Lexico: Extreme KV Cache Compression via Sparse Coding over Universal Dictionaries

Junhyuck Kim [1] **Jongho Park** [1] **Jaewoong Cho** [1] **Dimitris Papailiopoulos** [2 3]

Abstract

We introduce Lexico, a novel KV cache compression method that leverages sparse coding with a universal dictionary. Our key finding is that key-value cache in modern LLMs can be accurately approximated using sparse linear combination from a small, input-agnostic dictionary of ∼4k atoms, enabling efficient compression across different input prompts, tasks and models. Using orthogonal matching pursuit for sparse approximation, Lexico achieves flexible compression ratios through direct sparsity control. On GSM8K, across multiple model families (Mistral, Llama 3, Qwen2.5), Lexico maintains 90-95% of the original performance while using only 15-25% of the full KV-cache memory, outperforming both quantization and token eviction methods. Notably, Lexico remains effective in low memory regimes where 2-bit quantization fails, achieving up to 1.7× better compression on LongBench and GSM8K while maintaining high accuracy. Our code is available at https://github.com/krafton-ai/lexico.

1. Introduction

Transformers (Vaswani et al., 2017) have become the backbone of frontier Large Language Models (LLMs), driving progress in domains beyond natural language processing. However, Transformers are typically limited by their significant memory requirements. This stems not only from the large number of model parameters, but also from the having to maintain the key-value (KV) cache that grows proportional to the model size (i.e., the number of layers, heads, and also embedding dimension) and token length of the input. Additionally, serving each model session typically requires its own KV cache, limiting opportunities for reuse across different user inputs, with the exception of prompt caching that only works for identical input prefixes. This creates a bottleneck in generation speed for GPUs with limited memory (Yu et al., 2022). Thus, it has become crucial to alleviate KV cache memory usage while preserving its original performance across domains.

Extensive post-training KV cache optimization research (Kwon et al., 2023; Lin et al., 2024; Ye et al., 2024) offers off-the-shelf methods for pretrained LLMs, including selectively retaining tokens (Beltagy et al., 2020; Xiao et al., 2023; Zhang et al., 2024b) and quantizing KV cache to 2 or 4 bits (Liu et al., 2024b; He et al., 2024; Kang et al., 2024). However, eviction strategies have limitations on complex reasoning tasks that require retaining a majority of previous tokens, while quantizations to 2 or 4 bits have clear upper bounds on compression rates.

In this paper, we focus on utilizing low-dimensional structures for efficient KV cache compression. Prior work reports that each key vector lies in a low-rank subspace (Singhania et al., 2024; Wang et al., 2024b; Yu et al., 2024). Yet, it is unclear whether *all vectors* lie in the same subspace; if so, such redundancy remains to be taken advantage of. Thus, we naturally ask the following questions:

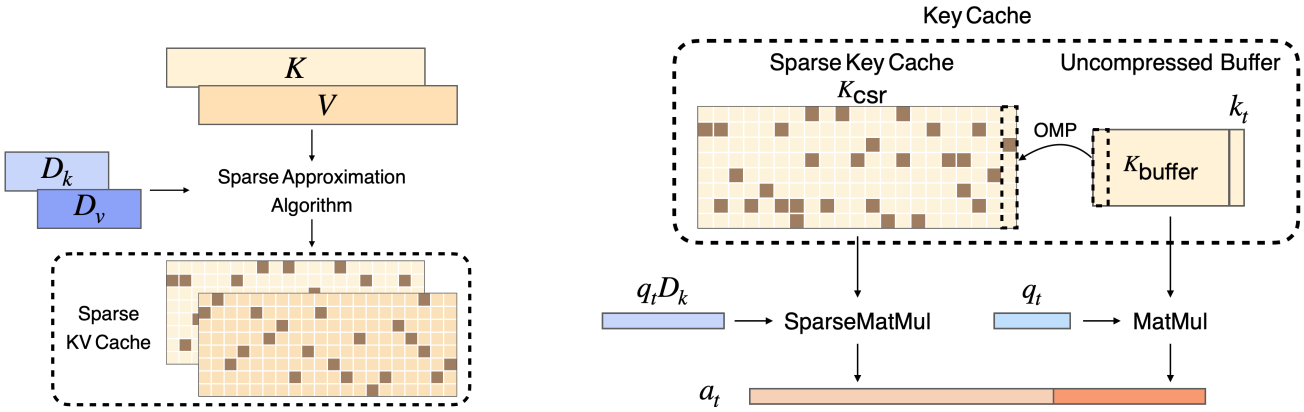
Do keys and values lie in low-dimensional subspaces across diverse input sequences? If so, can we leverage this for efficient KV cache compression?

Towards this end, we propose Lexico, a universal dictionary that serves as an overcomplete basis, which can sparsely decompose and reconstruct the KV cache with sufficiently small reconstruction error that can be directly controlled via the level of sparsity of each reconstruction.

In Section 2.2, we report our observation that a subset of key vectors cluster near each other, even though the keys are from different inputs, while some cluster in different subspaces. To take advantage of such a low-dimensional structure, we draw inspiration from compressed sensing and dictionary learning, areas of statistical learning and signal processing that developed algorithms for information compression across various domains (Candès et al., 2006; Donoho, 2006; Dong et al., 2014; Metzler et al., 2016).

[1]KRAFTON [2]University of Wisconsin-Madison [3]Microsoft Research. Correspondence to: Dimitris Papailiopoulos <dimitris@papail.io>.

Proceedings of the 42^{nd} International Conference on Machine Learning, Vancouver, Canada. PMLR 267, 2025.



(a) Prefilling Stage (b) Decoding Stage

Figure 1. **(a) Prefilling:** Following attention computation, Lexico uses OMP to find sparse representations of the KV vectors (3-8× smaller). **(b) Decoding:** Key cache consists of the compressed sparse key cache, $\boldsymbol{K}_{\text{csr}}$, and an full-precision buffer, $\boldsymbol{K}_{\text{buffer}}$, for the most recent tokens. $\boldsymbol{q}_t$, $\boldsymbol{k}_t$ represent the query, key vectors for the newly generated token. Computation is reduced by computing the query-dictionary product, $\boldsymbol{q}_t\boldsymbol{D}_k$, then multiplying $\boldsymbol{K}_{\text{csr}}$, to get the pre-softmax attention score.

Lexico is simple to learn, can be applied off-the-shelf for KV cache compression, and only occupies small constant memory regardless of input or batch size.

Methodologically, Lexico utilizes both sparsity-based compression (steps 1 and 2) and quantization (step 3) in three straightforward steps:

1. **Dictionary pretraining:** For our experiments, we train a dictionary on WikiText-103 (Merity, 2016) for each model. This dictionary is only trained once and used universally across all tasks. It only occupies constant memory and does not increase with batch size. We note that this dictionary can be trained from richer sources to improve the overall performance of our sparse approximation algorithms.
2. **Sparse decomposition:** During prefilling and decoding (Figure 1), Lexico decomposes key-value pairs into a sparse linear combination, which consists of s pairs of reconstruction coefficients and dictionary indices pointing. This step by itself provides high compression rates.
3. **Lightweight sparse coefficients:** We obtain higher KV cache compression rates by representing the sparse coefficients in 8 bits instead of `FP16`. Lowering precision to 8 bits yields minimal degradation. Lexico theoretically allows us to compress more than 2-bit quantization ($1/8$ of `FP16` KV cache size) if $s \leq 10$ when head dimension is 128.

Overall, we make the following contributions:

- **Near-lossless performance:** Given similar memory requirements, Lexico performs on par with or better than baseline quantization methods on challenging language tasks, such as LongBench (Bai et al., 2023) and GSM8K (Cobbe et al., 2021).
- **Compression rates beyond 2-bits:** Lexico's sparsity parameter enables both wider and more fine-grained control over desired memory usage. This allows us to explore performance when using under 15-20% of the original KV cache size, a low-memory regime previous compression methods could not explore.
- **Universality:** Instead of an input-dependent dictionary, we find a sufficiently small universal dictionary (per model) that can be used for all tasks and across multiple users. Advantageously, such dictionary does *not* scale with batch size and can be used off-the-shelf.

2. KV Cache Compression with Dictionaries

2.1. Background & Notation

During autoregressive decoding in Transformer, the key and value states for preceding tokens are independent of subsequent tokens. As a result, these key and value states are cached to avoid recomputation, thereby accelerating the decoding process.

Let the input token embeddings be denoted as $\boldsymbol{X} \in \mathbb{R}^{l_{\text{seq}} \times d}$, where l_{seq} and d are the sequence length and model hidden dimension, respectively. For simplicity, we focus on a single layer and express the computation of query, key, and value states at each attention head during the prefilling stage as:

$$\boldsymbol{Q}^{(h)} = \boldsymbol{X}\boldsymbol{W}_q^{(h)}, \quad \boldsymbol{K}^{(h)} = \boldsymbol{X}\boldsymbol{W}_k^{(h)}, \quad \boldsymbol{V}^{(h)} = \boldsymbol{X}\boldsymbol{W}_v^{(h)},$$

where $\boldsymbol{W}_q^{(h)}, \boldsymbol{W}_k^{(h)}, \boldsymbol{W}_v^{(h)} \in \mathbb{R}^{d \times m}$ are the model weights with m representing the head dimension.

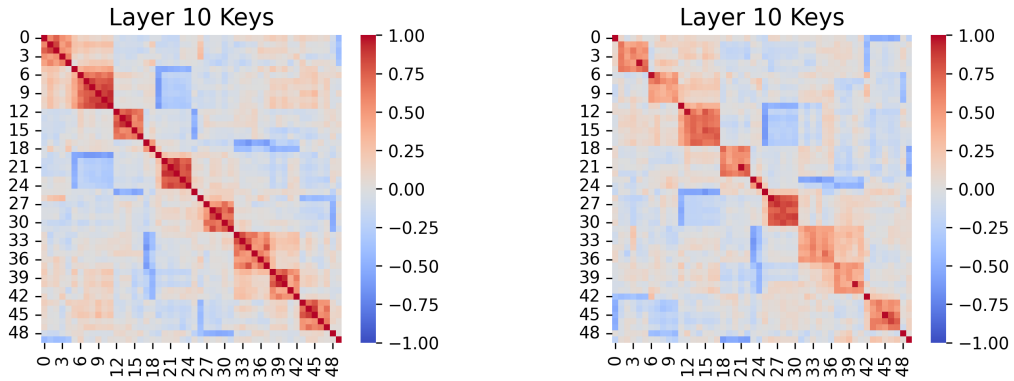


Figure 2. **Left** shows a pairwise cosine similarity matrix between key vectors generated from one input text from all heads in Layer 10 of Llama-3.1-8B-Instruct. Keys are sorted by similarity to demonstrate the clusters. **Right** shows the similarity matrix between key vectors from two *different* input texts. These plots indicate that there may exist a mixture of low-dimensional subspaces in the space of *all* possible keys, a hypothesis that naturally leads to dictionary learning.

Let t represent the current step in the autoregressive decoding, and let $\boldsymbol{x}_t \in \mathbb{R}^{1\times d}$ denote the embedding of the newly generated token. The KV cache up to but not including the current token, are denoted as $\boldsymbol{K}_{t-1}^{(h)}$ and $\boldsymbol{V}_{t-1}^{(h)}$, respectively. The typical output computation for each attention head $\boldsymbol{h}_t^{(h)}$ using the KV cache can be expressed as:

$$\boldsymbol{h}_t^{(h)} = \text{Softmax}\left(\frac{\boldsymbol{q}_t^{(h)}\left(\boldsymbol{K}_{t-1}^{(h)} \parallel \boldsymbol{k}_t^{(h)}\right)^\top}{\sqrt{m}}\right)\left(\boldsymbol{V}_{t-1}^{(h)} \parallel \boldsymbol{v}_t^{(h)}\right). \tag{1}$$

where $\boldsymbol{q}_t^{(h)}, \boldsymbol{k}_t^{(h)}, \boldsymbol{v}_t^{(h)}$ represent the query, key, and value vectors for the new token embedding $\boldsymbol{x}_t$. Here, $\parallel$ denotes concatenation along the sequence length dimension.

2.2. Sparse Approximation

Given a dictionary, our goal is to decompose and represent KV cache efficiently, *i.e.*, approximate a vector $\boldsymbol{k} \in \mathbb{R}^m$ as a linear combination of a few vectors (atoms) from an overcomplete dictionary $\boldsymbol{D} \in \mathbb{R}^{m\times N}$. This reconstruction is given by $\boldsymbol{k} = \boldsymbol{D}\boldsymbol{y}$, where $\boldsymbol{y} \in \mathbb{R}^N$ is the sparse representation vector such that $s = ||\boldsymbol{y}||_0$. For implementation, $\boldsymbol{y}$ only requires space proportional to s, not N.

We hypothesize that the KV cache, like other domains where sparse approximation is effective, contains inherent redundancy that can be leveraged for efficient compression. For instance, Figure 2 presents pairwise cosine similarity plots for keys generated during inference on a random subset of the WikiText dataset. Here, we observe that key vectors cluster in multiple different subspaces. Dictionary learning can take advantage of such redundancy, enabling KV vectors to be represented by a compact set of atoms with only a few active coefficients.

Sparse approximation, which aims to find $\boldsymbol{y}$ with minimum sparsity given $\boldsymbol{k}$ and $\boldsymbol{D}$, while ensuring a small reconstruction error, is NP-hard. This optimization problem is typically formulated as:

$$\min_{\boldsymbol{y}} \|\boldsymbol{y}\|_0 \text{ subject to } \|\boldsymbol{k} - \boldsymbol{D}\boldsymbol{y}\|_2 \le \delta \|\boldsymbol{k}\|_2,$$

for some relative error threshold $\delta > 0$.

In this work, we adopt Orthogonal Matching Pursuit (OMP) as the sparse approximation algorithm. Given an input key or value vector $\boldsymbol{k}$, a dictionary $\boldsymbol{D}$, and a target sparsity s, OMP iteratively selects dictionary atoms to minimize the ℓ_2-reconstruction error, with the process continuing until the specified sparsity s is reached. Our implementation of OMP builds on advanced methods that utilize properties of the Cholesky inverse (Zhu et al., 2020) to optimize performance. Additionally, we incorporate implementation details from (Lubonja et al., 2024) for efficient batched GPU execution and extend it to include an extra batch dimension, allowing for parallel processing across multiple dictionaries. The full algorithm is detailed in Appendix A.

2.3. Learning Layer-specific Dictionaries

Layer-specific dictionaries. While the sparse approximation algorithm is crucial, achieving a high compression ratio relies heavily on well-constructed dictionaries. In this section, we describe the process for training the dictionaries used in Lexico. We adopt distinct dictionaries for the key and value vectors in each transformer layer due to their different functionalities. We denote the key and value dictionaries at each layer as $\boldsymbol{D}_k$ and $\boldsymbol{D}_v \in \mathbb{R}^{m\times N}$, where N is the fixed dictionary size. With $N = 1024$, the dictionaries add an additional 16.8MB to the model's storage requirements for 7B/8B models.

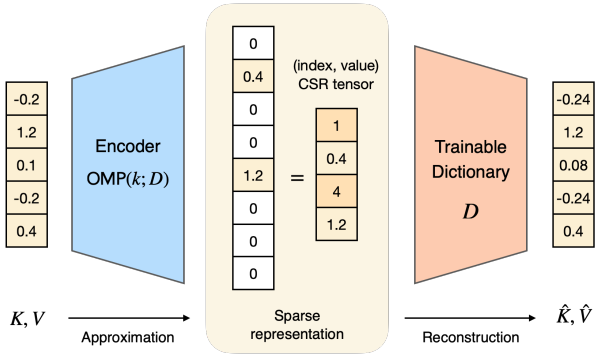


Figure 3. **Dictionary Learning of Lexico.** We train a linear layer $\boldsymbol{D}$ (our dictionary) that minimizes ℓ_2-reconstruction error of KV cache. The KV cache of layer i are used as training data for dictionary $\boldsymbol{D}^{(i)}$. Each step, we apply OMP with fixed $\boldsymbol{D}$ to represent KV as a vector of sparse coefficients; we then perform a step of gradient descent on $\boldsymbol{D}$ and repeat the process. A sparse vector can be efficiently stored as a CSR, using a tuple of 16-bit index and 8-bit value.

As shown in Figure 3, we train layer-specific KV dictionaries via direct gradient-based optimization. For a given key or value vector, denoted as $\boldsymbol{k} \in \mathbb{R}^m$ and a dictionary $\boldsymbol{D} \in \mathbb{R}^{m \times N}$, the OMP algorithm approximates the sparse representation $\boldsymbol{y} \in \mathbb{R}^N$. This process is parallelized across multiple dictionaries, but for simplicity, we present the notation for a single dictionary. The dictionary training objective minimizes the ℓ_2 norm of the reconstruction error, with the loss function $\mathcal{L} = \|\boldsymbol{k} - \boldsymbol{D}\boldsymbol{y}\|_2^2$. We enforce unit norm constraints on the dictionary atoms by removing any gradient components parallel to the atoms before applying updates.

Training. The dictionaries are trained on KV pairs generated from the WikiText-103 dataset using Adam (Kingma & Ba, 2014) with a learning rate of 0.0001 and a cosine decay schedule over 20 epochs. The dictionaries are initialized with a uniform distribution. Table 1 summarises training time for Llama-3.1-8B-Instruct on a single NVIDIA A100 at different sparsity s and dictionary size N.

We demonstrate our trained dictionaries reconstruct and generalize better than dictionaries trained using sparse autoencoders (similarly to those from Makhzani & Frey (2013); Bricken et al. (2023)) across several corpora in Table 2. Our method consistently achieves lower relative reconstruction errors, such as 0.19 ± 0.05 on out-of-domain dataset CNN/DailyMail, and this trend is consistent across other datasets.

Table 1. **Dictionary training time** (minutes) for Llama-3.1-8B-Instruct on an A100 GPU for different sparsity levels s and dictionary sizes N.

Dictionary size	s=4	s=8	s=16	s=32
N=1024	37	52	70	119
N=4096	78	100	160	322

Table 2. **Reconstruction error.** Relative reconstruction errors of different methods when training dictionary of size 1024 and sparsity $s = 32$ on WikiText-103. Sparse Autoencoder is a two-layer perceptron with hard top-k thresholding as activation (encoder as a linear layer + activation in Figure 3). Lexico is optimized using OMP as encoder. KV cache is generated from Llama-3.1-8B-Instruct.

Test Dataset	**Lexico**	**Sparse Autoencoder**	**Random Dictionaries**
WikiText-103	$\mathbf{0.17} \pm 0.06$	0.20 ± 0.05	0.27 ± 0.02
CNN/DailyMail	$\mathbf{0.19} \pm 0.05$	0.22 ± 0.04	0.27 ± 0.02
IMDB	$\mathbf{0.18} \pm 0.05$	0.22 ± 0.05	0.27 ± 0.02
TweetEval	$\mathbf{0.18} \pm 0.06$	0.21 ± 0.05	0.27 ± 0.02

Despite being trained only on WikiText-103, Lexico dictionaries demonstrate a degree of universality: our dictionaries achieve lower test loss on out-of-domain datasets such as TweetEval than the test loss on WikiText-103 for sparse autoencoders, offering significant compression with minimal reconstruction error. In the next subsection, we explore how low ℓ_2-reconstruction loss translates to strong performance preservation in language modeling.

2.4. Prefilling and Decoding with Lexico

During the prefilling stage, each layer generates the KV vectors for the input tokens, as illustrated in Figure 1a. Lexico uses full-precision KV vectors for attention computation, which are then passed to subsequent layers. Subsequently, OMP finds the sparse representations of the KV vectors using layer-specific key and value dictionaries, $\boldsymbol{D}_k$ and $\boldsymbol{D}_v$.

The compressed key and value caches are denoted as $\boldsymbol{K}_{\text{csr}}, \boldsymbol{V}_{\text{csr}} \in \mathbb{R}^{l_{\text{seq}} \times N}$ and replace the full-precision KV cache. Their reconstructions become $\hat{\boldsymbol{K}} = \boldsymbol{K}_{\text{csr}}\boldsymbol{D}_k^\top$ and $\hat{\boldsymbol{V}} = \boldsymbol{V}_{\text{csr}}\boldsymbol{D}_v^\top$.

Recall that at the t-th iteration of autoregressive decoding, each layer receives $\boldsymbol{q}_t$, $\boldsymbol{k}_t$, and $\boldsymbol{v}_t$, the query, key, and value vectors corresponding to the newly generated token. Similarly to prior work (Liu et al., 2024b; Kang et al., 2024), we find that keeping a small number of recent tokens in full precision improves the generative performance of the model. To achieve this, we introduce a buffer that temporarily stores recent tokens in an uncompressed state. The KV vectors stored in the buffer are denoted as $\boldsymbol{K}_{\text{buffer}}, \boldsymbol{V}_{\text{buffer}} \in \mathbb{R}^{n_b \times m}$, where n_b is the number of KV vectors in the buffer. The key cache up to, but not including the new token at iteration t, is then reconstructed as follows:

$$\hat{\boldsymbol{K}}_{t-1} = \boldsymbol{K}_{\text{csr}}\boldsymbol{D}_k^\top \parallel \boldsymbol{K}_{\text{buffer}}$$

Substituting this reconstruction into the Equation 1, the attention weights for each head $\boldsymbol{a}_t^{(h)}$ are computed as:

$$\boldsymbol{a}_t^{(h)} = \text{Softmax}\left(\frac{\boldsymbol{q}_t^{(h)}(\boldsymbol{K}_{\text{csr}}^{(h)}\boldsymbol{D}_k^\top \parallel \boldsymbol{K}_{\text{buffer}}^{(h)} \parallel \boldsymbol{k}_t^{(h)})^\top}{\sqrt{m}}\right)$$

A key implementation is that attention for the compressed sparse key cache and the uncompressed key cache is computed separately. For compressed sparse key cache, we first compute the product $\boldsymbol{q}_t^{(h)}\boldsymbol{D}_k$ before we multiply $\boldsymbol{K}_{\text{csr}}$, directly calculating the pre-softmax attention scores for compressed tokens. Attention for the buffer tokens is computed as usual. These scores are then concatenated with softmax to produce the final attention weights (Figure 1b). This process is formalized as:

$$\boldsymbol{a}_t^{(h)} = \text{Softmax}\left(\frac{\boldsymbol{q}_t^{(h)}\boldsymbol{D}_k\boldsymbol{K}_{\text{csr}}^{(h)\top} \mid \boldsymbol{q}_t^{(h)}(\boldsymbol{K}_{\text{buffer}}^{(h)} \parallel \boldsymbol{k}_t^{(h)})^\top}{\sqrt{m}}\right),$$

where $\mid$ represents concatenation along columns for attention scores.

When the buffer reaches capacity, OMP compresses the KV vectors for the oldest n_a tokens in the buffer. This process is independent of the attention computation for the newest token and can therefore be performed in parallel.

Time and space complexity. The sparse representations are stored in CSR format, with values encoded in `FP8(E4M3)`, and all indices, including offsets, are stored as `int16`. Each row in CSR corresponds to a single key or value vector. For a given sparsity level s, the memory usage includes: nonzero values (s bytes), dictionary indices ($2s$ bytes), and the offset array (2 bytes), resulting in a total size of $3s + 2$ bytes. For a head dimension of 128, a fully uncompressed vector using `FP16` takes 256 bytes, yielding a memory usage of $\frac{3s+2}{256} \times 100 \approx 1.17s\%$ (*e.g.*, 37.5% for $s = 32$).

In terms of time complexity, computing $\boldsymbol{q}_t\boldsymbol{K}_t^\top$ for a single head requires $O(l_{\text{seq}}m)$ multiplications. On the other hand, $\boldsymbol{q}_t\boldsymbol{D}_k\boldsymbol{K}_{\text{csr}}^\top$ needs $O(Nm+l_{\text{seq}}s)$ multiplications. This means that our computation is particularly well-suited for long-context tasks when $l_{\text{seq}} > N$ where N is anywhere between 1024 and 4096. For short contexts when $l_{\text{seq}} < N$, our method only adds a small overhead to attention computation in actuality.

3. Experiments

Setup. We evaluate our method on various models (Llama-3-8B, Llama-3.1-8B-Instruct, Llama-3.2-1B-Instruct, Llama-3.2-3B-Instruct, Mistral-7B-Instruct, Qwen2.5-14B-Instruct), using dictionaries trained on WikiText-103, as done in Section 2.3. To assess the effectiveness of Lexico in memory reduction while maintaining long-context understanding, we conduct experiments on selected tasks from LongBench (Bai et al., 2023), following the setup of Liu et al. (2024b). See Table 7 in Appendix B for task details.

Additionally, we evaluate generative performance on complex reasoning tasks, such as GSM8K (Cobbe et al., 2021) with 5-shot prompting and MMLU-Pro Engineering/Law (Wang et al., 2024a) with zero-shot chain-of-thought. We choose these MMLU-Pro subjects since they are deemed the most difficult as they require complex formula derivations or deep understanding of legal knowledge intricacies. We compare our method against two kinds of KV cache compression methods: namely, quantization-based compression and eviction-based compression. For quantization-based methods, we evaluate KIVI (Liu et al., 2024b), ZipCache (He et al., 2024), and the Hugging Face implementation for per-token quantization. For eviction-based methods, we evaluate PyramidKV (Cai et al., 2024) and SnapKV (Li et al., 2024). We refer to the 4-bit and 2-bit versions of KIVI as KIVI-4 and KIVI-2, respectively, and denote its quantization group size as g.

We report KV size as the average percentage of the compressed cache relative to the full cache at generation end. Sparsity s is set to match the KV size of the baseline.

Hyperparameter settings. For both experiments, Lexico uses a dictionary size of $N = 4096$, a buffer size of $n_b = 128$, and an approximation window size $n_a = 1$, compressing the oldest token with each new token generated. For KIVI-4 and KIVI-2, we use a quantization group size of $g = 32$ and a buffer size of $n_b = 128$, as is tested and recommended in Liu et al. (2024b), for LongBench. For GSM8K and MMLU-Pro, we test for stronger memory savings, so we use $g = 64$ and $n_b = 64$ for KIVI.

3.1. Experimental Results

LongBench results. Table 3 presents the performance of Lexico and KIVI on LongBench tasks. Lexico demonstrates better performance than KIVI with similar or even smaller KV sizes. Notably, Lexico enables exploration of extremely low memory regimes that KIVI-2 cannot achieve. At a memory usage of just 12.4% KV size, Lexico maintains reasonable long-context understanding, with only $5.6\%p$ and $4.4\%p$ performance loss on Llama-3.1-8B-Instruct and on Mistral-7B-Instruct-v0.3, respectively, compared to the full cache (`FP16`). The largest performance loss comes from tasks with the lowest full cache accuracy, Qasper, yet there is almost no loss in simpler tasks, such as TriviaQA. This indicates that difficult tasks that require more complex understanding are much more sensitive to performance loss. Hence, it is important to evaluate on GSM8K, one of the harder natural language reasoning tasks, as we do next.

Table 3. **Experimental results on LongBench.** For Lexico, we use $N = 4096$ as the dictionary size and $n_b = 128$ as the buffer size. For KIVI, we use $g = 32$ (group size for quantization) and $n_b = 128$ (buffer size). Sparsity level s is set to match average KV size of KIVI, while $s = 8$ corresponds to cache size unattainable by common 2-bit quantizations. Full cache is in `FP16`.

Method	KV Size	Qasper	QMSum	MultiNews	TREC	TriviaQA	SAMSum	LCC	RepoBench-P	Average
Llama-3.1-8B-Instruct										
Full Cache	100%	22.54	24.57	27.44	72.5	91.65	43.47	63.15	56.76	50.26
KIVI-4	33.2%	**22.83**	23.72	**27.95**	71.0	90.39	**44.25**	**62.93**	55.48	49.78
Lexico $_{s=24}$	30.6%	21.68	**24.25**	27.20	**72.5**	**91.58**	42.93	62.92	**56.51**	**49.95**
KIVI-2	21.1%	13.77	22.72	**27.35**	71.0	90.85	**43.53**	62.03	53.00	48.03
Lexico $_{s=16}$	21.4%	**15.45**	**23.13**	25.78	**72.5**	**92.25**	42.02	**63.01**	**55.58**	**48.71**
Lexico $_{s=8}$	**12.4%**	11.66	21.04	22.35	60.0	91.01	40.30	59.60	51.46	44.68
Mistral-7B-Instruct-v0.3										
Full Cache	100%	41.58	25.69	27.76	76.0	88.59	47.58	59.37	60.60	53.40
KIVI-4	33.2%	40.37	24.51	**27.75**	74.0	88.36	**47.56**	58.49	58.31	52.42
Lexico $_{s=24}$	30.6%	**41.01**	**25.32**	27.51	**76.0**	**88.84**	46.27	**59.98**	**59.44**	**53.05**
KIVI-2	21.1%	38.24	24.08	**26.99**	74.5	88.34	**47.66**	57.51	56.46	51.72
Lexico $_{s=16}$	21.4%	**40.34**	**24.97**	26.36	**76.0**	**89.31**	45.84	**59.31**	**59.50**	**52.70**
Lexico $_{s=8}$	**12.4%**	33.03	22.80	22.85	68.5	87.85	43.10	56.66	56.85	48.96

Table 4. **Experimental results on GSM8K.** For Lexico, we use $N = 4096$ as the dictionary size and $n_b = 128$ as the buffer size. For KIVI, we use $g = 64$ (group size for quantization) and $n_b = 64$ (buffer size). Sparsity level s is set to match the average KV size of KIVI, while $s = 4$ corresponds to cache size unattainable by common 2-bit quantizations. Full cache is in `FP16`. We include example generations of KIVI and Lexico in Appendix C.1.

(a) Llama 3.x 8B Models

Method	KV Size	Llama-3-8B	3.1-8B-Instruct
Full Cache	100%	49.89	79.61
KIVI-4	38.2%	**49.13**	**78.17**
Lexico $_{s=24}$	36.9%	48.29	76.88
KIVI-2	25.7%	40.56	67.93
Lexico $_{s=14}$	26.1%	**48.75**	**75.06**
Lexico $_{s=4}$	**15.8%**	40.03	51.71

(b) Mistral 7B v0.3 Model

Method	KV Size	7B-Instruct
Full Cache	100%	48.60
KIVI-4	38.2%	48.52
Lexico $_{s=20}$	32.7%	**48.60**
KIVI-2	25.7%	42.91
Lexico $_{s=10}$	22.0%	**44.35**
Lexico $_{s=4}$	**15.8%**	39.20

GSM8K results. The performance of Lexico on GSM8K compared to KIVI is shown in Table 4. With a KV size of 36.9%, Lexico on Llama 8B models experiences a slight accuracy drop of less than $3\%p$, underperforming KIVI-4 at a similar KV size. However, in the lower memory regime near 25% KV size, Lexico significantly outperforms KIVI-2, achieving a higher accuracy by $8.2\%p$ on the Llama-3-8B model and $7.1\%p$ on the Llama-3.1-8B-Instruct model. These results highlight the robustness of Lexico in low-memory settings, demonstrating that low reconstruction error can be achieved using only a few atoms from our universal dictionary. To further test the resilience of Lexico, we set the sparsity to $s = 4$, observing a noticeable drop in accuracy on the Llama-3.1-8B-Instruct model. Despite this, both Llama models maintain an accuracy above 40%, which is remarkable given that only 4 atoms from Lexico were used for each key-value vector, utilizing just 15.8% of the full cache, including the buffer.

The performance of Lexico on the Mistral-7B-Instruct model is even more impressive. We demonstrate that for Mistral, Lexico not only outperforms KIVI-4 and KIVI-2 but also achieves higher accuracy with even less memory usage. We also evaluate Lexico with $s = 4$ on the Mistral model and observe an accuracy of 39.2%, further demonstrating robustness in low-memory settings.

Results across model sizes and baselines. We illustrate the trade-off between memory usage and performance across six different KV cache compression methods on Llama models (1B, 3B, and 8B) in Figure 4. For all three model sizes, Lexico consistently lies on the Pareto frontier, achieving higher scores than other compression methods at similar KV cache budget sizes. Notably, Lexico demonstrates greater robustness at smaller model scales,

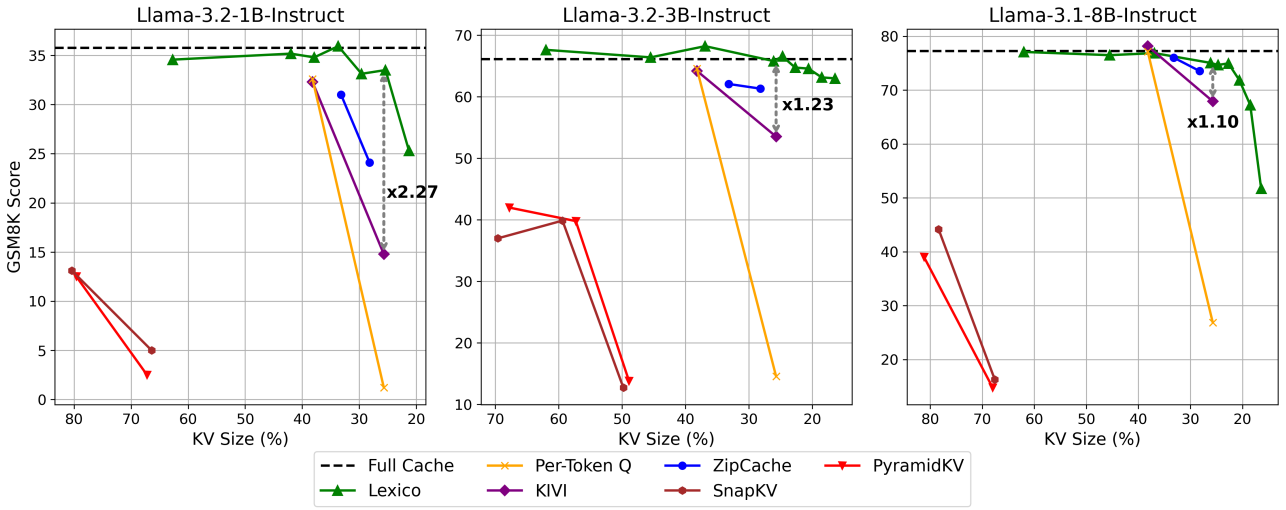


Figure 4. **Memory usage vs. performance of Lexico compared to other KV cache compression methods on GSM8K.** The figure illustrates the relationship between KV cache size and the performance of Lexico on Llama models on GSM8K. For Lexico, we use $N = 4096$ as the dictionary size and $n_b = 128$ as the buffer size. Lexico consistently outperforms both eviction-based methods (SnapKV, PyramidKV) and quantization-based methods (per-token quantization, KIVI, ZipCache).

with larger performance gaps observed for the 1B and 3B models. In the extremely low-memory regime below 20%, where quantization methods such as KIVI and ZipCache cannot achieve feasible cache sizes, Lexico achieves superior performance. Furthermore, while eviction-based methods (SnapKV, PyramidKV) can operate in these extremely low-memory settings, their performance lags significantly behind due to their incompatibility with Grouped Query Attention (GQA), making Lexico the effective choice for stringent memory constraints. We also evaluate Lexico on a larger model, Qwen2.5-14B-Instruct, with its weights quantized to 4 bits, comparing it against quantization methods. The results, illustrated in Figure 5, show that Lexico achieves a higher GSM8K score than KIVI under similar KV cache budgets. Additionally, Lexico enables higher compression ratios than 2-bit quantization methods, facilitating deployment under extreme memory-constrained scenarios.

MMLU-Pro results. Figure 6 illustrates the trade-offs between memory usage and performance for Lexico on the MMLU-Pro Engineering and Law subjects using the Llama-3.1-8B-Instruct model. Lexico outperforms eviction-based methods like SnapKV and PyramidKV across all memory settings, though its performance is comparable to quantization-based methods such as KIVI and ZipCache. However, in a low memory regime below 20% cache, our method still outperforms any other baseline. This highlights that Lexico supports a wide range of compression ratios quite effectively and that our dictionary is generalizable across input distributions.

3.2. Ablation Study

Error thresholding in sparse approximation. Lexico also supports a quality-controlled method for memory saving by allowing early termination of the sparse approximation process when a predefined error threshold is met. This approach conserves memory that would otherwise be used for marginal improvements in approximation quality. Detailed descriptions and results of this ablation study are provided in the Appendix D.1.

Performance without buffer. To evaluate the impact of the buffer, we first conducted experiments with varying sparsity without the buffer, with the results shown by the dashed lines in Figure 7 in Appendix D.2. The comparison shows that removing the buffer results in a more pronounced decline in performance, especially at lower KV sizes.

Table 5. **Balancing memory between buffer and sparse representation.** Performance of Lexico with the Llama-3.1-8B-Instruct model on LongBench tasks while varying the memory allocation between the buffer and the sparse representation, with the total KV cache size fixed at 25% of the original size.

Qasper			MultiNews			TREC		
s	n_b	F1 Score	s	n_b	ROUGE-L	s	n_b	Accuracy
1	862	6.38	1	503	17.20	1	1232	58.5
4	724	8.36	4	423	20.21	4	1035	63.5
8	517	14.58	8	302	21.27	**8**	**739**	**65.0**
12	**278**	**17.84**	**12**	**163**	**22.81**	12	398	63.5
16	0	8.27	16	0	10.70	16	0	54.5

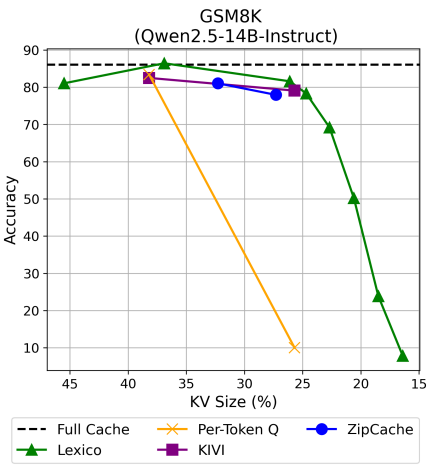


Figure 5. **Memory usage vs. performance of Qwen2.5-14B-Instruct with Lexico on GSM8K.** We compare the performance of Lexico against quantization methods on Qwen2.5-14B-Instruct, with its weights quantized to 4 bits. For Lexico, we use $N = 4096$ as the dictionary size and $n_b = 128$ as the buffer size.

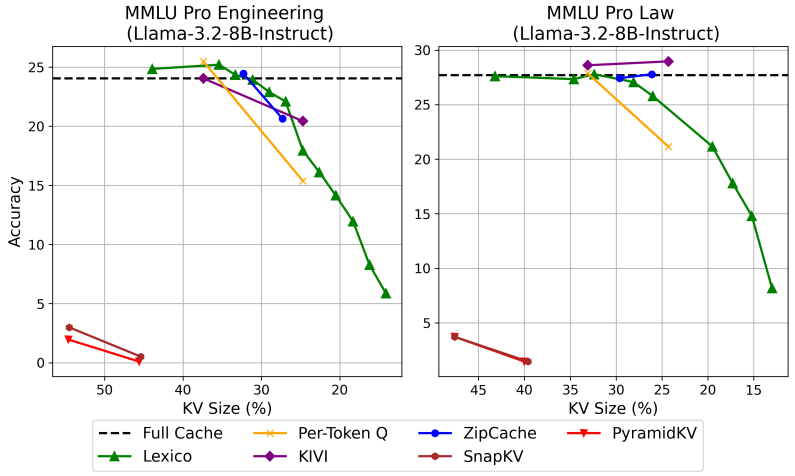


Figure 6. **Memory usage vs. performance of Llama-3.2-8B-Instruct with Lexico on MMLU-Pro Engineering/Law.** For Lexico, we use $N = 4096$ as the dictionary size and $n_b = 128$ as the buffer size. Lexico often outperforms both eviction-based methods (SnapKV, PyramidKV) and quantization-based methods (per-token quantization, KIVI, ZipCache). For Law, our method slightly underperforms around 25%, but in lower memory regimes, our method still outperforms any other baseline.

Balancing memory between buffer and sparse representation. As shown in Table 5, we examine how balancing memory allocation between the buffer and the sparse representation affects performance. By fixing the total KV cache size at 25% of the original, we vary the memory distribution between the buffer and the sparse representation across three LongBench tasks: Qasper, MultiNews, and TREC. The results demonstrate that long-context understanding ability while using Lexico is not solely reliant on the buffer or the sparse representation. Rather, there exist optimal balance points where performance is maximized for each task.

Adaptive dictionary learning. While our universal dictionaries demonstrate strong performance, we explore an adaptive learning method to better incorporate input-specific context. This adaptive approach improves performance by adding new dictionary atoms during generation when the predefined reconstruction error threshold is not met. These atoms, tailored to the input prompt, improve performance but cannot be shared across batches, requiring them to be included in the KV size calculation. While this approach boosts accuracy, it increases memory usage, limiting its ability to achieve low-memory regimes. Detailed methods and results are provided in Appendix D.3.

3.3. Latency Analysis

In this section, we present latency measurements of the forward pass and OMP portion of Lexico during decoding in Table 6. We run simple generation tests on a 1000-token input to the Llama-3.1-8B-Instruct model and generate up to 250 tokens. We compare dictionary sizes of $N = 1024$ and 4096, which primarily affects OMP computation time, while setting the sparsity level $s = 24$ and processing OMP in batches of $n_a = 8$.

Although Table 6 separately lists the forward pass and OMP time, these processes can be parallelized: generating one token takes the maximum of the two durations plus some

Table 6. **Latency measurements.** Time (ms) to compute one new token across 32 layers of Llama-3.1-8B-Instruct. We report the standard full-precision forward pass and Lexico 's forward pass / OMP step for dictionary sizes N=1024 and N=4096 at various sparsity levels s.

Computation type	**Full**	$N = 1024$				$N = 4096$			
		s=4	s=8	s=16	s=24	s=4	s=8	s=16	s=24
Standard forward pass ($\boldsymbol{q}\boldsymbol{K}^\top$)	48.39	–	–	–	–	–	–	–	–
Lexico: forward pass using $\boldsymbol{q}(\boldsymbol{K}_{\text{csr}}\boldsymbol{D}_k^\top)^\top$ and $\boldsymbol{V}_{\text{csr}}\boldsymbol{D}_v^\top$	–	53.90	54.67	55.46	55.56	57.24	56.64	57.13	56.35
Lexico: sparse approximation (OMP)	–	6.03	10.16	18.31	26.57	9.37	15.55	28.37	40.58

overhead. However, parallelization introduces a time–space tradeoff, as running OMP also increases GPU memory usage.

Higher latency may limit Lexico's suitability for latency-critical use cases. Nonetheless, our primary focus is on memory-constrained scenarios where a single GPU can be exceeded with just a batch size of one. By prioritizing memory efficiency, Lexico remains feasible where other methods risk out-of-memory errors, making it valuable for deployment in memory-limited settings.

4. Related Work

Prior work on KV cache optimization spans training-stage and deployment-focused methods. On the deployment side, Kwon et al. (2023) introduces a Paged Attention mechanism and the vLLM framework. While there is a significant and important line of research in this direction (Lin et al., 2024; Qin et al., 2024), this direction is orthogonal to our work and can often be used in tandem with quantization. Post-training techniques focus on three main categories: (1) evicting tokens based on attention scores or heuristics (Ge et al., 2023; Li et al., 2024; Liu et al., 2024a; Devoto et al., 2024; Dong et al., 2024), (2) quantizing KV cache with per-channel or two-dimensional schemes (Hooper et al., 2024; Liu et al., 2024b; Yue et al., 2024; Kang et al., 2024), and (3) merging tokens or memory blocks. These approaches can complement one another, as shown by Liu et al. (2024a) in successfully combining quantization and eviction.

A recent work, QJL, introduces a promising 1-bit quantization method for KV cache compression (Zandieh et al., 2025). However, QJL applies its transformation exclusively to keys (using per-token quantization for values), while Lexico compresses both keys and values, enabling a more comprehensive and memory-efficient solution. Another line of work, PQCache, leverages product quantization by partitioning vectors into sub-blocks and learning centroids for each subspace (Zhang et al., 2024a). Compression is achieved by replacing each sub-block with its nearest centroid. In contrast, Lexico employs sparse coding, representing each vector as a sparse linear combination of dictionary atoms.

5. Conclusion and Limitations

In conclusion, Lexico offers a novel way to compress KV caches by leveraging low-dimensional structures and sparse dictionary learning, revealing substantial redundancy among key cache vectors across diverse inputs. This enables near-lossless compression that often surpasses traditional quantization methods, while offering fine-grained control over memory usage. The universal dictionary remains compact and scalable, making it broadly applicable without increasing memory requirements, and yields strong memory savings especially for long-context tasks without discarding any tokens.

Despite these advantages, Lexico has several limitations. Most notably, as shown in Section 3.3, it does not offer latency improvements. While well-suited for highly memory-constrained scenarios, it may be less effective in high-throughput settings, as latency becomes a key performance bottleneck. Additionally, the OMP algorithm introduces GPU-side memory overhead, which may limit reductions in peak memory usage. These trade-offs suggest that Lexico is best suited for memory-constrained settings where throughput is not the primary bottleneck.

Future directions include further optimizing CSR tensors with customized quantizations, improving the OMP-related latency tradeoffs, and exploring "soft-eviction" strategies that dynamically adjust sparsity based on token importance. Additionally, while our current implementation adopts standard greedy OMP for its efficiency in memory-constrained settings, extending it to a beam search variant–which selects the top-B atoms at each iteration–could further improve reconstruction accuracy, albeit with increased computational overhead. Investigating such trade-offs between approximation quality and efficiency remains an interesting avenue for future work.

Impact Statement

This paper presents a method to efficiently compress KV cache when employing Transformer-based models. Although there are potential societal consequences of our work, there are none which we feel must be specifically highlighted.

References

Bai, Y., Lv, X., Zhang, J., Lyu, H., Tang, J., Huang, Z., Du, Z., Liu, X., Zeng, A., Hou, L., et al. Longbench: A bilingual, multitask benchmark for long context understanding. *arXiv preprint arXiv:2308.14508*, 2023.

Beltagy, I., Peters, M. E., and Cohan, A. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

Bricken, T., Templeton, A., Batson, J., Chen, B., Jermyn, A., Conerly, T., Turner, N., Anil, C., Denison, C., Askell, A., et al. Towards monosemanticity: Decomposing language models with dictionary learning. *Transformer Circuits Thread*, 2, 2023.

Cai, Z., Zhang, Y., Gao, B., Liu, Y., Liu, T., Lu, K., Xiong, W., Dong, Y., Chang, B., Hu, J., et al. Pyramidkv: Dynamic kv cache compression based on pyramidal information funneling. *arXiv preprint arXiv:2406.02069*, 2024.

Candès, E. J., Romberg, J., and Tao, T. Robust uncertainty principles: Exact signal reconstruction from highly incomplete frequency information. *IEEE Transactions on information theory*, 52(2):489–509, 2006.

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Devoto, A., Zhao, Y., Scardapane, S., and Minervini, P. A simple and effective l_2 norm-based strategy for kv cache compression. *arXiv preprint arXiv:2406.11430*, 2024.

Dong, H., Yang, X., Zhang, Z., Wang, Z., Chi, Y., and Chen, B. Get more with less: Synthesizing recurrence with kv cache compression for efficient llm inference. In *Forty-first International Conference on Machine Learning*, 2024.

Dong, W., Shi, G., Li, X., Ma, Y., and Huang, F. Compressive sensing via nonlocal low-rank regularization. *IEEE transactions on image processing*, 23(8):3618–3632, 2014.

Donoho, D. L. Compressed sensing. *IEEE Transactions on information theory*, 52(4):1289–1306, 2006.

Ge, S., Zhang, Y., Liu, L., Zhang, M., Han, J., and Gao, J. Model tells you what to discard: Adaptive kv cache compression for llms. In *The Twelfth International Conference on Learning Representations*, 2023.

He, Y., Zhang, L., Wu, W., Liu, J., Zhou, H., and Zhuang, B. Zipcache: Accurate and efficient kv cache quantization with salient token identification. *arXiv preprint arXiv:2405.14256*, 2024.

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

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

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

Li, Y., Huang, Y., Yang, B., Venkitesh, B., Locatelli, A., Ye, H., Cai, T., Lewis, P., and Chen, D. Snapkv: Llm knows what you are looking for before generation. *arXiv preprint arXiv:2404.14469*, 2024.

Lin, B., Peng, T., Zhang, C., Sun, M., Li, L., Zhao, H., Xiao, W., Xu, Q., Qiu, X., Li, S., et al. Infinite-llm: Efficient llm service for long context with distattention and distributed kvcache. *arXiv preprint arXiv:2401.02669*, 2024.

Liu, Z., Desai, A., Liao, F., Wang, W., Xie, V., Xu, Z., Kyrillidis, A., and Shrivastava, A. Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time. *Advances in Neural Information Processing Systems*, 36, 2024a.

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

Lubonja, A., Præsius, S. K., and Tran, T. D. Efficient batched cpu/gpu implementation of orthogonal matching pursuit for python. *arXiv preprint arXiv:2407.06434*, 2024.

Makhzani, A. and Frey, B. K-sparse autoencoders. *arXiv preprint arXiv:1312.5663*, 2013.

Merity, S. The wikitext long term dependency language modeling dataset. *Salesforce Metamind*, 9, 2016.

Metzler, C. A., Maleki, A., and Baraniuk, R. G. From denoising to compressed sensing. *IEEE Transactions on Information Theory*, 62(9):5117–5144, 2016.

Qin, R., Li, Z., He, W., Zhang, M., Wu, Y., Zheng, W., and Xu, X. Mooncake: A kvcache-centric disaggregated architecture for llm serving, 2024. *arXiv preprint arxiv:2407.00079*, 2024.

Singhania, P., Singh, S., He, S., Feizi, S., and Bhatele, A. Loki: Low-rank keys for efficient sparse attention. *arXiv preprint arXiv:2406.02542*, 2024.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017*, pp. 5998–6008, 2017.

Wang, Y., Ma, X., Zhang, G., Ni, Y., Chandra, A., Guo, S., Ren, W., Arulraj, A., He, X., Jiang, Z., et al. Mmlu-pro: A more robust and challenging multi-task language understanding benchmark. *arXiv preprint arXiv:2406.01574*, 2024a.

Wang, Z., Jin, B., Yu, Z., and Zhang, M. Model tells you where to merge: Adaptive kv cache merging for llms on long-context tasks. *arXiv preprint arXiv:2407.08454*, 2024b.

Xiao, G., Tian, Y., Chen, B., Han, S., and Lewis, M. Efficient streaming language models with attention sinks. In *The Twelfth International Conference on Learning Representations*, 2023.

Ye, L., Tao, Z., Huang, Y., and Li, Y. Chunkattention: Efficient self-attention with prefix-aware kv cache and two-phase partition. *arXiv preprint arXiv:2402.15220*, 2024.

Yu, G.-I., Jeong, J. S., Kim, G.-W., Kim, S., and Chun, B.-G. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, 2022.

Yu, H., Yang, Z., Li, S., Li, Y., and Wu, J. Effectively compress kv heads for llm. *arXiv preprint arXiv:2406.07056*, 2024.

Yue, Y., Yuan, Z., Duanmu, H., Zhou, S., Wu, J., and Nie, L. Wkvquant: Quantizing weight and key/value cache for large language models gains more. *arXiv preprint arXiv:2402.12065*, 2024.

Zandieh, A., Daliri, M., and Han, I. Qjl: 1-bit quantized jl transform for kv cache quantization with zero overhead. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 25805–25813, 2025.

Zhang, H., Ji, X., Chen, Y., Fu, F., Miao, X., Nie, X., Chen, W., and Cui, B. Pqcache: Product quantization-based kvcache for long context llm inference. *arXiv preprint arXiv:2407.12820*, 2024a.

Zhang, Z., Sheng, Y., Zhou, T., Chen, T., Zheng, L., Cai, R., Song, Z., Tian, Y., Ré, C., Barrett, C., et al. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems*, 36, 2024b.

Zhu, H., Chen, W., and Wu, Y. Efficient implementations for orthogonal matching pursuit. *Electronics*, 9(9):1507, 2020.

Appendix

A. Implementation Details

Algorithm 1 illustrates a naive implementation of OMP for understanding. In Lexico, we adopt the implementation of OMP v0 proposed by (Zhu et al., 2020), which minimizes computational complexity using efficient inverse Cholesky factorization. Additionally, we integrate methods from (Lubonja et al., 2024) for batched GPU execution and extend the implementation to handle multiple dictionaries in parallel.

Algorithm 1 OMP

Require: Signal $\boldsymbol{k} \in \mathbb{R}^m$, dictionary $\boldsymbol{D} \in \mathbb{R}^{m \times N}$, sparsity s
Ensure: Sparse representation $\boldsymbol{y} \in \mathbb{R}^n$
Initialize $\boldsymbol{r}^{(0)} \leftarrow \boldsymbol{k}$, $\mathbb{I}^{(0)} \leftarrow \emptyset$, $\boldsymbol{y}^{(0)} \leftarrow \boldsymbol{0}$
for $i = 1$ to s **do**
  $n^{(i)} \leftarrow \arg\max_{1 \le n \le N} \left\{ \left| \left(\boldsymbol{D}^\top \left(\boldsymbol{k} - \boldsymbol{D}\boldsymbol{y}^{(i)} \right) \right)_n \right| \right\}$
  $\mathbb{I}^{(i)} \leftarrow \mathbb{I}^{(i-1)} \cup \left\{ n^{(i)} \right\}$
  $\boldsymbol{y}^{(i+1)} \leftarrow \arg\min_{\boldsymbol{y} \in \mathbb{R}^N} \left\{ \left\| \boldsymbol{k} - \boldsymbol{D}\boldsymbol{y} \right\|_2, \text{Supp}\left(\boldsymbol{y}\right) \subset \mathbb{I}^{(i)} \right\}$
end for
return $\boldsymbol{y}$

Algorithm 2 Prefilling and decoding with Lexico

Parameter: sparsity s, buffer length n_b, approximation length n_a

procedure PREFILLING
  Input: $\boldsymbol{X} \in \mathbb{R}^{l_{\text{seq}} \times d}$
  $\boldsymbol{K} \leftarrow \boldsymbol{X}\boldsymbol{W}_k$, $\boldsymbol{V} \leftarrow \boldsymbol{X}\boldsymbol{W}_v$
  $\boldsymbol{K}_{\text{csr}} \leftarrow \text{OMP}\left(\boldsymbol{K}\left[: l_{\text{seq}} - n_b\right], \boldsymbol{D}_k, s\right)$
  $\boldsymbol{V}_{\text{csr}} \leftarrow \text{OMP}\left(\boldsymbol{V}\left[: l_{\text{seq}} - n_b\right], \boldsymbol{D}_v, s\right)$
  $\boldsymbol{K}_{\text{buffer}} \leftarrow \boldsymbol{K}\left[l_{\text{seq}} - n_b :\right]$, $\boldsymbol{V}_{\text{buffer}} \leftarrow \boldsymbol{V}\left[l_{\text{seq}} - n_b :\right]$
  KV cache $\leftarrow \boldsymbol{K}_{\text{csr}}, \boldsymbol{K}_{\text{buffer}}, \boldsymbol{V}_{\text{csr}}, \boldsymbol{V}_{\text{buffer}}$
  return $\boldsymbol{K}, \boldsymbol{V}$
end procedure

procedure DECODING
  Input: KV cache, $\boldsymbol{x}_t \in \mathbb{R}^{1 \times d}$
  $\boldsymbol{q}_t \leftarrow \boldsymbol{x}_t\boldsymbol{W}_q$, $\boldsymbol{k}_t \leftarrow \boldsymbol{x}_t\boldsymbol{W}_k$, $\boldsymbol{v}_t \leftarrow \boldsymbol{x}_t\boldsymbol{W}_v$
  $\boldsymbol{K}_{\text{csr}}, \boldsymbol{K}_{\text{buffer}}, \boldsymbol{V}_{\text{csr}}, \boldsymbol{V}_{\text{buffer}} \leftarrow$ KV cache
  $\boldsymbol{K}_{\text{buffer}} \leftarrow \text{Concat}\left(\left[\boldsymbol{K}_{\text{buffer}}, \boldsymbol{k}_t\right], \text{dim} = \text{token}\right)$
  $\boldsymbol{V}_{\text{buffer}} \leftarrow \text{Concat}\left(\left[\boldsymbol{V}_{\text{buffer}}, \boldsymbol{v}_t\right], \text{dim} = \text{token}\right)$
  $\boldsymbol{a}_t \leftarrow \text{Concat}\left(\left[\boldsymbol{q}_t\boldsymbol{D}_k\boldsymbol{K}_{\text{csr}}, \boldsymbol{q}_t\boldsymbol{K}_{\text{buffer}}\right], \text{dim} = \text{token}\right)$
  $\boldsymbol{a}_t \leftarrow \text{Softmax}\left(\boldsymbol{a}_t\right)$
  $\boldsymbol{V} \leftarrow \text{Concat}\left(\left[\boldsymbol{D}_v\boldsymbol{V}_{\text{csr}}, \boldsymbol{V}_{\text{buffer}}\right], \text{dim} = \text{token}\right)$
  $\boldsymbol{o}_t \leftarrow \boldsymbol{a}_t\boldsymbol{V}$
  if $\text{len}\left(\boldsymbol{K}_{\text{buffer}}\right) > n_b$ **then**
    $\boldsymbol{K'}_{\text{csr}} \leftarrow \text{OMP}\left(\boldsymbol{K}_{\text{buffer}}\left[: n_a\right], \boldsymbol{D}_k, s\right)$
    $\boldsymbol{V'}_{\text{csr}} \leftarrow \text{OMP}\left(\boldsymbol{V}_{\text{buffer}}\left[: n_a\right], \boldsymbol{D}_v, s\right)$
    $\boldsymbol{K}_{\text{csr}} \leftarrow \text{Concat}\left(\left[\boldsymbol{K}_{\text{csr}}, \boldsymbol{K'}_{\text{csr}}\right], \text{dim} = \text{token}\right)$
    $\boldsymbol{V}_{\text{csr}} \leftarrow \text{Concat}\left(\left[\boldsymbol{V}_{\text{csr}}, \boldsymbol{V'}_{\text{csr}}\right], \text{dim} = \text{token}\right)$
    $\boldsymbol{K}_{\text{buffer}} \leftarrow \boldsymbol{K}_{\text{buffer}}\left[n_a :\right]$, $\boldsymbol{V}_{\text{buffer}} \leftarrow \boldsymbol{V}_{\text{buffer}}\left[n_a :\right]$
  end if
  KV cache $\leftarrow \boldsymbol{K}_{\text{csr}}, \boldsymbol{K}_{\text{buffer}}, \boldsymbol{V}_{\text{csr}}, \boldsymbol{V}_{\text{buffer}}$
  return $\boldsymbol{o}_t$
end procedure

B. LongBench Task Statistics

Table 7. **Details of LongBench tasks used in experiments.**

Task	Task Type	Evaluation Metric	Average Length	# of Samples
Qasper	Single-doc QA	F1	3619	200
QMSum	Summarization	ROUGE-L	10614	200
MultiNews	Summarization	ROUGE-L	2113	200
TREC	Few-shot information retrieval	Accuracy	5177	200
TriviaQA	Few-shot reading comprehension	F1	8209	200
SAMSum	Few-shot dialogue summarization	ROUGE-L	6258	200
LCC	Code completion	Edit Similarity	1235	500
RepoBench-P	Code completion	Edit Similarity	4206	500

C. Qualitative Results

C.1. GSM8K Example Generation

The boxes highlight the correct intermediate answers that must be derived. The highlighted colors in red show incorrect reasoning and answers. We provide the original problem and solution below, along with outputs from Llama-3.1-8B-Instruct with full KV cache (FP16), KV-Dict (sparsity $s = 14$), and KIVI-2.

Original Problem & Solution

Problem: The Kennel house keeps 3 German Shepherds and 2 Bulldogs. If a German Shepherd consumes 5 kilograms of dog food and a bulldog consumes 3 kilograms of dog food per day, how many kilograms of dog food will they need in a week?

Solution: The German Shepherds consume $3 \times 5 = \boxed{15}$ kilograms of dog food per day.
The Bulldogs consume $2 \times 3 = \boxed{6}$ kilograms of dog food per day.
The dogs consume a total of $15 + 6 = \boxed{21}$ kilograms of dog food per day.
Therefore, they need $21 \times 7 = \boxed{147}$ kilograms of dog food in a week.

Final Answer: 147

Llama-3.1-8B-Instruct with Full Cache

First, find the total amount of food the German Shepherds consume in a day: 3 dogs $\times$ 5 kg/dog $=$ 15 kg.
Then find the total amount of food the Bulldogs consume in a day: 2 dogs $\times$ 3 kg/dog $=$ 6 kg.
Add the two amounts together to find the total amount of food the dogs consume in a day: 15 kg $+$6 kg $=$ 21 kg.
Multiply that number by 7 to find the total amount of food the dogs consume in a week: 21 kg/day $\times$7 days $=$ 147 kg.

Llama-3.1-8B-Instruct with KV-Dict ($s = 14$)

The kennel has 3 German Shepherds and 2 Bulldogs, so they have $3 + 2 = 5$ dogs.
The German Shepherds consume $5 \times 3 = 15$ kilograms of dog food per day.
The Bulldogs consume $2 \times 3 = 6$ kilograms of dog food per day.
So, the kennel needs $15 + 6 = 21$ kilograms of dog food per day.
In a week, they will need $21 \times 7 = 147$ kilograms of dog food.
The answer is 147.

Llama-3.1-8B-Instruct with KIVI-2

The Kennel house has 3 German Shepherds and 2 Bulldogs, so they have a total of $3 + 2 = \boxed{5}$ dogs.
Each German Shepherd consumes 5 kilograms of dog food per day, so the total amount of dog food consumed by the German Shepherds is $5 \times 5 = \boxed{25}$ kilograms per day.
Each Bulldog consumes 3 kilograms of dog food per day, so the total amount of dog food consumed by the Bulldogs is $2 \times 3 = \boxed{6}$ kilograms per day.
The total amount of dog food consumed per day is $25 + 6 = \boxed{31}$ kilograms.
The Kennel house will need $31 \times 7 = \boxed{217}$ kilograms of dog food in a week.

D. Ablation Study: Experimental Results

D.1. Error Thresholding in Sparse Approximation

For the error thresholding ablation study, detailed results are provided in Table 8. We set a maximum sparsity of 32, corresponding to the maximum number of iterations for the OMP algorithm. However, if the reconstruction error at any iteration falls below a predefined error threshold, we let the OMP terminate early, saving memory that would otherwise be used for minor approximation improvements. This approach is particularly compatible with OMP, as its greedy nature ensures that early termination yields the same results as using higher sparsity (less non-zero elements). Additionally, OMP inherently computes the residual at each iteration, allowing for continuous evaluation of the relative reconstruction error without requiring any additional computation.

Table 8. **Impact of error thresholding on LongBench performance and memory usage.** The table presents the performance of Lexico on the Llama-3.1-8B-Instruct model at various reconstruction error thresholds (δ) for early termination of the sparse approximation algorithm. A dictionary size of $N = 1024$ and `FP16` precision for the values of the CSR tensors are used.

Threshold (δ)	KV Size	Qasper	QMSum	MultiNews	TREC	TriviaQA	SAMSum	LCC	RepoBench-P	Average
					Llama-3.1-8B-Instruct					
Full Cache	100%	22.54	24.57	27.44	72.5	91.65	43.47	63.15	56.76	50.26
0.2	50.6%	20.03	23.65	26.44	72.5	91.61	43.47	62.72	56.63	49.63
0.3	41.1%	16.49	23.35	25.34	72.5	91.34	43.02	62.53	56.65	48.90
0.4	30.9%	16.08	22.91	23.77	69.5	90.79	42.70	61.28	54.82	47.73
0.5	22.8%	12.43	21.75	21.29	57.5	88.56	41.04	58.85	53.19	44.33

D.2. Performance without Buffer

In this section, we assess the effect of the buffer by comparing results with and without its use. The results for LongBench and GSM8K are presented in Table 9 and Table 10, respectively.

D.3. Adaptive Lexico

While we observe some degree of universality in our dictionaries, as shown in Table 2, their performance is particularly strong on WikiText-103, the dataset they were trained on. To better incorporate input context information, we propose an extension that adaptively learns the dictionary during generation. In this framework, we begin with a pre-trained universal dictionary as the initial dictionary. If, during the generation process, the sparse approximation fails to meet the predefined relative reconstruction error threshold, the problematic uncompressed key or value vector is normalized and added to the dictionary. The sparse representation of this vector is then stored with a sparsity of $s = 1$, where its index corresponds to the newly added atom and its value is the ℓ_2 norm of the uncompressed vector. The updated dictionary is subsequently used for further sparse approximations during the generation task. In this way, the adaptive learning framework incrementally refines the dictionaries, tailoring them to the specific generative task and enhancing overall performance at the cost of additional memory usage.

In our experiment, we use a universal dictionary of size 1024, allowing up to 1024 additional atoms to be added during generation. The maximum sparsity of 16 is used, with a buffer size of $n_b = 128$, and `FP16` precision for the values of the CSR tensors. Results of this experiment are presented in

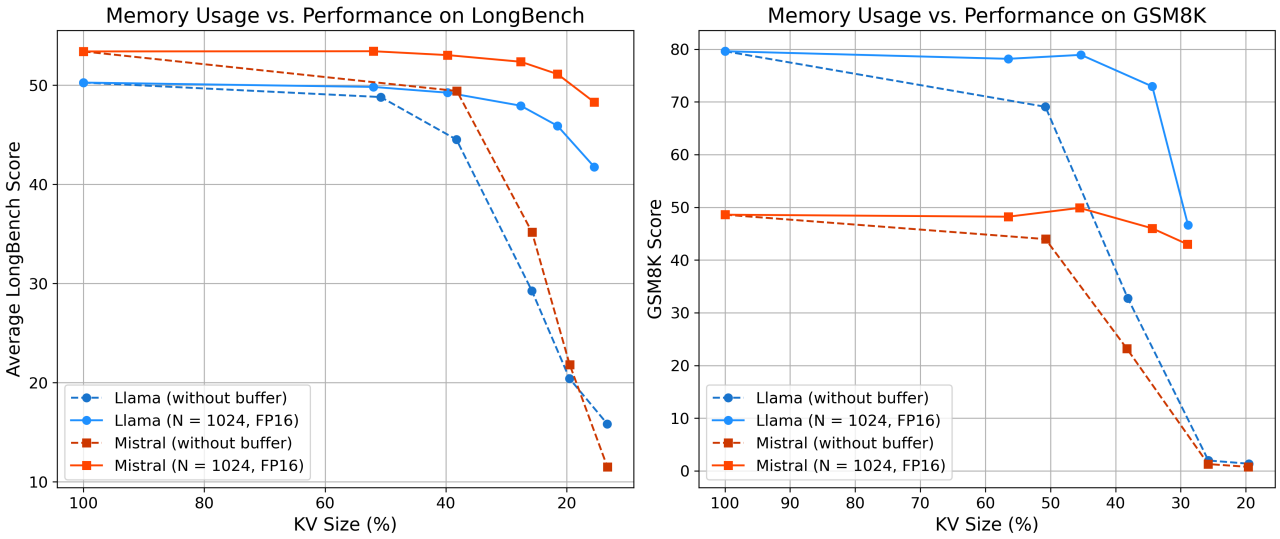


Figure 7. **Memory usage vs. performance of Lexico with and without buffer on LongBench and GSM8K.** The figure illustrates the impact of removing the buffer on the performance of Lexico when evaluated on the Llama 3.1-8B-Instruct and Mistral-7B-Instruct models for LongBench (left) and GSM8K (right) tasks. Solid lines represent configurations with a buffer, while dashed lines represent configurations without a buffer. We use a dictionary size of $N = 1024$ and `FP16` precision for the values of CSR tensors to vary sparsity and explore a wide range of KV sizes.

Table 9. **LongBench performance without buffer.** This table shows the impact of removing the buffer of Lexico on the performance of the Llama-3.1-8B-Instruct and Mistral-7B-Instruct models at varying sparsity levels. A dictionary size of $N = 1024$ and `FP16` precision for the values of the CSR tensors are used.

Sparsity	KV Size	Qasper	QMSum	MultiNews	TREC	TriviaQA	SAMSum	LCC	RepoBench-P	Average
Llama-3.1-8B-Instruct										
Full Cache	100%	13.10	23.46	26.94	72.5	91.65	43.47	63.15	56.76	48.88
$s = 32$	50.8%	14.87	26.51	26.57	71.5	92.48	42.88	61.54	54.04	48.80
$s = 24$	38.2%	13.37	25.02	22.54	65.0	91.75	39.71	52.21	46.48	44.51
$s = 16$	25.8%	8.27	13.74	10.70	54.5	77.51	20.45	26.53	22.46	29.27
$s = 12$	19.5%	6.31	10.15	5.66	39.0	53.70	6.83	22.18	19.46	20.41
$s = 8$	13.3%	2.74	8.05	4.17	36.5	34.45	4.27	18.24	18.32	15.84
Mistral-7B-Instruct-v0.3										
Full Cache	100%	41.58	25.69	27.76	76.0	88.59	47.58	59.37	60.60	53.40
$s = 32$	50.8%	40.27	25.21	27.53	76.5	89.01	45.77	58.64	59.07	52.75
$s = 24$	38.2%	37.46	24.41	27.34	75.5	88.66	43.87	48.55	49.50	49.41
$s = 16$	25.8%	25.57	18.49	15.19	71.5	81.91	27.90	19.39	21.45	35.18
$s = 12$	19.5%	18.59	13.11	5.95	58.0	50.13	2.86	13.38	12.60	21.83
$s = 8$	13.3%	10.32	6.98	2.67	31.5	20.01	2.27	10.18	8.11	11.51

Table 10. **GSM8K performance without buffer.** This table shows the impact of removing the buffer of Lexico on the performance of the Llama-3.1-8B-Instruct and Mistral-7B-Instruct models at varying sparsity levels. A dictionary size of $N = 1024$ and `FP16` precision for the values of the CSR tensors are used.

Sparsity	KV Size	Llama-3.1-8B-Instruct	Mistral-7B-Instruct-v0.3
Full Cache	100%	79.61	48.60
$s = 32$	50.8%	69.07	43.97
$s = 24$	38.2%	32.75	23.20
$s = 16$	25.8%	1.97	1.29
$s = 12$	19.6%	1.36	0.76

Table 11. **GSM8K performance of adaptive Lexico.** The table shows the GSM8K performance and KV cache sizes of adaptive Lexico on the Llama-3.1-8B-Instruct and Mistral-7B-Instruct-v0.3 models at varying reconstruction error thresholds (δ). A universal dictionary of size 1024 is used, with up to 1024 additional atoms added during generation. The maximum sparsity of $s = 16$, buffer size of $n_b = 128$, and `FP16` precision for CSR tensor values are applied.

Threshold (δ)	**Llama-3.1-8B-Instruct**		**Mistral-7B-Instruct-v0.3**	
	KV Size	**GSM8K Score**	**KV Size**	**GSM8K Score**
Full Cache	100%	79.61	100%	48.60
0.25	N/A	N/A	42.1%	48.07
0.30	43.5%	77.41	41.3%	48.14
0.35	42.0%	76.80	39.8%	47.76