VecInfer: Efficient LLM Inference with Low-Bit KV Cache via Outlier-Suppressed Vector Quantization

Dingyu Yao[1,2*] , **Chenxu Yang**[1,2] , **Zhengyang Tong**[1,2] ,**Zheng Lin**[1,2†] , **Wei Liu**[3], **Jian Luan**[3], **Weiping Wang**[1]
[1]Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[2]School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
[3]MiLM Plus, Xiaomi Inc, Beijing, China
{yaodingyu, yangchenxu, linzheng, wangweiping}@iie.ac.cn
{liuwei40, luanjian}@xiaomi.com

Abstract

The Key-Value (KV) cache introduces substantial memory overhead during large language model (LLM) inference. Although existing vector quantization (VQ) methods reduce KV cache usage and provide flexible representational capacity across bit-widths, they suffer severe performance degradation at ultra-low bit-widths due to key cache outliers that hinder effective codebook utilization. To address this challenge, we propose VecInfer, a novel VQ method for aggressive KV cache compression while enabling efficient inference. By applying smooth and Hadamard transformations, VecInfer suppresses outliers in the key cache, enabling the codebook to comprehensively cover the original data distribution and thereby reducing quantization difficulty. To facilitate efficient deployment, we design an optimized CUDA kernel that fuses computation with dequantization to minimize memory access overhead. Extensive evaluations demonstrate that VecInfer consistently outperforms existing quantization baselines across both long-context understanding and mathematical reasoning tasks. With only 2-bit quantization, VecInfer achieves performance comparable to full precision, while delivering up to $\mathbf{2.7}\times$ speedup in large-batch self-attention computation and $\mathbf{8.3}\times$ reduction in single-batch end-to-end latency on Llama-3.1-8B with a 196k sequence length[1].

1 Introduction

Recent transformer-based large language models (LLMs) (Comanici et al., 2025; DeepSeek-AI et al., 2025; Yang et al., 2025a) have achieved remarkable success in long-context tasks, including multi-document understanding (Bai et al., 2024) and complex reasoning (Li et al., 2025). To enable efficient inference, the Key-Value (KV) cache is a critical mechanism that stores previous key and value states to avoid redundant attention computations during autoregressive decoding. However, the KV cache size grows linearly with sequence length, posing significant challenges for efficient LLM inference and serving, particularly in terms of memory consumption and computational overhead.

	KIVI	ZipCache	CQ	MILLION	VecInfer
Quantization Scheme	SQ	SQ	VQ	VQ	VQ
Bitwidth Flexibility	↓	↓	↑	↑	↑
Accuracy @ Low-bit	↑	↑	↓↓	↓↓	↑
Fused Attention	✗	✗	✗	✓	✓
Inference Speed	↓↓	↓↓	↓↓	↑	↑

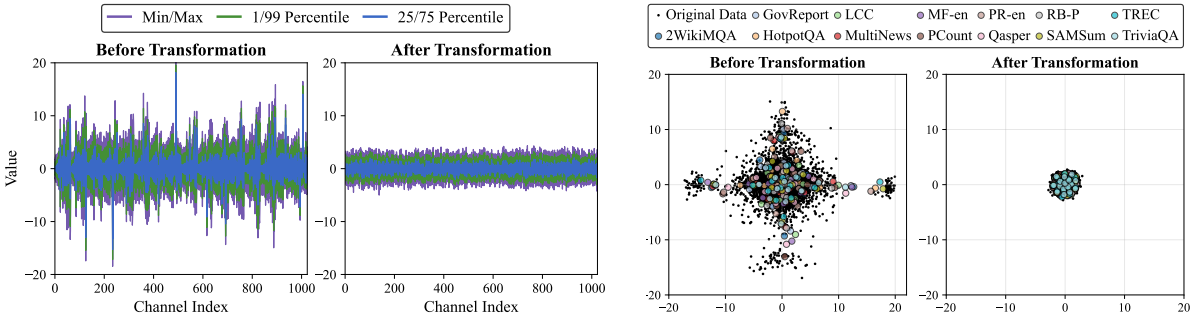
Table 1: Comparison of different KV cache quantization methods across multiple dimensions. VecInfer achieves expected gains in both accuracy and efficiency.

To reduce KV cache usage, quantization has emerged as a promising solution, primarily encompassing scalar quantization (SQ) and vector quantization (VQ). SQ (Hooper et al., 2024; Zirui Liu et al., 2023; He et al., 2024) maps floating-point values to fixed-point integers but offers limited flexibility across bit-widths. In contrast, VQ (Zhang et al., 2024; Wang et al., 2025; Liu et al., 2025) provides greater flexibility by mapping high-dimensional vectors to a finite set of codebook entries, where dequantization is reduced to an efficient lookup operation. Table 1 summarizes representative KV cache quantization methods and compares their strengths and limitations from multiple perspectives. Despite the memory savings, existing methods for low-bit KV cache still struggle to achieve the expected gains in both accuracy and efficiency.

To address these limitations, effective deployment of VQ-based KV cache must tackle two key challenges: ***(i) Lossless accuracy at low bit-widths***: VQ typically quantizes KV cache along the token dimension for hardware compatibility, rendering it highly sensitive to outliers. As shown in Figure 1b, outlier vectors lie far from any codebook

† Corresponding Author: Zheng Lin.
[1]Code is available at: https://github.com/ydyhello/VecInfer

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 31527–31543
July 2-7, 2026



(a) Channel-wise distribution of the key cache.

(b) Codebook representation (b2d4) of the key cache.

Figure 1: Key cache distribution and codebook representation for Llama-3.1-8B-Instruct at layer 16. (a) Dual transformation reduces channel-wise variation and suppresses outliers, resulting in a more uniform distribution. (b) This uniformity **facilitates task-independent codebook representations** and **ensures comprehensive coverage of the original data distribution**.

centroids, and the learned centroids tend to be task-dependent, further increasing quantization difficulty. ***(ii) Hardware-aligned inference speedup***: Performing dequantization before attention computation introduces significant overhead, severely limiting practical speedup. Therefore, realizing actual speed gains requires hardware-friendly kernel designs that minimize memory access and optimize thread scheduling.

Building upon these analyses, we propose VecInfer, a novel VQ-based method for aggressive low-bit KV cache compression. Our approach first applies smooth and Hadamard transformations to the key cache, which reduces quantization difficulty while preserving the computational equivalence between queries and keys. As demonstrated in Figure 1, this dual transformation suppresses outliers and produces a more uniform distribution, facilitating task-independent codebook representations and enabling the codebook to comprehensively cover the original data space. Then we implement a fused dequantization-computation CUDA kernel featuring fine-grained tiled computation and asynchronous pipeline execution to enhance efficiency.

We evaluate VecInfer in terms of both accuracy and efficiency across diverse LLMs, focusing on long-context and mathematical reasoning tasks. Experimental results show that VecInfer consistently outperforms existing baselines under lower bit-width configurations (1.25, 1.5, 2, 3, and 4 bits) and achieves substantial efficiency gains at both the kernel and end-to-end levels. Specifically, for Llama-3.1-8B with 196k sequence length using 2-bit KV cache quantization, VecInfer achieves up to $2.7\times$ speedup on H100 and $2.8\times$ on A100 for large-batch self-attention computation compared to the FP16 counterpart, and reduces single-batch end-to-end latency by $8.3\times$ on H100. Below are the key contributions of our work:

- We identify outliers as a major challenge in VQ and propose **VecInfer**, a novel VQ-based method for KV cache that employs dual transformation to reduce quantization difficulty.
- To enable efficient hardware acceleration, we design a fused dequantization-computation CUDA kernel with fine-grained tiled computation and asynchronous pipeline execution.
- Extensive experiments show that VecInfer outperforms baselines across diverse quantization bit-widths, downstream tasks, and model architectures, while significantly reducing self-attention and end-to-end latency.

2 Preliminaries

2.1 KV Cache and Attention

The KV cache eliminates redundant attention computations by storing key-value states during LLM inference, which consists of prefilling and decoding phases. During *prefilling*, the prompt is processed to produce the first output token and initialize the KV cache with key-value states $\mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times D}$, where N is the number of prompt tokens and D is the dim of attention. During *decoding*, the LLM performs autoregressive generation, producing the output sequence token by token. For the current input states $\mathbf{q}, \mathbf{k}, \mathbf{v} \in \mathbb{R}^{1 \times D}$, the KV cache is updated as $\mathbf{K} \leftarrow \mathrm{Concat}(\mathbf{K}, \mathbf{k})$ and $\mathbf{V} \leftarrow \mathrm{Concat}(\mathbf{V}, \mathbf{v})$. The self-attention mechanism captures connections among all tokens in the context through the KV

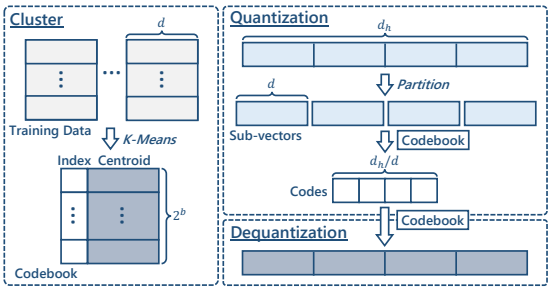


Figure 2: Typical vector quantization pipeline.

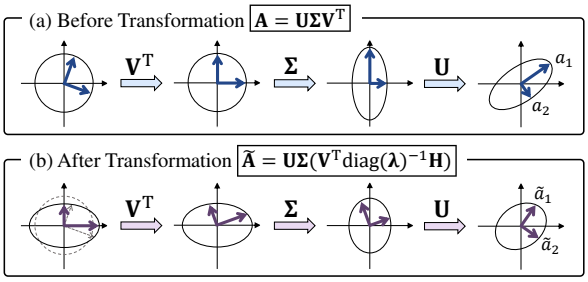


Figure 3: Transformation from $\mathbf{V}^\top$ to $\mathbf{A}$ via SVD.

cache, computing the attention output as:

$$\mathbf{s} = \mathbf{q}\mathbf{K}^\top/\sqrt{D},\ \mathbf{p} = \mathrm{softmax}(\mathbf{s}),\ \mathbf{o} = \mathbf{p}\mathbf{V}. \quad (1)$$

FlashAttention (Dao et al., 2022; Dao, 2024) is an IO-aware algorithm that reduces the memory overhead in the attention through tiling, re-computation, and online softmax operation.

2.2 Vector Quantization

VQ (Jegou et al., 2010) maps continuous vector spaces to a finite set of representative codebook vectors, treating each vector as a quantization unit. As shown in Figure 2, VQ employs K-Means to construct a codebook $\mathcal{C}$ comprising 2^b centroids, each with d dimensions. Given a d_h-dimensional vector $\mathbf{x} \in \mathbb{R}^{d_h}$, VQ partitions it into d_h/d disjoint sub-vectors: $[\mathbf{x}_1, \ldots, \mathbf{x}_i, \ldots, \mathbf{x}_{d_h/d}]$. Each sub-vector $\mathbf{x}_i \in \mathbb{R}^d$ is then assigned the index of its nearest centroid in $\mathcal{C}$, with the corresponding centroid index encoded as a b-bit representation:

$$j^* = \underset{j\in\{1,\ldots,2^b\}}{\arg\min} \|\mathbf{x}_i - \mathcal{C}_j\|^2,\ \mathrm{VQ}(\mathbf{x}_i, \mathcal{C}) = j^*. \quad (2)$$

These sub-vector indices are then combined to form a compressed representation of the original vector $\mathbf{x}$. VQ significantly reduces memory usage. Rather than storing the full d_h-dimensional vector using $d_h \times 2$ bytes (assuming 16-bit floating-point precision), VQ requires only $2^b \times d \times 2$ bytes for the codebook plus $(d_h/d) \times (b/8)$ bytes for the indices. However, during model inference, since the quantized data contains only codebook indices, direct arithmetic operations are not possible. Therefore, dequantization must be performed before each computation by retrieving the corresponding centroid from the codebook using the stored index.

3 Methodology

3.1 Rethinking Quantization Difficulty

While prior studies (Wang et al., 2025; Zhang et al., 2024) show that VQ alleviates outlier issues compared to SQ, this advantage is limited in practice. As shown in Figure 1b (left), outlier vectors remain distant from any codebook centroids, and these learned centroids exhibit highly task-dependence. Consequently, codebook entries are underutilized, which increases quantization difficulty.

Inspired by computational invariance in weight-activation transformations (Xiao et al., 2023; Ashkboos et al., 2024), we study how smooth and Hadamard transformations reduce key cache quantization difficulty while ensuring computational invariance between queries and keys. To analyze the effects of transformations, we employ singular value decomposition (SVD), which factorizes a matrix as $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$, where the orthogonal matrices $\mathbf{U}$ and $\mathbf{V}$ are rotations, and the diagonal matrix $\mathbf{\Sigma}$ is stretch (Lee et al., 2024). Figure 3(a) shows how the column vectors of $\mathbf{V}^\top$ are rotated and stretched to form the column vectors a_1 and a_2 of $\mathbf{A}$, representing the maximum and minimum values, respectively. Figure 3(b) shows that combining smooth and Hadamard transformations reduces the magnitude gap between $\tilde{a}_1$ and $\tilde{a}_2$, resulting in an outlier-free distribution.

As shown in Figure 1, the proposed dual transformation reduces channel-wise variation and suppresses outliers, producing a more uniform distribution. This uniformity facilitates task-independent codebook representations and ensures comprehensive coverage of the original data distribution. Notably, applying these transformations individually leads to sub-optimal uniformity, with full details provided in Appendix C.

3.2 Outlier-Suppressed Vector Quantization

Building on the above analyses, we propose VecInfer, a VQ-based KV cache compression method that applies smooth and Hadamard transformations before quantization to suppress key cache outliers and reduce quantization difficulty. The overall pipeline is illustrated in Figure 4.

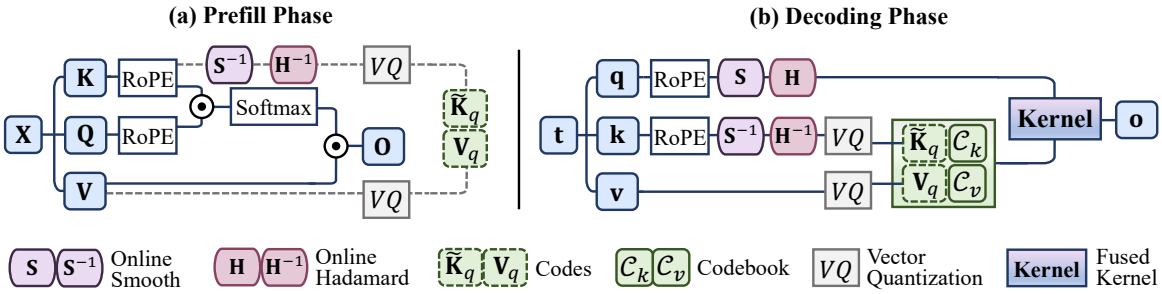


Figure 4: Overview of VecInfer. During inference, dual transformation is applied before vector quantization.

Dual Equivalent Transformation. We first smooth the keys via channel-wise scaling using a factor $\boldsymbol{\lambda} \in \mathbb{R}^D$, and apply the inverse scaling to the queries to preserve computational invariance in the query–key multiplication:

$$\mathbf{q} \leftarrow \mathbf{q}\text{diag}(\boldsymbol{\lambda}),\ \mathbf{K} \leftarrow \mathbf{K}\text{diag}(\boldsymbol{\lambda})^{-1}. \quad (3)$$

Here, the scaling factor is pre-computed offline from calibration samples and defined as:

$$\boldsymbol{\lambda}_i = \sqrt{\max(|\mathbf{K}_i|)},\ i = 1, 2, \ldots, D, \quad (4)$$

where $\mathbf{K}_i$ is the i-th channel of $\mathbf{K}$.

Since the smooth transformation reduces inter-channel variance without addressing intra-channel variance, significant percentile fluctuations persist (Figure 10(b)). To further suppress outliers, we apply an orthogonal Hadamard matrix $\mathbf{H}_D$ satisfying $\mathbf{H}_D\mathbf{H}_D^\top = \mathbf{I}$. For $D = 2^k$, the Walsh–Hadamard matrix is defined recursively as:

$$\mathbf{H}_{2^k} = \frac{1}{\sqrt{2}}\begin{bmatrix} \mathbf{H}_{2^{k-1}} & \mathbf{H}_{2^{k-1}} \\ \mathbf{H}_{2^{k-1}} & -\mathbf{H}_{2^{k-1}} \end{bmatrix}, \mathbf{H}_1 = \begin{bmatrix} 1 \end{bmatrix}. \quad (5)$$

By multiplying both queries and keys with $\mathbf{H}_D$, we ensure computational invariance between them:

$$\mathbf{q} \leftarrow \mathbf{q}\mathbf{H}_D,\ \mathbf{K} \leftarrow \mathbf{K}\mathbf{H}_D. \quad (6)$$

Lemma 1 (Hadamard). *For key states $\mathbf{K} \in \mathbb{R}^{N\times D}$ with $\text{sign}(K_{i,j}) \overset{i.i.d.}{\sim} \text{Uniform}\{-1, +1\}$, and a Hadamard matrix $\mathbf{H} \in \mathbb{R}^{D\times D}$ constructed as in Equation (5), the transformed matrix $\tilde{\mathbf{K}} = \mathbf{K}\mathbf{H}$ exhibits approximately Gaussian distribution by the central limit theorem, thereby redistributing the outliers of $\mathbf{K}$.*

Lemma 1 suggests that the Hadamard rotation effectively redistributes outliers across neighboring elements, yielding a more uniform distribution and further reducing the difficulty of quantization.

In summary, after smooth and Hadamard transformations, the attention score can be rewritten as:

$$\mathbf{s} = (\underbrace{\mathbf{q}\text{diag}(\boldsymbol{\lambda})\mathbf{H}_D}_{\tilde{\mathbf{q}}}) \cdot (\underbrace{\mathbf{K}\text{diag}(\boldsymbol{\lambda})^{-1}\mathbf{H}_D}_{\tilde{\mathbf{K}}})^\top. \quad (7)$$

Vector-Quantized KV Cache in Attention. To seamlessly integrate VQ into attention, we pre-sample outlier-suppressed keys and pre-train the codebook via K-Means (see Figure 2). As shown in Figure 4, during prefilling, a dual transformation is applied to the keys, followed by VQ on the transformed keys $\tilde{\mathbf{K}}$ and original values $\mathbf{V}$:

$$\tilde{\mathbf{K}}_q = \text{VQ}(\tilde{\mathbf{K}}, \mathcal{C}_k),\ \mathbf{V}_q = \text{VQ}(\mathbf{V}, \mathcal{C}_v), \quad (8)$$

where $\text{VQ}(\cdot)$ denotes the vector quantization function defined in Equation (2), and $\mathcal{C}_k, \mathcal{C}_v$ are the codebooks for keys and values, respectively.

During decoding, each newly arrived set of keys $\mathbf{k}$ undergoes online dual transformation. The transformed keys $\tilde{\mathbf{k}}$ and their corresponding values $\mathbf{v}$ are then quantized using the pre-trained codebooks. The quantized results are subsequently concatenated with previously quantized pairs:

$$\begin{aligned} \tilde{\mathbf{K}}_q &\leftarrow \text{Concat}(\tilde{\mathbf{K}}_q, \text{VQ}(\tilde{\mathbf{k}}, \mathcal{C}_k)), \\ \mathbf{V}_q &\leftarrow \text{Concat}(\mathbf{V}_q, \text{VQ}(\mathbf{v}, \mathcal{C}_v)). \end{aligned} \quad (9)$$

For output consistency, we apply the inverse transformation to queries $\mathbf{q}$. Denote the dequantization operator by $\text{VQ}^{-1}(\cdot)$. The attention computation is then given by:

$$\begin{aligned} \mathbf{s} &= \tilde{\mathbf{q}}(\text{VQ}^{-1}(\tilde{\mathbf{K}}_q, \mathcal{C}_k))^\top / \sqrt{D}, \\ \mathbf{p} &= \text{softmax}(\mathbf{s}),\ \mathbf{o} = \mathbf{p}(\text{VQ}^{-1}(\mathbf{V}_q, \mathcal{C}_v)). \end{aligned} \quad (10)$$

Notably, even after transformations, keys exhibit higher quantization sensitivity than values (see Appendix D for details). To preserve accuracy, higher bit-widths can be allocated to the keys.

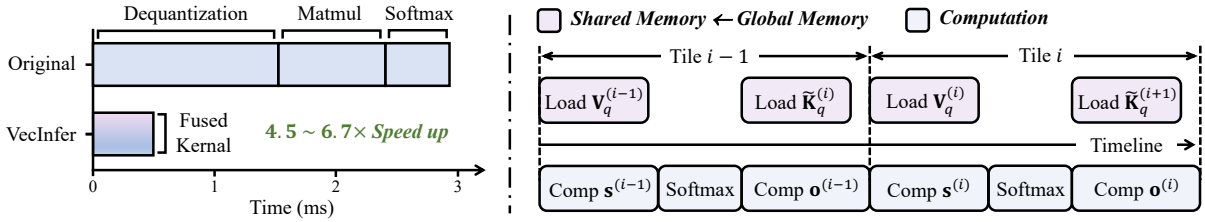


Figure 5: **Left**: Attention kernel speed comparison between VecInfer and the non-fused baseline on H100. **Right**: Workflow of the VecInfer kernel with fine-grained tiled computation and asynchronous pipeline execution.

3.3 Hardware-efficient Customized Kernel

During autoregressive decoding, each newly generated token requires dequantization of the low-bit KV cache, which introduces substantial overhead and complexity. To address this challenge, we propose a hardware-aligned kernel that fuses dequantization with attention computation. By minimizing global memory accesses, our kernel runs faster than the non-fused baseline (Figure 6 left). Figure 6 (right) illustrates the VecInfer kernel workflow, and the architecture incorporates the following key optimizations:

1. **Fine-Grained Tiled Computation.** Our implementation partitions the attention computation into tiles and loads them from global memory into shared memory, effectively mitigating the memory bandwidth bottleneck. Specifically, we adopt a three-dimensional grid configuration of (*batch_size, num_heads, num_splits*), where each thread block contains 128 threads that collectively process a single tile of quantized key-value pairs to compute the corresponding partial attention output.

2. **Asynchronous Pipeline Execution.** Our objective is to transfer quantized key-value pairs from global memory to shared memory for efficient access. To fully utilize CUDA cores, we leverage the `memcpy_async` API to overlap memory transfers with computation. During processing of the i-th tile, we asynchronously load value codes $\mathbf{V}_q^{(i)}$ while computing $\mathbf{s}^{(i)}$. Subsequently, during the computation of $\mathbf{o}^{(i)}$, we asynchronously prefetch key codes $\tilde{\mathbf{K}}_q^{(i+1)}$ for the next tile.

Additionally, we optimize the shared memory layout of key and value codes to minimize bank conflicts and improve throughput. The complete kernel algorithm is detailed in Appendix B.1. As shown in Figure 6, our optimized kernel achieves

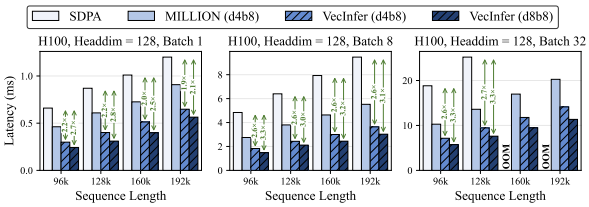


Figure 6: Kernel performance on H100 (80GB). Additional results are provided in Figure 9.

$2.6 \sim 3.3\times$ speedup over vanilla full attention on H100 for large-batch self-attention.

4 Experiments

4.1 Experimental Setup

Models and Tasks. In this paper, we conduct experiments across a diverse range of LLMs, including Llama-3.1-8B-Instruct, Mistral-7B-Instruct-v0.3, Qwen2.5-14B-Instruct, DeepSeek-R1-Distill-Llama-8B, and DeepSeek-R1-Distill-Qwen-14B, Qwen3-8B. To assess long-context performance, we evaluate VecInfer on 13 tasks from LongBench (Bai et al., 2024), which spans six categories: single/multi-document question answering, summarization, few-shot learning, code completion, and synthetic tasks. To evaluate reasoning ability, we use three datasets: GSM8K (Cobbe et al., 2021), MATH500 (Hendrycks et al., 2021), AIME24, and AMC2023. We follow the recommended sampling parameters, setting the temperature to 0.6 and the top-p to 0.95. The evaluation metric employed is Pass@1 accuracy. For GSM8K, MATH500, and AMC2023, the maximum output length is set to 16,384 tokens, while for AIME24, the maximum output length is set to 32,768 tokens. Refer to Appendix F for further details.

Baselines. We evaluate VecInfer against two representative baselines: KIVI (Zirui Liu et al., 2023) (scalar quantization) and MILLION (Wang et al., 2025) (vector quantization). We adopt the notation bngm for KIVI, where the KV cache is quantized

Method	Avg. bit	Config	SD.QA		MD.QA		Sum.		FS.L			Code		Synth.		Avg.
			Qspr	MulF	HQA	WMQA	GRpt	MulN	TREC	SMSM	TriQA	Repo	LCC	PsgC	PsgR	
Llama-3.1-8B	16	-	45.9	53.8	55.2	46.6	34.6	27.5	72.5	43.8	91.6	56.4	63.2	8.0	99.5	53.7
KIVI	3	b2g32	44.8	53.7	55.1	46.0	34.5	27.2	72.5	43.5	91.7	55.0	61.9	8.3	98.5	53.3
MILLION	3	d4b12	44.1	50.3	46.6	38.6	33.4	27.0	69.5	41.0	91.2	52.5	55.9	3.1	87.5	49.3
VecInfer	3	d4b12	46.7	53.9	54.8	47.2	34.1	27.3	72.5	43.3	92.1	55.2	62.9	7.6	99.5	**53.6**
KIVI	2.25	b2g128	43.2	51.9	55.5	46.0	32.7	27.0	71.5	43.3	91.4	53.2	59.7	8.1	99.0	52.4
MILLION	2	d4b8	39.4	49.3	52.5	40.0	26.8	24.6	69.0	38.3	91.7	44.9	50.7	7.8	91.5	48.2
VecInfer	2	d4b8	46.1	53.1	55.0	46.5	31.2	26.5	72.0	41.6	92.2	53.2	60.9	7.9	98.5	**52.7**
KIVI	1.5	b1g64	3.4	5.7	5.9	4.9	8.2	9.4	32.2	4.8	17.7	23.3	27.1	4.3	1.8	11.4
MILLION	1.5	d8b12	1.5	7.8	2.8	0.8	6.3	10.9	30.0	6.1	7.8	29.1	20.9	0.6	1.1	9.7
VecInfer	1.5	d8b12	43.7	52.2	54.7	46.2	29.2	26.1	71.0	39.4	91.7	51.4	60.2	7.6	99.5	**51.8**
VecInfer	1.25	K-d8b12/V-d8b8	41.2	49.6	54.2	45.4	25.7	24.2	69.0	39.1	90.8	50.4	57.7	7.2	99.5	**50.3**
Mistral-7B	16	-	38.6	49.7	51.0	36.4	34.3	26.5	76.0	47.6	88.5	60.6	59.4	6.0	96.5	51.5
KIVI	3	b2g32	37.3	48.3	50.3	36.6	34.1	26.3	76.0	47.2	88.4	59.1	57.6	8.0	92.0	50.8
MILLION	3	d4b12	32.4	47.5	44.5	33.0	32.5	26.2	74.5	45.1	86.2	56.7	51.8	4.1	74.5	46.8
VecInfer	3	d4b12	38.4	49.1	50.2	35.6	33.2	26.5	76.0	46.7	89.3	60.9	58.7	3.0	96.5	**51.1**
KIVI	2.25	b2g128	36.4	47.8	49.0	30.9	32.3	26.4	76.0	46.6	89.2	56.9	56.2	4.6	87.0	49.2
MILLION	2	d4b8	30.9	42.8	45.8	28.5	27.5	24.6	70.5	42.8	88.5	48.4	50.8	5.7	64.0	43.9
VecInfer	2	d4b8	36.7	49.1	51.7	34.0	31.3	26.5	76.0	45.4	89.4	60.5	57.7	3.1	92.5	**50.3**
KIVI	1.5	b1g64	3.8	5.5	5.0	4.1	7.6	6.2	31.7	1.5	6.6	18.9	25.2	1.3	8.4	9.7
MILLION	1.5	d8b12	8.5	18.2	11.2	8.2	15.7	20.9	54.0	15.8	27.1	26.7	27.0	2.1	3.8	18.4
VecInfer	1.5	d8b12	33.2	45.4	48.2	32.1	27.2	25.2	73.0	44.4	88.2	58.3	56.0	5.0	87.5	**48.0**
VecInfer	1.25	K-d8b12/V-d8b8	30.4	41.4	48.2	32.3	23.7	23.8	69.0	41.9	88.0	55.8	54.9	3.7	90.5	**46.5**
Qwen2.5-14B	16	-	45.4	53.9	61.6	57.9	29.7	21.9	76.5	47.7	90.1	48.8	61.3	9.0	98.6	54.0
KIVI	3	b2g32	46.4	51.5	61.5	58.4	28.8	21.5	77.0	47.2	89.7	48.5	61.4	10.4	92.0	53.4
MILLION	3	d4b12	31.9	50.0	56.2	47.4	27.2	21.0	76.0	46.7	90.5	40.1	45.9	2.4	72.5	46.8
VecInfer	3	d4b12	46.1	52.2	61.2	58.0	29.0	21.8	77.0	47.3	89.7	49.1	61.8	9.6	97.0	**53.9**
KIVI	2.25	b2g128	41.6	49.2	61.8	58.2	27.4	21.4	76.5	46.9	90.2	46.7	60.0	11.7	74.6	51.2
MILLION	2	d4b8	24.8	45.0	57.2	48.3	23.1	20.7	72.5	41.5	86.4	37.2	40.8	15.0	58.5	43.9
VecInfer	2	d4b8	43.7	51.3	59.6	57.4	26.8	21.3	77.0	46.2	90.1	46.7	60.7	13.0	90.8	**52.7**
KIVI	1.5	b1g64	3.4	4.1	3.5	3.1	10.1	7.9	30.9	5.5	11.6	23.3	23.4	1.8	2.8	10.1
MILLION	1.5	d8b12	3.9	3.9	0.9	0.9	7.2	8.4	42.0	11.6	5.3	28.6	21.3	0.5	0.0	10.3
VecInfer	1.5	d8b12	38.2	45.3	58.2	54.9	24.0	20.5	72.5	42.8	89.4	43.6	57.5	9.0	88.5	**49.6**
VecInfer	1.25	K-d8b12/V-d8b8	33.8	42.7	58.1	50.5	20.5	18.4	68.0	40.3	88.3	42.0	54.4	9.5	84.8	**47.0**

Table 2: The evaluation accuracy results on LongBench under different quantization configurations.

to n-bit precision with a group size of m. Both MILLION and VecInfer settings follow the notation dnbm, where each sub-vector has dimension n and codes are encoded using m bits. Additionally, the residual length for all methods is set to 128.

Implementation Details. The smoothing factors are calibrated offline using 256 random samples from the Pile dataset (Gao et al., 2020), each consisting of 512 tokens. This calibration process is efficient, requiring only a few seconds on an H100 GPU. The codebook is pre-trained on the Qasper dataset using K-means clustering, with the maximum number of iterations set to 30. Appendix E.1 demonstrates that the codebook trained by our method is task-independent.

4.2 Accuracy Evaluation

Long Context Tasks. To evaluate VecInfer's long-context performance, we conduct experiments on 13 datasets from LongBench, quantizing the KV cache under different quantization configurations. As shown in Table 2, VecInfer consistently outperforms baselines across the average precision range of 1.25 to 4 bits, thereby demonstrating the effectiveness of the proposed dual transformation. Furthermore, when using 2-bit precision for KV cache storage, VecInfer demonstrates only a 2.1% average accuracy drop, while achieving an 14.5% average performance improvement compared to MILLION, another VQ-based method.

Complex Reasoning Tasks. We evaluate the performance of long-CoT LLMs on mathematical reasoning tasks (Yang et al., 2025b,c), as shown in Table 3. When precision is reduced to 2-bit, both KIVI and MILLION experience significant performance degradation, failing to generate coherent responses. In contrast, VecInfer shows minimal performance degradation on complex reasoning tasks. We also find that model type and task diffi-

Method	Avg. bit	Config	DS-R1-Distill-Llama-8B				DS-R1-Distill-Qwen-14B				Qwen3-8B			
			MATH500	GSM8K	AIME24	AMC	MATH500	GSM8K	AIME24	AMC	MATH500	GSM8K	AIME24	AMC
Baseline	16	-	86.6	90.4	47.5	86.8	92.6	95.7	66.2	93.1	94.0	96.0	72.9	90.0
KIVI	4.25	b4g128	87.8	90.1	45.8	86.8	93.6	95.4	**67.1**	92.8	93.8	96.1	72.2	90.0
MILLION	4	d2b8	86.8	89.7	40.4	85.6	92.6	94.6	55.0	90.9	46.9	76.2	8.9	24.3
VecInfer	4	d2b8	**88.2**	**90.9**	**46.2**	**87.2**	**93.6**	**95.5**	66.7	**92.9**	**94.0**	**96.1**	**73.9**	**90.6**
KIVI	3	b2g32	86.8	88.2	37.1	81.8	92.0	**95.7**	62.0	90.3	93.4	94.5	72.5	86.8
MILLION	3	d4b12	86.0	89.0	35.0	79.6	91.4	94.7	47.1	87.8	85.4	94.0	12.7	61.5
VecInfer	3	d4b12	**88.2**	**90.1**	**44.2**	**88.2**	**94.2**	95.0	**67.1**	**91.3**	**93.4**	**95.3**	**75.0**	**90.0**
KIVI	2.25	b2g128	74.6	84.9	16.3	65.9	91.2	94.5	47.5	81.8	88.4	93.1	57.4	84.0
MILLION	2	d4b8	16.8	31.4	0.0	5.4	38.8	60.9	0.0	17.6	11.3	11.2	0.0	12.5
VecInfer	2	K-d4b10/V-d8b12	**80.0**	**87.0**	**26.7**	**78.5**	**92.6**	**94.7**	**53.8**	**86.3**	**90.6**	**95.6**	**67.1**	**86.3**

Table 3: Performance of large reasoning models on mathematical reasoning tasks. For AIME24 and AMC2023, 8 completions are generated per question.

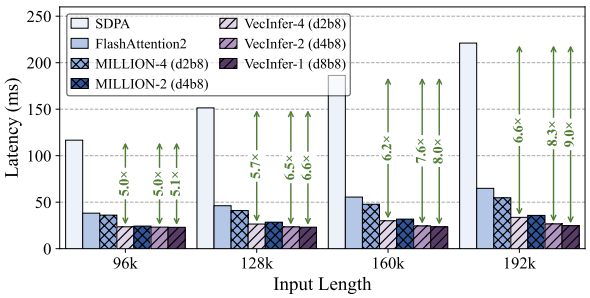


Figure 7: Decoding latency comparison between VecInfer and baselines for Llama-3.1-8B-Instruct on an H100 GPU. Notably, KIVI runs into OOM.

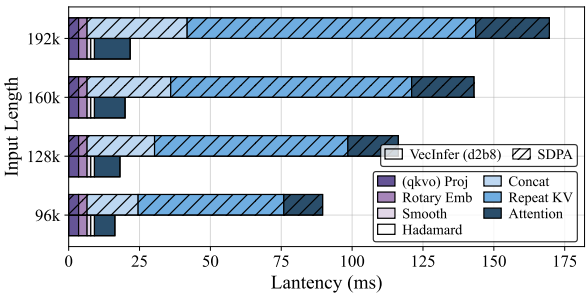


Figure 8: Latency breakdown of attention blocks for Llama-3.1-8B-Instruct on an H100 GPU.

culty significantly influence performance degradation. Specifically, at equivalent compression ratios, DeepSeek-R1-Distill-Qwen-14B and Qwen3-8B exhibit superior quantization tolerance compared to DeepSeek-R1-Distill-Llama-8B, while complex tasks (e.g., AIME24) suffer more substantial performance drops than simpler ones (e.g., GSM8K).

4.3 Efficiency Evaluation

End-to-End Latency. We evaluate VecInfer's end-to-end latency and compare it with several baselines, including Scaled Dot-Product Attention (SDPA), FlashAttention2 (Dao, 2024), and MILLION. At a 64k sequence length, KIVI (Zirui Liu et al., 2023) runs into out-of-memory (OOM) errors due to missing fused kernel support. As shown in Figure 7, VecInfer consistently achieves lower end-to-end latency than previous methods. For instance, at an input length of $l_{input} = 192$k and an output length of $l_{output} = 129$, VecInfer achieves decoding speedups of $9.0\times$, $8.3\times$, and $6.6\times$ under 1-bit, 2-bit, and 4-bit configurations, respectively. Moreover, the speedup advantage grows with sequence length.

Latency Breakdown. Figure 8 shows the latency breakdown of attention blocks across varying input lengths. Relative to SDPA, VecInfer reduces global memory read/write overhead by eliminating costly concatenation and repetition, thereby improving efficiency. At a sequence length of 196k under the 2-bit configuration, VecInfer achieves a $2.0\times$ speedup in self-attention, mainly by fusing all attention operations and dequantization into a single GPU kernel. The additional cost of smooth and Hadamard transformations is negligible, exerting minimal impact on overall performance.

4.4 Ablation Study

Effects of Different Transformations. Table 4 reports the comparative performance of different transformations on LongBench. Starting from VQ-only as the baseline, the smooth and Hadamard transformations independently improve average performance by 4.9% and 14.1%, respectively. Importantly, their combination delivers substantially greater gains than either transformation alone. The sequential application of smooth-then-Hadamard and Hadamard-then-smooth transformations yields comparable performance improvements.

Method	SD.QA	MD.QA	Sum.	FS.L	Code	Synth.	Avg.
Llama-3.1-8B-Instruct							
Original	38.2	47.2	23.6	57.5	51.7	52.6	46.1
S	42.6	48.4	25.0	63.6	51.7	50.8	48.3
H	47.2	49.6	26.1	66.7	55.2	53.6	51.0
H + S	46.8	**50.6**	26.7	67.3	55.1	**53.8**	51.4
S + H	**47.9**	50.5	**27.7**	**67.4**	**55.8**	53.6	**51.8**
Qwen2.5-14B-Instruct							
Original	24.9	50.0	18.7	57.4	46.1	44.7	41.6
S	32.6	52.6	19.7	58.9	48.4	42.0	43.7
H	41.6	56.0	21.2	67.6	48.9	48.6	49.0
H + S	**43.3**	**56.8**	21.4	67.9	49.2	48.6	49.4
S + H	41.8	56.6	**22.3**	**68.3**	**50.6**	**48.9**	**49.6**

Table 4: Ablation study on the effect of different transformations under 1.5-bit, where **S** and **H** represent smooth and Hadamard transformations, respectively.

Avg. bit	Config	SD.QA	MD.QA	Sum.	FS.L	Code	Synth.	Avg.
Llama-3.1-8B-Instruct								
2	d8b16	48.9	50.6	29.3	68.9	58.0	53.5	52.8
	d4b8	49.6	50.8	28.8	68.6	57.0	53.2	52.7
	d2b4	48.8	50.4	29.5	68.8	56.3	53.6	52.6
1.5	d8b12	47.9	50.5	27.7	67.4	55.8	53.6	51.8
	d4b6	46.2	50.4	22.8	67.0	53.8	53.3	50.2
	d2b3	43.7	49.6	20.1	66.7	51.7	53.8	49.1
Mistral-7B-Instruct-v0.3								
2	d8b16	43.2	42.4	28.8	70.4	59.6	49.0	50.5
	d4b8	42.9	42.9	28.9	70.3	59.1	47.9	50.3
	d2b4	42.5	42.3	28.1	69.7	58.6	46.8	49.7
1.5	d8b12	39.3	40.2	26.2	68.6	57.1	46.3	48.0
	d4b6	38.2	38.4	24.4	67.4	56.6	47.8	47.1
	d2b3	37.1	37.6	22.5	64.1	52.7	47.6	45.1

Table 5: Ablation study on the effect of codebook size.

Effects of Codebook Size. Codebook size directly influences the representational capacity of VQ and thus affects performance. As previously discussed, codebook size is calculated as $2^b \times d \times 2$ bytes. Table 5 indicates that for a given bit-width, increasing codebook size consistently enhances accuracy. However, this improvement comes at the cost of elevated shared memory overhead, which degrades computational efficiency. To balance this accuracy-efficiency trade-off, we adopt codebook sizes of $2^8 \times 4 \times 2$ bytes for 2-bit quantization and $2^{12} \times 8 \times 2$ bytes for 1.5-bit quantization.

5 Related Works

KV Cache Quantization. Existing methods optimize KV cache memory usage through quantization while preserving performance. These methods are typically categorized into SQ and VQ. SQ compresses data by encoding floating-point values as low-bit integers. KIVI (Zirui Liu et al., 2023) reduces quantization errors using per-channel quantization for key and per-token quantization for value. MiKV (Yang et al., 2024), ZipCache (He et al., 2024), and RotateKV (Su et al., 2025) employ mixed-precision per-token quantization, preserving high precision for salient tokens. TailorKV (Yao et al., 2025) identifies that different layers exhibit varying compression preferences and quantizes select quantization-friendly layers. In contrast, VQ encodes high-dimensional vectors using a finite codebook, improving bit utilization by leveraging inter-element correlations. CQ (Zhang et al., 2024) and MILLION (Wang et al., 2025) group multiple channels together for quantization by utilizing cross-channel dependencies. VQ-LLM (Liu et al., 2025) adaptively stores different codebook entries across the GPU's memory hierarchy. However, a key limitation of these methods is that outlier vectors deviate significantly from cluster centroids, substantially impairing quantization accuracy.

Efficient Attention. Beyond quantization, several studies have explored sparse attention mechanisms to enhance efficiency in LLMs. Eviction-based methods, including StreamingLLM (Xiao et al., 2024b), H2O (Zhang et al., 2023), and SnapKV (Li et al., 2024), selectively retain essential key-value pairs while permanently discarding less critical ones. Selection-based methods such as NSA (Yuan et al., 2025), Quest (Tang et al., 2024), and MoBA (Lu et al., 2025) identify the most important token blocks and optimize sparse attention patterns at the block level to enable efficient contiguous memory access. Importantly, these sparse attention techniques are orthogonal to quantization and can be effectively combined to achieve significant reductions in both memory footprint and computational costs during inference. Additionally, FlashAttention (Dao et al., 2022; Dao, 2024) employs a tiling strategy that partitions attention computation into blocks and performs operations within shared memory, thus reducing global memory access overhead.

6 Conclusion

In this paper, we propose VecInfer, a novel VQ method for aggressive KV cache compression while enabling efficient inference. VecInfer employs smooth and Hadamard transformations to suppress outliers in the key cache and improve codebook utilization, thereby reducing quantization difficulty. Extensive experiments show that VecInfer outperforms baselines on long-context and mathematical reasoning tasks. Moreover, by fus-

ing computation and dequantization into a single CUDA kernel, VecInfer significantly reduces attention and end-to-end latency. VecInfer facilitates the deployment of LLMs on resource-constrained GPUs, thereby extending their applicability while maintaining both accuracy and efficiency.

Limitations

The methodology offers multiple directions for further enhancement. First, while combining vector quantization with sparse attention patterns for mixed-precision KV cache compression is a promising approach, the trade-offs between accuracy and efficiency remain to be thoroughly explored. Second, integrating VecInfer seamlessly into existing serving frameworks (e.g., vLLM (Kwon et al., 2023) and SGLang (Zheng et al., 2024)) poses practical challenges, particularly because many frameworks lack native support or flexible APIs for KV cache compression, which can complicate deployment.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (Nos. 62472419, 62472420) and the Enterprise Project (No. E4V06811F3).

References

Saleh Ashkboos, Amirkeivan Mohtashami, Maximilian L. Croci, Bo Li, Pashmina Cameron, Martin Jaggi, Dan Alistarh, Torsten Hoefler, and James Hensman. 2024. Quarot: Outlier-free 4-bit inference in rotated LLMs. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. 2024. LongBench: A bilingual, multitask benchmark for long context understanding. In *Proc. of ACL*, pages 3119–3137. Association for Computational Linguistics.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *Preprint*, arXiv:2110.14168.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, Luke Marris, Sam Petulla, Colin Gaffney, Asaf Aharoni, Nathan Lintz, Tiago Cardal Pais, Henrik Jacobsson, Idan Szpektor, Nan-Jiang Jiang, and 3290 others. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *Preprint*, arXiv:2507.06261.

Tri Dao. 2024. Flashattention-2: Faster attention with better parallelism and work partitioning. In *The Twelfth International Conference on Learning Representations*.

Tri Dao, Daniel Y Fu, Stefano Ermon, Atri Rudra, and Christopher Re. 2022. Flashattention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, and 81 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *CoRR*, abs/2501.12948.

Shichen Dong, Wen Cheng, Jiayu Qin, and Wei Wang. 2024. Qaq: Quality adaptive quantization for llm kv cache. *Preprint*, arXiv:2403.04643.

Dayou Du, Shijie Cao, Jianyi Cheng, Luo Mai, Ting Cao, and Mao Yang. 2025. Bitdecoding: Unlocking tensor cores for long-context llms with low-bit kv cache. *Preprint*, arXiv:2503.18773.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. 2020. The pile: An 800gb dataset of diverse text for language modeling. *Preprint*, arXiv:2101.00027.

Naibin Gu, Peng Fu, Xiyu Liu, Ke Ma, Zheng Lin, and Weiping Wang. 2025. Adapt once, thrive with updates: Transferable parameter-efficient fine-tuning on evolving base models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14765–14783, Vienna, Austria. Association for Computational Linguistics.

Naibin Gu, Peng Fu, Xiyu Liu, Bowen Shen, Zheng Lin, and Weiping Wang. 2024. Light-PEFT: Lightening parameter-efficient fine-tuning via early pruning. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 7528–7541, Bangkok, Thailand. Association for Computational Linguistics.

Yefei He, Luoming Zhang, Weijia Wu, Jing Liu, Hong Zhou, and Bohan Zhuang. 2024. Zipcache: Accurate and efficient KV cache quantization with salient token identification. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset. *Preprint*, arXiv:2103.03874.

Ke Hong, Guohao Dai, Jiaming Xu, Qiuli Mao, Xiuhong Li, Jun Liu, Kangdi Chen, Yuhan Dong, and Yu Wang. 2024. Flashdecoding++: Faster large language model inference on gpus. *Preprint*, arXiv:2311.01282.

Coleman Richard Charles Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W. Mahoney, Sophia Shao, Kurt Keutzer, and Amir Gholami. 2024. KVQuant: Towards 10 million context length LLM inference with KV cache quantization. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2010. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128.

Huiqiang Jiang, YUCHENG LI, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Zhenhua Han, Amir H. Abdi, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2024. MInference 1.0: Accelerating pre-filling for long-context LLMs via dynamic sparse attention. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. *Preprint*, arXiv:2309.06180.

Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. Infinigen: efficient generative inference of large language models with dynamic kv cache management. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation*, OSDI'24, USA. USENIX Association.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. 2024. SnapKV: LLM knows what you are looking for before generation. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Zhong-Zhi Li, Duzhen Zhang, Ming-Liang Zhang, Jiaxin Zhang, Zengyan Liu, Yuxuan Yao, Haotian Xu, Junhao Zheng, Pei-Jie Wang, Xiuyi Chen, Yingying Zhang, Fei Yin, Jiahua Dong, Zhiwei Li, Bao-Long Bi, Ling-Rui Mei, Junfeng Fang, Xiao Liang, Zhijiang Guo, and 2 others. 2025. From system 1 to system 2: A survey of reasoning large language models. *Preprint*, arXiv:2502.17419.

Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. 2023. Scissorhands: Exploiting the persistence of importance hypothesis for LLM KV cache compression at test time. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Zihan Liu, Xinhao Luo, Junxian Guo, Wentao Ni, Yangjie Zhou, Yue Guan, Cong Guo, Weihao Cui, Yu Feng, Minyi Guo, Yuhao Zhu, Minjia Zhang, Chen Jin, and Jingwen Leng. 2025. Vq-llm: High-performance code generation for vector quantization augmented llm inference. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1496–1509.

Enzhe Lu, Zhejun Jiang, Jingyuan Liu, Yulun Du, Tao Jiang, Chao Hong, Shaowei Liu, Weiran He, Enming Yuan, Yuzhi Wang, Zhiqi Huang, Huan Yuan, Suting Xu, Xinran Xu, Guokun Lai, Yanru Chen, Huabin Zheng, Junjie Yan, Jianlin Su, and 6 others. 2025. Moba: Mixture of block attention for long-context llms. *Preprint*, arXiv:2502.13189.

Maxim Milakov and Natalia Gimelshein. 2018. Online normalizer calculation for softmax. *Preprint*, arXiv:1805.02867.

Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. 2024. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *Preprint*, arXiv:2407.08608.

Zunhai Su, Zhe Chen, Wang Shen, Hanyu Wei, Linge Li, Huangqi Yu, and Kehong Yuan. 2025. Rotatekv: Accurate and robust 2-bit kv cache quantization for llms via outlier-aware adaptive rotations. *Preprint*, arXiv:2501.16383.

Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. 2024. QUEST: Query-aware sparsity for efficient long-context LLM inference. In *Forty-first International Conference on Machine Learning*.

Zongwu Wang, Peng Xu, Fangxin Liu, Yiwei Hu, Qingxiao Sun, Gezi Li, Cheng Li, Xuan Wang, Li Jiang, and Haibing Guan. 2025. Million: Mastering long-context llm inference via outlier-immunized kv product quantization. *Preprint*, arXiv:2504.03661.

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

Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. 2023. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*, pages 38087–38099. PMLR.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2024b. Efficient streaming

language models with attention sinks. In *The Twelfth International Conference on Learning Representations*.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025a. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Chenxu Yang, Qingyi Si, Mz Dai, Dingyu Yao, Mingyu Zheng, Minghui Chen, Zheng Lin, and Weiping Wang. 2025b. Test-time prompt intervention. *Preprint*, arXiv:2508.02511.

Chenxu Yang, Qingyi Si, Yongjie Duan, Zheliang Zhu, Chenyu Zhu, Qiaowei Li, Zheng Lin, Li Cao, and Weiping Wang. 2025c. Dynamic early exit in reasoning models. *Preprint*, arXiv:2504.15895.

June Yong Yang, Byeongwook Kim, Jeongin Bae, Beomseok Kwon, Gunho Park, Eunho Yang, Se Jung Kwon, and Dongsoo Lee. 2024. No token left behind: Reliable kv cache compression via importance-aware mixed precision quantization. *ArXiv preprint*, abs/2402.18096.

Dingyu Yao, Bowen Shen, Zheng Lin, Wei Liu, Jian Luan, Bin Wang, and Weiping Wang. 2025. TailorKV: A hybrid framework for long-context inference via tailored KV cache optimization. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 20340–20359, Vienna, Austria. Association for Computational Linguistics.

Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, Yuxing Wei, Lean Wang, Zhiping Xiao, Yuqing Wang, Chong Ruan, Ming Zhang, Wenfeng Liang, and Wangding Zeng. 2025. Native sparse attention: Hardware-aligned and natively trainable sparse attention. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 23078–23097, Vienna, Austria. Association for Computational Linguistics.

Hui Zeng, Daming Zhao, Pengfei Yang, WenXuan Hou, Tianyang Zheng, Hui Li, Weiye Ji, and Jidong Zhai. 2025. Lethe: Layer- and time-adaptive kv cache pruning for reasoning-intensive llm serving. *Preprint*, arXiv:2511.06029.

Jintao Zhang, Haofeng Huang, Pengle Zhang, Jia wei, Jun Zhu, and Jianfei Chen. 2025a. Sageattention2: Efficient attention with thorough outlier smoothing and per-thread INT4 quantization. In *Forty-second International Conference on Machine Learning*.

Jintao Zhang, Rundong Su, Chunyu Liu, Jia Wei, Ziteng Wang, Pengle Zhang, Haoxu Wang, Huiqiang Jiang, Haofeng Huang, Chendong Xiang, and 1 others. A survey of efficient attention methods: Hardware-efficient, sparse, compact, and linear attention.

Jintao Zhang, Jia Wei, Pengle Zhang, Xiaoming Xu, Haofeng Huang, Haoxu Wang, Kai Jiang, Jun Zhu, and Jianfei Chen. 2025b. Sageattention3: Microscaling fp4 attention for inference and an exploration of 8-bit training. *Preprint*, arXiv:2505.11594.

Jintao Zhang, Jia wei, Pengle Zhang, Jun Zhu, and Jianfei Chen. 2025c. Sageattention: Accurate 8-bit attention for plug-and-play inference acceleration. In *The Thirteenth International Conference on Learning Representations*.

Tianyi Zhang, Jonah Wonkyu Yi, Zhaozhuo Xu, and Anshumali Shrivastava. 2024. KV cache is 1 bit per channel: Efficient large language model inference with coupled quantization. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark W. Barrett, Zhangyang Wang, and Beidi Chen. 2023. H2O: heavy-hitter oracle for efficient generative inference of large language models. In *Proc. of NeurIPS*.

Tianchen Zhao, Ke Hong, Xinhao Yang, Xuefeng Xiao, Huixia Li, Feng Ling, Ruiqi Xie, Siqi Chen, Hongyu Zhu, Yichong Zhang, and Yu Wang. 2025. Paroattention: Pattern-aware reordering for efficient sparse and quantized attention in visual generation models. *Preprint*, arXiv:2506.16054.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. Sglang: Efficient execution of structured language model programs. *Preprint*, arXiv:2312.07104.

Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. 2023. Kivi : Plug-and-play 2bit kv cache quantization with streaming asymmetric quantization.

A Related Works

A.1 FlashAttention

The query, key, and value matrices $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ are defined with dimensions $N \times D$, where N denotes the sequence length and D represents the dimension of attention. The self-attention computation is formulated as follows:

$$\mathbf{S} = \mathbf{Q}\mathbf{K}^\top / \sqrt{D},\ \mathbf{P} = \mathrm{softmax}(\mathbf{S}),\ \mathbf{O} = \mathbf{P}\mathbf{V}. \tag{11}$$

Standard attention implementation involves computing large intermediate matrices, specifically the $N \times N$ matrices $(\mathbf{S}, \mathbf{P})$, which need to be stored in the global memory. Due to the limited bandwidth and high latency of global memory access, standard attention incurs significant memory I/O overhead when reading and writing $(\mathbf{S}, \mathbf{P})$.

FlashAttention (Dao et al., 2022) is an IO-aware technique designed to reduce memory overhead in attention operations. It leverages online softmax to process the input matrices $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ in tiles, performing block-wise computation within fast shared memory.

FlashAttention2 (Dao, 2024) improves GPU resource utilization by implementing optimized parallelization strategies. In contrast to FlashAttention, which performs parallelization across both the batch and head dimensions, FlashAttention2 focuses on parallelizing the query length dimension. Additionally, the computation is restructured by placing $\mathbf{Q}$ in the outer loop, while $\mathbf{K}$ and $\mathbf{V}$ are placed in the inner loop. FlashAttention-2 tiles $\mathbf{Q} \in \mathbb{R}^{N \times D}$ along the token dimension into blocks $\mathbf{Q}_i$ of size $B_q \times D$, resulting in $T_q = N/B_q$ blocks in total. Similarly, $\mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times D}$ are partitioned along the token dimension into blocks $\mathbf{K}_i$ and $\mathbf{V}_i$ of size $B_{kv} \times D$, yielding $T_{kv} = N/B_{kv}$ blocks. The computation for each block is performed as follows:

$$\begin{aligned}
&\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^\top / \sqrt{D}, \\
&m_{ij} = \max\{m_{i,j-1}, \mathrm{rowmax}(\mathbf{S}_{ij})\}, \\
&\widetilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - m_{ij}), \\
&\ell_{ij} = e^{m_{i,j-1} - m_{ij}} \ell_{i,j-1} + \mathrm{rowsum}(\widetilde{\mathbf{P}}_{ij}), \\
&\mathbf{O}_{ij} = \mathrm{diag}(e^{m_{i,j-1} - m_{ij}}) \mathbf{O}_{i,j-1} + \widetilde{\mathbf{P}}_{ij} \mathbf{V}_j,
\end{aligned} \tag{12}$$

where $(\mathbf{S}_{ij}, \widetilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_q \times B_{kv}}$, $(m_{ij}, \ell_{ij}) \in \mathbb{R}^{B_q}$, and $\mathbf{O}_{ij} \in \mathbb{R}^{B_q \times D}$. Finally, the output $\mathbf{O}_i$ is computed as follows:

$$\mathbf{O}_i = \mathrm{diag}(\ell_{i,T_{kv}})^{-1} \mathbf{O}_{i,T_{kv}}. \tag{13}$$

FlashAttention3 (Shah et al., 2024) further enhances attention computation by leveraging the architectural features of Hopper GPUs. FlashAttention3 employs three key optimizations. First, it adopts producer–consumer warp specialization, assigning distinct warp groups to data loading and computation in order to hide transfer latency. Second, it interleaves block-wise GEMMs with online softmax operations. Third, it utilizes `FP8` hardware support to perform block quantization on the $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ matrices.

A.2 Attention Varients

LLM inference proceeds in two stages (Zhang et al.; Gu et al., 2024, 2025; Zeng et al., 2025): *(i) the prefilling phase*, which processes the input prompt to produce the first output token; and *(ii) the decoding phase*, which generates subsequent tokens autoregressively.

For the compute-bound prefilling stage, existing methods focus on maximizing parallel processing capabilities and computational throughput. Building on online softmax (Milakov and Gimelshein, 2018), FlashAttention (Dao et al., 2022; Dao, 2024; Shah et al., 2024) proposes an IO-aware exact attention algorithm that uses tiling to minimize memory transfers between GPU high bandwidth memory and on-chip SRAM. SageAttention (Zhang et al., 2025c,a,b) is an efficient quantization method for attention that enhances the efficiency of attention computation while maintaining precision. MInference (Jiang et al., 2024) introduces hybrid sparse attention computation to accelerate LLM inference in the prefilling phase. PAROAttention (Zhao et al., 2025) designs specialized sparsification and quantization techniques tailored to the unified block-wise pattern. Recent studies (Yuan et al., 2025; Lu et al., 2025) have explored trainable sparse attention to accelerate attention computation.

B Implementation Details of Kernel

B.1 Kernel Fusion

For the memory-bound decoding stage, existing methods focus on accelerating KV cache I/O operations. Some methods (Li et al., 2024; Zhang et al., 2023; Liu et al., 2023) are designed to reduce memory and computation costs through dynamic KV cache pruning. Quest (Tang et al., 2024) and InfLLM (Xiao et al., 2024a) retain the complete KV cache while retrieving only the most important tokens to reduce attention computation. Quanti-

Algorithm 1 Implementation of VecInfer.

1: **Input:** $\mathbf{q} \in \mathbb{R}^{1\times D}$, $\mathbf{K}, \mathbf{V} \in \mathbb{R}^{N\times D}$, $\mathcal{C}_k, \mathcal{C}_v \in \mathbb{R}^{2^b \times \frac{D}{M}}$, block size B.
2: **Preprocessing:** $\tilde{\mathbf{q}} = \mathbf{q}\mathrm{diag}(\boldsymbol{\lambda})\mathbf{H}_D$, $\tilde{\mathbf{K}} = \mathbf{K}\mathrm{diag}(\boldsymbol{\lambda})^{-1}\mathbf{H}_D$. // Dual transformation.
3: **Quantization:** $\tilde{\mathbf{K}}_q = \mathrm{VQ}(\tilde{\mathbf{K}}, \mathcal{C}_k)$, $\mathbf{V}_q = \mathrm{VQ}(\mathbf{V}, \mathcal{C}_v)$. // Vector quantization.
4: Compute $\tilde{\mathbf{q}}' = \mathrm{reshape}(\tilde{\mathbf{q}}, (M, \frac{D}{M}))$, $\mathbf{lut} = \tilde{\mathbf{q}}'\mathcal{C}_k^\top$.
5: Divide $\tilde{\mathbf{K}}_q, \mathbf{V}_q$ into $T = \lceil \frac{N}{B} \rceil$ blocks $\{\tilde{\mathbf{K}}_q^{(i)}\}, \{\mathbf{V}_q^{(i)}\}$.
6: Initialize $\mathbf{o} = (0)_{1\times D} \in \mathbb{R}^{1\times D}$, $\ell = (0), m = (-\infty)$ in SMEM.
7: Load $\mathcal{C}_v$ from GMEM to SMEM.
8: Load $\tilde{\mathbf{K}}_q^{(1)}$ from GMEM to SMEM.
9: **for** $i = 1$ **to** T **do**
10: Prefetch $\mathbf{V}_q^{(i)}$ from GMEM to SMEM. // Asynchronous memory copy operation.
11: Compute $\mathbf{s}^{(i)} = \mathrm{lookup}(\tilde{\mathbf{K}}_q^{(i)}, \mathbf{lut})/\sqrt{D}$.
12: Compute $m^{new} = \max\{m, \mathrm{rowmax}(\mathbf{s}^{(i)})\}$, $\mathbf{p}^{(i)} = \exp(\mathbf{s}^{(i)} - m^{new})$. // Online softmax.
13: Compute $\ell^{new} = \exp(m - m^{new})\ell + \mathrm{rowsum}(\mathbf{p}^{(i)})$.
14: Wait for $\mathbf{V}_q^{(i)}$ to be loaded in SMEM.
15: Prefetch $\tilde{\mathbf{K}}_q^{(i+1)}$ from GMEM to SMEM. // Asynchronous memory copy operation.
16: Compute $\mathbf{o} = \mathrm{diag}(\exp(m - m^{new}))\mathbf{o} + \mathbf{p}^{(i)}\mathrm{VQ}^{-1}(\mathbf{V}_q^{(i)}, \mathcal{C}_v)$.
17: Wait for $\tilde{\mathbf{K}}_q^{(i+1)}$ to be loaded in SMEM.
18: $\ell = \ell^{new}, m = m^{new}$.
19: **end for**
20: Compute $\mathbf{o} = \mathrm{diag}(\ell)^{-1}\mathbf{o}$.
21: Compute $L = m + \log(\ell)$.
22: Write $\mathbf{o}$, L to GMEM.
23: Return the output $\mathbf{o}$ and the logsumexp L.

zation methods compress the KV cache into low-precision representations (Zirui Liu et al., 2023; Hooper et al., 2024), thereby reducing memory overhead. FlashDecoding (Hong et al., 2024), a variant of FlashAttention, is specifically designed to improve efficiency in long-context decoding. To further reduce memory overhead and improve overall efficiency, BitDecoding (Du et al., 2025) enables efficient low-bit KV cache decoding by cooperatively leveraging CUDA Cores and Tensor Cores.

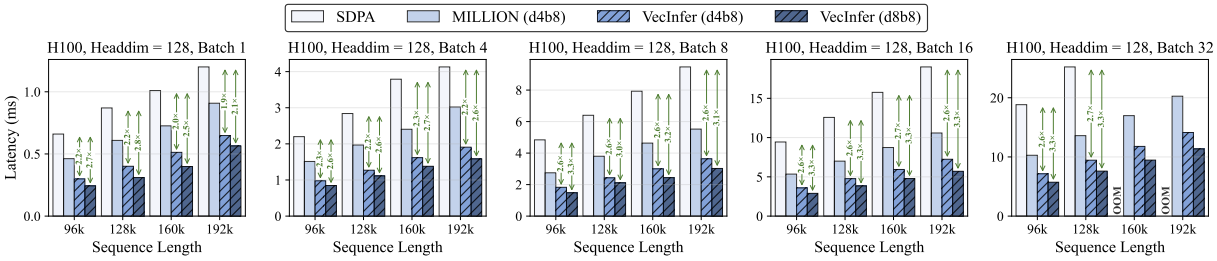
To reduce the overhead of reading/writing intermediate matrices in Equation (10), we adopt fused dequantization-computation kernel, which improves both memory and latency efficiency. Specifically, the implementation leverages fine-grained tiled computation and asynchronous pipeline execution, as outlined in Algorithm 1.

Given the quantized codes $\tilde{\mathbf{K}}_q, \mathbf{V}_q \in \mathbb{R}^{N\times M}$, a pre-computed lookup table $\mathbf{lut} \in \mathbb{R}^{M\times 2^b}$, and a value codebook $\mathcal{C}_v \in \mathbb{R}^{2^b \times \frac{D}{M}}$, we aim to compute the attention output $\mathbf{o} \in \mathbb{R}^{1\times D}$ and write it back to global memory. The algorithm begins by partitioning $\tilde{\mathbf{K}}_q$ and $\mathbf{V}_q$ into $T = \lceil \frac{N}{B} \rceil$ blocks, each of size $B \times M$ (line 5). Attention is then computed in a block-wise manner, with iterations described in lines 9-19. To minimize memory I/O, we adopt the online softmax scheme (Dao et al., 2022), which incrementally rescales partial results from each block to ensure a correct final output.

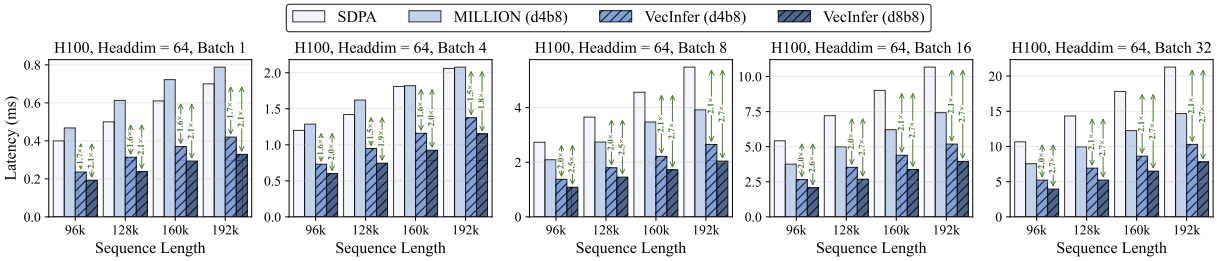
To fully utilize CUDA cores, we leverage the `memcpy_async` API to overlap memory transfers with computation. Specifically, while computing $\mathbf{s}^{(i)}$, we asynchronously load $\mathbf{V}_q^{(i)}$ from global memory into shared memory (line 10), thereby hiding memory latency. The first synchronization (line 14) ensures that this transfer has completed before the values are used. Similarly, during the computation of $\mathbf{o}^{(i)}$, we prefetch $\tilde{\mathbf{K}}_q^{(i+1)}$ for the next iteration (line 15). The second synchronization (line 17) then guarantees that the prefetched data are ready at the beginning of iteration $i{+}1$.

B.2 Kernel Latency Comparison

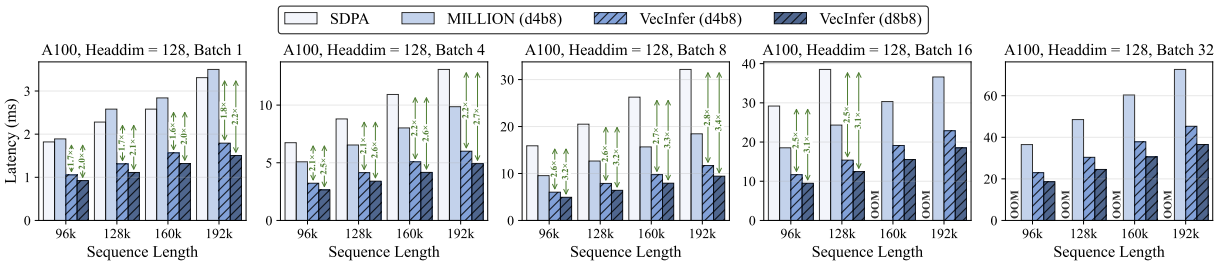
Figure 9 compares the speed of the VecInfer kernel against baselines using headdim=64 and headdim=128, with sequence lengths ranging from 96k to 192k, on A100 (40GB) and H100 (80GB) GPUs.



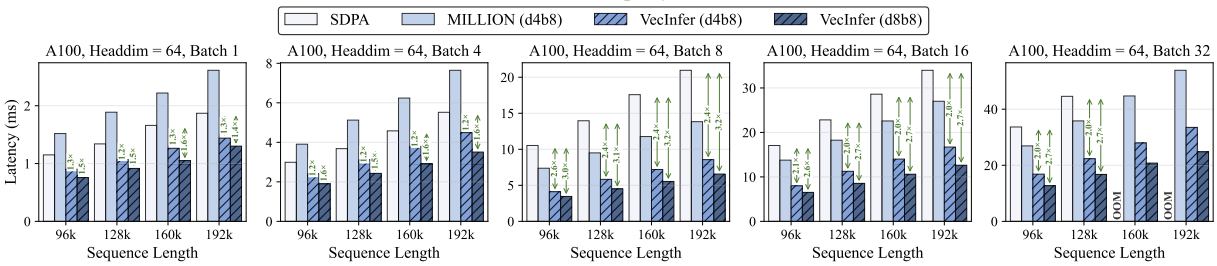
(a) H100 80GB, headdim=128, query heads=32, KV heads=8



(b) H100 80GB, headdim=64, query heads=32, KV heads=8



(c) A100 40GB, headdim=128, query heads=32, KV heads=8



(d) A100 40GB, headdim=64, query heads=32, KV heads=8

Figure 9: Comparison of kernel performance across different batch sizes and sequence lengths.

C Details of Different Transformations

To analyze the effects of different transformations, we use SVD, which factorizes a matrix as $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$, where $\mathbf{U}$ and $\mathbf{V}$ are orthogonal and $\mathbf{\Sigma}$ is diagonal. SVD decomposes transformations into rotation ($\mathbf{U}$, $\mathbf{V}$) and stretch ($\mathbf{\Sigma}$) components (Lee et al., 2024). For example, Figure 11(a) shows how the column vectors of $\mathbf{V}^\top$ are rotated and stretched to form the column vectors a_1 and a_2 of $\mathbf{A}$, which represent the maximum and minimum values, respectively. Figure 11(b) and Figure 11(c) demonstrate that smooth and Hadamard transformations, respectively, reduce the magnitude difference between vectors $\tilde{a}_1$ and $\tilde{a}_2$ by stretching and rotating the column vectors of $\mathbf{V}^\top$. In Figure 11(d), combining these two transformations achieves optimal balance between vectors $\tilde{a}_1$ and $\tilde{a}_2$. Overall, Figure 10 further demonstrates that applying these transformations individually yields suboptimal uniformity, whereas their combination achieves the most uniform distribution.

D Quantization Sensitivity of Key/Value Cache

Existing research (Dong et al., 2024) has demonstrated that the presence of outliers amplifies quantization errors in key cache, leading to distinct quantization sensitivities between key cache and value cache. However, our findings reveal that even with outlier suppression through transforma-

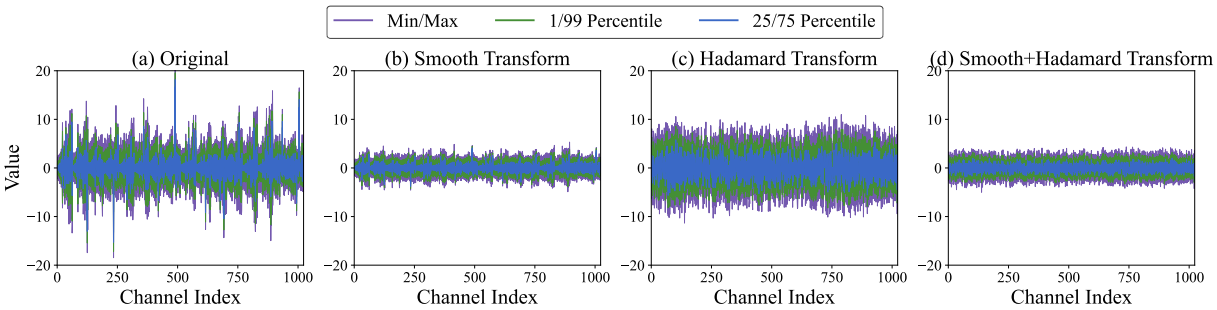


Figure 10: Distribution of key cache for Llama-3.1-8B-Instruct (layer 16) under different transformations.

Avg. bit	Key Config	Value Config	SD.QA		MD.QA		Sum.		FS.L			Code		Synth.		Avg.
			Qspr	MulF	HQA	WMQA	GRpt	MulN	TREC	SMSM	TriQA	Repo	LCC	PsgC	PsgR	
1.5	1.5-bit (d8b12)	1.5-bit (d8b12)	43.7	52.2	54.7	46.2	29.2	26.1	71.0	39.4	91.7	51.4	60.2	7.6	99.5	51.8
	1-bit (d8b8)	2-bit (d4b8)	37.2	45.4	52.1	41.9	21.7	23.8	64.5	36.9	88.4	43.9	52.6	8.0	96.0	47.1
	2-bit (d4b8)	1-bit (d8b8)	43.5	52.5	54.7	46.0	27.2	25.4	72.5	39.2	92.3	52.6	60.5	7.9	99.5	51.9
1.25	1.25-bit (d8b10)	1.25-bit (d8b10)	38.2	47.4	54.1	44.4	25.4	24.3	66.0	38.2	90.5	48.1	56.4	7.7	97.5	49.1
	1-bit (d8b8)	1.5-bit (d8b12)	36.3	44.9	52.4	42.2	21.1	23.2	62.5	36.0	87.8	43.2	52.9	8.0	97.0	46.7
	1.5-bit (d8b12)	1-bit (d8b8)	41.2	49.6	54.2	45.4	25.7	24.2	69.0	39.1	90.8	50.4	57.7	7.2	99.5	50.3

Table 6: Accuracy comparison under different quantization bit-width configurations using Llama-3.1-8B-Instruct.

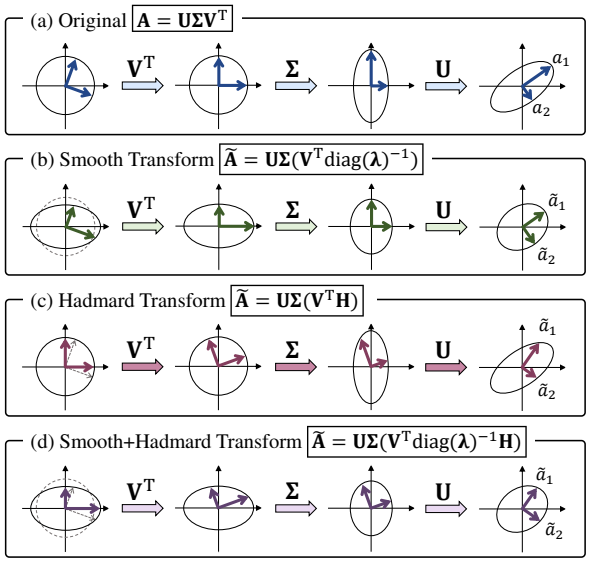


Figure 11: Transformation from $\mathbf{V}^{\top}$ to $\mathbf{A}$ via SVD.

tion, the transformed key cache still exhibits higher quantization sensitivity than value cache, as illustrated in Table 6. Our strategy involves implementing mixed precision by assigning separate storage precisions for key cache and value cache.

E Additional Experiments

E.1 Ablation Study: Task-independent Codebook

As shown in Table 7, the performance remains nearly consistent when using codebooks pretrained on different datasets, suggesting that the learned codebooks generalize well and are effectively task-independent.

E.2 Accuracy Evaluation

Table 8 reports additional results on LongBench. The results demonstrate that all models tested with VecInfer maintain lossless performance under 4-bit quantization, and VecInfer consistently outperforms other methods.

F Additional Details of Benchmarks

LongBench: A comprehensive benchmark covering six categories of evaluation tasks—single/multi-document question answering, summarization, code completion, synthetic tasks, and few-shot learning. Detailed information on all 13 datasets included in LongBench is presented in Table 9.

GSM8K: A carefully selected dataset comprising 1,319 grade-school math problems.

MATH500: A challenging math dataset comprising 500 problems from high school math competitions.

AIME24: A collection of 30 challenging mathematical problems sourced from the 2024 American Invitational Mathematics Examination.

AMC2023: A set of 40 problems from the 2023 American Mathematics Competitions, designed to rigorously test reasoning and problem-solving skills.

Dataset	SD.QA		MD.QA		Sum.		FS.L			Code		Synth.		Avg.
	Qspr	MulF	HQA	WMQA	GRpt	MulN	TREC	SMSM	TriQA	Repo	LCC	PsgC	PsgR	
Qspr	46.1	53.1	55.0	46.5	31.2	26.5	72.0	41.6	92.2	53.2	60.9	7.9	98.5	52.7
MulF	46.2	52.5	54.8	46.3	31.3	26.3	72.5	42.8	92.4	53.8	60.6	7.3	99.5	52.8
HQA	44.2	53.1	55.1	47.0	32.1	26.8	72.5	42.3	91.9	55.0	61.1	7.6	99.5	53.0
WMQA	45.7	52.0	55.3	46.5	31.2	26.4	72.5	41.5	92.4	54.6	61.1	7.7	99.0	52.8
TriQA	43.8	52.2	55.6	47.3	30.7	26.7	72.0	41.8	91.7	53.7	61.4	7.5	100.0	52.7

Table 7: Evaluation results using codebooks clustered from different datasets under 2-bit quantization with Llama-3.1-8B-Instruct.

G Information About Use Of Ai Assistants

In this paper, we only use AI tools for grammar checking and code completion.

H Proof of Lemma

Lemma 1 (Hadamard). *For key states* $\mathbf{K} \in \mathbb{R}^{N \times D}$ *with* $\text{sign}(K_{i,j}) \overset{\text{i.i.d.}}{\sim} \text{Uniform}\{-1, +1\}$*, and a Hadamard matrix* $\mathbf{H} \in \mathbb{R}^{D \times D}$ *constructed as in Equation (5), the transformed matrix* $\tilde{\mathbf{K}} = \mathbf{KH}$ *exhibits approximately Gaussian distribution by the central limit theorem, thereby redistributing the outliers of* $\mathbf{K}$.

Proof. We denote the (i, j)-th entry of matrices $\mathbf{K}$, $\mathbf{H}$, and $\tilde{\mathbf{K}}$ as $K_{i,j}$, $H_{i,j}$, and $\tilde{K}_{i,j}$, respectively. Consider any element of $\tilde{\mathbf{K}}$:

$$\tilde{K}_{i,j} = \sum_{l=1}^{D} K_{i,l} H_{l,j} = \sum_{l=1}^{D} |K_{i,l}| \cdot \epsilon_{i,l} \cdot H_{l,j}, \quad (14)$$

where $\epsilon_{i,l} = \text{sign}(K_{i,l}) \overset{\text{i.i.d.}}{\sim} \text{Uniform}\{-1, +1\}$, and $H_{l,j} \in \left\{-\frac{1}{\sqrt{D}}, +\frac{1}{\sqrt{D}}\right\}$. Then:

$$\epsilon_{i,l} \cdot H_{l,j} \overset{\text{i.i.d.}}{\sim} \text{Uniform}\left\{-\frac{1}{\sqrt{D}}, +\frac{1}{\sqrt{D}}\right\}. \quad (15)$$

The expectation of $\tilde{K}_{i,j}$ is given by:

$$\begin{aligned} \mathbb{E}[\tilde{K}_{i,j}] &= \mathbb{E}\left[\sum_{l=1}^{D} |K_{i,l}| \cdot \epsilon_{i,l} \cdot H_{l,j}\right] \\ &= \sum_{l=1}^{D} |K_{i,l}| \cdot \mathbb{E}[\epsilon_{i,l} \cdot H_{l,j}] = 0, \end{aligned} \quad (16)$$

The variance of $\tilde{K}_{i,j}$ is computed as follows:

$$\begin{aligned} \text{Var}(\tilde{K}_{i,j}) &= \text{Var}\left(\sum_{l=1}^{D} |K_{i,l}| \cdot \epsilon_{i,l} \cdot H_{l,j}\right) \\ &= \sum_{l=1}^{D} |K_{i,l}|^2 \cdot \text{Var}(\epsilon_{i,l} \cdot H_{l,j}). \end{aligned} \quad (17)$$

As $\mathbb{E}[\epsilon_{i,l} \cdot H_{l,j}] = 0$, we have:

$$\text{Var}(\epsilon_{i,l} \cdot H_{l,j}) = \mathbb{E}[(\epsilon_{i,l} \cdot H_{l,j})^2] = \frac{1}{D}. \quad (18)$$

Thus, the variance of $\tilde{K}_{i,j}$ becomes:

$$\text{Var}(\tilde{K}_{i,j}) = \sum_{l=1}^{D} |K_{i,l}|^2 \cdot \frac{1}{D}. \quad (19)$$

By the Lindeberg-Feller Central Limit Theorem, as $D \to \infty$, the sum $\tilde{K}_{i,j}$ converges in distribution to a Gaussian random variable:

$$\tilde{K}_{i,j} \xrightarrow{d} \mathcal{N}\left(0, \frac{1}{D} \sum_{l=1}^{D} K_{i,l}^2\right). \quad (20)$$

□

Method	Avg. bit	Config	SD.QA		MD.QA		Sum.		FS.L			Code		Synth.		Avg.
			Qspr	MulF	HQA	WMQA	GRpt	MulN	TREC	SMSM	TriQA	Repo	LCC	PsgC	PsgR	
Llama-3.1-8B	16	-	45.9	53.8	55.2	46.6	34.6	27.5	72.5	43.8	91.6	56.4	63.2	8.0	99.5	53.7
KIVI	4.25	b4g128	45.6	53.8	54.9	47.1	34.6	27.1	72.5	44.0	92.4	56.7	63.2	7.6	99.5	53.6
MILLION	4	d2b8	46.5	53.5	55.6	46.1	34.1	26.9	72.5	42.6	92.1	55.1	62.9	8.0	99.5	53.5
VecInfer	4	d2b8	45.6	54.0	55.3	46.6	34.3	26.8	72.5	43.1	92.0	56.0	63.1	7.6	99.5	**53.6**
Mistral-7B	16	-	38.6	49.7	51.0	36.4	34.3	26.5	76.0	47.6	88.5	60.6	59.4	6.0	96.5	51.5
KIVI	4.25	b4g128	38.0	50.4	51.1	36.6	34.4	26.6	76.0	46.9	88.9	58.8	58.3	5.0	96.0	51.3
MILLION	4	d2b8	36.9	51.2	51.0	36.0	33.6	26.4	75.5	45.8	88.9	59.5	58.6	2.5	93.0	50.7
VecInfer	4	d2b8	38.4	49.8	51.1	35.9	34.1	24.0	76.0	47.4	88.7	60.7	59.3	6.0	96.0	**51.4**
Qwen2.5-14B	16	-	45.4	53.9	61.6	57.9	29.7	21.9	76.5	47.7	90.1	48.8	61.3	9.0	98.6	54.0
KIVI	4.25	b4g128	45.6	52.2	61.5	58.5	29.3	21.8	77.0	47.5	90.4	49.2	61.6	10.9	98.2	54.1
MILLION	4	d2b8	43.5	51.7	61.0	57.7	28.0	22.6	76.5	46.4	89.9	49.0	60.2	10.9	93.5	53.1
VecInfer	4	d2b8	45.8	54.0	61.9	58.3	29.5	21.5	77.0	47.7	90.0	49.2	61.3	10.0	98.9	**54.3**

Table 8: Additional LongBench evaluation results under 4-bit quantization.

Label	Task	Capability	Metric	Avg. len	#data
Qspr	Qasper	Single-Doc. QA (SD.QA)	F1	3,619	200
MulFi	MultiFieldQA-en	Single-Doc. QA (SD.QA)	F1	4,559	150
HQA	HotpotQA	Multi-Doc. QA (MD.QA)	F1	9,151	200
WMQA	2WikiMultihopQA	Multi-Doc. QA (MD.QA)	F1	4,887	200
GRpt	GovReport	Summarization (Sum.)	Rouge-L	8,734	200
MulN	MultiNews	Summarization (Sum.)	Rouge-L	2,113	200
TREC	TREC	Few-shot Learning (FS.L)	Accuracy (CLS)	5,177	200
SMSM	SAMSum	Few-shot Learning (FS.L)	Rouge-L	6,258	200
TriQA	TriviaQA	Few-shot Learning (FS.L)	F1	8,209	200
Lcc	LCC	Code Completion (Code)	Edit Sim	1,235	500
Repo	RepoBench-P	Code Completion (Code)	Edit Sim	4,206	500
PsgC	PassageCount	Synthetic (Synth.)	Accuracy (EM)	11,141	200
PsgR	PassageRetrieval-en	Synthetic (Synth.)	Accuracy (EM)	9,289	200

Table 9: Details of LongBench.

Dataset	Max Output	Responses	Avg. len	#data
GSM8K	16,384	1	111	1,319
MATH500	16,384	1	121	500
AIME2024	32,768	8	156	30
AMC2023	16,384	8	137	40

Table 10: Details of mathematical reasoning benchmarks.