BITLADDER: BIT-LEVEL SELECTIVE LOADING OF THE KV-CACHE FOR EFFICIENT LONG-CONTEXT LLM INFERENCE

Anonymous authors
Paper under double-blind review

ABSTRACT

Long-context large language models (LLMs) repeatedly load a growing key-value (KV) cache during autoregressive decoding, making attention memory-bandwidth bound. Existing methods reduce this traffic through query-aware sparse attention or KV-cache quantization, but neither aligns loading with current token importance: sparse attention uses a coarse keep-or-skip mask that cannot differentiate retained contributions and may discard the aggregate contribution of long-tail tokens, whereas quantization uses static bit-width assignments that cannot track query-dependent importance. Our analysis shows that token importance is graded and changes across decoding steps. We present BitLadder, an algorithm-system co-designed framework for bit-level selective loading that dynamically adapts loaded KV-cache bit widths to token importance at each decoding step. To support dynamic bit widths without storing multiple quantized copies or performing per-step requantization, we propose truncation-consistent nested quantization (TCN-Quant), which encodes the KV-cache once and derives lower-bit representations by truncating the same code. To translate fewer loaded bits into practical speedup, we further design PlaneWeave, a memory layout that stores this nested representation so that a GPU kernel loads only the bits each token needs and supports efficient mixed-precision execution. We evaluate BitLadder on long-context understanding (LongBenchV2, MRCR) and reasoning (GPQA, MATH500, AIME25) benchmarks. At matched accuracy, BitLadder achieves 3.0× and 2.3× speedups over full attention and the state-of-the-art baseline, respectively.

1 INTRODUCTION

Long-context tasks are becoming increasingly prevalent, driving the deployment of long-context large language models (LLMs) (Anthropic, 2025; gpt, 2026; Cao et al., 2026; Kimi Team et al., 2026; OpenAI, 2025; DeepSeek-AI, 2025) ranging from document-level analysis (An et al., 2024; Delaunay et al., 2023) to repository-level code completion (Liu et al., 2023; Zhang et al., 2023a). However, increasing the context length also enlarges the key-value (KV) cache that autoregressive decoding must access. At each decoding step, because the current query attends to the cached keys and values of preceding tokens, KV-cache traffic per step grows linearly with context length. For long contexts, this repeated data movement makes KV-cache access a critical serving bottleneck (Tang et al., 2024; Ribar et al., 2023). Therefore, reducing the KV-cache bits loaded per step is essential for efficient long-context serving.

Existing work reduces KV-cache loading along two main directions. First, query-aware sparse attention methods estimate the importance of KV-cache entries under the current query and compute attention only over selected entries (Sun et al., 2025; Singhania et al., 2024; Liu et al., 2025). However, the sparse mask can only keep or skip entries at its selection granularity: retained entries are still read at the native bit width without further differentiating their contributions, whereas omitted entries do not participate in attention even when many weak contributions are non-negligible in aggregate. Second, KV-cache quantization reduces the bits transferred per token through uniform or mixed bit-width representations (Yang et al., 2024; Dong et al., 2024; Boroujeni et al., 2026). Uniform quantization assigns every token the same bit width and therefore ignores differences in token importance. Importance-aware mixed-bit methods assign different bit widths, but most assignments

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.

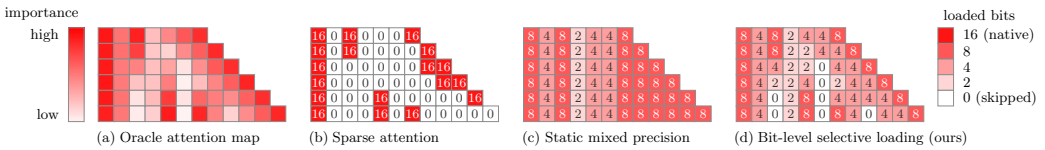


Figure 1: Different policies to reduce KV-cache loading. Bit-level selective loading aligns the loaded bit width with the token importance under each query.

are fixed before future queries reveal how token importance changes. Consequently, a token whose importance declines may retain an unnecessarily high bit width, while one that later becomes important remains limited by an earlier low-bit assignment. As shown in Figure 1, sparse attention is limited by its keep-or-skip mask, whereas static quantization is limited by fixed or preassigned bit widths. Both can distort the attention output and degrade model accuracy.

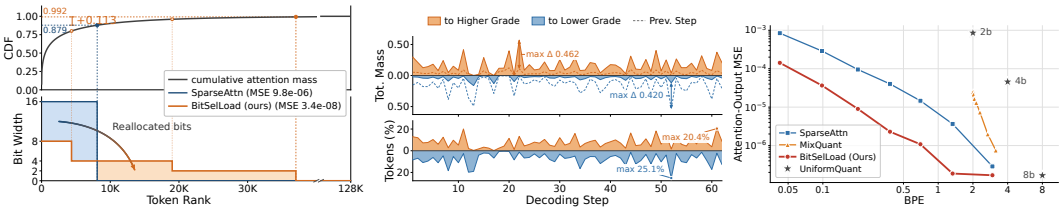
To address these limitations, we introduce bit-level selective loading, which dynamically assigns a loaded bit width to the KV-cache entry of each token according to its importance under the current query. Specifically, it maps token importance to one of several candidate bit widths, assigning more bits to more important tokens and progressively fewer bits as importance decreases, down to zero bits for negligible tokens. By aligning the loading budget with current token importance, this fine-grained bit-level allocation reduces memory traffic while retaining token contributions that affect model accuracy. However, a naive implementation must either store a separately quantized KV-cache copy for every candidate bit width or requantize KV entries at every decoding step. The former multiplies storage overhead, while the latter introduces prohibitive runtime overhead. The challenge is to support dynamic bit-width loading with low storage and quantization overhead.

To address this challenge, we present BitLadder, a framework for bit-level selective loading of the KV-cache during long-context decoding. First, we introduce truncation-consistent nested quantization (TCNQuant). TCNQuant quantizes the KV-cache once from native precision to the highest supported bit width. Any target lower-bit representation is then obtained by discarding the least significant bits (LSBs) and retaining the most significant bits (MSBs). The resulting truncated representation is strictly equivalent to directly quantizing the native-precision KV-cache to that target bit width. Therefore, this nested quantization supports efficient dynamic bit-width switching without storing multiple quantized copies or repeatedly requantizing the KV-cache during decoding. Second, we design PlaneWeave, a KV-cache memory layout tailored to the loading and computation pattern of nested quantization. The layout allows the GPU to fetch only the bits required by the target width and reduces the overhead for switching among bit widths. Consequently, it avoids wasting memory bandwidth and translates fewer loaded bits into greater practical speedups.

On Qwen3-8B (Yang et al., 2025) and DeepSeek-R1-Distill-Llama-8B (DeepSeek-AI, 2025) across different long-context understanding and reasoning benchmarks, BitLadder offers a better accuracy–efficiency trade-off than the evaluated sparse attention and quantization baselines, demonstrating that aligning loaded bit widths with query-dependent token importance reduces unnecessary KV-cache traffic while preserving important token contributions. Specifically, at matched accuracy, BitLadder achieves attention speedups of 3.0× over full attention and 2.3× over the state-of-the-art baseline.

Our contributions are summarized as follows.

- To our best knowledge, BitLadder is the first work of bit-level selective loading to select the loaded bit width of KV-cache dynamically at every decoding step.
- We introduce TCNQuant, a training-free truncation-consistent nested quantizer, together with PlaneWeave, a memory layout that serves this nested representation on tensor cores and thereby lets our GPU kernel execute dynamic bit-width loading efficiently.
- We evaluate our BitLadder on various long-context benchmarks, which demonstrate that our BitLadder can achieve 3.0× speedup over full attention and 2.3× speedup over state-of-the-art methods with full accuracy.



(a) Graded token importance. (b) Importance shifts across steps. (c) BPE versus MSE.

Figure 2: (a) Cumulative attention mass (top) and bit-width allocations (bottom) over tokens ranked by attention mass. (b) The total attention mass (top) and token fractions (bottom) for transitions to higher or lower importance grades between adjacent decoding steps. Dashed lines indicate the preceding-step mass. (c) Attention-output MSE versus KV-payload bits per element (BPE).

2 Background and Motivation

2.1 Long-Context LLM Inference and KV-cache Loading

LLM inference consists of the prefill stage and the decoding stage. After the prefill stage materializes the KV-cache, each Transformer layer repeatedly computes self-attention during autoregressive decoding. Consider one head of dimension d_h, where $\mathbf{k}_i$ and $\mathbf{v}_i$ are the cached key and value of preceding token i. At step t, the current query $\mathbf{q}_t$ attends to L_t preceding tokens:

$$a_{t,i} = \frac{\exp(\mathbf{q}_t^\top \mathbf{k}_i/\sqrt{d_h})}{\sum_{j=1}^{L_t} \exp(\mathbf{q}_t^\top \mathbf{k}_j/\sqrt{d_h})}, \qquad \mathbf{o}_t = \sum_{i=1}^{L_t} a_{t,i}\mathbf{v}_i. \tag{1}$$

Here, $a_{t,i}$ is the attention mass assigned to token i and indicates the token importance to the current query t (Zhang et al., 2023b; 2024), while $\mathbf{o}_t$ is the corresponding head output.

Full attention loads the keys and values of all L_t preceding tokens at every decoding step. With H_{kv} KV heads and a native element width of b_{native} bits, it loads $2H_{\text{kv}}d_h L_t b_{\text{native}}$ KV-payload bits per layer at step t, where the factor of two accounts for keys and values. This traffic grows linearly with L_t, making long-context decoding memory-bandwidth bound (Ribar et al., 2023; Lee et al., 2024).

2.2 Limitations of Existing KV-cache Loading Reduction Methods

Sparse attention exposes only a keep-or-skip decision To reduce KV-cache traffic, query-aware sparse attention selects a subset of cached keys and values for the current query and computes attention only over that subset (Tang et al., 2024; Chen et al., 2024; Hu et al., 2025; Dehghankar & Asudeh, 2026). However, Figure 2a shows that attention mass varies smoothly across cached entries. This graded distribution does not match the keep-or-skip decision of query-aware sparse attention: retained entries are all loaded at the native bit width even when their contributions differ substantially, whereas omitted long-tail entries can still have a non-negligible cumulative contribution. More importantly, under the same total KV-cache loading, distributing the available bits across importance levels preserves the attention output more accurately (with roughly $288\times$ lower MSE in Figure 2a) than restricting each entry to either zero or native precision, while covering more attention mass (i.e., 0.113). *Implication 1: The loading precision should vary with graded importance rather than be restricted to the binary choice between zero and the native bit width.*

Static KV-cache quantization cannot track query-dependent token importance. KV-cache quantization reduces loading traffic using either a uniform low bit width or static importance-aware mixed bit widths across tokens (Yang et al., 2024; Dong et al., 2024; Boroujeni et al., 2026; Zhang et al., 2025). However, as we group tokens into importance grades, we observe bidirectional movement between adjacent decoding steps (Figure 2b), with up to 20.4% of tokens moving to higher grades and 25.1% to lower grades. These transitions can involve substantial attention-mass gains and losses (peaking at 0.46 and 0.42, respectively), so a token's earlier importance may no longer reflect its contribution to the current query. Under static KV-cache quantization, downward shifts leave stale high-bit assignments to unimportant tokens, whereas upward shifts confine newly important tokens to stale low-bit assignments. *Implication 2: The loaded bit width should be updated at each decoding step according to the current query.*

2.3 BIT-LEVEL SELECTIVE LOADING

Together, the two implications motivate bit-level selective loading: at each decoding step, current token importance determines the loaded bit width $b_{t,i}$ for cached token i, with $b_{t,i} = 0$ indicating that the token is not loaded. This allocation varies both across tokens and across decoding steps.

Experimental Analysis of Loading Policies. To evaluate this allocation, let T_t denote the KV-payload bits loaded by a policy at step t. We measure loading cost as KV-payload bits per element (BPE), $T_t/(2H_{\mathrm{kv}}d_hL_t)$, and attention-output distortion as mean squared error (MSE) relative to full attention. Policy-specific BPE derivations appear in Appendix A. We compare bit-level selective loading with full attention, query-aware sparse attention, uniform quantization, and static mixed-bit allocation.[1] Figure 2c shows that bit-level selective loading achieves lowest MSE at matched BPE among the evaluated sparse and quantized baselines.

Challenges of Bit-Level Selective Loading. This loading–distortion advantage does not automatically translate into practical speedup. First, *multiple bit widths must be served efficiently*: storing a separate KV-cache copy for each width increases memory footprint, whereas requantizing entries at each decoding step adds runtime overhead. This calls for a single KV-cache representation that supports low-overhead bit-width switching. Second, *the selected widths must translate into efficient GPU accesses*: fixed-precision layouts may fetch unused bits, while width switching, indexing, and dequantization can offset the benefit of reduced traffic. The memory layout and execution path must therefore support selective reads with low decoding overhead.

3 METHODOLOGY

3.1 OVERVIEW

To address the challenges in Section 2.3, we propose BitLadder, a framework for efficient bit-level selective loading of the KV-cache during long-context decoding. BitLadder performs three stages (Algorithm 1). First, TCNQuant quantizes each newly generated KV entry once and stores it as an 8-bit nested representation (Section 3.2), from which truncation exposes lower precisions. Note that KV entries generated during prefill are stored in the same form at the prefill stage. Second, we assign each token block a loading bit width based on the current query and $n_b \ll L$ pooled summaries, without reading the full KV-cache during allocation (Section 3.3). Third, the resulting allocation drives mixed-bit attention, for which we propose PlaneWeave (Section 3.4), a memory layout that stores the nested codes so that the GPU fetches only the bits the allocation asks for and delivers them to the tensor cores with little unpacking work, turning the reduced KV-cache traffic into practical attention speedup. Appendix D states the five component functions that Algorithm 1 invokes.

Algorithm 1 BitLadder decoding step at step t

Input: query $\boldsymbol{q}_t \in \mathbb{R}^{h_q \times d_h}$; new key and value $\boldsymbol{k}_t, \boldsymbol{v}_t \in \mathbb{R}^{h_{kv} \times d_h}$; summaries $\bar{\boldsymbol{K}} \in \mathbb{Z}_{2^{b_s}}^{h_{kv} \times n_b \times d_h}$ over the $n_b = \lceil L/S \rceil$ blocks of size $S=16$, with accumulator $\boldsymbol{a}$ for the tail block; KV-cache $\mathcal{C}$ at one 8-bit nested code per element; threshold p
Output: $\boldsymbol{o}_t \in \mathbb{R}^{h_q \times d_h}$; updated $\bar{\boldsymbol{K}}, \boldsymbol{a}, \mathcal{C}$
$\boldsymbol{c} \in \mathbb{Z}_{256}^{2 \times h_{kv} \times d_h} \leftarrow \textsc{TCNQuant}(\boldsymbol{k}_t, \boldsymbol{v}_t)$
$\mathcal{C} \leftarrow \textsc{Append}(\mathcal{C}, \boldsymbol{c})$
$\bar{\boldsymbol{K}}, \boldsymbol{a} \leftarrow \textsc{UpdateSummary}(\bar{\boldsymbol{K}}, \boldsymbol{a}, \boldsymbol{k}_t)$
$\boldsymbol{s} \in \mathbb{R}^{h_q \times n_b} \leftarrow \mathrm{Softmax}\big(\boldsymbol{q}_t \bar{\boldsymbol{K}}^\top / \sqrt{d_h}\big)$
$\boldsymbol{\pi} \in \mathbb{R}^{h_{kv} \times n_b} \leftarrow \mathrm{mean}_{\text{q-group}}(\boldsymbol{s})$
$\boldsymbol{b} \in \{0,2,4,8\}^{h_{kv} \times n_b} \leftarrow \textsc{TieredTopP}(\boldsymbol{\pi}, p)$
$\boldsymbol{o}_t \leftarrow \textsc{MixedBitAttn}(\boldsymbol{q}_t, \boldsymbol{b}, \mathcal{C})$
return $\boldsymbol{o}_t,\ \bar{\boldsymbol{K}},\ \boldsymbol{a},\ \mathcal{C}$

3.2 TRUNCATION-CONSISTENT NESTED QUANTIZATION

Problem formulation. We store each KV entry once at a parent precision of B bits and derive each positive target precision b from that code. We use $B = 8$ and $\mathbb{B} = \{0, 2, 4, 8\}$, where $b = 0$ denotes skipping a block and is excluded from the quantization and reconstruction rules below. Each key or value vector of one token and one KV head forms a separate quantization group. Let x denote an element of such a group and Q_b a b-bit quantizer that reuses the same group metadata across

[1] For all low-bit policies, we apply OSCAR's offline rotation by default throughout this paper to mitigate outlier effects (Zhou et al., 2026). Note that we leverage only its rotation mechanism, without adopting its mixed-precision allocation strategy.

precisions. For a bit-reduction operator $T_{B\to b}$, we require the following agreement between direct quantization and reduction of the parent code:

$$Q_b(x) = T_{B\to b}\big(Q_B(x)\big), \qquad \forall\, x \text{ in the group}, \quad b \in \mathbb{B},\ 0 < b \le B. \tag{2}$$

A low-cost realization of $T_{B\to b}$ is truncation, which retains only the b most significant bits (MSBs) of the parent code : $T_{B\to b}(q) = q \gg (B-b) = \lfloor q/2^{B-b} \rfloor$, where $\gg$ denotes a logical right shift. To obtain the resulting code, we can simply load the retained MSBs while leaving the discarded LSBs in memory. We compare the bit-access requirements of truncation and rounding in Appendix B.

Definition 1 (Truncation consistency)**.** *A quantizer family is truncation-consistent if it satisfies Equation* (2) *under* $T_{B\to b}(q) = q \gg (B-b)$.

Affine quantization is not truncation-consistent. KV-cache quantization commonly adopts the uniform affine quantizer, which maps a group element x onto uniform steps of size s_b and reconstructs it at a grid point. For observed group extrema $x_{\min}$ and $x_{\max}$ with range $R = x_{\max} - x_{\min}$, its encoding and reconstruction are

$$Q_b^{\text{aff}}(x) = \text{round}\Big(\frac{x - x_{\min}}{s_b}\Big), \qquad \hat{x} = x_{\min} + s_b Q_b^{\text{aff}}(x), \qquad s_b = \frac{R}{2^b - 1}. \tag{3}$$

However, direct quantization and truncation can assign different codes to the same value, violating Definition 1. As illustrated in the upper half of Figure 3, for $B = 4$, $b = 2$, and $x = x_{\min} + 0.78R$, direct quantization gives $Q_2^{\text{aff}}(x) = 2$, whereas the parent code $Q_4^{\text{aff}}(x) = 12$ truncates to $12 \gg 2 = 3$. This disagreement arises because the affine rounding boundaries do not align with the groups of 2^{B-b} consecutive parent codes induced by truncation. We provide the derivation and counts of disagreeing parent codes in Appendix B.1.

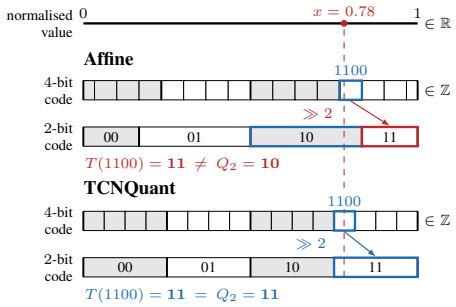


Figure 3: Truncation consistency at B=4 and b=2, where the affine grid staggers against the shaded truncation groups while TCNQuant coincides with them.

Truncation-consistent nested quantization. We therefore propose truncation-consistent nested quantization (TCNQuant), which partitions the group range into 2^b equal-width bins and assigns x the index of the bin containing it:

$$Q_b^{\text{TCN}}(x) = \min\Big(\Big\lfloor \frac{x - x_{\min}}{\Delta_b} \Big\rfloor, 2^b - 1\Big), \qquad \Delta_b \triangleq \frac{R}{2^b}, \tag{4}$$

where the saturation term handles the boundary case $x = x_{\max}$. Since $\Delta_b = 2^{B-b}\Delta_B$, every coarse bin is the exact union of 2^{B-b} consecutive fine bins, matching the partition that a right shift induces on their indices in the lower half of Figure 3. TCNQuant therefore satisfies Definition 1, with the interior and boundary cases proved in Appendix B.2. A single B-bit code per element thus serves every supported positive width using the same group metadata $(x_{\min}, \Delta_B)$.

Mid-point dequantization. Definition 1 constrains the integer code alone, whereas attention consumes reconstructed values, for which we choose the center of the corresponding bin:

$$D_b^{\text{TCN}}(q) = x_{\min} + \Big(q + \tfrac{1}{2}\Big)\Delta_b. \tag{5}$$

Because each coarse bin merges 2^{B-b} equally spaced fine bins, its center equals the arithmetic mean of their centers. We prove the mean-preserving property in Appendix B.3. This midpoint rule also bounds the error by $|x - D_b^{\text{TCN}}(q)| \le \Delta_b/2 = R/2^{b+1}$ for every value x assigned to code q.

3.3 Low-Precision Importance Estimation for Bit-Width Allocation

Block-level importance estimation. TCNQuant provides the supported positive widths, which we allocate according to the current query. Following query-aware block selection (Yuan et al., 2025), we group consecutive tokens into blocks $\mathcal{B}_j$ of size S per KV head. Their mean-pooled raw keys

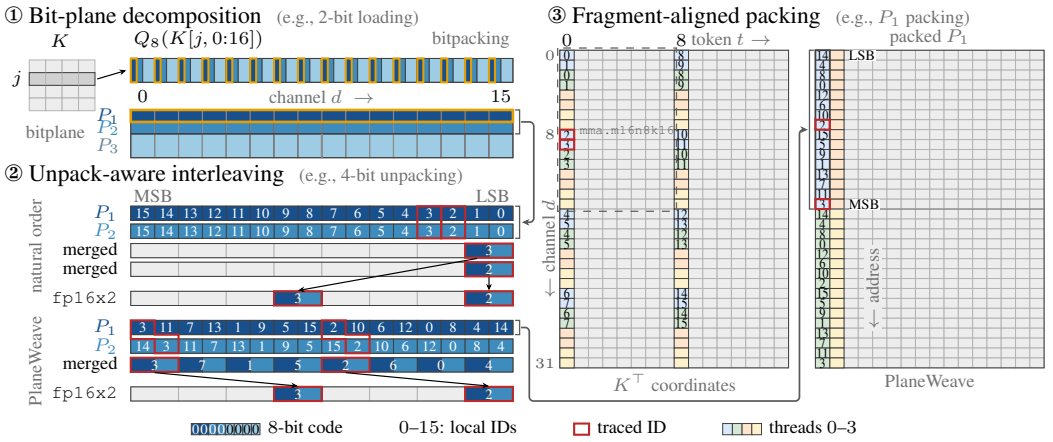


Figure 4: The design of PlaneWeave, where bit-plane decomposition makes each positive loading width a prefix of the stored planes, unpack-aware interleaving positions code pieces for reassembly, and fragment-aligned packing places each thread's MMA operands in packed words.

are separately quantized with TCNQuant at b_s bits. Let $\hat{\mathbf{k}}_j$ denote the reconstructed summary using its own group metadata. We compute

$$\bar{\mathbf{k}}_j = \frac{1}{|\mathcal{B}_j|} \sum_{i \in \mathcal{B}_j} \mathbf{k}_i, \qquad \pi_{t,j} = \mathrm{softmax}_j\left(\mathbf{q}_t^\top \hat{\mathbf{k}}_j / \sqrt{d_h}\right). \tag{6}$$

Here $\pi_{t,j}$ is an importance proxy to determine the importance of block $\mathcal{B}_j$. For grouped-query attention (GQA), we average these distributions over query heads sharing a KV head. Completed summaries remain fixed, while the tail block is updated from a full-precision running mean of raw keys without feeding quantized summaries back into that mean.

Tiered Top-P filtering. We extend binary Top-P selection (Lin et al., 2025) to assign block widths in $\{0, 2, 4, 8\}$. A per-tier fraction r sets cumulative proxy-mass targets $1-(1-r)^m$ for the $8/4/2$-bit tiers ($m = 1, 2, 3$), with final target $p = 1 - (1 - r)^3$. Thus p determines all three targets, with higher-importance blocks receiving higher widths. Keys and values share each block's width, with 0 bits denoting skipping. Summary maintenance and allocation details appear in Appendix D.

3.4 PlaneWeave: Memory Layout for the Nested KV-cache

Requirements on the memory layout. Turning the nested representation into practical attention speedup requires a layout suited to the attention kernel, which usually proceeds in three stages, loading the assigned bits, unpacking and dequantizing them, and computing attention on the recovered values, where each stage places one requirement on the layout. First, loads should fetch only the bits required by the assigned block width. Second, unpacking and dequantization should minimize instruction overhead to avoid offsetting the traffic saving. Third, the recovered values should be arranged for high-throughput tensor-core matrix-multiply-accumulate (MMA) computation. PlaneWeave addresses these requirements with bit-plane decomposition, unpack-aware interleaving, and fragment-aligned packing, respectively, as shown in Figure 4.

Bit-plane decomposition. A conventional bitpacking layout stores each element's code in one contiguous field, so reading the leading b bits also transfers the trailing bits that the assigned width discards. We instead group bits by significance rather than by element (Park et al., 2024), splitting each 8-bit parent code into $(2 + 2 + 4)$-bit segments from the MSB side and storing each segment as a separate tensor. The supported positive widths correspond to prefixes of these planes, with $b = 2$ reading the first tensor, $b = 4$ the first two, and $b = 8$ all three. As shown in Figure 4, for 16 elements along the head dimension, loading at 2 bits thus reads one 32-bit payload word in the bit-plane layout, compared with four when the 8-bit parent codes are packed contiguously.

Unpack-aware interleaving. Unpacking low-bit codes requires shifts and masks, while decomposition into planes adds the work of merging their pieces. We arrange bit positions when each entry is written so that pieces from different planes already occupy their offsets in the merged code, elim-

Table 1: Accuracy results on long-context understanding and reasoning benchmarks. LBV2 denotes LongBench v2, and GPQA-D denotes GPQA-Diamond.

	Long-Context Understanding				Long-Context Reasoning				
Method	**BPE↓**	**LBV2**	**MRCR**	**Avg.↑**	**BPE↓**	**AIME25**	**GPQA-D**	**MATH500**	**Avg.↑**
Qwen3-8B									
Dense	16.00	40.65	25.99	33.32	16.00	66.67	56.06	90.23	70.99
INT2	2.00	17.07	0.20	8.64	2.00	0.00	3.03	0.00	1.01
Quest	1.90	37.40	10.05	23.73	2.99	43.33	38.89	74.80	52.34
OSCAR	2.22	31.71	10.52	21.12	3.93	53.55	42.42	83.00	59.66
BitLadder	**1.79**	**38.62**	**20.59**	**29.61**	**2.69**	**60.00**	**42.42**	**81.80**	**61.41**
DeepSeek-R1-Distill-Llama-8B									
Dense	16.00	28.80	13.83	21.32	16.00	26.67	32.32	78.80	45.93
INT2	2.00	10.00	0.00	5.00	2.00	0.00	4.55	1.00	1.85
Quest	2.13	26.00	7.21	16.61	3.49	13.33	16.16	60.80	30.10
OSCAR	2.22	25.20	3.98	14.59	4.87	16.67	28.28	74.40	39.78
BitLadder	**2.03**	**26.40**	**7.96**	**17.18**	**3.28**	**20.00**	**33.84**	**74.80**	**42.88**

inating shifts during merging. The two values sharing a 32-bit destination register are also placed at the spacing of its 16-bit lanes, allowing a single shift to align both. Compared with natural order, where consecutive values occupy consecutive bit positions, unpack-aware interleaving costs 18 rather than 64 integer instructions at $b = 4$ and 24 rather than 96 at $b = 8$ per 16 elements along the head dimension. We derive these instruction counts for both packing orders in Appendix C.1 and provide additional optimizations for key and value dequantization in Appendix C.3.

Fragment-aligned packing. Once the loaded pieces are merged, the values must occupy the per-thread operand fragments prescribed by the MMA instruction. The usual route stages unpacked 16-bit values through shared memory and loads the fragments with `ldmatrix`, adding a store–synchronize–load round trip to every decoding step. Since the instruction fixes this mapping in advance, we instead determine which values share a word and which thread owns it when each entry is appended. We derive the concrete KV address mappings from the MMA operand layout in Appendix C.2. Coalesced loads of the packed planes then provide each thread with the values that unpack into its KV operand fragments, avoiding their shared-memory rearrangement on the read path. This placement follows the MMA shape rather than the assigned width, so one packing serves every supported positive width.

4 Evaluation

4.1 Experimental Setup

Models and hardware. We evaluate BitLadder on two widely used long-context models, DeepSeek-R1-Distill-Llama-8B (DeepSeek-AI, 2025) and Qwen3-8B (Yang et al., 2025), which span two architecture families and natively support context windows of up to 128K tokens. All efficiency results are evaluated on NVIDIA L40 GPUs.

Benchmarks. We evaluate accuracy on five benchmarks: (1) long-context understanding on LongBench v2 (Bai et al., 2025) and MRCR (Vodrahalli et al., 2024), and (2) long-context reasoning on AIME25 (Zhang & Math-AI, 2025), GPQA-Diamond (Rein et al., 2024), and MATH500 (Lightman et al., 2024). For accuracy evaluation, we use greedy decoding with a maximum output length of 32K tokens (8K for the retrieval benchmark MRCR).

Baselines. We compare against one representative method per loading policy of Section 2.3. Quest (Tang et al., 2024) is query-aware sparse attention, which applies a keep-or-skip mask at block granularity. Uniform INT2 quantization assigns every entry the same width. OSCAR (Zhou et al., 2026) is importance-aware mixed-precision quantization, which maintains a sliding window and permanently quantizes tokens to 2 bits once they leave the window. We follow the released Quest and OSCAR configurations. For BitLadder, the block size is $S = 16$ and summary width is $b_s = 4$. We adjust BitLadder's tiered Top-P threshold p and Quest's token budget to reach target BPEs.

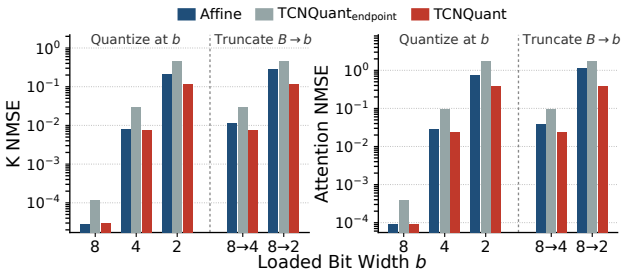


Figure 5: Reconstruction error (NMSE) of different quantizers across candidate bit widths. TCNQuant$_{\text{endpoint}}$ denotes TCNQuant without midpoint reconstruction.

Table 2: Effect of summary bits and block size. Extra BPE denotes summary bits read per KV element.

b_s	S	BPE↓	Acc.↑	Extra BPE↓
4	**16**	**2.09**	**38.62**	**0.13**
2	16	2.07	35.77	0.06
8	16	2.11	35.77	0.25
16	16	2.09	38.17	0.50
4	8	2.08	38.46	0.25
4	32	2.08	37.62	0.06
4	64	2.06	35.27	0.03

4.2 Accuracy Results

Long-Context Understanding. Table 1 shows that BitLadder outperforms all evaluated baselines on both understanding benchmarks across both models. At 1.79 and 2.03 BPE, BitLadder improves average accuracy over Quest by 5.88 and 0.57 percentage points on Qwen3-8B and DeepSeek-R1-Distill-Llama-8B, respectively. Note that uniform INT2 quantization severely degrades accuracy, whereas BitLadder comes closest to dense attention in average accuracy among the evaluated baselines, using even lower BPE than INT2 on Qwen3-8B. This advantage is consistent with importance-aware bit-width allocation, which preserves important tokens at higher precision.

Long-Context Reasoning. Experiments on long-context reasoning benchmarks also show that BitLadder achieves the highest average accuracy among the evaluated compressed methods on both models. As shown in Table 1, BitLadder improves average accuracy over OSCAR by 1.75–3.10 percentage points across the two models, while reducing average BPE by roughly one-third. These gains suggest that per-step bit-width adaptation helps preserve query-relevant context and thus mitigate error accumulation during long-context reasoning.

4.3 Ablation Studies

Quantization Analysis. Figure 5 shows that TCNQuant yields the lowest normalized mean squared error (NMSE) among the alternatives at low widths for both key reconstruction and attention output. Furthermore, its $8 \to b$ truncation also matches direct b-bit quantization. These results support the joint quantization and dequantization design in Section 3.2: TCNQuant combines low reconstruction error with truncation consistency, allowing multiple precisions to be served from one stored code.

Summary Bit Width b_s. Reducing b_s lowers summary traffic but can impair importance estimation. At comparable KV-payload BPE, $b_s = 4$ improves accuracy over $b_s = 2$, while higher precisions add traffic without further gains (Table 2). We therefore adopt $b_s = 4$.

Block Size S. Larger blocks reduce selection overhead but make bit-width allocation coarser, as more tokens share one width. In Table 2, $S = 16$ achieves the highest measured accuracy with lower summary traffic than $S = 8$, whereas larger blocks reduce traffic further at the expense of accuracy. We therefore choose $S = 16$ to maintain high accuracy with low selection overhead.

4.4 Efficiency Results

Attention Breakdown. Figure 6 decomposes attention latency into update, estimation, mixed-bit attention, and other costs. At 1.5 BPE, BitLadder achieves $2.2\times$, $3.6\times$, and $4.2\times$ speedups over dense attention at input lengths of 32K, 64K, and 96K tokens, respectively. Update and estimation costs remain relatively stable across the tested settings, while lower BPE reduces the mixed-bit attention cost. At longer contexts, attention savings increasingly outweigh update and estimation costs, yielding overall attention speedups.

Microbenchmarks. We conduct microbenchmarks on MRCR requests with at least 32K context length to quantify the accuracy-efficiency trade-offs of BitLadder. By sweeping BPE values, we characterize the trade-off between KV loading cost and accuracy. As demonstrated in Figure 7, bit-level selective loading, along with TCNQuant, enables BitLadder to preserve accuracy at about 2.3

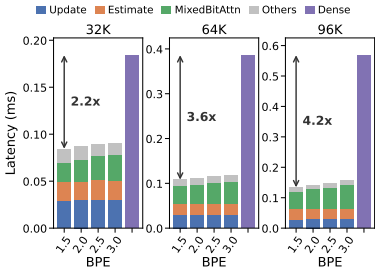


Figure 6: Attention latency breakdown.

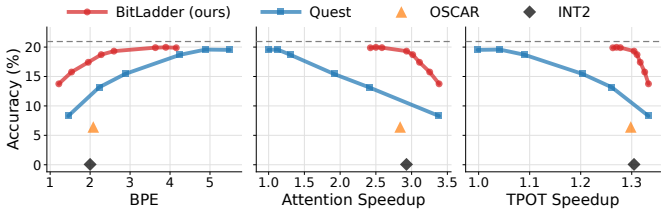


Figure 7: Accuracy–efficiency trade-offs. BitLadder preserves accuracy at about 2.3 BPE, with speedups of approximately $3.0\times$ for attention and $1.3\times$ for time per output token (TPOT).

BPE. PlaneWeave further translates this lower loading cost into practical speedups, yielding approximately $3.0\times$ acceleration for attention computation and a $1.3\times$ speedup in time per output token (TPOT). These results demonstrate that BitLadder effectively reduces KV traffic while efficiently managing the execution overhead of dynamic precision allocation.

5 Related Work

Sparse attention. KV-cache eviction methods reduce storage and traffic by discarding entries according to sink positions or accumulated attention scores, but discarded entries cannot serve later queries (Zhang et al., 2023b; Xiao et al., 2024b; Li et al., 2024). Query-aware sparse attention instead retains the cache and selects a query-dependent subset to load during decoding (Tang et al., 2024; Ribar et al., 2023; Xiao et al., 2024a; Liu et al., 2026b). Recent work extends sparse execution to prefill and other inference settings (Jiang et al., 2024; Liu et al., 2026a; Yuan et al., 2025; Lai et al., 2026; Bai et al., 2026). Despite these differences, selection remains a keep-or-skip decision at the chosen granularity. BitLadder maps query-dependent importance to graded loaded widths, including zero for skipping, retaining intermediate contributions at low traffic.

KV-cache quantization. KV-cache quantization reduces traffic by lowering the precision of cached entries. Uniform low-bit methods mitigate quantization error through tensor-specific grouping, sensitivity-aware non-uniform levels, joint weight-cache calibration, or offline rotation (Liu et al., 2024; Hooper et al., 2024; Yue et al., 2024; Zhou et al., 2026). Mixed-precision methods instead allocate different widths according to saliency, attention-derived quality targets, or an optimized bit budget (Yang et al., 2024; Dong et al., 2024; Boroujeni et al., 2026). Unlike these methods, BitLadder stores each KV entry once in a nested form and reselects its loaded width at every decoding step, letting one representation support query-dependent precision without additional quantized copies.

Nested quantization. The use of one representation at several bit widths connects our setting to nested quantization. Existing approaches, including any-precision networks, Matryoshka quantization, TruncQuant, and Any-Precision LLM, derive multiple deployment precisions from a shared representation of model weights (Yu et al., 2021; Nair et al., 2025; Kim et al., 2025; Park et al., 2024). These approaches construct the nested representation through training or post-training conversion of parameters available before inference. KV-cache entries, however, are generated online and therefore cannot directly undergo the same training or offline conversion process. BitLadder instead provides training-free nested quantization for online-generated KV-caches, enabling query-adaptive width loading from a single stored representation.

6 Conclusion

In this paper, we have presented BitLadder, a framework for bit-level selective loading that reduces KV-cache traffic by adapting loaded bit widths to query-dependent importance during long-context decoding. By combining TCNQuant with the PlaneWeave memory layout, BitLadder supports dynamic bit-width loading from a single stored representation and translates reduced memory traffic into efficient GPU execution. Experiments on long-context understanding and reasoning demonstrate a more favorable accuracy–efficiency trade-off than the evaluated baselines. Specifically, BitLadder achieves $3.0\times$ and $2.3\times$ speedups over full attention and the state-of-the-art baseline, respectively, while preserving comparable accuracy.

AI USE STATEMENT

In this work, we used generative AI tools to assist with mathematical proof derivation, implementation of the proposed method, and manuscript translation. We independently established the research framework and the paper's central argument, designed the method and experiments, and made research decisions. We did not use generative AI to generate or clean data or interpret experimental results. Additionally, we used these tools for language editing, drafting portions of the manuscript, refining section and paragraph organization within the defined framework, and literature searches. We reviewed all AI-assisted work: we checked the proof steps, assumptions, and conclusions; checked the implementation logic and correctness; reviewed the translated and drafted text for technical accuracy; and consulted the original publications to verify references and their support for the cited claims. We take responsibility for the final content of this work, including its text, mathematical claims, implementation, and reported results.

REPRODUCIBILITY STATEMENT

To support reproducibility, Section 3 describes BitLadder's quantization, query-aware bit-width allocation, and memory layout. Appendix B provides the quantizer's assumptions and proofs, while Appendices C and D detail layout derivations, system state, and component pseudocode. The models, baselines, default parameters, BPE calibration, decoding settings, and hardware used for efficiency evaluation are reported in Section 4.1. Section 4.4 specifies the context lengths and metrics used in the efficiency experiments. Our anonymized code and reproduction materials are available at `https://anonymous.4open.science/r/ICLR27-BitLadder-Artifact/`.

REFERENCES

GPT-5.6 Sol, 2026. URL `https://openai.com/index/previewing-gpt-5-6-sol/`.

Chenxin An, Shansan Gong, Ming Zhong, Xingjian Zhao, Mukai Li, Jun Zhang, Lingpeng Kong, and Xipeng Qiu. L-Eval: Instituting standardized evaluation for long context language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 14388–14411, 2024. doi: 10.18653/v1/2024.acl-long.776.

Anthropic. Introducing claude 4. `https://www.anthropic.com/news/claude-4`, May 2025.

Yushi Bai, Shangqing Tu, Jiajie Zhang, Hao Peng, Xiaozhi Wang, Xin Lv, Shulin Cao, Jiazheng Xu, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. LongBench v2: Towards deeper understanding and reasoning on realistic long-context multitasks. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2025.

Yushi Bai, Qian Dong, Ting Jiang, Xin Lv, Zhengxiao Du, Aohan Zeng, Jie Tang, and Juanzi Li. IndexCache: Accelerating sparse attention via cross-layer index reuse, 2026.

Sayed Pedram Haeri Boroujeni, Niloufar Mehrabi, Patrick Woods, Gabriel Hillesheim, and Abolfazl Razi. Don't waste bits! adaptive KV-cache quantization for lightweight on-device LLMs. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2026. URL `https://arxiv.org/abs/2604.04722`.

Ruisheng Cao, Mouxiang Chen, Jiawei Chen, Zeyu Cui, Yunlong Feng, Binyuan Hui, Yuheng Jing, Kaixin Li, Mingze Li, Junyang Lin, et al. Qwen3-coder-next technical report. *arXiv preprint arXiv:2603.00729*, 2026.

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, 2024. URL `https://arxiv.org/abs/2410.16179`.

Tri Dao, Daniel Y. 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 (NeurIPS)*, 2022.

DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL https://arxiv.org/abs/2501.12948.

Mohsen Dehghankar and Abolfazl Asudeh. Sparse attention as a range searching problem: Towards an inference-efficient index for kv cache, 2026. URL https://arxiv.org/abs/2605.06763.

Julien Delaunay, Hanh Thi Hong Tran, Carlos-Emiliano González-Gallardo, Georgeta Bordea, Nicolas Sidere, and Antoine Doucet. A comprehensive survey of document-level relation extraction (2016-2023), 2023. URL https://arxiv.org/abs/2309.16396.

Shichen Dong, Wen Cheng, Jiayu Qin, and Wei Wang. QAQ: Quality adaptive quantization for LLM KV cache, 2024. URL https://arxiv.org/abs/2403.04643.

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*, 2024.

Junhao Hu, Wenrui Huang, Weidong Wang, Zhenwen Li, Tiancheng Hu, Zhixia Liu, Xusheng Chen, Tao Xie, and Yizhou Shan. Raas: Reasoning-aware attention sparsity for efficient llm reasoning, 2025. URL https://arxiv.org/abs/2502.11147.

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. In *Advances in Neural Information Processing Systems*, 2024.

Jinhee Kim, Seoyeon Yoon, Taeho Lee, Joo Chan Lee, Kang Eun Jeon, and Jong Hwan Ko. TruncQuant: Truncation-ready quantization for DNNs with flexible weight bit precision, 2025.

Kimi Team et al. Kimi k3: Open frontier intelligence, 2026. URL https://arxiv.org/abs/2607.24653.

Xunhao Lai, Weiqi Xu, Yufeng Yang, Qiaorui Chen, Yang Xu, Lunbin Zeng, Xiaolong Li, Haohai Sun, Haichao Zhu, Vito Zhang, Jinkai Hu, Jiayao Li, Rui Gao, Zekun Li, Songquan Zhu, Jingkai Zhou, and Pengyu Zhao. MiniMax sparse attention, 2026.

Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. Infinigen: Efficient generative inference of large language models with dynamic kv cache management, 2024. URL https://arxiv.org/abs/2406.19707.

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, 2024.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *International Conference on Learning Representations (ICLR)*, 2024.

Chaofan Lin, Jiaming Tang, Shuo Yang, Hanshuo Wang, Tian Tang, Boyu Tian, Ion Stoica, Song Han, and Mingyu Gao. Twilight: Adaptive attention sparsity with hierarchical top-p pruning. *arXiv preprint arXiv:2502.02770*, 2025.

Anmin Liu, Ruixuan Yang, Huiqiang Jiang, Bin Lin, Minmin Sun, Yong Li, Chen Zhang, and Tao Xie. VecAttention: Vector-wise sparse attention for accelerating long context inference, 2026a.

Di Liu, Meng Chen, Baotong Lu, Huiqiang Jiang, Zhenhua Han, Qianxi Zhang, Qi Chen, Chengruidong Zhang, Bailu Ding, Kai Zhang, Chen Chen, Fan Yang, Yuqing Yang, and Lili Qiu. Retrievalattention: Accelerating long-context LLM inference via vector retrieval. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=8z3cOVER4z.

Di Liu, Ruitian Wang, Chen Chen, Mingliang Gong, Yongjie Yuan, Han Zhao, Yu Feng, Quan Chen, and Minyi Guo. AB-Sparse: Sparse attention with adaptive block size for accurate and efficient long-context inference, 2026b.

Tianyang Liu, Canwen Xu, and Julian McAuley. RepoBench: Benchmarking repository-level code auto-completion systems. *arXiv preprint arXiv:2306.03091*, 2023.

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, 2024.

Pranav Nair, Puranjay Datta, Jeff Dean, Prateek Jain, and Aditya Kusupati. Matryoshka quantization, 2025.

OpenAI. gpt-oss-120b & gpt-oss-20b model card, 2025. URL https://arxiv.org/abs/2508.10925.

Yeonhong Park, Jake Hyun, SangLyul Cho, Bonggeun Sim, and Jae W. Lee. Any-precision LLM: Low-cost deployment of multiple, different-sized LLMs. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 39682–39701, 2024.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. GPQA: A graduate-level google-proof Q&A benchmark. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=Ti67584b98.

Luka Ribar, Ivan Chelombiev, Luke Hudlass-Galley, Charlie Blake, Carlo Luschi, and Douglas Orr. SparQ attention: Bandwidth-efficient LLM inference, 2023.

Prajwal Singhania, Siddharth Singh, Shwai He, Soheil Feizi, and Abhinav Bhatele. Loki: Low-rank keys for efficient sparse attention, 2024. URL https://arxiv.org/abs/2406.02542.

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, 2025. URL https://arxiv.org/abs/2410.21465.

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*, 2024. URL https://arxiv.org/abs/2406.10774.

Kiran Vodrahalli, Santiago Ontanon, Nilesh Tripuraneni, et al. Michelangelo: Long context evaluations beyond haystacks via latent structure queries, 2024.

Chaojun Xiao, Pengle Zhang, Xu Han, Guangxuan Xiao, Yankai Lin, Zhengyan Zhang, Zhiyuan Liu, and Maosong Sun. InfLLM: Training-free long-context extrapolation for LLMs with an efficient context memory. In *Advances in Neural Information Processing Systems*, 2024a.

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

An Yang, Anfeng Li, Baosong Yang, et al. Qwen3 technical report, 2025.

June Yong Yang, Byeongwook Kim, Jeongin Bae, Beomseok Kwon, Gunho Park, Eunho Yang, Se Jung Kwon, and Dongsoo Lee. No token left behind: Reliable KV cache compression via importance-aware mixed precision quantization, 2024. URL https://arxiv.org/abs/2402.18096.

Haichao Yu, Haoxiang Li, Humphrey Shi, Thomas S. Huang, and Gang Hua. Any-precision deep neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pp. 10763–10771, 2021.

Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, Yuxing Wei, Lean Wang, Zhiping Xiao, et al. Native sparse attention: Hardware-aligned and natively trainable sparse attention. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 23078–23097, 2025.

Yuxuan Yue, Zhihang Yuan, Haojie Duanmu, Sifan Zhou, Jianlong Wu, and Liqiang Nie. WKVQuant: Quantizing weight and key/value cache for large language models gains more, 2024.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation, 2023a. URL https://arxiv.org/abs/2303.12570.

Tao Zhang, Ziqian Zeng, Hao Peng, Huiping Zhuang, and Cen Chen. Mixkvq: Query-aware mixed-precision kv cache quantization for long-context reasoning, 2025. URL https://arxiv.org/abs/2512.19206.

Yifan Zhang and Team Math-AI. American invitational mathematics examination (aime) 2025, 2025.

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*, 2023b.

Zhenyu Zhang, Shiwei Liu, Runjin Chen, Bhavya Kailkhura, Beidi Chen, and Atlas Wang. Q-hitter: A better token oracle for efficient llm inference via sparse-quantized kv cache. *Proceedings of Machine Learning and Systems*, 6:381–394, 2024.

Zhongzhu Zhou, Donglin Zhuang, Jisen Li, Ziyan Chen, Shuaiwen Leon Song, Ben Athiwaratkun, and Xiaoxia Wu. OSCAR: Offline spectral covariance-aware rotation for 2-bit KV cache quantization, 2026.

A BPE Derivations for Different Loading Policies

Let T_t denote the KV-payload bits loaded by a policy at step t, as in Section 2.3. With H_{kv} KV heads, head dimension d_h, and L_t cached tokens, the payload bits per element are

$$\mathrm{BPE}_t = \frac{T_t}{2H_{\mathrm{kv}} d_h L_t}, \tag{7}$$

where the factor of two accounts for keys and values, and H_{kv} is the quantity denoted by h_{kv} in Algorithm 1. A skipped entry contributes zero bits to T_t but remains in the denominator. All policies therefore count loaded payload bits per cached element, excluding quantization metadata and importance-estimation reads, which we account for separately below. This is a logical payload measure, not a measurement of physical memory transactions.

Full attention. Every cached element is read once at its native width, so Equation (7) gives $\mathrm{BPE}_t^{\mathrm{full}} = b_{\mathrm{native}}$, which is 16 for BF16. Full attention is the reference in Figure 2c and has zero distortion by definition.

Query-aware sparse attention. Let ρ_t be the fraction of cached KV elements retained at step t, each read at the native width. Then

$$\mathrm{BPE}_t^{\mathrm{sparse}} = \rho_t \, b_{\mathrm{native}}. \tag{8}$$

KV-cache quantization. Let $w_{t,b}$ be the fraction of cached KV elements stored at width b at step t, with $\sum_b w_{t,b} = 1$. Reading the entire quantized cache gives

$$\mathrm{BPE}_t^{\mathrm{quant}} = \sum_b w_{t,b} \, b. \tag{9}$$

Uniform quantization is the special case $w_{t,b_u} = 1$, giving $\mathrm{BPE}_t^{\mathrm{quant}} = b_u$. Static mixed-precision quantization fixes each entry's stored width on insertion. Other policies irreversibly lower stored

widths later, as OSCAR does when tokens leave its sliding window (Zhou et al., 2026). Thus, $w_{t,b}$ can change as entries are appended or existing entries are quantized to lower widths. These policies determine stored precision, unlike BitLadder's query-dependent selection of the loaded width from a single stored representation.

Bit-level selective loading. For the token-level notation of Section 2.3, suppose token i is read at width $b_{t,i}$ across KV heads. It contributes $2H_{\mathrm{kv}}d_h b_{t,i}$ payload bits, so

$$\mathrm{BPE}_t^{\mathrm{sel}} = \frac{1}{L_t}\sum_{i=1}^{L_t} b_{t,i}. \tag{10}$$

The method in Section 3.3 assigns one width $b_{t,h,j} \in \mathbb{B}$ to block $\mathcal{B}_j$ of KV head h, shared by its keys and values. Its corresponding payload average is

$$\mathrm{BPE}_t^{\mathrm{sel}} = \frac{\sum_{h=1}^{H_{\mathrm{kv}}}\sum_j |\mathcal{B}_j|\, b_{t,h,j}}{H_{\mathrm{kv}}L_t}. \tag{11}$$

The token count $|\mathcal{B}_j|$ accounts for an unfinished final block. An unweighted block average is equivalent only when all blocks have the same size. A zero width skips the block rather than reconstructing its entries at zero bits. Unlike static assignments, these widths can change with the query even at a fixed context length.

Side information. The payload BPE reported in Figure 2c and Section 4 does not include the following auxiliary reads. Lower payload BPE alone therefore does not determine total read traffic.

1. *Quantization metadata.* If a group of g elements uses two `fp32` parameters, its metadata costs $\mu(g) = 64/g$ bits per element. Each key or value group in Section 3.2 has $g = d_h$ elements, so reading every quantized group adds $\mu(d_h)$ to Equation (9). Let $\mathbf{1}\{\cdot\}$ denote the indicator function. When metadata is fetched only for retained blocks, bit-level selective loading instead adds
$$\mu(d_h)\,\frac{\sum_{h=1}^{H_{\mathrm{kv}}}\sum_j |\mathcal{B}_j|\,\mathbf{1}\{b_{t,h,j} > 0\}}{H_{\mathrm{kv}}L_t}. \tag{12}$$
Native-width full and sparse attention do not require this quantization metadata.
2. *Importance estimation.* Let an estimator read ν summary vectors of d_h elements per block and KV head, with each summary element stored at b_e bits. For $n_b = \lceil L_t/S \rceil$ blocks, the summary-payload reads add
$$\frac{\nu b_e n_b}{2L_t}, \qquad \text{or} \qquad \frac{\nu b_e}{2S} \quad \text{when } L_t = n_b S. \tag{13}$$
This term excludes the summaries' own metadata. BitLadder uses one pooled key summary per block and head, so $\nu = 1$ and $b_e = b_s$. At $b_s = 4$ and $S = 16$, full blocks contribute 0.125 BPE, reported as 0.13 Extra BPE in Table 2. These summary reads do not decrease when the payload allocation becomes more selective.

B Analysis of Truncation-Consistent Nested Quantization

Fix one quantization group: the key or value vector of one token and KV head. For bounds $x_{\min}$ and $x_{\max}$ with nonzero range $R = x_{\max} - x_{\min} > 0$, write

$$z = \frac{x - x_{\min}}{R} \in [0, 1] \tag{14}$$

for an element's normalized position. Then Equation (4) reads $Q_b^{\mathrm{TCN}}(x) = \min(\lfloor 2^b z \rfloor, 2^b - 1)$, and Equation (3) reads $Q_b^{\mathrm{aff}}(x) = \mathrm{round}((2^b - 1)z)$. The precisions B and b are integers with $0 < b \le B$. Truncation groups 2^{B-b} parent codes into one target code. As in Section 3.2, $b = 0$ denotes skipping and is outside these quantization and reconstruction rules. For a constant group ($R = 0$), zero codes and metadata $(x_{\min}, 0)$ reconstruct $x_{\min}$ exactly at every positive width without dividing by the range.

Lemma 1 (Nested floor)**.** *For every $y \in \mathbb{R}$ and every integer $m \ge 1$, $\lfloor \lfloor y \rfloor / m \rfloor = \lfloor y/m \rfloor$.*

Proof. Write $\lfloor y \rfloor = y - \theta$ with $\theta \in [0, 1)$, and divide with remainder, $\lfloor y \rfloor = m\,c + \rho$ with $c = \big\lfloor \lfloor y \rfloor / m \big\rfloor$ and $\rho \in \{0, \ldots, m-1\}$. Then $y/m = c + (\rho + \theta)/m$, and $0 \le (\rho + \theta)/m < m/m = 1$, so $\lfloor y/m \rfloor = c$. □

Truncation rather than rounding. Equation (2) leaves the reduction operator free. For $0 < b < B$, an alternative to truncation rounds to the nearest coarse code,

$$T^{\mathrm{round}}_{B\to b}(q) = \min\Big(\big\lfloor q/2^{B-b} + \tfrac{1}{2} \big\rfloor,\ 2^b - 1\Big) = \min\Big(\big(q + 2^{B-b-1}\big) \gg (B-b),\ 2^b - 1\Big), \quad (15)$$

The two operators differ in two respects, and both favor truncation. First, truncation is the shift $q \gg (B-b)$ and nothing else, whereas Equation (15) adds a bias before the shift and then clamps, the clamp being needed because the carry at $q = 2^B - 1$ would otherwise overflow the b-bit code. Second, rounding may require the most significant discarded bit, the guard bit at position $B-b-1$, in addition to the retained prefix. A fixed-width read therefore requires $b+1$ logical bits rather than b, an increment of $1/b$: 25% at $b = 4$ and 50% at $b = 2$. These percentages count logical bits, not measured memory transactions. Fetching an entire additional plane can cost more. At $b = B$, neither operator discards bits and no guard bit is needed.

B.1 Grid Misalignment of Affine Quantization

Rounding boundaries do not generally align. Write $N = 2^B - 1$, $n = 2^b - 1$, and $m = 2^{B-b}$, with $0 < b < B$. We use half-up rounding, $\mathrm{round}(y) = \lfloor y + \frac{1}{2} \rfloor$, to make the treatment of ties explicit. Truncation maps every m consecutive parent codes to one target code. Its transition from target code $k-1$ to k therefore occurs when the rounded parent code reaches mk, whereas direct quantization changes code when nz reaches $k - \frac{1}{2}$. The two boundaries are

$$z_k^{\mathrm{trunc}} = \frac{mk - \frac{1}{2}}{N}, \qquad z_k^{\mathrm{direct}} = \frac{k - \frac{1}{2}}{n}, \qquad k = 1, \ldots, n. \quad (16)$$

Since $N = mn + (m-1)$, their difference is $(m-1)(n+1-2k)/(2Nn)$. It vanishes only at the central boundary $k = (n+1)/2$, so the complete partitions differ for every $1 < b < B$, including the $8 \to 4$ and $8 \to 2$ reductions used here. For $b = 1$, the sole boundary is central and the codes agree, but this exception does not make the affine family consistent over the supported widths.

Misalignment affects a sizable share of the codes. The boundary mismatch can also be seen directly in the codes. In normalized coordinates, $Q_b^{\mathrm{aff}}(x) = \lfloor nz + \frac{1}{2} \rfloor$, and the parent code is $Q_B^{\mathrm{aff}}(x) = Nz + \varepsilon$ with $|\varepsilon| \le \frac{1}{2}$. Truncating that code divides it by m, and substituting $N = nm + (m-1)$ leaves

$$T_{B\to b}\big(Q_B^{\mathrm{aff}}(x)\big) = \Big\lfloor nz + \frac{(m-1)z}{m} + \frac{\varepsilon}{m} \Big\rfloor, \qquad Q_b^{\mathrm{aff}}(x) = \Big\lfloor nz + \frac{1}{2} \Big\rfloor, \quad (17)$$

where the position-dependent term $(m-1)z/m$ differs from the fixed rounding offset $1/2$. The two floor arguments differ by $\delta = (m-1)z/m + \varepsilon/m - \frac{1}{2}$, which lies in $[-\frac{1}{2} - \frac{1}{2m},\ \frac{1}{2} - \frac{1}{2m}]$. Thus the codes differ by at most one level and may do so in either direction. A one-level discrepancy changes the affine reconstruction by $s_b = R/(2^b - 1)$, which is a third of the group range at $b = 2$. For $B = 8$ and $b \in \{2, 4\}$, N/n is an odd integer, so every direct-quantization boundary is also a parent rounding boundary. The direct target code is therefore constant within each parent bin. Enumerating the 256 parent codes gives 56 disagreements at $b = 4$ and 42 at $b = 2$, split evenly between the two directions. These are code counts, not empirical disagreement probabilities under a data distribution. In contrast, TCNQuant agrees at every positive target width, as shown next.

B.2 Truncation Consistency of TCNQuant Codes

Proposition 1 (TCNQuant is truncation-consistent)**.** *For every integer $0 < b \le B$ and every x in the group, $Q_b^{\mathrm{TCN}}(x) = T_{B\to b}\big(Q_B^{\mathrm{TCN}}(x)\big)$, so the family $\{Q_b^{\mathrm{TCN}}\}$ satisfies Definition 1.*

Proof. For $R = 0$, all codes are zero at every positive width, so the equality holds. For $R > 0$, we distinguish the interior of the group range from its upper endpoint, since Equation (4) saturates only at the latter. If $z < 1$, then $\lfloor 2^B z \rfloor \le 2^B - 1$ and likewise $\lfloor 2^b z \rfloor \le 2^b - 1$, so the saturation

is inactive at both precisions and $Q_B^{\mathrm{TCN}}(x) = \lfloor 2^B z \rfloor$, $Q_b^{\mathrm{TCN}}(x) = \lfloor 2^b z \rfloor$. Applying Lemma 1 with $y = 2^B z$ and $m = 2^{B-b}$ gives $T_{B\to b}(\lfloor 2^B z \rfloor) = \lfloor \lfloor 2^B z \rfloor / 2^{B-b} \rfloor = \lfloor 2^b z \rfloor$, as claimed. If $z = 1$, the saturation is active at both precisions, so $Q_B^{\mathrm{TCN}}(x) = 2^B - 1$ and $Q_b^{\mathrm{TCN}}(x) = 2^b - 1$. Division with remainder gives $2^B - 1 = 2^{B-b}\big(2^b - 1\big) + \big(2^{B-b} - 1\big)$ with $0 \le 2^{B-b} - 1 < 2^{B-b}$, hence $T_{B\to b}(2^B - 1) = \lfloor (2^B - 1)/2^{B-b} \rfloor = 2^b - 1$. □

B.3 Mean Preservation of TCNQuant Reconstructions

Proposition 1 matches the codes produced at the two widths, while the quantity that reaches attention is the reconstructed value. Truncating to the b-bit code k merges the 2^{B-b} parent codes $\{2^{B-b}k + j\}_{j=0}^{2^{B-b}-1}$, so one b-bit reconstruction has to stand for all of them, and Equation (5) places it at their average.

Proposition 2 (Mean-preserving coarsening)**.** *For every $0 < b < B$ and every b-bit code k,*

$$D_b^{\mathrm{TCN}}(k) = \frac{1}{2^{B-b}} \sum_{j=0}^{2^{B-b}-1} D_B^{\mathrm{TCN}}\big(2^{B-b}k + j\big). \tag{18}$$

Proof. The 2^{B-b} parent codes are consecutive, so their mean is $2^{B-b}k + (2^{B-b} - 1)/2$, and D_B^{TCN} is affine in the code, so averaging the reconstructions is the same as reconstructing this mean. The half-step of Equation (5) completes the offset, $\big(2^{B-b}k + \frac{2^{B-b}-1}{2} + \frac{1}{2}\big)\Delta_B = 2^{B-b}\big(k + \frac{1}{2}\big)\Delta_B$, and $\Delta_b = 2^{B-b}\Delta_B$ turns this into $x_{\min} + \big(k + \frac{1}{2}\big)\Delta_b = D_b^{\mathrm{TCN}}(k)$: the midpoint of a coarse bin is the mean of the midpoints of the parent bins it merges. □

C Analysis of PlaneWeave

The three analyses in this section substantiate the layout claims of Section 3.4. First, we derive the unpacking instruction count at each supported positive width. Second, we derive the K and V address permutations for fragment-aligned packing and show how they remove read-path rearrangement. Third, we establish the algebraic folding of midpoint reconstruction around the key and value MMAs.

C.1 Unpacking Instruction Counts

Counting convention. Every count below is normalized to 16 unpacked elements: their 2-bit pieces occupy one 32-bit payload word. This is the same per-element accounting used in Section 3.4. After fragment-aligned packing, the elements in a physical word can span multiple tokens, as the mappings in Appendix C.2 show. We count two classes of integer instructions: shifts and the NVIDIA `LOP3` instruction, which applies a lookup-table-specified Boolean function bitwise to three 32-bit source registers. For the first masked insertion into a destination word, these inputs can be the shifted source, a constant mask, and the fixed mantissa-bit pattern, so the same `LOP3` both inserts code bits and sets the fixed bits. Later masked insertions use the accumulated word as the third input and preserve these fixed bits while updating the remaining code fields. We therefore count each masked insertion as one instruction and charge no separate instruction for inserting the fixed mantissa bits. Integer-to-float conversion and affine reconstruction are excluded because they are independent of the packing order. Appendix C.3 shows how reconstruction leaves the per-element path. We write $m(b)$ for the number of planes that width b reads, so $m(2) = 1$, $m(4) = 2$, and $m(8) = 3$.

Baseline: natural packing. One 32-bit destination register contains two 16-bit lanes, so the 16 elements form eight destination pairs. Under the natural order, the two elements of each pair occupy adjacent source positions and must be placed into their destination lanes independently for every plane. Each element requires one shift and one `LOP3` masked insertion, giving four instructions per pair and plane. The cost is therefore

$$C_{\mathrm{base}}(b) = \underbrace{8}_{\text{pairs}} \times \underbrace{(2 \times 2)}_{\substack{\text{two elements,}\\ \text{shift + insert}}} m(b) = 32,\ 64,\ 96 \qquad \text{at } b = 2,\ 4,\ 8. \tag{19}$$

The count grows with the number of planes because the two elements must be aligned independently in every plane.

Ours: unpack-aware interleaving. The two rules of Section 3.4 each remove one part of this count. Because the pieces of a value are already at the relative offset they occupy inside the reassembled code, merging them requires no shift. The first plane initializes each output word, and each of the remaining $m(b) - 1$ planes is combined by one `LOP3` under a constant mask. At $b = 4$, the two masks are complementary, allowing this `LOP3` to select between the two planes. Since the 16 reassembled codes occupy $16b$ bits, the merge produces $16b/32 = b/2$ output words. Because the two elements of a pair are stored at the spacing the target register expects, one shift aligns both. One `LOP3` then inserts the aligned codes under a constant mask. Thus each of the 8 pairs costs one shift and one `LOP3` irrespective of b. Hence

$$C_{\text{ours}}(b) = \underbrace{\frac{b}{2}}_{\text{output words}} \underbrace{\big(m(b)-1\big)}_{\text{merges per word}} + \underbrace{8}_{\text{pairs}} \times \left(\underbrace{1}_{\text{shared shift}} + \underbrace{1}_{\texttt{LOP3}} \right) = 16,\ 18,\ 24 \qquad \text{at } b = 2,\ 4,\ 8, \tag{20}$$

This is a reduction of 50%, 72%, and 75% against Equation (19).

A shared unpacking structure across widths. For a fixed active tile, the lane assignment, warp partitioning, MMA shape, and eight destination pairs are unchanged across positive widths. This gives the constant 16 in Equation (20). The assigned width selects the plane prefix, merge count, shift constants, and factor 2^{B-b} on Δ_B. Thus different widths reuse the same fragment mapping and unpacking structure, without requiring a different stored layout for each width.

C.2 Fragment-Aligned Address Mapping

Fragment-aligned packing aims to make each loaded cache word belong to the warp lane that will consume it, thereby removing the shared-memory rearrangement before the MMA. Let T denote the number of cached tokens in a physical packing tile. For a cache operand $X \in \{K, V\}$, let $c_m^X[t, d]$ be the plane-m piece at in-tile token index $t \in \{0, \dots, T-1\}$ and head-dimension index d. The operand layout of the MMA that consumes X specifies a bijection

$$\varpi_X : (t, d) \mapsto (w, \ell, r, u), \tag{21}$$

where w is the warp, ℓ the warp lane, r the register within its fragment, and u the 16-bit half-word within that register. Packing stores $c_m^X[t, d]$ in the plane-m word owned by (w, ℓ) and in the bit slot associated with (r, u). The pair (w, ℓ) selects the destination word through an operand-specific tensor permutation, while (r, u) selects the bit slot required by the interleaving of Appendix C.1. On the read path, loading and reassembling that word therefore places the value directly at the fragment position prescribed by the MMA, without a shared-memory rearrangement before the MMA.

The K mapping for $QK^\top$. In the implemented specialization shown in Figure 4, the allocation block and physical tile coincide at $S = T = 16$, the head dimension is $d_h = 128$, and K is the B operand of `mma.m16n8k16`. Token is its output dimension n, while head dimension is its reduction dimension k. Writing $t = (t_3 t_2 t_1 t_0)_2$ and $d = (d_6 \dots d_0)_2$, its instance of Equation (21) is

$$w = (d_6 d_5)_2, \qquad \ell = (t_2 t_1 t_0 d_2 d_1)_2, \qquad r = (t_3 d_4 d_3)_2, \qquad u = d_0. \tag{22}$$

For example, lane 0 of warp 0 receives two tokens, $t = 0, 8$, across the eight head-dimension coordinates $d = 0, 1, 8, 9, 16, 17, 24, 25$. These 16 pieces occupy exactly one 32-bit word in each 2-bit plane, while their positions within the word encode the required register and half-word. Thus the two sides of the fragment-aligned panel in Figure 4 show the same K elements before and after applying ϖ_K.

The V mapping for PV. For the value product PV, where P denotes the attention probabilities, V is also a B operand of `mma.m16n8k16`. In the implemented specialization, each warp consumes a 16×32 V tile comprising all 16 token coordinates and one 32-coordinate slice of the head dimension. Within this tile, head dimension is the output dimension n, while token is the reduction dimension k. Applying the B-fragment rule with these roles gives

$$w = (d_6 d_5)_2, \qquad \ell = (d_2 d_1 d_0 t_2 t_1)_2, \qquad r = (d_4 d_3 t_3)_2, \qquad u = t_0. \tag{23}$$

Accordingly, lane 0 of warp 0 receives four tokens, $t = 0, 1, 8, 9$, across four head-dimension coordinates, $d = 0, 8, 16, 24$. These 16 pieces again occupy one 32-bit word per 2-bit plane. The change from $u = d_0$ in Equation (22) to $u = t_0$ in Equation (23) reflects the exchanged reduction dimension: a K register pairs adjacent head-dimension coordinates of one token, whereas a V register pairs adjacent tokens at one head-dimension coordinate.

Scope of the specialization. The explicit maps above describe the default specialization with $S = T = 16$ and $d_h = 128$. For $d_h = 32G$, the construction extends over G head-dimension groups: $QK^\top$ sums their partial scores, whereas PV assigns them disjoint output-coordinate ranges. The maps shown here use $G = 4$. An allocation block of size $S = qT$ can likewise consist of q independently mapped physical tiles sharing one assigned width. Other sizes require a compatible packing or tail treatment. They are not specified by these maps. The block-size ablation in Table 2 varies the allocation granularity and does not imply that this single mapping covers every tested size. Within the default specialization, changing the assigned width changes the plane prefix but not either address map.

C.3 Folding Dequantization around the MMA

The midpoint reconstruction of Equation (5) is affine in the integer code, so its scale and offset can move outside the per-element path. For K, the parameters of one token and KV head are constant along the head dimension, which is the reduction dimension of $QK^\top$. Writing c_d for the code at coordinate d and Δ for its assigned-width bin,

$$\mathbf{q}^\top \hat{\mathbf{k}} = \sum_d q_d \Big[x_{\min} + \big(c_d + \tfrac{1}{2}\big)\Delta \Big] = \Delta \sum_d q_d c_d + \Big(x_{\min} + \frac{\Delta}{2} \Big) \sum_d q_d. \tag{24}$$

The MMA can therefore accumulate the codes, after which one scale and one offset correction recover each score. The sum $\sum_d q_d$ is shared by all keys of the query head.

For V, token-wise quantization makes $x_{\min,t}$ and Δ_t shared across all head-dimension coordinates of token t, although they vary along the token reduction dimension of PV. Writing p_t for the attention probability of token t and $c_{t,d}$ for its integer code at output coordinate d gives

$$o_d = \sum_t p_t \Big[x_{\min,t} + \big(c_{t,d} + \tfrac{1}{2}\big)\Delta_t \Big] = \sum_t \big(p_t \Delta_t\big) c_{t,d} + \sum_t p_t \Big(x_{\min,t} + \frac{\Delta_t}{2} \Big). \tag{25}$$

Thus $p_t \Delta_t$ is supplied as the probability-side MMA input without renormalizing P, and the second sum is one scalar added to every output coordinate. The mantissa-based conversion supplies $1024 + c$ rather than c. Substituting it into Equations (24) and (25) adds another term of the same scalar-correction form, so the offset need not be removed per element.

D System Implementation

Algorithm 2 expands the five functions called by Algorithm 1 for the default $S = 16$, $B = 8$, and $b_s = 4$ configuration. Token positions are zero-based in the indexing formulas below, and L denotes the current cache length. We write $\mathrm{meta}(X)$ for the auxiliary quantization metadata associated with a code tensor X. This notation makes explicit the metadata passed alongside codes in Algorithm 1. The metadata is stored separately from the integer code fields.

Cache state. The cache is $\mathcal{C} = (\boldsymbol{P}_1, \boldsymbol{P}_2, \boldsymbol{P}_3, \boldsymbol{M})$. Its three packed planes have logical dimensions $2 \times h_{kv} \times L \times d_h$ and hold the $2/2/4$-bit pieces of each parent code at precision $B = 8$. The leading dimension indexes keys and values, respectively. The metadata tensor $\boldsymbol{M} \in \mathbb{R}^{2 \times h_{kv} \times L \times 2}$ stores one $(x_{\min}, \Delta_B)$ pair per token and KV head for each operand. The L tokens form $n_b = \lceil L/S \rceil$ contiguous blocks $\mathcal{B}_j$ of at most $S = 16$ tokens. The last block may be incomplete. A positive width b requires $m(2) = 1$, $m(4) = 2$, or $m(8) = 3$ leading planes, while $b = 0$ skips the block entirely.

Quantization and appending. To populate this cache, TCNQuant forms one group per token and KV head, separately for keys and values, and partitions each group's range into 2^B equal-width bins. The stacked input $\boldsymbol{x} \in \mathbb{R}^{2 \times h_{kv} \times d_h}$ produces codes $\boldsymbol{c} \in \mathbb{Z}_{256}^{2 \times h_{kv} \times d_h}$, whose metadata accompanies them to Append. This function splits each code by significance into $2 + 2 + 4$ bits and places

the pieces using the key and value maps ϖ_K, ϖ_V of Appendix C.2. We write $\boldsymbol{P}_m[0]_{\varpi_K,t}$ and $\boldsymbol{P}_m[1]_{\varpi_V,t}$ for the key and value bit slots owned by token t. The maps act on $t \bmod S$ within block $\lfloor t/S \rfloor$. A packed word can contain pieces of several tokens, so assignment to either view updates the selected slots and preserves all others. The same maps serve every supported loaded width b. The accompanying metadata is stored separately at token index t.

Summary maintenance. The appended key also updates its block summary through UPDATESUMMARY. For the current block $j = \lfloor t/S \rfloor$, the `fp32` accumulator $\boldsymbol{a} \in \mathbb{R}^{h_{kv} \times d_h}$ stores the running mean of its raw keys. The recurrence incorporates $\boldsymbol{k}_t$ at local position $u = t - jS$, with $\boldsymbol{a} = \boldsymbol{k}_t$ when $u = 0$. Block summaries are quantized separately with TCNQuant at $b_s = 4$ bits. After each mean update, $Q_{b_s}^{\mathrm{TCN}}(\boldsymbol{a})$ replaces $\bar{\boldsymbol{K}}[:, j]$, and its per-head bounds and scale replace the corresponding entries of $\mathrm{meta}(\bar{\boldsymbol{K}})$. Both become final when the block is complete. The running mean never uses a quantized summary as input. The scoring shorthand in Algorithm 1 uses reconstructed summaries $D_{b_s}^{\mathrm{TCN}}(\bar{\boldsymbol{K}})$ with this metadata, as specified by Equation (6), rather than multiplying queries by integer codes directly.

Bit-width allocation. These reconstructed summaries supply the block scores of Section 3.3. Softmax normalization and averaging over query heads sharing a KV head produce $\boldsymbol{\pi}$, a proxy distribution over blocks rather than their exact attention mass. For each KV head, TIEREDTOPP sorts the distribution into order σ and computes prefix masses c_k. For $0 < p \leq 1$, the cumulative targets $\tau_m = 1 - (1 - r)^m$, $m = 1, 2, 3$, follow Section 3.3, with $\tau_3 = p$. We make the conventional Top-P boundary rule explicit: let $k_m = \min\{k : c_k \geq \tau_m\}$, so the first block reaching each target is retained. The nested prefixes end at $k_1 \leq k_2 \leq k_3$ and define the $8/4/2$-bit segments, while blocks after k_3 receive zero bits. Segments may be empty if their endpoints coincide. Because blocks are indivisible, the realized retained proxy mass can exceed p. These thresholds neither fix KV-payload BPE nor prescribe exact oracle attention mass.

Mixed-bit attention. Let $\mathcal{G}(h)$ denote the query heads sharing KV head h. Both TIEREDTOPP and MIXEDBITATTN in Algorithm 2 are written for one KV head: b_j denotes its block width, $\boldsymbol{q}_t$ its query-group slice, and $\mathcal{C}$ its cache slice. Applying them independently to all KV heads gives the tensors in Algorithm 1. For an active block, the leading $m(b_j)$ planes reassemble into codes of shape $2 \times |\mathcal{B}_j| \times d_h$ using Appendix C.1. The matching metadata supplies $(x_{\min}, \Delta_B)$ for midpoint reconstruction at $\Delta_{b_j} = 2^{B-b_j}\Delta_B$, with affine terms folded around the MMA as in Appendix C.3. The pseudocode expresses this reconstruction explicitly without requiring a materialized dequantized cache. For the valid tokens of each active block, $\mathrm{OnlineAttn}$ follows FlashAttention (Dao et al., 2022), maintaining an unnormalized output $\boldsymbol{o} \in \mathbb{R}^{|\mathcal{G}(h)| \times d_h}$, denominator ℓ, and running maximum η per query head. It includes the $1/\sqrt{d_h}$ score scaling and excludes unfilled tail positions. Dividing $\boldsymbol{o}$ by ℓ gives each group's output. These outputs are assembled in query-head order.

Algorithm 2 Component functions of Algorithm 1

function TCNQuant$(\boldsymbol{k}_t, \boldsymbol{v}_t)$
$\boldsymbol{x} \leftarrow \mathrm{Stack}(\boldsymbol{k}_t, \boldsymbol{v}_t)$
$x_{\min} \leftarrow \min_{d_h} \boldsymbol{x}, \quad x_{\max} \leftarrow \max_{d_h} \boldsymbol{x}, \quad \Delta_B \leftarrow (x_{\max} - x_{\min})/2^B$
$\boldsymbol{c} \leftarrow Q_B^{\mathrm{TCN}}(\boldsymbol{x}), \quad \mathrm{meta}(\boldsymbol{c}) \leftarrow (x_{\min}, \Delta_B)$
return $\boldsymbol{c}$
end function
function Append$(\mathcal{C}, \boldsymbol{c})$
$\boldsymbol{c}_1 \leftarrow \boldsymbol{c} \gg 6, \quad \boldsymbol{c}_2 \leftarrow (\boldsymbol{c} \gg 4)\ \&\ 3, \quad \boldsymbol{c}_3 \leftarrow \boldsymbol{c}\ \&\ 15$
for $m = 1$ **to** 3 **do**
    $\boldsymbol{P}_m[0]_{\varpi_K, t} \leftarrow \boldsymbol{c}_m[0], \quad \boldsymbol{P}_m[1]_{\varpi_V, t} \leftarrow \boldsymbol{c}_m[1]$
end for
$\boldsymbol{M}[:, :, t] \leftarrow \mathrm{meta}(\boldsymbol{c})$
return $\mathcal{C}$
end function
function UpdateSummary$(\bar{\boldsymbol{K}}, \boldsymbol{a}, \boldsymbol{k}_t)$
$j \leftarrow \lfloor t/S \rfloor, \quad u \leftarrow t - jS$
$\boldsymbol{a} \leftarrow \boldsymbol{k}_t$ if $u = 0$, otherwise $\big(u\,\boldsymbol{a} + \boldsymbol{k}_t\big)/(u+1)$
$x_{\min} \leftarrow \min_{d_h} \boldsymbol{a}, \quad \Delta_{b_s} \leftarrow (\max_{d_h} \boldsymbol{a} - x_{\min})/2^{b_s}$
$\bar{\boldsymbol{K}}[:, j] \leftarrow Q_{b_s}^{\mathrm{TCN}}(\boldsymbol{a}), \quad \mathrm{meta}(\bar{\boldsymbol{K}})[:, j] \leftarrow (x_{\min}, \Delta_{b_s})$
return $\bar{\boldsymbol{K}}, \boldsymbol{a}$
end function
function TieredTopP$(\boldsymbol{\pi}, p)$
$r \leftarrow 1 - (1-p)^{1/3}, \quad \tau_m \leftarrow 1 - (1-r)^m$ for $m = 1, 2, 3$
$\sigma \leftarrow \mathrm{ArgSort}_{\downarrow}(\boldsymbol{\pi}), \quad c_k \leftarrow \sum_{l \le k} \pi_{\sigma(l)}$
$k_m \leftarrow \min\{k : c_k \ge \tau_m\}$ for $m = 1, 2, 3$
$b_{\sigma(k)} \leftarrow \begin{cases} 8, & 1 \le k \le k_1, \\ 4, & k_1 < k \le k_2, \\ 2, & k_2 < k \le k_3, \\ 0, & k_3 < k \le n_b. \end{cases}$
return $\boldsymbol{b}$
end function
function MixedBitAttn$(\boldsymbol{q}_t, \boldsymbol{b}, \mathcal{C})$
$\boldsymbol{o} \leftarrow \boldsymbol{0}, \quad \ell \leftarrow 0, \quad \eta \leftarrow -\infty$
for $j : b_j > 0$ **do**
    $\boldsymbol{c} \leftarrow \mathrm{Concat}_{m \le m(b_j)} \boldsymbol{P}_m[:, \mathcal{B}_j]$
    $(x_{\min}, \Delta_B) \leftarrow \boldsymbol{M}[:, \mathcal{B}_j]$
    $\hat{\boldsymbol{K}}, \hat{\boldsymbol{V}} \leftarrow x_{\min} + \big(\boldsymbol{c} + \frac{1}{2}\big) 2^{B - b_j} \Delta_B$
    $\boldsymbol{o}, \ell, \eta \leftarrow \mathrm{OnlineAttn}(\boldsymbol{q}_t, \hat{\boldsymbol{K}}, \hat{\boldsymbol{V}}, \boldsymbol{o}, \ell, \eta)$
end for
return $\boldsymbol{o}/\ell$
end function