Improving Large Language Model Throughput with Efficient Long-Term Memory Management

Nikhil Jha
UC Berkeley
nikhiljha@berkeley.edu

Kevin Wang
UC Berkeley
kevwang@berkeley.edu

***Abstract*— Delivering low latency and high throughput when serving large language models (LLMs) requires the intelligent batching and caching of inputs and intermediate states. Existing systems optimize inference by storing the key-value cache (KV cache) of an LLM inference pass over the duration of one request or multiple parallel requests. However, at longer time intervals, GPU time is currently required to recompute the KV cache between requests. To address this problem, we propose a system built on top of PagedAttention which saves the KV cache over multiple requests. Our contributions include (1) the storage of the KV cache of the prefixes of prompts (prefix cache), (2) the ability for the prefix cache to be swapped between GPU memory, CPU memory, and disk, (3) scheduling policies that optimize the location of the prefix cache, and (4) benchmarks that test common workloads with prompt prefixes such as chat and text analysis. Our evaluations show that our system improves throughput by 2x vLLM.**

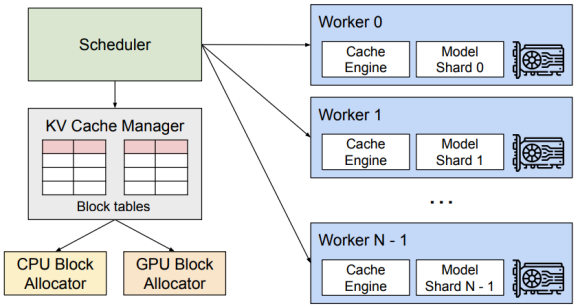
I. Introduction

Large language models such as GPT[1] and Llama[2] have shown an impressive range of abilities and use-cases, and have already had a significant impact over people's lives and work. Their popularization has led to their deployment across many applications, and many companies are working to host and use these models. However, running an LLM, especially at large scales, is uniquely expensive due to their parameter counts. High-powered GPUs are required to make use of the best models, along with supporting infrastructure for computation, storage, and networking. Therefore, possible increases in LLM throughput would be very beneficial to these systems.

Almost all LLMs are based on the Transformer model[3], specifically the Decoder-only variant. These models receive an input text (prompt) as a list of tokens, which are groups of characters or bytes mapped to vectors, and computes an output text sequentially as tokens as well. This computation relies in values derived from prior tokens (KV cache), and thus caching these values would eliminate the need for recomputation.

Systems such as PagedAttention[4] have been developed to effectively utilize the KV cache, focusing on increasing the efficiency within a single request. The focus of this paper is different: to improve the efficiency, specifically the throughput, of serving multiple similar requests over time, requests that have a common prefix. This is a very common use-case for an LLM, such as in chat interfaces where requests typically share the same system prompt and occasionally share the same conversation history, or in data analytics tasks, where the input text typically contains a fixed prompt followed by an item from the dataset.

In this paper, we build on top of vLLM, the reference implementation of PagedAttention, to create prefix caching. We do this in order to take advantage of the block-level memory management, existing swapping and scheduling capabilities, and wide support for popular LLMs already present in vLLM.



vLLM System Architecture (PagedAttention)

In total, our contributions are the following:

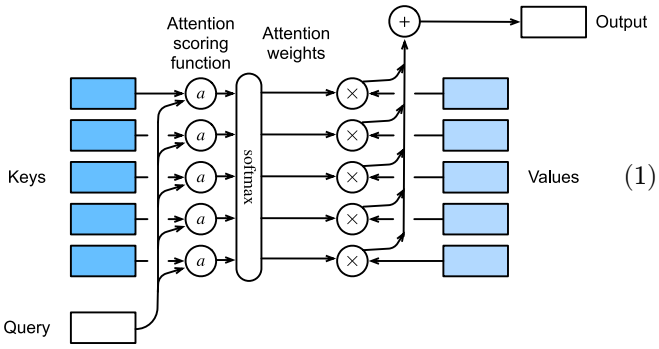
- We identify the inefficiencies in current LLM memory management techniques and quanity their impact on serving performance.
- We propose prefix caching, the storage of KV caches for common prefixes over longer spans of time.
- We develop scheduling algorithms for swapping prefix caches between GPU memory, CPU memory, and disk.
- We design and implement prefix caching on vLLM.

- We create benchmarks that properly evaluate the efficiency of LLMs on chat and data analytics workloads.
- We evaluate vLLM with prefix caching on these benchmarks and demonstrate that it improves throughput over state-of-the-art.

II. Background

In this section we describe the concepts which are foundational to prefix caching: the KV cache and PagedAttention.

A. KV Cache



(1)

The core mechanism of a Transformer model is the "self-attention" block, of which a model typically has many that are computed in parallel (multi-head attention) as well as many multi-head attention layers that are stacked on top of each other. The output of an attention block for each token is typically computed as softmax attention, where query (Q), key (K), and value (V) vectors are computed from the current and prior tokens.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V \tag{2}$$

Whereas the query is computed from the current token, crucially, the keys and values are derived from each token and do not change for subsequent tokens. A naive implementation would recompute them when generating each new token, thus unnecessarily increasing the amount of computation that is required by the GPU for every attention block.

A KV cache however, attempts to store the computed keys and values for future use, and thus for each subsequent token, only one set of keys and values would need to be computed. The rest of the computation will just be a dot product between the query and the stored keys, and then the scaling of each value with the softmax of that dot product.

Another important thing to note is that the KV cache of a token is position-dependent due to the model's positional encoding. This means that tokens that are repeated in a text cannot share the same KV cache.

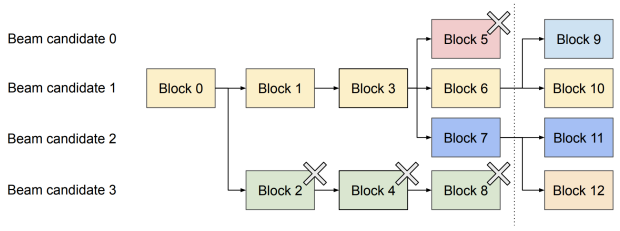
B. PagedAttention and vLLM

While the concept of caching reused results from an attention block instead of recomputing them is simple, but in practice, implementation faces several nontrivial challenges. Since the KV cache grows to the number and size of requests, it becomes difficult to store them all in the limited space available in GPU VRAM, which already also houses the model weights and other computations. Memory management techniques are necessary in order to either evict caches or swap them to another part of the system.

In addition, depending on the decoding algorithm done during generation, the KV cache may be utilized differently, causing additional complexity. For example, different beans in beam search may share varying amounts of the same prefix, and determining what can be shared and where to access the shared KV cache, as well as a proper scheduling algorithm that prevents race conditions, is needed in order to make use of the KV cache well.

PagedAttention[4] attempts to solve these problems by dividing the KV cache into blocks, which contain the keys and values for some fixed number of tokens. These blocks act similarly to pages in the virtual memory of a process, and can be swapped between GPU and CPU memory, with a set number of blocks that live in either regions.

PagedAttention is implemented in vLLM, which also handles request scheduling and batching. This allows vLLM to preemptively swap cache blocks in and out of GPU memory in response to the needs of the requests and decoding algorithms. In addition, vLLM also implements Python interfaces and web APIs for interacting with KV cached LLMs, with support for a large number of popular self-hostable models.



Beam Search example (PagedAttention)

III. Memory Inefficiencies in PagedAttention

PagedAttention effectively solves one of the largest areas of inefficiency in LLM inference, and has achieved state-of-the-art in this area. However, one limitation of this algorithm is that it only considers the usage of the KV cache in a short time span, that of a single request, or at most, multiple requests that share prefixes in the same batch.

A. Text Analysis Workloads

However, in many applications of LLMs, the prefix of requests may be shared in longer time spans, from a few seconds to even hours or days. For example, one common task given to LLMs is text analysis. Here is an example prompt one may use for this task.

> Classify the sentence as positive or negative.
> Think step-by-step, and provide reasoning why the text is identified as positive or negative.
> For example,
> Sentence: 'Same S##t, different day'
> Sentiment: negative
> and
> Sentence: 'This product has changed my life, I am so much more productive now'
> Sentiment: positive
> Sentence: {{sentence}}
> Sentiment:

Observe that a large amount of prompt text does not change between requests, and that this part is at the start of the prompt, and thus its keys and values are also the same between requests. In text analysis situations, the dataset is typically too large to process in a single batch, so it is streamed to the LLM in smaller batches. As a result, vLLM will waste GPU operations each batch on recomputing the prompt template's KV cache.

B. Chat Workloads

Another common task for an LLM is chat. In this workload, the LLM and the user take turns generating and sending text to each other. Upon each request, the LLM uses a system prompt that is shared between all chats as well the entire history of the current chat in order to provide a contextful response. This creates an opportunity to reuse the KV cache between requests in a chat.

By the nature of chat, requests in a single conversation are received synchronously and disjointly, and thus vLLM is again incapable of reusing the KV cache. In addition, the time between requests is unbounded. The user may choose to respond immediately, or come back to the chat after any amount of time. Any system that attempts to share a KV cache between requests in a chat workflow will need to have the capability of swapping the cache to a more persistent storage, as well as employ a scheduler that is able to reason with larger sets of data at longer time horizons, both of which vLLM lacks.

C. Latency versus Throughput

One disadvantage of storing the KV cache in a more persistent medium is latency. Swapping cache from disk to GPU memory is slow, and is likely orders of magnitude slower than recomputing the cache. This will result in a higher latency per request.

However in a GPU-constrained system, this would reduce the amount of computation required by the GPU per request, and thus increase the throughput. This is advantageous in a text analysis workload, where speed is measured in the amount of time it takes to process the entire dataset, as well as a chat workload, where a single GPU could be capable of serving even more requests.

Additionally, higher throughput means fewer queued requests at any moment, which could possibly result in lower latency as a request could be queued earlier. Moreover, the one-time cost of reading from disk is small compared to the entire LLM inference run required in a request, so in all, the latency impact should be minor.

IV. Methods

In this section, we describe the concepts of prefix caching and scheduling. In the subsequent section, we describe the specifics of implementing these concepts in vLLM.

A. Prefix Cache

A prefix cache is the KV cache for a contiguous set of input tokens starting from the first token, and it is uniquely defined by that set of tokens.

As the keys and values depend on their position as well as the previous tokens, it is not possible to generate a reusable cache for a non-prefix.

Prefix caches, just like PagedAttention KV caches, are stored in cache blocks of fixed size. The difference is that prefix caches do not follow the same scheduling pattern as regular KV cache. KV cache blocks can be reallocated when there are no actively running requests that use them, whereas prefix caches may persist past an initial usage. Its lifecycle, instead, is dictated by a scheduler specific for prefixes.

B. Prefix Scheduling

The prefix scheduler swaps prefix caches between three tiers of storage: GPU memory, CPU memory, and disk. Additionally, it also evicts prefixes from the cache when needed. Higher tiers generally swap incrementally down to lower tiers (GPU to CPU, CPU to disk), but when swapping up, prefixes are always swapped directly to GPU memory, in order to use the prefix in an upcoming request.

The prefix scheduler is generalized so that any scheduling algorithm may be used to schedule prefixes. Currently, the Least Recently Used, First-in-first-out, and randomized algorithms are implemented for the scheduler.

V. Implementation

We implement a number of new features in vLLM.

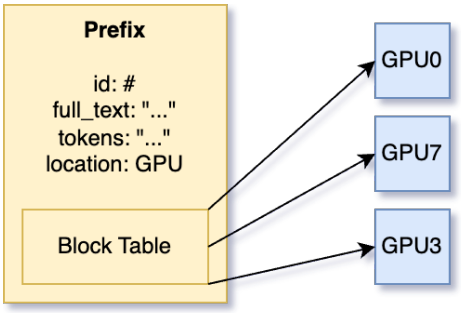
1. Swapping prefixes from GPU to CPU
2. Swapping prefixes between CPU and Disk
3. New scheduling policies to prioritize prefixes that are already on the GPU or CPU (previously only FCFS was supported)
4. New swapping policies (LRU, FIFO, Random)
5. A prefetching strategy for swapping disk to CPU and CPU to GPU ahead of time

A. Existing Implementation Bugs

Previously, vLLM could only persist caches on the GPU memory, and never evict them. This was designed for workloads where there is exactly one prompt, and is still a work-in-progress feature which has a couple bugs.

For example, the vLLM prefix kernel can only handle prefixes of a certain block size. For the purposes of this paper, we don't implement any methods to retrieve prefix caches smaller than the size of a single block. However, the API surface assumed that the user would know the block size and would crash if you specified a prefix size not an even multiple of the block size. We implemented logic to transparently truncate this.

B. Prefix Storage



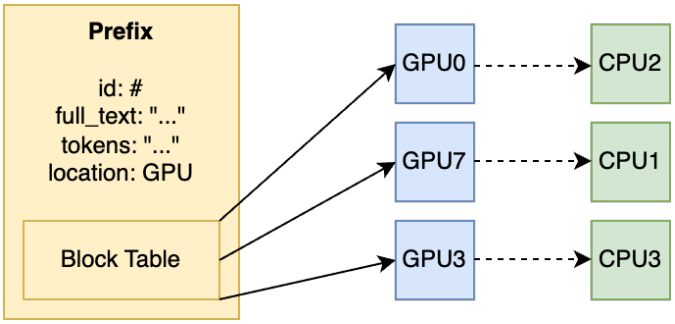
(3)

When a request comes in, we calculate the size of the prefix based on the particular application it is for (e.x. for analytics, the user knows what the prefix is). We then make a request to vLLM's block allocator to get a list of blocks that are on the same hardware device. Prefixes are currently always initialized on the GPU, since they need to be filled on the GPU and are useless until the Attention computation has been done at least once with the prefix. In the diagram above, the blue blocks represent segments of GPU data memory. They do not need to be contiguous, and will be loaded from in the correct order when executing an attention computation involving them.

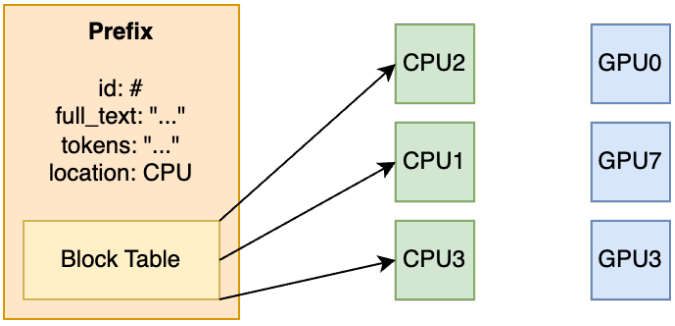
C. Swapping Implementation

To implement CPU swapping, we stored a centralized limit of the number of prefixes that should be allowed on each device: CPU and GPU. vLLM already partitions its memory space into blocks as part of its PagedAttention optimizations, so we could also have limited this by the number of blocks. We chose to limit it by number of prefixes so that comparisons across language models got apples-to-apples comparisons in terms of swapping behavior, as larger language models will have larger block sizes, and so counting the number of prefixes that get swapped is a clearer signal of swapping behavior.

When the scheduler wants to issue a swap, it writes down the swap it wants (a mapping of block IDs on the CPU to block IDs on the GPU, and vice versa for swapping out), and sends it to the GPU worker. The GPU worker issues an asynchronous copy via `cudaMemcpyAsync()` between the CPU and the GPU. This creates a CUDA event, which is saved for later by the CUDA runtime. The attention computation checks to see if there is an outstanding event, which means that the memcopy has not finished, and stalls until it is finished. Achieving good performance with this method therefore relies on scheduling and swapping in a manner that minimizes the total amount of time the system is in this stalled state.



The scheduler then calls the vLLM Block Manager to free the blocks that were swapped from, which does not require any operation other than to account that they are free and may be reallocated to future uses. The block manager is also responsible for acquiring the blocks to swap to when something needs to be evicted. At first we implemented swapping by retrieving from the GPU just in time for the request to be executed, but this ended up being too slow. See the prefetching and scheduling sections for the optimizations we applied to fix this.



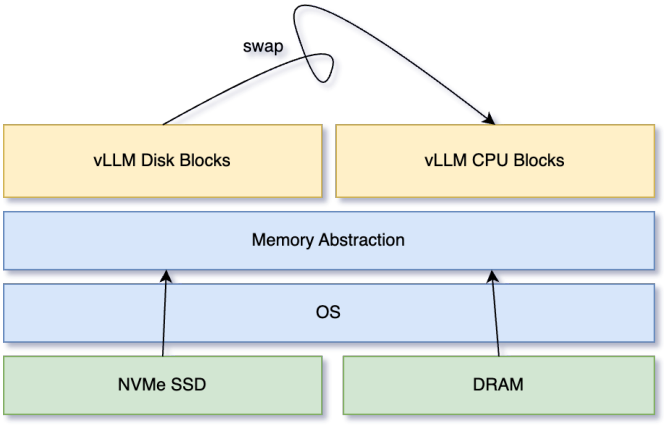
D. CPU Swapping Details

We realized later that counting prefixes instead of blocks was a mistake, because when it comes time to swap in a

prefix, both the size of the prefix and the number of prefix counts need to be checked, and so it's possible that some number of prefixes less than the maximum amount specified. It also requires maintaining an entirely separate state table from the existing table in vLLM. In the final version of this system that makes it into vLLM we will likely rewrite this logic to limit the number of prefixes based on the number of available blocks.

Another consideration was that when two requests need to be swapped in, two separate memory requests may be issued. We solved this by adding a lock onto each prefix that is cleared once a swap is complete.

E. Disk Swapping



To implement disk swapping, we extended the vLLM block manager, the vLLM GPU worker, the vLLM cache engine, and the vLLM CUDA kernel to support blocks on disk. One complication is that the KV cache blocks are in torch tensors, which torch requires that they live in memory. We could have serialized out the torch tensors and tried to load them back in, but we wanted to avoid any unneccesary overhead from the filesystem. We eventually settled on using `mmap` to memory map some blocks from disk into the CPU's address range.

By using `mmap`, as far as torch was aware, these KV caches were actually on CPU memory (just very slow). This significantly simplified the amount of work needed to be done, as a lot of the code from the CPU cache block management could be reused.

Another complication is that although CUDA is perfectly happy to copy memory from the host to the host (i.e. from CPU memory to Disk), it needs the memory to be pinned for it to trust that it can do it asynchronously while a kernel is running. This led to a slowdown in performance until we decided to pull in a separate C runtime to do the swapping entirely separate from CUDA. This increased the complexity of the swapping algorithm considerably, and we are hoping that there exists a workaround in CUDA to avoid this.

F. Prefix Scheduling

The existing scheduling method in vLLM is first-come-first-serve (FCFS). However, if we have cached prefixes, it makes more sense to prioritize prefixes that are on the GPU if any exist, and prioritize prefixes that are on the CPU before going to the GPU. As a result, we created two new scheduling algorithms, fastest prefix first (FPF) and Relaxed-FPF.

1) *Fastest-Prefix-First (FPF):*

This is an unfair scheduling algorithm that tries to fill the GPU with requests that match prefixes already on the GPU at all costs, no matter how much later those requests came. Although this leads to starvation in cases of high utilization of the vLLM server, it guarantees maximum throughput, because if there is something that can be executed fastest without waiting for swaps, it will get executed.

2) *Relaxed-Fastest-Prefix-First (Relaxed-FPF):*

This is a more fair scheduling algorithm that is more akin to a weighted FCFS. It computes a priority for each request, where N is the distance that the prefix is away from the GPU. For example, the GPU is 0, the CPU is 1, the Disk is 2, and the prefix not being computed yet is 3.

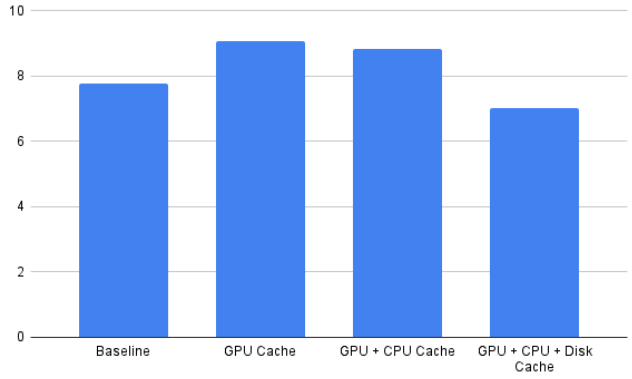
$$(\text{current\_time} - \text{arrival\_time}) * (4 - N) \tag{4}$$

G. Prefetching

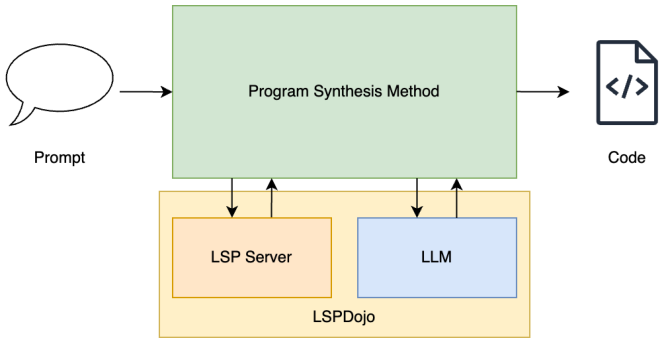
We also implement a prefetcher for the blocks to make sure that the caches on each device are loaded with useful caches. In the case where there are many more requests in the queue than capacity to serve them, the prefetcher looks ahead at the next N requests (where N is based on system capacity, in terms of CPU memory, GPU memory, GPU capability, and Disk space). If there are prefixes currently in the cache that are not used, and upcoming requests that need those prefixes, the prefetcher evicts the unneeded prefixes and immediately tries to swap in the prefixes that are about to be used.

Before we implemented the prefetcher, the performance of the system got worse the more swapping we did past the baseline of only storing the KV caches in the GPU. This leads us to believe that the swapping system (FIFO, LRU, etc.) is significantly less important than the prefetcher having enough information to swap prefixes in as early as possible.

Below is a chart of LLaMA-7B running on an RTX 3090 with 16GB of RAM. The performance degrades even below baseline when disk swapping is added. The cache eviction policy being used is LRU, and FCFS scheduling is being used.



VI. Case-Study: LSPDojo



We built LSPDojo, a framework for language models to take advantage of powerful static analysis tools by interacting with language servers, therefore allowing them to generate code with cor- rect syntax and typing and free of hallucinated functions. While current methods for large language model (LLM) program synthesis largely involve calling for the model to predict the next token, given only the prompt and the previously generated tokens as information. Techniques such as beam search are also used in some cases in order to traverse more of the search space, but they still rely entirely on the model's ability to determine how to continue a sequence without any external information, something that humans will even have a hard time doing.

LSPDojo is implemented as a client to vLLM server that uses static analysis features on a developer's local machine (i.e. IDE autocomplete, diagnostics, etc.) to constrain the (possibly incorrect) outputs of the language model into something definitely correct. Many of these techniques involve repeating a prefix many times to the same language server, which can be accelerated via this K/V cache optimization.

A. Tree Search Method

In the *tree search* method, the LLM generates N solutions. If the language server detects no errors, the solution is submitted. Otherwise, for each error, the code is cut off at the line of the error and the incorrect line as well as the error is inserted as a comment. Then, the LLM is asked to provide N solutions that complete the cut-off solution. This results a breadth-first search across all the errors in the program.

```
1:  function TREE-SEARCH-PSEUDOCODE(prompt, N)
2:      q ← {prompt}
3:      while |q| > 0 do
4:          p ← pop (q)
5:          gen ← vLLM (p, N)
6:          for i ∈ [0, N) do
7:              gen ← gen[i]
8:              lsp_i ← LSP (gen_i)
9:              if notHasErrors (lsp_i) then
10:                 return gen_i
11:             else
12:                 for (err, line) ∈ lsp_i do
13:                     p′ ← gen_i :line + err
14:                     push (q, p′)
```

Note in this pseudocode that the entire context of the program is sent back and forth to the serving server (line 5), which means that the K/V cache prefix can be reused.

VII. Benchmarks

In this section, we explain the two new benchmarks presented in this paper, as well as results from these benchmarks on our modified version of vLLM.

A. Text Analysis Benchmark

Text analysis workloads typically consist of a small set of templates, with textual data inserted into a template for each request.

For this benchmark, we simulate this workload by first generating a set of n unique prefixes to use as the template prefixes. We then send $k > n$ requests, each with a randomly chosen prefix along with a randomly generated body. These requests are either sent batched, to measure latency, or individually, to measure throughput.

B. Chat Benchmark

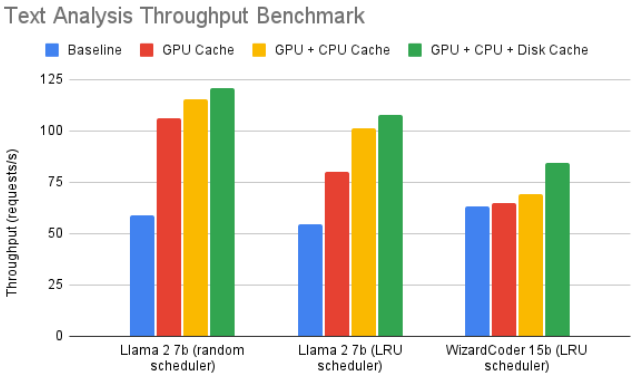
LLM chat conversations typically start with a hidden system prompt which instructs the model, and then a back-and-forth conversation between the user and the LLM.

1. System: {system_prompt}
2. User: {input_1}
3. Model: {output_1}
4. User: {input_2}
5. Model: {output_2}
6. ...

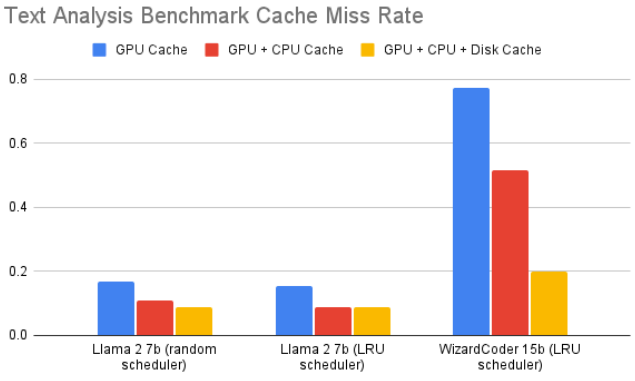
For our chat benchmark, we use ShareGPT[5], a realistic LLM chat dataset used to train the Vicuna language model. Each request contains a randomly sampled user input from the dataset, prepended by a system prompt which we use as the prefix.

Unlike the text analysis benchmark, where requests are passed directly into the model's Python object, the chat benchmark makes requests to vLLM's API server. This allows us to simulate a more realistic chat session which has network latencies and which the user and the model are running on different processes.

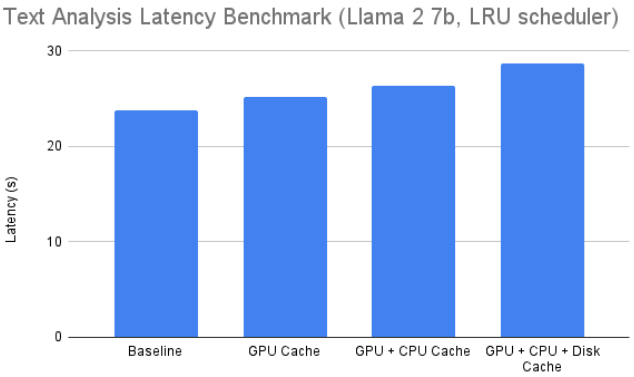
C. Benchmark Results



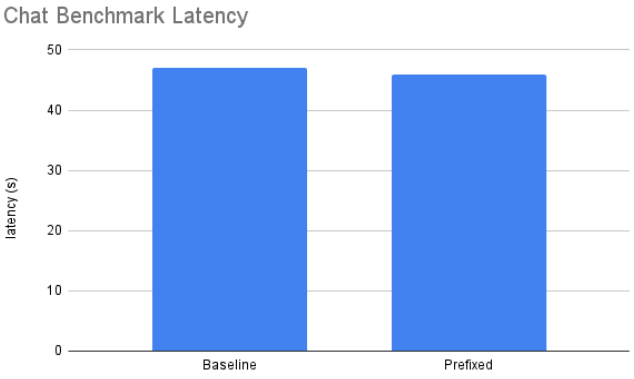
In this benchmark we measure the throughput of each of the serving methods, where a set of 128 unique prefixes are given, whereas each cache only has a certain amount of space. The GPU has space for 8 prefixes, the CPU has space for 32 prefixes, and the disk has space for 128 prefixes. The requests are streamed into the vLLM queue and the overall performance is recorded. All models above are using Fastest-Prefix-First scheduling, and the LRU caching eviction policy. The results for FIFO and Random cache eviction policies are (within some small amount of noise) identical to this benchmark.

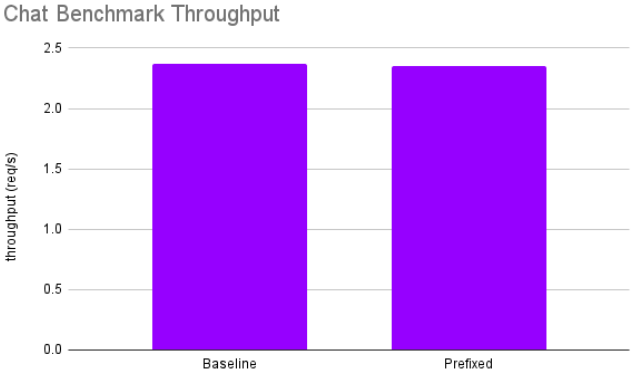


For the text analysis benchmark we record the overall miss rate for all caches.

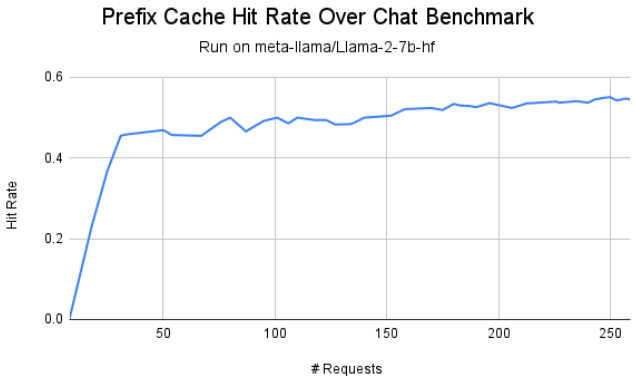


We record that, as expected, latency goes up the more the cache is hit. What is unexpected is that this benchmark shows that the GPU cache latency also increases. However, existing benchmarks that have been posted to GitHub for this feature show latency decreasing. We were not able to figure out why this is.





For the chat benchmark, both latency and throughput are virtually unchanged. This is expected, as we were not able to incrementally grow our prefix matrixes. See the related work section.



VIII. Discussion

Text analysis benchmarks show a strong increase in throughput as well as a small increase in latency, which is to our expectations. Additionally, the miss rate decreases as more storage locations are available, which makes sense since those locations add additional cache blocks to our pool. For a larger model like WizardCoder-15b, this is especially noticeable, indicating that prefix caching can potentially benefit larger models even more.

However, the latency and throughput measurements for the chat benchmark run counter to our priors. Latency and throughput both decrease, meaning that there is better latency and worse throughput when a prefix is specified. This may be because of additional unaccounted overhead in network bandwidth, which reduced the load on the GPU by throttling requests. Further analysis is needed to determine the root cause.

IX. Related Work

A. Model Serving

Deep neural network serving is an active area of research, with a diverse set of solutions tackling a wide set of problems. Foundational works in this field include TensorFlow-Serving[6], an easy and flexible general-purpose model server, Clockwork[7], which attempts to produce deterministic performance from models, and Clipper[8], which combines techniques such as caching, batching, and model selection to reduce latency and improve throughput and accuracy.

These systems are all generalized, which means they can effectively serve any deep learning model. However, this also means they do not take advantage of the autoregressive nature of the Transformer model, and thus cannot make the same types of optimizations as one which does.

B. Transformer Serving

The popularity of the transformer architecture has led to a number of systems that are developed for specifically serving them. These solutions range from kernel optimizations [9] to batching [10], [11] and parallelization [12], [13].

These techniques are largely orthogonal and can be used in conjunction with KV and prefix caching.

C. PagedAttention

The primary work that this paper is based off of is PagedAttention[4].

X. Future Work and Limitations

Swapping to disk works best when the request volume is so high that swapping can happen *in parallel* with the attention computation, so that the caches can always be kept hot and GPU utilization can be maximized. In cases where request count is low, the system starts needing to stall for swaps to complete. Since compute operations are generally orders of magnitude cheaper than bandwidth, this nearly always slows down the overall latency and throughput of the system. We should explore ways to abort a load that ended up being bad and recompute it at the last minute.

One thing we originally hoped to have as part of this paper is the ability to query parts of prefixes. This would be especially useful for the chat benchmark that we wrote, but did not end up running the full suite of benchmarks for against our method because we needed this optimization for it to work well. We want to implement this with a trie, so that all possible prefixes could be queried instead of an exact prefix.

We also need to find a way to swap between the Disk and the CPU without resorting to a new C program and without CUDA, as to avoid the penalty for copying to unpinned memory. Alternatively, we could find a way to mark the memory of a memmapped region as pinned, although this might interfere with other optimizations NVidia has done.

Finally, one thing that we wish to do is to test our system on larger models and more intense workloads. Real-world systems such as GPT-4 have tens to hundreds of billions of parameters, and serve many thousands of requests per minute. We anticipate that these circumstances are where prefix caching will do the most good.

XI. Conclusion

In this paper we introduce prefix caching, an effective way to eliminate redundant KV cache computations over longer time spans than what is currently possible with PagedAttention. Additionally, we provide an implementation of prefix caching built on top of vLLM, as well as benchmarks that measure the efficiency of models for different workloads that would benefit from a prefix cache.

Our results show that for text analysis workloads, caching the prefix significantly increases model throughput. Thus, implementing a prefix cache could result in large improvements to the efficiency of the system for these types of workloads.

Acknowledgement

Thank you Professors John Kubiatowicz and Joseph Gonzalez for advising this project during the course of CS 262A. Thank you to Shiyi Cao for writing the GPU prefix caching kernel and taking your time to explain all the concepts we were confused about. Thank you Simon Mo for for bringing us into the fold of vLLM and always being helpful when we needed it! Thanks to William Brandon for giving us the idea for this project.

References

[1] OpenAI, "GPT-4 Technical Report", *arXiv:2303.08774*, 2023.

[2] K. S. P. A. A. A. Y. B. N. B. S. B. P. B. S. B. D. B. L. B. C. C. F. M. C. G. C. D. E. J. F. J. F. W. F. B. F. C. G. V. G. N. G. A. H. S. H. R. H. H. I. M. K. V. K. M. K. I. K. A. K. P. S. K. M.-A. L. T. L. J. L. D. L. Y. L. Y. M. X. M. T. M. P. M. I. M. Y. N. A. P. J. R. R. R. K. S. A. S. R. S. E. M. S. R. S. X. E. T. B. T. R. T. A. W. J. X. K. P. X. Z. Y. I. Z. Y. Z. A. F. M. K. S. N. A. R. R. S. S. E. T. S. Hugo Touvron Louis Martin, "Llama 2: Open Foundation and Fine-Tuned Chat Models", *arXiv:2307.09288*, 2023.

[3] N. P. J. U. L. J. A. N. G. Ł. K. Ashish Vaswani Noam Shazeer and I. Polosukhin, "Attention is all you need", *Advances in neural information processing systems*, vol. 30, 2017.

[4] S. Z. Y. S. L. Z. C. H. Y. J. E. G. H. Z. I. S. Woosuk Kwon Zhuohan Li, "Efficient Memory Management for Large Language Model Serving with PagedAttention", *arXiv:2309.06180*, 2023.

[5] W.-L. Chiang *et al.*, "Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality". [Online]. Available: https://lmsys.org/blog/2023-03-30-vicuna/

[6] K. G. J. H. L. L. F. L. V. R. S. R. Christopher Olston Noah Fiedel and J. Soyke, "Tensorflow-serving: Flexible, high-performance ml serving", *arXiv:1712.06139*, 2017.

[7] "Serving DNNs like Clockwork: Performance Predictability from the Bottom Up", *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[8] G. Z. M. J. F. J. E. G. Daniel Crankshaw Xin Wang and I. Stoica, "Clipper: A Low-Latency Online Prediction Serving System", *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017.

[9] Z. Y. J. X. Y. M. W. C. W. H. F. Y. L. Z. Lingxiao Ma Zhiqiang Xie and L. Zhou, "Rammer: Enabling holistic deep learning compiler optimizations with rTasks", *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, 2020.

[10] "TurboTransformers: an efficient GPU serving system for transformer models", *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021.

[11] G.-W. K. S. K. Gyeong-In Yu Joo Seong Jeong and B.-G. Chun, "Orca: A Distributed Serving System for Transformer-Based Generative Models", *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022.

[12] M. Z. A. A. A. C. L. D. L. E. Z. J. R. S. S. O. R. e. a. Reza Yazdani Aminabadi Samyam Rajbhandari, "DeepSpeed Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale", *arXiv:2207.00032*, 2022.

[13] A. C. J. D. J. B. A. L. J. H. K. X. S. A. Reiner Pope Sholto Douglas and J. Dean, "Efficiently Scaling Transformer Inference", *arXiv:2211.05102*, 2022.