PALRAC: Parallel Linear Recurrence Accelerator for Tree-based Speculative Decoding

Sangheon Lee
Department of Electrical and Computer Engineering
Seoul National University
Seoul, Republic of Korea
leetkdgjs@capp.snu.ac.kr

Hyuk-Jae Lee✉
Department of Electrical and Computer Engineering
Seoul National University
Seoul, Republic of Korea
hyuk_jae_lee@capp.snu.ac.kr

Xuan Truong Nguyen
Department of Electrical and Computer Engineering
Seoul National University
Seoul, Republic of Korea
truongnx@capp.snu.ac.kr

Abstract

Linear recurrent models (LRMs) have emerged as efficient alternatives to Transformer-based large language models by replacing the attention mechanism with linear recurrence, enabling long-context inference with linear complexity and a constant memory footprint. Recent approaches further improve decoding throughput by combining LRMs with tree-based speculative decoding, where a target model verifies multiple draft sequences in a token tree in parallel. However, modern GPUs cannot fully exploit these algorithmic advantages due to the low arithmetic intensity of linear recurrence and the sparse computations induced by draft trees. This mismatch results in severe underutilization of GPU compute resources.

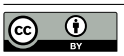
To address this inefficiency, we present PALRAC, a parallel linear recurrence accelerator for tree-based speculative decoding. PALRAC is built on a hardware-friendly formulation of linear recurrence for target model verification, consisting of parallel state update for verified tokens and tree-based output computation for draft tokens. Based on this formulation, PALRAC adopts a specialized microarchitecture with decoupled execution modules optimized for dense state computation and sparse interactions among draft tokens. To further improve efficiency under tight area and bandwidth constraints, we propose lazy state update and tree mask regularization, which reduce off-chip accesses and enable efficient on-chip data reuse for the sparse interactions, respectively. Our evaluation of linear recurrence operations shows that PALRAC achieves a geometric-mean speedup of 1.85× and a 16.2× improvement in energy efficiency over the GPU baseline, with a small area and power footprint. Moreover, integrated GPU–PALRAC simulation with micro-batch pipelining demonstrates a geometric-mean speedup of 1.40× in target model verification over GPU-only execution at a batch size of 32.

CCS Concepts

• **Computer systems organization → Neural networks**; **Special purpose systems**.

Keywords

Linear recurrent models, speculative decoding, hardware accelerator, linear recurrence, linear attention, LLM inference



This work is licensed under a Creative Commons Attribution 4.0 International License.
PACT '26, Chicago, IL, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2915-7/2026/10
https://doi.org/10.1145/3838684.3846866

ACM Reference Format:
Sangheon Lee, Hyuk-Jae Lee, and Xuan Truong Nguyen. 2026. PALRAC: Parallel Linear Recurrence Accelerator for Tree-based Speculative Decoding. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '26), October 19–22, 2026, Chicago, IL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3838684.3846866

1 Introduction

Large language models (LLMs) [4, 9] have demonstrated strong performance in various applications such as chatbots [1, 29] and code generation [6, 24]. Built on the Transformer architecture [34], these models mainly consist of *attention* layers for sequence modeling and *fully connected (FC)* layers for feature transformations. During inference, LLMs typically operate in two distinct phases: a *prefill stage* to capture the initial context and a *decode stage* to generate tokens sequentially. The prefill stage is compute-intensive because it processes all input tokens in parallel, whereas the decode stage requires substantial memory bandwidth due to repeated access to model parameters during autoregressive generation. As a result, accommodating these requirements necessitates the costly deployment of high-end GPUs [25, 26] and tensor processing units (TPUs) [13, 14] equipped with high-bandwidth memory (HBM) [15]. This highlights a strong need for cost-effective solutions for LLM inference from both algorithmic and architectural perspectives.

To overcome the high inference cost of Transformer-based LLMs, recent studies have explored two emerging trends: linear recurrent models (LRMs) [8, 17, 30, 32, 39] and speculative decoding [5, 20]. LRMs provide an efficient alternative to Transformers by replacing costly attention with a linear recurrence mechanism. Unlike Transformer-based LLMs with a growing KV cache [19], LRMs compress historical context into *a fixed-size recurrent state*, as illustrated in Figure 1(a), thereby achieving *linear computational complexity* and a *constant memory requirement* with respect to context length [8, 30, 32, 39]. Meanwhile, speculative decoding offers another opportunity to increase decoding throughput by enabling a target model to verify multiple draft tokens in parallel [5, 20, 36–38]. For more effective speculation, tree-based speculative decoding [21, 23] organizes draft tokens into a tree structure that represents multiple candidate sequences, as shown in Figure 1(b). Recent efforts [35–38, 41, 43] have integrated LRMs with speculative decoding to leverage both the computational efficiency of linear recurrence and the throughput benefit of parallel verification.

Modern GPUs, however, do not fully exploit the computational efficiency of linear recurrence in LRMs, especially during target model verification in tree-based speculative decoding. In particular,

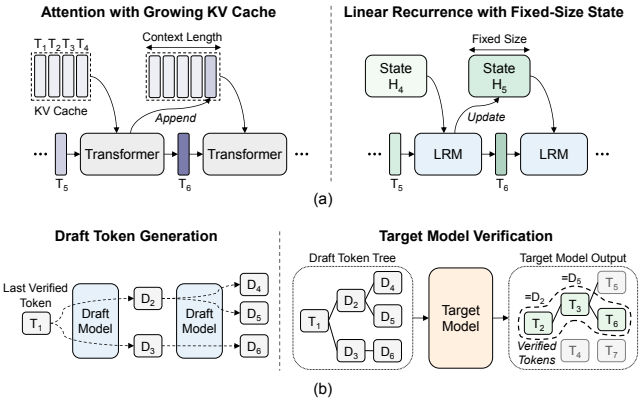


Figure 1: Two complementary approaches for efficient LLM inference: (a) LRMs with fixed-size recurrent states versus Transformers with growing KV caches, and (b) tree-based speculative decoding with draft token generation and parallel target model verification.

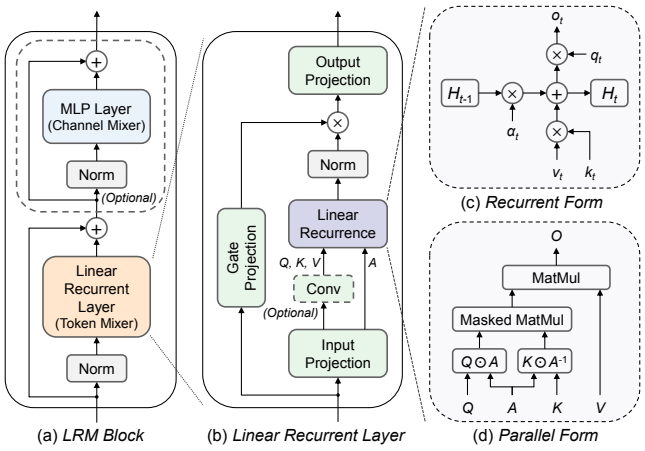


Figure 2: Generalized architecture of linear recurrent models (LRMs) highlighting the linear recurrent layer and its two equivalent formulations: the recurrent and parallel forms.

FC layers such as input and output projections map naturally to general matrix-matrix multiplication (GEMM) operations. Since speculative decoding combined with batching increases the arithmetic intensity of these GEMM workloads, they can effectively utilize GPU compute resources. In contrast, linear recurrence operations exhibit two distinct workload characteristics that hinder efficient GPU execution. First, they rely on element-wise operations over input vectors to compute recurrent states and outputs. Since these operations are performed on per-sequence recurrent states, their arithmetic intensity does not scale with batch size, leaving them fundamentally memory-bound even under speculative decoding. Second, tree-based speculative decoding introduces sparse token interactions because each draft token depends only on its ancestors in the draft tree rather than on all preceding tokens. Such sparsity leads to irregular memory access and computation patterns, whereas GPUs are optimized for dense matrix operations, resulting in severe underutilization of compute resources. These inefficiencies prevent linear recurrence operations from fully exploiting the massive computational throughput of modern GPUs, particularly their tensor cores, thereby motivating a specialized accelerator as a cost-effective alternative.

We present PALRAC, a parallel linear recurrence accelerator for tree-based speculative decoding. While prior accelerator efforts have targeted LRM inference with sequential verification of draft token trees (e.g., SpecMamba [43]), PALRAC is specifically designed to accelerate our parallel linear recurrence over draft trees. Our key contributions are as follows:

- **Tree-based Parallel Linear Recurrence.** We introduce a hardware-friendly formulation of linear recurrence for tree-based speculative decoding, which is decomposed into two stages: *parallel state update* for verified tokens and *tree-based output computation* for the draft token tree. This formulation eliminates redundant computation for intermediate recurrent states in both stages and exposes dense and sparse computation patterns amenable to efficient hardware acceleration.
- **PALRAC.** We present PALRAC, a parallel linear recurrence accelerator that implements the parallel formulation. PALRAC employs decoupled execution modules specialized for dense state computation and sparse draft-token interactions. To tackle the key design challenges of intensive off-chip memory traffic and irregular memory access patterns in draft token trees, PALRAC further incorporates *lazy state update* and *tree mask regularization*, reducing off-chip memory traffic and enabling efficient buffering with a reduced on-chip memory footprint, respectively.
- **Evaluation.** Experimental results on linear recurrence operations show that PALRAC with lazy state update achieves a geometric-mean speedup of 1.85× and a 16.2× improvement in energy efficiency compared to the GPU baseline. Lazy state update provides an additional 27.4% performance improvement over the base PALRAC design, while tree mask regularization reduces the total SRAM area by 35.7%. At the system level, integrated GPU–PALRAC execution with micro-batch pipelining delivers a geometric-mean speedup of 1.40× in target model verification over GPU-only execution at a batch size of 32.

2 Background and Related Work

2.1 Linear Recurrent Models

Linear recurrent models (LRMs), such as Mamba2 [8] and GLA [39], have emerged as efficient alternatives to Transformers by replacing the attention mechanism [34] with a linear recurrence. As illustrated in Figure 2(a), an LRM block consists of a linear recurrent layer as a token mixer and an optional MLP layer as a channel mixer. Similar to multi-head attention, the linear recurrent layer commonly adopts a multi-head structure, where multiple independent heads process the input sequence in parallel. Unlike attention, which explicitly computes pairwise query-key interactions between tokens, linear recurrence maintains a fixed-size recurrent state by accumulating the outer product of each token's key-value pair under data-dependent transitions. The linear recurrence output is then computed by applying the query to the recurrent state, allowing

the model to aggregate information from past tokens through the recurrent state rather than a token-wise KV cache. This state representation provides linear computational complexity and a constant memory footprint with respect to sequence length.

The mathematical duality of linear recurrence enables two equivalent execution forms: a recurrent form for fast autoregressive generation and a parallel form for efficient sequence processing [17].

Recurrent Form (Figure 2(c)). In the recurrent form, LRMs can be viewed as recurrent neural networks (RNNs). For a single linear recurrence head, let $q_t, k_t \in \mathbb{R}^{D_k}$ and $v_t \in \mathbb{R}^{D_v}$ denote the query, key, and value vectors at token index t, respectively, where D_k and D_v represent the per-head dimensions. Given a data-dependent diagonal transition vector $\alpha_t \in (0,1)^{D_k}$, the recurrent state $H_t \in \mathbb{R}^{D_k \times D_v}$ and recurrence output $o_t \in \mathbb{R}^{D_v}$ are defined as:

$$H_t = \mathrm{diag}(\alpha_t) H_{t-1} + k_t v_t^\top, \quad o_t = q_t^\top H_t \tag{1}$$

This recurrent form updates the fixed-size state in place and uses it to produce the output, enabling constant-time computation per token during autoregressive generation. However, it introduces a strong *sequential dependency*, as each state depends on the previous one.

Parallel Form (Figure 2(d)). To overcome the sequential dependencies of RNNs, LRMs exploit the associativity of linear operations to process sequences in parallel without materializing intermediate states [8, 32, 39]. Let $Q, K \in \mathbb{R}^{L \times D_k}$ and $V \in \mathbb{R}^{L \times D_v}$ denote the stacked input matrices for a sequence of length L. Using the cumulative transition matrix $A \in \mathbb{R}^{L \times D_k}$, where $A_t = \prod_{i=1}^{t} \alpha_i$, the parallel linear recurrence with a zero initial state is formulated as:

$$O = (M \odot (Q \odot A)(K \odot A^{-1})^\top) V \tag{2}$$

where $\odot$ denotes element-wise multiplication, A^{-1} represents the element-wise reciprocal of A, and M is the causal mask. Notably, Eq. (2) allows the recurrence output O to be computed through matrix multiplications, which can be accelerated on modern hardware such as GPU tensor cores.

2.2 Speculative Decoding for LRMs

To further improve LRM inference efficiency, recent studies have adopted speculative decoding [35–38, 41, 43]. Speculative decoding accelerates autoregressive generation by employing a lightweight draft model to propose multiple draft tokens, which are then verified in parallel by a larger target model [5, 20]. To increase the number of accepted tokens per verification, *tree-based speculative decoding* extends this paradigm by organizing draft tokens into a tree structure that incorporates multiple draft sequences [21, 23]. In Transformers, this *draft token tree* can be efficiently processed in a single forward pass using a topology-aware causal mask for attention [23].

Applying speculative decoding to LRMs, however, presents a fundamental challenge: *state backtracking* [37]. Because an LRM destructively updates its recurrent state at each token, the valid state must be recovered based on the verification results, i.e., the state corresponding to the accepted tokens. A naive approach would be to materialize all intermediate states in off-chip memory, but this would incur prohibitive memory costs. To address this, prior works [36, 37] recompute the valid state using the tokens verified in the previous iteration before processing new draft tokens. In addition, to reduce redundant computation for these verified tokens, an activation replay strategy [37] caches the activations of draft tokens for reuse upon verification. Based on these approaches to state backtracking, linear recurrence in target model verification can be divided into two stages: the state update stage for the verified tokens and the output computation stage for the draft token tree.

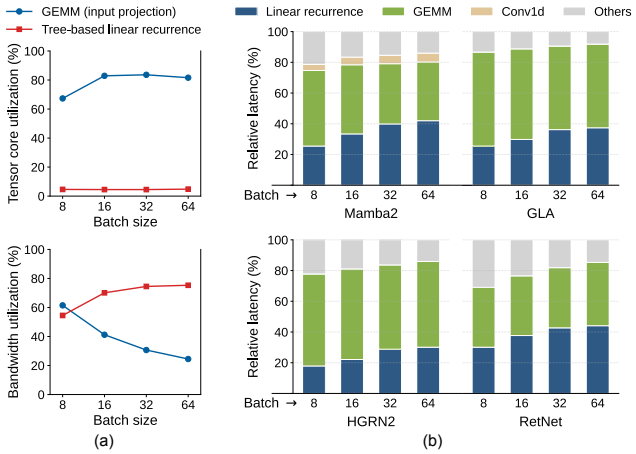


Figure 3: Characterization of target model verification for LRMs using 16-node draft token trees on an A100 GPU. (a) GPU resource utilization for Mamba2-2.7B. (b) Latency breakdown for various 2.7B LRMs across different batch sizes.

While the parallel form in Eq. (2) efficiently processes multiple tokens in parallel, extending it to the output computation stage of tree-based speculative decoding is non-trivial. Since each token has its own data-dependent transition, different paths in the draft tree induce different cumulative transitions, and therefore correspond to different recurrent states. Thus, SpecMamba [43] employs the recurrent form in Eq. (1) to process the draft token tree in the output computation stage, mitigating the storage overhead of intermediate recurrent states through tiled on-chip FIFO buffering. STree [38] enables parallel draft-tree processing for Mamba2 [8] by accumulating state transitions along the tree topology. However, its formulation cannot be directly applied to LRMs with general diagonal transition matrices.[1]

3 Motivation

3.1 Workload Characterization of Target Model Verification for LRMs

Speculative decoding improves generation throughput by enabling parallel target model verification. Specifically, FC layers benefit substantially from reusing shared weights across draft tokens and batched requests. However, linear recurrence offers limited data reuse across a batch, making its efficiency gains from parallel execution less pronounced. To quantify this gap, we analyze the execution characteristics of target model verification for LRMs during tree-based speculative decoding. Figure 3(a) presents the profiling results

[1] Although STree [38] assumes diagonal transition matrices, its derivation relies on commutativity between the transition matrix and an unstructured matrix. As this does not hold in general, its formulation is applicable only to LRMs with scalar transitions.

of STree [38] on Mamba2-2.7B using an A100 40GB GPU. The results reveal a clear contrast in tensor core utilization: GEMM operations achieve high utilization, whereas linear recurrence remains below 10% despite parallel processing of draft tokens. Furthermore, the latency breakdown in Figure 3(b) indicates that linear recurrence operations account for a substantial fraction of the total LRM execution time under batched execution, reaching a geometric mean of 36.5% across LRMs at a batch size of 64. Consequently, the GPU's compute units remain underutilized for a significant portion of execution. This leads not only to computational resource waste but also to energy inefficiency, as GPUs lack energy proportionality and consume substantial power even when compute units are largely underutilized [22, 42].

The root cause of these inefficiencies lies in the inherent workload characteristics of linear recurrence operations. The roofline analysis in Figure 4 shows that input projection GEMM operations become compute-bound as their arithmetic intensity increases with batch size. In contrast, linear recurrence operations remain memory-bound regardless of batch size due to the per-sequence recurrent states. Their arithmetic intensity remains far below the machine balance of modern GPUs (e.g., 200.6 FLOPs/byte for an A100 40GB GPU in BF16 precision [25]). Moreover, although the draft tree exposes sparse token interactions, GPUs typically process this sparsity through dense masked computation due to their SIMT execution model. This introduces redundant computation over invalid token pairs, further lowering the effective compute utilization of GPUs. These fundamental mismatches motivate offloading linear recurrence operations to a specialized hardware accelerator to improve both performance and energy efficiency.

3.2 Design Considerations and Challenges

While specialized hardware acceleration provides a promising solution to address the inefficiencies, designing a microarchitecture tailored to linear recurrence in tree-based speculative decoding poses two challenges. First, the size of the recurrent state $D_k \times D_v$ is substantially larger than the D_k- and D_v-dimensional input vectors, making state memory traffic the primary source of memory bandwidth pressure. Although speculative decoding partially amortizes the off-chip memory accesses for the recurrent state across multiple verified and draft tokens within a single iteration, state traffic remains the bottleneck. For instance, even when processing 32 draft tokens in GLA, memory traffic dedicated to the recurrent state still accounts for 84.8% of the total off-chip accesses. Therefore, bandwidth-efficient state management is a key design challenge to improve the performance of a linear recurrence accelerator.

Second, the accelerator datapath can be designed to exploit the sparsity of the draft token tree, thereby avoiding redundant computation. However, such sparsity-aware execution introduces irregular memory access patterns, which make on-chip data reuse difficult. Although buffering the input vectors of all draft tokens on-chip is feasible—as they are much smaller than the recurrent state—this still incurs substantial SRAM area overhead, compromising the area efficiency of the accelerator. Alternatively, to reduce SRAM usage, the output computation of the draft tree can be tiled along one of the vector dimensions. However, this approach introduces additional off-chip memory accesses due to overlapping data requirements

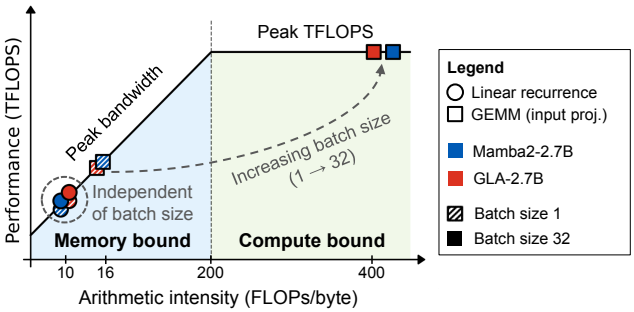


Figure 4: Analytical roofline comparison of input projection GEMM and linear recurrence operations during tree-based speculative decoding for Mamba2 [8] and GLA [39], assuming BF16 precision and 16 draft tokens.

across tiles. Moreover, when the accumulation dimension is dominant, such as D_k in the $QK^\top$ computation in Eq. (2), tiling along this dimension stalls the subsequent pipeline stage until all tiled accumulations are complete. Therefore, enabling efficient on-chip reuse under irregular memory access patterns induced by the draft tree is crucial for accelerator design.

4 PALRAC

4.1 Overview

Building on these design considerations, we propose PALRAC, a parallel linear recurrence accelerator for tree-based speculative decoding. Figure 5 shows the overall architecture of PALRAC, which can be integrated with existing xPUs (e.g., GPUs and TPUs) by attaching it to the host system interconnect. PALRAC comprises a scalable array of tree-based linear recurrence (TLR) cores, each processing up to 128 elements along the D_v dimension of a single head. For larger D_v, multiple cores are cascaded to form a single wider core, with the shared inputs fetched once from off-chip memory and multicast across the cores. This scalability allows the number of TLR cores to be configured to align with the computational capability and memory bandwidth of the target host system. Each TLR core is further divided into two decoupled modules, O_{state} and O_{draft}, corresponding to the two distinct components of our tree-based parallel linear recurrence formulation described in Section 4.2. This decoupled design allows PALRAC to specialize each execution module for its workload characteristics: the O_{state} module is tailored to dense computations for state-dependent output, while the O_{draft} module is optimized for sparse output computations over draft token trees. Furthermore, the TLR core pipelines the internal stages of the parallel linear recurrence within both modules and employs a streaming dataflow across the pipeline stages to maximize on-chip data reuse, avoiding the materialization of intermediate recurrent states and pairwise interaction matrices in on-chip buffers.

PALRAC further incorporates key microarchitectural optimizations to overcome the aforementioned design challenges. First, lazy state update is introduced to mitigate the memory bandwidth pressure caused by frequent updates of large recurrent states. Instead of writing the updated state to off-chip memory every iteration, it leverages underutilized compute resources to reconstruct the

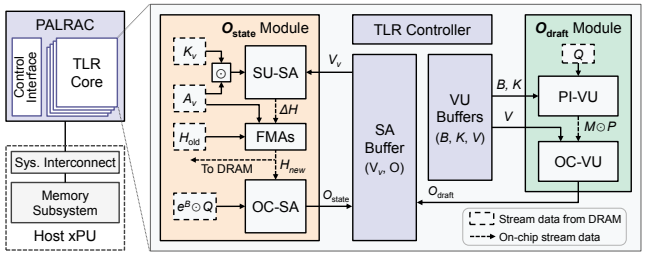


Figure 5: Overview of PALRAC and the microarchitecture of a tree-based linear recurrence (TLR) core.

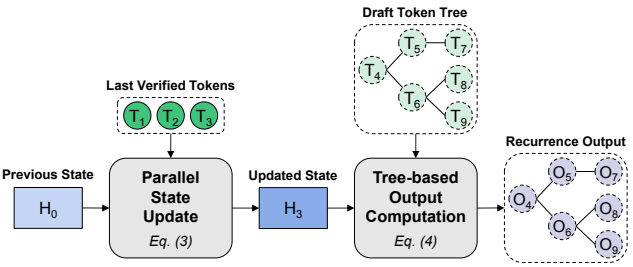


Figure 6: Proposed two-stage parallel linear recurrence for target model verification in tree-based speculative decoding: (1) *parallel state update* for newly verified tokens (Eq. (3)), and (2) *tree-based output computation* for the current draft token tree (Eq. (4)).

state on the fly. Second, to efficiently handle the irregular memory access patterns induced by the sparsity of the draft token tree, PALRAC employs tree mask regularization, which reorganizes the draft token tree to produce structured access patterns and enables a dynamic row-buffering dataflow. As a result, PALRAC achieves high-throughput, area-efficient acceleration for the proposed tree-based parallel linear recurrence under tight bandwidth and area constraints.

4.2 Tree-based Parallel Linear Recurrence Formulation

PALRAC is built on our reformulation of linear recurrence for tree-based speculative decoding. As illustrated in Figure 6, the linear recurrence operation during target model verification is performed in two distinct stages: 1) the *state update stage* for tokens verified in the previous iteration, and 2) the *tree-based output computation stage* for the current draft token tree.

Existing studies [37, 38, 43] perform the state update for verified tokens recurrently using Eq. (1). However, as shown in Figure 6, the state update stage does not require recurrence output computation for each token; it only requires the final state. Therefore, we employ a parallel state update formulation to jointly process the sequence of verified tokens, similar to the inter-chunk recurrence mechanism in the chunkwise parallel form of LRMs [8, 32, 39]. Specifically, leveraging the associativity of linear operations as in Eq. (2), we apply the cumulative transition of subsequent verified tokens to k_t prior to its outer product with v_t, instead of decaying the recurrent state at each step.

Formally, let L_V denote the number of tokens verified in the previous iteration. Adopting the activation replay strategy [37], we retrieve the cached key k_t, value v_t, and cumulative transition vectors $A_{t:L_V}$ for the verified tokens ($t = 1, \ldots, L_V$). Then the parallel state update is computed as follows:

$$H_{\text{new}} = \text{diag}(A_{1:L_V}) H_{\text{old}} + \sum_{t=1}^{L_V} (A_{t+1:L_V} \odot k_t) v_t^\top \tag{3}$$

where $A_{i:j} = \prod_{\tau=i}^{j} \alpha_\tau$ represents the cumulative transition vector over the interval $[i, j]$, with $A_{i:j} = \mathbf{1}$ for $i > j$. This formulation fundamentally shifts the decay mechanism from the large $D_k \times D_v$ recurrent state to the smaller D_k-dimensional key vectors, eliminating the computational overhead of decaying the full state at every token. By reducing the state update cost from $3L_V D_k D_v$ FLOPs to approximately $(2L_V + 1) D_k D_v$ FLOPs, it achieves higher computational efficiency compared to the recurrent approach.

Meanwhile, as further illustrated in Figure 6, the tree-based output computation stage does not require materializing recurrent states for the current draft tokens, since the recurrent state is updated in the next iteration to incorporate only the tokens accepted during verification. Thus, using the recurrent form in Eq. (1) here would incur substantial overhead, as it requires not only performing sequential state updates for every token but also duplicating the large $D_k \times D_v$ state at every branching node (i.e., a node with multiple children) of the draft token tree. In contrast, the parallel form in Eq. (2) avoids intermediate state materialization; it computes outputs directly through sequence-level matrix multiplications involving Q, K, and V, reducing both memory and computational costs.

However, while Transformers seamlessly accommodate tree structures via masked attention [23], LRMs face a structural challenge due to their fixed-size recurrent state, which inherently assumes a linear causality between tokens. Therefore, to correctly capture the causal relationships in a token tree, the cumulative transitions must align with the tree topology. To address this, we generalize the method from STree [38] to support diagonal transition matrices. Specifically, we process the draft token tree using the parallel-form linear recurrence by accumulating the diagonal transition vector α along paths in the tree.

To formalize this tree-based output computation for a draft tree with L tokens, we introduce a tree mask $M \in \{0, 1\}^{L \times L}$, where $M_{ij} = 1$ if token j is an ancestor of token i, including token i itself. Using the log-space stability technique in [39], we define the topology-aware log-cumulative transition vector $\beta_i = \sum_j M_{ij} \log \alpha_j$. Finally, the output O is derived as the sum of two terms: O_{state} capturing the historical context of the recurrent state, and O_{draft} capturing the local context of the draft token tree, as follows:

$$O = \underbrace{(\exp(B) \odot Q) H}_{O_{\text{state}}} + \underbrace{(M \odot P) V}_{O_{\text{draft}}} \tag{4}$$

where Q, K, V, and B denote the stacked matrices of q_i, k_i, v_i, and β_i, respectively, and $P \in \mathbb{R}^{L \times L}$ is the pairwise interaction matrix between draft tokens (akin to attention scores), formulated as $P_{ij} = q_i^\top (k_j \odot \exp(\beta_i - \beta_j))$.

4.3 TLR Core Microarchitecture

The primary objective of the TLR core is to accelerate the parallel state update and tree-based output computation defined in Eq. (3) and Eq. (4), respectively. As formulated, the recurrence output O decomposes into two independent terms, O_{state} and O_{draft}, which exhibit distinct computational characteristics. Specifically, O_{state} involves dense matrix operations for updating the state with verified tokens and computing the state-dependent output component for draft queries. In contrast, O_{draft} captures interactions among draft tokens determined by the tree mask M, exhibiting inherent sparsity. Consequently, as depicted in Figure 5, the TLR core employs two decoupled hardware modules for the proposed tree-based parallel linear recurrence, each tailored to its respective workload characteristics. Each module further comprises multiple pipelined stages, where data are streamed directly across these stages to maximize throughput and on-chip data reuse. This design avoids materializing the large recurrent state $H \in \mathbb{R}^{D_k \times D_v}$ in on-chip buffers, substantially reducing the on-chip memory footprint and enabling efficient processing under limited hardware resources.

O_{state} Module: As illustrated in Figure 5, the O_{state} module comprises two systolic arrays (SAs) with FMAs between them: a state update SA (SU-SA) and an output computation SA (OC-SA). The SU-SA and FMAs together perform the parallel state update for tokens verified in the previous iteration following Eq. (3). In particular, the SU-SA computes the state increment ΔH, and then the FMAs apply the cumulative transition factor to the previous state streamed from off-chip memory and combine it with ΔH to generate the updated state. The resulting H_{new} is streamed directly to the OC-SA to compute the O_{state} term in Eq. (4) for the current draft tokens. Both SAs adopt an output-stationary dataflow, efficiently accommodating variable input lengths (i.e., the number of verified tokens) during the parallel state update and maximizing partial sum reuse in the subsequent output computation.

O_{draft} Module: Although O_{draft} is formulated as a matrix multiplication, executing it as a dense operation incurs significant redundancy due to the sparsity of the tree mask M. To address this inefficiency, the O_{draft} module incorporates two cascaded vector units (VUs) designed to skip redundant operations by exploiting tree topology. As shown in Figure 7(a), the pairwise interaction VU (PI-VU) selectively calculates pairwise interactions P in row-major order, computing only for valid token pairs defined by the tree mask. Subsequently, the output computation VU (OC-VU) produces O_{draft} via sparse-dense matrix multiplication (SpMM) between the sparse matrix $M \odot P$ streamed from the PI-VU and the dense matrix V. Specifically, as illustrated in Figure 7(b), it adopts a row-wise product (RWP) approach [12] (i.e., Gustavson's algorithm [10]), which iterates over non-zero elements in each sparse row and accumulates the corresponding dense row vectors scaled by these scalar elements. By accessing the dense matrix at row granularity while skipping operations on invalid token pairs, RWP is highly amenable to parallel processing via vector operations.

Finally, the partial results from the two modules are accumulated in the output buffer before being committed to off-chip memory. To prevent write conflicts from asynchronous updates and ensure atomic accumulation, we use a 1-bit lock flag per token (i.e., per row of the output matrix O) in the buffer. Since the O_{draft} module updates partial sums in the output buffer only upon the completion of each row-wise operation, the synchronization overhead is negligible.

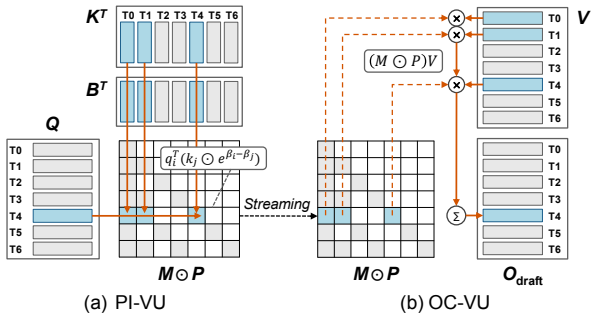


Figure 7: Execution dataflow of the vector units within the O_{draft} module: (a) the pairwise interaction VU (PI-VU) and (b) the output computation VU (OC-VU).

4.4 Lazy State Update

The recurrent state $H \in \mathbb{R}^{D_k \times D_v}$ in LRMs is updated by accumulating the outer products of keys $k \in \mathbb{R}^{D_k}$ and values $v \in \mathbb{R}^{D_v}$. Since the state is much larger than the input vectors of the linear recurrence, frequent off-chip memory accesses for state update become a primary bottleneck. This effect becomes more pronounced when the draft tree size is small, as the amount of computation decreases while off-chip memory traffic for the recurrent state remains substantial, leading to increased memory stalls in the compute units. Additionally, since the number of accepted tokens in Eq. (3) is determined at runtime, the SU-SA in the O_{state} module can be underutilized when the accepted token count is low.

To exploit these structurally underutilized compute resources and mitigate bandwidth pressure, we propose a *lazy state update* strategy that effectively reduces off-chip memory accesses. The key idea is to delay the write-back of the updated recurrent state H to off-chip memory. Instead, as shown in Figure 8, we reconstruct the latest state on the fly from lightweight input vectors—keys k, values v, and cumulative transitions A—thereby trading low-cost on-chip computation for high-cost off-chip writes.

Building upon the parallel state update formulation in Eq. (3), lazy state update leverages the associativity of the cumulative transition vectors $A_{i:j}$. Let $T = L_D + L_V$ denote the total number of pending tokens in the current lazy update process, where L_D and L_V represent the numbers of previously delayed tokens and newly verified tokens, respectively. Given the last state committed to off-chip memory, H_{mem}, the recurrent state H_T is computed as:

$$
\begin{aligned}
H_T = \operatorname{diag}\underbrace{\left(A_{L_D+1:T} \odot A_{1:L_D}\right)}_{A_{1:T}} H_{\text{mem}} \\
+ \sum_{t=1}^{L_D} (A_{L_D+1:T} \odot \underbrace{\left(A_{t+1:L_D} \odot k_t\right)}_{\text{Decayed Key } \tilde{k}_{t:L_D}}) v_t^\top \\
+ \sum_{t=L_D+1}^{T} \left(A_{t+1:T} \odot k_t\right) v_t^\top
\end{aligned}
\tag{5}
$$

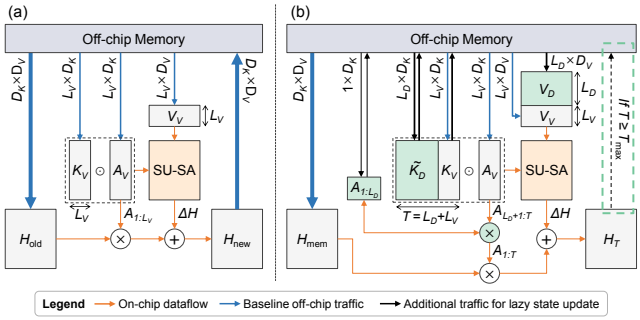


Figure 8: Comparison of state update strategies: (a) parallel state update based on Eq. (3) and (b) the proposed lazy state update based on Eq. (5). Changes are highlighted in green.

As formulated in Eq. (5), lazy state update applies cumulative transition vectors $A_{1:T}$ and $A_{t+1:T}$ to decay (i) the last committed state H_{mem} and (ii) the key vectors of delayed tokens, respectively. To implement this efficiently, we maintain two types of vectors in off-chip memory. First, the *state decay factor* $A_{1:T}$, which encodes the cumulative decay from the last committed state to the current step T, is updated iteratively. In each iteration, the previous vector $A_{1:L_D}$ is loaded, updated via element-wise multiplication with the cumulative transition vector of the newly verified tokens $A_{L_D+1:T}$, and then stored back. Compared to Eq. (3), this incurs only minimal memory and computational overhead to manage $A_{1:T} \in \mathbb{R}^{D_k}$. Second, we maintain the *decayed keys* $\tilde{k}_{t:T} = A_{t+1:T} \odot k_t$ in off-chip memory to avoid redundant decay operations for delayed tokens. In particular, the previously decayed keys $\tilde{k}_{t:L_D}$ are scaled by $A_{L_D+1:T}$ and written back to memory, except during state commit iterations. Although this approach introduces additional memory traffic for $\tilde{k}_{t:L_D}$ and v_t ($L_D(D_k + D_v)$ loads and TD_k stores), it allows the SU-SA to perform outer products between $\tilde{k}_{t:T}$ and v_t as in Eq. (3) without complicating the datapath.

Overall, the lazy state update strategy effectively replaces the heavy memory traffic required to store the updated state at every iteration with lightweight accesses for $\tilde{k}$, v, and A:

$$\underbrace{D_k \times D_v}_{\text{Store } H} \xrightarrow{\text{Lazy Update}} \underbrace{D_k + L_D(D_k + D_v)}_{\text{Load } A,\tilde{k},v} + \underbrace{(T+1)D_k}_{\text{Store } A,\tilde{k}} .$$

For instance, given $D_k = 128$ and $L_V = 8$, lazy state update yields a net reduction in memory traffic in each non-commit iteration as long as $T < 30$ for $D_v = 64$ [8] and $T < 48$ for $D_v = 128$ [30].

While lazy state update fundamentally reduces off-chip memory accesses for the recurrent state, it introduces a trade-off: the reconstruction overhead scales linearly with the number of delayed tokens. Therefore, once T reaches a state commit threshold $T_{\max}$, the reconstructed state H_T is committed to off-chip memory and the lazy state update process restarts. However, the best trade-off between off-chip memory traffic reduction and reconstruction overhead depends on the characteristics of the target workload. For instance, a larger draft tree size typically results in more verified tokens per iteration, which causes more frequent state commits for a given $T_{\max}$, making a higher $T_{\max}$ more beneficial. In addition,

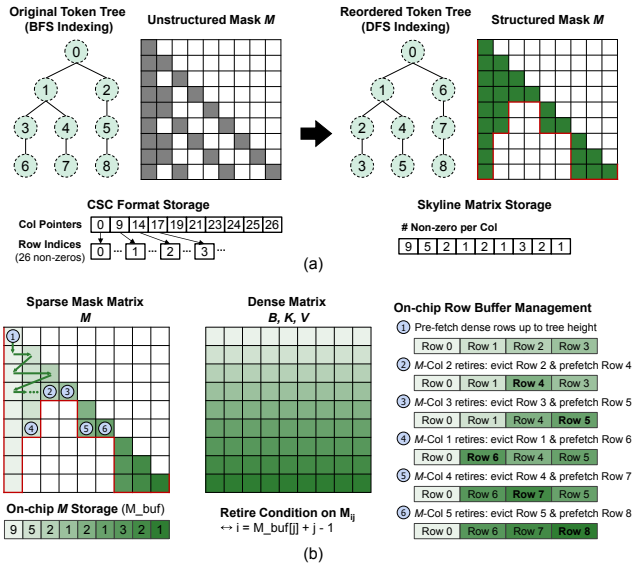


Figure 9: Illustrative example of tree mask regularization: (a) structured tree mask generation via DFS indexing and (b) dynamic row buffer management for the proposed dataflow.

LRMs with large recurrent state sizes can benefit more from increasing $T_{\max}$ because the state update traffic accounts for a larger portion of the total memory accesses. Therefore, PALRAC exposes $T_{\max}$ as a per-core configurable parameter to balance memory traffic reduction and reconstruction overhead across different workloads.

4.5 Tree Mask Regularization for Efficient Row Buffering

As discussed in Section 3.2, exploiting sparse token interactions in the draft tree creates a trade-off between on-chip buffer requirements and off-chip memory traffic. Buffering all input vectors of draft tokens on-chip avoids redundant off-chip memory accesses but incurs substantial area overhead, whereas tiling reduces the buffer footprint at the cost of additional off-chip memory accesses. To address these limitations, we leverage the observation that the sparsity pattern of the draft token tree mask M can be structured by controlling the node indexing order. Specifically, indexing tokens in the draft tree via depth-first search (DFS) aligns the non-zero elements within each column of M into contiguous rows, forming a *skyline matrix* [3], as the recursive nature of DFS maps the nodes within each subtree to a contiguous index range. Furthermore, pre-order traversal ensures that parent tokens are indexed prior to their children, thereby preserving the inherent causality within the draft token tree. Consequently, applying pre-order DFS transforms M into a lower-triangular skyline matrix, as illustrated in Figure 9(a). Unlike the conventional compressed sparse column (CSC) format, which stores both column pointers and row indices, the skyline matrix can be represented solely by per-column non-zero counts, reducing storage overhead.

In addition, the structured tree mask M induces strong temporal and spatial locality in the O_{draft} module's access patterns for

matrices B, K, and V. Specifically, the j-th rows of these matrices are required only while processing the subtree rooted at token j. Within the structured mask M, this subtree maps to the contiguous non-zero interval in the j-th column. Thus, the lifetime of each row of B, K, and V is determined by the corresponding non-zero interval, obviating the need to buffer the entire matrices on-chip.

Another key insight is that the number of simultaneously live B, K, and V rows—required to eliminate redundant off-chip accesses—is bounded by the tree height, rather than the total number of tokens in the tree. Since the tree mask M encodes ancestor-descendant relationships, the maximum number of ancestors of a given token (i.e., the non-zero elements in a row of M) is limited by the tree height. This enables B, K, and V data to be reused entirely within small on-chip buffers, each sized to store a number of rows equal to the tree height. For instance, for a 32-node tree of height 6, only 18.75% of the data requires buffering, significantly reducing the on-chip memory footprint.

Based on these insights, as depicted in Figure 9(b), we propose tree mask regularization to enable efficient data reuse with a reduced on-chip memory footprint. First, by leveraging the skyline structure of M, the TLR core maintains only the non-zero element counts for each column. This not only minimizes the on-chip storage for M but also eliminates the row-index decoding inherent in typical CSC formats, thereby simplifying the control logic and reducing the access latency of M. During execution, the TLR controller utilizes the column-wise non-zero counts to dynamically manage the residency of the B, K, and V rows. Specifically, as the two VUs in O_{draft} produce and consume the intermediate matrix $M \odot P$ in row-major order, the TLR controller tracks the remaining non-zero elements in the active M-columns, with the number of active columns bounded by the tree height. While computing the i-th row of $M \odot P$, if M_{ij} is identified as the final non-zero element in the j-th column, the TLR controller evicts the respective j-th rows of B, K, and V upon completion of their processing, and prefetches rows for a newly active M-column. As a result, this row-buffering dataflow enabled by tree mask regularization requires only small on-chip buffers while ensuring that each row of B, K, and V is fetched from off-chip memory exactly once.

5 Evaluation

5.1 Methodology

Simulator and Baselines. To validate our tree-based parallel linear recurrence algorithm and establish the baseline GPU performance for linear recurrence operations, we first extend the FLA (Flash Linear Attention) library [40] to support tree-based speculative decoding for LRMs. Specifically, we modify its Triton-based kernels [33] for linear recurrence to implement our tree-based parallel linear recurrence algorithm. Using this FLA-based implementation, we profile the execution of the linear recurrence kernels via NVIDIA Nsight Systems [27] to obtain the GPU baseline. We then develop an in-house cycle-level simulator for functional and performance validation of PALRAC. To isolate the architectural benefits of our design and avoid discrepancies in memory subsystem modeling, we use the off-chip memory model of GPGPU-Sim 4.0 [2] embedded in Accel-Sim [18], a GPU simulation framework for modern GPUs.

Table 1: Target model configurations used for evaluation.

Model	Layers	Heads	D_k	D_v	State decay	State / layer*
GLA-2.7B	32	5	256	512	diagonal	1.25 MiB
HGRN2-2.7B	32	20	128	128	diagonal	0.625 MiB
Mamba2-2.7B	64	80	128	64	scalar	1.25 MiB
RetNet-2.7B	32	10	256	512	scalar	2.5 MiB

*State sizes assume BF16 storage.

Table 2: PALRAC configuration and TLR core microarchitecture.

PALRAC configuration	
Data type	BF16
Clock frequency	1.0 GHz
Number of TLR cores	16
Off-chip memory BW	512 GB/s
TLR core microarchitecture	
O_{state} module	16×16 SU-SA, 16 FMAs, 16×16 OC-SA
O_{draft} module	16-lane PI-VU, 16-lane OC-VU

We compare four execution configurations to evaluate PALRAC. **GPU-base** executes linear recurrence operations on a single NVIDIA A100 40GB GPU using the FLA-based kernels. To evaluate the effectiveness of our accelerator design, we further introduce an accelerator baseline, **ACC-base**, which employs recurrent-form linear recurrence for the state update and output computation stages as in prior works [37, 43]. It adopts a datapath spatially unrolled along the D_k dimension and processes the D_v dimension in tiles, following the linear recurrence unit implementation of SpecMamba [43]. For a fair comparison, we choose the D_v tile size and operating frequency such that ACC-base matches the peak compute throughput of PALRAC. Although this design is not inherently configurable across different D_k values, we instantiate ACC-base for each evaluated model. We compare these baselines against **PALRAC-base**, which implements our tree-based parallel linear recurrence with tree mask regularization, and **PALRAC-lazy**, which additionally enables lazy state update. In our performance simulations, both ACC-base and the 16-core PALRAC are modeled with an off-chip memory bandwidth of 512 GB/s.

Workloads and Settings. We evaluate PALRAC on four representative LRMs: GLA [39], HGRN2 [30], Mamba2 [8], and RetNet [32]. We use their 2.7B-parameter variants as target models in tree-based speculative decoding. Table 1 summarizes the model configurations used in the evaluation. We examine linear recurrence operations for these models across a range of batch sizes (8, 16, 32, 64) and draft tree sizes (8, 32) to comprehensively characterize performance under diverse workloads. To ensure a consistent target model verification workload without assuming a specific draft model, we set target mean accepted lengths (MALs), i.e., the average number of accepted tokens per verification, to 3 and 6 for the 8- and 32-node draft token trees, respectively. At each target model verification, we sample an accepted length for each request to capture iteration-level

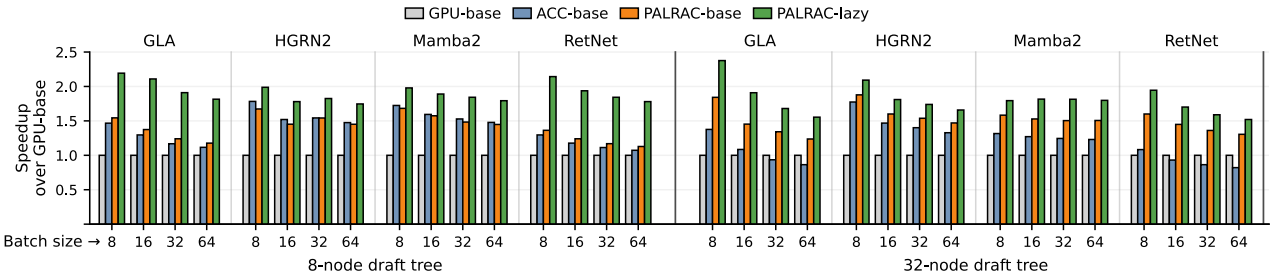


Figure 10: Speedup of linear recurrence operations across LRMs for different batch sizes and draft tree sizes.

variation, while ensuring that the mean across the batch matches the target MAL.

Hardware Implementations. We implement the datapath of the TLR core in Verilog and synthesize it using Synopsys Design Compiler with the 7nm predictive process design kit ASAP7 [7] to estimate its logic area and power, targeting a 1 GHz clock frequency. The on-chip SRAM buffers and queues are modeled with FN-CACTI [31], and their area and power are combined with the synthesized logic results to obtain the total on-chip area and power of the TLR core. We use AccelWattch [16] to estimate the energy consumption of GPU execution, as well as the off-chip memory access energy incurred by the TLR core. Table 2 summarizes the PALRAC configuration and TLR core microarchitecture used in the evaluation.

System Integration. To evaluate PALRAC's system-level impact, we model an integrated GPU–PALRAC system using the same A100 GPU configuration as GPU-base and following the system organization shown in Figure 5. In the evaluated system, PALRAC is attached to the GPU through the system interconnect and shares its DRAM subsystem. The GPU and PALRAC exchange projected activations and linear recurrence outputs through the shared DRAM, which also stores the recurrent states accessed by PALRAC. The evaluated latency accounts for all associated DRAM traffic under GPU-prioritized arbitration. For the control interface, PALRAC is exposed to the GPU as a memory-mapped I/O (MMIO) device, allowing the GPU to initiate execution and monitor completion through memory writes and reads, respectively. To model integrated execution, we collect GPU execution traces for LRM blocks from our FLA-based implementation. We replay these traces in Accel-Sim, replacing the linear recurrence kernels with a lightweight GPU kernel that controls PALRAC through MMIO accesses, while modeling offload-control overhead and contention in the shared DRAM subsystem.

5.2 Tree-based Linear Recurrence Performance

Figure 10 compares the performance of linear recurrence operations across different LRMs, batch sizes, and draft tree sizes. On average, the proposed PALRAC configurations outperform both GPU-base and ACC-base across the evaluated workloads. PALRAC-base achieves a geometric-mean speedup of 1.45× over GPU-base, whereas ACC-base provides a more modest speedup of 1.26×. This performance gap demonstrates the effectiveness of our tree-based

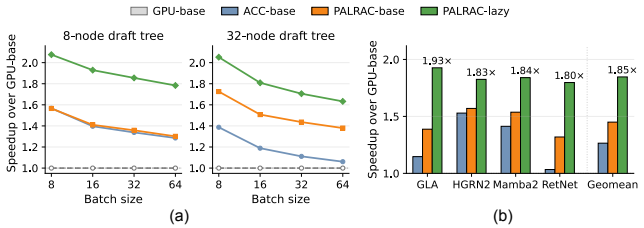


Figure 11: Speedup of linear recurrence operations over GPU-base across (a) batch sizes and (b) evaluated LRMs.

parallel linear recurrence formulation and the decoupled microarchitecture, which separately optimizes the dense and sparse computational patterns. Furthermore, by reducing the off-chip memory traffic through lazy state update, PALRAC-lazy achieves the highest geometric-mean speedup of 1.85× over the GPU baseline, corresponding to a 1.46× improvement over ACC-base.

Figure 11(a) illustrates the speedup of linear recurrence operations across batch sizes for each draft tree size. In general, the speedup slightly decreases as batch size increases, because larger batches expose more parallelism in the GPU and better amortize kernel overhead, improving the execution efficiency of GPU-base. Although ACC-base avoids off-chip materialization of intermediate states through spatial pipelining and tiling, its recurrent formulation still requires intermediate state computation during the state update and output computation stages. In contrast, PALRAC-base leverages the parallel formulation to perform only the computations required for each stage, reducing the total amount of computation. As a result, the performance advantage of PALRAC-base becomes more pronounced as the draft tree size increases. When the draft tree size is small, however, the performance gap between PALRAC-base and ACC-base narrows. This is because the parallel linear recurrence becomes more memory-bound at smaller draft tree sizes, reducing the benefit of PALRAC's lower computational cost.

The model-wise performance results in Figure 11(b) reveal two distinct trends. First, PALRAC-base delivers larger speedups over GPU-base for HGRN2 and Mamba2 than for GLA and RetNet. The off-chip memory traffic of the recurrent state scales with $D_k \times D_v$, whereas non-state vector traffic grows only linearly with D_k and D_v. Thus, in models with smaller per-head dimensions, non-state vector data accounts for a larger fraction of the total off-chip memory traffic. This can lower the arithmetic intensity of linear recurrence,

further exacerbating the inefficiency of GPU-base and thereby increasing the relative benefit of PALRAC-base. Second, by contrast, the additional gain from PALRAC-lazy over PALRAC-base is more pronounced for GLA and RetNet, as their larger per-head states provide greater opportunity for reducing state update traffic through lazy state update. Overall, these results show that the compute-efficient design of PALRAC-base and the memory traffic reduction of PALRAC-lazy complement each other, enabling PALRAC to achieve high performance across diverse model configurations.

5.3 Effectiveness of Lazy State Update

We next evaluate the effectiveness of lazy state update by comparing PALRAC-lazy with PALRAC-base. Overall, lazy state update provides an additional 27.4% geometric-mean improvement over PALRAC-base. Since the performance advantage of lazy state update stems from reducing off-chip memory traffic, its benefit becomes larger when PALRAC's performance is more constrained by off-chip memory bandwidth. As the draft tree size increases, however, the workload of output computation over the draft tree grows, whereas the off-chip memory accesses for the recurrent state remain constant. As a result, for a given batch size, increasing the draft tree size amortizes the cost of recurrent-state memory traffic over a larger amount of computation, thereby reducing the effective bandwidth pressure imposed by recurrent state updates in PALRAC. Consistent with this analysis, for the 8-node draft tree, PALRAC-lazy yields an additional 32.9–38.0% performance improvement over PALRAC-base across batch sizes, whereas the corresponding gain for the 32-node draft tree is 18.5–20.0%.

Figure 12 further examines the theoretical reduction in off-chip memory traffic achieved by lazy state update. The traffic reduction initially increases as $T_{\max}$ grows, since a larger $T_{\max}$ allows more state commits to be deferred. However, as more state updates are delayed, the amount of delayed-token data required for state reconstruction also increases. Consequently, for each model, the benefit of lazy state update decreases beyond a certain $T_{\max}$ that provides the peak reduction. Moreover, the peak traffic reduction is achieved at a higher $T_{\max}$ as the draft tree size increases. This is because larger draft trees tend to have higher MALs, leading to more frequent state commits for a given $T_{\max}$. These results indicate a trade-off between off-chip memory traffic reduction and reconstruction overhead. In the evaluation, we use a static $T_{\max}$ for each model and draft tree size based on offline profiling to balance these two effects. Our sensitivity analysis at batch size 32 shows that the statically profiled $T_{\max}$ remains robust when MAL varies by ±2 around the MALs used for profiling, incurring at most 2.73% slowdown relative to thresholds tuned separately for each MAL.

5.4 Area and Power Analysis

Table 3 presents the area breakdown of the hardware components in PALRAC. In the baseline design without tree mask regularization, the on-chip SRAM occupies 0.1188 mm^2, corresponding to 45.2% of the total TLR core area. This SRAM overhead is dominated by the VU buffers for the O_{draft} module: the baseline 3×16 KiB VU buffers occupy 0.0577 mm^2, accounting for 63.2% of the total SRAM capacity and 48.6% of the total SRAM area. Since tree mask regularization bounds the number of simultaneously live rows by the tree height, it substantially reduces the on-chip buffering required for sparse draft tree processing. Specifically, by replacing the baseline assumption of maximum draft tree sizes of 32 for $D_k = 256$ and 64 for $D_k = 128$ with maximum tree heights of 8 and 16, respectively, we reduce the VU buffer capacity to one-fourth of the baseline. As a result, the area of the VU buffers decreases by 73.5%, from 0.0577 mm^2 to 0.0153 mm^2. Overall, tree mask regularization reduces the total TLR core area by 16.1% and the total SRAM area by 35.7%.

The power breakdown of the PALRAC components is also detailed in Table 3. The SRAM power is conservatively modeled using the ideal throughput of PALRAC without memory stalls due to off-chip memory accesses, which provides an upper-bound estimate of SRAM activity. Within a single TLR core, the O_{state} module dominates power consumption, accounting for 84.4% of the total power due to the dense computations over the large recurrent state, while the O_{draft} module and SRAM structures contribute only 10.2% and 5.4%, respectively. Since the proposed PALRAC consists of 16 TLR cores, it incurs a total area of 3.5248 mm^2 and power consumption of 3.27 W. Notably, compared to the A100 GPU, which has a die size of 826 mm^2 and a TDP of 400 W, PALRAC occupies only 0.43% of its die area and consumes 0.82% of its TDP. These results demonstrate that PALRAC requires only a small area and power footprint while delivering substantial performance gains for linear recurrence operations.

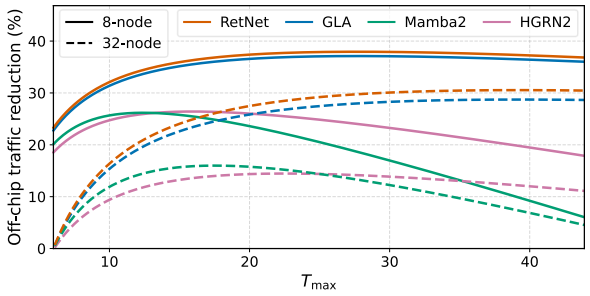


Figure 12: Theoretical reduction in off-chip memory traffic achieved by lazy state update as a function of the state commit threshold ($T_{\max}$) for different draft tree sizes.

Table 3: Area and power breakdown of PALRAC.

Component	Area (mm^2)	Power (mW)
O_{state} module	0.1281	172.49
O_{draft} module	0.0158	20.88
SA buffer (1×20 KiB)	0.0360	5.17
VU buffers*	0.0153 (0.0577)	1.80 (3.45)
Queues (8×1 KiB)	0.0251	4.03
Logic total	0.1439	193.37
SRAM total	0.0764 (0.1188)	11.00
TLR core	0.2203 (0.2627)	204.37
PALRAC (16 TLR cores)	3.5248 (4.2032)	3269.92

Note: Values in parentheses indicate the baseline without tree mask regularization.
*Tree mask regularization reduces the size of VU buffers from 3×16 KiB to 3×4 KiB.

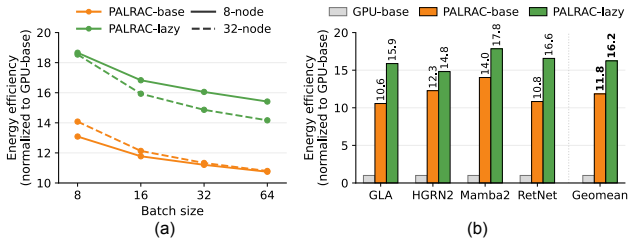


Figure 13: Energy efficiency of linear recurrence operations, normalized to GPU-base, across (a) batch sizes for different draft tree sizes and (b) evaluated LRMs.

5.5 Energy Efficiency

Figure 13 shows the energy efficiency improvement of PALRAC over the GPU for processing linear recurrence operations. The evaluated energy consumption includes off-chip memory accesses. In the PALRAC configurations, off-chip memory accesses dominate the total energy because the power consumption of PALRAC itself is very small. As shown in Figure 13(a), PALRAC-base and PALRAC-lazy exhibit different energy efficiency trends with respect to draft tree size. PALRAC-base is more energy efficient with the 32-node draft tree because the larger tree better utilizes its compute-efficient design, whereas PALRAC-lazy is more energy efficient with the 8-node draft tree because memory cost is more dominant at smaller tree sizes. Meanwhile, both configurations show more pronounced gains at small batch sizes. This is because the GPU's energy consumption increases sublinearly with batch size, as small batch sizes suffer from lower kernel execution efficiency due to limited thread block concurrency. In contrast, PALRAC exhibits almost linear scaling because it maintains consistently high efficiency across batch sizes. Nevertheless, PALRAC still maintains significant energy savings even at a batch size of 64.

Figure 13(b) further illustrates the energy efficiency of linear recurrence operations for each target LRM, averaged across batch sizes and draft tree sizes. Consistent with the performance trends in Section 5.2, PALRAC-base is more energy efficient for HGRN2 and Mamba2 than for GLA and RetNet, whereas PALRAC-lazy provides greater benefits for GLA and RetNet and therefore delivers larger additional energy-efficiency gains over PALRAC-base for these models. Overall, across all evaluated models, PALRAC-base and PALRAC-lazy achieve geometric-mean energy-efficiency improvements of 11.8× and 16.2× over the GPU baseline, respectively.

5.6 Target Model Verification Performance

To examine PALRAC's system-level impact on target model verification, we evaluate two execution schemes for offloading linear recurrence operations: serial offload and micro-batch pipelining [11]. For both execution schemes, linear recurrence operations are executed on PALRAC with lazy state update enabled, while the remaining operations are executed on the GPU. In serial offload, these operations are serialized by their data dependencies: PALRAC consumes the outputs of the GPU's input projections, while subsequent GPU operations depend on the recurrence outputs. As characterized in Section 3.1, GEMM operations, such as input projections, become compute-bound as batch size increases, whereas linear recurrence operations remain primarily memory-bound. Thus, serial offload does not exploit the memory bandwidth available in the shared DRAM subsystem during compute-bound GPU execution. To take advantage of the complementary resource demands, we employ micro-batch pipelining, which overlaps linear recurrence operations on PALRAC with GPU operations across different micro-batches.

Table 4: Target model verification speedup over GPU-only execution at batch size 32.

Execution scheme	Draft tree	GLA	HGRN2	Mamba2	RetNet
Serial offload	8-node	1.23×	1.17×	1.24×	1.30×
	32-node	1.12×	1.10×	1.14×	1.14×
Micro-batch pipelining	8-node	1.45×	1.30×	1.49×	1.53×
	32-node	1.36×	1.28×	1.35×	1.44×

Table 4 reports target model verification speedups over GPU-only execution at a batch size of 32. Across the four models and two draft tree sizes, serial offload achieves a geometric-mean speedup of 1.18× by accelerating linear recurrence operations on PALRAC while the GPU and PALRAC execute sequentially. For micro-batch pipelining, we partition the batch into two micro-batches of 16 sequences and assign them to separate CUDA streams [28]. This concurrent execution hides part of the linear recurrence latency behind GPU computation by exploiting the shared DRAM bandwidth left underutilized, increasing the geometric-mean speedup to 1.40×.

5.7 Scalability

Host Bandwidth and Core Scaling. We further examine how PALRAC scales with host memory bandwidth and TLR core count. Since linear recurrence operations are memory-bound on GPUs, increasing the host GPU's bandwidth can reduce the performance advantage of a fixed PALRAC configuration. To analyze this bandwidth sensitivity, we conservatively assume that linear recurrence throughput on the GPU scales ideally with memory bandwidth, from 1.555 TB/s for GPU-base to 3.35 TB/s for GPU-H100-BW, a bandwidth-scaled baseline matching the memory bandwidth of an H100 SXM GPU [26]. For each GPU baseline, we evaluate 16-, 32-, and 48-core PALRAC-lazy configurations at batch size 32 while preserving 32 GB/s of memory bandwidth per core.

Table 5 reports geometric-mean speedups of linear recurrence operations across the evaluated models and draft tree sizes. With 16 TLR cores, PALRAC-lazy achieves a speedup of 1.78× over GPU-base, but only 0.83× over GPU-H100-BW. However, increasing the TLR core count to 32 and 48 raises the speedup over GPU-H100-BW to 1.64× and 2.42×, respectively. Notably, performance scaling with TLR core count becomes sublinear on the GPU-base host as memory bandwidth saturates, whereas the higher bandwidth of GPU-H100-BW enables nearly linear scaling. These results highlight the benefit of PALRAC's scalable design, which allows the core count to be matched to the host system's compute capability and memory bandwidth (Section 4.1).

Model Size. We next evaluate synthetic 7B and 20B model configurations by increasing the number of layers and heads while preserving the per-head dimensions, following common depth and

Table 5: Linear recurrence speedup of PALRAC over GPU baselines at batch size 32 under different host GPU memory bandwidths and TLR core counts.

GPU baseline	Bandwidth	16 cores	32 cores	48 cores
GPU-base	1.555 TB/s	1.78×	3.16×	3.66×
GPU-H100-BW	3.35 TB/s	0.83×	1.64×	2.42×

Table 6: Linear recurrence speedup of PALRAC over GPU-base across model scales.

Model scale	8-node tree	32-node tree	Overall
2.7B	1.91×	1.79×	1.85×
7B-scaled	1.84×	1.75×	1.79×
20B-scaled	1.77×	1.73×	1.75×

width scaling of Transformer-based LLMs [4, 9]. Relative to each 2.7B baseline, the 7B and 20B configurations scale the layer/head counts by 1.3125×/1.4× and 1.9375×/2.0×, respectively. We retain the 16-core PALRAC-lazy configuration with 512 GB/s of memory bandwidth across all model scales. Since TLR cores are time-multiplexed across layers and heads, larger models increase total execution time and memory traffic without requiring higher peak memory bandwidth for a given PALRAC configuration. As shown in Table 6, the geometric-mean speedup across models, batch sizes, and draft tree sizes decreases modestly from 1.85× for the 2.7B baselines to 1.79× and 1.75× for the 7B-scaled and 20B-scaled configurations, respectively. This decrease can be attributed to the greater GPU parallelism provided by additional heads, particularly at smaller batch sizes, consistent with the trend in Figure 11(a). Nevertheless, PALRAC preserves most of its performance advantage over GPU-base across the evaluated model scales.

6 Conclusion

This paper presents PALRAC, a parallel linear recurrence accelerator for tree-based speculative decoding in LRMs. We introduce a parallel linear recurrence formulation for target model verification, which consists of two stages: parallel state update for verified tokens and tree-based output computation for draft tokens. Based on this formulation, PALRAC employs decoupled execution modules tailored to dense state-dependent computation and sparse draft-token interactions. Moreover, PALRAC introduces lazy state update to reduce off-chip memory traffic and tree mask regularization to enable area-efficient sparse computation over draft token trees. For linear recurrence operations, PALRAC achieves a geometric-mean speedup of 1.85× over the GPU baseline and improves energy efficiency by 16.2×, while requiring only a small area and power footprint. At a batch size of 32, system-level simulation with micro-batch pipelining further shows that these operation-level gains translate into a geometric-mean speedup of 1.40× in target model verification over GPU-only execution. These results demonstrate that specialized hardware acceleration is an effective approach for enabling efficient tree-based speculative decoding in LRMs.

Acknowledgments

This work was supported in part by Institute of Information & communications Technology Planning & Evaluation (IITP) under the Graduate School of Artificial Intelligence Semiconductor (IITP-2026-RS-2023-00256081) grant funded by the Korea government (MSIT), and in part by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (RS-2024-00337079).

References

[1] Anthropic. 2024. Claude. https://claude.ai
[2] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 163–174.
[3] Klaus-Jürgen Bathe. 1996. *Finite Element Procedures*. Prentice Hall.
[4] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. 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, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, Vol. 33. 1877–1901.
[5] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. 2023. Accelerating Large Language Model Decoding with Speculative Sampling. *arXiv preprint arXiv:2302.01318* (2023). https://arxiv.org/abs/2302.01318
[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374* (2021). https://arxiv.org/abs/2107.03374
[7] Lawrence T. Clark, Vinay Vashishtha, Lucian Shifren, Aditya Gujja, Saurabh Sinha, Brian Cline, Chandramouli Ramamurthy, and Greg Yeric. 2016. ASAP7: A 7-nm FinFET Predictive Process Design Kit. *Microelectronics Journal* 53 (2016), 105–115.
[8] Tri Dao and Albert Gu. 2024. Transformers are SSMs: Generalized Models and Efficient Algorithms Through Structured State Space Duality. In *Proceedings of the 41st International Conference on Machine Learning*. 10041–10071.
[9] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The Llama 3 Herd of Models. *arXiv preprint arXiv:2407.21783* (2024). https://arxiv.org/abs/2407.21783
[10] Fred G. Gustavson. 1978. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Trans. Math. Software* 4, 3 (Sept. 1978), 250–269.
[11] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Advances in Neural Information Processing Systems*, Vol. 32. 103–112.
[12] Ranggi Hwang, Minhoo Kang, Jiwon Lee, Dongyun Kam, Youngjoo Lee, and Minsoo Rhu. 2023. GROW: A Row-Stationary Sparse-Dense GEMM Accelerator for Memory-Efficient Graph Convolutional Neural Networks. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 42–55.
[13] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Clifford Young, Xiang Zhou, Zongwei Zhou, and David A Patterson. 2023. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*.
[14] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham,

Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*.

[15] Hongshin Jun, Jinhee Cho, Kangseol Lee, Ho-Young Son, Kwiwook Kim, Hanho Jin, and Keith Kim. 2017. HBM (High Bandwidth Memory) DRAM Technology and Architecture. In *2017 IEEE International Memory Workshop (IMW)*. 1–4.

[16] Vijay Kandiah, Scott Peverelle, Mahmoud Khairy, Junrui Pan, Amogh Manjunath, Timothy G. Rogers, Tor M. Aamodt, and Nikos Hardavellas. 2021. AccelWattch: A Power Modeling Framework for Modern GPUs. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21)*. 738–753.

[17] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. 2020. Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention. In *Proceedings of the 37th International Conference on Machine Learning*. 5156–5165.

[18] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. 2020. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 473–486.

[19] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*. 611–626.

[20] Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast Inference from Transformers via Speculative Decoding. In *Proceedings of the 40th International Conference on Machine Learning*. 19274–19286.

[21] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2024. EAGLE-2: Faster Inference of Language Models with Dynamic Draft Trees. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. 7421–7432.

[22] Ravi Reddy Manumachu and Alexey Lastovetsky. 2022. On Energy Nonproportionality of CPUs and GPUs. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 34–44.

[23] 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 (ASPLOS '24)*. 932–949.

[24] Microsoft. 2024. GitHub Copilot. https://github.com/features/copilot

[25] NVIDIA. 2022. NVIDIA A100 GPU. https://www.nvidia.com/en-us/data-center/a100/

[26] NVIDIA. 2024. NVIDIA H100 GPU. https://www.nvidia.com/en-us/data-center/h100/

[27] NVIDIA. 2024. NVIDIA Nsight Systems. https://developer.nvidia.com/nsight-systems

[28] NVIDIA. 2026. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[29] OpenAI. 2024. ChatGPT. https://chat.openai.com

[30] Zhen Qin, Songlin Yang, Weixuan Sun, Xuyang Shen, Dong Li, Weigao Sun, and Yiran Zhong. 2024. HGRN2: Gated Linear RNNs with State Expansion. *arXiv preprint arXiv:2404.07904* (2024). https://arxiv.org/abs/2404.07904

[31] Divya Praneetha Ravipati, Rajesh Kedia, Victor M. van Santen, Jörg Henkel, Preeti Ranjan Panda, and Hussam Amrouch. 2022. FN-CACTI: Advanced CACTI for FinFET and NC-FinFET Technologies. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 30, 3 (2022), 339–352.

[32] Yutao Sun, Li Dong, Shaohan Huang, Shuming Ma, Yuqing Xia, Jilong Xue, Jianyong Wang, and Furu Wei. 2023. Retentive Network: A Successor to Transformer for Large Language Models. *arXiv preprint arXiv:2307.08621* (2023). https://arxiv.org/abs/2307.08621

[33] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. 2019. Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL 2019)*. 10–19.

[34] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Advances in Neural Information Processing Systems*, Vol. 30. 5998–6008.

[35] Junxiong Wang, Tushaar Gangavarapu, Jing Nathan Yan, and Alexander M. Rush. 2024. MambaByte: Token-free Selective State Space Model. *arXiv preprint arXiv:2401.13660* (2024). https://arxiv.org/abs/2401.13660

[36] Junxiong Wang, Daniele Paliotta, Avner May, Alexander M. Rush, and Tri Dao. 2024. The Mamba in the Llama: Distilling and Accelerating Hybrid Models. In *Advances in Neural Information Processing Systems*, Vol. 37. 62432–62457.

[37] Yangchao Wu, Yonatan Dukler, Matthew Trager, Alessandro Achille, Wei Xia, and Stefano Soatto. 2024. Snakes and Ladders: Accelerating State Space Model Inference with Speculative Decoding. In *Proceedings of the 4th NeurIPS Efficient Natural Language and Speech Processing Workshop*. 292–304.

[38] Yangchao Wu, Zongyue Qin, Alex Wong, and Stefano Soatto. 2025. STree: Speculative Tree Decoding for Hybrid State-Space Models. *arXiv preprint arXiv:2505.14969* (2025). NeurIPS 2025 poster. https://arxiv.org/abs/2505.14969

[39] Songlin Yang, Bailin Wang, Yikang Shen, Rameswar Panda, and Yoon Kim. 2024. Gated Linear Attention Transformers with Hardware-Efficient Training. In *Proceedings of the 41st International Conference on Machine Learning*. 56501–56523.

[40] Songlin Yang and Yu Zhang. 2024. *FLA: A Triton-Based Library for Hardware-Efficient Implementations of Linear Attention Mechanism*. https://github.com/fla-org/flash-linear-attention

[41] Haoran You, Yichao Fu, Zheng Wang, Amir Yazdanbakhsh, and Yingyan Celine Lin. 2024. When Linear Attention Meets Autoregressive Decoding: Towards More Effective and Efficient Linearized Large Language Models. In *Proceedings of the 41st International Conference on Machine Learning*. 57350–57366.

[42] Jie You, Jae-Won Chung, and Mosharaf Chowdhury. 2023. Zeus: Understanding and Optimizing GPU Energy Consumption of DNN Training. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 119–139.

[43] Linfeng Zhong, Songqiang Xu, Huifeng Wen, Tong Xie, Qingyu Guo, Yuan Wang, and Meng Li. 2025. SpecMamba: Accelerating Mamba Inference on FPGA with Speculative Decoding. In *2025 IEEE/ACM International Conference on Computer-Aided Design (ICCAD '25)*.