TEMPER: AMORTISING DATA-MOVEMENT OVERHEAD FOR DISTRIBUTED MOE INFERENCE ON MEMORY-CONSTRAINED GPUS

Anonymous authors
Paper under double-blind review

ABSTRACT

Mixture-of-Experts (MoE) models scale the capacity of large language models through sparse per-token computation, making them attractive for on-premises deployment on memory-constrained GPU servers. In practice, however, serving these large models requires expert parallelism together with weight offloading, which shifts the inference bottleneck from computation to data movement. Cross-GPU token exchange and host-to-device expert transfers dominate the execution timeline, and their exposed costs are further amplified by imbalanced routing and limited expert reuse. In this work, we present TEMPER, a distributed MoE inference system that amortises data-movement overhead along two complementary axes, reducing the exposed cost of each transfer and increasing the computation each transfer serves. TEMPER first determines token destinations before payload transfer, and overlaps token exchange with expert loading. Progressive expert replication then adds execution locations for hot experts, while replica-aware dispatch redistributes their token load. Finally, windowed layer-major execution reuses each fetched or replicated expert across several scheduling steps within a window of adjacent layers, and overlaps next-layer routing with in-flight expert transfers across steps. On Qwen3-30B-A3B with four PCIe-connected RTX 3080 Ti GPUs, TEMPER reduces mean end-to-end latency by 76.8-82.0% relative to three baselines under shared absolute arrival rates. Across system-specific normalized arrival rates, the reduction is 60.7-79.5%.

1 INTRODUCTION

With the rapid advancement of artificial intelligence, Large Language Models (LLMs) have demonstrated remarkable capabilities across a wide range of tasks. Driven by concerns over privacy protection and high network latency (Yu et al., 2024; Carlini et al., 2021), on-premises deployments on commodity GPUs [1] have attracted growing interest. However, the pursuit of improved accuracy for complex tasks has led to a continuous increase in model sizes, thereby exacerbating the computational and memory burdens on GPUs (Lee et al., 2024; Sheng et al., 2023). Confronting this dilemma, the Mixture-of-Experts (MoE) architecture (Lepikhin et al., 2021; Fedus et al., 2022) has emerged as a key solution, leveraging structural sparsity to reduce inference overhead by selectively activating a small subset of parameters while maintaining expressiveness.

As the MoE framework scales model parameters to hundreds of billions with expert weights accounting for approximately 90%, hosting the entire model on a single GPU is generally infeasible (Liu et al., 2024; Jiang et al., 2024). To address this memory bottleneck while maintaining accuracy, existing solutions typically fall into two categories: (1) **parallelism** (Fedus et al., 2022; Lepikhin et al., 2021), which distributes model weights (expert weights in **Expert Parallelism**) across multiple GPUs; and (2) **parameter offloading** (Lee et al., 2024; Kamahori et al., 2024), which transfers parameters to host memory and reloads them on demand, thereby alleviating the burden on GPUs.

[1] In this work, we mainly consider single-node multi-GPU servers with modest compute capacity and constrained interconnect bandwidth, serving workloads that exceed end-device capabilities but do not require cloud-scale clusters.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

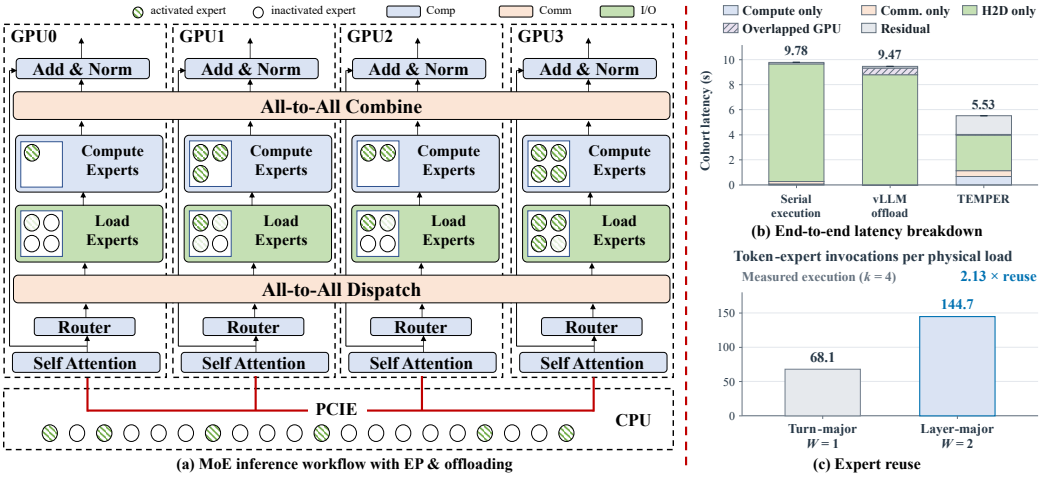

Figure 1: Data movement and expert reuse in offloaded expert parallelism. (a) A serial workflow comprising token dispatch, on-demand expert loading, expert computation, and output aggregation. (b) Estimated cohort-latency breakdown. GPU activity is partitioned into mutually exclusive intervals from a separately profiled rank; the residual is mean unprofiled latency minus total profiled GPU-active time. Whiskers show the range of two unprofiled runs. (c) Measured expert reuse under turn-major ($W = 1$) and windowed layer-major ($W = 2$) execution, with the same workload and 560 expert slots per GPU ($k = 4$). Reuse counts token-expert invocations per physical load, including unused loads.

However, vanilla parallelism still requires that the aggregate VRAM exceed the model size, whereas integrating parameter offloading with expert parallelism on edge servers introduces two forms of data movement into each MoE layer: cross-GPU exchanges of token activations and outputs, and host-to-GPU transfers of nonresident expert weights, as shown in Figure 1. In the serial workflow in Figure 1(a), expert loading follows token dispatch, exposing the two transfers sequentially. Figure 1(b) shows that weight loading remains a major latency component in the evaluated baselines. Transfer efficiency also depends on how much computation each loaded expert serves before eviction. Under the same workload and cache capacity, windowed layer-major execution increases token-expert invocations per physical load by $2.13\times$ (Figure 1(c)). These observations motivate coordinating transfer overlap with expert reuse.

In this work, we introduce **TEMPER**, a **T**ransfer-**E**fficient distributed **M**oE system that reduces data-movement overhead through decoupled communication, budgeted expert **P**lacement, and windowed **E**xecution **R**eordering. **Our key insight is that transfer efficiency depends on both the latency exposed on the critical path and the useful computation served by each loaded expert. TEMPER** separates global token scheduling from payload transfer using compact expert-demand metadata, allowing token exchange to overlap with expert loading. It further coalesces activations by destination rank and aggregates routing-weighted outputs locally before returning them. Guided by demand estimates, progressive expert replication creates additional execution locations for hot experts, while replica-aware dispatch redistributes their token load across these locations. Within a bounded window of adjacent layers, **TEMPER** processes multiple scheduling steps in layer-major order to increase the reuse of fetched and replicated experts. To maintain cross-layer concurrency, it overlaps next-layer routing with ongoing expert transfers across steps while preserving collective order. The primary contributions of this work are as follows:

- We decouple token scheduling from payload exchange to coalesce activations by destination rank, aggregate expert outputs locally, and overlap token communication with expert loading (Sec. 4.1).
- We combine progressive expert replication with replica-aware dispatch, redistributing token demand across available expert instances according to their estimated completion costs (Sec. 4.2).

- We introduce windowed layer-major execution to reuse fetched and replicated experts across scheduling steps and overlap next-layer routing with ongoing expert transfers while preserving collective order (Sec. 4.3).
- Experiments with Qwen3-30B-A3B on MMLU and LongBench using four PCIe-connected RTX 3080 Ti GPUs demonstrate 1.57-$1.92\times$ the throughput of the vLLM baselines and 4.26-$4.62\times$ that of KTransformers across the evaluated normalized online arrival rates. Mean E2E latency decreases by 60.7-79.5% relative to these baselines (Sec. 5).

2 Background

MoE architectures replace selected dense feed-forward networks with a gating network and N_e experts. For each token, the gate selects the top-K experts according to routing scores, and their outputs are combined through a weighted sum (Lepikhin et al., 2021; Fedus et al., 2022). This design expands model capacity without a proportional increase in per-token computation, but retains a substantial parameter memory footprint.

2.1 Expert Parallelism and Offloading

Expert parallelism (EP) distributes experts across GPUs. Token activations are dispatched to the GPUs hosting the selected experts, typically through all-to-all communication, and expert outputs are returned for weighted aggregation. When aggregate GPU memory is insufficient, expert offloading stores nonresident weights in host memory and transfers them to GPUs for execution. EP with offloading thus combines inter-GPU token exchange with host-to-GPU weight transfers. Fiddler selects CPU or GPU execution based on computation and weight-transfer costs (Kamahori et al., 2024). HybriMoE combines dynamic CPU-GPU scheduling with expert prefetching and caching (Zhong et al., 2025). However, their scheduling formulations do not cover inter-GPU token exchange or dispatch among expert replicas.

Input-dependent routing can produce uneven expert workloads. Expert replication redistributes computation among equivalent instances without changing logical expert selection. Lina estimates next-layer expert demand from profiled routing paths and refines replica allocation using actual routing results (Li et al., 2023). DeepSeek-V3 periodically adjusts redundant expert placement according to observed loads (Liu et al., 2024). These replication policies primarily target load balance. However, they do not explicitly organize execution across scheduling steps to amortize replica-loading costs under limited GPU memory.

2.2 Inference Scheduling and Expert Reuse

Inference scheduling governs batch composition and execution order. ORCA introduces iteration-level scheduling, allowing batch membership to change between model iterations (Yu et al., 2022). Under parameter offloading, execution order also governs weight reuse. FlexGen reuses each layer's loaded weights across multiple microbatches before advancing to the next layer (Sheng et al., 2023). MoE-Lightning adopts this ordering, pipelines CPU-GPU computation with data transfers, and supports multi-GPU tensor parallelism (Cao et al., 2025). Klotski groups computations by expert across batches to improve weight reuse and computation-transfer overlap (Fang et al., 2025).

These schedules do not explicitly coordinate expert reuse with token exchange under EP. Distributing expert workloads across GPUs can shorten local pipelines, but reducing layer latency requires coordinating token exchange with expert loading and computation. Reuse across scheduling steps further requires to the same resident expert instances. Execution reordering must preserve layer dependencies, per-request autoregressive dependencies, and collective order, while accommodating expert weights, intermediate activations, and KV caches within the available memory budget.

3 Problem Formulation

We consider MoE inference on a set of GPU workers $\mathcal{D}$, where worker d has memory capacity M_d. Expert (ℓ, e) denotes expert e in layer ℓ, with weight size $w_{\ell e}$. Each expert has a single host-memory copy and may have copies on multiple GPUs. Initial GPU residency is given. Request $r \in \mathcal{R}$ arrives

at time a_r and induces layer and autoregressive dependencies. A token t is identified by its request and sequence position. Let $\mathcal{S}_{\ell t}$ denote the expert indices selected for token t at layer ℓ, revealed upon completion of the corresponding routing operation.

An online policy π determines batching, token assignment, GPU weight loading and eviction, and execution order. For each $e \in \mathcal{S}_{\ell t}$, the destination $d_{\ell te} \in \mathcal{D}$ specifies the GPU executing that expert computation. Each selected computation executes exactly once, preserving the routing weights used for output aggregation. Decisions depend only on arrived requests, available routing results, current resource states, and available predictions. Destination assignments determine token communication and the required GPU expert copies. Creating a GPU copy requires a weight transfer; retaining copies on multiple GPUs provides replication.

A feasible schedule must satisfy memory, dependency, and resource constraints. Let $\mathcal{C}_d(\tau)$ contain the experts with GPU storage allocated on worker d at time τ, including weights being loaded. Memory usage satisfies $\sum_{(\ell,e)\in\mathcal{C}_d(\tau)} w_{\ell e} + U_d(\tau) \le M_d$, where $U_d(\tau)$ includes live activations, KV caches, communication buffers, workspace, and non-expert parameters. Buffers remain allocated through their final use. Expert computation requires both input availability and ready weights on the assigned GPU; these weights must remain resident until the computation finishes. Execution must preserve layer and autoregressive dependencies, with each operation starting only after its prerequisites have completed. Concurrent operations must respect shared compute and transfer capacities, and collective invocations must follow the same sequence across participating workers.

Let C_r^π denote the completion time of request r under policy π. We seek a causal policy minimizing expected mean request latency:

$$\min_{\pi\in\Pi_{\text{feas}}} \mathbb{E}\left[\frac{1}{|\mathcal{R}|}\sum_{r\in\mathcal{R}}(C_r^\pi - a_r)\right], \tag{1}$$

where Π_{feas} contains policies satisfying the above constraints and completing every request. The decisions are coupled across GPUs and time: distributing computation can increase token communication and weight transfers; replication consumes memory whose benefit depends on subsequent reuse; and reordering execution can amortize loading costs while increasing request waiting and intermediate-memory occupancy. Future arrivals and routing demands are unknown, requiring these trade-offs to be resolved with incomplete information.

4 System Design

Figure 2 illustrates TEMPER. Expert weights are backed by CPU memory and dynamically loaded into GPU caches. TEMPER coordinates three mechanisms: decoupled communication and expert loading, progressive expert replication and replica-aware token assignment, and windowed execution for expert reuse.

4.1 Decoupled Communication and Expert Loading

After routing, TEMPER exchanges per-expert token counts before transferring activations. These counts allow the replica-aware planner to determine token destinations and weight-loading requirements. Token dispatch and asynchronous expert loading can then overlap, with computation starting once both inputs and weights are ready. TEMPER also groups assigned invocations by token and destination GPU. For token t at layer ℓ, let $\mathcal{S}_{\ell t}^{(d)} = \{e \in \mathcal{S}_{\ell t} : d_{\ell te} = d\}$ contain the selected experts assigned to GPU d. Each nonempty remote group transfers one activation vector, together with its expert indices and routing weights. The destination computes a weighted partial result

$$p_{\ell td} = \sum_{e\in\mathcal{S}_{\ell t}^{(d)}} \alpha_{\ell te} f_{\ell e}(h_{\ell t}), \qquad y_{\ell t} = \sum_{d\in\mathcal{D}} p_{\ell td}, \tag{2}$$

where $h_{\ell t}$ is the expert input, $f_{\ell e}$ is the expert function, and $\alpha_{\ell te}$ is the original routing weight. Each remote destination returns one partial result per token; local assignments require no inter-GPU transfer. This grouping reduces repeated transfers while preserving logical expert selection and routing weights.

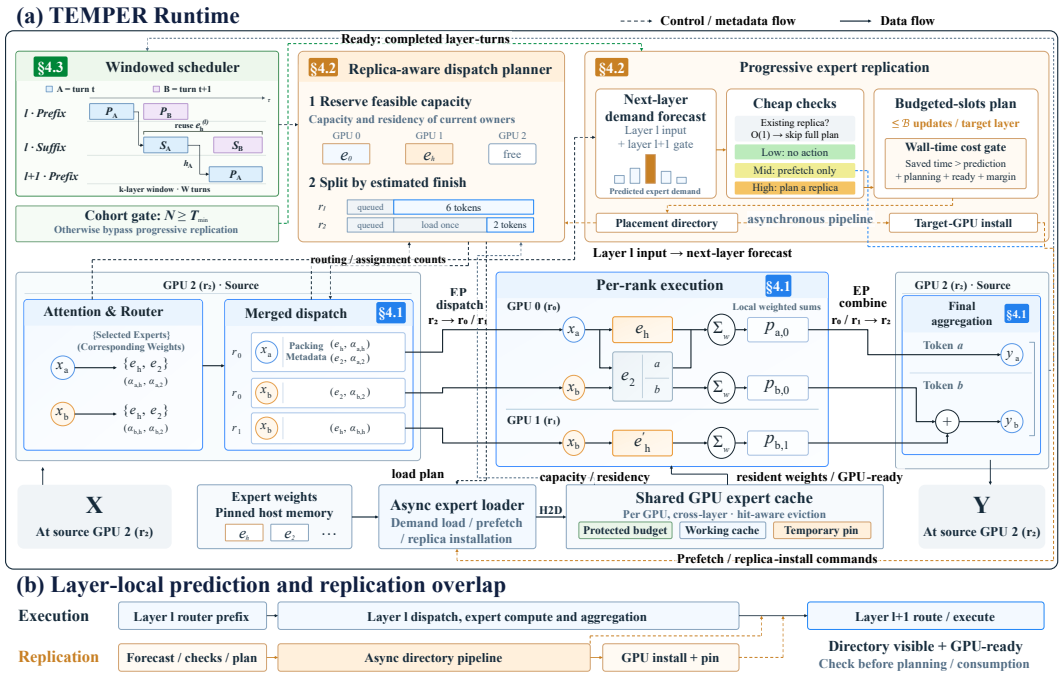

Figure 2: Overview of TEMPER. (a) Runtime coordination of token exchange, expert loading, replica planning, and windowed execution. (b) Next-layer prediction and replica preparation overlap with current-layer execution. Solid and dashed arrows denote data and control flow, respectively.

4.2 Progressive Expert Replication and Replica-aware Token Assignment

Progressive expert replication prepares additional instances using predicted demand, a replica budget, and cost-based admission. Replica-aware token assignment uses actual routing counts to allocate computation. The placement directory records designated execution locations separately from GPU residency, namely a designated instance may require loading before execution.

Prediction-guided progressive replication. The predictor applies layer ℓ's gate to the attention input of layer $\ell - 1$, producing $\widehat{\mathcal{S}}_{\ell t}$ before actual routing. These predictions guide expert prefetching and provide the demand estimates for progressive replication. For the tokens $\mathcal{T}$ in the current execution cohort (Appendix A), the predicted demand is $\widehat{q}_{\ell e} = \sum_{t \in \mathcal{T}} \mathbf{1}[e \in \widehat{\mathcal{S}}_{\ell t}]$. When predictions match actual routing, prefetched weights can reduce loading stalls, while prepared replicas provide additional execution locations. Unpredicted nonresident experts are loaded asynchronously according to the actual assignments.

Replication planning requires $|\mathcal{T}| \geq T_{\min}$. For each candidate expert-GPU pair, an $O(1)$ directory lookup skips replica creation if the expert already has an instance assigned to that target GPU. Instances on other GPUs do not exclude the candidate destination. A coarse screen uses predicted demand and cost thresholds to select among no additional speculative action, ordinary prefetching, and replica planning. Only the last branch constructs a placement update.

The planner constructs a feasible update a under a configurable budget $\mathcal{B}$ on new expert-GPU instances per layer and execution cohort. Any slot replacements preserve at least one designated location for every expert, respect per-GPU placement capacity, and avoid duplicate instances of the same expert on one GPU. For placement P, let $\mathcal{D}_{\ell e}(P)$ denote the designated GPUs for expert (ℓ, e). The predicted GPU load is

$$\widehat{L}_{\ell d}(P) = \sum_{e} \frac{\widehat{q}_{\ell e} \mathbf{1}[d \in \mathcal{D}_{\ell e}(P)]}{|\mathcal{D}_{\ell e}(P)|}. \tag{3}$$

For an update a producing placement P^a, the estimated execution-time gain is $\widehat{G}_\ell(a) = \gamma[\max_d \widehat{L}_{\ell d}(P) - \max_d \widehat{L}_{\ell d}(P^a)]$, where γ converts predicted invocation load into time. The gain is evaluated for the resulting placement as a whole, accounting for interactions among added replicas. This approximation distributes demand uniformly across designated instances; actual assignments are determined by the dispatcher.

After planning, the update is admitted only if $\widehat{G}_\ell(a) > c_{\text{pred}} + \max(c_{\text{plan}}(a), c_{\text{floor}}) + c_{\text{ready}}(a) + \epsilon$. Here, c_{pred} is the measured prediction cost, charged once per layer and execution cohort; $c_{\text{plan}}(a)$ is the measured planning cost; and c_{floor} is a planning-cost floor. The term $c_{\text{ready}}(a)$ estimates the elapsed time to install the update's new instances and make their weights available on the target GPUs. The margin ϵ sets the minimum estimated net benefit. This test provides a cost-based admission heuristic.

Accepted updates are submitted during the preceding layer, with directory coordination and weight installation proceeding asynchronously. The target layer checks update visibility before token assignment and waits for GPU readiness before computation. New replicas receive temporary protection across the intended turns, after which ordinary cache management resumes. Placement persists across windows, and an existing directory entry whose weights have been evicted can be reloaded without repeating replica planning.

Replica-aware token assignment. For a fixed layer, let n_{ue} denote the number of invocations for expert e originating from GPU u. The dispatcher first reserves a baseline location for every active expert, processing experts in descending demand. A reservation is accepted only if the remaining experts still admit a capacity-feasible assignment. The configured working capacity accommodates this baseline, and remaining capacity supports additional instances.

Nonempty source-expert groups are then processed in descending order of n_{ue}. For destination d, let L_d denote its accumulated estimated cost. The planner constructs the completion-cost sequence

$$F_{ued}(j) = L_d + a_{ed} + j\,(c_{\text{comp}} + \mathbf{1}[u \neq d] c_{\text{remote}})\,, \qquad j \geq 1, \tag{4}$$

where c_{comp} is the per-invocation computation cost, c_{remote} penalizes remote assignment, and a_{ed} is a one-time loading cost. The loading term is zero for ready weights or instances whose loading cost has already been charged in the current plan.

The planner assigns the n_{ue} invocations by repeatedly choosing the GPU with the lowest estimated completion cost for its next invocation. If GPU d receives $m_d > 0$ invocations, the planner updates $L_d \leftarrow F_{ued}(m_d)$ and records any newly activated instance. Computations wait for required weights already being loaded on their destination GPU. Assignments are then grouped by token and destination for communication (Sec. 4.1).

4.3 Windowed Execution and Expert Reuse

TEMPER processes W scheduling turns within a window of k adjacent layers(Appendix A). Each layer is divided into a local prefix comprising attention, normalization, and routing, and an execution suffix comprising count exchange, assignment, dispatch, expert execution, and output aggregation. Within each window, suffixes follow layer-major order across the W turns, shortening the interval between accesses to the same expert instances.

GPU expert storage is shared across layers. A bounded protection budget retains instances according to observed demand while preserving baseline working capacity. When replacement is necessary, the cache first considers evictable instances from the preceding layer, ordered by their latest actual routing hit counts, with LRU tie-breaking. If no such instance is available, it falls back to LRU among other eligible instances. Loading, protected, and in-use weights are excluded from eviction. Per-turn activations, saved prefix results, and request KV state are included in the memory budget.

For turns A and B, the prefix of B at layer ℓ can overlap with the suffix of A. Once A completes layer ℓ, its prefix at layer $\ell+1$ can overlap with the remaining suffix of B at layer ℓ. Prefix results are saved for subsequent execution, avoiding repeated attention computation and KV-cache updates. All collective operations retain a common layer-major invocation order across workers. This schedule combines expert reuse across turns with overlap between current-layer execution and next-layer routing and expert preparation.

5 EVALUATION

We evaluate TEMPER under normalized online loads, shared absolute arrival rates, and offline workloads, examining throughput, request latency, and memory usage, followed by component ablations. At $\rho = 1$, TEMPER achieves 1.79-$1.85\times$ the throughput of the vLLM baselines and 4.26-$4.37\times$ that of KTransformers. Under absolute arrival rates, TEMPER reduces mean E2E latency by 76.8-82.0% across the three baselines and also improves first-token responsiveness on both workloads.

5.1 EXPERIMENTAL SETUP

We evaluate Qwen3-30B-A3B (Yang et al., 2025) with unquantized BF16 weights on a server equipped with two Intel Xeon Gold 6248 CPUs and 240 GiB of available host memory. Each experiment uses four NVIDIA RTX 3080 Ti GPUs, each with 12 GiB of memory, connected through PCIe without NVLink. Experiments run sequentially to avoid resource interference.

We compare TEMPER with vLLM-EP, vLLM-DP, and KTransformers (KVCache-AI, 2024). Both vLLM baselines use the same codebase as TEMPER and layer-granularity parameter offloading. vLLM-EP combines four-way data and expert parallelism, whereas vLLM-DP runs independent model instances and uses the same layer-granularity offloader. KTransformers uses four-way tensor and expert parallelism, retaining eight experts per layer on each GPU and executing the remaining experts on CPUs. Our CPUs lack AMX and AVX-512 BF16 support, precluding the corresponding accelerated CPU expert kernels. The vLLM systems allow four concurrent sequences per worker, while KTransformers allows four concurrent requests in total.

TEMPER uses $k = 4$ layers and $W = 2$ scheduling turns per window. Each GPU provides 560 expert-cache slots, including 96 protected slots, and a 512 MiB KV cache. Progressive replication uses $T_{\min} = 1024$ tokens and a budget of $\mathcal{B} = 1$ new instance per layer and cohort.

We use MMLU (Hendrycks et al., 2021) prompts of 63-94 tokens with 64-token outputs and LongBench (Bai et al., 2024) prompts of 3,844-4,312 tokens with 128-token outputs. Early termination is disabled to control output length. Online traces use exponential inter-arrival times. In the offline setting, denoted *fixed*, requests arrive simultaneously within each request group.

For the normalized arrival-rate sweep, the offered rate is $\lambda = \rho\lambda_{\text{ref}}$, where λ_{ref} is a separately calibrated, system-specific reference throughput and $\rho \in \{0.5, 0.75, 1.0, 1.25\}$. Equal ρ represents equal relative load, with different absolute arrival rates across systems. We additionally evaluate shared absolute arrival rates. Within each workload, all systems receive identical requests at the same scheduled arrival times.

We report achieved request throughput, TTFT, TPOT, and E2E latency. Throughput is measured over the complete request trace, including arrival gaps and the final drain period. TTFT and E2E latency include queueing. Memory measurements report the maximum sampled GPU memory across workers and peak host process-tree RSS, including startup and warmup. Additional runtime settings and measurement details are provided in Appendix B.

5.2 PERFORMANCE

Figure 3 compares performance across normalized arrival rates and offline workloads. At $\rho = 1$, TEMPER achieves 1.79-$1.80\times$ the throughput of the vLLM baselines on MMLU and 1.82-$1.85\times$ on LongBench. The corresponding gains over KTransformers are $4.26\times$ and $4.37\times$. Across the three baselines, mean E2E latency decreases by 60.7-65.4% and 73.3-74.8%, respectively. These latency gains primarily arise during generation, with mean TPOT decreasing by 63.8-65.5% on MMLU and 78.5-78.6% on LongBench relative to the vLLM baselines. Windowed execution increases the computation served by each loaded expert through weight reuse across scheduling turns. On LongBench, progressive replication additionally prepares expert instances for predicted demand, enabling replica-aware dispatch to assign subsequent work to them. Retaining and reusing these instances amortizes loading costs over more token-expert invocations, reducing the weight-transfer overhead per output token.

vLLM-EP and vLLM-DP achieve similar throughput, differing by only 0.2% on MMLU and 1.9% on LongBench at $\rho = 1$. Their similar throughput suggests that changing the parallelism configu-

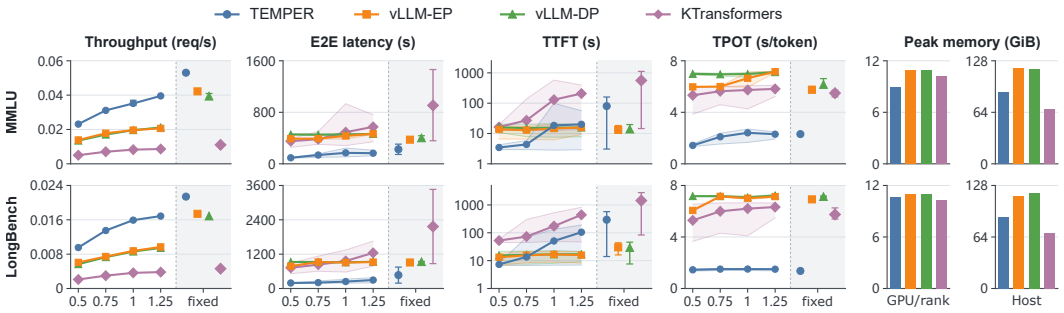

Figure 3: Serving performance on MMLU (top) and LongBench (bottom). *fixed* denotes offline execution with simultaneous request arrivals. Latency bands and whiskers indicate request-level min-max ranges. TTFT uses a logarithmic scale. Memory bars are reported at $\rho = 0.5$.

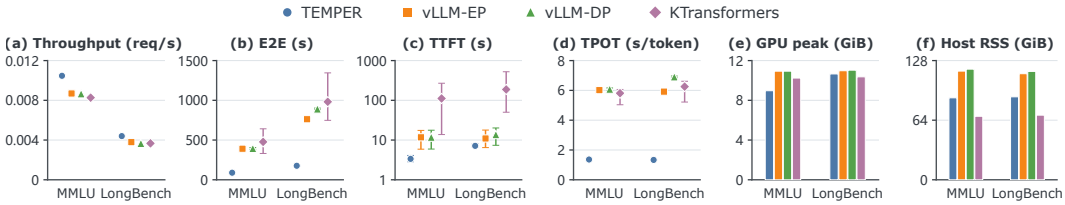

Figure 4: Serving performance under shared absolute arrival rates. Latency markers and whiskers show means and request-level min-max ranges.

ration alone does not resolve the shared weight-movement cost. TEMPER additionally coordinates weight preparation and increases the computation served by each loaded instance.

Under offline workloads, TEMPER retains throughput gains of 1.26-1.35× on MMLU and 1.23-1.27× on LongBench over the vLLM baselines. Simultaneous arrivals provide more concurrent work over which all systems can amortize loading costs, consistent with the smaller throughput gap than at $\rho = 1$. However, on fixed LongBench, TEMPER's mean TTFT is 9.8-10.0× that of the vLLM baselines, despite 48.5-50.0% lower E2E latency. Layer-major execution allows several turns to share loaded experts before advancing to later layers, which can defer an individual request's first token. The reduction in TPOT compensates for this initial delay in the measured workloads.

The normalized arrival-rate sweep exhibits a similar trade-off between first-token responsiveness and generation efficiency. On LongBench, increasing ρ from 0.5 to 1 raises TEMPER's mean TPOT by only 4.1%, while mean TTFT increases 6.9-fold. The increase in TTFT accounts for 85.1% of the E2E latency increase. At $\rho = 1$, TEMPER's mean TTFT is approximately 3.0× that of the vLLM baselines. This pattern is consistent with increased waiting and resource contention during prefill as requests overlap, while expert reuse sustains efficient generation. The comparison also reflects TEMPER's higher absolute offered rate at equal ρ.

Figure 4 compares the systems under identical requests and arrival traces. TEMPER achieves 1.20-1.27× the throughput of the three baselines on MMLU and 1.16-1.20× on LongBench. Mean E2E latency decreases by 77.2-81.3% and 76.8-82.0%, respectively. These throughput measurements include arrival gaps and the final drain period, reflecting completion of the finite request trace.

Under these shared arrival rates, TEMPER also reduces mean TTFT relative to the vLLM baselines by 71.3-71.5% on MMLU and 34.8-48.1% on LongBench. Mean TPOT decreases by 77.4-80.7% across the two workloads, consistent with sustained generation benefits from amortizing expert-loading costs. The lower TTFT is consistent with reduced prefill waiting and contention at these arrival rates. Thus, the higher TTFT observed at equal ρ does not persist under the shared arrival traces, highlighting the dependence of first-token responsiveness on the evaluated load.

Table 1: Cumulative ablation of TEMPER. Throughput is measured in tokens/s.

Configuration	MMLU	LongBench
Native	2.830	0.563
+ Communication	5.781	2.060
+ Window	7.666	3.016
+ Predictive prefetch	8.321	3.622
+ Progressive expert replication (Full TEMPER)	**8.319**	**4.032**

Table 2: Runtime activity of prefetching and progressive replication.

Metric	MMLU	LongBench
GPU replica installations/run	2	5
Existing-replica planning skips/run	9	12
Predictive prefetch submissions/run	32	51
New H2D replica copies	0	3

For runs at $\rho = 0.5$ in Figure 3, TEMPER reduces peak GPU memory by 18.0% on MMLU and 3.1% on LongBench relative to the vLLM baselines. Peak host RSS decreases by 22.6-25.1% across the two workloads. These peaks include startup and warmup allocations (Appendix B.3). TEMPER's bounded expert cache limits weight residency. Longer inputs require additional activation and saved-prefix storage, consistent with the smaller GPU-memory reduction on LongBench.

TEMPER's peak host RSS is 28.4-29.8% higher than KTransformers across the two workloads. Its peak GPU memory is also 2.8% higher on LongBench. KTransformers limits GPU weight residency to eight experts per layer per GPU and executes the remaining expert work on CPUs. Its lower request concurrency further limits simultaneously active request state. This configuration exhibits a smaller host-memory footprint alongside higher generation latency. At $\rho = 1$, KTransformers' mean TPOT is $2.38\times$ and $4.11\times$ that of TEMPER on MMLU and LongBench, respectively. Figure 4 shows a similar memory trade-off under shared arrivals.

5.3 Ablation Study

Table 1 evaluates TEMPER through cumulative ablations starting from Native. Communication decouples token exchange from expert loading and groups transfers by destination. Window enables expert reuse through windowed layer-major execution. Predictive Prefetch prepares next-layer weights using predicted demand, and Progressive Expert Replication adds execution locations for replica-aware token assignment. Each configuration retains the preceding optimizations.

Communication provides the largest incremental gain, raising throughput to $2.04\times$ and $3.66\times$ Native on MMLU and LongBench, respectively, by coalescing transfers and overlapping dispatch with expert loading. Window adds 32.6% and 46.4% through cross-turn expert reuse, consistent with the $2.13\times$ increase in invocations per load in Figure 1(c). Predictive Prefetch advances next-layer weight transfers, with throughput improving by another 8.5% and 20.1%. Enabling replication further improves LongBench throughput by 11.3%, while MMLU throughput remains essentially unchanged. Additional execution locations support workload redistribution through replica-aware dispatch and reuse across turns, although their benefit varies with the workload.

Table 2 reports the runtime activity. MMLU and LongBench record 32 and 51 predictive prefetch submissions, two and five GPU replica installations, and nine and 12 existing-replica planning skips per run, respectively. These skips avoid repeated planning for experts already assigned to the target GPU. The recorded numbers of new H2D replica copies are zero and three, respectively, so installation counts should not be equated with new weight transfers. The MMLU results also show that recorded replica activity does not necessarily yield additional throughput. The cumulative gains remain conditional on preceding optimizations and do not isolate their interactions.

6 Conclusion

We presented **TEMPER**, a distributed MoE inference system for memory-constrained multi-GPU servers. By integrating decoupled communication and expert loading, progressive expert replication and replica-aware token assignment, and windowed layer-major execution, **TEMPER** overlaps token communication with weight transfers and amortizes loading costs through expert reuse. Experiments on PCIe-connected commodity GPUs demonstrate higher throughput and lower end-to-end latency than the three evaluated baselines.

AI USE STATEMENT

In this work, we used generative AI tools mainly for writing polish. We have not used generative AI tools for related work searching and research ideation or execution, and generating synthetic datasets or proving mathematical claims are not applicable to this work. Additionally, we used generative AI tools for code debugging. We have reviewed all AI-assisted work. All revisions to the writing and code debugging generated by generative AI tools have been reviewed by two authors. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

REFERENCES

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. LongBench: A bilingual, multitask benchmark for long context understanding. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3119–3137, Bangkok, Thailand, Aug 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.172. URL https://aclanthology.org/2024.acl-long.172.

Shiyi Cao, Shu Liu, Tyler Griggs, Peter Schafhalter, Xiaoxuan Liu, Ying Sheng, Joseph E. Gonzalez, Matei Zaharia, and Ion Stoica. Moe-lightning: High-throughput moe inference on memory-constrained gpus. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, pp. 715–730, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400706981. doi: 10.1145/3669940.3707267. URL https://doi.org/10.1145/3669940.3707267.

Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Ulfar Erlingsson, et al. Extracting training data from large language models. In *30th USENIX security symposium (USENIX Security 21)*, pp. 2633–2650, 2021.

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, pp. 574–588, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400710797. doi: 10.1145/3676641.3716261. URL https://doi.org/10.1145/3676641.3716261.

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. URL http://jmlr.org/papers/v23/21-0998.html.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021.

Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mixtral of experts, 2024. URL https://arxiv.org/abs/2401.04088.

Keisuke Kamahori, Yile Gu, Kan Zhu, and Baris Kasikci. Fiddler: CPU-GPU orchestration for fast inference of mixture-of-experts models. In *5th Workshop on practical ML for limited/low resource settings*, 2024. URL https://openreview.net/forum?id=WX7lxohjFe.

KVCache-AI. Ktransformers: A flexible framework for experiencing cutting-edge llm inference optimizations, 2024. URL https://github.com/kvcache-ai/ktransformers.

Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. Infinigen: efficient generative inference of large language models with dynamic kv cache management. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation*, OSDI'24, USA, 2024. USENIX Association. ISBN 978-1-939133-40-3.

Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. {GS}hard: Scaling giant models with conditional computation and automatic sharding. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=qrwe7XHTmYb.

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)*, pp. 945–959, Boston, MA, July 2023. USENIX Association. ISBN 978-1-939133-35-9. URL https://www.usenix.org/conference/atc23/presentation/li-jiamin.

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

Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pp. 31094–31116. PMLR, 2023.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Da Yu, Peter Kairouz, Sewoong Oh, and Zheng Xu. Privacy-preserving instructions for aligning large language models. In *Proceedings of the 41st International Conference on Machine Learning*, pp. 57480–57506, 2024.

Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/yu.

Shuzhang Zhong, Yanfan Sun, Ling Liang, Runsheng Wang, Ru Huang, and Meng Li. Hybrimoe: Hybrid cpu-gpu scheduling and cache management for efficient moe inference. In *Proceedings of the 62nd Annual ACM/IEEE Design Automation Conference*, DAC '25. IEEE Press, 2025. ISBN 9798331503048. doi: 10.1109/DAC63849.2025.11133274. URL https://doi.org/10.1109/DAC63849.2025.11133274.

A Scheduling Terminology

A scheduling turn comprises the dependency-ready token batches selected on the participating EP workers. The selected computations must satisfy their layer and autoregressive dependencies before execution. An execution cohort groups W such turns, which TEMPER processes in layer-major order within windows of k adjacent layers.

For progressive replication, $\mathcal{T}$ contains all non-padding tokens across the W turns and all EP workers in the current cohort. The threshold $T_{\min}$ applies to this global token count, and the budget $\mathcal{B}$ limits new expert-GPU instances per layer and execution cohort.

A request group instead denotes complete requests submitted simultaneously in the offline experiments. It is an external workload unit and need not correspond to a single execution cohort.

B Additional Experimental Details

B.1 Runtime Configuration

Each experiment uses four RTX 3080 Ti GPUs from one PCIe switch group. TEMPER and the vLLM baselines use GPUs 0-3, while KTransformers uses GPUs 4-7; both groups have the same within-group PCIe topology and no NVLink. The systems run sequentially on the same server and use the same unquantized BF16 model.

The vLLM systems use eager execution, Triton MoE kernels, and asynchronous scheduling, with dual-batch overlap disabled. vLLM-EP uses linear expert placement and the standard all-gather/reduce-scatter communication backend. Both vLLM baselines use the layer-granularity `PrefetchOffloader`, configured with group size 1, one group, and prefetch step 2, and offload only expert weights. vLLM-DP disables expert parallelism and therefore performs no inter-GPU expert-token dispatch.

TEMPER uses a persistent expert cache shared across layers, with 560 slots per GPU, 96 protected slots, and at most two protected slots per layer. No separate staging slots are allocated. The cache uses layer-hit information with LRU tie-breaking, and expert weights are backed by pinned host memory. Progressive replication uses a 1 ms planning-cost floor, a 4 ms GPU-copy cost estimate, and a 5 ms minimum net-benefit margin. The copy estimate is calibrated from a single-expert microbenchmark and does not include the full distributed directory-commit latency.

KTransformers uses FlashInfer attention, uniform expert placement, ten CPU inference threads, and two thread pools. Its validated GPU residency is eight experts per layer per rank, or 32 across the four GPUs. Its chunked-prefill limits are 16,384 tokens for MMLU and 2,048 for LongBench; the latter uses a GPU-prefill threshold of 4,096 tokens. The maximum sequence lengths are 256 and 4,608, respectively. These are also the per-worker batched-token limits for the vLLM systems.

B.2 Workloads and Measurement Protocol

MMLU requests are drawn from the test split, while LongBench requests use the `gov_report_e` subset. Systems receive matched request sequences within each comparison. Outputs are fixed to 64 tokens for MMLU and 128 for LongBench by disabling early termination.

Each online run contains 16 measured requests. Inter-arrival times are sampled from an exponential distribution and rescaled to the target trace duration. The arrival schedule has no ramp-up phase or client-side concurrency limit. Before measurement, each system processes four independent warmup requests, each generating eight tokens. Warmup is excluded from throughput and latency measurements, while the resulting cache and placement state is retained.

For the normalized arrival-rate sweep in Figure 3, the offered rate is $\lambda = \rho\lambda_{\mathrm{ref}}$, with $\rho \in \{0.5, 0.75, 1.0, 1.25\}$ and a separately calibrated, system-specific reference throughput. The original calibration rates remain fixed when additional offline results become available. Equal ρ therefore indicates equal relative load, with different absolute arrival rates across systems.

For the shared-rate comparison in Figure 4, all systems use $\lambda = 0.0111$ requests/s on MMLU and $\lambda = 0.00462146$ requests/s on LongBench. Within each workload, request identities, ordering, scheduled arrival times, and output lengths are identical across systems.

Offline MMLU runs comprise four successive request groups of 16 requests, while offline LongBench runs comprise one such group. All requests within a group are submitted simultaneously. These externally submitted request groups are distinct from the execution cohorts formed internally by TEMPER. Offline results use three runs per system for MMLU and one for LongBench.

B.3 Metrics and Aggregation

Request throughput is the number of completed requests divided by the measured trace duration, including arrival gaps and the final drain period. Time to first token (TTFT) and end-to-end (E2E) latency include waiting after request submission. Time per output token (TPOT) is computed from successive output-token intervals within each request and then averaged across requests.

For KTransformers, TPOT is estimated from streaming text-delta intervals. We verify that the event count matches the output-token count and that an N-token output contains exactly $N - 1$ intervals. The result is cross-checked against $(\mathrm{E2E} - \mathrm{TTFT})/(N - 1)$. This validated streaming-interval proxy is distinct from native token-level timestamps.

Figure 3 reports mean throughput across runs and mean latency across measured requests. Latency bands and whiskers show pooled request-level min-max ranges, while throughput whiskers show run-level min-max ranges. Figure 4 reports throughput from a single run and mean latency over its 16 requests. Its latency whiskers indicate within-run request-level min-max ranges. Single-run throughput has no across-run range. All ranges are descriptive, not confidence intervals.

Memory bars in Figure 3 use the $\rho = 0.5$ runs. Those in Figure 4 use the corresponding shared-rate runs. Memory peaks are taken over the full monitored service lifecycle, including startup, warmup, serving, and cleanup. GPU memory is the maximum sampled usage over participating workers and time, rather than the sum across GPUs. Host memory is peak process-tree RSS, including model storage, runtime allocations, and buffers. RSS does not measure unique physical memory ownership, and the measurements do not decompose the difference between TEMPER and KTransformers by allocation type.