USA
dengcheng.zhu@bytedance.com

Wei Tang
ByteDance
San Jose, California, USA
tangwei.0101@bytedance.com

Fei Liu
ByteDance
San Jose, California, USA
fei.liu@bytedance.com

Liguang Xie
ByteDance
Seattle, Washington, USA
liguang.xie@bytedance.com

Wu Xiang
ByteDance
Beijing, China
xiangwu@bytedance.com

Rui Shi
ByteDance
Beijing, China
shirui@bytedance.com

Yue Li
ByteDance
San Jose, California, USA
theyueli@bytedance.com

Henry Hu
ByteDance
San Jose, California, USA
henry.hu@bytedance.com

Hui Zhang
ByteDance
San Jose, California, USA
hui.zhang@bytedance.com

Jianping Jiang
Xconn-Tech
San Jose, California, USA
jp.jiang@xconn-tech.com

Dong Li
University of California, Merced
Merced, California, USA
dli35@ucmerced.edu

Abstract

Large language models (LLMs) training or inference across multiple nodes introduces significant pressure on GPU memory and interconnect bandwidth. The Compute Express Link (CXL) shared memory pool offers a scalable solution by enabling memory sharing across nodes, reducing over-provisioning and improving resource utilization. We propose CXL-CCL, a collective communication library, leveraging the CXL shared memory pool to support cross-node GPU operations without relying on traditional RDMA-based networking. Our design addresses the challenges in synchronization, data interleaving, and communication parallelization faced by using the CXL shared memory pool for collective communications. Evaluating on multiple nodes with a TITAN-II CXL switch and six Micron CZ120 memory cards, we show that CXL-CCL achieves highly efficient collective operations across hosts, demonstrating CXL's potential for scalable, memory-centric GPU communication. Our evaluation demonstrates that CXL-CCL achieves average performance improvements of 1.34× for AllGather, 1.84× for Broadcast, 1.94× for Gather, and 1.07× for Scatter, compared to the original RDMA-based implementation over 200 Gbps InfiniBand. In addition, an LLM training case study shows 1.11× speedup compared with the InfiniBand while reducing interconnect hardware cost by 2.75×.

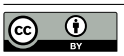
CCS Concepts

• **Computer systems organization** → **Distributed architectures**; • **Computing methodologies** → *Distributed computing methodologies.*

Keywords

CXL shared memory pool, collective communication, GPU communication

ACM Reference Format:
Dong Xu, Han Meng, Xinyu Chen, Dengcheng Zhu, Wei Tang, Fei Liu, Liguang Xie, Wu Xiang, Rui Shi, Yue Li, Henry Hu, Hui Zhang, Jianping Jiang, and Dong Li. 2026. CXL-CCL: Inter-Node Collective GPU-Communication Using a CXL Shared Memory Pool. In *2026 International Conference on Supercomputing (ICS '26), July 06–09, 2026, Belfast, United Kingdom.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3797905.3807846



ICS '26, Belfast, United Kingdom

ACM ISBN 979-8-4007-2522-7/2026/07
https://doi.org/10.1145/3797905.3807846

1 Introduction

Large Language Models (LLMs) continue to grow in architectural complexity, driving major advances in natural language processing and generative AI [8, 20, 47]. As a result, large-scale training increasingly relies on multi-GPU configurations distributed across

multiple servers, where efficient collective communication and cross-node synchronization are essential for sustaining throughput, minimizing idle time, and achieving scalable performance.

In this distributed setting, collective communication plays a central role in maintaining both efficiency and consistency across GPUs and nodes [30, 31, 33, 42, 45]. NVIDIA's NCCL library [31] provides highly optimized implementations of key collectives—such as all-reduce, broadcast, all-gather, and reduce-scatter—which are heavily used in data-parallel and model-parallel training and inference. For instance, data-parallel training depends on all-reduce to synchronize gradients after each backward pass, while model parallel approaches rely on all-gather and reduce-scatter to exchange activations or parameter shards. Architectures like Mixture of Experts (MoE) [9, 10, 29] further introduce all-to-all communication to route and aggregate token batches across distributed expert layers. These communication patterns demand high-bandwidth, low-latency communication to avoid bottlenecks at scale. NCCL enables efficient use of high-speed interconnects such as NVLink and InfiniBand [32], making it a foundational component of scalable LLM training and inference. However, these technologies also introduce challenges, including high infrastructure cost, limited availability across cloud and on-premise environments, and increased software-stack complexity—particularly when coordinating NCCL, MPI, and framework-level backend. Effectively managing these challenges is crucial to unlocking the full scalability potential of large distributed training and inference systems.

The emergence of the Compute Express Link (CXL) [2] creates opportunities for building high-performance, shared memory pools. By offering a direct, low-latency load/store interface, CXL enables CPUs and GPUs to access remote memory with efficiency approaching that of local memory. Recent systems such as Beluga [70] and TraCT [73] leverage the CXL shared memory pool to store generated KV caches in LLM, allowing decoding servers to fetch required data directly from the pool. In these designs, the CXL memory pool primarily serves as a shared long-term storage layer.

In this paper, we argue that using a CXL shared memory pool for collective communication represents an additional—and high-performance—use case. Beyond providing a unified memory space across nodes, the CXL shared memory pool effectively acts as an interconnect fabric among nodes and GPUs. By exploiting memory semantics defined in the CXL protocol, collective operations can write data directly into the pool and later read the required data without relying on a complex, layered communication stack. This simplifies data movement and coordination across nodes.

However, designing efficient collective communication mechanisms atop a CXL shared memory pool introduces three challenges. *First*, current nodes do not support fine-grained cache-line interleaving in the CXL shared memory pool as in DRAM. As a result, concurrent accesses from multiple ranks may contend for the same CXL device, preventing full utilization of the aggregate bandwidth offered by the CXL switch [60]. *Second*, the paradigm of shared memory pool for communication naturally introduces dependency between the sender and receiver ranks, which limits communication and computation parallelisms we can leverage to improve performance. *Third*, there is limited support for cross-node synchronization in the CXL shared memory pool, making it difficult to establish communication coordination. As a result, when a collective operation begins, each communication rank lacks a reliable method to determine data availability, causing incorrect behavior and poorly timed reads.

To address the above challenges, we propose CXL-CCL, a collective communication library built for the CXL shared memory pool. First, CXL-CCL incorporates two software-level interleaving techniques, aiming to maximize utilization of the bandwidth provided by the CXL switch. Second, CXL-CCL enables fine-grained data partitioning, and hence enables overlap between dependent write and read operations within the memory pool, allowing communication tasks to proceed with higher concurrency and reduced idle time. Third, CXL-CCL introduces a lightweight in-memory locking mechanism, referred to as the doorbell mechanism. By using a computation-driven doorbell allocation strategy, CXL-CCL avoids maintaining heavyweight metadata and reduces redundant memory-pool accesses during lock acquisition.

The major contributions of this paper are summarized as follows:

- To the best of our knowledge, this is the first work to use a CXL shared memory pool to perform collective communication across GPUs located on different nodes. This work introduces a fundamentally new method for high-performance collective communication.
- We develop CXL-CCL, a collective communication library for GPUs that leverages a realistic CXL-based shared memory pool. We characterize the performance of this new memory architecture from the perspective of memory pool-based communication, which is unprecedented.
- We evaluate CXL-CCL on a multi-node GPU cluster. Our evaluation results indicate that CXL-CCL achieves 1.34× speedup for AllGather, 1.84× for Broadcast, 1.94× for Gather, 1.07× for Scatter, 1.5× for AllReduce, 1.43× for ReduceScatter, 1.70× for Reduce, and 1.53× for AlltoAll, averaged over varying message sizes, compared to the original RDMA-based implementation over 200 Gbps InfiniBand. Our LLM training case study shows 1.11× speedup compared with using RDMA-based implementation over 200 Gbps InfiniBand, while reducing interconnect hardware cost by 2.75×..

2 Background

2.1 GPU Collective Communication Library: NCCL

NVIDIA Collective Communications Library (NCCL) is a high performance library designed for optimized GPU-to-GPU communication. It plays a critical role in scaling AI and HPC workloads across multiple GPUs and nodes by accelerating collective communication primitives such as AllReduce, Broadcast, and Reduce. NCCL uses topology-aware algorithms to deliver high bandwidth and low latency over interconnects including NVLink, PCIe, and Ethernet. By exposing efficient low-level communication primitives, NCCL enables deep learning frameworks like PyTorch [3] and TensorFlow [4] to orchestrate data movement effectively, making large-scale distributed training both feasible and highly performant. We use the terms "node" and "server" interchangeably in the rest of the paper.

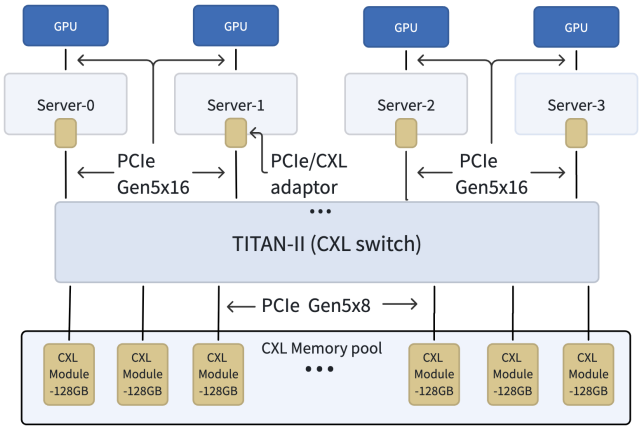


Figure 1: Architecture of the CXL shared memory pool.

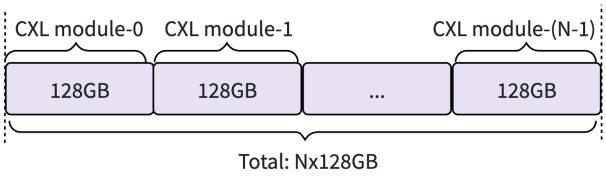


Figure 2: Sequentially stacked memory address space.

Listing 1: Pseudo code for mapping a CXL pool via DAX and memory pinning for GPU DMA

```
1  fd = open("/dev/dax0.0", O_RDWR);
2  base = mmap(NULL, size, PROT_READ | PROT_WRITE,
       MAP_SHARED,fd, offset);//mapping
3
4  ptr  = (uint8_t*) base;  // raw byte-address space
5  addr = ptr + alloc_offset;// manual allocation
6  addr[0] = value;        // CPU store
7  x = addr[0];            // CPU load
8
9  cudaHostRegister(base, size, cudaHostRegisterDefault);
       // pin (page-lock) for GPU DMA
10
11 cudaMemcpyAsync(gpu_buf, addr, bytes,
       cudaMemcpyDefault, stream);//CXL<->GPU example
12
13 munmap(base, size);
14 close(fd);
```

2.2 Compute Express Link (CXL)

CXL is an emerging interconnect protocol designed to enable efficient memory sharing between host CPUs and accelerators. It encompasses three complementary sub-protocols: `CXL.io`, which handles device management and DMA transfers; `CXL.cache`, which provides coherent device accesses to host memory; and `CXL.mem`, which allows the host to perform load/store operations on device-attached memory for expansion and pooling. The CXL specification 3.0 further strengthens these capabilities by introducing switching and enhanced cache coherency, making large-scale memory sharing and pooling more efficient and practical.

This paper focuses on memory expansion solutions built on the `CXL.mem` sub-protocol. At present, the hardware design and production for `CXL.mem` remain in an early stage. Most commercially available devices support only CXL 2.0 which enables memory expansion within a single server. Xconn-tech [60] has introduced the first commercial CXL 2.0 switch, offering `CXL.mem` interfaces with 256 lanes, multi-host concurrent access, a minimal 64-byte I/O latency of 658ns, and a maximum switching bandwidth of 2 TB/s. The emergence of CXL 2.0 switches marks a significant step forward, extending the protocol beyond the single-node memory expansion model in CXL 1.1 and enabling the construction of large-scale shared memory pools.

Figure 1 illustrates our system for study. All servers in the figure are connected to an Xconn-tech CXL switch through a Gen5x16 cable. Besides the servers, there is a memory-pool box encapsulating multiple CXL cards. Each card is connected to the switch via a Gen5x8/Gen5x16 cable.

Sharing through DAX. A CXL shared memory pool can serve as a memory resource accessible to multiple servers. However, current Linux kernels do not provide a native NUMA mode that allows multiple hosts to treat shared CXL memory as standard system memory. Consequently, the shared pool is typically exposed through Direct Access (DAX), allowing applications to map it directly into their address space. Using DAX, the processes access the shared CXL region with load/store semantics while bypassing the page cache, making it a practical mechanism for inter-node CXL memory sharing given the limitation of current operating system.

Sequentially stacked memory addresses. A common approach to scaling a CXL memory pool is to stack multiple CXL memory cards so that their capacities accumulate. In this configuration, each card contributes a fixed amount of memory capacity, and the system exposes the combined capacity as a single, contiguous address space. Sequential stacking builds the pool by placing each device's memory range directly after another one in a predetermined order. For instance, with six CXL devices each providing 128 GB, the total pool size becomes 768 GB: the addresses [0, 128 GB) map to device 0, [128 GB, 256 GB) to device 1, and so on, until [640 GB, 768 GB) to device 5, shown in Figure 2.

Accessing the CXL shared memory pool follows a workflow, illustrated in Listing 1. The listing shows how a host process uses the DAX (Device-DAX) interface to map the shared pool and prepare the region for GPU DMA transfers. At Line 1, the process opens the DAX character device (e.g., /dev/dax0.0) with read/write permissions, treating the shared pool as a file-like resource. Line 2 maps a *size*-byte window of the device to the virtual address space of the process through mmap, producing a CPU-accessible pointer ("base") at an aligned offset. After mapping, the code performs manual layout management within the region: it casts "base" to a raw byte pointer (Line 4) and computes an application-specific address ("addr") by adding an allocation offset (Line 5). The host can then issue memory loads and stores to the shared pool (Lines

Table 1: MLC results.

	Local DRAM	CXL memory pool
Latency	214ns	658ns

6–7), effectively reading and writing shared data buffers. To support efficient GPU accesses, the mapped pages are pinned using "cudaHostRegister" (Line 9), ensuring they are page-locked and suitable for high-throughput DMA. The host can then launch asynchronous transfers between the GPU buffer and the shared-pool address using "cudaMemcpyAsync" (Line 11), enabling GPU-driven communication through the shared memory pool. Once the memory operations complete, the process unmaps the region with "munmap" (Line 13) and closes the device handle (Line 14).

3 Performance Characteristics of the CXL Shared Memory Pool

We present our evaluation using microbenchmarks that measure memory latency and bandwidth to drive the design of CXL-CCL. The evaluation covers four aspects: (1) CPU access latency to the CXL-backed address space, (2) GPU↔CXL data transfer bandwidth, (3) concurrent read/write streams directed to the same CXL device to examine contention, and (4) concurrent streams targeting different CXL devices to evaluate scalability. Together, these results characterize the fundamental overheads of using the CXL shared memory pool and the achievable throughput under both single-stream and multi-stream workloads. Section 5.1 shows the hardware setup for our test.

CPU access latency to the CXL shared memory pool. We use Intel Latency Checker (MLC) to measure the access latency of local DRAM (local NUMA node). As shown in Table 1, the CXL memory pool shows 3.1× higher latency than local DRAM. This difference is expected: DRAM is directly attached to the CPU's memory controller, while the CXL memory is accessed through the PCIe/CXL link and CXL switch, introducing additional hops and protocol overhead. Despite the higher latency, the CXL pool remains valuable for capacity expansion and for data that do not lie on the critical per-instruction path, particularly when access patterns are streaming or batched.

GPU↔CXL data transfer bandwidth. To measure GPU↔CXL data transfer bandwidth, the CXL memory pool is first mapped into each server's user space through DevDAX (e.g., `/dev/dax*`). A fixed-size region is then pinned using `cudaHostRegister`, allowing it to serve as a CUDA host buffer for DMA transfers. The bandwidth is evaluated using timed, repeated `cudaMemcpyAsync` operations between a GPU buffer and the pinned CXL-mapped region, with warmup iterations included to eliminate one-time initialization overheads. Both transfer directions are measured—GPU→CXL (writes to the pool) and CXL→GPU (reads from the pool). The transfer size is swept from small to large (e.g., MB to GB) to capture both latency-dominated and bandwidth-dominated behaviors, and the resulting bandwidth is reported as "bytes transferred/time".

The evaluation proceeds in two stages. First, a single-server test measures peak bandwidth when only one server transfers data to or from the CXL shared pool, providing a best-case estimation of per-server bandwidth without contention. Second, a two-server test runs the same benchmark concurrently on both servers, with configurations where both issue GPU→CXL writes, both issue CXL→GPU reads, or one reads while the other writes. This setup reveals how the shared pool and switch fabric behave under cross-server contention and whether bandwidth scales, saturates, or degrades with concurrent accesses from multiple servers.

Figure 3a presents the measured bandwidth with exclusive accesses to the memory-pool devices. We observe that: (1) although the GPU is connected to the server via a PCIe Gen5 x16 link, the maximum sustained bandwidth for excessive read and write operations is constrained by the performance of the CXL memory module, which is attached through a PCIe Gen5 x8 interface. For relatively large message sizes (e.g., 1 MB), the achievable bandwidth approaches approximately 20 GB/s. We additionally conduct an experiment that issues multiple read or write requests to multiple CXL memory devices via multiple concurrent streams. Nevertheless, the aggregate peak bandwidth of all these requests does not exceed the peak bandwidth reported in Figure 3a. We found that the primary cause is a hardware limitation of GPU, which is equipped with only a single DMA engine per data-transfer direction. Consequently, even in the presence of multiple CXL devices, a single GPU cannot fully saturate its PCIe link.

Observation 1: Bandwidth increases with message size, reaching 20GB/s for 1MB transfers. However, performance is limited by a single CXL device on PCIe Gen5 x8 and the GPU's single DMA engine per direction, which prevents full utilization of the PCIe Gen5 x16 link, even with multiple CXL devices.

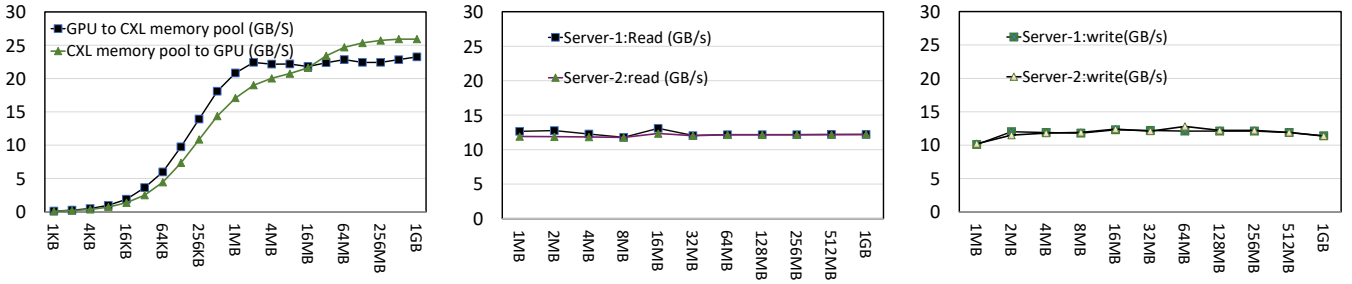
Figure 3b and Figure 3c illustrate concurrent read and write operations initiated from multiple servers. Results for message sizes below 1 MB are omitted because, during the evaluation, the data transfer time for smaller messages was so short that it was not possible to reliably initiate read and write operations concurrently. From the results, we observe that when multiple requests are issued to the same CXL device, the available bandwidth is distributed approximately evenly among the requests. This behavior is likely attributable to the scheduling or arbitration policy implemented by the CXL device's internal controller.

Observation 2: Issuing concurrent, similar requests to the same CXL device can substantially degrade the effective performance perceived by each individual server.

4 Design of CXL-CCL

4.1 Overview

The traditional GPU collective communication has limitations. Figure 4 illustrates the traditional copy–RDMA pipeline used for send or receive operations between two GPUs. This send–receive primitive serves as the foundational communication unit for building higher-level collective communication algorithms such as the ring-based one [55]. The CPU will be responsible for launching the GPU kernel (including the whole send–receive pipeline) for each rank. This pipeline introduces several limitations. First, it consumes GPU compute resources (streaming multiprocessors) and HBM bandwidth to copy data between the user buffer and the FIFO buffer associated with the pipeline. These copy operations contend with concurrent GPU computation, degrading performance and forcing



(a) Data transfer bandwidth, evaluated with single node-exclusive accesses.

(b) Concurrent GPU reads from the memory pool.

(c) Concurrent GPU writes to the memory pool.

Figure 3: Performance characterization of the CXL shared memory pool. X-axis represents the transferred data volume. Y-axis represents the bandwidth(GB/s).

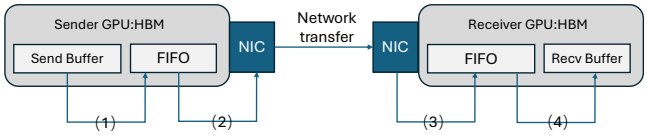


Figure 4: The traditional copy-RDMA communication pipeline in NCCL.

the users to balance resources between communication and compute. Second, the pipeline requires data to be partitioned and transferred through multiple RDMA requests, each forming a stage of the copy–RDMA pipeline. Pipelining necessitates frequent GPU–CPU synchronization, as the CPU must intervene to verify kernel completion before dispatching the subsequent RDMA request. This introduces a control-plane overhead at every stage, effectively serializing the task-handover process. More critically, this restricts each RDMA operation to a single data chunk. Finally, each pipeline is managed by a dedicated thread block in GPU, or "Channel". This static binding requires allocating a separate FIFO buffer for every independent pipeline. As a result, increasing the number of thread blocks to accelerate a collective operation proportionally increases GPU memory consumption.

With a CXL shared memory pool, data access becomes analogous to accessing local DRAM. As a result, developers are relieved from managing low-level network protocols, complex work-request preparation, and performance tuning typically required for RDMA based communication. Moreover, eliminating CPU-GPU synchronization simplifies coordination between data movement and computation.

Listing 2 presents the core structure of a communication primitive. The shared memory pool is mapped and registered in the CUDA address space, enabling direct memory transfers between the node and CXL device. Execution begins by writing data from GPU memory to the memory pool using `cudaMemcpy` with the flag `cudaMemcpyDeviceToHost`. After the write completes, a synchronization ensures that all participating tasks have finished their data transfers before proceeding. If additional operations—such as reduction or aggregation—are required, data is then read back from the

Listing 2: Core structure of a collective communication primitive in CXL-CCL.

```
void primitive_collective_comm_task(...){

    // memory pool is already mapped and registered
        into CUDA space.
    ...
    // write the data to the pool
    cudaMemcpy(dst, src,size, cudaMemcpyDeviceToHost);

    synchronization ;

    // do the read and reduce operation if needed
    cudaMemcpy(dst, src,size, cudaMemcpyHostToDevice);

    // do the reduction operation if needed
}
```

memory pool into GPU memory using `cudaMemcpyHostToDevice`. This method provides flexible and modular support for collective operations by decoupling communication from computation while maintaining explicit control over memory transfers.

In CXL-CCL, each node has a unified address space with simplified space control. During startup, the BIOS on each node detects the attached CXL devices and reserves a contiguous physical address space for them. CXL-CCL then leverages this foundation by having each node manage the CXL memory pool in DAX mode. In this mode, the CXL memory is exposed as a block device, allowing user-space processes to use `mmap()` to map the entire memory region directly into their virtual address spaces. This direct-mapping mechanism is the key to both resource partitioning and data sharing. Crucially, this approach provides a unified memory view across all nodes, facilitating simple and efficient memory management.

4.2 Challenges

The design of CXL-CCL faces three challenges.

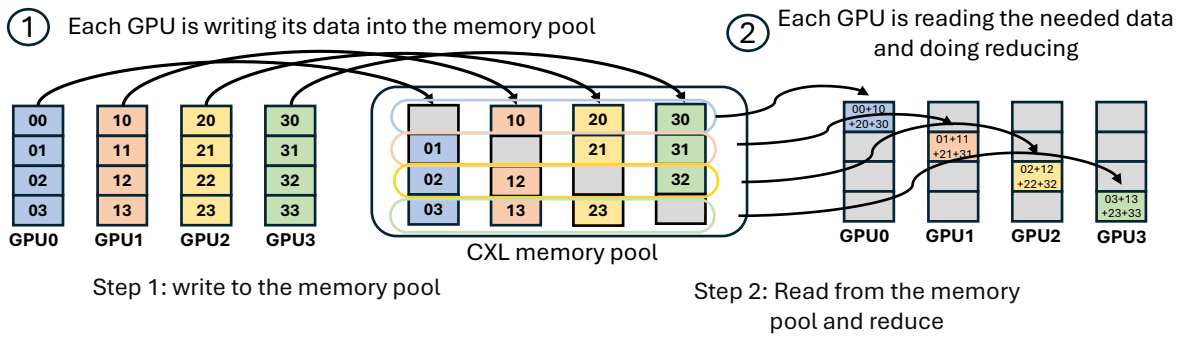


Figure 5: An example: ReduceScatter with four GPUs via a CXL shared memory pool.

1. Support for fine-grained interleaving at the cache-line level is not available. The CXL shared memory pool can scale capacity by stacking multiple devices, but the memory pool lacks the hardware-managed, cache-line–level interleaving found in conventional multi-channel DRAM systems, which leads to load imbalance across devices and limits memory bandwidth. In the traditional DRAM, fine-grained interleaving automatically stripes adjacent cache lines across memory channels, balancing requests and maximizing parallel bandwidth. In contrast, a CXL memory pool exposes each device as an independent target, so data placement is not automatically distributed across devices. Without an explicit placement mechanism, concurrent accesses from nodes or ranks may converge to the same device, creating hot spots, severe contention, and highly variable latency. As a result, the system cannot fully exploit the memory pool's aggregate bandwidth, and overall performance can degrade substantially. To address this limitation, we design a data placement scheme that proactively distributes data and accesses across devices, reducing contention and improving utilization of available memory-level parallelism.

2. Sequential data publication and retrieval. Using a shared memory pool for collective communication naturally introduces read-after-write (RAW) dependency: a rank must first publish their data to the pool, after which other ranks can safely retrieve it to perform the communication. RAW enforces a strict order on memory operations and limits communication performance. To improve the performance, CXL-CCL partitions communication data into smaller segments and leverages a doorbell mechanism to establish execution order. This method overlaps data publication and retrieval, enabling them to proceed asynchronously and improving performance.

3. Difficulty in supporting read-after-write. In CXL-CCL, each rank independently writes its local data into the shared memory region, and other ranks subsequently read the required data to complete the collective communication operation. On a single node, RAW correctness is naturally respected by hardware cache coherence. However, the CXL hardware does not provide full cache coherence across nodes or CXL devices. Consequently, a rank cannot reliably determine when another rank's write has completed and become globally visible. Without a RAW guarantee, cross-rank synchronization is difficult to establish, and the consumer may observe stale or partially written data. To address this limitation, an explicit synchronization and notification mechanism is required to signal data readiness before reads can safely proceed. We introduce a lightweight doorbell mechanism on the memory pool that allows the consumer to wait for specific data to become available, ensuring correctness with minimal overhead.

CXL-CCL has three major components, discussed as follows.

4.3 Bandwidth Aggregation

The CXL shared memory pool provides a scalable interface between host processors and memory devices. Both capacity and bandwidth can be expanded by integrating additional CXL devices into the memory enclosure. While this increases aggregate bandwidth, the pool does not support fine-grained, cacheline-level interleaving across devices as conventional multi-channel DRAM does. As illustrated in Figure 3b and Figure 3c, when two nodes simultaneously access the same device, those memory accesses converge on the same CXL device, and each node effectively receives only half of that device's available bandwidth. To fully exploit the aggregated bandwidth, we introduce a software-level interleaving mechanism to enable data distribution across CXL devices.

The conventional approach for interleaving often relies on the maintenance of a large pool of memory blocks, from which each rank dynamically acquires a block when it needs to write data [1, 11]. However, this approach necessitates the management of substantial metadata and the support of parallel allocation, since all ranks may begin writing their data into the pool concurrently once a request is issued. Furthermore, it requires careful placement of allocations across different devices to minimize concurrent read/write conflicts. In CXL-CCL, we propose an alternative mechanism: preallocated, model-guided memory regions for each rank. The key observation is that collective communications exhibit regular and predictable access patterns, which can be systematically characterized and therefore proactively arranged across CXL devices. We discuss our model (or formula) to guide the data placement or interleaving across CXL devices as follows.

Variables. We define the following variables to illustrate our interleaving method.

- ND: the number of CXL memory devices in the pool.
- DS: the capacity of each CXL memory device.

- $rank\_id$: the identifier assigned to each process in a communicator. It is an integer from 0 to $N-1$, where N is the total number of processes in the communicator.
- $Data\_id$: the logical id of the data. This ID is a logical identifier used to refer to data in the $sendBuffer$. Its range depends on whether the data is sent to a single rank or distributed across multiple ranks. For example, in Figure 5, with 4 ranks, the $sendBuffer$ is divided into 4 segments, each destined for a different rank. In this case, data_id ranges from 0 to 3.
- DB_{Offset}: the size of the preallocated doorbell buffer.

In a collective communication primitive, each rank maintains a sendBuffer and a recvBuffer. As illustrated in Figure 5, each rank performs the collective in two main steps. First, it writes the data from its sendBuffer into the shared memory pool. Then, it reads the required data from the pool into its recvBuffer. Throughout these steps, each rank must know both where to write its outgoing data and where to read its incoming data.

Determining the location to write the data. As shown in Table 2, the collective communication primitives can be categorized into two types: (1) 1 to N (or N to 1), and (2) N to N. For each category, we use a different interleaving method.

The following formulas determine how CXL-CCL calculates the device location to store the data from the $sendBuffer$ for each rank for the type (1).

$$\text{device\_index} = \text{data\_id}\%ND \tag{1}$$

$$\text{device\_block\_id} = \frac{\text{data\_id}}{\text{ND}} \tag{2}$$

$$\begin{aligned}\text{device\_location} = \text{DB\_offset} + \text{device\_block\_id} \times \text{block\_size} \\ + \text{device\_index} \times \text{DS}\end{aligned} \tag{3}$$

Equation 1 determines the target CXL memory device for a given data block. The term ($data\_id$ mod ND) indicates that CXL-CCL employs a round-robin strategy to interleave data block across all CXL devices based on the identifier $data\_id$. This scheme evenly distributes data stored in sendBuffer for the root rank for the type (1) collectives.

During data retrieval, each rank reads from the pool using a different offset. For instance, if the root rank is 0, rank 1 reads from `data_0`, while rank 2 reads from `data_1`. In this manner, the root rank publishes its data across all CXL devices in the memory pool, and other ranks access data from distinct devices in parallel. This design exploits the aggregated bandwidth of the memory pool and enhances parallel I/O efficiency.

Equation 2 computes the identifier of a data block within a specific device. In this context, a data block denotes a logical block within the CXL memory space. The size of this block can be determined at runtime based on the message size associated with a given collective communication request. The resulting $device\_block\_id$ is further employed as an index to retrieve the corresponding doorbell entry from the doorbell buffer (discussed in Section 4.5).

Equation 3 computes the final device location. It consists of three components: the preallocated doorbell buffer, the offset associated with the requested data blocks, and the offset corresponding to the $device\_index$ devices. By adding the mapped $baseAddr$, the resulting value specifies the address within the memory pool used for the write operation.

For the type (2) (N to N), since all ranks write and read the data. The above method could incur concurrent read and write to the same devices. Hence, we propose another interleaving method that assigns different devices to different ranks. The following formulas decide how we get the CXL memory location for a specific data.

$$\text{device\_id} = \text{data\_id}\%\text{device\_per\_rank} \tag{4}$$

In Equation 4, we introduce a variable: $device\_per\_rank$, calculated by $ND/TOTAL\_RANK$. To calculate $device\_block\_id$ and $device\_location$, we use the same computation logic as in Equations 2 and 3. The only difference is that for each rank, the assignment is limited to a mutually exclusive range of devices instead of all the devices.

Read data from the memory pool. At this step, each rank computes the memory-pool address of the required data block using Equations 1, 2 and 3. Each rank then performs the reduction operation if the corresponding collective communication includes a reduction. At this step, different ranks still read data from different devices.

Example. Figure 6 shows an example using `reduceScatter`. All ranks write their data according to the formulas described above. Each rank (GPU) will be assigned with two CXL devices. During the data publishing step, each rank writes exclusively to its own designated devices, thereby eliminating concurrent writes to the same device. For each rank, we establish a deterministic ordering of the data blocks to be published. CXL-CCL always publishes the data block starting from the $(rank\_id + 1)\%TOTAL\_RANKS$ and then goes to the next one. For instance, in Figure 6, rank 0 first publishes `data-01` to `device-0`, followed by publishing `data-02` to `device-1`. All ranks start publishing their data simultaneously. When rank 0 reads `data-30` (waiting until the data becomes available) from `device-6`, rank 3 is simultaneously writing `data-31` to `device-7`. In this manner, we avoid concurrent reads and writes targeting the same CXL device.

4.4 Asynchrony and Overlapping

In Figure 5, a straightforward approach to preserve communication correctness is to introduce an explicit synchronization barrier between Step 1 and Step 2. This ensures that all data for communication are fully written to and visible in the memory pool before any subsequent accesses occur. However, this approach enforces strictly sequential read and write operations for each rank, thereby limiting performance.

We observe that true data dependencies arise only between a producing rank and the ranks that consume its data: the producer must first write its data into the memory pool, and any consumer must wait until that write has completed. Leveraging this observation, CXL-CCL allows each rank's data publication (write) and data retrieval (read) to proceed asynchronously, as long as these inter-rank dependencies are respected.

In particular, CXL-CCL creates two distinct streams per rank: a writeStream for issuing write operations to the memory pool and a readStream for fetching data from the pool. Using these two streams enables read and write operations at each rank to execute

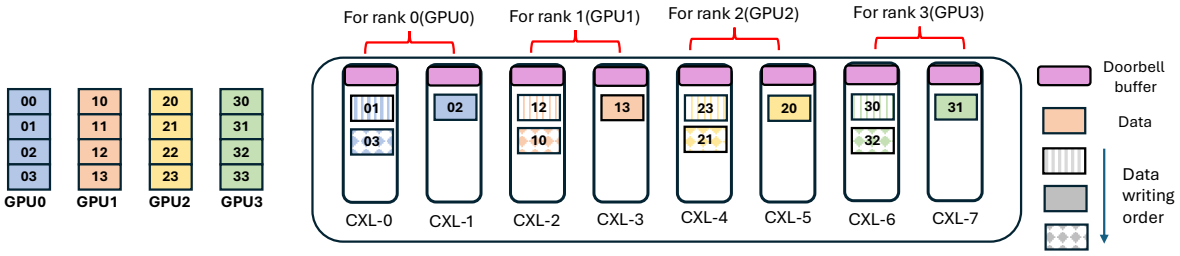


Figure 6: An example of spreading data across multiple CXL devices.

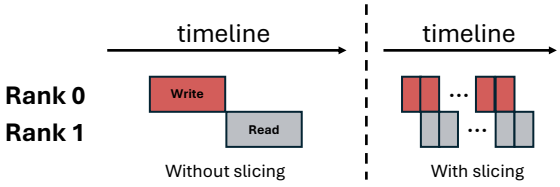


Figure 7: Communication overlapping.

in parallel without blocking one another, while still preserving the necessary order constraints between producers and consumers. In addition, CXL-CCL partitions the overall data buffer into multiple fine-grained chunks, each associated with its own doorbell for synchronization. This chunking strategy allows producers to publish data incrementally and consumers to retrieve ready chunks immediately, thereby overlapping publication and retrieval. This method increases memory-level parallelism and reduces end-to-end communication latency.

Consider a case where rank 0 publishes N bytes into the shared memory pool (see Figure 7). Without data chunking, the remote rank (rank 1) must wait for all N bytes to be produced before starting to transfer data from the shared memory pool to its local buffers, causing idling hardware and repeatedly polling the semaphore (the doorbell mechanism discussed in Section 4.5). With partitioning into smaller chunks, the consumer can start transferring as soon as a chunk is available. As shown in Figure 7, this allows rank 0's publication to overlap with rank 1's retrieval, increasing concurrency and reducing idle time.

4.5 Lightweight Locking Mechanism

To enable synchronization across nodes, we must establish a lock mechanism. Existing inter-node locking mechanisms—such as centralized lock services [74, 75] and lease-based locks [41]—provide mutual exclusion for shared data, but they rely on inter-node messaging in the critical path. This dependency introduces high latency and easily becomes a bottleneck for fine-grained synchronization. Recent work [73] proposes two-level locking, but it is primarily designed for long-lived shared data where exclusive access must be maintained for extended periods. In contrast, data involved in collective communication is short-lived and highly latency-sensitive. Maintaining heavyweight metadata or performing allocation operations to acquire locks exacerbates this latency. To address the above challenges, CXL-CCL introduces a lightweight, index-calculation-based mechanism that enables lock acquisition through a single, simple index computation.

Figure 8 generally depicts the locking mechanism in CXL-CCL with two steps. Each data chunk is associated with a dedicated semaphore stored in the shared memory pool, making it accessible to all nodes. CXL-CCL assigns update permission for this semaphore to the data owner—the rank responsible for producing the corresponding data chunk. The semaphore has two states, STALE and READY. Before the owner completes its write, the semaphore remains in the STALE state. Once the write finishes, the owner updates the semaphore to READY, signaling that the data chunk is safe for consumption. Figure 8 illustrates the above workflow.

Listing 3 shows the pseudo code in CXL-CCL library. In primitive_task(...), the code uses a doorbell (semaphore)-based synchronization to establish the producer-consumer synchronization between ranks when exchanging data through the memory pool. Lines 3–7 implement the publish (write) phase: the rank first copies the data from GPU memory into the pool using `cudaMemcpyDeviceToHost` (Line 4). After the copy completes, it selects the corresponding write doorbell entry (Line 5) and sets it to READY (Line 6) to announce that the data chunk is now valid and can be consumed. The doorbell update is then explicitly flushed (Line 7) to ensure visibility across sockets and prevent other ranks from observing stale cached values. Lines 8–13 implement the consumption (read) readiness check: the rank switches to the peer's read doorbell (Line 9) and spins until the doorbell indicates the chunk is ready (Lines 10–13). If the doorbell remains in the STALE state, the consumer forces cache coherence by cache-line flushing and re-reading (Line 11), ensuring it does not proceed with an outdated value. Here, this flushing operation only invalidates the doorbell in the cache. It does not change the status of the doorbell in the memory pool.

Finally, once the doorbell confirms readiness, another rank can safely read the corresponding data chunk from the pool and perform reduce operation (or other collective operations) if required (Lines 14–15), guaranteeing communication correctness while coordinating concurrent access among ranks.

Pre-allocated doorbell Buffers. CXL-CCL pre-allocates doorbell buffers in the CXL memory pool. This design minimizes the number of memory transactions and avoids contention or latency spikes caused by dynamic memory management. As shown in Table 1, accessing the CXL memory pool introduces 3.1× higher latency compared to local DRAM. Pre-allocating the doorbell buffers ensures that each rank can efficiently access synchronization primitives with minimal latency, leading to improved performance during collective communication.

5 Evaluation

We implement CXL-CCL on NCCL 2.28.3 with 1020 lines of code (including 30 modified lines and 980 added lines).

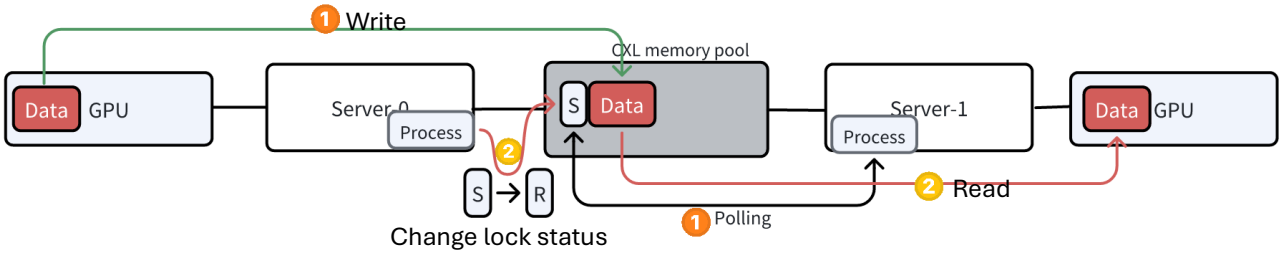


Figure 8: The workflow of locking mechanism in CXL-CCL.

Listing 3: Pseudo code for how to use the doorbell-based synchronization across nodes in CXL-CCL.

```
void primitive_task(...){
    ...
    // write the data to the pool
    cudaMemcpy(dst, src,size, cudaMemcpyDeviceToHost);
    Doorbell* db_ptr = doorbell+index_write; //
        doorbell for the write
    *db_ptr = READY; // Make the doorbell ready
    flush_doorbell(db_ptr); // flush the change
    // read
    db_ptr = doorbell+index_read; // get the doorbell
    while(*db_ptr){
        if(*db_ptr == STALE) flush_doorbell(db_ptr);
        else break;
    }
    // do the read and reduce operation if needed
    ...
}
```

5.1 Evaluation Setup

Hardware. We use three nodes, each equipped with an Intel(R) Xeon(R) 6960P CPU (72 physical cores), 256 GB of DRAM, and one NVIDIA H100 GPU with 80 GB device memory, connected through a PCIe Gen5×16 link. All three nodes connect to a TITAN-II CXL switch using PCIe/CXL Gen5×16 links. Behind the switch, a memory box provides a CXL shared memory pool using six Micron CZ120 CXL Type-3 memory cards, each with 128 GB capacity and a PCIe/CXL Gen5×8 interface, for a total pool size of 768 GB. We disable the DDIO [69] to avoid using LLC for the data transfer and set the memory pool address space as a non-cacheable area. Large-scale multi-host CXL shared memory pool platforms remain difficult to access commercially. Our evaluation platform, using realistic hardware (not emulation or simulation), represents the state of the art. CXL-CCL is intended solely for research use and is not incorporated into Bytedance's technologies.

Software. All nodes run Linux 6.2.6 with CUDA 12.8. We use nccl-tests 2.17.8 for evaluation. *ncclSendRecv* are not tested because they are point-to-point primitives.

Baseline. We use InfiniBand with 200 Gb/s as the baseline. All collective communication primitives are tested with variable message sizes, ranging from 1MB to 4GB. We categorize the primitives into two types: N to N, and N to 1 (or 1 to N). Table 2 shows the primitives we test.

We have three implementations of CXL-CCL: (1) CXL-CCL-All, which is a full-featured implementation of CXL-CCL; (2) CXL-CCL-Aggregate has bandwidth aggregation but the aggregation occurs at a coarse-granularity(at data-block level). There is no "asynchrony and overlapping". (3) CXL-CCL-Naive allocates memory in the memory pool sequentially without using memory interleaving. There is no "asynchrony and overlapping".

Scalability test. We employ an emulation-based approach to conduct the scalability evaluation because of limited access to hardware. Specifically, we develop an emulator that reproduces system performance as the number of nodes is scaled. In this emulator, we assume that concurrent read or write requests targeting the same CXL device share the available bandwidth uniformly, aligned with the results presented in Section 3. Requests directed to different CXL devices are assumed to be mutually independent, i.e., no performance interference is modeled across devices. The emulator is configured to employ a total of six CXL devices.

5.2 Overall Performance

Figure 9 reports the overall performance of CXL-CCL across NCCL primitives.

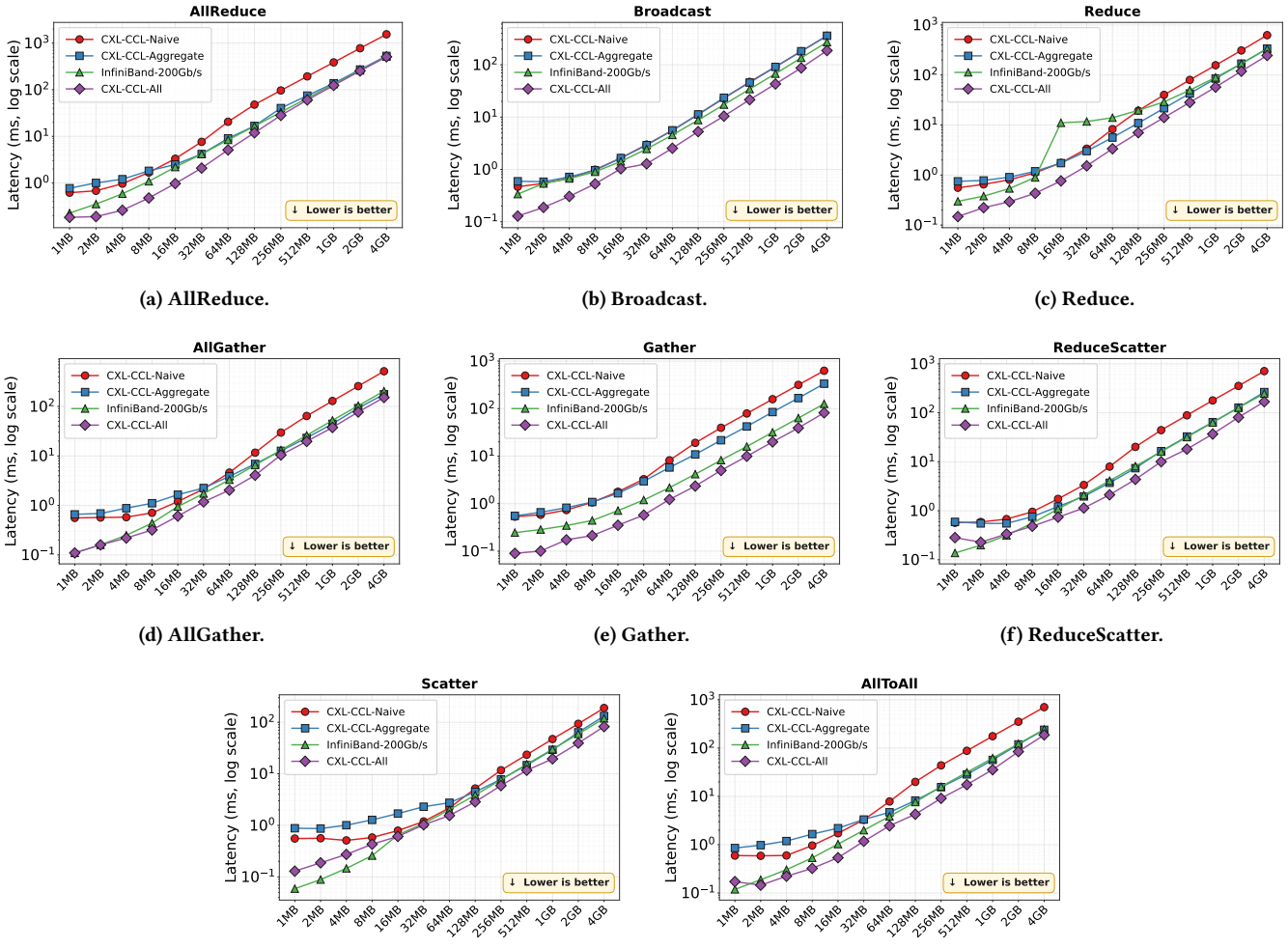
AllReduce implements an N-to-N pattern in which each rank must independently reduce data from all other ranks. Because each rank must perform its own full reduction, partially reduced results cannot be reused across ranks. In contrast, the ring-based algorithm [55] on the InfiniBand allows intermediate results to be forwarded and reused by downstream ranks. Due to this limitation, CXL-CCL-All achieves an average of only 1.05× relative performance compared with the InfiniBand when the message size goes beyond 256 MB.

Broadcast follows a 1-to-N communication pattern. Relative to CXL-CCL-Naive, CXL-CCL-Aggregate has similar performance since natively CXL-CCL-Aggregate uses a large data chunk size. However, CXL-CCL-All can reduce end-to-end latency by 1.9× –3.6×. Such improvement stems from partitioning and distributing the root's data across all CXL devices, allowing the root to write without interfering with concurrent read operations issued by other ranks. During the read phase, the ranks access disjoint CXL devices by varying their initial data-chunk offsets in the memory pool. Compared with the InfiniBand interconnect, CXL-CCL-All reduces latency by 1.3× –2.8×.

Reduce follows an N-to-1 pattern, with the root rank performing the reduction while all other ranks write their data into the memory pool. Relative to CXL-CCL-Naive, CXL-CCL-Aggregate and CXL-CCL-All reduce end-to-end latency by 0.7×–1.9× and 2.2×–3.7×,

Table 2: NCCL primitives. N denotes the buffer size per rank, and `nranks` is the number of participating ranks.

Primitive	SendSize	RecvSize	Description
`ncclAllReduce`	N	N	All ranks contribute N elements and receive reduction results.
`ncclBroadcast`	N (root only)	N (all ranks)	Root rank broadcasts N elements to all other ranks.
`ncclReduce`	N	N (root only)	All ranks send N elements to the root, which reduces and receives the result.
`ncclAllGather`	N	$N \times$ nranks	Each rank gathers N elements from every other rank.
`ncclReduceScatter`	N	N/nranks	All ranks reduce N elements; each rank receives a portion of N/nranks.
`ncclGather`	N	$N \times$ nranks (root)	Each rank sends N elements to the root which gathers all data.
`ncclScatter`	$N \times$ nranks (root)	N	Root scatters N elements to other ranks from its buffer of the size $N \times$ nranks.
`ncclAllToAll`	N	N	Each rank sends N/nranks elements to every other rank and receives the same from each.



(a) AllReduce. **(b) Broadcast.** **(c) Reduce.**

(d) AllGather. **(e) Gather.** **(f) ReduceScatter.**

(g) Scatter. **(h) AllToAll.**

Figure 9: CXL-CCL performance using the CXL shared memory pool. The x axis is the message size.

respectively. Compared with the InfiniBand, CXL-CCL-All achieves an additional 1.3×–2.6× speedup.

All-Gather follows an N-to-N communication pattern. Compared with CXL-CCL-Naive, CXL-CCL-Aggregate and CXL-CCL-All reduce end-to-end latency by 0.7×–2.9× and 1.8×–5.1×, respectively. CXL-CCL-All consistently achieves the highest performance

because it eliminates most concurrent read/write operations targeting the same CXL device, and its fine-grained interleaving strategy reduces the waiting time experienced by each rank.

Relative to the InfiniBand interconnect, CXL-CCL-All achieves 1.01×–1.6× speedup. This improvement stems from the CXL memory pool incorporating a sufficiently large number of CXL devices, providing ample aggregate bandwidth to all participating ranks.

Gather follows an N-to-1 communication pattern. With the interleaving policy and fine-grained overlapping, CXL-CCL-All achieves 4.2×–8.0× speedup over CXL-CCL-Naive and 1.1×–2.7× speedup over the InfiniBand.

ReduceScatter. Compared with AllReduce, ReduceScatter requires only a subset of data from each rank. Figure 5 illustrates the workflow: unlike AllReduce, each rank in ReduceScatter processes only its assigned portion rather than the full dataset from all other ranks. With this reduced data requirement, CXL-CCL-All achieves 1.0×–4.4× speedup over CXL-CCL-Naive and 0.48×–1.9× speedups relative to the InfiniBand. When the message size is small, we found that CXL-CCL-All perform worse than InfiniBand due to software overheads such as cudaMemcpy invocation and synchronization. CXL-CCL-All will use fine-grained chunks to transfer the message. For example, with message size equals to 1 MB, each individual transfer is much smaller than 1 MB. In this case, each rank only needs $1/nRank$ MB data from every other rank. Then for each data block ($1/nRank$ MB), CXL-CCL-All will split it into fine-grained data chunks. In this regime, software overheads such as cudaMemcpy invocation and synchronization become more pronounced. This hurts performance, compared with InfiniBand that relies on a more GPU-driven communication path. However, as the message size increases, the transfer granularity grows, so the software overhead is quickly amortized. Consequently, our approach achieves better performance than InfiniBand for larger message sizes. This trend is similarly visible in Scatter and AllToAll.

Scatter implements a 1-to-N pattern. Leveraging the interleaving and fine-grained overlapping, CXL-CCL-All delivers 1.16×–4.2× speedup over CXL-CCL-Aggregate and 0.46×–1.53× speedup over the InfiniBand.

AllToAll follows an N-to-N communication pattern. Its workflow is similar to ReduceScatter, with the key difference that AllToAll does not perform any reduction. Each rank exchanges distinct data segments with every other rank. With this pattern, CXL-CCL-All achieves 1.0×–4.3× speedup over CXL-CCL-Naive and 0.7×–1.9× speedup relative to the InfiniBand.

5.3 Scalability Evaluation

We evaluate four representative primitives for study. We scale the number of nodes from 3, 6 to 12, while the message size varies from 128 MB to 4 GB. Throughout all experiments, we employ 6 CXL devices in the memory pool. We do not scale the system to a larger number of nodes (e.g., tens or even hundreds of nodes), because the CXL shared memory pool is expected to be shared within a small number of nodes in a data center [19, 69, 72]. Figure 10 shows the results.

In AllReduce, each rank must access the data contributed by all other ranks to perform its local reduction. As the number of ranks increases, the volume of data that must be read from the shared memory pool grows proportionally. When the system scales from three to six servers, the execution time increases by 2.1×–3.0× because each rank must read roughly 2.5× more data. Additional contention also emerges due to concurrent read and write operations targeting the same set of devices, as the CXL shared memory pool contains only six CXL devices.

When the system is further expanded to twelve nodes, the total execution time rises by 8.7×–12.2×. This substantial increase is primarily due to the fact that each rank must read data from other eleven ranks, significantly amplifying communication and memory-access overhead. Moreover, with all twelve nodes simultaneously issuing both read and write requests, creating contention on shared resources becomes unavoidable and increasingly severe. Compared with InfiniBand, the current NCCL implementation can eliminate redundant reductions and data transfers within the ring, which allows NCCL over InfiniBand in this scenario to achieve better scalability.

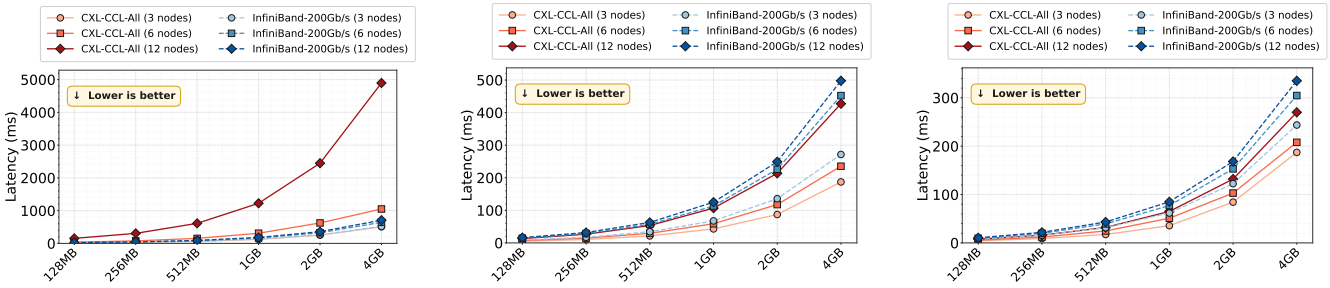
In Broadcast, only the root rank performs write operations. When scaling to six nodes, adjusting the initial read-chunk identifiers allows the read workload to be distributed across different devices for each rank. However, until the root completes its writes, one device inevitably experiences concurrent read and write traffic, since all writes originate from the root. Under this configuration, total execution time increases to 1.26×–1.40× of the three-node setup. When scaling to twelve nodes, the execution time grows to approximately 2.5× of the three-node configuration. Compared to InfiniBand, CXL-CCL-All achieves an average speedup of approximately 1.54× across all cases.

In AllToAll, each rank performs symmetric I/O, reading and writing an equivalent amount of data. Unlike AllReduce, AllToAll requires only a subset of each rank's data rather than a global aggregation. For instance, if the message size is N bytes, then each rank will receive N/nRanks bytes from every rank (including itself). Consequently, when you increase the number of nodes to 6 or 12, the total data traffic remains unchanged for the same message size N. Nevertheless, increasing the number of nodes leads to more simultaneous read/write operations on our limited CXL devices. As the system scales to six and twelve nodes, the end-to-end latency increases by 1.11×–1.43× and 1.44×–1.83×, respectively.

5.4 Sensitivity Study

We conduct a sensitivity study to evaluate how the number of data chunks used during partitioning affects performance. Using the All-Gather primitive as a representative case and set the message size equals 1 GB to illustrate the impact of different chunk configurations.

Figure 11 presents the results obtained by varying the slicing factor, i.e., the number of chunks into which the data is partitioned. For each message size, we observe that the performance changes as a function of the slicing factor. The maximum performance variation reaches approximately 9%. The configurations with very few chunks (e.g., a single chunk) yield the worst performance. This degradation arises because large chunks strengthen data dependencies between sender and receiver ranks, reducing the degree of overlap between communication and computation. As a result, the system is unable to fully utilize the available bidirectional bandwidth of the CXL memory devices, which in turn degrades overall performance. In our tests, using 4 or 8 as the number of chunks is good for performance.



(a) AllReduce. (b) Broadcast. (c) AllToAll.

Figure 10: Scalability evaluation using the CXL shared memory pool.

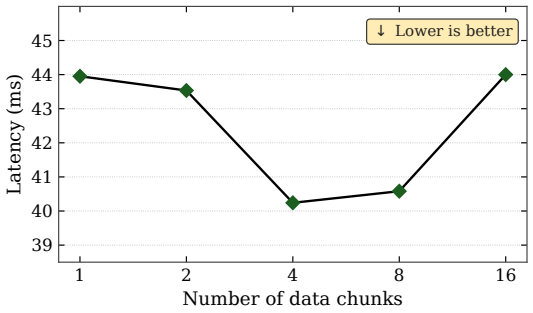


Figure 11: Sensitivity study for end-to-end latency with respect to the number of data chunks.

5.5 Using CXL-CCL for LLM Training

We evaluate CXL-CCL using a real-world LLM training scenario. Specifically, given the GPU memory limitation, we train a Llama-3-8B model on the dataset Wikipedia [12], using PyTorch's Fully sharded Data Parallelism (FSDP) to perform multi-GPU training.

FSDP uses `AllGather` and `ReduceScatter` to shuffle model parameters and to reduce gradients across distributed nodes. Using CXL-CCL-All, we achieve 1.11× speedup, compared with the baseline (RDMA over InfiniBand). Furthermore, considering the cost of constructing the interconnect, using the CXL shared memory pool costs 2.75× less than the InfiniBand, since an InfiniBand switch (supporting 200 Gbps for each port) costs $16K, and the CXL switch costs only $5.8K [69].

6 Related Work

Prior work has extensively investigated CXL-based memory architectures and their system-level implications. Early studies characterized the latency, bandwidth, and coherence behavior of CXL using FPGA prototypes or early CXL hardware, providing the first quantitative understanding of this emerging interconnect [13, 19, 22, 53]. Research on tiered memory systems [5, 15, 17–19, 24, 25, 27, 28, 34–36, 38–40, 43, 48, 56–59, 61, 62, 64–67] has also begun to examine how to achieve better performance when CXL-based memory expansions are employed. Subsequent work demonstrated the practicality of CXL in a wide range of application domains, including in-memory databases, graph processing, and deep learning workloads [6, 7, 14, 21, 23, 26, 36–38, 44, 46, 49, 50, 52, 63, 67, 68, 71].

Several studies further compared CXL with RDMA-based solutions and showed that CXL can provide superior performance or complementary benefits for memory-centric workloads by offering native load/store semantics rather than network-style message operations [49, 54]. Nevertheless, most of these efforts primarily focus on CPU-oriented memory expansion, performance characterization, or capacity-oriented shared memory usage. They do not investigate how CXL can be leveraged to support GPU-centric collective communication, nor do they address the synchronization and data placement challenges that arise when multiple GPUs across different nodes concurrently access a shared CXL memory pool.

More recently, a growing body of work explores exposing CXL devices as shared memory across multiple nodes. CXL-SHM proposes a general shared-memory substrate and demonstrates its use for distributed data structures and RDMA offload, assuming device-side atomic operations [76]. Tigon studies CXL-based shared memory for in-memory databases and shows that hardware coherence support is limited in scale and cannot be extended to the full device capacity [16]. cMPI replaces the traditional point-to-point network path in MPI communication with CXL shared memory to enable inter-node message passing [51]. These systems primarily treat CXL as a general shared-memory substrate or as an alternative communication medium for distributed applications. In contrast, this work focuses on enabling efficient GPU collective communication on top of a shared CXL memory pool. CXL-CCL directly exploits the memory semantics provided by CXL to coordinate data exchange among GPUs and explicitly addresses challenges that are unique to collective operations, including the lack of cross-node synchronization, the absence of hardware-managed fine-grained interleaving across multiple CXL devices, and the need to overlap data publication and consumption to reduce end-to-end latency.

7 Conclusions

Efficient collective GPU communication is essential for high performance AI workloads. Different from the traditional approaches that rely on RDMA and high-performance interconnect, we introduce a fundamentally new method that leverages a CXL shared memory pool for GPU collective communication. Based upon the emerging CXL-based architecture and memory copy-based communication implementation, we demonstrate the superior performance of our approach. When compared with a 200 Gbps InfiniBand, our evaluation results indicate that CXL-CCL achieves large speedup with various message sizes.

References

[1] 2020. Buddy and Slab Allocators. https://students.mimuw.edu.pl/ZSO/Wyklady/06_memory2/BuddySlabAllocator.pdf.

[2] 2026. Compute Express Link (CXL). https://computeexpresslink.org/.

[3] 2026. PyTorch. https://pytorch.org/.

[4] 2026. TensorFlow. https://www.tensorflow.org/.

[5] Neha Agarwal and Thomas F. Wenisch. 2017. Thermostat: Application-transparent Page Management for Two-tiered Main Memory. *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (2017). https://api.semanticscholar.org/CorpusID:8753499

[6] Minseon Ahn, Andrew Chang, Donghun Lee, Jongmin Gim, Jungmin Kim, Jaemin Jung, Oliver Rebholz, Vincent Pham, Krishna Malladi, and Yang Seok Ki. 2022. Enabling CXL Memory Expansion for In-Memory Database Management Systems. In *International Workshop on Data Management on New Hardware*.

[7] Moiz Arif, Kevin Assogba, M Mustafa Rafique, and Sudharshan Vazhkudai. 2022. Exploiting cxl-based memory for distributed deep learning. In *Proceedings of the 51st International Conference on Parallel Processing*. 1–11.

[8] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609* (2023).

[9] Weilin Cai, Le Qin, and Jiayi Huang. 2025. MoC-System: Efficient Fault Tolerance for Sparse Mixture-of-Experts Model Training. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25)*. ACM, 655–671. https://doi.org/10.1145/3676641.3716006

[10] Shiyi Cao, Shu Liu, Tyler Griggs, Peter Schafhalter, Xiaoxuan Liu, Ying Sheng, Joseph E. Gonzalez, Matei Zaharia, and Ion Stoica. 2025. MoE-Lightning: High-Throughput MoE Inference on Memory-constrained GPUs. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 715–730. https://doi.org/10.1145/3669940.3707267

[11] Jonathan Corbet. 2023. Weighted interleaving for memory tiering. https://lwn.net/Articles/948037/.

[12] Wikimedia Foundation. [n. d.]. *Wikimedia Downloads*. https://dumps.wikimedia.org

[13] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct access,{High-Performance} memory disaggregation with {DirectCXL}. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 287–294.

[14] Yunyan Guo and Guoliang Li. 2024. A CXL-Powered Database System: Opportunities and Challenges. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 5593–5604.

[15] Taekyung Heo, Yang Wang, Wei Cui, Jaehyuk Huh, and Lintao Zhang. 2022. Adaptive Page Migration Policy With Huge Pages in Tiered Memory Systems. *IEEE Trans. Comput.* 71, 1 (2022), 53–68. https://doi.org/10.1109/TC.2020.3036686

[16] Yibo Huang, Haowei Chen, Newton Ni, Vijay Chidambaram, Dixin Tang, Emmett Witchel, Zhiting Zhu, and Zhipeng Jia. 2025. Tigon: A distributed database for a CXL pod. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25), Boston, MA*.

[17] Yingchao Huang and Dong Li. 2017. Performance Modeling for Optimal Data Placement on GPU with Heterogeneous Memory Systems. In *IEEE International Conference on Cluster Computing*.

[18] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. MEMTIS: Efficient Memory Tiering with Dynamic Page Classification and Page Size Determination. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 17–34. https://doi.org/10.1145/3600006.3613167

[19] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. 2023. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 574–587.

[20] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).

[21] Haifeng Liu, Long Zheng, Yu Huang, Jingyi Zhou, Chaoqiang Liu, Runze Wang, Xiaofei Liao, Hai Jin, and Jingling Xue. 2024. Enabling efficient large recommendation model training with near cxl memory processing. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 382–395.

[22] Jinshu Liu, Hamid Hadian, Yuyue Wang, Daniel S Berger, Marie Nguyen, Xun Jian, Sam H Noh, and Huaicheng Li. 2025. Systematic cxl memory characterization and performance analysis at scale. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 1203–1217.

[23] Jie Liu, Bogdan Nicolae, and Dong Li. 2023. Lobster: Load Balance-Aware I/O for Distributed DNN Training. In *Proceedings of the 51st International Conference on Parallel Processing* (Bordeaux, France) *(ICPP '22)*. Association for Computing Machinery, New York, NY, USA, Article 26, 11 pages. https://doi.org/10.1145/3545008.3545090

[24] Jiawen Liu, Jie Ren, Roberto Gioiosa, Dong Li, and Jiajia Li. 2021. Sparta: high-performance, element-wise sparse tensor contraction on heterogeneous memory. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 318–333. https://doi.org/10.1145/3437801.3441581

[25] LWN.net. [n. d.]. AutoNUMA Balancing. "https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/virtualization_tuning_and_optimization_guide/sect-virtualization_tuning_optimization_guide-numa-auto_numa_balancing".

[26] Bin Ma, Xingjian Ding, Tekin Bicer, Pengfei Su, and Dong Li. 2026. CoCoDiff: Optimizing Collective Communications for Distributed Diffusion Transformer Inference Under Ulysses Sequence Parallelism. arXiv:2604.14561 [cs.DC] https://arxiv.org/abs/2604.14561

[27] Adnan Maruf, Ashikee Ghosh, Janki Bhimani, Daniela Campello, Andy Rudoff, and Raju Rangaswami. 2022. MULTI-CLOCK: Dynamic Tiering for Hybrid Memory Systems. *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)* (2022), 925–937. https://api.semanticscholar.org/CorpusID:248865268

[28] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 742–755. https://doi.org/10.1145/3582016.3582063

[29] Siyuan Mu and Sen Lin. 2026. A Comprehensive Survey of Mixture-of-Experts: Algorithms, Theory, and Applications. arXiv:2503.07137 [cs.LG] https://arxiv.org/abs/2503.07137

[30] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient large-scale language model training on GPU clusters using megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 58, 15 pages. https://doi.org/10.1145/3458817.3476209

[31] NVIDIA. [n. d.]. NVIDIA Collective Communications Library (NCCL). https://developer.nvidia.com/nccl

[32] NVIDIA Corporation. 2021. *Introduction to InfiniBand*. White Paper. NVIDIA. https://network.nvidia.com/pdf/whitepapers/IB_Intro_WP_190.pdf

[33] Pytorch. 2022. Fully sharded data parallelism. https://pytorch.org/blog/introducing-pytorch-fully-sharded-data-parallel-api/

[34] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. 2021. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (2021). https://api.semanticscholar.org/CorpusID:239029009

[35] Jie Ren, Jiaolin Luo, Ivy Peng, Kai Wu, and Dong Li. 2021. Optimizing large-scale plasma simulations on persistent memory-based heterogeneous memory with effective data placement across memory hierarchy. In *Proceedings of the ACM International Conference on Supercomputing* (Virtual Event, USA) *(ICS '21)*. Association for Computing Machinery, New York, NY, USA, 203–214. https://doi.org/10.1145/3447818.3460356

[36] Jie Ren, Jiaolin Luo, Kai Wu, Minjia Zhang, Hyeran Jeon, and Dong Li. 2021. Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 598–611.

[37] Jie Ren, Bin Ma, Shuangyan Yang, Benjamin Francis, Ehsan K. Ardestani, Min Si, and Dong Li. 2025. Machine Learning-Guided Memory Optimization for DLRM Inference on Tiered Memory. arXiv:2511.08568 [cs.PF] https://arxiv.org/abs/2511.08568

[38] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. {ZeRO-Offload}: Democratizing {Billion-Scale} model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 551–564.

[39] Jie Ren, Dong Xu, Junhee Ryu, Kwangsik Shin, Daewoo Kim, and Dong Li. 2024. MTM: Rethinking Memory Profiling and Migration for Multi-Tiered Large Memory. In *Proceedings of the Nineteenth European Conference on Computer Systems* (<conf-loc>, <city>Athens</city>, <country>Greece</country>, </conf-loc>) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 803–817. https://doi.org/10.1145/3627703.3650075

[40] Jie Ren, Minjia Zhang, and Dong Li. 2020. HM-ANN: Efficient Billion-Point Nearest Neighbor Search on Heterogeneous Memory. In *Conference on Neural Information Processing Systems (NeurIPS)*.

[41] Andre Rodriguez and William Osborn. 2025. Distributed Locking: Performance Analysis and Optimization Strategies. arXiv:2504.03073 [cs.DC] https://arxiv.org/abs/2504.03073

[42] Joshua Romero, Junqi Yin, Nouamane Laanait, Bing Xie, M. Todd Young, Sean Treichler, Vitalii Starchenko, Albina Borisevich, Alex Sergeev, and Michael Matheson. 2022. Accelerating Collective Communication in Data Parallel Training across Deep Learning Frameworks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 1027–1040. https://www.usenix.org/conference/nsdi22/presentation/romero

[43] Jee Ho Ryoo, Lizy K. John, and Arkaprava Basu. 2018. A Case for Granularity Aware Page Migration. In *Proceedings of the 2018 International Conference on Supercomputing* (Beijing, China) *(ICS '18)*. Association for Computing Machinery, New York, NY, USA, 352–362. https://doi.org/10.1145/3205289.3208064

[44] Shintaro Sano, Yosuke Bando, Kazuhiro Hiwada, Hirotsugu Kajihara, Tomoya Suzuki, Yu Nakanishi, Daisuke Taki, Akiyuki Kaneko, and Tatsuo Shiozawa. 2023. GPU graph processing on cxl-based microsecond-latency external memory. In *Proceedings of the SC'23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*. 962–972.

[45] Min Si, Pavan Balaji, Yongzhou Chen, Ching-Hsiang Chu, Adi Gangidi, Saif Hasan, Subodh Iyengar, Dan Johnson, Bingzhe Liu, Regina Ren, Deep Shah, Ashmitha Jeevaraj Shetty, Greg Steinbrecher, Yulun Wang, Bruce Wu, Xinfeng Xie, Jingyi Yang, Mingran Yang, Kenny Yu, Minlan Yu, Cen Zhao, Wes Bland, Denis Boyda, Suman Gumudavelli, Prashanth Kannan, Cristian Lumezanu, Rui Miao, Zhe Qu, Venkat Ramesh, Maxim Samoylov, Jan Seidel, Srikanth Sundaresan, Feng Tian, Qiye Tan, Shuqiang Zhang, Yimeng Zhao, Shengbao Zheng, Art Zhu, and Hongyi Zeng. 2026. Collective Communication for 100k+ GPUs. arXiv:2510.20171 [cs.DC] https://arxiv.org/abs/2510.20171

[46] Dinghong Song, Yuan Feng, Yiwei Wang, Shangye Chen, Cyril Guyot, Filip Blagojevic, Hyeran Jeon, Pengfei Su, and Dong Li. 2025. AttnCache: Accelerating Self-Attention Inference for LLM Prefill via Attention Cache. arXiv:2510.25979 [cs.CL] https://arxiv.org/abs/2510.25979

[47] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805* (2023).

[48] Vishal Verma. 2022. Tiering-0.8. https://git.kernel.org/pub/scm/linux/kernel/git/vishal/tiering.git/log/?h=tiering-0.8.

[49] Xi Wang, Jie Liu, Jianbo Wu, Shuangyan Yang, Jie Ren, Bhanu Shankar, and Dong Li. 2024. Exploring and evaluating real-world cxl: use cases and system adoption. *arXiv preprint arXiv:2405.14209* (2024).

[50] Xi Wang, Jie Liu, Shuangyan Yang, Jongryool Kim, Pengfei Su, and Dong Li. 2026. Hybrid Adaptive Tuning for Tiered Memory Systems. arXiv:2604.12165 [cs.OS] https://arxiv.org/abs/2604.12165

[51] Xi Wang, Bin Ma, Jongryool Kim, Byungil Koh, Hoshik Kim, and Dong Li. 2025. cMPI: Using CXL Memory Sharing for MPI One-Sided and Two-Sided Inter-Node Communications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2216–2232.

[52] Xi Wang, Tal Zussman, Yuang Xu, Bin Ma, Asaf Cidon, and Dong Li. 2026. TierBPF: Page Migration Admission Control for Tiered Memory via eBPF. arXiv:2604.12300 [cs.OS] https://arxiv.org/abs/2604.12300

[53] Zhonghua Wang, Yixing Guo, Kai Lu, Jiguang Wan, Daohui Wang, Ting Yao, and Huatao Wu. 2024. Rcmp: Reconstructing rdma-based memory disaggregation via cxl. *ACM Transactions on Architecture and Code Optimization* 21, 1 (2024), 1–26.

[54] Xingda Wei, Haotian Wang, Tianxia Wang, Rong Chen, Jinyu Gu, Pengfei Zuo, and Haibo Chen. 2023. Transactional indexes on (RDMA or cxl-based) disaggregated memory with repairable transaction. *arXiv preprint arXiv:2308.02501* (2023).

[55] Bryan Woolley. 2015. NCCL: Multi-GPU Collective Communication Library. https://images.nvidia.com/events/sc15/pdfs/NCCL-Woolley.pdf.

[56] K. Wu, Y. Huang, and D. Li. 2017. Unimem: Runtime Data Management on Non-Volatile Memory-based Heterogeneous Main Memory. In *International Conference for High Performance Computing, Networking, Storage and Analysis*.

[57] Kai Wu, Jie Ren Ivy Peng, and Dong Li. 2021. ArchTM: Architecture-Aware, High Performance Transaction for Persistent Memory. In *USENIX Conference on File and Storage Technologies*.

[58] Kai Wu, Jie Ren, and Dong Li. 2018. Runtime Data Management on Non-Volatile Memory-based Heterogeneous Memory for Task-Parallel Programs. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 401–413. https://doi.org/10.1109/SC.2018.00034

[59] Panruo Wu, Dong Li, Zizhong Chen, Jeffrey Vetter, and Sparsh Mittal. 2016. Algorithm-Directed Data Placement in Explicitly Managed Non-Volatile Memory. In *ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*.

[60] Xconn. 2025. Xconn Technologies. https://www.xconn-tech.com/.

[61] Zhen Xie, Wenqian Dong, Jie Liu, Ivy Peng, Yanbao Ma, and Dong Li. 2021. MD-HM: memoization-based molecular dynamics simulations on big memory system. In *Proceedings of the ACM International Conference on Supercomputing* (Virtual Event, USA) *(ICS '21)*. Association for Computing Machinery, New York, NY, USA, 215–226. https://doi.org/10.1145/3447818.3460365

[62] Zhen Xie, Jie Liu, Jiajia Li, and Dong Li. 2023. Merchandiser: Data Placement on Heterogeneous Memory for Task-Parallel HPC Applications with Load-Balance Awareness. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (<conf-loc>, <city>Montreal</city>, <state>QC</state>, <country>Canada</country>, </conf-loc>) *(PPoPP '23)*. Association for Computing Machinery, New York, NY, USA, 204–217. https://doi.org/10.1145/3572848.3577497

[63] Dong Xu, Yuan Feng, Kwangsik Shin, Daewoo Kim, Hyeran Jeon, and Dong Li. 2024. Efficient Tensor Offloading for Large Deep-Learning Model Training based on Compute Express Link. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–18. https://doi.org/10.1109/SC41406.2024.00100

[64] Dong Xu, Junhee Ryu, Jinho Baek, Kwangsik Shin, Pengfei Su, and Dong Li. 2024. FlexMem: adaptive page profiling and migration for tiered memory. In *Proceedings of the 2024 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA, USA) *(USENIX ATC'24)*. USENIX Association, USA, Article 50, 17 pages.

[65] Zi Yan, Daniel Lustig, David W. Nellans, and Abhishek Bhattacharjee. 2019. Nimble Page Management for Tiered Memory Systems. *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (2019). https://api.semanticscholar.org/CorpusID:102348046

[66] Shuo Yang, Kai Wu, Yifan Qiao, Dong Li, and Jidong Zhai. 2017. Algorithm-Directed Crash Consistence in Non-Volatile Memory for HPC. In *IEEE International Conference on Cluster Computing*.

[67] Shuangyan Yang, Minjia Zhang, Wenqian Dong, and Dong Li. 2023. Betty: Enabling Large-Scale GNN Training with Batch-Level Graph Partitioning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 103–117. https://doi.org/10.1145/3575693.3575725

[68] Shuangyan Yang, Minjia Zhang, and Dong Li. 2025. Buffalo: Enabling Large-Scale GNN Training via Memory-Efficient Bucketization. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 1066–1081. https://doi.org/10.1109/HPCA61900.2025.00083

[69] Xinjun Yang, Qingda Hu, Junru Li, Feifei Li, Yicong Zhu, Yuqi Zhou, Qiuru Lin, Jian Dai, Yang Kong, Jiayu Zhang, et al. 2025. Beluga: A CXL-Based Memory Architecture for Scalable and Efficient LLM KVCache Management. *arXiv preprint arXiv:2511.20172* (2025).

[70] Xinjun Yang, Qingda Hu, Junru Li, Feifei Li, Yicong Zhu, Yuqi Zhou, Qiuru Lin, Jian Dai, Yang Kong, Jiayu Zhang, Guoqiang Xu, and Qiang Liu. 2025. Beluga: A CXL-Based Memory Architecture for Scalable and Efficient LLM KVCache Management. arXiv:2511.20172 [cs.DC] https://arxiv.org/abs/2511.20172

[71] Xinjun Yang, Yingqiang Zhang, Hao Chen, Feifei Li, Gerry Fan, Yang Kong, Bo Wang, Jing Fang, Yuhui Wang, Tao Huang, et al. 2025. Unlocking the Potential of CXL for Disaggregated Memory in Cloud-Native Databases. In *Companion of the 2025 International Conference on Management of Data*. 689–702.

[72] Xinjun Yang, Yingqiang Zhang, Hao Chen, Feifei Li, Gerry Fan, Yang Kong, Bo Wang, Jing Fang, Yuhui Wang, Tao Huang, Wenpu Hu, Jim Kao, and Jianping Jiang. 2025. Unlocking the Potential of CXL for Disaggregated Memory in Cloud-Native Databases. In *Companion of the 2025 International Conference on Management of Data* (Berlin, Germany) *(SIGMOD/PODS '25)*. Association for Computing Machinery, New York, NY, USA, 689–702. https://doi.org/10.1145/3722212.3724460

[73] Dongha Yoon, Younghoon Min, Hoshik Kim, Sam H. Noh, and Jongryool Kim. 2025. TraCT: Disaggregated LLM Serving with CXL Shared Memory KV Cache at Rack-Scale. arXiv:2512.18194 [cs.DC] https://arxiv.org/abs/2512.18194

[74] Dong Young Yoon, Mosharaf Chowdhury, and Barzan Mozafari. 2018. Distributed Lock Management with RDMA: Decentralization without Starvation. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1571–1586. https://doi.org/10.1145/3183713.3196890

[75] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. 2020. NetLock: Fast, Centralized Lock Management Using Programmable Switches. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 126–138. https://doi.org/10.1145/3387514.3405857

[76] Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. 2023. Partial failure resilient memory management system for (cxl-based) distributed shared memory. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 658–674.