USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

SwiftEP: Accelerating MoE Inference with Buffer Fusion and TMA Offloading

Xingyi Li, *unaffiliated;* Yadong Liu and Xiaojie Huang, *Tencent;* Yiran Zhang, Shuai Wang, and Shangguang Wang, *unaffiliated;* Zhehao Lin and Yinben Xia, *Tencent;* Chang Yu, *Nanjing University;* Qihang Liu, Xuan Zhang, Hao Lu, Xiang Li, Zekun He, Yachen Wang, and Xianneng Zou, *Tencent*

https://www.usenix.org/conference/nsdi26/presentation/li-xingyi

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

SwiftEP: Accelerating MoE Inference with Buffer Fusion and TMA Offloading

Xingyi Li[♡,*], Yadong Liu[◊,*], Xiaojie Huang[◊], Yiran Zhang[♡], Shuai Wang[♡], Shangguang Wang[♡], Zhehao Lin[◊], Yinben Xia[◊], Chang Yu[†], Qihang Liu[◊], Xuan Zhang[◊], Hao Lu[◊], Xiang Li[◊], Zekun He[◊], Yachen Wang[◊], Xianneng Zou[◊]

[♡] *Unaffiliated* [◊] *Tencent* [†] *Nanjing University*

Abstract

Large Language Models (LLMs) increasingly rely on Mixture-of-Experts (MoE) architectures to scale computation efficiently. Expert Parallelism (EP), which distributes experts across GPUs, introduces all-to-all communication overhead during the *dispatch* and *combine* phases, especially in the prefill stage, which dominates the inference performance. Existing communication libraries, such as DeepEP, suffer from excessive GPU SM utilization and underutilized interconnect bandwidth, limiting prefill performance.

In this paper, we identify two root causes: redundant buffer copies and inefficient intra-server transfers over NVLink. To address these, we propose SwiftEP, an all-to-all communication library tailored for MoE prefill, combining *buffer fusion* and *Tensor Memory Accelerator (TMA) offloading*. Buffer fusion eliminates redundant staging copies, enabling true zero-copy communication, while TMA offloading maximizes NVLink utilization and supports efficient multicast/reduce operations. SwiftEP further incorporates RDMA scatter-gather lists, QP transmission parallelization, and CUDA IPC to handle dynamic token placement and inter-GPU memory access. Evaluation on 16- and 32-GPU clusters shows that SwiftEP achieves up to 119.7% higher algorithm bandwidth, reduces SM occupancy by up to 66.7%, and improves request serving capacity by 21.2% compared to DeepEP.

1 Introduction

Large Language Models (LLMs) have become the foundation of modern AI services, powering applications such as conversational agents [8, 27, 36, 59], code generation [12, 28], and information retrieval [7, 33, 57]. To improve efficiency at scale, many recent systems adopt the Mixture-of-Experts (MoE) architecture [25, 35, 53–55], which activates only a small subset of experts per input token. To fully utilize multi-GPU clusters, MoE models are typically deployed with Expert Parallelism (EP) [50], where experts are distributed across devices.

While EP attempts to utilize more computation capacity, it introduces all-to-all communication [41] in two critical operations: (i) *dispatch*, which routes tokens to their assigned experts, and (ii) *combine*, which gathers expert outputs back to the originating GPUs. As a result, communication efficiency in these two phases is critical to the end-to-end performance of MoE-based LLM inference. This challenge is especially acute during the *prefill* stage of MoE model inference, which dominates the overall Time-To-First-Token (TTFT) [20, 66], the most user-visible latency metric. Unlike the *decode* stage, the prefill stage processes thousands of tokens in parallel across all layers, amplifying the cost of dispatch and combine operations. As a result, accelerating EP communication in the prefill stage is essential to improve inference responsiveness.

Recent communication libraries, such as DeepEP [64], have introduced optimized support for EP communication, improving dispatch and combine performance. Specifically, DeepEP employs Two micro-Batch Overlapping (TBO) [15], allowing the computation and communication of two micro-batches to overlap. However, our analysis shows that recent communication libraries still face two fundamental limitations. First, they consume excessive GPU Streaming Multiprocessor (SM) resources to orchestrate communication, thereby reducing compute capacity for tensor computation operations with TBO enabled. Second, they fail to fully saturate interconnect bandwidth, leaving a substantial gap from the theoretical peak. Even with TBO enabled, the communication time becomes notably higher than the computation time, making communication the performance bottleneck and thus impacting the overall application latency. These limitations leave significant performance of both computation and communication on the table in prefill workloads.

We identify two root causes of inefficiency. 1) Buffer separation: tokens are first produced in a *compute tensor buffer* and then copied into an *RDMA send buffer* before dispatch. Moreover, if intra-server token *multicast* is used to avoid redundant inter-server communication, an *NVLink receive buffer* [30, 62] will also be involved in data copy. These fragmented, small-scale copies tie up valuable SMs. 2) Inefficient

[*] Equal contribution.

intra-server transfers: during both dispatch and combine, existing load/store (*ld*/*st*) and Copy Engine (CE) [40, 42, 45] mechanisms fail to saturate high-performance NVLink bandwidth. This limitation arises because large volumes of *ld*/*st* operations executed by SM lead to register spilling, and the synchronization latency between the CPU and GPU in the CE is high.

We argue that *overcoming these limitations requires rethinking how buffers should be managed and how intra-server communication should be executed over NVLink*. Our key insights are two-fold. First, fusing compute and communication buffers eliminates redundant staging copies, freeing SMs for computation. Second, the Tensor Memory Accelerator (TMA) [46], a hardware-driven, asynchronous tensor-copy engine, can efficiently drive intra-server data transfer.

Motivated by these observations, we propose SwiftEP, a novel all-to-all communication library tailored to the prefill stage of MoE model inference via two complementary mechanisms: *buffer fusion* and *Tensor Memory Accelerator (TMA) offloading*. Buffer fusion removes redundant data movement by directly coupling the tensor compute buffer with the RDMA send buffer for inter-server transfers and with the NVLink receive buffer for intra-server transfers. This allows tensors produced by the inference framework to be transmitted in a *true* zero-copy fashion, freeing scarce SM resources for computation. In parallel, TMA offloading unleashes the potential of high-bandwidth NVLink by delegating intra-server transfers to dedicated hardware engines. As a result, it removes the burden of issuing large volumes of ld/st operations from general-purpose SM threads, drives NVLink utilization toward its peak, and enables efficient hardware-supported multicast during dispatch and reduce during combine.

Despite these benefits, putting SwiftEP into practice raises several technical challenges. First, buffer fusion exposes small, scattered data segments in a large fused buffer, lowering transmission efficiency and quickly exhausting the limited Memory Translation Table (MTT) cache [11]. Second, eliminating intermediate copies in receivers eliminates the opportunity to reorganize tokens after reception, requiring senders to place them directly into final destinations that are unknown a priori due to dynamic token counts. Third, TMA was originally designed for intra-GPU data transfers and lacks native support for remote GPU memory, necessitating new extensions to support inter-GPU transfers.

To address these challenges, SwiftEP first leverages the RDMA Scatter-Gather List (SGL) [2, 58] technique to reduce the number of Work Queue Elements (WQEs) required for small, scattered data segments. Next, it employs Queue Pair (QP) transmission parallelization to hide multi-level address translation latency incurred by large fused buffers. To enable direct writes into the final locations of receive buffers, SwiftEP redesigns the *notify* operation to broadcast the complete token distribution to all GPUs within a server. Finally, it uses CUDA Context Inter-Process Communication (IPC) [44] to grant TMA the permissions necessary for efficient, direct memory accesses across GPUs.

We implement SwiftEP by extending the state-of-the-art MoE communication library DeepEP, requiring no changes to the inference framework other than exposing metadata of the fused buffers (*e.g.*, base addresses and memory layouts). We evaluate SwiftEP on a multi-GPU testbed with 16 and 32 GPUs (EP=16 and EP=32). Microbenchmark results show that, compared to DeepEP, SwiftEP achieves up to 93.8%/119.7% higher algorithm bandwidth for EP=16/EP=32, while reducing the number of SMs occupied by all-to-all communication by 50%/66.7%. These optimizations in bandwidth and SM utilization translate to a 21.2% higher request serving capacity when deploying SwiftEP in the DeepSeek-R1 [17] model serving environment.

This paper makes the following contributions.

- We analyze the SM overutilization and algorithm bandwidth underutilization in state-of-the-art communication libraries for EP.
- We propose SwiftEP, a novel all-to-all communication library for the prefill stage of MoE model inference, leveraging two complementary mechanisms: buffer fusion and TMA offloading.
- Our evaluation demonstrates that compared to DeepEP, SwiftEP achieves up to 119.7% higher algorithm bandwidth, reduces SM occupancy by 66.7%, and increases request serving capacity by 21.2% for the DeepSeek-R1 model.

This work does not raise any ethical issues.

2 Background

2.1 Mixture-of-Experts Models

Large language models (LLMs) achieve state-of-the-art performance but suffer from prohibitive computational costs as their scale increases [51, 52]. Mixture-of-Experts (MoE) architectures were proposed to address this issue by activating only a fraction of parameters per input, enabling trillion-scale models without proportional increases in computational costs. Specifically, MoE replaces the dense feed-forward network in each Transformer block with a set of parallel small networks, *i.e.*, *experts*. An MoE layer employs a gating network to route each token to a small subset of experts, typically the top-k with the highest routing scores, *i.e.*, top-k routing. The selected experts process the token in parallel, and their outputs are combined through a weighted aggregation based on the gating probabilities.

Today, MoE has become a central strategy for balancing model scaling and computational costs in cutting-edge LLMs. For instance, DeepSeek-V3 [35] (685B total parameters, 256

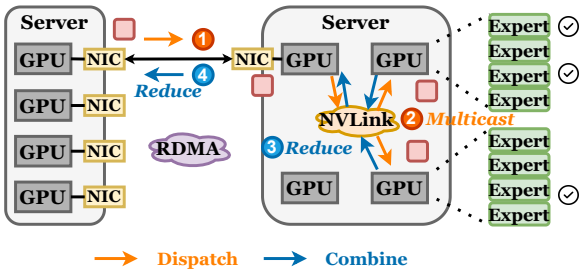


Figure 1: Communication process in expert parallelism.

experts) activates only 8 experts per token, achieving performance rivaling dense models like LLaMA-3.1-405B [5] while saving inference costs by more than 60%.

During inference, LLM execution is typically divided into two distinct phases: *prefill* and *decode*. The *prefill* phase processes the entire input prompt in parallel to initialize hidden states and KV caches, while the *decode* phase generates tokens autoregressively, one token at a time. The prefill phase is both communication- and computation-intensive, but exhibits high parallelism across tokens, making it well-suited for throughput-oriented optimization.

2.2 Expert Parallelism

In practice, experts are typically distributed across multiple computing devices, such as GPUs or NPUs, a strategy referred to as *expert parallelism* (EP). In this way, each device hosts a subset of experts. For example, consider an MoE layer with 64 experts distributed across 16 GPUs, with each GPU hosting 4 experts. There are two communication phases in EP, namely, *dispatch* and *combine*. To support these phases efficiently, communication libraries such as DeepEP have been introduced. Figure 1 shows the communication process in EP during the prefill stage in DeepEP.

During the *dispatch* operation, each token is sent from its source GPU that hosts the attention layer to the destination GPUs that host the selected top-k experts. To reduce redundant inter-server communication, the source GPU only sends one copy of the token to the forwarder GPU on each server. Upon arrival at the server, the forwarder GPU distributes multiple copies to different experts on various GPUs via intra-server NVLink-based transmission. According to recent quantization strategies [18, 39] aimed at reducing data transmission volume, the *dispatch* operation in DeepEP typically supports both FP8 [39] and BF16 [60] data formats.

In the *combine* operation, the forwarder GPU first *reduces* the outputs for the same token from all experts within a server, then sends them back to the source GPU. Finally, the source GPU performs another *reduce* operation, that is, reducing the outputs of the same token from different servers. In MoE models, since the attention computation does not support low-precision formats such as FP8, the output of the attention mechanism, which is also the data transmitted during the *combine* operation, typically remains in BF16 format.

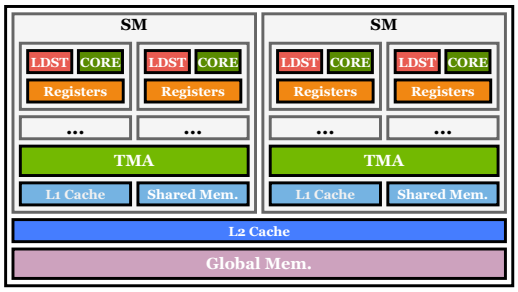


Figure 2: NVIDIA Hopper GPU architecture.

With the top-k routing, it is common that the same token is routed to different experts on the same server. Therefore, the token *multicast* and token *reduce* operations in DeepEP can significantly improve EP performance by avoiding redundant inter-server communication, enabling MoE models to scale to thousands of experts.

2.3 GPU Architecture and Interconnects

LLM inference relies on GPUs to exploit massive parallelism and achieve high performance for both computation and communication kernels [13, 29, 34]. In particular, the *prefill* phase is communication- and computation-intensive, making it essential to allocate compute resources and leverage GPU components efficiently.

Figure 2 shows the NVIDIA Hopper GPU architecture based on the *Streaming Multiprocessor (SM)* [3]. Each SM manages many threads grouped into *thread blocks* and scheduled in *warps* that follow the SIMT execution model. SMs provide *register files* for thread-private data and an on-chip memory hierarchy. Threads in a block share on-chip *shared memory*, while all threads can access off-chip *global memory* (*e.g.*, High-Bandwidth Memory (HBM)).

NVIDIA's *Tensor Memory Accelerator (TMA)*, introduced in Hopper architecture [46], asynchronously moves data between global and shared memory. Moreover, TMAs can reduce register pressure and thread overhead while supporting in-flight reductions. Its non-blocking nature allows threads to continue computation immediately after issuing transfers.

Scalable AI performance also relies on inter-GPU interconnects. These include high-bandwidth intra-server links (*e.g.*, NVLink/NVSwitch [47]) and inter-server networks (*e.g.*, InfiniBand or RoCE [10, 56]), both using dedicated buffers for efficient memory-semantic communication. We provide details for GPU architecture and interconnects in Appendix A.

3 Observations & Motivation

3.1 SM Overutilization in Communication

Due to the dependency between computation and dispatch/combine operations, LLM inference naturally schedules them sequentially: it first performs the computation of the attention layer, then issues *dispatch* before the MoE layer, and

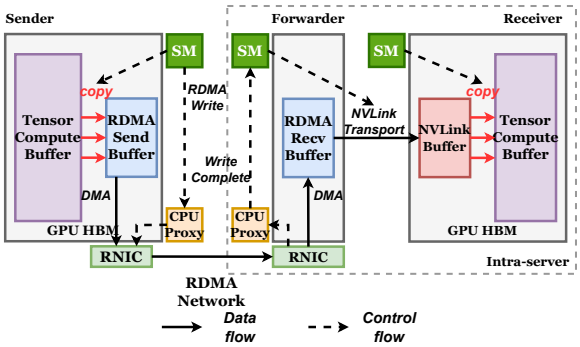


Figure 3: MoE communication workflow.

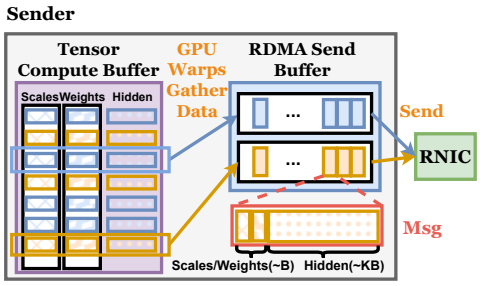


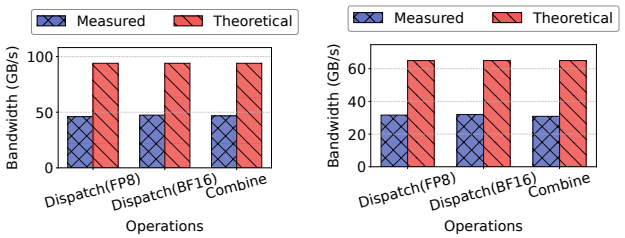
Figure 4: Data gather based on SM warps in GPUs.

finally aggregates the results of different experts through *combine* to start the next layer [1, 35]. While straightforward, this sequential execution often leaves SM resources underutilized during communication, since communication alone cannot fully saturate all SMs.

To overcome this inefficiency, Two micro-Batch Overlapping (TBO) [15, 21] has been proposed: the workload is split into two micro-batches, and one micro-batch performs computation while the other simultaneously executes all-to-all communication. This overlapping allows communication to progress in parallel with computation, significantly improving SM utilization and overall inference performance.

Nevertheless, TBO is a double-edged sword: when computation and communication are executed concurrently, the SMs available to computation are reduced due to contention with communication kernels, which can limit the inference performance. For instance, on an NVIDIA H20 GPU, communication operations in the inference prefill stage usually occupy 24 SMs to obtain the highest bandwidth by default [64], leaving only 54 out of 78 SMs for computation. In other words, computation can leverage at most 69% of the GPU's SM resources compared to the ideal case where all SMs are dedicated to computation, which directly constrains the achievable computation performance.

Causes: To understand why communication consumes such a large portion of SM resources, we profiled the execution of DeepEP (with version [16]) during the prefill dispatch stage on NVIDIA H20 GPUs. The analysis reveals that most of the occupied SMs are used for explicitly copying small, discontinuous data segments between the *tensor compute buffer* and the *RDMA send buffer*, as well as the subsequent



(a) EP=16 (b) EP=32

Figure 5: The measured and theoretical algorithm bandwidth of DeepEP in inference prefill stage.

NVLink transfers and data copying between the *NVLink buffer* and the *tensor compute buffer*, as illustrated in Figure 3.

Specifically, in RDMA-based communication, dedicated send buffers are used to stage data before transmission. These buffers provide a pre-registered, contiguous memory region shown in Figure 4 that the RDMA Network Interface Controller (RNIC) can access for direct network transfer, enabling efficient zero-copy communication[1]. On the other hand, for MoE models, each token is associated with multiple small, discontinuous data segments: the hidden state from the attention layer, the top-k expert indices, the FP8 scales, and their corresponding routing weights. Although scattered across different GPU memory regions, these data must be transmitted together in a single RDMA message to ensure that all parts of a token arrive simultaneously at the recipient GPU's memory during inter-server communication. This guarantees immediate availability for computation and maintains high computational throughput. Similarly, in the subsequent NVLink transfers and data copying between the NVLink buffer and the tensor compute buffer, DeepEP uses SMs to copy the message to multicontiguous memory. Consequently, DeepEP dedicates a total of 71% warps of 24 SMs to copy these data from different buffers and in reverse.

3.2 NVLink Bandwidth Underutilization

To empirically evaluate the bandwidth utilization of DeepEP, we conduct experiments under two cluster configurations: a 2-node, 16-GPU setup (EP=16) and a 4-node, 32-GPU setup (EP=32), with each node comprising $n = 8$ interconnected GPUs. The nodes are interconnected via a RoCEv2 network. Each server is equipped with 8 GPUs, each connected via a 400Gbps RDMA CX7 NIC, with a 900GBps full-duplex NVLink interconnect between GPUs. All experiments follow the DeepSeek-V3/R1 [17, 35] pre-training settings, configured with 4096 tokens per batch, 7168 hidden dimensions, and top-8 experts. Measurements are performed for both FP8 and BF16 dispatch, as well as BF16 combine. The measured algorithm bandwidth [14] is computed as the total bytes trans-

[1]Note that the zero-copy property applies to transfers from the RDMA send buffer to the RDMA receive buffer. Data must still be explicitly copied from the tensor compute buffer into the RDMA send buffer.

mitted during the communication phases divided by the execution time of the corresponding communication kernel.

To evaluate whether the measured algorithm bandwidth has reached its maximum, we build a theoretical model to estimate the ideal algorithm bandwidth of dispatch. For the ease of analysis, we neglect negative factors such as network congestion and SM constraints. This assumption enables us to evaluate the maximum achievable communication performance for a given RDMA/NVLink bandwidth. The traffic patterns generated by the DeepEP test tools are used as input, from which we obtain the theoretical upper bound of bandwidth, $B_{\max}$. Assume the total data volume is S. Under EP=16, the RDMA domain transfer time is $t_1 = (8/15 \cdot S)/400\text{Gbps}$, and the NVLink domain transfer time is $t_2 = (7/15 \cdot S)/900\text{Gbps}$. Since the dispatch transfer essentially forms a pipeline composed of all token transmissions, we have $t_2 < t_1$. After ignoring other overheads, the theoretical upper bound of algorithm bandwidth is $B_{\max} = S/t_1 = 400\text{Gbps} \times 15/8 = 750\text{Gbps}$ ($\approx$ 94GB/s). Similarly, for EP=32, $B_{\max} = 400\text{Gbps} \times 31/24 \approx$ 516.7Gbps ($\approx$ 65GB/s). As shown in Figure 5, the results demonstrate that at EP = 16 and 32, there remains significant potential for communication performance optimization.

Causes: To investigate the impact of NVLink forwarding on the measured algorithm bandwidth, we modify the CUDA kernel implementation of intra-server NVLink communication in DeepEP. Instead of physically forwarding data across NVLink, we emulate the process by completing the transfer after the *ideal communication time* (*i.e.*, data size divided by the peak NVLink bandwidth). The bandwidth achieved under this emulated setting closely matches the ideal maximum algorithm bandwidth, indicating that the current DeepEP implementation introduces a forwarding bottleneck in intra-server communication, thereby constraining overall performance.

From a source code analysis, we notice that DeepEP performs intra-server forwarding using a load/store (*ld/st*) mechanism [45]: data is first loaded from GPU memory into registers and then written to remote GPU memory across NVLink via store instructions. Consequently, the number of ld/st instructions that can be issued in parallel within a single SM, along with the registers available for the ld/st mechanism, directly limits NVLink communication performance. For instance, an NVIDIA H20 GPU is equipped with 64K 32-bit registers per SM, yet even when all of them are utilized, the effective NVLink bandwidth achieved with the ld/st mechanism reaches only about 50% of the theoretical peak. Even with TBO enabled, the communication time becomes notably higher than the computation time, making communication the performance bottleneck and thus impacting the overall application latency.

3.3 Motivation

Our experimental observations reveal that a significant portion of SMs is occupied by copying small, non-contiguous

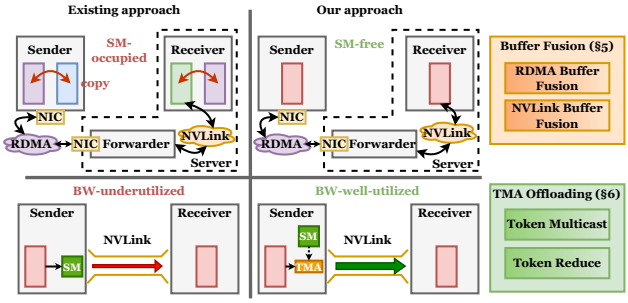


Figure 6: Overview of SwiftEP.

data between the tensor compute buffer and the communication buffer. This overhead arises from the separation of these buffers: tensors are produced in the compute buffer but must be copied to the RDMA buffer before transmission, as well as from the NVLink buffer to the tensor compute buffer after transmission. Fusing these buffers would allow data to be generated directly in a memory region also registered for RDMA/NVLink, thereby eliminating redundant copies and saving SM resources.

Furthermore, efficient intra-server token multicast and reduction over NVLink are crucial for approaching the theoretical bandwidth limit. Current methods using Copy Engines (CEs) or ld/st instructions are inefficient for maximizing NVLink bandwidth. Alternatively, TMA provides a hardware-accelerated, asynchronous mechanism for moving data between global and shared memory. Unlike software-managed approaches, TMA performs data transfers using dedicated hardware, reducing SM involvement.

In summary, existing communication libraries underuse NVLink bandwidth and overspend SM resources, hindering MoE inference performance. To this end, we redesign buffer management and NVLink communication by introducing a new framework that eliminates redundant data movement and leverages TMA for high-efficiency data transfer.

4 SwiftEP Overview

As shown in Figure 6, SwiftEP improves MoE model inference by tackling two key bottlenecks: (1) redundant data movement between compute and communication buffers, and (2) underutilized NVLink bandwidth in intra-server communication. SwiftEP introduces a communication library that uses buffer fusion to remove unnecessary data transfers and TMA offloading to maximize NVLink bandwidth.

Buffer Fusion. SwiftEP eliminates redundant data movement by tightly coupling tensor compute buffers with communication buffers. For inter-server communication, fusion occurs on the sender side, combining the tensor compute buffer with the RDMA send buffer. For intra-server communication, fusion is applied on the receiver side, where the NVLink buffer is fused with the tensor compute buffer. In both cases, communication buffers are pre-allocated, and the metadata of the fused buffer (*e.g.*, base address and mem-

ory layout) is exposed to the inference framework. In this way, only minimal modifications (*i.e.*, modifying the memory addresses for inputs and outputs within the framework) are made in the inference framework, maintaining programming transparency to model developers.

Buffer fusion allows the inference framework to write attention layer outputs directly into the RDMA send buffer and read MoE layer inputs directly from the NVLink receive buffer, eliminating intermediate copies and reducing both SM resource usage and communication latency.

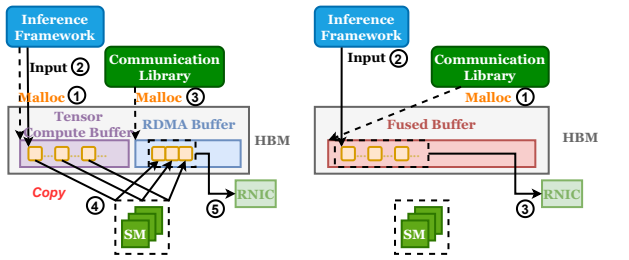
TMA Offloading. SwiftEP uses NVIDIA's TMA to offload NVLink transfers from SMs to dedicated hardware. An SM thread initiates an asynchronous TMA load or store instruction and is immediately freed for other tasks, while the hardware handles the data transfer. This approach reduces SM workload significantly and improves NVLink bandwidth utilization, enabling efficient token multicast during dispatch and token reduce during combine.

4.1 Challenges

While buffer fusion and TMA offloading are promising to significantly improve communication efficiency, utilizing them in practice introduces several technical challenges.

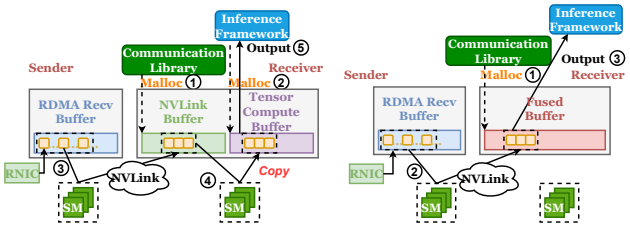
Challenge 1: Transferring small, scattered data segments within a large buffer is inefficient. In MoE models, the data segments to be transferred, such as hidden states or top-k weights, are scattered in the tensor compute buffer when produced by attention layers, and their sizes are typically small (*e.g.*, ∼32 bytes for top-k weights in a token). In traditional separated buffer designs, these small segments can be gathered into larger continuous regions when copied to the RDMA send buffer. However, after buffer fusion, such intermediate copies are eliminated, making independent data gathering within the fused buffer difficult. Additionally, the RNIC relies on its Memory Translation Table (MTT) to manage RDMA-registered memory, but the on-chip MTT cache can store only a limited number of entries. Fusion significantly enlarges the buffer beyond the MTT cacheable range, necessitating a multi-level virtual-to-physical address translation. Each MTT lookup traverses the PCIe interconnect and accesses GPU memory, introducing substantial latency [22, 61].

Challenge 2: The sender GPU cannot predetermine token destinations in fused receive buffers. For MoE layers, tokens from multiple GPUs must be concatenated in memory before performing matrix multiplications. In traditional separated buffer designs, this requirement can be handled during the copy from the NVLink receive buffer to the tensor compute buffer, allowing the memory layout to be adjusted as needed. Under fusion, the intra-server sender must write tokens directly to their final memory locations on the receiver, but these locations depend on dynamic factors like token counts from various sources. Without knowing these locations in advance, the sender risks data overwrites or layout corruption.



(a) Separated buffer (b) Fused buffer

Figure 7: Separated buffer vs. Fused buffer in the inter-server dispatch communication.



(a) Separated buffer (b) Fused buffer

Figure 8: Separated buffer vs. Fused buffer in the intra-server dispatch communication.

Challenge 3: TMA does not natively support remote GPU memory access. TMA is designed for asynchronous intra-GPU memory copies and lacks built-in support for accessing remote GPU memory over NVLink. To use TMA for inter-GPU transfers in MoE communication, it must be extended to support remote memory access while preserving its benefits in hardware offloading and asynchronous operation.

5 Buffer Fusion

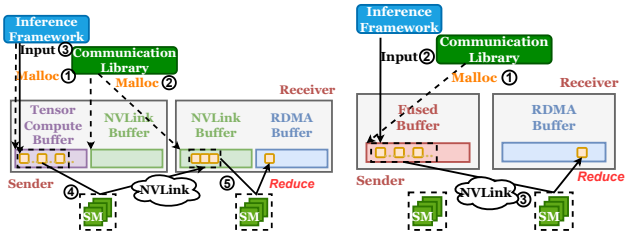
In this section, we first introduce different buffer fusion types employed by SwiftEP, and then demonstrate how to address the challenges brought by these buffer fusions.

5.1 Buffer Fusion Type

Since inter-server communication uses RDMA and intra-server communication uses NVLink, SwiftEP employs two types of buffer fusion: RDMA buffer fusion and NVLink buffer fusion.

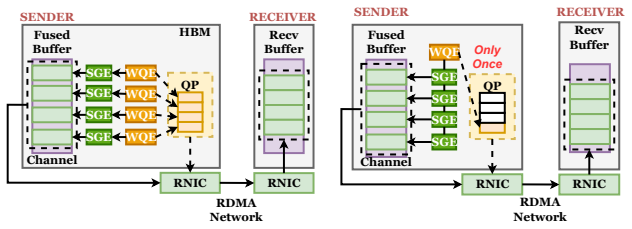
RDMA buffer fusion. Figure 7 compares the separated buffer design commonly used in existing communication libraries and the fused buffer design introduced by SwiftEP in the inter-server *dispatch* communication. By fusing the RDMA send buffer with the tensor compute buffer, SwiftEP eliminates redundant data movement, thereby freeing the SM resources that would otherwise be consumed by buffer copies. These released SM resources can instead be utilized to accelerate other computation tasks.

NVLink buffer fusion. Similarly, Figure 8 illustrates the contrast between the separated buffer design and the fused



(a) Separated buffer (b) Fused buffer

Figure 9: Separated buffer vs. Fused buffer in the intra-server combine communication.



(a) Without coalescing (b) Coalescing

Figure 10: Different WQE post mechanisms.

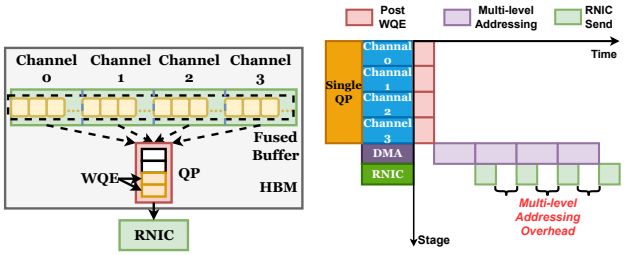
buffer design in intra-server *dispatch* communication. Unlike RDMA buffer fusion, which applies to the sender GPU, NVLink buffer fusion integrates the NVLink receive buffer with the tensor compute buffer on the receiver GPU, thereby freeing SM resources on the receiving side. For intra-server *combine* communication, which is the reverse process of dispatch, SwiftEP also adopts NVLink buffer fusion. The input to combine is the output of MoE computation, and this output has the same layout as the data produced during dispatch. Leveraging this property, SwiftEP simply reuses the fused NVLink buffer pre-allocated in the dispatch phase, which is shown in Figure 9.

5.2 Optimization for RDMA Buffer Fusion

To address Challenge 1 introduced by RDMA send buffer fusion, we design two mechanisms: WQE post coalescing and QP transmission parallelization.

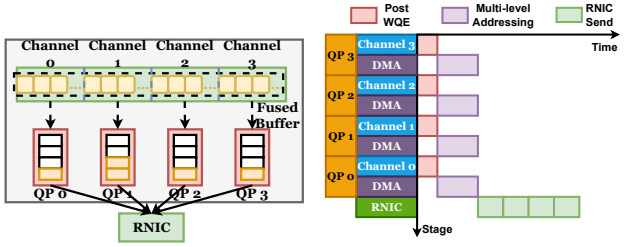
5.2.1 WQE Post Coalescing

In existing communication libraries, the RNIC typically processes only one data segment per Work Queue Element (WQE) due to memory continuity requirements. After buffer fusion eliminates the data-gathering step inherent in separate-buffer designs, each small, scattered data segment must be transmitted individually. As shown in Figure 10(a), the sender posts one WQE per segment to the Queue Pair (QP). Each WQE contains the base address and length of one segment in a single Scatter-Gather Element (SGE). These WQEs are processed serially by the RNIC, which directly accesses each segment from the fused buffer. Transmitting numerous small, scattered segments in this manner greatly increases the number of WQE postings, raising control overhead and becoming a major performance bottleneck.



(a) All channels bind to one QP (b) Multi-level addressing overhead

Figure 11: Single QP transmission in existing libraries.



(a) One channel exclusively binds to one QP (b) Multi-level addressing overhead is masked

Figure 12: QP transmission parallelization.

SwiftEP takes full advantage of the capability in modern RNICs to transmit scattered data segments efficiently using a single WQE. A WQE can contain multiple SGEs arranged as a linked list, with each SGE describing a different memory region. This allows the RNIC's DMA engine to read from or write to multiple non-contiguous memory locations in parallel when processing one WQE. As illustrated in Figure 10(b), SwiftEP uses this multi-SGE coalescing capability within each communication channel by packaging small, scattered data segments into the SGEs of a single WQE. This significantly reduces the number of WQEs the inference framework must post. As a result, GPU kernels can post WQEs that map scattered memory regions directly, avoiding costly data aggregation while maintaining high RNIC throughput.

5.2.2 QP Transmission Parallelization

In existing communication libraries like DeepEP and NCCL, the RDMA buffer is divided into multiple communication channels, each handling a portion of the tensor data (Figure 11(a)). However, all channels share a single QP, meaning every WQE must be posted through that same QP. This forces the RNIC to process WQEs sequentially, with each DMA operation requiring an MTT lookup to identify the data to be transmitted. After fusing the tensor compute buffer with the RDMA buffer, the larger fused buffer increases the chance of multi-level page table walks for each DMA operation. Serialized execution worsens this issue, as the latency from address translation across channels accumulates, leading to significant performance degradation, as illustrated in Figure 11(b).

To address this, SwiftEP introduces QP transmission parallelization. As depicted in Figure 12(a), the fused buffer is partitioned into multiple communication channels, each assigned to a dedicated QP. This allows WQEs to be posted in parallel, enabling the RNIC to execute DMA operations concurrently across multiple QPs rather than serializing them through a single QP. By leveraging the RNIC's innate ability to process WQEs concurrently, the design converts the communication bottleneck into parallelized data streams.

This parallelization directly reduces the performance impact of multi-level page table walks. While one QP's DMA engine is delayed by a page table walk in GPU memory, other QPs continue transmitting data, accessing independent MTT cache entries or overlapping memory operations. Such interleaved execution effectively masks translation latency that would otherwise be serialized in a single-QP design. As shown in Figure 12(b), parallel DMA invocation minimizes idle time between transmissions and significantly improves RDMA bandwidth utilization.

5.3 Optimization for NVLink Buffer Fusion

To address Challenge 2 from NVLink receive buffer fusion, SwiftEP redesigns the intra-server communication flow of the *notify* operation. Instead of sending token counts individually to each target GPU, the sender GPU broadcasts the full token distribution to all GPUs in the server. This allows every GPU to fully understand the memory layout expected by the upper-layer inference framework. As a result, during intra-server dispatch communication, the sender GPU can compute destination addresses directly and place tokens into the correct memory locations on receiver GPUs, avoiding data overwrites and maintaining the fused memory layout.

As mentioned in Section 5.1, the combine operation is the reverse of dispatch, and thus the outputs of experts are naturally stored in the fused NVLink buffer. These outputs can be transferred within the server via either a sender-driven *push* or receiver-driven *pull*. SwiftEP uses a *pull* mechanism shown in Figure 9(b) for three reasons. First, the receiver GPU already knows the buffer address of expert outputs from its prior role as sender during token multicast. Second, data on the sender is guaranteed ready when combine starts, avoiding extra synchronization. Third, TMA supports element-wise reduction, allowing the receiver GPU to reduce outputs from different experts during the pull operation using only its own SMs. This approach improves efficiency and saves sender SM resources.

6 TMA Offloading

6.1 Extending TMA to Inter-GPU Transfer

SwiftEP pioneers the use of TMA for inter-GPU transfer within a server, extending its conventional role in intra-GPU communication. This is achieved by enabling TMA to access memory on remote GPUs within the same server. To do so, SwiftEP employs CUDA Context IPC: each GPU obtains a memory handle via *cudaIpcGetMemHandle* to the base address of a pre-allocated device memory region (i.e., an NVLink buffer) and shares it with all other GPUs in the server. This grants TMA the permissions needed for efficient direct memory access across GPUs, enabling memory-semantic transfers over NVLink.

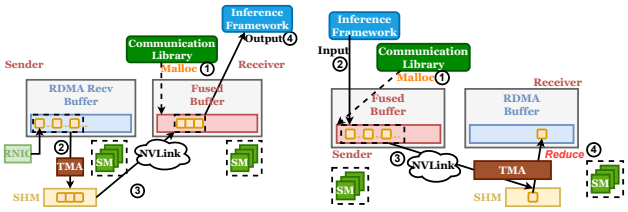
Unlike computational workflows where TMA is often used for isolated data movements before or after kernel execution, SwiftEP integrates TMA load and store operations into a unified sequence to form complete memory-semantic NVLink transfers. Prior to transmission, the SM allocates a suitably sized shared memory region for data staging. In a sender-initiated *push* operation, an SM thread first issues a TMA load to load data from global memory into shared memory. It then triggers a TMA store to send the data over NVLink to the receiver's global memory. For a receiver-initiated *pull* operation, an SM thread first loads remote data via TMA over NVLink into its shared memory, and then stores it to local global memory.

Offloading token multicast operations to TMA. As illustrated in Figure 13(a), SwiftEP offloads token multicast operations during dispatch to TMA. The multicast offloading uses a *push* mechanism, as it must await the completion of inter-server RDMA transfers. Additionally, SwiftEP allocates shared memory to support token multicast under various data formats, such as FP8 and BF16. During the token multicast operation, the SM thread within the sender GPU executes a TMA load instruction to load the token from global memory into the SM's shared memory. The thread is immediately released after issuing this command, allowing it to participate in other computational tasks. Upon completion of the asynchronous load, the SM thread invokes TMA to trigger an asynchronous store operation via NVLink transmission, and delivers the data to the receiver GPU's global memory.

Offloading token reduce operations to TMA. As shown in Figure 13(b), SwiftEP offloads token reduce operations during combine to TMA via a *pull* mechanism, as described in Section 5.3. Shared memory is allocated to support BF16 reduction operations. The workflow proceeds as follows: the SM thread of the receiver GPU in the intra-server domain executes a TMA load instruction to initiate an NVLink transfer, loading the token into the SM's shared memory. The thread is then released until the NVLink transfer completes. Once loaded, the same thread executes a store-with-accumulate instruction via TMA, directly adding the token from shared memory to its designated location on the receiver GPU's global memory.

6.2 Optimization for TMA Offloading

To reduce the latency of NVLink transmissions initiated by TMA, SwiftEP introduces dynamic shared memory allocation and alignment. SwiftEP further improves NVLink bandwidth



(a) Token multicast operations (b) Token reduce operations

Figure 13: Offloading NVLink Transmission to TMA.

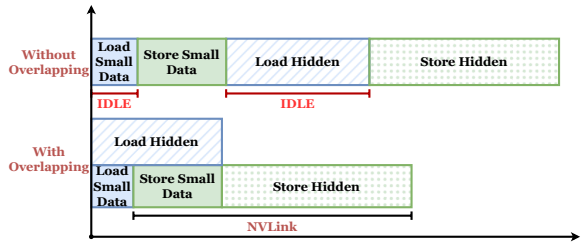


Figure 14: Without vs. With data transmission overlapping.

utilization by overlapping data transmissions during multicast operations.

Dynamic shared memory allocation and alignment. In CUDA, shared memory can be allocated either statically or dynamically. Although both methods have allocation limits, dynamic allocation supports much larger capacities with often several hundred kilobytes per SM. This enables TMA to send or receive a full token in a single instruction during token multicast and token reduce operations. SwiftEP further aligns the dynamically allocated memory to the granularity of each token's hidden state. This alignment allows TMA to read from and write to global memory using the fewest memory transactions, thereby reducing NVLink latency by avoiding multiple transfer cycles.

Data transmission overlapping. During token multicast offloading, only store operations use NVLink bandwidth; load operations do not. Executing these transfers sequentially wastes bandwidth while loading data segments, as depicted in Figure 14. To address this, SwiftEP overlaps the loading of large data (*i.e.*, hidden states) with the loading and storing of small data (*i.e.*, scale factors, expert indices, and weights). As shown in Figure 14, the long-running load of hidden states is initiated asynchronously first, based on the asynchronous copy capability of TMA. Concurrently, the load and store operations for small data are executed, effectively hiding the latency of loading the hidden states.

7 Implementation

We implement SwiftEP by extending the *internode* kernel of the state-of-the-art MoE communication library DeepEP, with a focus on optimizing the dispatch and combine operations during inference prefill. SwiftEP requires only targeted modifications to mainstream inference frameworks, adjusting memory mapping parameters for attention computation outputs and MoE computation inputs, while its internal execution logic remains fully transparent to upper-layer frameworks and compatible with various models. The implementation consists of approximately 2500 lines of CUDA and C++ code. We open-sourced the code of SwiftEP at https://github.com/deepseek-ai/DeepEP/tree/tencent-zcopy [2].

Key technical enhancements are summarized below, with detailed descriptions provided in the Appendix B.

Multi-SGE transmission. We extend the InfiniBand GPUDirect Async (IBGDA) [48] network operation interface to support multi-SGE transmission. Multiple threads are used to concurrently fill each SGE data segment. A ring buffer mechanism is applied when the total data size exceeds the work queue length.

QP transmission parallelization. SwiftEP replaces DeepEP's original InfiniBand Reliable Connection (IBRC) [48] interface with IBGDA for RDMA transmission, enabling explicit configuration of the number of QPs during RDMA connection setup. The number of QPs is determined based on predefined communication channels to match application demands.

TMA remote GPU memory access. Low-level PTX instructions for TMA are abstracted into reusable load/store operations. These are combined to form complete NVLink transmission procedures between remote GPUs, providing a uniform programming model for distributed memory access over NVLink. All inter-GPU communication routines in SwiftEP use this encapsulated interface.

8 Evaluation

In this section, we evaluate the performance of SwiftEP to validate its capability to improve GPU communication efficiency and computational resource utilization in LLM inference.

8.1 Experiment Setup

Testbed. Experiments are conducted on two GPU clusters: a 2-node cluster (EP=16) and a 4-node cluster (EP=32). Each node is a server equipped with eight NVIDIA H20 GPUs interconnected via NVLink/NVSwitch, providing a bidirectional bandwidth of 900 GBps. GPUs across nodes communicate over a RoCEv2 network with a topology similar to HPN [49] and Astral [38]. Each GPU is attached to a dual-port 2×200 Gbps NVIDIA ConnectX-7 NIC [43] running MLNX_OFED_LINUX-5.8-2.0.3.0, and all NICs are connected to a TH5 switch. For end-to-end evaluation, we additionally deploy a 2-node cluster with eight NVIDIA Hopper GPUs per node, which provide higher compute capability and communication load, while keeping all other configurations unchanged.

Baselines. We adopt DeepEP, the state-of-the-art MoE model communication library, as the baseline. To systematically

[2]More details of the code are provided in https://github.com/deepseek-ai/DeepEP/pull/453.

Table 1: Occupied SM count in dispatch and combine on each NVIDIA H20 GPU. Both DeepEP and SwiftEP typically occupy the same number of GPU SMs during dispatch and combine operations.

	DeepEP		SwiftEP	
	FP8	**BF16**	**FP8**	**BF16**
EP=16	24	24	12	12
EP=32	24	24	8	8

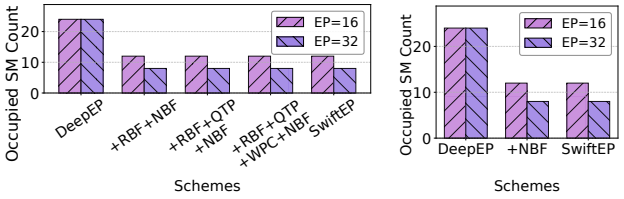
evaluate the contributions of individual techniques introduced in SwiftEP, we conduct comprehensive ablation studies. SwiftEP provides enhancements over the baseline in two key aspects: SM occupancy reduction and bandwidth improvement. For SM occupancy reduction, four techniques are introduced: (1) **R**DMA **B**uffer **F**usion (RBF), (2) **N**VLink **B**uffer **F**usion (NBF), (3) **W**QE **P**ost **C**oalescing (WPC), and (4) **Q**P **T**ransmission **P**arallelization (QTP). Among these, RBF, NBF, WPC, and QTP are applied in the dispatch phase, and NBF is also employed in the combine phase. The bandwidth improvement is achieved through three primary techniques: (1) **T**MA-**O**ffloaded token multicast and reduce operations (TO), (2) **D**ynamic **S**hared **M**emory allocation and alignment (DSM), and (3) **D**ata **T**ransmission **O**verlapping (DTO). All three techniques are applied in the dispatch phase, while TO and DSM are also utilized in the combine phase.

8.2 SM Occupancy Reduction

We first evaluate the SM occupancy reduction by SwiftEP during the dispatch and combine phases of the inference prefill stage under varying token counts. The evaluation uses a model with 4096 tokens, 7168 hidden size and selects the top-8 experts per token.

As shown in Table 1, compared to DeepEP, SwiftEP reduces the number of SMs occupied during dispatch and combine per NVIDIA H20 GPU by 50% for EP=16 and by 66.7% for EP=32. Each NVIDIA H20 GPU contains 78 SMs. Under the computation-communication overlap mechanism in DeepSeek's Two mirco-Batch Overlapping (TBO), communication kernels persistently occupy SMs during prefill. The number of SMs for SwiftEP is manually configured via parameter configuration file settings, typically set to the minimum count required to saturate the bandwidth, similar to the approach used in DeepEP.

This improvement is primarily achieved by replacing SM-bound data copy operations in communication kernels, thereby freeing SM resources for compute tasks. We further conduct an ablation study on the specific techniques responsible for reducing SM occupancy, including replacing IBRC with IBGDA, RBF, WPC, QTP, and NBF, to evaluate their individual impacts and validate our design. While assessing SM occupancy, we also measure the bandwidth impact of these optimizations during dispatch. The results confirm that



(a) Dispatch (b) Combine

Figure 15: Impact of different optimization techniques of buffer fusion on SM occupancy reduction.

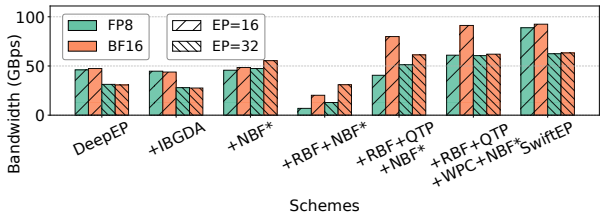


Figure 16: Impact of the optimization techniques of buffer fusion of dispatch on bandwidth improvement. * indicates that the subsequent experiments include IBGDA.

our buffer fusion strategies are effective and that the reduction in SM occupancy does not degrade bandwidth performance.

8.2.1 Ablation Study

RDMA Buffer Fusion (RBF) and NVLink Buffer Fusion (NBF). As shown in Figure 15(a), introducing NBF and RBF into the dispatch operation is the primary reason for reduced SM occupancy in SwiftEP. By removing data copies between tensor, RDMA, and NVLink buffers that required dedicated warps in DeepEP, SwiftEP consolidates only essential communication warps into fewer SMs, significantly reducing SM occupancy during dispatch. Figure 15(b) shows that NBF similarly reduces SM occupancy in the combine phase by eliminating redundant data movement between tensor compute and NVLink buffers. However, as illustrated in Figure 16, introducing RBF alone causes bandwidth to decline by up to 85%. This is due to enlarged RDMA memory regions introducing RNIC multi-level addressing overhead and increased latency, along with fragmented transmission of small messages raising WQE operation costs.

IBRC vs. IBGDA. Figure 16 shows that replacing IBRC in DeepEP with IBGDA results in a very slight decrease in bandwidth. IBRC relies on the GPU initiating communication via a CPU proxy, while IBGDA involves the GPU SM initiating communication directly. When DeepEP uses IBGDA with a single shared QP across all communication channels, the GPU SM threads handle WQEs serially, which cannot fully utilize the GPU's parallel processing capability. However, IBGDA offers richer and more flexible interfaces, making it easier to implement optimizations such as QTP and WPC.

QP Transmission Parallelization (QTP). With QTP, as shown in Figure 16, SwiftEP achieves bandwidth improvements from 2× to 5.8×. This confirms that QTP effectively

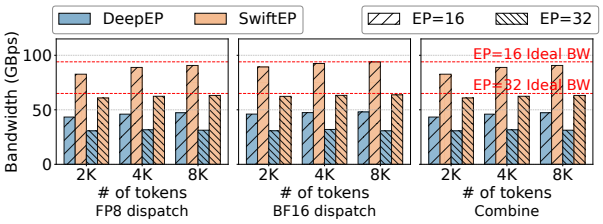
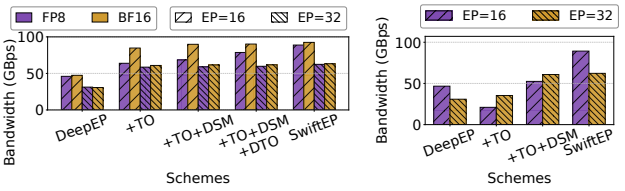


Figure 17: Bandwidth under different communication operations of DeepEP vs. SwiftEP.



(a) Dispatch (b) Combine

Figure 18: Impact of different optimization techniques of TMA offloading on bandwidth improvement.

masks RNIC addressing overhead and enhances bandwidth utilization, demonstrating that the SM occupancy reduction does not compromise bandwidth.

WQE Post Coalescing (WPC). Introducing WPC moderately improves bandwidth in SwiftEP by up to 50.4% with FP8, as shown in Figure 16. WPC reduces WQE operation overhead and fully leverages DMA parallelism. The greater improvement with FP8 is due to its more fragmented data distribution from transmitting additional quantization scaling factors, making it more responsive to WPC optimization.

8.3 Algorithm Bandwidth Performance

We evaluate the bandwidth improvement of SwiftEP during the dispatch and combine phases of the inference prefill stage across varying token counts.

As shown in Figure 17, SwiftEP achieves up to 101.8% higher bandwidth than DeepEP in dispatch with FP8, 107.5% with BF16 and 119.7% in combine. In both phases, the bandwidth of SwiftEP closely approaches the theoretical upper bound. By offloading token multicast in dispatch and reduce in combine to TMA, SwiftEP significantly enhances NVLink efficiency, resulting in substantially higher bandwidth. We further conduct a detailed ablation study on the techniques enabling these gains, including TO, DSM, and DTO.

8.3.1 Ablation Study

TMA-Offloaded token multicast and reduce operations (TO). As shown in Figure 18(a), offloading NVLink transmission in dispatch to TMA increases bandwidth by up to 92.6%, confirming that TO effectively alleviates the performance bottleneck and improves bandwidth utilization. However, Figure 18(b) indicates that offloading the token reduce operation in combine using a naive TMA approach reduces bandwidth. This is due to the limited statically allocated shared memory: unlike the sequential multicast in dispatch, the reduce requires each SM to handle multiple tokens concurrently, drastically reducing available shared memory per token. Consequently, each reduce operation requires more iterative steps, increasing latency and lowering effective bandwidth.

Dynamic Shared Memory allocation and alignment (DSM). Figure 18(a) shows that introducing DSM improves bandwidth by up to 7.1% over the naive TO method, due to reduced processing latency in multicast and more efficient TMA memory transactions. In the combine phase (Figure 18(b)), DSM increases bandwidth by 1.7× to 2.5× compared to the naive approach, eliminating the shared memory capacity bottleneck in TMA-driven NVLink transmission for Reduce.

Data Transmission Overlapping (DTO). Adding DTO yields further bandwidth gains, especially with FP8 where bandwidth increases by up to 13.1% (Figure 18(a)). FP8's quantization introduces smaller, more fragmented data packets. DTO leverages TMA's asynchronous capability to improve NVLink utilization and mitigate the performance bottleneck caused by transmitting fragmented small data during inference.

8.4 End-to-End Evaluation

We evaluate SwiftEP's end-to-end performance using the DeepSeek-R1 model with EP=16 and Data Parallelism of 16 for attention layers. Inference involves only dispatch and combine communication, without *allreduce* or NCCL. Experiments use the SGLang framework with TBO and FP8 dispatch, a batch size of 32, and an average input length of 1024. We set the output length to 1 to evaluate the performance of the prefill stage. The context length is 64K, and the chunked prefill size is 32K. We use a widely adopted benchmark dataset named sharegpt_v3_unfiltered_cleaned_split [6], a standard cleaned and pre-split subset of the ShareGPT corpus [4]. Compared to DeepEP, SwiftEP achieves a 21.2% improvement in average Input Tokens Per Second (ITPS) per GPU. ITPS describes how quickly an LLM processes the input prompt during the prefill stage, while Time-To-First-Token (TTFT) is the delay before the model produces its first output token. Since prefill time scales roughly with *input tokens count / ITPS*, higher ITPS generally leads to lower TTFT.

This improvement stems from two key enhancements. First, TMA offloading removes the NVLink transmission bottleneck, boosting intra-server communication bandwidth and reducing its share of total execution time. Second, since computation and communication proceed concurrently with TBO enabled, buffer fusion eliminates redundant memory copies, freeing SMs for computation.

9 Discussion and Limitations

Adaptation to MoE model training. In inter-server buffer fusion, a major bottleneck arises from the millisecond-scale

latency of memory registration when tensor buffers are registered for RDMA after computation. To alleviate this overhead, we pre-register a fused buffer of the required size and modify the inference framework so that computation results are written directly into the pre-registered buffer. This design enables RDMA communication to start immediately and allows the same buffer to be reused across layers during inference. However, the training phase imposes a critical constraint: intermediate activations must be retained for backward propagation, which precludes straightforward buffer reuse. To address this, an alternating pre-registration strategy with multiple fused buffers is necessary.

RNIC capability limits. WQE post coalescing introduces a subtle trade-off related to PCIe efficiency. In GPU-to-GPU communication, GPU threads post WQEs to the QP and ring the doorbell, triggering the RNIC DMA engine to parse each WQE and issue multiple PCIe read requests to fetch SGEs for transmission. Each PCIe read incurs link-layer header overhead [31]. Consequently, when a single WQE aggregates too many SGEs, the cumulative PCIe header overhead can outweigh the control-plane savings from WQE coalescing, leading to performance degradation. Empirically, this threshold typically lies in the hundreds of SGEs, such as on CX7 NICs [43]. Operating well within this range, SwiftEP achieves optimal performance without encountering this limitation.

QP scalability. While QP parallel transmission increases the number of required QPs, this overhead remains manageable for MoE models. Expert parallelism is naturally bounded by the total number of experts, up to 384 in current open-source models [54]. Furthermore, since libraries like TCCL [32] and DeepEP optimize within-node communication, each RNIC only needs to communicate with at most *experts/8* remote peers. With 4 QPs per RNIC pair, even the largest MoE models utilize only 192 concurrent QPs, which is far below the CX7 NIC's capacity of thousands [43], thereby avoiding QP context cache misses entirely.

10 Related Work

Collective communication library for training and inference. Existing work on communication libraries [23, 24, 32, 63, 64] has improved performance and reduced network congestion by optimizing communication patterns. For instance, NCCL [24] uses a ring-based allreduce to avoid full-mesh traffic, while DeepEP [64] employs NVLink for efficient token multicast, minimizing inter-node transfers. A key limitation across these libraries is the separation of tensor and RDMA buffers, necessitating explicit data movement through GPU SMs and introducing overhead. Our approach reduces SM resource usage via buffer fusion, shortening communication time in overlap scenarios. Moreover, we are the first to use TMA for intra-node transfers, improving NVLink efficiency and reducing dispatch bottlenecks. MegaScale-Infer [68] introduces an Attention–FFN–Disaggregation serving architecture and proposes an M2N communication library for the inference decode stage that leverages CPUs for inter-node communication. In contrast, SwiftEP focuses on the prefill stage and adopts direct GPU-to-GPU communication for both inter- and intra-node transfers.

TMA reduce offloading and NVSwitch SHARP. Token multicast and reduce are fundamental to MoE communication. Although NVSwitch SHARP [9] and programmable network switches [26, 37, 65] support in-network computation for symmetric collectives (e.g., *allreduce*), MoE communication is sparse and asymmetric: tokens are dynamically routed to top-k experts, and reductions are performed only within selected GPU subsets. This irregularity makes in-network offloading impractical. DeepEP [64] performs *reduce* on GPU SMs, incurring nontrivial SM overhead. In contrast, we offload token-wise MoE reduction to TMA, originally designed for GEMM, thereby reducing SM utilization and improving efficiency, especially in fine-grained dispatch and combine phases.

Inter-node communication performance. Prior work [19, 67] improves inter-node performance through flow control and advanced QP management to alleviate network congestion. In contrast, this work does not target inter-server communication or congestion control. SwiftEP is orthogonal to these efforts, focusing instead on intra-node communication, efficient memory management, and computation–communication overlap within GPU domains.

11 Conclusion

In this work, we identified critical inefficiencies in existing EP communication libraries for MoE-based LLM inference, including excessive SM consumption and underutilization of interconnect bandwidth. We proposed SwiftEP, an all-to-all communication library that leverages buffer fusion and TMA offloading to achieve true zero-copy communication and maximize NVLink utilization. Experimental results on multi-GPU clusters demonstrate that SwiftEP significantly outperforms DeepEP, achieving higher algorithm bandwidth, reduced SM occupancy, and improved inference request throughput. SwiftEP provides a practical solution for accelerating MoE inference, enabling more responsive LLM serving.

Acknowledgements

We sincerely thank our shepherd Dejan Kostić and anonymous reviewers for their insightful comments. This work is supported in part by the National Key Research and Development Program of China (No. 2024YFB2907000), and by the National Science Foundation of China (NSFC) under Grant No. 62302055 and NSFC under Grant No. 62502472. Yiran Zhang and Shuai Wang are the corresponding authors.

References

[1] [3/N] MoE Refactor: Simplify DeepEP Output by chwan · Pull Request #8421 · sgl-project/sglang. https://github.com/sgl-project/sglang/pull/8421.

[2] RDMA Core Userspace Libraries and Daemons. https://github.com/linux-rdma/rdma-core.

[3] GPU Performance Background User's Guide. https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/, 2023.

[4] Sharegpt: Publicly shared chatgpt conversations. https://huggingface.co/datasets/ShareGPT, 2023.

[5] Llama-3.1-405B. https://github.com/meta-llama/llama-models/blob/main/models/llama3_1/MODEL_CARD.md, 2024.

[6] sharegpt_v3_unfiltered_cleaned_split dataset. https://huggingface.co/datasets/learnanything/sharegpt_v3_unfiltered_cleaned_split, 2024.

[7] Qingyao Ai, Ting Bai, Zhao Cao, Yi Chang, Jiawei Chen, Zhumin Chen, Zhiyong Cheng, Shoubin Dong, Zhicheng Dou, Fuli Feng, Shen Gao, Jiafeng Guo, Xiangnan He, Yanyan Lan, Chenliang Li, Yiqun Liu, Ziyu Lyu, Weizhi Ma, Jun Ma, Zhaochun Ren, Pengjie Ren, Zhiqiang Wang, Mingwen Wang, Ji-Rong Wen, Le Wu, Xin Xin, Jun Xu, Dawei Yin, Peng Zhang, Fan Zhang, Weinan Zhang, Min Zhang, and Xiaofei Zhu. Information retrieval meets large language models: A strategic report from chinese ir community, 2023.

[8] Merav Allouch, Amos Azaria, and Rina Azoulay. Conversational agents: Goals, technologies, vision and challenges. *Sensors*, 21(24):8448, 2021.

[9] Korzh Anton, Pharris Brian, Comly Nick, Eassa Ashraf, and Elmeleegy Amr. 3x faster allreduce with nvswitch and tensorrt-llm multishot. Last accessed: 2023-10-05.

[10] InfiniBand Trade Association. InfiniBand Specification Frequently Asked Questions. https://www.infinibandta.org/ibta-specification/.

[11] Thomas W Barr, Alan L Cox, and Scott Rixner. Translation caching: skip, don't walk (the page table). *ACM SIGARCH Computer Architecture News*, 38(3):48–59, 2010.

[12] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*, 2022.

[13] Jack Choquette. Nvidia hopper gpu: Scaling performance. In *2022 IEEE Hot Chips 34 Symposium (HCS)*, pages 1–46. IEEE Computer Society, 2022.

[14] Deepseek-AI. DeepEP/tests/test_internode.py at main · deepseek-ai/DeepEP. https://github.com/deepseek-ai/DeepEP/blob/main/tests/test_internode.py.

[15] Deepseek-AI. deepseek-ai/profile-data: Analyze computation-communication overlap in V3/R1. https://github.com/deepseek-ai/profile-data.

[16] DeepSeek-AI. Initial commit · deepseek-ai/DeepEP@ebfe47e. https://github.com/deepseek-ai/DeepEP/commit/ebfe47e46fa86a5b41be96c326a92f62ed6721da.

[17] DeepSeek-AI and et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025.

[18] Maxim Fishman, Brian Chmiel, Ron Banner, and Daniel Soudry. Scaling fp8 training to trillion-token llms, 2025.

[19] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, et al. Rdma over ethernet for distributed training at meta scale. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 57–70, 2024.

[20] In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt cache: Modular attention reuse for low-latency inference. *Proceedings of Machine Learning and Systems*, 6:325–338, 2024.

[21] Raja Gond, Nipun Kwatra, and Ramachandran Ramjee. TokenWeave: Efficient Compute-Communication Overlap for Distributed LLM Inference. http://arxiv.org/abs/2505.11329, July 2025. arXiv:2505.11329 [cs].

[22] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 202–215, New York, NY, USA, 2016. Association for Computing Machinery.

[23] Mert Hidayetoglu, Simon Garcia de Gonzalo, Elliott Slaughter, Pinku Surana, Wen-mei Hwu, William Gropp, and Alex Aiken. Hiccl: A hierarchical collective communication library. In *2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 950–961. IEEE, 2025.

[24] Zhiyi Hu, Siyuan Shen, Tommaso Bonato, Sylvain Jeaugey, Cedell Alexander, Eric Spada, James Dinan, Jeff Hammond, and Torsten Hoefler. Demystifying nccl: An in-depth analysis of gpu communication protocols and algorithms. *arXiv preprint arXiv:2507.04786*, 2025.

[25] Chao Jin, Ziheng Jiang, Zhihao Bai, Zheng Zhong, Juncai Liu, Xiang Li, Ningxin Zheng, Xi Wang, Cong Xie, Qi Huang, et al. Megascale-moe: Large-scale communication-efficient training of mixture-of-experts models in production. *arXiv preprint arXiv:2505.11432*, 2025.

[26] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. {ATP}: In-network aggregation for multi-tenant learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 741–761, 2021.

[27] Liliana Laranjo, Adam G Dunn, Huong Ly Tong, Ahmet Baki Kocaballi, Jessica Chen, Rabia Bashir, Didi Surian, Blanca Gallego, Farah Magrabi, Annie YS Lau, et al. Conversational agents in healthcare: a systematic review. *Journal of the American Medical Informatics Association*, 25(9):1248–1258, 2018.

[28] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning, 2022.

[29] Chang-Chi Lee, CP Hung, Calvin Cheung, Ping-Feng Yang, Chin-Li Kao, Dao-Long Chen, Meng-Kai Shih, Chien-Lin Chang Chien, Yu-Hsiang Hsiao, Li-Chieh Chen, et al. An overview of the development of a gpu with integrated hbm on silicon interposer. In *2016 IEEE 66th Electronic Components and Technology Conference (ECTC)*, pages 1439–1444. IEEE, 2016.

[30] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R Tallent, and Kevin J Barker. Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):94–110, 2019.

[31] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *IEEE Trans. Parallel Distrib. Syst.*, 31(1):94–110, January 2020.

[32] Baojia Li, Xiaoliang Wang, Jingzhu Wang, Yifan Liu, Yuanyuan Gong, Hao Lu, Weizhen Dang, Weifeng Zhang, Xiaojie Huang, Mingzhuo Chen, et al. Tccl: Co-optimizing collective communication and traffic routing for gpu-centric clusters. In *Proceedings of the 2024 SIGCOMM Workshop on Networks for AI Computing*, pages 48–53, 2024.

[33] Xiaoxi Li, Jiajie Jin, Yujia Zhou, Yuyao Zhang, Peitian Zhang, Yutao Zhu, and Zhicheng Dou. From matching to generation: A survey on generative information retrieval. *ACM Transactions on Information Systems*, 43(3):1–62, 2025.

[34] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, page 39–55, 2008.

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

[36] Na Liu, Liangyu Chen, Xiaoyu Tian, Wei Zou, Kaijiang Chen, and Ming Cui. From llm to conversational agent: A memory enhanced architecture with fine-tuning of large language models, 2024.

[37] Shuo Liu, Qiaoling Wang, Junyi Zhang, Wenfei Wu, Qinliang Lin, Yao Liu, Meng Xu, Marco Canini, Ray CC Cheung, and Jianfei He. In-network aggregation with transport transparency for distributed training. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 376–391, 2023.

[38] Qingkai Meng, Hao Zheng, Zhenhui Zhang, ChonLam Lao, Chengyuan Huang, Baojia Li, Ziyuan Zhu, Hao Lu, Weizhen Dang, Zitong Lin, et al. Astral: A datacenter infrastructure for large language model training at scale. In *Proceedings of the ACM SIGCOMM 2025 Conference*, pages 609–625, 2025.

[39] Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, Naveen Mellempudi, Stuart Oberman, Mohammad Shoeybi, Michael Siu, and Hao Wu. Fp8 formats for deep learning, 2022.

[40] NVIDIA. Advanced API Performance: Async Copy. https://developer.nvidia.com/blog/advanced-api-performance-async-copy/.

[41] NVIDIA. Collective Operations. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/collectives.html.

[42] NVIDIA. CUDA C++ Programming Guide — CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[43] NVIDIA. NVIDIA ConnectX-7 Adapter Cards User Manual. https://docs.nvidia.com/networking/display/connectx7vpi.

[44] NVIDIA. NVIDIA DRIVE OS 5.1 Linux SDK. https://docs.nvidia.com/drive/drive_os_5.1.6.1L/nvvib_docs/index.html#page/DRIVE_OS_Linux_SDK_Development_Guide/Graphics/nvsci_nvsciipc.html.

[45] NVIDIA. PTX ISA 9.0 documentation. https://docs.nvidia.com/cuda/parallel-thread-execution/.

[46] NVIDIA. NVIDIA Hopper Architecture In-Depth. https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth, 2022.

[47] NVIDIA. NVIDIA NVLink and NVLink Switch. https://www.nvidia.com/en-us/data-center/nvlink/, 2025.

[48] Markthub Pak, Dinan Jim, Potluri Sreeram, and Howell Seth. Improving Network Performance of HPC Systems Using NVIDIA Magnum IO NVSHMEM and GPUDirect Async, 2025.

[49] Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, et al. Alibaba hpn: A data center network for large language model training. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 691–706, 2024.

[50] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale, 2022.

[51] Siddharth Samsi, Dan Zhao, Joseph McDonald, Baolin Li, Adam Michaleas, Michael Jones, William Bergeron, Jeremy Kepner, Devesh Tiwari, and Vijay Gadepally. From words to watts: Benchmarking the energy costs of large language model inference, 2023.

[52] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

[53] Gemini Team. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities, 2025.

[54] Kimi Team, Yifan Bai, and Yiping Bao *et al.*. Kimi k2: Open agentic intelligence, 2025.

[55] Qwen Team. Qwen3: A series of large language models, 2025.

[56] Mellanox Technologies. RoCE in the data center. https://network.nvidia.com/files/pdf/whitepapers/roce_in_the_data_center.pdf.

[57] Nandan Thakur, Nils Reimers, Andreas Rücklé, Abhishek Srivastava, and Iryna Gurevych. Beir: A heterogenous benchmark for zero-shot evaluation of information retrieval models. *arXiv preprint arXiv:2104.08663*, 2021.

[58] Shin-Yeh Tsai and Yiying Zhang. Lite kernel rdma support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 306–324, 2017.

[59] Lorainne Tudor Car, Dhakshenya Ardhithy Dhinagaran, Bhone Myint Kyaw, Tobias Kowatsch, Shafiq Joty, Yin-Leng Theng, and Rifat Atun. Conversational agents in health care: scoping review and conceptual analysis. *Journal of medical Internet research*, 22(8):e17158, 2020.

[60] Shibo Wang and Pankaj Kanwar. Bfloat16: The secret to high performance on cloud tpus. *Google Cloud Blog*, 4(1), 2019.

[61] Zilong Wang, Layong Luo, Qingsong Ning, Chaoliang Zeng, Wenxue Li, Xinchen Wan, Peng Xie, Tao Feng, Ke Cheng, Xiongfei Geng, Tianhao Wang, Weicheng Ling, Kejia Huo, Pingbo An, Kui Ji, Shideng Zhang, Bin Xu, Ruiqing Feng, Tao Ding, Kai Chen, and Chuanxiong Guo. SRNIC: A scalable architecture for RDMA NICs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1–14, Boston, MA, April 2023. USENIX Association.

[62] Ying Wei, Yi Chieh Huang, Haiming Tang, Nithya Sankaran, Ish Chadha, Dai Dai, Olakanmi Oluwole, Vishnu Balan, and Edward Lee. 9.3 nvlink-c2c: A coherent off package chip-to-chip interconnect with 40gbps/pin single-ended signaling. In *2023 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 160–162. IEEE, 2023.

[63] Guanbin Xu, Zhihao Le, Yinhe Chen, Zhiqi Lin, Zewen Jin, Youshan Miao, and Cheng Li. {AutoCCL}: Automated collective communication tuning for accelerating distributed and parallel {DNN} training. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 667–683, 2025.

[64] Chenggang Zhao, Shangyan Zhou, Liyue Zhang, Chengqi Deng, Zhean Xu, Yuxuan Liu, Kuai Yu, Jiashi Li, and Liang Zhao. Deepep: an efficient expert-parallel communication library. https://github.com/deepseek-ai/DeepEP, 2025.

[65] Changgang Zheng, Haoyue Tang, Mingyuan Zang, Xinpeng Hong, Aosong Feng, Leandros Tassiulas, and Noa Zilberman. Dinc: Toward distributed in-network computing. *Proceedings of the ACM on Networking*, 1(CoNEXT3):1–25, 2023.

[66] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving, 2024.

[67] Yang Zhou, Zhongjie Chen, Ziming Mao, ChonLam Lao, Shuo Yang, Pravein Govindan Kannan, Jiaqi Gao, Yilong Zhao, Yongji Wu, Kaichao You, et al. An extensible software transport layer for gpu networking. *arXiv preprint arXiv:2504.17307*, 2025.

[68] Ruidong Zhu, Ziheng Jiang, Chao Jin, Peng Wu, Cesar A. Stuardo, Dongyang Wang, Xinlei Zhang, Huaping Zhou, Haoran Wei, Yang Cheng, Jianzhe Xiao, Xinyi Zhang, Lingjun Liu, Haibin Lin, Li-Wen Chang, Jianxi Ye, Xiao Yu, Xuanzhe Liu, Xin Jin, and Xin Liu. Megascale-infer: Efficient mixture-of-experts model serving with disaggregated expert parallelism. In *Proceedings of the ACM SIGCOMM 2025 Conference*, SIGCOMM '25, page 592–608, New York, NY, USA, 2025. Association for Computing Machinery.

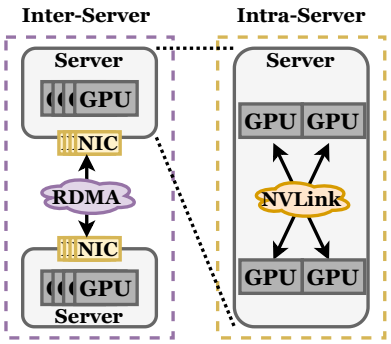


Figure 19: GPU Interconnection.

A GPU Architecture and Interconnects

This appendix provides expanded technical details on GPU architecture components and interconnect technologies.

Streaming Multiprocessor and Thread Execution. Each SM manages a large number of lightweight threads, organized into *thread blocks*—the fundamental unit of programmer-defined parallelism. Threads within a block are grouped into *warps*, the smallest scheduling and execution unit, typically containing 32 threads. All threads in a warp execute the same instruction simultaneously on different data under the SIMT (Single Instruction, Multiple Threads) model. A warp cannot execute multiple tasks concurrently, even if some threads are idle due to branching. Each SM contains a fixed-size *register file* for private per-thread data and an on-chip memory hierarchy of *L1 cache* and software-managed *shared memory*. The shared memory is accessible by all threads in a block and has the same lifetime as the block.

Tensor Memory Accelerator (TMA). TMA is an asynchronous copy engine enabling high-efficiency bidirectional transfers between global and shared memory. For store operations (shared to global memory), it natively supports element-wise *reduce* operations (*e.g.*, *add*, *min*, *max*) and bit-wise *and/or* for common data types such as FP32 and BF16. Its key advantages include: (1) eliminating the use of SM registers for address calculation and data staging during memory movement; (2) drastically reducing thread participation—only one thread per warp is needed to initiate a TMA operation versus requiring all warp threads in traditional copies; (3) inherent asynchrony, which releases the initiating thread immediately to perform other computations while the transfer is handled by dedicated hardware.

GPU Interconnect. Figure 19 illustrates the two-domain interconnect hierarchy. *Intra-server interconnects*, such as *NVLink/NVSwitch* [47], provide high-bandwidth communication (e.g., hundreds of GB/s per GPU) among GPUs within a

single server, typically supporting up to 8 devices. For peer-to-peer transfers, a GPU exposes a dedicated *NVLink buffer* memory address space to other GPUs in the server, enabling direct memory-semantic operations.

Inter-server interconnects rely on high-performance networking like InfiniBand or RDMA over Converged Ethernet (RoCE) [10, 56], offering lower bandwidth per NIC (e.g., tens of GB/s) but essential for multi-server scalability. Before RDMA communication, a GPU driver pre-registers a memory region as an *RDMA buffer*, allowing the RNIC to perform direct read/write operations for GPU-to-GPU transfers without host CPU involvement. This hierarchical interconnect enables efficient token exchange in large-scale deployments.

B Implementaion Details

This appendix provides further implementation details for the key techniques introduced in SwiftEP.

Multi-SGE Transmission. The multi-SGE transmission mechanism extends the IBGDA interface to support scatter-gather operations across non-contiguous memory regions. Multiple threads are employed to populate each Scatter-Gather Element (SGE) concurrently. If the total size of the data segments exceeds the Work Queue (WQ) length, a ring buffer is used to wrap the data appropriately, ensuring continuous transmission without overflow.

QP Transmission Parallelization. By adopting the IBGDA interface in place of IBRC, SwiftEP gains finer control over Queue Pair (QP) configuration. During RDMA connection establishment, the number of QPs is explicitly set according to the number of predefined communication channels. This allows the system to scale in alignment with application communication requirements, improving concurrency and throughput.

TMA Remote GPU Memory Access. The TMA interface is abstracted into modular load and store operations using PTX instructions. These operations are combined to construct end-to-end NVLink data transmission procedures between GPUs. This encapsulation offers a consistent and high-level interface for remote memory access, which is utilized across all inter-GPU communication tasks within SwiftEP, improving both usability and performance.