Every μs Matters: Achieving Near Speed-of-Light Latency in GPU Collectives

Siyuan Shen*
ETH Zürich
Zürich, Switzerland
siyuan.shen@inf.ethz.ch

Anton Korzh
NVIDIA Corporation
Santa Clara, California
akorzh@nvidia.com

John Bachan
NVIDIA Corporation
Santa Clara, California
jbachan@nvidia.com

Tiancheng Chen
ETH Zürich
Zürich, Switzerland
tiancheng.chen@inf.ethz.ch

Arnav Goel
NVIDIA Corporation
Santa Clara, California
arnavg@nvidia.com

Ludwig Schneider
NVIDIA Corporation
Santa Clara, California
lschneider@nvidia.com

Pouya Kousha
NVIDIA Corporation
Santa Clara, California
pkousha@nvidia.com

Zhenhao He
NVIDIA Corporation
Zürich, Switzerland
zhenhaoh@nvidia.com

Sylvain Jeaugey
NVIDIA Corporation
Grenoble, France
sjeaugey@nvidia.com

Kamil Iskra
NVIDIA Corporation
Chicago, Illinois
kiskra@nvidia.com

Nishank Chandawala
NVIDIA Corporation
Santa Clara, California
nchandawala@nvidia.com

Jeff R. Hammond
NVIDIA Helsinki Oy
Helsinki, Finland
jeffpapers@nvidia.com

Torsten Hoefler
ETH Zürich
Zürich, Switzerland
torsten.hoefler@inf.ethz.ch

arXiv:2607.16100v1 [cs.DC] 17 Jul 2026

***Abstract*—GPU collective communication is typically optimized for bandwidth, yet many emerging workloads are increasingly limited by latency. Long-context decode-heavy large language model (LLM) inference is a prime example, where serving large models requires multiple GPUs, and many small collectives lie directly on the critical path of token generation. Therefore, even μs of overhead can impact performance and cost. In this work, we study how to approach the hardware Speed-of-Light (SoL) lower bound for GPU collectives within a scale-up network. We identify key principles for near-optimal designs, including barrier-free synchronization and efficient use of symmetric memory and multicast. Building on NCCL's device-side API, we develop low-latency interfaces for constructing custom collective kernels and use them to implement new symmetric collectives in NCCL. Microbenchmarks show substantial latency reductions for small and medium messages, reducing overhead to within 7% of the absolute SoL lower bound. When integrated into real applications, these kernels improve inter-token latency and throughput in LLM inference and accelerate cuSOLVERMp, demonstrating benefits for both AI inference and traditional HPC workloads.**

I. Introduction

Deep learning has driven rapid growth in GPU cluster scale and interconnect capability. As large language models (LLMs) scale, inference increasingly spans multiple GPUs, bringing collective communication onto the critical path of token generation. For example, serving DeepSeek-V3-class models [1] requires at least 8×H200 GPUs [2]. The impact of collective communication is especially evident in decode-heavy workloads, where modern systems may generate millions of tokens per request in applications such as code generation and agent-style workflows [3], [4]. In these settings, even small communication overheads accumulate and directly impact the quality of service. In addition, for long-context inference, the KV-cache memory grows with sequence length, and the batch size is often reduced to fit within device memory. As a result, collectives such as AllReduce are invoked frequently with relatively small message sizes during decoding, making latency, rather than bandwidth, the dominant bottleneck. Consequently, recent inference frameworks such as vLLM [3], SGLang [5],

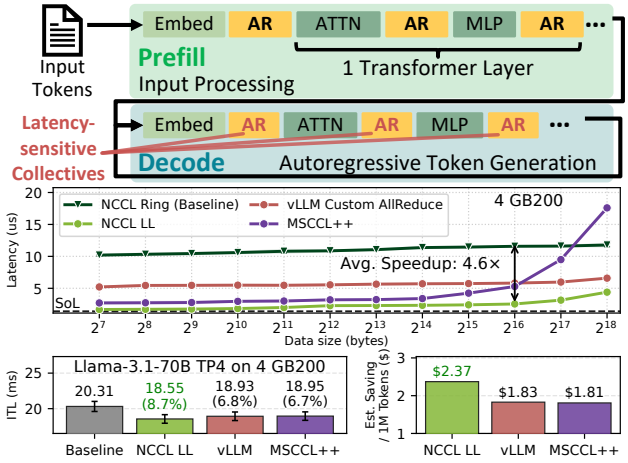

Fig. 1: In long-context, small-batch tensor-parallel (TP) LLM inference, many small AllReduce operations lie on the critical path, making collective latency a crucial bottleneck. The microbenchmark shows that our NCCL low-latency kernel reduces small-message AllReduce latency relative to other implementations, approaching the speed-of-light (SoL) bound. This translates into lower inter-token latency (ITL) and higher cost savings for Llama-3.1-70B inference.

*The majority of this work was done during an internship at NVIDIA.

and TensorRT-LLM [6] treat communication latency as a first-class optimization target and implement custom low-latency GPU kernels for operations such as AllReduce.

Low-latency collectives are essential not only for LLM inference, but also for many traditional scientific and HPC applications. Many simulations and solvers perform frequent small global reductions within tightly synchronized phases, such as time-stepping loops and particle simulations. In these settings, collective latency lies on the critical path and can limit strong scaling. Prior work such as LLAMP shows that widely used HPC workloads, including MILC and LULESH, are measurably sensitive to collective latency [7]. At the same time, many GPU-accelerated scientific applications do not yet fully exploit GPU-native communication libraries. These observations indicate that low-latency GPU collectives can improve not only LLM inference, but also scalability and time-to-solution in traditional scientific workloads.

Despite this growing recognition, existing approaches still leave performance on the table. We observed that even the best available implementations often remain above the **"speed-of-light"** (SoL) bound, by which we mean the absolute hardware lower bound imposed by the interconnect and memory system. Figure 1 illustrates this effect for the long-context, small-batch decode setting that we target. On 4 GB200 GPUs, the NCCL low-latency kernel we introduce in this work reduces average latency for small messages from 11.0 μs for NCCL ring to 2.37 μs, yielding an 8.7% ITL reduction for the inference workload of Llama-3.1-70B. Using CoreWeave's on-demand price of $42/hour for 4 GB200 [8] and converting output throughput into cost per 1M output tokens, the measured data implies that each μs removed from AllReduce latency reduces cost by about 0.9%. While this saving may appear insignificant, it compounds into substantial cost reduction at the trillion-token scale of modern LLM services [4], [9].

In this work, we begin by identifying global memory barriers across participating GPUs as a key source of latency in existing collective implementations. To this end, we present several techniques, including LL, sentinel-based synchronization, double buffering, and a novel two-shot AllReduce algorithm. By combining these techniques, we eliminate expensive global memory barriers entirely while preserving correctness and efficiency. Building on these, we develop a set of experimental application programming interfaces (APIs) on top of NCCL's latest device communication APIs that encapsulate these low-latency mechanisms into reusable primitives for efficiently prototyping custom low-latency kernels. Leveraging this interface, we implement several new AllReduce kernels within NCCL that are directly usable in practice.

To evaluate these techniques and the resulting collectives, we conduct detailed microbenchmarks showing that our designs approach the hardware SoL latency bound across a wide range of node configurations. We also integrate our low-latency kernels into real workloads, including vLLM and cuSOLVERMp. These case studies demonstrate consistent and measurable performance improvements over standard NCCL collectives and other state-of-the-art frameworks, confirming the practical benefits of latency-centric collective optimization for both LLM inference and traditional HPC workloads.

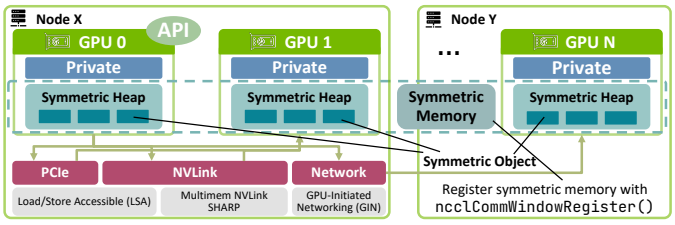

Fig. 2: Overview of device-initiated communication and symmetric memory in NCCL [19]. When GPUs are in the same node or NVLink domain, LSA operations are supported over PCIe and NVLink, while multimem operations are enabled via NVLink SHARP for hardware-accelerated multicast and reduction. For inter-node communication, GPU-initiated networking (GIN) supports GDAKI and proxy-assisted data transfers over InfiniBand and RoCE [20].

Our contributions in this work are as follows:

- We systematically analyze existing techniques for reducing collective latency, characterize where they are effective, and compose them into barrier-free designs that approach the hardware lower bound.
- We design and implement a set of low-latency communication APIs on top of NCCL's device-side APIs to facilitate development of custom collective kernels.
- We develop new low-latency AllReduce algorithms, including a novel two-shot LL128 atomic design, using the proposed low-latency APIs.
- We conduct extensive evaluations through microbenchmarks and real workloads, including vLLM inference and cuSOLVERMp, demonstrating noticeable performance gains.

II. Background

A. GPU Communication Libraries

Efficient communication is essential in modern GPU-accelerated systems, where both AI and scientific workloads rely on tightly coupled GPU execution [10]. To support this, specialized GPU communication libraries have emerged as a critical software layer. NCCL is one of the most widely used libraries, providing collective and point-to-point operations optimized for various interconnects [11]. While NCCL primarily targets optimized collectives, NVSHMEM adopts a partitioned global address space (PGAS) model that enables one-sided communication and exposes finer device-side control [12]. Similar libraries exist across vendors, including AMD's RCCL and rocSHMEM, and Intel's oneCCL [13]–[15]. In contrast, traditional frameworks like MPI [16], originally designed for CPU-based systems, have been extended to support GPUs (e.g., CUDA-aware MPI [17]), but often underperform vendor-optimized libraries for large-scale collectives while remaining competitive for point-to-point communication [18]. These observations highlight that GPU-native communication libraries have become an essential complement to traditional frameworks in modern HPC and AI systems.

B. Device-Initiated Communication and Symmetric Memory

In addition to optimized collective primitives, modern CCLs increasingly support device-driven communication, enabling kernels to directly initiate and orchestrate data movement. Early support for this appeared in NVSHMEM, which allows kernels to perform remote memory operations (e.g., put/get) and synchronization without host involvement. More recently, NCCL 2.28 introduces device-side communication APIs that enable kernels to directly invoke communication primitives [20], [21]. These capabilities are enabled by underlying hardware and runtime support, including GPU Virtual Memory Management (VMM), which provides a unified virtual address space across GPUs, and GPUDirect Async Kernel-Initiated (GDAKI), which allows GPUs to directly interact with network interfaces without CPU intervention.

Symmetric memory originates from the SHMEM family of PGAS models. Remotely accessible data objects, called **symmetric objects**, have identical type, size, and layout on each processing element (PE), which corresponds to a GPU in this case. This allows remote access using the same logical address together with a PE identifier. These objects reside in the **symmetric heap**, which supports one-sided operations such as get, put, and atomics.

On GPUs connected through PCIe or NVLink and supported by CUDA Virtual Memory Management (VMM), symmetric memory regions can be mapped into a unified virtual address space, making them **load/store accessible (LSA)**. On systems with NVSwitch and NVLink SHARP (NVLS), **multimem load/store** instructions can further accelerate communication by enabling multicast and in-network reduction. A visualization is shown in Fig. 2. Overall, symmetric memory reduces address translation overhead and enables low-latency data exchange. Thus, NCCL is gradually replacing its collectives with symmetric-memory-based kernels, and previous implementations are now referred to as **legacy kernels** [19], [22].

We base our implementation on NVIDIA hardware and NCCL because this stack represents one of the most widely adopted platforms for GPU collectives in modern AI and HPC ecosystems [23], [24]. We chose NCCL rather than NVSHMEM since NCCL is extensively used as a communication backend across both deep learning frameworks and other scientific libraries, including PyTorch [25], TensorFlow [26], vLLM [3], cuSOLVERMp [27], and cuBLASMp [28]. This level of integration makes NCCL a more suitable choice.

Most of the design principles are, nevertheless, not NVIDIA-specific. LL, sentinel synchronization, and double buffering require GPU-initiated access to peer memory, remote writes that become visible to GPU-side polling, and device-side ordering or fence operations before buffer reuse. All platforms providing these properties can implement the proposed protocols and kernels.

Additionally, this work focuses on collectives within a scale-up network, specifically GPUs residing in the **same NVLink domain**. We exclude scale-out communication for several reasons. First, in modern LLM inference, which is a primary target of this work, parallel groups are typically confined to a single scale-up domain. Second, multi-node systems commonly employ hierarchical collectives that separate local and scale-out phases, making improvements within the scale-up domain complementary to higher-level optimizations [29], [30]. Finally, emerging GPU systems increasingly expand the size and capability of these domains, allowing a growing fraction of latency-sensitive workloads to execute entirely within a single scale-up network [31], [32].

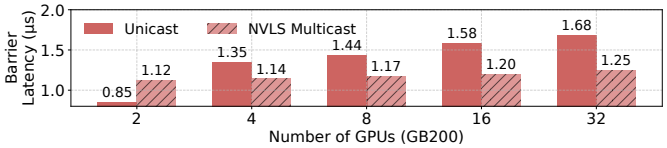

Fig. 3: Barrier latency on GB200 as a function of the number of GPUs for unicast and multicast implementations.

III. Toward Near Speed-of-Light AllReduce

In this section, we describe how we approach near speed-of-light (SoL) latency, i.e., the hardware lower bound, for **AllReduce** (R is capitalized following NCCL's notation). We focus on AllReduce because it is one of the most widely used collectives in HPC and distributed machine learning and a frequent optimization target in practice [33]–[38]. Moreover, many implementations decompose AllReduce into ReduceScatter and AllGather or Reduce and Broadcast, so techniques that minimize AllReduce latency often apply directly to these building blocks. Thus, optimizing AllReduce benefits a broader class of collectives.

A. Low-Latency AllReduce Algorithms

When the message size of an AllReduce operation is small, **its latency is primarily determined by the number of synchronizations needed**, or communication phases. Consequently, algorithms such as tree-based or recursive-doubling AllReduce, which require $O(\log N)$ rounds of synchronization for N ranks, typically outperform ring-based algorithms that require $O(N)$ rounds for small messages [11], [39]. In a scale-up network, the number of synchronizations can be reduced further to $O(1)$ using one-shot or two-shot AllReduce algorithms, which are widely adopted in most communication libraries and frameworks [3], [6], [11], [12], [30].

1) One-shot AllReduce: In a one-shot AllReduce, the entire reduction is completed in a single communication phase, where each GPU fetches data from all peers, performs the reduction locally, and writes the result to the output. In **pull** mode, GPUs read remote data via loads, while in **push** mode they write data to remote buffers before reducing locally. Push is generally faster, requiring only half a GPU-to-GPU RTT versus a full RTT for remote loads, but needs additional buffering for incoming data. Consequently, push-based one-shot designs are often preferred for latency-sensitive collectives.

2) Two-shot AllReduce: A two-shot AllReduce is decomposed into two phases. In the ReduceScatter phase, each GPU exchanges partitions of its input with peers and performs the

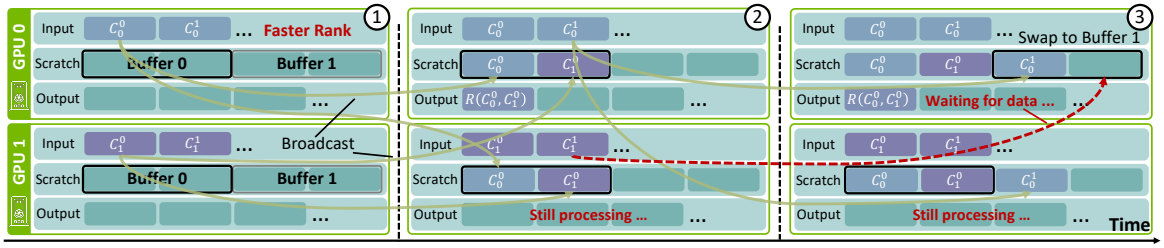

Fig. 4: Example illustrating bidirectional communication with double buffering. Two ranks exchange chunked inputs, where chunk i from rank r is denoted C_r^i. The scratch space is divided into Buffers 0 and 1, each can store two chunks. The black border marks the active buffer. Steps ①, ②, and ③ show events in chronological order, with time progressing to the right. Arrows represent cross-GPU data transfer. The mechanism is independent of whether LL or sentinel synchronization is used.

reduction on its assigned chunk, producing a partial result. Either push or pull semantics may be used in this phase. In the AllGather phase, the reduced chunks are exchanged so that every GPU has the complete result. Compared to one-shot, two-shot introduces an additional synchronization but significantly reduces the communication volume, improving performance for moderate message sizes. For N ranks reducing M bytes of data, one-shot incurs $O(N \cdot M)$ total communication volume, whereas two-shot reduces this to $O(M)$.

B. Cost of Memory Barriers

After examining several one-shot and two-shot AllReduce implementations in state-of-the-art frameworks and libraries [3], [6], [12], [30], we observe that, regardless of push or pull communication, these designs typically rely on explicit memory barriers to synchronize peers and signal data readiness at the thread-block level. Using NCCL's `ncclLsaBarrierSession`, which implements a memory barrier for load-store accessible (LSA) devices, we measure the overhead of such synchronization under relaxed memory ordering and report the results in Fig. 3. Although NCCL's implementation may not be fully optimized, alternative designs follow the same fundamental pattern: a flag is propagated to all peers, and each GPU waits until it observes the corresponding signals from every other participant. Therefore, it is representative of the inherent cost in such approaches.

The measured barrier latency shows that each barrier will incur more than 1 μs of overhead. In many AllReduce kernels, two such barriers are required. As illustrated in Fig. 1, when a small-message AllReduce completes in roughly 5 μs on four GPUs, two barrier calls alone will account for about 40% of the total latency. As emphasized earlier, when targeting near SoL performance, every μs matters. Thus, **eliminating these barriers entirely can yield substantial performance gains**.

IV. Designing Barrier-Free Collectives

Having established that memory barriers introduce non-negligible overhead, we next consider alternative synchronization mechanisms that achieve the same purpose with lower latency. As discussed in Section III-A, AllReduce can be implemented using either push or pull communication. Since our goal is to approach the SoL, we focus on **push mode**, which trades additional buffer space, referred to here as a **scratch buffer**, for roughly half of a GPU-to-GPU RTT.

1) LL: The first technique we introduce is LL, short for low latency, which originates from the LL protocol in NCCL [11] and is also used in libraries such as NVSHMEM [12] and MSCCL++ [30]. Conventional synchronization signals data arrival using explicit flags and enforced ordering between data and signals. LL removes the signaling step by packing the 8-byte flag with the 8-byte data and transmitting them atomically with 16-byte atomic stores, allowing the receiver to determine data readiness by checking the flag directly. This design halves the effective payload bandwidth and doubles scratch buffer usage, making LL mostly suitable for very small messages.

2) Sentinel: Instead of embedding a signal in the transmitted data, the receiving scratch buffer can be initialized with a sentinel value that is unlikely to appear in valid computations, such as the floating-point value `-NaN`. Data is then written directly to the buffer, and the receiver polls until the value changes from the sentinel, indicating that valid data has arrived. Compared to LL, this approach preserves full effective bandwidth and uses less scratch space, making it more efficient for moderately larger messages. However, it has a few drawbacks. First, unlike LL, which updates its flag each iteration, the sentinel method requires the buffer to be reset before reuse, complicating buffer management. Second, transmitted values must never match the sentinel because a matching value would prevent the receiver from detecting data arrival. Users must therefore exclude such values to ensure correctness.

3) Bidirectional Communication & Double Buffering: Although LL and sentinel synchronization eliminate explicit memory barriers for single exchanges, they are insufficient when messages require multiple iterations due to limited buffer space. In such cases, inputs are partitioned and processed in chunks. Conventional approaches insert barriers between iterations to prevent buffer overwrites. We can eliminate them by using bidirectional communication and double buffering.

Fig. 4 demonstrates the mechanism. Consider two ranks whose scratch space cannot hold all input data simultaneously. In step ①, both ranks begin with the scratch buffer set to Buffer 0. In step ②, they exchange their first chunks C_r^0. Because Rank 0 processes data faster, it completes the

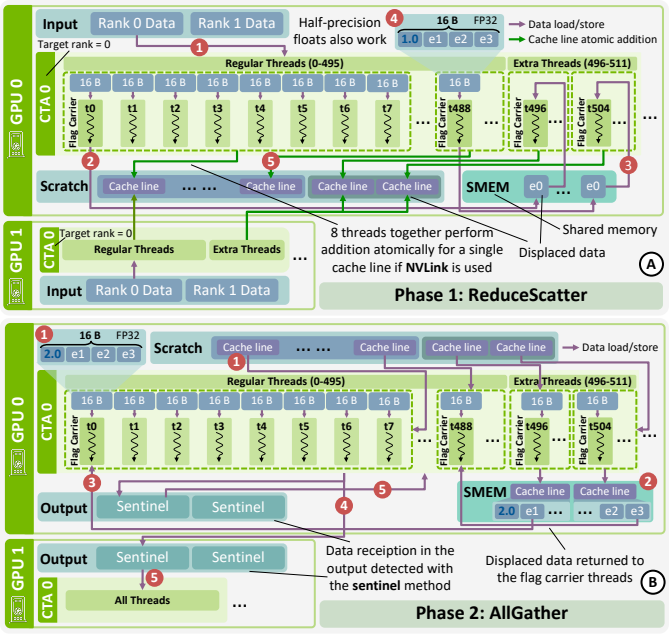

Fig. 5: Overview of the two-shot LL128 atomic AllReduce algorithm. Threads within a CTA are divided into two groups: 496 **regular threads** and 16 **extra threads**, which handle displaced elements. The dashed green boxes indicate groups of 8 threads that operate on a 128-byte cache line. Panel Ⓐ illustrates the ReduceScatter stage, while Panel Ⓑ shows the AllGather stage. The example assumes an out-of-place operation where the output buffer is initialized with sentinel values before kernel execution. To support half-precision floats, each element is only 2 bytes and only 8 extra threads are needed.

reduction R and writes the result to its output while Rank 1 is still processing. In step ③, Rank 0 advances to the next iteration and switches to Buffer 1 to broadcast chunk C_0^1. It then waits for C_1^1 before proceeding, ensuring that data in Buffer 0 is not overwritten before Rank 1 finishes reading it.

This bidirectional exchange, combined with double buffering, ensures that a rank cannot overwrite a peer's buffer for the next iteration until it has received data from that peer in the current one. This assumes that each rank communicates with a given peer at most once per iteration, avoiding multiple stores to the same remote address. In effect, *each receive from a peer serves as an implicit permission for the next send*, analogous to credit-based flow control. This mechanism allows multiple reduction iterations without costly global memory barriers.

4) LL128 Atomic AllReduce: The last technique we present for removing the global memory barrier is a new AllReduce algorithm that uses a synchronization mechanism different from LL and sentinel. Since it resembles NCCL's LL128 protocol [11] and relies on atomic additions, we refer to it as the *two-shot LL128 atomic* algorithm. An overview is shown in Fig. 5. Like the standard two-shot design, the algorithm proceeds in two phases, which we describe in detail below.

a) ReduceScatter: As in standard ReduceScatter algorithms, the input is partitioned into N chunks, where N is the number of GPUs. To exploit GPU parallelism, CTAs are

AllReduce Algorithm	Comm. Volume per GPU	Latency, # Synchronizations	Scratch Space / Iter	Deterministic
One-shot (LL)	$2(N-1)M$	1	$2ND$	✔
One-shot (Sentinel)	$(N-1)M$	1	ND	✔
Two-shot (LL)	$4(N-1)\frac{M}{N}$	2	$2D$	✔
Two-shot (Sentinel)	$2(N-1)\frac{M}{N}$	2	D	✔
Two-shot (LL128 Atomic)	$\approx 2(N-1)\frac{M}{N}$	2	$\approx \frac{D}{N}$	✖

TABLE I: Comparison of low-latency AllReduce algorithms. N denotes the number of GPUs, M the total message size, and D the amount of data reduced per iteration. Latency is expressed as the number of synchronizations required per iteration. "*Scratch Space / Iter*" denotes the scratch buffer capacity required to reduce D bytes of data per iteration.

evenly distributed across ranks, and each CTA is assigned a **target rank**. For example, in Fig. 5, CTA 0 on both GPU 0 and 1 processes the chunk belonging to GPU 0. Each CTA then reads its assigned partition and performs the steps below.

1 Threads operate in groups of 8, and each thread processes 16 bytes from the assigned partition. For FP32 data, each thread handles 4 elements ($e0$–$e3$). Together, the 8 threads operate on 128 bytes, which matches the size of a cache line.

2 Within each group, the first thread acts as the **flag carrier**. It moves its first element e_0 into a shared memory region reserved for displaced values. A `__syncthreads()` then ensures that the displaced data is visible to all threads.

3 Threads in the extra-thread group read the displaced elements from shared memory.

4 Each flag carrier sets the first element of its vector to 1.

5 All threads then perform an **atomic add** to the scratch buffer corresponding to the target rank. NVLink ensures that these operations are applied atomically at the cache-line level.

b) AllGather: Since NVLink performs 128-byte writes atomically, when the first element of a cache line (the flag) equals the number of ranks N, it indicates that all ranks have contributed. The AllGather phase then proceeds as follows.

1 Each CTA whose target rank matches its own rank polls the corresponding region in the scratch buffer. Within each group of eight threads, the flag carrier repeatedly checks the first element of its vector until the value equals N.

2 Once the data is confirmed to be ready, the extra threads write their 16-byte displaced elements into shared memory. A `__syncthreads()` is executed afterwards to ensure visibility of data.

3 The flag carriers from the regular thread group then read the displaced elements from shared memory, restoring the correct element within their vectors.

4 All regular threads write their data to the corresponding region of the output buffer.

5 CTAs then poll the output buffer partitions associated with their target ranks until the complete data becomes available.

5) Algorithm Comparison: The two-shot LL128 atomic algorithm was proposed to improve scalability. The motivation is twofold. First, atomic additions synchronized at the L2

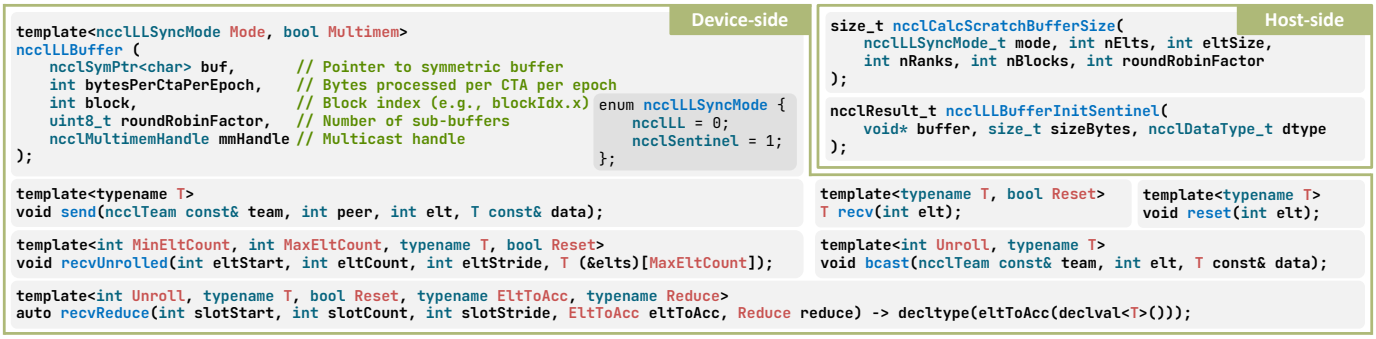

Fig. 6: Overview of the proposed low-latency API, showing device-side and host-side functions.

cache may be cheaper than waiting for all data and performing reductions within CTAs. Second, the required scratch buffer space is only $\frac{D}{N}$. The algorithm also wastes far less bandwidth than LL. For 32-bit floats, it requires only 4 extra bytes per 128 bytes of data ($\approx 3\%$). For 16-bit floats, it requires 2 extra bytes per 128 bytes ($\approx 1.5\%$).

The algorithm has several limitations. It requires NVLink to guarantee cache-line–level atomic addition and supports only single- and half-precision types due to the availability of vectorized atomics in CUDA [40], [41]. It is also limited to addition, as the embedded flag relies on commutativity, and CUDA does not provide vectorized atomic multiplication. In practice, this is not too restrictive since addition dominates most target workloads, such as LLM inference. Finally, the algorithm is non-deterministic because floating-point atomic ordering is not guaranteed. A one-shot variant is possible, but different ranks may observe different results, which violates AllReduce semantics [16]. The trade-offs of different low-latency algorithms are summarized in Table I.

In terms of numerical stability, the algorithm obeys the standard forward-error bound for floating-point summation. For N ranks and unit roundoff u in the accumulation format, assuming no overflow or underflow,

$$\left| \mathrm{fl}\left(\sum_{i=1}^{N} x_i\right) - \sum_{i=1}^{N} x_i \right| \le \gamma_{N-1} \sum_{i=1}^{N} |x_i|, \ \gamma_k = \frac{ku}{1-ku}.$$

As an example, for FP32 and 64 ranks, the worst-case coefficient is $\gamma_{63} \approx 3.8 \times 10^{-6}$. The bound is larger for FP16 and BF16. Thus, LL128 atomic should be treated as a performance-oriented option for workloads that tolerate the precision of the selected accumulation format.

V. Low-Latency API Design

Building on the techniques described previously, we translate them into APIs designed to meet three requirements. ① Compatibility with NCCL's existing device-side interface. ② Encapsulation of the low-latency techniques introduced earlier through a unified abstraction. ③ Sufficient flexibility to serve as building blocks for custom communication kernels. Guided by these principles, we develop a set of experimental low-latency APIs and integrate them into NCCL. Figure 6 provides an overview. Due to space constraints, we describe only the key primitives below. Full documentation and additional examples are available in the released source code[1].

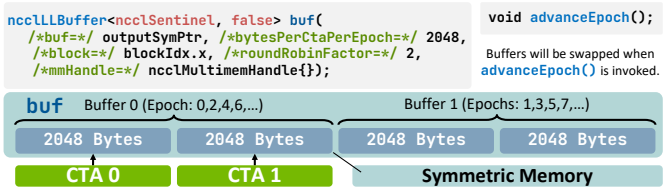

Fig. 7: Example construction and layout of a `ncclLLBuffer` object. Each CTA is assigned a fixed-size region per epoch, while epochs alternate between sub-buffers. Invoking `advanceEpoch()` switches the active sub-buffer and increments the internal `epoch` value.

a) `ncclLLBuffer`: To improve flexibility, we adopt a **buffer-centric** design for the new APIs, centered on the device-side `ncclLLBuffer` object. This abstraction wraps arbitrary symmetric memory and exposes low-latency primitives that operate directly on the buffer. A unified interface supports both LL and sentinel modes, selectable via the `ncclLLSyncMode` template parameter at initialization, allowing users to switch synchronization mechanisms with minimal changes. The LL128 method is not included because it requires 8 threads to operate as a unit, which does not fit the thread-level API design and is incompatible with the proposed primitives. A second template parameter controls whether NVLS multicast instructions are used.

To support multi-buffered execution, `ncclLLBuffer` organizes memory according to parameters such as `bytesPerCtaPerEpoch` and `roundRobinFactor`. Buffer addresses are derived from the current epoch and CTA index, allowing different iterations to operate on disjoint memory regions. Figure 7 shows an example partitioning of a symmetric buffer under this scheme. When `roundRobinFactor` is set to 0, the buffer is no longer subdivided, the offset remains fixed, and `advanceEpoch()` performs no operation. In this mode, users need to manage buffer offsets explicitly. The object also maintains an internal `epoch` value, which is used as the flag transmitted alongside the data when the LL protocol is selected.

[1]https://github.com/ss16118/low-latency-nccl

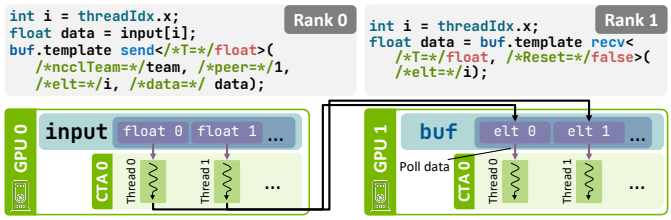

Fig. 8: Illustration of `send()` and `recv()` of a `ncclLLBuffer` for a single CTA.

b) send: The `send` primitive writes a value into a specified peer slot. The destination layout is determined by the type `T`: each element occupies `sizeof(T)` bytes, or $2 \times$ `sizeof(T)` for LL mode. This layout is used consistently across all API functions templated on `T`. In LL mode, the payload is packed with its flag whose size also equals `sizeof(T)`. In sentinel mode, the receiving buffer must be initialized with sentinel values prior to transmission, and it is the user's responsibility to ensure this.

c) recv: The `recv` primitive polls a single buffer slot until valid data is observed, according to the chosen synchronization mode. In sentinel mode, it waits until the value differs from the sentinel, whereas in LL mode it waits until the flag matches the current epoch. The template parameter `T` used for `recv` must match that of the corresponding `send` operation. After reading, the slot can optionally be reset for reuse, as controlled by the `Reset` parameter. The visibility of the reset value is not guaranteed after the function returns. Users can enforce visibility by issuing a `__threadfence()`.

d) recvUnrolled: `recvUnrolled` extends `recv` to support receiving multiple elements with compile-time unrolling. It allows users to specify minimum and maximum element counts, enabling the compiler to generate optimized polling code. Elements in the range `[0, MinEltCount)` are always accessed, while elements in `[MinEltCount, MaxEltCount)` are accessed conditionally based on `eltCount`. This design is especially useful when receiving data from multiple peers and can significantly improve performance when `eltCount` is known in advance.

e) recvReduce: The `recvReduce` primitive combines reception and reduction. It receives elements from multiple peers, converts them to an accumulator type, and applies a user-defined reduction operator, returning the accumulated result. Internally, it leverages `recvUnrolled()` for data reception and inherits its compile-time unrolling parameters.

f) bcast: The `bcast` primitive writes a value to all peers simultaneously. When multicast support is enabled, the implementation uses hardware multicast to distribute data in a single operation. Otherwise, it iterates over peers. The template unroll factor controls loop expansion for improved throughput when broadcasting to multiple ranks.

g) reset: The `reset` operation clears a specified buffer slot. In sentinel mode, it restores the sentinel value corresponding to the given type `T`, whereas in LL mode it sets the slot contents to zero. Resetting does not guarantee global visibility, and users may enforce it with an explicit memory fence. Although functions such as `recv()` and `recvReduce()` can optionally perform a reset after data reception, providing a dedicated primitive offers finer control over the buffer and allows users to decouple it from data reception when needed. The `resetRange()` function supports resetting multiple slots, but is omitted here for brevity.

h) Host-side Functions: The API also provides host-side utilities. The function `ncclCalcScratchBufferSize()` computes the minimum buffer size needed for a given configuration, accounting for element size, rank count, CTA count, synchronization mode, and the round-robin factor, which determines the number of sub-buffers required for double or multiple buffering. The initialization function `ncclLLBufferInitSentinel()` prepares the given buffers for sentinel mode by filling them with type-specific sentinel values.

The host interface is intentionally minimal to simplify the development of custom kernels. Users only need to allocate and initialize symmetric buffers as usual, without additional orchestration. Once wrapped by the device-side abstraction, all necessary low-latency primitives will be exposed, allowing developers to focus on algorithm design rather than the setup.

i) Constraints: The low latency APIs presented here are intentionally fine grained. Unlike NVSHMEM, which provides APIs at the thread, warp, and block levels, our design exposes only thread level primitives. This choice maximizes user control and allows implementations to achieve the lowest possible latency. Moreover, the APIs favor bidirectional communication patterns to achieve safety and optimal performance. As an example, algorithms with independent per-iteration communication, such as one-shot and two-shot AllReduce, can operate without barriers because each step uses disjoint buffers and each rank both sends and receives within the same iteration. In contrast, algorithms with inter-step dependencies and without bidirectional communication, such as ring AllReduce, still require explicit memory barriers. In a ring schedule, each rank must receive a chunk before reducing and forwarding it, as advancing early risks overwriting unconsumed data.

j) NCCL Collective Integration: Using the proposed APIs, we implement one-shot, two-shot, and two-shot LL128 atomic AllReduce, along with other collectives. Taking AllReduce as an example, algorithms can be selected via `NCCL_SYM_KERNEL`: `AllReduce_LLBuffer` for one-shot, `AllReduce_LLBuffer_Twoshot` for two-shot, and `AllReduce_LL128_Atomic` for the LL128 atomic variant. The synchronization mode is controlled by `NCCL_SYM_LLBUFFER_SYNC`, allowing users to switch between LL and sentinel mechanisms for the same algorithm. Additionally, two-shot AllReduce requires symmetric (LSA) output buffers, while one-shot only relies on symmetric scratch buffers initialized by NCCL and has no such requirements.

The released artifact also includes low-latency Broadcast, Reduce, ReduceScatter, and AllGather kernels constructed from the same LL and sentinel primitives.

```
ncclLLBuffer<ncclLL, false> llBuf(
/*buf=*/ scratchSymPtr,
/*bytesPerCtaPerEpoch=*/ bytesPerCtaPerEpoch,
/*block=*/ blockIdx.x,                                   Initialize buffer
/*roundRobinFactor=*/ 2, // Double buffering
/*mmHandle=*/ ncclMultimemHandle{}
);
for (int i = tid; i < nElts; i += nthreads) {
   float data = inputBuf[i];
   int slot = threadIdx.x + rank * blockDim.x;   Load and Broadcast
   llBuf.template bcast<4, float>(team, slot, data);

   float result = llBuf.template recvReduce<4, float, false>(
   /*eltStart=*/ threadIdx.x,
   /*eltCount=*/ nRanks,
   /*eltStride=*/ blockDim.x,                    Receive and reduce
   /*eltToAcc=*/ [](float val) -> float { return val; },
   /*reduce=*/ [](float a, float b) -> float { return a + b; },
   );
   outputBuf[i] = result;
   llBuf.advanceEpoch();
}
```

Fig. 9: Example implementation of a one-shot AllReduce using the low-latency API with `ncclLL` synchronization. Colored regions highlight the three stages of the algorithm.

A. Example One-shot AllReduce

Fig. 9 shows a one-shot AllReduce implemented with the proposed low-latency API that uses `ncclLL` synchronization and supports arbitrary message sizes. A `ncclLLBuffer` is first constructed over symmetric scratch memory with double buffering. Each thread iterates over its assigned elements, loads a value from the input buffer, and invokes `bcast` to distribute it to peer slots. It then calls `recvReduce` to poll, receive, and reduce values from all ranks. The result is written to the output buffer, and `advanceEpoch()` switches to the next sub-buffer. This example illustrates that the API expresses collective algorithms succinctly while reducing the effort required to implement custom low-latency kernels.

VI. Measuring the Speed-of-Light of AllReduce

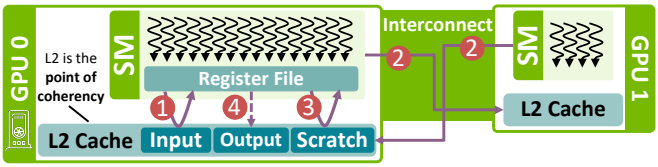

Fig. 10: Minimal data movement in an AllReduce. All buffers are assumed to reside in L2, which serves as the point of coherency (PoC) across GPUs in NVIDIA's systems. **SM** (streaming multiprocessor) is used instead of CTA to emphasize the hardware unit executing the memory operations.

In this section, we describe how the SoL lower bound can be estimated. We define the SoL as the minimal data movement required to complete the AllReduce while ignoring **all other overheads**, such as instruction scheduling and computation. Since small-message latency is determined by the transfer of the smallest unit handled by the memory system, we focus on the movement of a single 128-byte cache line.

Fig. 10 illustrates the minimal data movements required for an AllReduce operation. Since remote stores are faster than remote loads and sufficient buffer space is assumed, the SoL bound corresponds to a one-shot push algorithm. The resulting data movement consists of the following components.

1. The first step loads the data into the SM register file. To obtain the lower bound, we assume an L2 cache hit, which is reasonable since the message sizes considered typically fit in L2. This load incurs one L2 RTT, denoted as $L_{\text{L2\_RTT}}$.
2. The data is then broadcast to the scratch buffers of all GPUs. For the SoL estimate, we assume that the stores to all peers are issued simultaneously. The data becomes visible in the remote GPU L2 cache after a latency of $L_{\text{remote\_store}}$. The cost of writing to the local scratch buffer is ignored because it is much smaller than the latency of remote stores.
3. Once the data arrives, it is immediately loaded from L2 into the SM. Assuming an L2 hit, this incurs another $L_{\text{L2\_RTT}}$. Contributions from all peers are assumed to arrive simultaneously and are fetched in parallel.
4. Reduction is performed inside the SM, and the result is written to the output buffer. The latency of this final store is not included because once the store instruction is issued, the memory system will finish the write in the background.

Combining the components above, the SoL latency of an AllReduce can be expressed as

$$L_{\text{SoL}} = 2L_{\text{L2\_RTT}} + L_{\text{remote\_store}}.$$

Under the SoL assumption, sending data to N peers incurs the same latency as sending to a single peer. Therefore, L_{SoL} **represents an absolute lower bound that is independent of the number of ranks involved**.

To measure $L_{\text{L2\_RTT}}$, we benchmark the latency of a single `__threadfence()`. This instruction enforces ordering and visibility at the L2 level, forcing the SM to wait until outstanding memory transactions reach the cache. Hence, its latency provides a good approximation of the L2 RTT.

To estimate $L_{\text{remote\_store}}$, we measure the RTT when a single value is ping-ponged between two GPUs, which can be decomposed as $L_{\text{ping\_pong}} = 2L_{\text{L2\_RTT}} + 2L_{\text{remote\_store}}$. Each round-trip involves one remote store to the peer GPU and one remote store in the return direction, with an L2 access on both sides. Therefore, we can estimate $L_{\text{remote\_store}}$ as

$$L_{\text{remote\_store}} = (L_{\text{ping\_pong}} - 2L_{\text{L2\_RTT}})/2.$$

On two GB200, we measured the $L_{\text{L2\_RTT}}$ and $L_{\text{remote\_store}}$ as 0.306 µs and 0.792 µs, respectively. Therefore, the SoL latency of an AllReduce is computed to be **1.404 µs**.

VII. Microbenchmarks

A. Experimental Setup

We evaluate microbenchmarks on a GB200 NVL72 system, where 4 Blackwell GPUs reside within a node and 72 GPUs are connected within a single NVLink domain with 130 TB/s aggregate bandwidth. To ensure reproducibility, we use the NVIDIA vLLM container (v26.02) [42], which includes Ubuntu 24.04, CUDA 13.1, vLLM 0.15.1, PyTorch 2.11, and OpenMPI 4.1.9. Each experiment is repeated over 10 trials, and we report the mean. Error bars denote standard deviation and are typically too small to be visible.

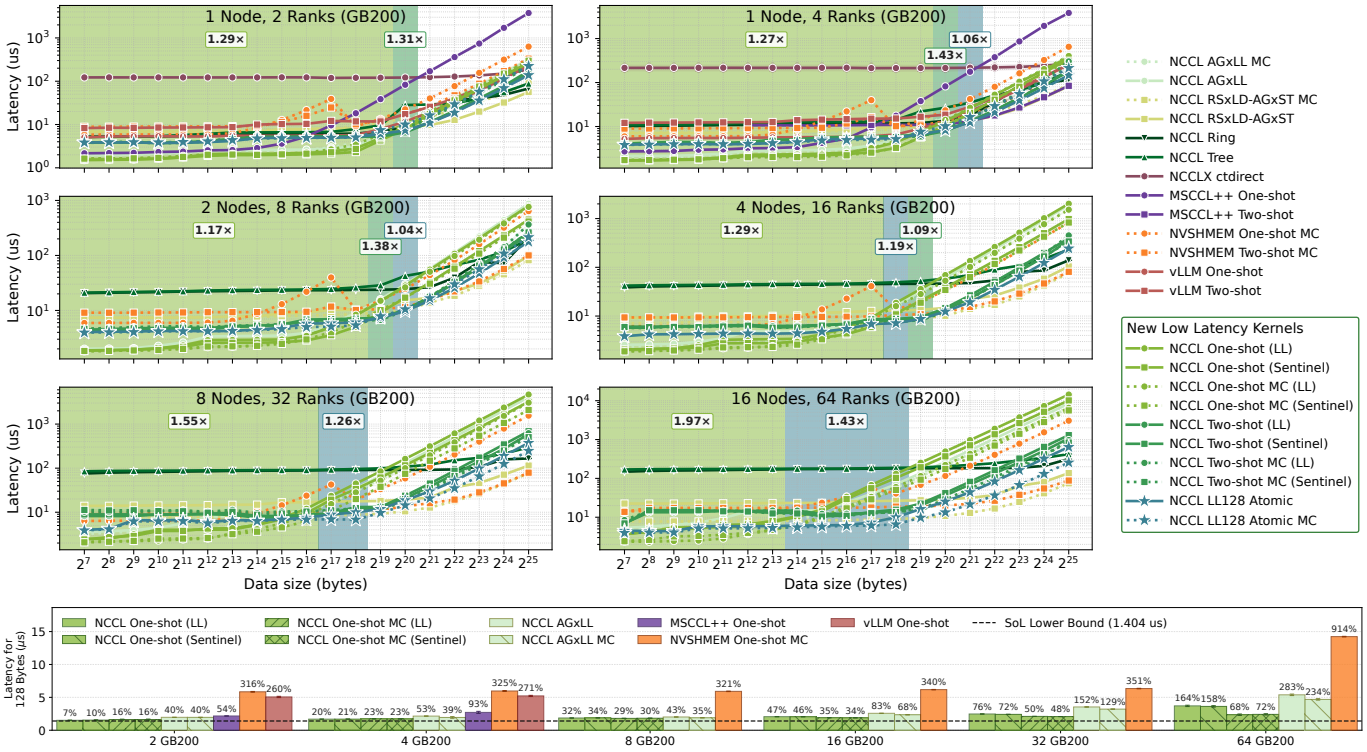

Fig. 11: Top plot shows out-of-place AllReduce latency versus message size on GB200 for 2 to 64 GPUs. Each subplot shows the mean over 10 trials for AllReduce implementations from NCCL, NCCLX, NVSHMEM, MSCCL++, and vLLM. Shaded regions mark message sizes where one of our kernels is fastest, regardless of the synchronization mode: light green for `LLBuffer` one-shot, dark green for `LLBuffer` two-shot, and blue-gray for LL128 atomic. Labels within these regions report the geometric-mean speedup over the fastest existing implementation at each message size in the region. Dashed lines indicate multicast variants. Bottom plot shows the latency at 128B for selected one-shot kernels across GPU counts. The dashed horizontal line marks the measured SoL lower bound. Percentages report the overhead of each kernel relative to this bound.

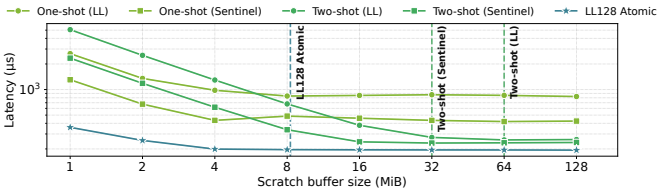

Fig. 12: Impact of scratch buffer size on AllReduce latency for our `LLBuffer`-based kernels on 2 nodes with 8 GB200. Vertical dashed lines indicate the minimum buffer size required for each kernel to process the full message in a single iteration.

B. Impact of Scratch Buffer

Before benchmarking AllReduce latency, we first study the impact of scratch buffer capacity to determine practical default sizes for our kernels. Fig. 12 shows results for `LLBuffer`-based kernels on 8 GPUs with a 32 MiB message. All kernels use 64 CTAs with up to 512 threads per CTA.

When the buffer is small ($<$ 8 MiB), two-shot kernels are slower than one-shot kernels. Each iteration of the two-shot design requires two synchronization steps, which dominate latency. Although fewer iterations are needed overall, the additional synchronization cost outweighs this benefit. As the buffer size increases, the performance of two-shot kernels improves rapidly and then plateaus once the entire message fits within a single iteration. This indicates that, ideally, achieving the best performance for two-shot kernels requires a scratch buffer large enough to process the data in one iteration.

In one-shot kernels, every rank writes to all peers simultaneously, which means that increasing the buffer size increases the amount of data sent per iteration, which raises instantaneous bandwidth pressure on the NVLink fabric. Hence, beyond a certain point, increasing the scratch size provides little benefit and can even slightly degrade performance.

The LL128 atomic kernel is particularly attractive because it is both space efficient and fast. Instead of storing intermediate data from all ranks, it accumulates contributions directly into the destination buffer, significantly reducing the scratch space needed to process the entire message.

Based on these results, we select 4 MiB of scratch buffer for one-shot kernels, as they show limited benefit from larger buffers. For two-shot and LL128 atomic kernels, we choose 64 MiB as the default.

C. Latency

With the scratch buffer size fixed, we perform a comprehensive evaluation across various data sizes and GPU counts. We compare our new low-latency kernels against the implementations from the state-of-the-art libraries and frameworks, including NCCL's legacy ring, tree algorithms, and symmetric memory kernels (v2.29.1), NVSHMEM (v3.5.21), MSCCL++ (v0.8.0), and vLLM custom AllReduce (v0.15.1). For all NCCL kernels, we used up to 64 CTAs with 512 threads per CTA. For other libraries, we used their default configurations.

Results are shown in Fig. 11. In the bottom plot, we zoom in on the case where the message size is a single cache line, and compare the latency of different one-shot AllReduce implementations to the SoL bound. We only show the data for 32-bit floats, as the latency for 16-bit floats (i.e., float16 and bfloat16) is very similar.

For NCCL, the `AGxLL` and `RSxLD-AGxST` kernels correspond to its latest one-shot and two-shot symmetric AllReduce algorithms, respectively. For NCCLX CTran, we use the `ctdirect` algorithm, which supports only single-node execution, so results are reported for 2 and 4 GPUs. The ring variant (`ctring`) is omitted, as it supports only one rank per node and is not optimized for low latency. The same single-node limitation applies to vLLM's custom AllReduce. Although MSCCL++ provides one-shot and two-shot AllReduce algorithms, including multicast variants, the multicast implementations consistently hang on GB200 and are excluded. The non-multicast variants are evaluated only on 2 and 4 GPUs, since multi-node collectives are not supported at the time of writing. A hierarchical AllReduce does exist for 2 nodes, but only with 8 GPUs per node. The two-shot algorithm also does not support 2 ranks or message sizes below 4 KiB, resulting in missing data points in the figure.

Based on the results, we make the following observations:

a) Observation 1: Our `LLBuffer`-based one-shot AllReduce achieves the lowest latency for small messages across all GPU counts. As shown in Fig. 11, our kernels incur only about 7% overhead over the SoL bound at 2 GPUs, while competing implementations remain noticeably farther away. Even at 64 GPUs, multicast one-shot variants stay within roughly 70% overhead of the SoL bound. While NCCL AGxLL and MSCCL++ also use LL-style synchronization, our implementation benefits from targeted optimizations including aggressive compile-time unrolling and parallel polling and reduction across ranks, which reduce detection and accumulation latency. We also observe a crossover between LL and sentinel synchronization. LL performs slightly better at very small sizes, whereas sentinel becomes preferable as message size and rank count increase, avoiding the flag overhead of LL.

b) Observation 2: When the number of GPUs is small, the advantage of the LL128 atomic kernel over the standard two-shot design is limited. L2 atomic operations incur slightly higher latency due to serialization compared to accumulating values in the scratch buffer. As the number of GPUs increases, however, the scalability of atomic operations becomes more apparent, allowing the LL128 kernel to outperform two-shot kernels over a wider range of small and medium message sizes.

c) Observation 3: At small scale, hardware multicast may incur slight overhead compared to unicast, but it plays a key role in improving collective scalability. The kernels that outperform our `LLBuffer`-based designs are primarily multicast variants that leverage the `multimem.ld_reduce` instruction to combine contributions across ranks. This greatly reduces both the number of explicit memory operations and the amount of software-managed reduction. The benefit becomes more pronounced at larger scales, where offloading reduction to the NVLink/NVSwitch fabric is particularly effective [43].

From the microbenchmarks, we observe that the proposed `LLBuffer`-based kernels cannot fully replace the existing symmetric kernel, namely the two-shot `RSxLD-AGxST`, as their polling overhead increases with GPU count and message size. However, they significantly reduce latency for small to medium messages, approaching the hardware limit, thereby complementing existing kernels.

VIII. Case Studies

After extensively benchmarking the kernels, we modify NCCL to select the best-performing low-latency algorithm based on the collected empirical results. For example, at 4 ranks, messages below 1 MiB use one-shot kernels, while those between 1 and 2 MiB use two-shot kernels. In this section, we evaluate the impact of these selections on two target workloads, LLM inference and cuSOLVERMp.

A. LLM Inference

For LLM inference, we conduct experiments on the same NVL72 GB200 system described earlier, using the same NVIDIA vLLM container from Section VII. We select vLLM as the inference framework due to its wide adoption and active development [3], [44], [45]. Experiments are conducted on several popular open-weight LLMs under two configurations: 1 node with 4 GPUs (TP=4) and 2 nodes with 8 GPUs (TP=8).

For all models, we use long input contexts of 100–200k tokens and generate 16K output tokens, with a batch size of 8 to emulate long-context inference scenarios. This choice reflects the growing importance of long-context workloads in practice [46]–[49]. Each setting is evaluated over 5 trials with vLLM's serving benchmark. We present the results in Fig. 13.

We evaluate several configurations against the NCCL baseline using only legacy kernels. *NCCL LL* employs the proposed `LLBuffer`-based one-shot kernels, while *No MC* disables NVLS multicast. *Sym Mem* enables PyTorch symmetric memory, registering input and output buffers as symmetric memory to unlock NCCL's two-shot and LL128 atomic kernels. Without it, AllReduce is limited to one-shot LL kernels and falls back to legacy implementations beyond the LL threshold. We report vLLM custom AllReduce and MSCCL++ only for the single-node case, since both are restricted to single-node execution as mentioned previously.

Across all models, low-latency kernels consistently improve performance. The best configuration reduces ITL by 7–13%

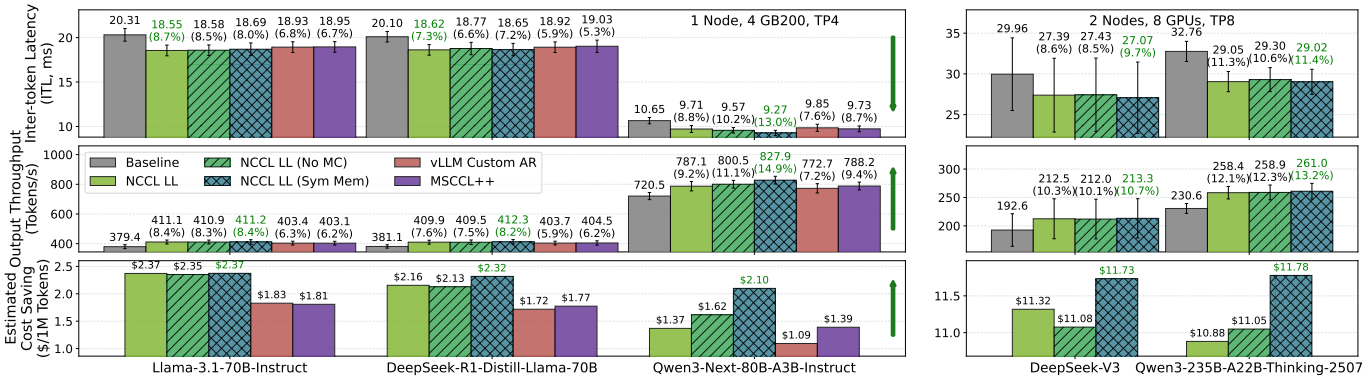

Fig. 13: Effect of low-latency collectives on vLLM inference. Rows report mean inter-token latency (ITL), output throughput, and estimated cost savings per 1M output tokens relative to the baseline. Percentages show the improvement over the baseline. Green labels highlight the best-performing configuration for each model. Green arrows indicate the direction of improvement.

on 4 GPUs and 9–11% on 8 GPUs, with similar gains in throughput. Results hold across diverse model architectures, including dense (Llama), mixture-of-experts (DeepSeek), and hybrid attention (Qwen3-Next). *NCCL LL* alone performs well because decode steps involve small AllReduce operations, where one-shot `LLBuffer` kernels approach the SoL bound. *Sym Mem* provides additional gains by enabling more efficient two-shot kernels for larger messages, particularly in vLLM's mixed prefill/decode execution where some operations exceed the LL threshold. The benefit is more pronounced in throughput than ITL, as throughput measures total generated tokens over wall-clock time and thus captures improvements across the entire generation process, including prefill.

We estimate cost savings by converting output throughput into dollars per 1M tokens using CoreWeave's GB200 pricing [8]. For hourly price p and output throughput r, the estimated cost is $p \cdot 10^6/(3600r)$. We recognize that production serving is often disaggregated [50], [51], with separate node pools for prefill and decode, so this is not a full deployment cost model. Rather, it is an estimate of the savings attributable to faster collectives. Under this estimate, gains exceed $11 per 1M output tokens in the 8-GPU setting for large models such as DeepSeek V3. In the 4-GPU case, although per-token savings are smaller, they accumulate into meaningful cost reductions for decode-heavy workloads at production scale.

B. Traditional HPC

To evaluate a representative traditional HPC workload, we use cuSOLVERMp. Many production GPU-accelerated applications, including GROMACS and LAMMPS, still rely on MPI or CUDA-aware MPI rather than NCCL [17], [52]–[55]. Although QMCPACK [56] exposes NCCL support, it is limited to the Auxiliary-Field Quantum Monte Carlo (AFQMC) method and is not well maintained [57], so we exclude it. NVIDIA HPCG, the High Performance Conjugate Gradients benchmark adapted to use NCCL, is a potential alternative but is also excluded as it relies exclusively on point-to-point communication [58]. We therefore use cuSOLVERMp, NVIDIA's distributed dense linear algebra library [59], as a practical case study. Its generalized symmetric-definite eigensolvers are widely used in electronic-structure workloads [60].

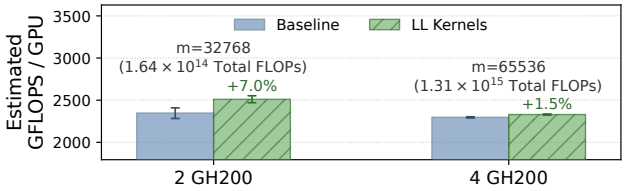

Fig. 14: Performance of `mp_sygvd` with and without the new low-latency kernels. Bars show mean GFLOPS per GPU over 5 trials, with error bars indicating standard deviation. Percentage annotations show the improvement over the baseline.

The experiments were conducted on the Alps supercomputing cluster. Each Alps node is equipped with four NVIDIA Grace Hopper Superchips (GH200) connected via 150 GB/s NVLink for intra-node communication [10], [61]. As Alps does not provide an NVSwitch-based scale-up fabric across multiple nodes, we restrict this study to a single node. We chose this platform over GB200 as it reflects a more accessible system for domain scientists. Experiments were performed in NVIDIA's PyTorch container (v25.10) running Ubuntu 24.04, CUDA 12.6, OpenMPI 4.1.7, and cuSOLVERMp 0.7.2. In our setup, larger matrix sizes (m=32768 and m=65536) exceed single-GPU memory capacity and are therefore executed in distributed mode. We exclude MSCCL++ from this comparison due to MPI errors and compare only against the NCCL baseline using legacy kernels.

Figure 14 shows that the proposed low-latency kernels consistently improve cuSOLVERMp across the two configurations. The gains are more pronounced for m=32768, where communication constitutes a larger fraction of runtime. Note that cuSOLVERMp does not register buffers as symmetric memory, so only one-shot kernels were used for message sizes below 1 MiB. Overall, these results indicate that low-latency collectives benefit not only LLM inference but also traditional HPC workloads as NCCL-based communication becomes more widely adopted.

IX. Related Work and Discussion

We are aware that several frameworks and communication libraries also provide low-latency AllReduce implementations, such as SGLang [5] and FlashInfer [62]. These are not included in our microbenchmark comparison, as their designs largely resemble the custom AllReduce in vLLM, and therefore exhibit similar performance characteristics. In TensorRT-LLM [6], AllReduce variants based on sentinel-style synchronization are available, but they still rely on global barrier flags for coordination, which introduces additional overhead compared to our designs.

There also exist libraries such as DeepEP [63] and NCCL EP [64] that provide low-latency primitives tailored for expert parallel workloads. These optimizations target a narrower class of communication patterns and do not generalize to other workloads. In contrast, our API is general-purpose and can be used to implement similar functionality when needed.

We do not compare against NIXL [65], as it targets a different design space. NIXL is primarily a transport and orchestration layer for GPU communication, focusing on scheduling and integration across heterogeneous backends rather than optimizing the latency of collectives.

There are several directions for future work. First, kernel selection is currently driven by empirical measurements and could be improved with an accurate performance model. Developing such a model would require detailed knowledge of GPU architecture, including factors such as warp scheduling and instruction-level behavior, which is beyond the scope of this work. Second, the current API design focuses on thread-level primitives to maximize performance. Future extensions could provide warp- or block-level abstractions with fewer constraints and improve usability.

X. Conclusion

In this work, we studied how to approach the absolute hardware lower bound for scale-up GPU collectives within a single NVLink domain and identified key principles for low-latency design. Building on NCCL's device-side APIs, we introduced a new set of APIs that simplify the construction of custom low-latency collective kernels. Using these APIs, we implemented several new AllReduce kernels that substantially reduce latency for small and medium messages, bringing SoL overhead down to about 7% in the best case and consistently outperforming state-of-the-art frameworks. In vLLM inference, the best configuration reduces ITL by up to 13% on 4 GPUs and 11% on 8 GPUs across diverse LLMs, with similar throughput gains and estimated savings of more than $11 per million output tokens for a large model such as DeepSeek-V3. We also observe consistent single-node speedups for cuSOLVERMp on the Alps supercomputer. Overall, these results demonstrate that low-latency optimization delivers measurable benefits in both AI inference and traditional HPC applications, and that the proposed APIs provide a practical foundation for building latency-critical collective kernels.

XI. Acknowledgments

The authors thank Andrei Ivanov for his invaluable assistance in collecting the experimental data. This work would not have been possible without his support. The research was conducted as part of the FastTrackAI project at the Singapore-ETH Centre, which was established collaboratively between ETH Zurich and the National Research Foundation, Singapore. This research is supported by the National Research Foundation, Singapore (NRF), and the Ministry of Digital Development and Information (MDDI) under the AI Visiting Professorship (Award No. AIVP-2025-005). This work also received funding from the European Research Council (Project PSAP, No. 101002047). The authors used ChatGPT-5.4 [66] to assist with light editing and proofreading. All content and ideas remain the original work of the authors.

References

[1] DeepSeek-AI, A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan, D. Dai, D. Guo, D. Yang, D. Chen, D. Ji, E. Li, F. Lin, F. Dai, F. Luo, G. Hao, G. Chen, G. Li, H. Zhang, H. Bao, H. Xu, H. Wang, H. Zhang, H. Ding, H. Xin, H. Gao, H. Li, H. Qu, J. L. Cai, J. Liang, J. Guo, J. Ni, J. Li, J. Wang, J. Chen, J. Chen, J. Yuan, J. Qiu, J. Li, J. Song, K. Dong, K. Hu, K. Gao, K. Guan, K. Huang, K. Yu, L. Wang, L. Zhang, L. Xu, L. Xia, L. Zhao, L. Wang, L. Zhang, M. Li, M. Wang, M. Zhang, M. Zhang, M. Tang, M. Li, N. Tian, P. Huang, P. Wang, P. Zhang, Q. Wang, Q. Zhu, Q. Chen, Q. Du, R. J. Chen, R. L. Jin, R. Ge, R. Zhang, R. Pan, R. Wang, R. Xu, R. Zhang, R. Chen, S. S. Li, S. Lu, S. Zhou, S. Chen, S. Wu, S. Ye, S. Ye, S. Ma, S. Wang, S. Zhou, S. Yu, S. Zhou, S. Pan, T. Wang, T. Yun, T. Pei, T. Sun, W. L. Xiao, W. Zeng, W. Zhao, W. An, W. Liu, W. Liang, W. Gao, W. Yu, W. Zhang, X. Q. Li, X. Jin, X. Wang, X. Bi, X. Liu, X. Wang, X. Shen, X. Chen, X. Zhang, X. Chen, X. Nie, X. Sun, X. Wang, X. Cheng, X. Liu, X. Xie, X. Liu, X. Yu, X. Song, X. Shan, X. Zhou, X. Yang, X. Li, X. Su, X. Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Y. Zhang, Y. Xu, Y. Xu, Y. Huang, Y. Li, Y. Zhao, Y. Sun, Y. Li, Y. Wang, Y. Yu, Y. Zheng, Y. Zhang, Y. Shi, Y. Xiong, Y. He, Y. Tang, Y. Piao, Y. Wang, Y. Tan, Y. Ma, Y. Liu, Y. Guo, Y. Wu, Y. Ou, Y. Zhu, Y. Wang, Y. Gong, Y. Zou, Y. He, Y. Zha, Y. Xiong, Y. Ma, Y. Yan, Y. Luo, Y. You, Y. Liu, Y. Zhou, Z. F. Wu, Z. Z. Ren, Z. Ren, Z. Sha, Z. Fu, Z. Xu, Z. Huang, Z. Zhang, Z. Xie, Z. Zhang, Z. Hao, Z. Gou, Z. Ma, Z. Yan, Z. Shao, Z. Xu, Z. Wu, Z. Zhang, Z. Li, Z. Gu, Z. Zhu, Z. Liu, Z. Li, Z. Xie, Z. Song, Z. Gao, and Z. Pan, "Deepseek-v3 technical report," 2025.

[2] RiseUnion, "Deepseek-v3/r1 671b deployment guide: Gpu requirements," 2025. Reports deployments requiring up to 32 accelerators for full-precision inference.

[3] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[4] M. Aubakirova, A. Atallah, C. Clark, J. Summerville, and A. Midha, "State of ai: An empirical 100 trillion token study with openrouter," 2026.

[5] L. Zheng, L. Yin, Z. Xie, C. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez, C. Barrett, and Y. Sheng, "Sglang: efficient execution of structured language model programs," in *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, (Red Hook, NY, USA), Curran Associates Inc., 2024. https://dl.acm.org/doi/10.5555/3737916.3739916.

[6] N. Corporation, "Tensorrt-llm: A library for optimizing large language model inference," 2023. Accessed: 2024-05-20.

[7] S. Shen, L. Huang, M. Chrapek, T. Schneider, J. Dayal, M. Gajbe, R. Wisniewski, and T. Hoefler, "Llamp: Assessing network latency tolerance of hpc applications with linear programming," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '24, IEEE Press, 2024. https://doi.org/10.1109/SC41406.2024.00070.

[8] CoreWeave, "CoreWeave Pricing: Instance Pricing," 2026. Accessed: March 2026.

[9] F. Xu, "Scaling-up pytorch inference: Serving billions of daily nlp inferences with onnx runtime," 2022. Microsoft Open Source Blog, accessed: March 2026.

[10] L. Fusco, M. Khalilov, M. Chrapek, G. Chukkapalli, T. Schulthess, and T. Hoefler, "Understanding data movement in tightly coupled heterogeneous systems: A case study with the grace hopper superchip," 2024.

[11] Z. Hu, S. Shen, T. Bonato, S. Jeaugey, C. Alexander, E. Spada, J. Dinan, J. Hammond, and T. Hoefler, " Demystifying NCCL: An In-Depth Analysis of GPU Communication Protocols and Algorithms ," in *2025 IEEE Symposium on High-Performance Interconnects (HOTI)*, (Los Alamitos, CA, USA), pp. 48–59, IEEE Computer Society, Aug. 2025. https://doi.ieeecomputersociety.org/10.1109/HOTI66940.2025.00024.

[12] N. Corporation, "NVSHMEM communication library," 2022. Accessed on 2024-05-20.

[13] Advanced Micro Devices, Inc., "ROCm Communication Collectives Library (RCCL) Documentation," 2024. Accessed: February 2026.

[14] Advanced Micro Devices, Inc., *rocSHMEM 3.2.0 Documentation*. AMD ROCm Documentation, 2026. Accessed: 2026-02-12.

[15] UXL Foundation, *oneAPI Specification 1.3-rev-1*, 2024. Accessed: February 2026.

[16] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard Version 5.0*, June 2025.

[17] J. Kraus, "An introduction to cuda-aware mpi," *NVIDIA Technical Blog*, July 2025.

[18] D. De Sensi, L. Pichetti, F. Vella, T. De Matteis, Z. Ren, L. Fusco, M. Turisini, D. Cesarini, K. Lust, A. Trivedi, D. Roweth, F. Spiga, S. Di Girolamo, and T. Hoefler, "Exploring gpu-to-gpu communication: Insights into supercomputer interconnects," in *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 1–15, IEEE, Nov. 2024. http://dx.doi.org/10.1109/SC41406.2024.00039.

[19] J. Bachan, K. Ouyang, M. Mubarak, T. Gillis, B. Chang, D. Bureddy, G. Congiu, K. Caton, K. Aubrey, and X. Li, "Enabling fast inference and resilient training with nccl 2.27," Jul 2025. Technical Blog.

[20] K. Hamidouche, J. Bachan, P. Markthub, P.-J. Gootzen, E. Agostini, S. Jeaugey, A. Shafi, G. Theodorakis, and M. G. Venkata, "Gpu-initiated networking for nccl," 2025.

[21] NVIDIA Corporation, *Device-Initiated Communication — NCCL 2.29.1 Documentation*. NVIDIA, 2025. Accessed: 2026-02-12.

[22] S. Jeaugey, J. Bachan, P. Markthub, Z. He, S. Das, and F. Ghodsian, "Fusing communication and compute with new device api and copy engine collectives in nvidia nccl 2.28," Nov. 2025. NVIDIA Technical Blog.

[23] PyTorch Contributors, *Distributed communication package – torch.distributed*. PyTorch Foundation, 2026. PyTorch 2.10 documentation, last updated 2026-01-08, accessed 2026-02-12.

[24] Erlangen National High Performance Computing Center (NHR@FAU), *PyTorch – NHR@FAU HPC Documentation*. NHR@FAU, 2026. Accessed 2026-02-12.

[25] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Z. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," *CoRR*, vol. abs/1912.01703, 2019.

[26] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[27] NVIDIA Corporation, *cuSOLVER API Reference*. NVIDIA Corporation, 2024. Version 13.1, accessed 2026-02-12.

[28] NVIDIA Corporation, *cuBLASMp: A High-Performance CUDA Library for Distributed Dense Linear Algebra*. NVIDIA Corporation, 2026. Accessed 2026-02-12.

[29] P. Singhania, S. Singh, L. D. Hough, A. Srivastava, H. Menon, C. F. Jekel, and A. Bhatele, "Llm inference beyond a single node: From bottlenecks to mitigations with fast all-reduce communication," 2025.

[30] C. Hwang, P. Cheng, R. Dathathri, A. Jangda, S. Maleki, M. Musuvathi, O. Saarikivi, A. Shah, Z. Yang, B. Li, C. Rocha, Q. Zhou, M. Ghazimirsaeed, S. Anantharamu, and J. Jose, "Msccl++: Rethinking gpu communication abstractions for ai inference," in *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '26, (New York, NY, USA), p. 1201–1215, Association for Computing Machinery, 2026.

[31] D. Patel, M. Xie, D. Nishball, I. Chiam, P. Zhou, Doug, and W. Chu, "Nvidia gtc 2025 - built for reasoning, vera rubin, kyber, cpo, dynamo inference, jensen math, feynman," Mar. 2025.

[32] xAI, "Colossus: xai's supercomputer for grok," 2024. Describes a large-scale GPU cluster with tightly coupled high-bandwidth interconnect.

[33] J. Tang, L. Robison, M. Koop, and W. Wang, "Recent improvement to open mpi allreduce and the impact to application performance," Sept. 2024.

[34] D. Xiong, L. Chen, Y. Jiang, D. Li, S. Wang, and S. Wang, "Revisiting the time cost model of allreduce," 2024.

[35] A. Weingram, Y. Li, H. Qi, D. Ng, L. Dai, and X. Lu, "xccl: A survey of industry-led collective communication libraries for deep learning," *J. Comput. Sci. Technol.*, vol. 38, p. 166–195, Mar. 2023. https://doi.org/10.1007/s11390-023-2894-6.

[36] M. Chrapek, M. Khalilov, and T. Hoefler, "Hear: Homomorphically encrypted allreduce," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, (New York, NY, USA), Association for Computing Machinery, 2023. https://doi.org/10.1145/3581784.3607099.

[37] D. E. Bernholdt, S. Boehm, G. Bosilca, M. Gorentla Venkata, R. E. Grant, T. Naughton, H. P. Pritchard, M. Schulz, and G. R. Vallee, "A survey of mpi usage in the us exascale computing project," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 3, p. e4851, 2020. https://doi.org/10.1002/cpe.4851.

[38] I. Laguna, R. Marshall, K. Mohror, M. Ruefenacht, A. Skjellum, and N. Sultana, "A large-scale study of mpi usage in open-source hpc applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, (New York, NY, USA), Association for Computing Machinery, 2019. https://doi.org/10.1145/3295500.3356176.

[39] S.-M. Hammer, S. Schmid, R. Singh, and V. Addanki, "Short-circuiting rings for low-latency allreduce," 2025.

[40] NVIDIA Corporation, "Cuda c++ programming guide." https://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2024. CUDA Toolkit Documentation.

[41] NVIDIA Corporation, "Parallel thread execution isa version 8.0." https://docs.nvidia.com/cuda/parallel-thread-execution/, 2023. NVIDIA CUDA PTX Instruction Set Architecture.

[42] NVIDIA Corporation, "vllm container (version 26.02-py3)," 2026. Accessed: 2026-03-29.

[43] M. Khalilov, S. D. Girolamo, M. Chrapek, R. Nudelman, G. Bloch, and T. Hoefler, "Network-offloaded bandwidth-optimal broadcast and allgather for distributed ai," in *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–17, 2024. https://dl.acm.org/doi/10.1109/SC41406.2024.00109.

[44] S. Kolluru, "Comparative analysis of large language model inference serving systems: A performance study of vllm and huggingface tgi," 2025.

[45] NeoSignal, "vllm – technology radar entry," 2025. Accessed: 2026-03-29.

[46] J.-S. Denain and A. Ho, "The huge potential implications of long-context inference," 2025. Accessed: 2026-03-29.

[47] Y. Chung, G. T. Kakkar, Y. Gan, B. Milne, and F. Özcan, "Is long context all you need? leveraging llm's extended context for nl2sql," *Proceedings of the VLDB Endowment*, vol. 18, p. 2735–2747, Apr. 2025. http://dx.doi.org/10.14778/3742728.3742761.

[48] Y. Zhang, R. Sun, Y. Chen, T. Pfister, R. Zhang, and S. Arik, "Chain of agents: Large language models collaborating on long-context tasks," 2024.

[49] H. Sun, L. Li, M. Xiao, and C. Xu, "Breaking the boundaries of long-context llm inference: Adaptive kv management on a single commodity gpu," 2025.

[50] BentoML Team, "Prefill-decode disaggregation." https://bentoml.com/llm/inference-optimization/prefill-decode-disaggregation, 2025. Explains motivation and benefits of PD disaggregation.

[51] L. Li, D. Li, B. Gong, and Y. Zhang, "Slo-aware compute resource allocation for prefill-decode disaggregated llm inference," 2026.
[52] GROMACS Development Team, *GROMACS 2024.3 Installation Guide*, 2024. Accessed: March 2026.
[53] LAMMPS Development Team, *LAMMPS Documentation: GPU Package*, 2026. Accessed: March 2026.
[54] A. Myers, W. Zhang, A. Almgren, T. Antoun, J. Bell, A. Huebl, and A. Sinn, "Amrex and pyamrex: Looking beyond the exascale computing project," *The International Journal of High Performance Computing Applications*, vol. 38, no. 6, pp. 599–611, 2024. https://doi.org/10.1177/10943420241271017.
[55] A. Huebl, "AMReX issue #4821: Device-initiated collectives in NCCL/RCCL." https://github.com/AMReX-Codes/amrex/issues/4821, 2025. GitHub issue, accessed January 2026.
[56] J. Kim, A. D. Baczewski, T. D. Beaudet, A. Benali, M. C. Bennett, M. A. Berrill, N. S. Blunt, E. J. L. Borda, M. Casula, D. M. Ceperley, S. Chiesa, B. K. Clark, R. C. Clay, K. T. Delaney, M. Dewing, K. P. Esler, H. Hao, O. Heinonen, P. R. C. Kent, J. T. Krogel, I. Kylänpää, Y. W. Li, M. G. Lopez, Y. Luo, F. D. Malone, R. M. Martin, A. Mathuriya, J. McMinis, C. A. Melton, L. Mitas, M. A. Morales, E. Neuscamman, W. D. Parker, S. D. Pineda Flores, N. A. Romero, B. M. Rubenstein, J. A. R. Shea, H. Shin, L. Shulenburger, A. F. Tillack, J. P. Townsend, N. M. Tubman, B. Van Der Goetz, J. E. Vincent, D. C. Yang, Y. Yang, S. Zhang, and L. Zhao, "Qmcpack: an open source ab initio quantum monte carlo package for the electronic structure of atoms, molecules and solids," *Journal of Physics: Condensed Matter*, vol. 30, p. 195901, apr 2018. https://doi.org/10.1088/1361-648X/aab9c3.
[57] QMCPACK Developers, "Qmcpack issue #4654," 2023. GitHub issue in the QMCPACK repository.
[58] NVIDIA, "NVIDIA HPCG Benchmark," 2024. Accessed: 2026-04-08.
[59] NVIDIA, *cuSOLVERMp: A High-Performance CUDA Library for Distributed Dense Linear Algebra*, 2026. Accessed: March 2026.
[60] A. Marek, V. Blum, R. Johanni, V. Havu, B. Lang, T. Auckenthaler, A. Heinecke, H.-J. Bungartz, and H. Lederer, "The elpa library: scalable parallel eigenvalue solutions for electronic structure theory and computational science," *Journal of Physics: Condensed Matter*, vol. 26, p. 213201, may 2014. https://doi.org/10.1088/0953-8984/26/21/213201.
[61] CSCS, "New research infrastructure: 'alps' supercomputer inaugurated," *Swiss National Supercomputing Center*.
[62] Z. Ye, L. Chen, R. Lai, W. Lin, Y. Zhang, S. Wang, T. Chen, B. Kasikci, V. Grover, A. Krishnamurthy, and L. Ceze, "Flashinfer: Efficient and customizable attention engine for llm inference serving," 2025.
[63] C. Zhao, S. Zhou, L. Zhang, C. Deng, Z. Xu, Y. Liu, K. Yu, J. Li, and L. Zhao, "Deepep: an efficient expert-parallel communication library," 2025.
[64] A. Goldman, N. Boker, M. Sheraizin, N. Admoni, A. Polyakov, S. Bhattacharya, F. Yu, K. Sun, G. Theodorakis, H.-C. Yin, P.-J. Gootzen, A. Shafi, A. Ravid, S. D. Girolamo, M. G. Venkata, and G. Bloch, "Nccl ep: Towards a unified expert parallel communication api for nccl," 2026.
[65] A. Ranadive, T. Stamler, S. Lee, and M. Khazraee, "Enhancing distributed inference performance with the nvidia inference transfer library," Mar. 2026. NVIDIA Technical Blog.
[66] OpenAI, "Gpt-5.4 chatbot," 2026.