CFMoE: Cache-Friendly Mixture of Experts Model Serving System on Edge Devices

Haoyu Zhang[†1,2], Hongyi Wang[†1,3], Jianwei Zheng[1✉]
Yu Li[1], Tianyang Jiang[1], Zhenhua Li[3], Ning Jia[1]
[1]Huawei Technologies Co., Ltd [2]Peking University [3]Tsinghua University

Abstract

Deploying large language models (LLMs) on edge devices has garnered significant attention due to their inherent privacy and security benefits. However, the enormous parameter scale of modern LLMs poses fundamental challenges for resource-constrained edge devices. Mixture-of-Experts (MoE) models emerge as a promising solution by dynamically activating only a subset of experts during inference, and thus enhance computational efficiency. Nevertheless, the runtime uncertainty in expert activation still leads to high memory demands during inference. To address this, prior works concentrate on either compressing or offloading experts, all striving around the tradeoffs among the model precision, memory consumption, and serving throughput, and thus form an "impossible triangle".

To reconciliate the impossible triangle, we propose **CFMoE**, a **C**ache-**F**riendly **MoE** serving system that holistically integrates optimizations spanning the model architecture, fine-tuning strategy, and inference engine. The key insight behind CFMoE is that the edge-deployed MoE serving system needs to strengthen and leverage the locality of expert routing in order to break through the theoretical caching ceiling caused by the inherent load-balancing constraints in MoE models. Specifically, CFMoE integrates three novel techniques: 1) locality-aware expert routing mechanism, 2) data-aware elastic fine-tuning strategy, and 3) prefill-decoding disaggregated caching management. Evaluation results from state-of-the-art MoE models demonstrate that CFMoE significantly reduces memory usage by 32.9% on average without decreasing serving throughput or degrading model precision.

[†]Both authors contributed equally to this work.



This work is licensed under a Creative Commons Attribution 4.0 International License.
MobiCom '26, Austin, TX, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2505-0/26/10
https://doi.org/10.1145/3795866.3844455

CCS Concepts

• **Computing methodologies** → **Natural language processing**; • **Computer systems organization** → **Embedded systems**; • **Information systems** → *Storage architectures.*

Keywords

Mixture-Of-Experts Models, Edge Computing, On-Device Model Serving, Expert Caching, Large Language Models

ACM Reference Format:
Haoyu Zhang, Hongyi Wang, Jianwei Zheng, Yu Li, Tianyang Jiang, Zhenhua Li, Ning Jia. 2026. CFMoE: Cache-Friendly Mixture of Experts Model Serving System on Edge Devices. In *The 32nd Annual International Conference on Mobile Computing and Networking (MobiCom '26), October 26–30, 2026, Austin, TX, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3795866.3844455

1 Introduction

Large language models (LLMs) have demonstrated exceptional performance in various domains, including text understanding [39], content generation [49], and mathematical reasoning [4]. Most widely-used LLMs (e.g., GPTs [2, 9] and LLaMas family [15, 40]) are conventionally hosted on large-scale datacenters with cutting-edge GPUs to provide cloud-based services. Recently, the local deployment of LLMs on edge devices (e.g., PCs and smartphones) has attracted considerable attention, primarily owing to inherent benefits of better privacy and security. Furthermore, on-device LLMs enable seamless integration with operating systems, thereby potentially catalyzing novel edge applications, such as personalized virtual assistants[33], offline document summarization [34], and on-device content generation [27].

Currently, most mainstream on-device LLMs rely on dense architectures, such as Llama3 [15] and the Tencent Hunyuan series [19–22]. As these models scale continuously in pursuit of superior performance, their escalating inference demands for memory and computation have outpaced advancements in mobile hardware, resulting in diminished serving throughput or outright failures from out-of-memory errors that impede on-device deployment. To address this, Mixture-of-Experts (MoE) models [3, 7, 13, 28, 38, 46] offer a compelling alternative by substituting the single feed-forward

network in the Transformer with multiple specialized experts, routed via a gating network. Activating only a subset of experts per token substantially lowers the active parameters, yielding higher throughput than equivalently sized dense models and rendering MoE architectures particularly apt for resource-constrained edge devices [10, 14].

Nevertheless, the runtime uncertainty in expert activation leaves many idle experts occupying valuable memory resources, creating a significant memory bottleneck for on-device MoE serving [10, 42, 50]. As shown in Table 2, even with quantization (e.g., INT4 and INT8), state-of-the-art MoE models [3, 25, 44, 48] still require memory far beyond the capacity of mainstream consumer-grade edge devices [42].

To mitigate this memory bottleneck without compromising model precision, prior studies [10, 36, 47] explore expert offloading strategies (MoE-offloading for short), which exploit the heterogeneous storage hierarchies to offload experts from faster but smaller-capacity storage tiers to slower but larger-capacity ones and dynamically load experts on demand. However, the limited on-device cache capacity inherently degrades the cache hit rate, introducing excessive overhead from frequent expert swaps and a consequent reduction in serving throughput.

To further alleviate the side effects of MoE-offloading, expert prefetching methods (MoE-prefetching for short) [8, 24, 36, 47] are proposed to overlap expert loading and computation by predicting upcoming experts. However, since expert activation patterns are highly dynamic and input-dependent, it is highly challenging to accurately forecast future expert activations using historical input patterns alone. Consequently, inaccurate predictions inevitably result in misfetches, which introduce substantial extra expert loading overhead. As shown in Figure 3, as the prefetch hit rate decreases, the serving throughput of MoE systems will drop sharply. Furthermore, this issue is particularly severe on edge devices, where prefetch failures are far more costly.

As mentioned above, existing optimizations for on-device MoE model serving entangle in a hard-to-reconcile triangle among *model precision*, *memory usage*, and *serving throughput*. To fundamentally reconcile this "impossible triangle", we conduct a measurement study to reveal the root cause that constrains the practical deployment of MoE models on edge devices. Combined with source code investigation of mainstream MoE models [3, 7, 23, 28], we discover that **the root cause lies in the poor locality in expert routing between consecutive tokens which stems from the load balance constraint adopted by these models**.

The load balance constraint is originally introduced to accommodate the deployment of MoE models in cloud, where expert parallelism (EP) across multiple GPUs or NPUs is the standard practice to accelerate training and inference [11]. In such a setting, each expert must process inputs of identical shape to enable efficient batched execution [28]. Without load-balancing, token assignments across experts would be highly skewed, causing overloaded experts to truncate inputs and under-utilized experts to pad them, both of which severely deteriorate serving throughput. To mitigate this imbalance, mainstream MoE models enforce a load-balancing constraint that encourages the router to distribute tokens evenly across experts within a batch. While this design is well-suited for cloud-scale serving, it inevitably sacrifices locality between consecutive tokens, since the routing is biased towards balancing workloads rather than preserving temporal or semantic continuity in expert activation.

Moreover, such a cloud-oriented design becomes detrimental when MoE models are deployed on edge devices. First, unlike the cloud servers equipped with ample High Bandwidth Memory (HBM) to accommodate entire MoE models, edge devices are typically built on a Unified Memory Architecture (UMA) where a limited DRAM capacity is shared among the SoC's processing units (CPU, GPU, NPU) and supplemented by slow flash storage to hold the model. This makes the locality of expert routing a critical factor, as frequent expert swaps between DRAM and flash significantly increase I/O overhead and decrease serving throughput. Second, the advantage of load-balancing for EP vanishes on edge devices, as their hardware is inherently unsuited for EP. Specifically, the GPU of mobile devices is mainly reserved for rendering and the single-core NPU cannot support expert parallelism. Third, inference on the edge usually operates with batch size equal to one, leading to distinct activation patterns: almost all experts of each layer are activated in prefill, whereas only top-k experts are activated during decoding.

Based on the above insights, we propose **CFMoE**, an end-to-end **C**ache-**F**riendly **MoE** serving system for edge devices, featuring an integrated optimization of training and inference. In the training phase, CFMoE designs a novel cache affinity constraint (CAC) and incorporates it into the expert routing network, reconstructing the vanilla MoE model into an expert locality-aware architecture that aligns expert activation with cache efficiency (§4.2). Second, to efficiently adapt the MoE model to the CAC while preserving model performance, CFMoE proposes a data-aware fine-tuning strategy by leveraging Low-Rank Approximation (LoRA) [18], which elastically adds LoRA adapters to the locality-deficient layers based on layer-wise profiling of expert routing (§4.3).

In the inference phase, CFMoE introduces a prefill-decoding (PD) disaggregated caching management mechanism to accommodate the stage-wise characteristics of expert activations. This mechanism employs a hybrid expert cache layout customized for each stage, coupled with a priority-based eviction policy, to optimize cache utilization and improve the cache hit rate (§4.4). Building on this, to further reduce the I/O overhead of loading experts when the cache miss occurs,

CFMoE asynchronously fetches experts from the flash by integrating an asynchronous I/O framework [5] for saturating I/O bandwidth of storage.

We implement CFMoE with 3.5K+ lines of Python and 15K+ lines of C/C++ code, and evaluate its real-world performance with SOTA MoE models and representative open-source datasets. The evaluation results show that CFMoE reduces *memory usage* by 32.9% on average without compromising *model precision* or decreasing *serving throughput*. This memory saving leaves more capacity for co-running applications, improving end-user experience on edge devices.

Contribution. We make the following contributions:

- We make an in-depth study on expert activation patterns of MoE models, revealing the root cause of the "impossible triangle" and potential optimization opportunities.
- We design and implement CFMoE through training-inference co-design, integrating optimizations across the model architecture, fine-tuning strategy, and inference engine.
- We conduct a comprehensive evaluation of CFMoE under various realistic edge scenarios of MoE serving, and demonstrate its effectiveness and generalizability.

The relevant code and data have been released at https://cfmoe.github.io to benefit the community.

2 Background and Related Works

2.1 A Primer on MoE Models

The rapid growth of LLM parameters poses a significant challenge for efficient deploying and serving these models. To mitigate this issue, the MoE architecture has emerged as a promising solution [3, 11, 14, 23, 35, 43], which leverages sparse expert activation to scale model parameters while maintaining a relatively low inference overhead.

As illustrated in Figure 1, compared to dense LLMs, MoE models replace the feed-forward network (FFN) block in each Transformer layer with an MoE block that consists of a lightweight gating network and multiple parallel experts. For each input token, the gating network of each layer produces a probability distribution over experts (i.e., expert routing) and routes the token to the top-k experts with the highest scores. Only the selected top-k experts are activated and involved in computation, while the others remain inactive, thereby significantly reducing the number of parameters used by per token and improving computational efficiency.

2.2 On-device Deployment of MoE Models

Owing to sparse expert activation, MoE models are particularly promising for deployment on resource-constrained edge devices. By activating only a small subset of parameters for each token, an MoE model keeps its per-token computation comparable to that of a much smaller dense model, while leveraging a substantially larger total model capacity to achieve better performance. As shown in Table 1, Qwen3-30B-A3B, an MoE model with 30B total parameters but only 3B activated parameters per token, consistently outperforms its same-family dense counterparts, Qwen3-4B (4B parameters) and Qwen3-8B (8B parameters), across all evaluated datasets. These results demonstrate the model-quality advantage of MoE models under comparable activated computation and motivate their use for on-device serving.

However, the multi-expert design leads to an explosive growth in total model parameters. As shown in Table 2, state-of-the-art MoE models contain tens of billions of parameters. Even after applying quantization techniques such as INT8 or INT4, these models still require more than 8 GB of memory to be served. Worse still, unlike cloud GPUs and NPUs with high-capacity HBM dedicated to model inference, GPUs and NPUs in mainstream consumer-grade edge devices (e.g., PCs and phones) are typically integrated into the System on Chip (SoC) and lack dedicated memory. As a result, model parameters must reside in unified system memory (i.e., DRAM), which is shared with system services and user applications, and typically limited to 8–32 GB. Thus, on-device MoE serving faces a severe memory bottleneck. To mitigate this, existing studies primarily explore three directions: expert compression, expert offloading, and expert prefetching (referred to as MoE-compression, MoE-offload, and MoE-prefetch, respectively).

MoE-compression. Existing methods reduce the memory footprint mainly by simplifying the model through expert pruning [12, 32, 37] or knowledge distillation [1, 16, 30]. Expert-Sparsity [31] prunes or skips less important experts; MoE++ [26] uses three zero-computation expert types to skip partial expert computation or entire MoE layers according to token complexity; and DeepSpeedMoE [3] uses staged distillation to build a 12.5%-smaller Mixture-of-Students. Such compression inevitably compromises precision, especially when aggressive enough to meet edge memory limits.

MoE-offload. To reduce the memory usage without sacrificing model precision, MoE-offload methods leverage hierarchical storage as shown in Figure 2. They dynamically offload inactive experts to lower tiers (e.g., DRAM on cloud servers or flash storage on edge devices), and load active ones into high-speed caches (e.g., HBM on cloud servers or DRAM on edge devices) in a load-on-demand (LoD) manner. For example, SiDA-MoE [8] offloads inactive experts from the GPU memory to the CPU memory, and llm.npu [45] keeps only active experts in NPUs for on-device MoE serving.

However, unlike cloud servers where HBM can accommodate most experts, edge devices provide limited memory (8–32 GB) and face much more frequent cache misses and expert loading from flash to DRAM. Moreover, while cloud servers

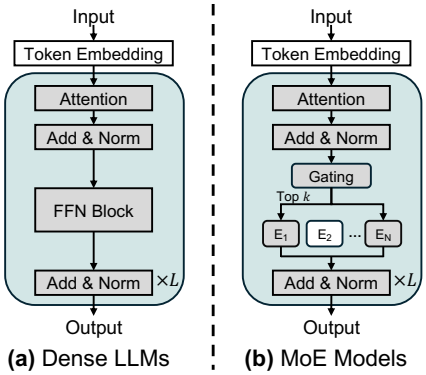


Figure 1: Structures of dense (a) and MoE (b) models.

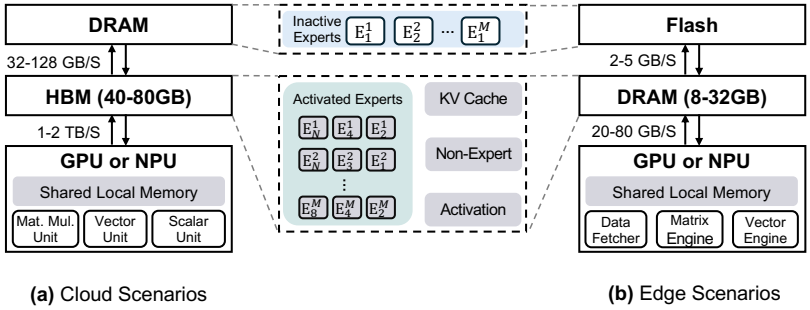


Figure 2: MoE serving systems with expert offloading strategies in cloud (a) and edge (b) scenarios.

Table 1: Comparison between same-family dense and MoE models across 12 benchmark datasets on multiple tasks.

			Dense models		MoE model
Dataset	**Task**	**Metric**	**Qwen3-4B**	**Qwen3-8B**	**Qwen3-30B-A3B**
Alpaca	Q&A	ROUGE-L	18.1	24.7	**33.8**
WikiText	Q&A	ROUGE-L	16.2	23.8	**26.4**
SAMSum	Q&A	ROUGE-L	28.5	35.2	**37.4**
GSM8K	Math	Accuracy	87.79	89.84	**91.81**
MaWPS	Math	Accuracy	74.3	81.2	**86.1**
MathQA	Math	Accuracy	59.8	62.4	**66.1**
AQUA	Math	Accuracy	52.6	55.1	**59.2**
BoolQ	Com. Sen.	Accuracy	85.1	86.5	**89.2**
PIQA	Com. Sen.	Accuracy	75.9	86.4	**89.8**
SIQA	Com. Sen.	Accuracy	66.2	78.7	**82.1**
OBQA	Com. Sen.	Accuracy	69.7	81.0	**86.2**
MMLU	Com. Sen.	Accuracy	72.99	76.89	**81.38**

Table 2: Memory demands of state-of-the-art MoE models v.s. Hardware specifications for mainstream edge devices. "#Params": the number of total parameters; "INT8"/"INT4": the model size after INT8/INT4 quantization. "Memory": the system DRAM capacity; "CPU-DRAM": the R/W bandwidth between DRAM and CPU; "NPU-DRAM": the R/W bandwidth between DRAM and NPU; "DRAM-Flash": the R/W bandwidth between the flash storage and DRAM.

Model demand				Hardware specification				
Model	**#Params**	**INT8**	**INT4**	**Type**	**Memory**	**CPU-DRAM**	**NPU-DRAM**	**DRAM-Flash**
Mixtral-8x7B [25]	47 billion	47 GB	23.5 GB	PC	32 GB	40–50 GB/s	70-80 GB/s	4–5 GB/s
Yuan-M32 [44]	40 billion	40 GB	20 GB	PC	16 GB	40–50 GB/s	70-80 GB/s	4–5 GB/s
Qwen3-MoE-32B [48]	32 billion	32 GB	16 GB	Phone	12GB	20–30 GB/s	20-30 GB/s	2–4 GB/s
DeepSeek-MoE-16B [3]	16 billion	16 GB	8 GB	Phone	8GB	20–30 GB/s	20-30 GB/s	2–4 GB/s

benefit from DRAM-to-HBM bandwidth (hundreds of GB/s), edge devices are constrained by much lower Flash-to-DRAM bandwidth (2–5 GB/s). Together, these limitations substantially increase expert-loading I/O overhead and degrade the serving throughput of MoE-offload on edge devices.

MoE-prefetch. MoE-prefetch predicts future active experts and overlaps their loading with current-layer computation, avoiding the GPU/NPU stalls of LoD. Lina [29] and MoE-Infinity [47] exploit historical cross-layer correlations, while SiDA-MoE [8], Pre-Gated MoE [24], and Fate [10] redesign routing or add predictors. Runtime routing uncertainty nevertheless causes irrelevant prefetches and extra

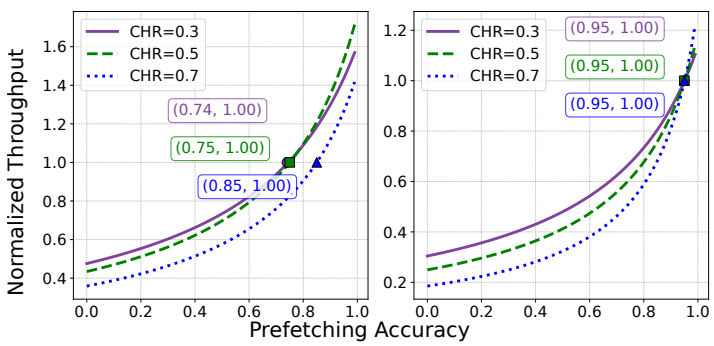


Figure 3: Normalized serving throughput of DeepSeek-MoE-16B under load-on-demand (LoD) and MoE-prefetch across prefetch accuracies in cloud (left) and edge (right) scenarios.

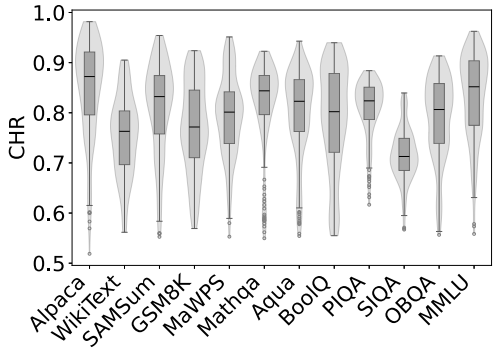


Figure 4: The CHR distribution of vanilla DeepSeek-MoE-16B on 12 benchmarks, with 2 GB cache size and OPT policy.

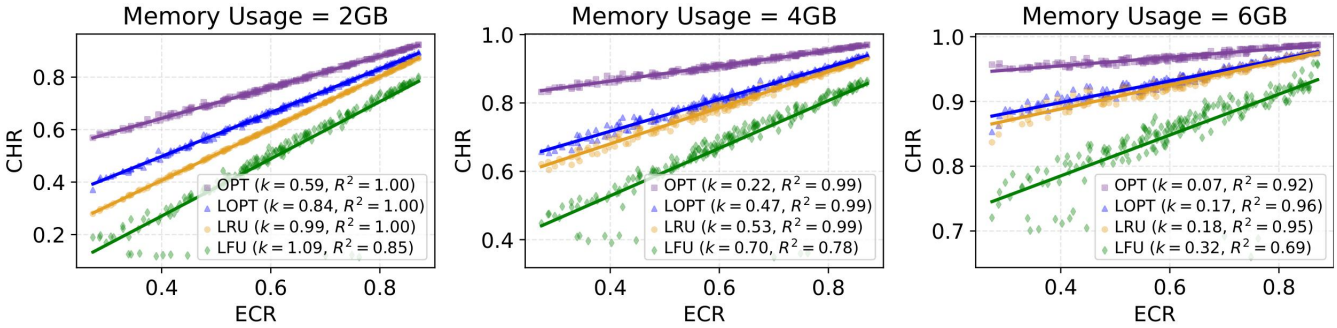


Figure 5: The correlation between CHR and ECR with different cache sizes and eviction policies.

I/O, especially when edge I/O bandwidth and batch sizes cannot hide loading latency. We quantify this effect by recording LoD expert-activation traces for DeepSeek-MoE-16B at CHRs 0.3, 0.5, and 0.7 and replaying MoE-prefetch in cloud ($F_{\text{Cloud}} \approx 1000$ TFLOPS, $B_{\text{GPU}} \approx 2000$ GB/s, and $B_{\text{IO}} \approx 500$ GB/s) and edge ($F_{\text{Edge}} \approx 1.7$ TFLOPS, $B_{\text{NPU}} \approx 68$ GB/s, and $B_{\text{IO}} \approx 3.3$ GB/s) settings. MoE-prefetch outperforms LoD only above 95% accuracy on edge, versus about 75% in the cloud; because such accuracy is rarely achieved, its edge benefit is limited and can be negative.

Comparison with our work. Under limited edge memory and slow Flash-to-DRAM I/O, MoE-compression compromises precision, whereas MoE-offload and MoE-prefetch reduce throughput, forming an "impossible triangle" among *model precision, memory usage, and serving throughput*. Our integrated training and inference optimizations reconcile this trade-off for resource-constrained edge devices.

3 Motivation

In order to preserve model precision while adapting to the limited memory on edge devices, we adopt the MoE-offloading direction as our foundation. However, as discussed in §2.2, MoE-offloading approaches suffer from degraded serving throughput in edge scenarios, which cannot be effectively alleviated by expert prefetching.To address this challenge, we first dissect the key factors impacting the serving throughput and identify low expert routing locality as the primary bottleneck (§3.1). We then trace the root cause to the cloud-oriented design of MoE routing, which fails to align with constraints and execution patterns of edge devices (§3.2).

3.1 Factors Affecting Serving Throughput

We conduct a measurement study to pinpoint the key factors impacting the serving throughput of MoE-offloading approaches, which is primarily constrained by the cache hit rate. Specifically, we profile the cache hit rate of a representative MoE model, DeepSeek-MoE-16B (after INT4 quantization), across 12 popular benchmarks (listed in Table 5 in Appendix A.2) under varying expert cache sizes (varying from 1 GB to 8 GB) and expert eviction policies, including the least recently used (LRU), the least frequently used (LFU), the theoretical optimal (OPT), and the limited optimal (LOPT) policies. Among them, the LRU and LFU are widely adopted strategies, while the OPT policy leverages full knowledge of future expert activation to evict the farthest-used expert, thus representing the theoretical upper bound of the cache hit rate. The LOPT policy represents optimized cache policies that make eviction decisions by predicting expert activations within a limited future window (empirically set to 10 tokens).

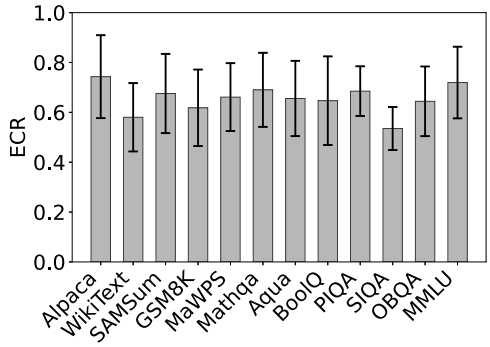


Figure 6: The ECR distribution of the vanilla DeepSeek-MoE-16B on 12 benchmarks.

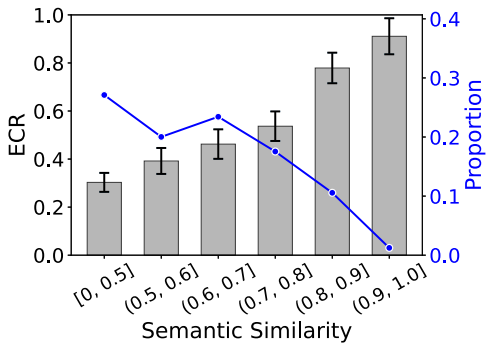


Figure 7: ECR and the distribution of adjacent token pairs across semantic similarity ranges.

Frustratingly, we note that even with the OPT policy, the cache hit rate is still unsatisfying, especially when the cache size is small. For instance, when the memory budget is below 2 GB, the hit rate drops below 0.8. Moreover, this rate is unattainable even by optimized (LOPT) eviction policies, indicating that most existing system-level cache optimizations can not effectively address the problem. Surprisingly, as illustrated in Figure 4, even under identical settings of intuitively influential factors (e.g., cache size and cache policy), different requests still exhibit substantial hit rate variations.

To demystify these counter-intuitive observations, we trace expert activation patterns during inference and reveal the locality of expert routing as a critical contributor to the cache hit rate. Specifically, we quantify routing locality by the *expert consistency rate* (ECR), the average Jaccard similarity between the activated-expert sets of consecutive tokens; its formal definition is provided in Appendix A.1.

As shown in Figure 5 and Table 5 in Appendix A.2, the cache hit rate exhibits a strong and approximately linear correlation with ECR, with a particularly steeper slope under lower cache sizes. This highlights expert routing locality as a critical determinant of cache effectiveness and, consequently, serving throughput. Moreover, the performance gap between different eviction policies narrows as ECR increases, suggesting that even suboptimal eviction strategies can achieve high throughput when expert routing locality is preserved.

Unfortunately, we notice that current MoE models generally suffer from low expert routing locality in practice. As shown in Figure 6, the ECR of DeepSeek-MoE-16B across the 12 benchmarks remains relatively low, ranging from 0.535 to 0.743. Therefore, such a poor locality constitutes the primary bottleneck limiting on-device serving throughput.

3.2 Root Cause Analysis

To figure out the underlying reason of the low locality in expert routing, we first analyze the relationship between the semantic similarity (measured by cosine similarity) of adjacent token pairs and their ECR across all datasets. As shown in Figure 7, 78.2% of adjacent token pairs have semantic similarity below 0.8; and even for highly similar pairs (cosine similarity $\in (0.7, 0.8]$), the ECR remains low at only ≈ 0.52. This observation suggests that additional factors also play an important role in determining which experts are activated.

Driven by this, we then conduct a source code investigation of mainstream MoE models [3, 7, 11]. As a result, we trace the root cause as the load-balancing constraint adopted by MoE models, which is aimed at improving computational efficiency in cloud scenarios, where MoE models are typically trained and served in a distributed manner [7, 11].

To support the deployment of MoE models in cloud environments, existing systems adopt expert parallelism (EP) to accelerate inference and reduce per-device memory pressure. In such a setting, experts are distributed across multiple GPUs or NPUs, and tokens are routed via all-to-all communication to the devices hosting their assigned experts. EP relies on batched matrix multiplication, which requires all expert inputs to have identical shapes for efficient execution on modern hardware. However, the uncertainty of MoE routing often leads to skewed token assignments across experts. Therefore, to satisfy the shape requirement, overloaded experts must truncate their token inputs, while under-utilized experts require padding to match the shape, resulting in either degraded model precision or a waste of computation.

To address the load imbalance problem, mainstream MoE models adopt a load-balancing constraint on the expert routing [11], which is formulated as: $\mathcal{L}_{\text{LBC}} = \lambda \cdot \frac{1}{E} \sum_{e=1}^{E} \frac{c_e}{S} \cdot m_e$, where E is the number of experts in a layer, S is the total number of tokens within a training batch, and m_e and c_e denote the average routing probability and the number of assigned tokens of expert e, respectively. This constraint promotes an even distribution of tokens across experts, thereby balancing the workloads and improving the efficiency of EP.

However, though well-suited for cloud-scale serving, the load-balancing constraint inevitably degrades routing locality by enforcing more uniform token assignments, inevitably reducing consecutive-token expert overlap and thus limiting

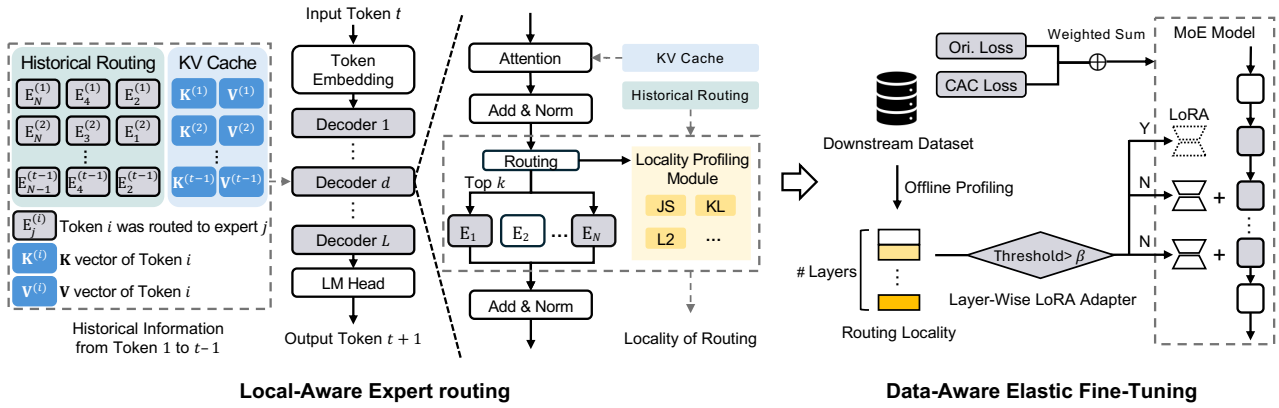


Figure 8: Architecture of the training phase of CFMoE.

CHR during inference (§3.1). This design becomes especially detrimental when MoE models are deployed on edge devices for the following three reasons:

- **Limited memory capacity.** Unlike cloud servers with sufficient HBM to accommodate the entire MoE model, edge devices typically adopt a Unified Memory Architecture (UMA), where limited DRAM is shared by the SoC's CPU, GPU, and NPU. Thus, only a subset of experts can reside in DRAM, while the rest must be stored in flash and loaded on demand during inference. In this setting, poor expert routing locality becomes a key bottleneck: frequent expert swaps between DRAM and flash incur substantial I/O overhead, severely degrading serving throughput.
- **Insufficient support for EP.** The benefits of load balancing diminish on edge devices, whose SoC-integrated accelerators (GPUs and NPUs) are not well suited for EP. Specifically, mobile GPUs are often reserved for graphics rendering, while NPUs typically operate as single-core units and lacks the multi-threading support for running multiple experts in parallel. Moreover, these accelerators do not support the all-to-all communication required by EP, preventing cloud-style expert distribution across cores.
- **Distinct activation patterns.** On edge devices, inference usually runs with batch size one, leading to highly different expert activation patterns between prefill and decoding. Prefill activates nearly all experts in each layer due to the large number of prompt tokens, whereas decoding activates only a small top-k subset per token. This shift breaks the more uniform activation pattern assumed by global load balancing, reducing its effectiveness.

Therefore, rather than improving inference efficiency, the load-balancing constraint fundamentally limits cache hit rates and thereby throttles the serving throughput on edge devices. This motivates our reconstruction of the MoE routing module, hoping to boost the throughput by essentially addressing the adverse impact of the cloud-oriented design.

4 Design and Implementation

4.1 Overview

Guided by our analysis in §3, we propose CFMoE, a cache-friendly MoE serving system that consists of two phases including offline training and online low-memory inference. CFMoE supports either a global adapter trained on aggregated multi-task data or optional task-specific adapters for further specialization. In the training phase (Figure 8), CFMoE first proposes a novel edge-oriented cache affinity constraint (CAC) and integrates it into the gating module of the vanilla MoE model, reconstructing it into an expert locality-aware architecture(§4.2). Second, to efficiently adapt the MoE model to the CAC and downstream tasks, CFMoE introduces a data-aware LoRA fine-tuning (DAFT) strategy (§4.3).

During inference (Figure 9), CFMoE introduces a PD disaggregated caching management mechanism to accommodate the distinct activation patterns on edge devices, coupled with a priority-based eviction policy to additionally optimize cache utilization and improve the cache hit rate (§4.4). Additionally, CFMoE integrates an asynchronous I/O framework called io_uring to further reduce the overhead of loading experts from the flash to DRAM by saturating I/O bandwidth.

4.2 Locality-Aware Expert Routing

To overcome the low cache hit rate caused by cloud-oriented load balancing, CFMoE reconstructs the original expert routing into a locality-aware architecture. As shown in Figure 8, this is achieved by introducing a profiling module to quantify expert locality, which subsequently guides a novel edge-oriented cache affinity constraint (CAC) to enhance locality.

Specifically, CFMoE inserts a locality profiling module into the MoE routing block to measure routing locality. We quantify it as the distance between the expert activation distributions of consecutive tokens, and benchmark several mainstream metrics, including Euclidean distance (ℓ_2 norm [6]),

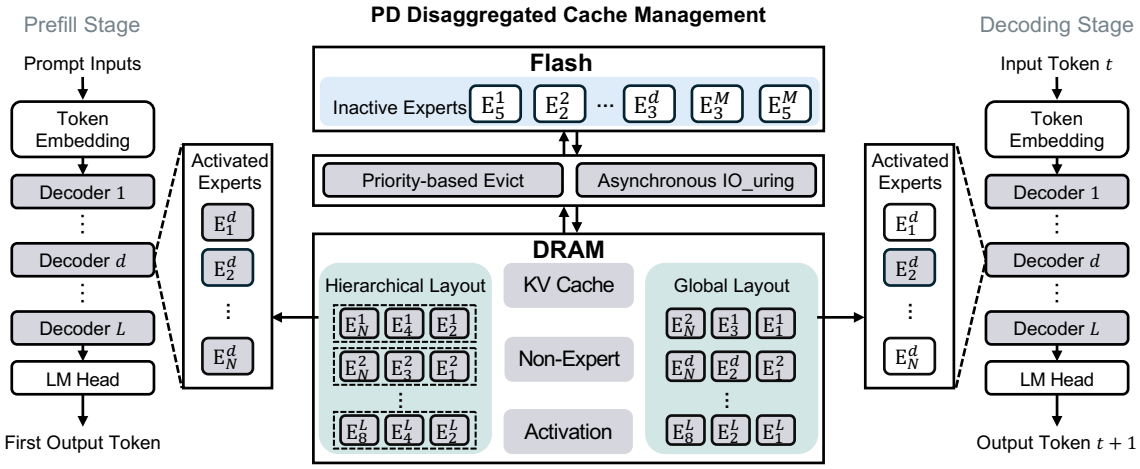


Figure 9: Architecture of the inference phase of CFMoE.

Kullback–Leibler divergence [17], and Jensen–Shannon divergence [41]. Among them, Jensen–Shannon divergence provides better training stability and faster convergence, while achieving a balanced trade-off between cache affinity and model precision, and is thus adopted as the final metric.

For training, we measure each layer's routing locality using the Jensen–Shannon divergence between the expert activation distributions of consecutive tokens; the full definition is provided in Appendix A.1.

Based on per-token routing locality, CFMoE proposes a novel cache affinity constraint (CAC) for edge scenarios and incorporates it as an auxiliary loss to make the vanilla MoE model locality-aware. The CAC is defined as the average routing locality across all tokens, sequences, and MoE layers:

$$\mathcal{L}_{\text{CAC}} = \frac{1}{S \cdot (T-1) \cdot M} \sum_{s=1}^{S} \sum_{t=2}^{T} \sum_{k=1}^{M} \mathcal{L}_k^{(s)}(t),$$

where S is the number of sequences in a batch, and T is the number of tokens in each sequence. Minimizing CAC encourages consecutive tokens to activate similar experts, thereby improving cache hit rate on memory-constrained devices and improving serving throughput.

4.3 Data-Aware Elastic Fine-Tuning

Adapting the locality-aware expert routing to diverse MoE models and heterogeneous edge scenarios while preserving model precision is particularly challenging due to the prohibitive cost of conventional fine-tuning. Fortunately, as shown in Figure 14 in Appendix A.2, the locality of expert routing in the vanilla MoE model varies significantly across tasks, models, and even layers. Based on this observation, as depicted in Figure 8, CFMoE introduces a data-aware fine-tuning (DAFT) strategy by elastically adding LoRA adapters only to layers with poor locality, thereby substantially reducing training overhead.

In detail, before fine-tuning, DAFT extracts a calibration subset from the downstream dataset and performs offline profiling to characterize the initial layer-wise expert routing locality of the MoE model. Guided by this profiling, DAFT then elastically inserts plug-and-play LoRA adapters into the critical layers whose initial routing locality falls below a empirical threshold β. The value of β is chosen to strike a balance between fine-tuning cost and the model's ECR.

During the fine-tuning, the training objective combines the original loss with CAC as a weighted auxiliary term:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{Ori}} + \alpha \cdot \mathcal{L}_{\text{CAC}},$$

where α is a hyperparameter that controls the importance of the CAC term for balancing the routing locality and model precision. Furthermore, by adopting the LoRA fine-tuning technique [18], which freezes the original model parameters and instead updates the newly inserted low-rank adapters, CFMoE further minimizes training overhead.

4.4 PD Disaggregated Caching Management

To enable efficient deployment of fine-tuned MoE models on edge devices, CFMoE incorporates an inference engine with prefill-decode (PD) disaggregated caching management, enhanced by a priority-based eviction policy and an asynchronous I/O framework.

As discussed in §4.2, expert activation patterns differ substantially between prefill and decoding on edge devices. However, existing MoE systems [8, 10, 47] typically adopt a global layout, where experts across layers share a unified cache pool. This design is particularly inefficient in the prefill phase: when many experts are activated in the current layer, they

can evict experts needed by subsequent layers, causing excessive cache misses and I/O overhead. To address this issue, CFMoE uses distinct cache layouts for prefill and decoding.

In the prefill phase, CFMoE adopts a *hierarchical* cache layout. The cache is partitioned into layer-specific sub-blocks, so experts of each layer are placed only within their assigned sub-block, eliminating inter-layer conflicts. To handle cases where one layer activates too many experts, CFMoE further reserves a shared temporary block for overflowed experts, ensuring that the total capacity of the layer-specific block and the temporary block can always accommodate all experts of one MoE layer. In the decoding phase, CFMoE switches back to a global cache layout. Because only a small top-k set of experts is activated in each layer, fixed per-layer partitions would leave much of the reserved space unused. A global layout instead allows unused cache space to be shared across layers, improving memory utilization.

To realize this mechanism, the inference engine initializes the cache as hierarchical sub-blocks based on device resources and model architecture, and places experts into their designated sub-blocks together with a shared temporary block for overflow handling. When execution switches from prefill to decoding, the engine merges all sub-blocks into a single global cache layout. When a new request starts and execution returns to prefill, the cache is partitioned again, and experts exceeding the per-layer capacity are evicted.

To further improve cache hit rate, CFMoE adopts a priority-based eviction policy that jointly considers recency (LRU) and frequency (LFU), and dynamically adjusts their relative importance according to observed access patterns. For example, CFMoE favors LRU under short-term hotspot access and LFU when expert accesses are more persistent. Building on these, CFMoE further improves serving throughput by reducing expert-fetch overhead. When cache misses occur, existing MoE systems typically use blocking I/O calls (e.g., read and write) to switch experts, which underutilizes available I/O bandwidth. To address this, CFMoE integrates the asynchronous I/O framework `io_uring` [5] via `liburing`. By invoking its user-level APIs such as `io_uring_prep_readv` and `io_uring_prep_writev`, CFMoE fetches experts asynchronously, thereby better utilizing storage bandwidth.

4.5 Implementation

Training Optimization. The locality-aware expert routing mechanism and data-aware elastic fine-tuning strategy are implemented with 3,500 lines of code in Python. We implement them as an optimization module which can be seamlessly applied to MoE-based architectures. Given a target MoE model and downstream dataset, the module first selects a small calibration subset (30% of the total samples, which is sufficient to represent routing behaviors) and runs inference to record expert activation patterns, profiling the layer-wise routing locality. Based on this, it identifies low-locality layers and inserts plug-and-play LoRA adapters into them. Subsequently, the module fine-tunes the model with our modified training objective that incorporates the CAC loss weighted alongside the original task loss, yielding a locality-aware MoE model tailored for the target task.

Inference Engine. We implement the on-device inference engine in 15,000 lines of C++. Built on top of CPU and NPU runtime interfaces, it implements transformer operators such as matrix multiplication, softmax, and RMS normalization. To support low-memory execution, the engine configures the memory budget during graph construction and initializes a hierarchical cache layout. At startup, it loads a memory-bounded subset of experts from each layer. During inference, the engine dynamically switches between hierarchical and global cache layouts for prefill and decoding, respectively. On a cache miss, it computes eviction priority from recency and frequency, evicts low-priority experts, and asynchronously loads the required ones with io_uring. The engine also supports inference-time optimizations such as low-bit quantization (e.g., INT8/INT4) and operator fusion.

5 Evaluation

We describe the setup in §5.1, evaluate overall performance in §5.2, isolate each module's contribution in §5.3, analyze sensitivity to key hyperparameters in §5.4, and quantify fine-tuning overhead in §5.5.

5.1 Experimental Setup

Hardware. We evaluate CFMoE on four representative edge execution settings: $D_{\text{PC+CPU}}$, $D_{\text{PC+NPU}}$, $D_{\text{Phone+CPU}}$, and $D_{\text{Phone+NPU}}$. $D_{\text{PC+CPU}}$ and $D_{\text{PC+NPU}}$ share the same PC platform, equipped with an Intel Ultra 5 125H CPU, 16 GB LPDDR5 memory, and a dedicated NPU. $D_{\text{Phone+CPU}}$ and $D_{\text{Phone+NPU}}$ share the same smartphone platform, equipped with a Kirin 9020 processor, 12 GB RAM, and a dedicated NPU. We denote the measured CPU/NPU↔DRAM bandwidth as B_{CPU} and B_{NPU}, and the DRAM↔Flash bandwidth as B_{IO}. On the PC, B_{CPU}, B_{NPU}, and B_{IO} are 45–55 GB/s, 75–85 GB/s, and 4–6 GB/s, respectively; on the phone, they are 35–45 GB/s, 65–75 GB/s, and 3–5 GB/s, respectively.

Models. We use DeepSeek-MoE-16B for evaluation because it balances capability and edge deployability. It contains 27 MoE layers, each with 2 shared experts and 64 routed experts, of which top-k=6 are activated per token in addition to the shared experts. Under 4-bit quantization, each expert occupies 4.866 MB, making expert-level loading practical on commodity edge devices. We further profile its serving characteristics on each device, including per-layer compute

time and single-expert loading time along Flash→DRAM and DRAM→NPU/CPU, as reported in Table 4 in Appendix A.2. For cross-architecture validation, we additionally evaluate CFMoE using Qwen3-30B-A3B on PCs ($D_{\text{PC+CPU}}$ and $D_{\text{PC+NPU}}$), since its fully loaded memory footprint (≥ 15GB with INT4 quantization) exceeds smartphone's memory capacity (typically ≤ 12 GB).

Finetuning Settings. The rank of each LoRA adapter is set as 8. As introduced in §4.2, DAFT has two key hyperparameters: α, which controls the weight of the CAC term, and β, which sets the expert-locality threshold. We empirically set $\alpha = 0.5$ to improve routing locality while preserving model precision. To choose β, we sweep different thresholds and measure the resulting fraction of trainable parameters. Based on the results, we choose $\beta = 0.7$, which reduces the trainable parameter count by nearly 30% while maintaining high routing locality for efficient inference.

Datasets. We use the 12 representative datasets for evaluation. We present their primary tasks and evaluation metrics for model precision in Table 3.

Metrics. We propose five metrics to evaluate CFMoE: 1) *end-to-end throughput*, which measures the average number of output tokens generated per second over an entire inference request; 2) *prefill throughput*, which is computed as the total number of input tokens divided by the time to generate the first output token; 3) *decoding throughput*, which measures the average number of output tokens generated per second during decoding; 4) *task-specific scores*, which evaluate model precision on each dataset; and 5) *memory usage reduction*, which measures the reduction in memory budget required by CFMoE to sustain the same throughput as LoD.

Baselines. As discussed in §2.2, compression techniques such as expert pruning [12, 32, 37] often compromise model precision for serving efficiency. Moreover, MoE-prefetch methods [8, 10, 24, 29, 47] require highly accurate expert prediction; otherwise, mis-prefetching overhead can offset their benefits on edge devices. Therefore, we use LoD as the primary baseline and include representative MoE-compression and MoE-prefetch methods as additional baselines. For fairness, all baselines use the fine-tuned models (without the CAC loss) for each dataset. We implement LoD on the on-device inference engine without CFMoE optimizations mentioned in §4.4, where experts are loaded from flash on demand and evicted by LRU policy under a given memory budget. For MoE-prefetch, we simulate expert prediction with 90% accuracy. A uniform 4-bit (INT4) quantization is applied to CFMoE, LoD, and MoE-prefetch, while MoE-compression quantizes only the experts to INT2 for flash I/O and restores them to INT4 for DRAM caching and computation.

5.2 Overall Performance

End-to-end Serving Throughput. We evaluate the end-to-end serving throughput of CFMoE across both MoE architectures and compare it against the respective on-device MoE serving baselines under architecture-specific memory budgets. We group the 12 datasets into three categories (Math, Q&A, and Common Sense) and report the mean throughput of each category (see Figure 10) and the weighted mean across all datasets for Qwen3-30B-A3B (see Figure 11).

For DeepSeek-MoE-16B, CFMoE achieves average speedups of 1.29× on $D_{\text{PC+CPU}}$, 1.40× on $D_{\text{PC+NPU}}$, 1.32× on $D_{\text{Phone+CPU}}$, and 1.41× on $D_{\text{Phone+NPU}}$ over LoD. The gains are pronounced under tight memory budgets: with <2 GB of memory, CFMoE improves throughput by up to 2.26× on $D_{\text{PC+CPU}}$, 2.56× on $D_{\text{PC+NPU}}$, 2.34× on $D_{\text{Phone+CPU}}$, and 2.57× on $D_{\text{Phone+NPU}}$. For Qwen3-30B-A3B, CFMoE achieves average speedups of 1.12× on $D_{\text{PC+CPU}}$ and 1.16× on $D_{\text{PC+NPU}}$ over LoD. With <4 GB of memory, CFMoE improves throughput by up to 1.26× on $D_{\text{PC+CPU}}$ and 1.32× on $D_{\text{PC+NPU}}$.

Additionally, although MoE-compression provides an average 1.29× speedup over LoD across all devices (Figure 10), CFMoE remains 1.04× faster. MoE-prefetch achieves 0.97× the throughput of LoD on average and incurs a slowdown of up to 28.2% due to mis-prefetching overhead.

Overall, CFMoE consistently improves serving throughput over the respective baselines across MoE architectures, benchmark categories, and hardware platforms.

Prefill and Decoding Throughput. Figures 11 and 12 break down throughput into the prefill and decoding stages for both MoE architectures. Across both architectures, CFMoE consistently improves both prefill and decoding throughput over LoD, with larger gains under tighter memory budgets. For DeepSeek-MoE-16B, CFMoE achieves average prefill and decoding speedups of 1.29× and 1.36×, respectively, over LoD across all devices. For Qwen3-30B-A3B, the corresponding speedups are 1.09× and 1.14×. With <2 GB of memory, the prefill and decoding speedups on DeepSeek-MoE-16B reach up to 2.41× and 2.59×, respectively; with <4 GB on Qwen3-30B-A3B, they reach up to 1.23× and 1.33×. These comparable gains indicate that the end-to-end speedup of CFMoE comes from improvements in both stages.

These gains primarily stem from improved expert routing locality. As demonstrated in Table 3, for DeepSeek-MoE-16B, CFMoE increases ECR by 0.117 on average over the baselines, with gains ranging from 0.011 to 0.266. As discussed in §3.1, the higher ECR improves cache affinity, reduces expert-loading I/O overhead, and thereby increases both prefill and decoding throughput.

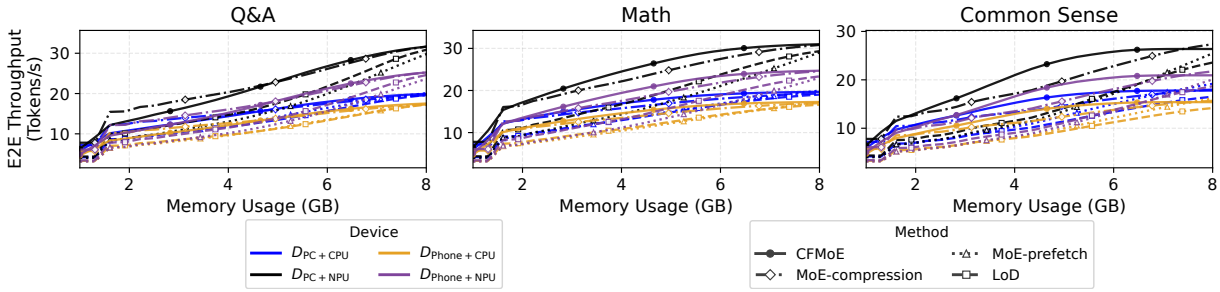


Figure 10: End-to-end serving throughput under different memory budgets across three dataset categories for all devices using DeepSeek-MoE-16B, including the evaluated baselines and CFMoE.

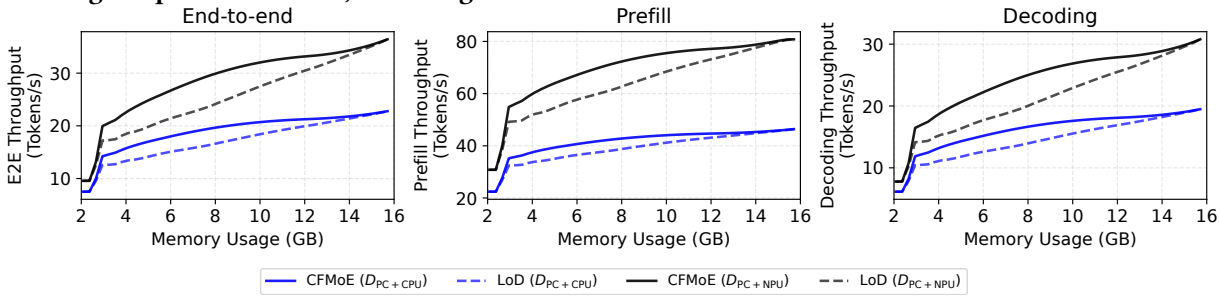


Figure 11: End-to-end, prefill, and decoding throughput under different memory budgets averaged across the 12 benchmark datasets using Qwen3-30B-A3B on the two PC settings, including the LoD baseline and CFMoE.

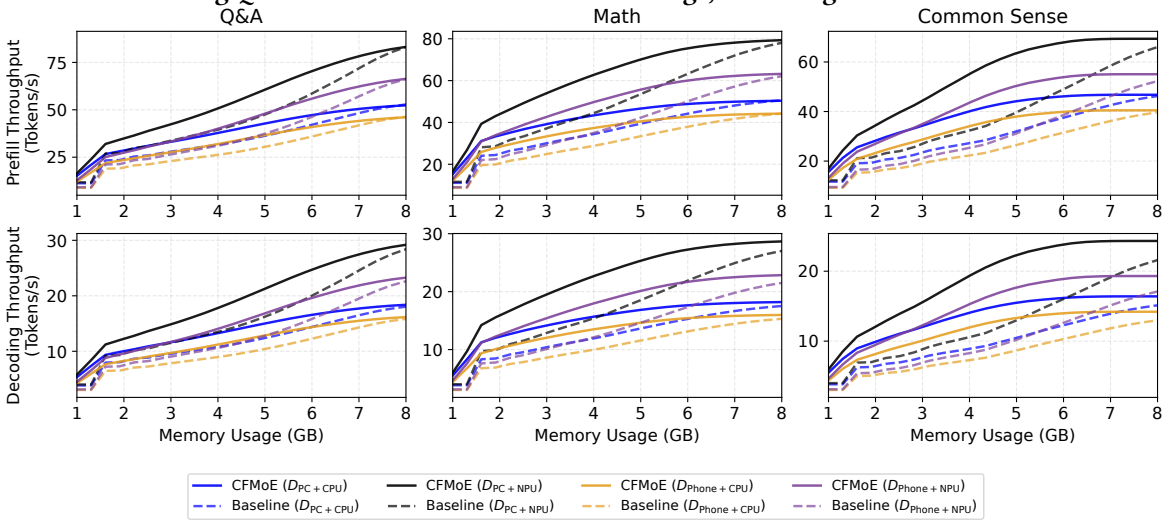


Figure 12: Prefill and decoding throughput under different memory budgets across the three dataset categories for all devices using DeepSeek-MoE-16B, including the LoD baseline and CFMoE.

Furthermore, we observe that the throughput improvements of CFMoE vary across benchmark categories. In particular, the gains are smaller on Q&A benchmarks than on Math and Common Sense benchmarks. A likely reason is that Q&A tasks impose stronger constraints on routing behavior, leaving less flexibility for CAC to improve locality. Consequently, CFMoE yields smaller ECR gains on Q&A workloads, which in turn limits the achievable improvement.

Model Precision. We evaluate the impact of CFMoE 's cache affinity constraint on model precision across representative benchmarks, including Q&A, summarization, math problem solving, and common sense reasoning. As shown in Table 3, for DeepSeek-MoE-16B, CFMoE preserves model precision relative to LoD and MoE-prefetch (0.529 vs. 0.526 on average), whereas MoE-compression results in an average score decrease of 0.030. We attribute the slight precision gains observed on several datasets partly to the semantic similarity among tokens within the same request: by enhancing the locality of expert routing, CFMoE encourages

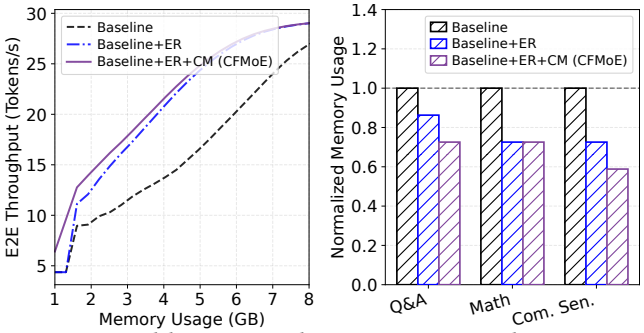


Figure 13: Ablation results using DeepSeek-MoE-16B on $D_{\text{PC+NPU}}$. The left panel shows throughput speedup across memory budgets. The right shows normalized memory usage needed to sustain the baseline throughput at a 2 GB budget.

consecutive tokens to be assigned to similar experts, which may promote expert specialization and model generalization. For Qwen3-30B-A3B, CFMoE similarly preserves model precision relative to LoD and MoE-prefetch (0.689 vs. 0.691).

Memory Usage. We further measure LoD throughput over architecture-specific memory budgets on all devices, and then compute the memory required by CFMoE to sustain the same throughput. For DeepSeek-MoE-16B, using budgets of 2, 4, 6, and 8 GB, CFMoE reduces memory demand by 32.9% on average across all benchmarks and devices. In particular, with a 4 GB memory budget, CFMoE reduces memory usage by more than 60% on MMLU and SIQA benchmarks across all devices. For Qwen3-30B-A3B, using budgets of 4, 8, 12, and 16 GB, CFMoE similarly reduces memory usage by 24.2% on average. These results demonstrate that CFMoE significantly reduces memory usage without compromising serving throughput or model precision, thereby making MoE serving more practical on edge devices.

5.3 Ablation Study

To evaluate the contribution of each component, we compare three variants: the baseline, the baseline with training-phase optimizations, and the full CFMoE design with both training- and inference-phase optimizations. Figure 13 reports the results on $D_{\text{PC+NPU}}$. The training-phase optimization alone improves average throughput by 1.27×, while the inference-phase optimization provides a further 1.10× gain. Under a 2 GB memory budget, the training-phase optimization yields an average 1.34× improvement across all benchmarks, and the inference-phase optimization contributes another 1.11×. Similar trends are observed on the other devices.

We also adopt the same methodology to evaluate memory usage. The training-phase optimization reduces memory demand by 22.9% on average, and the inference-phase optimization provides a further 9.1% reduction while sustaining the same throughput. These results show that both phases contribute materially to CFMoE, and together enable efficient on-device deployment of MoE models.

5.4 Sensitivity Analysis

We evaluate the sensitivity of DAFT to the CAC weight α and the layer-selection threshold β using DeepSeek-MoE-16B. We vary α from 0.1 to 1.0 while fixing $\beta = 0.7$, and vary β from 0.5 to 1.0 while fixing $\alpha = 0.5$. The LoRA rank is fixed at $r = 8$. For every setting, we measure the average task-specific score and ECR across the 12 datasets.

Increasing α initially improves routing locality and serving throughput, but an excessively large value places too much emphasis on cache affinity and begins to affect model precision. In particular, increasing α from 0.1 to 0.5 improves the average ECR from 0.592 to 0.681 and increases serving throughput by 19.2%, while changing the average task score by only 0.1%. Further increasing it to 1.0 yields only 0.005 additional ECR improvement but decreases the task score by 1.8%. This motivates our choice of $\alpha = 0.5$. We also observe stable convergence under all evaluated values: across 5 random seeds, the standard deviation of the final training loss remains below 0.0138.

The threshold β controls the fraction of low-locality layers adapted by DAFT and therefore governs the trade-off between fine-tuning cost and routing locality. Increasing β from 0.5 to 1.0 increases the average number of adapted MoE layers from 0.75 to 27, improving ECR by 0.114 but increasing the number of trainable parameters by 3500%. At $\beta = 0.7$, DAFT adapts 20.42 of the 27 MoE layers on average and reduces trainable parameters by nearly 30% relative to applying LoRA uniformly, while preserving precision (Table 3).

5.5 Fine-Tuning Overhead

We quantify the one-time costs of routing profiling and DAFT on a single NVIDIA A100 GPU with 80 GB memory. For each task-specific adapter, we sample 30% of the training dataset as the calibration subset; validation and test datasets are never used. Profiling requires only a forward pass through the frozen model to record layer-wise routing locality and incurs no gradient computation. Across the 12 datasets, profiling DeepSeek-MoE-16B takes 21.5 min on average (3.6–71.5 min), with a peak memory consumption of 68.76 GB.

DAFT uses AdamW with a learning rate of 1e-4, a batch size of 32, and 3 training epochs. With $\alpha = 0.5$, $\beta = 0.7$, and LoRA rank $r = 8$, it freezes the backbone and updates an adapter with fewer than 1% of the parameters of the original model. Fine-tuning DeepSeek-MoE-16B takes 20.13 h on average (9.52–69.91 h) with one NVIDIA A100 GPU. The resulting adapters occupy 464.54 MB on average, ranging from 22.75 MB to 614.23 MB across the 12 datasets. For Qwen3-30B-A3B, profiling and DAFT take 24.08 min and 24.76 h,

Table 3: The ECR and model precision of the baselines and CFMoE on 12 benchmark datasets using DeepSeek-MoE-16B. "Task" denotes the primary task, and "Metric" denotes the evaluation metric for model precision. The middle three columns report the shared baseline ECR, the ECR of CFMoE, and the improvement of CFMoE over the baselines. The rightmost four columns report model precision for LoD and MoE-prefetch (LoD/Pref.), MoE-compression (MoE-Comp.), and CFMoE, followed by the improvement of CFMoE over LoD.

Dataset	Task	Metric	ECR			Model Precision			
			Baselines	CFMoE	Improv.	LoD/Pref.	MoE-Comp.	CFMoE	Improv.
Alpaca	Q&A	Rouge-L	0.786	0.855	+ 0.070	0.186	0.178	0.198	+ 0.007
WikiText	Q&A	Rouge-L	0.385	0.522	+ 0.137	0.128	0.124	0.145	+ 0.011
SAMSum	Q&A	Rouge-L	0.497	0.510	+ 0.011	0.198	0.186	0.186	− 0.007
GSM8K	Math	Accuracy	0.619	0.754	+ 0.136	0.405	0.385	0.410	+ 0.005
MaWPS	Math	Accuracy	0.619	0.742	+ 0.123	0.780	0.720	0.795	+ 0.015
Mathqa	Math	Accuracy	0.732	0.798	+ 0.066	0.534	0.505	0.520	− 0.014
Aqua	Math	Accuracy	0.711	0.759	+ 0.049	0.290	0.280	0.303	+ 0.013
BoolQ	Com. Sen.	Accuracy	0.320	0.456	+ 0.136	0.730	0.678	0.726	− 0.004
PIQA	Com. Sen.	Accuracy	0.597	0.727	+ 0.130	0.850	0.811	0.855	+ 0.005
SIQA	Com. Sen.	Accuracy	0.400	0.666	+ 0.266	0.844	0.793	0.847	+ 0.003
OBQA	Com. Sen.	Accuracy	0.389	0.573	+ 0.184	0.842	0.786	0.829	− 0.013
MMLU	Com. Sen.	Accuracy	0.715	0.814	+ 0.100	0.525	0.510	0.530	+ 0.005

respectively, and produce a 413.44 MB adapter. Compared with applying rank-8 LoRA to every MoE layer, DAFT reduces fine-tuning time by 19.3% and adapter size by 24.4%.

These offline costs are incurred only once when an adapter is prepared and are amortized over subsequent inference requests. CFMoE supports a global adapter trained on a balanced aggregation dataset as well. During inference, if a request carries a label, the inference engine routes it to the corresponding adapter. Otherwise, a lightweight request-level MLP router selects one with an accuracy of 93.4%. The router occupies 4.22 MB and incurs 0.02 ms of latency.

6 Discussion

Broader Impact. Compared with existing on-device MoE deployment solutions, which mainly focus on model compression and expert prefetching, we explore a new direction: restructuring the routing module to fundamentally improve the runtime cache hit rate. This shift in perspective may inspire future research on adaptive expert routing policies co-designed with hardware caching behaviors, cross-layer consistency in sparse activations, and dynamic specialization of experts based on memory hierarchy awareness.

Generalizability and expert expressiveness. CFMoE is broadly applicable without requiring task-specific fine-tuning. DAFT can be performed once at the backbone level on generic corpora and then reused across downstream tasks. Additionally, when training MoE models from scratch, CAC can even directly replace conventional load-balancing regularization, removing the need for DAFT altogether. Moreover, CFMoE does not sacrifice expert expressiveness as CAC acts only as a soft auxiliary loss. Therefore, it promotes temporal coherence within semantically consistent spans without overriding the main task objective during semantic shifts. This allows routing to remain adaptive and preserves expert diversity, avoiding expert collapse.

Impact of Device Characteristics. The performance gains of CFMoE are shaped by both device I/O bandwidth and compute capability: they tend to become more pronounced on devices with stronger compute capability or lower I/O bandwidth, where expert-loading I/O accounts for a larger fraction of end-to-end latency. On both the PC and the phone, CFMoE achieves larger average speedups with the NPU than with the CPU because faster NPU computation increases the relative contribution of expert-loading I/O to end-to-end latency. Consequently, reducing expert loading eliminates a larger fraction of end-to-end latency and yields a greater relative speedup. Across device types, the gains are similar, with slightly higher average speedups on the phone.

7 Conclusion

This paper presents our effort to address the memory bottleneck of deploying MoE-based LLMs on resource-constrained edge devices. Existing solutions reduce memory usage at the expense of model precision or serving throughput. We address this challenge with CFMoE, an end-to-end serving system that introduces coordinated optimizations across model architecture, fine-tuning strategy, and inference engine design. Our results show that CFMoE reduces runtime memory usage by 32.9% without sacrificing model precision or serving throughput. In a broader sense, our work provides a practical path toward efficient and scalable deployment of MoE-based LLMs on edge devices.

References

[1] Rishabh Agarwal, Nino Vieillard, Yongchao Zhou, Piotr Stanczyk, Sabela Ramos, Matthieu Geist, and Olivier Bachem. 2024. On-Policy Distillation of Language Models: Learning from Self-Generated Mistakes. In *Proc. of ICLR*.

[2] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2014. Language Models Are Few-Shot Learners. In *Proc. of MIT Press NIPS*. 1877–1901.

[3] Damai Dai, Chengqi Deng, Chenggang Zhao, R. X. Xu, Huazuo Gao, Deli Chen, Jiashi Li, Wangding Zeng, Xingkai Yu, Y. Wu, Zhenda Xie, Y. K. Li, Panpan Huang, Fuli Luo, Chong Ruan, Zhifang Sui, and Wenfeng Liang. 2024. DeepSeekMoE: Towards Ultimate Expert Specialization in Mixture-of-Experts Language Models. arXiv:2401.06066 [cs]

[4] DeepSeek-AI, Daya Guo, Dejian Yang, and et al. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv:2501.12948 [cs]

[5] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. 2022. Understanding Modern Storage APIs: A Systematic Study of Libaio, SPDK, and Io_uring. In *Proc. of ACM SYSTOR*. 120–127.

[6] Ivan Dokmanic, Reza Parhizkar, Juri Ranieri, and Martin Vetterli. 2015. Euclidean Distance Matrices: Essential Theory, Algorithms, and Applications. *IEEE Signal Processing Magazine* 32 (2015), 12–30.

[7] Nan Du, Yanping Huang, Andrew M. Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, Barret Zoph, Liam Fedus, Maarten Bosma, Zongwei Zhou, Tao Wang, Yu Emma Wang, Kellie Webster, Marie Pellat, Kevin Robinson, Kathleen Meier-Hellstern, Toju Duke, Lucas Dixon, Kun Zhang, Quoc V. Le, Yonghui Wu, Zhifeng Chen, and Claire Cui. 2022. GLaM: Efficient Scaling of Language Models with Mixture-of-Experts. In *Proc. of ACM ICLM*. 5547–5569.

[8] Zhixu Du, Shiyu Li, Yuhao Wu, Xiangyu Jiang, Jingwei Sun, Qilin Zheng, Yongkai Wu, Ang Li, Hai "Helen" Li, and Yiran Chen. 2024. SiDA-MoE: Sparsity-Inspired Data-Aware Serving for Efficient and Scalable Large Mixture-of-Experts Models. In *Proc. of Machine Learning and Systems*. 224–238.

[9] Tyna Eloundou, Sam Manning, Pamela Mishkin, and Daniel Rock. 2023. GPTs Are GPTs: An Early Look at the Labor Market Impact Potential of Large Language Models. arXiv:2303.10130 [econ]

[10] Zhiyuan Fang, Zicong Hong, Yuegui Huang, Yufeng Lyu, Wuhui Chen, Yue Yu, Fan Yu, and Zibin Zheng. 2025. Fate: Fast Edge Inference of Mixture-of-Experts Models via Cross-Layer Gate. arXiv:2502.12224 [cs]

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

[12] Elias Frantar and Dan Alistarh. 2023. Sparsegpt: Massive language models can be accurately pruned in one-shot. In *Proc. of ACM ICML*. 10323–10337.

[13] Trevor Gale, Deepak Narayanan, Cliff Young, and Matei Zaharia. 2023. Megablocks: Efficient sparse training with mixture-of-experts. *Proc. of ACM MLSys* 5 (2023), 288–304.

[14] Github.com. 2025. Gpt-Oss Series Model. https://github.com/openai/gpt-oss.

[15] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, and et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs]

[16] Yuxian Gu, Li Dong, Furu Wei, and Minlie Huang. 2024. MiniLLM: Knowledge Distillation of Large Language Models. arXiv:2306.08543 [cs]

[17] John R Hershey and Peder A Olsen. 2007. Approximating the Kullback Leibler divergence between Gaussian mixture models. In *Proc. IEEE ICASSP*. IV–317.

[18] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. 2022. Lora: Low-rank adaptation of large language models.. In *Proc. of ICLR*.

[19] huggingface.co. 2025. Hunyuan-0.5B-Instruct Model. https://huggingface.co/tencent/Hunyuan-0.5B-Instruct.

[20] huggingface.co. 2025. Hunyuan-1.8B-Instruct Model. https://huggingface.co/tencent/Hunyuan-1.8B-Instruct.

[21] huggingface.co. 2025. Hunyuan-4B-Instruct Model. https://huggingface.co/tencent/Hunyuan-4B-Instruct.

[22] huggingface.co. 2025. Hunyuan-7B-Instruct Model. https://huggingface.co/tencent/Hunyuan-7B-Instruct.

[23] Changho Hwang, Wei Cui, Yifan Xiong, Ziyue Yang, Ze Liu, Han Hu, Zilong Wang, Rafael Salas, Jithin Jose, Prabhat Ram, et al. 2023. Tutel: Adaptive mixture-of-experts at scale. *Proceedings of Machine Learning and Systems* 5 (2023), 269–287.

[24] Ranggi Hwang, Jianyu Wei, Shijie Cao, Changho Hwang, Xiaohu Tang, Ting Cao, and Mao Yang. 2024. Pre-gated moe: An algorithm-system co-design for fast and scalable mixture-of-expert inference. In *Proc. of ACM/IEEE ISCA*. 1018–1031.

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

[26] Peng Jin, Bo Zhu, Li Yuan, and Shuicheng Yan. 2024. MoE++: Accelerating Mixture-of-Experts Methods with Zero-Computation Experts. arXiv:2410.07348 [cs]

[27] Bosung Kim, Kyuhwan Lee, Isu Jeong, Jungmin Cheon, Yeojin Lee, and Seulki Lee. 2025. On-Device Sora: Enabling Training-Free Diffusion-based Text-to-Video Generation for Mobile Devices. arXiv:2502.04363 [cs]

[28] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2020. GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding. arXiv:2006.16668 [cs]

[29] Jiamin Li, Yimin Jiang, Yibo Zhu, Cong Wang, and Hong Xu. 2023. Accelerating distributed {MoE} training and inference with lina. In *Proc. of USENIX ATC*. 945–959.

[30] Liunian Harold Li, Jack Hessel, Youngjae Yu, Xiang Ren, Kai-Wei Chang, and Yejin Choi. 2024. Symbolic Chain-of-Thought Distillation: Small Models Can Also "Think" Step-by-Step. arXiv:2306.14050 [cs]

[31] Xudong Lu, Qi Liu, Yuhui Xu, Aojun Zhou, Siyuan Huang, Bo Zhang, Junchi Yan, and Hongsheng Li. 2024. Not All Experts are Equal: Efficient Expert Pruning and Skipping for Mixture-of-Experts Large Language Models. In *Proc. of ACL*. 6159–6172.

[32] Xinyin Ma, Gongfan Fang, and Xinchao Wang. 2023. Llm-pruner: On the structural pruning of large language models. In *Proc. MIT Press NIPS*. 21702–21720.

[33] Sven Mayer, Gierad Laput, and Chris Harrison. 2020. Enhancing Mobile Voice Assistants with WorldGaze. In *Proc. of ACM CHI*. 1–10.

[34] Thang M. Pham, Phat T. Nguyen, Seunghyun Yoon, Viet Dac Lai, Franck Dernoncourt, and Trung Bui. 2024. SlimLM: An Efficient Small Language Model for On-Device Document Assistance. arXiv:2411.09944 [cs]

[35] Mohammad Siavashi, Faezeh Keshmiri Dindarloo, Dejan Kostic, and Marco Chiesa. 2025. Priority-Aware Preemptive Scheduling for Mixed-Priority Workloads in MoE Inference. In *Proc. of ACM EuroMLSys*. 132–138.

[36] Xiaoniu Song, Zihang Zhong, Rong Chen, and Haibo Chen. 2025. ProMoE: Fast MoE-based LLM Serving Using Proactive Caching. arXiv:2410.22134 [cs]

[37] Mingjie Sun, Zhuang Liu, Anna Bair, and J Zico Kolter. 2024. A Simple and Effective Pruning Approach for Large Language Models. In *Proc. of ICLR*.

[38] Yehui Tang, Yichun Yin, Yaoyuan Wang, Hang Zhou, Yu Pan, and et al. 2025. Pangu Ultra MoE: How to Train Your Big MoE on Ascend NPUs. arXiv:2505.04519 [cs]

[39] Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, and et al. 2024. Gemini 1.5: Unlocking Multimodal Understanding across Millions of Tokens of Context. arXiv:2403.05530 [cs]

[40] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, and et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs]

[41] S.-C. Tsai, W.-G. Tzeng, and H.-L. Wu. 2005. On the Jensen–Shannon Divergence and Variational Distance. *IEEE Transactions on Information Theory* 51 (2005), 3333–3336.

[42] Haodong Wang, Qihua Zhou, Zicong Hong, and Song Guo. 2025. D^{2}MoE: Dual Routing and Dynamic Scheduling for Efficient On-Device MoE-based LLM Serving. arXiv:2504.15299 [cs]

[43] Chenpeng Wu, Qiqi Gu, Heng Shi, Jianguo Yao, and Haibing Guan. 2025. Samoyeds: Accelerating MoE Models with Structured Sparsity Leveraging Sparse Tensor Cores. In *Proc. of ACM EuroSys*. 293–310.

[44] Shaohua Wu, Jiangang Luo, Xi Chen, Lingjun Li, Xudong Zhao, Tong Yu, Chao Wang, Yue Wang, Fei Wang, Weixu Qiao, et al. 2024. Yuan 2.0-m32: Mixture of experts with attention router. *arXiv preprint arXiv:2405.17976* (2024).

[45] Daliang Xu, Hao Zhang, Liming Yang, Ruiqi Liu, Gang Huang, Mengwei Xu, and Xuanzhe Liu. 2025. Fast on-device LLM inference with npus. In *Proc. of ACM ASPLOS*. 445–462.

[46] Fuzhao Xue, Zian Zheng, Yao Fu, Jinjie Ni, Zangwei Zheng, Wangchunshu Zhou, and Yang You. 2024. Openmoe: An early effort on open mixture-of-experts language models. *arXiv preprint arXiv:2402.01739* (2024).

[47] Leyang Xue, Yao Fu, Zhan Lu, Luo Mai, and Mahesh Marina. 2025. MoE-Infinity: Efficient MoE Inference on Personal Machines with Sparsity-Aware Expert Cache. arXiv:2401.14361 [cs]

[48] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, and et al. 2025. Qwen3 Technical Report. arXiv:2505.09388 [cs]

[49] Jingfeng Yang, Hongye Jin, Ruixiang Tang, Xiaotian Han, Qizhang Feng, Haoming Jiang, Shaochen Zhong, Bing Yin, and Xia Hu. 2024. Harnessing the Power of LLMs in Practice: A Survey on ChatGPT and Beyond. *ACM Transactions on Knowledge Discovery from Data* 18, 6 (July 2024), 1–32.

[50] Rongjie Yi, Liwei Guo, Shiyun Wei, Ao Zhou, Shangguang Wang, and Mengwei Xu. 2025. EdgeMoE: Empowering Sparse Large Language Models on Mobile Devices. arXiv:2308.14352 [cs]

A Routing and Profiling Details

A.1 Routing Metrics

CFMoE uses two complementary routing-locality metrics. ECR measures the overlap between the discrete expert sets activated by consecutive tokens and is used in our evaluation, whereas Jensen–Shannon divergence compares their routing probability distributions and provides the differentiable per-layer locality used by CAC during training.

Expert consistency rate. The expert consistency rate (ECR) is the average Jaccard similarity between the activated-expert sets of consecutive tokens:

$$\mathrm{ECR} = \frac{1}{T-1}\sum_{t=2}^{T}\frac{|\mathcal{A}(t)\cap\mathcal{A}(t-1)|}{|\mathcal{A}(t)\cup\mathcal{A}(t-1)|},$$

where $\mathcal{A}(t)$ denotes the expert set activated by token t, and T is the total number of tokens in the sequence.

Per-layer routing locality. For tokens t and $t-1$ in sequence s, let $A^{(s)}(t), A^{(s)}(t-1) \in \mathbb{R}^{M\times N}$ be their expert activation probability matrices, where M is the number of MoE layers and N is the number of experts per layer. The locality of the k-th layer at token t is:

$$\begin{aligned}\mathcal{L}_k^{(s)}(t) &= JS\Big(A_{k,1:N}^{(s)}(t) \parallel A_{k,1:N}^{(s)}(t-1)\Big)\\ &= \frac{1}{2}\sum_{j=1}^{N} A_{k,j}^{(s)}(t)\cdot\ln\left(\frac{A_{k,j}^{(s)}(t)}{\frac{1}{2}\left[A_{k,j}^{(s)}(t)+A_{k,j}^{(s)}(t-1)\right]}\right)\\ &\quad+\frac{1}{2}\sum_{j=1}^{N} A_{k,j}^{(s)}(t-1)\cdot\ln\left(\frac{A_{k,j}^{(s)}(t-1)}{\frac{1}{2}\left[A_{k,j}^{(s)}(t)+A_{k,j}^{(s)}(t-1)\right]}\right).\end{aligned}$$

A.2 Profiles and Benchmarks

This section reports the device, benchmark, and layer-wise profiles referenced in the main text.

Device profiles. Table 4 reports the per-layer computation time and per-expert transfer time used to characterize the four evaluation devices.

Table 4: Per-layer computation time and per-expert data transfer time (Flash→DRAM and DRAM→xPU) of DeepSeek-MoE-16B on different devices.

Device	Compute (ms)	Flash→DRAM (ms)	DRAM→xPU (ms)
$D_{\text{PC+CPU}}$	1.18	1.42	0.12
$D_{\text{PC+NPU}}$	0.71	1.52	0.07
$D_{\text{Phone+CPU}}$	1.32	1.70	0.14
$D_{\text{Phone+NPU}}$	0.92	1.85	0.09

Benchmark statistics. Table 5 summarizes the benchmark-level routing-locality measurements used in our motivation and evaluation.

Table 5: Twelve benchmarks involved in our measurement. The rightmost three columns denote the CHR of DeepSeek-MoE-16B on each benchmark (with 2 GB cache size and OPT policy), the average ECR, and the Pearson correlation between CHR and ECR.

Dataset	CHR (2GB)	ECR	Pearson
Alpaca	0.850	0.743	0.999
WikiText	0.753	0.580	0.999
SAMSum	0.809	0.676	0.999
GSM8K	0.772	0.618	0.999
MaWPS	0.795	0.661	0.999
Mathqa	0.820	0.690	0.997
Aqua	0.800	0.656	0.997
BoolQ	0.788	0.647	0.999
PIQA	0.809	0.685	0.997
SIQA	0.716	0.535	0.999
OBQA	0.789	0.644	0.999
MMLU	0.834	0.720	0.999

Layer-wise locality. Figure 14 complements the benchmark-level statistics with the layer-wise routing-locality profiles used by DAFT.

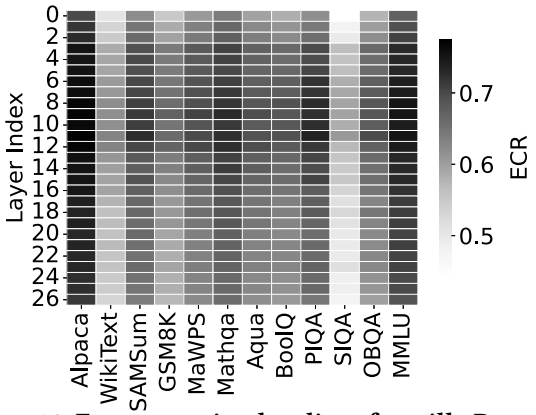


Figure 14: Expert routing locality of vanilla DeepSeek-MoE-16B on different benchmarks and layers.