JENGA: Effective Memory Management for Serving LLM with Heterogeneity

Chen Zhang[†*], Kuntai Du[‡*], Shu Liu[◇], Woosuk Kwon[◇], Xiangxi Mo[◇], Yufeng Wang[§], Xiaoxuan Liu[◇]
Kaichao You[†*], Zhuohan Li[◇], Mingsheng Long[†], Jidong Zhai[†], Joseph Gonzalez[◇], Ion Stoica[◇]
[†]*Tsinghua University* [‡]*University of Chicago* [◇]*UC Berkeley* [§]*Independent Researcher*

Abstract

Large language models (LLMs) are widely used but expensive to run, especially as inference workloads grow. To lower costs, maximizing the request batch size by managing GPU memory efficiently is crucial. While PagedAttention has recently been proposed to improve the efficiency of memory management, we find that the growing heterogeneity in the embeddings dimensions, attention, and access patterns of modern LLM architectures introduces new challenges for memory allocation.

In this paper, we present JENGA, a novel memory allocation framework for heterogeneous embeddings in LLMs. JENGA tackles two key challenges: (1) minimizing memory fragmentation when managing embeddings of different sizes, and (2) enabling flexible caching and eviction policies tailored to the specific token-dependency patterns of various layers. JENGA employs a two-level memory allocator, leveraging the least common multiple (LCM) of embedding sizes to optimize memory usage and providing APIs to express layer-specific caching logic to enhance memory reuse.

We implemente JENGA on vLLM, a state-of-the-art LLM inference engine, and evaluate it with diverse LLMs, datasets, and GPU configurations. Evaluations show that JENGA improves GPU memory utilization by up to 79.6%, and increases serving throughput by up to 4.92× (1.80× on average).

1 Introduction

The broad adoption of LLMs for services like ChatGPT [13] and GitHub Copilot [17] has driven a surge in demand for GPUs to power LLM serving. Serving LLMs requires many expensive GPUs to achieve the desired service latency objectives. Even hyperscalers like Microsoft struggle to secure sufficient GPU capacity for their LLM-powered services [12]. Because tokens are generated sequentially during inference, LLM serving workloads often fail to adequately utilize the compute capacity of modern GPUs.

[*]Part of the work was done during visiting UC Berkeley.

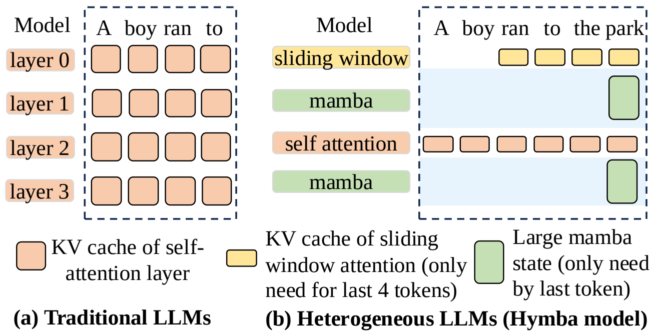

Model
A boy ran to
layer 0
layer 1
layer 2
layer 3
Model
A boy ran to the park
sliding window
mamba
self attention
mamba
KV cache of self-attention layer
KV cache of sliding window attention (only need for last 4 tokens)
Large mamba state (only need by last token)
(a) Traditional LLMs
(b) Heterogeneous LLMs (Hymba model)

Figure 1: Traditional LLMs (left) v.s. Latest LLMs (right). LLMs are becoming more and more heterogeneous and produce KV caches with different sizes and dependencies, which demands a new GPU memory manager design.

One way to reduce the cost of LLM serving is to increase GPU utilization by processing multiple requests in parallel (i.e., batching requests). While batching often has minimal impact on latency, batching introduces a new challenge. Each generated token may depend on the computed embeddings of all previous tokens in the same request. Therefore, we need to store the embeddings for the prefixes of all requests belonging to the same batch in the GPU memory. This makes GPU memory *capacity* the new bottleneck to fully utilizing the GPU compute. Thus, efficient memory management is the key to improving the GPU throughput, and consequently alleviating the high costs of LLM serving.

By borrowing ideas from memory management in operating systems, PagedAttention [28] introduced mechanisms to reduce fragmentation of the embeddings for all previous tokens (KV-Cache). PagedAttention manages a mapping between virtual and physical pages. Each page is a fixed-size multiple of an embedding, allowing for more efficient allocation and minimizing fragmentation. As a result, PagedAttention was able to improve throughput by 2 - 4× compared to previous state-of-the-art solutions [57], and is now used by virtually all LLM inference engines, including vLLM [28], SGLang [59] and TensorRT-LLM [36].

The design of PagedAttention was built on two fundamental assumptions about LLM architectures, which held true when monolithic Transformers [50] were dominant:

1. **Fixed-size embeddings**: Embeddings are the same size in different tokens and layers. As such, the granularity of memory allocation is naturally a fixed page size, which is a multiple of the embedding size.
2. **Full-prefix dependency**: Generating the next token on a request depends on *all* previous tokens of the request. As such, all tokens in the prefix share the same life cycle.

Two years after the introduction of PagedAttention, LLM architectures have evolved to incorporate more *heterogeneous* components (Figure 1), which invalidates both assumptions.

First, recent models often have heterogeneous embeddings with different sizes. For example, vision-language models (VLMs) such as LLaVA [34] and InternVL [11] retain vision embeddings for image inputs as well as KV cache for each token. The sizes of these embeddings are naturally different as they encode different types of information. Moreover, recent approaches [18, 31] combine Mamba [23] with the regular LLM architecture, where the Mamba state representing a sequence is typically larger than the KV cache associated with a single token.

Second, to handle long contexts more efficiently, some new LLM architectures use only a subset of the prefix tokens to generate the next token. We call this token-dependency pattern *prefix-subset dependency*. For instance, Google's Gemma-2 [48] interleaves full attention with sliding window attention [6]: some layers attend to the entire prefix, while other layers only attend to a sliding window of the most recent tokens. Building on this, NVIDIA's Hymba [18] further integrates Mamba layers, which solely rely on the representation of the last token. These architectures invalidate the second assumption that the generation of the next token depends on *all* previous tokens of the query.

As a result, the throughput of PagedAttention can drop by up to 4.91× compared to an ideal solution which only stores the required tokens needed by each layer to compute the next token, and tightly packs different-size embeddings in the KV cache (see Section 7).

To close this gap, we propose JENGA, a new memory management framework for the KV cache. There are two challenges which directly follow from the violation of the two assumptions that JENGA needs to address: (1) minimize the fragmentation when handling different-size embeddings, and (2) customize memory eviction and caching policies for each type of layer to minimize cache misses.

To address the first challenge, JENGA uses a two-level memory allocator. At the bottom layer, we still have fixed-size pages, while at the top layer we have different-size embeddings. To minimize internal fragmentation, we use the know model architecture to pick a compatible page size so that it is a multiple of each embedding size. In particular, we choose the page size as *least common multiple* (LCM) of token embedding sizes. For example, if we have embeddings of two different sizes, 2KB and 3KB, respectively, we pick a page with a compatible size of 6KB. We call such an allocator *LCM allocator*. Note that the LCM allocator can be seen as a particular case of a slab memory allocator [7], commonly used in modern operating systems. However, here we take advantage of the fact that we know a priori all embedding sizes to simplify the design and minimize the fragmentation. Furthermore, JENGA uses a request-aware allocation strategy to further reduce fragmentation.

To address the second challenge, JENGA allows the application (in this case LLM serving) to customize eviction and caching policies for a wide range of attention layers such as sliding window, Mamba, and cross-attention. JENGA achieves this by providing a general mechanism to handle prefix-subset dependencies, and enable attention variants to easily customize this mechanism by precisely specifying the exact prefix subset required to generate the next token.

We have implemented and evaluated JENGA in a wide variety of models and use cases: from heterogeneous attention layers within the same model, to KV caches of draft and target models in speculative decoding [29], and to vision embeddings of VLMs. Compared to vLLM, a state-of-the-art LLM serving engine, JENGA improves memory utilization by up to 79.6%, and achieves up to 4.92× higher throughput (1.80× on average) without impacting end-to-end latency.

In summary, this paper makes the following contributions.

- We identify the growing heterogeneity of new LLM architectures, both in terms of embedding sizes and token dependencies to generate a new token.
- We propose a two-level memory allocation system for KV cache that efficiently supports different-size embeddings, and provides flexibility to customize caching and eviction policies to maximize the hit rate for different types of layers.
- A prototype implementation, JENGA, on top of vLLM, a production-level inference engine that provides support for a wide range of optimizations and features.
- Achieve a 1.80× increase in throughput over state-of-the-art LLM inference engines, without any impact on latency.

2 Background

This section introduces the basic concepts in LLM inference.

KV Cache. LLMs are autoregressive models that generate tokens iteratively, one at a time. The inference engine must store KV caches—intermediate tensors produced by the attention layers—for each token inside GPU because The computation of a new token depends on interactions between its embedding and the previously stored intermediate KV cache tensors.

The size of KV caches can be substantial. For instance, the KV cache size for Llama 3.1 8B [21] is approximately 1.2 GB for a single request with ten thousand tokens. Moreover, LLM serving systems typically batch tens of requests for efficient inference [28], requiring them to manage tens of

GBs of KV caches. Therefore, the KV cache needs to be managed carefully to achieve high inference efficiency.

Prefix Caching. The KV caches can be retained in GPU memory even after the corresponding request's generation is complete. This allows subsequent requests with shared prefixes to reuse these caches, thereby reducing redundant computation. Prefix caching is particularly effective when multiple queries share a common prefix, such as when asking different questions about the same long document.

3 Heterogeneous LLMs and Challenges

3.1 Heterogeneous LLMs

Nowadays, the state-of-the-art LLMs go beyond stacking homogeneous full-context self-attention layers. Many new types of attention layers have been introduced to the LLMs, making LLM architecture heterogeneous. In this subsection, we discuss four sources of heterogeneity, as shown in Figure 2.

(a) Sparse attention In the traditional self-attention layer, the KV cache size grows linearly with respect to the request length. Sparse attention variants aim to reduce the KV cache size by attending to only a subset of prefix tokens. The widely adopted version of sparse attention is *sliding-window attention*, where each token only attends to a fixed number of adjacent tokens inside the sliding window. To trade-off between model quality and KV cache size, recent models typically use a mix of full- and sliding-window attention layers (Figure 2a.1), including Google's Gemma-2 [48] and Mistral AI's Ministral model [4]. More advanced sparse attention variants, such as dynamically dropping some of the tokens [8, 55] (Figure 2a.2), are also proposed to control the KV cache size.

(b) State space models [23], or linear attention models [27, 37, 39], take the idea of sparse attention to the extreme: for every token, it uses a large but fixed-size tensor to capture the context information of its previous tokens. These tensors are updated recurrently during decoding. These layers are also mixed with self-attention layers (as in Jamba [31], Figure 2b). Thus, the memory allocator needs to coordinate two different patterns, i.e., a small number of large fixed-size states, one for each state space layer, and a large number of small KV cache blocks, one for each token of each self-attention layer.

(c) Multi-modal language models typically accept inputs of multiple modalities in addition to text as input and generate text output. We show an example of such Vision Language Model (VLM) in Figure 2c.1 [34]. This VLM contains a *vision encoder* that takes images as input and generates vision embeddings in the format of *image tokens*. Then, the *LLM* merges image tokens and text tokens into one sequence and performs autoregressive text generation with self-attention layers as normal LLMs. The memory allocator needs to manage the vision embedding cache, which only contains image tokens, and the KV cache of LLM parts, which contains both text tokens and image tokens. Due to KV cache compression techniques such as grouped query attention (GQA) [5], the KV cache size of a token also differs from the size of the vision embedding of an image token.

Moreover, as shown in Figure 2c.2, some VLMs, including Meta's Llama 3.2 vision model [21] and NVIDIA's NVLM model [14], utilize cross-attention to integrate the results from the image encoder into the text decoder. These LLMs have interleaving self-attention within text tokens (with KV cache for text tokens) and cross-attention between image tokens and text tokens (with encoder KV cache for image tokens). These two types of tokens can have different KV cache sizes.

(d) Serving multiple concurrent models There is also the need to serve multiple models inside a single LLM inference engine. An example is speculative decoding (Figure 2d). It ① uses a small model to quickly propose new tokens sequentially, and ② uses a large model to verify the correctness of the tokens in parallel so that it can generate multiple new tokens in each forwarding pass of the large model and keep the same quality. The KV cache size of each token differs a lot between the two models.

3.2 Heterogeneous KV Cache Size Causes Memory Fragmentation

The above heterogeneity leads to the need to allocate a KV cache of different sizes for different tokens. In this section, we analyze the fragmentation of PagedAttention when serving heterogeneous LLMs using Llama 3.2 11B Vision model as an example. This model contains 32 self-attention layers, which require KV cache for text tokens, and eight cross-attention layers, which require KV cache for image tokens.

Since the original PagedAttention can only deal with homogeneous layers, it needs to allocate KV cache for both text and image tokens for all layers (Figure 3). Suppose a request has T text tokens and I image tokens, the embedding size per layer per token is E bytes, then PagedAttention needs to store $(T+I)\times(32+8)\times E$ bytes of memory, while ideally, we only need to store text token KV cache for self-attention layers and image token KV cache for cross-attention layers, and the necessary memory for a single request should be $(T\times 32+I\times 8)\times E$.

In the MMMU-pro [58] dataset, which contains 6193 image tokens and 43 text tokens for each request on average, the resulting memory waste is 79.6%. Similarly, the memory waste of Gemma-2 and Ministral, two models that combine self-attention and sliding window attention, is up to 25% and 56.25%, respectively. Therefore, a new memory allocator is needed to reduce the memory waste of heterogeneous LLMs.

3.3 Heterogeneous Dependency Leads to Challenges in Prefix Caching

In addition to memory fragmentation, heterogeneous dependency patterns also introduces challenges for prefix caching. We summarize the challenges in prefix caching as follows:

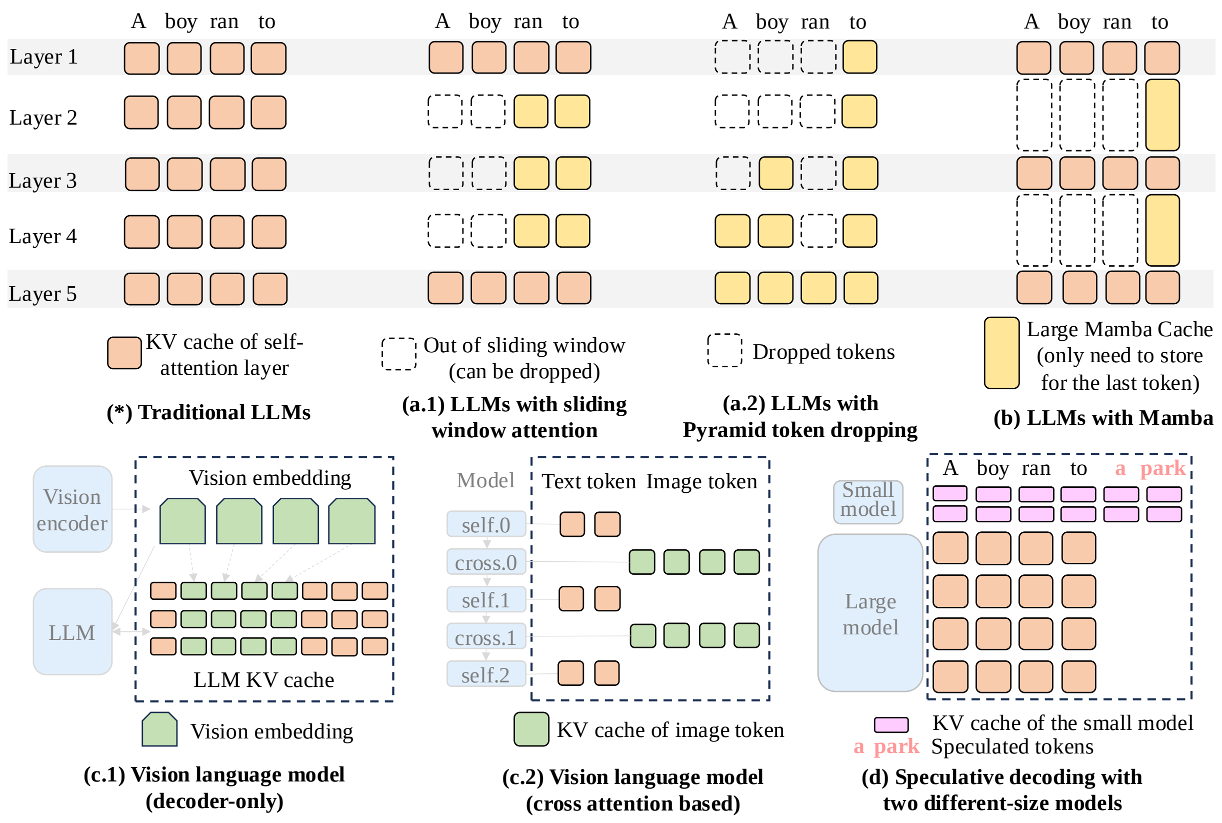

A boy ran to
Layer 1
Layer 2
Layer 3
Layer 4
Layer 5
KV cache of self-attention layer
Out of sliding window (can be dropped)
Dropped tokens
Large Mamba Cache (only need to store for the last token)
(*) Traditional LLMs
(a.1) LLMs with sliding window attention
(a.2) LLMs with Pyramid token dropping
(b) LLMs with Mamba
Vision encoder
LLM
Vision embedding
LLM KV cache
Model
Text token Image token
self.0
cross.0
self.1
cross.1
self.2
KV cache of image token
Small model
Large model
A boy ran to a park
KV cache of the small model
a park Speculated tokens
(c.1) Vision language model (decoder-only)
(c.2) Vision language model (cross attention based)
(d) Speculative decoding with two different-size models

Figure 2: Contrasting traditional LLMs (top left) and latest LLMs. LLMs are becoming more and more heterogeneous: the KV cache sizes may differ, the KV cache dependencies are different, and the LLM architecture can also diverge

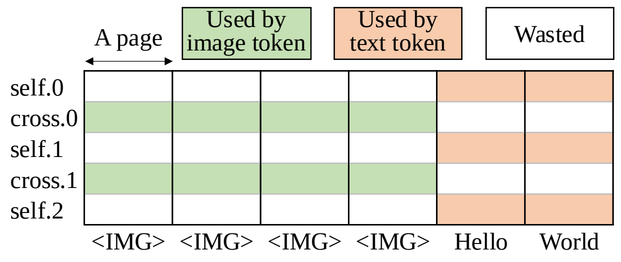


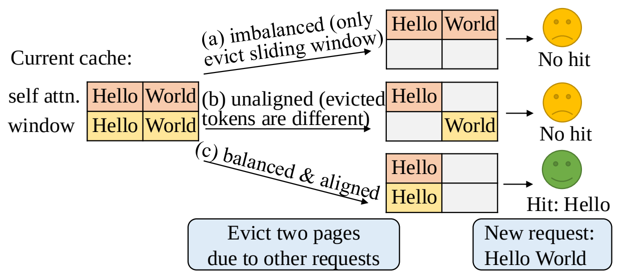
Figure 3: Visualizing the memory waste of Llama 3.2 vision model with 2 cross-attention layers (image tokens) and 3 self-attention layers (text tokens).

Different hit and eviction rules across layer types Different layer types have distinct cache hit rules due to their different token-dependency patterns. Self-attention layers compute attention between a token and all its prefixes, requiring all prefix tokens to remain unevicted to achieve a cache hit. In contrast, efficient attention mechanisms, such as those using sliding windows, only attend to a subset of prefix tokens to generate a new token. A cache hit occurs as long as this subset remains unevicted. For example, consider a prompt [~~token1~~, token2, token3, ~~token4~~], where ~~token~~ indicates an evicted token. When the sliding window size is 2, [token1, token2, token3] is still a valid prefix cache hit because token1 lies outside the sliding window, and its KV cache is not needed.

These differences in cache hit rules also necessitate customized eviction policies. For example, in sliding window layers, tokens outside the window should be prioritized for eviction over the most recent tokens.

Balanced eviction across different types It is important to balance the number of evicted tokens across different layer types. A model-wise prefix cache hit requires the prefix to exist in the prefix cache of all layer types. If one layer type evicts too many tokens, such as the sliding window layer in Figure 4a, it can prevent a cache hit, even if the prefix remains in the KV caches of other layers. However, balance does not imply evicting the same number of tokens for all layers. Different layers need different numbers of tokens to achieve similar cache hit rates, and the eviction strategy should consider the unique properties of each layer type to optimize overall performance.

Aligned eviction of different types Eviction policies across layers must be aligned to ensure that similar sets of tokens are evicted. Each token is represented by multiple pages, one for each layer type, and a cache hit of that token requires it to remain unevicted in all types. If different layers evict different sets of tokens ("world" of self-attention layer and "hello" of sliding window layer in Figure 4b), tokens inside the union of



Current cache:
self attn.
window
Hello
World
Hello
World
(a) imbalanced (only evict sliding window)
(b) unaligned (evicted tokens are different)
(c) balanced & aligned
No hit
No hit
Hit: Hello
Evict two pages due to other requests
New request: Hello World

Figure 4: Balanced and aligned cache eviction policy can improve hit rate.

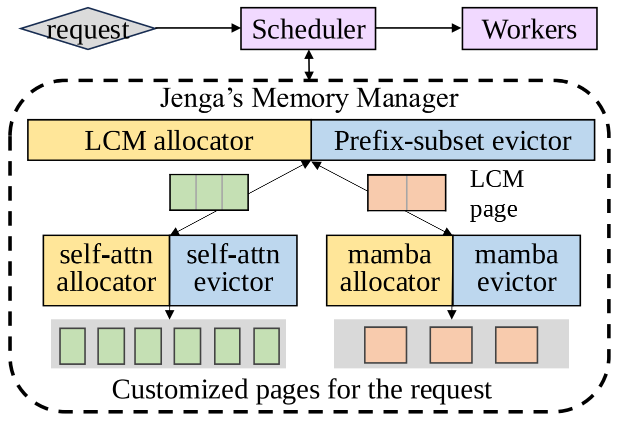


Figure 5: Overview of JENGA: a two-level memory management system for different types of layers. JENGA is composed of the LCM allocator for first-level page allocation and the prefix subset evictor for page deallocation. Within the page, a customized allocator and evictor manage the memory for the specific layer type.

these sets will become unable to hit, and overall prefix cache hit rate is reduced. To address this, cache eviction policies need to be aligned across layer types, ensuring that similar sets of tokens are evicted to maximize the cache hit rate.

4 Two-level Memory Allocation

4.1 Overview

The heterogeneity of LLMs, as discussed in §3, motivates JENGA, a two-level memory management system that allocates memory for different types of layers by introducing a compatibility layer and a customization layer. The overview of JENGA is shown in Figure 5.

For memory allocation, JENGA introduces the *LCM allocator* to allocate pages with sizes compatible across all layer types, and *customized allocators* for each specific layer type (e.g., self-attention and mamba). The customized allocators allocate pages with the specialized page size of their type from the compatible pages. For prefix cache management, JENGA introduces a *prefix-subset evictor* to coordinate the eviction among different layer types, and *customized evictors* to customize the eviction strategy of each type.

Figure 6 shows the memory layout of JENGA for Llama

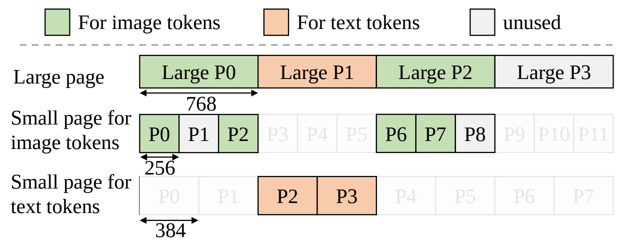


Figure 6: Two-level allocation for Llama 3.2 vision model.

vision model with page size 256 for image tokens and 384 for text tokens.[1] JENGA uses the LCM of all page sizes as the compatible page size, which is $LCM(256,384) = 768$ in this case. We will compare LCM with other potential options of the compatible page size, e.g., the GCD or MAX of all layers, in §4.4. JENGA first partitions the entire KV Cache memory into large pages of LCM size and uses the LCM allocator to manage them. The customized allocators request some large pages from the LCM allocator (large pages 0 and 2 for image tokens, large page 1 for text tokens), partition the large pages into small pages tailored to that type, and allocate the small pages as needed (the 4 256-byte small pages for 4 image tokens and 2 384-byte small pages for 2 text tokens in request `<IMG><IMG><IMG><IMG>Hello World`).

Specifically, the customized small page allocators interact with the LCM allocator as follows:

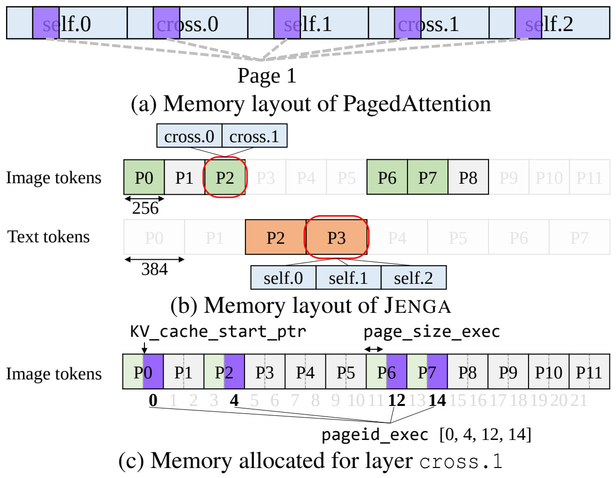
- `allocate()` for allocating a small page of that type. If all small pages for this customized allocator are allocated, it requests a new large page from the LCM allocator and partitions it into free small pages. Then, the allocator allocates an unused small page.
- `free(small_page_id)` to free a small page. The customized allocator marks the small page as unused. If all small pages within a large page are unused, the large page is returned to the LCM allocator.

The LCM-based two-level allocation strategy prevents external fragmentation among large pages, and §4.3 can reduce internal fragmentation inside each large page. Additionally, for each layer type, allocated pages can be fully represented by small page IDs of that type (e.g. small pages P2 and P3 for text tokens) so the attention kernels can be executed as if there is only one layer type and do not need to consider the complexity introduced by multiple types.

4.2 Execution with New Memory Layout

Although JENGA employs a memory layout distinct from the standard PagedAttention, JENGA can reuse the PagedAttention workers with very little change. This section explains how JENGA works with existing PagedAttention workers.

[1]We assume the KV cache size per token of each layer is 128, and the model contains 2 cross attention layers (with KV size per image token $128 \times 2 = 256$) plus 3 self-attention layers (with KV size per text token $128 \times 3 = 384$). For simplicity of explanation, we set $tokens\_per\_page = 1$ in this paper, but JENGA is capable of arbitrary *tokens_per_page*.



(a) Memory layout of PagedAttention

(b) Memory layout of JENGA

(c) Memory allocated for layer `cross.1`

Figure 7: Memory layout of PagedAttention and JENGA

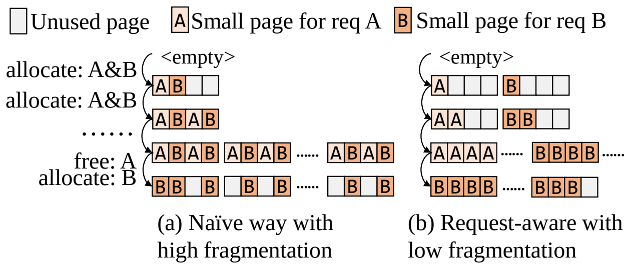
New memory layout for inter-type memory exchange As shown in Figure 7a, in standard PagedAttention, each logical page is divided into multiple slices in the physical memory, making it difficult to exchange pages between different types of KV cache. Specifically, the memory layout of the standard PagedAttention follows a *layer-page partition* that first partitions the memory into layers and then partitions each layer into pages. This layout simplifies model execution because when executing one layer, we only need to pass the memory of that layer. This layout is widely used by inference engines including vLLM [28], SGLang [59], TGI [26], and attention libraries, e.g., FlashAttention [16], FlashInfer [56], but does not meet the need of JENGA.

To enable memory sharing between memory types, JENGA proposes a *page-layer partition* to make each small page consecutive. As shown in Figure 7b, JENGA partitions the memory into pages and then partitions each page into layers.

Despite this new memory layout, JENGA is still compatible with PagedAttention workers and kernels. As shown in Figure 7c, memory allocated for a layer (e.g., `cross.1`) can still be represented by a customized `start_ptr`, `page_size`, and `page_id`, maintaining consistency with the PagedAttention kernel interface.

JENGA using PagedAttention workers with minimal change JENGA requires no modifications to PagedAttention kernels. Attention can be computed as usual by passing the above `kv_cache_start_ptr`, `page_size_exec`, `pageid_exec` to the libraries.

The changes to the inference engine workers are also minimal. Major changes only include: (1) allocate a single KV cache tensor and assign different offsets (`KV_cache_start_ptr`) for each layer, instead of allocating fixed-size tensors for each layer (2) prepare attention metadata (parameters passed to attention kernels, e.g., page ID), for each layer type, rather than a global metadata for all layers.



(a) Naïve way with high fragmentation

(b) Request-aware with low fragmentation

Figure 8: Internal fragmentation with two requests A&B, and 4 small pages per large page

4.3 Request Aware Allocation to Reduce Internal Fragmentation of Large Pages

A naive allocation strategy can lead to large internal fragmentation due to the different allocation and free patterns in LLM workloads. As shown in Figure 8a, memory allocation for a single request is often interleaved with allocations for other requests, whereas memory deallocation for a single request typically occurs together. When a request is completed, numerous small pages will be freed. However, only very few large pages can be returned to the large-page allocator because the small pages of that request are sharing the large pages with other requests due to the interleaved allocation. This results in severe internal fragmentation.

JENGA addresses the memory fragmentation problem by aligning the allocation and deallocation patterns for each request. Specifically, as shown in Figure 8b, JENGA allocates small pages within a single large page to the same request whenever feasible. This ensures that large pages can be returned to the large-page allocator once the request is completed. For attention variants where the freeing of different pages within the same request does not occur simultaneously, this approach remains effective, as adjacent small pages are typically freed shortly after one another.

The complete allocation algorithm considering the interaction between the allocator for the two levels, as well as the request aware allocation, is shown below. Each large page and all its small pages are associated with a specific request. JENGA prioritizes allocating small pages to their associated requests by the following algorithm:

1. Allocate an unused small page associated with that request.
2. If fail, request a new large page from the large-page allocator, mark all small pages inside the large page as associated with this request, and allocate one of these small pages.
3. If steps 1-2 fail, allocate an unused small page in this small-page allocator but associated with other requests.

4.4 Discussion: Different Choices of Compatibility Layer

The aligned embedding size can motivate many different design decisions of the compatibility layer, each with its own advantages and limitations. We think that LCM is the most flexible and extensible choice among them.

```
class LayerSupportsPrefixCache:
  def update_last_access(r: Request, time: int);
  def set_prefix_length(r: Request);
  def get_possible_prefix(is_hit: List[bool]);
```

(a) Interface for customized prefix caching of different layer type

```
class SlidingWindowLayer(LayerSupportsPrefixCache):
  def update_last_access(r: Request, time: int):
    for i in range(r.len-sliding+1, r.len+1):
      self.evictor.update_last_access(r.page[i], time)

  def set_prefix_length(r: Request):
    for i in range(0, r.len):
      self.evictor.set_prefix_length(r.page[i], i)

  def get_possible_prefix(is_hit: List[bool]):
    return {p | ∀x ∈ [0,sliding), is_hit[p-x] is True}
```

(b) Prefix caching support of sliding window layers

Figure 9: Layer property aware prefix caching

GCD page Using the greatest common denominator (GCD) of different embedding sizes as the compatible page size. This approach will have no internal fragmentation. However, it significantly reduces LLM inference speed. This is because the most efficient GPU kernels typically require the KV cache to be contiguous along specific tensor dimensions. The GCD solution may have to partition the tensor along these dimensions, requiring the customization of GPU kernels for a wide range of GCDs. This greatly increases GPU kernel engineering overhead, and even with such customization, performance often falls short of that achieved by the most efficient kernels. For example, MuxServe [20] uses a GCD page to serve multiple models but restricts itself to models with the same size per head to avoid excessive GPU kernel development.

MAX page Take the maximum of different embedding sizes as the compatible page size. This solution will have internal fragmentation for layers with smaller page size. A potential workaround is to increase the number of tokens per page for these small layers. However, this results in coarser granularity for both memory allocation and cache hits. For example, Jamba 52B's large mamba state requires assigning 1344 tokens to each self-attention page to avoid internal fragmentation, which exceeds the typical number of tokens in real-world requests, such as 1085.04 on average in ShareGPT [49].

LCM page Take the least common multiple (LCM) of different embedding sizes as the compatible page size, which is used by JENGA. The LCM page does not need new GPU kernels or an extremely large number of tokens assigned to each page. However, it may lead to internal fragmentation within each large page due to unused small pages. JENGA addresses this problem by request-aware allocation (§4.3). Another potential problem is that the LCM might be extremely large. In practice, for all models supported by vLLM v0.6.4, the largest LCM comes from Jamba, where it is 84× the small page size used in the model, and we do not observe any performance degradation in that model.

5 Customizable Prefix Caching

The prefix caching system of an inference engine needs to support two tasks:

1. **Cache eviction:** Evict an existing page from the cache to free up space for a new page.
2. **Cache hit:** Identifying the cached prefix for a request

For both tasks, the expected behavior varies across different layer types, and thus needed to be customized inside the inference engine. JENGA provides unified interfaces to customize each layer, and a global prefix-subset evictor that manages the diverse layer types by invoking these APIs. The interface is shown in Figure 9a, with `update_last_access` and `set_prefix_length` for customized eviction rule, and `get_possible_prefix` for customized hit rule. Further details on cache eviction and cache hit mechanisms are provided in §5.1 and §5.2.

5.1 Customizable Cache Eviction

As discussed in §3.3, JENGA's cache eviction policy must ensure both balance and alignment across different layer types. This section illustrates JENGA's approach using least recently used (LRU) eviction as an example.

Balanced eviction by `update_last_access` JENGA provides a coarse grain API, `update_last_access`, to set unified last-access times across different layers, thus enabling balanced eviction. In LRU eviction, the page with the earliest last-access time is evicted. By aligning last-access times for tokens in the same request but different layers, JENGA ensures that eviction priorities across layers remain similar.

Figure 10 shows the timeline and last access time of a model with one self-attention layer and one sliding window layer when running the following two requests:

- Request 1: input `[A B C D]` and output `[E F]`
- Request 2: input `[A B C D G]` and output `[H]`

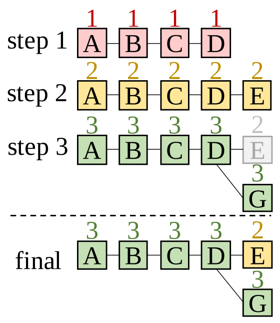
For simplicity of explanation, we assume all layer types have the same page size, such that each "large page" contains one "small page". In this scenario, the terms "large page" and "small page" can be used interchangeably and are collectively referred to as "page." The general case, where multiple page sizes coexist, is addressed in §5.4.

For self-attention layer, the last access times of all running tokens are updated in each step. For example, in Figure 10b, tokens [A B C D] are updated in step 1, and then tokens [A B C D E] in step 2 and [A B C D G] in step 3.

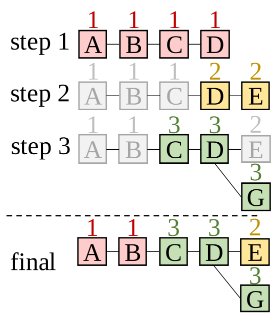
For the sliding-window layer, JENGA updates the last access time only for tokens within the sliding window, as these are the tokens actively involved in the attention computation. For example, in step 2, tokens [A B C] are not updated because generating token [F] only needs the KV cache of tokens [D E]. This customization ensures that tokens outside the sliding window retain an earlier last access time, making them a higher priority for eviction.

Step	Request	self attention	sliding window
1	1's prefill	ABCD->E	ABCD->E
2	1's decode	ABCDE->F	DE->F
3	2's prefill	ABCDG->H	CDG->H

(a) Timeline of the 2 requests. ABC->D means access KV cache of tokens ABC and generate token D. Note that the generated token does not have KV cache. Assume sliding window size 2.



(b) Self-attention layer



(c) Sliding window layer

Figure 10: Last access time after two requests

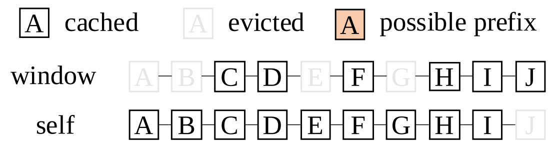
Despite these customizations, eviction remains balanced since tokens from the same request share identical last-access timestamps across layers. For example, JENGA will evict tokens exclusive to Request 1 (e.g., [E] with time_stamp 2) in both layers before those from Request 2 (e.g., [C D G] with time_stamp 3) based on the last access time.

Aligned eviction by `set_prefix_length` JENGA provides a fine-grained API, `set_prefix_length`, to refine eviction priorities for pages with the same last-access time to ensure aligned eviction. By assigning identical prefix length values to the page of the corresponding token across layers, JENGA ensures consistent eviction priorities of these pages. For example, tokens [A B C D G] in both layers can be assigned lengths [1, 2, 3, 4, 5], respectively. Thus, the token with the highest length (e.g., [G] in the two layers) is evicted before other tokens (e.g., [C D]), maintaining alignment across layers.

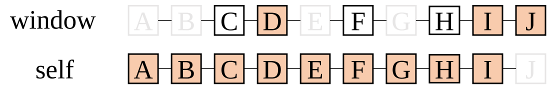
5.2 Customizable Cache Hit

Cache hit rules differ across layer types. For example, a prefix hit in a self-attention layer requires all tokens in the prefix to remain unevicted, whereas a sliding window layer only requires the last sliding_window_size tokens of the prefix to remain unevicted.

JENGA provides the `get_possible_prefix` API to custom the cache hit rules for different layer types. Given the parameter `is_hit` indicating whether the KV value of each token is cached, the function should return all valid prefixes of that layer. For example, for request [ABCDEFGHIJ], with the KV cache status in Figure 11a, valid prefixes for a self-attention layer are [A], [AB], ..., [ABCDEFGHI], while valid prefixes for a sliding window layer are [ABCD], [ABCDEFGHI], and [ABCDEFGHIJ]. The prefix [ABC] is invalid for a sliding window layer because both B and C must remain cached to achieve such hit.



(a) The KV cache



(b) The possible prefixes for request `ABCDEFGHIJ`

Figure 11: Customizable cache hit

Upon receiving a new request, the compatibility layer invokes `get_possible_prefix` for each layer type to identify valid prefixes. The longest common prefix valid across all layers is selected as the cache hit prefix.

5.3 The Customization of Different Layers

Sliding Window Layer JENGA updates the last-access time only for tokens within the sliding window and require only these tokens to remain cached for a prefix hit. Its implementation is shown in Figure 9b.

Mamba Layer For Mamba layers, caching is specialized due to the significantly larger per-token state compared to other attention-based layers. Instead of caching all tokens, which requires too much memory, JENGA only caches the state of every 512 tokens. Then, JENGA can hit the prefixes with length a multiplier of 512, which is implemented by the `get_possible_prefix` interface. Additionally, only the last cached token's access time is updated via `update_last_access`. We also notice Marconi [38], a concurrent work that provides an advanced algorithm to select the set of tokens to cache. Its algorithm can be integrated into JENGA to enhance the prefix caching.

Vision Embedding Cache and Vision Cross Attention Cache In these caches, evicting even a single image token will require the recomputation of the entire vision encoder. To minimize the number of recomputed images, it is better to evict all tokens from one image than to evict an equivalent number of tokens across multiple images. To achieve this, JENGA assigns a randomized `prefix_length` to each image and applies this value to all corresponding tokens via `set_prefix_length`. Tokens from the image with the highest random value are prioritized for eviction.

5.4 Cache Eviction of LCM Page Table

JENGA manages memory with a two-level approach: *large pages* that are compatible across all layer types and *small pages* that are customizable for specific layer types. This design enables JENGA to achieve two benefits simultaneously: 1. Fine-grained, customizable cache hits for small pages.

2. Efficient memory sharing via coarse-grained large-pages. However, effective prefix caching requires careful coordination between fine-grained small-page cache hits and coarse-grained large-page exchanges between different layer types.

Specifically, each small page can be in one of three states: (1) *empty* page with no valid KV cache and is not used by any requests; (2) *evictable* page with valid KV cache and is not used by any requests (3) *used* page that is used by running requests and thus unevictable. Moreover, we call a large page *empty* if all its small pages are empty, and *evictable* if all its small pages are evictable.

The allocation algorithm in JENGA should (1) prioritize unused pages and perform eviction only when necessary (2) try its best to keep small pages inside a large page in the same state to avoid internal fragmentation like Figure 8a.

To allocate a new small page of a specific type for a particular request, JENGA implements the following steps:

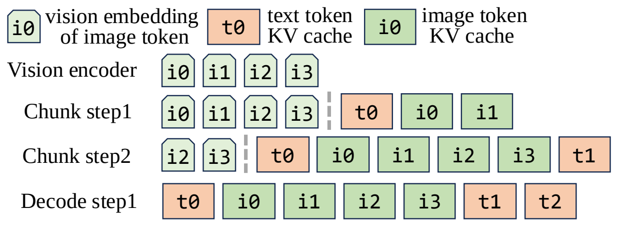
1. **Allocate a request-associated unused small page.** Each small page is associated with a specific request (§4.3). JENGA first attempts to allocate an empty small page of the required type that is associated with the current request.
2. **Allocate from an empty large page.** If no suitable small page is available, JENGA requests an empty large page from the large-page allocator, associates its small pages with the request, and allocates one of these small pages.
3. **Allocate by evicting a large page.** If no unused large page exists, JENGA evicts an evictable large page using the LRU eviction policy. The LRU timestamp of a large page is set to the latest last access time among all its small pages. After eviction, all small pages within the large page are marked empty, and one of them is allocated.
4. **Allocate an arbitrary unused small page.** If steps 1–3 fail, JENGA allocates an unused small page of the required type that may not be associated with the specific request.
5. **Allocate by evicting a small page.** As a last resort, JENGA evicts an evictable small page of the required type using the LRU eviction policy and allocates it.

6 Case Studies

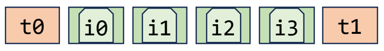
6.1 Speculative Decoding and Multiple Model

Speculative decoding involves an extra small model (the speculator) in the LLM inference engine. JENGA can be naturally extended to support such case as it can allocate two different KV cache sizes for the two models automatically with negligible fragmentation.

Moreover, JENGA can be extended to serve multiple models inside the same LLM inference engine. After registering all models with the `custom_kv_cache` API, JENGA can have a compatible page size for all models, which can be the granularity to exchange pages between inference engines. We leave the full support of serving multiple models as a future work.



(a) Vision embedding cache for allocate-on-demand KV cache



(b) Vision embedding cache for fully-allocated KV cache

Figure 12: Vision embedding and chunked prefill KV cache

6.2 Vision Embedding Cache of VLMs

As the vision embedding cache can be treated as another type of layer with a specific hidden size, JENGA can automatically handle the different page sizes of vision embedding tokens and LLM KV cache tokens.

Moreover, with JENGA's customizable memory management, ***vision embedding cache does not increase the peak memory consumption of the model*** when the vision embedding cache per token is smaller than the LLM KV cache size per token, which is held by all VLMs in vLLM v0.6.4. After being generated by the vision encoder, the vision embedding cache will be consumed by chunked prefill steps of the LLM part. Current inference engines support chunked prefill of LLM part in two ways, in both of which JENGA can effectively manage the vision embedding:

Allocate-on-demand KV cache with free-on-demand vision embedding cache Some inference engines only allocate the KV cache for tokens used in the current chunked prefill step. JENGA can free the vision embedding of image tokens once consumed by the prefill step. Therefore, the peak memory usage is affected by the shrink of vision embedding and enlargement of the KV cache simultaneously and will not grow too large. Figure 12a shows the timeline of request `[t0 i0 i1 i2 i3 t1]`, where `t*` and `i*` refers to text token and image token respectively. JENGA process this request with the following steps: (1) runs the vision encoder and generates the vision embedding; (2) runs the chunked prefill of 3 tokens `[t0 i0 i1]` and generates their KV cache; (3) frees the consumed vision embedding of image tokens `[i0 i1]` and runs the chunked prefill of 3 tokens `[i2 i3 t1]`; (4) frees the consumed vision embedding of image tokens `[i2 i3]` and runs a decode step. The peak memory usage is the KV cache of all tokens, plus at most *chunk_prefill_size* tokens of vision embedding. The *chunk_prefill_size* is much smaller than the number of tokens in all scheduled requests, and the vision embedding size per token is small compared to the KV cache size per token, so JENGA can almost remove the memory consumption of vision embedding cache.

Fully allocated KV cache and cache reusing by vision embedding Other inference engines allocate the full KV cache of all prefill tokens at the first chunked prefill step. As the

Model	Dataset	H100	L4
Llama 3.2 Vision (mllama)	MMMU-pro	11B	11B*
Gemma-2	arXiv-QA	27B	9B
Ministral	arXiv-QA	8B	8B*
Jamba-1.5	MMLU-pro	52B*	OOM
character.ai	MMLU-pro	70B*	8B
PyramidKV	MMLU-pro	70B*	8B
Llama 3.1	MMLU-pro	70B*	8B

Table 1: Model and dataset. * means with FP8 quantization.

vision embedding cache of one token is used before generating the KV cache of that token, JENGA supports reusing the KV cache for vision embedding cache, as shown in Figure 12b, which completely removes the memory consumption of vision embedding cache.

7 Evaluation

JENGA is implemented with about 4000 lines of Python code on top of vLLM and does not require any change of CUDA kernels. JENGA is compatible with all the 90 models in vLLM v0.6.4. JENGA is transparent to users of the inference engine. JENGA can parse all possible embedding sizes from the model structure and perform the memory allocation automatically. In this section, we evaluate the performance of JENGA.

7.1 Evaluation Setup

Platform We evaluate JENGA on two GPU platforms: (1) NVIDIA H100 80GB GPU with 2 Intel Xeon Platinum 8480C CPUs, equipped with CUDA 12.5. This is the default platform in the evaluation unless explicitly mentioned (2) NVIDIA L4 24GB GPU with 2 AMD EPYC 7F52 16-Core Processor, equipped with CUDA 12.4.

Baselines We perform end-to-end evaluation of JENGA by using vLLM v0.6.3 and only change the memory management system. We also offers a break down experiment to compare JENGA with the memory management system of other state-of-the-art LLM inference engines, including vLLM [28], SGLang [59], and TGI [26]. We don't perform end-to-end evaluation on these engines as they only support a very small subset of the evaluated models.

Models We include a wide range of heterogenous LLMs. Llama vision model [21] is a multi-modal model with cross attention layers. Gemma-2 [48] and Ministral [4] are two models with sliding window layers. Jamba [31] contains Mamba [23] layers. Character.ai [10] is a private model with sliding window layers and KV cache sharing. We implement the model based on their blog post on top of Llama. PyramidKV [55] is a sparse attention model that drops different number of tokens in different layers. All these models mix the aforementioned attention variants with standard self-attention layers. We also use standard Llama with self attention layers only for evaluating the overhead of JENGA. The size of the model is listed in Table 1. We also use LLaVA-OneVision [30] 7B, InternVL2 [11] 8B, Phi-3 Vision [1] 4B, and Paligemma2 [47] 10B to evaluate the vision embedding cache. Note that Paligemma2 is a model mixed with three types of memory, e.g., vision embedding cache, sliding window KV cache and self-attention KV cache. We do not evaluate Hamba [18] which is discussed in §1 because it lacks essential GPU kernels and is not supported by vLLM yet.

Dataset We use MMLU-pro [21] for text-only models and MMMU-pro [58] for multi-modality models. MMLU-pro's maximum length is only 3076, which is shorter than the sliding window size of Gemma-2 and Ministral. These two models will be degenerated into self-attention-only models with MMLU-pro dataset. Therefore, we use arXiv-QA, a long-context dataset that do question answering on a collection of arXiv articles [25] for the two models.

7.2 End-to-end Evaluation

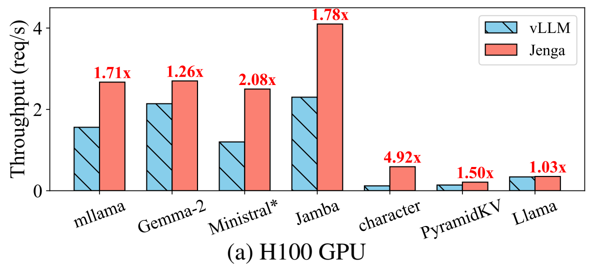
End-to-end throughput Figure 13 compares the end-to-end inference throughput of vLLM and JENGA on both H100 and L4 GPUs. JENGA achieves up to 4.92× speedup (1.80× on average) on H100 and 3.29× speedup (1.69× on average) on L4. The speedup comes from both less memory waste and better prefix caching. We will provide more breakdowns in §7.3. JENGA also achieves comparable throughput with vLLM on the standard Llama, proving that it can also be used to serve self-attention-only models without introducing new overhead.

The smallest Jamba model is 52B, which cannot be filled in one L4 24GB GPU, so this model is skipped on the L4 platform. vLLM fails to serve the longest request in the dataset for the Ministral model on the L4 platform, while JENGA can serve it due to the reduced memory usage. The throughput of Ministral is much smaller than other models because the requests have an average length of 92408, much longer than the thousands-of-token requests in other models.

End-to-end latency Figure 14 shows the latency under different request rates of the Llama Vision model. When the request rate is low (<1.2 requests/s), the latency of vLLM and JENGA is similar, with only a 2.6% difference on average, proving that JENGA does not sacrifice model latency. When the request rate grows, JENGA reduces the end-to-end latency by up to 2.24× and time to first token by up to 29.43× due to less memory waste and larger batch size. The time per output token (TPOT) of JENGA is larger than vLLM because JENGA batches more requests and has more computation in each step. JENGA can achieve the same TPOT if scheduling the same number of requests in each step.

7.3 Break down

Decode batch size JENGA improves the LLM serving engine's throughput and end-to-end latency by maximally enlarging the batch size. To measure the batch size of JENGA compared to other inference engines, we pick three open-source inference engines (vLLM [28], SGLang [59] and



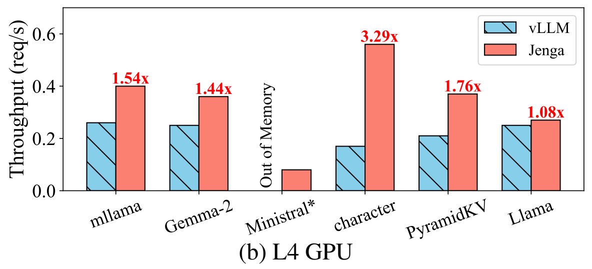


Figure 13: End-to-end throughput. Amplified Ministral's throughput in both vLLM and JENGA by 10× for better visualization.

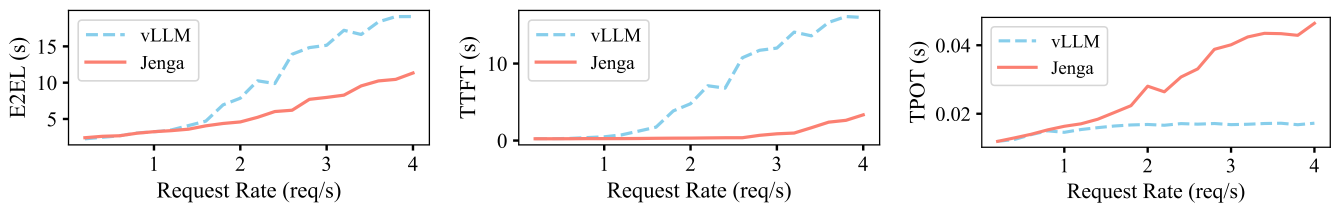


Figure 14: Averaged Latency for the Llama Vision Model (mllama) with changing request rates. E2EL denotes end-to-end latency, TTFT denotes time to first token, and TPOT denotes time per output token.

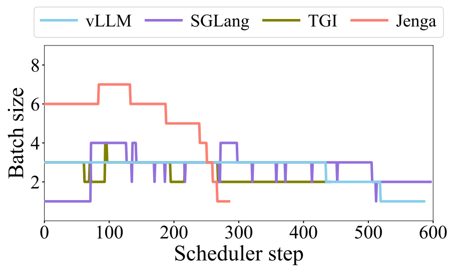


Figure 15: Timeline of decode batch size for Ministral model of JENGA and existing LLM inference engines.

TGI [26]) that are widely used in production. As for the workload, we use a simulated workload, where there are 20 requests coming to the inference engine all at once, with input length randomly drawn from 55-110 thousand tokens and with output length from 50-100 tokens. This simulates the typical long document question-answering workload, where the input length can be excessively long, but the output length is relatively short. We then visualize the batch size of LLM decoding steps in Figure 15. The average batch size of JENGA is 5.39, 1.95× larger than the average batch size of other inference engines (2.63, 2.74, and 2.50 for vLLM, SGLang, and TGI, respectively). As a result, JENGA finishes LLM inference within 300 steps, while other inference engines need around 600 steps. Note that TGI finishes earlier as it does not support the `--ignore-eos` flag [51], and thus will generate fewer tokens compared to the other inference engines.

Fragmentation analysis Figure 16 shows the memory used for each part during the inference of Ministral model. We use a *static* trace that the request length distribution does not change over time and a *dynamic* trace that the average length forms a uniform distribution over time. We divide the use of GPU memory into five types, (1) the model *weight*, (2) the memory *reserved* for the inference engine for things like model activations and cuda graphs, (3) the memory used for storing KV caches required by new token generation (*used* in Figure 16ab, *used-self* and *used-window* in Figure 16cd), (4) the *wasted* memory that is allocated but not stores useful KV cache, and (5) the *unallocated* KV cache memory. For the two traces, vLLM wastes 38.2% KV cache memory on average as it fails to free the KV cache of sliding window layers for tokens outside the window, while JENGA only has 0.04% KV cache memory waste, which comes from the unused small pages inside the large pages and the last page that is only partially filled. Moreover, in the dynamic trace, JENGA can dynamically allocate the KV cache memory to sliding window layers and self-attention layers based on the workload, with the rate of self-attention KV cache ranging from 27.8% to 54.5% among all allocated KV cache memory.

Prefix caching Figure 17 evaluates the prefix caching system of JENGA by using a different number of articles in the arXiv dataset [25] and asking multiple questions at the end of each article. When the number of articles is small (e.g., <= 3, both the two systems can cache all the articles and provide similar throughput. The reason for the slight overhead of JENGA is that JENGA needs to allocate memory twice, one for self-attention layers and the other for sliding window layers, while vLLM only allocates once for all layers. When the number of articles is big, JENGA has up to 1.60× higher cache hit rate as the customized sliding window eviction rule prioritizes the eviction of KV cache for tokens outside the sliding window, while vLLM treats all layers as self-attention layers. The higher cache hit rate saves more computation and thus provides up to 1.77× throughput improvement.

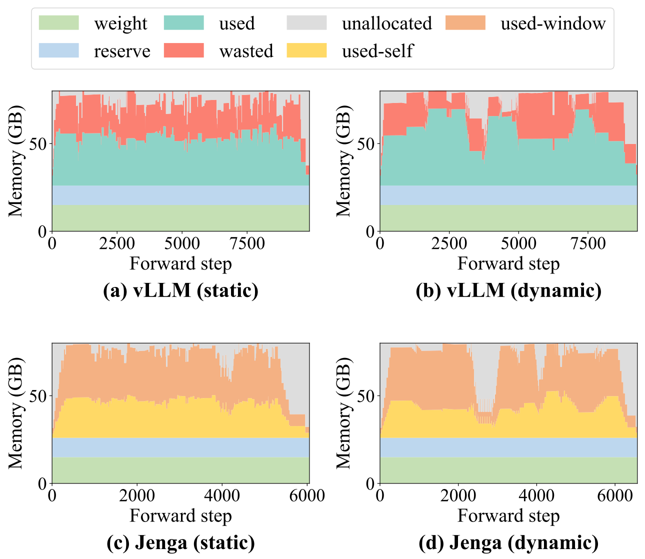


Figure 16: Timeline of memory usage for Ministral model. vLLM shows significant wasted memory due to memory fragmentation (in red), while JENGA minimizes waste.

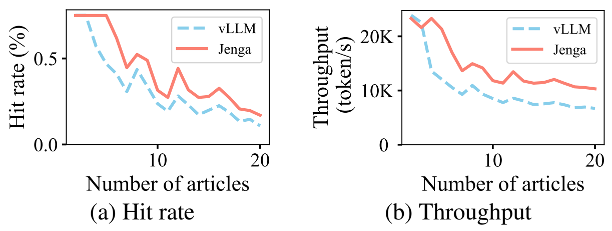


Figure 17: Prefix caching with different number of articles.

7.4 Case Study

Vision embedding cache for VLMs Figure 18 shows the performance improvement of vision language models due to JENGA's support of vision embedding cache. Without such a cache, inference engines such as vLLM and SGLang need to re-run the vision encoder part in each chunked prefill step. With the vision embedding cache, JENGA only needs to run the vision encoder once for each request, leading to 1.88× and 1.60× improvement in throughput and latency over vLLM, respectively. The models are evaluated on the MMMU-pro dataset with chunked prefill batch size 1024.

Speculative decoding Figure 19 compares the performance of speculative decoding in vLLM and JENGA, which includes a small model and a large model running simultaneously. The large models use the model size in Table 1, and the small model sizes are 2B for Gemma2 and 1B for other models, where the 1B model of ministral is an example model created by us following the model configuration of Llama 3.2 1B.

vLLM-max refers to the scenario of using a uniform page size as in the PagedAttention [51], where the page size needs to be set as the page size of the large model. *vLLM-manual* uses a manually-designed memory allocation strategy for speculative decoding by SmartSpec [35]. This strategy has

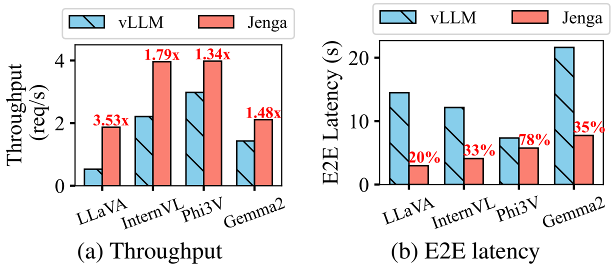


Figure 18: Vision language model with chunked prefill.

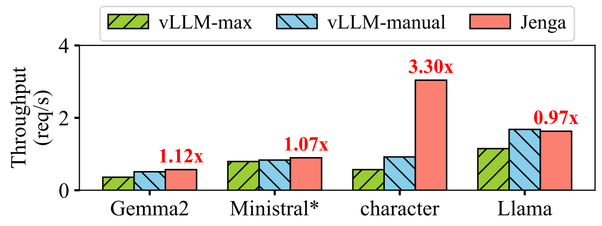


Figure 19: Speculative decoding. Amplified the throughput of Ministral by 10 × for better visualization.

no memory fragmentation when the model only contains self-attention layers but does not work perfectly on heterogeneous LLMs. JENGA can achieve the same performance as *vLLM-manual* in standard Llama, showing the automatic memory management in JENGA can reach the optimal case for self-attention-only models. Moreover, JENGA can achieve an averaged 1.58× throughput improvement on heterogeneous LLMs without redesigning the memory allocation strategy.

8 Related Work

LLM serving systems Many works have optimized LLM serving for different scenarios. Orca [57] enables token-level request batching to enhance inference throughput. LoongServe [52] and DeepSpeed-FastGen [24] improve the efficiency of long-sequence inference through advanced scheduling algorithms. DistServe [60] and SARATHI [3] mitigate output token latency variance by controlling the number of tokens processed per step. Parrot [32], SGLang [59], and XGrammar [19] enable users to specify the output structure. MuxServe [20] allows multiple models to be served on a single GPU. These approaches have successfully optimized serving for homogeneous models with standard self-attention layers, and can be further extended to support heterogeneous models with the help of JENGA.

LLM Memory allocation Various methods have been proposed to reduce memory fragmentation during LLM inference. PagedAttention [28] eliminates fragmentation caused by variable-length sequences by dividing the KV cache into fixed-size pages, but fails to handle heterogeneous models with different embedding sizes. vAttention [40] utilizes GPU virtual memory to allocate contiguous KV cache memory for each request. However, it suffers from coarse-grained memory allocation and significant allocation and deallocation overhead of GPU driver. Moreover, virtual-memory-based mechanisms cannot track the prefix-subset dependency to perform effective prefix caching. Several studies propose memory allocation algorithms for specific model type, e.g., SLoRA [44] for LORA, Marconi [38] for Mamba, and SmartSpec [35] for speculative decoding. In contrast, JENGA provides a general solution that is compatible to a wider range of LLM memory types.

LLM Memory optimization There are also works to reduce the peak GPU memory usage of LLM inference. FlashAttention [15, 16, 42] reduces the memory footprint of attention kernels by tiling. CachedAttention [22] and Mooncake [41] enable KV cache offloading to larger memory pools, such as CPU memory or disk storage. JENGA can provide fixed-size offloading granularity and suggest the offload order of pages when extending these systems to heterogeneous LLM senarios. Solutions such as FlexGen [45], MoE-Lightning [9], Moe-Infinity [54], and PowerInfer [46] further support weight and activation offloading. Additionally, various attention mechanisms have been developed to reduce KV cache size, including MQA [43], GQA [5], and MLA [33], which decrease KV memory requirements per token, and several KV cache pruning techniques [2, 8, 53, 55] to reduce the number of tokens inside the KV cache. These novel KV cache designs can be easily integrated into inference engines with the help of JENGA.

9 Conclusion

This paper introduces JENGA, an efficient memory allocation framework for managing heterogeneous embeddings in modern LLM architectures. By utilizing a two-level memory allocator, JENGA reduces memory fragmentation and enables customizable caching policies for different types of embeddings. In our experiments, JENGA achieved up to 79.6% higher GPU memory utilization and $1.26 - 4.91\times$ increased serving throughput across a variety of models and scenarios.

References

[1] Marah Abdin, Jyoti Aneja, Hany Awadalla, Ahmed Awadallah, Ammar Ahmad Awan, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Jianmin Bao, Harkirat Behl, et al. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv preprint arXiv:2404.14219*, 2024.

[2] Muhammad Adnan, Akhil Arunkumar, Gaurav Jain, Prashant Nair, Ilya Soloveychik, and Purushotham Kamath. Keyformer: Kv cache reduction through key tokens selection for efficient generative inference. *Proceedings of Machine Learning and Systems*, 6:114–127, 2024.

[3] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming {Throughput-Latency} tradeoff in {LLM} inference with {Sarathi-Serve}. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 117–134, 2024.

[4] Mistral AI. Introducing the world's best edge models, 2024. Accessed: 2024-12-08.

[5] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245*, 2023.

[6] Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

[7] Jeff Bonwick et al. The slab allocator: An object-caching kernel memory allocator. In *USENIX summer*, volume 16. Boston, MA, USA, 1994.

[8] Zefan Cai, Yichi Zhang, Bofei Gao, Yuliang Liu, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Baobao Chang, Junjie Hu, et al. Pyramidkv: Dynamic kv cache compression based on pyramidal information funneling. *arXiv preprint arXiv:2406.02069*, 2024.

[9] Shiyi Cao, Shu Liu, Tyler Griggs, Peter Schafhalter, Xiaoxuan Liu, Ying Sheng, Joseph E Gonzalez, Matei Zaharia, and Ion Stoica. Moe-lightning: High-throughput moe inference on memory-constrained gpus. *arXiv preprint arXiv:2411.11217*, 2024.

[10] Character.ai. Optimizing ai inference at character.ai, 2024. Accessed: 2024-12-10.

[11] Zhe Chen, Jiannan Wu, Wenhai Wang, Weijie Su, Guo Chen, Sen Xing, Muyan Zhong, Qinglong Zhang, Xizhou Zhu, Lewei Lu, et al. Internvl: Scaling up vision foundation models and aligning for generic visual-linguistic tasks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 24185–24198, 2024.

[12] CNBC. Microsoft warns of service disruptions if it can't get enough a.i. chips for its data centers.

[13] CNBC. Openai's active user count soars to 300 million people per week. *CNBC*, December 2024.

[14] Wenliang Dai, Nayeon Lee, Boxin Wang, Zhuolin Yang, Zihan Liu, Jon Barker, Tuomas Rintamaki, Mohammad Shoeybi, Bryan Catanzaro, and Wei Ping. Nvlm: Open frontier-class multimodal llms. *arXiv preprint arXiv:2409.11402*, 2024.

[15] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

[16] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

[17] CIO Dive. Github copilot subscriber count surges alongside revenue growth, 2024. Accessed: 2024-12-09.

[18] Xin Dong, Yonggan Fu, Shizhe Diao, Wonmin Byeon, Zijia Chen, Ameya Sunil Mahabaleshwarkar, Shih-Yang Liu, Matthijs Van Keirsbilck, Min-Hung Chen, Yoshi Suhara, et al. Hymba: A hybrid-head architecture for small language models. *arXiv preprint arXiv:2411.13676*, 2024.

[19] Yixin Dong, Charlie F Ruan, Yaxing Cai, Ruihang Lai, Ziyi Xu, Yilong Zhao, and Tianqi Chen. Xgrammar: Flexible and efficient structured generation engine for large language models. *arXiv preprint arXiv:2411.15100*, 2024.

[20] Jiangfei Duan, Runyu Lu, Haojie Duanmu, Xiuhong Li, Xingcheng Zhang, Dahua Lin, Ion Stoica, and Hao Zhang. Muxserve: Flexible multiplexing for efficient multiple llm serving. *arXiv preprint arXiv:2404.02015*, 2024.

[21] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[22] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. {Cost-Efficient} large language model serving for multi-turn conversations with {CachedAttention}. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 111–126, 2024.

[23] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.

[24] Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, et al. Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference. *arXiv preprint arXiv:2401.08671*, 2024.

[25] Huggingface. arxiv-march-2023, 2024. Accessed: 2024-12-10.

[26] huggingface. Text generation inference, 2024. Accessed: 2024-12-10.

[27] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In *International conference on machine learning*, pages 5156–5165. PMLR, 2020.

[28] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

[29] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pages 19274–19286. PMLR, 2023.

[30] Bo Li, Yuanhan Zhang, Dong Guo, Renrui Zhang, Feng Li, Hao Zhang, Kaichen Zhang, Peiyuan Zhang, Yanwei Li, Ziwei Liu, et al. Llava-onevision: Easy visual task transfer. *arXiv preprint arXiv:2408.03326*, 2024.

[31] Opher Lieber, Barak Lenz, Hofit Bata, Gal Cohen, Jhonathan Osin, Itay Dalmedigos, Erez Safahi, Shaked Meirom, Yonatan Belinkov, Shai Shalev-Shwartz, et al. Jamba: A hybrid transformer-mamba language model. *arXiv preprint arXiv:2403.19887*, 2024.

[32] Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. Parrot: Efficient serving of llm-based applications with semantic variable. *arXiv preprint arXiv:2405.19888*, 2024.

[33] Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, et al. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*, 2024.

[34] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. Visual instruction tuning. *Advances in neural information processing systems*, 36, 2024.

[35] Xiaoxuan Liu, Cade Daniel, Langxiang Hu, Woosuk Kwon, Zhuohan Li, Xiangxi Mo, Alvin Cheung, Zhijie Deng, Ion Stoica, and Hao Zhang. Optimizing speculative decoding for serving large language models using goodput. *arXiv preprint arXiv:2406.14066*, 2024.

[36] NVIDIA. Tensorrt-llm: High-performance inference for large language models, 2024. Accessed: 2024-12-08.

[37] Antonio Orvieto, Samuel L Smith, Albert Gu, Anushan Fernando, Caglar Gulcehre, Razvan Pascanu, and Soham De. Resurrecting recurrent neural networks for long sequences. In *International Conference on Machine Learning*, pages 26670–26698. PMLR, 2023.

[38] Rui Pan, Zhuang Wang, Zhen Jia, Can Karakus, Luca Zancato, Tri Dao, Ravi Netravali, and Yida Wang. Marconi: Prefix caching for the era of hybrid llms. *arXiv preprint arXiv:2411.19379*, 2024.

[39] Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Chung, Matteo Grella, et al. Rwkv: Reinventing rnns for the transformer era. *arXiv preprint arXiv:2305.13048*, 2023.

[40] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. vattention: Dynamic memory management for serving llms without pagedattention. *arXiv preprint arXiv:2405.04437*, 2024.

[41] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu Mooncake. Kimi's kvcache-centric architecture for llm serving. *arXiv preprint arXiv:2407.00079*, 2024.

[42] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *arXiv preprint arXiv:2407.08608*, 2024.

[43] Noam Shazeer. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019.

[44] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, et al. Slora: Scalable serving of thousands of lora adapters. *Proceedings of Machine Learning and Systems*, 6:296–311, 2024.

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

[46] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. Powerinfer: Fast large language model serving with a consumer-grade gpu. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 590–606, 2024.

[47] Andreas Steiner, André Susano Pinto, Michael Tschannen, Daniel Keysers, Xiao Wang, Yonatan Bitton, Alexey Gritsenko, Matthias Minderer, Anthony Sherbondy, Shangbang Long, et al. Paligemma 2: A family of versatile vlms for transfer. *arXiv preprint arXiv:2412.03555*, 2024.

[48] Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, et al. Gemma 2: Improving open language models at a practical size. *arXiv e-prints*, pages arXiv–2408, 2024.

[49] ShareGPT Team. Sharegpt: Share your wildest chatgpt conversations with one click, 2024. Accessed: 2024-12-09.

[50] A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

[51] vLLM Team. vLLM v0.6.0: 2.7x Throughput Improvement and 5x Latency Reduction — blog.vllm.ai. `https://blog.vllm.ai/2024/09/05/perf-update.html`. [Accessed 10-12-2024].

[52] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. Loongserve: Efficiently serving long-context large language models with elastic sequence parallelism. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 640–654, 2024.

[53] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. *arXiv preprint arXiv:2309.17453*, 2023.

[54] Leyang Xue, Yao Fu, Zhan Lu, Luo Mai, and Mahesh Marina. Moe-infinity: Activation-aware expert offloading for efficient moe serving. *arXiv preprint arXiv:2401.14361*, 2024.

[55] Dongjie Yang, XiaoDong Han, Yan Gao, Yao Hu, Shilin Zhang, and Hai Zhao. Pyramidinfer: Pyramid kv cache compression for high-throughput llm inference. *arXiv preprint arXiv:2405.12532*, 2024.

[56] Zihao Ye, Lequn Chen, Ruihang Lai, Yilong Zhao, Size Zheng, Junru Shao, Bohan Hou, Hongyi Jin, Yifei Zuo, Liangsheng Yin, Tianqi Chen, and Luis Ceze. Accelerating self-attentions for llm serving with flashinfer, February 2024.

[57] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, 2022.

[58] Xiang Yue, Tianyu Zheng, Yuansheng Ni, Yubo Wang, Kai Zhang, Shengbang Tong, Yuxuan Sun, Botao Yu, Ge Zhang, Huan Sun, et al. Mmmu-pro: A more robust multi-discipline multimodal understanding benchmark. *arXiv preprint arXiv:2409.02813*, 2024.

[59] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Efficiently programming large language models using sglang. *arXiv e-prints*, pages arXiv–2312, 2023.

[60] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, 2024.