

000 001 002 003 004 005 FORGE: COMPILING A UNIFIED ABSTRACTION INTO 006 SCALABLE KERNELS FOR LINEAR ATTENTION 007 008 009

010 **Anonymous authors**
011 Paper under double-blind review
012
013
014
015
016
017
018
019
020
021
022
023
024
025
026
027
028

029 ABSTRACT 030

031 The quadratic complexity of softmax attention poses a major bottleneck for long-
032 context modeling, motivating a surge of linear attention variants with linear com-
033 plexity. Unlike softmax attention, which benefits from optimized kernels, lin-
034 ear attention lacks general-purpose, hardware-efficient support and scalable dis-
035 tributed implementations. We introduce Forge, a domain-specific compiler that
036 automates the generation of high-performance, scalable kernels for a wide range
037 of linear attention models directly from high-level PyTorch code. At its core,
038 Forge employs an intuitive programming abstraction that decomposes any linear
039 attention algorithm into three canonical phases: intra-chunk computation, inter-
040 chunk state propagation, and output merging. This unified abstraction enables
041 Forge to perform domain-specific optimizations, automatically generating kernels
042 that fuse computation and communication at a fine-grained tile level and elim-
043 inating host synchronization. Our evaluation demonstrates that Forge combines
044 programmability with performance: a wide range of linear attention variants can
045 be implemented in just a few dozen lines of code, while the generated kernels
046 deliver 1.01x-4.9x the performance of state-of-the-art expert-optimized library and
047 scale with near-linear efficiency on scalar gated linear attention to 16 million to-
048 kens on 128 GPUs, surpassing the state-of-the-art distributed baseline by up to
049 7.2x.

050 1 INTRODUCTION 051

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

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

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

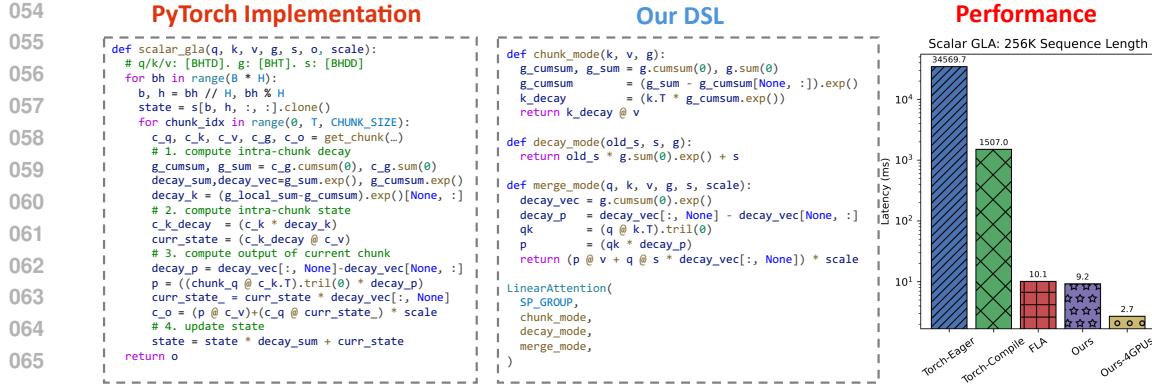


Figure 1: Comparison between PyTorch and our proposed DSL in writing scalar gated linear attention (data type conversion is omitted for simplicity). And the performance comparison of different approaches. Test shape: 32 heads with 128 head dimension. Torch-Eager fails to parallelize the computation, 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: **1 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. **2 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. **3 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 $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_L]^\top \in \mathbb{R}^{L \times d}$, 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):

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

Equation 2 is the matrix form of Equation 1 where $\mathbf{Q} := [\mathbf{q}_1, \dots, \mathbf{q}_L]^\top$, $\mathbf{K} := [\mathbf{k}_1, \dots, \mathbf{k}_L]^\top$, $\mathbf{V} := [\mathbf{v}_1, \dots, \mathbf{v}_L]^\top \in \mathbb{R}^{L \times d}$ 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 $\mathcal{O}(L^2d)$ 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 $\mathcal{O}(Ld^2)$. The recurrence form is expressed as:

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

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:

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

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).

162 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
 163 after processing chunk i . The chunk-wise formulation separates computation into intra-chunk and
 164 inter-chunk two parts, and then merge both together to get the final output as shown in Figure 2.
 165

166

$$\mathbf{S}_{[i]} = \mathbf{S}_{[i-1]} + \sum_{j=(i-1)C}^{iC} \mathbf{k}_j^\top \mathbf{v}_j \quad (5)$$

167

168

169

170

171

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

172

173

174

175

176

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

178 The inter chunk item can be viewed as readout memory from start of the sequence to the start of
 179 current chunk while the intra chunk item is processing the information in current chunk. When
 180 chunk size C is set to sequence length L , it becomes the fully parallel form as in Equation 4. The
 181 chunk size is a tradeoff between parallelism and FLOPs.
 182

183 2.3 DSLS AND DOMAIN SPECIFIC COMPILER

184 A domain-specific language (DSL) trades generality for performance by restricting program ex-
 185 pressivity to a particular domain, creating opportunities for targeted optimization. Deep learning
 186 compilers like TVM (Chen et al., 2018), ThunderKitten (Spector et al., 2024), TileLang (Wang
 187 et al., 2025) and `torch.compile` (Ansel et al., 2024) exemplify this approach: by operating on a
 188 constrained set of primitives (i.e. `tir` in TVM and `aten` operators in PyTorch), they can systematically
 189 explore a focused design space to apply transformations like operator fusion and loop tiling, thereby
 190 automatically generating high-performance code from high-level descriptions.
 191

192 This paradigm is particularly effective for operations that possess rich computational structure but
 193 high implementation complexity. By exposing domain-relevant abstractions, a DSL allows develop-
 194 ers to specify *what* to compute, while delegating the complex details of *how* to execute it efficiently
 195 to the compiler. This separation of concerns is key to enabling automated, domain-specific opti-
 196 mizations without burdening the user with low-level hardware details.
 197

198 Applied to the domain of linear attention, an effective DSL must therefore provide abstractions
 199 that are expressive enough to capture the diverse patterns found in various state update rules and
 200 parallel scan formulations. Simultaneously, its compiler must be able to recognize these common
 201 patterns and systematically generate optimized kernels for them, bridging the gap between high-
 202 level algorithmic design and performance, hardware-aware code. Because chunked parallel forms
 203 are complex to implement and offer opportunities to fully exploit hardware, this work focuses on
 204 the prefill phase of linear attention (used in inference or the forward pass of training). The backward
 205 pass can be implemented in a similar manner (Qin et al., a).
 206

207 3 FORGE

208 In this section, we propose a unified abstraction of diverse linear attention variants. This abstraction
 209 enables programmers to easily express linear attention semantics without worrying about implemen-
 210 tation details and kernel performance.
 211

212 3.1 PROGRAMMING ABSTRACTION

213 Despite the numerous linear attention variants designed by researchers, we unify these variants into
 214 three commonly shared phases based on chunk-wise parallel form introduced in subsection 2.2. 1
 215 Intra-Chunk Computation. The first phase computes a local state within each chunk of the input

216 Table 1: A comparison of representative linear attention variants that can be easily mapped to our
 217 three-phase abstraction, including HGRN (Qin et al., 2023), RetNet (Sun et al., 2023), Mamba2 (Dao
 218 & Gu, 2024), GLA (Yang et al., 2023), and GDN (Yang et al., 2024a). Despite diverse state
 219 types (vector vs. matrix) and decay mechanisms (element-wise product vs. matrix multiplication).
 220 v_t, k_t, q_t are value, key and query projections; $\alpha_t, \beta_t, r_t, i_t$ are gates; \odot is the Hadamard product.

Model	Update rule	Read-out	State + Decay type
HGRN	$h_t = \alpha_t \odot h_{t-1} + (1 - \alpha_t) \odot v_t$	$o_t = h_t \odot q_t$	vector + data-dependent vector
RetNet	$S_t = \gamma S_{t-1} + v_t k_t^\top$	$o_t = S_t q_t$	matrix + data-independent scalar
Mamba2	$S_t = \gamma_t S_{t-1} + v_t k_t^\top$	$o_t = S_t q_t$	matrix + data-dependent scalar
GLA	$S_t = S_{t-1} \odot (1\alpha_t^\top) + v_t k_t^\top$	$o_t = S_t q_t$	matrix + data-dependent vector
GDN	$S_t = \alpha_t S_{t-1} (I - \beta_t k_t k_t^\top) + \beta_t v_t k_t^\top$	$o_t = S_t q_t$	matrix + data-dependent matrix

221 sequence. In this stage, computation across different chunks is embarrassingly parallel, as there are
 222 no data dependencies between them. Each chunk is processed independently, transforming its
 223 sequence of inputs into a relative state summary. ② Inter-Chunk State Propagation. The second phase
 224 addresses the dependencies between chunks. To compute the correct global state at the beginning of
 225 each chunk, the state summaries from all preceding chunks must be accumulated. For instance, in
 226 the case of vanilla linear attention, this propagation corresponds to a prefix sum (scan) operation, as
 227 shown in Equation 5. This phase is inherently sequential due to the temporal dependencies between
 228 chunk states. And cross-device communication happen in this phase. ③ Merging and Output Gen-
 229 eration. The final phase merges the results of the intra-chunk and inter-chunk computations. Here,
 230 operations are once again parallel across chunks. Each chunk utilizes the global state propagated
 231 from Phase 2 and its local inputs to compute its final output sequence.

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

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

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

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

246 3.2 COMPILATION AND CODE GENERATION

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

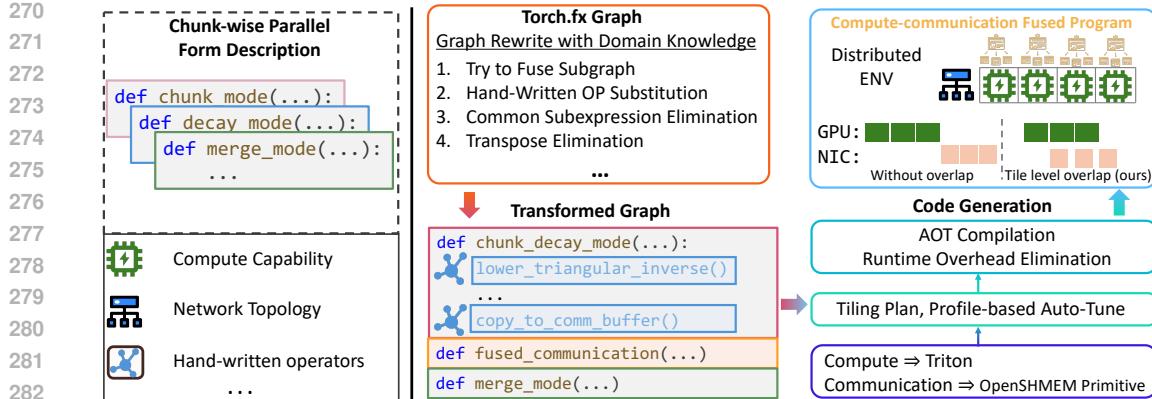


Figure 3: An overview of the Forge compilation pipeline. Our compiler ingests a high-level description of a linear attention algorithm, specified using our DSL (`chunk_mode`, `decay_mode`, `merge_mode`). This description is traced into a graph structure, which then undergoes a series of domain-specific 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 additionally 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

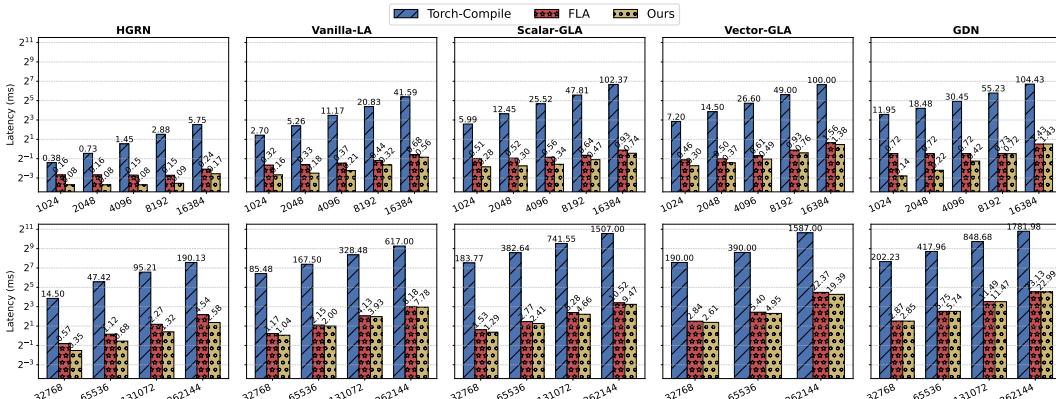


Figure 4: Latency comparison of different linear attention variants under varying sequence lengths on a single H100 GPU. Each subplot corresponds to one model, with the top row showing short to medium sequences (1K–16K) and the bottom row showing long sequences (32K–256K).

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 BatchSize = 1, NumHeads = 32, and 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

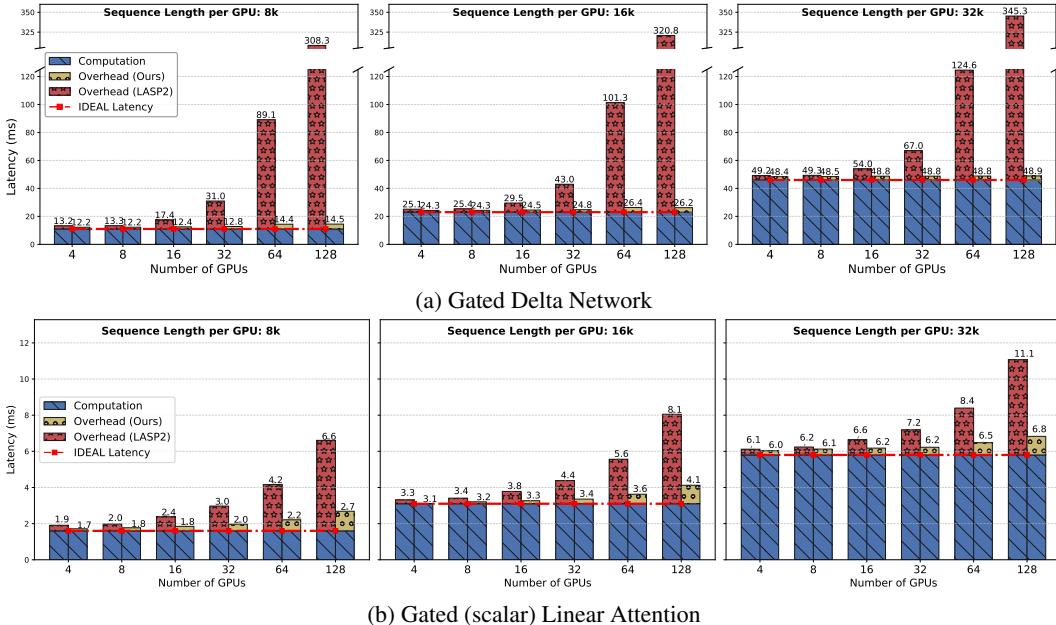


Figure 5: Weak scaling performance comparison for GDN and GLA models. Both figures show latency breakdown across a varying number of GPUs with a fixed sequence length per GPU.

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 BatchSize = 4, NumHeads = 32, 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 (207 μ s) is over 4.4 times the actual kernel execution time (47 μ s). 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-

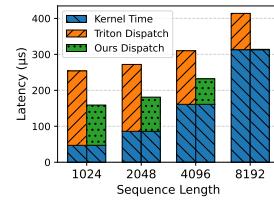


Figure 6: Execution time decomposition.

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

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: ① 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 $i + 1$. ② 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

486 achieves both high performance and broad applicability across modern linear attention models. We
487 hope this work will accelerate the development of new architectures and inspire further research at
488 the intersection of deep learning algorithms and domain-specific compilation.
489

490

491

492

493

494

495

496

497

498

499

500

501

502

503

504

505

506

507

508

509

510

511

512

513

514

515

516

517

518

519

520

521

522

523

524

525

526

527

528

529

530

531

532

533

534

535

536

537

538

539

540 7 REPRODUCIBILITY STATEMENT
541

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

548 REFERENCES
549

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

553 Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky,
554 Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. Pytorch 2: Faster machine learning
555 through dynamic python bytecode transformation and graph compilation. In *Proceedings of the*
556 *29th ACM International Conference on Architectural Support for Programming Languages and*
557 *Operating Systems, Volume 2*, pp. 929–947, 2024.

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

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

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

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

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

575 Tri Dao and Albert Gu. Transformers are SSMs: Generalized models and efficient algorithms
576 through structured state space duality. In *International Conference on Machine Learning (ICML)*,
577 2024.

578 Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-
579 efficient exact attention with io-awareness. *Advances in neural information processing systems*,
580 35:16344–16359, 2022.

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

586 Juechu Dong, Boyuan Feng, Driss Guessous, Yanbo Liang, and Horace He. Flex attention: A
587 programming model for generating optimized attention kernels. *arXiv preprint arXiv:2412.05496*,
588 2024.

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

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

594 Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):
 595 1735–1780, 1997.

596

597 William Hu, Drew Wadsworth, Sean Siddens, Stanley Winata, Daniel Y. Fu, Ryann Swann, Muham-
 598 mad Osama, Christopher Ré, and Simran Arora. Hipkittens: Fast and furious amd kernels, 2025.
 599 URL <https://arxiv.org/abs/2511.08083>.

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

602

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

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

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

614

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

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

620

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

624

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

628

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

632

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

635

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

638

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

642

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

646

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

650

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

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

650

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

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

655

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

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

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

665

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

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

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

674

675 Songlin Yang and Yu Zhang. Fla: A triton-based library for hardware-efficient implementa-
 676 tions of linear attention mechanism, January 2024. URL <https://github.com/fla-org/flash-linear-attention>.

677

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

680

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

683

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

686

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

690

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

694

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

698

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

702

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

702 **A USAGE OF LLMs**
703

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

710
711 **B PARALLISIM SCHEUDLER**
712

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

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

720
721 **B.1 NOTATION**
722

723 The vanilla linear attention is defined by the recurrence:

$$724 \quad \mathbf{S}_t = \mathbf{S}_{t-1} + \mathbf{k}_t \mathbf{v}_t^T \quad (8)$$

$$725 \quad \mathbf{o}_t = \mathbf{q}_t \mathbf{S}_t \quad (9)$$

726 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
727 $\mathbf{S}_t \in \mathbb{R}^{D_k \times D_v}$ is the state matrix. We consider batched inputs $\mathbf{Q}, \mathbf{K} \in \mathbb{R}^{B \times T \times H \times D_k}$ and
728 $\mathbf{V} \in \mathbb{R}^{B \times T \times H \times D_v}$. The sequence of length T is divided into N_C chunks of size C , such that
729 $T = N_C \times C$. Let N_{SM} be the number of Streaming Multiprocessors (SMs) on the target GPU,
730 representing its parallel execution capacity.

731
732 **B.2 STRATEGY 1: DECOUPLED THREE-PHASE EXECUTION**
733

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

736
737 **Parallelism Analysis.** The degree of parallelism varies significantly between phases.

- 739 • **Phase 1 (Intra-Chunk State Computation):** Computations for each chunk are indepen-
740 dent. The total number of parallel tasks is $B \times H \times N_C$. This phase exhibits the highest
741 degree of parallelism.
- 742 • **Phase 2 (Inter-Chunk State Propagation):** The prefix sum (scan) is sequential across the
743 N_C dimension. Parallelism is limited to the batch (B) and head (H) dimensions. Further
744 parallelism, which we denote as P_{state} , can be achieved by partitioning the state matrix
745 $\mathbf{S} \in \mathbb{R}^{D_k \times D_v}$ and processing its partitions on different thread blocks. The total number of
746 parallel tasks is $B \times H \times P_{state}$.
- 747 • **Phase 3 (Merge):** Similar to Phase 1, the merge operation is independent across chunks,
748 offering a high degree of parallelism with $B \times H \times N_C$ tasks.

749
750 **Memory Access Analysis.** The defining characteristic of this strategy is the materialization of
751 intermediate states in global memory (GMEM) between phases.

- 752 • **Phase 1:** Reads \mathbf{K} and \mathbf{V} from GMEM. Writes the intermediate, chunk-local states $\mathbf{S}' \in$
753 $\mathbb{R}^{B \times H \times N_C \times D_k \times D_v}$ back to GMEM.

$$754 \quad \text{GMEM Traffic}_{P1} = \underbrace{BHT(D_k + D_v)}_{\text{Reads}} + \underbrace{BHN_CD_kD_v}_{\text{Writes}} \quad (10)$$

756 • **Phase 2:** Reads the $B \times H \times N_C$ chunk-local states from GMEM, performs the scan, and
 757 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.
 758

759
$$\text{GMEM Traffic}_{P2} = \underbrace{BHN_CD_kD_v}_{\text{Reads}} + \underbrace{BHN_CD_kD_v}_{\text{Writes}} = 2BHN_CD_kD_v \quad (11)$$

 760

761 The total GMEM traffic introduced between Phase 1 and Phase 2 is $3 \times B \times H \times N_C \times D_k D_v$.
 762

763 **B.3 STRATEGY 2: FUSED INTRA- AND INTER-CHUNK EXECUTION**

764 This strategy fuses Phase 1 and Phase 2, computing the local state of a chunk and immediately uses
 765 it to update the global state, all within on-chip memory.
 766

767 **Parallelism Analysis.** By fusing the phases, the execution is constrained by the most sequential
 768 part, which is the inter-chunk scan. Therefore, the maximum number of parallel tasks is identical to
 769 that of Phase 2 in the decoupled strategy: $B \times H \times P_{state}$. The high parallelism across the chunk
 770 dimension (N_C) is sacrificed.
 771

772 **Memory Access Analysis.** The primary advantage of this strategy is the elimination of the inter-
 773 intermediate GMEM traffic.
 774

775 • **Fused Phase (1+2):** Each of the $B \times H \times P_{state}$ parallel tasks loads its corresponding
 776 slice of \mathbf{K} and \mathbf{V} from GMEM chunk by chunk. The intermediate, chunk-local states are
 777 generated and accumulated entirely within SMEM. The only state written to GMEM is the
 778 final updated global state for each chunk, which is required by the Merge phase.
 779

780
$$\text{GMEM Traffic}_{P1+P2} = \underbrace{BHT(D_k + D_v)}_{\text{Reads}} + \underbrace{BHN_CD_kD_v}_{\text{Writes}} \quad (12)$$

 781

782 Compared to Strategy 1, this approach saves $2 \times B \times H \times N_C \times D_k D_v$ worth of GMEM traffic,
 783 which is the cost of one full read and one full write of the intermediate state tensor.
 784

785 **B.4 HEURISTIC SCHEDULING ALGORITHM**

786 The choice between the Decoupled and Fused strategies presents a classic trade-off between parallelism
 787 and memory locality.
 788

789 • **Strategy 1 (Decoupled)** is favored when the GPU has a high degree of parallelism (N_{SM}
 790 is large) that is not saturated by the task parallelism of Strategy 2 ($B \times H \times P_{state}$). The
 791 performance gain from launching more parallel tasks in Phase 1 and 3 must outweigh the
 792 latency incurred by the extra GMEM I/O.
 793

794 • **Strategy 2 (Fused)** is favored when the task parallelism of the scan ($B \times H \times P_{state}$)
 795 is already sufficient to saturate the GPU’s SMs, or when the cost of reading/writing the
 796 intermediate states is the dominant performance bottleneck.
 797

798 We can formulate a simple heuristic to guide this choice as shown in Algo 1.
 799

800 In practice, we set T_{comp_chunk} to 0 to reduce the runtime overhead and omit the micro-benchmark
 801 effort. The insight is that most linear attention variants is memory bound instead of compute-
 802 intensive.
 803

804 **C COMPUTATIONAL PATTERNS AND MODEL COVERAGE**

805 This appendix elaborates on the claim that our implemented kernels for a few representative models
 806 can support a much broader range of linear attention variants. The mapping from our representative
 807 implementations to the computational patterns and the models they cover is detailed below and
 808 summarized in Table 3.
 809

810 **Algorithm 1** Parallelism Scheduler)

811 **Require:** Shapes B, T, H, d_k, d_v , chunk size C , tile sizes t_k, t_v

812 **Require:** Device params: memory bandwidth BW (bytes/s), P_{\max} (hardware concurrency cap)

813 **Require:** Cost param: $T_{\text{comp.chunk}}$ (measured per-chunk compute time in seconds)

814 **Require:** Thresholds: P_{\min} (saturation threshold, tasks), element size s (bytes)

815 **Ensure:** Return strategy $\in \{\text{FUSED}, \text{DECOUPLED}\}$

816 1: $N_c \leftarrow \lceil T/C \rceil$

817 2: $P_{\text{state}} \leftarrow \lceil d_k/t_k \rceil \cdot \lceil d_v/t_v \rceil$

818 3: $P_{\text{dec}} \leftarrow B \cdot H \cdot N_c \cdot P_{\text{state}}$

819 4: $P_{\text{fused}} \leftarrow B \cdot H \cdot P_{\text{state}}$

820 5: $\text{GMEM}_{\text{dec}} \leftarrow (2C(d_k + d_v) + 5d_kd_v) \cdot s$

821 6: $\text{GMEM}_{\text{fused}} \leftarrow (2C(d_k + d_v) + 4d_kd_v) \cdot s$

822 ▷ Fast path: if fused already provides enough parallelism, prefer fused to save GMEM IO

823 7: **if** $P_{\text{fused}} \geq P_{\min}$ **then**

824 **return** **FUSED**

825 **end if**

826 ▷ Otherwise estimate per-chunk runtime under each mapping

827 10: $\tilde{P}_{\text{dec}} \leftarrow \min(P_{\text{dec}}, P_{\max})$

828 11: $\tilde{P}_{\text{fused}} \leftarrow \min(P_{\text{fused}}, P_{\max})$

829 12: $\hat{T}_{\text{dec}} \leftarrow \frac{\text{GMEM}_{\text{dec}}}{BW \cdot \tilde{P}_{\text{dec}}} + \frac{T_{\text{comp.chunk}}}{\tilde{P}_{\text{dec}}}$

830 13: $\hat{T}_{\text{fused}} \leftarrow \frac{\text{GMEM}_{\text{fused}}}{BW \cdot \tilde{P}_{\text{fused}}} + \frac{T_{\text{comp.chunk}}}{\tilde{P}_{\text{fused}}}$

831 14: **if** $\hat{T}_{\text{fused}} \leq \hat{T}_{\text{dec}}$ **then**

832 **return** **FUSED**

833 15: **else**

834 **return** **DECOUPLED**

835 16: **end if**

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

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

866 Representative Kernel	867 Covered Models (Examples)
868 HGRN	HGRN (Qin et al., 2023), Hawk (RG-LRU) (De et al., 2024)
869 Scalar-GLA	RetNet (Sun et al., 2023), Mamba-2 (Dao & Gu, 2024), Lightning 870 Attention (Qin et al., 2024)
871 Vector-GLA	GLA (Yang et al., 2023), HGRN-2 (Qin et al., b), RWKV-6 (Peng 872 et al., 2023)
873 GDN	DeltaNet (Yang et al., 2024b), GatedDeltaNet (Yang et al., 2024a), 874 DeltaProduct (Siems et al., 2025)

876 D DISCUSSION ON EXTENSIBILITY AND SUSTAINABILITY

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

883 D.1 ALGORITHMIC EXPRESSIVENESS: GROUNDED IN ASSOCIATIVITY

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

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

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

- 896 • **The Delta Rule:** Despite involving data-dependent updates (e.g., $h_t = h_{t-1} + \beta_t(v_t -$
 897 $h_{t-1})$), the Delta Rule preserves the associative structure within local chunks, allowing
 898 Forge to effectively fuse operations.
- 899 • **Element-wise Decay:** Varying decay rates (as seen in Vector-Gated GLA) are fully sup-
 900 ported, as the element-wise multiplication distributes over addition, maintaining the scan
 901 property.

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

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

915 D.2 HARDWARE SUSTAINABILITY: HIERARCHICAL DECOUPLING

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

918 **Layered Compilation.** Forge operates as a high-level graph compiler rather than a low-level as-
 919 sembler. It does not directly emit hardware-specific assembly (ISA). Instead, it lowers the PyTorch-
 920 based algorithmic description into an intermediate kernel DSL. In our current implementation, we
 921 target Triton, effectively delegating low-level complexities such as register allocation, instruction
 922 scheduling, and tensor core management (e.g., WGMMA on Hopper) to the Triton compiler.
 923

924 **Backend Agnosticism.** This layered design makes Forge inherently adaptable. While the current
 925 backend is Triton, the architecture allows for retargeting the mapping layer to other emerging DSLs
 926 (e.g., ThunderKitten (Spector et al., 2024) or TileLang (Wang et al., 2025)). These DSLs provide
 927 better performance with the cost of hardware overfit (e.g. ThunderKittens target on NVidia GPU,
 928 HipKittens (Hu et al., 2025) target solely on AMD GPU while Triton support multiple hardware
 929 backend). Triton allows us to minimize the use of inline assembly (e.g. PTX on NVGPU), reserving
 930 it only for specific extensions not yet exposed by the intermediate representation.
 931

932 **Forward Compatibility.** Consequently, as hardware architectures evolve (e.g., the transition to
 933 NVIDIA Blackwell), Forge benefits automatically from updates made by the community to the in-
 934 termediate compiler stack. This ensures that user-level code remains stable and performant without
 935 requiring modification, solving the maintenance bottleneck often associated with hand-written ker-
 936 nels.
 937

938 E DISCUSSION ON GENERALIZABILITY TO SOFTMAX ATTENTION

940 While Forge is explicitly designed for Linear Attention, the underlying mathematical associativity,
 941 which enables our optimization is shared by Softmax attention (exploited via the online softmax
 942 trick (Dao et al., 2022)). In this section, we analyze the theoretical feasibility of extending Forge to
 943 support Softmax attention, and the design rationale behind our decision to specialize in the Linear
 944 Attention domain.
 945

Extending the Forge abstraction to support Softmax-based mechanisms is theoretically feasible but
 946 would necessitate two primary modifications to the current three-phase abstraction:
 947

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

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

959 Supporting this would require introducing a native `rescale(output, old_stats,`
 960 `new_stats)` primitive into the Forge DSL.¹
 961

962 **Despite the feasibility of these extensions, we deliberately narrowed the scope of Forge to Lin-
 963 ear Attention to maximize expressiveness.**
 964

965 **The Tension between Universality and Specificity.** Constructing a "universal" DSL often neces-
 966 sitates a rigid abstraction that compromises the ability to model complex, domain-specific patterns.
 967 A recent framework, AttentionEngine (Chen et al., 2025), attempts to unify both Softmax and Linear
 968 Attention. However, to maintain this broad generality, its abstraction struggles to express advanced
 969 Linear Attention variants that feature intricate dependencies, such as the **Delta Rule** or **Vector-
 970 Gated GLA**. By specializing in Linear Attention, Forge avoids these constraints, supporting these
 971 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**, NumHeads = 32, and HeadDim = 128. The **SeqLen** in table means sequence length per GPU(global sequence length is equal to **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 \times 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 \times 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-

1026 ing 2. Forge enable pre-compiled kernel (AOT) and static dispatch at runtime. In this example, we
 1027 tune the kernel on $H=[1,4,8,16,32]$ and $D=[64, 128]$, so there are a lot of auto-generated code en-
 1028 coded pre-tuned information for dispatcher (our AOT optimization). Using Forge to implement this
 1029 kernel only involves 50+ lines of code (LOC), which the generated triton code and host launcher is
 1030 around 1200 LOC. Even without including the static dispatcher and host launcher in count, the gen-
 1031 erated Triton kernel alone has over 400 LOC. And this is only partially generated content, because
 1032 Forge also generates code for other parallel strategies.

1033 Then we demonstrate the implementation of DeltaNet using Forge in Listing 3. Note that
 1034 DeltaNet involves complex data dependencies where intermediate results computed in the chunk
 1035 phase (specifically U and W) are required in the merge phase. Forge introduces a primitive
 1036 `forge.cache_result` to explicit mark these tensors. The compiler then automatically handles
 1037 the memory layout and data movement to ensure these values are efficiently reused across phases
 1038 without redundant re-computation or manual memory management.

1039

1040 G.1 IMPLEMENTATION AND CODE GENERATION FOR SCALAR GLA

```

1041
1042 def chunk_mode_scalar_gla(k: Tensor, v: Tensor, g: Tensor) -> Tensor:
1043     """k: [C, K], v: [C, V], g: [C]"""
1044     g_cumsum = g.cumsum(dim=0)
1045     g_cumsum_last = g.sum(dim=0)
1046     g_cumsum = (g_cumsum_last - g_cumsum).unsqueeze(0).exp()
1047     k_fp32 = k.permute([1, 0]).to(g.dtype)
1048     k_decay = (k_fp32 * g_cumsum).to(v.dtype)
1049     chunk_state = k_decay @ v
1050     return chunk_state
1051
1052 def decay_mode_scalar_gla(prev_s: Tensor, chunk_state: Tensor, g: Tensor) -> Tensor:
1053     """g: [C]"""
1054     g_sum = g.sum(dim=0).exp()
1055     return prev_s * g_sum + chunk_state
1056
1057 def merge_mode_scalar_gla(
1058     q: Tensor,
1059     k: Tensor,
1060     v: Tensor,
1061     g: Tensor,
1062     chunk_state: Tensor,
1063     scale: Tensor,
1064 ) -> Tensor:
1065     """q: [C, K], k: [C, K], v: [C, V], chunk_state: [K, V]"""
1066     g_cumsum = g.cumsum(0)
1067     chunk_state = chunk_state.to(q.dtype)
1068     p = (q @ k.T).tril(0)
1069     p = (p * (g_cumsum[..., None] - g_cumsum[None, ...]).exp()).to(v.dtype)
1070     return (p @ v + q @ chunk_state * g_cumsum.exp() [..., None]) * scale
1071
1072 CONST_H = ConstExpr("H", H)
1073 CONST_K = ConstExpr("K", K)
1074 CONST_V = ConstExpr("V", V)
1075 meta = {
1076     "q": SymbTensor(["T", CONST_H, CONST_K], dtype=dtype),
1077     "k": SymbTensor(["T", CONST_H, CONST_K], dtype=dtype),
1078     "v": SymbTensor(["T", CONST_H, CONST_V], dtype=dtype),
1079     "g": SymbTensor(["T", CONST_H], dtype=torch.float32),
1080     "prev_s": SymbTensor(["NS", CONST_H, CONST_K, CONST_V], dtype=torch.float32),
1081     "chunk_state": SymbTensor(["NC", CONST_H, CONST_K, CONST_V], dtype=dtype),
1082     "scale": 1 / math.sqrt(K),
1083 }
```

```

1080
1081     LinearAttention(
1082         input_meta=meta,
1083         sp_group=pg if args.sp else None,
1084         enable_aot=args.aot,
1085         code_dir=args.dir,
1086         chunk_mode=chunk_mode_scalar_gla,
1087         decay_mode=decay_mode_scalar_gla,
1088         merge_mode=merge_mode_scalar_gla,
1089     )

```

Listing 1: Implementation of Scalar GLA in Forge

```

1090     fuse_chunk_decay_kernel_signature = (
1091         "*bf16:16, "
1092         "*bf16:16, "
1093         "*fp32, "
1094         "*bf16:16, "
1095         "*bf16:16, "
1096         "*fp32, "
1097         "*i32, "
1098         "i32, "
1099         "i32, "
1100         "%USE_INITIAL_STATE, "
1101         "%H, "
1102         "%K, "
1103         "%V, "
1104         "%CHUNK, "
1105         "%BLK_K, "
1106         "%BLK_V, "
1107         "%USE_TMA"
1108     )
1109
1110
1111     def get_fuse_chunk_decay_kernel_info(B: int, T: int, H: int, K: int, V: int):
1112         """Static dispatcher for fuse_chunk_decay_kernel. Auto-generated."""
1113         D = [K, V]
1114         key = (B, T, H, D)
1115         if key in (
1116             (1, 4096, 4, [64, 64]),
1117             (2, 1024, 4, [64, 64]),
1118         ):
1119             BLK_K = 32
1120             BLK_V = 32
1121             num_warp = 8
1122             num_ctas = 1
1123             num_stages = 3
1124             maxnreg = None
1125             elif key in (
1126                 (1, 1024, 1, [64, 64]),
1127                 (1, 1024, 1, [128, 128]),
1128                 (1, 1024, 4, [64, 64]),
1129                 (1, 1024, 4, [128, 128]),
1130                 (1, 1024, 8, [64, 64]),
1131                 (1, 1024, 8, [128, 128]),
1132                 (1, 1024, 16, [64, 64]),
1133                 (1, 1024, 32, [64, 64]),
1134                 (1, 2048, 1, [64, 64]),
1135                 (1, 2048, 1, [128, 128]),
1136                 (1, 2048, 4, [64, 64]),
1137                 (1, 2048, 4, [128, 128]),
1138                 (1, 2048, 8, [64, 64]),
1139                 (1, 2048, 8, [128, 128]),
1140                 (1, 2048, 16, [64, 64]),
1141             ):

```

```

1134 53      (1, 2048, 32, [64, 64]),  

1135 54      (1, 4096, 1, [64, 64]),  

1136 55      (1, 4096, 1, [128, 128]),  

1137 56      (1, 4096, 4, [128, 128]),  

1138 57      (1, 4096, 8, [64, 64]),  

1139 58      (1, 4096, 8, [128, 128]),  

1140 59      (1, 4096, 16, [64, 64]),  

1141 60      (1, 4096, 32, [64, 64]),  

1142 61      (1, 8192, 1, [64, 64]),  

1143 62      (1, 8192, 1, [128, 128]),  

1144 63      (1, 8192, 4, [64, 64]),  

1145 64      (1, 8192, 4, [128, 128]),  

1146 65      (1, 8192, 8, [64, 64]),  

1147 66      (1, 8192, 8, [128, 128]),  

1148 67      (1, 8192, 16, [64, 64]),  

1149 68      (1, 8192, 32, [64, 64]),  

1150 69      (1, 16384, 1, [64, 64]),  

1151 70      (1, 16384, 1, [128, 128]),  

1152 71      (1, 16384, 4, [64, 64]),  

1153 72      (1, 16384, 4, [128, 128]),  

1154 73      (1, 16384, 8, [64, 64]),  

1155 74      (1, 16384, 8, [128, 128]),  

1156 75      (1, 16384, 16, [64, 64]),  

1157 76      (1, 16384, 32, [64, 64]),  

1158 77      (1, 32768, 1, [64, 64]),  

1159 78      (1, 32768, 1, [128, 128]),  

1160 79      (1, 32768, 4, [64, 64]),  

1161 80      (1, 32768, 4, [128, 128]),  

1162 81      (1, 32768, 8, [64, 64]),  

1163 82      (1, 32768, 8, [128, 128]),  

1164 83      (1, 32768, 16, [64, 64]),  

1165 84      (1, 32768, 32, [64, 64]),  

1166 85      (1, 65536, 1, [64, 64]),  

1167 86      (1, 65536, 1, [128, 128]),  

1168 87      (1, 65536, 4, [64, 64]),  

1169 88      (1, 65536, 4, [128, 128]),  

1170 89      (1, 65536, 8, [64, 64]),  

1171 90      (1, 65536, 8, [128, 128]),  

1172 91      (1, 65536, 16, [64, 64]),  

1173 92      (1, 65536, 32, [64, 64]),  

1174 93      (1, 131072, 1, [64, 64]),  

1175 94      (1, 131072, 1, [128, 128]),  

1176 95      (1, 131072, 4, [64, 64]),  

1177 96      (1, 131072, 4, [128, 128]),  

1178 97      (1, 131072, 8, [64, 64]),  

1179 98      (1, 131072, 8, [128, 128]),  

1180 99      (1, 131072, 16, [64, 64]),  

1181 100     (1, 131072, 32, [64, 64]),  

1182 101     (1, 262144, 1, [64, 64]),  

1183 102     (1, 262144, 1, [128, 128]),  

1184 103     (1, 262144, 4, [64, 64]),  

1185 104     (1, 262144, 4, [128, 128]),  

1186 105     (1, 262144, 8, [64, 64]),  

1187 106     (1, 262144, 8, [128, 128]),  

1188 107     (1, 262144, 16, [64, 64]),  

1189 108     (1, 262144, 32, [64, 64]),  

1190 109     (2, 1024, 1, [64, 64]),  

1191 110     (2, 1024, 1, [128, 128]),  

1192 111     (2, 1024, 4, [128, 128]),  

1193 112     (2, 1024, 8, [64, 64]),  

1194 113     (2, 1024, 16, [64, 64]),  

1195 114     (2, 2048, 1, [64, 64]),  

1196 115     (2, 2048, 1, [128, 128]),  

1197 116     (2, 2048, 4, [64, 64]),  

1198 117     (2, 2048, 4, [128, 128]),

```

```

1188      118      (2, 2048, 8, [64, 64]),
1189      119      (2, 2048, 16, [64, 64]),
1190      120      (2, 4096, 1, [64, 64]),
1191      121      (2, 4096, 1, [128, 128]),
1192      122      (2, 4096, 4, [64, 64]),
1193      123      (2, 4096, 4, [128, 128]),
1194      124      (2, 4096, 8, [64, 64]),
1195      125      (2, 4096, 16, [64, 64]),
1196      126      (2, 8192, 1, [64, 64]),
1197      127      (2, 8192, 1, [128, 128]),
1198      128      (2, 8192, 4, [64, 64]),
1199      129      (2, 8192, 4, [128, 128]),
1200      130      (2, 8192, 8, [64, 64]),
1201      131      (2, 8192, 16, [64, 64]),
1202      132      (2, 16384, 1, [64, 64]),
1203      133      (2, 16384, 1, [128, 128]),
1204      134      (2, 16384, 4, [64, 64]),
1205      135      (2, 16384, 4, [128, 128]),
1206      136      (2, 16384, 8, [64, 64]),
1207      137      (2, 16384, 16, [64, 64]),
1208      138      (2, 32768, 1, [64, 64]),
1209      139      (2, 32768, 1, [128, 128]),
1210      140      (2, 32768, 4, [64, 64]),
1211      141      (2, 32768, 4, [128, 128]),
1212      142      (2, 32768, 8, [64, 64]),
1213      143      (2, 32768, 16, [64, 64]),
1214      144      (2, 65536, 1, [64, 64]),
1215      145      (2, 65536, 1, [128, 128]),
1216      146      (2, 65536, 4, [64, 64]),
1217      147      (2, 65536, 4, [128, 128]),
1218      148      (2, 65536, 8, [64, 64]),
1219      149      (2, 65536, 16, [64, 64]),
1220      150      (2, 131072, 1, [64, 64]),
1221      151      (2, 131072, 1, [128, 128]),
1222      152      (2, 131072, 4, [64, 64]),
1223      153      (2, 131072, 4, [128, 128]),
1224      154      (2, 131072, 8, [64, 64]),
1225      155      (2, 131072, 16, [64, 64]),
1226      156      ) :
1227      157      BLK_K = 32
1228      158      BLK_V = 32
1229      159      num_warp = 8
1230      160      num_ctas = 1
1231      161      num_stages = 4
1232      162      maxnreg = None
1233      163      elif key in (
1234      164      (1, 1024, 32, [128, 128]),
1235      165      (1, 2048, 32, [128, 128]),
1236      166      (1, 4096, 32, [128, 128]),
1237      167      (1, 8192, 32, [128, 128]),
1238      168      (1, 16384, 32, [128, 128]),
1239      169      (1, 32768, 32, [128, 128]),
1240      170      (1, 65536, 32, [128, 128]),
1241      171      (1, 131072, 32, [128, 128]),
1242      172      (1, 262144, 32, [128, 128]),
1243      173      (2, 1024, 32, [128, 128]),
1244      174      (2, 2048, 16, [128, 128]),
1245      175      (2, 2048, 32, [128, 128]),
1246      176      (2, 4096, 16, [128, 128]),
1247      177      (2, 4096, 32, [128, 128]),
1248      178      (2, 8192, 32, [128, 128]),
1249      179      (2, 16384, 16, [128, 128]),
1250      180      (2, 16384, 32, [128, 128]),
1251      181      (2, 32768, 32, [128, 128]),
1252      182      (2, 131072, 16, [128, 128]),

```

```

1242 183 ) :
1243 184     BLK_K = 32
1244 185     BLK_V = 64
1245 186     num_warps = 4
1246 187     num_ctas = 1
1247 188     num_stages = 3
1248 189     maxnreg = None
1249 190     elif key in (
1250 191         (2, 1024, 16, [128, 128]),
1251 192     ) :
1252 193         BLK_K = 32
1253 194         BLK_V = 64
1254 195         num_warps = 8
1255 196         num_ctas = 1
1256 197         num_stages = 3
1257 198         maxnreg = None
1258 199     elif key in (
1259 200         (1, 1024, 16, [128, 128]),
1260 201         (1, 2048, 16, [128, 128]),
1261 202         (1, 4096, 16, [128, 128]),
1262 203         (1, 8192, 16, [128, 128]),
1263 204         (1, 16384, 16, [128, 128]),
1264 205         (1, 32768, 16, [128, 128]),
1265 206         (1, 65536, 16, [128, 128]),
1266 207         (1, 131072, 16, [128, 128]),
1267 208         (1, 262144, 16, [128, 128]),
1268 209         (2, 1024, 8, [128, 128]),
1269 210         (2, 1024, 32, [64, 64]),
1270 211         (2, 2048, 8, [128, 128]),
1271 212         (2, 2048, 32, [64, 64]),
1272 213         (2, 4096, 8, [128, 128]),
1273 214         (2, 4096, 32, [64, 64]),
1274 215         (2, 8192, 8, [128, 128]),
1275 216         (2, 8192, 16, [128, 128]),
1276 217         (2, 8192, 32, [64, 64]),
1277 218         (2, 16384, 8, [128, 128]),
1278 219         (2, 16384, 32, [64, 64]),
1279 220         (2, 32768, 8, [128, 128]),
1280 221         (2, 32768, 16, [128, 128]),
1281 222         (2, 32768, 32, [64, 64]),
1282 223         (2, 65536, 8, [128, 128]),
1283 224         (2, 65536, 16, [128, 128]),
1284 225         (2, 65536, 32, [64, 64]),
1285 226         (2, 131072, 8, [128, 128]),
1286 227         (2, 131072, 32, [64, 64]),
1287 228     ) :
1288 229         BLK_K = 32
1289 230         BLK_V = 64
1290 231         num_warps = 8
1291 232         num_ctas = 1
1292 233         num_stages = 4
1293 234         maxnreg = None
1294 235     elif key in (
1295 236         (2, 65536, 32, [128, 128]),
1296 237         (2, 131072, 32, [128, 128]),
1297 238     ) :
1298 239         BLK_K = 64
1299 240         BLK_V = 128
1300 241         num_warps = 8
1301 242         num_ctas = 1
1302 243         num_stages = 4
1303 244         maxnreg = None
1304 245     else:
1305 246         raise ValueError(f"Unsupported config for fuse_chunk_decay_kernel
1306 247             : BTHD={key}")

```

```

1296 247
1297 248    return {
1298 249        "CHUNK": 64,
1299 250        "USE_INITIAL_STATE": True,
1300 251        "H": H,
1301 252        "K": D[0],
1302 253        "V": D[1],
1303 254        "BLK_K": BLK_K,
1304 255        "BLK_V": BLK_V,
1305 256        "num_warps": num_warps,
1306 257        "num_stages": num_stages,
1307 258    }
1308 259
1309 260
1310 263 @aot_compile_spaces ({
1311 264    "fuse_chunk_decay_kernel": {
1312 265        "signature": fuse_chunk_decay_kernel_signature,
1313 266        "grid": ["(%K + %BLK_K - 1) / %BLK_K", "(%V + %BLK_V - 1) / %BLK_V", "H_MUL_NS"],
1314 267        "triton_algo_infos": [
1315 268            get_fuse_chunk_decay_kernel_info(B, T, H, K, V)
1316 269            for B, T, H, (K, V) in [
1317 270                (1, 262144, 32, [64, 64]),
1318 271                (1, 262144, 8, [128, 128]),
1319 272                (2, 1024, 16, [128, 128]),
1320 273                (2, 1024, 4, [64, 64]),
1321 274                (2, 131072, 1, [128, 128]),
1322 275                (2, 131072, 1, [64, 64]),
1323 276                (2, 131072, 16, [128, 128]),
1324 277                (2, 131072, 16, [64, 64]),
1325 278                (2, 131072, 32, [128, 128]),
1326 279                (2, 131072, 32, [64, 64]),
1327 280                (2, 131072, 4, [128, 128]),
1328 281                (2, 131072, 4, [64, 64]),
1329 282                (2, 131072, 8, [128, 128]),
1330 283                (2, 131072, 8, [64, 64]),
1331 284                (2, 32768, 32, [128, 128]),
1332 285                (2, 65536, 16, [128, 128])
1333 286            ]
1334 287        ],
1335 288    }
1336 289 })
1337 290 @triton.autotune(
1338 291     configs=[
1339 292         triton.Config({"BLK_K": BLK_K, "BLK_V": BLK_V, "USE_TMA": USE_TMA},
1340 293         num_warps=num_warps, num_stages=num_stages)
1341 294         for num_warps in [4, 8]
1342 295         for num_stages in [3, 4]
1343 296         for BLK_K in [128, 64, 32]
1344 297         for BLK_V in [128, 64, 32]
1345 298         for USE_TMA in [True, False]
1346 299     ],
1347 300     key=[],
1348 301 @triton.jit
1349 302 def fuse_chunk_decay_kernel(
1350 303     k,
1351 304     v,
1352 305     g,
1353 306     prev_s,
1354 307     out_0,
1355 308     out_1,
1356 309     cu_seqlens,

```

```

1350      NS,
1351      H_MUL_NS,
1352      chunk_offsets_with_ini,
1353      USE_INITIAL_STATE: tl.constexpr,
1354      H: tl.constexpr,
1355      K: tl.constexpr,
1356      V: tl.constexpr,
1357      CHUNK: tl.constexpr,
1358      BLK_K: tl.constexpr,
1359      BLK_V: tl.constexpr,
1360      USE_TMA: tl.constexpr,
1361  ) :
1362      NUM_BLK_K = (K + BLK_K - 1) // BLK_K
1363      NUM_BLK_V = (V + BLK_V - 1) // BLK_V
1364      NUM_BLK_KV = NUM_BLK_K * NUM_BLK_V
1365
1366      i_k, i_v, i_sh = tl.program_id(0), tl.program_id(1), tl.program_id(2)
1367      i_s, i_h = i_sh // H, i_sh % H
1368      bos, eos = tl.load(cu_seqlens + i_s).to(tl.int32), tl.load(cu_seqlens
1369      + i_s + 1).to(tl.int32)
1370      T = eos - bos
1371      NC = tl.cdiv(T, CHUNK)
1372      boh = tl.load(chunk_offsets_with_ini + i_s).to(tl.int32)
1373
1374      out_0 = out_0 + (boh * H + i_h).to(tl.int64) * K * V
1375      prev_s = prev_s + (i_s * H + i_h) * K * V
1376      initial_decay = 1.0
1377      # out_1: [NC + NS, H]. NOTE: **not** in log space
1378      ptr_out_1 = out_1 + boh * H + i_h
1379      # store initial decay
1380      tl.store(ptr_out_1, initial_decay)
1381      ptr_out_1 += H
1382      # [BK, BV]
1383      blk_prev_s = tl.zeros([BLK_K, BLK_V], dtype=tl.float32)
1384      if USE_INITIAL_STATE:
1385          ptr_prev_s = tl.make_block_ptr(prev_s, (K, V), (V, 1), (i_k *
1386          BLK_K, i_v * BLK_V), (BLK_K, BLK_V), (1, 0))
1387          blk_prev_s = tl.load(ptr_prev_s, boundary_check=(0, 1)).to(tl.
1388          float32)
1389          ptr_out_0 = tl.make_block_ptr(out_0, (K, V), (V, 1), (i_k*BLK_K, i_v*
1390          BLK_V), (BLK_K, BLK_V), (1, 0)) # fmt: skip
1391          tl.store(ptr_out_0, blk_prev_s.to(ptr_out_0.dtype.element_ty))
1392          out_0 += H * K * V
1393      if USE_TMA:
1394          k_desc = tl.make_tensor_descriptor(
1395              k + bos * H * K + i_h * K,
1396              shape=[T, K],
1397              strides=[H * K, 1],
1398              block_shape=[CHUNK, BLK_K],
1399          )
1400          v_desc = tl.make_tensor_descriptor(
1401              v + bos * H * V + i_h * V,
1402              shape=[T, V],
1403              strides=[H * V, 1],
1404              block_shape=[CHUNK, BLK_V],
1405          )
1406          out_0_desc = tl.make_tensor_descriptor(
1407              out_0,
1408              shape=[NC, K, V],
1409              strides=[H * K * V, V, 1],
1410              block_shape=[1, BLK_K, BLK_V],
1411          )
1412
1413      for i_c in range(NC):

```

```

1404      # load (trans) 'k': (T, H, K,) => (BLK_K, CHUNK,)
1405      if not USE_TMA:
1406          cur_k = k + bos * H * K + i_h * K  # fmt: skip
1407          ptr_k_0 = tl.make_block_ptr(cur_k, (K, T,), (1, H * K,), (i_k
1408          * BLK_K, i_c * CHUNK,), (BLK_K, CHUNK,), (0, 1,))  # fmt: skip
1409          blk_k_0 = tl.load(ptr_k_0, boundary_check=(0, 1,))  # fmt:
1410      skip
1411      else:
1412          blk_k_0 = k_desc.load([i_c * CHUNK, i_k * BLK_K]).trans()
1413      # load 'v': (T, H, V,) => (CHUNK, BLK_V,)
1414      if not USE_TMA:
1415          cur_v = v + bos * H * V + i_h * V  # fmt: skip
1416          ptr_v_1 = tl.make_block_ptr(cur_v, (T, V,), (H * V, 1,), (i_c
1417          * CHUNK, i_v * BLK_V,), (CHUNK, BLK_V,), (1, 0,))  # fmt: skip
1418          blk_v_1 = tl.load(ptr_v_1, boundary_check=(1, 0,))  # fmt:
1419      skip
1420      else:
1421          blk_v_1 = v_desc.load([i_c * CHUNK, i_v * BLK_V])
1422
1423      # call_external_func: 'chunk_local_cumsum' to pre-compute g
1424
1425      # load 'g': (T, H,) => (CHUNK,)
1426      cur_g = g + bos * H + i_h * 1  # fmt: skip
1427      ptr_g_2 = tl.make_block_ptr(cur_g, (T,), (H,), (i_c * CHUNK,), (CHUNK,
1428      , 0,))  # fmt: skip
1429      blk_g_2 = tl.load(ptr_g_2, boundary_check=(0,))  # fmt: skip
1430      # load_last_g: => ()
1431      cur_g = g + bos * H + i_h * 1 + (CHUNK - 1) * H  # fmt: skip
1432      ptr_last_g_3 = cur_g + i_c * CHUNK * H  # fmt: skip
1433      last_g_3 = tl.load(ptr_last_g_3)
1434      sub: torch.Size([]), ('CHUNK',) => ('CHUNK',)
1435      sub_4 = last_g_3 - blk_g_2
1436      unsqueeze_5 = sub_4[None, :]
1437      exp: (1, 'CHUNK') => (1, 'CHUNK')
1438      exp_6 = tl.exp(unsqueeze_5)
1439      to: ('BLK_K', 'CHUNK') => ('BLK_K', 'CHUNK')
1440      blk_k_0_float32_7 = blk_k_0.to(tl.float32)
1441      mul: ('BLK_K', 'CHUNK'), (1, 'CHUNK') => ('BLK_K', 'CHUNK')
1442      mul_8 = blk_k_0_float32_7 * exp_6
1443      to: ('BLK_K', 'CHUNK') => ('BLK_K', 'CHUNK')
1444      mul_8_bfloat16_9 = mul_8.to(tl.bfloat16)
1445      matmul: ('BLK_K', 'CHUNK'), ('CHUNK', 'BLK_V') => ('BLK_K', 'BLK_V')
1446      matmul_10 = tl.dot(mul_8_bfloat16_9, blk_v_1).to(mul_8_bfloat16_9
1447      .dtype)
1448      # exp: () => ()
1449      exp_1_11 = tl.exp(last_g_3)
1450      mul: ('BLK_K', 'BLK_V'), () => ('BLK_K', 'BLK_V')
1451      mul_1_12 = blk_prev_s * exp_1_11
1452      # add: ('BLK_K', 'BLK_V'), ('BLK_K', 'BLK_V') => ('BLK_K', 'BLK_V')
1453      add_13 = mul_1_12 + matmul_10
1454      mul: ('s3',), () => ('s3',)
1455      mul_tensor_14 = initial_decay * exp_1_11
1456      # store => ('BLK_K', 'BLK_V')
1457      # assume output layout: [NC_WITH_INI, H, K, V]

```

```

1458     427         tl.store(ptr_out_0, add_13.to(out_0_ty), boundary_check=(1,
1459     427         0))
1460     428             out_0 += H * K * V
1461     429         else:
1462     430             out_0_desc.store([i_c, i_k * BLK_K, i_v * BLK_V], add_13.to(
1463     431             out_0_ty) [None, :, :])
1464     432                 blk_prev_s = add_13.to(blk_prev_s.dtype)
1465     433                     # store => ('s3',)
1466     434                     # output layout: [NC_WITH_INI, H, s3]
1467     435                     out_1_ty = out_1.dtype.element_ty
1468     436                     tl.store(ptr_out_1 + i_c * H, mul_tensor_14.to(out_1_ty))
1469     437                     initial_decay = mul_tensor_14
1470     438
1471     439     def launch_fuse_chunk_decay(
1472     440         g: torch.Tensor,
1473     441         prev_s: torch.Tensor,
1474     442         v: torch.Tensor,
1475     443         k: torch.Tensor,
1476     444         cu_seqlens: torch.IntTensor,
1477     445         cached_results: dict,
1478     446         dist_scan: DistScanContext = None,
1479     447         lazy_update: bool = True,
1480     448     ) -> dict[str, torch.Tensor]:
1481     449         """
1482     450             perform "decayed scan" on 'chunked_states'
1483     451             """
1484     452             # state_dtype = chunk_state.dtype
1485     453             chunk_decay = None
1486     454             prev_rank_state_sum = None
1487     455
1488     456             # TODO: now fix chunk_size to 64
1489     457             CHUNK = 64
1490     458             chunk_indices = prepare_chunk_indices(cu_seqlens, CHUNK)
1491     459             chunk_offsets = prepare_chunk_offsets(cu_seqlens, CHUNK)
1492     460             chunk_offsets_with_ini = prepare_chunk_offsets_with_ini(cu_seqlens,
1493     461             CHUNK)
1494     462             NS, NC = len(cu_seqlens) - 1, chunk_indices.shape[0]
1495     463             NC_WITH_INI = NC + NS
1496     464             use_aot = os.environ.get('FORGE_USE_AOT', '0')
1497     465             FORGE_USE_AOT = True if use_aot.lower() in ['1', 'true', 'yes'] else
1498     466             None
1499     467             T, H, K, V = *k.shape, v.shape[-1]
1500     468             updated_states = k.new_empty([NC_WITH_INI, H, K, V])
1501     469
1502     470             def alloc_fn(size: int, alignment: int, stream: int):
1503     471                 return torch.empty(size, device='cuda', dtype=torch.int8)
1504     472
1505     473             triton.set_allocator(alloc_fn)
1506     474
1507     475             def grid(meta):
1508     476                 BLK_K = meta['BLK_K']
1509     477                 BLK_V = meta['BLK_V']
1510     478                 NUM_BLK_K = (K + BLK_K - 1) // BLK_K
1511     479                 NUM_BLK_V = (V + BLK_V - 1) // BLK_V
1512     480                 NUM_BLK_KV = NUM_BLK_K * NUM_BLK_V
1513     481                 H_MUL_NS = H * NS
1514     482                 return (NUM_BLK_K, NUM_BLK_V, H_MUL_NS)
1515     483
1516     484             chunk_decay = torch.empty([NC_WITH_INI, H], dtype=torch.float32)
1517     485             if FORGE_USE_AOT is None:
1518     486                 fuse_chunk_decay_kernel[grid](
1519     487                     g=g,
1520     488                     prev_s=prev_s,
1521     489                     v=v,
1522     490                     k=k,
1523     491                     chunk_decay=chunk_decay,
1524     492                     NS=NS,
1525     493                     NC=NC,
1526     494                     cu_seqlens=cu_seqlens,
1527     495                     chunk_indices=chunk_indices,
1528     496                     chunk_offsets=chunk_offsets,
1529     497                     chunk_offsets_with_ini=chunk_offsets_with_ini,
1530     498                     dist_scan=dist_scan,
1531     499                     lazy_update=lazy_update)
1532     500
1533     501             return updated_states
1534     502
1535     503
1536     504
1537     505
1538     506
1539     507
1540     508
1541     509
1542     510
1543     511
1544     512
1545     513
1546     514
1547     515
1548     516
1549     517
1550     518
1551     519
1552     520
1553     521
1554     522
1555     523
1556     524
1557     525
1558     526
1559     527
1560     528
1561     529
1562     530
1563     531
1564     532
1565     533
1566     534
1567     535
1568     536
1569     537
1570     538
1571     539
1572     540
1573     541
1574     542
1575     543
1576     544
1577     545
1578     546
1579     547
1580     548
1581     549
1582     550
1583     551
1584     552
1585     553
1586     554
1587     555
1588     556
1589     557
1590     558
1591     559
1592     560
1593     561
1594     562
1595     563
1596     564
1597     565
1598     566
1599     567
1600     568
1601     569
1602     570
1603     571
1604     572
1605     573
1606     574
1607     575
1608     576
1609     577
1610     578
1611     579
1612     580
1613     581
1614     582
1615     583
1616     584
1617     585
1618     586
1619     587
1620     588
1621     589
1622     590
1623     591
1624     592
1625     593
1626     594
1627     595
1628     596
1629     597
1630     598
1631     599
1632     600
1633     601
1634     602
1635     603
1636     604
1637     605
1638     606
1639     607
1640     608
1641     609
1642     610
1643     611
1644     612
1645     613
1646     614
1647     615
1648     616
1649     617
1650     618
1651     619
1652     620
1653     621
1654     622
1655     623
1656     624
1657     625
1658     626
1659     627
1660     628
1661     629
1662     630
1663     631
1664     632
1665     633
1666     634
1667     635
1668     636
1669     637
1670     638
1671     639
1672     640
1673     641
1674     642
1675     643
1676     644
1677     645
1678     646
1679     647
1680     648
1681     649
1682     650
1683     651
1684     652
1685     653
1686     654
1687     655
1688     656
1689     657
1690     658
1691     659
1692     660
1693     661
1694     662
1695     663
1696     664
1697     665
1698     666
1699     667
1700     668
1701     669
1702     670
1703     671
1704     672
1705     673
1706     674
1707     675
1708     676
1709     677
1710     678
1711     679
1712     680
1713     681
1714     682
1715     683
1716     684
1717     685
1718     686
1719     687
1720     688
1721     689
1722     690
1723     691
1724     692
1725     693
1726     694
1727     695
1728     696
1729     697
1730     698
1731     699
1732     700
1733     701
1734     702
1735     703
1736     704
1737     705
1738     706
1739     707
1740     708
1741     709
1742     710
1743     711
1744     712
1745     713
1746     714
1747     715
1748     716
1749     717
1750     718
1751     719
1752     720
1753     721
1754     722
1755     723
1756     724
1757     725
1758     726
1759     727
1760     728
1761     729
1762     730
1763     731
1764     732
1765     733
1766     734
1767     735
1768     736
1769     737
1770     738
1771     739
1772     740
1773     741
1774     742
1775     743
1776     744
1777     745
1778     746
1779     747
1780     748
1781     749
1782     750
1783     751
1784     752
1785     753
1786     754
1787     755
1788     756
1789     757
1790     758
1791     759
1792     760
1793     761
1794     762
1795     763
1796     764
1797     765
1798     766
1799     767
1800     768
1801     769
1802     770
1803     771
1804     772
1805     773
1806     774
1807     775
1808     776
1809     777
1810     778
1811     779
1812     780
1813     781
1814     782
1815     783
1816     784
1817     785
1818     786
1819     787
1820     788
1821     789
1822     790
1823     791
1824     792
1825     793
1826     794
1827     795
1828     796
1829     797
1830     798
1831     799
1832     800
1833     801
1834     802
1835     803
1836     804
1837     805
1838     806
1839     807
1840     808
1841     809
1842     810
1843     811
1844     812
1845     813
1846     814
1847     815
1848     816
1849     817
1850     818
1851     819
1852     820
1853     821
1854     822
1855     823
1856     824
1857     825
1858     826
1859     827
1860     828
1861     829
1862     830
1863     831
1864     832
1865     833
1866     834
1867     835
1868     836
1869     837
1870     838
1871     839
1872     840
1873     841
1874     842
1875     843
1876     844
1877     845
1878     846
1879     847
1880     848
1881     849
1882     850
1883     851
1884     852
1885     853
1886     854
1887     855
1888     856
1889     857
1890     858
1891     859
1892     860
1893     861
1894     862
1895     863
1896     864
1897     865
1898     866
1899     867
1900     868
1901     869
1902     870
1903     871
1904     872
1905     873
1906     874
1907     875
1908     876
1909     877
1910     878
1911     879
1912     880
1913     881
1914     882
1915     883
1916     884
1917     885
1918     886
1919     887
1920     888
1921     889
1922     890
1923     891
1924     892
1925     893
1926     894
1927     895
1928     896
1929     897
1930     898
1931     899
1932     900
1933     901
1934     902
1935     903
1936     904
1937     905
1938     906
1939     907
1940     908
1941     909
1942     910
1943     911
1944     912
1945     913
1946     914
1947     915
1948     916
1949     917
1950     918
1951     919
1952     920
1953     921
1954     922
1955     923
1956     924
1957     925
1958     926
1959     927
1960     928
1961     929
1962     930
1963     931
1964     932
1965     933
1966     934
1967     935
1968     936
1969     937
1970     938
1971     939
1972     940
1973     941
1974     942
1975     943
1976     944
1977     945
1978     946
1979     947
1980     948
1981     949
1982     950
1983     951
1984     952
1985     953
1986     954
1987     955
1988     956
1989     957
1990     958
1991     959
1992     960
1993     961
1994     962
1995     963
1996     964
1997     965
1998     966
1999     967
2000     968
2001     969
2002     970
2003     971
2004     972
2005     973
2006     974
2007     975
2008     976
2009     977
2010     978
2011     979
2012     980
2013     981
2014     982
2015     983
2016     984
2017     985
2018     986
2019     987
2020     988
2021     989
2022     990
2023     991
2024     992
2025     993
2026     994
2027     995
2028     996
2029     997
2030     998
2031     999
2032     999
2033     999
2034     999
2035     999
2036     999
2037     999
2038     999
2039     999
2040     999
2041     999
2042     999
2043     999
2044     999
2045     999
2046     999
2047     999
2048     999
2049     999
2050     999
2051     999
2052     999
2053     999
2054     999
2055     999
2056     999
2057     999
2058     999
2059     999
2060     999
2061     999
2062     999
2063     999
2064     999
2065     999
2066     999
2067     999
2068     999
2069     999
2070     999
2071     999
2072     999
2073     999
2074     999
2075     999
2076     999
2077     999
2078     999
2079     999
2080     999
2081     999
2082     999
2083     999
2084     999
2085     999
2086     999
2087     999
2088     999
2089     999
2090     999
2091     999
2092     999
2093     999
2094     999
2095     999
2096     999
2097     999
2098     999
2099     999
2100     999
2101     999
2102     999
2103     999
2104     999
2105     999
2106     999
2107     999
2108     999
2109     999
2110     999
2111     999
2112     999
2113     999
2114     999
2115     999
2116     999
2117     999
2118     999
2119     999
2120     999
2121     999
2122     999
2123     999
2124     999
2125     999
2126     999
2127     999
2128     999
2129     999
2130     999
2131     999
2132     999
2133     999
2134     999
2135     999
2136     999
2137     999
2138     999
2139     999
2140     999
2141     999
2142     999
2143     999
2144     999
2145     999
2146     999
2147     999
2148     999
2149     999
2150     999
2151     999
2152     999
2153     999
2154     999
2155     999
2156     999
2157     999
2158     999
2159     999
2160     999
2161     999
2162     999
2163     999
2164     999
2165     999
2166     999
2167     999
2168     999
2169     999
2170     999
2171     999
2172     999
2173     999
2174     999
2175     999
2176     999
2177     999
2178     999
2179     999
2180     999
2181     999
2182     999
2183     999
2184     999
2185     999
2186     999
2187     999
2188     999
2189     999
2190     999
2191     999
2192     999
2193     999
2194     999
2195     999
2196     999
2197     999
2198     999
2199     999
2200     999
2201     999
2202     999
2203     999
2204     999
2205     999
2206     999
2207     999
2208     999
2209     999
2210     999
2211     999
2212     999
2213     999
2214     999
2215     999
2216     999
2217     999
2218     999
2219     999
2220     999
2221     999
2222     999
2223     999
2224     999
2225     999
2226     999
2227     999
2228     999
2229     999
2230     999
2231     999
2232     999
2233     999
2234     999
2235     999
2236     999
2237     999
2238     999
2239     999
2240     999
2241     999
2242     999
2243     999
2244     999
2245     999
2246     999
2247     999
2248     999
2249     999
2250     999
2251     999
2252     999
2253     999
2254     999
2255     999
2256     999
2257     999
2258     999
2259     999
2260     999
2261     999
2262     999
2263     999
2264     999
2265     999
2266     999
2267     999
2268     999
2269     999
2270     999
2271     999
2272     999
2273     999
2274     999
2275     999
2276     999
2277     999
2278     999
2279     999
2280     999
2281     999
2282     999
2283     999
2284     999
2285     999
2286     999
2287     999
2288     999
2289     999
2290     999
2291     999
2292     999
2293     999
2294     999
2295     999
2296     999
2297     999
2298     999
2299     999
2300     999
2301     999
2302     999
2303     999
2304     999
2305     999
2306     999
2307     999
2308     999
2309     999
2310     999
2311     999
2312     999
2313     999
2314     999
2315     999
2316     999
2317     999
2318     999
2319     999
2320     999
2321     999
2322     999
2323     999
2324     999
2325     999
2326     999
2327     999
2328     999
2329     999
2330     999
2331     999
2332     999
2333     999
2334     999
2335     999
2336     999
2337     999
2338     999
2339     999
2340     999
2341     999
2342     999
2343     999
2344     999
2345     999
2346     999
2347     999
2348     999
2349     999
2350     999
2351     999
2352     999
2353     999
2354     999
2355     999
2356     999
2357     999
2358     999
2359     999
2360     999
2361     999
2362     999
2363     999
2364     999
2365     999
2366     999
2367     999
2368     999
2369     999
2370     999
2371     999
2372     999
2373     999
2374     999
2375     999
2376     999
2377     999
2378     999
2379     999
2380     999
2381     999
2382     999
2383     999
2384     999
2385     999
2386     999
2387     999
2388     999
2389     999
2390     999
2391     999
2392     999
2393     999
2394     999
2395     999
2396     999
2397     999
2398     999
2399     999
2400     999
2401     999
2402     999
2403     999
2404     999
2405     999
2406     999
2407     999
2408     999
2409     999
2410     999
2411     999
2412     999
2413     999
2414     999
2415     999
2416     999
2417     999
2418     999
2419     999
2420     999
2421     999
2422     999
2423     999
2424     999
2425     999
2426     999
2427     999
2428     999
2429     999
2430     999
2431     999
2432     999
2433     999
2434     999
2435     999
2436     999
2437     999
2438     999
2439     999
2440     999
2441     999
2442     999
2443     999
2444     999
2445     999
2446     999
2447     999
2448     999
2449     999
2450     999
2451     999
2452     999
2453     999
2454     999
2455     999
2456     999
2457     999
2458     999
2459     999
2460     999
2461     999
2462     999
2463     999
2464     999
2465     999
2466     999
2467     999
2468     999
2469     999
2470     999
2471     999
2472     999
2473     999
2474     999
2475     999
2476     999
2477     999
2478     999
2479     999
2480     999
2481     999
2482     999
2483     999
2484     999
2485     999
2486     999
2487     999
2488     999
2489     999
2490     999
2491     999
2492     999
2493     999
2494     999
2495     999
2496     999
2497     999
2498     999
2499     999
2500     999
2501     999
2502     999
2503     999
2504     999
2505     999
2506     999
2507     999
2508     999
2509     999
2510     999
2511     999
2512     999
2513     999
2514     999
2515     999
2516     999
2517     999
2518     999
2519     999
2520     999
2521     999
2522     999
2523     999
2524     999
2525     999
2526     999
2527     999
2528     999
2529     999
2530     999
2531     999
2532     999
2533     999
2534     999
2535     999
2536     999
2537     999
2538     999
2539     999
2540     999
2541     999
2542     999
2543     999
2544     999
2545     999
2546     999
2547     999
2548     999
2549     999
2550     999
2551     999
2552     999
2553     999
2554     999
2555     999
2556     999
2557     999
2558     999
2559     999
2560     999
2561     999
2562     999
2563     999
2564     999
2565     999
2566     999
2567     999
2568     999
2569     999
2570     999
2571     999
2572     999
2573     999
2574     999
2575     999
2576     999
2577     999
2578     999
2579     999
2580     999
2581     999
2582     999
2583     999
2584     999
2585     999
2586     999
2587     999
2588     999
2589     999
2590     999
2591     999
2592     999
2593     999
2594     999
2595     999
2596     999
2597     999
2598     999
2599     999
2600     999
2601     999
2602     999
2603     999
2604     999
2605     999
2606     999
2607     999
2608     999
2609     999
2610     999
2611     999
2612     999
2613     999
2614     999
2615     999
2616     999
2617     999
2618     999
2619     999
2620     999
2621     999
2622     999
2623     999
2624     999
2625
```

```

1512 488         out_1=chunk_decay,
1513 489         cu_seqlens=cu_seqlens,
1514 490         H=H,
1515 491         NS=NS,
1516 492         K=K,
1517 493         V=V,
1518 494         CHUNK=CHUNK,
1519 495         H_MUL_NS=H * NS,
1520 496         out_0=updated_states,
1521 497         chunk_offsets_with_ini=chunk_offsets_with_ini,
1522 498         USE_INITIAL_STATE=True,
1523 500     )
1524 501
1525 503     else:
1526 504
1527 505         from foge.aot_utils import forge_aot_ops
1528 506
1529 507         algo_info = forge_aot_ops.
1530 508         fuse_chunk_decay_kernel__triton_algo_info_t()
1531 509         for _k, _v in get_fuse_chunk_decay_kernel_info(NS, T, H, K, V).
1532 510         items():
1533 511             setattr(algo_info, _k, _v)
1534 512             forge_aot_ops.fuse_chunk_decay_kernel(
1535 513                 0, # torch.cuda.current_stream().cuda_stream,
1536 514                 k.data_ptr(), # k
1537 515                 v.data_ptr(), # v
1538 516                 g.data_ptr(), # g
1539 517                 prev_s.data_ptr(), # prev_s
1540 518                 updated_states.data_ptr(), # out_0
1541 519                 chunk_decay.data_ptr(), # out_1
1542 520                 cu_seqlens.data_ptr(), # cu_seqlens
1543 521                 NS, # NS
1544 522                 H * NS, # H_MUL_NS
1545 523                 chunk_offsets_with_ini.data_ptr(), # chunk_offsets_with_ini
1546 524                 algo_info,
1547 525             )
1548 526
1549 527         final_chunk_indices = chunk_offsets_with_ini[1:] - 1
1550 528         final_state_local = updated_states[final_chunk_indices, ...]
1551 529         final_decay_local = chunk_decay[final_chunk_indices, ...]
1552 530
1553 531         if dist_scan.pg.size() > 1:
1554 532             prev_rank_state_sum = dist_scan.forward(
1555 533                 final_state_local=final_state_local,
1556 534                 final_decay_local=final_decay_local,
1557 535                 decay_type=DecayType.SCALAR,
1558 536                 lazy_update=True,
1559 537             )
1560 538
1561 539         cached_results.update({})
1562 540         if prev_rank_state_sum is not None:
1563 541             return {
1564 542                 "chunk_state": updated_states,
1565 543                 "chunk_decay": chunk_decay,
1566 544                 "prev_rank_state_sum": prev_rank_state_sum,
1567 545             }
1568 546         else:
1569 547             return {
1570 548                 "chunk_state": updated_states,
1571 549                 "chunk_decay": None,
1572 550                 "prev_rank_state_sum": None,
1573 551             }
1574 552         merge_mode_kernel_signature = (
1575 553             "*bf16:16, "

```

```

1566      "*bf16:16, "
1567      "*bf16:16, "
1568      "*fp32, "
1569      "*bf16:16, "
1570      "fp32, "
1571      "*bf16:16, "
1572      "i32:16, "
1573      "*bf16:16, "
1574      "i32:16, "
1575      "*bf16:16, "
1576      "*i32, "
1577      "i32, "
1578      "*i32, "
1579      "%FUSE_SP_STATE_UPDATE, "
1580      "%H, "
1581      "%K, "
1582      "%V, "
1583      "%CHUNK, "
1584      "%BLK_K, "
1585      "%BLK_V, "
1586      "%USE_TMA"
1587
1588  def get_merge_mode_kernel_info(B: int, T: int, H: int, K: int, V: int):
1589      """Static dispatcher for merge_mode_kernel. Auto-generated."""
1590      D = [K, V]
1591      key = (B, T, H, D)
1592      if key in (
1593          (1, 4096, 4, [128, 128]),
1594          (1, 131072, 32, [128, 128]),
1595          (1, 262144, 32, [128, 128]),
1596          (2, 1024, 8, [128, 128]),
1597          (2, 4096, 1, [128, 128]),
1598          (2, 8192, 1, [128, 128]),
1599      ):
1600          BLK_K = 128
1601          BLK_V = 128
1602          num_warps = 4
1603          num_ctas = 1
1604          num_stages = 3
1605          maxnreg = None
1606      elif key in (
1607          (1, 4096, 32, [128, 128]),
1608          (1, 16384, 8, [128, 128]),
1609          (1, 16384, 32, [128, 128]),
1610          (1, 32768, 8, [128, 128]),
1611          (1, 32768, 16, [128, 128]),
1612          (1, 32768, 32, [128, 128]),
1613          (1, 65536, 8, [128, 128]),
1614          (1, 65536, 32, [128, 128]),
1615          (1, 131072, 4, [128, 128]),
1616          (1, 131072, 8, [128, 128]),
1617          (1, 131072, 16, [128, 128]),
1618          (1, 262144, 4, [128, 128]),
1619          (1, 262144, 8, [128, 128]),
1620          (1, 262144, 16, [128, 128]),
1621          (2, 4096, 16, [128, 128]),
1622          (2, 4096, 32, [128, 128]),
1623          (2, 16384, 8, [128, 128]),
1624          (2, 16384, 32, [128, 128]),
1625          (2, 32768, 16, [128, 128]),
1626          (2, 32768, 32, [128, 128]),
1627      ):
1628          BLK_K = 128
1629          BLK_V = 128
1630          num_warps = 4
1631          num_ctas = 1
1632          num_stages = 3
1633          maxnreg = None
1634
1635      return {
1636          "B": B,
1637          "T": T,
1638          "H": H,
1639          "K": K,
1640          "V": V,
1641          "BLK_K": BLK_K,
1642          "BLK_V": BLK_V,
1643          "num_warps": num_warps,
1644          "num_ctas": num_ctas,
1645          "num_stages": num_stages,
1646          "maxnreg": maxnreg
1647      }

```

```

1620      616      (2, 65536, 8, [128, 128]),
1621      617      (2, 65536, 16, [128, 128]),
1622      618      (2, 131072, 4, [128, 128]),
1623      619      (2, 131072, 8, [128, 128]),
1624      620      (2, 131072, 16, [128, 128]),
1625      621      ) :
1626      622      BLK_K = 128
1627      623      BLK_V = 128
1628      624      num_warp = 8
1629      625      num_ctas = 1
1630      626      num_stages = 3
1631      627      maxnreg = None
1632      628      elif key in (
1633      629      (1, 1024, 16, [128, 128]),
1634      630      (1, 1024, 32, [128, 128]),
1635      631      (1, 2048, 8, [128, 128]),
1636      632      (1, 2048, 16, [128, 128]),
1637      633      (1, 16384, 1, [128, 128]),
1638      634      (1, 32768, 1, [128, 128]),
1639      635      (2, 2048, 4, [128, 128]),
1640      636      (2, 65536, 32, [128, 128]),
1641      637      (2, 131072, 32, [128, 128]),
1642      638      ) :
1643      639      BLK_K = 128
1644      640      BLK_V = 128
1645      641      num_warp = 4
1646      642      num_ctas = 1
1647      643      num_stages = 4
1648      644      maxnreg = None
1649      645      elif key in (
1650      646      (1, 2048, 32, [128, 128]),
1651      647      (1, 4096, 16, [128, 128]),
1652      648      (1, 8192, 16, [128, 128]),
1653      649      (1, 8192, 32, [128, 128]),
1654      650      (1, 16384, 16, [128, 128]),
1655      651      (1, 65536, 4, [128, 128]),
1656      652      (1, 65536, 16, [128, 128]),
1657      653      (2, 1024, 32, [128, 128]),
1658      654      (2, 2048, 32, [128, 128]),
1659      655      (2, 8192, 8, [128, 128]),
1660      656      (2, 8192, 16, [128, 128]),
1661      657      (2, 8192, 32, [128, 128]),
1662      658      (2, 16384, 16, [128, 128]),
1663      659      (2, 32768, 4, [128, 128]),
1664      660      (2, 32768, 8, [128, 128]),
1665      661      (2, 65536, 4, [128, 128]),
1666      662      ) :
1667      663      BLK_K = 128
1668      664      BLK_V = 128
1669      665      num_warp = 8
1670      666      num_ctas = 1
1671      667      num_stages = 4
1672      668      maxnreg = None
1673      669      elif key in (
1674      670      (1, 1024, 1, [64, 64]),
1675      671      (1, 4096, 1, [64, 64]),
1676      672      ) :
1677      673      BLK_K = 128
1678      674      BLK_V = 32
1679      675      num_warp = 4
1680      676      num_ctas = 1
1681      677      num_stages = 3
1682      678      maxnreg = None
1683      679      elif key in (
1684      680      (1, 1024, 1, [128, 128]),

```

```

1674 681      (1, 1024, 4, [64, 64]),
1675 682      (1, 1024, 8, [64, 64]),
1676 683      (1, 2048, 1, [64, 64]),
1677 684      (1, 2048, 1, [128, 128]),
1678 685      (1, 2048, 4, [64, 64]),
1679 686      (2, 1024, 1, [64, 64]),
1680 687      (2, 2048, 1, [64, 64]),
1680 688  ) :
1681 689      BLK_K = 128
1682 690      BLK_V = 32
1683 691      num_warp = 4
1684 692      num_ctas = 1
1685 693      num_stages = 4
1685 694      maxnreg = None
1686 695  elif key in (
1687 696      (1, 1024, 8, [128, 128]),
1688 697      (1, 1024, 32, [64, 64]),
1689 698      (1, 2048, 4, [128, 128]),
1689 699      (1, 4096, 4, [64, 64]),
1690 700      (1, 4096, 8, [64, 64]),
1691 701      (1, 4096, 8, [128, 128]),
1692 702      (1, 8192, 1, [128, 128]),
1693 703      (1, 8192, 8, [64, 64]),
1694 704      (1, 8192, 8, [128, 128]),
1695 705      (1, 8192, 32, [64, 64]),
1695 706      (1, 16384, 1, [64, 64]),
1696 707      (1, 16384, 32, [64, 64]),
1697 708      (1, 32768, 1, [64, 64]),
1698 709      (1, 32768, 4, [64, 64]),
1699 710      (1, 32768, 8, [64, 64]),
1700 711      (1, 32768, 16, [64, 64]),
1700 712      (1, 65536, 1, [128, 128]),
1701 713      (1, 131072, 1, [128, 128]),
1702 714      (1, 131072, 32, [64, 64]),
1703 715      (1, 262144, 1, [64, 64]),
1704 716      (1, 262144, 32, [64, 64]),
1705 717      (2, 1024, 4, [64, 64]),
1705 718      (2, 1024, 4, [128, 128]),
1706 719      (2, 1024, 8, [64, 64]),
1707 720      (2, 1024, 16, [64, 64]),
1708 721      (2, 1024, 32, [64, 64]),
1709 722      (2, 2048, 4, [64, 64]),
1710 723      (2, 2048, 8, [64, 64]),
1710 724      (2, 2048, 8, [128, 128]),
1711 725      (2, 2048, 16, [64, 64]),
1712 726      (2, 2048, 16, [128, 128]),
1713 727      (2, 2048, 32, [64, 64]),
1714 728      (2, 4096, 4, [64, 64]),
1714 729      (2, 4096, 4, [128, 128]),
1715 730      (2, 4096, 8, [64, 64]),
1716 731      (2, 4096, 8, [128, 128]),
1717 732      (2, 4096, 16, [64, 64]),
1718 733      (2, 4096, 32, [64, 64]),
1719 734      (2, 8192, 1, [64, 64]),
1719 735      (2, 8192, 4, [128, 128]),
1720 736      (2, 16384, 1, [64, 64]),
1721 737      (2, 16384, 1, [128, 128]),
1722 738      (2, 16384, 4, [128, 128]),
1723 739      (2, 16384, 8, [64, 64]),
1724 740      (2, 16384, 32, [64, 64]),
1725 741      (2, 32768, 16, [64, 64]),
1725 742      (2, 32768, 32, [64, 64]),
1726 743      (2, 65536, 1, [64, 64]),
1727 744      (2, 65536, 1, [128, 128]),
1727 744      (2, 65536, 16, [64, 64]),

```

```

1728 746 ) :
1729 747     BLK_K = 128
1730 748     BLK_V = 64
1731 749     num_warp = 4
1732 750     num_ctas = 1
1733 751     num_stages = 3
1734 752     maxnreg = None
1735 753     elif key in (
1736 754         (1, 4096, 1, [128, 128]),
1737 755     ) :
1738 756         BLK_K = 128
1739 757         BLK_V = 64
1740 758         num_warp = 8
1741 759         num_ctas = 1
1742 760         num_stages = 3
1743 761         maxnreg = None
1744 762         elif key in (
1745 763             (1, 1024, 4, [128, 128]),
1746 764             (1, 1024, 16, [64, 64]),
1747 765             (1, 2048, 8, [64, 64]),
1748 766             (1, 2048, 16, [64, 64]),
1749 767             (1, 2048, 32, [64, 64]),
1750 768             (1, 4096, 16, [64, 64]),
1751 769             (1, 4096, 32, [64, 64]),
1752 770             (1, 8192, 1, [64, 64]),
1753 771             (1, 8192, 4, [64, 64]),
1754 772             (1, 8192, 4, [128, 128]),
1755 773             (1, 8192, 16, [64, 64]),
1756 774             (1, 16384, 4, [64, 64]),
1757 775             (1, 16384, 4, [128, 128]),
1758 776             (1, 16384, 8, [64, 64]),
1759 777             (1, 16384, 16, [64, 64]),
1760 778             (1, 32768, 4, [128, 128]),
1761 779             (1, 32768, 32, [64, 64]),
1762 780             (1, 65536, 1, [64, 64]),
1763 781             (1, 65536, 4, [64, 64]),
1764 782             (1, 65536, 8, [64, 64]),
1765 783             (1, 65536, 16, [64, 64]),
1766 784             (1, 65536, 32, [64, 64]),
1767 785             (1, 131072, 1, [64, 64]),
1768 786             (1, 131072, 4, [64, 64]),
1769 787             (1, 131072, 8, [64, 64]),
1770 788             (1, 131072, 16, [64, 64]),
1771 789             (1, 262144, 1, [128, 128]),
1772 790             (1, 262144, 4, [64, 64]),
1773 791             (1, 262144, 8, [64, 64]),
1774 792             (1, 262144, 16, [64, 64]),
1775 793             (2, 1024, 16, [128, 128]),
1776 794             (2, 8192, 4, [64, 64]),
1777 795             (2, 8192, 8, [64, 64]),
1778 796             (2, 8192, 16, [64, 64]),
1779 797             (2, 8192, 32, [64, 64]),
1780 798             (2, 16384, 4, [64, 64]),
1781 799             (2, 16384, 16, [64, 64]),
1782 800             (2, 32768, 1, [64, 64]),
1783 801             (2, 32768, 1, [128, 128]),
1784 802             (2, 32768, 4, [64, 64]),
1785 803             (2, 32768, 8, [64, 64]),
1786 804             (2, 65536, 4, [64, 64]),
1787 805             (2, 65536, 8, [64, 64]),
1788 806             (2, 65536, 32, [64, 64]),
1789 807             (2, 131072, 1, [64, 64]),
1790 808             (2, 131072, 1, [128, 128]),
1791 809             (2, 131072, 4, [64, 64]),
1792 810             (2, 131072, 8, [64, 64]),

```

```

1782     811         (2, 131072, 16, [64, 64]),
1783     812         (2, 131072, 32, [64, 64]),
1784     813     ) :
1785     814         BLK_K = 128
1786     815         BLK_V = 64
1787     816         num_warpss = 4
1788     817         num_ctas = 1
1789     818         num_stages = 4
1790     819         maxnreg = None
1791     820     elif key in (
1792     821         (2, 1024, 1, [128, 128]),
1793     822         (2, 2048, 1, [128, 128]),
1794     823         (2, 4096, 1, [64, 64]),
1795     824     ) :
1796     825         BLK_K = 128
1797     826         BLK_V = 64
1798     827         num_warpss = 8
1799     828         num_ctas = 1
1800     829         num_stages = 4
1801     830         maxnreg = None
1802     831     else:
1803     832         raise ValueError(f"Unsupported config for merge_mode_kernel: BTHD
1804     833     ={key} ")
1805     834
1806     835     return {
1807     836         "CHUNK": 64,
1808     837         "FUSE_SP_STATE_UPDATE": True,
1809     838         "H": H,
1810     839         "K": D[0],
1811     840         "V": D[1],
1812     841         "BLK_K": BLK_K,
1813     842         "BLK_V": BLK_V,
1814     843         "num_warpss": num_warpss,
1815     844         "num_stages": num_stages,
1816     845     }
1817     846
1818     847
1819     848
1820     849 @aot_compile_spaces({
1821     850     "merge_mode_kernel": {
1822     851         "signature": merge_mode_kernel_signature,
1823     852         "grid": ["(%K + %BLK_K - 1) / %BLK_K * (%V + %BLK_V - 1) / %
1824     853         "BLK_V)", "%H", "NC"],
1825     854         "triton_algo_infos": [
1826     855             get_merge_mode_kernel_info(B, T, H, K, V)
1827     856             for B,T,H,(K,V) in[
1828     857                 (1, 1024, 8, [64, 64]),
1829     858                 (1, 2048, 1, [128, 128]),
1830     859                 (1, 2048, 16, [128, 128]),
1831     860                 (1, 2048, 4, [64, 64]),
1832     861                 (1, 2048, 8, [128, 128]),
1833     862                 (1, 262144, 32, [128, 128]),
1834     863                 (1, 32768, 1, [128, 128]),
1835     864                 (1, 32768, 4, [128, 128]),
1836     865                 (1, 4096, 1, [128, 128]),
1837     866                 (1, 4096, 1, [64, 64]),
1838     867                 (1, 4096, 4, [128, 128]),
1839     868                 (2, 1024, 16, [128, 128]),
1840     869                 (2, 1024, 8, [128, 128]),
1841     870                 (2, 131072, 1, [128, 128]),
1842     871                 (2, 131072, 1, [64, 64]),
1843     872                 (2, 131072, 16, [128, 128]),
1844     873                 (2, 131072, 16, [64, 64]),
1845                 (2, 131072, 32, [128, 128]),
1846             ]
1847         ]
1848     849
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2699
2700
2701
2702
2703
2704
2705
2706
2707
2708
2709
2710
2711
2712
2713
2714
2715
2716
2717
2718
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
2750
2751
2752
2753
2754
2755
2756
2757
2758
2759
2760
2761
2762
2763
2764
2765
2766
2767
2768
2769
2770
2771
2772
2773
2774
2775
2776
2777
2778
2779
2780
2781
2782
2783
2784
2785
2786
2787
2788
2789
2790
2791
2792
2793
2794
2795
2796
2797
2798
2799
2800
2801
2802
2803
2804
2805
2806
2807
2808
2809
2810
2811
2812
2813
2814
2815
2816
2817
2818
2819
2820
2821
2822
2823
2824
2825
2826
2827
2828
2829
2830
2831
2832
2833
2834
2835
2836
2837
2838
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849
2850
2851
2852
2853
2854
2855
2856
2857
2858
2859
2860
2861
2862
2863
2864
2865
2866
2867
2868
2869
2870
2871
2872
2873
2874
2875
2876
2877
2878
2879
2880
2881
2882
2883
2884
2885
2886
2887
2888
2889
2890
2891
2892
2893
2894
2895
2896
2897
2898
2899
2900
2901
2902
2903
2904
2905
2906
2907
2908
2909
2910
2911
2912
2913
2914
2915
2916
2917
2918
2919
2920
2921
2922
2923
2924
2925
2926
2927
2928
2929
2930
2931
2932
2933
2934
2935
2936
2937
2938
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949
2950
2951
2952
2953
2954
2955
2956
2957
2958
2959
2960
2961
2962
2963
2964
2965
2966
2967
2968
2969
2970
2971
2972
2973
2974
2975
2976
2977
2978
2979
2980
2981
2982
2983
2984
2985
2986
2987
2988
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2999
2999

```

```

1836 874      (2, 131072, 32, [64, 64]),
1837 875      (2, 131072, 4, [128, 128]),
1838 876      (2, 131072, 4, [64, 64]),
1839 877      (2, 131072, 8, [128, 128]),
1840 878      (2, 131072, 8, [64, 64]),
1841 879      (2, 16384, 16, [128, 128]),
1842 880      (2, 16384, 4, [128, 128]),
1843 881      (2, 16384, 8, [64, 64]),
1844 882      (2, 2048, 1, [128, 128]),
1845 883      (2, 2048, 1, [64, 64]),
1846 884      (2, 2048, 16, [128, 128]),
1847 885      (2, 2048, 4, [128, 128]),
1848 886      (2, 32768, 32, [128, 128]),
1849 887      (2, 32768, 32, [64, 64]),
1850 888      (2, 32768, 8, [128, 128]),
1851 889      (2, 4096, 1, [64, 64]),
1852 890      (2, 4096, 4, [64, 64]),
1853 891      (2, 4096, 8, [128, 128]),
1854 892      (2, 65536, 1, [128, 128]),
1855 893      (2, 65536, 1, [64, 64]),
1856 894      (2, 65536, 16, [64, 64]),
1857 895      (2, 65536, 4, [128, 128]),
1858 896      (2, 8192, 1, [128, 128]),
1859 897      (2, 8192, 32, [128, 128])
1860 898      ]
1861 899      ],
1862 900      }
1863 901  })
1864 902  @triton.autotune(
1865 903      configs=[
1866 904          triton.Config({"BLK_K": BLK_K, "BLK_V": BLK_V, "USE_TMA": USE_TMA
1867 905  }, num_warps=num_warps, num_stages=num_stages)
1868 906          for num_warps in [4, 8]
1869 907          for num_stages in [3, 4]
1870 908          for BLK_K in [128]
1871 909          for BLK_V in [128, 64, 32]
1872 910          for USE_TMA in [True, False]
1873 911      ],
1874 912      key=[]
1875 913  )
1876 914  @triton.jit
1877 915  def merge_mode_kernel(
1878 916      q,
1879 917      k,
1880 918      v,
1881 919      g,
1882 920      chunk_state,
1883 921      scale,
1884 922      prev_rank_state_sum,
1885 923      stride_d0_ns_prev_rank_state_sum,
1886 924      chunk_decay,
1887 925      stride_d0_nc_with_ini_chunk_decay,
1888 926      out_0,
1889 927      cu_seqlens,
1890 928      NS,
1891 929      H_MUL_NS,
1892 930      chunk_indices,
1893 931      NC,
1894 932      FUSE_SP_STATE_UPDATE: tl.constexpr,
1895 933      H: tl.constexpr,
1896 934      K: tl.constexpr,
1897 935      V: tl.constexpr,
1898 936      CHUNK: tl.constexpr,
1899 937      BLK_K: tl.constexpr,
2000 938      BLK_V: tl.constexpr,

```

```

1890 938     USE_TMA: tl.constexpr,
1891 939 ) :
1892 940     NUM_BLK_K = (K + BLK_K - 1) // BLK_K
1893 941     NUM_BLK_V = (V + BLK_V - 1) // BLK_V
1894 942     NUM_BLK_KV = NUM_BLK_K * NUM_BLK_V
1895 943
1896 944     i_v, i_h, i_gc = tl.program_id(0), tl.program_id(1), tl.program_id(2)
1897 945     i_s, i_c = tl.load(chunk_indices + i_gc * 2).to(tl.int32), tl.load(
1898 946     chunk_indices + i_gc * 2 + 1).to(tl.int32)
1899 947     bos, eos = tl.load(cu_seqlens + i_s).to(tl.int32), tl.load(cu_seqlens
1900 948     + i_s + 1).to(tl.int32)
1901 949     T = eos - bos
1902 950     # NC = tl.cdiv(T, CHUNK)
1903 951
1904 952     # FIXME: set `i_k` temporarily
1905 953     i_k = 0
1906 954     out_0 = out_0 + (bos * H + i_h) * V
1907 955
1908 956     if USE_TMA:
1909 957         q_desc = tl.make_tensor_descriptor(
1910 958             q + bos * H * K + i_h * K,
1911 959             shape=[T, K],
1912 960             strides=[H * K, 1],
1913 961             block_shape=[CHUNK, BLK_K],
1914 962         )
1915 963         k_desc = tl.make_tensor_descriptor(
1916 964             k + bos * H * K + i_h * K,
1917 965             shape=[T, K],
1918 966             strides=[H * K, 1],
1919 967             block_shape=[CHUNK, BLK_K],
1920 968         )
1921 969         v_desc = tl.make_tensor_descriptor(
1922 970             v + bos * H * V + i_h * V,
1923 971             shape=[T, V],
1924 972             strides=[H * V, 1],
1925 973             block_shape=[CHUNK, BLK_V],
1926 974         )
1927 975         chunk_state_desc = tl.make_tensor_descriptor(
1928 976             chunk_state + (i_gc + i_s).to(tl.int64) * H * K * V + i_h * K
1929 977             * V,
1930 978             shape=[K, V],
1931 979             strides=[V, 1],
1932 980             block_shape=[BLK_K, BLK_V],
1933 981         )
1934 982
1935 983     if FUSE_SP_STATE_UPDATE:
1936 984         prev_rank_state_sum_desc = tl.make_tensor_descriptor(
1937 985             prev_rank_state_sum + i_s * stride_d0_ns_prev_rank_state_sum
1938 986             + i_h * K * V,
1939 987             shape=[K, V],
1940 988             strides=[V, 1],
1941 989             block_shape=[BLK_K, BLK_V],
1942 990         )
1943 991
1944 992     # load `q`: (T, H, K,) => (CHUNK, BLK_K,)
1945 993     if not USE_TMA:
1946 994         cur_q = q + bos * H * K + i_h * K # fmt: skip
1947 995         ptr_q_0 = tl.make_block_ptr(cur_q, (T, K,), (H * K, 1,), (i_c *
1948 996             CHUNK, i_k * BLK_K,), (CHUNK, BLK_K,), (1, 0,)) # fmt: skip
1949 997         blk_q_0 = tl.load(ptr_q_0, boundary_check=(1, 0,)) # fmt: skip
1950 998     else:
1951 999         blk_q_0 = q_desc.load([i_c * CHUNK, i_k * BLK_K])
1952
1953     # load (trans) `k`: (T, H, K,) => (BLK_K, CHUNK, )
1954     if not USE_TMA:
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997

```

```

1944      cur_k = k + bos * H * K + i_h * K  # fmt: skip
1945      ptr_k_1 = tl.make_block_ptr(cur_k, (K, T,), (1, H * K,), (i_k *
1946      BLK_K, i_c * CHUNK,), (BLK_K, CHUNK,), (0, 1,))  # fmt: skip
1947      blk_k_1 = tl.load(ptr_k_1, boundary_check=(0, 1,))  # fmt: skip
1948  else:
1949      blk_k_1 = k_desc.load([i_c * CHUNK, i_k * BLK_K]).trans()
1950  # load 'v': (T, H, V,) => (CHUNK, BLK_V,)
1951  if not USE_TMA:
1952      cur_v = v + bos * H * V + i_h * V  # fmt: skip
1953      ptr_v_2 = tl.make_block_ptr(cur_v, (T, V,), (H * V, 1,), (i_c *
1954      CHUNK, i_v * BLK_V,), (CHUNK, BLK_V,), (1, 0,))  # fmt: skip
1955      blk_v_2 = tl.load(ptr_v_2, boundary_check=(1, 0,))  # fmt: skip
1956  else:
1957      blk_v_2 = v_desc.load([i_c * CHUNK, i_v * BLK_V])
1958  # call_external_func: 'chunk_local_cumsum' to pre-compute g
1959  # load 'g': (T, H,) => (CHUNK,)
1960      cur_g = g + bos * H + i_h * 1  # fmt: skip
1961      ptr_g_3 = tl.make_block_ptr(cur_g, (T,), (H,), (i_c * CHUNK,), (CHUNK
1962      ,), (0,))  # fmt: skip
1963      blk_g_3 = tl.load(ptr_g_3, boundary_check=(0,))  # fmt: skip
1964  # load 'chunk_state': (NC_WITH_INI, H, K, V,) => (BLK_K, BLK_V,)
1965  if not USE_TMA:
1966      cur_chunk_state = chunk_state + (i_gc + i_s).to(tl.int64) * H * K
1967      * V + i_h * K * V  # fmt: skip
1968      ptr_chunk_state_4 = tl.make_block_ptr(cur_chunk_state, (K, V,), (V,
1969      1,), (i_k * BLK_K, i_v * BLK_V,), (BLK_K, BLK_V,), (1, 0,))  # fmt
1970  : skip
1971      blk_chunk_state_4 = tl.load(ptr_chunk_state_4, boundary_check=(1,
1972      0,))  # fmt: skip
1973  else:
1974      blk_chunk_state_4 = chunk_state_desc.load([i_k * BLK_K, i_v *
1975      BLK_V])
1976  # comments not available for op 'if_beg'
1977  if FUSE_SP_STATE_UPDATE:
1978      # load 'prev_rank_state_sum': (NS, H, K, V,) => (BLK_K, BLK_V,)
1979  if not USE_TMA:
1980      cur_prev_rank_state_sum = prev_rank_state_sum + i_s *
1981      stride_d0_ns_prev_rank_state_sum + i_h * K * V  # fmt: skip
1982      ptr_prev_rank_state_sum_5 = tl.make_block_ptr(
1983      cur_prev_rank_state_sum, (K, V,), (V, 1,), (i_k * BLK_K, i_v * BLK_V
1984      ,), (BLK_K, BLK_V,), (1, 0,))  # fmt: skip
1985      blk_prev_rank_state_sum_5 = tl.load(ptr_prev_rank_state_sum_5
1986      , boundary_check=(1, 0,))  # fmt: skip
1987  else:
1988      blk_prev_rank_state_sum_5 = prev_rank_state_sum_desc.load([
1989      i_k * BLK_K, i_v * BLK_V])
1990  # load 'chunk_decay': (NC_WITH_INI, H,) => (,,
1991      cur_chunk_decay = chunk_decay + (i_gc + i_s).to(tl.int64) *
1992      stride_d0_nc_with_ini_chunk_decay + i_h * 1  # fmt: skip
1993      ptr_chunk_decay_6 = cur_chunk_decay  # fmt: skip
1994      blk_chunk_decay_6 = tl.load(ptr_chunk_decay_6)
1995  # to: ('BLK_K', 'BLK_V') => ('BLK_K', 'BLK_V')
1996      blk_prev_rank_state_sum_5_float32_7 = blk_prev_rank_state_sum_5.
1997      to(tl.float32)
1998  # mul: (), ('BLK_K', 'BLK_V') => ('BLK_K', 'BLK_V')
1999      mul_tensor_8 = blk_chunk_decay_6 *
2000      blk_prev_rank_state_sum_5_float32_7
2001  # to: ('BLK_K', 'BLK_V') => ('BLK_K', 'BLK_V')
2002      mul_tensor_8_bfloat16_9 = mul_tensor_8.to(tl.bfloat16)

```

```

1998 1046      # add: ('BLK_K', 'BLK_V'), ('BLK_K', 'BLK_V') => ('BLK_K', 'BLK_V')
1999 1047      ')
2000 1048      add_tensor_10 = mul_tensor_8_bfloat16_9 + blk_chunk_state_4
2001 1049      # comments not available for op 'bind_var'
2002 1050      blk_chunk_state_4 = add_tensor_10
2003 1051      # comments not available for op 'end_if'
2004 1052      # to: ('BLK_K', 'BLK_V') => ('BLK_K', 'BLK_V')
2005 1053      blk_chunk_state_4_bfloat16_11 = blk_chunk_state_4.to(tl.bfloat16)
2006 1054      # matmul: ('CHUNK', 'BLK_K'), ('CHUNK', 'BLK_K') => ('CHUNK', 'CHUNK')
2007 1055      ')
2008 1056      matmul_12 = tl.dot(blk_q_0, blk_k_1).to(blk_q_0.dtype)
2009 1057      # tril: ('CHUNK', 'CHUNK') => ('CHUNK', 'CHUNK')
2010 1058      tril_13 = tl.where(tl.arange(0, CHUNK)[:, None] >= tl.arange(0, CHUNK)
2011 1059      )[None, :], matmul_12, 0)
2012 1060      # unsqueeze: ('CHUNK',) => ('CHUNK', 1)
2013 1061      unsqueeze_14 = blk_g_3[:, None]
2014 1062      # unsqueeze: ('CHUNK',) => (1, 'CHUNK')
2015 1063      unsqueeze_1_15 = blk_g_3[None, :]
2016 1064      # sub: ('CHUNK', 1), (1, 'CHUNK') => ('CHUNK', 'CHUNK')
2017 1065      sub_16 = unsqueeze_14 - unsqueeze_1_15
2018 1066      # exp: ('CHUNK', 'CHUNK') => ('CHUNK', 'CHUNK')
2019 1067      exp_17 = tl.exp(sub_16)
2020 1068      # mul: ('CHUNK', 'CHUNK'), ('CHUNK', 'CHUNK') => ('CHUNK', 'CHUNK')
2021 1069      mul_18 = tril_13 * exp_17
2022 1070      # to: ('CHUNK', 'CHUNK') => ('CHUNK', 'CHUNK')
2023 1071      mul_18_bfloat16_19 = mul_18.to(tl.bfloat16)
2024 1072      # matmul: ('CHUNK', 'CHUNK'), ('CHUNK', 'BLK_V') => ('CHUNK', 'BLK_V')
2025 1073      ')
2026 1074      matmul_1_20 = tl.dot(mul_18_bfloat16_19, blk_v_2).to(
2027 1075      mul_18_bfloat16_19.dtype)
2028 1076      # matmul: ('CHUNK', 'BLK_K'), ('BLK_K', 'BLK_V') => ('CHUNK', 'BLK_V')
2029 1077      ')
2030 1078      matmul_2_21 = tl.dot(blk_q_0, blk_chunk_state_4_bfloat16_11).to(
2031 1079      blk_q_0.dtype)
2032 1080      # exp: ('CHUNK',) => ('CHUNK',)
2033 1081      exp_1_22 = tl.exp(blk_g_3)
2034 1082      # unsqueeze: ('CHUNK',) => ('CHUNK', 1)
2035 1083      unsqueeze_2_23 = exp_1_22[:, None]
2036 1084      # mul: ('CHUNK', 'BLK_V'), ('CHUNK', 1) => ('CHUNK', 'BLK_V')
2037 1085      mul_1_24 = matmul_2_21 * unsqueeze_2_23
2038 1086      # add: ('CHUNK', 'BLK_V'), ('CHUNK', 'BLK_V') => ('CHUNK', 'BLK_V')
2039 1087      add_25 = matmul_1_20 + mul_1_24
2040 1088      # mul: ('CHUNK', 'BLK_V'), () => ('CHUNK', 'BLK_V')
2041 1089      mul_2_26 = add_25 * scale
2042 1090      # store => ('CHUNK', 'BLK_V')
2043 1091      # assume output layout: [T, H, V]
2044 1092      out_0_ty = out_0.dtype.element_ty
2045 1093      ptr_out_0 = tl.make_block_ptr(out_0, (T, V,), (H * V, 1,), (i_c *
2046 1094      CHUNK, i_v * BLK_V,), (CHUNK, BLK_V,), (1, 0)) # fmt: skip
2047 1095      tl.store(ptr_out_0, mul_2_26.to(out_0_ty), boundary_check=(1, 0))
2048 1096      def launch_merge_mode(
2049 1097      chunk_decay: torch.Tensor,
2050 1098      v: torch.Tensor,
2051 1099      k: torch.Tensor,
2052 1100      q: torch.Tensor,
2053 1101      g: torch.Tensor,
2054 1102      prev_rank_state_sum: torch.Tensor,
2055 1103      chunk_state: torch.Tensor,
2056 1104      scale,
2057 1105      cu_seqlens: torch.IntTensor,
2058 1106      cached_results: dict,
2059 1107      fuse_sp_update: bool,
2060 1108      ) -> torch.Tensor:
2061 1109      T, H, K, V = *k.shape, v.shape[-1]

```

```

20521103
20531104     if scale is None:
20541105         scale = k.shape[-1] ** -0.5
20551106
20561107     # TODO: now fix chunk_size to 64
20571108     CHUNK = 64
20581109     chunk_indices = prepare_chunk_indices(cu_seqlens, CHUNK)
20591110     chunk_offsets = prepare_chunk_offsets(cu_seqlens, CHUNK)
20601111     chunk_offsets_ini = prepare_chunk_offsets_with_ini(cu_seqlens, CHUNK)
20611112     NS, NC = len(cu_seqlens) - 1, chunk_indices.shape[0]
20621113     NC_WITH_INI = NC + NS
20631114     out = torch.empty_like(v)
20641115     use_aot = os.environ.get('FORGE_USE_AOT', '0')
20651116     FORGE_USE_AOT = True if use_aot.lower() in ['1', 'true', 'yes'] else
20661117     None
20671118
20681119     def alloc_fn(size: int, alignment: int, stream: int):
20691120         return torch.empty(size, device='cuda', dtype=torch.int8)
20701121
20711122     triton.set_allocator(alloc_fn)
20721123
20731124     def grid(meta):
20741125         BLK_K = meta['BLK_K']
20751126         BLK_V = meta['BLK_V']
20761127         NUM_BLK_K = (K + BLK_K - 1) // BLK_K
20771128         NUM_BLK_V = (V + BLK_V - 1) // BLK_V
20781129         NUM_BLK_KV = NUM_BLK_K * NUM_BLK_V
20791130         H_MUL_NS = H * NS
20801131         return (NUM_BLK_KV, H, NC, )
20811132
20821133     if FORGE_USE_AOT is None:
20831134         merge_mode_kernel[grid](
20841135             chunk_decay=chunk_decay,
20851136             stride_d0_nc_with_ini_chunk_decay=H,
20861137             v=v,
20871138             k=k,
20881139             q=q,
20891140             g=g,
20901141             prev_rank_state_sum=prev_rank_state_sum,
20911142             stride_d0_ns_prev_rank_state_sum=H*K*V,
20921143             chunk_state=chunk_state,
20931144             scale=scale,
20941145             cu_seqlens=cu_seqlens,
20951146             H=H,
20961147             NS=NS,
20971148             K=K,
20981149             V=V,
20991150             CHUNK=CHUNK,
21001151             H_MUL_NS=H * NS,
21011152             out_0=out,
21021153             chunk_indices=chunk_indices,
21031154             NC=NC,
21041155             FUSE_SP_STATE_UPDATE=fuse_sp_update,
21051156         )
21061157
21071158     else:
21081159
21091160         from forge.aot_utils import forge_aot_ops
21101161
21111162         algo_info = forge_aot_ops.merge_mode_kernel__triton_algo_info_t()
21121163         for _k, _v in get_merge_mode_kernel_info(NS, T, H, K, V).items():
21131164             setattr(algo_info, _k, _v)
21141165         forge_aot_ops.merge_mode_kernel(
21151166             0, # torch.cuda.current_stream().cuda_stream,
21161167             q.data_ptr(), # q

```

```

2106
2107     k.data_ptr(),  # k
2108     v.data_ptr(),  # v
2109     g.data_ptr(),  # g
2110     chunk_state.data_ptr(),  # chunk_state
2111     scale,  # scale
2112     prev_rank_state_sum.data_ptr() if prev_rank_state_sum else 0,
2113     # prev_rank_state_sum
2114     H*K*V,  # stride_d0_ns_prev_rank_state_sum
2115     chunk_decay.data_ptr() if chunk_decay else 0,  # chunk_decay
2116     H,  # stride_d0_nc_with_ini_chunk_decay
2117     out.data_ptr(),  # out_0
2118     cu_seqlens.data_ptr(),  # cu_seqlens
2119     NS,  # NS
2120     H * NS,  # H_MUL_NS
2121     chunk_indices.data_ptr(),  # chunk_indices
2122     NC,  # NC
2123     algo_info,
2124     )
2125
2126     cached_results.update({})
2127     return out
2128
2129 def fused_op():
2130     cached_results = {}
2131     # AUTO: precompute decay outside
2132     g_cumsum = chunk_local_cumsum(g[None, ...], 64, cu_seqlens=cu_seqlens
2133     ).squeeze(0)
2134     updated_states = launch_fuse_chunk_decay(
2135         g=g_cumsum,
2136         prev_s=prev_s,
2137         v=v,
2138         k=k,
2139         cu_seqlens=cu_seqlens,
2140         cached_results=cached_results,
2141         dist_scan=dist_scan,
2142         lazy_update=lazy_update,
2143     )
2144     chunk_state = updated_states['chunk_state']
2145     prev_rank_state_sum = updated_states.get('prev_rank_state_sum', None)
2146     chunk_decay = updated_states.get('chunk_decay', None)
2147     o = launch_merge_mode(
2148         chunk_decay=chunk_decay,
2149         v=v,
2150         k=k,
2151         q=q,
2152         g=g_cumsum,
2153         prev_rank_state_sum=prev_rank_state_sum,
2154         chunk_state=chunk_state,
2155         scale=scale,
2156         cu_seqlens=cu_seqlens,
2157         cached_results=cached_results,
2158         fuse_sp_update=fuse_sp_update,
2159     )
2160
2161     return o, chunk_state

```

Listing 2: Generated Scalar GLA kernel by Forge

G.2 IMPLEMENTATION FOR DELTANET

```

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

```

```

2160      5     U = T @ (v * b[..., None])
2161      6     W = T @ (k * b[..., None])
2162      7     # Explicitly cache results for reuse in other phases
2163      8     forge.cache_result(U, "u")
2164      9     forge.cache_result(W, "w")
2165     10    S = k.T @ U
2166     11    return S
2167
2168  13  def decay_mode_deltanet(prev_s: Tensor, k: Tensor, w: Tensor, chunk_state
2169      : Tensor) -> Tensor:
2170      """
2171      Inter-Chunk State Propagation
2172      """
2173      # Calculate decay matrix based on cached W
2174      17  decay = forge.identity(k, k.size(1)) - k.T @ w
2175      18  return decay @ prev_s.to(decay.dtype) + chunk_state
2176
2177  21  def merge_mode_deltanet(q: Tensor, k: Tensor, u: Tensor, w: Tensor,
2178      chunk_state: Tensor, scale: Tensor) -> Tensor:
2179      """
2180      Output Merging
2181      Note: 'u' and 'w' are automatically injected from the cached results
2182      """
2183      26  new_v = u - w @ chunk_state
2184      27  return ((q @ k.T).tril(0) @ new_v + q @ chunk_state) * scale
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213

```

Listing 3: Forge Implementation of DeltaNet with Intermediate Result Caching