Seminar Cloud Computing

GPU Memory Management and Optimization in the Cloud

Kadir Kalkan
Technische Universität München

04.12.2042

Abstract

The growing size of Key-Value (KV) caches with increasing context lengths in Large Language Models (LLMs) introduces significant GPU memory management challenges in cloud environments. Traditional memory allocation methods are constrained by inefficiencies such as fragmentation and redundant duplication, limiting scalability and cost-effectiveness. This paper examines the PagedAttention method, which segments KV caches into smaller blocks to reduce fragmentation, and the vLLM serving engine, which enables dynamic memory sharing across distributed GPUs. Additionally, the paper discusses the performance of this proposed method, identifies its limitations, and proposes improvements for efficient and scalable memory management in cloud-based LLM services.

1 Introduction

The rapid development of Large Language Models (LLMs) has revolutionized natural language processing, enabling applications such as chatbots, search engines, and coding assistants[1][2]. However, as LLMs scale in both model size and sequence length, the demands for computational and memory resources have grown exponentially. Consequently, optimizing LLM inference has become a pressing concern, particularly in cloud computing environments where scalability and cost-effectiveness are paramount.

One of the most significant challenges in deploying LLMs at scale is the disparity between advancements in GPU computational throughput and memory capacity. While computational speed has seen rapid improvements in recent GPUs, memory capacity has not kept pace, remaining a critical bottleneck for large-scale applications. This limitation poses a bottleneck in scenarios where memory requirements, driven by components such as the Key-Value (KV) cache, outpace computational demands. Unlike static model weights, the KV cache scales dynamically with the context length, storing the keys and values of all tokens in a sequence to facilitate efficient reference to prior context. As context lengths increase, the memory footprint of the KV cache can exceed that of the model parameters, underscoring the need for innovative memory management strategies.

The memory challenges faced by LLMs are compounded in cloud environments. Cloud-based LLM services often encounter highly dynamic workloads characterized by unpredictable sequence lengths and varied batch sizes. Traditional memory allocation strategies, such as reserving memory for the maximum possible sequence length, result in significant inefficiencies due to internal and external fragmentation. Additionally, advanced decoding techniques worsen memory demands by duplicating KV cache entries across multiple candidate sequences. These factors collectively limit the scalability and cost-efficiency of LLM serving systems, especially in high-throughput scenarios.

To address these challenges, PagedAttention and vLLM framework [8] introduced efficient memory management techniques inspired by operating system concepts. PagedAttention segments the KV cache into smaller, dynamically allocated blocks, akin to paging in virtual memory systems. This approach minimizes memory fragmentation and enables efficient sharing across GPU workers. Meanwhile, vLLM incorporates a centralized scheduler that coordinates the execution of distributed GPU workers, leveraging the benefits of PagedAttention to optimize memory utilization.

This paper builds upon these techniques, focusing on

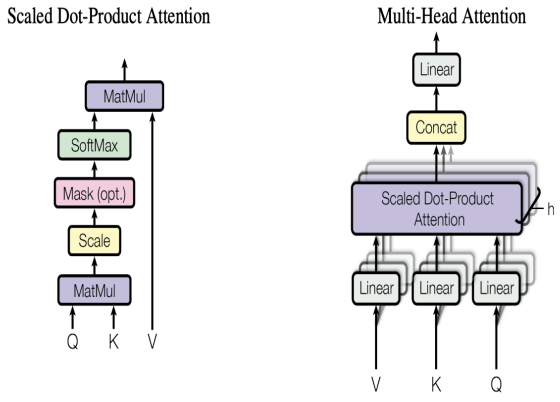

Figure 1: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

GPU memory optimization for LLM inference in the cloud. In addition to exploring the core principles of PagedAttention and vLLM, its performance and some possible enhancements are mentioned.

In the following sections of this paper, Section 2 provides background on transformer architectures and KV cache usage, highlighting the motivation for efficient memory management. Section 3 delves into the design and implementation of PagedAttention and vLLM, while Section 4 discusses its performance and potential enhancements with existing related works. Finally, Section 5 summarizes this work.

2 Background and Motivation

The introduction of the Transformer model [15] marked a significant breakthrough in deep learning for sequential data processing, particularly in natural language processing. The core architecture of Transformers relies on self-attention mechanisms, which enable models to capture long-range dependencies across input sequences by calculating pairwise attention scores between tokens. Transformers are composed of stacked *transformer blocks*, each of which includes an *attention layer* and a *multi-layer perceptron* (MLP) module.

In the context of Large Language Models (LLMs), transformers are utilized in an *auto-regressive* fashion, where the model predicts each token based on previous tokens in the sequence. To efficiently perform attention over long sequences, a cache of *Key* and *Value* pairs, referred to as the *KV cache*, is maintained. This cache grows linearly with the sequence length, presenting unique memory management challenges as context length increases.

2.1 Transformer Attention Mechanism and KV Cache

In the transformer architecture, each input token is represented by a feature vector $\mathbf{X}_i$. For each token, three distinct projections, known as the *Query* ($\mathbf{Q}_i$), *Key* ($\mathbf{K}_i$), and *Value* ($\mathbf{V}_i$) vectors, are generated through linear transformations:

$$\mathbf{Q}_i = \mathbf{W}_q \mathbf{X}_i, \quad \mathbf{K}_i = \mathbf{W}_k \mathbf{X}_i, \quad \mathbf{V}_i = \mathbf{W}_v \mathbf{X}_i \tag{1}$$

where $\mathbf{W}_q$, $\mathbf{W}_k$, and $\mathbf{W}_v$ are learned weight matrices. The attention mechanism then computes the similarity between the query $\mathbf{Q}_i$ for the current token and the keys $\mathbf{K}_j$ of all previous tokens to generate an attention score for each pair. This is mathematically represented as:

$$\mathbf{A}_i = \text{softmax}\left(\frac{\mathbf{Q}_i \mathbf{K}_j^\top}{\sqrt{d_k}}\right), \quad j = 1, \ldots, i-1 \tag{2}$$

where d_k is the dimension of the keys, and the softmax function normalizes the attention scores. These scores determine the weight with which each value $\mathbf{V}_j$ contributes to the output for the current token.

The output for the token i, denoted by $\mathbf{O}_i$, is a weighted sum of the value vectors of all preceding tokens:

$$\mathbf{O}_i = \sum_{j=1}^{i-1} A_{ij} \mathbf{V}_j \tag{3}$$

where A_{ij} represents the normalized attention score between tokens i and j. Finally, $\mathbf{O}_i$ is projected to the output space using a fully connected layer:

$$\mathbf{Y}_i = \mathbf{W}_o \mathbf{O}_i \tag{4}$$

where $\mathbf{W}_o$ is a learned weight matrix for the output layer See Figure 1.

2.2 KV Cache Usage in LLMs

- **Prefill Phase:** In the prefill phase, the entire user prompt is processed. The key and value vectors for all prompt tokens are computed and stored in the KV cache. This allows subsequent token generation to effectively leverage the prompt's context.
- **Autoregressive Generation Phase:** This phase sequentially generates the remaining output tokens, one token at a time. During each

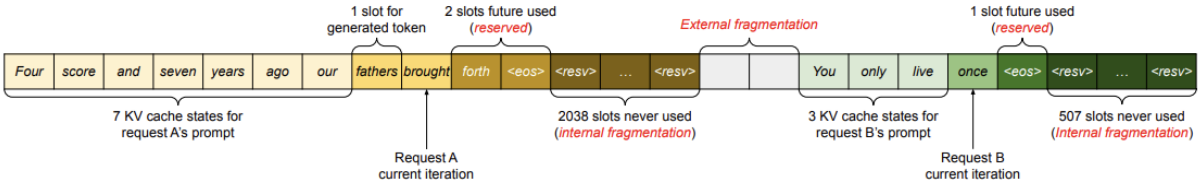

Figure 2: KV cache memory management in existing systems

step, the model relies on the KV cache to retrieve the key and value vectors of previously generated tokens, ensuring contextual continuity in the generated text. This phase concludes when the sequence either reaches its maximum allowable length (as set by the user or constrained by the LLM) or when an end-of-sequence (`<eos>`) token is generated.

2.3 Motivation for Efficient Memory Management

In auto-regressive LLMs, as the sequence grows, the model needs to keep track of an increasingly large number of past tokens, which is facilitated by the KV cache. The KV cache stores the **K** and **V** vectors for all previous tokens, allowing the model to compute attention scores without re-computing these vectors for each new token. This caching mechanism significantly enhances efficiency but introduces a memory challenge.

- **Memory Fragmentation:** Fixed memory allocation schemes, such as reserving space for the maximum possible sequence length, can result in significant internal memory fragmentation, particularly when handling requests with varying input and output lengths. Internal fragmentation refers to allocated memory that remains unused due to the misalignment of memory block sizes and request requirements.
- **Redundant Duplication:** Advanced decoding algorithms, such as beam search, involve generating multiple output candidate sequences, often leading to redundant duplication of KV cache entries. This duplication further worsen memory consumption, limiting the system's ability to handle larger batch sizes.

PagedAttention and the vLLM serving engine were designed to address these issues. vLLM employs a centralized scheduler to orchestrate distributed GPU workers effectively. This system includes a KV cache manager that leverages PagedAttention to optimize KV cache memory usage by adopting a paged approach inspired by virtual memory management in operating systems. The KV cache is divided into smaller blocks, enabling dynamic allocation and efficient sharing across GPU workers, thus reducing both fragmentation and redundant memory usage.

3 PagedAttention and vLLM

PagedAttention [8] is a technique that optimizes KV cache management for LLMs. Inspired by the paging technique used in operating systems, PagedAttention segments the KV cache into smaller "pages" to reduce memory fragmentation and facilitate efficient memory usage across multiple requests. This approach allows the memory to grow and shrink dynamically, reducing the wastage commonly associated with large, monolithic memory allocations.

3.1 Memory Management in Existing Systems

In traditional large language model (LLM) serving systems, key-value (KV) caches for a single request are stored as contiguous tensors across all token positions[8]. This design decision stems from the need to accommodate deep learning frameworks, where most operations are optimized for contiguous memory storage. To handle the unpredictable lengths of outputs, these systems allocate memory based on the maximum potential sequence length, regardless of the actual input or final output length.

As illustrated in Figure 2, memory pre-allocation introduces inefficiencies. For example, in two separate requests, one with a maximum length of 2048 and another with 512, three primary types of memory waste occur:

- **Reserved memory slots** for tokens that might

not be generated during inference,

- **Internal fragmentation**, which results from overestimating the sequence length, leading to unused space within allocated memory blocks,
- **External fragmentation** from memory allocators like the buddy allocator, where leftover memory remains unusable for token generation.

These forms of fragmentation lead to significant memory waste.

PagedAttention tackles this problem by drawing inspiration from operating system (OS) paging techniques. Rather than allocating a large, contiguous memory chunk for the entire potential sequence length, PagedAttention organizes the KV cache in smaller, manageable memory "pages." This approach allows memory to be dynamically allocated and freed as tokens are generated, adapting to the actual sequence length and reducing unused memory. By doing so, PagedAttention minimizes memory waste, reduces fragmentation, and enables better memory sharing across multiple decoding processes, leading to more efficient memory management during LLM serving.

3.2 Method

PagedAttention introduces a significant improvement over traditional attention algorithms by enabling the storage of continuous keys and values within non-contiguous memory spaces. This approach relies on partitioning the key-value (KV) cache of each sequence into distinct KV blocks, each containing the key and value vectors associated with a predetermined number of tokens. During attention computation, the PagedAttention kernel efficiently identifies and retrieves these KV blocks individually, enhancing memory organization and reducing computational overhead by streamlining access patterns.

A distributed LLM serving engine, vLLM, is built top of Pagedattention and it adopts a centralized scheduler to coordinate the execution of distributed GPU workers. The key idea behind vLLM's memory manager is similar to the virtual memory in operating systems. vLLM applies concepts from virtual memory to manage the KV cache in large language model (LLM) services. This approach, enabled by PagedAttention, organizes the KV cache as fixed-size KV blocks, similar to pages in virtual memory. Each request's KV cache is represented as a series of logical KV blocks that are filled sequentially, from left to right, as new tokens and their associated KV caches are generated.

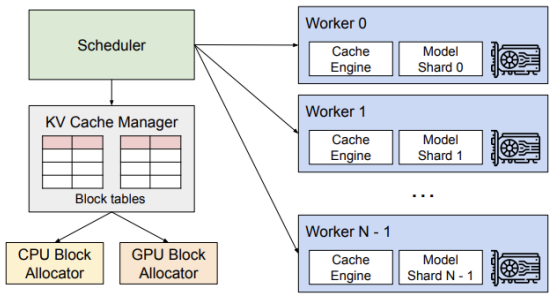

Figure 3: vLLM system overview.

On GPU workers, a block engine allocates a contiguous chunk of GPU DRAM, which is then divided into physical KV blocks; the same division also occurs in CPU RAM for swapping purposes. The KV cache manager handles the physical KV cache memory on GPU workers using instructions provided by the centralized scheduler. (see Figure 3).

The KV block manager in vLLM introduces block tables that map logical KV blocks to their physical counterparts for each request. Each entry in these tables keeps track of the physical blocks associated with a specific logical block as well as the count of filled positions. This separation between logical and physical KV blocks enables dynamic expansion of the KV cache memory, eliminating the need for reserving memory for all potential positions upfront. As a result, it significantly reduces memory waste compared to traditional systems, which often allocate memory inefficiently by pre-reserving unused space. Figure 4 demonstrates how vLLM executes PagedAttention and manages memory during the decoding process of a single input sequence:

1. **Memory allocation during prompt processing:** Similar to the behavior of virtual memory in operating systems, vLLM does not need to allocate memory upfront for the maximum possible sequence length. Instead, it allocates only the necessary KV blocks for storing the KV cache generated during prompt processing. In this case, the prompt contains 7 tokens, so vLLM maps the first 2 logical KV blocks (0 and 1) to 2 physical KV blocks (7 and 1, respectively). During the prefill phase, vLLM generates the KV cache for the prompts and the initial output token using a conventional self-attention algorithm. The KV cache for the first 4 tokens is stored in logical block 0, and the next 3 tokens are placed in logical block 1. Any remaining slots are reserved for subsequent autoregressive gener-

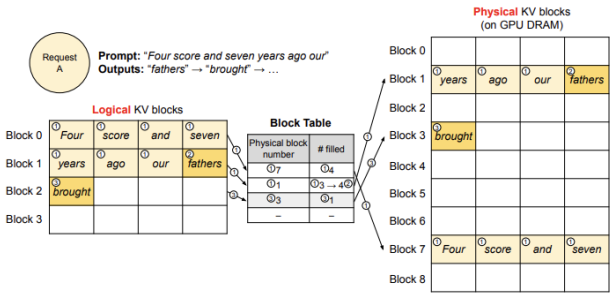

Figure 4: Block table translation in vLLM.

ation steps.

2. **Initial autoregressive decoding step:** During the first autoregressive decoding step, vLLM generates a new token with the PagedAttention algorithm using physical blocks 7 and 1. Since one slot remains available in the last logical block, the newly generated KV cache is stored there, and the block table's filled record is updated.

3. **Second decoding step and new physical block allocation:** In the second decoding step, when the last logical block becomes fully utilized, the next generated KV cache is stored in a new logical block. To achieve this, a new physical block (physical block 3) is allocated, and the block table is updated accordingly to reflect this new mapping.

3.2.1 Scheduling and Preemption

In each decoding iteration, vLLM begins by selecting a set of candidate sequences for batching, allocating physical blocks for any new logical blocks that are needed. If the request traffic exceeds the system's capacity, vLLM prioritizes requests using a first-come-first-serve (FCFS) scheduling policy, ensuring fair access and avoiding request starvation. After determining the batch, vLLM concatenates the input tokens for the current iteration into a single sequence, which it feeds to the large language model (LLM). During computation, vLLM employs the PagedAttention kernel to access the existing KV cache, stored as logical KV blocks, and to save any newly generated KV cache into physical KV blocks. By allowing multiple tokens to be stored within a single KV block, the PagedAttention kernel can process the KV cache across a greater number of positions in parallel, optimizing hardware utilization and reducing overall latency.

LLM services encounter a distinct challenge as input prompt lengths can vary widely, and the length of each output is unpredictable, depending on both the prompt and the model. As the volume of requests and outputs increases, vLLM may exhaust the available GPU physical blocks needed to store newly generated KV cache. Traditional eviction policies typically rely on heuristics to predict which block will be accessed furthest in the future, evicting that block to free up space. However, because all blocks within a sequence are accessed together in vLLM, an all-or-nothing eviction policy is applied—either all blocks of a sequence are evicted or none at all. Additionally, sequences within the same group are always preempted or rescheduled together due to shared memory considerations. When it is necessary to recover an evicted block, vLLM employs two strategies: swapping and recomputation.

- **Swapping:** In vLLM, when physical blocks on the GPU are needed for new tokens, the system copies evicted blocks to CPU memory to manage limited GPU resources. As illustrated in Figure 3, vLLM includes a CPU block allocator alongside the GPU block allocator to oversee the physical blocks swapped to CPU RAM. When GPU memory runs low on free physical blocks, vLLM selects certain sequences for eviction, transferring their KV cache to the CPU to free up space for incoming data. This design enables vLLM to handle larger workloads without compromising memory availability on the GPU.

- **Recomputation:** In this case, vLLM simply recompute the KV cache when the preempted sequences are rescheduled.

vLLM employs both recomputation and swapping as recovery mechanisms for KV cache, with each method suited to different conditions. Experimental results indicate that swapping is costly when block sizes are small, as it requires frequent, small data transfers between CPU and GPU. Conversely, the overhead for recomputation remains constant across block sizes, as it bypasses KV block transfers entirely. This makes recomputation more efficient for small block sizes, while swapping becomes advantageous with larger block sizes. Choosing an optimal block size is crucial: if blocks are too small, the GPU's parallel processing capabilities are underutilized; if blocks are too large, internal fragmentation and reduced sharing potential increase. vLLM experiments reveal that a block size

of 16 effectively balances GPU utilization with minimal fragmentation for most workloads, leading vLLM to adopt 16 as its default block size.

3.2.2 Other Decoding scenarios

How PagedAttention and vLLM perform simple decoding is shown. In this section, how they perform more complex decoding scenarios that exhibit complex access patterns and more opportunities for memory sharing will be demonstrated.

1. **Parallel sampling:** In LLM-based program assistants, LLMs often generate multiple outputs for a single input prompt, giving users a range of options[1][6]. vLLM leverages its PagedAttention and paged memory management capabilities to efficiently share KV cache memory across these sequences. For example, during parallel decoding, shared prompts only reserve one copy of the prompt state, with logical blocks mapped to common physical blocks. This memory sharing is managed through reference counts, and when modifications are necessary, vLLM uses a copy-on-write mechanism to allocate new physical blocks, ensuring efficient and isolated memory updates for each sequence.

2. **Shared prefix:** Typically, LLM users provide a detailed task description, including instructions and example inputs and outputs, referred to as a system prompt. This description is concatenated with the specific task input to create the prompt for the request, and the LLM generates outputs based on the complete prompt. For applications with shared prefixes among multiple user prompts, LLM service providers can store the KV cache of the prefix in advance to minimize redundant computation. In vLLM, this is facilitated by reserving physical blocks for predefined shared prefixes, akin to how operating systems manage shared libraries across processes. User input prompts with the shared prefix can map their logical blocks to these cached physical blocks, with the last block marked for copy-on-write. As a result, prompt phase computation only needs to process the user's specific task input.

3.2.3 Distributed Execution

Large language models (LLMs) often have parameter sizes that exceed the capacity of a single GPU, necessitating model partitioning across multiple GPUs and executing in a model-parallel manner. vLLM accommodates distributed settings effectively by implementing Megatron-LM-style tensor model parallelism for Transformer models[13], following an SPMD (Single Program Multiple Data) execution approach. This setup partitions linear layers for block-wise matrix multiplication, where GPUs synchronize intermediate results through an all-reduce operation. Specifically, the attention operation is divided by attention head dimension, allowing each SPMD process to handle a subset of attention heads.

In this configuration, different GPU workers share access to both the KV block manager and the logical-to-physical block mappings. This unified mapping enables each GPU worker to operate using physical blocks specified by the scheduler for each input request, as illustrated 3. Although all GPU workers have access to the same physical block IDs, each worker only stores part of the KV cache for the attention heads it processes. For each decoding step, the scheduler prepares input token IDs and block tables for all batch requests, broadcasting these to GPU workers. The workers then execute the model using the provided input token IDs and retrieve the KV cache entries according to the block table. Throughout execution, GPU workers independently synchronize intermediate results using all-reduce without further intervention from the scheduler.

At the end of each iteration, GPU workers send the sampled tokens back to the scheduler. This design means GPU workers do not require frequent synchronization on memory management; they receive all necessary memory management data at the beginning of each decoding iteration along with the step inputs, allowing them to proceed autonomously.

3.3 Performance

The performance evaluation of vLLM and PagedAttention primarily uses NVIDIA A100 GPUs, configured in various ways to assess throughput and memory efficiency for different models and workloads. Evaluated models include OPT models of sizes 13B, 66B, and 175B parameters. Experiments were performed using configurations ranging from a single NVIDIA A100 GPU (40 GB) for smaller models to multi-GPU setups for larger ones (e.g., 8 GPUs for OPT-175B). For workload testing, two datasets—ShareGPT and Alpaca—were used to simulate real-world conversational and instructional tasks.

In terms of performance, vLLM demonstrates significant improvements over state-of-the-art systems

like FasterTransformer[11] and Orca[16]. On the ShareGPT dataset, vLLM achieves 1.7× to 2.7× higher request rates compared to the most optimized versions of Orca and up to 22× higher request rates compared to FasterTransformer. This improvement stems from vLLM's ability to efficiently manage KV cache memory through PagedAttention, allowing more requests to be batched and processed concurrently without incurring memory inefficiencies. Similarly, on the Alpaca dataset, vLLM maintains higher throughput, particularly in scenarios with long or complex sequences, by optimizing KV cache sharing and block-level memory management.

Context length	**10k**	**100k**	**500k**	**1000k**
KV Cache size	8.19GB	81.9GB	409.6GB	819.2GB
Misc size	26GB	26GB	26GB	26GB

Table 1: LLaMA2-13B[14], KV Cache size with context length

4 Discussion

Despite vLLM's ability to improve throughput through the PagedAttention technique, which distributes memory across multiple GPU workers, it does have certain limitations. This section will examine the challenges associated with PagedAttention and vLLM. Additionally some suggestions about GPU memory optimizaitons for LLMs in cloud will be provided. By analyzing these limitations, the constraints of vLLM's current design can be better understood and potential areas for improvement can be explored.

4.1 Memory challenges for LLM serving on the cloud:

In this section some potetntial improvments for memory challanges for LLMs serving on the cloud will be provided:

- **Inequalities in memory demands obstacles efficient model parallelism:**The significant difference in memory demands creates obstacles for efficient model parallelism. Unlike the continuously growing KV Cache during the autogeneration process, memory needs for the remaining activation tensors stay constant, as shown in Table 1[9]. This imbalance between the attention layer and other model layers presents a major challenge in implementing efficient model parallelism. To accommodate the large KV Cache necessary for handling long-context tasks, more GPUs must be utilized. However, tensor sizes in non-attention layers do not scale with increasing context lengths. As a result, traditional model parallelism involves finer subdivisions of these layers across more GPUs, which reduces resource utilization efficiency.

  PagedAttention has proposed splitting KV Caches into smaller blocks to enable finer memory management and reduce fragmentation. However, this solution still require collecting and arranging all blocks within the local GPU memory for attention computations. However, If KV Caches are distributed and attention modules are executed across multiple resources the vast capabilities of cloud environments can be better leveraged.

- **Inefficient resource management in the cloud environment because of the dynamicity of KV Cache size:**The variable size of the KV Cache complicates efficient resource management in cloud environments. Due to the auto-regressive design in LLMs, the final sequence length remains unknown until the generation process completes, usually marked by an "end" token. Consequently, memory requirements are highly dynamic and unpredictable, ranging from several gigabytes to hundreds of gigabytes.

  This unpredictability makes pre-planned resource allocation unfeasible. Resources need to be dynamically allocated and released according to the real-time needs of the auto-regressive process. If the memory demand for a given context surpasses the available GPU capacity in an instance, the entire task must be transferred to a larger instance with more GPUs, a process called live migration. Live migration is resource-intensive and, according to experiments of Infinite-LLM[9], can be up to 25 times more costly than standard inference. Alternatively, assigning additional GPUs from the beginning leads to resource waste for tasks with shorter contexts, further complicating efficient resource allocation.

  PagedAttention manages KV Caches by breaking them into fine-grained sub-blocks, similar to how operating systems handle pages. However, this approach is limited to using CPU memory for swapping, which is inefficient in cloud environments. The finite capacity of CPU memory restricts the maximum context length that LLM

services can support and does not fully utilize the extensive memory resources available across cloud systems. Leveraging the combined memory capabilities of GPUs and CPUs in data centers could help create an efficient memory pool designed specifically for LLM services, enabling the effective handling of extremely long context lengths.

Infinite-LLM[9] introduces a suite of key techniques aimed at addressing major challenges in serving large language models (LLMs) with extended context lengths. To tackle the first challenge of memory imbalance, it offers a new attention algorithm called DistAttention, which breaks down traditional attention computations into smaller units called macro-attentions (MAs) and their associated KV Caches (rBlocks). This approach separates KV Cache computation from the transformer block, allowing independent model parallelism and more efficient memory management for attention layers compared to other layers within the transformer architecture. For non-attention layers, established model parallelism strategies are applied, while attention layers adaptively allocate memory and computational resources based on KV Cache fluctuations across the data center.

To address the second challenge, Infinite-LLM introduces DistKV-LLM, a distributed LLM service engine that integrates seamlessly with DistAttention. DistKV-LLM optimizes KV Cache management by coordinating memory usage across GPUs and CPUs in the data center. When a memory shortfall arises due to KV Cache growth, it identifies and borrows memory space from less burdened instances. This process is facilitated by two main components: the rManager, which virtualizes GPU and CPU memories within each instance to handle both local and remote memory requests, and the gManager, which serves as a global coordinator to ensure effective resource management among distributed rManagers. Additionally, DistKV-LLM employs the DGFM algorithm to mitigate memory fragmentation within distributed KV Caches, ensuring efficient memory usage and improving the performance and reliability of LLM services.

Infinite-LLM demonstrates significant performance advantages over vLLM by offering superior throughput and support for longer context lengths. When compared directly to vLLM, Infinite-LLM achieves 1.4 to 5.3 times higher throughput due to its efficient memory coordination and parallelism strategies, while also supporting context lengths that are 2 to 19 times longer. This performance boost is primarily attributed to its use of distributed memory management via the DistKV-LLM system, which dynamically allocates memory across distributed GPUs and CPUs, minimizing resource wastage and reducing performance bottlenecks often faced by vLLM under large memory demands and variable context lengths.

4.2 Eviction Policies

vLLM encounters limitations and inefficiencies when dealing with the memory-intensive needs of LLMs due to its reliance on recomputation and swapping.

- **Swapping Overheads:** vLLM swaps evicted KV cache blocks between GPU and CPU memory to manage the limited GPU memory capacity. While swapping can temporarily resolve memory constraints, it introduces latency due to the time required for I/O data transfer between GPU and CPU. This I/O-bound process can significantly slow down LLM response times, particularly with larger model sizes or longer sequences.
- **Recomputation Costs:** For evicted KV cache blocks, PagedAttention may rely on recomputation, which involves recalculating KV values from the original tokens. Although recomputation avoids the need for storage, it is highly GPU-intensive, as it requires performing the attention and feed-forward computations for each token in the evicted sequence. As a result, recomputation can increase response latency and limit the efficiency of batch processing, especially when a high volume of requests or large model sizes are in use.

To address these limitations, a new state restoration technique could be applied to enhance the efficiency of KV cache management. This technique could focus on reducing both computational and I/O overheads by utilizing a more balanced and resource-efficient approach. By optimizing state restoration without relying solely on recomputation or swapping

HCache[4] addresses the latency challenges in LLM state restoration by leveraging intermediate activations, or hidden states, instead of original tokens for KV cache restoration. Since these hidden states are only half the size of the KV cache, this approach significantly reduces I/O transmission times, cutting them by half compared to conventional KV offloading. Additionally, by using hidden states rather than the complete tokens, HCache eliminates the need to

recompute the full attention and feed-forward operations, resulting in a sixfold decrease in computation time. HCache's pipeline efficiency further enhances performance by simultaneously handling I/O transmission and recomputation, effectively minimizing idle time (or pipeline bubbles) and balancing resource utilization across computation and data transfer tasks.

HCache provides an accelerated state restoration process that mitigates the latency imposed by recomputation and I/O-intensive swapping. This method can reduce the time-to-first-token (TTFT) by up to 1.93 times over KV offloading and by up to 5.73 times over token recomputation approaches, making it a highly effective solution for high-throughput language model serving.

4.3 KV-cache storage in noncontiguous virtual memory

Despite drawing inspiration from demand paging, PagedAttention diverges from conventional demand paging by requiring applications to modify their code to handle dynamically allocated physical memory. In contrast, traditional demand paging is designed to be transparent to applications. Below are the main issues that arise from this approach:

- **Requires re-writing the attention kernel (GPU code):** For objects stored in virtually contiguous memory, index-based lookups allow for straightforward and efficient access. However, by storing the KV-cache in non-contiguous virtual memory, PagedAttention necessitates rewriting the GPU code so the attention kernel can access all elements of the KV-cache appropriately. This code modification requirement creates a barrier to utilizing new attention optimizations in production environments.
- **Adds software complexity and redundancy (CPU code):** PagedAttention also imposes additional responsibilities on developers by requiring a memory manager within the serving framework. This manager must handle the (de)allocation of KV-cache and keep track of dynamically allocated KV-cache blocks. In essence, this approach duplicates OS-level demand paging functionality within user code, adding complexity and redundancy.
- **Introduces performance overhead:** PagedAttention can slow down execution in critical paths due to two main factors. First, GPU kernels need to perform extra operations to retrieve KV-cache data from non-contiguous memory blocks, often slowing down attention computations by more than 10%. Additionally, the user-space memory manager can add CPU overhead, increasing costs by up to another 10%.

vAttention[12] leverages a distinctive approach to memory management by separating the allocation of virtual and physical memory for the KV-cache. Specifically, it reserves a large, contiguous virtual memory buffer for the KV-cache at the start, much like traditional reservation-based allocators. However, unlike these allocators, vAttention defers the allocation of physical memory to runtime, adding physical memory blocks only as they are needed. This is similar to the dynamic allocation concept in PagedAttention but without forcing non-contiguous virtual memory. By preserving virtual contiguity, vAttention avoids the fragmentation and associated overhead of managing scattered memory blocks.

vAttention works well in modern systems where memory limitations primarily concern physical, not virtual, memory. For example, 64-bit systems commonly offer a vast virtual address space, often up to 128TB per process, allowing ample room for contiguous virtual memory reservations. This setup enables vAttention to reduce memory waste and improve performance while maintaining a simplified, contiguous virtual memory layout for the KV-cache.

5 Summary

This paper explores GPU memory management challenges for large language models (LLMs) in cloud environments, emphasizing solutions for managing the dynamic growth of Key-Value (KV) caches. It evaluates PagedAttention, which segments KV caches to reduce memory fragmentation, and the vLLM serving engine, which coordinates memory allocation across distributed GPUs.

Although vLLM offers a promising approach to efficient memory management for LLMs, there are still areas that require improvement, such as developing more adaptive eviction policies and implementing scalable memory pooling across GPUs and CPUs. Future work could focus on integrating these enhancements into vLLM, leveraging the techniques discussed in the paper

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[2] Rohan Anil, Andrew M Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, et al. Palm 2 technical report. *arXiv preprint arXiv:2305.10403*, 2023.

[3] Felix Brakel, Uraz Odyurt, and Ana-Lucia Varbanescu. Model parallelism on distributed infrastructure: A literature review from theory to llm case-studies. *arXiv preprint arXiv:2403.03699*, 2024.

[4] Shiwei Gao, Youmin Chen, and Jiwu Shu. Fast state restoration in llm serving with hcache. *arXiv preprint arXiv:2410.05004*, 2024.

[5] Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W Mahoney, and Kurt Keutzer. Ai and memory wall. *IEEE Micro*, 2024.

[6] Github. https://github.com/features/copilot, 2022.

[7] Cunchen Hu, Heyang Huang, Junhao Hu, Jiang Xu, Xusheng Chen, Tao Xie, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, et al. Memserve: Context caching for disaggregated llm serving with elastic memory pool. *arXiv preprint arXiv:2406.17565*, 2024.

[8] 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.

[9] Bin Lin, Chen Zhang, Tao Peng, Hanyu Zhao, Wencong Xiao, Minmin Sun, Anmin Liu, Zhipeng Zhang, Lanbo Li, Xiafei Qiu, et al. Infinite-llm: Efficient llm service for long context with distattention and distributed kvcache. *arXiv preprint arXiv:2401.02669*, 2024.

[10] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context. *arXiv preprint arXiv:2310.01889*, 2023.

[11] NVIDIA. Fastertransformer, 2023. Accessed: 2023.

[12] 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.

[13] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[14] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[15] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. advances in neural information processing systems. *Advances in neural information processing systems*, 30(2017), 2017.

[16] 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.

[17] Junzhe Zhang, Sai Ho Yeung, Yao Shu, Bingsheng He, and Wei Wang. Efficient memory management for gpu-based deep learning systems. *arXiv preprint arXiv:1903.06631*, 2019.

[18] Zixuan Zhou, Xuefei Ning, Ke Hong, Tianyu Fu, Jiaming Xu, Shiyao Li, Yuming Lou, Luning Wang, Zhihang Yuan, Xiuhong Li, et al. A survey on efficient inference for large language models. *arXiv preprint arXiv:2404.14294*, 2024.