KVBuffer: IO-aware Serving for Linear Attention

Longwei Zou
Department of Computer Science
Yale University
New Haven, CT 06511
longwei.zou@yale.edu

Lin Zhong
Department of Computer Science
Yale University
New Haven, CT 06511
lin.zhong@yale.edu

Abstract

Linear attention has recently gained significant attention for long-context inference due to its constant decoding cost with respect to context length. However, existing serving systems typically serve linear attention by recurrently computing and updating a large linear attention state in every decoding step. Since the state is much larger than the per-token key and value, recurrent decoding incurs substantial memory access and becomes inefficient for serving linear attention. In this paper, we propose KVBuffer, an IO-aware serving mechanism for linear attention. By buffering recent keys and values, KVBuffer enables serving systems to compute linear attention outputs in more flexible and memory-efficient ways. For decoding, KVBuffer enables chunkwise computation, which reduces average memory access and decoding latency by deferring state updates and applying them in batch. For speculative decoding, KVBuffer verifies draft tokens in parallel and avoids storing temporary states. For short contexts, KVBuffer computes attention outputs directly from buffered keys and values, without creating or updating the linear attention state. We implement KVBuffer in SGLang for Qwen3.5 and Nemotron-3 series models. Our evaluations show that KVBuffer can speedup batch decoding by up to $1.47\times$ and increase the maximum number of serving requests by $5\times$ for speculative decoding when verifying four draft tokens.

1 Introduction

In recent years, long-context workloads have become increasingly prevalent in the applications of Large Language Models (LLMs), particularly agentic applications. To address the growing demand for efficient long-context processing, linear attention has attracted growing interest due to its constant decoding cost and bounded GPU memory footprint with respect to context length. With techniques such as gating[21], decaying mechanisms[4] and delta rule[22, 20], linear attention-based LLMs have substantially narrowed the quality gap with those using softmax-based attention. As a result, many recent LLMs[15, 24, 10] employ hybrid architectures that interleave linear and softmax attentions to balance model quality and inference efficiency, achieving improved trade-offs over standard Transformer designs.

Existing serving systems[7, 25], unfortunately, are very inefficient for linear attention-based models, because they compute the linear attention state recurrently (§2.2). Typically, a serving system maintains one linear attention state for each request. After prefilling the prompt into linear attention state, the serving system updates the state with the newly generated key and value in each decoding step, then produces the attention layer output by querying the updated state. The inefficiency stems from the large size of the linear attention state and how it is updated. The linear attention state is usually larger than the per-token key and value (KV) by two orders of magnitude, e.g., 2MB in Qwen3-Next Gated DeltaNet Layer[15]. Existing serving systems read and write the linear attention state in every decoding step, which consumes a substantial portion of the memory bandwidth and is sequential in nature. The problem is even more pronounced in two common cases. In speculative decoding, the serving system must recurrently compute a temporary linear attention state for each

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

draft token and store these temporary states until draft tokens are verified, further multiplying memory and memory access by the number of draft tokens. For example, verifying 4 draft tokens occupies an additional 294.75MB of memory per-request in the Qwen3-Next model, imposing a substantial burden on GPU memory. For short contexts, the linear attention state can use more memory than the keys and values (KVs) of all tokens. In this case, computing attention output directly from all KVs can be more memory-efficient than recurrently maintaining the linear attention state.

In this work, we present `KVBuffer`, an IO-aware serving mechanism for linear attention. The key idea is to buffer the KVs of consecutive tokens and update the linear attention state in batch, in parallel, instead of recurrently. Although this incurs additional memory use, especially bandwidth use, by buffered KVs, it is more than offset by the savings from updating the linear attention state less frequently. As shown in §3.2, by balancing the increased memory reads to buffered keys and values against the reduced memory writes from less frequent state updates, `KVBuffer` minimizes the average memory access per decoding step with a buffer size of $2\sqrt{d}$, where d is the hidden dimension. This memory efficiency, however, comes with a latency trade-off. When the buffer is not full, `KVBuffer` avoids updating the linear attention state and thus reduces per-token decoding latency. Once the buffer is full, `KVBuffer` must flush the buffered KVs and update the linear attention state, introducing latency for that decoding step. Since the state is updated with buffered KVs in parallel on GPU, the latency remains modest and does not scale linearly with the buffer size (See §3.2). For speculative decoding, instead of recurrently computing and creating a temporary linear attention state for each draft token, `KVBuffer` supports parallel verification of multiple draft tokens by buffering their corresponding keys and values, followed by a single update to the linear attention state using only the accepted tokens. For contexts with d or fewer tokens, `KVBuffer` buffers all KVs and directly computes the attention layer output from KVs, *without* ever creating or updating a linear attention state in memory. Note in this case, the buffer size can be up to d, instead of $2\sqrt{d}$.

We implement `KVBuffer` in SGLang for Qwen3.5 and Nemotron-3 series models. Qwen3.5 and Nemotron-3 both adopt a hybrid architecture that incorporates Gated DeltaNet and Mamba-2 as their linear attention modules, respectively. Experimental results show that `KVBuffer` speedup batch decoding by up to $1.47\times$. For speculative decoding, `KVBuffer` reduces verification latency and increases the maximum number of serving requests as the number of draft tokens increases, improving end-to-end serving throughput by up to $1.9\times$. Finally, we demonstrate that KV-only decoding is more efficient than both recurrent and chunkwise decoding for short-context requests.

2 Background

2.1 Linear Attention and Its Computation Forms

Let the sequence length be L and the hidden dimension be d. Standard softmax attention computes the attention output as:

$$\begin{aligned}\mathbf{Q}, \mathbf{K}, \mathbf{V} &= \mathbf{X}\mathbf{W}_Q, \mathbf{X}\mathbf{W}_K, \mathbf{X}\mathbf{W}_V \\ \mathbf{O} &= \text{Softmax}((\mathbf{Q}\mathbf{K}^T) \odot \mathbf{M})\mathbf{V}\end{aligned}$$

where $\mathbf{X} \in \mathbb{R}^{L\times d}$ is the input, $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V \in \mathbb{R}^{d\times d}$ are learnable parameters, and $\mathbf{M}$ denotes the causal mask defined as $\mathbf{M}_{ij} = 1$ when $j \leq i$, otherwise $-\infty$. During inference, softmax attention needs to store keys and values of all previous tokens, gradually increasing the memory occupation and access with regard to the context length.

Linear attention[6] removes the softmax operation and computes attention output as follows. For simplicity, we omit the normalization and query/key feature maps.

$$\begin{aligned}\mathbf{Q}, \mathbf{K}, \mathbf{V} &= \mathbf{X}\mathbf{W}_Q, \mathbf{X}\mathbf{W}_K, \mathbf{X}\mathbf{W}_V \\ \mathbf{O} &= ((\mathbf{Q}\mathbf{K}^T) \odot \mathbf{M})\mathbf{V} \end{aligned} \tag{1}$$

We refer to Eq. 1 as the *parallel form* of linear attention, because the attention outputs of all tokens can be computed simultaneously. For a single decoding step t, the same computation can be written as:

$$\boldsymbol{q}_t, \boldsymbol{k}_t, \boldsymbol{v}_t = \boldsymbol{x}_t \mathbf{W}_Q, \boldsymbol{x}_t \mathbf{W}_K, \boldsymbol{x}_t \mathbf{W}_V$$
$$\boldsymbol{o}_t = \sum_{i=0}^{t} \boldsymbol{q}_t \boldsymbol{k}_i^T \boldsymbol{v}_i \tag{2}$$

where $\boldsymbol{x}_t, \boldsymbol{o}_t, \boldsymbol{q}_t, \boldsymbol{k}_t, \boldsymbol{v}_t \in \mathbb{R}^{1\times d}$ denote the input, output, query, key, and value vectors of the t-th token. For convenience, we also refer Eq. 2 as parallel form.

Eq. 2 can also be computed *recurrently* by maintaining a fixed-size state:

$$\mathbf{S}_t = \mathbf{S}_{t-1} + \boldsymbol{k}_t^T \boldsymbol{v}_t$$
$$\boldsymbol{o}_t = \boldsymbol{q}_t \mathbf{S}_t \tag{3}$$

where $\mathbf{S}_t \in \mathbb{R}^{d\times d}$ is the linear attention state at step t. Compared with the parallel form, the *recurrent form* avoids storing and accessing all previous keys and values. As a result, it has constant memory access and decoding latency, which makes it attractive for long-context inference.

By expanding the recurrent form over m steps, we can derive the *chunkwise form* [21] as follows:

$$\boldsymbol{o}_j = \boldsymbol{q}_j \mathbf{S}_{t-m} + \sum_{i=t-m+1}^{j} \boldsymbol{q}_j \boldsymbol{k}_i^T \boldsymbol{v}_i, \quad \text{for } j = t-m+1, \ldots, t \tag{4}$$

$$\mathbf{S}_t = \mathbf{S}_{t-m} + \sum_{i=t-m+1}^{t} \boldsymbol{k}_i^T \boldsymbol{v}_i \tag{5}$$

The chunkwise form computes the attention output $\boldsymbol{o}_j$ for token j using both the linear attention state of step $t-m$ and the keys and values of tokens $t-m+1$ through j, as shown in Eq. 4. After every m decoding steps, it updates the state using the keys and values of the most recent m tokens, as shown in Eq. 5. Therefore, it allows state updates to be deferred and applied in batches, which is useful for reducing average memory access during decoding. The parallel and recurrent forms can be considered special cases of this chunkwise form.

Several variants[21, 20, 2, 24] of linear attention have been developed to improve its long-context retrieval performance. The three computation forms discussed above are also applicable to these variants. In this work, we use Gated Delta Networks (GDN)[20] and Mamba-2[2] for evaluation, since they have been widely adopted in recent hybrid models[15, 24]. We provide more details on GDN and Mamba-2 in Appendix A.2 and A.3, repsectively.

2.2 Serving with Linear Attention

When serving linear attention-based models, existing serving systems, such as vLLM and SGLang, allocate and maintain a linear attention state in the memory pool for each request. After prefilling the prompt into the linear attention state, they recurrently update the state and compute the attention output by Eq. 3, i.e., using the recurrent form. Moreover, in speculative decoding, the serving system recurrently computes a temporary linear attention state for each draft token and stores these temporary states in the state pool, which incurs additional consumption of N state slots, where N is the number of draft tokens.

As shown in Table 1, the three computation forms of linear attention have different trade-offs among memory storage, memory read and memory write during decoding. The parallel form stores the keys and values of all previous tokens, so both its storage and memory reads grow with the context length L. However, it avoids maintaining the linear attention state, making it more efficient when the context length $L < d$. The recurrent form maintains a fixed-size state, giving constant storage and memory access with respect to the context length. This property makes it suitable for long-context decoding and thus is adopted in existing serving systems during decoding. However, recurrent decoding requires reading and writing the full state in every decoding step, which is expensive

Table 1: Memory storage and average per-token memory access during decoding for different forms of linear attention computation. L denotes the context length, d the hidden dimension, and m the chunk size. Memory access is measured in bytes, assuming that the linear attention state is stored in FP32 and $\boldsymbol{q}, \boldsymbol{k}, \boldsymbol{v}, \boldsymbol{o}$ are stored in FP16. For the recurrent form, the state update and attention output computation are fused into a single kernel.

Computation Forms	Memory Storage	Memory Access	
		Read	Write
Parallel	$4Ld$	$4Ld + 2d$	$6d$
Recurrent	$4d^2$	$4d^2 + 6d$	$4d^2 + 2d$
Chunkwise	$4d^2 + 4md$	$4(1 + 1/m)d^2 + 2(m + 4)d$	$4d^2/m + 6d$

because the state is d times larger than a per-token KV. Compared to recurrent form, chunkwise form introduces additional memory access from reading the buffered keys and values of recent tokens for each decoding step, but it amortizes state updates across multiple tokens and thus reduces average memory access.

Considering memory access, recurrent form is not always the most efficient choice for serving linear attention. In §3, we introduce `KVBuffer` mechanism. By buffering keys and values, `KVBuffer` allows us to flexibly select the most efficient computation form for different decoding scenarios.

3 `KVBuffer`

We now describe the design of `KVBuffer`, an IO-aware serving mechanism that buffers recently generated keys and values to enable more flexible computation forms for linear attention decoding. We first overview `KVBuffer` and its paged memory management (§3.1). We then show how `KVBuffer` supports chunkwise decoding, which reduces average memory access and decoding latency (§3.2). Next, we describe how `KVBuffer` enables parallel verification of draft tokens in speculative decoding (§3.3). Finally, we demonstrate that `KVBuffer` also supports decoding in parallel form, which computes attention output only with KVs and is more efficient than both recurrent and chunkwise forms for short contexts (§3.4).

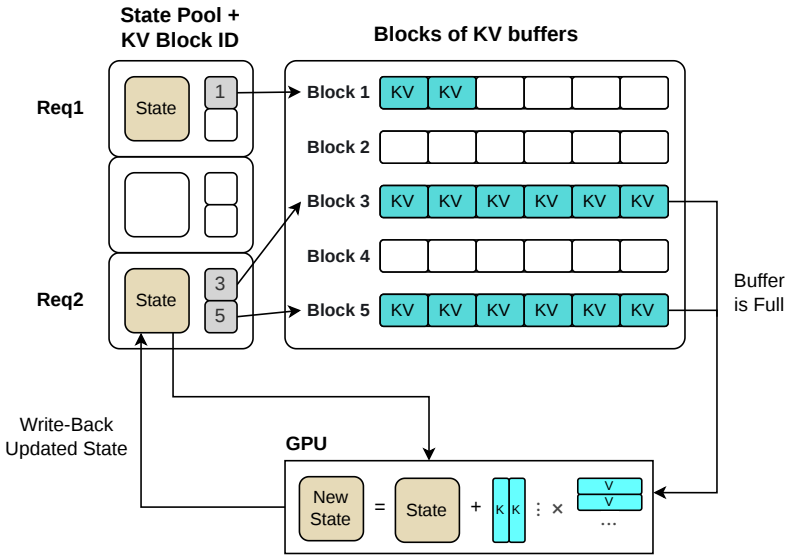


Figure 1: `KVBuffer` Design. We partition the memory for KV buffers into blocks, each of which can store 6 KVs. Each request has two blocks. During decoding, the serving system loads state along with buffered KVs to compute attention output. When the buffer is full, the state is updated with all buffered KVs on GPU and the updated state will be written back to the state slot.

3.1 Design Overview

As shown in Figure 1, in addition to the linear attention state, allocated from a pool like existing serving systems, `KVBuffer` introduces a buffer of KVs for each request. To accommodate dynamically growing KVs, `KVBuffer` draws inspiration from paged attention[7] and allocates the KV buffer for a request from a pool of memory shared system-wide. The basic unit of allocation is a block whose capacity is configured by the user. This design allows the KV buffer of a request to be dynamically sized and avoids memory fragmentation, thereby efficiently supporting linear attention decoding in chunkwise and parallel forms.

3.2 Chunkwise Decoding with `KVBuffer`

By using the linear attention state together with the buffered KVs, `KVBuffer` supports chunkwise decoding, which is more memory efficient than the recurrent form. In the prefilling stage, the serving system computes the prompt into the linear attention state. During decoding, the system loads the state along with buffered KVs to compute the linear attention output using Eq. 4. It places newly generated KVs in the KV buffer. When the KV buffer is full, `KVBuffer` updates the linear attention state with all buffered KVs in batch, according to Eq. 5.

Ideally, when both the recurrent and chunkwise forms are memory-bound, their decoding latency is proportional to the number of memory accesses. Therefore, we can estimate the speedup by the memory access ratio between recurrent and chunkwise forms. Following Table 1, we assume that the linear attention state is stored in FP32, and $\boldsymbol{q}, \boldsymbol{k}, \boldsymbol{v}, \boldsymbol{o}$ vectors are stored in FP16. Without loss of generality, we assume there is a single attention head. The speedup is shown as follows:

$$\text{Speedup}_{\text{chunkwise_decoding}}(m) \approx \frac{4(d+1)}{2d + \frac{4d}{m} + m + 7} \tag{6}$$

where m is the chunk size and d is the hidden dimension. The ideal speedup is maximized at $m^* = 2\sqrt{d}$, which corresponds to the optimal KV buffer size for chunkwise decoding with `KVBuffer`. When the memory bandwidth of GPU is high enough, the chunwise decoding could be compute-bound. Therefore, the buffer size m should be selected based on preliminary sweeping experiments. We evaluate different settings of GPUs and details the buffer size m selection in 4.2.1.

On the other hand, while `KVBuffer` reduces per-token decoding latency, it must flush the buffered KVs when the buffer is full, which incurs additional state update latency for that step. However, since the state update over all buffered KVs is performed in parallel on the GPU, the latency remains modest and does not scale linearly with the buffer size. Following the same assumptions in Table 1, the arithmetic intensity of the state update in Eq. 5 is $m/(4 + 2m/d)$. When m is not very large, e.g., $m = 2\sqrt{d}$, the state update is still memory-bound. Consequently, its latency is approximately proportional to its memory access, $8d^2 + 4md$, where the buffer size dependent term $4md$ is relatively small compared with the state access term $8d^2$ for the optimal buffer size.

3.3 Parallel Verification for Speculative Decoding

Speculative decoding is an important technique to accelerate LLM decoding by verifying multiple draft tokens in parallel [8, 9]. However, existing serving systems for linear attention incur significant memory overhead by maintaining and recurrently computing a temporary linear attention state for each draft token, until accepted tokens are determined.

`KVBuffer` enables memory-efficient parallel verification of draft tokens. During verification, the serving system can buffer KVs of draft tokens, and compute their attention outputs using the chunkwise form in Eq. 7. After having determined accepted tokens, the serving system updates the linear attention state with buffered KVs of only accepted tokens, as in Eq. 8 and Figure 2, reducing overall memory use.

$$\mathbf{O} = \mathbf{Q}\mathbf{S}_{t-j} + ((\mathbf{Q}\mathbf{K}^T) \odot \mathbf{M})\mathbf{V} \tag{7}$$

$$\mathbf{S}_t = \mathbf{S}_{t-j} + \mathbf{K}_{acc}^T \mathbf{V}_{acc} \tag{8}$$

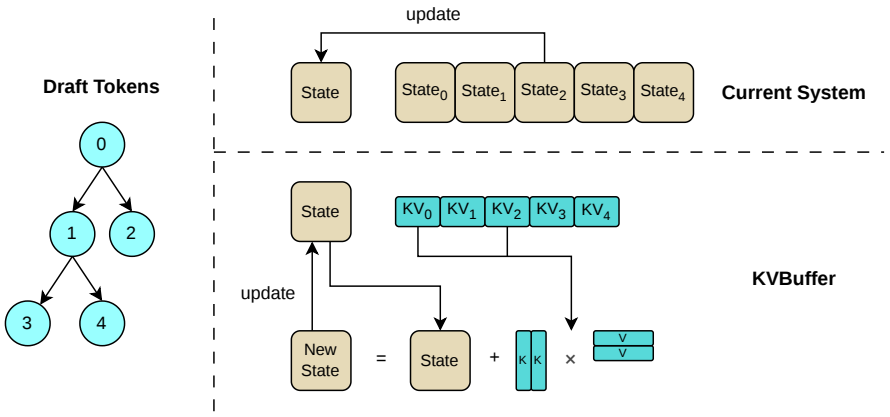


Figure 2: Speculative Decoding with `KVBuffer`. Existing serving systems allocate a temporary linear attention state for each draft token and use the recurrent form for speculative decoding verification. After determining that the accepted draft tokens are 0 and 2, the state of this request is replaced by the temporary state_2. In contrast, `KVBuffer` buffers the KV for each draft tokens and updates the state with KVs of only accepted tokens, reducing overall memory use.

Without loss of generality, we suppose that m draft tokens are sequential and all accepted. The arithmetic intensities of attention output computation and state update in Eq. 7 and Eq. 8 are $\frac{2md^2+2m(m+1)d}{4d^2+12md}$ and $\frac{m}{4+2m/d}$, respectively, both of which indicate memory-bound operations when m is relatively small. Therefore, the expected speedup can be approximated by the ratio of memory access, given by Eq. 9, which approaches $\frac{m+1}{3}$ when $d >> m$.

$$\text{Speedup}_{\text{parallel_verify}}(m) \approx \frac{(m+1)d+2m}{3d+4m} \tag{9}$$

Note that for the recurrent form, we estimate the memory access by assuming that the verification steps are fused into a single kernel and we only read the initial state S_{t-j} once because draft tokens are sequential. For $m = 2$, speculative decoding with buffered verification has approximately the same runtime as the recurrent form. For $m > 2$ and $d >> m$, the speedup increases linearly with m.

Moreover, `KVBuffer` reduces the per-request memory footprint during verification. Instead of maintaining a temporary linear attention state for each draft token, `KVBuffer` only buffers the draft keys and values and verifies them in parallel. Since state is much larger than KV, this reduction allows the serving system to support approximately $m\times$ more concurrent requests within the same memory budget. As a result, `KVBuffer` also improves overall serving throughput.

3.4 KV-only Decoding for Short Contexts

As shown in Table 1, for contexts with d tokens or fewer, storing and accessing all keys and values of previous tokens are more memory-efficient than both maintaining the linear attention state with recurrent and chunkwise forms. Therefore, we prefer to compute the attention output with parallel form when context length $L < d$.

The paged memory management of `KVBuffer` naturally supports the parallel form, which requires to buffer keys and values of up to d tokens. `KVBuffer` allows dynamic allocation and deallocation of KV blocks to avoid memory fragmentation, similar to paged KV cache management.

After prefilling, the serving system buffers keys and values of all tokens. While decoding, the serving system buffers the newly generated key and value and computes attention output in parallel form, as in Eq. 2. Once the context length $L \geq d$, the serving system can compress the keys and values into linear attention state as Eq. 5 and then turn to chunkwise decoding as discussed above.

Given context length of L, the arithmetic intensity of parallel computation form in Eq. 2 is $\frac{L}{L+2}$, which is close to 1. It indicates that the parallel form is also memory-bound. Therefore, the speedup can be approximated by the ratio of average memory access between parallel and chunkwise form,

given by Eq. 10. When $m = 2\sqrt{d}$, Eq. 10 simplifies to $\frac{d+2\sqrt{d}+7/2}{L+2}$. The speedup decreases monotonically as L increases.

$$\text{Speedup}_{\text{kv_only}}(m) \approx \frac{d + 2d/m + m/2 + 7/2}{L + 2} \tag{10}$$

The state size is often closely related to the memory retrieval capability of the model. However, increasing the state size can be challenging in practice because recurrent decoding must access the entire linear attention state in every decoding step, regardless of the context length. As a result, large states become prohibitively expensive, especially for short-context requests. By enabling decoding in parallel form, `KVBuffer` makes it feasible to scale the state size while maintaining inference efficiency for short-context decoding, further unlocking the potential of linear attention.

4 Experiment

4.1 Experimental Setup

We implement `KVBuffer` in SGLang v0.5.10 [25] and configure the `KVBuffer` block size according to the decoding scenario. For chunkwise decoding, the block size is equal to the buffer size, which is selected through a parameter sweep. For speculative decoding, we set both the per-request buffer size and block size to the maximum number of draft tokens verified in one iteration. In these two scenarios, the buffer of each request fits into a single `KVBuffer` block. This simplifies buffer management and kernel implementation, as the kernel receives a single block ID per request rather than a variable-length list of block IDs. Moreover, for speculative decoding, we resets the buffer after every verification iteration. This design simplifies memory management by avoiding dynamic block allocation during verification and supports fixed-shape parallel verification compatible with CUDA Graphs. For KV-only decoding, we use a `KVBuffer` block size of 16, which is a common page size in KV-cache management. As a result, each request occupies up to $\lceil m/16 \rceil$ blocks, where m denotes the buffer size.

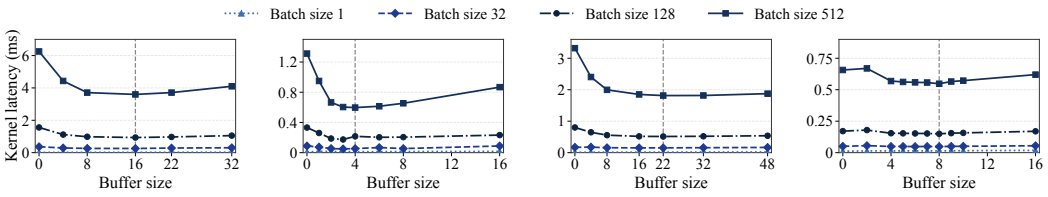
The total number of available blocks is determined by the number of states allocated in the state pool, which is controlled by the user-defined memory fraction reserved for states in SGLang[25]. For KV-only decoding, `KVBuffer` does not initialize the state pool because no linear-attention state is maintained. Instead, the total number of blocks is computed directly from the available memory, user-defined memory fraction and the block size.

We evaluate `KVBuffer` on Qwen3.5 and Nemotron-3 series models. Qwen3.5 and Nemotron-3 both adopt a hybrid architecture and use Gated Delta Networks (GDN)[20] and Mamba-2[2] as their linear attention module, respectively. For GDN, we buffer the decay factor α, key $\boldsymbol{k}$, and delta value $\boldsymbol{u}$ for each token to support chunkwise computation, as described in Appendix A.2. For Mamba-2, we buffer the decay factor α, state-expansion vector B and discretization-scaled input Δx as described in Appendix A.3. Unless otherwise specified, the linear attention state is stored in FP32, while buffered keys and values are stored in BF16. For speculative decoding, we use Multi-Token-Prediction as the draft model for Qwen3.5-4B and evaluate on the ShareGPT dataset.

Table 2: Models and serving hardware. Head dimension denotes the Mamba-2 channel width or the GDN key/value width; both Mamba-2 models have state dimension 128.

Model	Linear Attention Head Dim	Precision	Hardware
Nemotron-3-Nano-4B	80	BF16	1 × L40S
Nemotron-3-Super-120B-A12B	64	NVFP4	1 × B300
Qwen3.5-4B	128	BF16	1 × L40S
Qwen3.5-122B-A10B	128	NVFP4	1 × B300

We conduct experiments on NVIDIA L40S and B300 GPUs, with the hardware configuration for each model listed in Table 2. We implement `KVBuffer`-related kernels in Triton, including kernels for chunkwise decoding, parallel verification in speculative decoding, batched state update, and decoding in parallel form. Additional experimental results are presented in Appendix A.4.



(a) Nemotron-3 Nano-4B (b) Nemotron-3 Super-120B-A12B (c) Qwen3.5 4B (d) Qwen3.5 122B-A10B

Figure 3: Kernel latency of four models with different buffer sizes m and batch sizes. $m = 0$ denotes recurrent decoding. For $m > 0$, latency is averaged over a full buffer cycle and includes the amortized state-update cost. Vertical dashed lines mark the best tested buffer size at batch size 512 for each model, which is used in the following kernel latency and end-to-end batch inference evaluations.

4.2 Experimental Results

4.2.1 Chunkwise Decoding with `KVBuffer`

We first sweep over different buffer sizes m for all four models, as shown in Figure 3. We measure kernel latency at batch sizes 1, 32, 128, and 512 for a full buffer cycle, including the amortized state-update cost. As shown in the Figure 3, latency initially decreases as state updates are amortized over more decoding steps, then increases as accessing buffered tokens becomes more expensive, consistent with the trade-off in §3.2. The best tested buffer sizes are 16, 4, 22, and 8 for Nemotron-3-Nano, Nemotron-3-Super, Qwen3.5-4B, and Qwen3.5-122B, respectively. For Qwen3.5-4B, the best tested buffer size $m = 22$ is close to $2\sqrt{128} \approx 22.6$. However, Qwen3.5-122B uses the same head dimension but reaches its minimum at $m = 8$ on B300. The smaller optima on B300 are consistent with the possibility discussed in §3.2 that high memory bandwidth makes flush operation compute-bound. As shown in Figure 4, chunkwise decoding with the best tested buffer sizes achieves kernel latency reductions of $0.42\times$, $0.55\times$, $0.45\times$, and $0.17\times$ the recurrent-decoding latency for the four models, respectively.

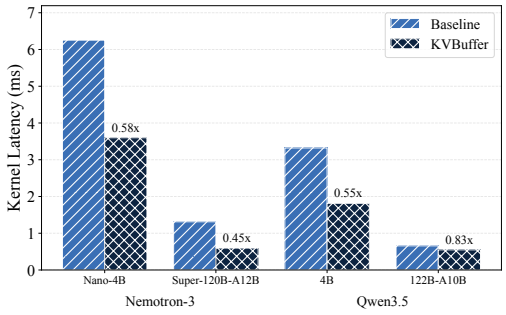


Figure 4: Kernel latency at batch size 512, using the best tested buffer size for each model. `KVBuffer` latency is averaged over a full buffer cycle and includes the state-update cost.

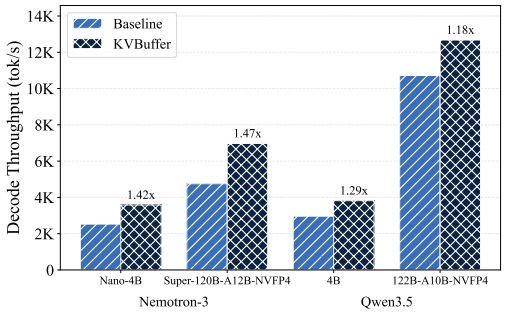


Figure 5: End-to-end batch inference decoding throughput for different models. We use a batch size of 128 for the 4B models and 512 for the 120B and 122B models.

We evaluate end-to-end batch inference with an input length of 256 tokens and an output length of 1,024 tokens. For each model, we select the buffer size with the lowest kernel latency at batch size 512. As shown in Figure 5, `KVBuffer` improves recurrent decoding throughput by $1.18\times$–$1.47\times$ across four models covering both GDN and Mamba-2 architectures and different model scales. Qwen3.5 achieves a smaller speedup than Nemotron-3 at similar model scales. The reason is that Qwen3.5 has a smaller linear attention state compared with Nemotron-3. Consequently, linear attention operations account for a smaller fraction of overall runtime, reducing the end-to-end benefit from the KVBuffer optimization.

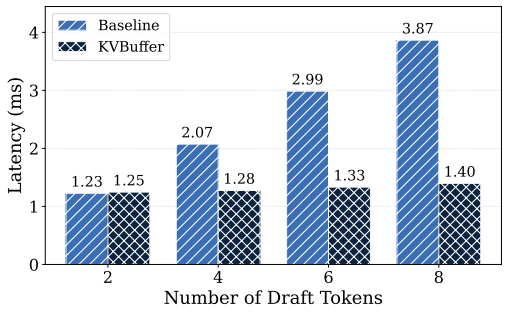


Figure 6: Kernel latency of speculative decoding verification. The latency of `KVBuffer` includes both attention output computation and the state update. `KVBuffer` incurs only a modest latency increase by verifying draft tokens in parallel.

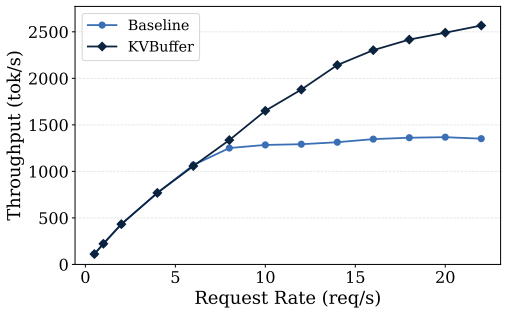


Figure 7: End-to-end serving throughput with speculative decoding. By avoiding the storage of temporary states for draft tokens, `KVBuffer` sustains higher request rates and improves overall throughput.

4.2.2 Parallel Verification for Speculative Decoding

Parallel Verification Latency We next evaluate `KVBuffer` for speculative decoding verification on Qwen3.5-4B model. Figure 6 compares the kernel latency of verifying different numbers of draft tokens. For recurrent decoding, verification latency increases linearly with the number of draft tokens because the system must compute and store a temporary state for each draft token. In contrast, `KVBuffer` verifies draft tokens by buffering their keys and values and computing attention outputs in chunkwise form, which incurs only modest additional latency as the number of draft tokens increases. When verifying 8 draft tokens, `KVBuffer` achieves a $2.77\times$ kernel speedup, closely matching the analysis in §3.3, which predicts an approximately $3\times$ speedup.

End-to-end Throughput We also evaluate the end-to-end serving throughput of `KVBuffer` under speculative decoding. Figure 7 shows the throughput under different request rates, with the number of draft tokens set to 4. Because recurrent verification must store a temporary state for each draft token, it supports fewer concurrent requests than chunkwise verification. As a result, `KVBuffer` increases the maximum number of serving requests by $5\times$ and sustains higher request rates, achieving up to a $1.9\times$ throughput improvement.

4.2.3 KV-only Decoding for Short Contexts

Finally, we evaluate different computation forms for short-context requests with a batch size of 128. As shown in Figure 8, the latency of KV-only decoding gradually increases with context length because it must read more buffered keys and values. As discussed in §3.4, its speedup over chunkwise decoding decreases as the context grows; for $d = 128$ and $m = 16$, the analysis predicts a speedup of 1.0 at $L \approx 154$. The measured trend is consistent with this prediction, supporting the advantage of KV-only decoding for short contexts, with a crossover near the head dimension d.

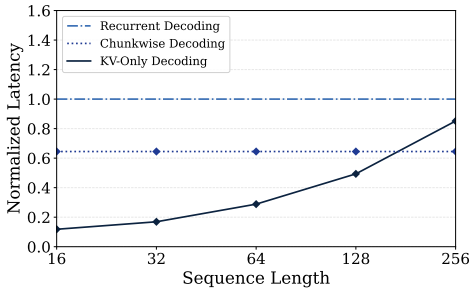


Figure 8: Normalized kernel latency of different computation forms at batch size 128.

5 Related Work

Linear Attention Linear attention has been widely studied as an efficient alternative to softmax attention for long-context inference. By removing the softmax operation, linear attention reduces the quadratic dependence on sequence length and enables recurrent decoding with a fixed-size state. However, vanilla linear[6] attention suffers from degraded long-context retrieval and model quality compared with softmax attention. Recent variants mitigate this problem by introducing more expressive memory-control mechanisms, including data-independent decay mechanisms[5, 17, 12,

18], data-dependent decay mechanisms[4, 2, 21, 13] and delta rule[22, 20, 16]. These advances have made linear attention increasingly practical for real-world long-context applications.

Hybrid Architecture Recent LLMs[15, 24, 10] increasingly adopt hybrid architectures that interleave linear attention layers with standard softmax-attention layers. These hybrid architectures preserve the strong memory retrieval capability of softmax attention while using linear attention layers to reduce the memory storage and decoding cost for long-context inference. Models such as Qwen3-Next[15] and Kimi-Linear[24] demonstrate that hybrid architectures can achieve favorable quality-efficiency trade-offs, making linear attention an increasingly important building block for LLMs. In this work, we use Qwen3.5 and Nemotron-3 as representative hybrid architecture models for our evaluation.

Serving for Linear Attention Modern LLM serving systems, such as vLLM[7] and SGLang[25], improve inference throughput through techniques such as paged KV cache management[7], continuous batching[23], prefix caching[25], and speculative decoding[8, 9]. These techniques are primarily designed for Transformer models with softmax attention, where serving is KV cache-centric. In contrast, linear attention maintains a fixed-size state and no longer preserves the keys and values for all previous tokens, introducing a different memory-management problem for serving systems. Several recent works have begun to study serving techniques for LLMs with hybrid architectures. For prefix caching, Marconi[11] proposes efficient prefix-cache management for hybrid LLMs by checkpointing the linear attention state at appropriate positions. For prefill-decoding disaggregation, Prefill-as-a-Service[14] explores cross-cluster prefill offloading for hybrid linear-attention models by exploiting their reduced memory footprint. For speculative decoding, STree[19] proposes a tree-based parallel verification algorithm for SSM models[2, 4]. However, it does not investigate the linear attention models and the corresponding serving system support required for chunkwise form computation. In this work, we study the memory management for linear attention serving and show that it can improve memory efficiency across diverse decoding scenarios.

6 Discussion and Limitation

Serving for Different Computation Forms `KVBuffer` enables linear attention to be served using different computation forms. However, dynamically selecting the most efficient form for each request is non-trivial in practice. Chunkwise decoding and KV-only decoding require different kernels and have different batching requirements, so switching computation forms based on prompt length can introduce additional scheduling overhead. In this work, we do not dynamically route requests across different decoding forms during serving. As hybrid models continue to scale and the hidden dimension d increases, KV-only decoding may become increasingly important. One possible direction is to disaggregate short-context and long-context requests onto separate servers. Another direction is to interleave different decoding forms within the same batch, similar to chunked prefill[1]. We leave the design of such dynamic scheduling mechanisms to future work.

KV-Based Prefix Caching Existing serving systems typically use the linear attention state as the checkpoint for prefix caching. However, because the linear attention state is so large, checkpointing is usually performed at coarse granularity, making it difficult to split or reuse prefixes at arbitrary positions. In this work, we show that the linear attention state can be reconstructed from buffered keys and values. This observation suggests an alternative prefix-caching design: instead of caching only linear attention states, serving systems may cache keys and values for selected prefixes and reconstruct the state on demand. Such a design could enable finer-grained prefix reuse and more flexible memory management for linear attention-based models.

7 Conclusion

In this paper, we present `KVBuffer`, an IO-aware serving mechanism for linear attention. By buffering recent keys and values, `KVBuffer` enables flexible computation forms across different decoding scenarios, including chunkwise decoding, speculative decoding verification, and short-context decoding. These forms reduce unnecessary memory access and substantially improve the efficiency of serving hybrid models.

Acknowledgments

This work is supported in part by Yale University and by National Science Foundation (NSF) Athena AI Institute (Award #2112562)

References

[1] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, and Ramachandran Ramjee. SARATHI: efficient LLM inference by piggybacking decodes with chunked prefills. *CoRR*, abs/2308.16369, 2023.

[2] Tri Dao and Albert Gu. Transformers are ssms: Generalized models and efficient algorithms through structured state space duality. In Ruslan Salakhutdinov, Zico Kolter, Katherine A. Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp, editors, *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, Proceedings of Machine Learning Research, pages 10041–10071. PMLR / OpenReview.net, 2024.

[3] DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *CoRR*, abs/2501.12948, 2025.

[4] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *CoRR*, abs/2312.00752, 2023.

[5] Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022.

[6] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, Proceedings of Machine Learning Research, pages 5156–5165. PMLR, 2020.

[7] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace, editors, *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 611–626. ACM, 2023.

[8] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, Proceedings of Machine Learning Research, pages 19274–19286. PMLR, 2023.

[9] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE: speculative sampling requires rethinking feature uncertainty. In Ruslan Salakhutdinov, Zico Kolter, Katherine A. Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp, editors, *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, Proceedings of Machine Learning Research, pages 28935–28948. PMLR / OpenReview.net, 2024.

[10] MiniMax. Minimax-m1: Scaling test-time compute efficiently with lightning attention. *CoRR*, abs/2506.13585, 2025.

[11] Rui Pan, Zhuang Wang, Zhen Jia, Can Karakus, Luca Zancato, Tri Dao, Yida Wang, and Ravi Netravali. Marconi: Prefix caching for the era of hybrid llms. In Matei Zaharia, Gauri Joshi, and Yingyan (Celine) Lin, editors, *Proceedings of the Eighth Conference on Machine Learning and Systems, MLSys 2025, Santa Clara, CA, USA, May 12-15, 2025*. OpenReview.net/mlsys.org, 2025.

[12] Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Chung, Leon Derczynski, Xingjian Du, Matteo Grella, Kranthi Kiran GV, Xuzheng He, Haowen Hou, Przemyslaw Kazienko, Jan Kocon, Jiaming Kong, Bartlomiej Koptyra, Hayden Lau, Jiaju Lin, Krishna Sri Ipsit Mantri, Ferdinand Mom, Atsushi Saito, Guangyu Song, Xiangru Tang, Johan S. Wind, Stanislaw Wozniak, Zhenyuan Zhang, Qinghua Zhou, Jian Zhu, and Rui-Jie Zhu. RWKV: reinventing rnns for the transformer era. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*, Findings of ACL, pages 14048–14077. Association for Computational Linguistics, 2023.

[13] Bo Peng, Daniel Goldstein, Quentin Anthony, Alon Albalak, Eric Alcaide, Stella Biderman, Eugene Cheah, Xingjian Du, Teddy Ferdinan, Haowen Hou, Przemyslaw Kazienko, Kranthi Kiran GV, Jan Kocon, Bartlomiej Koptyra, Satyapriya Krishna, Ronald McClelland Jr., Niklas Muennighoff, Fares Obeid, Atsushi Saito, Guangyu Song, Haoqin Tu, Stanislaw Wozniak, Ruichong Zhang, Bingchen Zhao, Qihang Zhao, Peng Zhou, Jian Zhu, and Rui-Jie Zhu. Eagle and finch: RWKV with matrix-valued states and dynamic recurrence. *CoRR*, abs/2404.05892, 2024.

[14] Ruoyu Qin, Weiran He, Yaoyu Wang, Zheming Li, Xinran Xu, Yongwei Wu, Weimin Zheng, and Mingxing Zhang. Prefill-as-a-service: KVCache of next-generation models could go cross-datacenter. *CoRR*, abs/2604.15039, 2026.

[15] Qwen Team. Qwen3-next-80b-a3b. `https://qwen.ai/blog?id=4074cca80393150c248e508aa62983f9cb7d27cd`, September 2025. Blog post, accessed May 5, 2026.

[16] Julien Siems, Timur Carstensen, Arber Zela, Frank Hutter, Massimiliano Pontil, and Riccardo Grazzi. Deltaproduct: Improving state-tracking in linear rnns via householder products. In Danielle Belgrave, Cheng Zhang, Laura N. Montoya, Hsuan-Tien Lin, Razvan Pascanu, Piotr Koniusz, Marzyeh Ghassemi, Nancy Chen, Iván Vladimir Meza Ruíz, and Arturo Loaiza-Bonilla, editors, *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2025, NeurIPS 2025, San Diego, CA, USA, December 2-7, 2025 / Mexico City, Mexico, November 30 - December 5, 2025*, 2025.

[17] Jimmy T. H. Smith, Andrew Warrington, and Scott W. Linderman. Simplified state space layers for sequence modeling. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.

[18] Yutao Sun, Li Dong, Shaohan Huang, Shuming Ma, Yuqing Xia, Jilong Xue, Jianyong Wang, and Furu Wei. Retentive network: A successor to transformer for large language models. *CoRR*, abs/2307.08621, 2023.

[19] Yangchao Wu, Zongyue Qin, Alex Wong, and Stefano Soatto. Stree: Speculative tree decoding for hybrid state-space models. *CoRR*, abs/2505.14969, 2025.

[20] Songlin Yang, Jan Kautz, and Ali Hatamizadeh. Gated delta networks: Improving mamba2 with delta rule. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025.

[21] Songlin Yang, Bailin Wang, Yikang Shen, Rameswar Panda, and Yoon Kim. Gated linear attention transformers with hardware-efficient training. In Ruslan Salakhutdinov, Zico Kolter, Katherine A. Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp, editors, *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, Proceedings of Machine Learning Research, pages 56501–56523. PMLR / OpenReview.net, 2024.

[22] Songlin Yang, Bailin Wang, Yu Zhang, Yikang Shen, and Yoon Kim. Parallelizing linear transformers with the delta rule over sequence length. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang, editors, *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024.

[23] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for transformer-based generative models. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 521–538. USENIX Association, 2022.

[24] Yu Zhang, Zongyu Lin, Xingcheng Yao, Jiaxi Hu, Fanqing Meng, Chengyin Liu, Xin Men, Songlin Yang, Zhiyuan Li, Wentao Li, Enzhe Lu, Weizhou Liu, Yanru Chen, Weixin Xu, Longhui Yu, Yejie Wang, Yu Fan, Longguang Zhong, Enming Yuan, Dehao Zhang, Yizhi Zhang, T. Y. Liu, Haiming Wang, Shengjun Fang, Weiran He, Shaowei Liu, Yiwei Li, Jianlin Su, Jiezhong Qiu, Bo Pang, Junjie Yan, Zhejun Jiang, Weixiao Huang, Bohong Yin, Jiacheng You, Chu Wei, Zhengtao Wang, Chao Hong, Yutian Chen, Guanduo Chen, Yucheng Wang, Huabin Zheng, Feng Wang, Yibo Liu, Mengnan Dong, Zheng Zhang, Siyuan Pan, Wenhao Wu, Yuhao Wu, Longyu Guan, Jiawen Tao, Guohong Fu, Xinran Xu, Yuzhi Wang, Guokun Lai, Yuxin Wu, Xinyu Zhou, Zhilin Yang, and Yulun Du. Kimi linear: An expressive, efficient attention architecture. *CoRR*, abs/2510.26692, 2025.

[25] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark W. Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang, editors, *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024.

A Appendix

A.1 Additional Discussion

Sampling `KVBuffer` can also benefit sampling-based decoding algorithms. For example, in the beam search, the serving system needs to maintain multiple candidate branches. If each branch is served recurrently, the system must compute and store a separate linear attention state for each candidate branch, resulting in substantial memory overhead. With `KVBuffer`, the system can instead buffer the keys and values of candidate tokens and update the linear attention state only after the accepted branch is determined.

Aligning Pretraining and Inference Computation Linear attention models are usually pretrained with chunkwise computation[21]. However, existing serving systems typically use the recurrent form during inference, creating a mismatch between the computation form used in pretraining and that used in inference. For long-context generation, this mismatch can lead to accuracy degradation or instability during RL post-training[3]. Therefore, `KVBuffer` not only improves serving efficiency, but also helps align inference and pretraining computation, potentially improving inference stability.

A.2 Gated Delta Networks

Gated Delta Networks (GDN)[20] is among the most widely adopted variants of linear attention in recent LLMs with hybrid architecture. It incorporates a data-dependent gating mechanism α_t, inspired by Mamba2[2], together with the delta rule[22] to selectively update the state, leading to improved performance.

The recurrent computation form of GDN is defined as:

$$\tilde{\mathbf{S}}_{t-1} = \alpha_t \mathbf{S}_{t-1} \tag{11}$$

$$\begin{aligned} \mathbf{S}_t &= \tilde{\mathbf{S}}_{t-1} - \boldsymbol{k}_t^T(\boldsymbol{k}_t\tilde{\mathbf{S}}_{t-1}) + \boldsymbol{k}_t^T(\beta_t\boldsymbol{v}_t + (1-\beta_t)\boldsymbol{k}_t\tilde{\mathbf{S}}_{t-1}) \\ &= ((\mathbf{I} - \beta_t\boldsymbol{k}_t^T\boldsymbol{k}_t))\tilde{\mathbf{S}}_{t-1} + \beta_t\boldsymbol{k}_t^T\boldsymbol{v}_t \end{aligned} \tag{12}$$

Here, α_t and β_t denote the decay factor and the learning rate at step t, respectively, and $\tilde{\mathbf{S}}_t$ is the gated intermediate state. Eq. 11 implements the data-dependent gating mechanism that enables the forgetting of long-term memory. Eq. 12 corresponds to the delta rule. Specifically, it removes the retrieved old value $\boldsymbol{k}_t\tilde{\mathbf{S}}_{t-1}$ from gated state $\tilde{\mathbf{S}}_{t-1}$ and replaces it with a new combination value $(\beta_t\boldsymbol{v}_t + (1-\beta_t)\boldsymbol{k}_t\tilde{\mathbf{S}}_{t-1})$. This formulation makes explicit that the update consists of erasing stale information and writing new content in a controlled manner.

As in linear attention, the recurrent form of GDN maintains a fixed-size state $\mathbf{S}_t$, resulting in constant computational and memory cost with respect to context length. Compared to linear attention, GDN introduces additional gating and updating operations, which enhance expressivity and allow more flexible control over memory updates, leading to substantially better model quality than linear attention.

GDN can also be written in a parallel form:

$$\begin{aligned} \gamma_t &= \prod_{i \in \text{Par}(t)} \alpha_i \\ \mathbf{\Gamma}_{ij} &= \frac{\gamma_i}{\gamma_j}, i \geq j \text{ and } \mathbf{M}_{ij} \neq 0, \text{otherwise } 0 \\ \mathbf{A} &= [\mathbf{I} + \text{strictLower}(\text{Diag}(\boldsymbol{\beta})(\mathbf{\Gamma} \odot \mathbf{K}^T\mathbf{K}))]^{-1} \\ \tilde{\mathbf{K}} &= \text{Diag}(\boldsymbol{\gamma})\mathbf{A}\text{Diag}(\boldsymbol{\beta})\mathbf{K}; \tilde{\mathbf{V}} = \mathbf{A}\text{Diag}(\boldsymbol{\beta})\mathbf{V} \\ \mathbf{O} &= (\mathbf{Q}\mathbf{K}^T \odot \mathbf{\Gamma})\tilde{\mathbf{V}} \end{aligned} \tag{13}$$

At the single-token level, this parallel formulation can be expressed as:

$$
\begin{aligned}
\boldsymbol{u}_t &= \beta_t \boldsymbol{v}_t - \beta_t (\Sigma_{i=1}^{j} \frac{\gamma_t}{\gamma_{t-i}} \boldsymbol{k}_t \boldsymbol{k}_{t-i}^T \boldsymbol{u}_i) \\
\boldsymbol{o}_t &= \Sigma_{i=0}^{j} \frac{\gamma_t}{\gamma_{t-i}} \boldsymbol{q}_t \boldsymbol{k}_{t-i}^T \boldsymbol{u}_i
\end{aligned} \tag{14}
$$

where $\text{Par}(t)$ denotes the set of prefix tokens, including the token t, and $\mathbf{\Gamma}$ is the gated mask encoding the cumulative effect of α.

In addition to the recurrent and parallel forms, GDN also supports a chunkwise computation form, as shown in Eq. 15:

$$
\begin{aligned}
\gamma_t &= \prod_{i \in \text{Par}(t)} \alpha_i \\
\mathbf{\Gamma}_{ij} &= \frac{\gamma_i}{\gamma_j}, i \geq j \text{ and } \mathbf{M}_{ij} \neq 0, \text{otherwise } 0 \\
\mathbf{A} &= [\mathbf{I} + \text{strictLower}(\text{Diag}(\boldsymbol{\beta})(\mathbf{\Gamma} \odot \mathbf{K}^T \mathbf{K}))]^{-1} \\
\tilde{\mathbf{K}} &= \text{Diag}(\boldsymbol{\gamma}) \mathbf{A} \text{Diag}(\boldsymbol{\beta}) \mathbf{K}; \tilde{\mathbf{V}} = \mathbf{A} \text{Diag}(\boldsymbol{\beta}) \mathbf{V} \\
\mathbf{U} &= \tilde{\mathbf{V}} - \tilde{\mathbf{K}} \mathbf{S}_{t-j-1} \\
\mathbf{O} &= \text{Diag}(\boldsymbol{\gamma}) \mathbf{Q} \mathbf{S}_{t-j-1} + (\mathbf{Q} \mathbf{K}^T \odot \mathbf{\Gamma}) \mathbf{U}
\end{aligned} \tag{15}
$$

The corresponding single-token chunkwise computation is:

$$
\begin{aligned}
\gamma_t &= \prod_{i=t-m}^{t} \alpha_i \\
\boldsymbol{u}_t &= \beta_t \boldsymbol{v}_t - \beta_t (\gamma_t \boldsymbol{k}_t \mathbf{S}_{t-j-1} + \Sigma_{i=1}^{j} \frac{\gamma_t}{\gamma_{t-i}} \boldsymbol{k}_t \boldsymbol{k}_{t-i}^T \boldsymbol{u}_i) \\
\boldsymbol{o}_t &= \gamma_t \boldsymbol{q}_t \mathbf{S}_{t-j-1} + \Sigma_{i=0}^{j} \frac{\gamma_t}{\gamma_{t-i}} \boldsymbol{q}_t \boldsymbol{k}_{t-i}^T \boldsymbol{u}_i \\
\mathbf{S}_t &= \gamma_t \mathbf{S}_{t-j-1} + \Sigma_{i=0}^{j} \frac{\gamma_t}{\gamma_{t-i}} \boldsymbol{k}_i^T \boldsymbol{u}_i
\end{aligned} \tag{16}
$$

For GDN, we need to buffer the decay factor α_t, key $\boldsymbol{k}_t$ and delta value $\boldsymbol{u}_t$. Since α_t is a scalar, the additional memory overhead is small. As shown in Table 3, the overall storage and memory access of GDN remain close to that of linear attention.

Table 3: Memory storage and average per-token memory access during decoding for different forms of GDN. L denotes the context length, d the hidden dimension, and m the chunk size. Memory access is measured in bytes, assuming that the linear attention state is stored in FP32 and $\boldsymbol{q}, \boldsymbol{k}, \boldsymbol{v}, \boldsymbol{o}$ are stored in FP16. For the recurrent form, the state update and attention output computation are fused into a single kernel.

Computation Forms	Memory Storage	Memory Access	
		Read	Write
Parallel	$4Ld + 2L$	$4Ld + 2d + 2L + 2$	$6d + 2$
Recurrent	$4d^2$	$4d^2 + 6d + 4$	$4d^2 + 2d$
Chunkwise	$4d^2 + 4md + 2m$	$4(1 + 1/m)d^2 + 2(m + 4)d + m + 5$	$4d^2/m + 6d + 2$

Furthermore, based on the memory-access estimates in Table 3, the estimated speedups of GDN for chunkwise decoding, speculative-decoding verification, and KV-only decoding are as follows:

$$\text{Speedup}_{\text{gdn_chunkwise_decoding}} = \frac{8d^2 + 8d + 4}{4d^2 + \frac{8d^2}{m} + 2md + 14d + m + 7} \approx \frac{4(d+1)}{2d + \frac{4d}{m} + m + 7} \tag{17}$$

$$\text{Speedup}_{\text{gdn_parallel_verify}} = \frac{4(m+1)d^2 + 8md + 4m}{12d^2 + 16md + 8m} \approx \frac{(m+1)d + 2m}{3d + 4m} \tag{18}$$

$$\text{Speedup}_{\text{gdn_kv_only}} = \frac{4d^2 + \frac{8d^2}{m} + 2md + 14d + m + 7}{4Ld + 8d + 2L + 4} \approx \frac{d + 2d/m + m/2 + 7/2}{L + 2} \tag{19}$$

These estimates show that the speedups of GDN are close to those of linear attention. This is because GDN only requires buffering additional scalar decay factors, while the dominant storage and memory-access costs remain determined by the key, value, and state tensors.

A.3 Mamba-2

For simplicity, we consider one Mamba-2 head and omit the head index. In Mamba-2, the state-transition matrix has the scalar-times-identity form $A_t = a_t I$. Therefore, we use the scalar a_t in place of A_t for simplicity.

Let x_t denote the token representation, B_t the state-expansion vector, C_t the state-contraction vector, a_t the discrete state-decay factor, and S_t the Mamba-2 state. Mamba-2 recurrently updates its state and computes its output o_t as:

$$S_t = a_t S_{t-1} + B_t x_t^T \tag{20}$$

$$o_t = C_t^T S_t \tag{21}$$

Define the cumulative decay from position s to position u as $\lambda_{t \leftarrow s} = \prod_{r=s+1}^{t} a_r$. Note that $\lambda_{s \leftarrow s} = 1$.

By unrolling the recurrence, we can derive the chunkwise computation form as:

$$o_j = \lambda_{j \leftarrow t-m} C_j^T S_{t-m} + \sum_{i=t-m+1}^{j} \lambda_{j \leftarrow i} (C_j^T B_i) x_i, \text{for} j = t-m+1, ..., t \tag{22}$$

$$S_t = \lambda_{t \leftarrow t-m} S_{t-m} + \sum_{i=t-m+1}^{t} \lambda_{t \leftarrow i} B_i x_i^T \tag{23}$$

Define the lower-triangular decay matrix:

$$\Gamma_{ij} = \lambda_{i \leftarrow j}, i \geq j, \text{otherwise } 0 \tag{24}$$

and stack the token vectors row-wise into X, B, C, we can derive the parallel form as:

$$O = (CB^T \odot \Gamma)X \tag{25}$$

Therefore, Mamba-2 can support `KVBuffer` for the three scenarios we discussed in the paper.

A.4 Additional Experiment

A.4.1 End-to-end Serving Performance

We evaluate end-to-end online serving performance for standard decoding using requests sampled from the ShareGPT dataset. As shown in Table 4, `KVBuffer` provides increasingly clear benefits as the request rate and decode concurrency increase. At moderate to high request rates, it improves throughput by up to 10%, reduces mean end-to-end latency by up to 18%, and reduces mean time per output token by up to 20%.

Table 4: End-to-end standard decoding serving performance on ShareGPT. Each entry is reported as "`KVBuffer`/ baseline", followed by their ratio in parentheses.

Model	Request Rate (req/s)	Throughput (tok/s)	E2E (ms)	TTFT (ms)	TPOT (ms)
Nemotron-3-Nano-4B	1	219.4/219.4 (**1.00**×)	2924.1/2979.2 (**0.98**×)	48.6/49.4 (**0.98**×)	12.91/13.16 (**0.98**×)
	8	1066.2/997.0 (**1.07**×)	5117.9/6104.5 (**0.84**×)	63.7/69.3 (**0.92**×)	23.94/28.89 (**0.83**×)
	16	1206.6/1099.4 (**1.10**×)	6980.9/8515.2 (**0.82**×)	98.0/112.6 (**0.87**×)	36.96/46.32 (**0.80**×)
Nemotron-3-Super-120B-A12B	1	217.8/218.2 (**1.00**×)	4183.8/4081.1 (**1.03**×)	114.9/112.3 (**1.02**×)	19.95/19.59 (**1.02**×)
	8	818.0/788.7 (**1.04**×)	11335.7/12247.5 (**0.93**×)	211.4/222.7 (**0.95**×)	69.42/75.94 (**0.91**×)
	16	941.9/901.4 (**1.04**×)	12122.7/13186.9 (**0.92**×)	251.3/260.6 (**0.96**×)	130.42/145.50 (**0.90**×)
Qwen3.5-4B	1	219.4/219.5 (**1.00**×)	3553.4/3529.5 (**1.01**×)	60.5/63.3 (**0.96**×)	16.20/16.21 (**1.00**×)
	8	1019.5/985.5 (**1.03**×)	5809.5/6345.3 (**0.92**×)	53.8/55.3 (**0.97**×)	28.56/31.29 (**0.91**×)
	16	1141.0/1080.9 (**1.06**×)	7249.8/8256.8 (**0.88**×)	64.7/68.8 (**0.94**×)	41.76/48.05 (**0.87**×)
Qwen3.5-122B-A10B	1	223.2/223.3 (**1.00**×)	1744.5/1679.7 (**1.04**×)	99.7/99.7 (**1.00**×)	8.48/8.16 (**1.04**×)
	8	1223.0/1228.6 (**1.00**×)	6144.6/6218.2 (**0.99**×)	129.1/123.6 (**1.04**×)	35.62/36.17 (**0.98**×)
	16	1503.4/1521.0 (**0.99**×)	7844.4/7919.9 (**0.99**×)	134.2/130.4 (**1.03**×)	66.62/72.14 (**0.92**×)

A.4.2 Numerical Stability and Accuracy

We evaluate kernel-level numerical equivalence using Gaussian random inputs for Qwen3-Next deployed across four NVIDIA L40S GPUs. We run up to 1,024 decoding steps with the same initial state and token inputs for both methods. Table 5 demonstrates the mean element-wise absolute error, mean relative error, and cosine similarity between the outputs of chunkwise and recurrent decoding. The results show that chunkwise decoding remains numerically stable and produces outputs that are very close to those of recurrent decoding over long decoding steps. We also evaluate the numerical equivalence for the linear attention state tensor. Table 6 show that the state tensors also remain highly similar across long decoding steps.

Table 5: Numerical differences between output vectors produced by chunkwise and recurrent decoding at different decode lengths for Qwen3-Next. The buffer size is set to 16.

Decode Length	1	4	8	16	32	64	128	256	512	1024
Abs. Error ($\times 10^{-8}$)	0.1295	4.892	3.997	4.000	4.614	4.220	3.981	3.511	4.924	3.682
Rel. Error ($\times 10^{-3}$)	0.00057	0.4801	0.5462	0.8494	1.300	0.8206	0.8707	0.7215	2.371	0.5063
Cos. Distance ($\times 10^{-8}$)	0.010	2.120	2.820	2.350	4.100	2.470	2.190	2.470	3.700	2.410

Table 6: Numerical differences between the linear attention states produced by chunkwise and recurrent decoding for Qwen3-Next. The buffer size is set to 16.

Metric	16	32	64	128	256	512	1024
Abs. Error ($\times 10^{-7}$)	9.446	9.171	9.396	9.120	9.252	9.350	9.162
Rel. Error ($\times 10^{-3}$)	1.304	1.371	1.520	1.239	1.212	1.295	1.830
Cos. Distance ($\times 10^{-8}$)	4.360	4.320	4.130	4.240	4.380	4.250	4.320

Parallel decoding exhibits similar numerical stability. As shown in Table 7 and Table 8, its linear attention outputs and states closely match those of recurrent decoding, with errors remaining stable across all tested decoding lengths.

Table 7: Numerical differences between output vectors produced by parallel and recurrent decoding at different decode lengths for Qwen3-Next. The buffer size is set to 16.

Metric	1	4	8	16	32	64	128	256	512	1024
Abs. Error ($\times 10^{-8}$)	0.03104	4.477	4.446	4.428	4.486	4.570	4.454	4.414	4.496	4.417
Rel. Error ($\times 10^{-3}$)	0.0003971	0.6138	1.114	1.061	0.9969	1.277	1.170	1.224	1.135	1.046
Cos. Distance ($\times 10^{-8}$)	0.000	1.970	3.170	2.970	3.090	3.190	3.020	3.040	3.010	2.930

Table 8: Numerical differences between the linear attention states produced by parallel and recurrent decoding for Qwen3-Next. The buffer size is set to 16.

Metric	**16**	**32**	**64**	**128**	**256**	**512**	**1024**
Abs. Error ($\times 10^{-7}$)	9.194	9.178	9.238	9.222	9.215	9.209	9.223
Rel. Error ($\times 10^{-3}$)	1.370	1.389	1.388	1.338	1.376	1.336	1.372
Cos. Distance ($\times 10^{-8}$)	4.280	4.280	4.300	4.300	4.300	4.290	4.280

Furthermore we evaluate the downstream task accuracy on 1,000 GSM8K questions using the 5-shot setting for chunkwise decoding. GSM8K usually has tens of generated tokens per question and uses exact-match accuracy measurement. The results in Table 9 show that chunkwise decoding with various buffer sizes has close accuracy with recurrent decoding.

Table 9: GSM8K accuracy for Qwen3-Next serving with different buffer sizes. A buffer size of 0 denotes the recurrent decoding.

Buffer Size	**0**	**4**	**8**	**16**	**32**	**64**
Chunkwise Decoding	0.945	0.949	0.943	0.941	0.941	0.945
Parallel Decoding	0.945	0.945	0.941	0.948	0.946	0.948

Finally, we examine whether small numerical differences affect speculative decoding by comparing the average number of accepted tokens under parallel and recurrent verification. On Qwen3-Next, both the baseline and `KVBuffer` achieve an average of 2.99 accepted tokens, indicating that `KVBuffer` does not noticeably affect token acceptance.

A.4.3 Breakdown

We profile end-to-end batched decoding for Nemotron-3-Nano-4B on an L40S GPU with buffer size of 16. Table 10 reports the GPU kernel time for each operation category, averaged over 64 decoding steps. The recurrent SSM update accounts for 63.9% of the baseline GPU runtime. `KVBuffer` replaces this operation with buffered decoding and periodic state flushing, reducing the average SSM-related runtime from 31.409 ms to 20.429 ms, corresponding to a 1.537x speedup for the core SSM computation. This reduction translates into a 1.314x end-to-end decoding speedup.

Table 10: Breakdown of GPU kernel time per decode step for end-to-end batched inference with Nemotron-3-Nano-4B on an L40S GPU. Times are averaged over 64 decoding steps. "SSM update" includes only the core SSM operations, while the associated projections are categorized as GEMMs.

Category	**Baseline**		`KVBuffer`		**Speedup**
	Time (ms)	**Ratio**	**Time (ms)**	**Ratio**	
SSM Update	31.409	63.9%	0	0.0%	–
`KVBuffer` Decode	0	0.0%	17.986	48.1%	–
`KVBuffer` Flush	0	0.0%	2.443	6.5%	–
Conv	0.328	0.7%	0.310	0.8%	–
GEMM	13.477	27.4%	13.130	35.1%	–
Normalization	1.099	2.2%	0.805	2.2%	–
Other	2.820	5.7%	2.710	7.2%	–
Overall SSM Operations	31.409	63.9%	20.429	54.6%	**1.537**×
Overall	**49.132**	**100%**	**37.384**	**100%**	**1.314**×

A.4.4 Per-step Decoding Latency

As shown in Figure 9, we measure the end-to-end latency of 16 consecutive decoding steps for Nemotron-3-Nano-4B on an L40S GPU with a buffer size of 16. As the buffer fills, `KVBuffer` decoding latency gradually increases because each step reads and processes more buffered keys and values. When the buffer becomes full at step 15, the buffer is flushed, producing a periodic latency spike.

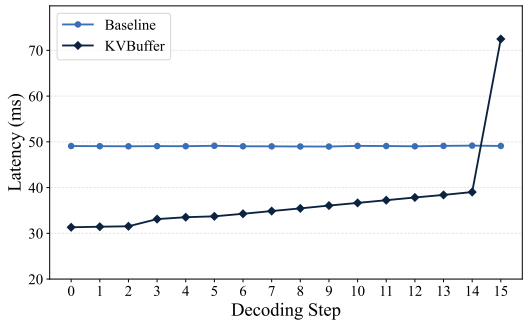


Figure 9: Per-step Decoding Latency

A.4.5 Memory Overhead

Table 11 reports the additional memory required to store the buffered keys and values for Qwen3-Next model. With a buffer size of 22, which we use for standard chunkwise decoding, `KVBuffer` requires 2.344 MB per request per GPU. This corresponds to approximately 12.7% of the original per-request linear-attention state memory. Because the maximum number of concurrent requests is determined by the capacity of the state-memory pool, adding a buffer of size 22 reduces the theoretical maximum request capacity by approximately 11.3%.

For speculative decoding with four draft tokens, `KVBuffer` introduces only 0.426 MB of additional memory per request per GPU, corresponding to a 2.3% memory overhead and an approximately 2.26% reduction in theoretical request capacity. Therefore, its memory impact is small in the typical speculative-decoding setting.

Table 11: Per-request memory overhead of `KVBuffer` at different buffer sizes. The measurements are obtained when serving Qwen3-Next on four L40S GPUs. All values are reported per request and per GPU.

Buffer Size	KVBuffer (MB)	Linear Attention State (MB)
4	0.426	18.42
8	0.853	18.42
16	1.705	18.42
22	2.344	18.42
32	3.410	18.42

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The main claims in the abstract and introduction reflect the contributions and scope accurately.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We discuss the limitations of this work in §6.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [Yes]

Justification: We state the assumptions for each theoretical result and provide proof sketches.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: We disclose all details needed to reproduce the experimental results.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [No]

Justification: We have not released the code at submission time because the algorithm is simple to implement, and our current implementation is built on SGLang and still requires additional engineering before release.

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: We provide experimental setting in §4.1.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [No]

   Justification: For each kernel-level experiment, we perform a warm-up phase and then report the average over 10 repeated runs; since the measured results are stable across runs, we omit error bars for clarity.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)

- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: We provide details of computer resources in §4.1.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: This research is conducted in accordance with the NeurIPS Code of Ethics.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [N/A]

    Justification: This work focuses on system-level efficiency improvements for serving linear-attention LLMs and does not introduce new models, datasets, or deployment-specific applications, so it has no direct societal impact beyond general inference-efficiency gains.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.

- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: This paper does not release new models, datasets, or other assets with high risk for misuse, so no additional safeguards are required.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: We credit and mention the assets usage in the paper.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.

- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [N/A]

    Justification: This paper does not release new assets.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: This paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: This paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.

- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [N/A]

    Justification: LLMs were used only for writing polishing and did not contribute to the core methodology, experimental design, scientific analysis, or originality of this research.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.