JASP: A JOINT ATTENTION-SCORE PROXY FOR LONG-CONTEXT KV-CACHE OFFLOADING

Anonymous authors
Paper under double-blind review

ABSTRACT

Long-context LLM inference is increasingly important, but its growing KV cache consumes substantial GPU memory and slows autoregressive decoding. CPU offloading relieves this memory pressure, but sparse decoding must still identify important tokens from the complete history and retrieve their K/V states at every step. This process relies on a GPU-resident proxy whose accuracy determines how few tokens can be retrieved, while its footprint controls the history-dependent bytes read during scoring and is therefore a major contributor to scan latency. Existing projection-based proxies typically construct a separate index for each KV head, repeatedly storing and scanning related retrieval information. We show that KV heads share substantial query-conditioned attention-score structure across layers. Based on this observation, we introduce JASP, a joint attention-score proxy that learns one score-aligned code per historical token across KV heads. Unlike key reconstruction, its objective directly preserves the query–key scores that determine token rankings. Query-specific scoring preserves distinct token rankings, and each KV group retains its own selected token set. JASP therefore removes redundant historical proxy traffic while preserving head-specific retrieval behavior. We further combine GPU-driven selected-row retrieval with capacity-balanced CPU/GPU KV placement to translate the compact proxy into serving gains. Under saturated serving, JASP achieves 1.9–2.6$\times$ the aggregate decode throughput of FlashAttention-2 across 16K–128K contexts. JASP scores 75.9 on the complete 128K RULER suite, compared with dense attention's 78.9, while matching dense quality on LongBench. Code is released at Anonymous/JASP.

1 INTRODUCTION

Large language models based on the Transformer architecture (Vaswani et al., 2017) are increasingly deployed for long-context inference (Bai et al., 2024; Hsieh et al., 2024). During autoregressive decoding, the key–value (KV) cache grows linearly with context length and batch size. Keeping the full cache on the GPU consumes substantial memory, limiting feasible context length and serving concurrency, while reading it at every step makes attention increasingly memory-bandwidth bound (Li et al., 2024a; Kwon et al., 2023; Dao et al., 2022). Sparse KV offloading addresses this capacity bottleneck by retaining the full KV history in host memory and transferring only selected rows to the GPU at each step (Lee et al., 2024; Zhang et al., 2025; Sun et al., 2025; Li et al., 2025). This enables longer contexts and potentially larger serving batches, but capacity efficiency does not guarantee execution efficiency. Before attention can use the selected rows, the system must identify them from the full history and retrieve them across PCIe. These overheads can exceed the savings from attending to fewer tokens, making sparse-offloaded decoding slower than GPU-resident dense attention. The goal is therefore not merely to fit long contexts, but to convert the resulting memory savings into higher serving throughput.

To identify the KV rows without scanning the full cache, sparse-offloading systems maintain a compact GPU-resident *proxy* over the historical cache. At every decoding step, the current query scans this proxy to estimate attention scores and rank the tokens, after which only the selected original K/V rows are retrieved. The proxy must be accurate: ranking errors can discard important tokens, whereas a faithful ranking allows the system to retrieve a much smaller subset without sacrificing attention quality. It must also have a small footprint because the entire proxy is scanned at every decoding step. For memory-bound proxy scoring, a smaller footprint reduces both GPU storage and

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.

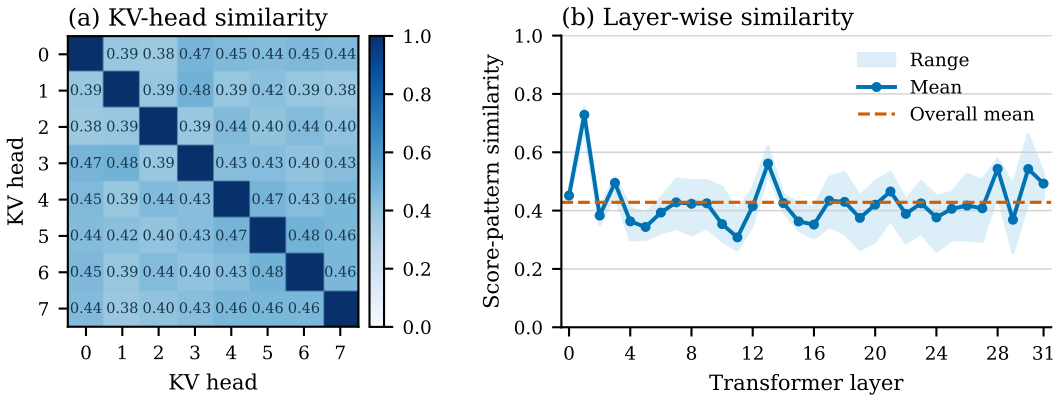


Figure 1: **KV heads share query-conditioned score structure.** Score-pattern similarity is substantial across KV heads and remains consistent across layers. Details are in Appendix E.3.

scan traffic, a major component of selection latency. The central proxy-design challenge is therefore to preserve accurate token rankings with the smallest possible footprint.

Projection-based proxies commonly use PCA to address this trade-off, retaining the dominant directions of each KV head's keys to estimate token rankings in a lower-dimensional space (Singhania et al., 2024). However, they typically build an independent subspace and proxy for every KV head, preventing shared retrieval structure across heads from being represented jointly. We find substantial shared structure: under real decoding queries, different KV heads produce similar attention-score patterns over the same history, and this similarity persists throughout the network. This query-conditioned score space is precisely what a retrieval proxy should preserve, because it approximates the query–key scores that determine token rankings rather than reconstructing the keys themselves. Figure 1 directly visualizes this similarity. Consequently, independent head-wise proxies spend separate capacity representing related score information, limiting their accuracy under a small proxy footprint.

Building on this observation, we introduce JASP, a joint attention-score proxy that decouples a shared key-side index from query-specific scoring. On the key side, JASP jointly learns across KV heads and stores only one compact code for each historical token. Because this index grows with the context and is scanned at every decoding step, sharing it removes redundant per-head storage and memory traffic. Joint learning also gives each head access to a richer shared subspace, while a score-reconstruction objective directs this capacity toward the query–key scores that determine token rankings. On the query side, KV-group-specific factor blocks preserve distinct score interpretations and retrieval decisions. These factors are applied only to the current query, so their cost does not grow with the context. JASP then retrieves the selected original K/V for exact attention over the selected set.

Beyond token selection, sparse offloading must retrieve the selected KV rows efficiently and use CPU and GPU memory jointly. JASP combines a GPU-driven UVA gather with HugePages and row-coalesced access, avoiding CPU-side staging during retrieval. It also partitions full K/V by token position between a CPU prefix and GPU tail, storing each token on exactly one device and balancing capacity across both. At 1% dynamic retention, JASP recovers 98.8% of the attention mass captured by oracle top-k and scores 75.9 on the complete 128K RULER suite, compared with dense attention's 78.9. Under saturated serving across 16K–128K contexts, JASP supports approximately twice the batch of dense implementations and achieves 1.9–2.6$\times$ their aggregate decode throughput.

Our main contributions are as follows:

- We reveal substantial query-conditioned attention-score structure shared across KV heads, showing that independent per-head proxies redundantly encode retrieval-relevant information.

- We propose JASP, which fits one score-aligned key-side index shared across heads while retaining query-specific scoring and KV-group-specific selection, yielding accurate rankings with a compact proxy.
- We integrate GPU-driven retrieval, HugePages, and capacity-balanced KV placement into a CPU-offloaded serving system, demonstrating its quality, serving capacity, and decode throughput on long-context benchmarks.

2 RELATED WORK

2.1 SPARSE KV SELECTION AND RETRIEVAL PROXIES.

Long-context sparse attention either evicts KV entries permanently or preserves the history and reselects tokens for each query. H2O (Zhang et al., 2023) and Scissorhands (Liu et al., 2023) retain historically important tokens, StreamingLLM (Xiao et al., 2024) preserves attention sinks and recent tokens, and Keyformer, SnapKV, and PyramidInfer derive retained sets from accumulated or prompt-time signals (Adnan et al., 2024; Li et al., 2024b; Yang et al., 2024). These methods bound cache size, but evicted tokens cannot be recovered when their relevance changes. Orthogonal compressors instead reduce KV precision, width, or depth (Liu et al., 2024c; Hooper et al., 2024; Xu et al., 2024; Liu et al., 2024a). Query-dependent methods preserve the history and rank it at every decoding step. SparQ (Ribar et al., 2024) selects query-relevant key dimensions, Quest (Tang et al., 2024) bounds page-level scores with channel extrema, MagicPIG (Chen et al., 2025) samples through locality-sensitive hashing, and PQCache (Zhang et al., 2025) estimates query–key products from product-quantized keys. Projection-based proxies include Loki (Singhania et al., 2024), which applies PCA independently to each attention head; ShadowKV (Sun et al., 2025), which reconstructs selected keys from a low-rank representation; and LRQK (Li et al., 2025), which factorizes queries and keys within each head. SALS projects pre-RoPE KV states from multiple heads into a shared latent space and selectively reconstructs retrieved keys, targeting compression of the resident KV cache rather than a compact index for retrieving offloaded model-precision states (Mu et al., 2025). Despite their different approximations, offloading-oriented proxies generally organize history-dependent information per head or optimize it within each head, so aggressively shrinking each proxy can discard information needed for accurate ranking.

2.2 KV OFFLOADING AND HETEROGENEOUS-MEMORY SERVING.

KV offloading moves model-precision K/V states from scarce GPU memory to larger host memory, extending context length and serving concurrency. General systems schedule model states across heterogeneous memory (Sheng et al., 2023), disaggregate prefill and decoding (Zhong et al., 2024), or compress and stream reusable KV state (Liu et al., 2024b). KV-specific sparse-offloading systems instead keep selection metadata on the GPU and transfer only chosen model-precision rows. InfiniGen (Lee et al., 2024) and PQCache (Zhang et al., 2025) follow this pattern, while SpeCache (Jie et al., 2025) predicts future selections and overlaps transfer with computation. ParisKV (Qi et al., 2026) removes CPU-side gathering by letting GPU kernels access mapped host memory through CUDA Unified Virtual Addressing. Sparse transfer lowers PCIe volume, but proxy scanning, irregular access, address translation, and synchronization can remain costly. Moreover, placing all model-precision K/V on the CPU can limit serving batches while compact GPU proxies leave device memory underused.

3 METHOD

Figure 2 summarizes JASP's end-to-end execution. During dense prefill, JASP uses sampled post-RoPE (Su et al., 2024) queries and keys to construct a request-specific, score-aligned proxy. It stores one compact code for each historical token on the GPU, while partitioning the original model-precision K/V between a CPU prefix and a GPU tail. At each decode step, the new code is appended to the shared index and the new K/V to the GPU tail. Each query head then scores the shared history through its KV-group-specific factor block, and the resulting relevance scores are aggregated within each KV group to select one retrieval set. Finally, a GPU-driven kernel gathers the selected original K/V rows from both memory pools and supplies them to standard attention.

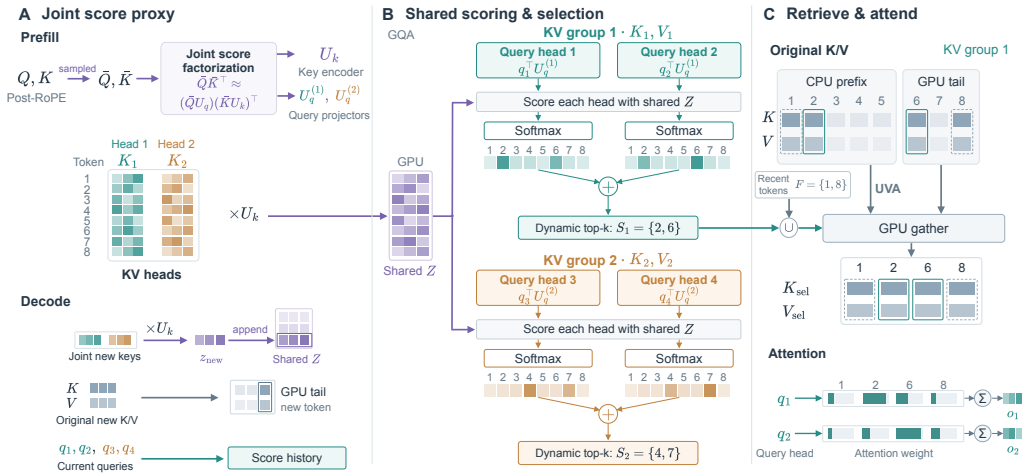


Figure 2: **JASP overview.** A score-aligned code is shared across KV heads, while query-specific scores produce a separate top-k set for each KV group. JASP gathers the selected original K/V rows from the CPU prefix and GPU tail for exact sparse attention.

3.1 Problem Setup

Consider an attention layer with H_q query heads, H_{kv} KV heads, head dimension d, and T historical tokens. We refer to a KV head and the query heads mapped to it as a *KV group*, following grouped-query attention (Ainslie et al., 2023); MHA is the special case in which each group contains one query head. Query head h attends to KV head $g(h)$, producing the full attention score for historical token i from their post-RoPE representations:

$$s_h(i) = \frac{q_h^\top k_{g(h),i}}{\sqrt{d}}. \tag{1}$$

At each decode step, a GPU-resident proxy scans all historical tokens and estimates these scores without accessing the complete model-precision KV cache. Each KV group uses the estimates to select k dynamic positions, combines them with fixed sink and local positions, and retrieves the corresponding original K/V for standard attention. The proxy is therefore responsible for token ranking and selection, while the final attention is computed from the original K/V states.

The proxy has an asymmetric cost structure. Its historical representation grows linearly with T and must be read in full at every decode step, so its footprint sets both persistent GPU-memory usage and the bytes read by the memory-bound scan, making it a major contributor to scan latency. In contrast, query-side factors are applied only to the current query and do not grow with the context. This asymmetry motivates separating shared historical structure from query-specific interpretation: the former should be represented only once in the growing index, while the latter can remain head-specific outside it. Such a design removes redundant historical traffic without forcing different query heads to agree on token relevance. The following section formalizes this separation as joint attention-score reconstruction.

3.2 Joint Attention-Score Proxy

3.2.1 Joint Score-Space Formulation

Motivated by the cross-head score structure in Figure 1, we seek a single low-rank space that can reuse shared information across KV heads. The central challenge is to expose these cross-head correlations without changing which KV head each query attends to. We achieve this by placing all KV heads in a joint coordinate space while restricting each query to its assigned KV group.

For each historical token i, we concatenate its keys across all KV heads as $\bar{k}_i = [k_{1,i}; \ldots; k_{H_{kv},i}] \in \mathbb{R}^{H_{kv}d}$. Query q_h is embedded only into the block associated with $g(h)$, with all other blocks set

to zero. Denoting this block-sparse representation by $\bar{q}_h$, the construction preserves the original score exactly because $\bar{q}_h^\top \bar{k}_i = q_h^\top k_{g(h),i}$. We stack sampled joint keys as the rows of $\bar{K}$, and block-embedded queries across heads and positions as the rows of $\bar{Q}$. Their rows need not be position-wise paired because $\bar{Q}\bar{K}^\top$ represents all sampled query–history interactions.

JASP then learns a single rank-r factorization over this joint score space:

$$(U_q^*, U_k^*) = \arg\min_{U_q, U_k} \left\| \bar{Q}\bar{K}^\top - (\bar{Q}U_q)(\bar{K}U_k)^\top \right\|_F^2, \quad U_q, U_k \in \mathbb{R}^{H_{kv} d \times r}. \tag{2}$$

We omit the common $1/\sqrt{d}$ factor because it scales every target score equally. This objective directs the limited rank toward the score structure exposed by the query distribution. Although the learned factors span all KV heads, the block-sparse queries ensure that each target remains the interaction between a query and its assigned KV head. JASP can therefore reuse score-relevant structure across heads without changing the original attention connectivity.

3.2.2 Request-adaptive factorization

JASP fits the joint proxy once per request and layer using post-RoPE queries and keys sampled uniformly from dense prefill. Calibration uses the all-pairs score objective without a causal mask because the sampled rows estimate request-specific query and key score statistics rather than reproduce the prefill attention matrix. This request-specific calibration adapts the proxy to the score structure induced by the current prompt. Let n_q and n_k denote the numbers of sampled query and key rows. We compute their second moments as $\Sigma_q = \bar{Q}^\top \bar{Q}/n_q$ and $\Sigma_k = \bar{K}^\top \bar{K}/n_k$. Because each embedded query occupies only one KV-group block, Σ_q is block diagonal. In contrast, each row of $\bar{K}$ contains all KV heads of the same token, allowing Σ_k to retain their cross-head correlations.

Let $L\Lambda R^\top$ denote the singular value decomposition of $\Sigma_q^{1/2}\Sigma_k^{1/2}$. The leading r singular directions yield

$$U_q = \Sigma_q^{-1/2} L_r \Lambda_r^{1/2}, \qquad U_k = \Sigma_k^{-1/2} R_r \Lambda_r^{1/2}. \tag{3}$$

This construction gives the optimal rank-r approximation of the joint score matrix in Eq. 2. Separate query and key factors account for their different second-order statistics without requiring them to share the same projection. The solution requires only one truncated decomposition rather than iterative optimization. In practice, we use regularized inverse square roots for numerical stability; the complete derivation and calibration configuration are provided in Appendix A.

3.2.3 Shared index and query-specific scoring

After learning U_q and U_k, JASP materializes a shared historical index. Writing U_k as KV-group blocks $U_k^{(g)} \in \mathbb{R}^{d \times r}$, token i produces a single code $z_i = \bar{k}_i^\top U_k = \sum_g k_{g,i}^\top U_k^{(g)} \in \mathbb{R}^r$. This code combines information from all KV heads into one row rather than separate per-head histories. Each row is quantized to int8 with one scale and stored on the GPU. During decoding, every new token contributes one additional code, while its original K/V is retained separately for subsequent retrieval.

We similarly partition U_q into KV-group blocks $U_q^{(g)} \in \mathbb{R}^{d \times r}$. Query head h forms $a_h = q_h^\top U_q^{(g(h))}$ and estimates token i's score as $\hat{s}_h(i) = a_h z_i^\top / \sqrt{d}$. Although heads in the same group use the same factor block, their distinct query vectors produce independent rankings. For a history of length T, the shared index stores Tr int8 elements and T scales per layer, independent of the number of KV heads; independent head-wise proxies instead scale with the sum of their per-head ranks. Our fused scorer reuses each shared-index tile across query heads while producing separate logits, reducing bandwidth-dominant proxy traffic without removing query-specific relevance. Implementation details are provided in Appendix A.4.

3.2.4 KV-group token selection

In GQA, multiple query heads consume the same K/V rows. Retrieving a separate token set for every query head would therefore duplicate KV traffic. JASP instead constructs one set for each KV group while preserving the relevance evidence of its individual query heads. Let $\mathcal{H}$ denote the valid proxy history. We first normalize each head's proxy logits and then aggregate their probability mass

within the group:

$$p_h(i) = \frac{\exp(\hat{s}_h(i))}{\sum_{j\in\mathcal{H}} \exp(\hat{s}_h(j))}, \quad u_g(i) = \sum_{h:g(h)=g} p_h(i), \qquad \mathcal{S}_g = \mathrm{TopK}_k(u_g; \mathcal{C}), \tag{4}$$

where $\mathrm{TopK}_k(u_g; \mathcal{C})$ returns the k positions in $\mathcal{C} \subseteq \mathcal{H}$ with the largest $u_g(i)$, and $\mathcal{C}$ excludes the fixed sink and local positions. Normalizing before aggregation prevents differences in logit scale from causing one query head to dominate the group. Under a fixed group-level budget, selecting the largest $u_g(i)$ retains the greatest total proxy attention mass across the group's query heads.

The final retrieval indices are the duplicate-free union of $\mathcal{S}_g$ and the fixed sink and local positions. The corresponding original K/V rows are retrieved once for the KV group, but every query head still computes its own softmax weights and value aggregation over that set. JASP therefore shares retrieval without sharing attention weights: approximation is confined to token selection, while the attention applied to the selected original K/V remains head-specific. For MHA, each KV group contains one query head, and the procedure reduces directly to per-head top-k selection.

3.3 Serving integration

3.3.1 GPU-driven retrieval of original K/V

After token selection, a conventional offloading pipeline transfers the selected indices to the CPU, gathers the corresponding K/V rows in host memory, and stages the compact result back to the GPU. These synchronization and staging operations extend the decode critical path. UVA offers a more direct alternative by mapping pinned host memory into the GPU address space, allowing a GPU kernel to fetch selected K/V rows without returning the indices to the CPU. However, UVA provides addressability rather than efficient access by itself. Sparse selection produces scattered host addresses, so direct UVA retrieval can incur frequent address translations and poorly utilized PCIe transactions, leaving much of the available link bandwidth unused.

JASP addresses these bottlenecks at both the memory and kernel levels. We allocate the CPU KV prefix in huge-page-preferred mapped memory, reducing translation pressure across the large host-resident cache. For each selected token, cooperative threads read the contiguous feature dimension of its K/V row using coalesced vector accesses, converting an irregular row selection into efficient row-wise transactions. Selected positions in the GPU tail are read directly from device memory. Both paths write the original model-precision K/V into the same compact buffer consumed by sparse attention. Thus, JASP retains the control-flow advantage of UVA while making the underlying host-memory accesses substantially more efficient. Allocation, fallback, and kernel-layout details are provided in Appendix B.

3.3.2 Capacity-balanced KV placement

Offloading commonly places all model-precision K/V in host memory, leaving the GPU with only the model and proxy. Once the proxy is compact, this can strand GPU memory while host capacity limits the serving batch. JASP instead partitions each request by token position: an older prefix resides in mapped host memory and a contiguous suffix remains on the GPU. Because the shared proxy covers the complete history, selection is independent of placement. New K/V rows append to the GPU suffix, and every original row has one physical location.

We determine the split from the available CPU and GPU capacities. Let S be the full KV size per request at the configured sequence horizon, ϵ its GPU-resident proxy and auxiliary state, and α the fraction of original KV assigned to the GPU. After reserving model and runtime memory, capacities C_{GPU} and C_{CPU} constrain a batch of size N_b by

$$N_b(\alpha S + \epsilon) \le C_{\mathrm{GPU}}, \qquad N_b(1-\alpha)S \le C_{\mathrm{CPU}}. \tag{5}$$

JASP derives a capacity-balanced target from these constraints, then uses a slightly smaller GPU fraction to accommodate K/V appended during decoding. This fraction is set once per deployment and fixed during serving. As host capacity grows, the solution assigns less KV to the GPU and approaches CPU-only placement; when host memory is scarce, it uses otherwise idle GPU capacity to admit more requests. The closed-form target, decode headroom, and complete accounting appear in Appendix C.

Table 1: **128K proxy quality**; per-layer MiB.

Method	Footprint↓	Mass↑	M/O↑	IoU↑	NRE↓
PQCache	**16**	0.653	0.720	0.127	0.471
Loki	64	0.735	0.810	0.124	0.437
Quest	32	0.767	0.846	0.135	0.349
ShadowKV	33	0.820	0.906	0.283	0.255
ParisKV	112	0.871	0.963	0.494	0.194
LRQK	256	0.886	0.980	0.529	0.178
JASP	**16**	**0.893**	**0.988**	**0.610**	**0.163**

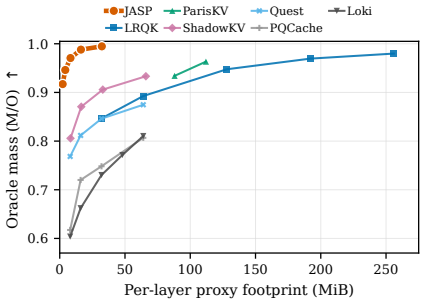


Figure 3: **Quality vs. proxy footprint.**

4 Experiments

Setup. We evaluate Llama-3.1-8B-Instruct (Grattafiori et al., 2024) on one NVIDIA RTX 6000D GPU and use 128 GiB of CPU memory. JASP keeps one quarter of the model-precision K/V on the GPU and the remainder on the CPU. We evaluate two categories of sparse baselines. Loki and Quest perform GPU-resident sparse attention, whereas PQCache, ShadowKV, ParisKV, and LRQK support KV offloading. FlashAttention-2 (Dao, 2024) provides the dense system reference. For each KV group, every sparse method dynamically retrieves 1% of the historical tokens, together with the same 16 sink tokens and 128-token local window after removing duplicate positions. Layers 0 and 1 use full attention in all quality and efficiency comparisons. Appendix D provides method-internal configurations, task protocols, hardware details, and supplementary experimental results.

4.1 Proxy Quality and Efficiency

Table 1 evaluates proxy quality and footprint at 128K using each method's recommended proxy configuration. Mass measures the attention probability captured by selected tokens, while M/O reports the fraction of oracle top-k mass recovered under the same budget. IoU measures overlap with the oracle selection, and NRE is the normalized error between sparse and dense attention outputs. At 1% retention, JASP captures 0.893 of dense attention mass, corresponding to 98.8% of the mass captured by exact-score Oracle under the same budget. Compared with LRQK, it improves IoU from 0.529 to 0.610 while reducing proxy footprint by 93.8%. At the same 16 MiB footprint as PQCache, JASP improves M/O by 37.2%. It also reduces NRE by 15.8% relative to ParisKV. JASP therefore combines the minimum proxy footprint with the highest proxy quality. To examine how proxy quality scales with footprint, we sweep each method's compression configuration in Figure 3. With only 8.5 MiB, JASP reaches 97.1% M/O, matching LRQK at 192 MiB and exceeding ParisKV at 112 MiB. At 32 MiB, JASP reaches 99.5% M/O versus 84.7% for LRQK. Across the sweep, JASP consistently achieves higher retrieval quality with a smaller proxy.

4.2 End-to-End Model Quality

We next evaluate how proxy quality translates into downstream performance. RULER tests long-context retrieval and reasoning through synthetic tasks. As shown in Table 2, JASP scores 75.9 across the complete RULER suite at 128K, matching the 75.9 achieved by exact-score Oracle selection under the same 1% budget. On the two largest gaps, MK3 and FWE, JASP scores 49.0 and 48.3, versus 51.0 and 47.8 for Oracle and 63.5 and 69.7 for dense attention. The similar losses under Oracle show that the retrieval budget, rather than proxy error alone, drives much of the remaining gap. Oracle maximizes retained attention mass but does not upper-bound downstream quality.

LongBench complements these synthetic tests with real-world language tasks that assess broader generation quality. Table 3 shows that JASP reaches 36.4, within 0.1 points of the 36.5 Oracle result and slightly above dense attention at 36.2. Its strongest results include MultiNews and GovReport, where it outperforms Loki by 0.7 and 0.4 points, respectively. To test cross-model quality generality, we further evaluate JASP with Qwen2.5-7B (Yang et al., 2025). Figure 4 shows its task-level gains and losses relative to dense attention. JASP averages 36.0 across the same task suite, within 0.1 points of dense attention at 36.1, demonstrating comparable quality beyond the primary Llama model.

Table 2: **Complete RULER results at 128K.**

Method	S1	S2	S3	MK1	MK2	MK3	MV	MQ	VT	FWE	CWE	QA1	QA2	Avg.
Dense	100.0	97.0	99.5	96.0	88.5	63.5	99.5	100.0	98.0	69.7	0.5	72.0	41.5	78.9
Oracle	100.0	96.0	99.5	95.5	86.5	51.0	99.5	100.0	98.5	47.8	0.5	71.5	41.0	75.9
PQCache	4.5	33.0	3.5	31.0	26.0	1.5	18.0	27.5	26.5	51.5	0.5	62.5	34.0	24.6
Loki	100.0	95.5	99.0	96.0	83.0	20.0	100.0	100.0	98.5	57.0	0.5	73.0	43.0	74.3
Quest	100.0	96.5	95.0	95.0	62.5	3.0	97.5	99.5	99.0	68.2	0.5	69.5	38.0	71.1
ShadowKV	100.0	95.0	99.5	96.5	71.0	23.5	97.0	100.0	90.0	62.0	0.5	69.0	40.5	72.7
ParisKV	100.0	96.0	99.5	95.5	85.5	35.5	100.0	99.5	99.0	50.8	0.5	71.5	42.0	75.0
LRQK	100.0	96.0	99.5	96.0	86.5	47.5	100.0	100.0	99.0	43.7	0.5	71.5	41.5	75.5
JASP	99.0	96.0	99.5	96.0	86.5	49.0	99.5	100.0	99.0	48.3	1.0	72.0	41.5	75.9

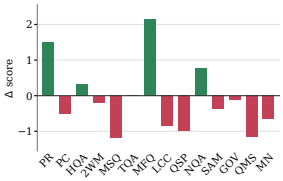


Figure 4: **Cross-model quality.** Per-task difference from dense.

Table 3: **LongBench quality.**

Method	PR	PC	HQA	2WM	MSQ	TQA	MFQ	LCC	QSP	NQA	SAM	GOV	QMS	MN	Avg.
Dense	98.0	7.2	16.7	16.4	11.3	91.7	27.4	64.6	13.1	32.0	43.8	34.6	23.4	26.8	36.2
Oracle	96.9	8.0	16.4	16.8	12.2	91.8	29.3	64.3	12.8	31.9	43.9	36.5	22.6	27.6	36.5
PQCache	95.8	8.7	17.0	15.2	10.5	91.2	25.4	62.8	11.0	30.7	43.0	30.9	22.0	24.5	34.9
Loki	96.6	7.9	18.3	16.8	12.9	91.9	29.2	64.3	12.1	31.3	43.6	36.4	21.6	27.1	36.4
Quest	93.5	9.1	15.8	14.7	10.1	92.0	24.5	62.5	9.9	31.1	41.5	26.6	22.4	21.6	33.9
ShadowKV	92.8	9.2	17.1	15.7	10.8	91.8	26.1	60.7	12.1	31.7	41.9	32.6	22.0	22.4	34.8
ParisKV	95.5	7.8	17.8	16.4	12.5	91.8	28.2	64.1	12.4	31.6	42.9	32.6	21.6	26.0	35.8
LRQK	96.0	7.9	16.7	16.3	11.7	91.6	28.4	64.2	12.7	31.1	43.9	35.4	22.3	23.9	35.9
JASP	96.1	8.5	16.7	16.5	12.7	91.8	29.4	64.1	13.0	31.6	43.1	36.8	22.0	27.8	36.4

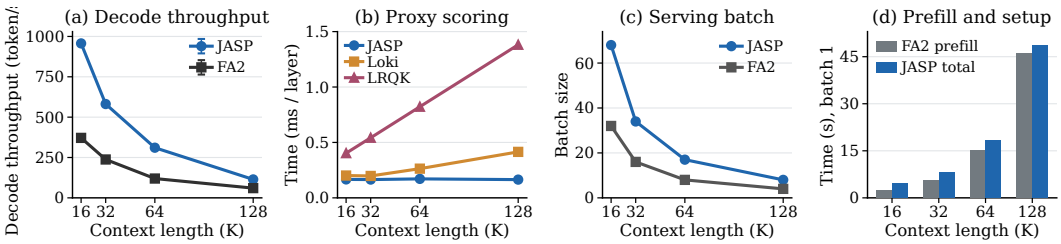


Figure 5: **Serving efficiency.** (a) Decode throughput, (b) per-layer proxy scoring time, (c) serving batch, and (d) batch-one setup cost.

4.3 Serving Efficiency

We evaluate serving throughput across 16K–128K contexts on the same GPU with 128 GiB of CPU memory. Each point averages three independent runs, with less than 0.1% relative standard deviation. Under saturated serving, Figure 5a shows that JASP delivers 1.89–2.60× the aggregate decode throughput of FlashAttention-2. At 128K, JASP reaches 114.1 tokens/s, compared with 60.3 tokens/s for FlashAttention-2. This gain comes from three complementary advantages. JASP's joint index reduces redundant per-head proxy traffic, allowing 128K proxy scoring to complete in 0.165 ms per layer, 2.5× and 8.4× faster than Loki and LRQK in Figure 5b. Its optimized UVA gather efficiently retrieves the selected KV rows, while capacity-balanced placement supports batches of 68, 34, 17, and 8 as context grows, approximately twice those of the dense implementations in Figure 5c. Together, these advantages produce JASP's throughput gains. We also measure proxy-construction overhead during prefill. Figure 5d shows that JASP takes 48.8 seconds at 128K, only about 6% longer than FlashAttention-2's 46.0 seconds. Appendix G.2 further compares JASP with other sparse-offloading methods.

To test whether the serving gains transfer across model architectures, we additionally evaluate saturated serving on Qwen2.5-7B. Its native context window limits this experiment to 32K, so we measure 8K–32K contexts. JASP achieves 1.68–1.92× the decode throughput of FlashAttention-2 while admitting more than twice as many concurrent requests throughout this range. These results show that JASP's compact proxy and offloading design transfer across model families. Detailed results are reported in Appendix E.2.

Table 4: **Equal-footprint mechanism ablations.**

	Design			Reconstruction error ↓		Retrieval ↑	
Variant	Objective	Key index	Query-specific	Key	Score	M/O	IoU
Per-head key	Key	Per-head	✓	0.578	0.119	0.931	0.324
Per-head score	Score	Per-head	✓	0.591	0.099	0.959	0.428
Block-diagonal score	Score	Block-diag.	✓	0.465	0.097	0.962	0.435
Group-mean Q	Score	Joint	✗	0.350	0.125	0.979	0.549
JASP	Score	Joint	✓	0.350	**0.053**	**0.988**	**0.610**

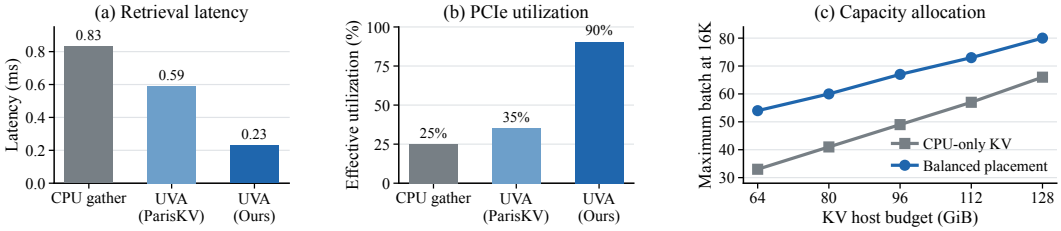


Figure 6: **System ablations.** (a) Retrieval latency, (b) effective PCIe utilization, and (c) maximum 16K batch across KV host budgets.

4.4 Core Mechanism Analysis

Table 4 isolates JASP's design choices at a fixed historical proxy footprint. For otherwise identical per-head proxies, replacing key reconstruction with score reconstruction lowers score error from 0.119 to 0.099, raises M/O from 0.931 to 0.959, and raises IoU from 0.324 to 0.428. Aligning the proxy with attention scores improves token ranking over key reconstruction.

With the score objective and index size fixed, global rank allocation under a block-diagonal key structure offers only small gains over the per-head control. JASP's joint index instead lowers score error from 0.097 to 0.053 and raises IoU from 0.435 to 0.610, isolating the value of cross-head structure rather than rank pooling alone. Averaging query heads within each KV group before scoring increases score error to 0.125 and reduces IoU to 0.549. JASP thus shares the history index while preserving query-specific ranking evidence.

4.5 System Ablations

Figure 6a compares retrieval of identical model-precision KV rows at 128K and 1% retention. CPU gather with host-to-device copying takes 0.83 ms, standard UVA takes 0.59 ms, and our optimized UVA takes 0.23 ms. These paths respectively reach 25.0%, 35.2%, and 90.3% of the platform's 24.02 GiB/s saturated UVA bandwidth in Figure 6b. For the same payload, our UVA path is 2.6× as fast as standard UVA through more effective PCIe bandwidth utilization.

Figure 6c isolates the capacity benefit of balanced placement at 16K, where larger batches make the difference clearer. For each host-memory budget, the offline planner assigns KV storage across CPU and GPU to maximize the admissible batch. This raises capacity from 33 to 54 requests at 64 GiB and from 66 to 80 at 128 GiB. Gains are largest under scarce host memory, as otherwise idle GPU capacity stores part of the KV cache. As host memory grows, the placement shifts toward CPU-only storage. Balancing both memory pools thus increases serving capacity without changing token selection.

5 Conclusion

JASP exploits shared query-conditioned score structure across KV heads for compact, accurate retrieval. Efficient KV access and capacity-balanced placement enable 1.89–2.60× FlashAttention-2's aggregate decode throughput under saturated serving at 16K–128K, while scoring 75.9 on the complete 128K RULER suite and matching dense attention on LongBench.

AI USE STATEMENT

In this work, we used generative AI tools to assist with grammar correction and language polishing, figure preparation, manuscript typesetting, and code debugging. The authors reviewed all AI-assisted outputs and take responsibility for the final content of this work.

REFERENCES

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

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, and Sumit Sanghai. GQA: Training generalized multi-query transformer models from multi-head checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 4895–4901. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.emnlp-main.298.

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. LongBench: A bilingual, multitask benchmark for long context understanding. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3119–3137. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.172.

Zhuoming Chen, Ranajoy Sadhukhan, Zihao Ye, Yang Zhou, Jianyu Zhang, Niklas Nolte, Yuandong Tian, Matthijs Douze, Leon Bottou, Zhihao Jia, and Beidi Chen. MagicPIG: LSH sampling for efficient LLM generation. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=ALzTQUgW8a.

Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=mZn2Xyh9Ec.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*, volume 35, pp. 16344–16359, 2022. doi: 10.52202/068431-1189.

Aaron Grattafiori et al. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024. URL https://arxiv.org/abs/2407.21783.

Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W. Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. KVQuant: Towards 10 million context length LLM inference with KV cache quantization. In *Advances in Neural Information Processing Systems*, volume 37, pp. 1270–1303, 2024. doi: 10.52202/079017-0040.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. RULER: What's the real context size of your long-context language models? In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=kIoBbc76Sy.

Shibo Jie, Yehui Tang, Kai Han, Zhi-Hong Deng, and Jing Han. SpeCache: Speculative key-value caching for efficient generation of LLMs. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 27917–27928. PMLR, 2025.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023. doi: 10.1145/3600006.3613165.

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

Haoyang Li, Yiming Li, Anxin Tian, Tianhao Tang, Zhanchao Xu, Xuejia Chen, Nicole Hu, Wei Dong, Qing Li, and Lei Chen. A survey on large language model acceleration based on KV cache management. *arXiv preprint arXiv:2412.19442*, 2024a.

Tenghui Li, Guoxu Zhou, Xuyang Zhao, Yuning Qiu, and Qibin Zhao. Efficient low rank attention for long-context inference in large language models. In *Advances in Neural Information Processing Systems*, volume 38, pp. 38421–38447, 2025. doi: 10.52202/085713-1284.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. SnapKV: LLM knows what you are looking for before generation. In *Advances in Neural Information Processing Systems*, volume 37, pp. 22947–22970, 2024b. doi: 10.52202/079017-0722.

Akide Liu, Jing Liu, Zizheng Pan, Yefei He, Gholamreza Haffari, and Bohan Zhuang. MiniCache: KV cache compression in depth dimension for large language models. In *Advances in Neural Information Processing Systems*, volume 37, pp. 139997–140031, 2024a. doi: 10.52202/079017-4443.

Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, Michael Maire, Henry Hoffmann, Ari Holtzman, and Junchen Jiang. CacheGen: KV cache compression and streaming for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pp. 38–56. ACM, 2024b. doi: 10.1145/3651890.3672274.

Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. Scissorhands: Exploiting the persistence of importance hypothesis for LLM KV cache compression at test time. In *Advances in Neural Information Processing Systems*, volume 36, pp. 52342–52364, 2023. doi: 10.52202/075280-2279.

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. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 32332–32344. PMLR, 2024c.

Junlin Mu, Hantao Huang, Jihang Zhang, Minghui Yu, Tao Wang, and Yidong Li. SALS: Sparse attention in latent space for KV cache compression. In *Advances in Neural Information Processing Systems*, volume 38, pp. 384–403. Curran Associates, Inc., 2025. doi: 10.52202/085713-0013. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/00a0ebcad584c59dbc439c2af8793638-Paper-Conference.pdf.

Yanlin Qi, Xinhang Chen, Huiqiang Jiang, Qitong Wang, Botao Peng, and Themis Palpanas. ParisKV: Fast and drift-robust KV-cache retrieval for long-context LLMs. *arXiv preprint arXiv:2602.07721*, 2026.

Luka Ribar, Ivan Chelombiev, Luke Hudlass-Galley, Charlie Blake, Carlo Luschi, and Douglas Orr. SparQ Attention: Bandwidth-efficient LLM inference. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 42558–42583. PMLR, 2024.

Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. FlexGen: High-throughput generative inference of large language models with a single GPU. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 31094–31116. PMLR, 2023.

Prajwal Singhania, Siddharth Singh, Shwai He, Soheil Feizi, and Abhinav Bhatele. Loki: Low-rank keys for efficient sparse attention. In *Advances in Neural Information Processing Systems*, volume 37, pp. 16692–16723, 2024. doi: 10.52202/079017-0532.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. RoFormer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024. doi: 10.1016/j.neucom.2023.127063.

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. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 57355–57373. PMLR, 2025.

Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. QUEST: Query-aware sparsity for efficient long-context LLM inference. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 47901–47911. PMLR, 2024.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=NG7sS51zVF.

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.

An Yang et al. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2025. URL https://arxiv.org/abs/2412.15115.

Dongjie Yang, Xiaodong Han, Yan Gao, Yao Hu, Shilin Zhang, and Hai Zhao. PyramidInfer: Pyramid KV cache compression for high-throughput LLM inference. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 3258–3270. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.findings-acl.195.

Hailin Zhang, Xiaodong Ji, Yilin Chen, Fangcheng Fu, Xupeng Miao, Xiaonan Nie, Weipeng Chen, and Bin Cui. PQCache: Product quantization-based KVCache for long context LLM inference. *Proceedings of the ACM on Management of Data*, 3(3):1–30, 2025. doi: 10.1145/3725338.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang Wang, and Beidi Chen. H2O: Heavy-hitter oracle for efficient generative inference of large language models. In *Advances in Neural Information Processing Systems*, volume 36, pp. 34661–34710, 2023. doi: 10.52202/075280-1506.

Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 193–210. USENIX Association, 2024.

A JOINT ATTENTION-SCORE PROXY: DERIVATION AND FULL INFERENCE

This appendix derives the factorization used by JASP, specifies how its request-adaptive proxy is calibrated and stored, and then follows one sparse attention layer through prefill and autoregressive decoding. All queries and keys below are post-RoPE unless stated otherwise.

A.1 JOINT EMBEDDING AND OBJECTIVE

Let $w = H_{kv}d$. For token i, the joint key vector is $\bar{k}_i = [k_{1,i}; \ldots; k_{H_{kv},i}] \in \mathbb{R}^w$. A query head h is embedded into the block belonging to its KV group $g(h)$:

$$\bar{q}_h = [0; \ldots; q_h; \ldots; 0] \in \mathbb{R}^w.$$

The embedding preserves the uncompressed score because $\bar{q}_h^\top \bar{k}_i = q_h^\top k_{g(h),i}$. Stacking sampled embedded queries gives $\bar{Q} \in \mathbb{R}^{n_q \times w}$, while stacking sampled joint keys gives $\bar{K} \in \mathbb{R}^{n_k \times w}$. Their rows need not be paired by position: $\bar{Q}\bar{K}^\top$ contains every sampled query–history interaction. With $M = U_q U_k^\top$, the main-text objective is equivalently

$$\min_{\mathrm{rank}(M) \le r} \|\bar{Q}(I - M)\bar{K}^\top\|_F^2.$$

This all-pairs score objective is therefore determined by the marginal query and key second moments, rather than a paired-position cross-covariance or a CCA objective.

A.2 CLOSED-FORM SOLUTION

Let $\Sigma_q = \bar{Q}^\top \bar{Q}/n_q$ and $\Sigma_k = \bar{K}^\top \bar{K}/n_k$. Then

$$\frac{1}{n_q n_k}\|\bar{Q}(I - M)\bar{K}^\top\|_F^2 = \|\Sigma_q^{1/2}(I - M)\Sigma_k^{1/2}\|_F^2.$$

The embedded-query construction makes Σ_q block diagonal, with one $d \times d$ block per KV group, whereas Σ_k is formed in the full joint key space and retains cross-head correlations. If

$$\Sigma_q^{1/2}\Sigma_k^{1/2} = L\Lambda R^\top,$$

then the Eckart–Young–Mirsky theorem gives the optimal rank-r map

$$M_r = \Sigma_q^{-1/2} L_r \Lambda_r R_r^\top \Sigma_k^{-1/2}.$$

The implementation splits M_r symmetrically into $U_q = \Sigma_q^{-1/2} L_r \Lambda_r^{1/2}$ and $U_k = \Sigma_k^{-1/2} R_r \Lambda_r^{1/2}$. It symmetrizes both moments, computes their eigendecompositions, and floors each eigenvalue at 10^{-6} times the largest one before taking square roots and inverse square roots. This produces stable regularized inverse square roots for near-singular calibration moments. The closed form is exact for positive-definite moments; eigenvalue flooring solves the corresponding regularized numerical problem.

With $w = H_{kv}d$, forming the full key moment costs $O(n_k w^2)$, while the block-sparse query moment costs $O(n_q d^2)$; the dense factorization costs $O(w^3)$ time and $O(w^2)$ workspace. This cost is paid once per request and layer during prefill, whereas decoding scans only the rank-r history code.

The returned factors are reshaped to $[H_{kv}, d, r]$. This shape does not mean that each KV head receives an independent rank-r code. Instead, the stored token row is

$$z_i = \bar{k}_i^\top U_k = \sum_{g=1}^{H_{kv}} k_{g,i}^\top U_k^{(g)} \in \mathbb{R}^r,$$

which is one code shared across all KV heads. The query block $U_q^{(g)}$ is shared by the query heads belonging to KV group g; their projected queries and logits remain different because their input vectors are different.

A.3 Calibration and proxy representation

The factors are request-adaptive. During dense prefill, each sparse layer samples the same set of at most 8192 uniformly spaced positions from its post-RoPE Q and K. The factorization is fitted once from these samples and then held fixed throughout decoding. All prompt keys are projected to shared codes, including positions not used for fitting.

Each code is quantized independently to signed int8. For an unquantized code z_i, the implementation stores $c_i = \mathrm{round}(z_i/\gamma_i)$ and one FP16 scale $\gamma_i = \max_j |z_{i,j}|/127$, with clamping for the all-zero case. The current query projection is quantized analogously at runtime, and the int8 dot product is rescaled before softmax. Quantization therefore changes only the numerical representation of the learned proxy, not its latent width or token budget. For the optimized Llama-3.1-8B path, $H_q = 32$, $H_{kv} = 8$, $d = 128$, and $r = 128$; the shared history has shape $[B, 1, T, 128]$. The mathematical construction admits other head counts, and the fused scorer uses the deployed model's KV-head count and head dimension. This supports both the Llama and Qwen configurations used in our experiments without changing the proxy objective.

A.4 Complete inference procedure

The following sequence describes one sparse layer in the native offloaded path. Layers configured for full attention bypass proxy construction, selection, and retrieval.

1. **Dense prefill and fit.** Run causal full attention, collect post-RoPE Q/K samples, fit U_q, U_k, encode every prompt position into one shared int8 history, and partition the original model-precision K/V (FP16 in our implementation) into a CPU prefix and GPU tail.
2. **Append current state.** At a decode step, compute post-RoPE Q/K and the model-precision value for the new token. Append the original K/V to the monotonic GPU tail. A fused kernel projects the new K through all $U_k^{(g)}$ blocks, sums the contributions, quantizes the resulting shared row, and appends it to the proxy. The fitted factors remain unchanged.
3. **Score the history.** For each query head, apply the factor block of its KV group, quantize the projected current query, and compute rescaled int8 dot products against the shared history. The scorer rounds its allocation to a 256-position bucket and masks unused rows with negative infinity before softmax.
4. **Form one set per KV group.** Apply softmax independently to every query head's logits, sum the probabilities within each GQA group, and select dynamic top-k positions only from the eligible interval. This interval excludes sink and local positions, so concatenating the dynamic result with the fixed positions produces a duplicate-free keep set.
5. **Retrieve and attend.** A GPU kernel interprets every absolute index as either a GPU-tail row or a CPU-prefix UVA row and emits compact H_{kv}-wide model-precision K/V. Grouped-query FlashAttention computes standard attention over this selected set, followed by the model's unchanged output projection.

The proxy is used only to select token positions. Attention is always computed from the corresponding original model-precision K/V rather than reconstructed proxy states.

B Efficient Retrieval Implementation Details

B.1 UVA mapped pinned memory

A naive CPU-offloading implementation performs a GPU–CPU round trip at every layer: the GPU computes the selected indices, transfers them to the CPU, the CPU gathers the corresponding KV rows, and the gathered block is copied back to the GPU. This path introduces synchronization on the decode critical path and makes the CPU participate in every layer's retrieval.

We avoid this by storing the CPU-resident KV cache in mapped pinned memory and exposing the same allocation to GPU kernels through CUDA Unified Virtual Addressing (UVA). The CPU view is used to populate the KV cache after prefill, while the CUDA view is consumed by the retrieval kernel during decoding. Since the selected indices remain on the GPU, the gather kernel can directly

issue loads to the CPU-resident KV buffer through the UVA pointer. This removes the index transfer, CPU-side gather, and explicit host-to-device copy from the per-layer retrieval path.

B.2 Transparent HugePages for IOMMU translation

Although UVA removes CPU-side gather, it exposes a different bottleneck: address translation for GPU-initiated accesses to CPU memory. Under the default 4 KB page size, a long-context KV buffer spans many page-table entries. Random row access over this buffer can exceed IOMMU TLB capacity, causing frequent TLB misses and page-table walks. These misses add PCIe round trips that are not accounted for by the selected KV volume alone.

We reduce this translation footprint by backing the mapped KV allocation with 2 MiB Transparent HugePages whenever possible. In our implementation, the CPU buffer is allocated with `mmap`, marked with `MADV_HUGEPAGE`, faulted in to encourage huge-page promotion, and then registered with CUDA using mapped pinned-memory registration. The device pointer returned by CUDA is used by the retrieval kernel exactly like a regular UVA pointer. When huge pages are honored, the same KV buffer requires up to $512\times$ fewer page translations than with 4 KiB pages.

Transparent HugePages are opportunistic. The runtime checks the system THP policy and selects the `mmap`/`MADV_HUGEPAGE` backend when the active mode is `always` or `madvise`; otherwise it uses ordinary mapped pinned allocation. It does not inspect every allocation to prove physical huge-page promotion. A failed `madvise` can therefore continue with regular pages, while an allocation or CUDA-registration failure is reported. These cases affect performance but not tensor values.

B.3 Row-coalesced gather kernel

General-purpose gather operators are not specialized for KV-cache retrieval from UVA-mapped memory. The access pattern is irregular across token positions but contiguous within each selected KV row. We exploit this structure with a row-coalesced CUDA kernel.

Each warp is assigned to one selected row, indexed by a batch item, KV head, and token position. Threads within the warp cooperatively load contiguous elements along that row's feature dimension using vectorized memory operations. The evaluated d=128 path uses compile-time specialization and unrolling; other dimensions use a generic path. This design turns random token selection into contiguous row reads and improves effective PCIe bandwidth.

B.4 Correctness

UVA changes the access path but reads the same CPU-resident KV rows. HugePages change only the preferred backing-page size, and row coalescing changes only the memory-access schedule. The gather output is the selected uncompressed FP16 K/V tensor regardless of whether a row resides in the CPU prefix or GPU tail. These mechanisms therefore do not alter the selected token set or the attention computed over that set.

C Capacity-Balanced Placement Details

Balancing the two main-text capacity constraints gives the continuous interior solution

$$N_b^* = \frac{C_{\mathrm{GPU}} + C_{\mathrm{CPU}}}{S + \epsilon}, \qquad \alpha^* = \frac{C_{\mathrm{GPU}} S - C_{\mathrm{CPU}} \epsilon}{S(C_{\mathrm{GPU}} + C_{\mathrm{CPU}})}.$$

Thus the interior solution is feasible when $0 \le \alpha^* \le 1$. The lower bound requires $C_{\mathrm{GPU}} S \ge C_{\mathrm{CPU}} \epsilon$, meaning that the GPU budget can hold the shared index and still participate in full-KV storage.

If $\alpha^* < 0$, the optimum lies at $\alpha = 0$: the full KV cache is placed on CPU and the system reduces to conventional CPU offloading, with batch size

$$N_{b,\alpha=0} = \min\left(\frac{C_{\mathrm{GPU}}}{\epsilon}, \frac{C_{\mathrm{CPU}}}{S}\right).$$

If $\alpha^* > 1$, the optimum lies at $\alpha = 1$: the full KV cache is placed on GPU, and the batch size is

$$N_{b,\alpha=1} = \frac{C_{\mathrm{GPU}}}{S + \epsilon}.$$

The capacity-balanced target is obtained by clipping the interior solution to the feasible interval:

$$\alpha_{\text{target}} = \text{clip}(\alpha^*, 0, 1).$$

The runtime uses a tail fraction no larger than α_{target} after reserving GPU capacity for K/V appended during decoding. This fraction is computed offline and passed to the runtime; the runtime neither queries instantaneous free memory nor rebalances requests online.

The dominant per-request terms for L sparse layers, context length N, and shared index rank r are

$$\begin{aligned} S_{\text{cpu}}(\alpha) &= 2LH_{kv}(1-\alpha)Nd\, b_{\text{fp16}}, \\ S_{\text{tail}}(\alpha) &= 2LH_{kv}\alpha Nd\, b_{\text{fp16}}, \\ S_{\text{proxy}} &= LN(b_{\text{int8}}r + b_{\text{scale}}). \end{aligned}$$

Here b_{fp16}, b_{int8}, and b_{scale} denote the bytes used by one FP16 value, one int8 code value, and one row scale, respectively. Complete admission accounting also includes U_q/U_k, decode headroom in the tail, top-k indices, scorer workspaces, model weights, activations, and allocator reserve. Importantly, the shared proxy term has no H_{kv} multiplier.

C.1 Monotonic GPU-tail management during decoding

After prefill, the CPU prefix is frozen and the GPU tail contains the newest $\lfloor \alpha N \rfloor$ prompt positions. Each new token's post-RoPE key rows and model-precision value rows append only to the GPU tail, avoiding CPU writes on the decode critical path. The configured prompt-tail fraction remains fixed, while the physical tail grows within the reserved generation headroom. The tail is monotonic rather than a sliding window: because the CPU does not mirror these rows, discarding an old tail row would lose original K/V. If the reserved allocation is exhausted, the runtime expands it while preserving every absolute-position mapping. The planned generation budget must therefore be included when selecting the offline split.

D Experimental Protocol

All experiments use one NVIDIA RTX 6000D GPU with 128 GiB of CPU memory, connected through PCIe Gen4 $\times$16. Quality tables report deterministic greedy-decoding scores on fixed evaluation sets. Unless a microbenchmark states otherwise, system throughput uses 16 warm-up steps and one 128-step timed decode run.

D.1 Controlled baseline configurations

Table 5 lists the method-internal settings. Unified controls (dynamic rows per KV group, sink/local positions, layers, model, and context) override a paper's native token budget, while internal compression parameters follow the paper. No GQA method receives one full budget per query head: query-head scores are reduced to one exact-size set for the associated KV group before K/V retrieval. These adapters preserve each method's proxy and native scoring rule, changing only how its candidates are mapped to the common group-level budget.

At the 128K setting, $T = 131{,}072$ prompt tokens gives $k_{\text{dyn}} = 1{,}310$ rows per KV group. Because the candidate interval excludes the 16 sink and 128 local positions, their duplicate-free union contains exactly 1,454 rows, or 1.109% of the history. We consistently call 1% the *dynamic* retention ratio rather than the effective total ratio.

Footprint includes every history-dependent object read by the selector at each step: stored codes and row scales, page summaries, and method-specific auxiliary indexes. Fixed-size projection matrices and query-only workspaces are excluded because they do not scale with context length.

D.2 Metrics and proxy-analysis protocols

Let D and D^* denote the selected and oracle dynamic sets, and let F be the common duplicate-free sink/local set. Mass is the full-attention probability assigned to $S = D \cup F$, M/O divides this

Table 5: Method-internal configurations in controlled quality evaluation.

Method	Internal configuration	Unified-budget adaptation
PQCache	m=2, b=6	dynamic rows fixed per KV group
Loki	rank 32, pre-RoPE transform	dynamic rows fixed per KV group
Quest	page size 16	pages ranked natively; final page truncated by recency
ShadowKV	chunk 8, rank 160	published query/outlier split within total budget
ParisKV	4-bit summaries, 16 subspaces	adaptive candidates; common dynamic budget
LRQK	rank 32, two updates, tolerance 10^{-2}	query distributions summed per group
JASP	joint rank 128, row-wise int8	one group-specific set from shared history

Table 6: Compression sweeps used for Figure 3. Token selection remains fixed at 1% dynamic retention with the common sink/local set; only the history-dependent proxy configuration changes.

Method	Swept parameter	Fixed configuration
PQCache	subspaces per head: 1, 2, 4, 8	6-bit subvector codes
Loki	per-head rank: 4, 8, 16, 24, 32	pre-RoPE projection
Quest	page size: 8, 16, 32, 64	native page summaries
ShadowKV	chunk size: 4, 8, 16, 32	rank 160; published query/outlier split
ParisKV	released m=8 and m=16 codebooks	three 4-bit summary configurations
LRQK	per-head rank: 4, 8, 16, 24, 32	two updates; tolerance 10^{-2}
JASP	joint rank: 16, 32, 64, 128, 256	row-wise int8 shared code

value by the mass of $S^* = D^* \cup F$, and IoU compares only the equal-cardinality dynamic sets as $|D \cap D^*|/|D \cup D^*|$. Attention-output NRE is

$$\mathrm{NRE} = \frac{\|O_S - O_{\mathrm{dense}}\|_F}{\|O_{\mathrm{dense}}\|_F},$$

where O_S uses exact attention over the selected original model-precision FP16 K/V rows and O_{dense} uses the complete history. M/O therefore measures closeness to exact-score selection at the same budget, not the fraction of total dense attention mass retained. We compute each metric on held-out decoding queries, average within each layer–sequence capture, and then average captures. Table 1 and Figure 8b aggregate ten 128K prompts over all 30 sparse layers, yielding 300 layer–sequence captures per configuration. The footprint sweeps in Figures 3 and 8a use ten prompts and all 30 sparse layers, yielding 300 captures per configuration. Both protocols use 16 held-out autoregressive queries per capture. Prompt-cluster bootstrap intervals confirm JASP's gains over LRQK: the 95% intervals are $[0.0070, 0.0100]$ for M/O, $[0.0748, 0.0926]$ for IoU, and $[-0.0170, -0.0133]$ for NRE.

Table 4 uses ten 128K sequences and all deployed sparse layers, giving 300 layer–sequence captures. Every variant receives the same calibration data and 16 held-out autoregressive queries per capture. The history-dependent footprint is fixed to 128 int8 code dimensions per token plus one FP16 row scale. Per-head variants use rank 16 for each of eight KV heads, whereas block-diagonal and joint variants use one rank-128 code. Fixed-size projection factors are excluded because they do not grow with the history.

Key reconstruction is evaluated on key rows disjoint from calibration. For a stored code Z, we fit the optimal linear decoder D^* on calibration rows and report $\|K_{\mathrm{test}} - Z_{\mathrm{test}} D^*\|_F / \|K_{\mathrm{test}}\|_F$. Per-head variants fit one decoder per KV head, while joint variants decode the concatenated keys from their shared code. Score error is the relative Frobenius error between exact and proxy query–key logits on the held-out decode queries. The Group-mean-Q control averages query heads within each KV group before scoring and shares the resulting proxy scores across those heads; all other variants retain each query head's score evidence. The table reports these raw relative-error values directly, without normalization to any method. Retrieval uses the common sink and local set, identical candidate positions, and an exact 1% dynamic budget.

Table 7: Key versus score fitting with the same joint proxy footprint.

Variant	Key err.↓	Score err.↓	M/O↑	IoU↑
Joint key	0.324	0.069	0.978	0.514
JASP	0.350	0.053	0.988	0.610

E Supplementary Proxy Analyses

E.1 Joint key reconstruction control

To isolate the role of score alignment within a shared representation, we compare JASP with a joint key-reconstruction proxy on the same 300 captures used in Table 4. The control concatenates the keys across KV heads and fits a rank-128 uncentered PCA basis U on the same calibration rows. It stores the joint code $z_i = U^\top \bar{k}_i$ and scores each query using $(U^\top \bar{q}_h)^\top z_i/\sqrt{d}$, where $\bar{q}_h$ is the zero-padded query defined in Appendix A. Both methods store 128 INT8 values and one FP16 scale per historical token, retain query-specific scoring, and use identical held-out queries and retrieval budgets. Thus, this comparison changes the fitting objective without changing the shared layout or historical storage cost.

Table 7 shows that the joint key objective yields lower key-reconstruction error, whereas JASP lowers score error from 0.069 to 0.053 and raises IoU from 0.514 to 0.610. M/O also improves from 0.978 to 0.988. Thus, exploiting cross-head structure alone does not account for the full retrieval benefit: score alignment improves token selection even when both proxies share the same joint representation layout and storage budget.

E.2 Robustness and cross-model generalization

JASP is fitted once per request and sparse layer. Table 8 evaluates calibration sensitivity over 11 captures and 16 held-out autoregressive queries per capture. Uniform sampling nearly saturates at 2,048 positions: raising the count to 8,192 changes M/O from 0.9899 to 0.9905 and IoU from 0.630 to 0.637. Coverage matters more than count. Random sampling matches uniform sampling, whereas a contiguous prefix misses query-relevant directions and reduces M/O to 0.827. Factor fitting takes 71–79 ms per layer over this range. Table 9 tests whether the mechanism itself transfers beyond Llama by repeating the equal-footprint comparison on Qwen2.5-7B over 280 layer–prompt captures. Every variant stores one 128-byte INT8 code and one FP16 scale per historical token; the per-head codes are concatenated before row-wise quantization so their scale cost is identical. Relative to the stronger block-diagonal control, JASP reduces score error by 29.4%, raises M/O by 1.09 percentage points, and raises IoU by 9.85 percentage points. Together with the task-level LongBench results in Table 11, this shows that the benefit of sharing cross-head score structure generalizes across model families.

Table 8: Calibration robustness at 1% dynamic retention.

Axis	Setting	M/O↑	IoU↑
Positions	512 uniform	0.9869	0.595
	1,024 uniform	0.9892	0.619
	2,048 uniform	0.9899	0.630
	4,096 uniform	0.9903	0.634
	8,192 uniform	**0.9905**	**0.637**
Placement	2,048 prefix	0.8274	0.206
	2,048 middle	0.8887	0.334
	2,048 suffix	0.9814	0.582
	2,048 random	**0.9900**	**0.631**
	2,048 uniform	0.9899	0.630

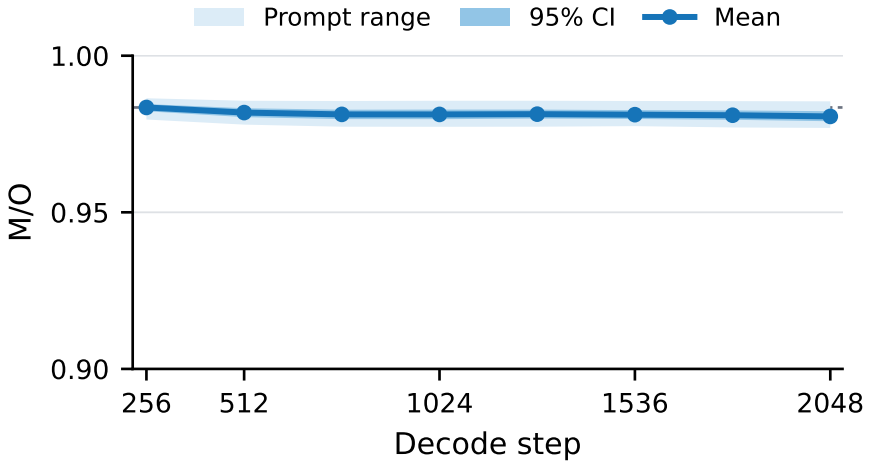


Figure 7: Long-generation proxy stability at 128K. JASP is fitted once during prefill and remains frozen for 2,048 autoregressive steps. The line shows the mean over ten prompts; bands show 95% bootstrap intervals and prompt ranges.

Table 11: Complete Qwen2.5-7B LongBench results. PR: Passage Retrieval; PC: Passage Count; HQA: HotpotQA; 2WM: 2WikiMultihopQA; MSQ: MuSiQue; TQA: TriviaQA; MFQ: MultiFieldQA; QSP: Qasper; NQA: NarrativeQA; SAM: SAMSum; GOV: GovReport; QMS: QMSum; MN: MultiNews.

Method	PR	PC	HQA	2WM	MSQ	TQA	MFQ	LCC	QSP	NQA	SAM	GOV	QMS	MN	Avg.
Dense	67.0	2.0	38.7	17.8	21.7	89.2	36.6	70.4	14.8	16.7	47.1	31.9	24.6	27.4	36.1
JASP	68.5	1.5	39.0	17.6	20.5	89.2	38.8	69.5	13.8	17.5	46.7	31.8	23.4	26.7	36.0

Table 9: Equal-footprint mechanism validation on Qwen2.5-7B.

Variant	Key err.↓	Score err.↓	M/O↑	IoU↑	NRE↓
Per-head score	0.398	0.075	0.980	0.571	0.242
Block-diagonal score	0.335	0.073	0.980	0.572	0.242
JASP	**0.289**	**0.051**	**0.991**	**0.670**	**0.231**

Table 10: Cross-model serving efficiency on Qwen2.5-7B.

	Serving batch		Decode throughput (token/s)		
Context	JASP	FA2	JASP	FA2	JASP / FA2
8K	333	148	2771.1	1653.7	1.68×
16K	177	74	1672.0	869.3	1.92×
32K	88	37	871.2	514.9	1.69×

Table 10 follows the main serving protocol: each method uses its admitted batch, followed by 16 warm-up steps and one 128-step timed decode run. JASP uses the same 1% retrieval budget, 25% GPU tail, and 128 GiB host memory as the primary experiment; FA2 retains physical per-request KV caches on the GPU. Qwen2.5-7B is evaluated through its native 32K context window.

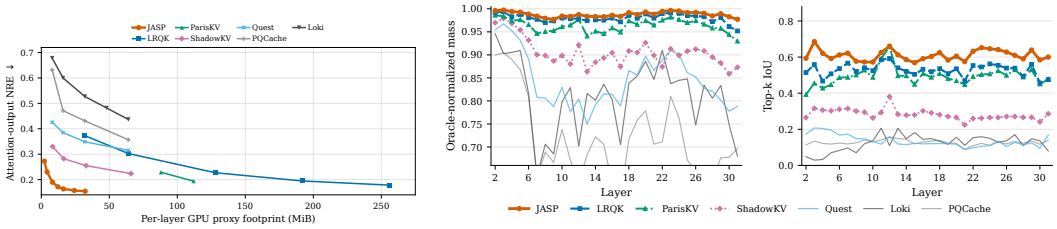
Figure 7 evaluates the frozen proxy over ten distinct prompts and all 32 layers. M/O remains high throughout long decoding, changing from 0.9835 over the first 256 steps to 0.9806 over the final 256 steps. This is a drop of only 0.29 percentage points, with a prompt-cluster bootstrap interval of $[0.16, 0.41]$ points. Thus, JASP preserves over 98% of oracle attention mass throughout 2,048 decode steps without refitting its proxy.

E.3 Cross-head attention-score similarity

We analyze Llama-3.1-8B-Instruct using ten sequences with 128K-token contexts and collect activations from all 32 layers, yielding 320 layer–sequence captures. For each capture, we evaluate 16 autoregressive decoding queries that are not used for proxy calibration. For every KV head, each historical token is represented by its raw post-RoPE attention scores under the four associated query heads across these decoding steps. We measure pairwise similarity using linear centered kernel alignment, which compares the token-wise geometry of two score representations while being insensitive to uniform scaling and orthogonal changes of basis. The heatmap in Figure 1 averages each pairwise similarity over all captures, while the layer-wise curve averages over KV-head pairs and sequences at each layer. All query samples and key positions used in this analysis are held out from JASP calibration.

E.4 Complete proxy-quality results

Figure 8 complements the main-text retrieval frontier with attention-output error and layer-wise proxy quality. JASP attains the lowest NRE throughout its footprint range while consistently preserving more oracle attention mass and recovering the oracle token set more accurately across sparse layers 2–31.



(a) Attention-output NRE across proxy footprints.

(b) Layer-wise M/O and IoU on sparse layers 2–31.

Figure 8: Supplementary proxy-quality results at 128K.

Table 12 gives the complete retention sweep. The fixed 16 sink tokens and 128-token local window are excluded from the dynamic ratio and added identically for every method.

Table 12: Proxy-quality sweep on sparse layers 2–31 at 128K context. The results use ten prompts at every retention point. Per-layer footprint is rounded to MiB.

	Cost	0.5%			1%			2%		
Method	Footprint	Mass	M/O	IoU	Mass	M/O	IoU	Mass	M/O	IoU
PQCache	16	0.610	0.697	0.089	0.653	0.720	0.127	0.706	0.755	0.174
Loki	64	0.696	0.795	0.105	0.735	0.810	0.124	0.778	0.831	0.149
Quest	32	0.720	0.823	0.101	0.767	0.846	0.135	0.816	0.872	0.178
ShadowKV	33	0.766	0.878	0.244	0.820	0.906	0.283	0.863	0.923	0.324
ParisKV	112	0.846	0.970	0.518	0.871	0.963	0.494	0.892	0.954	0.451
LRQK	256	0.850	0.975	0.485	0.886	0.980	0.529	0.920	0.985	0.572
JASP	16	0.859	0.985	0.574	0.893	0.988	0.610	0.926	0.991	0.650

Table 12: Proxy-quality sweep (continued).

	Cost	5%			10%			20%		
Method	Footprint	Mass	M/O	IoU	Mass	M/O	IoU	Mass	M/O	IoU
PQCache	16	0.792	0.819	0.255	0.864	0.878	0.335	0.930	0.936	0.437
Loki	64	0.838	0.866	0.195	0.883	0.897	0.245	0.925	0.930	0.319
Quest	32	0.881	0.911	0.248	0.926	0.941	0.316	0.963	0.969	0.407
ShadowKV	33	0.917	0.948	0.388	0.952	0.967	0.447	0.978	0.984	0.524
ParisKV	112	0.906	0.937	0.359	0.909	0.924	0.272	0.949	0.954	0.351
LRQK	256	0.958	0.991	0.635	0.979	0.995	0.688	0.992	0.998	0.749
JASP	16	0.962	0.995	0.707	0.981	0.997	0.754	0.993	0.999	0.802

F Downstream Evaluation Details

For compactness, the main text reports benchmark subtasks using abbreviations. For RULER, S1–S3 and MK1–MK3 denote the three single-key and multi-key needle-in-a-haystack variants; MV and MQ denote multi-value and multi-query retrieval; VT denotes variable tracking; FWE and CWE denote frequent- and common-word extraction; and QA1–QA2 denote the two question-answering variants. For LongBench, PR denotes *PassageRetrieval-en*; PC denotes *PassageCount*; HQA denotes *HotpotQA*; 2WM denotes *2WikiMultihopQA*; MSQ denotes *MuSiQue*; TQA denotes *TriviaQA*; MFQ denotes *MultiFieldQA-en*; LCC denotes *LCC*; QSP denotes *Qasper*; NQA denotes *NarrativeQA*; SAM denotes *SAMSum*; GOV denotes *GovReport*; QMS denotes *QMSum*; MN denotes *MultiNews*; and Avg reports the average score across all listed tasks.

All downstream evaluations use the same fixed positions across sparse methods: the duplicate-free union of 16 sink tokens and a 128-token local window. Unless otherwise stated, the proxy additionally selects $k_{\mathrm{dyn}} = \lfloor 0.01T \rfloor$ dynamic positions, where T is the prompt length. We report this dynamic ratio separately from the effective total keep set, which also contains the fixed positions.

For the controlled quality comparison, every method uses full attention in layers 0–1 and its sparse selector in layers 2–31. This common layer configuration prevents a method's recommended dense-layer count from changing both its quality and cost.

For RULER, we evaluate all 13 tasks at 128K using 200 validation samples per task generated by the official RULER pipeline; Table 2 reports the complete results. FWE uses the official metric: the fraction of the three reference words found in each response, averaged over samples. The suite average weights all 13 tasks equally. Prompt tokens plus each task's reserved generation budget must not exceed 131,072; samples that violate this contract abort rather than being silently truncated. For LongBench, we evaluate the fixed 14-task subset listed in Table 3, covering QA, retrieval, summarization, counting, and code. We use the official zero-shot task prompts, official task metrics and generation limits. We evaluate the first 200 examples of each official task split and all 150 examples of MultiFieldQA-en, totaling 2,750 examples for each model. Inputs are middle-truncated at the token-ID level to $131{,}072 - T_{\max_\mathrm{new}}$. We report the LongBench average only over the displayed tasks and therefore describe it as a fixed evaluation suite rather than a full-benchmark aggregate. All evaluations use greedy decoding. Full attention is the quality reference, and every sparse method computes exact attention over its selected original K/V.

Paired uncertainty analysis. We assess the LongBench macro-average with a task-stratified paired bootstrap. Within each task, we resample matched JASP and dense examples with replacement, then average task means equally across the 14 tasks. Over 100,000 resamples, the JASP-minus-dense difference is $+0.22$ points with a 95% percentile interval of $[-0.13, 0.57]$ for Llama, and -0.09 points with an interval of $[-0.52, 0.34]$ for Qwen. Both intervals contain zero, and their point estimates show only small differences from dense attention.

Table 13: Sharing controls on the fixed downstream diagnostic subset.

Variant	MK2	MK3	FWE	QA2	Avg.
Per-head score	64.0	18.0	40.7	42.0	41.2
Block-diagonal score	68.0	36.0	44.0	46.0	48.5
JASP	72.0	46.0	52.7	46.0	54.2

Table 14: Diagnostic average across dynamic retrieval budgets.

Method	0.5%	1%	2%
Block-diagonal score	40.3	48.5	52.5
ParisKV	48.8	51.0	55.2
LRQK	49.7	53.3	56.3
ShadowKV	43.3	46.8	52.7
JASP	51.2	54.2	57.3

F.1 Downstream diagnostics of cross-head sharing

We use a fixed 200-example diagnostic subset at 128K: the first 50 validation examples of each of MK2, MK3, FWE, and QA2, in dataset order. MK2 and MK3 test multi-key retrieval at different difficulty levels, FWE tests frequency aggregation, and QA2 provides a question-answering reference. These tasks probe complementary behaviors without repeating the full benchmark for every configuration; individual examples are not selected by prediction correctness. This diagnostic average is distinct from the complete 13-task average in Table 2.

We use the official task metrics on a 0–100 scale. MK2 and MK3 measure reference-answer recall, FWE measures the fraction of its three reference words recovered, and QA2 accepts a match to any reference answer. Each task score is the mean over its 50 examples, and Avg. is the unweighted mean of the four task scores. Equal task weighting keeps each behavior equally represented; we do not give FWE extra weight because it has three answers. All configurations use the same samples, greedy decoding, dense layers 0–1, and 16 sink plus 128 local tokens. Compatible configurations reuse a single dense prefill per example and decode independently from the original prompt cache. JASP's 200 responses match its previously saved responses exactly.

Table 13 holds the score-fitting objective and 1% dynamic retrieval budget fixed while changing the cross-head structure. All three variants store 128 INT8 coordinates and one FP16 scale per token. The per-head control assigns 16 coordinates to each of eight KV heads, whereas the block-diagonal control allows global rank allocation without cross-head key covariance. The controls concatenate their coordinates before row-wise quantization, matching JASP's scale overhead as well as code size; their stored query and key factors also occupy the same space. JASP improves the diagnostic average by 13.0 points over the per-head control and 5.7 points over the block-diagonal control, including MK3 gains of 28.0 and 10.0 points. Task-stratified paired bootstrap with 20,000 resamples gives 95% percentile intervals of $[9.0, 17.3]$ and $[2.3, 9.3]$ for these average gains. These results connect cross-head sharing to downstream quality at a fixed index and retrieval budget.

F.2 Dynamic retrieval-budget sensitivity

Using the same diagnostic subset and metrics, we vary the dynamic retrieval ratio while keeping each method's proxy configuration and the fixed-token policy unchanged. Baselines use their quality adapters in the shared evaluation pipeline; these are not native-runtime throughput measurements. Table 14 shows that JASP's average rises from 51.2 at 0.5% to 57.3 at 2%. At 0.5%, it reaches a score close to ParisKV's 51.0 at 1%; at 1%, it exceeds the block-diagonal control's 52.5 at 2%. At the task level, increasing JASP's budget from 0.5% to 2% raises MK3 from 40.0 to 52.0 and FWE from 46.7 to 59.3, while MK2 and QA2 remain at 72.0 and 46.0. Thus, the additional retrieval budget primarily benefits multi-key retrieval and frequency aggregation in this subset, while the shared proxy already preserves the other two task scores at the lower budget.

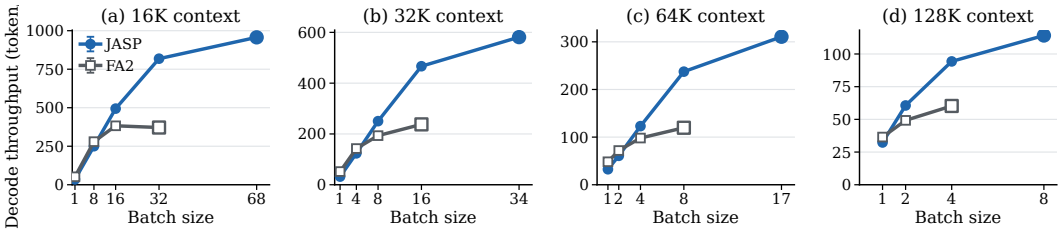


Figure 9: Decode throughput across serving batch sizes. Points report the mean and standard deviation over three independent runs under a 128-GiB CPU-memory budget. Each curve ends at the largest batch supported by the corresponding method.

G System Evaluation Details

Unless otherwise stated, all system experiments use Llama-3.1-8B-Instruct on the hardware described above. The implementation uses PyTorch, custom GPU kernels for proxy scoring and retrieval, and FlashAttention-2 for selected-token attention. JASP uses the same 1% dynamic budget, duplicate-free sink/local policy, and sparse layers as in the quality evaluation, with 25% of the original K/V placed in the GPU tail. At each context length, we report steady-state decode throughput at the largest serving batch supported by the available CPU and GPU capacity; every batch item is an independent request with its own proxy and K/V allocation.

G.1 Throughput across serving batches

Figure 9 traces steady-state decode throughput across serving batch sizes. At batch size one, JASP is slightly slower than FlashAttention-2 because every decode step additionally performs proxy scoring, token selection, and sparse KV retrieval. These fixed per-layer operations cannot be effectively amortized by a single request. JASP targets long-context serving under sustained load, where multiple requests are decoded concurrently and the serving batch grows.

As the batch size increases, JASP progressively amortizes this fixed overhead and overtakes FlashAttention-2 at batches 16, 8, 4, and 2 for 16K, 32K, 64K, and 128K contexts, respectively. At the largest batch supported by both methods, JASP achieves 1.57–2.20× higher throughput, demonstrating that its advantage already exists under identical serving batches. Its compact proxy and capacity-balanced KV placement further increase the maximum batches from 32/16/8/4 for FlashAttention-2 to 68/34/17/8 for JASP. At these capacity limits, JASP delivers 1.89–2.60× higher aggregate throughput. JASP's serving advantage therefore comes from two complementary sources: more efficient execution at the same batch size and the ability to serve substantially more concurrent long-context requests. At the maximum batches, the relative standard deviation across the three independent runs is 0.003–0.086% for JASP and 0.0003–0.030% for FlashAttention-2.

G.2 Native retrieval-system comparison

Table 15 compares released offloading systems both at their paper-recommended speed operating points and under the common 1% dynamic budget. ParisKV's native setting uses Top-100 retrieval, while LRQK's native runtime setting uses Top-1024 active tokens. The 1% setting selects 163, 327, 655, and 1,310 dynamic tokens at 16K, 32K, 64K, and 128K, respectively. All configurations use full attention in layers 0–1 and are measured under the same hardware and memory setting.

Table 15: Serving throughput and speedup over FlashAttention-2. Each entry reports tokens/s followed by the relative speedup.

Method and setting	Dynamic Top-k	16K	32K	64K	128K
FlashAttention-2	Full	371.5 / 1.00×	237.7 / 1.00×	119.7 / 1.00×	60.3 / 1.00×
ParisKV, native	100	349.5 / 0.94×	291.0 / 1.22×	213.6 / 1.78×	**130.7 / 2.17**×
ParisKV, common	1%	412.3 / 1.11×	183.8 / 0.77×	92.7 / 0.77×	46.5 / 0.77×
LRQK, native	1,024	48.0 / 0.13×	32.5 / 0.14×	28.7 / 0.24×	23.8 / 0.40×
LRQK, common	1%	137.6 / 0.37×	71.9 / 0.30×	32.4 / 0.27×	16.4 / 0.27×
JASP	1%	**956.9 / 2.58**×	**581.3 / 2.45**×	**310.7 / 2.60**×	114.1 / 1.89×

LRQK primarily uses sparse offloading to support longer contexts and larger serving batches. Its per-head proxy remains large, and selected rows are retrieved through a CPU-mediated path, so the resulting capacity savings do not translate into high serving throughput. ParisKV takes a further step toward acceleration by combining GPU-driven UVA retrieval with a fixed Top-100 budget. Although this sharply reduces retrieval volume, the budget remains 100 tokens regardless of context length and therefore covers a progressively smaller fraction of the history. At 128K it retains only 0.076% of the dynamic history, yielding a different quality operating point. When ParisKV retrieves the common 1% budget, its 128K throughput decreases to 46.5 tokens/s.

JASP combines a compact joint proxy, optimized UVA retrieval, and capacity-balanced placement to reduce scoring traffic, transfer selected K/V efficiently, and support larger serving batches. It retrieves the full 1% budget—1,310 dynamic tokens at 128K—while delivering 1.89–2.60× the serving throughput of FlashAttention-2. Under the common budget, JASP is the fastest system at every evaluated context length.

Quality at the native Top-100 operating point. We evaluate ParisKV's native runtime on the same 200-example diagnostic subset defined in Appendix F.1, using the same task metrics and equal-task average. The native configuration keeps its 128 sink tokens, 512-token local window, and 512-step update interval, with full attention in layers 0–1. Table 16 reports this operating point alongside the common 1% quality configurations. JASP improves the average from 44.8 to 54.2 and MK3 from 16.0 to 46.0. The gap is task dependent: native ParisKV scores 55.3 on FWE, versus JASP's 52.7, and both score 46.0 on QA2. Its native throughput therefore does not represent an equal-quality comparison on this diagnostic subset. Because the native and common configurations also differ in fixed-token and update policies, their quality difference is not attributable to dynamic Top-k alone.

Table 16: Quality of the native ParisKV speed operating point.

Method	Dynamic budget	MK2	MK3	FWE	QA2	Avg.
ParisKV, native	Top-100	62.0	16.0	55.3	46.0	44.8
ParisKV, common	1%	70.0	34.0	54.0	46.0	51.0
JASP	1%	72.0	46.0	52.7	46.0	54.2

Table 17: Synchronized KV-gather latency in milliseconds.

	Batch size 1				Batch size 4	
Configuration	16K	32K	64K	128K	32K	128K
CPU gather + H2D	0.216	0.338	0.475	0.832	0.837	2.096
UVA, regular pages	0.084	0.145	0.269	0.591	0.591	2.740
UVA, HugePages	0.048	0.073	0.125	0.297	0.299	1.960
JASP gather, HugePages	0.049	0.075	0.126	0.231	0.231	0.891
+ 25% GPU tail	0.043	0.062	0.101	0.181	0.179	0.706

G.3 Controlled retrieval profiling

We isolate the effects of page backing, gather implementation, and GPU-tail reads using eight KV heads with head dimension 128 and FP16 K/V. Each configuration retrieves the same uniformly sampled, duplicate-free 1% token indices from identical data. The UVA reference uses ParisKV's released gather kernel. Both UVA kernels use the same mapped-memory registration path, with either `MADV_NOHUGEPAGE` or `MADV_HUGEPAGE`; process memory mappings confirm actual huge-page backing in the latter case. CPU gather uses `index_select` followed by host-to-device copies, with indices already on the CPU. We perform three independent allocation rounds, each with five warm-up calls and 30 timed calls, and report the median of the per-round median synchronized wall times. All gathered tensors match the CPU reference bitwise. These measurements cover retrieval alone, not full-model decoding.

Table 17 shows that HugePages reduce the reference UVA latency by 49.7% at 128K and batch size one. Under the same huge-page backing, JASP's gather further reduces latency by 22.4%, followed by another 21.5% reduction from GPU-tail reads. At batch size four, the kernel change reduces latency from 1.960 to 0.891 ms, a 54.5% reduction. The two kernels perform similarly at shorter contexts and batch size one, so the kernel advantage depends on the workload. Ordinary UVA can also be slower than CPU gather, as the 128K batch-four result demonstrates. These results separate the benefits of page backing and gather scheduling from the reduced host traffic provided by GPU-resident rows.

The tail experiment directly addresses the final 25% of rows on the GPU; uniform sampling yields approximately 25% GPU hits. To isolate this access path, the diagnostic retains a full host buffer and a GPU copy of the tail, rather than measuring placement capacity. Testing four, eight, and sixteen warps per block yields similar latency, supporting the default of four. The measurements establish timing differences but do not directly measure IOMMU translation or TLB misses, since hardware performance counters were not available.