AVSG: ACCELERATED VECTORIZED SPARSE GATHER FOR EFFICIENT KV CACHE OFFLOAD IN SPARSE-ATTENTION LLM SERVING

Anonymous authors
Paper under double-blind review

ABSTRACT

Dynamic sparse attention reduces long-context attention computation by selecting only a subset of tokens, but still requires access to the full key-value (KV) cache, leaving serving memory-bound. Offloading the KV cache to host memory reduces device memory pressure but places host-to-device (H2D) transfers on the decoding critical path. In DeepSeek Sparse Attention (DSA), substantial overlap in selected KV entries across decoding steps creates an opportunity for device-resident reuse. Exploiting this reuse efficiently, however, presents three critical challenges: (1) costly matching of selected tokens against entries retained in the HBM buffer, (2) uneven distribution of the remaining H2D transfers across accelerator cores, and (3) retaining frequently accessed KV entries within limited HBM capacity. We present AVSG (Accelerated Vectorized Sparse Gather), an operator that addresses these challenges for DSA while preserving its exact token selections. AVSG uses vectorized hash matching to identify reusable HBM-buffer slots and reserve slots for entries requiring transfer, then evenly partitions these H2D transfers across accelerator cores. Lifetime-based buffer management retains frequently accessed KV entries in the device buffer across decoding steps, and shared slot reservations extend reuse across tokens within a multi-token prediction iteration. On a single NPU, vectorized hash matching is $2.80\times$ faster than scalar dual-pointer matching, miss-only transfer raises effective H2D bandwidth by up to $34.44\times$ over request-level assignment, and an 8K-entry buffer reaches a 94.83% hit rate on real requests. These gains translate into end-to-end improvement: on a serving stack processing a real-world production dataset, AVSG reduces time per output token by 39% and increases output throughput by $1.27\times$ relative to the same KV offload layout without HBM-buffer reuse, demonstrating the benefit of efficient matching, balanced transfer, and effective HBM residency in production-scale serving.

1 INTRODUCTION

Modern large language model (LLM) applications, including agentic tool use and retrieval-augmented assistants, increasingly operate on contexts approaching one million tokens, as exemplified by recent models such as GLM-5.3 (Zhipu AI, 2026), and DeepSeek-V4 (DeepSeek-AI, 2026). For inference serving, the central objective is to sustain high aggregate throughput while controlling per-token latency, commonly measured by tokens per second (TPS) and time per output token (TPOT).

Serving such long contexts is fundamentally memory intensive. A request's key-value (KV) cache grows linearly with context length and can exceed the high-bandwidth memory (HBM) capacity of a single device (Xu et al., 2026). Only a small number of long-context requests can therefore remain resident in HBM, limiting serving concurrency. DeepSeek Sparse Attention (DSA) (DeepSeek-AI, 2024) reduces attention computation by selecting a fixed top-K subset of historical tokens. However, because an indexer may select any earlier token, the full historical KV state and indexer-key state must remain addressable. Sparse computation alone does not eliminate the storage bottleneck.

KV cache offloading addresses this capacity problem by placing the full history in host memory and retaining only a working set on the accelerator. This frees HBM for additional requests, but places host-to-device (H2D) transfers on the decoding critical path. Sparse attention cannot proceed until the

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.

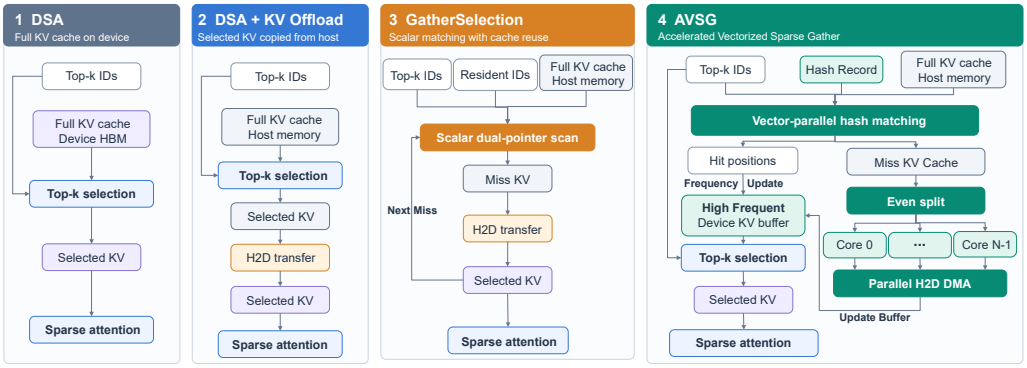


Figure 1: Evolution of KV transfer strategies for DSA inference. (1) HBM-resident full KV cache limits batch size. (2) KV offloading transfers all selected entries each step. (3) GatherSelection reuses a fixed-size cache through sequential matching. (4) AVSG reduces offloading latency through vectorized matching, balanced multi-core transfers, and retention of frequently accessed KV entries.

selected KV entries have been transferred to device memory. A naive implementation treats decoding steps independently and transfers all selected entries at every step.

Device-resident reuse pools (e.g., GatherSelection[1]) and prefetching can reduce this cost but must address three coupled challenges. **First, matching overhead:** each selected token must be matched against the HBM buffer and mapped to a slot before attention; sequential lookup can offset transfer-time savings from reuse. **Second, fragmented transfers:** filtering out buffer hits leaves scattered KV entries for H2D transfer, with demands varying across requests and steps. Assignment by request or selected position can imbalance core workloads, limiting effective transfer bandwidth. **Third, effective HBM residency:** a limited buffer must retain frequently selected entries beyond the previous step while reclaiming less useful ones; repeatedly evicting useful entries wastes reuse opportunities and transfer bandwidth. Jointly addressing these challenges is necessary to translate reduced transfer volume into lower decoding latency.

We present AVSG (Accelerated Vectorized Sparse Gather), an exact post-selection gather operator for offloaded DSA inference (Figure 1-4). AVSG jointly addresses these challenges by reducing redundant KV transfers and accelerating the matching and movement of selected entries, without modifying the indexer or its token selections. **Our contributions** are as follows:

- **Vectorized Hash Matching via Open Addressing**: We replace scalar dual-pointer matching with vectorized hash probing, decoupling device buffer capacity S from selection size K without a proportional matching penalty. Matching is $2.80\times$ faster at $S = K = 2048$, while latency grows sublinearly with K and S over the tested ranges (§3.2–3.3; §4.2).
- **Evenly Partitioned Multi-Core H2D Transfer**: We balance cache-miss KV transfers across AI-Cores to reduce transfer imbalance, achieving up to $34.44\times$ and $3.48\times$ the effective H2D bandwidth of request-level assignment and equal top-K partitioning (§3.4; §4.3).
- **Device Cache for Frequently Accessed KV Entries**: We combine per-slot aging with hit-triggered lifetime refresh to retain frequently accessed KV entries across decoding steps and reclaim inactive slots. On the evaluated real requests, an 8K-entry buffer achieves a 94.83% hit rate, 13.22% above the adjacent-step reuse rate (§3.5; §4.4).
- **Shared-Buffer Reuse Across MTP Tokens**: We sequentially process the tokens of a multi-token prediction (MTP) iteration through a shared buffer, allowing later tokens to reuse earlier slot reservations. This achieves 61.87% reuse in the first decode iteration from an empty cache under MTP-3 (§3.6; §4.4).

Together, these mechanisms reduce decode TPOT by 39% and increase throughput by $1.27\times$ over the same KV offload layout without HBM-buffer reuse during end-to-end serving evaluation (§4.5).

[1] https://github.com/hicann/cann-recipes-infer/tree/master/ops/ascendc/src/gather_selection_kv_cache

2 Background and Related Work

LLM Inference and KV Cache Management. PagedAttention reduces KV fragmentation through paged allocation, while RadixAttention reuses prefixes across requests through a radix tree (Kwon et al., 2023; Zheng et al., 2024). Neither reduces an active request's KV storage requirement, so offloading remains necessary when aggregate KV demand exceeds HBM capacity.

Sparse Attention Mechanisms. Sparse attention reduces computation through reformulation or token selection (Wu et al., 2026a). Training-free methods use structured patterns or approximate selection without retraining, at potential accuracy cost (Xiao et al., 2024; Jiang et al., 2024; Tang et al., 2024; Lee et al., 2024; Wu et al., 2025a;b; 2026b); training-time methods incorporate sparsity during training, including MoBA, NSA, InfLLM-v2, and DSA (Lu et al., 2025; Yuan et al., 2025; Zhao et al., 2025; DeepSeek-AI, 2024). DSA uses a lightweight learned indexer to select top-K tokens for sparse MLA, bounding per-operation KV reads while full KV and indexer-key storage still grow with context length. IndexCache (Bai et al., 2026) eliminates up to 75% of per-layer indexer computations through cross-layer reuse; this is orthogonal to AVSG's acceleration of post-selection KV materialization.

KV cache offloading. KV-cache offloading extends the storage hierarchy beyond accelerator HBM to support long-context execution, cross-request reuse, and disaggregated serving. LMCache (Liu et al., 2025) offloads KV states to hierarchical storage for cross-query reuse; CachedAttention (Gao et al., 2024) organizes host-memory/disk tiers for cross-turn reuse with overlapped access; Mooncake (Qin et al., 2025) pools CPU, DRAM, SSD, and NIC resources into a distributed cache within a prefill–decode disaggregated system; Strata (Xie et al., 2026) combines GPU-assisted KV-cache I/O with cache-aware scheduling; and Pensieve (Yu et al., 2025) maintains conversational state across requests with a two-tier GPU–CPU cache. These systems target storage capacity, request-level reuse, and cache placement—not the per-operation selection of KV entries that sparse attention requires.

KV offloading acceleration. Fine-grained, sparse-attention-oriented systems instead reduce the cost of materializing the KV entries each attention operation needs. ArkVale (Chen et al., 2024) evicts and recalls KV pages by query-dependent importance; InfiniGen (Lee et al., 2024) prefetches via next-layer attention rehearsal; FreeKV (Liu et al., 2026b) and LiteCache (Yi et al., 2026) reuse previously selected KV across adjacent decoding steps with speculative correction; SparseServe (Zhou et al., 2025) adds fragmentation-aware transfers and working-set-aware batching; ECHO (Liu et al., 2026a) hides offload latency with lossless intra/inter-query prefetching fused with indexer computation. GatherSelection (Figure 1(3)) belongs to this family but starts from the exact Top-K identifiers: it sort–merge matches requested against cached token ids, retains same-row hits, and fetches only misses from the full KV cache. AVSG preserves this exact post-selection interface but replaces repeated sort–merge matching with a persistent hash-and-lifetime pool with explicit slot remapping, combining vectorized hash probing, fine-grained core allocation, and overlapped remapping and H2D transfer to retain frequently accessed KV entries across steps without modifying the indexer.

3 Method

3.1 Overview

We distinguish the full host-resident KV cache from AVSG's *HBM buffer*, which holds selected KV entries and their reserved destinations. Unless otherwise stated, a buffer hit means matching finds a previously reserved, reusable slot; a buffer miss requires a new slot reservation and an H2D copy. These terms describe reuse in the HBM buffer, not whether the KV entry exists in the full KV cache.

Figure 2 illustrates the AVSG operator. At each decode step DSA supplies a set of K top-K token indices, and AVSG must make the corresponding KVs resident in device memory before the attention operator reads them. Because the full KV cache remains in host memory, the quantity AVSG controls is how much of it crosses the H2D link: instead of fetching every selected index, AVSG carries a device cache buffer of S slots across steps and transfers only indices not already resident. Within a step, AVSG resolves every index against the buffer (Figure 2a), classifying it as hit or miss and

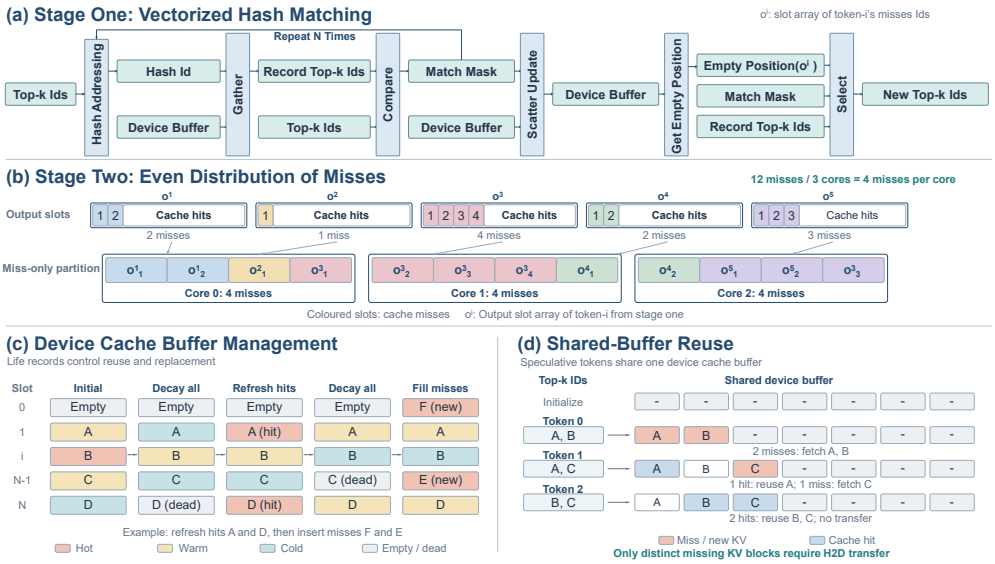


Figure 2: Detailed AVSG algorithm design. **(a)** Vectorized open-addressing matching and updates of top-K IDs. **(b)** Even distribution of KV misses across AICores for efficient H2D transfer. **(c)** Retention of frequently accessed KV entries and replacement of cold entries. **(d)** A shared buffer unions top-K selections across a request's MTP tokens to avoid duplicate copies.

assigning each miss a destination slot, then fetches exactly the misses from host memory spread across the available AICores (Figure 2b).

What makes the hit/miss split worth exploiting is the policy deciding which indices stay resident. Each slot carries a life record that decays every step, is refreshed on hit, and marks the slot reusable once expired, so the buffer keeps the indices DSA selects repeatedly and releases the rest (Figure 2c). Let N_{mtp} denote the number of additional speculative tokens per request. The initial token and these N_{mtp} speculative tokens share a single buffer and are processed one after another (Figure 2d), so indices one token fetches as misses are hits for the tokens behind it.

3.2 Hash-Stat e: A Device-Resident Selection Pool

To map selected token IDs to reusable buffer slots and allocate destinations for misses, AVSG organizes its metadata in a *Hash-State* table. The table resides entirely in on-chip memory and supports SIMD Flynn (1972) instructions, like probing, comparison, and updates.

Problem formulation. At each decode step, each token contributes K selected indices $\boldsymbol{x} = (x_0, \ldots, x_{K-1})$ (e.g., K=2048). Each request contributes $N_q = N_{\text{mtp}} + 1$ tokens. A batch of B requests contains $N_g = B \times N_q$ tokens. Tokens within a request share an S slot hash table. Each table retains KV address fetched on previous steps until its life records expire (§3.5), allowing repeated selections to reuse cached entries. Given $\boldsymbol{x}$ and the request's table, Stage One maps each selected index x_i ($i \in \{0, \ldots, K-1\}$) to a hit indicator $m_i \in \{0, 1\}$ and a KV slot $o_i \in \{0, \ldots, S-1\}$:

$$m_i = \begin{cases} 1, & \text{if } x_i \text{ is resident in the table,} \\ 0, & \text{otherwise (a cache miss),} \end{cases} \tag{1}$$

where o_i names the slot that already holds the token (hit) or is freshly allocated to hold it (miss); misses are fetched from host memory in Stage Two. Probing is governed by three parameters: two distinct primes a_1, a_2 that set a probe's initial bucket and stride, a retry budget R bounding the probe chain (default R=3, i.e., R+1 candidate buckets per token), and the maximum lifetime L, to which a hit resets the slot's lifetime (default L=16, chosen via the ablation in Appendix A).

Hash-State table. To support these probes and lifetime updates, AVSG maintains a *Hash-State* table $\boldsymbol{s} = (E_0, \ldots, E_{S-1})$ of 64-bit entries. Each entry $E_j = (c_j, q_j)$ stores a full 32-bit selected index c_j

and a 32-bit metadata word q_j, partitioned as

$$q_j = \Big[\underbrace{\text{lifetime}}_{8\text{ bits}} \;\Big|\; \underbrace{\text{invalid}}_{1\text{ bit}} \;\Big|\; \underbrace{\text{id_footprinter}}_{(23-m)\text{ bits}} \;\Big|\; \underbrace{\text{slot}}_{m\text{ bits}}\Big]. \tag{2}$$

The capacity is a power of two, $S{=}2^m$, with m the slot bit width, so one packed word can identify any slot (e.g., $m{=}13$ gives $S{=}8192$). In Equation 2, fields are listed from the most significant bit to the least, so slot occupies the low m bits of q_j. c_j stores the selected indice occupying slot j, while q_j's lifetime tracks its access recency for eviction. Hash probing visits bucket h_i for x_i; following the slot field o_i of q_{h_i} retrieves c_{o_i}, which is compared with x_i to determine m_i. The invalid and id_footprinter fields of q_{h_i} is used to determine whether o_i can be overwritten during allocation (§3.3).

This Hash-State layout enables vectorized parallel access to the table in an AICore's unified buffer (UB) using SIMD instructions. Section 3.3 describes how these instructions implement parallel matching and hash-table state updates.

Notation convention. Boldface symbols such as $\boldsymbol{m}, \boldsymbol{o}, \boldsymbol{h}, \boldsymbol{d}, \boldsymbol{c}$ denote the length-K vector counterparts of the scalars m_i, o_i, h_i, d_i, c_i, and every operation on them executes as one SIMD instruction. A *masked assignment* $\boldsymbol{y}[\boldsymbol{m}] \leftarrow \boldsymbol{t}$ updates only the elements of $\boldsymbol{y}$ whose mask $\boldsymbol{m}_i{=}1$, leaving the rest unchanged; $\boldsymbol{y}[\neg\boldsymbol{m}]$ updates the elements with $\boldsymbol{m}_i{=}0$.

3.3 Stage One: Three-Pass Vectorized Matching

Stage One turns the hit/miss classification of Eq. 1 into three passes over the UB-resident table, each with a distinct responsibility (Algorithm 1): a *refresh pass* that maintains recency, a *classification pass* that produces the outputs Stage Two consumes, and an *allocation pass* that installs the misses into the table. The refresh pass decays all lifetimes by one only for the first token of each request, then calls Probe for the current token and restores the lifetimes of matched entries to L before allocating misses. The classification pass reuses the hit mask $\boldsymbol{m}$ and bucket indices $\boldsymbol{h}$ from the refresh pass to record hit slots o_i and compute the token's miss count n_g. Allocation fills remaining slots for misses, yielding device positions for downstream attention without a separate gather kernel. The per-token miss counts, collected across the batch as $\boldsymbol{n} = (n_0, \ldots, n_{N_g-1})$, drive Stage Two's load balancing (§3.4). The allocation pass frees capacity—collecting dead slots and, when fewer than the miss count, decaying all records further until enough die, the vectorized analogue of evicting the n_g least-recent entries—then installs each miss into a freed position and links a hash bucket to that slot.

Probe primitive. The refresh pass calls the read-only primitive $\mathrm{Probe}(\boldsymbol{x}, \boldsymbol{s})$ once per token, returning the hit mask $\boldsymbol{m}$ and bucket-index vector $\boldsymbol{h}$ (Appendix B, Eq. 8): given a token's top-K list $\boldsymbol{x}$, compute initial buckets $\boldsymbol{h} \leftarrow (a_1\boldsymbol{x}) \bmod S$ and odd strides $\boldsymbol{d} \leftarrow ((a_2\boldsymbol{x}) \bmod S) \,|\, 1$, then repeat up to R times—gather each bucket's metadata word, follow its slot, gather the cached id $\boldsymbol{c}$, and advance unresolved probes by their strides—with one final un-advanced comparison yielding the hit mask $\boldsymbol{m} \leftarrow [\boldsymbol{c} = \boldsymbol{x}]$, so each x_i examines at most $R{+}1$ buckets. Every probe runs the full vector of ids, costing $O(1)$ SIMD instructions regardless of K and S; this constant-time probing is what lets the hash table be scaled to raise the hit rate without a proportional matching penalty.

Allocation mechanics. Mapping index x_i to slot o_i requires checking whether q_{h_i} can be overwritten: its slot field may still point to a slot whose previous occupant has been replaced. AVSG uses fingerprint verification and lazy reclamation to identify and overwrite such stale mappings. Using the fields of q_{h_i} defined in §3.2, AVSG verifies the mapping by comparing the stored fingerprint with that of c_{o_i}. Since bit 23 of the 24-bit index field is reserved as the invalid flag, a top-K id x has $F = 2^{23}/S = 2^{23-m}$ representable fingerprints, encoded element-wise as $\phi(x) = x \bmod F$ and $\widehat{\phi}(x) = \phi(x)\, S$; the packed word

$$q_{h_i} = 2FS\,\ell_{h_i} + FS\,b_{h_i} + \widehat{\phi}(c_{o_i}) + o_i \tag{3}$$

places the preserved lifetime ℓ_{h_i}, the invalid flag b_{h_i}, the stored fingerprint, and the slot o_i in the bit ranges declared in Equation 2, so summation is equivalent to concatenation. Stale pointers are reclaimed lazily: because AVSG keeps no reverse index from a slot back to its pointing buckets, an eager repair would require a full $O(S)$ scan, so staleness is detected at bucket-reuse time, riding on the metadata read that must happen anyway before an overwrite—a bucket is reusable if its mapping is invalid, if the pointed slot is inactive, or if the fingerprint no longer matches the slot's occupant,

$$\mathrm{reusable}(q_{h_i}) = \neg\ \mathrm{valid}(q_{h_i})\ \vee\ \neg\ \mathrm{active}\big(q_{o_i}.\text{lifetime}\big)\ \vee\ \big[\phi_{h_i} \neq \phi(c_{o_i})\big], \tag{4}$$

Algorithm 1 Stage One: Three-Pass Vectorized Matching Per Token

Input: $g \in \{0, \ldots, N_q - 1\}$: token index within the request; selected indices $\boldsymbol{x}$ and request-specific table $\boldsymbol{s}$
Output: Slot vector $\boldsymbol{o}$; per-token miss count n_g; updated table $\boldsymbol{s}$

Pass 1 — Refresh: decay, then restore records of hits (Steps 1–2 of Algorithm 3)
if $g = 0$:
  $\boldsymbol{s}$.lifetime $\leftarrow \boldsymbol{s}$.lifetime $- 1$
$\boldsymbol{m}, \boldsymbol{h} \leftarrow \mathrm{Probe}(\boldsymbol{x}, \boldsymbol{s})$ # hit mask $\boldsymbol{m}$, bucket indices $\boldsymbol{h}$; Eq. 8
$\boldsymbol{o}^{\mathrm{match}} \leftarrow q_{\boldsymbol{h}[\boldsymbol{m}]}$.slot
$q_{\boldsymbol{o}^{\mathrm{match}}}$.lifetime $\leftarrow L$
Pass 2 — Classify: emit slot indices and per-token miss counts.
$\boldsymbol{o}[\boldsymbol{m}] \leftarrow \boldsymbol{o}^{\mathrm{match}}$
$n_g \leftarrow \sum_{i=0}^{K-1}(1 - m_i)$ # m_i is element i of the hit mask $\boldsymbol{m}$
Pass 3 — Allocate: free slots, install misses, link buckets (Steps 3–4 of Algorithm 3)
$\boldsymbol{e} \leftarrow \mathrm{CollectDead}(\boldsymbol{s})$ # Eq. 12, Appendix B
while $|\boldsymbol{e}| < n_g$:
  $\boldsymbol{s}$.lifetime $\leftarrow \boldsymbol{s}$.lifetime $- 1$; $\boldsymbol{e} \leftarrow \mathrm{CollectDead}(\boldsymbol{s})$
$\boldsymbol{t} \leftarrow \boldsymbol{x}[\neg\, \boldsymbol{m}]$; $\boldsymbol{o}^{\mathrm{miss}} \leftarrow \boldsymbol{e}[: n_g]$ # selected indices that miss take freed slots
$\boldsymbol{s} \leftarrow \mathrm{UpdateState}(\boldsymbol{t}, \boldsymbol{o}^{\mathrm{miss}}, \boldsymbol{s})$ # Algorithm 2
$\boldsymbol{o}[\neg\, \boldsymbol{m}] \leftarrow \boldsymbol{o}^{\mathrm{miss}}$ # fill placeholders

where $\mathrm{valid}(q_{h_i}) = [b_{h_i} = 0]$ and $\mathrm{active}(\ell) = [\ell \geq 0]$. Here ϕ_{h_i} denotes the stored fingerprint, and $o_i = q_{h_i} \bmod S$ extracts the slot field defined in §3.2; the full decode chain appears in Appendix F. An undetected stale pointer may increase probing overhead, but full 32-bit index comparisons prevent false hits (§3.2). The allocation pass applies the reusable predicate to all misses in vectorized form (Algorithm 2, Appendix B), using the same initial buckets and odd strides. Every miss writes its selected index and initializes the lifetime in its allocated slot; only probes that find reusable buckets install the corresponding mappings.

3.4 Fine-Grained Multi-Core H2D Transfer

After matching, each selected index x_i with $m_i = 0$ requires an H2D transfer from position x_i in the host KV cache to slot o_i in the device buffer. Each AICore performs these transfers through its Direct Memory Access (DMA) engine,reading host memory directly without an intermediate staging copy. Under DSA's single-MLA-head configuration, a batch contains $N_g = B \times N_q$ tokens, each contributing K selected indices and their matching results, which together determine the batch's H2D transfer workload.

With C AICores, let W_c denote the H2D transfer workload of core $c \in \{0, \ldots, C-1\}$. $\mathcal{A}_c$ is defined as the set of selected-index positions (g, i) assigned to core c, where g identifies a token and i a position in its selected indices. Since only misses require transfer, each position contributes $1 - m_{g,i}$ KV entries, where $m_{g,i}$ is its hit indicator. The resulting transfer workload is

$$W_c = \sum_{(g,i)\in\mathcal{A}_c} (1 - m_{g,i}) = |\mathcal{A}_c| - \sum_{(g,i)\in\mathcal{A}_c} m_{g,i}. \tag{5}$$

AVSG constructs each core's assignment $\mathcal{A}_c$ using only misses, so $m_{g,i} = 0$ for all $(g, i) \in \mathcal{A}_c$. Consequently, $\sum_{(g,i)\in\mathcal{A}_c} m_{g,i} = 0$, yielding $W_c = |\mathcal{A}_c|$ from Eq. 5. For token g, the miss count is $n_g = \sum_{i=0}^{K-1}(1 - m_{g,i})$, giving $N_{\text{miss}} = \sum_{g=0}^{N_g-1} n_g$ transfers across the batch. AVSG orders these misses by token and partitions them into C disjoint assignments covering every miss exactly once. Each core receives either $\lfloor N_{\text{miss}}/C \rfloor$ or $\lceil N_{\text{miss}}/C \rceil$ consecutive misses. For any two cores $c, d \in \{0, \ldots, C-1\}$,

$$W_c = |\mathcal{A}_c| \in \left\{ \left\lfloor \frac{N_{\text{miss}}}{C} \right\rfloor, \left\lceil \frac{N_{\text{miss}}}{C} \right\rceil \right\}, \qquad |W_c - W_d| \leq 1. \tag{6}$$

Thus, the H2D workloads of any two cores differ by at most one KV entry, regardless of the original request or cache-hit distribution. Section 4.3 compares AVSG with request-level assignment and equal top-K partitioning; Appendix D analyzes the transfer imbalance of these baselines.

3.5 Device Cache for Frequently Accessed KV Entries

Within a request, N_q tokens, each with K selected indices, share an S-slot hash table. To accommodate all corresponding slots even when the N_qK indices are distinct, the capacity must satisfy

$$S \geq N_qK = (N_{\text{mtp}} + 1)K, \tag{7}$$

ensuring enough capacity for the request's full working set even if every selection misses, without evicting entries still needed in the current step.

Lifetime-Based Slot Management. The lifetime field of q_j carries an 8-bit counter (0–L) that tracks access recency. The updates in Algorithm 1 ensure that unmatched old slots expire before slots matched by the current request. Global pre-decay leaves unmatched old slots with life at most $L - 1$, while the request-wide refresh restores every matched slot to L. Together with the capacity bound in Eq. 7, this preserves the slots required by the current request. The policy implements LRU-like aging based on access recency, rather than an exact LRU ordering. Appendix B provides the consolidated update procedure and a proof of this preservation guarantee.

Frequently Accessed Data Residency. Because the resident set is determined by access recency, the hash table content tracks the workload's top-K selection pattern. For each selected index x_i, the lifetime associated with c_{o_i} is set to L, either by refreshing a hit or initializing a newly allocated entry. Without another refresh, the slot becomes reclaimable after more than L unit decays. Repeated selections thus prolong residency, while unaccessed entries eventually expire.

3.6 Shared-Buffer Reuse Across MTP Tokens

AVSG supports shared-buffer reuse across MTP tokens through two mechanisms: reuse of matching information to avoid redundant KV transfers within a step, and speculative-entry invalidation to preserve correctness across steps.

Progressive Reuse of Matching Information. A later token can match a slot reserved by an earlier token even before that slot's KV data arrive: both tokens refer to the same destination slot, and the corresponding miss requires only one H2D copy. At cold start the initial token reserves slots for its misses, while later tokens reuse those reservations and reduce the total transfer volume. We evaluate the resulting first-step reuse in §4.4.

Speculative-Entry Invalidation. For a selected index x_i, the slot field of q_{h_i} identifies o_i, and (§3.4) copies the corresponding host KV entry into device slot o_i. If x_i is a speculative index, its host KV entry may be overwritten during subsequent decoding, leaving the device copy at o_i stale. Retaining the mapping could then yield a hit ($m_i = 1$) based on the unchanged index and reuse KV data that no longer match the host entry, compromising inference accuracy. AVSG therefore marks the selected index in the referenced slot as empty ($c_{o_i} \leftarrow -1$) for each speculative index. Subsequent comparisons with x_i then fail, preventing stale hits and requiring a fresh H2D copy when the index is reselected.

4 Evaluation

4.1 Experimental Setup

We compare three configurations under identical request workloads: **Baseline (No Offload)**, which keeps all KV in HBM with device capacity capped by the memory left after model weights; **KV Offload**, which offloads the full KV cache to host memory and transfers every activated KV at each decode step with no reuse; and **AVSG**, which augments the same offload layout with a cache buffer that retains frequently accessed KV entries, vectorized hash matching, and fine-grained two-stage transfer. The component-level experiments (§4.2–§4.4) isolate each mechanism on a single NPU; the end-to-end serving comparison (§4.5) runs the full 5P1D serving stack, whose platform and memory layout we describe there.

4.2 Vectorized Hash Matching Performance

Vectorized hash matching reduces lookup latency while allowing the device buffer to grow without a proportional matching penalty, supporting the design in §3.3. Figure 3 shows a reduction from

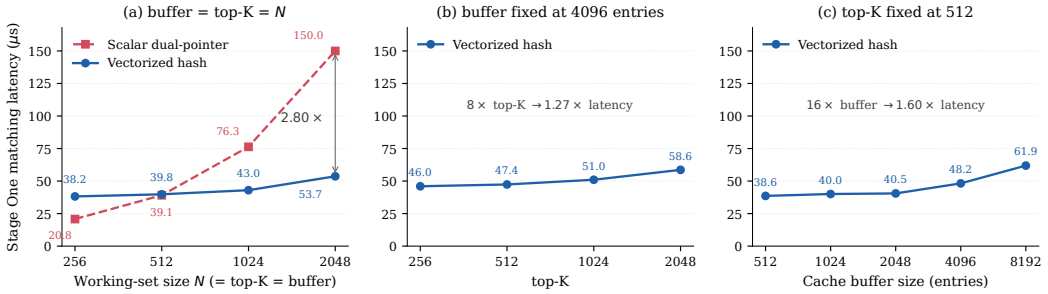


Figure 3: Stage One matching latency (median of 1000 iterations). **(a)** Scalar versus vectorized matching at $S = K = N$. **(b)** Varying K at $S = 4096$. **(c)** Varying S at $K = 512$.

150.0 to 53.7 μs at $K = S = 2048$ relative to the scalar dual-pointer baseline (64.2% lower latency, 2.80× faster). Increasing the selection size K by 8× at fixed buffer capacity S raises latency by only 1.27×; increasing S by 16× at fixed K costs 1.60×. This sublinear growth over the tested ranges makes it practical to retain more reusable KV entries without a comparable increase in lookup time. Appendix C provides the profiling protocol and detailed analysis.

4.3 Fine-Grained Multi-Core H2D Transfer Efficiency

Evenly distributing actual cache misses across AI-Cores improves H2D efficiency both when the batch is small and cache reuse leaves fewer KVs to transfer, as intended by §3.4. In Figure 4(a), AVSG achieves up to 34.44× the bandwidth of request-level assignment ($B = 1$, 25% reuse), demonstrating the benefit of spreading a request's transfers across cores. Against equal top-K partitions, it achieves up to 3.48× higher bandwidth ($B = 8$, 75% reuse; Figure 4(b)), supporting the choice to balance the remaining copies after identifying cache hits. Full measurements and analysis appear in Appendix D (Table 3).

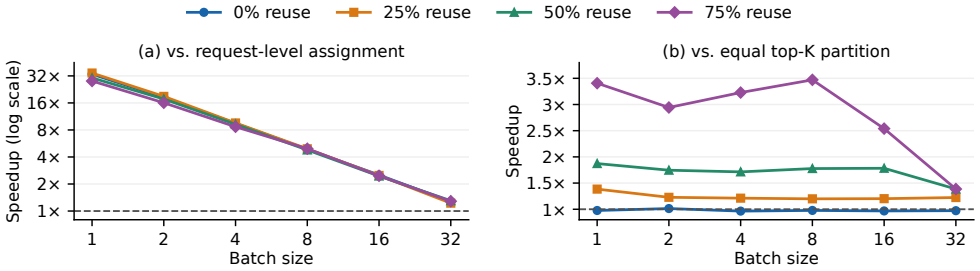


Figure 4: H2D bandwidth gains from evenly splitting cache misses across AI-Cores: **(a)** over request-level assignment, which leaves cores idle at small batches; **(b)** over equal top-K partitions, whose transfer workloads can become uneven after cache reuse.

4.4 Cache Reuse Rate Analysis

AVSG reduces repeated KV copies by retaining frequently accessed entries across decode steps and sharing slot reservations within each MTP iteration (§3.5, §3.6). On the 2000 requests of §4.5 with $K = 2048$, adjacent-step reuse—the fraction of selected entries also selected in the preceding step—is 81.61%. The 4K and 8K buffers achieve hit rates of 88.72% and 94.83%, respectively, showing that additional capacity preserves useful entries beyond the previous step. Table 1 also reports the best synthetic gains at 50%, 70%, and 90% adjacent-step reuse. Even from an empty buffer, MTP-3 enables four tokens (one initial and three speculative) to achieve 61.87% reuse, reducing average KV copies from 8192 to $8192(1 - 61.87\%) \approx 3124$. The initial token still misses; later tokens reuse its slot reservations before the collective H2D transfer. Appendix E examines capacity across reuse distributions and cold-start reuse across MTP degrees, while Appendix A evaluates lifetime versus reuse and kernel execution time.

Table 1: Cache reuse ($K = 2048$). Rates are percentages; gain is 8K hit rate minus adjacent-step reuse (pp). Synthetic rows maximize this gain at each reuse level, all with historical recall mean 0.9.

Data	Adjacent-step reuse	4K hit rate	8K hit rate	Gain (pp)
Synthetic	50.00	84.29	94.15	+44.15
Synthetic	70.00	90.12	96.79	+26.79
Synthetic	90.00	96.21	99.04	+9.04
Real	81.61	88.72	94.83	+13.22

MTP-3 cold start ($S = 8$K, four tokens per request)

Metric	Independent buffers	Shared buffer
Cache hit rate (%)	0	61.87
KV entries copied (avg.)	8192	3124

4.5 End-to-End Serving Performance

The component experiments above validate each mechanism in isolation; this subsection asks whether the gains survive integration into a full serving stack. All three settings run a 5P1D dis-aggregated topology (12 nodes of 910B NPUs) serving GLM-5.3-w4a8c16 with 3-token speculative decoding. The workload is an agent dataset with a total of 2000 real chat requests collected from a production system. The average prompt length is 48,000 tokens, and we test with a maximum output of 32768 tokens, a concurrency of 64, and a request rate of 1. We extend the closed-loop benchmarking methodology of Wang et al. (2025): throughout the evaluation the system is bounded either by fixed in-flight concurrency or by the request injection rate, so it is continuously stressed up to its throughput limit. In this regime, higher throughput naturally comes with a larger TTFT, as more concurrently served requests contend for prefill compute, so the two metrics trade off and must be read together.

Table 2: End-to-end cold-start comparison (5P1D, GLM-5.3-w4a8c16). TPOT from the second token onward; HBM usage is the per-chip decode average.

Metric	**No Offload**	**KV Offload**	**AVSG**
Device KV capacity (tokens)	145,536	762,496	762,496
Decode HBM usage (avg)	91.8%	91.6%	95.5%
AVG_TTFT (s)	7.92	5.94	8.69
AVG_TPOT (ms)	43	91	**55**
TP99_TPOT (ms)	64	126	**77**
AVG_E2E (s)	119.1	120.2	**75.8**
Output TPS (tok/s)	472.6	451.2	**571.4**
Achieved request rate (req/s)	0.44	0.40	**0.54**

No Offload attains the lowest TPOT (43 ms, no H2D on the decode path) but is capacity-capped: its 145K-token device pool cannot sustain the 48K-token prompts at concurrency 64. KV Offload removes the cap but pays 91 ms per step transferring the full selection. AVSG shares KV Offload's exact layout (identical device KV pool and host DRAM offload), so the gap isolates the access strategy: reusing buffer hits and transferring only misses reuse cuts TPOT 39% to 55 ms and raises throughput $1.27\times$ to 571 tok/s, at the cost of $\sim$4 GB more HBM.

5 Conclusion

AVSG reduces the decoding overhead of KV offloading for DSA while preserving exact token selections, showing that the overlap in selected KV entries across decoding steps is a first-class resource for serving efficiency. Its three mechanisms each remove one bottleneck: vectorized hash matching accelerates lookup $2.80\times$ over scalar dual-pointer matching, balanced miss-only transfer raises effective H2D bandwidth by up to $34.44\times$ over request-level assignment, and lifetime-based residency attains a 94.83% hit rate on real requests, with shared slot reservations extending reuse across MTP tokens (61.87% even from an empty buffer under MTP-3). On a serving stack processing a real-world production dataset, AVSG cuts TPOT by 39% and sustains $1.27\times$ the output throughput of the same offload layout without reuse. Matrix-level vectorization, latency-hiding state updates, and multi-tier memory hierarchies remain open directions (Appendix H).

REFERENCES

Yushi Bai, Qian Dong, Ting Jiang, Xin Lv, Zhengxiao Du, and Aohan Zeng. Indexcache: Accelerating sparse attention via cross-layer index reuse. arXiv preprint 2603.12201, Zhipu AI and Tsinghua University, 2026. URL https://arxiv.org/abs/2603.12201.

Renze Chen, Zhuofeng Wang, Beiquan Cao, Tong Wu, Size Zheng, Xiuhong Li, Xuechao Wei, Shengen Yan, Meng Li, and Yun Liang. ArkVale: Efficient generative llm inference with recallable key-value eviction. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 113134–113155. Curran Associates, Inc., 2024. doi: 10.52202/079017-3595. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/cd4b49379efac6e84186a3ffce108c37-Paper-Conference.pdf.

DeepSeek-AI. Deepseek-v3 technical report. arXiv preprint 2412.19437, DeepSeek, 2024. URL https://arxiv.org/abs/2412.19437.

DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence. arXiv preprint 2606.19348, DeepSeek, 2026. URL https://arxiv.org/abs/2606.19348.

Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, 1972. doi: 10.1109/TC.1972.5009071.

Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. Cost-efficient large language model serving for multi-turn conversations with cachedattention. In *Proceedings of the 2024 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'24, USA, 2024. USENIX Association. ISBN 978-1-939133-41-0.

Huiqiang Jiang, Yucheng Li, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Zhenhua Han, Amir H. Abdi, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. MInference 1.0: Accelerating pre-filling for long-context llms via dynamic sparse attention, 2024. URL https://arxiv.org/abs/2407.02490.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hoffman, Tianle Yu, Liuyijie Zhang, Joseph E. 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 (SOSP)*. ACM, 2023.

Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. InfiniGen: Efficient generative inference of large language models with dynamic kv cache management. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation*, OSDI'24, USA, 2024. USENIX Association. ISBN 978-1-939133-40-3.

Guangda Liu, Wenhao Chen, Chengwei Li, Zhenyu Ning, Jing Lin, Yiwu Yao, Quan Chen, Shixuan Sun, Jieru Zhao, and Minyi Guo. ECHO: Efficient KV cache offloading with lossless prefetching for serving native sparse attention LLMs. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*, Seattle, WA, July 2026a. USENIX Association. URL https://www.usenix.org/conference/osdi26/presentation/liu-guangda.

Guangda Liu, Chengwei Li, Zhenyu Ning, Jing Lin, Yiwu Yao, Danning Ke, Minyi Guo, and Jieru Zhao. FreeKV: Boosting kv cache retrieval for efficient llm inference, 2026b. URL https://arxiv.org/abs/2505.13109.

Yuhan Liu, Yihua Cheng, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Rui Zhang, Kuntai Du, and Junchen Jiang. LMCache: An efficient kv cache layer for enterprise-scale llm inference, 2025. URL https://arxiv.org/abs/2510.09665.

Enzhe Lu, Zhejun Jiang, Jingyuan Liu, Yulun Du, Tao Jiang, Chao Hong, Shaowei Liu, Weiran He, Enming Yuan, Yuzhi Wang, Zhiqi Huang, Huan Yuan, Suting Xu, Xinran Xu, Guokun Lai, Yanru Chen, Huabin Zheng, Junjie Yan, Jianlin Su, Yuxin Wu, Neo Y. Zhang, Zhilin Yang, Xinyu Zhou, Mingxing Zhang, and Jiezhong Qiu. MoBA: Mixture of block attention for long-context llms, 2025. URL https://arxiv.org/abs/2502.13189.

Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation — a KVCache-centric architecture for serving LLM chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pp. 155–170, Santa Clara, CA, February 2025. USENIX Association. ISBN 978-1-939133-45-8. URL https://www.usenix.org/conference/fast25/presentation/qin.

Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. Quest: Query-aware sparsity for efficient long-context llm inference, 2024. URL https://arxiv.org/abs/2406.10774.

Jun Wang, Yunxiang Yao, Wenwei Kuang, Runze Mao, Zhenhao Sun, Zhuang Tao, Ziyang Zhang, Dengyu Li, Jiajun Chen, Zhili Wang, et al. Omniinfer: System-wide acceleration techniques for optimizing llm serving throughput and latency. *arXiv preprint arXiv:2511.22481*, 2025.

Chong Wu, Jiawang Cao, Renjie Xu, Zhuoheng Ran, Maolin Che, Wenbo Zhu, and Hong Yan. DuSA: Fast and accurate dual-stage sparse attention mechanism accelerating both training and inference. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 41087–41113. Curran Associates, Inc., 2025a. doi: 10.52202/085713-1372. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/3ab868033387edbdd775b8edd78ed056-Paper-Conference.pdf.

Chong Wu, Maolin Che, Renjie Xu, Zhuoheng Ran, and Hong Yan. ELFATT: Efficient linear fast attention for vision transformers. In *Proceedings of the 33rd ACM International Conference on Multimedia*, MM '25, pp. 9140–9149, New York, NY, USA, 2025b. Association for Computing Machinery. ISBN 9798400720352. doi: 10.1145/3746027.3754825. URL https://doi.org/10.1145/3746027.3754825.

Chong Wu, Maolin Che, and Hong Yan. The cur decomposition of self-attention matrices in vision transformers. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 48(4):4792–4809, 2026a. doi: 10.1109/TPAMI.2025.3646452.

Chong Wu, Zhenan Feng, Renjie Xu, Houwang Zhang, Jiawang Cao, Maolin Che, Wenbo Zhu, and Hong Yan. BFLA: Block-filtered long-context attention mechanism, 2026b. URL https://arxiv.org/abs/2605.12193.

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.

Zhiqiang Xie, Ziyi Xu, Mark Zhao, Yuwei An, Vikram Sharma Mailthody, Scott Mahlke, Michael Garland, and Christos Kozyrakis. Strata: Hierarchical context caching for long context language model serving. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*, pp. 1–16, Seattle, WA, July 2026. USENIX Association. ISBN 978-1-939133-55-7. URL https://www.usenix.org/conference/osdi26/presentation/xie-zhiqiang.

Anyi Xu, B Li, Bangcai Lin, Bing Xue, BingCheng Xian, Bingzheng Xu, Bochao Wu, Bowei Zhang, Boyi Deng, CC Yu, et al. Deepseek-v4. 1-flash: Pushing the limits of kv cache compression. *arXiv preprint arXiv:2609.19969*, 2026.

Jiawei Yi, Ping Gong, Youhui Bai, Zewen Jin, Shengnan Wang, Jiaqi Ruan, Jia He, Jiaan Zhu, Pengcheng Wang, Haibo Wang, Weiguang Wang, Xia Zhu, and Cheng Li. LiteCache: A query similarity-driven, gpu-centric kvcache subsystem for efficient llm inference, 2026. URL https://arxiv.org/abs/2511.14510.

Lingfan Yu, Jinkun Lin, and Jinyang Li. Stateful large language model serving with pensieve. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, pp. 144–158, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400711961. doi: 10.1145/3689031.3696086. URL https://doi.org/10.1145/3689031.3696086.

Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, Y. X. Wei, Lean Wang, Zhiping Xiao, Yuqing Wang, Chong Ruan, Ming Zhang, Wenfeng Liang, and Wangding Zeng. Native sparse attention: Hardware-aligned and natively trainable sparse attention, 2025. URL https://arxiv.org/abs/2502.11089.

Weilin Zhao, Zihan Zhou, Zhou Su, Chaojun Xiao, Yuxuan Li, Yanghao Li, Yudi Zhang, Weilun Zhao, Zhen Li, Yuxiang Huang, Ao Sun, Xu Han, and Zhiyuan Liu. InfLLM-V2: Dense-sparse switchable attention for seamless short-to-long adaptation, 2025. URL https://arxiv.org/abs/2509.24663.

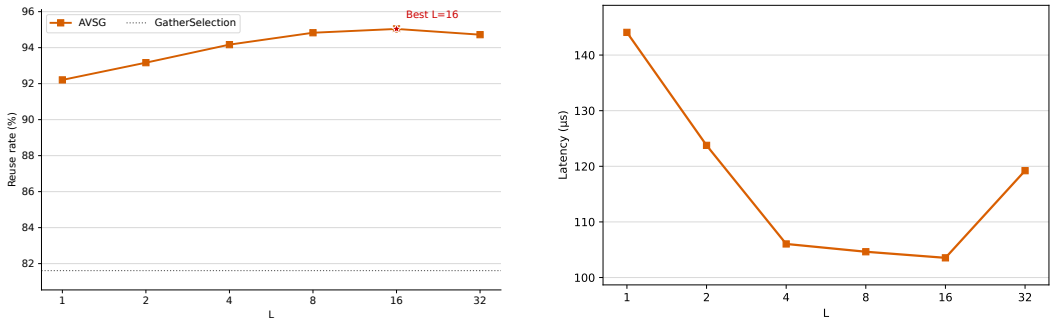
Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 62557–62583. Curran Associates, Inc., 2024. doi: 10.52202/079017-2000. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/724be4472168f31ba1c9ac630f15dec8-Paper-Conference.pdf.

Zhipu AI. Glm-5.3. Hugging Face model card, 2026. URL https://huggingface.co/zhipuai/GLM-5.3.

Qihui Zhou, Peiqi Yin, Pengfei Zuo, and James Cheng. SparseServe: Unlocking parallelism for dynamic sparse attention in long-context llm serving, 2025. URL https://arxiv.org/abs/2509.24626.

A MAX LIFETIME ABLATION

AVSG Lifetime Ablation. We evaluate AVSG record lifetime using traces from a representative transformer layer for the same 2000 real chat requests as §4.5, with Top-$K = 2048$, an 8K-entry selection pool, and MTP-0. We sweep $L \in \{1, 2, 4, 8, 16, 32\}$ over the same set of steady decoding steps from these requests. As shown in Figures 5a and 5b, the request-averaged reuse rate increases from 92.21% at $L = 1$ to 94.83% at $L = 8$ and peaks at 95.04% for $L = 16$, compared with 81.61% for GatherSelection. Meanwhile, the mean kernel latency decreases from 104.62 μs to its minimum of 101.27 μs at $L = 16$. Both metrics regress slightly at $L = 32$. Therefore, we select $L = 16$ as the default lifetime.



(a) Reuse rate.

(b) Mean kernel latency.

Figure 5: Sensitivity of AVSG to the maximum record lifetime L using the 2000 real requests of §4.5 with Top-$K = 2048$, an 8K-entry selection pool, and MTP-0.

B CONSOLIDATED ALGORITHM DETAILS

This appendix gathers the three algorithms referenced but omitted from the main text: the vectorized state update that installs misses into the hash table, the per-step life-cycle update that consolidates the four lifetime operations of Algorithm 1, and the MTP-aware processing that sequences Algorithm 1 over the initial token and all speculative tokens of one decode step.

Probe: hit masks and bucket indices. Given a token's selected indices $\boldsymbol{x} = (x_0, \ldots, x_{K-1})$ and the request's hash state table $\boldsymbol{s}$, the read-only primitive returns

$$(\boldsymbol{m}, \boldsymbol{h}) = \mathrm{Probe}(\boldsymbol{x}, \boldsymbol{s}), \qquad \boldsymbol{m} = (m_0, \ldots, m_{K-1}), \quad \boldsymbol{h} = (h_0, \ldots, h_{K-1}), \tag{8}$$

where $m_i \in \{0, 1\}$ indicates whether probing finds a matching selected index, and $h_i \in \{0, \ldots, S - 1\}$ is the matching bucket on a hit or the final examined bucket on a miss. In particular, h_i is a hash bucket index, not the destination slot o_i.

For each i, initialize the bucket and odd probe stride as

$$h_i^{(0)} = (a_1 x_i) \bmod S, \qquad d_i = ((a_2 x_i) \bmod S) \mid 1, \tag{9}$$

where $\mid$ denotes bitwise OR. The superscript r denotes the probe round. At each round $r = 0, \ldots, R$, decode the candidate's slot field and compare its stored selected index:

$$o_i^{(r)} = q_{h_i^{(r)}} \bmod S, \qquad m_i^{(r)} = \left[c_{o_i^{(r)}} = x_i\right]. \tag{10}$$

Here $o_i^{(r)}$ is the candidate slot, not an allocated destination for a miss. Square brackets around a proposition denote its Boolean indicator. For $r < R$, only unresolved probes advance:

$$h_i^{(r+1)} = \begin{cases} h_i^{(r)}, & m_i^{(r)} = 1, \\ (h_i^{(r)} + d_i) \bmod S, & m_i^{(r)} = 0. \end{cases} \tag{11}$$

Since Probe does not modify the table, a matched probe remains matched. The returned values are $m_i = m_i^{(R)}$ and $h_i = h_i^{(R)}$, requiring at most $R + 1$ bucket comparisons per selected index. On a

hit, the caller recovers $o_i = q_{h_i} \bmod S$; on a miss, h_i does not identify a reusable KV entry, and allocation supplies o_i separately. Hit detection compares the full selected index; the invalid and fingerprint fields govern allocation rather than this comparison. All recurrences apply element-wise across the K probes.

Algorithm 2 UpdateState (vectorized)

Input: $\boldsymbol{t}$: selected indices that miss; $\boldsymbol{o}^{\text{miss}}$: allocated cache positions for $\boldsymbol{t}$; $\boldsymbol{s} = (E_0, E_1, \ldots, E_{S-1})$: state table
Output: updated $\boldsymbol{s}$; all misses retain their allocated output positions

$\boldsymbol{h} \leftarrow (a_1 \cdot \boldsymbol{t}) \bmod S$ # initial hash buckets
$\boldsymbol{d} \leftarrow ((a_2 \cdot \boldsymbol{t}) \bmod S) \mid 1$ # odd probe strides
for $i = 0$ **to** R :
  $\boldsymbol{v} \leftarrow \text{reusable}(q_{\boldsymbol{h}})$ # Eq. 4
  if $i < R$:
    $\boldsymbol{h}[\neg \boldsymbol{v}] \leftarrow (\boldsymbol{h}[\neg \boldsymbol{v}] + \boldsymbol{d}[\neg \boldsymbol{v}]) \bmod S$ # advance only unresolved probes
$E_{\boldsymbol{o}^{\text{miss}}}.c \leftarrow \boldsymbol{t}$ # all misses have allocated slots
$q_{\boldsymbol{o}^{\text{miss}}}.\text{lifetime} \leftarrow L$
$\widehat{\boldsymbol{\phi}} \leftarrow \widehat{\phi}(\boldsymbol{t})$ # encoded fingerprints of selected indices that miss
$E_{\boldsymbol{h}[\boldsymbol{v}]}.q \leftarrow 2FS\, q_{\boldsymbol{h}[\boldsymbol{v}]}.\text{lifetime} + \widehat{\boldsymbol{\phi}}[\boldsymbol{v}] + \boldsymbol{o}^{\text{miss}}[\boldsymbol{v}]$
One packed mapping per bucket: preserve lifetime, update fingerprint/slot, clear invalid
Same-bucket writes retain one nondeterministic writer; unresolved probes install no mapping

Collision and probe-budget semantics. The last assignment is a packed-word scatter. For competing writes to the same bucket, the surviving word must contain one writer's complete fingerprint and destination position, rather than fields from different writers. This expresses the hash write-collision behavior without a separate deterministic winner-selection step. The final mask $\boldsymbol{v}$ selects probes that found a reusable bucket within rounds 0–R; no write is issued for exhausted probes. Misses whose mappings are overwritten or never installed still keep their allocated slots and are copied for the current operation. Subsequent probes may miss these entries and fetch them again; a hit is accepted only after comparing the full cached token id, so losing a mapping does not create a false hit. The fingerprint belongs to the hash bucket $E_h.q$, while the cached id and its lifetime belong to the allocated slot $E_{o^{\text{miss}}}$, as defined in §3.2.

CollectDead. Given the hash state table $\boldsymbol{s} = (E_0, \ldots, E_{S-1})$, $\text{CollectDead}(\boldsymbol{s})$ returns the indices of slots whose lifetime has expired:

$$\boldsymbol{e} = \text{CollectDead}(\boldsymbol{s}) = \big[j \mid 0 \le j < S,\ q_j.\text{lifetime} < 0\big], \tag{12}$$

where the brackets denote a compact vector of slot indices in increasing order, each included once. The function compares all slot lifetimes with zero and compacts the matching indices without modifying the table. Here, n_g denotes the number of selected indices that miss for the current token g. Thus, $|\boldsymbol{e}|$ is the number of reclaimable slots, and $\boldsymbol{e}[:n_g]$ selects the first n_g slots once $|\boldsymbol{e}| \ge n_g$. A slot with lifetime zero is not yet reclaimable; further decay and state updates are performed by the caller.

Algorithm 3 Life-Cycle Update (per token)

Input: $\boldsymbol{s} = (E_0, E_1, \ldots, E_{S-1})$: state table; L: maximum record lifetime; $\boldsymbol{o} = (o_0, \ldots, o_{K-1})$: slots corresponding to the current token's K selected indices, with hit positions populated; g: token index within the request; $\boldsymbol{m} = (m_0, \ldots, m_{K-1})$: hit mask aligned with those selected indices; $\boldsymbol{o}^{\text{miss}}$: newly allocated slots

if $g = 0$:
  $\boldsymbol{s}.\text{lifetime} \leftarrow \boldsymbol{s}.\text{lifetime} - 1$ # Step 1: Global pre-decay
$q_{\boldsymbol{o}[\boldsymbol{m}]}.\text{lifetime} \leftarrow L$ # Step 2: Refresh hits
Step 3: Ensure $\ge |\boldsymbol{o}^{\text{miss}}|$ dead slots
$\boldsymbol{e} \leftarrow \text{CollectDead}(\boldsymbol{s})$; **while** $|\boldsymbol{e}| < |\boldsymbol{o}^{\text{miss}}|$: $\boldsymbol{s}.\text{lifetime} \leftarrow \boldsymbol{s}.\text{lifetime} - 1$; $\boldsymbol{e} \leftarrow \text{CollectDead}(\boldsymbol{s})$
Step 4: Initialize newly allocated slots
$q_{\boldsymbol{o}^{\text{miss}}}.\text{lifetime} \leftarrow L$

Slot preservation under sequential token processing. Within each request, tokens execute Passes 1–3 sequentially. Global pre-decay runs only for $g = 0$, leaving every old slot with lifetime at most $L - 1$. Each token refreshes its hits and initializes its newly allocated slots to L before the next token is processed; slots selected only by later tokens are not refreshed in advance.

To bound reclamation across the request, consider the cumulative unit decays performed during allocation. After L such decays, every old slot that has never been refreshed is reclaimable. Any slot refreshed or initialized by a processed token still has nonnegative lifetime, since it has undergone at most L decays since being set to L. Before allocating the current token's misses, the number of slots already required by processed selections plus those misses is at most $N_q K \leq S$. Thus, by the Lth cumulative decay, enough other slots are reclaimable, and no further decay is needed. All output slots already assigned within the request therefore remain available until Stage Two and attention consume them. An old entry selected only by a later token may be reclaimed earlier; that token then treats it as a miss and fetches it again. This policy provides lifetime-based aging rather than exact LRU replacement.

Algorithm 4 MTP-Aware AVSG Processing (per decode step)

Input: $\boldsymbol{X} \in \mathbb{Z}^{(N_{\text{mtp}}+1)\times K}$: top-$K$ index matrix; $\boldsymbol{s} = (E_0, E_1, \ldots, E_{S-1})$: shared state table; N: verified context length (excluding speculative MTP tokens)
Output: $\boldsymbol{O} \in \mathbb{Z}^{(N_{\text{mtp}}+1)\times K}$: buffer position matrix; updated $\boldsymbol{s}$

```
# Stage One: execute all three passes for each token in sequence
for g = 0 to N_q − 1 :
    Set x to row g of X and execute Algorithm 1 with token index g and table s
    Store o in row g of O and append this token's misses and destination slots to the transfer list
Synchronize all matching cores (SyncAll)
Execute Stage Two for the accumulated miss list (§3.4)
Complete H2D transfers before attention consumes O
# End of step: speculative invalidation
v ← (s.c ≥ N)
E_v.c ← −1 # mark speculative entries as empty
```

Speculative Invalidation. The N_{mtp} additional tokens are speculative: only those passing verification are committed, while rejected tokens trigger KV regeneration on the host in the next step. Retaining entries for rejected tokens would make hash matching return spurious hits—reading stale device kv instead of the regenerated ones, causing *silent accuracy degradation*. The end-of-step invalidation in Algorithm 4 prevents this by marking every entry whose cached id $c_j \geq N$ (the speculative range beyond the verified context length) as empty, forcing misses and fresh H2D transfers of the regenerated data. Invalidating *all* speculative entries—including those that may pass verification—costs at most $N_{\text{mtp}} \times K$ extra miss lookups in the next step, negligible against the correctness guarantee.

C Vectorized Hash Matching: Full Protocol and Results

We isolate Stage One and compare the vectorized open-addressing hash matching against the scalar dual-pointer implementation it replaces. Both variants run in an identical test harness and differ only in the matching algorithm, so any latency difference is attributable to the algorithm alone. We report the median `Duration` of the algorithm over 1000 consecutive iterations, collected with the Profiler Tools on a single NPU; median, trimmed mean, and arithmetic mean agree to within 2% at every operating point, so no conclusion below depends on the choice of statistic.

Matching Performance. Under a matched configuration in which both implementations face an identical working set, vectorized matching overtakes the scalar dual-pointer just above N=512 and is 2.80× faster at N=2048 (Figure 3(a)). The matched configuration is dictated by the baseline: the dual-pointer implementation couples its buffer capacity to the top-K length, admitting only $S = K$ and exposing no independent control over S. We therefore sweep $S = K = N$ for both variants—conceding nothing to the vectorized algorithm, whose cost is nearly independent of S (Figure 3(c)), while removing the one degree of freedom the baseline cannot exercise. Under this protocol the scalar cost grows linearly in the number of indices to match, from 20.8 μs at N=256

to $150.0\,\mu$s at $N{=}2048$, whereas over the same $8\times$ range the vectorized cost moves only from 38.2 to $53.7\,\mu$s. Below the crossover the scalar path is cheaper (the vectorized implementation carries a larger fixed overhead); above it the linear term dominates and the gap widens monotonically.

Decoupling from K and S. With S fixed at 4096 entries, an $8\times$ increase in K costs only $1.27\times$ more latency, from 46.0 to $58.6\,\mu$s (Figure 3(b))—the same $8\times$ increase raises the scalar cost by $7.2\times$; with K fixed at 512, a $16\times$ increase in S, from 512 to 8192, costs $1.60\times$, from 38.6 to $61.9\,\mu$s (Figure 3(c)). Because the hash probe is $O(1)$ SIMD per entry, lookup cost is governed by the retry budget R rather than by table capacity—this is the property AVSG depends on: the buffer can be enlarged to raise the hit rate and cut H2D traffic without a proportional matching penalty. The residual drift (larger K or S presents a longer vector to sweep, and added elements are not entirely free) matters only in ratio: latency rises roughly an order of magnitude more slowly than the volume of vector data matched, whereas the scalar path pays $7.2\times$ for an $8\times$ larger working set—close to proportional. The S dimension has no scalar counterpart to plot, since the dual-pointer admits only $S = K$; a baseline pursuing the same hit-rate benefit would have to raise K in lockstep and pay the linear cost of Figure 3(a).

D H2D Core-Binding Strategies: Definitions and Full Results

This appendix defines the three core-binding strategies compared in §4.3, analyzes their transfer workloads, and presents the experimental setup and full bandwidth results.

S1 (Request-level) assigns one core to each request, processing all miss KV entries within that request serially; parallelism is capped by the batch size B, leaving most cores idle when B is small.

S2 (Top-K position level) pools all top-K positions from all top-K index sets into a single array and splits it evenly across all C cores by fixed-width partition, so each core receives an equal share of positions regardless of their hit/miss status; cores issue transfers for every assigned position, with no-op transfers for hit positions wasting transfer capacity proportional to the cache hit rate.

S3 (Miss-KVs-level, AVSG) pools all cache misses into a single miss list and distributes them evenly across C cores (§3.4 of the main text); every issued transfer corresponds to a real miss KV, so no transfer capacity is wasted on hits.

Transfer imbalance analysis. Using the notation of §3.4 and the workload definition in Eq. 5, we analyze why request-level assignment and equal top-K partitioning can produce imbalanced H2D transfers.

Under request-level assignment, distribute the B requests as evenly as possible across the C cores, assigning one extra request to each of the first $r_B = B \bmod C$ cores. Each request contributes $N_q K$ selected-index positions, so

$$|\mathcal{A}_c^{\mathrm{req}}| = \begin{cases} \big(\lfloor B/C \rfloor + 1\big) N_q K, & 0 \le c < r_B, \\ \lfloor B/C \rfloor N_q K, & r_B \le c < C. \end{cases} \tag{13}$$

When B is not divisible by C, $|\mathcal{A}_c^{\mathrm{req}}|$ varies across cores, which can lead to imbalanced transfer workloads W_c. To balance assigned positions at a finer granularity, equal top-K partitioning flattens all $N_g K$ positions and divides them into near-equal contiguous ranges:

$$\mathcal{A}_c^{\mathrm{pos}} = \left\{ (g,i) : \left\lfloor \frac{cN_gK}{C} \right\rfloor \le gK + i < \left\lfloor \frac{(c+1)N_gK}{C} \right\rfloor \right\}, \tag{14}$$

where $0 \le g < N_g$ and $0 \le i < K$. These sets differ in size by at most one, but cache-hit counts may still differ across cores. For two distinct cores $c, d \in \{0, \ldots, C-1\}$ with $|\mathcal{A}_c^{\mathrm{pos}}| = |\mathcal{A}_d^{\mathrm{pos}}|$, their transfer-workload difference is

$$W_c - W_d = \sum_{(g,i) \in \mathcal{A}_d^{\mathrm{pos}}} m_{g,i} - \sum_{(g,i) \in \mathcal{A}_c^{\mathrm{pos}}} m_{g,i}. \tag{15}$$

Thus, differences in cache-hit counts directly produce transfer imbalance; a partition containing only hits has $W_c = 0$ even if another partition still requires transfers.

Experimental setup. To evaluate these strategies, we select batch sizes $B \in \{1, 2, 4, 8, 16, 32\}$ from the source CSV and cache reuse rates $r \in \{0, 0.25, 0.5, 0.75, 1\}$, with miss rate $m = 1 - r$,

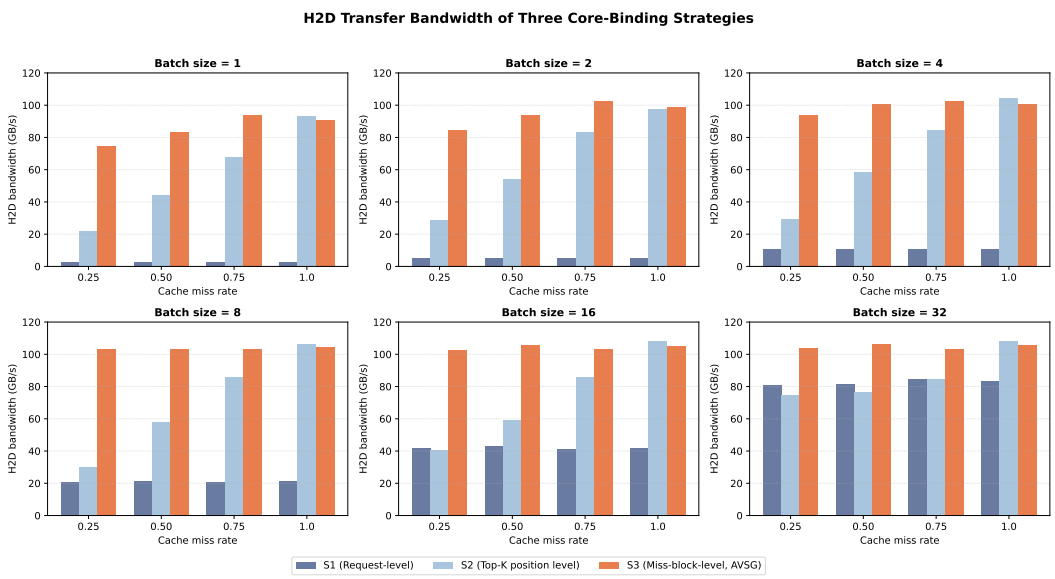


Figure 6: Effective H2D bandwidth (GB/s) of three core-binding strategies under varying batch sizes ($B = 1, 2, 4, 8, 16, 32$) and cache miss rates ($m = 0.25, 0.5, 0.75, 1.0$). Each subplot corresponds to one batch size. Within each subplot, the three bars per miss rate represent S1 (request-level binding), S2 (Top-K position level binding), and S3 (miss-KV-level binding, AVSG). Higher bandwidth indicates more efficient use of the transfer engines. Experimental configuration: top-$K = 2048$, $N_{\text{mtp}} = 0$, $H = 1$, per-token payload $(64 + 512) \times 2 = 1152$ bytes.

$K = 2048$, $N_{\text{mtp}} = 0$ (one initial token per request, no speculative tokens), $H = 1$, and 1152 bytes per selected token.

Metric. For each (strategy, batch size, miss rate) combination, we compute the effective H2D bandwidth in GB/s. Only miss cache are transferred, so the number of moved tokens is $B\,(N_{\text{mtp}} + 1)\,H\,K\,m$, and at 1152 bytes per token the transfer volume is $V = 1152\,B\,(N_{\text{mtp}} + 1)\,H\,K\,m$ bytes. Dividing V by the measured wall-clock transfer time t yields the effective bandwidth V/t.

Speedup source data. Table 3 reports the CSV columns `BW_S1_baseline_GBps`, `BW_S2_master_GBps`, and `BW_S3_balance_GBps`, with speedups computed as S3/S1 and S3/S2 before rounding. For every nonzero transfer volume this is also the inverse ratio of transfer times. At 100% reuse the transfer volume is zero; kernel runtimes may remain nonzero, but they do not define a bandwidth ratio.

Full results. Figure 6 compares absolute bandwidth across the tested batch sizes and nonzero miss rates. Table 3 provides the full measurements and speedup ratios, including the zero-transfer case at 100% reuse.

Bandwidth trends. S1's effective bandwidth scales almost linearly with batch size—from 2.7 GB/s at B=1 to ∼83 GB/s at B=32—because its assignment granularity is the request, so parallelism cannot exceed B; even at B=32 it trails S3 by 20–25 GB/s since misses within each request remain serialized. S2 matches S3 when m=1.0 (every position is a genuine miss; 108 vs. 104.9 GB/s at B=16, the small excess reflecting its simpler equal-width partition), but as m decreases its cores issue no-op transfers for hit positions: at B=16, m=0.25, S2 drops to 40.5 GB/s while S3 holds 102.7 GB/s. S3's dynamic work partition ensures every issued transfer points to a real miss KV, and the work is spread evenly across all C engines regardless of batch size or miss rate.

Relation to the main-text comparison. Figure 4 uses S3/S1 on a logarithmic vertical axis and S3/S2 on a linear vertical axis; both show categorical batch sizes 1–32 and reuse rates of 0–75%, with a dashed line at equal bandwidth. The largest gains are 34.44× at $B = 1$, $r = 0.25$ against request-level assignment and 3.48× at $B = 8$, $r = 0.75$ against equal top-K partitions. Equal numbers of assigned positions do not guarantee equal numbers of misses: cache hits can leave different amounts of useful transfer work on different cores, while hit positions also incur no-op issue

Table 3: S1, S2, and S3 bandwidth comparison for batch sizes 1 to 32. Bandwidth values retain the source precision (GB/s); speedups are S3/S1 and S3/S2, rounded to two decimals. The last row covers all six batch sizes at 100% reuse: no KV data are transferred, so bandwidth speedup is undefined.

Batch size	Reuse (%)	S1 (GB/s)	S2 (GB/s)	S3 (GB/s)	S3/S1	S3/S2
1	0	2.7164	93.0307	90.8505	33.45×	0.98×
1	25	2.7205	67.6668	93.7022	34.44×	1.38×
1	50	2.7351	44.4206	83.1734	30.41×	1.87×
1	75	2.6629	21.9380	74.7369	28.07×	3.41×
2	0	5.3596	97.4847	98.8166	18.44×	1.01×
2	25	5.4038	83.2732	102.2639	18.92×	1.23×
2	50	5.3269	53.9419	94.1647	17.68×	1.75×
2	75	5.2613	28.7019	84.4718	16.06×	2.94×
4	0	10.6228	104.2751	100.6043	9.47×	0.96×
4	25	10.6920	84.4470	102.3319	9.57×	1.21×
4	50	10.8849	58.7102	100.6075	9.24×	1.71×
4	75	10.8392	29.1469	94.0896	8.68×	3.23×
8	0	21.4230	106.6558	104.2904	4.87×	0.98×
8	25	20.8603	85.8649	102.9668	4.94×	1.20×
8	50	21.5623	58.0890	103.2277	4.79×	1.78×
8	75	20.9256	29.8066	103.4507	4.94×	3.47×
16	0	41.6336	108.4312	104.9474	2.52×	0.97×
16	25	41.4463	85.8342	103.2037	2.49×	1.20×
16	50	43.0587	59.2252	105.5702	2.45×	1.78×
16	75	41.6196	40.4595	102.7434	2.47×	2.54×
32	0	83.1604	108.4119	105.4712	1.27×	0.97×
32	25	84.4773	84.4850	103.3840	1.22×	1.22×
32	50	81.7463	76.8601	106.5611	1.30×	1.39×
32	75	80.8333	75.0018	103.9796	1.29×	1.39×
All	100	0.0000	0.0000	0.0000	–	–

overhead in this baseline. AVSG instead partitions the actual miss list. The bandwidth measurements capture the combined effect of these costs, rather than isolating their individual contributions. At zero reuse, equal-position and miss-only partitioning have similar workloads and near-parity bandwidth; the full values are reported in Table 3.

E Cache Reuse: Capacity and MTP Results

This appendix reports the full synthetic protocol behind the selection-pool capacity study of §4.4. The synthetic experiments use a maximum KV length of 16K and five random seeds, with Top-K=2048, lifetime L=8, and MTP-0. We fix the adjacent-step similarity γ to 0.5, 0.7, or 0.9, while sampling historical recall from $\mathcal{N}(\mu_{\text{hist}}, 0.01^2)$ with $\mu_{\text{hist}} \in \{0.5, 0.7, 0.9\}$, clipped to $[0, 1]$. Table 4 lists the reuse rate of the 4K and 8K pools in all nine settings and on the 2000 real chat requests used in §4.5; the improvement is larger under low-to-moderate locality, where additional capacity preserves more reusable entries from earlier decoding steps, and gradually diminishes as the reuse rate approaches saturation.

Selection of main-text configurations. For each fixed adjacent-step reuse γ, we select the historical recall setting that maximizes the 8K-buffer hit rate minus 100γ percentage points. Among the three tested historical recall means, $\mu_{\text{hist}} = 0.9$ maximizes this gain at each $\gamma \in \{0.5, 0.7, 0.9\}$, giving gains of 44.15, 26.79, and 9.04 pp, respectively. These are best-case configurations within the sweep, not averages across historical recall settings.

Cold-start MTP protocol. On the same 2000 real chat requests used in §4.5, we fix $K = 2048$ and $S = 8$K and simulate MTP degrees 0–3, where degree $d = N_{\text{mtp}}$ includes the initial decode token and d additional speculative tokens. All speculative tokens are assumed accepted. Each request starts with an empty cache, and reuse is measured over all selected KV entries in its first decode iteration, then averaged across requests. Independent processing starts each token with an empty buffer and yields zero reuse. Shared-buffer processing records the first token's misses and their reserved slots for

Table 4: AVSG reuse rate (%) for Top-K=2048 with 4K and 8K selection pools. Synthetic: max KV length 16K, L=8, MTP-0, five seeds; adjacent-step similarity γ fixed, historical recall $\sim \mathcal{N}(\mu_{\text{hist}}, 0.01^2)$, clipped to $[0, 1]$. Real: request-level macro over the 2000 requests used in §4.5.

Data	γ	μ_{hist}	4K pool	8K pool
Synthetic	50.00%	0.5	60.00%	68.19%
Synthetic	50.00%	0.7	70.44%	81.27%
Synthetic	50.00%	0.9	84.29%	94.15%
Synthetic	70.00%	0.5	74.76%	82.55%
Synthetic	70.00%	0.7	81.94%	89.99%
Synthetic	70.00%	0.9	90.12%	96.79%
Synthetic	90.00%	0.5	88.90%	94.91%
Synthetic	90.00%	0.7	92.61%	97.02%
Synthetic	90.00%	0.9	96.21%	99.04%
Real	81.61%	–	88.72%	94.83%

subsequent tokens, then transfers all misses after matching; the first token itself still has zero reuse. The resulting request-averaged reuse rates are 0%, 38.71%, 53.79%, and 61.87% for degrees 0–3, respectively (Figure 7).

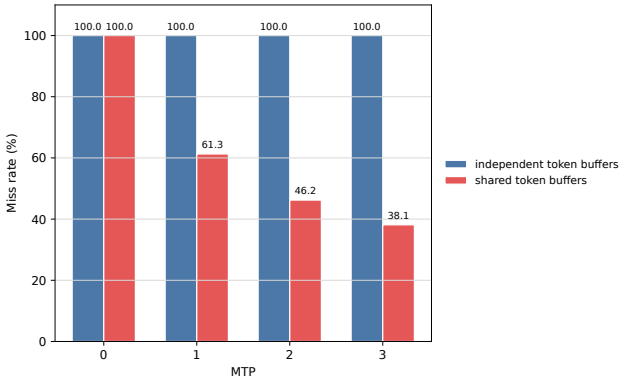


Figure 7: Cold-start cache miss rate with shared and independent token buffers. Sharing the cache buffer across MTP tokens enables intra-iteration reuse and progressively reduces the miss rate as the MTP degree increases.

F Metadata-Word Decode Chain

This appendix expands the inline decode of §3.3 into its full equation form. For the packed metadata word q_h stored in bucket h (Eq. 3 of the main text), AVSG first extracts the index field:

$$u_h = q_h \bmod (2FS). \tag{16}$$

The invalid flag, mapped cache position, and stored fingerprint are then decoded as

$$b_h = \left\lfloor \frac{u_h}{FS} \right\rfloor, \qquad o_h = u_h \bmod S, \qquad \phi_h = \left\lfloor \frac{u_h \bmod (FS)}{S} \right\rfloor, \tag{17}$$

and since $0 \le u_h < 2FS$, the decoded invalid flag satisfies $b_h \in \{0, 1\}$. A bucket contains a valid mapping iff

$$\text{valid}(q_h) = [b_h = 0], \tag{18}$$

which feeds the reusable predicate of Eq. 4 in the main text.

G End-to-End Serving: Full Experimental Setting

Serving platform. All settings run on a 12-node Ascend 910B NPU (64 GB HBM per chip) cluster with a 5P1D dis-aggregated topology—5 prefill instances (each 2 nodes, TP=16, EP=16) plus 1

decode instance (2 nodes, DP=16, EP=16)—serving GLM-5.3-w4a8c16 with 3-token speculative decoding. The workload is an agent dataset with a total of 2000 real chat requests collected from a real-world production system, with an average prompt length of 48k tokens. We evaluate with a maximum output of 32768 tokens, a concurrency of 64, and a request rate of 1. The benchmark script extends from (Wang et al., 2025) to maintain a fixed concurrency or request rate throughout the evaluation, so that the sustained load drives the system toward its throughput upper bound; as throughput rises, TTFT grows accordingly. Each setting starts from a fresh server with an empty prefix cache, and all settings serve identical request streams. In the offload settings, MLA latent KV is swapped to host DRAM via hugepage mmap into a 632 GB pool, while HBM retains the DSA retrieval structure plus a device-side KV window.

Memory layout. Because the three settings differ in where KV lives, we verify the layout from server logs rather than assuming it. After model loading, the profile stage leaves 16.22 GB HBM available per chip under No Offload versus 19.50 GB under both offload settings (identical code path), which caps No Offload's device KV pool at 145,536 tokens versus 762,496 tokens ($5.2\times$) for offload—and caps its supported context at 145,536 versus 524,288 tokens. Crucially, AVSG and KV offload share the identical memory layout; the only difference between them is the access strategy (per-step full H2D versus AVSG's hit reuse with miss-only transfer), so the performance gap measured between them is attributable to gather selection alone.

H Future Directions

Matrix-Level Vectorization. While AVSG currently employs 256-bit SIMD vectorization for hash matching, the inherent parallelism in sparse token lookup could be further exploited through matrix-level operations. By reformulating the matching algorithm as batched matrix computations, we could leverage tensor cores and matrix multiplication units on modern accelerators, potentially achieving even higher throughput than the current 200GB/s.

Decoupled State Update with Latency Hiding. Profiling reveals that most latency in Stage One matching comes from state record updates (lifetime counter maintenance for LRU eviction). Critically, these state updates do not affect the subsequent attention computation flow—they only influence future cache eviction decisions. This dependency structure enables two optimizations: (1) overlap state record updates with Stage Two data transfer or attention computation, effectively hiding the update latency, and (2) compute attention for cache-hit tokens immediately while simultaneously loading cache-miss tokens in parallel, reducing the critical path latency for mixed hit/miss scenarios.

Additional Directions. Other promising directions include: (1) adaptive hash table sizing based on workload characteristics and top-K patterns, (2) learned prefetching policies that predict future access patterns from attention distributions, (3) cross-layer optimization with DSA's token selection to minimize transfer latency, (4) support for dynamic sparse attention patterns that change during generation, and (5) extensions to multi-tier memory hierarchies including NVMe and CXL-attached memory. These enhancements could further improve the efficiency of sparse attention serving systems beyond DeepSeek Sparse Attention.