Subquadratic Attention with a Hierarchical Tree-Structured Indexer

Leon Lufkin*, Cutter Dawes*, Alexander Speicher, Kamesh Krishnamurthy†, Tomás Figliolia†, Beren Millidge†

[1] Zyphra Research * Equal contribution † Senior Author

Correspondence: {leon,tom}@zyphra.com

arXiv:2610.42112v1 [cs.CL] 30 Oct 2026

***Abstract*—The quadratic scaling of attention is a fundamental bottleneck limiting the long-context capabilities of large language models (LLMs). A variety of approaches have attempted to circumvent this, but they either fail to match attention's expressiveness (e.g., sliding-window or linear attention variants with fixed-size state) or only achieve constant-factor speedups (e.g., sparse attention methods whose indexers themselves scale quadratically). In this paper, we derive an asymptotically subquadratic sparse attention method based on a hierarchical, tree-structured indexer, which we call Zyphra Hierarchical Attention (ZHA). ZHA constructs a tree of recursively compressed key summaries over the KV cache, which future queries traverse from the top down to retrieve a sparse subset of keys. Crucially, by iteratively growing the tree's depth, the total cost including routing and final attention grows sublinearly in sequence length, resulting in overall subquadratic (but superlinear) scaling for attention. We demonstrate that ZHA can be cheaply applied to existing pretrained models with only a short KL distillation training stage over a few billion tokens, and maintains performance on long-context retrieval and challenging reasoning, mathematical, and agentic coding evaluation tasks. Moreover, we show that our method is highly robust to tree geometry including very deep trees of up to 6 levels. Finally, we develop a detailed performance model and show that our approach scales subquadratically, dramatically outperforms competing methods beyond 1 million context, and ultimately has the potential to extend effective context windows to the hundreds of millions or billions of tokens.**

I. Introduction

Attention is the fundamental sequence-mixing primitive in modern transformer architectures [41], and it is the basis for the highly flexible and expressive sequence modeling that underpins the recent rapid and dramatic advances in AI. Attention expresses an all-to-all mapping between sequence elements, where each token in the sequence is compared to every other token to identify those which are relevant. It is highly expressive and unstructured, yet parallelizable and efficiently implementable on GPUs. These properties have made it the predominant sequence-mixing architecture and enabled massive scaling of model size and sequence lengths.

However, the all-to-all nature of attention leads to fundamentally unfavorable scaling in computational and memory costs. In prefill, its quadratic computational cost dominates other operations, while during decode, re-reading the whole key-value (KV) cache per new token imposes memory bandwidth and storage costs which linearly increase in the sequence length. This has prohibited extremely long LLM contexts, despite continuing gains from longer reasoning and input lengths across a wide range of increasingly challenging tasks [2, 9].

This has motivated a variety of approaches that reduce the computational cost of attention [38]. State-space models and sliding-window attention [6, 12, 15, 24] replace the full KV cache with a fixed-size state that is updated along the sequence. While these eliminate the asymptotic cost of attention with linear prefill and constant decode cost, their fixed-size state saturates at long contexts, leading to worse performance [21]. Practical "hybrid" methods [13, 14, 26, 39] thus often interleave a fixed-state mixer with full attention, reaping the computational benefits in most layers while restricting the attention costs to only a few, necessary layers. However, at sufficiently long contexts, the few layers using full attention still dominate.

Other approaches aim to compress the sequence directly by online-compressing multiple tokens into a single token during prefill or decode (e.g., CSA [8], NSA [46], HNet [20]), or by compressing the KV cache by reducing the number of independent heads [1, 11, 28]. A related set of methods focuses on offline KV cache eviction, where tokens deemed to be of little value are evicted periodically from the KV cache, reducing the cost of all future queries [23]. However, although head-compression methods such as GQA and MLA have become ubiquitous, they are fundamentally limited in possible compression rate by the number of heads. Meanwhile, sequence-compression and KV-eviction methods theoretically enable large compression rates but have fared less well empirically. Many of these methods both only obtain a constant factor speedup, and also are usually brittle and scale poorly in practice. A core reason for this brittleness is that it is practically impossible to know ahead of time, or on average, whether a token is useful or not.

This leads directly to a family of *fine-grained sparse attention* methods that enable each query to retrieve its own sparse subset of keys and values from the context (e.g., DSA [29], MSA [25]). In these methods, a lightweight indexer attention scores every KV pair's relevance to each query, and full attention is performed only on the top-K most relevant pairs. Since K is fixed, the full attention has a constant cost. Crucially, this approach still permits each query to select an independent subset of keys most important to it, which enables higher precision for retrieval than compressing the same set of keys for all queries. However, this flexibility comes with two costs. First, the full KV cache must be stored, even if not computed

over every token. Second, and most importantly, it relocates the cost of attention into the indexer. In practice, one can obtain significant speedups by making the indexer cheap with fewer heads or lower precision, but at contexts near 1M tokens, 60-90% of time in these layers is still spent on indexing [3, 47]. A subfamily of *chunk-sparse* attention methods [18, 19, 46] attempts to address this by compressing the indexer itself, and sparsely selecting chunks from the KV cache over which to perform attention. Of interest is Huang et al. [19] which uses an adaptive sparsity function α-entmax, similar to our top-p routing in spirit, and Hu et al. [18], which constructs chunk summaries as an attention-weighted sum, similar to our own key construction method. However, these methods tend to be empirically brittle and only perform chunking over a single level, thus also only obtaining constant factor speedup.

This leads to the conclusion that to construct an effective attention replacement yielding better than constant-factor speedups, two properties must be satisfied. The first property, derived from sparse attention, is *query-dependent sparsity*, since different queries can require substantially different information from the KV cache. The second property is a *growing but sublinear active state size*. New sequence elements add predictive information indefinitely, so a fixed-size state will inevitably reach capacity and degrade in performance. By contrast, full attention can be viewed as a non-parametric state that grows linearly with sequence length. This linear growth is maximally expressive but highly redundant [4, 10, 30, 48], since the predictive information of each new token diminishes: the millionth token contains information, but likely less than the first. The challenge is to design a state that can grow dynamically, but sublinearly, at the rate that predictive information accumulates in the sequence.

Motivated by these considerations, we introduce a novel hierarchical sparse attention method, which we call Zyphra Hierarchical Attention (ZHA), that can achieve better than constant-factor speedups over dense attention. By analyzing the structure of attention's representations, we find that the size of its effective state is sublinear and that it is amenable to a hierarchical indexer. Moreover, we show that we can adapt pretrained models to use our hierarchical indexer with distillation training of only a few billion tokens and very few additional parameters, without changing any of the base model's weights. We convert models up to 35B parameters and observe virtually no degradation on challenging evaluations covering mathematical reasoning, agentic coding, and long-context retrieval. We also perform a detailed study of various indexer geometries and demonstrate that ZHA is highly robust to such variations and, crucially, can maintain performance at large depths.

Finally, we develop a detailed theoretical performance model of our method as well as comprehensive empirical sparsity results for a variety of indexer geometries. Our results demonstrate that ZHA can remain asymptotically subquadratic, subject to careful top-p scaling, when extrapolated to extremely long sequences of tens of millions to billions of tokens, and can thus theoretically strongly outperform sparse attention methods at these context lengths in our performance model. We model ZHA's speedups against dense attention and MSA with a roofline model [45], which predicts a crossover point against the latter around 0.5M tokens for a performant geometry with $p = 0.75$.

II. Motivation

A. The Natural Sparsity of Attention

Our work starts from the observation that attention's representations on language data already admit a sublinear state. Importantly, sublinear compressibility is not fundamental to the attention architecture but rather determined by the data and task distribution. Adversarial sequences consisting of random tokens from which arbitrary tokens must be retrieved uniformly throughout the sequence would require full quadratic attention to process. However, we observe that natural language sequences appear, on average, to follow a sublinear growth of predictive information with sequence length.

We operationalize this notion by studying the size of the top-p KV subset of global attention layers in a number of language models on a dataset of long-context fiction books. The top-p set is the smallest subset of keys whose attention probabilities sum to at least p. Top-p can be thought of as a denoiser for identifying attention's "true" receptive field. We can write the output of attention as

$$\begin{aligned}\text{Attention}(q, K, V) = \quad & p\ \text{Attention}(q, K_p, V_p) \\ & + (1-p)\ \text{Attention}(q, K_{\bar{p}}, V_{\bar{p}}).\end{aligned} \tag{1}$$

where (K_p, V_p) denotes the top-p subset for query q, and $(K_{\bar{p}}, V_{\bar{p}})$ contains the remaining KV pairs. When performing top-p attention, we keep only the first term in equation (1), without the prefactor p. If it is the case that only a subset of tokens are relevant for a query, it is still difficult for attention to set the other tokens' probabilities precisely to zero, but they should still have far smaller scores than the relevant subset. So, we can interpret the first term in equation (1) as the true "signal" and the second term as noise. By choosing the appropriate $p < 1$, we can extract the desired signal.

We find the size of the top-p subset in each attention layer grows as a sublinear power law in the context length, with exponent depending on p (see Figure 1). Appendix A-A shows top-p curves broken down by layer and head.[1] Table I provides a complementary comparison on ZHA rollout selection counts, where power laws are preferred in many configurations but not uniformly across checkpoint geometries and query-tile widths. These fits describe router-selected leaf unions, rather than dense top-p sets.

That this sublinear growth is a power law already suggests hierarchical structure, although does not prove this structure is efficiently searchable in practice. Power laws are typically

[1]Assessing whether an entire layer or model admits a power law is more conservative than studying a single attention head. Here, we wish to see that all the layers with significant attention exhibit sublinear growth in their top-p subset sizes, establishing that compressibility extends across the model rather than being confined to isolated heads.

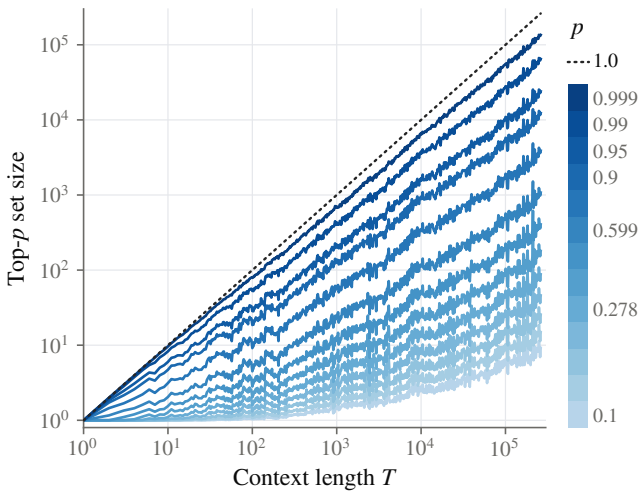


Figure 1: Dense top-p set size versus causal context length in Qwen3.6-35B-A3B on six fiction books (averaged across books, layers, and heads). We observe that at all $p < 1$ (shades of blue), the top-p set size increases sublinearly as a power law, indicating that longer sequences become increasingly compressible.

associated with scale invariance, that is, an underlying representation that remains the same at every level of analysis. Here, this would likely correspond to the statistics of some summary of tokens remaining the same as we summarize groups of tokens of geometrically increasing size. If we could learn such statistics and use them for routing, then perhaps we could route recursively through these geometric groups at increasing levels of precision according to a simple decision rule.

We note that this power-law structure does not always hold, and it seems to depend on positional encoding. Qwen3.6-35B-A3B uses partial RoPE, whereas Nemotron-3.5-Lightning-30B-A3B, which we study in Figure 12, uses NoPE and does not show as robust power-law top-p scaling, though it still performs comparably with ZHA (see Appendix B-D).

B. Sublinear Retrieval through Hierarchical Search

That attention for an individual query is sparse does not imply that most KV entries can be permanently discarded. The relevant subset can change with the query such that keys receiving negligible attention for a given query may nevertheless become important for a later query. Sparse individual attention distributions can therefore coexist with broad coverage of the cache across the union of future queries.

To assess whether sparse attention supports permanent KV eviction, we measure the union of the top-p subsets over increasing numbers of future queries, repeating this analysis for several values of p. More precisely, among sequences from long fiction books of length $T = 261{,}548$ tokens, we consider the window of the W most-recent queries ($1 \le W < T$) and study

$$U_p(W) = \frac{1}{T-W} \left| \bigcup_{q=T-W}^{T-1} S_p(q) \right|,$$

where $S_p(q)$ is the top-p subset of indices (using full causal attention) for the query at position q, but this subset of indices is restricted to the prefix $\{0, \ldots, T-W-1\}$ after selection, without renormalization. Thus, U is the fraction of the prefix covered by the union of all the top-p sets across all W most-recent queries.

If $U_p(W)$ is near 1, this suggests that nearly all keys prior to time $T-W$ are in the top-p set for some query at or after time $T-W$. We see in Figure 2 that this is inevitably the case as W grows large. That is, the union expands to cover the entire cache, indicating that entries receiving negligible attention from some queries may still be relevant to others. These results motivate retaining the full KV cache while retrieving a query-dependent subset for each attention computation.

If we are to identify the top-p subset asymptotically faster than quadratic attention, then the work needed to identify relevant entries must grow sublinearly with context length, i.e., we cannot compare every query directly to every prior key. A hierarchical, geometric tree-structured indexer offers a natural mechanism for addressing this search problem. Coarse summaries guide the query toward promising groups of keys. If a group of keys is deemed relevant via its summary, then those keys are scored again at a higher resolution, while if they are not deemed relevant, then they can all be discarded together without needing to examine them individually. This can be an effective way to reduce the number of query-key comparisons, provided that the summaries can sufficiently capture relevance.

It is not obvious a-priori that a tree is expressive enough to identify the top-p subset with sufficient accuracy, since

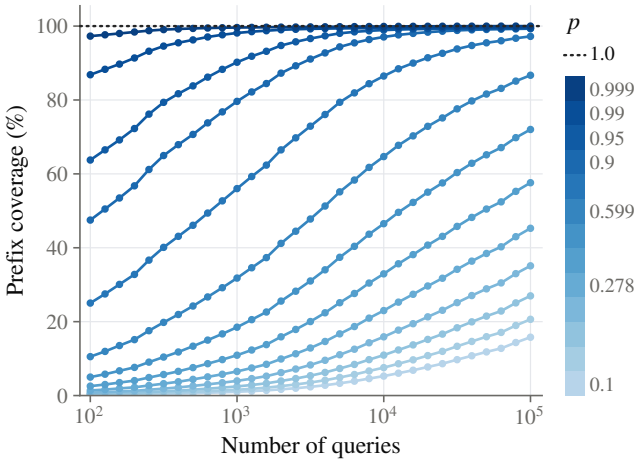


Figure 2: Fraction of KV entries of a 256K sequence covered by the union of top-p attention subsets as the number of future queries increases, for several values of p. We observe that for the higher p values we use in practice, a substantial fraction of the total KV cache is included in the union of the queries.

exact computation of this subset requires scoring the entire KV cache. However, we saw in Section II-A that the top-p subset specifically follows a power-law, which suggests an underlying tree-like structure that we aim to learn.

III. Architecture

ZHA is designed to approximate quadratic attention despite subquadratic computational complexity. Our approach involves first constructing a tree of recursively compressed keys, each of which summarizes a chunk of keys in the level below it (which may themselves be compressed). Compression is performed using chunk-wise attention scores from trained reference queries. Then, once the tree is constructed, we route through the tree with a given query by selecting the top-p nodes at a given level to expand.

We train just the hierarchical indexer, whose only learnable components are a small set of reference queries. *ZHA may therefore be applied to existing pretrained models with only a small distillation phase*; in fact, we find that a short KL warmup of only a few billion tokens is sufficient to match the base model's performance. ZHA does not require finetuning the full model, and can support a wide range of attention configurations such as different positional encodings [37], gating [34], and head-structures such as GQA, MLA, or CCA [1, 11, 28]. An overview of the ZHA architecture is shown in Figure 3.

A. Tree Construction

To construct the tree, ZHA recursively creates key summaries that compress contiguous chunks of keys in the level below it (which themselves are either key summaries or, at the bottom of the tree, the original keys). During decode, we construct the key summaries immediately upon generation of a complete chunk of keys, maintaining a buffer until sufficient keys have accumulated to construct that chunk's summary. Supposing the chunk size is C, this implies that we add a compressed key to the first level of the tree every C decode steps, to the second level every C^2 steps, and so on. Our approach is intentionally lightweight, so that tree construction adds negligible latency amortized across the sequence (around 1% or less in our roofline model).

To perform well, it is vital that the compressed keys are a faithful summarization of the underlying chunk of keys. Thus, we aim to directly approximate a chunk's total attention mass over a wide distribution of queries. To perform this approximation, we use the observation that the log unnormalized attention mass of a chunk is a convex function of the query that is naturally approximated by a piecewise-affine tiling of tangent planes. In particular, we take the maximum over a set of tangents centered at some point in the query distribution (these become our reference queries), each of which is cheaply computable by appropriately attention-weighting the keys and measuring the entropy of the attention weights at that point.

To make this concrete, denote a contiguous chunk of keys by $K_C = [k_0, \ldots, k_{C-1}]$ with chunk size C, and let R be the number of tangents and d the head dimension. Its log unnormalized attention mass for a given query is the log-sum-exp (LSE) of its scaled attention scores:

$$F_C(q) := \mathrm{LSE}_{j\in[C]}\left(q^\top k_j/\sqrt{d}\right) = \log \sum_{j\in[C]} \exp(q^\top k_j/\sqrt{d}).$$

The first-order Taylor approximation of F_C at some reference query $\hat{q}_r$ gives the tangent $t_r(q) = q^\top \hat{k}_r/\sqrt{d} + \hat{b}_r$, where the compressed key and bias are

$$\hat{k}_r = \mathrm{Attn}(\hat{q}_r, K_C, K_C) = \sum_{j\in[C]} \frac{\exp(\hat{q}_r^\top k_j/\sqrt{d})k_j}{\sum_{j'\in[C]}\exp(\hat{q}_r^\top k_{j'}/\sqrt{d})}$$

$$\hat{b}_r = F_C(\hat{q}_r) - \hat{q}_r^\top \hat{k}_r/\sqrt{d}$$

Provided a set of tangents representing a given chunk, its LSE for a future query q is approximated by $\max_{r\in[R]} t_r(q)$. To summarize a complete node at level $\ell > 1$ in the tree, which represents C^ℓ original keys, we apply the same construction to its C child summaries for each reference query, including the child biases in the attention scores and adding their attention-weighted biases to the entropy of the merge weights.[2] For models with GQA or other KV-head groupings, we construct one tree per KV head, shared by the query heads in its group. To provide some intuition for the variety of geometries, Figure 4 illustrates trees varying chunk size, depth, and number of reference queries.

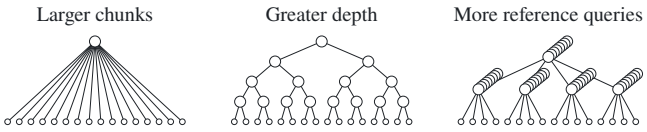


Figure 4: Illustrative ZHA tree geometries of the three core tree parameters. We can trade off depth and width for a given level of compression in the tree, since compression scales as C^L. Secondly, at the cost of additional compute, we can increase the number of reference queries.

B. Training the Module

Our chunk summarization technique relies upon a suitable choice of reference queries. Our approach is to learn them during a short KL-warmup phase; these are in fact the only learnable component in our ZHA mechanism. We find it sufficient to learn a single set of R reference queries per tree, shared by all nodes across the sequence, at all levels in the tree, and, in the case of GQA, by the query heads sharing the tree. With a reasonable number of reference queries R, the ZHA parameters add a negligible increase to the parameter count of the models we test.[3] For models that incorporate RoPE, we rotate the shared reference queries to the last token position of each node's span, at every tree level. Let $\hat{q}_{r,c}$ denote

[2]When the same reference query is used throughout a subtree, this merge recovers the summary computed directly from its original keys. With chunk-dependent RoPE rotations, it instead gives an affine lower bound that need not be tangent to the original chunk's log attention mass.

[3]For example, with $R = 16$ and one tree per KV head in Qwen3.6-35B-A3B, the parameter count increases by approximately 0.00024% (81,920 / 34.66B).

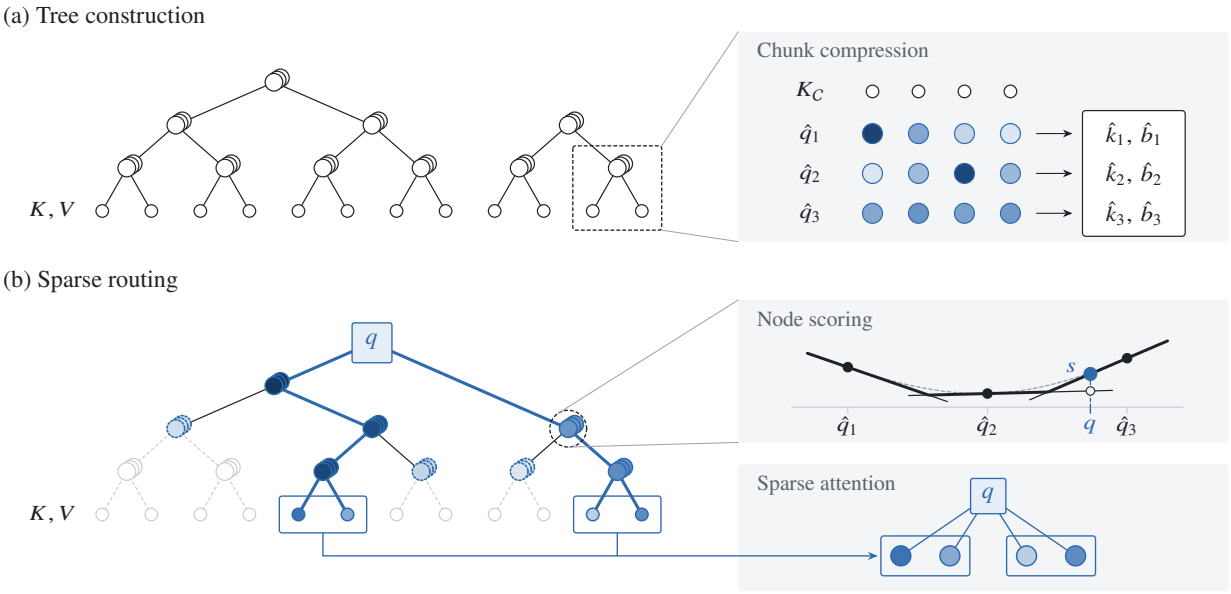


Figure 3: Overview of the ZHA architecture: (a) tree construction and (b) sparse routing. During tree construction, for each chunk of contiguous key nodes, we construct a compressed key node at the higher level by an attention-weighted sum of the keys, where the attention is computed with respect to the learned reference queries. In practice, we use multiple reference queries and store the key nodes for each in parallel. This process continues recursively with higher level key nodes being constructed from the summaries of lower levels. During routing, we start at the top of the tree and compute dot-product similarity scores between our query and the key nodes of that level. We select the set of nodes to expand using top-p, and then recursively descend the tree until we reach the leaf nodes corresponding to raw KV cache entries. The union of all the leaves we reach defines the sparse KV cache subset that we dispatch to the main attention.

the reference query r rotated for node c. Before merging the children of a parent node P, we evaluate all child summaries at the parent's rotated reference query $\hat{q}_{r,P}$. Each child c was originally summarized at its own rotated reference query $\hat{q}_{r,c}$. To evaluate its affine approximation at the parent's query, we add $\frac{1}{\sqrt{d}}\langle \hat{k}_{r,c}, \hat{q}_{r,P} - \hat{q}_{r,c} \rangle$ to its estimated log attention mass. This correction is simply the compressed key's score at the parent's reference query minus its score at the child's reference query.

From a trained dense-attention model, the ZHA conversion procedure is remarkably simple and compute-efficient. In particular, we freeze the base model (including all native attention components) and learn the reference query parameters by matching the predicted chunk probabilities to the true attention distribution pooled into chunks at each level. For a query q, let $a_j(q)$ denote its dense attention weight on key j, and let $\mathcal{P}_\ell(q)$ partition the original key indices visible to q into chunks $\mathcal{C}$ at tree level ℓ. The target and predicted chunk probabilities are

$$p_{\ell,\mathcal{C}}(q) = \sum_{j \in \mathcal{C}} a_j(q), \quad \hat{p}_{\ell,\mathcal{C}}(q) = \frac{\exp s_{\ell,\mathcal{C}}(q)}{\sum_{\mathcal{C}' \in \mathcal{P}_\ell(q)} \exp s_{\ell,\mathcal{C}'}(q)},$$

where $s_{\ell,\mathcal{C}}(q) = \max_{r \in [R]} t_{\ell,\mathcal{C},r}(q)$ is the chunk's approximate log attention mass.[4] We supervise the shared reference queries at every tree level by minimizing

$$\mathcal{L}_{\mathrm{KL}} = \sum_{\ell=1}^{L} \mathbb{E}_q [D_{\mathrm{KL}}(p_\ell(q) \,\|\, \hat{p}_\ell(q))] \,.$$

Here L is the number of tree levels, and the expectation averages over eligible queries, attention heads, and model layers at each tree level. In practice, learning the reference query parameters requires only a few billion tokens, and can be done at sequence lengths significantly shorter than pretrained backbone's full context length.[5]

C. Routing Through the Tree

Having constructed the tree, we must now sparsely traverse it during inference. A given query retrieves from the tree by scoring nodes and selecting a subset of nodes at each level, recursively descending by repeating this process for the selected nodes' children, until terminating at a sparse subset of the original KV, from which we compute the attention output.

In addition to the low-latency yet expressive attention mass approximation and the multiple levels of hierarchy, a core innovation of this work is that we utilize top-p as our selection mechanism (described in Section II). Top-p allows the selection size to adapt to the attention distribution and sequence length,

[4] For the current incomplete chunk, we use its exact log attention mass in place of $s_{\ell,\mathcal{C}}(q)$.

[5] In the open-source models we tested, which range from 4B to 35B parameters and were trained at context lengths from 256K to 512K, we trained for 4B tokens at 32K context length, with the reported Qwen and Nemotron runs using a peak learning rate of 3.2×10^{-2} and the WSD schedule described in Appendix B-A.

in contrast to fixed-selection methods such as top-K [27]. If only a few nodes contain most of the mass, top-p is not obliged to select marginal nodes to satisfy a fixed K budget. Conversely, if attention mass is near-uniform, a large fraction of the nodes at a given level can be selected for expansion. We posit that this property enables length extrapolation beyond that seen in other methods, which we confirm for a variety of open-source models in Section IV. The hierarchy also keeps the ordering cost of top-p selection manageable. Pruning at each level limits the number of candidate nodes considered at the next, reducing the size of subsequent ordering operations. In practice, we find that these candidate sets remain small enough to make top-p selection feasible at every level of the tree.

The concrete routing procedure is as follows. Each q denotes one query head's query; selection and child expansion remain independent across query heads, including those sharing a KV tree. Let $\mathcal{A}_\ell(q) = \{\mathcal{C}_1, \ldots, \mathcal{C}_{n_\ell}\}$ denote the complete candidate chunks considered at level ℓ, indexed in decreasing order of score $s_{\ell,\mathcal{C}}(q)$. Note that the ordering and $n_\ell = |\mathcal{A}_\ell(q)|$ vary depending on q, and there may be incomplete chunks at each level in the tree depending on the exact sequence length. At each level, we pick the top-p nodes; i.e., for chosen $p \in (0, 1)$, we select chunks

$$\mathcal{S}_\ell(q) = \{\mathcal{C}_1, \ldots, \mathcal{C}_{m_\ell(q)}\},$$

where

$$m_\ell(q) = \min\left\{ m \in [n_\ell] : \frac{\sum_{i=1}^{m} \exp s_{\ell,\mathcal{C}_i}(q)}{\sum_{i=1}^{n_\ell} \exp s_{\ell,\mathcal{C}_i}(q)} \geq p \right\}.$$

Thus, the selected chunks form the smallest prefix containing at least a fraction p of the estimated attention mass among the current candidates. Let $\mathcal{U}_\ell(q)$ denote the complete chunks at level ℓ with no complete parent. Starting with all complete top-level chunks, $\mathcal{A}_L(q) = \mathcal{U}_L(q)$, we pass the selected chunks' children and any parentless chunks to the next level:

$$\mathcal{A}_{\ell-1}(q) = \left(\bigcup_{\mathcal{C} \in \mathcal{S}_\ell(q)} \mathrm{ch}(\mathcal{C}) \right) \cup \mathcal{U}_{\ell-1}(q), \qquad \ell = L, \ldots, 2,$$

where $\mathrm{ch}(\mathcal{C})$ denotes the child chunks of $\mathcal{C}$. We exclude the first key (index 0) from the tree construction and instead always retain it[6] together with the visible keys in the current chunk, $\mathcal{C}_{\mathrm{cur}}(q)$, giving the token indices

$$\mathcal{I}(q) = \{0\} \cup \left(\bigcup_{\mathcal{C} \in \mathcal{S}_1(q)} \mathcal{C} \right) \cup \mathcal{C}_{\mathrm{cur}}(q).$$

During decode, query heads sharing a KV head jointly load the union of their final selected leaf chunks; this sharing does not merge their upper-level routing selections or attention masks.[7]

[6]In many models, the first token acts as an attention sink, receiving a large share of attention mass regardless of content; we found it beneficial to always attend to it.

[7]For block prefill, loads are shared across query positions within each head's tile, with per-query masks retained.

Each head computes standard scaled dot-product attention over its own selected KV:

$$o(q) = \mathrm{Attn}\big(q, K_{\mathcal{I}(q)}, V_{\mathcal{I}(q)}\big).$$

IV. Evaluations

We evaluate ZHA on a suite of reasoning and retrieval tasks with context lengths extending to hundreds of thousands of tokens, demonstrating that the module can match the dense model while attending to a small fraction of the total KV cache. Across these settings, ZHA is impressively robust to choice of geometry, whose effect on retrieval accuracy is nonetheless highly predictable by a simple model. These results suggest that, not only does ZHA enable efficient retrieval from past context, this retrieval remains effective out to very long contexts.

A. Reasoning and Retrieval

For the primary suite of evaluations, we convert Qwen3.6-35B-A3B [35] with ZHA routers of varying tree geometry. To start, we train both deep (chunk size 4, six tree levels; C4 L6) and wide (chunk size 256, one tree level; C256 L1) routers, each with $R = 16$ reference queries and trained on 4B tokens with the backbone frozen (for more training details, please see Appendix B-A). For our evaluations, we vary $p \in \{0.8, 0.9, 0.95\}$, constant at every tree level. Both routers perform within noise of the dense baseline on reasoning, agentic coding, and long-context benchmarks. The exception is RULER-13 for the deep router, which scores 93.3% versus 96.0% for dense; however, this is just a 3% drop in performance while attending to 1%-6% of the original KV cache. See Figure 5 for the full suite of evaluation results for ZHA (deep and wide) and dense.

We also compare ZHA against Minimax Sparse Attention (MSA) [25], which selects the top 16 blocks of 128 tokens using a learned indexer. We use the same Qwen backbone and train MSA's indexer on 40B tokens using only the KL warmup stage, with the backbone frozen. MSA stays close to dense attention on AIME 2026, MMLU-Pro, and GPQA Diamond, but falls well behind both ZHA routers on long-context retrieval. On LongBench v2, where MSA's fixed budget amounts to roughly 1% of the context, the deep router attends to a similar 1.1% of decode keys yet scores 14 points higher, even though MSA was trained on 10x the token budget of ZHA. For comparison, MSA is also included in Figure 5.

Finally, we also train ZHA onto Nemotron-3.5-Lightning-30B-A3B [33], to assess applicability to an architecture without explicit positional encodings. Generally Nemotron performs comparably to its baseline on the agentic coding benchmarks, but suffers moderately on the RULER benchmark. Full results for Nemotron are reported in Appendix B-D.

B. Effect of Geometry and Sparsity

Thus far we have shown that ZHA is competitive with dense attention, whether the tree is wide (with chunk size 256) or deep (with 6 levels), and that these variants remain competitive even while attending to a fraction of the native KV cache. However, the space of tree geometries, spanned by choice of reference

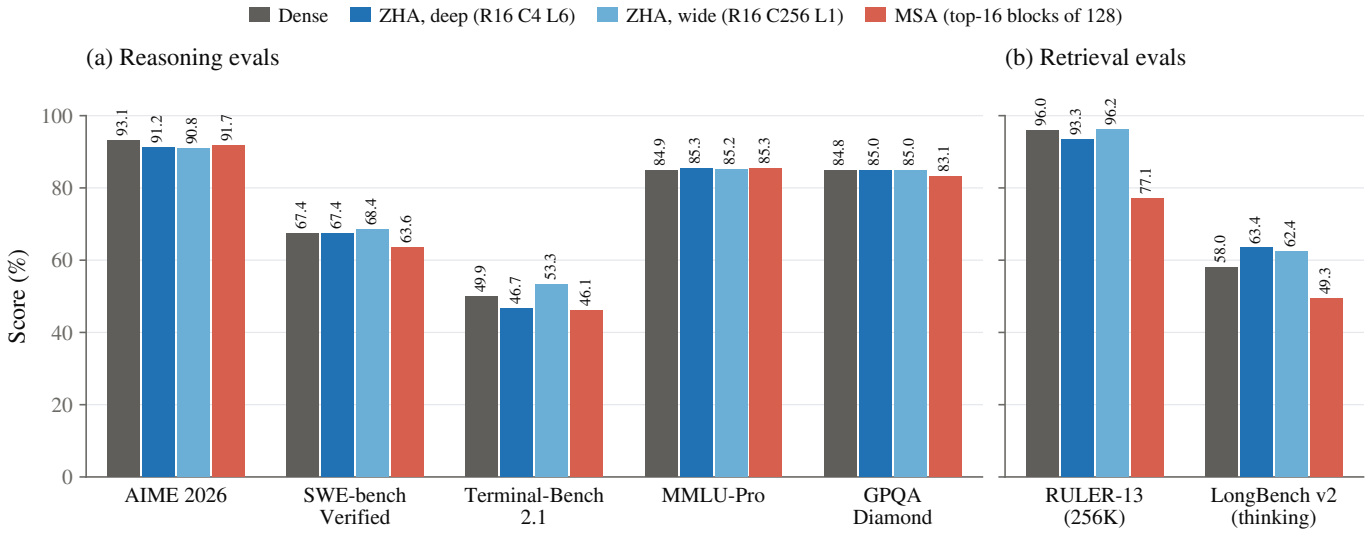


Figure 5: (a) Reasoning and (b) retrieval evaluation scores with dense attention, MSA, and deep (C4 L6) or wide (C256 L1) ZHA ($p = 0.95$, $R = 16$, converted from Qwen3.6-35B-A3B). Appendix B gives evaluation details and results across routing thresholds. We observe that ZHA with both wide and deep geometries maintains performance even at high sparsity levels, while MSA suffers performance degradation especially on retrieval tasks such as RULER and LongBench v2.

count R, chunk size C, tree depth L, and routing threshold p, is combinatorially complex. Despite preliminary evidence of robustness, it is easy to see why variation along any of the axes above may have a direct effect on performance: each changes the effective compression and sparsity in different ways, and each may introduce error in the router's approximation to the true top-p set. Moreover, depth and width introduce potentially qualitatively different kinds of error into the approximation. Here, we explore the effect of geometry and sparsity along a few particularly revealing slices of the design space; then, to demonstrate the predictability of performance as a function of these variables, we develop a simple yet accurate model of RULER accuracy in terms of (R, C, L, p) that we fit with just four parameters.

Robustness across geometries. To more thoroughly probe the effect on performance, we sweep a wide range of geometries by varying R, C, and L (this yields 28 different configs; see Figure 18 in Appendix C for the 28 used for fitting). For each, we use the same 4B-token KL warmup and then evaluate on RULER (at lengths of 64K, 128K, and 256K) while varying $p \in \{0.7, 0.8, 0.9, 0.95, 0.99\}$. Across 13 tasks at 256K, the aggregate RULER accuracy as a function of percentage of decode keys attended is shown in Figure 6(a), color-coded by the top-level node span C^L, our nominal compression factor. The primary observation is that, when attending to between 10–30% of keys, retrieval performance is robust to choice of geometry. Beyond this, performance generally degrades with increasing key sparsity, with the rate depending on both top-level compression and its decomposition into C and L. Figure 6(c) isolates the effect of reference count R at fixed $C = 8$ and $L = 2$. Figure 20 in Appendix C shows the 256K RULER Pareto frontiers across geometries for decode keys attended and modeled speedup over dense.

Depth at fixed compression. Because ZHA's subquadratic scaling properties rely on increasing tree depth, a critical question for ZHA's promise is whether depth specifically leads to sharp degradation in retrieval. To isolate the effect of tree depth L from top-level compression C^L, Figure 6(b) shows RULER accuracy versus decode keys attended for a slice of ZHA geometries with fixed $C^L = 4096$. In fact, it appears that increasing depth enables sustaining effective retrieval out to greater sparsity and that the deeper geometries are more performant at a fixed sparsity level. However, this alone does not suggest that ZHA can indefinitely increase depth while scaling to longer contexts (as this increases top-level compression C^L at fixed C), but it does provide strong evidence that depth itself is not the limiting factor. Notably, to our knowledge, our approach is the first hierarchical attention variant to have demonstrated convincing scaling to depths greater than 1 while maintaining close parity to dense attention.

Predicting retrieval accuracy. To quantify and generalize the accuracy-sparsity tradeoff, we develop a simple model predicting RULER accuracy from input (R, C, L, p). The model is

$$d_\ell(R, C) = \log\left(1 + \frac{C^\ell}{R^r}\right),$$
$$\widehat{A} = A_{\text{dense}}(N) \prod_{\ell=1}^{L} \exp\left[-a(1-p)^b d_\ell^h\right]. \tag{2}$$

Intuitively, the term $\log(1 + C^\ell/R^r)$ represents compression difficulty at level ℓ, increasing with node span and decreasing with reference capacity. The product compounds per-level accuracy retention over the tree. The factor $(1-p)^b$ accounts

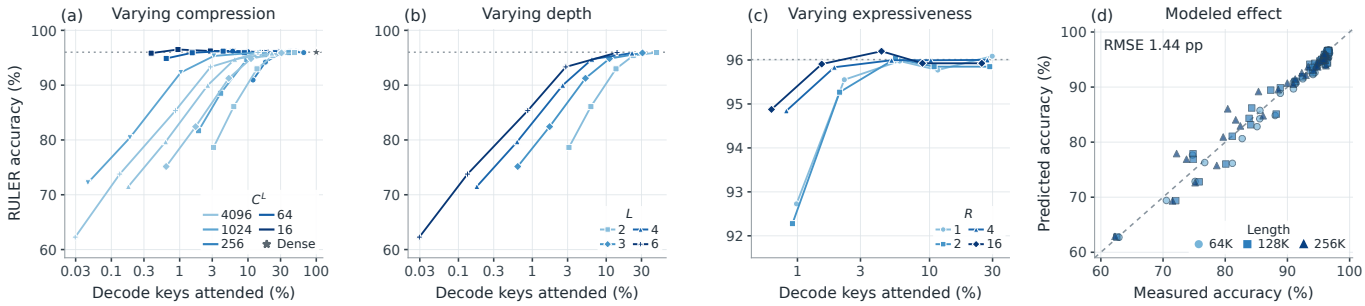


Figure 6: Geometry and sparsity have predictable effects on RULER accuracy (mean across tasks). Greater sparsity without degrading performance is achieved by: (a) decreasing compression C^L, (b) increasing depth at fixed compression ($C^L = 4096$), or (c) making key summaries more expressive (with increasing reference-query number R) at fixed chunk size and depth. (The horizontal dotted lines in panels (a)–(c) show dense attention.) (d) For the simple geometry-to-performance model, predictions versus measurements across 64K, 128K, and 256K, holding out each length in turn (RMSE: 1.44 percentage points).

for the effect of sparsity, whose penalty can grow nonlinearly.[8] We fit four parameters: a, which specifies the overall penalty strength; b, which controls sensitivity to sparsity; h, which controls sensitivity to each level's difficulty; and r, which controls reference capacity. Each exponential maps a level's penalty to an accuracy-retention factor; their product scales the dense-attention accuracy $A_{\text{dense}}(N)$ at that sequence length (N). Aside from length-dependence inherited from this baseline, sequence length does not enter this simple model.

We fit the model using 28 geometries from the sweep across $R \in \{1, 2, 4, 16\}$, evaluated for 100 RULER examples per task at $p \in \{0.7, 0.8, 0.9, 0.95\}$ and lengths (64K, 128K, 256K), together with $p = 0.99$ evaluations for 13 of these geometries. Fitting uses settings complete at all three lengths; the empirical panels include complete 256K settings. We fit the four parameters (constraining $a > 0$, $1 \le b \le 2$, $0.2 \le h \le 6$, and $0 \le r \le 6$) by nonlinear least squares on accuracy (in percentage points). Here we fit to the 13-task mean, and Appendix C fits each task independently with the same form. To evaluate held-out prediction accuracy (for the aggregate and per-task fits separately) and particularly length transfer, we measure predicted versus actual RULER accuracy on a held-out sequence length while fitting on the other two lengths; we do this separately for the three sequence lengths, and for each we fit on 250 cells (125 geometry-threshold settings at 2 lengths) and evaluate on 125 cells. Each fold reuses the same geometries and thresholds at the other lengths and uses the measured dense baseline at the held-out length, so this tests transfer of the relative accuracy penalty across lengths conditional on that baseline. Figure 6(d) shows all 150 $R = 16$ leave-one-length-out predictions pooled across the three lengths, with an RMSE of 1.44 percentage points (the 256K subset has an RMSE of 1.65 percentage points). The relatively low prediction-error of such a simple model suggests that the accuracy varies surprisingly smoothly with sparsity and tree geometry over the fitted range.

[8] We constrain $1 \le b \le 2$ to retain flexibility while avoiding unstable low-p extrapolation for near-ceiling tasks.

V. Speedup

ZHA is designed to achieve fundamentally subquadratic latency scaling, in contrast to the variety of chunk-sparse methods that yield significant but constant-factor speedups. Yet, however principled this design, its practicality depends on whether it can run efficiently on GPUs. While we do not release optimized kernels in this work, we do investigate the question of efficient GPU implementation via a detailed roofline model that enables extrapolation of ZHA's latency scaling in comparison to dense attention and other baselines.

A. Roofline Model

Although we leave the full details to Appendix D (which provides a treatment of the top-p selection algorithm among other specifics), we briefly describe the roofline model we use to estimate how ZHA latency scales with sequence length. Suppose T is the context length, $L(T)$ the ultimate tree depth, and ℓ the routing level. The modeled ZHA attention-subsystem latency $\boldsymbol{t}_{\text{ZHA}}$ is given by a sum reflecting its serial execution through the tree:

$$\begin{aligned}\boldsymbol{t}_{\text{ZHA}}(T) = t_{\text{tree}}(T) + \lambda_A + \boldsymbol{\Phi}_{\text{attn}}(T) \\ + \sum_{\ell=1}^{L(T)} \left[\lambda_\ell + \boldsymbol{\Phi}_{\text{routing},\ell}(T) + \boldsymbol{\Phi}_{\text{scan},\ell}(T)\right]\end{aligned} \tag{3}$$

Above, t_{tree} accounts for prefill construction or amortized decode maintenance, including launches. λ_ℓ, λ_A measure the per-level and final-attention launch overheads, respectively. The three kinds of execution stages are reference scoring $\boldsymbol{\Phi}_{\text{routing},\ell}$, node selection $\boldsymbol{\Phi}_{\text{scan},\ell}$, and retained-token exact attention $\boldsymbol{\Phi}_{\text{attn}}$.

Each constituent kernel or pass $\boldsymbol{\Phi}_x$ (indexed by x) follows a three-term roofline model accounting for compute and memory bandwidth along with a worker-wave term $\boldsymbol{t}_{x,\text{worker waves}}$ that accounts for finite block concurrency:

$$\boldsymbol{\Phi}_x(T) = \max\left\{\begin{array}{l}\mathbf{FLOPs}_x(T)/P_x, \\ \mathbf{bytes}_x(T)/\text{BW}_x, \\ \boldsymbol{t}_{x,\text{worker waves}}(T)\end{array}\right\}. \tag{4}$$

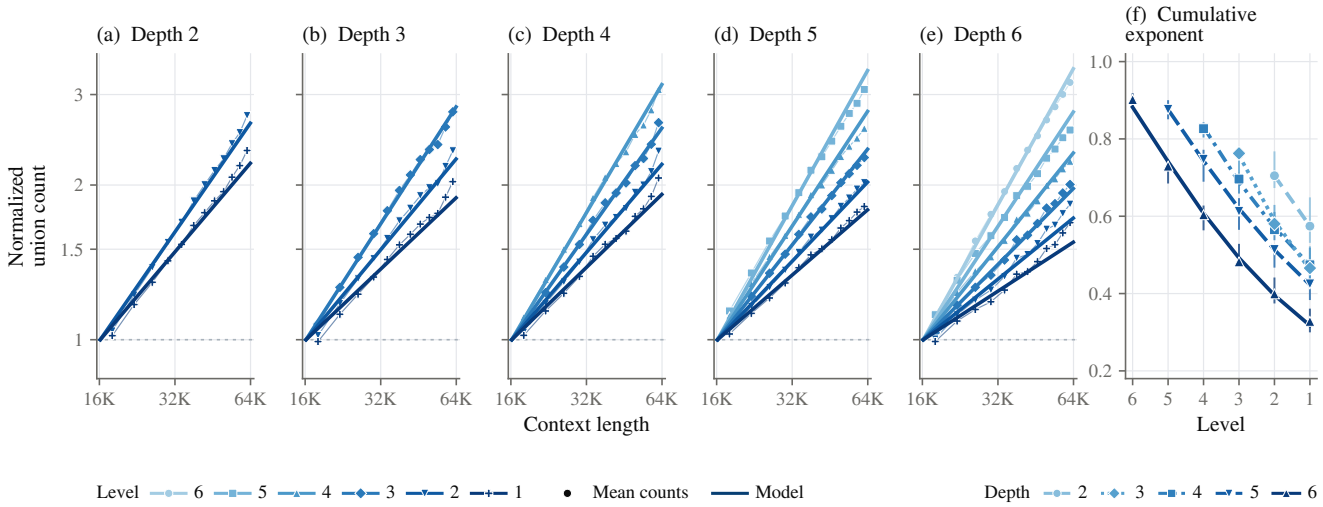


Figure 7: Per-level selection scales sublinearly in the ZHA router with predictable power-law exponent. (a)–(e): For ZHA geometries with depths $L = 2$–6 ($C = 4$, $R = 16$, $p = 0.8$), per-level selection counts (normalized by first-bin count per-level) on AIME problems fitted by power-law scaling curves. (f): Per-level exponents fitted by cumulative top-down model, demonstrating that the across-depth variation in their exponents is well-described by a simple model that accumulates top-down contributions.

Above, $\mathbf{FLOPs}_x$ denotes the floating-point operation count and P_x the effective FLOP/s (scalar for scan); similarly, $\mathbf{bytes}_x$ denotes the HBM traffic and BW_x the effective bytes/s. For stages with sequential kernels or passes, the execution times are summed.

Though the compute and memory terms are familiar to any roofline model, it's worth noting why the worker-wave term is of particular importance for ZHA. This term is

$$\boldsymbol{t}_{x,\text{worker waves}}(T) = \boldsymbol{\tau}_x(T) \left\lceil \frac{\boldsymbol{N}_{\text{tiles},x}(T)}{N_{\text{SM}} c_x} \right\rceil, \tag{5}$$

where $\boldsymbol{N}_{\text{tiles},x}$ is the number of thread blocks, $N_{\text{SM}} c_x$ the number that can execute concurrently, and $\boldsymbol{\tau}_x$ the per-block execution time at that concurrency. The ceiling therefore counts the sequential waves needed to execute the stage. This matters for ZHA because pruning at one level reduces work at the next, potentially leaving individual stages with limited parallelism and thus idle SMs in the higher levels of the tree.

B. Sparsity Scaling

Predicting latency therefore requires understanding how the number of nodes selected at each level scales with sequence length. Extrapolating this behavior to longer sequences requires observing how the selected top-p set scales at each level, and then modeling how this scaling varies across levels, particularly as we increase the tree's ultimate depth.

Empirical per-level exponents. Inspired by the robustly sublinear scaling observed in the top-p selection set for dense models (see Section II), we fit a power law to the number of nodes selected per level. We fit these curves with per-level KV-group union counts collected across heads and layers of ZHA generations to AIME 2026 questions (response lengths 16K–64K), with geometries of $C = 4$, $R = 16$, and depths $L = 2$–6 (a subset of the Qwen3.6-35B-A3B conversions from Section IV-B).

For a fixed geometry and routing threshold p, let $S_\ell(T)$ denote the mean selected-node count per query at context length T and level ℓ, counting the union within each KV group at every level and averaging over groups (with $\ell = L$ at the top and $\ell = 1$ at the leaf chunks). Our power-law summary $\widehat{S}_\ell(T)$ has the form

$$\widehat{S}_\ell(T) = c_\ell T^{\alpha_\ell}, \tag{6}$$

where c_ℓ is a constant factor and α_ℓ is the scaling exponent that we fit to the empirical selection counts.[9] Figure 7(a)–(e) shows the empirical number of selected nodes for the (color-coded) levels across fixed-C, different-depth geometries. We also plot lines denoting each level's fitted depth-model power law, demonstrating good quality-of-fit to the selection counts.

Modeled per-level exponents. In order to extrapolate ZHA's latency to longer sequences with increasing tree depth, we develop a model for how the exponent varies per-level. Using the same variable-depth geometries as a case study, we begin by modeling per-level exponents given some fixed ultimate depth L. Notice that, in Figure 7, the scaling exponent is highest at the top level and decreases proceeding down each tree, consistently for each geometry.[10] This pattern inspires our approach to modeling the per-level exponent $\widehat{\alpha}_\ell$ as a product of

[9] In particular, we estimate observed exponents α_ℓ by fitting each problem's log counts against log context length with $1/T$ weights, then taking the median slope across problems at each level. See Appendix E for more details.

[10] Figure 7 shows $p = 0.8$; the corresponding data for $p \in \{0.7, 0.9, 0.95\}$ may be found in Figure 22 in Appendix E and broadly corroborates these same trends. The exception is the deepest geometry ($C = 4$, $L = 6$, $R = 16$) at $p = 0.7$, whose exponents rise again toward the leaf levels. This setting achieves only 44.0% accuracy on AIME, compared with 93.1% for dense attention, so its anomalous scaling should be interpreted in the context of severe accuracy degradation.

top-down contributions by top-level exponent β and per-level factors β_ℓ:

$$\begin{aligned} \varepsilon_\ell &= 2\varepsilon\left(1 - 2^{-(L-\ell)}\right), \\ \beta_\ell &= \beta + \varepsilon_\ell, \qquad \varepsilon_L = 0, \\ \widehat{\alpha}_\ell &= \prod_{k=\ell}^{L} \beta_k, \qquad \widehat{\alpha}_L = \beta. \end{aligned} \tag{7}$$

This product includes the current level and every level above it: $\widehat{\alpha}_{L-1} = \beta_{L-1}\beta$ and $\widehat{\alpha}_{L-2} = \beta_{L-2}\beta_{L-1}\beta$. To account for sparsity introduced by higher-level pruning and KV-group unioning, ε_ℓ is a saturating additive correction to the factor β_ℓ. We fit the overall shape parameters (β, ε) separately for each geometry and p.[11] In Appendix E, we compare to more general additive corrections, from a constant additive factor to increasing linear drift.

Figure 7(f) shows the empirically-estimated per-level exponents α_ℓ along with their fitted cumulative form $\widehat{\alpha}_\ell$ for all five depths; note that this form accurately captures the across-level exponent trends at $p = 0.8$. Appendix E completes the comparison across all four routing thresholds and gives the problem counts, full-resolution fitting, display binning, and bootstrap details.

Crucially, to predict sparsity scaling for longer sequences, we must also extrapolate the within-tree profiles above to increasingly deeper trees. Given the sparsity of available data (20 measurements, spanning five depths per p), we present upper and lower bounds rather than attempt a tight fit. In particular, we parameterize the top exponent as

$$\widehat{\beta}(L, p) = b_0 + b_p p + [d_0 + d_p p]\, L. \tag{8}$$

The constant-top scenario sets $d_0 = d_p = 0$; the linear-top scenario fits all four coefficients directly to the observed top-level exponents α_L, independently of the within-tree profile fit. To isolate the effect of ultimate depth, we fit this model on the top-level exponents across the variable-depth geometries (the same 20 measurements referred to above); Appendix E reports the exact coefficients and validation procedure. From this limited data, linear-top is better than constant-top, though neither provide a particularly good fit. However, they jointly provide a loose bound on the top-level exponent with increasing depth, which we use to extrapolate ZHA's latency out to very long contexts.

C. Latency Comparison

Given this roofline model and selection scaling framework, we compare the theoretical latency scaling of ZHA to dense attention and MSA for sequence lengths up to 10^9 tokens. For this analysis, we restrict to a light-weight ZHA geometry ($R = 1$, $C = 8$, $p = 0.75$; Qwen 35B with BF16) that retains high performance across our long-context reasoning and retrieval evaluation suite (see Figure 15 in Appendix B), and compare its modeled latency to dense attention and MSA. To model the per-level exponent extrapolation with depth, we transfer the $C = 4$, $R = 16$ AIME exponent fits and calibrate count amplitudes to $R = 1$, $C = 8$, $L = 2$ AIME measurements. At each sequence length, we choose the ultimate tree depth that minimizes latency. As shown in Figure 8, ZHA becomes faster than dense attention at approximately 48K tokens and MSA at 0.48M tokens, reaching about 373–417× and 52–58× speedup, respectively, at 10^9 tokens. The selected depth is $L = 5$ at 10^9 tokens in both scenarios. Figure 24 in Appendix E compares $p = 0.7, 0.75, 0.8$.

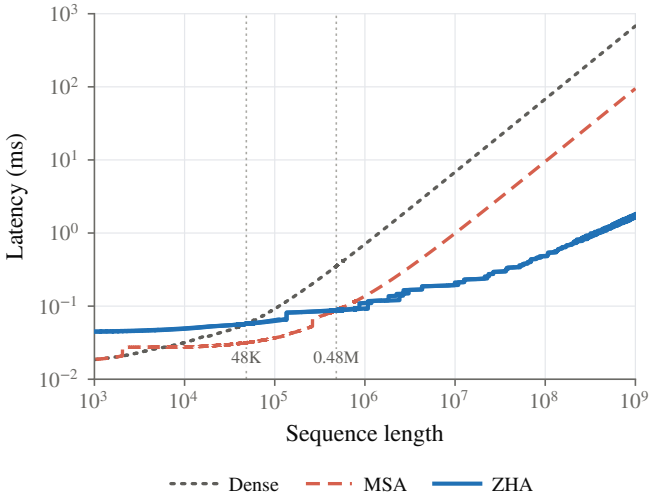


Figure 8: For a fixed top-p, and hence sparsity level, ZHA (here, $R = 1$, $C = 8$, $p = 0.75$) scales subquadratically according to our decode roofline model, eventually dramatically outperforming both dense attention and the MSA sparse attention variant. Importantly, we have not empirically validated the scaling at these ultra-long context lengths due to the lack of existing LLMs which can operate at these contexts. Because of this, while we can obtain subquadratic latency, it is unclear to what extent reasoning and retrieval performance degrades at the level of depths required for these ultra-long contexts. Latencies are modeled per-token decode times for the attention subsystem of one layer at batch size 1.

VI. Analysis

Several design choices in the development of ZHA rely upon assumptions about the underlying attention distribution, and therefore how best to approximate it with subquadratic compute. Here, we empirically investigate two such phenomena: the dynamic sparsity introduced by top-p, and the learned distribution and usage dynamics of the router's reference queries.

A. Variable Sparsity

In contrast to fixed top-K thresholds such as MSA, ZHA's top-p threshold dynamically varies the number of attended keys. Not only does this facilitate proper scaling of retrieval

[11] Specifically, we fit (β, ε) for each geometry by equally weighted least squares against the empirical exponents α_ℓ, allowing signed ε while keeping every β_ℓ positive, with an independent amplitude c_ℓ calibrated to pooled mean counts by $1/T$-weighted least squares in log space at each level. See Appendix E for more details.

with sequence length, this also enables adaptive retrieval depending on the query and preceding context. The utility of the latter depends on the extent to which the underlying attention dynamics have differing sparsity depending on the setting and task, and whether the model can adaptively route through the tree according to these dynamics.

First, we compare attention sparsity across the evaluation domains, which require retrieval patterns ranging from narrow lookup to diffuse aggregation. At context lengths of approximately 64K tokens, 90% of a query head's dense attention mass is concentrated in a median of 2.3% of 16-key chunks on RULER, 11.7% on SWE-bench Verified, and 15.0% on fiction books.[12] A representative ZHA geometry ($R = 16$, $C = 16$, $L = 3$), also at $p = 0.9$, retains 0.44%, 5.0%, and 7.4% of chunks, respectively. These chunks contain a median of 80%, 78%, and 78% of dense attention mass, with individual RULER tasks ranging from 74–82%. Though the router is consistently sparser than the dense top-p set (reflecting iterative pruning across the three levels), the pattern across tasks is consistent. Furthermore, the variation for both dense and ZHA matches our intuition: in RULER, its synthetic filler carries little that the answer depends on; in agentic contexts like SWE-bench, the relevant information is spread over code, tool outputs, and earlier reasoning; and in fiction books, the information is diffuse throughout the entire context.

RULER provides a particularly controlled setting in which to isolate how specific tasks induce different attention distributions such as either diffuse or concentrated. As such, we compare the size of ZHA's retrieved key set to the geometric mean over all 13 RULER subtasks. The tasks fall into three sparsity tiers: tasks with a single target in repetitive filler (`variable_tracking`, `niah_single_1`) read about half as many keys as the typical task; needles in essay text and document QA sit near the middle (0.82–1.26×); and tasks that must scan many candidates, either aggregating items (`cwe`, `fwe`) or distinguishing many look-alike needles (`niah_multikey_2/3`), read about 1.6× as many. To concretely illustrate how ZHA achieves variable sparsity, Figure 25 depicts a representative attention head's traversal through the tree for `niah_single_1` and `niah_multikey_2`; both traversals ultimately attend to the correct answer-containing chunk, but the former tree is far more aggressively pruned than the latter. Note that this adaptive sparsity is impossible for a fixed-budget top-K mechanism, which instead forces a tradeoff between a large budget wasting keys on concentrated inputs, and a small one missing context needed for diffuse retrieval. ZHA's top-p threshold avoids this choice entirely.

B. Reference-Query Structure

Another question concerns the nature of the learned structure in the reference queries, which comprise the only learned components in the router and therefore are critical for its efficacy. Recall that each reference query produces a compressed key and bias defining an affine approximation to the chunk's log attention mass. Taking the maximum over these approximations yields a piecewise-affine approximation to the true function. In this setup, each reference query serves as a tangent plane's center point, and therefore it might reflect some central tendency in the query distribution. However, this maximum over the reference-query-mediated scores in the KL training objective does not explicitly enforce balanced usage or distinct reference query representations. In the worst case, they might simply collapse while still adding a constant-factor computational cost.

Thus, a reference query is only useful insofar as it is selected by the max-pooling operation. We therefore measure the selection rates of reference queries across levels of the tree, using a sample of length-32K natural language documents from our training mix.[13] The reference queries develop a coherent pattern of utilization both within and between tree levels; this is summarized in Figure 9. In particular, the selection rate among reference queries within a level follows a gradual decay, which becomes increasingly concentrated among a few reference queries at higher tree levels and more balanced toward the leaves. Furthermore, comparing the rank correlation in the selection rate orderings between levels, the higher levels consistently use the same reference queries most and least commonly, whereas the lowest level's usage pattern diverges, suggesting different selection dynamics depending on the level's compression factor. The across-level usage patterns are further broken down in Appendix G, demonstrating rich selection-rate structure in which reference query usage varies smoothly across levels.

Moreover, we investigate the learned structure, using a deep geometry ($R = 16$, $C = 4$, $L = 6$) with moderate sparsity ($p = 0.9$) as a case study. First, Figure 10 summarizes their geometric variation. Comparing the reference queries (here, 16) in a KV group; their significant pairwise cosine similarity, with a mean pairwise cosine similarity of 0.44 (averaged over all 20 KV groups), show they occupy a cone of directions rather than collapsing to identical vectors. This structure emerges across geometries and independently trained routers with varying chunk sizes and depths learn closely matching reference queries (Appendix H). This similarity across geometries is clearest for the most frequently selected references. The reference query norms have some spread while remaining fairly concentrated. Interestingly, selection also favors smaller-norm reference queries. Since smaller norms lead to smoother weighting, it may indicate that more uniform averaging of keys becomes advantageous at higher levels. Together, the distribution in cosine similarity and norms suggests that the reference queries have organized into a coherent and non-collapsed geometric structure.

[12] We exclude key 0 and the current partial chunk, which ZHA always attends, and record dense attention along the router's own decode. Medians are taken over decode steps, the 10 full-attention layers, and query heads within each sequence, then across sequences.

[13] 133 documents of at least length-32K in one training-mix shard; these are chat-formatted, mostly long mathematical reasoning, plus general chat, agentic coding, and long-document QA. At every 8th query position, we count which reference query wins each kept node. As the routers may have seen this data, Appendix G repeats the analysis on RULER.

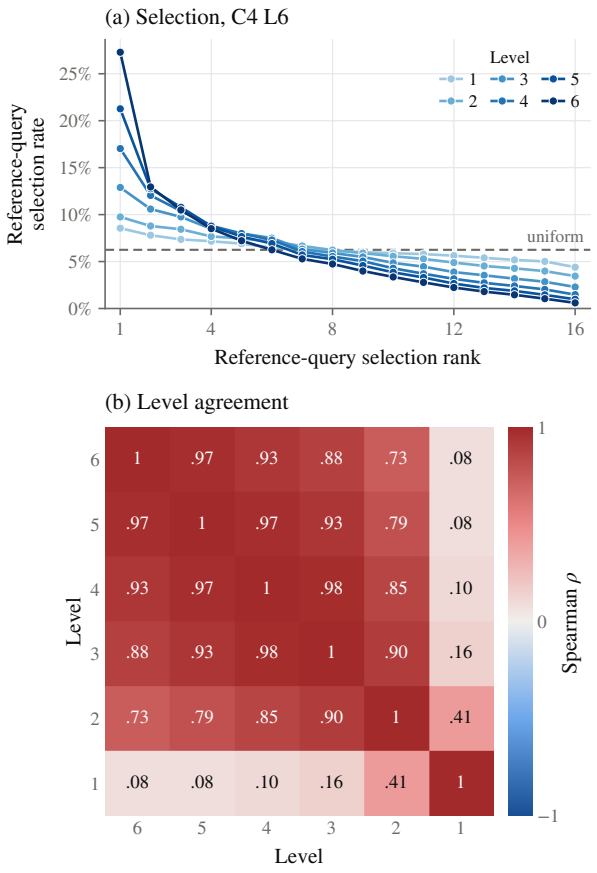


Figure 9: Reference-query selection rates per tree level for the C4 L6 ZHA router on natural text. (a) Within each KV group and level, reference queries are independently ranked by selection frequency. Each point shows the level's share of selections at that rank, averaged over layers and KV groups. Higher levels concentrate selections among fewer reference queries, while lower levels use them more uniformly. (b) Spearman correlation between levels' selection rates. Higher levels have strongly aligned reference-query rankings. Agreement generally weakens as levels become further separated, with the chunk level showing the greatest divergence from the upper levels.

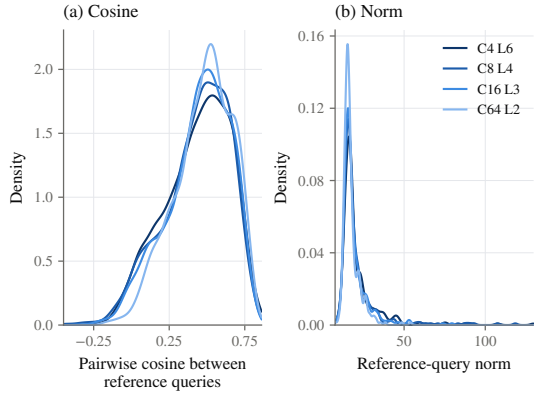


Figure 10: (a) Cosine similarity distribution of every pair of reference queries within a KV group; (b) norm distribution of every reference query. Reference queries within a KV group become slightly more similar and their norms more concentrated as the tree gets shallower and its chunks larger.

VII. Discussion

In this paper, we have introduced ZHA, a tree-structured hierarchical sparse attention method that can achieve asymptotically subquadratic attention scaling, subject to specific constraints on the tree geometry and the top-p value. Moreover, we have shown that ZHA can be applied to existing base models in a short distillation phase with only a few billion tokens, and it maintains performance on challenging long-context reasoning, agentic coding, and retrieval tasks.

While empirically ZHA appears to perform very well for contexts up to 256k, performance does degrade at increasingly higher levels of sparsity, and thus ultimately at longer context lengths. Here, we present perhaps the simplest possible version of the tree-structured indexer idea, which only learns reference queries in a separate independent KL-distillation phase. There are potentially many improvements that could be made to the core algorithm or training recipe to improve its performance. For instance, chunk boundaries could be dynamically learned instead of set arbitrarily; we could introduce a recency bias into the tree so that it becomes more detailed closer to the current token, rather than uniformly distributing mass over the sequence; and we could finetune the base model parameters on long-context training sets with a highly aggressive ZHA geometry to let the rest of the model learn to compensate for the degradation caused by the sparsity. We could also augment the tree with compressed value summaries, allowing nodes pruned during routing to contribute coarse attention outputs. This would preserve approximate visibility over the full KV cache span, combining token-level attention over selected regions with summary-level contributions from pruned subtrees.

Parameter scaling. Our method is surprisingly simple, requiring only a single KL-warmup phase in which we train a set of reference queries (these are the only learnable parameters we add) to match the attention mass induced by regular dense attention. Although we initially tried using ideas from Hu et al. [18] such as a reference-query LoRA and an explicit factorization of the attention probabilities across different levels of the tree, we found that these methods added complexity and did not improve performance in our preliminary ablations.

One interesting trend we noted during our initial development of ZHA, is that the performance of ZHA improves with model scale. We found that small models under 1B parameters often struggled significantly with RULER and other retrieval tasks during conversion to ZHA because the original attention scores we were distilling from were very noisy themselves. Our success with distillation improved dramatically when moving to larger models in the 4B scale and up. This makes us optimistic that ZHA will continue to improve with further increases to model scale.

Positional encoding. Another interesting question concerns the impact of different positional encoding schemes on the effectiveness of ZHA. The primary experiments we ran were on the Qwen model series which uses p-RoPE, as were our experiments on the structure of naturalistic data in Figure 1. However, we find that RoPE induces a very particular structure into the attention scores over long contexts, where most heads represent a strong recency bias while a few heads appear to primarily attend to the lowest RoPE frequencies and are sensitive much more uniformly across the context. NoPE attention, by contrast, can induce a more diffuse attention structure, and thus may extrapolate or give different asymptotic scalings for ZHA than RoPE models. In Appendix B-D, we also tested the hybrid-NoPE Nemotron3 models and found that ZHA was able to be trained and maintain evaluation performance for these models too. However, the long-context extrapolation capabilities of ZHA with NoPE remain uncertain.

Asymptotic length scaling. The most important limitation in our results is that, although we have extrapolated performance out to millions of tokens of context and beyond, we have not empirically validated the evaluation performance of ZHA at these context lengths. The primary limitation is finding models with sufficiently long contexts that are small enough to adapt without substantial computational expense. In future work, we plan to scale up the method to frontier open-source models and determine whether our method continues to scale to 1 million context and beyond.

Of course, this problem of evaluating performance at contexts much beyond 1 million tokens is not unique to ZHA, and is in fact shared by all attention-based models. Though this in part derives from data limitations, it also points to the inspiration for sparse attention methods: attention's quadratic scaling makes long-context training and inference prohibitively expensive. Beyond this computational bottleneck, however, it is still unclear whether dense attention itself is even performant at longer sequence lengths. Prior work has in fact found that softmax can be "dispersive", in the sense that it upweights noise within the preceding context, leading to increasingly uniform attention scores that eventually prevent effective retrieval [42].

Therefore, an argument may be made that sparsity is not only necessary for computational efficiency, but also to continue to scale language modelling to ultra-long contexts. It may be the case that sparsity's pruning effectively "denoises" the past context, sidestepping the dispersiveness of dense attention and uncovering the truly relevant history to be retrieved. Indeed, our finding that top-p grows like a sublinear power law is highly suggestive that the subset of information that is truly relevant for predicting the next token has fundamentally different scaling properties than dense retrieval of the entire context.

A core assumption of our work is that this top-p set contains sufficient information to enable effective language modeling. ZHA's empirical success in part validates this assumption; however, it is admittedly unclear whether this will remain the case indefinitely. Similarly, it is unclear whether top-p, and the underlying predictive information that we suppose it picks up, grows sublinearly to indefinite context lengths.

Supposing that retrieving the true top-p set is "all you need", the remaining source of error is properly routing to that set. Due to asymptotic divergence of different power-law curves, precisely matching the true scaling of the top-p set is a surprisingly subtle point. In short, it turns out that our method of applying top-p iteratively down a growing tree requires carefully tuning the per-level p, and failing to do so results in two possibilities. First, fixing $p < 1$ as we do currently, the "cumulative" $p_{\text{eff}} = p^L \to 0$ as depth $L \to \infty$. In this case, the retrieval state asymptotes towards some constant size and performance necessarily degrades. Alternatively, if we rescaled per-level p to $p^{1/L}$ to counteract this effect, we could then observe the opposite pathology where p tends to 1 as depth increases, leading to all of the nodes at the higher levels being selected and the upper levels of the tree effectively becoming redundant. If this occurs then we return to quadratic scaling and hence the diffusive pathology of fully dense attention.

To remain subquadratic in the limit while also maintaining performance, then, we need some method to scale p dynamically with depth such that the effective p at the leaf nodes decreases, but at a slow enough rate that it tracks the decrease in predictive information in the sequence with increasing context length, similar to that we found in Figure 1. This was roughly a power-law scaling. However, this depends not on the architecture, but on the intrinsic distribution of the data and thus is deeply dependent upon the statistical structure of ultra-long context data. Our analysis suggests that naturalistic data does exhibit an amenable scaling, but we only investigated data up to 256K context, and thus extrapolating this to hundreds of millions or billions of tokens could fail. Whether or not naturalistic text data continues to exhibit power-law self-similar structure at these ultra-long scales thus remains an important avenue for future work.

References

[1] Joshua Ainslie, James Lee-Thorp, Michiel De Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. GQA: Training generalized multi-query transformer models from multi-head checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 4895–4901, 2023.

[2] Yushi Bai, Shangqing Tu, Jiajie Zhang, Hao Peng, Xiaozhi Wang, Xin Lv, Shulin Cao, Jiazheng Xu, Lei Hou, Yuxiao Dong, et al. LongBench v2: Towards deeper understanding and reasoning on realistic long-context multitasks. *arXiv preprint arXiv:2412.15204*, 2024.

[3] 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. *arXiv preprint arXiv:2603.12201*, 2026.

[4] Zefan Cai, Yichi Zhang, Bofei Gao, Yuliang Liu, Yucheng Li, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Junjie Hu, et al. PyramidKV: Dynamic KV cache

compression based on pyramidal information funneling. *arXiv preprint arXiv:2406.02069*, 2024.

[5] Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, et al. Introducing SWE-bench Verified, 2024. URL https://openai.com/index/introducing-swe-bench-verified/.

[6] Tri Dao and Albert Gu. Transformers are SSMs: Generalized models and efficient algorithms through structured state space duality, 2024. URL https://arxiv.org/abs/2405.21060.

[7] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

[8] DeepSeek-AI. DeepSeek-V4: Towards highly efficient million-token context intelligence, 2026. URL https://arxiv.org/abs/2606.19348.

[9] DeepSeek-AI et al. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[10] Yuan Feng, Junlin Lv, Yukun Cao, Xike Xie, and S Kevin Zhou. Ada-KV: Optimizing KV cache eviction by adaptive budget allocation for efficient LLM inference. *Advances in Neural Information Processing Systems*, 38: 113152–113188, 2026.

[11] Tomas Figliolia, Nicholas Alonso, Rishi Iyer, Quentin Anthony, and Beren Millidge. Compressed convolutional attention: Efficient attention in a compressed latent space. *arXiv preprint arXiv:2510.04476*, 2025.

[12] Zichuan Fu, Wentao Song, Yejing Wang, Xian Wu, Yefeng Zheng, Yingying Zhang, Derong Xu, Xuetao Wei, Tong Xu, and Xiangyu Zhao. Sliding window attention training for efficient large language models, 2025. URL https://arxiv.org/abs/2502.18845.

[13] Paolo Glorioso, Quentin Anthony, Yury Tokpanov, Anna Golubeva, Vasudev Shyam, James Whittington, Jonathan Pilault, and Beren Millidge. The Zamba2 suite: Technical report. *arXiv preprint arXiv:2411.15242*, 2024.

[14] Paolo Glorioso, Quentin Anthony, Yury Tokpanov, James Whittington, Jonathan Pilault, Adam Ibrahim, and Beren Millidge. Zamba: A compact 7B SSM hybrid model. *arXiv preprint arXiv:2405.16712*, 2024.

[15] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces, 2024. URL https://arxiv.org/abs/2312.00752.

[16] Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. RULER: What's the real context size of your long-context language models? *arXiv preprint arXiv:2404.06654*, 2024.

[17] Shengding Hu, Yuge Tu, Xu Han, Chaoqun He, Ganqu Cui, Xiang Long, Zhi Zheng, Yewei Fang, Yuxiang Huang, Weilin Zhao, et al. MiniCPM: Unveiling the potential of small language models with scalable training strategies. *arXiv preprint arXiv:2404.06395*, 2024. URL https://arxiv.org/abs/2404.06395.

[18] Xiang Hu, Xinyu Wei, Hao Gu, Minshen Zhang, Tian Liang, Huayang Li, Lei Zhu, Yan Wang, Sirui Han, Yushi Bai, et al. Hierarchical sparse attention done right: Toward infinite context modeling. *arXiv preprint arXiv:2607.02980*, 2026.

[19] Yuxiang Huang, Nuno MT Gonçalves, Federico Alvetreti, Lei Li, Xu Han, Edoardo M Ponti, André FT Martins, and Marcos V Treviso. DashAttention: Differentiable and adaptive sparse hierarchical attention. *arXiv preprint arXiv:2605.18753*, 2026.

[20] Sukjun Hwang, Brandon Wang, and Albert Gu. Dynamic chunking for end-to-end hierarchical sequence modeling. *arXiv preprint arXiv:2507.07955*, 2025.

[21] Samy Jelassi, David Brandfonbrener, Sham M. Kakade, and Eran Malach. Repeat after me: Transformers are better than state space models at copying. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235, pp. 21502–21521, 2024. URL https://proceedings.mlr.press/v235/jelassi24a.html.

[22] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

[23] Jang-Hyun Kim, Dongyoon Han, and Sangdoo Yun. Fast KVzip: Efficient and accurate LLM inference with gated KV eviction, 2026. URL https://arxiv.org/abs/2601.17668.

[24] Aakash Lahoti, Kevin Y. Li, Berlin Chen, Caitlin Wang, Aviv Bick, J. Zico Kolter, Tri Dao, and Albert Gu. Mamba-3: Improved sequence modeling using state space principles, 2026. URL https://arxiv.org/abs/2603.15569.

[25] Xunhao Lai, Weiqi Xu, Yufeng Yang, Qiaorui Chen, Yang Xu, Lunbin Zeng, Xiaolong Li, Haohai Sun, Haichao Zhu, Vito Zhang, et al. MiniMax sparse attention. *arXiv preprint arXiv:2606.13392*, 2026.

[26] Opher Lieber, Barak Lenz, Hofit Bata, Gal Cohen, Jhonathan Osin, Itay Dalmedigos, Erez Safahi, Shaked Meirom, Yonatan Belinkov, Shai Shalev-Shwartz, et al. Jamba: A hybrid transformer-Mamba language model. *arXiv preprint arXiv:2403.19887*, 2024.

[27] 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. In *Advances in Neural Information Processing Systems*, volume 38, 2025. URL https://arxiv.org/abs/2502.02770.

[28] Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai

Dai, Daya Guo, et al. DeepSeek-V2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*, 2024.

[29] Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, et al. DeepSeek-V3.2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025.

[30] Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. Scissorhands: Exploiting the persistence of importance hypothesis for LLM KV cache compression at test time. *Advances in Neural Information Processing Systems*, 36:52342–52364, 2023.

[31] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019. URL https://arxiv.org/abs/1711.05101.

[32] MathArena. AIME 2026 dataset, 2026. URL https://huggingface.co/datasets/MathArena/aime_2026/tree/d2de22f3c656b4f56cf8981212186377d1e23bc3.

[33] NVIDIA. NVIDIA Nemotron 3.5 Super and Lightning, 2026. URL https://research.nvidia.com/labs/nemotron/Nemotron-3.5-Super-and-Lightning/.

[34] Zihan Qiu, Zekun Wang, Bo Zheng, Zeyu Huang, Kaiyue Wen, Songlin Yang, Rui Men, Le Yu, Fei Huang, Suozhi Huang, et al. Gated attention for large language models: Non-linearity, sparsity, and attention-sink-free. *Advances in Neural Information Processing Systems*, 38:100092–100118, 2026.

[35] Qwen Team. Qwen3.6-35B-A3B model card, 2026. URL https://huggingface.co/Qwen/Qwen3.6-35B-A3B.

[36] 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. *arXiv preprint arXiv:2311.12022*, 2023.

[37] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. RoFormer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568: 127063, 2024.

[38] Bohao Tang, Zhen Qin, Yuqi Pan, Zheng Li, and Pengfei Liu. Block sparse attention with log-linear complexity. *arXiv preprint arXiv:2609.31093*, 2026.

[39] Kimi Team, Yu Zhang, Zongyu Lin, Xingcheng Yao, Jiaxi Hu, Fanqing Meng, Chengyin Liu, Xin Men, Songlin Yang, Zhiyuan Li, et al. Kimi Linear: An expressive, efficient attention architecture. *arXiv preprint arXiv:2510.26692*, 2025.

[40] Terminal-Bench Team. Terminal-Bench 2.1, 2026. URL https://www.tbench.ai/news/terminal-bench-2-1.

[41] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, 2017. URL https://arxiv.org/abs/1706.03762.

[42] Petar Veličković, Christos Perivolaropoulos, Federico Barbero, and Razvan Pascanu. Softmax is not enough (for sharp size generalisation). *arXiv preprint arXiv:2410.01104*, 2024.

[43] Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyan Jiang, et al. MMLU-Pro: A more robust and challenging multi-task language understanding benchmark. *arXiv preprint arXiv:2406.01574*, 2024.

[44] Robert Washbourne, Rishi Iyer, Tomas Figliolia, Henry Zheng, Ryan Lorig-Roach, Sungyeon Yang, Pritish Yuvraj, Quentin Anthony, Yury Tokpanov, Xiao Yang, et al. ZAYA1-8B technical report. *arXiv preprint arXiv:2605.05365*, 2026.

[45] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52 (4):65–76, 2009. doi: 10.1145/1498765.1498785.

[46] 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.

[47] Wen Zan, Jiaqi Zhang, Jianchao Tan, Hong Liu, Cunguang Wang, Xiang Li, Duyue Ma, Guanyu Wu, Yifan Lu, Fengcun Li, et al. LongCat sparse attention: Taming the lightning via streaming-aware hierarchical cross-layer indexing. *arXiv preprint arXiv:2608.01662*, 2026.

[48] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. H2O: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems*, 36:34661–34710, 2023.

Appendix

Appendix A: Top-p Scaling by Layer and Head 16
A-A Dense Top-p Sets by Layer and Head . . . 16
A-B Affine Versus Power-Law Fits on Rollouts . . . 17
A-C Nemotron Top-p Scaling . . . 18

Appendix B: Training and Evaluation Details 20
B-A Training Setup and Compute . . . 20
B-B Evaluation Protocols . . . 21
B-C Qwen3.6-35B-A3B . . . 21
B-D Nemotron-3.5-Lightning-30B-A3B . . . 23

Appendix C: Geometry, Retrieval Accuracy, and Efficiency 24
C-A Geometry Sweep . . . 24
C-B Effect of Geometry on Per-Task RULER Accuracy . . . 24
C-C Geometry–Performance Pareto Frontier . . . 26

Appendix D: Roofline Model 27
D-A Dense . . . 27
D-B ZHA . . . 30
D-C MSA . . . 33

Appendix E: Sparsity Scaling Across Routing Thresholds 35
E-A Estimating Per-Level Growth . . . 35
E-B Modeling Levels Within a Tree . . . 35
E-C Adding Tree Depth . . . 35
E-D Robustness of the Depth Trend . . . 37
E-E From Selection Counts to Latency . . . 37

Appendix F: Retrieval Traversal Examples 38

Appendix G: Reference-Query Agreement Across Tree Levels 39
G-A Individual layer and KV-group examples . . . 40

Appendix H: Reference-Query and Reference-Key Similarity 41

APPENDIX A
TOP-p SCALING BY LAYER AND HEAD

A. Dense Top-p Sets by Layer and Head

In Section II, we average dense top-p set sizes over layers and query heads. Figure 11 shows all 10 full-attention layers and 16 query heads of Qwen3.6-35B-A3B separately, averaging only across the six books. Common logarithmic axes allow comparison with the linear $p = 1$ reference, which retains all keys. Sublinear trends appear throughout the full-attention stack, with substantial variation in set size and growth rate. The aggregate trend is thus not confined to a few sparse heads, but does not imply a common exponent.

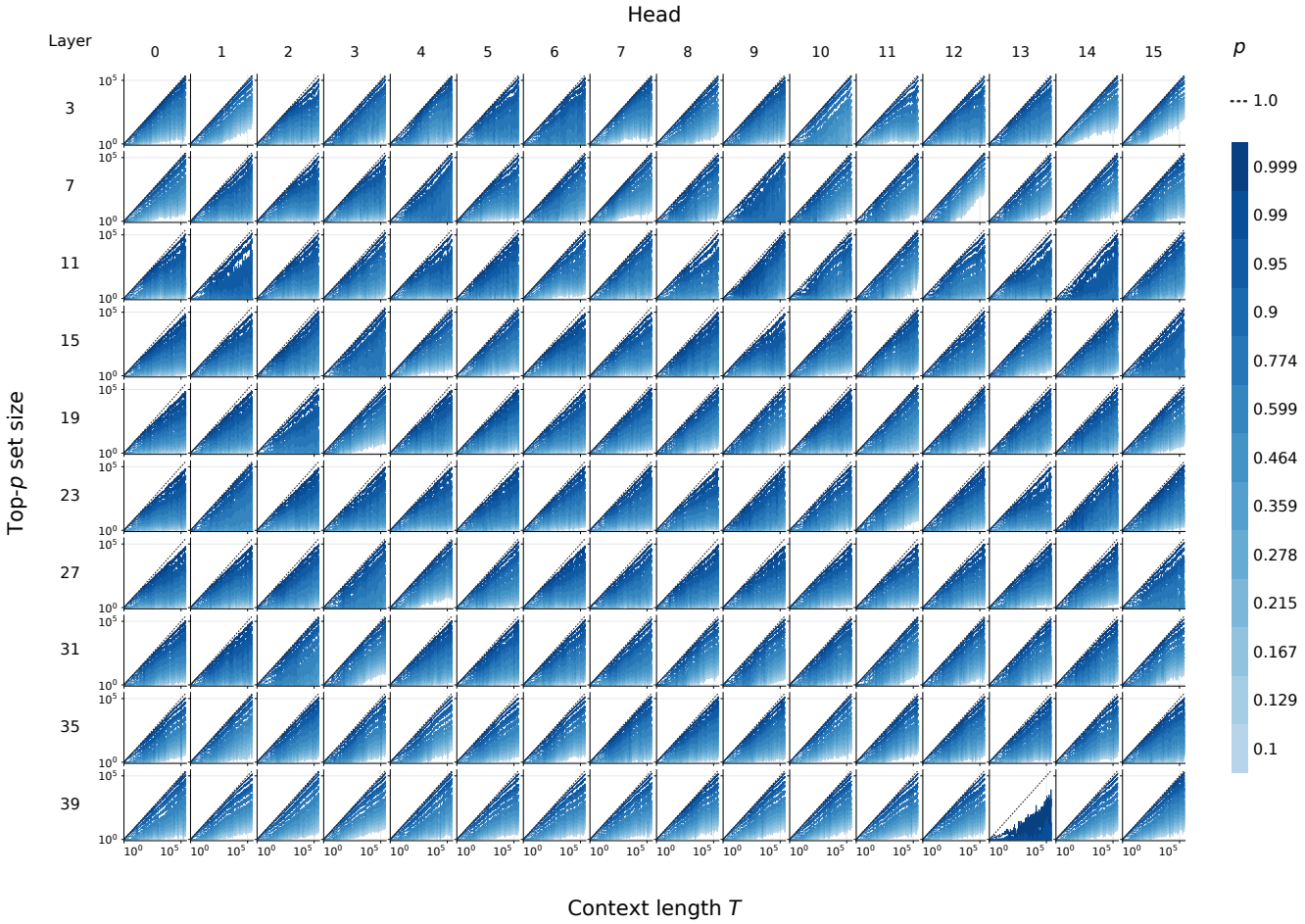


Figure 11: Dense top-p set size for individual query heads in Qwen3.6-35B-A3B, using the same six-book cohort and thresholds as Figure 1. Rows identify full-attention layers and columns identify query heads, with zero-based indices. Each panel averages only across books. All panels share logarithmic axis limits; colors denote p and the dotted black line gives the $p = 1$ reference.

B. Affine Versus Power-Law Fits on Rollouts

Table I provides a complementary comparison using ZHA leaf-level selection counts, not dense top-p set sizes. Each count unions selected leaf nodes across the eight query heads sharing a KV head and excludes mandatory sink and local tokens. Here, C is the checkpoint's base KV chunk size, Q is the query-tile width, and p is the routing mass threshold.

We fit each rollout–layer–KV-head trace separately, averaging tile counts in 256-query bins. Both models minimize squared log-count residuals on the same observations. Raw and cumulative-maximum targets are fitted separately, with running maxima computed within each rollout before bin averaging. The Raw and Cum-max columns each give the affine win rate among 9,600 paired fits.

The preferred model depends on the configuration. At $p = 0.9$ and $(C, Q) = (8, 16)$, the affine win rate is 3.76% for raw counts and 5.36% for cumulative-maximum counts. For $(C, Q) = (256, 256)$ at the same threshold, the corresponding rates are 99.97% and 100%. Power laws are therefore preferred in many configurations, but not uniformly across checkpoint geometries and query-tile widths.

Table I: Percentage of individual rollout–layer–head fits for which the affine model beats the power law, $\rho = (1 - R^2_{\text{lin}})/(1 - R^2_{\text{pow}}) < 1$, evaluated in log-$y$ space at the leaf level. Raw and Cum-max denote raw and cumulative-maximum targets. Each percentage uses 9,600 paired fits (30 problems $\times$ 16 rollouts $\times$ 10 layers $\times$ 2 KV heads), without averaging across rollouts before fitting.

(a) $Q = 1$

C	p	Raw (%)	Cum-max (%)
4	0.70	56.43	39.18
4	0.80	11.95	8.59
4	0.90	11.27	6.77
4	0.95	11.94	7.99
8	0.70	12.50	8.67
8	0.80	9.11	4.61
8	0.90	10.38	6.82
8	0.95	10.40	7.35
16	0.70	5.86	2.94
16	0.80	7.34	5.00
16	0.90	11.52	7.28
16	0.95	9.57	8.13
64	0.70	1.64	0.72
64	0.80	4.41	1.81
64	0.90	7.49	4.07
64	0.95	8.09	15.56
256	0.80	6.09	41.63
256	0.90	12.18	83.65
256	0.95	29.55	97.17

(b) $Q = 16$

C	p	Raw (%)	Cum-max (%)
4	0.70	49.83	30.26
4	0.80	5.89	2.30
4	0.90	4.42	3.25
4	0.95	4.36	6.69
8	0.70	6.89	2.67
8	0.80	2.95	2.17
8	0.90	3.76	5.36
8	0.95	3.73	8.79
16	0.70	1.80	1.43
16	0.80	2.26	3.57
16	0.90	3.25	7.08
16	0.95	3.76	17.36
64	0.70	0.47	1.96
64	0.80	1.69	16.67
64	0.90	10.81	53.89
64	0.95	24.41	83.21
256	0.80	46.68	96.72
256	0.90	74.31	99.84
256	0.95	90.93	100.00

(c) $Q = 32$

C	p	Raw (%)	Cum-max (%)
4	0.70	47.78	27.04
4	0.80	4.15	1.91
4	0.90	3.15	3.88
4	0.95	3.39	8.42
8	0.70	5.59	2.39
8	0.80	1.82	2.49
8	0.90	2.66	6.55
8	0.95	3.21	10.94
16	0.70	1.41	1.60
16	0.80	1.84	4.13
16	0.90	2.32	10.47
16	0.95	7.54	22.64
64	0.70	0.55	8.65
64	0.80	4.31	35.40
64	0.90	22.18	74.70
64	0.95	51.15	93.01
256	0.80	67.74	98.88
256	0.90	89.13	99.96
256	0.95	96.57	100.00

(d) $Q = 64$

C	p	Raw (%)	Cum-max (%)
4	0.70	44.09	22.85
4	0.80	2.85	1.88
4	0.90	2.32	4.98
4	0.95	2.83	10.80
8	0.70	4.32	2.22
8	0.80	1.26	2.97
8	0.90	2.26	8.55
8	0.95	3.24	14.33
16	0.70	1.17	1.90
16	0.80	1.47	5.08
16	0.90	3.08	14.16
16	0.95	12.45	30.91
64	0.70	3.11	29.47
64	0.80	19.73	62.84
64	0.90	53.82	90.22
64	0.95	78.69	98.55
256	0.80	85.46	99.73
256	0.90	96.49	100.00
256	0.95	99.34	100.00

(e) $Q = 128$

C	p	Raw (%)	Cum-max (%)
4	0.70	39.41	18.83
4	0.80	2.06	2.00
4	0.90	2.11	6.66
4	0.95	2.72	11.85
8	0.70	3.29	2.23
8	0.80	1.05	3.83
8	0.90	2.32	9.64
8	0.95	7.48	17.28
16	0.70	1.05	2.58
16	0.80	1.40	6.04
16	0.90	8.27	19.96
16	0.95	24.26	45.98
64	0.70	13.19	43.44
64	0.80	40.36	75.22
64	0.90	73.65	94.26
64	0.95	90.56	99.28
256	0.80	95.44	99.98
256	0.90	99.39	100.00
256	0.95	99.95	100.00

(f) $Q = 256$

C	p	Raw (%)	Cum-max (%)
4	0.70	32.93	15.39
4	0.80	1.54	2.04
4	0.90	2.26	8.24
4	0.95	3.94	11.57
8	0.70	2.46	2.59
8	0.80	1.17	4.68
8	0.90	4.08	11.08
8	0.95	15.96	22.85
16	0.70	1.27	3.31
16	0.80	2.70	7.78
16	0.90	20.52	29.79
16	0.95	49.83	62.56
64	0.70	33.61	58.35
64	0.80	63.20	83.81
64	0.90	87.25	96.84
64	0.95	96.89	99.64
256	0.80	99.42	100.00
256	0.90	99.97	100.00
256	0.95	100.00	100.00

C. Nemotron Top-p Scaling

Figure 12 repeats the dense analysis for Nemotron-3.5-Lightning-30B-A3B on the same six books. Here, we find that while top-p set sizes again grow sublinearly, the stronger log-log curvature and late-context fluctuations make a single power law less descriptive. Sublinear growth thus need not follow a clean power law. Although the models differ in positional encoding, this comparison does not isolate it as the cause.

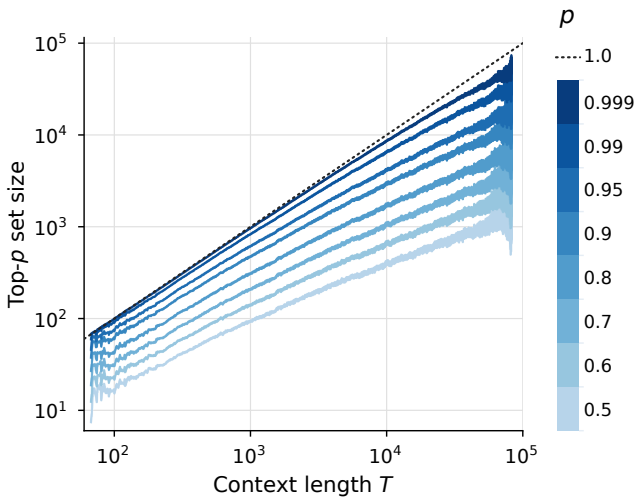


Figure 12: Dense top-p set size for Nemotron-3.5-Lightning-30B-A3B using the same six-book cohort as Figure 1.

APPENDIX B
TRAINING AND EVALUATION DETAILS

A. Training Setup and Compute

Table II summarizes the training regime for the four ZHA routers and the MSA indexer used in our evaluations. Qwen denotes Qwen3.6-35B-A3B and Nemotron denotes Nemotron-3.5-Lightning-30B-A3B.

	Qwen ZHA		Nemotron ZHA		Qwen MSA
	Deep	Wide	Deep	Wide	Index
Trained module	ref. queries	ref. queries	ref. queries	ref. queries	index proj. + norms
Geometry	C4 F4 L6	C256 L1	C4 F4 L6	C256 L1	B128 top-16
Sparse-attention layers	10 / 40	10 / 40	6 / 52	6 / 52	10 / 40
Trainable params	81,920	81,920	24,576	24,576	7.87M
Frozen backbone params	34.66B	34.66B	31.58B	31.58B	34.66B
Sequence length	32,768	32,768	32,768	32,768	32,768
Batch, rows (tokens)	64 (2.1M)	64 (2.1M)	64 (2.1M)	64 (2.1M)	320 (10.5M)
Steps	1,900	1,900	1,900	1,900	3,815
Training tokens	3.98B	3.98B	3.98B	3.98B	40.0B
AdamW ϵ	1×10^{-15}	1×10^{-15}	1×10^{-15}	1×10^{-15}	1×10^{-8}
Peak learning rate	3.2×10^{-2}	3.2×10^{-2}	3.2×10^{-2}	3.2×10^{-2}	3×10^{-4}
LR schedule	WSD	WSD	WSD	WSD	cosine
Weight decay	0	0	0	0	0.1 / 0
Gradient clipping (norm)	1.0	1.0	1.0	1.0	1.0
Final KL (ZHA: level sum)	$1.172^{\dagger}$	0.242	0.850	0.179	0.120

Table II: Training setup and compute for the evaluated ZHA routers and MSA index. Backbones are frozen; ZHA trains only reference queries. Deep and wide ZHA use $R = 16$.

Training recipe. ZHA is trained by minimizing KL to dense attention over tree nodes at each level. This is a local objective applied to each attention block independently. By default, unless otherwise specified, ZHA shares 16 reference queries per KV group. The same reference queries are used for all levels of the tree. We train at 32,768 context using document-masked attention. Unless otherwise specified, the backbone is in BF16 and the routing parameters are in FP32. We train for 1,900 steps, totaling approximately 4 billion tokens, and we use the final checkpoint of each training run for evaluation.

For MSA, we use 128-token blocks and select the top 16 blocks per query. We perform warmup by minimizing KL over the key blocks.

Optimization. We train with the AdamW optimizer [31] using $\beta = (0.9, 0.95)$ and gradient-norm clipping at 1.0. ZHA's reference-query learning rate is 3.2×10^{-2}, with no weight decay. We use the warmup–stable–decay (WSD) learning rate schedule [17] with 20 warmup steps and linear decay over the final 20% of steps to 1% of the peak learning rate. MSA uses a peak learning rate of 3×10^{-4}, 200 warmup steps, and cosine decay to 10% of peak; weight decay is 0.1 on projections and zero on norm gains. The MSA paper does not report its optimizer settings. We chose ours from the best of three 40B-token pilot runs that varied the learning rate, batch size and schedule. Nemotron's stored FP32 MoE correction biases are frozen and excluded from the trainable parameter count.

Data and compute accounting. For our Qwen and Nemotron training runs, we sample approximately 4B tokens (13.8M documents) from a 100B-token mid-training mixture we used for the ZAYA1-8B model [44] for each model. We retokenized the data with the appropriate tokenizer. We trained for a single epoch over the corpus. We chose mid-training data since we believed it was likely similar to the data distribution used for final mid-training and SFT stages of the Qwen and Nemotron models prior to RL.

Training KL. When we report the final KL, we are averaging the last 100 steps. We report ZHA's total sums over tree levels; Table III gives the individual values, with level 0 denoting the raw KV cache chunks. The MSA value is its block-level objective. These differently aggregated objectives should not be compared directly across depths or methods.

Router	L0	L1	L2	L3	L4	L5
Qwen deep	0.021	0.097	0.180	0.253	0.302	0.319
Qwen wide	0.242	–	–	–	–	–
Nemotron deep	0.016	0.069	0.137	0.192	0.217	0.221
Nemotron wide	0.179	–	–	–	–	–

Table III: Final per-level ZHA training KL, averaged over logged points in the last 100 steps.

B. Evaluation Protocols

Table IV: Evaluation datasets, scoring metrics, and inference protocols.

Benchmark	**Evaluated subset**	**Reported metric**	**Generation budget**	**Decoding / harness**
AIME 2026 [32][1]	30 problems, 16 samples each	avg@16 (%)	81,920 output tokens	Thinking; temperature 1.0, top-p 0.95, top-k 20, presence penalty 1.5
SWE-bench Verified [5, 22]	500 tasks, 1 run each	Resolved tasks (%)	32,768 tokens/turn; 204,800-token context; 250 steps	mini-swe-agent 2.4.6; bash text actions; thinking retained across turns; temperature 1.0, top-p 0.95, top-k 20
Terminal-Bench 2.1 [40][2]	All 89 tasks, 5 attempts each	Mean per-task success rate (%)	81,920 tokens/turn; 262,144-token context; 3 h per trial	Harbor 0.24 / Terminus-2; thinking retained across turns; temperature 1.0, top-p 0.95, top-k 20
MMLU-Pro [43]	Full test split: 12,032 questions; zero-shot CoT; 1 sample each	Accuracy (%)	81,920 output tokens	Thinking; temperature 1.0, top-p 0.95, top-k 20
GPQA Diamond [36]	All 198 questions, 8 samples each	avg@8 (%)	81,920 output tokens	Thinking; temperature 1.0, top-p 0.95, top-k 20
RULER-13 [16]	13 tasks $\times$ 100 examples at 256K; 64K and 128K in appendix	Mean task score (%)	Official per-task caps: 30–128 tokens	Greedy; thinking disabled
LongBench v2 [2][3]	205 questions with original contexts $\geq$128K tokens; 1 sample each	Accuracy using official answer keys (%)	32,768 reasoning + 4,096 answer tokens; prompts capped at 225,280 tokens	Native thinking; temperature 1.0, top-p 0.95, top-k 20, presence penalty 1.5

[1]Uses the pinned MathArena dataset revision containing the problem-10 statement error.
[2]Uses the Qwen3.6 model-card protocol: flat 3 h per trial and 32 CPUs / 48 GB per container.
[3]Prompts exceeding the cap retain their first and last halves.

Evaluation protocols. AIME 2026 reports avg@16 over 30 problems, grading the final boxed answer after reasoning. SWE-bench Verified reports resolved tasks out of 500. Terminal-Bench 2.1 averages five attempts per task over 89 tasks with a three-hour agent limit. MMLU-Pro uses 12,032 questions, and GPQA Diamond uses 198 questions with eight samples each. RULER-13 averages 13 official task scores at 256K tokens, with 100 examples per task. LongBench v2 uses 205 questions with contexts $\geq$128K tokens, a 32K-token reasoning budget, and a 4K-token answer budget.

C. Qwen3.6-35B-A3B

Figure 13 extends the main-text dense vs ZHA comparisons at $p = 0.95$ to routing thresholds $p \in \{0.95, 0.9, 0.8\}$. Each threshold is applied uniformly at each tree level. The figure reports both benchmark scores and, where available, the share of keys attended during decoding, showing the tradeoff between accuracy and keys attended. Furthermore it demonstrates how sparsity compounds with depth: at the same routing threshold p, the deep router achieves greater sparsity than the wide router. We also compare the sparsity of both ZHA geometries with MSA in Figure 14. For both figures; blue bars denote the deep router and green bars the wide router; darker shades indicate higher thresholds.

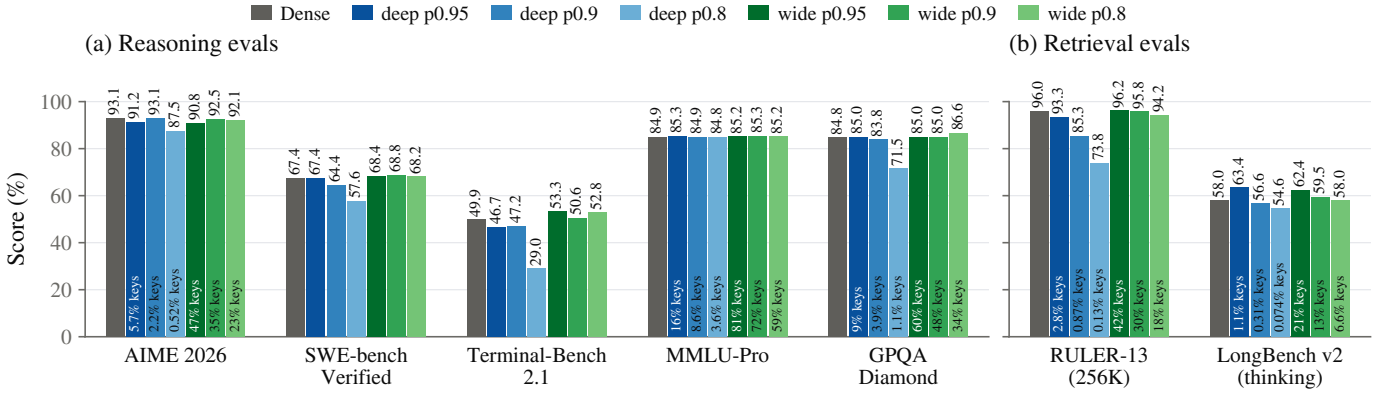


Figure 13: Qwen3.6-35B-A3B scores and decode key shares across routing thresholds for deep (C4 L6, blue) and wide (C256 L1, green) ZHA, with $R = 16$. Darker shades indicate higher p. (a) Reasoning; (b) retrieval. Labels above bars show the benchmark scores and labels inside bars give key shares where available.

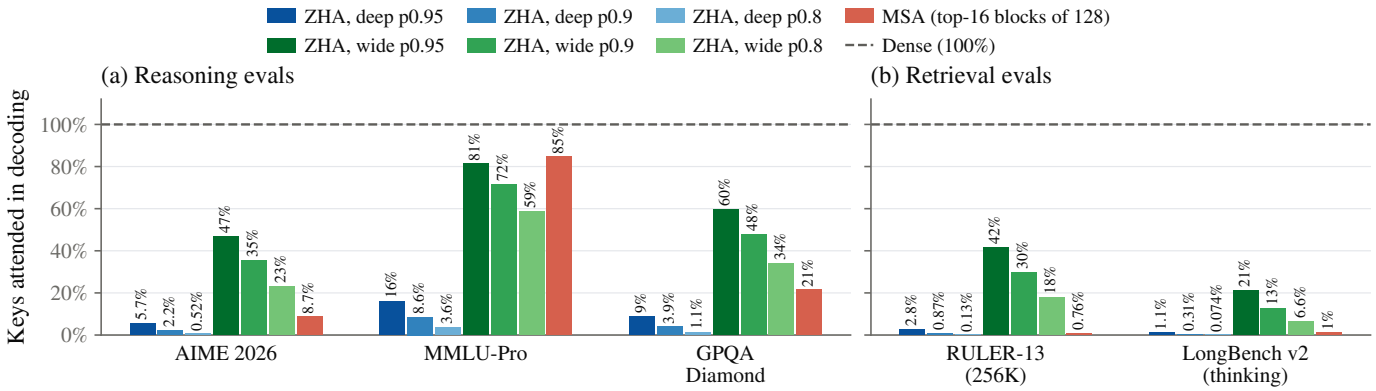


Figure 14: Qwen3.6-35B-A3B decode key shares for deep (C4 L6, blue) and wide (C256 L1, green) ZHA ($R = 16$) at $p \in \{0.95, 0.9, 0.8\}$, compared with MSA (top-16 blocks of 128 tokens). (a) Reasoning; (b) retrieval. Darker shades indicate higher p; the dashed line marks dense attention (100%).

Small-reference router.

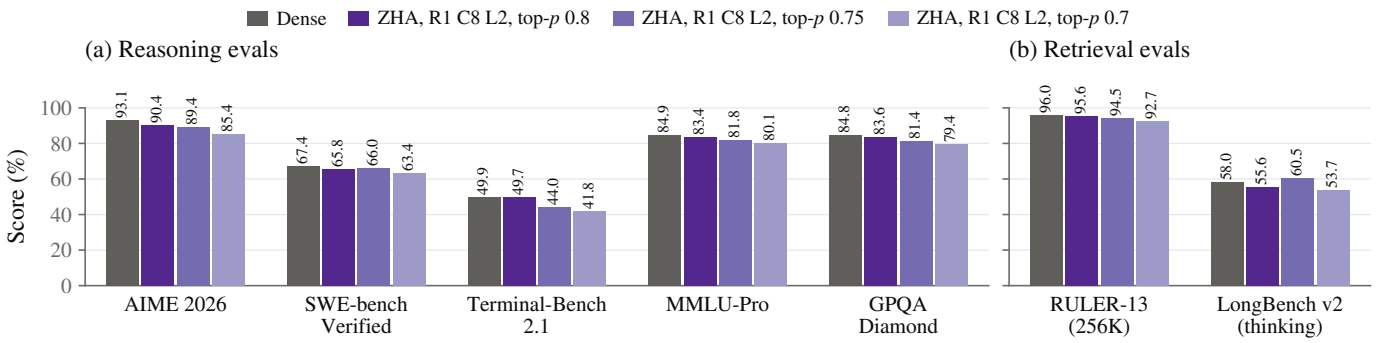


Figure 15: Dense attention vs. a ZHA router with one reference query per KV group, 8-token chunks, fan 8 and two levels (R1 C8 L2) on Qwen3.6-35B-A3B, with uniform routing top-p 0.8, 0.75 and 0.7. (a) Reasoning (b) Retrieval.

Router precision.

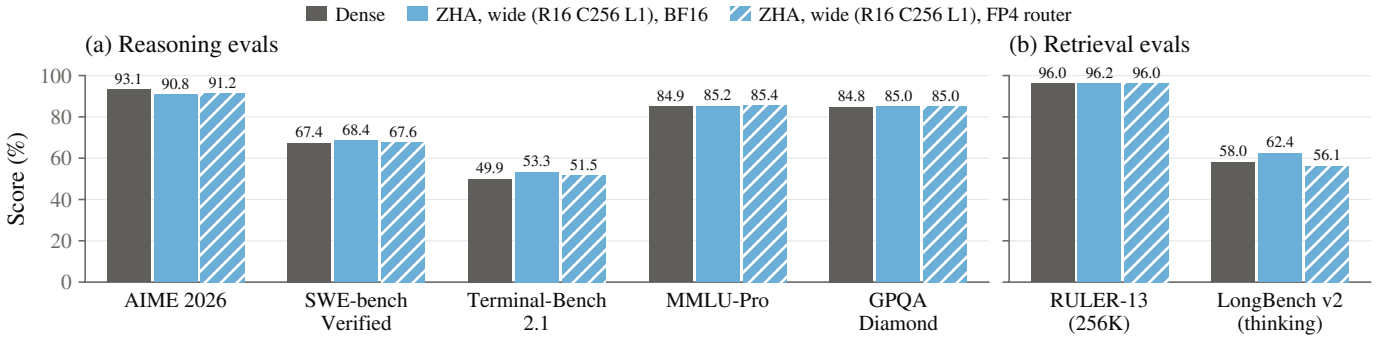


Figure 16: Dense attention vs. the wide ZHA router (16 reference queries per KV group, 256-token chunks, one level; top-p 0.95) on Qwen3.6-35B-A3B, with the router in BF16 and in NVFP4 (router operands, reference-key cache and scoring in FP4; backbone, KV cache and the sparse attention in BF 16). (a) Reasoning (b) Retrieval

D. Nemotron-3.5-Lightning-30B-A3B

On Nemotron-3.5-Lightning-30B-A3B, which has no explicit positional encodings, ZHA retains performance close to the dense baseline on the reasoning and coding benchmarks. On RULER-13 however at p=0.95, the deep and wide geometry scores 25.3 and 10.2 percentage points below the dense baseline, respectively. Figure 17 shows results across routing thresholds.

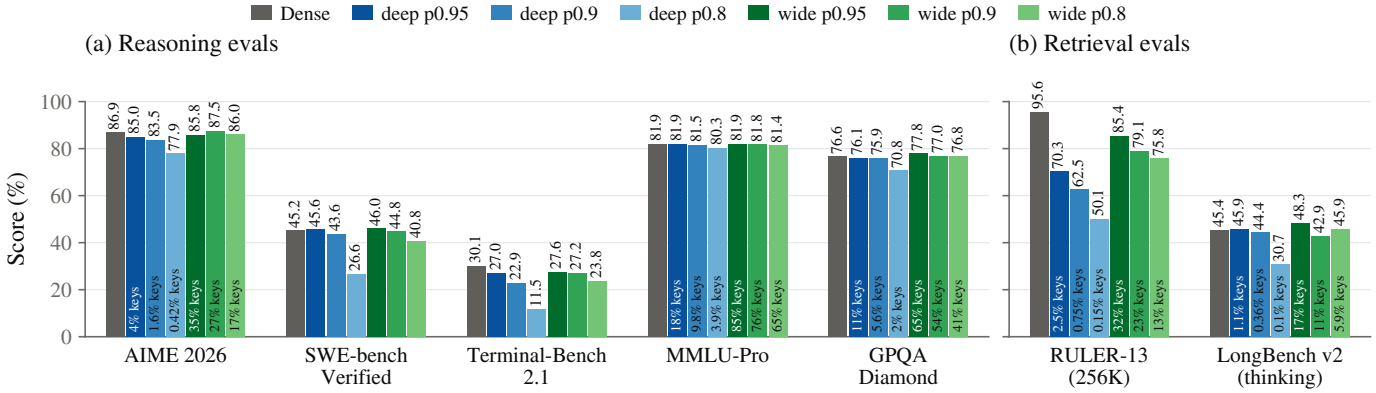


Figure 17: Nemotron-3.5-Lightning-30B-A3B scores and decode key shares across routing thresholds for deep (C4 L6, blue) and wide (C256 L1, green) ZHA, with $R = 16$. Darker shades indicate higher p. (a) Reasoning; (b) retrieval. Labels above bars show the benchmark scores and labels inside bars give key shares where available.

APPENDIX C
GEOMETRY, RETRIEVAL ACCURACY, AND EFFICIENCY

Here we provide supplementary material for the performance analysis of a variety of ZHA geometries and sparsities, including further distinguishing their performance per retrieval task and studying which geometries are on the sparsity–performance Pareto frontier.

A. Geometry Sweep

We are careful to cover a wide range of geometries to obtain enough coverage on the effect of each variable in (R, C, L, p). Figure 18 summarizes the 28 geometries used in our fits, spanning reference counts $R \in \{1, 2, 4, 16\}$, chunk sizes C from 4 to 256, and depths L from 1 to 6. We focus on coverage of top-level compression C^L, ranging from 16 to 4096; matched-compression comparisons varying the decomposition into C and L; and repeated (C, L) pairs across R to isolate the effect of reference-query expressiveness. Each geometry is evaluated at $p \in \{0.7, 0.8, 0.9, 0.95, 0.99\}$ and sequence lengths of 64K, 128K, and 256K.

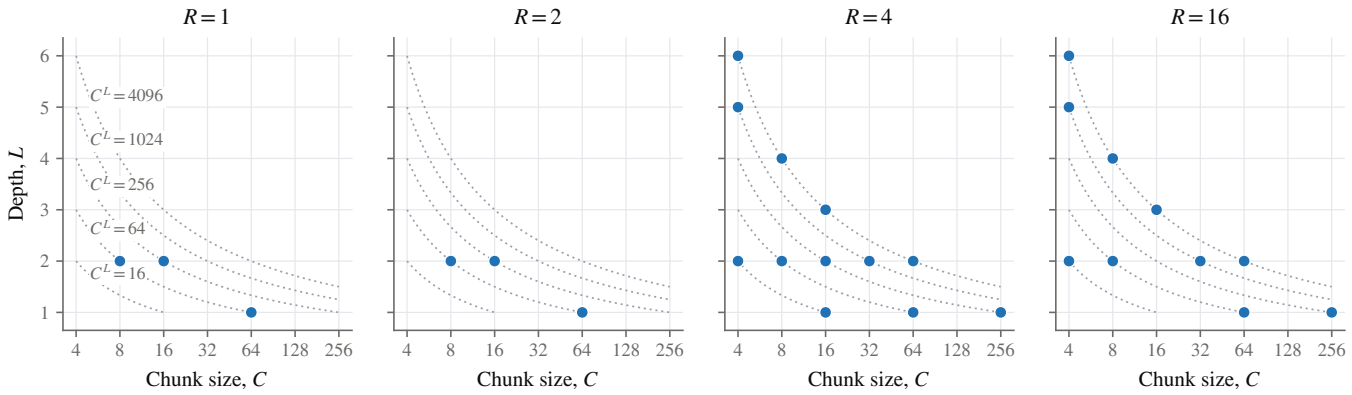


Figure 18: The 28 geometries used in the RULER fits. Each panel fixes the reference count R; dotted curves mark constant top-level comparession C^L. These geometries are evaluated at $p \in \{0.7, 0.8, 0.9, 0.95, 0.99\}$ and sequence lengths of 64K, 128K, and 256K.

B. Effect of Geometry on Per-Task RULER Accuracy

Section IV-B fitted retrieval accuracy in terms of the ZHA geometry and sparsity, but performance was measured as the aggregate accuracy across the 13 RULER tasks, but there is enormous variation in the qualitative retrieval dynamics and difficulty of the RULER subtasks. Some tasks require not only sharp but narrow retrieval in order to ignore a large set of distractor needles, while other tasks require multi-needle or multi-hop retrieval. As such, it is meaningful to study the accuracy of ZHA geometries per RULER task, as this may reveal particular weaknesses of different geometries and inform which provide the best tradeoff of speed and performance.

Figure 19(a)–(c) show the same views of geometry's effect on retrieval accuracy, that separate the effect of varying compression (C^L), depth (L), and per-node expressiveness (R); panel (d) shows the displays the simple geometry-to-performance model's quality-of-fit to the empirical RULER accuracy data (in this case, per task). The first observation is the variety in the "loss-less sparsity" (i.e., without accuracy loss) permitted by each task; some admit good performance with a fraction of a percent of dense keys attended (e..g, single needle 1) whereas others see decay by 10% of keys attended (e.g., common-word extraction). This reflects that ZHA's efficacy is bounded by the inherent compressibility of the underlying data. Nevertheless, note also that the same trends with geometry remain, and are consistent across geometries: greater compression can hurt performance, and increasing depth or expressiveness enables greater sparsity at fixed performance. Furthermore, though there is some variability in the fitting quality per task (often when there is little spread in accuracy, whether high or low), the geometry-to-performance model is broadly a good fit to the empirical RULER accuracy across tasks and sequence lengths.

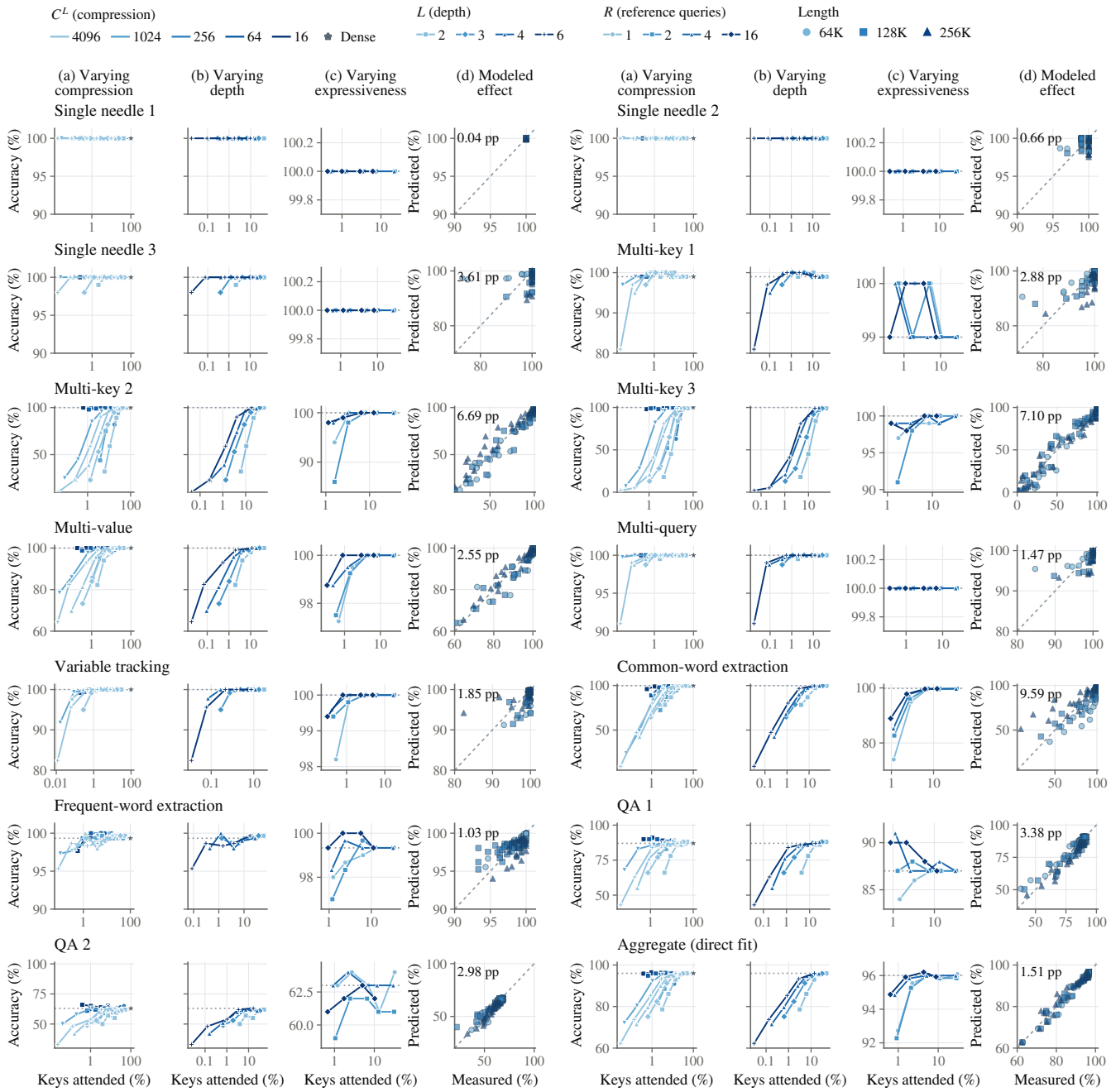


Figure 19: Per-task geometry, sparsity, and prediction accuracy for Qwen3.6-35B-A3B with reference-query-only routing; the final group shows the direct aggregate fit. Panels (a), (b), and (d) fix $R = 16$. (a) Measured accuracy at 256K versus decode keys attended across 11 geometries, colored by C^L. (b) The matched $C^L = 4096$ geometries, colored by depth L. (c) The $C = 8$, $L = 2$ slice at 256K, colored by reference count R. Curves sweep $p \in \{0.7, 0.8, 0.9, 0.95\}$, with $p = 0.99$ included where available; markers denote depth in (a,b) and R in (c). Key shares use each task's native decode counters (pooled counters for the aggregate), and the key-share axes are logarithmic. Dotted horizontal lines show dense attention. (d) All 150 leave-one-length-out predictions per response over 64K, 128K, and 256K, colored and marked by held-out length; inset values give RMSE in percentage points and dashed diagonals show equality. Each task has four independent parameters in equation (2), fitted at the other two lengths across $R \in \{1, 2, 4, 16\}$ using the measured dense baseline at the held-out length. Accuracy uses 100 examples per task; vertical-axis ranges vary by task.

C. Geometry–Performance Pareto Frontier

To obtain a holistic view of the aggregate effect of different geometries on retrieval performance, we study the 28 geometries above together in Figure 20, comparing accuracy with decode keys attended and modeled decode-attention speedup. At 256K, the high-accuracy end of the sparsity frontier favors $R = 16, C = 4, L = 2$, which achieves 95.8% accuracy at $p = 0.7$ while attending to 0.38% of keys. However, the deeper $L = 5$–6 variants reach still lower key fractions but with substantially lower accuracy. The modeled speedup frontier instead favors shallower or lower-R trees: at $p = 0.7$, $R = 4, C = 16, L = 1$ gives 96.4% accuracy at $1.93\times$, $R = 4, C = 64, L = 1$ gives 95.5% at $2.15\times$, and $R = 1, C = 8, L = 2$ gives 92.7% at $2.28\times$. That these shallower or more lightweight trees are favored over the deeper variants reflects that minimizing the attended key count alone does not maximize speedup, as routing and selection also contribute to attention cost.

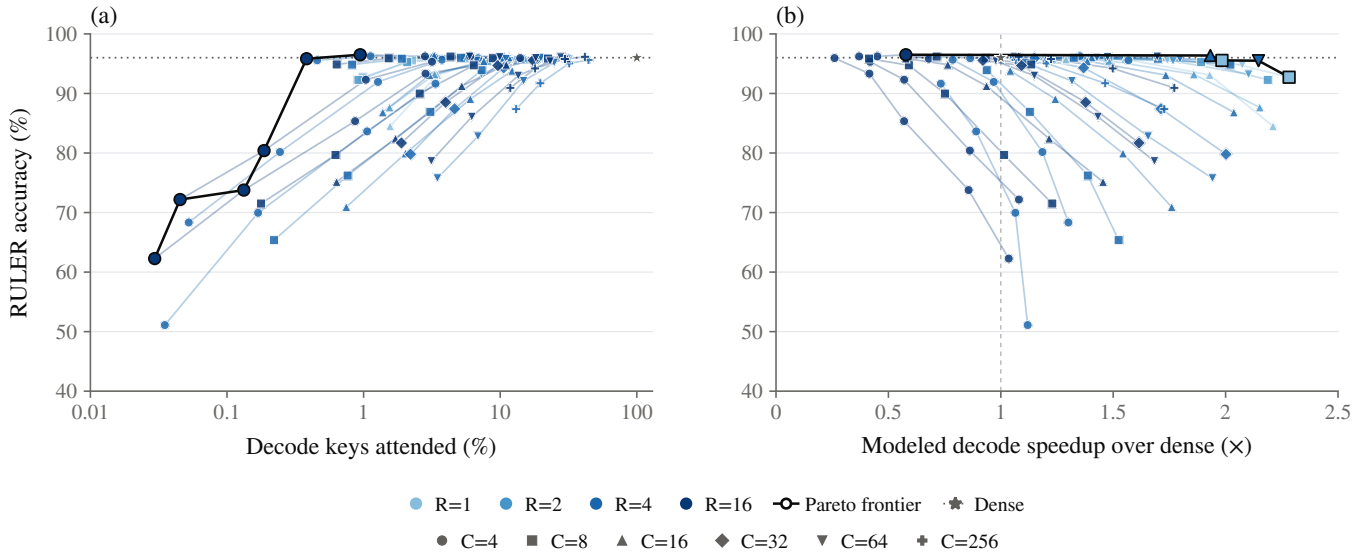


Figure 20: Measured RULER-13 accuracy at 256K versus (a) measured decode key share and (b) modeled decode-attention speedup over dense for Qwen3.6-35B-A3B (115 settings, 28 geometries). Speedups are H100 BF16 attention-layer projections at batch size one, not measured end-to-end timings; missing higher-level union counts use the slower modeled scenario. Colors denote R, markers C, and curves sweep p within each geometry. Black lines mark Pareto frontiers; dotted horizontal lines show dense accuracy, and the vertical guide marks $1\times$.

APPENDIX D
ROOFLINE MODEL

We use roofline models to compute the theoretical latencies of various attention mechanisms to estimate their optimized real-world speeds. Specifically, we build models for dense attention, ZHA, and MSA. We study prefill and (amortized) decode.

A. Dense

We begin with dense attention, using the same execution variables as ZHA below. At fixed workload dimensions, these quantities are deterministic.

Table V: Variables for the dense-attention roofline model.

Variable	Definition
B	Batch size.
$T,\ Q(T)$	Context and query lengths; $Q(T) = 1$ in decode and $Q(T) = T$ in full prefill.
$H_q,\ H_{\mathrm{KV}},\ G$	Query and KV head counts, and GQA group size $G = H_q/H_{\mathrm{KV}}$.
$d_k,\ d_v$	Key and value dimensions per head.
$s_{\mathrm{QKV}},\ s_{\mathrm{accum}}$	Bytes per stored Q/K/V or output element, and per accumulator or softmax statistic.
x	Kernel stage: attention, split-KV partials, or merge.
$\lambda_x,\ \lambda_A$	Per-kernel launch overhead; $\lambda_A = \lambda_{\mathrm{attn}}$ for unsplit attention.
$\Phi_x(T)$	Execution latency of stage x, excluding launches.
$\mathrm{FLOPs}_x(T),\ \mathrm{bytes}_x(T)$	Operation count and memory traffic of stage x.
$P_x,\ \mathrm{BW}_x$	Effective compute throughput (FLOP/s) and memory bandwidth (bytes/s).
$N_{\mathrm{qtiles},x}(T)$	Query-side tile count, before key partitioning.
$N_{\mathrm{tiles},x}(T)$	Launch-grid block count, rather than inner-loop tiles.
$\kappa_{Q,x},\ \kappa_{H,x}$	Query positions and query heads per tile within one KV group.
$\kappa_{\mathrm{attn}},\ J(T)$	Keys per split-KV partial block, and splits per query: $J(T) = \lceil T/\kappa_{\mathrm{attn}} \rceil$.

Latency. For each kernel, the roofline execution time is

$$\Phi_x(T) = \max\left\{ \frac{\mathrm{FLOPs}_x(T)}{P_x}, \frac{\mathrm{bytes}_x(T)}{\mathrm{BW}_x} \right\}. \tag{9}$$

Dense attention has no tree construction, routing, or selection. Its unsplit latency is simply

$$t_{\mathrm{dense}}(T) = \lambda_A + \Phi_{\mathrm{attn}}(T). \tag{10}$$

For split-KV decode or noncausal attention, the partial and merge kernels give

$$t_{\mathrm{dense,split}}(T) = \lambda_{\mathrm{partial}} + \Phi_{\mathrm{partial}}(T) + \lambda_{\mathrm{merge}} + \Phi_{\mathrm{merge}}(T). \tag{11}$$

Each stage uses its own effective compute throughput and memory bandwidth.

Work and traffic. With rectangular query/head tiling, the query-side count is

$$N_{\mathrm{qtiles},x}(T) = BH_{\mathrm{KV}} \left\lceil \frac{Q(T)}{\kappa_{Q,x}} \right\rceil \left\lceil \frac{G}{\kappa_{H,x}} \right\rceil = \begin{cases} BH_{\mathrm{KV}} & \text{grouped-head decode } (Q(T) = 1, \kappa_{H,x} = G) \\ BH_q \left\lceil \frac{T}{\kappa_{Q,x}} \right\rceil & \text{prefill } (Q(T) = T, \kappa_{H,x} = 1) \end{cases} \tag{12}$$

The $\kappa_{\bullet,x}$ are kernel tile sizes.

For unsplit attention over all T keys, the stage counts are

$$N_{\mathrm{tiles,attn}}(T) = N_{\mathrm{qtiles,attn}}(T), \tag{13}$$

$$\mathrm{FLOPs}_{\mathrm{attn}}(T) = 2BQ(T)H_q(d_k + d_v)T, \tag{14}$$

$$\mathrm{bytes}_{\mathrm{attn}}(T) = s_{\mathrm{QKV}}(d_k + d_v)\left[N_{\mathrm{qtiles,attn}}(T)T + BQ(T)H_q\right]. \tag{15}$$

For causal prefill with one-head query tiles, let $n(T) = \lceil T/\kappa_{Q,\mathrm{attn}} \rceil$. Set $\mathrm{FLOPs}_{\mathrm{attn}}(T) = BH_q(d_k + d_v)T(T+1)$ and replace the KV-read count $N_{\mathrm{qtiles,attn}}(T)T$ in $\mathrm{bytes}_{\mathrm{attn}}(T)$ by

$$BH_q \left[\frac{\kappa_{Q,\mathrm{attn}} n(T)(n(T)-1)}{2} + T \right]. \tag{16}$$

The query and output traffic is unchanged.

For split-KV decode or noncausal attention, the counts are

$$N_{\text{tiles,partial}}(T) = N_{\text{qtiles,partial}}(T)J(T), \qquad N_{\text{tiles,merge}}(T) = N_{\text{qtiles,merge}}(T), \tag{17}$$

$$\text{FLOPs}_{\text{partial}}(T) = 2BQ(T)H_q(d_k + d_v)T, \qquad \text{FLOPs}_{\text{merge}}(T) = BQ(T)H_q\left[2J(T)d_v + 3J(T) - 1\right], \tag{18}$$

$$\text{bytes}_{\text{partial}}(T) = s_{\text{QKV}}(d_k + d_v)N_{\text{qtiles,partial}}(T)T + BQ(T)H_q\left[s_{\text{QKV}}d_k + s_{\text{accum}}J(T)(d_v + 2)\right], \tag{19}$$

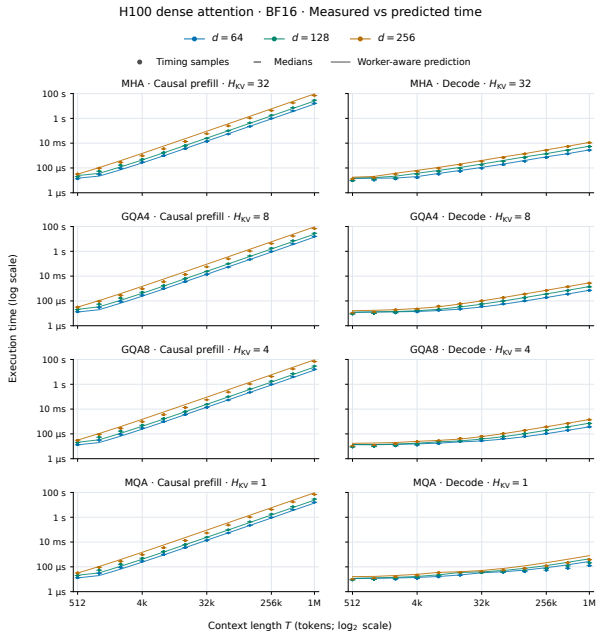
$$\text{bytes}_{\text{merge}}(T) = BQ(T)H_q\left[s_{\text{accum}}J(T)(d_v + 2) + s_{\text{QKV}}d_v\right]. \tag{20}$$

For the merge, P_{merge} is an effective scalar arithmetic rate. Query loads are assumed cached across splits. Each partial writes a d_v-element vector and two softmax statistics at s_{accum} bytes per element; the merge reads them and writes the final output at s_{QKV} bytes per element. Both launches are included explicitly.

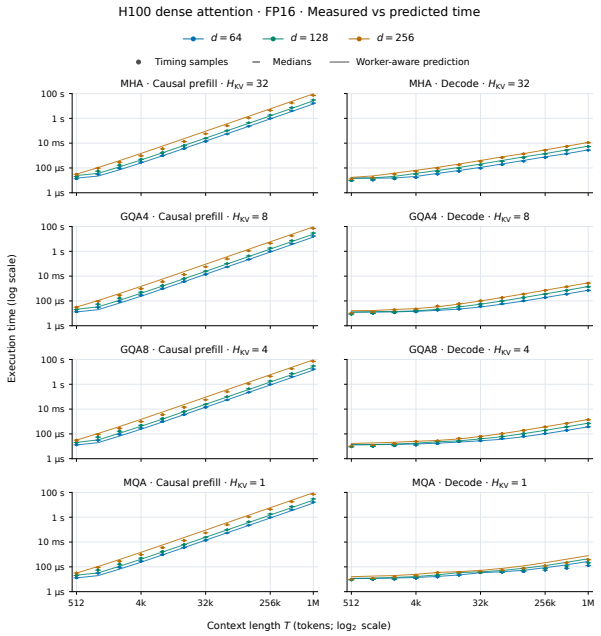
The existing worker-aware calibration was checked against native FlashAttention [7] on 576 held-out H100 cases (144/row). Each uses $B = 1$, $H_q = 32$; $d_k = d_v \in \{64, 128, 256\}$ and $H_{\text{KV}} \in \{1, 4, 8, 32\}$, with $T = 2^9, \ldots, 2^{20}$. The error is $e = 100(t_{\text{model}}/t_{\text{Flash}} - 1)$; calibration uses separate cases.

Mode	Storage	Median e	Median $\lvert e\rvert$	90th percentile $\lvert e\rvert$
Causal prefill	BF16	-0.9%	15.2%	57.2%
Causal prefill	FP16	-1.0%	16.5%	54.9%
Decode	BF16	12.9%	12.9%	54.1%
Decode	FP16	13.2%	13.3%	53.8%

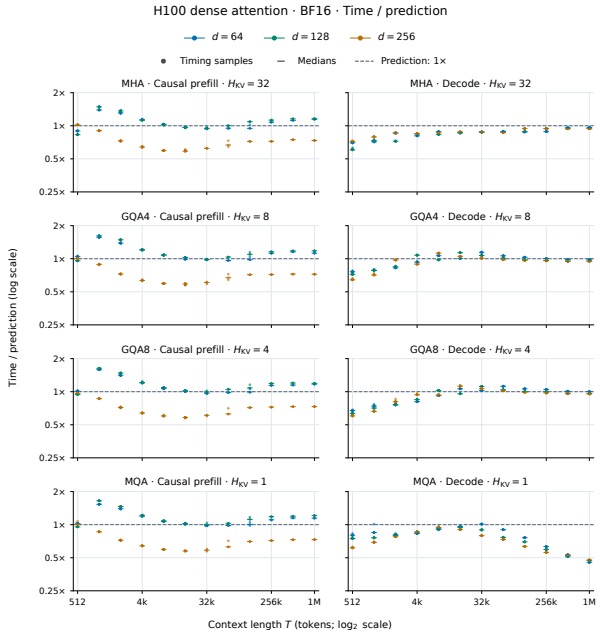
Decode includes the profiled split/merge geometry. These are attention-kernel timings, not end-to-end model timings.



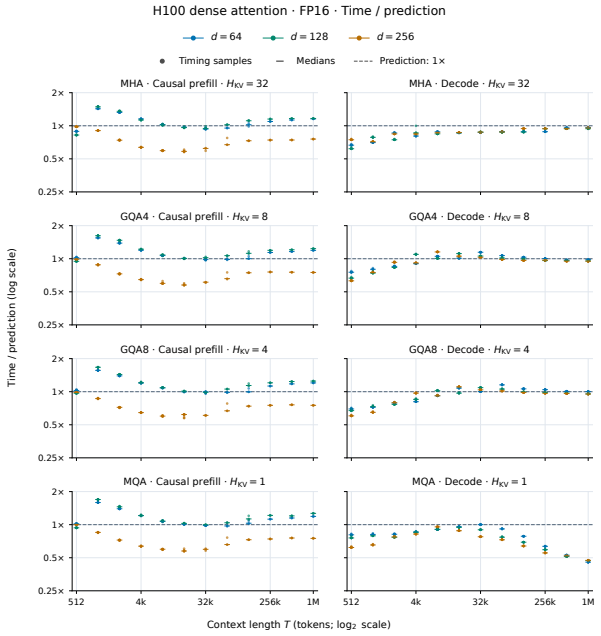
(a) BF16 latency.



(b) FP16 latency.



(c) BF16 timing/prediction.



(d) FP16 timing/prediction.

Figure 21: Held-out dense-attention latency and timing divided by prediction. Within each subfigure, rows are MHA, GQA4, GQA8, and MQA; columns are causal prefill and decode. Colours distinguish $d \in \{64, 128, 256\}$. Dots show all recorded timing samples and short ticks mark medians; solid lines in the latency views show worker-aware predictions. Each CUDA-event sample averages a replay batch, not an individual invocation. The lower views show $t_{\text{Flash}}/t_{\text{model}}$; the dashed $1\times$ line denotes exact prediction, and values below it indicate overprediction. Matching views retain the same layout, colours, marks, and axis limits across precisions; the ratio views share the same ratio scale.

B. ZHA

We now extend the dense model to ZHA, including tree construction, routing, and selection. We model it as

$$\boldsymbol{t}_{\mathrm{ZHA}}(T) = t_{\mathrm{tree}}(T) + \sum_{\ell=1}^{L(T)} \underbrace{\left[\lambda_\ell + \boldsymbol{\Phi}_{\mathrm{routing},\ell}(T) + \boldsymbol{\Phi}_{\mathrm{scan},\ell}(T)\right]}_{\text{Level } \ell \text{ of tree}} + \underbrace{\lambda_A + \boldsymbol{\Phi}_{\mathrm{attn}}(T)}_{\text{Final attention}}. \tag{21}$$

We use λ to represent kernel launch overhead and $\boldsymbol{\Phi}$ to capture execution time, accounting for compute, memory bandwidth, and available parallel workers. Boldface indicates data-dependent random quantities, including deterministic special cases. Workload dimensions, kernel choices, and calibration coefficients are fixed, not random, in this model. Context-length dependence is explicit; in particular, $Q(T)$ and $L(T)$ are deterministic functions set by the workload and tree configuration. Dependence on the input and query is implicit, with routing parameters fixed. For fusion, replace routing and selection by one execution term with its own worker and launch parameters. Variable definitions are provided in Tables VI to VIII. We choose this model for a few reasons:

1) The sum over levels captures the inherent *serial* processing through the depth of tree.
2) It separates kernel launch overhead from execution time, i.e., it does not assume perfect pipelining, which may be challenging with our architecture.
3) It captures data-dependent sparsity and latency.
4) It accounts for potential under-utilization of a GPU at higher levels of the tree.

Table VI: Variables for the ZHA roofline model.

Variable	Definition
B	Batch size.
$T,\ Q(T)$	Context and query lengths; $Q(T) = 1$ in decode and $Q(T) = T$ in full prefill.
$H_q,\ H_{\mathrm{KV}}$	Numbers of query heads and key/value heads, respectively.
$d_k,\ d_v$	Query/key and value dimensions per head, respectively.
G	Query heads per KV head (GQA group size): $G = H_q/H_{\mathrm{KV}}$.
s_{QKV}	Bytes per stored Q/K/V, compressed-key, reference-bias, or routing-score element; BF16 baseline: $s_{\mathrm{QKV}} = 2$.
s_{accum}	Bytes per stored accumulator or softmax/tree statistic element.
R	Number of reference queries.
C	Tree chunk size: tokens per leaf chunk and children per internal node.
$L(T),\ \ell$	Configured tree depth at context length T and level index, respectively. A fixed-depth configuration has constant $L(T)$.
$p,\ K$	Selection threshold and fixed-count selection budget, respectively, when used by the chosen selector.
$\boldsymbol{t}_{\mathrm{ZHA}}(T)$	Total modeled attention-subsystem latency, including tree work and launches.
$t_{\mathrm{tree}}(T)$	Deterministic tree-construction cost for prefill or amortized maintenance cost for decode, including launches.
$\boldsymbol{\Phi}_{\mathrm{routing},\ell}(T)$	Routing execution; scores R references for each of the $\boldsymbol{K}_\ell(T)$ candidate nodes at level ℓ.
$\boldsymbol{\Phi}_{\mathrm{scan},\ell}(T)$	Selection execution; scans the scored frontier and applies selection[14] to decide which nodes to retain or expand.
$\boldsymbol{\Phi}_{\mathrm{attn}}(T)$	Exact-attention execution over the $\boldsymbol{K}_0(T)$ retained tokens, excluding launch overhead.
$\boldsymbol{K}_\ell(T)$	Candidate nodes before selection at $\ell \geq 1$; retained tokens at $\ell = 0$. Per query/head or a grouped union, depending on context.
$\boldsymbol{S}_\ell(T)$	Nodes retained after selection at $\ell \geq 1$. Frontier statistics must satisfy the expansion relation below; they are not independent free counts.
$\lambda_\ell,\ \lambda_A$	Total launch overhead for routing/selection at level ℓ and final attention, respectively; sum the actual kernel launches.

With C children per retained node, expansion gives

$$\boldsymbol{K}_{\ell-1}(T) = C\,\boldsymbol{S}_\ell(T), \qquad 1 \leq \ell \leq L(T). \tag{22}$$

At $\ell = 1$, the children are raw tokens. Partially filled nodes contribute only their actual children.

1) Worker-aware Execution For a sparse stage x, with the routing level ℓ implicit when applicable, we use the same worker-aware maximum:

$$\boldsymbol{\Phi}_x(T) = \max\left(\boldsymbol{t}_{x,\text{worker waves}}(T), \frac{\mathbf{FLOPs}_x(T)}{P_x}, \frac{\mathbf{bytes}_x(T)}{\mathrm{BW}_x}\right). \tag{23}$$

[14] e.g., top-p, top-K, min-p, attention threshold, etc.

This is a standard roofline model except for the $\boldsymbol{t}_{x,\text{worker waves}}(T)$ term. In $\mathbf{bytes}_x(T)$, each stored Q/K/V, compressed-key, reference-bias, or routing-score element contributes s_{QKV} bytes per transfer. Accumulation precision is separate from storage width; intermediates kept on-chip incur no HBM traffic. The throughput, occupancy, and block-latency parameters correspond to each stage's operand and accumulation types. For worker waves, we use

$$\boldsymbol{t}_{x,\text{worker waves}}(T) = \boldsymbol{\tau}_x(T) \left\lceil \frac{\boldsymbol{N}_{\text{tiles},x}(T)}{N_{\text{SM}} c_x} \right\rceil . \tag{24}$$

For fixed work per block, $\boldsymbol{\tau}_x(T) = \tau_x$ is constant in this model.

Table VII: Kernel and worker variables for the roofline model.

Variable	Definition
x	Kernel label: routing, scan, attention, split partial/merge, fusion, or tree summarization/merge. Sparse stage quantities also carry ℓ when level-specific.
$\boldsymbol{\Phi}_x(T)$	Sparse-stage execution latency, excluding kernel launches.
$\boldsymbol{t}_{x,\text{worker waves}}(T)$	Worker-wave contribution to sparse-stage execution latency.
$\mathbf{FLOPs}_x(T),\ \mathbf{bytes}_x(T)$	Sparse-stage operation count and memory traffic (bytes).
$\boldsymbol{N}_{\text{tiles},x}(T)$	Product of launch-grid dimensions, not inner-loop tiles. In 1-D: $\lceil \boldsymbol{n}_x(T)/\kappa_x \rceil$. Unsplit attention is a deterministic special case.
$\boldsymbol{n}_x(T)$	Work-item count for a generic 1-D partition.
$N_{\text{qtiles},x}(T)$	Deterministic query-side tile count, before key/frontier partitioning.
N_{SM}	Number of GPU execution units: SMs on NVIDIA, CUs on AMD.
$\boldsymbol{Q}(T),\ \boldsymbol{K}(T),\ \boldsymbol{V}(T)$	Query, key, and value matrices; $\boldsymbol{Q}(T)$ is distinct from the scalar query length $Q(T)$.
$\boldsymbol{K}_c(T)$	Compressed-key matrix used in routing.
$\boldsymbol{k}_i,\ \boldsymbol{k}_i^{(\ell)},\ i$	Raw-token key, compressed key at level ℓ, and key index within the level, respectively. These are stored, fixed-index keys.
κ_x	Items per block for a generic 1-D partition.
$\kappa_{Q,x},\ \kappa_{H,x}$	Query positions and query heads per tile, respectively, within one KV group.
$\kappa_{\text{routing}},\ \kappa_{\text{scan}}$	Compressed keys per routing block and node scores per scan block.
$\kappa_{\text{attn}},\ \kappa_{\text{radix}}$	Retained tokens per split-KV block and fixed full-frontier radix pass count, respectively.
λ_x	Per-kernel launch overhead; also applies to the tree kernels.
$P_x,\ \text{BW}_x$	Effective compute throughput (FLOP/s) and memory bandwidth (bytes/s).
c_x	Effective blocks (AMD: workgroups) resident per SM/CU; limited by registers, shared memory/LDS, and hardware limits.
$\tau_x,\ \boldsymbol{\tau}_x(T)$	Per-block execution latency at concurrency c_x, excluding launches: fixed τ_x for fixed work, or $\boldsymbol{\tau}_x(T)$ for data-dependent work.

For reference, the following are full-device hardware limits, not calibrated kernel parameters.[15]

Quantity	A100 SXM4	H100 SXM5	MI300X	MI300A
Workers N_{SM}	108 SMs	132 SMs	304 CUs	228 CUs
Hardware bound on c_x	≤ 32	≤ 32	≤ 32	≤ 32

For separate kernels, we use the provisional occupancy assumptions

$$c_x = \begin{cases} 2, & x \in \{\text{routing}, \text{scan}\}, \\ 1, & x = \text{attn}. \end{cases} \tag{25}$$

These values are not GPU-independent constants: a different GPU or kernel can change c_x, and $\boldsymbol{\tau}_x(T)$ must be calibrated at that concurrency.

Using the query-side tile counts defined above, use κ_{routing} compressed keys per routing block and κ_{scan} node scores per scan block:

$$\boldsymbol{N}_{\text{tiles,routing},\ell}(T) = N_{\text{qtiles,routing}}(T) \left\lceil \frac{R\boldsymbol{K}_\ell(T)}{\kappa_{\text{routing}}} \right\rceil , \tag{26}$$

$$\boldsymbol{N}_{\text{tiles,scan},\ell}(T) = N_{\text{qtiles,scan}}(T) \left\lceil \frac{\boldsymbol{K}_\ell(T)}{\kappa_{\text{scan}}} \right\rceil , \tag{27}$$

$$\boldsymbol{N}_{\text{tiles,attn}}(T) = N_{\text{qtiles,attn}}(T) \begin{cases} 1, & \text{unsplit attention}, \\ \lceil \boldsymbol{K}_0(T)/\kappa_{\text{attn}} \rceil , & \text{split-KV partials}. \end{cases} \tag{28}$$

[15] Sources: A100 architecture, H100 architecture, Ampere tuning, Hopper tuning, AMD specifications, and HIP hardware limits.

Here κ_{attn} is retained tokens per split-KV block; split-KV also needs a separately timed merge pass with $N_{\text{qtiles,attn}}(T)$ blocks. Unsplit attention instead loops over retained keys within each block, so $\boldsymbol{K}_0(T)$ affects $\boldsymbol{\tau}_{\text{attn}}(T)$, not its block count. FLOPs and block latency include all packed query positions and heads. These are per-pass counts; additional selection or reduction passes add their own execution and launch costs.

Gather costs are reflected in the block latencies[16].

FLOPs. The dominant matrix work is

$$\mathbf{FLOPs}_{\text{routing},\ell}(T) \simeq 2BQ(T)H_q R d_k \, \boldsymbol{K}_\ell(T), \tag{29}$$

$$\mathbf{FLOPs}_{\text{attn}}(T) \simeq 2BQ(T)H_q(d_k + d_v) \, \boldsymbol{K}_0(T). \tag{30}$$

The routing contribution is $\boldsymbol{Q}(T)\boldsymbol{K}_c(T)^\top$; selection is accounted for separately. The two attention contributions are $\boldsymbol{Q}(T)\boldsymbol{K}(T)^\top$ and multiplication of the attention weights by $\boldsymbol{V}(T)$.

For radix selection, $\mathbf{FLOPs}_{\text{scan},\ell}(T)$ counts the ordinary floating-point arithmetic of the chosen implementation. P_{scan} is a scalar arithmetic rate, not Tensor Core throughput. Exponentials, bit operations, comparisons, and index handling are reflected in $\boldsymbol{\tau}_{\text{scan},\ell}(T)$.

Bytes. For traffic, assume attention outputs and reference biases use the same storage precision s_{QKV}, measured in bytes. For streaming radix selection, we have to do κ_{radix} full-frontier passes[17], each reading one stored score per candidate. The byte counts are

$$\mathbf{bytes}_{\text{routing},\ell}(T) = \underbrace{R s_{\text{QKV}}(d_k+1) N_{\text{qtiles,routing}}(T)\boldsymbol{K}_\ell(T)}_{\text{compressed keys + biases}} + \underbrace{s_{\text{QKV}} d_k BQ(T)H_q}_{\text{reference queries}} \tag{31}$$

$$\mathbf{bytes}_{\text{scan},\ell}(T) = s_{\text{QKV}}\kappa_{\text{radix}} BQ(T)H_q \boldsymbol{K}_\ell(T), \tag{32}$$

$$\mathbf{bytes}_{\text{attn,unsplit}}(T) = s_{\text{QKV}}(d_k+d_v)\left[N_{\text{qtiles,attn}}(T)\boldsymbol{K}_0(T) + BQ(T)H_q\right], \tag{33}$$

$$\mathbf{bytes}_{\text{attn,split}}(T) = \mathbf{bytes}_{\text{attn,unsplit}}(T) + 2s_{\text{accum}} BQ(T)H_q \left\lceil \frac{\boldsymbol{K}_0(T)}{\kappa_{\text{attn}}} \right\rceil (d_v+2). \tag{34}$$

Here $\lceil \boldsymbol{K}_0(T)/\kappa_{\text{attn}} \rceil$ is the number of KV splits, and s_{accum} denotes the bytes per stored accumulator/statistic element.

Split-KV writes and rereads a d_v-element partial vector and two softmax statistics per query/head and split. Query loads are assumed cached; partial and merge kernels are timed separately.

These leading payload counts include reference biases but omit index/mask reads, retained-ID or mask writes, quantization scales, and algorithm-specific scratch.

2) Tree construction In addition to routing through the tree, we must consider the cost of constructing it. Use the baseline stable-state merge, without Taylor/RoPE re-anchoring, with reference queries already prepared. A leaf computes reference/key scores and weighted key sums: $4RCd_k$ leading FLOPs per query head. A parent rescales and sums C child weighted sums independently for each reference: $2RCd_k$ leading scalar FLOPs.

Per reference, retain a maximum, weight sum, entropy, and d_k-vector, using s_{accum} bytes per element for subsequent merges. Emit normalized reference keys at s_{QKV} bytes per element in the same kernel, rather than an additional mean key or a later normalization pass. The counts below include the leaf kernel's two key reads and the parent kernel's repeated maximum read.

Table VIII: Variables for tree construction and maintenance.

Variable	Definition
$n,\ n_\ell(T)$	Full nodes per sequence; the balanced full-node count at level ℓ is $n_\ell(T) = \lfloor T/C^\ell \rfloor$.
$N_{\text{tiles},x}(n)$	Deterministic launch-grid size for a tree kernel processing n full nodes per sequence.
$\text{FLOPs}_x(n),\ \text{bytes}_x(n)$	Deterministic operation count and memory traffic for that tree kernel.
$t_{\text{summ}}(n),\ t_{\text{merge}}(n)$	Leaf-summarization and parent-merge latencies, respectively, including launches.
$t_{\text{tree}}^{\text{prefill}}(T)$	Cost of constructing all full nodes in prefill.
$\bar{t}_{\text{tree}}^{\text{decode}}(T)$	Tree-maintenance cost per decoded token, amortized over full-node completions.

Tree kernels use the free parameters in Table VII, with $x = \text{summ}$ or $x = (\text{tree}, \text{merge})$.

[16]Is this the best place to account for them?

[17]Is this really true? Is it possible to keep most of this on-chip or in L2?

For $n > 0$ full nodes per sequence, with $G = H_q/H_{\mathrm{KV}}$, the kernel counts are

$$N_{\mathrm{tiles,summ}}(n) = BH_{\mathrm{KV}}n, \qquad N_{\mathrm{tiles,tree,merge}}(n) = BH_q n, \tag{35}$$

$$\mathrm{FLOPs}_{\mathrm{summ}}(n) = 4BH_q nRCd_k, \qquad \mathrm{FLOPs}_{\mathrm{tree,merge}}(n) = 2BH_q nRCd_k, \tag{36}$$

$$\mathrm{bytes}_{\mathrm{summ}}(n) = BH_q n\left[2s_{\mathrm{QKV}}d_k(C/G + R) + s_{\mathrm{accum}}R(d_k + 3)\right], \tag{37}$$

$$\mathrm{bytes}_{\mathrm{tree,merge}}(n) = BH_q nR\left[s_{\mathrm{accum}}C(d_k + 4) + s_{\mathrm{accum}}(d_k + 3) + s_{\mathrm{QKV}}d_k\right]. \tag{38}$$

The kernel costs are

$$t_{\mathrm{summ}}(n) = \lambda_{\mathrm{summ}} + \max\left\{\tau_{\mathrm{summ}}\left\lceil\frac{N_{\mathrm{tiles,summ}}(n)}{N_{\mathrm{SM}}c_{\mathrm{summ}}}\right\rceil, \frac{\mathrm{FLOPs}_{\mathrm{summ}}(n)}{P_{\mathrm{summ}}}, \frac{\mathrm{bytes}_{\mathrm{summ}}(n)}{\mathrm{BW}_{\mathrm{summ}}}\right\}, \tag{39}$$

$$t_{\mathrm{merge}}(n) = \lambda_{\mathrm{tree,merge}} + \max\left\{\tau_{\mathrm{tree,merge}}\left\lceil\frac{N_{\mathrm{tiles,tree,merge}}(n)}{N_{\mathrm{SM}}c_{\mathrm{tree,merge}}}\right\rceil, \frac{\mathrm{FLOPs}_{\mathrm{tree,merge}}(n)}{P_{\mathrm{tree,merge}}}, \frac{\mathrm{bytes}_{\mathrm{tree,merge}}(n)}{\mathrm{BW}_{\mathrm{tree,merge}}}\right\}. \tag{40}$$

Leaf blocks share keys across G query heads; parent blocks are per query head. Softmax, entropy, and normalization enter the calibrated block latencies; $P_{\mathrm{tree,merge}}$ is a scalar rate, not a Tensor Core rate. Both costs are zero for $n = 0$.

For the balanced full-node model, let $n_\ell(T) = \lfloor T/C^\ell \rfloor$. Prefill batches all nodes at each level; decode averages one completed node per sequence every C^ℓ tokens:

$$t_{\mathrm{tree}}^{\mathrm{prefill}}(T) = t_{\mathrm{summ}}(n_1(T)) + \sum_{\ell=2}^{L(T)} t_{\mathrm{merge}}(n_\ell(T)), \tag{41}$$

$$\bar{t}_{\mathrm{tree}}^{\mathrm{decode}}(T) = \frac{t_{\mathrm{summ}}(1)}{C} + \sum_{\ell=2}^{L(T)} \frac{t_{\mathrm{merge}}(1)}{C^\ell} \tag{42}$$

$$= \frac{t_{\mathrm{summ}}(1)}{C} + \frac{t_{\mathrm{merge}}(1)}{C(C-1)}\left(1 - C^{-(L(T)-1)}\right). \tag{43}$$

These counts exclude partial boundary nodes. The decode expression is an average, not a tail latency; an unchanged resident tree has $t_{\mathrm{tree}}(T) = 0$.

C. MSA

MSA uses one index-query head per KV group and *one shared index-key head*, both of width d_I. It scores every visible token, max-pools within C-token blocks, then selects k blocks per group. The local block occupies one of these slots.

The shared workload and hardware variables retain the definitions in Tables V and VII; Table IX defines the additional notation used here. The block size C and selection counts below are local to MSA.

Table IX: Additional variables for the MSA roofline model.

Variable	Definition
$C,\ k$	Tokens per block and maximum selected blocks per KV group, including the local block.
$M(T),\ k_T$	Visible block count and selected block count at context length T.
$r(T),\ K(T)$	Tokens in the final, local block and total retained tokens per KV group.
$d_I,\ d_{\mathrm{model}}$	Index-head width and model hidden width.
W_I	Combined index-projection width: $W_I = (H_{\mathrm{KV}} + 1)d_I$.
κ_{proj}	Index-projection output channels per tile.
κ_I	Tokens per index-scan tile; a multiple of C.
$\kappa_{H,I},\ \kappa_{H,A}$	Index-query heads and main-attention query heads per tile, respectively.
ν_I	Index-query tile count in decode: $\nu_I = B\lceil H_{\mathrm{KV}}/\kappa_{H,I}\rceil$.
$s,\ s_c,\ s_{\mathrm{id}}$	Bytes per stored element ($s = s_{\mathrm{QKV}}$), pooled score, and integer block ID.
x	Kernel stage: index projection (proj), index scan (I), top-k, or main attention (A).
$N_x,\ F_x,\ V_x$	Stage launch-grid block count, FLOPs, and memory traffic in bytes.
$t_{\mathrm{sparse}}^{\mathrm{MSA}}(K)$	Main-attention latency over K retained tokens, including its launches.
$k_i,\ t_j$	Selected blocks at position i, $k_i = \min(k, M(i))$, and final query position in index-query tile j.
$C_b,\ \boldsymbol{n}_b$	Stored length of block b and number of queries selecting it within its batch and KV group.
$\kappa_{\mathrm{packed}},\ \boldsymbol{a}_b$	Query-head pairs per packed tile and packed tiles for block b: $\boldsymbol{a}_b = \lceil G\boldsymbol{n}_b/\kappa_{\mathrm{packed}}\rceil$.
E	Total query–block selections, summed over batch and KV groups: $E = \sum_b \boldsymbol{n}_b$.

For decode ($Q = 1$), define

$$M(T) = \lceil T/C \rceil, \qquad k_T = \min(k, M(T)), \tag{44}$$

$$r(T) = T - C(M(T) - 1), \qquad K(T) = C(k_T - 1) + r(T). \tag{45}$$

Thus $K(T) = T$ when all blocks fit, and $K(T) \le kC$ otherwise. The selected addresses vary, but these counts are deterministic. The decode stage counts are:

Stage x	N_x	F_x	V_x
Index projections	$B\lceil W_I/\kappa_{\text{proj}} \rceil$	$2Bd_{\text{model}}W_I$	$s[d_{\text{model}}W_I + B(d_{\text{model}} + W_I)]$
Index + max-pool	$\nu_I\lceil T/\kappa_I \rceil$	$2BH_{\text{KV}}d_I T$	$sd_I(\nu_I T + BH_{\text{KV}}) + s_c BH_{\text{KV}} M$
Top-k	BH_{KV}	—	$BH_{\text{KV}}(s_c M + s_{\text{id}}k_T)$

The two index projections are fused into one launch; cached historical index keys are not reprojected. Projection weights and inputs, and index queries across key tiles, are assumed cached within their stage. Only pooled block scores reach HBM, not a T-element token-score vector per index query. Pooling comparisons and exp-free top-k work are charged through their block latencies; they are not Tensor Core FLOPs. The top-k row assumes each group's candidate list fits on-chip.

Using the same worker-aware maximum, with stage-specific parameters,

$$\Phi_x(T) = \max\left\{\tau_x(T)\left\lceil \frac{N_x(T)}{N_{\text{SM}}c_x}\right\rceil, \frac{F_x(T)}{P_x}, \frac{V_x(T)}{\text{BW}_x}\right\}, \tag{46}$$

$$\boxed{t_{\text{MSA}}(T) = \sum_{x\in\{\text{proj},I,\text{top}k\}} [\lambda_x + \Phi_x(T)] + t^{\text{MSA}}_{\text{sparse}}(K(T)).} \tag{47}$$

Omit F_x/P_x for top-k, whose scalar work is included in $\tau_{\text{top}k}(M, k_T)$. There is one top-$k$ launch, not k_T launches. For query-outer main attention, the unsplit counts are

$$N_A = BH_{\text{KV}}\lceil G/\kappa_{H,A} \rceil, \tag{48}$$

$$F_A = 2BH_q(d_k + d_v)K(T), \tag{49}$$

$$V_A = s(d_k + d_v)[N_A K(T) + BH_q]. \tag{50}$$

For split-KV, use the partial/merge counts in equations (17) to (20), replacing the *key length* by $K(T)$, not the query length. The same two-statistic partial-state convention applies. As elsewhere, final-attention index reads, gather penalties, and padding are not separately counted; they require kernel-specific calibration.

For causal prefill, sum the visible per-position counts:

$$F^{\text{prefill}}_{\text{proj}} = 2BTd_{\text{model}}W_I, \tag{51}$$

$$F^{\text{prefill}}_I = BH_{\text{KV}}d_I T(T+1), \tag{52}$$

$$F^{\text{prefill}}_A = 2BH_q(d_k + d_v)\sum_{i=1}^{T} K(i). \tag{53}$$

Pooled-score and selected-ID counts become $BH_{\text{KV}}\sum_i M(i)$ and $BH_{\text{KV}}\sum_i k_i$, respectively. If index query tile j ends at t_j, its key-read count uses t_j and its key-grid count uses $\lceil t_j/\kappa_I \rceil$; do not multiply the final-position decode cost by T.

Native MSA prefill instead uses KV-outer scheduling. Let $\boldsymbol{n}_b$ be the number of queries selecting block b within its KV group, and $\boldsymbol{a}_b = \lceil G\boldsymbol{n}_b/\kappa_{\text{packed}} \rceil$ its packed query tiles. Sums below also include batch and KV group; C_b is the stored block length and $E = \sum_b \boldsymbol{n}_b = BH_{\text{KV}}\sum_i k_i$. For one CTA per packed query tile, the partial-attention grid and traffic are

$$\boldsymbol{N}_{\text{partial}} = \sum_b \boldsymbol{a}_b, \tag{54}$$

$$\mathbf{bytes}_{\text{partial}} = s(d_k + d_v)\sum_b C_b\boldsymbol{a}_b + sd_kGE + s_{\text{accum}}(d_v + 2)GE. \tag{55}$$

Add reverse-index/scheduling launches and scratch traffic, then combine k_i partials per query using the merge model. Native chunking or persistent schedules must use their actual launch grids, partial counts, and traffic. Block popularity therefore affects worker waves and KV reuse, even at fixed k; the decode comparison below does not claim to reproduce this native prefill schedule.

APPENDIX E
SPARSITY SCALING ACROSS ROUTING THRESHOLDS

The latency extrapolations in Section V assume a sublinear power-law scaling at each tree level that changes with depth, and therefore we fit those exponents and model their dependence on depth in. Here we extend the main-text $p = 0.8$ results to other thresholds, explain how these trends are estimated and validated, and connect them to the latency model.

A. Estimating Per-Level Growth

Estimating the per-level sublinear exponent from empirical selection rates requires carefully controlling for outliers in sequence length and example count. We seek typical problem-level growth, limiting the influence of unusually long generations or extreme slopes. The Qwen3.6-35B-A3B sweep uses the same 30 AIME 2026 problems, with 16 generations per setting: $C = 4$, $R = 16$, $L = 2$–6, and $p \in \{0.7, 0.8, 0.9, 0.95\}$ (9,600 completions).

We replay saved generations with single-query tiles to measure shared KV footprints: each selected complete node is counted once across the eight query heads in its group, separately at each position, layer, and tree level. Counts are averaged over two groups, 10 full-attention layers, and complete 256-token bins, excluding the always-retained first key and current partial chunk.

For each problem, we average available generations within each bin and fit a free-intercept log–log slope over 16K–64K (192 native bins). Objective weights $1/T$ approximate equal weighting in log context length. Requiring three complete bins leaves 29–30 problems per setting. Their median slope defines α_ℓ, weighting problems equally without imputing missing tails.

B. Modeling Levels Within a Tree

Provided empirically estimated exponents per level, we may fit our depthwise exponent model. For a fixed tree, β sets the top exponent and ε its correction toward the leaves in equation (7). Equally weighted least squares fits both to the observed α_ℓ separately for each geometry and p, with signed ε, $0.01 \le \beta \le 1$, and $\beta_\ell \ge 0.01$. Note that joint fitting can shift β from the observed top slope, and that the factors describe combined pruning and KV-group union effects.

The parameter r determines how much of the accumulated correction carries into the next lower level:

$$\varepsilon_L = 0, \qquad \varepsilon_{\ell-1} = \varepsilon + r\varepsilon_\ell, \qquad \varepsilon_\ell = \begin{cases} \varepsilon(1 - r^{L-\ell})/(1-r), & r \neq 1, \\ \varepsilon(L-\ell), & r = 1. \end{cases} \tag{56}$$

Here $r = 0$ gives a constant correction below the top, $r = 1/2$ approaches 2ε, and $r = 1$ gives linear drift. Across 20 settings (80 exponent summaries), their RMSEs, weighting settings equally, are 0.0218, 0.0130, and 0.0156. We favor $r = 1/2$ as a compact descriptive choice that balances accuracy across settings: paired bootstraps favor $r = 1/2$ over $r = 0$, though not clearly over $r = 1$; its advantage comes mainly from $p = 0.7$, $L = 6$.

Figure 22 shows decreasing exponent profiles at $p = 0.9, 0.95$; the deepest $p = 0.7$ tree instead rises toward the leaves. We normalize each level and geometry by the first-bin count to make growth rates comparable. For visualizations, we further average 16 native-bin means into each of 12 display bins; fits retain native resolution. With profile slopes fixed, amplitudes c_ℓ fit pooled arithmetic-mean counts by $1/T$-weighted least squares in log space, weighting available generations equally within each bin.

The results for these other p values broadly corroborate the main-text $p = 0.8$ findings; scaling exponents decrease smoothly and predictably proceeding top-down the tree. The exception is the deepest geometry ($C = 4$, $L = 6$, $R = 16$) at $p = 0.7$, whose exponents rise again toward the leaf levels. This setting achieves only 44.0% accuracy on AIME, compared with 93.1% for dense attention, so its anomalous scaling should be interpreted in the context of severe accuracy degradation.

C. Adding Tree Depth

In order to estimate the selection exponents not only within fitted trees, but to extrapolate to larger depths, we must determine what happens when another level is added. To isolate this effect, Figure 23 fits only the five observed top slopes α_L at each p, independently of lower levels:

$$\widehat{\beta}_{\text{const}}(L, p) = a_p, \qquad \widehat{\beta}_{\text{lin}}(L, p) = a_p + b_p(L - 4). \tag{57}$$

The latency scenarios share the dependence on p. Equally weighted least squares on all 20 top slopes gives this form of equation (8):

$$\begin{aligned} \widehat{\beta}_{\text{const}}(p) &= 0.3736 + 0.5536p, \\ \widehat{\beta}_{\text{lin}}(L, p) &= 0.3429 + 0.4129p + [0.007697 + 0.03518p]L. \end{aligned} \tag{58}$$

Comparing a linear-top model to a constant-top, the linear-top follows the observed trend more closely, reducing top-exponent RMSE from 0.0574 to 0.0226. On heldout depths or thresholds beyond the fitted settings, linear-top again improves on constant-top: RMSE 0.0317 versus 0.0706 for held-out depths, and 0.0442 versus 0.0597 for held-out thresholds.

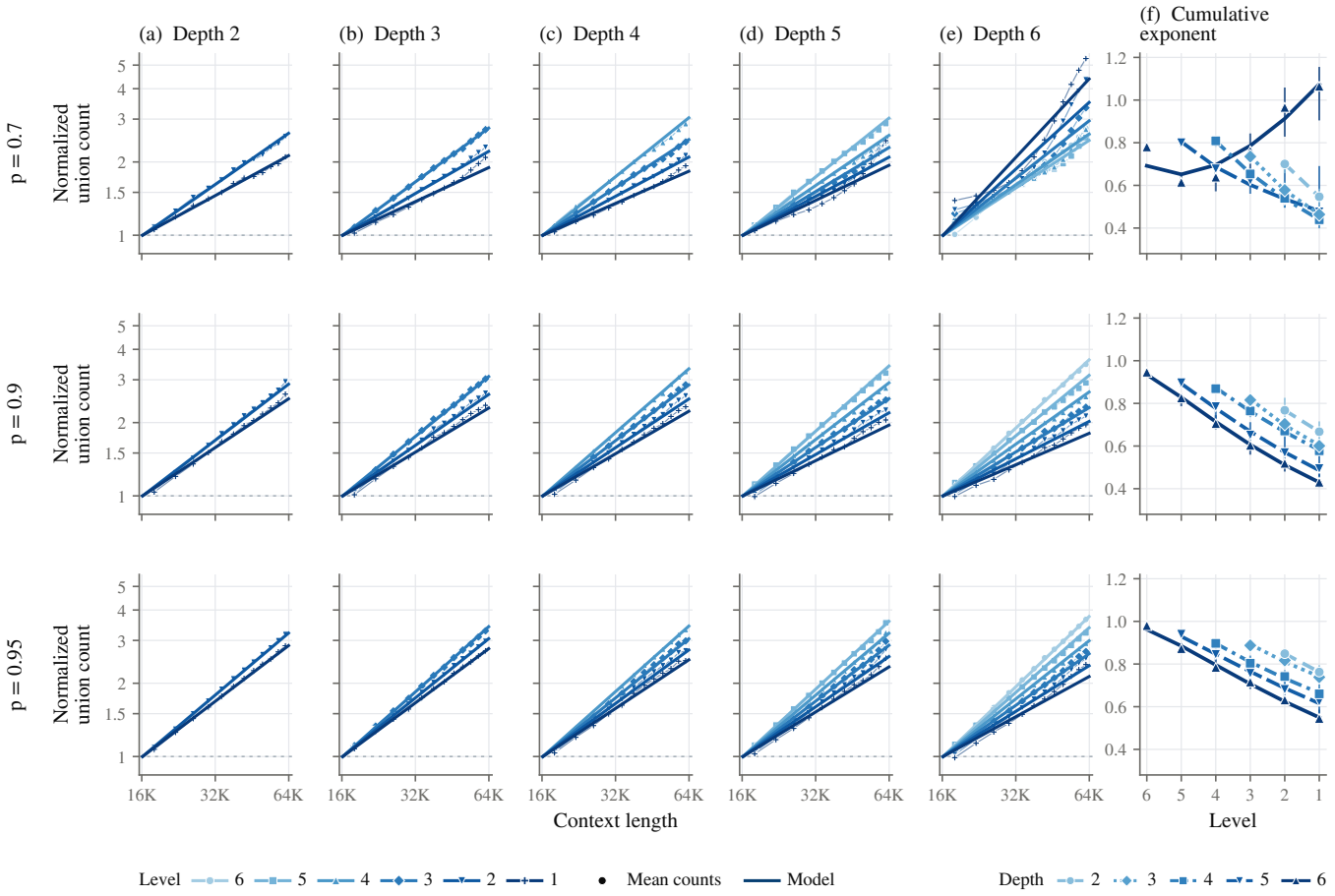


Figure 22: Per-level selection scaling for $C = 4$, $R = 16$, and $p = 0.7, 0.9, 0.95$, extending Figure 7. The first five columns show depths 2–6; the sixth compares observed median exponents with the $r = 1/2$ profile. Bars show pointwise 95% whole-problem bootstrap intervals.

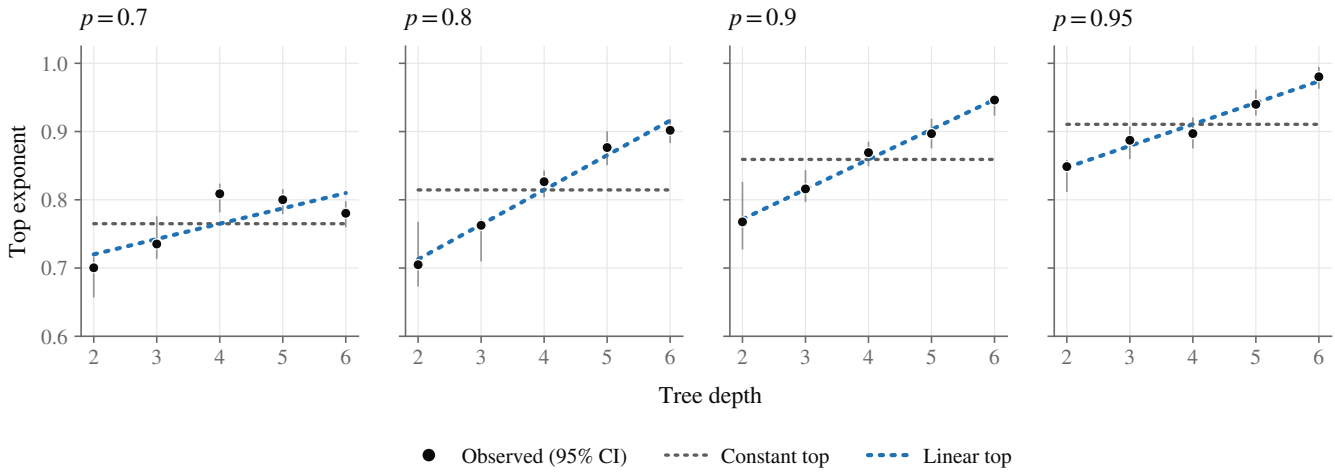


Figure 23: Observed top-level exponents versus total tree depth for $C = 4$, $R = 16$. Dotted lines fit constant (gray) and linear (blue) depth dependence separately at each p, using only the five top-level observations. Bars show pointwise 95% whole-problem bootstrap intervals.

D. Robustness of the Depth Trend

Though the trends identified appear robust, it is important to qualify that we fit in a low-data regime. The deepest trees and longest contexts have the least support: $L = 6$ has only 4–15 complete top-level candidates over 16K–64K, and fewer generations reach 64K. Many positional bins therefore do not imply many independent examples.

We quantify uncertainty with 2,000 whole-problem bootstrap draws, keeping each sampled problem's generations, bins, and levels together and recomputing the median slopes and model fits. Model comparisons pair draws across depths and thresholds and repeat the holdouts. We take the median which reduces sensitivity to extreme slopes, while still not removing differences in the length ranges covered by each problem.

E. From Selection Counts to Latency

Having fitted a model for how the sparsity exponents grow with depth, we can plug in the per-level work that implies to the latency model. We transfer the $C = 4$, $R = 16$ exponent scenarios by depth and calibrate count amplitudes to native $R = 1$, $C = 8$, $L = 2$ AIME measurements (480 generations per p), using pooled counts and the same weighted log fit. Existing $C = 8$ checks find that compression-based transfer can improve the top exponent while worsening the full profile. They also use different texts and, for $R = 1$, a different reference count, so we do introduce error by transferring between adjacent geometries.

The plugged-in selection rates model the per-level expansion in the latency model. That is, the model expands parent unions to obtain candidate counts, adds nodes without complete parents, and includes the first key and partial chunk in final token reads.[18] Figure 24 extends the main-text $p = 0.75$ comparison to $p = 0.7, 0.8$; note that these are tightly clustered relative to the baselines, reflecting their different scaling exponent.

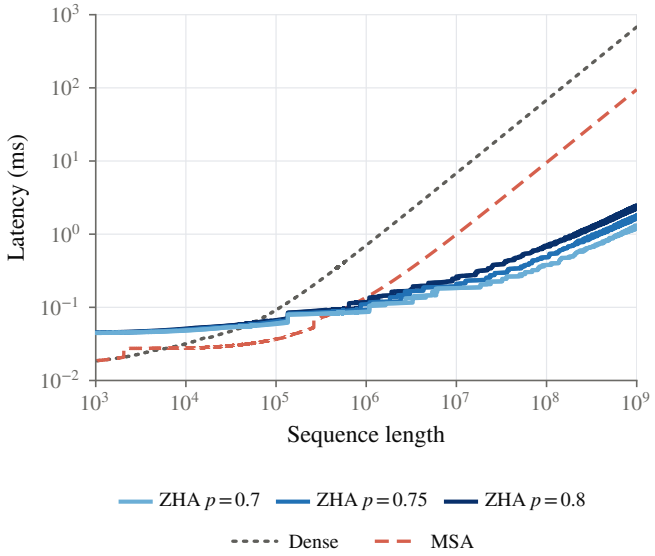


Figure 24: Modeled BF16 decode latency for dense attention, MSA, and ZHA ($R = 1$, $C = 8$) at $p = 0.7, 0.75, 0.8$, with the H100 assumptions and nondecreasing latency-minimizing depth of Figure 8. Latencies are modeled per-token decode times for the attention subsystem of one layer at batch size 1.

[18] At the internal 128K reference, extrapolated from 16K–64K, added-parent occupancy follows $q_{\text{parent}} = 1 - (1 - q_{\text{child}})^{8}$, where q is the selected fraction of nodes.

APPENDIX F
RETRIEVAL TRAVERSAL EXAMPLES

Aggregate key shares show how much of the context ZHA attends to, but not where that budget goes within the tree. Figure 25 follows one retrieval head through each level of a three level router on two 64K RULER prompts. The router's search widens depending on the task difficulty: on the multi-key prompt, whose distractor needles resemble the target, it expands more nodes at multiple regions of the context, while for the single-needle prompt the router is more targeted. These examples come from a single layer, head and decode step; they are meant to illustrate the mechanism, not to represent typical behaviour. Nevertheless, these worked examples demonstrate the power of hierarchical routing in action, where the model is able to accurately find the needle while only activating a small fraction of the total underlying KV cache. Our RULER scores in the main text imply that this retrieval is highly robust in practice, even to challenging multi-key + distractor RULER subtasks.

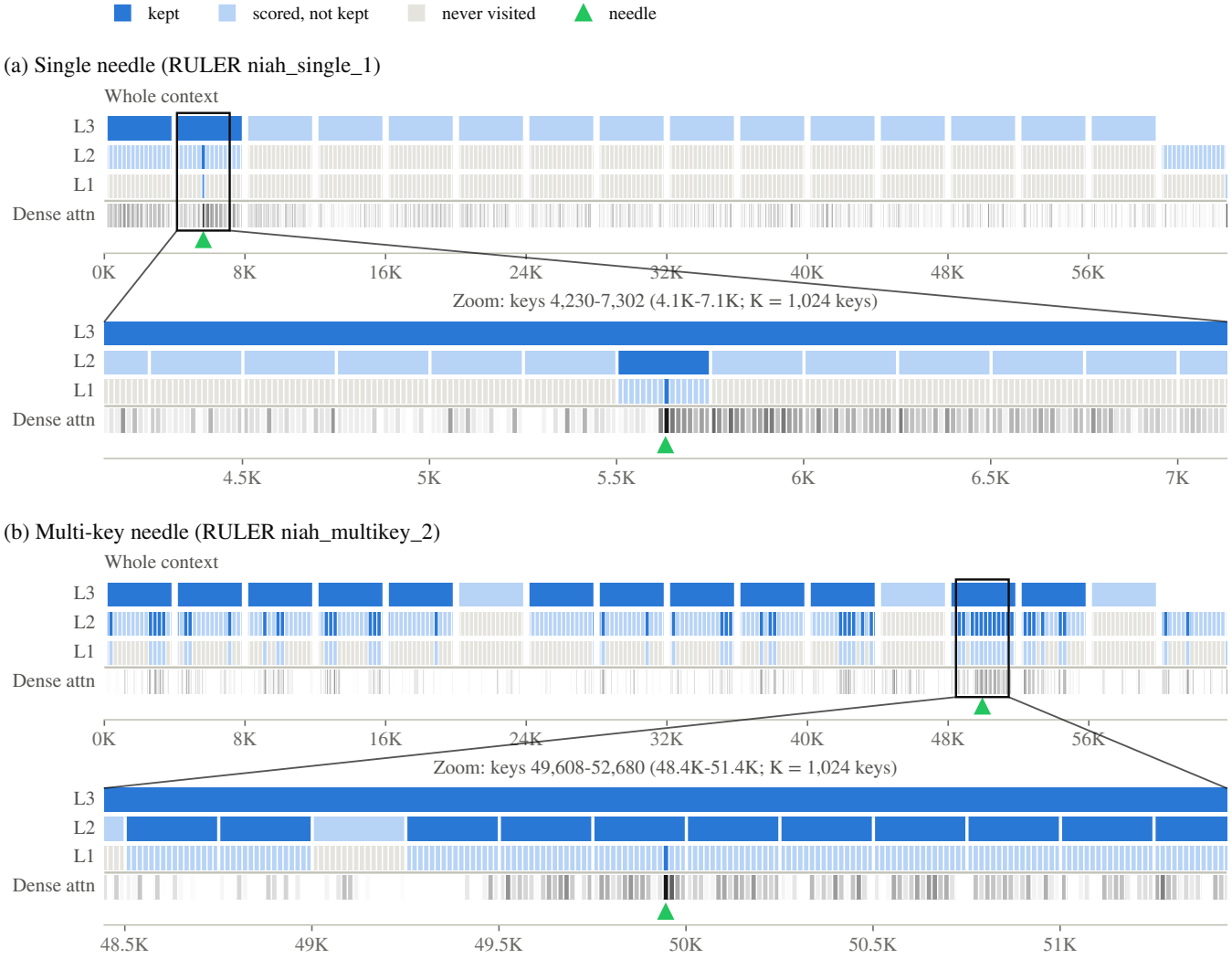


Figure 25: Retrieval traversals at 64K tokens for (a) single-needle and (b) multi-key 2 RULER tasks (Qwen3.6-35B-A3B, C16 L3, $R = 16$, $p = 0.9$). Both show layer 35, query head 8, while emitting the first answer digit. Dark blue blocks are scored and expanded nodes; light blue blocks scored but pruned nodes; grey blocks are nodes which were never scored by the router. Dense-attention rows show chunk-level mass on a logarithmic scale. The green marker indicates the ground-truth needle location. Boxed regions are enlarged below to reveal finer traversal details. These examples illustrate task-dependent routing with top-p: for single-needle, the router explores few branches before reaching the answer-containing chunk; for multi-key 2, whose context contains many distractor needles, it explores substantially more upper-level nodes. Both traversals ultimately attend to a single answer-containing chunk.

APPENDIX G
REFERENCE-QUERY AGREEMENT ACROSS TREE LEVELS

We extend the reference-query usage analysis in Figure 9 to all four geometries and compare aggregate level agreement with individual layer/KV-group examples. Level ℓ contains nodes spanning C^ℓ keys for the geometries shown. Our analysis demonstrates that a similar pattern of rank-correlation between reference queries across levels holds across a range of tree geometries.

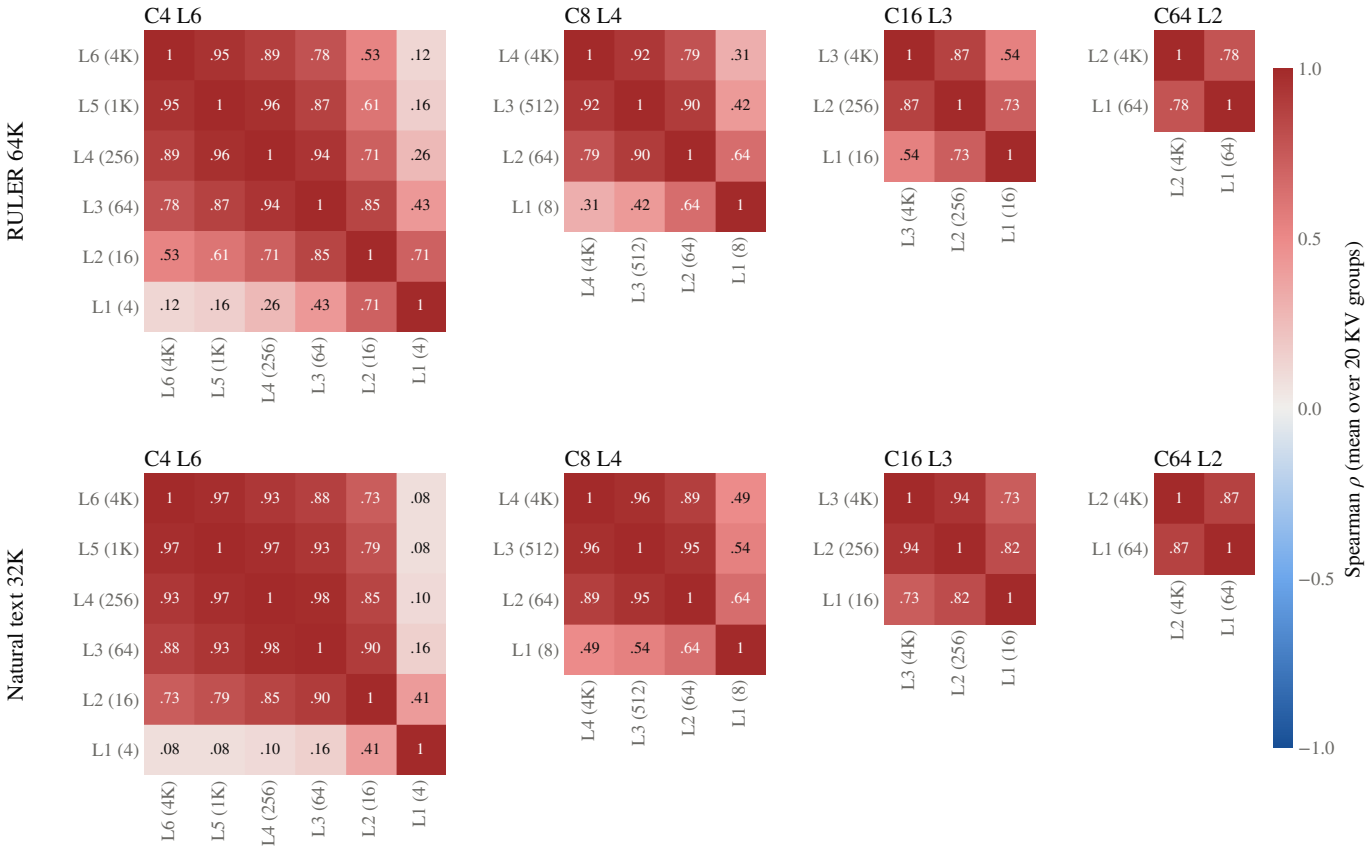


Figure 26: Agreement between tree levels in reference-query selection for Qwen3.6-35B-A3B ZHA ($R = 16$). Columns show C4 L6, C8 L4, C16 L3 and C64 L2; rows show RULER at 64K tokens and natural text at 32K tokens. For each layer/KV group, Spearman correlation is computed between the 16 reference-query selection shares at each pair of levels, then averaged over 20 groups (10 layers $\times$ 2 KV groups). Parenthesized axis labels give node spans in keys. Adjacent levels are similar in their reference selection rate while further apart levels have a different selection priority. This behavior holds across all geometries.

A. *Individual layer and KV-group examples*

We show the actual reference selection behavior per level for a typical KV group in Figure 27, and two outlier behaviors in Figure 28 and Figure 29. The reference queries are shared across levels. In general, selection becomes increasingly concentrated at higher levels, while remaining more uniform at the lowest level.

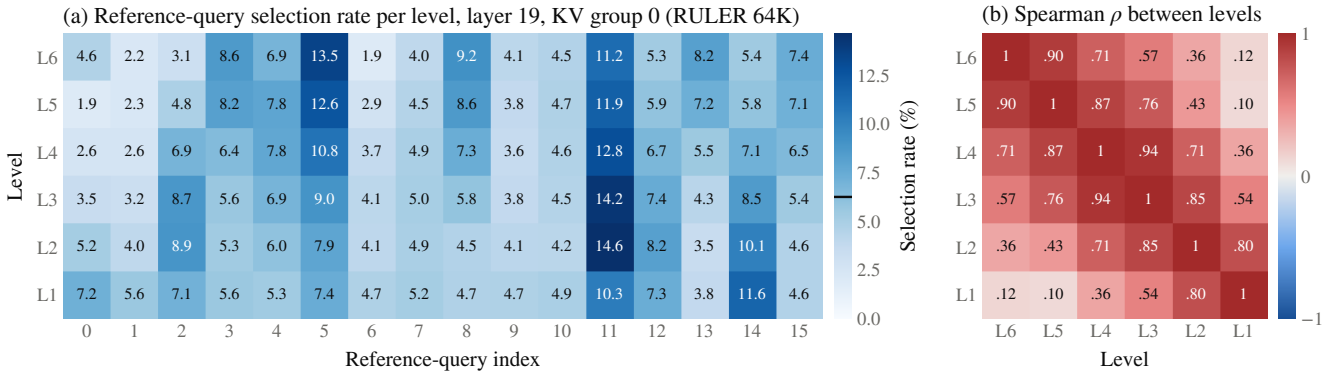


Figure 27: Typical selection behavior across levels: layer 19, KV group 0 of the C4 L6 ZHA router on RULER at 64K tokens. Adjacent levels have similar reference rankings, but agreement weakens over larger level separations; the chunk and top-level rankings are nearly unrelated (L1 versus L6: $\rho = 0.12$).

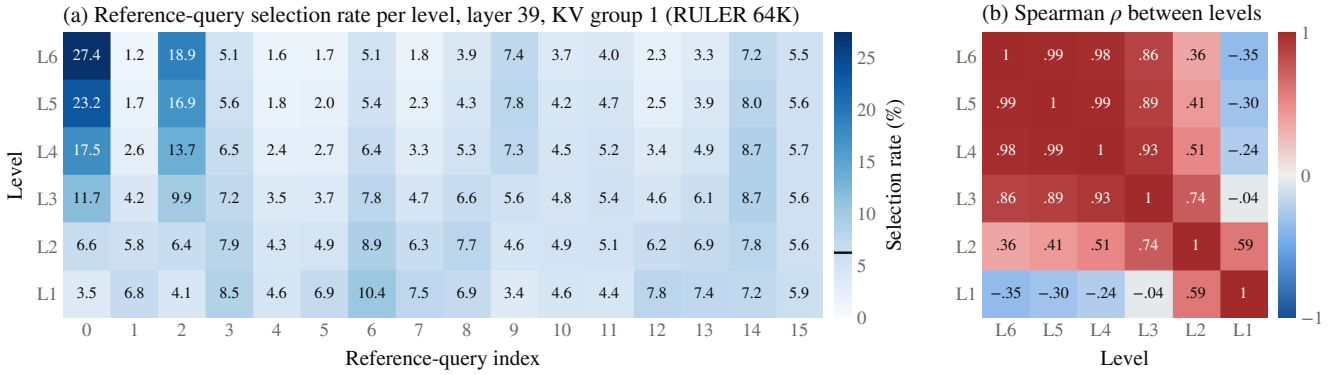


Figure 28: Outlier selection across levels: layer 39, KV group 1 of the C4 L6 ZHA router on RULER at 64K tokens. Upper levels agree closely, while L1 reference rankings are negatively correlated with the top levels (L1 versus L6: $\rho = -0.35$).

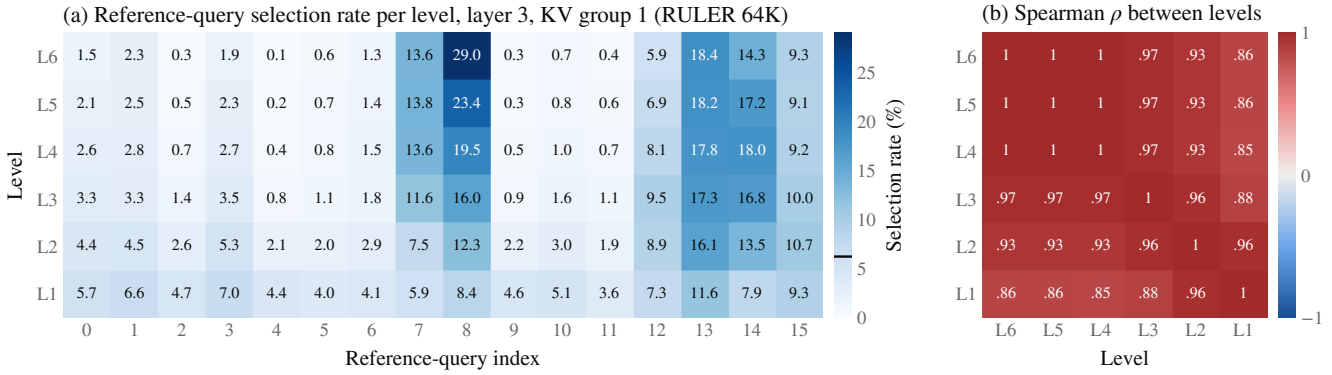


Figure 29: Outlier selection across levels: Similar reference rankings across all levels: layer 3, KV group 1 of the C4 L6 ZHA router on RULER at 64K tokens. Chunk and top-level rankings remain strongly aligned ($\rho = 0.86$).

APPENDIX H
REFERENCE-QUERY AND REFERENCE-KEY SIMILARITY

ZHA learns R reference queries per KV group without explicitly regularizing their structure. Different geometries impose different chunk sizes, depths, and compression levels, raising the question of whether a common reference structure nonetheless emerges. Figures 30 to 32 compare the 16 references across four geometries, examining both the query vectors and the reference keys they produce. Across most layers, matched references are similar across geometries, while reference keys remain differentiated within each geometry. This suggests that training finds a largely geometry-independent, specialized basis. All plots use the four independently trained $R = 16$ geometries C4 L6, C8 L4, C16 L3, and C64 L2; a top level node covers exactly 4096 keys in each geometry.

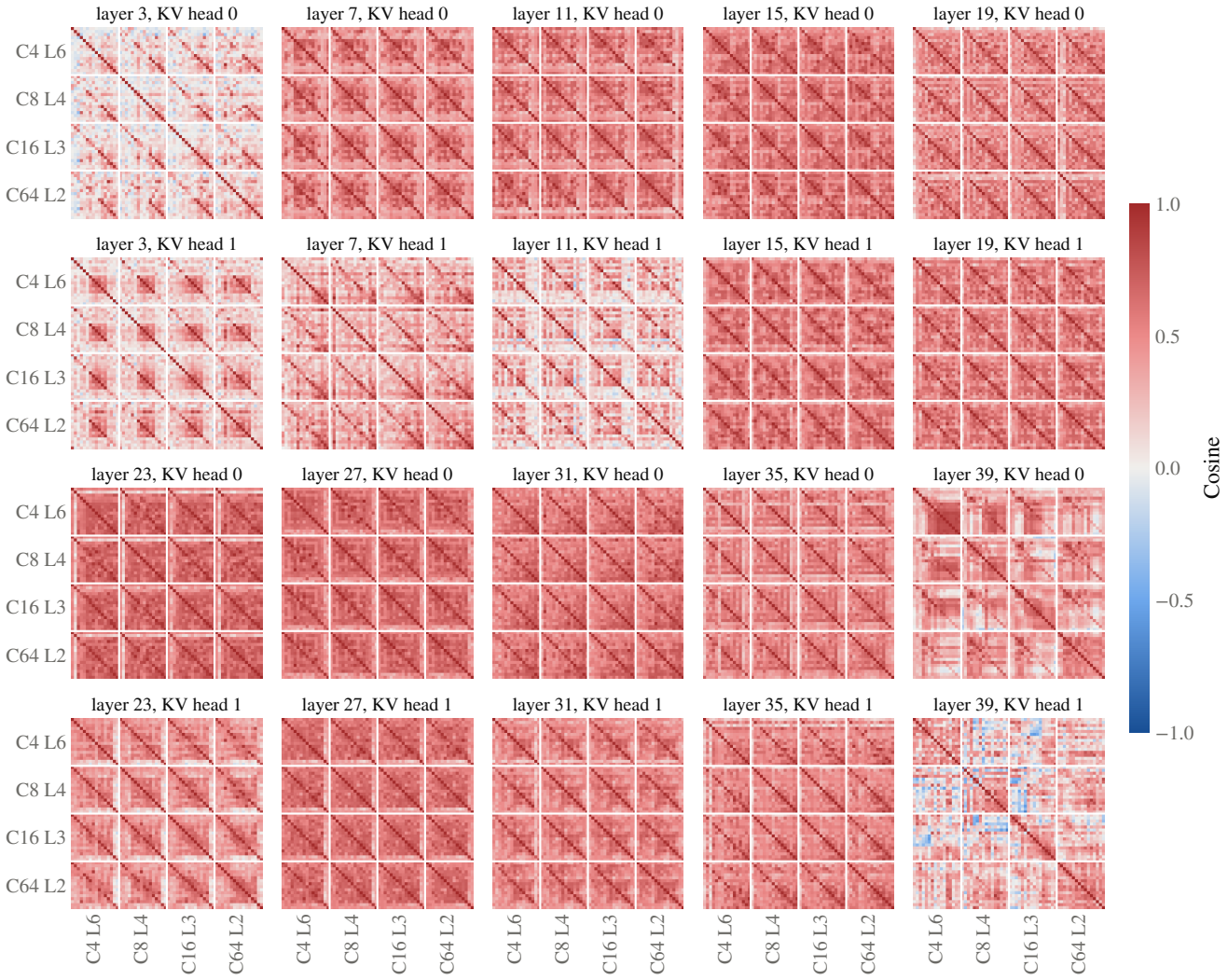


Figure 30: Reference-query cosine similarities across the four Qwen3.6-35B-A3B ZHA geometries. The C16 L3 references are ordered by average-linkage clustering with cosine distance; references in each other geometry are Hungarian-matched to them by cosine similarity. Matched references therefore occupy corresponding positions. Diagonal structure within the off-diagonal blocks indicates similar reference directions across geometries.

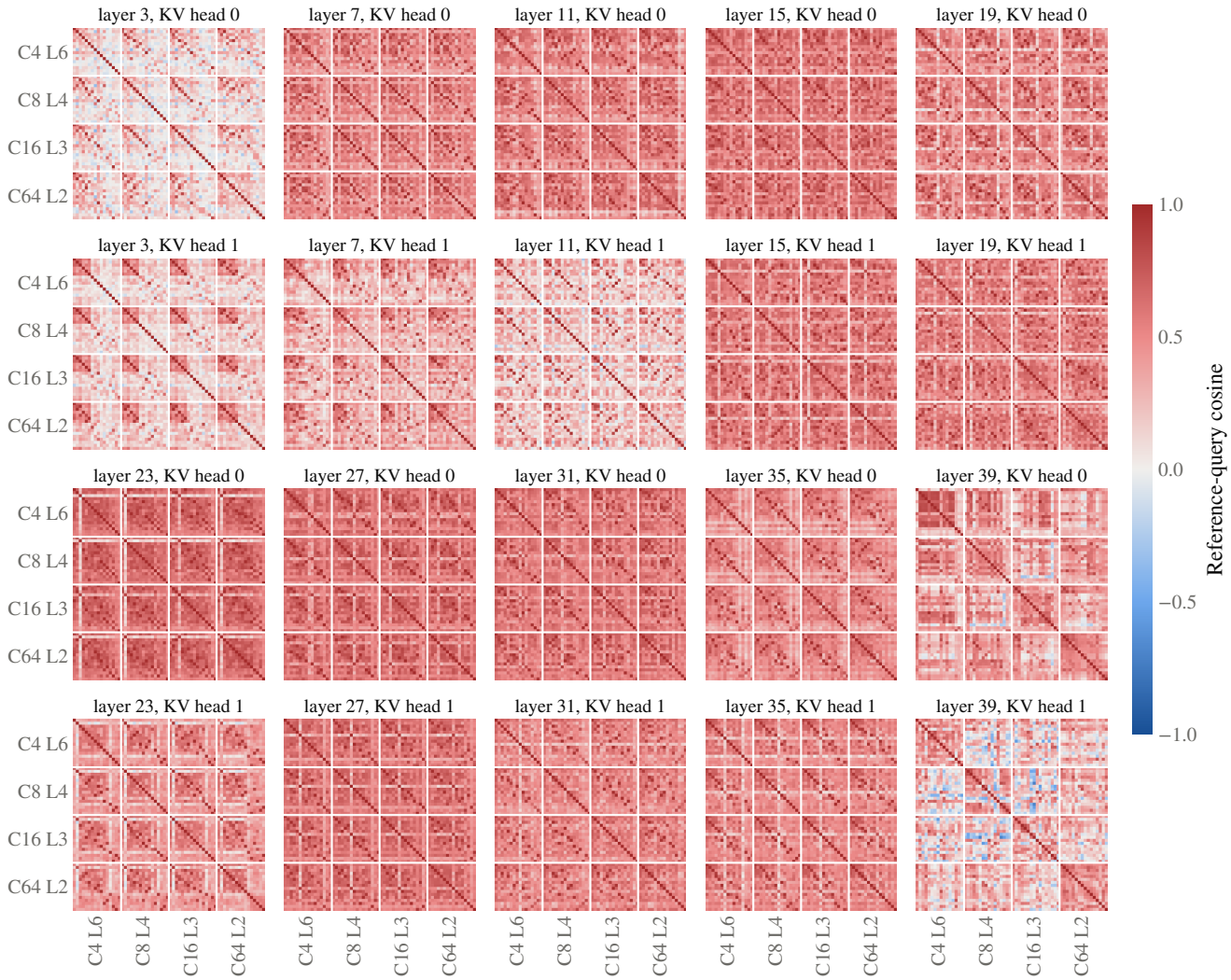


Figure 31: Reference-query cosine similarities with references independently ordered by top-level selection frequency within each geometry and KV group, most to least frequent. Frequencies were measured during routed prefill on 16 RULER prompts at 64K tokens and $p = 0.9$. No cross-geometry matching is applied; diagonal structure therefore indicates correspondence in both direction and selection rank.

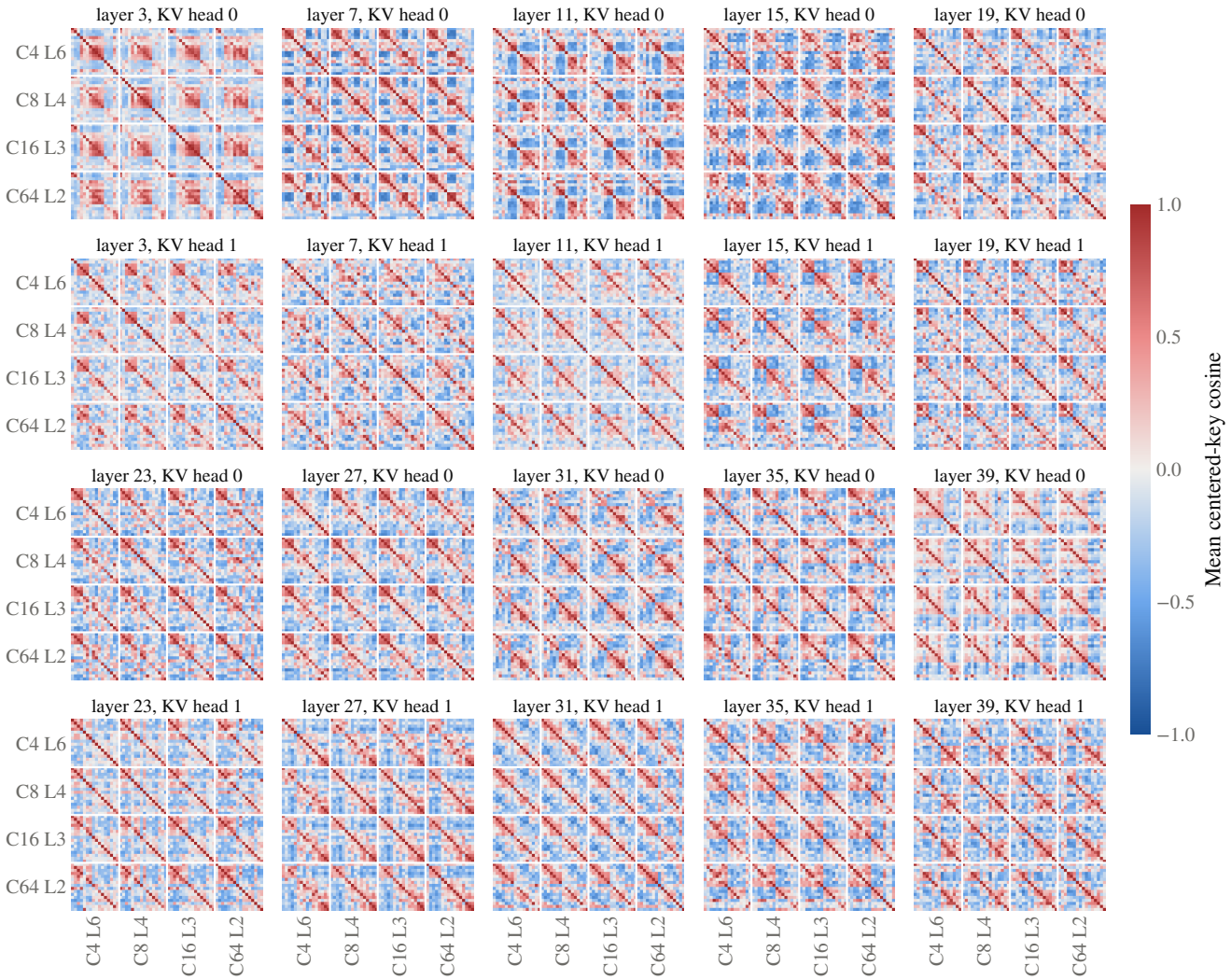


Figure 32: Mean cosine similarities of centered reference keys across geometries, evaluated on the same 60 top-level nodes from four RULER prompts at 64K tokens. Each node covers 4096 keys. Within each node and KV group, the mean over the 16 reference keys is subtracted before computing cosine similarities, which are then averaged over nodes. C16 L3 references are clustered using reference-key cosine similarity; other geometries are Hungarian-matched using the same measure. The cross-geometry correspondence extends to the pooled key representations.