EQUIP: EQUivariant preserving In-Place updates for Efficient token-pruning

Arun Ramachandran[1,2], R. Govindarajan[2], Murali Annavaram[3], Prakash Raghavendra[1]

[1]AMD India Private Limited, [2]Indian Institute of Science, [3]University of Southern California

aruncoimbatore.ramachandran@amd.com, govind@iisc.ac.in, annavara@usc.edu, prakash.raghavendra@amd.com

Abstract

Token-pruning has emerged as a primary focus in large language models (LLMs) to enhance model efficiency while preserving accuracy, especially for large sequence lengths. However, the eviction operation of token-pruning methods causes "holes" in KV tensors, posing two major challenges: (1) The shift operation, required to make the KV tensor contiguous, results in significant copy overheads; (2) The changes in position indices due to token eviction lead to increased computational requirements for Rotary Positional Encoding (RoPE). To address these issues, we introduce *EQUIP*, an EQUivariant preserving in-place token update mechanism that ensures the equivariance property of the operations performed in the attention computation. *EQUIP* offers two fundamental advantages: First, it combines eviction and a subsequent token insertion into an in-place replacement operation, which reduces the KV cache copy overheads significantly. Second, *EQUIP* reduces recomputation of rotation operations through a combination of in-place update, caching and a re-indexing strategy. Together, these optimizations enable EQUIP to achieve geomean speedups of 1.62× (or 1.47×) on CPU (GPU) over StreamingLLM, and 3.45× (or 1.86×) on CPU (GPU) over Heavy Hitters (H2O). EQUIP with Paged Attention achieves speedups of 4.18×(2.61×) on CPU (GPU) over auto-regressive baselines. *EQUIP* matches the model accuracy of baseline pruning methods while delivering superior performance.

1 Introduction

As Large Language Models (LLMs) (Zhu et al., 2023; Wang et al., 2024; Park et al., 2024; Tang et al., 2024) grow in size and complexity (Gemini Team and Google, 2023; Shoeybi et al., 2019), their computational resources and energy consumption have escalated. At the same time, the demand for processing long sequence lengths is also increasing. Processing large contexts requires allocation and management of large key-value (KV) caches, further increasing the computational and storage demands of LLMs.

These challenges have driven research towards optimization techniques such as token-pruning techniques (Zhang et al., 2023; Xiao et al., 2024; Li et al., 2024a; Yang et al., 2024; NVIDIA, 2025i; Feng et al., 2025; Zhang et al., 2024), which filter out less salient tokens. Pruning in turn reduces the computational and memory overheads in KV cache updates, enabling longer decode lengths while maintaining accuracy. Further, the reduced and fixed KV cache size (independent of the sequence length) allows for larger batch sizes to be processed, resulting in significant throughput improvement as demonstrated in prior works (Xiao et al., 2024; Zhang et al., 2023). While these token-pruning methods achieve considerable performance gains over baseline methods, they still leave significant performance improvement opportunities on the plate.

Two major operations in KV cache management performed by token-pruning methods are eviction (of less salient tokens) and insertion of new tokens. Eviction and insertion are typically implemented, respectively, as a KV cache copy operation (with KV cache elements shifted in position due to eviction) and an append operation (for the new token that is inserted). The shift and append operations need repeated memory copy operations which are inefficient. In this context, we make an important observation that in-place update operation preserves equivariance over attention computation operations (e.g., Batch Matrix Multiplication, softmax and Batch Matrix Multiplication operations). This is similar to the finding in (Kondor and Trivedi, 2018) that convolution operation is equivariant to translation. Based on this observation, EQUIP proposes to replace the expensive shift and append operations with in-place update allowing equivariance property to preserve the computational output.

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 26303–26323

EQUIP implementation achieves the same accuracy as that of the original token-pruning methods, while removing much of their overheads. Further, we note that a key strength of *EQUIP* lies in its broader applicability as identified below.

1. Compatibility with Contemporary Architectures: Modern architectures commonly use positional encodings such as Rotary Positional Encoding (RoPE) (Su et al., 2021) to model word order. In token-pruning methods (Xiao et al., 2024; Zhang et al., 2023), token eviction alters the positional indices of past tokens, forcing RoPE to be recomputed for all affected positions and thereby making positional encoding a major source of inference latency. RoPE computations naturally decompose into static rotated keys (RK) and dynamic position IDs. Existing designs typically recompute RK for simplicity and to avoid shift-and-append operations on RK, resulting in additional computational overhead. To fully exploit the benefits of token-pruning methods while retaining the advantages of RoPE, we introduce three optimizations in *EQUIP*: (1) caching RK tensors, (2) update the new RK value in place of evicted tokens, and (3) introduce an inexpensive positional reindexing scheme that preserves the same accuracy as the original RoPE embeddings.

2. Applicability to Custom Kernels Custom kernels, such as Paged Attention (Kwon et al., 2023), frequently encounter challenges in supporting popular token-pruning techniques, such as StreamingLLM (Xiao et al., 2024) or H2O (Zhang et al., 2023) due to limitations such as the non-contiguity in the KV cache allocation (vLLM Project, 2025d,e). We demonstrate that these limitations and scalability obstacles can be effectively overcome using our *EQUIP* approach, which converts evict and insert operations into efficient in-place update operations.

3. Applicability to Different Token-Pruning Methods: *EQUIP* supports both static and dynamic policies and can accommodate variable token budgets across layers and handle both contiguous (sliding-window) (Xiao et al., 2024) and random (data-driven) eviction patterns (Zhang et al., 2023). Together, these configurations span the practical design space of adaptive and hybrid token-pruning schemes, allowing the proposed approach to integrate with and extend several state-of-the-art methods (Li et al., 2024a; Yang et al., 2024).

4. Applicability in Multi-Token Eviction and Insertion: *EQUIP* extends naturally to multi-token evictions/updates. This enables supporting speculative decoding (Xia et al., 2024) and chunked updates in real-time document editing (He et al., 2024b). In speculative decoding, multiple tokens are generated speculatively using simpler models, which are then validated in parallel using the original model, often accepting more than one token in each iteration, thereby requiring simultaneous evictions and insertions. Similarly, real-time document or code edits in LLMs frequently involve multiple deletions and insertions. In these applications, our *EQUIP* approach can be effectively deployed to achieve higher computational efficiency by reducing KV cache copy overhead.

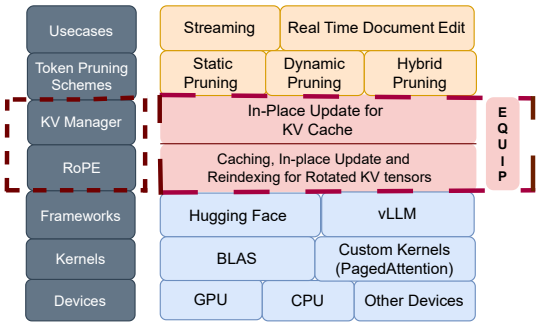


Figure 1: Full-stack enablement with *EQUIP*.

The above discussion on broader applicability of *EQUIP* establishes its effectiveness in the full-stack enablement of diverse token-pruning schemes across different LLM implementations and GeMM kernels, as illustrated in Figure 1.

We conduct experimental evaluations to study the impact of *EQUIP* on the end-to-end inference speedup of different pruning methods (StreamingLLM (Xiao et al., 2024) and Heavy Hitters (Zhang et al., 2023)) on two different LLM inference engines (Hugging Face (HuggingFace, 2024) and vLLM (Kwon et al., 2023)). Our evaluation demonstrates consistent speedup gains of *EQUIP* across diverse LLM models and hardware. Performance evaluation with Intel CPUs and AMD[1], Instinct™ MI210 GPUs demonstrates that *EQUIP* achieves a geomean end-to-end inference speedup of 1.62× on CPUs, and 1.47× on GPU over a strong baseline, namely StreamingLLM (Xiao et al., 2024). Additionally, *EQUIP* achieves a speedup of 1.86× over Heavy Hitters (Zhang et al., 2023) and 2.61× speedup over Paged Attention kernels (Kwon et al., 2023) on GPUs.

[1]AMD, AMD Instinct™ MI210, and AMD ROCm™ are trademarks of Advanced Micro Devices, Inc. © 2026 Advanced Micro Devices, Inc.

2 Background

2.1 Attention Computation

An LLM consists of a sequence of L layers, each layer comprising Attention and Multi-Layer Perceptron (MLP) Blocks. The attention computation starts with the current token x, and computes the query, key and value vectors as:

$$Q = W_q \cdot x; \quad K = W_k \cdot x; \quad V = W_v \cdot x$$

where W_q, W_k, and W_v are weight matrices.

For performing the attention operation (Shazeer, 2019) for a batch size B and across L layers, the K and V matrices become tensors of dimension $[B, L, n, D]$, where n is the current context length (this includes the prompt length along with the total tokens generated so far) and D is the total number of hidden dimensions. For a maximum context length of N, these tensors can be as large as $[B, L, N, D]$. In multi-head attention, the model's hidden dimension (D), is processed using H parallel attention heads, each of size d. Additional details are provided in appendix A.

2.2 Token-pruning Methods

Deploying LLMs in streaming applications such as multi-round dialogue (Zhang et al., 2025a) results in higher memory consumption and copy overhead as the size of KV cache grows in each round. Also, LLMs often exhibit limited generalization capability for texts that exceed pretrained sequence lengths. To address these challenges, several recent studies (Li et al., 2024a) (Xiao et al., 2024) (NVIDIA, 2025i) (Feng et al., 2025) (Yang et al., 2024) (NVIDIA, 2025b) (Zhang et al., 2024) (Xiao et al., 2025) have focused on identifying a critical subset of tokens that influence output predictions in each iteration and retaining only those tokens in the KV Cache.

KV cache management in StreamingLLM (Xiao et al., 2024) retains the first s tokens (attention sink) and the most recent r tokens. After the initial $(s+r)$ tokens, generation of each new token requires the eviction of the oldest token (from r recent tokens) and appending the new token. The compaction of KV cache (illustrated in Figure 5(a)) is implemented as a shift-and-append operation that copies the entire K and V tensors, which incurs substantial data movement. Further, implementing eviction across multiple heads requires scatter and gather operations (Zhang et al., 2023) which increases the complexity and associated overheads.

2.3 Rotary Positional Encoding (RoPE)

RoPE operates by applying position-dependent rotations to subspaces of the query (Q) and key (K) vectors before the attention computation. This mechanism effectively encodes both the absolute position of tokens and their relative distances within the sequence. The resulting RoPE-enhanced tensors subsequently serve as inputs to the Scaled Dot-Product Attention (SDPA) mechanism.

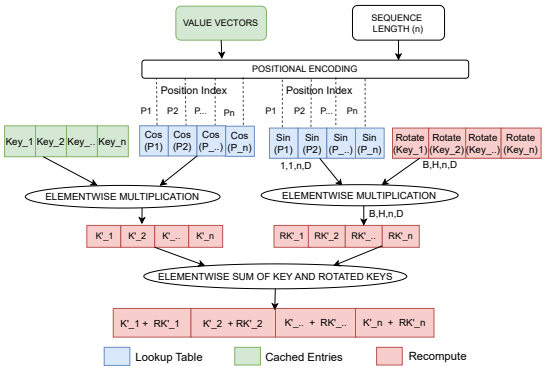


Figure 2: Rotary position embeddings (RoPE).

RoPE rotational transformation is typically achieved via an operation of the form $k'_p = k_p \odot \cos(p\theta) + \text{Rotate\_Half}(k_p) \odot \sin(p\theta)$, where $\odot$ denotes element-wise multiplication, p is the position index of the token and Rotate_Half(k_p) permutes feature pairs within k_p (details given in Appendix A.2). Figure 2 illustrates the RoPE computation. `Rotate_Half` values, multiplied element-wise by $\sin(p\theta)$, yield RK values $(\text{RK}_1, \text{RK}_2, \ldots, \text{RK}_n)$. The key values are cached in the KV cache. The sinusoidal values corresponding to sequence positions are precomputed and stored in a lookup table, and hence do not incur any computation overhead during decode. The Rotate_Half values are computed for each new token. Implementations differ in caching or not caching these Rotate_Half values; the latter version requires the Rotate_Half values to be recomputed incurring significant computational overhead.

Token eviction in the KV cache disrupts the positional integrity of tokens. This necessitates the recomputation of positional embeddings $(k_p \odot \cos(p\theta)$ and Rotate_Half$(k_p) \odot \sin(p\theta))$ to maintain the correct mapping between token positions and their corresponding embeddings. Token pruning methods (Xiao et al., 2024) typically do not cache the results of the Rotate_Half, and hence their positional embeddings must be recalculated

for the shifted tokens, leading to increased computational cost. We quantify this overhead in section 3. Caching the output of the Rotate_Half operation, on the other hand, requires a shift-and-append operation on the cached Rotate_Half value during token eviction, resulting in copy overhead.

3 Motivation

We identify two significant bottlenecks associated with token-pruning: (1) memory copy overhead related to KV cache Management and (2) additional computational requirements associated with RoPE.

3.1 KV cache Management Efficiency

Figure 3 reports the normalized end-to-end latency of tokens as the sequence length increases[2] for three different implementations, namely Hugging Face, StreamingLLM, and the proposed *EQUIP*. In the first two schemes, the KV cache is copied from one iteration to the next. For the token-pruning approaches (StreamingLLM and *EQUIP*), we have used $s = 4$ and $r = 252$. While the end-to-end latency increases linearly with the sequence length for Hugging Face (HF), it increases up to $n = 256$ and then remains flat for StreamingLLM. This illustrates the advantage of token-pruning which keeps the size of KV cache fixed. But can we do better?

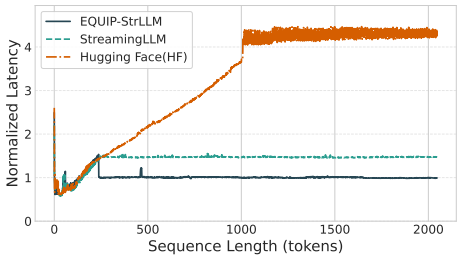


Figure 3: Impact of Efficient KV Management on End-to-End Inference for Llama2-7B (Batch Size = 32).

In the proposed *EQUIP* scheme (denoted as *EQUIP*_StrLLM in Figure 3), the shift-and-append operation of KV cache is replaced with an in-place write for $n > 256$. As a result, the copy overhead incurred in KV cache update is further reduced with *EQUIP*_StrLLM, resulting in further reduction in the end-to-end latency compared to StreamingLLM.

[2]Details of the experimental platform are presented in Section 5

3.2 RoPE Computational Efficiency

In the baseline implementation (without token-pruning), as the absolute positions of tokens do not change, the rotated keys (RK) vector remains the same once computed. Hence, both K' and RK vectors can be cached and reused in subsequent token generation.

In token-pruning methods (Xiao et al., 2024; Zhang et al., 2023), however, new tokens are inserted into the sequence after evicting certain token(s). This potentially changes the effective positional ids of a subset of tokens, necessitating the re-computation of RoPE. Note that even with the change in positional ids, the Rotate_Half values (RK values) remain the same, can benefit from caching. However, shift-and-append operation is required in the cached RK values to account for evicted tokens. Further, the sinusoidal values corresponding to sequence positions need to be element-wise multiplied with the shifted RK values, resulting in increased computational overhead.

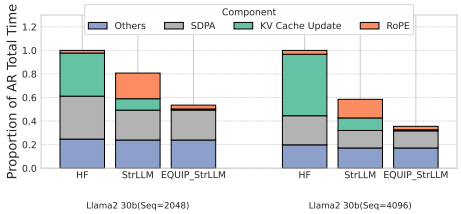


Figure 4: Breakup of End-to-End Inference Time for Batch Size = 16,Cache Size $(s + r)$=256.

Figure 4 shows the breakup of the end-to-end latency in terms of the different computational components, such as KV cache update, SDPA time, RoPE computation time. We note that while the token-pruning method reduces the KV cache management time, the increased RoPE computation time, gives away a significant part of these gains. To recover the lost performance, we propose a smart re-indexing scheme and caching the RK vectors. This approach avoids recompute and shift-and-append, resulting in an overall performance gain of up to 64%.

4 *Equivariance Preserving UPdate-in-Place* (EQUIP) Scheme

As discussed in the previous section, token-pruning methods implement token eviction and insertion as shift-and-append operations which involve significant copy overhead, that is incurred in each iteration

and in each layer, thereby making the decode stage even more memory-bound.

We ask the question, can the key-value corresponding to the new token be written in-place of the evicted token? Will the resulting permuted K and V tensors preserve the attention computation as in the shift-and-append implementation? First, we formally define the *EQUIP* scheme and describe the benefits of in-place replacement. Finally, we establish the equivalence of the SDPA computation under the *EQUIP* scheme.

As mentioned earlier, the token-pruning methods limit the size of K and V tensors to $[B{\cdot}H, n, d]$, where $n = (s + r)$. The shift-and-append operation (as illustrated in Figure 5(a) keeps the tensor contiguous and all the dimensions in order. With our *EQUIP* scheme, however, the new token(s) is (are) written in-place (of the evicted) token(s) as shown in 5(b). This makes the sequence dimension of the K and V tensors to be permuted.

For illustrative purpose, if we assume a batch size $B = 1$ and number of heads $H = 1$, then the dimensions of K and V tensors become $[1, n, d]$, and we can write $K^T = [k_1, k_2, \cdots, k_n]$, where k_i is a column vector. With our *EQUIP* scheme, the K^T tensor would be $[k_{p_1}, k_{p_2}, \cdots, k_{p_n}]$, where $[p_1, p_2, \cdots p_n]$ is a permutation of $[1, 2, \cdots, n]$. In case of StreamingLLM (Xiao et al., 2024) which retains s initial tokens (attention sink) and r recent tokens, and $n = (s + r)$, $[p_1, p_2, \cdots p_s] = [1, 2, \cdots s]$ and $[p_{s+1}, \cdots, p_n]$ is a cyclic permutation of $[(n - r + 1), \cdots, n]$. For H2O (Zhang et al., 2023), the permutation does not have any specific structure as the position of the retained tokens depends on the saliency of the tokens. We refer to the tensors obtained using the *EQUIP* scheme as permuted (more specifically, permuted in the sequence dimension) tensors, and denote them as $P(K)$ and $P(V)$.

4.1 Benefits of *EQUIP* in KV cache Update

Let N be the maximum sequence length. In the baseline implementation (without attention sinks), the K and V tensors grow up to $[B{\cdot}H, N, d]$. The number of copy operations involved for generating N Token in the KV cache update is $\sum_{n=1}^{N} 2B \cdot H \cdot n{\cdot}d \approx B{\cdot}H{\cdot}N^2{\cdot}d$. In the token-pruning methods, the K and V tensors are limited $[B{\cdot}H, n, d]$, where $n = (s{+}r)$. The KV cache update, implemented as a shift-and-append operation, incurs $B \cdot H \cdot r^2 \cdot d$.

Our EQUivariant preserving (*EQUIP*) scheme considers each of the K and V tensors as *logically* partitioned into two parts, one that is unmodified from the previous iteration and another where the in-place write is performed. In case of the StreamingLLM (Xiao et al., 2024), only one token and hence one column of K and V tensors are written in-place in each iteration. For H2O, up to n columns can be rewritten in each iteration. With this, the copy overhead for the generation of N tokens for K and V tensors reduces by a factor of r to $(B{\cdot}H{\cdot}r{\cdot}d)$.

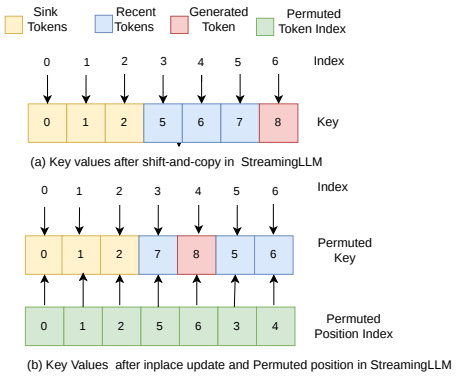


Figure 5: Key Values and Position Index in StreamingLLM with *EQUIP*.

4.2 Benefits of *EQUIP* in RoPE

In the context of token-pruning as tokens get evicted, the absolute positions associated with the Key vectors keep changing (with the shift-and-append operation) as new tokens are generated. Figure 5 depicts the transformed positions of tokens after two tokens are evicted using the attention pruning mechanism for StreamingLLM and *EQUIP*.

RoPE operations in the context of streaming algorithms comprise of two essential steps: (1) looking up precomputed positional encodings based on the positional IDs corresponding to the current sequence of length n, followed by (2) element-wise multiplication of the retrieved positional encodings with Key vectors. As part of the indexing step, for the different positional IDs, we look up the corresponding position encoding vectors, effectively yielding encodings of dimensions $[1, 1, n, d]$. These positional encodings are then broadcasted for elementwise operations with the Key tensors, shaped $[B, H, n, d]$. The complexity of elementwise multiplication for a single iteration remains to be $B \times H \times n \times d$. Rather than physically rearranging cached token representations to enforce ordered positional IDs, *EQUIP* preserves the origi-

nal, potentially permuted positional IDs associated with tokens (as shown in part (b) of Figure 5 in the cache to achieve equivalent element-wise multiplication results). The permuted position IDs are stored explicitly (negligible overhead), and applying RoPE with those IDs to the permuted K layout yields exactly the same transformed keys as after a physical reordering.

Our empirical results show that the element-wise operations in RoPE (shown in Figure 5(a)) has negligible performance difference from permuted index (as shown in Figure 5(b)). Traditional approaches either recompute Rotate_Half from the key cache for each token generation or cache Rotate_Half but still pay the cost of shift-and-append to keep the cache contiguous. *EQUIP* eliminates both costs with in-place update.

4.3 Putting it together: *EQUIP* with RoPE

Figure 6 illustrates the original token-pruning process and the transformed data flow using *EQUIP*. We explain each step in the dataflow shown in the right side of Figure 6. The top box ① indicates the standard initial projection of Q, K, and V values (same as in the original approach). The next box ② performs *EQUIP*'s in-place update of K and V. This overwrites the evicted entries in K_{cache} and V_{cache} with the new K and V. The box marked "Rotate_Half Compute" ④ computes $R = \text{Rotate\_Half}(K)$ for the new key only, and the following box performs an in-place update of RK in the Rotate_Half cache represented as RK_{cache}. Box marked ⑤ demonstrates in-place update of Rotate_Half. The box marked "Reindexing PositionIds" ③ maintains logical position IDs to reflect the permuted order of tokens in the EQUIP KV cache. These IDs are then used to gather $\cos$ / $\sin$ from precomputed RoPE tables. Using the reindexed position IDs together with K_{cache} and Rotate_Half cache, RoPE embeddings ⑥ are computed as

$$\text{RoPE}_K = K_{\text{cache}} \odot \cos\theta + RK_{\text{cache}} \odot \sin\theta,$$

which are then used in the SDPA computation ⑦. The corresponding code implementations are presented in Appendix B.

4.4 Scalability with Custom Kernels

Extending frameworks like vLLM to robustly handle operations like shift and append for specialized caching strategies introduces significant implementation overhead for KV Eviction whose kernels are not mature. Since *EQUIP* allows for an in-place update, one could streamline the logic required to identify the precise target block_id and block_offset within a paged KV cache for incoming key-value pairs. This allows for an in-place write operation, thereby obviating the need for explicit data shifting or complex append logic that might otherwise be required when customizing Paged Attention mechanisms. As demonstrated in the code presented in Figures 16 (Appendix B, we integrated the in-place update approach into the vLLM Paged Attention benchmark to validate the feasibility of in-place updates. This method eliminates the development overhead associated with custom kernel implementations for managing cache eviction and update strategies in scenarios involving attention sinks.

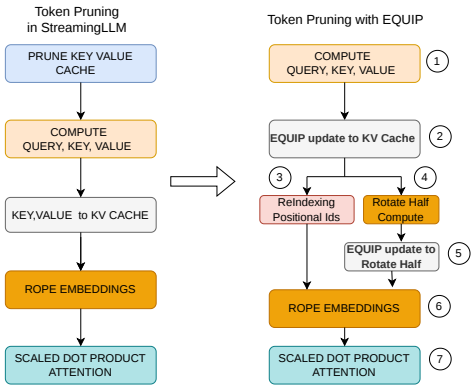


Figure 6: Token-pruning flow and transformations with *EQUIP*.

4.5 Operation Level Equivalence with *EQUIP*

With our *EQUIP* scheme, the K and V tensors are permuted in the sequence dimension. We use the notation $P_i(K)$ and $P_i(V)$ to denote that the tensor's ith dimension[3] are permuted. We use the following definition of equivariant, which is similar to (Kondor and Trivedi, 2018).

Definition 1. *A function is said to be permutation equivariant if its output preserves the permutation. That is, if* $f(P_i(K)) = P_i(f(K))$.

Lemma 1. *Softmax computation is permutation equivariant. That is,* $softmax(P_i(K)) = P_i(softmax(K))$.

Lemma 2. *If the columns of matrix* B *are permuted in* $A \times B$*, then the resulting value is the same as permuting the result matrix* $A \times B$ *using the same permutation. That is* $A \times P_2(B) = P_2(A \times B)$

[3]We count the dimension from left to right.

Next, we establish that the result $A \times B$ is *same* if the columns of A matrix and the rows of B matrix are permuted using the same permute function.

Lemma 3. $A \times B = P_2(A) \times P_1(B)$

Finally we show that our *EQUIP* approach which permutes the K and V tensor results in the same attention output as the shift-and-append approach.

Theorem 1. *Attention* (Q, K, V) = *Attention* $(Q, P_2(K), P_2(V))$.

Appendix C presents the proof for Lemmas and Theorem 1.

5 Experimental Results

We evaluate *EQUIP* for end-to-end inference throughput on a 60-core Intel Xeon Platinum 8490H Sapphire Rapids (SPR) server with 768 GB RAM and MI210 GPUs (128 GB RAM). All experiments use Python 3.10 and PyTorch 2.6.0+cu124. Transformer models and tokenization use Hugging Face Transformers v4.34.0. We evaluate against Paged Attention (v1 - the default implementation) of vLLM v0.8.3. CPU experiments run on Ubuntu 22.04.5 LTS with transparent hugepages enabled; we use opensource OneDNN (Intel Corporation, 2024). GPU experiments use the ROCm 5.7 driver stack.

We compare two versions of *EQUIP*, one on top of StreamingLLM and another on H2O. We refer to these versions as *EQUIP*_StrLLM and *EQUIP*_H2O respectively. Before we present the performance results, we validated that *EQUIP*'s in-place update mechanism preserves the model accuracy of the baseline token-pruning method.

5.1 *EQUIP* Performance on CPU Server

5.1.1 Impact of KV cache Size, Batch Size and Sequence Length

Figure 7a reports the end-to-end inference speedup achieved with *EQUIP* (normalized w.r.t. StreamingLLM performance) for different Llama models.[4] We observe a geomean speedup of 1.62× over StreamingLLM and 4.48× over H2O. Note the end-to-end performance of H2O is poor compared to StreamingLLM due to the scatter-gather operation introduced by token eviction. Our in-place update completely eliminates this overhead, making *EQUIP*_H2O performance competitive with *EQUIP*_StrLLM. The performance improvements achieved by *EQUIP* increase with larger KV cache size. This is because the computational overheads for the shift-and-append operations increase for both the KV cache update and RoPE computation.

Figure 7b demonstrates the sensitivity of inference performance to batch size across different LLM models. As the batch size increases, the KV cache update overheads increase proportionally in StreamingLLM and H2O. Similarly, the computational requirements of RoPE also scale with batch size. In contrast, in *EQUIP*, the computational requirements of KV cache and RoPE update are kept constant (proportional to the number of evicted token(s) in each iteration), leading to higher performance benefits of *EQUIP* with larger batch size. We report the impact of sequence length (at a batch size of 8) in Figure 8a. *EQUIP*_StrLLM achieves a geomean speedup of 1.35× and 1.58× over StreamingLLM.

5.1.2 Impact of Individual Optimizations

Figure 8b shows the breakup in performance gains due to in-place update in KV cache management and RoPE operations for Llama2 30B model. While both optimizations individually result in reasonable performance gains, collectively they achieve significant speedup (1.50× – 1.64×) over the baseline StreamingLLM for BS=16.

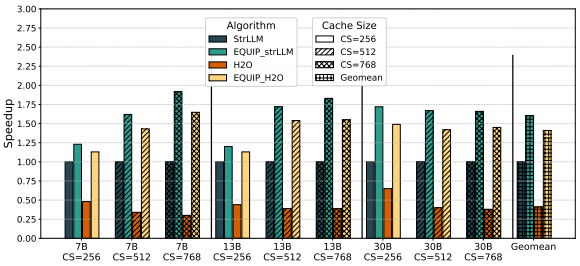
5.1.3 Scalability Results on Multi-Instance and Multi-core Implementation

Our experimental results reveal that *EQUIP*_StrLLM achieves considerable speedup (1.25× – 1.7×) (details presented in Appendix D) over StreamingLLM even when multiple concurrent instances of the inference engines were run on all 60 cores of the SPR server. Further, *EQUIP*_H2O achieves a sustained performance improvement of 2.35× on Llama2-7B (BS=8, cache size=512) across core counts 16, 32, 48 and 60.

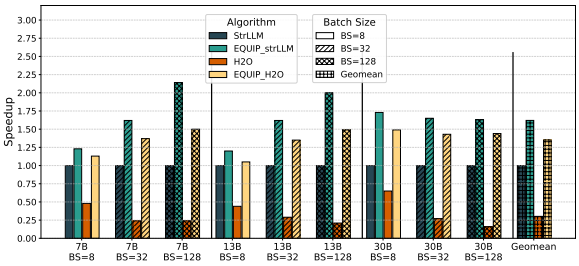
5.2 *EQUIP* Performance on GPU

Figure 9 presents the end-to-end performance of *EQUIP* on the MI210 GPU across various KV cache sizes and sequence lengths. The performance gains range from 1.33× – 1.73×. Table 1 demonstrates the performance gains of *EQUIP* with H2O on MI210 GPUs is similar.

[4]The performance on GPTj6B and GPT20B show a similar trend and are presented in Appendix D

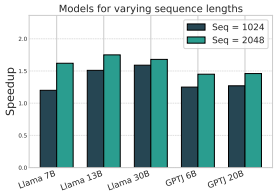


(a) *EQUIP* speedup across different KV cache sizes.

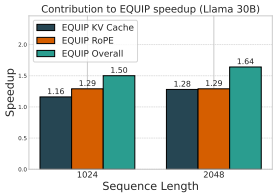


(b) *EQUIP* speedup across different batch sizes.

Figure 7: *EQUIP* Speedup across KV cache sizes and batch sizes on SPR (Sequence length = 1024)

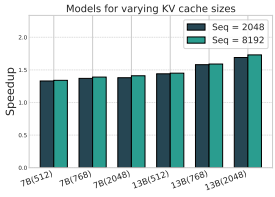


(a) Impact of sequence lengths on speedup (Batch size=8).

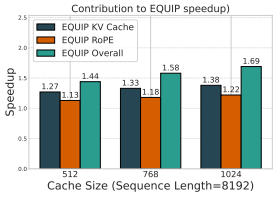


(b) Benefits of individual optimizations with EQUIP (Batch size=16).

Figure 8: *EQUIP*_StrLLM benefits on SPR.



(a) Impact of sequence lengths on speedups



(b) Benefits of individual optimizations

Figure 9: *EQUIP*_strLLM benefits on MI210.(Batch size = 8)

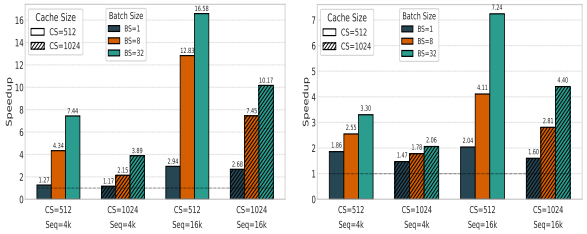
Table 1: *EQUIP*_H2O Speedup over H2O for different KV cache sizes on MI210 GPU (Batch Size=8, Seq. Length=2048).

Model	KV cache Size $(s+r)$		
	512	768	1024
Llama 2-7B	1.61	1.90	1.96
Llama 2-13B	1.89	1.99	2.04

Figure 9b reports the speedup achieved by individual optimizations. They are comparable, although the contribution due to KV cache copy overhead is a bit higher.

5.3 vLLM integration and Comparison

Using our *EQUIP* approach, we implemented StreamingLLM on Paged Attention kernels. Without the *EQUIP* approach, it is difficult to implement token-pruning approaches in Paged Attention kernels, and no such implementation exists as stated in (vLLM Project, 2025e). The performance of our implementation, referred to as *EQUIP*_Str_vLLM, on Llama 2-7B model is depicted in Figure 10. Note that *EQUIP*_Str_vLLM uses pruned KV cache while vLLM uses the full KV cache. On SPR CPU, the throughput of our *EQUIP*_Str_vLLM improves by a factor of up to 7.44× and 16.58× (over vLLM) for sequence lengths of 4K and 16K and at a batch size of 32. On MI-210 GPU, *EQUIP*_Str_vLLM achieves performance gains in the range 1.47× – 7.24× over the auto-regressive Paged Attention.



(a) *EQUIP* Speedup - SPR (b) *EQUIP* Speedup - MI210

Figure 10: *EQUIP*_str_vllm speedup over Auto-regressive Paged Attention on Llama2 7B.

Appendix D presents additional experimental results on different Llama models, multi-token eviction, and detailed accuracy results.

5.4 *EQUIP* under Multi-token Eviction and Speculative Decoding

We evaluate *EQUIP* integrated with StreamingLLM under a speculative decoding setup. Using Llama 2-7B with RoPE disabled on Intel SPR with speculation length γ=8 and acceptance probability 0.6, *EQUIP* achieves wall-clock speedups of 1.47× (BS=8, cache=768) to 1.67× (BS=16, cache=1024) over StreamingLLM with speculative decoding. We report speedup results for multi-token eviction (Table 6) which corroborate well with speculative decoding results.

5.5 Performance for Smaller Batch Size and Sequence Lengths

The performance gains of EQUIP are modest at small batch sizes, and sequence lengths. At small batch sizes (BS=1 or 2), MLP computation time dominates over attention computation and KV cache update, limiting the observable speedup. Similarly, token-pruning kicks in only after the initial $(s+r)$ tokens are generated, and benefits of *EQUIP* accrue only when the sequence length grows well past $(s + r)$. As batch size and sequence length increase, KV cache copy and RoPE recomputation become the dominant bottlenecks, and *EQUIP*'s in-place updates achieve progressively larger gains.

Table 2 reports the speedup of *EQUIP* over StreamingLLM for cache size $(s + r) = 756$ on Llama 2-7B (Intel SPR), illustrating the transition from MLP-dominated to memory-movement-dominated execution.

Table 2: *EQUIP*_StrLLM speedup over StreamingLLM (Llama 2-7B, cache size $(s + r)$=756, Intel SPR).

	Seq=1024	Seq=1536	Seq=2048	Seq=4096
BS=1	1.037	1.065	1.067	1.07
BS=4	1.042	1.116	1.152	1.209
BS=8	1.147	1.486	1.899	2.45

5.6 Accuracy Preservation

EQUIP preserves exact numerical parity with the underlying token-pruning method. We validated this using the perplexity metric on long sequences where token eviction is actively occurring. Table 9 reports perplexity for StreamingLLM and *EQUIP* on Llama 2-7B across different cache configurations; the values are identical, confirming that the in-place update introduces no accuracy degradation. At the layer level, maximum deviations are less than 10^{-9} at SDPA and less than 10^{-5} at RoPE outputs (see Figure 19 in Appendix). On ARC-Easy under a streaming setup (Llama 2-7B, 1024 samples, 78,349 tokens, cache size=512), *EQUIP* and StreamingLLM achieve identical accuracy of 74.90%.

5.7 Comparison with KV Cache Quantization

KV cache quantization methods such as KVQuant (Hooper et al., 2024) and KIVI (Liu et al., 2024) reduce memory by compressing stored key-value representations to lower bit-widths. We compare token-pruning (and *EQUIP*) with quantization along three dimensions.

Accuracy. For sequences beyond the trained context window, standard auto-regressive inference degrades significantly as the model has not been trained to attend over such long contexts. Quantized models are less accurate than the baseline, and this gap is likely to continue at longer sequence lengths. On the other hand, token-pruning with attention sinks (Xiao et al., 2024) (Zhang et al., 2023) helps reduce this accuracy loss by maintaining a bounded working set of the most relevant tokens, enabling stable perplexity at sequence lengths well beyond the trained window. *EQUIP* is an optimization over token-pruning and achieves the same accuracy and perplexity as the underlying token-pruning method.

Performance. Quantization reduces the per-token memory footprint but does not address unbounded KV cache growth; as the context grows, the number of cached tokens continues to increase, and so does the total memory and computation required for attention. Token-pruning, by contrast, bounds the KV cache size to a fixed budget of $(s+r)$ tokens, preventing unbounded growth and enabling constant-memory, constant-computation inference for arbitrarily long sequences.

Orthogonality. Pruning and quantization are orthogonal and can co-exist: quantization compresses what is stored, while pruning decides what to keep. Combining token-pruning methods (with *EQUIP*) and quantization-based compression would yield compounding benefits—reduced per-token storage from quantization and reduced cache management overhead from *EQUIP*. A study on this is deferred to future work.

6 Conclusion

Existing token-pruning methods incur inefficiencies in KV cache updates and RoPE computations. We introduce a simple EQUivariant preserving (*EQUIP*) scheme that transforms token eviction and insertion operations into an in-place replacement, reducing the KV cache memory management cost significantly. Second, we present three optimizations (caching RK tensors, in-place update, and reindexing) that improve the efficiency of RoPE computations. Together, these contributions significantly enhance the efficiency of long-context inference, resulting in significant improvement in end-to-end inference throughput. We demonstrate that the proposed *EQUIP* technique exhibits broader applicability across diverse pruning models.

7 Limitations

Memory overhead of cached rotated keys. Caching rotated-key (RK) tensors alongside K removes repeated `Rotate_Half` computations and avoids shifting the RoPE half-cache whenever tokens are evicted, as discussed in Section 3 and in the integrated *EQUIP* dataflow of Figure 6. This design trades additional memory for lower compute: because pruning fixes the number of live tokens, the extra footprint stays bounded and is at most $1.5\times$ that of conventional approaches that do not retain an RK cache in the regimes we study. Deployments with tight memory budgets can disable the RoPE-cache optimization and trade higher RoPE cost for a smaller working set.

Token-pruning policies with positional dependency. Section A surveys token-pruning methods, including SnapKV (Li et al., 2024a), whose pooling can depend on absolute indices within a local window. While EQUIP can be applied across different token-pruning methods, doing so may require developing custom pooling functions tailored to their position-dependent behaviors.

Models without RoPE. For architectures that do not apply RoPE to queries and keys (e.g., OPT), the rotation-specific savings vanish. The remaining benefit comes from the in-place KV cache update analyzed in Section 3 and illustrated in Figure 5, rather than from reindexing and caching in the RoPE path.

Acknowledgments

The authors thank Prof. Chiranjib Bhattacharyya (IISc) and Viren Radhakrishnan (AMD/IISc) for their insights and comments in this work. This work was supported in part by AMD Inc. through the AMD–IISc collaboration. The authors gratefully acknowledge AMD's financial and technical support from AMD Research and Advanced Development (RAD), as well as the collaborative environment that aided experimental evaluation and interpretation of results. This material is also based upon work supported by the REAL@USC-Meta Center and a VMware gift. Last, the authors thank the anonymous reviewers for their detailed feedback and suggestions, which enhanced the rigor and presentation of this article.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mo Bavarian, Jeff Belgum, Irwan Bello, and 260 others. 2023. Gpt-4 technical report. *arXiv preprint*.

Amazon. 2024. Amazon codewhisperer. `https://aws.amazon.com/codewhisperer/`. [n.d.].

Reza Yazdani Aminabadi, Samyam Rajbhandari, Minjia Zhang, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Jeff Rasley, Shaden Smith, Olatunji Ruwase, and Yuxiong He. 2022. Deepspeed inference: Enabling efficient inference of transformer models at unprecedented scale. *SC '22: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*.

E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. 1999. LAPACK – linear algebra PACKage. `https://www.netlib.org/lapack/`.

Rie Kubota Ando and Tong Zhang. 2005. A framework for learning predictive structures from multiple tasks and unlabeled data. *Journal of Machine Learning Research*, 6:1817–1853.

Anthropic. 2024. Claude. `https://claude.ai`. [n.d.].

Apache TVM. 2025. Optimize llm tutorial. `https://tvm.apache.org/docs/how_to/tutorials/optimize_llm.html`. Accessed April 11, 2025.

Somashekaracharya G. Bhaskaracharya, Julien Demouth, and Vinod Grover. 2020. Automatic kernel generation for volta tensor cores.

Bing. 2024. Bing ai. `https://www.bing.com/chat`. [n.d.].

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, and 12 others. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33.

Zhuoming Chen, Avner May, Ruslan Svirschevski, Yuhsun Huang, Max Ryabinin, Zhihao Jia, and Beidi Chen. 2024. Sequoia: Scalable, robust, and hardware-aware speculative decoding. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez,

Ion Stoica, and Eric P. Xing. 2023. Vicuna: An open-source chatbot impressing GPT-4 with 90%* ChatGPT quality. https://lmsys.org/blog/2023-03-30-vicuna. Accessed: 2023-03-30.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, and 48 others. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint*.

Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. 2019. BoolQ: Exploring the surprising difficulty of natural yes/no questions. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pages 2924–2936.

Shabnam Daghaghi, Nicholas Meisburger, Mengnan Zhao, Yong Wu, Sameh Gobriel, Charlie Tai, and Anshumali Shrivastava. 2021. Accelerating SLIDE deep learning on modern CPUs: Vectorization, quantizations, memory optimizations, and more. In *Proceedings of Machine Learning and Systems (MLSys)*.

Tri Dao. 2024. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations (ICLR)*.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Dao-AILab. 2024. Flashattention issue #59. Accessed: 2024-10-26.

Zachary Devito. Aten. https://github.com/zdevito/ATen.

Yuan Feng, Junlin Lv, Yukun Cao, Xike Xie, and S. Kevin Zhou. 2025. Identify critical kv cache in llm inference from an output perturbation perspective. *Preprint*, arXiv:2502.03805. Submitted on 6 Feb 2025.

Suyu Ge, Yunan Zhang, Liyuan Liu, Minjia Zhang, Jiawei Han, and Jianfeng Gao. 2024. Model tells you what to discard: Adaptive KV cache compression for LLMs. In *International Conference on Learning Representations (ICLR)*.

Gemini Team and Google. 2023. Gemini: A family of highly capable multimodal models.

Nathan Godey, Alessio Devoto, Yu Zhao, Simone Scardapane, Pasquale Minervini, Éric de la Clergerie, and Benoît Sagot. 2025. Q-filters: Leveraging qk geometry for efficient kv cache compression. *Preprint*, arXiv:2503.02812. Submitted on 4 Mar 2025.

Gaël Guennebaud and Benoit Jacob et al. *Eigen: C++ Template Library for Linear Algebra*.

Pujiang He, Shan Zhou, Wenhuan Huang, Changqing Li, Duyi Wang, Bin Guo, Chen Meng, Sheng Gui, Weifei Yu, and Yi Xie. 2024a. Inference performance optimization for large language models on cpus. *arXiv preprint*.

Zhenyu He, Jun Zhang, Shengjie Luo, Jingjing Xu, Zhi Zhang, and Di He. 2024b. Let the code LLM edit itself when you edit the code. *arXiv preprint arXiv:2407.03157*.

Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W. Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. 2024. KVQuant: Towards 10 million context length LLM inference with KV cache quantization. In *Advances in Neural Information Processing Systems (NeurIPS)*.

HuggingFace. 2022. Issue: Question about loading model to a specific device using 'transformers' library. https://github.com/huggingface/transformers/issues/17653.

HuggingFace. 2024. Transformers: State-of-the-art natural language processing for pytorch, tensorflow, and jax. Accessed: 2024-07-06.

A. H. Hunter, Chris Kennelly, Paul Turner, Darryl Gove, Tipp Moseley, and Parthasarathy Ranganathan. 2021. Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 257–273.

Intel. 2024a. The AI PC opportunity. Intel White Paper. Originally available at https://www.intel.com/content/dam/www/central-libraries/us/en/documents/2024-01/the-ai-pc-opportunity-white-paper.pdf; accessed January 2024.

Intel. 2024b. Mkl: Improved small matrix performance using just-in-time (jit) code. Accessed: 2024-08-01.

Intel Corporation. 2024. *oneDNN API Documentation*. Accessed: 2024-10-31.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7b.

Risi Kondor and Shubhendu Trivedi. 2018. On the generalization of equivariance and convolution in neural networks to the action of compact groups. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 2747–2755.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*. ACM.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning (ICML)*, pages 19274–19286.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. 2024a. SnapKV: LLM knows what you are looking for before generation. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2024b. EAGLE-2: Faster inference of language models with dynamic draft trees. In *Empirical Methods in Natural Language Processing (EMNLP)*.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2024c. EAGLE: Speculative sampling requires rethinking feature uncertainty. In *International Conference on Machine Learning (ICML)*.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2025. EAGLE-3: Scaling up inference acceleration of large language models via training-time test.

Manlai Liang, JiaMing Zhang, Xiong Li, and Jinlong Li. 2025. LagKV: Lag-relative information of the KV cache tells which tokens are important. *Preprint*, arXiv:2504.04704.

Linux Kernel Community. 2024. Transparent hugepages – Linux kernel documentation. https://docs.kernel.org/admin-guide/mm/transhuge.html. Accessed: 2024-07-21.

Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. 2024. KIVI: A tuning-free asymmetric 2bit quantization for KV cache. In *International Conference on Machine Learning (ICML)*.

llama. 2023. Llama-7b: A large language model, hugging face and huggyllama. https://huggingface.co/huggyllama/llama-7b. Accessed: 2024-07-23.

Xinyin Ma, Gongfan Fang, and Xinchao Wang. 2023. Llm-pruner: On the structural pruning of large language models. In *Advances in Neural Information Processing Systems*.

Meta. 2024. Introducing meta llama 3: The most capable openly available llm to date.

Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. 2018. Can a suit of armor conduct electricity? A new dataset for open book question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. OpenBookQA dataset: https://huggingface.co/datasets/allenai/openbookqa.

MLCommons. 2024. Inference data center benchmarks. Accessed: 2024-08-01.

NVIDIA. 2022. Fastertransformer. https://github.com/NVIDIA/FasterTransformer. Accessed: 2024-08-01.

NVIDIA. 2023. Tensorrt-llm. Accessed: 2024-08-02.

NVIDIA. 2025a. Chunkkvpress. https://github.com/NVIDIA/kvpress/blob/main/kvpress/presses/chunkkv_press.py.

NVIDIA. 2025b. Chunkpress. https://github.com/NVIDIA/kvpress/blob/main/kvpress/presses/chunk_press.py.

NVIDIA. 2025c. Expectedattention. https://github.com/NVIDIA/kvpress/blob/main/kvpress/presses/expected_attention_press.py.

NVIDIA. 2025d. Finchpress. https://github.com/NVIDIA/kvpress/blob/main/kvpress/presses/finch_press.py.

NVIDIA. 2025e. Keyrerotationpress. https://github.com/NVIDIA/kvpress/blob/main/kvpress/presses/key_rerotation_press.py.

NVIDIA. 2025f. Observedattention. https://github.com/NVIDIA/kvpress/blob/main/kvpress/presses/observed_attention_press.py.

NVIDIA. 2025g. Randompress. https://github.com/NVIDIA/kvpress/blob/main/kvpress/presses/random_press.py.

NVIDIA. 2025h. Scorerpress. https://github.com/NVIDIA/kvpress/blob/main/kvpress/presses/scorer_press.py.

NVIDIA. 2025i. Tova. https://github.com/NVIDIA/kvpress/blob/main/kvpress/presses/tova_press.py.

S. Park, J. Choi, S. Lee, and U. Kang. 2024. A comprehensive survey of compression algorithms for language models. *arXiv preprint*.

Aaron Pham, Chaoyu Yang, Sean Sheng, Shenyang Zhao, Sauyon Lee, Bo Jiang, Fog Dong, Xipeng Guan, and Frost Ming. 2023. Openllm: Operating llms in production.

Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. 2025. vAttention: Dynamic memory management for serving LLMs without PagedAttention. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

PyTorch. 2024. Pytorch serve. Accessed: 2024-08-01.

Ranajoy Sadhukhan, Jian Chen, Zhuoming Chen, Vashisth Tiwari, Ruihang Lai, Jinyuan Shi, Ian En-Hsu Yen, Avner May, Tianqi Chen, and Beidi Chen. 2025. MagicDec: Breaking the latency-throughput tradeoff for long context generation with speculative decoding. In *International Conference on Learning Representations (ICLR)*.

Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. 2024. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Noam Shazeer. 2019. Fast transformer decoding: One write-head is all you need. *arXiv preprint*.

Haihao Shen, Hanwen Chang, Bo Dong, Yu Luo, and Hengyu Meng. 2023. Efficient LLM inference on CPUs. In *NeurIPS 2023 Workshop on Efficient Natural Language and Speech Processing (ENLSP-III)*.

Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. FlexGen: High-throughput generative inference of large language models with a single GPU. In *International Conference on Machine Learning (ICML)*.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint*.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. 2021. RoFormer: Enhanced transformer with rotary position embedding. *arXiv preprint arXiv:2104.09864*.

Y. Tang, Y. Wang, J. Guo, Z. Tu, K. Han, H. Hu, and D. Tao. 2024. A survey on transformer compression. *arXiv preprint*.

TensorFlow. 2019. Hugging face: State-of-art natural language processing. TensorFlow Blog, https://blog.tensorflow.org/2019/11/hugging-face-state-of-art-natural.html.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, and 49 others. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint*.

vLLM. 2024. vllm-benchmarks. Accessed: 2024-10-26.

vLLM. 2025. Basic. https://docs.vllm.ai/en/latest/contributing/model/basic.html. Accessed April 11, 2025.

vLLM Community. 2025. Dynamic kv cache management in vllm. https://github.com/vllm-project/vllm. Accessed: 2025-02-05.

vLLM-CPU. 2024a. Getting started: Cpu installation, vllm documentation. Accessed: 2024-10-26.

vLLM-CPU. 2024b. vllm-project: Cpu attention implementation. Retrieved July 6, 2024.

vLLM Project. 2025a. attention.cpp. https://github.com/vllm-project/vllm/blob/main/csrc/cpu/attention.cpp. Retrieved April 1, 2025. File path may have changed in later vLLM versions.

vLLM Project. 2025b. Discussion #547. https://github.com/vllm-project/vllm/discussions/547. Accessed: 2025-04-01.

vLLM Project. 2025c. Dynamic KV cache compression based on vLLM framework. https://github.com/vllm-project/vllm/issues/10942. RFC.

vLLM Project. 2025d. Issue 10491. https://github.com/vllm-project/vllm/issues/10491. Accessed: 2025-02-05.

vLLM Project. 2025e. Issue 1304. https://github.com/vllm-project/vllm/issues/1304. Accessed: 2025-02-05.

vLLM Team. Speculative Decoding. https://docs.vllm.ai/en/latest/features/spec_decode.html. Accessed: 2025-03-28.

Zhongwei Wan, Xin Wang, Che Liu, Samiul Alam, Yu Zheng, Jiachen Liu, Zhongnan Qu, Shen Yan, Yi Zhu, Quanlu Zhang, Mosharaf Chowdhury, and Mi Zhang. 2023. Efficient large language models: A survey. *arXiv preprint*.

Ben Wang and Aran Komatsuzaki. 2021. GPT-J-6B: A 6 billion parameter autoregressive language model. https://huggingface.co/docs/transformers/en/model_doc/gptj. Accessed: 2024-08-01.

W. Wang, W. Chen, Y. Luo, Y. Long, Z. Lin, L. Zhang, B. Lin, D. Cai, and X. He. 2024. Model compression and efficient inference for large language models: A survey. *arXiv preprint*.

Bingyang Wu, Yinmin Zhong, Zili Zhang, Shengyu Liu, Fangyue Liu, Yuanhang Sun, Gang Huang, Xuanzhe Liu, and Xin Jin. 2024. Fast distributed inference serving for large language models. Accessed: July 6, 2024.

Heming Xia, Zhe Yang, Qingxiu Dong, Peiyi Wang, Yongqi Li, Tao Ge, Tianyu Liu, Wenjie Li, and Zhifang Sui. 2024. Unlocking efficiency in large language model inference: A comprehensive survey of speculative decoding. In *Findings of the Association for Computational Linguistics (ACL)*.

Guangxuan Xiao, Jiaming Tang, Jingwei Zuo, Junxian Guo, Shang Yang, Haotian Tang, Yao Fu, and Song Han. 2025. DuoAttention: Efficient long-context LLM inference with retrieval and streaming heads. In *International Conference on Learning Representations (ICLR)*.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2024. Efficient streaming language models with attention sinks. In *International Conference on Learning Representations (ICLR)*.

Dongjie Yang, Xiaodong Han, Yan Gao, Yao Hu, Shilin Zhang, and Hai Zhao. 2024. PyramidInfer: Pyramid KV cache compression for high-throughput LLM inference. In *Findings of the 2024 Conference of the Association for Computational Linguistics (ACL)*.

Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. ORCA: A distributed serving system for transformer-based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

Chen Zhang, Xinyi Dai, Yaxiong Wu, Qu Yang, Yasheng Wang, Ruiming Tang, and Yong Liu. 2025a. A survey on multi-turn interaction capabilities of large language models.

Libo Zhang, Zhaoning Zhang, Xubaizhou, Rui Li, Zhiliang Tian, Songzhu Mei, and Dongsheng Li. 2025b. Dovetail: A CPU/GPU heterogeneous speculative decoding for LLM inference. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Xuan Zhang, Cunxiao Du, Chao Du, Tianyu Pang, Wei Gao, and Min Lin. 2024. Simlayerkv: A simple framework for layer-level kv cache reduction. *Preprint*, arXiv:2410.13846.

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. H2o: Heavy-hitter oracle for efficient generative inference of large language models. In *NeurIPS 2023 Proceedings*.

Yilong Zhao, Chien-Yu Lin, Kan Zhu, Zihao Ye, Lequn Chen, Size Zheng, Luis Ceze, Arvind Krishnamurthy, Tianqi Chen, and Baris Kasikci. 2024. Atom: Low-bit quantization for efficient and accurate llm serving. In *Proceedings of Machine Learning and Systems (MLSys)*, pages 196–209.

Xunyu Zhu, Jian Li, Yong Liu, Can Ma, and Weiping Wang. 2023. A survey on model compression for large language models. *arXiv preprint arXiv:2308.07633*.

A Background

A.1 Attention Computation

An LLM consists of a sequence of L layers, each layer comprising Attention and Multi-Layer Perceptron (MLP) Blocks. The attention computation starts with the current token x, and computes the query, key and value vectors as:

$$Q = W_q \cdot x; \quad K = W_k \cdot x; \quad V = W_v \cdot x$$

where W_q, W_k, and W_v are weight matrices.

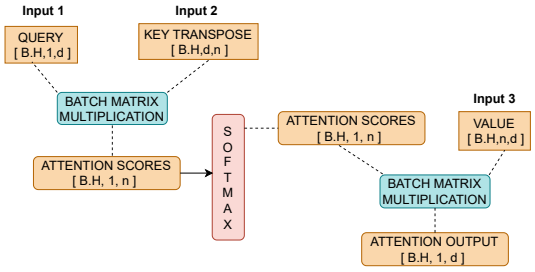


Figure 11: Scaled dot-product attention (SDPA).

For performing the attention operation (Shazeer, 2019) for a batch size B and across L layers, the K and V matrices become tensors of dimension $[B, L, n, D]$, where n is the current context length (this includes the prompt length along with the total tokens generated so far) and D is the total number of hidden dimensions. For a maximum context length of N, these tensors can be as large as $[B, L, N, D]$.

In multi-head attention, the model's hidden dimension (D), is processed using H parallel attention heads.Each head operates on query Q, key K and value V vectors that are projected to a dimension d. These parameters are typically chosen such that the overall hidden dimension D is equal to the number of heads H multiplied by the dimension per-head d (i.e., $D = H \cdot d$), which implies $d = D/H$. The Q, K, and V vectors are reshaped to merge the H heads with the batch size B. For each individual layer of L, the dimension of Q, K, V vector becomes $[B{\cdot}H, 1, d]$ while the k and v tensors corresponding to the (n-1) tokens so far are of dimension $[B{\cdot}H, n-1, d]$. The concatenation of KV matrix with vector results in $[B{\cdot}H, n, d]$. The batch matrix multiplication of $(Q{\cdot}K^T)$ results in a squeezed tensor of shape $[B \cdot H, 1, n]$. The second operation associated with Scaled Dot-Product Attention (SDPA). (SDPA) is element-wise softmax, where the contracted tensor remains as $[B{\cdot}H, 1, n]$. In the final step (refer to Figure 11), the softmax

outputs are batch multiplied with V, resulting in $[B{\cdot}H, 1, d]$.

The self-attention computation for the $(n+1)$-th token relies on the Key (K) and Value (V) tensors computed from the previous n tokens. To eliminate redundant calculations, the key-value (KV) cache mechanism (Ge et al., 2024) efficiently stores these tensors, allowing fast retrieval during subsequent token computations. Although implementation specifics differ across inference systems (HuggingFace, 2024; Aminabadi et al., 2022; Kwon et al., 2023), the fundamental operation of updating the cache with K and V tensors typically results either in data movement and memory management overhead or involves custom kernel implementations to optimize performance.

A.2 Rotary Positional Encoding (RoPE)

RoPE has emerged as a prevalent technique for incorporating positional information in transformer architectures. It operates by applying position-dependent rotations to subspaces of the query (Q) and key (K) vectors before the attention computation. This mechanism effectively encodes both the absolute position of tokens and their relative distances within the sequence. As illustrated in Figure 2, these rotations, often derived from precomputed sinusoidal values corresponding to sequence positions (e.g., for n positions across d dimensions), are applied element-wise to the Q and K tensors. The resulting RoPE-enhanced tensors subsequently serve as inputs (input1, input2 and input3) as in Figure 11 to the Scaled Dot-Product Attention (SDPA) mechanism.

RoPE rotational transformation is typically achieved via an operation of the form $k'_p = k_p \odot \cos(p\theta) + \text{Rotate\_Half}(k_p) \odot \sin(p\theta)$, where $\odot$ denotes element-wise multiplication, p is the position index of the token and $\text{Rotate\_Half}(k_p)$ permutes feature pairs within k_p. For example, if

$$k_p = [k_{p,1}, k_{p,2}, \ldots, k_{p,d-1}, k_{p,d}],$$

then

$$\text{Rotate\_Half}(k_p) = [-k_{p,2}, k_{p,1}, \ldots, -k_{p,d}, k_{p,d-1}],$$

under the assumption that d is even.

The positional scaling factors $\cos(p\theta)$ and $\sin(p\theta)$ depend on the position p and are typically precomputed for all positions up to a maximum sequence length. For a given position p, these factors form vectors of shape $[1, d]$, where d is the feature dimension per attention head.

To apply RoPE to the keys for a token at a particular position index p, the corresponding slice of the Key tensor, which is a $[1, d]$ vector, is broadcast across the batch (B) and head (H) dimensions to match the $[B, H, 1, d]$ shape of the Key vector for the element-wise multiplication. This operation over n tokens results in an output tensor of dimensions $[B, H, n, d]$. In the RoPE computation, the right part of Figure 2 (which computes the element-wise multiplication with $p.Sin(\theta)$) will be referred to as Rotate_Half part, while the left part will be referred to as the straight part. The k tensor for the straight part is anyway stored in the KV cache. The rotate part, which requires the d dimensional vector to be rotated and appropriately negated in the even positions, is performed on the fly during the RoPE computation. This operation ($\text{Rotate\_Half}(k_p)$) incurs significant computational cost.

A.3 Token-pruning Methods

Deploying LLMs in streaming applications, such as multi-round dialogue (Zhang et al., 2025a), faces a significant challenge as the size of KV cache grows in each round. This results in even higher memory consumption and copy overhead. Also, LLMs often exhibit limited generalization capability for texts that exceed pretrained sequence lengths. To address these challenges, several recent studies (Zhang et al., 2023) (Xiao et al., 2024) (Li et al., 2024a) have focused on identifying a critical subset of tokens that influence output predictions. The key tokens are identified using metrics like attention weights or gradient scores. We focus on two major token-pruning schemes, viz., StreamingLLM (Xiao et al., 2024) and H2O (Zhang et al., 2023).

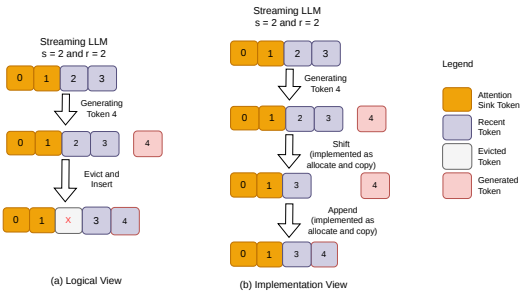


Figure 12: StreamingLLM: Evict-and-Insert implemented as shift-and-append Operations.

Figure 12 illustrates a key-value (KV) cache management strategy employing fixed attention sinks

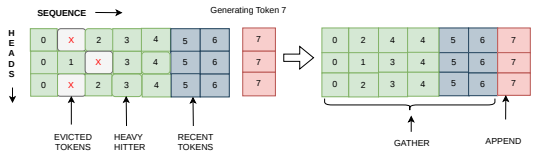


Figure 13: H2O: Evict-and-Insert implemented using gather-and-append Operations.

(of size s) alongside a rolling buffer for recent tokens (of size r). As each new token is generated, it is appended to this buffer, while the oldest token (apart from the initial attention sinks) is evicted. Thus, after the initial set of $(s + r)$ tokens, the size of the K and V tensors are limited to the dimension $[B{\cdot}H, (s + r), d]$. Managing the KV cache after eviction involves one of the following computationally expensive strategies.

(1) KV cache Management with shift-and-append: In this method, token eviction is implemented as a copy operation of the entire K and V tensors, leaving out the evicted row/column. Additionally an append operation for insertion of the new token introduces another memory allocation and copying overhead. This approach is implemented by Huggingface (HuggingFace, 2024).

(2) KV cache Management with RoPE: Token eviction in the key-value (KV) cache introduces fragmentation, which disrupts the positional integrity of tokens. Specifically, when tokens are evicted and the remaining tokens are shifted, the positional indices associated with each token are altered. This necessitates the recomputation of positional embeddings, such as those used in Rotary Positional Embedding (RoPE) schemes, to maintain the correct mapping between token positions and their corresponding embeddings. Unlike Keys that are cached, in scenarios where the intermediate results of the Rotate_Half operation are not cached, each positional embedding must be recalculated for all the tokens, leading to increased computational cost. While it is possible to cache the output of the Rotate_Half operation for every key in the cache, this approach still requires a shift-and-append operation on the cache whenever a token is evicted as illustrated in Figure 18.

H2O implementation of a dynamic key-value (KV) cache pruning strategy considers both recent local tokens and global contextual information to manage cache entries. The cache capacity, denoted as n, is sum of a predefined number h of Heavy Hitters and recent entries (r). When the total length of KV entries surpasses this capacity n, the top HH H2O are identified based on their saliency scores. Figure 13 specifically demonstrates the eviction mechanism, where these top 4 heavy hitters and r recent tokens are prioritized for retention. The selected heavy hitters may be different in different head computations.

This Dynamic token-pruning is implemented through boolean masks corresponding to the top h H2O tokens. The masks make use of gather operations to copy the desired KV cache entries (HuggingFace, 2024), incurring allocation and copy overheads.

While state of the art inference serving systems (vLLM) (Kwon et al., 2023) supports window attention (Jiang et al., 2023), their extension to token-pruning schemes is an actively explored project which is under progress (vLLM Project, 2025e). The presence of attention sinks complicates the implementation. Handling custom Paged Attention without impacting performance poses significant challenges and hence vLLM does not support StreamingLLM or other token-pruning techniques (vLLM Project, 2025c) (vLLM Project, 2025e) (vLLM Project, 2025d).

B Pseudocode

```
def apply_rotary_pos_emb_single(x,cos,sin,
                                reindex_position_ids):
    cos_seq = cos.squeeze(1).squeeze(0)
# Squeeze the first 2 dimensions that are 1 [seq_len,dim]
    sin_seq = sin.squeeze(1).squeeze(0)
# Squeeze the first 2 dimensions that are 1 [seq_len,dim]
    cos_gathered = cos_seq[reindex_position_ids].unsqz(1)
# Gather along position_ids [bs, 1, seq_len, dim]
    sin_gathered = sin_seq[reindex_position_ids].unsqz(1)
# Gather along reindex_position_ids ids [bs,1,seq_len,dim]
    x_embed = (x * cos_gathered) +
        (cached_rotate_half  * sin_gathered)
    return x_embed
```

Figure 14: RoPE computation with *EQUIP*.

```
def equip(K_Cache,V_Cache,Cache_Rotate_half,evict_index,
                                                K,v):
   K_Cache[evict_index:evict_index+1] = k #key
   V_Cache[evict_index:evict_index+1] = v  #value
  Cache_Rotate_half[evict_index:evict_index+1] =
   Rotate_Half(k)  #key
```

Figure 15: *EQUIP* update.

C Proof for Operation Level Equivalence with *EQUIP*

In this section we establish the operation level equivalence of our *EQUIP* approach for attention compu-

```
def inplace_update_streaming_cache(token_id: int,
                                   block_size: int,
                                   block_tables: torch.Tensor,
                                   key_cache: torch.Tensor,
                                   value_cache: torch.Tensor,
                                   keys: torch.Tensor,
                                   values: torch.Tensor)
                                   -> None:

    # block index and offset Computation
    #  for the token position
    block_idx = token_id // block_size
    offset = token_id % block_size
    # Map logical block index to physical block index
    physical_block_idx = block_tables[0, block_idx].item()
    # Destination key and value cache slices for
    #in-place update
    dest_key = key_cache[physical_block_idx, offset]
    dest_value = value_cache[physical_block_idx, offset]
    # Source key and value vectors to be copied into
cache
    src_key = keys[0]
    src_value = values[0]
    # in-place update of the cached key and value for
    # StreamingLLM
    dest_key.copy_(src_key)
    dest_value.copy_(src_value)
```

Figure 16: StreamingLLM KV Update (Preprocessing step for Paged Attention Kernels).

tation. Specifically, we show that the results computed by the Scaled Dot-Product Attention (SDPA) operations remain the same across shift-and-append and our *EQUIP*. We then extend the proof for RoPE and attention masks. To make the section self-contained, we repeat the definitions, lemmas and theorem.

C.1 SDPA Equivalence

With our *EQUIP* scheme, the K and V tensors are permuted in the sequence dimension. We use the notation $P_i(K)$ and $P_i(V)$ to denote that the tensor's ith dimension[5] are permuted.

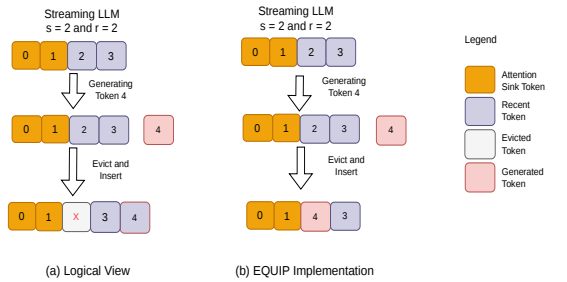


Figure 17: Evict-and-Insert implemented as in-place operation with *EQUIP*.

We use the following definition of equivariant, which is similar to (Kondor and Trivedi, 2018).

Definition 1. *A function is said to be permutation equivariant if its output preserves the permutation. That is, if* $f(P_i(K)) = P_i(f(K))$

[5]We count the dimension from left to right.

Lemma 1. *Softmax computation is permutation equivariant. That is,*

$$Softmax(P_i(K)) = P_i(Softmax(K))$$

Proof: The Softmax computation of $K = [k_{b,i,j}]$, for $b \in [1, B{\cdot}H]$, $i \in [1, n]$ and $j \in [1, d]$ along the second dimension is defined as:

$$Softmax(K) = [z_{b,i,j}], \text{where}$$

$$z_{b,i,j} = \frac{e^{k_{b,i,j}}}{\sum_{j=1}^{d} e^{x_{b,i,j}}}$$

It is easy to see that the denominator in the RHS expression of $z_{b,i,j}$ is unaffected when the second dimension of K are permuted. Thus the $z_{b,i,j}$ computed without permutation is the same as $z_{p_i,j}$. Hence Softmax of K under *EQUIP* is the same as that computed under the shift-and-append approach, except that the second dimensions are permuted. Thus $Softmax(P_i(K)) = P_i(Softmax(K))$.

Next we show that permutation equivariance under matrix multiplication w.r.t. the columns of the second operand.

Lemma 2. *If the columns of matrix B are permuted in $A \times B$, then the resulting value is the same as permuting the result matrix $A \times B$ using the same permutation. That is $A \times P_2(B) = P_2(A \times B)$*

Proof: Consider two matrices $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times k}$. Their product $C = A \times B$ is defined as:

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} \cdot B_{k,j} \tag{1}$$

With the columns of B permuted (denoted by B_{k,p_j}, where p_j represents the permuted position of the jth column), the RHS of $A \times P_2(B)$ computes $\sum_{k=1}^{n} A_{i,k} \cdot B_{k,p_j}$, which is C_{i,p_j}. Thus $A \times P_2(B) = P_2(A \times B)$

Finally, we establish that the result $A \times B$ is same if the columns of A matrix and the rows of B matrix are permuted using the same permute function and they are multiplied with each other.

Lemma 3.

$$A \times B = P_2(A) \times P_1(B)$$

Proof: Let $C'_{i,j}$ represent the (i, j) of the product $P_2(A) \times P_1(B)$. Then

$$C'i, j = \sum_{k=1}^{n} A_{i,p_k} \cdot B_{p_k,j} = C_{i,j} \tag{2}$$

This merely changes the order in which the elements are accumulated, leaving the overall result unchanged.

Finally we show that our *EQUIP* approach which permutes the K and V tensor to avoid the copy overhead results in the same attention output as the shift-and-append approach. We note here that our *EQUIP* approach permutes the K and V tensors in the second dimension in the same way. Hence the permutation operation P_2 is same for both.

Theorem 1. *Attention* (Q, K, V) = *Attention* $(Q, P_2(K), P_2(V))$

Proof: With our *EQUIP* approach, both the K and V tensors are permuted in the sequence dimension, i.e., $P_2(K)$ and $P_2(V)$. The attention computation performed is:

$$\text{Attention}(Q, P_2(K), P_2(V)) = \Big(\text{Softmax}\Big(\frac{Q \times (P_2(K))^T}{\sqrt{C'}}\Big)\Big) \times P_2(V) \quad (3)$$

In the SDPA computation, the first dimension of Q, K and V tensors is for batch size and number of heads. In the batched matrix multiplication K^T results in a tensor of dimension $[B{\cdot}H, d, n]$. Hence $(P_2(K))^T$ permutes the sequence dimension or dimension 3 in the transposed matrix. Thus $(P_2(K))^T = (P_3(K^T)$. Thus the first term in the attention computation becomes $Q \times P_3(K^T)$. From Lemma 2, we can rewrite this as $P_3(Q \times K^T)$. Lemma 1, establishes

$$Softmax(P_3(Q \times K^T)) = P_3(Softmax(Q \times K^T).$$

Finally when $P_3(Softmax(Q \times K^T)$ is multiplied with $P_2(V)$, from Lemma 3, the result is the same as

$$Softmax(Q \times K^T) \times V,$$

as the accumulation is done on third and second dimensions of K and V tensors, respectively. .

We now illustrate the actual dimension and permuted rows/columns of the tensors in the SDPA computation.

$$\Big(\text{Softmax}\Big(\frac{Q \times (P_2(K))^T}{\sqrt{C'}}\Big)\Big) \times P_2(V) \quad (4)$$

$$= \text{Softmax}\Big(\frac{Q_{BH,1,d} \times K^T_{BH,d,p(n)}}{\sqrt{C'}}\Big) \times V_{BH,p(n),d} \quad (5)$$

This can be rewritten as:

$$= \text{Softmax}\Big(\frac{QK^T_{BH,1,p(n)}}{\sqrt{C'}}\Big) \times V_{BH,p(n),d} \quad (6)$$

Substituting $Q \times K^T$ as W, we get

$$= \text{Softmax}\Big(\frac{W_{BH,1,p(n)}}{\sqrt{C'}}\Big) \times V_{BH,p(n),d} \quad (7)$$

If applying Softmax on W and performing element-wise normalization result in a new matrix

$$= S_{BH,1,p(n)} \times V_{BH,p(n),d} = SV_{BH,1,d} \quad (8)$$

Theorem 1 establishes that the result of the SDPA computation[6] performed after the in-place update is identical to that obtained by the token-pruning methods with the shift-and-append operation.

We proved in Theorem 1 that for any consistent permutation P_2 of the sequence axis,

$$\text{Attention}(Q, K, V) = \text{Attention}\big(Q, P_2(K), P_2(V)\big) \quad (9)$$

C.2 RoPE Equivalence

We now extend the proof of equivalence for SDPA computation with RoPE. Figure 18 depicts the RoPE computation implementation in *EQUIP*.

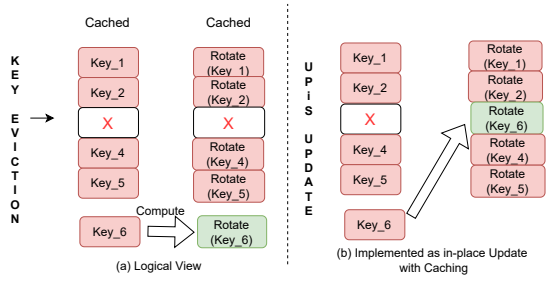


Figure 18: Rotate Keys: Evict-and-Insert implemented as in-place operation with *EQUIP*.

Lemma 4. *For tensors A and B of the same shape and a permutation P_2 applied along the sequence axis,*

$$P_2(A + B) = P_2(A) + P_2(B)$$

[6]Note: While the proof is presented for Multi-Head Attention (MHA), its underlying assumptions naturally extend to Multi-Query Attention (MQA), General-Query Attention (GQA), and Multi-Latent Attention (MLA).

Proof RoPE-transformed key vector for a single token as the sum of two components:

$$\mathrm{RoPE}(k) = k' = k'_1 + k'_2, \tag{10}$$

$$\text{where} \quad k'_1 = k \odot \cos(p\theta) \quad \text{and}$$
$$k'_2 = \mathrm{Rotate\_Half}(k) \odot \sin(p\theta).$$

Because P_2 permutes only the sequence axis and is applied identically to K and V, the equivariance of attention in Eq. (9) applies to each linear component individually:

$$\mathrm{Attention}\big(Q, P_2(k'_1), P_2(V)\big) = \mathrm{Attention}\big(Q, k'_1, V\big), \tag{11}$$

$$\mathrm{Attention}\big(Q, P_2(k'_2), P_2(V)\big) = \mathrm{Attention}\big(Q, k'_2, V\big). \tag{12}$$

Using Lemma 4 and summing the two equalities establish Equation 9 even with ROPE embedding.

The correctness proof is under the assumption that the per-position RoPE coefficients $cos(\theta), sin(\theta)$ must be re-indexed correctly under P2 (equivalently, rotation coefficients are indexed by logical position as described in Figure 5).

C.3 Attention Mask and Attention Bias (ALiBI) Equivalence

We now extend Eq. (9) to masked attention. Masked attention is defined as

$$\mathrm{Attention}(Q, K, V, M) = \tag{13}$$

$$\mathrm{Softmax}\Big(\Big(\frac{QK^\top}{\sqrt{d}} + M\Big)V\Big), \tag{14}$$

where M is a tensor. We claim

Theorem 2.

$$\mathrm{Attention}\big(Q, K, V, M\big) = \mathrm{Attention}\big(Q, P_2(K), P_2(V), P_2(M)\big) \tag{15}$$

where $P_2(M)$ denotes the mask permuted by P_2

Proof. Let $L = (QK^\top)/\sqrt{d}$. Permuting K by P_2 permutes the sequence dimension of L. From Lemma 4

$$P_2(L + M) = P_2(L) + P_2(M).$$

Softmax applied along the key axis is equivariant to column permutations:

$$\mathrm{Softmax}\big(P_2(X)\big) = P_2\big(\mathrm{Softmax}(X)\big)$$

when P_2 permutes columns of X. Hence

$$\mathrm{Softmax}\big(P_2(L + M)\big) = \mathrm{Softmax}\big(P_2(L) + P_2(M)\big)$$
$$= P_2\big(\mathrm{Softmax}(L + M)\big).$$

Multiplying by $P_2(V)$ and using the permutation properties of matrix multiplication yields Eq. (15).

D Performance of *EQUIP* under Other Scenarios

We present additional experimental results in this section.

D.1 Scalability across models

Table 3 presents the end-to-end speedup of *EQUIP*_H2O over H2O for GPT-J and GPT-NeoX across a range of sequence lengths and cache sizes. *EQUIP* helps to achieve 1.25× — 1.52× performance improvement even in these LLMs on Intel SPR.

Table 3: *EQUIP*-H2O Speedup - Sequence Length=2048, Batch Size=8.

Cache Size	Seq len=1024		Seq len=2048	
	GPTJ 6B	GPT-Neox 20B	GPTJ 6B	GPT-Neox 20B
512	1.25×	1.27×	1.45×	1.41×
768	1.28×	1.30×	1.52×	1.43×

D.2 Performance on Llama-3 Models

We further validate *EQUIP* on the newer Llama-3-8B model on Intel SPR (cache size=512, sequence length=1536). *EQUIP* achieves speedups of 1.30×, 1.43×, and 1.52× at batch sizes 8, 16, and 32 respectively, confirming that the approach generalizes to recent model architectures.

D.3 Multi-Instance and Multi-Core Scaling

Our experimental results reveal that *EQUIP*_StrLLM achieves considerable end to end speedup (1.25× – 1.7×) over StreamingLLM even when multiple concurrent instances of the inference engines were run on all 60 cores of the SPR server. Further, *EQUIP*_H2O achieves a sustained performance improvement across core counts 16, 32, 48 and 60.

D.4 Performance on NVIDIA A100 GPUs

To demonstrate hardware generalizability, we evaluate *EQUIP* on NVIDIA A100 (40 GB VRAM)

GPUs. Table 4 reports the performance of *EQUIP* over StreamingLLM for Llama 2-7B with a pruned KV cache size of 512 and maximum sequence length of 2048. *EQUIP* consistently outperforms StreamingLLM across all batch sizes, with speedups reaching up to 1.58× at BS=32.

Table 4: *EQUIP* speedup over StreamingLLM on NVIDIA A100 (Llama 2-7B, cache size=512, seq. length=2048).

Batch Size	Throughput of StrLLM (tokens/sec)	Throughput of EQUIP_StrLLM (tokens/sec)	Speedup
8	195	226	1.15×
16	253	353	1.39×
32	289	457	1.58×

D.5 Inter-Token Latency

Beyond throughput, we report the inter-token latency for tokens generated once the pruned KV cache is full. On Intel SPR, for Llama 2-7B model (BS=8), as the pruned cache size increases from 512 to 768 to 1024 tokens, *EQUIP* demonstrates reduction in progressive inter-token latency by 1.53×, 1.74×, and 1.93× respectively for a sequence length of 8192. These results confirm that *EQUIP*'s in-place updates translate to measurable per-token latency reduction, not only aggregate throughput gains.

D.6 Impact of NoPE

How well does *EQUIP* perform in models (such as OPT) that do not have positional encoding? For this, we conduct experiments with RoPE (Rotary Positional Encoding) disabled in Llama models. This configuration, termed No Positional Encoding (NOPE), was applied to the baseline model (utilizing StreamingLLM) and our *EQUIP* framework (*EQUIP*_StrLLM). Table 5 presents the speedup results for a sequence length of 2048 under this NoPE condition. The findings demonstrate that *EQUIP*'s efficient KV cache update mechanism contributes to improved performance even in the absence of RoPE and increases with the pruned cache size.

Table 5: *EQUIP* Speedup with NoPE on MI210 (Sequence Length=2048, Batch Size=8).

Models	KV cache Size $(s+r)$		
	512	768	1024
Llama 2 7B	1.182	1.253	1.284
Llama 2 13B	1.216	1.295	1.326

D.7 Multiple Evictions and Insertions

Many real-time and streaming scenarios require to evict and insert several KV entries at once (e.g., in Speculative Decoding (Chen et al., 2024) and in real-time code edits (He et al., 2024b)). Hence we evaluated *EQUIP* under this scenario, where multiple evict-and-insert are implemented as in-place update. In this experiment, we measured the improvement in KV cache update latency for the two pruning schemes –H2O and StreamingLLM – across a range of batch sizes, attention heads, and head dimensions. In all experiments, we fixed the pruned cache capacity ($(s+r)$) at 1024 entries and evicted 64 tokens per generation step. Table 6 reports the resulting speedups of *EQUIP* over the respective baseline kernels. Across configurations, *EQUIP* consistently achieves considerable acceleration (3.29× – 39.90× for H2O and 3.58× – 50.58× for StreamingLLM) on both CPU and GPU. We anticipate that these gains will translate to reasonable speedups in end-to-end latencies as well.

Table 6: KV Update Speedup with *EQUIP* on MI210 and SPR (Cache size =1024, Evict=64 tokens).

Batch Size	Heads	Head Size	Speedup over H2O		Speedup over StrLLM	
			on MI210	on SPR	on MI210	on SPR
1	64	64	3.29	11.62	3.95	26.54
1	64	128	8.22	27.32	4.07	36.04
1	128	64	8.79	19.36	3.58	27.65
8	64	64	8.13	35.73	4.53	47.94
8	64	128	8.04	38.12	6.00	46.67
8	128	64	8.67	39.90	7.35	45.04
32	64	64	6.98	39.75	6.69	50.58
32	64	128	10.7	31.43	8.40	47.34

D.8 Comparison with other pruning methods

Next we evaluate the benefits of EQUIP on other pruning methods, such as LagKV (Liang et al., 2025) and SnapKV (Li et al., 2024a). We evaluate the performance gains due to *EQUIP*(over the respective baseline method) for attention kernel (SDPA) alone for a batch size of 128 and sequence length of 1024, and cache size of 512. *EQUIP* demonstrates scalability across models and token-pruning techniques (refer to Table 7).

Table 7: *EQUIP* Attention Speedup on MI210 - Sequence Length=1024, Batch Size=128, Cache Size=512.

Models	*EQUIP* StrLLM	*EQUIP* LagKV	*EQUIP* SnapKV
Llama 2 7B	7.42	4.06	2.89
OPT 13B	5.58	3.36	1.80

D.9 Overhead with Re-Indexing

We clarify that neither SDPA computation nor KV updates in *EQUIP* incur any reindexing overhead. Reindexing is only required when: (1) complex saliency metrics demand access to adjacent positional indices, or (2) gathering cosine/sine parameters for RoPE.

We also evaluate the efficiency of Rotary Positional Encoding (RoPE) under standard contiguous positional indices and several non-contiguous indexing schemes induced by *EQUIP* KV cache management. The index patterns considered are: (1) contiguous indices, as in sliding-window attention; (2) random but shared non-contiguous indices that are consistent across all sequences in a batch; and (3) random batching, where non-contiguous indices vary independently for each sequence in the batch.

Our empirical results show that RoPE's computational efficiency is effectively invariant to the choice of index pattern. Profiling indicates that the dominant cost lies in the element-wise rotary operations, rather than in the index-dependent retrieval of positional parameters. In particular, the gather operations required to fetch parameters for scattered indices account for less than 3% of the total RoPE execution time.

Table 8: Efficiency of RoPE with *EQUIP*. All values are normalized to the contiguous-index baseline.

BS	Heads	Seq Len	Contig.	Rand.Bat.	Rand
1	32	4096	1.00	0.91	0.96
1	32	16384	1.00	1.01	0.99
1	64	4096	1.00	1.02	0.99
1	64	16384	1.00	1.01	0.99
32	32	4096	1.00	1.00	1.00
32	32	16384	1.00	0.99	1.00
32	64	4096	1.00	1.00	1.01
32	64	16384	1.00	0.96	0.97
128	32	4096	1.00	1.04	1.00
128	32	16384	1.00	1.00	1.04
128	64	4096	1.00	1.01	1.00
128	64	16384	1.00	1.02	1.10
geo-mean			1.00	1.00	0.99

D.10 Accuracy

EQUIP in-place update mechanism inherently preserves the same model accuracy as the baseline token-pruning method. Our experiments on Llama 7B and 13B models demonstrate exact perplexity parity and token match for all generated tokens. Figure 19 shows perplexity match on Llama 2-7B model for sequence length up to 8192. At layer-wise precision for FP32/BF16, maximum deviations are less than 10^{-9} at SDPA and less than 10^{-5} at RoPE outputs; deviations arise solely from accumulation-order rounding.

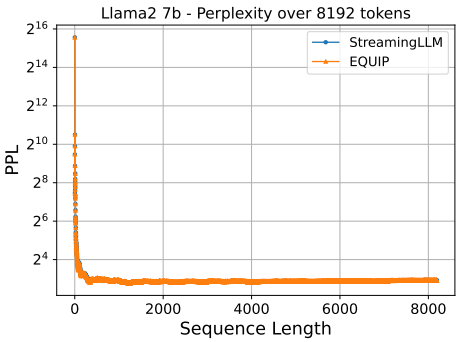


Figure 19: Llama2-7B Perplexity.

Table 9: Perplexity comparison: StreamingLLM vs. *EQUIP* on Llama 2-7B (a WikiText-2 training example, sequence length 2048 tokens) (Xiao et al., 2024). Cache config s+r denotes s sink tokens and r recent tokens.

Cache Config	StreamingLLM	*EQUIP*
4+508 (512)	11.19	11.19
4+1020 (1024)	10.79	10.79
4+2044 (1536)	10.19	10.19

D.11 Memory Overhead

Caching rotated key (RK) increases the memory requirement somewhat moderately. More specifically, the increase in memory requirement is 50% relative to the bounded (pruned) KV cache (not the full unpruned cache). When measured against the total memory footprint (model weights + KV cache + activations), this corresponds to only 5–15% overhead across tested configurations. Concretely, the footprint for Llama 2-7B increases from 16.15 GB to 17.1 GB (6% increase), and for Llama 2-30B from 64.07 GB to 65.7 GB at BS=8 (2% increase).