LSH Tells You What To Discard: An Adaptive Locality-Sensitive Strategy for KV Cache Compression

Anonymous Author(s)
Affiliation
Address
email

Abstract

Transformer-based large language models (LLMs) use the key-value (KV) cache to significantly accelerate inference by storing the key and value embeddings of past tokens. However, this cache consumes significant GPU memory. In this work, we introduce LSH-E: an algorithm that uses locality-sensitive hashing (LSH) to compress the KV cache. LSH-E quickly locates tokens in the cache that are cosine dissimilar to the current query token. This is achieved by computing the Hamming distance between binarized Gaussian projections of the current token query and cached token keys, with a projection length much smaller than the embedding dimension. We maintain a lightweight binary structure in GPU memory to facilitate these calculations. At every decoding step, the key and value of the current token replace the embeddings of a token expected to produce the lowest attention score. We demonstrate that LSH-E can compress KV cache by 30%-70% while maintaining high performance across reasoning, multiple-choice, and long-context retrieval tasks.

1 Introduction

The advent of large language models (LLMs) has enabled sharp improvements over innumerable downstream natural language processing (NLP) tasks, such as summarization and dialogue generation [1, 2]. The hallmark feature of LLMs, the attention module [3, 4, 5], enables contextual processing over sequences of tokens. To avoid repeated dot products over key and value embeddings of tokens, a key-value (KV) cache is maintained in VRAM to maintain these calculations. This technique is particularly popular with decoder LLMs.

However, the size of the KV cache scales quadratically with sequence length n and linearly with the number of attention layers and heads. For example, maintaining the KV cache for a sequence of 4K tokens in half-precision (FP16) can require approximately $\sim$16GB of memory for most models within the Llama 3 family [6]. These memory costs are exacerbated with batched inference and result in high decoding latency [7]. Consequently, there is significant interest in compressing the size of the KV cache to enable longer context windows and low-resource, on-device deployment.

An emerging strategy for reducing the size of the KV cache is *token eviction*. This approach drops the key and value embeddings for past tokens in the cache, skipping future attention calculations involving these tokens. Various token eviction/retention policies have been explored in recent literature, including the profiling of token type preferences [8], retention of heavy-hitter tokens [9, 10], and dropping tokens based on the high L_2 norms of their key embeddings [11]. The latter approach [11] is intriguing as eviction decisions are performed pre-attention. However, this L_2 dropout strategy only performs well on long-context retrieval tasks. It is specialized to retain only those tokens with the highest attention, which we find unsuitable for free-form reasoning tasks. Existing literature suggests that retaining tokens with a diverse spectrum of attention scores (skewing high) is necessary [12, 9, 13].

Is there a non-attentive KV cache compression strategy that is performant over a wide variety of tasks? This work answers this question positively by introducing a novel strategy, LSH-E, that *dynamically* determines token eviction pre-attention via locality-sensitive hashing (LSH) [14, 15]. LSH-E evicts a past token from the cache whose key embedding is highly cosine dissimilar to the current query token embedding. The intuition behind this strategy is that high cosine dissimilarity indicates a low dot-product attention score. To efficiently scan for cosine (dis)similar tokens without performing attention, LSH-E leverages the SimHash [15, 14] to instead compare Hamming distances between c-length binary hashes of cached key embeddings and the current query embedding.

LSH-E requires minimal overhead: for a total sequence length of ℓ tokens with embedding dimension d, LSH-E maintains a constant-window-size (k), low-cost binary array (with compressed embedding dimension c) in GPU memory of size $c \times k$ bytes, where $c \ll d$ and $k \ll \ell$. Cached tokens with key embeddings that register low Hamming similarity measurements to decoded query embeddings are gradually replaced.

Our contributions are as follows:

- We introduce a novel *attention-free* token eviction strategy, LSH-E, that leverages locality-sensitive hashing (LSH) to quickly locate which token in the cache is the least relevant to the current query. This ranking procedure consists entirely of cheap Hamming distance calculations. The associated binary array for computing these similarities requires minimal memory overhead.
- **Novel Attention-Free Token Eviction:** For a Llama 3 model, LSH-E can compress the KV cache by 30%-70% with minimal performance drop. LSH-E demonstrates high performance on reasoning tasks (GSM8K free-form [16], MedQA free-form [16]), long-context retrieval (Needle-in-a-Haystack, Common Word task, Ruler QA [17]), and multiple-choice (GSM8K MC, MedQA MC).
- **State-of-the-Art Performance:** To the best of our knowledge, LSH-E achieves state-of-the-art performance for attention-free eviction across a wide variety of tasks. LSH-E outperforms L_2 eviction in high-compression regimes over free-form reasoning and MC tasks, while performing comparably in long-context retrieval tasks specifically designed for the L_2 eviction method to perform well on.
- **Open-Source Implementation:** Upon public release of our manuscript, we will release an open-source implementation of LSH-E through a fork of the popular cold-compress library (`https://github.com/AnswerDotAI/cold-compress`).

2 Preliminaries

In this section, we review technical concepts crucial to attention and locality-sensitive hashing. We assume some base level of similarity with transformers, but for precise mathematical formalism, we refer the reader to [18].

Scaled Dot-Product Attention. Consider a sequence of n tokens with e-dimensional real-valued representations $x_1, x_2, \ldots, x_n$. Let $Q = [q_1\, q_2 \cdots q_n] \in \mathbb{R}^{n \times d}$, $K = [k_1\, k_2 \cdots k_n] \in \mathbb{R}^{d \times n}$ where $q_i = W_q x_i$, $k_i = W_k x_i$ and $W, K \in \mathbb{R}^{d \times e}$. The query and key projectors W_q and W_k are pre-trained weight matrices. We also define a value matrix $V = [v_1\, v_2\, v_2 \cdots v_n] \in \mathbb{R}^{d_{out} \times n}$ with $v_i = W_v x_i$ with trainable $V \in \mathbb{R}^{d_{out} \times d}$, the scaled dot-product attention mechanism is given as

$$\text{Attention}(Q, K, V) = V \cdot \text{softmax}\Big(\frac{Q^\top K}{\sqrt{d}}\Big). \tag{1}$$

Typically, attention layers contain multiple heads $\{h_i\}_{i=1}^{J}$ each with distinct query, key, and value projectors $\{W_q^{(h_i)}, W_k^{(h_i)}, W_v^{(h_i)}\}_{i=1}^{J}$. In a multi-head setup, attention is computed in parallel across all heads, and the outputs are concatenated together and then passed through a linear layer for processing by the next transformer block. As Q, K, V are updated with each new incoming token, to avoid significant re-computation, the current state of $Q^\top K$, Q, and K are maintained in the KV cache. Our goal is to bypass attention computation and caching for select tokens, i.e., sparsify the attention matrix $Q^\top K$, K, and V.

Locality-sensitive hashing. We will now describe a family of locality-sensitive hashing (LSH) functions able to efficiently approximate nearest neighbors (per cosine similarity) of key/query vectors in high-dimensional $\mathbb{R}^d$ through comparison in a reduced c-dimensional space (per Hamming distance) with $c \ll d$ [19, 15]. Formally for our setup, $dist_d(x, y) \triangleq \cos\theta_{x,y} = \frac{x^\top y}{||x||\,||y||}$ and $dist_c(p, q) \triangleq \mathrm{d}_H(p, q)$ which denotes the Hamming distance. We will project each vector from $\mathbb{R}^d$ into $\mathbb{Z}_2^c$, the space of c-bit binary strings (which is often referred to as a *binary hash code*). To acquire a c-bit long hash code from an input vector $x \in \mathbb{R}^d$, we define a random projection matrix $R \in \mathbb{R}^{c\times d}$ with iid entries $\sim \mathcal{N}(0, 1)$. We then define $h(x) = \mathrm{sgn}(Rx)$, where $\mathrm{sgn}(\cdot)$ (as an abuse of conventional notation) is the element-wise Heaviside step function ($x \geq 0 = 1$, $x < 0 = 0$). For two unit vectors $x, y \in \mathbb{R}^d$ we have

$$\frac{1}{c} \cdot \mathbb{E}[\mathrm{d}_H\big(h(x), h(y)\big)] = \frac{\theta_{x,y}}{\pi}, \tag{2}$$

where $\theta_{x,y} = \arccos(\cos(\theta_{x,y}))$. We do not prove equation 2 in this work; see Theorem §3.1 in [14, Theorem 3.1]. In particular, if x and y are close in angle, the Hamming distance between $h(x)$ and $h(x)$ is low in expectation. Increasing the hash dimension c reduces variance.

3 LSH-E: An LSH Eviction Strategy

We will now formalize our eviction method. We assume that the KV cache has a limited and fixed budget $\mathcal{C}$ and conceptually divide the KV cache management during LLM inference into two stages: the initial Prompt Encoding Stage and then a Decoding Stage (i.e., textual generation). Unless otherwise noted, d refers to the query/key embedding dimension of tokens.

Policy. Let $S_t \subset [n]$ denote the set of indices of tokens retained in the KV cache at the t-th time step, where n is the current number of seen tokens. We shall momentarily define the eviction policy, which we denote as a function $P : S_{t-1} \to S_t$ subject to $|S_t| \leq C$ for all t, where C is a cache budget. P inserts and evicts embeddings into the key cache $\mathcal{K}$, value cache $\mathcal{V}$ and a binary LSH hash table $\mathcal{H}$ (see Section 2), while maintaining the budget C. For time step t, $\mathcal{K}_t = K_{S_t}$, $\mathcal{V}_t = V_{S_t}$ and $\mathcal{H} = h(\mathcal{K}_t)$, where K and V are the key and value matrices of the attention head, and K_{S_t} and V_{S_t} are the set of key vectors and value vectors indexed by S_t. The function F_{score} assigns a score for each key inside the KV cache.

We define F_{score} as the negative of hamming distances D between the hash code of a query vector q and $\mathcal{H}$: $F_{score}(q, \mathcal{K}) = -d_H(h(q), \mathcal{H})$, which is an array which contains all Hamming distances between q and key codes in $\mathcal{H}$. The eviction index e_t at any step t is selected as $e_t \leftarrow \underset{e\in S_t}{\arg\min}\, F_{score}(q_t, \mathcal{H}_t)$, which is the index of the token whose key code is furthest away from the query code. We define our policy as,

$$P(S_{t-1}) = (S_{t-1} \setminus e_t) \cup t, \tag{3}$$

i.e., we evict e_t from the list of cached indices, and the current token index is inserted.

Prompt Encoding Stage. During the prompt encoding stage, the model processes the prompt, $x_{prompt} = [x_1, ..., x_N] \in \mathbb{R}^{N\times d}$, to compute the initial KV cache $\mathcal{K}_0$ and $\mathcal{V}_0$.

Let C be the maximum number of tokens that can fit within the cache budget. The KV cache is first filled to full by storing the first C tokens, i.e., $\mathcal{S}_0 = [C]$. We calculate and store $\mathcal{H}_0 = h(\mathcal{K}_0) = \bigcup_{i\in S_0} h(k_i)$, i.e., the hashes of all key embeddings for tokens $1 \leq i \leq C$. Then, for each token n with $C < n \leq N$ remaining in the prompt, an entry inside the cache is selected for eviction. This protocol is detailed in Algorithm 1. Steps 3-5 in Algorithm 1 update and clear corresponding columns/rows in the cache; $M(e_n, e_n)$ is a mask which clears row e_n and column e_n from the attention matrix.

Algorithm 1 LSH-E (timestep t)

Require: key cache $\mathcal{K}_{t-1}$, key matrix K (k_t added), value cache $\mathcal{V}_{t-1}$, value matrix V (v_t added), cache indices S_{t-1}, query q_t, hash table $\mathcal{H}$, attention matrix $Q^\top K$ ($q_t K$ added)

1: $e_t \leftarrow \arg\min_{e \in S_{t-1}} F_{score}(q_t, \mathcal{K}_{t-1})$
2: $S_t \leftarrow P(S_{t-1})$ ▷ Apply the policy: evict token e_t, add token t
3: **del** $H_{t-1}[e_t]$ ▷ Remove the key hash of token e_t
4: $H_t \leftarrow H_{t-1}$.**add**$(h(k_t))$ ▷ Add key hash of token t
5: $\mathcal{K}_t \leftarrow K_{S_{t-1}}$ ▷ Update key cache
6: $\mathcal{V}_t \leftarrow V_{S_{t-1}}$ ▷ Update value cache
7: $Q^\top K \leftarrow (Q^\top K)_t \odot M(e_t, e_t)$ ▷ Update attention matrix – clear column/row e_t

Decoding Stage. Let $x_{decoding} = [z_1, ...z_T] \in \mathbb{R}^{T \times d}$ be the generated tokens during autoregressive decoding. The generation phase updates the KV cache with each new token generated at time step t by the same process as described in Algorithm 1. (For notational simplicity, set $t \leftarrow N + 1$ to denote the first decoded token after the prompt.)

Complexity Our strategy reduces KV cache memory overhead to constant C and computation overhead to constant per token.

4 Experiments

Tasks We evaluated our LSH eviction strategy across various tasks to demonstrate its effectiveness in reducing the memory cost of the KV cache while preserving the language quality of the generated text. Our experiments are split into three main categories: free-response question answering, multiple choice, and long-context retrieval. Our long context modeling tasks include the multi-key needle-in-a-haystack task and the common words task from [17]. Question answering tasks include GSM8K [16] and MedQA [20].

Metrics The question-answering tasks were evaluated using BERTScore, ROUGE, and GPT4-Judge. We prompt GPT-4 to look at both the model prediction and the ground truth answer, then provide a score from 1 - 5 on the coherence, faithfulness, and helpfulness of the answer in addition to how similar the prediction was to the ground truth. In this section, we report the average of these four scores. For details on individual scores and the prompts given to GPT-4, please refer to Appendix A. For multiple-choice tasks, we report accuracy. The metric used to evaluate long context modeling tasks is the string matching score.

Configuration and Setup We conducted experiments using Llama3 8B-Instruct model [6] at different cache budgets on Nvidia L4 GPUs. We keep the first 4 and the most recent 10 tokens of the prompt in the KV cache at all times. We chose the L_2 norm-based eviction, a similar method that does not depend on the attention score, as our baseline for comparison.

4.1 Free Response Question Answering

We tested each strategy against tasks that require generating accurate answers using multi-step reasoning to assess their potential side effect on the LLM's language quality and reasoning ability given a constrained KV cache budget.

GSM8K GSM8K consists of grade-school-level math problems that typically require multiple reasoning steps. As shown in Figure 1, our LSH eviction strategy consistently outperforms the L_2 norm-based method across various cache sizes. Notably, even when the KV cache budget is set to 50% of the full capacity, the LSH eviction strategy maintains a high answer quality, with minimal degradation in BERTScore F1, ROUGE-L, and GPT4-Judge scores.

MedQA MedQA is a free-form multiple choice question answering dataset collected from professional medical board exams. We use a processed version and sample 100 questions from it. Each question has 5 choices and only one correct answer, along with ground truth explanations and reasoning steps. Figure 2 illustrates that our LSH-eviction method achieves better performance than

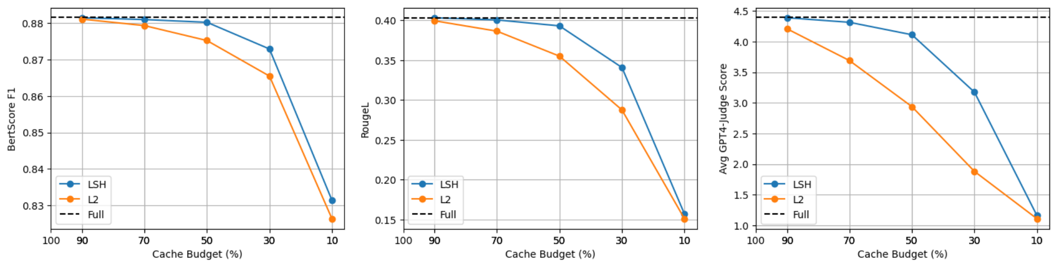


Figure 1: **GSM8K Cache Performance:** as measured by BertScore F1, Rouge-L, and GPT-4-as-a-judge, on GSM8K Free Response Question Answering for cache budgets of 90%, 70%, 50%, 30%, and 10%. LSH outperforms L_2 for all three metrics for every cache budget, with the most significant difference for the 50% and 30% budgets.

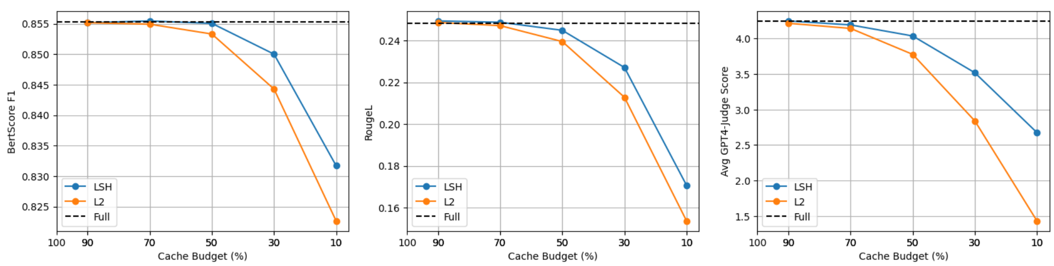


Figure 2: **MedQA Cache Performance:** Performance, as measured by BertScore F1, Rouge-L, and GPT-4-as-a-judge on the MedQA free response question answering dataset for cache budgets of 90, 70, 50, 30, and 10. LSH outperforms L_2 for all three metrics for every cache budget, with a significant difference for the 30% and 10% budgets.

L_2 for every cache budget tested. For both datasets, the LSH method produced more coherent and helpful answers across all cache budgets than L_2.

4.2 Multiple Choice Question Answering

We evaluated our method on multiple-choice versions of GSM8K and MedQA. Multiple choice is a more difficult test of a model's reasoning capability, as it takes away the ability to use intermediate results in the generated text. The model has to keep useful tokens during prompt compression in order to pick the correct answer choice.

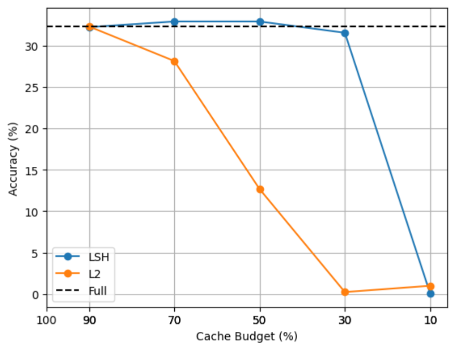
GSM8K Results For the multiple choice experiments, LSH significantly outperforms L_2 for cache budgets of 30% and 50%. As shown in Figure 3a, L_2's accuracy drops significantly at smaller cache sizes, while LSH's performance does not significantly drop until the cache budget is set at 10%.

MedQA Results As per Figure 3b, the MedQA multiple choice experiment, LSH offers better performance than L_2 for all tested cache budgets except 50%.

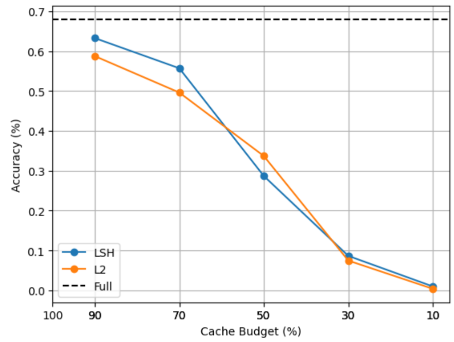
4.3 Long-Context Retrieval

To evaluate LSH's ability to retain and retrieve important pieces of information from long contexts, we used the Needle-in-a-Haystack and Common Words tasks from [17]. These tests are excellent benchmarks to assess the compression method's capability to keep important tokens inside the KV cache and drop the unimportant ones.

Needle-in-a-Haystack In this task, the model must extract specific information buried within a large body of text. As illustrated in Figure 4b, our LSH eviction strategy LSH outperforms L_2 at every cache budget bar 90%, and both methods see a sharp drop in the ability to recall the "needle" after the cache budget drops to 50% and lower. L2SH outperforms L_2 for these smaller cache sizes.

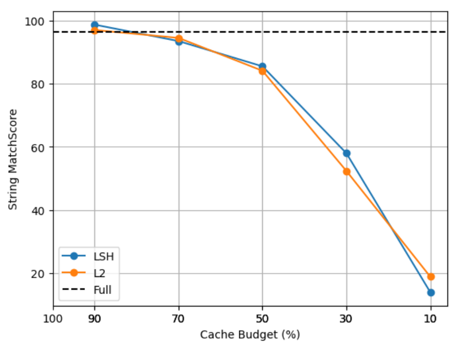


(a) GSM8K MC

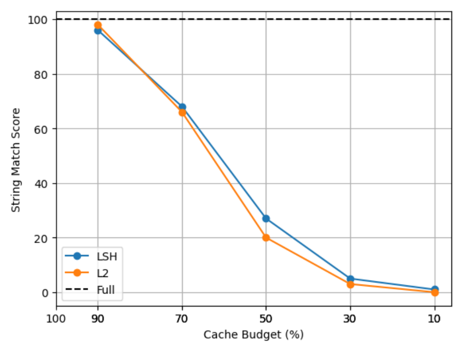


(b) MedQA MC

Figure 3: **Multiple Choice Tasks.** We examine the performance of Llama3.1-8B-Instruct on the multiple-choice versions of GSM8K and MedQA. For GSM-8K observe the LSH-E performs either better or minimally worse than the uncompressed baseline for compression ratios up to 70%, while the performance L_2 eviction strategy rapidly decreases at $> 30\%$ KV cache compression. For MedQA MC, LSH-E is able to maintain high performance for 10% compression, but both methods lose at performance $> 30\%$ compression.



(a) Common Words



(b) Needle-In-A-Haystack

Figure 4: **Long-Context Retrieval.** In these tasks, the LLM is asked to recover a special token hidden amongst a long context of irrelevant information. Although the L_2 strategy is specifically designed for this category of task, we observe that LSH is able to either exceed or closely match the performance of L_2 eviction for nearly all budgets.

Common Words In the Common Words task, the model must identify the most frequent words from a long list. Figure 4a shows LSH performed on par with L_2 in general and slightly better at 30%, 50%, and 90% cache budget. Both methods outperform the full cache model at 90% cache size, indicating that some cache compression can actually increase performance. Neither method experienced a significant drop in performance until the cache budget was reduced to 30%.

5 Conclusion

In this work, we introduce LSH-E, an attention-free token eviction strategy that efficiently reduces the memory footprint of key-value (KV) caches in LLMs. Leveraging locality-sensitive hashing (LSH) to estimate the cosine similarity between current query tokens and past cached tokens, LSH-E dynamically evicts the least relevant tokens using low-cost Hamming distance calculations. Our experiments demonstrate that LSH-E can compress the KV cache by 30%-70% for Llama 3 models with minimal impact on performance across a range of tasks, including free-form reasoning, long-context retrieval, and multiple-choice.

References

[1] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.

[2] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*, 2022.

[3] Dzmitry Bahdanau. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[4] Minh-Thang Luong. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

[5] A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

[6] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[7] Yao Fu. Challenges in deploying long-context transformers: A theoretical peak performance analysis. *arXiv preprint arXiv:2405.08944*, 2024.

[8] Suyu Ge, Yunan Zhang, Liyuan Liu, Minjia Zhang, Jiawei Han, and Jianfeng Gao. Model tells you what to discard: Adaptive kv cache compression for llms. *arXiv preprint arXiv:2310.01801*, 2023.

[9] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

[10] Zhenyu Zhang, Shiwei Liu, Runjin Chen, Bhavya Kailkhura, Beidi Chen, and Atlas Wang. Q-hitter: A better token oracle for efficient llm inference via sparse-quantized kv cache. *Proceedings of Machine Learning and Systems*, 6:381–394, 2024.

[11] Alessio Devoto, Yu Zhao, Simone Scardapane, and Pasquale Minervini. A simple and effective l_2 norm-based strategy for kv cache compression. *arXiv preprint arXiv:2406.11430*, 2024.

[12] Zhiyu Guo, Hidetaka Kamigaito, and Taro Watanabe. Attention score is not all you need for token importance indicator in kv cache reduction: Value also matters. *arXiv preprint arXiv:2406.12335*, 2024.

[13] Sifan Long, Zhen Zhao, Jimin Pi, Shengsheng Wang, and Jingdong Wang. Beyond attentive tokens: Incorporating token importance and diversity for efficient vision transformers. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10334–10343, 2023.

[14] Michel X Goemans and David P Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM (JACM)*, 42(6):1115–1145, 1995.

[15] Moses S Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388, 2002.

[16] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[17] Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, and Boris Ginsburg. Ruler: What's the real context size of your long-context language models? *arXiv preprint arXiv:2404.06654*, 2024.

[18] Mary Phuong and Marcus Hutter. Formal algorithms for transformers. *arXiv preprint arXiv:2207.09238*, 2022.

[19] Alexandr Andoni, Piotr Indyk, and Ilya Razenshteyn. Approximate nearest neighbor search in high dimensions. In *Proceedings of the International Congress of Mathematicians: Rio de Janeiro 2018*, pages 3287–3318. World Scientific, 2018.

[20] Di Jin, Eileen Pan, Nassim Oufattole, Wei-Hung Weng, Hanyi Fang, and Peter Szolovits. What disease does this patient have? a large-scale open domain question answering dataset from medical exams. *Applied Sciences*, 11(14):6421, 2021.

Appendix

A Granular GSM8K and MedQA scores

		GSM8K			MedQA		
Cache Budget (%)	Strategy	Precision	Recall	F1	Precision	Recall	F1
10	L2	0.8585	0.7983	0.8270	0.8330	0.8126	0.8226
	LSH	**0.8602**	**0.8067**	**0.8323**	**0.8570**	**0.8080**	**0.8317**
30	L2	0.8853	0.8487	0.8665	0.8554	0.8336	0.8443
	LSH	**0.8934**	**0.8557**	**0.8740**	**0.8665**	**0.8343**	**0.8500**
50	L2	0.8907	0.8611	0.8756	0.8659	0.8412	0.8533
	LSH	**0.8970**	**0.8652**	**0.8807**	**0.8689**	**0.8417**	**0.8551**
70	L2	0.8946	0.8653	0.8796	0.8679	0.8425	0.8549
	LSH	**0.8964**	**0.8666**	**0.8812**	**0.8687**	**0.8427**	**0.8555**
90	L2	0.8961	0.8665	0.8810	0.8681	0.8427	0.8552
	LSH	**0.8965**	**0.8670**	**0.8814**	**0.8682**	0.8427	0.8552
100	Full	0.8967	0.8672	0.8816	0.8682	0.8428	0.8553

Table 1: GSM8K and MedQA Question Answering BertScores.

		GSM8K				MedQA			
Cache Budget (%)	Strategy	Rouge 1	Rouge 2	Rouge L	Rouge Lsum	Rouge 1	Rouge 2	Rouge L	Rouge Lsum
10	L2	0.1961	0.0494	0.1533	0.1795	0.3043	0.0717	0.1536	0.2885
	LSH	**0.2044**	**0.0510**	**0.1558**	**0.1840**	**0.3457**	**0.1102**	**0.1706**	**0.3242**
30	L2	0.3979	0.1515	0.2924	0.3410	0.4285	0.1461	0.2128	0.4070
	LSH	**0.4529**	**0.1900**	**0.3471**	**0.3882**	**0.4495**	**0.1701**	**0.2271**	**0.4256**
50	L2	0.4800	0.2070	0.3588	0.4109	0.4736	0.1845	0.2395	0.4495
	LSH	**0.5133**	**0.2379**	**0.3972**	**0.4404**	**0.4808**	**0.1935**	**0.2449**	**0.4554**
70	L2	0.5103	0.2337	0.3907	0.4364	0.4837	0.1943	0.2472	0.4580
	LSH	**0.5213**	**0.2424**	**0.4040**	**0.4460**	**0.4871**	**0.1974**	**0.2488**	**0.4611**
90	L2	0.5191	0.2403	0.4014	0.4438	0.4866	0.1966	0.2487	0.4606
	LSH	**0.5224**	**0.2433**	**0.4055**	**0.4465**	**0.4870**	**0.1973**	**0.2494**	**0.4610**
100	Full	0.5239	0.2449	0.4054	0.4474	0.4865	0.1976	0.2484	0.4602

Table 2: GSM8K and MedQA Question Answering Rouge Scores.

		GSM8K				MedQA			
Cache Budget (%)	Strategy	Similar to GT	Coherent	Faithful	Helpful	Similar to GT	Coherent	Faithful	Helpful
10	L2	1.0020	1.3140	1.0940	1.0320	1.1031	1.6955	1.6395	1.2829
	LSH	**1.0360**	**1.4860**	**1.2000**	**1.1100**	**1.9695**	**3.5167**	**2.6650**	**2.5472**
30	L2	1.4300	2.5340	1.9700	1.9820	1.9391	3.6326	2.9420	2.8428
	LSH	**2.6920**	**3.8880**	**3.3900**	**3.3680**	**2.5108**	**4.4145**	**3.5334**	**3.6130**
50	L2	2.3060	3.5760	3.1420	3.1120	2.8497	4.5108	3.7967	3.9499
	LSH	**3.5660**	**4.5780**	**4.2880**	**4.3040**	**3.0216**	**4.7299**	**4.1385**	**4.2544**
70	L2	3.1200	4.2660	3.9520	3.9420	3.1945	4.7554	4.2348	4.3851
	LSH	**3.8400**	**4.6960**	**4.4540**	**4.4820**	**3.2318**	**4.8094**	**4.2917**	**4.4342**
90	L2	3.6060	4.5660	4.3140	4.3560	3.2652	4.8183	4.3183	4.4578
	LSH	**3.9120**	**4.7240**	**4.5200**	**4.5420**	**3.2908**	**4.8389**	**4.3546**	**4.5069**
100	Full	3.9240	4.7340	4.5760	4.5980	3.3369	4.8173	4.3418	4.5000

Table 3: GSM8K and MedQA Question Answering ChatGPT as a Judge

B Memory Usage

Table 4 below compares the memory usage of KV cache and relevant data structures of L_2 and LSH on the GSM8K and MedQA question answering experiments. LSH maintains $hash(\mathcal{K})$, a binary hash matrix of the attention keys, in memory and, therefore, has slightly higher memory usage than L2. Our implementation uses 8-bits for binary values instead of 1-bit. Using 1-bit binary numbers will reduce the memory overhead of LSH by a factor of 8 and narrow the difference in memory usage between LSH and L2.

		GSM8K		MedQA	
Cache Budget (%)	Strategy	Compression Ratio	Cache Memory (GB)	Compression Ratio	Cache Memory (GB)
10	L_2	0.8355	0.7603	0.9289	2.5342
	LSH	0.8380	0.8120	0.8812	2.6338
30	L_2	0.6234	1.7740	0.6957	7.3492
	LSH	0.6018	1.8531	0.6360	7.5786
50	L_2	0.3968	2.7876	0.4175	12.1641
	LSH	0.3716	2.8941	0.3901	12.5235
70	L_2	0.1967	3.8013	0.1803	17.2325
	LSH	0.1857	3.9351	0.1740	17.7285
90	L_2	0.0859	4.8150	0.0498	22.0474
	LSH	0.0823	4.9761	0.0483	22.6734
100	Full	0.0000	12.6934	0.0000	51.1181

Table 4: GSM8K and MedQA Question Answering KV Cache Memory Usage. LSH maintains a binary hash matrix of attention keys in memory and therefore has slightly higher memory usage than L_2. Our implementation uses 8-bits for binary values instead of 1-bit. Using 1-bit binary numbers will reduce the memory overhead of LSH by a factor of 8 and decrease the difference of memory usage between LSH and L_2.

C Ablation on LSH Dimension

LSH Dim	BertScore F1	Rouge L	GPT4 as a Judge	Compression Ratio	Cache Memory (GB)
4	0.8807	0.3974	4.3833	0.3728	2.8062
8	0.8802	**0.3975**	**4.4113**	0.3734	2.8355
16	**0.8807**	0.3972	4.3753	0.3716	2.8941
24	0.8802	0.3951	4.3733	0.3711	2.9527
32	0.8796	0.3926	4.3220	0.3710	3.0113
64	0.8797	0.3900	4.2333	0.3702	3.2456

Table 5: GSM8K Question Answering performance for different LSH dimensions. Cache budget is fixed at 50%. LSH dimension does not have a significant impact on performance. Small LSH dimensions perform slightly better than larger LSH dimensions.

To determine the effect of the LSH dimension, we conducted an ablation study using the GSM8K dataset. Fixing the cache budget to 50%, we tested LSH dimensions of 4, 8, 16, 32 and 64 bits. Table 5 shows the results. The LSH dimension does not have a major impact on performance. In fact, 8 bits performed the best, but just slightly higher than using more dimensions. This demonstrates that LSH does not need to have a high dimensionality and has minimal storage overhead. Using 8 bits, the storage overhead is 1 byte * cache size. For example, in a Llama3 70B-Instruct deployment with 80 layers, 8 KV-heads, sequence length of 8192, batch size of 8 and 50% cache budget, LSH 8-bit, 16-bit and 32-bit only use an extra 20MB, 40MB, and 80MB respectively, which are significantly smaller than the KV cache size of 640GB.