Q-Bimodal Attention: Prefix and Tail Dispatch for Efficient Tree-Structured LLM Decoding

Anonymous ACL submission

Abstract

Tree-structured decoding workloads generate many continuations from a shared context, causing redundant KV-prefix reads and inefficient attention execution. We observe that tree attention is query-bimodal: shared prefixes favor wide prefill-style kernels, while branch-specific tails favor decode-native execution. We propose a fixed-cardinality dispatcher that selects between two execution strategies: a fused prefill-style tail strategy and a decode-native tail strategy, using a calibrated cost model. This keeps the dispatch space small while adapting to prefix length, batch size, and tail shape. On H100 GPUs with Llama-3.2-1B, Llama-3.1-8B and Llama-3-70B, our method consistently outperforms paged decoding, fixed shared-prefix execution, and existing dynamic tree-attention planning, achieving up to 6.74× over PagedAttention and up to 2.73× over the strongest competing baseline.

1 Introduction

Tree-structured generation is a common pattern in modern LLM inference. Beam search (Vijayakumar et al., 2018), speculative decoding (Miao et al., 2024; Li et al., 2025), few-shot learning (Brown et al., 2020), multi-step reasoning (Yao et al., 2023; Hao et al., 2023) all generate multiple continuations from a shared context. Although these workloads differ in how branches are produced, they induce the same attention-time structure: a long shared prefix that fans out into many branch-specific continuations. If each branch is served as an independent request, every branch repeatedly rereads the same shared KV prefix from HBM, even though the prefix is identical across branches.

Prior systems reduce this redundancy in different ways, as summarized in Table 1, where K denotes the number of leaves in the tree. PagedAttention (Kwon et al., 2023) improves KV-cache management, but does not share prefix reads across branches. DeFT (Yao et al., 2025) and Cascade Attention (Ye et al., 2025) reuse prefixes across multiple KV spans, but their execution structure is fixed rather than selected dynamically for each decoding step. FastTree (Pan et al., 2025) dynamically plans over tree-structured KV caches, but its search space grows with the input tree. Moreover, it executes all levels using a prefill-style kernel template, including branch-specific tails that are better suited to decode-style execution. As shown in Figure 1, this design forces tail queries to use the same tile size as shared-prefix queries. Because GPU kernels require fixed tile sizes, the small query counts in tail regions lead to substantial query-tile padding waste.

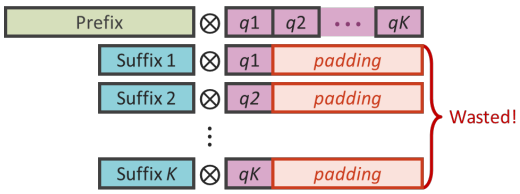

Figure 1: Poor query padding for single kernel tree attention.

We observe that tree attention is query-bimodal: shared-prefix attention favors wide prefill-style execution, while branch-specific tail attention favors decode-native execution. To address this mismatch, we propose Q-Bimodal Tree Attention, which dynamically analyzes the tree structure to select among a small set of execution strategies and launches a dedicated decode kernel when doing so is more efficient.

Our contributions are as follows:

- We identify query-shape bimodality as a central bottleneck in tree-structured attention: shared prefixes require wide query tiles, while branch-specific tails require decode-native execution.

- We show that a dedicated decoding kernel can remove tail-side query-tile padding, but that it is not universally faster because it introduces launch, merge, and occupancy overheads.

- We design a tree-attention kernel with query-bimodal runtime dispatch, achieving best performance across diverse decoding scenarios.

2 Method

We propose a fixed-cardinality dispatcher for tree-structured attention. The goal is to exploit KV sharing across branches while balancing between query tile padding overhead and extra decoding kernel launch overhead. At each step, the dispatcher summarizes the current batch, evaluates a small set of execution strategies, and launches the strategy with the lowest predicted cost.

2.1 Problem Setup

We consider a batch of prompts, each with K active branches. At each decoding step, every branch generates one query token. Branches may share a common prefix, may share intermediate KV spans, and eventually diverge into branch-specific tails.

2.2 Dispatch Space

Instead of searching over all nodes of the KV-sharing tree, we restrict the runtime to two strategies:

$$\mathcal{S} = \{s_{2\mathrm{f}}, s_{2\mathrm{d}}\}. \tag{1}$$

Here, $s_{2\mathrm{f}}$ is two-level shared-prefix execution with a fused prefill-style tail, $s_{2\mathrm{d}}$ is two-level shared-prefix execution with a decode-native tail.

2.3 Cost Model

At every decoding step t with work W_t, the dispatcher selects the strategy with minimum predicted cost:

$$s_t^\star = \arg\min_{s \in \mathcal{S}} C_s(W_t; \theta). \tag{2}$$

where θ is a set of device-calibrated constants. For fused shared-prefix strategy, the cost contains an effective bandwidth term and a wave-quantized tile term:

$$C_{\mathrm{fused}}(s) = B_s/\beta + \left\lceil \frac{M_s^{\mathrm{pref}}}{N_{\mathrm{SM}}} \right\rceil \tau + \lceil \log_2 D_s \rceil \, \tau_{\mathrm{merge}}. \tag{3}$$

Here, $B_s = \sum_{\ell=1}^{D_s} b_\ell$ is the total KV bytes loaded across the cascade levels of strategy s, β is calibrated HBM bandwidth, M_s^{pref} is the number of prefix tiles, N_{SM} is the number of GPU SMs, and τ is calibrated time per tile. The final term models the parallel merge over the D_s cascade levels. For the two-level fused strategy, this term contributes one merge round. Deeper cascade variants follow the same formula but are only considered in the ablation.

For the decode-tail strategy, the prefix is costed as a fused shared-prefix level, while the tail is costed with a decode-native model that includes the fixed overhead of launching the separate decode-tail kernel:

$$C_{\mathrm{dt}} = C_{\mathrm{prefix}} + \tau_{\mathrm{launch}}^{\mathrm{dt}} + \frac{B_{\mathrm{tail}}}{\beta} + \left\lceil \frac{M_s^{\mathrm{tail}}}{N_{\mathrm{SM}}} \right\rceil \tau_{\mathrm{tail}}. \tag{4}$$

Here, B_{tail} is tail KV traffic, M_s^{tail} is the number of decode-tail work tiles, τ_{tail} is calibrated decode-kernel work, and $\tau_{\mathrm{launch}}^{\mathrm{dt}}$ is the calibrated fixed overhead of launching and entering the separate decode-tail kernel. The calibration takes around 3 minutes on a H100 GPU per device.

2.4 Planning Complexity

The dispatcher evaluates a constant number of strategies and caps the exposed cascade depth at $D_{\max} = 2$. Therefore, planning cost is

$$O(|\mathcal{S}|\, K D_{\max}) = O(K). \tag{5}$$

This bound is independent of the number of nodes in the underlying KV-sharing tree. Deeper sharing structure, if present, is folded into the branch-tail term rather than introducing additional planner states.

3 Experiments

3.1 Experiment Setup

We evaluate our dispatcher on tree-structured decoding workloads using Llama-3.2-1B, Llama-3.1-8B on 1 H100 GPU, and Llama-3-70B on 4 H100 GPUs with tensor parallelism. We refer to FlashInfer's multi-level cascade attention baseline as MLCA. We compare against PagedAttention, MLCA, FastTree, and DeFT. We use the publicly released implementations of DeFT and FastTree, both of which report H100 results in their original papers, and use FlashInfer's production-ready MLCA kernel. Our kernel is implemented using FlashInfer's attention template, so it shares

Table 1: Comparison of different methods in terms of prefix sharing, dynamic dispatch, decision space, plan complexity, and Q-Bimodal support.

Method	Share Prefix	Dynamic Dispatch	Decision Space	Plan Complexity	Q-Bimodal
Paged	no	no	$O(1)$	–	-
MLCA	yes	no	$O(1)$	–	no
DeFT	yes	no	$O(1)$	–	no
FastTree	yes	yes	$O(2^K)$	$O(K \log K)$	no
Ours	yes	yes	$O(1)$	$O(K)$	yes

the same underlying attention-kernel infrastructure. Additional details are in Appendix A.

3.2 Scenarios

We evaluate four tree-structured decoding scenarios that cover different prefix lengths, batch sizes, and tree shapes. Here, K denotes the number of leaves in the decoding tree.

multi-doc-qa (Yang et al., 2018; Chan et al., 2025) represents multi-document question answering workloads, where multiple questions are asked over several concatenated documents. This setting has a very long shared prefix, up to 80K tokens, a small batch size of $B = 1$, and a large number of branches, $K = 128$, and an output length of 256 tokens.

multi-few-shot (Brown et al., 2020) represents few-shot prompting workloads, where the prompt contains multiple task demonstrations and the model generates outputs for multiple tasks. We use a relatively short shared prefix of 4K tokens, batch size $B = 16$ for the 1B model and $B = 4$ for the 8B and 70B models, $K = 128$ branches, and an output length of 256 tokens.

self-consistency (Wang et al., 2023) represents parallel sampling workloads, where a single prompt is expanded into many independent candidate continuations that are aggregated after decoding—for example, majority voting over chain-of-thought samples, or best-of-N selection with a reward model or verifier. Unlike the multi-query scenarios above, the K branches all share one common prefix and never re-branch, forming a *static* two-level tree: a single shared context fanning out into K parallel tails. This setting has a moderate prefix of 32K tokens for 1B, 16K tokens for 8B, a small batch size of $B = 1$, and $K = 16$ branches. To create a long decode stress case, we generate 12K output tokens.

beam-search (Vijayakumar et al., 2018) represents a dynamic decoding workload in which the output expands into a tree according to accumulated logit scores. The goal is to explore multiple hypotheses and find a higher-scoring continuation. Unlike the previous three scenarios, the tree structure in beam search is multi-level and difficult to predict in advance, making it the most challenging setting for tree attention. We use a 16K-token prefix, a moderate batch size of $B = 16$ for 1B, $B = 8$ for 8B and 70B models, $K = 64$ branches, and an output length of 256 tokens.

3.3 Results

Figure 2 shows the end-to-end model execution performance across all evaluated scenarios. Our method achieves the best performance in every setting and for all evaluated model sizes. Compared with PagedAttention, our method delivers substantial speedups, ranging from $1.06\times$ to $6.74\times$, demonstrating the benefit of exploiting shared prefixes and tree structure during decoding.

FastTree and MLCA are the strongest competing baselines, with the better method varying across settings. Even when comparing against the stronger of these two baselines for each configuration, our method consistently delivers speedups across all settings, ranging from 1.09× to 2.73×. This demonstrates the robustness and effectiveness of the Q-Bimodal design.

The speedup varies with workload structure and model configuration. The 1B and 8B models, both with a GQA ratio of 4, achieve higher speedups than the 70B model, which has a GQA ratio of 8. This is expected, since models with a higher GQA ratio have a smaller per-query KV footprint. Workloads with longer shared prefixes, larger batch sizes, or more branches expose greater redundancy in PagedAttention's memory traffic, enabling our method to deliver larger performance gains.

Table 2 further breaks down execution time across planning and forward computation. The dispatcher keeps planning overhead low, while the Q-Bimodal kernel substantially reduces forward time compared with all baselines.

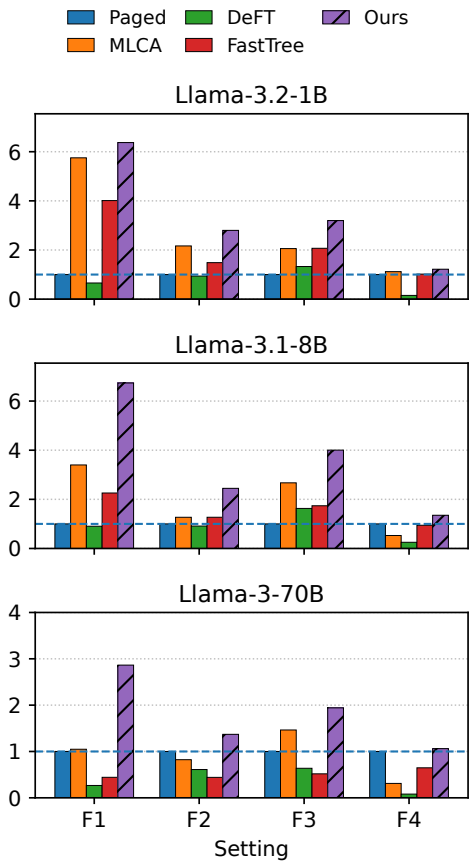

Figure 2: TPOT speedup over PagedAttention. PagedAttention is normalized to $1.0\times$ for each setting, and higher is better. F1 is multi-document QA, F2 is multi-few-shot, F3 is beam-search, and F4 is self-consistency.

Table 2: Timing breakdown for Llama-3.1-8B on beam-search. Build metadata and decision time capture CPU overhead, with decision time corresponding to kernel-strategy selection. Forward time is the model forward-pass GPU computation time. Numbers are reported in ms.

Method	**Build Metadata**	**Decision**	**Forward**
DeFT	1,533	0	21,382
MLCA	1,417	0	12,493
Ours	1,167	93	7,717
FastTree	1,962	786	19,083
Paged	960	0	37,215

Model	Scenario	s_{2f}-only	s_{2d}-only	Dispatcher	s_{2f}/s_{2d}
1B	F1	9,929	2,241	2,243	s_{2d}
	F2	9,152	7,757	7,791	s_{2d}
	F3	9,862	8,731	8,760	s_{2d}
	F4	62,819	61,451	59,637	35%/65%
8B	F1	29,091	6,609	6,619	s_{2d}
	F2	8,471	6,829	6,850	s_{2d}
	F3	12,971	9,871	9,909	s_{2d}
	F4	169,475	160,849	130,142	17%/83%
70B	F1	52,139	11,317	11,348	s_{2d}
	F2	16,398	14,330	14,340	s_{2d}
	F3	26,309	18,054	18,158	s_{2d}
	F4	377,826	406,988	378,352	72%/28%

Table 3: Dispatcher diagnosis across models and scenarios. Lower is better. The s_{2f}-only and s_{2d}-only columns report forced static execution with a single strategy. F1 = multi-doc-qa, F2 = multi-few-shot, F3 = beam-search, and F4 = self-consistency. Numbers are reported in ms.

3.4 Dispatcher Analysis

We analyze the execution plans selected by the runtime dispatcher. For each workload, we compare the adaptive dispatch policy against runs in which a single dispatch strategy is forced throughout the entire decode. We use the fastest forced strategy as a static oracle. Although the true optimum may vary across decode steps, exhaustively searching over all per-step strategy sequences is prohibitively expensive. The best static policy therefore serves as a practical and strong oracle baseline.

Table 3 reports the dispatcher diagnosis results. The self-consistency benchmark highlights the benefits of dynamic dispatch. Early in decoding, when tail work is limited, the dispatcher selects s_{2f} to avoid the overhead of launching a separate decode-tail kernel. As the tails become longer, it switches to s_{2d}, where decode-native tail execution becomes more efficient. This adaptive behavior allows the dispatcher to outperform both static strategies for the 1B and 8B models, demonstrating that no single static execution strategy is uniformly optimal across all decode steps.

Overall, for workloads whose optimal strategy remains fixed across steps, the dispatcher successfully selects the best strategy with minimal runtime overhead, typically less than 1%. For workloads whose optimal strategy changes over time, the dispatcher can outperform static strategies by switching strategies during decoding.

4 Conclusion

We presented Q-Bimodal Tree Attention, which separates shared-prefix execution from branch-specific tail execution for efficient tree-structured decoding. By combining prefix sharing with decode-native tail execution, our method reduces redundant KV reads and tail-side query padding. Experiments across diverse decoding workloads show consistent speedups over strong baselines, demonstrating that query-bimodal execution is a key design principle for efficient tree-structured LLM decoding.

Limitations

Our cost model is a simple estimator rather than an exact runtime predictor. It is designed to provide a good-enough ranking of execution strategies with low decision overhead, not to precisely estimate the latency of each strategy.

Our dispatcher also covers only a fixed set of strategies. One can certainly construct tree shapes for which these fixed strategies are not optimal. However, for recent tree-structured decoding applications, this restricted strategy set is already sufficient and performs well in practice.

References

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, and 12 others. 2020. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA. Curran Associates Inc.

Brian J. Chan, Chao-Ting Chen, Jui-Hung Cheng, and Hen-Hsen Huang. 2025. Don't do rag: When cache-augmented generation is all you need for knowledge tasks. In *Companion Proceedings of the ACM on Web Conference 2025*, WWW '25, page 893–897, New York, NY, USA. Association for Computing Machinery.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Jiahua Hong, Zhen Wang, Daisy Zhe Wang, and Zhiting Hu. 2023. Reasoning with language model is planning with world model. *Preprint*, arXiv:2305.14992.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA. Association for Computing Machinery.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2025. EAGLE-3: Scaling up inference acceleration of large language models via training-time test. In *Annual Conference on Neural Information Processing Systems*.

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. 2024. Specinfer: Accelerating large language model serving with tree-based speculative inference and verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 932–949, New York, NY, USA. Association for Computing Machinery.

Zaifeng Pan, Yitong Ding, Yue Guan, Zheng Wang, Zhongkai Yu, Xulong Tang, Yida Wang, and Yufei Ding. 2025. Fasttree: Optimizing attention kernel and runtime for tree-structured llm inference. In *Proceedings of Machine Learning and Systems*.

Ashwin K Vijayakumar, Michael Cogswell, Ramprasath R. Selvaraju, Qing Sun, Stefan Lee, David Crandall, and Dhruv Batra. 2018. Diverse beam search: Decoding diverse solutions from neural sequence models. *Preprint*, arXiv:1610.02424.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. *Preprint*, arXiv:2203.11171.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. 2018. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Jinwei Yao, Kaiqi Chen, Kexun Zhang, Jiaxuan You, Binhang Yuan, Zeke Wang, and Tao Lin. 2025. Deft: Decoding with flash tree-attention for efficient tree-structured llm inference. In *The Thirteenth International Conference on Learning Representations*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *Preprint*, arXiv:2305.10601.

Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. 2025. Flashinfer: Efficient and customizable attention engine for llm inference serving. *arXiv preprint arXiv:2501.01005*.

A Baseline and Implementations

Hardware and software. All methods run within the same driver, using the same paged KV allocator (page size 16, refcounted copy-on-write) and the same standalone Llama implementation. Thus, weight loading, RMSNorm, RoPE, and projections are identical across methods. The code is implemented in Python 3.11 with PyTorch 2.9 and the FlashInfer attention library. Methods differ only in the per-step attention call.

We build a radix tree for all methods and apply LCA cache optimization to shorten the index-building process. For FastTree (Pan et al., 2025) and DeFT (Yao et al., 2025), we use the published attention kernels directly from the authors' public artifacts and leave them unchanged.

B Query-Tile Padding Overhead of Fused Tree Attention

The only structural difference between $s_{2\mathrm{f}}$ and $s_{2\mathrm{d}}$ is how they execute the branch-specific tail level. Strategy $s_{2\mathrm{f}}$ folds the tail into the same fused prefill-style kernel used for the shared prefix, whereas $s_{2\mathrm{d}}$ executes the tail with a decode-native kernel and combines the prefix and tail results through an online-softmax bridge. This appendix isolates the cost of that choice: the query-tile padding incurred by $s_{2\mathrm{f}}$.

Source of the overhead. A prefill-style kernel processes queries in fixed-width tiles of size T (the kernel's `CTA_TILE_Q`). As in the per-tile cost model in Section 2, each tile performs the full score computation $\mathbf{QK}^\top$ and value aggregation $\mathbf{PV}$ over its entire KV span, regardless of how many query rows are actually occupied. Empty rows are masked at writeback, but the tensor-core work is still determined by the tile width T, not by the number of valid rows.

At a decode step, each branch contributes only G real query rows: one newly generated token folded over its GQA group of size $G = n_{\mathrm{qo}}/n_{\mathrm{kv}}$. Nevertheless, $s_{2\mathrm{f}}$ pads every tail tile to width T. The MMA utilization of a tail tile is therefore

$$U = \frac{G}{T}, \qquad \text{wasted fraction} = 1 - \frac{G}{T}. \tag{6}$$

Because this wasted work is repeated across the tail KV span, the penalty grows as branch tails become longer during decoding. In contrast, the decode-native tail in $s_{2\mathrm{d}}$ processes exactly the G rows per branch (`CTA_Q=1` with GQA fold-in) and therefore incurs no query padding.

Padding fraction. Since $U = G/T$ is determined entirely by the model's GQA ratio and the launched query-tile width, the per-tile padding fraction is exact. Strategy $s_{2\mathrm{f}}$ uses a single fused kernel pool, so tail rows are launched with the same large tile widths used for shared-prefix execution, $T \in \{64, 128\}$; even the smallest prefill-style tile that a fused shared-prefix engine would use for tails is $T = 16$. Table 4 reports the resulting waste.

Even under the most favorable prefill tile, $T = 16$, $s_{2\mathrm{f}}$ discards 50–75% of its tail MMA cycles. At the tile widths it actually launches, $T \in \{64, 128\}$, the wasted fraction exceeds 87%. We confirmed this behavior empirically by instrumenting the launched tiles over a 127-step decode on Llama-3.2-1B with $K = 32$, $B = 32$, and $L_p = 8192$: a prefill-style tail tile with $T = 16$ records a 75% query-padding fraction at every step, exactly matching $1 - G/T$, while the decode-native tail records 0% by construction.

Table 4: Tail-tile query-padding fraction $1 - G/T$ for the fused prefill-style tail of $s_{2\mathrm{f}}$, by GQA group size G and launched tile width T. The decode-native tail of $s_{2\mathrm{d}}$ has 0% padding for every entry.

Model	G	$T = 16$	$T = 64$	$T = 128$
1B	4	75.0%	93.75%	96.88%
8B	4	75.0%	93.75%	96.88%
70B	8	50.0%	87.50%	93.75%

End-to-end effect. The padding overhead is most visible when there is a lot of tail work, so that tail MMA takes up meaningful time. The multi-doc-qa scenario (F1 in Table 3) falls in this regime: across all three models, forced $s_{2\mathrm{f}}$ execution is 4.4–4.6× slower than forced $s_{2\mathrm{d}}$ execution.

In the opposite regime, where there is little tail work, s2f can be competitive. For example, on F2 with the 8B model, $s_{2\mathrm{f}}$ is within 1.24× of $s_{2\mathrm{d}}$. This crossover is exactly what the dispatcher exploits: it selects $s_{2\mathrm{f}}$ while tails are short and the extra decode-kernel launch in $s_{2\mathrm{d}}$ is not yet worthwhile, then switches to $s_{2\mathrm{d}}$ once tails grow long enough for the padding tax to dominate.

C Dispatch Space Ablation

We next ask whether additional dispatch strategies for deeper sharing levels are necessary. This ablation uses the Llama-3.1-8B long-decode beam-search setting with $K = 32$, $L_p = 8192$, $B = 8$, and runs for 2000 decode steps on 2 H100 GPUs. We force the dispatcher to use deeper variants and measure only decode steps 1500–2000, where the beam-search tree is most likely to contain longer intermediate shared prefixes that deeper strategies could exploit. These deeper variants are not part of the runtime dispatch space in Section 2.2; we evaluate them only as an offline ablation to justify removing deeper strategies.

Table 5: Depth ablation for Llama-3.1-8B with TP=2. Forward time is measured over decode steps 1500–2000. The time change is reported relative to the corresponding two-level variant within each kernel family; positive values indicate slower execution.

Strategy	Fwd. time (ms)	Δ time vs 2L
s_{2d}	10,739	–
s_{3d}	11,651	+8.5%
s_{4d}	10,731	-0.1%
s_{5d}	10,999	+2.4%

The results in Table 5 show that increasing the number of fused cascade levels does not materially improve forward time, as more levels can only share shorter prefixes and bring more merge overhead.

D Calibration

The cost model in Section 2 is parameterized by a small set of device-calibrated constants $\theta = \{\beta, \tau, \tau_{\text{merge}}, \tau_{\text{tail}}, \tau^{\text{dt}}_{\text{launch}}\}$. We do *not* re-tune the attention kernels themselves: FlashInfer's JIT already specializes register allocation, shared-memory layout, and instruction selection per SM, so calibration only measures the behaviors the JIT cannot expose at decision time. Each constant is recovered by a short microbenchmark run once at engine initialization. The number of SMs N_{SM} is read directly from the device properties and is not timed. The full sweep takes roughly three minutes on an H100.

Attention bandwidth β. The terms B_s/β and B_{tail}/β in Equations 3 and 4 price each strategy's KV-load traffic, so β must reflect the bandwidth attention actually sustains under paged, strided KV reads—not the contiguous-copy rate, which overstates the effective bandwidth of strided access. We therefore measure β from the attention kernel itself: we run a large batched paged decode whose working set (several GiB) overflows the L2 cache, count the KV bytes physically read, and divide by the measured kernel time. This isolates the sustained HBM read rate from launch and cache-residency effects and recovers a rate close to the device's peak HBM bandwidth on H100.

Per-tile prefill cost τ. We time a single fused prefill call with T query rows against a fixed-length KV span, swept over the tile widths the kernel JIT-compiles, $T \in \{16, 64, 128\}$. This yields the per-tile term used by the wave-quantized prefix cost.

Decode-tail cost τ_{tail}. We time a paged-decode call at one query per branch ($\text{CTA}_Q = 1$) against KV spans of varying length, and fit the per-token decode work from how latency grows with the KV tokens loaded. This yields the τ_{tail} term used by the wave-quantized decode-tail cost in Equation 4.

Merge cost τ_{merge}. The per-round merge cost τ_{merge} in Equation 3 is calibrated as a single constant, measured from the latency of one `merge_state_in_place` call. A depth-D_s cascade combines its levels in $\lceil \log_2 D_s \rceil$ parallel merge rounds along the critical path, matching the final term of Equation 3.

Decode-tail launch overhead $\tau^{\text{dt}}_{\text{launch}}$. We measure the fixed overhead of the separate decode-tail launch by timing a decode-native tail call on the smallest valid KV span and subtracting the bandwidth and work-tile terms predicted by the calibrated β and τ_{tail}. This residual captures the kernel-launch and kernel-entry overhead that is paid whenever s_{2d} uses a separate decode-tail kernel, independent of tail length.

E Absolute Results

Table 6 shows the absolute numbers for Figure 2, demonstrating the effectiveness of our method.

F Reproduction

Models and precision. Llama-3.2-1B-Instruct and Llama-3.1-8B-Instruct are served with `bf16` weights and a `bf16` KV cache on a single H100 (no tensor parallelism). Llama-3-70B-Instruct is served with `bf16` weights and a `bf16` KV cache under tensor parallelism (TP$=4$) across four H100s. All checkpoints are the public instruction-tuned releases. The whole set takes around 2 full days to run.

Workload data. The four scenarios are built from two public datasets, GSM8K and HotpotQA (Yang et al., 2018), using real text rather than synthetic tokens. We sample from the GSM8K `test` split (1,319 problems) and the HotpotQA `distractor validation` split (7,405 examples, ten paragraphs each). *multi-doc-qa* (F1) concatenates $\sim$575 HotpotQA documents into an $\sim$80K-token shared prefix and appends K=128 distinct HotpotQA questions. *multi-few-shot* (F2) and *self-consistency* (F4) use GSM8K few-shot exemplars

Table 6: End-to-end decode results. Values are median `decode_total_ms`; lower is better. Error bars report half-range over measured repeats. Bold entries denote the row winner among the displayed methods.

Model	Scenario	Paged	FastTree	DeFT	MLCA	Ours
Llama-3.2-1B	multi-doc-qa	14293±0.0%	3563±0.1%	21755±0.0%	2485±0.0%	**2243±0.2%**
	multi-few-shot	21805±0.1%	14682±0.3%	23237±0.0%	10070±0.2%	**7791±0.2%**
	beam-search	28447±0.3%	14058±0.8%	21300±0.4%	13877±0.5%	**8760±0.3%**
	self-consistency	72659±0.1%	71564±0.4%	479295±0.0%	64982±0.1%	**59637±0.1%**
Llama-3.1-8B	multi-doc-qa	44584±0.1%	19745±0.1%	49263±0.0%	13115±0.0%	**6619±0.2%**
	multi-few-shot	16768±0.1%	13195±0.3%	18371±0.1%	13204±0.2%	**6850±0.1%**
	beam-search	39580±0.3%	22943±0.6%	24355±0.1%	14727±0.0%	**9909±0.4%**
	self-consistency	175956±0.0%	186359±0.1%	699269±0.1%	333107±0.0%	**130142±0.0%**
Llama-3-70B-Instruct	multi-doc-qa	32502±0.0%	73429±0.0%	121912±0.1%	31004±0.0%	**11348±0.1%**
	multi-few-shot	19649±0.1%	44519±0.1%	32183±0.0%	23878±0.1%	**14340±0.1%**
	beam-search	35252±0.8%	68110±0.4%	55292±0.4%	24079±0.3%	**18158±0.8%**
	self-consistency	403123±0.2%	621572±0.1%	5003149±0.1%	1295019±0.3%	**378352±1.0%**

as the shared prefix; F2 packs 20 exemplars plus $K{=}128$ questions per bundle (up to the full 1,319-problem split). *beam-search* (F3) uses HotpotQA prompts as the prefix and grows the $B{\times}K$ beam tree online from accumulated logit scores, so its multi-level sharing structure is data-dependent and not fixed in advance. The GSM8K test split is downloaded on first use; the HotpotQA prefix files are built by released preprocessing scripts.

Timing protocol. For every (scenario, method) cell we run a warmup iteration, discard it, and report the median over 5 measured repeats; the CUDA cache is cleared and garbage collection is run between repeats. We report total decode time over all steps, split into *plan* time (CPU-side metadata build plus strategy decision) and *forward* time (GPU compute). F1–F3 decode 256 tokens per branch; F4 is the long-decode stress cell at 12,000 tokens. The speedups in Figure 2 are time-per-output-token (TPOT) relative to PagedAttention in the same cell.

Calibration and determinism. The cost-model constants θ are recalibrated once per device at engine initialization (Appendix D); the sweep takes roughly three minutes. Every method shares the same paged allocator and model implementation, so the reported differences reflect only attention-execution time.

G Single-Step Strategy-Ranking Validation

The cost model is designed to select an execution strategy rather than predict absolute latency. Since many terms common to all strategies cancel out or are absorbed into calibrated constants, we evaluate the model by its decision quality: whether it selects the fastest available strategy for a given tree-attention shape.

We use a single-step tree-kernel microbenchmark that isolates the attention strategy decision from full model execution. Each benchmark instance represents one decode attention step over a balanced prefix-sharing tree. The benchmark uses random KV tensors and does not load model weights. All cells are timed on H100 using fp16, CUDA events, 5 warmup iterations, and 30 measured iterations. We report the median GPU time in milliseconds.

Since our cost model is able to measure different cascade attention depths, we add a new candidate strategy s_{pb} , which is simply PagedAttention, and is selected when the cost model finds that prefix sharing is not beneficial.

For each shape, we measure all forced strategies in the candidate set and define the oracle as the fastest measured strategy among them. Regret is the measured time of the cost-model picker divided by the oracle time. Ratios slightly below 1.0 are due to run-to-run noise because the picker and forced strategies are timed in separate runs.

Tree shapes. We use a compact taxonomy of prefix-sharing structures. The flat-beam shapes contain one shared root prefix and multiple short per-beam tails; they vary root length and fan-out. The batched-beam and multi-chain shapes contain many independent prompts, each with its own shared prefix and a small number of branches, testing the high-occupancy batched regime. The diverse-groups shapes include an intermediate shared suffix before the final leaves, testing whether the two-level strategies remain effective on deeper trees. The long-decode shape places the timed step

after the per-branch tail has grown large, probing the crossover between fused and decode-tail execution. The adversarial shape uses a tiny shared prefix, where sharing should not pay off. The wide-fanout shapes use a fixed root prefix with many leaves, stressing prefix reuse and tail-side padding.

Table 7 shows that the picker matches the measured oracle on 15 of the 16 evaluated shapes. The only mismatch occurs on `dbs-g4-k6`, where the selected $s_{2\mathrm{d}}$ and oracle $s_{2\mathrm{f}}$ latencies are very close. The oracle is not constant: batched or narrow shapes often favor $s_{2\mathrm{f}}$, wide-fanout shapes favor $s_{2\mathrm{d}}$, and the tiny-prefix adversarial shape favors s_{pb}. This confirms that the dispatch decision depends on the tree structure and that no single fixed strategy is optimal across the evaluated space.

Across all shapes, the picker achieves a $2.58\times$ geometric-mean speedup over s_{pb} and a $1.004\times$ geometric-mean regret relative to the measured oracle. The remaining mismatch is acceptable because the competing $s_{2\mathrm{f}}$ and $s_{2\mathrm{d}}$ measurements are nearly tied; at this level, effects such as hardware scheduling noise and other low-level execution details are difficult to model exactly. These results validate the intended use of the cost model: it need not predict absolute latency, but it must provide a good enough ranking to select an appropriate kernel for each tree shape.

H Artifact Licenses and Terms of Use

We use publicly available software, models, datasets, and baseline artifacts whose licenses or terms permit research use. Our released artifact will contain our implementation, benchmark scripts, configuration files, and instructions needed to reproduce the experiments. Unless otherwise stated in the artifact repository, we release our own code under the Apache-2.0 license. The artifact does not redistribute third-party model weights, datasets, CUDA/NVIDIA software, or vendored dependencies whose licenses require users to obtain them separately.

Software dependencies. Our implementation is written in Python and uses PyTorch and FlashInfer as the main software dependencies. Python is distributed under the Python Software Foundation License Version 2. PyTorch is distributed under a permissive BSD-style license. FlashInfer is distributed under the Apache-2.0 license. For PagedAttention baselines, we rely on the public vLLM implementation, which is also distributed under the Apache-2.0 license. CUDA and NVIDIA GPU software are used subject to the NVIDIA CUDA Toolkit End User License Agreement and applicable NVIDIA driver/software license terms.

Models. The experiments use Llama-3.2-1B, Llama-3.1-8B, and Llama-3-70B. These models are governed respectively by the Llama 3.2 Community License Agreement, the Llama 3.1 Community License Agreement, and the Meta Llama 3 Community License Agreement, together with their Acceptable Use Policies. We do not redistribute the model weights in our artifact. Users who reproduce the experiments must obtain the models from the official Meta or Hugging Face distribution channels and agree to the applicable Llama license terms.

Datasets. The multi-document QA workload uses HotpotQA. The HotpotQA dataset is distributed under the CC BY-SA 4.0 license, while the accompanying HotpotQA code is distributed under the Apache-2.0 license.

Third-party research artifacts. We compare against public implementations of DeFT and FastTree. DeFT is released under the MIT license. The FastTree artifact for tree-structured LLM inference is released under the Apache-2.0 license. We use these artifacts only for benchmarking and baseline comparison, and we preserve their copyright notices, license files, and citation requirements. If our artifact includes wrapper code or integration scripts for these baselines, we document the original upstream repositories and licenses.

Use and distribution. The intended use of our artifact is academic research and reproducibility of the results reported in this paper. Users are responsible for complying with the licenses and terms of all third-party components, including software libraries, model weights, datasets, CUDA/NVIDIA software, and baseline artifacts. No additional restrictions are imposed by us beyond the license selected for our own released code and the terms of the third-party artifacts listed above.

Family	Shape	s_{pb}	MLCA	FastTree	Dispatcher	s_{2f}	s_{2d}	Picked	Oracle	Regret
flat-beam	flat-lp8k-k16	0.076	0.154	0.063	**0.043**	0.043	0.064	s_{2f}	s_{2f}	-
flat-beam	flat-lp32k-k16	0.272	0.495	0.169	**0.083**	0.091	0.085	s_{2d}	s_{2d}	-
flat-beam	flat-lp8k-k4	0.034	0.142	0.061	**0.031**	0.031	0.063	s_{2f}	s_{2f}	-
flat-beam	flat-lp8k-k64	0.257	0.188	0.217	**0.069**	0.268	0.068	s_{2d}	s_{2d}	-
batched-beam	bbeam-b20-lp8k-k16	1.366	0.904	0.424	**0.358**	0.358	0.359	s_{2f}	s_{2f}	-
batched-beam	bbeam-b8-lp16k-k32	2.186	0.712	1.827	**0.292**	0.557	0.292	s_{2d}	s_{2d}	-
diverse-groups	dbs-g4-k4	0.079	0.172	0.059	**0.049**	0.049	0.066	s_{2f}	s_{2f}	-
diverse-groups	dbs-g4-k5	0.128	0.175	0.081	**0.055**	0.055	0.072	s_{2f}	s_{2f}	-
diverse-groups	dbs-g4-k6	0.130	0.180	0.176	0.064	**0.059**	0.064	s_{2d}	s_{2f}	8%
diverse-groups	dbs-g4-k8	0.134	0.183	0.183	**0.064**	0.264	0.064	s_{2d}	s_{2d}	-
diverse-groups	dbs-g8-k8	0.265	0.216	0.224	**0.078**	0.279	0.078	s_{2d}	s_{2d}	-
multi-chain	chain-b32-c4	0.316	0.737	0.306	**0.256**	0.255	0.256	s_{2f}	s_{2f}	-
long-decode	ld-lp8k-tail2k	0.103	0.277	0.137	**0.090**	0.091	0.098	s_{2f}	s_{2f}	-
adversarial	adv-tiny-prefix	0.031	0.065	0.059	**0.031**	0.034	0.059	s_{pb}	s_{pb}	-
wide-fanout	wide-k128	0.515	0.243	0.407	**0.108**	0.292	0.110	s_{2d}	s_{2d}	-
wide-fanout	wide-k256	1.034	0.338	0.551	**0.186**	0.318	0.186	s_{2d}	s_{2d}	-

Table 7: Single-step strategy-ranking validation on curated tree-attention shapes. All values are median GPU time in milliseconds. The picker is the cost-model selected strategy. The oracle is the fastest measured strategy among s_{pb}, s_{2f}, and s_{2d}. MLCA and FastTree are external baselines and are not included in the oracle. Regret is reported only when the picked strategy differs from the oracle.