Paper Reproduction: Efficient Memory Management for Large Language Model Serving with PagedAttention

Abstract

This paper presents a reproduction study of the work "Efficient Memory Management for Large Language Model Serving with PagedAttention," which addresses critical inefficiencies in memory management for large language model (LLM) serving. The original paper proposes PagedAttention, an innovative attention algorithm inspired by operating systems' virtual memory management, to tackle internal and external memory fragmentation issues. We validate the effectiveness of this approach using the vLLM framework, which incorporates PagedAttention for enhanced KV cache management and advanced request scheduling techniques. Our experiments confirm significant improvements in throughput, reduced resource wastage, and maintained model accuracy, underscoring the framework's robustness in high-demand LLM serving environments.

Keywords: PagedAttention, virtual memory management, batching techniques, inference optimization.

1 Introduction

In this paper reproduction, I delve into the paper "Efficient Memory Management for Large Language Model Serving with PagedAttention [1]", a pivotal study addressing the challenges of memory management in serving large language models (LLMs). The increasing demand for applications powered by LLMs, such as programming assistants and universal chatbots, necessitates efficient resource utilization due to the high operational costs associated with LLM requests. Traditional systems face significant inefficiencies, particularly in managing the KV cache memory, which is crucial for maximizing throughput.

A notable issue in these systems is the large device memory waste, characterized by reserved memory slots and internal fragmentation caused by over-provisioning memory based on the potential maximum sequence length of a request. Additionally, external fragmentation occurs due to unused blocks of memory left by memory allocators, and non-shared memory during decoding results in the KV cache not being shared. These factors contribute to suboptimal memory usage and increased costs.

Moreover, low inference performance is exacerbated by asynchronous request arrival and variability in request sizes, which traditional batching methods handle inefficiently by processing entire batches before moving on to new requests. The original paper introduces PagedAttention, an innovative attention algorithm inspired by operating systems' virtual memory management techniques. By dividing the KV cache into non-contiguous blocks, PagedAttention mitigates internal and external fragmentation and facilitates memory sharing.

The authors demonstrate the effectiveness of this approach through vLLM, a distributed LLM serving engine that significantly enhances throughput without compromising model accuracy. This paper was chosen for reproduction due to its novel approach to a pressing problem in LLM deployment, its potential to greatly reduce resource wastage, and its implications for advancing the efficiency of AI services. Through this reproduction, we aim to validate the paper's findings and explore the broader applicability of PagedAttention in real-world LLM serving scenarios.

2 Related works

2.1 Traditional Memory Management in Deep Learning Frameworks

In traditional deep learning systems [2] [3], memory management has been primarily focused on storing tensors in contiguous blocks of memory. This approach is prevalent in frameworks like TensorFlow and PyTorch, where the KV cache for each request is stored as a contiguous tensor. However, this method leads to significant inefficiencies in LLM serving due to unpredictable output lengths and the necessity of maximum sequence length pre-allocation. This results in reserved memory slots, internal and external fragmentation, and overall memory waste. While compaction has been suggested as a potential solution to these fragmentation issues, it is impractical in performance-sensitive environments such as LLM servicing due to the large size of KV caches.

2.2 Paged Memory Systems

The concept of paged memory systems, as implemented in operating systems [4], offers a potential solution to the memory fragmentation problem. By dividing memory into fixed-size blocks or pages, paged memory systems effectively manage fragmentation and allow for more flexible memory allocation. Inspired by this, PagedAttention was developed to manage KV cache memory in a non-contiguous manner, reducing fragmentation and enabling memory sharing across requests. This approach draws parallels to virtual memory management, where blocks are allocated as needed, allowing for a more efficient and flexible use of GPU memory.

2.3 Batching Techniques for Improved Compute Utilization

Batching multiple requests is a well-established method for improving compute utilization in LLM serving. Traditional batching strategies, however, face challenges due to asynchronous request arrivals and variability in input and output lengths. Naive approaches result in significant queuing delays and computational waste due to padding. To overcome these challenges, fine-grained batching mechanisms such as cellular batching and iteration-level scheduling have been proposed. These techniques operate at the iteration level, allowing for dynamic batch composition and minimizing the need for padding. By optimizing the queuing process and computational efficiency, these methods significantly enhance the throughput of LLM serving systems.

2.4 Innovations in Prefill and Decoding Stages

The prefill and decoding stages in LLM inference present unique opportunities for optimization [5]. The prefill stage, which is compute-intensive, can be parallelized to improve efficiency. Conversely, the decoding stage, characterized by its memory-intensive nature, involves iterative KV cache read and write operations. Recognizing the independence of these two stages, pipeline parallelization offers substantial optimization potential. By decoupling and concurrently executing these stages, systems can achieve higher throughput and reduce latency, thus optimizing the overall LLM serving process.

In summary, the related works in memory management and batching techniques provide critical insights into addressing the inefficiencies in traditional LLM serving. Innovations like PagedAttention and fine-grained batching mechanisms offer promising solutions to enhance memory utilization and compute efficiency, paving the way for more effective large language model deployment.

3 Method

3.1 KV Cache Memory Management: PagedAttention

The vLLM framework(Figure 6) introduces a key optimization in the management of the KV cache, addressing inefficiencies inherent in traditional memory allocation methods used in model inference frameworks. Typically, these frameworks pre-allocate fixed-size memory spaces for each input, which leads to inefficiencies due to the variability in prompt and generated text lengths across different requests. To overcome this challenge, vLLM adopts a storage management mechanism based on PagedAttention.

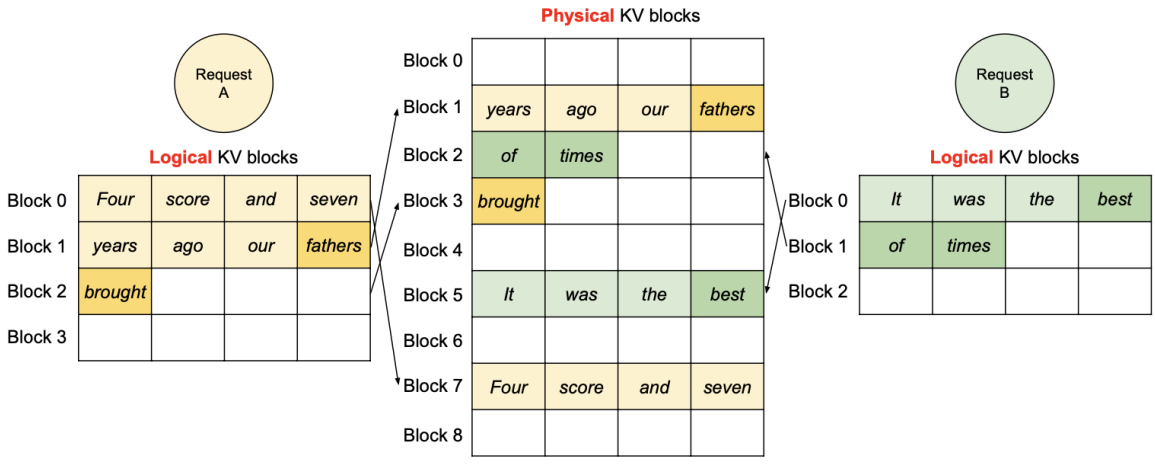

Figure 1. Overview of the KV Cache Memory Framework

PagedAttention is an attention algorithm inspired by virtual memory and paging techniques found in operating systems. It diverges from traditional attention algorithms by allowing KV vectors to be stored in non-contiguous memory spaces, offering several advantages:

- KV Cache Partitioning: PagedAttention divides the KV cache of each sequence into multiple KV blocks, with each block containing a fixed number of tokens' key (K) and value (V) vectors. This partitioning facilitates more granular and efficient memory allocation.
- Non-Contiguous Storage: Unlike traditional methods that require contiguous storage of KV vectors, PagedAttention enables these KV blocks to be stored non-contiguously in physical memory. This design enhances the flexibility of memory management, allowing for better utilization of available memory resources.
- Dynamic Allocation: PagedAttention dynamically allocates KV blocks as needed, reducing memory fragmentation and allowing for memory sharing across different requests or sequences within the same request. By allocating memory on demand, PagedAttention minimizes unused memory space and optimizes overall memory usage.

Through these innovations, PagedAttention significantly enhances the efficiency and flexibility of KV cache management in LLM serving systems, thereby improving throughput and reducing resource wastage. This approach represents a substantial advancement in addressing the memory management challenges associated with large language models.

3.2 Request Scheduling: Method

Beyond optimizing memory management, vLLM enhances inference performance, particularly in handling online processing scenarios. A crucial technique in this optimization is the implementation of advanced batching strategies, which address the dynamic and often unpredictable nature of online requests.

Batching Techniques

- Static Batching: Traditional static batching struggles with varying sequence lengths, leading to decreased GPU utilization as sequences complete at different times. This approach is insufficient for maintaining optimal performance under fluctuating request loads.
- Continuous Batching: To improve upon static batching, vLLM employs continuous batching(Figure 2). This method dynamically recreates batches, ensuring that GPU capacity is optimally utilized after each token generation. By continuously adjusting batches, vLLM maximizes GPU utilization and maintains high throughput, even as sequence lengths and request volumes vary.

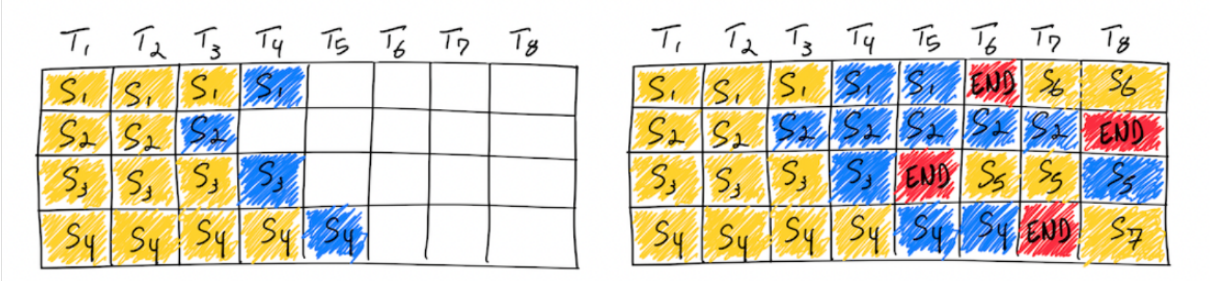

Figure 2. Comparison of Static Batching with Continuous Batching

Preemption Strategies

Handling resource contention efficiently is critical for supporting online processing. vLLM introduces preemption strategies to manage scenarios where a sudden influx of requests demands attention.

- Swapping: In cases of resource preemption, vLLM transfers evicted KV cache blocks to CPU memory. This approach allows the system to pause new requests temporarily, ensuring that preempted sequences can complete without loss of data.
- Recomputation: When preempted sequences are rescheduled, vLLM recomputes the KV cache. This ensures continuity and accuracy in processing, albeit at the cost of additional computation, which is managed to maintain system performance.

These request scheduling techniques ensure that vLLM can efficiently handle high volumes of online requests, maximizing GPU utilization and maintaining robust throughput. By dynamically adjusting to the demands of online inference, vLLM provides a responsive and efficient serving solution for large language models.

4 Implementation details

In this section, I compare the implementation details of the request scheduling mechanisms in vLLM as described in the paper with the actual code I compiled and re-product the experiment results. This comparison aims to highlight the congruencies and discrepancies between the theoretical framework and practical execution.

4.1 Comparing with the released source codes: System Framework and Processing Flow

I begin by introducing the Framework and Processing Flow section, as it constitutes the core of the code architecture(Figure 3). This section lays the foundation for understanding how the system orchestrates and manages the processing of user requests, ensuring efficient resource utilization and high throughput. By delineating the roles and interactions between various components, it provides a comprehensive overview of the system's operational dynamics. In essence, the Framework and Processing Flow section serves as the blueprint for the system's execution strategy, detailing the mechanisms through which it achieves its performance objectives.

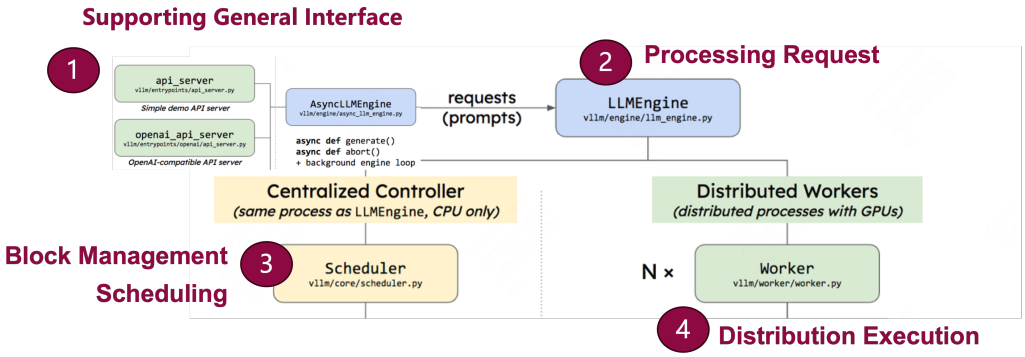

Figure 3. System Framework and Processing Flow

Preemption Strategies

Request Initialization: In the source code, user requests are indeed initiated via standard interfaces, such as the OpenAI API. The code confirms that requests are processed by the asyncLLMEngine, where prompts are encapsulated and dispatched to the LLM Engine, matching the paper's description.

LLM Engine and Scheduler Coordination: The synchronization between the LLM Engine and the Scheduler is evident in the code. The Scheduler's role in managing task execution priority and the LLM Engine's responsibility for triggering these tasks aligns with the paper's assertions. The source code implements this coordination through well-defined classes and functions that reflect these roles. In LLM Engine, the "addrequest" and "step" functions are implemented as described. The code includes the generation of SequenceGroup instances and their addition to the Scheduler's waiting queue through "addrequest". The "step" function is also present, performing prefill operations or decoding iterations.

Scheduler: Examination of the source code reveals the presence of the key queues managed by the Scheduler, namely the waiting, running, and swapped queues. The logic for moving sequence groups between these queues based on resource availability and execution state is clearly implemented, adhering closely to the paper's outlined methodology. During execution, sequence groups are initially placed in the waiting queue and progressively moved to the running queue(Figure 4). If memory shortages or preemption events occur, decisions are made based on the number of sequences within a sequence groups to either recompute or temporarily swap out. Swapped sequence groups are placed in the swapped queue, where they receive the highest priority for resources. If a recomputation decision is made, previous results are discarded, and the sequence groups is returned to the waiting queue. The waiting queue operates on a first-in, first-out basis, with data in the swapped queue having higher priority.

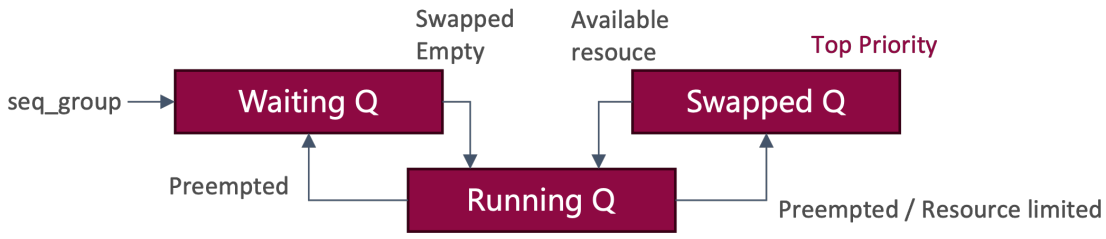

Figure 4. Processing Flow with Queue

- Waiting Queue: Contains data that has not undergone prefill; each sequence groups here has only one sequence (prompt).
- Running Queue: Stores all sequence groups sent for inference in the last stage.
- Swapped Queue: Holds sequence groups that were preempted in previous scheduling phases.

Worker Execution: The source code showcases the worker module as a distributed system, with each worker corresponding to a single GPU. The responsibilities of maintaining the KV cache and executing model computations are accurately reflected in the distributed setup within the codebase. The partitioning of model components across different workers is consistent with the described approach, ensuring scalability and efficiency.

Overall, the released source code closely aligns with the methodologies and processes described in the paper. The practical implementation effectively translates the theoretical concepts into functional code, demonstrating the feasibility and robustness of the proposed request scheduling strategies in real-world applications.

4.2 Comparing with the released source codes: KV Cache Management Implementation

The implementatione(Figure 5) of KV Cache Management in vLLM is centered around efficient block and cache management strategies, ensuring optimal use of available memory resources for inference processes. The system architecture divides responsibilities between a central controller and worker nodes, each playing distinct roles in managing the KV cache.

Block Management

- Logical Block Management: The central controller oversees the logical allocation of KV cache blocks. It handles the high-level organization and management of blocks, maintaining a mapping table that records the relationships between logical and physical blocks.
- Physical Block Management: At the worker node level, physical blocks are managed. This involves the allocation of block IDs to represent actual physical memory segments. The worker nodes are responsible for the tangible allocation and deallocation of these memory blocks, based on demands from the central controller.
- Mapping Table Management: The central controller maintains a mapping table that links logical block IDs to physical block locations. This mapping ensures that memory requests are efficiently translated into physical memory operations, facilitating seamless data retrieval and storage.

Cache Management

- Real Memory Allocation/Swap: Worker nodes manage the actual memory allocation and swapping processes. They allocate memory for KV cache blocks according to the central controller's directives and handle swapping of blocks between CPU and GPU memory as needed.
- Profiling for Memory Allocation: To determine the appropriate amount of memory to allocate for KV cache, vLLM uses a profiling step during model deployment initialization. Profiling involves running simulated experiments to estimate the number of KV cache physical blocks that can be allocated on GPU and CPU. ProfileNumAvailableBlocks: This profiling step, engine calculates the memory available for KV cache by subtracting the memory used for a single forward pass (without KV cache) from the total GPU memory. By simulating a forward pass with fabricated data, the system can measure the memory footprint and adjust the allocation accordingly
- Once the available memory for KV cache is determined through profiling, the total number of physical blocks is calculated.

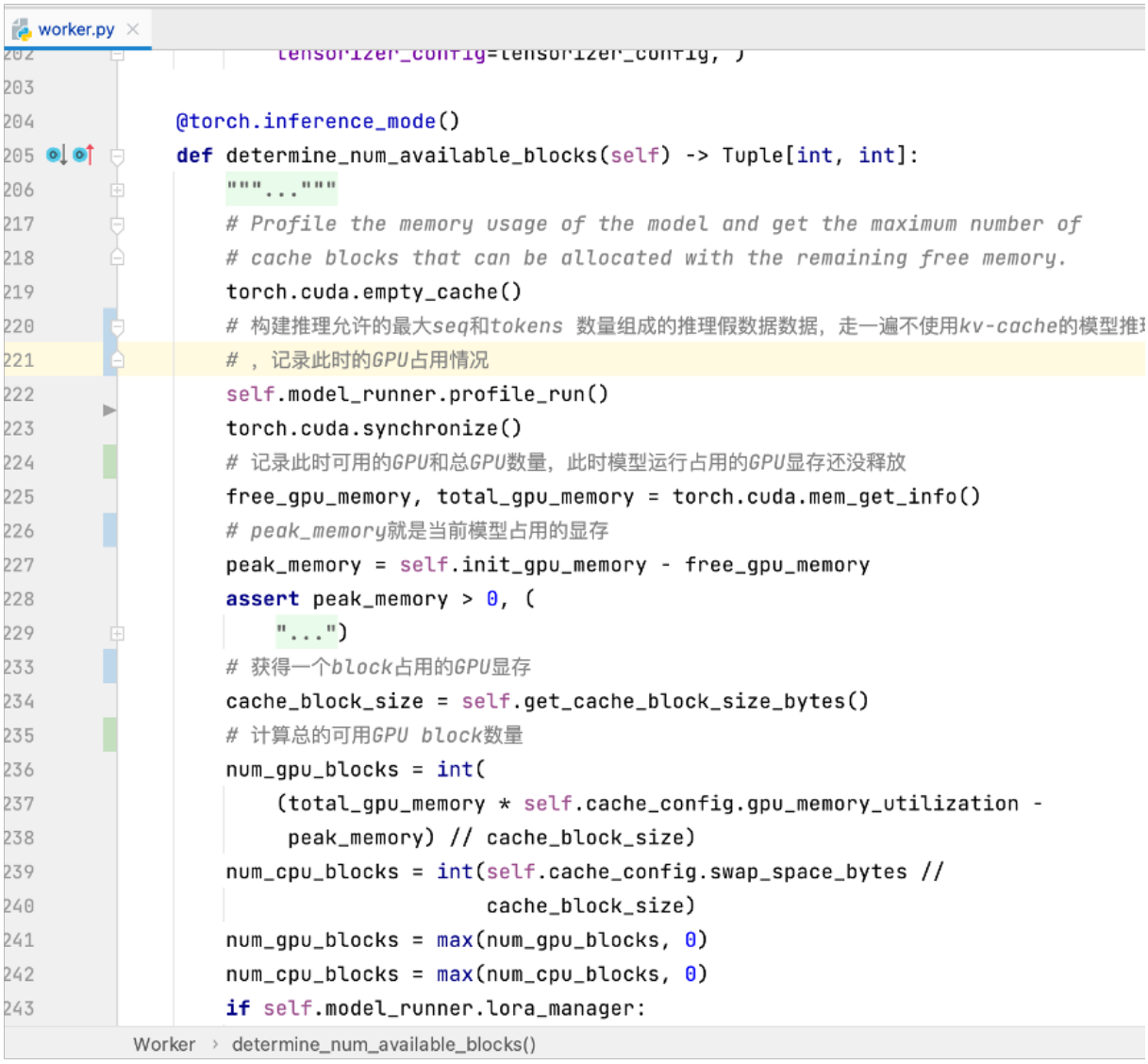

Figure 5. Implementation of Memory Mangement

By implementing these detailed KV Cache Management strategies, vLLM efficiently manages memory resources, optimizing inference performance and resource utilization across different model and context scenarios. This careful balance of logical planning and physical execution forms the backbone of vLLM's high-throughput serving capabilities.

4.3 Experimental environment setup

In this section, we outline the setup of the experimental environment used to reproduce the results of the vLLM framework, detailing the hardware specifications, model selection, and the steps for initiating the LLM inference server and running the benchmark tests.

Hardware Specifications: The experiments were conducted on a high-performance computing setup with the following specifications:

- GPU: Nvidia A100 with 40GB of memory, providing the necessary computational power for efficient model inference.

- CPU: Dual Intel 6430 processors, ensuring robust processing capabilities to handle concurrent tasks and manage data flow efficiently.
- Memory: 64GB of DDR5 RAM, configured with 16 modules running at 4800MHz, supporting fast data access and manipulation during inference operations.

Model Selection: The vLLM framework was evaluated using three distinct models from the Qwen-2.5 series, each varying in size to demonstrate scalability and performance across different model complexities:

- Qwen-2.5-14B
- Qwen-2.5-7B
- Qwen-2.5-3B
- Llama-3.1-8B

4.4 Main contributions

The vLLM framework significantly advances large language model (LLM) inference efficiency and scalability through innovative request scheduling and KV cache management. Its sophisticated scheduling system efficiently processes user requests via a coordinated mechanism involving the LLM Engine, Scheduler, and distributed worker nodes. This system incorporates standardized request initialization and seamless synchronization, employing core functions to navigate tasks across waiting, running, and swapped queues, thereby enhancing throughput and resource allocation. Each distributed worker is linked to a single GPU, facilitating scalable model computation and optimizing performance. In terms of KV cache management, vLLM divides tasks between a central controller and worker nodes: the central controller focuses on logical block management and mapping, while worker nodes oversee physical memory allocation and necessary swapping between GPU and CPU. A profiling process simulates a forward pass to determine optimal memory allocation for KV cache, ensuring effective resource utilization. Overall, vLLM's primary contribution lies in its ability to translate complex theoretical concepts into a robust and scalable practical system, resulting in a high-throughput solution capable of efficient inference processing and optimal resource allocation across diverse models and contexts.

5 Results and analysis

5.1 Steps for Experimentation

Starting the LLM Inference Server: Begin by launching the inference server for the selected model. For instance, to start the server with the LLM model, use the following command. This command initializes the server with logging disabled, sets a maximum model length, and allocates 80

Running the Benchmark: After the server is up, the benchmarking tests are conducted using the benchmark code provided by vLLM. The following command runs the benchmark with a specific dataset and seed. This step evaluates the server's performance and responsiveness, helping to verify the reproducibility of the experimental results.

By following this setup, the experiments replicate the conditions necessary to assess the vLLM framework's performance, ensuring that the results are consistent and reliable.

5.2 Experiment Results

Model	RPS	Num prompts	Median TTFT	Median TPOT	Median ITL	Avg prompt throughput (Token / s)	Avg generation throughput (Token / s)
Qwen-2.5-3B	4	1200	70.42	12.04	10.00	4144.90	486.58
	8	2400	97.59	20.60	13.87	8275.43	972.24
Qwen-2.5-7B	4	1200	144.79	28.55	19.07	4129.82	502.01
	8	2400	519.91	219.41	82.96	7768.34	942.53
Llama 3.1 8B	4	1200	1431.97	111.24	13.87	4089.62	493.21
	8	2400	2017.46	121.18	18.19	7365.31	927.04
Qwen-2.5-14B	exceptiongroup.ExceptionGroup: unhandled errors in a TaskGroup						

Figure 6. Comparison of Static Batching with Continuous Batching

Evaluating the performance of large language models (LLMs) hinges on three critical metrics: Throughput, Time-to-First-Token (TTFT), and Time-Per-Output-Token (TPOT). These metrics are essential for understanding the efficiency and responsiveness of LLM systems.

Throughput (Tokens/s): This metric measures the number of tokens generated by the system per unit of time. It is calculated by dividing the total number of generated tokens by the total inference time. High throughput is vital for handling a large volume of requests efficiently, making it crucial for real-time applications and scenarios with many simultaneous users.

Time-to-First-Token (TTFT, s): TTFT indicates the latency from receiving a request to generating the first token of the response. It is a critical metric for user experience, especially in interactive applications where quick feedback is expected. Lower TTFT results in faster initial responses and a more responsive application.

Time-Per-Output-Token (TPOT, ms): Also known as inter-token latency (ITL), TPOT measures the average time required to generate each token after the first one. This metric reflects the model's speed in generating tokens during inference, with lower TPOT indicating quicker and smoother token generation.

By monitoring and optimizing these metrics—throughput, TTFT, and TPOT—practitioners can make informed decisions regarding model deployment, resource allocation, and system configurations.

6 Conclusion and future work

In this reproduction study, we validated the effectiveness of the "Efficient Memory Management for Large Language Model Serving with PagedAttention" paper, confirming its significant contributions to addressing memory management challenges in LLM serving. Our experiments reaffirmed that PagedAttention, with its innovative use of non-contiguous memory blocks inspired by operating systems' paged memory systems, can effectively mitigate memory fragmentation while enhancing throughput without compromising model accuracy. The vLLM framework, through its advanced request scheduling and KV cache management strategies,

demonstrated impressive improvements in resource utilization and performance efficiency, making it a robust solution for deploying large language models.

For future work, further experimentation is proposed to explore the scalability of PagedAttention across various model sizes and architectures beyond the Qwen-2.5 series and llama series. Investigating the integration of more dynamic memory management techniques could enhance the adaptability of the framework in diverse operational environments. Additionally, assessing the impact of real-world asynchronous request patterns and variability in input-output sizes will provide deeper insights into optimizing batching techniques and preemption strategies. Expanding the study to include energy consumption metrics could also offer valuable perspectives on sustainable AI serving practices. Through these future explorations, we aim to extend the applicability and efficiency of the vLLM framework, contributing to the broader advancement of AI services.

References

[1] 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*, SOSP '23, page 611–626, New York, NY, USA, 2023. Association for Computing Machinery.

[2] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, high-performance ml serving. *ArXiv*, abs/1712.06139, 2017.

[3] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. *PyTorch: an imperative style, high-performance deep learning library*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[4] Minh Pham, Hao Li, Yongke Yuan, Chengcheng Mou, Kandethody Ramachandran, Zichen Xu, and Yicheng Tu. Dynamic memory management in massively parallel systems: a case on gpus. In *Proceedings of the 36th ACM International Conference on Supercomputing*, ICS '22, New York, NY, USA, 2022. Association for Computing Machinery.

[5] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.