CPEN 511 Project Final Report

Jiajun Huang, Tom Wang

***Abstract*—Large language models (LLMs) utilize KV caching to enhance inference efficiency by reducing redundant computation. However, the KV cache size increases with model complexity, request batching, and sequence length, leading to memory fragmentation and reduced GPU utilization. Existing solutions, such as vLLM, leverage block-based memory management to mitigate these issues. In this work, we explore the feasibility of integrating operating system-inspired page fault prefetching and eviction strategies into KV cache management. Through extensive experimentation, we evaluate predictive techniques for swap-in operations and analyze the impact of adaptive watermark tuning on memory efficiency. Our findings suggest that prefetching may not be viable due to deterministic scheduling, but adaptive watermarking presents an opportunity to optimize GPU memory utilization under dynamic workloads. Future research may explore reinforcement learning-based approaches to refine memory management strategies for improved inference performance.**

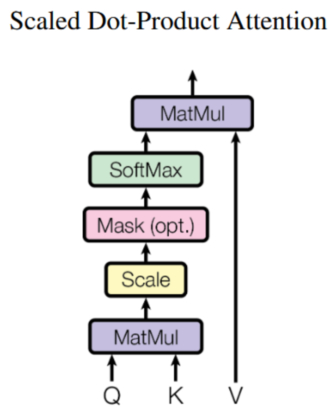

Fig. 1. attention mechanism in transformer model

I. Introduction

KV cache has been known to be a critical component in LLM systems [4, 10, 11]. It serves as a key mechanism for storing reusable KV pairs in transformer-based auto-regressive models. KV caching occurs during multiple token generation steps and only happens in the decoder. With KV cache, the decoder can store the key-value pairs that are generated in the previous token generation steps and then use these cached key-value pairs to generate the next token. This significantly reduces the computation cost of the decoder and improves inference speed [5]

However, KV cache size grows linearly with the size of the language model, number of batched requests, and sequence context lengths. This leads to growing memory requirements, resulting in significant fragmentation of GPU memory and low GPU memory utilization [4]. Memory IO bottlenecks the throughput of the model [10]. On the other hand, `CUDA error: out of memory` is a common error that halts execution when running these models with limited GPU memory [11]. These challenges highlight the importance of efficient KV cache management in high-performance inference systems.

vLLM introduced a novel approach to GPU memory management by organizing memory into blocks and employing a block-based memory management system. This system, which bears a strong resemblance to OS virtual memory management, optimizes memory usage effectively [4]. The vLLM framework has demonstrated significant improvements in GPU memory utilization and reduced fragmentation, making it a promising solution for managing KV caches in transformer-based models.

In this project, we investigate the feasibility of KV cache prefetching and eviction strategies inspired by OS page fault handling mechanisms. Additionally, we seek to improve the scheduling strategies used in vLLM for KV cache memory management, with the goal of achieving more efficient memory management under dynamic workloads.

II. background

A. KV Cache

In transformer models, the attention mechanism relies on three key matrices: Query (Q), Key (K), and Value (V), as illustrated in Figure 1. These matrices are used to compute the attention scores between the query and key, which are then applied to calculate a weighted sum of the value matrix. This process forms the core of the attention mechanism, enabling the model to focus on relevant parts of the input sequence [9].

During the inference of transformer models, the decoder generates tokens one by one (**auto-regressive**). For each iteration, we append the new generated token to the tail of the input for the next iteration. However, without optimization, the model would recompute the Key and Value matrices for all preceding tokens at every step, leading to significant redundant computation. KV cache can store the Key and Value matrices generated in previous steps, allowing the model to reuse them in subsequent iterations. This reduces the computation cost of the decoder and improves the inference speed [5].

B. vLLM

Existing LLM models are observed to *pre-allocate* a contiguous chunk of memory to match the maximum token length of the model. This is inefficient as the memory is not fully utilized. KV cache size grows and shrink dynamically [4].

vLLM proposed **PagedAttention**, which is a new way to manage the GPU memory into blocks and use a block-based memory management system to optimize the memory usage which is very similar to OS Virtual Memory management shown in Figure 2.

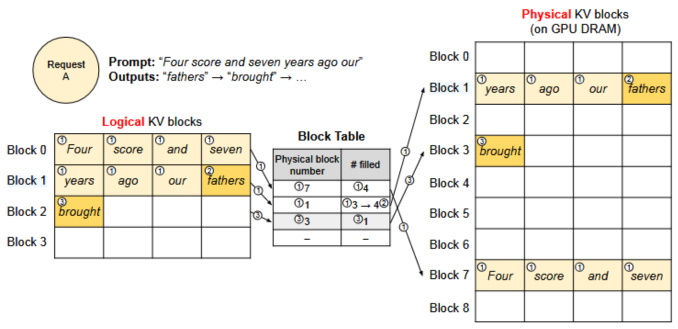

Fig. 2. Example of token storage blocks and their translation to memory blocks

Blocks serve as the fundamental units of memory allocation in vLLM, analogous to pages in OS virtual memory management. Block tables manage these memory blocks similarly to how page tables manage pages in OS virtual memory management. This approach significantly reduces fragmentation and redundant memory usage in GPU memory. [4]

C. OS Page Fault Prefetching and Eviction Techniques

In OS virtual memory management, page fault prefetching and eviction techniques are used to optimize the memory usage. Page fault prefetching is a technique that pre-fetches the pages that are likely to be accessed in the future. Page fault eviction is a technique that evicts the pages that are least likely to be accessed in the future. Numerous literature has investigated ways to optimize GPU memory management through eviction, prefetch and recomputation. There exist all kinds of approaches to optimize cache and page management in OS, including LSTM-based prefetching [3], attention-based prefetching [7], heuristic-based prefetching [6], reinforcement learning-based prefetching [1], ML-based eviction [8] etc.

III. Problem Statement

On edge devices with limited GPU memory, the GPU often cannot hold the entire KV cache. As a result, KV cache blocks must be swapped between GPU and CPU memory, which can significantly increase latency and reduce system throughput. The vLLM framework provides a mechanism to perform this swapping during inference. However, slow PCIe data transfers introduce additional latency. Therefore, improving KV cache swapping is an important task for enhancing performance.

IV. Proposal 1

The initial proposal is to leverage the same methodology as existing OS page prefetching to predict the next Swap-In operation and perform the prefetching operation in advance.

V. Observition

A. vLLM KV cache Management

We explored the basic structure of the codebase, the workflow of vLLM, and the mechanisms for triggering block swapping. We successfully identified all the locations related to KV cache operations.

- **Sequence Allocation:** vLLM allocates a block table for each sequence. The block allocation mechanism is located in the `BlockTable` class `allocate` method.
- **Block Allocation:** For every sequence, vLLM allocates a block for each token. The block allocation mechanism is located in the `SelfAttnBlockSpaceManager` class `append_slots` method.
- **Block Swap-In:** The block can be either in GPU memory or CPU memory. The block swapping mechanism is located in the `SelfAttnBlockSpaceManager` class `swap_in` method.
- **Block Swap-Out:** The block swapping mechanism is located in the `SelfAttnBlockSpaceManager` class `swap_out` method.
- **Block Freeing:** When a sequence is finished, the block table is freed. The block freeing mechanism is located in the `SelfAttnBlockSpaceManager` class `free` method. Inside the `free` method, the block table and all the blocks are freed as well.

Initially, we assumed that KV cache block swapping was managed at the individual block level. However, after investigating the code, we discovered that swapping is handled at the sequence level—each sequence maintains its own KV cache block table, and swapping operations are performed on all KV blocks associated with a sequence.

B. Swap-In pattern predictability

To show the feasibility of our proposal, we first need to show that the Swap-In operation is predictable. Since we found out that the KV cache blocks are not managed individually so we decided to predict the next sequence number to be Swap-In.

1) Experiment Setup: To set up the experiment, we collect the trace using the function mentioned above. By feeding 400 sequences into vLLM, we generated a trace consisting of 3,214 KV cache memory accesses. From this trace, we extracted the Swap-In sequences as training data. To evaluate the predictability of these memory accesses, we employed deep learning models to analyze the trace and predict future accesses.

Our approach involves using a sliding window of previous memory access to predict the next access. The window size is a tunable hyperparameter that affects model performance.

2) Experiment result: We experimented with multiple predictive approaches. Initially, we applied regression models to predict the next memory access block, followed by classification models. Below, we detail our experiments and findings.

For regression-based prediction, we employed both Multi-Layer Perceptron (MLP) and Long Short-Term Memory (LSTM) models. However, MLP significantly outperformed LSTM in this task. While the regression model did not achieve perfect accuracy, its predictions were typically close to the true values, mostly off within ±2. We also examined the impact of window size on the accuracy of the model as demonstrated in Figure 3.

Next, we explored a classification-based approach for prediction. To define discrete classes, we assigned labels by

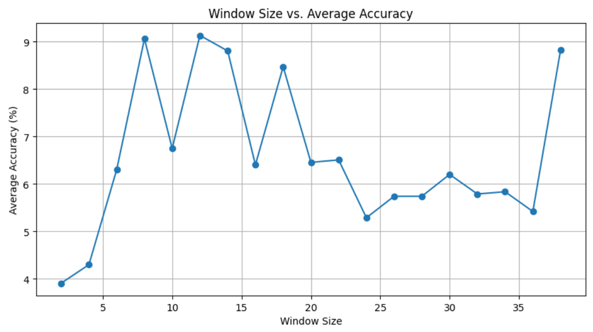

Fig. 3. Regression MLP model prediction accuracy vs. window size.

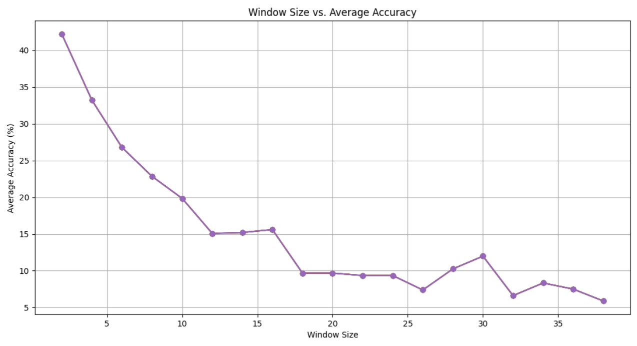

Fig. 4. Classification MLP model prediction accuracy vs. window size.

computing the modulus of each sequence number with a predefined value, effectively converting sequence numbers into categorical classes. However, applying the modulus operation removes the most significant digits of the original values. To address this, we reconstructed the most common (mode) of the significant digits from the batch data.

We found that the classification model outperformed the regression approach in terms of accuracy. Additionally, we investigated how the window size influences the classification model's performance, as demonstrated in Figure 4.

C. Is prefetching suitable for vLLM

As we experiment with the predictability of the Swap-In pattern by implementing the prefetching algorithm in vLLM, we have unfortunately discovered that the Swap-In operation is deterministic. The scheduling algorithm is shown in 1.

1) Overview of scheduling mechanism of vllm: Algorithm 1 shows a simplified version of a scheduling algorithm for a vLLM system. The algorithm takes a key-value cache C and a set of tasks T as input. The algorithm maintains a running queue R, a waiting queue W, and a swap queue S. The algorithm iterates over the waiting queue and schedules tasks to run on the vLLM system. If a task requires additional memory while scheduling computation, the algorithm swaps out the task at the tail of running queue to the swap queue. The algorithm continues to schedule tasks until all tasks are completed.

2) When Does Swap happen?: One thing to note is that Swap-Out only happens when performing an in-order scheduling and an additional block is needed for the target sequence.

Algorithm 1 vLLM Scheduling Algorithm

```
1: procedure SCHEDULING(KV cache $C$, Set of tasks $T$)
2:     $R \leftarrow \emptyset$                                  ▷ Running queue
3:     $S \leftarrow \emptyset$                                  ▷ Swap queue
4:     $W \leftarrow$ sort($T$)                                 ▷ Waiting queue
5:     while $W$ is not empty do
6:         while $C$ is not full and $W$ is not empty do
7:             $t \leftarrow W.deque()$
8:             $R.enque(t)$
9:             $C.allocate(t)$
10:        end while
11:        for $t \in R$ do     ▷ Iterate over running tasks from the beginning
12:            $t.schedule()$
13:            if $t$ need another block for new tokens then
14:                if $C$ is not full then
15:                    $C.append\_block(t)$
16:                else
17:                    $S.enque(R.pop())$  ▷ Swap out the last task in running queue
18:                    $C.append\_block(t)$
19:                end if
20:            end if
21:            if $t$ is finished then
22:                $R.remove(t)$
23:                $C.deallocate(t)$
24:            end if
25:        end for
26:        if no swap out happened then  ▷ There might be no memory pressure
27:            while $C$ is not full and $S$ is not empty do
28:                $t \leftarrow S.deque()$
29:                $R.enque(t)$
30:                $C.allocate(t)$
31:            end while
32:        end if
33:    end while
34: end procedure
```

In this case, FILO makes sense because the tail tasks are usually not really scheduled and the sequence contains few blocks. Thus no fancy eviction algorithm is needed since data dependency is rare and clear.

This data fetching operation is already hidden under computation, where it dispatches all the running tasks and then fetches Swap-Out blocks, which, in a sense, can already be considered as prefetching.

The data fetching mechanism is greatly different from the OS memory access pattern.

VI. Proposal 2

A. Overview

As shown above, prefetching is not feasible for vLLM due to the simplicity of scheduling.

Another way to avoid memory bound and improve performance is to reduce the memory traffic. Given that the memory traffic is only caused by

- Allocate and Free Blocks (not avoidable)
- Swap-in and Swap-out Blocks (aim to avoid)

We want to reduce the number of blocks swapped in and out. The idea is to not fully utilize the KV cache, leaving some space for scheduled blocks to grow given the scheduling policy in Algorithm 1.

B. *vLLM's Watermark*

vLLM is aware that sequences need blocks to grow. Thus, a `watermark` parameter is provided to the user. The watermark is a threshold that indicates how much of the KV cache should be used. The default value is 0.01, meaning that 1% of the KV cache is reserved for scheduled blocks to grow gracefully.

However, we suspect that a fixed threshold is not optimal. The watermark should be adaptive to the situation and workload.

- If the workload is memory bound, then the watermark should be larger, so that more blocks can grow in the KV cache.
- If the workload is compute bound, then the watermark should be smaller, so that more blocks can be swapped in and out.
- If there are many newly allocated blocks, then the watermark should be larger, so that more blocks can grow in the KV cache.
- If most of requests are likely terminating soon, then the watermark should be smaller, so that more blocks can be swapped in and out.

C. *Hypothesis*

We hypothesize that the space to be reserved should be related to the nature of the workload and the condition of the requests already in KV cache.

VII. Experiment

A. *Experiment setup*

Since it is hard to control the workload variant, we first investigate the relationship between the number of requests in the KV cache and watermark. We will use the throughput of the system as a metric.

We investigate 4 LLM models of different sizes:

1) Facebook 125m
2) Facebook 1.3b
3) Facebook 2.7b
4) Deepseek R1 Distilled Qwen 1.5b

Each of them are chosen such that there are 100-500 GPU blocks available for KV cache. We run the 400 prompts on them for 10 times for each data point we want to collect. We take an average of them to account for the random termination nature of LLMs.

We start our experiment with a naive brute force search.

B. *Naive Search*

Given that each block contains 16 tokens and for each inference, 1 more token is appended to the block, if we assume that the probability of different number of tokens are in the last block is uniformly distributed, then we can calculate the probability of the last block being full of each request: [4]

$$P_{full} = \frac{1}{16}$$

Let there be N requests in the KV cache, then the probability of the last block of N requests being full is:

$$P_{full} = \frac{1}{16} \cdot N$$

which is the expected number of free blocks we need to reserve for each inference.

However, it is not likely that only one block is needed for each request to grow. The number of blocks needed for each request to grow is likely to be larger than 1. It is not possible to quantify it, thus we will use a naive search to find the optimal watermark.

From exploration, we find that the average block length for each of the requests is around 7.5. Thus we search around this range.

Let $a \in [0, 12]$ be the **opt_factor** of the watermark, thus we reserve $\frac{aN}{16}$ blocks when scheduling. For practical reasons, we will only search for $a \in [0, 12]$ with step size of 0.2. To account for the variance, we will run each experiment 10 times and take the average. The watermark is set to be 0.01 by default, and we will use this as the baseline.

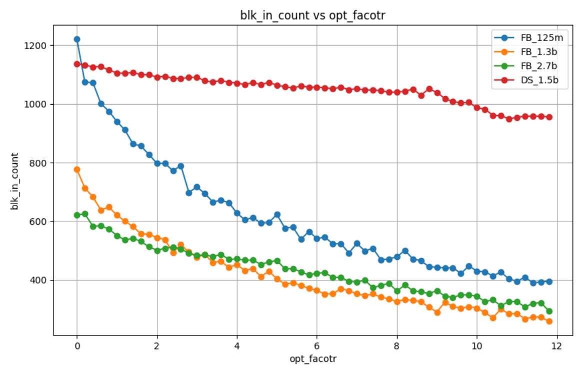

Fig. 5. Block Swapped Count vs. Opt Factor

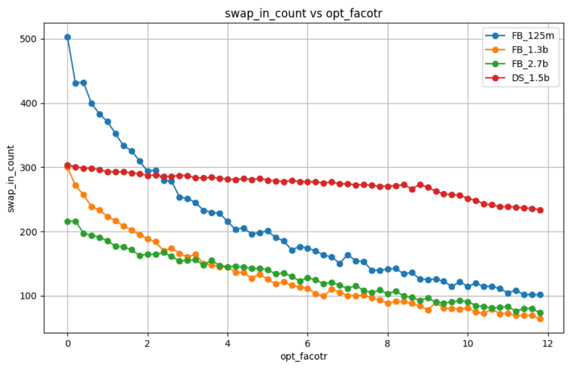

Fig. 6. Swap Operation Count vs. Opt Factor

1) Naive Search Results: First thing that we observe is from Figure 5 and 6, the number of blocks swapped and swap operations are decreasing with the opt factor. This is expected, as the larger the opt factor, the more blocks are reserved for each request to grow, which will reduce the number of blocks and swap operations.

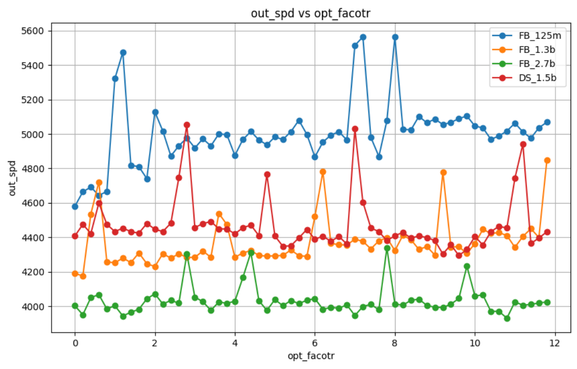

Fig. 7. Output Throughput vs. Opt Factor

At the same time, the output throughput in Figure 7 is behaving chaotically. There is not a clear trend as the opt factor increases.

- The smaller models FB_125m and FB_1.3b showed a slight trend of improvement, but the improvement is not significant.
- Deepseek model showed a slight trend of underperformance, but the difference is not significant. This might be due to the fact that the model is well optimized as their technical report suggested. [2]
- There are a few high peaks in all models, it might suggest that a sweet spot exists, but it is hard to find.

The results are as expected. For larger models, compute bound is more likely to happen, which is why the throughput for FB_2.7b did not show a significant improvement. For smaller models, the throughput is more likely to be memory bound, which is why the throughput for FB_125m and FB_1.3b showed a slight improvement. Deepseek model is known to be well optimized, so it is not surprising that the throughput decreased slightly.

The peaks in the throughput suggest that it is possible to find a sweet spot for the watermark, but it is likely not linear, which means we need to search for the optimal watermark in a more sophisticated way.

C. Naive Search 2

Another way to search for the optimal watermark is to follow the average block length. Let there be b number of blocks in the KV cache, and N number of requests in the KV cache. The average block length is:

$$\text{avg\_blk\_len} = \frac{b}{N}$$

We can use this as a heuristic to find the optimal watermark. The idea is to set a threshold t such that we stop scheduling when the average block length $\frac{b}{N}$ is larger than t. We do a naive search for t in the range of $[1.8, 7.5]$ with step size of 0.2. We will use the same setup as the previous experiment. The expected t should be slightly smaller than 7.5, as we expect the average block length to be around 7.5 and the average prompt length is 2 blocks.

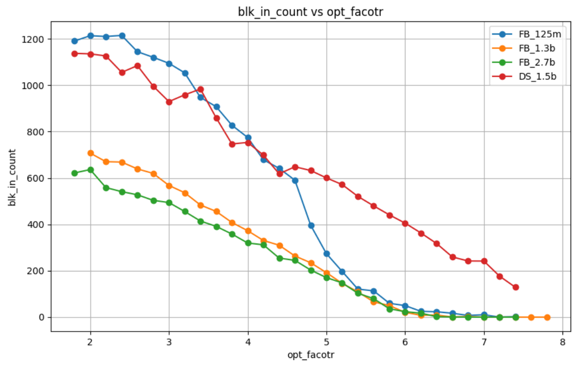

Fig. 8. Block Swapped Count vs. Opt Factor

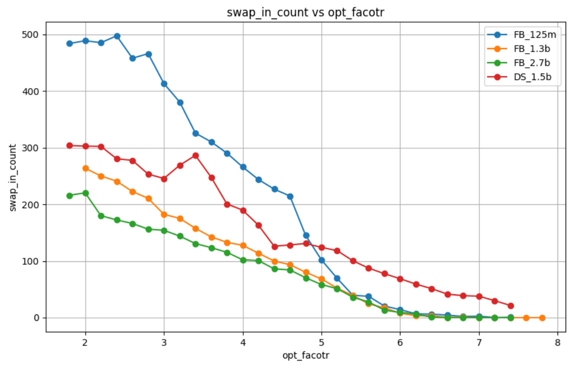

Fig. 9. Swap Operation Count vs. Opt Factor

1) Naive Search 2 Results: Figure 8 and 9 show the number of blocks swapped and swap operations are decreasing with the opt factor. The block number and swap operations are approaching zero as the opt factor approaches 7.5 which is as expected. However, we do see that Deepseek model is not as stable as the rest of the models. There are multiple sudden rises in the number of blocks swapped and swap operations. This shows its different characteristics from the other models.

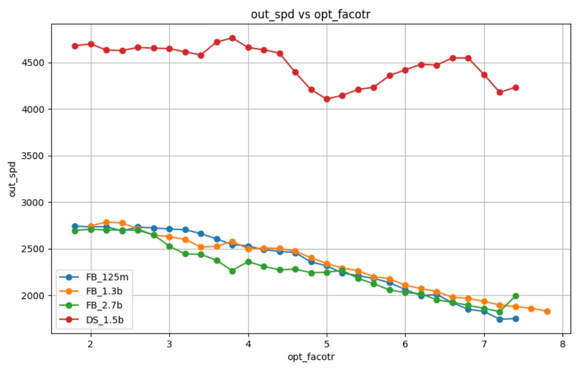

Fig. 10. Output Throughput vs. Opt Factor

Figure 10 shows the output throughput vs. opt factor. For all the facebook models, there is no significant rise and the general trend is decreasing. This is not as expected, as we expect the throughput to be increasing initially because of the memory bound and then decreasing due to frequent scheduling overhead. On the other hand, Deepseek model shows a significant rise when opt factor is around 3.6 and another sudden rise at opt factor 5. Again surprising and weird.

D. Summarize Observations

The experiments reveal several key observations:

- **Block Swapping and Swap Operations:** Both experiments show a consistent decrease in the number of blocks swapped and swap operations as the opt factor increases. This aligns with the expectation that reserving more space reduces memory traffic.
- **Output Throughput:** The output throughput exhibits inconsistent trends:
  - Smaller models (FB_125m, FB_1.6b) show slight improvements in throughput, indicating memory-bound workloads benefit from increased reserved space.
  - Larger models (FB_2.7b) and well-optimized models (Deepseek) show minimal or unexpected changes, suggesting compute-bound workloads or inherent optimizations dominate performance.
  - Peaks in throughput suggest potential sweet spots for the watermark, but these are not linear or easily predictable.
- **Deepseek Model Behavior:** The Deepseek model demonstrates unique characteristics, with sudden rises in block swapping and throughput at specific opt factors. This indicates its behavior differs significantly from other models.
- **General Trends:** While block swapping and swap operations decrease predictably, throughput trends are less clear, highlighting the complexity of finding an optimal watermark.

These observations suggest that while naive search methods provide insights, more sophisticated approaches are needed to identify optimal watermark values effectively.

VIII. Conclusion

In this report, we have explored the possible ways to improve the memory bound in vLLM. Although the initial proposal to leverage OS prefetching techniques was not successful, we have identified a potential solution to improve the performance of vLLM by adjusting the watermark in the KV cache. By doing brute-force search, we have found that it is possible to improve the performance of vLLM by adjusting the watermark in the KV cache. The results show that the throughput can be improved only if a proper watermark is set for the current workload and the current state of the KV cache.

Further work is needed to generalize the watermark adjustment to different workloads and different states of the KV cache.

IX. Future Work

From the two simple heuristic search, we can see that the watermark is a very important parameter in vLLM. The naive search shows that the watermark can be adjusted to improve the performance of vLLM. However, the naive search is not a general solution and it is not guaranteed to work for all workloads. We need to find a more general solution to adjust the watermark. We can use machine learning techniques to learn the optimal watermark for different workloads and different states of the KV cache. By training a model on a large dataset of workloads and their corresponding optimal watermarks, we can potentially find a more general solution that works for a wider range of workloads.

Reinforcement learning (RL) is a powerful tool for optimizing complex systems. In the context of vLLM, RL can be used to dynamically adjust the watermark based on the current workload and the state of the KV cache. By training an RL agent to learn the optimal watermark for different workloads and states, we can potentially improve the performance of vLLM even further.

We have explored the possibility of using RL to adjust the watermark in vLLM. The idea map is to use:

- **State**: The current state of the KV cache, including the number of requests in the KV cache, the number of blocks swapped in and out, and the number of tokens in each block.
- **Action**: The action taken by the RL agent, which is to adjust the watermark in the KV cache.
- **Reward**: The reward given to the RL agent, which is the throughput of the system after adjusting the watermark.
- **Policy**: The policy learned by the RL agent, which is to adjust the watermark based on the current state of the KV cache.

The difficulty of this approach is that the state space is that

- The action space is large, as there are many possible values for the watermark.

- The reward function is not easy to define, as the throughput of the system is not a simple function of the watermark. This is also random since LLM inference is not deterministic.

At this stage of time, we ran out of time to implement this approach. However, we believe that this is a promising direction for future work. We can use RL to learn the optimal watermark for different workloads and different states of the KV cache. By training an RL agent to learn the optimal watermark, we can potentially improve the performance of vLLM even further.

References

[1] Rahul Bera et al. "Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning". en. In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. Virtual Event Greece: ACM, Oct. 2021, pp. 1121–1137. ISBN: 978-1-4503-8557-2. DOI: 10.1145/3466752.3480114. URL: https://dl.acm.org/doi/10.1145/3466752.3480114 (visited on 01/23/2025).

[2] DeepSeek-AI et al. *DeepSeek-V3 Technical Report*. 2025. arXiv: 2412.19437 [cs.CL]. URL: https://arxiv.org/abs/2412.19437.

[3] Milad Hashemi et al. *Learning Memory Access Patterns*. arXiv:1803.02329. Mar. 2018. URL: http://arxiv.org/abs/1803.02329 (visited on 10/20/2024).

[4] Woosuk Kwon et al. *Efficient Memory Management for Large Language Model Serving with PagedAttention*. arXiv:2309.06180 [cs]. Sept. 2023. DOI: 10.48550/arXiv.2309.06180. URL: http://arxiv.org/abs/2309.06180 (visited on 01/20/2025).

[5] João Lages. *KV Caching Explained*. Accessed: 2025-01-25. 2023. URL: https://medium.com/@joaolages/kv-caching-explained-276520203249.

[6] Hasan Al Maruf and Mosharaf Chowdhury. "Effectively Prefetching Remote Memory with Leap". en. In: ().

[7] Zhan Shi et al. "A hierarchical neural model of data prefetching". en. In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Virtual USA: ACM, Apr. 2021, pp. 861–873. ISBN: 978-1-4503-8317-2. DOI: 10.1145/3445814.3446752. URL: https://dl.acm.org/doi/10.1145/3445814.3446752 (visited on 01/31/2025).

[8] Zhenyu Song et al. "Learning Relaxed Belady for Content Distribution Network Caching". In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 529–544. ISBN: 978-1-939133-13-7. URL: https://www.usenix.org/conference/nsdi20/presentation/song.

[9] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762.

[10] Ahmet Caner Yüzügüler, Jiawei Zhuang, and Lukas Cavigelli. *PRESERVE: Prefetching Model Weights and KV-Cache in Distributed LLM Serving*. arXiv:2501.08192 [cs] version: 1. Jan. 2025. DOI: 10.48550/arXiv.2501.08192. URL: http://arxiv.org/abs/2501.08192 (visited on 01/18/2025).

[11] Youpeng Zhao, Di Wu, and Jun Wang. *ALISA: Accelerating Large Language Model Inference via Sparsity-Aware KV Caching*. arXiv:2403.17312 [cs]. Mar. 2024. DOI: 10.48550/arXiv.2403.17312. URL: http://arxiv.org/abs/2403.17312 (visited on 01/25/2025).