GPU Hierarchy Meets Structured Matrices: Fast Algorithms for State Space Models

Berlin Chen* **Caitlin Wang*** **Aakash Lahoti*** **Kevin Li***

Jay Shah **Jack Carlisle** **Timmy Liu** **Mengyu Guo**

J Zico Kolter **Albert Gu** **Tri Dao**

Abstract

Linear-time sequence models such as State Space Models (SSMs) offer an efficient alternative to self-attention and have demonstrated strong language modeling performance at scale. In practice, however, their asymptotic advantage is often lost at common training and prefill lengths—optimized FlashAttention kernels remain faster than existing SSM implementations up to 8k tokens. We narrow this performance gap with two algorithmic improvements: First, we introduce a split-sequence algorithm that decouples the compute tile from the sequence-parallel split length, preserving efficient matrix multiplication tiles while substantially reducing boundary-state traffic and latency. Second, we resolve a longstanding *speed–stability tradeoff* in computing the 1-semiseparable (SS) decay mask, the main non-matmul bottleneck shared by all chunk-wise linear-time kernels (Mamba-2, Mamba-3, GDN, KDA). Existing kernels use the fast *diff segsum* (prefix subtraction), which suffers monotonicity violations and catastrophic cancellation—failures observed in real pretraining (e.g., Nemotron-H); the stable alternative (*direct segsum*) materializes the dense mask and is $\sim 30\%$ slower. Our *hierarchy-aware segsum* decomposes the SS mask into 1-SS diagonal blocks and rank-one off-diagonal blocks, computed via warp-level scans and cross-warp aggregation directly in registers, achieving the stability of direct segsum at the speed of diff segsum. We implement these algorithms using CuTe-DSL with warp specialization, asynchronous TMA, WGMMA, and layout-aware accumulator fragments. Across sequence lengths, our H100 kernels reach 91% of measured memcpy bandwidth, improve prefill time by $4.1\times$ over the official Triton Mamba-2 kernel, and **outperform FlashAttention-4 starting at 1–2k sequence length**—to our knowledge, the first time a linear-time layer beats optimized attention at common 2–4k pretraining and prefill lengths. Our hierarchy-aware segsum is 1.35–$1.40\times$ faster than a naive stable direct-segsum implementation.

1 Introduction

Sequence models with linear compute and constant memory, such as Mamba-2,3 [Dao and Gu, 2024, Lahoti et al., 2026], Gated DeltaNet (GDN) [Yang et al., 2024a], and Kimi Delta Attention (KDA) [Kimi Team et al., 2025], have emerged as efficient alternatives to Transformers, demonstrating superior language modeling performance in large-scale hybrids such as Nemotron-H [NVIDIA et al., 2025], Qwen3-Next [Qwen Team, 2025], and Kimi Linear. However, this linear compute does not consistently translate into faster training and prefill at common sequence lengths—Kimi Linear and Nemotron-H pretrain at 4k and 8k tokens, respectively, yet the released Mamba-2 kernels reach parity with FlashAttention-4 [Zadouri et al., 2026] only around 8k tokens on H100 (Figure 3).

*Equal contribution (co-first authors).

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

Attention is typically faster, despite its quadratic scaling in FLOPs, as it has benefited from a sustained line of software-hardware co-design around the GPU hierarchy. For instance, FlashAttention [Dao et al., 2022] fuses online softmax with attention's matrix products in tiles that fit in on-chip SRAM, avoiding materialization of the full attention matrix in high-bandwidth memory (HBM), and FlashAttention-3 [Shah et al., 2024] uses warp specialization and pipelining to overlap softmax with asynchronous tensor-core matrix multiplications. While linear-time models have benefited from co-design—Mamba-2 [Dao and Gu, 2024] uses a chunkwise algorithm built around tensor-core matrix multiplications—their kernels remain early on this trajectory (Appendix C).

In this work, we *organize the computation of linear-time sequence models around the GPU hierarchy, from the device down to individual threads* (Appendix B). The guiding principle is to expose parallel work to utilize GPU resources while keeping intermediate data local to the computations that reuse it. Building on Mamba-2's chunkwise algorithm, we develop two complementary improvements that apply this principle at different levels. Across the device, **Split Sequence** balances parallelism across streaming multiprocessors (SMs) with reuse of the recurrent state within threadblocks to reduce HBM traffic while keeping the arithmetic cost low. Within each chunk, **hierarchy-aware segsum** computes decay-mask entries in the threads that apply them, avoiding a shared-memory round trip.

Balancing parallelism across SMs: Split Sequence. Split Sequence reorganizes how we distribute work across threadblocks. Mamba-2's chunkwise algorithm divides a sequence into chunks of a fixed size, typically 64, and assigns each chunk to a threadblock that runs on an SM. Each threadblock computes local interactions using matrix multiplications, which have quadratic cost in chunk size but run efficiently on tensor cores. It then accounts for earlier chunks by passing recurrent states between chunks through HBM, which is linear in the number of chunks.

Split Sequence separates two decisions that chunkwise algorithms tie together: how much work each threadblock receives and how large its local matrix computations are. Current chunk sizes generate many more threadblocks than are needed to saturate SMs, and consequently incur large recurrent state traffic through HBM. Simply enlarging these chunks would reduce state traffic but increase the FLOPs in the local pairwise interactions. Instead, we assign a larger *sequence split* to each threadblock and process it recurrently through small chunks. The threadblock retains intermediate states on-chip and passes states through HBM only at split boundaries.

GPU hierarchy guides the choice of both sizes. We choose the split length based on sequence length, batch size, number of heads, and number of SMs, so splitting supplies parallel work when batch and head parallelism alone are insufficient. State and head dimensions, tensor-core instruction shapes, and register and shared-memory capacity guide the chunk size; our Hopper implementation uses a chunk size of 64. *At batch size one and a sequence length of 65k, Split Sequence increases H100 throughput from 0.70 to over 2.05 TB/s compared with the same kernel without splitting (Figure 1).*

Mapping matrix structure to threads: Hierarchy-aware Segsum. Within each chunk, the algorithm computes local interactions as

$$\mathbf{Y}_{\text{local}} = ((\mathbf{Q}\mathbf{K}^\top) \odot \mathbf{E})\mathbf{V},$$

where $\mathbf{Q}$,$\mathbf{K}$,$\mathbf{V}$ denote queries, keys, and scaled values, and $\mathbf{E}$ is a causal decay mask. The matrix products run on tensor cores and a straightforward *direct segsum* implementation constructs $\mathbf{E}$ in a row-wise layout. To match the layout of $\mathbf{Q}\mathbf{K}^\top$, it writes the mask to shared memory and reloads it into registers, increasing latency. We instead exploit the mask's 1-semiseparable structure: we decompose $\mathbf{E}$ into smaller blocks, with 1-semiseparable diagonal blocks and rank-one off-diagonal blocks. We compute these blocks in parallel across warps, matching the layout of $\mathbf{Q}\mathbf{K}^\top$ so that each thread constructs and applies its mask entries in registers without the shared-memory round trip. *Our hierarchy-aware segsum reduces SSD forward latency from 0.54–0.57 ms to 0.40–0.41 ms on H100 (Figure 2).*

Stable mask construction. An alternative used in Mamba-2 and GDN kernels is *diff segsum*, which computes segment sums by subtracting prefix sums. Diff segsum matches our hierarchy-aware implementation's speed but suffers from numerical instability. We analyze two failure modes: catastrophic cancellation and loss of monotonicity in parallel prefix scans, which can produce positive segment sums from non-positive decay increments. This instability was observed during Nemotron-H pretraining.[2] Using decay increments captured from that run, we find a chunk of size 128 where prefix subtraction changes a segment sum of approximately -72.3 to $+128$, causing FP32 exponentiation to overflow. Our hierarchy-aware segsum instead forms segment sums directly from

[2]The Mamba-2 kernel patch addressing the overflow is documented in state-spaces/mamba#713.

the decay increments, avoiding prefix subtraction while computing each mask entry in the thread that applies it. *It therefore retains the numerical stability of direct segsum at the speed of diff segsum.*

Results. We implement these algorithms in CuTe DSL for Mamba-2 and Mamba-3.[3] On H100, our Mamba-2 forward kernel reaches 91% of measured `memcpy` bandwidth and is $4.1\times$ faster than the released Triton kernel. In the evaluated configurations, the crossover with FlashAttention-4 moves to 1–2k tokens for the forward pass and 2–4k for the combined forward and backward passes, with performance gains also demonstrated on Blackwell. Using these kernels in a complete Mamba-2 mixer yields 1.50–$2.80\times$ prefill speedups and 1.24–$1.90\times$ speedups for the combined forward and backward passes, including projections, convolution, gating, and normalization.

2 Preliminaries

Notation. Bold capital letters denote matrices, $\odot$ denotes elementwise multiplication, and $\mathrm{diag}(v)$ denotes the diagonal matrix with entries given by a vector v. We use $\mathtt{L}$ for sequence length, d for state dimension, and d_v for value dimension. We describe one sequence and one head, omitting the independent batch and head indices. At token t, let $x_t, o_t \in \mathbb{R}^{d_v}$ denote the input and output, $b_t, c_t \in \mathbb{R}^d$ the input and readout vectors, and $\boldsymbol{S}_t \in \mathbb{R}^{d\times d_v}$ the recurrent state. The scalar $a_t \le 0$ denotes the log-decay and $\Delta_t > 0$ the discretization step size.

Mamba-2 recurrence. Mamba-2 uses a scalar-decay state space model [Dao and Gu, 2024],

$$\boldsymbol{S}_t = e^{a_t}\boldsymbol{S}_{t-1} + \Delta_t b_t x_t^\top, \qquad o_t^\top = c_t^\top \boldsymbol{S}_t, \qquad \boldsymbol{S}_{-1} = \boldsymbol{S}_{\mathrm{init}}, \tag{1}$$

where $\boldsymbol{S}_{\mathrm{init}}$ is the initial state. Each step decays the previous state, adds the current input contribution, and uses c_t to produce the output. The state summarizes earlier tokens for subsequent steps.

Parallel computation. The state space duality (SSD) formulation expresses the same computation through matrix multiplications within chunks and recurrent state propagation between chunks. These matrix multiplications expose parallel work that runs efficiently on GPU tensor cores. For a chunk of $\mathtt{C}$ tokens, stack the vectors $b_i^\top, c_i^\top, x_i^\top$ into $\boldsymbol{B}, \boldsymbol{C} \in \mathbb{R}^{\mathtt{C}\times d}$ and $\boldsymbol{X} \in \mathbb{R}^{\mathtt{C}\times d_v}$, using local indices $i, j = 0,\ldots,\mathtt{C}-1$. The causal decay mask $\boldsymbol{E} \in \mathbb{R}^{\mathtt{C}\times\mathtt{C}}$ accounts for the decay between each pair of tokens:

$$E_{ij} = \begin{cases} \exp\left(\sum_{k=j+1}^{i} a_k\right), & j \le i, \\ 0, & j > i, \end{cases} \qquad \Delta = (\Delta_0,\ldots,\Delta_{\mathtt{C}-1})^\top. \tag{2}$$

An empty sum is zero, so $E_{ii} = 1$. The attention notation in the introduction corresponds to $\mathbf{Q} = \boldsymbol{C}$, $\mathbf{K} = \boldsymbol{B}$, and $\mathbf{V} = \mathrm{diag}(\Delta)\boldsymbol{X}$; $\boldsymbol{E}$ denotes the same decay mask in both formulations.

Given the incoming state $\boldsymbol{S}_{\mathrm{in}}$, a chunk returns its token outputs and an outgoing state. The incoming state decays as it moves through the chunk, while each input contributes to the outgoing state with the decay accumulated after its position. We collect these factors and the total chunk decay as

$$f_i = \exp\left(\sum_{k=0}^{i} a_k\right), \qquad \beta_j = \Delta_j \exp\left(\sum_{k=j+1}^{\mathtt{C}-1} a_k\right), \qquad \gamma = \exp\left(\sum_{k=0}^{\mathtt{C}-1} a_k\right). \tag{3}$$

The two outputs of the chunk computation are then

$$\boldsymbol{O} = \underbrace{\mathrm{diag}(f)\boldsymbol{C}\boldsymbol{S}_{\mathrm{in}}}_{\text{incoming-state readout}} + \underbrace{\big((\boldsymbol{C}\boldsymbol{B}^\top)\odot\boldsymbol{E}\big)\mathrm{diag}(\Delta)\boldsymbol{X}}_{\text{within-chunk interactions}}, \tag{4}$$

$$\boldsymbol{S}_{\mathrm{out}} = \gamma\boldsymbol{S}_{\mathrm{in}} + \boldsymbol{B}^\top\mathrm{diag}(\beta)\boldsymbol{X}. \tag{5}$$

The local output has the form $((\mathbf{Q}\mathbf{K}^\top)\odot\boldsymbol{E})\mathbf{V}$ from the introduction; the incoming-state term accounts for all earlier chunks. These identities hold for any contiguous interval of the sequence. An interval can therefore be computed using the parallel formulation directly, or subdivided into smaller chunks connected by the state recurrence. The choice of interval size and its assignment to GPU threadblocks determine the resulting computation and communication costs.

3 Method

Our goal is to compute the forward pass of Equation (1) efficiently on GPUs. For simplicity, we develop our algorithmic techniques for Mamba-2; the same scheduling and mask-construction

[3] Code: `https://github.com/state-spaces/mamba`.

principles also apply to Mamba-3, GDN, and KDA [Lahoti et al., 2026, Yang et al., 2024a, Kimi Team et al., 2025]. We first introduce **Split Sequence** to distribute sequence splits across SMs, choosing the split length to expose enough parallel work (Section 3.1). Compared with scheduling every chunk independently, processing several chunks within a split reduces state traffic through HBM while preserving small, efficient matrix computations. We then compute each chunk's outputs using the parallel formulation. Here, **hierarchy-aware segsum** constructs and applies the decay mask $\boldsymbol{E}$ to $\boldsymbol{C}\boldsymbol{B}^\top$ directly in registers, avoiding the shared-memory round trip of row-wise direct segsum (Section 3.2). Unlike diff segsum, it forms interval sums directly instead of subtracting prefixes, preserving numerical stability (Section 4).

3.1 Balancing Parallelism Across SMs: Split Sequence

3.1.1 The Split Sequence Algorithm

We partition a sequence of length $\mathtt{L}$ into contiguous *sequence splits* of at most $\mathtt{T}$ tokens, giving $\mathtt{J} = \lceil \mathtt{L}/\mathtt{T} \rceil$ splits indexed by $m = 0, \ldots, \mathtt{J}-1$. Earlier tokens affect a split only through its incoming recurrent state. SSD's state-passing decomposition [Dao and Gu, 2024] separates the local computation from this dependence in three stages (Figure 5, Appendix A):

1. Local computation (chunk state). Each threadblock independently computes one split's outgoing state from zero initialization,

$$\bar{\boldsymbol{S}}^{(m)} = (\boldsymbol{B}^{(m)})^\top \mathrm{diag}(\beta^{(m)}) \boldsymbol{X}^{(m)}, \tag{6}$$

and its total decay $\bar{\gamma}^{(m)}$ from Equation (3). Superscript (m) denotes the operands and decay factors of Equations (3) and (5) for split m. The threadblock writes the summary $(\bar{\boldsymbol{S}}^{(m)}, \bar{\gamma}^{(m)})$ to HBM.

2. State passing. Starting from $\boldsymbol{S}_{\mathrm{in}}^{(0)} = \boldsymbol{S}_{\mathrm{init}}$, propagate states in split order:

$$\boldsymbol{S}_{\mathrm{out}}^{(m)} = \bar{\gamma}^{(m)} \boldsymbol{S}_{\mathrm{in}}^{(m)} + \bar{\boldsymbol{S}}^{(m)}. \tag{7}$$

Each outgoing state becomes the next split's incoming state, which is stored in HBM; the last outgoing state is $\boldsymbol{S}_{\mathrm{final}}$.

3. Compute outputs (chunk scan). Threadblocks now compute their splits' outputs in parallel from these incoming states. By Equation (4), the readout decomposes as

$$\boldsymbol{O}^{(m)} = \mathrm{diag}(f^{(m)}) \boldsymbol{C}^{(m)} \boldsymbol{S}_{\mathrm{in}}^{(m)} + \bar{\boldsymbol{O}}^{(m)}, \tag{8}$$

The zero-initialized local readout $\bar{\boldsymbol{O}}^{(m)}$ is computed in this stage; the threadblock writes $\boldsymbol{O}^{(m)}$ to HBM.

Batch elements and heads are independent throughout. These identities specify each split's inputs and outputs; Section 3.1.3 describes how the threadblock evaluates them through smaller chunks.

3.1.2 State Traffic and Parallelism Across SMs

We choose the split length $\mathtt{T}$ to supply parallel work across SMs, independently of the chunk size $\mathtt{C}$ used for local matrix computation. A chunk-parallel schedule ties these choices through $\mathtt{T} = \mathtt{C}$: choosing $\mathtt{C} = 64$ for matrix computations also sets the granularity of parallel work. For batch size B and head count H, let $P(\mathtt{T})$ denote the number of threadblocks in each parallel stage and $Q(\mathtt{T})$ the estimated state I/O in bytes. Each summary is written and read once, and each incoming state is written and read once, for four transfers of $b_S d d_v$ bytes per split, where b_S is the number of bytes per state entry. Thus,

$$P(\mathtt{T}) = BH \left\lceil \frac{\mathtt{L}}{\mathtt{T}} \right\rceil, \qquad Q(\mathtt{T}) \simeq 4 b_S d d_v P(\mathtt{T}). \tag{9}$$

These estimates exclude input/output tensors, scalar decay metadata, and cache effects.

Consider one sequence of $\mathtt{L} = 32{,}768$ tokens ($B = 1$) with $H = 32$ heads. With $\mathtt{C} = 64$, the chunk-parallel schedule launches $32 \times 512 = 16{,}384$ threadblocks per parallel stage, compared with 132 SMs on an H100 SXM [NVIDIA, 2022b]. Assigning a whole sequence to each threadblock would instead expose only $BH = 32$ threadblocks. Our split selection chooses $\mathtt{J} = 4$ splits per head, each of length $\mathtt{T} = 8192$, giving 128 threadblocks—roughly one per SM.

The cost of the finer partition is state traffic: every independently scheduled chunk requires states to pass through HBM, even when additional threadblocks provide little benefit to utilization. For $d = 128$, $d_v = 64$, and FP32 states, each state occupies 32 KiB. At $\mathtt{C} = 64$, one tensor of chunk states occupies 512 MiB and the four transfers total about 2 GiB. Split Sequence retains intermediate states within each split, reducing this estimate to 16 MiB, or $128\times$ less state I/O.

Schedule	Split length $\mathtt{T}$	Threadblocks $P(\mathtt{T})$	State I/O $Q(\mathtt{T})$
Chunk-parallel, $\mathtt{C}=64$	64	16,384	2 GiB
Split Sequence	8192	128	16 MiB

Larger splits also shorten state passing from $\lceil \mathtt{L}/\mathtt{C} \rceil$ chunk summaries to $\lceil \mathtt{L}/\mathtt{T} \rceil$ split summaries per head—from 512 to four in this example with $\mathtt{C}=64$. The SM count guides how many splits are useful, while a minimum split length of 4096 limits communication and scheduling overhead; achieved utilization also depends on the resources each threadblock requires.

When batch and head parallelism already provide enough work, no sequence splitting is needed. For example, with $B = 16$ independent sequences of $\mathtt{L} = 2048$ tokens and $H = 32$ heads, the heuristic chooses $\mathtt{T}=\mathtt{L}$, leaving $BH = 512$ threadblocks. Relative to $\mathtt{C}=64$, the three-stage state I/O estimate falls from 2 GiB to 64 MiB, a $32\times$ reduction. With only one split per sequence/head, the incoming state is already known, so summary computation and state passing can be skipped entirely.

3.1.3 Computation Within an SM

A larger split need not increase the size of its matrix computations. Applying the parallel formulation directly to all $\mathtt{T}$ tokens would form $\mathtt{T}\times\mathtt{T}$ local interactions, with total cost $O(BH\mathtt{L}\mathtt{T}(d+d_v))$. Instead, we apply the block decomposition again within each split, dividing it into chunks of size $\mathtt{C}$ (Figure 5, right). The threadblock uses the parallel formulation in Equation (4) within each chunk and the recurrent update in Equation (5) between chunks. The state stays in registers as the threadblock streams successive chunks and writes their outputs; these internal boundaries require no state transfer through HBM. The local interaction work is therefore

$$BH\frac{\mathtt{L}}{\mathtt{C}}O\big(\mathtt{C}^2(d+d_v)\big)=O(BH\mathtt{L}\mathtt{C}(d+d_v)), \tag{10}$$

independently of the split length $\mathtt{T}$. *Split Sequence thus reduces boundary-state traffic without enlarging the local interactions.*[4] Tensor-core instruction shapes, state and head dimensions, and register and shared-memory capacity guide the choice of $\mathtt{C}$; our Hopper implementation uses $\mathtt{C} = 64$. The threadblock streams the split's inputs, so increasing $\mathtt{T}$ does not require the entire split to fit on-chip.

Both the summary and output stages traverse these small chunks in sequence order. The summary stage computes only state updates and total decay, while the output stage also computes the token outputs from the split's incoming state. When multiple splits are used, recomputing these state updates adds $O(BH\mathtt{L}dd_v)$ arithmetic and rereads inputs, but avoids loading a state for every chunk. Appendix A gives the complete Split Sequence algorithm and its kernel implementation.

3.2 Hierarchy-Aware Mask Construction and Application

3.2.1 The Mask Construction Problem

Within each chunk, the decay mask $\boldsymbol{E}$ scales the matrix product $\boldsymbol{A}=\boldsymbol{C}\boldsymbol{B}^\top$ before the second matrix multiplication in Equation (4). The entries of $\boldsymbol{A}$ reside in registers distributed across threads according to the tensor-core accumulator layout. Efficient mask application therefore requires computing each decay value in the appropriate thread while limiting arithmetic, communication, and storage.

Naive attempt 1: Compute each row independently. For a 64-token chunk, a straightforward direct-segsum implementation assigns each 64-entry row to two threads. They scan the log-decays to construct the row's segment sums and exponentiate them to obtain the mask values. Processing rows independently exposes parallel work, but different rows repeatedly process overlapping decay intervals.

This assignment also differs from the accumulator layout: the threads constructing a mask entry generally do not hold the corresponding entry of $\boldsymbol{A}$. Applying the mask consequently requires *thread-ownership reassignment*, implemented by writing the completed mask to shared memory and reloading its entries into the consuming threads.

Naive attempt 2: Compute directly in the accumulator layout. A second approach assigns mask construction to the threads holding the corresponding accumulator entries. Each row is scanned through successive groups of columns, with shuffle instructions exchanging partial sums between cooperating threads. This removes the shared-memory round trip for the completed mask.

[4] Tiled Flash Linear Attention also reduces state materialization through larger chunks and introduces tiling within them [Beck et al., 2025]. Its pairwise interactions span the larger chunk. Split Sequence connects small chunks recurrently, so the interaction size remains $\mathtt{C}$ even as the assigned split grows.

However, each row still performs its own scan through the causal column groups. In the Hopper layout, scanning an eight-entry row segment requires two shuffle rounds; processing all eight groups of a 64-entry row can produce a dependency chain of 16 rounds. Different rows repeat work over overlapping intervals. Although upper-triangular entries can be skipped, the rank-one structure of the lower off-diagonal blocks remains unused.

Limits of overlapping mask construction with matrix multiplication. Because the mask depends only on the log-decays, its construction can begin while the first matrix multiplication executes. Overlap hides only the work completed before the accumulator becomes available; any remaining scan operations delay mask application. Precomputing and retaining the entire mask in registers also increases register pressure, since those values must remain available alongside the matrix-product accumulators.

Exploit the mask's 1-semiseparable structure. These attempts motivate a construction that reuses interval computations while producing mask values in their consuming threads. The mask's 1-semiseparable structure provides this opportunity: diagonal blocks are smaller causal masks, while strictly lower blocks are rank one (Appendix D.2). We use local scans for the diagonal blocks and reconstruct off-diagonal entries from reusable factors. Precomputing compact within-group suffixes and group totals allows each thread to construct and apply its mask values as needed.

3.2.2 Decomposing the Mask into Reusable Factors

Decomposition of an off-diagonal block. Divide a compute chunk into contiguous groups of s tokens, numbered from zero. Our Hopper implementation uses $s = 8$ to match the eight-entry row segments in the accumulator layout shown in Figure 6. A strictly lower off-diagonal $s \times s$ block pairs rows in one group with columns in an earlier group. Each decay interval consists of a suffix within the column's group, the complete intervening groups, and a prefix within the row's group. For example, consider groups containing tokens 0–7, 8–15, and 16–23. The entry $E_{18,2}$ contains the decay from token 3 through token 18:

$$E_{18,2} = \exp\Bigg(\underbrace{\sum_{k=3}^{7} a_k}_{\text{within-group suffix}} + \underbrace{\sum_{k=8}^{15} a_k}_{\text{intervening group sum}} + \underbrace{\sum_{k=16}^{18} a_k}_{\text{within-group prefix}} \Bigg).$$

Changing the row from 18 to 19 changes only the prefix; changing the column from 2 to 3 changes only the suffix. The intervening group sum is common to every entry in this block. Thus, its 64 entries can be reconstructed from eight suffixes, eight prefixes, and one common intervening sum. This is the rank-one off-diagonal structure used by the algorithm. Diagonal blocks still require local causal interval sums, and upper blocks are zero.

Within-group suffixes. For a column token j in group h, define

$$\mathsf{suffix}_j = \sum_{k=j+1}^{(h+1)s-1} a_k. \tag{11}$$

A reverse exclusive scan precomputes the s suffix values in each group; the suffix at the last token is zero. We store these values in shared memory. During mask application, a thread holding A_{ij} reads the suffix for column j. Rows in later groups reuse the suffix for that column.

Within-group prefixes. For a row token i in group g, define

$$\mathsf{prefix}_i = \sum_{k=gs}^{i} a_k. \tag{12}$$

There are s prefix values per row group, each including its row token's increment. The consuming threads precompute these prefixes by local scans and retain them in registers alongside the local diagonal-scan results. Thus, prefixes are not written to shared memory; each thread retains the row values needed for its own accumulator entries.

Sum over intervening groups. We also sum each group's log-decays once and store the resulting total in shared memory:

$$\mathsf{total}_h = \sum_{k=hs}^{(h+1)s-1} a_k.$$

For $s = 8$, this scalar is the eight-token sum, or `sum8`. For a row i in group g and a column j in an earlier group $h < g$, the three contributions give

$$E_{ij} = \exp\left(\mathsf{prefix}_i + \sum_{v=h+1}^{g-1} \mathsf{total}_v + \mathsf{suffix}_j\right). \tag{13}$$

The intervening sum is accumulated as the thread visits column groups from right to left. For the immediately preceding column group, there are no intervening groups, so the thread initializes a running row value $r_i \leftarrow \mathsf{prefix}_i$. After processing column group h, moving one group farther left makes group h part of the intervening interval, so each thread updates $r_i \leftarrow r_i + \mathsf{total}_h$. Reading one shared scalar therefore accounts for crossing the entire group. Each thread updates its running row value independently and forms its mask entry as $E_{ij} = \exp(r_i + \mathsf{suffix}_j)$. The intervening sum is folded into r_i; no separate array of intervening sums is stored for the off-diagonal blocks.

Resulting storage footprint. The $s+s+1$ quantities describe the mathematical contributions to one off-diagonal block. In the implementation, suffixes and group totals are shared across blocks, while row prefixes and accumulated group sums are combined in the register-resident values r_i. Across a compute chunk, the shared mask data therefore consists of `C` suffixes and $\mathtt{C}/s$ group totals. For $\mathtt{C} = 64$ and $s = 8$, this is $64 + 8 = 72$ shared values, compared with 4096 entries in a dense 64×64 mask tile. These counts concern shared mask data; local diagonal-scan results and running row values also occupy registers, with their placement determined by thread ownership. Threads construct and apply mask entries as needed, avoiding storage of the full mask.

3.2.3 Hierarchy-Aware Mask Construction

The Mask Subroutine has two phases per chunk (Appendix D.1).

Phase 1: Precompute shared quantities. For each token group, compute the within-group suffixes and group total, then store them in shared memory. This computation depends only on the log-decays and can overlap the matrix multiplication producing $\boldsymbol{C}\boldsymbol{B}^\top$.

Phase 2: Construct and apply the mask. The consuming threads compute their within-group prefixes and diagonal interval sums locally. They reconstruct off-diagonal entries using the shared suffixes and group totals, extending their partial sums as they traverse earlier column groups. Once the matrix-product entries are available, each thread applies the corresponding decay factors directly in registers. Mask application still requires $O(\mathtt{C}^2)$ entrywise work; the construction reduces repeated scans, communication, and intermediate storage.

4 Numerical Characteristics of Segsum Computation

We analyze two ways to construct the decay mask $\boldsymbol{E}$ in Equation (2): prefix subtraction (diff segsum) and direct interval summation (direct segsum).

4.1 Diff Segsum

Each row of E is a reverse prefix product of $\exp(a_k)$ values. Let $s_k = \sum_{t=1}^{k} a_t$ be the prefix sum. Then $M_{ij} := \log E_{ij} = s_i - s_j$ for $j \le i$. We precompute and cache $s_1, \ldots, s_{\mathtt{C}}$, then retrieve s_i and s_j to compute each E_{ij} by subtraction and exponentiation, avoiding materialization of the dense $\mathtt{C} \times \mathtt{C}$ mask.

This approach results in an efficient kernel design; however, it has two numerical disadvantages: it relies on a monotonicity invariant of the prefix sums that is not guaranteed under associative parallel scans in floating-point arithmetic, and it is prone to catastrophic cancellation when subtracting two large prefixes. We discuss each in turn, and provide empirical evidence in Section 5.2.

Violation of monotonicity. Each decay increment satisfies $a_t \le 0$. The prefix sum s_i is therefore monotone (non-increasing) in exact arithmetic. Violating this ordering can make the exponent $s_i - s_j$ positive, destabilizing the SSM.

In floating-point arithmetic, sequential cumulative sum preserves monotonicity: rounded floating-point addition is monotone under IEEE-754, hence $s_k + a_{k+1} \le s_k$ implies

$$s_{k+1} = \mathrm{round}(s_k + a_{k+1}) \le \mathrm{round}(s_k) = s_k.$$

Efficient cumulative sum, however, typically uses a parallel prefix scan, e.g., Hillis–Steele. These scans assume associativity, which floating-point addition does not satisfy. Even with non-positive increments, different reduction orders can produce rounded prefixes that violate the monotone ordering preserved by the sequential recurrence (Appendix E.1).

Remark. This violation has been observed during pretraining of the hybrid Mamba-2 models in Nemotron-H. Extreme differences in decay magnitudes cause diff segsum to overflow and produce `NaN` outputs unless positive segment sums are clamped to zero.[5]

Catastrophic cancellation. Even when the monotone invariance holds, the unstable form $M_{ij} = s_i - s_j$ requires subtracting two prefixes of potentially large magnitude. Let $P := \max(|s_i|, |s_j|)$ denote the prefix magnitude. In fp32, values near P are quantized to a grid with spacing $\mathrm{ulp}(P) := 2^{\lfloor \log_2 P \rfloor - 23}$, the unit-in-the-last-place at scale P, so each rounded prefix carries an absolute rounding error of up to $\mathrm{ulp}(P)/2$. Such an error may persist in the subtraction approach, and may be large relative to magnitude of the true value, i.e., $\sum_{k=j+1}^{i} a_k$. See Appendix E.1 for an illustrative example.

Remark. Clamping, i.e., $\exp(\min(M_{ij}, 0))$, restores the valid domain $(0, 1]$ and is the default in the standard Mamba implementation. However, it does not address catastrophic cancellation: clamping only truncates positive errors to zero, leaving negative-side rounding errors unchanged. Addressing both failure modes motivates direct segsum.

4.2 Direct Segsum

Direct segsum forms $M_{ij} = \sum_{k=j+1}^{i} a_k$ from the increments a_k, without prefix subtraction:

1. *Broadcast* the vector $\boldsymbol{a} = (a_1, \ldots, a_{\mathtt{C}})$ along the row dimension to form a $\mathtt{C} \times \mathtt{C}$ matrix whose entry at row i, column j is a_j for every i.
2. *Mask* the diagonal and lower triangle, leaving a_j at row i, column j when $j > i$ and zero elsewhere. Row i now holds the increments $a_{i+1}, \ldots, a_{\mathtt{C}}$ in columns $i+1, \ldots, \mathtt{C}$, and zeros at columns $1, \ldots, i$.
3. *Cumsum* along each row independently. The entry at row i, column j becomes the sum of that row's entries in columns $1, \ldots, j$, which equals $M_{ji} = \sum_{k=i+1}^{j} a_k$.

Sign preservation. Step 3 only adds non-positive numbers. Under IEEE-754 rounding mode, $\mathrm{round}(x + y) \leq 0$ whenever $x, y \leq 0$. By induction along the row, every running sum encountered during the row-wise cumsum remains non-positive, hence $\hat{M}_{ij} \leq 0$ holds by construction. In contrast to the diff segsum, no clamping is needed here.

Error scaling. The error in direct segsum is controlled by the magnitude of the *segsum entry*, whereas the error in diff segsum depends on the potentially much larger *prefix* scale. Clamping cannot remove this gap: it corrects sign violations but not the cancellation error from subtracting large rounded prefixes (Appendix E.2).

Kernel considerations. The row-wise implementation adds a dense $\mathtt{C}^2$-entry shared-memory round trip on the critical non-matmul path, increasing memory usage and runtime. Existing implementations therefore face a speed–stability tradeoff: diff segsum can be implemented efficiently but suffers numerical issues, while direct segsum has superior numerical properties but is inefficient when implemented naively. The hierarchy-aware algorithm in Section 3.2 maintains the favorable numerical properties of direct segsum while being efficient to compute.

5 Experiments

5.1 Speed benchmarks

We benchmark on a single NVIDIA H100 80GB HBM3 GPU: **Mamba-2 (Baseline)** [Dao and Gu, 2024], **Mamba-2 (CuTe)** (ours, Appendix A), **Mamba-3 (Baseline)** [Lahoti et al., 2026], **Mamba-3 (CuTe)** (ours), **Mamba-2 (Fused)** [Astra et al.], and **FA4** (causal). Configurations differ by experiment. The throughput sweep (Figure 1) uses $\mathtt{H} = 32$, one group, and $\mathtt{BL} = 65{,}536$; the segsum ablation (Figure 2) uses $\mathtt{H} = 64$ and $\mathtt{BL} = 32{,}768$. Both use state dimension $\mathtt{D} = 128$ and value dimension $\mathtt{D_v} = 64$. Each sweep varies `L` while scaling `B` inversely.

Throughput. Figure 1 reports memory throughput for three variants: the Triton baseline, our CuTe kernel without splitting, and with sequence splitting (Section 3.1). With ample batch parallelism, both CuTe variants hit 2.69–2.73 TB/s ($\sim 91\%$ of `memcpy`, 2.99 TB/s), vs. only 0.54–0.64 TB/s for the Triton baseline (**4.1× gap**). As `L` grows and `B` shrinks, $\mathtt{B} \cdot \mathtt{H}$ drops below the available SM count; no-split throughput falls to 1.45 TB/s at L=32k and 0.70 TB/s at L=65k. The split variant sets $n_{\text{split}} = 1$

[5] The kernel patch is documented in state-spaces/mamba#713.

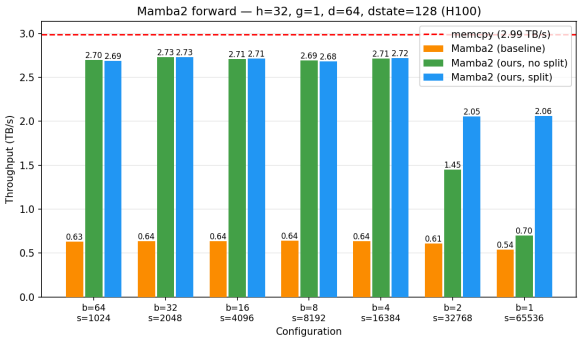


Figure 1: SSD forward throughput on H100: $\mathtt{H} = 32$, one group, $\mathtt{D} = 128$, $\mathtt{D_v} = 64$, $\mathtt{BL} = 65{,}536$. CuTe reaches 91% of `memcpy`; splitting preserves throughput at long lengths.

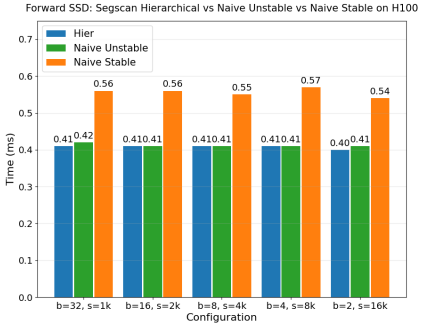
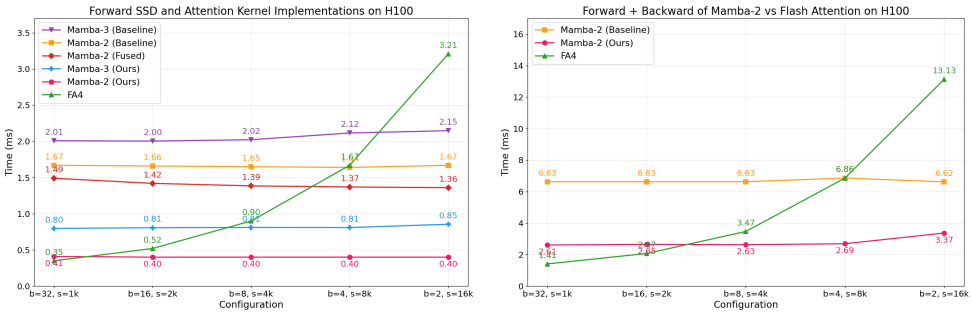


Figure 2: SSD forward latency on H100: $\mathtt{H} = 64$, $\mathtt{D} = 128$, $\mathtt{D_v} = 64$, $\mathtt{BL} = 32{,}768$. Only the segsum algorithm changes across variants.



(a) Forward pass.

(b) Forward + backward pass.

Figure 3: Mamba/SSD vs. causal FA4 latency on H100, with $\mathtt{BL} = 32{,}768$. Batch size, sequence length, and per-head dimensions ($\mathtt{D} = 128$, $\mathtt{D_v} = 64$) match; head counts differ (Mamba: 64; FA4: 16). Mamba-2 CuTe crosses FA4 between 1–2k tokens in *(a)* forward and between 2–4k in *(b)* forward+backward.

when $\mathtt{B} \cdot \mathtt{H}$ already fills the device (matching the no-split kernel within 0.01 TB/s), and splits only when needed, sustaining 2.05+ TB/s ($\sim 69\%$ of `memcpy`) at $\mathtt{L}$=65k,$\mathtt{B}$=1.

Comparison with attention. Figure 3 fixes $\mathtt{BL} = 32{,}768$, matching $\mathtt{B}$, $\mathtt{L}$, and per-head dimensions: SSD state and attention query/key dimensions are 128, and value dimensions are 64. Mamba uses 64 heads and FA4 uses 16, giving total output widths of 4096 and 1024, respectively. Cross-family speedups and crossover lengths refer to these kernel workloads, not equal-width models. Our SSD kernel takes $\sim$0.40 ms across $\mathtt{L}$, while FA4 latency increases from 0.35 ms at 1k to 3.21 ms at 16k ($8\times$ gap). The crossover occurs between 1k and 2k tokens; at 8k our kernel is $4\times$ **faster**. Prior subquadratic kernels reached parity only at $\mathtt{L} \geq 8$k. The Triton Mamba-2 kernel runs at $\sim$1.65 ms, slower than FA4 below 4k.

Comparison with other Mamba implementations. Figure 3 shows our CuTe kernels are fastest for each architecture: $>4\times$ over Mamba-2 Triton and $>2.5\times$ over Mamba-3 Triton. Appendix G compares optimized kernels (TileLang Mamba-2, Ant Ling cuLA KDA, Qwen FlashQLA GDN); our Mamba-2 kernel is fastest overall and hits the H100 `memcpy` roofline up to 16k tokens, while GDN/KDA's more complex operators make peak bandwidth harder to achieve.

Comparison with segsum algorithms Figure 2 compares SSD forward time using hierarchy-aware segsum (Section 3.2), naive direct segsum, and diff segsum (Section 4). All use the same CuTe SSD kernel, changing only segsum. Our hierarchy-aware implementation runs in $\sim$0.40–0.41 ms across all batch/sequence configurations, matching diff segsum's speed while retaining direct segsum's numerical stability. The naive stable implementation takes $\sim$0.54–0.57 ms, roughly $1.35\times$–$1.40\times$ slower.

Performance portability to Blackwell. We port our forward kernel to Blackwell SM100 and benchmark BF16 on NVIDIA GB200 (Figure 4), matching the H100 attention comparison: $\mathtt{BL} = 32{,}768$, $\mathtt{D} = 128$, and $\mathtt{D_v} = 64$, with 64 heads and one group for Mamba-2 and 16 heads for causal FA4. Our

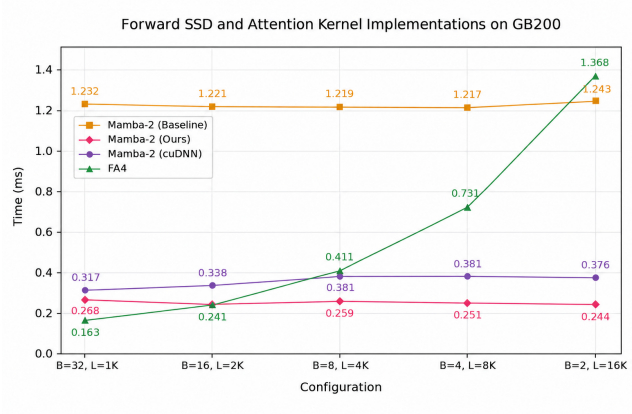


Figure 4: BF16 forward latency on NVIDIA GB200 (Blackwell): $\texttt{BL} = 32{,}768$, $\texttt{D} = 128$, $\texttt{D}_\texttt{v} = 64$; Mamba-2 uses 64 heads and one group, FA4 uses 16 heads. Our kernel beats both Mamba-2 baselines throughout the sweep and causal FA4 from 4k onward. Lower is better.

hierarchy-aware implementation maintains nearly constant latency, from 0.244 to 0.268 ms, achieving speedups of 4.60–$5.09\times$ over the official Mamba-2 baseline and 1.18–$1.54\times$ over NVIDIA's cuDNN Frontend FROST implementation [NVIDIA, 2026]. It reaches near parity with FA4 at 2k tokens and outperforms it at all tested lengths from 4k onward, reaching $5.61\times$ at 16k. These results show that the performance benefits extend from Hopper to Blackwell.

5.2 Numerical stability

We validate Section 4's analysis by sweeping the prefix scale $P \in \{2^{20}, 2^{24}, 2^{28}, 2^{30}\}$ and tail-to-ULP ratio $\rho = |a_t|/\mathrm{ulp}(P)$. Each chunk has $T = 64$ entries: the first 48 build the prefix using normalized weights from $\mathrm{Unif}[0,1]$, and the final 16 are sampled as $\mathrm{Unif}(0.5, 1.5) \cdot \rho \cdot \mathrm{ulp}(P)$. All increments are non-positive, so a positive $s_T - s_k$ is a rounding artefact. We compare sequential cumsum, Hillis–Steele, and Triton's `tl.cumsum` at `num_warps`$=4$, with 256 trials per cell.

Monotonicity violations. Figure 7 reports the probability of a positive rounded segment sum $s_T - s_k > 0$ and the worst exponentiated violation. Sequential cumsum preserves monotonicity across the sweep; both associative scans violate it. Hillis–Steele peaks near 7% at $\rho = 0.125$, while `tl.cumsum(num_warps=4)` remains below 4%. At $\rho = 0.25$, both associative scans reach worst-case displacements of $+64$ at $P = 2^{30}$, giving $\exp(64) \approx 6 \times 10^{27}$. The largest displacement is $+128$ for Hillis–Steele at $\rho = 0.10$, for which $\exp(128) \approx 4 \times 10^{55}$, well past the fp32 $\exp$ overflow boundary at $\exp(88.7) \approx 1.65 \times 10^{38}$.

Segsum method comparison. Figure 8 compares the segsum approaches discussed in Section 4. *Diff Segscan* (the prefix-subtract method) inherits the monotonicity violation: at $\log_2 P = 30, \rho = 0.25$ it emits 29 positive segment arguments across the 256 trials, with worst exponentiated errors of order 10^{27}. *Diff Segscan (0 clamped)* removes the positive arguments, but its near-diagonal argument error is essentially unchanged (21.9 versus 22.0): clamping truncates positive errors to zero while leaving the negative-side cancellation error intact. *Direct Segscan* (the broadcast–mask–cumsum stable form) reduces the near-diagonal mean argument error to 1.03, more than $20\times$ lower, and produces no positive or overflow arguments by construction, so its exponentiated values remain in $(0, 1]$ and the worst exponentiated error is 1.2×10^{-20}.

6 Conclusion and Limitations

Split Sequence decouples local matrix computations from sequence parallelism, while hierarchy-aware segsum applies the 1-semiseparable mask in tensor-core accumulators. The latter avoids diff segsum's monotonicity violations and FP32 overflows at the same speed. Together, these yield CuTe-DSL kernels for Mamba-2/3 that match FA-4 at 1k tokens and run $4\times$ faster than the official Triton kernel. The GB200 results extend these benefits to Blackwell, with 4.60–$5.09\times$ speedups over the official Mamba-2 baseline. Our kernels target training and prefill, not autoregressive decoding, and each new linear-time architecture requires re-deriving its matrix structure—though matching structure to the GPU hierarchy should transfer broadly.

References

Rishi Astra, Tri Dao, and Adnan Hoque. Accelerating mamba2 with kernel fusion – pytorch. URL https://pytorch.org/blog/accelerating-mamba2-with-kernel-fusion/.

Michael Bauer, Henry Cook, and Brucek Khailany. CudaDMA: Optimizing GPU memory bandwidth via warp specialization. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450307710. doi: 10.1145/2063384.2063400. URL https://doi.org/10.1145/2063384.2063400.

Maximilian Beck, Korbinian Pöppel, Phillip Lippe, and Sepp Hochreiter. Tiled flash linear attention: More efficient linear rnn and xlstm kernels, 2025. URL https://arxiv.org/abs/2503.14376.

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

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022. URL https://arxiv.org/abs/2205.14135.

inclusionAI. cula: Cuda linear attention kernels. GitHub repository, 2026. URL https://github.com/inclusionAI/cuLA.

Kimi Team, 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, 2025. URL https://arxiv.org/abs/2510.26692.

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

NVIDIA. Nvidia Tesla V100 GPU architecture, 2017.

NVIDIA. Nvidia A100 tensor core GPU architecture, 2020.

NVIDIA. Nvidia H100 tensor core GPU architecture, 2022a.

NVIDIA. Nvidia h100 tensor core gpu white paper. Technical report, NVIDIA, 2022b. URL https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c.

NVIDIA. Add Mamba-2 forward and backward support for FROST LA. cuDNN Frontend, GitHub pull request #1577, 2026. URL https://github.com/NVIDIA/cudnn-frontend/pull/1577.

NVIDIA, :, Aaron Blakeman, Aarti Basant, Abhinav Khattar, Adithya Renduchintala, Akhiad Bercovich, Aleksander Ficek, Alexis Bjorlin, Ali Taghibakhshi, Amala Sanjay Deshmukh, Ameya Sunil Mahabaleshwarkar, Andrew Tao, Anna Shors, Ashwath Aithal, Ashwin Poojary, Ayush Dattagupta, Balaram Buddharaju, Bobby Chen, Boris Ginsburg, Boxin Wang, Brandon Norick, Brian Butterfield, Bryan Catanzaro, Carlo del Mundo, Chengyu Dong, Christine Harvey, Christopher Parisien, Dan Su, Daniel Korzekwa, Danny Yin, Daria Gitman, David Mosallanezhad, Deepak Narayanan, Denys Fridman, Dima Rekesh, Ding Ma, Dmytro Pykhtar, Dong Ahn, Duncan Riach, Dusan Stosic, Eileen Long, Elad Segal, Ellie Evans, Eric Chung, Erick Galinkin, Evelina Bakhturina, Ewa Dobrowolska, Fei Jia, Fuxiao Liu, Gargi Prasad, Gerald Shen, Guilin Liu, Guo Chen, Haifeng Qian, Helen Ngo, Hongbin Liu, Hui Li, Igor Gitman, Ilia Karmanov, Ivan Moshkov, Izik Golan, Jan Kautz, Jane Polak Scowcroft, Jared Casper, Jarno Seppanen, Jason Lu, Jason Sewall, Jiaqi Zeng, Jiaxuan You, Jimmy Zhang, Jing Zhang, Jining Huang, Jinze Xue, Jocelyn Huang, Joey Conway, John Kamalu, Jon Barker, Jonathan Cohen, Joseph Jennings, Jupinder

Parmar, Karan Sapra, Kari Briski, Kateryna Chumachenko, Katherine Luna, Keshav Santhanam, Kezhi Kong, Kirthi Sivamani, Krzysztof Pawelec, Kumar Anik, Kunlun Li, Lawrence McAfee, Leon Derczynski, Lindsey Pavao, Luis Vega, Lukas Voegtle, Maciej Bala, Maer Rodrigues de Melo, Makesh Narsimhan Sreedhar, Marcin Chochowski, Markus Kliegl, Marta Stepniewska-Dziubinska, Matthieu Le, Matvei Novikov, Mehrzad Samadi, Michael Andersch, Michael Evans, Miguel Martinez, Mike Chrzanowski, Mike Ranzinger, Mikolaj Blaz, Misha Smelyanskiy, Mohamed Fawzy, Mohammad Shoeybi, Mostofa Patwary, Nayeon Lee, Nima Tajbakhsh, Ning Xu, Oleg Rybakov, Oleksii Kuchaiev, Olivier Delalleau, Osvald Nitski, Parth Chadha, Pasha Shamis, Paulius Micikevicius, Pavlo Molchanov, Peter Dykas, Philipp Fischer, Pierre-Yves Aquilanti, Piotr Bialecki, Prasoon Varshney, Pritam Gundecha, Przemek Tredak, Rabeeh Karimi, Rahul Kandu, Ran El-Yaniv, Raviraj Joshi, Roger Waleffe, Ruoxi Zhang, Sabrina Kavanaugh, Sahil Jain, Samuel Kriman, Sangkug Lym, Sanjeev Satheesh, Saurav Muralidharan, Sean Narenthiran, Selvaraj Anandaraj, Seonmyeong Bak, Sergey Kashirsky, Seungju Han, Shantanu Acharya, Shaona Ghosh, Sharath Turuvekere Sreenivas, Sharon Clay, Shelby Thomas, Shrimai Prabhumoye, Shubham Pachori, Shubham Toshniwal, Shyamala Prayaga, Siddhartha Jain, Sirshak Das, Slawek Kierat, Somshubra Majumdar, Song Han, Soumye Singhal, Sriharsha Niverty, Stefania Alborghetti, Suseella Panguluri, Swetha Bhendigeri, Syeda Nahida Akter, Szymon Migacz, Tal Shiri, Terry Kong, Timo Roman, Tomer Ronen, Trisha Saar, Tugrul Konuk, Tuomas Rintamaki, Tyler Poon, Ushnish De, Vahid Noroozi, Varun Singh, Vijay Korthikanti, Vitaly Kurin, Wasi Uddin Ahmad, Wei Du, Wei Ping, Wenliang Dai, Wonmin Byeon, Xiaowei Ren, Yao Xu, Yejin Choi, Yian Zhang, Ying Lin, Yoshi Suhara, Zhiding Yu, Zhiqi Li, Zhiyu Li, Zhongbo Zhu, Zhuolin Yang, and Zijia Chen. Nemotron-h: A family of accurate and efficient hybrid mamba-transformer models, 2025. URL https://arxiv.org/abs/2504.03624.

Qwen Team. Qwen3-Next. Model release, 2025. URL https://qwen.ai/blog?id=qwen3-next.

Qwen Team. Flashqla: Cp-/bwd-friendly fused linear attention kernels for gdn. GitHub repository, 2026. URL https://github.com/QwenLM/FlashQLA.

Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision, 2024. URL https://arxiv.org/abs/2407.08608.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017. URL http://arxiv.org/abs/1706.03762.

Lei Wang, Yu Cheng, Yining Shi, Zhengju Tang, Zhiwen Mo, Wenhao Xie, Lingxiao Ma, Yuqing Xia, Jilong Xue, Fan Yang, and Zhi Yang. Tilelang: A composable tiled programming model for ai systems, 2025. URL https://arxiv.org/abs/2504.17577.

Songlin Yang, Jan Kautz, and Ali Hatamizadeh. Gated delta networks: Improving mamba2 with delta rule, 2024a. URL https://arxiv.org/abs/2412.06464.

Songlin Yang, Bailin Wang, Yikang Shen, Rameswar Panda, and Yoon Kim. Gated linear attention transformers with hardware-efficient training, 2024b. URL https://arxiv.org/abs/2312.06635.

Ted Zadouri, Markus Hoehnerbach, Jay Shah, Timmy Liu, Vijay Thakkar, and Tri Dao. Flashattention-4: Algorithm and kernel pipelining co-design for asymmetric hardware scaling, 2026. URL https://arxiv.org/abs/2603.05451.

A Split Sequence Algorithm and Kernel Implementation

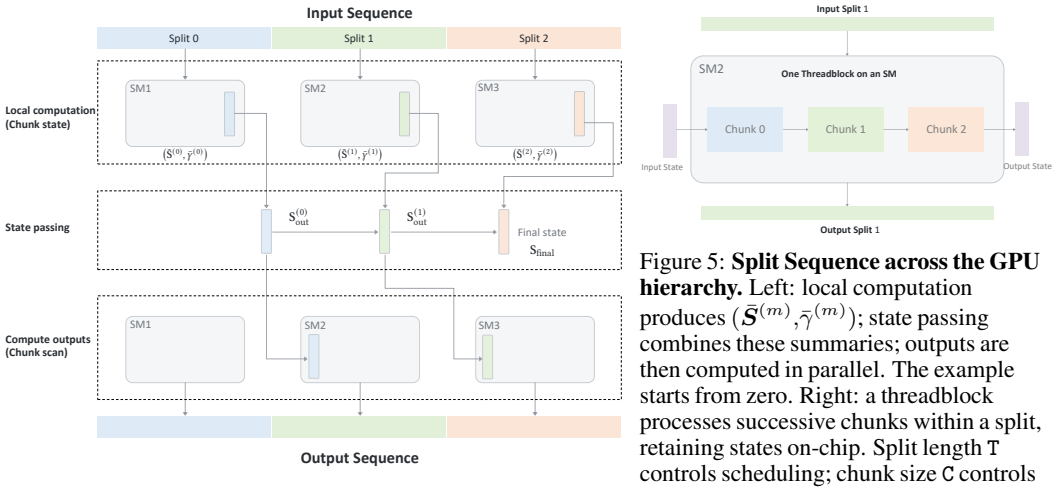


Figure 5: **Split Sequence across the GPU hierarchy.** Left: local computation produces $(\bar{\boldsymbol{S}}^{(m)},\bar{\gamma}^{(m)})$; state passing combines these summaries; outputs are then computed in parallel. The example starts from zero. Right: a threadblock processes successive chunks within a split, retaining states on-chip. Split length `T` controls scheduling; chunk size `C` controls local matrix computation.

A.1 Complete Algorithm

The following algorithm combines the three stages of Section 3.1.1 with the recurrent chunk computation described in Section 3.1.3. The chunk subroutine evaluates Equations (4) and (5); its decay-mask operation is implemented by the Mask Subroutine in Appendix D.1.

Split Sequence

Inputs: $\boldsymbol{B},\boldsymbol{C},\boldsymbol{X},a,\Delta$, initial state $\boldsymbol{S}_{\text{init}}$, lengths `T`,`C`
Outputs: $\boldsymbol{O}$ and final state $\boldsymbol{S}_{\text{final}}$

The CHUNK SUBROUTINE computes Equations (4) and (5)
Chunk decay factors $\beta_{[n]},\gamma_{[n]}$ are defined in Equation (3)

Partition chunks into `J` ordered splits $I_0,\ldots,I_{\mathtt{J}-1}$, each of nominal length `T`
Stage 1: Local computation (chunk state)
for all splits m in parallel, one threadblock per split **do**
  $\bar{\boldsymbol{S}}\leftarrow 0,\ \bar{\gamma}\leftarrow 1$
  for chunks $n\in I_m$ in sequence order **do**
    $\bar{\boldsymbol{S}}\leftarrow\gamma_{[n]}\bar{\boldsymbol{S}}+\boldsymbol{B}_{[n]}^{\top}\mathrm{diag}(\beta_{[n]})\boldsymbol{X}_{[n]}$ ▷ Equation (5)
    $\bar{\gamma}\leftarrow\gamma_{[n]}\bar{\gamma}$
  end for
  Write summary $(\bar{\boldsymbol{S}}^{(m)},\bar{\gamma}^{(m)})\leftarrow(\bar{\boldsymbol{S}},\bar{\gamma})$ to HBM
end for
Stage 2: State passing
$\boldsymbol{S}\leftarrow\boldsymbol{S}_{\text{init}}$
for $m=0,\ldots,\mathtt{J}-1$ **do**
  Write $\boldsymbol{S}_{\text{in}}^{(m)}\leftarrow\boldsymbol{S};\quad\boldsymbol{S}\leftarrow\bar{\gamma}^{(m)}\boldsymbol{S}+\bar{\boldsymbol{S}}^{(m)}$
end for
Stage 3: Compute outputs (chunk scan)
for all splits m in parallel, one threadblock per split **do**
  $\boldsymbol{S}\leftarrow\boldsymbol{S}_{\text{in}}^{(m)}$
  for chunks $n\in I_m$ in sequence order **do**
    $(\boldsymbol{O}_{[n]},\boldsymbol{S})\leftarrow$ CHUNK SUBROUTINE$(\boldsymbol{S};n)$ ▷ state stays local
    Write $\boldsymbol{O}_{[n]}$ to HBM
  end for
  if $m=\mathtt{J}-1$ **then**
    Write $\boldsymbol{S}_{\text{final}}\leftarrow\boldsymbol{S}$
  end if
end for
return $(\boldsymbol{O},\boldsymbol{S}_{\text{final}})$

A.2 Kernel Implementation

The split-sequence algorithm fixes the threadblock-level schedule: each CTA owns one (b,h,m) split and processes the $\mathtt{T}/\mathtt{C}$ chunks inside it. The remaining goal is to organize work inside the CTA so that non-matmul operations do not stall tensor-core execution. Our Hopper kernel uses three cooperating warp groups: a *producer* that streams B, C, X and decay parameters into shared memory, a *state* group that holds the recurrent state $S \in \mathbb{R}^{\mathtt{D}\times\mathtt{D}_\mathtt{v}}$ in registers, and an *output* group that computes $CB^\top$, CS, and PX.

Fusing chunk-local computation. Within a split, chunks are processed sequentially, but each chunk is computed using a warp-specialized pipeline. For chunk $[n]$, the state warp group updates the register-resident state with

$$S \leftarrow \gamma_{[n]} S + B_{[n]}^\top \mathrm{diag}(\beta_{[n]}) X_{[n]}. \tag{14}$$

In parallel, the output warp group computes the chunk-local interaction matrix $C_{[n]}B_{[n]}^\top$, applies the causal segment-sum mask,

$$P_{[n]} = \left(C_{[n]}B_{[n]}^\top\right) \odot E_{[n]}, \tag{15}$$

and then computes $P_{[n]}X_{[n]}$ before adding the inter-chunk contribution $C_{[n]}S_{[n]}$. Thus the large intermediates—the recurrent state, the score matrix $CB^\top$, and the masked matrix P—remain in registers or shared memory. The kernel only writes the final outputs, and in Step 1 of the split sequence algorithm, the final split state.

Forward and reverse scan templates. The scan direction is determined by the triangular structure of the segment-sum matrix. In the forward pass, $E_{[n]}$ is lower triangular, so both the chunk recurrence and split-level state passing run left-to-right. The dC kernel has the same direction and reuses the forward-scan template with operands permuted. In contrast, the dB and dX kernels involve the transpose mask $E_{[n]}^\top$, which is upper triangular and therefore induces a right-to-left recurrence. These kernels use the reverse-scan template: the CTA sweeps chunks from right to left, and split-level state passing also runs right-to-left.

The three-stage schedule is illustrated in Figure 5 and specified by the Split Sequence algorithm above.

B GPU Architecture Background

Memory hierarchy. Storage on modern GPUs is hierarchical. Global memory (GMEM), also known as high-bandwidth memory (HBM), is the off-chip DRAM that is accessible to all processors. The execution units on a GPU are organized into streaming multiprocessors (SMs), and each SM contains shared memory (SMEM), a programmer-managed on-chip memory for staging and reusing data loaded from HBM, as well as a register file (RMEM) that provides storage for the threads within each SM. As compute throughput has outpaced memory bandwidth [NVIDIA, 2017, 2020, 2022a], the effective utilization of SMEM and RMEM to avoid repeated HBM accesses has become critical.

Thread hierarchy. Modern GPUs contain parallel streaming multiprocessors (SMs) and specialized units for low-precision matrix multiplication. Each SM manages a number of threadblocks, also called cooperative thread arrays (CTAs). Within each threadblock, threads are grouped into warps of 32 threads that execute in lockstep. Threads within a warp can communicate using fast shuffle instructions, while warps within a threadblock can communicate through shared memory. The thread and memory hierarchies are closely interlinked: registers are associated with individual threads, SMEM is shared among threads within a CTA, and HBM is accessible across the GPU. On NVIDIA Hopper GPUs, the thread hierarchy further includes warpgroups, which consist of four contiguous warps, and threadblock clusters, which group threadblocks. Further, the hardware support for asynchrony allows for warp-specialized kernels, where the warps of a CTA are designated into producer or consumer roles that issue either data movement or computation, improving the compiler's ability to generate optimal instruction schedules [Bauer et al., 2011]. Given this architecture, an ideal kernel algorithm would leverage asynchrony to maximize memory bandwidth and compute throughput, while minimizing data movement across all levels of memory hierarchy.

C Related Work

Hardware-efficient sequence modeling implementations. There has been many recent works that focuses on improving the speed and efficiency of sequence models by developing hardware-aware algorithms to compute them. The Flash Attention family is one notable example which has focused on

progressively speeding up the attention mechanism [Vaswani et al., 2017]. Earlier versions of Flash Attention focused on parallelism through online softmax and better partitioning, and recent iterations have focused on utilizing newly introduced GPU hardware in both the Hopper and Blackwell families to improve asynchronity and non-matmul operation speeds.

Similar principles have recently been applied to attention alternatives, namely linear recurrent models. While Mamba-1 utilized a hardware-aware scan-based algorithm, it was still less hardware efficient than attention-based mechanisms due to the lack of matmuls used. Chunk-based formulations of Gated Linear Attention (part of Flash Linear Attention) [Yang et al., 2024b], Gated DeltaNet [Yang et al., 2024a], and Mamba-2 [Dao and Gu, 2024] were able to represent the recurrence in a matrix-form which enabled the usage of dedicated matrix multiplication units in modern GPUs.

However, one constraint faced by many chunked algorithms is the large HBM memory movement associated with the storage of the state materialized at each chunk boundary, especially when the total sequence length grows large. To counteract this, Yang et al. [2024b] also introduced a non-materialized version of the GLA implementation which parallelized across batch size, heads, and head dimensions and iteratively processed across the sequence length. Tiled Flash Linear Attention [Beck et al., 2025] also aimed to reduce the number of states that needed to be materialized by increasing the chunk size $\mathtt{C}$ by adding tiling inside the larger chunk.

One aspect of our work similarly addresses the HBM I/O stemming from state materialization; however, we provide a hierarchical algorithm which utilizes modern GPU functionality by partitioning the entire sequence across CTAs where each CTA processes multiple chunks, thereby reducing the states needed to be communicated across splits. Concurrently, FlashQLA [Qwen Team, 2026] has leveraged warp-specialization to attain considerable speedup in GDN training, and cuLA [inclusionAI, 2026] has similarly exploited hardware features such as TMA copy to attain fast prefill kernels for Kimi Delta Attention. In addition, our work also addresses the segment-sum issue found in current linear models. By exploiting the structure of the decay mask, our algorithm is able to utilize the native hardware hierarchy and retain both stability and speed during computation.

D Hierarchy-Aware Segsum Details

D.1 Mask Construction and Application

Mask Subroutine

Inputs: Distributed accumulator $\boldsymbol{A}$, log-decays $a_i \le 0$, subblock size s dividing $\mathtt{C}$
Outputs: $\boldsymbol{A} \odot \boldsymbol{E}$, updated in place

Phase 1: precompute shared quantities

Divide the chunk into contiguous groups of s tokens, numbered from zero
Compute within-group suffix sums and group total values by direct scans
Share suffix and total with their consuming threads

Phase 2: construct and apply the mask

for all row groups g in parallel, in their owning threads **do**
    Compute prefix_i and local diagonal interval sums by direct scans; retain them locally
    Set owned entries in column groups to the right of g to zero
    In the diagonal block, set $A_{ij} \leftarrow A_{ij} \exp\left(\sum_{k=j+1}^{i} a_k\right)$ for $j \le i$, and 0 otherwise
    $r_i \leftarrow \mathsf{prefix}_i$ for each owned row i ▷ row prefix plus intervening groups
    for $h = g-1, g-2, \ldots, 0$ **do** ▷ right-to-left *mask column groups*
        For owned entries in column group h, set $A_{ij} \leftarrow A_{ij} \exp(r_i + \mathsf{suffix}_j)$
        $r_i \leftarrow r_i + \mathsf{total}_h$ ▷ extend the interval to the next group on the left
    end for
end for
return $\boldsymbol{A}$ ▷ now equals $\boldsymbol{P} = (\boldsymbol{C}\boldsymbol{B}^\top) \odot \boldsymbol{E}$

D.2 Example of the 1-semiseparable Block Decomposition

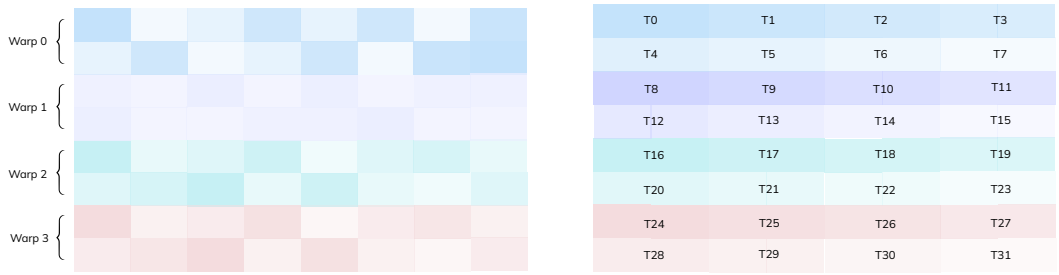
This structure is illustrated by an 8×8 mask with block size 4. For readability, we index tokens from 1 to 8 and write $\alpha_k = e^{a_k}$:

$$E = \begin{array}{c|cccc|cccc} & \multicolumn{4}{c|}{G_0} & \multicolumn{4}{c}{G_1} \\ \hline G_0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ & \alpha_2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ & \alpha_2\alpha_3 & \alpha_3 & 1 & 0 & 0 & 0 & 0 & 0 \\ & \alpha_2\alpha_3\alpha_4 & \alpha_3\alpha_4 & \alpha_4 & 1 & 0 & 0 & 0 & 0 \\ \hline G_1 & \alpha_2\cdots\alpha_5 & \alpha_3\cdots\alpha_5 & \alpha_4\alpha_5 & \alpha_5 & 1 & 0 & 0 & 0 \\ & \alpha_2\cdots\alpha_6 & \alpha_3\cdots\alpha_6 & \alpha_4\cdots\alpha_6 & \alpha_5\alpha_6 & \alpha_6 & 1 & 0 & 0 \\ & \alpha_2\cdots\alpha_7 & \alpha_3\cdots\alpha_7 & \alpha_4\cdots\alpha_7 & \alpha_5\cdots\alpha_7 & \alpha_6\alpha_7 & \alpha_7 & 1 & 0 \\ & \alpha_2\cdots\alpha_8 & \alpha_3\cdots\alpha_8 & \alpha_4\cdots\alpha_8 & \alpha_5\cdots\alpha_8 & \alpha_6\cdots\alpha_8 & \alpha_7\alpha_8 & \alpha_8 & 1 \end{array}$$

The diagonal blocks are smaller 1-SS matrices, the upper-right block is zero by causality, and the lower-left block is rank one with middle aggregate 1, since there are no intervening groups:

$$E_{G_1,G_0} = \underbrace{\begin{pmatrix} \alpha_5 \\ \alpha_5\alpha_6 \\ \alpha_5\alpha_6\alpha_7 \\ \alpha_5\alpha_6\alpha_7\alpha_8 \end{pmatrix}}_{\text{row factor}} \underbrace{1}_{\text{middle aggregate}} \underbrace{\begin{pmatrix} \alpha_2\alpha_3\alpha_4 & \alpha_3\alpha_4 & \alpha_4 & 1 \end{pmatrix}}_{\text{column factor}}.$$

D.3 WGMMA Accumulator Layout



(a) Warp-level decomposition.

(b) Thread-level decomposition.

Figure 6: Native WGMMA accumulator layout for a 64×64 output tile. (Left) The tile is distributed across a warp-group of four warps. Each square denotes an 8×8 sub-block; each warp owns a contiguous 16×64 row stripe, equivalently a 2×8 block row in the collapsed view. (Right) Within one 8×8 sub-block, the entries are distributed across the 32 threads of a warp, with each thread holding two row-contiguous accumulator entries. We map the 1-SS segsum decomposition onto this layout and apply decay factors directly to the $CB^\top$ accumulator fragments.

E Numerical Analysis

E.1 Illustrative Examples for Segsum Numerical Failure Modes

E.1.1 Monotonicity Violation Example

Consider the case $c = 4$, with large initial $a_0 = -2^{27}$ and tail values $a_1 = -6.46$, $a_2 = -2.71$, $a_3 = -1.34$. Note that with respect to a_0, the tail values are less than half the ULP threshold, $\mathrm{ulp}(2^{27})/2 = 8$. Sequential cumsum therefore drops every tail increment ($s_0 = s_1 = s_2 = s_3 = -2^{27}$) and so preserves monotonicity trivially. Hillis–Steele scan, however, realizes the prefix sums by differing orders of operations

$$s_2 = a_0 + (a_1 + a_2), \qquad s_3 = (a_0 + a_1) + (a_2 + a_3).$$

The pair $a_1 + a_2 = -9.17$ *exceeds* the half-ULP threshold and rounds s_2 down by one ULP to $-2^{27} - 16$; the pair $a_2 + a_3 = -4.05$ does not, so s_3 stays at -2^{27}. The fp32 difference is $s_3 - s_2 = +16$, even though the exact value $a_3 = -1.34 < 0$. The monotonicity requirement is thus violated.

E.1.2 Catastrophic Cancellation Example

Consider the following two prefixes:

$$s_j = -(2^{30}+63), \qquad s_i = -(2^{30}+65), \qquad \text{true } M_{ij} = s_i - s_j = -2.$$

Here, $P = 2^{30}$, with $\mathrm{ulp}(P) = 128$, thus the rounding threshold is 64. Round-to-nearest results in $s_j \to -2^{30}$ (since $63 < 64$) and $s_i \to -2^{30} - 128$ (since $65 > 64$); the two prefixes round in *opposite* directions, and the rounded difference is -128, an error of 126 on a true value of 2.

E.2 Error Scaling in Segsum

Let $u = 2^{-24}$ denote the fp32 unit roundoff, and write $\gamma_n \coloneqq nu/(1-nu)$. Standard floating-point summation bounds imply that a sum formed through n rounded additions incurs a forward error bounded by γ_n times the relevant accumulation scale.

The third step in *direct segsum* forms $M_{ij} = \sum_{k=j+1}^{i} a_k$ by summing the segment increments directly. Because all $a_k \le 0$, every partial sum is also non-positive, and its magnitude is bounded by the magnitude of the final segment sum. Hence

$$|\hat{M}_{ij}^{\mathrm{direct}} - M_{ij}| \le \gamma_{i-j} \sum_{k=j+1}^{i} |a_k| = \gamma_{i-j}|M_{ij}| \approx (i-j)u|M_{ij}|.$$

For *diff segsum*, the segsum is computed via $M_{ij} = s_i - s_j$. Let $P_{ij} \coloneqq \max(|s_i|,|s_j|)$, and let λ denote the longest addition path in the prefix algorithm ($\lambda = c-1$ for sequential cumsum and $\lambda = \log_2 c$ for an ideal balanced associative scan). Then each prefix already carries absolute error of order $\gamma_\lambda P_{ij}$, and the final subtraction contributes one additional rounding step. Writing $\hat{M}_{ij}^{\mathrm{diff}} = \mathrm{round}(\hat{s}_i - \hat{s}_j)$, we can show the following bound:

$$|\hat{M}_{ij}^{\mathrm{diff}} - M_{ij}| \le \big(2\gamma_\lambda + 2u + 2u\gamma_\lambda\big) P_{ij}.$$

Thus, direct segsum is controlled by the magnitude of the *segsum entry* being queried, whereas diff segsum is controlled by the much larger *prefix* scale. In the near-diagonal regime, where $|M_{ij}| \ll P_{ij}$, the latter can be worse by a factor on the order of $P_{ij}/|M_{ij}|$, up to the modest ratio between γ_λ and γ_{i-j}. Clamping cannot remove this gap, because it corrects sign violations but does not reduce the absolute error caused by subtracting two large rounded prefixes.

F Numerical Stability Experiment Results

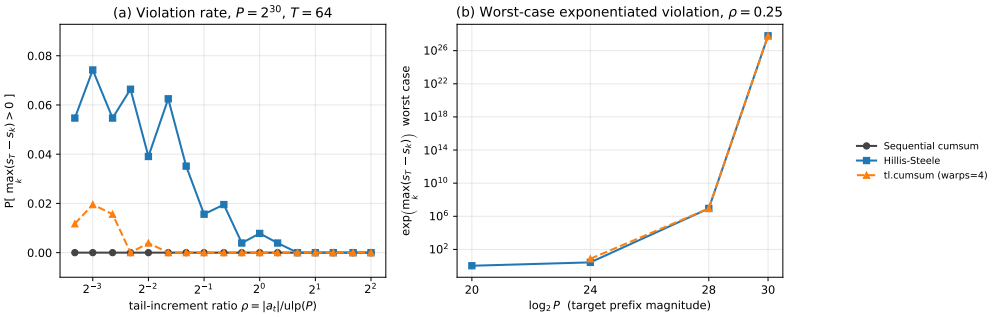


Figure 7: Monotonicity violations of fp32 prefix scans, $T = 64$, with 256 trials. *(a)* Probability that the scan emits some $s_T - s_k > 0$ at $P = 2^{30}$, as a function of the tail-to-ULP ratio ρ. The sequential cumsum (zero across the sweep) is the only scan that preserves the rounded invariant. *(b)* Worst-case exponentiated violation $\exp(\max_k(s_T - s_k))$ at $\rho = 0.25$, log-y axis.

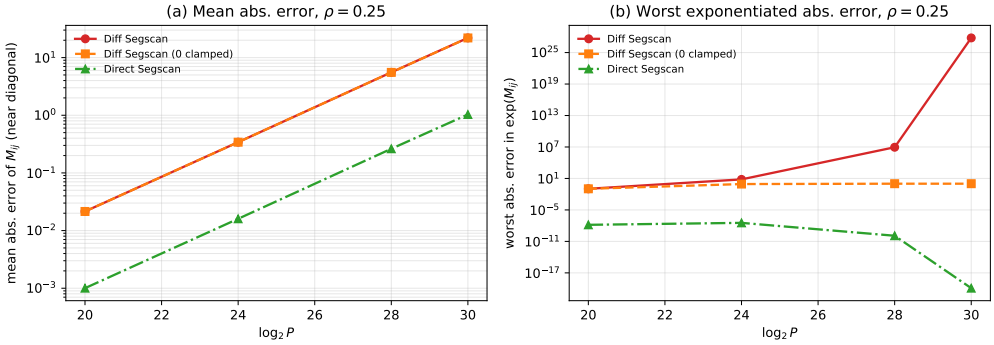


Figure 8: Segsum error along the strict near-diagonal band ($1 \le i - j \le 4$), with source prefix from `tl.cumsum(num_warps=4)`, $\rho = 0.25$, averaged over 256 trials. The reference M_{ij}^{ref} is computed by direct segsum in fp64, and all error metrics are evaluated in fp64. *(a)* Mean argument error $|M_{ij}^{\mathrm{fp32}} - M_{ij}^{\mathrm{ref}}|$ versus prefix scale $\log_2 P$. *(b)* Worst-case error of the exponentiated segsum $|\exp(M_{ij}^{\mathrm{fp32}}) - \exp(M_{ij}^{\mathrm{ref}})|$.

G Comparison to optimized linear-time layer baselines

In addition to comparing against the official Mamba-2 implementation, we compare the Mamba-2 forward kernel introduced in this paper against the most optimized external implementations of related linear-time layers that we could find. The fastest implementation in this benchmark is our current Mamba-2 kernel; the purpose of the comparison is to contextualize it against highly engineered baselines for nearby architectures. This comparison is not meant to be an apples-to-apples model comparison: Mamba-2, Gated DeltaNet (GDN), and Kimi Delta Attention (KDA) implement slightly different recurrences and have different gating/state-update structure. In particular, GDN and KDA are algorithmically more complicated than Mamba-2, so achieving raw `memcpy`-level bandwidth is intrinsically harder for those layers.

Models and implementations. We include four forward-pass implementations, all measured on an NVIDIA H100 with bfloat16 activations and fixed total token count.

- **Mamba-2 / SSD** [Dao and Gu, 2024]. This is the scalar-decay state-space duality layer studied in the main text. **Mamba2 (ours)** is our CuTe-DSL implementation, using the split-sequence and hierarchy-aware segsum optimizations described in this work.
- **Mamba-2 in TileLang.** **Mamba2 (TileLang)** is a public TileLang [Wang et al., 2025] example implementation of the Mamba-2/SSD forward pass. It is useful as an independent DSL baseline for the same high-level layer.

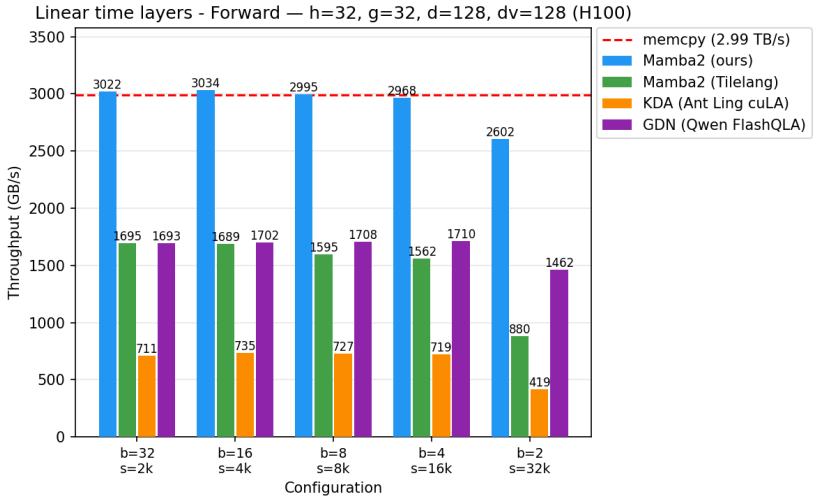


Figure 9: Forward throughput of optimized linear-time layer implementations on H100. All runs use bfloat16 activations with $\mathtt{H} = 32$, $g = 32$, $\mathtt{D} = 128$, and $\mathtt{D_v} = 128$. The dashed red line is the measured `memcpy` reference bandwidth. The models are related but not identical, so the comparison should be interpreted as a systems-level constant-factor comparison rather than a claim of exact semantic equivalence.

- **KDA.** Kimi Delta Attention is the core linear-attention mechanism in Kimi Linear [Kimi Team et al., 2025]. **KDA (Ant Ling cuLA)** is the cuLA implementation [inclusionAI, 2026] from Ant Ling, written in CUTLASS/C++ and targeting Hopper. This is a heavily optimized systems implementation rather than a simple reference kernel.
- **GDN.** Gated DeltaNet [Yang et al., 2024a] augments the delta rule with a learned gate/decay. **GDN (Qwen FlashQLA)** is Qwen's FlashQLA implementation [Qwen Team, 2026], written in TileLang [Wang et al., 2025] and optimized for modern NVIDIA GPUs.

Shape caveat. This benchmark uses a different shape from the main-paper Mamba-2 experiments in order to match the constraints of the GDN/KDA baselines. We set $\mathtt{D_v} = 128$ rather than 64 because the GDN and KDA kernels benchmarked here only support $\mathtt{D_v} = 128$; our Mamba-2 kernel supports both. We also set the number of query/key heads equal to the number of value heads ($\mathtt{H}_{qk} = \mathtt{H}_v$, i.e., $g = 32$) for fairness with these baselines, whereas the main-paper experiments use a grouped-query Mamba-2 setting with $\mathtt{H}_{qk} = 1$ and many value heads.

Results. Figure 9 shows that the Mamba-2 kernel from this paper is the fastest implementation overall and reaches the memory bandwidth ceiling for essentially all short and medium sequence lengths: it sustains 2.97–3.03 TB/s from 2k through 16k tokens, matching the measured `memcpy` line of 2.99 TB/s within noise. At 32k tokens the throughput drops to 2.60 TB/s, but even there it remains close to the hardware bandwidth limit and substantially ahead of the other implementations.

The key point is not that Mamba-2, GDN, and KDA are identical layers—they are not—but that the strongest implementations we could find for these nearby linear-time architectures still leave a large gap to raw memory bandwidth. This gap should be interpreted with the model differences in mind: GDN and KDA perform more involved updates than Mamba-2, so they have more arithmetic, dependencies, and kernel-level bookkeeping per byte moved, making the `memcpy` roofline a harder target. Mamba2 (TileLang) reaches roughly 1.6–1.7 TB/s at 2k–16k and falls to 0.88 TB/s at 32k. FlashQLA's GDN kernel is very strong and stable, around 1.7 TB/s up to 16k and 1.46 TB/s at 32k. The cuLA KDA kernel is also a sophisticated CUTLASS/C++ implementation, but in this configuration reaches roughly 0.7 TB/s up to 16k and 0.42 TB/s at 32k.

Reaching the measured `memcpy` roofline leaves little room for further speedup without reducing data movement. At 32k, reduced batch parallelism and sequence-split overhead lower achieved bandwidth, though our kernel remains faster than the other evaluated implementations.

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 accurately reflect the paper's contribution on improvements to the SSM algorithm.
   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 limitations, e.g., focus on WGMMA hardware, in our conclusion section.
   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: [N/A]
   Justification: Our paper does not include theoretical results.
   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 discuss the experimental protocol in Section 5.
   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: The implementation is publicly available at `https://github.com/state-spaces/mamba`. We report hardware, model dimensions, and evaluation procedures, but do not yet provide a complete command-by-command reproduction guide for all experiments.
   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 the experimental settings and details in Section 5.
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: GPU benchmarks are reproducible and highly consistent given similar hardware and setting, which we have detailed.
   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: Our paper provides the hardware and software used to benchmark our kernels.
   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: No additional harms beyond standard concerns exist from this line of research.

   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: [No]

    Justification: No additional harms beyond standard concerns exist from this line of research.

    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: Our work describes algorithmic improvements to improve the latency of SSM models. We are not releasing data or models that have high risk of misuse.

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: The open source assets that we use have been properly credited.

    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: We do not release any new assets at this time.

    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: Our paper does not work 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: Our research did not interact with participants.

    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: [Yes]
Justification: We utilized LLMs for writing, editing, and formatting, and used coding agents to help implement our algorithm in CuTe.
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.