ROTATION-ALIGNED KEY CHANNEL PRUNING FOR EFFICIENT VISION-LANGUAGE MODEL INFERENCE

Anonymous authors
Paper under double-blind review

ABSTRACT

Vision-Language Models suffer severe KV cache pressure at inference, as a single image often encodes into thousands of tokens. To reduce this cost, most prior works exploit token sparsity through token pruning, but permanently discarding visual content substantially degrades fine-grained perception. This motivates a complementary axis, *feature sparsity*, which compresses the channel dimension to preserve more visual tokens at the same KV cache budget. Existing Key channel pruning methods, however, face a structural trade-off: token-wise channel pruning is accurate but unstructured and slow, while head-wise pruning is hardware-friendly but degrades at high compression. We attribute this degradation to informative Key directions that change across tokens and inputs, so no fixed channel subset serves them all. We propose **RotateK**, a training-free method that builds a query-weighted Key basis for each input at prefill, allowing a head-wise mask to retain attention-relevant information. To make this practical, we implement a graph-capturable top-k eigensolver, a fused sparse-channel decode kernel, and a paged KV layout in vLLM. Across six VLMs, up to 32B parameters, and four token pruning methods, RotateK consistently improves over token pruning alone at matched KV budgets and is comparable in accuracy to token-wise channel pruning, while increasing KV capacity and generation throughput.

1 INTRODUCTION

Recent progress in Large Language Models (LLMs) has naturally extended to multi-modal systems such as Vision-Language Models (VLMs) (Liu et al., 2024b; Chen et al., 2024b; Bai et al., 2025). By inheriting the broad world knowledge and reasoning capabilities of their language backbones, VLMs have demonstrated remarkable visual understanding. However, they also come with the inference bottlenecks of LLMs, most notably the KV cache, whose size grows linearly with sequence length. This issue is more severe in the visual setting, where a single image is encoded as hundreds to thousands of tokens (Zhang et al., 2024a; Shen et al., 2024; Chen et al., 2025), and video or multi-image inputs can easily reach tens of thousands (Tu et al., 2024). As a result, the visual KV cache often dominates memory usage during inference (Xing et al., 2024; Chen et al., 2024a; Wan et al., 2024), making its compression a key requirement for practical VLM deployment.

In recent years, visual KV cache compression has been predominantly driven by *token pruning* (Yang et al., 2025; Chen et al., 2024a; Alvar et al., 2025; Khaki et al., 2025; Ye et al., 2025), which permanently discards visual tokens deemed less informative. While effective on many benchmarks, this irreversible loss degrades tasks whose relevant content is spread across the scene (Alvar et al., 2025), such as document understanding (Masry et al., 2022; Liu et al., 2024d) and visual grounding (Wang et al., 2024). The KV cache, however, scales not only with the number of tokens L, but also with the channel dimension d. This suggests that, rather than relying solely on token sparsity, compression can also be achieved along the feature dimension. By incorporating *feature sparsity* via pruning less informative channels, the same memory budget can be met while retaining more visual tokens, thereby mitigating the information loss inherent in token pruning (Figure 1).

In practice, prior feature-dimension compression methods have primarily focused on pruning Key channels (Xu et al., 2024; Liao et al., 2026; Zhang et al., 2025c). Existing Key channel pruning methods largely fall into two regimes: head-wise pruning (Xu et al., 2024; Zhang et al., 2025c), where a shared set of channels is pruned across tokens within each head, and token-wise pruning (Liao et al.,

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.

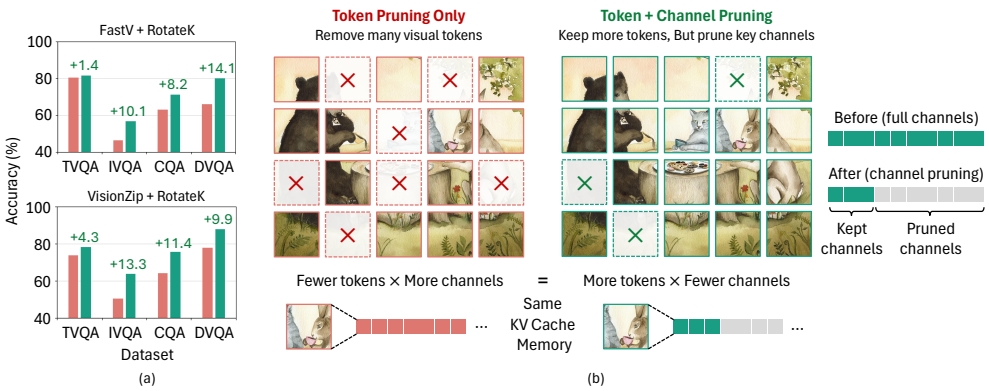


Figure 1: **Token Pruning only vs. Token and Channel Pruning**. (a) Comparison between token pruning only (red) and joint token-channel pruning with RotateK (green) under similar KV cache budgets. For FastV, we compare 0.20× tokens against FastV (0.30× tokens) + RotateK (0.25× channels). For VisionZip, we compare 0.22× tokens against VisionZip (0.35× tokens) + RotateK (0.25× channels). (b) Unlike token pruning, which removes many visual tokens entirely, our approach preserves more visual tokens by additionally pruning less informative key channels under the same KV cache memory budget.

2026), where pruning decisions vary per token. These approaches reflect a trade-off between efficiency and expressivity: token-wise pruning achieves higher accuracy (Liao et al., 2026), suggesting that channel importance varies significantly across tokens, but requires a mask of size $\mathcal{O}(Ld)$ and incurs substantial memory and IO overhead during decoding; in contrast, head-wise pruning is far more efficient with a lightweight $\mathcal{O}(d)$ mask, but suffers from notable accuracy degradation at high compression ratios due to its inability to capture per-token differences. This raises a central question: Can token-dependent channel importance be captured within a structured, head-wise mask?

In this work, we argue that the limitation lies not in the head-wise mask itself, but in the channel basis in which it is applied. This token-dependent channel importance is largely a property of the representation: informative directions in the Key space are not aligned with individual channels. If the Key space is rotated so that these directions concentrate in a few shared channels, a single head-wise mask can retain them for all tokens. The appropriate rotation, however, shifts with the visual input. Consequently, a basis fixed in advance, whether calibrated offline or lightly trained (Lin et al., 2025), cannot fully capture the informative directions of every input. We propose **RotateK**, which constructs the basis per input at prefill, without training. Because pruning perturbs attention scores rather than Keys, RotateK weights each channel by the magnitude of recent queries before extracting the basis, concentrating the information that future queries actually use into a few leading channels. The basis is shared across tokens within each head, so pruning in it remains structured, which enables efficient decoding and paged serving. Our key contributions are as follows:

- **Basis mismatch in visual Key channels.** Comparing token-wise and head-wise masks, we find that informative Key channels are largely token-specific: of the 32 channels a token-wise mask keeps per token, only about 3 are shared by all tokens, and a head-wise mask spends a quarter of its budget on channels most tokens do not need. This holds across 4,416 (layer, head) pairs from two model families and four datasets (Sec. 2).
- **A training-free, input-adaptive Key basis.** While pruning head-wise, RotateK is comparable in accuracy to token-wise pruning (Sec. 4.2) and outperforms bases fixed in advance, whether calibrated, lightly trained, or learned as a channel mask, at aggressive compression. Combined with token pruning, it improves over token-only pruning in all configurations at matched KV budgets (Sec. 4.1; Tables 17, 18).
- **Real speedups from structured sparsity.** RotateK yields measured speedups, not only memory savings. Its head-wise sparsity enables a fused kernel over compressed Keys and a fixed-width paged layout in vLLM, achieving 1.60× KV capacity and up to 1.29× per-step decode speedup over full-channel attention. A graph-capturable eigensolver keeps basis construction at 0.60 ms per layer (2.9% of prefill at 8K visual tokens) (Secs. 3.2, 4.3).

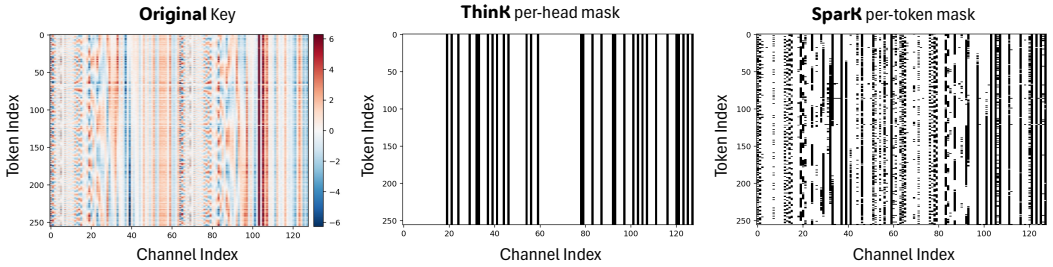


Figure 2: **Key Channel Pruning Masks**. (a) Visualization of the original visual Key states, where few channels exhibit strong outlier patterns across tokens. (b) ThinK applies a shared head-wise mask, retaining the same channels for all tokens. (c) SparK applies token-wise masks that vary across tokens, reflecting token-dependent channel importance. The heterogeneous patterns suggest that informative channels are not aligned across tokens in the original channel basis. Results are experimented using LLaVA-NeXT-8B (layer-4 and head-4) on the TextVQA validation set.

2 Basis Mismatch in Visual Key Channels

Outliers in visual Key channels. Key channel pruning exploits a structural property of Key states (see "Related Works" in Appendix A): a few channels have consistently large magnitudes across tokens. Such outliers are common in LLMs (Liu et al., 2024f; Xu et al., 2024; Zhang et al., 2025c) and arise from the interaction between RoPE and learned Key projections (Barbero et al., 2024; Qiao et al., 2025). Since VLMs inherit the same backbone and RoPE, visual Key states show the same structure. In Figure 2(a), for example, channels near 40 and 100 are clear outliers.

Token-dependent channel importance. Such universally dominant channels, however, are few, so preserving them alone may be insufficient. To investigate this further, we compare the masks retaining 32 channels (25%) of ThinK (head-wise; Xu et al., 2024) and SparK (token-wise; Liao et al., 2026) in Figure 2(b)-(c). Only a few channels, corresponding to the universal outliers, are kept for all tokens (solid black columns in Figure 2(c)), while most retained channels vary across tokens. RoPE amplifies this dependency, as channels near indices 20 and 80 oscillate in magnitude across token positions, so no fixed subset is uniformly informative. This pattern is not specific to the head shown. Across 4,416 (layer, head) pairs spanning two model families and four datasets, 91% of the channels SparK retains per token are token-specific, and 26% of ThinK's budget falls on channels that most tokens do not consider important (Appendix E.4).

Toward token-invariant basis. This token dependency poses a fundamental challenge for structured Key channel pruning, which shares a static channel mask across tokens. We interpret it as a *basis mismatch*. Each token admits a compact informative subspace, but these directions are not aligned across tokens in the original channel coordinates. This suggests a natural remedy. Instead of pruning channels in the original basis, we first rotate the Key space so that informative directions align across tokens, allowing a shared head-wise mask to preserve them even at aggressive compression. Moreover, the informative subspace depends not only on the token but also on the input. A basis fitted offline on calibration data, or a mask learned on one visual domain, transfers poorly to others (Sec. 4.2), which motivates constructing the basis per input at inference time.

3 Proposed Methods

3.1 Overview of RotateK

Our goal is to compress the Key states $K \in \mathbb{R}^{N \times d}$ of N visual tokens (observed at prefill) from d to k channels with minimal perturbation to the attention scores $q_t K^\top$ at decode step t, where $q_t \in \mathbb{R}^d$ is the new query. Following the remedy suggested in Sec. 2, RotateK proceeds in two steps: **(1) rotate** the channel space with an orthogonal matrix $R \in \mathbb{R}^{d \times d}$ so that informative directions concentrate in a few shared channels, and **(2) prune** the remaining channels in the rotated basis (Figure 3(a)). We first show that only step (2) introduces error, then describe how RotateK chooses R to keep this error small and finally how it constructs R efficiently at inference time (Sec. 3.2). Figure 4 illustrates the overall inference flow (algorithm in Appendix G).

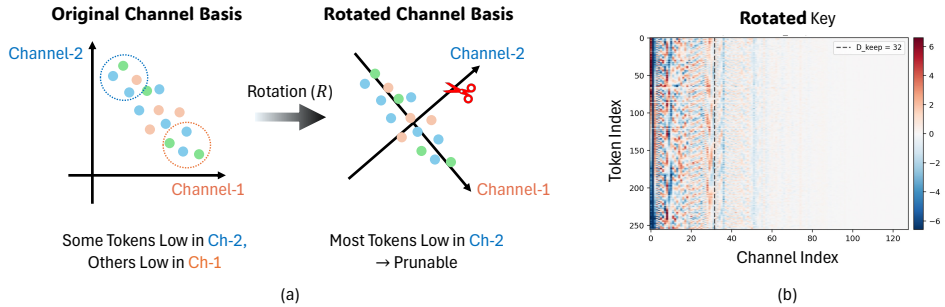


Figure 3: **High-level Idea of RotateK**. (a) In the original channel basis, different visual tokens exhibit different low-importance channels, making structured head-wise pruning ineffective. RotateK applies an orthogonal rotation R to align token-dependent importance into a shared channel basis, where most tokens consistently exhibit low activations on the same channels. (b) Visualization of the rotated Keys after applying RotateK, where informative channels become concentrated into a smaller subset of aligned channels across tokens, enabling effective structured key channel pruning.

Relation to rotation-based compression. Orthogonal rotations are common in compression but serve different goals. Quantization methods (Liu et al., 2024e) rotate to spread outliers across channels, and SliceGPT (Ashkboos et al., 2024a) rotates static weights before pruning them. RotateK instead rotates the visual Key cache, which changes with every input and is attended to by queries. This difference shapes each component below: the basis is built per input, weighted by queries, and computed on the inference path (Appendix A.3). Appendix B.2 compares these design choices with prior low-rank KV methods such as Palu (Chang et al., 2024) and ShadowKV (Sun et al., 2024a).

Step (1): Rotation is lossless. RotateK computes a rotation matrix R ($R^\top R{=}I_d$) *once* at the end of prefill, separately for each attention head, and reuses it for every subsequent decode step; the construction of R is described in the following paragraphs. If we rotate both keys and queries by R and retain all d channels, the attention scores are exactly preserved,

$$(q_t R)\,(KR)^\top \;=\; q_t\,RR^\top\,K^\top \;=\; q_t K^\top, \tag{1}$$

since $RR^\top{=}I_d$. Step (1) alone therefore introduces no approximation error and no compression. It merely re-expresses the channels in a different basis.

Step (2): Channel pruning introduces the approximation. Compression appears once we prune channels *in the rotated basis*. Let $R_k{\in}\mathbb{R}^{d\times k}$ collect the top-k columns of R, so that $R_k^\top R_k{=}I_k$. At prefill, RotateK caches the truncated rotated keys $\tilde{K}{=}KR_k \in \mathbb{R}^{N\times k}$ in place of the full $N{\times}d$ matrix K; at decode, the incoming query is rotated to $\tilde{q}_t{=}q_tR_k \in \mathbb{R}^k$ and computed with $\tilde{K}$. Notably, pruning the rotated channel is essentially structured (*i.e.*, head-wise) as a rotation matrix is shared across tokens. The resulting attention scores no longer match $q_tK^\top$:

$$\tilde{q}_t\tilde{K}^\top \;=\; (q_tR_k)\,(KR_k)^\top \;=\; q_t\,P_k\,K^\top, \qquad P_k := R_kR_k^\top, \tag{2}$$

because P_k is a rank-k projector and $P_k{\neq}I_d$ whenever $k{<}d$. The approximation error introduced by step (2) is therefore exactly:

$$q_tK^\top - \tilde{q}_t\tilde{K}^\top \;=\; q_t\,(I_d - P_k)\,K^\top, \tag{3}$$

the projection of $q_tK^\top$ onto the $d-k$ discarded channels of the rotated basis. The remainder of this section is concerned with choosing R, and hence R_k and P_k, so that this residual is small for the decode queries q_t that will arrive after R_k is fixed.

Query-weighted PCA. A natural choice for R is PCA on the Key states K, which yields the best rank-k approximation of K itself. The pruning error in (3), however, is incurred on the attention scores $q_tK^\top$, not on the Keys. We therefore weight each Key channel by the typical magnitude of queries along it before extracting principal directions, so that channels where both the Key activations and the query components are large are preserved preferentially.

Since the decode queries are unavailable at prefill, we use the W most recent prefill queries $Q_W{\in}\mathbb{R}^{W\times d}$ as a proxy (W=32, following Xu et al. (2024)), with per-channel ℓ_2 norms $(\boldsymbol{\sigma}_W)_j =$

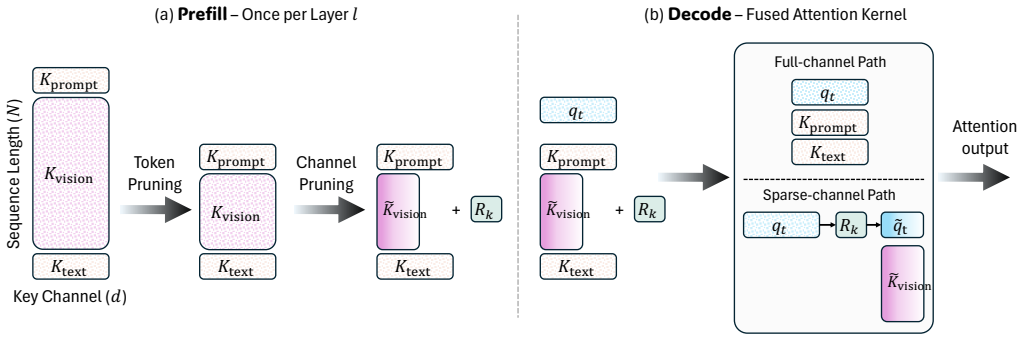


Figure 4: **Inference Flow of RotateK**. (a) Following visual token compression, RotateK applies head-wise rotation-based channel pruning to visual Key states during prefill, caching the compressed KV states together with the corresponding rotation matrices. (b) During decoding, queries are transformed using the cached rotations, while a fused attention kernel (details; Appendix D) combines full-channel attention for prompt/text tokens with sparse-channel attention for visual tokens.

$\|(Q_W)_{:,j}\|_2$. We apply PCA to the mean-centered, query-weighted Keys $K_q = (K - \mu)\,\mathrm{diag}(\boldsymbol{\sigma}_W)$, where $\mu \in \mathbb{R}^d$ is the per-head channel mean of K over the N visual tokens, whose covariance is:

$$C_q \;=\; K_q^\top K_q \;=\; \mathrm{diag}(\boldsymbol{\sigma}_W)\, C\, \mathrm{diag}(\boldsymbol{\sigma}_W), \tag{4}$$

with $C = (K-\mu)^\top (K-\mu)$ the centered Key covariance. The rotation $R = [r_1, \ldots, r_d]$ collects the eigenvectors of C_q in decreasing eigenvalue order, and R_k is its first k columns. Since μ is generally not in the column space of R_k, its discarded component adds a constant bias $q_t^\top (I_d - R_k R_k^\top)\,\mu$ which is restored at decode. Per-input construction accounts for most of RotateK's advantage over bases fixed in advance (Table 15); query weighting is an empirical, score-aware design on top of it, reducing the attention-score error on decode queries relative to standard PCA (Appendix B.1).

3.2 Hardware-efficient Implementation

Post-hoc reweighting. A naïve implementation of query-weighted PCA would rescale the Keys into K_q before computing its covariance, materializing another $N \times d$ tensor and adding $\mathcal{O}(Nd)$ memory traffic at every prefill. We avoid this by applying the weighting to the covariance C instead,

$$C_q = K_q^\top K_q = \mathrm{diag}(\boldsymbol{\sigma}_W)\, C\, \mathrm{diag}(\boldsymbol{\sigma}_W) = (\boldsymbol{\sigma}_W \boldsymbol{\sigma}_W^\top) \odot C, \tag{5}$$

where $\odot$ is the Hadamard product, so query weighting adds only $\mathcal{O}(d^2)$ per head, independent of the context length N (typically $d \ll N$).

Cholesky-based subspace iteration. A full eigendecomposition for PCA is cheap in FLOPs but slow in practice, as `torch.linalg.eigh` launches tens of fragmented CUDA kernels. Since only the top-k eigenspace is needed for R_k, RotateK instead runs T=5 steps of subspace iteration, each followed by Cholesky-QR orthonormalization (Appendix F), which matches the accuracy of full eigendecomposition (Appendix B.1). Each step issues four kernels with fixed shapes, so the whole loop is captured as a single CUDA graph, and basis construction costs about 0.60 ms per layer regardless of N (Appendix D.5).

KV cache budget. Since Values are kept at full width, retaining a fraction ρ_t of visual tokens and ρ_c of Key channels yields a KV budget of $\rho = \rho_t(1 + \rho_c)/2$; *e.g.*, $0.40\times$ tokens with $0.25\times$ channels matches $0.25\times$ token-only pruning. Decode compute is matched as well, since score and Value aggregation costs $\rho_t\rho_c Nd + \rho_t Nd = 2\rho Nd$, so the two configurations differ only in prefill, where joint pruning processes more visual tokens.

4 Experimental Results

Dataset and Implementation Details. Following the `lmms-eval` (Zhang et al., 2025a) protocol, we evaluate on VQA benchmarks (TextVQA, InfoVQA, DocVQA, ChartQA, VizWiz-VQA) and open-ended generation benchmarks (LLaVA-in-the-Wild, MM-Vet). Throughout the paper,

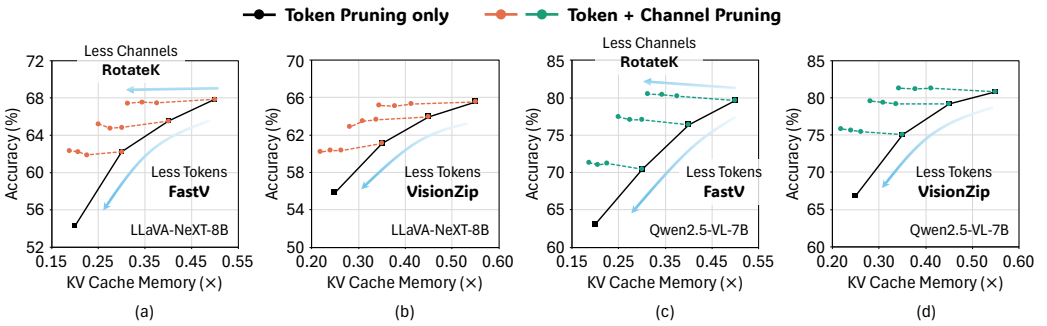


Figure 5: **Accuracy-KV Cache Trade-offs**. Accuracy-memory trade-offs on ChartQA using LLaVA-NeXT-8B (orange) and Qwen2.5-VL-7B (green) under varying token and channel keep ratios. Compared to token-only pruning (black), RotateK jointly prunes tokens and Key channels, consistently achieving higher accuracy under the same KV cache budget.

Table 1: **Comparison under Fixed KV Cache Budgets**. Accuracy comparison between token-only pruning and joint token-channel pruning under identical KV cache budgets. Across both LLaVA-NeXT-8B and Qwen2.5-VL-7B-Instruct, RotateK consistently improves over token-only pruning and prior Key channel pruning. More results are provided in Appendix E.1 (Table 17).

Method	Token	Channel	KV Cache	TVQA	IVQA	CQA	DVQA	VizWiz	Mean
LLAMA3-LLAVA-NEXT-8B									
Baseline	1.00×	1.00×	1.00×	65.40	32.27	69.08	72.44	57.50	59.34
$\text{FastV}_{\text{token only}}$	0.25×	1.00×	0.25×	62.69	25.91	59.64	56.39	58.56	52.64
FastV + ThinK	0.40×	0.25×	0.25×	54.07	25.51	60.56	51.72	57.70	49.91
FastV + RotateK	0.40×	0.25×	0.25×	62.67	28.37	65.25	62.46	58.34	55.42
$\text{VisionZip}_{\text{token only}}$	0.28×	1.00×	0.28×	63.33	27.70	57.32	64.08	58.97	54.28
VisionZip + ThinK	0.45×	0.25×	0.28×	54.22	27.81	58.52	53.77	57.81	50.43
VisionZip + RotateK	0.45×	0.25×	0.28×	62.62	28.97	62.88	66.99	58.94	56.08
QWEN2.5-VL-7B-INSTRUCT									
Baseline	1.00×	1.00×	1.00×	82.92	80.12	83.04	94.40	70.58	82.21
$\text{FastV}_{\text{token only}}$	0.25×	1.00×	0.25×	81.49	52.76	67.44	75.04	69.13	69.17
FastV + ThinK	0.40×	0.25×	0.25×	77.81	63.01	77.12	81.74	69.25	73.79
FastV + RotateK	0.40×	0.25×	0.25×	82.38	65.79	77.40	87.28	70.32	76.63
$\text{VisionZip}_{\text{token only}}$	0.28×	1.00×	0.28×	76.84	58.46	70.32	84.71	68.97	71.86
VisionZip + ThinK	0.45×	0.25×	0.28×	75.72	67.13	79.64	85.83	68.93	75.45
VisionZip + RotateK	0.45×	0.25×	0.28×	80.84	70.51	79.56	91.29	70.13	78.47

TextVQA, InfoVQA, DocVQA, and ChartQA are abbreviated as TVQA, IVQA, DVQA, and CQA, respectively. For latency evaluation, we compare Triton-based FlashAttention kernels against our Triton-based sparse attention kernel under identical settings. Unless otherwise specified, RotateK operates in an online calibration-free mode, recomputing the PCA-based rotation matrix during each prefill stage. All experiments are conducted on a single NVIDIA A100-80GB GPU.

Models and baselines. We evaluate on LLaVA-NeXT-8B (`llama3-llava-next-8b-hf`) and Qwen2.5-VL-7B-Instruct as main backbones, and additionally on Qwen3-VL-8B-Instruct, InternVL3.5-8B, Qwen2.5-VL-32B-Instruct, and InternVL3.5-14B. RotateK is integrated into four token pruning methods: VisionZip (Yang et al., 2025), FastV (Chen et al., 2024a), PyramidDrop (Xing et al., 2024), and SparseVLM (Zhang et al., 2024b). We compare against training-free Key channel pruning (ThinK (Xu et al., 2024) and SparK (Liao et al., 2026)) under identical KV budgets and against training-based alternatives (LeanK (Zhang et al., 2025c) and a lightly trained rotation following MatryoshkaKV (Lin et al., 2025)). Experiment details are in Appendix C.

4.1 COMPARISON WITH TOKEN PRUNING

Accuracy-KV cache trade-offs. Figure 5 compares token-only and joint token-channel pruning on ChartQA across compression ratios, where horizontal points labeled "Less Channels" apply additional channel pruning, with Key channel keep ratios of 0.500, 0.375, and 0.250, on top of fixed

Table 2: **Generalization across Answer Format, Token Pruners, and Model Scale.** (a) Open-ended evaluation with descriptive answers on LLaVA-in-the-Wild (LLaVA-W) and MM-Vet. (b,c) Mean accuracy (%) over five VQA benchmarks (TVQA, IVQA, CQA, DVQA, VizWiz) with (b) two additional token pruners and (c) larger VLMs. Subscripts denote token keep ratios (layer-averaged for PyramidDrop and SparseVLM); joint pruning additionally keeps 0.25× Key channels, so each group is compared at a matched KV budget. Full results are in Appendix E.1 (Table 18).

Method	LLaVA-W	MMVet
LLaVA-NeXT-8B		
Baseline	84.10	27.14
$\text{FastV}_{0.19}$	84.60	24.50
$\text{FastV}_{0.30}$ + ThinK	80.40	25.05
$\text{FastV}_{0.30}$ + RotateK	85.50	26.97
$\text{VisionZip}_{0.22}$	81.10	24.59
$\text{VisionZip}_{0.35}$ + ThinK	75.30	23.72
$\text{VisionZip}_{0.35}$ + RotateK	84.70	26.97
Qwen2.5-VL-7B		
Baseline	109.30	45.55
$\text{FastV}_{0.19}$	100.00	32.61
$\text{FastV}_{0.30}$ + ThinK	99.80	32.61
$\text{FastV}_{0.30}$ + RotateK	106.90	41.24
$\text{VisionZip}_{0.22}$	104.00	34.77
$\text{VisionZip}_{0.35}$ + ThinK	99.10	35.00
$\text{VisionZip}_{0.35}$ + RotateK	106.20	44.04

(a) Open-ended generation

Method	Mean
LLaVA-NeXT-8B	
Baseline	59.34
$\text{PyramidDrop}_{0.25}$	50.86
$\text{PyramidDrop}_{0.40}$ + ThinK	50.85
$\text{PyramidDrop}_{0.40}$ + RotateK	55.90
$\text{SparseVLM}_{0.30}$	54.04
$\text{SparseVLM}_{0.48}$ + ThinK	51.91
$\text{SparseVLM}_{0.48}$ + RotateK	57.43
Qwen2.5-VL-7B	
Baseline	82.21
$\text{PyramidDrop}_{0.25}$	60.67
$\text{PyramidDrop}_{0.40}$ + ThinK	70.35
$\text{PyramidDrop}_{0.40}$ + RotateK	73.13
$\text{SparseVLM}_{0.30}$	64.99
$\text{SparseVLM}_{0.48}$ + ThinK	71.65
$\text{SparseVLM}_{0.48}$ + RotateK	74.73

(b) Additional token pruners

Method	Mean
Qwen2.5-VL-32B	
Baseline	76.86
$\text{FastV}_{0.25}$	57.13
$\text{FastV}_{0.40}$ + ThinK	64.45
$\text{FastV}_{0.40}$ + RotateK	67.02
$\text{VisionZip}_{0.28}$	63.68
$\text{VisionZip}_{0.45}$ + ThinK	69.59
$\text{VisionZip}_{0.45}$ + RotateK	71.98
InternVL3.5-14B	
Baseline	76.76
$\text{FastV}_{0.25}$	63.14
$\text{FastV}_{0.40}$ + ThinK	67.07
$\text{FastV}_{0.40}$ + RotateK	69.19
$\text{VisionZip}_{0.28}$	54.48
$\text{VisionZip}_{0.45}$ + ThinK	62.18
$\text{VisionZip}_{0.45}$ + RotateK	64.45

(c) Larger models

token keep ratios. While token-only pruning degrades rapidly at aggressive compression, RotateK consistently achieves higher accuracy under the same KV cache budget, remaining robust even at 75% channel sparsity.

Comparison under fixed KV cache budgets. Table 1 compares RotateK across five benchmarks at matched KV budgets. The degradation caused by token-only pruning varies widely across datasets; InfoVQA on Qwen2.5-VL-7B is hit hardest, and RotateK improves it by more than 10 points. RotateK also outperforms ThinK in every configuration mean (Tables 17, 18) with especially large gains on DocVQA for LLaVA-NeXT-8B. Its benefits are not limited to settings where token-only pruning collapses: on relatively robust benchmarks such as TextVQA and VizWiz, it remains comparable or better at the same budget.

Comparison on open-ended benchmarks. Since channel pruning acts on the Keys read at every decode step, we also evaluate beyond short-answer VQA, where a single word or phrase is generated. Table 2(a) reports results on LLaVA-in-the-Wild (Liu et al., 2024a) and MM-Vet (Yu et al., 2023), which require descriptive, sentence-level answers scored by a GPT judge. Token pruning with RotateK outperforms both token-only pruning and token pruning with ThinK in all eight settings, with the largest gains on Qwen2.5-VL-7B MM-Vet (up to +9.3 points over token-only pruning).

Generality across settings. The gains hold across token pruners, model families, scale, and long visual inputs. RotateK improves the five-benchmark mean over token-only pruning in all configurations of Tables 2(b,c), by up to +12.5 points (PyramidDrop, Qwen2.5-VL-7B), and outperforms ThinK in every configuration mean. On the four additional backbones, RotateK exceeds ThinK in all benchmark cells (Table 2(c) and Table 18). On long videos (MLVU-needle; Zhou et al., 2025, 64 frames) and high-resolution images (V*Bench; Wu & Xie, 2024, ∼5K visual tokens), joint pruning stays on par with token-only pruning at the default budget and recovers +7.3 to +7.8 points under tighter KV budgets (Appendix E.2).

4.2 Comparison with Channel Pruning

Structured vs. unstructured channel pruning. Since RotateK concentrates token-dependent channel importance into a smaller subset of channels, we examine how effectively it closes the gap between head-wise (structured; ThinK) and token-wise (unstructured; SparK) channel pruning. Figure 6 sweeps channel sparsity on TextVQA across two VLMs and two token pruners. At 50% sparsity, all methods perform similarly. As sparsity increases, ThinK degrades sharply, losing up to

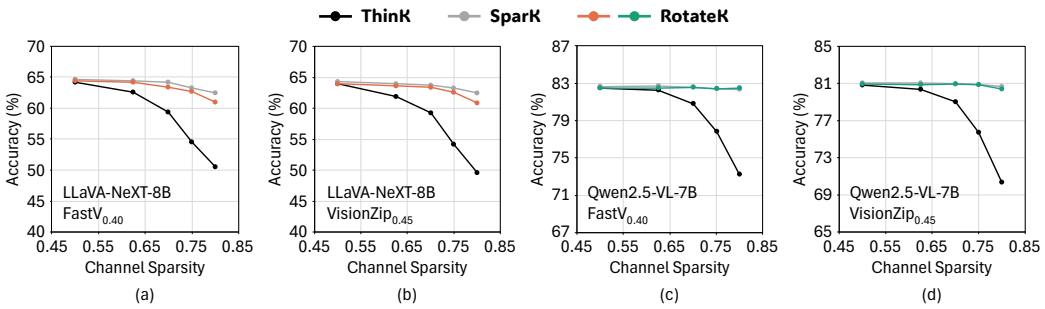


Figure 6: **Channel Pruning Robustness.** Accuracy on TextVQA under increasing channel sparsity for ThinK, SparK, and RotateK integrated with FastV and VisionZip on LLaVA-NeXT-8B and Qwen2.5-VL-7B. While ThinK rapidly degrades at high sparsity, RotateK maintains strong performance even at aggressive channel pruning levels, demonstrating improved robustness of structured Key channel pruning.

9.7 points at 75% sparsity, whereas RotateK stays within 0.7 points of SparK across all four settings, despite using a single head-wise mask. SparK is ahead by about 1.5 points only at 80% sparsity on LLaVA-NeXT-8B, likely due to its mean-based interpolation for pruned channels. Since ThinK and RotateK both apply a single head-wise mask and differ mainly in the basis in which it is applied, this result directly supports basis mismatch as the main source of head-wise degradation. We report SparK in this sweep rather than in the fixed-budget comparisons of Table 1, since its per-token masks require additional memory that the KV budget does not account for (Figure 7(e)).

Training-free vs. training-based. An alternative to per-input rotation is to fix the Key basis or channel mask offline, with the backbone frozen: for example, a static PCA rotation calibrated on held-out data, a rotation trained by distillation following MatryoshkaKV (Lin et al., 2025), or a channel mask learned by LeanK (Zhang et al., 2025c). None matches our per-input approach at aggressive compression. In particular, at 12.5% Key channels, RotateK outperforms the static and trained rotations by up to +17.2 and +8.6 points, and at 25% it exceeds LeanK's best out-of-domain mask on all three benchmarks, without any training (Appendix E.5).

4.3 Computational Efficiency

While prior Key channel pruning methods, such as ThinK and SparK, analyze accuracy and memory overhead, their latency behavior remains underexplored. We therefore provide a detailed analysis of the latency and memory overhead of the pruning methods.

Prefill latency. Basis construction costs a near-constant $\approx$0.60 ms per layer, dominated by the subspace iteration (0.52 ms), whose cost is independent of the number of visual tokens N; covariance, rotation, and cache transform are linear in N and together take under 0.1 ms at N=4K. As prefill time grows with N, this fixed cost is increasingly amortized, falling from 10.6% to 5.3% and 2.9% of prefill at 2K, 4K, and 8K visual tokens (Appendix D.5). Also, at a matched KV budget, joint pruning additionally processes 1.6$\times$ the visual tokens; measured end-to-end this costs +23 to +63 ms per request at $B\geq$8 (Appendix D.5).

Decode latency. Unlike prefill, decode latency differs clearly across methods (Figure 7(b,d)). ThinK and SparK grow progressively slower with input length and batch size, reaching up to 1.8$\times$ the full-channel latency. This is mainly because they reconstruct dense full-channel Keys at every step, concatenate them with unpruned Keys (*e.g.*, recent tokens), and run `FlashAttention2` on the recovered tensor, adding memory IO that outweighs the savings.

RotateK instead attends directly over the sparse-channel Keys; since its sparsity is head-wise, sparse- and full-channel Keys share one fused kernel, reducing decode latency to 0.62$\times$ of the full-channel baseline at batch size 8 and 0.61$\times$ at 128K input tokens. Figure 7(h) confirms that recovery and concatenation account for the extra latency of ThinK and SparK.

These slowdowns have different causes. For ThinK, reconstruction is an implementation choice, since head-wise pruning can run on a kernel like ours at a cost similar to RotateK, as LeanK does with its own (Zhang et al., 2025c). In contrast, for SparK, the overhead is algorithmic. Per-token

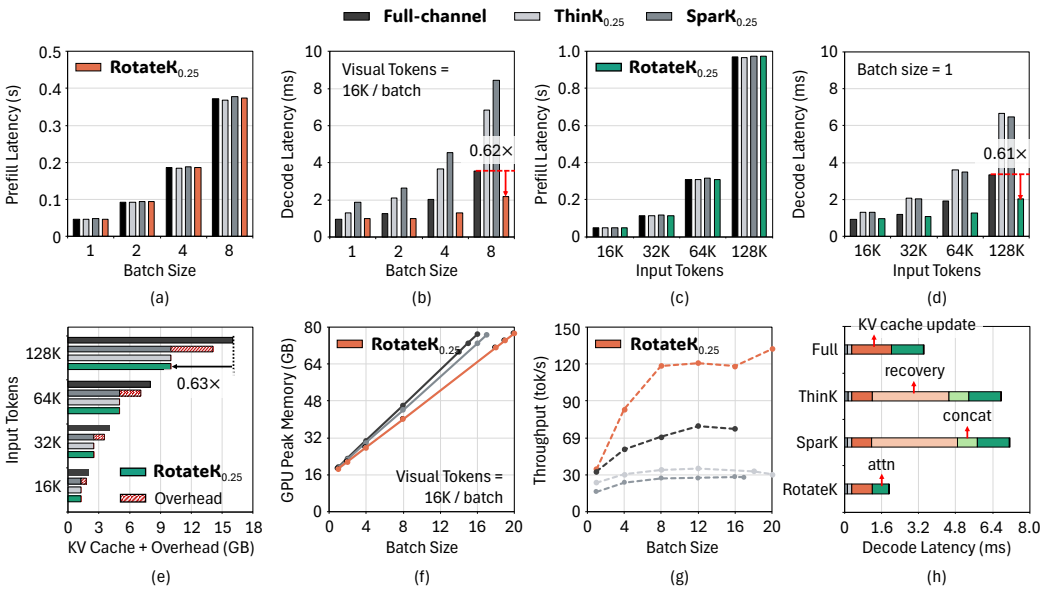


Figure 7: **Latency and Memory Analysis** of RotateK, ThinK, and SparK. (a,c) RotateK adds negligible prefill overhead across batch sizes and sequence lengths. (b,d) ThinK and SparK reconstruct full-channel Keys before attention, whereas RotateK attends directly over sparse-channel Keys, substantially reducing decode latency. (e) RotateK incurs far less memory overhead, lowering effective KV cache usage. (f,g) The smaller footprint enables larger batches and higher throughput. (h) Decode latency breakdown: full-channel Key recovery dominates ThinK and SparK.

channel sets preclude a fixed-width, paged layout, require an $\mathcal{O}(Ld)$ mask (Figure 7(e)), and cause irregular memory access under any kernel. RotateK thus claims no decode-speed advantage over optimized head-wise pruning. It matches their structured cost, plus basis construction at prefill, while approaching token-wise accuracy.

Memory overhead. Figure 7(e,f) compares KV cache size and GPU peak memory. SparK's token-wise pruning masks introduce substantial memory overhead, largely offsetting the benefit of channel pruning. In contrast, ThinK and RotateK use head-wise sparsity with negligible overhead, allowing larger batch sizes. Larger batches, however, do not necessarily yield higher throughput: as shown in Figure 7(g), ThinK, with its reference implementation, can still fall below the full-channel baseline due to its slower decoding, whereas RotateK consistently achieves higher throughput.

Deployment in a paged serving engine. Because RotateK's sparsity is head-wise, the compressed cache has a fixed per-token width that fits paged KV management without changes to the block manager or scheduler, which token-wise masks such as SparK cannot. We implement a vLLM (V1, PagedAttention) backend that stores 32 Key and 128 Value channels per token in region-split pages and reuses the unequal-dimension decode path built for multi-head latent attention (MLA). Relative to the full-channel baseline without token pruning, this yields 1.60× KV-cache capacity, a 1.56× faster decode-attention kernel, and up to 1.29× whole-step decode speedup, growing with context length and batch size (Appendix D.4).

5 Conclusion

We presented RotateK, a structured Key channel pruning framework for efficient VLM inference. We showed that the trade-off between head-wise and token-wise Key channel pruning stems from a basis mismatch that fixed bases, heuristic or learned, cannot fully resolve at aggressive compression, and addressed it with a training-free, query-weighted basis constructed per input. Across four VLM families up to 32B parameters and four token pruning methods, joint token–channel pruning with RotateK consistently improves over token-only pruning at matched KV budgets. Its structured sparsity translates into real efficiency, from a fused decode kernel to 1.60× KV capacity in a paged serving engine, and composes with KV quantization (Appendix E.3).

AI USE STATEMENT

In this work, we used generative AI tools to assist in writing and debugging experimental code, and to help design, run, and analyze some supporting experiments (*e.g.*, the Key-basis analysis in Appendix B.2). We have not used generative AI tools to develop theoretical models, formulate mathematical claims or proofs, or propose the research hypotheses, and the remaining required-disclosure tasks (generating synthetic data, translation, dataset cleaning, and qualitative data analysis) are not applicable to this work. Additionally, we used generative AI tools to draft and edit parts of the manuscript and to produce plotting code for figures. We have reviewed all AI-assisted work: all AI-assisted text was reviewed and revised by the authors, and all AI-assisted code and experiments were verified by the authors, including by reproducing the reported numbers. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

The method is fully specified in Sec. 3, with pseudocode in Appendix G and the subspace iteration in Appendix F. The decode kernel design, latency protocol, and vLLM integration are described in Appendices D.1-D.5. Models, token pruning settings, and baseline configurations are given in Sec. 4 and Appendix C, including the LeanK porting details in Appendix E.5. All accuracy results use the public lmms-eval framework with the splits stated in Appendix C.1. The code is submitted as supplementary material, and we will publicly release it upon publication.

ETHICS STATEMENT

This work improves the efficiency of Vision-Language Model (VLM) inference by reducing KV cache memory and decoding latency through Key channel pruning. More efficient VLM inference may lower computational and energy costs, enabling broader accessibility of multimodal AI systems on resource-constrained hardware and reducing the environmental footprint of large-scale deployment. At the same time, as with other advances in efficient AI inference, improved efficiency may facilitate wider deployment of VLMs, including applications that could generate misleading, biased, or harmful content. In addition, compression methods may introduce uneven performance degradation across tasks or domains, potentially affecting reliability in safety-critical settings. Our work focuses solely on inference efficiency and does not introduce new model capabilities beyond the underlying VLMs.

REFERENCES

Saeed Ranjbar Alvar, Gursimran Singh, Mohammad Akbari, and Yong Zhang. Divprune: Diversity-based visual token pruning for large multimodal models. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pp. 9392–9401, 2025.

Saleh Ashkboos, Maximilian L Croci, Marcelo Gennari Do Nascimento, Torsten Hoefler, and James Hensman. Slicegpt: Compress large language models by deleting rows and columns. In *The Twelfth International Conference on Learning Representations*, 2024a.

Saleh Ashkboos, Amirkeivan Mohtashami, Maximilian L Croci, Bo Li, Pashmina Cameron, Martin Jaggi, Dan Alistarh, Torsten Hoefler, and James Hensman. Quarot: Outlier-free 4-bit inference in rotated llms. *Advances in Neural Information Processing Systems*, 37:100213–100240, 2024b.

Shuai Bai, Yuxuan Cai, Ruizhe Chen, Keqin Chen, Xionghui Chen, Zesen Cheng, Lianghao Deng, Wei Ding, Chang Gao, Chunjiang Ge, et al. Qwen3-vl technical report. *arXiv preprint arXiv:2511.21631*, 2025.

Federico Barbero, Alex Vitvitskyi, Christos Perivolaropoulos, Razvan Pascanu, and Petar Veličković. Round and round we go! what makes rotary positional encodings useful? *arXiv preprint arXiv:2410.06205*, 2024.

Chi-Chih Chang, Wei-Cheng Lin, Chien-Yu Lin, Chong-Yan Chen, Yu-Fang Hu, Pei-Shuo Wang, Ning-Chi Huang, Luis Ceze, Mohamed S Abdelfattah, and Kai-Chiang Wu. Palu: Compressing kv-cache with low-rank projection. *arXiv preprint arXiv:2407.21118*, 2024.

Liang Chen, Haozhe Zhao, Tianyu Liu, Shuai Bai, Junyang Lin, Chang Zhou, and Baobao Chang. An image is worth 1/2 tokens after layer 2: Plug-and-play inference acceleration for large vision-language models. In *European Conference on Computer Vision*, pp. 19–35. Springer, 2024a.

Yukang Chen, Fuzhao Xue, Dacheng Li, Qinghao Hu, Ligeng Zhu, Xiuyu Li, Yunhao Fang, Haotian Tang, Shang Yang, Zhijian Liu, et al. Longvila: Scaling long-context visual language models for long videos. In *International Conference on Learning Representations*, volume 2025, pp. 18227–18246, 2025.

Zhe Chen, Jiannan Wu, Wenhai Wang, Weijie Su, Guo Chen, Sen Xing, Muyan Zhong, Qinglong Zhang, Xizhou Zhu, Lewei Lu, et al. Internvl: Scaling up vision foundation models and aligning for generic visual-linguistic tasks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 24185–24198, 2024b.

Samir Khaki, Junxian Guo, Jiaming Tang, Shang Yang, Yukang Chen, Konstantinos N Plataniotis, Yao Lu, Song Han, and Zhijian Liu. Sparsevila: Decoupling visual sparsity for efficient vlm inference. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 23784–23794, 2025.

Huanxuan Liao, Yixing Xu, Shizhu He, Guanchen Li, Xuanwu Yin, Dong Li, Emad Barsoum, Jun Zhao, and Kang Liu. Spark: Query-aware unstructured sparsity with recoverable kv cache channel pruning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 31961–31969, 2026.

Bokai Lin, Zihao Zeng, Zipeng Xiao, Siqi Kou, Tianqi Hou, Xiaofeng Gao, Hao Zhang, and Zhijie Deng. Matryoshkakv: Adaptive kv compression via trainable orthogonal projection. In *International Conference on Learning Representations*, volume 2025, pp. 86669–86690, 2025.

Haokun Lin, Haobo Xu, Yichen Wu, Jingzhi Cui, Yingtao Zhang, Linzhan Mou, Linqi Song, Zhenan Sun, and Ying Wei. Duquant: Distributing outliers via dual transformation makes stronger quantized llms. *Advances in Neural Information Processing Systems*, 37:87766–87800, 2024.

Haotian Liu, Chunyuan Li, Yuheng Li, and Yong Jae Lee. Improved baselines with visual instruction tuning. In *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 26286–26296. IEEE, 2024a.

Haotian Liu, Chunyuan Li, Yuheng Li, Bo Li, Yuanhan Zhang, Sheng Shen, and Yong Jae Lee. Llavanext: Improved reasoning, ocr, and world knowledge, 2024b.

Ting Liu, Liangtao Shi, Richang Hong, Yue Hu, Quanjun Yin, and Linfeng Zhang. Multi-stage vision token dropping: Towards efficient multimodal large language model. *arXiv preprint arXiv:2411.10803*, 2024c.

Yuliang Liu, Zhang Li, Mingxin Huang, Biao Yang, Wenwen Yu, Chunyuan Li, Xu-Cheng Yin, Cheng-Lin Liu, Lianwen Jin, and Xiang Bai. Ocrbench: on the hidden mystery of ocr in large multimodal models. *Science China Information Sciences*, 67(12):220102, 2024d.

Zechun Liu, Changsheng Zhao, Igor Fedorov, Bilge Soran, Dhruv Choudhary, Raghuraman Krishnamoorthi, Vikas Chandra, Yuandong Tian, and Tijmen Blankevoort. Spinquant: Llm quantization with learned rotations. *arXiv preprint arXiv:2405.16406*, 2024e.

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*, 2024f.

Ahmed Masry, Xuan Long Do, Jia Qing Tan, Shafiq Joty, and Enamul Hoque. Chartqa: A benchmark for question answering about charts with visual and logical reasoning. In *Findings of the association for computational linguistics: ACL 2022*, pp. 2263–2279, 2022.

Ye Qiao, Haocheng Xu, Xiaofan Zhang, and Sitao Huang. Rethinking rope scaling in quantized llm: Theory, outlier, and channel-band analysis with weight rescaling. *arXiv preprint arXiv:2510.00028*, 2025.

Utkarsh Saxena, Gobinda Saha, Sakshi Choudhary, and Kaushik Roy. Eigen attention: Attention in low-rank space for kv cache compression. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 15332–15344, 2024.

Yuzhang Shang, Mu Cai, Bingxin Xu, Yong Jae Lee, and Yan Yan. Llava-prumerge: Adaptive token reduction for efficient large multimodal models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 22857–22867, 2025.

Xiaoqian Shen, Yunyang Xiong, Changsheng Zhao, Lemeng Wu, Jun Chen, Chenchen Zhu, Zechun Liu, Fanyi Xiao, Balakrishnan Varadarajan, Florian Bordes, et al. Longvu: Spatiotemporal adaptive compression for long video-language understanding. *arXiv preprint arXiv:2410.17434*, 2024.

Prajwal Singhania, Siddharth Singh, Shwai He, Soheil Feizi, and Abhinav Bhatele. Loki: Low-rank keys for efficient sparse attention. *Advances in Neural Information Processing Systems*, 37: 16692–16723, 2024.

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

Yuxuan Sun, Ruikang Liu, Haoli Bai, Han Bao, Kang Zhao, Yuening Li, Jiaxin Hu, Xianzhi Yu, Lu Hou, Chun Yuan, et al. Flatquant: Flatness matters for llm quantization. *arXiv preprint arXiv:2410.09426*, 2024b.

Jintao Tong, Wenwei Jin, Pengda Qin, Anqi Li, Yixiong Zou, Yuhong Li, Yuhua Li, and Ruixuan Li. Flowcut: Rethinking redundancy via information flow for efficient vision-language models. *arXiv preprint arXiv:2505.19536*, 2025.

Dezhan Tu, Danylo Vashchilenko, Yuzhe Lu, and Panpan Xu. Vl-cache: Sparsity and modality-aware kv cache compression for vision-language model inference acceleration. *arXiv preprint arXiv:2410.23317*, 2024.

Zhongwei Wan, Ziang Wu, Che Liu, Jinfa Huang, Zhihong Zhu, Peng Jin, Longyue Wang, and Li Yuan. Look-m: Look-once optimization in kv cache for efficient multimodal long-context inference. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 4065–4078, 2024.

Weihan Wang, Qingsong Lv, Wenmeng Yu, Wenyi Hong, Ji Qi, Yan Wang, Junhui Ji, Zhuoyi Yang, Lei Zhao, Xixuan Song, et al. Cogvlm: Visual expert for pretrained language models. *Advances in Neural Information Processing Systems*, 37:121475–121499, 2024.

Penghao Wu and Saining Xie. V*: Guided visual search as a core mechanism in multimodal llms. In *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 13084–13094. IEEE, 2024.

Long Xing, Qidong Huang, Xiaoyi Dong, Jiajie Lu, Pan Zhang, Yuhang Zang, Yuhang Cao, Conghui He, Jiaqi Wang, Feng Wu, et al. Pyramiddrop: Accelerating your large vision-language models via pyramid visual redundancy reduction. *arXiv preprint arXiv:2410.17247*, 2024.

Yuhui Xu, Zhanming Jie, Hanze Dong, Lei Wang, Xudong Lu, Aojun Zhou, Amrita Saha, Caiming Xiong, and Doyen Sahoo. Think: Thinner key cache by query-driven pruning. *arXiv preprint arXiv:2407.21018*, 2024.

Senqiao Yang, Yukang Chen, Zhuotao Tian, Chengyao Wang, Jingyao Li, Bei Yu, and Jiaya Jia. Visionzip: Longer is better but not necessary in vision language models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 19792–19802, 2025.

Shuo Yang, Ying Sheng, Joseph E Gonzalez, Ion Stoica, and Lianmin Zheng. Post-training sparse attention with double sparsity. *arXiv preprint arXiv:2408.07092*, 2024.

Weihao Ye, Qiong Wu, Wenhao Lin, and Yiyi Zhou. Fit and prune: Fast and training-free visual token pruning for multi-modal large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 22128–22136, 2025.

Weihao Yu, Zhengyuan Yang, Linjie Li, Jianfeng Wang, Kevin Lin, Zicheng Liu, Xinchao Wang, and Lijuan Wang. Mm-vet: Evaluating large multimodal models for integrated capabilities. *arXiv preprint arXiv:2308.02490*, 2023.

Kaichen Zhang, Bo Li, Peiyuan Zhang, Fanyi Pu, Joshua Adrian Cahyono, Kairui Hu, Shuai Liu, Yuanhan Zhang, Jingkang Yang, Chunyuan Li, et al. Lmms-eval: Reality check on the evaluation of large multimodal models. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 881–916, 2025a.

Peiyuan Zhang, Kaichen Zhang, Bo Li, Guangtao Zeng, Jingkang Yang, Yuanhan Zhang, Ziyue Wang, Haoran Tan, Chunyuan Li, and Ziwei Liu. Long context transfer from language to vision. *arXiv preprint arXiv:2406.16852*, 2024a.

Qizhe Zhang, Aosong Cheng, Ming Lu, Renrui Zhang, Zhiyong Zhuo, Jiajun Cao, Shaobo Guo, Qi She, and Shanghang Zhang. Beyond text-visual attention: Exploiting visual cues for effective token pruning in vlms. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 20857–20867, 2025b.

Yike Zhang, Zhiyuan He, Huiqiang Jiang, Chengruidong Zhang, Yuqing Yang, Jianyong Wang, and Lili Qiu. Leank: Learnable k cache channel pruning for efficient decoding. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 31110–31125, 2025c.

Yuan Zhang, Chun-Kai Fan, Junpeng Ma, Wenzhao Zheng, Tao Huang, Kuan Cheng, Denis Gudovskiy, Tomoyuki Okuno, Yohei Nakata, Kurt Keutzer, et al. Sparsevlm: Visual token sparsification for efficient vision-language model inference. *arXiv preprint arXiv:2410.04417*, 2024b.

Junjie Zhou, Yan Shu, Bo Zhao, Boya Wu, Zhengyang Liang, Shitao Xiao, Minghao Qin, Xi Yang, Yongping Xiong, Bo Zhang, et al. Mlvu: Benchmarking multi-task long video understanding. In *2025 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 13691–13701. IEEE, 2025.

A Related Works

A.1 Visual Token Pruning

Token-axis compression of the visual KV cache falls into two regimes. Pre-LLM methods, exemplified by VisionZip (Yang et al., 2025) and LLaVA-PruMerge (Shang et al., 2025), discard tokens at the vision encoder using saliency signals such as [CLS] attention (Zhang et al., 2025b) or token-flow propagation (Tong et al., 2025); lacking access to the language query, they risk removing tokens critical to the downstream task. In-LLM methods defer pruning until after vision-text attention: FastV (Chen et al., 2024a) drops visual tokens with low attention after an early layer, with extensions that adapt pruning ratios across layers (Ye et al., 2025), progressively drop tokens at multiple stages (PyramidDrop; Xing et al., 2024), rank tokens by text-guided attention and recycle pruned ones (SparseVLM; Zhang et al., 2024b), or apply dual attention filtering (Liu et al., 2024c). However, visual token pruning still incurs substantial degradation on fine-grained tasks such as ChartQA, DocVQA, and InfoVQA, where informative content is spread across the scene (Khaki et al., 2025).

A.2 KV Channel Pruning

A complementary line of work compresses the KV cache along the channel axis, with most effort on Key channels (Xu et al., 2024; Zhang et al., 2025c; Liao et al., 2026; Yang et al., 2024). ThinK (Xu et al., 2024) performs training-free structured pruning via a query-driven saliency score, retaining a fixed channel subset per layer. LeanK (Zhang et al., 2025c) learns a model-wise static mask through two-stage calibration and supplies a custom GPU kernel that turns the sparsity into measurable decoding speedup. Both fall under a head-wise regime, pruned channels are fixed across all tokens within a head, yielding lightweight masks but unable to adapt to per-token variation. SparK (Liao et al., 2026), in contrast, prunes at the token-wise level with a prune-and-recover mechanism; this finer granularity recovers accuracy lost under coarse pruning, but the unstructured sparsity scales with sequence length and resists hardware speedup. Notably, all of these methods are developed and evaluated only in the text-only LLM setting; their behavior in the vision-language regime, where visual tokens dominate the cache, remains largely unexplored.

A.3 Rotation-based Compression

Since attention is invariant to a change of basis, orthogonal rotations can make a model more amenable to compression without altering its outputs. For quantization, QuaRot (Ashkboos et al., 2024b) and SpinQuant (Liu et al., 2024e) apply Hadamard or learned rotations to suppress activation outliers, and DuQuant (Lin et al., 2024) and FlatQuant (Sun et al., 2024b) extend this with grouped or block-diagonal rotations. SliceGPT (Ashkboos et al., 2024a) carries the same invariance into structural weight pruning.

For the KV cache, Palu (Chang et al., 2024), Eigen Attention (Saxena et al., 2024), and MatryoshkaKV (Lin et al., 2025) compress Keys via low-rank projections obtained offline from weights, calibration data, or training, and Loki (Singhania et al., 2024) uses a calibrated Key PCA to approximate attention scores for token selection. ShadowKV (Sun et al., 2024a) instead computes a prompt-dependent SVD of pre-RoPE Keys at prefill, keeps them in low-rank form while offloading Values to the CPU, and at decode reconstructs the Keys of chunks selected by sparse attention and re-applies RoPE. RotateK differs in three respects: its basis is weighted by query magnitudes to target attention scores rather than Key reconstruction; it operates on post-RoPE Keys, so the basis is absorbed into the query and attention is computed in the compressed space without reconstruction; and its structured form admits a fixed-width paged KV layout that composes with token pruning. Appendix B.2 measures two design choices, head sharing and RoPE placement, directly on visual Keys.

Relation to rotation-based weight pruning. SliceGPT also rotates and then prunes, but it acts on a static weight matrix, whereas RotateK acts on a dynamic Key cache consumed by queries inside attention. This difference invalidates three of SliceGPT's design choices, and each component of RotateK addresses one. (i) A single offline basis does not transfer across visual inputs, so RotateK builds the basis per prefill (Table 15: online vs. static PCA, up to +17.2 points at 12.5% channels). (ii) A per-prefill decomposition must sit on the inference path, so RotateK replaces full eigendecomposition with a graph-capturable subspace iteration (0.60 ms per layer). (iii) The error that matters

is on attention scores, not on the pruned object itself, so RotateK weights each Key channel by the magnitude of recent queries (attention-score error $0.264 \rightarrow 0.225$).

B Ablation of RotateK Design Choices

We first ablate RotateK's own choices of objective and eigensolver (B.1), and then relate its basis to other Key bases used in prior work (B.2).

B.1 Objective and Eigensolver

Table 3: **Ablation of RotateK Design Choices**. Three short-answer VQA benchmarks (accuracy, %) are evaluated. We isolate two axes: (i) **Q-aware** (query-weighted PCA, our default) vs. **Q-agnostic** (K-only PCA); and (ii) **Cholesky**-based truncated subspace iteration (our default) vs. full **eigh** (`torch.linalg.eigh`). Highlighted entries (red) mark non-trivial accuracy drops when query-awareness is removed. We omit the (eigh, Q-agnostic) corner since it is dominated by both alternatives along each axis.

RotateK variant	TextVQA	InfoVQA	ChartQA
LLaVA-NeXT + FastV			
Cholesky, Q-aware (ours)	63.38	27.26	56.00
eigh, Q-aware	64.52	27.52	56.20
Cholesky, Q-agnostic	61.70	27.78	57.20
LLaVA-NeXT + VisionZip			
Cholesky, Q-aware (ours)	65.50	28.81	58.20
eigh, Q-aware	66.54	28.91	57.60
Cholesky, Q-agnostic	63.46	29.61	56.40
Qwen2.5-VL + FastV			
Cholesky, Q-aware (ours)	83.50	58.05	63.80
eigh, Q-aware	83.92	58.21	63.60
Cholesky, Q-agnostic	84.50	57.00	63.40
Qwen2.5-VL + VisionZip			
Cholesky, Q-aware (ours)	80.06	64.46	68.60
eigh, Q-aware	80.50	64.95	68.40
Cholesky, Q-agnostic	80.66	65.00	67.20

We ablate two RotateK design choices: (i) Cholesky-based truncated subspace iteration vs. full eigendecomposition (`torch.linalg.eigh`), and (ii) query-weighted PCA vs. K-only PCA. Table 3 reports validation-lite accuracy across two backbones (LLaVA-NeXT, Qwen2.5-VL) crossed with two token-pruning counterparts (FastV, VisionZip).

Setup. We retain 25% of Key channels (75% pruned). FastV uses $K{=}2$ with token keep ratio 0.30; VisionZip uses dominant ratio 0.20 and contextual ratio 0.05 ($0.25\times$ tokens). Backbones are `llama3-llava-next-8b-hf` and Qwen2.5-VL-7B-Instruct. For tractability, we use lmms-eval lite validation splits ($\sim$100-500 examples each).

Cholesky-based subspace iteration matches full eigh in accuracy. Across all twelve (backbone, token-pruner, dataset) cells, the gap between our Cholesky variant and full `eigh` stays within $\sim$1 percentage point, with neither solver consistently dominating (largest gap $+1.14$ pts on LLaVA-NeXT + FastV/TextVQA in favour of `eigh`; the Cholesky variant wins 3 of 4 ChartQA cells, and all gaps stay within 1.2 points). The Cholesky-based solver therefore contributes purely on the *latency* axis, fewer kernel launches and CUDA-graph capture, while preserving the rotation basis quality of the full eigendecomposition. This accuracy parity is what licenses us to use the faster Cholesky path as the default throughout the main results.

Query weighting is a low-risk default for downstream accuracy. In Table 3, query weighting improves accuracy by up to 2.04 points, most notably on LLaVA-NeXT-8B, while the K-only variant leads by at most 1.20 points where it is ahead. When the recent-query distribution Q_W is roughly isotropic across channels, $\mathrm{diag}(\boldsymbol{\sigma}_W) \approx \alpha I$ and the query-weighted covariance collapses to a uniformly scaled K-only covariance with the same top-k eigenbasis, which explains the cases where the two variants are close. Query weighting is therefore the safer choice: it costs little where queries are isotropic and helps where they are not.

Query weighting improves attention-score fidelity. Downstream accuracy reflects many factors beyond the Key basis, so we also measure the quantity the design targets directly (Table 4): the rel-

Table 4: **Reconstruction Error of Key Bases.** (relative Frobenius, lower is better), averaged over the 12 (backbone, token pruner, dataset) cells of Table 3. "Overall" averages over every (sample, layer, KV head) unit across both backbones, so LLaVA-NeXT-8B, with more layers and KV heads, carries more weight. "Cells won" counts the settings in which query-weighted PCA achieves lower error than query-agnostic PCA.

		Query-agnostic (K-only PCA)	Query-weighted (ours)	Full query covariance	Cells won by ours
Attention score	LLaVA-NeXT-8B	0.290	**0.236**	0.330	6/6
	Qwen2.5-VL-7B	0.205	**0.199**	0.281	4/6
	Overall	0.264	**0.225**	0.315	10/12
Key reconstruction	Overall	**0.418**	0.550	0.763	0/12

ative Frobenius error of attention scores, alongside that of Key reconstruction, computed on decode queries at 32 of 128 channels, per (sample, layer, KV head) unit (12,800 units per LLaVA-NeXT cell and 5,600 per Qwen2.5-VL cell). Query weighting lowers the attention-score error from 0.264 to 0.225 overall, and in every setting on LLaVA-NeXT-8B.

This gain in score fidelity comes at the cost of higher Key reconstruction error in every setting, which is expected rather than a counter-result. K-only PCA is the Frobenius-optimal rank-k approximation of K and must win on Key error, but Key error is not the objective. Weighting by the full query covariance is worse in all settings (Table 4): our per-channel norms are its diagonal restriction, and the full $d \times d$ covariance estimated from only $W{=}32$ queries overfits the observation window and fails to generalize to decode queries. Although the orthogonal projector built from the weighted covariance does not exactly minimize the expected score error, query weighting is an empirical, score-aware design that lowers this error at negligible cost (an $\mathcal{O}(d^2)$ Hadamard product on the already-computed covariance), and we therefore adopt it as the default.

B.2 Relation to Other Key Bases

Table 5: **Key-basis design axes.** Every method keeps k directions of a basis; they differ only in how the basis is obtained. "Absorbable" means the basis can be folded into the query so decode attends in the compressed space without reconstructing Keys.

Method	Built	Shared across	RoPE	Objective	Absorbable
ThinK / LeanK	per input / offline	head	post	channel subset	yes
Static PCA / Loki / Eigen Attention	offline	head	post	Key reconstruction	yes
MatryoshkaKV (Lin et al., 2025)	trained	head	–	distillation	yes
Palu (Chang et al., 2024)	offline (weights)	head group	pre	weight reconstruction	no
ShadowKV (Sun et al., 2024a)	per input	layer	pre	Key reconstruction	no
Grouped post-RoPE (future dir.)	per input	head group	post	–	yes
RotateK	per input	head	post	query-weighted	yes

Pruning Key channels in a rotated basis is a rank-k projection of the Keys, so Key channel and low-rank methods differ mainly in how they choose the basis (Table 5). RotateK builds a per-input, per-head, post-RoPE basis; its objective is ablated in B.1 and per-input construction in Appendix E.5 (Tables 15, 16). For context, we measure the two remaining axes on visual Keys: whether heads share a basis, and whether it is built before or after RoPE.

What we measure. Pruning a head's Keys to k directions keeps their projection onto those directions and discards the rest. We therefore report the *kept energy*: the fraction of the variance of a head's visual Keys that lies in the k kept directions (1.0 means nothing is discarded; the mean is kept exactly by the bias term of Sec. 3.1, so we measure variance around it). Throughout, $k{=}32$ of $d{=}128$ channels (25%), as in the main experiments.

Data. We run the prefill of LLaVA-NeXT-8B and Qwen2.5-VL-7B, without token pruning, on 48 images (16 each from TextVQA, ChartQA and InfoVQA), recompute each layer's Keys from its own projection and RoPE (M-RoPE for Qwen2.5-VL; these match the Keys written to the KV cache

exactly), and take 1,024 random visual tokens per image. Every number below is an average over all images, layers and KV heads. We use plain PCA for the data-derived basis.

Table 6: **Head sharing and RoPE placement.** Kept energy for a per-head vs. a per-layer (shared) basis, built before or after RoPE, at the same number of stored values per token.

	Per-head basis		Shared across heads	
	before RoPE	after RoPE	before RoPE	after RoPE
LLaVA-NeXT (1D RoPE)	0.89	0.76	0.99	0.97
Qwen2.5-VL (M-RoPE)	0.88	0.88	0.97	0.97

Head sharing and RoPE placement. We compare a separate basis per head (32 directions each) with one basis for all heads of a layer, obtained from the heads' Keys concatenated ($H_{kv} \times 32$ directions of $H_{kv} \times 128$, so the same number of values is stored per token). We build each basis both before and after RoPE (Table 6).

Our key observations are as follows: (1) RoPE placement matters little: it lowers the kept energy of a per-head basis under 1D RoPE (LLaVA-NeXT, 0.89 vs. 0.76) but not under M-RoPE (Qwen2.5-VL, 0.88 vs. 0.88). (2) Sharing matters more: a basis shared by all heads of a layer keeps 0.97 on both models, against 0.76 and 0.88 per head, with almost no gap between before and after RoPE.

Extending RotateK. This suggests grouping heads when building RotateK's per-input basis, similar to ShadowKV's layer-wise decomposition (Sun et al., 2024a). Built after RoPE, each head's block of the shared basis can still be absorbed into its query, so decode needs no Key reconstruction. The cost is storage: a shared basis holds $(H_{kv}k) \times (H_{kv}d)$ values per layer and input instead of $H_{kv} \times d \times k$, *i.e.* $H_{kv}\times$ more, which is comparable to the Key values themselves at about 10^3 visual tokens (LLaVA-NeXT, 25%: 262K vs. 240K per layer). We leave the group size and a decode kernel to future work.

C Additional Experimental Details

C.1 Dataset and Implementation Details.

Following the `lmms-eval` (Zhang et al., 2025a) protocol, we evaluate on both discriminative VQA benchmarks and open-ended generation benchmarks. The VQA benchmarks include TextVQA, InfoVQA, DocVQA, ChartQA, and VizWiz-VQA, covering OCR-intensive reasoning, document understanding, chart interpretation, and visually grounded question answering. Following prior work, we evaluate on the validation splits for all VQA benchmarks except ChartQA, where the test split is used. We further evaluate on LLaVA-in-the-Wild and MM-Vet to measure open-ended visual reasoning and generation quality. For benchmarks requiring generative evaluation, GPT-4o-mini is used as the judge model following the default `lmms-eval` setup.

Unless otherwise specified, RotateK operates in an online calibration-free setting, recomputing the PCA-based rotation matrix directly from visual activations during each prefill stage. The resulting compressed KV states and rotation matrices are cached and reused during decoding. We additionally evaluate an offline calibrated variant in the ablation studies, where the rotation basis is precomputed from calibration data and reused across samples.

For latency evaluation, all methods are implemented using Triton-based attention kernels for fair comparison. Specifically, full-channel baselines use the Triton implementation of FlashAttention-2, while RotateK uses our custom Triton sparse attention kernel operating directly on compressed Key states without reconstructing full-channel Keys. Unless otherwise stated, all experiments are conducted on a single NVIDIA A100-80GB GPU.

C.2 Baselines

We evaluate RotateK on LLaVA-NeXT-8B (`llama3-llava-next-8b-hf`) and Qwen2.5-VL-7B-Instruct as main backbones, and additionally on Qwen3-VL-8B-Instruct, InternVL3.5-8B, Qwen2.5-VL-32B-Instruct, and InternVL3.5-14B. To isolate the effect of Key channel pruning from token reduction, RotateK is integrated into four visual token pruning frameworks: VisionZip (Yang et al., 2025), FastV (Chen et al., 2024a), PyramidDrop (Xing et al., 2024), and SparseVLM (Zhang et al., 2024b).

For VisionZip, we follow the original implementation and fix the contextual ratio to 0.05, while sweeping the dominant token ratio over $\{0.40, 0.30\}$. For FastV, we use the standard pruning layer $K{=}2$ ($K{=}4$ for Qwen3-VL) with token keep ratios in $\{0.40, 0.30\}$. PyramidDrop and SparseVLM settings are given in Appendix E.1. On top of these, RotateK prunes 62.5% or 75% of visual Key channels. We compare against ThinK and SparK under identical KV cache budgets, together with token-only baselines, and against LeanK and fixed or trained rotations in Appendix E.5.

D RotateK Decode Kernel Implementation

At each decode step RotateK serves two attention paths against a single query: a full-rank path over the S_{full} non-vision tokens at head dimension d, and a rotated–truncated path over the S_v vision tokens at $d_{\text{keep}} < d$, projected by the per-head PCA basis $R \in \mathbb{R}^{B \times H_{kv} \times d \times d_{\text{keep}}}$. A naive implementation would issue four launches: a $Q \cdot R$ matmul, a sparse flash-decoding over the vision keys, a dense flash-decoding over the non-vision keys, and an online-softmax merge. We collapse these into two Triton kernels by inlining $Q \cdot R$ at the top of the sparse kernel and fusing the merge into the full-d kernel, matching the launch count of a dense decoder. The split-K factor over vision tokens is $N_s = \max(1, \min(\lceil S_v / \text{BLOCK_N} \rceil, 64))$.

D.1 Two-kernel decomposition

A natural single-kernel design fuses the full-d and sparse paths and gates the full-d loop on `if pid_s == 0`. Triton cannot statically eliminate the branch, so it allocates the union of both paths' register tiles for every split program, even though only the $\text{pid}_s = 0$ program executes the full-d loop. The $[\text{BLOCK_N}, d]$ full-d K, V tile inflates per-program register count from 111 to 141, dropping SM occupancy from 25% to 19% and breaking memory-latency hiding, the fused kernel ends up reading less memory than the dense baseline yet running slower. Splitting the work into a register-lean sparse kernel and a small full+merge kernel restores sparse-path occupancy.

D.2 Phase 1: sparse split-K with inlined rotation

Launched on a (B, H_q, N_s) grid; each program processes $\lceil S_v / N_s \rceil$ vision tokens for one query head and writes a partial (m, ℓ, acc) tuple to HBM scratch. Rather than launch a separate prelude that materialises the rotated query to HBM, we compute it (and an optional bias term) at the top of every sparse program in registers,

$$q_{\text{sparse}}[b, h, k] = \sum_{j=1}^{d} q_{\text{full}}[b, h, j]\, R[b,\, h_{kv},\, j,\, k], \tag{6}$$

$$\beta[b, h] = \sum_{j=1}^{d} q_{\text{full}}[b, h, j]\, \delta\mu[b,\, h_{kv},\, j], \tag{7}$$

trading N_s-fold reads of $R[b, h_{kv}]$ for one fewer kernel launch and one fewer HBM round-trip. The inner loop is a standard online-softmax update over $K_{\text{sparse}} \in \mathbb{R}^{\text{BLOCK_N} \times d_{\text{keep}}}$ and $V_{\text{sparse}} \in \mathbb{R}^{\text{BLOCK_N} \times d}$; V retains the full dimension since truncation is a basis change for $QK^\top$ only. We use $\text{BLOCK_N} = 64$, with `num_warps=1, num_stages=3` to triple-buffer K, V loads against the inner FMAs.

D.3 Phase 2: combined full-d and merge

Launched on a (B, H_q) grid, this kernel runs the full-d attention over the prompt and text tokens and then folds in the N_s sparse partials via the standard online-softmax merge of (m, ℓ, acc) tuples, writing the final output as $\text{acc}/(\ell + \varepsilon)$. Combining the two sub-tasks into a single launch matters because both are launch-overhead-bound: the non-vision segment is short and the merge processes only $N_s \leq 64$ partials per (b, h), so issuing them separately would double the launch overhead with little inner work to hide it. The kernel uses `num_warps=1, num_stages=2`; deeper pipelining buys little for the short non-vision loop.

After the two-kernel split, the inlined $Q \cdot R$ prelude, and the full+merge fusion, RotateK decode issues two Triton launches per layer per step, the same as a dense flash-decoding kernel. The four scratch buffers shared between phases are cached in a module-level dictionary keyed by $(B, H_q, d, N_s, \text{device}, \text{dtype})$, eliminating per-call allocator overhead at negligible memory cost.

Table 7: **In-engine Measurements on vLLM.** (a) KV-cache capacity at fixed memory and decode-attention kernel latency (context 16K, batch 32). (b) Per-step decode speedup vs. context length (Qwen2.5-VL-7B, batch 16). (c) Per-step decode speedup vs. batch size (context 16K).

Metric	Baseline	RotateK	Gain
KV capacity (tokens)	507,072	811,664	1.60×
Attention kernel (ms)	0.647	0.415	1.56×

(a) Capacity and kernel

Context	Speedup
2K	1.04×
4K	1.07×
8K	1.10×
16K	1.14×
30K	1.21×

(b) vs. context length

Batch	Qwen2.5-VL-7B (H_{kv}=4)	LLaVA-NeXT-8B (H_{kv}=8)
1	1.03×	1.05×
4	1.08×	1.12×
8	1.10×	1.19×
16	1.14×	1.29×
20	1.16×	1.25×

(c) vs. batch size

D.4 Deployment in a Paged Serving Engine

We implement RotateK as a custom attention backend for vLLM v0.8.5 (V1, PagedAttention) under full CUDA-graph capture. The adaptation is localized to one backend class and a page-size setting, with no change to the block manager, scheduler, model code, or decode kernel.

Four design choices make this possible. (i) *Single allocation of asymmetric width.* Instead of two full-width slots per token (128 Key + 128 Value channels), we allocate one slot of width 160 (32 Key + 128 Value). Since vLLM derives block capacity from the page size, proportionally more tokens fit per block at the same memory, and the block table and slot mapping are unchanged. (ii) *Region-split layout.* Within a page, Keys and Values occupy separate contiguous regions rather than being interleaved per token. Score computation reads every Key but no Value; under interleaving, a Key fetch pulls the adjacent Value into the same cache line, and compression yields no bandwidth saving. (iii) *Shared block index.* Both regions are indexed by the same physical block number, so allocating or freeing a block governs both regions and the paging logic is untouched. (iv) *Reuse of the MLA decode path.* The kernel must handle a query/Key dimension of 32 against a Value dimension of 128. vLLM and SGLang already support unequal dimensions for multi-head latent attention, and RotateK reuses this path. The rotation is applied inside the backend as an opaque custom op, preserving CUDA-graph capture.

Results. Table 7 summarizes the in-engine measurements. At fixed memory, the 160-wide slot raises KV-cache capacity from 507K to 812K tokens, a $1.60\times$ gain that matches the ideal $256/160$ ratio, and the decode-attention kernel runs $1.56\times$ faster (Table 7(a)). The per-step decode speedup grows with context length, from $1.04\times$ at 2K to $1.21\times$ at 30K (Table 7(b)), and with batch size, reaching up to $1.16\times$ on Qwen2.5-VL-7B and $1.29\times$ on LLaVA-NeXT-8B (Table 7(c)). The larger gain on LLaVA-NeXT-8B follows from its twice as many KV heads (H_{kv}=8 vs. 4), which make decoding more KV-bound.

The per-step speedup is smaller than the kernel speedup because it is diluted by the RotateK-independent MLP and by weights re-read at every decode step, and it therefore grows as decoding becomes more KV-bound with longer contexts and larger batches. On a single 80 GB GPU, the memory-bound ceiling is about $1.4\times$ (16 GB weights + 32 GB Keys + 32 GB Values vs. 16 + 8 + 32 GB).

Table 8: **Per-layer Cost of Basis Construction.** LLaVA-NeXT-8B with 4K visual tokens (d=128, k=32 retained channels, T=5 iterations) on an A100-80GB GPU, measured in steady state after CUDA-graph compilation. The subspace iteration dominates the cost and is independent of N; the stages linear in N together take under 0.1 ms, far below the full prefill of the same layer.

Stage	Complexity	Scales with N	ms/layer
fp32 cast	$\mathcal{O}(Nd)$	linear	0.04
Covariance $K^\top K$	$\mathcal{O}(Nd^2)$	linear	0.05
Symmetrize	$\mathcal{O}(d^2)$	no	0.01
Subspace iteration	$\mathcal{O}(Td^2k)$	no	0.52
Rotation KR_k	$\mathcal{O}(Ndk)$	linear	0.04
Mean correction	$\mathcal{O}(dk)$	no	0.00
Total construction			≈0.60
Full prefill (reference)	$\mathcal{O}(N^2d)$	quadratic	10.58

D.5 Prefill Overhead and End-to-End Latency

Table 9: **Construction Overhead Relative to Full Prefill.** Same setup as Table 8, varying the number of visual tokens N. Since basis construction is nearly constant in N while prefill grows with it, the relative overhead roughly halves each time N doubles.

Visual tokens	Full prefill (ms/layer)	RotateK overhead (ms/layer)	Relative
2,048	5.45	+0.58	+10.6%
4,096	10.58	+0.56	+5.3%
8,192	21.38	+0.61	+2.9%

Basis construction. Table 8 breaks down the per-layer cost of basis construction on LLaVA-NeXT-8B with 4K visual tokens (d=128, 32 retained channels), measured in steady state after compilation. Table 9 reports the overhead relative to full prefill. Every stage is at most linear in N or independent of it, whereas prefill self-attention is quadratic, so the relative overhead decays as $\mathcal{O}(d/N)$.

End-to-end latency at matched KV budgets. Basis construction is not the only prefill cost of joint pruning: at a matched KV budget it also processes more visual tokens than token-only pruning, because *retaining more visual evidence at the same cache size is exactly what the channel axis buys*. We therefore measure end-to-end latency on LLaVA-NeXT-8B using real TextVQA, ChartQA, and InfoVQA validation images at native resolution, generating 128 tokens on a single A100-80GB GPU (Table 10). Each request is replicated B times to form a serving batch, and we report medians over tasks and datasets. End-to-end time is a hook-free wall-clock measurement of `generate()`, while prefill and decode are measured with CUDA events in a separate instrumented run, with token pruning and basis construction counted in prefill.

Results at $B\geq8$. Relative to the dense model, joint pruning reduces end-to-end latency by 1.35-1.96× at B=8-32, with decode up to 1.93× faster. Against token-only pruning at the *same* KV budget (Token$_{0.28}$ vs. RotateK$_{0.45}$ and Token$_{0.25}$ vs. RotateK$_{0.40}$), the joint configuration retains 1.6× the visual tokens and pays for them in prefill: $+24$ **to** $+34$ **ms per request** at $B\geq8$, nearly flat in batch because the extra work scales with the batch it serves. Decode, by contrast, converges to token-only as the batch grows, reaching break-even at B=32 (0.99-1.00×). End-to-end, the joint configuration therefore costs $+23$ **to** $+63$ **ms per request** at $B\geq8$ — a one-time prefill charge whose share of the total shrinks both with batch and with generated length.

Where KV compression applies at all. We include B=1 to mark where the regime begins, not as an operating point. There, neither axis, token or channel pruning, buys anything: token-only pruning discards 68% of the visual tokens yet decodes at 0.99× dense, and peak memory is identical across every arm (18.2 GB), because a single request of ∼2K visual tokens puts the model under neither memory nor bandwidth pressure. The joint configuration is 0.93× dense there, and the gap lies entirely in decode (its prefill is in fact 66 ms faster than dense) as a fixed per-step cost of $+2.2$ ms ($+11\%$) in the split-channel attention path. That cost is independent of the token count: the iso-token control, which retains the same tokens, decodes within 0.2% of dense. KV compression along either axis becomes meaningful only once the cache is a material share of decode traffic, so the figures below are quoted for $B\geq8$.

Isolating RotateK's own cost. The iso-token control in Table 10 retains the same number of visual tokens as the joint configuration but spends 1.6× the KV budget to do so. Against it, RotateK's own prefill cost is $+2.5$ **to** $+3.6$ **ms per request** at $B\geq8$: basis construction accounts for roughly a tenth of the prefill gap and the retained tokens for the remainder. Its decode is *faster* than the control from B=16 onward (1.14× and 1.24× at B=16, 32 for VisionZip; 1.11× and 1.21× for FastV), and grows with batch because the Key-channel saving pays off once the KV cache occupies a larger share of decode memory traffic. This end-to-end trend is the diluted counterpart of the in-engine per-step speedups in Table 7(b,c); here the saving is measured in an eager end-to-end run.

Spending the channel axis. These results reflect how the memory freed by channel pruning is spent. Token-only pruning has a single knob — how many visual tokens to keep — so once the KV budget is fixed, nothing further can be traded. The channel axis adds a second. Because Values are kept at full width, pruning Keys to a quarter of their channels shrinks the cache to 0.625× (Sec. 3.2), and that saving can be taken in either of two ways. Spent on tokens, it holds 1.6× as many visual tokens at the same budget, buying +1.8 to +2.8 points of mean accuracy on LLaVA-NeXT-8B and +6.6 to

Table 10: **End-to-End Latency at Matched KV Budgets.** Wall time (s) of prefill, decode, and end-to-end generation on LLaVA-NeXT-8B with 128 generated tokens, on real TextVQA / ChartQA / InfoVQA validation images at native resolution (median 2,193 visual tokens); medians over 30 image-question pairs on one A100-80GB. Subscripts denote token keep ratios; joint pruning additionally keeps $0.25\times$ Key channels, so each **bold** pair shares the same KV budget. The *italic* columns are an **iso-token control**: token-only pruning run at the joint configuration's token count, and therefore at $1.6\times$ its KV budget. Comparing across the two neighbouring pairs separates the cost of retaining more tokens from the cost of RotateK itself.

			VisionZip			FastV		
B	Stage	Dense	**Token**$_{0.28}$	*Token*$_{0.45}$	**RotateK**$_{0.45}$	**Token**$_{0.25}$	*Token*$_{0.40}$	**RotateK**$_{0.40}$
1	Prefill	0.18	0.07	*0.10*	0.12	0.06	*0.09*	0.11
	Decode	2.57	2.59	*2.57*	2.85	2.53	*2.52*	2.84
	End-to-End	2.91	2.82	*2.84*	3.13	2.75	*2.77*	3.11
8	Prefill	1.32	0.43	*0.67*	0.70	0.44	*0.62*	0.64
	Decode	3.67	2.71	*2.89*	2.92	2.65	*2.79*	2.95
	End-to-End	5.33	3.45	*3.89*	3.95	3.40	*3.71*	3.90
16	Prefill	2.62	0.85	*1.32*	1.37	0.86	*1.21*	1.26
	Decode	5.42	3.26	*3.84*	3.38	3.17	*3.63*	3.27
	End-to-End	8.56	4.60	*5.67*	5.26	4.54	*5.35*	5.04
32	Prefill	5.22	1.69	*2.62*	2.71	1.70	*2.39*	2.47
	Decode	8.87	4.53	*5.70*	4.59	4.33	*5.22*	4.33
	End-to-End	14.98	7.06	*9.18*	8.18	6.91	*8.48*	7.65

+7.5 on Qwen2.5-VL-7B (Table 1) for a one-time prefill cost. Kept instead, it shrinks the cache by $1.60\times$ at an unchanged token count, raising batch capacity and throughput, while also speeding up decode (Sec. 4.3, Table 7). Channel pruning is therefore not a competitor to token sparsity but a second knob orthogonal to it — one that turns a fixed KV budget from a point into a curve, along which accuracy and capacity can be traded without changing the model.

E Additional Experimental Results

E.1 Comparison under Fixed KV Cache Budgets

Additional results under tighter KV cache budgets are provided in Table 17. Consistent with the main results in Table 1, RotateK continues to outperform token-only pruning and prior Key channel pruning methods across most settings at the same KV cache budget, and the gap generally widens at more aggressive compression ratios. The table also reports per-benchmark results for the two additional token pruners of Table 2(b), whose setup we describe below.

PyramidDrop and SparseVLM. Neither official release supports our backbones, so we port both algorithms training-free onto the staged-cut infrastructure used for FastV; single-stage PyramidDrop reproduces FastV exactly, validating the port. PyramidDrop cuts at layers [2, 8, 16] following the authors' guidance, keeping a geometric fraction λ^i of image tokens ranked by the last prompt token's attention. SparseVLM uses text-guided rater scoring with staged cuts at layers [2, 6, 15] and token recycling; since its official keep counts are absolute and tuned to LLaVA-1.5's 576 image tokens, we re-express them as fractions of the original image-token count. Budgets are matched within each group by layer-averaged visual KV: $0.25\times$ for PyramidDrop (given its fixed per-stage keep fractions) and $0.30\times$ for SparseVLM (measured at runtime, since the number of recycled tokens varies per input), with per-stage fractions solved per backbone so that the Key channel factor of joint pruning lands on the same target. All entries use full validation splits.

E.2 Long-Context and Retrieval Benchmarks

Table 11: **Long-video Retrieval on MLVU-needle.** All methods stay within 5 points of the dense model. At the same KV budget, joint pruning with RotateK matches or exceeds token-only pruning.

Method	KV Cache	Accuracy (%)
Dense	1.00×	74.9
$\text{FastV}_{\text{token only}}$	0.25×	70.7
FastV + ThinK	0.25×	70.1
FastV + RotateK	0.25×	71.0
$\text{VisionZip}_{\text{token only}}$	0.28×	71.5
VisionZip + ThinK	0.28×	73.2
VisionZip + RotateK	0.28×	73.0

Table 12: **High-resolution Visual Search on V*Bench.** At the default budget, token pruning alone is not yet damaging, and joint pruning performs on par. Under tighter budgets, channel pruning lets RotateK retain about $1.6\times$ more visual tokens (in parentheses) at the same KV budget, recovering +7.3-7.8 points.

Method	KV Cache	Accuracy (%)
Default budget		
$\text{FastV}_{\text{token only}}$	0.25×	52.36
FastV + ThinK	0.25×	54.97
FastV + RotateK	0.25×	54.97
$\text{VisionZip}_{\text{token only}}$	0.28×	79.06
VisionZip + ThinK	0.28×	78.53
VisionZip + RotateK	0.28×	78.53
Tighter budgets		
$\text{VisionZip}_{\text{token only}}$	0.08×	68.06 (345)
VisionZip + RotateK	0.08×	75.39 (561)
$\text{VisionZip}_{\text{token only}}$	0.06×	65.45 (259)
VisionZip + RotateK	0.06×	73.30 (432)

Video needle-in-a-haystack. MLVU-needle evaluates whether a VLM can locate and use relevant visual evidence in long videos (Zhou et al., 2025; 355 multiple-choice questions, median 7.9 minutes). We use Qwen2.5-VL-7B with 64 frames. Needle retrieval is intrinsically robust to compression, since a single salient region survives most pruning schemes, so margins are small for all methods (Table 11).

High-resolution reasoning. V*Bench (Wu & Xie, 2024; 191 questions, up to ∼4M pixels, ∼5K visual tokens) requires locating a small target in a cluttered scene. At the default budget, VisionZip alone is not damaging on V*, so joint pruning neither helps nor hurts. Under tighter budgets, however, retaining about $1.6\times$ more tokens via channel pruning recovers +7.3-7.8 points (Table 12). RotateK also remains stable where ThinK does not: with VisionZip at $0.45\times$ tokens and 12.5% Key channels, ThinK collapses to 23.6% while RotateK holds 78.5%.

E.3 Composition with KV Cache Quantization

Table 13: **RotateK combined with KIVI.** Mean accuracy (%) over TVQA, CQA, DVQA, IVQA, and VizWiz. "RotateK-k%" keeps k% of Key channels, and KIVI (Liu et al., 2024f) quantizes the remaining KV cache to 4 or 2 bits.

Method	Qwen2.5-VL-7B	LLaVA-NeXT-8B
FP16	79.2	58.2
KIVI-4bit	79.2	58.0
KIVI-2bit	79.0	58.3
RotateK-50%	78.9	57.7
+ KIVI-4bit	79.0	57.6
+ KIVI-2bit	78.0	56.9
RotateK-25%	78.4	55.5
+ KIVI-4bit	78.5	55.6
+ KIVI-2bit	78.0	54.9

RotateK reduces the number of Key channels, while KV quantization reduces the bits per element, so the two act on independent axes and their memory savings multiply. We combine RotateK with KIVI (2- and 4-bit; Liu et al., 2024f) on the five VQA benchmarks (`lmms-eval lite`) in Table 13. Quantization alone is lossless on both backbones (within $\pm$0.2 points of FP16), and adding it on top of RotateK costs at most 0.1 points at 4-bit and 0.4-0.9 points at 2-bit beyond RotateK's own pruning. RotateK with 50% Key channels and 4-bit KIVI yields a $4.3\times$ KV reduction within 0.6 points of FP16, and RotateK with 25% Key channels and 2-bit KIVI yields $8.5\times$.

E.4 Token Dependency of Key Channel Importance

Table 14: **Mask statistics of ThinK vs. SparK at 25% channels.** Pooling both backbones gives a token dependency of 0.906 and a channel agreement of 0.74, corresponding to the 91% token-specific channels and the 26% of ThinK's budget on channels most tokens skip reported in Sec. 2. The larger per-layer spread on Qwen2.5-VL-7B comes almost entirely from its first two layers.

	Token dependency		**Channel agreement**	
	LLaVA-NeXT-8B	Qwen2.5-VL-7B	LLaVA-NeXT-8B	Qwen2.5-VL-7B
Mean over all comparisons	0.922	0.869	0.728	0.777
Range of per-dataset means	0.917-0.926	0.857-0.881	0.710–0.751	0.771–0.784
Std. of per-head means	0.004	0.013	0.024	0.018
Std. of per-layer means	0.028	0.164	0.087	0.091
excluding Qwen layers 0-1	0.028	0.065	0.087	0.089

We analyze the channel masks of ThinK (head-wise) and SparK (token-wise) at 32 of 128 channels, and define two statistics. *Token dependency* (within SparK) is the fraction of channels a token keeps that are not kept by every token; high values indicate token-specific selection. *Channel agreement* (ThinK vs. SparK) is the fraction of channels in ThinK's head-wise mask that are also kept by a majority of tokens in SparK. We sweep 2 model families $\times$ 4 datasets $\times$ 3 images $\times$ all decoder layers $\times$ all KV heads, for 4,416 (layer, head) comparisons (Table 14).

Of the 32 channels SparK keeps per token, only about three are kept by every token, and 26% of ThinK's fixed budget falls on channels that most tokens do not treat as important. Both statistics are nearly invariant across KV heads, datasets, and model families; the only deviation is in the first two layers of Qwen2.5-VL.

E.5 Comparison with Fixed and Trained Bases

A natural alternative to constructing the basis per input is to fix it offline, either by calibrating it once on held-out data or by learning it. We compare RotateK against three such alternatives: a static PCA rotation, a lightly trained rotation following MatryoshkaKV (Lin et al., 2025), and LeanK (Zhang et al., 2025c), which learns a head-wise channel mask. The first two change the basis but fix it across inputs, while LeanK keeps the original basis and learns which channels to keep.

Table 15: **RotateK vs. Static and Lightly Trained Rotations.** Accuracy (%) on lmms-eval lite validation splits. Static and trained entries are averaged over three calibration and training domains.

Method	**Training**	TextVQA	ChartQA	InfoVQA
LLaVA-NeXT-8B, 25% Key channels				
Static PCA	calibration	62.0	59.5	30.0
Lightly trained rotation	KD	60.5	58.3	29.9
RotateK (online)	none	65.4	61.4	31.9
LLaVA-NeXT-8B, 12.5% Key channels				
Static PCA	calibration	46.4	52.0	27.3
Lightly trained rotation	KD	49.9	53.1	27.5
RotateK (online)	none	53.5	56.6	27.9
Qwen2.5-VL-7B, 25% Key channels				
Static PCA	calibration	82.9	76.7	77.1
Lightly trained rotation	KD	85.4	77.4	78.6
RotateK (online)	none	85.0	78.0	78.5
Qwen2.5-VL-7B, 12.5% Key channels				
Static PCA	calibration	60.9	73.3	66.5
Lightly trained rotation	KD	69.5	74.5	69.5
RotateK (online)	none	78.1	77.4	71.4

Fixed and trained rotations. *Setup.* The static PCA rotation is computed once per (layer, head) from calibration data and reused for all inputs, without query weighting. The lightly trained rotation follows MatryoshkaKV: a per-(layer, head) orthogonal projection parameterized by the Cayley map, initialized from PCA, and optimized by knowledge distillation from the uncompressed model with the backbone frozen, so that only the projections are trained. Both are applied at the same Key channel budget as RotateK. We evaluate on LLaVA-NeXT-8B and Qwen2.5-VL-7B at 25% and 12.5% Key channels, using `lmms-eval lite` validation splits, and average the static and trained entries over three calibration and training domains.

Observations. In Table 15, (i) At the mild 25% budget, all three variants remain close, and training yields no consistent benefit: the trained rotation is ahead of RotateK by at most 0.4 points on two of the three Qwen2.5-VL benchmarks, within noise, and behind it on all LLaVA-NeXT benchmarks. (ii) At the aggressive 12.5% budget, fixed bases degrade sharply. On Qwen2.5-VL-7B TextVQA, static PCA drops to 60.9, and light training recovers it only to 69.5, consistent with the observation in MatryoshkaKV that trained projections outperform static PCA under heavy compression. RotateK, however, retains 78.1 without any training, ahead of the trained rotation by 8.6 points and of static PCA by 17.2 points. The same ordering holds on LLaVA-NeXT-8B (53.5 vs. 49.9 vs. 46.4).

Table 16: **Comparison with LeanK.** "LeanK ← X" denotes a mask trained on dataset X; same-dataset cells are omitted. Qwen2.5-VL-7B, FastV ($0.30\times$ tokens), 25% Key channels.

Method	**Training**	TextVQA	ChartQA	InfoVQA
ThinK	none	78.38	63.80	55.22
LeanK ← TextVQA	mask	–	59.80	57.35
LeanK ← ChartQA	mask	82.64	–	56.70
LeanK ← InfoVQA	mask	78.12	58.80	–
RotateK	none	83.50	63.80	58.05

Learned channel masks. *Setup.* LeanK is released only for a text LLM pipeline, so we port its objective rather than its code, and calibrate on VLM data rather than transplanting a text-trained mask. We use Qwen2.5-VL-7B-Instruct with FastV (K=2, token keep ratio 0.30) and 25% Key channels, evaluated on lmms-eval lite validation splits (n=500 each). The backbone is frozen, and the only trainable tensor is LeanK's channel scaling factor $\alpha \in \mathbb{R}^{28\times4\times128}$ (one entry per layer, KV head, and channel; 14,336 scalars in total). Each training step runs two forward passes, a dense teacher without gradients and an α-scaled student, and we follow LeanK's self-distillation and ℓ_1 objective with its published two-stage recipe (stage 1: learning rate 0.04, λ=0.06, 2,000 updates; stage 2: learning rate 0.02, λ=0, 200 updates). Binarization uses LeanK's own implementation.

Each mask is trained on one dataset and evaluated on the other two, so that every LeanK entry is out-of-distribution, as a fixed offline mask would be in deployment.

Observations. In Table 16, (i) RotateK exceeds the better of the two out-of-distribution LeanK masks on all three benchmarks: 83.50 vs. 82.64 on TextVQA, 63.80 vs. 59.80 on ChartQA, and 58.05 vs. 57.35 on InfoVQA. (ii) More notably, the calibration domain alone moves LeanK by up to 4.5 points on TextVQA (78.12 when trained on InfoVQA vs. 82.64 when trained on ChartQA). In VLM workloads, which channels matter depends strongly on the visual domain, so a mask fixed offline inherits the data it was fitted on, whereas RotateK derives its basis from each input.

Summary. The two comparisons reach the same conclusion from opposite directions: rotating the basis but fixing it offline, and keeping the basis but learning the mask, both fall short of per-input construction at aggressive compression. For the learned alternatives we evaluate, the limitation in the VLM setting is thus not capacity but *fixedness*. We focus on alternatives that preserve RotateK's deployment setting, a frozen backbone with a head-wise structured Key cache; learned low-rank projections that jointly retrain the query and key paths form a separate, training-based category that would likewise be fitted to a fixed domain.

Table 17: **Comparison under Fixed KV Cache Budgets**. Extends Table 1 with additional KV cache budgets and token pruners. PyramidDrop and SparseVLM budgets are layer-averaged.

Method	Token	Channel	KV Cache	TVQA	IVQA	CQA	DVQA	VizWiz	mean
LLAMA3-LLAVA-NEXT-8B									
Baseline	1.00×	1.00×	1.00×	65.40	32.27	69.08	72.44	57.50	59.34
$\text{FastV}_{\text{token only}}$	0.25×	1.00×	0.25×	62.69	25.91	59.64	56.39	58.56	52.64
FastV + ThinK	0.40×	0.25×	0.25×	54.07	25.51	60.56	51.72	57.70	49.91
FastV + RotateK	0.40×	0.25×	0.25×	62.67	28.37	65.25	62.46	58.34	55.42
$\text{FastV}_{\text{token only}}$	0.20×	1.00×	0.20×	61.03	24.25	54.28	51.23	58.16	49.79
FastV + ThinK	0.30×	0.25×	0.19×	54.07	23.84	58.76	47.33	57.41	48.28
FastV + RotateK	0.30×	0.25×	0.19×	61.53	25.40	62.36	57.59	58.53	53.08
$\text{VisionZip}_{\text{token only}}$	0.28×	1.00×	0.28×	63.33	27.70	57.32	64.08	58.97	54.28
VisionZip + ThinK	0.45×	0.25×	0.28×	54.22	27.81	58.52	53.77	57.81	50.43
VisionZip + RotateK	0.45×	0.25×	0.28×	62.62	28.97	62.88	66.99	58.94	56.08
$\text{VisionZip}_{\text{token only}}$	0.22×	1.00×	0.22×	62.07	26.11	55.16	59.18	58.95	52.29
VisionZip + ThinK	0.35×	0.25×	0.22×	54.24	27.81	55.56	50.76	58.20	49.31
VisionZip + RotateK	0.35×	0.25×	0.22×	62.73	27.70	60.16	64.73	59.33	54.93
$\text{PyramidDrop}_{\text{token only}}$	0.25×	1.00×	0.25×	61.35	25.62	58.36	51.30	57.68	50.86
PyramidDrop + ThinK	0.40×	0.25×	0.25×	55.24	25.94	62.36	53.06	57.66	50.85
PyramidDrop + RotateK	0.40×	0.25×	0.25×	63.21	28.92	65.56	64.31	57.50	55.90
$\text{SparseVLM}_{\text{token only}}$	0.30×	1.00×	0.30×	63.37	26.99	62.48	58.89	58.46	54.04
SparseVLM + ThinK	0.48×	0.25×	0.30×	55.67	26.97	63.56	55.66	57.68	51.91
SparseVLM + RotateK	0.48×	0.25×	0.30×	63.80	30.02	67.56	67.68	58.09	57.43
QWEN2.5-VL-7B-INSTRUCT									
Baseline	1.00×	1.00×	1.00×	82.92	80.12	83.04	94.40	70.58	82.21
$\text{FastV}_{\text{token only}}$	0.25×	1.00×	0.25×	81.49	52.76	67.44	75.04	69.13	69.17
FastV + ThinK	0.40×	0.25×	0.25×	77.81	63.01	77.12	81.74	69.25	73.79
FastV + RotateK	0.40×	0.25×	0.25×	82.38	65.79	77.40	87.28	70.32	76.63
$\text{FastV}_{\text{token only}}$	0.20×	1.00×	0.20×	80.44	46.73	63.08	66.06	68.34	64.93
FastV + ThinK	0.30×	0.25×	0.19×	77.76	54.79	71.40	75.58	68.82	69.67
FastV + RotateK	0.30×	0.25×	0.19×	81.80	56.82	71.32	80.13	69.86	71.99
$\text{VisionZip}_{\text{token only}}$	0.28×	1.00×	0.28×	76.84	58.46	70.32	84.71	68.97	71.86
VisionZip + ThinK	0.45×	0.25×	0.28×	75.72	67.13	79.64	85.83	68.93	75.45
VisionZip + RotateK	0.45×	0.25×	0.28×	80.84	70.51	79.56	91.29	70.13	78.47
$\text{VisionZip}_{\text{token only}}$	0.22×	1.00×	0.22×	74.19	50.58	64.40	78.01	68.84	67.20
VisionZip + ThinK	0.35×	0.25×	0.22×	73.83	60.95	76.44	82.78	68.35	72.47
VisionZip + RotateK	0.35×	0.25×	0.22×	78.48	63.86	75.84	87.91	69.38	75.09
$\text{PyramidDrop}_{\text{token only}}$	0.25×	1.00×	0.25×	79.15	39.71	48.28	68.11	68.11	60.67
PyramidDrop + ThinK	0.40×	0.25×	0.25×	77.00	55.30	71.64	79.04	68.79	70.35
PyramidDrop + RotateK	0.40×	0.25×	0.25×	81.79	57.75	72.00	84.44	69.68	73.13
$\text{SparseVLM}_{\text{token only}}$	0.30×	1.00×	0.30×	80.34	48.40	53.48	73.74	68.97	64.99
SparseVLM + ThinK	0.48×	0.25×	0.30×	76.64	61.81	70.92	80.20	68.70	71.65
SparseVLM + RotateK	0.48×	0.25×	0.30×	81.87	64.63	70.72	86.46	69.96	74.73

Table 18: **Comparison under Fixed KV Cache Budgets on Additional Backbones.** Recent (Qwen3-VL-8B, InternVL3.5-8B) and larger (Qwen2.5-VL-32B, InternVL3.5-14B) VLMs. Qwen3-VL uses FastV with $K{=}4$ owing to its DeepStack backbone; all others use $K{=}2$. The Qwen2.5-VL-32B baseline does not always exceed the 7B model under exact-match scoring; per-sample inspection shows that many of its "errors" are correct but more verbose answers penalized by strict string matching.

Method	Token	Channel	KV Cache	TVQA	IVQA	CQA	DVQA	VizWiz	mean
QWEN3-VL-8B-INSTRUCT									
Baseline	1.00×	1.00×	1.00×	81.56	80.88	84.92	94.98	69.36	82.34
$\text{FastV}_{\text{token only}}$	0.25×	1.00×	0.25×	76.44	43.95	59.92	69.20	69.55	63.81
FastV + ThinK	0.40×	0.25×	0.25×	75.98	56.12	72.52	78.49	69.66	70.55
FastV + RotateK	0.40×	0.25×	0.25×	79.67	57.84	73.84	83.17	70.35	72.97
$\text{VisionZip}_{\text{token only}}$	0.28×	1.00×	0.28×	74.81	55.24	72.24	80.41	69.88	70.52
VisionZip + ThinK	0.45×	0.25×	0.28×	74.94	65.01	78.88	85.85	68.84	74.70
VisionZip + RotateK	0.45×	0.25×	0.28×	80.01	68.44	79.72	91.54	70.11	77.96
INTERNVL3.5-8B									
Baseline	1.00×	1.00×	1.00×	77.98	71.06	86.60	89.56	58.16	76.67
$\text{FastV}_{\text{token only}}$	0.25×	1.00×	0.25×	74.54	39.96	65.56	54.16	57.57	58.36
FastV + ThinK	0.40×	0.25×	0.25×	73.93	49.26	78.36	69.86	57.09	65.70
FastV + RotateK	0.40×	0.25×	0.25×	76.63	50.46	78.60	72.99	57.77	67.29
$\text{VisionZip}_{\text{token only}}$	0.28×	1.00×	0.28×	62.61	36.93	43.16	48.85	57.27	49.76
VisionZip + ThinK	0.45×	0.25×	0.28×	68.76	47.50	61.04	62.63	56.43	59.27
VisionZip + RotateK	0.45×	0.25×	0.28×	72.02	48.65	61.60	66.69	57.63	61.32
QWEN2.5-VL-32B-INSTRUCT									
Baseline	1.00×	1.00×	1.00×	77.25	80.67	69.12	92.55	64.73	76.86
$\text{FastV}_{\text{token only}}$	0.25×	1.00×	0.25×	70.24	44.91	45.24	61.64	63.60	57.13
FastV + ThinK	0.40×	0.25×	0.25×	71.40	55.14	58.56	73.40	63.76	64.45
FastV + RotateK	0.40×	0.25×	0.25×	74.84	57.52	60.92	77.22	64.61	67.02
$\text{VisionZip}_{\text{token only}}$	0.28×	1.00×	0.28×	66.04	52.06	58.60	77.21	64.51	63.68
VisionZip + ThinK	0.45×	0.25×	0.28×	70.41	65.37	64.56	84.07	63.53	69.59
VisionZip + RotateK	0.45×	0.25×	0.28×	73.54	67.77	65.20	88.93	64.48	71.98
INTERNVL3.5-14B									
Baseline	1.00×	1.00×	1.00×	76.82	73.34	86.88	90.24	56.53	76.76
$\text{FastV}_{\text{token only}}$	0.25×	1.00×	0.25×	74.83	46.27	76.92	61.54	56.15	63.14
FastV + ThinK	0.40×	0.25×	0.25×	72.68	54.77	82.12	70.00	55.80	67.07
FastV + RotateK	0.40×	0.25×	0.25×	75.90	56.56	82.88	74.10	56.52	69.19
$\text{VisionZip}_{\text{token only}}$	0.28×	1.00×	0.28×	65.76	40.87	59.00	51.66	55.10	54.48
VisionZip + ThinK	0.45×	0.25×	0.28×	68.75	50.57	71.80	64.96	54.83	62.18
VisionZip + RotateK	0.45×	0.25×	0.28×	71.93	52.30	72.84	69.40	55.80	64.45

F CHOLESKY-BASED SUBSPACE ITERATION

RotateK computes the PCA basis online at the end of prefill, making the efficiency of eigendecomposition critical for practical deployment. A standard PCA pipeline first computes the covariance matrix $C_q \in \mathbb{R}^{d\times d}$ at cost $\mathcal{O}(Nd^2)$, followed by a full symmetric eigendecomposition at cost $\mathcal{O}(d^3)$. However, RotateK ultimately retains only the top-k channels, making the computation of the full d-dimensional eigenspace unnecessary. To avoid this overhead, RotateK directly estimates only the top-k eigenspace using a Cholesky-based subspace iteration method.

We first briefly review the intuition behind subspace iteration. Consider the eigendecomposition of the covariance matrix:

$$C_q = U\Lambda U^\top, \tag{8}$$

where $U = [u_1, \dots, u_d]$ contains the eigenvectors and $\Lambda = \mathrm{diag}(\lambda_1, \dots, \lambda_d)$ contains the eigenvalues ordered as $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d \geq 0$. By definition, each eigenvector satisfies:

$$C_q u_i = \lambda_i u_i. \tag{9}$$

That is, multiplication by C_q preserves the direction of an eigenvector while scaling its magnitude by the corresponding eigenvalue.

Now consider an arbitrary initial vector $v^{(0)} \in \mathbb{R}^d$, which can be expressed in the eigenbasis of C_q:

$$v^{(0)} = a_1 u_1 + a_2 u_2 + \cdots + a_d u_d. \tag{10}$$

Applying the covariance matrix repeatedly yields:

$$v^{(1)} = C_q v^{(0)} = a_1\lambda_1 u_1 + a_2\lambda_2 u_2 + \cdots, \tag{11}$$

$$v^{(t)} = C_q^t v^{(0)} = a_1\lambda_1^t u_1 + a_2\lambda_2^t u_2 + \cdots. \tag{12}$$

Since the dominant eigenvalue satisfies $\lambda_1^t \gg \lambda_2^t$ as t increases, the component along u_1 progressively dominates:

$$v^{(t)} \approx u_1. \tag{13}$$

This is the classical power iteration method: repeated multiplication by the covariance matrix amplifies directions associated with large eigenvalues. In the context of PCA, the eigenvectors of $C_q = K_q^\top K_q$ correspond to directions of large query-weighted variance, so power iteration progressively aligns vectors with the principal PCA directions.

RotateK, however, requires not only the dominant eigenvector but the entire top-k eigenspace. Subspace iteration extends power iteration from a single vector to a k-dimensional basis:

$$V^{(0)} = [v_1^{(0)}, \dots, v_k^{(0)}] \in \mathbb{R}^{d\times k}. \tag{14}$$

At each iteration, the basis is multiplied by the covariance matrix:

$$V^{(t+1)} \leftarrow C_q V^{(t)}. \tag{15}$$

Repeated multiplication progressively amplifies directions associated with the largest eigenvalues, causing the column space of $V^{(t)}$ to converge toward the top-k eigenspace of C_q.

A naive iteration, however, causes all basis vectors to collapse toward the dominant eigenvector u_1. To maintain a valid k-dimensional subspace, each iteration additionally performs orthogonalisation. Rather than using a full QR decomposition, RotateK employs a more lightweight Cholesky-QR procedure. First, the Gram matrix is constructed:

$$G^{(t+1)} = (V^{(t+1)})^\top V^{(t+1)}. \tag{16}$$

Since $G^{(t+1)}$ is positive definite, it admits a Cholesky factorisation:

$$G^{(t+1)} = L^{(t+1)}(L^{(t+1)})^\top. \tag{17}$$

The basis is then orthogonalised via:

$$V^{(t+1)} \leftarrow V^{(t+1)}(L^{(t+1)})^{-\top}. \tag{18}$$

This transformation guarantees orthonormal columns since:

$$(V^{(t+1)})^\top V^{(t+1)} = L^{-1}G^{(t+1)}(L^{-1})^\top \tag{19}$$

$$= L^{-1}LL^\top(L^{-1})^\top = I. \tag{20}$$

In practice, Algorithm 1 adds a small relative ridge $\rho = \epsilon\,\mathrm{tr}(G)/k$ for numerical stability, with $\epsilon = 10^{-4}$ by default (10^{-6} for the query-agnostic variant, whose spectrum is better conditioned), so the basis is orthonormal only approximately, with $V^\top V = I - \rho L^{-1}L^{-\top}$. This has no measurable effect on accuracy, as the exactly orthonormal `eigh` basis performs on par (Table 3).

The dominant cost of each iteration arises from the matrix multiplication $C_q V^{(t)}$, resulting in a per-iteration complexity of $\mathcal{O}(d^2k)$ and a total eigensolve complexity of $\mathcal{O}(Td^2k)$ over T iterations. The overall PCA complexity therefore becomes:

$$\mathcal{O}(Nd^2 + Td^2k), \tag{21}$$

compared to $\mathcal{O}(Nd^2 + d^3)$ for full eigendecomposition.

Importantly, at the small matrix sizes relevant to KV-cache pruning (*e.g.*, d=128), practical latency is dominated less by arithmetic throughput than by GPU kernel dispatch overhead. Full eigendecomposition launches many small CUDA kernels for tridiagonalisation, eigensolving, and eigenvector reconstruction, whereas subspace iteration repeatedly executes only a small set of structured operations (matrix multiplication, Gram matrix construction, Cholesky factorisation, and triangular solve) with fixed tensor shapes. This structure enables efficient CUDA graph capture and replay, substantially reducing wall-clock latency for online PCA despite comparable FLOPs.

G ALGORITHMIC OVERVIEW OF ROTATEK

Algorithm 1 RotateK – Prefill (Rotation Construction).

Require: Key states $K \in \mathbb{R}^{N\times d}$, recent queries $Q_W \in \mathbb{R}^{W\times d}$, target rank k, iteration count T, ridge factor ϵ

Ensure: Cached rotated keys $\tilde{K}$, truncated rotation R_k, mean residual $\delta\mu$

 # Step 1: centered, query-weighted covariance

$\mu \leftarrow \frac{1}{N}\sum_{n=1}^{N} K_n$ ▷ per-channel mean of K

$\bar{K} \leftarrow K - \mathbf{1}_N \mu^\top$ ▷ centered keys

$\bar{C} \leftarrow \bar{K}^\top \bar{K}$ ▷ centered covariance, $\mathcal{O}(Nd^2)$

$\sigma_j \leftarrow \|(Q_W)_{:,j}\|_2$ for $j = 1, \ldots, d$ ▷ per-channel query magnitudes

$C_q \leftarrow (\sigma\,\sigma^\top) \odot \bar{C}$ ▷ Hadamard, $\mathcal{O}(d^2)$

 # Step 2: subspace iteration with shifted Cholesky-QR

Sample $V \in \mathbb{R}^{d\times k}$ with i.i.d. $\mathcal{N}(0,1)$ entries

for $t = 1, \ldots, T$ **do**

  $V \leftarrow C_q V$ ▷ GEMM, $\mathcal{O}(d^2 k)$

  $G \leftarrow V^\top V$ ▷ Gram, $\mathcal{O}(dk^2)$

  $\rho \leftarrow \epsilon \cdot \mathrm{tr}(G)/k$ ▷ trace-scaled ridge

  $L \leftarrow \mathrm{Cholesky}(G + \rho I_k)$ ▷ $\mathcal{O}(k^3)$

  $V \leftarrow V L^{-\top}$ ▷ triangular solve

end for

$R_k \leftarrow V$ ▷ top-k basis of C_q

 # Step 3: cache rotated keys and mean correction

$\tilde{K} \leftarrow K R_k$ ▷ stored in place of K

$\delta\mu \leftarrow \mu - R_k\, R_k^\top \mu$ ▷ $(I_d - P_k)\,\mu$, used at decode

return $\tilde{K},\ R_k,\ \delta\mu$

Algorithm 2 RotateK – Decode (per step).

Require: Query $q_t \in \mathbb{R}^d$, cached visual $(\tilde{K}, R_k, \delta\mu)$, prompt+text keys $K_{\mathrm{pt}} \in \mathbb{R}^{M\times d}$ and concatenated values V

Ensure: Attention output $\tilde{a}_t$

$\tilde{q}_t \leftarrow q_t R_k$ ▷ rotate query, $\mathcal{O}(dk)$

$b_t \leftarrow q_t^\top \delta\mu$ ▷ scalar mean-correction bias

$s_{\mathrm{vis}} \leftarrow \big(\tilde{q}_t \tilde{K}^\top + b_t\, \mathbf{1}_N^\top\big)/\sqrt{d}$ ▷ rotated visual scores ($\in \mathbb{R}^N$)

$s_{\mathrm{pt}} \leftarrow q_t\, K_{\mathrm{pt}}^\top/\sqrt{d}$ ▷ unrotated prompt+text scores ($\in \mathbb{R}^M$)

$s \leftarrow \big[\, s_{\mathrm{vis}};\ s_{\mathrm{pt}} \,\big]$ ▷ concatenate over the $N + M$ tokens

$\tilde{a}_t \leftarrow \mathrm{softmax}(s) \cdot V$ ▷ weighted sum of values

return $\tilde{a}_t$

H LIMITATIONS AND FUTURE WORK

RotateK builds its basis from prefill queries, which serve as a proxy for the queries that follow. This proxy fits single-turn inference, the setting we target. Extending it to settings where later queries may drift from the prefill, such as multi-turn dialogue or shared-prefix serving over a cached image, is a natural next step, for example by periodically refreshing the basis or adopting a query-agnostic variant. The basis is also shared across all tokens within a head, which keeps pruning structured and hardware-friendly. RotateK thus matches token-wise pruning, which selects channels separately for each token, at most sparsity levels, with a small gap remaining only at the most aggressive setting (Sec. 4.2); closing this gap while preserving structured sparsity is left to future work.

On efficiency, RotateK spends additional prefill compute on the extra visual tokens it retains, which is precisely what enables its accuracy gains. Relative to token-only pruning at the same KV budget, this amounts to a modest end-to-end cost, while its decode overhead vanishes at larger batch sizes. Its benefits are most pronounced when visual tokens dominate the KV cache, which is the regime this work targets.

Finally, our evaluation covers several open-weight VLM families with explicit per-head KV caches, an architecture still common among widely used VLMs. MLA-based VLMs are emerging (*e.g.*, Kimi-VL, DeepSeek-VL2), but their jointly compressed KV latents require a different choice of compression space and decode path. We leave this extension and its accuracy–efficiency trade-off to future work. Combining the per-input basis with training-based Key compression, for example by applying the rotation on top of a learned projection, is another natural extension. Grouping heads when building the per-input basis, which keeps more Key energy at the same storage per token (Appendix B.2), is a further extension.