SpvLLM: Efficient KV Cache Sparsification in vLLM

Yao Xiao
Harvard University
yaoxiao@g.harvard.edu

Dagim Gebrie
Harvard University
dagimgebrie@college.harvard.edu

Abstract

The accelerated scaling of large language models (LLMs) requires efficient memory management techniques. vLLM reduces memory fragmentation in LLM inference by proposing a PagedAttention mechanism whereby a virtual memory similar to the ones used in operating systems memory paging is used. However, as LLMs handle longer contexts and generate longer sequences, the KV cache still occupies a substantial amount of memory. To optimize memory usage, KV cache sparsification techniques are employed to evict less important tokens based on attention scores. The eviction leads to internal fragmentation which negatively impacts performance. Freeing blocks whose slots are all evicted alleviates the problem but still suffers from internal fragmentation. Copying non-evicted KV cache to newly allocated memory blocks solves the fragmentation issue, but incurs copying overhead that further leads to preemption, degrading its performance. In this paper, we propose SpvLLM that not only frees fully deactivated blocks, but also reuses freed slots of partially deactivated blocks, reducing internal fragmentation by up to 55.7% and achieving up to 2.21× higher end-to-end throughput and 48.9% lower latency.

Keywords

LLM Inference, KV Cache Sparsification, vLLM

1 Introduction

vLLM is a is a distributed serving system for LLM inference. To mitigate memory fragmentation, it divides the key-value (KV) cache into blocks of fixed size and memory is allocated per block instead of the whole sequence at once. This handles the complexity associated with variable size memory pieces due to the variable length outputs of LLMs. The usage of fixed-size blocks is particularly important because it reduces the granularity of allocating memory, minimizing the external fragmentations caused by oversubscribing memory for LLM outputs. Moreover, there is no internal fragmentation except in the final block which may not be fully filled, which is minimal.

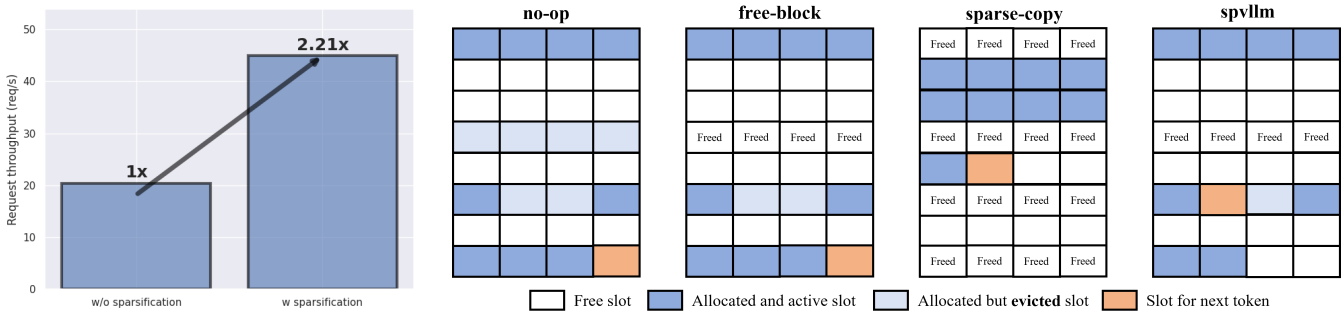
However, the case is different when KV cache sparsification comes in to manage growing context and output lengths. Keeping all tokens in the KV cache wastes a substantial amount of memory. 1a shows the improvement in throughput (tokens/s) achieved through sparsification compared to the raw approach. KV cache sparsification selectively removes tokens from the KV cache based on specific criteria such as low cumulative attention scores to free up memory. Some methods such as SnapKV [4] focus only on long contexts and perform KV cache eviction only during the prefill phase. This makes memory reuse very straightforward since the cache has not been changing dynamically yet. On the other hand, other techniques such as StreamingLLM [8] and H2O [9] perform eviction throughout the entire process, but they break the memory management of vLLM by introducing internal fragmentations in all blocks instead of only the final one. Internal fragmentation is problematic because it increases memory waste and compromises the ability of our system to process more data faster (higher throughput) and respond quicker (lower latency). We investigate the following strawmans and propose our solution SpvLLM.

Strawman 0 (*no-op*): This is a baseline strategy of KV cache sparsification where tokens are evicted based on some sparsification criteria, but the memory associated with the evicted tokens is not reclaimed. The free slots remain inactive and nothing is done to reorganize memory usage.

Strawman 1 (*free-block*): When evicting a single token, the block it belongs to can be freed for reuse only when all tokens in that block are freed. This strawman does not introduce extra overhead, but the freeing of full blocks is not guaranteed to happen and preserved tokens likely scatter all over the KV cache, so it still suffers from significant internal fragmentation.

Strawman 2 (*sparse-copy*): Whenever an eviction happens, allocate sufficiently many new KV blocks and copy only the preserved tokens in the original KV cache to the new KV blocks. This mitigates internal fragmentation except in the final block (same as the original vLLM), but incurs copying overhead especially for large KV caches and when eviction happens frequently. In particular, additional blocks must be pre-allocated for sparse copying, which can further cause preemption due to non-sufficient memory for such pre-allocation. In the worst case where eviction happens per iteration (which is the default of StreamingLLM [8] and H2O [9]), this overhead can lead to serious performance degradation.

Solution (SpvLLM): We propose that freed slots of evicted tokens can be reused even in cases that full blocks cannot be



(a) (b)

Figure 1: (a) Improvement in request throughput (req/s) with KV cache sparsification, showing up to 2.21× performance boost compared to the original vLLM without sparsification. (b) Illustration of how KV cache sparsification is managed by different strategies. *no-op* leaves evicted slots unused. *free-block* frees fully evicted blocks but leaves evicted slots unused. *sparse-copy* copies all preserved slots to newly allocated blocks. SpvLLM is able to reuse evicted slots to reduce internal fragmentation.

freed. In particular, we observe that the order of tokens in the KV cache does not matter. The scaled dot-product attention computation can be formulated as follows:

$$q_i = W_Q x_i, \quad k_i = W_K x_i, \quad v_i = W_V x_i,$$
$$o_i = \sum_{j=1}^{i} \left(\frac{\exp\left(\frac{q_i^\top k_j}{\sqrt{H}}\right)}{\sum_{t=1}^{i} \exp\left(\frac{q_i^\top k_t}{\sqrt{H}}\right)} \right) v_j,$$

where:

- q_i is the query vector for token i,
- k_i is the key vector for token i,
- v_i is the value vector for token i,
- W_Q, W_K, and W_V are weight matrices for queries, keys, and values, respectively,
- H is the size of the hidden dimension, which is the scaling factor here.

It is obvious from the formulation above that attention computation is both agnostic of key cache ordering (because of the inner sum) and of value cache ordering (because of the outer sum). Hence, newly generated tokens can directly be filled into the freed slots (if any) instead of allocating new slots to mitigate internal fragmentation issues caused by KV cache sparsification. In the simplest case of performing KV cache sparsification per decoding step and evicting one token per sparsification, this fills up the fragmented space immediately after each eviction, leading to minimal internal fragmentation at the same level as the original vLLM. This also does not involve copying, incurring minimal runtime overhead.

2 Design

In this section, we will introduce the high-level design of our system on top of the vLLM architecture.

Attention score retrieval: Most KV cache sparsification strategies are based on attention scores, i.e., $QK^\top$. However, attention scores are not accessible in the original vLLM system. SpvLLM collects the attention scores from the temporary logits storing partial qk results in the PagedAttention kernel. Note, however, that this is not possible for certain attention backends such as FlashAttention.

Block masks: The original vLLM system do not have a masking mechanism, but for *no-op*, *free-block*, and SpvLLM, tokens at certain positions of the blocks need to be masked once they are evicted to avoid contributing to the attention output, retaining the correct computation result. This is achieved by extending the vLLM block manager to keep track of additional boolean masks on its blocks, marking whether each slot in a block has been evicted or not. This information is then passed down to the PagedAttention kernel, which will forcefully set the qk values at the masked slots to $-\infty$. This will diminish after the scaled softmax operation for obtaining attention weights, thus not contributing to the final attention output.

Sparse copying kernel: vLLM can copy full blocks of KV cache due to its CoW (copy-on-write) mechanism, but the *sparse-copy* strategy requires copying at per-slot granularity. In order to minimize the impact of sparse copying operation itself on performance, we implement a fast sparse copying kernel parallelized on a 3D grid:

(1) **num_layers:** The number of attention layers.
(2) **num_seqs:** The number of sequences in the current batch to be processed.
(3) **num_blocks** × **block_size:** The block dimension, i.e., the number of slots involved in sparse copying.

Source-to-destination block mapping and slot mapping are constructed and wrapped into tensors in advance for the

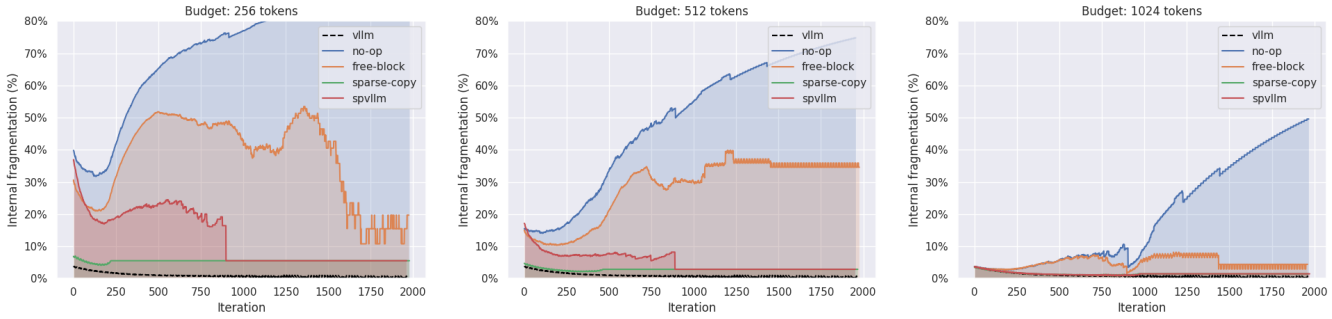


Figure 2: Internal fragmentation over the iterations for KV cache management strategies with token budgets of 256, 512, and 1024. The black dashed line represents the original vLLM without KV cache sparsification.

kernel, with −1 marking slots that will not be copied over to the destination.

Others: There are various assumptions in the original vLLM system that will be broken by KV cache sparsification.

- It is not true that the number of blocks occupied by a sequence can be directly inferred from its length.
- It is not true that the position ID of each new token is the current length of its sequence.
- It is not true that the KV cache of the new token will fill the last logical block occupied by its sequence, because SpvLLM may reuse previously deactivated slots instead of allocating new slots (blocks).

Every part of the original vLLM system that relies on these assumptions must be handled carefully.

3 Evaluation

In this section, we evaluate the performance of different memory management strategies, i.e., *no-op*, *free-block*, *sparse-copy*, and SpvLLM, under KV cache sparsification. We will perform micro-benchmarks on internal fragmentation (§3.2) and sparse copying kernel overhead (§3.3), as well as end-to-end performance evaluation (§3.4).

3.1 Experimental Setup

The implementation of the KV cache sparsification framework involved approximately 2000 lines of additional C++, CUDA, and Python code on top of vLLM 0.6.2. These changes were evaluated on a single instance with one NVIDIA Tesla T4 GPU (16GP GDDR6 memory) with a dataset of 1000 randomly sampled single-round conversations from the ShareGPT dataset [7], a collection of user-shared real-world conversations with ChatGPT. The average input length of the sampled dataset is 224.7 tokens the average output length is 208.9 tokens. All 1000 requests arrive at the server at the same time to exhaust system resources. The attention backend used for testing is xFormers [3]. It is slightly slower than FlashAttention (the default of vLLM), but it is impossible to extract attention scores with the FlashAttention backend. The KV cache sparsification mechanism is H2O [9], evicting one token per sequence per decode iteration. The testing framework disables CPU cache offloading of vLLM, meaning that requests need to be recomputed upon preemption.

Implementation notes: Due to limited time, we did not implement a computationally correct version of SpvLLM, but with the correct memory pattern. All strawman strategies, i.e., no-op, free-block, and sparse-copy, are fully implemented. We also did not figure out the reason for low KV blocks utilization, so we normalized our benchmarking results by mean KV blocks utilization to simulate that 100% *of KV blocks are exhausted. This may not be accurate since we cannot ensure linear relation.*

3.2 Internal Fragmentation

Internal fragmentation is measured by the number of active slots, i.e., not evicted, divided by the total number of slots in the KV blocks occupied by each sequence. The larger internal fragmentation, the lower GPU memory utilization. From Figure 2, we can see that the original vLLM without KV cache sparsification has negligible internal fragmentation, which only comes from the last block of each sequence. With *no-op*, internal fragmentation grows continuously because no evicted slots can be reused. *free-block* reduces fragmentation by freeing KV blocks that are fully deactivated. SpvLLM further improves by reusing deactivated slots. With the setup of evicting one token per iteration, SpvLLM will simply be refilling the one evicted slot per iteration once reaching the budget, thus no longer introducing fragmentation. *sparse-copy* has the same level of internal fragmentation as the original vLLM since it fills contiguously the KV blocks by copying and falls back to the non-sparsified scenario.

Apart from the over-time analysis of internal fragmentation, Figure 3 provides a snapshot of peak internal fragmentation at 99th percentile across the different strategies. Among *no-op*, *free-block*, and SpvLLM, SpvLLM has the lowest peak internal fragmentation across all different token budgets, by up 21.2% less than *free-block* and 55.7% less than

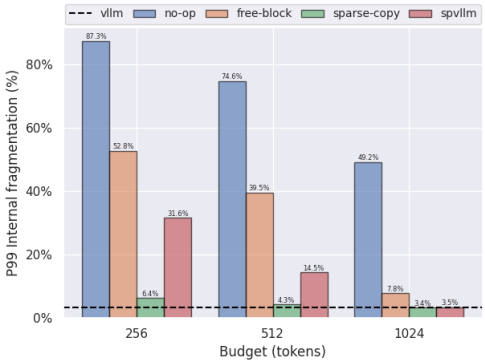


Figure 3: 99th percentile internal fragmentation.

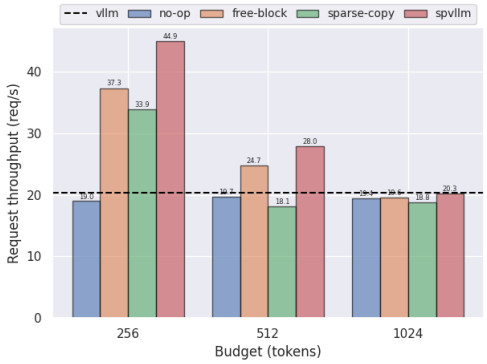


Figure 4: Request throughput (requests per second).

no-op when limiting to 256 tokens. This is aligned with our previous observation. *sparse-copy* is consistently at the same level as without KV cache sparsification (black dashed baseline). Moreover, all KV cache management strategies have lower internal fragmentation as token budget increases, because fewer tokens need to be evicted and the chance for fragmentation to occur is lower.

3.3 Sparse Copying Kernel Overhead

This micro-benchmark measures the total overhead incurred by launching the sparse copying kernel over time, using the *sparse-copy* KV cache management strategy.

Budget	# Copies	Total overhead (ms)
256	2007	179.14
512	2007	155.03
1024	2004	162.01

Table 1: Copying overhead for the copy strategy remains consistent and negligible across all token budgets.

From Table 1, we can see that the overhead introduced directly by the sparse copying kernel is almost negligible, being less than 1 second in a total 2 to 3 minutes' duration, independent of the token budget. However, this does not mean that the overall overhead of the *sparse-copy* strategy is low. With the most intuitive implementation, destination blocks must be pre-allocated when the source blocks cannot yet be freed, which may waste a significant amount of memory and lead to preemptions. This micro-benchmark only implies that the overhead of the kernel execution alone will make nearly no negative contributions to the performance.

3.4 End-to-End Performance

Eventually, we perform end-to-end benchmarking to check the impact of different KV cache management strategies and different token budgets on end-to-end throughput and latency when serving on our dataset.

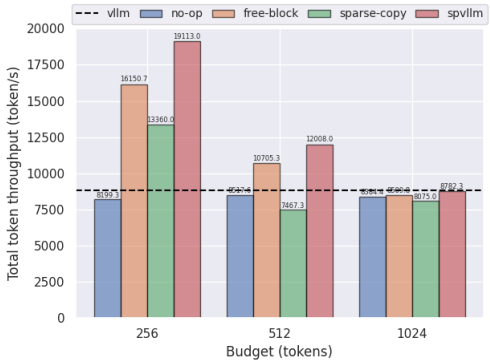


Figure 5: Total token throughput (number of tokens processed per second).

Throughput: From Figure 4, we can see that larger KV cache budget leads to lower throughput in general. Though there is less need for sparsification, larger budgets means larger KV cache per sequence, leading to fewer sequences that can be batch processed together and thus lower throughput. Focusing on different KV cache management strategies, we see that SpvLLM shines as the strategy with the highest end-to-end throughput, followed by *free-block*, achieving up to 2.21× higher throughput compared with no KV cache sparsification with a budget of 256 tokens. All strategies start showing closer performance for moderate (512 tokens) and larger (1024 tokens) budgets. We see similar results from Figure 5, which measures the throughput as the total number of tokens processed per second.

One might expect *free-block* to show poor performance at smaller batches due to frequent fragmentation, but it is important to note that in these cases batch size dominates the relationship. Impacts of fragmentation and copying are still visible, such as the low performance of the no-op due to frequent fragmentation. For *sparse-copy*, we showed in §3.3 that the sparse copying kernel has an almost negligible impact on performance, but it is not performing the best

Figure 6: 99th percentile time-to-first-token (TTFT), time-per-output-token (TPOT), and inter-token latency (ITL).

among the strategies despite its lowest fragmentation. The reason is that, within the vLLM framework, the scheduler must decide the memory plan before actual execution takes place. This means that destination blocks need to be pre-allocated before source blocks are freed for further reuse, when sparse copying is to happen. This is not considered internal fragmentation, but is another form of memory waste that degrades end-to-end performance. Moreover, we observer that with *sparse-copy* there is approximately 19.5% requests that have been preempted, compared to 0 for all the other strategies. A preemption, due to disabled CPU cache offloading mechanism, means recomputing the whole request, also explaining the unexpectedly low performance of *sparse-copy*.

Why *sparse-copy* causes preemption, not the others: This is mostly because the provision is correct for the other strategies. Provisioning means estimating and scheduling a suitable number of requests per batch. It is non-trivial, when a request is first scheduled, to provision when a sparse copying may happen. It is thus likely that too many requests have been scheduled to run, leading to lack of space to sparse copy to for certain running requests. Consequently, those requests would have to be preempted for recomputation.

Latency: Figure 6 demonstrates the end-to-end latency metrics. We see negligible differences in TTFT, because KV cache sparsification makes hardly any difference in the prefill phase (i.e., when generating the first token). For TPOT and ITL, due to normalization (as mentioned in the implementation note at the end of §3.1) and the likely non-linear relationship between TPOP/ITL and KV blocks utilization, the strategies that have close performance may not be reflecting the correct latency results. Regardless, it shows an overall inverse pattern as end-to-end throughput, which is as expected. SpvLLM achieves up to 17.6% lower time-per-output-token and 48.9% lower inter-token latency than the baseline of no KV cache sparsification.

4 Related Work

4.1 PagedAttention and vLLM

vLLM is a high-throughput, memory-efficient serving framework for LLM inference. It is a state-of-the-art system for LLM inference and a widely acknowledged standard in both industry and academia. It introduces the PagedAttention mechanism that allows for optimized memory usage during inference. Traditional frameworks for serving LLMs often require significant memory overhead, limiting the model's ability to scale efficiently. vLLM addresses this by managing memory allocation dynamically, which not only reduces fragmentation but also enables models to handle longer contexts and larger batches. As a result, vLLM can achieve higher throughput compared to conventional serving solutions while still maintaining low latency.

4.2 KV Cache Sparsification

Across the literature, various KV cache sparsification techniques have emerged to manage memory efficiently in large language models by evicting less important tokens from the KV cache. These methods use different methods to determine which tokens to evict, and are applied in different LLM inference systems (but none of them are integrated with the memory management mechanism of vLLM). Some of these KV cache sparsification methods are as follows:

StreamingLLM [8]: StreamingLLM manages memory dynamically by using a rolling cache KV cache. On top of the a sliding window, it identifies attention sinks, which refer to early tokens that receive high attention scores, and preserve them in the KV cache. Cumulative attention scores are computed across layers for each token and those with the lowest score are evicted.

SnapKV [4]: SnapKV performs a single KV cache eviction in the prefill phase by observing that important tokens remain important throughout the inference. It takes a snapshot of the prompt and sum attention scores across layers within a certain window. A voting mechanism is applied and only

tokens that consistently show high score are retained. However, due to not evicting dynamically in the decode phase, SnapKV handles only long contexts but not long outputs.

H2O [9]: H2O, or the Heavy-Hitter Oracle, identifies tokens with high cumulative attention scores in recent decoding steps. Per eviction, it greedily discards a token with the lowest cumulative attention score among all tokens that are generated more than k steps earlier. This greedy eviction decision allows H2O to quickly identify critical tokens without looking ahead to future tokens. By default, H2O performs an eviction per decode iteration and evicts one token each time.

PyramidKV [1]: PyramidKV distinguishes token importance per layer. It identifies that different layers can have different KV cache budgets, thus allowing even higher level of sparsification than other methods like H2O. In particular, tokens are ranked based on their attention scores per layer, and again only highest-ranking tokens are retained in the KV cache. Lower layers are given more KV cache budget as they capture more foundational information, and higher layers are given less KV cache budget. Per eviction, not only low-ranking tokens are evicted, but those that exceed the layer's allocated budget are also removed.

More: There are many more other KV cache sparsification mechanisms, such as ScissorHands [5], Adaptive Compression [2], Dynamic Memory Compression [6], etc.They are more or less similar to the aforementioned strategies and will not be detailed here.

5 Conclusion

The division of the KV cache into blocks of fixed-size by vLLM minimizes memory waste in between allocations, reducing fragmentation. It does not have built-in support for KV cache sparsification, and we investigated different KV cache management strategies when integrating KV cache sparsification in vLLM. Our paper demonstrated that strategies such as *no-op*, where the memory associated with evicted tokens is not reclaimed and free slots remain inactive, suffer from high internal fragmentation which leads to degraded performance.

Allocating sufficiently many new KV cache blocks and sparse copying only the preserved tokens from the original KV cache to the new blocks, as in the *sparse-copy* strategy, incurred negligible overhead on the actual copying operation (done by a dedicated CUDA kernel), but still showed lower than expected end-to-end throughput. We identified that the need for pre-allocating destination blocks before being able to free source blocks causes a different kind of memory waste, and also leads to preemption which is the culprit of low performance.

The *free-block* strategy is a net improvement on top of *no-op*, and same for SpvLLM on top of *free-block*. Indeed we see significant reduction in internal fragmentation with SpvLLM, as well as up to 2.21× higher end-to-end throughput and up to 48.9% lower end-to-end latency.

As future work, we will correct the implementation SpvLLM computationally, and we will figure out the reason for low KV block utilization and mitigate the need for normalization on the end-to-end performance metrics. The *sparse-copy* strategy also has room for improvement, with better provisioning and memory planning.

6 Acknowledgement

We would like to thank Professor Minlan Yu, for her invaluable guidance and mentorship.

References

[1] Zefan Cai., Yichi Zhang, Bofei Gao, Yuliang Liu, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Baobao Chang, Junjie Hu, and Wen Xiao. 2024. PyramidKV: Dynamic KV Cache Compression based on Pyramidal Information Funneling. https://doi.org/10.48550/ARXIV.2406.02069

[2] Suyu Ge, Yunan Zhang, Liyuan Liu, Minjia Zhang, Jiawei Han, and Jianfeng Gao. 2023. Model Tells You What to Discard: Adaptive KV Cache Compression for LLMs. https://doi.org/10.48550/ARXIV.2310.01801

[3] Benjamin Lefaudeux, Francisco Massa, Diana Liskovich, Wenhan Xiong, Vittorio Caggiano, Sean Naren, Min Xu, Jieru Hu, Marta Tintore, Susan Zhang, Patrick Labatut, Daniel Haziza, Luca Wehrstedt, Jeremy Reizenstein, and Grigory Sizov. 2022. xFormers: A modular and hackable Transformer modelling library. https://github.com/facebookresearch/xformers

[4] Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. 2024. SnapKV: LLM Knows What You are Looking for Before Generation. https://doi.org/10.48550/ARXIV.2404.14469

[5] Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. 2023. Scissorhands: Exploiting the Persistence of Importance Hypothesis for LLM KV Cache Compression at Test Time. https://doi.org/10.48550/ARXIV.2305.17118

[6] Piotr Nawrot, Adrian Łańcucki, Marcin Chochowski, David Tarjan, and Edoardo M. Ponti. 2024. Dynamic Memory Compression: Retrofitting LLMs for Accelerated Inference. (2024). https://doi.org/10.48550/ARXIV.2403.09636

[7] ShareGPT Team. 2023. ShareGPT. https://sharegpt.com/

[8] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2023. Efficient Streaming Language Models with Attention Sinks. https://doi.org/10.48550/ARXIV.2309.17453

[9] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang Wang, and Beidi Chen. 2023. H$_2$O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models. https://doi.org/10.48550/ARXIV.2306.14048