HATA: Trainable and Hardware-Efficient Hash-Aware Top-k Attention for Scalable Large Model Inference

Anonymous ACL submission

Abstract

Large Language Models (LLMs) have emerged as a pivotal research area, yet the attention module remains a critical bottleneck in LLM inference, even with techniques like KVCache to mitigate redundant computations. While various top-k attention mechanisms have been proposed to accelerate LLM inference by exploiting the inherent sparsity of attention, they often struggled to strike a balance between efficiency and accuracy. In this paper, we introduce **HATA (Hash-Aware Top-k Attention)**, a novel approach that systematically integrates low-overhead learning-to-hash techniques into the Top-k attention process. Different from the existing top-k attention methods which are devoted to seeking an absolute estimation of qk score, typically with a great cost, HATA maps queries and keys into binary hash codes, and acquires the relative qk score order with a quite low cost, which is sufficient for realizing top-k attention. Extensive experiments demonstrate that HATA achieves up to **7.2**$\times$ **speedup** compared to vanilla full attention while maintaining model accuracy. In addition, HATA outperforms the state-of-the-art top-k attention methods in both accuracy and efficiency across multiple mainstream LLM models and diverse tasks. To foster academic collaboration, we will open-source the **HATA** implementation soon.

1 Introduction

Recently, KVCache has become a paradigm for the inference of large language model (LLM) (Kwon et al., 2023; Zheng et al., 2023), due to its benefit of mitigating redundant computation in the decoding stage. In this situation, massive KV states loading becomes the bottleneck, especially for long sequences and large batch sizes (Ribar et al., 2023; Tang et al., 2024).

Top-k Attention (Gupta et al., 2021) has emerged as a promising approach to accelerate LLM inference by leveraging the inherent sparsity in attention. By selectively retaining only the top-k most relevant tokens in the KVCache, top-k attention significantly reduces the KVCache loading overhead. However, existing top-k attention algorithms face notable challenges in achieving an optimal trade-off between efficiency and accuracy. Low-rank methods, such as Loki (Singhania et al., 2024) and InfiniGen (Lee et al., 2024), reduce overhead by computing dot-products over a subset of projected dimensions, but they introduce significant computational costs due to the extensive requirements for channel extraction. On the other hand, block-wise methods like Quest (Tang et al., 2024) and InfLLM (Xiao et al., 2024) improve efficiency by grouping contiguous key-value pairs into blocks, but they often compromise accuracy as critical keys may be excluded based on their coarse-grained estimation of query-key (qk) scores.

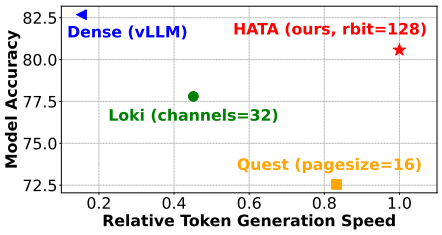

Figure 1: Comparison of accuracy and token generation speed. For detailed analysis, refer to Sec 5.

In this paper, we introduce **Hash-Aware Top-k Attention (HATA)**, a novel approach that systematically integrates low-overhead learning-to-hash techniques into the top-k attention process. Unlike existing methods that focus on precise numerical estimation of qk scores, HATA maps queries and keys into binary hash codes, acquiring the relative qk score order with minimal computational cost. This approach eliminates costly high-fidelity score approximations, enabling significant speedup while preserving the quality of top-k selection. As illustrated in Figure 1, HATA shows superiority in balancing the accuracy and efficiency, compared to

state-of-the-art methods.

HATA leverages the success of **learning-to-hash** (Wang et al., 2012; Weiss et al., 2008), which has been widely used in similarity-based retrieval tasks such as image search and machine learning. By training hash functions based on the query-key pairs of LLM attention, HATA is able to encode any query and key vector into a binary code, which further enable HATA to achieve low-overhead but precise token selection, making it a hardware-efficient solution for accelerating LLM inference.

Extensive experiments demonstrate that HATA achieves up to 7.2× speedup compared to vanilla full attention while maintaining model accuracy. Furthermore, HATA outperforms state-of-the-art top-k attention methods in both accuracy and efficiency across multiple mainstream LLM models and diverse tasks.

In summary, our contributions are as follows:

- We frame key retrieval in top-k attention as a lightweight ordinal comparison task, eliminating the need for costly high-fidelity score approximation.
- We introduce HATA, which systematically integrates learning-to-hash techniques into top-k attention mechanisms to solve this ordinal comparison task.
- We provide hardware-aware optimizations for HATA and validate its effectiveness on multiple models and datasets.

2 Background and Motivation

2.1 LLM Inference

The LLM model consists of multiple transformer layers, each processing continuous token embeddings to iteratively generate the next token embedding. At the core of each transformer layer is the attention module, which computes as follows:

$$\begin{aligned} Q, K, V &= \text{Proj}\left(X\right), \\ AttnOut &= \text{Softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V. \end{aligned} \tag{1}$$

LLM inference is autoregressive. When generating text, the model produces one token at a time, and each new token depends on the ones already generated. This process continues until some stopping condition is met, like reaching an end-of-sequence token or a maximum length. However, the autoregressive nature leads to significant computational redundancy, making attention the primary bottleneck in LLM inference (Dao et al., 2022; Dao, 2024; You et al., 2024).

2.2 KVCache

To accelerate the attention module, the KVCache approach has been proposed to cache and reuse intermediate results to eliminate computational overhead. In more detail, it decouples the inference process into **prefill** and **decode** stages. During the *prefill stage*, the input prompt is processed in parallel, computing and caching the K and V vectors for all tokens across the transformer-attention layers, which initializes the KVCache. In the subsequent *decode stage*, tokens are generated sequentially: at each step, the model computes only the Q/K/V vector of the current token, retrieves cached K/V vectors, and computes attention scores to predict the next token, while appending the new token's K/V vectors to the cache.

Despite the KVCache's computational efficiency, the attention mechanism remains a critical bottleneck for modern LLMs in complex scenarios involving long-context sequences or large batch sizes. Recent studies (Ribar et al., 2023; Tang et al., 2024) reveal that even with KVCache, the attention module dominates inference latency—for instance, consuming over 70% of total runtime when processing 32K-token sequences with Llama2-7B. This inefficiency is contributed not only by the computation complexity but also by memory bandwidth constraints. At each decoding step, the model must load the entire cached Key and Value vectors, incurring massive data movement costs that scale with context length and batch size. Consequently, with KVCache, optimizing attention's memory access patterns has emerged as a pivotal challenge for enabling scalable LLM deployment.

2.3 Top-k Attention

The top-k attention mechanism (Gupta et al., 2021) reduces memory bandwidth overhead under the KVCache framework by exploiting the sparsity of attention distributions. As formalized in Equation (2), it computes attention scores only for the top-k keys with the highest query-key (qk) scores, bypassing computation for low-scoring tokens. While this sparsity preserves model accuracy and reduces FLOPs, it does not fully eliminate the memory bottleneck: as shown in (Ribar et al., 2023), the mechanism still requires loading all keys from the KVCache to evaluate qk scores, incurring at least half of the original memory traffic.

To improve the efficiency of top-k attention, recent work has focused on approximating qk scores with low-cost estimators. Methods like SparQ (Ribar et al., 2023), Loki (Singhania et al., 2024), and InfiniGen (Lee et al., 2024) reduce computational overhead by computing dot-products over a subset of projected dimensions rather than the full embedding space. While these approximations retain theoretical error bounds, they face a dimensionality-accuracy trade-off: preserving estimation fidelity requires retaining a critical mass of dimensions, leading to limited performance gains.

$$\begin{aligned} qkScore &= \text{Softmax}(qK^T) \\ Index &= \text{TopK}(qkScore, k) \\ AttnOut &= \text{Attn}(q, K[Index], V[Index]) \end{aligned} \tag{2}$$

On the other side, block-based approximations, such as Quest (Tang et al., 2024) and InfLLM (Xiao et al., 2024), partition keys into contiguous blocks and estimate upper bounds for aggregate qk scores per block. Tokens within blocks exceeding a score threshold are retained for attention computation. While this reduces the search space, two issues arise. Critical tokens are often dispersed across blocks, and selecting entire blocks forces loading irrelevant intra-block keys, wasting memory bandwidth. Moreover, the coarse-grained estimation may not well distinguish important and irrelevant tokens, hindering the final accuracy.

2.4 Motivation

Prior top-k attention methods operate under the strong assumption that precise numerical estimation of qk scores is essential to replicate the effectiveness of full attention. Thus, they incur significant computational or memory overhead to minimize approximation errors in absolute qk scores.

However, in this paper, we challenge this assumption by demonstrating that only relative qk score ordering—not absolute numerical magnitude—is required to identify the most relevant keys. By reformulating the problem as a lightweight *ordinal comparison* task (e.g., determining whether $s_{qk_i} > s_{qk_j}$) rather than a *numerical regression* task, we eliminate the need for costly high-fidelity score approximations. This relaxation enables remarkable reduction in computation and memory access while preserving top-k selection quality, as precise score magnitudes are irrelevant to the ranking outcome.

Learning-to-hash (Wang et al., 2012) offers a principled framework to achieve our goal, as it maps high-dimensional continuous vectors (e.g. queries and keys) into compact binary hash codes while preserving their relative similarity relationships, i.e., similar vectors are assigned adjacent binary hash codes with small Hamming distances. Nevertheless, integrating learning-to-hash into top-k attention introduces critical challenges:

- **Modeling.** Learning-to-hash was widely used for retrieval tasks, such as image retrieval and information search. To apply learning-to-hash to top-k attention computing, designing an effective hashing model for learning hash codes of query and keys is of great importance.
- **Implementation.** A high-performance implementation is also indispensable to achieve a practical improvement of LLM inference.

3 HATA's Design

To address the aforementioned three challenges, we propose Hash-Aware Top-k Attention (HATA), a trainable and hardware-efficient approach based on learning-to-hash.

In Sec 3.1, we formally define the query-key-based learning-to-hash problem and design a training loss function to learn hash codes while preserving similarity. We also incorporate bits balance and uncorrelation constraints (Wang et al., 2012; Weiss et al., 2008) to enhance hash bit quality. In Sec 3.2, we introduce HATA's workflow, leveraging the learned hash function to significantly accelerate LLM inference.

3.1 Learning-to-Hash for Top-k Attention

Building on learning-to-hash, we design a hash function to map query/key vectors to binary codes while preserving their relative similarity. The learning process is detailed below.

3.1.1 Hash Modeling

Inspired by the learning-to-hash model defined in (Wang et al., 2012), given a query q and multiple keys $K := \{k_i\}_{i=1}^n$, we learn the hash codes of q

and K by solving the following problem:

$$\min \quad \sum_i \text{sim}(q, k_i)||h(q) - h(k_i)||^2 \quad (3)$$

$$\text{s.t.} \quad h(q), h(k_i) \in \{-1, 1\}^r \quad (4)$$

$$\sum_{i=1}^{n} h(k_i) = 0 \quad (5)$$

$$\frac{1}{n}\sum_{i=1}^{n} h(k_i)h(k_i)^T = I_r \quad (6)$$

where $h(\cdot)$ is the hash function to be learned and $\text{sim}(q, k_i)$ defines the similarity of original query q and key k_i. Note that the objective function Equation (3) tends to assign adjacent binary codes for qk pairs exhibiting high similarity, which matches the goal of similarity-preserving hashing. The constraint (4) ensures that the query and keys are encoded into r binary codes. The constraints (5) and (6) are called bits balance and uncorrelation constraints, respectively.

The hash function $h(\cdot)$ is commonly defined as $h(x) = \text{sign}(xW_H)$, where W_H is the trainable hash weights. Due to the non-differentiability of the sign function, we relax $h(x)$ as:

$$h(x) = 2 \cdot \text{Sigmoid}(\sigma \cdot xW_H) - 1, \quad (7)$$

where $\sigma \in (0, 1)$ is a hyper-parameter to prevent gradient vanishing.

For tractability, the balance constraint (5) is further relaxed by minimizing $||\sum_i h(k_i)||^2$, and according to (Wang et al., 2012) the uncorrelation constraint (6) can be relaxed by minimizing $||W_H^T W_H - I_r||$. Then the query-key hashing problem is reformulated as:

$$\min \quad \epsilon \sum_i s_i||h(q) - h(k_i)||^2 + \eta||\sum_i h(k_i)||^2 + \lambda||W_H^T W_H - I_r|| \quad (8)$$

$$\text{s.t.} \quad h(x) = 2 \cdot \text{Sigmoid}(\sigma \cdot xW_H) - 1$$

where s_i is $\text{sim}(q, k_i)$ for short, and ϵ, λ, η control the impact of each objective. Detailed training settings are provided in the Appendix B.2.

Equation (8) formulates hash function learning for a single query and its corresponding keys. To generalize to real-world cases, we extend it to multiple queries and their corresponding keys, as below:

$$\min \quad \epsilon \sum_j \sum_i s_{j,i}||h(q_j) - h(k_{j,i})||^2 + \eta||\sum_{j,i} h(k_{j,i})||^2 + \lambda||W_H^T W_H - I_r|| \quad (9)$$

$$\text{s.t.} \quad h(x) = 2 \cdot \text{Sigmoid}(\sigma \cdot xW_H) - 1$$

where $s_{j,i} = \text{sim}(q_j, k_i)$. Problem (9) is the final hashing model for learning effective hash function $h(\cdot)$, which plays a key role in designing efficient top-k attention algorithm later.

Note that the attention module typically involves multiple independent heads which usually have different characteristics, so we also train a separate hash weight W_H for each attention head.

Algorithm 1 HATA Prefill Stage

Input: $\boldsymbol{Q} \in \mathbb{R}^{s\times d}$, $\boldsymbol{K} \in \mathbb{R}^{s\times d}$, $\boldsymbol{V} \in \mathbb{R}^{s\times d}$, key cache $\boldsymbol{K^{cache}} \in \mathbb{R}^{0\times d}$, value cache $\boldsymbol{V^{cache}} \in \mathbb{R}^{0\times d}$, key code cache $\boldsymbol{K_H^{cache}} \in \mathbb{R}^{0\times rbit/32}$
▷ Call HashEncode to encode key
$\boldsymbol{K_H} \leftarrow$ HashEncode($\boldsymbol{K}$)
▷ Fill hashcode cache
$\boldsymbol{K_H^{cache}} \leftarrow \boldsymbol{K_H}$
▷ Fill KVCache
$\boldsymbol{K^{cache}} \leftarrow \boldsymbol{K}$, $\boldsymbol{V^{cache}} \leftarrow \boldsymbol{V}$
▷ Calculate attention output
$\boldsymbol{O} \leftarrow$ Attention($\boldsymbol{Q}$, $\boldsymbol{K}$, $\boldsymbol{V}$)

3.1.2 Training Data Construction

The training samples are constructed based on real datasets. Specifically, given a sequence, during the prefill stage, we collect $Q := [q_1, q_2, \ldots, q_n]$ and $K := [k_1, \ldots, k_n]$ of each attention head. For each head, we sample a q_j from Q and compute the qkScore between q_j and K. Based on the qkScore, the top 10% of $(q_j k_i)$ pairs are designated as positive samples with linearly decayed labels $s_{j,i} \in [1, 20]$, while the remaining 90% receive fixed negative labels $s_{j,i} = -1$. The label $s_{j,i}$ measures the similarity between q_j and k_i. The training data are organized as triplets $(q_j, k_i, s_{j,i})$ for storage. Since the sequence can be very long, it is easy to generate thousands or even millions of qk pairs for training. To enhance data diversity, we generate training data from dozens of sequences. The details of this process are presented in Appendix B.1.

3.2 HATA Top-k Attention Algorithm

HATA integrates learning-to-hash to top-k attention via two algorithmic innovations: (1) HATA

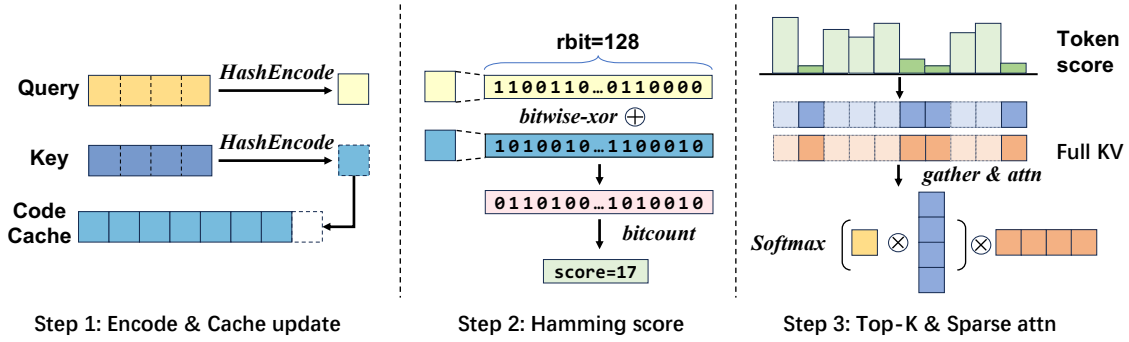

Figure 2: Workflow of HATA in the decode stage.

Algorithm 2 HashEncode

Input: vector $\boldsymbol{V} \in \mathbb{R}^{s \times d}$
Parameter: hash weight $\boldsymbol{W_H} \in \mathbb{R}^{d \times rbit}$
Output: hash code $\boldsymbol{V_H} \in \mathbb{N}^{s \times rbit/32}$
▷ Project input vector into hash code
$\boldsymbol{V_H} \leftarrow \text{Sign}(\text{MatMul}(\boldsymbol{V}, \boldsymbol{W_H}))$
▷ Pack hash code bits into integer format
$\boldsymbol{V_H} \leftarrow \text{BitPack}(\boldsymbol{V_H})$

Prefill: caching hash codes of K; (2) HATA Decoding: efficient top-k key-value detection through hash space.

HATA prefill stage. As shown in Algorithm 1, HATA modifies the original prefill workflow by additionlly computing and caching the hash codes of the keys (lines 2–5), which is critical for accelerating subsequent LLM decoding stages. The hash codes are generated by HashEncode, as shown in Algorithm 2, which leverages Matmul, Sign, and BitPack operators to produce $rbit$ binary code. The hash weight W_H in the HashEncode is obtained through hash training as described in Sec 3.1. Note that the time complexity of HashEncode is $O(s \times d \times rbit)$, where s is the sequence length and d is the vector dimension, while Attention's complexity is $O(s^2 d + s^2)$. Given $rbit \ll s$, the extra prefill overhead from HATA is negligible, accounting for less than 1% of total computation in real tasks.

HATA decode stage. As illustrated in Algorithm 3 and Figure 2, HATA enhances the decode workflow with the following three steps. First, in the *Encode & Cache update* step (lines 3–9), HATA first applies HashEncode to the newly generated query Q and key K, producing query code (Q_H) and key code (K_H), and then updates the key code cache $\boldsymbol{K_H^{cache}}$. Second, it computes the qk hash scores S measured by the Hamming distances between Q_H and all cached key codes in K_H^{cache} (including

Algorithm 3 HATA Decode Stage

Input: $\boldsymbol{Q} \in \mathbb{R}^{1 \times d}$, $\boldsymbol{K} \in \mathbb{R}^{1 \times d}$, $\boldsymbol{V} \in \mathbb{R}^{1 \times d}$, key cache $\boldsymbol{K^{cache}} \in \mathbb{R}^{s \times d}$, value cache $\boldsymbol{V^{cache}} \in \mathbb{R}^{s \times d}$, key code cache $\boldsymbol{K_H^{cache}} \in \mathbb{R}^{s \times rbit/32}$, top-$k$ number N
▷ Update KVCache
$\boldsymbol{K^{cache}} \leftarrow [\boldsymbol{K^{cache}}; \boldsymbol{K}]$
$\boldsymbol{V^{cache}} \leftarrow [\boldsymbol{V^{cache}}; \boldsymbol{V}]$
▷ Call HashEncode to encode query and key
$\boldsymbol{Q_H} \leftarrow \text{HashEncode}(\boldsymbol{Q})$
$\boldsymbol{K_H} \leftarrow \text{HashEncode}(\boldsymbol{K})$
▷ Update code cache with K_H
$\boldsymbol{K_H^{cache}} \leftarrow [\boldsymbol{K_H^{cache}}; \boldsymbol{K_H}]$
▷ Calculate distance in Hamming space
$\boldsymbol{S} \leftarrow \text{bitcount}(\text{bitwise\_xor}(\boldsymbol{Q_H}, \boldsymbol{K_H^{cache}}))$
▷ Select top-k key-value pairs
$\boldsymbol{Idx} \leftarrow \text{TopK}(\boldsymbol{S}, N)$
$\boldsymbol{K^{sparse}} \leftarrow \text{Gather}(\boldsymbol{K^{cache}}, \boldsymbol{Idx})$
$\boldsymbol{V^{sparse}} \leftarrow \text{Gather}(\boldsymbol{V^{cache}}, \boldsymbol{Idx})$
▷ Calculate sparse attention output
$\boldsymbol{O} \leftarrow \text{Attention}(\boldsymbol{Q}, \boldsymbol{K^{sparse}}, \boldsymbol{V^{sparse}})$

the current K_H) using hardware-efficient operations: `bitwise_xor` and `bitcount` (lines 10–11). In situations where multiple queries target the same KVCache, such as GQA, we additionally aggregate the scores S for shared KVCache. Third, based on the hash scores, HATA selects and gathers the most relevant keys and values (lines 13–15), which are then fed into sparse attention (line 17).

4 Hardware-Efficient Optimizations

HATA is implemented in PyTorch (Ansel et al., 2024) and FlashInfer (Ye et al., 2025), comprising 1,470 lines of C++/CUDA code (for custom GPU kernels) and 940 lines of Python code (for high-level orchestration). To bridge the gap between theoretical efficiency and practical performance, we

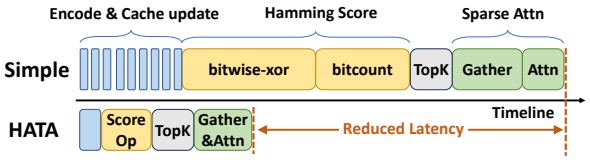

Figure 3: HATA's optimizations, compared to the conventional implementation (denoted as 'Simple').

introduce three hardware-efficient optimizations, as illustrated in Figure 3, targeting compute and memory bottlenecks in attention with long contexts and large batches.

Kernel fusion for hash encoding. The **Encode & Cache update** phase involves a chain of operations such as linear projection, sign function, BitPack, and cache updates. Although each operation takes only a few microseconds on the GPU, the CPU requires tens of microseconds to dispatch them, starving GPU compute units. By fusing these into a single CUDA kernel, we significantly reduce CPU-GPU synchronization, consequently cutting end-to-end inference latency.

High-performance hamming score operator. The Hamming score is computed by matching bits between query and key codes. However, PyTorch lacks high-performance operator support for this computation. To address this, we design an efficient GPU operator with the following hardware-optimized steps: First, both the query and key are loaded as multiple integers, and XOR is applied to produce intermediate integers, where '1' indicates a mismatch and '0' a match. Next, the `popc`/`popcll` instructions count the number of '1's in each integer. Finally, a high-performance reduction operator aggregates these counts to compute the final score. To further boost GPU efficiency, we optimize memory bandwidth by employing coalesced memory access when transferring data from HBM to SRAM.

Fuse gather with FlashAttention. For **Sparse Attn**, the separate gather operations for selected keys and values result in redundant data transfers between HBM and SRAM, diminishing the benefits of hashing. To address this, we integrate the gather operation with the widely-used FlashAttention kernel (Dao et al., 2022; Dao, 2024), streamlining data flow and reducing memory access overhead.

5 Empirical Evaluation

In this section, we evaluate HATA's performance in terms of both accuracy and efficiency.

5.1 Experimental Setup

Experiment platform. We conduct experiments on a machine equipped with a 48GB HBM GPU delivering up to 149.7 TFLOPS (FP16) and 96 cores. The system runs Ubuntu 24.04, utilizing CUDA 12.1, PyTorch 2.4 (Ansel et al., 2024), FlashInfer (Ye et al., 2025).

Baselines and configurations. We compare HATA with the state-of-the-art baselines: Loki (Singhania et al., 2024) (low-rank) and Quest (Tang et al., 2024) (block-level), both of which are variants of top-k attention. In addition, we further compare HATA with MagicPIG (Chen et al., 2024), which accelerates top-k attention through locality sensitive hashing (LSH) (Gionis et al., 1999). LSH is another kind of hashing method, which mainly utilizes random projections to generate hash codes. Different from learning-to-hash, LSH typically requires massive hash bits to ensure accuracy. More details about LSH can be seen in (Gionis et al., 1999). We adopt the recommended configurations (e.g., channels, block size) from the original papers for all baselines. For HATA, we set rbit=128, a versatile configuration that maintains quality across most tasks. Following (Tang et al., 2024), we use vanilla attention for the first two layers, which are typically outlier layers in top-k attention methods. We additionally add the vanilla transformer with full attention mechanism (denoted by dense) as a reference baseline to demonstrate the effectiveness and efficiency of HATA.

Models and datasets. We mainly evaluate HATA on two mainstream large language models: Llama2 (Together, 2023) and Llama3.1 (MetaAI, 2024). The test datasets include two widely used benchmarks: Longbench-e (Bai et al., 2023) and RULER (Hsieh et al., 2024). LongBench-e is a multitask benchmark involving QA, document summarization, and code understanding. RULER focuses on retrieval tasks over extremely long contexts.

Due to space constraints, we only report selected results here. Full results including more models and tasks are provided in Appendix A.

5.2 Accuracy Evaluation

Evaluation on LongBench-e. From Table 1 we can see that for both Llama2 and Llama3.1, HATA can achieve similar results compared with the vanilla full attention mechanism, and outperforms all the other baselines in most cases. Consistent results are obtained on the other Longbench-e tasks,

Methods	Llama-2-7B-32K-Instruct				Llama-3.1-8B-Instruct				AVG.
	LCC	Repo	Trec	Gov	LCC	Repo	Trec	Gov	
Dense	67.53	55.03	69.00	32.01	67.24	52.36	71.66	35.03	56.23
Loki (32 channels)	58.68	44.41	69.00	30.51	61.29	48.47	**72.33**	34.74	52.43
Quest (BlockSize=16)	65.14	52.57	67.57	24.83	58.81	46.72	71.33	33.64	52.58
MagicPIG (K=10, L=150)	66.43	**55.81**	69.00	31.29	53.39	42.35	63.67	32.58	51.82
HATA (rbit=128)	**68.42**	54.92	**69.34**	**31.90**	**67.25**	**51.72**	71.66	**35.02**	**56.28**

Table 1: Partial accuracy results on LongBench-e (Bai et al., 2023) with a sparse token budget of 512. For MagicPIG, the token budget is approximately 2-3% of the sequence length. Full results are in Appendix A.2.

Methods	Llama-2-7B-32K-Instruct				Llama-3.1-8B-Instruct				AVG.
	NS3	NMK2	NMV	QA2	NS3	NMK2	NMV	QA2	
Dense	91.67	81.25	66.67	36.46	100.00	77.08	94.27	40.62	73.50
Loki (32 channels)	0.00	0.00	0.00	16.67	96.88	59.38	**91.67**	35.29	37.49
Quest (BlockSize=16)	52.08	54.17	52.34	34.38	47.92	53.12	78.91	38.54	51.43
MagicPIG (K=10, L=150)	54.17	71.88	59.38	35.42	51.04	20.83	44.79	38.54	47.01
HATA (rbit=128)	**83.33**	**78.12**	**65.62**	**37.50**	**100.00**	**69.79**	89.06	**40.62**	**70.51**

Table 2: Partial accuracy results on RULER (Bai et al., 2023). For Llama-2-7B-32K-Instruct, the context length is 32K and sparse token budget is 1024 (3.13%). For Llama-3.1-8B-Instruct, the context length is 128K and sparse token budget is 2048 (1.56%). For MagicPIG, the token budget is approximately 2-3% of the sequence length. Full results are in Appendix A.2.

as shown in Appendix A.2.

Evaluation on RULER. Next, we test all the methods on the long-context tasks. RULER can be used to construct retrieval, tracing, aggregation and QA tasks with any length. Note that the input sequence length should not surpass the maximum context window size of model. Hence, we test Llama2 and Llama3.1 on 32k-long and 128k-long sequences, respectively. We set the sparse budget as 1024 for Llama2 and 2048 for Llama3.1 (only 3.12% and 1.56% of total sequence length). The results shown in Table 2 is in line with results test on Longbench-e. For long-context inference, HATA can still maintain the accuracy of the vanilla full attention mechanism, while all the other competitors has obvious accuracy degradation, which shows the superiority of HATA.

5.3 Efficiency Evaluation

In this subsection, we compare the efficiency of HATA with other baselines. We first evaluate the end-to-end model inference efficiency, followed by an in-depth analysis of decoding efficiency across varying input scales. For Quest, we directly use their open-source high-performance implementation. For the full attention baseline (dense), we adopt the recently widely-used vLLM (Kwon et al., 2023) implementation. For Loki, since it did not provide a high-performance implementation, here we give an efficient realization based on triton, which is detailed in Appendix C. Note that MagicPIG adopts the offloading technology to save the HMB memory, which will introduce additional latency, so directly comparing time efficiency is not fair. In Appendix A.4, we give an offloading version of HATA, named HATA-off, and compare HATA-off with MagicPIG.

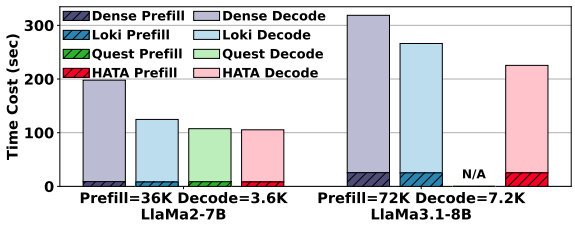

Figure 4: End-to-end performance comparison of LLM inference under 1.56% token selection.

End-to-end inference efficiency. Both HATA and the above-mentioned compared methods are designed for speeding up the LLM decoding. In Figure 4, we compare the decoding time cost of all the methods with the same sequence length. In

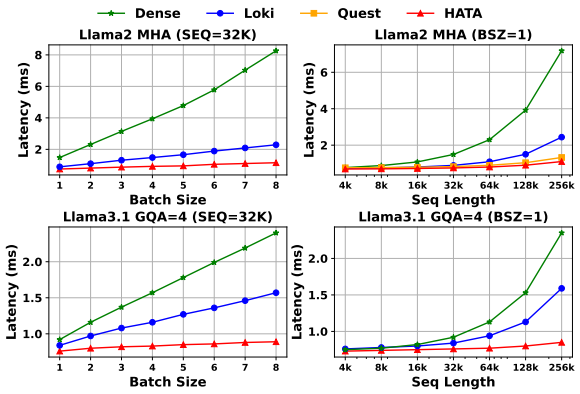

Figure 5: Performance comparison of a single transformer layer under 1.56% token selection.

addition, we also show the prefill time cost to measure the end-to-end efficiency performance of these methods comprehensively. Here we only report the time efficiency of Quest on Llama2, since its open-source high-performance implementation does not support GQA so far. From Figure 4, we see that HATA, Loki, and Quest all have significant speedup in decoding compared with the vanilla attention mechanism, and among them, HATA achieves the highest decoding efficiency. On the other hand, we can see that for LoKi, Quest, and HATA, the prefill time is similar to the vanilla attention mechanism, so all of them can improve the end-to-end inference efficiency. Though it is expected that Quest can achieve similar time efficiency to HATA, HATA can achieve better accuracy under the same budget.

Decoding efficiency across varying input scales. We further evaluate HATA across varying batch sizes and input sequence lengths. Due to GPU memory constraints, we evaluate only a single transformer layer of Llama2 and Llama3.1. Since prefill costs are similar across baselines, we focus on decoding step latency. Furthermore, since the high-performance implementation of the open-source Quest is limited to a batch size of 1 and MHA models, we evaluate it solely across varying sequence lengths on Llama2. As shown in Figure 5, HATA outperforms all the baselines. Notably, with longer sequences and larger batches, HATA achieves greater speedups. With batch size = 8 and sequence length = 32K, HATA reaches up to 7.20× speedup over Dense and 1.99× over Loki. At batch size = 1 and sequence length = 256K, HATA achieves up to 6.51× speedup over Dense, 2.21× over Loki and 1.19× over Quest. These results demonstrate HATA's high inference efficiency across tasks of varying scales.

6 Related Works

Our work HATA advances top-k attention for accelerating KVCache-enabled LLM inference, but significantly differs from existing top-k attention methods. Prior top-k attention methods (Singhania et al., 2024; Ribar et al., 2023; Lee et al., 2024; Tang et al., 2024; Xiao et al., 2024) assume precise qk score estimation is essential to replicate full attention, incurring high computational or memory overhead to minimize errors. Other hashing-based methods for LLMs fail to achieve practical inference acceleration. MagicPIG (Chen et al., 2024) employs locality-sensitive hashing but relies on high-bit representations, limiting speed and sacrificing accuracy. HashAttention (Desai et al., 2024), a concurrent work, also uses learning-to-hash but adopts a custom training approach, lacks extensive testing across datasets and models, and overlooks system challenges in applying hashing to top-k attention. Some works (Sun et al., 2021) attempt hashing in LLM training but fail to transfer it to inference due to fundamental differences between the two phases.

Other orthogonal approaches focus on compressing KVCache content. Eviction methods (Zhang et al., 2024b; Adnan et al., 2024) remove less important tokens but risk information loss and dynamic token importance shifts, potentially degrading output quality. Quantization methods (Liu et al., 2024b; Hooper et al., 2024) compress the KVCache, though their speedup gains are limited by low compression ratios.

Finally, the offloading methods (Lee et al., 2024; Sheng et al., 2023; Sun et al., 2024) transfer KVCache to CPU memory to reduce HBM memory usage. HATA is orthogonal to these methods and can be combined with them. Appendix A.4 demonstrates that HATA can be easily and efficiently combined with KVCache offloading.

7 Conclusion

We introduced Hash-Aware Top-k Attention (HATA), a hardware-efficient method for faster LLM inference. HATA offers a systematic exploration and validation of the integration of learning-to-hash techniques into top-k attention mechanisms, achieving up to 7.2× speedup over dense attention and outperforming SOTA methods in accuracy and performance, establishing it as an effective solution for LLM inference acceleration.

8 Limitations

With learning-to-hash, HATA has achieved notable success in top-k attention. However, it still has the following limitations:

Larger-scale training. HATA 's training data consists of millions of query-key pairs sampled from a limited number of sequences, which is sufficient to train effective hash weights. However, expanding the diversity and scale of the training data could further enhance the quality of the hash weights. We plan to explore this in the future to improve HATA 's performance across a wider range of tasks.

Fields of application. HATA is designed to accelerate LLM inference with long contexts or large batch sizes. For small batch sizes and short context sequences, HATA does not provide significant speedup, as the attention module is not the bottleneck in these cases.

MLA adaptor. Over the past month, Multi-Latent Head Attention (MLA) in DeepSeek (Liu et al., 2024a) has gained significant attention. While we've evaluated HATA on MHA and GQA tasks, it remains untested with MLA, which we leave as future work.

References

Muhammad Adnan, Akhil Arunkumar, Gaurav Jain, Prashant Nair, Ilya Soloveychik, and Purushotham Kamath. 2024. Keyformer: Kv cache reduction through key tokens selection for efficient generative inference. *Proceedings of Machine Learning and Systems*, 6:114–127.

Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, and 30 others. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM.

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, and 1 others. 2023. Longbench: A bilingual, multitask benchmark for long context understanding. *arXiv preprint arXiv:2308.14508*.

Yushi Bai, Shangqing Tu, Jiajie Zhang, Hao Peng, Xiaozhi Wang, Xin Lv, Shulin Cao, Jiazheng Xu, Lei Hou, Yuxiao Dong, and 1 others. 2024. Longbench v2: Towards deeper understanding and reasoning on realistic long-context multitasks. *arXiv preprint arXiv:2412.15204*.

Zhuoming Chen, Ranajoy Sadhukhan, Zihao Ye, Yang Zhou, Jianyu Zhang, Niklas Nolte, Yuandong Tian, Matthijs Douze, Leon Bottou, Zhihao Jia, and 1 others. 2024. Magicpig: Lsh sampling for efficient llm generation. *arXiv preprint arXiv:2410.16179*.

Tri Dao. 2024. FlashAttention-2: Faster attention with better parallelism and work partitioning.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359.

Aditya Desai, Shuo Yang, Alejandro Cuadron, Ana Klimovic, Matei Zaharia, Joseph E Gonzalez, and Ion Stoica. 2024. Hashattention: Semantic sparsity for faster inference. *arXiv preprint arXiv:2412.14468*.

Aristides Gionis, Piotr Indyk, Rajeev Motwani, and 1 others. 1999. Similarity search in high dimensions via hashing. In *Vldb*, volume 99, pages 518–529.

Ankit Gupta, Guy Dar, Shaya Goodman, David Ciprut, and Jonathan Berant. 2021. Memory-efficient transformers via top-k attention. *arXiv preprint arXiv:2106.06899*.

Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. 2024. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *arXiv preprint arXiv:2401.18079*.

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

Greg Kamradt. 2023. Needle in a haystack - pressure testing llms. `https://github.com/gkamradt/LLMTest_NeedleInAHaystack`. Accessed, Feb. 2025.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626.

Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. Infinigen: Efficient generative inference of large language models with dynamic kv cache management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 155–172.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, and 1 others. 2024a. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. 2024b. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. *arXiv preprint arXiv:2402.02750*.

MetaAI. 2024. Introducing llama 3.1: Our most capable models to date. https://ai.meta.com/blog/meta-llama-3-1/. Accessed, Feb. 2025.

QwenTeam. 2024. Qwen2.5: A party of foundation models. https://qwenlm.github.io/blog/qwen2.5/. Accessed, Feb. 2025.

QwenTeam. 2025. Qwen2.5-1m: Deploy your own qwen with context length up to 1m tokens. https://qwenlm.github.io/blog/qwen2.5-1m/. Accessed, Feb. 2025.

Luka Ribar, Ivan Chelombiev, Luke Hudlass-Galley, Charlie Blake, Carlo Luschi, and Douglas Orr. 2023. Sparq attention: Bandwidth-efficient llm inference. *arXiv preprint arXiv:2312.04985*.

Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pages 31094–31116. PMLR.

Prajwal Singhania, Siddharth Singh, Shwai He, Soheil Feizi, and Abhinav Bhatele. 2024. Loki: Low-rank keys for efficient sparse attention. *arXiv preprint arXiv:2406.02542*.

Hanshi Sun, Li-Wen Chang, Wenlei Bao, Size Zheng, Ningxin Zheng, Xin Liu, Harry Dong, Yuejie Chi, and Beidi Chen. 2024. Shadowkv: Kv cache in shadows for high-throughput long-context llm inference. *arXiv preprint arXiv:2410.21465*.

Zhiqing Sun, Yiming Yang, and Shinjae Yoo. 2021. Sparse attention with learning to hash. In *International Conference on Learning Representations*.

Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. 2024. Quest: Query-aware sparsity for efficient long-context llm inference. *arXiv preprint arXiv:2406.10774*.

Philippe Tillet, Hsiang-Tsung Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 10–19.

Together. 2023. Llama-2-7b-32k-instruct. https://huggingface.co/togethercomputer/Llama-2-7B-32K-Instruct. Accessed, Feb. 2025.

Jun Wang, Sanjiv Kumar, and Shih-Fu Chang. 2012. Semi-supervised hashing for large-scale search. *IEEE transactions on pattern analysis and machine intelligence*, 34(12):2393–2406.

Yair Weiss, Antonio Torralba, and Rob Fergus. 2008. Spectral hashing. *Advances in neural information processing systems*, 21.

Chaojun Xiao, Pengle Zhang, Xu Han, Guangxuan Xiao, Yankai Lin, Zhengyan Zhang, Zhiyuan Liu, and Maosong Sun. 2024. Infllm: Training-free long-context extrapolation for llms with an efficient context memory. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. 2025. Flashinfer: Efficient and customizable attention engine for llm inference serving. *arXiv preprint arXiv:2501.01005*.

Haoran You, Yichao Fu, Zheng Wang, Amir Yazdanbakhsh, and Yingyan (Celine) Lin. 2024. When linear attention meets autoregressive decoding: towards more effective and efficient linearized large language models. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Xinrong Zhang, Yingfa Chen, Shengding Hu, Zihang Xu, Junhao Chen, Moo Hao, Xu Han, Zhen Thai, Shuo Wang, Zhiyuan Liu, and 1 others. 2024a. Infinitebench: Extending long context evaluation beyond 100k tokens. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15262–15277.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, and 1 others. 2024b. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems*, 36.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody_Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, and 1 others. 2023. Efficiently programming large language models using sglang.

Model	Abbr.	Configs	Values
Llama-2-7B-32K-Instruct (Together, 2023)	Llama2	#Layer #Attention Heads #KV Heads Hidden Size Max Context Length	32 32 32 4096 32768
Llama-3.1-8B-Instruct (MetaAI, 2024)	Llama3.1	#Layer #Attention Heads #KV Heads Hidden Size Max Context Length	32 32 8 4096 131072
Qwen2.5-14B-Instruct-1M (QwenTeam, 2025)	Qwen2.5-14B	#Layer #Attention Heads #KV Heads Hidden Size Max Context Length	48 40 8 5120 1010000
Qwen2.5-32B-Instruct (QwenTeam, 2024)	Qwen2.5-32B	#Layer #Attention Heads #KV Heads Hidden Size Max Context Length	64 40 8 5120 131072

Table 3: Configurations of the models we used for evaluation.

Methods	Settings
Dense	Inference with the full KVCache (dense attention)
top-k	Exact top-k attention
Loki (Singhania et al., 2024)	Number of channels = 32
Quest (Tang et al., 2024)	Block size = 32
MagicPIG (Chen et al., 2024)	K=10, L=150
HATA	Trained hash weights, 128 bits

Table 4: Configurations for the evaluated methods.

A Additional Evaluation Results

In this section, we present supplemental evaluation results.

- In A.1, we provide detailed configurations of the models and top-k attention algorithm baselines used for evaluation.
- In A.2, we supplement Sec 5.2 by reporting the full results on RULER and LongBench.
- In A.3, we additionally compare HATA with dense model in three commonly used benchmarks (InfiniBench, NIAH and LongBench-v2).
- In A.4, we provide the inference performance comparison between HATA-off and MagicPIG.
- In A.5, we conduct ablation studies on HATA, analyzing the effects of hash bits and token budget on inference accuracy, as well as the efficiency gains achieved through the optimizations discussed in Sec 4.
- In A.6, we show that HATA can successfully scale to larger models (Qwen2.5-14B and Qwen2.5-32B) and handle longer contexts (up to 256K tokens).

Task	Llama-2-7B-32K-Instruct					Llama-3.1-8B-Instruct				
	Dense	Loki	Quest	MagicPIG	HATA	Dense	Loki	Quest	MagicPIG	HATA
LCC	67.53	58.68	65.14	66.43	68.42	67.24	61.29	58.81	53.39	67.25
PRetr	11.89	11.97	15.53	10.01	10.61	99.67	99.67	99.67	98.83	99.67
HQA	15.30	14.91	13.64	14.69	15.65	60.21	59.48	60.03	56.28	60.19
TQA	85.03	85.30	85.18	86.17	85.83	91.64	91.45	90.79	77.90	91.94
Repo	55.03	44.41	52.57	55.81	54.92	52.36	48.47	46.72	42.35	51.72
Sam	39.32	38.95	39.24	38.94	39.61	42.55	41.99	39.75	34.28	42.35
Trec	69.00	69.00	67.57	69.00	69.34	71.66	72.33	71.33	63.67	71.66
MQA	22.44	22.11	19.33	21.70	22.39	54.82	54.47	51.50	49.10	55.17
2Wiki	13.13	13.09	12.51	13.29	13.44	44.08	44.33	43.90	37.84	43.82
Gov	32.01	30.51	24.83	31.29	31.90	35.03	34.74	33.64	32.58	35.02
PCnt	1.17	0.52	1.20	1.08	0.34	13.19	12.74	13.16	9.96	12.44
MltN	24.51	23.82	16.61	23.74	25.06	26.19	25.85	25.69	24.57	26.07
Qaspr	11.76	12.82	10.93	11.06	12.31	44.68	45.15	43.52	38.20	43.95
AVG.	34.47	32.78	32.64	34.09	34.60	54.10	53.23	52.19	47.61	53.94

Table 5: Accuracy results on **LongBench-e** (Bai et al., 2023) for Llama2 and Llama3.1 with sparse token budget=512. For MagicPIG, the token budget is approximately 2-3% of the sequence length.

A.1 Models and Baselines

Table 3 summarizes key parameters of the evaluated models. Llama2 uses multi-head attention (MHA), while the other three employ group-query attention (GQA). Table 4 lists configurations of all top-k attention methods used for comparison.

A.2 Supplement Accuracy Results

In this section, we supplement the accuracy comparison with three other baselines as mentioned in Sec 5.2. Table 5 present the full results for Llama-2-7B-32K-Instruct and Llama-3.1-8B-Instruct on LongBench-e respectively, while Table 6 shows all accuracy comparison of the 11 tasks in RULER.

A.3 Addtional Accuracy Results

We additionally test HATA across three commonly used benchmarks: InfiniBench (Zhang et al., 2024a), LongBench-v2 (Bai et al., 2024) and Needle-in-a-Haystack (Kamradt, 2023). In all the three benchmarks, HATA achieves near-lossless accuracy compared with dense model.

InfiniteBench. InfiniteBench covers tasks of QA, coding, dialogue, summarization, and retrieval, with an average length of 214K. We evaluated HATA on this benchmark using Llama3.1 to demonstrate its effectiveness in complex long-context scenarios. The results are shown in Table 7.

LongBench-v2. LongBench-v2 is an update of the LongBench benchmark, which comprises 503 multiple-choice questions with context lengths spanning from 8K to an extensive 2M words. We employed the Llama3.1 model on LongBench-v2. The accuracy results are categorized based on two key dimensions: task difficulty (Easy, Hard) and context length (Short, Medium, Long). As shown in Tabel 8, HATA consistently maintains model accuracy across most tasks, and even outperforms the exact top-k attention in certain scenarios.

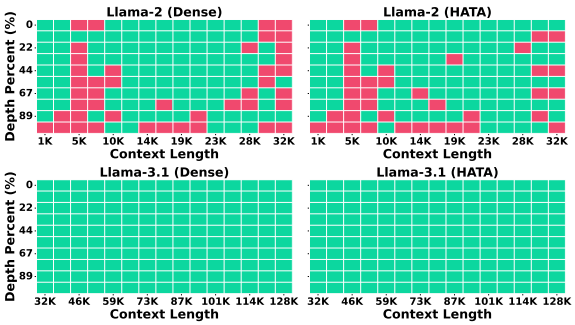

Figure 6: Needle-in-a-Haystack evaluation results. For HATA, the sparse token budget is 512 for Llama2 and 2048 for Llama3.1.

Needle-in-a-Haystack. Needle-in-a-Haystack is a retrieval task. By varying the haystack length and the depth of the needle, we can comprehensively evaluate the effectiveness of HATA in retrieval tasks. For Llama2, we set the haystack length ranging from 1K to 32K to fit within the model's context window. While for Llama3.1, we extended the range from 32K to 128K. As shown in Figure 6, HATA achieves accuracy results similar

Task	Llama-2-7B-32K-Instruct					Llama-3.1-8B-Instruct				
	Dense	Loki	Quest	MagicPIG	HATA	Dense	Loki	Quest	MagicPIG	HATA
NS1	93.75	25.00	100.00	97.92	100.00	100.00	98.96	100.00	94.79	98.96
NS2	100.00	2.08	95.83	93.75	98.96	98.96	97.92	93.75	69.79	98.96
NS3	91.67	0.00	52.08	54.17	83.33	100.00	96.88	47.92	51.04	100.00
NMK1	93.75	0.00	87.50	83.33	93.75	97.92	96.88	97.35	65.62	96.88
NMK2	81.25	0.00	54.17	71.88	78.12	77.08	59.38	53.12	20.83	69.79
NMV	66.67	0.00	52.34	59.38	65.62	94.27	91.67	78.91	44.79	89.06
NMQ	52.08	0.00	56.51	43.49	54.17	96.09	94.79	90.10	57.81	94.53
VT	21.04	1.04	26.87	16.04	20.00	51.04	50.00	61.25	41.67	50.21
FWE	48.61	19.44	36.36	51.39	43.40	75.35	57.99	63.19	57.99	71.18
QA1	30.21	14.58	23.96	30.21	28.12	78.12	76.04	73.96	67.71	76.04
QA2	36.46	16.67	34.38	35.42	37.50	40.62	35.29	38.54	38.54	40.62
AVG.	65.04	7.16	56.37	57.91	63.91	82.68	77.80	72.55	55.51	80.57

Table 6: Accuracy results on **RULER(32K)** (Bai et al., 2023) for **Llama2** with sparse token budget=1024 (3.13%) and **RULER(128K)** for **Llama3.1** with sparse token budget=2048 (1.56%). For MagicPIG, the token budget is approximately 2-3% of the sequence length.

Methods	Sum	Choice	BookQA	DialQA	ZhQA	NumStr	Passkey	Debug	MathFind	AVG.
Dense	20.36	57.64	38.33	18.50	27.57	97.80	100.00	22.59	23.71	45.17
HATA	19.27	57.64	37.52	18.50	27.27	96.44	100.00	22.59	23.71	44.77

Table 7: Accuracy results on **InfiniteBench** (Zhang et al., 2024a) for **Llama3.1** model with sparse token budget=2048. Samples exceeding the model's maximum context window are truncated to fit within it.

Methods	Easy.Short	Easy.Medium	Easy.Long	Hard.Short	Hard.Medium	Hard.Long	Total
Dense	44.07	28.41	31.11	32.23	25.98	25.40	30.42
top-k	40.68	25.00	33.33	29.75	25.20	23.81	28.63
HATA	38.98	27.27	35.56	29.75	26.77	25.40	29.62

Table 8: Accuracy results on **LongBench-v2** (Bai et al., 2024) for **Llama3.1** model with sparse token budget=1024. Samples exceeding the model's maximum context window are truncated to fit within it.

to the dense attention.

A.4 HATA-off and Offloading Efficiency

We also expand an offloading version of HATA, named HATA-off. This section first introduces HATA-off, followed by an performance comparison with MagicPIG (Chen et al., 2024).

HATA-off. Inspired by InfiniGen (Lee et al., 2024), we implement HATA-off using KVCache offloading with approximate query-based prefetching. This improvement helps us precompute the top-k indices for the next layer, creating an opportunity to optimize prefetching key-value pairs. Specifically, we use the input embeddings from layer l to generate approximate queries for layer $l+1$, which allows us to compute top-k indices. This enables prefetching keys/values for the next layer, effectively hiding the CPU-GPU data transfer overhead.

Efficiency evaluation. We compare the inference efficiency of HATA-off with MagicPIG using Llama2 and Llama3.1. Both systems were equipped with PCIe 4.0 and configured with 48 threads in total. The results are presented in Table 9. In the prefill phase, HATA-off achieves speedups of 6.04× and 1.32× on the two models, respectively. This performance difference arises because MagicPIG relies on locality-sensitive hashing (LSH), requiring excessively large hash bits (e.g., 1,500 bits per 2,048-dimensional vector) to preserve similarity, which introduces significant computational overhead, especially for MHA models like Llama2

Time Cost	Llama2 MPIG	Llama2 HATA-off	Llama3.1 MPIG	Llama3.1 HATA-off
Prefill	49.89	8.26	33.24	25.09
Decode	38.21	15.04	41.69	15.86
Total	88.10	23.30	74.93	40.95

Table 9: Offloading performance comparison between HATA-off and MagicPIG (MPIG). We set the prefill length as 36K and 72K for **Llama2** and **Llama3.1** respectively, and the decode length is set as 500 for both model. For MagicPIG, the token budget is approximately 2-3% of the sequence length, and for HATA-off we set the token budgets as 1.56%.

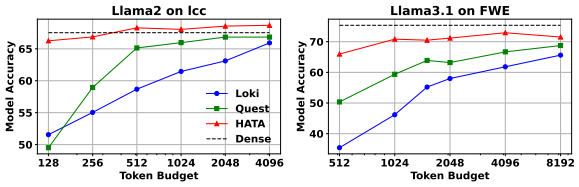

Figure 7: Token budget ablation.

with a large number of KV heads. In the decoding phase, HATA-off achieves speedups of 2.54× and 2.63×, owing to two key advantages: (1) reduced hash bit count and an optimized score operator minimize memory access and computation overhead, and (2) prefetching top-k KVs and computing attention on the GPU outperforms MagicPIG's CPU-based attention implementation. These results highlight HATA-off's effectiveness in offloading scenarios, enabling efficient ultra-long-sequence inference and scalable deployment of large models.

A.5 Ablation Study

In this subsection, we conduct ablation studies on HATA. For accuracy, we investigate the impact of sparse token budget and the number of hash bits on HATA's performance. For inference efficiency, we examine the performance improvements brought by the optimization introduced in Sec 4.

Token budget ablation. First, we examine the impact of token budgets on HATA's performance. As shown in Figure 7, HATA consistently outperforms Quest and Loki under the various budgets. Notably, as budgets decrease, HATA's accuracy degrades minimally, maintaining acceptable performance even under 0.4% token ratio, highlighting the strong potential of learning-to-hash.

Hash bits ablation. Next, we explore the effect of hash bit count (rbit) on inference accuracy. As

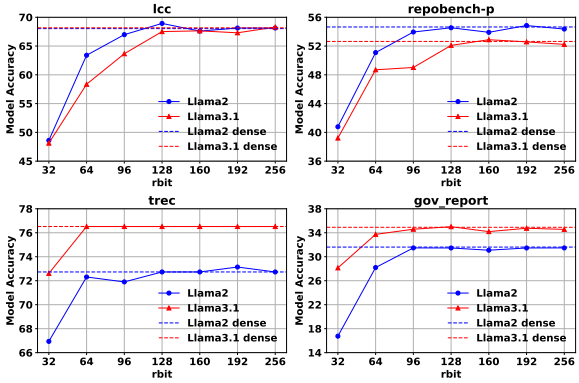

Figure 8: Hash bits ablation.

depicted in Figure 8, increasing rbit from 32 to 128 leads to improved accuracy across four datasets and two models. At rbit=128, accuracy approaches near-lossless levels, comparable to dense attention, with further increases causing only minor fluctuations. Therefore, we adopt rbit=128 as an optimal setting, balancing accuracy and computational efficiency.

Optimizations ablation. Lastly, we evaluate the impact of HATA's optimizations on inference efficiency: high-performance hamming score operator (**Score**), fused gather with FlashAttention (**FusedAttn**), and kernel fusion for hash encoding (**Encode**). Using Llama2's attention module with 128K input, we apply these optimizations incrementally. Figure 9 shows that **Score** reduces the total latency of attention module by 53.2%, **FusedAttn** by 23.8%, and **Encode** by 7.6%. The fully-optimized HATA achieves a 6.53× speedup over a simple PyTorch implementation.

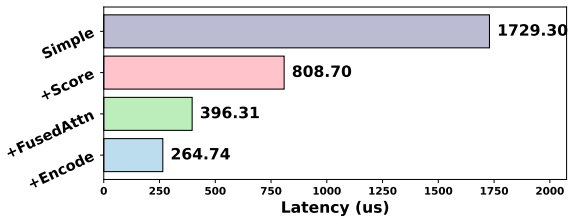

Figure 9: Performance ablation study of HATA optimizations under 1.56% token budget.

A.6 Scalability to Larger-Scale Tasks

We further scale HATA to larger models (14B and 32B) and longer context inputs (256K).

We assessed HATA's accuracy on **Qwen2.5-14B** and **Qwen2.5-32B** using LongBench-e, as de-

Methods	LCC	PRetr	HQA	TQA	Repo	Sam	Trec	MQA	2Wiki	Gov	PCnt	MltN	Qaspr	AVG.
Dense	44.32	100.00	65.96	88.41	36.25	45.52	76.34	53.73	60.68	31.93	22.83	22.14	41.41	53.04
HATA	44.86	99.67	65.87	88.49	37.41	45.41	76.67	53.45	60.70	31.25	20.50	22.02	41.46	52.90

Table 10: Accuracy results on **LongBench-e** (Bai et al., 2023) for **Qwen2.5-14B-Instruct-1M** (QwenTeam, 2025) model with sparse token budget=512.

Methods	LCC	PRetr	HQA	TQA	Repo	Sam	Trec	MQA	2Wiki	Gov	PCnt	MltN	Qaspr	AVG.
Dense	54.04	99.83	69.27	86.26	36.03	43.60	75.67	52.28	60.69	30.14	22.00	21.91	44.08	53.52
HATA	53.90	100.00	68.58	87.55	36.22	42.75	75.67	52.29	60.51	30.17	22.00	21.79	43.70	53.47

Table 11: Accuracy results on **LongBench-e** (Bai et al., 2023) for **Qwen2.5-32B-Instruct** (QwenTeam, 2024) model with sparse token budget=512.

Methods	NS1	NS2	NS3	NMK1	NMK2	NMV	NMQ	VT	FWE	QA1	QA2	AVG.
Dense	100.00	100.00	100.00	100.00	90.00	85.00	97.50	100.00	95.00	60.00	40.00	87.95
top-k	100.00	100.00	100.00	100.00	90.00	81.25	98.75	100.00	88.33	60.00	40.00	87.12
HATA	100.00	100.00	100.00	100.00	95.00	85.00	97.50	96.00	85.00	60.00	45.00	88.05

Table 12: Accuracy results on **RULER(256K)** (Bai et al., 2023) for **Qwen2.5-14B-Instruct-1M** (QwenTeam, 2025) model with sparse token budget=4096 (1.56%).

tailed in Table 10 and Table 11, respectively. For both 14B and 32B models, HATA maintains near-lossless accuracy, underscoring its efficacy with large-scale models.

We further evaluated HATA's performance on extreme-long contexts using RULER-256K on Qwen2.5-14B-Instruct-1M. The results, as shown in the table, demonstrate that HATA achieves comparable accuracy to exact top-k attention, highlighting its capability to handle ultra-long context inputs effectively.

B Configuration Details of HATA Training

B.1 Data Sampling

We trained hash weights based on query and key data sampled from real world datasets. Detailed sampling steps for a given sequence are as follows:

1. For a given token sequence of length n, generate its query $Q := [q_1, q_2 \dots q_n] \in \mathbb{R}^{n \times d}$ and key $K := [k_1, k_2 \dots k_n] \in \mathbb{R}^{n \times d}$ by prefilling.

2. Randomly sample one query $q_m \in \mathbb{R}^{1 \times d}, m \in [\lfloor \frac{n}{2} \rfloor, n)$, and then accordingly sample all the keys that comply with the causal constraint: $K_m = [k_1, k_2 \dots k_m] \in \mathbb{R}^{m \times d}$. Then we form m qk pairs $\{(q_m, k_1), (q_m, k_2) \dots (q_m, k_m)\}$.

3. Compute qk score $Score = q_m K_m^T \in \mathbb{R}^{1 \times m}$ and sort it in descending order.

4. Split the qk pairs into positive and negative samples and assign similarity labels:

   For the qk pairs whose score lies in top 10% of sorted $Score$, we view them as positive samples. They are assigned linearly decayed labels in $[1, 20]$;

   For the qk pairs whose score lies in bottom 90% of sorted $Score$, we view them as negative samples, and assign fixed -1 as their similarity labels.

5. Finally, we get m triplets:

   $$\{(q_m, k_1, s_1), (q_m, k_2, s_2), \dots, (q_m, k_m, s_m)\}$$

   where s_i is the similarity label we assigned in the previous step. A triplet is a basic unit for training. These triplets are independent of each other during training. They can be arbitrarily combined or shuffled along with data sampled from other sequences, which will help improve the generalization of training and avoid overfitting.

After introducing how to collect samples from a single sequence, we clarify from where the sequences are sampled:

- 5 samples from Qasper of LongBench (Bai et al., 2023) for short sequences;
- 2 samples each from LSHT and RepoBench-P of LongBench for medium-length sequences;
- 2 samples from LongBench-v2 (Bai et al., 2024) for ultra-long sequences.

The sampled sequences cover diverse domains including Chinese and English QA, code understanding, ensuring the diversity of training data.

To fit within the model's context window, we truncated some long sequences. The final training set for each model comprises 150K–300K qk pairs.

B.2 Training Setup

In this section, we report the detailed settings of hash training. Firstly, in Table 13, we detail the hyperparameter values during training, which are shared by all the models.

During training, in order to facilitate data IO and shuffling, we organize the sampled data into chunks of 32K size. In each epoch, several chunks (2 for Llama2 and 3 for Llama3.1, Qwen2.5-14B, Qwen2.5-32B) will be loaded for training. Each training epoch will perform multiple iterations on these data. For all the models, 15 epochs and 20 iterations are required to train one layer's hash weights.

Class	Hyper-parameter	Value
Custom Hyperparamters	σ	0.1
	ϵ	0.01
	λ	1.0
	η	2.0
SGD Optimizer Hyperparamters	LR	0.1
	Weight decay	10^{-6}
	Momentum	0.9

Table 13: Hyperparameter values during hash training.

C High-Performance Implementation for Loki

As explained in Sec 5.3, Loki (Singhania et al., 2024) lacks a high-performance implementation. While Loki has provided a kernel fusion of gather and matrix multiplication, their implementation neither integrates with the widely-used FlashAttention2 kernels (Dao, 2024) nor provides efficient end-to-end inference, preventing fair performance comparisons. To address these limitations, we develop a high-performance Loki implementation with these optimizations:

Fuse gather with FlashAttention. We employ the identical high-performance fused gather-FlashAttention kernel for Loki as described in Sec 4, ensuring fair comparison.

High-performance score operator. Similar to HATA's high-performance hamming score operator (see Sec 4), we implemented an optimized scoring operator for Loki. This triton-based (Tillet et al., 2019) kernel computes approximate scores for token selection using the first R channels of PCA-projected query and key vectors, eliminating the redundant memory access overhead of low-rank queries and keys.

Static KVCache. Static KVCache refers to a pre-allocated GPU memory space for storing key-value pairs. During a decoding step, this approach only requires copying the newly generated key-value pair into the allocated space, eliminating the costly tensor concatenation operation, which involves heavy data copy.