KIVI: A Tuning-Free Asymmetric 2bit Quantization for KV Cache

Zirui Liu [*1] **Jiayi Yuan** [*1] **Hongye Jin** [2] **Shaochen (Henry) Zhong** [1]
Zhaozhuo Xu [3] **Vladimir Braverman** [1] **Beidi Chen** [4] **Xia Hu** [1]

Abstract

Efficiently serving large language models (LLMs) requires batching many requests together to reduce the cost per request. Yet, the key-value (KV) cache, which stores attention keys and values to avoid re-computations, significantly increases memory demands and becomes the new bottleneck in speed and memory usage. This memory demand increases with larger batch sizes and longer context lengths. Additionally, the inference speed is limited by the size of KV cache, as the GPU's SRAM must load the entire KV cache from the main GPU memory for each token generated, causing the computational core to be idle during this process. A straightforward and effective solution to reduce KV cache size is quantization, which decreases the total bytes taken by KV cache. However, there is a lack of in-depth studies that explore the element distribution of KV cache to understand the hardness and limitation of KV cache quantization. To fill the gap, we conducted a comprehensive study on the element distribution in KV cache of popular LLMs. Our findings indicate that the key cache should be quantized per-channel, i.e., group elements along the channel dimension and quantize them together. In contrast, the value cache should be quantized per-token. From this analysis, we developed a tuning-free 2bit KV cache quantization algorithm, named KIVI. With the hardware-friendly implementation, KIVI can enable Llama (Llama-2), Falcon, and Mistral models to maintain almost the same quality while using $\mathbf{2.6}\times$ less peak memory usage (including the model weight). This reduction in memory usage enables up to $\mathbf{4}\times$ larger batch size, bringing $\mathbf{2.35}\times \sim \mathbf{3.47}\times$ throughput on real LLM inference workload. The source code is available at https://github.com/jy-yuan/KIVI.

1. Introduction

Large Language Models (LLMs) have demonstrated strong performance across a wide range of tasks (Brown et al., 2020; Taylor et al., 2022; Yuan et al., 2023). However, their deployment is very costly, requiring a large number of hardware accelerators such as GPUs. Given these substantial costs, one natural way to reduce the cost per request is to combine a sufficient number of requests together for batch processing. However, in this batch inference scenario, the key-value cache (KV cache), which holds the attention keys and values during generation to prevent re-computations, is becoming the new memory and speed bottleneck. This bottleneck becomes more pronounced with larger batch sizes and longer context lengths. For instance, in 540B PaLM, with a batch size of 512 and a context length of 2048, KV cache alone can take 3TB. This is 3 times the size of the model's parameters (Pope et al., 2023). Also, the GPU SRAM has to load the whole KV cache from the GPU device memory for every token generated, during which the computational cores are idle. Thus, reducing KV cache size in LLMs while maintaining accuracy is important.

Existing works towards this problem can be roughly divided into three categories. First, some work suggests reducing the number of heads in KV cache, such as multi-query attention (Shazeer, 2019) and multi-group attention (Ainslie et al., 2023). However, these methods require either training the model from scratch or fine-tuning the existing model. Second, another research line reduces KV cache size by evicting unimportant tokens (Zhang et al., 2023). Third, some other works try to solve this problem from the system perspective, e.g., offloading KV cache (Sheng et al., 2023) or extending virtual memory and paging techniques into the attention mechanism (Kwon et al., 2023).

To reduce the size of KV cache, the most simple and effective way is to reduce the total bytes taken by KV cache, namely, quantization. Unlike the well-studied weight quan-

[*]Equal contribution. The order of authors is determined by flipping a coin. [1]Department of Computer Science, Rice University [2]Department of Computer Science, Texas A&M University [3]Department of Computer Science, Stevens Institute of Technology [4]Department of Electrical and Computer Engineering, Carnegie Mellon University. Correspondence to: Zirui Liu <zl105@rice.edu>, Jiayi Yuan <jy101@rice.edu>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024.

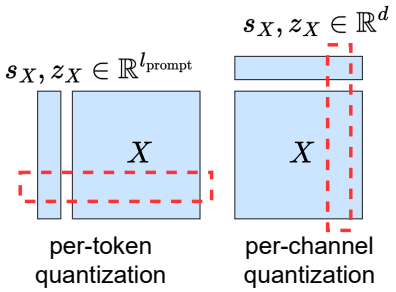

Figure 1: Definition of per-token and per-channel quantization. $\boldsymbol{X} \in \mathbb{R}^{l_{\text{prompt}} \times d}$ is key/value cache where l_{prompt} is the number of tokens and d is the number of channels. z_X is the zero-point, s_X is the scaling factor.

tization (Lin et al., 2023; Xiao et al., 2023a; Zhao et al., 2024), to the best of our knowledge, only a few studies applied the vanilla 4bit round-to-nearest quantization to KV cache (Sheng et al., 2023; Zhang et al., 2023; Zhao et al., 2024) due to the streaming nature of KV cache or other complications. There is a lack of in-depth studies that explore the element distribution of KV cache to understand the hardness and limitation of KV cache quantization. To fill the gap, we study the element distribution of KV cache. Our analysis suggests:

- For the key cache, there are a few fixed channels whose magnitudes are very large, which is consistent with previous finding (Lin et al., 2023; Xiao et al., 2023a). Thus, as shown in Figure 1 right, key cache should be quantized per-channel, i.e., group elements along the channel dimension and quantize them together. In this way, it can confine the error to each individual channel, without impacting the other normal channels.
- For the value cache, there is no obvious outlier pattern. Although value cache has no obvious outlier pattern, we experimentally show that it can only be quantized per-token because it is used to calculate the attention output, which is essentially a value cache mixer. As shown in Figure 1 left, the per-token quantization can confine the error inside each individual token and ensure that the quantization of one token does not adversely impact the others.

Based on the above insights, we propose KIVI, a plug-and-play extreme low-bit KV cache quantization method. KIVI quantizes key cache per-channel and quantizes value cache per-token. The per-token value cache quantization aligns well with the streaming nature of auto-regressive inference, allowing newly quantized tensors to be directly appended to the existing quantized value cache by token dimension. However, for per-channel key cache quantization, the quantization process spans different tokens, which cannot be directly implemented in this streaming setting. Since the number of tokens in key cache can be arbitrary, our key idea is to split key cache into two parts. The **first** part is the grouped key cache, which contains several groups of tokens and each group has a certain number of tokens. The **second** part is the residual key cache, which does not have a sufficient number of tokens to form a complete group. Similarly, we split value cache into the grouped and residual parts to maintain the accuracy. We only apply group-wise quantization to the grouped key cache and value cache, while the residual key cache and value cache are kept in full precision. The grouped and residual parts can be combined using tiled matrix multiplication when computing attention scores. Our contributions are summarized as follows:

- **Extensive analysis regarding the outlier patterns and quantization error of KV cache in commonly-used LLMs**. Our observations suggest that key cache should be quantized per-channel and value cache should be quantized per-token. We also explain in depth why these caches require different quantization approaches.
- **A new plug-and-play 2bit KV cache quantization algorithm without any fine-tuning, KIVI, with hardware-friendly implementation.** We conduct an extensive evaluation for KIVI with Llama (Llama-2), Mistral, and Falcon on popular generation tasks. KIVI can efficiently compress KV cache to 2bit and bring $\mathbf{2.6}\times$ peak memory usage reduction for Llama-2-7B, with little to no accuracy drop. With our efficient system implementation, this memory reduction, in return, enables up to $\mathbf{4}\times$ larger batch size and brings $\mathbf{2.35}\times \sim \mathbf{3.47}\times$ throughput.

2. Background: Attention Inference-Time Workflow

The LLM attention inference-time workflow involves two phases: i) the *prefill* phase, where the input prompt is used to generate KV cache for each transformer layer of LLMs; and ii) the *decoding* phase, where the model uses and updates KV cache to generate the next token, one at a time.

Prefill Phase. Let $\boldsymbol{X} \in \mathbb{R}^{b \times l_{\text{prompt}} \times d}$ be the input tensor, where b is the batch size, l_{prompt} is the length of the input prompt, and d is the model hidden size. For convenience, we ignore the layer index here. The key, value tensors can be computed by

$$\boldsymbol{X}_K = \boldsymbol{X}\boldsymbol{W}_K, \boldsymbol{X}_V = \boldsymbol{X}\boldsymbol{W}_V,$$

where $\boldsymbol{W}_K, \boldsymbol{W}_V \in \mathbb{R}^{d \times d}$ are the key and value layer weight, respectively. After obtaining $\boldsymbol{X}_K$ and $\boldsymbol{X}_V$, they are cached in the memory for the ease of decoding.

Decoding Phase. Let $\boldsymbol{t} \in \mathbb{R}^{b\times 1\times d}$ be the current input token embedding. Let $\boldsymbol{t}_K = \boldsymbol{t}\boldsymbol{W}_K$ and $\boldsymbol{t}_V = \boldsymbol{t}\boldsymbol{W}_V$ be the key and value layer output, respectively. We first update KV cache:

$$\boldsymbol{X}_K \leftarrow \text{Concat}(\boldsymbol{X}_K, \boldsymbol{t}_K),$$
$$\boldsymbol{X}_V \leftarrow \text{Concat}(\boldsymbol{X}_V, \boldsymbol{t}_V),$$

then calculate the attention output as:

$$\begin{aligned} \boldsymbol{t}_Q &= \boldsymbol{t}\boldsymbol{W}_Q, \\ \boldsymbol{A} &= \text{Softmax}(\boldsymbol{t}_Q \boldsymbol{X}_K^\top), \\ \boldsymbol{t}_O &= \boldsymbol{A}\boldsymbol{X}_V, \end{aligned} \tag{1}$$

where $\boldsymbol{W}_Q$ is the weight matrix of the query layer. For ease of illustration, we ignore the attention output layer and the other parts of the inference workflow.

Memory and Speed Analysis. The above process is repeated until a special token indicating the sentence's conclusion is reached. Let $l_{\textbf{gen}}$ be the number of generated tokens. From the above analysis, the shape of KV cache is $b \times (l_{\textbf{prompt}} + l_{\textbf{gen}}) \times d$. To get a sense of the scale, consider the OPT-175B model with a batch size b 512, a prompt length $l_{\textbf{prompt}}$ 512, and an output length $l_{\textbf{gen}}$ 32. The KV cache requires 1.2TB, which is 3.8 times the model weights (Sheng et al., 2023). Besides the memory, the inference speed is also decided by the KV cache size. The GPU needs to load KV cache from GPU main memory to GPU SRAM once for every token generated during which the computational core of the chip is essentially idle (Pope et al., 2023; Kwon et al., 2023).

3. Methodology

In scenarios with long contexts or batched inferences, the memory and speed bottlenecks are storing and loading KV cache. The most simple and effective way to alleviate this problem is to reduce the total bytes occupied by KV cache, specifically, quantization. Following this motivation, we first evaluate the performance of the existing quantization method in Section 3.1. Our observations suggest that key and value cache should be quantized along different dimensions. We analyze the rationale behind this observation in Section 3.2. Then based on the analysis, we propose KIVI, a new KV cache quantization method along with its streaming data structure, detailed in Section 3.3.

3.1. Preliminary Study of KV Cache Quantization

As we analyzed in Section 2, KV cache functions as a streaming data structure, where the new tensor arrives sequentially. Thus, optimization-based methods like GPTQ (Frantar et al., 2022) are unsuitable for quantizing KV cache due to the overhead. To the best of our knowledge, the most flexible way for quantizing KV cache is the round-to-nearest quantization. The $B-$bit integer quantization-dequantization process can be expressed as:

$$Q(\boldsymbol{X}) = \lfloor \frac{\boldsymbol{X} - z_X}{s_X} \rceil, \quad \boldsymbol{X}' = Q(\boldsymbol{X}) \cdot s_X + z_X,$$

where $z_X = \min \boldsymbol{X}$ is the zero-point, $s_X = (\max \boldsymbol{X} - \min \boldsymbol{X})/(2^B - 1)$ is the scaling factor, and $\lfloor \cdot \rceil$ is the rounding operation. Here we ignore the batch size for ease of understanding. As shown in Figure 1, $\boldsymbol{X}$ is quantized along either the token or channel dimension group-wisely.

Considering the streaming nature of KV cache, previous studies often apply per-token quantization to both key and value cache since the newly quantized KV cache can be naively added to the existing quantized one along the token dimension (Sheng et al., 2023). While per-channel quantization is non-trivial, we have designed a padding method to implement per-channel quantization to explore its effect on both key and value cache.

Table 1: The results of simulated KV cache group-wise quantization with various configurations. The group size is set as 32. $\mathbb{C}$ stands for per-channel quantization and $\mathbb{T}$ stands for per-token quantization. Please check the whole evaluation in Table 3.

Llama-2-13B	CoQA	TruthfulQA
16bit	66.37	29.53
4bit (K - $\mathbb{T}$, V - $\mathbb{T}$)	66.48	29.51
2bit (K - $\mathbb{T}$, V - $\mathbb{T}$)	52.93	24.98
2bit (K - $\mathbb{C}$, V - $\mathbb{C}$)	2.88	0.74
2bit (K - $\mathbb{T}$, V - $\mathbb{C}$)	2.80	0.26
2bit (K - $\mathbb{C}$, V - $\mathbb{T}$)	**63.53**	**28.60**

Setting. In Table 1, we show the results of fake KV cache group-wise quantization with different configurations on the Llama-2-13B model for the CoQA and TruthfulQA tasks. We use a group size of 32 for all configurations. Here fake quantization means we simulate the quantization process by first quantizing KV cache into lower precision and then dequantizing it in the attention layer. For per-channel quantization, if the number of tokens is not divided evenly into groups, we add zero-padding to ensure it can be grouped perfectly. In this way, we ensure that all tokens in KV cache are quantized for a fair comparison. The detailed experimental setting can be found in Section 4.1. Specifically, we observe that:

OB 1. When using the commonly used per-token quantization to both key and value caches, INT4 precision can maintain accuracy. However, reducing it to INT2 results in a notable accuracy drop.

OB 2. When value cache is quantized per-channel, the accuracy significantly worsens regardless of how key cache

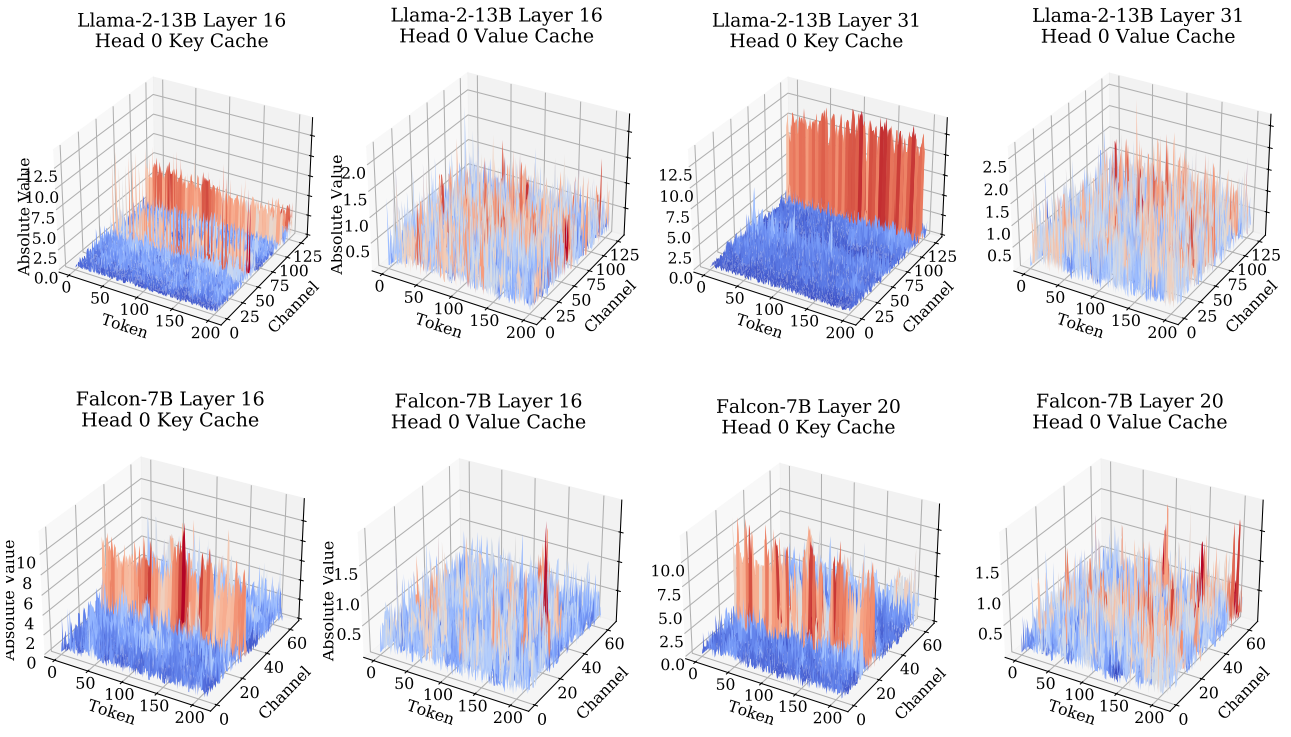

Figure 2: Magnitude of key and value cache for Llama-2-13B and Falcon-7B. We observe (1) for key cache, there are a few channels whose magnitudes are very large. (2) for value cache, there is no obvious outlier pattern.

is quantized.

OB 3. When using a lower numerical precision such as INT2, the most accurate approach is to quantize key cache per-channel and value cache per-token.

3.2. Why Key and Value Cache Should Quantize Along Different Dimensions?

In Table 1, we observe that quantizing key cache per-channel and value cache per-token to 2bit results in a very small accuracy drop. Here we analyze why this configuration delivers better accuracy. In Figure 2 we visualize the original KV cache distribution at different layers. We observe that **in key cache, some fixed channels exhibit very large magnitudes, whereas in value cache, there is no significant pattern for outliers.**

Analysis of Key Cache. The above observation for key cache aligns with previous findings that certain fixed columns in activations exhibit larger outliers (Dettmers et al., 2022; Lin et al., 2023). The persistence of outliers within each channel means that per-channel quantization can confine the quantization error to each individual channel without impacting the other normal channels. Thus, Figure 2 explains why key cache should be quantized per-channel. In Table 2 we show key cache relative reconstruction error $\|\frac{\boldsymbol{X}_K-\boldsymbol{X}'_K}{\boldsymbol{X}_K}\|_F$, along with the relative attention score error $\|\frac{\boldsymbol{A}-\boldsymbol{A}'}{\boldsymbol{A}}\|_F$ where $\boldsymbol{A}' = \text{Softmax}(\boldsymbol{t}_Q \boldsymbol{X}'^{\top}_K)$. We observe that the per-token quantization can lead to almost $5\times$ larger attention score error than per-channel quantization, which is consistent with Figure 2.

Table 2: The relative error statistics averaged over all layers and all heads

Llama-2-13B	K Per-Token	K Per-Channel
Avg. $\|\frac{\boldsymbol{X}_K-\boldsymbol{X}'_K}{\boldsymbol{X}_K}\|_F$	13.67	4.55
Avg. $\|\frac{\boldsymbol{A}-\boldsymbol{A}'}{\boldsymbol{A}}\|_F$	47.0	9.6
Attention sparsity	84.3%	
	V Per-Token	V Per-Channel
Avg. $\|\frac{\boldsymbol{X}_V-\boldsymbol{X}'_V}{\boldsymbol{X}_V}\|_F$	4.57	3.73
Avg. Δ	3.55	49.89

Analysis of Value Cache. Unlike key cache, value cache does not show the channel-wise outlier pattern. Furthermore, Figure 2 alone cannot explain **OB2**, which indicates value cache should only be quantized per-token. This is because Figure 2 implies that errors should be comparable for both per-token and per-channel quantization, given the absence

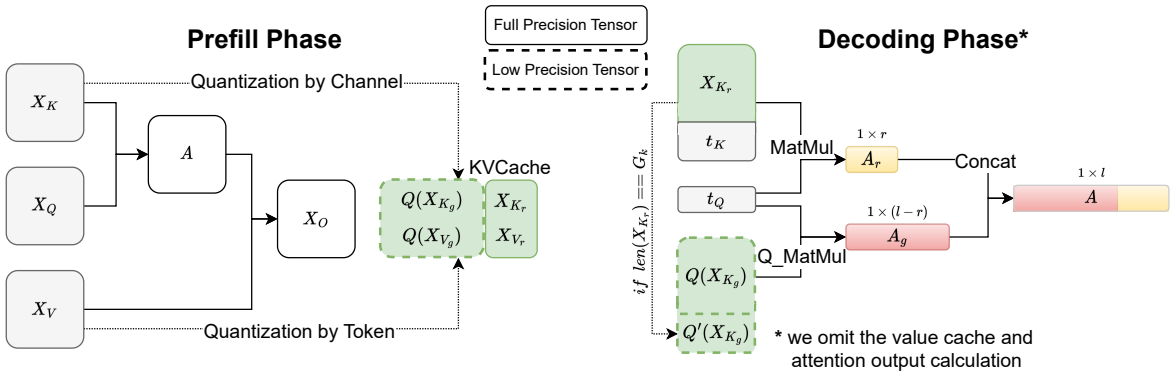

Figure 3: The overview of KIVI algorithm. For ease of illustration, we omit the value cache and attention output parts. The detailed pseudo-code is provided in Algorithm 1. Here "Q_Matmul" is the mix-precision matrix multiplication which fuses the dequantization with matrix multiplication at the tiling level.

of a clear pattern. As shown in Equation (1), value cache is used to calculate the attention output $\boldsymbol{t}_O$. Instead of analyzing the quantization error of value cache $\boldsymbol{X}_V$, in Table 2 we analyze the relative error $\Delta = \|\frac{\boldsymbol{A}\boldsymbol{X}_V - \boldsymbol{A}\boldsymbol{X}'_V}{\boldsymbol{A}\boldsymbol{X}_V}\|_F$ with different quantization configurations. Surprisingly, we observe that the per-token quantization error is almost $15\times$ smaller than per-channel quantization, which explains why **OB2** happens. The intuition behind this observation stems from the attention sparsity. Equation (1) can be written as:

$$[\boldsymbol{A}\boldsymbol{X}_V]_{i*} = \sum_{j=1}^{l_{\text{prompt}}} \boldsymbol{A}_{ij}[\boldsymbol{X}_V]_{j*}, \tag{2}$$

where $[\boldsymbol{X}_V]_{j*}$ is the j-th row of $\boldsymbol{X}_V$. From Equation (2), the attention output is the weighted summation of value cache across various tokens, with the weights being the attention scores. Since the attention score is highly sparse (Tian et al., 2023), the output is just the combination of value caches of a few important tokens. The per-token quantization can confine the error to each individual token. Thus, quantizing other tokens does not affect the accuracy of important tokens. Consequently, per-token quantization leads to a much smaller relative error Δ.

3.3. KIVI: Algorithm and System Support

Algorithm. As we previously analyzed, key cache should be quantized per-channel and value cache should be quantized per-token. Recall that key and value cache of newly generated tokens arrive sequentially. From the implementation perspective, per-token quantization aligns well with streaming settings, allowing newly quantized tensors to be directly appended to the existing quantized value cache by token dimension. However, for per-channel quantization, the quantization process spans across different tokens, which cannot be directly implemented in the streaming setting. As shown in Figure 3, our key idea to solve this problem is to group key cache every G tokens and quantize them separately. Because the number of tokens in $\boldsymbol{X}_K$ can be arbitrary, we split $\boldsymbol{X}_K$ into two parts, namely, the grouped part $\boldsymbol{X}_{K_g} = \boldsymbol{X}_K[:l-r]$ and residual part $\boldsymbol{X}_{K_r} = \boldsymbol{X}_K[l-r:]$, where l is the number of tokens inside the current key cache $\boldsymbol{X}_K$, r is the number of residual tokens, where $l-r$ can be divisible by G.

Since $\boldsymbol{X}_{K_g}$ can be evenly divided into $(l-r)/G$ groups, we only store $Q(\boldsymbol{X}_{K_g})$ with group-wise quantization, while $\boldsymbol{X}_{K_r}$ is kept in full precision. During the decoding process, each newly arrived key cache $\boldsymbol{t}_K$ is added to $\boldsymbol{X}_{K_r}$ and once $\boldsymbol{X}_{K_r}$ reaches R tokens, which is a hyperparameter - residual length, we quantize and concatenate it with the previously quantized $Q(\boldsymbol{X}_{K_G})$. Then we reset $\boldsymbol{X}_{K_r}$ to an empty tensor. We note that R should be divisible by G. With tiled matrix multiplication, the raw attention logits is then calculated as:

$$\begin{aligned}
\boldsymbol{A}_g &= \boldsymbol{t}_Q Q(\boldsymbol{X}_{K_g}^\top), \\
\boldsymbol{X}_{K_r} &= \text{Concat}([\boldsymbol{X}_{K_r}, \boldsymbol{t}_K]), \\
\boldsymbol{A}_r &= \boldsymbol{t}_Q \boldsymbol{X}_{K_r}^\top, \\
\boldsymbol{A} &= \text{Concat}([\boldsymbol{A}_g, \boldsymbol{A}_r]).
\end{aligned} \tag{3}$$

For value cache, similar to key cache, we also split it into two parts and keep the most recent value cache in full precision, namely, $\boldsymbol{X}_{V_g}$ and $\boldsymbol{X}_{V_r}$. Specifically, we maintain a queue and each newly arrived value cache is pushed into the queue. Once the queue reaches the predefined residual length R, the most outdated value cache is poped. Then the poped value cache is quantized per-token and concatenated with the previously quantized value cache along the token dimension.

As shown in Figure 3, we also emphasize that during the prefill phase, the exact key and value tensors are passed to the next layers, although only the quantized KV cache is retained in memory. The whole algorithm can be found in Appendix A Algorithm 1.

Analysis. In KIVI, the grouped key cache $\boldsymbol{X}_{K_g}$ and value cache $\boldsymbol{X}_{V_g}$ is quantized, while the residual key cache $\boldsymbol{X}_{K_r}$ and value cache $\boldsymbol{X}_{V_r}$ is kept in full precision. By design,

there are at most R tokens inside $\boldsymbol{X}_{K_r}$ or $\boldsymbol{X}_{V_r}$. In practice, we set $R \leq 128$ and the sequence length $l_{\text{prompt}} + l_{\text{gen}}$ is often much longer than R. Thus the memory overhead from $\boldsymbol{X}_{K_r}$ and $\boldsymbol{X}_{V_r}$ is negligible when considering the benefit from extreme low-bit quantization, especially for the long context scenarios. Also, since the newly arrived key and value tensors are added to $\boldsymbol{X}_{K_r}$ and $\boldsymbol{X}_{V_r}$ in full precision, KIVI maintains a full precision KV cache sliding window for the local relevant tokens. This window size is expected to be $\frac{R}{2}$ for key cache, and R for value cache. **Later in the experiment section, we show that this full precision sliding window is crucial for obtaining desirable performance on hard tasks, such as GSM8K.**

System Support. We provide a hardware-friendly implementation for running KIVI on GPUs. To minimize the overhead, we have fused the dequantization process with matrix multiplication, e.g., Q_MatMul in Figure 3, using CUDA. We also implement the group-wise quantization kernel in Triton. Our method is fully compatible with weight-only quantization.

4. Experiments

4.1. Settings

Models. We evaluate KIVI using three popular model families: Llama/Llama-2 (Touvron et al., 2023a;b)), Falcon (Penedo et al., 2023) and Mistral (Jiang et al., 2023). Llama and Mistral model is based on multi-head attention, while Falcon is based on multi-query attention (Shazeer, 2019). We use the Hugging Face Transformers codebase and implement the KIVI algorithm upon it. Following previous work (Sheng et al., 2023), the group size G in Algorithm 1 for quantization is set as 32 across all experiments, the residual length R for key and value cache is set to 128.

Tasks. As we analyzed in Section 2, the KV cache size grows larger with a longer context. Thus, we evaluate KIVI under the normal context length and long context setting, respectively. Specifically, we adopt generation tasks from LM-Eval (Gao et al., 2021) for normal context length evaluation and LongBench (Bai et al., 2023) for long context evaluation, respectively[1]. For LM-eval, we adopt CoQA (Exact match accuracy), TruthfulQA (BLEU score), and GSM8K (Exact match accuracy). For LongBench, we chose tasks from four subgroups. Specifically, Qasper (F1 score) is a Single-Document QA task; QMSum (ROUGE score) and MultiNews (ROUGE score) are Summarization tasks; TREC (classification score), TriviaQA (F1 score), and SAMSum (ROUGE score) are Few-shot Learning tasks; and LCC (similarity score) and RepoBench-P (similarity score) is Code Completion task. The maximum sequence length in LongBench was set to 8192 for the Mistral model and 4096 for other models.

[1]The closed-end tasks such as MMLU are not ideal to evaluate KIVI since they only involve one decoding step and directly fetch the output logits, which is not suitable for studying the impact of compressed KV cache.

4.2. Accuracy and Efficiency Analysis

4.2.1. Comparison Between Different Quantization Configurations

We first utilize the fake quantization to demonstrate the effectiveness of our asymmetric quantization, namely, quantizing key cache per-channel and value cache per-token. Here fake quantization is exactly the same as in Table 1. The results are shown in Table 3. We observe that "2bit (K per-channel, V per-token)" consistently achieves the best results compared to all other configurations. This is consistent with our previous analysis. We also note that for hard generation tasks such as GSM8K, the fake "2bit (K per-channel, V per-token)" quantization results are significantly worse than the full precision counterparts. However, for KIVI in Table 3, we observe that the accuracy drop is only around 2% for GSM8K across different models. As we analyzed in Section 3.3, the difference between fake "2bit (K per-channel, V per-token)" quantization and KIVI is that KIVI maintains a full precision key and value cache sliding window for the local relevant tokens. This sliding window is crucial to maintaining accuracy for hard generation tasks such as mathematical reasoning.

4.2.2. Accuracy Comparison on Generation Tasks

LM-Eval Results. We benchmark KIVI on CoQA, TruthfulQA, and GSM8K tasks using LM-Eval framework. All dataset parameters were set to default. We compare the standard 16bit configuration with our KIVI compression techniques across Llama-2-7B, Llama-2-13B, Falcon-7B, and Mistral-7B. As shown in Table 3, we observe that for the Llama and Mistral model, KIVI only has up to 2% accuracy drop despite the KV cache being stored in 2bit. For instance, in the Llama-2-7B model, the transition from 16bit to 2bit only slightly decreases accuracy. Similar trends are observed in other Llama-family models. Since Falcon-7B adopts multi-query attention and only has one head for KV cache, it is already highly compressed compared to Llama-based models. Thus, in Table 3, 4bit KIVI is needed to maintain the accuracy, while 2bit KIVI may have a large accuracy drop in this case.

LongBench Results. The performance of KIVI over various models in the LongBench dataset is summarised in Table 4. We apply KIVI over Llama2-7B, Llama2-13B, Llama2-

Table 3: Performance comparison between 16bit, 4-bit per-token quantization, four fake 2bit KV cache quantization similar to those in Table 1, KIVI-2 (2bit) / KIVI-4 (4bit) across various models. We emphasize that unlike KIVI, which preserves a small portion of full precision key cache $\boldsymbol{X}_{K_r}$ and value cache $\boldsymbol{X}_{V_r}$, all tokens in fake KV cache quantization are quantized for a fair comparison. $\mathbb{C}$ stands for per-channel quantization and $\mathbb{T}$ stands for per-token quantization.

Model		CoQA	TruthfulQA	GSM8K
Llama-2-7B	16bit	63.88	30.76	13.50
	4bit (K - $\mathbb{T}$, V - $\mathbb{T}$)	64.82	29.85	12.28
	2bit (K - $\mathbb{C}$, V - $\mathbb{T}$)	59.08	33.10	5.76
	2bit (K - $\mathbb{T}$, V - $\mathbb{T}$)	39.88	18.29	0.83
	2bit (K - $\mathbb{C}$, V - $\mathbb{C}$)	3.60	0.27	0.00
	2bit (K - $\mathbb{T}$, V - $\mathbb{C}$)	1.30	0.49	0.08
	KIVI-4	63.78	30.80	13.80
	KIVI-2	63.05	33.95	12.74
Llama-2-13B	16bit	66.37	29.53	22.67
	4bit (K - $\mathbb{T}$, V - $\mathbb{T}$)	66.73	29.14	20.92
	2bit (K - $\mathbb{C}$, V - $\mathbb{T}$)	63.53	28.60	12.21
	2bit (K - $\mathbb{T}$, V - $\mathbb{T}$)	52.93	24.98	4.55
	2bit (K - $\mathbb{C}$, V - $\mathbb{C}$)	2.88	0.74	0.00
	2bit (K - $\mathbb{T}$, V - $\mathbb{C}$)	2.80	0.26	0.08
	KIVI-4	66.38	29.49	23.65
	KIVI-2	66.23	29.84	20.77
Falcon-7B	16bit	59.83	23.20	4.55
	4bit (K - $\mathbb{T}$, V - $\mathbb{T}$)	58.53	22.94	3.26
	2bit (K - $\mathbb{C}$, V - $\mathbb{T}$)	43.93	20.82	1.29
	2bit (K - $\mathbb{T}$, V - $\mathbb{T}$)	25.72	0.91	0.53
	2bit (K - $\mathbb{C}$, V - $\mathbb{C}$)	41.95	17.11	1.52
	2bit (K - $\mathbb{T}$, V - $\mathbb{C}$)	19.53	0.94	0.15
	KIVI-4	59.67	22.58	4.47
	KIVI-2	57.48	24.98	3.41
Mistral-7B	16bit	67.40	30.45	38.36
	4bit (K - $\mathbb{T}$, V - $\mathbb{T}$)	67.80	29.83	36.85
	2bit (K - $\mathbb{C}$, V - $\mathbb{T}$)	61.65	29.64	26.46
	2bit (K - $\mathbb{T}$, V - $\mathbb{T}$)	54.55	25.86	5.00
	2bit (K - $\mathbb{C}$, V - $\mathbb{C}$)	24.40	24.86	2.27
	2bit (K - $\mathbb{T}$, V - $\mathbb{C}$)	10.73	19.12	0.99
	KIVI-4	66.95	30.49	37.30
	KIVI-2	66.35	32.17	36.01

7B-Chat, Llama2-13B-Chat, Falcon-7B and Mistral-7B. Table 4 suggests that KIVI is an effective method for KV cache compression with minimal impact on accuracy across various hard long context generation tasks. **We also present additional results using Mistral-7B-v0.2 and LongChat-7B-v1.5 on LongBench, which can be found in Table 8 and Table 9, respectively.**

4.2.3. Ablation

In this section, we benchmark KIVI on GSM8K, one of the hardest generation tasks, to show the effect of hyperparameters group size G and residual length R on the model performance. For full results of KIVI with a residual length of 32, please refer to Appendix B.

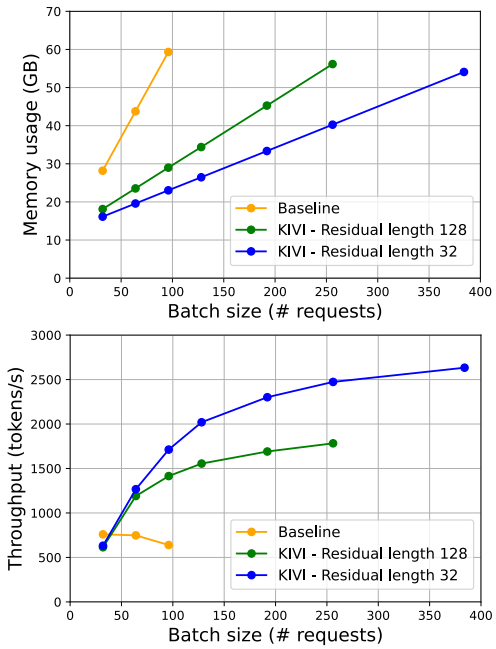

Figure 4: Memory usage and throughput comparison between 2bit KIVI and 16bit baseline. KIVI can achieve higher throughput by enabling a larger batch size.

The effect of group size. We fix the residual length at 128 and vary the group sizes to 32, 64, and 128. From Table 5, we observe that group sizes 32 and 64 yield similar results, whereas the performance significantly decreases when the group size reaches 128. Note the zero-point and the scaling factor mentioned in Section 3.1 are calculated according to this group size; where the choice of group size will greatly impact the KV cache compression effect under a long input.

The effect of residual length. We fix the group size at 32 and vary the residual length across 32, 64, 96, and 128. As shown in Table 5, there is no consistent pattern between residual lengths and model accuracy. Namely, while a residual length of 128 achieves good results, 32 and 96 yield similar outcomes, but a residual length of 64 results in the worst performance. We emphasize that while we observe no significance among residual lengths of $\{32, 96, 128\}$, **having a reasonably large residual length is important; as it brings much performance boosts on hard tasks like GSM8K, again as shown in Table 5.**

4.2.4. Efficiency Comparison

To evaluate the wall-clock time efficiency of KIVI, following vLLM (Kwon et al., 2023), we synthesize workloads based on ShareGPT (sha, 2023), which contain input and output texts of real LLM services. On average, the data set has an input prompt length l_{prompt} of 161 and an output length l_{gen} of 338 (Kwon et al., 2023). We increase the batch

Table 4: Performance evaluation of KIVI on various models across a range of benchmarks in LongBench. We highlight the average performance of our method. **More similar results on Mistral-7B-v0.2 and LongChat-7b-v1.5 can be found in Table 8 and Table 9**

Model		Qasper	QMSum	MultiNews	TREC	TriviaQA	SAMSum	LCC	RepoBench-P	Average
Llama2-7B	16bit	9.52	21.28	3.51	66	87.72	41.69	66.66	59.82	44.52
	KIVI-4	9.28	21.42	3.88	66	87.72	41.82	66.8	59.83	44.59
	KIVI-2	9.31	20.5	1.14	66	87.42	42.71	66.88	60.23	**44.27**
Llama2-13B	16bit	9.32	21.38	3.71	70	87.87	43.55	66.61	56.42	44.85
	KIVI-4	9.16	20.86	3.21	69	86.97	44.26	65.3	57.08	44.48
	KIVI-2	8.58	20.69	6.19	69.5	87.78	44.3	65.08	55.46	**44.69**
Llama2-7B-Chat	16bit	19.65	20.54	26.36	63	84.28	41.12	59.75	52.93	45.95
	KIVI-4	19.62	20.7	25.49	63	84.13	40.87	59.27	53.56	45.83
	KIVI-2	19.32	20.46	25.48	63	84.84	40.6	58.71	52.97	**45.67**
Llama2-13B-Chat	16bit	24.18	20.37	25.69	67.5	86.9	42.18	50.23	50.64	45.96
	KIVI-4	23	20.36	26.06	67.5	87.2	42.04	52.55	52.77	46.44
	KIVI-2	23.59	20.76	25.25	67.5	87.17	41.56	49.93	48.45	**45.52**
Falcon-7B	16bit	1.48	2.35	11.09	13	5.84	2.44	23.86	9.69	8.71
	KIVI-4	1.04	2.41	11.98	13	5.84	2.36	23.72	9.92	**8.78**
	KIVI-2	1.98	3.61	6.78	10	6.24	2.73	22.18	10.12	**7.95**
Mistral-7B	16bit	8.12	19.98	19.99	67.5	89.8	41.69	66.59	58.99	46.58
	KIVI-4	7.89	20.06	20.58	67.5	89.8	41.56	66.45	58.62	46.56
	KIVI-2	6.92	19.71	17.92	66.5	89.63	41.66	65.52	58.99	**45.85**

Table 5: Ablation study of KIVI by changing group size G and residual length R.

Model	Group Size	GSM8K
Llama2-13B	32	20.77
	64	21.00
	128	17.29

Model	Residual Length	GSM8K
Llama2-13B	32	20.62
	64	19.86
	96	20.55
	128	20.77

size until out of memory and report the peak memory usage and throughput between KIVI (with residual length 32 and 128) and FP16 baseline for the Llama-2-7B model. The hardware here is a single NVIDIA A100 GPU (80GB).

As shown in Figure 4, with similar maximum memory usage, KIVI enables up to $\mathbf{4}\times$ larger batch size and gives $\mathbf{2.35}\times \sim \mathbf{3.47}\times$ larger throughput. This throughput number can grow larger with longer context length and output length. We also note that **this speed-up can be greatly increased if we further fuse the KV cache quantization process with previous operations.** We leave it as one of future work.

5. Related Work

Many machine learning system work have been proposed to scale up LLM training and inference process (Pope et al., 2023; Wang et al., 2023; 2022). Among them, quantization techniques have been widely applied (Frantar et al., 2022; Lin et al., 2023; Kim et al., 2023; Xu et al., 2023). A main branch of LLM quantization is weight-only quantization, which involves the quantization of model weights to lower precision. For instance, AWQ (Lin et al., 2023) cleverly quantizes model weights to INT4 and INT3 using an activation-aware manner. GPTQ (Frantar et al., 2022) utilizes approximate second-order information to quantize model weights both accurately and efficiently. SqueezeLLM (Kim et al., 2023) adopts the concept of sensitivity-based non-uniform quantization along with Dense-and-Sparse decomposition. This line of work is orthogonal to ours, as they can be combined together.

SmoothQuant (Xiao et al., 2023a) is a post-training quantization method that is more closely related to our work. This method uses equivalent transformations to balance the quantization complexity for both activation and weight, making the activation easier to quantize. SmoothQuant can compress KV cache to 8bit with minor performance loss. However, it faces a significant accuracy drop when scaled down to 4bit or less (Zhao et al., 2024). FlexGen adopts 4-bit per-token quantization for both key and value cache.

One important recipe of KIVI is the per-channel quantization scheme designed base on observation made in Section 3.2 and Figure 2. ATOM (Zhao et al., 2024) also identifies that the Key cache exhibits more outliers compared to the Value cache. KIVI provides further extensive analysis and leverages this observation to implement per-channel

quantization. A similar observation and approach have been independently discovered and developed in the concurrent work KVQuant (Hooper et al., 2024).

vLLM (Kwon et al., 2023) and S3 (Jin et al., 2023) are system-level works, which include memory management through the use of PagedAttention or memory usage prediction. They can lower the memory requirements of KV cache and simultaneously increase model throughput. This research direction is orthogonal to our work, since system-level optimizations can also be applied upon our algorithm.

Several other works also consider compressing KV cache by evicting tokens. H2O (Zhang et al., 2023) retains only a small portion of tokens that contribute significantly to the attention scores. Similarly, Scissorhands (Liu et al., 2024) exploit the persistence of the importance hypothesis in KV cache sparsification. StreamingLLM (Xiao et al., 2023b) is based on the observation of "attention sink" and maintains only a few initial tokens to preserve performance. Unlike these works, our KIVI retains all input tokens and compresses them into lower precision. This line of work is orthogonal to ours, as they also can be combined together.

6. Conclusion and Future Work

In this paper, we systematically analyze KV cache element distribution in popular LLMs. We conclude that key cache should be quantized per-channel and value cache should be quantized per token. Based on these observations, we propose KIVI, a plug-and-play 2bit KV cache quantization algorithm without the need for any tuning. In real LLM workload, KIVI allows up to $\mathbf{4}\times$ larger batch sizes and $\mathbf{3.47}\times$ throughput. In the future, we will further optimize the implementation to reduce the overhead of quantization process during the prefill and decoding phase.

Acknowledgments

The authors thank the anonymous reviewers for their helpful comments. Dr. Beidi Chen is supported by a research gift from Moffett.AI. Dr. Vladimir Braverman is partially supported by the Ministry of Trade, Industry and Energy (MOTIE) and Korea Institute for Advancement of Technology (KIAT) through the International Cooperative R&D program, the Naval Research (ONR) grant N00014-23-1-2737, and NSF CNS 2333887 award. Dr. Xia Hu is supported by NSF grants NSF IIS-2224843. The views and conclusions contained in this paper are those of the authors and should not be interpreted as representing any funding agencies.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

ShareGPT Team. https://sharegpt.com/, 2023.

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245*, 2023.

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. Longbench: A bilingual, multitask benchmark for long context understanding. *arXiv preprint arXiv:2308.14508*, 2023.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm. int8 (): 8-bit matrix multiplication for transformers at scale. *arXiv preprint arXiv:2208.07339*, 2022.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

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. *Version v0. 0.1. Sept*, page 8, 2021.

Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *arXiv preprint arXiv:2401.18079*, 2024.

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

Yunho Jin, Chun-Feng Wu, David Brooks, and Gu-Yeon Wei. S3: Increasing gpu utilization during generative inference for higher throughput. *arXiv preprint arXiv:2306.06000*, 2023.

Sehoon Kim, Coleman Hooper, Amir Gholami, Zhen Dong, Xiuyu Li, Sheng Shen, Michael W Mahoney, and Kurt Keutzer. Squeezellm: Dense-and-sparse quantization. *arXiv preprint arXiv:2306.07629*, 2023.

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 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Xingyu Dang, and Song Han. Awq: Activation-aware weight quantization for llm compression and acceleration. *arXiv preprint arXiv:2306.00978*, 2023.

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. *Advances in Neural Information Processing Systems*, 36, 2024.

Guilherme Penedo, Quentin Malartic, Daniel Hesslow, Ruxandra Cojocaru, Alessandro Cappelli, Hamza Alobeidli, Baptiste Pannier, Ebtesam Almazrouei, and Julien Launay. The refinedweb dataset for falcon llm: outperforming curated corpora with web data, and web data only. *arXiv preprint arXiv:2306.01116*, 2023.

Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems*, 5, 2023.

Noam Shazeer. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019.

Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pages 31094–31116. PMLR, 2023.

Ross Taylor, Marcin Kardas, Guillem Cucurull, Thomas Scialom, Anthony Hartshorn, Elvis Saravia, Andrew Poulton, Viktor Kerkez, and Robert Stojnic. Galactica: A large language model for science. *arXiv preprint arXiv:2211.09085*, 2022.

Yuandong Tian, Yiping Wang, Beidi Chen, and Simon Du. Scan and snap: Understanding training dynamics and token composition in 1-layer transformer. *arXiv preprint arXiv:2305.16380*, 2023.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023a.

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*, 2023b.

Zhuang Wang, Zhaozhuo Xu, Xinyu Wu, Anshumali Shrivastava, and TS Eugene Ng. Dragonn: Distributed randomized approximate gradients of neural networks. In *International Conference on Machine Learning*, pages 23274–23291. PMLR, 2022.

Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023.

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*, pages 38087–38099. PMLR, 2023a.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. *arXiv preprint arXiv:2309.17453*, 2023b.

Zhaozhuo Xu, Zirui Liu, Beidi Chen, Yuxin Tang, Jue Wang, Kaixiong Zhou, Xia Hu, and Anshumali Shrivastava. Compress, then prompt: Improving accuracy-efficiency trade-off of llm inference with transferable prompt. *arXiv preprint arXiv:2305.11186*, 2023.

Jiayi Yuan, Ruixiang Tang, Xiaoqian Jiang, and Xia Hu. Large language models for healthcare data augmentation: An example on patient-trial matching. In *AMIA Annual Symposium Proceedings*, volume 2023, page 1324. American Medical Informatics Association, 2023.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. H _2 o: Heavy-hitter oracle for efficient generative inference of large language models. *arXiv preprint arXiv:2306.14048*, 2023.

Yilong Zhao, Chien-Yu Lin, Kan Zhu, Zihao Ye, Lequn Chen, Size Zheng, Luis Ceze, Arvind Krishnamurthy, Tianqi Chen, and Baris Kasikci. Atom: Low-bit quantization for efficient and accurate llm serving. *Proceedings of Machine Learning and Systems*, 6:196–209, 2024.

A. Detailed Implementations

In this section, we present the algorithm for KIVI as discussed in Section 3.3. Specifically, we provide the pseudocode for KIVI when calculating the attention output in the prefill and decoding phases.

Algorithm 1: The KIVI Prefill & Decoding Algorithm

parameter: group size G, residual length R

procedure `Prefill`**:**

Input: $\boldsymbol{X} \in \mathbb{R}^{l_{\text{prompt}} \times d}$

$\boldsymbol{X}_K = \boldsymbol{X}\boldsymbol{W}_K, \boldsymbol{X}_V = \boldsymbol{X}\boldsymbol{W}_V$

$\boldsymbol{X}_{V_g} = \boldsymbol{X}_V[: l_{\text{prompt}} - R], \boldsymbol{X}_{V_r} = \boldsymbol{X}_V[l_{\text{prompt}} - R :]$

$Q(\boldsymbol{X}_{V_g}) \leftarrow \text{GroupQuant}(\boldsymbol{X}_{V_g}, \text{dim=token}, \text{numGroup=}d//G)$

$Q(\boldsymbol{X}_{K_g}), \boldsymbol{X}_{K_r} \leftarrow$ `KeyQuant`$(\boldsymbol{X}_K)$

`KV cache` $\leftarrow Q(\boldsymbol{X}_{K_g}), \boldsymbol{X}_{K_r}, Q(\boldsymbol{X}_{V_g}), \boldsymbol{X}_{V_r}$

return $\boldsymbol{X}_K, \boldsymbol{X}_V$

end

procedure `Decoding`**:**

Input: `KV cache`, $\boldsymbol{t} \in \mathbb{R}^{1 \times d}$

$\boldsymbol{t}_Q = \boldsymbol{t}\boldsymbol{W}_Q, \boldsymbol{t}_K = \boldsymbol{t}\boldsymbol{W}_K, \boldsymbol{t}_V = \boldsymbol{t}\boldsymbol{W}_V$

$Q(\boldsymbol{X}_{K_g}), \boldsymbol{X}_{K_r}, Q(\boldsymbol{X}_{V_g}), \boldsymbol{X}_{V_r} \leftarrow$ `KV cache`

$\boldsymbol{X}_{K_r} \leftarrow \text{Concat}([\boldsymbol{X}_{K_r}, \boldsymbol{t}_K], \text{dim=token})$

$\boldsymbol{X}_{V_r} \leftarrow \text{Concat}([\boldsymbol{X}_{V_r}, \boldsymbol{t}_V], \text{dim=token})$

if *len*$(\boldsymbol{X}_{K_r}) = R$ **then**

$Q(\boldsymbol{X}_{K_r}), \_ \leftarrow$`KeyQuant`$(\boldsymbol{X}_{K_r})$

$Q(\boldsymbol{X}_{K_g}) \leftarrow \text{Concat}([Q(\boldsymbol{X}_{K_g}), Q(\boldsymbol{X}_{K_r})], \text{dim=token})$

$\boldsymbol{X}_{K_r} \leftarrow$ empty tensor.

end

if *len*$(\boldsymbol{X}_{V_r}) > R$ **then**

$Q(\boldsymbol{X}_{V_r'}) \leftarrow \text{GroupQuant}(\boldsymbol{X}_{V_r}[: -R], \text{dim=token}, \text{numGroup} = d//G)$

$Q(\boldsymbol{X}_{V_g}) \leftarrow \text{Concat}([Q(\boldsymbol{X}_{V_g}), Q(\boldsymbol{X}_{V_r'})], \text{dim=token})$

$\boldsymbol{X}_{V_r} \leftarrow \boldsymbol{X}_{V_r}[-R :]$

end

$\boldsymbol{A} \leftarrow \text{Concat}([\boldsymbol{t}_Q Q(\boldsymbol{X}_{K_g})^\top, \boldsymbol{t}_Q \boldsymbol{X}_{K_r}^\top], \text{dim=token})$

$\boldsymbol{A}_g = \text{Softmax}(\boldsymbol{A})[: -R], \boldsymbol{A}_r = \text{Softmax}(\boldsymbol{A})[-R :]$

$\boldsymbol{t}_O \leftarrow \boldsymbol{A}_g Q(\boldsymbol{X}_{V_g}) + \boldsymbol{A}_r \boldsymbol{X}_{V_r}$

`KV cache` $\leftarrow Q(\boldsymbol{X}_{K_g}), \boldsymbol{X}_{K_r}, Q(\boldsymbol{X}_{V_g}), \boldsymbol{X}_{V_r}$

return $\boldsymbol{t}_O$

end

function `KeyQuant`$(\boldsymbol{X}_K \in \mathbb{R}^{l \times d})$**:**

$r = l\%R$,

$\boldsymbol{X}_{K_g} = \boldsymbol{X}_K[: l - r], \boldsymbol{X}_{K_r} = \boldsymbol{X}_K[l - r :]$

$Q(\boldsymbol{X}_{K_g}) \leftarrow \text{GroupQuant}(\boldsymbol{X}_{K_g}, \text{dim=channel}, \text{numGroup=}l//G)$

return $Q(\boldsymbol{X}_{K_g}), \boldsymbol{X}_{K_r}$

end

B. More Experimental Results

In our efficiency evaluation, we observe that with a residual length of 32, KIVI achieves a significantly higher memory compression rate, which in turn leads to increased throughput. Additionally, our ablation study reveals that changing the residual length from 128 to 32 does not result in a substantial performance gap. We demonstrate KIVI with a residual length of 32 across all benchmark datasets. As shown in Tables 6 and 7, KIVI with a residual length of 32 also delivers performance comparable to that of the 16-bit full model.

We also present additional results using Mistral-7B-v0.2 and LongChat-7B-v1.5 on LongBench, which can be found in Table 8 and Table 9, respectively.

Table 6: Performance comparison between 16bit, KIVI-2 (2bit) / KIVI-4 (4bit) with residual length 128 and 32 across various models. $R32$ stands for residual length 32.

Model		CoQA	TruthfulQA	GSM8K
Llama-2-7B	16bit	63.88	30.76	13.50
	KIVI-2 $R128$	63.05	33.95	12.74
	KIVI-2 $R32$	62.85	33.01	13.57
Llama-2-13B	16bit	66.37	29.53	22.67
	KIVI-2 $R128$	66.23	29.84	20.77
	KIVI-2 $R32$	66.57	29.35	20.62
Falcon-7B	16bit	59.83	23.20	4.55
	KIVI-4 $R128$	59.67	22.58	4.47
	KIVI-4 $R32$	59.73	22.96	3.94
	KIVI-2 $R128$	57.48	24.98	3.41
	KIVI-2 $R32$	57.50	25.70	2.20
Mistral-7B	16bit	67.40	30.45	38.36
	KIVI-2 $R128$	66.35	32.17	36.01
	KIVI-2 $R32$	65.90	31.21	34.34

Table 7: Performance evaluation of KIVI with residual length 128 and 32 on various models across a range of benchmarks in LongBench. $R32$ stands for residual length 32.

Model		Qasper	QMSum	MultiNews	TREC	TriviaQA	SAMSum	LCC	RepoBench-P	Average
Llama2-7B	16bit	9.52	21.28	3.51	66	87.72	41.69	66.66	59.82	44.52
	KIVI-2 $R128$	9.31	20.5	1.14	66	87.42	42.71	66.88	60.23	44.27
	KIVI-2 $R32$	9.26	20.53	0.97	66	87.42	42.61	66.22	59.67	44.08
Llama2-13B	16bit	9.32	21.38	3.71	70	87.87	43.55	66.61	56.42	44.85
	KIVI-2 $R128$	8.58	20.69	6.19	69.5	87.78	44.3	65.08	55.46	44.69
	KIVI-2 $R32$	8.38	20.74	7.01	69.5	87.78	44.43	64.89	55.31	44.75
Llama2-7B-Chat	16bit	19.65	20.54	26.36	63	84.28	41.12	59.75	52.93	45.95
	KIVI-2 $R128$	19.32	20.46	25.48	63	84.84	40.6	58.71	52.97	45.67
	KIVI-2 $R32$	19.1	20.08	25.33	63	85.04	39.8	57.91	52.38	45.33
Llama2-13B-Chat	16bit	24.18	20.37	25.69	67.5	86.9	42.18	50.23	50.64	45.96
	KIVI-2 $R128$	23.59	20.76	25.25	67.5	87.17	41.56	49.93	48.45	45.52
	KIVI-2 $R32$	23.56	20.9	25.45	67.5	87.42	41.4	48.93	48.81	45.49
Falcon-7B	16bit	1.48	2.35	11.09	13	5.84	2.44	23.86	9.69	8.71
	KIVI-4 $R128$	1.04	2.41	11.98	13	5.84	2.36	23.72	9.92	8.78
	KIVI-4 $R32$	1.03	2.45	11.99	13.5	5.84	2.46	23.88	9.95	8.88
	KIVI-2 $R128$	1.98	3.61	6.78	10	6.24	2.73	22.18	10.12	7.95
	KIVI-2 $R32$	2.28	3.23	6.73	10	6.31	2.88	22.71	10.45	8.07
Mistral-7B	16bit	8.12	19.98	19.99	67.5	89.8	41.69	66.59	58.99	46.58
	KIVI-2 $R128$	6.92	19.71	17.92	66.5	89.63	41.66	65.52	58.99	45.85
	KIVI-2 $R32$	6.84	19.81	17.2	66.5	89.63	42.82	65.13	58.06	45.74

Table 8: The results of LongChat-7B-v1.5-32K with KIVI on LongBench. The model has 32K context length. We use a 32 group size and 128 residual length for both KIVI-2 and KIVI-4 The baseline is of full precision.

	NarrativeQA	**Qasper**	**MultiFieldQA**	**HotpotQA**	**MuSiQue**	**2WikiMQA**	**GovReport**	**QMSum**
Baseline	20.65	29.42	43.15	33.05	14.66	24.14	30.85	22.84
w./ KIVI-2	20.79	28.69	41.02	32.91	13.82	23.00	30.47	22.59
w./ KIVI-4	20.49	28.90	43.24	33.07	14.66	24.86	31.40	22.84
	MultiNews	**LCC**	**RepoBench-P**	**TriviaQA**	**SAMSum**	**TRec**	**PR**	**Avg**
Baseline	26.55	54.83	58.94	83.99	40.75	66.50	30.50	38.72
w./ KIVI-2	26.28	54.11	57.62	83.19	41.28	66.50	32.25	38.30
w./ KIVI-4	26.52	54.06	58.77	83.88	40.62	67.00	31.50	38.79

Table 9: The results of Mistral-7B-Instruct-v0.2 with KIVI on LongBench. The model has 32K context length and applies group query attention, which uses 8 heads for KV Cache instead of the full 32 heads. We use a 32 group size and 128 residual length for both KIVI-2 and KIVI-4 The baseline is of full precision.

	NarrativeQA	**Qasper**	**MultiFieldQA**	**HotpotQA**	**MuSiQue**	**2WikiMQA**	**GovReport**	**QMSum**
Baseline	21.02	29.41	47.13	36.53	19.13	21.76	32.59	23.99
w./ KIVI-2	20.61	28.73	44.88	35.47	17.95	20.68	32.55	23.65
w./ KIVI-4	20.97	29.41	46.52	36.25	19.53	21.66	32.97	24.06
	MultiNews	**LCC**	**RepoBench-P**	**TriviaQA**	**SAMSum**	**TRec**	**PR**	**Avg**
Baseline	27.09	53.49	51.40	86.23	43.04	71.00	89.33	43.54
w./ KIVI-2	26.54	53.03	51.16	86.00	43.34	71.00	80.83	42.43
w./ KIVI-4	26.89	53.33	51.41	86.23	43.34	71.00	89.42	43.53