RippleKV: Cross-Layer KV Cache Allocation via Perturbation Propagation

Dongjie Xu[1], **Kai Qian**[1], **Julius**[2], **Weijie Shi**[3], **Yuxuan Sun**[4], **Minghua Tang**[5], **Fenglei Jin**[2], **Hanchi Dong**[4], **Jiajie Xu**[1]

[1]Soochow University
[2]Tencent Inc.
[3]The Hong Kong University of Science and Technology
[4]Beijing University of Posts and Telecommunications
[5]Jiangnan University

arXiv:2608.08684v1 [cs.LG] 9 Aug 2026

Abstract

Long-context LLM inference is bottlenecked by KV cache memory, yet distributing a limited cache budget across layers remains challenging. Existing methods rely on proxies such as layer depth, attention statistics, or representation change. These proxies do not measure how perturbations at each layer propagate to the output and may therefore cause sensitive layers to be underallocated while tolerant layers are overallocated. To address this issue, we propose RippleKV, which allocates cache across layers by estimating how perturbations to each layer's value cache affect the final predictive distribution. RippleKV independently injects norm-adaptive perturbations into each layer's value cache and measures the induced KL divergence at the model output over a small calibration set. Averaging these responses yields a sensitivity profile specific to the model that need not vary monotonically with depth. RippleKV then converts the sensitivity profile into layer budget multipliers by normalizing the sensitivity scores and applying an exponential mapping. A ratio parameter controls the allocation disparity between sensitive and tolerant layers, while a final normalization preserves the KV cache budget. Experiments on LongBench demonstrate that RippleKV achieves the highest average performance among the evaluated KV cache compression methods under matched cache budgets.

Introduction

Large language models (LLMs) have been increasingly extended to long-context scenarios, enabling applications such as document summarization, question answering, information retrieval, and code generation (Roziere et al. 2023; Shaham et al. 2023). These applications often require processing long documents or multi-hop evidence (Zhang et al. 2024; Kamalloo et al. 2023). However, during autoregressive decoding, the KV cache grows linearly with the context length and the number of Transformer layers, leading to substantial memory and bandwidth pressure (Xiao et al. 2024; Zhang et al. 2023; Liu et al. 2024; Adnan et al. 2024). Therefore, KV cache compression has become critical for efficient long-context LLM inference.

Existing KV cache compression methods reduce memory usage by retaining only a subset of cached tokens. StreamingLLM (Xiao et al. 2024) preserves attention sinks and recent tokens, while Scissorhands, H_2O, and SnapKV (Liu et al. 2023; Zhang et al. 2023; Li et al. 2024) select important tokens based on persistent or accumulated attention scores. More recent layer-aware methods further assign different cache budgets across Transformer layers to improve cache utilization (Cai et al. 2024; Qin et al. 2025). These methods typically determine layer budgets using proxy signals such as layer depth, attention statistics, or representation changes. However, these proxies do not directly quantify the relative cache requirements of different layers under a fixed global budget, potentially underallocating sensitive layers while overallocating tolerant ones.

A principled allocation strategy should capture not only the structural or local characteristics of each layer, but also how strongly changes to its cache propagate to the final model output. Our analysis reveals that cache requirements vary substantially across layers and exhibit a nonmonotonic relationship with depth, with layers that incur greater damage under compression often appearing at intermediate positions. This heterogeneity cannot be adequately captured by fixed depth schedules or layer-local statistics alone. More importantly, we find that the shift in the final predictive distribution caused by perturbing a layer's Value cache is strongly aligned with the actual damage caused by compressing that layer. These observations suggest that downstream output sensitivity provides a more faithful basis for distributing a fixed global cache budget across layers.

Motivated by these observations, we propose RippleKV, a cross-layer KV cache allocation method guided by downstream output sensitivity. During offline profiling, RippleKV perturbs the Value cache of one layer at a time while keeping its Key cache and all other layers unchanged, and measures the induced KL divergence in the final output distribution. Averaging these responses produces a model specific sensitivity profile that reflects how strongly cache changes at each layer propagate to the model output. RippleKV then converts this profile into layer budgets through a scale invariant exponential mapping, where a ratio parameter controls the allocation disparity across layers while normalization preserves the global cache budget. Layers with stronger downstream effects receive larger budgets, whereas more tolerant layers are compressed more aggressively. Since RippleKV only redistributes cache capacity across layers while retaining the original token scoring and selection strategy, it can

Copyright © 2027, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

be integrated with existing KV cache compression methods without additional model evaluations during inference. Experiments on LongBench across three model families and multiple cache budgets show that RippleKV achieves the highest average performance in all evaluated settings while maintaining comparable runtime and memory efficiency.

Our contributions are summarized as follows:

- We demonstrate that layer-wise compression damage is highly heterogeneous and nonmonotonic with depth, and that the final output response to controlled Value cache perturbations provides a more faithful signal for cross-layer budget allocation than layer position.
- We introduce RippleKV, a cross-layer KV cache allocation framework built on perturbation propagation. RippleKV estimates layer sensitivity from the effect of controlled Value cache perturbations on the final output and derives model specific budgets through a scale invariant allocation rule under a fixed global cache constraint.
- Extensive experiments on LongBench show that RippleKV achieves the best average performance across three model families and multiple cache budgets while maintaining comparable runtime and memory efficiency.

Related Work

KV Cache Compression. KV cache compression reduces the memory and computational overhead of long-context inference by retaining only important tokens. StreamingLLM (Xiao et al. 2024) preserves attention sinks and recent tokens for stable streaming generation, while H_2O (Zhang et al. 2023) retains recent tokens and heavy hitters identified by accumulated attention scores. TOVA (Oren et al. 2024) performs online eviction by removing the least-attended token at each decoding step. SnapKV (Li et al. 2024) estimates token importance from an observation window at the end of the prompt. These methods primarily determine which tokens to retain within each layer, while generally using uniform or predefined cache capacities across layers.

Layer-Wise KV Cache Allocation. Recent studies have shown that different Transformer layers have heterogeneous cache requirements, motivating non-uniform allocation under a fixed global cache budget. PyramidKV (Cai et al. 2024) and PyramidInfer (Yang et al. 2024) allocate progressively smaller cache budgets to higher layers based on pyramidal information patterns. More recent methods dynamically determine layer budgets using task-dependent attention patterns, spatial and temporal attention dynamics, Key representation deviation, or intermediate reconstruction errors (Zhou et al. 2024; Qin et al. 2025; Kim et al. 2025; Shen et al. 2025; Lin et al. 2025). Existing approaches therefore mainly rely on structural priors, attention statistics, or layer-local discrepancies, which may not fully capture how compression effects propagate through subsequent layers. In contrast, RippleKV allocates layer-wise cache budgets based on how cache perturbations affect the final output distribution.

Rethinking Cache Allocation Across Layers

Compression Damage Does Not Follow a Fixed Depth Pattern

Existing layer-wise allocation strategies often assign cache budgets according to layer depth. However, layer position does not directly quantify how compressing a layer's KV cache affects the final model prediction. We therefore conduct an isolated single-layer compression analysis to obtain an empirical measure of layer sensitivity. For each layer ℓ, we compress only its KV cache while keeping all other layers uncompressed, using the same compression setting for every layer. Let p_i^{full} denote the predictive distribution of the model with the full cache for input x_i, and let $p_{i,\ell}^{\text{comp}}$ denote the corresponding distribution when only layer ℓ is compressed. We define the compression damage of layer ℓ as

$$D_\ell = \frac{1}{N}\sum_{i=1}^{N} D_{\text{KL}}\left(p_i^{\text{full}} \parallel p_{i,\ell}^{\text{comp}}\right). \tag{1}$$

A larger D_ℓ indicates that compressing layer ℓ causes a greater change in the predictive distribution and should therefore be treated more conservatively during cache allocation.

Across the 32 layers of Llama-3.1-8B-Instruct, the mean isolated compression damage spans 0.0030–0.0524, a $17.7\times$ difference, with the maximum occurring at an intermediate layer. Layer index correlates only weakly with compression damage (mean $|\rho| = 0.359$; Table 1), showing that depth captures only a coarse trend and can lead fixed schedules to underallocate cache to sensitive layers while overallocating it to tolerant ones.

Perturbation Response Captures Compression Sensitivity

Although D_ℓ directly measures compression damage, obtaining it requires applying a specific compression operation to every layer, making it unsuitable as a general allocation signal. We instead perturb the value cache of one layer at a time while leaving all other layers unchanged and measure the resulting shift in the final predictive distribution. Let $p_{i,\ell}^{\text{pert}}$ denote the output distribution after perturbing layer ℓ. We define the perturbation response as

$$s_\ell = \frac{1}{N}\sum_{i=1}^{N} D_{\text{KL}}\left(p_i^{\text{full}} \parallel p_{i,\ell}^{\text{pert}}\right). \tag{2}$$

Because s_ℓ is measured at the final output after the perturbation propagates through subsequent layers, it captures the end-to-end effect of modifying a layer's cache rather than its immediate local effect (Jing et al. 2026; Dong et al. 2020).

To assess ranking quality, we compute the absolute Spearman correlation between each signal and the isolated compression damage in Eq. (1).

Layer signal	Mean $\lvert\rho\rvert \uparrow$	Range
Layer index	0.359	0.218–0.480
RippleKV perturbation response	**0.799**	**0.723–0.857**

Table 1: Absolute Spearman correlation with isolated compression damage.

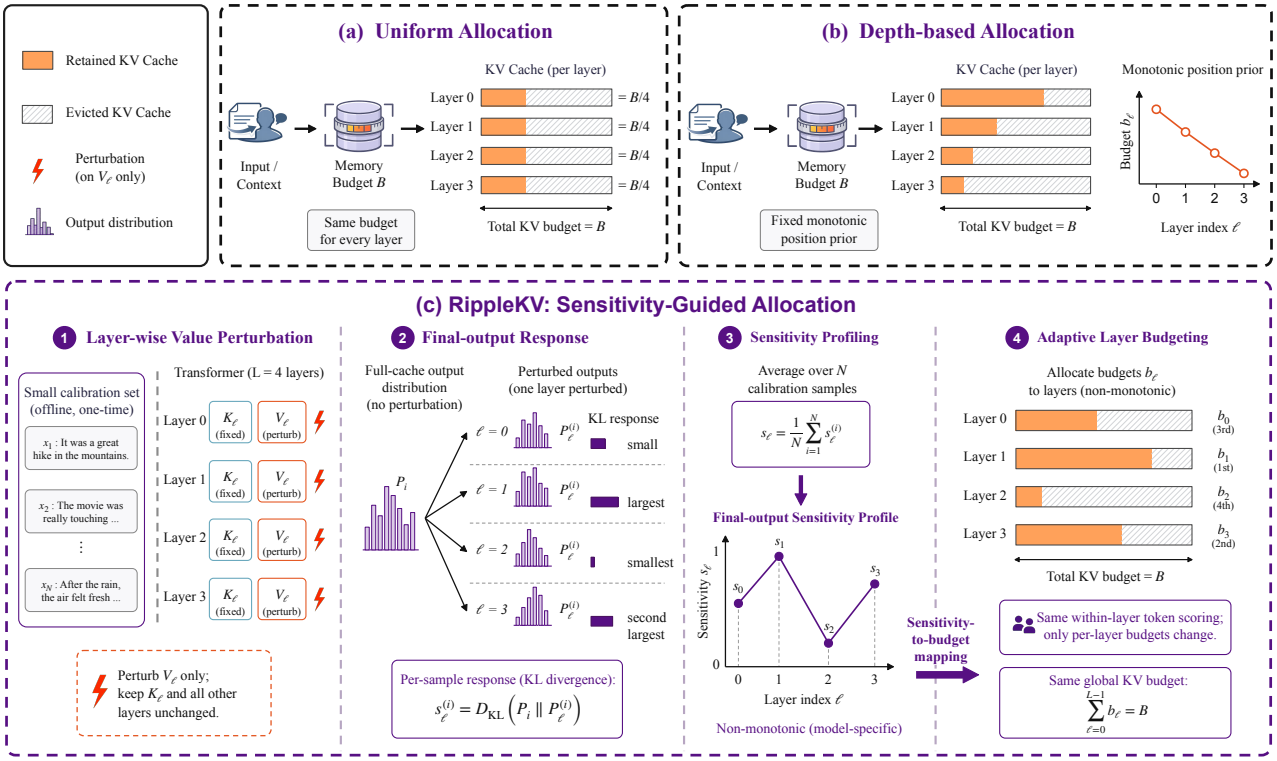

Figure 1: Overview of RippleKV. Unlike uniform or depth based allocation, RippleKV measures the final output response to layer specific Value cache perturbations and allocates the global cache budget according to the resulting sensitivity profile.

As shown in Table 1, the RippleKV perturbation response achieves a mean correlation of 0.799, substantially exceeding layer index (0.359). Its correlation remains between 0.723 and 0.857, showing that the final-output response consistently captures compression sensitivity beyond the coarse depth prior. This result motivates its use for layer-wise cache allocation.

Methodology

Problem Formulation

Given a decoder only Transformer with L layers and an input sequence $\mathbf{x} = \{x_t\}_{t=1}^{T}$, the prefilling stage produces a KV cache at each layer:

$$\mathcal{C}_\ell = \{(\mathbf{k}_{\ell,t}, \mathbf{v}_{\ell,t})\}_{t=1}^{T}, \qquad \ell = 0, \ldots, L-1, \tag{3}$$

where $\mathbf{k}_{\ell,t}$ and $\mathbf{v}_{\ell,t}$ denote the key and value states of token t at layer ℓ. During autoregressive decoding, each new query attends to the cached states of previous tokens. The total KV cache size therefore grows linearly with both the sequence length and the number of layers, while the attention cost increases with the number of retained tokens.

KV cache compression reduces these costs by retaining only a subset of token positions at each layer. Let $\mathcal{S}_\ell \subseteq \{1, \ldots, T\}$ denote the positions retained at layer ℓ. The resulting compressed cache is

$$\widehat{\mathcal{C}}_\ell = \{(\mathbf{k}_{\ell,t}, \mathbf{v}_{\ell,t}) \mid t \in \mathcal{S}_\ell\}, \tag{4}$$

where $b_\ell = |\mathcal{S}_\ell|$ is the cache budget assigned to layer ℓ. Under a global cache budget B, the layer budgets satisfy

$$\sum_{\ell=0}^{L-1} b_\ell = B. \tag{5}$$

Given a fixed token selection strategy, we seek a layer budget allocation $\mathbf{b} = \{b_\ell\}_{\ell=0}^{L-1}$ that preserves generation quality under the global budget constraint in Eq. (5).

Overview of RippleKV

As illustrated in Figure 1, RippleKV consists of offline sensitivity profiling followed by cache compression during inference. Given a small calibration set, it first obtains the reference output distributions using the full cache. RippleKV then perturbs the Value cache of one compressible layer at a time, while keeping its Key cache and all other caches unchanged. The resulting change in the final output distribution is measured by KL divergence and averaged across calibration examples to form a sensitivity profile specific to the model. Since the response is measured after propagating through the subsequent layers, it captures the influence of each layer on the final prediction.

The sensitivity profile is then converted into layer budgets $\mathbf{b} = \{b_\ell\}_{\ell \in \mathcal{A}}$ under the global constraint $\sum_{\ell \in \mathcal{A}} b_\ell = B$. Layers with stronger responses receive larger budgets, while less sensitive layers are compressed more aggressively. This produces a nonmonotonic allocation while preserving the

Algorithm 1 RippleKV

Input: Model $\mathcal{M}$; calibration set $\mathcal{D}_{\text{cal}} = \{\mathbf{x}_i\}_{i=1}^{N}$; compressible layers $\mathcal{A}$; global cache budget B; perturbation strength α; target budget ratio r; weight bound γ
Output: Sensitivity profile $\mathbf{s}$ and layer budgets $\mathbf{b}$

Sensitivity Profiling
$s_\ell \leftarrow 0, \quad \forall \ell \in \mathcal{A}$
for all $\mathbf{x}_i \in \mathcal{D}_{\text{cal}}$ **do**
  $\mathbf{P}_i \leftarrow \textsc{ReferencePredict}(\mathcal{M}, \mathbf{x}_i)$
  for all $\ell \in \mathcal{A}$ **do**
    $\mathbf{Q}_{i,\ell} \leftarrow \textsc{PerturbedPredict}(\mathcal{M}, \mathbf{x}_i, \ell, \alpha)$
    $d_{i,\ell} \leftarrow \textsc{WindowKL}(\mathbf{P}_i, \mathbf{Q}_{i,\ell})$
    $s_\ell \leftarrow s_\ell + d_{i,\ell}$
  end for
end for
$s_\ell \leftarrow s_\ell / N, \quad \forall \ell \in \mathcal{A}$

Budget Allocation
$s_{\min} \leftarrow \min_{\ell \in \mathcal{A}} s_\ell$, $s_{\max} \leftarrow \max_{\ell \in \mathcal{A}} s_\ell$
for all $\ell \in \mathcal{A}$ **do**
  $\widehat{s}_\ell \leftarrow \dfrac{s_\ell - s_{\min}}{s_{\max} - s_{\min} + \epsilon}$
  $\widetilde{w}_\ell \leftarrow r^{\widehat{s}_\ell}$
end for
$\mathbf{w} \leftarrow \textsc{ClipNormalize}(\widetilde{\mathbf{w}}, \gamma)$
for all $\ell \in \mathcal{A}$ **do**
  $b_\ell \leftarrow \text{round}\left(\dfrac{B}{|\mathcal{A}|} w_\ell\right)$
end for
return $\mathbf{s}, \mathbf{b}$

global cache budget. During inference, RippleKV changes only the budget assigned to each layer and retains the original token scoring strategy. The sensitivity profile is computed once offline and reused without additional model evaluations. The complete procedure is summarized in Algorithm 1.

Layer Sensitivity Estimation

Given a calibration set $\mathcal{D}_{\text{cal}} = \{\mathbf{x}_i\}_{i=1}^{N}$, we divide each example into a prefix used to construct the KV cache and an evaluation window $\mathcal{W}_i$. We first run the model with the intact cache and record the reference output distribution $\mathbf{P}_{i,t}$ at each position $t \in \mathcal{W}_i$. For every compressible layer ℓ, we then apply a controlled perturbation to its prefix Value cache and evaluate the same output window. The intervention is restricted to layer ℓ: its Key cache and the caches of all other layers remain unchanged, and no cached position is removed during profiling.

Let $\mathbf{v}_{i,\ell,h,u}$ denote the Value vector at layer ℓ, KV head h, and prefix position u. For each position eligible for compression, RippleKV constructs

$$\widetilde{\mathbf{v}}_{i,\ell,h,u} = \mathbf{v}_{i,\ell,h,u} + \alpha \left\|\mathbf{v}_{i,\ell,h,u}\right\|_2 \boldsymbol{\epsilon}_{i,\ell,h,u}, \tag{6}$$

where $\boldsymbol{\epsilon}_{i,\ell,h,u} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ and α controls the perturbation strength. Scaling the noise by the norm of each Value vector adapts the perturbation to its local magnitude and reduces sensitivity to absolute activation scale differences across layers and tokens. Positions protected by the base compression method are excluded from perturbation. Since the Key cache is fixed, this intervention leaves the attention weights at the target layer unchanged while modifying the content aggregated from its cached values.

Let $\mathbf{Q}_{i,\ell,t}$ denote the output distribution at position t after perturbing layer ℓ. We measure the response of layer ℓ on example $\mathbf{x}_i$ by averaging the output divergence over the evaluation window:

$$d_{i,\ell} = \frac{1}{|\mathcal{W}_i|} \sum_{t \in \mathcal{W}_i} D_{\text{KL}}\left(\mathbf{P}_{i,t} \parallel \mathbf{Q}_{i,\ell,t}\right). \tag{7}$$

The sensitivity estimate for layer ℓ is then obtained by averaging across the calibration set:

$$s_\ell = \frac{1}{N} \sum_{i=1}^{N} d_{i,\ell}. \tag{8}$$

A larger s_ℓ indicates that the model output is more responsive to a controlled Value cache perturbation at layer ℓ. Repeating the procedure for all compressible layers yields the model specific sensitivity profile $\mathbf{s} = \{s_\ell\}_{\ell \in \mathcal{A}}$.

The profiling procedure requires no gradient computation and is performed only once for each model. The resulting sensitivity profile is independent of the target cache budget and can therefore be reused across different compression settings. During inference, RippleKV directly applies the corresponding layer budgets without additional perturbation evaluations.

Sensitivity Guided Layer Budget Allocation

Given the sensitivity profile $\mathbf{s} = \{s_\ell\}_{\ell \in \mathcal{A}}$, RippleKV converts the response of each layer into a cache budget, where $\mathcal{A}$ denotes the set of compressible layers. Raw KL responses may have different numerical scales across models and calibration sets. Directly using them as budget weights would therefore make the allocation sensitive to their absolute magnitude. We first normalize the sensitivity values as

$$\widehat{s}_\ell = \frac{s_\ell - s_{\min}}{s_{\max} - s_{\min}}, \qquad \ell \in \mathcal{A}, \tag{9}$$

where $s_{\min} = \min_{\ell \in \mathcal{A}} s_\ell$ and $s_{\max} = \max_{\ell \in \mathcal{A}} s_\ell$. When all layers have identical sensitivity, we set $\widehat{s}_\ell = 0$ for every layer, yielding uniform allocation. Otherwise, $\widehat{s}_\ell \in [0, 1]$ preserves the ordering of the sensitivity scores and makes the allocation invariant to positive affine transformations of the raw profile.

We then map each normalized score to a positive budget multiplier:

$$\widetilde{w}_\ell = \exp\left(\log r \cdot \widehat{s}_\ell\right) = r^{\widehat{s}_\ell}, \qquad \ell \in \mathcal{A}, \tag{10}$$

where $r \geq 1$ controls the allocation disparity across layers. For a nonconstant sensitivity profile, the ratio between the largest and smallest multipliers before clipping is exactly r. Thus, $r = 1$ recovers uniform allocation, while increasing r assigns progressively more cache to sensitive layers. The mapping is monotonic, so a layer with a larger sensitivity score cannot receive a smaller initial multiplier than a less sensitive layer.

Method	Single-Document QA			Multi-Document QA			Summarization			Few-shot Learning			Synthetic		Code		Avg.
	NrtvQA	Qasper	MF-en	HotpotQA	2WikiMQ	Musique	GovReport	QMSum	MultiNews	TREC	TriviaQA	SAMSum	PCount	PRe	Lcc	RB-P	
Llama-3.1-8B-Instruct, Full Cache																	
Full Cache	30.72	47.06	55.28	59.51	51.83	32.66	35.25	24.94	27.05	29.50	91.71	40.92	10.70	100.00	54.13	47.58	46.18
Llama-3.1-8B-Instruct, KV Cache Budget = 10%																	
StreamingLLM	21.67	18.44	23.03	37.24	22.28	14.06	25.03	19.13	20.05	28.00	90.70	35.31	4.00	16.00	52.35	**52.57**	29.99
H_2O	15.72	**28.97**	20.19	32.81	**28.04**	10.39	**27.92**	**21.25**	**23.56**	**39.00**	89.35	34.80	**10.42**	54.00	50.17	46.08	33.29
SnapKV	**24.33**	20.80	23.63	**44.74**	24.10	19.88	25.56	19.80	20.08	33.50	91.49	39.84	5.50	54.50	51.09	48.94	34.24
PyramidKV	23.39	20.76	23.30	44.71	24.78	18.66	24.98	20.68	19.96	33.50	**91.66**	39.59	6.00	54.50	51.06	48.63	34.13
RippleKV	23.62	22.54	**24.22**	44.37	27.84	**20.46**	25.56	20.35	20.40	34.00	91.49	**40.62**	6.00	**58.50**	**52.59**	48.57	**35.07**
Llama-3.1-8B-Instruct, KV Cache Budget = 20%																	
StreamingLLM	22.18	23.09	24.92	41.44	27.23	18.16	28.08	20.14	22.68	28.00	91.71	34.94	6.00	28.50	52.18	**51.16**	32.53
H_2O	20.36	**33.74**	27.47	39.07	35.45	16.34	**30.43**	**22.93**	**25.56**	**39.00**	89.83	36.91	**9.54**	76.50	50.78	47.81	37.61
SnapKV	**28.57**	25.39	31.19	54.40	38.20	**25.35**	27.92	21.70	22.75	35.00	**92.31**	41.13	8.16	86.50	53.42	46.92	39.93
PyramidKV	27.16	25.53	29.92	**54.64**	**40.51**	24.29	27.34	21.65	22.74	32.00	91.56	41.33	8.56	86.50	53.53	47.32	39.66
RippleKV	26.50	27.88	**32.87**	52.68	38.80	24.85	28.24	21.98	22.75	35.00	91.38	**42.22**	8.10	**90.00**	**54.52**	47.81	**40.35**
Llama-3.1-8B-Instruct, KV Cache Budget = 30%																	
StreamingLLM	24.23	28.29	26.57	43.67	30.92	21.10	29.32	21.02	24.23	31.50	**91.49**	36.79	6.25	36.50	50.90	**49.86**	34.54
H_2O	20.20	**37.40**	34.39	45.50	42.38	19.83	**31.69**	**23.26**	**25.67**	**40.00**	90.99	37.92	9.20	84.00	50.61	48.10	40.07
SnapKV	27.43	30.93	38.05	54.77	**46.72**	27.74	29.84	22.67	24.07	38.50	91.41	41.98	9.55	92.50	53.25	46.77	42.26
PyramidKV	27.45	30.24	36.83	**55.62**	44.43	26.81	28.75	22.81	23.91	35.50	91.26	41.21	**10.55**	**95.00**	53.53	47.21	41.94
RippleKV	**28.79**	33.53	**40.74**	54.84	45.89	**29.08**	30.15	22.93	24.00	38.50	91.41	**42.33**	9.55	**95.00**	**53.79**	46.91	**42.97**
Qwen2.5-7B-Instruct, KV Cache Budget = 10%																	
StreamingLLM	20.40	16.67	22.56	25.12	17.53	7.79	26.29	18.70	18.04	**44.75**	69.49	36.16	2.50	15.50	61.61	**63.14**	29.14
H_2O	13.39	**20.30**	**27.54**	26.03	**24.16**	6.05	**29.32**	**20.78**	**22.09**	43.00	81.36	35.92	4.00	38.50	63.06	62.69	32.39
SnapKV	19.98	14.04	25.28	36.77	23.35	**16.76**	26.30	18.38	18.87	42.00	87.47	40.14	6.00	46.00	**63.37**	61.56	34.14
PyramidKV	17.37	13.97	25.02	30.18	23.09	9.15	25.49	17.99	19.04	42.00	87.55	40.32	3.50	35.83	62.96	61.04	32.16
RippleKV	**20.52**	15.05	24.97	**37.05**	23.84	15.94	26.72	18.48	19.31	41.00	**88.00**	**40.59**	**7.00**	**47.50**	62.65	62.06	**34.42**
Mistral-7B-Instruct-v0.3, KV Cache Budget = 10%																	
StreamingLLM	**18.65**	13.10	23.29	31.23	20.23	14.02	25.35	19.74	18.69	35.00	52.14	32.20	4.00	14.50	50.67	52.89	26.61
H_2O	9.06	**26.99**	24.98	20.85	23.42	10.68	**28.31**	**21.52**	**24.25**	**47.50**	85.45	**39.29**	**4.72**	47.50	45.23	51.71	31.97
SnapKV	17.69	13.61	**30.62**	33.04	22.53	**18.42**	25.32	20.24	20.54	38.00	87.90	28.51	3.78	53.50	52.54	**54.42**	32.54
PyramidKV	16.84	13.58	30.15	34.50	22.99	15.23	24.95	20.49	20.70	38.50	**88.21**	29.12	3.96	49.50	52.43	54.38	32.22
RippleKV	18.37	13.73	29.72	**35.49**	**23.78**	17.69	25.32	20.50	20.27	39.25	87.78	29.87	4.55	**57.50**	**52.85**	54.32	**33.19**

Table 2: Performance comparison over LongBench datasets. The best result is highlighted in bold, and the second best is underlined.

To avoid extreme allocations, RippleKV applies clipping followed by normalization:

$$\mathbf{w} = \mathrm{ClipNormalize}\left(\widetilde{\mathbf{w}}, \gamma\right), \qquad \frac{1}{|\mathcal{A}|}\sum_{\ell\in\mathcal{A}} w_\ell = 1, \quad (11)$$

where $\gamma \geq 1$ specifies the clipping interval $[1/\gamma, \gamma]$. The multipliers are clipped using this interval and then rescaled to have mean one over the compressible layers. This step suppresses unusually large or small allocations while preserving the overall cache budget and the relative pattern induced by the sensitivity profile.

Finally, the budget assigned to layer ℓ is

$$b_\ell = \mathrm{round}\left(\frac{B}{|\mathcal{A}|} w_\ell\right), \qquad \ell \in \mathcal{A}, \quad (12)$$

where $B/|\mathcal{A}|$ is the average budget under uniform allocation. Layers with stronger perturbation responses therefore receive budgets above the uniform average, whereas less sensitive layers are compressed more aggressively. Since the sensitivity profile determines only the relative allocation, the same profile can be reused under different global cache budgets by changing B.

RippleKV changes only how the global budget is distributed across layers. Given b_ℓ, the base compression method applies its original token scoring and selection rule to retain the required number of cached positions at layer ℓ. In our implementation, we preserve the SnapKV scoring rule and replace only its uniform layer budgets with the sensitivity guided allocation. This isolates the effect of layer budget allocation from changes in token importance estimation and introduces no additional model evaluation during inference.

Experiments

Experimental Settings

Backbone Models. We evaluate RippleKV on three widely used instruction-tuned LLMs: Llama-3.1-8B-Instruct (Grattafiori et al. 2024), Mistral-7B-Instruct-v0.3 (Jiang et al. 2023), and Qwen2.5-7B-Instruct (Hui et al. 2024). These models represent different model families, enabling us to assess the generalizability of RippleKV across Transformer architectures.

Datasets. We conduct experiments on LongBench (Bai et al. 2024), a standard benchmark for long-context understanding. Following prior KV cache compression studies, we report results across six task categories: single-document QA, multi-document QA, summarization, few-shot learning, synthetic tasks, and code completion.

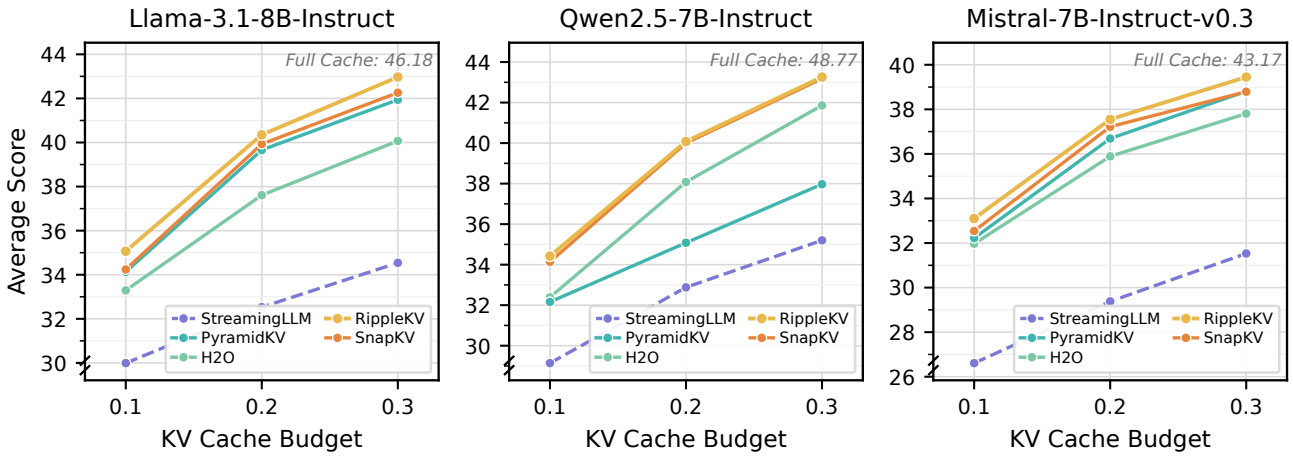

Figure 2: Average performance on LongBench under different KV cache budgets.

Baselines. We use Full Cache, which preserves all KV states, as the reference setting. We compare RippleKV with StreamingLLM (Xiao et al. 2024), which retains attention sinks and recent tokens; H_2O (Zhang et al. 2023), which evicts tokens based on accumulated attention scores; SnapKV (Li et al. 2024), which selects tokens using a recent observation window; and PyramidKV (Cai et al. 2024), which allocates different cache budgets across layers. All compression methods use the same total cache budget.

Implementation Details. All experiments are implemented using Hugging Face Transformers and PyTorch on NVIDIA A100 80GB GPUs. We follow the standard LongBench evaluation protocol with task-specific decoding settings. All compression methods are evaluated at cache retention ratios of 10%, 20%, and 30%. Uniform baselines allocate the budget evenly across layers, whereas PyramidKV and RippleKV use non-uniform layer-wise allocation under the same total cache budget. We adopt the official or commonly used hyperparameters for all baselines.

Main Results on LongBench

Table 2 compares RippleKV with competing methods on LongBench. RippleKV achieves the highest average score across all five compressed settings. On Llama-3.1-8B-Instruct, it obtains average scores of 35.07, 40.35, and 42.97 under cache budgets of 10%, 20%, and 30%, outperforming the strongest baselines by 0.83, 0.42, and 0.71 points, respectively. This consistent advantage demonstrates the robustness of RippleKV across different compression levels. Moreover, increasing the cache budget from 10% to 30% narrows its gap to Full Cache from 11.11 to 3.21 points. Figure 2 shows a similar trend across all three backbone models: RippleKV consistently retains the lead and steadily approaches Full Cache as more cache is preserved.

RippleKV also generalizes consistently across model families. Under the challenging 10% cache budget, it achieves average scores of 34.42 on Qwen2.5-7B-Instruct and 33.19 on Mistral-7B-Instruct-v0.3, outperforming the strongest baselines by 0.28 and 0.65 points, respectively. Compared with the layer-aware PyramidKV, RippleKV improves the category-average Multi-Document QA score on Qwen2.5 from 20.81 to 25.61 and the Synthetic score on Mistral from 26.73 to 31.03. Although no method dominates every individual dataset, RippleKV consistently achieves the strongest aggregate performance across task categories. These findings demonstrate the effectiveness of allocating cache budgets according to end-to-end layer sensitivity rather than a fixed layer-wise allocation pattern.

Method	Latency (s)↓	Throughput (tok/s)↑
Context Length = 8K		
Full Cache	7.58	37.38
StreamingLLM	6.73	42.84
SnapKV	6.84	42.37
PyramidKV	6.88	42.09
RippleKV	6.67	43.48
Context Length = 128K		
Full Cache	59.46	11.85
StreamingLLM	43.35	42.23
SnapKV	43.87	40.92
PyramidKV	43.87	42.51
RippleKV	43.02	42.99

Table 3: Efficiency comparison at different context lengths.

Efficiency Analysis

Table 3 reports end-to-end inference latency and decoding throughput at representative context lengths of 8K and 128K, with additional results provided in the Appendix. At 8K, RippleKV achieves 6.67 seconds of latency and 43.48 tokens per second; at 128K, it records 43.02 seconds and 42.99 tokens per second, respectively. RippleKV thus matches or slightly outperforms existing KV cache compression methods in runtime efficiency, indicating that layer-wise budget allocation

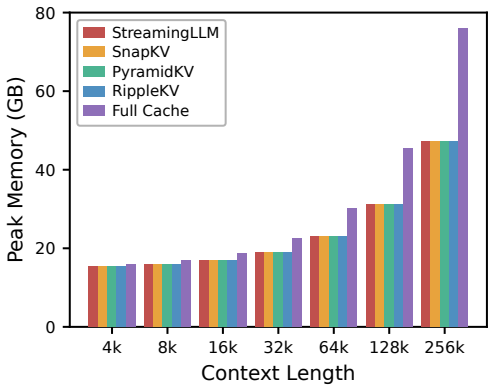

Figure 3: Peak memory across different context lengths.

Category	w/o LBA	w/o DS	Full
Single-Doc QA	22.79	22.26	**23.46**
Multi-Doc QA	29.52	29.35	**30.89**
Summarization	21.79	21.85	**22.10**
Few-shot	**55.45**	54.87	55.37
Synthetic	30.00	30.25	**32.25**
Code	47.16	47.40	**50.58**

Table 4: Ablation results across LongBench task categories.

adds no measurable online overhead. The sensitivity profile is computed once during offline calibration, while inference only applies the precomputed budgets.

Figure 3 further compares peak memory usage across context lengths from 4K to 256K. RippleKV closely matches the memory footprint of StreamingLLM, SnapKV, and PyramidKV, showing that redistributing cache capacity across layers does not increase the total cache budget. Compared with Full Cache, RippleKV reduces peak memory from 45.53 to 31.13 GiB at 128K and from 76.09 to 47.29 GiB at 256K, corresponding to reductions of 31.6% and 37.8%, respectively. These results show that RippleKV improves cache allocation while preserving the runtime and memory efficiency of existing KV cache compression methods.

Ablation Study

We examine two key design choices in RippleKV: layer-wise budget allocation (LBA) and downstream sensitivity (DS). For *w/o LBA*, we replace sensitivity guided allocation with a uniform cache budget across layers. For *w/o DS*, we retain the same data driven allocation procedure and global cache budget, but replace the downstream signal, measured by the change in the final predictive distribution induced by perturbing the Value cache, with a layer local response score. This design isolates the contribution of downstream sensitivity measured at the model output from that of varying cache budgets across layers.

As shown in Table 4, the full model performs best on five of the six task categories. Removing LBA causes drops of 3.42 points on Code, 2.25 points on Synthetic, and 1.37 points on Multi-Document QA. Replacing downstream sensitivity with the layer-local signal yields corresponding drops of 3.18, 2.00, and 1.54 points. Although uniform allocation improves Few-shot Learning by 0.08 points, it underperforms the full model on all remaining categories. These results show that non-uniform layer allocation improves performance under a fixed cache budget, while downstream sensitivity provides a stronger allocation signal than layer-local responses.

Task Category	Allocation Ratio R		
	1.25	1.50	1.75
Single-Doc QA	23.82	23.46	23.57
Multi-Doc QA	29.74	30.89	30.33
Summarization	21.81	22.10	22.01
Few-shot	54.99	55.37	55.47
Synthetic	31.50	32.25	31.75
Code	50.29	50.58	50.31
Average	34.67	35.07	34.89

Table 5: Sensitivity of RippleKV to the allocation ratio R across LongBench task categories.

Hyperparameter Sensitivity

We study the sensitivity of RippleKV to the allocation ratio R, which controls the budget disparity between more and less sensitive layers. We vary R over $\{1.25, 1.50, 1.75\}$ while keeping the global cache budget and all other settings fixed. As shown in Table 5, the corresponding average LongBench scores are 34.67, 35.07, and 34.89, with a maximum difference of only 0.41 points. Performance also remains stable across task categories, although the preferred value varies slightly by category. We therefore use $R = 1.50$, which achieves the highest overall score, as the default setting. These results indicate that RippleKV is robust to moderate changes in the allocation ratio and does not rely on narrowly tuned hyperparameters.

Conclusion

In this work, we introduced RippleKV, a cross-layer KV cache allocation method guided by perturbation propagation. We showed that layer-wise compression damage is highly heterogeneous and does not follow a fixed depth pattern, limiting the reliability of allocation strategies based on structural priors. RippleKV instead measures how controlled Value cache perturbations affect the final predictive distribution and uses the resulting model dependent sensitivity profile to distribute a fixed global cache budget across layers. Experiments on LongBench across three model families and multiple cache budgets show that RippleKV consistently achieves the strongest average performance among the evaluated methods while maintaining comparable inference efficiency and memory usage. These results demonstrate that downstream output sensitivity provides an effective basis for cross-layer KV cache allocation.

References

Adnan, M.; Arunkumar, A.; Jain, G.; Nair, P. J.; Soloveychik, I.; and Kamath, P. 2024. Keyformer: Kv cache reduction through key tokens selection for efficient generative inference. *Proceedings of Machine Learning and Systems*, 6: 114–127.

Bai, Y.; Lv, X.; Zhang, J.; Lyu, H.; Tang, J.; Huang, Z.; Du, Z.; Liu, X.; Zeng, A.; Hou, L.; et al. 2024. 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)*, 3119–3137.

Cai, Z.; Zhang, Y.; Gao, B.; Liu, Y.; Li, Y.; Liu, T.; Lu, K.; Xiong, W.; Dong, Y.; Hu, J.; et al. 2024. Pyramidkv: Dynamic kv cache compression based on pyramidal information funneling. *arXiv preprint arXiv:2406.02069*.

Dong, Z.; Yao, Z.; Arfeen, D.; Gholami, A.; Mahoney, M. W.; and Keutzer, K. 2020. Hawq-v2: Hessian aware trace-weighted quantization of neural networks. *Advances in neural information processing systems*, 33: 18518–18529.

Grattafiori, A.; Dubey, A.; Jauhri, A.; Pandey, A.; Kadian, A.; Al-Dahle, A.; Letman, A.; Mathur, A.; Schelten, A.; Vaughan, A.; et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Hui, B.; Yang, J.; Cui, Z.; Yang, J.; Liu, D.; Zhang, L.; Liu, T.; Zhang, J.; Yu, B.; Lu, K.; et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Jiang, A. Q.; Sablayrolles, A.; Mensch, A.; Bamford, C.; Chaplot, D. S.; de Las Casas, D.; Bressand, F.; Lengyel, G.; Lample, G.; Saulnier, L.; Lavaud, L. R.; Lachaux, M.; Stock, P.; Scao, T. L.; Lavril, T.; Wang, T.; Lacroix, T.; and Sayed, W. E. 2023. Mistral 7B. *CoRR*, abs/2310.06825.

Jing, T.; Wu, N.; Kang, C.; Yu, D.; Li, C.; and Liu, P. 2026. Beyond Layer Importance in Layer-wise Sparsity: An Inter-Layer Perturbation-Absorption Perspective. *arXiv preprint arXiv:2606.15161*.

Kamalloo, E.; Dziri, N.; Clarke, C.; and Rafiei, D. 2023. 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)*, 5591–5606.

Kim, M.; Kundu, A.; Kim, H.; Dixit, R.; and Cho, M. 2025. EpiCache: Episodic KV Cache Management for Long Conversational Question Answering. *CoRR*, abs/2509.17396.

Li, Y.; Huang, Y.; Yang, B.; Venkitesh, B.; Locatelli, A.; Ye, H.; Cai, T.; Lewis, P.; and Chen, D. 2024. Snapkv: Llm knows what you are looking for before generation. *Advances in Neural Information Processing Systems*, 37: 22947–22970.

Lin, X.; Wang, J.; Kondrateva, O.; Shi, Y.; Li, B.; and Zhang, G. L. 2025. CompressKV: Semantic Retrieval Heads Know What Tokens are Not Important Before Generation. *arXiv preprint arXiv:2508.02401*.

Liu, Z.; Desai, A.; Liao, F.; Wang, W.; Xie, V.; Xu, Z.; Kyrillidis, A.; and Shrivastava, A. 2023. Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time. *Advances in Neural Information Processing Systems*, 36: 52342–52364.

Liu, Z.; Yuan, J.; Jin, H.; Zhong, S.; Xu, Z.; Braverman, V.; Chen, B.; and Hu, X. 2024. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. *arXiv preprint arXiv:2402.02750*.

Oren, M.; Hassid, M.; Yarden, N.; Adi, Y.; and Schwartz, R. 2024. Transformers are multi-state rnns. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, 18724–18741.

Qin, Z.; Cao, Y.; Lin, M.; Hu, W.; Fan, S.; Cheng, K.; Lin, W.; and Li, J. 2025. CAKE: Cascading and Adaptive KV Cache Eviction with Layer Preferences. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Roziere, B.; Gehring, J.; Gloeckle, F.; Sootla, S.; Gat, I.; Tan, X. E.; Adi, Y.; Liu, J.; Sauvestre, R.; Remez, T.; et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Shaham, U.; Ivgi, M.; Efrat, A.; Berant, J.; and Levy, O. 2023. ZeroSCROLLS: A zero-shot benchmark for long text understanding. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, 7977–7989.

Shen, Y.; Yuan, S.; Zhang, Z.; Wang, X.; Jiang, D.; and Nguyen, C. 2025. LAVa: Layer-wise KV Cache Eviction with Dynamic Budget Allocation. In Christodoulopoulos, C.; Chakraborty, T.; Rose, C.; and Peng, V., eds., *Findings of the Association for Computational Linguistics: EMNLP 2025, Suzhou, China, November 4-9, 2025*, 13672–13692. Association for Computational Linguistics.

Xiao, G.; Tian, Y.; Chen, B.; Han, S.; and Lewis, M. 2024. Efficient streaming language models with attention sinks. In *International Conference on Learning Representations*, volume 2024, 21875–21895.

Yang, D.; Han, X.; Gao, Y.; Hu, Y.; Zhang, S.; and Zhao, H. 2024. Pyramidinfer: Pyramid kv cache compression for high-throughput llm inference. In *Findings of the Association for Computational Linguistics: ACL 2024*, 3258–3270.

Zhang, T.; Ladhak, F.; Durmus, E.; Liang, P.; McKeown, K.; and Hashimoto, T. B. 2024. Benchmarking large language models for news summarization. *Transactions of the Association for Computational Linguistics*, 12: 39–57.

Zhang, Z.; Sheng, Y.; Zhou, T.; Chen, T.; Zheng, L.; Cai, R.; Song, Z.; Tian, Y.; Ré, C.; Barrett, C.; et al. 2023. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems*, 36: 34661–34710.

Zhou, X.; Wang, W.; Zeng, M.; Guo, J.; Liu, X.; Shen, L.; Zhang, M.; and Ding, L. 2024. DynamicKV: Task-aware adaptive KV cache compression for long context LLMs. *arXiv preprint arXiv:2412.14838*.