Characterizing the Behavior and Impact of KV Caching on Transformer Inferences under Concurrency

Jie Ye, Jaime Cernuda, Avinash Maurya, Xian-He Sun, Anthony Kougas, Bogdan Nicolae

Characterizing the Behavior and Impact of KV Caching on Transformer Inferences under Concurrency

Jie Ye, Jaime Cernuda, Avinash Maurya*, Xian-He Sun, Anthony Kougkas, Bogdan Nicolae*
Illinois Institute of Technology, * Argonne National Laboratory
{jye20, jcernudagarcia}@hawk.iit.edu, {sun, akougkas}@iit.edu, {amaurya, bnicolae}@anl.gov

***Abstract*— Pre-training of LLMs and transformers is known to take weeks if not months even with powerful HPC systems. However, inferences are an equally important problem: once pre-trained, the model needs to serve a large number of inferences submitted under concurrency by multiple users. Thus, speeding up each inference request is instrumental in achieving high throughput and latency at scale. To avoid redundant recomputation in each decode iteration, a Key-Value (KV) cache is used to store previously computed keys (K) and values (V), speeding up token generation. GPU memory is primarily consumed by model weights and the remainder is used by the KV cache. Thus, the free GPU space available to the KV cache is a scarce resource that needs to be managed in an efficient way in order to minimize the overhead of redundant recomputations. There are many optimizations applied in this context: batching of inference requests to enable them to run in the same forward pass (and thus increase the parallelism and inference throughput), different KV cache eviction policies (simply drop KV entries and recompute them later vs. swap to host memory), etc. Under these circumstances, the decision of what batching strategy, what KV cache eviction policy to apply and how the KV cache impacts the inference performance is non-trivial. Unlike the case of pre-training, state-of-art studies are scarce in this context. To fill this gap, in this paper we study the impact of KV caching. Specifically, we instrument vLLM to measure and analyze fine-grain metrics (token throughput, KV cache memory access patterns, load balancing of the forward passes), during different inference stages (prefill, decode) in several scenarios that involve concurrent inference requests using several benchmarks. Based on the measurements and associated observations, we identify several opportunities to improve the design of inference frameworks.**

***Index Terms*—LLM inference, KV cache profiling, access pattern characterization.**

I. Introduction

Large Language Models (LLMs) have revolutionized the field of Natural Language Processing (NLP) by introducing powerful AI systems capable of understanding and generating human-like text with remarkable fluency and coherence. These models, trained on vast amounts of data, can perform a wide range of tasks: literature search [1], knowledge distillation [2], and complex reasoning [3] enabling researchers to navigate complex scientific problems more efficiently.

LLMs are only a starting point in unlocking the generative abilities of broader transformers to accelerate science: analyzing and plotting experimental data [4], formulating hypotheses [5], designing experiments [6], and even predicting promising research directions. To this end, modern transformers combine multi-modal data, leverage domain-specific representations, capture correlations through complex attention mechanisms [7] (e.g., self-attention, cross-attention), and compose specialized architectures (e.g., mixture of experts [8]) [9], [10]. They form the core of foundational models (FMs). The potential for innovations in this space has barely been tapped.

In a quest for more emergent behavior (advanced capabilities not explicitly trained for but emerging spontaneously due to the massive scale and exposure to vast amounts of data during training), the scale and complexity of LLMs and transformers continuously increase [11]. This requires larger training infrastructures and drastically escalates their energy footprint. Pre-training of LLMs and transformers is known to take weeks if not months even with powerful HPC systems. For this reason, related work studies have dedicated significant attention to understanding and accelerating the pre-training of LLMs [12], [13].

Scope: inferences are an equally important challenge because once pre-trained, a transformer needs to serve a large number of inferences submitted under concurrency by multiple users. For instance, at Meta, inferences make up 65% of the total energy consumption while pre-training only consumes 35% [14]. The goal is to maximize the throughput of the inference requests while minimizing the latency of each inference request, which is difficult due to the growing complexity of inference frameworks. However, despite this growing complexity, state-of-the-art studies have dedicated comparatively less attention to inferences compared with pre-training. This lack of insight is becoming a significant limitation in the design and development of new inference techniques and ideas. In this paper, we aim to fill this gap.

The need to study KV caching: at the core of each LLM inference request are two phases: (1) a prefill phase, during which the entire input prompt is processed by the attention mechanism in parallel, typically during a single forward pass; (2) a decode phase that involves an iterative prediction of the next most likely token (each in a separate forward pass), which is then appended to the prompt and the process is repeated until a special termination token or a maximum predefined number of tokens is reached.

To avoid redundant recomputations during each decode iteration (whose attention layers repeatedly process the correlations between all pairs of tokens), a Key-Value (KV) cache is used to store previously computed keys (K) and values (V). Each KV pair represents intermediate computations performed by the attention layers on positional embeddings that can be reused from one iteration to another. A fast KV cache has the potential to greatly speed up the token generation and thus the inference requests.

There are two major factors that affect the effectiveness of KV caching. First, inference requests are not processed in isolation, because running a full forward pass to generate a single token has a high initialization overhead and under-utilizes the potential of modern GPUs for massive parallelism. Instead, the phases of inference requests are batched together using various scheduling strategies such that they share the same forward passes as much as possible, while at the same time avoiding imbalance that leads to stragglers [15]. Thus, batching results in complex access

patterns to the KV cache under concurrency. Second, since GPU memory is much faster than host memory, the KV cache needs to use the former for maximum benefits. However, GPU memory is a scarce resource that is primarily consumed by the model parameters and other data structures needed to run the forward passes. While the model parameters and associated data structures use a fixed amount of GPU memory, the space needed by KV pairs grows linearly with sequence length and batch sizes due to the auto-regressive nature of LLMs [16]. Consequently, the KV cache often cannot hold all needed KV pairs in GPU memory and needs to either evict some KV pairs to the host memory or drop and recompute them later. This is a non-trivial decision given the high cost of swapping to host memory vs. the relatively simple recomputations needed to reconstruct KV pairs.

Contributions: this paper contributes in two important directions. First, it proposes a series of techniques to instrument state-of-the-art inference frameworks (such as vLLM [17]) in order to collect detailed qualitative and quantitative measurements during concurrent batched inferences at fine granularity. Second, it uses these techniques to characterize the behavior and impact of KV caching under a variety of scenarios that involve different LLM models, types of inference requests, batching strategies, and cache eviction strategies. We summarize our contributions as follows:

- We highlight the need for hybrid instrumentation that gathers metadata and performance metrics relevant to both GPUs and CPUs, fast on-device buffering techniques for the instrumentation of GPU kernels (notably GPU memory accesses due to KV cache operations), and integration with external tools (e.g., NSight) to log the unpredictable interleaving of monitored tasks under parallelism (§ IV).
- We introduce a methodology to study the impact of KV caching under concurrency. In particular, we focus on how to leverage our instrumentation tool to capture relevant performance metrics and metadata during the experiments, what inference benchmarks and inference runtime settings to choose, and what key qualitative and quantitative metrics to record (§ V).
- Based on the methodology, we run extensive experiments that identify several interesting patterns and trends: batching benefits inference throughput only up to a point, which is less than the full context window; chunking of the prefill phase has a significant negative impact on inference throughput especially when the forward pass is fast; KV cache accesses happen in bursts with significant idle time in between; there is a complex trade-off between recomputations vs. swapping to host memory (§ VI).

II. Background

In this section, we briefly revisit important concepts that modern inference frameworks are based on.

Transformer Inferences: Just like in the case of regular deep learning models, inferences of transformer models are based on *forward passes* that take a prompt (a sequence of tokens) as input and generate a reply as output. The reply is the most likely sequence of tokens that continues the prompt, similar in scope to sequence-to-sequence models. Unlike regular deep learning models, the reply is constructed iteratively, one token at a time. This happens in two phases. First, the *prefill* phase generates the first output token. Then, the output token is appended to the prompt and the next output token is generated in the *decode* step. The decode step is repeated until a maximum number of output tokens is reached or a special termination token (<EOS>) is generated. The initial prefill and the successive decode steps run each in a separate forward pass.

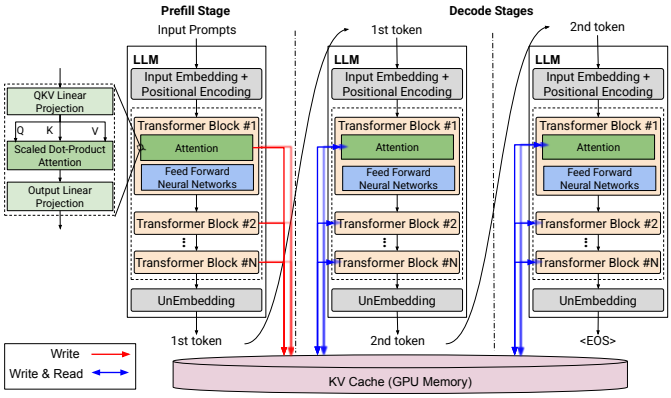


Fig. 1: Transformer Architecture (prefill phase + decode phase).

Transformer Architecture: is typically made of repetitive blocks, each of which is based on *attention* layers [7] that capture the meaning and the correlations between the tokens, which are used in the predictions. Most popular generative Large Language Models (LLMs) such as GPT [9], OPT [18], Llama [19], and Qwen [20] are based on the decoder-only blocks that incorporate *self-attention* layers, a specialization of attention layers that identify positional correlations between all pairs of tokens in a given sequence of tokens. Besides attention layers, there are other layers that are part of each block, but their role is beyond the scope of this work. The architecture and inference process is illustrated in Fig.1. The forward pass follows a sequential pattern: the output of one block is used as input for the next block. Thus, parallelization of computations happens within each block but not across blocks. Once the final output token is generated, it is appended to the prompt and the new input is used in the next forward pass.

Self-Attention Layers: the correlations captured by the self-attention layers between all pairs of tokens are based on *positional embeddings* that give each token a position-dependent meaning. Specifically, given a sequence of tokens represented as a vector X, three matrices $Q = X \cdot W_q$ (queries), $K = X \cdot W_k$ (keys) and $V = X \cdot W_c$ (values) are computed based on learned weights W_q, W_k and W_v that project each token into a multi-dimensional set of d_k features. Then, an attention score for all tokens $A = (Q \cdot K^T \cdot V)/\sqrt{d_k}$ is computed that is used further in successive layers and blocks to eventually predict the next token. Since these operations are matrix multiplications, they are highly parallelizable. Furthermore, to capture correlations from multiple perspectives simultaneously, Multi-Head Attention (MHA) [7] is widely employed in LLMs, which relies on the idea of using different W_q, W_k and W_v (heads) to compute different attention scores in an embarrassingly parallel fashion (another level of parallelism in addition to parallelization of matrix multiplications), which are then aggregated into a final attention score. Due to the large number and size of intermediate data structures, attention layers consume large amounts of GPU memory (e.g., about 5.5 GB for running 1 request with 1024 input tokens using the Yarn-llama-2-7B model in a single forward

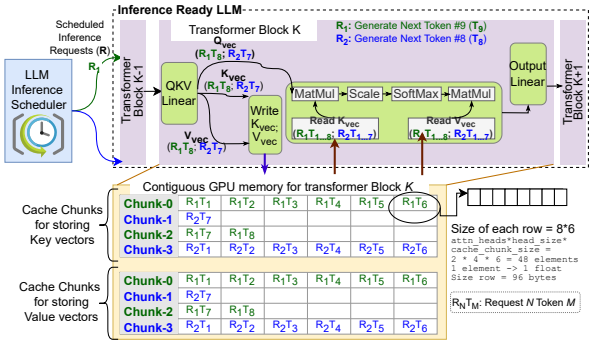


Fig. 2: KV-Cache layout of each Transformer Block in vLLM.

pass). Other attention optimizations (e.g., Grouped-Query Attention (GQA) [21] and Multi-Query Attention (MQA) [22]) have been proposed to strike a trade-off between memory space and quality. In this work we focus on traditional MHA.

KV Caching: after the prefill phase is done, it is important to note that X remains unchanged in successive iterations except for the last token that was appended. Thus, during the decode phase, it is enough to compute only the attention of the new token. We have a new row $Q_i = x_{new} \cdot W_q$, but we can reuse K^T and V from the previous iteration (to which we append a new column and row respectively that correspond to the new token). With each newly generated token, K^T and V are growing. Thus, a naive caching solution that pre-allocates a contiguous memory space to fit the worst-case scenario (maximum number of tokens) is not only wasteful (most of the time, the termination token is encountered before the maximum number of tokens), but also prone to fragmentation and synchronization bottlenecks (due to concurrent access). To avoid this issue, *PagedAttention* is used by state-of-the-art KV caching approaches such as vLLM [17]. Specifically, the KV cache is organized into fixed-sized chunks that can be allocated on-demand to accommodate the growth of K^T and V. Each attention head manages its own set of chunks, which enables scalable access under concurrency. The access patterns shift from write-intensive during the prefill phase to read-intensive (with occasional writes) during the decode phase. Fig. 2 illustrates vLLM's Blocked KV cache data layout within each Transformer Block.

Batching of Inference Requests: thanks to caching, the decode phase needs to perform cheap incremental computations. If only one inference request is allowed to run in each forward pass, this would severely under-utilize the potential of GPUs for massive parallelism. Even in the case of the prefill phase, if the number of tokens in the prompt is small, this would also lead to GPU under-utilization. Therefore, a common strategy is to batch multiple independent requests together [23], [24] such that they share the same forward pass. This is possible because one can build large matrices out of independent requests and simply use different weights (e.g. from different attention layers) for different parts of the matrices inside the same GPU kernels. Since each inference request spans over multiple forward passes, different scheduling trade-offs are possible. For example, Orca [25] proposes a fine-grained approach known as *iteration-level batching* or *continuous batching*, which allows a new inference request to join the current batch or an existing request to leave the current batch at any iteration. Another important aspect is how to interleave the prefill and decode phases. Some strategies prioritize the prefill phases (i.e. batch together the prefill phases of the scheduled inference requests, then, after all prefill phases are done, batch together the decode phases). The intuition behind this strategy is to achieve load balancing in the forward passes during the decode phases (i.e., one token for each inference request), at the expense of potential imbalance during the prefill phases. For long prompts, this imbalance is significant [26]. Other strategies mix prefill phases with decode phases. In this case, the prefill phases may stretch over more forward passes than in the previous case, increasing the risk of stragglers that slow down some forward passes, but on the upside, the load balancing of the initial forward passes may be better. Another common optimization is chunking of the prefill phase: instead of trying to fit the full prompt of an inference request in a forward pass, a fixed maximum number of tokens (chunk size) can be taken from each inference request. This approach is illustrated by SarathiServe [26] and DeepSpeed-MII [27] and results in better load balancing when batching prefill phases together, at the expense of stretching the prefill phases over more forward passes. Batching strategies have non-trivial implications on KV caching because they increase the number of accesses under concurrency proportionally to the degree of load balancing (i.e., the better the load balancing is, the more concurrent accesses to the KV cache, thus putting more stringent scalability requirements on the KV cache).

KV Cache Swapping vs. Recomputations: when activating batching, the KV cache needs to hold the continuously growing K^T and V matrices of multiple self-attention heads belonging to multiple inference requests at the same time. Given that the spare GPU memory used for caching is scarce (especially when using large transformers with many parameters), it is not feasible to fit all these intermediate results in the spare GPU memory in most situations. Thus, the KV cache needs to implement an eviction policy. A common system-level solution is to use multi-level caching that simply evicts some cache chunks to the host memory, then brings them back to the GPU cache when they need to be read, which is similar to the *swapping* mechanism employed by operating systems to run out-of-core processes. However, swapping to host memory is typically a slow I/O operation (it is subject to the low bandwidth of PCI-Express links that typically connect GPUs with the host side). Given the low latency of GPU memory accesses and massive parallelism of GPU computations, under certain circumstances, it may be faster to recompute some K^T and V matrices instead of using swapping. In this case, the eviction policy of the KV cache can simply drop some cache chunks. The question of when is swapping beneficial over recomputations and the other way around is non-trivial since it depends on many factors: batching strategy, complexity of the computations in the forward passes, etc. Therefore, it is important to study this dimension.

III. Related Work

In addition to the strategies mentioned in § II, there are several other batching and KV caching optimizations that were described in the literature. FastServe [28] uses iteration-level preemptive scheduling to minimize queuing delays for the batching of long

requests, thus enabling better load balancing when there is significant variability in the inference request sizes. Multi-level caching using disaggregated resource management is also a popular direction. For example, systems like Splitwise [29], DistServe [30], and TetriInfer [31], separate the prefill and decode phases, allowing their scheduling and batching on different compute nodes. Such approaches enable better utilization of the aggregated GPU memory of multiple GPUs, at the expense of introducing I/O overheads due to the need to access remote GPU memory in order to reuse K and V. SwapAdvisor [32] uses genetic algorithms to control memory allocations and swap decisions. vDNN [33] employs offloading and prefetching. TSPLIT [34] uses tensor splitting to enable fine-grain control of the KV cache. vLLM [17] uses either recompute or swap, implementing an all-or-nothing eviction policy configurable by the user. On the other hand, STR [35] can dynamically combine both techniques. Given the multitude of trade-offs, many such approaches are complementary and can be combined. For the purpose of this work, we stick to the most popular strategies mentioned in § II.

Profiling of Transformer Inferences: The PyTorch Profiler [36] coupled with the PyTorch ecosystem is capable of monitoring GPU memory usage and categorizing it over time. NSight System [37], NVIDIA's profiler, focuses on profiling GPU kernels, their interleaving across different streams and their overlappings with host-GPU memory transfer operations. Omnitrace [38] serves as AMD's equivalent solution. For Python-based applications, Scalene [39] provides high-precision, per-line memory profiling, capable of detecting memory leaks and tracking GPU memory trends in real-time. Such low-level tools are indispensable building blocks for tools that aim to understand higher-level patterns under concurrency. Despite many optimization strategies introduced by state-of-the-art for transformer inferences, there is limited availability of tools and studies that help understand the patterns of KV caching. To our best knowledge, *we are the first* to build such tools and use them for the purpose of capturing and characterizing the impact of KV caching on transformer inferences under various degrees of concurrency introduced by batching, as well as different KV caching strategies.

IV. Instrumentation

Our first contribution is an open-source [1] tool that can be used to instrument state-of-the-art inference frameworks for the purpose of understanding KV caching patterns at fine granularity. We illustrate and implement our instrumentation tool based on vLLM, but the underlying ideas and design principles are generic and can be applied for other frameworks such as DeepSpeed-MII [27]. Specifically, our tool is based on the following design principles:

Hybrid collection of metadata and performance metrics: during the forward passes needed to run the inference requests, there is a tight interaction at fine granularity between the host and GPU side. For example, tasks that start on the host side need to schedule GPU kernels and memory copy operations, but may finish on the GPU side. Thus, coordinating the timestamps of events that are triggered on different devices is needed in order to accurately measure the duration of representative metrics. To enable such capability, we build an event management infrastructure that allows GPU kernels to directly record when an event happens in a separate on-device buffer, which is then flushed to the host memory and combined with host-side buffers later in order to match the events and generate the desired metrics. This approach has two advantages: (1) it avoids the synchronization overheads introduced by host-side-only monitoring approaches (e.g. waiting for CUDA events); (2) it avoids the overheads of in-situ processing, aggregation, or logging of metrics. This is important for the purpose of minimizing the influence of instrumentation on the accuracy of the measurements.

On-device multi-buffering to capture KV cache access patterns: The most challenging aspect of understanding the impact of KV caching on inferences is to accurately capture the duration of the very fast operations such as reads and writes to the GPU memory where the working set of the KV cache is residing (and thus where the most frequent accesses under concurrency happen). Due to the organization of the KV cache as chunks rather than as a contiguous space and the parallelization at different levels (batching, multi-head attention layers), it is often the case that read and write operations happen in different GPU kernels. Thus, it is not enough to coordinate events between the host side and the GPU side, but also between different GPU kernels. Furthermore, since the accesses happen under concurrency, the recording of events also needs to happen under concurrency, raising scalability issues. As a consequence, we cannot afford to log individual events directly from GPU kernels. Instead, we devise a strategy that allocates separate, on-device buffers where related read and write events are grouped together and collected. Each read and write event only records minimal metadata. The buffer size is large enough to hold all events produced by all concurrent threads in a single iteration. Between iterations, these buffers are flushed to the host memory and then processed asynchronously on the CPU. To avoid the need for synchronized access to the buffers, each thread is assigned a separate segment of the buffer and the events are interleaved later during the asynchronous processing.

Complement Low-Level Profiling Tools: we need to complement the events and measurements of operations that happen within GPU kernels with performance profiling information that is collected by low-level tools such as NSight [37]. To this end, we opted for a post-processing infrastructure that leverages the SQLite database format provided by NSight to load the time-series of events into Pandas DataFrames. Then, we combine the time-series with the JSON files, which results in a unified view of all performance metrics and metadata needed for the analytics.

V. Methodology

A. Leveraging the Instrumentation Tool

Using our tool described in § IV, we have instrumented the forward pass of vLLM [17] as follows. We identified in what GPU kernels the writes and reads to the KV cache are issued (`reshape_and_cache_kernel` and `paged_attention_kernel` respectively). Then, we divided the read logging buffer into two segments: the first segment logs the K cache read events, while the second segment logs the V cache read events. The separation is important because K and V are accessed at different times during the attention computation. For GPU events, we log the chunk ID, offset in the chunk, duration of operation, thread ID. On the host side, we record the metadata of each transformer block's time slice

[1] https://github.com/Jye-525/UnBoxKV-IO

TABLE I: Configurations of different models.

Model	Yarn-Llama2-7B	Llama-3.1-8B	OPT-13B
Model Size (GB)	14	16	26
Hidden Size	4096	4096	5120
Layers	32	32	40
Attention Heads	32	32	40
Key-Value Heads	32	8	40
Max Context-Length	64K	128K	2K

during the forward pass (ID of the forward step, ID of the transformer block, the kernel start/end timestamp, each request start/end timestamp, and additional metadata corresponding to what requests were batched together in the forward pass). The asynchronous post-processing on the CPU reconstructs the event timeline by interleaving the event timestamps for each read and write request and their corresponding forward pass orders. We then leverage the resulting time-series to capture the duration and composition (what requests are batched together) of the forward passes, as well as the read/write access pattern at fine granularity.

B. Metrics

We employ different metrics in our experiments according to different exploration scenarios. We often use *throughput (tokens per second)* when exploring how different batching strategies impact the inference performance, which is either aggregated over multiple forward passes per-phase (prefill vs. decode) or over both phases (reflecting the end-to-end performance). Token throughput is not sufficient to explain the impact of KV cache evictions. Therefore, for recomputation vs. swapping studies, we utilize the *resume overhead* and *KV cache utilization* as the key metrics. Specifically, we measure the aggregated time taken to generate a new token when resuming the evicted inference requests (recomputation duration and swap-out/swap-in duration respectively), accounting only for the delays caused by the recomputation vs. swapping for the group of batched requests (thus accounting for overlaps during concurrency). The *KV cache utilization*, represented as a percentage, is calculated by dividing the number of blocks currently in use by the total number of blocks available in the cache.

C. Workloads

Models: in our evaluations, we use three representative LLMs of different sizes and attention mechanisms summarized in Table I. Yarn-Llama-2-7B is an extension of the widely used Llama-2 7B model for supporting longer context instead of the default 4K tokens. Llama-3.1-8B is a more recent model using GQA wherein 4 attention heads share a key-value head to save KV cache memory, demonstrating a different architecture in transformer blocks. Lastly, we study the OPT-13B model, the largest among widely used models that fits an A100-40GB GPU. The Yarn-Llama-2-7B and Llama-3.1-8B models use `BF16` precision to support a wider range of values for the model parameters, while OPT-13B model uses `FP16` precision to preserve accuracy constraints.

Fine-tuning of batching and caching strategies: vLLM provides parameters to enable or disable chunked-prefill and to set the chunk size. We extend vLLM by adding another user-configurable parameter to enable or disable mixed batching. If both mixed batching and chunked-prefill are off, vLLM will use its default continuous batching. Additionally, we modify vLLM to make recompute and swap configurable for evaluating KV cache overflow management.

Dataset and inference request patterns: we use the SharedGPT [40] dataset, a collection of user-shared conversations with ChatGPT, in which sequence lengths range from 4 to 2.3K tokens. Besides SharedGPT, we also use synthetic datasets generated by a fixed-length generator and a variable-length generator that follows a *Zipf* distribution. The generator allows us greater flexibility at controlling the input and output sequence lengths. It produces random tokens, which is acceptable because we are only interested in the performance metrics, and they are not affected by the content of the tokens. We choose the *Zipf* distribution because most real-world datasets adhere to it [41]. The fixed-length generator produces requests with predefined prompt and output lengths, while the variable-length generator creates requests with sequence lengths within a specified range, allowing for a configurable *prompt-to-output ratio*. These synthetic datasets help us analyze the impact of the KV cache on inference and the handling of KV cache overflow under different conditions.

D. Experimental Setup

Platform: we conduct all the experiments on the ALCF's Polaris platform [2], a 560-node HPE Apollo 6500 Gen 10+ based system. Each node has 4×A100 GPUs with 40 GB HBM2 on each (aggregated GPU memory of 160 GB), 1× AMD EPYC Milan 7543P processor with 32 Zen3 Cores (64 threads), 1×512 GB DDR4 RAM, and 2× NVMe SSDs of 1.6 TB each. The GPUs are interconnected via NVLink (600 GB/s). The peak unidirectional host-to-device (H2D) and device-to-host (D2H) throughput for pageable host memory are 19 GB/s and 12 GB/s respectively. For pinned host memory, the H2D and D2H throughputs are 24.5 GB/s and 26.1 GB/s. Since we aim to explore the KV cache impact on inference by studying the widely used moderate-sized models ($\leq$13B), we limit our experiments to a single A100 GPU. This setup allows us to better analyze the intricate memory behaviors and impact of the KV cache, providing a baseline for understanding the potential limitations. Moreover, single-GPU configurations are commonly adopted in real-world cloud deployments due to cost and power limitations, highlighting the importance of evaluating single-GPU performance across various models.

Software: each node runs NVIDIA Driver 535.154.05, CUDA 12.4.1, Python 3.11.8, and PyTorch 2.2 on top of the Cray SUSE Linux Enterprise Server 15.4 OS. Our experiments utilize the vLLM (0.4.2) to serve inference queries since it is popular in the community and many studies use it as a baseline for comparison [42], [43].

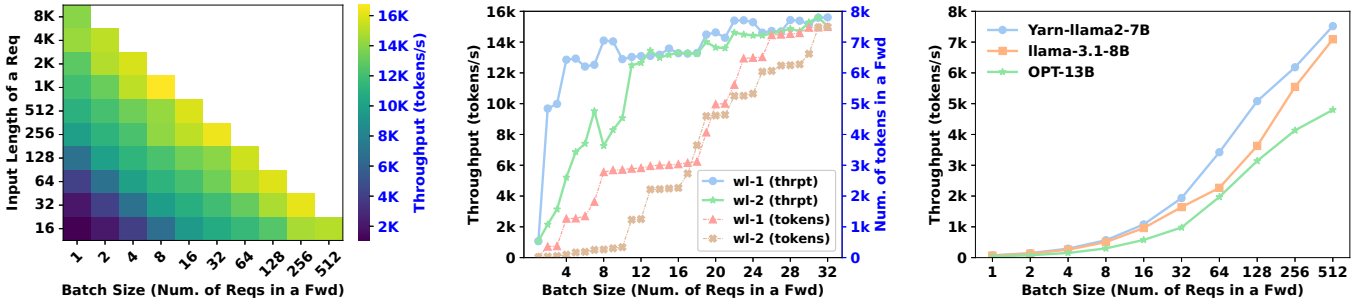
VI. Analysis

Based on the methodology discussed in § V, this section studies the behavior and patterns of running inference requests under various scenarios as introduced below.

A. Throughput under Different Batching Strategies

First, we assume enough GPU memory is available to run all scheduled inference requests without overflowing the KV cache capacity allocated from spare GPU memory. This is a best-case scenario that maximizes the benefits of KV caching.

[2]https://docs.alcf.anl.gov/polaris/hardware-overview/machine-overview/



(a) Throughput of fixed-length prefill-only inference requests (Yarn-llama-2-7B model).

(b) Throughput of variable-length prefill-only inference requests using two different randomized orders (Yarn-llama-2-7B model).

(c) Throughput of decode-only requests across varying batch sizes.

Fig. 3: Throughput of prefill-only or decode-only requests processed in a forward pass.

Prefill vs. Decode phase: given the distinct computational characteristics of the prefill and decode phases, we begin with a study that isolates their performance. Fortunately, this has already been implemented by the batching strategy that prioritizes the prefill phase.

Fig. 3a shows the throughput of the prefill phase when using fixed-sized inference requests (using the generator mentioned in § V-C) batched together in the same forward pass. We chose the Yarn-llama-2-7B model that supports a relatively large context window of 8K tokens while leaving enough spare GPU memory available to enable full GPU caching. We run a single forward pass and fill the context window with an increasing number of inference requests of variable sizes (starting from a large 8K token request up to 512 inference requests of 16 tokens each). As expected, the peak throughput is reached when the context window is fully filled. There is minimal difference in terms of how many requests are used to fill the context window, showing that the fused matrices that are computed in the same kernels using different attention layers have negligible overheads. Interesting to note is that the token throughput is close to peak also when the context window is only half full, which means it is not necessary to fill the full context window in order to saturate the parallelism of GPUs.

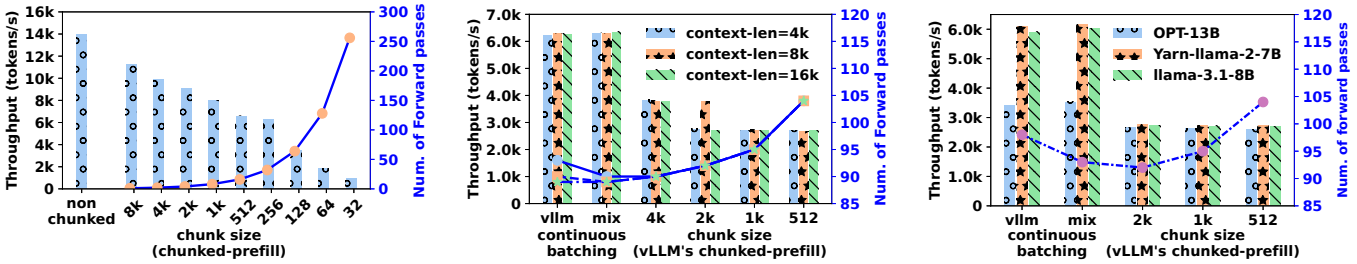
Fig.3b focuses on variable-length inference requests, which are generated from the SharedGPT dataset. Again we vary the batch size (number of requests sharing the same forward pass). This time, depending on what inference requests are batched together, the total number of tokens may differ for the batch size. To study this effect, we shuffle the inference requests to obtain two different orders (denoted *wl-1* and *wl-2*). The total number of tokens as a function of batch size is on the right side of the Y-axis, while the corresponding throughput is on the left side of the Y-axis. A similar observation as in the case of fixed-sized requests applies: the total number of tokens is the most impactful factor that determines the throughput, and a high throughput is reached even when the context window is significantly under-utilized (even at 25% utilization we reach 75% of the peak throughput). While not explicitly illustrated, we verified that the same observations hold for other models (*OPT-13B* and *Llama-3.1-8B*).

Fig.3c shows the throughput of decode-only requests across various batch sizes running with different models. In this case, there is no difference between fixed-sized requests and variable-sized inference requests, because in both cases the forward pass processes one token from each request. As expected, the throughput is solely affected by the batch size (number of requests). It grows rapidly for a small number of requests and exhibits a slowdown later. The transformer architecture also plays an important role: the OPT-13B model peaks at 4k tokens/second, while the other models can go much higher. In any case, given the extreme granularity of requests during the decode phase (a single token), the overall throughput is much lower compared with the prefill phase).

Key observations: the prefill stage reaches close to peak token processing throughput even when the context window of the forward pass is significantly under-utilized. KV caching performance is not affected by such differences. This has important implications in the design of batching strategies, as it leaves a significant degree of freedom in terms of how many forward passes to run and how many requests to batch together in each of these forward passes. A similar observation holds for the decode phase. However, in this case the peak throughput is much lower, resulting imbalance between the two phases that needs to be accounted for.

Continuous-batching vs. chunked-prefill: as explained in § II, a common strategy to avoid long forward passes due to imbalance caused by the prefill phase of large inference requests is to split the prefill phase into chunks. However, this comes at the expense of overheads due to chunk management and launching of multiple forward passes. Thus, in our next series of experiments, we aim to understand this effect. For continuous-batching, we evaluated two approaches: (1) vLLM's prefill-prioritizing policy, which handles prefill and decode requests in separate forward passes, and 2) a mixed (or hybrid) policy, which allows to process prefill and decode requests together in a forward pass.

Fig.4a shows the throughput of a single 8K request (prefill phase) that fills the whole window of the Yarn-llama-2-7B model (8K) used in the experiment. Since we are dealing with a single request, the prefill-prioritizing vs. mixed batching are equivalent and we refer to it as *non-chunked*. We compare this with the chunked-prefill batching using a variable chunk size (number of tokens included on the X axis). With a smaller chunk size, the number of forward passes increases. As expected, the throughput



(a) Throughput of a single 8K prefill-only request using variable chunk sizes (Yarn-llama-2-7B model, 8K context window).

(b) Throughput of variable-length requests running with Yarn-llama-2-7B model configured with different context lengths (#Reqs=20).

(c) Throughput of variable-length requests running with different models that set with same context-length (2K) (#Reqs=20).

Fig. 4: Throughput and the number of forward passes for continuous vs. chunked-prefill batching.

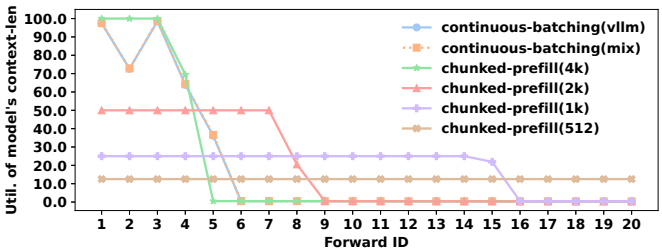


Fig. 5: Utililization of Yarn-llama-2-7B model's context-length over 20 forward passes across various batching strategies (context length=4K).

of running the prefill of the entire request drops dramatically for a smaller chunk size. Interesting to note is that an 8K chunk size (thus a single forward pass) does not reach the same throughput as the non-chunked batching strategies. This is due to the vLLM using the more efficient *FlashAttention2* algorithm [44] for processing non-chunked requests, while chunked-prefill requests are limited to a less efficient implementation.

Fig.4b presents the end-to-end throughput (prefill and decode) for 20 variable-length requests using the Yarn-llama2-7B model, configured with different context windows. The goal is to investigate if the context window impacts throughput. We create 20 variable-length requests using the *Zipf* generator and send them to vLLM simultaneously. We observe that the context window has minimal impact on the throughput for a given batching approach (which is consistent with the observations about limited impact of under-utilizing the context window). Interesting to note though is that prioritizing the prefill phase vs. allowing mixed prefill and decode phases show minimal differences in favor of the latter. Also surprising is the sharp drop in throughput when comparing the chunked-prefill strategy with the non-chunked strategy: while the number of forward passes plays a significant role (visible drop going from 4K to 512 chunks), more important is the impact of the flash-attention optimizations.

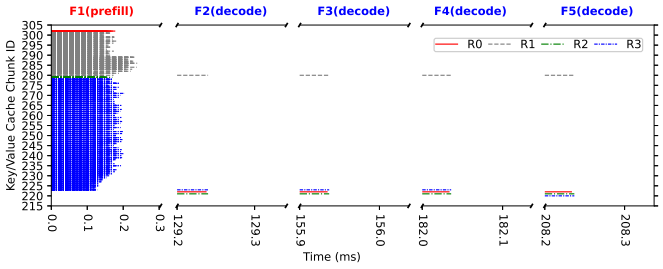
To complement the previous results, Fig.4c depicts the end-to-end throughput for 20 variable-length requests using different models, configured with the same context window (i.e., 2K), across various batching strategies. The goal is to assess whether the model architecture affects throughput. The workload used is the same as in Fig.4b. Our findings show that model architecture impacts the throughput when using continuous-batching approach. For instance, OPT-13B model is about 1.7x slower than the Yarn-llama-2-7B model, which can be attributed to its higher computational demands since more transformer blocks and more attention heads per block. Additionally, model architecture has minimal effect on throughput when using the chunked-prefill approach.

Fig.5 presents the utilization of the Yarn-llama-2-7B model's context window (4K) over 20 forward passes across various batching strategies. The goal is to explore to what degree the context window is filled over time from one forward pass to another. As expected, the 4K chunked-prefill strategy and continuous-batching fully utilize the model's context length in the beginning, but all approaches see a sharp drop later. Given the performance penalty of chunking, this means it may be beneficial to deactivate chunking later when the decode phase dominates.

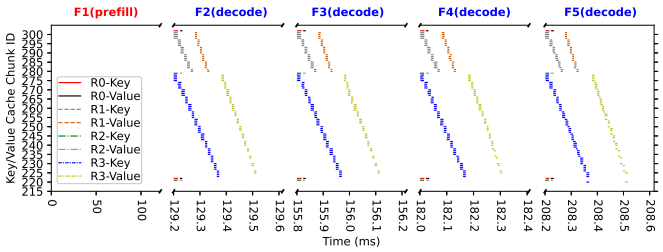
Key observations: Although promising as a strategy to improve load balancing during the prefill stages of batched inference requests, chunking strategies currently suffer from limitations caused by sub-optimal implementation of attention computations, which results in low throughputs even in the presence of KV caching. Another limitation is the unnecessary overhead of using chunking strategies (even with optimal attention implementations) when the decode phases dominate the forward passes. This leaves an opportunity to design adaptive strategies that selectively use chunking based on the prefill imbalance.

B. KV Cache Access Patterns

Based on the instrumentation described in § V-A, in our next series of experiments we aim to understand the KV cache access patterns under concurrency. Just like before, we still assume the KV cache is large enough to avoid recomputations or swapping. Fig. 6a illustrates the KV cache write pattern over time. It covers 5 forward passes (F1-F5) for 4 inference requests of variable sizes (R1-R4). For simplicity, we only show the write pattern in transformer block 0 (it is repetitive for the other transformer blocks). Also, due to the large number of cache chunks and write operations involved, we depict the cache chunk ID (instead of memory offset on the Y axis). The figure shows two distinct patterns corresponding to the prefill and decode stages. Between the stages (prefill or decode), there are significant gaps, which is why the X-axis shows interruptions. As can be observed, most of the writes happen in the prefill phase in parallel (same beginning



(a) Key/Value cache write pattern over time in transformer block 0.



(b) Key/Value cache read pattern over time in transformer block 0.

Fig. 6: KV cache write and read pattern over time in transformer block 0, covering 5 forward passes of 4 inference requests. The Y-axis denotes the chunk ID of the K or V cache. Each cache chunk is 128 KB (a K or V cache in a single transformer block). All transformer blocks hold the same number of KV chunks. The model is Yarn-llama-2-7B.

timestamp) across different cache chunks. This shows that multi-head attention parallelization combined with batching puts high concurrency pressure on the KV cache initially. Later, during the decode phases, incremental computations on a single token result in much less write pressure overall, but still under concurrency. During the gaps in the X axis, other transformer blocks (running serially) repeat the same pattern. Interesting to note though is that there are significant periods of no KV cache accesses even outside of the gaps (due to the computations of other layers).

Fig. 6b is symmetrical to the previous figure but focuses on the read pattern under concurrency. As expected, no read accesses happen during the prefill phase (as there are no previous intermediate results that can be reused). In the case of the decode phases, there are several interesting observations. First, the read pattern repeats not only from one forward pass to another, but is also highly predictable within each forward pass. Specifically, each inference request accesses the cache blocks in increasing order of IDs (even under multi-head attention parallelism) and never switches back and forth between different cache chunks. Second, there is less read parallelism within the same inference request compared with the write pattern (different start timestamps). However, there is full parallelism across different inference requests. Third, there is a significant lag between the K accesses and V accesses.

As depicted, no KV cache read during the prefill stage and most read operations occur during the decode stages. Each decode step needs to read all the previously computed keys and tokens for each request. Moreover, other decode steps follow a consistent Key cache and Value cache read pattern. In every transformer block of a decode step, all requests read from the Key cache simultaneously and read the Value cache after the key

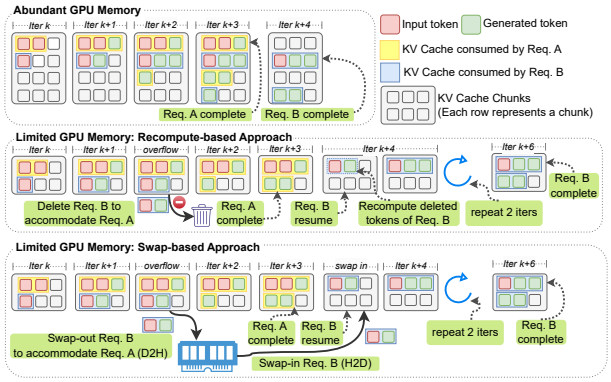


Fig. 7: A comparative illustration of recomputation and swap approaches for managing KV cache overflow.

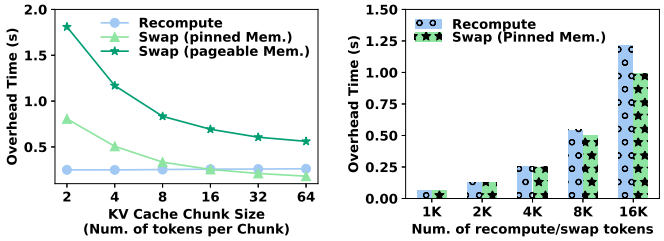
cache. Within a given request, although each request reads the cache chunks (key or value cache chunks) in parallel, its read parallelism is configured based on the number of CUDA warps (e.g., it is 4 in this test case). Additionally, the time cost of a decode step is about 25ms, which is long enough for perfecting the KV cache of the next step in advance. The sequential access between transformer blocks makes it possible to interleave offloading and prefetching the KV cache during a forward.

Key observations: The write pattern shows significant periods of inactivity within the same phase of the same transformer block, both for prefill and decode. Furthermore, since transformer blocks are processed serially, even larger periods of inactivity are observed between the phases of the same block. This means the cache chunks are immutable for long periods of time. Coupled with highly regular and relatively scarce read patterns, this leaves ample opportunities to asynchronously reorganize and dynamically optimize the KV cache (including swapping strategies) that can be exploited by future work.

C. Recomputations vs. Swapping

Next we focus our studies on scenarios that require more KV cache space than is available on the GPU, which triggers an eviction policy that is either based on recomputations or swapping, as mentioned in § II.

Behavior of recomputations vs. swapping: to understand better how these two opposing strategies impact the KV caching behavior when the inference requests are batched together, we illustrate in Fig. 7 a representative example. Consider two inference requests, A and B, that are batched together. The key observation is that we don't know apriori how many tokens each of these two requests will generate, which means we don't know how many decode phases each of them will trigger, and, important for the purpose of KV caching, how much cache space to reserve. Thus, inference frameworks implement an optimistic approach. They allow both of them to share the same forward pass as long as there is enough cache space available on the GPU. However, with each new forward pass, the KV cache fills up. When no more space is available, B is suspended and its K and V matrices evicted such that A can continue. The two strategies differ only in terms of how evictions are handled: swapping blocks until K and V are flushed



(a) Variable KV cache chunk size. (b) Variable number of evicted tokens. Cache chunk size is 16.

Fig. 8: Overhead of recomputations vs. swapping for a single eviction for variable KV cache chunk size and number of evicted tokens (Yarn-llama-2-7B model, 16K context window).

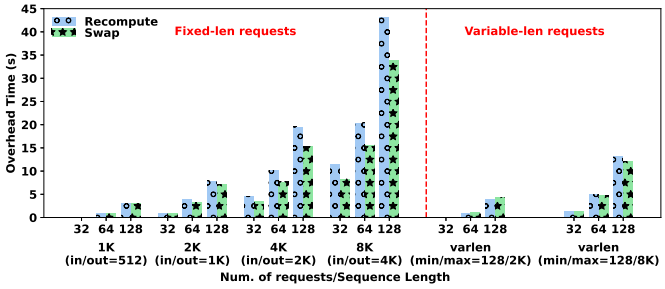


Fig. 9: Overhead of recomputations vs. swapping for a variable number of submitted requests with different sequence lengths (fixed-length vs variable-length) using the Yarn-llama2-7B model, 16K token context window.

to host memory, while the recomputation strategy simply drops them. Later, when A has finished, B can resume. At this point, the swapping strategy resumes the decode phases by bringing K and V back to the GPU cache from the host memory, while the recomputation strategy starts from scratch with the prefill phase.

Key observations: Important to note is that the recomputation strategy may introduce prefill stages at any point, thus complicating the load balancing that the batching strategy aims to achieve. Thus, the KV caching eviction policy has significant implications on the batching strategy, underlining the importance of co-design. Furthermore, current implementations handle cache evictions at the granularity of a full request. This is suboptimal if the inference requests chosen to continue overflow the KV cache capacity only by a small amount, highlighting an opportunity for partial evictions.

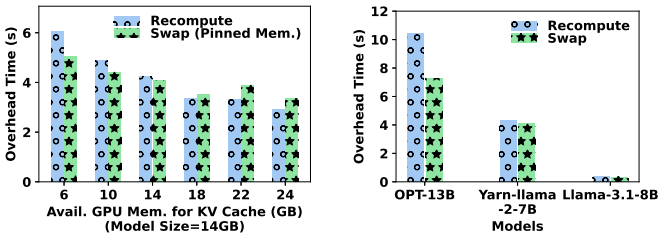
Variable KV cache chunk size: recomputation vs. swapping strategies are impacted by a variety of factors. Next, we isolate the effect of such important factors. One such factor is the KV cache chunk size, which affects the performance of swapping (many small GPU-host transfers achieve a lower I/O throughput compared with fewer larger transfers). Note that the KV cache chunk size S is a function of number of tokens and other model parameters (specifically $S = 2 \cdot N \cdot L \cdot D \cdot sizeof(FP16)$, where N is number of tokens, L is number of transformer blocks, D is size of K and V per token). The result is doubled because K and V occupy the same amount of space each. For Yarn-llama-2-7B L and D are 32 and 4096 respectively. We use N instead of S when referring to cache chunk sizes, since the KV cache is configured using N instead of S.

To study this effect, we generate two large synthetic requests R1 (430 prompt tokens and 1024 reply tokens) and R2 (3600 prompt tokens and 496 output tokens) that force a single eviction. We batch them together and run the corresponding inferences. In the case of the swapping strategy, we evaluate two approaches to allocate the host memory: *pageable* and *pinned* memory pages. The former provides slower I/O transfer throughput but allows the OS kernel more flexible host memory management (e.g. to allocate unused pages for filesystem caching). The latter allows faster I/O transfers thanks to DMA that does not involve the CPU, at the cost of reserving the memory pages for exclusive access. Fig. 8a depicts the results. As expected, the KV cache chunk size has minimal effect on recomputation overhead because recomputations access only the GPU memory, which is optimized for small accesses. On the other hand, small chunks significantly increase the overhead of the swapping strategy, as host-GPU memory transfers are not optimized for small sizes. Interesting to note is that both pageable and pinned host memory suffer from similar decrease in performance for decreasing chunk sizes, which can be explained by the fact that the paging overhead remains constant (a host memory page is small, usually 4 KB). Also interesting to note is that paged host memory is not fast enough at any KV cache chunk size to offset the recomputation overheads. On the other hand, pinned host memory does so even at small chunk sizes (16 tokens or about 8 MB).

Variable number of evicted tokens: next we study how much slowdown the inferences suffer as the evicted size S (corresponding to an increasing number of tokens N, X axis of Fig. 8b) varies. In this case, we configure the swapping strategy to use only pinned host memory and a fixed cache chunk size of 16 tokens. Similarly, we use two synthetic inference requests R1 and R2, but vary prompt and output sizes such that a single eviction is triggered for R2 after a given number of tokens was processed. As expected, both strategies experience an increasing slowdown for an increasing quantity of cached data being evicted. However, there is an increasing gap between recomputations and swapping, with the latter exhibiting a significant edge (almost 20%) over the former especially at a large quantity of evicted intermediate results (corresponding to 8K and 16K tokens).

Variable sequence length (prompt + reply): next, we study the impact of variable sequence length (the sum of prompt and reply length). We evaluate both fixed-length and variable-length requests generated using a fixed-length and *Zipf* generator, as explained in § V-C. For variable-length requests, we configure the prompt-to-output ratio to 0.5 to emulate the short input and long output scenarios. The requests are sent to vLLM simultaneously. As can be observed in Fig.9, the overhead from recomputation and swapping increases for an increasing sequence length, regardless of whether the requests are fixed-length or variable-length. Swapping has a noticeable edge for very large fixed sequence lengths. This edge diminishes in the case of variable sequence lengths thanks to less recomputation overheads, but nevertheless is noticeable in all configurations.

Variable number of submitted inference requests: vLLM



(a) Variable spare GPU capacity (Yarn-llama-2-7B model, 8K context window).

(b) Different LLM architectures.

Fig. 10: Overhead of recomputations vs. swapping for a variable spare GPU memory allocated for KV caching and different LLM architectures.

enqueues all received requests and batches them together based on a first come first served basis. This also impacts the KV caching behavior. To study this effect, we vary the number of requests simultaneously enqueued from 32 to 128. These requests are generated apriori using our synthetic workload generator and reused across all configurations (to enable a fair comparison). As expected, Fig. 9 shows an increasing overhead for an increasing number of enqueued requests due to an increasing number of evictions during batching. The increase in overhead is predictable (2x increase in number of enqueued requests results in at least 2x higher overhead). Again, swapping has a slight edge due to accumulated small differences in overheads over multiple evictions.

Variable spare GPU memory for KV caching: another important parameter that affects the number of evictions is the available spare GPU memory for KV caching. To study the impact of this parameter, we use several sizes: 6 GB, 10 GB, 14 GB, 18 GB, 22 GB, and 24 GB. To trigger a significant number of evictions, we generate 128 variable-length requests using our synthetic *Zipf* generator (with a prompt-to-output ratio of 0.5), which roughly follows the distribution of sequence lengths encountered in the SharedGPT benchmark. Like before, the requests are submitted simultaneously to vLLM. Fig 10a confirms the overhead of both recomputation and swapping decreases for an increasing spare capacity reserved for the KV cache, which is expected due to less evictions. Interesting to note is that swapping outperforms recomputations when the spare GPU capacity is scarce, an effect that can be explained by the fact that recomputations trigger prefill phases, which when they are not isolated incidents begin to cause scheduling issues due to load balancing, in addition to the individual recomputation overheads. The situation is reversed for large KV cache capacity, in which case recomputations outperform swapping.

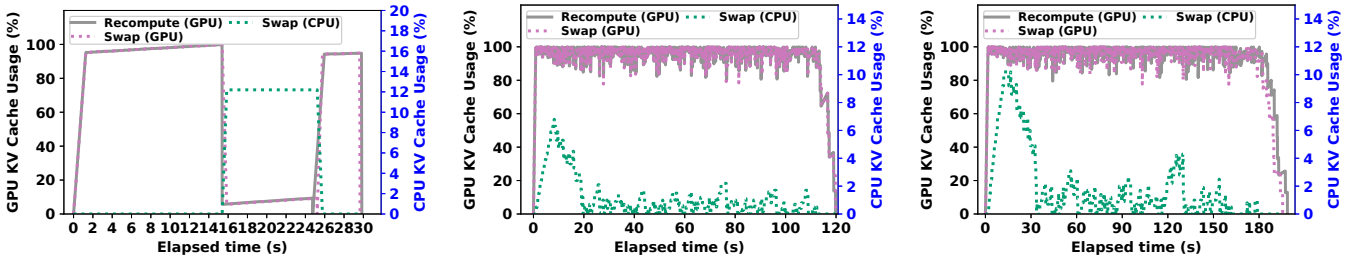
Variable transformer architecture: the transformer architecture has an impact not only on the sizes of the KV cache chunks (different sizes of K and V per token) but also on the duration of the forward passes. Both aspects directly impact the overhead of the swapping and recomputation strategies respectively. To understand this impact, we evaluate three different models: OPT-13B, Yarn-Llama-2-7B, and Llama-3.1-8B. The configuration of each model is described in Table I. During the experiment, we reserved 10 GB of GPU memory for the KV cache. Again, we generate 128 variable-length requests using our synthetic Zipf generator (the sequence length of each request is within [512, 2048]) and then send them to vLLM simultaneously. Fig.10b shows that OPT-13B has the highest overhead (e.g., about 10.4 seconds with recomputation and 7.2 seconds with swapping), while the Llama-3.1-8B model has the lowest overhead (e.g., 0.38 seconds with recomputation and 0.34 seconds with swapping). The higher overhead is expected since OPT-13B is the larger model and therefore has higher computational complexity and longer forward passes. Additionally, it triggers more frequent evictions compared with the other two LLMs. These two factors contribute to a clear advantage of swapping over recomputations in the case of OPT-13, while its advantage over recomputations is significantly diminished for the other two LLMs.

Key observations: Swapping strategies are not effective unless the allocated host memory is pinned, which reduces the opportunities for the OS to optimize the host memory management. Small KV cache chunk sizes result in many simultaneous evictions, especially for large inference requests, which reduces the host-GPU memory I/O transfer throughput and thus diminishes the effectiveness of swapping. However, large cache chunk sizes may result in under-utilization. Fine-tuning is necessary. Swapping keeps an edge over recomputations for most other variable parameters: number of evicted tokens, variable sequence length, number of simultaneously submitted requests. Swapping is better than recomputations when the spare GPU memory is scarce, but worse when the spare GPU memory capacity is large. The model architecture plays an important role, as it affects the sizes of cache chunks, duration of forward passes and number of triggered evictions. Overall, there are complex trade-offs that motivate the need for novel adaptive multi-level KV caching.

Study of KV cache utilization: to zoom on the findings discussed above, we selected three representative scenarios: (1) 16K evicted tokens (from the variable number of evicted tokens experiments); (2) 6 GB of spare GPU memory allocated to the KV cache (from the variable GPU spare capacity experiments); (3) OPT-13B model (from the different model architecture experiments). These scenarios represent extremes that underline the differences in behavior between the swapping vs. recomputation strategies.

Fig.11a showcases the GPU and CPU KV cache utilization over time for a single 16K token eviction. In this case, the GPU KV cache utilization rapidly increases to 95% within about 1.2 seconds (corresponding to the prefill phase), then slowly increases over time until the KV cache eviction is triggered (during decode phases). After the eviction (at 15 seconds), the GPU KV cache utilization significantly decreases for both recomputations and swapping. When the evicted request is resumed at 24.5 seconds, the GPU KV cache utilization increases for both recomputations and swapping, while the host cache utilization decreases for swapping. This clearly shows that swapping interrupts the inference process during both eviction and resumption due to blocking transfers between the GPU and host memory. In contrast, recomputations only interrupt the inference process during resumption.

Fig.11b illustrates the GPU and host cache utilization over time for a small spare GPU memory cache of 6 GB. The GPU cache utilization starts at nearly 100% and remains almost stable



(a) Zooming on Fig. 8b when number of evicted tokens = 16K.

(b) Zooming on Fig. 10a when GPU mem. for KV cache = 6 GB.

(c) Zooming on Fig. 10b when using **OPT-13B** model

Fig. 11: GPU and CPU KV cache usage over time when using recompute vs. swap approach by selecting three cases from the overhead experiment.

with a fluctuation between 80% and 100% until the end for both recomputations (grey curve) and swapping (dotted orchid curve). This small fluctuation is caused by constant KV cache evictions, but overall the utilization remains high as expected. For the swapping approach, the CPU KV cache shows a bursty increase (reaching 6% at the peak) at the beginning (within the first 20 seconds), followed by a drop to almost 0. This indicates that there is a significant amount of data movement between GPU memory and host memory in the initial stage. After the first 20 seconds, the host cache repeats a pattern of utilization between 0% to 2%. These fluctuations correspond to evictions, another confirmation of frequent evictions under scarce GPU memory.

Fig.11c showcases the GPU and host KV cache utilization over time for the OPT-13B model. We observe the GPU KV cache starts with nearly 100% utilization and remains almost stable with a fluctuation between 80% and 100% until the end for both recomputations (grey curve) and swapping (dotted orchid curve). Again, the fluctuations are caused by evictions. In the case of swapping, the host KV cache shows a bursty increase (reaching 10% at the peak) in the beginning (within the first 31 seconds), followed by a drop to almost 0. Again, this can be traced back to a significant amount of data movement between GPU and host memory in the initial stage. Similarly, after the first 31 seconds, the host cache utilization frequently increases and decreases within a range between 0% and 4%. This is also a confirmation of frequent evictions in the case of the OPT-13B model, though the amount of data transferred between GPU and host memory is significantly smaller than in the initial stage.

Key observations: Studying the KV cache utilization for both GPU and host memory is an important tool in understanding how evictions impact the effectiveness of KV caching. Synchronous data transfers between the host and GPU memory leave a clearly observable trace, with large data movements in the beginning that gradually become smaller later. Combined with the access pattern analysis in § VI-B, these patterns hint at opportunities to hide the overhead of GPU-host data transfers through asynchronous cache management techniques.

VII. Conclusions

Despite inferences being an integral part of modern LLM usage, there is a gap in the state-of-the-art with respect to profiling modern LLM inference frameworks that combine several optimizations in order to understand what parameters matter, how to fine-tune them, and how to improve important building blocks such as KV caching. To this end, we propose a series of design principles and a corresponding implementation as profiling tool that enables scalable instrumentation of inference frameworks to collect fine-grain metadata and performance metrics needed to characterize the behavior of the inferences under concurrency.

Using this tool, we instrumented vLLM, a popular inference framework. We characterize in-depth the behavior of vLLM under concurrency for a diversity of scenarios that combine several workloads, different model architectures, and several optimizations: different batching strategies (prefill first, mixed prefill and decode, chunked prefill) and different GPU KV cache eviction policies: swap to host memory, drop and recompute.

The results show interesting trade-offs that relate to load balancing of forward passes to avoid stragglers through chunking vs. overhead running multiple forward passes and sub-optimal implementation of attention computations when only a partial view is available due to chunking, flexibility of batching due to limited penalty of under-utilizing the context window during forward passes, differences in utilization of the context window when prefills dominate over decodes and the other way around, significant KV cache idle durations when accesses are missing or happen occasionally, swapping vs. recomputations.

In future work, we will extend our instrumentation to seamlessly integrate with lower-level tools such as NVIDIA NSight Systems. Additionally, we will leverage our findings to co-design novel batching and multi-level KV cache strategies that adaptively optimize the highlighted trade-offs. Finally, we didn't explore an important dimension: the use of multiple GPUs both within the same compute node (tensor parallel) and across compute nodes (data parallel). This presents opportunities to study new distributed patterns and corresponding optimization strategies.

Acknowledgment

This material is based upon work supported by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research, under Contracts DE-AC02-06CH11357. Additionally, this work is partially supported by the National Science Foundation (NSF), Office of Advanced Cyberinfrastructure, under Grants CSSI-2411318, Core-2313154, CSSI-2104013, and CSSI-2411386.

REFERENCES

[1] D. Tilwani, Y. Saxena, A. Mohammadi, E. Raff, A. Sheth, S. Parthasarathy, and M. Gaur, "Reasons: A benchmark for retrieval and automated citations of scientific sentences using public and proprietary llms," *arXiv preprint arXiv:2405.02228*, 2024.

[2] J. Gou, B. Yu, S. J. Maybank, and D. Tao, "Knowledge distillation: A survey," *International Journal of Computer Vision*, vol. 129, no. 6, pp. 1789–1819, 2021.

[3] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg *et al.*, "Sparks of artificial general intelligence: Early experiments with gpt-4," *arXiv preprint arXiv:2303.12712*, 2023.

[4] D. A. Boiko, R. MacKnight, B. Kline, and G. Gomes, "Autonomous chemical research with large language models," *Nature*, vol. 624, no. 7992, pp. 570–578, 2023.

[5] B. S. Manning, K. Zhu, and J. J. Horton, "Automated social science: Language models as scientist and subjects," National Bureau of Economic Research, Tech. Rep., 2024.

[6] G. Bhatt, Y. Chen, A. M. Das, J. Zhang, S. T. Truong, S. Mussmann, Y. Zhu, J. Bilmes, S. S. Du, K. Jamieson *et al.*, "An experimental design framework for label-efficient supervised finetuning of large language models," *arXiv preprint arXiv:2401.06692*, 2024.

[7] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[8] A. Q. Jiang, A. Sablayrolles, A. Roux, A. Mensch, B. Savary, C. Bamford, D. S. Chaplot, D. d. l. Casas, E. B. Hanna, F. Bressand *et al.*, "Mixtral of experts," *arXiv preprint arXiv:2401.04088*, 2024.

[9] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[10] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan *et al.*, "Deepseek-v3 technical report," *arXiv preprint arXiv:2412.19437*, 2024.

[11] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

[12] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, 2020, pp. 3505–3506.

[13] A. Maurya, J. Ye, M. M. Rafique, F. Cappello, and B. Nicolae, "Deep optimizer states: Towards scalable training of transformer models using interleaved offloading," in *Proceedings of the 25th International Middleware Conference*, 2024, pp. 404–416.

[14] D. Castro, "Rethinking concerns about ai's energy use," Center for Data Innovation, Jan 2024. [Online]. Available: https://www2.datainnovation.org/2024-ai-energy-use.pdf

[15] Mastering llm techniques: Inference optimization. [Online]. Available: https://developer.nvidia.com/blog/mastering-llm-techniques-inference-optimization/

[16] B. Wu, S. Liu, Y. Zhong, P. Sun, X. Liu, and X. Jin, "Loongserve: Efficiently serving long-context large language models with elastic sequence parallelism," *arXiv preprint arXiv:2404.09526*, 2024.

[17] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 611–626.

[18] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin *et al.*, "Opt: Open pre-trained transformer language models," *arXiv preprint arXiv:2205.01068*, 2022.

[19] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[20] A. Yang, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Li, D. Liu, F. Huang, H. Wei *et al.*, "Qwen2. 5 technical report," *arXiv preprint arXiv:2412.15115*, 2024.

[21] J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, F. Lebrón, and S. Sanghai, "Gqa: Training generalized multi-query transformer models from multi-head checkpoints," *arXiv preprint arXiv:2305.13245*, 2023.

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

[23] Nvidia triton dynamic batching. [Online]. Available: https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/user_guide/model_configuration.html#dynamic-batcher

[24] Faster transformer. [Online]. Available: https://github.com/NVIDIA/FasterTransformer

[25] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A distributed serving system for {Transformer-Based} generative models," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 521–538.

[26] A. Agrawal, N. Kedia, A. Panwar, J. Mohan, N. Kwatra, B. Gulavani, A. Tumanov, and R. Ramjee, "Taming {Throughput-Latency} tradeoff in {LLM} inference with {Sarathi-Serve}," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024, pp. 117–134.

[27] C. Holmes, M. Tanaka, M. Wyatt, A. A. Awan, J. Rasley, S. Rajbhandari, R. Y. Aminabadi, H. Qin, A. Bakhtiari, L. Kurilenko *et al.*, "Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference," *arXiv preprint arXiv:2401.08671*, 2024.

[28] B. Wu, Y. Zhong, Z. Zhang, S. Liu, F. Liu, Y. Sun, G. Huang, X. Liu, and X. Jin, "Fast distributed inference serving for large language models," 2024. [Online]. Available: https://arxiv.org/abs/2305.05920

[29] P. Patel, E. Choukse, C. Zhang, A. Shah, Íñigo Goiri, S. Maleki, and R. Bianchini, "Splitwise: Efficient generative llm inference using phase splitting," 2024. [Online]. Available: https://arxiv.org/abs/2311.18677

[30] Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, X. Jin, and H. Zhang, "Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving," 2024. [Online]. Available: https://arxiv.org/abs/2401.09670

[31] C. Hu, H. Huang, L. Xu, X. Chen, J. Xu, S. Chen, H. Feng, C. Wang, S. Wang, Y. Bao, N. Sun, and Y. Shan, "Inference without interference: Disaggregate llm inference for mixed downstream workloads," 2024. [Online]. Available: https://arxiv.org/abs/2401.11181

[32] C.-C. Huang, G. Jin, and J. Li, "Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1341–1355.

[33] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design," 2016. [Online]. Available: https://arxiv.org/abs/1602.08124

[34] X. Nie, X. Miao, Z. Yang, and B. Cui, "Tsplit: Fine-grained gpu memory management for efficient dnn training via tensor splitting," in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, 2022, pp. 2615–2628.

[35] Z. Zong, L. Lin, L. Lin, L. Wen, and Y. Sun, "Str: Hybrid tensor re-generation to break memory wall for dnn training," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 8, pp. 2403–2418, 2023.

[36] PyTorch Team, "Pytorch profiler," https://pytorch.org/tutorials/recipes/recipes/profiler.html, 2023, accessed: 2024-10-09.

[37] NVIDIA Corporation, "Nvidia nsight systems," https://developer.nvidia.com/nsight-systems, 2023, accessed: 2024-10-09.

[38] AMD Corporation, "Omnitrace: Amd's gpu profiler," https://github.com/ROCm/omnitrace/, 2023, accessed: 2024-10-09.

[39] E. D. Berger, S. Stern, and J. A. Pizzorno, "Triangulating python performance issues with Scalene," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. Boston, MA: USENIX Association, Jul. 2023, pp. 51–64. [Online]. Available: https://www.usenix.org/conference/osdi23/presentation/berger

[40] Sharegpt dataset. [Online]. Available: https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered/tree/main

[41] Y. Wang, Y. Chen, Z. Li, Z. Tang, R. Guo, X. Wang, Q. Wang, A. C. Zhou, and X. Chu, "Towards efficient and reliable llm serving: A real-world workload study," *arXiv preprint arXiv:2401.17644*, 2024.

[42] L. Yu and J. Li, "Stateful large language model serving with pensieve," *arXiv preprint arXiv:2312.05516*, 2023.

[43] C. Hu, H. Huang, J. Hu, J. Xu, X. Chen, T. Xie, C. Wang, S. Wang, Y. Bao, N. Sun *et al.*, "Memserve: Context caching for disaggregated llm serving with elastic memory pool," *arXiv preprint arXiv:2406.17565*, 2024.

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