ZipMoE: Efficient On-Device MoE Serving via Lossless Compression and Cache-Affinity Scheduling

Yuchen Yang [*1] **Yaru Zhao** [*1] **Pu Yang** [1] **Shaowei Wang** [1] **Zhi-Hua Zhou** [2 3]

Abstract

While Mixture-of-Experts (MoE) architectures substantially bolster the expressive power of large-language models, their prohibitive memory footprint severely impedes the practical deployment on resource-constrained edge devices, especially when model behavior must be preserved without relying on lossy quantization. In this paper, we present ZIPMOE, an efficient and *semantically lossless* on-device MoE serving system. ZIPMOE exploits the synergy between the hardware properties of edge devices and the statistical redundancy inherent to MoE parameters via a caching-scheduling co-design with provable performance guarantee. Fundamentally, our design shifts the paradigm of on-device MoE inference from an I/O-bound bottleneck to a *compute-centric* workflow that enables efficient parallelization. We implement a prototype of ZIPMOE and conduct extensive experiments on representative edge computing platforms using popular open-source MoE models and real-world workloads. Our evaluation reveals that ZIPMOE achieves up to 72.77% inference latency reduction and up to 6.76× higher throughput than the state-of-the-art systems. Our code is available at: https://github.com/npnothard/ZipMoE-ICML26.

1. Introduction

Recent years have witnessed increasing interest in deploying large language models (LLMs) on edge and mobile devices. On-device inference obviates the need to transmit sensitive data to remote clouds, improves service availability, and avoids the service interruption caused by the unreliable wide-area networks(Yang & Wang, 2024a;b), thereby enabling a broad class of mobile AI applications, including embodied intelligence (Sun et al., 2024), personal assistants (Qu et al., 2025), and chatbots (Neumann et al., 2025). Among existing LLM architectures, Mixture-of-Experts (MoE), a classic ensemble learning method (Zhou, 2025; Zhang & Zhou, 2013; Jacobs et al., 1991), has been found well effective in training modern LLMs. By decomposing the model into specialized experts and activating only a small subset per token, MoE substantially increases model capacity while keeping computational overhead sub-linear to the model size. However, the sparse activation nature of MoE models fundamentally shifts the system bottleneck from computation to memory, which presents a critical barrier to their practical on-device deployment under tight memory constraints.

To improve memory efficiency for on-device LLM inference, prior works have primarily focused on model quantization (Frantar et al., 2022; Fu et al., 2025) and pruning (Ma et al., 2023), which reduce model size by representing model weights or structures in lower-precision or more compact forms. Building on the insight that quantization sensitivity is highly non-uniform, varying across tensors (Zhou et al., 2025; Duanmu et al., 2025), experts (Yi et al., 2025), and even tokens within the same expert (Wang et al., 2025), state-of-the-art solutions adopt fine-grained, adaptive bit-width allocation combined with system-aware optimizations to balance the accuracy–efficiency trade-off (Sheng et al., 2023). While effective in reducing memory footprint, such *lossy* compression fundamentally shifts model behavior beyond what existing evaluation metrics (e.g., perplexity and zero-shot accuracy) can capture, particularly regarding security (Kumar et al., 2024; Liu et al., 2025). It is proved feasible to construct adversarial LLMs with quantized parameters that induce vulnerable code generation, over-refusal, and content injection attacks (Shu et al., 2023), whereas their full-precision counterparts behave benignly (Egashira et al., 2024). These findings highlight that quantization-based inference systems can introduce vulnerabilities to malicious attacks, particularly in unmonitored on-device deployments.

An alternative line of work exploits the inherent activation sparsity of MoE models by caching a subset of experts

[*]Equal contribution [1]School of Electronic Science and Engineering, Nanjing University, China. [2]National Key Laboratory for Novel Software Technology, Nanjing University, China. [3]School of Artificial Intelligence, Nanjing University, China. Correspondence to: Shaowei Wang <wangsw@nju.edu.cn>.

Proceedings of the 43^{rd} International Conference on Machine Learning, Seoul, South Korea. PMLR 306, 2026.

in GPU memory while offloading inactive parameters to lower-tier storage. These offloaded tensors are fetched either proactively via data-driven predictors (Du et al., 2024; Song et al., 2024) or reactively upon gate activation. To mitigate I/O bottlenecks caused by frequently transferring experts over PCIe, existing serving systems employ pipelined execution to overlap GPU inference with I/O events (Cao et al., 2025; Fang et al., 2025). Despite improving throughput on consumer- and server-grade hardware, these approaches exhibit fundamental mismatches with real-world on-device deployment. First, on-device applications (e.g., interactive chatbots and assistants) prioritize low-latency interactive inference where the batch size is typically one (Kong et al., 2024), severely limiting the efficacy of pipelining and batch-level optimizations. Second, existing systems operate on the premise that CPU main memory serves as a distinct external storage for offloaded tensors (Eliseev & Mazur, 2023; Yu et al., 2026; Xue et al., 2024). This assumption rarely holds true and is rendered impractical on edge platforms (e.g., Jetson platforms, mobile phones, and Raspberry Pi) that are predominantly built on mobile system-on-chips (SoCs), where CPUs and GPUs share the same physical memory pool and bus resources. Consequently, existing solutions fail to exploit, or even account for the unique performance characteristics of shared-memory architectures. Note that this oversight also severely undermines emerging CPU-GPU hybrid inference systems (Chen et al., 2025; Kamahori et al., 2024). These fundamental mismatches between existing MoE serving systems and practical on-device deployments motivate us to reconsider a core question:

How can MoE models be efficiently served on mobile and edge platforms without altering their algorithmic behavior?

In this paper, we tackle this challenge by presenting ZIPMOE, the *first* system to accelerate MoE inference on mobile and edge computing platforms through lossless compression. Recognizing the significant statistical redundancy within specific bit-fields of MoE parameters, we first introduce a compression-decompression pipeline to obtain parameters that (i) minimize PCIe I/O through *lossless* compression, and (ii) support parallel decompression on the CPU. This pipeline enables bit-level operations on tensor entries and ensures high efficiency by leveraging a zero-copy paradigm and a memory-coalesced GPU kernel for tensor recovery, which incurs negligible runtime overhead by fully exploiting the unified memory architecture (UMA) of mobile SoCs. Second, we explore hierarchical, differentiated caching to manage tensors across distinct compression states for fine-grained memory control. We devise a cache-pool planning algorithm based on probabilistic modeling of expert skewness and dynamic programming to determine the optimal memory budget for each compression state. This hybrid caching strategy enables parallel, asynchronous on-demand decompression, orchestrated by our cache-affinity scheduling algorithm, which we prove to perform within a constant factor of the global optimum. Ultimately, our design rethinks expert loading in MoE inference by *computing the required expert tensors* rather than *blocking on memory transfers*, effectively unleashing the potential of the powerful yet often underutilized multi-core CPUs on modern SoCs.

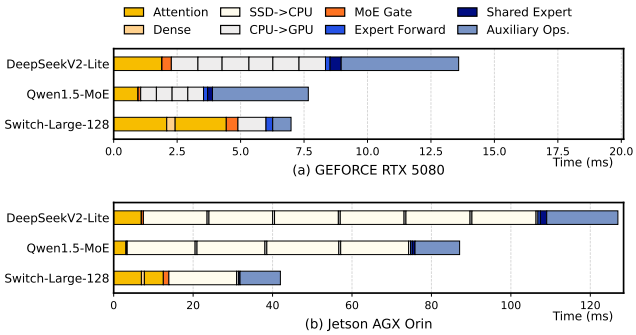

Figure 1. Latency break-down of decoding layers in representative MoE models on (a) Server environment, where experts are offloaded to CPU with 512 GB RAM; and (b) Edge environment, where experts are offloaded to NVMe SSD (Aigo DP35) with 2 GB/s read speed.

2. Background and Motivation

2.1. System-Hardware Mismatch in Existing Offloading Solutions

Unlike server-grade systems, mainstream mobile and edge computing platforms (e.g., NVIDIA Jetson, Raspberry Pi, Apple Silicon) are predominantly UMA-based hardware, where heterogeneous processors (e.g., CPUs and GPUs) share a single, coherent physical memory pool. To examine the implications of this architectural shift, we benchmark offloading-based MoE inference on both a server platform (Fig.1 (a)) and an edge device (Fig.1 (b)). The results reveal that existing offloading paradigms are ill-suited for edge hardware, primarily due to the following degradations:

Exacerbated I/O Stalls. Due to the memory capacity limitation in UMA-based mobile platforms, on-device MoE is characterized by intensive read operations from NVMe SSDs rather than CPU memory, whose read bandwidth (typically $1-5$ GB/s) is significantly lower than that of server-grade CPU memory ($16-32$ GB/s). Consequently, inference speed is severely bottlenecked by storage I/O. As shown in Figure 1, I/O stalls surge from 38.5% to 80.1% for decoder-only models and from 15.8% to 41.7% for encoder-decoder models when moving from server to edge settings.

Underutilized Compute Power. Although UMA theoretically accelerates in-memory operations, as evidenced by the fact that edge environment offers a $2.12\times$ speedup in host-to-device transfers in our benchmark, this benefit is

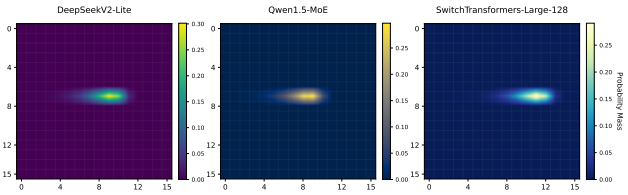

Figure 2. The probability mass heatmaps of integer representations of the exponent bits extracted from different MoE parameters. The Shannon entropy values for the three models are 2.651 bits, 2.563, and 2.554 bits, respectively.

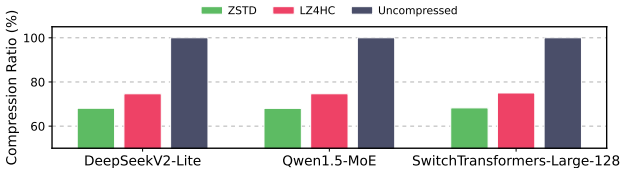

Figure 3. The compression ratios of different MoE parameters using different lossless compressors.

negated by the excessive expert I/O latency on the critical path of MoE inference. Under typical on-device workloads with batch size one, prolonged I/O stalls leave both CPU and GPU resources largely idle, resulting in severe underutilization of the available compute capacity.

These findings demonstrate a fundamental mismatch between existing MoE serving systems and the hardware characteristics of mobile SoCs.

2.2. Information-Inefficiency in MoE Parameters

The current *de facto* Brain Floating Point (BF16) representation used by modern LLMs is information-theoretically inefficient, which burdens the I/O efficiency.

BF16 Parameters. In the BF16 format, each real number is represented by a *sign* bit, 8 *exponent* bits, and 7 *mantissa* bits, and can be computed as $(-1)^{\text{sign}} \times 2^{\text{exponent}-127} \times (1.\text{mantissa})$. The well-balanced nature between numerical range (up to $\pm 3.39 \times 10^{38}$) and precision (down to 1.18×10^{-38}) has made BF16 widely adopted in both LLM training (Kalamkar et al., 2019) and inference (Kurtic et al., 2025).

Low-Entropy Exponent Bits. Prior studies have observed pronounced entropy heterogeneity across different bit-fields in BF16 parameters (Hershcovitch et al., 2025; Yubeaton et al., 2025; Hao et al., 2024; Zhang et al., 2025), where sign and mantissa bits exhibit near-random distributions, while exponent bits are highly redundant. To quantify this effect in MoE models, we analyze the expert network parameters and plot the probability mass heatmaps of the exponent-bit symbols. As shown in Fig.2, the distributions of exponent symbols are consistently skewed across all evaluated models, with the support set containing only 14.06%, 11.33%, and 14.84% of symbols, respectively. Such strong skewness directly indicates substantial statistical redundancy and high compressibility of BF16 representations from an information-theoretic perspective. We further apply two general-purpose lossless compressors, LZ4HC (Collet et al., 2024) and ZSTD (Collet et al., 2025), to MoE parameters. The normalized compressed sizes are shown in Fig. 3. LZ4HC reduces model size to 74%, while ZSTD achieves 68%, approaching the Shannon entropy lower bound (calculated using numerical results in Fig.2 as 66.02%, 65.96%, and 66.52% for the three models, respectively). These results indicate that current off-the-shelf compressors can already yield substantial size reduction.

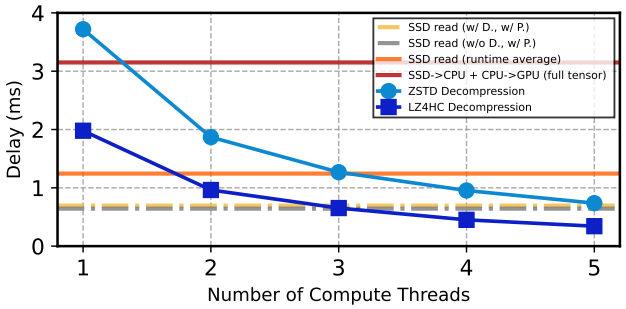

Figure 4. Comparison between decompression delay and I/O delay in various cases, where D. denotes the existence of decompression in other threads, P. denotes page cache is enabled. All read operations are tested to read the same amount of bytes with decompressed tensors (the size of exponent bits), while full tensor represents the end-to-end delay of transferring the complete tensor to GPU. All experiments are carried out on Jetson AGX Orin 64G with Samsung 970 EVO SSD.

Lossless Compression for LLM Inference. Despite its clear potential for reducing storage and I/O overhead, lossless compression remains largely under-explored in LLM inference, particularly for MoE models. Existing approaches either target specialized hardware such as FPGAs (Yubeaton et al., 2025), or rely on nvCOMP (NVIDIA, 2025), which offers little support for AArch64 CPUs prevalent in mobile and edge platforms. While DFloat11 (Zhang et al., 2025) enables on-the-fly GPU decompression for general LLMs, its computational overhead becomes prohibitive when it cannot be amortized across large batches, which is rarely encountered in on-device inference. Most notably, existing lossless compression schemes are agnostic to the conditional activation patterns of MoE models and fail to address the scalability challenges posed by inevitable parameter offloading under tunable memory budgets. This limitation is particularly critical on mobile and edge platforms, where the operating system manages multi-tenant workloads and system memory is shared across concurrent applications.

2.3. From I/O-Bound to CPU-Parallel Expert Access

The above findings demonstrate that compressing offloaded parameters is a promising approach to reduce I/O traffic. Motivated by the observation that modern mobile platforms

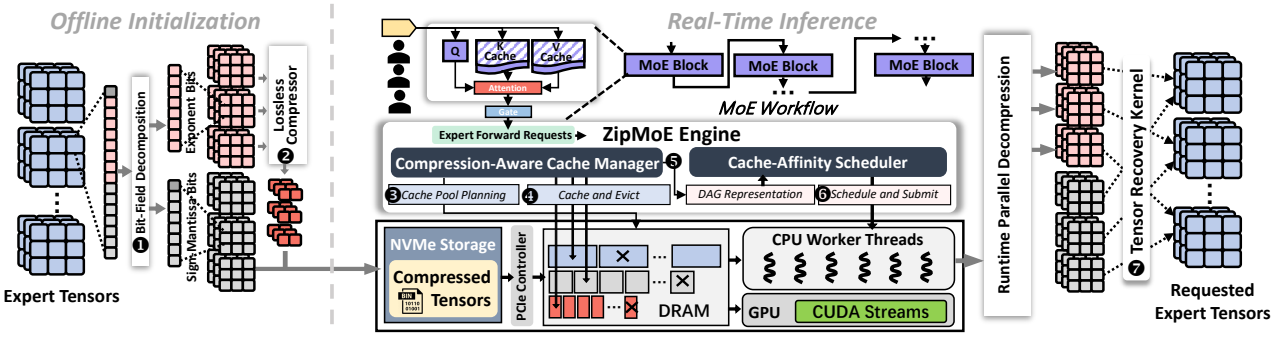

Figure 5. ZIPMOE System Overview

are equipped with multi-core CPUs, we benchmark the decompression throughput of LZ4HC and ZSTD against raw tensor I/O. Our results in Fig. 4 reveal the following insights, which motivate the design principles of ZIPMOE.

(I) Decompression is Not on the Critical Path. When reading the same number of bytes as the decompressed tensor payload, multi-threaded decompression incurs lower latency than SSD I/O once ≥ 3 worker threads are employed. This indicates that decompression can be efficiently parallelized across CPU cores. Moreover, since the exponent bits occupy the same storage footprint as the sign and mantissa bits combined, decompression latency can be effectively hidden by overlapping it with the loading these incompressible bits without introducing additional latency overhead.

(II) Resource Interference is Negligible. Since SSD reads and decompression-induced DRAM writes compete for the same memory controller bandwidth, we evaluated the impact of parallel background decompression on I/O performance. Our measurements show a modest 7.47% reduction in peak SSD read throughput. We note that this overhead is negligible in practice. As shown in Fig. 4 (orange line), the average runtime SSD throughput is significantly lower than the peak bandwidth due to system-level factors (e.g., low page-cache hit rates). Consequently, the contention at the memory controller does not become a performance bottleneck, implying that decompression can safely proceed in parallel with I/O without causing severe interference.

(III) Opportunities in Compression-Aware Caching. Recall that effective decompression overlap is achieved with as few as 3 worker threads and yields even more speedups over full tensor read. Meanwhile, mainstream mobile devices typically feature 6–12 CPU cores. This imbalance indicates that substantial CPU compute capacity remains underutilized during MoE inference. Our key insight is that, since sign and mantissa bits are largely incompressible and constitute 50% of the full tensor size, selectively caching them enables $2\times$ tensor coverage under the same memory budget. Upon a partial cache hit, redundant CPU threads can recover the compressed exponent bits in parallel, with the decompression latency fully hidden by the I/O of other cache-missed experts, thereby achieves the same zero latency as a full-tensor cache hits. Note that the same principle also applies when caching compressed exponent bits. These observations motivate a hierarchical caching design that is explicitly compression-aware, enabling I/O reduction beyond the theoretical lower bound of entropy.

3. ZipMoE Design

3.1. System Overview

Fig. 5 illustrates the architecture of ZIPMOE, which consists of two stages: *offline initialization* and *real-time inference*.

Offline Initialization. Executed once prior to deployment, this stage converts model parameters into lossless compressed representations. ZIPMOE first performs bit-field decomposition (Fig. 5 ❶) to separate tensor elements into low-entropy exponent bits and high-entropy sign–mantissa bits. The extracted exponent bits are partitioned into K shards and then compressed (denoted as `E-chunks`), while the sign–mantissa bits are packed into byte-aligned representations (denoted as `SM-chunks`). All components, together with the required metadata, are serialized into a binary format and offloaded to NVMe SSD. (Fig. 5 ❷)

Real-time Inference. To start up, the *compression-aware cache manager* determines the cache capacity allocated to tensors at different compression states, i.e. the compressed `E-chunks`, `SM-chunks`, and fully reconstructed tensors, under a given memory budget and execution configuration (Fig. 5 ❸). At runtime, once the gate network reveals the selected experts, corresponding expert requests are issued to the *cache-affinity scheduler*, which dispatches them to appropriate cache pools based on their access patterns (Fig. 5 ❹). For each request, ZIPMOE constructs a cache-aware directed acyclic graphs (DAGs) that captures the required I/O, decompression, and reconstruction operations (Fig. 5 ❺). These operations are scheduled to execute asynchronously across L parallel CPU worker threads, enabling overlapped `SM-chunk` loading and `E-chunk` decompression (Fig. 5 ❻). Finally, the decompressed tensors are reassembled into BF16 format using a memory-coalesced GPU kernel for model inference (Fig. 5 ❼).

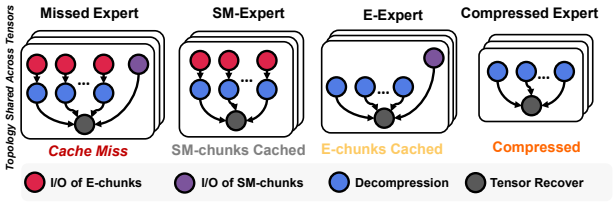

Figure 6. DAG structures of expert requests with different compression states.

3.2. Compression States

ZIPMOE achieves fine-grained memory management at runtime by allowing the `E-chunks`, `SM-chunks`, and full tensors of each expert to be cached separately. To this end, we introduce the *compression state* abstraction to track the residency of these components in memory. Specifically, an expert is defined as an `E-expert` if its `E-chunks` are cached, and an `SM-expert` if its `SM-chunks` are cached. An expert is termed as `compressed-expert` when both components reside in memory.

DAG Representation. To enable adaptive resource scheduling, each expert reconstruction task is modeled as a DAG, whose nodes represent the fine-grained operations such as decompression, SSD read, and tensor recovery, and the directed edges denote the precedence constraints. ZIPMOE dynamically constructs the DAG topology for each task based on the expert's runtime compression state. Note that the task granularity is defined at the tensor level. For an expert comprising N tensors, all tensors share the same topology and generate N independent reconstruction tasks. We show all possible DAG structures in Fig. 6.

3.3. Cache-Affinity Scheduler

Once the selected experts are revealed, a set $\mathcal{Q}$ of DAG tasks are issued to the scheduler. Let ρ denote the compression ratio of the lossless compressor used by ZIPMOE, which is defined as the size of compressed exponent bits relative to their original representation. For each task $j \in \mathcal{Q}$, we denote the I/O latency of a single `SM-chunk` as u. Accordingly, reading one compressed `E-chunk` incurs an I/O cost of $\frac{\rho}{K}u$, and its corresponding decompression cost is denoted as c. By $n(j)$ we denote the expert ID corresponding to task j. For each expert n, let p_n represent the GPU execution time required to process all tokens routed to that expert. We create a dedicated I/O thread for all SSD reads, a pool of L CPU worker threads for decompression, and a CUDA stream for tensor recovery. Our objective is to minimize the makespan of each sparse layer, i.e., the completion time of the last expert execution.

We design an efficient scheduling algorithm to orchestrate the fine-grained operations in the execution of every sparse MoE layer, aiming to maximize parallelism across heterogeneous resources while minimizing idle time caused by I/O stalls.

Task Partition. Based on their critical execution paths, we partition all expert tasks into two disjoint classes. *Type-I tasks* correspond to experts whose reconstruction requires loading `SM-chunks`, and therefore incur expensive blocking on the I/O thread. *Type-II tasks* include the remaining experts whose `SM-chunks` of the required tensors are already cached in memory. We sort Type-I and Type-II tasks independently in non-increasing order of their corresponding expert execution time $p_{n(j)}$, yielding two ordered sequences σ_{I} and σ_{II}, respectively.

Block Construction. Building on the insight that `SM-chunk` loading dominates the critical path of *Type-I* tasks, the scheduler creates blocks by grouping *Type-I* tasks and *Type-II* tasks to overlap `SM-chunk` loading and decompression. Specifically, the scheduler iteratively selects the first unscheduled task in σ_{I} as the *base* of a new block. Within each block, `E-chunks` are loaded before `SM-chunks`, and the I/O order among the same type of chunks follows the scheduling order of their corresponding tasks in the block. The scheduler then incrementally inserts tasks from the head of σ_{II} into the block at positions that do not introduce additional idle time on the decompression threads. If no such position exists, the task is placed after all *Type-II* tasks (or *Type-I* tasks, if no *Type-II* task exists) with more token requests. The insertion process terminates when either all Type-II tasks have been considered, or when the system becomes compute-bound, i.e., when the completion time of the l-th ($1 \le l \le \min\{L, K\}$) earliest decompression thread lags behind the I/O thread by at least $\frac{l\cdot\rho}{K}\cdot u$. Beyond this point, additional Type-II tasks no longer contribute to hiding I/O latency and the resulting sequence of tasks forms a block. The scheduler repeatedly applies this procedure to construct subsequent blocks until all tasks are assigned. Blocks are executed sequentially, while tasks within each block follow the constructed priority order. We provide the full pseudocode in Appendix A and prove that the resulting schedule achieves a constant-bounded approximation to the optimal makespan in Appendix B.

Theorem 3.1. *Let `ALG` be the makespan achieved by* ZIPMOE*'s scheduler and `OPT` be the optimal value. It holds that:*

$$\mathtt{ALG} \le \left(3 - \frac{1}{L}\right) \cdot \mathtt{OPT}, \tag{1}$$

where L is the number of decompression threads.

Memory-Coalesced Tensor Recovery Kernel. Reassembling exponent bits and sign-mantissa bits into BF16 tensors is inherently memory-bound, where the computation consists only of lightweight bit manipulations, while the kernel must stream a large volume of tensor elements from global memory. We optimize this process via a vectorized GPU kernel that aggregates data access to ensure memory coa-

lescing. Specifically, each CUDA thread fetches contiguous segments of the loaded `SM-chunks` and the decompressed `E-chunks` via vectorized load instructions. This design allows bit-level reconstruction to be performed entirely within registers and writes the recovered tensors back using vectorized stores. As a result, the kernel minimizes instruction overhead, saturates DRAM bandwidth, and achieves high-throughput tensor recovery that effectively overlaps with the asynchronous CPU decompression pipeline.

3.4. Compression-Aware Cache Management

To enable fine-grained memory control, ZIPMOE partitions the total available memory into a hierarchy of cache pools: the $\mathcal{F}$ pool for full tensors, the $\mathcal{C}$ pool for compressed tensors (containing both `E-chunks` and `SM-chunks`), the $\mathcal{S}$ pool dedicated to `SM-chunks`, and the $\mathcal{E}$ pool dedicated to `E-chunks`. These pools offer different latency–memory trade-offs and collectively enable flexible cache management under a fixed memory budget.

Rank-Based Workload Modeling. In MoE workloads, the specific identities of frequently activated experts often fluctuate across different prompts. To capture the inherent workload skewness while remaining agnostic to expert identities, we introduce a *rank-based abstraction*. Specifically, we aggregate historical expert activation counts for a given batch size to derive a rank-based marginal inclusion probability list $(f_r)_{r\in\mathcal{N}}$. This list characterizes the stationary distribution of expert activations with respect to their popularity ranks, independent of the particular expert IDs occupying those ranks at runtime.

Pool Dispatching. The system maintains a *runtime* frequency list to record the actual activation counts of each expert. We define an ordered cache hierarchy $\mathcal{F} \prec \mathcal{C} \prec \mathcal{S} \prec \mathcal{E}$. An expert is dispatched to the first cache pool i in the hierarchy for which its observed rank r satisfies $r < \tau_i = \sum_{j:j\preceq i} S_j + \delta$, where S_j denotes the capacity of pool j, and δ is a *tolerance margin* introduced to absorb noise in runtime statistics that may deviate from the long-term ranking implied by the workload model. In cases where the assigned cache pool overflows, experts with the lowest activation frequency are evicted. If an expert's rank exceeds all thresholds, it is treated as rarely activated and is evicted immediately after execution.

Hierarchical Cache Planning. We determine the optimal cache partition by minimizing the expected makespan of sparse MoE layers. To this end, we analytically model the cache behavior of feasible partitions using the rank-based workload abstraction and the defined cache hierarchy. Aligning with the runtime dispatching logic, we map the cache pools onto the *expert frequency ranks* sequentially. Consequently, each pool p is assigned to a *contiguous rank interval* $[u_p, v_p]$ with a size equal to its capacity S_p. Expert ranks falling outside all allocated cache pools (i.e., the tail of the rank list) are assigned to a virtual *Miss Pool*, denoted by $\mathcal{M}$. We model the expert activation process as a series of independent Bernoulli trials. Specifically, for a pool p (including $\mathcal{M}$), an expert at rank $r \in [u_p, v_p]$ is selected with probability q_r, which is derived from the marginal probability f_r (details in Appendix D). Under this formulation, the number of cache hits within any pool follows a *Poisson binomial distribution*. We employ dynamic programming (DP) to compute the probability of observing exactly h_p hits within the specific rank interval of pool p. Let $\mathrm{DP}[i][j][p]$ denote the probability of observing j hits among the first i experts within the interval assigned to pool p. The DP table transition is given by: $\mathrm{DP}[i][j][p] = \mathrm{DP}[i-1][j][p](1-q_{u_p+i}) + \mathrm{DP}[i-1][j-1][p]q_{u_p+i}$, where q_{u_p+i} denotes the probability of selecting the i-th expert in the pool's assigned interval. The probability of observing exactly h_p hits in pool p is thus given by $\Phi_p(h_p) = \mathrm{DP}[S_p][h_p][p]$. Finally, since exactly k experts are selected per token layer-wise, we derive the joint probability of a cache hit pattern $\mathbf{h} = (h_\mathcal{F}, h_\mathcal{C}, h_\mathcal{S}, h_\mathcal{E})$ as a conditional probability: $\mathbb{P}\left(\mathbf{h} \mid \sum_{p\in\Lambda\cup\{\mathcal{M}\}} h_p = k\right) = \frac{\Phi_\mathcal{M}(k_{\mathrm{rem}})}{\Phi_\mathcal{N}(k)} \cdot \prod_{p\in\Lambda}\Phi_p(h_p)$, where $\Lambda \in \{\mathcal{F}, \mathcal{C}, \mathcal{S}, \mathcal{E}\}$ is the set of activated cache pools, which can be selected flexibly by users based on specific hardware and OS conditions. $k_{\mathrm{rem}} = k - \sum_{p\in\Lambda} h_p$ is the number of hits falling into the *Miss Pool*, $\Phi_\mathcal{M}$ represents the probability distribution of the tail experts, and $\Phi_\mathcal{N}(k)$ is the probability of selecting exactly k experts from the entire rank list $\mathcal{N}$, computed from the global DP table. We prove in Appendix D that this procedure yields a maximum entropy distribution, ensuring the solution is *Bayes robust* (Grünwald & Dawid, 2004).

Theorem 3.2. *Among all probability distributions over k-sized expert subsets that are consistent with the observed individual expert selection counts, the distribution produced by the DP procedure achieves maximum entropy.*

The planning algorithm then enumerates feasible cache partition configurations via grid search. For each configuration, we compute the expected makespan by aggregating the latency costs of all feasible cache hit and miss combinations weighted by their corresponding probabilities derived above. The configuration that minimizes the expected makespan is selected for runtime cache partitioning. The complete planning algorithm is detailed in Appendix C.

4. Implementation

We prototype ZIPMOE on top of the HuggingFace Transformers (Wolf et al., 2020) framework for inference, and integrate the *lz4* and *zstd* libraries as lossless compression backends. The system frontend, profiler, and cache planning components are implemented in 2.6K lines of Python code, while the execution engine, scheduler, and memory management system comprise 8K lines of C++/CUDA code.

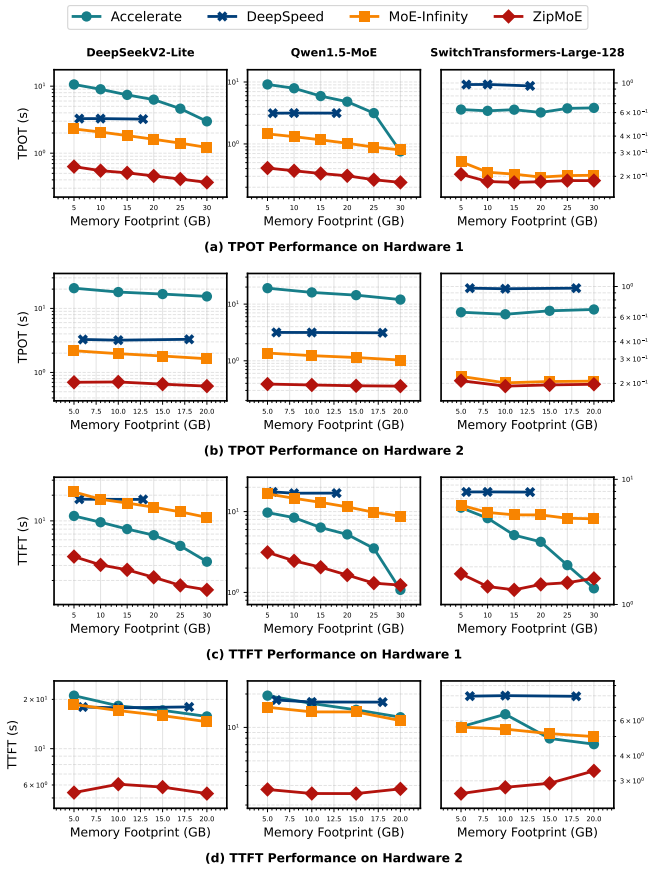

Figure 7. Comparison of TPOT and TTFT performances on diverse MoE models across different memory budgets.

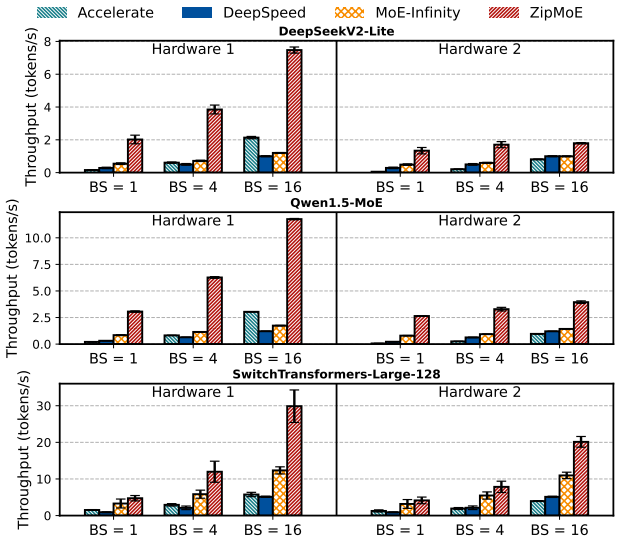

Figure 8. Comparison of system throughput on diverse MoE models under different batch sizes. BS denotes batch size.

ZIPMOE pre-allocates cache pools as contiguous memory regions to reduce memory fragmentation. To benefit from UMA, we adopt a zero-copy paradigm that reads `SM-chunks` directly into host-pinned memory registered for direct GPU access, avoiding redundant data transfers. Motivated by operating system (OS) considerations, the prototype consolidates each `E-chunk` read with its subsequent decompression into a single operation to better leverage the page cache. This implementation preserves the design principles in §3 while simplifying runtime overhead and improving affinity with the OS.

5. Evaluation

Models and Datasets. We evaluate ZIPMOE on two decoder-only models, DeepSeekV2-Lite (Liu et al., 2024) and Qwen1.5-MoE (Yang et al., 2024), as well as an encoder-decoder model, SwitchTransformers-Large-128 (Fedus et al., 2022). All models are obtained from Hugging Face without modification. We randomly sample prompts from the ShareGPT (Xu, 2024) dataset and align the inputs across all baseline systems to ensure fair evaluation.

Hardware. We adopt the Jetson AGX Orin 64GB (Hardware 1) and 32GB (Hardware 2) as our edge computing testbeds. Both devices operate on Jetpack 6.2.1 with Ubuntu 22.04, and are equipped with a Samsung 970 EVO SSD that provides a disk read speed of 3.5 GB/s.

Baselines. We compare ZIPMOE with the following state-of-the-art LLM serving systems: (1) *MoE-Infinity* (Xue et al., 2024): A high-performance MoE serving system that exploits sparsity-aware expert caching and prefetching. (2) *DeepSpeed* (Aminabadi et al., 2022): A leading inference engine that supports model offloading and parameter partitioning. We adopt DeepSpeed ZeRO-3 for SSD offloading during MoE inference. (3) *Accelerate* (Gugger et al., 2022): A widely-used library that provides simple APIs for model offloading and distributed inference. All systems are configured to operate under their default SSD offloading modes with aligned DRAM memory footprints for fair comparison.

5.1. Experimental Results

Real-Time Responsiveness. Fig. 7 shows the real-time responsiveness measured by Time-Per-Output-Token (TPOT) and Time-To-First-Token (TTFT). For decoder-only models, we observe that ZIPMOE substantially improves performance on both hardware environments, achieving 62.65%-97.97% TPOT reduction and 53.25%-87.90% TTFT reduction in scenarios where offloading is mandatory. For encoder-decoder models, although the advantage diminishes due to highly skewed expert activation and a less I/O-intensive structure, ZIPMOE still yields a 4.99%-81.24% TPOT reduction and up to an 83.45% TTFT reduction. These advantages are enabled not only by the sophisticated caching-scheduling co-design, but also by the low-overhead implementation and its effective utilization of the page cache during fixed-memory experiments, which improves both

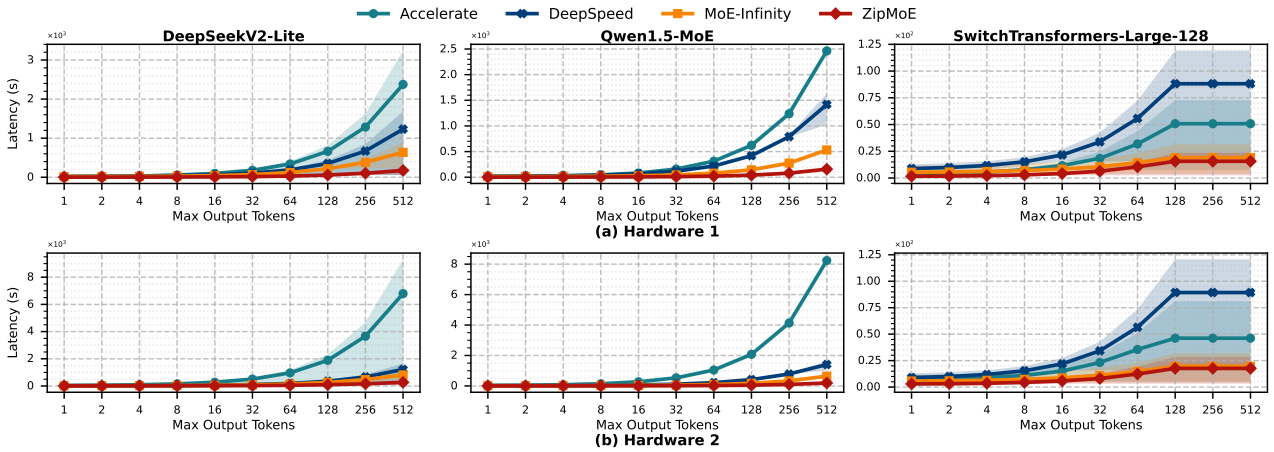

Figure 9. Comparison of end-to-end latency on diverse MoE models across different output lengths.

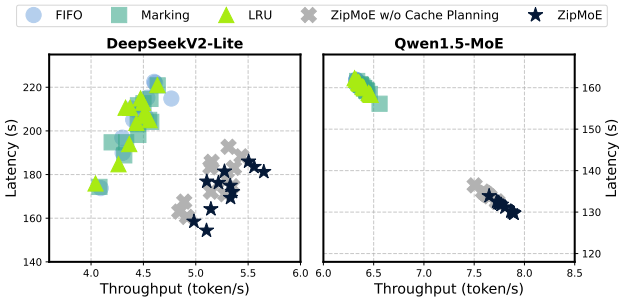

Figure 10. Impact of cache management strategies on the trade-off between latency and throughput.

memory and I/O efficiency. The results highlight the ZIPMOE's superiority in real-time and interactive tasks.

System Throughput. Fig. 8 compares the throughput of different systems in batch inference settings with batch sizes of 1, 4, and 16. The memory budget for ZIPMOE, *MoE-Infinity*, and *Accelerate* is fixed at 20GB and 10GB on Hardware 1 and Hardware 2, respectively. For *DeepSpeed*, we observe extreme discontinuity between its runtime memory footprint and configuration, therefore memory budget control is coarse-grained. We fix its runtime memory budget to 18GB and 10GB on Hardware 1 and Hardware 2, respectively. Note that since *DeepSpeed* employs a sliding-window offloading pipeline, its performance remains agnostic to the specific memory footprint provided the budget is below the model size (see Fig. 7). It can be observed that ZIPMOE consistently surpasses all baselines, achieving $1.79\times$-$42.49\times$ improvement for decoder-only models and $1.31\times$-$5.82\times$ improvement for encoder-decoder models. These advantages are attributed to ZIPMOE's scheduler, which is specifically designed for I/O efficiency via decompression parallelization. As more experts are activated per layer in batch inference, the benefits of ZIPMOE's scheduler for parallelization are amplified, thus significantly reducing I/O blocking. Overall, we conclude that ZIPMOE also excels in batch inference for offline background tasks.

End-To-End Latency. Fig. 9 presents the end-to-end latency results under different output token limits, where the lines represent the average values and the shaded areas cover between the top and bottom 10% values. Note that sublinear trends may appear in the figure, as in certain cases the number of output tokens do not actually reach the limit. It can be observed that ZIPMOE is consistently superior to all baselines across all maximum output token numbers, accelerating end-to-end inference speed by $3.03\times$-$42.49\times$ for decoder-only models and $1.11\times$-$5.64\times$ for encoder-decoder models. These results confirm ZIPMOE's versatility in providing consistent inference acceleration across diverse MoE architectures and hardware environments.

Ablation Study. Fig. 10 illustrates the impact of the cache management module on system performance in terms of the latency-throughput trade-off. We successively isolate the hierarchical cache planning algorithm and replace the eviction strategy of *ZipMoE* with First-In-First-Out (FIFO), Marking(Fiat et al., 1991), and Least Recently Used (LRU). All experiments are conducted under 10GB memory footprint and batch size=16. It can be observed that while all baseline caching algorithms perform similarly, ZIPMOE's built-in caching algorithm outperforms all of them. Furthermore, with cache planning that optimizes the cache pools prior to inference, ZIPMOE's performance is further boosted. To further quantify the contribution of each component, we report a detailed breakdown of performance gains in Tab. 1. Here, *Avg. basic* represents the average performance of LRU, FIFO, and Marking, *C* denotes ZIPMOE's heterogeneous cache pool (without planning), and *P* denotes cache pool planning. The results indicate that the core system accounts for the majority of the throughput improvement, constituting approximately 76% of the total gain. This leap is primarily driven by the paradigm shift enabled by ZIPMOE's lossless decompression engine

Table 1. Quantitative breakdown of ZIPMOE's performance gains across different cache management strategies.

(a) Model: DeepSeekV2-Lite 16B

Method	**Throughput** (tokens/s)	**ΔThroughput** (tokens/s)	**E2E** (s)	**ΔE2E** (s)
Baseline	1.60	-	585.96	-
ZipMoE (avg. basic)	4.43	+2.83	204.05	-381.91
ZipMoE (+C)	5.18	+0.75	176.68	-27.37
ZipMoE (+C+P)	5.30	+0.12	173.23	-3.45
Total gain	+3.70	-	-412.73	-

(b) Model: Qwen1.5-MoE 14B

Method	**Throughput** (tokens/s)	**ΔThroughput** (tokens/s)	**E2E** (s)	**ΔE2E** (s)
Baseline	1.99	-	515.12	-
ZipMoE (avg. basic)	6.39	+4.40	160.33	-354.79
ZipMoE (+C)	7.64	+1.25	134.10	-26.23
ZipMoE (+C+P)	7.79	+0.15	131.47	-2.63
Total gain	+5.80	-	-383.65	-

Table 2. Isolation of page cache gain. *Reduction* denotes the percentage of reduced latency. *Retention* denotes the percentage of performance advantage over the baseline (MoE-Infinity) that is preserved as the page cache is reduced.

Occupied RAM	**Time-To-First-Token**			**Time-Per-Output-Token**		
	Value	**Reduction**	**Retention**	**Value**	**Reduction**	**Retention**
0 GB	2.18 s	84.94%	100.0%	0.46 s	71.64%	100.0%
20 GB	2.43 s	83.21%	98.0%	0.48 s	70.46%	98.4%
24 GB	5.62 s	61.17%	72.0%	0.54 s	66.41%	92.7%
28 GB	5.22 s	63.90%	75.2%	0.61 s	62.18%	86.8%
32 GB	6.27 s	56.64%	66.7%	0.75 s	53.32%	74.4%
Baseline	14.46 s	-	-	1.61 s	-	-

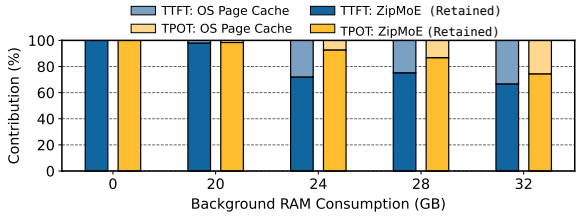

Figure 11. Breakdown of contributions under different background RAM pressures.

and CPU-parallel decompression scheduling. It is worth noting that these gains stem not only from the decompression engine itself, but also from optimized system-level behaviors, including improved utilization of the OS page cache, more efficient GPU memory management, and refined low-level optimizations in the inference framework. Our proposed cache management approach contributes the remaining approximately 24% of the throughput improvement. Compared with using heterogeneous cache pools alone, the cache planning algorithm provides a moderate performance gain while achieving lower latency and higher throughput in a Pareto-optimal manner.

Isolating OS Page Cache. ZIPMOE elastically utilizes the OS page cache whenever additional RAM is available and not occupied by background processes from co-located applications. To investigate the performance contribution of OS page cache utilization, we artificially inject background memory consumption during inference to progressively reduce the available RAM for page caching. The results are presented in Tab. 2 and Fig. 11. Since the OS page cache is elastic, increasing background memory pressure effectively reduces the cacheable memory space, forcing the system to rely primarily on ZIPMOE's internal caching and decompression mechanisms. The results show that even when background RAM consumption reaches 32 GB, leaving negligible space for the OS page cache, ZIPMOE still achieves a 56.64% reduction in TTFT and a 53.32% reduction in TPOT compared with the baseline, retaining approximately 66.7%–74.4% of its original performance advantage. The results confirm that the majority of the performance gains stem from the core ZIPMOE system engine, while the OS page cache provides an opportunistic performance enhancement when additional memory resources are available.

6. Discussion

Hardware Generalizability. Although the current evaluation of ZIPMOE is conducted on NVIDIA Jetson AGX Orin, the overall system design is intentionally hardware-agnostic and applicable to any shared-memory platforms. For systems in which the CPU and GPU (or other accelerators) maintain dedicated physical memory, our approach can be extended by treating CPU memory as an additional cache tier and introducing asynchronous host-to-device transfers to support CPU-GPU hybrid offloading and inference. We consider extending ZIPMOE to such heterogeneous-memory systems an important direction for future work.

Other Compressors. Our current implementation supports multiple compression backends, including LZ4, LZ4HC, and ZSTD. Since the effectiveness of a compressor depends on both hardware characteristics and implementation efficiency, our codebase provides APIs that allow future integration of customized or hardware-specific compressors for further performance optimization.

7. Conclusion

We proposed ZIPMOE, an efficient MoE inference system tailored for mobile and edge computing platforms. ZIPMOE improves on-device memory and I/O efficiency through fine-grained orchestration of sparse layer execution via a caching-scheduling co-design. Our testbed evaluation of ZIPMOE on diverse edge platforms demonstrates significant improvements in both inference latency and system throughput while preserving identical model behavior.

Acknowledgements

The authors would like to thank the anonymous reviewers, whose invaluable comments helped improve the presentation of this paper substantially. This work was partially supported by the Jiangsu Science Foundation Leading-edge Technology Program (BK20232003).

Impact Statement

This paper presents work whose goal is to advance the field of machine learning. There are many potential societal consequences of our work, none of which we feel must be specifically highlighted here.

References

Aminabadi, R. Y., Rajbhandari, S., Awan, A. A., Li, C., Li, D., Zheng, E., Ruwase, O., Smith, S., Zhang, M., Rasley, J., et al. DeepSpeed-Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15. IEEE, 2022.

Cao, S., Liu, S., Griggs, T., Schafhalter, P., Liu, X., Sheng, Y., Gonzalez, J. E., Zaharia, M., and Stoica, I. 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*, pp. 715–730, 2025.

Chen, H., Xie, W., Zhang, B., Tang, J., Wang, J., Dong, J., Chen, S., Yuan, Z., Lin, C., Qiu, C., et al. Ktransformers: Unleashing the Full Potential of CPU/GPU Hybrid Inference for MoE Models. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, pp. 1014–1029, 2025.

Chen, X.-H., Dempster, A. P., and Liu, J. S. Weighted Finite Population Sampling to Maximize Entropy. *Biometrika*, 81(3):457–469, 1994. ISSN 00063444. URL http://www.jstor.org/stable/2337119.

Collet, Y. et al. LZ4: Extremely Fast Compression Algorithm. https://github.com/lz4/lz4, 2024.

Collet, Y. et al. Zstandard. https://github.com/facebook/zstd, 2025.

Davies, S., Kulkarni, J., Rothvoss, T., Tarnawski, J., and Zhang, Y. Scheduling with Communication Delays via LP Hierarchies and Clustering. In *Proc. IEEE FOCS'20*, Durham, NC, USA, Nov. 2020.

Du, Z., Li, S., Wu, Y., Jiang, X., Sun, J., Zheng, Q., Wu, Y., Li, A., Li, H., and Chen, Y. SiDA: Sparsity-Inspired Data-Aware Serving For Efficient and Scalable Large Mixture-of-Experts Models. *Proceedings of Machine Learning and Systems*, 6:224–238, 2024.

Duanmu, H., Li, X., Yuan, Z., Zheng, S., Duan, J., Zhang, X., and Lin, D. MxMoE: Mixed-precision Quantization for MoE with Accuracy and Performance Co-Design. In *Forty-second International Conference on Machine Learning*, 2025.

Egashira, K., Vero, M., Staab, R., He, J., and Vechev, M. Exploiting LLM Quantization. *Advances in Neural Information Processing Systems*, 37:41709–41732, 2024.

Eliseev, A. and Mazur, D. Fast Inference of Mixture-of-Experts Language Models with Offloading. *arXiv preprint arXiv:2312.17238*, 2023.

Fang, Z., Huang, Y., Hong, Z., Lyu, Y., Chen, W., Yu, Y., Yu, F., and Zheng, Z. 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*, pp. 574–588, 2025.

Fedus, W., Zoph, B., and Shazeer, N. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.

Fiat, A., Karp, R. M., Luby, M., McGeoch, L. A., Sleator, D. D., and Young, N. E. Competitive Paging Algorithms. *Journal of Algorithms*, 12(4):685–699, 1991.

Frantar, E., Ashkboos, S., Hoefler, T., and Alistarh, D. GPTQ: Accurate Post-training Compression for Generative Pretrained Transformers. *arXiv preprint arXiv:2210.17323*, 2022.

Fu, M., Yu, H., Shao, J., Zhou, J., Zhu, K., and Wu, J. Quantization without Tears. In *Proceedings of the Computer Vision and Pattern Recognition Conference (CVPR)*, pp. 4462–4472, June 2025.

Grünwald, P. D. and Dawid, A. P. Game Theory, Maximum Entropy, Minimum Discrepancy and Robust Bayesian Decision Theory. *The Annals of Statistics*, 32(4):1367–1433, 2004. ISSN 00905364.

Gugger, S., Debut, L., Wolf, T., Schmid, P., Mueller, Z., Mangrulkar, S., Sun, M., and Bossan, B. Accelerate: Training and Inference at Scale Made Simple, Efficient and Adaptable. https://github.com/huggingface/accelerate, 2022.

Hao, Y., Cao, Y., and Mou, L. NeuZip: Memory-Efficient Training and Inference with Dynamic Compression of Neural Networks. *arXiv preprint arXiv:2410.20650*, 2024.

Hershcovitch, M., Wood, A., Choshen, L., Girmonsky, G., Leibovitz, R., Ozeri, O., Ennmouri, I., Malka, M., Chin, P., Sundararaman, S., et al. ZipNN: Lossless Compression for AI Models. In *2025 IEEE 18th International Conference on Cloud Computing (CLOUD)*, pp. 186–198. IEEE, 2025.

Jacobs, R. A., Jordan, M. I., Nowlan, S. J., and Hinton, G. E. Adaptive Mixtures of Local Experts. *Neural Computation*, 3(1):79–87, 1991.

Kalamkar, D., Mudigere, D., Mellempudi, N., Das, D., Banerjee, K., Avancha, S., Vooturi, D. T., Jammalamadaka, N., Huang, J., Yuen, H., et al. A Study of BFLOAT16 for Deep Learning Training. *arXiv preprint arXiv:1905.12322*, 2019.

Kamahori, K., Tang, T., Gu, Y., Zhu, K., and Kasikci, B. Fiddler: CPU-GPU Orchestration for Fast Inference of Mixture-of-Experts Models. *arXiv preprint arXiv:2402.07033*, 2024.

Kong, R., Li, Y., Feng, Q., Wang, W., Ye, X., Ouyang, Y., Kong, L., and Liu, Y. SwapMoE: Serving Off-the-Shelf MoE-based Large Language Models with Tunable Memory Budget. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 6710–6720, 2024.

Kumar, D., Kumar, A., Agarwal, S., and Harshangi, P. Fine-Tuning, Quantization, and LLMs: Navigating Unintended Outcomes. *arXiv preprint arXiv:2404.04392*, 2024.

Kurtic, E., Marques, A. N., Pandit, S., Kurtz, M., and Alistarh, D. "Give Me BF16 or Give Me Death"? Accuracy-Performance Trade-Offs in LLM Quantization. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 26872–26886, 2025.

Liu, A., Feng, B., Wang, B., Wang, B., Liu, B., Zhao, C., Dengr, C., Ruan, C., Dai, D., Guo, D., et al. Deepseek-v2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model. *arXiv preprint arXiv:2405.04434*, 2024.

Liu, R., Sun, Y., Zhang, M., Bai, H., Yu, X., Yu, T., Yuan, C., and Hou, L. Quantization Hurts Reasoning? An Empirical Study on Quantized Reasoning Models. *arXiv preprint arXiv:2504.04823*, 2025.

Ma, X., Fang, G., and Wang, X. LLM-Pruner: On the Structural Pruning of Large Language Models. *Advances in Neural Information Processing Systems*, 36:21702–21720, 2023.

Maiti, B., Rajaraman, R., Stalfa, D., Svitkina, Z., and Vijayaraghavan, A. Scheduling Precedence-Constrained Jobs on Related Machines with Communication Delay. In *Proc. IEEE FOCS'20*, Durham, NC, USA, Nov. 2020.

Neumann, A. T., Yin, Y., Sowe, S., Decker, S., and Jarke, M. An LLM-Driven Chatbot in Higher Education for Databases and Information Systems. *IEEE Transactions on Education*, 68(1):103–116, 2025. doi: 10.1109/TE.2024.3467912.

NVIDIA. nvCOMP: GPU-accelerated compression and decompression library. https://developer.nvidia.com/nvcomp, 2025.

Qu, G., Chen, Q., Wei, W., Lin, Z., Chen, X., and Huang, K. Mobile Edge Intelligence for Large Language Models: A Contemporary Survey. *IEEE Communications Surveys & Tutorials*, 27(6):3820–3860, 2025. doi: 10.1109/COMST.2025.3527641.

Sheng, Y., Zheng, L., Yuan, B., Li, Z., Ryabinin, M., Chen, B., Liang, P., Ré, C., Stoica, I., and Zhang, C. FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU. In *International Conference on Machine Learning*, pp. 31094–31116. PMLR, 2023.

Shu, M., Wang, J., Zhu, C., Geiping, J., Xiao, C., and Goldstein, T. On the Exploitability of Instruction Tuning. *Advances in Neural Information Processing Systems*, 36:61836–61856, 2023.

Song, X., Zhong, Z., Chen, R., and Chen, H. ProMoE: Fast MoE-based LLM Serving using Proactive Caching. *arXiv preprint arXiv:2410.22134*, 2024.

Sun, J., Zhang, Q., Duan, Y., Jiang, X., Cheng, C., and Xu, R. Prompt, Plan, Perform: LLM-based Humanoid Control via Quantized Imitation Learning. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 16236–16242, 2024. doi: 10.1109/ICRA57147.2024.10610948.

Wang, H., Zhou, Q., Hong, Z., and Guo, S. D2MoE: Dual routing and dynamic scheduling for efficient on-device MoE-based LLM serving. In *Proceedings of the 31st Annual International Conference on Mobile Computing and Networking*, pp. 574–588, 2025.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R.,

Funtowicz, M., et al. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 38–45. Association for Computational Linguistics, Oct 2020. URL https://www.aclweb.org/anthology/2020.emnlp-demos.6.

Xu, M. ShareGPT-GPT4. https://huggingface.co/datasets/shibing624/sharegpt_gpt4, 2024. Hugging Face Dataset.

Xue, L., Fu, Y., Lu, Z., Mai, L., and Marina, M. MoE-Infinity: Efficient MoE Inference on Personal Machines with Sparsity-Aware Expert Cache. *arXiv preprint arXiv:2401.14361*, 2024.

Yang, A., Yang, B., Hui, B., Zheng, B., Yu, B., Zhou, C., Li, C., Li, C., Liu, D., Huang, F., Dong, G., Wei, H., Lin, H., Tang, J., Wang, J., Yang, J., Tu, J., Zhang, J., Ma, J., Xu, J., Zhou, J., Bai, J., He, J., Lin, J., Dang, K., Lu, K., Chen, K., Yang, K., Li, M., Xue, M., Ni, N., Zhang, P., Wang, P., Peng, R., Men, R., Gao, R., Lin, R., Wang, S., Bai, S., Tan, S., Zhu, T., Li, T., Liu, T., Ge, W., Deng, X., Zhou, X., Ren, X., Zhang, X., Wei, X., Ren, X., Fan, Y., Yao, Y., Zhang, Y., Wan, Y., Chu, Y., Liu, Y., Cui, Z., Zhang, Z., and Fan, Z. Qwen2 Technical Report. *arXiv preprint arXiv:2407.10671*, 2024.

Yang, P. and Wang, S. Parallelism Flattener: DAG Encoding-Driven Dependency-Aware Offloading in Edge Clouds. In *Proc. IEEE Vehicular Technology Conference (VTC'25-Fall)*, Chengdu, China, Oct. 2025.

Yang, Y. and Wang, S. EdgeOPT: A Competitive Algorithm for Online Parallel Task Scheduling with Latency Guarantee in Mobile Edge Computing. *IEEE Trans. Commun.*, 72(11):7077 – 7092, Nov. 2024a.

Yang, Y. and Wang, S. Online scheduling and splittable routing for serverless functions at network edge. In *Proc. IEEE GLOBECOM'24*, Cape Town, South Africa, Dec. 2024b.

Yi, R., Guo, L., Wei, S., Zhou, A., Wang, S., and Xu, M. EdgeMoE: Empowering Sparse Large Language Models on Mobile Devices. *IEEE Transactions on Mobile Computing*, 24(8):7059–7073, 2025. doi: 10.1109/TMC.2025.3546466.

Yu, H., Cui, X., Zhang, H., Wang, H., and Wang, H. Taming Latency-Memory Trade-Off in MoE-Based LLM Serving via Fine-Grained Expert Offloading. In *Proceedings of the 21st European Conference on Computer Systems*, pp. 176–191, 2026.

Yubeaton, P., Mahmoud, T., Naga, S., Taheri, P., Xia, T., George, A., Khalil, Y., Zhang, S. Q., Joshi, S., Hegde, C., et al. Huff-LLM : End-to-End Lossless Compression for Efficient LLM Inference. *arXiv preprint arXiv:2502.00922*, 2025.

Zhang, M.-L. and Zhou, Z.-H. Exploiting Unlabeled Data to Enhance Ensemble Diversity. *Data Mining and Knowledge Discovery*, 26(1):98–129, 2013.

Zhang, T., Hariri, M., Zhong, S., Chaudhary, V., Sui, Y., Hu, X., and Shrivastava, A. 70% Size, 100% Accuracy: Lossless LLM Compression for Efficient GPU Inference via Dynamic-Length Float (DFloat11). In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=xdNAVP7TGy.

Zhou, Y., Zhang, J., Wang, Y., Xie, Z., Chen, K., Shou, L., et al. FloE: On-the-Fly MoE Inference on Memory-constrained GPU. In *Forty-second International Conference on Machine Learning*, 2025.

Zhou, Z.-H. *Ensemble Methods: Foundations and Algorithms*. Chapman and Hall/CRC, 2nd edition, 2025.

A. ZIPMOE's Scheduler

In this section, we describe ZIPMOE's scheduler in detail. Using offline-profiled decompression and I/O delays, the scheduler constructs blocks via *Algorithm* 1. Then the compute workers and the I/O thread fetch the first available operations in a work-conserving manner, following the priority order of the constructed blocks.

Algorithm 1 Cache-Affinity Scheduling

Input: DAG Information in request set $\mathcal{Q}$.
Output: An ordered set $\mathcal{B}$ containing the constructed blocks.
Partition $\mathcal{Q}$ into *Type-I* tasks $\mathcal{Q}_{\text{I}}$ and *Type-II* tasks $\mathcal{Q}_{\text{II}}$
$\sigma_{\text{I}} \leftarrow$ Sort tasks $i \in \mathcal{Q}_{\text{I}}$ in non-increasing order of $p_n(i)$, and group tasks from the same expert consecutively.
$\sigma_{\text{II}} \leftarrow$ Sort tasks $j \in \mathcal{Q}_{\text{II}}$ in non-increasing order of $p_n(j)$, and group tasks from the same expert consecutively.
Initialize $\mathcal{B} \leftarrow \emptyset$
while σ_{I} is **not** empty **do**
    Create ordered list $U \leftarrow (\sigma_{II}, \sigma_I)$
    Create new block $B \leftarrow \emptyset$
    Append the task in σ_I with the highest priority to B
    while B is **not** compute-dominant **do**
        $j^* \leftarrow$ task in U with the highest priority
        Attempt to insert j^* at the earliest position in B that introduces no additional idle period on any thread
        if no such position exists **then**
            if there exists a type-II job **then**
                Append j^* to the tail of any *Type-II* job i^* with $p_{i^*} \geq p_{j^*}$
            else
                Append j^* to the tail of some *Type-I* job i^* with $p_{i^*} \geq p_{j^*}$
            end if
        end if
        Remove $\{j^*\}$: $U \leftarrow U \setminus \{j^*\}$
        if $U = \emptyset$ **then**
            break
        end if
    end while
    Append B to the back of $\mathcal{B}$: $\mathcal{B} \leftarrow (\mathcal{B}, B)$
end while
return Ordered set $\mathcal{B} = (B_1, B_2, \cdots, B_M)$

In *Algorithm* 1, the compute-bound property is formally defined as follows.

Definition A.1 (Compute-Bound)**.** Let $\mathcal{L}$ be the set of compute threads with $|\mathcal{L}| = L$. For each block $B_i, i \in [M]$, let $F_{I/O}(B_i)$ be the completion time of its I/O thread, and let $F_c^{(l)}(B_i)$ be the completion time of its l-th compute thread. A block is compute-dominant if it holds for all $l \in \mathcal{L}$ that:

$$F_c^{(l)}(B_i) - F_{I/O}(B_i) \geq k \cdot \frac{\rho}{K} \cdot u, 0 \leq k \leq \min\{L, K\}. \tag{2}$$

B. Theoretical Analysis of ZIPMOE's Scheduling Policy

In this section, we prove the approximation ratio of our proposed scheduling algorithm. Our objective is to minimize the completion time of each sparse layer. This can be modeled using the concept of *makespan*, as formally defined in classic scheduling literature (Yang & Wang, 2025; Maiti et al., 2020; Davies et al., 2020). We start our theoretical analysis by defining some necessary notions.

Definition B.1 (Charge)**.** For any schedule A, let $\delta_{v,j}(A)$ denote the idle time interval between the end of the last active operation on the thread processing chunk v of job j and the start of its decompression. We define $\tilde{c}_{v,j}(A) := c_{v,j} + \delta_{v,j}(A)$ to be the charged time of operation v of job j. Further, the total charged time of any algorithm A is:

$$\Delta(A) = \sum_{j \in \mathcal{Q}} \sum_{v \in D_j} \delta_{v,j}(A), \tag{3}$$

where D_j is the set of decompression operations of job j.

Remark. Intuitively, the charged time of any decompression operation is the idle time before execution on its compute thread, which is caused by I/O blocking of its required (and possibly other tasks with higher priorities within the block) `E-chunks`.

Definition B.2 (Critical-Path)**.** Let $\tau_j \in \{\mathtt{M}, \mathtt{C}, \mathtt{S}, \mathtt{E}\}$ be the compression state (expert-type) of task j, representing missed expert, compressed expert, `SM-expert`, and `E-expert`, respectively. For any task $j \in \mathcal{Q}$, the critical path z_j is the least time required to finish all the workloads following the precedence constraints, that is

$$z_j = \rho u \cdot \mathbb{1}_{\tau_j \in \{\mathtt{M},\mathtt{S}\}} + \max\left\{ \frac{K \cdot c}{\min\{K, L\}}, u \cdot \mathbb{1}_{\tau_j \in \{\mathtt{M},\mathtt{E}\}} \right\} + p_j. \tag{4}$$

We can first provide the lower-bound for the optimal solution.

Lemma B.3. *Define v_j as the I/O workload of task $j \in \mathcal{Q}$, i.e., $v_j = (1+\rho)u\mathbb{1}_{\tau_j \in M} + u\mathbb{1}_{\tau_j \in E} + \rho u \mathbb{1}_{\tau_j \in S}$. Let $I := \sum_{j \in \mathcal{Q}} v_j$ be the total I/O workload, $C := \sum_{j \in \mathcal{Q}} K \cdot c$ be the total compute workload, $P := \sum_{j \in \mathcal{Q}} \frac{p_{n(j)}}{|\{i : n(i) = n(j)\}|}$ be the total token workload, and $Z := \max_{j \in \mathcal{Q}} \{z_j\}$ be the longest single-job critical path. We have*

$$\mathtt{OPT} \geq \max\left\{ I, \frac{C}{L}, P, Z \right\}. \tag{5}$$

Proof. The lemma is trivial, since each of the lower bounds is in the critical path of the set of requests, any of them must be smaller than `OPT`. □

Corollary B.4. *We also have $\mathtt{OPT} \geq \frac{C + \Delta(\mathtt{OPT})}{L}$.*

Next, we prove that the total charged time in our proposed algorithm (`ALG`) lower-bounds the optimal solution (`OPT`).

Lemma B.5. *The total charge of the optimal solution cannot be lower than Algorithm 1. It holds that:*

$$\Delta(\mathtt{ALG}) \leq \Delta(\mathtt{OPT}). \tag{6}$$

Proof. We introduce an auxiliary schedule A'. In A', the I/O thread prioritizes loading the compressed `E-chunks` required for decompression above all else, and the compute thread pool operates in a work-conserving manner, executing any ready decompression task immediately. Note that in our setting, decompression bubbles (idle periods on worker threads) are only caused by the unavailability of `E-chunks` due to I/O latency. Since A' minimizes the waiting time for these dependencies by prioritizing their I/O, it represents a theoretical lower bound on idle time. Thus, $\Delta(A') \leq \Delta(\mathtt{OPT})$.

Now, we compare $\Delta(\mathtt{ALG})$ with $\Delta(A')$. Let $\mathcal{B} = \{B_1, B_2, \ldots, B_M\}$ be the ordered set of blocks constructed by `ALG`. According to the block Construction policy, a block is finalized and pushed to $\mathcal{B}$ only when it becomes *compute-bound* (i.e., the estimated decompression completion lags sufficiently behind I/O) or when no tasks remain in $\sigma_{\mathrm{I}} \cup \sigma_{\mathrm{II}}$.

Consider the final block B_M. We distinguish two cases:

CASE I: There is no bubble in block B_M.

In this case, we claim that B_m must be compute-dominant for every $m < M$. Since if it is not the case, there has to be an $m < M$ that is not compute-dominant, and by the design of the algorithm we have to add remaining tasks to this block until either it becomes compute-dominant, or no task remains, which means that $m = M$. Both cases contradict to the claim. Therefore, since every $B_m, m \in [M-1]$ is compute-dominant, the bubbles (if there are any) can only exist within the blocks. In that case, it must hold that $c < \rho u$, since otherwise no bubble can be produced as the I/O of K exponent chunks must be able to cover a single compute operation. If $c < \rho u$, by Definition A.1, blocks will never be compute-bound, thus we only have $M = 1$. Therefore, it holds that either no bubble exists for all $B_m, m \in [M-1]$, or $M = 1$. In both cases, $\Delta(\texttt{ALG}) = 0 \leq \Delta(\texttt{OPT})$.

CASE II: There exist some bubbles in block B_M.

In this case, we claim that $M = 1$ and $\Delta(\texttt{ALG}) \leq \Delta(A)$ for every feasible schedule A. To show that our claim holds, note that when bubbles exist within the block, it can only be that the compute cannot cover the I/O of K exponent chunks: $c < \rho u$. In this case, B_{M-1} cannot be compute-bound and by the design of our algorithm (*Definition* A.1), jobs in B_{M-1} must be in the same block with B_M. By the same reasoning, we conclude that all jobs fall within the same block, so $M = 1$. When $M = 1$, since our I/O thread always loads the exponent bits first and the compute thread works in work-conserving manner. Thus, ALG has the same bubbles as in A', we have $\Delta(\texttt{ALG}) = \Delta(A') \leq \Delta(A)$ for every feasible schedule A.

To conclude, we have $\Delta(\texttt{ALG}) \leq \Delta(A') \leq \Delta(\texttt{OPT})$, which proves the lemma.

□

Once a schedule is determined, the start time and completion time of any job in its *Gantt* is fixed. However, we can move the time intervals of decompression operations across threads provided that the target thread's idle period is enough. This gives chances to reduce the charged time for each individual operation by manipulating the assigned threads of the decompression. Let each feasible configuration of decompression thread assignment in the *Gantt* chart corresponds to a charging scheme, we have the next lemma to upper-bound the charged times for each individual operation.

Lemma B.6. *In ALG, there exists a charging such that $\delta_{v,j}(\texttt{ALG}) \leq \rho u$ for every $j \in \mathcal{Q}$ and $v \in D_j$.*

Proof. Let the start time of operation v of job j be denoted by $s_{v,j}$. Since in ALG, the latest $s_{v,j}$ should wait for the completion of the last exponent chunk, which takes $K \cdot \frac{\rho}{K} \cdot u = \rho u$ time. If $\delta_{v,j} > 0$, there must exist a thread that completes the last decompression before the completion of this I/O. For this thread, there may exist a preceding compute operation that overlaps with the I/O of the eponent chunks of j by $X_{v,j} > 0$ amount of time. So the charged workload can be constructed by $\rho u - X_{v,j} > 0$, which is no larger than ρu. □

Theorem B.7 (Restated)**.** *Let ALG be the makespan achieved by* ZipMoE*'s scheduler and OPT be the optimal value. It holds that:*

$$\texttt{ALG} \leq \left(3 - \frac{1}{L}\right) \cdot \texttt{OPT}, \tag{7}$$

where L is the number of decompression threads.

Proof. Suppose ALG divides the jobs into M blocks: $\mathcal{B} = \{B_1, B_2, ..., B_M\}$. We define the following quantities.

- $F_I(k) = \sum_{i=1}^{k} \sum_{j \in B_i} v_j =: I(B_{1\to k})$: The completion time of all the I/O operations in blocks $B_1, B_2, ..., B_k$.
- $F_C(k)$: The completion time of all the compute operations in blocks $B_1, B_2, ..., B_k$. Define its sum compute workload as $C(B_{1\to k})$.
- $\tilde{C}(B_{1\to k}) := \sum_{i=1}^{k} \sum_{j \in B_i} \sum_{v \in D_j} \tilde{c}_{v,j}\,(\texttt{ALG}) = C(B_{1\to k}) + \Delta\,(\texttt{ALG})$: The total charged compute workload of in blocks $B_1, B_2, ..., B_k$ under algorithm ALG.
- $P(B_k)$: The sum processing time of all the token executions in block B_k. Let $P(B_{k\to m}) := \sum_{i=k}^{m} P(B_k)$.

For any subset of requests Q, the latest ready time for the token executions in the subset is the maximum of the time when all I/O operations are ready and the time when all computations are ready. Therefore, the completion time of Q can be upper-bounded by:

$$\texttt{ALG} \leq \max_{1\leq k\leq M} \left\{\max\left\{F_I(k), F_C(k)\right\} + P(B_{k\to M})\right\}. \tag{8}$$

We first argue that the compute completion time $F_C(k)$ is upper-bounded by:

$$F_C(k) \leq \texttt{OPT} + \left(1-\frac{1}{L}\right) \max_{j\in B_{1\to k}, v\in D_j} \tilde{c}_{v,j}. \tag{9}$$

To show this, we fix the constant k and consider the last operation v^* finished by ALG, let l^* be the threads that processes v^*, and let $\mathcal{L}_{j^*}$ be the set of active threads (waiting for the charged time is also considered active) at a time arbitrarily shortly before the start of v^*. Let $v \to l$ denote operation v is assigned to thread l. Note that since our schedule is work-conserving, we have:

$$\sum_{j\in\sum_{j\in B_{1\to k}}} \sum_{v\in D_j: v\to l^*} \tilde{c}_{v,j} \leq \tilde{c}_{v^*,j} + \sum_{j\in\sum_{j\in B_{1\to k}}} \sum_{v\in D_j: v\to l} \tilde{c}_{v,j}, \forall l \in \mathcal{L}_{j^*}, \tag{10}$$

because otherwise we would have assigned the operation to an earlier ready thread. Therefore, it holds that:

$$\begin{aligned}
\tilde{C}(B_{1\to k}) &= \left(\sum_{j\in B_{1\to k}} \sum_{v\in D_j: v\to l_1} \tilde{c}_{v,j}\right) + \left(\sum_{j\in B_{1\to k}} \sum_{v\in D_j: v\to l_2} \tilde{c}_{v,j}\right) + \cdots + \left(\sum_{j\in B_{1\to k}} \sum_{v\in D_j: v\to l^*} \tilde{c}_{v,j}\right) - \tilde{c}_{v^*,j} + \tilde{c}_{v^*,j} \\
&\geq \sum_{l\in\mathcal{L}\setminus\mathcal{L}_{j^*}} \sum_{j\in B_{1\to k}} \sum_{v\in D_j: v\to l} \tilde{c}_{v,j} + |\mathcal{L}_{j^*}| \left(F_C(k) - \tilde{c}_{v^*,j}\right) + \tilde{c}_{v^*,j} \\
&= \sum_{l\in\mathcal{L}\setminus\mathcal{L}_{j^*}} \sum_{j\in B_{1\to k}} \sum_{v\in D_j: v\to l} \tilde{c}_{v,j} + |\mathcal{L}_{j^*}| F_C(k) - \left(|\mathcal{L}_{j^*}| - 1\right) \tilde{c}_{v^*,j},
\end{aligned} \tag{11}$$

where the inequality follows from Eq. (10). Rearranging, we have

$$\begin{aligned}
F_C(k) &\leq \frac{1}{|\mathcal{L}_{j^*}|} \sum_{l\in\mathcal{L}_{j^*}} \sum_{j\in B_{1\to k}} \sum_{v\in D_j: v\to l} \tilde{c}_{v,j} + \left(1-\frac{1}{|\mathcal{L}_{j^*}|}\right) \tilde{c}_{v^*,j} \\
&\leq \frac{1}{|\mathcal{L}_{j^*}|} \sum_{l\in\mathcal{L}_{j^*}} \sum_{j\in B_{1\to k}} \sum_{v\in D_j: v\to l} \tilde{c}_{v,j} + \left(1-\frac{1}{L}\right) \tilde{c}_{v^*,j},
\end{aligned} \tag{12}$$

where the second inequality follows from the fact that $|\mathcal{L}_{j^*}| \leq L$. Now, we are left to show that the first term of Eq. (12) can be upper-bounded by OPT. To see this, we consider two cases.

Case I: $K \geq L$. In this case, $\mathcal{L}_{j^*} = \mathcal{L}$ and $\min\{K, L\} = L$, since the average completion time of all the threads is a lower-bound of the max-completion time, it follows that:

$$\begin{aligned}
\frac{1}{|\mathcal{L}_{j^*}|} \sum_{l\in\mathcal{L}_{j^*}} \sum_{j\in B_{1\to k}} \sum_{v\in D_j: v\to l} \tilde{c}_{v,j} &= \frac{1}{L}\tilde{C}(B_{1\to k}) \\
&= \frac{1}{L}\left(C(B_{1\to k}) + \Delta(\texttt{ALG})\right) \\
&\leq \frac{1}{L}\left(C(B_{1\to k}) + \Delta(\texttt{OPT})\right) \leq \texttt{OPT},
\end{aligned} \tag{13}$$

where the inequality follows from Lemma 2 and Corollary 1.

Case II: $K < L$. In this case, $\min\{K, L\} = K$. If $|\mathcal{L}_{j^*}| = L$, then the case is the same with *Case I* and the same bound holds. If $|\mathcal{L}_{j^*}| < L$, we point out that this can only be the case when $L > |B_{1\to k}| \cdot K$, since otherwise the last operation v^* was either assigned to a thread with higher workload, or delayed to create idle time on the thread. Both contradict to the work-conservation property of our algorithm. Therefore $L > |B_{1\to k}| \cdot K$, meaning that the threads are abundant, and we have

$$\sum_{l\in\mathcal{L}_{j^*}} \sum_{j\in B_{1\to k}} \sum_{v\in D_j: v\to l} \tilde{c}_{v,j} = \texttt{OPT}, \tag{14}$$

which implies Eq. (9).

Let X_b denote the event that all $j \in \mathcal{Q}$ are either `E-experts` or compressed experts, such that no bubble exists in the schedule, hence $\delta_{v,j} = 0, \forall j \in \mathcal{Q}, v \in D_j$. Combining all the results, we have

$$
\begin{aligned}
\texttt{ALG} &\le \max_{1\le k\le M} \left\{\max\left\{F_I(k), F_C(k)\right\} + P(B_{k\to M})\right\} \\
&\le \max_{1\le k\le M} \left\{\max\left\{I(B_{1\to k}), \texttt{OPT} + \left(1-\frac{1}{L}\right) \max_{j\in\mathcal{Q}, v\in D_j} \tilde{c}_{v,j}\right\} + P(B_{k\to M})\right\} \\
&= \max_{1\le k\le M} \left\{\max\left\{I(B_{1\to k}), \texttt{OPT} + \left(1-\frac{1}{L}\right) \max_{j\in\mathcal{Q}, v\in D_j} (c + \delta_{v,j}\mathbb{1}_{X_b})\right\} + P(B_{k\to M})\right\} \\
&\le \max_{1\le k\le M} \left\{\max\left\{I(B_{1\to k}), \texttt{OPT} + \left(1-\frac{1}{L}\right) (c + \rho u\mathbb{1}_{X_b})\right\} + P(B_{k\to M})\right\} \\
&\le \max_{1\le k\le M} \left\{\max\left\{I(B_{1\to k}), \texttt{OPT}\right\} + \left(1-\frac{1}{L}\right) (c + \rho u\mathbb{1}_{X_b}) + P(B_{k\to M})\right\} \\
&= \max_{1\le k\le M} \left\{\max\left\{I(B_{1\to k}), \texttt{OPT}\right\} + P(B_{k\to M})\right\} + \left(1-\frac{1}{L}\right) (c + \rho u\mathbb{1}_{X_b}) \\
&\le \max_{1\le k\le M} \left\{\max\left\{\texttt{OPT}, \texttt{OPT}\right\} + \texttt{OPT}\right\} + \left(1-\frac{1}{L}\right) Z \\
&\le 2\cdot\texttt{OPT} + \left(1-\frac{1}{L}\right)\cdot\texttt{OPT} \\
&= \left(3-\frac{1}{L}\right)\cdot\texttt{OPT},
\end{aligned}
\tag{15}
$$

where the third inequality holds by Lemma B.6, the fourth inequality is due to the fact that $\max\{a, b+d\} \le \max\{a,b\}+d$ holds for every $d \ge 0$, and the fifth inequality follows from Definition B.2 and Lemma B.3.

□

C. ZIPMOE's Cache Pool Planning

This section presents the detailed algorithmic procedures for right-sizing the cache pools for tensors with varying compression states. Let $\mathcal{N}$ denote the set of all expert ranks in a sparse layer (from 1 to the number of experts per layer), and $\mathcal{S}$ be any subset of $\mathcal{N}$. We denote the *rank-based* expert inclusion probability list as $\mathbf{F}_{\mathcal{N}} = [f_1, f_2, \cdots, f_{|\mathcal{N}|}]$, and its normalized counterpart, the *rank-based* expert selection probability list, as $\mathbf{P}_{\mathcal{N}} = [q_1, q_2, \cdots, q_{|\mathcal{N}|}]$. Procedures for obtaining $\mathbf{P}_{\mathcal{N}}$ from $\mathbf{F}_{\mathcal{N}}$ leverages the *modified iterative proportional fitting algorithm*, which is detailed in Appendix D. We further define $\mathbf{P}_{\mathcal{S}}$ as the partial distribution, represented by a sub-list extracted from $\mathbf{P}_{\mathcal{N}}$ according to the indices in $\mathcal{S}$. We first show how to compute the probability $\Phi_{\mathcal{S}}(h)$ of having exactly h experts sampled from $\mathcal{S}$.

Algorithm 2 Probabilistic Modeling of Cache Hits

Input: The partial probability distribution $\mathbf{P}_{\mathcal{S}}$ for rank-based expert activation.
Output: The probability distribution over the number of experts selected from $\mathcal{S}$.
Initialize the output distribution $\Phi_{\mathcal{S}} \leftarrow \mathbf{0}_{1\times(|\mathcal{S}|+1)}$
Initialize boundary condition $\Phi_{\mathcal{S}}(0) \leftarrow 1$
for $r = 1, 2, \cdots, |\mathcal{S}|$ **do**
    for $h = |\mathcal{S}|, |\mathcal{S}| - 1, \cdots, 1$ **do**
        ▷ *Update probability using the transition rule.*
        $\Phi_{\mathcal{S}}(h) \leftarrow \Phi_{\mathcal{S}}(h) \cdot (1 - q_r) + \Phi_{\mathcal{S}}(h-1) \cdot q_r$
    end for
    $\Phi_{\mathcal{S}}(0) \leftarrow \Phi_{\mathcal{S}}(0) \cdot (1 - q_r)$
end for
return Probability distribution $\Phi_{\mathcal{S}}$

Next, we demonstrate how ZIPMOE estimates the sparse layer makespan for any given cache hit pattern $\mathbf{h} = (h_{\mathcal{F}}, h_{\mathcal{C}}, h_{\mathcal{S}}, h_{\mathcal{E}})$. Specifically, the makespan is estimated using average decompression and I/O delays derived from offline profiling. To circumvent the high computational overhead of fully simulating *Algorithm* 1, we employ a heuristic approximation: whenever a decompression operation requires an `E-chunk` read, we penalize its latency by incorporating the corresponding reading delay. We then estimate the final makespan as the maximum of two potential bottlenecks: (1) the average computational workload distributed across L threads, and (2) the aggregate I/O workload. This approach yields an efficient approximation of the original schedule's performance (excluding token execution).

Algorithm 3 Makespan Estimation

Input: Number of activated experts k, cache hit pattern $\mathbf{h} = (h_{\mathcal{F}}, h_{\mathcal{C}}, h_{\mathcal{S}}, h_{\mathcal{E}})$, single `E-chunk` decompression delay c, single `E-chunk` reading delay v, `SM-chunk` reading delay u, number of threads L, number of exponent shards in each tensor K, number of tensors per expert n.
Output: Estimation of the sparse layer makespan.
▷ *Compute the total I/O workload*
$\mathtt{n}_{\mathtt{SM}} \leftarrow n \cdot \left(k - \sum_{p\in\{\mathcal{F},\mathcal{C},\mathcal{S}\}} h_p\right)$
$\mathtt{n}_{\mathtt{E}} \leftarrow n \cdot K \cdot \left(k - \sum_{p\in\{\mathcal{F},\mathcal{C},\mathcal{E}\}} h_p\right)$
$T_{I/O} \leftarrow \mathtt{n}_{\mathtt{SM}} \cdot u + \mathtt{n}_{\mathtt{E}} \cdot v$
▷ *Compute the total decompression workload*
$\mathtt{n}_{\mathtt{D}} \leftarrow n \cdot K \cdot (k - h_{\mathcal{F}})$
$T_{decomp} \leftarrow (\mathtt{n}_{\mathtt{E}} \cdot v + \mathtt{n}_{\mathtt{D}} \cdot c)\,/L$
return Estimated makespan $\max\left\{T_{I/O}, T_{decomp}\right\}$

Finally, the memory for each cache pool can be allocated using our *hierarchical cache planning* outlined by *Algorithm* 4. Although theoretically feasible, we do not restrict users to partition the memory space for each compression state in practice. Instead, we can control the granularity of the cache pools by selecting a subset $\Lambda \in \{\mathcal{F}, \mathcal{C}, \mathcal{S}, \mathcal{E}\}$. This flexibility allows users to choose the appropriate subset based on the hardware and operating system conditions.

Algorithm 4 Hierarchical Cache Pool Planning

Input: The *rank-based* expert inclusion probability list as $\mathbf{F}_{\mathcal{N}}$, candidate set of cache pools $\Lambda \in (h_{\mathcal{F}}, h_{\mathcal{C}}, h_{\mathcal{S}}, h_{\mathcal{E}})$, number of activated experts k, single `E-chunk` decompression delay c, single `E-chunk` reading delay v, `SM-chunk` reading delay u, number of threads L, number of exponent shards in each tensor K, number of tensors per expert n, grid size $0 < \delta < 1$.
Output: Memory ratios for each cache pool.
Initialization: $\texttt{minCost} \leftarrow \infty$, $\hat{\boldsymbol{\gamma}} := (\gamma_{\mathcal{F}}, \gamma_{\mathcal{C}}, \gamma_{\mathcal{S}}; \gamma_{\mathcal{E}}) \leftarrow (1, 0, 0, 0)$
Obtain the expert selection probability list $\mathbf{P}_{\mathcal{N}}$ via *modified iterative proportional fitting algorithm* in (25)
for each feasible memory ratios $\boldsymbol{\gamma}$ such that: $\sum_{p\in\Lambda} \gamma_p = 1, \gamma_{p'} = 0$ for $p' \notin \Lambda$, with step length δ **do**
    Compute the maximum number of experts S_p for each cache pool $p \in \Lambda$
    ▷ *Pack the cache pools into ranks according to the cache hierarchy, compute the distribution via DP*
    $u \leftarrow 0$
    for $p \in \Lambda \cup \mathcal{M}$ **do**
        Compute cache hit distribution Φ_p on positions $\{u+1, u+2, \cdots, u+S_p\}$ via *Algorithm* 2
        $u \leftarrow u + S_p$
    end for
    Compute cache hit distribution $\Phi_{\mathcal{N}}$ via *Algorithm* 2
    ▷ *Estimate the expected makespan under current allocation*
    $\texttt{Cost} \leftarrow 0$
    for $h_{\mathcal{F}} = 0, 1, \cdots, S_{\mathcal{F}}$ **do**
        for $h_{\mathcal{C}} = 0, 1, \cdots, S_{\mathcal{C}}$ **do**
            for $h_{\mathcal{S}} = 0, 1, \cdots, S_{\mathcal{S}}$ **do**
                for $h_{\mathcal{E}} = 0, 1, \cdots, S_{\mathcal{E}}$ **do**
                    $\texttt{Delay} \leftarrow$Estimate delay using *Algorithm* 3
                    $k_{\text{rem}} \leftarrow k - \sum_{p\in\Lambda} h_p$
                    if $k_{\text{rem}} < 0$ **then**
                        break
                    end if
                    $\texttt{Cost} \leftarrow \texttt{Cost} + \texttt{Delay} \cdot \frac{\Phi_{\mathcal{M}}(k_{\text{rem}})}{\Phi_{\mathcal{N}}(k)} \prod_{p\in\Lambda} \Phi_p(h_p)$
                end for
            end for
        end for
    end for
    if $\texttt{Cost} < \texttt{minCost}$ **then**
        $\texttt{minCost} \leftarrow \texttt{Cost}$
        $\hat{\boldsymbol{\gamma}} \leftarrow \boldsymbol{\gamma}$
    end if
end for
return Optimal memory ratio for each cache pool $\hat{\boldsymbol{\gamma}}$

D. Theoretical Analysis of Cache Pool Planning

Theorem D.1 (Restated)**.** *Among all probability distributions over k-sized expert subsets that are consistent with the observed individual expert selection counts, the distribution produced by the DP procedure achieves maximum entropy.*

Proof. To see this, we first construct the mathematical formulation of the entropy maximization problem. Let $\Omega = 2^{\mathcal{N}}$ be the power set of $\mathcal{N}$ and $\mathcal{P}(\mathcal{S})$ be the probability of selecting the set $\mathcal{S}$ of expert IDs. We have:

$$\begin{aligned} \max_{\mathcal{P}} & \sum_{\mathcal{S}\in\Omega} \mathcal{P}(\mathcal{S}) \log\left(\frac{1}{\mathcal{P}(\mathcal{S})}\right) \\ \text{s.t.} \quad & C_1 : \mathcal{P}(\mathcal{S}) = 0, \forall |\mathcal{S}| \neq k, \\ & C_2 : \sum_{\mathcal{S}\in\Omega:|\mathcal{S}|=k} \mathcal{P}(\mathcal{S})\, \mathbb{1}_{i\in\mathcal{S}} = f_i, \forall i \in \mathcal{N}, \\ & C_3 : \sum_{\mathcal{S}\in\Omega} \mathcal{P}(\mathcal{S}) = 1. \end{aligned} \tag{16}$$

It is straightforward that Eq. (16) is a convex optimization problem, whose Lagrangian can be given by:

$$\mathcal{L} = -\sum_{\mathcal{S}\in\Omega} \mathcal{P}(\mathcal{S}) \log \mathcal{P}(\mathcal{S}) + \alpha\left(\sum_{\mathcal{S}\in\Omega} \mathcal{P}(\mathcal{S}) - 1\right) + \sum_{\mathcal{S}\in\Omega:|\mathcal{S}|\neq k} \beta_{\mathcal{S}} \mathcal{P}(\mathcal{S}) + \sum_{i\in\mathcal{N}} \lambda_i \left(\sum_{\mathcal{S}\in\Omega:|\mathcal{S}|=k} \mathcal{P}(\mathcal{S})\, \mathbb{1}_{i\in\mathcal{S}} - f_i\right), \tag{17}$$

where α, $\beta_{\mathcal{S}}$, and λ_i are the Lagrange multipliers. The optimal solution $(\boldsymbol{\mathcal{P}}^*, \alpha^*, \boldsymbol{\beta}^*, \boldsymbol{\lambda}^*)$ of Eq. (16) satisfies the Karush-Kuhn-Tucker conditions:

$$\begin{aligned} \left.\frac{\partial \mathcal{L}}{\partial \mathcal{P}(\mathcal{S})}\right|_{\boldsymbol{\mathcal{P}}=\boldsymbol{\mathcal{P}}^*} &= -\log \mathcal{P}(\mathcal{S}) - 1 + \alpha + \mathbb{1}_{|\mathcal{S}|\neq k} + \sum_{i\in\mathcal{S},|\mathcal{S}|=k} \lambda_i = 0, \forall \mathcal{S} \in \Omega, \\ \left.\frac{\partial \mathcal{L}}{\partial \alpha}\right|_{\alpha=\alpha^*} &= \sum_{\mathcal{S}\in\Omega} \mathcal{P}(\mathcal{S}) - 1 = 0, \\ \left.\frac{\partial \mathcal{L}}{\partial \beta_{\mathcal{S}}}\right|_{\boldsymbol{\beta}=\boldsymbol{\beta}^*} &= \mathcal{P}(\mathcal{S}) = 0, \forall \mathcal{S} \in \Omega : |\mathcal{S}| \neq k, \\ \left.\frac{\partial \mathcal{L}}{\partial \lambda_i}\right|_{\boldsymbol{\lambda}=\boldsymbol{\lambda}^*} &= \sum_{\mathcal{S}\in\Omega:|\mathcal{S}|=k} \mathcal{P}(\mathcal{S})\, \mathbb{1}_{i\in\mathcal{S}} - f_i = 0, \forall i \in \mathcal{N}, \end{aligned} \tag{18}$$

solving which yields:

$$\mathcal{P}^*(\mathcal{S}) = \frac{\mathbb{1}_{|\mathcal{S}|=k}}{\mathcal{Z}} \exp\left(\sum_{i\in\mathcal{S},|\mathcal{S}|=k} \lambda_i^*\right), \tag{19}$$

where $\mathcal{Z} = \sum_{\mathcal{S}\in\Omega:|\mathcal{S}|=k} \exp\left(\sum_{i\in\mathcal{S},|\mathcal{S}|=k} \lambda_i^*\right)$ is the normalization factor.

Next, we show that Eq. (16) coincides with the distribution $\mathbb{P}$ resulted from our DP procedure. Let X_i be the indicator for expert selection, where $X_i = 1$ implies that expert i is selected in our sampling procedure, $X_i = 0$ otherwise. For any set S with cardinality k, our *Algorithm*2 samples this set with the following conditional probability:

$$\mathbb{P}\left[\mathcal{S} = S \mid \sum_{i\in\mathcal{N}} X_i = k\right] = \frac{\mathbb{P}\left[\mathcal{S} = S, \sum_{i\in\mathcal{N}} X_i = k\right]}{\mathbb{P}\left[\sum_{i\in\mathcal{N}} X_i = k\right]}. \tag{20}$$

Let q_i be the selection probability for expert i. Since $|S| = k$, we have:

$$\begin{aligned}\mathbb{P}\left[\mathcal{S} = S, \sum_{i\in\mathcal{N}} X_i = k\right] &= \left(\prod_{i\in S} q_i\right)\left(\prod_{i\in\mathcal{N}\setminus S}(1-q_i)\right) \\ &= \frac{\left(\prod_{i\in S} q_i\right)\left(\prod_{i\in\mathcal{N}\setminus S}(1-q_i)\right)\left(\prod_{i\in S}(1-q_i)\right)}{\prod_{i\in S}(1-q_i)} \\ &= \left(\prod_{i\in S}\frac{q_i}{1-q_i}\right)\left(\prod_{i\in\mathcal{N}}(1-q_i)\right) \\ &= \Gamma\left(\prod_{i\in S}\frac{q_i}{1-q_i}\right),\end{aligned} \tag{21}$$

where $\Gamma = \prod_{i\in\mathcal{N}}(1-q_i)$ is constant. On the other hand,

$$\mathbb{P}\left[\sum_{i\in\mathcal{N}} X_i = k\right] = \sum_{J\in\Omega:|J|=k}\mathbb{P}\left[[\mathcal{S} = J, \sum_{i\in\mathcal{N}} X_i = k\right] = \Gamma\sum_{J\in\Omega:|J|=k}\left(\prod_{i\in J}\frac{q_i}{1-q_i}\right), \tag{22}$$

where the last equality follows from Eq. (21). Combining Eq. (21) and Eq. (22), we have:

$$\begin{aligned}\mathbb{P}\left[\mathcal{S} = S \mid \sum_{i\in\mathcal{N}} X_i = k\right] &= \frac{\mathbb{P}\left[\mathcal{S} = S, \sum_{i\in\mathcal{N}} X_i = k\right]}{\mathbb{P}\left[\sum_{i\in\mathcal{N}} X_i = k\right]} = \frac{\prod_{i\in S}\frac{q_i}{1-q_i}}{\sum_{J\in\Omega:|J|=k}\left(\prod_{i\in J}\frac{q_i}{1-q_i}\right)} \\ &= \frac{\mathbb{1}_{|S|=k}\exp\left(\sum_{i\in S}\lambda_i\right)}{\sum_{J\in\Omega:|J|=k}\exp\left(\sum_{j\in J}\lambda_j\right)},\end{aligned} \tag{23}$$

where the last equality follows by the definition $\lambda_i := \log\left(\frac{q_i}{1-q_i}\right)$. Thus, $\mathbb{P}$ and $\mathcal{P}$ belong to the same parametric family of distributions. To show that $\mathbb{P}$ and $\mathcal{P}$ are essentially identical, we must find a set of sampling probabilities q_i^* for each expert rank i such that constraint C_2 in Eq. (16) is satisfied. This problem belongs to the class of weighted finite population sampling problems. It was shown by (Chen et al., 1994) that for any feasible set of inclusion probabilities $\{f_i\}_{i=1}^N$, there exists a unique set of positive weights $\{w_i^*\}_{i=1}^N$ (up to a scaling factor) such that the resulting distribution satisfies:(i) $P(\mathcal{S}) \propto \prod_{i\in\mathcal{S}} w_i^*$;(ii) $\sum_{\mathcal{S}\ni i,|\mathcal{S}|=k} P(\mathcal{S}) = f_i, \forall i \in \mathcal{N}$;and (iii) $\sum_{i\in\mathcal{N}} f_i = k$. Such set of w_i^* can be found via a *modified iterative proportional fitting algorithm* (Chen et al., 1994), which we restate as follows. We first define the *elementary symmetric polynomial* with respect to integer n and set $\mathcal{C}$ as:

$$\mathcal{R}(n, \mathcal{C}) := \sum_{S\subseteq\mathcal{C},|S|=n}\prod_{j\in S} w_j. \tag{24}$$

Let the iteration be indexed by t. Starting with $w_i^{(0)} = f_i$, we update the weights according to: $w_i^{(t+1)} \leftarrow w_i^{(t)} \cdot \frac{f_i}{f_i^{(t)}}$, where

$$f_i^{(t)} := w_i^{(t)}\frac{\mathcal{R}(k-1, \mathcal{N}\setminus\{i\})}{\mathcal{R}(k, \mathcal{N})}\bigg|_{\boldsymbol{w}=\boldsymbol{w}^{(t)}}, \forall i \in \{1, 2, \cdots, |\mathcal{N}|\}. \tag{25}$$

Such an iteration converges monotonically and geometrically to w_i^*. Finally, we can set $w_i^* = \exp(\lambda_i^*) = \frac{q_i^*}{1-q_i^*}$, or equivalently $q_i^* = \frac{w_i^*}{1+w_i^*}$. The resulting set of $\{q_i^*\}$ satisfies constraint C_2 of Eq. (16) and, therefore, produces a distribution that coincides with the maximum entropy distribution $\mathcal{P}^*(\mathcal{S})$.

□

E. Extended Results

E.1. Offline Initialization

We report the offline initialization time in Tab. 3. Overall, offline initialization spans from 6 minutes to 20 minutes, depending on specific models and compressor configurations. As this process is performed only once per model, the cost is practically affordable. Regarding memory overhead, ZipMoE employs parallel and asynchronous compression in its offloading engine.

Table 3. Offline compression latency for different models and compressors.

Model	**Compression Time (s)**	
	LZ4HC	**ZSTD**
DeepSeekV2-Lite 16B	760.02	427.99
Qwen1.5-MoE 14B	680.89	384.05
SwitchTransformers-Large-128 26B	1186.96	594.68

In addition to the memory required for model shards, extra RAM is required to buffer batches of tensors and support the compression pipeline. In practice, we observe less than 1 GB of additional memory overhead while achieving high compression throughput.

E.2. Sensitivity Analysis

We analyze the sensitivity of the cache-planning module under distribution shifts in expert activation probabilities. Specifically, we inject controlled distribution shifts into the cache-planning procedure by introducing a shift factor α to smooth the historical rank-based expert activation count list $\mathbf{F}_{\mathcal{N}}$. The shifted activation count list $\tilde{\mathbf{F}}_{\mathcal{N}}$ is defined as

$$\tilde{\mathbf{F}}_{\mathcal{N}} = (1-\alpha) \times \mathbf{F}_{\mathcal{N}} + \alpha \times \mathbb{E}(\mathbf{F}_{\mathcal{N}}).$$

We then use the shifted list $\tilde{\mathbf{F}}_{\mathcal{N}}$ as the input to the cache-planning module, thereby introducing a discrepancy between the planning distribution and the actual workload distribution. The experimental results are presented in Tab. 4.

Table 4. Sensitivity against workload distribution shift.

	TTFT		**TPOT**	
Shift Factor α	**Latency (s)**	**Slowdown**	**Latency (s)**	**Slowdown**
0.0	2.1783	1.000×	0.4576	1.000×
0.2	2.3055	1.058×	0.4709	1.029×
0.4	2.3634	1.085×	0.4762	1.041×
0.6	2.3616	1.084×	0.4777	1.044×
0.8	2.5285	1.161×	0.4989	1.090×
1.0	2.5442	1.168×	0.5011	1.095×

The results demonstrate that ZIPMOE's cache-planning module remains highly robust under workload distribution shifts. Even under the largest shift setting, the system incurs only a 16.8% increase in TTFT and a 9.5% increase in TPOT.

Here, we explain the rationale behind the robustness of this module. In practical deployments, while the specific IDs of activated experts may change across different prompts, the skewness (shape) of the rank-based activation probability distribution remains highly stable. As the cache-planning of ZIPMOE relies on rank-based popularity rather than absolute expert IDs, it successfully captures the invariant skewness. Hence, the planning is robust. We validated this by collecting inference traces using DeepSeekV2-Lite 16B model on the ShareGPT corpus. The trace distribution is illustrated in Fig. 12.

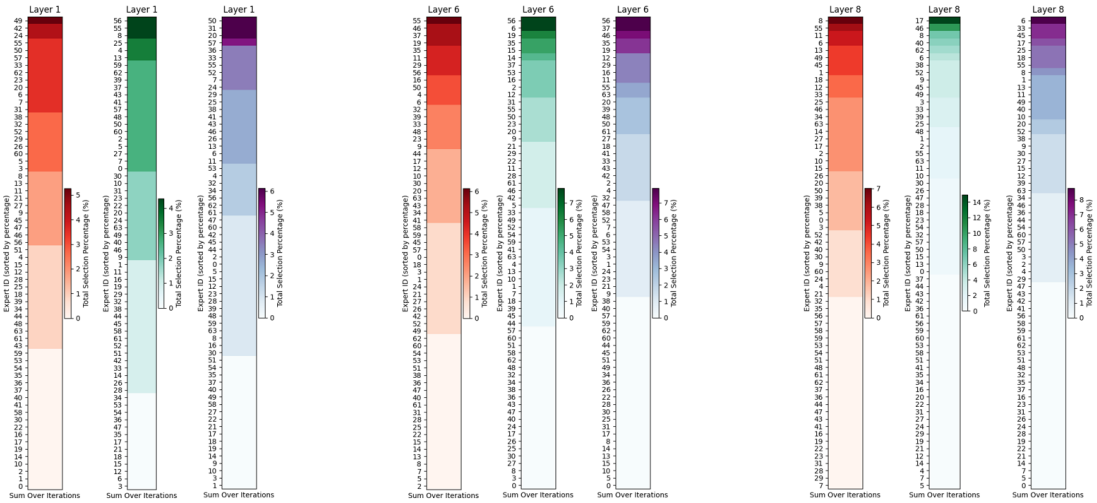

Figure 12. Rank-based distribution of expert activations. Different colors correspond to different prompts.

Table 5. TTFT performance (in seconds) of Qwen1.5-MoE 14B under different memory footprints.

Baseline \RAM	**5GB**	**10GB**	**15GB**	**20GB**	**25GB**	**30GB**
ZipMoE	3.1312	2.4597	2.0486	1.6392	1.3020	1.2303
FineMoE	20.6469	20.6426	20.7792	23.6066	23.4409	19.4810

Table 6. TPOT performance (in seconds) of Qwen1.5-MoE 14B under different memory footprints.

Baseline \RAM	**5GB**	**10GB**	**15GB**	**20GB**	**25GB**	**30GB**
ZipMoE	0.4082	0.3688	0.3331	0.3060	0.2644	0.2395
FineMoE	20.4272	20.4053	20.0166	23.1935	22.9447	19.4125

E.3. More Baseline Comparison

We compare ZIPMOE with a more recent baseline *FineMoE* (Yu et al., 2026) in Tab. 5 and Tab. 6. Somewhat surprisingly, although *FineMoE* achieves strong performance in cloud environments, its efficiency degrades significantly on edge devices. This is due to miss prefetching quickly saturates PCIe bandwidth of edge devices, which is more likely to occur in single-batch on-device workloads.

Table 7. Comparison of different compression techniques across varying memory footprints.

Memory	**TTFT (s)**			**TPOT (s)**		
Footprint (GB)	LZ4HC	ZSTD	Δ	LZ4HC	ZSTD	Δ
10	3.0456	3.3183	+0.2727	0.5484	0.5980	+0.0497
15	2.6663	2.8940	+0.2277	0.5088	0.5393	+0.0305
20	2.1783	2.4235	+0.2452	0.4576	0.4889	+0.0313
25	1.7508	1.9697	+0.2188	0.4115	0.4219	+0.0104
30	1.5525	1.6544	+0.1018	0.3666	0.3614	-0.0052

E.4. Comparison between Different Compressors

We compare the system performance using different compressor backends in Tab. 7. Overall, the choice between LZ4HC and ZSTD as the compression backend does not lead to significant performance differences. LZ4HC shows a slight advantage due to its better balance between compression ratio and decompression speed.

That said, ZIPMOE's current implementation exposes an open API that supports a wide range of compressor backends. This flexibility allows users to select the most suitable compressor for their target edge devices.