eLLM: Elastic Memory Management Framework for Efficient LLM Serving

Jiale Xu*
Shanghai Jiao Tong University
Shanghai Qi Zhi Institute
China
jerry_xu@sjtu.edu.cn

Rui Zhang*
Ant Group
China
george.zr@antgroup.com

Yi Xiong*
Ant Group
China
alex.xy@antgroup.com

Cong Guo†
Shanghai Jiao Tong University
Shanghai Qi Zhi Institute
China
guocong@sjtu.edu.cn

Zihan Liu
Shanghai Jiao Tong University
Shanghai Qi Zhi Institute
China
altair.liu@sjtu.edu.cn

Yangjie Zhou
National University of Singapore
Singapore
yj_zhou@nus.edu.sg

Weiming Hu
Shanghai Jiao Tong University
Shanghai Qi Zhi Institute
China
weiminghu@sjtu.edu.cn

Hao Wu
Ant Group
China
wh391609@antgroup.com

Changxu Shao
Ant Group
China
shaochangxu.scx@antgroup.com

Ziqing Wang
Ant Group
China
serina.wzq@antgroup.com

Yongjie Yuan
Ant Group
China
yuanyongjie.yyj@antgroup.com

Junping Zhao†
Ant Group
China
junping.zjp@antgroup.com

Minyi Guo
Shanghai Jiao Tong University
Shanghai Qi Zhi Institute
China
guo-my@cs.sjtu.edu.cn

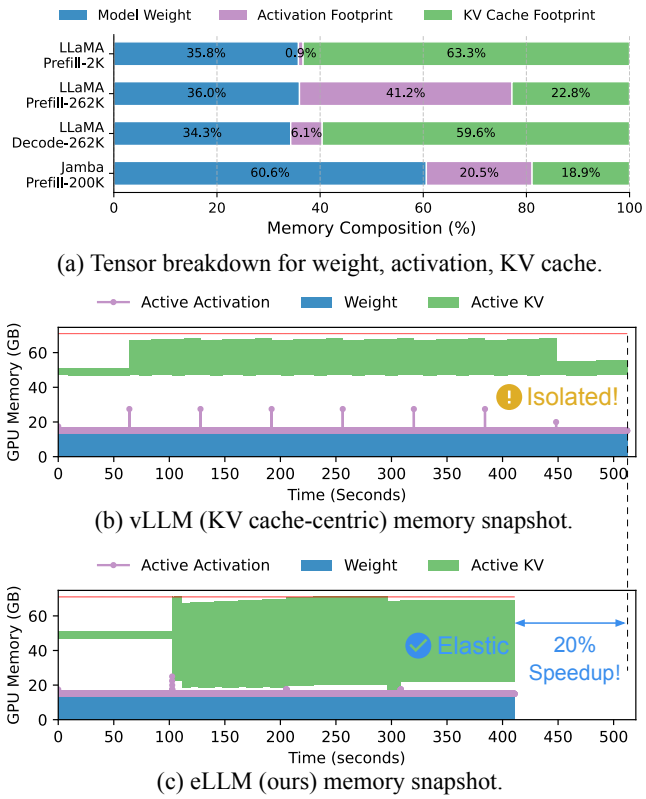
Jingwen Leng†
Shanghai Jiao Tong University
Shanghai Qi Zhi Institute
China
leng-jw@cs.sjtu.edu.cn

Abstract

Large Language Models (LLMs) are increasingly being deployed in datacenters. Serving these models requires careful memory management, as their memory usage includes static weights, dynamic activations, and key-value (KV) caches. While static weights are constant and predictable, dynamic components such as activations and KV caches change frequently during runtime, presenting significant challenges for efficient memory management. Modern LLM serving systems typically handle runtime memory and KV caches at distinct abstraction levels: runtime memory management relies on static tensor abstractions, whereas KV caches utilize a page table-based virtualization layer built on top of the tensor abstraction. This virtualization dynamically manages KV caches to mitigate memory fragmentation. However, this dual-level approach fundamentally isolates runtime memory and KV cache management, resulting in suboptimal memory utilization under dynamic workloads. Consequently, this isolation can lead to a nearly 20% drop in throughput.

To address these limitations, we propose eLLM, an elastic memory management framework inspired by the classical memory ballooning mechanism in operating systems. The core components of eLLM include: (1) Virtual Tensor Abstraction, which decouples the virtual address space of tensors from the physical GPU memory, creating a unified and flexible memory pool; (2) an Elastic Memory Mechanism that dynamically adjusts memory allocation through runtime memory inflation and deflation, leveraging CPU memory as an extensible buffer; and (3) a Lightweight Scheduling Strategy employing SLO-aware policies to optimize memory utilization and effectively balance performance trade-offs under stringent SLO constraints. Comprehensive evaluations demonstrate that eLLM significantly outperforms state-of-the-art systems, 2.32× higher decoding throughput, and supporting 3× larger batch sizes for 128K-token inputs.

*These authors contributed equally to this work.
†Corresponding author.



(a) Tensor breakdown for weight, activation, KV cache.

(b) vLLM (KV cache-centric) memory snapshot.

(c) eLLM (ours) memory snapshot.

Figure 1. In serving LLaMA3-8B-262K (32 requests with 32768-2048 length) on a single NVIDIA A100 (80GB): (a) Memory footprint breakdown; (b) vLLM's isolated allocation for activations and KV cache in separate spaces causes underutilization and suboptimal performance; (c) eLLM enables dynamic memory allocation, maximizing utilization and achieving a 1.2× speedup.

1 Introduction

Large Language Models (LLMs) have drawn widespread attention for their advanced semantic understanding and human-like reasoning, transforming modern industry and daily life. Mainstream models like GPT [18] and DeepSeek [6] are now widely deployed in cloud services, processing tens of millions of requests daily.

However, compared to earlier, more compact DNNs such as CNN [11] and BERT [7], LLMs pose significant memory management challenges that lead to GPU underutilization and limited throughput. Firstly, LLMs have a dramatically increased memory footprint. For example, LLaMA-70B [27] contains 648× more parameters than BERT [7], and its peak memory usage can reach up to 168 GB due to extensive activation tensors, even though a single NVIDIA A100 GPU provides only 80 GB.

Secondly, LLM inference exhibits significant dynamic behavior with irregular memory allocation. The auto-regressive mechanism [18] requires storing all generated key and value tensors as a dynamically growing KV cache, in contrast to the static weight and activation tensors managed by frameworks like PyTorch [19]. In early LLM systems [32], static pre-allocation based on maximum request length for the KV cache causes severe fragmentation [14]. vLLM [14] first addressed the dynamic KV cache challenge by introducing the PagedAttention mechanism, which mitigates memory fragmentation and has become a de facto standard in modern LLM service systems [3, 24, 29, 35]. Although vLLM shows that handling dynamic aspects can boost throughput, its approach is limited to the KV cache while isolating activation memory allocation. The broader dynamic challenges in both activation and KV cache tensors call for a comprehensive reevaluation of memory management strategies in modern LLM systems.

Generally, both the activation and KV cache exhibit significant dynamic behavior in LLM serving workloads. As shown in Fig. 1 (a), we classify tensors into three categories: weight, activation, and KV cache.

First, **variations in request length** substantially affect the memory footprint of tensors. Compared to a 2K input (1st row in Fig. 1 (a)), LLaMA-3-8b with a 262K input (2nd row) uses a much larger fraction of activation tensors during the prefill phase. Since request lengths are unpredictable during serving, existing frameworks (including vLLM [14]) pre-allocate activation tensors based on the maximum possible length to avoid dynamic allocation overhead. Although this was less problematic with shorter requests (e.g., <8K tokens for LLaMA-2 [27]), the rising demand for long-content inference has led to notable GPU memory underutilization.

Second, **variations in inference phases** cause significant fluctuations in activation usage. For instance, Fig. 1 (a) shows that LLaMA3-8b with a 262K input (2nd row) drops the activation proportion from over 40% to under 10% (3rd row) as it transitions from prefill to decoding. Similarly, that can lead to memory underutilization when conducting two phases in a single GPU. A solution like DistServe [35] splits prefill and decoding onto separate GPUs, but this increases resource demands and communication overhead while still using static activation allocation and PageAttention. Thus, it does not fundamentally resolve the memory issue.

Furthermore, recent model architectures (e.g., GQA [4], MLA [6], Jamba [15]) have focused on KV cache compression to reduce memory pressure in long-context scenarios, enabling models to serve on fewer GPUs. Specifically, Fig. 1 (a) (4th row) indicates that KV cache memory of Jamba model [15] now accounts for only 19%, while activation memory has increased nearly dozens-fold, from 0.3% for the 2K-length model (see Section 3.1). This shift suggests that the bottleneck has moved from being KV cache-centric to involving all dynamic tensors.

Existing solutions have some limitations. Chunked prefill [3] partitions prompts into chunks and batched with

decoding, reduced the dynamics of memory workload, but incurs severe inefficiency in long-context inference due to KV cache read amplification [2]. Prefill-decoding disaggregation schemes (e.g., DistServe [35], Splitwise [20], Mooncake [23]) and dynamic parallelism paradigms (e.g., LoongServe [29], Llumnix [24]) often relies on introducing redundant parameter replicas on multiple GPUs to achieve elasticity.

Fundamentally, we identified that the root cause of the suboptimal memory utilization for activation tensors lies in the isolation between the runtime memory space and the KV cache space, as shown in Fig. 1(b). Since the three types of tensors are isolated, an insurmountable gap is created between their memory spaces, ultimately diminishing memory utilization efficiency and system performance.

To address these limitations, we propose **eLLM**, a flexible memory management framework inspired by the OS-level mechanism known as memory ballooning—a feature widely used in virtualization platforms that allows the host system to dynamically expand its effective memory pool by reclaiming unused memory from guest virtual machines. On a logical level, eLLM continues to abstract KV caches and activations as distinct logical entities, enabling specialized memory management strategies tailored to their differing access patterns, thereby improving efficiency. On a physical level, eLLM unifies all physical memory into a shared pool that can be dynamically allocated between KV caches and activation memory, reducing fragmentation and maximizing utilization. Finally, as shown in Fig. 1 (c), the activation and KV caches can "borrow" memory from each other, which directly translates into an approximate 20% performance improvement in the LLaMA-3-8b model. This result demonstrates the effectiveness and advantages of complete dynamic memory management.

In its design, eLLM introduces two types of virtual tensor, which decouple the virtual address space of tensor objects from underlying physical memory chunks. Building upon this tensor abstraction, eLLM designed the elastic memory mechanism that dynamically adjusts memory allocation through inflation and deflation operations. Additionally, it leverages the CPU as an extensible buffer for GPU memory, enhancing memory flexibility and alleviating GPU memory pressure. Finally, eLLM proposes a lightweight scheduling strategy that optimizes memory allocation, task scheduling, and resource transitions by coordinating elastic memory management. Extensive evaluations across diverse models and workloads demonstrate eLLM's substantial improvements, achieving up to a 2.32× increase in offline inference throughput.

In summary, we make following contributions:

- **Identifying LLM memory bottlenecks**: A system-level analysis quantifies the impact of KV cache and activation memory isolation, revealing 1.2–2.3× lower throughput.
- **Proposing elastic tensor abstraction**: By decoupling virtual address from physical memory and unifying memory management, eTensor enables KV cache and activation memory sharing, overcoming isolation constraints.
- **Designing elastic memory mechanisms**: A dynamic inflation/ deflation mechanism boosts GPU memory utilization, while a CPU-based elastic buffer and an SLO-driven scheduling strategy further optimize memory management, reducing the request queueing time.
- **Comprehensive evaluations**: Comprehensive evaluations across diverse models and workloads demonstrate substantial improvements in eLLM, achieving 295× lower TTFT, 2.32× higher decoding throughput, and 3× larger batch size in 128k/8k long context inference.

2 Background

2.1 LLM Inference

The decoder-only Transformer architecture [18], composed of stacked Transformer layers integrating an attention mechanism and a token-wise feed-forward network (FFN), serves as the foundational design for most LLMs today. The success of this architecture is primarily attributed to its causal attention mechanism, which restricts each token's attention to preceding tokens, thereby enabling auto-regressive [18] generation and ensuring coherent outputs.

Within this architecture, the inference process of LLMs is divided into two phases. In the prefill phase, input tokens are processed to create corresponding KV caches and generate the first output token. During the decoding phase, each new token is generated based on the previous token and the accumulated KV cache. This approach eliminates the need to reprocess all preceding tokens during self-attention computations [21]. Two key service-level objective (SLO) metrics characterize these phases: Time To First Token (TTFT), which measures the latency from input arrival to the generation of the first output token, and Time Per Output Token (TPOT), which represents the average time between consecutive tokens [35].

2.2 Memory Management in LLM Serving Systems

2.2.1 Memory Efficiency. GPU memory has become a critical bottleneck restricting the high-performance deployment of LLM inference services [9]. Memory constraints not only limit the deployability of models but also directly impact both system throughput and response latency [14].
1) The memory-bound nature of auto-regressive generation in LLMs [18] leads to inefficient utilization of GPU resources. As a result, modern LLM serving systems employ advanced batching strategies, aggregating multiple requests into a single batch to increase computational intensity and amortize framework overhead [32]. However, the attainable batch size is fundamentally constrained by the available GPU memory, thereby limiting overall system throughput.

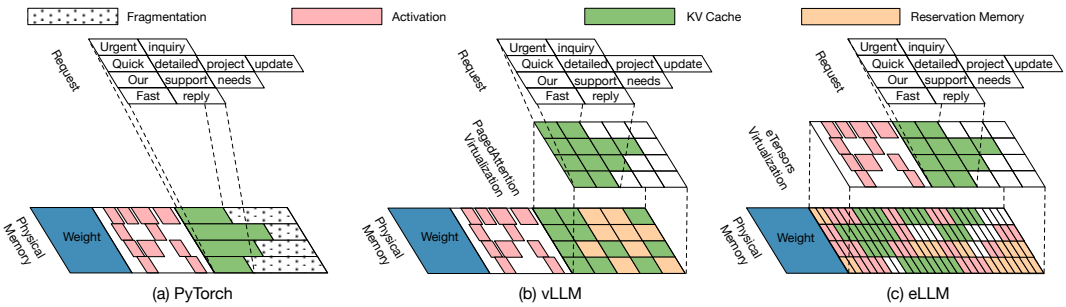


Figure 2. (a) Early LLM Systems (e.g., PyTorch [19]) use static tensor model, failing to handle dynamic KV cache expansion, causing fragmentation. (b) By virtualizing the KV cache, vLLM alleviates memory fragmentation but separates activations and the KV cache into different abstraction levels, thereby isolating activation from the KV cache space. (c) eLLM independently manages the KV cache and activations at the logical level while consolidating memory into a unified physical pool, thereby maximizing memory utilization.

2) When the available GPU memory falls short of accommodating incoming requests, these requests are enqueued, pending the completion of ongoing tasks and the release of enough resources. This process may incur considerable queuing delays, thereby exacerbating response latency [24].

2.2.2 Memory Management Mechanisms.

The memory models in early LLM serving systems had limitations. Early LLM serving systems [32] inherited the static tensor model abstracted from the deep learning (DL) framework [19]. They allocated fixed-size physically contiguous memory blocks for tensors and employed a pool-based memory reuse strategy for management. This management approach proves efficient and reasonable for activations and parameters, as the sizes of their instantiated tensors remain constant throughout their lifetime [14]. However, the KV cache has unique memory patterns, including dynamic expansion and unpredictable lifecycles and sequence lengths. Conventional memory management schemes inevitably lead to severe fragmentation, as illustrate in Figure 2(a).

PagedAttention has become the de facto standard in modern LLM serving systems [14]. vLLM addresses this limitation by introducing the PagedAttention mechanism. It introduces a virtualized abstraction specifically for KV cache, managed through a page table mechanism that eliminates the requirement for contiguous memory allocation, as illustrate in Figure 2(b). During service initialization, GPU memory is pre-allocated based on the model's maximum request length to satisfy runtime memory requirements (primarily for activations and parameters). The remaining memory is organized into pre-configured KV blocks designed to sufficiently accommodate dynamic KV cache demands. By enhancing memory efficiency in GPU memory-bound workloads, PagedAttention directly improves computational throughput, establishing itself as a standard approach in mainstream LLM serving systems.

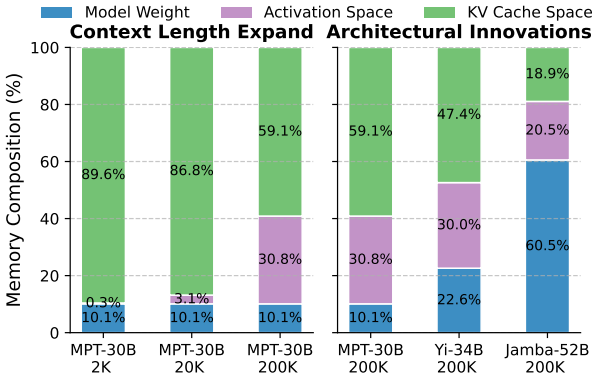


Figure 3. Shifting of available memory composition (with NVIDIA A100 80GB GPUs).

3 Motivation

Although modern LLM serving systems, e.g., vLLM [14], achieve nearly fragmentation-free dynamic management of KV cache space, challenges still remain in achieving overall memory efficiency.

3.1 Dynamic Memory in LLM

Variations in Request Length. As model context lengths surge from thousands [27] to millions [10, 16] (even reaching billion-level scales [8]), memory composition undergoes fundamentally shifts. Figure 3 left panel reveals that extending context length from 2K to 200K in same architectures causes activation space to escalate from 0.3% to 30.8%, while KV cache space dominance collapses from 89.6% to 59.1%.

Model architectural innovations. Architectural innovations (e.g., GQA [4], MLA [6], Jamba [15]) mitigate long context memory pressure through efficient KV caching. These designs slash GPU requirements for long contexts. For example, MPT-30B (8 GPUs) → Yi-34B (4 GPUs) Jamba-52B (2 GPUs) with a context of 200k, although their parameter

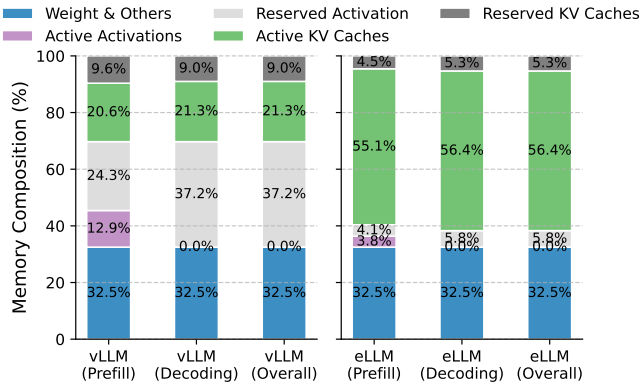


Figure 4. Comparison of memory utilization between vLLM and eLLM (same config as Figure 1).

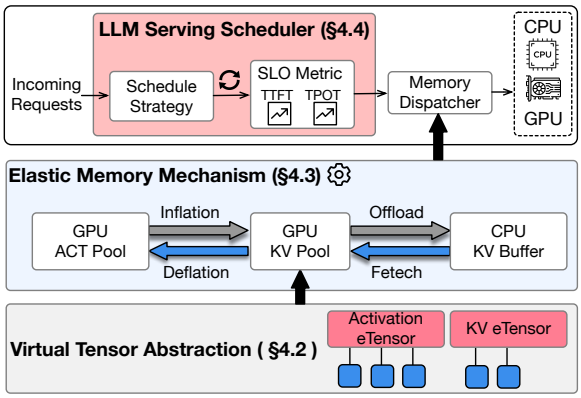


Figure 5. Overview of eLLM system.

counts have increased. Figure 3 right panel illustrates how the ratio of activation space to KV cache space size has reversed from 0.52 to 1.08 as the architecture evolves.

We posit that the shift in memory composition, driven by the interaction between expanding context length and architectural innovations, is expected to intensify over time.

3.2 Memory Underutilization

Figure 4 left depicts memory utilization patterns in vLLM serving systems under real-world workloads [31]. And Figure 4 right is in our proposed eLLM, which is relatively optimal for memory utilization.

Underutilized Activation Space. We evaluate vLLM with 2K input length with LLaMA3-8B with maximum context length 262K. In prefill phase, merely 35% of the allocated activation memory is active, leaving 65% idle. Notely, in the real dataset [25], over 90% of request batches utilize less than 30% of the model's maximum context length.

In the decoding phase, activation memory utilization falls further to just 1%. Because, unlike the prefill phase, where more tokens are processed at once, each decoding step processes only a small number of tokens. With the widespread adoption of advanced Test-Time Compute techniques, such as CoT [28], decoding cycles have become increasingly prolonged. This extremely low decoding utilization dominates the overall inference process.

Overloaded KV Cache Space. Historical KV cache accumulation: modern serving architectures maintain historical KV caches within shared memory space alongside new requests. The expansion of context windows and the adoption of prefix caching techniques have significantly increased retained KV entries; however, available KV cache space has paradoxically decreased (§3.1). This inefficiency in memory utilization has become a significant bottleneck in modern LLM service systems.

3.3 Root Reason: Memory Isolation.

We identify that memory inefficiency arises from the isolation of activation and KV cache spaces in existing systems. This isolation is analogous to the separation between kernel space and user space in an OS: activations rely on framework-level static tensor abstractions that directly interface with physical memory, while the KV cache is managed by PagedAttention through memory management policies built atop these static abstractions, which are decoupled from physical resources. This isolation hinders the dynamic redistribution of memory resources based on workload demands, leading to underutilized activation memory, which exacerbates KV cache space pressure and ultimately constrains overall system performance.

3.4 Our Solution: Elastic Memory

Memory Ballooning is a classic OS mechanism that relaxes memory isolation, allowing dynamic memory reallocation between host system and virtual machines. Its core mechanism leverages page tables from the OS's virtual memory management system to enable memory reallocation through mapping relationship propagation. Inspired by this concept, we propose eLLM, an elastic memory management framework that dynamically allocates memory resources for the KV cache and activations based on actual demand.

4 Design

In this section, we formally introduce eLLM, the elastic memory management system designed to mitigate memory isolation challenges while optimizing memory utilization.

4.1 eLLM Overview

eLLM consists of three core components: Virtual Tensor Abstraction, Elastic Memory Mechanism, and Lightweight Scheduling Strategy, as illustrated in Figure 5.

First, eLLM proposes a virtualized tensor abstraction for both KV cache and activations, which reflects their respective access characteristics, decouples them from physical

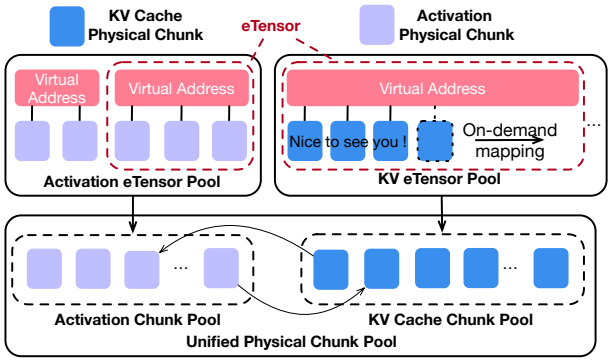


Figure 6. eTensor abstraction for KV cache and activation tensors, dividing the virtual address apart from the physical memory chunks. We can transfer the physical memory chunks because they are identical.

resources, and forms the foundation for elastic memory management. Then, based on this abstraction, eLLM introduces the Elastic Memory Mechanism that dynamically rearranges KV caches and activation memory space through remapping, and employs CPU memory as an elastic buffer to further alleviate the memory pressure on GPU. Finally, by leveraging the elastic mechanism, eLLM designs a lightweight scheduling strategy to efficiently utilize the memory while achieving effective trade-offs under SLO constraints.

4.2 Virtual Tensor Abstraction

eLLM introduces a novel tensor abstraction, called eTensor, tailored for LLM workloads, leveraging GPU Virtual Memory Management (VMM) to bridge the abstraction gap between activations and KV cache in existing LLM serving systems. From kernel function perspective, eTensor can be viewed as an array pointer structure that references a contiguous segment within the GPU's virtual address space. Beneath the virtual address layer, the vmm maintains a complete mapping of physical chunks, enabling compute kernels to access all necessary data stored in global GPU memory. This enables the dynamic propagation of mapping relationships across different tensors.

4.2.1 KV eTensor and Activation eTensor. Given the distinct allocation granularities and usage patterns of KV cache and activations, eTensor introduces two tensor types: KV eTensor and activation eTensor, each supported by specialized strategies to optimize memory utilization and efficiency, as illustrated in Figure 6.

The KV cache is inherently structured as large, regular memory blocks exhibiting stable size expansion, infrequent access patterns, and persistent retention during inference. Consequently, in KV eTensor, a virtual address space segment equal to the model's context length, is pre-allocated for each request at maximum concurrency, which ensures the logical continuity of the KV cache. Physical chunks are allocated on-demand during actual writes, effectively eliminating unnecessary physical memory consumption. Conversely, activations comprise smaller memory blocks characterized by shorter lifespans and higher access frequencies. Therefore, activation eTensors, characterized by non-uniformly sized virtual address segments, inherently require frequent and fine-grained management of the virtual address space. Both types of eTensors structure their virtual address segments (termed tensor slots) to strictly align with the granularity of physical memory chunks, thereby achieving an optimal balance between access efficiency and fragmentation control.

4.2.2 eTensor Pools and Unified Physical Pool. Considering the potential overhead introduced by tensor virtualization due to virtual-to-physical address mappings, eLLM implements tailored memory pool management strategies for each type of eTensor. Rather than immediately unmapping eTensor instances from physical resources at the end of their lifecycle, the system marks them as mapped, available tensor slots and tracks their mapping sizes, thereby enabling efficient memory reuse. The KV eTensor pool employs a Best-Fit algorithm for incoming requests. Given the set of available pre-mapped tensor slots $R = \{r \mid r \in Mapped \wedge r.state = Available\}$ and a target memory size s, the algorithm prioritizes selecting the smallest feasible slot satisfying the requirement $r_i = \arg\min_{r_j \in R} \{\text{size}(r_j) \mid \text{size}(r_j) \geq s\}$. If no such slot exists, on-demand mapping is triggered. The Activation eTensor pool retains the framework's native Best-Fit with Coalescing (BFC) strategy.

Due to the separate pooling mechanism of eTensor, all physical chunks are labeled with their corresponding eTensor category, referred as ownership. However, they essentially belong to a unified physical memory pool. This feature enables zero-overhead identifier conversion only through mapping relationship propagation, allowing eLLM to dynamically allocate resources based on runtime workloads.

4.3 Elastic Memory Mechanism

The elastic memory mechanism aims to enhance memory utilization and system performance under LLM workloads. It introduces two levels of elasticity: intra-GPU memory *inflation* and *deflation*, which enable timely adjustment of physical memory allocation for activation and KV cache in response to the dynamic workloads; and GPU-CPU *offloading* and *fetching*, which employ CPU memory as an elastic buffer to further alleviate the memory pressure on GPU.

4.3.1 Memory Inflation and Deflation. Memory *inflation* and *deflation* operations are the core mechanism of eLLM, enabling it to break memory isolation through mapping relation propagation, which fundamentally involves dynamically maintaining the binding relationship between virtual address spaces and physical memory chunks, built upon the eTensor abstraction.

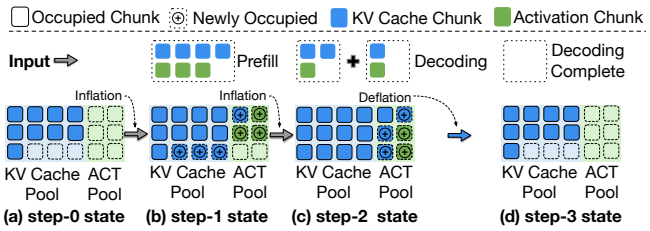


Figure 7. Illustrative example of memory inflation/deflation.

Inflation operation dynamically expands the physical memory capacity of the KV cache by borrowing from the active memory pool, much like inflating a balloon. This process involves the following steps: ❶ Inflation trigger: upon KV cache allocation requests, the system first verifies whether the KV memory pool contains sufficient physical memory chunks. If insufficient, a borrowing request is issued to the activation memory pool. ❷ Memory reclamation: the active pool fulfills the request by triggering a lightweight garbage collection (GC) phase. This GC phase identifies and unmaps physical memory chunks allocated to inactive eTensor objects. ❸ Ownership transfer: reclaimed chunks are logically migrated from the activation pool to the KV cache pool through ownership transfer. ❹ On-demand remapping: the GPU virtual memory manager dynamically maps these chunks to target KV eTensors' virtual address spaces. Deflation operation is the reverse process of the above.

Figure 7 demonstrates the intra-GPU elastic memory mechanism through an illustrative example. (a) Initially, the GPU retains historical KV caches. In this state, existing LLM serving systems are unable to process newly arrived requests immediately due to insufficient KV cache space. (b) With memory inflation, eLLM can handle prefill requests promptly by borrowing idle activation chunks. (c) Through memory inflation, eLLM enables larger batch processing even under high memory pressure, thereby improving computational resource utilization given the memory-bound nature of the decoding phase. (d) With memory deflation, eLLM can return the borrowed memory. In practice, this process is triggered lazily to avoid unnecessary overhead.

4.3.2 Memory Offloading and Fetching.

eLLM further incorporates elastic memory management between GPU-CPU, reducing service responsiveness pressure under high memory stress. In online long-context LLM serving, optimizing system response latency (i.e., TTFT) faces more challenges compared to TPOT: (1) more memory contention leading to severe request queuing delays, and (2) the computational complexity of the prefill stage scaling superlinearly. Therefore, eLLM enables proactively offloading KV cache for part of requests to CPU DRAM during the prefill stage. This lowers the memory admission barrier for request execution, effectively reducing queuing delays to improve TTFT. Additionally, it has the potential to aggregate larger decoding batch sizes, thereby increasing throughput.

Its technical feasibility is based on following observations. Firstly, newly generated KV caches not immediately needed can be offloaded proactively and fetched back only when their corresponding request's decoding is scheduled. Secondly, the multi-layer structure of Transformers naturally supports overlapping computation and communication through layer-wise pipelining. Thirdly, migration of KV cache requires only $O(N)$ linear communication overhead, whereas the prefill stage's self-attention computation incurs $O(N^2)$ complexity. For example, in practical workloads with A100 GPUs and the LLaMA-3-8B model, offloading overhead can be completely hidden by computation. However, while CPU buffer improves request queuing conditions, it inherently introduces a prefill-prefer tendency that adversely affects TPOT. Therefore, eLLM introduces a simple yet effective policy to trade-off between SLO metrics (§4.4.2).

4.4 Lightweight Scheduling Strategy

To address the memory inefficiency (§3.2) in LLM serving, eLLM introduces a lightweight yet effective scheduling algorithm, which leverages its elastic memory mechanism to dynamically allocate memory resources and optimize system performance. The basic scheduling strategy adopts a simple heuristic approach, relying solely on metadata updates and binary decisions to maximize memory utilization under constrained memory conditions. Furthermore, by incorporating a concise SLO-aware buffer scaling policy, eLLM effectively balances the trade-off between TTFT and TPOT, ensuring SLO compliance for latency-sensitive online services.

4.4.1 Request Scheduling.

To avoid potential deadlock risks from resource borrowing, eLLM eliminates the "hold and wait" condition. All essential KV cache and activation memory resources required for the current iteration must be simultaneously allocated.

Algorithm 1 presents the core scheduling strategy for eLLM, which maximizes the number of concurrent LLM service requests by elastically allocating resources, while strictly adhering to the memory constraints. Capitalizing on the substantial variation in activation memory demands across different phases of LLM inference, we use phase-specific resource allocation strategies: 1) Prefill phase: This phase requires significant activation and KV cache memory resources. During memory-intensive scenarios, we implement KV cache offloading to CPU buffers, thereby reducing GPU memory pressure. 2) Decoding phase: Given the minimal activation memory requirements in this phase, we optimize resource utilization by fetching KV cache back into GPU memory, taking advantage of the reduced memory footprint. Both phases will maximize GPU memory utilization through inflation and deflation.

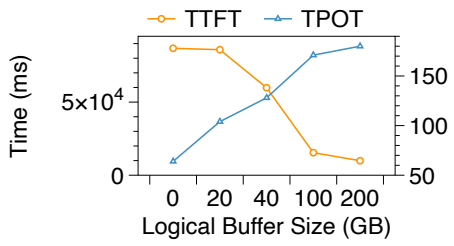


Figure 8. SLO metrics with different CPU buffer size.

Algorithm 1: Scheduling with Elastic Memory.

Input: Free KV cache physical chunks: P_{kv};
Free Activation physical chunks: P_{act};
Total physical chunks: P_T; Pending requests: Q;
Memory threshold: θ; Available CPU buffer: P_B.
Output: Inflation amount: I (> 0: $act \rightarrow kv$, < 0: $kv \rightarrow act$); Batched requests: B;

$B \leftarrow \emptyset, O \leftarrow \emptyset, I \leftarrow 0, M_{kv} \leftarrow 0, M_{act} \leftarrow 0$
if *Prefill Phase* **then**
  foreach $r \in Q$ **do**
    $ACT_r \leftarrow requiredAct(r), KV_r \leftarrow requiredKV(r)$
    if $P_T - (M_{kv} + M_{act} + KV_r + ACT_r) \geq \theta$ **then**
      $B \leftarrow B \cup r, M_{kv} \leftarrow M_{kv} + KV_r,$
       $M_{act} \leftarrow M_{act} + ACT_r$
    else if $P_T - (M_{kv} + M_{act} + ACT_r) \geq \theta$
     $KV_r \leq P_B$ **then**
      $B \leftarrow B \cup r, M_{act} \leftarrow M_{act} + ACT_r$
      $P_B \leftarrow P_B - KV_r$ `// Offloading.`
    else
      break
if *Decoding Phase* **then**
  foreach $r \in Q$ **do**
    $ACT_r \leftarrow requiredAct(r), KV_r \leftarrow requiredKV(r)$
    `// Fetching KV cache if swapped out.`
    if $P_T - (M_{kv} + M_{act} + KV_r + ACT_r) \geq \theta$ **then**
      $B \leftarrow B \cup r, M_{kv} \leftarrow M_{kv} + KV_r,$
       $M_{act} \leftarrow M_{act} + ACT_r$
    else
      break
`// Memory Balloning:`
if $P_{kv} < M_{kv}$ $P_{act} > M_{act}$ **then**
  $I \leftarrow M_{kv} - P_{kv}$ `// Inflation.`
else if $P_{act} < M_{act}$ ***and*** $P_{kv} > M_{kv}$ **then**
  $I \leftarrow P_{act} - M_{act}$ `// Deflation.`
$update(P_{kv}, P_{act})$
return B, I

4.4.2 SLO-aware Buffer Scaling Policy.

Figure 8 illustrates the trade-off between TTFT and TPOT under varying CPU buffer sizes. To achieve an optimal balance between them, it is essential to determine an appropriate buffer size. However, a fixed-size buffer is suboptimal due to the significant impact of LLM's dynamic workload. To address this, eLLM introduce a concept of logical buffer, an abstraction of the physical buffer that dynamically adjusts the actual usable size within its fixed capacity. This enables flexible adjustment of buffer space without evicting stored data, enhancing system adaptability to dynamic workloads.

Algorithm 2: SLO-Aware Logical Buffer Scaling

Input: TTFT violation event (True/False): E_{TTFT};
TPOT violation event (True/False): E_{TPOT};
CPU physical memory size: $B_{\max}$;
Current logical buffer size (initialized to 1): B_{logic};
Buffer tuning factor: α.
Output: Updated logical buffer size: B_{logic}.

if E_{TPOT} **then**
  $B_{logic} \leftarrow \max(B_{logic}/\alpha, 1)$
else if E_{TTFT} **then**
  $B_{logic} \leftarrow \min(B_{logic} * \alpha, B_{logic})$
return B_{logic}

Building on this, eLLM proposes an SLO-aware logical buffer scaling algorithm, as shown in Algorithm 2. If a TPOT violation is detected, the logical buffer size is reduced to limit prefill requests, thereby optimizing TPOT. Conversely, if a TTFT violation is detected, the logical buffer size is increased to optimize TTFT. In eLLM, a violation event is triggered if the TTFT or TPOT exceeds the predefined SLO threshold three times within a specific scheduling iteration window (empirically set to 5 iterations). The buffer tuning factor α is a hyperparameter (default α = 2) that controls the rate of buffer size adjustment.

5 Implementation

5.1 Overlapping (Un)mapping Overheads

To further overlap (up)mapping overhead, we introduce several techniques, including decoding speculative pre-mapping and asynchronous unmapping.

Decoding speculative pre-mapping. Due to the autoregressive nature where each sequence generates only one token at a iteration, we leverage branch speculation principles to assume each sequence will produce the next token and proactively initiate asynchronous memory allocation in advance. This approach effectively masks mapping overhead through minimal additional memory mapping (<50MB), achieving full overlap of decoding mapping costs.

Asynchronous unmapping. When unmapping a tensor slot is triggered, the system does not require immediate unmapping of the slot followed by reassignment of its associated physical chunk. Instead, by leveraging the GPU VMM's capability to map a single physical chunk to multiple virtual addresses, the physical chunk can be initially assigned to a

new tensor slot. The unmapping of the original slot can then be performed asynchronously, effectively overlapping the unmapping overhead.

5.2 eLLM System Implementation

eLLM is built on top of vLLM v0.5.5, incorporating approximately 4000 lines of C++ and Python code. To support KV eTensor, we developed a C++ library and integrated it into vLLM, replacing the original KV Cache tensor object, and modified the minimum allocation granularity from a page to a physical memory chunk. For activation eTensor management, we modify the caching allocator of PyTorch, replacing traditional memory allocation methods such as `cudaMalloc` with GPU VMM APIs. These changes provide more efficient memory handling and better resource utilization. Additionally, we developed a C++ library to achieve elastic memory management.

6 Evaluation

6.1 Experimental Setup

Baselines. We compare eLLM with the following state-of-the-art LLM serving systems:

- **vLLM[14]** : A state-of-the-art LLM serving system widely adopted in industry. vLLM employs efficient KV cache management via PagedAttention and statically allocates memory for activations and KV cache during initialization. Furthermore, we have thoroughly evaluated the performance of vAttention [22], which is nearly identical to that of vLLM. Therefore, the comparison between vLLM and eLLM reflects the performance difference between vAttention and eLLM.
- **vLLM with Chunked Prefill[3]** : vLLM integrates chunked prefill optimization as an optional feature, segmenting prefill requests into smaller chunks (default: 512 tokens) and batching them together with decoding requests that implicitly minimize the pre-allocated activation memory. Due to compatibility constraints, chunked prefill is not applicable to Jamba [15]. We denote the optimized vLLM variant as **vLLM-CP** in our evaluations.
- **DistServe [35]**: A Prefill-Decode disaggregation system designed for online serving. It implicitly improves memory efficiency by assigning distinct phases to different devices, thereby minimizing the activation memory allocation during decoding.

Models. We evaluate eLLM on three popular models with distinct architectures:

- **Llama3-8B-262K [26]**: A representative Grouped Query Attention (GQA) [4] model supports chunked prefill [3];
- **Jamba [15]** : A novel hybrid Mamba-MoE architecture that achieves superior performance and efficiency but does not support chunked prefill yet due to its unique design;
- **OPT-13B [33]**: A standard Multi-Head Attention (MHA) model compatible with DistServe [35].

Testbed. We evaluate eLLM and all baseline models on a server configured with eight NVIDIA A100 80GB GPUs and Intel(R) Xeon(R) Platinum 8369B CPU with 1 TB RAM. These GPUs are connected via NVLink thus don't have the PCIe contention when transferring data between GPU and CPU. We use PyTorch 2.4.0, CUDA Driver 12.4, and vLLM v0.5.5 for the evaluation. Moreover, we explore the performace of Prefill-decoding disaggregation system on a server with two NVIDIA L40S (48GB) GPUs and Intel Xeon(R) Gold 6462C with 2 TB RAM.

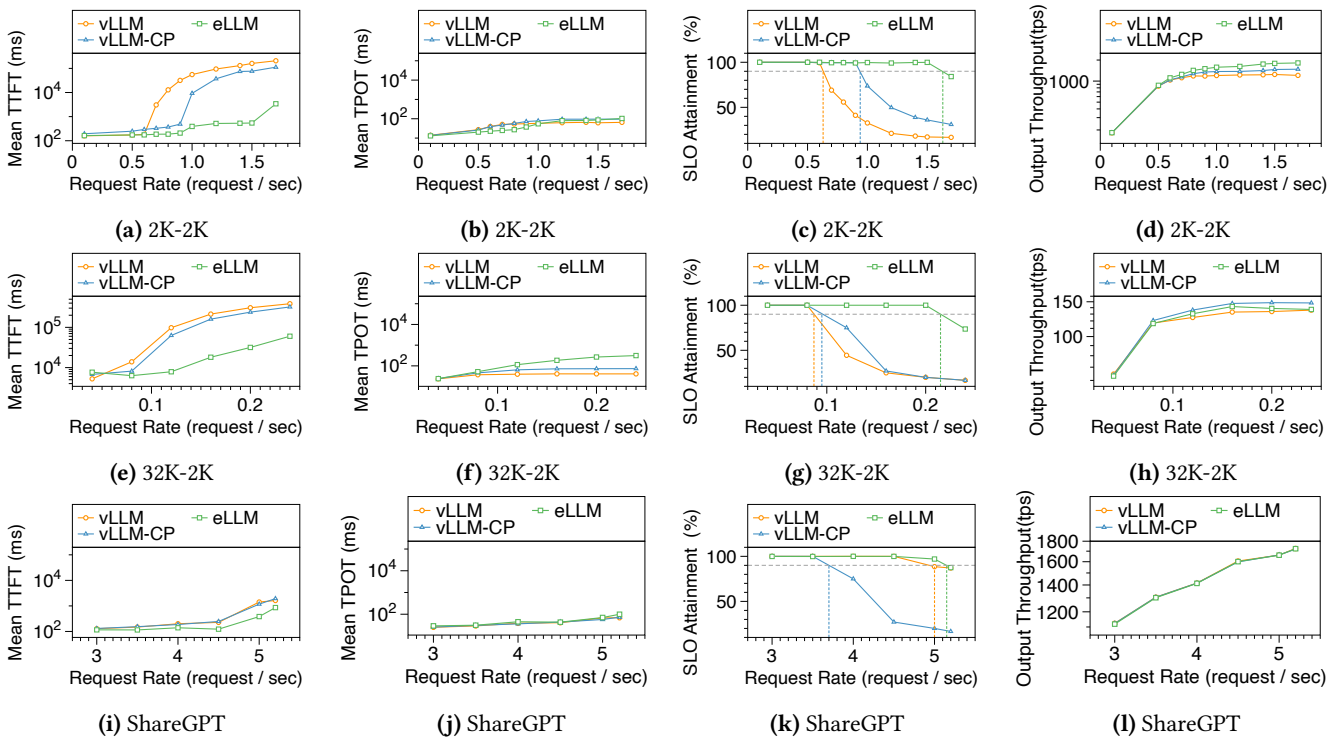
Workloads. We comprehensively evaluate eLLM under both online serving and offline inference scenarios employing popular real-world dataset (shareGPT [25]) and synthetic datasets (fixed length with 2k-2k, 32k-2k). In the online serving scenario, the server is configured to issue requests following a Poisson process at varying arrival rates. In contrast, during offline inference, all requests are arrived at the beginning of the experiment [14].

Metrics. We evaluate eLLM on diverse metrics to demonstrate its effectiveness in LLM serving scenarios. For the online serving evaluation, we leverage TTFT, TPOT, output throughput, and goodput to fully illustrate the effectiveness of eLLM. The Goodput is defined as the maximum request rate that achieves 90% SLO attainment rate [35]. Following the prior works [29], we set the SLO constraint as $25 \times$ TTFT and TPOT of request finished with no request contention (i.e. request rate 0.01 in our evaluation). For the offline inference scenario, we focus on the metric of total throughput, decode throughput, and maximum batch size.

6.2 Online Serving Evaluation on Single GPU

Figure 9 illustrates the online serving evaluation of Llama3-8B on a single A100 (80GB) GPU under various workloads. Figure 9a, Figure 9e, and Figure 9i showcase the TTFT performance of eLLM under different workloads. eLLM consistently outperforms vLLM across all workloads, with the most significant improvement observed in the 2k-2k workload, where eLLM achieves up to 295× and 140× faster TTFT compared to vLLM and vLLM-CP respectively. This improvement is attributed to eLLM's ability to dynamically adjust memory capacity between activations and KV cache, thereby reducing the queueing delay of prefill reqests.

Specifically, in Figure 9a, vLLM-CP and eLLM both exhibit its ability to reduce queueing time because vLLM-CP reserve a small fraction of activation memory, leading to more available KV Cache. However, with the request rate increasing, vLLM-CP's performance degrades due to further mechanism to reduce the queueing time. In contrast, eLLM not only make full use of the intra-GPU elasticity but also leverage the GPU-CPU elasticity to reduce the queueing time, leading to the best performance. The larger decoding batch of eLLM also contributes to the improved output, as shown in Figure 9h, Figure 9l, and Figure 9d.



(a) 2K-2K **(b)** 2K-2K **(c)** 2K-2K **(d)** 2K-2K

(e) 32K-2K **(f)** 32K-2K **(g)** 32K-2K **(h)** 32K-2K

(i) ShareGPT **(j)** ShareGPT **(k)** ShareGPT **(l)** ShareGPT

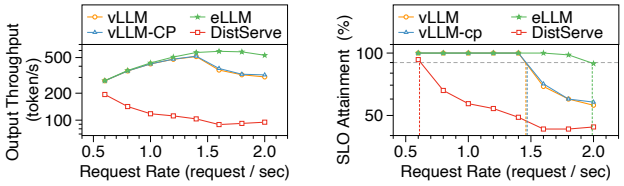
Figure 9. Online serving evaluation with SLO-constraints on Llama3-8B-262K model with one A100 (80GB) GPU. **1)** Fig (a)(b)(c)(d) is conducted on input 2k output 2k workload. **2)** Fig (e)(f)(g)(h) is conducted on input 32k output 2k workload. **3)** Fig (i)(j)(k)(l) is conducted on ShareGPT workload.

For TPOT metric, because eLLM and vLLM-CP has more available KV Cache memory for decoding, their TPOT is higher than vLLM in all workloads within the SLO constraints. Overall, the trade-off of eLLM between TTFT and TPOT leads to a better SLO attainment and upto 2.5× and 2.26× higher goodput compared to vLLM and vLLM-CP in Figure 9c, Figure 9g, and Figure 9k. When it comes to the ShareGPT workload in Figure 9i, eLLM achieves lower enhancement compared to previous workloads due to the relatively small input and output size, which limits the benefits of eLLM's elasticity memory features.

6.3 Online Serving Evaluation on Multiple GPUs

We evaluated the online performance of eLLM in parallel execution scenarios, as shown in Figure 10, which allowed us to include DistServe with prefill-decoding disaggregation for comparison. For eLLM, vLLM, and vLLM-CP, the degree of tensor parallelism was set to 2, whereas for DistServe, we configured one prefill instance and one decoding instance.

Our observations indicate that eLLM consistently achieves the highest performance across all tested configurations. Although DistServe theoretically allows decoding instances to minimize pre-allocated activation space, its actual performance remains suboptimal due to two main factors: First, DistServe executes prefill and decoding phases separately on dedicated GPUs. Consequently, computational resources on one GPU are underutilized when the corresponding phase is idle. Second, model weights in DistServe are replicated across GPUs, resulting in additional consumption of KV cache space and a reduction in the effective batch size during the decoding phase. In contrast, vLLM and vLLM-CP achieve higher SLO attainment rates by simultaneously leveraging both GPUs for prefill and decoding phases and avoiding redundant storage of model weights in GPU memory.



(a) OPT-13B **(b)** OPT-13B

Figure 10. SLO attainment and goodput evaluation with SLO-constraint, which is conducted on OPT-13B model with two L40S 48GB, P=1, D=1 for DistServe and TP=2 for other systems. The dataset for OPT-13B is synthetic with fixed input of 1024 tokens and output length 512 tokens.

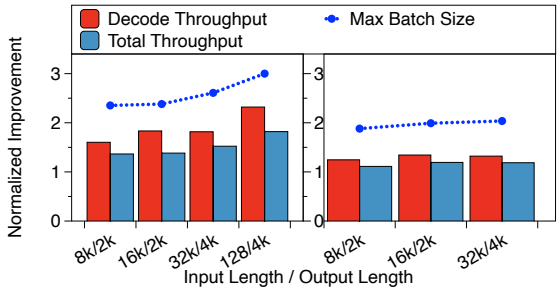


Figure 11. The normalized performance of the total throughput, the decode throughput and the max batch size when varying the input and output size compared to vLLM. The left figure showcases the performance evaluation of Jamba-Mini [15] on 2 A100 (80GB) GPUs, while the right figure presents the corresponding analysis for Llama3-8B-262K using a single A100 (80GB) GPU.

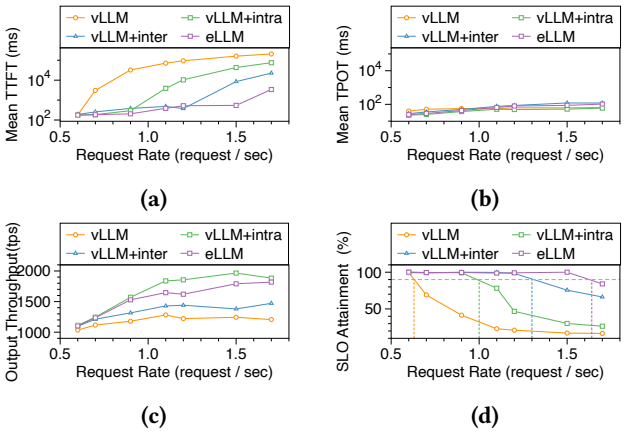


Figure 12. Ablation study on different features of eLLM. This experiment is conducted on 2K-2K workload that in the online serving evaluation with the same settings.

6.4 Offline Inference Evaluation

Figure 11 illustrates the normalized performance of eLLM compared to vLLM on multiple metrics in the offline inference scenarios under different data distributions. eLLM achieves better performance in total throughput and decode throughput. The core reason is that eLLM can flexibly schedule plenty of idle activation memory to the KV Cache in the decoding phase, effectively increasing the capacity of the KV Cache and supporting the larger batch size. With the increase of the input and output size, the performance enhancement of eLLM is more significant. The main reason is that eLLM can accommodate a large batch of requests when the input token number is small. Under the 128k/8k data distribution, eLLM has the best performance which increases the total throughput and the decode throughput by 1.82× and 2.32× respectively compared to vLLM.

Figure 11 compares the offline performance of eLLM and vLLM across various configrations. With a higher activation-to-KV-size ratio, Jamba achieves greater throughput improvement, indicating better KV cache utilization. As input sequence length increases, vLLM's performance degrades rapidly due to memory constraints, while eLLM sustains larger batch sizes by efficiently borrowing memory, leading to significant throughput gains. Notably, under the 128k-8k data distribution with the Jamba architecture, eLLM achieves optimal performance, improving total and decode throughput by 1.82× and 2.32×, respectively.

6.5 Ablation Study

We conducted an ablation study on the two elastic features of eLLM: intra-GPU elasticity (denoted as `vLLM+intra`) and GPU-CPU elasticity (denoted as `vLLM+inter`). Using vLLM as the baseline, we evaluated their respective effectiveness, with the results presented in Figure 12.

As illustrated in Figure 12a, both the intra-GPU elasticity and GPU-CPU elasticity expand the available memory to accommodate new requests, thereby outperforming the baseline in terms of TTFT. eLLM combines both, achieving a TTFT acceleration of up to 295× compared to the baseline. Meanwhile, as shown in Figure 12b, the introduction of these features maintains relatively stable TPOT.

Figure 12c and Figure 12d illustrate the trade-offs between throughput and SLO compliance within eLLM. With respect to throughput, both the intra-GPU elasticity and GPU-CPU elasticity contribute to an increased decoding batch size, thereby enhancing throughput. However, their combination does not always yield the optimal throughput, primarily due to the overhead of GPU-CPU elasticity's PCIe transfers, which cannot fully be overlapped. For SLO compliance, relying solely on intra-GPU elasticity still results in memory bottlenecks during prefill phase under high request arrival rates, leading to degraded goodput. In contrast, eLLM achieves a better balance by dynamically leveraging both GPU and CPU memory elasticity, resulting in up to a 1.5× improvement in throughput and up to 2.5× goodput improvement.

6.6 System Execution Time Breakdown

To quantify the overhead introduced by eLLM, we break down the overall execution time into three key components: CPU scheduling time, VMM operation time, and model execution time. CPU scheduling time, the overhead from scheduling logic during LLM inference, accounts for less than 1% of total execution time due to eLLM's lightweight scheduling strategy, making its performance impact negligible. VMM operation time refers to the overhead incurred by eLLM's internal calls to VMM API functions, while model execution time refers to the duration of GPU computation during model inference. In online service evaluation scenarios (Figure 9), VMM operation time accounts for 1% to 5% of total

execution time, benefiting from targeted strategy design, including eTensor pooling, decoding speculative pre-mapping, and asynchronous unmapping, thus ensuring moderate overhead and minimal impact on system performance.

7 Related Work

Existing research on long-context LLM memory challenges generally falls into four categories:

Lightweight Algorithms. Prior work reduces memory demands by optimizing attention architecture (e.g., low-rank attention [4, 6], hybrid transformer [15]) and compression techniques (e.g., quantization [13, 17], sparsification [1, 30, 34]). Although effective in reducing memory usage, these approaches often compromise model accuracy. eLLM is orthogonal to these approaches and achieves memory optimization without sacrificing precision.

Efficient Kernels. Existing LLM serving systems utilize optimized GPU kernels that take advantage of memory hierarchies to improve memory access efficiency and reduce memory footprints. Examples include Flash Attention [5] and Flash-Decoding [12]. These techniques are orthogonal to our approach and have been integrated into eLLM.

Memory Management. vLLM [14] introduces PagedAttention, which manages KV cache through page table, enabling non-contiguous memory allocation, reducing memory fragmentation in early systems and has become the de facto standard for modern LLM serving. However, it does not address the issue of memory space isolation, which restricts resource utilization and hinders dynamic workload management. In comparison, eLLM addresses the increasingly severe memory inefficiency issues in long context scenarios and shows potential to become a new paradigm for memory management.

8 Conclusion

Modern LLM serving systems face significant memory utilization bottlenecks due to the isolation of memory spaces. Our proposed eLLM framework addresses this through an elastic memory paradigm inspired by OS-level virtualization techniques. By introducing virtual tensor abstraction to unify memory pools, dynamic inflation/deflation mechanisms to optimize GPU utilization, and CPU-GPU memory orchestration with SLO-aware scheduling, eLLM achieves significantly improvements: 2.32× higher decoding throughput, and supporting 3× larger batch sizes for 128K-token inputs. In the future, eLLM will incorporate the dynamic nature of parameters, enabling more adaptive and efficient learning, in line with current advancements in deep learning trends.

References

[1] Muhammad Adnan, Akhil Arunkumar, Gaurav Jain, Prashant J. Nair, Ilya Soloveychik, and Purushotham Kamath. 2024. Keyformer: KV Cache reduction through key tokens selection for Efficient Generative Inference. In *Proceedings of the Seventh Annual Conference on Machine Learning and Systems, MLSys 2024, Santa Clara, CA, USA, May 13-16, 2024*, Phillip B. Gibbons, Gennady Pekhimenko, and Christopher De Sa (Eds.). mlsys.org.

[2] Amey Agrawal, Junda Chen, Íñigo Goiri, Ramachandran Ramjee, Chaojie Zhang, Alexey Tumanov, and Esha Choukse. 2024. Mnemosyne: Parallelization Strategies for Efficiently Serving Multi-Million Context Length LLM Inference Requests Without Approximations. *CoRR* abs/2409.17264 (2024).

[3] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, Ada Gavrilovska and Douglas B. Terry (Eds.). USENIX Association, 117–134.

[4] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, 4895–4901.

[5] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (Eds.).

[6] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, and Wangding Zeng. 2024. DeepSeek-V3 Technical Report. *CoRR* abs/2412.19437 (2024).

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4171–4186.

[8] Jiayu Ding, Shuming Ma, Li Dong, Xingxing Zhang, Shaohan Huang, Wenhui Wang, Nanning Zheng, and Furu Wei. 2023. LongNet: Scaling Transformers to 1, 000, 000, 000 Tokens. *CoRR* abs/2307.02486 (2023). https://doi.org/10.48550/arXiv.2307.02486

[9] Cong Guo, Rui Zhang, Jiale Xu, Jingwen Leng, Zihan Liu, Ziyu Huang, Minyi Guo, Hao Wu, Shouren Zhao, Junping Zhao, and Ke Zhang. 2024. GMLake: Efficient and Transparent GPU Memory Defragmentation

for Large-scale DNN Training with Virtual Memory Stitching. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir (Eds.). ACM, 450–466. https://doi.org/10.1145/3620665.3640423

[10] Chi Han, Qifan Wang, Hao Peng, Wenhan Xiong, Yu Chen, Heng Ji, and Sinong Wang. 2024. LM-Infinite: Zero-Shot Extreme Length Generalization for Large Language Models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), NAACL 2024, Mexico City, Mexico, June 16-21, 2024*, Kevin Duh, Helena Gómez-Adorno, and Steven Bethard (Eds.). Association for Computational Linguistics, 3991–4008. https://doi.org/10.18653/v1/2024.naacl-long.222

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 770–778.

[12] Ke Hong, Guohao Dai, Jiaming Xu, Qiuli Mao, Xiuhong Li, Jun Liu, Kangdi Chen, Yuhan Dong, and Yu Wang. 2024. FlashDecoding++: Faster Large Language Model Inference with Asynchronization, Flat GEMM Optimization, and Heuristics. In *Proceedings of the Seventh Annual Conference on Machine Learning and Systems, MLSys 2024, Santa Clara, CA, USA, May 13-16, 2024*, Phillip B. Gibbons, Gennady Pekhimenko, and Christopher De Sa (Eds.). mlsys.org.

[13] Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W. Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. 2024. KVQuant: Towards 10 Million Context Length LLM Inference with KV Cache Quantization. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (Eds.).

[14] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace (Eds.). ACM, 611–626. https://doi.org/10.1145/3600006.3613165

[15] Opher Lieber, Barak Lenz, Hofit Bata, Gal Cohen, Jhonathan Osin, Itay Dalmedigos, Erez Safahi, Shaked Meirom, Yonatan Belinkov, Shai Shalev-Shwartz, Omri Abend, Raz Alon, Tomer Asida, Amir Bergman, Roman Glozman, Michael Gokhman, Avashalom Manevich, Nir Ratner, Noam Rozen, Erez Shwartz, Mor Zusman, and Yoav Shoham. 2024. Jamba: A Hybrid Transformer-Mamba Language Model. *CoRR* abs/2403.19887 (2024).

[16] Hao Liu, Wilson Yan, Matei Zaharia, and Pieter Abbeel. 2024. World Model on Million-Length Video And Language With Blockwise RingAttention. *CoRR* abs/2402.08268 (2024). https://doi.org/10.48550/arXiv.2402.08268

[17] Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. 2024. KIVI: A Tuning-Free Asymmetric 2bit Quantization for KV Cache. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

[18] OpenAI. 2023. GPT-4 Technical Report. *CoRR* abs/2303.08774 (2023). https://doi.org/10.48550/arXiv.2303.08774

[19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 8024–8035.

[20] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *51st ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2024, Buenos Aires, Argentina, June 29 - July 3, 2024*. IEEE, 118–132.

[21] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2023. Efficiently Scaling Transformer Inference. In *Proceedings of the Sixth Conference on Machine Learning and Systems, MLSys 2023, Miami, FL, USA, June 4-8, 2023*, Dawn Song, Michael Carbin, and Tianqi Chen (Eds.). mlsys.org.

[22] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. 2025. vAttention: Dynamic Memory Management for Serving LLMs without PagedAttention. arXiv:2405.04437 [cs.LG] https://arxiv.org/abs/2405.04437

[23] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. Mooncake: Trading More Storage for Less Computation - A KVCache-centric Architecture for Serving LLM Chatbot. In *23rd USENIX Conference on File and Storage Technologies, FAST 2025, Santa Clara, CA, February 25-27, 2025*, Haryadi S. Gunawi and Vasily Tarasov (Eds.). USENIX Association, 155–170.

[24] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. 2024. Llumnix: Dynamic Scheduling for Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, Ada Gavrilovska and Douglas B. Terry (Eds.). USENIX Association, 173–191.

[25] Sharegpt teams. 2023. Sharegot. https://sharegpt.com/

[26] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. *CoRR* abs/2302.13971 (2023). https://doi.org/10.48550/arXiv.2302.13971

[27] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *CoRR* abs/2307.09288 (2023).

[28] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models.

In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (Eds.).

[29] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. 2024. LoongServe: Efficiently Serving Long-Context Large Language Models with Elastic Sequence Parallelism. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, SOSP 2024, Austin, TX, USA, November 4-6, 2024*, Emmett Witchel, Christopher J. Rossbach, Andrea C. Arpaci-Dusseau, and Kimberly Keeton (Eds.). ACM, 640–654. https://doi.org/10.1145/3694715.3695948

[30] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2024. Efficient Streaming Language Models with Attention Sinks. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

[31] Zhangchen Xu, Fengqing Jiang, Luyao Niu, Yuntian Deng, Radha Poovendran, Yejin Choi, and Bill Yuchen Lin. 2024. Magpie: Alignment Data Synthesis from Scratch by Prompting Aligned LLMs with Nothing. *CoRR* abs/2406.08464 (2024). https://huggingface.co/datasets/Magpie-Align/Magpie-Reasoning-V2-250K-CoT-Deepseek-R1-Llama-70B

[32] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, Marcos K. Aguilera and Hakim Weatherspoon (Eds.). USENIX Association, 521–538.

[33] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona T. Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. 2022. OPT: Open Pre-trained Transformer Language Models. *CoRR* abs/2205.01068 (2022).

[34] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark W. Barrett, Zhangyang Wang, and Beidi Chen. 2023. H2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.).

[35] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, Ada Gavrilovska and Douglas B. Terry (Eds.). USENIX Association, 193–210.