CSR:Achieving 1 Bit Key-Value Cache via Sparse Representation

Hongxuan Zhang[1, 2*†], **Yao Zhao**[2†], **Jiaqi Zheng**[1], **Chenyi Zhuang**[2], **Jinjie Gu**[2], **Guihai Chen**[1]
[1]Nanjing University
[2]Ant Group
x_zhang@smail.nju.edu.cn, nanxiao.zy@antgroup.com, jzheng@nju.edu.cn,
{chenyi.zcy, jinjie.gujj}@antgroup.com, gchen@nju.edu.cn

Abstract

The emergence of long-context text applications utilizing large language models (LLMs) has presented significant scalability challenges, particularly in memory footprint. The linear growth of the Key-Value (KV) cache, which stores attention keys and values to reduce redundant computations, can significantly increase memory usage and may prevent models from functioning properly in memory-constrained environments. To address this issue, we propose a novel approach called Cache Sparse Representation (CSR), which converts the KV cache by transforming the dense Key-Value cache tensor into sparse indexes and weights, offering a more memory-efficient representation during LLM inference. Furthermore, we introduce NeuralDict, a novel neural network-based method to automatically generate the dictionary used in our sparse representation. Our extensive experiments demonstrate that CSR matches the performance of state-of-the-art KV cache quantization algorithms while ensuring robust functionality in memory-constrained environments.

Introduction

The introduction of large language models (LLMs) has brought about a new wave of exciting AI applications, including document summarization, code analysis, extended multi-turn applications, tool learning, and more. Among these applications, those involving long text have garnered significant interest, such as RAG (Retrieval-Augmented Generation). RAG tackles the challenge of generating accurate and pertinent content, particularly in scenarios where queries extend beyond the training data or require up-to-date knowledge, by integrating external information sources. This fusion of RAG with LLMs expands the scope of LLMs and makes them increasingly applicable for specialized and knowledge-driven tasks in real-world contexts. However, the significant number of parameters in LLMs, amounting to tens or hundreds of billions, results in high memory and computation requirements during generation tasks, especially when handling long contexts like RAG. To effectively support large language models (LLMs), it is crucial to batch multiple requests together to minimize the cost per request.

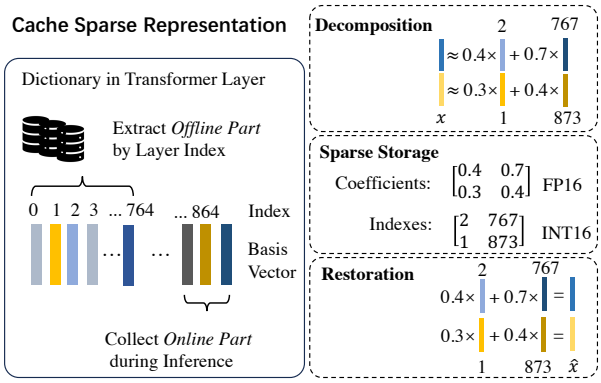


Figure 1: The core idea of CSR is to decompose the original dense vectors into sparse coefficients and indexes based on a dictionary, which consists of an offline-trained *NeuralDict* that learns features related to the KV cache space during training, and components gathered online, thus significantly reducing the memory footprint required by the KV cache.

The key-value (KV) cache utilized to store attention keys and values, and thereby avoid redundant computations, can lead to a substantial increase in memory usage and become a bottleneck for both speed and memory efficiency. The cache's memory consumption grows linearly with the length of the input prompt and the number of tokens generated, overshadowing even the substantial memory requirements of the model parameters. This presents scalability challenges for large language models, as the cache's linearly expanding memory footprint hinders its ability to handle longer sequences. Therefore, it is imperative to develop methods for compressing the KV cache to enable long-sequence inference in a memory-efficient manner.

We provide an overview of existing methods that help mitigate KV cache overhead as follows:(Shazeer 2019) introduces Multi-Query Attention, a variant of Multi-Head Attention (MHA)(Vaswani et al. 2017). MQA allows different attention heads to share the same KV caches, thereby reducing memory usage. Furthermore, (Ainslie et al. 2023) introduces Grouped-Query Attention (GQA), which strikes a balance between the performance degradation caused by MQA and the memory footprint associated with Multi-

*The work done during the internship at Ant Group.
†These authors contributed equally.

Head Attention (MHA). Recent works like YOCO(2024), which reuses global KV caches via cross-attention, and MLA(2024), which compresses KV caches into latent vectors, both improve memory efficiency too. These modifications to the attention mechanism also contribute to a reduction in the memory footprint of the KV cache. Another set of techniques utilizes quantization to reduce the number of bits used by the original stored data type, sacrificing data precision for memory footprint. Additionally, some researchers have taken an alternative approach by lowering the memory footprint of the KV cache through the eviction of unimportant cache parts from the GPU. We will delve into a detailed discussion of these two methods in Related Work.

In this paper, we propose CSR (Cache Sparse Representation), which offers a sparse representation of the KV cache and provides an equivalent but less memory-intensive representation for the original KV cache during LLM inference. Our contributions are outlined as follows:

1. CSR provides an innovative approach to reducing the high memory footprint of the KV cache in long-text LLM applications. It works across different attention mechanisms in transformers while remaining independent of established methods such as KV cache eviction.
2. Our extensive experiments on various models and datasets demonstrate that CSR performs comparably to 4-bit or 2-bit KV cache quantization algorithms under conditions with sufficient memory, while also maintaining robust performance with less than 1 bit per channel in memory-constrained environments.

Preliminary

Sparse Representation

Sparse representation is a well-researched fields in computer vision and pattern recognition. However, to the best of our knowledge, no work yet try using sparse representation to reduce the memory footprint used during inference of LLM. Suppose we have a dictionary $D = [\mathbf{d_1}, \mathbf{d_2}, ..., \mathbf{d_N}] \in \mathbb{R}^{d\times N}$ and each *basis* vector $\mathbf{d}_n$ in D is an l_2-**norm unity** vector. Define the sparsity of a representation vector $\mathbf{r}$ as the l_0 norm of $\mathbf{r}$, which means the number of the nonzero elements of vector $\mathbf{r}$. Given dictionary D and limit maximum representation sparsity as s, for a dense origin vector $\mathbf{x} \in \mathbb{R}^d$, the way to find the sparse representation of $\mathbf{x}$ is solving the following optimization problem:

$$\mathbf{r}(\mathbf{x}, D, s) = \arg\min \|\mathbf{x} - D\mathbf{r}\|^2 \quad \text{s.t.} \quad \|\mathbf{r}\|_0 \leq s \quad (1)$$

where $\mathbf{r} \in \mathbb{R}^N$ means sparse representation of $\mathbf{x}$ with sparsity no greater than s. Among different types of algorithms for solving equation (1), Matching Pursuit(MP) (Mallat and Zhang 1993) is the most widely used one to generate sparse representation satisfying sparsity limitations. MP iteratively choose the best atom from the dictionary based on a certain similarity measurement to approximately obtain the sparse solution. First of all, the residual vector is initialized as $\mathbf{R}_0 = \mathbf{x}$, $\mathbf{r} = \mathbf{0} \in \mathbb{R}^d$. The MP algorithm will determine the optimal atom vector index i_g and the corresponding coefficient c_g through the following formulas:

$$c_g = sup\,|\mathbf{R}_g \cdot \mathbf{d}_n| \quad (2)$$

$$i_g = \underset{n}{\text{argmax}}\,|\mathbf{R}_g \cdot \mathbf{d}_n| \quad (3)$$

where $1 \leq g \leq s$ represents the number of current iterations. Subsequently, update $\mathbf{r}[i_g] = c_g$, and the residual vector R_g is updated based on the part that was already approximated in the previous iteration as following:

$$\mathbf{R}_{g+1} = \mathbf{R}_g - c_g \times \mathbf{d}_{i_g} \quad (4)$$

MP will repeat calculating (2)-(4) until c_s and i_s are calculated and $\|\mathbf{r}\|_0 = s$ exactly.

KV Cache in Attention

LLM inference can be divided into the prefill phase and the decoding phase. In the prefill phase, each token of the input prompt is used to generate KV cache for every transformer layer of LLMs. The model uses and updates the KV cache to generate the next token autoregressively in the decoding phase. Since the KV cache mechanism for different attention heads is the same, we will not consider the attention head index in the subsequent discussion.

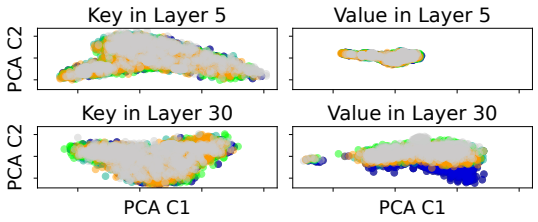
Assuming a model's hidden size is d and the number of key (or value) attention heads is h, let $X_p^\lambda \in \mathbb{R}^{b\times l\times h\times d_h}$ represent the activations of the input prompt p's tokens after being forwarded into transformer layer λ, where b is batch size, l is the length of prompt tokens, and $d_h = d//h$ is the tensor size for each attention head. $W_K^\lambda, W_V^\lambda \in \mathbb{R}^{d\times d}$ of the current layer will map X_p^λ to key and value cache through the following equation:

$$X_{p,\{K,V\}}^\lambda = X_p^\lambda W_{\{K,V\}}^\lambda \quad (5)$$

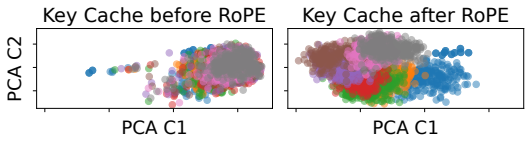
Here, λ is the transformer layer index. $X_{p,K}^\lambda$, $X_{p,V}^\lambda$ are cached in the memory as KV cache of prompt p for layer λ (Here we temporarily ignore the effect of position embedding). During the autoregressive decoding phase, each forward pass generates a new token t, and its corresponding activations after passing through layer λ are represented as $X_t^\lambda \in \mathbb{R}^{b\times 1\times d}$. After being mapped to the key (K) and value (V) space using W_K^λ and W_V^λ, the corresponding $X_{t,K}^\lambda$ and $X_{t,V}^\lambda$ are appended to the KV cache of layer λ. Throughout the remainder of the paper, we will use $X_{\{K,V\}}^\lambda$ to refer to the K or V cache space of the transformer layer λ.

KV Cache Sparse Representation:CSR

In this section, we introduce our method, Cache Sparse Representation (CSR), which utilizes the dictionary that fully extracts KV cache features and replaces dense KV cache vectors with sparse indexes and coefficients to significantly reduce the memory footprint during inference. We initially present our intuitions collected during the LLM inference stage, which directly guide the dictionary construction of CSR. Subsequently, we provide a comprehensive overview of the CSR procedure and delve into the detailed process of constructing the dictionary required by CSR.



(a) Distribution analysis of X_K and X_V across different prompts.



(b) Comparison of X_K with and without RoPE processing.

Figure 2: (a) The distribution of X_K among prompts is nearly identical. While there is substantial spatial overlap in X_V in the shallow layer, few noticeable differences still emerge in the deep layers (e.g., Layer 30). (b) X_K from the same layer is evenly segmented into 8 groups based on their positions. It is evident that, in comparison with the keys processed by RoPE, the keys that have not undergone RoPE processing are thoroughly intermingled, which is better for extracting offline basis vectors.

Intuitions

We extracted a range of prompts from wikitext dataset(Merity et al. 2016), and forward them into Llama2-7B-Chat which is a widely used LLM. Subsequently, we gathered the KV cache generated during model inference. To aid in subsequent observation and research, we reduced the collected KV cache to a two-dimensional space through PCA in the channel dimension. This allowed us to derive the following observations through analysis.

Difference among prompts is nearly ignorable. Following PCA dimensionality reduction to a two-dimensional space, we observe that the spaces covered by different prompts are nearly identical, as depicted in Figure 2a where different colors mean different prompts. This finding suggests that a portion of the constructed dictionary can be shared across different query prompts. We refer this query-independent part as the *offline* part. Note that few noticeable differences still exist in the deep transformer layers. We propose the *online* part to deal with this issue.

Position embedding makes nonstationary Keys. An important consideration in determining the sparse representation for Keys is managing the positional embedding, such as Rotary Positional Embedding (RoPE for short)(Su et al. 2024) which are applied to Keys and Queries in most LLMs to embed relative positional information between Keys and Queries. The nature of these positional embedding causes the Keys to be relatively unstable with regard to position, as depicted in Figure 2b. Due to this phenomenon, we opt to pre-process the Key cache of tokens before introducing the position embedding.

Adjacent transformer layers' KV space is similar. To analyze the differences in $X^{\lambda}_{\{K,V\}}$ between Transformer layers, we first normalize the collected KV cache into l_2-norm unity vectors, then perform PCA in pairs on adjacent layers to reduce the dimension to 2. After that we generate a two-dimensional histogram of 200×200 bins to obtain the discrete distribution of $X_{\{K,V\}}$. Finally, we measure the difference of KV cache space from two transformer layers by calculating the JS divergence between these discrete distributions, part of the results are shown in the figure 3. We observes that the distribution of $X_{\{K,V\}}$ between most adjacent layers is similar. So we decide to to construct a multi-layers shared offline dictionary based on the similarity between layers in order to save memory footprint as much as possible. Take X_K as an example to illustrate the heuristic idea we follow, the set of aggregated layers is denoted as $\mathcal{M}_K = \{\Lambda_1, ...\Lambda_i\}$, and $\Lambda_i = \{\lambda_m, ...\lambda_n\}$. For $\forall i \neq j, \Lambda_i \cap \Lambda_j = \varnothing$, and $\bigcup_i \Lambda_i$ is the set of all transformer layers. Any transformer layer pair (λ_m, λ_n) in the same Λ_i satisfies the following:

$$JSD(P_K^{\lambda_m} \| P_K^{\lambda_n}) \leq \delta_1, \forall \lambda_m, \lambda_n \in \Lambda_i \tag{6}$$

$$\sum_{\lambda_m \in \Lambda_i} JSD(P_K^{\lambda_m} \| P_K^{\lambda_{m+1}}) \leq \delta_2, with \quad \lambda_{m+1} \in \Lambda_i \tag{7}$$

where $P_K^{\lambda_m}$ represents the discrete distribution obtained through dimensional histogram after reducing the dimensions of the vector in the attention head to 2 dimensions using PCA, δ_1 and δ_2 are thresholds used to adjust the aggregation strategy. Both aim to prevent the cache space from becoming excessively large after aggregation.

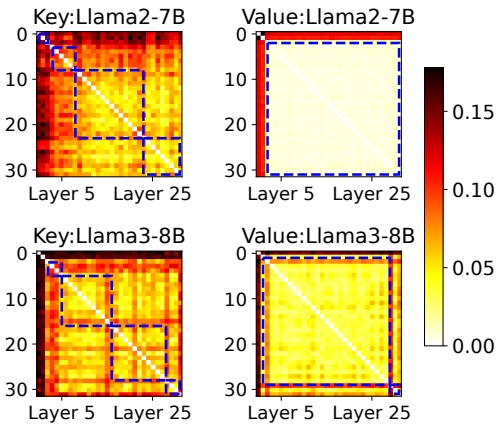


Figure 3: JS divergence for $X_{\{K,V\}}$ from different transformer layers. The lighter the color, the smaller the distribution difference between two layers. We use blue boxes to highlight adjacent layers with similar KV cache space.

Based on these observations, we propose the following guidelines for constructing the dictionary needed for CSR:

- First, the construction of the dictionary will be divided into two parts: offline and online.
- Second, we choose to preprocess X_K prior to the embedding of positional information.

Algorithm 1: NeuralDict

procedure NEURDICT($\mathcal{C}, m, N, E$)
  Input: The calibration corpus dataset $\mathcal{C}$, language model m, offline dictionary size N and training procedure epochs number E.
  Perform inference on dataset $\mathcal{C}$ using model m and collect X_K^m, X_V^m for each layer in model m
  Generate $\mathcal{M}_K$ and $\mathcal{M}_V$ based on Equation 6 and 7.
  TRAINONMERGEDLAYERS($\mathcal{M}_K, N, X, E$)
  TRAINONMERGEDLAYERS($\mathcal{M}_V, N, X, E$)

procedure TRAINONMERGEDLAYERS($\mathcal{M}, N, X, E$)
  for $\Lambda_i \in \mathcal{M}$ **do**
    X=concatenate [X^{λ_n} for $\lambda_n \in \Lambda_i$]
    TRAINNEURDICT(N, X, E)

procedure TRAINNEURDICT($N, X_{\{K,V\}}, e$)
  Input: Offline dictionary size N, and Key cache or Value cache $X_{\{K,V\}}$ in corpus dataset, epochs e to train
  Initialize $W_D = [d_1, \ldots, d_N]$ with cluster centroids of $X_{\{K,V\}}$.
  W_D = RENORM(W_D)
  for $\hat{e} = [1, \ldots, e]$ **do**
    for Batch $\mathcal{B} \in X^{l,h}_{\{K,V\}}$ **do**
      calculate $\mathcal{L}$ using equation 11
      backward $\mathcal{L}$ and update W_D
      W_D = RENORM(W_D)

procedure RENORM(W_D)
  ▷ *Normalize the vector to ensure its l_2 norm unity* ◁
  for i in [1,2,..., N] **do**
    $\widehat{w}_i = \frac{w_i}{\|w_i\|_2}$
  $\widehat{W} = [\widehat{w}_1; \widehat{w}_2; \ldots; \widehat{w}_N]$
  return $\widehat{W}$

- Third, we fuse the $X_{\{K,V\}}$ from adjacent transformer layers to construct a multi-layers shared offline dictionary based on the similarity between layers in order to save memory footprint as much as possible.

CSR's Workflow

We divide CSR into two stages. The first stage is the Preparation Stage, in which CSR probes $X_{\{K,V\}}$ of each transformer layer using the calibration corpus dataset for a new language model, and then aggregates $X_{\{K,V\}}$ of each layer following (6) and (7), and trains to obtain an offline dictionary that can be shared by multiple layers. Another stage is Inference Stage, in which CSR replaces the original KV cache of the language model and utilizes the sparse representations to reduce the GPU memory footprint.

Preparation Stage

The primary concern is how to construct a dictionary that can approximately represent each KV cache tensor in LLM generated by the current query by selecting only s bases in the dictionary. Clustering is a widely used unsupervised learning method for extracting features from a vector space. However, the clustering algorithm does not directly interact with the process of calculating the sparse representation in CSR. As a result, the dictionary constructed by clustering does not take into consideration the features of residual tensors beyond the first iteration in MP. To address these issues, we propose a novel neural network-based method named *NeuralDict* to automatically resolve this problem.

NeuralDict The offline dictionary construction remains consistent across $\mathcal{M}_K$ or $\mathcal{M}_V$ of the model. We utilize the calibration set $\mathcal{C}$ as the corpus dataset to assess the distribution of $X_{\{K,V\}}$ in each layer of the large language model m. For a model m with hidden states size in each attention head as d_h, CSR split the hidden states evenly into s_n chunks according to the order of channels, and given the dictionary D with a size of N, the dictionary we aim to create can be viewed as a matrix $W_D \in \mathbb{R}^{(d_h/s_n)\times N}$. This matrix W_D can be considered as the learnable weights in a single linear layer neural network without any bias or activation function. We utilize the mean squared error as shown in Equation 8 to train W_D. Take Key cache as example:

$$\mathcal{L}_{MSE} = \sum_{x \in X_K} \|\mathbf{x} - W_D \mathbf{r}(\mathbf{x}, W_D, s)\|_2^2 \tag{8}$$

where $\mathbf{r}(\mathbf{x}, W_D, s)$ represents the sparse representation vector calculated by the MP algorithm, and W_D serves as the basis vector's dictionary. The mean squared error $\mathcal{L}_{MSE}$ will be excessively large and difficult to optimize when s is too small, while a large s will result in a prolonged MP process, leading to lower training efficiency. After updating W_D through loss backpropagation, we apply an additional update to W_D as:

$$W_D = \text{ReNorm}(W_D) \tag{9}$$

where ReNorm denotes the normalize each vector to l_2 norm unity as shown in Algorithm 1.

Adaptive Regularization to Encourage Diversity. We include the following regularization term to promote the diversity of vectors in W_D to prevent the training from getting trapped in local optima:

$$\mathcal{L}_{div} = \frac{1}{N^2} \|I - W_D^T W_D\|_F^2 \tag{10}$$

$\|\cdot\|_F$ denotes the Frobenius norm, and $I \in \mathbb{R}^{N\times N}$ represents the identity matrix. Since the magnitude of the mean squared error (MSE) loss varies with the transformer layer while the diversity term does not, we incorporate an adaptive coefficient to adjust the weight of $\mathcal{L}_{MSE}$ and $\mathcal{L}_{div}$:

$$\mathcal{L} = \mathcal{L}_{MSE} + \beta \mathcal{L}_{div} \tag{11}$$

where $\beta = \text{clamp}(0.1 \times \frac{\hat{\mathcal{L}}_{MSE}}{\hat{\mathcal{L}}_{div}}, 0, 1.0)$. Note that $\hat{\mathcal{L}}_{MSE}$ and $\hat{\mathcal{L}}_{div}$ represent the calculated values without any gradient information. The purpose of limiting β to 1.0 is to prevent the model from overly focusing on reducing $\mathcal{L}_{div}$ and disregarding $\mathcal{L}_{MSE}$ when $\mathcal{L}_{div}$ is sufficiently small. The whole training procedure is shown in Algorithm 1.

Inference Stage

When the language model's transformer layer is loaded into the GPU, CSR will load the layer's corresponding offline dictionary onto the same device. Note that due to the existence of Merged Layers, we prefer to load layers corresponding to the same offline dictionary onto the same single device. The whole process of how CSR take place of original KV cache is illustrated in Figure 1.

Build Dictionary For prompt p, CSR build $D_K^\lambda(p)$ and $D_V^\lambda(p)$ as dictionaries for the Key and Value cache. For each transformer layer λ. CSR will extract the corresponding part from the offline dictionary for the transformer layer according to the layer index as show in Figure 1. The bank of the online part is divided into two sections. During the prefill phase, CSR will perform online sampling methods from the calculated KV cache. In order to prevent poor fitting results caused by out-of-distribution entries in the KV cache during inference, we follow the KV quantization framework, such as (Kang et al. 2024), and design a separate module named *Guard part* for handling outlier entries.

KV Decomposition and Sparse Storage CSR will compute the sparse representation for the tokens in the prompt using $D_K^\lambda(q)$ or $D_V^\lambda(q)$ for the $X_{K,V}$ by solving problem 1 using the Matching Pursuit algorithm. The maximum sparsity is set to be s which is so-called *MP-level*, the Matching Pursuit algorithm will perform s iterations to generate sparse representations with a sparsity of s for the entire $X_{\{K,V\}}$. We denote the sparse representations of the KV cache as $\mathbf{r}(X_{\{K,V\}}, D_{\{K,V\}}^\lambda(q), s)$ with sparsity s in layer λ. Please note that there are no more than s non-zero elements in $\mathbf{r}$.Therefore, it is only necessary to store the index and coefficient of these non-zero elements. The index indicates the position of the selected basis vector in the dictionary, while the value represents the corresponding coefficient.

De-Sparse to Restore To meet the needs of calculating attention scores, CSR will de-sparse $\mathbf{r}$ into tensor form:

$$\tilde{X}_{\{K,V\}}^\lambda = D_{\{K,V\}}^\lambda \mathbf{r}(X_{\{K,V\}}^\lambda, D_{\{K,V\}}^\lambda, s) \qquad (12)$$

Here, $\tilde{X}_{\{K,V\}} \in \mathbb{R}^{b \times l_g \times h \times d_h}$, and l_g represents the number of prompt tokens and generated tokens. $\tilde{X}_{\{K,V\}}$ will be used in the attention score calculation instead of original KV cache. When new tokens are generated, the corresponding KV cache will also be replaced by CSR.

Analysis for CSR The initial $X_{K,V}$ of each attention head comprises d_h floating-point values, with fp16 as the prevalent datatype in LLM inference. With CSR, only $s \times s_n$ fp16 values for coefficients, accompanied by $s \times s_n$ INT16 values for indexes. The compression rate can be calculated as $\frac{16 \times d_h}{16 s \times s_n + 16 s \times s_n} = \frac{d_h}{2 s \times s_n}$, which implies that for CSR(s, s_n), the number of bits of the corresponding quantization algorithm is $\frac{16}{d_h / 2 s \times s_n} = \frac{32 s \times s_n}{d_h}$ bits. Taking LLaMA3-8B whose d_h =128 as an example, for CSR($s = 4, s_n = 1$) or ($s = 2, s_n = 2$), the corresponding quantization bit count is 1 bit.

Experiments

Experiments Settings

Models We applied CSR to multiple large language models (LLMs) based on the HuggingFace Transformers library[1]. To evaluate CSR's effectiveness across different attention mechanisms, we conducted experiments on Llama2-7B-chat(Touvron et al. 2023b,a), Llama3-8B-Instruct(AI@Meta 2024), and Baichuan2-7B-chat(Baichuan 2023). Specifically, Llama2-7B-chat and Baichuan2-7B-chat utilize MHA, while Llama3-8B-Instruct adopts GQA.

Benchmark The primary goal of CSR is to reduce the memory usage of the KV cache by identifying sparse representations for the KV cache within a long context setting. To evaluate its effectiveness, we utilized the LongBench benchmark (Bai et al. 2023), which is a bilingual and multitask benchmark designed to assess the long context understanding capabilities of LLM. In our evaluation, we relied on standard metrics such as F1 score, ROUGE score, and similarity score. These metrics align with the settings established in (Liu et al. 2024) for different datasets within the LongBench.

CSR In the experiments, unless stated otherwise, the Value Cache uses $s_n = 2$, and the Key Cache uses $s_n = 1$. For simplicity, CSR-s denotes the MP-level. Since $s_n = 2$ for the Value Cache, its maximum MP-level is half that of the Key Cache. For example, CSR-8 corresponds to $s = 8, s_n = 1$ for the Key Cache, and $s = 4, s_n = 2$ for the Value Cache. The experimental results presented in the main content uniformly adopt reverse sequential sampling for the online sampling part due to space constraints. In CSR's online part, the *Guard* size per layer is 8192, and the sampling size is 4096 for Llama2 and Baichuan2. For Llama3, the *Guard* size is 2048, with a sampling size of 1024. For further details on the offline part and additional ablation studies, please refer to our extended version[2].

Baselines We selected state-of-the-art (SOTA) KV cache quantization algorithms to establish a robust baseline for measuring CSR performance. These included:

- KIVI(2024): KIVI introduced a tuning-free quantization algorithm known as KIVI-2 and KIVI-4 for 2 bits and 4 bits correspondingly, and quantize the Key cache per-channel and the Value cache per-token.
- GEAR(2024): GEAR applies 4-bit quantization to the majority of entries in the KV cache and utilizes a low-rank matrix to approximate the quantization error. Additionally, GEAR uses a sparse matrix to handle outliers.

Hardware Environment A single NVIDIA A100 GPU (80GB) with 128GB memory.

Robust Performance on Various Tasks

Initially, we present a comparison between CSR and various quantization algorithms on the Llama2-7B-chat model. For KIVI and GEAR, we perform grid search on hyperparameters and show the results of the best obtained. The hidden size of each attention head in the Llama2-7B-chat model is

[1]https://huggingface.co/docs/transformers

[2]https://arxiv.org/pdf/2412.11741

Method	2wiki-mqa	hotp-otqa	musi-que	trec	narr-ative-qa	qas-per	qm-sum	sam-sum	lcc	trivi-aqa	multi-field-qa_en	Avg
FP16	26.47	33.84	9.33	68.14	16.65	17.17	20.81	40.88	58.25	82.74	35.62	37.26
GEAR	26.52	32.65	9.01	68.17	16.78	18.03	21.12	41.66	57.59	83.53	36.65	37.42
KIVI-4	26.08	33.48	9.53	68.14	17.14	17.16	20.61	40.33	58.07	82.49	36.06	37.19
CSR-16	26.19	**34.19**	9.22	68.14	**17.44**	16.99	21.00	40.62	**58.07**	82.85	36.23	37.36
KIVI-2	26.06	32.13	9.71	68.14	16.69	19.30	20.79	39.31	57.21	83.23	36.32	37.17
CSR-8	**26.86**	**33.52**	9.06	**68.17**	**16.70**	16.15	20.48	38.52	**57.47**	82.91	35.41	36.93
CSR-6	26.28	33.76	9.13	67.75	16.18	14.96	20.68	38.24	56.53	83.41	33.43	36.39
CSR-4	25.54	31.90	8.55	66.29	15.70	13.42	20.29	37.37	54.19	81.64	31.50	35.13

Table 1: We conducted experiments on CSR methods and corresponding quantization methods with an identical number of bits, using Llama2-7B-chat. We highlight the data where our method performs better within the same precision group. As there is no equivalent method for quantization below 2 bits, we only present CSR-4 and CSR-6.

Model	Meth-od	2wiki-mqa	hotp-otqa	musi-que	trec	narr-ative-qa	qas-per	qm-sum	sam-sum	lcc	trivi-aqa	mult-ifield-qa_en	Avg
Lla-ma3	FP16	32.17	33.23	17.87	74.54	20.24	29.22	22.83	42.63	56.49	88.79	38.81	41.51
	CSR-16	31.83	35.26	17.87	74.29	20.13	29.56	22.39	41.19	57.58	89.04	40.05	41.74
	CSR-8	29.42	35.82	17.39	73.92	19.68	26.98	22.44	40.73	57.10	89.62	36.90	40.91
	CSR-6	29.51	34.76	18.04	73.16	19.99	24.26	21.94	39.86	55.79	89.35	35.73	40.22
	CSR-4	28.52	35.28	16.87	71.61	19.70	22.03	21.54	38.19	54.95	89.01	33.14	39.17
Baic-huan2	FP16	20.13	26.52	11.51	73.46	17.66	21.02	22.04	17.05	63.53	81.02	39.78	35.79
	CSR-16	19.64	25.99	11.68	73.46	17.98	19.94	21.79	17.82	62.86	80.44	39.69	35.57
	CSR-8	18.90	25.92	11.63	73.29	17.24	19.26	21.70	20.24	61.67	81.50	38.37	35.43
	CSR-6	18.73	24.38	11.61	72.96	17.85	16.70	21.47	22.28	60.87	79.26	38.20	34.94
	CSR-4	18.06	24.36	10.40	70.97	16.79	15.01	20.86	23.12	58.36	78.29	33.58	33.62

Table 2: Experiments on Longbench using Llama3-8B and Baichuan2-7B show that CSR is also effective for these models.

128 so according to the previous analysis, CSR-8 is equivalent to 2 bits in quantization, and CSR-16 is equivalent to 4 bits in quantization. We grouped several methods according to equivalent quantization levels, namely FP16 corresponding to 16 bits, GEAR, KIVI-4 and CSR-16 corresponding to 4 bits, and KIVI-2 and CSR-8 corresponding to 2 bits in Table 1. The performance of various methods on multiple datasets, is presented in Table 1. For the 4-bit group, our method performs better than KIVI and GEAR on most datasets, while for the 2-bit group, our method and KIVI have their own advantages and disadvantages. We conclude that CSR, KIVI, and GEAR exhibit similar performances and CSR can provide performance comparable to state-of-the-art 4-bit or 2-bit quantization algorithms.

Effective CSR with Less Than 2 bit: There is no way to reduce from 2bit to 1bit for quantization based methods. However, CSR can provide sparse representation for all KV caches at less than 2 bits or even 1 bit per channel, thus alleviating the tight memory resources of the GPU without any KV cache eviction. We conducted extensive experiments with CSR-6, equivalent to 1.5 bit, and CSR-4, equivalent to only 1 bit. In this scenario, CSR can still maintain performance on most datasets, with only a slight performance drop as shown in Table 1. Even CSR-4 only drops 5.7% in model performance compared to FP16, with less than $\frac{1}{10}$ memory occupied by KV cache.

CSR Works Well for Various Language Models

CSR is independent of the attention mechanism utilized in LLM, making it theoretically applicable to various models. In order to validate the versatility of our method across different models, we conducted more experiments on Baichuan2-7B, Llama3-8B-Instruct. As depicted in Table 2. The results demonstrate that despite providing at least an 8x compression ratio compared to the original data type, CSR still delivers strong performance across all models. The performance loss of CSR-4 compared to FP16 is 5.7% for Llama3-8B-Instruct and 6.0% for Baichuan2-7B-chat according to the value in *Avg* column.

Memory Foorpint

We plotted the relationship between the inference length and memory footprint of KV cache for different models using different methods. As shown in Figure 4, the additional

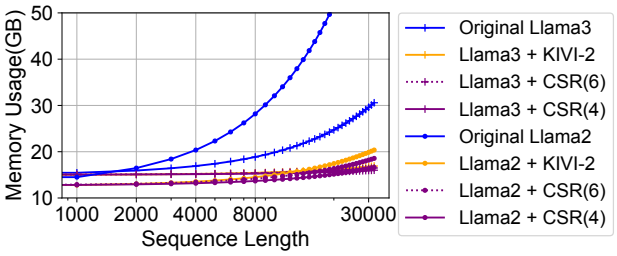


Figure 4: The figure is based on the Llama2-7B-chat and Llama3-8B-Instruct models, and shows the memory footprint used when using different methods for inference with batch size = 4. The x-axis is sequence length in log-scale, and the y-axis is the occupied memory.

memory overhead introduced by the offline or online dictionary is almost negligible. Compared with the original KV cache, both CSR and quantization algorithms have greatly reduced the memory occupied by the KV cache. Compared with quantization, which cannot be further reduced from 2 bits, CSR provides the possibility of further reducing memory usage in long context scenarios.

Effect of s_n and NeuralDict Size

We analyze the role of s_n based on the training results on *NeuralDict* for Llama2-7B-chat model. The MSE loss on the test dataset is shown in the table 3. Overall, from the perspective of Key cache and Value cache, the improvement of splitting the Value cache is very obvious, but there is almost no improvement on Key cache. Specifically, when $s = 8$, the most significant improvement occurs when s_n increases from 1 to 2 for Value cache. After a comprehensive trade-off between performance and compression rate, we fix $s_n = 1$ for the Key and $s_n = 2$ for Value in CSR in actual use.

Value Cache				Key Cache			
size	s	s_n	**Loss**	**size**	s	s_n	**Loss**
8192	4	1	0.259	8192	4	1	0.079
	4	2	0.253		4	2	0.068
	4	4	0.093		4	4	0.052
	8	1	0.213		8	1	0.055
	8	2	0.094		8	1	0.047
	8	4	0.012		8	1	0.037
16384	4	1	0.263	16384	4	1	0.077
	4	2	0.223		4	2	0.058
	4	4	0.078		4	4	0.042
	8	1	0.194		8	1	0.047
	8	2	0.079		8	2	0.043
	8	4	0.008		8	4	0.033

Table 3: MSE loss on test dataset of the converged *NeuralDict* trained with different s and s_n parameters.

Related Work

KV Cache Quantization Quantization is an alternative method for reducing memory and compute requirements during generation tasks, particularly in processing extremely long contexts. Prior research, such as that by (Hooper et al. 2024; Yue et al. 2024), has focused on quantizing the KV cache. Meanwhile, (Liu et al. 2024) proposes quantizing the key cache per-channel and the value cache per-token, (Kang et al. 2024) propose to use SVD to reduce the quantization error. However, these approaches are not applicable when the per-token quantization falls below 2 bits.

KV Cache Eviction Various approaches exist to minimize the KV cache footprint, with the common objective of retaining only a small subset of keys and values. One technique utilizes the attention mechanism's localized pattern, namely the attention sink, as proposed by (Xiao et al. 2023). This involves employing a finite attention window to retain only the *sink* token and a fixed number of recent tokens. Another strategy involves implementing a KV cache eviction policy considering the attention mechanism's sparsity. For example, (Zhang et al. 2023; Ge et al. 2023) suggest discarding non-essential parts of the KV cache to reduce memory usage during large language model (LLM) inference. Moreover, (Liu et al. 2023) identifies a repetitive attention pattern during inference processes, recommending the retention of only *pivotal* tokens. Additionally, (Anagnostidis et al. 2023) employs a learnable mechanism to identify uninformative tokens, implementing adaptive sparse attention that requires fine-tuning on the pre-trained model.

Conclusion

This paper introduces CSR, a framework for optimizing the memory footprint of the KV cache during LLM inference, based on compressed sensing algorithms. Our experiments on widely-used LLMs and long-context datasets have demonstrated that CSR's performance comparable to quantized algorithms when memory resources are relatively abundant (in comparison to 2-bit or 4-bit KV cache quantized algorithms). Furthermore, CSR exhibits robust performance even when memory is more constrained, aiming for less than 2 bits per channel. Notably, even with a per-channel bit count as low as 1, CSR can maintain robust performance. We believe that CSR provides an alternative approach for compressing the KV cache independently of quantization-related algorithms. We conclude that CSR can operate effectively in memory-constrained environments and maintain strong performance across a variety of tasks, even with extremely low memory usage of KV cache.

Limitations

Compared with the quantization algorithm, CSR further reduces the memory occupied by the KV cache. However, the process of detecting the KV cache space of the model through the calibration dataset and then obtaining a part of the dictionary through offline training is time-consuming. We leave the research for a more efficient way to obtain the offline dictionary as future exploration.

Acknowledgments

This work was supported by Ant Group Research Intern Program. This work was also supported in part by National Key R&D Program of China (2022YFB2901800) and the NSF of China (62422207, 62172206).

References

AI@Meta. 2024. Llama 3 Model Card.

Ainslie, J.; Lee-Thorp, J.; de Jong, M.; Zemlyanskiy, Y.; Lebr'on, F.; and Sanghai, S. K. 2023. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. *ArXiv*, abs/2305.13245.

Anagnostidis, S.; Pavllo, D.; Biggio, L.; Noci, L.; Lucchi, A.; and Hofmann, T. 2023. Dynamic Context Pruning for Efficient and Interpretable Autoregressive Transformers. *ArXiv*, abs/2305.15805.

Bai, Y.; Lv, X.; Zhang, J.; Lyu, H.; Tang, J.; Huang, Z.; Du, Z.; Liu, X.; Zeng, A.; Hou, L.; Dong, Y.; Tang, J.; and Li, J. 2023. LongBench: A Bilingual, Multitask Benchmark for Long Context Understanding. arXiv:2308.14508.

Baichuan. 2023. Baichuan 2: Open Large-scale Language Models. *arXiv preprint arXiv:2309.10305*.

Ge, S.; Zhang, Y.; Liu, L.; Zhang, M.; Han, J.; and Gao, J. 2023. Model tells you what to discard: Adaptive kv cache compression for llms. *arXiv preprint arXiv:2310.01801*.

Hooper, C.; Kim, S.; Mohammadzadeh, H.; Mahoney, M. W.; Shao, Y. S.; Keutzer, K.; and Gholami, A. 2024. KVQuant: Towards 10 Million Context Length LLM Inference with KV Cache Quantization. *ArXiv*, abs/2401.18079.

Kang, H.; Zhang, Q.; Kundu, S.; Jeong, G.; Liu, Z.; Krishna, T.; and Zhao, T. 2024. Gear: An efficient kv cache compression recipefor near-lossless generative inference of llm. *arXiv preprint arXiv:2403.05527*.

Liu, Z.; Desai, A.; Liao, F.; Wang, W.; Xie, V.; Xu, Z.; Kyrillidis, A.; and Shrivastava, A. 2023. Scissorhands: Exploiting the Persistence of Importance Hypothesis for LLM KV Cache Compression at Test Time. *ArXiv*, abs/2305.17118.

Liu, Z.; Yuan, J.; Jin, H.; Zhong, S.; Xu, Z.; Braverman, V.; Chen, B.; and Hu, X. 2024. KIVI: A Tuning-Free Asymmetric 2bit Quantization for KV Cache. *ArXiv*, abs/2402.02750.

Mallat, S. G.; and Zhang, Z. 1993. Matching pursuits with time-frequency dictionaries. *IEEE Transactions on signal processing*, 41(12): 3397–3415.

Merity, S.; Xiong, C.; Bradbury, J.; and Socher, R. 2016. Pointer Sentinel Mixture Models. arXiv:1609.07843.

Shao, Z.; Dai, D.; Guo, D.; Liu), B. L. B.; and Wang, Z. 2024. DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model. *ArXiv*, abs/2405.04434.

Shazeer, N. M. 2019. Fast Transformer Decoding: One Write-Head is All You Need. *ArXiv*, abs/1911.02150.

Su, J.; Ahmed, M.; Lu, Y.; Pan, S.; Bo, W.; and Liu, Y. 2024. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568: 127063.

Sun, Y.; Dong, L.; Zhu, Y.; Huang, S.; Wang, W.; Ma, S.; Zhang, Q.; Wang, J.; and Wei, F. 2024. You Only Cache Once: Decoder-Decoder Architectures for Language Models. *ArXiv*, abs/2405.05254.

Touvron, H.; Lavril, T.; Izacard, G.; Martinet, X.; Lachaux, M.-A.; Lacroix, T.; Rozière, B.; Goyal, N.; Hambro, E.; Azhar, F.; Rodriguez, A.; Joulin, A.; Grave, E.; and Lample, G. 2023a. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971.

Touvron, H.; Martin, L.; Stone, K.; Albert, P.; Almahairi, A.; Babaei, Y.; Bashlykov, N.; Batra, S.; Bhargava, P.; Bhosale, S.; et al. 2023b. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Vaswani, A.; Shazeer, N. M.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, L.; and Polosukhin, I. 2017. Attention is All you Need. In *Neural Information Processing Systems*.

Xiao, G.; Tian, Y.; Chen, B.; Han, S.; and Lewis, M. 2023. Efficient Streaming Language Models with Attention Sinks. *ArXiv*, abs/2309.17453.

Yue, Y.; Yuan, Z.; Duanmu, H.; Zhou, S.; Wu, J.; and Nie, L. 2024. WKVQuant: Quantizing Weight and Key/Value Cache for Large Language Models Gains More. *ArXiv*, abs/2402.12065.

Zhang, Z. A.; Sheng, Y.; Zhou, T.; Chen, T.; Zheng, L.; Cai, R.; Song, Z.; Tian, Y.; Ré, C.; Barrett, C. W.; Wang, Z.; and Chen, B. 2023. H2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models. *ArXiv*, abs/2306.14048.