InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management

Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim,
Seoul National University

https://www.usenix.org/conference/osdi24/presentation/lee

This paper is included in the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-40-3

Open access to the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management

Wonbeom Lee[†] Jungi Lee[†] Junghwan Seo Jaewoong Sim
Seoul National University

Abstract

Transformer-based large language models (LLMs) demonstrate impressive performance across various natural language processing tasks. Serving LLM inference for generating long contents, however, poses a challenge due to the enormous memory footprint of the transient state, known as the key-value (KV) cache, which scales with the sequence length and batch size. In this paper, we present InfiniGen, a novel KV cache management framework tailored for long-text generation, which synergistically works with modern offloading-based inference systems. InfiniGen leverages the key insight that a few important tokens that are essential for computing the subsequent attention layer in the Transformer can be speculated by performing a minimal rehearsal with the inputs of the current layer and part of the query weight and key cache of the subsequent layer. This allows us to prefetch only the essential KV cache entries (without fetching them all), thereby mitigating the fetch overhead from the host memory in offloading-based LLM serving systems. Our evaluation on several representative LLMs shows that InfiniGen improves the overall performance of a modern offloading-based system by up to 3.00× compared to prior KV cache management methods while offering substantially better model accuracy.

1 Introduction

Large language models (LLMs) have opened a new era across a wide range of real-world applications such as chatbots [40, 76], coding assistants [11, 43], language translations [1, 68], and document summarization [64, 74]. The remarkable success of LLMs can largely be attributed to the enormous model size, which enables effective processing and generation of long contents. For instance, while the maximum sequence length of the first version of GPT was restricted to 512 tokens [51], the latest version, GPT-4, can handle up to 32K tokens, which is equivalent to approximately 50 pages of text [3]. Some recently announced models such as Claude 3 [6] and Gemini 1.5 [53] can even process up to 1 million tokens, significantly expanding the context window by several orders of magnitude.

In addition to the well-studied challenge of the model size, deploying LLMs now encounters a new challenge due to the substantial footprint of the transient state, referred to as the *key-value (KV) cache*, during long context processing and generation. For generative LLM inference, the keys and values of all preceding tokens are *stored* in memory to avoid redundant and repeated computation. Unlike the model weights, however, the KV cache scales with the output sequence length, often consuming even more memory capacity than the model weights. As the demand for longer sequence lengths (along with larger batch sizes) continues to grow, the issue of the KV cache size will become more pronounced in the future.

Meanwhile, modern LLM serving systems support offloading data to the CPU memory to efficiently serve LLMs within the hardware budget [5, 57]. These offloading-based inference systems begin to support even offloading the KV cache to the CPU memory, thereby allowing users to generate much longer contexts beyond the GPU memory capacity. However, transferring the massive size of the KV cache from the CPU memory to the GPU becomes a new performance bottleneck in LLM inference.

In this work, we propose InfiniGen, a KV cache management framework designed to synergistically work with modern offloading-based inference systems. InfiniGen builds on two key design principles. First, it speculates and chooses the KV cache entries that are critical to produce the next output token, dropping the non-critical ones, by conducting a minimal *rehearsal* of attention computation for Layer i at Layer $i-1$. Second, it leverages the CPU memory capacity and maintains the KV cache pool on the CPU, rather than on the GPU, to ensure that the critical KV cache values can be identified for all outputs and layers with a large *window size* while alleviating the concerns about limited GPU memory capacity for long content generation.

In particular, InfiniGen manipulates the model weights *offline* to make the speculation far more efficient and precise, by

[†]Equal contribution

skewing the Transformer architecture query and key matrices to emphasize certain important columns. During the prefill stage, while the prompt and input of an inference request are initially processed, InfiniGen generates *partial* weights for use in the subsequent decoding (i.e., output generation) stage. At Layer $i-1$ of the decoding stage, InfiniGen speculates on the attention pattern of the next layer (Layer i) using the attention input of Layer $i-1$, a partial query weight, and a partial key cache of Layer i. Based on the speculated attention pattern, InfiniGen prefetches the essential KV cache entries from the CPU memory for attention computation at Layer i. By dynamically adjusting the number of KV entries to prefetch, InfiniGen brings only the necessary amount of the KV cache to the GPU, thereby greatly reducing the overhead of the KV cache transfer. In addition, InfiniGen manages the KV cache pool by dynamically removing the KV cache entries of infrequently used tokens.

We implement InfiniGen on a modern offloading-based inference system [57] and evaluate it on two representative LLMs with varying model sizes, batch sizes, and sequence lengths. Our evaluation shows that InfiniGen achieves up to a 3.00× speedup over the existing KV cache management methods while offering up to a 32.6 percentage point increase in accuracy. In addition, InfiniGen consistently provides performance improvements with larger models, longer sequence lengths, and larger batch sizes, while prior compression-based methods lead to saturating speedups.

In summary, this paper makes the following contributions:

- We present InfiniGen, a dynamic KV cache management framework that synergistically works with modern offloading-based LLM serving systems by intelligently managing the KV cache in the CPU memory.
- We propose a novel KV cache prefetching technique with ephemeral pruning, which speculates on the attention pattern of the subsequent attention layer and brings only the essential portion of the KV cache to the GPU while retaining the rest in the CPU memory.
- We implement InfiniGen on a modern offloading-based inference system and demonstrate that it greatly outperforms the existing KV cache management methods, achieving up to 3.00× faster performance while also providing better model accuracy.

2 Background

This section briefly explains the operational flow and the KV caching technique of large language models and introduces the singular value decomposition (SVD) as a method of skewing matrices for a better understanding of our proposed framework, which we discuss in Section 4.

2.1 Large Language Models

Large language models (LLMs) are composed of a stack of Transformer blocks, each of which contains an attention layer followed by a feed-forward layer [61]. The input tensor (X) of the Transformer block has a dimension of $N \times D$, where N is the number of query tokens, and D is the model dimension. This input tensor (X) is first layer-normalized (LayerNorm), and the layer-normalized tensor (X_a) is fed into the attention layer as input. The attention input (X_a) is multiplied by three different *weight* matrices (W_Q, W_K, W_V) to generate Query (Q), Key (K), and Value (V) matrices. Each weight matrix has a dimension of $D \times D$. Thus, Query, Key, and Value have a dimension of $N \times D$. These matrices are reshaped to have a dimension of $H \times N \times d$, where H is the number of attention heads and d is the head dimension; note that $D = H \times d$.

Each head individually performs attention computation, which can be formulated as follows: $\text{softmax}(QK^T)V$.[1] The attention output, after a residual add (adding to the input tensor X) and layer normalization, is fed into the feed-forward layer. The feed-forward network (FFN) consists of two consecutive linear projections and a non-linear activation operation between them. The output of FFN after a residual add becomes the output of a Transformer block, which has the *same* dimensionality as the input of the Transformer block (i.e., $N \times D$). This allows us to easily scale LLMs by adjusting the number of Transformer blocks.

2.2 Generative Inference and KV Caching

Generative LLM inference normally involves two key stages: the *prefill* stage and the *decoding* stage. In the prefill stage, LLMs summarize the context of the input sequence (i.e., input prompt) and produce a *new* token that serves as the initial input for the decoding stage. Subsequently, using this new token, LLMs run the decoding stage to generate the next token. The newly generated token is then fed back into the decoding stage as input, creating an *autoregressive* process for token generation. In this work, we refer to each token generation in the decoding stage as an *iteration*.

To generate a new token that aligns well with the context, LLMs need to compute the relationship between the last token and all the previous ones, including the tokens from the input sequence, in the attention layer. A naïve approach to this is to recompute the keys and values of all the previous tokens at every iteration. However, this incurs a significant overhead due to redundant and repeated computation. Furthermore, the computation overhead linearly grows with the number of the previous tokens; i.e., the overhead becomes larger for longer sequences.

To avoid such overhead, the keys (K) and values (V) of all the previous tokens are typically *memoized* in memory,

[1] In this work, we refer to the results of QK^T and $\text{softmax}(QK^T)$ as *attention scores* and *attention weights*, respectively.

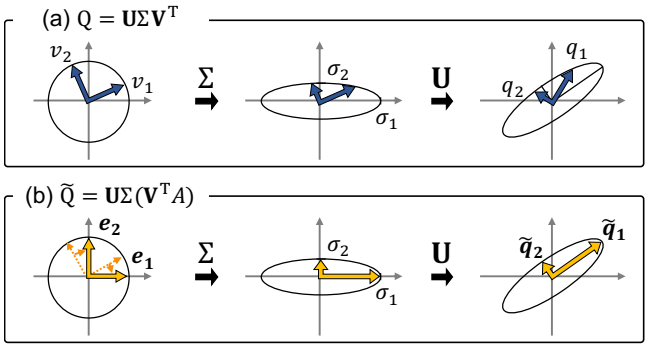

Figure 1: Transformation from matrix $\mathbf{V}^T$ to matrix Q in terms of SVD. The orthogonal matrix A maximizes the difference in magnitude between the column vectors of Q.

which is known as the *KV cache*. The KV cache then keeps updated with the key and value of the generated token at each iteration. As such, the dimension of the KV cache at the i-th iteration can be expressed as $H \times (N+i) \times d$. If batched inference is employed, the size of the KV cache also grows linearly to the batch size. By employing the KV cache, we can avoid repeated computation and produce the key and value of only one token at each iteration. Note that in the decoding stage, the input to the Transformer block (X) has a dimension of $1 \times D$, and the dimension of the attention score matrix becomes $H \times 1 \times (N+i)$ at the i-th iteration.

2.3 Outliers in Large Language Models

Large language models have outliers in the Transformer block input tensors. The outliers refer to the elements with substantially larger magnitudes than the other elements. The outliers in LLMs appear in a few fixed channels (i.e., columns in a 2D matrix) across the layers. Prior work has shown that outliers are due to the intrinsic property of the model (e.g., large magnitudes in a few fixed channels of layer normalization weights) [19, 65].

2.4 Singular Value Decomposition

We observe that skewing the query and key matrices to make a small number of channels much larger than others and using only those channels to compute the attention score matrix can effectively predict which tokens are important. In essence, we multiply the Q and K matrices with an orthogonal matrix A to make it align with the direction that Q stretches the most, to produce the respective skewed matrices $\tilde{Q}$ and $\tilde{K}$. We explain in detail why we use an orthogonal matrix in Section 4.2.

To find such an orthogonal matrix A, we employ the singular value decomposition (SVD), which is a widely used matrix factorization technique in linear algebra. For a real matrix Q of size $m \times n$, its SVD factorization can be expressed as follows:

$$Q = \mathbf{U}\Sigma\mathbf{V}^T,$$

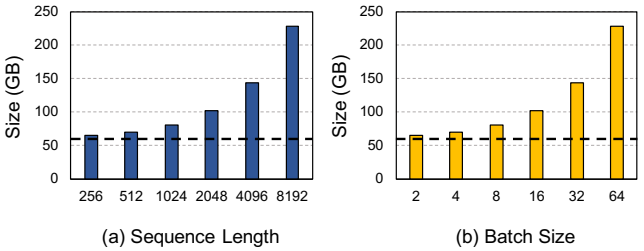

Figure 2: Total size of the KV cache and model weights of OPT-30B for different sequence lengths and batch sizes. The batch size of (a) is 16, and the sequence length of (b) is 2048. The dotted line represents the size of the model weights.

where $\mathbf{U}$ and $\mathbf{V}$ are orthogonal matrices of size $m \times m$ and $n \times n$, respectively.[2] Σ is an $m \times n$ diagonal matrix, which has nonzero values $(\sigma_1, \sigma_2, ..., \sigma_k)$ on the diagonal, where $k = \min(m,n)$. In terms of linear transformation, it is well known that a transformation of a vector $v \in \mathbb{R}^n$ by a real matrix B (i.e., the product of B and v) is a rotation and/or reflection in $\mathbb{R}^n$ if the B matrix is orthogonal. If B is an $m \times n$ diagonal matrix, each dimension of v is stretched by the corresponding diagonal entry of B and is projected to $\mathbb{R}^m$.

For example, Figure 1 shows how the column vectors v_1 and v_2 of $\mathbf{V}^T$ would transform to column vectors q_1 and q_2 of Q, when m and n are 2. In Figure 1(a), the orthogonal unit vectors v_1 and v_2 are first stretched to the points on an ellipse whose semi-axis lengths correspond to the diagonal entries in Σ. The vectors are then rotated and/or reflected to q_1 and q_2 by matrix $\mathbf{U}$. On the other hand, Figure 1(b) shows how orthogonal matrix A performs rotation to make the resulting $\tilde{q}_1$ much larger than $\tilde{q}_2$. Specifically, A rotates vectors v_1 and v_2 to e_1 and e_2, which map to the semi-axes of the ellipse. In this way, the vectors are stretched to the maximum and minimum by the matrix Σ. This process emphasizes the magnitude of $\tilde{q}_1$ over $\tilde{q}_2$, which allows us to effectively predict the attention score using only $\tilde{q}_1$ while omitting $\tilde{q}_2$.

3 Motivation

In this section, we first explain that the KV cache size becomes a critical issue for long-text generation in LLM inference, and it becomes more problematic when deploying modern offloading-based inference systems (Section 3.1). We then discuss why the existing KV cache management methods cannot fundamentally address the problem in the offloading-based inference system (Section 3.2).

3.1 KV Cache in LLM Inference Systems

As discussed in Section 2.2, today's LLM serving systems exploit KV caching to avoid redundant computation of key and

[2] Note that this $\mathbf{V}$, typeset with a different font, is one of the resulting matrices of SVD and is distinct from the V of the Value matrix in the Transformer attention layer.

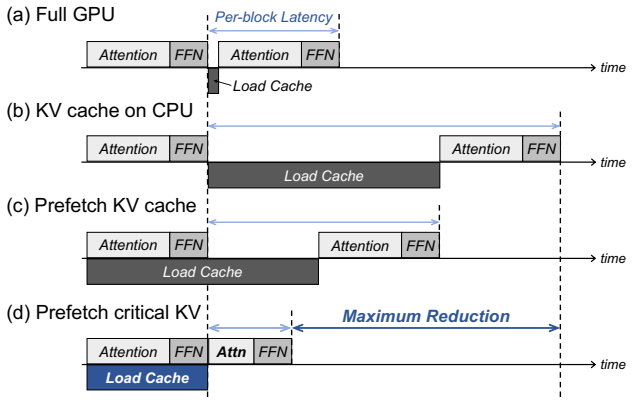

Figure 3: Comparison between different execution styles of Transformer blocks.

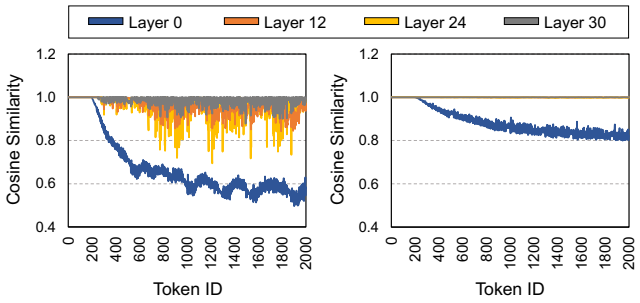

Figure 4: Cosine similarity between the attention weights of the base model with full cache and (a) H_2O or (b) Optimal. H_2O and Optimal use 200 tokens for attention computation. We use OPT-6.7B and a random sentence with 2000 tokens from the PG-19 dataset [52].

value projections during the decoding stage. While this is an effective solution for short sequence generation with a single client request, the KV cache quickly becomes a key memory consumer when we generate long sequences or employ modern request batching techniques [57, 71].

Figure 2 shows the combined size of LLM weights and the KV cache across different sequence lengths and batch sizes. As depicted in the figure, the model size remains constant regardless of sequence lengths or batch sizes, whereas the KV cache size linearly scales with them. Note that modern LLM serving systems, such as NVIDIA Triton Inference Server [45] and TensorFlow Serving [47], already support *batched* inference for better compute utilization and higher throughput in serving client requests. When individual requests are batched, each request retains its own KV cache, thereby increasing the overall KV cache size for the inference. Even for a single client request, beam search [59] and parallel sampling [20] are widely used to generate better outputs or to offer clients a selection of candidates [11, 24]. The techniques also increase the size of the KV cache like batched inference as multiple sequences are processed together. Consequently, the KV cache size can easily exceed the model size for many real-world use cases, as also observed in prior work [37, 49, 57, 78]. This can put substantial pressure on GPU memory capacity, which is relatively scarce and expensive.

LLM Inference Systems with Offloading. Modern LLM serving systems such as DeepSpeed [5] and FlexGen [57] already support offloading the model weights or the KV cache to the CPU memory. When it comes to offloading-based inference systems, the KV cache size becomes more problematic due to the low PCIe bandwidth between the CPU and GPU, which becomes a new and critical bottleneck.

Figure 3 depicts a high-level timing diagram between different execution styles of Transformer blocks. Figure 3(a) represents the case when the KV cache entirely resides in the GPU memory (Full GPU). In this case, the load latency of the KV cache (Load Cache) involves a simple read operation from the GPU memory, which is negligible due to the high bandwidth of GPU memory. However, the maximum batch size or sequence length is limited by the GPU memory capacity, which is relatively smaller than the CPU memory.

To enable a larger batch size or a longer sequence length, we can offload the KV cache to CPU memory (KV cache on CPU), as shown in Figure 3(b). While offloading-based inference systems alleviate the limitation on the batch size and sequence length, transferring hundreds of gigabytes of the KV cache to the GPU for attention computation significantly increases the overall execution time of Transformer blocks due to the limited PCIe bandwidth.

Even when we apply a conventional prefetching technique (Prefetch KV cache), as shown in Figure 3(c), only part of the load latency can be hidden by the computation of the preceding Transformer block. Note that although compressing the KV cache via quantization could potentially reduce the data transfer overhead in offloading-based systems [57], it does not serve as a fundamental solution as quantization does not address the root cause of the KV cache problem, which is the linear scaling of KV entries with the sequence length. This necessitates intelligent KV cache management to mitigate the performance overhead while preserving its benefits.

3.2 Challenges in KV Cache Management

The fundamental approach to mitigating the transfer overhead of the KV cache from the CPU to GPU is to reduce the volume of the KV cache to load by identifying the *critical* keys and values for computing attention scores, as shown in Figure 3(d). It is widely recognized that the keys and values of certain tokens are more important than others in attention computation [9, 10, 14, 33, 63]. As explained in Section 2.1, after computing the attention score, the softmax operation is applied, which emphasizes a few large values of tokens. Therefore, skipping attention computation for some less critical tokens does not significantly degrade the model accuracy, provided the token selection is appropriate.

In this context, several recent works propose to reduce

the KV cache size through key/value evictions at runtime within a constrained KV cache budget [37, 78]. However, all the prior works assume the *persistence* of attention patterns across iterations; that is, if a token is deemed unimportant in the current iteration (i.e., having a low attention weight), it is likely to remain unimportant in the generation of future tokens. Under the assumption, they evict the tokens with a low attention weight from the KV cache at each iteration when the KV cache size exceeds its budget. The keys and values of the evicted tokens are *permanently* excluded from the subsequent iterations while being removed from the memory. Although the recent works on managing the KV cache can be applied to offloading-based inference systems, we observe that they do not effectively address the challenges in KV cache management below and thus have subpar performance with offloading-based inference systems.

C1: Dynamic nature of attention patterns across iterations. Figure 4 shows the cosine similarity between the attention weights of the baseline model, which uses the KV cache of all prior tokens for computing attention weights (i.e., a maximum of 2000 tokens in the experiment), and two different KV cache management methods (H_2O and Optimal) with a KV cache budget of 200 tokens.[3] H_2O [78] is a state-of-the-art technique that retains only a small percentage of important tokens in the KV cache to reduce its size. It *assesses* the importance of each token in every iteration and *removes* unimportant ones before the next iteration to keep the KV cache size in check (i.e., using a narrow assessment window). In contrast, Optimal represents the scenario where we choose the same number of tokens as H_2O from the KV cache at each iteration but retain all prior keys and values (i.e., using a wider assessment window). In other words, Optimal selects 200 tokens out of the entire sequence of previous tokens at each iteration.

The figure indicates that despite H_2O-like approaches assuming that the attention pattern does not change across iterations, this is not the case in practice. The tokens deemed unimportant in the current iteration could become important in subsequent iterations. Consequently, H_2O exhibits high similarity until around 200 iterations (i.e., within the KV cache budget), but as the sequence length extends beyond the KV cache budget, it starts to struggle with the dynamic nature of the attention pattern, resulting in lower cosine similarity than the Optimal case. Note that while we only show the scenario of a KV cache budget of 200 out of a total sequence length of 2000 tokens for brevity, this issue would become more pronounced as the sequence length surpasses it.

C2: Adjusting the number of KV entries across layers. Figure 4 also illustrates that the impact of the KV cache eviction *varies* across the layers in LLMs. For Layer 0, both H_2O and Optimal show a significant drop in cosine similarity as the token ID increases. This implies that Layer 0 has a *broader* attending pattern than other layers; i.e., the attention weights are relatively similar between key tokens. Thus, the selected 200 tokens with the large attention weight do *not* adequately represent the attention pattern of the baseline model for this layer, as they are likely only slightly larger than the others, not strongly so. In such cases, it becomes necessary to compute the attention weight with a larger number of tokens.

[3]The cosine similarity measures how much each row of the attention weight is similar to the case of the full KV cache. If they are similar, the generated tokens will also be similar. Thus, a low cosine similarity indicates low accuracy far from the baseline model with the full KV cache.

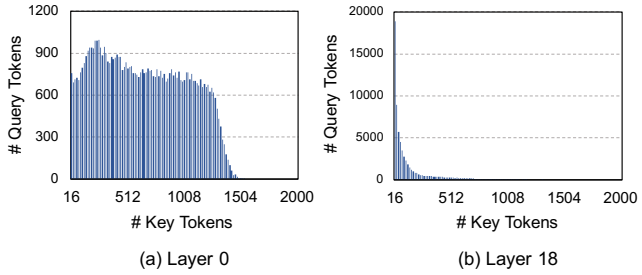

Figure 5: Histogram that shows the number of key tokens needed to achieve 0.9 out of 1.0 total attention weight for (a) Layer 0 and (b) Layer 18 of the OPT-6.7B model. The bin width is set to 16. We observe that the distribution dynamically changes across the layers.

To estimate how many keys/values from the KV cache need to be retained, we sort the attention weight for each query token in descending order and sum the key tokens until the cumulative weight reaches 0.9. Figure 5 presents a histogram of the number of query tokens (y-axis) requiring the number of key tokens (x-axis) needed to reach a weight of 0.9 (out of the total attention weight of 1.0) in two different layers: Layer 0 and Layer 18. Layer 0 shows a broad distribution, indicating a significant variation in the number of key tokens required to achieve a weight of 0.9 for each query token. In contrast, Layer 18 exhibits a highly skewed distribution, suggesting that the majority of the query tokens in this layer require only a few key tokens to reach a weight of 0.9. This implies that we need to *dynamically* adjust the number of key tokens participating in attention computation across different layers to make efficient use of the KV cache budget.

C3: Adjusting the number of KV entries across queries. H_2O sets the number of key/value tokens to retain as a fixed percentage of the input sequence length. The KV cache budget remains constant regardless of how many tokens have been generated. By analyzing the data from Figure 5 on Layer 18, we observe that this fixed KV cache budget has some limitations. For instance, with an input sequence length of 200 and a 20% KV cache budget, H_2O maintains 40 key/value tokens throughout token generations. However, most of the subsequent query tokens require more than 40 tokens to effectively represent the attention weight of the baseline model; for example, the 500^{th}, 1000^{th}, 1500^{th}, and 2000^{th} tokens need 80, 146, 160, and 164 key tokens, respectively, to reach a cumulative attention weight of 0.9. This implies an inadequate amount of the key/value tokens to properly represent the attention weight of the baseline model. Furthermore, the

number of key tokens required to reach 0.9 varies even for the adjacent query tokens; for instance, the 998th, 999th, 1000th, 1001st, and 1002nd tokens need 172, 164, 146, 154, and 140 key tokens, respectively. Fixing the KV cache budget without accounting for the variance between query tokens inevitably results in ineffective KV cache management. Therefore, we need to *dynamically* adjust the amount of the key/value tokens loaded and computed for each query token to efficiently manage the KV cache.

Summary. Prior works aiming to reduce the KV cache size through token eviction inherently have some challenges. Given the dynamic attention pattern across iterations, permanently excluding evicted tokens from future token generation can result in a non-negligible drop in accuracy. Instead, we need to dynamically *select* critical tokens from the KV cache while avoiding the outright eviction of less important ones. Furthermore, the fixed size of the KV cache budget in prior works leads to inefficient KV cache management. The number of key/value tokens required for each layer differs, and each query token demands a varying number of key/value tokens to effectively represent the attention pattern of the baseline model. Failing to account for these variations may result in ineffective KV cache management. Thus, we need to dynamically adjust the number of key/value tokens to select from the KV cache while considering the variances between layers and query tokens.

4 InfiniGen Design

In this section, we present InfiniGen, a KV cache management framework for offloading-based inference systems. We first show the high-level overview of our proposed KV cache management solution (Section 4.1) and discuss the opportunities of KV cache prefetching that we observe (Section 4.2). We then explain our prefetching module (Section 4.3), which builds on the offloading-based inference systems, and discuss how InfiniGen manages the KV cache on CPU memory regarding the memory pressure (Section 4.4).

4.1 Overview

Figure 6 shows an overview of our KV cache management framework, InfiniGen, which enables offloading the KV cache with low data transfer overhead. The key design principle behind InfiniGen is to exploit the abundant CPU memory capacity to increase the *window size* when identifying the important tokens in the KV cache. As such, the majority of the tokens for the KV cache are kept in the CPU memory as we generate new tokens, not completely discarding them unlike prior works [37, 78]. However, we do not bring the entire KV cache to the GPU for attention computation, but load and compute with only the keys and values of a *few important tokens*, dropping other unimportant ones dynamically. To do so, we maintain the KV cache pool in the CPU memory and selectively and speculatively load a few of tokens.

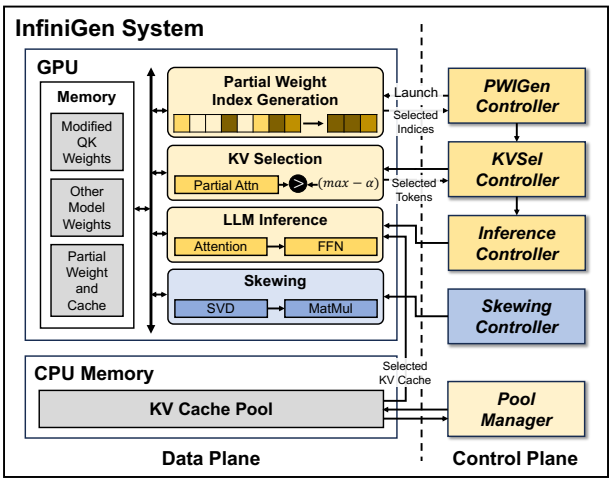

Figure 6: Overview of InfiniGen design.

In detail, we use the attention input of the *previous* Transformer layer to speculate and prefetch the keys and values of the important tokens for the *current* layer. The speculation is done by performing a minimal rehearsal of attention computation of the current layer in the preceding layer. This allows for reducing the waste of PCIe bandwidth by only transferring the keys and values critical for attention computation while preserving model accuracy. In addition, although the KV cache is offloaded to CPU memory, which is much cheaper and larger than GPU memory, we manage the KV cache pool size so as not to put too much pressure on CPU memory.

As shown in Figure 6, there are two major components in the InfiniGen runtime. The first includes the Partial Weight Index Generation Controller, KV Selection Controller, and Inference Controller. These controllers cooperate to speculate and prefetch the critical KV cache entries while serving LLM inference. Additionally, to aid in prefetching, the Skewing Controller performs offline modifications on the model weights. We explain each operation in Section 4.3. The second component is the Pool Manager. It manages the KV cache pool on CPU memory under CPU memory pressure, which we discuss in Section 4.4.

4.2 Prefetching Opportunities

In the following, we first explain why using the attention input of the previous layer for speculation makes sense. We then show how we modify the query and key weight matrices to make our speculation far more effective.

Attention Input Similarity. Our prefetching module builds on the key observation that the attention inputs of consecutive attention layers are highly similar in LLMs. There are two major reasons behind this. The first is the existence of *outliers* in LLMs, as discussed in Section 2.3, and the second is due to layer normalization (LayerNorm).

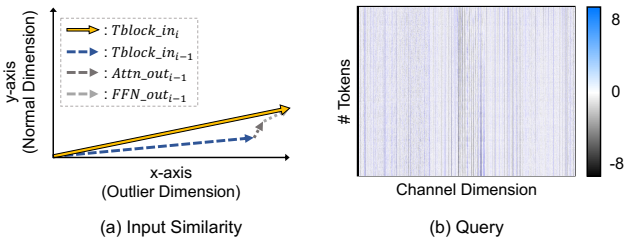

Figure 7: (a) Visualization of input similarity between consecutive Transformer blocks. (b) Query matrix of Layer 18 of the OPT-13B model. We only show channels from 3000 to 4000 for a clearer view of column-wise patterns.

To begin with, the input to the Transformer block i (Tblock_in$_i$) can be formulated as follows:

$$\begin{aligned} Attn\_out_{i-1} &= Attn(LN(Tblock\_in_{i-1})) \\ FFN\_out_{i-1} &= FFN(LN(Tblock\_in_{i-1} + Attn\_out_{i-1})) \\ Tblock\_in_i &= Tblock\_in_{i-1} + Attn\_out_{i-1} + FFN\_out_{i-1}, \end{aligned} \quad (1)$$

where Tblock_in$_{i-1}$ is an input for Layer $i-1$, which is first layer-normalized (LN) and is fed into the attention layer in the Transformer block. After performing attention, we obtain the output (Attn_out$_{i-1}$), which is added to Tblock_in$_{i-1}$ because of the residual connection. Then, the sum of Tblock_in$_{i-1}$ and Attn_out$_{i-1}$ is again layer-normalized and is fed into the FFN layer. Afterward, we obtain the FFN output (FFN_out$_{i-1}$), which is added to the sum of Tblock_in$_{i-1}$ and Attn_out$_{i-1}$ again due to the residual connection. Finally, the sum of Tblock_in$_{i-1}$, FFN_out$_{i-1}$, and Attn_out$_{i-1}$ is used as input to the next Transformer block (Tblock_in$_i$).

Now, we show why the attention input of Layer i is similar to the one of Layer $i-1$ with the example in Figure 7(a). In the figure, there are four vectors, each of which corresponds to a term in Equation 1. The x-axis represents an outlier channel among the model dimension, while the y-axis represents a normal channel (i.e., other than the outlier channel). In practice, there exist more normal channels and only a few outlier channels in the input tensors, but we only present one channel each for both outlier and normal channels for clarity.

Tblock_in$_{i-1}$ is highly skewed along the outlier channel (x-axis) due to a few outlier channels containing significantly large values compared to those in the normal channels. In contrast, Attn_out$_{i-1}$ and FFN_out$_{i-1}$ have relatively small values for both outlier and normal channels (i.e., short vectors). This is because the attention and FFN inputs are *layer-normalized*, reducing the magnitude of each value. The small magnitude of the attention and FFN inputs naturally results in their output values being relatively small compared to Tblock_in$_{i-1}$. Consequently, Tblock_in$_i$ is highly influenced by Tblock_in$_{i-1}$, rather than Attn_out$_{i-1}$ or FFN_out$_{i-1}$. Highly similar inputs between consecutive Transformer blocks lead to similar inputs across the attention layers, as the attention input is a layer-normalized one of the Transformer block input.

Table 1: Average cosine similarity between the Transformer block input of Layer i (Tblock_in$_i$) and the other three tensors (Tblock_in$_{i-1}$, Attn_out$_{i-1}$, FFN_out$_{i-1}$) across the layers. We use a random sentence with 2000 tokens from the PG-19 dataset [52].

Tensors	OPT-6.7B	OPT-13B	OPT-30B	Llama-2-7B	Llama-2-13B
Tblock_in$_{i-1}$	**0.95**	**0.96**	**0.97**	**0.89**	**0.91**
Attn_out$_{i-1}$	0.29	0.28	0.36	0.31	0.27
FFN_out$_{i-1}$	0.34	0.28	0.35	0.37	0.34

Table 1 shows the cosine similarity between Tblock_in$_i$ and the other three tensors (Tblock_in$_{i-1}$, Attn_out$_{i-1}$, FFN_out$_{i-1}$). As shown in the table, Tblock_in$_i$ is highly dependent on the Tblock_in$_{i-1}$ rather than others. InfiniGen leverages this key observation to speculate on the attention pattern of Layer i using the attention input of Layer $i-1$. Note that Tblock_in gradually changes across the layers; the inputs to distant layers are distinct.

Skewed Partial Weight. We observe that the attention score highly depends on a few columns in the query and key matrices. Figure 7(b) shows the values in a query matrix of Layer 18 of the OPT-13B model, where the column-wise patterns indicate that there exist certain columns with large magnitudes in the matrix; we observe the same patterns in the key and query matrices across different layers and models. The large magnitude columns have a great impact on the attention pattern because the dot product between the query and key is highly affected by these few columns. The column-wise pattern in the attention input indicates that there is little variance between each row in the outlier channels. Thus, the dot product between any row of the attention input and a column of the weight matrix could have a similarly large magnitude, which induces the outlier channels in the query and key matrices.

Going one step further, if we make a few columns in the query and key matrices have much larger magnitude than others, a much smaller number of columns significantly affects the attention pattern. We can do this by multiplying the query and key weight matrices with the same orthogonal matrix A. Since the transpose of the orthogonal matrix is the inverse of itself, the proposed operation does not change the final result, as shown in Equation 2 (i.e., this is mathematically equivalent to QK^T, not an approximation):

$$\begin{aligned} \tilde{Q} &= X_a \times W_Q \times A, \qquad \tilde{K} = X_a \times W_K \times A \\ \tilde{Q} \times \tilde{K}^T &= X_a \times W_Q \times A \times (X_a \times W_K \times A)^T \\ &= X_a \times W_Q \times A \times A^T \times W_K^T \times X_a^T \\ &= X_a \times W_Q \times W_K^T \times X_a^T \\ &= X_a \times W_Q \times (X_a \times W_K)^T \\ &= Q \times K^T, \end{aligned} \quad (2)$$

where $\tilde{Q}$ and $\tilde{K}$ are skewed query and key matrices, while W_Q and W_K are query and key weight matrices. X_a denotes the attention input. We set the orthogonal matrix A whose direction

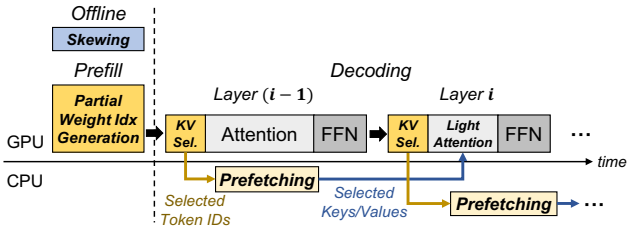

Figure 8: Operation flow of the prefetching module of InfiniGen.

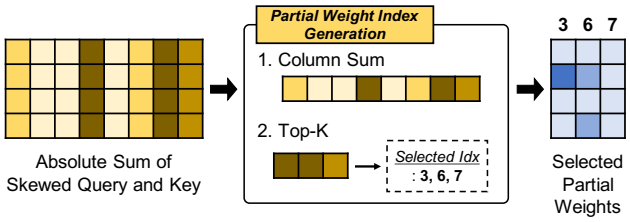

Figure 9: Partial weight generation in the prefill stage.

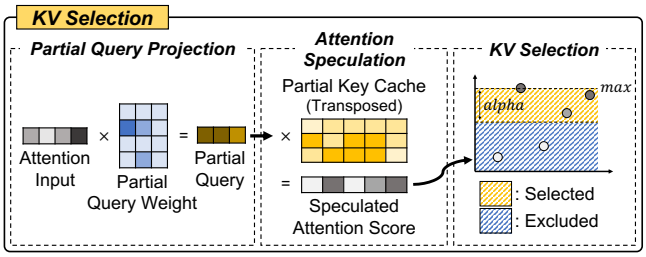

Figure 10: Attention score speculation in the decoding stage.

aligns with the direction that the query matrix stretches the most. Specifically, we first decompose the query matrix using SVD and obtain $\mathbf{U}$, Σ, and $\mathbf{V}$. We then set A to orthogonal matrix $\mathbf{V}$ to align the column vectors with the standard unit vectors as $\mathbf{V}^T\mathbf{A} = \mathbf{V}^T\mathbf{V} = I$, where I is an identity matrix. We formulate the skewed query matrix as follows:

$$\tilde{Q} = Q \times A = \mathbf{U}\Sigma\mathbf{V}^T \times A = \mathbf{U}\Sigma\mathbf{V}^T \times \mathbf{V} \quad (3)$$

In this way, we can make a few columns with large magnitudes in $\tilde{Q}$ without altering the result of computation, as discussed in Section 2.4.

4.3 Efficiently Prefetching KV Cache

Prefetching Scheme. Figure 8 shows the operation flow of the prefetching module in InfiniGen. In the offline phase, InfiniGen modifies the *weight* matrices to generate skewed query and key matrices. To achieve this, InfiniGen first runs the forward pass of the model once with a sample input. During this process, InfiniGen gathers the query matrix from each layer and performs singular value decomposition (SVD) of each query matrix. The skewing matrix (A_i) of each layer is obtained using the decomposed matrices of the query matrix, as shown in Equation 3. This matrix is then multiplied with each of the query and key weight matrices in the corresponding layer. Importantly, after the multiplication, the dimensions of the weight matrices remain unchanged. Note that the skewing is a *one-time* offline process and does not incur any runtime overhead because we modify the weight matrices that are invariant at runtime. As we exploit the column-wise pattern, which stems from the intrinsic property of the model rather than the input, whenever we compute the query and key for different inputs after the skewing, the values exhibit a high degree of skewness, thereby improving the effectiveness of our prefetching module. Note that skewing does not alter the original functionality. Even with the skewing, the attention layer produces identical computation results.

Prefill Stage. In the prefill stage, InfiniGen selects several important columns from the query weight matrix and the key cache to speculate on the attention pattern, and generates *partial* query weight and key cache matrices used in the decoding stage. Figure 9 shows how InfiniGen creates these partial matrices. Because we multiply each column in the query matrix with the corresponding row in the transposed key matrix, it is essential to select the *same* column indices in the query weight matrix and the key cache to obtain a proper approximation of the attention score. However, the indices of the outlier columns of the skewed query ($\tilde{Q}$) and key ($\tilde{K}$) matrices may not align exactly. To obtain partial matrices that capture the outliers, we first take the element-wise absolute values of the skewed query and key matrices, then add these two matrices together. This helps us calculate the sum of each column and perform top-k operation only once while accommodating the outlier columns of both query and key matrices. We then sum the elements in each column and select the top-k columns in the matrix; we choose 30% of the columns in our work. Using the sum of column values captures the global trend of each column while minimizing the effect of variance in each row. The selected columns better approximate the attention pattern because of the use of skewed query and key matrices.

Decoding Stage. In the decoding stage, InfiniGen speculates on the attention pattern of the next layer and determines the critical keys and values to prefetch. Figure 10 shows how InfiniGen computes the *speculated* attention score. At Layer $i-1$, we use the partial query weight matrix and key cache of Layer i, which are identified in the prefill stage, along with the attention input of Layer $i-1$. After multiplying the partial query and partial key cache, InfiniGen selects tokens with high attention scores.

We set the threshold considering the maximum value of the speculated attention score. We select only the tokens with an attention score greater than the maximum score subtracted by *alpha*. It is noted that subtraction from the attention score results in division after softmax. For example, assume that the attention score of the 3rd token is the maximum attention score minus 5. Once we apply softmax to the attention scores, the attention weight of the 3rd token is the maximum attention weight divided by $e^5 \approx 148.4$. Even though we do not use this

token, it does not noticeably hurt the accuracy of the model since it accounts for less than 1% of importance ($\approx 1/148.4$) after softmax. Thus, InfiniGen only prefetches the keys and values of the tokens with an attention score larger than the highest attention score minus *alpha*. As multiple attention heads are computed in parallel, we ensure that each head in the same layer fetches the same number of tokens by averaging the number of tokens between the maximum score and the threshold across the heads.

By reducing the amount of KV cache to load and compute, InfiniGen effectively reduces the loading latency (i.e., data transfer from CPU to GPU) while maintaining an output quality similar to that of the original model with a full KV cache. Moreover, as InfiniGen does not require a fixed number of tokens to load from CPU memory, it utilizes only the necessary PCI interconnect bandwidth. InfiniGen initiates speculation and prefetching from Layer 1 because the outliers, which are essential for exploiting input similarity, emerge during the computation in Layer 0.

4.4 KV Cache Pool Management

We manage the KV cache as a pool, offloading to the CPU memory and prefetching only the essential amount to the GPU. While CPU memory is more affordable and larger than GPU memory, it still has limited capacity. Hence, for certain deployment scenarios, it might be crucial to confine the size of the KV cache pool and remove less important KV entries that are infrequently selected by query tokens. We extend the design to incorporate a user-defined memory size limit. During runtime, when the size of the CPU memory reaches a user-defined limit, the KV cache pool manager selects a victim KV entry for eviction. Subsequently, the manager overwrites the selected victim with the newly generated key and value, along with updating the corresponding partial key cache residing in the GPU. It is noted that the order of KV entries can be arbitrary, as long as the key and value of the same token maintain the same relative location in the KV cache pool.

The policy of selecting a victim is important since it directly impacts model accuracy. We consider a counter-based policy along with two widely used software cache eviction policies: FIFO [7, 69, 70] and Least-Recently-Used (LRU) [2]. The FIFO-based policy is easy to implement with low overhead but results in a relatively large accuracy drop since it simply evicts the oldest residing token. The LRU-based policy generally exhibits a smaller decrease in accuracy but often entails a higher runtime overhead. In general, LRU-based policy uses a doubly linked list with locks to promote accessed objects to the head, which requires atomic memory updates for accessed KV entries. In the case of the counter-based policy, the pool manager increments a counter for each prefetched KV entry and selects a victim with the smallest count in the KV cache pool. If any counter becomes saturated, all the counter values are reduced by half. We observe that the counter-based policy and the LRU-based one show comparable model accuracy, which we discuss in Section 5.2. We opt for a counter-based approach due to its simpler design and to avoid atomic memory updates for better parallelism.

5 Evaluation

5.1 Experimental Setup

Model and System Configuration. We use Open Pre-trained Transformer (OPT) models [77] with 6.7B, 13B, and 30B parameters for evaluation. The 7B and 13B models of Llama-2 [60] are also used to demonstrate that InfiniGen works effectively across different model architectures. We run the experiments on a system equipped with an NVIDIA RTX A6000 GPU [44] with 48GB of memory and an Intel Xeon Gold 6136 processor with 96GB of DDR4-2666 memory. PCIe 3.0 $\times$16 interconnects the CPU and GPU.

Workload. We evaluate using few-shot downstream tasks and language modeling datasets. We use five few-shot tasks from the lm-evaluation-harness benchmark [23]: COPA [54], OpenBookQA [42], WinoGrande [55], PIQA [8], and RTE [62]. The language modeling datasets used are WikiText-2 [41] and Penn Treebank (PTB) [38]. Additionally, randomly sampled sentences from the PG-19 dataset [52] are used to measure the speedup with long sequence lengths.

Baseline. We use two inference environments that support KV cache offloading: CUDA Unified Virtual Memory (UVM) [4] and FlexGen [57]. On UVM, all data movements between the CPU and GPU are implicitly managed by the UVM device driver, thereby enabling offloading without requiring intervention from the programmer. In contrast, FlexGen uses explicit data transfers between the CPU and GPU. For the FlexGen baseline, unless otherwise specified, we explicitly locate all the KV cache in the CPU memory. The model parameters are stored in the GPU memory as much as possible, with the remainder in the CPU memory. We compare InfiniGen with two different KV cache management methods: H_2O [78] and Quantization [57]. H_2O, a recent method in KV cache management, maintains the KV cache of the important or recent tokens by assessing the importance of each token and discarding others. Quantization-based compression applies group-wise asymmetric quantization to the KV cache.

Key Metric. We evaluate accuracy (%) to assess the impact of approximation when InfiniGen, H_2O, and Quantization are used. For the language modeling tasks with WikiText-2 and PTB, we use perplexity as a metric; lower perplexity means better accuracy. To present performance improvements, we measure the wall clock time during inference with varying batch sizes and sequence lengths. The partial weight ratio is set to 0.3. We set alpha to 4 for OPT and 5 for Llama-2, resulting in using less than 10% of the KV cache on average across the layers. For each layer, we allow sending up to 20% of the total KV cache to the GPU if it contains more candidates.

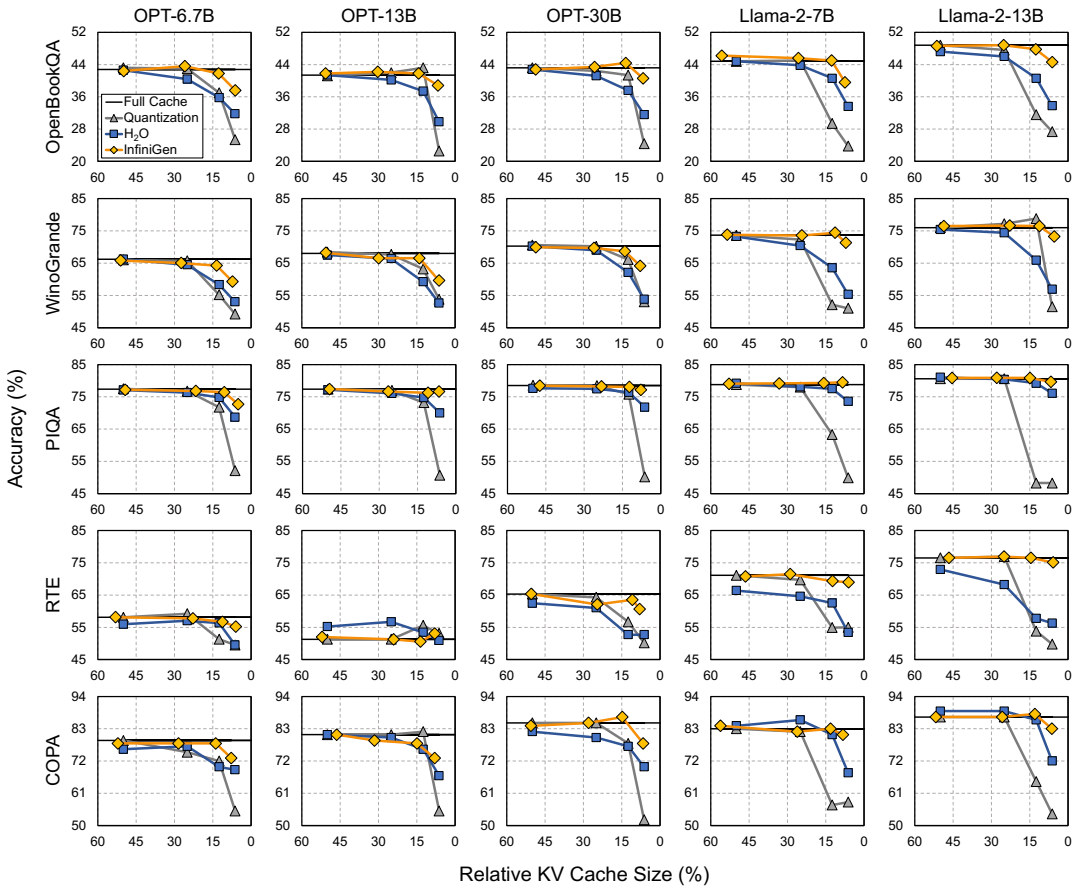

Figure 11: Accuracy of LLMs on 5-shot tasks in lm-evaluation-harness.

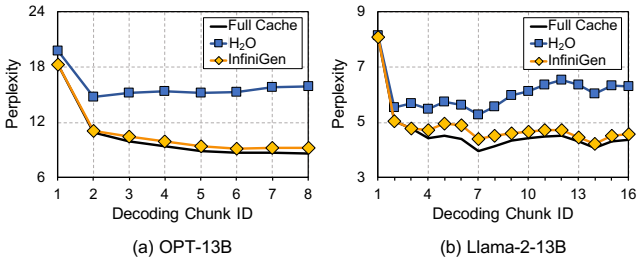

Figure 12: Perplexity of OPT-13B and Llama-2-13B for WikiText-2 dataset. Lower is better. Perplexity is computed for each decoding chunk that contains 256 tokens.

The partial weight ratio and alpha are determined based on a sensitivity study for each model to balance accuracy and inference latency, which we discuss in Section 6.1.

5.2 Language Modeling

Accuracy on lm-evaluation-harness. Figure 11 shows the accuracy of the baselines and InfiniGen across different models with 5-shot tasks. The relative KV cache size indicates the size of the KV cache involved in the attention computation compared to the full-cache baseline (e.g., a 10% relative KV cache size means that 10% of the full KV cache size is used). InfiniGen consistently shows better accuracy across the models and tasks when the relative KV cache size is less than 10%, whereas the others exhibit a noticeable accuracy drop due to insufficient bit widths (Quantization) or permanent KV cache elimination (H_2O). This implies that our proposed solution can effectively reduce the KV cache transfer overhead while preserving model accuracy. For relative KV cache sizes larger than 10%, the accuracy with InfiniGen closely matches that of the full-cache baseline. In some cases, InfiniGen even shows slightly better accuracy than the full-cache baseline. This is likely because reducing the amount of the KV cache participating in the attention computation can help the model focus more on critical tokens.

Sequence Length. Figure 12 shows the perplexity of two different models with InfiniGen and the baselines, as the sequence length increases. In this experiment, H_2O is configured to use the same amount of KV cache as InfiniGen. The sequence lengths are 2048 and 4096 for OPT-13B and Llama-2-13B, respectively. For a clearer view, we evaluate perplexity with consecutive 256 tokens as a group, which is referred to as a decoding chunk in the figure. The results show that even though the sequence length becomes longer (i.e., the decoding chunk ID increases), the perplexity of InfiniGen remains consistently comparable to the full-cache baseline,

Table 2: Perplexity on WikiText-2 and PTB with 2048 sequence length with or without KV cache memory limits. Lower is better.

Scheme	OPT-6.7B		OPT-13B		OPT-30B		Llama-2-7B		Llama-2-13B	
	Wiki	PTB	Wiki	PTB	Wiki	PTB	Wiki	PTB	Wiki	PTB
100%	11.68	13.86	10.55	12.78	10.14	12.31	5.69	22.53	5.25	31.94
80-FIFO%	19.64	16.82	30.99	33.84	30.66	35.45	22.26	61.88	21.41	32.34
80-LRU%	11.68	13.85	10.55	12.78	10.14	12.31	5.69	22.53	5.25	31.94
80-Counter%	11.68	13.86	10.55	12.78	10.14	12.31	5.69	22.53	5.25	31.94

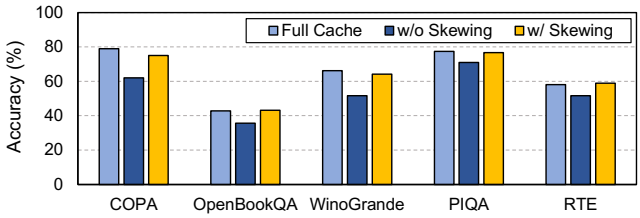

Figure 13: Accuracy on the lm-evaluation-harness benchmark with or without skewing on OPT-6.7B.

while H_2O shows an increasing divergence from the baseline. H_2O suffers from permanent KV cache elimination and may not retain a sufficient amount of KV cache in certain layers due to its fixed budget. In contrast, InfiniGen dynamically computes attention using only the essential amount of KV cache for each layer. The difference is likely to widen as the models become capable of handling much longer sequences.

Effect of Skewing. Figure 13 shows the accuracy with or without key/query skewing on the OPT-6.7B model. For the experiment, we use a fixed KV cache budget of 20%, instead of using a dynamic approach, to clearly show the effect of skewing. We observe that several language models (e.g., Llama-2) show a small drop in accuracy without skewing. For some models such as OPT-6.7B, however, we see a large accuracy drop if we do not apply the skewing method as shown in Figure 13. This indicates that in the case of OPT-6.7B, the partial weight does not adequately represent the original matrix without skewing. After applying our skewing method, we achieve accuracy similar to the full-cache baseline. Our skewing method effectively skews key and query matrices such that a few columns can better represent the original matrices.

KV Cache Pool Management. Table 2 shows the perplexity of five different models with or without limiting the memory capacity for WikiText-2 and PTB. We compare FIFO-based, LRU-based, and Counter-based victim selection policies in Section 4.4 under the 80% memory limit of a full KV cache. We also present the perplexity results with no memory limit (100%). The FIFO-based approach shows the worst model performance because it simply deletes the oldest KV entry regardless of its importance. The LRU and Counter-based approaches show perplexity that is almost similar to that with no memory limit. We choose a Counter-based victim selection policy instead of an LRU-based approach because the LRU-based approach typically needs to maintain a doubly linked list queue with locks for atomic memory updates.

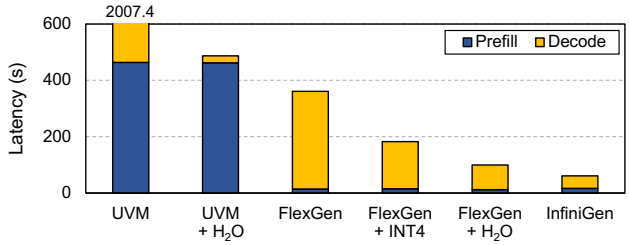

Figure 14: Inference latency on OPT-13B with a sequence length of 2048 (1920 input and 128 output tokens) and a batch size of 20.

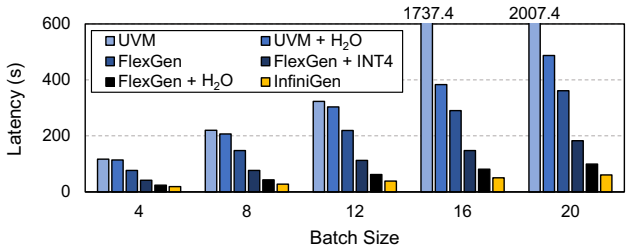

Figure 15: Inference latency for 5 different batch sizes on OPT-13B with a sequence length of 2048 (1920 input and 128 output tokens).

5.3 Performance

In this section, we refer to H_2O (with a KV cache budget of 20%) and 4-bit quantization implemented on top of FlexGen as H_2O and INT4.

Inference Latency. Figure 14 shows the inference latency including the prefill and decoding stages. We use the OPT-13B model with 1920 input tokens, 128 output tokens, and a batch size of 20. InfiniGen achieves 1.63×-32.93× speedups over the baselines. The performance benefit mainly comes from the significantly reduced amount of KV cache to load from the CPU memory due to our dynamic approach.

UVM shows an extremely long latency because the working set size (i.e., the size of the model parameters and KV cache) is larger than the GPU memory capacity, thereby leading to frequent page faults and data transfers between the CPU and GPU. The prefill stage of UVM + H_2O also shows a long latency due to the page faults and data transfers. However, because all required data are migrated to the GPU memory after the prefill stage, UVM + H_2O shows a substantially shorter decoding latency. FlexGen loads the full KV cache with high precision (i.e., FP16) from the CPU memory for every attention computation. On the other hand, INT4 and H_2O load relatively small amounts of the data from the CPU because of the low-bit data format (INT4) or a smaller size of the KV cache (H_2O). However, they still load larger amounts of data than InfiniGen; even with low precision, INT4 loads the KV cache of all the previous tokens; H_2O always loads the same amount of data no matter how many tokens are actually important in each layer. As a result, InfiniGen achieves better performance than both of them.

Batch Size. Figure 15 shows the inference latency across different batch sizes. The results show that InfiniGen achieves

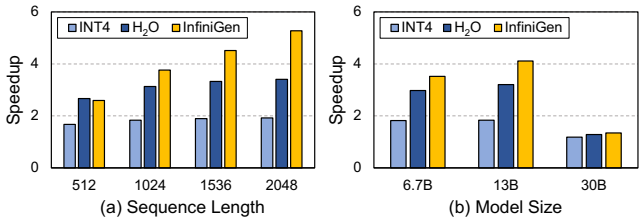

Figure 16: Speedup over the FlexGen baseline across (a) sequence lengths and (b) model sizes.

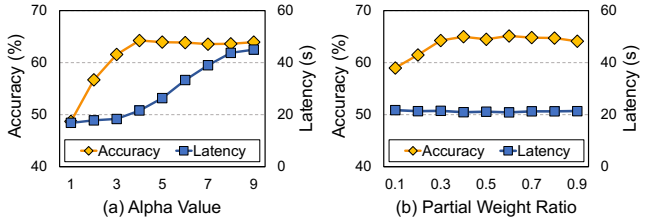

Figure 17: Accuracy and inference latency across (a) alpha values and (b) partial weight ratios.

lower latency than others across the batch sizes (1.28×-34.64×). As the batch size increases, the performance gap between InfiniGen and others becomes larger. UVM and UVM + H_2O show increasing latency mainly due to frequent page faults in the prefill stage. For UVM, the latency also rapidly increases at a batch size of 16 because the working set size exceeds the GPU memory capacity for *both* prefill and decoding stages. As the batch size keeps increasing, UVM + H_2O will face the same problem as well.

The latency of FlexGen almost linearly increases with the batch size because the KV cache transfer occupies the majority of the inference latency. As we increase the batch size from 4 to 20, the throughput (tokens per second) of InfiniGen increases from 27.36 to 41.99, while INT4 and H_2O offer a small increase in throughput (from 12.22 to 14.02 and from 21.31 to 25.70 each). By dynamically adjusting the amount of the KV cache to load, InfiniGen achieves scalable performance across the batch sizes.

Sequence Length. Figure 16(a) shows the speedup of INT4, H_2O, and InfiniGen over FlexGen on OPT-13B across different sequence lengths. With a batch size of 8, we use four different input/output configurations. Each configuration comprises 128 output tokens and 384, 896, 1408, 1920 input tokens (i.e., a total number of tokens ranging from 512 to 2048). The speedup of InfiniGen continues to increase across the sequence lengths (up to 5.28×), whereas INT4 and H_2O show saturating speedups (up to 1.92× and 3.40×). This suggests that neither INT4 nor H_2O provides a scalable solution for KV cache management. INT4 shows a negligible increase in speedup due to the inherent growth in the size of the KV cache. Similarly, H_2O lacks scalability due to its fixed ratio of the KV cache budget; as the sequence length increases, H_2O stores and loads more KV cache.

Even though the sequence length increases, the number of tokens that each token attends to does not increase linearly. For instance, in the OPT-13B model, we count the number of important tokens with attention scores larger than $(max - 4)$ and identify that, on average, 37, 60, 66, and 73 tokens are assessed as *important* for sequence lengths of 512, 1024, 1536, and 2048, respectively. H_2O, which employs 20% of a fixed KV cache budget, loads 409 tokens for the sequence length of 2048, while only 73 tokens are relatively important. In contrast, InfiniGen naturally captures this trend (i.e., a non-linear increase in the number of important tokens) by dynamically observing the speculated attention score.

Model Size. Figure 16(b) shows the speedup of INT4, H_2O, and InfiniGen over FlexGen on three different model sizes. We use 1920 input tokens and 128 output tokens with a batch size of 4 for the experiment. The results show that InfiniGen outperforms others across the model sizes. As the model size increases from 6.7B to 13B, the speedup of InfiniGen also increases by 1.17×, while others do not lead to a noticeable increase in speedup. For most of the layers, InfiniGen loads a smaller amount of KV cache than H_2O because a relatively small number of tokens are needed. Thus, InfiniGen performs better than H_2O as the model size becomes larger due to the increased number of Transformer blocks. For the 30B model, the model parameters do not fit in the GPU memory. As such, we offload 30% of the model parameters to the CPU. In this case, the size of the offloaded parameters is 1.7× larger than the KV cache size. Even so, InfiniGen shows a 1.34× speedup over FlexGen, while others achieve 1.18× and 1.28× each.

6 Analysis and Discussion

6.1 Sensitivity Study

We use the OPT-6.7B model with 1920 input tokens, 128 output tokens, and a batch size of 8. The accuracy is evaluated with the WinoGrande task in lm-evaluation-harness.

Threshold and Alpha. As discussed in Section 4.3, we load the KV cache of the tokens with a speculated attention score greater than the threshold (i.e., the maximum attention score minus alpha). Increasing alpha results in fetching more KV entries to the GPU, thus increasing inference latency but also improving accuracy. Figure 17(a) shows such trade-offs between accuracy and inference latency for nine different alpha values with a partial weight ratio of 0.3. The results show that more KV entries are fetched and involved in attention computation as alpha increases, thereby leading to better accuracy. For the alpha values beyond 4, however, since most important tokens are already included, the accuracy does not further increase, while the cost for KV transfers and attention computation keeps increasing. This trend is similarly observed in other models, and we thus opt for an alpha value of 4 or 5 to strike a balance between inference latency and accuracy.

Partial Weight Ratio. Figure 17(b) shows the accuracy and

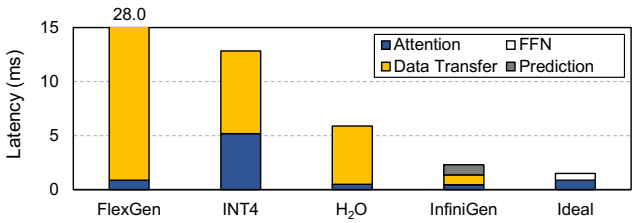

Figure 18: Latency breakdown of a Transformer block for OPT-13B with a sequence length of 2048 and a batch size of 8.

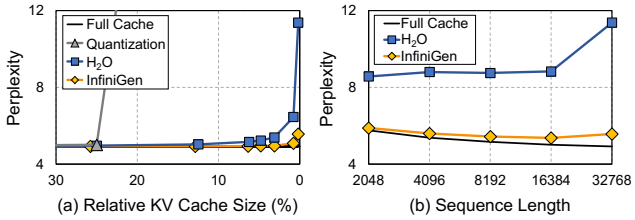

Figure 19: Perplexity of Llama-2-7B-32K across (a) relative KV cache sizes with a sequence length of 32768 and (b) sequence lengths while retaining 64 tokens. Lower is better. Llama-2-7B-32K is a fine-tuned version capable of processing up to 32K tokens using position interpolation [12]. Quantization is omitted in (b) since the KV cache cannot be compressed below 6.25% (i.e., 1 bit).

inference latency across different partial weight ratios with an alpha value of 4. As shown in the figure, the amount of partial weights has a negligible impact on inference latency because the cost for computing the speculated attention score is relatively small. Note that the amount of KV cache to transfer is *not* related to the partial weight ratio. However, increasing the partial weight ratio results in higher memory consumption for partial weights and key cache (e.g., doubling the ratio doubles the memory consumption overhead). The accuracy also does not noticeably differ beyond a ratio of 0.3. In our work, we opt for a partial weight ratio of 0.3 to achieve better accuracy while considering memory consumption overhead.

6.2 Overhead

Prefetching Overhead. Figure 18 shows the latency breakdown of executing a single Transformer block for the OPT-13B model; FFN is not shown in the figure for schemes other than Ideal because it is fully overlapped with data transfer time. Ideal is a scenario where all the computations (i.e., attention and FFN) are performed on the GPU without any data transfer between the CPU and GPU. As shown in the results, the key performance bottleneck of FlexGen and H_2O is the data transfer overhead, which occupies 96.9% and 91.8% of the execution time, respectively. For INT4, due to the quantization and dequantization overhead, attention computation also occupies a large portion of the execution time in addition to the data transfer. InfiniGen, on the other hand, considerably improves the inference speed over FlexGen by reducing the amount of data transfer with our dynamic KV cache prefetching. Furthermore, InfiniGen is only 1.52× slower than Ideal, while others show 3.90×-18.55× slowdowns.

Memory Consumption. InfiniGen uses the partial query weight and key cache for speculation. For a ratio of 0.3, the sizes of the partial query weight and key cache are only 2.5% and 15% of the total model parameters and total KV cache, respectively. While we simply store them in the GPU during our experiments, we can manage the storage overhead in various optimized ways if needed. For example, we can store only the column indices of the partial query weight and retrieve the column vectors from the full query weight matrix (which already resides in the GPU) as needed for partial query projection. Additionally, we can place the partial key cache in the CPU and perform speculation on the CPU after fetching the partial query from the GPU. Even a naïve method of lowering the partial ratio would likely still provide better accuracy compared to other methods while reducing storage overhead. In summary, by minimally sacrificing inference performance, we can greatly reduce the storage overhead on the GPU if necessary.

6.3 Long Context Window

Figure 19 shows the perplexity of the Llama-2-7B-32K model, which can process up to 32K tokens, across the relative cache sizes and sequence lengths. We use the WikiText-2 dataset for the experiment. As the context window size increases for future LLMs, the relative portion of the KV cache that the GPU can retain would decrease due to the limited capacity of GPU memory.

Figure 19(a) shows that InfiniGen maintains perplexity levels close to the full-cache baseline even as the relative KV cache size decreases, without leading to a noticeable increase in perplexity even with much smaller cache sizes. In contrast, other methods increase perplexity compared to the full-cache baseline and significantly diverge at certain sizes due to insufficient bit widths for preserving adequate information on all keys and values (Quantization) or the permanent removal of KV cache entries (H_2O). As shown in Figure 19(b), the perplexity gap between InfiniGen and H_2O widens for longer sequence lengths, which is likely to increase further for sequence lengths beyond 32K. This implies that InfiniGen can scale to longer sequences and better preserve model accuracy compared to others.

We further speculate on how InfiniGen would benefit in an era of million-token context windows by analyzing a model capable of handling 1 million tokens. Figure 20(a) shows that the percentage of query tokens that attend to less than 1% of key tokens increases as the sequence length becomes longer. InfiniGen can adapt to this changing trend by dynamically adjusting the amount of the KV cache to load, whereas prior fixed-budget/pruning approaches would not easily adjust the effective KV cache size. Figure 20(b) further shows that the attention weights of key tokens can change across iterations;

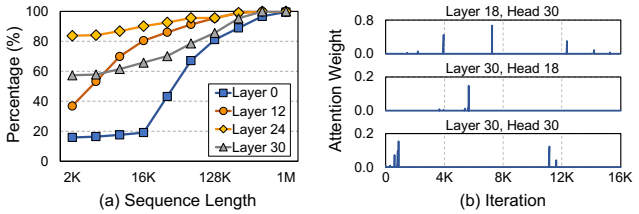

Figure 20: Analysis of 1 million tokens using Llama-3-8B-1048K. (a) Percentage of query tokens that attend to less than 1% of the key tokens across the sequence lengths. (b) Attention weight of sampled key tokens from different layers and heads across the last 16K iterations out of 1 million tokens.

the sampled key tokens show sudden *spikes* after thousands of iterations with significantly low attention weights (e.g., the 7425th iteration out of the last 16K iterations in Layer 18, Head 30). We observe that the prior approaches that permanently eliminate tokens while they are unimportant could lose the critical contexts if they become important again at later iterations. In contrast, InfiniGen can preserve model performance by keeping the temporarily unimportant KV entries for potential future use.

7 Related Work

DNN Serving Systems. A systematic approach to enabling an efficient and fast model serving system is an important topic that has been widely studied by both academia and industry. Some prior works focus on distributed systems with predictable latency for service-level objectives (SLOs) [15, 16, 25, 56]. Other works improve parallelism and throughput of the system through preemption [28, 75], fine-grained batching [17, 21, 71], or memory optimizations [18, 35, 58].

Several other works aim at achieving high throughput execution with limited GPU memory by offloading parameters to secondary storage (e.g., CPU memory and disk). Some of them build on CUDA Unified Memory [46] with prefetching [31, 39], while others explicitly move tensors in and out as needed for computation [29, 30, 48, 72, 73]. FlexGen [57] is a recent LLM serving system that enables high-throughput inference on a single GPU by offloading weights and KV cache to CPU memory and disk. InfiniGen is orthogonal to FlexGen and can work in conjunction with it to efficiently offload and prefetch the KV cache.

KV Cache Management. vLLM [35] mitigates the KV cache memory waste from fragmentation and duplication. StreamingLLM [67] enables LLMs to generate longer sequence lengths than the trained ones. However, since neither vLLM nor StreamingLLM reduces the size of the KV cache, data transfers still incur a significant overhead in offloading-based inference systems. InfiniGen complements the KV cache management to reduce the data transfer overhead, which is a major bottleneck in offloading-based systems.

Efficient LLM Inference. There are lines of research that exploit *quantization* or *sparsity* to make LLMs efficient through algorithmic methods [13, 19, 22, 34, 66] or hardware-software co-design [26, 27, 36, 50]. Regarding sparsity, most algorithm-based works focus on reducing the model size by exploiting the sparsity of weights. Alternatively, H_2O and Sparse Transformer [13] leverage the row-wise (i.e., token-level) sparsity in the KV cache by *permanently* removing certain KV entries. On the other hand, most hardware-software co-design studies focus on relaxing the quadratic computational complexity in the prefill stage by skipping non-essential key tokens with the aid of specialized hardware. However, they often do not reduce memory access as they identify the important key tokens only after scanning all the elements of the key tensors.

Kernel fusion [18, 32] is another approach to mitigating the quadratic memory overhead of attention in the prefill stage. InfiniGen can be implemented with kernel fusion techniques to alleviate the overhead of KV cache access in the decoding stage. To our knowledge, this is the *first* work to enable efficient LLM inference by prefetching only essential KV entries in offloading-based inference systems.

8 Conclusion

The size of the KV cache poses a scalability issue in high-throughput offloading-based inference systems, even surpassing the model parameter size. Existing KV cache eviction policies show a large accuracy drop and do not efficiently use the interconnect bandwidth when they are employed in offloading-based LLM systems. We propose InfiniGen, an offloading-based dynamic KV cache management framework that efficiently executes inference of large language models. InfiniGen exploits the attention input of the previous layer to speculatively prefetch the KV cache of important tokens. We manipulate the query and key weights to make the speculation more efficient. InfiniGen shows substantially shortened inference latency while preserving language model performance. It also shows much better scalability regarding the batch size, sequence length, and model size compared to prior solutions.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd Petros Maniatis for their valuable feedback. This work was supported in part by a research grant from Samsung Advanced Institute of Technology (SAIT) and by the artificial intelligence semiconductor support program to nurture the best talents (No. RS-2023-00256081) supervised by Institute for Information & Communications Technology Planning & Evaluation (IITP). The Institute of Engineering Research at Seoul National University provided research facilities for this work. Jaewoong Sim is the corresponding author.

References

[1] DeepL. https://www.deepl.com/translator.

[2] Memcached. https://memcached.org.

[3] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[4] Tyler Allen and Rong Ge. In-depth analyses of unified virtual memory system for gpu accelerated computing. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2021.

[5] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2022.

[6] Anthropic. The claude 3 model family: Opus, sonnet, haiku. 2024. https://www.anthropic.com/claude.

[7] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. The CacheLib caching engine: Design and experiences at scale. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[8] Yonatan Bisk, Rowan Zellers, Ronan bras, Jianfeng Gao, and Choi Yejin. Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2020.

[9] Beidi Chen, Tri Dao, Eric Winsor, Zhao Song, Atri Rudra, and Christopher Ré. Scatterbrain: Unifying sparse and low-rank attention. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

[10] Beidi Chen, Zichang Liu, Binghui Peng, Zhaozhuo Xu, Jonathan Lingjie Li, Tri Dao, Zhao Song, Anshumali Shrivastava, and Christopher Re. Mongoose: A learnable lsh framework for efficient neural network training. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021.

[11] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[12] Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. Extending context window of large language models via positional interpolation. *arXiv preprint arXiv:2306.15595*, 2023.

[13] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.

[14] Krzysztof Marcin Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Quincy Davis, Afroz Mohiuddin, Lukasz Kaiser, David Benjamin Belanger, Lucy J Colwell, and Adrian Weller. Rethinking attention with performers. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021.

[15] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. Inferline: latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2020.

[16] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[17] Weihao Cui, Han Zhao, Quan Chen, Hao Wei, Zirui Li, Deze Zeng, Chao Li, and Minyi Guo. Dvabatch: Diversity-aware multi-entry multi-exit batching for efficient processing of dnn services on gpus. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2022.

[18] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[19] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Gpt3.int8(): 8-bit matrix multiplication for transformers at scale. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[20] Angela Fan, Mike Lewis, and Yann Dauphin. Hierarchical neural story generation. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2018.

[21] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. Turbotransformers: an efficient gpu serving system for transformer models. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2021.

[22] Elias Frantar and Dan Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2023.

[23] Leo Gao, Jonathan Tow, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Kyle McDonell, Niklas Muennighoff, et al. A framework for few-shot language model evaluation, 2021.

[24] GitHub. Copilot. https://github.com/features/copilot.

[25] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving dnns like clockwork: Performance predictability from the bottom up. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[26] Cong Guo, Jiaming Tang, Weiming Hu, Jingwen Leng, Chen Zhang, Fan Yang, Yunxin Liu, Minyi Guo, and Yuhao Zhu. Olive: Accelerating large language models via hardware-friendly outlier-victim pair quantization. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2023.

[27] Tae Jun Ham, Yejin Lee, Seong Hoon Seo, Soosung Kim, Hyunji Choi, Sung Jun Jung, and Jae W Lee. Elsa: Hardware-software co-design for efficient, lightweight self-attention mechanism in neural networks. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2021.

[28] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent gpu-accelerated dnn inferences. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[29] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. Autotm: Automatic tensor movement in heterogeneous memory systems using integer linear programming. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[30] Chien-Chin Huang, Gu Jin, and Jinyang Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[31] Jaehoon Jung, Jinpyo Kim, and Jaejin Lee. Deepum: Tensor migration and prefetching in unified memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[32] Sheng-Chun Kao, Suvinay Subramanian, Gaurav Agrawal, Amir Yazdanbakhsh, and Tushar Krishna. Flat: An optimized dataflow for mitigating attention bottlenecks. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[33] Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2020.

[34] Woosuk Kwon, Sehoon Kim, Michael W Mahoney, Joseph Hassoun, Kurt Keutzer, and Amir Gholami. A fast post-training pruning framework for transformers. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

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

[36] Jungi Lee, Wonbeom Lee, and Jaewoong Sim. Tender: Accelerating large language models via tensor decomposition and runtime requantization. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2024.

[37] Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[38] Mitch Marcus, Grace Kim, Mary Ann Marcinkiewicz, Robert MacIntyre, Ann Bies, Mark Ferguson, Karen Katz, and Britta Schasberger. The penn treebank: Annotating predicate argument structure. In *Proceedings of the Workshop on Human Language Technology*, 1994.

[39] Pak Markthub, Mehmet E. Belviranli, Seyong Lee, Jeffrey S. Vetter, and Satoshi Matsuoka. Dragon: Breaking gpu memory capacity limits with direct nvm access. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2018.

[40] Pierre-Emmanuel Mazare, Samuel Humeau, Martin Raison, and Antoine Bordes. Training millions of personalized dialogue agents. In *Conference on Empirical*

Methods in Natural Language Processing (EMNLP), 2018.

[41] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2017.

[42] Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. Can a suit of armor conduct electricity? a new dataset for open book question answering. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2018.

[43] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2023.

[44] NVIDIA. NVIDIA RTX A6000 Graphics Card. https://www.nvidia.com/en-us/design-visualization/rtx-a6000/.

[45] NVIDIA. Triton inference server. https://developer.nvidia.com/triton-inference-server.

[46] Nvidia. Unified memory programming. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd.

[47] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, high-performance ml serving. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[48] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[49] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. In *Proceedings of the Machine Learning and Systems (MLSys)*, 2023.

[50] Zheng Qu, Liu Liu, Fengbin Tu, Zhaodong Chen, Yufei Ding, and Yuan Xie. Dota: detect and omit weak attentions for scalable transformer acceleration. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[51] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[52] Jack W Rae, Anna Potapenko, Siddhant M Jayakumar, Chloe Hillier, and Timothy P Lillicrap. Compressive transformers for long-range sequence modelling. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2020.

[53] Machel Reid, Nikolay Savinov, Denis Teplyashin, Dmitry Lepikhin, Timothy Lillicrap, Jean-baptiste Alayrac, Radu Soricut, Angeliki Lazaridou, Orhan Firat, Julian Schrittwieser, et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*, 2024.

[54] Melissa Roemmele, Cosmin Bejan, and Andrew Gordon. Choice of plausible alternatives: An evaluation of commonsense causal reasoning. In *AAAI Spring Symposium Series*, 2011.

[55] Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale. *Communications of the ACM*, 2021.

[56] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, 2019.

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

[58] Yining Shi, Zhi Yang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang, and Lidong Zhou. Welder: Scheduling deep learning memory access via tile-graph. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2023.

[59] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2014.

[60] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[61] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[62] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. In *EMNLP Workshop BlackboxNLP*, 2018.

[63] Sinong Wang, Belinda Z. Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020.

[64] Yiming Wang, Zhuosheng Zhang, and Rui Wang. Element-aware summarization with large language models: Expert-aligned evaluation and chain-of-thought method. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2023.

[65] Xiuying Wei, Yunchen Zhang, Xiangguo Zhang, Ruihao Gong, Shanghang Zhang, Qi Zhang, Fengwei Yu, and Xianglong Liu. Outlier suppression: Pushing the limit of low-bit transformer language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[66] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning (ICML)*, 2023.

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

[68] Haoran Xu, Young Jin Kim, Amr Sharaf, and Hany Hassan Awadalla. A paradigm shift in machine translation: Boosting translation performance of large language models. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024.

[69] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. Fifo queues are all you need for cache eviction. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, 2023.

[70] Tzu-Wei Yang, Seth Pollen, Mustafa Uysal, Arif Merchant, and Homer Wolfmeister. CacheSack: Admission optimization for google datacenter flash caches. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2022.

[71] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[72] Yuan Yu, Martín Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, et al. Dynamic control flow in large-scale machine learning. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys)*, 2018.

[73] Haoyang Zhang, Yirui Eric Zhou, Yu Xue, Yiqi Liu, and Jian Huang. G10: Enabling an efficient unified gpu memory and storage architecture with smart tensor migrations. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2023.

[74] Haoyu Zhang, Jingjing Cai, Jianjun Xu, and Ji Wang. Pretraining-based natural language generation for text summarization. In *Conference on Computational Natural Language Learning (CoNLL)*, 2019.

[75] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. Shepherd: Serving dnns in the wild. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*, 2023.

[76] Saizheng Zhang, Emily Dinan, Jack Urbanek, Arthur Szlam, Douwe Kiela, and Jason Weston. Personalizing dialogue agents: I have a dog, do you have pets too? In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2018.

[77] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

[78] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. H_2O: Heavy-hitter oracle for efficient generative inference of large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.