THINK GLOBAL, ACT LOCAL: EFFICIENT KV CACHE QUANTIZATION VIA GLOBAL PREFILL CALIBRATION OF EARLY DECODING

Anonymous authors
Paper under double-blind review

ABSTRACT

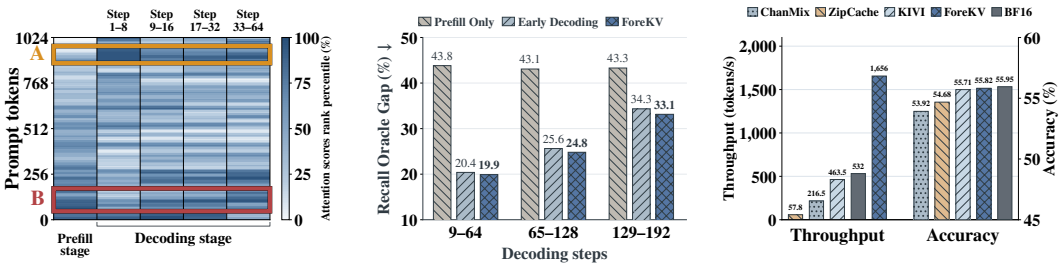
Efficient mixed-precision KV cache quantization requires identifying which cached tokens will remain important during future decoding, before aggressively compressing them. To address this challenge, we propose **ForeKV**, an efficient KV cache quantization framework that estimates future token importance from early decoding queries and calibrates this local estimate with global query information from prefill. Our key insight is that the first few decoding queries provide the earliest direct evidence of the model's emerging attention behavior, but such evidence is limited by the short observation window. Therefore, we delay compression for a few decoding steps to collect real decoding queries, estimate token importance from these local observations, and calibrate the estimate using the mean query over the global prefill sequence. The resulting query directly scores cached keys to guide mixed-precision allocation without materializing the full attention matrix. We further design a paged attention kernel for K caches with token-wise K4/K2 allocation and channel-wise quantization, enabling efficient decoding directly over the compressed cache. Experiments across multiple LLM backbones on LongBench show that ForeKV remains close to BF16 at an effective compressed-cache storage ratio of approximately $5.0\times$, with the average score dropping by no more than approximately 1%. On Qwen3-8B, our end-to-end ForeKV implementation achieves up to $3.1\times$ the generation throughput of the BF16 baseline under the same 80 GB GPU memory budget.

1 INTRODUCTION

Large language models (LLMs) have improved their ability to process long sequences, expanding their use in applications such as dialogue (Zheng et al., 2023), document understanding (Bai et al., 2024), question answering (Kamalloo et al., 2023), and information retrieval (Xu et al., 2024). Recent models such as Llama 3.1 (Grattafiori et al., 2024), Qwen2.5 (Yang et al., 2024b), Qwen3 (Yang et al., 2025), and Qwen3.5 (Team, 2026) have further extended the context lengths supported by LLMs. These longer contexts increase the KV cache size linearly with the sequence length, creating a growing memory burden during inference. Architectural designs such as Multi-Query Attention (Shazeer, 2019) and Grouped-Query Attention (Ainslie et al., 2023) reduce this cost by sharing key-value heads. However, when the model architecture is fixed, further memory reduction can be achieved by optimizing the KV cache directly during inference.

More recently, KV cache quantization has emerged as an effective approach to reducing memory usage by converting cached keys and values from floating-point formats to integer representations with fewer bits (Liu et al., 2024; Wu et al., 2025). Fixed-precision quantization assigns the same bit width to the entire cache, but quantizing the entire cache to very low precision, such as 2-bit, introduces large quantization error and can even degrade model accuracy. To address this limitation, mixed-precision quantization retains important tokens at higher precision while compressing the remaining tokens to lower precision (Yang et al., 2024a; He et al., 2024). The main difficulty is to determine which cached tokens will remain important during future decoding. Attention scores are commonly used as an indicator of token importance (He et al., 2024), but the importance of a cached token changes with the decoding queries that attend to it. In most KV compression pipelines, the cache is compressed immediately after prefill, before real decoding queries become available. This makes it

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.



(a) (b) (c)

Figure 1: **Motivation and effectiveness of ForeKV.** (a) **Token attention score distribution.** Colors indicate the percentile rank of attention scores over prompt tokens, with darker colors representing higher scores. (b) **Future importance estimation.** Oracle Gap measures the distance to the importance ranking obtained from future decoding queries, where lower is better. (c) **Accuracy and throughput.** LongBench accuracy and peak output throughput of Qwen3-8B under an 80GB GPU memory budget. Detailed protocols for panels (a) and (b) are provided in Appendix A.

difficult to use decoding information when estimating which tokens will remain important later. Recent studies have explored future-aware importance estimation by approximating information from future decoding queries (Devoto et al., 2025; Wang et al., 2025). As illustrated by the representative example in Figure 1(a), the attention-score distribution over prompt tokens continues to change across decoding windows, making future token importance difficult to infer from prefill information alone. As highlighted by region A in Figure 1(a), some tokens that receive little attention during prefill become consistently important once decoding begins, illustrating the value of early decoding queries for estimating future token importance. However, region B shows that some tokens receiving limited attention in early decoding become more important later, while already receiving high attention during prefill. This complementary pattern motivates us to use global prefill information to improve the short-window estimate from early decoding queries. Consistent with this observation, Figure 1(b) provides a broader retrospective evaluation: early decoding queries provide a better estimate of future token importance than prefill information alone, while ForeKV consistently brings this short-window estimate closer to the future oracle through global prefill calibration.

In this paper, we propose **ForeKV**, a mixed-precision KV cache quantization method that estimates which cached tokens will remain important during future decoding. ForeKV follows a simple principle: **Think Global, Act Local**. Since real decoding queries are unavailable if the cache is compressed immediately after prefill, we delay compression for a few decoding steps and use the first observed queries as direct evidence of future token importance. These early queries form the main estimate, but their observation window is short. To improve this estimate, we use the mean query over the full prefill sequence to provide global query information and calibrate the early decoding queries. The resulting query representation directly scores cached keys, and the estimated importance is used to retain important K tokens at higher precision while quantizing the remaining tokens to lower precision. This token-dependent precision allocation also introduces a practical challenge for efficient KV cache execution. Prior work has shown that low-bit key quantization benefits from channel-wise quantization because key activations exhibit strong channel-wise outliers (Liu et al., 2024). ForeKV keeps this channel-wise quantization structure while assigning K4 and K2 precision to different tokens. Existing fixed-precision kernels are not designed for this heterogeneous layout. To support efficient decoding, we develop a paged attention backend tailored to token-wise K4/K2 allocation with channel-wise key quantization, allowing attention to operate directly on the compressed mixed-precision cache.

Our contributions are summarized as follows:

- We study how to foresee future token importance for mixed-precision KV cache quantization. Our analysis shows that early decoding queries provide direct evidence of which cached tokens remain important later in generation, and that global prefill information improves this short-window estimate.
- Based on this insight, we propose **ForeKV** following the principle of **Think Global, Act Local**. ForeKV delays KV compression to observe the first real decoding queries and

calibrates this local estimate using the mean query over the full prefill sequence before allocating precision to cached K tokens.

- We develop a paged attention backend tailored to K caches with token-wise K4/K2 allocation and channel-wise quantization, enabling decoding attention to operate directly on the compressed mixed-precision cache.
- We evaluate ForeKV across multiple LLM backbones, KV quantization settings, and long-context benchmarks, showing that it preserves model accuracy under aggressive KV-cache storage compression, while our end-to-end implementation improves decoding throughput under the same GPU memory budget.

2 Related Work

KV cache quantization. Recent work on KV cache quantization aims to preserve model quality at 2-bit or lower precision. Outliers and the different numerical properties of keys and values make such low-bit quantization challenging. KIVI addresses this asymmetry by quantizing keys per channel and values per token (Liu et al., 2024). KVQuant further improves low-bit accuracy with pre-RoPE key quantization and explicit outlier handling (Hooper et al., 2024). To make low-bit encoding more stable, PolarQuant (Wu et al., 2025) and RotateKV (Su et al., 2025) transform KV representations before quantization, while NSNQuant (Son et al., 2026) uses normalization and Hadamard transformation for low-bit vector quantization.

KV cache eviction. KV cache eviction reduces memory by retaining a subset of cached tokens during generation. Early methods rely on positional or historical attention signals. For example, StreamingLLM (Xiao et al., 2024) preserves attention sinks and recent tokens, whereas H2O (Zhang et al., 2023) identifies important tokens based on accumulated attention. For more adaptive selection, CAKE (Qin et al., 2025) adjusts cache budgets across layers according to attention dynamics. To estimate future relevance, Lookahead Q-Cache (Wang et al., 2025) uses pseudo queries, while Expected Attention (Devoto et al., 2025) estimates importance from a query distribution. However, eviction discards KV entries entirely, making model quality sensitive to token-selection errors, especially under tight cache budgets.

Mixed-precision KV cache quantization. Mixed-precision KV quantization assigns higher precision to selected cache entries under a fixed budget. MiKV (Yang et al., 2024a) reuses token importance scores from eviction policies, while ZipCache (He et al., 2024) assigns token bit widths by attention saliency. Beyond token-level allocation, KVTuner (Li et al., 2025) and KVmix (Li et al., 2026) use layer sensitivity, while ChanMix (Liao & Wen, 2026) considers differences across KV channels. PM-KVQ (Liu et al., 2026) and MoQAE (Tao et al., 2025) explore block- and chunk-level allocation. MixKVQ (Zhang et al., 2026) uses query relevance and quantization sensitivity to select high-precision key channels. For token-level allocation, estimating importance remains challenging because most future decoding queries are unavailable at assignment. ForeKV delays compression until the first real decoding queries arrive, using their short-window mean to estimate future key importance and the mean query over the full prefill sequence to provide global information for calibrating this estimate before assigning precision to cached keys.

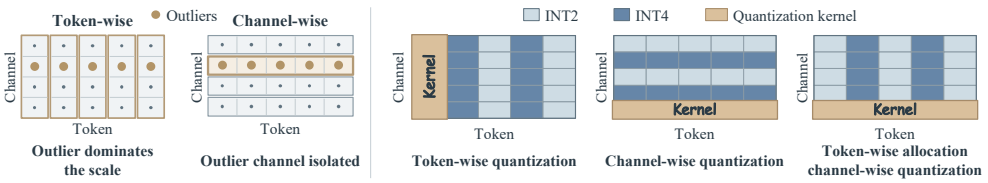
3 Method

3.1 Problem Formulation

Given an input prompt, autoregressive inference consists of a prefill stage followed by token-by-token decoding, during which the KV states of preceding tokens are cached and reused. Focusing on one decoder layer and omitting layer and head indices for clarity, the attention for a decoding query $\mathbf{q}$ over cached keys $\mathbf{K}$ and values $\mathbf{V}$ is computed as:

$$\mathbf{a} = \mathrm{softmax}\left(\frac{\mathbf{q}\mathbf{K}^\top}{\sqrt{d}}\right), \qquad \mathbf{o} = \mathbf{a}\mathbf{V}. \tag{1}$$

Here, $\mathbf{K}, \mathbf{V} \in \mathbb{R}^{N\times d}$ contain the cached states of N preceding tokens, where d is the head dimension. $\mathbf{a} \in \mathbb{R}^N$ denotes the attention weights and $\mathbf{o} \in \mathbb{R}^d$ is the corresponding attention output.



(a) Why channel-wise quantization? (b) ForeKV quantization design

Figure 2: **Precision allocation and quantization granularity.** (a) Channel-wise quantization isolates the quantization ranges of outlier channels from those of other channels. (b) ForeKV assigns precision across tokens while quantizing keys per channel.

Token-Wise Mixed Precision with Channel-Wise Key Quantization. KIVI (Liu et al., 2024) shows that keys benefit from channel-wise quantization, while values are better suited to token-wise quantization. As illustrated in Figure 2(a), large-magnitude key activations tend to concentrate in particular channels. With token-wise quantization, these outliers can dominate the shared quantization range and reduce the resolution available to other dimensions. Instead, channel-wise quantization computes a separate range for each channel, limiting the influence of outliers on other channels. While this choice addresses channel outliers, it does not determine which tokens should receive higher precision. According to Eq. 1, the attention weight of each cached key $\mathbf{k}_i$ depends on its interaction with the decoding query $\mathbf{q}$. Therefore, ForeKV uses query-dependent importance to assign different bit widths across tokens while retaining channel-wise key quantization, as illustrated in Figure 2(b). The value cache uses token-wise quantization at a uniform low bit width.

3.2 Future-Aware Key Estimation and Allocation

Query Representation and Calibration. To guide token-wise precision allocation, we delay KV cache compression for T_d decoding steps and use the first real decoding queries to estimate future key importance. For a given KV head in GQA, let $\mathcal{G} = \{1, \ldots, G\}$ index the G query heads sharing its keys and values. We compute the query means and calibration independently for each associated query head, omitting the head indices when discussing a fixed query head.

Let $\mathbf{q}_t^d$ denote the query at decoding step t. We form the early-decoding estimate $\boldsymbol{\mu}_d = T_d^{-1} \sum_{t=1}^{T_d} \mathbf{q}_t^d$. These queries provide direct evidence from the current generation, but their short observation window may not cover attention patterns that emerge later. We therefore use information from the full prefill sequence as a global reference to calibrate this estimate. Directly scoring N cached keys with all N prefill queries would require $\mathcal{O}(N^2 d)$ query–key computation per query head. Instead, we summarize the prefill queries by their mean:

$$\mathbf{Q}_p = [\mathbf{q}_1^p, \ldots, \mathbf{q}_N^p]^\top \in \mathbb{R}^{N \times d}, \qquad \boldsymbol{\mu}_p = \frac{1}{N} \sum_{j=1}^{N} \mathbf{q}_j^p. \tag{2}$$

This provides a compact reference over the entire prompt, reducing the query–key computation for scoring with this reference to $\mathcal{O}(Nd)$.

The remaining question is how much the early estimate should rely on this global reference. We use the directional consistency of the early queries as a confidence signal. To measure this consistency independently of query magnitude, we normalize each early query and compute the length of their mean direction:

$$\mathbf{u}_t = \frac{\mathbf{q}_t^d}{\|\mathbf{q}_t^d\|_2}, \qquad c_d = \left\| \frac{1}{T_d} \sum_{t=1}^{T_d} \mathbf{u}_t \right\|_2, \tag{3}$$

where $c_d \in [0, 1]$. A larger c_d indicates greater agreement among the observed query directions. We use this concentration to set the relative weight of early decoding and prefill:

$$\rho_d = \frac{c_d}{1 - c_d}, \qquad 0 \le c_d < 1. \tag{4}$$

The early-decoding estimate is then calibrated with the global prefill reference:

$$\hat{\boldsymbol{\mu}} = \frac{\rho_d \boldsymbol{\mu}_d + \boldsymbol{\mu}_p}{\rho_d + 1}. \tag{5}$$

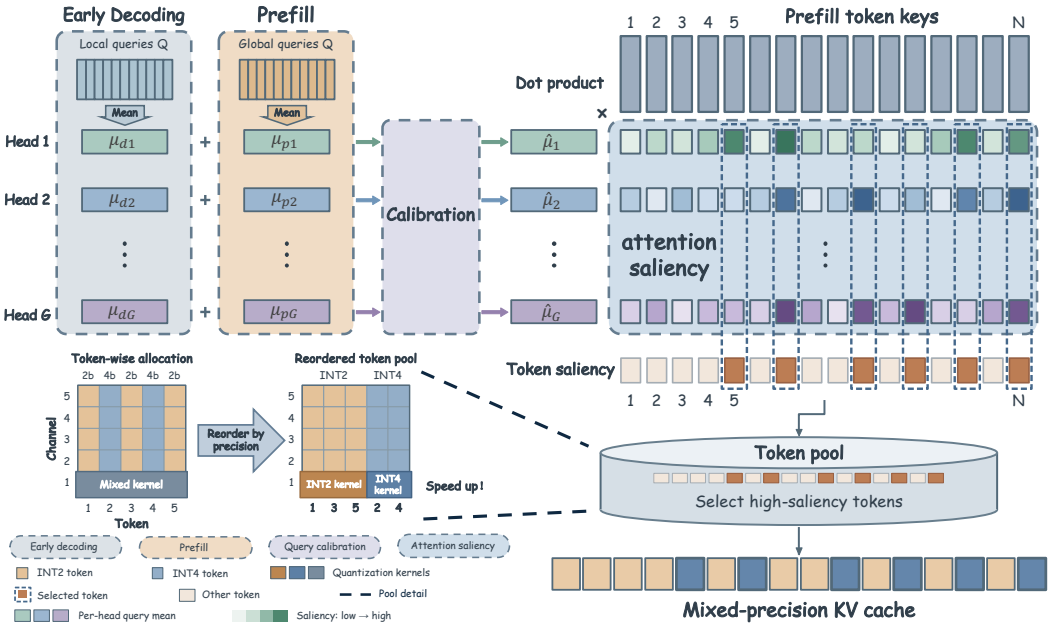


Figure 3: **Overview of ForeKV.** For the G query heads sharing one KV head, ForeKV calibrates early decoding query means with global prefill information. The maximum saliency across these heads guides token-wise key precision allocation, while a precision-grouped layout supports channel-wise key quantization and fused decoding attention.

When the early queries agree in direction, the calibrated representation relies more on $\boldsymbol{\mu}_d$; when they are more dispersed, $\boldsymbol{\mu}_p$ contributes more global information. In implementation, we evaluate the equivalent form $\hat{\boldsymbol{\mu}} = c_d\boldsymbol{\mu}_d + (1 - c_d)\boldsymbol{\mu}_p$, which also handles $c_d = 1$ without division by zero. Further details on the concentration measure, its interpretation, and numerical treatment are provided in Appendix H.1.

Key Scoring and Precision Allocation.

The calibrated query representations estimate which cached keys are likely to be important during subsequent decoding, as illustrated in Figure 3. For the same KV head, let $\hat{\boldsymbol{\mu}}^{(g)}$ denote the calibrated representation of query head $g \in \mathcal{G}$. We retain the query-head index here to aggregate scores across the associated heads. The estimated attention weight on cached key $\mathbf{k}_i$ is:

$$\hat{a}_i^{(g)} = \frac{\exp\left((\hat{\boldsymbol{\mu}}^{(g)})^\top \mathbf{k}_i/\sqrt{d}\right)}{\sum_{j=1}^{N} \exp\left((\hat{\boldsymbol{\mu}}^{(g)})^\top \mathbf{k}_j/\sqrt{d}\right)}. \tag{6}$$

The softmax is normalized over cached tokens independently for each query head. For each key, we then take the maximum of these normalized scores across the query heads sharing the same KV head:

$$s_i = \max_{g\in\mathcal{G}} \hat{a}_i^{(g)}. \tag{7}$$

The resulting s_i reflects the strongest estimated attention to key i across the associated query heads. We use this aggregated score to select one set of higher-precision keys for the KV head, rather than selecting a separate token set for each query head.

For each KV head, we rank all N cached keys by s_i. Let N_H denote the number of keys assigned higher precision under the target memory budget. The high- and low-precision token sets are defined as:

$$\mathcal{I}_H = \mathrm{TopK}\left(\{s_i\}_{i=1}^N, N_H\right), \qquad \mathcal{I}_L = \{1,\ldots,N\} \setminus \mathcal{I}_H. \tag{8}$$

Here, TopK returns the indices of the highest-scoring keys. Keys in $\mathcal{I}_H$ are assigned higher precision, while those in $\mathcal{I}_L$ use lower precision. The value cache does not participate in this importance-

based allocation and remains uniformly quantized at low precision. The same allocation and quantization procedure applies to every decoder layer.

3.3 Efficient Mixed-Precision Execution

Token-wise precision allocation assigns different key precisions along the sequence dimension, which is not directly suited to regular channel-wise packing. Following the sets $\mathcal{I}_H$ and $\mathcal{I}_L$, ForeKV reorders the cached tokens by their assigned key precision and stores them in separate paged regions. Within each page, keys are quantized per channel, with the quantization parameters shared across tokens. Values follow the same token permutation for aligned access, but remain uniformly quantized at low precision.

During decoding, the two regions are processed within a single attention kernel. Let $\widetilde{\mathbf{K}}_H$ and $\widetilde{\mathbf{K}}_L$ denote the higher- and lower-precision key regions, and let $\widetilde{\mathbf{V}}_H$ and $\widetilde{\mathbf{V}}_L$ denote their corresponding value subsets. The attention output can be written as:

$$\mathbf{o} = \mathrm{softmax}\left(\frac{\left[\mathbf{q}\widetilde{\mathbf{K}}_H^\top,\ \mathbf{q}\widetilde{\mathbf{K}}_L^\top\right]}{\sqrt{d}}\right)\begin{bmatrix}\widetilde{\mathbf{V}}_H\\ \widetilde{\mathbf{V}}_L\end{bmatrix}. \tag{9}$$

The kernel dequantizes the cached states on the fly and maintains a separate online-softmax state and attention output for each query head. Within each head, the state is shared across both precision regions, avoiding separate attention kernel launches for different key precisions. Newly generated KV states are temporarily kept in a small FP16 buffer and incorporated into the same per-head normalization before being periodically packed into the paged cache. Implementation details and kernel pseudocode are provided in Appendix H.2.

4 Experiments

4.1 Experimental Setup

Backbone Models. Experiments cover eleven Llama and Qwen backbones ranging from 3B to 27B parameters, including Llama-2-7B-Chat (Touvron et al., 2023), Llama-3.2-3B-Instruct, Llama-3.1-8B-Instruct (Grattafiori et al., 2024), Qwen2.5-3B/7B-Instruct (Yang et al., 2024b), Qwen3-4B/8B/14B (Yang et al., 2025), and Qwen3.5-4B/9B/27B (Team, 2026). LongBench results for six representative backbones are shown in Figure 4, with complete results provided in Appendix B.

Evaluating Tasks. We evaluate ForeKV on LongBench (Bai et al., 2024) and RULER (Hsieh et al., 2024). LongBench covers question answering, summarization, few-shot learning, synthetic tasks, and code completion. RULER complements this evaluation by testing retrieval, multi-hop tracing, aggregation, and question answering across context lengths of 4K, 8K, 16K, and 32K. For both benchmarks, we use the official task metrics and report average scores over the evaluated tasks, with RULER results presented separately for each context length.

Baseline Methods. The baselines include the full-precision FP16/BF16 cache, KIVI (Liu et al., 2024), ZipCache (He et al., 2024), PolarQuant (Wu et al., 2025), and ChanMix (Liao & Wen, 2026). All methods use the same backbone and evaluation protocol within each comparison. We report estimated effective KV-cache compression ratios accounting for quantized payloads, quantization parameters, and full-precision decoding buffers, with calculations detailed in Appendix E.

Implementation Details. Unless otherwise specified, ForeKV uses a global prefill query mean and the first $T_d = 8$ decoding queries for calibration. The top-ranked 40% of keys are quantized to INT4 and the remaining keys to INT2, while values are uniformly quantized to INT2. Keys use 32-token quantization pages, and newly generated KV states are accumulated in a full-precision buffer and quantized together every 128 tokens. Under the 4K storage-accounting setting, this corresponds to approximately $5.0\times$ effective KV-cache compression. Experiments are conducted on NVIDIA A100 80GB GPUs. Further implementation details are provided in Appendix H.

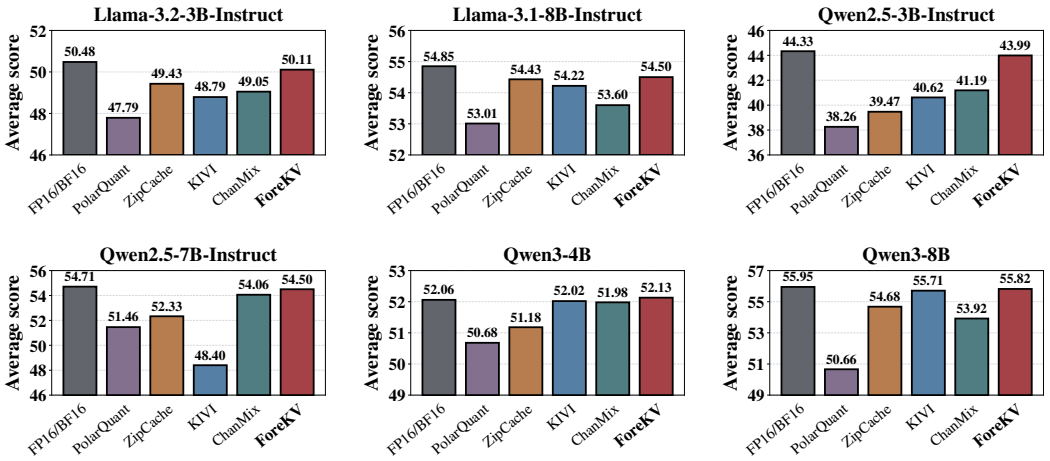


Figure 4: Average LongBench scores across different backbone models.

Table 1: Performance on LongBench tasks across six representative backbones. Comp. denotes the effective KV-cache compression ratio relative to FP16/BF16. The highest compression ratio is shown in **bold**. For performance, the best compressed-cache result within each backbone setting is highlighted in **bold**, and the second best is underlined.

Method	Comp.	Code		Multi-Document QA			Few-shot Learning		Single-Document QA		Synthetic & Summarization		Avg. ↑
		LCC	RB-P	HotpotQA	Musique	2WikiMQA	SAMSum	TREC	Qasper	MF-en	MultiNews	PR-en	
Llama-2-7B-Chat													
FP16/BF16	1.0×	59.96	53.05	30.76	9.63	26.75	41.31	62.00	19.63	33.42	25.91	7.50	33.63
ZipCache	4.7×	56.59	52.18	30.46	9.30	**28.03**	41.24	62.00	18.34	30.18	25.06	6.00	32.67
KIVI	4.9×	58.89	52.70	30.01	9.46	25.14	40.36	62.00	19.25	32.01	25.26	7.00	32.92
PolarQuant	1.6×	47.24	31.93	19.33	4.98	17.61	30.84	59.50	13.11	18.21	21.49	4.58	24.44
ChanMix	4.8×	51.22	47.72	26.22	8.92	26.01	41.48	**63.00**	**24.68**	**33.93**	**25.88**	6.50	32.32
ForeKV (ours)	**5.0×**	**59.75**	**53.08**	**30.66**	**9.51**	26.79	**41.66**	62.00	18.87	33.34	24.20	**7.50**	**33.40**
Llama-3.2-3B-Instruct													
FP16/BF16	1.0×	52.13	54.20	52.90	26.36	40.32	42.45	72.00	39.45	51.43	26.00	98.00	50.48
ZipCache	4.7×	51.49	53.89	52.18	25.14	38.62	42.42	72.00	38.60	49.89	26.00	93.50	49.43
KIVI	4.9×	50.28	52.79	52.69	25.00	**40.97**	41.98	**72.50**	36.57	48.93	25.46	89.50	48.79
PolarQuant	1.6×	40.80	45.40	51.54	**26.46**	40.07	40.50	71.50	37.68	**50.84**	25.39	95.50	47.79
ChanMix	4.8×	45.63	48.84	53.13	25.07	39.48	**43.30**	71.00	39.78	50.19	**26.13**	97.00	49.05
ForeKV (ours)	**5.0×**	**51.91**	**54.08**	**53.28**	25.02	39.84	42.17	71.00	**40.11**	50.51	25.82	**97.50**	**50.11**
Qwen2.5-3B-Instruct													
FP16/BF16	1.0×	56.51	57.06	46.15	19.95	36.39	44.73	68.50	37.30	49.08	22.92	49.00	44.33
ZipCache	4.7×	53.44	53.70	42.21	14.01	33.13	43.81	67.00	33.92	40.55	21.95	30.50	39.47
KIVI	4.9×	53.82	55.10	44.73	17.83	34.68	43.65	68.50	35.57	44.72	22.20	26.00	40.62
PolarQuant	1.6×	44.43	44.63	44.94	19.54	34.40	43.00	66.50	34.57	45.00	22.38	21.50	38.26
ChanMix	4.8×	49.87	51.51	46.11	19.55	**35.96**	**45.07**	**69.50**	**37.75**	**46.61**	**22.71**	28.50	41.19
ForeKV (ours)	**5.0×**	**56.49**	**56.56**	**46.67**	**20.83**	35.82	45.05	68.50	36.10	46.10	21.82	**50.00**	**43.99**
Qwen3-8B													
FP16/BF16	1.0×	68.85	65.30	59.47	36.72	42.95	44.16	71.50	48.13	53.55	24.85	100.00	55.95
ZipCache	4.7×	68.32	64.53	57.77	33.35	42.38	44.71	71.50	46.32	49.57	24.49	98.50	54.68
KIVI	4.9×	68.82	64.88	59.04	35.30	**43.35**	43.93	71.50	**48.03**	53.14	**24.77**	**100.00**	55.71
PolarQuant	1.6×	58.39	58.25	56.63	35.37	40.68	39.33	66.50	40.64	48.85	23.37	89.25	50.66
ChanMix	4.8×	56.45	56.90	59.33	**37.10**	42.79	**44.73**	**72.00**	47.12	52.66	24.57	99.50	53.92
ForeKV (ours)	**5.0×**	**69.01**	**65.37**	**59.41**	35.44	43.29	44.69	71.50	47.50	**53.43**	24.42	**100.00**	**55.82**

4.2 Main Results on LongBench

We compare ForeKV with representative KV cache quantization methods on LongBench tasks across four backbone models. As shown in Figure 4, ZipCache improves over uniform low-bit quantization by assigning different precisions to tokens according to their estimated importance, while

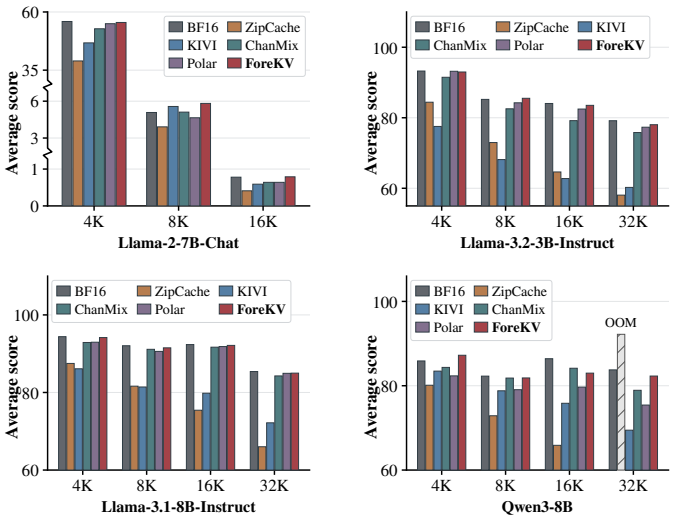


Figure 5: Average RULER scores across different context lengths and backbone models.

Table 2: Performance comparison with different key-selection criteria. The default configuration is highlighted in **bold**.

Criterion	Avg.
Random	82.63
Prefill Mean	83.61
Decode Mean	84.56
ForeKV	**84.75**

Table 3: Performance comparison with different K4 ratios. The default configuration is highlighted in **bold**.

K4 Ratio	Avg.
0%	80.44
40%	**84.75**
60%	84.99
100%	85.04

ChanMix further exploits channel-wise sensitivity to better preserve informative cache components under a similar compression budget. These methods show that non-uniform precision allocation is effective for maintaining long-context performance. ForeKV, however, forecasts future attention relevance from early decoding behavior and allocates higher precision to tokens that are likely to remain important. As further detailed in Table 1, ForeKV remains closest to the full-precision reference across the representative backbones while operating at an effective KV-cache storage compression ratio of approximately $5.0\times$.

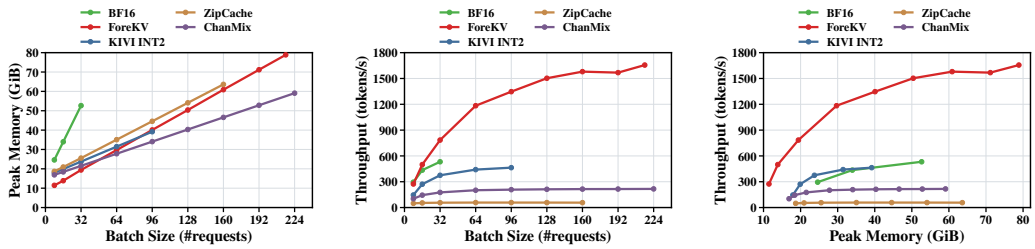
4.3 Evaluations On RULER Dataset

We also evaluate ForeKV against other KV cache quantization methods on RULER across different context lengths and four backbone models. RULER (Hsieh et al., 2024) covers retrieval, multi-hop tracing, aggregation, and question answering, allowing us to examine whether compression preserves the ability to locate, connect, and combine information within long contexts. As shown in Figure 5, ForeKV achieves leading average performance in most evaluated settings and generally remains close to the full-precision reference. These results support combining local decoding evidence with global prefill information for precision allocation. Early decoding queries reflect the emerging attention pattern, while prefill statistics provide a broader reference for calibrating the estimate obtained from this limited observation window. Together, they guide higher precision toward keys relevant to subsequent generation, helping preserve long-context performance under aggressive compression. By comparison, KIVI and ZipCache exhibit more pronounced degradation as the context grows, particularly on the Llama-3 backbones. ChanMix and PolarQuant remain more competitive, although their relative performance varies across models and context lengths.

4.4 Ablation Study

In this section, we present a series of ablation studies on RULER-4K to examine the effectiveness of our proposed precision allocation criterion and the efficiency of high precision allocation. Unless otherwise specified, we use Llama-3.2-3B-Instruct with 40% high precision keys as the default configuration. Additional ablation results and experimental details are provided in Appendix D.

Effectiveness of Future-Aware Key Selection. As shown in Table 2, using prefill information to identify high-precision keys already improves over random allocation, indicating that query-side statistics provide a meaningful signal for distinguishing important historical keys. Estimating importance from the mean of early decoding queries further improves performance, as these queries better reflect the model's attention behavior during generation. ForeKV further combines the local evidence from early decoding with the broader information captured during prefill, using the latter to calibrate the decoding query estimate and obtain a more reliable estimate of future key importance.



(a) (b) (c)

Figure 6: Memory usage and throughput comparison on Qwen3-8B generating 4096 tokens. ForeKV achieves higher throughput under the same memory budget.

Effectiveness of Selective Precision Allocation. Having established the importance of selecting the right keys, we next examine how much of the key cache needs to be preserved at higher precision. As shown in Table 3, assigning INT4 precision to only 40% of the keys already captures most of the benefit over an all-INT2 cache, whereas increasing the high-precision ratio to 60% or even 100% provides only marginal additional gains in this configuration. This saturation suggests diminishing returns from uniformly increasing key precision beyond 40% in the evaluated setting. Therefore, we adopt 40% high-precision keys as a practical default, retaining most of the observed accuracy benefit while maintaining a stronger compression ratio.

4.5 Efficiency Evaluation

We evaluate ForeKV and the baseline methods on Qwen3-8B under a shared end-to-end generation protocol, varying the batch size while generating 4,096 tokens per request. This long-generation stress test examines whether KV-cache compression translates into practical memory and throughput gains. Figure 6(c) shows that ForeKV achieves substantially higher throughput at comparable memory costs. Detailed settings are provided in Appendix F.

Peak Memory Usage. As shown in Figure 6(a), the end-to-end ForeKV implementation substantially reduces peak memory relative to full-precision caching, supporting larger batches in this long-generation setting. Although ChanMix achieves a lower footprint at large batch sizes, it encounters OOM at the next tested configuration, while KIVI encounters a runtime limitation at batch size 256. Beyond compressing the stored cache, our attention kernel limits temporary memory by consuming packed KV directly without reconstructing the dense cache, allowing ForeKV to sustain efficient large-batch generation close to the device's memory limit.

Throughput Analysis. Regarding generation throughput, Figure 6(b) shows that ForeKV benefits more from increasing batch size, while ZipCache, ChanMix, and KIVI approach saturation earlier. This difference partly reflects their cache processing paths: ZipCache and ChanMix reconstruct dense KV tensors before attention, whereas KIVI processes quantized and full-precision branches separately. Our kernel integrates low-bit cache access with attention computation, reducing intermediate memory traffic and improving batched execution. Figure 6(c) further shows that ForeKV achieves approximately $3.1\times$ the generation throughput of BF16 under the same 80GB GPU memory budget at each method's best-performing tested batch size, demonstrating more effective use of the available memory capacity.

5 Conclusion

We presented ForeKV, a mixed-precision KV cache quantization framework following *Think Global, Act Local*. By briefly delaying compression, ForeKV estimates future key importance from early decoding queries and calibrates this estimate with global prefill information. Our paged attention kernel supports token-wise precision allocation with channel-wise key quantization directly on the compressed cache. Experiments on LongBench and RULER show that ForeKV remains close to full-precision performance under aggressive compression. On Qwen3-8B, our end-to-end ForeKV implementation achieves up to $3.1\times$ the generation throughput of BF16 under the same 80GB GPU memory budget, demonstrating the practical benefit of combining future-aware allocation with efficient compressed-cache execution.

AI USE STATEMENT

Generative AI tools were used to assist with language editing, LaTeX formatting, and figure-generation utilities. They were not used to generate experimental data or to make unverified scientific claims. All experimental settings, numerical results, and final scientific claims were reviewed and verified by the authors, who take responsibility for the final content of this work.

ETHICS STATEMENT

This work studies inference-time KV cache quantization using publicly available model checkpoints and benchmark datasets. It does not involve human participants or the collection of personal data. The method aims to reduce the memory cost of long-context language-model inference, while its use remains subject to the licenses, safety policies, and deployment practices of the underlying models and datasets.

REPRODUCIBILITY STATEMENT

We document the evaluated model backbones, evaluation settings for LongBench and RULER, quantization configurations, and ForeKV hyperparameters to support reproducibility. Appendix H.1 details the directional-concentration calibration of early-decoding query means using full-prefill query means, while Appendix H describes the mixed-precision cache organization and execution pseudocode. The appendices also provide additional per-task results, ablation studies, future-importance diagnostics, KV-cache storage accounting, and the experimental setup for the efficiency and memory evaluations.

REFERENCES

Joshua Ainslie, James Lee-Thorp, Michiel De Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 4895–4901, 2023.

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, et al. Longbench: A bilingual, multitask benchmark for long context understanding. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 1: Long papers)*, pp. 3119–3137, 2024.

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

Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations*, volume 2024, pp. 35549–35562, 2024.

Alessio Devoto, Maximilian Jeblick, and Simon Jégou. Expected attention: Kv cache compression by estimating attention from future queries distribution. *arXiv preprint arXiv:2510.00636*, 2025.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Yefei He, Luoming Zhang, Weijia Wu, Jing Liu, Hong Zhou, and Bohan Zhuang. Zipcache: Accurate and efficient kv cache quantization with salient token identification. *Advances in Neural Information Processing Systems*, 37:68287–68307, 2024.

Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun S Shao, Kurt Keutzer, and Amir Gholami. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *Advances in Neural Information Processing Systems*, 37:1270–1303, 2024.

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

Ehsan Kamalloo, Nouha Dziri, Charles LA Clarke, and Davood Rafiei. Evaluating open-domain question answering in the era of large language models. In *Proceedings of the 61st annual meeting of the Association for Computational Linguistics (volume 1: long papers)*, pp. 5591–5606, 2023.

Fei Li, Song Liu, Weiguo Wu, Shiqiang Nie, and Jinyu Wang. Kvmix: Gradient-based layer importance-aware mixed-precision quantization for kv cache. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 31563–31572, 2026.

Xing Li, Zeyu Xing, Yiming Li, Linping Qu, Hui-Ling Zhen, Wulong Liu, Yiwu Yao, Sinno Jialin Pan, and Mingxuan Yuan. Kvtuner: Sensitivity-aware layer-wise mixed-precision kv cache quantization for efficient and nearly lossless llm inference. *arXiv preprint arXiv:2502.04420*, 2025.

Chengxi Liao and Zeyi Wen. Channel-aware mixed-precision quantization for efficient long-context inference. In *The Fourteenth International Conference on Learning Representations*, 2026.

Tengxuan Liu, Shiyao Li, Jiayi Yang, Tianchen Zhao, Feng Zhou, Xiaohui Song, Guohao Dai, Shengen Yan, Huazhong Yang, and Yu Wang. Pm-kvq: Progressive mixed-precision kv cache quantization for long-cot llms. In *International Conference on Learning Representations*, volume 2026, pp. 107881–107900, 2026.

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

Ziran Qin, Yuchen Cao, Mingbao Lin, Wen Hu, Shixuan Fan, Ke Cheng, Weiyao Lin, and Jianguo Li. Cake: Cascading and adaptive kv cache eviction with layer preferences. In *International Conference on Learning Representations*, volume 2025, pp. 89945–89977, 2025.

Noam Shazeer. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019.

Donghyun Son, Euntae Choi, and Sungjoo Yoo. Nsnquant: A double normalization approach for calibration-free low-bit vector quantization of kv cache. *Advances in Neural Information Processing Systems*, 38:43124–43159, 2026.

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

Wei Tao, Haocheng Lu, Xiaoyang Qu, Bin Zhang, Kai Lu, Jiguang Wan, and Jianzong Wang. Moqae: Mixed-precision quantization for long-context llm inference via mixture of quantization-aware experts. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 10810–10820, 2025.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL `https://qwen.ai/blog?id=qwen3.5`.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

Yixuan Wang, Shiyu Ji, Yijun Liu, Yuzhuang Xu, Yang Xu, Qingfu Zhu, and Wanxiang Che. Lookahead q-cache: Achieving more consistent kv cache eviction via pseudo query. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 34146–34162, 2025.

Songhao Wu, Ang Lv, Xiao Feng, Yufei Zhang, Xun Zhang, Guojun Yin, Wei Lin, and Rui Yan. Polarquant: Leveraging polar transformation for efficient key cache quantization and decoding acceleration. *arXiv preprint arXiv:2502.00527*, 2025.

Haojun Xia, Xiaoxia wu, Jisen Li, Tsai-chuan Wu, Junxiong Wang, Jue Wang, Chenxi Li, Aman Singhal, Alay Dilipbhai Shah, Alpay Ariyak, Donglin Zhuang, Zhongzhu Zhou, Ben Athiwaratkun, Zhen Zheng, and Shuaiwen Song. Kitty: Accurate and efficient 2-bit kv cache quantization with dynamic channel-wise precision boost. In A. Chowdhery and Z. Jia (eds.), *Proceedings of Machine Learning and Systems*, volume 8, pp. 511–525. MLSys, 2026. URL https://proceedings.mlsys.org/paper_files/paper/2026/file/e4d8d1b5120be349d3fff8878650cf45-Paper-Conference.pdf.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In *International Conference on Learning Representations*, volume 2024, pp. 21875–21895, 2024.

Peng Xu, Wei Ping, Xianchao Wu, Lawrence McAfee, Chen Zhu, Zihan Liu, Sandeep Subramanian, Evelina Bakhturina, Mohammad Shoeybi, and Bryan Catanzaro. Retrieval meets long context large language models. In *International Conference on Learning Representations*, volume 2024, pp. 49569–49584, 2024.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

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

Qwen An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxin Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yi-Chao Zhang, Yunyang Wan, Yuqi Liu, Zeyu Cui, Zhenru Zhang, Zihan Qiu, Shanghaoran Quan, and Zekun Wang. Qwen2.5 technical report. *ArXiv*, abs/2412.15115, 2024b. URL https://api.semanticscholar.org/CorpusID:274859421.

Tao Zhang, Ziqian Zeng, Hao Peng, Huiping Zhuang, and Cen Chen. Mixkvq: Query-aware mixed-precision kv cache quantization for long-context reasoning. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7189–7204, 2026.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in neural information processing systems*, 36:34661–34710, 2023.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems*, 36:46595–46623, 2023.

A Motivation Study Details

A.1 Attention Dynamics Across Prefill and Decoding

We visualize attention changes using uncompressed Qwen3-4B on a single constructed urban transport-planning prompt requesting a summary and recommendations. The input contains 1024 tokens, including the chat template. Using BF16 eager attention and greedy decoding with thinking disabled, we collect attention through causal replay of the prompt and the first 64 generated tokens. We examine layer 18, query head 16 (corresponding to KV head 4), using zero-based indices and no cross-head or cross-layer averaging. Attention probabilities are computed over all causally visible keys before restricting the analysis to prompt keys, without renormalization.

For each prompt key, the prefill score sums its received attention from prompt queries, excludes self-attention, and divides by 1023. Decoding scores average attention over windows 1–8, 9–16, 17–32, and 33–64; step 1 denotes the first generated token's own query. To visualize relative changes, we sum scores within each eight-token block, rank the resulting 128 blocks independently within each stage, and smooth their percentile ranks along token order with weights $(0.25, 0.50, 0.25)$. Colors thus represent within-stage relative ranks, not attention probabilities or high-precision selection ratios. Regions A and B are selected after inspection as illustrative examples.

A.2 Future-Importance Estimation

To examine how prefill and early decoding information help predict future key importance, we compare three estimators on Qwen3-8B. Prefill Only uses the global prefill query mean, whereas Early Decoding uses the mean of the first eight decoding queries. ForeKV-Concentration, labeled ForeKV in Figure 1(b), calibrates this local estimate with global prefill information through our directional-concentration weighting. All three estimators select 40% of the candidate keys.

We construct retrospective oracle selections using the decoding queries from windows 9–64, 65–128, and 129–192, respectively. For each window, the estimated and oracle selections use the same candidate keys and selection budget. We measure their agreement using Recall@K and report its complement as:

$$\text{Oracle Gap}(W) = 100 \left(1 - \frac{|S \cap S_W^*|}{|S_W^*|}\right), \tag{10}$$

where S is the estimated selection and S_W^* is the oracle selection for future window W. Oracle Gap thus measures the percentage of oracle-selected keys missed by an estimator, with lower values indicating better agreement. The successive windows assess whether the information available during early decoding remains useful as generation progresses; future queries are used only to construct the evaluation references.

B Full LongBench Results

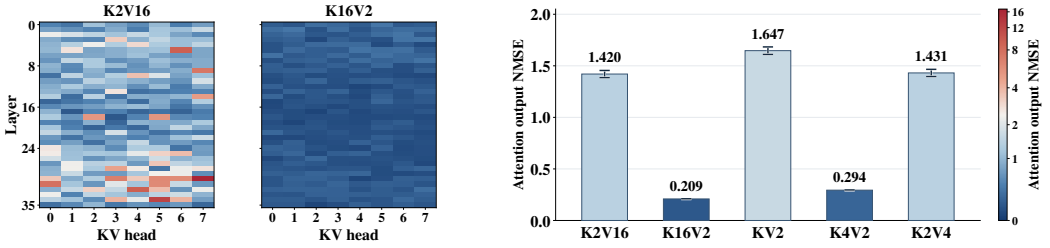
Extended LongBench Results. Beyond the representative backbones reported in the main text, we extend the evaluation to eleven Llama and Qwen models ranging from 3B to 27B parameters. As summarized in Table 4, this broader evaluation allows us to examine whether the effectiveness of ForeKV persists across both model families and substantially different model scales. ForeKV remains highly competitive throughout the model suite and achieves the best compressed-cache average on many of the evaluated backbones. More importantly, its performance remains consistently close to the full-precision cache as the model size increases, suggesting that future-aware precision allocation continues to preserve useful historical information rather than relying on characteristics specific to a particular backbone or parameter scale.

The larger models further reveal differences in practical scalability among compression methods. In particular, ZipCache encounters OOM on the 14B and 27B settings and therefore cannot complete these evaluations under the same hardware budget. This suggests that reducing KV storage alone may not translate directly into favorable memory scaling when temporary tensors or cache reconstruction remain in the execution path. Such overhead becomes more consequential as model size grows, making the efficiency of compressed cache execution increasingly important. ForeKV, in contrast, continues to operate on these larger backbones while maintaining performance close

Table 4: Performance on LongBench tasks across seven additional backbone settings. Comp. denotes the effective KV-cache compression ratio relative to FP16/BF16. The highest compression ratio is shown in **bold**. Avg. is computed over all tasks. For performance, the best compressed-cache result within each backbone setting is highlighted in **bold**, and the second best is underlined.

Method	Comp.	Code		Multi-Document QA			Few-shot Learning		Single-Document QA		Synthetic & Summarization		Avg. ↑
		LCC	RB-P	HotpotQA	Musique	2WikiMQA	SAMSum	TREC	Qasper	MF-en	MultiNews	PR-en	
Llama-3.1-8B-Instruct													
FP16/BF16	1.0×	62.90	56.80	57.90	33.05	48.68	44.32	72.50	44.88	56.00	26.77	99.50	54.85
ZipCache	4.7×	61.86	**56.34**	58.41	31.99	**49.72**	**44.29**	72.50	41.72	**56.20**	26.20	99.50	54.43
KIVI	4.9×	61.93	54.90	58.29	31.97	48.22	43.77	72.50	43.43	54.86	26.54	**100.00**	54.22
PolarQuant	1.6×	52.86	48.57	57.93	32.43	49.53	42.44	72.50	44.50	55.62	**26.76**	**100.00**	53.01
ChanMix	4.8×	54.38	51.11	**58.62**	32.18	49.39	43.85	**73.00**	**44.55**	56.01	26.56	**100.00**	53.60
ForeKV (ours)	**5.0×**	**62.63**	56.06	57.82	**32.89**	48.48	43.82	72.50	43.68	55.14	26.52	**100.00**	**54.50**
Qwen2.5-7B-Instruct													
FP16/BF16	1.0×	59.87	67.72	58.19	30.89	46.33	46.23	72.00	44.13	52.54	23.91	100.00	54.71
ZipCache	4.7×	58.63	63.38	56.86	25.92	43.51	45.32	**71.50**	42.55	48.07	23.34	96.50	52.33
KIVI	4.9×	56.46	60.52	49.31	27.89	39.86	44.20	68.50	34.81	47.55	22.96	80.33	48.40
PolarQuant	1.6×	55.39	61.00	55.44	29.77	42.28	44.39	69.50	42.17	50.84	23.26	92.00	51.46
ChanMix	4.8×	**60.70**	65.93	56.79	29.85	**46.69**	**46.32**	71.00	43.21	**52.68**	**23.98**	97.50	54.06
ForeKV (ours)	**5.0×**	59.88	**67.54**	**57.97**	**30.49**	46.37	46.25	**71.50**	**44.10**	51.97	23.47	**100.00**	**54.50**
Qwen3-4B													
FP16/BF16	1.0×	52.74	53.47	55.81	30.95	43.64	43.55	73.00	43.44	54.44	25.09	96.50	52.06
ZipCache	4.7×	52.04	51.10	55.58	32.24	43.03	42.91	**73.00**	42.92	50.16	24.97	95.00	51.18
KIVI	4.9×	51.84	53.35	55.81	**32.48**	43.34	43.28	**73.00**	**44.31**	53.13	25.18	**96.50**	52.02
PolarQuant	1.6×	51.24	53.05	54.32	31.77	42.88	41.85	71.00	41.76	52.04	25.04	92.54	50.68
ChanMix	4.8×	52.42	**53.61**	**56.55**	31.19	**44.07**	43.28	**73.00**	43.50	52.79	**25.32**	96.00	51.98
ForeKV (ours)	**5.0×**	**52.60**	53.50	55.97	31.39	43.80	**43.33**	**73.00**	44.11	**54.01**	25.18	**96.50**	**52.13**
Qwen3-14B													
FP16/BF16	1.0×	58.81	57.35	63.53	40.12	54.29	43.79	75.00	45.73	53.78	24.83	100.00	56.11
KIVI	4.9×	58.13	57.36	**63.23**	40.40	54.10	44.00	**76.00**	45.58	52.87	24.67	**100.00**	56.03
PolarQuant	1.6×	55.85	55.91	61.64	39.71	53.27	42.47	73.00	43.91	52.69	24.68	96.25	54.49
ChanMix	4.8×	**58.69**	56.94	62.64	40.25	54.12	**45.11**	75.50	**45.62**	**52.97**	24.86	**100.00**	56.06
ForeKV (ours)	**5.0×**	58.36	**57.74**	63.21	**40.56**	**54.16**	44.08	75.50	45.42	52.96	**25.09**	**100.00**	**56.10**
Qwen3.5-4B													
FP16/BF16	1.0×	38.82	44.32	64.81	44.88	57.69	35.65	63.50	45.81	58.23	24.18	92.21	51.83
ZipCache	4.7×	38.42	43.30	64.39	44.48	57.22	**36.93**	64.50	44.46	55.92	23.80	90.42	51.26
KIVI	4.9×	39.06	44.16	**64.91**	44.76	57.44	35.09	64.00	45.34	57.62	23.99	92.37	51.70
PolarQuant	1.6×	38.91	**44.44**	64.31	44.62	57.44	34.31	63.00	**45.73**	**58.29**	24.13	91.29	51.50
ChanMix	4.8×	**39.46**	43.11	64.79	44.45	**58.78**	34.79	**66.00**	45.42	57.35	**24.23**	**93.46**	**51.99**
ForeKV (ours)	**5.0×**	39.01	43.23	64.60	**45.02**	58.02	35.51	65.00	45.06	57.70	24.21	93.00	51.85
Qwen3.5-9B													
FP16/BF16	1.0×	54.48	54.78	61.76	48.20	61.50	37.36	69.00	51.59	56.44	23.88	100.00	56.27
ZipCache	4.7×	54.29	54.51	61.41	**47.85**	61.47	36.94	68.50	51.08	56.26	23.54	**100.00**	55.99
KIVI	4.9×	54.23	**54.85**	61.26	47.21	61.50	37.07	68.00	51.32	56.63	**23.97**	**100.00**	56.00
PolarQuant	1.6×	54.57	54.09	61.77	47.83	61.75	36.16	66.50	**51.34**	56.71	23.62	**100.00**	55.85
ChanMix	4.8×	53.77	53.57	61.86	47.20	**62.15**	**40.06**	**70.50**	50.85	**57.01**	23.93	**100.00**	**56.45**
ForeKV (ours)	**5.0×**	**54.84**	54.10	**62.04**	46.92	60.47	38.28	65.50	50.87	56.03	23.64	**100.00**	55.70
Qwen3.5-27B													
FP16/BF16	1.0×	60.29	62.17	67.95	55.27	67.54	40.31	70.50	50.87	55.53	23.55	100.00	59.45
KIVI	4.9×	60.48	**62.32**	67.13	55.14	**67.62**	**40.78**	71.00	50.52	**55.33**	23.64	**100.00**	59.45
PolarQuant	1.6×	59.98	61.58	67.13	55.14	67.54	39.62	70.00	50.49	55.27	23.61	**100.00**	59.12
ChanMix	4.8×	**60.49**	61.39	67.19	54.94	67.37	39.48	72.50	50.67	54.46	**23.66**	**100.00**	59.29
ForeKV (ours)	**5.0×**	60.10	61.84	**67.41**	**55.98**	67.49	40.23	**73.00**	**50.98**	54.49	23.64	**100.00**	**59.56**

to the full-precision cache. Its paged mixed-precision attention directly consumes the compressed cache, reducing the need for large intermediate representations and allowing the storage reduction to translate more effectively into practical memory savings. Together, these results highlight both the cross-scale robustness and implementation scalability of ForeKV.



(a) (b) (c)

Figure 7: **K/V quantization sensitivity on Qwen3-8B.** Attention-output NMSE with (a) 2-bit keys and FP16 values, (b) FP16 keys and 2-bit values, and (c) different K/V precision assignments. The two heatmaps share the same color scale. Error bars denote 95% confidence intervals.

C Asymmetric Key–Value Precision Allocation

To examine whether keys and values should receive the same precision budget, we conduct a controlled quantization-sensitivity study on Qwen3-8B. We sample eight examples from each of four LongBench tasks: Qasper, GovReport, passage_retrieval_en, and RepoBench-P, with prompts truncated to 4,096 tokens. Using fixed full-precision activations, we evaluate the last $\min(256, T)$ prompt queries against the complete prompt KV cache without a causal mask. Keys and values use the same symmetric quantizer with groups of 32 along the head dimension. We measure attention-output normalized mean squared error (NMSE) relative to the full-precision reference across all layers and KV heads. As shown in Figure 7(a) and (b), quantizing only keys to 2 bits introduces substantially larger attention-output errors than quantizing only values, with average NMSEs of 1.420 and 0.209, respectively. This difference is consistent with their respective roles: key perturbations alter the attention weights, whereas value perturbations affect the representations being aggregated. More importantly, Figure 7(c) shows that K4V2 produces much lower distortion than K2V4 under the same payload bit budget, reducing NMSE from 1.431 to 0.294. Increasing value precision alone offers limited improvement while keys remain heavily quantized, whereas increasing key precision substantially reduces the error even with 2-bit values. These results support prioritizing additional precision for keys, motivating ForeKV's asymmetric design: values remain uniformly in INT2, while mixed-precision allocation is applied to the key cache.

D Additional Ablation Studies

In this section, we examine the sensitivity of ForeKV to its observation window and quantization settings using Llama-3.2-3B-Instruct. The LongBench evaluation selects one representative dataset from each task category: PassageRetrieval-en, LCC, HotpotQA, TriviaQA, Qasper, and QMSum. We report the unweighted average across these six datasets as the overall score. We vary the early-decoding window T_d, token group size, and value precision in separate sweeps. We additionally evaluate accuracy, peak memory, and per-sample runtime on RULER-4K to examine the practical cost of delaying compression.

Observation Window. As shown in Table 5, collecting several decoding queries improves performance over relying on a single query, but extending the window does not yield monotonic gains. The multi-query settings remain close in accuracy, with $T_d = 8$ achieving the highest score in this sweep. We therefore retain a short observation window that captures decoding information without unnecessarily postponing compression.

Token Group Size. We next vary the number of tokens sharing quantization parameters within each key channel. Although smaller groups allow more localized quantization ranges, the scores remain nearly identical across the tested group sizes. Increasing the group size therefore reduces quantization-parameter overhead without appreciably affecting accuracy in this setting. This supports our choice of 32-token groups as a storage-efficient alternative to finer-grained quantization.

Value Precision. To determine whether additional precision should also be assigned to values, we compare uniform INT2, uniform INT4, and a mixed-precision policy in which values follow their paired keys. Under the latter, values paired with selected high-precision keys use INT4, while the

Table 5: Sensitivity to the observation window, token group size, and value precision on the six-task LongBench subset. The three blocks report separate parameter sweeps. **Bold** marks the default configuration within each sweep.

Observation Window		Token Group Size		Value Precision	
T_d	Avg.	Group size	Avg.	Policy	Avg.
1	57.77	8	58.28	**All INT2**	**58.26**
2	58.22	16	58.18	V follows K	58.26
4	58.13	**32**	**58.26**	All INT4	58.25
8	**58.26**				
16	58.07				

Table 6: Effect of the early-decoding observation window on RULER-4K with Llama-3.2-3B-Instruct. Peak memory is reported in MB and runtime in milliseconds per sample. **Bold** marks the default window length.

T_d	Avg.	Peak Memory (MB)		Time per Sample (ms)	
		Allocated	Reserved	Prefill	Decode
1	83.74	10355.18	10942	143.34	698.66
2	83.52	10355.18	10942	142.96	687.14
4	84.07	10355.20	10944	143.17	690.93
8	**84.75**	**10355.20**	**10944**	**143.07**	**677.98**
16	85.36	10354.92	10944	143.08	670.12

remaining values use INT2. Neither selectively nor uniformly increasing value precision improves the reported average over uniform INT2. These results support keeping values at low precision and concentrating the additional precision budget on keys.

Cost of Delayed Compression. Having examined accuracy sensitivity, we further investigate whether a longer observation window increases memory or runtime costs. Table 6 shows that RULER-4K performance generally improves as more early queries become available, while peak allocated and reserved memory remain nearly unchanged. Prefill time is also stable, and the recorded decoding time per sample shows no upward trend. Thus, extending the observation window within the tested range does not introduce an appreciable increase in measured peak memory or per-sample runtime. Together with the LongBench results, these findings support a short observation phase as a practical way to obtain decoding information before precision allocation.

E KV Cache Compression Analysis

Nominal bit-width alone does not fully characterize the memory footprint of a quantized KV cache, since practical implementations additionally store quantization parameters and may temporarily retain newly generated KV states in full precision. We therefore report the effective KV-cache compression ratio by accounting for the quantized payload, scale and zero-point parameters, and full-precision decoding buffers. Model weights and other runtime memory are excluded. Throughout this section, the reported compression ratio refers to the storage footprint of the compressed KV cache under this accounting and should be distinguished from end-to-end peak GPU memory.

Let b, h, and l denote the batch size, KV hidden dimension, and cache length, respectively. The reference FP16/BF16 KV cache occupies $2blh \cdot 16$ bits. We use $l = 4096$ throughout this analysis and define the compression ratio as $C = 1/R$, where R is the normalized cache size relative to FP16/BF16.

ZipCache. ZipCache assigns 4-bit precision to 60% of salient tokens and 2-bit precision to the remaining 40%, corresponding to an average payload precision of 3.2 bits. With residual length

$r = 100$, its normalized cache size is:

$$R_{\text{ZipCache}} = \frac{2blh \cdot 3.2 + 3h \cdot 16 + 2bl \cdot 16 + 2b\frac{100}{2}h \cdot 16}{2blh \cdot 16} \approx 0.2128, \qquad \mathbf{C}_{\text{ZipCache}} \approx \mathbf{4.70}\times. \tag{11}$$

KIVI. Our KIVI baseline quantizes both keys and values to 2 bits with group size $g = 32$ and residual length $r = 128$. Each quantization group additionally stores an FP16/BF16 scale and zero point. Its normalized cache size is:

$$R_{\text{KIVI}} = \frac{2blh \cdot 2 + \frac{2 \cdot 2blh \cdot 16}{32} + 2b\frac{128}{2}h \cdot 16}{2blh \cdot 16} \approx 0.2031, \qquad \mathbf{C}_{\text{KIVI}} \approx \mathbf{4.92}\times. \tag{12}$$

ChanMix. ChanMix starts from a 2-bit representation and increases the precision of 30% retrieval-sensitive channels, resulting in an average payload factor of 1.3 over the 2-bit baseline. It stores scale and zero-point parameters in FP8 and uses group size $g = 32$ and residual length $r = 128$. Its normalized cache size is:

$$R_{\text{ChanMix}} = \frac{2blh \cdot 2 \cdot 1.3 + \frac{2 \cdot 2blh \cdot 8}{32} + 2b\frac{128}{2}h \cdot 16}{2blh \cdot 16} \approx 0.2094, \qquad \mathbf{C}_{\text{ChanMix}} \approx \mathbf{4.78}\times. \tag{13}$$

PolarQuant. The PolarQuant configuration used in our experiments quantizes the key cache to approximately 4 bits while retaining the value cache in FP16/BF16. Its normalized KV-cache size is:

$$R_{\text{PolarQuant}} = \frac{blh \cdot 4 + blh \cdot 16}{2blh \cdot 16} = 0.625, \qquad \mathbf{C}_{\text{PolarQuant}} = \mathbf{1.60}\times. \tag{14}$$

ForeKV. ForeKV assigns INT4 precision to a fraction $\alpha = 0.4$ of historical keys and INT2 precision to the remaining 60%, while values are uniformly quantized to INT2. Keys are quantized with page size $g = 32$, with one FP16/BF16 scale and zero point per page and channel, whereas values use one FP16/BF16 scale and zero point per $d_h = 128$-dimensional KV head. The effective storage cost of the compressed historical cache is:

$$\beta_{\text{F}} = 4\alpha + 2(1-\alpha) + 2 + \frac{32}{g} + \frac{32}{d_h} = 6.05 \text{ bits}. \tag{15}$$

ForeKV does not preserve a fixed full-precision prefix, layer, or recent-token window. Instead, newly generated KV states are accumulated in an FP16/BF16 buffer of capacity $r = 128$ and quantized as a complete block once the buffer is full. Its occupancy therefore ranges from 0 to $r - 1$, with an average of $\bar{r} = (r-1)/2 = 63.5$. Since buffered KV states have not yet entered the compressed cache, they are counted only once. The resulting normalized cache size is:

$$R_{\text{ForeKV}} = \frac{b(l-\bar{r})h\beta_{\text{F}} + 2b\bar{r}h \cdot 16}{2blh \cdot 16} \approx 0.2016, \qquad \mathbf{C}_{\text{ForeKV}} \approx \mathbf{4.96}\times. \tag{16}$$

Under this unified accounting, the effective compression ratios of PolarQuant, ZipCache, KIVI, ChanMix, and ForeKV are approximately $1.6\times$, $4.7\times$, $4.9\times$, $4.8\times$, and $5.0\times$, respectively. Therefore, the latter four methods operate under closely matched compressed-cache storage budgets, allowing their accuracy differences to primarily reflect how the available precision is allocated rather than differences in stored-cache capacity.

F Efficiency and Memory Analysis

In this section, we provide additional details for the end-to-end efficiency evaluation reported in Section 4.5. All experiments use Qwen3-8B on a single A100 80GB GPU. We adopt the fixed GSM8K (Cobbe et al., 2021) prompt used by the Kitty (Xia et al., 2026) latency benchmark, containing 1183 input tokens, and force each request to generate 4096 tokens with greedy decoding. Each configuration is warmed up twice and measured over six repeated runs. Throughput is computed as the total number of generated tokens divided by the average end-to-end generation time, while peak allocated and reserved GPU memory are obtained from the corresponding CUDA memory statistics.

We retain the practical execution path provided by each baseline rather than replacing it with a common attention kernel. KIVI uses its quantized KV-cache implementation with separate quantized and residual branches. ZipCache reconstructs the compressed cache into a dense representation before the standard attention computation. ChanMix similarly performs quantization and packing with its customized kernels, but reconstructs dense FP16/BF16 KV tensors before invoking FlashAttention (Dao, 2024). Instead, ForeKV uses its paged mixed-precision attention kernel to consume the packed K4/K2 cache directly. The reported results capture not only the storage cost of each representation, but also the runtime overhead required to make the compressed cache usable by attention.

These implementation differences become increasingly visible as the batch size grows. ZipCache and ChanMix show relatively early throughput saturation, consistent with the additional reconstruction and memory movement in their decoding paths. ChanMix also reaches OOM at the next tested large-batch configuration, while KIVI encounters a runtime limitation at batch size 256 and cannot complete that setting. ForeKV avoids full-cache reconstruction and continues to benefit from larger batches, allowing its compressed representation to translate more directly into both memory capacity and decoding throughput. These observations provide additional context for the memory–throughput trade-off reported in Figure 6.

G LIMITATIONS

ForeKV introduces a future-aware perspective on mixed-precision KV-cache quantization by incorporating real early-decoding evidence into the precision allocation process and calibrating it with global prefill information. This design allows the initial precision layout to better reflect subsequent decoding behavior, while the paged backend enables direct execution over the resulting mixed-precision cache. Despite these strengths, several limitations and areas for future work remain. ForeKV currently summarizes the observed query behavior with one calibrated representation per query head and keeps it fixed after the initial observation window. Although our future-importance analysis in Appendix A.2 shows that early decoding information remains useful across subsequent decoding windows, a fixed representation does not explicitly adapt to attention shifts that may emerge later in long generations. Future work could explore lightweight online recalibration or richer query representations while retaining the low overhead of the current design. Additionally, ForeKV postpones the initial cache compression until the first T_d decoding queries are observed. Appendix D shows that this delay introduces little additional peak-memory or runtime cost in our evaluated settings, while progressive or overlapped cache packing could further reduce the transient full-precision phase. Finally, the current fused backend is specialized to the paged K4/K2 decode layout described in Section 3.3 and Appendix H.2. Extending this execution path to broader precision configurations and model settings could further improve the generality of ForeKV. In the future, we will continue to explore adaptive future-query modeling and more general compressed-cache execution to improve both the flexibility and efficiency of long-context inference.

H IMPLEMENTATION DETAILS

In this section, we describe the implementation of ForeKV during prefill and decoding. Listing 1 summarizes the cache lifecycle in PyTorch-style pseudocode, while Listing 2 details the fused decode attention kernel.

During prefill, we compute a full-prompt query mean independently for each of the G query heads sharing a KV head. Compression is delayed for the first $T_d = 8$ decoding steps. Each early-decoding mean is calibrated with the prefill mean from the same head using directional concentration. The calibrated means produce separate softmax scores over cached tokens; taking the maximum across the G heads yields one saliency score per key. Unless otherwise specified, the top-ranked 40% of cached keys in each KV head receive INT4 precision, while the remaining keys and all values use INT2. The same allocation applies to every decoder layer, without prefix protection. Tokens are reordered by key precision, with the same permutation applied to values. Keys are quantized channel-wise within pages of 32 tokens, whereas values are quantized token-wise.

The calibrated query representations are frozen after the initial compression and reused to score newly generated keys. New KV states accumulate in a 128-token FP16 buffer. Once full, its contents are quantized and appended to the paged cache before the current attention computation. Existing

packed entries are not reclassified. The fused kernel processes both precision regions and the active buffer without reconstructing the complete dense cache.

H.1 Details of Directional-Concentration Calibration

We use the directional consistency of early decoding queries to set the relative contributions of the decoding and prefill means. For a fixed query head, let $\mathbf{q}_t^d$ denote its query at decoding step t and $\boldsymbol{\mu}_d = T_d^{-1} \sum_{t=1}^{T_d} \mathbf{q}_t^d$ the early-decoding mean. The full-prefill mean is denoted by $\boldsymbol{\mu}_p$, with the head index omitted for clarity. Concentration is computed over decoding steps within this head:

$$c_d = \left\| \frac{1}{T_d} \sum_{t=1}^{T_d} \frac{\mathbf{q}_t^d}{\max(\|\mathbf{q}_t^d\|_2, \epsilon)} \right\|_2, \qquad \epsilon = 10^{-6}. \tag{17}$$

Normalization is used only to compute concentration; both query means retain their original magnitudes. We calibrate the early-decoding mean with the full-prefill reference as:

$$\hat{\boldsymbol{\mu}} = c_d \boldsymbol{\mu}_d + (1 - c_d) \boldsymbol{\mu}_p. \tag{18}$$

Consistent early queries place more weight on the decoding mean, whereas dispersed directions increase the contribution of the prefill reference. Each query head has its own concentration coefficient and calibrated mean.

Numerical stability. The denominator floor in Eq. 17 prevents division by zero during query normalization. We evaluate Eq. 18 directly, without forming $\rho_d = c_d/(1 - c_d)$ or clipping c_d below one. This expression is equivalent to the weight-ratio formulation in Eq. 5 for $0 \leq c_d < 1$ and yields $\hat{\boldsymbol{\mu}} = \boldsymbol{\mu}_d$ when $c_d = 1$.

Pseudocode. Listing 1 shows one layer and one KV head with its G query heads, omitting batching and the outer layer and KV-head loops. Query collection is represented by callbacks, and `key_saliency` follows the scoring procedure in the main paper.

H.2 Fused Mixed-Precision Decode Attention

Listing 2 implements `mixed_attention` with one program per batch element and KV head, sharing KV tiles across its associated query heads. Each query head maintains its own FP32 online-softmax state and output accumulator across INT4 pages, INT2 pages, and the active FP16 buffer. Page-table lookups and valid-length masks handle physical addressing and padding. Packed keys use zero-copy 32-bit views, yielding eight INT4 or sixteen INT2 codes per word. With a head dimension of 128 and 64-token tiles, only tile-sized KV operands are reconstructed. The helper operations are inlined within one attention kernel launch, which writes the final attention output for each query head.

Listing 1: ForeKV cache management in PyTorch-style pseudocode.

```
# One KV head: Q = [G, T, d]; K, V = [T, d]. Q and K are post-RoPE.
class ForeKVCache:
    def __init__(self, Td, page_size, buffer_size, low_ratio,
                 prefix_tokens, v_bits, eps=1e-6):
        self.Td, self.page = Td, page_size
        self.r, self.low_ratio = buffer_size, low_ratio
        self.prefix, self.v_bits = prefix_tokens, v_bits
        self.eps, self.mu = eps, None
        self.buffer, self.pages = FP16Buffer(), None

    def observe_prefill(self, Qp):
        self.mu_p = Qp.mean(dim=-2).detach()

    def compress_after_early(self, Qd, K, V):
        assert Qd.shape[-2] == self.Td
        mu_d = Qd.mean(dim=-2)
        U = Qd / Qd.norm(dim=-1, keepdim=True).clamp_min(self.eps)
        c = U.mean(dim=-2).norm(dim=-1, keepdim=True)
        self.mu = (c * mu_d + (1 - c) * self.mu_p).detach()

        # Freeze the query; pack the current cache only once.
        p = min(self.prefix, K.shape[-2])
        n_low = int((K.shape[-2] - p) * self.low_ratio)
        high = select_keys(self.mu, K, n_low, prefix=p)
        self.pages = pack_pages(K, V, high, page_size=self.page,
                                k_bits=(4, 2), v_bits=self.v_bits)
        del self.mu_p

    def decode(self, q, k, v):
        self.buffer.append(k, v)
        if len(self.buffer) >= self.r:
            K, V = self.buffer.active()
            n = K.shape[-2]
            n_low = int(n * self.low_ratio)
            if self.low_ratio > 0:
                n_low = min(max(n_low, 1), n)
            high = select_keys(self.mu, K, n_low)
            self.pages.append(pack_pages(
                K, V, high, page_size=self.page,
                k_bits=(4, 2), v_bits=self.v_bits))
            self.buffer.clear()
        # At a flush step, attention uses the newly packed block.
        return mixed_attention(q, self.pages, self.buffer)


def select_keys(mu, K, n_low, prefix=0):
    score = key_saliency(mu, K)
    low = score[prefix:].topk(n_low, largest=False).indices + prefix
    high = torch.ones_like(score, dtype=torch.bool)
    high[low] = False
    return high
```

Listing 2: ForeKV's fused mixed-precision decode attention (Kernel A).

```
# One program per (batch, KV head); Q and packed K are post-RoPE.
def mixed_attention_kernel(Q, pages, buffer, out, scale, BT=64):
    b, h = program_id(0), program_id(1)
    q = load_query_group(Q, b, h).to(fp16)           # [G, d]
    G, d = q.shape
    m = full((G,), -inf, fp32)
    l, acc = zeros((G,), fp32), zeros((G, d), fp32)

    # Separate precision streams, but the same running state.
    for pool, bits in ((pages.k4[b, h], 4), (pages.k2[b, h], 2)):
        for start in range(0, pool.n_pages * pool.page_size, BT):
            t = start + arange(BT)
            page, slot = t // pool.page_size, t % pool.page_size
            active = page < pool.n_pages
            pid = load(pool.table, page, mask=active, other=0)
            n = load(pool.valid_lengths, page, mask=active, other=0)
            valid = active & (slot < n)

            # K uses a zero-copy word view; V keeps its token layout.
            words = load_k_u32(pool, pid, slot, valid)
            sk, zk = load_k_params(pool, pid, valid)
            K = (unpack_k(words, bits) * sk + zk).to(fp16)
            vc, sv, zv = load_v_codes_params(pool, pid, slot, valid)
            V = (vc * sv[:, None] + zv[:, None]).to(fp16)
            m, l, acc = update(q, K, V, valid, scale, m, l, acc)

    for start in range(0, buffer.count, BT):
        t = start + arange(BT)
        valid = t < buffer.count
        K, V = load_buffer(buffer, b, h, t, valid)
        m, l, acc = update(q, K, V, valid, scale, m, l, acc)
    store_query_group(out, b, h,
                      (acc / maximum(l[:, None], 1e-20)).to(fp16))


def unpack_k(words, bits):
    # Eight INT4 or sixteen INT2 codes per unsigned 32-bit word.
    shifts = arange(32 // bits) * bits
    codes = (words[:, :, None] >> shifts) & ((1 << bits) - 1)
    return codes.reshape(words.shape[0], -1).to(fp16)


# Inlined tile update; statistics and output accumulation are FP32.
def update(q, K, V, valid, scale, m, l, acc):
    s = dot(q, K.T) * scale
    s = where(valid[None, :], s, -inf)
    mb = max(s, axis=1)
    p = where(valid[None, :], exp(s - mb[:, None]), 0)
    lb = sum(p, axis=1)
    ab = dot(p.to(fp16), V)
    mn = maximum(m, mb)
    a, b = exp(m - mn), exp(mb - mn)
    acc = a[:, None] * acc + b[:, None] * ab
    l = a * l + b * lb
    return mn, l, acc
```