NetMoE: Orchestrating Fine-Grained Overlap for Efficient Distributed MoE Inference

Zibo Wang[1*], Yadong Liu[2*], Zhehao Lin[1], Peisheng Guo[2], Zhirui Zhu[1], Zuxi Chen[1], Qianyu Jiang[1], Tiancheng Weng[1], Xingyi Li[2], Xinxin Liu[1], Xiaojie Huang[2], Yinben Xia[2], Yuxi Wang[3], Mingzhuo Chen[2], Xuan Zhang[2], Weizhen Dang[2], Hao Lu[2], Yuanyuan Gong[2], Xiang Li[2], ZeKun He[2], Yachen Wang[2], Xianneng Zou[2], Yiran Zhang[3], Rong Gu[1], Qingkai Meng[1], Peirui Cao[1], Zhaochen Zhang[1], Qing Wang[1], Zhibin Wang[1†], Sheng Zhong[1], Chen Tian[1]

[1] *State Key Laboratory for Novel Software Technology, Nanjing University* [2] *Tencent* [3] *Unaffiliated*

Abstract

Mixture-of-Experts (MoE) architectures have emerged as a dominant paradigm for scaling large language models (LLMs). However, meeting strict service level objectives (SLOs) during MoE inference is challenging because generating each token typically triggers substantial all-to-all communication across the GPU cluster. Overlapping communication with computation is a promising approach to reduce MoE inference latency. Yet we find that, even though existing overlap designs have been pushed towards extreme optimization and are theoretically efficient, mapping them onto real GPU and network hardware still exposes multilevel inefficiencies spanning computation padding, comp-comm handoff latency, and coarse-grained communication injection.

In this paper, we propose NetMoE, an optimization framework that orchestrates computation, comp-comm handoff, and communication injection under a shared granularity-alignment principle for fine-grained overlap in MoE inference. NetMoE integrates three mechanisms: (1) hardware-fitted transposed general matrix multiplication (GEMM), which achieves low-padding computation via near-zero overhead transposition; (2) lightweight comp-comm handoff, which trades available network-bandwidth headroom for lower preparation latency through quantization bypass and direct buffer aliasing; and (3) high-concurrency communication injection, which improves per-SM work queue element (WQE) injection efficiency so that communication preparation remains within the computation window. We integrate NetMoE into the high-performance SGLang framework. Evaluations on DeepSeek-V3 and Qwen3 demonstrate that NetMoE effectively translates overlap potential into real-world speedups, reducing end-to-end latency to 0.79× ∼ 0.84× (avg. 0.81×) relative to the sequential baseline. Furthermore, microbenchmarks confirm the efficacy of our components: the hardware-fitted GEMM achieves matrix transposition with negligible overhead, while our lightweight stack reduces communication signaling duration to 0.19× ∼ 0.43× of native DeepEP.

CCS Concepts: • **Computer systems organization** → **Cloud computing**; • **Networks** → **Cloud computing**; • **Computing methodologies** → **Machine learning**.

Keywords: Decode; GEMM; Communication; Overlap

ACM Reference Format:
Zibo Wang, Yadong Liu, Zhehao Lin, Peisheng Guo, Zhirui Zhu, Zuxi Chen, Qianyu Jiang, Tiancheng Weng, Xingyi Li, Xinxin Liu, Xiaojie Huang, Yinben Xia, Yuxi Wang, Mingzhuo Chen, Xuan Zhang, Weizhen Dang, Hao Lu, Yuanyuan Gong, Xiang Li, ZeKun He, Yachen Wang, Xianneng Zou, Yiran Zhang, Rong Gu, Qingkai Meng, Peirui Cao, Zhaochen Zhang, Qing Wang, Zhibin Wang, Sheng Zhong, Chen Tian. 2027. NetMoE: Orchestrating Fine-Grained Overlap for Efficient Distributed MoE Inference. In *22nd European Conference on Computer Systems (EuroSys '27), April 19–23, 2027, Rabat, Morocco.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3842654.3848575

1 Introduction

In recent years, MoE architectures have emerged as a dominant paradigm for scaling LLMs. By activating only a sparse subset of parameters for each token, MoE enables models to expand their capacity to trillions of parameters while maintaining manageable computational costs. This paradigm has been widely adopted across the industry, driving the development of foundational models ranging from early pioneers like GShard [25] and Switch Transformer [12], to state-of-the-art (SOTA) powerhouses such as Mistral's Mixtral 8x7B [21], Kimi's K2 [50], StepFun's Step-3 [47], DeepSeek-V3 [6]/R1 [9] and Qwen3 [53].

However, deploying these massive models for online serving presents a significant system challenge rooted in the MoE architecture itself. To accommodate the massive parameter size, experts are typically distributed across multiple GPU servers (*i.e.*, expert parallelism, or EP) [24, 25, 45]. Consequently, the inference process necessitates dynamic token

*Zibo Wang and Yadong Liu contributed equally.
†Zhibin Wang is the corresponding author.

EuroSys '27, Rabat, Morocco

ACM ISBN 979-8-4007-2971-3/2027/04
https://doi.org/10.1145/3842654.3848575

routing, which triggers frequent inter-server `All-to-All` communication to dispatch tokens to their target experts. Without dedicated optimization, this `All-to-All` communication executes serially with expert computation and is fully exposed on the critical path. Our profiling shows that it accounts for over 40% of decode-phase time per output token (TPOT), severely constraining the system's ability to meet stringent SLOs. To mitigate this latency overhead, overlapping communication with computation has become a common strategy: by executing communication in parallel with expert computation, systems aim to hide network latency and remove it from the execution critical path.

Recent overlap designs have become increasingly fine-grained to mitigate latency penalties, operating on smaller batches and dispatching communication on partial results before the full batch completes its computation—often at microsecond-scale intervals. As analyzed in §2.2, this trend is reflected in three concrete design choices: single-batch overlap (SBO) [2, 4, 51, 52] to enable intra-batch signal-based overlap, GPU-initiated communication [3, 14, 35] to bypass CPU-induced jitter and scheduling overheads, and vertical tightening [47] to restrict cluster scale and maintain small batch sizes, thereby avoiding latency penalties induced by the severe load imbalance inherent in massive EP [24, 25, 45].

While these fine-grained paradigms are theoretically efficient, we find that the end-to-end latency remains high in practice. The paradigms themselves have the potential to hide communication latency effectively. However, the underlying primitives on which fine-grained overlap execution depends—GEMM kernels, data preparation routines, and WQE injection mechanisms—were originally designed for throughput-oriented workloads. When applied to fine-grained overlap, where each computation wave lasts only tens of microseconds, these primitives introduce latency overheads that can no longer be amortized. A GEMM wave denotes a set of tiles scheduled concurrently on the available streaming multiprocessors (SMs); it is distinct from a 32-thread warp. Specifically, the MoE overlap execution can be decomposed into three stages—computation, comp-comm handoff, and communication injection—each of which suffers from such a mismatch:

(1) Compute Inefficiency: Modern tensor cores achieve high throughput by processing fixed-size matrix tiles, imposing strict alignment constraints on GEMM dimensions. On Hopper, the warp group matrix multiply-accumulate (WGMMA) instructions require the M dimension (i.e., the token count) to be a multiple of 64 [32]. In fine-grained MoE decode, however, the number of tokens routed to each expert per decode iteration is typically far smaller (e.g., 8–32), forcing heavy zero-padding and wasting a substantial portion of compute cycles on invalid calculations.

(2) Comp-Comm Handoff Inefficiency: Between computation and All-to-All communication lies a handoff stage in which the system prepares data for transmission—quantizing activations and copying results into RDMA send buffers. We find that this handoff incurs substantial latency: in SBO scenarios, the preparation phase is more than 2× longer than the concurrent computation, leaving communication latency exposed on the critical path rather than hidden behind it.

(3) Communication Inefficiency: When employing GPU-initiated communication, existing libraries rely on coarse-grained injection mechanisms—where an entire warp (the fundamental scheduling unit of 32 threads) is dedicated to committing a WQE to the NIC. This warp-level injection granularity limits the number of requests that each SM can inject concurrently. As a result, the limited concurrency of the warp-level control plane causes communication injection to extend beyond the computation window, exposing latency on the critical path and undermining the benefits of overlap.

To bridge this gap, we propose NetMoE, an optimization framework that orchestrates computation and communication primitives for fine-grained overlap in MoE inference. Our design is driven by the governing principle of granularity alignment—the insight that software primitives must be refactored from coarse-grained, throughput-oriented blocks into fine-grained, latency-centric units to match the hyper-fragmented workload. This refactoring is inherently joint: computation, comp-comm handoff, and communication injection compete for the same latency budget, and improving one squeezes another—a faster GEMM, for instance, leaves less time for the communication side. Reducing handoff overhead alone is therefore not enough: injection must be made comparably dense. NetMoE's contribution is this coordination: the three stages are orchestrated together. This paper makes the following contributions:

(1) Hardware-Fitted Transposed GEMM: To address the padding overheads imposed by the rigid alignment constraints of tensor cores, we propose a mathematically equivalent transposition strategy ($C^T = B^T \times A^T$). This approach maps the highly fragmented token dimension to the hardware's flexible N-dimension. Combined with a fused transposed epilogue, this design significantly reduces the padding overhead in small-batch scenarios without incurring extra memory movement costs.

(2) Lightweight Comp-Comm Handoff: To eliminate the exposed latency in the handoff stage, we reconstruct the data path between computation and communication. Adopting a bandwidth-for-latency trade-off, we bypass compute-intensive quantization and implement a zero-copy mechanism via buffer aliasing. This compresses the handoff phase down to pure signaling, so communication can launch as soon as computation completes and be hidden within the computation window.

(3) High-Concurrency Communication Injection: To address the coarse-grained injection granularity in GPU-initiated communication, we break the traditional warp-level

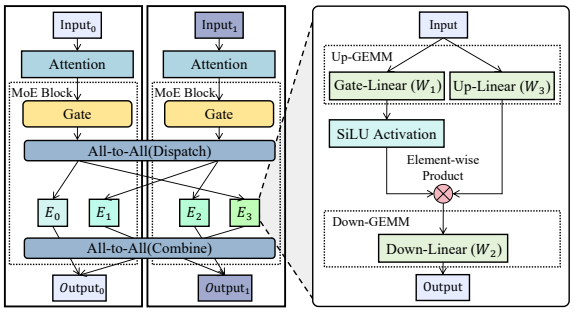


Figure 1. Transformer Layer in The MoE Architecture.

injection paradigm. NetMoE introduces a thread-level injection mechanism with a multi-queue, lock-free design, allowing each SM to issue more WQEs concurrently. This improves per-SM injection concurrency, drastically increases the control plane concurrency, and strictly compresses the injection latency within the computation window.

We implemented NetMoE within SGLang and evaluated it on modern NVIDIA GPU clusters. Our results demonstrate that NetMoE successfully resolves the critical bottlenecks of fragmented workloads, effectively translating the theoretical benefits of computation-communication overlap into practical performance gains. Specifically, for real-world DeepSeek-V3 and Qwen3 decode tasks, NetMoE achieves a robust reduction in end-to-end latency, reaching 0.79× ∼ 0.84× (avg. 0.81×) of the sequential baseline.

2 Background and Motivation

2.1 MoE Inference Workflow

Modern MoE-based LLMs replace the standard dense feed-forward network (FFN) with an MoE block. As shown in Figure 1, following self-attention, the hidden states enter the MoE block, which executes in four stages:

1. **Gating and Routing:** The gating network evaluates input tokens to select the top-k relevant experts and calculates their corresponding routing weights.
2. **Dispatch:** Tokens are redistributed to devices hosting their assigned experts via an `All-to-All` dispatch primitive for processing.
3. **Expert Computation:** Experts process tokens using FFNs parameterized by W_1, W_3, W_2. To improve hardware utilization, frameworks typically fuse W_1 and W_3 into a single *Up-GEMM*. After activation (e.g., SiLU), the result is projected back by W_2 via a *Down-GEMM*.
4. **Combine and Reduction:** Results return to source ranks via an `All-to-All` combine primitive. Finally, outputs from the top-k experts are weighted by routing scores and summed to generate the MoE layer output.

LLM inference operates in two phases: *Prefill* and *Decode*. The prefill phase processes the input prompt in parallel to populate the key-value (KV) cache, with performance measured by time to first token (TTFT). Subsequently, the decode phase generates tokens autoregressively—one token at a time—based on the preceding context, where performance is defined by TPOT.

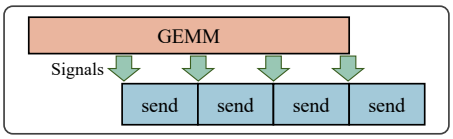


Figure 2. Signal-based Single Batch Overlap. Computation and communication execute on separate streams; in our combine path, an expert's outputs become sendable only after that expert's computation completes.

2.2 Design Trend under Tight SLOs

In real-time LLM serving, the fluidity of user interaction is governed by the TPOT of the decode phase. To guarantee responsiveness, production systems enforce stringent SLOs. Driven by these constraints, the system architecture for large-scale inference is converging towards specific design paradigms that prioritize latency over pure throughput. We analyze this trend across three critical dimensions: overlap strategy, communication control, and scaling pathways.

(1) Overlap Strategy: The Shift to Single-batch Overlap. An important idea to reduce end-to-end latency is to hide communication overhead behind computation through overlap. Existing techniques generally fall into two paradigms based on their overlap granularity:

- **Coarse-grained: Multi-batch Overlap (MBO)** [11, 22, 54]: MBO pipelines the execution of independent requests. By overlapping one micro-batch's communication with a subsequent batch's computation, MBO maximizes system throughput. However, this serialization inherently introduces queuing delays, making it ideal for offline tasks but suboptimal for latency-sensitive online inference.
- **Fine-grained: Single-batch Overlap (SBO)** [2, 4, 51, 52]: Conversely, SBO achieves interleaved execution by breaking data dependencies within a single inference iteration. As illustrated in Figure 2, it employs a signal-based synchronization mechanism. The workload is partitioned into smaller tiles; immediately upon completing the computation for a specific subset, the computation kernel emits a signal to trigger the corresponding `All-to-All` communication. This allows the network transmission of the processed data to proceed in parallel while the GPU continues computing the subsequent subsets.

Trend: SBO has emerged as the preferred paradigm for latency-critical scenarios, as it is the only strategy capable of reducing end-to-end latency for individual user requests.

(2) Communication Control: The Move to GPU-Initiated Communication. As overlap granularity shrinks to microsecond scale, the communication control plane must keep pace. Two paradigms exist based on where WQEs are constructed, with different control-plane latencies:

- **High-latency: CPU-Initiated Communication** [29, 56]: The GPU relies on a CPU proxy thread to orchestrate network operations, construct WQEs and ring the NIC doorbell. While robust, this approach incurs PCIe round-trip latency and OS scheduling jitter. In the microsecond-scale regime of SBO, the CPU proxy becomes a straggler, unable to match the GPU's high-frequency request generation.
- **Low-latency: GPU-Initiated Communication** [3, 14, 35]: Emerging technologies like InfiniBand GPUDirect Async (IBGDA) expose the NIC's control path directly to the GPU. This allows the GPU to issue WQEs and notify the NIC autonomously via memory-mapped IO (MMIO), effectively removing the CPU from the critical path.

Trend: Recognizing the bottleneck of CPU-initiated communication, high-performance inference systems are increasingly transitioning to GPU-initiated communication to eliminate the control plane overhead.

(3) Scaling Pathway: Consolidating for Efficiency. The physical distribution of the workload defines the theoretical boundaries of latency and stability. While theoretical analysis often favors massive scaling, practical deployment experiences reveal a different reality.

- **Skewed: Massive Horizontal Scaling**: Production systems like DeepSeek-V3 [6] adopt large-scale EP (e.g., EP 320 with 320 GPUs) to maximize throughput in decode phase. However, this approach suffers heavily from the well-known expert workload imbalance issue [26, 28]. While ad-hoc solutions like duplicating popular experts can mitigate stragglers, they incur significant memory overheads and lack the flexibility to adapt to dynamic shifts in data distribution, making performance unpredictable under varying traffic.
- **Balanced: Vertical Tightening**: Recent systems (e.g., Step-3 [47]) and industrial findings advocate for partitioning the cluster into small decode pods (e.g., 16∼32 GPUs) and constraining batch sizes to meet SLO targets and memory budgets, a strategy we term *vertical tightening*. This design balances per-card computational load by putting multiple experts onto each device, explicitly avoiding latency penalties induced by the severe load imbalance inherent in massive EP. Moreover, smaller decode pods enable more elastic scaling and narrower fault blast radii, improving both operational agility and system resilience.

Trend: Vertical tightening is increasingly favored in production to mitigate the instability of massive EP. Validated by our experience managing tens of thousands of GPUs, this strategy balances manageability with strict SLO compliance, circumventing the straggler traps of distributed systems.

2.3 Motivation: The Efficiency Gap in Fine-Grained Execution

While the architectural commitments in §2.2 (SBO, GPU-initiated communication, vertical tightening) provide the necessary foundation for low latency, they are largely insufficient to guarantee it in practice. These paradigms have the potential to hide communication latency effectively, but adopting them exposes a more insidious set of bottlenecks. The core conflict arises from a fundamental mismatch: the *hyper-fragmented, fine-grained* overlap workload is built atop software primitives—GEMM kernels, data preparation routines, and WQE injection mechanisms—that were designed for *coarse-grained, high-throughput* workloads. By enforcing SBO and vertical tightening, the workload is shattered into microsecond-scale kernels and messages. At this granularity, the overheads introduced by these primitives can no longer be amortized. Instead of vanishing into the background, they dominate the critical path, negating the theoretical gains of the overlap strategy. Specifically, the MoE overlap execution can be decomposed into three stages—computation, comp-comm handoff, and communication injection—each of which suffers from such a mismatch:

(1) Computation Inefficiency: Hardware Misalignment and Padding Penalty. A fundamental efficiency gap in distributed MoE inference arises from the growing divergence between the stochastic nature of sparse computation and the rigid execution requirements of high-performance hardware. Modern hardware accelerators, such as GPUs [31], NPUs [27] and TPUs [23], achieve massive throughput by employing dense, tiled execution engines (e.g., tensor cores). These units are architecturally optimized for large matrix dimensions and operate under the assumption of high data regularity. However, the MoE inference workflow inherently breaks this regularity. Particularly in the latency-critical decode phase, the combination of small global batch sizes and dynamic routing results in highly fragmented workloads. Each expert receives a variable and often minimal number of tokens, creating a mismatch where the compute demand fails to meet the hardware-level execution granularity. This forces the system to perform sub-optimal padding, leading to underutilized computational cycles and wasted arithmetic operations.

A Case Study: This inefficiency is particularly acute in the Hopper WGMMA execution path studied here, where a group of 128 threads asynchronously drives tensor cores. However, unlike per-thread instructions, WGMMA instructions process data in fixed-size blocks, which is known as single instruction multiple data (SIMD). This design imposes a strict alignment floor on the GEMM dimensions, specifically requiring the M dimension (i.e., the number of tokens) to be a multiple of 64. This hardware constraint creates a severe mismatch for MoE decode. Hopper also supports warp-level MMA; the issue is therefore a choice of instruction path, not the absence of smaller tiles. We explain the FP8 support and padding trade-offs in §4.1.2. In scenarios where an expert receives only a small batch of tokens (e.g., 8–32), standard libraries like CUTLASS [40] or DeepGemm [8] are forced to

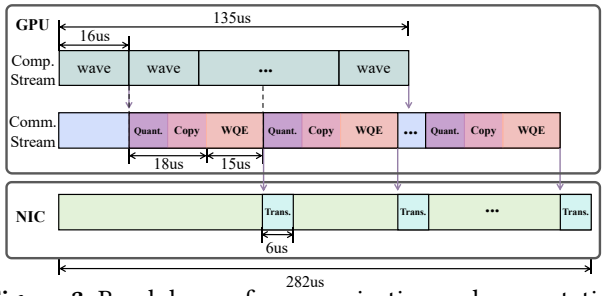


Figure 3. Breakdown of communication and computation latencies on a 16-GPU cluster.

pad the input to the 64-row boundary. For a typical workload of 16 tokens, this padding results in a 75% computation overhead, effectively wasting substantial hardware cycles on invalid, zero-padded calculations.

(2) Comp-Comm Handoff Inefficiency: Exposed Preparation Latency. The second bottleneck lies in the handoff stage between computation and communication, which becomes dominant in the latency-critical decode phase. Standard networking stacks typically enforce a separation between user-space computation memory and NIC-accessible pinned memory, necessitating intermediate copy steps. Furthermore, libraries often employ complex preprocessing like quantization to optimize bandwidth. While these techniques reduce physical network traffic, they turn the handoff into a compute-intensive stage. In the microsecond-scale regime of SBO, the execution time of these copy and quantization kernels often exceeds the computation window itself, leaving the handoff latency exposed on the critical path.

A Case Study: We profile this overhead on a 16-GPU cluster ($EP = 16$) processing a standard decode workload (16 tokens per expert) using an open-sourced SBO implementation [2]. In the GPU execution model, the GEMM workload is partitioned into tiles, and a *wave* refers to a set of these tiles that are executed concurrently and finish together on the available SMs. An expert may span multiple waves; its communication waits for all its output tiles. As illustrated in Figure 3, the computation window for a single GEMM wave is approximately 16 μs. However, the communication stream consumes 33 μs before actual network transmission begins—18 μs for the handoff stage (memory copying and quantization) and 15 μs for WQE injection. The handoff alone already exceeds the 16 μs computation window, and the subsequent injection cost widens the gap further, exposing a 17 μs bubble on the critical path. Consequently, the total execution time reaches 283 μs, far exceeding the pure computation time of 135 μs, negating the overlap benefit. Here we focus on the handoff cost (18 μs); the WQE injection bottleneck (15 μs) is analyzed later.

(3) Communication Inefficiency: Coarse-Grained Communication Injection. The third hurdle is the low concurrency of WQE injection in GPU-initiated communication. In fine-grained overlap, each computation wave triggers dozens of small network requests that must be injected within the microsecond-scale computation window. Existing communication stacks—ranging from throughput-oriented libraries like NCCL [39] and DeepEP [7] to lightweight, latency-optimized implementations such as NVSHMEM [36]/ IBGDA [35] and DeepEP's low-latency (LL) mode—all organize WQE injection at the warp level. Under this paradigm, an entire warp (32 threads) is dedicated to committing a single work request to the NIC. This coarse injection granularity limits the number of WQEs each SM can issue concurrently. When the required number of WQEs per wave exceeds the available warp-issue capacity, some requests must queue for later scheduling rounds, elongating the injection phase and exposing latency on the critical path.

A Case Study: We revisit the same decode setting in §2.3(2): EP=16, 16 tokens per expert on average, using the open-source SBO implementation [2]. In this EP configuration, each GPU hosts 16 experts, and each GEMM wave completes the computation of two experts. Since sending each token requires one WQE, each wave produces 32 token-level WQEs that must be injected promptly. The open-source SBO implementation reserves 3 SMs for communication. On NVIDIA GPUs, each SM has four warp schedulers; therefore, the three communication SMs provide only 12 warp-issue lanes for WQE-injection warps in a scheduling round. With warp-level IBGDA, issuing 32 WQEs therefore requires multiple scheduling rounds: some WQEs must wait until earlier WQE-injection warps make progress. This serialization elongates the communication injection phase and exposes it on the critical path, even though the total data volume is far from saturating the RDMA link. While allocating more SMs to communication might temporarily alleviate this symptom for small token counts, the same bottleneck inevitably reappears once the number of tokens slightly increases. Therefore, the root cause is not insufficient network bandwidth or compute resources, but the limited concurrency of the control plane caused by coarse warp-level WQE injection. A true solution must fundamentally increase the injection concurrency within the control plane.

3 Overview

NetMoE unleashes the potential of fine-grained overlap by reshaping computation and communication primitives to match its microsecond-scale execution granularity. Our design is driven by the governing principle of granularity alignment. We posit that the bottlenecks identified in §2.3—wasteful padding, exposed comp-comm handoff latency, and coarse-grained communication injection—are symptoms of a mismatch between fine-grained overlap and throughput-oriented system primitives. Guided by this principle, NetMoE orchestrates computation, comp-comm handoff, and communication injection so that all stages fit the microsecond-scale overlap window. As illustrated in Figure 4, this orchestration consists of three mechanisms:

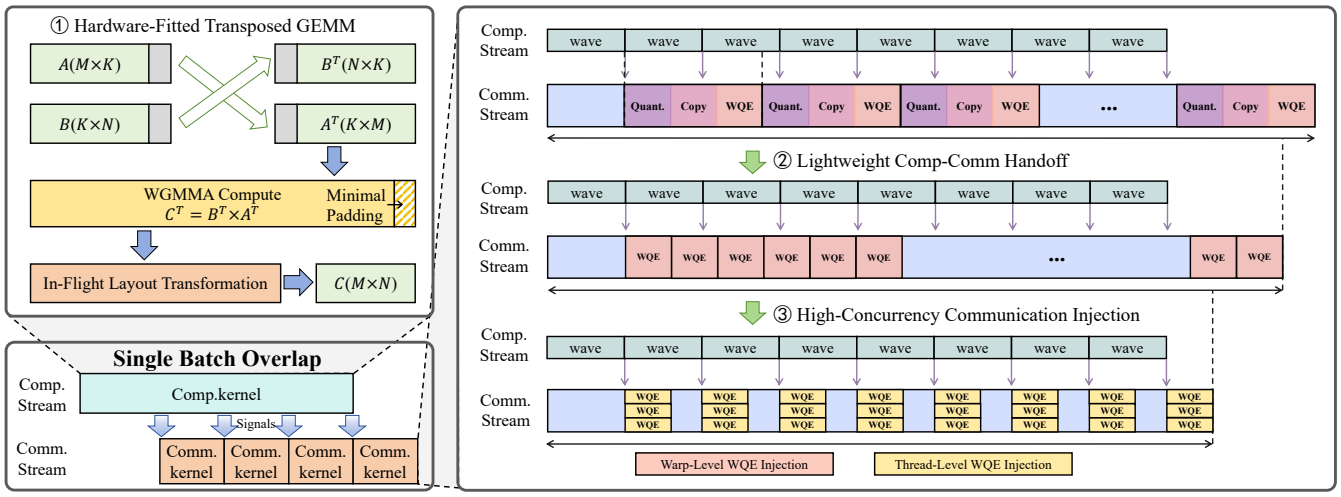


Figure 4. The Overall Architecture of NetMoE.

(1) Hardware-Fitted Transposed GEMM: Standard GEMM kernels suffer from significant waste due to rigid M-dimension alignment. NetMoE introduces hardware-fitted transposed GEMM (§4.1). By reformulating the computation to $C^T = B^T \times A^T$, we shift the fragmented token dimension to the hardware's flexible N-dimension. Crucially, we fuse the transposition logic directly into the data movement path, achieving this transformation with near-zero overhead. This design enables fine-grained alignment to drastically reduce padding waste without incurring extra latency.

(2) Lightweight Comp-Comm Handoff: To compress the handoff stage so that it fits within the computation window, NetMoE eliminates redundant processing overheads via a two-fold optimization. We adopt a bandwidth-for-latency trade-off, bypass the compute-intensive quantization, and simultaneously implement a zero-copy mechanism via buffer aliasing (§4.2). By stripping away both the quantization logic and the intermediate memory copying stages, this design reduces the handoff to signaling, enabling communication once the corresponding expert's output is complete.

(3) High-Concurrency Communication Injection: To address the coarse-grained injection granularity in GPU-initiated communication, NetMoE replaces the traditional warp-level paradigm with a thread-level, multi-queue, lock-free WQE injection mechanism (§4.3). This improves per-SM injection concurrency, keeping communication injection within the computation window while preserving most SM resources for computation.

4 System Design

4.1 Hardware-Fitted Transposed GEMM

To mitigate the compute inefficiency arising from the hardware misalignment described in §2.3, NetMoE introduces the hardware-fitted transposed GEMM. Unlike conventional methods that blindly pad the M dimension to meet rigid hardware constraints, our approach refactors the GEMM kernels to map the variable token workload onto the hardware's flexible N dimension, thereby minimizing the overall padding overhead.

Table 1. Supported Matrix Shapes for Hopper WGMMA Instructions.[32] The M dimension is rigidly fixed at 64, whereas the N dimension supports fine-grained granularity, providing the optimization opportunity exploited by NetMoE's transposition strategy.

Data Type	M Dim (Rigid)	N Dim (Flexible)	K Dim
`fp16` / `bf16`	64	8, 16, 24, . . . , 256	16
`tf32`	64	8, 16, 24, . . . , 256	8
`fp8` (`e4m3/e5m2`)	64	8, 16, 24, . . . , 256	32
`int8`	64	8, 16, 24, . . . , 256	32

4.1.1 Insight: Fitting Workload to N-dimension Flexibility.

As established in our motivation, NVIDIA Hopper WGMMA instructions impose strict alignment constraints on the M dimension, necessitating padding to 64 elements. However, as shown in Table 1, the hardware exhibits significantly higher flexibility for the N dimension, supporting fine-grained alignment steps of 8. Our key insight is to mathematically rotate the problem space by transforming the computation from $C = A \times B$ into $C^T = B^T \times A^T$. This transformation effectively maps the fragmented and unaligned M dimension of the original problem to the N dimension of the transposed system, enabling the kernel to perform computation with fine-grained alignment, reclaiming the majority of the computational cycles previously wasted on invalid padding calculations.

4.1.2 Design Choice: Warp-Level MMA versus WGMMA.

The MMA-versus-WGMMA choice follows from precision support. Warp-level `mma.sync` offers smaller tiles—the BF16 and FP8 `m16n8k*` forms use $M = 16$ and $N = 8$ [37]—but Hopper has no native FP8 warp-level MMA: the FP8 `m16n8k32` instruction is lowered by converting FP8 operands

to FP16, forfeiting native FP8 throughput [41]. Since FP8 expert computation is widely adopted in large-scale MoE inference, as in DeepSeek-V3 [6], we keep native FP8 WGMMA with DeepGEMM's TMA pipeline, reducing padding by mapping the fragmented token dimension to WGMMA's flexible N dimension.

4.1.3 Challenge: The Overhead of Explicit Transposition. Implementing this transposed strategy in a standard pipeline introduces a new bottleneck: transposition overhead. A naive implementation would typically follow a multi-stage chain: first explicitly transposing inputs A and B to produce A^T and B^T, executing the transposed GEMM, and finally transposing the result C^T back to C to ensure compatibility with downstream operations. This computation chain introduces three discrete transposition steps. As demonstrated in our evaluation (§6.2), this approach is untenable: the prohibitive overhead of explicit transposition causes the total latency to explode to $7.93 \sim 14.75\times$ of the vanilla GEMM implementation. Consequently, the entire chain must be collapsed into a single, integrated process where the transpositions are realized with zero overhead by repurposing the existing data movement pipeline.

4.1.4 Mechanism: Zero-Overhead Input & Output Transformation. NetMoE achieves the required transpositions without additional latency by deeply integrating the transformation logic into the kernel pipeline through two techniques:

- **Implicit Input Transposition via Input Swapping.** To maintain high throughput, compute kernels require input matrices to be stored in a K-major format for contiguous memory access. We exploit this architectural requirement by simply swapping the input pointers of A and B while reconfiguring their logical shapes at the kernel entry point. Since the WGMMA instructions read operands in a K-major fashion, this swap effectively forces the hardware to treat the memory layout of B as B^T and A as A^T, ensuring that the first two transpositions are completed virtually, without any physical data movement or instruction overhead.
- **In-Flight Layout Transformation via Coordinate Decoupling.** To minimize overhead, our key design principle is to treat the final $C^T \rightarrow C$ transposition not as a separate data-movement operator, but as an in-flight layout transformation along the GEMM epilogue data path. Since GEMM results naturally flow from registers to shared memory (SMem) and then to global memory during the epilogue, NetMoE embeds the layout transformation into this mandatory write-back path, avoiding any additional memory pass. Realizing this is non-trivial because the epilogue store path uses XOR-based swizzling to prevent SMem bank conflicts. Directly modifying the swizzle functions would tightly couple the transposition logic with architecture-specific bank-conflict handling, making the implementation fragile and hard to maintain. Instead, NetMoE adopts coordinate decoupling. It transforms the logical coordinates of each output element before feeding them into the standard swizzle functions, so the existing conflict-free memory access path is reused unchanged, while the output lands directly in the desired C layout in global memory. This realizes a zero-overhead transposed epilogue through in-flight layout transformation.

4.2 Lightweight Comp-Comm Handoff

Existing frameworks such as MSCCL++ and ParallelKittens already provide zero-copy communication primitives [19, 49]. Our design instead starts from the overlap budget: in the combine pipeline studied here, quantization delays the handoff more than compression helps transmission. Bypassing quantization removes the intervening operator and allows GEMM outputs to be written directly into registered RDMA send buffers. We formalize this bandwidth-for-latency choice below.

4.2.1 Insight: Formulating the Overlap Optimization. In a fine-grained overlap pipeline, each wave executes computation (T_{comp}) concurrently with the communication of the previous wave. The communication itself consists of GPU-side handoff (T_{handoff}), WQE injection (T_{inject}), and physical network transmission (T_{trans}). The per-wave execution time is dictated by the slowest pipeline stage:

$$\min\ T_{\text{wave}} = \max\big(T_{\text{comp}},\ T_{\text{handoff}} + T_{\text{inject}},\ T_{\text{trans}}\big)$$

For a fixed hardware architecture and model configuration, the computation time T_{comp} acts as a constant lower bound. Therefore, achieving the optimal solution (perfect overlap) requires compressing the communication-side terms such that:

$$T_{\text{handoff}} + T_{\text{inject}} \le T_{\text{comp}} \quad \text{and} \quad T_{\text{trans}} \le T_{\text{comp}} \tag{1}$$

However, standard libraries fail this condition. In our profiled setting (§2.3), we observe a severe imbalance: T_{handoff} alone (18 μs) exceeds T_{comp} (16 μs), whereas T_{trans} (6 μs) is far below it ($T_{\text{trans}} \ll T_{\text{comp}}$). This inequality reveals a critical opportunity: we can deliberately sacrifice the slack in T_{trans} to heavily compress T_{handoff}.

4.2.2 Mechanism: Pipeline Reconstruction via Dual-Optimization. Guided by this bandwidth-for-latency tradeoff, NetMoE introduces two mechanisms to eliminate the handoff overhead:

- **Strategic Quantization Bypass.** Standard MoE libraries typically perform on-the-fly quantization prior to transmission to compress data volume and minimize T_{trans}. In particular, they often employ complex algorithms like *LogFMT*—which involves logarithmic scaling and block-wise normalization—to preserve accuracy for outlier activations. While this is distinct from and much more compute-intensive than simple FP8 type casting, it is highly effective

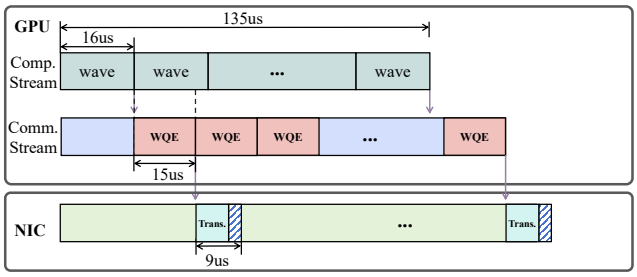


Figure 5. Latency breakdown after quantization bypass and buffer aliasing.

for throughput-oriented bulk transfers. Under our fine-grained overlap formulation, however, the priorities invert. The arithmetic execution of such complex quantization constitutes the majority of T_{handoff}. Because the payload per wave is exceedingly small, T_{trans} inherently falls far below T_{comp}, rendering any bandwidth savings redundant. By intentionally bypassing this compute-intensive preprocessing, NetMoE trades the unused bandwidth slack to eliminate the arithmetic latency, stripping the primary bottleneck from T_{handoff}.

- **Direct Buffer Aliasing.** After removing quantization, the remaining T_{handoff} stems from copying GEMM outputs from computation buffers to NIC-accessible pinned memory. In traditional stacks, this intermediate copy is unavoidable because a quantization kernel sits between the GEMM and the network, necessitating a separate memory hop. By bypassing quantization, NetMoE removes the only intermediate operator on the data path, creating the structural opportunity to implement a strict zero-copy pathway. We achieve this by pre-registering the RDMA send buffers at initialization and aliasing the GEMM output pointers directly to them. Consequently, GEMM results land immediately in NIC-readable memory, dropping copy latency to zero. This unified buffer design intrinsically streamlines system overheads: rather than allocating separate computation and communication tensors, the aliased buffer serves both roles, meaning the overall memory footprint actually decreases. Furthermore, this memory sharing requires no complex synchronization. Because each memory unit is written exactly once by the GEMM before the corresponding WQE is injected, there is no intra-GEMM buffer reuse. The buffer is only recycled between model layers, a process naturally governed by existing layer-level execution barriers, eliminating the need for any additional locking mechanisms or NIC-to-compute signaling.

4.2.3 Benefit: Achieving the Optimal Lower Bound.

Revisiting §2.3, Figure 5 demonstrates how NetMoE rebalances the pipeline to eliminate the handoff bottleneck:

- **Compressing T_{handoff}:** By bypassing quantization and eliminating intermediate copies, the handoff phase drops to near zero.
- **Controlled T_{trans} Inflation:** Bypassing quantization increases the physical transmission time from 6 μs to 9 μs, consistent with the 1.6× traffic increase from sending BF16 activations in place of LogFMT's 10/16 compression. Crucially, 9 μs remains strictly bounded by T_{comp} (16 μs), safely exploiting the bandwidth slack.

4.3 High-Concurrency Communication Injection

While the lightweight handoff eliminates data preparation overheads, the coarse-grained WQE injection still causes the communication injection latency to spill over the computation window as in Figure 5. To resolve this, NetMoE fundamentally redesigns the communication injection mechanism and introduces a thread-level WQE injection design that increases per-SM WQE injection density, enabling concurrent WQE emission.

4.3.1 Insight: Densifying Injection Granularity.

Our design is driven by a fundamental observation: the limited per-SM injection efficiency is not inherent to the hardware, but is an artifact of the software's injection granularity. The standard warp-level injection model used by SOTA libraries artificially limits the number of WQEs that each SM can issue concurrently. A GPU SM is a massive throughput engine capable of managing thousands of concurrent threads. By shifting the atomic unit of injection from the coarse-grained warp to the fine-grained thread, we can theoretically increase the WQE injection density by 32× compared with one-WQE-per-warp injection. This allows us to sustain the required injection rate with fewer scheduling rounds, maintaining high control plane concurrency and keeping communication injection strictly within the computation window.

4.3.2 Challenge: Hardware Constraints in High-Concurrency Injection.

Implementing thread-level injection faces a significant hardware barrier regarding how the GPU interacts with the NIC. Previous works typically utilize warp-level granularity to avoid contention within the NIC driver queues. The underlying issue is that NIC queues were originally designed for CPU-level concurrency rather than GPU-level concurrency. When hundreds of GPU threads attempt to submit WQEs simultaneously to a standard queue, they trigger two performance-degrading phenomena:

1. **Lock Contention:** To ensure data consistency, the driver logic must serialize access to the shared queue state, causing threads to stall while waiting for locks.
2. **Pointer Update Serialization:** The atomic updates of the producer index (PI) within the queue become a bottleneck under high-frequency access.

To avoid this contention, standard libraries aggregate requests at the warp level, effectively reducing concurrency.

Therefore, to successfully transition from warp-level to thread-level injection, we must fundamentally resolve these two contention issues, enabling massive parallelism without triggering the queue-level synchronization penalties of the driver.

4.3.3 Mechanism: Conflict-Free Thread-Level Injection. NetMoE addresses these hardware constraints by decoupling message injection from the warp structure. We refactor the RDMA issuance logic to allow each thread to independently manage requests. A queue pair (QP) is the RDMA endpoint containing the work queues used to submit operations. This involves two techniques:

- **Thread-Independent WQE Construction.** We remove intra-warp synchronization instructions (like `__shfl_sync` and `__syncwarp`) from the execution path. In our design, threads do not need to coordinate with peers to construct requests. Instead, each thread autonomously: i) calculates its target slot address in the per-QP WQE ring buffer, ii) populates the necessary fields (opcode, address, length), and iii) triggers the doorbell register. To avoid write conflicts, each thread is statically assigned a disjoint WQE slot based on its thread ID and queue mapping; therefore, the target address can be computed arithmetically without locks or atomics. This converts the control flow from a synchronized operation into parallel execution streams.
- **Multi-Queue Mapping Strategy.** To mitigate contention on shared queue states, we abandon the single queue per warp paradigm in favor of a multi-queue mapping approach. Instead of funneling all traffic through a centralized bottleneck, we distribute the injection workload across multiple parallel QPs. By partitioning the concurrency domain (e.g., mapping distinct QPs to different thread groups), this design drastically reduces lock conflicts and pointer interference. Unlike conventional CPU-side multi-queue designs, which scale with the number of CPU cores, NetMoE maps GPU threads directly onto NIC queues and combines this mapping with thread-level WQE slot allocation. It effectively leverages the capabilities of modern commercial NICs, which support large numbers of active queues, enabling us to sustain high injection rates without the scalability limits of a serialized control path.

4.3.4 Benefit: Higher Per-SM Injection Efficiency. This transition from warp-level to thread-level execution increases the number of independent WQEs that each SM can inject within the same scheduling window.

- **Denser WQE Injection:** The effective injection capacity now scales with the number of active threads rather than the number of active warps. To issue 32 independent WQEs, traditional warp-level injection requires 32 full warps, whereas NetMoE can issue them using the 32 threads within a single warp.
- **Hidden Preparation Latency:** This directly addresses the case study in §2.3(3). In that setting, the baseline can issue only 12 WQEs in the first scheduling round with 3 communication SMs, leaving 20 out of 32 WQEs queued. By contrast, NetMoE's thread-level injection can issue all 32 WQEs without such warp-level serialization. By drastically increasing the control plane concurrency per SM, NetMoE effectively compresses the WQE issue time, ensuring the entire communication initiation phase fits strictly within the brief computation window.

5 Implementation

We prototype NetMoE by building upon an open-source SBO implementation [2]. In the combine path, the signal marks expert completion: an expert's outputs are released for transmission once all of its tiles are computed, even if its computation spans several waves, while other experts continue computing. In our implementation, we replace the standard GEMM with our efficient transposed GEMM. Furthermore, we integrate the fine-grained communication optimizations proposed in §4 into the combine phase to maximize overlap efficiency. Our primary implementation is developed within SGLang [57]. We are committed to open science and plan to release our full source code in the near future to facilitate community adoption.

Dispatch versus combine. The SBO implementation [2] overlaps the shared-expert computation with the dispatch receive phase, and the Down-GEMM with the combine send. The shared expert operates on the local input tokens and need not wait for routed tokens to arrive, so it provides useful work while the dispatch receive completes. Combine send, by contrast, depends on newly produced expert outputs, and this dependency is what exposes the handoff and injection costs that NetMoE targets. We therefore retain the baseline dispatch schedule and focus our kernel changes on combine.

In the combine phase, the core task is to route the computation result of each token back to its original source rank. The naive warp-level implementation would assign each warp to handle both local (NVLink) and remote (RDMA) destinations based on the token's origin. However, with our thread-level WQE injection mechanism, the conditional branching required to distinguish between transport protocols causes severe warp divergence. Since threads within a warp execute in lockstep, this divergence forces the hardware to serialize the execution of the conflicting paths. To resolve this, we adopt a warp specialization strategy.

Specialized RDMA Pipeline. We strictly segregate the execution resources. The threads allocated to the communication SMs are dedicated exclusively to RDMA initiation. By removing the logic for intra-node communication from these kernels, we circumvent the performance degradation induced by warp divergence. All threads in the communication group focus on constructing and injecting WQEs for remote destinations, ensuring that the instruction cache efficiency of our thread-level injection mechanism is fully preserved.

Receiver-Initiated NVLink Integration. For intra-node traffic (NVLink), we transition from a sender-initiated (push) model to a receiver-initiated (pull) model. This design is driven by the observation that NVLink communication performance is heavily bound by SMem resources [10, 14].

- **Scaling via Compute SMs:** Since NVLink transfers require substantial SMem for buffering, limiting this workload to a few communication SMs creates a bottleneck. By shifting the responsibility to the destination's compute SMs (which are performing the aggregation), we can distribute the NVLink workload across far more SMs.
- **Execution Flow:** Once the computation is finished, the aggregation kernels on the destination SMs directly access the peer memory via NVLink to pull the results. This effectively expands the aggregate SMem capacity available for communication buffering, allowing the transfer to complete significantly faster than if it were constrained by the limited resources of the communication SMs.

6 Evaluation

6.1 Setup

6.1.1 Testbed. We evaluate NetMoE on a cluster comprising two nodes, each equipped with 8 NVIDIA H20 GPUs (96 GB HBM3). The GPUs are interconnected via NVLink within nodes, while inter-node communication is supported by one dedicated 400 Gbps RDMA NIC per GPU. This environment represents a compute-constrained regime, characterized by high memory bandwidth (4.0 TB/s) relative to its restricted peak processing power. Since this configuration constitutes the majority of our internal inference clusters, meeting strict SLOs is notably challenging, underscoring the practical value of our work in this prevalent scenario.

6.1.2 Software Environment. We compile our kernels using CUDA 12.8 [38] and use PyTorch 2.10 [43] as the frontend interface. The baseline communication primitives are provided by DeepEP [7], while the matrix multiplication kernels are based on DeepGemm 2.2.0 [8]. For end-to-end integration and comparison, we implement NetMoE within the SGLang v0.5.6post2 [57] inference engine.

6.1.3 Benchmarks and Baselines. We organize our evaluation into three hierarchical levels: primitive micro-benchmarks, operator-level overlap performance, and end-to-end inference latency.

Micro-benchmarks. To validate the efficacy of our individual architectural innovations, we first benchmark the computation and communication primitives in isolation:

- **Computation**: We evaluate the efficiency of our hardware-fitted transposed GEMM by comparing it against two baselines: (1) the original `DeepGemm` [8] implementation, and (2) a naive transposition approach that wraps `DeepGemm` with explicit PyTorch-based transpose operations (pre- and post-computation). Additionally, we report the pure computation time of this naive baseline-excluding transposition overheads-as a proxy for the theoretical optimal execution time.
- **Communication Control Plane**: We evaluate the efficiency of our communication implementation under fragmented workloads. A GEMM kernel executes concurrently with the send kernel, and emits a signal once an expert's computation completes, releasing that expert's outputs for transmission. We measure the aggregate signaling duration, defined as the time interval from the initiation of the first send operation to the completion of the final send operation. We compare NetMoE against the native DeepEP [7] LL mode to quantify the reduction in preparation overhead when handling high-frequency communication triggers.

Operator-level Evaluation. We evaluate the combined execution latency of the GEMM computation and the `Combine All-to-All` communication phase. We compare NetMoE against three distinct baselines:

- **No-overlap:** The sequential execution baseline, where the all-to-all combine communication initiates only after the GEMM computation has fully completed.
- **Naive Signal-Based Overlap with DeepEP (NSBOD) [2]:** While state-of-the-art overlap systems like FlashOverlap [17] exist, their All-to-All implementations remain closed-source. Therefore, we adopt the SBO implementation as our primary baseline, which represents the current best practice for open-source MoE communication overlap. We extract the computation and communication operators from this implementation, termed NSBOD, and measure the total latency spanning from the start of the GEMM kernel to the completion of the combine receive operation.
- **Comet [54]:** We execute the official benchmark from the Comet repository[1] to measure the same total time interval (GEMM initiation to combine reception). Since Comet currently lacks inter-node support for this overlap pattern, we restrict this comparison to intra-node scenarios.

Ablation Configuration. To quantify the individual contributions of our architectural innovations, we further design an incremental ablation study. We utilize the NSBOD implementation as the starting point and sequentially integrate our proposed techniques: first enabling lightweight comp-comm handoff (+LCH), followed by high-concurrency communication injection (+HCCI), and finally incorporating the hardware-fitted transposed GEMM (+HFTG), which constitutes the complete NetMoE design.

End-to-End Evaluation. To demonstrate the practical impact on production-grade workloads, we integrate NetMoE into SGLang [57], a high-performance serving framework. We evaluate the end-to-end inference latency using the DeepSeek-V3 [6] and Qwen3-235B-A22B [53] on the ShareGPT dataset,

[1] https://github.com/bytedance/flux/blob/main/test/python/moe_gather_rs/test_moe_gather_rs.py

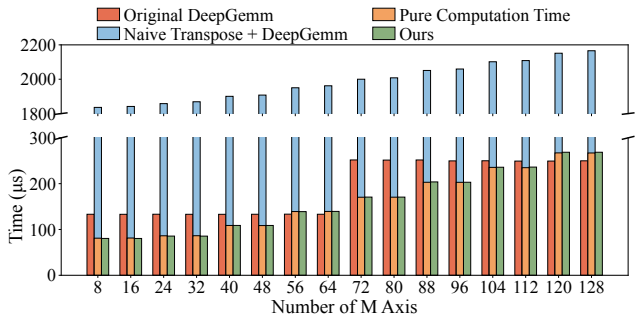


Figure 6. Performance comparison of GEMM kernels. The experiments are conducted using expert dimensions (**K = 7168, N = 2048**) derived from the DeepSeek-V3/R1 architecture, with the token count M varying from 8 to 128. The x-axis is the per-expert token count M (not the total batch size); "Pure Computation Time" measures only the GEMM in the naive transpose-based baseline, excluding the explicit transpose operations.

with input and output sequence lengths set to 4K and 1.5K tokens, respectively. Both models use top-8 routing, and the sequence lengths follow a public SGLang/Ant Group DeepSeek serving benchmark [55]. We discuss the generality of this setting in §7.2. We benchmark the NetMoE-enhanced SGLang against two configurations of the original SGLang:

1. **No-Overlap (Original SGLang):** The standard execution with overlap disabled, serving as the sequential baseline.
2. **NSBOD:** The native implementation enabling SBO.

Each experiment is repeated three times, and we report the average TPOT. This setup allows us to verify performance gains in real-world token generation phases against both the sequential baseline and the naive overlap implementation.

6.2 Computation Kernel Efficiency

We conducted experiments using expert dimensions derived from the DeepSeek-V3/R1 architecture. Since our approach refines the alignment granularity to $M = 8$, we swept the token count M with a stride of 8. As shown in Figure 6, the original `DeepGemm` baseline exhibits a distinct step-wise latency driven by its rigid 64-alignment: execution time remains constant across $M = 8$ to 64, and jumps to a higher plateau for $M = 72$ to 128.

For the Naive implementation, the pure computation time drops to 0.61∼1.07× of the original baseline, validating the theoretical gain of reduced padding. However, the prohibitive overhead of explicit transposition causes the total latency to explode to 7.93∼14.75×. In contrast, our hardware-fitted transposed GEMM achieves a total latency within 0.98∼1.01× of the naive pure computation. This confirms that our design effectively eliminates transposition costs, successfully translating theoretical compute savings into an average 17.26%, max 40% latency reduction, or equivalently 0.60∼1.07× execution time, compared to original `DeepGemm`. We observe

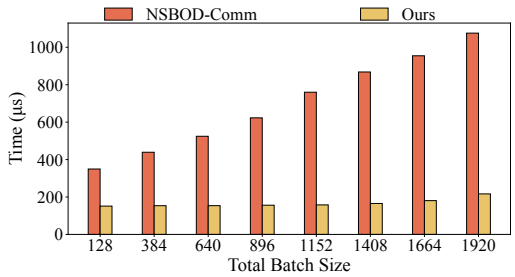


Figure 7. Benefit from Lightweight Comp-Comm Handoff.

that this advantage narrows as M approaches multiples of 64, with a slight regression at exact multiples. This is expected: as padding waste vanishes, the baseline benefits from coarser tiling granularity that yields higher peak utilization. The two layouts differ in tile granularity: the baseline tiles the original N dimension up to 256, whereas the transposed layout maps it to WGMMA's fixed 64-row M tile. Nevertheless, our approach provides essential flexibility for irregular shapes. Future iterations will incorporate a runtime heuristic to dynamically dispatch the original kernel when M aligns with the 64-boundary.

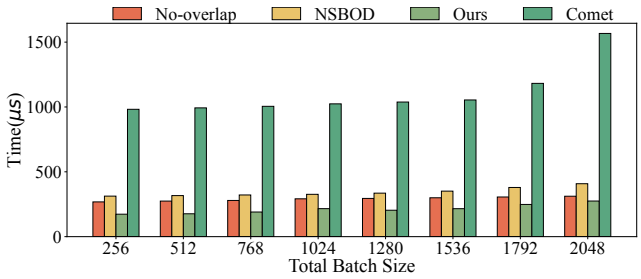
6.3 Benefits of Lightweight Comp-Comm Handoff

We develop a customized test harness to simulate the execution of a single MoE block iteration: the dispatch phase is pre-executed to generate the random token distribution across experts. Timing commences at the initiation of the first send operation and terminates upon the completion of the final send operation, thereby capturing the entire communication preparation and signaling overhead. We vary the total batch size (BS) to sweep through different workload intensities, with the average token count per expert calculated as

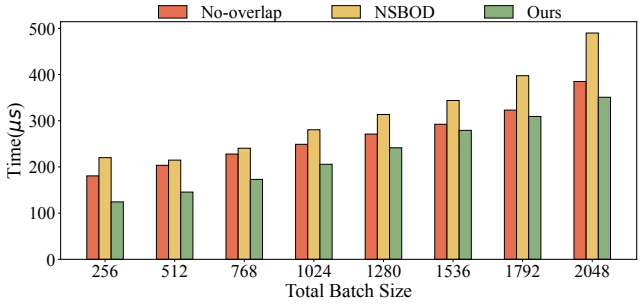
$$Avg\_Token\_Per\_Expert = \frac{BS \times Top\_K}{N_{expert}}. \tag{2}$$

Hyperparameters of the GEMM remain consistent across all runs: $K = 2048$, $N = 7168$, and $N_{expert} = 256$. Here BS is the total number of input tokens across the EP group in a decode step, and $Top\_K = 8$; for example, $BS = 256$ averages eight tokens per routed expert, though actual counts vary with routing. The compared paths use the same non-transposed GEMM, so the measured differences reflect handoff and injection behavior.

Figure 7 presents the aggregate send operation duration when restricting the communication resource to a single SM. The results demonstrate that the measured combine-send duration with NetMoE is 0.19× ∼ 0.43× that of native DeepEP. Corroborated by our profiling analysis, this efficiency stems from the synergistic effect of NetMoE's optimization stack: specifically the quantization bypass, zero-copy mechanism, and high-concurrency thread-level injection. These techniques effectively strip away heavy software overheads, enabling NetMoE to compress the communication preparation latency strictly within the computation window even with



(a) Experiment Result of Single-Node Configuration.



(b) Experiment Result of Dual-Node Configuration.

Figure 8. Combined Execution Latency of the GEMM (**K = 7168, N = 2048**) and Combine All-to-All Phase on single-node (8 GPUs) and dual-node (16 GPUs) configurations. The x-axis represents the Total Batch Size. The average number of tokens per expert is calculated as Equation (2).

minimal hardware resources (1 SM). In contrast, under such tight constraints, the native DeepEP stack fails to sustain the required WQE injection rate due to its limited control plane concurrency, causing a substantial portion of communication latency to be exposed on the critical path.

6.4 Operator-level Performance

We conduct experiments on both single-node (8 GPUs) and dual-node (16 GPUs). We follow the experiment method in §6.3, but timing commences at the initiation of the GEMM and terminates upon the completion of the final combine-receive operation. Note that both configurations use the same hyperparameter $N_{expert} = 256$; thus each GPU in the 8-GPU setup handles more experts and a larger per-card token volume, yielding slightly higher normalized latency.

Figure 8(a) presents the single-node performance. NetMoE maintains a decisive lead, reducing latency to 0.64× ∼ 0.88× (avg. 0.73×) of the No-Overlap baseline. Conversely, NSBOD regresses to 1.12× ∼ 1.31× (avg. 1.18×). Most notably, Comet exhibits a severe performance anomaly in this decode scenario, with latency surging to 3.51× ∼ 5.03× (avg. 3.79×). Our code inspection shows that Comet statically reserves 32 SMs for communication in its configuration. While this allocation can improve overlap for large-granularity workloads, in decode it leaves far fewer SMs available for expert GEMM execution, substantially stretching the computation phase and dominating the end-to-end latency.

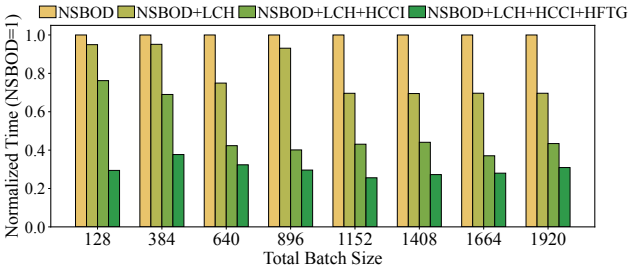


Figure 9. Result of Ablation Study.

Figure 8(b) illustrates the results for the dual-node scenario. Compared to the No-Overlap baseline, NetMoE significantly reduces execution time to 0.68× ∼ 0.95× (avg. 0.83×). In stark contrast, corroborating our analysis in §2.3, NSBOD fails to effectively hide communication overheads. Due to the heavy software stack overhead exposed on the critical path, NSBOD exhibits a performance regression, with latency increasing to 1.06× ∼ 1.27× (avg. 1.16×) of the sequential No-Overlap baseline.

6.5 Incremental Ablation Study

This ablation runs on two nodes with communication restricted to a single SM per GPU, following the constrained setup of §6.3. To mitigate the impact of random perturbations, we conduct 10 independent trials for each data point and report the average results. Furthermore, to isolate the performance contribution of each proposed technique, we design an incremental ablation path starting from the NSBOD baseline, sequentially enabling lightweight comp-comm handoff (LCH), high-concurrency communication injection (HCCI), and hardware-fitted transposed GEMM (HFTG).

Figure 9 presents the results of our ablation study. To clearly visualize the relative performance improvements, we normalize the execution latency of the NSBOD baseline to 1.

- **+ LCH:** Integrating lightweight comp-comm handoff effectively mitigates the preparation overhead, reducing the execution latency to 0.69× ∼ 0.95× (avg. 0.79×) of the NSBOD baseline.
- **+ HCCI:** Further incorporating high-concurrency communication injection improves per-SM WQE injection efficiency and reduces exposed injection latency, bringing the latency down to 0.37× ∼ 0.76× (avg. 0.49×).
- **+ HFTG (Full NetMoE):** Finally, the addition of hardware-fitted transposed GEMM completes the design. This achieves a total latency reduction to 0.25× ∼ 0.37× (avg. 0.30×) relative to NSBOD, explicitly validating the effectiveness of our orchestrated optimization stack.

The ablation ratios use one communication SM per GPU, whereas Figure 8(b) uses the standard SBO allocation of three communication SMs per GPU; this difference in communication resources contributes to the different relative gains.

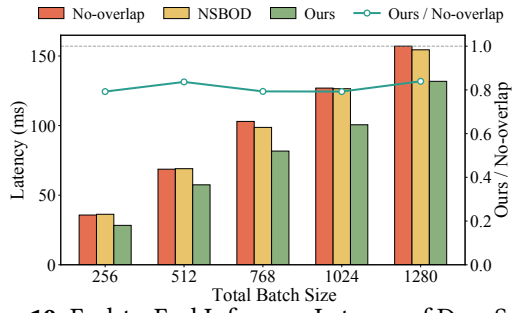


Figure 10. End-to-End Inference Latency of DeepSeek-V3 on SGLang.

6.6 End-to-End Performance

We evaluate the end-to-end latency using SGLang's official bench_serving tool[2]. We adopt a Prefill-Decode (PD) disaggregated deployment (1P1D), where both instances span two machines (16 GPUs). Focusing on the decoding stage, we vary the total batch size as the primary independent variable to capture the inherent stochasticity of the MoE gating mechanism. To ensure statistical stability and mitigate the impact of random perturbations, we conduct 3 independent trials for each data point and report the average results.

As illustrated in Figure 10, NetMoE demonstrates a robust performance advantage. Across the tested workload intensities, NetMoE consistently reduces the mean TPOT to $0.79\times \sim 0.84\times$ of the No-Overlap baseline, achieving an average latency reduction to $0.81\times$. In contrast, NSBOD yields negligible end-to-end gains: its mean TPOT remains largely on par with the No-Overlap baseline, ranging from $0.96\times$ to $1.02\times$. This confirms that without the orchestrated optimization stack, the heavy software overheads inherent in native overlap implementations (e.g., wasteful computation padding, exposed comp-comm handoff latency, and coarse-grained communication injection) consume most of the overlap benefit, leaving little to no speedup. By eliminating these bottlenecks, NetMoE successfully translates the theoretical potential of overlap into tangible end-to-end speedups. We also validate NetMoE on the Qwen3-235B-A22B model, where NetMoE achieves an average TPOT of $0.88\times$ relative to the No-Overlap baseline, confirming the generalizability of our approach across different MoE architectures.

7 Discussion

7.1 Hardware Dependencies and Generality

Our performance evidence comes from Hopper-based H20 GPUs. The three mechanisms have different dependencies. The transposed GEMM exploits Hopper's WGMMA shapes, K-major operands, TMA movement, and shared-memory epilogue layout. Other accelerators may benefit from mapping an irregular token dimension to a more flexible instruction dimension, but the opportunity depends on their native shapes, supported precisions, and memory layouts. The same principle extends to Blackwell: its `tcgen05.mma` still executes fixed-shape tiles with rigid alignment on the token dimension [34], so the padding inefficiency—and the benefit of remapping the fragmented token dimension to a more flexible one—persists on that architecture as well. Realizing it there, however, is not a drop-in port: `tcgen05.mma` and tensor memory differ from Hopper's WGMMA, TMA, and shared-memory epilogue, so the transposed kernel must be re-derived for the Blackwell backend. AMD GPUs and NPUs would similarly require backend-specific instruction and layout choices.

The handoff principle is less architecture-specific: avoid preprocessing when its latency exceeds the benefit from compression, and write directly into communication-accessible storage where the memory interface permits. Fusing FP8 conversion into the GEMM epilogue is an alternative that avoids a separate launch and copy, but the conversion still precedes the point at which outputs become sendable, so it can lengthen the critical path unless that work can be hidden; bypassing preprocessing removes it entirely. Its applicability depends on buffer registration, output-format compatibility, visibility ordering, and sufficient network headroom. A faster compute engine shortens the overlap window, while a slower or contended network lengthens transmission; either can change the favorable choice in Equation 1.

High-concurrency injection requires device-initiated queue access and correct WQE publication and completion ordering. The principle of distributing independent requests across threads and queues can transfer to backends exposing similar semantics, but the IBGDA/QP implementation and resource allocation must be adapted. Our two-node testbed already exercises multiple GPUs and NICs. Porting our DeepGEMM/DeepEP integration to another serving framework requires adapting buffer management, streams, and completion signals, even when the underlying kernels can be reused. Multi-tenant deployments additionally require queue/resource isolation and consideration of shared-network contention.

7.2 Workload Coverage and Deployment Scope

The end-to-end evaluation uses ShareGPT with 4K input and 1.5K output tokens, a setting with a public serving precedent [55]. Agentic, reasoning, and longer-context workloads are not covered: they can shift expert popularity, routing skew, and cross-node traffic, as well as KV-cache pressure, feasible batch size, and the attention-to-FFN time ratio. Our kernels do not assume a fixed routing pattern, but their latency gains depend on these workload properties. Microbenchmark sweeps cover a range of GEMM shapes and token counts, isolating primitive behavior.

[2]https://github.com/sgl-project/sglang/blob/v0.5.6.post2/python/sglang/bench_serving.py

Small decode pods with constrained batches are used in our production serving deployments. The SGLang/Ant Group report also uses EP16 on H20 and discusses the TPOT limitations of large-batch TBO [55]; Step-3 reports a 32-GPU decode instance [47]. These examples motivate our target regime. With 256 routed experts and top-8 routing, batch size at 128-2048 corresponds to averages of 4–64 tokens per expert, around which per-expert counts fluctuate under routing skew. The dispatch overlap discussed in §5 likewise depends on the availability of independent shared-expert work.

8 Related Work

Our work builds on distributed MoE, GPU networking, kernel optimization and overlap strategies. Several prior systems jointly optimize computation and communication. Our focus is the interaction of padding, handoff preparation, and WQE injection in small-batch distributed MoE decode.

MoE Systems and Scheduling. MoE systems focus on macroscopic scheduling and load balancing to mitigate expert skew. GShard [25] and Switch Transformers [12] drop tokens under static capacity limits to preserve regular shapes. For dynamic workloads, FastMoE [15] and FasterMoE [16] use dynamic shadowing, while Tutel [20] and DeepSpeed-MoE [45] apply adaptive parallelism and hierarchical All-to-All for global resource allocation. MegaBlocks [13] reduces memory fragmentation via block-sparse computation. These systems treat compute and communication as black boxes, overlooking micro-architectural overheads like kernel launch latency that dominate low-latency inference.

Kernel Optimization. GEMM optimization maximizes tensor core use for dense matrices. CUTLASS [40] and cuBLAS [33] optimize data hierarchies, while Stream-K [42] improves load balance for irregular grids. FlashAttention [46] uses IO-awareness to improve performance. Standard kernels remain rigid, requiring dimensions to align with hardware tiling (e.g., multiples of 64 for Hopper WGMMA). This forces padding for the small, jagged token counts in MoE. TMA-Adaptive FP8 Grouped GEMM [48] reduces padding in the TMA data-movement path by adapting descriptors for irregular M-dimension groups, but leaves WGMMA's compute-instruction padding unchanged. NetMoE is orthogonal: it removes the compute padding itself by moving the fragmented token dimension from M to N via transposed GEMM.

Computation-Communication Overlap. Overlap is fundamental to hiding latency. PipeDream [30] and GPipe [18] use pipeline parallelism with micro-batches, increasing latency. Recent works shift towards fine-grained overlap. Async-TP [44] explores asynchronous tensor parallelism but lacks MoE-specific routing optimizations. FlashOverlap [17] uses signaling to trigger tile-level communication. Works like UCCL-EP [29] optimize MoE communication but rely on CPU-initiated control planes, which introduces PCIe jitter unsuited for microsecond-scale SBO. Flux [5] and Comet [54] push fine-grained overlap: Flux fuses communication into kernels, while Comet reorders instructions via dependency analysis. Our work complements overlap scheduling by modifying the primitives on its critical path. NetMoE orchestrates primitives for fine-grained overlap, eliminating padding and networking overhead to enable efficient overlap even at single-token granularity.

Zero-copy communication frameworks. MSCCL++ [19] provides zero-copy, one-sided, asynchronous transfers, synchronization primitives, and a DSL that supports fusion. ParallelKittens [49] provides reusable multi-GPU kernel primitives and a scheduling template; direct transfers into pre-allocated destination buffers avoid intermediate staging. In the evaluated combine path, NetMoE bypasses LogFMT when uncompressed transmission still fits the computation window, enabling GEMM outputs to alias RDMA send buffers (§4.2). It coordinates this handoff choice with padding reduction and denser WQE injection.

Fused MoE execution. FlashMoE [1] fuses dispatch, expert computation, and combine into one persistent GPU kernel, reducing launch overhead and enabling device-side pipelining. NetMoE instead retains separately scheduled GEMM and communication kernels connected by completion signals. This preserves kernel modularity, while requiring explicit buffer and synchronization integration; fusion integrates scheduling more tightly but couples the compute and communication implementations [17]. These are complementary design points, and our layout and handoff ideas may also benefit fused implementations.

9 Conclusion

We introduce NetMoE, an optimization framework for orchestrating fine-grained overlap in distributed MoE inference. By integrating hardware-fitted transposed GEMM, lightweight comp-comm handoff, and high-concurrency communication injection, NetMoE eliminates critical-path overheads, achieving an average end-to-end latency of 0.81× relative to the sequential baseline, a scenario where naive strategies regress. Microbenchmarks confirm that our approach compresses injection duration to 0.19× ∼ 0.43×, establishing NetMoE as a robust solution for low-latency MoE serving.

Acknowledge

This work was supported by the Key Program of the National Natural Science Foundation of China under Grant Nos. 62325205, 62502198, 62472242, U25B2035, 62272215 and 62572225, the Natural Science Foundation of Jiangsu Province under Grant Nos. BK20243053 and BK20251224 and the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (Nos. JYB2025XDXM118 and JYB2025XDXM901), and the "111 Center"(No. B26023).

References

[1] Osayamen Jonathan Aimuyo, Byungsoo Oh, and Rachee Singh. FlashMoE: Fast distributed MoE in a single kernel, 2025. https://arxiv.org/abs/2506.04667.

[2] antgroup. Accessed: 2026-01. single batch overlap for moe models. https://github.com/sgl-project/sglang/pull/9660.

[3] Alex Brooks, Philip Marshall, David Ozog, Md. Wasi ur Rahman, Lawrence Stewart, and Rithwik Tom. Intel(r) shmem: Gpu-initiated openshmem using sycl, 2024.

[4] Weilin Cai, Juyong Jiang, Le Qin, Junwei Cui, Sunghun Kim, and Jiayi Huang. Shortcut-connected expert parallelism for accelerating mixture-of-experts. *ArXiv*, abs/2404.05019, 2024.

[5] Li-Wen Chang, Wenlei Bao, Qi Hou, et al. Flux: Fast software-based communication overlap on gpus through kernel fusion, 2024.

[6] DeepSeek-AI. Deepseek-v3 technical report, 2024.

[7] DeepSeek-AI. Deepep: an efficient expert-parallel communication library. 2025. https://github.com/deepseek-ai/DeepEP.

[8] DeepSeek-AI. Deepgemm: A high-performance fp8 gemm library for nvidia hopper. 2025. https://github.com/deepseek-ai/DeepGEMM.

[9] DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025.

[10] Mahesh Doijade, Andrey Alekseenko, Ania Brown, Alan Gray, and Szilárd Páll. Redesigning gromacs halo exchange: Improving strong scaling with gpu-initiated nvshmem. In *Proceedings of the SC '25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC Workshops '25, page 1314–1329, New York, NY, USA, 2025. Association for Computing Machinery.

[11] Zhiyuan Fang, Yuegui Huang, Zicong Hong, Yufeng Lyu, Wuhui Chen, Yue Yu, Fan Yu, and Zibin Zheng. Klotski: Efficient mixture-of-expert inference via expert-aware multi-batch pipeline. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '25, page 574–588, New York, NY, USA, 2025. Association for Computing Machinery.

[12] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.

[13] Trevor Gale, Deepak Narayanan, Cliff Young, and Matei Zaharia. MegaBlocks: Efficient Sparse Training with Mixture-of-Experts. *Proceedings of Machine Learning and Systems*, 5, 2023.

[14] Khaled Hamidouche, John Bachan, Pak Markthub, et al. Gpu-initiated networking for nccl, 2025.

[15] Jiaao He, Jiezhong Qiu, Aohan Zeng, Zhilin Yang, Jidong Zhai, and Jie Tang. Fastmoe: A fast mixture-of-expert training system, 2021.

[16] Jiaao He, Jidong Zhai, Tiago Antunes, Haojie Wang, Fuwen Luo, Shangfeng Shi, and Qin Li. Fastermoe: modeling and optimizing training of large-scale dynamic pre-trained models. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, page 120–134, New York, NY, USA, 2022. Association for Computing Machinery.

[17] Ke Hong, Xiuhong Li, Minxu Liu, Qiuli Mao, Tianqi Wu, Zixiao Huang, Lufang Chen, Zhong Wang, Yichong Zhang, Zhenhua Zhu, Guohao Dai, and Yu Wang. Efficient and adaptable overlapping for computation and communication via signaling and reordering. In *Proceedings of the 21st European Conference on Computer Systems*, EUROSYS '26, page 1894–1911, New York, NY, USA, 2026. Association for Computing Machinery.

[18] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[19] Changho Hwang, Peng Cheng, Roshan Dathathri, Abhinav Jangda, Saeed Maleki, Madan Musuvathi, Olli Saarikivi, Aashaka Shah, Ziyue Yang, Binyang Li, Caio Rocha, Qinghua Zhou, Mahdieh Ghazimirsaeed, Sreevatsa Anantharamu, and Jithin 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, page 1201–1215, New York, NY, USA, 2026. Association for Computing Machinery.

[20] Changho Hwang, Wei Cui, Yifan Xiong, Ziyue Yang, et al. Tutel: Adaptive mixture-of-experts at scale. *ArXiv*, abs/2206.03382, 2022.

[21] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, et al. Mixtral of experts. *ArXiv*, abs/2401.04088, 2024.

[22] Chao Jin, Ziheng Jiang, Zhihao Bai, Zheng Zhong, et al. Megascale-moe: Large-scale communication-efficient training of mixture-of-experts models in production. *ArXiv*, abs/2505.11432, 2025.

[23] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Clifford Young, Xiang Zhou, Zongwei Zhou, and David A Patterson. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[24] Yechan Kim, Hwijoon Lim, and Dongsu Han. Scaling beyond the GPU memory limit for large mixture-of-experts model training. In *Forty-first International Conference on Machine Learning*, 2024.

[25] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, et al. {GS}hard: Scaling giant models with conditional computation and automatic sharding. In *International Conference on Learning Representations*, 2021.

[26] Jiamin Li, Yimin Jiang, Yibo Zhu, Cong Wang, and Hong Xu. Accelerating distributed MoE training and inference with lina. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 945–959, Boston, MA, July 2023. USENIX Association.

[27] Heng Liao, Jiajin Tu, Jing Xia, Hu Liu, Xiping Zhou, Honghui Yuan, and Yuxing Hu. Ascend: a scalable and unified architecture for ubiquitous deep neural network computing : Industry track paper. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 789–801, 2021.

[28] Juncai Liu, Jessie Hui Wang, and Yimin Jiang. Janus: A unified distributed training framework for sparse mixture-of-experts models. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 486–498, New York, NY, USA, 2023. Association for Computing Machinery.

[29] Ziming Mao, Yihan Zhang, Chihan Cui, Kaichao You, Zhongjie Chen, Zhiying Xu, Scott Shenker, Costin Raiciu, Yang Zhou, and Ion Stoica. Uccl-ep: Portable expert-parallel communication. *ArXiv*, abs/2512.19849, 2025.

[30] Deepak Narayanan, Aaron Harlap, et al. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 1–15, New York, NY, USA, 2019. Association for Computing Machinery.

[31] Nvidia. The accelerated computing platform for next-generation workloads. https://www.nvidia.com/en-us/data-center/technologies/hopper-architecture/.

[32] Nvidia. Accessed: 2026-01. supported matrix shape of hopper. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#asynchronous-warpgroup-level-matrix-shape.

[33] Nvidia. cublas. https://developer.nvidia.com/cublas.

[34] NVIDIA. CUTLASS documentation: Blackwell SM100 GEMMs. https://docs.nvidia.com/cutlass/latest/media/docs/cpp/blackwell_functionality.html.

[35] Nvidia. Ibgda. https://docs.nvidia.com/nvshmem/release-notes-install-guide/prior-releases/release-280.html.

[36] Nvidia. Nvidia nvshmem. https://developer.nvidia.com/nvshmem-downloads.

[37] NVIDIA. PTX ISA: Warp-level matrix multiply-accumulate instructions. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#warp-level-matrix-instructions-mma.

[38] Nvidia. Stream management functions of the low-level cuda driver application programming interface. https://docs.nvidia.com/cuda/cuda-driver-api/.

[39] Nvidia. Nvidia collective communication library. 2025. https://docs.nvidia.com/deeplearning/nccl.

[40] Nvidia. Cutlass. 2026. https://docs.nvidia.com/cutlass/latest/.

[41] NVIDIA CUTLASS contributors. [QST] FP8 GEMM, 2024. Issue #1986. https://github.com/NVIDIA/cutlass/issues/1986.

[42] Muhammad Osama, Duane Merrill, Cris Cecka, Michael Garland, and John Douglas Owens. Stream-k: Work-centric parallel decomposition for dense matrix-matrix multiplication on the gpu. *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2023.

[43] Adam Paszke, Gross, et al. *PyTorch: an imperative style, high-performance deep learning library.* Curran Associates Inc., Red Hook, NY, USA, 2019.

[44] Pytorch. Introducing async tensor parallelism in pytorch. 2024. https://discuss.pytorch.org/t/distributed-w-torchtitan-introducing-async-tensor-parallelism-in-pytorch/209487.

[45] 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.

[46] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 68658–68685. Curran Associates, Inc., 2024.

[47] StepFun, Bin Wang, Bojun Wang, Changyi Wan, Guanzhe Huang, et al. Step-3 is large yet affordable: Model-system co-design for cost-effective decoding, 2025.

[48] Zhongling Su, Rong Fu, Weihan Cao, Jianfei Gao, Minxi Jin, Zhilin Pei, and Hui Wang. Tma-adaptive fp8 grouped gemm: Eliminating padding requirements in low-precision training and inference on hopper, 2025.

[49] Stuart H. Sul, Simran Arora, Benjamin F. Spector, and Christopher Ré. Parallelkittens: Systematic and practical simplification of multi-gpu ai kernels. In A. Chowdhery and Z. Jia, editors, *Proceedings of Machine Learning and Systems*, volume 8, pages 1076–1089. MLSys, 2026.

[50] Kimi Team, Yifan Bai, Yiping Bao, Y. Charles, Cheng Chen, Guanduo Chen, et al. Kimi k2: Open agentic intelligence, 2026.

[51] Meituan LongCat Team. Longcat-flash-omni technical report. *ArXiv*, abs/2511.00279, 2025.

[52] Meituan LongCat Team, Bayan, Bei Li, et al. Longcat-flash technical report, 2025.

[53] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025.

[54] Shulai Zhang, Ningxin Zheng, Haibin Lin, Ziheng Jiang, et al. Comet: Fine-grained computation-communication overlapping for mixture-of-experts, 2025.

[55] Tianyu Zhang, Peng Zhang, Yusong Gao, and Yun Zhang. Together with SGLang: Best practices for serving DeepSeek-R1 on H20-96G, 2025. https://www.lmsys.org/blog/2025-09-26-sglang-ant-group/.

[56] Chenggang Zhao, Chengqi Deng, Chong Ruan, et al. Insights into deepseek-v3: Scaling challenges and reflections on hardware for ai architectures. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ISCA '25, page 1731–1745, New York, NY, USA, 2025. Association for Computing Machinery.

[57] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 62557–62583. Curran Associates, Inc., 2024.