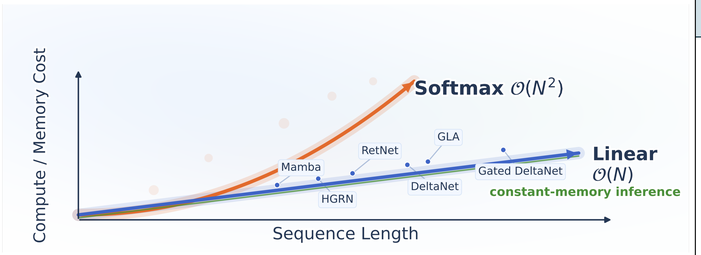
Flex Linear Attention: Compiling a Unified Abstraction into Scalable Kernels for Linear Attention

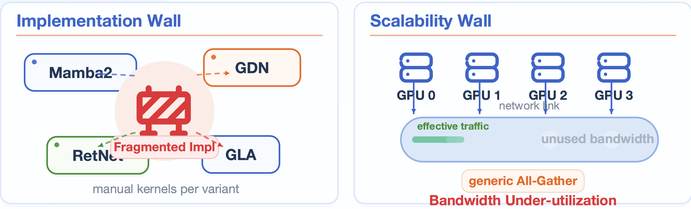
ByteDance | Seed
Shanghai Jiao Tong University
上海人工智能实验室 Shanghai Artificial Intelligence Laboratory
Haojie Duanmu, Size Zheng, Ningxin Zheng, Jianqiao Lu, Xuegui Zheng, Xingcheng Zhang, Li-wen Chang, Xin Liu, Dahua Lin

Observation & Motivation

The Promise: Linear Attention Is Highly Efficient



The Reality: Implementation & Scalability Wall



- Unlike transformer with Flash-Attn as de-facto impl, the linear attention landscape is **highly fragmented**.
- Researchers suffer from kernel tuning and severe bandwidth under-utilization (e.g., All-Gather) when scaling with context parallel.

The Insight: Unification through structure invariance

- **Associative Recurrence:** Despite diverse decay mechanisms (scalar, vector, matrix), all linear attn rely on associativity.
- **Algebraic Unrolling:** *Any* sequential update can be formally decoupled into **independent local compute** and **dependent state propagation**—**Naturally cross device**.
- **Unified Mapping:** Diverse models therefore map identically to 3-Phase decomposition, unlocking a universal **tile-level compute-comm fusion** strategy without model-specific heuristics.

$$S_{[i]} = \text{Chunk\_Decay} \cdot S_{[i-1]} + \text{Local\_Update}$$

Can we provide a solution to bridge the gap between the rapid evolution of linear attention algorithms and the difficulty of developing scalable kernels?

Unified 3-Phase Programming Abstraction

Naïve Torch Implementation

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

FlexLA User Interface

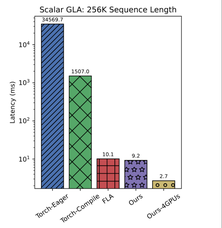
```
def chunk_mode(k, v, g):
  g_cumsum, g_sum = g.cumsum(0), g.sum(0)
  g_cumsum        = (g_sum - g_cumsum[None, :]).exp()
  k_decay         = (k.T * g_cumsum.exp())
  return k_decay @ v

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


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

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

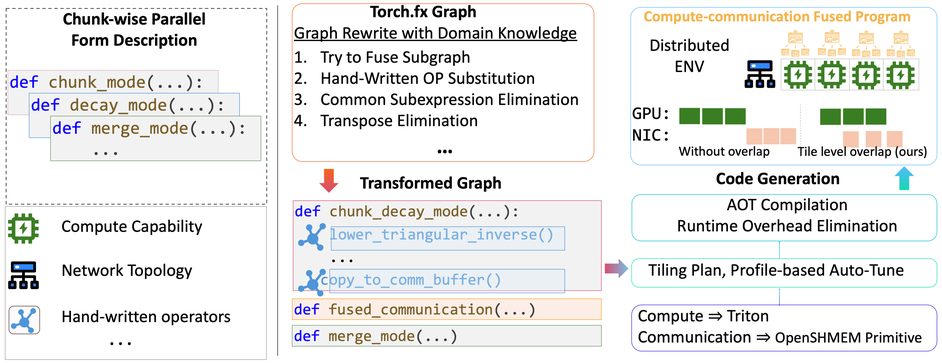
Compilation results



Lines of Code	
FlexLA	50+
Generated Triton	1200+

1. `chunk_mode`: How local inputs are processed into a chunk state? ⇒ main computation.
2. `decay_mode`: How chunk-states propagate? ⇒ cross-device communication in this stage.
3. `merge_mode`: How the global state and local inputs are combined? ⇒ overlap with comm.

Compilation Pipeline & System Optimizations



- **Domain-Specific Rewrite:** substitute complex math to tuned Triton templates.
- **Comm-Comp Fusion:** Overlaps fine-grained P2P comm with computation, no collective.
- **Optimized Launch:** AOT compilation + static dispatch eliminates triton launch overhead.

Experiment

Single Device Kernel

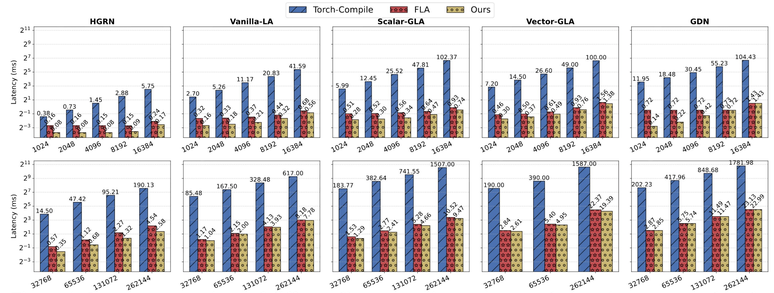
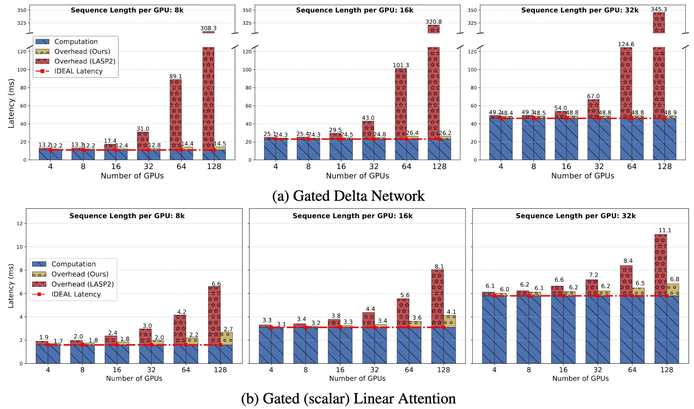


Figure 4: Latency comparison of different linear attention variants under varying sequence lengths on a single H100 GPU. Each subplot corresponds to one model, with the top row showing short to medium sequences (1K–16K) and the bottom row showing long sequences (32K–256K).

Performance: FlexLA matches FLA's efficiency across diverse linear attn variants!

Distributed Weak Scaling



(a) Gated Delta Network

(b) Gated (scalar) Linear Attention

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

Near-linear Scaling: Generated CP kernels scale to 16M tokens across 128 GPUs!

Why it scales?

1.Communication are tailored to state propagation instead of generic All-Gather.

2.Tile-level fusion overlaps communication with chunk computation.