FORGE: COMPILING A UNIFIED ABSTRACTION INTO SCALABLE KERNELS FOR LINEAR ATTENTION

Anonymous authors
Paper under double-blind review

ABSTRACT

The quadratic complexity of softmax attention poses a major bottleneck for long-context modeling, motivating a surge of linear attention variants with linear complexity. Unlike softmax attention, which benefits from optimized kernels, linear attention lacks general-purpose, hardware-efficient support and scalable distributed implementations. We introduce Forge, a domain-specific compiler that automates the generation of high-performance, scalable kernels for a wide range of linear attention models directly from high-level PyTorch code. At its core, Forge employs an intuitive programming abstraction that decomposes any linear attention algorithm into three canonical phases: intra-chunk computation, inter-chunk state propagation, and output merging. This unified abstraction enables Forge to perform domain-specific optimizations, automatically generating kernels that fuse computation and communication at a fine-grained tile level and eliminating host synchronization. Our evaluation demonstrates that Forge combines programmability with performance: a wide range of linear attention variants can be implemented in just a few dozen lines of code, while the generated kernels deliver 1.01x-4.9x the performance of sate-of-the-art expert-optimized library and scale with near-linear efficiency on scalar gated linear attention to 16 million tokens on 128 GPUs, surpassing the state-of-the-art distributed baseline by up to 7.2x.

1 INTRODUCTION

Transformer models rely on self-attention, which has quadratic time and memory complexity with respect to sequence length. As models handle increasingly long contexts, this quadratic bottleneck severely limits scalability. In response, many efficient-attention mechanisms have been proposed. In particular, linear attention methods remove the softmax nonlinearity and reorder computations to achieve linear computation complexity and constant-memory inference. This has led to a proliferation of innovative architectures, such as Mamba (Gu & Dao, 2023; Dao & Gu, 2024), RetNet (Sun et al., 2023), RWKV (Peng et al., 2023), GLA (Yang et al., 2023), HGRN (Qin et al., b) and Gated DeltaNet (Yang et al., 2024a). These architectures demonstrate capabilities competitive with, or even superior to, standard transformers.

Unlike softmax attention, which has benefited from highly optimized and now-standardized kernels like Flash-Attention (Dao et al., 2022) and Ring-Attention (Liu et al., 2023) that efficiently map its computation and communication to modern AI infrastructure, the landscape for linear attention is far more fragmented. Flash-Linear-Attention (FLA) (Yang & Zhang, 2024) provide a valuable collection of triton kernels for a series of linear attention variants. But it essentially relies on expert developers to provide manual implementation for each variants. The rapid evolution of linear attention variants means that a one-size-fits-all solution does not exist. This forces researchers into a costly and inefficient cycle of manual kernel development for each new variant, a process fraught with two major challenges.

First, the implementation of high-performance kernels is an arduous task requiring deep hardware expertise. While many linear attention models share a conceptual similarity, their specific state update rules and memory access patterns can differ substantially. Achieving hardware efficiency necessitates not only fusing the state update rule into a single kernel but also manually tuning hardware-specific parameters like pipeline schedules and tile sizes. Even with high-level DSLs like

PyTorch Implementation

```
def scalar_gla(q, k, v, g, s, o, scale):
  # q/k/v: [BHTD]. g: [BHT]. s: [BHDD]
  for bh in range(B * H):
    b, h = bh // H, bh % H
    state = s[b, h, :, :].clone()
    for chunk_idx in range(0, T, CHUNK_SIZE):
      c_q, c_k, c_v, c_g, c_o = get_chunk(…)
      # 1. compute intra-chunk decay
      g_cumsum, g_sum = c_g.cumsum(0), c_g.sum(0)
      decay_sum,decay_vec=g_sum.exp(), g_cumsum.exp()
      decay_k = (g_local_sum-g_cumsum).exp()[None, :]
      # 2. compute intra-chunk state
      c_k_decay  = (c_k * decay_k)
      curr_state = (c_k_decay @ c_v)
      # 3. compute output of current chunk
      decay_p = decay_vec[:, None]-decay_vec[None, :]
      p = ((chunk_q @ c_k.T).tril(0) * decay_p)
      curr_state_ = curr_state * decay_vec[:, None]
      c_o = (p @ c_v)+(c_q @ curr_state_) * scale
      # 4. update state
      state = state * decay_sum + curr_state
  return o
```

Our DSL

```
def chunk_mode(k, v, g):
  g_cumsum, g_sum = g.cumsum(0), g.sum(0)
  g_cumsum        = (g_sum - g_cumsum[None, :]).exp()
  k_decay         = (k.T * g_cumsum.exp())
  return k_decay @ v

def decay_mode(old_s, s, g):
  return old_s * g.sum(0).exp() + s

def merge_mode(q, k, v, g, s, scale):
  decay_vec = g.cumsum(0).exp()
  decay_p   = decay_vec[:, None] - decay_vec[None, :]
  qk        = (q @ k.T).tril(0)
  p         = (qk * decay_p)
  return (p @ v + q @ s * decay_vec[:, None]) * scale

LinearAttention(
  SP_GROUP,
  chunk_mode,
  decay_mode,
  merge_mode,
)
```

Performance

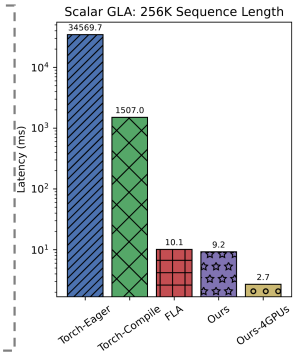


Figure 1: Comparison between PyTorch and our proposed DSL in writing scalar gated linear attention (data type conversion is omitted for simplicity). And the performance comparison of different approaches. Test shape: 32 heads with 128 head dimension. Torch-Eager fails to parallelize the computation, while Torch-Compile also performs poorly. FLA provides expert optimized triton kernels, while program generated by Forge offers comparable performance to handwritten kernels with the scalability to distributed environments.

Triton (Tillet & Cox, 2019), developers must often delve into low-level hardware details, such as managing barriers or dealing with shared memory capacity limitation, to extract maximum performance for each variant. This creates a high barrier to entry and slows down the pace of innovation.

Second, existing solutions lack robust support for distributed execution, which is non-negotiable for scaling to contexts of hundreds of thousands or millions of tokens. FLA offers a bunch of single-device kernels but do not address the distributed scaling problem. When sequence lengths exceed the memory capacity of a single accelerator, distributed sequence parallelism becomes essential. However, enabling sequence-parallel linear attention is non-trivial: it typically requires custom communication schedules tailored to each variant's state update rule. Existing sequence parallel schemes, such as LASP and LASP-2 (Sun et al., 2024a; 2025) are designed for specific architectures and employ generic communication primitives (e.g., `All-Gather` from NCCL). The mismatch of existing communication primitive and dataflow of distributed linear attention leading to significant network bandwidth underutilization(Chou et al., 2025).

Can we provide a solution to bridge the gap between the rapid evolution of linear attention algorithms and the difficulty of developing scalable kernels? We observe that many of these difficulties stem from not exploiting the common structure underlying linear-attention variants. Our central insight is that most linear-attention variants share a small set of canonical operations and data exchanges. Based on this, we introduce Forge, a compiler-driven framework that allows implementing the majority of linear attention variants in a few lines of idiomatic PyTorch code and scaling them to distributed system. As shown in Figure 1, our DSL expresses linear attention in three modular functions. The compiler translates the DSL into high performance kernels: reducing latency from 34.6 seconds (PyTorch eager) to 9.2 ms, even better than SOTA hand-written kernel from FLA. More importantly, the latency was further reduced to 2.7 ms when scaled to 4 GPUs distributed system.

The frontend of Forge ingests a user-defined computation logic for a linear attention variant. Its backend then intelligently maps this logic, along with potential communication operations, onto hardware accelerators and network interfaces, applying domain-specific optimizations to generate high-performance, distributed-aware kernels. Forge is built around three key principles: ❶ **A Linear-Attention-Specific Programming Abstraction.** We formulate our programming abstraction based on the canonical chunk-parallel representation of linear attention. This model decomposes the computation into three intuitive phases: a compute phase that processes local chunks of the sequence in parallel, an update phase that communicates and updates the inter-chunk states, and a merge phase that combines the global state with local chunk results. This abstraction aligns directly with the mathematical structure of chunk-wise parallel form, allowing researchers to translate their algorithm's formulation into our framework with minimal effort by providing simple PyTorch callable. ❷ **Native Compute-Communication Fusion.** At compilation time, Forge lowers the three user-provided callables into Triton code. We leverage Triton-Distributed (Zheng et al., 2025a)

as our compiler backend, which extends Triton with native communication primitives. This allows our compiler to generate fine-grained, tile-level communication instructions that are fused directly with computation. By creating custom communication patterns tailored to the algorithm, we bypass the overhead and limitations of standard libraries like NCCL, enabling a more efficient use of the underlying network fabric and dramatically improving hardware utilization. ❸ **Targeted Optimization of System Bottlenecks.** Beyond kernel fusion, we identify and optimize other critical system-level bottlenecks that affect real-world system performance. Forge employs a suite of techniques, including Ahead-of-Time (AOT) compilation with static kernel dispatcher built on top of Triton to reduce runtime overhead and an adaptive parallelism scheduler that dynamically explores the optimal configuration of compute resources, further boosting the end-to-end efficiency of linear attention execution.

To demonstrate its flexibility, we implement a broad range of linear attention variants using Forge, each requiring only dozens of lines of code. This programmability does not come at the cost of performance. On a single GPU, our generated kernels achieve 1.01x to 4.9x the performance of the state-of-the-art FLA library of expert-tuned kernels. Furthermore, in distributed settings, Forge demonstrates near-linear scalability on up to 128 GPUs, outperforming the leading open-source baseline by up to 7.2x.

2 PRELIMINARY

2.1 LINEAR ATTENTION ARCHITECTURE

Given a sequence $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_L]^\top \in \mathbb{R}^{L\times d}$, the input of attention block: $\mathbf{q}_i, \mathbf{k}_i, \mathbf{v}_i = \mathbf{W_q}\mathbf{x}_i, \mathbf{W_k}\mathbf{x}_i, \mathbf{W_v}\mathbf{x}_i$ where $\mathbf{x}_i, \mathbf{q}_i, \mathbf{k}_i, \mathbf{v}_i, \mathbf{y}_i \in \mathbb{R}^d$ and the weights $\mathbf{W_q}, \mathbf{W_k}, \mathbf{W_v} \in \mathbb{R}^{d\times d}$. Transformers employ softmax attention as a token mixer (Vaswani et al., 2017):

$$\mathbf{o}_i = \sum_{j=1}^{i} \frac{\exp(\mathbf{q}_i^\top \mathbf{k}_j)}{\sum_{p=1}^{i} \exp(\mathbf{q}_i^\top \mathbf{k}_p)} \mathbf{v}_j \qquad (1) \qquad\qquad \mathbf{O} = \text{softmax}(\mathbf{Q}\mathbf{K}^\top \odot \mathbf{M})\mathbf{V} \qquad (2)$$

Equation 2 is the matrix form of Equation 1 where $\mathbf{Q} := [\mathbf{q}_1, \ldots, \mathbf{q}_L]^\top$, $\mathbf{K} := [\mathbf{k}_1, \ldots, \mathbf{k}_L]^\top$, $\mathbf{V} := [\mathbf{v}_1, \ldots, \mathbf{v}_L]^\top \in \mathbb{R}^{L\times d}$ and $\mathbf{M} \in \{-\infty, 1\}^{L\times L}$ is a causal mask.

Such matrix form is well suited to modern accelerators, which excel at large matrix multiplications, but it incurs $\mathcal{O}(L^2 d)$ complexity. If we remove the softmax operation, the computation becomes associative: $\mathbf{o}_i = \mathbf{q}_i(\mathbf{k}_i \mathbf{v}_i^\top)$ which reduces the complexity to $O(Ld^2)$. The recurrence form is expressed as:

$$\mathbf{S}_t = \mathbf{S}_{t-1} + \mathbf{k}_t \mathbf{v}_t^\top, \quad \mathbf{o}_t = \mathbf{q}_t \mathbf{S}_t. \qquad (3)$$

Here $\mathbf{S} \in \mathbb{R}^{d\times d}$ is the state (or memory) updated in each time step. Equation 3 highlights the key idea of linear attention: replacing the exponential kernel in softmax attention with a linear recurrence. Although this formulation resembles RNNs (Hochreiter & Schmidhuber, 1997). The critical difference is that dependencies across time steps remain *linear*, which makes parallel training possible. Indeed, linear attention can be written in fully parallel form:

$$\mathbf{O} = (\mathbf{Q}\mathbf{K}^\top \odot \mathbf{M})\mathbf{V} \qquad (4)$$

Modern linear attention often augments the recurrence with a decay or gating mechanism, e.g., $\mathbf{S}_t = \mathbf{G}_t \odot \mathbf{S}_{t-1} + \mathbf{k}_t \mathbf{v}_t^\top$ (Gu & Dao, 2023), or with more sophisticated update rules such as the delta rule (Yang et al., 2024b): $\mathbf{S}_t = \mathbf{S}_{t-1}(\mathbf{I} - \beta_t \mathbf{k}_t \mathbf{k}_t^\top) + \beta_t \mathbf{v}_t \mathbf{k}_t^\top$, which enhance memory utilization.

2.2 CHUNK-WISE PARALLEL FORM OF LINEAR ATTENTIONS

The fully parallel form in Equation 4 achieves maximum hardware utilization but retains quadratic complexity. Conversely, the recurrent form in Equation 3 has linear complexity but is inherently sequential and hardware-inefficient. In practice, linear attention strikes a balance by adopting chunk-wise parallelization (Hua et al., 2022; Sun et al., 2023; Yang et al., 2023).

Specifically, the sequence of length L is partitioned into $\frac{L}{C}$ chunks of size C. Let $\mathbf{Q}_{[i]}, \mathbf{K}_{[i]}, \mathbf{V}_{[i]} \in \mathbb{R}^{C\times d}$ denote the query, key, and value matrices of the i-th chunk, and let $\mathbf{S}_{[i]} \in \mathbb{R}^{d\times d}$ be the state after processing chunk i. The chunk-wise formulation separates computation into intra-chunk and inter-chunk two parts, and then merge both together to get the final output as shown in Figure 2.

$$\mathbf{S}_{[i]} = \mathbf{S}_{[i-1]} + \underbrace{\sum_{j=(i-1)C}^{iC} \mathbf{k}_j^\top \mathbf{v}_j}_{\textbf{Matrix: } \mathbf{K}_{[i]}{}^\top \mathbf{V}_{[i]}} \tag{5}$$

$$\mathbf{O}_{[i]} = \underbrace{\mathbf{Q}_{[i]}\mathbf{S}_{[i-1]}}_{\text{inter}} + \underbrace{\left(\mathbf{Q}_{[i]}\mathbf{K}_{[i]}^\top \odot \mathbf{M}\right)\mathbf{V}_{[i]}}_{\text{intra}} \tag{6}$$

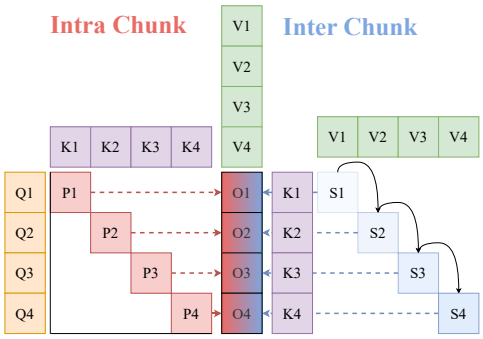


Figure 2: Chunk-wise parallel form demonstration for linear attention.

The inter chunk item can be viewed as readout memory from start of the sequence to the start of current chunk while the intra chunk item is processing the information in current chunk. When chunk size C is set to sequence length L, it becomes the fully parallel form as in Equation 4. The chunk size is a tradeoff between parallelism and FLOPs.

2.3 DSLs and Domain Specific Compiler

A domain-specific language (DSL) trades generality for performance by restricting program expressivity to a particular domain, creating opportunities for targeted optimization. Deep learning compilers like TVM (Chen et al., 2018), ThunderKitten (Spector et al., 2024), TileLang (Wang et al., 2025) and `torch.compile` (Ansel et al., 2024) exemplify this approach: by operating on a constrained set of primitives (i.e. tir in TVM and aten operators in PyTorch), they can systematically explore a focused design space to apply transformations like operator fusion and loop tiling, thereby automatically generating high-performance code from high-level descriptions.

This paradigm is particularly effective for operations that possess rich computational structure but high implementation complexity. By exposing domain-relevant abstractions, a DSL allows developers to specify *what* to compute, while delegating the complex details of *how* to execute it efficiently to the compiler. This separation of concerns is key to enabling automated, domain-specific optimizations without burdening the user with low-level hardware details.

Applied to the domain of linear attention, an effective DSL must therefore provide abstractions that are expressive enough to capture the diverse patterns found in various state update rules and parallel scan formulations. Simultaneously, its compiler must be able to recognize these common patterns and systematically generate optimized kernels for them, bridging the gap between high-level algorithmic design and performance, hardware-aware code. Because chunked parallel forms are complex to implement and offer opportunities to fully exploit hardware, this work focuses on the prefill phase of linear attention (used in inference or the forward pass of training). The backward pass can be implemented in a similar manner (Qin et al., a).

3 Forge

In this section, we propose a unified abstraction of diverse linear attention variants. This abstraction enables programmers to easily express linear attention semantics without worrying about implementation details and kernel performance.

3.1 Programming Abstraction

Despite the numerous linear attention variants designed by researchers, we unify these variants into three commonly shared phases based on chunk-wise parallel form introduced in subsection 2.2. ❶ Intra-Chunk Computation. The first phase computes a local state within each chunk of the input

Table 1: A comparison of representative linear attention variants that can be easily mapped to our three-phase abstraction, including HGRN (Qin et al., 2023), RetNet (Sun et al., 2023), Mamba2 (Dao & Gu, 2024), GLA (Yang et al., 2023), and GDN (Yang et al., 2024a). Despite diverse state types (vector vs. matrix) and decay mechanisms (element-wise product vs. matrix multiplication). $\boldsymbol{v}_t, \boldsymbol{k}_t, \boldsymbol{q}_t$ are value, key and query projections; $\boldsymbol{\alpha}_t, \beta_t, \boldsymbol{r}_t, \boldsymbol{i}_t$ are gates; $\odot$ is the Hadamard product.

Model	Update rule	Read-out	State + Decay type
HGRN	$\boldsymbol{h}_t = \boldsymbol{\alpha}_t \odot \boldsymbol{h}_{t-1} + (1 - \boldsymbol{\alpha}_t) \odot \boldsymbol{v}_t$	$\boldsymbol{o}_t = \boldsymbol{h}_t \odot \boldsymbol{q}_t$	vector + data-dependent vector
RetNet	$\boldsymbol{S}_t = \gamma\, \boldsymbol{S}_{t-1} + \boldsymbol{v}_t \boldsymbol{k}_t^\top$	$\boldsymbol{o}_t = \boldsymbol{S}_t \boldsymbol{q}_t$	matrix + data-independent scalar
Mamba2	$\boldsymbol{S}_t = \gamma_t\, \boldsymbol{S}_{t-1} + \boldsymbol{v}_t \boldsymbol{k}_t^\top$	$\boldsymbol{o}_t = \boldsymbol{S}_t \boldsymbol{q}_t$	matrix + data-dependent scalar
GLA	$\boldsymbol{S}_t = \boldsymbol{S}_{t-1} \odot (\mathbf{1}\boldsymbol{\alpha}_t^\top) + \boldsymbol{v}_t \boldsymbol{k}_t^\top$	$\boldsymbol{o}_t = \boldsymbol{S}_t \boldsymbol{q}_t$	matrix + data-dependent vector
GDN	$\boldsymbol{S}_t = \alpha_t\, \mathbf{S}_{t-1}(\mathbf{I} - \beta_t \boldsymbol{k}_t \boldsymbol{k}_t^\top) + \beta_t \boldsymbol{v}_t \boldsymbol{k}_t^\top$	$\boldsymbol{o}_t = \mathbf{S}_t \boldsymbol{q}_t$	matrix + data-dependent matrix

sequence. In this stage, computation across different chunks is embarrassingly parallel, as there are no data dependencies between them. Each chunk is processed independently, transforming its sequence of inputs into a relative state summary. ❷ Inter-Chunk State Propagation. The second phase addresses the dependencies between chunks. To compute the correct global state at the beginning of each chunk, the state summaries from all preceding chunks must be accumulated. For instance, in the case of vanilla linear attention, this propagation corresponds to a prefix sum (scan) operation, as shown in Equation 5. This phase is inherently sequential due to the temporal dependencies between chunk states. And cross-device communication happen in this phase. ❸ Merging and Output Generation. The final phase merges the results of the intra-chunk and inter-chunk computations. Here, operations are once again parallel across chunks. Each chunk utilizes the global state propagated from Phase 2 and its local inputs to compute its final output sequence.

Based on these insights, we designed our programming abstraction around three corresponding callable: `chunk_mode`, `decay_mode` and `merge_mode` correspond to three phases. This abstraction empowers users to implement a new linear attention variant by simply defining its chunk-wise parallel logic in idiomatic PyTorch code, decoupling algorithmic expression from system optimization. Furthermore, this three-phase decomposition also helps us optimize the program: we separate the parts of the entire program that can be executed in parallel and the parts that must be executed serially and may involve cross-device communication. With this information, Forge can perform more aggressive and accurate optimizations.

As shown in Table 1, prominent linear attention variants employ vastly different state representations and decay mechanisms. Nonetheless, all can be expressed using the chunk-wise parallel formulation. For example, the chunk-wise parallel form for Mamba2 can be specified as:

$$\boldsymbol{S}_{[t]} = (\prod \boldsymbol{\alpha}) \odot \boldsymbol{S}_{[t-1]} + \boldsymbol{V}\boldsymbol{K}^\top \qquad \boldsymbol{O} = \boldsymbol{Q}^\top \boldsymbol{K} \odot \boldsymbol{M} \odot \boldsymbol{G}\boldsymbol{V}^\top + \boldsymbol{S}\boldsymbol{Q} \tag{7}$$

Under our framework, a user only needs to implement these equations within our three-phase programming abstraction in native PyTorch code. Forge then automatically handles all subsequent code generation, performance tuning, and scaling to distributed systems.

3.2 Compilation and Code Generation

Given the chunk-wise parallel form description in our DSL, Forge performs a series of graph-level and system-level transformations to produce optimized program as illustrated in Figure 3. The user-defined function is first captured as Torch.fx graph (Reed et al., 2022) via tracing. We choose fx graph as our intermediate representation (IR) because it is the most powerful tool in the PyTorch ecosystem, most torch operators can be directly captured, which makes our DSL expressive enough to implement most of linear attention variants. We first replace special op code in fx graph as custom instructions (e.g. `placeholder` is replaced with load instruction). Domain-specific optimization passes are then applied. To enable fusion of non-trivial operators (e.g. `lower_triangular_inverse` in GDN), we provide a bunch of custom triton kernels commonly used in linear attention domain and mapping them to corresponding torch operators. Once the compiler see these operators, for instance, `torch.inverse`, they will be marked and subsequently substituted with custom triton source code in code generation phase. Common optimization passes like transpose elimination are also applied in this phase. The IR is further rewritten with system-resource awareness. Forge take the hardware information to generate hardware-specific in-

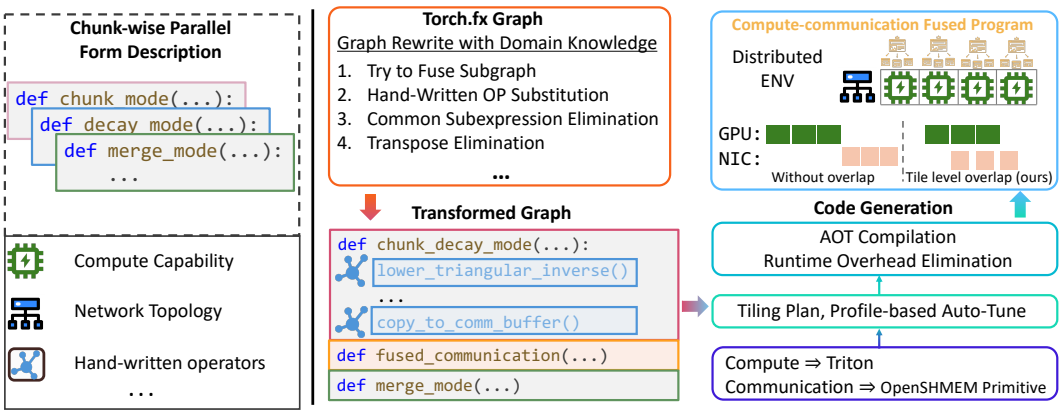


Figure 3: An overview of the Forge compilation pipeline. Our compiler ingests a high-level description of a linear attention algorithm, specified using our DSL (`chunk_mode`, `decay_mode`, `merge_mode`). This description is traced into a graph structure, which then undergoes a series of domain-specific optimization passes. The optimized graph is then compiled to Triton with native communication support, enabling fine-grained, tile-level overlap between computation and communication for better utilization of GPU and Network Interface Card (NIC).

structions. For instance, Tensor Memory Accelerator (TMA) availability is marked as an attribute of load instruction.

Recent studies have demonstrated that fine-grained compute–communication fusion can more effectively hide latency in distributed settings (Chang et al., 2024; Zheng et al., 2025b). We adopt this technique in Forge by automatically fusing computation and communication at the tile level, thereby reducing data dependency scope and eliminating the frequent GPU–host synchronizations inherent to traditional overlap strategies (Jangda et al., 2022). Within our programming abstraction, all cross-device communication is confined to the second phase, i.e., *inter-chunk state propagation*. Consequently, Forge first analyzes the data dependencies in this phase, then determines the corresponding computational tiling and communication tiling strategies based on the network topology, selects the appropriate communication mode, and generates computation and on-device communication instructions.

Finally, the IR is lowered into triton source code. Different with torch inductor (Ansel et al., 2024) targeting on official triton, Forge targets on Triton-Distributed (Zheng et al., 2025a), who addtionaly provide fine-grained communication control. On device computation is translated into computation primitives provided by Triton while the communication logic is mapped to the OpenShmem-style communication primitives provided exclusively by Triton-distributed, which are ultimately translated into GPU-initiated communication operations.

3.3 Performance Optimizations

With domain-specific knowledge of linear attention, Forge can explore a compact yet effective optimization space. In particular, the choice of whether to fuse different phases introduces an important trade-off. For example, fusing `chunk_mode` with `decay_mode` avoids materializing intermediate states in global memory, thereby reducing memory traffic. However, such fusion also limits available parallelism, since computations can no longer be scheduled independently at the chunk level. There are many such trade-offs and Forge will handle all of these to get better performance. Forge employs a parallelism scheduling algorithm that dynamically chooses an optimal parallelization strategy based on input shapes and hardware information internally. The specifics of this parallelism scheduler are detailed in Appendix B.

Beyond the optimization for target program, we further incorporate a set of optimizations tailored to the compilation system itself. In practice, the Triton runtime introduces overheads on the order of hundreds of microseconds, often exceeding the actual most of linear attention kernel execution time at short to medium sequence length (e.g. 2K or even 4K). While approaches like CUDA-Graph can mitigate launch overhead for workloads with static input shapes, they often incur significant memory costs and are unsuitable for the dynamic workloads in inference scenarios. To address these limitations natively, we extend Triton compiler with a custom Ahead-of-Time (AOT) compilation

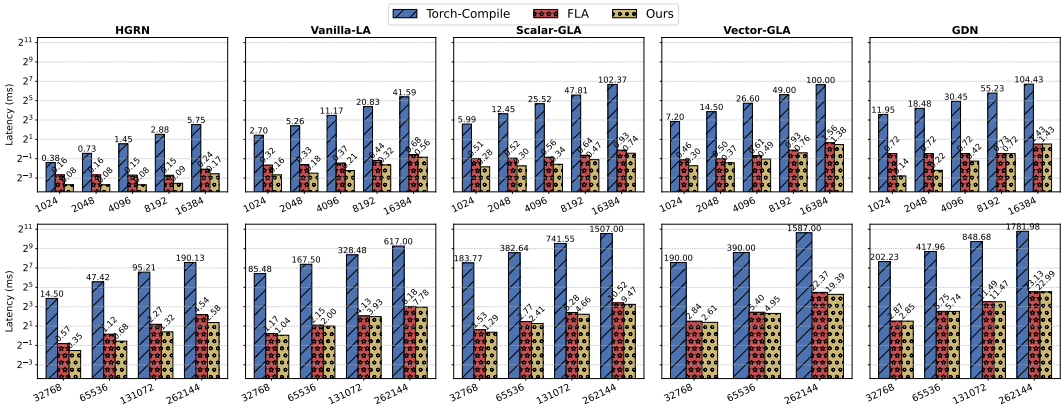


Figure 4: Latency comparison of different linear attention variants under varying sequence lengths on a single H100 GPU. Each subplot corresponds to one model, with the top row showing short to medium sequences (1K–16K) and the bottom row showing long sequences (32K–256K).

module. Specifically, our module compiles Triton source code into pre-linked dynamic libraries ahead of execution. At runtime, Forge employs a profile-guided static dispatcher that bypasses the Triton runtime entirely, invoking the optimal pre-compiled binary directly through the CUDA Driver API. The static dispatcher is automatically generated by Forge from an offline performance database, ensuring that the empirically best-performing kernel is selected for any given workload without incurring runtime overhead from hash lookups or dynamic compilation logic.

Another problem is redundant compilation. Since our system is specialized for linear attention, we observe that certain input tensor dimensions (e.g., head dimension and number of heads) remain relatively static across runs, while sequence length is typically dynamic. This property facilitates efficient AOT compilation: Forge allows users to specify constant dimensions and their admissible ranges via input metadata. Forge enumerates the Cartesian product of these ranges and generates all potentially required kernels in advance. Furthermore, by leveraging PyTorch's symbolic tracing, our system supports tensors with symbolic shapes, ensuring that recompilation is unnecessary unless static dim change.

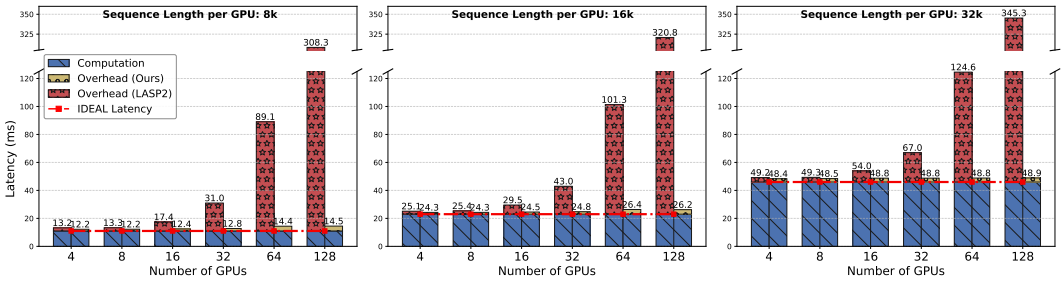
4 Experiments

We implement several high-performance linear attention kernels using Forge, including HGRN, vanilla linear attention, scalar GLA, vector GLA, and Gated DeltaNet (Qin et al., 2023; Dao & Gu, 2024; Yang et al., 2023; 2024a). While many linear attention models differ in their parameterization, the computational patterns we implement cover over ten existing model designs (Appendix C).

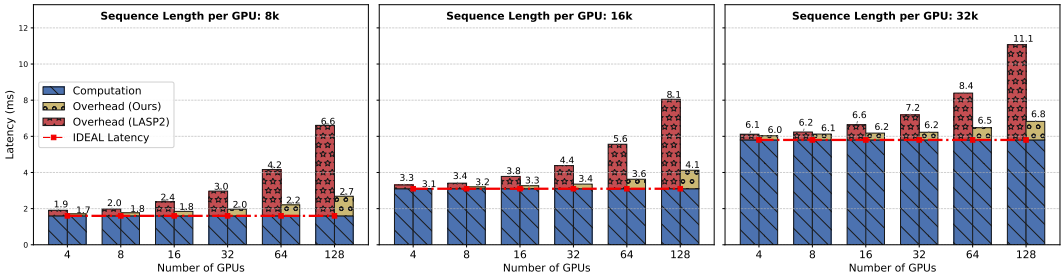
4.1 Single-Device Evaluation

Figure 4 reports the latency of kernels generated by Forge across sequence lengths ranging from 1K to 256K on a single H100 GPU. We compare with two baselines: Torch-Compile, representing a general-purpose compiler without domain-specific knowledge, and FLA (commit hash: `02766e71`), the state-of-the-art library of providing expert-tuned Triton kernels for linear attention. For all variants except HGRN, we fix $\mathrm{BatchSize} = 1$, $\mathrm{NumHeads} = 32$, and $\mathrm{HeadDim} = 128$, and vary sequence length. For HGRN, we follow its original single-head configuration. A batch size of one is a standard and reasonable choice, as a single long sequence in linear attention is computationally equivalent to a batch of packed shorter sequences.

Torch-Compile consistently exhibits poor performance, as it fails to apply the domain-specific fusion strategies required for linear attention. FLA achieves strong performance by carefully hand-tuning IO-aware tiling. Our work, Forge, automating this optimization process, consistently matches or outperforms FLA. On HGRN, Forge achieves 1.64–2.02× speedup, highlighting its ability to discover optimization opportunities beyond expert tuning (We check the generated code and find Forge use a more efficient threads allocation). On scalar and vector GLA as well as vanilla linear attention, Forge provides stable improvements (1.1–2.0×) while for GDN, performance converges to FLA at longer sequences. The speedup is most pronounced on short to medium sequence length. In this



(a) Gated Delta Network



(b) Gated (scalar) Linear Attention

Figure 5: Weak scaling performance comparison for GDN and GLA models. Both figures show latency breakdown across a varying number of GPUs with a fixed sequence length per GPU.

regime, system-level overheads and the choice of parallelism strategy are the dominant factors, and Forge effectively eliminate these bottlenecks, as discussed in subsection 3.3.

4.2 Sequence Parallel Evaluation

We further evaluate weak scaling behavior under distributed training, comparing against LASP2 (Sun et al., 2025), the strongest open-source baseline at the time of writing. ZeCO (Chou et al., 2025) reports improved scaling via pipelined communication, but its implementation is not public. We benchmark two representative workloads: GDN and scalar GLA, which feature matrix-multiplication and element-wise decay mechanisms, respectively. Experiments were conducted on a cluster of up to 128 NVIDIA H20 GPUs, interconnected with NVSwitch within nodes and InfiniBand between nodes. We fix $\mathrm{BatchSize} = 4$, $\mathrm{NumHeads} = 32$, $\mathrm{HeadDim} = 128$, and maintain a constant workload per GPU while scaling the total sequence length from 128K (on 4 GPUs) to 4 million tokens (on 128 GPUs). The ideal outcome for weak scaling is a constant execution time.

Figure 5 show that Forge exhibits near-ideal weak scaling for both workloads: latency remains flat as GPU count increases. This is attributable to two core features of our compiler. First, it generates communication patterns that avoid the data redundancy incurred by the All-Gather primitive used in LASP2. Second, its ability to fuse computation and communication effectively hides the latency of the local state update. This is particularly impactful for GDN, whose matrix-based update is time consuming. In contrast, the communication and computation redundancy of LASP2 is amplified as the number of nodes increases, causing its performance to degrade significantly (e.g., from 49.2ms on 4 GPUs to 345ms on 128 GPUs for GDN). This confirms that Forge eliminates redundant communication and achieves scalable performance on large GPU clusters.

4.3 Ablation Study

AOT compilation with static dispatcher. We measure the end-to-end latency of the execution of Scalar GLA kernel including both kernel execution and to demonstrate Triton runtime overhead and the efficiency of our solution as shown in Figure 6. At a sequence length of 1024, this overhead (207µs) is over 4.4 times the actual kernel execution time (47µs). Our static dispatcher mitigates this issue by reducing the overhead by 46%, yielding a 1.6x end-to-end speedup on these latency-sensitive inputs. As the sequence length grows and execution time be-

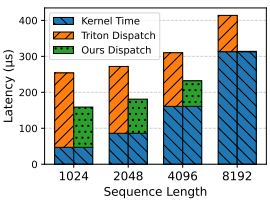


Figure 6: Execution time decomposition.

comes the dominant factor, our dispatcher consistently maintains a negligible overhead that is effectively eliminated at 8192 tokens (reducing from 101µs to just 1µs).

Tile Level Compute-Communication Overlapping. We conduct a targeted study to isolate and quantify the benefits of our tile-level compute-communication fusion. We set two baselines for the inter-rank state propagation: ❶ Serial communication, where each rank i waits to receive all state data from rank $i-1$ before performing its local update and sending to rank $i+1$. ❷ Pipelined baseline, which chunks the state and uses standard NCCL send/recv operations to overlap the computation of one chunk with the communication of the next. Our method, in contrast, generates a single kernel that fuses the state update computation with communication primitives at the tile level. The results, presented in Table 2 show the standard Pipelined (PyTorch) baseline is slightly slower than the naive Serial implementation, demonstrating that host-managed pipelining can be counterproductive due to the significant overhead of launching numerous small operations and the required host-device synchronization. Conversely, our fused kernel substantially reduces the total time, achieving a 1.56x speedup over the serial baseline.

Table 2: State communication time on 8xH800 GPUs. State size is 67MB.

Method	Time (us)
Serial	873
Torch-Pipeline	902
Ours	560

5 Related Work

There are a wide range of approaches to address the quadratic complexity of softmax attention. Sparse attention mechanisms (Zaheer et al., 2020; Xiao et al., 2023; Yuan et al., 2025) leverage structured or un-structured sparsity in attention to skip computation. Quantized attention (Shah et al., 2024; Zhang et al., 2024) use exploit low-precision arithmetic unit in modern hardware to get higher throughput. There are also various techniques to reduce key–value (KV) cache overhead. KIVI (Liu et al., 2024b) and SKVQ (Duanmu et al., 2024) directly compress KV cache using quantization while grouped-query attention (GQA) (Ainslie et al., 2023), and multi-head latent attention (MLA) (Liu et al., 2024a) alter the attention architecture to reduce memory overhead.

Another line of work proposes architectural alternatives with lower complexity, including linear attention variants (Katharopoulos et al., 2020; Dao & Gu, 2024; Peng et al., 2023; Yang et al., 2024a) as well as test-time-training approaches (Sun et al., 2024b; Behrouz et al., 2024). For linear attention sequence parallelism, LASP (Sun et al., 2024a) first extend linear attention to distributed environments with serial send-receive primitive. LASP2 (Sun et al., 2025) improves on this by leveraging collective communication primitives, yet both approaches still incur significant bandwidth under-utilization. ZeCO (Chou et al., 2025) introduces a pipelined send–receive scheme to hide send/receive latency but relies on manual chunk-size tuning and does not detail its implementation. Therefore, it was not included in our comparison.

AI compilers have been developed to optimize a broad range of workloads. `torch.compile` (Ansel et al., 2024), TVM (Chen et al., 2018), and TASO (Jia et al., 2019) are effective for common operators but their optimization spaces do not cover linear attention. Operator-level compilers such as Triton (Tillet & Cox, 2019), ThunderKitten Spector et al. (2024), TileLang (Wang et al., 2025), and Triton-Distributed (Zheng et al., 2025a) provide expressive abstractions for modern accelerators, but supporting the large and growing family of linear attention variants still requires substantial manual development effort. The most closely related work, FlexAttention (Dong et al., 2024), targets block-sparse softmax attention and is limited to single-device settings.

6 Conclusion

We presented Forge, a domain-specific compiler for linear attention that unifies diverse algorithmic variants under a common three-phase abstraction. By generating hardware-efficient kernels with native distributed execution support, Forge bridges the gap between rapidly evolving linear attention research and the complexity of hand-tuned implementations. Our evaluation demonstrates that Forge

achieves both high performance and broad applicability across modern linear attention models. We hope this work will accelerate the development of new architectures and inspire further research at the intersection of deep learning algorithms and domain-specific compilation.

7 REPRODUCIBILITY STATEMENT

We are committed to the reproducibility of our work. Most of algorithms, implementation details, and experimental setups are described in the paper. Due to organizational policies requiring an internal review prior to public release, the source code is not included with the submission. However, we are committed to open-sourcing the code and will provide it to reviewers upon request for the purpose of evaluation.

REFERENCES

Joshua Ainslie, James Lee-Thorp, Michiel De Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245*, 2023.

Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 929–947, 2024.

Ali Behrouz, Peilin Zhong, and Vahab Mirrokni. Titans: Learning to memorize at test time. *arXiv preprint arXiv:2501.00663*, 2024.

Li-Wen Chang, Wenlei Bao, Qi Hou, Chengquan Jiang, Ningxin Zheng, Yinmin Zhong, Xuanrun Zhang, Zuquan Song, Chengji Yao, Ziheng Jiang, et al. Flux: Fast software-based communication overlap on gpus through kernel fusion. *arXiv preprint arXiv:2406.06858*, 2024.

Feiyang Chen, Yu Cheng, Lei Wang, Yuqing Xia, Ziming Miao, Lingxiao Ma, Fan Yang, Jilong Xue, Zhi Yang, Mao Yang, et al. Attentionengine: A versatile framework for efficient attention mechanisms on diverse hardware platforms. *arXiv preprint arXiv:2502.15349*, 2025.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 578–594, 2018.

Yuhong Chou, Zehao Liu, Ruijie Zhu, Xinyi Wan, Tianjian Li, Congying Chu, Qian Liu, Jibin Wu, and Zejun Ma. Zeco: Zero communication overhead sequence parallelism for linear attention. *arXiv preprint arXiv:2507.01004*, 2025.

Tri Dao and Albert Gu. Transformers are SSMs: Generalized models and efficient algorithms through structured state space duality. In *International Conference on Machine Learning (ICML)*, 2024.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

Soham De, Samuel L Smith, Anushan Fernando, Aleksandar Botev, George Cristian-Muraru, Albert Gu, Ruba Haroun, Leonard Berrada, Yutian Chen, Srivatsan Srinivasan, et al. Griffin: Mixing gated linear recurrences with local attention for efficient language models. *arXiv preprint arXiv:2402.19427*, 2024.

Juechu Dong, Boyuan Feng, Driss Guessous, Yanbo Liang, and Horace He. Flex attention: A programming model for generating optimized attention kernels. *arXiv preprint arXiv:2412.05496*, 2024.

Haojie Duanmu, Zhihang Yuan, Xiuhong Li, Jiangfei Duan, Xingcheng Zhang, and Dahua Lin. Skvq: Sliding-window key and value cache quantization for large language models. *arXiv preprint arXiv:2405.06219*, 2024.

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

William Hu, Drew Wadsworth, Sean Siddens, Stanley Winata, Daniel Y. Fu, Ryann Swann, Muhammad Osama, Christopher Ré, and Simran Arora. Hipkittens: Fast and furious amd kernels, 2025. URL https://arxiv.org/abs/2511.08083.

Weizhe Hua, Zihang Dai, Hanxiao Liu, and Quoc Le. Transformer quality in linear time. In *International conference on machine learning*, pp. 9099–9117. PMLR, 2022.

Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 402–416, 2022.

Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 47–62, 2019.

Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In *International conference on machine learning*, pp. 5156–5165. PMLR, 2020.

Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, et al. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*, 2024a.

Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context. *arXiv preprint arXiv:2310.01889*, 2023.

Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. *arXiv preprint arXiv:2402.02750*, 2024b.

Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Chung, Matteo Grella, et al. Rwkv: Reinventing rnns for the transformer era. *arXiv preprint arXiv:2305.13048*, 2023.

Zhen Qin, Xuyang Shen, Dong Li, and Yiran Zhong. Accelerating linear attention design by unifying forward & backward propagation. In *ES-FoMo III: 3rd Workshop on Efficient Systems for Foundation Models*, a.

Zhen Qin, Songlin Yang, Weixuan Sun, Xuyang Shen, Dong Li, Weigao Sun, and Yiran Zhong. Hgrn2: Gated linear rnns with state expansion. In *First Conference on Language Modeling*, b.

Zhen Qin, Songlin Yang, and Yiran Zhong. Hierarchically gated recurrent neural network for sequence modeling. *Advances in Neural Information Processing Systems*, 36:33202–33221, 2023.

Zhen Qin, Weigao Sun, Dong Li, Xuyang Shen, Weixuan Sun, and Yiran Zhong. Lightning attention-2: A free lunch for handling unlimited sequence lengths in large language models. *arXiv preprint arXiv:2401.04658*, 2024.

James Reed, Zachary DeVito, Horace He, Ansley Ussery, and Jason Ansel. torch. fx: Practical program capture and transformation for deep learning in python. *Proceedings of Machine Learning and Systems*, 4:638–651, 2022.

Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *Advances in Neural Information Processing Systems*, 37:68658–68685, 2024.

Julien Siems, Timur Carstensen, Arber Zela, Frank Hutter, Massimiliano Pontil, and Riccardo Grazzi. Deltaproduct: Improving state-tracking in linear rnns via householder products. *arXiv preprint arXiv:2502.10297*, 2025.

Benjamin F Spector, Simran Arora, Aaryan Singhal, Daniel Y Fu, and Christopher Ré. Thunderkittens: Simple, fast, and adorable ai kernels. *arXiv preprint arXiv:2410.20399*, 2024.

Weigao Sun, Zhen Qin, Dong Li, Xuyang Shen, Yu Qiao, and Yiran Zhong. Linear attention sequence parallelism. *arXiv preprint arXiv:2404.02882*, 2024a.

Weigao Sun, Disen Lan, Yiran Zhong, Xiaoye Qu, and Yu Cheng. Lasp-2: Rethinking sequence parallelism for linear attention and its hybrid. *arXiv preprint arXiv:2502.07563*, 2025.

Yu Sun, Xinhao Li, Karan Dalal, Jiarui Xu, Arjun Vikram, Genghan Zhang, Yann Dubois, Xinlei Chen, Xiaolong Wang, Sanmi Koyejo, et al. Learning to (learn at test time): Rnns with expressive hidden states. *arXiv preprint arXiv:2407.04620*, 2024b.

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

Philippe Tillet and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pp. 10–19, 2019.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Lei Wang, Yu Cheng, Yining Shi, Zhengju Tang, Zhiwen Mo, Wenhao Xie, Lingxiao Ma, Yuqing Xia, Jilong Xue, Fan Yang, et al. Tilelang: A composable tiled programming model for ai systems. *arXiv preprint arXiv:2504.17577*, 2025.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. *arXiv preprint arXiv:2309.17453*, 2023.

Songlin Yang and Yu Zhang. Fla: A triton-based library for hardware-efficient implementations of linear attention mechanism, January 2024. URL `https://github.com/fla-org/flash-linear-attention`.

Songlin Yang, Bailin Wang, Yikang Shen, Rameswar Panda, and Yoon Kim. Gated linear attention transformers with hardware-efficient training. *arXiv preprint arXiv:2312.06635*, 2023.

Songlin Yang, Jan Kautz, and Ali Hatamizadeh. Gated delta networks: Improving mamba2 with delta rule. *arXiv preprint arXiv:2412.06464*, 2024a.

Songlin Yang, Bailin Wang, Yu Zhang, Yikang Shen, and Yoon Kim. Parallelizing linear transformers with the delta rule over sequence length. *Advances in neural information processing systems*, 37:115491–115522, 2024b.

Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, YX Wei, Lean Wang, Zhiping Xiao, et al. Native sparse attention: Hardware-aligned and natively trainable sparse attention. *arXiv preprint arXiv:2502.11089*, 2025.

Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. Big bird: Transformers for longer sequences. *Advances in neural information processing systems*, 33:17283–17297, 2020.

Jintao Zhang, Jia Wei, Haofeng Huang, Pengle Zhang, Jun Zhu, and Jianfei Chen. Sageattention: Accurate 8-bit attention for plug-and-play inference acceleration. *arXiv preprint arXiv:2410.02367*, 2024.

Size Zheng, Wenlei Bao, Qi Hou, Xuegui Zheng, Jin Fang, Chenhui Huang, Tianqi Li, Haojie Duanmu, Renze Chen, Ruifan Xu, et al. Triton-distributed: Programming overlapping kernels on distributed ai systems with the triton compiler. *arXiv preprint arXiv:2504.19442*, 2025a.

Size Zheng, Jin Fang, Xuegui Zheng, Qi Hou, Wenlei Bao, Ningxin Zheng, Ziheng Jiang, Dongyang Wang, Jianxi Ye, Haibin Lin, et al. Tilelink: Generating efficient compute-communication overlapping kernels using tile-centric primitives. *arXiv preprint arXiv:2503.20313*, 2025b.

A USAGE OF LLMS

For the preparation of this manuscript, we utilized LLM, as writing assistant to enhance the quality of the prose. Our process was interactive: the authors provided initial drafts and specific sentences to the LLM and used its suggestions to refine the text. Majority of prompts aimed at refining sentences to improve conciseness, in addition to correcting grammar and improving overall clarity. The core scientific ideas, methodology, and experimental results were developed exclusively by the human authors, who bear full responsibility for all claims and content within this paper.

B PARALLISIM SCHEDULER

Forge automatically explore different parallel schemes for linear attention during compilation. With the domain knowledge of linear attention, we can shrink the search space into two mainly used schemes, which is partially explored in FLA (Yang & Zhang, 2024).

In this section, we provide a theoretical analysis of two parallel execution strategies for linear attention using vanilla linear attention on GPU as an example. Then we derive a heuristic scheduling algorithm 1 that Forge used to build a parallelism scheduler, selecting the optimal strategy based on the input tensor shapes and hardware characteristics.

B.1 NOTATION

The vanilla linear attention is defined by the recurrence:

$$\mathbf{S}_t = \mathbf{S}_{t-1} + \mathbf{k}_t \mathbf{v}_t^T \tag{8}$$

$$\mathbf{o}_t = \mathbf{q}_t \mathbf{S}_t \tag{9}$$

where $\mathbf{k}_t, \mathbf{q}_t \in \mathbb{R}^{D_k}$ and $\mathbf{v}_t \in \mathbb{R}^{D_v}$ are the key, query, and value vectors at timestep t, and $\mathbf{S}_t \in \mathbb{R}^{D_k \times D_v}$ is the state matrix. We consider batched inputs $\mathbf{Q}, \mathbf{K} \in \mathbb{R}^{B \times T \times H \times D_k}$ and $\mathbf{V} \in \mathbb{R}^{B \times T \times H \times D_v}$. The sequence of length T is divided into N_C chunks of size C, such that $T = N_C \times C$. Let N_{SM} be the number of Streaming Multiprocessors (SMs) on the target GPU, representing its parallel execution capacity.

B.2 STRATEGY 1: DECOUPLED THREE-PHASE EXECUTION

This strategy directly maps the chunk-wise parallel algorithm onto our three-phase programming model. Each phase is a separate kernel launch.

Parallelism Analysis. The degree of parallelism varies significantly between phases.

- **Phase 1 (Intra-Chunk State Computation):** Computations for each chunk are independent. The total number of parallel tasks is $B \times H \times N_C$. This phase exhibits the highest degree of parallelism.
- **Phase 2 (Inter-Chunk State Propagation):** The prefix sum (scan) is sequential across the N_C dimension. Parallelism is limited to the batch (B) and head (H) dimensions. Further parallelism, which we denote as P_{state}, can be achieved by partitioning the state matrix $\mathbf{S} \in \mathbb{R}^{D_k \times D_v}$ and processing its partitions on different thread blocks. The total number of parallel tasks is $B \times H \times P_{state}$.
- **Phase 3 (Merge):** Similar to Phase 1, the merge operation is independent across chunks, offering a high degree of parallelism with $B \times H \times N_C$ tasks.

Memory Access Analysis. The defining characteristic of this strategy is the materialization of intermediate states in global memory (GMEM) between phases.

- **Phase 1:** Reads $\mathbf{K}$ and $\mathbf{V}$ from GMEM. Writes the intermediate, chunk-local states $\mathbf{S}' \in \mathbb{R}^{B \times H \times N_C \times D_k \times D_v}$ back to GMEM.

$$\text{GMEM Traffic}_{P1} = \underbrace{BHT(D_k + D_v)}_{\text{Reads}} + \underbrace{BHN_C D_k D_v}_{\text{Writes}} \tag{10}$$

- **Phase 2:** Reads the $B \times H \times N_C$ chunk-local states from GMEM, performs the scan, and writes the updated global states $\mathbf{S}_{global} \in \mathbb{R}^{B \times H \times N_C \times D_k \times D_v}$ back to GMEM.

$$\text{GMEM Traffic}_{P2} = \underbrace{BHN_C D_k D_v}_{\text{Reads}} + \underbrace{BHN_C D_k D_v}_{\text{Writes}} = 2BHN_C D_k D_v \tag{11}$$

The total GMEM traffic introduced between Phase 1 and Phase 2 is $3 \times B \times H \times N_C \times D_k D_v$.

B.3 Strategy 2: Fused Intra- and Inter-Chunk Execution

This strategy fuses Phase 1 and Phase 2, computing the local state of a chunk and immediately uses it to update the global state, all within on-chip memory.

Parallelism Analysis. By fusing the phases, the execution is constrained by the most sequential part, which is the inter-chunk scan. Therefore, the maximum number of parallel tasks is identical to that of Phase 2 in the decoupled strategy: $B \times H \times P_{state}$. The high parallelism across the chunk dimension (N_C) is sacrificed.

Memory Access Analysis. The primary advantage of this strategy is the elimination of the intermediate GMEM traffic.

- **Fused Phase (1+2):** Each of the $B \times H \times P_{state}$ parallel tasks loads its corresponding slice of $\mathbf{K}$ and $\mathbf{V}$ from GMEM chunk by chunk. The intermediate, chunk-local states are generated and accumulated entirely within SMEM. The only state written to GMEM is the final updated global state for each chunk, which is required by the Merge phase.

$$\text{GMEM Traffic}_{P1+P2} = \underbrace{BHT(D_k + D_v)}_{\text{Reads}} + \underbrace{BHN_C D_k D_v}_{\text{Writes}} \tag{12}$$

Compared to Strategy 1, this approach saves $2 \times B \times H \times N_C \times D_k D_v$ worth of GMEM traffic, which is the cost of one full read and one full write of the intermediate state tensor.

B.4 Heuristic Scheduling Algorithm

The choice between the Decoupled and Fused strategies presents a classic trade-off between parallelism and memory locality.

- **Strategy 1 (Decoupled)** is favored when the GPU has a high degree of parallelism (N_{SM} is large) that is not saturated by the task parallelism of Strategy 2 ($B \times H \times P_{state}$). The performance gain from launching more parallel tasks in Phase 1 and 3 must outweigh the latency incurred by the extra GMEM I/O.
- **Strategy 2 (Fused)** is favored when the task parallelism of the scan ($B \times H \times P_{state}$) is already sufficient to saturate the GPU's SMs, or when the cost of reading/writing the intermediate states is the dominant performance bottleneck.

We can formulate a simple heuristic to guide this choice as shown in Algo 1.

In practice, we set $T_{\text{comp\_chunk}}$ to 0 to reduce the runtime overhead and omit the micro-benchmark effort. The insight is that most linear attention variants is memory bound instead of compute-intensive.

C Computational Patterns and Model Coverage

This appendix elaborates on the claim that our implemented kernels for a few representative models can support a much broader range of linear attention variants. The mapping from our representative implementations to the computational patterns and the models they cover is detailed below and summarized in Table 3.

Algorithm 1 Parallelism Scheduler)

Require: Shapes B, T, H, d_k, d_v, chunk size C, tile sizes t_k, t_v
Require: Device params: memory bandwidth BW (bytes/s), $P_{\max}$ (hardware concurrency cap)
Require: Cost param: $T_{\text{comp\_chunk}}$ (measured per-chunk compute time in seconds)
Require: Thresholds: $P_{\min}$ (saturation threshold, tasks), element size s (bytes)
Ensure: Return strategy $\in \{\text{FUSED}, \text{DECOUPLED}\}$

$N_c \leftarrow \lceil T/C \rceil$
$P_{\text{state}} \leftarrow \lceil d_k/t_k \rceil \cdot \lceil d_v/t_v \rceil$
$P_{\text{dec}} \leftarrow B \cdot H \cdot N_c \cdot P_{\text{state}}$
$P_{\text{fused}} \leftarrow B \cdot H \cdot P_{\text{state}}$
$\text{GMEM}_{\text{dec}} \leftarrow \big(2C(d_k + d_v) + 5d_k d_v\big) \cdot s$
$\text{GMEM}_{\text{fused}} \leftarrow \big(2C(d_k + d_v) + 4d_k d_v\big) \cdot s$
▷ Fast path: if fused already provides enough parallelism, prefer fused to save GMEM IO
if $P_{\text{fused}} \geq P_{\min}$ **then**
  return FUSED
end if
▷ Otherwise estimate per-chunk runtime under each mapping
$\tilde{P}_{\text{dec}} \leftarrow \min(P_{\text{dec}}, P_{\max})$
$\tilde{P}_{\text{fused}} \leftarrow \min(P_{\text{fused}}, P_{\max})$
$\hat{T}_{\text{dec}} \leftarrow \dfrac{\text{GMEM}_{\text{dec}}}{BW \cdot \tilde{P}_{\text{dec}}} + \dfrac{T_{\text{comp\_chunk}}}{\tilde{P}_{\text{dec}}}$
$\hat{T}_{\text{fused}} \leftarrow \dfrac{\text{GMEM}_{\text{fused}}}{BW \cdot \tilde{P}_{\text{fused}}} + \dfrac{T_{\text{comp\_chunk}}}{\tilde{P}_{\text{fused}}}$
if $\hat{T}_{\text{fused}} \leq \hat{T}_{\text{dec}}$ **then**
  return FUSED
else
  return DECOUPLED
end if

Vector-State Linear RNNs (represented by HGRN) Our HGRN kernel embodies the computational pattern of linear recurrent networks where the state is a vector, updated via element-wise operations with gated inputs. This is a common pattern for models aiming for high efficiency with a compact state. Models sharing this fundamental structure include the original HGRN (Qin et al., 2023) and Hawk (RG-LRU) (De et al., 2024).

Matrix-State with Scalar Decay (represented by scalar-GLA) The scalar-GLA kernel represents the widely adopted matrix-state linear attention pattern. In this formulation, the state is a matrix updated via an outer product of key and value vectors, combined with a simple data-dependent or data-independent scalar decay. This pattern is foundational to many prominent and powerful models such as RetNet (Sun et al., 2023), Mamba-2 (Dao & Gu, 2024), and Lightning Attention (Qin et al., 2024).

Matrix-State with Vector Decay (represented by vector-GLA) Our vector-GLA implementation captures the pattern of matrix-state models that employ a more expressive, data-dependent vector decay. This allows for per-feature state transition dynamics, offering a richer representation than a single scalar decay. Models in this category include the original Gated Linear Attention (GLA) (Yang et al., 2023), HGRN-2 (Qin et al., b), and RWKV-6 (Peng et al., 2023).

Delta-Rule Updates (represented by GDN) Finally, our Gated DeltaNet (GDN) kernel is representative of a family of models based on the delta rule for state updates. This update mechanism can be interpreted as applying a series of Householder transformations to the state matrix, enabling more complex state transitions. This pattern is central to the family of DeltaNets, including the original DeltaNet (Yang et al., 2024b), GatedDeltaNet (Yang et al., 2024a), and DeltaProduct (Siems et al., 2025).

Table 3: Mapping of representative kernels implemented in subsection 4.1 to the broader set of models they cover. This demonstrates the generality of our compiler.

Representative Kernel	Covered Models (Examples)
HGRN	HGRN (Qin et al., 2023), Hawk (RG-LRU) (De et al., 2024)
Scalar-GLA	RetNet (Sun et al., 2023), Mamba-2 (Dao & Gu, 2024), Lightning Attention (Qin et al., 2024)
Vector-GLA	GLA (Yang et al., 2023), HGRN-2 (Qin et al., b), RWKV-6 (Peng et al., 2023)
GDN	DeltaNet (Yang et al., 2024b), GatedDeltaNet (Yang et al., 2024a), DeltaProduct (Siems et al., 2025)

D Discussion on Extensibility and Sustainability

The longevity and utility of a Domain-Specific Language (DSL) depend heavily on whether its abstraction captures the invariant properties of the target domain rather than transient heuristics. In this section, we discuss the future-proofing of Forge from two perspectives: algorithmic expressiveness and hardware sustainability.

D.1 Algorithmic Expressiveness: Grounded in Associativity

The universality of Forge's three-phase abstraction (Intra-Chunk, Inter-Chunk, and Merge) derives from the mathematical foundation of efficient sequence modeling: the **associative property**.

Mathematical Invariance. Virtually all Linear Attention and State Space Duality (SSD) models aim to achieve $O(N)$ computational complexity by formulating the attention mechanism as a recurrence or a parallel prefix scan. Mathematically, any algorithm that can be decomposed into a chunk parallel form fits the Forge abstraction. This is not an ad-hoc design choice but a direct mapping of the underlying algebraic structure required for parallelization.

Handling Complex Dependencies. Forge is expressive enough to handle complex state updates found in modern architectures, provided they satisfy chunk-wise associativity. For instance:

- **The Delta Rule:** Despite involving data-dependent updates (e.g., $h_t = h_{t-1} + \beta_t(v_t - h_{t-1})$), the Delta Rule preserves the associative structure within local chunks, allowing Forge to effectively fuse operations.
- **Element-wise Decay:** Varying decay rates (as seen in Vector-Gated GLA) are fully supported, as the element-wise multiplication distributes over addition, maintaining the scan property.

Theoretical Limitations and Edge Cases. The boundary of Forge's applicability is strictly defined by **non-associative recurrences**. If an architecture introduces a dependency where the state update $h_t = f(h_{t-1}, x_t)$ involves a non-linear function f that prevents parallel(e.g., passing the hidden state through a complex MLP at every step before the next update), it cannot be expressed in Forge.

A notable example is the Test-Time Training (TTT) layer (Sun et al., 2024b). While TTT can be viewed through the lens of linear attention or fast weights, its gradient-based updates involve nonlinearities that break associativity. Consequently, TTT cannot be accelerated via the parallel prefix scans used in Forge. However, it is worth noting that this limitation is mutual: by abandoning associativity, such models fundamentally forego the massively parallel efficiency that characterizes the linear attention regime targeted by Forge.

D.2 Hardware Sustainability: Hierarchical Decoupling

To ensure sustainability amidst rapid hardware evolution, Forge employs a strict hierarchical decoupling between the algorithmic description and hardware-specific instructions.

Layered Compilation. Forge operates as a high-level graph compiler rather than a low-level assembler. It does not directly emit hardware-specific assembly (ISA). Instead, it lowers the PyTorch-based algorithmic description into an intermediate kernel DSL. In our current implementation, we target `Triton`, effectively delegating low-level complexities such as register allocation, instruction scheduling, and tensor core management (e.g., `WGMMA` on Hopper) to the Triton compiler.

Backend Agnosticism. This layered design makes Forge inherently adaptable. While the current backend is Triton, the architecture allows for retargeting the mapping layer to other emerging DSLs (e.g., ThunderKitten (Spector et al., 2024) or TileLang (Wang et al., 2025)). These DSLs provide better performance with the cost of hardware overfit (e.g. ThunderKittens target on NVidia GPU, HipKittens (Hu et al., 2025) target solely on AMD GPU while Triton support multiple hardware backend). Triton allows us to minimize the use of inline assembly (e.g. PTX on NVGPU), reserving it only for specific extensions not yet exposed by the intermediate representation.

Forward Compatibility. Consequently, as hardware architectures evolve (e.g., the transition to NVIDIA Blackwell), Forge benefits automatically from updates made by the community to the intermediate compiler stack. This ensures that user-level code remains stable and performant without requiring modification, solving the maintenance bottleneck often associated with hand-written kernels.

E Discussion on Generalizability to Softmax Attention

While Forge is explicitly designed for Linear Attention, the underlying mathematical associativity, which enables our optimization is shared by Softmax attention (exploited via the online softmax trick (Dao et al., 2022)). In this section, we analyze the theoretical feasibility of extending Forge to support Softmax attention, and the design rationale behind our decision to specialize in the Linear Attention domain.

Extending the Forge abstraction to support Softmax-based mechanisms is theoretically feasible but would necessitate two primary modifications to the current three-phase abstraction:

- **Generalizing the Inter-Chunk State.** In Linear Attention, the state propagated between chunks is typically a feature map (e.g., $S \in \mathbb{R}^{d \times d}$) governed by a linear recurrence. In contrast, Softmax attention requires the propagation of normalization statistics, specifically the running maximum m and the running sum l to stabilize the computation of exponentials. Extending Forge would require modifying the `decay_mode` phase to support the propagation of these scalar or vector statistics alongside, or instead of, the matrix state.
- **Introducing a Rescaling Primitive.** The `merge_mode` in Forge is currently designed for linear combinations (accumulation). Softmax attention, however, mandates a renormalization step when merging partial results. Specifically, the output of a preceding block O_1 must be scaled down based on the difference between its local maximum m_1 and the new global maximum m_{new}:

$$O_{new} = O_1 \times e^{m_1 - m_{new}} + O_2 \times e^{m_2 - m_{new}} \tag{13}$$

Supporting this would require introducing a native `rescale(output, old_stats, new_stats)` primitive into the Forge DSL.Ï

Despite the feasibility of these extensions, we deliberately narrowed the scope of Forge to Linear Attention to maximize expressiveness.

The Tension between Universality and Specificity. Constructing a "universal" DSL often necessitates a rigid abstraction that compromises the ability to model complex, domain-specific patterns. A recent framework, AttentionEngine (Chen et al., 2025), attempts to unify both Softmax and Linear Attention. However, to maintain this broad generality, its abstraction struggles to express advanced Linear Attention variants that feature intricate dependencies, such as the **Delta Rule** or **Vector-Gated GLA**. By specializing in Linear Attention, Forge avoids these constraints, supporting these complex patterns effortlessly.

From an ecosystem perspective, Softmax attention is already well-served by highly optimized libraries and compilers (e.g., FlexAttention (Dong et al., 2024)). In contrast, the Linear Attention landscape is characterized by rapid algorithmic fragmentation, where new variants are proposed frequently but lack efficient kernel support. Forge addresses this specific "N-to-1" compiler challenge, solving a critical bottleneck that is currently more pressing for the research community than further optimizing Softmax kernels.

Finally, while the abstraction of Forge is specialized, the *system-level optimizations* are broadly applicable. For example, our Ahead-of-Time (AOT) compilation pipeline and static dispatcher, designed to eliminate runtime Python overheads, are universal optimizations. These techniques can be directly adopted by Softmax-focused DSLs (such as FlexAttention) to significantly improve performance, particularly in short-sequence regimes where dispatch latency is a dominant factor.

F Experiments: Distributed Evaluation on H100 Cluster

To verify the robustness of our distributed scaling strategy, we conducted additional experiments on an **8x H100 node** connected via NVLink.

Table 4 presents the latency comparison between Ring-Attention (a standard distributed softmax attention baseline) (Liu et al., 2023) with FlashAttention-3 which exploit advanced hardware feature on Hopper GPU, LASP-2, and Forge (both LASP2 and Forge using the Scalar GLA variant). The experiments were conducted with a **batch size of 1**, $\mathrm{NumHeads} = 32$, and $\mathrm{HeadDim} = 128$. The **SeqLen** in table means sequence length per GPU(global sequence length is equal to $\mathbf{SeqLen} \times$ **NumberOfGPUs**).

The result shows Forge consistently outperforms the LASP-2 baseline across all sequence lengths. For instance, at a sequence length of 16k on 4 GPUs, Forge achieves a **1.17× speedup** over LASP-2 (0.96ms vs. 1.13ms). Even with the high bandwidth of NVLink on H100s, Forge maintains its efficiency advantage. This confirms that our fine-grained compute-communication overlap strategy is effective not only on bandwidth-constrained hardware (like the H20 used in the main result) but also on high-performance flagship clusters. As expected, Linear Attention methods (both LASP-2 and Forge) are orders of magnitude faster than Ring-Attention, especially at longer sequences (e.g., at 512k global sequence length, Forge is over **160× faster** than Ring-Attention).

These results validate that the performance gains reported in the main paper are not artifacts of the H20 hardware or larger batch sizes, but rather stem from the fundamental efficiency of Forge's generated kernels and scheduling logic.

Table 4: **Distributed Latency Comparison on H100 GPUs (Batch Size = 1).** We compare the end-to-end latency of Forge (Scalar GLA) against Ring-Attention (Softmax) and LASP-2 (State-of-the-art Linear Attention Sequence Parallelism). Forge consistently achieves the lowest latency across all sequence lengths and GPU configurations.

# GPUs	SeqLen Per GPU	Latency (ms)		
		Ring-Attn	LASP-2	**Forge (Ours)**
4	8192	8.40	1.02	**0.79**
	16384	17.77	1.13	**0.96**
	32768	61.65	1.66	**1.57**
	65536	229.43	2.90	**2.76**
8	8192	19.32	1.00	**0.77**
	16384	38.09	1.15	**0.97**
	32768	133.14	1.67	**1.62**
	65536	462.30	2.92	**2.84**

G Code Example

In this section we provide code examples to implement different linear attention variants and the code generate by Forge. Firt we show the Scalar GLA in Listing 1 and the generated code in List-

ing 2. Forge enable pre-compiled kernel (AOT) and static dispatch at runtime. In this example, we tune the kernel on H=[1,4,8,16,32] and D=[64, 128], so there are a lot of auto-generated code encoded pre-tuned information for dispatcher (our AOT optimization). Using Forge to implement this kernel only involves 50+ lines of code (LOC), which the generated triton code and host launcher is around 1200 LOC. Even without including the static dispatcher and host launcher in count, the generated Triton kernel alone has over 400 LOC. And this is only partially generated content, because Forge also generates code for other parallel strategies.

Then we demonstrate the implementation of DeltaNet using Forge in Listing 3. Note that DeltaNet involves complex data dependencies where intermediate results computed in the chunk phase (specifically U and W) are required in the merge phase. Forge introduces a primitive `forge.cache_result` to explicit mark these tensors. The compiler then automatically handles the memory layout and data movement to ensure these values are efficiently reused across phases without redundant re-computation or manual memory management.

G.1 Implementation and code generation for Scalar GLA

```
def chunk_mode_scalar_gla(k: Tensor, v: Tensor, g: Tensor) -> Tensor:
    """k: [C, K], v: [C, V], g: [C]"""
    g_cumsum = g.cumsum(dim=0)
    g_cumsum_last = g.sum(dim=0)
    g_cumsum = (g_cumsum_last - g_cumsum).unsqueeze(0).exp()
    k_fp32 = k.permute([1, 0]).to(g.dtype)
    k_decay = (k_fp32 * g_cumsum).to(v.dtype)
    chunk_state = k_decay @ v
    return chunk_state

def decay_mode_scalar_gla(prev_s: Tensor, chunk_state: Tensor, g: Tensor)
     -> Tensor:
    """g: [C]"""
    g_sum = g.sum(dim=0).exp()
    return prev_s * g_sum + chunk_state

def merge_mode_scalar_gla(
    q: Tensor,
    k: Tensor,
    v: Tensor,
    g: Tensor,
    chunk_state: Tensor,
    scale: Tensor,
) -> Tensor:
    """q: [C, K], k: [C, K], v: [C, V], chunk_state: [K, V]"""
    g_cumsum = g.cumsum(0)
    chunk_state = chunk_state.to(q.dtype)
    p = (q @ k.T).tril(0)
    p = (p * (g_cumsum[..., None] - g_cumsum[None, ...]).exp()).to(v.
    dtype)
    return (p @ v + q @ chunk_state * g_cumsum.exp()[..., None]) * scale

CONST_H = ConstExpr("H", H)
CONST_K = ConstExpr("K", K)
CONST_V = ConstExpr("V", V)
meta = {
    "q": SymbTensor(["T", CONST_H, CONST_K], dtype=dtype),
    "k": SymbTensor(["T", CONST_H, CONST_K], dtype=dtype),
    "v": SymbTensor(["T", CONST_H, CONST_V], dtype=dtype),
    "g": SymbTensor(["T", CONST_H], dtype=torch.float32),
    "prev_s": SymbTensor(["NS", CONST_H, CONST_K, CONST_V], dtype=torch.
    float32),
    "chunk_state": SymbTensor(["NC", CONST_H, CONST_K, CONST_V], dtype=
    dtype),
    "scale": 1 / math.sqrt(K),
}
```

```
LinearAttention(
    input_meta=meta,
    sp_group=pg if args.sp else None,
    enable_aot=args.aot,
    code_dir=args.dir,
    chunk_mode=chunk_mode_scalar_gla,
    decay_mode=decay_mode_scalar_gla,
    merge_mode=merge_mode_scalar_gla,
)
```

Listing 1: Implementation of Scalar GLA in Forge

```
fuse_chunk_decay_kernel_signature = (
    "*bf16:16, "
    "*bf16:16, "
    "*fp32, "
    "*bf16:16, "
    "*bf16:16, "
    "*fp32, "
    "*i32, "
    "i32, "
    "i32, "
    "*i32, "
    "%USE_INITIAL_STATE, "
    "%H, "
    "%K, "
    "%V, "
    "%CHUNK, "
    "%BLK_K, "
    "%BLK_V, "
    "%USE_TMA"
)


def get_fuse_chunk_decay_kernel_info(B: int, T: int, H: int, K: int, V: int):
    """Static dispatcher for fuse_chunk_decay_kernel. Auto-generated."""
    D = [K, V]
    key = (B, T, H, D)
    if key in (
        (1, 4096, 4, [64, 64]),
        (2, 1024, 4, [64, 64]),
    ):
        BLK_K = 32
        BLK_V = 32
        num_warps = 8
        num_ctas = 1
        num_stages = 3
        maxnreg = None
    elif key in (
        (1, 1024, 1, [64, 64]),
        (1, 1024, 1, [128, 128]),
        (1, 1024, 4, [64, 64]),
        (1, 1024, 4, [128, 128]),
        (1, 1024, 8, [64, 64]),
        (1, 1024, 8, [128, 128]),
        (1, 1024, 16, [64, 64]),
        (1, 1024, 32, [64, 64]),
        (1, 2048, 1, [64, 64]),
        (1, 2048, 1, [128, 128]),
        (1, 2048, 4, [64, 64]),
        (1, 2048, 4, [128, 128]),
        (1, 2048, 8, [64, 64]),
        (1, 2048, 8, [128, 128]),
        (1, 2048, 16, [64, 64]),
```

```
        (1, 2048, 32, [64, 64]),
        (1, 4096, 1, [64, 64]),
        (1, 4096, 1, [128, 128]),
        (1, 4096, 4, [128, 128]),
        (1, 4096, 8, [64, 64]),
        (1, 4096, 8, [128, 128]),
        (1, 4096, 16, [64, 64]),
        (1, 4096, 32, [64, 64]),
        (1, 8192, 1, [64, 64]),
        (1, 8192, 1, [128, 128]),
        (1, 8192, 4, [64, 64]),
        (1, 8192, 4, [128, 128]),
        (1, 8192, 8, [64, 64]),
        (1, 8192, 8, [128, 128]),
        (1, 8192, 16, [64, 64]),
        (1, 8192, 32, [64, 64]),
        (1, 16384, 1, [64, 64]),
        (1, 16384, 1, [128, 128]),
        (1, 16384, 4, [64, 64]),
        (1, 16384, 4, [128, 128]),
        (1, 16384, 8, [64, 64]),
        (1, 16384, 8, [128, 128]),
        (1, 16384, 16, [64, 64]),
        (1, 16384, 32, [64, 64]),
        (1, 32768, 1, [64, 64]),
        (1, 32768, 1, [128, 128]),
        (1, 32768, 4, [64, 64]),
        (1, 32768, 4, [128, 128]),
        (1, 32768, 8, [64, 64]),
        (1, 32768, 8, [128, 128]),
        (1, 32768, 16, [64, 64]),
        (1, 32768, 32, [64, 64]),
        (1, 65536, 1, [64, 64]),
        (1, 65536, 1, [128, 128]),
        (1, 65536, 4, [64, 64]),
        (1, 65536, 4, [128, 128]),
        (1, 65536, 8, [64, 64]),
        (1, 65536, 8, [128, 128]),
        (1, 65536, 16, [64, 64]),
        (1, 65536, 32, [64, 64]),
        (1, 131072, 1, [64, 64]),
        (1, 131072, 1, [128, 128]),
        (1, 131072, 4, [64, 64]),
        (1, 131072, 4, [128, 128]),
        (1, 131072, 8, [64, 64]),
        (1, 131072, 8, [128, 128]),
        (1, 131072, 16, [64, 64]),
        (1, 131072, 32, [64, 64]),
        (1, 262144, 1, [64, 64]),
        (1, 262144, 1, [128, 128]),
        (1, 262144, 4, [64, 64]),
        (1, 262144, 4, [128, 128]),
        (1, 262144, 8, [64, 64]),
        (1, 262144, 8, [128, 128]),
        (1, 262144, 16, [64, 64]),
        (1, 262144, 32, [64, 64]),
        (2, 1024, 1, [64, 64]),
        (2, 1024, 1, [128, 128]),
        (2, 1024, 4, [128, 128]),
        (2, 1024, 8, [64, 64]),
        (2, 1024, 16, [64, 64]),
        (2, 2048, 1, [64, 64]),
        (2, 2048, 1, [128, 128]),
        (2, 2048, 4, [64, 64]),
        (2, 2048, 4, [128, 128]),
```

```
        (2, 2048, 8, [64, 64]),
        (2, 2048, 16, [64, 64]),
        (2, 4096, 1, [64, 64]),
        (2, 4096, 1, [128, 128]),
        (2, 4096, 4, [64, 64]),
        (2, 4096, 4, [128, 128]),
        (2, 4096, 8, [64, 64]),
        (2, 4096, 16, [64, 64]),
        (2, 8192, 1, [64, 64]),
        (2, 8192, 1, [128, 128]),
        (2, 8192, 4, [64, 64]),
        (2, 8192, 4, [128, 128]),
        (2, 8192, 8, [64, 64]),
        (2, 8192, 16, [64, 64]),
        (2, 16384, 1, [64, 64]),
        (2, 16384, 1, [128, 128]),
        (2, 16384, 4, [64, 64]),
        (2, 16384, 4, [128, 128]),
        (2, 16384, 8, [64, 64]),
        (2, 16384, 16, [64, 64]),
        (2, 32768, 1, [64, 64]),
        (2, 32768, 1, [128, 128]),
        (2, 32768, 4, [64, 64]),
        (2, 32768, 4, [128, 128]),
        (2, 32768, 8, [64, 64]),
        (2, 32768, 16, [64, 64]),
        (2, 65536, 1, [64, 64]),
        (2, 65536, 1, [128, 128]),
        (2, 65536, 4, [64, 64]),
        (2, 65536, 4, [128, 128]),
        (2, 65536, 8, [64, 64]),
        (2, 65536, 16, [64, 64]),
        (2, 131072, 1, [64, 64]),
        (2, 131072, 1, [128, 128]),
        (2, 131072, 4, [64, 64]),
        (2, 131072, 4, [128, 128]),
        (2, 131072, 8, [64, 64]),
        (2, 131072, 16, [64, 64]),
    ):
        BLK_K = 32
        BLK_V = 32
        num_warps = 8
        num_ctas = 1
        num_stages = 4
        maxnreg = None
    elif key in (
        (1, 1024, 32, [128, 128]),
        (1, 2048, 32, [128, 128]),
        (1, 4096, 32, [128, 128]),
        (1, 8192, 32, [128, 128]),
        (1, 16384, 32, [128, 128]),
        (1, 32768, 32, [128, 128]),
        (1, 65536, 32, [128, 128]),
        (1, 131072, 32, [128, 128]),
        (1, 262144, 32, [128, 128]),
        (2, 1024, 32, [128, 128]),
        (2, 2048, 16, [128, 128]),
        (2, 2048, 32, [128, 128]),
        (2, 4096, 16, [128, 128]),
        (2, 4096, 32, [128, 128]),
        (2, 8192, 32, [128, 128]),
        (2, 16384, 16, [128, 128]),
        (2, 16384, 32, [128, 128]),
        (2, 32768, 32, [128, 128]),
        (2, 131072, 16, [128, 128]),
```

```
    ):
        BLK_K = 32
        BLK_V = 64
        num_warps = 4
        num_ctas = 1
        num_stages = 3
        maxnreg = None
    elif key in (
        (2, 1024, 16, [128, 128]),
    ):
        BLK_K = 32
        BLK_V = 64
        num_warps = 8
        num_ctas = 1
        num_stages = 3
        maxnreg = None
    elif key in (
        (1, 1024, 16, [128, 128]),
        (1, 2048, 16, [128, 128]),
        (1, 4096, 16, [128, 128]),
        (1, 8192, 16, [128, 128]),
        (1, 16384, 16, [128, 128]),
        (1, 32768, 16, [128, 128]),
        (1, 65536, 16, [128, 128]),
        (1, 131072, 16, [128, 128]),
        (1, 262144, 16, [128, 128]),
        (2, 1024, 8, [128, 128]),
        (2, 1024, 32, [64, 64]),
        (2, 2048, 8, [128, 128]),
        (2, 2048, 32, [64, 64]),
        (2, 4096, 8, [128, 128]),
        (2, 4096, 32, [64, 64]),
        (2, 8192, 8, [128, 128]),
        (2, 8192, 16, [128, 128]),
        (2, 8192, 32, [64, 64]),
        (2, 16384, 8, [128, 128]),
        (2, 16384, 32, [64, 64]),
        (2, 32768, 8, [128, 128]),
        (2, 32768, 16, [128, 128]),
        (2, 32768, 32, [64, 64]),
        (2, 65536, 8, [128, 128]),
        (2, 65536, 16, [128, 128]),
        (2, 65536, 32, [64, 64]),
        (2, 131072, 8, [128, 128]),
        (2, 131072, 32, [64, 64]),
    ):
        BLK_K = 32
        BLK_V = 64
        num_warps = 8
        num_ctas = 1
        num_stages = 4
        maxnreg = None
    elif key in (
        (2, 65536, 32, [128, 128]),
        (2, 131072, 32, [128, 128]),
    ):
        BLK_K = 64
        BLK_V = 128
        num_warps = 8
        num_ctas = 1
        num_stages = 4
        maxnreg = None
    else:
        raise ValueError(f"Unsupported config for fuse_chunk_decay_kernel
    : BTHD={key}")
```

```

    return {
        "CHUNK": 64,
        "USE_INITIAL_STATE": True,
        "H": H,
        "K": D[0],
        "V": D[1],
        "BLK_K": BLK_K,
        "BLK_V": BLK_V,
        "num_warps": num_warps,
        "num_stages": num_stages,
    }



@aot_compile_spaces({
    "fuse_chunk_decay_kernel": {
        "signature": fuse_chunk_decay_kernel_signature,
        "grid": ["(%K + %BLK_K - 1) / %BLK_K", "(%V + %BLK_V - 1) / %BLK_V", "H_MUL_NS"],
        "triton_algo_infos": [
            get_fuse_chunk_decay_kernel_info(B, T, H, K, V)
            for B,T,H,(K,V) in[
                (1, 262144, 32, [64, 64]),
                (1, 262144, 8, [128, 128]),
                (2, 1024, 16, [128, 128]),
                (2, 1024, 4, [64, 64]),
                (2, 131072, 1, [128, 128]),
                (2, 131072, 1, [64, 64]),
                (2, 131072, 16, [128, 128]),
                (2, 131072, 16, [64, 64]),
                (2, 131072, 32, [128, 128]),
                (2, 131072, 32, [64, 64]),
                (2, 131072, 4, [128, 128]),
                (2, 131072, 4, [64, 64]),
                (2, 131072, 8, [128, 128]),
                (2, 131072, 8, [64, 64]),
                (2, 32768, 32, [128, 128]),
                (2, 65536, 16, [128, 128])
            ]
        ],
    }
})
@triton.autotune(
    configs=[
        triton.Config({"BLK_K": BLK_K, "BLK_V": BLK_V, "USE_TMA": USE_TMA}, num_warps=num_warps, num_stages=num_stages)
        for num_warps in [4, 8]
        for num_stages in [3, 4]
        for BLK_K in [128, 64, 32]
        for BLK_V in [128, 64, 32]
        for USE_TMA in [True, False]
    ],
    key=[],
)
@triton.jit
def fuse_chunk_decay_kernel(
    k,
    v,
    g,
    prev_s,
    out_0,
    out_1,
    cu_seqlens,
```

```
    NS,
    H_MUL_NS,
    chunk_offsets_with_ini,
    USE_INITIAL_STATE: tl.constexpr,
    H: tl.constexpr,
    K: tl.constexpr,
    V: tl.constexpr,
    CHUNK: tl.constexpr,
    BLK_K: tl.constexpr,
    BLK_V: tl.constexpr,
    USE_TMA: tl.constexpr,
):
    NUM_BLK_K = (K + BLK_K - 1) // BLK_K
    NUM_BLK_V = (V + BLK_V - 1) // BLK_V
    NUM_BLK_KV = NUM_BLK_K * NUM_BLK_V

    i_k, i_v, i_sh = tl.program_id(0), tl.program_id(1), tl.program_id(2)
    i_s, i_h = i_sh // H, i_sh % H
    bos, eos = tl.load(cu_seqlens + i_s).to(tl.int32), tl.load(cu_seqlens
     + i_s + 1).to(tl.int32)
    T = eos - bos
    NC = tl.cdiv(T, CHUNK)
    boh = tl.load(chunk_offsets_with_ini + i_s).to(tl.int32)

    out_0 = out_0 + (boh * H + i_h).to(tl.int64) * K * V
    prev_s = prev_s + (i_s * H + i_h) * K * V
    initial_decay = 1.0
    # out_1: [NC + NS, H]. NOTE: **not** in log space
    ptr_out_1 = out_1 + boh * H + i_h
    # store initial decay
    tl.store(ptr_out_1, initial_decay)
    ptr_out_1 += H
    # [BK, BV]
    blk_prev_s = tl.zeros([BLK_K, BLK_V], dtype=tl.float32)
    if USE_INITIAL_STATE:
        ptr_prev_s = tl.make_block_ptr(prev_s, (K, V), (V, 1), (i_k *
    BLK_K, i_v * BLK_V), (BLK_K, BLK_V), (1, 0))
        blk_prev_s = tl.load(ptr_prev_s, boundary_check=(0, 1)).to(tl.
    float32)
    ptr_out_0 = tl.make_block_ptr(out_0, (K, V), (V, 1), (i_k*BLK_K, i_v*
    BLK_V), (BLK_K, BLK_V), (1, 0))  # fmt: skip
    tl.store(ptr_out_0, blk_prev_s.to(ptr_out_0.dtype.element_ty))
    out_0 += H * K * V

    if USE_TMA:
        k_desc = tl.make_tensor_descriptor(
            k + bos * H * K + i_h * K,
            shape=[T, K],
            strides=[H * K, 1],
            block_shape=[CHUNK, BLK_K],
        )
        v_desc = tl.make_tensor_descriptor(
            v + bos * H * V + i_h * V,
            shape=[T, V],
            strides=[H * V, 1],
            block_shape=[CHUNK, BLK_V],
        )
        out_0_desc = tl.make_tensor_descriptor(
            out_0,
            shape=[NC, K, V],
            strides=[H * K * V, V, 1],
            block_shape=[1, BLK_K, BLK_V],
        )

    for i_c in range(NC):
```

```
        # load (trans) `k`: (T, H, K,) => (BLK_K, CHUNK,)
        if not USE_TMA:
            cur_k = k + bos * H * K + i_h * K  # fmt: skip
            ptr_k_0 = tl.make_block_ptr(cur_k, (K, T,), (1, H * K,), (i_k * BLK_K, i_c * CHUNK,), (BLK_K, CHUNK,), (0, 1,))  # fmt: skip
            blk_k_0 = tl.load(ptr_k_0, boundary_check=(0, 1,))  # fmt: skip
        else:
            blk_k_0 = k_desc.load([i_c * CHUNK, i_k * BLK_K]).trans()

        # load `v`: (T, H, V,) => (CHUNK, BLK_V,)
        if not USE_TMA:
            cur_v = v + bos * H * V + i_h * V  # fmt: skip
            ptr_v_1 = tl.make_block_ptr(cur_v, (T, V,), (H * V, 1,), (i_c * CHUNK, i_v * BLK_V,), (CHUNK, BLK_V,), (1, 0,))  # fmt: skip
            blk_v_1 = tl.load(ptr_v_1, boundary_check=(1, 0,))  # fmt: skip
        else:
            blk_v_1 = v_desc.load([i_c * CHUNK, i_v * BLK_V])



        # call_external_func: `chunk_local_cumsum` to pre-compute g

        # load `g`: (T, H,) => (CHUNK,)
        cur_g = g + bos * H + i_h * 1  # fmt: skip
        ptr_g_2 = tl.make_block_ptr(cur_g, (T,), (H,), (i_c * CHUNK,), (CHUNK,), (0,))  # fmt: skip
        blk_g_2 = tl.load(ptr_g_2, boundary_check=(0,))  # fmt: skip
        # load_last_g: => ()
        # load `g` [(CHUNK - 1)]: (T, H,) => (,)
        cur_g = g + bos * H + i_h * 1 + (CHUNK - 1) * H  # fmt: skip
        ptr_last_g_3 = cur_g + i_c * CHUNK * H  # fmt: skip
        last_g_3 = tl.load(ptr_last_g_3)
        # sub: torch.Size([]), ('CHUNK',) => ('CHUNK',)
        sub_4 = last_g_3 - blk_g_2
        # unsqueeze: ('CHUNK',) => (1, 'CHUNK')
        unsqueeze_5 = sub_4[None, :]
        # exp: (1, 'CHUNK') => (1, 'CHUNK')
        exp_6 = tl.exp(unsqueeze_5)
        # to: ('BLK_K', 'CHUNK') => ('BLK_K', 'CHUNK')
        blk_k_0_float32_7 = blk_k_0.to(tl.float32)
        # mul: ('BLK_K', 'CHUNK'), (1, 'CHUNK') => ('BLK_K', 'CHUNK')
        mul_8 = blk_k_0_float32_7 * exp_6
        # to: ('BLK_K', 'CHUNK') => ('BLK_K', 'CHUNK')
        mul_8_bfloat16_9 = mul_8.to(tl.bfloat16)
        # matmul: ('BLK_K', 'CHUNK'), ('CHUNK', 'BLK_V') => ('BLK_K', 'BLK_V')
        matmul_10 = tl.dot(mul_8_bfloat16_9, blk_v_1).to(mul_8_bfloat16_9.dtype)
        # exp: () => ()
        exp_1_11 = tl.exp(last_g_3)
        # mul: ('BLK_K', 'BLK_V'), () => ('BLK_K', 'BLK_V')
        mul_1_12 = blk_prev_s * exp_1_11
        # add: ('BLK_K', 'BLK_V'), ('BLK_K', 'BLK_V') => ('BLK_K', 'BLK_V')
        add_13 = mul_1_12 + matmul_10
        # mul: ('s3',), () => ('s3',)
        mul_tensor_14 = initial_decay * exp_1_11
        # store => ('BLK_K', 'BLK_V')
        # assume output layout: [NC_WITH_INI, H, K, V]
        out_0_ty = out_0.dtype.element_ty
        if not USE_TMA:
            ptr_out_0 = tl.make_block_ptr(out_0, (K, V,), (V, 1,), (i_k * BLK_K, i_v * BLK_V,), (BLK_K, BLK_V,), (1, 0))  # fmt: skip
```

```
            tl.store(ptr_out_0, add_13.to(out_0_ty), boundary_check=(1, 0))
            out_0 += H * K * V
        else:
            out_0_desc.store([i_c, i_k * BLK_K, i_v * BLK_V], add_13.to(out_0_ty)[None, :, :])
        blk_prev_s = add_13.to(blk_prev_s.dtype)
        # store => ('s3',)
        # output layout: [NC_WITH_INI, H, s3]
        out_1_ty = out_1.dtype.element_ty
        tl.store(ptr_out_1 + i_c * H, mul_tensor_14.to(out_1_ty))
        initial_decay = mul_tensor_14

def launch_fuse_chunk_decay(
    g: torch.Tensor,
    prev_s: torch.Tensor,
    v: torch.Tensor,
    k: torch.Tensor,
    cu_seqlens: torch.IntTensor,
    cached_results: dict,
    dist_scan: DistScanContext = None,
    lazy_update: bool = True,
) -> dict[str, torch.Tensor]:
    '''
    perform "decayed scan" on `chunked_states`
    '''
    # state_dtype = chunk_state.dtype
    chunk_decay = None
    prev_rank_state_sum = None

    # TODO: now fix chunk_size to 64
    CHUNK = 64
    chunk_indices = prepare_chunk_indices(cu_seqlens, CHUNK)
    chunk_offsets = prepare_chunk_offsets(cu_seqlens, CHUNK)
    chunk_offsets_with_ini = prepare_chunk_offsets_with_ini(cu_seqlens, CHUNK)
    NS, NC = len(cu_seqlens) - 1, chunk_indices.shape[0]
    NC_WITH_INI = NC + NS
    use_aot = os.environ.get('FORGE_USE_AOT', '0')
    FORGE_USE_AOT = True if use_aot.lower() in ['1', 'true', 'yes'] else None
    T, H, K, V = *k.shape, v.shape[-1]
    updated_states = k.new_empty([NC_WITH_INI, H, K, V])

    def alloc_fn(size: int, alignment: int, stream: int):
        return torch.empty(size, device='cuda', dtype=torch.int8)

    triton.set_allocator(alloc_fn)

    def grid(meta):
        BLK_K = meta['BLK_K']
        BLK_V = meta['BLK_V']
        NUM_BLK_K = (K + BLK_K - 1) // BLK_K
        NUM_BLK_V = (V + BLK_V - 1) // BLK_V
        NUM_BLK_KV = NUM_BLK_K * NUM_BLK_V
        H_MUL_NS = H * NS
        return (NUM_BLK_K, NUM_BLK_V, H_MUL_NS,)

    chunk_decay = torch.empty([NC_WITH_INI, H], dtype=torch.float32)
    if FORGE_USE_AOT is None:
        fuse_chunk_decay_kernel[grid](
            g=g,
            prev_s=prev_s,
            v=v,
            k=k,
```

```
            out_1=chunk_decay,
            cu_seqlens=cu_seqlens,
            H=H,
            NS=NS,
            K=K,
            V=V,
            CHUNK=CHUNK,
            H_MUL_NS=H * NS,
            out_0=updated_states,
            chunk_offsets_with_ini=chunk_offsets_with_ini,
            USE_INITIAL_STATE=True,
        )

    else:

        from foge.aot_utils import forge_aot_ops

        algo_info = forge_aot_ops.
    fuse_chunk_decay_kernel__triton_algo_info_t()
        for _k, _v in get_fuse_chunk_decay_kernel_info(NS, T, H, K, V).
    items():
            setattr(algo_info, _k, _v)
        forge_aot_ops.fuse_chunk_decay_kernel(
            0,  # torch.cuda.current_stream().cuda_stream,
            k.data_ptr(),  # k
            v.data_ptr(),  # v
            g.data_ptr(),  # g
            prev_s.data_ptr(),  # prev_s
            updated_states.data_ptr(),  # out_0
            chunk_decay.data_ptr(),  # out_1
            cu_seqlens.data_ptr(),  # cu_seqlens
            NS,  # NS
            H * NS,  # H_MUL_NS
            chunk_offsets_with_ini.data_ptr(),  # chunk_offsets_with_ini
            algo_info,
        )


    final_chunk_indices = chunk_offsets_with_ini[1:] - 1
    final_state_local = updated_states[final_chunk_indices, ...]
    final_decay_local = chunk_decay[final_chunk_indices, ...]

    if dist_scan.pg.size() > 1:
        prev_rank_state_sum = dist_scan.forward(
            final_state_local=final_state_local,
            final_decay_local=final_decay_local,
            decay_type=DecayType.SCALAR,
            lazy_update=True,
        )

    cached_results.update({})
    if prev_rank_state_sum is not None:
        return {
            "chunk_state": updated_states,
            "chunk_decay": chunk_decay,
            "prev_rank_state_sum": prev_rank_state_sum,
        }
    else:
        return {
            "chunk_state": updated_states,
            "chunk_decay": None,
            "prev_rank_state_sum": None,
        }
merge_mode_kernel_signature = (
    "*bf16:16, "
```

```
    "*bf16:16, "
    "*bf16:16, "
    "*fp32, "
    "*bf16:16, "
    "fp32, "
    "*bf16:16, "
    "i32:16, "
    "*bf16:16, "
    "i32:16, "
    "*bf16:16, "
    "*i32, "
    "i32, "
    "i32, "
    "*i32, "
    "i32, "
    "%FUSE_SP_STATE_UPDATE, "
    "%H, "
    "%K, "
    "%V, "
    "%CHUNK, "
    "%BLK_K, "
    "%BLK_V, "
    "%USE_TMA"
)


def get_merge_mode_kernel_info(B: int, T: int, H: int, K: int, V: int):
    """Static dispatcher for merge_mode_kernel. Auto-generated."""
    D = [K, V]
    key = (B, T, H, D)
    if key in (
        (1, 4096, 4, [128, 128]),
        (1, 131072, 32, [128, 128]),
        (1, 262144, 32, [128, 128]),
        (2, 1024, 8, [128, 128]),
        (2, 4096, 1, [128, 128]),
        (2, 8192, 1, [128, 128]),
    ):
        BLK_K = 128
        BLK_V = 128
        num_warps = 4
        num_ctas = 1
        num_stages = 3
        maxnreg = None
    elif key in (
        (1, 4096, 32, [128, 128]),
        (1, 16384, 8, [128, 128]),
        (1, 16384, 32, [128, 128]),
        (1, 32768, 8, [128, 128]),
        (1, 32768, 16, [128, 128]),
        (1, 32768, 32, [128, 128]),
        (1, 65536, 8, [128, 128]),
        (1, 65536, 32, [128, 128]),
        (1, 131072, 4, [128, 128]),
        (1, 131072, 8, [128, 128]),
        (1, 131072, 16, [128, 128]),
        (1, 262144, 4, [128, 128]),
        (1, 262144, 8, [128, 128]),
        (1, 262144, 16, [128, 128]),
        (2, 4096, 16, [128, 128]),
        (2, 4096, 32, [128, 128]),
        (2, 16384, 8, [128, 128]),
        (2, 16384, 32, [128, 128]),
        (2, 32768, 16, [128, 128]),
        (2, 32768, 32, [128, 128]),
```

```
        (2, 65536, 8, [128, 128]),
        (2, 65536, 16, [128, 128]),
        (2, 131072, 4, [128, 128]),
        (2, 131072, 8, [128, 128]),
        (2, 131072, 16, [128, 128]),
    ):
        BLK_K = 128
        BLK_V = 128
        num_warps = 8
        num_ctas = 1
        num_stages = 3
        maxnreg = None
    elif key in (
        (1, 1024, 16, [128, 128]),
        (1, 1024, 32, [128, 128]),
        (1, 2048, 8, [128, 128]),
        (1, 2048, 16, [128, 128]),
        (1, 16384, 1, [128, 128]),
        (1, 32768, 1, [128, 128]),
        (2, 2048, 4, [128, 128]),
        (2, 65536, 32, [128, 128]),
        (2, 131072, 32, [128, 128]),
    ):
        BLK_K = 128
        BLK_V = 128
        num_warps = 4
        num_ctas = 1
        num_stages = 4
        maxnreg = None
    elif key in (
        (1, 2048, 32, [128, 128]),
        (1, 4096, 16, [128, 128]),
        (1, 8192, 16, [128, 128]),
        (1, 8192, 32, [128, 128]),
        (1, 16384, 16, [128, 128]),
        (1, 65536, 4, [128, 128]),
        (1, 65536, 16, [128, 128]),
        (2, 1024, 32, [128, 128]),
        (2, 2048, 32, [128, 128]),
        (2, 8192, 8, [128, 128]),
        (2, 8192, 16, [128, 128]),
        (2, 8192, 32, [128, 128]),
        (2, 16384, 16, [128, 128]),
        (2, 32768, 4, [128, 128]),
        (2, 32768, 8, [128, 128]),
        (2, 65536, 4, [128, 128]),
    ):
        BLK_K = 128
        BLK_V = 128
        num_warps = 8
        num_ctas = 1
        num_stages = 4
        maxnreg = None
    elif key in (
        (1, 1024, 1, [64, 64]),
        (1, 4096, 1, [64, 64]),
    ):
        BLK_K = 128
        BLK_V = 32
        num_warps = 4
        num_ctas = 1
        num_stages = 3
        maxnreg = None
    elif key in (
        (1, 1024, 1, [128, 128]),
```

```
        (1, 1024, 4, [64, 64]),
        (1, 1024, 8, [64, 64]),
        (1, 2048, 1, [64, 64]),
        (1, 2048, 1, [128, 128]),
        (1, 2048, 4, [64, 64]),
        (2, 1024, 1, [64, 64]),
        (2, 2048, 1, [64, 64]),
    ):
        BLK_K = 128
        BLK_V = 32
        num_warps = 4
        num_ctas = 1
        num_stages = 4
        maxnreg = None
    elif key in (
        (1, 1024, 8, [128, 128]),
        (1, 1024, 32, [64, 64]),
        (1, 2048, 4, [128, 128]),
        (1, 4096, 4, [64, 64]),
        (1, 4096, 8, [64, 64]),
        (1, 4096, 8, [128, 128]),
        (1, 8192, 1, [128, 128]),
        (1, 8192, 8, [64, 64]),
        (1, 8192, 8, [128, 128]),
        (1, 8192, 32, [64, 64]),
        (1, 16384, 1, [64, 64]),
        (1, 16384, 32, [64, 64]),
        (1, 32768, 1, [64, 64]),
        (1, 32768, 4, [64, 64]),
        (1, 32768, 8, [64, 64]),
        (1, 32768, 16, [64, 64]),
        (1, 65536, 1, [128, 128]),
        (1, 131072, 1, [128, 128]),
        (1, 131072, 32, [64, 64]),
        (1, 262144, 1, [64, 64]),
        (1, 262144, 32, [64, 64]),
        (2, 1024, 4, [64, 64]),
        (2, 1024, 4, [128, 128]),
        (2, 1024, 8, [64, 64]),
        (2, 1024, 16, [64, 64]),
        (2, 1024, 32, [64, 64]),
        (2, 2048, 4, [64, 64]),
        (2, 2048, 8, [64, 64]),
        (2, 2048, 8, [128, 128]),
        (2, 2048, 16, [64, 64]),
        (2, 2048, 16, [128, 128]),
        (2, 2048, 32, [64, 64]),
        (2, 4096, 4, [64, 64]),
        (2, 4096, 4, [128, 128]),
        (2, 4096, 8, [64, 64]),
        (2, 4096, 8, [128, 128]),
        (2, 4096, 16, [64, 64]),
        (2, 4096, 32, [64, 64]),
        (2, 8192, 1, [64, 64]),
        (2, 8192, 4, [128, 128]),
        (2, 16384, 1, [64, 64]),
        (2, 16384, 1, [128, 128]),
        (2, 16384, 4, [128, 128]),
        (2, 16384, 8, [64, 64]),
        (2, 16384, 32, [64, 64]),
        (2, 32768, 16, [64, 64]),
        (2, 32768, 32, [64, 64]),
        (2, 65536, 1, [64, 64]),
        (2, 65536, 1, [128, 128]),
        (2, 65536, 16, [64, 64]),
```

```
    ):
        BLK_K = 128
        BLK_V = 64
        num_warps = 4
        num_ctas = 1
        num_stages = 3
        maxnreg = None
    elif key in (
        (1, 4096, 1, [128, 128]),
    ):
        BLK_K = 128
        BLK_V = 64
        num_warps = 8
        num_ctas = 1
        num_stages = 3
        maxnreg = None
    elif key in (
        (1, 1024, 4, [128, 128]),
        (1, 1024, 16, [64, 64]),
        (1, 2048, 8, [64, 64]),
        (1, 2048, 16, [64, 64]),
        (1, 2048, 32, [64, 64]),
        (1, 4096, 16, [64, 64]),
        (1, 4096, 32, [64, 64]),
        (1, 8192, 1, [64, 64]),
        (1, 8192, 4, [64, 64]),
        (1, 8192, 4, [128, 128]),
        (1, 8192, 16, [64, 64]),
        (1, 16384, 4, [64, 64]),
        (1, 16384, 4, [128, 128]),
        (1, 16384, 8, [64, 64]),
        (1, 16384, 16, [64, 64]),
        (1, 32768, 4, [128, 128]),
        (1, 32768, 32, [64, 64]),
        (1, 65536, 1, [64, 64]),
        (1, 65536, 4, [64, 64]),
        (1, 65536, 8, [64, 64]),
        (1, 65536, 16, [64, 64]),
        (1, 65536, 32, [64, 64]),
        (1, 131072, 1, [64, 64]),
        (1, 131072, 4, [64, 64]),
        (1, 131072, 8, [64, 64]),
        (1, 131072, 16, [64, 64]),
        (1, 262144, 1, [128, 128]),
        (1, 262144, 4, [64, 64]),
        (1, 262144, 8, [64, 64]),
        (1, 262144, 16, [64, 64]),
        (2, 1024, 16, [128, 128]),
        (2, 8192, 4, [64, 64]),
        (2, 8192, 8, [64, 64]),
        (2, 8192, 16, [64, 64]),
        (2, 8192, 32, [64, 64]),
        (2, 16384, 4, [64, 64]),
        (2, 16384, 16, [64, 64]),
        (2, 32768, 1, [64, 64]),
        (2, 32768, 1, [128, 128]),
        (2, 32768, 4, [64, 64]),
        (2, 32768, 8, [64, 64]),
        (2, 65536, 4, [64, 64]),
        (2, 65536, 8, [64, 64]),
        (2, 65536, 32, [64, 64]),
        (2, 131072, 1, [64, 64]),
        (2, 131072, 1, [128, 128]),
        (2, 131072, 4, [64, 64]),
        (2, 131072, 8, [64, 64]),
```

```
        (2, 131072, 16, [64, 64]),
        (2, 131072, 32, [64, 64]),
    ):
        BLK_K = 128
        BLK_V = 64
        num_warps = 4
        num_ctas = 1
        num_stages = 4
        maxnreg = None
    elif key in (
        (2, 1024, 1, [128, 128]),
        (2, 2048, 1, [128, 128]),
        (2, 4096, 1, [64, 64]),
    ):
        BLK_K = 128
        BLK_V = 64
        num_warps = 8
        num_ctas = 1
        num_stages = 4
        maxnreg = None
    else:
        raise ValueError(f"Unsupported config for merge_mode_kernel: BTHD={key}")

    return {
        "CHUNK": 64,
        "FUSE_SP_STATE_UPDATE": True,
        "H": H,
        "K": D[0],
        "V": D[1],
        "BLK_K": BLK_K,
        "BLK_V": BLK_V,
        "num_warps": num_warps,
        "num_stages": num_stages,
    }



@aot_compile_spaces({
    "merge_mode_kernel": {
        "signature": merge_mode_kernel_signature,
        "grid": ["((%K + %BLK_K - 1) / %BLK_K) * ((%V + %BLK_V - 1) / %BLK_V)", "%H", "NC"],
        "triton_algo_infos": [
            get_merge_mode_kernel_info(B, T, H, K, V)
            for B,T,H,(K,V) in[
                (1, 1024, 8, [64, 64]),
                (1, 2048, 1, [128, 128]),
                (1, 2048, 16, [128, 128]),
                (1, 2048, 4, [64, 64]),
                (1, 2048, 8, [128, 128]),
                (1, 262144, 32, [128, 128]),
                (1, 32768, 1, [128, 128]),
                (1, 32768, 4, [128, 128]),
                (1, 4096, 1, [128, 128]),
                (1, 4096, 1, [64, 64]),
                (1, 4096, 4, [128, 128]),
                (2, 1024, 16, [128, 128]),
                (2, 1024, 8, [128, 128]),
                (2, 131072, 1, [128, 128]),
                (2, 131072, 1, [64, 64]),
                (2, 131072, 16, [128, 128]),
                (2, 131072, 16, [64, 64]),
                (2, 131072, 32, [128, 128]),
```

```
                (2, 131072, 32, [64, 64]),
                (2, 131072, 4, [128, 128]),
                (2, 131072, 4, [64, 64]),
                (2, 131072, 8, [128, 128]),
                (2, 131072, 8, [64, 64]),
                (2, 16384, 16, [128, 128]),
                (2, 16384, 4, [128, 128]),
                (2, 16384, 8, [64, 64]),
                (2, 2048, 1, [128, 128]),
                (2, 2048, 1, [64, 64]),
                (2, 2048, 16, [128, 128]),
                (2, 2048, 4, [128, 128]),
                (2, 32768, 32, [128, 128]),
                (2, 32768, 32, [64, 64]),
                (2, 32768, 8, [128, 128]),
                (2, 4096, 1, [64, 64]),
                (2, 4096, 4, [64, 64]),
                (2, 4096, 8, [128, 128]),
                (2, 65536, 1, [128, 128]),
                (2, 65536, 1, [64, 64]),
                (2, 65536, 16, [64, 64]),
                (2, 65536, 4, [128, 128]),
                (2, 8192, 1, [128, 128]),
                (2, 8192, 32, [128, 128])
            ]
        ],
    }
})
@triton.autotune(
    configs=[
        triton.Config({"BLK_K": BLK_K, "BLK_V": BLK_V, "USE_TMA": USE_TMA
    }, num_warps=num_warps, num_stages=num_stages)
        for num_warps in [4, 8]
        for num_stages in [3, 4]
        for BLK_K in [128]
        for BLK_V in [128, 64, 32]
        for USE_TMA in [True, False]
    ],
    key=[],
)
@triton.jit
def merge_mode_kernel(
    q,
    k,
    v,
    g,
    chunk_state,
    scale,
    prev_rank_state_sum,
    stride_d0_ns_prev_rank_state_sum,
    chunk_decay,
    stride_d0_nc_with_ini_chunk_decay,
    out_0,
    cu_seqlens,
    NS,
    H_MUL_NS,
    chunk_indices,
    NC,
    FUSE_SP_STATE_UPDATE: tl.constexpr,
    H: tl.constexpr,
    K: tl.constexpr,
    V: tl.constexpr,
    CHUNK: tl.constexpr,
    BLK_K: tl.constexpr,
    BLK_V: tl.constexpr,
```

```
    USE_TMA: tl.constexpr,
):
    NUM_BLK_K = (K + BLK_K - 1) // BLK_K
    NUM_BLK_V = (V + BLK_V - 1) // BLK_V
    NUM_BLK_KV = NUM_BLK_K * NUM_BLK_V

    i_v, i_h, i_gc = tl.program_id(0), tl.program_id(1), tl.program_id(2)
    i_s, i_c = tl.load(chunk_indices + i_gc * 2).to(tl.int32), tl.load(chunk_indices + i_gc * 2 + 1).to(tl.int32)
    bos, eos = tl.load(cu_seqlens + i_s).to(tl.int32), tl.load(cu_seqlens + i_s + 1).to(tl.int32)
    T = eos - bos
    # NC = tl.cdiv(T, CHUNK)

    # FIXME: set `i_k` temporarily
    i_k = 0
    out_0 = out_0 + (bos * H + i_h) * V

    if USE_TMA:
        q_desc = tl.make_tensor_descriptor(
            q + bos * H * K + i_h * K,
            shape=[T, K],
            strides=[H * K, 1],
            block_shape=[CHUNK, BLK_K],
        )
        k_desc = tl.make_tensor_descriptor(
            k + bos * H * K + i_h * K,
            shape=[T, K],
            strides=[H * K, 1],
            block_shape=[CHUNK, BLK_K],
        )
        v_desc = tl.make_tensor_descriptor(
            v + bos * H * V + i_h * V,
            shape=[T, V],
            strides=[H * V, 1],
            block_shape=[CHUNK, BLK_V],
        )
        chunk_state_desc = tl.make_tensor_descriptor(
            chunk_state + (i_gc + i_s).to(tl.int64) * H * K * V + i_h * K * V,
            shape=[K, V],
            strides=[V, 1],
            block_shape=[BLK_K, BLK_V],
        )

    if FUSE_SP_STATE_UPDATE:
        prev_rank_state_sum_desc = tl.make_tensor_descriptor(
            prev_rank_state_sum + i_s * stride_d0_ns_prev_rank_state_sum + i_h * K * V,
            shape=[K, V],
            strides=[V, 1],
            block_shape=[BLK_K, BLK_V],
        )

    # load `q`: (T, H, K,) => (CHUNK, BLK_K,)
    if not USE_TMA:
        cur_q = q + bos * H * K + i_h * K  # fmt: skip
        ptr_q_0 = tl.make_block_ptr(cur_q, (T, K,), (H * K, 1,), (i_c * CHUNK, i_k * BLK_K,), (CHUNK, BLK_K,), (1, 0,))  # fmt: skip
        blk_q_0 = tl.load(ptr_q_0, boundary_check=(1, 0,))  # fmt: skip
    else:
        blk_q_0 = q_desc.load([i_c * CHUNK, i_k * BLK_K])

    # load (trans) `k`: (T, H, K,) => (BLK_K, CHUNK,)
    if not USE_TMA:
```

```
        cur_k = k + bos * H * K + i_h * K  # fmt: skip
        ptr_k_1 = tl.make_block_ptr(cur_k, (K, T,), (1, H * K,), (i_k * BLK_K, i_c * CHUNK,), (BLK_K, CHUNK,), (0, 1,))  # fmt: skip
        blk_k_1 = tl.load(ptr_k_1, boundary_check=(0, 1,))  # fmt: skip
    else:
        blk_k_1 = k_desc.load([i_c * CHUNK, i_k * BLK_K]).trans()

    # load `v`: (T, H, V,) => (CHUNK, BLK_V,)
    if not USE_TMA:
        cur_v = v + bos * H * V + i_h * V  # fmt: skip
        ptr_v_2 = tl.make_block_ptr(cur_v, (T, V,), (H * V, 1,), (i_c * CHUNK, i_v * BLK_V,), (CHUNK, BLK_V,), (1, 0,))  # fmt: skip
        blk_v_2 = tl.load(ptr_v_2, boundary_check=(1, 0,))  # fmt: skip
    else:
        blk_v_2 = v_desc.load([i_c * CHUNK, i_v * BLK_V])

    # call_external_func: `chunk_local_cumsum` to pre-compute g

    # load `g`: (T, H,) => (CHUNK,)
    cur_g = g + bos * H + i_h * 1  # fmt: skip
    ptr_g_3 = tl.make_block_ptr(cur_g, (T,), (H,), (i_c * CHUNK,), (CHUNK,), (0,))  # fmt: skip
    blk_g_3 = tl.load(ptr_g_3, boundary_check=(0,))  # fmt: skip
    # load `chunk_state`: (NC_WITH_INI, H, K, V,) => (BLK_K, BLK_V,)
    if not USE_TMA:
        cur_chunk_state = chunk_state + (i_gc + i_s).to(tl.int64) * H * K * V + i_h * K * V  # fmt: skip
        ptr_chunk_state_4 = tl.make_block_ptr(cur_chunk_state, (K, V,), (V, 1,), (i_k * BLK_K, i_v * BLK_V,), (BLK_K, BLK_V,), (1, 0,))  # fmt: skip
        blk_chunk_state_4 = tl.load(ptr_chunk_state_4, boundary_check=(1, 0,))  # fmt: skip
    else:
        blk_chunk_state_4 = chunk_state_desc.load([i_k * BLK_K, i_v * BLK_V])

    # comments not available for op `if_beg`
    if FUSE_SP_STATE_UPDATE:
        # load `prev_rank_state_sum`: (NS, H, K, V,) => (BLK_K, BLK_V,)
        if not USE_TMA:
            cur_prev_rank_state_sum = prev_rank_state_sum + i_s * stride_d0_ns_prev_rank_state_sum + i_h * K * V  # fmt: skip
            ptr_prev_rank_state_sum_5 = tl.make_block_ptr(cur_prev_rank_state_sum, (K, V,), (V, 1,), (i_k * BLK_K, i_v * BLK_V,), (BLK_K, BLK_V,), (1, 0,))  # fmt: skip
            blk_prev_rank_state_sum_5 = tl.load(ptr_prev_rank_state_sum_5, boundary_check=(1, 0,))  # fmt: skip
        else:
            blk_prev_rank_state_sum_5 = prev_rank_state_sum_desc.load([i_k * BLK_K, i_v * BLK_V])

        # load `chunk_decay`: (NC_WITH_INI, H,) => (,)
        cur_chunk_decay = chunk_decay + (i_gc + i_s).to(tl.int64) * stride_d0_nc_with_ini_chunk_decay + i_h * 1  # fmt: skip
        ptr_chunk_decay_6 = cur_chunk_decay  # fmt: skip
        blk_chunk_decay_6 = tl.load(ptr_chunk_decay_6)
        # to: ('BLK_K', 'BLK_V') => ('BLK_K', 'BLK_V')
        blk_prev_rank_state_sum_5_float32_7 = blk_prev_rank_state_sum_5.to(tl.float32)
        # mul: (), ('BLK_K', 'BLK_V') => ('BLK_K', 'BLK_V')
        mul_tensor_8 = blk_chunk_decay_6 * blk_prev_rank_state_sum_5_float32_7
        # to: ('BLK_K', 'BLK_V') => ('BLK_K', 'BLK_V')
        mul_tensor_8_bfloat16_9 = mul_tensor_8.to(tl.bfloat16)
```

```
        # add: ('BLK_K', 'BLK_V'), ('BLK_K', 'BLK_V') => ('BLK_K', 'BLK_V')
        add_tensor_10 = mul_tensor_8_bfloat16_9 + blk_chunk_state_4
        # comments not available for op `bind_var`
        blk_chunk_state_4 = add_tensor_10
        # comments not available for op `end_if`
    # to: ('BLK_K', 'BLK_V') => ('BLK_K', 'BLK_V')
    blk_chunk_state_4_bfloat16_11 = blk_chunk_state_4.to(tl.bfloat16)
    # matmul: ('CHUNK', 'BLK_K'), ('CHUNK', 'BLK_K') => ('CHUNK', 'CHUNK')
    matmul_12 = tl.dot(blk_q_0, blk_k_1).to(blk_q_0.dtype)
    # tril: ('CHUNK', 'CHUNK') => ('CHUNK', 'CHUNK')
    tril_13 = tl.where(tl.arange(0, CHUNK)[:, None] >= tl.arange(0, CHUNK)[None, :], matmul_12, 0)
    # unsqueeze: ('CHUNK',) => ('CHUNK', 1)
    unsqueeze_14 = blk_g_3[:, None]
    # unsqueeze: ('CHUNK',) => (1, 'CHUNK')
    unsqueeze_1_15 = blk_g_3[None, :]
    # sub: ('CHUNK', 1), (1, 'CHUNK') => ('CHUNK', 'CHUNK')
    sub_16 = unsqueeze_14 - unsqueeze_1_15
    # exp: ('CHUNK', 'CHUNK') => ('CHUNK', 'CHUNK')
    exp_17 = tl.exp(sub_16)
    # mul: ('CHUNK', 'CHUNK'), ('CHUNK', 'CHUNK') => ('CHUNK', 'CHUNK')
    mul_18 = tril_13 * exp_17
    # to: ('CHUNK', 'CHUNK') => ('CHUNK', 'CHUNK')
    mul_18_bfloat16_19 = mul_18.to(tl.bfloat16)
    # matmul: ('CHUNK', 'CHUNK'), ('CHUNK', 'BLK_V') => ('CHUNK', 'BLK_V')
    matmul_1_20 = tl.dot(mul_18_bfloat16_19, blk_v_2).to(
    mul_18_bfloat16_19.dtype)
    # matmul: ('CHUNK', 'BLK_K'), ('BLK_K', 'BLK_V') => ('CHUNK', 'BLK_V')
    matmul_2_21 = tl.dot(blk_q_0, blk_chunk_state_4_bfloat16_11).to(
    blk_q_0.dtype)
    # exp: ('CHUNK',) => ('CHUNK',)
    exp_1_22 = tl.exp(blk_g_3)
    # unsqueeze: ('CHUNK',) => ('CHUNK', 1)
    unsqueeze_2_23 = exp_1_22[:, None]
    # mul: ('CHUNK', 'BLK_V'), ('CHUNK', 1) => ('CHUNK', 'BLK_V')
    mul_1_24 = matmul_2_21 * unsqueeze_2_23
    # add: ('CHUNK', 'BLK_V'), ('CHUNK', 'BLK_V') => ('CHUNK', 'BLK_V')
    add_25 = matmul_1_20 + mul_1_24
    # mul: ('CHUNK', 'BLK_V'), () => ('CHUNK', 'BLK_V')
    mul_2_26 = add_25 * scale
    # store => ('CHUNK', 'BLK_V')
    # assume output layout: [T, H, V]
    out_0_ty = out_0.dtype.element_ty
    ptr_out_0 = tl.make_block_ptr(out_0, (T, V,), (H * V, 1,), (i_c * CHUNK, i_v * BLK_V,), (CHUNK, BLK_V,), (1, 0))  # fmt: skip
    tl.store(ptr_out_0, mul_2_26.to(out_0_ty), boundary_check=(1, 0))

def launch_merge_mode(
    chunk_decay: torch.Tensor,
    v: torch.Tensor,
    k: torch.Tensor,
    q: torch.Tensor,
    g: torch.Tensor,
    prev_rank_state_sum: torch.Tensor,
    chunk_state: torch.Tensor,
    scale,
    cu_seqlens: torch.IntTensor,
    cached_results: dict,
    fuse_sp_update: bool,
) -> torch.Tensor:
    T, H, K, V = *k.shape, v.shape[-1]
```

```
    if scale is None:
        scale = k.shape[-1] ** -0.5

    # TODO: now fix chunk_size to 64
    CHUNK = 64
    chunk_indices = prepare_chunk_indices(cu_seqlens, CHUNK)
    chunk_offsets = prepare_chunk_offsets(cu_seqlens, CHUNK)
    chunk_offsets_ini = prepare_chunk_offsets_with_ini(cu_seqlens, CHUNK)
    NS, NC = len(cu_seqlens) - 1, chunk_indices.shape[0]
    NC_WITH_INI = NC + NS
    out = torch.empty_like(v)
    use_aot = os.environ.get('FORGE_USE_AOT', '0')
    FORGE_USE_AOT = True if use_aot.lower() in ['1', 'true', 'yes'] else
    None

    def alloc_fn(size: int, alignment: int, stream: int):
        return torch.empty(size, device='cuda', dtype=torch.int8)

    triton.set_allocator(alloc_fn)

    def grid(meta):
        BLK_K = meta['BLK_K']
        BLK_V = meta['BLK_V']
        NUM_BLK_K = (K + BLK_K - 1) // BLK_K
        NUM_BLK_V = (V + BLK_V - 1) // BLK_V
        NUM_BLK_KV = NUM_BLK_K * NUM_BLK_V
        H_MUL_NS = H * NS
        return (NUM_BLK_KV, H, NC,)

    if FORGE_USE_AOT is None:
        merge_mode_kernel[grid](
            chunk_decay=chunk_decay,
            stride_d0_nc_with_ini_chunk_decay=H,
            v=v,
            k=k,
            q=q,
            g=g,
            prev_rank_state_sum=prev_rank_state_sum,
            stride_d0_ns_prev_rank_state_sum=H*K*V,
            chunk_state=chunk_state,
            scale=scale,
            cu_seqlens=cu_seqlens,
            H=H,
            NS=NS,
            K=K,
            V=V,
            CHUNK=CHUNK,
            H_MUL_NS=H * NS,
            out_0=out,
            chunk_indices=chunk_indices,
            NC=NC,
            FUSE_SP_STATE_UPDATE=fuse_sp_update,
        )

    else:

        from forge.aot_utils import forge_aot_ops

        algo_info = forge_aot_ops.merge_mode_kernel__triton_algo_info_t()
        for _k, _v in get_merge_mode_kernel_info(NS, T, H, K, V).items():
            setattr(algo_info, _k, _v)
        forge_aot_ops.merge_mode_kernel(
            0,  # torch.cuda.current_stream().cuda_stream,
            q.data_ptr(),  # q
```

```
            k.data_ptr(),  # k
            v.data_ptr(),  # v
            g.data_ptr(),  # g
            chunk_state.data_ptr(),  # chunk_state
            scale,  # scale
            prev_rank_state_sum.data_ptr() if prev_rank_state_sum else 0,  # prev_rank_state_sum
            H*K*V,  # stride_d0_ns_prev_rank_state_sum
            chunk_decay.data_ptr() if chunk_decay else 0,  # chunk_decay
            H,  # stride_d0_nc_with_ini_chunk_decay
            out.data_ptr(),  # out_0
            cu_seqlens.data_ptr(),  # cu_seqlens
            NS,  # NS
            H * NS,  # H_MUL_NS
            chunk_indices.data_ptr(),  # chunk_indices
            NC,  # NC
            algo_info,
        )

    cached_results.update({})
    return out

def fused_op():
    cached_results = {}
    # AUTO: precompute decay outside
    g_cumsum = chunk_local_cumsum(g[None, ...], 64, cu_seqlens=cu_seqlens).squeeze(0)
    updated_states = launch_fuse_chunk_decay(
        g=g_cumsum,
        prev_s=prev_s,
        v=v,
        k=k,
        cu_seqlens=cu_seqlens,
        cached_results=cached_results,
        dist_scan=dist_scan,
        lazy_update=lazy_update,
    )
    chunk_state = updated_states['chunk_state']
    prev_rank_state_sum = updated_states.get('prev_rank_state_sum', None)
    chunk_decay = updated_states.get('chunk_decay', None)
    o = launch_merge_mode(
        chunk_decay=chunk_decay,
        v=v,
        k=k,
        q=q,
        g=g_cumsum,
        prev_rank_state_sum=prev_rank_state_sum,
        chunk_state=chunk_state,
        scale=scale,
        cu_seqlens=cu_seqlens,
        cached_results=cached_results,
        fuse_sp_update=fuse_sp_update,
    )
    return o, chunk_state
```

Listing 2: Generated Scalar GLA kernel by Forge

G.2 Implementation for DeltaNet

```
def chunk_mode_deltanet(k: Tensor, v: Tensor, b: Tensor) -> Tensor:
    I = forge.identity(k, k.size(0))
    # Note: forge handles the specific fp32 accumulation
    T = (I + forge.matmul_out_fp32(k * b[..., None], k.T).tril(-1)).inverse().to(k.dtype)
```

```
    U = T @ (v * b[..., None])
    W = T @ (k * b[..., None])
    # Explicitly cache results for reuse in other phases
    forge.cache_result(U, "u")
    forge.cache_result(W, "w")
    S = k.T @ U
    return S

def decay_mode_deltanet(prev_s: Tensor, k: Tensor, w: Tensor, chunk_state
    : Tensor) -> Tensor:
    """
    Inter-Chunk State Propagation
    """
    # Calculate decay matrix based on cached W
    decay = forge.identity(k, k.size(1)) - k.T @ w
    return decay @ prev_s.to(decay.dtype) + chunk_state

def merge_mode_deltanet(q: Tensor, k: Tensor, u: Tensor, w: Tensor,
    chunk_state: Tensor, scale: Tensor) -> Tensor:
    """
    Output Merging
    Note: 'u' and 'w' are automatically injected from the cached results
    """
    new_v = u - w @ chunk_state
    return ((q @ k.T).tril(0) @ new_v + q @ chunk_state) * scale
```

Listing 3: Forge Implementation of DeltaNet with Intermediate Result Caching