Spotlight Attention: Towards Efficient LLM Generation via Non-linear Hashing-based KV Cache Retrieval

Anonymous Authors[1]

Abstract

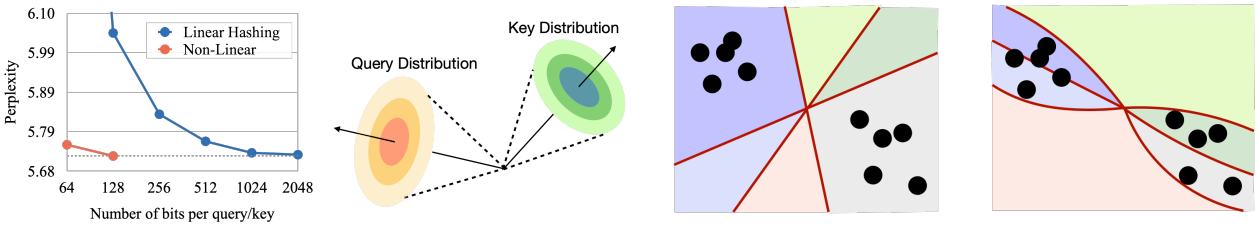
Eliminating the key-value (KV) cache burden of Large Language Models (LLMs) demonstrates great potential for inference acceleration, where dynamically selecting important KV caches during decoding holds advantages for performance maintenance. However, existing methods rely on random linear hashing to retrieve important tokens, which exhibit poor encoding efficiency since queries and keys in LLMs are orthogonally distributed in two narrow cones. In this work, we propose **Spotlight Attention**, a novel method that uses non-linear hashing functions to optimize the embedding distribution of queries and keys, thereby achieving more efficient and robust coding. Furthermore, we develop a lightweight training framework based on a Bradley-Terry ranking objective-based loss, which enables the non-linear hashing module associated with 7B LLMs to be optimized on GPUs with 16GB of onboard memory in 8 hours. Experimental results show that Spotlight Attention drastically improves the retrieval precision while shortening the hash code length 5× compared to traditional linear hashing. At last, we exploit the computational advantages of bitwise operations by implementing specialized Triton kernels. This achieve hashing retrieval for 1M tokens in under 300 μs on a single RTX3090, with end-to-end latency up to 6× faster than FlashAttention 2.5.8 or FlashInfer 0.1.6, which combines fused attention and paged attention. All the training and evaluation stuff can be found at Anonymous/Spotlight.

1. Introduction

Large Language Models (LLMs) are propelling groundbreaking advancements in various natural language tasks, significantly enhancing applications such as content creation and chat assistance. Generally, the inference process of LLMs can be divided into **(i)** the pre-filling phase calculates the key-value (KV) cache for input tokens in the prompt prior to autoregressive generation, and **(ii)** the decoding phase auto-regressively generates tokens, producing one token per forward pass based on the KV cache. Among them, the decoding phase serves as the primary inference bottleneck due to the frequent exchanges between on-board and on-chip memory for model parameters and KV cache, which limits GPU scalability (Agrawal et al., 2023) more so than the pre-filling phase that processes input prompts in parallel. For example, deploying LLaMA2-7B (Touvron et al., 2023) on a single RTX3090 for a single request achieves nearly 100% GPU utilization during the pre-filling phase but drops to below 10% on average during decoding, which largely restraining the inference efficiency of LLMs.

To alleviate this inference bottleneck, extensive research has focused on heuristically eliminating the KV cache burden based on attention scores (Zhang et al., 2023; Xiao et al., 2024; Oren et al., 2024). While effective for short sequences, such irreversible removal of KV cache can significantly degrade performance on long-sequence tasks, especially in Needle-in-a-Haystack scenarios (nee, 2023). To explain, tokens initially considered unimportant and removed might later attain higher attention scores during the prolonged decoding phase, which is crucial for output quality (Tang et al., 2024; Li et al., 2025). To overcome this limitation, recent works have turned to retain all KV cache tokens and dynamically selecting important tokens for computation during decoding (Tang et al., 2024; Chen et al., 2024b), which is the focus of this paper.

Despite the convincing performance of such on-the-fly KV cache selection, how to effectively pick up those important tokens remain challenging. As a pioneering effort, Quest (Tang et al., 2024) selects tokens by matching queries and keys in a block-wise manner. While effective, such coarse-grained selection hardly guarantee precise localization of important tokens. MagicPIG (Chen et al., 2024b) advances Quest by implementing token-level cache retrieval, specifically utilizing Local Sensitive Hashing (LSH) to encode queries and keys into hash codes and pinpointing the best matches as the selected tokens. However, as depicted in Figure 1(a), the efficacy of such linear hashing heavily depends on the hash code length, *e.g.*, a hash code length of 1,024 bits per query/key is necessitated to achieve promising



(a) Linear hashing struggles. (b) Query-key distribution. (c) Linear hashing. (d) Non-linear hashing.

Figure 1. (a) Linear hashing shows acceptable perplexity only when hash code size exceeds 1,024 bits, while non-linear hashing achieves the same level with just 128 bits. (b) According to (Chen et al., 2024b), queries and keys are typically distributed within two narrow, nearly orthogonal cones in space. (c) An illustrative case where all queries and keys are mapped to identical hash codes. (d) Non-linear hashing function can better fit skewed distribution and significantly increase code quality.

token retrieval. Considering the already substantial size of the KV cache, storing these lengthy hash codes markedly impairs deployment efficiency.

Delving deeper, prior work (Chen et al., 2024b) has discovered that the queries and keys within LLMs typically form nearly orthogonal cones within the embedding space, as depicted in Figure 1(b). Given the truth that linear hashing function partitions the embedding space using random hyperplanes, such orthogonal distribution of queries and keys barely lead to satisfying encoding quality, which can even result in a collapse of the hashing outcomes, *i.e.*, identical codes for all queries and keys, as shown in Figure 1(c). Therefore, extremely long hash codes are necessary to mine meaningful information and accurately match essential tokens. MagicPIG attempted to mitigate this issue by normalizing the keys before retrieval. However, this approach remains suboptimal as it overlooks the query distribution and introduces bias to the retrieval process.

To address the aforementioned limitation, we propose Spotlight Attention, a novel method that replaces random hyperplanes with curved surfaces for space partitioning via a non-linear MLP hashing function. As depicted in Figure 1(d), this non-linearity can better fit skewed distributions, thereby improving code quality. In particular, we utilize the Bradley-Terry ranking objective (Bradley & Terry, 1952) to optimize the non-linear MLP layer, wherein the learning target involves minimizing the difference between the LSH output and the ground truth top-k indices obtained via the vanilla attention scores. This learning process is exceptionally efficient, with the LLM backbone remaining frozen and requiring only a minimal amount of calibration data. As a result, the optimized non-linear hashing function can match the performance of linear hashing while using 5$\times$ shorter hash codes, achieving the same efficiency as block-level selection methods like Quest. We further implemented efficient Triton kernels for the hash code processing, including bit-packing and bitwise NXOR operations, achieving significant latency reductions in practice. For instance, our method reduces LLM inference latency by 5.84$\times$ for 32K sequences and 7.52$\times$ for 1M sequences, with only $\sim$2% performance degradation.

Our contributions are threefold:

- We propose Spotlight Attention for accelerating LLM inference, which employs non-linear hashing function to encode and match queries and key values within LLMs, thereby efficiently select critical KV cache for model inference.
- We develop a lightweight training framework based on the Bradley-Terry ranking objective, which effectively optimizes the non-linear hashing layer using only a small amount of calibration data.
- Extensive experiments demonstrate that Spotlight Attention can drastically reduce LLM inference latency while maintaining strongest performance retention in comparison with state-of-the-art methods

2. Related Work

This section covers the spectrum of studies on LLM KV cache pruning that closely related to our work, which we heuristically categorize into static pruning, dynamic pruning with permanent eviction, and dynamic pruning without permanent eviction.

Static KV Cache Pruning. These methods compress the KV cache once after the pre-filling phase, using the compressed cache for subsequent decoding. For example, FastGen (Ge et al., 2024) introduces a pattern-aware approach by identifying five fundamental attention structures and applying targeted selection strategies. SnapKV (Li et al., 2024) further simplifies FastGen by focusing solely on retrieving tokens based on their importance scores, showing that only a subset of prompt tokens carry critical information for response generation and retain their significance during the whole decoding phase. However, without pruning during

decoding, these methods are primarily suited for scenarios with long prompts and relatively short responses.

Dynamic Pruning with Permanent Eviction. This category of methods performs dynamic KV cache pruning during the decoding phase, permanently removing pruned KV cache tokens from memory. For example, H2O (Zhang et al., 2023) leverages cumulative attention scores to retain high-impact tokens. NACL (Chen et al., 2024a) identifies a fundamental limitation in H2O, namely their dependence on potentially biased local attention statistics. To overcome this issue, they develop an alternative approach implementing a diversified random eviction strategy. Keyformer (Adnan et al., 2024) highlights that token removal distorts the underlying softmax probability distribution. Considering the pivotal role of softmax distributions in token significance evaluation, they incorporate regularization techniques to mitigate these distributional perturbations. Unlike static KV cache selection, these methods enable dynamic pruning during decoding, making them better suited for tasks requiring extensive generation. However, they assume that critical information is concentrated in a small subset of KV cache tokens, a condition that dose not always hold. As Chen et al. (2024b) point out, token importance can vary significantly across tasks, leading to premature eviction of tokens before they are needed. For example, H2O may fail to answer questions like *a is b, c is d, a is ?* due to forgetting earlier facts.

Dynamic Pruning without Permanent Eviction.

The limited applicability of permanent token eviction methods has led to a shift toward non-permanent eviction approaches. These methods assume the importance of KV cache tokens varies with each query, requiring importance estimation at every decoding step. Instead of permanently evicting unimportant tokens, they exclude them from attention calculations for that step only. While this improves accuracy, it demands frequent importance estimation, unlike permanent eviction methods that prune tokens in batches after many steps. Research has therefore focused on optimizing the efficiency and accuracy of these estimations. Quest (Tang et al., 2024) groups KV cache tokens into blocks, estimating block importance via the dot product between queries and block representations derived from the minimum and maximum key values. Although efficient, this approach suffers from internal fragmentation, as entire blocks are processed even if only a few tokens are important. MagicPIG (Chen et al., 2024b) eliminates this issue by mapping queries and keys to hash codes for token-level retrieval via Hamming distance, avoiding fragmentation but reducing efficiency. Building on MagicPIG, our method significantly shortens hash code, drastically reducing computation while preserving accuracy.

3. Methodology

3.1. Preliminary

Attention Computing. We first present the basic preliminaries for attention computation and KV cache pruning during the decoding phase of LLMs. We define the query, key, and value inputs to the attention module as $Q, K, V \in \mathbb{R}^{1\times d}$, where d is the embedding dimension. We use $\oplus$ to denote concatenation, and $K_{\text{cache}}, V_{\text{cache}} \in \mathbb{R}^{n\times d}$ represent the key-value cache generated during the pre-filling phase and previous decoding steps. With these definitions, the standard attention is calculated as follows:

$$\mathcal{A} = \text{softmax}\left(\frac{f(Q, K_{\text{cache}}) \oplus QK^\top}{\sqrt{d}}\right)(V_{\text{cache}} \oplus V), \quad (1)$$

where $f(X, X') = XX'^\top$ is inner-product.

KV Cache Pruning. As the decoding sequence length increases, the size of the key-value cache $\{K, V\}_{\text{cache}}$ can grow exceedingly large, creating an LLM inference bottleneck. KV cache pruning that selectively preserves only essential portions of the cache for computation serves as an efficient way to alleviate this problem. Given a desired cache budget K, it first identifies the indices $\mathcal{I}$ of top-K important tokens and then extracts a subset of the KV cache $\{K, V\}_{\text{subset}}$ for attention computation as

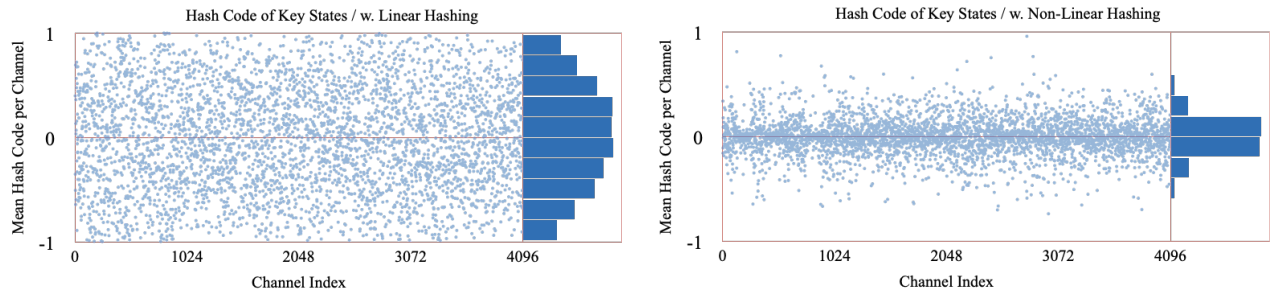
$$\{K, V\}_{\text{subset}} = \text{gather}(\{K, V\}_{\text{cache}}, \mathcal{I}). \quad (2)$$

As previously discussed, existing methods for pruning the KV cache either permanently eliminate cache entries not in the set $\mathcal{I}$ (Zhang et al., 2023; Li et al., 2024) or retain all caches but dynamically determining $\mathcal{I}$ during the decoding process (Tang et al., 2024; Chen et al., 2024b). In this paper, we focus on the latter due to its superior performance preservation.

3.2. Revisiting Token-level Cache Retrieval

Upper bound performance. Token-level cache retrieval refers to dynamically performing cache selection in granularity of single token (Chen et al., 2024b). Here we first conduct a preliminary experiment to examine the upper-bound performance of such token-level retrieval by utilizing full-precision attention scores $f(Q, K_{\text{cache}})$ to identify important tokens.[1] Surprisingly, by fixing the first two layers in line with Quest (Tang et al., 2024) and pruning only the subsequent layers, 98% of KV cache can be safely pruned with only a 0.1 increase in perplexity on PG19, revealing significant untapped potential. This indicates that token-level cache retrieval can achieve nearly lossless KV cache compression, contrasting sharply with existing results (Chen et al., 2024b).

[1]The pseudocode for this upper-bound performance test is detailed in Appendix A.2.



(a) Linear hashing. (b) Non-linear hashing.

Figure 2. Bit channel statistics. Each of the 32 key heads is processed with 128 hashing functions, forming 4,096 bit channels. A mean near -1 or 1 indicates most keys map to the same bit, rendering the channel uninformative. The non-linear hashing approach significantly reduces such channels.

Using LSH to Retrieve Important KV Entries. Although the oracle attention score effectively indicates important tokens, it is impractical to pre-compute f in real-world scenarios. Consequently, MagicPiG employs $\tilde{f}$, an approximation of f using Locality-Sensitive Hashing (LSH) to approximately retrieve critical KV entries. Specifically, it utilizes a linear hashing function $\mathcal{H}$ to produce $\tilde{f}$ as

$$\tilde{f}(X, X') = \mathcal{H}(X) \otimes \mathcal{H}(X'), \tag{3}$$

where $\otimes$ denotes an operation similar to matrix multiplication, substituting all floating-point multiplications with NXOR operations. The indices $\mathcal{I}$ representing the top-k largest values in $\tilde{f}(Q, K_{\text{cache}})$ are subsequently leveraged to retrieve the most relevant KV entries.

LSH is a technique aimed at clustering similar input items into identical buckets with high probability, making it suitable for dense vector spaces. A common approach is the use of random hyperplanes. For a vector $x \in \mathbb{R}^d$, a random matrix $R \in \mathbb{R}^{d \times d_{\mathcal{H}}}$ is used to calculate the hash code as follows:

$$\mathcal{H}(x) = \text{sign}(xR), \tag{4}$$

Once hash codes are generated, original vector similarities can be rapidly evaluated using the Hamming distance, facilitating efficient similarity searches in vector spaces.

Although the random hyperplane model described in Eq. (4) is common, various alternative hashing techniques exist. For instance, angular LSH suggests swapping the matrix R with a rotation matrix to enhance performance. However, these methods generally depend on linear hashing without the need for further optimizations. Yet, in the latent spaces of LLMs, queries and keys often exist in cone-shaped regions and exhibit orthogonality, posing challenges for linear hashing, where non-trivial boundaries frequently map a majority of keys to identical bits, causing significant encoding inefficiencies. Empirically, employing a 1024-bit hash code resulted in an Intersection over Union (IoU) below 40% between the LSH-predicted top-k set and the ground truth, highlighting a critical limitation of the methods.

Spotlight Attention. To address these challenges, we introduce Spotlight attention, which employs a non-linear hashing function designed to adapt to the query and key distributions. This method significantly enhances the information density of hash codes, enabling more effective partitioning of skewed distributions by non-linear boundaries. Particularly, to overcome the limitations observed with linear hashing, we propose a non-linear MLP hashing function. Differing from the standard linear hashing R, we utilize a two-layer MLP:

$$\text{MLP}(x) = W_2\big(\text{SiLU}(W_1 x + b_1)\big), \tag{5}$$

Where W_1, b_1, and W_2 are learnable parameters. Following this transformation, hash codes are obtained similarly to those in Eq. (4) as:

$$\mathcal{H}(x) = \text{sign}(\text{MLP}(x)). \tag{6}$$

By optimizing the parameters W_1, b_1, and W_2, we can develop a hashing function with significantly greater capacity, capable of fitting specific data distributions.

Optimizations. Optimizing the hashing function $\mathcal{H}$ to better capture the distributions of queries and keys are crucial for generating higher-quality hash codes. Next, we introduce two intuitive training objectives and explain why they are unsuitable for this scenario.

Objective 1: Language Modeling Loss. A straightforward approach to optimizing $\mathcal{H}$ is to directly minimize the final language modeling loss through a few hundred steps of post-training on a pretraining corpus like RedPajama (Weber et al., 2024). However, this method has significant drawbacks. First, it requires full forward and backward passes through the entire LLM, making it computationally prohibitive. More importantly, designing a "soft top-k" mechanism to differentiate the top-k operator in Eq. (2) poses a substantial challenge.

Table 1. IoU / PPL changes before and after training. IoU, averaged across all heads and layers, with detailed scores in Appendix B.1. **(i)** The upper-bound performance shows minimal degradation from the original model. **(ii)** Training enhances the MLP hashing function more effectively than the linear hashing function. **(iii)** Post-training, Spotlight Attention achieves an IoU of ∼0.40, correctly predicting over half of the top-k indices.

Method	Original	Upper Bound	Linear Hashing		Spotlight Attention	
			Before	After	Before	After
LLaMA2-7B	- / 5.58	1.00 / 5.69	0.17 / 5.86	0.20 / 5.84	0.05 / 20.31	0.41 / 5.72
LLaMA2-7B-Chat	- / 7.10	1.00 / 6.87	0.17 / 7.34	0.19 / 7.45	0.05 / 21.34	0.42 / 6.98
LLaMA3-8B	- / 6.45	1.00 / 6.63	0.15 / 7.12	0.18 / 7.07	0.07 / 148.2	0.34 / 6.59

Objective 2: Attention Reconstruction Loss. An alternative approach uses cross-entropy loss to align the output distributions of $\tilde{f}$ and f. This method allows for independent layer-wise optimization, significantly reducing computational costs. However, it has a critical drawback: *the primary objective is to identify which KV cache entries to include in the computation, not to rank their relative importance.* Training with this loss misallocate capacity by prioritizing the ranking of the excluded KV caches, inadvertently diverging from the core goal. This limitation leads to significant performance degradation.

In summary, a both effective and efficient objective must meet three criteria: **(i)** independent training for each layer, **(ii)** differentiability at every step, and **(iii)** direct optimization of our goal without capacity misallocation. Our proposed method fulfills these requirements by incorporating a loss based on the Bradley-Terry objective.

Ranking Loss. At each layer, we start by following the inference procedure of Eq.(2). Using the top-k indices $\mathcal{I}$, we partition $\tilde{f}(Q, K_{\text{cache}})$ from Eq.(3) into two subsets:

$$\begin{aligned} B &= \text{gather}\left(\tilde{f}(Q, K_{\text{cache}}), \mathcal{I}\right), \\ C &= \text{gather}\left(\tilde{f}(Q, K_{\text{cache}}), \sim \mathcal{I}\right). \end{aligned} \tag{7}$$

where B and C correspond to elements indexed by $\mathcal{I}$ and its complement, respectively. Next, we compute a matrix $Z \in \mathbb{R}^{k\times(n-k)}$ to encode the differences between B, C:

$$Z_{i,j} = B_i - C_j, \quad \forall i, j, \tag{8}$$

To ensure all elements of Z are positive, we define the objective function as:

$$\mathcal{L}_{\text{rank}} = \frac{1}{k(n-k)} \sum_{i,j} \log \sigma(\beta Z_{i,j} - \alpha), \tag{9}$$

where β and α are positive constants used to amplify the separation between B and C, facilitating convergence. By minimizing this loss, the MLP hashing function is optimized to better distinguish between B and C. During inference, there is no need to compute B, C or Z. Instead, the top-k operation is directly applied to $\tilde{f}(Q, K_{\text{cache}})$ to predict the indices $\mathcal{I}$ of the most relevant KV caches. The core of this loss design lies in filtering out supervising signals related to internal ranking within B and C, effectively addressing the issue of capacity misallocation.

Make Hashing Function Differentiable. After computing the loss, errors can be backpropagated to the MLP hashing function. However, the sign function's non-differentiability blocks gradient flow. To resolve this, we substitute the sign function with a soft sign function during training:

$$\text{softsign}(x) = \frac{\gamma x}{1 + \gamma|x|}, \tag{10}$$

where $\gamma \in \mathbb{R}$ is a hyperparameter controlling the extent of smoothing. This soft sign function is used only during training. In inference, the non-differentiable sign function is reinstated.

4. Experimentation

This section addresses two key questions: **(i) How effective is our method?** We first assess improvements in the hashing function brought by our training method and then benchmark it on language modeling, few-shot learning, LongBench, and Needle In A Haystack. **(ii) How efficient is our method?** We develop custom Triton kernels to measure the latency of each step and compare the results with Quest (Tang et al., 2024) as well as two highly efficient attention kernels.

Implementation Details. We use LLaMA3-8B (Meta-AI, 2024), LLaMA2-7B, and LLaMA2-7B-Chat (Touvron et al., 2023) as base models. The MLP hashing function has input, intermediate, and output dimensions all set to 128, with a distinct MLP for each head in every layer. This produces a hash code length of 128 bits, substantially shorter than MagicPIG's minimum of 720 bits. Only the hashing functions are trainable, while all other parameters remain frozen. Training data consists of 8,192 samples, equally drawn from the Book and Arxiv subsets of RedPajama (Weber et al., 2024) To enhance efficiency, hidden states for all layers are precomputed and stored, enabling independent layer-wise

Table 2. Comparison of language modeling perplexity on PG19 (#1), ProofPile (#2), and CodeParrot (#3). LLaMA2 uses 4K left-to-right truncation, while LLaMA3 uses 8K. **(i)** Spotlight Attention matches Quest's performance with less than 81 token budget, compared to Quest's 1024. **(ii)** Compared to the original LLM, Spotlight Attention achieves 98% pruning with only a ∼0.2 PPL increase in average. **(iii)** Cross base model comparison reveals that Quest is more sensitive to Group Query Attention (GQA) (Ainslie et al., 2023) than Spotlight Attention.

Method	Configuration	Frozen Layers	LLaMA2-7B			LLaMA2-7B-Chat			LLaMA3-8B		
			#1	#2	#3	#1	#2	#3	#1	#2	#3
Original			6.879	4.277	3.679	9.212	5.943	4.786	8.604	3.517	5.219
Quest	1024 Token Budget	[0,1]	7.116	4.404	3.854	9.282	5.930	4.879	9.912	4.024	5.893
	512 Token Budget		7.735	4.754	4.054	9.820	6.226	5.084	12.434	4.927	6.646
	256 Token Budget		9.746	5.775	4.571	12.058	7.440	5.686	17.320	6.749	8.693
	128 Token Budget		15.494	8.578	5.996	18.719	11.083	7.345	27.510	11.631	14.205
Spotlight Attention	80% Pruned	[0,1]	6.887	4.278	3.682	9.107	5.860	4.767	8.612	3.519	5.228
	90% Pruned		6.908	4.285	3.689	9.058	5.796	4.754	8.651	3.529	5.239
	95% Pruned		6.959	4.304	3.703	9.067	5.748	4.752	8.734	3.552	5.285
	98% Pruned		7.106	4.364	3.768	9.262	5.770	4.806	8.977	3.621	5.434

training, with no joint fine-tuning performed. Training is conducted with $\gamma = 64$, a pruning rate of 98%, learning rate of 1×10^{-3}, $\beta = 1$, and $\alpha = 3$, for one epoch. Additional training details are in Appendix A.1. While the pruning rate is fixed at 98% durning training, this constraint is lifted during evaluation. For LLaMA2-7B and LLaMA3-8B, a 98% pruning rate retains only 81 and 163 tokens respectively, which is the fewest among all compared methods.

Reproducibility. We ensure reproducibility by using moderate hyperparameters, fixed random seed, and deterministic decoding. All datasets, scripts and model weights are available in the code repository, with reproducibility guaranteed when using the same Python library versions.

4.1. Baselines

Upper Bound. As introduced in Section 3.2, this baseline employs original attention to identify top-k indices, serving as a theoretical upper bound on performance.

Linear Hashing. This method uses training-free angular LSH (Andoni et al., 2015) with the same hash code length as Spotlight Attention, providing a reference for MagicPIG's linear hashing under a unified framework. Initialization details for angular LSH are in Appendix A.5.

Quest. We use Quest's official implementation with default hyperparameters, adjusting only the token budget: beyond the default 1024 budget, we evaluate ultra-low budgets to align with Spotlight Attention. Specifically, for LLaMA2-7B and LLaMA3-8B, we set budgets to 128 and 256, compared to Spotlight Attention's 81 and 163.

MagicPIG. We use MagicPIG's official implementation with default hyperparameters, which retains 64 local tokens and 4 initial tokens by default, differing from Spotlight Attention.

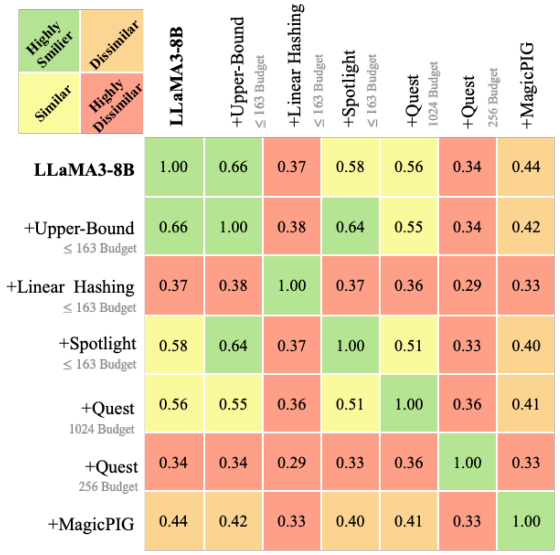


Figure 3. Text-level output similarity measured by Rouge-L. **(i)** Spotlight Attention achieves the highest output fidelity, despite its minimal token budget and the absence of local window or sink tokens, and **(ii)** although gaps exist between predicted top-k indices and ground truth, Spotlight Attention's performance remains close to the upper bound.

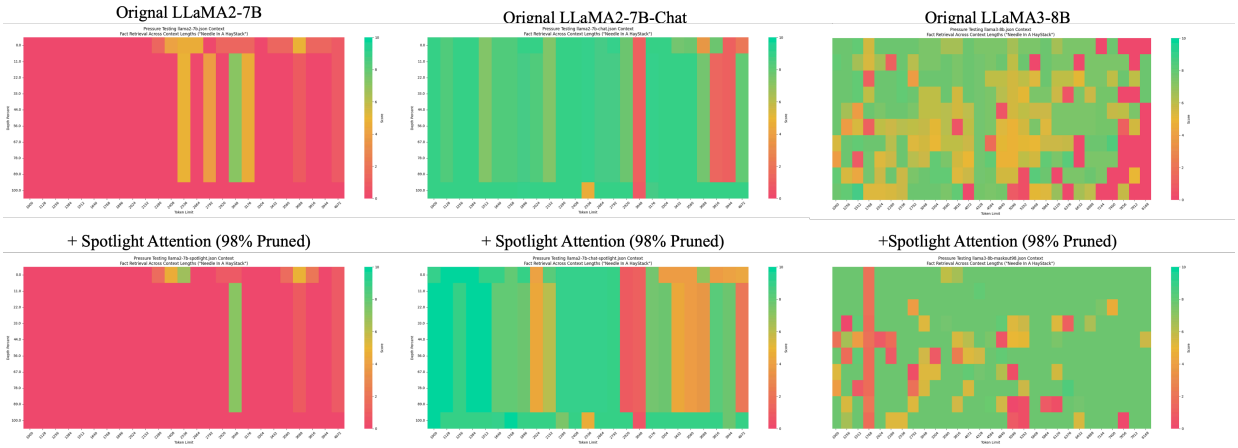
4.2. Effectiveness

Training significantly improves hashing functions. Using IoU as the evaluation metric (introduced in Section 1), we assess the effectiveness of the training framework by comparing the IoU improvement of linear hashing and MLP hashing functions before and after training. Results in Table 1 show that training significantly enhances hash code quality while also demonstrating the superiority of the MLP hashing function over linear hashing function .

Table 3. Performance comparison of different methods on LongBench's long-text downstream tasks: NarrativeQA (1-1), Qasper (1-2), MultiFieldQA-en (1-3), MultiFieldQA-zh (1-4), HotpotQA (2-1), 2WikiMultihopQA (2-2), MuSiQue (2-3), DuReader (2-4), GovReport (3-1), QMSum (3-2), MultiNews (3-3), VCSUM (3-4), TREC (4-1), TriviaQA (4-2), SAMSum (4-3), LSHT (4-4), PassageCount (5-1), PassageRetrieval-en (5-2), PassageRetrieval-zh (5-3), LCC (6-1), and RepoBench-P (6-2). **(i)** With 98% tokens pruned and *no local window or global sink tokens*, Spotlight Attention outperforms Quest and MagicPIG. **(ii)** Spotlight Attention achieves performance on par with the original model, even in tasks like summarization and few-shot learning. **(iii)** On subsets of Chinese (1-4, 2-4, 3-4), other LLaMA2-7B-Chat based models generated answers in English, while Quest produced responses in Chinese, achieving overwhelmingly high scores.

Method	Configuration	Single-Doc.					Multi-Doc.					Summarization					Few-Shot					Synthetic				Code		
		#1-1	#1-2	#1-3	#1-4	Avg.	#2-1	#2-2	#2-3	#2-4	Avg.	#3-1	#3-2	#3-3	#3-4	Avg.	#4-1	#4-2	#4-3	#4-4	Avg.	#5-1	#5-2	#5-3	Avg.	#6-1	#6-2	Avg.
LLaMA2-7B		8.73	7.18	15.42	13.84	11.29	6.55	8.27	2.91	11.38	7.27	15.06	19.80	6.03	9.30	12.54	68.00	30.62	30.83	18.25	36.92	1.26	6.97	8.00	5.41	63.66	56.63	60.14
+Quest	1024 Token Budget	8.78	9.78	17.95	14.86	12.84	8.91	8.51	2.95	12.53	8.22	18.38	20.84	9.63	8.40	14.31	66.00	67.06	30.35	17.50	42.22	1.69	6.47	7.38	5.18	63.88	58.82	61.35
+Quest	128 Token Budget	9.17	4.69	13.69	5.76	8.32	6.00	5.16	2.07	8.05	5.32	6.73	17.78	3.27	3.34	7.78	45.00	31.53	14.53	8.00	24.76	0.83	4.06	1.50	2.13	47.88	44.16	46.02
+MagicPIG	Default	11.85	7.20	20.01	14.23	13.32	8.30	8.23	4.76	12.39	8.42	16.13	20.71	2.21	8.34	11.84	66.50	88.48	35.04	19.50	52.38	0.99	7.78	8.52	5.76	64.95	58.63	61.79
+Spotlight	90% Pruned ($\leq$ 409)	10.39	7.16	15.86	14.21	11.90	6.58	8.44	3.09	11.61	7.43	16.32	19.31	10.24	8.51	13.59	67.50	38.08	30.06	18.50	38.67	2.07	8.27	6.79	5.71	64.10	56.76	60.43
+Spotlight	98% Pruned ($\leq$ 81)	10.76	8.09	16.63	12.95	12.10	6.46	7.98	2.98	11.25	7.09	14.78	19.74	13.88	8.88	14.32	64.50	44.47	26.49	15.00	37.61	2.27	7.14	6.04	5.15	63.71	56.00	59.85
LLaMA2-7B-Chat		18.71	24.83	31.63	8.36	20.88	31.76	28.22	12.66	2.48	18.78	27.18	20.36	26.17	0.24	18.48	64.50	77.85	40.76	15.50	49.65	1.64	2.75	3.42	2.60	54.57	48.74	51.65
+Quest	1024 Token Budget	19.14	17.78	27.12	22.09	21.53	35.99	26.67	12.96	12.33	21.98	27.21	20.62	26.21	14.17	22.05	62.50	78.24	40.53	15.25	49.13	2.00	11.00	6.84	6.61	53.71	50.46	52.08
+Quest	128 Token Budget	14.11	12.78	17.59	9.42	13.47	28.38	21.03	8.99	8.53	16.73	11.55	18.44	20.91	8.39	14.82	38.50	66.44	31.69	10.50	36.78	0.32	7.50	3.30	3.70	43.21	39.55	41.38
+MagicPIG	Default	18.84	23.79	28.80	9.43	20.21	31.97	28.23	11.96	3.17	18.83	26.32	20.31	25.49	0.14	18.65	65.50	85.59	40.52	16.25	51.96	1.52	3.50	4.24	3.08	57.94	52.17	55.05
+Spotlight	90% Pruned ($\leq$ 409)	18.43	24.76	32.3	7.76	20.81	32.16	29.88	12.91	2.80	19.44	27.55	20.22	26.16	0.17	18.53	63.50	75.72	41.92	16.00	49.29	2.35	3.75	4.25	3.45	58.07	53.44	55.75
+Spotlight	98% Pruned ($\leq$ 81)	18.06	25.36	30.99	7.90	20.58	30.95	26.13	12.51	3.79	18.34	27.24	20.58	26.36	0.32	18.63	59.50	74.29	41.27	16.50	47.89	2.97	5.50	5.75	4.74	57.72	52.62	55.17
LLaMA3-8B		4.99	13.30	21.40	21.73	15.36	9.06	11.68	6.21	12.40	9.84	28.80	23.01	3.78	3.56	14.79	71.00	28.48	36.87	35.00	42.83	2.00	6.72	27.61	12.11	49.69	48.18	48.93
+Quest	1024 Token Budget	5.47	13.29	21.41	22.52	15.67	9.45	11.16	6.46	14.69	10.44	26.66	22.16	3.40	5.28	14.37	62.50	55.48	35.75	31.00	46.18	1.87	10.22	19.14	10.41	57.00	62.35	59.67
+Quest	256 Token Budget	5.90	12.68	19.03	18.80	14.10	9.13	11.78	6.44	13.88	10.30	17.41	20.76	2.24	4.51	11.23	47.50	55.10	32.41	26.75	40.44	2.25	6.82	11.61	6.89	61.13	58.07	59.60
+MagicPIG	Default	3.90	13.53	17.51	18.71	13.41	9.16	11.59	6.03	13.70	10.12	23.58	23.90	1.37	4.80	13.41	71.50	90.37	44.02	33.50	59.84	1.20	7.15	13.87	7.40	69.93	65.61	67.77
+Spotlight	90% Pruned ($\leq$ 819)	4.87	13.63	21.54	20.91	15.23	9.16	11.83	6.34	11.89	9.80	27.09	23.35	3.36	3.37	14.29	70.50	39.32	36.89	34.00	45.17	2.03	6.16	24.51	10.90	53.36	47.62	50.49
+Spotlight	98% Pruned ($\leq$ 163)	4.59	14.14	20.48	21.64	15.21	9.82	11.97	6.31	11.89	9.99	25.97	23.28	4.37	3.58	14.30	68.50	51.33	34.06	33.50	46.84	2.07	6.09	20.11	9.42	52.26	44.83	48.54

Figure 4. Needle In A Haystack test results. **(i)** For LLaMA2-7B-Chat and LLaMA3-8B, Spotlight Attention preserves performance even with 98% pruning rate. **(ii)** For LLaMA2-7B, which inherently struggles with instruction following, Spotlight Attention provides no benefit.



Language Modeling. We evaluate language modeling perplexity on PG19 (Rae et al., 2020), ProofPile (pro, 2022), and CodeParrot (lon, 2024) using each model's maximum supported context length. Results, compared with Quest (Tang et al., 2024), are presented in Table 2. Both Spotlight Attention and Quest processing all tokens except the first sequentially according to the decoding procedure. [2].

LongBench. We evaluate performance on *all* subsets from LongBench (Bai et al., 2024), comparing with Quest and MagicPIG. LongBench's sensitivity to prompts and search strategies motivates our use of greedy search for all models to reduce variability. For LLaMA2-7B and LLaMA3-8B, we do not use chat template, while LLaMA2-7B-Chat uses its official chat template. Results are show in Table 3.

Needle In A HayStack. We use the offline evaluation version of Needle In A HayStack[3], which *differs* from ChatGPT scoring by using ROUGE to measure output accuracy. The Needle dataset is highly prompt-sensitive, we provide the needle, retrieval question, and haystack context in Appendix A.3. Additionally, to reduce output variability, all evaluated methods use greedy search. As shown in Figure 4, Spotlight Attention achieves performance on par with the original model, except for LLaMA2-7B.

Output Similarity. The absolute scores on LongBench are insightful, but practical adoption depends more on a

[2]Although MagicPIG can also be processed this way, its latency is nearly an order of magnitude higher than Quest's, so it is excluded from the comparison. For more details, see **Issue#1** and **Issue#2**

[3]`LLMTest_NeedleInAHaystack-Local`

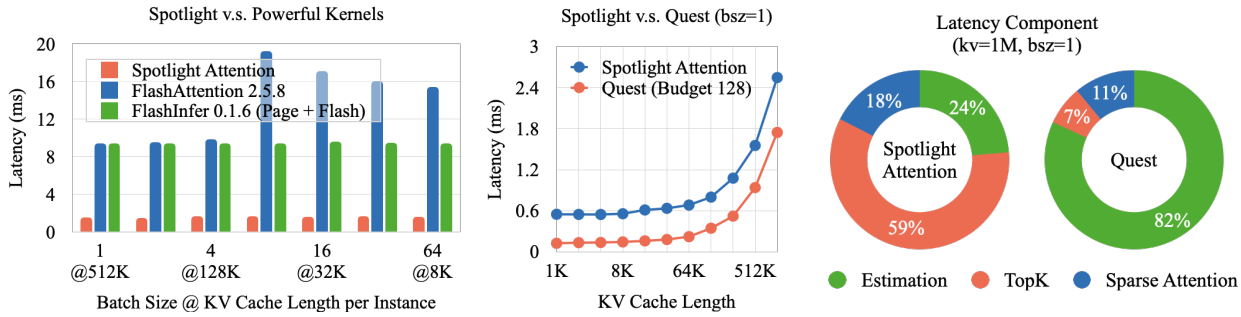


Figure 5. (a) Spotlight attention achieves 6× speedup over other efficient kernels across various configurations. (b) Its latency is comparable to Quest's official kernel, with bit-packing and LSH implemented via Triton, top-k using PyTorch, and sparse attention leveraging FlashAttention, while Quest employs CUDA for all three steps. (c) Top-k dominates Spotlight's end-to-end latency, suggesting limited gains from further optimizing other components.

method's output fidelity. To evaluate this, we measured the average *Rouge-L* score between answers generated by the original model, Spotlight Attention, Quest, and MagicPIG, as shown in Figure 3. For more detailed breakdowns, see Appendix B.2.

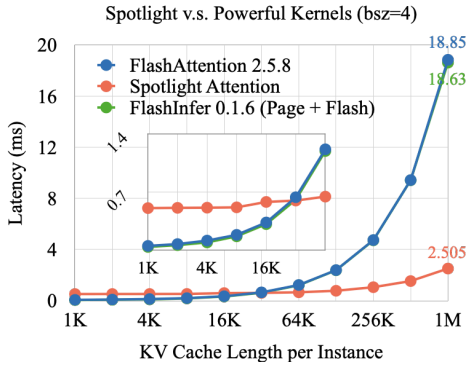


Figure 6. Spotlight Attention delivers lower end-to-end latency than state-of-the-art efficient attention kernels.

4.3. Efficiency

We evaluate *all steps* of Spotlight, including hashing, LSH attention, top-k, and sparse attention (see Figure 6). Spotlight Attention achieves up to a 7.52× speedup over FlashAttention 2.5.8 (Dao, 2024) when processing 262K tokens. For shorter sequences, increasing batch size yields similar speedup ratios (see Figure 5(a)). Overall, processing more tokens concurrently leads to greater acceleration.

5. Conclusion and Limitation

We introduce Spotlight Attention, an advancement over Quest and MagicPIG, incorporating a non-linear hashing function and optimization framework to address the underfitting of MagicPIG's linear hashing while significantly reducing hash code length.

Spotlight Attention demonstrates strong potential in downstream tasks, yet limitations persist. First, IoU remains ∼40% despite the non-linear hashing, indicating room for improvement. Moreover, evaluations across diverse datasets are inherently incomplete, leaving uncertainties about the impact of extreme pruning in real applications.

Impact Statement

By addressing the critical bottleneck in the decoding phase of LLM inference, our method enables faster and more resource-efficient deployment of these models.

From an environmental perspective, the increased efficiency of LLM inference contributes to lower energy consumption and reduced carbon footprint, aligning with global sustainability goals. By minimizing the need for extensive computational resources, our method supports the development of greener AI technologies.

However, as with any significant technological advancement, ethical concerns must be considered. Lowering the cost and resource requirements for deploying such models may inadvertently enable the misuse of these models, including the creation of harmful or malicious language systems. It is essential to address these risks through responsible research practices and the development of robust safeguards.

References

Hugggingface dataset: hoskison-center/proof-pile, 2022. URL https://huggingface.co/datasets/hoskinson-center/proof-pile.

Github repository: Needle-in-a-haystack, 2023. URL https://github.com/gkamradt/LLMTest_NeedleInAHaystack.

Github repository: lm-eval-harness, 2024. URL https://github.com/EleutherAI/lm-evaluation-harness.

Huggingface dataset: namespace-pt/long-llm-data, 2024. URL https://huggingface.co/datasets/namespace-Pt/long-llm-data.

Adnan, M., Arunkumar, A., Jain, G., Nair, P., Soloveychik, I., and Kamath, P. Keyformer: Kv cache reduction through key tokens selection for efficient generative inference. *MLSys*, 2024.

Agrawal, A., Panwar, A., Mohan, J., Kwatra, N., Gulavani, B. S., and Ramjee, R. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills, 2023.

Ainslie, J., Lee-Thorp, J., de Jong, M., Zemlyanskiy, Y., Lebrón, F., and Sanghai, S. Gqa: Training generalized multi-query transformer models from multi-head checkpoints, 2023.

Andoni, A., Indyk, P., Laarhoven, T., Razenshteyn, I., and Schmidt, L. Practical and optimal lsh for angular distance. In *NeurIPS*, 2015.

Bai, Y., Lv, X., Zhang, J., Lyu, H., Tang, J., Huang, Z., Du, Z., Liu, X., Zeng, A., Hou, L., Dong, Y., Tang, J., and Li, J. LongBench: A bilingual, multitask benchmark for long context understanding. In *ACL*, 2024.

Bradley, R. A. and Terry, M. E. Rank analysis of incomplete block designs: I. the method of paired comparisons. *Biometrika*, 1952.

Chen, Y., Wang, G., Shang, J., Cui, S., Zhang, Z., Liu, T., Wang, S., Sun, Y., Yu, D., and Wu, H. NACL: A general and effective KV cache eviction framework for LLM at inference time. In *NACL*, 2024a.

Chen, Z., Sadhukhan, R., Ye, Z., Zhou, Y., Zhang, J., Nolte, N., Tian, Y., Douze, M., Bottou, L., Jia, Z., and Chen, B. MagicPIG: LSH sampling for efficient LLM generation. In *NeurIPS*, 2024b.

Dao, T. Flashattention-2: Faster attention with better parallelism and work partitioning. In *ICLR*, 2024.

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 *ICLR*, 2024.

Li, H., Li, Y., Tian, A., Tang, T., Xu, Z., Chen, X., Hu, N., Dong, W., Li, Q., and Chen, L. A survey on large language model acceleration based on kv cache management, 2025.

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. In *NeurIPS*, 2024.

Meta-AI. The llama 3 herd of models. Technical report, 2024.

Oren, M., Hassid, M., Yarden, N., Adi, Y., and Schwartz, R. Transformers are multi-state rnns, 2024.

Rae, J. W., Potapenko, A., Jayakumar, S. M., Hillier, C., and Lillicrap, T. P. Compressive transformers for long-range sequence modelling. In *ICLR*, 2020.

Tang, J., Zhao, Y., Zhu, K., Xiao, G., Kasikci, B., and Han, S. QUEST: Query-Aware Sparsity for Efficient Long-Context LLM Inference. In *ICML*, 2024.

Touvron, H., Martin, L., Stone, K., and Scialom, T. Llama 2: Open foundation and fine-tuned chat models. Technical report, 2023.

Weber, M., Fu, D. Y., Anthony, Q. G., Oren, Y., Adams, S., Alexandrov, A., Lyu, X., Nguyen, H., Yao, X., Adams, V., Athiwaratkun, B., Chalamala, R., Chen, K., Ryabinin, M., Dao, T., Liang, P., Re, C., Rish, I., and Zhang, C. Redpajama: an open dataset for training large language models. In *NeurIPS*, 2024.

Xiao, G., Tian, Y., Chen, B., Han, S., and Lewis, M. Efficient streaming language models with attention sinks. In *ICLR*, 2024.

Zhang, Z., Sheng, Y., Zhou, T., Chen, T., Zheng, L., Cai, R., Song, Z., Tian, Y., Re, C., Barrett, C., Wang, Z., and Chen, B. H2o: Heavy-hitter oracle for efficient generative inference of large language models. In *NeurIPS*, 2023.

A. Implementation Details

A.1. Training Configuration

Table 4 summarizes the training configuration. LLaMA2 (Touvron et al., 2023) and LLaMA3 (Meta-AI, 2024) adopt different truncation lengths to align with their respective maximum token limits. For additional details, including low-level specifics and weight files, please refer to our open-source code.

Table 4. Detailed training configuration.

General			Learning Rate					Gradient	
Precision	Num Iters	Batch Size	Max LR	Min LR	Warm Up Iters	Warm Up Method	Annealing	Accum	Clipping
bf16	8,192	1	0.001	0	81	linear	cosine	1	1.0

Optimizer				Data					
Optimizer	β_1	β_2	Weight Decay	Corpus	Arxiv Samples	Book Samples	LLaMA2 Trunc	LLaMA3 Trunc	Trunc Side
adamw	0.9	0.98	0.1	arxiv, book	4,096	4,096	4.096	8,192	right

A.2. Pseudo-Code of Oracle Attention Pruning

In the upper-bound performance test, pruning indices are directly obtained from the dot product scores of queries and keys. After that, these indices are used to update the attention mask, as illustrated in the following pseudo-code.

```
def self_attention(hidden_states, attention_mask):
    # ATTENTION_MASK SIZE: (BATCH_SIZE, NUM_HEADS, SEQ_LEN, SEQ_LEN)

    query = q_proj(hidden_states)
    key = k_proj(hidden_states)
    value = v_proj(hidden_state)

    # QUERY/KEY/VALUE SIZE: (BATCH_SIZE, NUM_HEADS, SEQ_LEN, HEAD_DIM)

    score = query @ key.transpose(-1,-2)
    num_pruning = 0.98 * key.shape[-2]
    idx_pruning = torch.topk(score, dim=-2, k=num_pruning, largest=False).indices

    negative_inf = torch.finfo(attention_mask.dtype).min
    attention_mask.scatter_(attention_mask, index=idx_pruning, value=negative_inf)

    attn_output = attention_kernel(query, key, value, is_causal=False, attn_mask=attention_mask)

    return o_proj(attn_output)
```

A.3. Prompt Used in Needle In A Haystack Test

Needle In A Haystack is a prompt-sensitive test, emphasizing relative performance changes before and after applying Spotlight Attention rather than absolute performance. The haystack, needle, and question used in our evaluation are illustrated in Figure 7, with the prompt chosen from a set that proved effective in practice, as shown in Figure 8.

Haystack, Needle and Question

Haystack

Paul Graham Essays

Needle

The best thing to do in San Francisco is eat a sandwich and sit in Dolores Park on a sunny day.

Question

What is the best thing to do in San Francisco?

Figure 7. The haystack, needle, and retrieval question used in the Needle In A Haystack test.

Prompt

LLaMA2-7B / LLaMA3-8B

You are a helpful AI bot that answers questions for a user. Keep your response short and direct. **<context>** **<retrieval question>** Don't give information outside the document or repeat your findings. The document definitely contains the answer, and I'm 100% sure. So try your best to find it.

LLaMA2-7B-Chat

<s>[INST] You are a helpful AI bot that answers questions for a user. Keep your response short and direct. **<context>** **<retrieval question>** Don't give information outside the document or repeat your findings. The document definitely contains the answer, and I'm 100% sure. So try your best to find it. [/INST]

Figure 8. The prompts used in the Needle In A Haystack test.

A.4. Bit-Packing Procedure

Bit-packing (Figure 9) is essential since PyTorch lacks a native bit type, and boolean values are stored as full bytes. Without compaction, storage consumption would increase significantly. The bit-packing logic groups 8 boolean values and iteratively packs them into a single byte, as shown in the pseudocode accompanying Figure 9.

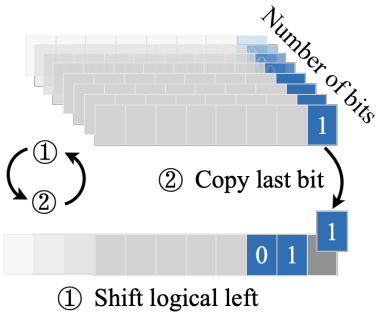


Figure 9. Bit-packing.

```
def bit_packing(tensor):
    n, d = tensor.shape
    assert d % 8 == 0

    tensor = tensor.chunk(8, dim=-1)
    output = torch.zeros((n, d // 8), dtype=torch.int8)
    for x in tensor:
        output <<= 1
        output |= x & 0x01

    return output
```

A.5. Weight Initialization for Linear Hashing

For the linear hashing function, we follow the approach of angular LSH by using a rotation matrix as the initial value for the linear hashing, offering better performance than standard random initialization. To generate a d-dimensional rotation matrix, we adopt the classic QR decomposition method. We first construct a random matrix $R \in \mathbb{R}^{d\times d}$, where each element is independently sampled from a standard normal distribution:

$$R \in \mathbb{R}^{n\times n}, R_{i,j} \sim \mathcal{N}(0,1). \tag{11}$$

We then perform QR decomposition on R, yielding an orthogonal matrix $\mathbf{Q}$ and an upper triangular matrix $\mathbf{R}$:

$$R = \mathbf{QR}, \quad \mathbf{Q}^\top\mathbf{Q} = \mathbf{I}. \tag{12}$$

The matrix $\mathbf{Q}$ not necessarily in the special orthogonal group $\mathtt{SO}(d)$, as its determinant can be either $+1$ or -1. To ensure $\mathbf{Q} \in \mathtt{SO}(d)$, we flip the sign of the first column of $\mathbf{Q}$ if $\det(\mathbf{Q}) < 0$.

B. More Experimental Results

B.1. Per-Head Retrieve Accuracy

The IoU reported in the experiments section represents the average across all heads and layers. Due to the diverse behavior of LLM heads, individual IoU scores vary widely. Detailed scores, including per-head IoU comparisons for different models using Spotlight Attention and linear hashing functions before and after training, are shown in Figure 10.

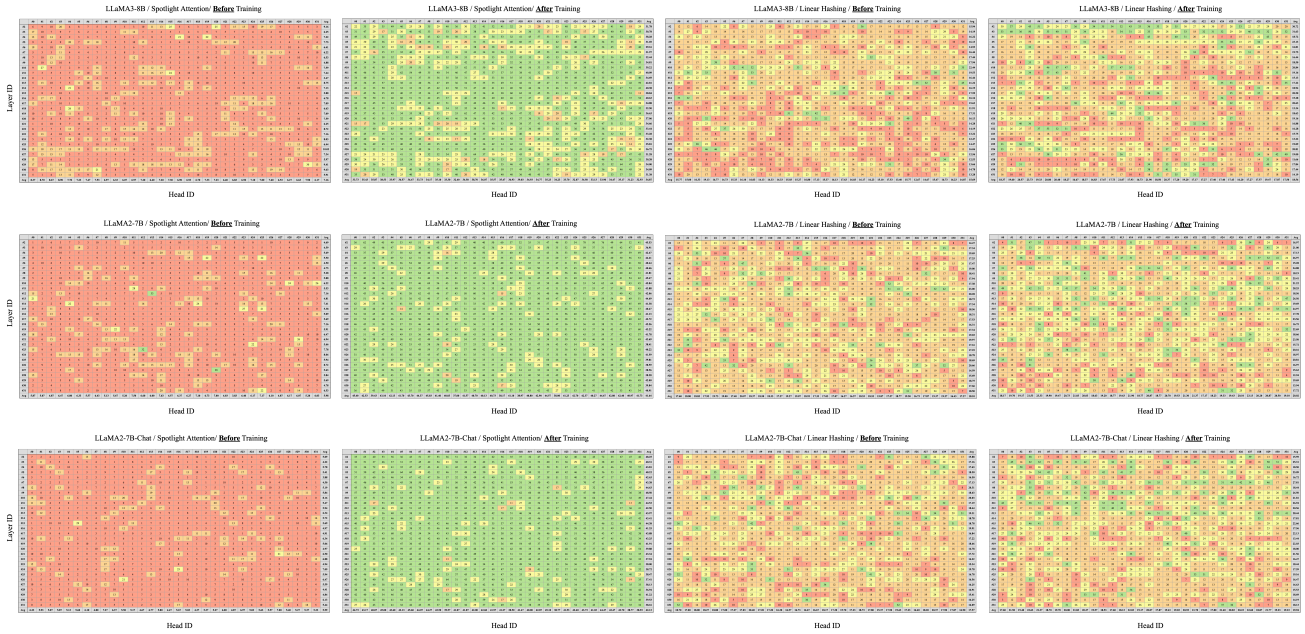
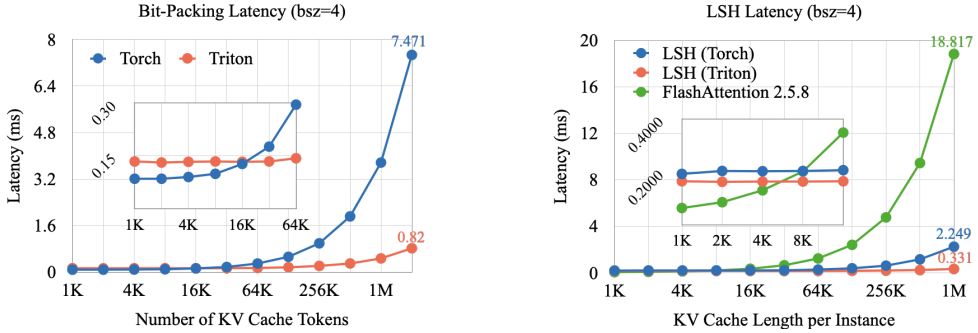


Figure 10. Per-head retrieve accuracy measured by Intersection over Union (IoU) of top-k sets predicted by oracle and Spotlight. **(i)** The results of linear hashing reveals that most heads maintain low IoU both pre- and post-training, suggesting their latent distributions are challenging to approximate with linear functions. **(ii)** Random parameter initialization results in low before-training IoU for Spotlight Attention, which improves significantly after training.

B.2. Detailed Text-Level Output Similarity

In the experimental section, we report the average Rouge-L score across all sub-datasets as the final similarity measure. However, individual sub-datasets scores exhibit significant variation. To eliminate potential confusion, we set the Rouge-L value between identical outputs to 1. However, this dose not apply to certain special cases, such as outputs consisting solely of `\n` characters, where tokenization issues may prevent a perfect score. For clarity, we provide a more detailed similarity measure for each method relative to the original LLaMA3-8B model in Figure 12.



(a) Bit-Packing Latency. (b) LSH Latency.

Figure 11. (a)Our Triton-implmeneted bit-packing kernel achieves substantial speedup. (b) Bitwise NXOR significantly outperforms traditional floating-point operations.

B.3. Latency of Bit-Packing

Efficient bit-packing is crucial for practical applicability. We observed that PyTorch's bit-packing implementation fails to meet performance requirements, significantly slowing down the prefilling phase. To address this, we developed a highly efficient Triton kernel and compared its performance against PyTorch across various context lengths. As shown in

Figure 11(a), the Triton kernel achieves an 8.0× speedup over PyTorch for packing 96K tokens.

B.4. Latency of LSH Retrieval

Efficient LSH execution is critical for minimizing decoding latency, making it a key optimization focus. We compare our Triton implementation with PyTorch and FlashAttention 2.5.8. With 1M keys, Triton LSH achieves a 56× speedup over FlashAttention 2.5.8 and a 6.7× speedup over PyTorch, highlighting the efficiency of NXOR operations and ultra-short encodings.

B.5. Few-Shot Learning.

To evaluate Spotlight Attention with short context lengths, we restrict the KV cache to 20 tokens and assess performance on 5-shot learning datasets from LM-Eval-Harness (eva, 2024), including GLUE, SuperGLUE, OpenBookQA, HellaSwag, PiQA, Winogrande, ARC-E, ARC-C, MathQA, and MMLU.

As shown in Table 5, Spotlight Attention delivers strong performance. Comparisons with Quest and MagicPIG are ommited, as their large local windows or budgets consume most of the prompt length on most datasets, rendering such comparisons less meaningful. Instead, we emphasize the relative performance between Spotlight Attention and the original model.

Table 5. A 5-shot learning comparison between original model and Spotlight Attention under a fixed budget of 20 tokens was conducted across: GLUE (1), SuperGLUE (2), OpenBookQA (3), HellaSwag (4), PiQA (5), Winogrande (6), ARC-E (7), ARC-C (8), MathQA (9), and MMLU (10).

	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	Win Rate
LLaMA2-7B	0.489	0.646	0.342	0.604	0.776	0.733	0.793	0.478	0.262	0.468	56%
+Spotlight	0.494	0.632	0.336	0.581	0.786	0.748	0.793	0.469	0.284	0.466	44%
LLaMA2-7B-Chat	0.625	0.717	0.346	0.598	0.763	0.729	0.811	0.474	0.295	0.482	89%
+Spotlight	0.582	0.664	0.35	0.578	0.759	0.703	0.809	0.467	0.275	0.482	11%
LLaMA3-8B	0.618	0.727	0.378	0.631	0.804	0.772	0.85	0.534	0.418	0.663	38%
+Spotlight	0.620	0.736	0.378	0.624	0.804	0.773	0.851	0.533	0.411	0.664	62%

	LLaMA3-8B	Upper Bound 98% Pruned	Linear Hashing 98% Pruned	Spotlight 98% Pruned	Quest 1024 Budget	Quest 256 Budget	MagicPIG
NarrativeQA	0.95	0.67	0.43	0.59	0.53	0.36	0.58
Qasper	1.00	0.57	0.34	0.54	0.57	0.37	0.48
MultiFieldQA-en	0.93	0.71	0.43	0.62	0.62	0.43	0.49
MultiFieldQA-zh	0.64	0.58	0.24	0.47	0.52	0.33	0.33
HotpotQA	1.00	0.76	0.48	0.66	0.70	0.49	0.69
2WikiMultihopQA	1.00	0.83	0.53	0.75	0.77	0.61	0.69
MuSiQue	1.00	0.78	0.50	0.69	0.72	0.56	0.73
DuReader	0.43	0.69	0.18	0.55	0.37	0.16	0.25
GovReport	1.00	0.58	0.26	0.55	0.51	0.23	0.34
QMSum	1.00	0.48	0.33	0.44	0.39	0.28	0.48
MultiNews	0.28	0.82	0.60	0.79	0.84	0.05	0.03
VCSUM	0.21	0.46	0.19	0.39	0.41	0.06	0.10
TREC	1.00	0.71	0.60	0.63	0.64	0.51	0.70
TriviaQA	1.00	0.66	0.31	0.53	0.55	0.34	0.51
SAMSum	1.00	0.54	0.30	0.48	0.45	0.28	0.34
LSHT	0.07	0.39	0.02	0.26	0.18	0.01	0.02
PassageCount	1.00	0.82	0.58	0.80	0.65	0.59	0.70
PassageRetrieval-en	1.00	0.71	0.41	0.55	0.60	0.42	0.41
PassageRetrieval-zh	1.00	0.66	0.33	0.60	0.61	0.36	0.46
LCC	0.81	0.67	0.34	0.60	0.70	0.40	0.46
RepoBench-P	0.70	0.71	0.31	0.59	0.47	0.30	0.41
Average	0.81	0.66	0.37	0.58	0.56	0.34	0.44

Figure 12. Detailed similarity between **(i)** the outputs of different models (including LLaMA3-8B itself) and **(ii)** those of LLaMA3-8B.