GEAR: An Efficient Error Reduction Framework for KV Cache Compression in LLM Inference

Hao Kang[*†], **Qingru Zhang**[*†], **Souvik Kundu**[◇], **Geonhwa Jeong**[†], **Zaoxing Liu**[†], **Tushar Krishna**[†], **Tuo Zhao**[†]
[†]Georgia Institute of Technology, Atlanta [◇]Intel Labs, San Diego
{hkang342, qingru.zhang}@gatech.edu, {souvikk.kundu}@intel.com

Abstract

Key-value (KV) caching has become the de-facto technique to accelerate generation speed for large language models (LLMs) inference. However, the growing cache demand with increasing sequence length has transformed LLM inference to be a *memory bound* problem, significantly constraining the system throughput. Existing methods rely on dropping unimportant tokens or quantizing entries group-wise. Such methods, however, often incur high approximation errors to represent the compressed matrices. The autoregressive decoding process further compounds the error of each step, resulting in critical deviation in model generation and deterioration of performance. To tackle this challenge, we propose GEAR, an efficient *error reduction framework* that augments a quantization scheme with two error reduction components and achieves near-lossless performance at high compression ratios. GEAR first applies quantization to majority of entries of similar magnitudes to ultra-low precision. It then employs a low-rank matrix to approximate the quantization error, and a sparse matrix to remedy individual errors from outlier entries. By adeptly integrating three techniques, GEAR is able to fully exploit their synergistic potentials. Our experiments show that GEAR can maintain similar accuracy to that of FP16 cache with improvement up to 24.42% over the SOTA baselines at 2-bit compression. Additionally, compared to LLM inference with FP16 KV cache, GEAR can reduce peak-memory of up to $2.39\times$, bringing $2.1\times \sim 5.07\times$ throughput improvement. Our code will be publicly available.

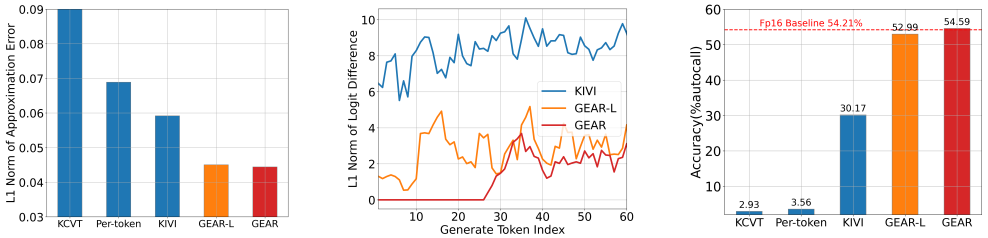
1 Introduction

Autoregressive large language models (LLMs) [4, 39, 27, 28] have marked a significant milestone in natural language processing (NLP) and artificial intelligence (AI) [29, 3, 19], showcasing exceptional performances across a wide range of applications, such as content creation and dialogue systems [36, 26, 32]. When serving these LLMs for generative inference, *KV cache*-ing has become a routine practice. It stores previously computed Key/Value tensors from attention calculation and reuses them while generating next tokens [21], avoiding intensive recalculation to improve the generation speed.

Despite its prominence, the memory consumption of the KV cache grows rapidly with the model size and sequence length, imposing significant constraints on system throughput. For instance, in the case of a 30 billion-parameter LLM with an input length of 1024 and batch size of 128, the resulting KV cache can occupy up to 180 GB of memory [40]. To alleviate this pressure on limited GPU memory capacity, inference systems typically resort to *offloading* [1, 23] – transferring the KV cache to CPU memory or NVMe storage. This, however, can still introduce non-trivial overhead due to the limited PCIe bandwidth between GPUs and CPUs on many devices. Therefore, it is crucial to reduce the intensive memory footprint of the emerging bottleneck of KV cache in generative inference.

[*]Equal Contribution.

4th NeurIPS Efficient Natural Language and Speech Processing Workshop (ENLSP-IV 2024).



(a) Approx. error on GSM8k-CoT (b) Difference in prediction logits (c) Accuracy on GSM8k-CoT

Figure 1: (1a) compares the approximation error when compressing KV caches to 2-bit for LLaMA3-8B on GSM8k (w. CoT). (1b) presents difference in prediction logits from FP16 baseline after compressing KV caches of an GSM8k (w. CoT) example, indicating the approximation error can be severely compounded along steps and critically divert model generations. (1c) shows reducing the error can significantly improve the performance.

To address this issue, *token dropping* methods have been proposed to compress the cache size while maintaining the generative performance [40, 15, 10]. These approaches harness the sparsity observed in attention scores to evict embeddings of less important tokens from the KV cache while retaining frequently attended ones. For example, H_2O [40] utilizes accumulated attention scores to evaluate token importance and reduces cache size by dropping tokens with lower scores. In addition, *quantization* is another widely-adopted compression scheme that maps full-precision tensor values into discrete levels and store them at lower precision, e.g., INT4 or INT8 [37, 7, 23]. For example, FlexGen [23] employs a fine-grained group-wise asymmetric quantization that groups KV entries per-token, divides g contiguous entries as a group, and quantize the tensor group-wise. Two concurrent works [16, 11] further study KV entry distribution and propose to quantize Key cache per-channel and quantize Value cache per-token, compressing the cache size by a high ratio.

The existing methods can effectively compress the cache size to low-precision while achieving near-lossless performance on natural language understanding tasks like multiple-choice QA, text classification, or simple summarization task [40, 16]. However, a stark contrast emerges when applying these methods to complex generative tasks that require models to generate longer responses or involve reasoning, such as mathematical problem-solving [5] and chain-of-thought (CoT) reasoning [33]. Their performance deteriorates under a high compression ratio[2] (e.g., 4-bit/2-bit quantization or dropping $> 50\%$ tokens [10]), which is noticeable in both types of methods[3]. This phenomenon can be attributed to the non-trivial approximation error induced by them, i.e., difference between original KV and the compressed ones. For simple tasks, models are required to generate only few tokens where necessary information for correct prediction can often be derived from a small set of important contextual tokens. Consequently, a relatively large approximation error does not significantly hinder the generation of target tokens. In contrast, the complex tasks require models to generate longer sequences conditioned on prompts that often contains densely correlated information (e.g., CoT reasoning). The autoregressive decoding can compound the approximation error at every step. Consequently, the negative effect of even a relatively small error can be magnified along generation steps, adversely affecting subsequent generation. As an example, Figure 1 presents the approximation error of various methods on GSM8k and illustrates the deviation in token generations due to the accumulated error, which degrades the accuracy a lot. Therefore, the crux of the issue lies in high approximation errors of these methods, especially under high compression ratios.

To address this challenge, we propose *GEAR* (GEnerative Inference with Approximation Error Reduction), an efficient error reduction framework that augments existing KV cache quantization schemes with two error-reduction techniques, and adeptly integrate them to exploit their full potentials. Generally speaking, our framework consists of three components to decompose KV matrices: (i) First, we apply an existing *quantization method* to efficiently quantize the majority (e.g., 98%) of entries of similar magnitudes to ultra-low precision. (ii) Then, we introduce a *low-rank matrix* to efficiently approximate the quantization residuals. (iii) Finally, we employ a *sparse matrix* consisting of a negligible ratio of entries of large magnitudes to remedy the individual errors caused by these outliers. Such a composite approximation decouples the coherent parts from incoherent parts of the approximation error: the low-rank matrix captures the majority of coherent basis of quantization error while the sparse matrix rectifies the incoherency existing in individual outliers. Meanwhile, as shown by our empirical evidence in Section 4.2, these two lightweight components result in negligible memory and computational overheads, demonstrating high efficiency. As such, GEAR can effectively reduce the approximation error in a highly efficient way and achieve superior performance

[2]We define the compression ratio as tensor size in FP16 divided by that in compressed format.

[3]Please refer to Section 4 for our empirical evidence.

on *both* complex and relatively simple tasks at high compression ratios in a *plug-and-play* manner. We find that using both sparse and low-rank components is necessary for GEAR to establish the best performance, highlighting their complementary nature. Remarkably, for those prioritizing efficiency, equipping low-rank approximation alone for quantization can still effectively reduce the approximation error, yielding both significant efficiency and performance improvement. We refer to this lite version of GEAR as *GEAR-L*. Additionally, we incorporate a *streaming buffer* strategy for GEAR to further improve inference efficiency.

We conduct experiments on diverse tasks and models to demonstrate the effectiveness of GEAR. Specifically, we evaluate compression performance with LLaMA2-7B/13B [28], Mistral-7B [12], and LLaMA3-8B[17] on generative tasks including mathematical reasoning (GSM8k[5] and AQuA[14]), symbolic reasoning (BigBench Hard[24]), and long-context understanding (LongBench[2]). We show that GEAR consistently outperforms the baseline methods especially at high compression ratios such as 2-bit precision. For example, when compressing KV caches to 2-bit, GEAR achieves an remarkable average accuracy improvement of **14.95**% over the best-performing baseline across various models and datasets. Regarding the inference efficiency, compared to the FP16 baseline, GEAR can reduce the peak memory up to **2.39**×, bring **2.10**× $\sim$ **5.07**× throughput improvement.

2 Background

Multi-head attention. A typical transformer model consists of L stacked layers, where each layer contains two submodules: a multi-head attention (MHA) and a feed-forward network (FFN). Given the input token embeddings as $\boldsymbol{X} \in \mathbb{R}^{n\times d}$, MHA performs attention function in parallel H heads:

$$\text{MHA}\left(\boldsymbol{X}\right) = \text{Concat}(\boldsymbol{H}^{(1)}, ..., \boldsymbol{H}^{(H)})\boldsymbol{W}_o, \quad \boldsymbol{H}^{(i)} = \text{Softmax}\left(\boldsymbol{Q}^{(i)}\boldsymbol{K}^{(i)\top}/\sqrt{d_H}\right)\boldsymbol{V}^{(i)} \tag{1}$$

where $\boldsymbol{Q}^{(i)} = \boldsymbol{X}\boldsymbol{W}_{q_i}, \boldsymbol{K}^{(i)} = \boldsymbol{X}\boldsymbol{W}_{k_i}, \boldsymbol{V}^{(i)} = \boldsymbol{X}\boldsymbol{W}_{v_i}$ are Query/Key/Value matrices, and $\boldsymbol{W}_{q_i}, \boldsymbol{W}_{k_i}, \boldsymbol{W}_{v_i} \in \mathbb{R}^{d\times d_H}$ are projection matrices of head i. d_H is typically set to d/H.

Prefill and decoding. Suppose the model generates n_g tokens. At the first generation step, the input tokens $\boldsymbol{X}_0 \in \mathbb{R}^{n\times d}$ are prefilled. Then $\boldsymbol{K}^{(i)}$ and $\boldsymbol{V}^{(i)}$ at every head and every layer are cached for subsequent generation, resulting in initial KV caches of prefill phrase: $\boldsymbol{K}_0 = \text{Concat}(\boldsymbol{K}^{(1)}, \ldots, \boldsymbol{K}^{(H)})$, $\boldsymbol{V}_0 = \text{Concat}(\boldsymbol{V}^{(1)}, \ldots, \boldsymbol{V}^{(H)})$ and $\boldsymbol{K}_0, \boldsymbol{V}_0 \in \mathbb{R}^{n\times d}$. At each step t ($1 \le t \le n_g$) of autoregressive decoding, the model predicts a new token $\boldsymbol{x}_t$ conditioned on the input and previously generated tokens. At the following step, MHA only needs to compute the Query/Key/Value vectors[4] ($\boldsymbol{q}_t, \boldsymbol{k}_t, \boldsymbol{v}_t \in \mathbb{R}^d$) for the newly generated token $\boldsymbol{x}_t$ and appends $\boldsymbol{k}_t, \boldsymbol{v}_t$ to the KV cache: $\boldsymbol{K}_t = \boldsymbol{K}_{t-1}\|\boldsymbol{k}_t$, $\boldsymbol{V}_t = \boldsymbol{V}_{t-1}\|\boldsymbol{v}_t$. Then it performs the attention (1) between $\boldsymbol{q}_t$ and $\boldsymbol{K}_t$, $\boldsymbol{V}_t$ only.

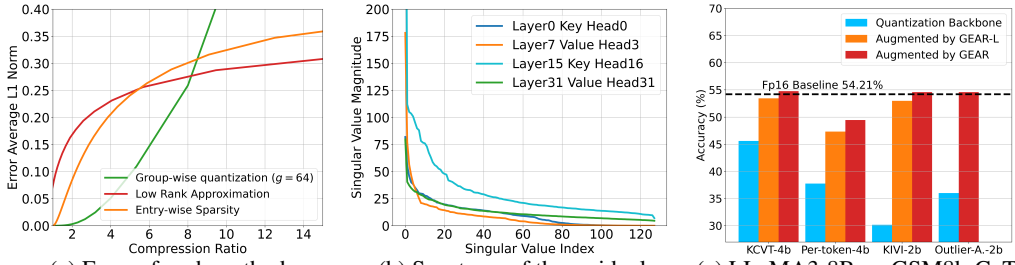
Group-wise Quantization. Group-wise quantization is widely applied to compress KV cache[23, 16, 11]. Given a tensor $\boldsymbol{X} \in \mathbb{R}^{n\times d}$ in full precision, the vanilla method groups entries per-token by placing g consecutive entries of a token into one group, e.g., the i-th group $\boldsymbol{X}_{\mathcal{G}_i}$ contains entries with indices $\mathcal{G}_i = \{(t_i, c_i), \ldots, (t_i, c_i + g)\}$ where (t_i, c_i) is the beginning index of group i and g is group size. Then, it quantizes $\boldsymbol{X}$ group-wise: $\widehat{\boldsymbol{X}} = \text{Quant}_{b,g}^{\text{(per-token)}}$ with

$$\text{Quant}_{b,g}^{\text{(per-token)}}(\boldsymbol{X})_{\mathcal{G}_i} = \lceil(\boldsymbol{X}_{\mathcal{G}_i} - \min \boldsymbol{X}_{\mathcal{G}_i})/\Delta_i\rfloor\,,\ \Delta_i = (\max \boldsymbol{X}_{\mathcal{G}_i} - \min \boldsymbol{X}_{\mathcal{G}_i})/(2^b - 1) \tag{2}$$

where b is the bit-width, $\widehat{\boldsymbol{X}}$ is the quantized tensor in b-bit precision, and $\lceil\cdot\rfloor$ is the rounding function. Δ_i and $\min \boldsymbol{X}_{\mathcal{G}_i}$ are the scaling factor and zero-point of group i. Two concurrent works (KIVI [16] and KVQuant [11]) explore the entry distribution of KV cache and show that some fixed channels exhibit very large magnitudes, and propose to quantize Key cache per-channel while quantizing Value cache per-token, achieving the start-of-the-art 2-bit compression.

Intuitively, more fine-grained grouping with smaller group size, such as $g = 64$ in KIVI [16], leads to more accurate approximation and yields better performance. However, small group size induces considerable memory overheads due to the increased number of scaling factors and zero-points stored in full precision for each group. Meanwhile, fine-grained grouping for per-channel quantization leads to maintaining a residual subset of KV tokens in full precision until they form a complete group [16]. Hence, the residual length of this full-precision parts should be set as a multiple of group size (e.g., 128 as set by KIVI), further resulting in additional considerable overheads. To leverage the SOTA quantization scheme while minimizing overheads, we choose per-channel Key

[4]For simplicity, we concatenate multi-head embeddings here.



(a) Error of each method (b) Spectrum of the residual (c) LLaMA3-8B on GSM8k-CoT

Figure 2: (2a, 2b) We randomly sample a GSM8k example and analyze its KV caches by LLaMA2-7B. (2a): the minimal approximation error of each individual technique when approximating the Value cache of the first layer; (2b): spectrum of the residual $\boldsymbol{R}_h$ decays rapidly. (2c): As an efficient error-reduction framework, GEAR is orthogonal to any off-the-shelf quantization and can augment them to achieve near-lossless accuracy.

and per-token Value quantization without fine-grained grouping as a lite quantization backbone. We refer to it as *KCVT*, a variant of KIVI with coarse-grained per-vector grouping where all Key entries of one channel forms a group of size n and all Value entries of one token forms a group of size d, significantly reducing the scaling and zero point storage overhead.

3 Method

The GEAR framework consists of three important components to decompose and compress a KV cache matrix: (i) a quantized matrix $\widehat{\boldsymbol{D}}$ to serve as a compressed backbone; (ii) a low-rank matrix $\boldsymbol{L}$ to approximate the quantization residual; (iii) a sparse matrix $\boldsymbol{S}$ to capture the individual outliers.

Outlier-aware quantization. Inspired by the recent study on weight quantization [13], we observe that the quantized backbone $\widehat{\boldsymbol{D}}$ and the sparse matrix $\boldsymbol{S}$ complement each other in the KV cache compression. Specifically, the quantization scheme can result in non-trivial quantization errors within each group due to the existence of outlier entries. Therefore, a straightforward strategy is to filter out these outlier before quantization. To align with grouping of per-channel Key and per-token Value quantization, we leverage a per-vector outlier filtering. Given an input tensor $\boldsymbol{X} = \boldsymbol{K}_t$ (or $\boldsymbol{V}_t$), we extract both $\frac{s}{2}\%$ of maximum and minimum entries of each channel (or token) vector and store them in full precision with a sparse matrix $\boldsymbol{S} = \text{Filter}_s(\boldsymbol{X})$ where

$$\text{Filter}_s(\boldsymbol{X})_{ij} = \begin{cases} \boldsymbol{X}_{ij} & \text{if } \boldsymbol{X} = \boldsymbol{K}_t \text{ and } \boldsymbol{X}_{ij} \text{ in top/bottom } \frac{s}{2}\% \text{ of the } j\text{-th channel } \boldsymbol{X}_{\cdot j}, \\ \boldsymbol{X}_{ij} & \text{if } \boldsymbol{X} = \boldsymbol{V}_t \text{ and } \boldsymbol{X}_{ij} \text{ in top/bottom } \frac{s}{2}\% \text{ of the } i\text{-th token } \boldsymbol{X}_{i\cdot}, \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

Then, we perform the quantization on the extracted matrix and obtain the quantized backbone:

$$\widehat{\boldsymbol{D}} = \text{Quant}_b^{(\text{Selected scheme})}(\boldsymbol{X} - \boldsymbol{S}). \tag{4}$$

The outlier extraction technique has been applied by Kim et al. [13] to augment training-dependent non-uniform weight quantization. In contrast to their application scenario, we explore the potential of outlier extraction techniques in conjunction with tuning-free uniform quantization for KV caches. It is important to note that, compared to weights, KV cache compression presents unique challenges because KV caches can contain more outliers, making its accurate quantization more challenging than weights [35]. Our empirical results in Section 4.3 also show that the outlier-aware quantization faces challenges in achieving ultra-low precision compression such as 2-bit on complex generative tasks. To achieve effective high-ratio compression, it is often necessary to extract a large portion of outliers stored in a sparse matrix. However, representing such a sparse matrix with two index vectors and one value vector in full precision results in considerable memory overheads. These suggest that, while using $\boldsymbol{S}$ can reduce the error, it still falls short of fully remediating the error in an efficient way.

Low-rank approximation. To reduce the approximation error more efficiently, we resort to *low-rank approximation*. As inspired by fact that various attention heads encode diverse contextual information within different channel ranges [25, 31, 38], we propose to apply *head-wise low-rank decomposition* on the residual $\boldsymbol{R} = \boldsymbol{X} - (\widehat{\boldsymbol{D}} + \boldsymbol{S}) \in \mathbb{R}^{n \times d}$. Specifically, we first reshape $\boldsymbol{R}$ along channel dimension and obtain H multi-head sub-matrices $\{\boldsymbol{R}_h = \boldsymbol{R}[:, (h-1)d_H : hd_H] \in \mathbb{R}^{n \times d_H} | 1 \le h \le H\}$ where $\boldsymbol{R}_h$ is the residual of head h. Suppose $\boldsymbol{R}_h$ has singular value decomposition as $\sum_{i=1}^{k} \sigma_i \boldsymbol{u}_i \boldsymbol{m}_i^\top$, where $\sigma_1 \ge \cdots \ge \sigma_k$ are singular values and $\boldsymbol{u}_i, \boldsymbol{m}_i$ are the corresponding singular vectors. As shown in Figure 2b, we empirically observe that the spectrum of residual matrices drops rapidly at the beginning. This suggests the existence of a coherent component within the residual. This component is represented by the top singular values/vectors, and shared among tokens, indicating the

Table 1: Results on CoT reasoning tasks, which are hard generative task. Here, *KV Size* is the average % of the remaining size of compressed KV caches with respect to the size in FP16. The best results are shown in **bold**. *N.A.* represents the extermely degenerated performance.

Model			LLaMA3-8B			LLaMA2-13B			Mistral-7B			All
Method	Bit b	Ave. KV size	GSM8k Acc	AQuA Acc	BBH Acc	GSM8k Acc	AQuA Acc	BBH Acc	GSM8k Acc	AQuA Acc	BBH Acc	Ave. Acc
FP16	16	100%	54.21	38.19	53.66	30.34	21.65	40.79	42.84	35.04	47.92	40.52
Per-token Q.$_{g=64}$	4	34.2%	37.07	39.37	46.42	20.85	18.90	34.72	31.47	29.13	28.88	31.94
KCVT Quant	4	27.1%	45.59	36.61	51.67	21.14	21.05	36.71	30.31	24.37	46.86	34.92
KIVI$_{g=64,\,n_b=64}$	4	34.2%	46.25	36.22	48.03	22.14	21.65	37.76	32.83	25.98	44.56	35.05
GEAR-L$^{(\text{KCVT})}_{r=4}$	4	29.0%	53.44	38.98	52.23	**30.25**	23.23	38.52	**43.06**	33.07	47.42	40.02
GEAR$^{(\text{KCVT})}_{s=2\%,r=4}$	4	31.0%	**54.76**	**40.55**	**52.74**	30.17	**24.05**	**40.63**	41.93	**34.57**	**47.84**	**40.80**
Per-token Q.$_{g=64}$	2	21.7%	3.56	9.84	4.72	*N.A.*	10.54	*N.A.*	*N.A.*	11.42	5.93	7.67
KIVI$_{g=64,\,n_b=64}$	2	21.7%	30.17	25.36	30.92	16.60	17.72	29.43	23.35	22.44	31.28	25.25
GEAR-L$^{(\text{KIVI})}_{r=4}$	2	23.6%	52.62	38.19	**51.44**	26.61	20.87	39.44	39.27	29.92	46.36	38.34
GEAR$^{(\text{KIVI},g=64)}_{s=2\%,r=4}$	2	27.6%	**54.59**	**38.19**	50.30	**30.27**	**23.62**	**39.67**	**43.14**	**33.96**	**48.03**	**40.20**

vector similarity. By these top singular values and vectors, we can efficiently capture and recover this coherent information, leading to an effective approximation to the quantization residual. To this end, we introduce a matrix $\boldsymbol{L} = \text{Concat}(\boldsymbol{L}_1, \ldots, \boldsymbol{L}_H)$, where $\boldsymbol{L}_h$ is a low-rank matrix:

$$\boldsymbol{L}_h = \boldsymbol{A}_h \boldsymbol{B}_h^\top = \text{SVDSolver}_r(\boldsymbol{R}_h) \tag{5}$$

$\boldsymbol{A}_h \in \mathbb{R}^{n\times r}, \boldsymbol{B}_h \in \mathbb{R}^{d_H \times r}$ and r is much smaller than n, d_H. For example, when $n = 1024$ and $d_H = 128$, $r = 4$ is sufficient to achieve near-lossless high-ratio compression. For SVDSolver$(\cdot)$, we employ an efficient *power iteration* algorithm [30]. This algorithm calculates $\boldsymbol{A}_h, \boldsymbol{B}_h$ rapidly while ensuring that $\boldsymbol{A}_h \boldsymbol{B}_h^\top$ accurately approximates the top-r singular values/vectors $\sum_{i=1}^{r} \sigma_i \boldsymbol{u}_i \boldsymbol{m}_i^\top$ (please see Appendix B for the algorithm details). In the multi-batch setting, we apply low-rank approximation to input tensors batch-wise and head-wise.

In summary, GEAR integrates three compression techniques to provide an efficient solution for minimizing the approximation error. Specifically, the quantized backbone $\widehat{\boldsymbol{D}}$ leverages the entry-wise similarity and compresses the majority of entries to the ultra-low precision. The low-rank matrix $\boldsymbol{L}_h$ capitalizes on vector-wise similarity to extract the commonly shared information within the residuals. The sparse matrix $\boldsymbol{S}$ compensates for the extraction of sparse information existing in individual outliers and compliments the quantization process. As such, GEAR effectively reduces the approximation error, achieving high-ratio KV cache compression. We recommend to use GEAR with all three components for the best performance – both near-lossless 4-bit and 2-bit performance as an alternate to SOTA methods. However, to prioritize efficiency, one can resort to a lite version of GEAR, namely *GEAR-L*, that equips only low-rank approximation to restore quantization error, costing less memory-overhead while improving accuracy significantly. Finally, we highlight that, as an efficient error-reduction framework, GEAR(-L) is *orthogonal* to any off-the-shelf quantization scheme and can augment them to achieve near-lossless accuracy as shown in Figure 2c and Section 4.

Streaming Buffer. GEAR also introduces a *streaming buffer* strategy during decoding to significantly boost its inference speed. Specifically, when serving the long-sequence generation, GEAR stores KV vectors of newly generated tokens in full precision to a small buffer $\mathcal{B}$ of size n_b (e.g., $n_b = 20$). When the buffer reaches its capacity every n_b decoding steps, GEAR conduct the compression for new tokens in $\mathcal{B}$ while the subsequent low-rank approximation is only performed on the new tokens. The concurrent work, KIVI [16], introduces a similar buffering approach to cache residual tokens until they complete a group. Hence, their residual buffer size should be set as a multiple of group size. In the case of coarse-grained grouping of KCVT, the buffer size can be set arbitrarily and we select a small size like $n_b = 20$ to enhance the inference speed while avoiding the non-trivial memory overheads. We summarize the detailed algorithm of GEAR in Algorithm 1 of Appendix A.

4 Experiments

We use GEAR as a *plug-and-play* KV cache compression for generative inference with various LLM models (including LLaMA2-7B/13B [28], Mistral-7B [12] and LLaMA3-8B [17]) on generative tasks including math reasoning (GSM8k [5] and AQuA [14]), symbolic reasoning (BigBench Hard (BBH) [24]) with CoT prompting [33], and long-context understanding (LongBench [2]).

Table 2: Results on GSM8k 5-shot and LongBench evaluation. Here, *KV Size* is the average % of the remaining size of compressed KV caches with respect to that in FP16 (i.e., the inverse of compression ratio). The best results are shown in **bold**. Results marked as † are taken from other papers.

Dataset		GSM8k 5-shot			LongBench w. LLaMA2-7B				
Method	Bit b	Ave. KV size	7B Acc	8B Acc	QMSum Rouge	SAMSum Rouge	GovReport Rouge	21 Tasks Ave. Ave. KV	21 Tasks Ave. Ave. score
FP16	16	100%	13.50	49.89	21.28	43.57	26.06	100%	26.82
Per-token Q.$_{g=64}$	4	38.2%	10.54	45.64	20.91	39.15	**28.50**	31.6%	27.31
KCVT Quant	4	27.1%	12.51	43.14	20.91	33.89	24.32	25.7%	26.06
KIVI$_{g=64,\,n_b=64}$	4	38.2%	**13.41**	48.37	20.81	40.98	26.86	31.6%	27.58
GEAR-L$^{(\text{KCVT})}_{r=4}$	4	30.4%	12.51	47.23	21.18	**41.39**	26.93	27.3%	27.65
GEAR$^{(\text{KCVT})}_{s=2\%,r=4}$	4	32.4%	13.19	**49.43**	**21.28**	41.32	26.97	29.3%	**27.80**
Per-token Q.$_{g=64}$	2	25.7%	0.08	0.83	19.78	40.31	25.50	17.5%	27.69
KIVI$_{g=32,\,n_b=128}$	2	34.9%	12.74†	42.54	20.50†	42.71†	25.72	19.7%	27.83
GEAR-L$^{(\text{KIVI},g=64)}_{r=4}$	2	27.5%	12.63	47.01	**20.69**	42.35	26.67	19.1%	**27.90**
GEAR$^{(\text{KIVI},g=64)}_{s=2\%,r=4}$	2	31.5%	**13.04**	**49.96**	20.59	**43.22**	**27.73**	23.1%	25.48

Implementation optimization and details. To minimize the overheads, we demonstrate via GPU kernel support and optimize the implementation for GEAR as follows. Firstly, we fuse the de-quantization with matrix multiplication using CUDA to improve decoding latency. Secondly, we integrate the streaming buffer for both the Key and Value such that newly generated Key/Value caches are all compressed every n_b steps. Moreover, due to streaming buffer during decoding, low-rank approximation is performed every n_b steps for only buffered tokens with ultra low rank ($r = 2$), improving compression efficiency. Thirdly, we preform the forward pass of low-rank matrices on a separate path where down projection (e.g., $\boldsymbol{q}_h^\top \boldsymbol{B}_h$) is first computed, followed by up projection (e.g., $(\boldsymbol{q}_h^\top \boldsymbol{B}_h)\boldsymbol{A}_h^\top$), reducing computational complexity of their forward pass.

We apply GEAR and baseline methods to open-source pre-trained LLMs available at *Huggingface* [34], using our inference framework written in *PyTorch* [20]. As we focus on evaluating the impact of KV Cache compression, we keep all other tensors in FP16, unless otherwise stated. We focus on ultra-low precision quantization and report the results of 4-bit and 2-bit quantization. For GEAR, we fix the sparsity ratio s at 2%, set the rank r to 4 for inputs in prefill phase, and set the rank to 2 for each group of n_b new tokens in decoding phase. We find that the efficient KCVT quantization achieves effective 4-bit compression and hence leverage it as 4-bit quantization backbone for GEAR due to its efficiency. However, in case of 2-bit compression, its performance degenerates a lot and the quantization schemes have to resort to fine-grained grouping to establish acceptable accuracy. Hence, we use KIVI as 2-bit quantization backbone for GEAR. As demonstrated by [16] that KIVI is not sensitive to group size g and residual length n_b (Table 5 in [16]), we thus select the group size as 64 and the residual length as 64 for both GEAR and KIVI in order to lower KV size overheads. The superscript in bracket shown in Table 1 and 2 identifies the backbone quantization scheme.

Baselines. We compare GEAR with the following baseline methods:

• *Per-token group-wise quantization* (used in FlexGen [23]) is a widely-adopted method that quantizes KV cache per-token with fine-grained grouping.

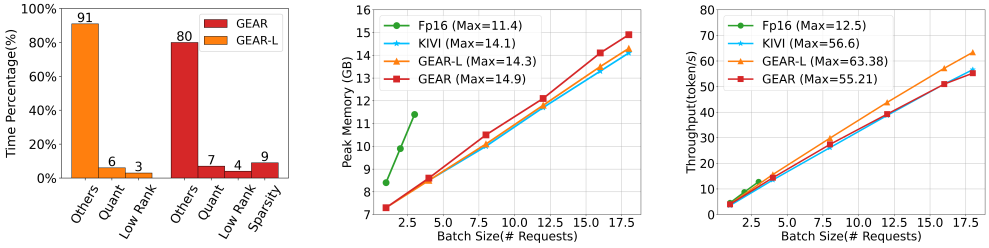
• *KIVI* [16] is a concurrent KV cache quantization method that achieves start-of-the-art 2-bit KV cache compression. This method quantizes Key cache per-channel and quantizes Value cache per-token with fine-grained grouping, and stored residual tokens of length n_b in full precision.

• *KCVT quantization* is a variant of KIVI that quantize Key cache per-channel and Value cache per-token without fine-grained grouping. This is a per-vector quantization that induces lower overheads.

• *H_2O* [40] is a recent token dropping method evicting unimportant tokens with lower accumulated attention scores, which we compare with in Table 10.

4.1 Main Results

Generative performance on hard CoT reasoning tasks. We compare different methods with LLaMA3-8B, LLaMA2-13B, and Mistral-7B on three challenging CoT generative tasks: GSM8k, AQuA, and BBH with 8-shot CoT demonstrations. GSM8k [5] and AQuA [14] are widely used



(a) Time breakdown analysis (b) Peak memory comparison (c) Throughput comparison

Figure 3: (3a) wall-clock time percentage of each component in GEAR. (3b): GEAR reduces the peak memory and enable much larger batch size than FP16. (3c): GEAR improve throughput significantly over FP16.

math reasoning datasets that test models' ability of arithmetic reasoning. BBH [24] is a suite of language and symbolic reasoning problems consisting of 6.5k problems within 23 subsets. Given the complexity of these tasks, we use the chain-of-thought prompts created by [9] to improve reasoning, which contains 8-shot demonstrations of multi-step reasoning. With the CoT demonstrations, we have the average prefill length of GSM8k, AQuA, and BBH as 900, 1304, 1021 respectively (see Appendix E). We then prompt model to generate 256 tokens and extract answers from them. Therefore, our experiments involve long-sequence generation. Notably, as mentioned in Section 1, CoT prompts often contains densely correlated information across multiple reasoning steps and models need to pay close attention across steps to derive answers correctly. Hence, a relatively small compression error can be magnified along generation steps, resulting in significant deviation in model generations.

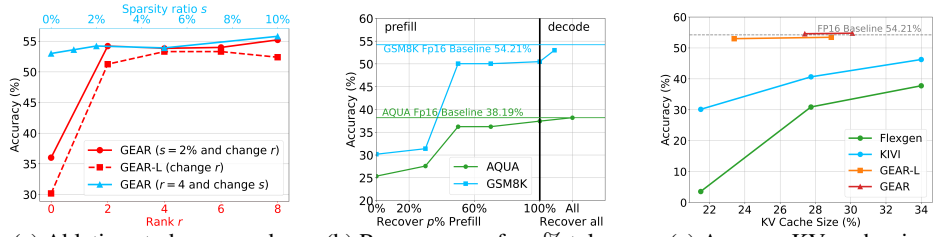
Table 1 presents experimental results on these hard CoT reasoning tasks. We see that GEAR and GEAR-L achieves better or on par performance compared with baseline methods on all datasets and all models in both 4-bit and 2-bit compression. For example, in the case of 2-bit compression, GEAR achieves 47.83% average accuracy on LLaMA3-8B across three datasets, which is near-lossless compared to FP16 baseline (48.69%) and significantly outperforms the best-performing baseline (28.82%, KIVI). Notably, GEAR-L also establish remarkable performance – near-lossless 4-bit compression and superior 2-bit performance compared to baselines, while demonstrating lower KV size and higher inference efficiency. Meanwhile, as shown in Table 1 and Figrue 2c, regardless quantization backbone we choose, our method can always improve upon them by integrating the error-reduction techniques, showcasing its generalization ability as an efficient error-reduction framework. Thus, we highlight that **GEAR(-L) is orthogonal to any off-the-shelf quantization scheme and can augment them in a plug-and-play manner to achieve near-lossless accuracy at minimal memory overheads**.

Generative performance on relatively easy tasks. We also compare different methods on relatively easy tasks without CoT reasoning. Specifically, we evaluate the performance with LLaMA2-7B on LongBench [2], which is a suit of 21 long-context understanding tasks including question answering, summarization, code completion, etc. (please see Appendix E for task metrics and dataset statistics). The average input length of LongBench is 3642. We follow the evaluation method in [2], apply their evaluation metrics and report the average score across all 21 tasks. Besides, we also follow [16] and compare the performance using LLaMA2-7B and LLaMA3-8B on GSM8k with standard 5-shot prompts. Such 5-shot demonstrations consists of 5 sampled questions and their one-step (or two-step) answers and do not involve complex CoT. Models are prompted to answer the question without multi-step reasoning, which is simpler than the setting of 8-shot CoT prompting.

Table 2 present the experimental results on these relatively simpler tasks. We see that quantization methods can already achieve near-lossless 4-bit/2-bit compression on these tasks, showcasing their effectiveness on simpler tasks. For example, for 2-bit compression, per-token group-wise quantization and KIVI both yield around 27.7% average scores across 21 tasks of LongBench. Moreover, KIVI establish near-lossless 2-bit performance on GSM8k with 5-shot standard examples for both LLaMA2-7B and LLaMA3-8B models. After incorporating error-reduction techniques, GEAR and GEAR-L can achieve better or on par performance compared to baseline quantization methods. For example, GEAR achieves 49.96% accuracy on GSM8k (5-shot) when compressing KV caches of LLaMA3-8B to 2-bit, which is 7.42% higher then KIVI.

4.2 Inference Efficiency Comparison

In this section, we evaluate wall-clock time, memory, and throughput of GEAR on a single NVIDIA V100 GPU (16GB). Specifically, we set the input and generation length as 1000 and 500 respectively, and evaluate with LLaMA2-7B. We increase the batch size until out of memory and report the peak memory/throughput between FP16 KV caches and 2-bit quantization: KIVI, GEAR, and GEAR-L.



(a) Ablation study on s and r (b) Recover error for $p\%$ tokens (c) Acc. v.s. KV cache size

Figure 4: Analysis and ablation study with LLaMA3-8B on GSM8k-CoT under 2-bit compression.

We use the same hyperparameters as in Section 4.1. Here, to maximize the batch size for all methods, we compress model weights to 8-bit, using bitsandbytes from *Huggingface Transformers* [34].

In this inference setting, we first provide a time breakdown analysis for GEAR that compares total computational time of different components during generative inference: (i) *quantization-related time* that consists of total quantization and dequantization time after equipping our CUDA kernel; (ii) *low-rank time* that includes total time of SVD approximation by Algorithm 2 and forward pass of low-rank matrices; (iii) *sparsity time* that contains total computational time of outlier extraction and matrix multiplication involving $\boldsymbol{S}$ during forward pass; (iv) *other time* that is primarily about model forward pass and obtained by subtracting total wall-clock time with time summation of aforementioned three items. We use the maximum batch size here (which is 18) and report the average over three trials. Figure 3a presents the time percentage of each component in GEAR and GEAR-L during generative inference. It suggests that, while yielding significant performance gains, low-rank and sparse components are lightweight and do not induce unacceptable overheads. The primary complexity still stems from model forward pass. The additional latency by low-rank and sparsity components can be negligible due to our optimized implementation and inference techniques.

Figure 3b present the peak memory comparison across different batch sizes under the same inference setting. We see that, given the same batch size, GEAR significantly reduces the peak memory compared to FP16 baseline, increasing the maximum severing number (i.e., batch size) from 3 to 18. Moreover, Figure 3c shows the throughput comparison across various batch sizes. The results demonstrate that, compared to FP16 baseline, our method significantly improves the throughput by up to $5.07\times$. Meanwhile, GEAR-L achieves slightly better throughput than KIVI due to our improved streaming strategy. We persent the detailed results of Figure 3b and 3c in Appendix F.

4.3 Analysis and Ablation Study

Ablation study on sparsity ratio s and rank r. We study the sensitivity of GEAR to the sparsity ratio s and rank r. Figure 4a shows 2-bit quantization of GEAR and GEAR-L using LLaMA3-8B on GSM8k (w. CoT) when varying s or r. We see that GEAR does not require abundant sparse either low-rank components – a small sparse ratio ($s = 2\%$ for GEAR) and a small rank ($r = 4$ for GEAR and GEAR-L) is adequate to achieve near-lossless 2-bit compression, demonstrating high efficiency of our method. Further increasing s or r may improve the accuracy but not significantly, which however results in additional memory overheads. More importantly, discarding low-rank component can significantly degenerate the performance of GEAR and GEAR-L, highlighting its vital role in error reduction. On the other hand, discarding sparse matrices can hurt performance but not significantly because the incoherent error from outlier entries can also be partially remedied by entry grouping of quantization. Thus, we highlight GEAR-L for those prioritizing efficiency.

Applying error reduction to different amounts of tokens. To further demonstrate the effectiveness of error reduction, we study the performance variation of GEAR-L when applying low-rank approximation to varying number of tokens with LLaMA3-8B on GSM8k and AQuA (w. CoT). Specifically, we split tokens into (i) input tokens in prefill phrase and (ii) generated tokens in decoding phrase. By default, we recover quantization errors for all of them. Alternatively, we can take $p\%$ most recent prefill tokens and only apply low-rank approximation to them. Figure 4b presents the performance of GEAR-L when changing p. We see that the performance of GEAR-L degenerates when decreasing the number of token applied error reduction, validating the effectiveness of error-reduction technique.

Different compression ratios. Figure 4c compares the performance of various methods on GSM8k (w. CoT) when compressing KV caches of LLaMA3-8B to different remaining size. We see that GEAR and GEAR-L consistently outperform other quantization baseline methods, achieving near-lossless accuracy across various compression ratios and showcasing their effectiveness as an efficient error-reduction framework for KV cache quantization.

5 Conclusions

In this paper, we present GEAR, an efficient error-reduction framework that can augment any off-the-shelf KV cache quantization scheme with two lightweight error reduction techniques in a plug-and-play manner to achieve near-lossless accuracy at high-ratio compression. GEAR demonstrates the SOTA performance on complex generative tasks involving reasoning, achieving an average accuracy improvement of 14.95% at 2-bit KV quantization compared to the alternatives.

References

[1] R. Y. Aminabadi, S. Rajbhandari, M. Zhang, A. A. Awan, C. Li, D. Li, E. Zheng, J. Rasley, S. Smith, O. Ruwase, and Y. He. Deepspeed inference: Enabling efficient inference of transformer models at unprecedented scale, 2022.

[2] Y. Bai, X. Lv, J. Zhang, H. Lyu, J. Tang, Z. Huang, Z. Du, X. Liu, A. Zeng, L. Hou, Y. Dong, J. Tang, and J. Li. Longbench: A bilingual, multitask benchmark for long context understanding, 2023.

[3] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[4] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020. URL `https://arxiv.org/abs/2005.14165`.

[5] K. Cobbe, V. Kosaraju, M. Bavarian, M. Chen, H. Jun, L. Kaiser, M. Plappert, J. Tworek, J. Hilton, R. Nakano, C. Hesse, and J. Schulman. Training verifiers to solve math word problems, 2021.

[6] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.

[7] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer. Llm. int8 (): 8-bit matrix multiplication for transformers at scale. *arXiv preprint arXiv:2208.07339*, 2022.

[8] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers, 2023.

[9] Y. Fu, L. Ou, M. Chen, Y. Wan, H. Peng, and T. Khot. Chain-of-thought hub: A continuous effort to measure large language models' reasoning performance, 2023.

[10] S. Ge, Y. Zhang, L. Liu, M. Zhang, J. Han, and J. Gao. Model tells you what to discard: Adaptive kv cache compression for llms, 2023.

[11] C. Hooper, S. Kim, H. Mohammadzadeh, M. W. Mahoney, Y. S. Shao, K. Keutzer, and A. Gholami. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *arXiv preprint arXiv:2401.18079*, 2024.

[12] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed. Mistral 7b, 2023.

[13] S. Kim, C. Hooper, A. Gholami, Z. Dong, X. Li, S. Shen, M. W. Mahoney, and K. Keutzer. Squeezellm: Dense-and-sparse quantization, 2023.

[14] W. Ling, D. Yogatama, C. Dyer, and P. Blunsom. Program induction by rationale generation: Learning to solve and explain algebraic word problems. *ACL*, 2017.

[15] Z. Liu, A. Desai, F. Liao, W. Wang, V. Xie, Z. Xu, A. Kyrillidis, and A. Shrivastava. Scissorhands: Exploiting the persistence of importance hypothesis for LLM KV cache compression at test time. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL `https://openreview.net/forum?id=JZfg6wGi6g`.

[16] Z. Liu, J. Yuan, H. Jin, S. Zhong, Z. Xu, V. Braverman, B. Chen, and X. Hu. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. *arXiv preprint arXiv:2402.02750*, 2024.

[17] Meta. Introducing meta llama 3: The most capable openly available llm to date, 2024. URL `https://ai.meta.com/blog/meta-llama-3/`.

[18] I. Mirzadeh, K. Alizadeh, S. Mehta, C. C. Del Mundo, O. Tuzel, G. Samei, M. Rastegari, and M. Farajtabar. Relu strikes back: Exploiting activation sparsity in large language models. *arXiv preprint arXiv:2310.04564*, 2023.

[19] OpenAI. Gpt-4 technical report, 2023.

[20] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 8024–8035, 2019.

[21] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, A. Levskaya, J. Heek, K. Xiao, S. Agrawal, and J. Dean. Efficiently scaling transformer inference, 2022.

[22] L. Ribar, I. Chelombiev, L. Hudlass-Galley, C. Blake, C. Luschi, and D. Orr. Sparq attention: Bandwidth-efficient llm inference, 2023.

[23] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, D. Y. Fu, Z. Xie, B. Chen, C. Barrett, J. E. Gonzalez, P. Liang, C. Ré, I. Stoica, and C. Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu, 2023.

[24] M. Suzgun, N. Scales, N. Schärli, S. Gehrmann, Y. Tay, H. W. Chung, A. Chowdhery, Q. V. Le, E. H. Chi, D. Zhou, and J. Wei. Challenging big-bench tasks and whether chain-of-thought can solve them, 2022.

[25] I. Tenney, D. Das, and E. Pavlick. Bert rediscovers the classical nlp pipeline, 2019.

[26] R. Thoppilan, D. D. Freitas, J. Hall, N. Shazeer, A. Kulshreshtha, H.-T. Cheng, A. Jin, T. Bos, L. Baker, Y. Du, Y. Li, H. Lee, H. S. Zheng, A. Ghafouri, M. Menegali, Y. Huang, M. Krikun, D. Lepikhin, J. Qin, D. Chen, Y. Xu, Z. Chen, A. Roberts, M. Bosma, V. Zhao, Y. Zhou, C.-C. Chang, I. Krivokon, W. Rusch, M. Pickett, P. Srinivasan, L. Man, K. Meier-Hellstern, M. R. Morris, T. Doshi, R. D. Santos, T. Duke, J. Soraker, B. Zevenbergen, V. Prabhakaran, M. Diaz, B. Hutchinson, K. Olson, A. Molina, E. Hoffman-John, J. Lee, L. Aroyo, R. Rajakumar, A. Butryna, M. Lamm, V. Kuzmina, J. Fenton, A. Cohen, R. Bernstein, R. Kurzweil, B. Aguera-Arcas, C. Cui, M. Croak, E. Chi, and Q. Le. Lamda: Language models for dialog applications, 2022.

[27] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample. Llama: Open and efficient foundation language models, 2023.

[28] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023.

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

[30] T. Vogels, S. P. Karimireddy, and M. Jaggi. Powersgd: Practical low-rank gradient compression for distributed optimization. *CoRR*, abs/1905.13727, 2019. URL http://arxiv.org/abs/1905.13727.

[31] E. Voita, D. Talbot, F. Moiseev, R. Sennrich, and I. Titov. Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned, July 2019. URL https://aclanthology.org/P19-1580.

[32] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler, E. H. Chi, T. Hashimoto, O. Vinyals, P. Liang, J. Dean, and W. Fedus. Emergent abilities of large language models. *Transactions on Machine Learning Research*, 2022. ISSN 2835-8856. URL https://openreview.net/forum?id=yzkSU5zdwD. Survey Certification.

[33] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023.

[34] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, et al. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

[35] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han. Smoothquant: Accurate and efficient post-training quantization for large language models, 2023.

[36] A. Yuan, A. Coenen, E. Reif, and D. Ippolito. Wordcraft: Story writing with large language models. In *27th International Conference on Intelligent User Interfaces*, IUI '22, page 841–852, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391443. doi: 10.1145/3490099.3511105. URL https://doi.org/10.1145/3490099.3511105.

[37] O. Zafrir, G. Boudoukh, P. Izsak, and M. Wasserblat. Q8bert: Quantized 8bit bert. In *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing - NeurIPS Edition (EMC2-NIPS)*. IEEE, Dec. 2019. doi: 10.1109/emc2-nips53020.2019.00016. URL http://dx.doi.org/10.1109/EMC2-NIPS53020.2019.00016.

[38] Q. Zhang, C. Singh, L. Liu, X. Liu, B. Yu, J. Gao, and T. Zhao. Tell your model where to attend: Post-hoc attention steering for LLMs. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=xZDW0oejD.

[39] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin, T. Mihaylov, M. Ott, S. Shleifer, K. Shuster, D. Simig, P. S. Koura, A. Sridhar, T. Wang, and L. Zettlemoyer. Opt: Open pre-trained transformer language models, 2022.

[40] Z. Zhang, Y. Sheng, T. Zhou, T. Chen, L. Zheng, R. Cai, Z. Song, Y. Tian, C. Ré, C. Barrett, Z. Wang, and B. Chen. H_2o: Heavy-hitter oracle for efficient generative inference of large language models, 2023.

[41] Y. Zhao, C.-Y. Lin, K. Zhu, Z. Ye, L. Chen, S. Zheng, L. Ceze, A. Krishnamurthy, T. Chen, and B. Kasikci. Atom: Low-bit quantization for efficient and accurate llm serving, 2023.

A Detailed Algorithm of GEAR

Algorithm 1 GEAR

Input: The initial $\{\boldsymbol{K}_0, \boldsymbol{V}_0\}$ of each layer, the sparsity ratio s, the bit-width b, the rank for prefill token r_p, the rank for generated token r_g, the buffer $\mathcal{B}$.
(Prefill Phase):
for $\boldsymbol{X} \in \{\boldsymbol{K}_0, \boldsymbol{V}_0\}$ **do**
  Compute $\boldsymbol{S} = \text{Filter}_s(\boldsymbol{X})$;
  Compute $\widehat{\boldsymbol{D}} = \text{Quant}_b(\boldsymbol{X} - \boldsymbol{S})$;
  Compute $\boldsymbol{R} = \boldsymbol{X} - \widehat{\boldsymbol{D}} - \boldsymbol{S}$;
  for $h = 1, \ldots, H$ **do**
    Compute $\boldsymbol{L}_h = \text{SVDSolver}_{r_p}(\boldsymbol{R}_h)$;
  end for
  Concatenate $\boldsymbol{L} = \text{Concat}(\boldsymbol{L}_1, \ldots, \boldsymbol{L}_H)$;
  Replace $\boldsymbol{X}$ with $\widehat{\boldsymbol{D}} + \boldsymbol{L} + \boldsymbol{S}$.
end for
(Decoding Phase):
for $t = 1, \ldots, n_g$ **do**
  if $t \mod n_b = 0$ **then**
    for $\boldsymbol{X} \in \{\boldsymbol{K}_{\mathcal{B}}, \boldsymbol{V}_{\mathcal{B}}\}$ **do**
      Compute $\boldsymbol{S} = \text{Filter}_s(\boldsymbol{X})$;
      Compute $\widehat{\boldsymbol{D}} = \text{Quant}_b(\boldsymbol{X} - \boldsymbol{S})$;
      for $h = 1, \ldots, H$ **do**
        Compute $\boldsymbol{L}_h = \text{SVDSolver}_{r_g}(\boldsymbol{X} - \widehat{\boldsymbol{D}} - \boldsymbol{S})$;
      end for
      Concatenate $\boldsymbol{L} = \text{Concat}(\boldsymbol{L}_1, \ldots, \boldsymbol{L}_H)$;
      Replace $\boldsymbol{X}$ with $\widehat{\boldsymbol{D}} + \boldsymbol{L} + \boldsymbol{S}$.
    end for
    Append $\boldsymbol{K}_t = \boldsymbol{K}_{t-n_b} \| \boldsymbol{K}_{\mathcal{B}}, \boldsymbol{V}_t = \boldsymbol{V}_{t-n_b} \| \boldsymbol{V}_{\mathcal{B}}$.
  else
    Generate new token $\boldsymbol{x}_t$ and Push $\boldsymbol{k}_t$ to $\boldsymbol{K}_{\mathcal{B}}$ and Push $\boldsymbol{v}_t$ to $\boldsymbol{V}_{\mathcal{B}}$.
  end if
end for

B Power Iteration Algorithm as SVDSolver

The power iteration algorithm is presented in Algorithm 2.

Algorithm 2 Low rank approximation of the error tensor

Require: Input matrix $\boldsymbol{X} \in \mathbb{R}^{n \times d}$ loop iteration L, low rank fraction r.
Output: $\boldsymbol{A} \in \mathbb{R}^{n \times r}, \boldsymbol{B} \in \mathbb{R}^{d \times r}, \boldsymbol{A}\boldsymbol{B}^\top = \boldsymbol{L}$
`random_initialize`$(\boldsymbol{A})$,
`random_initialize`$(\boldsymbol{B})$
while $l < L$ **do**
  if $l == L - 1$ **then**
    $\boldsymbol{B} \leftarrow$ `QRdecompostion`$(\boldsymbol{B})$
  end if
  $\boldsymbol{A} = \boldsymbol{X}\boldsymbol{B}$
  if $l == L - 1$ **then**
    $\boldsymbol{A} \leftarrow$ `QRdecompostion`$(\boldsymbol{A})$
  end if
  $\boldsymbol{B} = \boldsymbol{X}^T\boldsymbol{A}$
  $l \leftarrow l + 1$
end while

C More Analysis

The role of low-rank approximation. We compare GEAR with outlier-aware quantization to highlight the importance of low-rank approximation. Specifically, we apply the same evaluation settings as Section 4.1. Table 8 in Appendix G presents the 2-bit performance of outlier-aware KIVI quantization. The results suggest that employing outlier extraction alone for quantization can improve the performance but cannot achieve near-lossless 2-bit performance that GEAR does. Outlier-aware quantization still faces challenges in achieving high-ratio compression. In contrast, low-rank approximation plays a pivotal role in fully remedy approximation errors and achieving near-lossless high-ratio compression.

D More Discussion on Related Works

LLM weights compression. LLM weight compression can significantly reduce the memory footprint and data transfer cost. GPTQ [8] accelerated the optimal brain quantization for LLM weights by orders of magnitude. SqueezeLLM [13] successfully compressed the model weights to 3 bits by extracting the outlier values and quantize the remaining values according to hessian matrix within 10% perplexity increases. These algorithms are effective and could compress weights to 2 or 3 bits with acceptable loss of accuracy. However, these methods often require significant latency overhead and gradient information to work. Thus their are not fit for KV cache compression since KV cache does not have any trainable parameter and changes every generation stage, requiring efficient light-weight method for online compression.

LLM KV cache compression. Activation and KV cache compression are harder than weight compression since they are more sensitive and related to model inputs. SmoothQuant [35] achieved 8-bit compression both for activation (KV caches included) and weights by adjusting the scaling factors to reduce outlier error and demonstrates near lossless performance on simple generative tasks. Atom [41] successfully compressed KV Cache to 4 bits on simple generative tasks within 5% performance degradation by combining 4-bit and 8-bit channel-wise quantization. Another line of work explored KV pruning via token dropping based on attention score analysis. In specific, H_2O [40] and FastGen [10] proposed to prune KV via dropping tokens based on attention score to decrease the KV cache size. SparQ [22] not only dropped tokens according to attention score sparsity but also incorporated the error of the pruned value cache. These pruning and quantization algorithms often work well on summarizing tasks and zero-shot inference. However, for fine-tuned models, CoT inference, and generative reasoning datasets, attention scores are denser and each token contains important information that can not be ignored. Moreover, token dropping needs to weigh each token based on attention score, which makes these methods hard to deploy with FlashAttention [6]. Additionally, recent works have demonstrated the attention sparsity to be a function of the non-linearity choice of the model [18], showing its vulnerability as a metric for KV compression.

E Dataset Statistics

Here, we show the statistics of all datasets including input length in prefill phrase, generation length and the number of evaluation examples.

Table 3: Statistics of GSM8k, AQuA and BBH.

	# Evaluation Example	Prefill Lenght	Generation Length
GSM8k with 8-shot CoT	1319	900	256
AQuA with 8-shot CoT	254	1304	196
BBH with 3-shot CoT	6511	1021	196
GSM8k with 5-shot examples	1319	672	96

Table 4: Statistics of LongBench.

	# Evaluation Example	Prefill Lenght	Generation Length
LongBench (Ave.)	4750	3642	256

Table 5: An overview of the dataset statistics in LongBench from [2].

Dataset	ID	Source	Avg len	Metric	Language	#data
Single-Document QA						
NarrativeQA	$1-1$	Literature, Film	18,409	F1	English	200
Qasper	$1-2$	Science	3,619	F1	English	200
MultiFieldQA-en	$1-3$	Multi-field	4,559	F1	English	150
MultiFieldQA-zh	$1-4$	Multi-field	6,701	F1	Chinese	200
Multi-Document QA						
HotpotQA	$2-1$	Wikipedia	9,151	F1	English	200
2WikiMultihopQA	$2-2$	Wikipedia	4,887	F1	English	200
MuSiQue	$2-3$	Wikipedia	11,214	F1	English	200
DuReader	$2-4$	Baidu Search	15,768	Rouge-L	Chinese	200
Summarization						
GovReport	$3-1$	Government report	8,734	Rouge-L	English	200
QMSum	$3-2$	Meeting	10,614	Rouge-L	English	200
MultiNews	$3-3$	News	2,113	Rouge-L	English	200
VCSUM	$3-4$	Meeting	15,380	Rouge-L	Chinese	200
Few-shot Learning						
TREC	$4-1$	Web question	5,177	Accuracy (CLS)	English	200
TriviaQA	$4-2$	Wikipedia, Web	8,209	F1	English	200
SAMSum	$4-3$	Dialogue	6,258	Rouge-L	English	200
LSHT	$4-4$	News	22,337	Accuracy (CLS)	Chinese	200
Synthetic Task						
PassageCount	$5-1$	Wikipedia	11,141	Accuracy (EM)	English	200
PassageRetrieval-en	$5-2$	Wikipedia	9,289	Accuracy (EM)	English	200
PassageRetrieval-zh	$5-3$	C4 Dataset	6,745	Accuracy (EM)	Chinese	200
Code Completion						
LCC	$6-1$	Github	1,235	Edit Sim	Python/C#/Java	500
RepoBench-P	$6-2$	Github repository	4,206	Edit Sim	Python/Java	500

F More Inference Analysis Comparison

F.1 Detailed results on a single V100 GPU

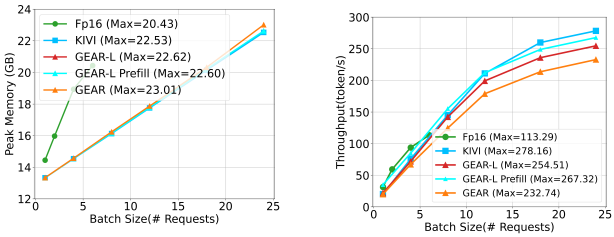
Table 6 shows detailed results of inference efficiency comparison in Section 4.2, which is on a single NVIDIA V100. Also, to measure the peak memory save-up, we measure the memory consumption under the same batch size for both GEAR and FP16 KV cache baseline, which is 18 (the maximum batch size of GEAR on V100 GPU). Then, we apply the same inference setting and batch size for FP16 KV cache baseline and test its corresponding memory consumption on a GPU with larger GPU memory that accommodate more batches. The results shows that GEAR can reduce the memory up to $2.39\times$ compared to FP16 KV cache baseline.

F.2 Inference Efficiency Comparison on a RTX Titan GPU

To futher evaluate the thoughput and memory usage of GEAR, we only apply GEAR-L,GEAR-L Prefill and GEAR on a RTX Titan GPU with 24GB memory. We choose LLaMA2-7b as our base model. GEAR-L Prefill is an lite version of GEAR-L that only apply error reduction algorithm to prefill tokens. In Section 4.3, we discuss the accuracy improved by GEAR-L Prefill compared with KIVI. Here we present the Peak Memory and throughputs comparison in Figrue 5. With larger GPU memory, GEAR-L Prefill, GEAR-L and GEAR add acceptable latency and achieves $2.10\times$ throughput improvement compared to Fp16 baseline.

Table 6: Detailed results in Section 4.2 using a single NIVIDA V100 GPU.

Method	Batch Size	Time (s)	Peak Memory (GB)	Throughputs (token/s)
FP16	1	117	8.44	4.27
	2	118	9.94	8.47
	3 (max)	120	11.44	12.5
KIVI-2bit	1	142	7.28	3.52
	4	148	8.49	13.51
	8	153	10.10	26.14
	12	155	11.71	38.71
	16	157	13.32	50.96
	18 (max)	159	14.11	56.6
GEARL-2bit	1	122	7.28	4.1
	4	128	8.53	15.63
	8	134	10.13	29.85
	12	137	11.76	43.8
	16	140	13.37	57.14
	18 (max)	142	14.16	63.38
GEARL-2bit	1	126	7.31	3.97
	4	139	8.64	14.38
	8	146	10.53	27.4
	12	153	12.06	39.22
	16	157	14.07	50.95
	18 (max)	163	14.63	55.21



(a) Memory Usage Comparison (b) Throughput Comparison

Figure 5: Peak memory and throughput comparison with LLaMA2-7b on an RTX Titan 24GB GPU.

F.3 KV Cache Component

Here we discuss the components of KV Cache. Every quantization backbone at least contains quantized integer and scale&zero point (we refer this as SZ FP16 in Figure 6). The size of former one is decided by quantization bit-width and the latter one is decided by group number of quantization algorithm. Another component is from the streaming buffer of FP16 residual tokens. When GEAR or GEAR-L combine with KCVT quantization, buffer size can be small. When combining with KIVI, buffer size should be larger than group size. GEAR and GEAR-L also have overheads stemming from sparsity and low rank components. From Figure 6, we can tell that, KCVT induces small streaming buffer overheads due to its large group size. In contrast, due to small group size of KIVI, it induces larger residual overheads and memory consumption from scaling factors and zero-points.

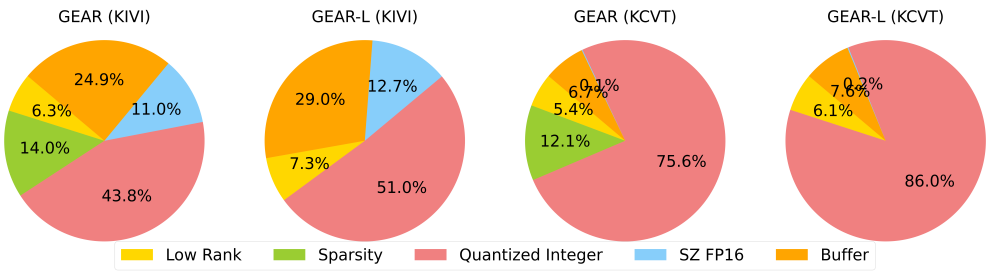


Figure 6: KV Cache memory distribution for Mistral-7B on GSM8K-CoT task

F.4 Comparison on Maximum Sequence Length

In this section, we compare the maximum sequence length for different methods under the same inference setting as in Section 4.2. We use a LLaMA2-7B, set the batch size as 1, and test the maximum sequence length n for different methods. Similarly, we compress model weights to 8-bit as in Section 4.2 and apply FlashAttention to save the memory usage and allow longer sequence length. The hyperparameters are the same as Section 4.2. Table 7 presents the maximum length for FP16 and GEAR and we can see that GEAR can increase the maximum sequence length by around 2k, making previously impossible long-sequence generation feasible.

Table 7: Maximum sequence length comparison.

Method	**Bit** b	**Max Length**
FP16 KV Cache	16	5319
$\text{GEAR}^{(\text{KIVI})}_{s=2\%,r=4}$	2	7291

G Comparison between GEAR and Outlier-Aware Quantization

In this section, we present the comparison between GEAR and outlier-aware quantization to further demonstrate the importance of low-rank approximation. Specifically, we apply the same evaluation settings as Section 4.1. Table 8 present the results.

Table 8: Comparison of GEAR with outlier-aware quantization on CoT reasoning tasks.

Model			**LLaMA3-8B**			**LLaMA2-13B**			**Mistral-7B**		
Method	**Bit** b	**KV size**	**GSM8k** Acc	**AQuA** Acc	**BBH** Acc	**GSM8k** Acc	**AQuA** Acc	**BBH** Acc	**GSM8k** Acc	**AQuA** Acc	**BBH** Acc
FP16	16	100%	54.21	38.19	53.66	30.34	21.65	40.79	42.84	35.04	47.92
$\text{KIVI}_{g\,=\,64,\,n_b\,=\,64}$	2	21.5%	30.17	25.36	30.92	16.60	17.72	29.43	23.35	22.44	31.28
$\text{Outlier-A.}^{(\text{KIVI})}_{s=2\%}$	2	24.5%	36.01	36.22	36.59	18.19	18.90	33.21	37.64	22.44	36.29
$\textbf{GEAR-L}^{(\text{KIVI})}_{r=4}$	2	23.4%	52.99	38.19	**51.44**	26.61	20.87	39.44	39.27	29.92	46.36
$\textbf{GEAR}^{(\text{KIVI},g=64)}_{s=2\%,r=4}$	2	27.4%	**54.59**	**38.19**	50.30	**30.27**	**23.62**	**39.67**	**43.14**	**33.96**	**48.03**

H KV Cache Average Size for Different Datasets

Table 9 presents the detailed KV cache size comparison across different methods, models and datasets as shown in Table 1.

Table 9: Average KV Cache size for different datasets and differet models.

Model			**LLaMA3-8B KV Cache**			**LLaMA2-13B KV Cache**			**Mistral-7B KV Cache**		
Method	**Bit** b	**Ave.** KV	**GSM8k**	**AQuA**	**BBH**	**GSM8k**	**AQuA**	**BBH**	**GSM8k**	**AQuA**	**BBH**
FP16	16	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
Per-token $\text{Q.}_{g\,=\,64}$	4	34.2%	35.2%	33.0%	34.4%	35.2%	33.0%	34.4%	35.2%	33.0%	34.4%
KCVT Quant	4	27.1%	26.7%	27.2%	27.2%	27.5%	26.7%	27.2%	27.5%	26.7%	27.2%
$\text{KIVI}_{g\,=\,64,\,n_b\,=\,64}$	4	34.2%	35.2%	33.0%	34.4%	35.2%	33.0%	34.4%	35.2%	33.0%	34.4%
$\textbf{GEAR-L}^{(\text{KCVT})}_{r=4}$	4	29.0%	29.7%	28.9%	29.4%	29.3%	28.4%	29.0%	29.3%	28.5%	29.0%
$\textbf{GEAR}^{(\text{KCVT})}_{s=2\%,\rho=2\%}$	4	31.0%	31.7%	30.9%	31.4%	31.3%	30.4%	31.0%	31.3%	30.5%	31.0%
Per-token $\text{Q.}_{g\,=\,64}$	2	21.7%	22.7%	20.5%	21.9%	22.7%	20.5%	21.9%	22.7%	20.5%	21.9%
$\text{KIVI}_{g\,=\,64,\,n_b\,=\,64}$	2	21.7%	22.7%	20.5%	21.9%	22.7%	20.5%	21.9%	22.7%	20.5%	21.9%
$\textbf{GEAR-L}^{(\text{KIVI})}_{r=4}$	2	23.6%	25.0%	22.7%	24.1%	24.5%	22.2%	23.7%	24.5%	22.2%	23.7%
$\textbf{GEAR}^{(\text{KIVI})}_{s=2\%,r=4}$	2	27.6%	29.0%	26.7%	28.1%	28.5%	26.2%	27.7%	28.5%	26.2%	27.7%

I Comparison with token dropping.

We evaluate the performance of H_2O [40] for reducing KV cache size on GSM8k with LLaMA2-7B. Table 10 presents its accuracy when dropping 50% tokens, which suggests H_20 cannot effectively preserve the performance nor achieve high compression ratio. For complex tasks involving reasoning or long-sequence generation (such as GSM8k), models need to closely attend to most contextual information to generate correct answers. Token dropping methods, however, can make some information directly invisible, resulting in deviation in generation and degradation of performance.

Table 10: Accuracy of H_2O on GSM8k with LLaMA2-7B.

Method	**Bit** b	**KV size**	**CoT Acc.**
FP16	16	100%	16.33
H_2O	16	50%	6.82
$\text{GEAR}^{(\text{KCVT})}_{s=2\%,r=4}$	4	32.4%	16.14

J Discussion on the Prompts

Question: If there are 3 cars in the parking lot and 2 more cars arrive, how many cars are in the parking lot?
Let's think step by step
There are originally 3 cars. 2 more cars arrive. 3 + 2 = 5. The answer is 5.

Figure 7: Example of GSM8k-CoT prompt. The Red, Green, and Blue colored portions correspond to the example question, a common preceding prompt, and the example answer prompt, respectively. Here, we use the common prompt to improve the reasoning of the LLM.
For the GSM8k dataset, there is a fixed prompt for all evaluations. The prompt contains 8 examples with clear guidance step by step. For the MMLU and BBH dataset, there are individual prompts for each sub dataset. Figure 7 shows one of the example in GSM8K dataset.