项目复现：Gated Linear Attention Transformers with Hardware-Efficient Training（ICML 2024）

摘要

基于注意力机制的大型语言模型（LLM）在自然语言处理等领域表现出色，但传统注意力机制的二次复杂度限制了其应用。为了解决这一问题，研究者提出了线性注意力机制，通过巧妙设计将复杂度降低至线性级别。Gated Linear Attention (GLA) 作为线性注意力机制的改进，进一步提升了模型性能。GLA 引入了门控机制，增强了模型对输入序列的建模能力，使其更关注关键信息。将 GLA 应用于 Transformer 架构得到的 GLA Transformer，在中等规模语言模型任务上表现出色，尤其在处理长序列方面具有显著优势。GLA Transformer 训练速度快、并行化程度高，具有更强的长程依赖建模能力。

本次复现以 parallel scan 的形式实现了 GLA 的 cuda kernel，保证效率的同时使整个 kernel 更加具有可读性。同时还实现了 GLA Transformer 的 chunk wise 计算形式，将并行和串行的计算统一为 chunk 的形式。并且，进一步实现了高效的隐状态管理，使隐藏状态的重置转移继承等等操作更加方便。这些代码能够让研究者在 GLA 的基础上进行新架构的研究更加方便和直观。

并且，本次复现还在 ARtest 和小说文本上进行了训练和测试。在对 GLA 进行微改进后，在 ARtest 上得到了和类似略优的结果，在小说文本上得到了更低的 PPL，说明了本次复现代码的可用性。

关键词：大模型；自注意力；门控线性注意力

1 引言

基于注意力机制的大型语言模型（LLM）在自然语言处理、计算机视觉等领域展现出前所未有的强大能力。然而，传统注意力机制的二次复杂度使其在处理长文本时面临效率瓶颈，严重制约了模型的应用范围。为了克服这一难题，研究者们提出了线性注意力机制，通过巧妙的设计，将注意力计算的复杂度降低至线性级别，从而实现高效并行训练和线性时间复杂度的推理。

但是线性注意力通常会舍弃较多的长期信息，这导致模型对长程依赖的学习能力降低。目前，学术界研究的重点便是如何在保障架构能够高效并行训练，高效串行推理的同时，还具有较好的长期记忆能力。在降低计算资源需求的同时，保证与经典 attention 相同的模型表现。

而 Gated Linear Attention 就是尝试结合这三种特性的一次非常好的尝试。因此，复现这个工作相当有意义。

2 相关工作

2.1 自注意力机制

自注意力机制（Self-Attention）是一种在深度学习模型中广泛应用的机制，尤其在处理序列数据时表现出色。[2] 它通过计算序列中每个元素与其他元素之间的关系，来捕捉序列内部的复杂依赖关系。这种机制使得模型能够在处理序列数据时，更加灵活地关注序列中的不同部分，从而更好地提取特征。

Self-Attention 的核心思想是让模型自己去学习序列中不同部分之间的关系。具体来说，对于一个输入序列，Self-Attention 会为序列中的每个元素计算一个 Query 向量、Key 向量和 Value 向量。Query 向量表示当前元素，Key 向量表示序列中的其他元素，Value 向量表示序列中每个元素的实际值。通过计算 Query 向量和 Key 向量之间的相似度（通常使用点积），得到一个注意力权重分布。然后，将 Value 向量按照注意力权重进行加权求和，得到该元素的输出表示。

假设输入序列为 X = x1, x2, ..., xn，其中 xi 是序列中的第 i 个元素。对于序列中的第 i 个元素，其 Self-Attention 的计算过程可以表示为：

$$Q = XW_Q, K = XW_K, V = XW_V.$$

$$Attention(Q, K, V) = softmax(\frac{QK}{\sqrt{d}})V.$$

其中，W_Q, W_K, W_V 是可学习的参数矩阵。Self-Attention 能够捕捉序列中任意两个位置之间的依赖关系，而不需要事先假设一个固定的窗口大小。Self-Attention 的计算可以高度并行化，从而提高计算效率。Attention 权重可以直观地显示模型在处理序列时关注了哪些部分，具有一定的可解释性。

然而，其固有的二次复杂度问题限制了其在处理长序列数据时的应用。在计算注意力分数时，需要计算每个元素与其他所有元素的相似度，这导致了注意力矩阵的规模与序列长度的平方成正比。对于长序列，注意力矩阵的规模会变得非常大，导致计算资源消耗巨大，甚至超出硬件的承载能力，同时也导致模型的推理成本在长文本下变得非常巨大。

2.2 线性注意力

线性注意力是一种旨在降低传统自注意力机制（Self-Attention）二次复杂度，提升计算效率的变体 [1]。它通过一系列优化手段，将注意力机制的计算复杂度从 $O(n^2)$ 降低到 $O(n)$，使得模型能够更有效地处理长序列数据。传统的自注意力机制需要计算每个 token 与其他所有 token 的相似度，这导致了二次复杂度的计算量，在处理长序列时效率低下。长序列数据在自然语言处理、计算机视觉等领域非常常见，但传统的自注意力机制难以有效地处理这些数据。

线性注意力通常使用更简单的线性函数来近似 Softmax 操作，从而避免了 Softmax 操作带来的高计算成本。同时通过将 Query、Key 和 Value 映射到更低维的空间以减少了计算量。在使用线性函数代替注意力机制后，数学上就不必先计算 QK 相似性，而可以先计算更小的 KV 矩阵，不仅仅减少了计算量，而且带来了一种循环形式的 Attention：

$$S_t = S_{t-1} + k_t^T v_t, o_t = q_t S_t.$$

线性注意力显著降低了计算复杂度，使得模型能够更快速地处理长序列数据。由于计算量减少，线性注意力对内存的需求也更低，通常内存需求不随着推理长度增加。但是，相比于传统的自注意力机制，线性注意力的表达能力可能有所下降，这主要是因为近似 Softmax 操作和维度降低等操作会损失一些信息。

3 本文方法

3.1 GLA 概述

Gated Linear Attention (GLA) 作为线性注意力机制的一种改进，进一步提升了模型的性能 [3]。GLA 通过引入门控机制，在不破坏线性注意力原有优点的基础上，增强了模型对输入序列的建模能力。门控机制能够自适应地调整不同信息的权重，使得模型更加关注关键信息，从而提高模型的表达能力和泛化能力。

当将 GLA 应用于 Transformer 架构时，得到的 GLA Transformer 在多个方面展现出优异的性能。在中等规模语言模型任务上，GLA Transformer 与 LLaMA、RetNet、Mamba 等先进模型相比具有相当的竞争力。特别是在处理长序列方面，GLA Transformer 表现出显著优势，能够从较短的 2K 训练数据泛化到超过 20K 的序列，同时保持较低的困惑度，这表明 GLA Transformer 具有更强的长程依赖建模能力。此外，GLA Transformer 在训练速度上也具有显著优势，其并行化程度更高，吞吐量更大，能够在更短的时间内完成模型训练。

3.2 GLA Kernel

本次要实现的 GLA 的 kernel 实现如图 1所示，使用类似 RNN 的循环形式来加速计算。并且在线性注意力的基础上，引入了输入依赖的 G 门对隐藏状态 h 进行选择性的遗忘.

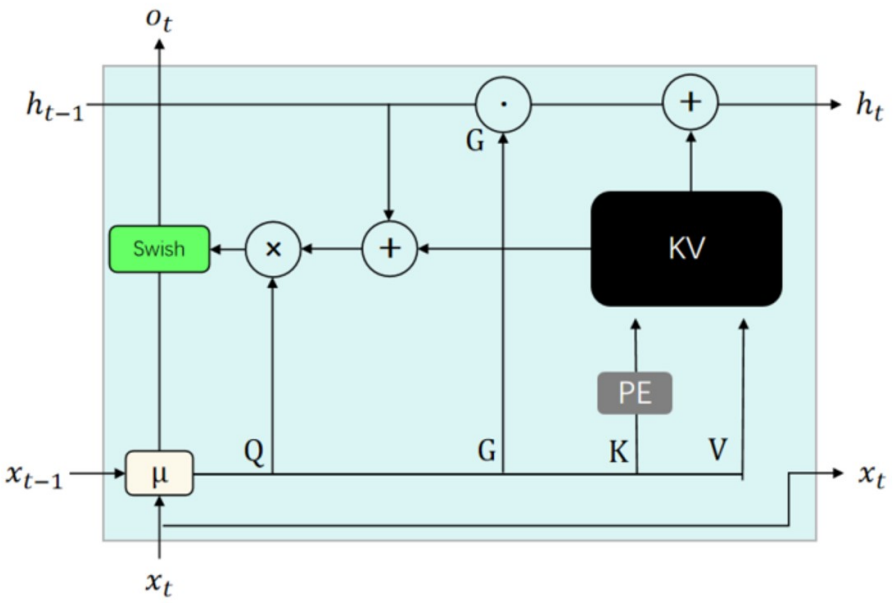

图 1. GLA Kernel

其计算方式如下：

$$S_t = g_t S_{t-1} + k_t^T v_t, o_t = q_t S_t.$$

3.3 Chunk wise 形式

Chunk Wise 的计算形式将架构的并行训练和串行推理形式进行了统一，优化了信息缓存和计算的方式。如图 2所示。

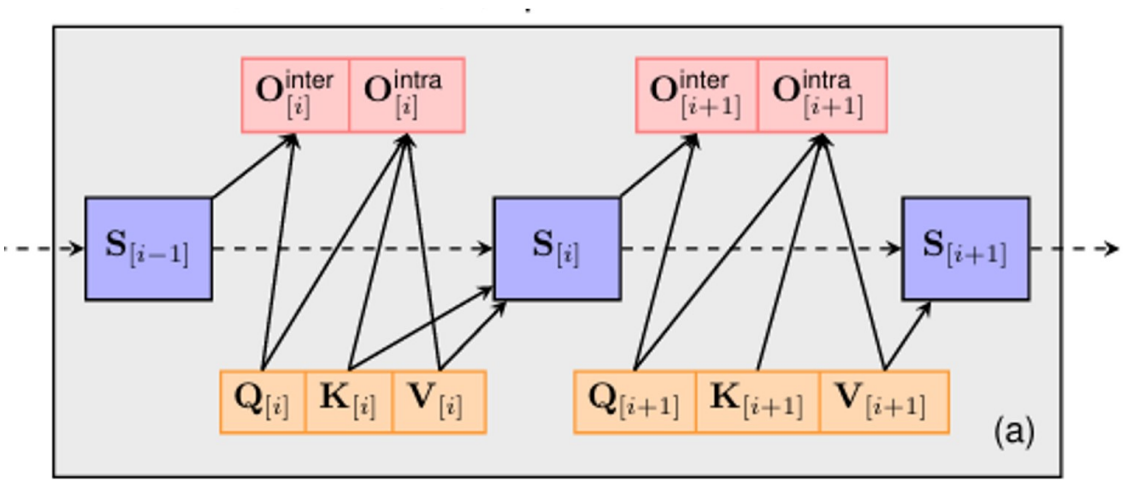

图 2. Chunk wise 形式

4 复现细节

4.1 与已有开源代码对比

目前，GLA 在 FLA2 库中有开源实现，但其基于 Token-Parallel 的方式进行实现，该种实现适合工业部署，但导致架构难以进行优化更改。本次实现将基于 Parallel-Scan 的方式来实现，保持基本同级别的性能表现的同时提高代码的可拓展性，可读性高，并给予用户对 CUDA kernel 内部数据精度更多的控制。并且，本次复现也会对 chunk-wise 隐状态存储进行更好的封装，方便用户调用。总的来说，原代码更适用于工业部署，本次复现的代码更适用于学术研究。

4.2 cuda kernel 实现

本次实现采用 Parallel Scan 方式，对输入序列按 token 逐个扫描，并借助 GLA 架构在 element wise 上无前后依赖的特性来进行高效计算。图 3， 4是核函数的输入和定位代码。图 5是基于 parallel scan 的前线计算代码。

```
template <typename dtype>
__global__ void kernel_forward(const int num_token, const int dim_feature,
                               const dtype * __restrict__ const q_global,
                               const dtype * __restrict__ const k_global,
                               const dtype * __restrict__ const g_global,
                               const dtype * __restrict__ const v_global,
                                     dtype * __restrict__ const y_global,
                                     dtype * __restrict__ const s_global)
{
    const int idx_batch   = blockIdx.x;
    const int idx_head    = blockIdx.y;
    const int idx_feature = threadIdx.x;
    __shared__ ctype q[siz_head], k[siz_head], g[siz_head];
```

图 3. 前向 kernel 输入

```
// s (siz_batch, dim_feature=(num_head, siz_head), siz_head)
const int bgn_s =  idx_batch     * dim_feature           * siz_head +
                   idx_head                   * siz_head * siz_head +
                   0                                     * siz_head +
                   idx_feature;

// load s0 to reg, align state (dim_k=-2)
ctype s[siz_head] = {0};
#pragma unroll
for(int index = 0; index < siz_head; index++){
    s[index] = ctype(s_global[bgn_s + index * siz_head]);
}

// q k v y (siz_batch, num_token, dim_feature=(num_head, siz_head))
const int bgn_seq_feature =  idx_batch      * num_token * dim_feature           +
                             0                          * dim_feature           +
                             idx_head                                 * siz_head +
                             idx_feature;
const int end_seq_feature = (idx_batch + 1) * num_token * dim_feature           +
                             0                          * dim_feature           +
                             idx_head                                 * siz_head +
                             idx_feature;
```

图 4. 定位计算

```
// from token0 to tokenN , calculate y
for (int idx_token_feature = bgn_seq_feature;
         idx_token_feature < end_seq_feature;
         idx_token_feature +=    dim_feature){

    __syncthreads();  // load global data to shared mem and reg
    q[idx_feature]    = ctype(q_global[idx_token_feature]);
    k[idx_feature]    = ctype(k_global[idx_token_feature]);
    g[idx_feature]    = ctype(g_global[idx_token_feature]);
    const ctype v_val = ctype(v_global[idx_token_feature]);
    __syncthreads();
    ctype y_val = 0;

    #pragma unroll  // calculate y_t
    for(int index = 0; index < siz_head; index++){
        s[index] = g[index] * s[index] + k[index] * v_val;
        y_val += q[index] * s[index];
    }

    y_global[idx_token_feature] = dtype(y_val); //return y_token_feature
}
```

图 5. 前向核心计算

图 6， 7， 8是对应的反向梯度计算的代码，其将 KV 和 QG 的梯度分两次计算，并利用 KV 梯度的中间结果来加速 G 梯度的计算。

```
void cuda_backward(int siz_batch, int num_token, int dim_feature, int num_head,
                   ftype *q, ftype *k, ftype *g, ftype *v, ftype *s,
                   ftype *grad_y, ftype *grad_s,
                   ftype *grad_q, ftype *grad_k, ftype *grad_g, ftype *grad_v){

    dim3 grid_dim(siz_batch, num_head);
    dim3 block_dim(siz_head);

    ctype *cache_grad_gkv, *cache_cumprod_g;
    cudaMalloc(&cache_grad_gkv, siz_batch * num_token * dim_feature * sizeof(ctype));
    cudaMalloc(&cache_cumprod_g, siz_batch * num_token * dim_feature * sizeof(ctype));

    kernel_backward<<<grid_dim, block_dim>>>(num_token, dim_feature,
                                             q, k, g, v, s,
                                             grad_y, grad_s,
                                             grad_q, grad_k, grad_g, grad_v,
                                             cache_grad_gkv, cache_cumprod_g);

    cudaFree(cache_grad_gkv);
    cudaFree(cache_cumprod_g);
}
```

图 6. 反向 kernel 输入

```
// calculate grad k v, and cache the data needed by grad g
#pragma unroll
for (int index = 0; index < siz_head; index++){

    ctype yq_align_k = cache_yq_align_k[index] + grad_y_val    * q[index];
    ctype yq_align_v = cache_yq_align_v[index] + grad_y[index] * q_val   ;

    grad_k_val += yq_align_v * v[index];
    grad_v_val += yq_align_k * k[index];
    grad_k_val += grad_s_align_v[index] * cumprod_g_align_v_val    * v[index];
    grad_v_val += grad_s_align_k[index] * cumprod_g_align_k[index] * k[index];

    grad_g_val += cache_yq_align_v[index] * v[index];
    cache_yq_align_k[index] = yq_align_k * g[index];
    cache_yq_align_v[index] = yq_align_v * g_val   ;

    cumprod_g_align_k[index] *= g[index];
}
cumprod_g_align_v_val *= g_val;
```

图 7. KV 梯度计算

```
    // calculate grad q g
    #pragma unroll
    for (int index = 0; index < siz_head; index++){

        grad_s_dg_val += grad_s_align_v[index] * cumprod_g_val * s[index];

        extra_grad_g_val += grad_y[index] * s[index];

        s[index] = g_val * s[index] + k_val * v[index];

        grad_q_val += grad_y[index] * s[index];
    }

    // return grad_g_token_feature
    grad_g_global[idx_token_feature] = dtype((grad_y_dg_val + grad_s_dg_val) / (g_val + 1e-16));

    // combine grad gN+1 on grad_y
    grad_y_dg_val = grad_y_dg_val + cache_gkv_val - extra_grad_g_val * q_val * g_val;

    // return grad_q_token_feature
    grad_q_global[idx_token_feature] = dtype(grad_q_val);
```

图 8. QG 梯度计算

图 9是 kernel 的编译结果，图 10是 kernel 和 naive 形式的 torch 代码在 fp64 下的精度差比较，可见差距非常小，并且也通过了 torch 自带的梯度检测。证明了 kernel 的可用性。

```
Using /home/xrete/.cache/torch_extensions/py38_cu121 as PyTorch extensions root...
Creating extension directory /home/xrete/.cache/torch_extensions/py38_cu121/gated_att_1...
Detected CUDA files, patching ldflags
Emitting ninja build file /home/xrete/.cache/torch_extensions/py38_cu121/gated_att_1/build.ninja...
Building extension module gated_att_1...
Allowing ninja to set a default number of workers... (overridable by setting the environment variable MAX_JOBS=N)
[1/3] c++ -MMD -MF main.o.d -DTORCH_EXTENSION_NAME=gated_att_1 -DTORCH_API_INCLUDE_EXTENSION_H -DPYBIND11_COMPILER_TYPE=\"_gcc\" -DPYBIND11_
[2/3] /usr/local/cuda/bin/nvcc  -DTORCH_EXTENSION_NAME=gated_att_1 -DTORCH_API_INCLUDE_EXTENSION_H -DPYBIND11_COMPILER_TYPE=\"_gcc\" -DPYBI
ptxas info    : 10 bytes gmem, 80 bytes cmem[4]
ptxas info    : Compiling entry function '_Z15kernel_backwardIdEviiPKT_S2_S2_S2_S2_S2_PS0_S3_S3_S3_S3_PdS4_' for 'sm_89'
ptxas info    : Function properties for _Z15kernel_backwardIdEviiPKT_S2_S2_S2_S2_S2_PS0_S3_S3_S3_S3_PdS4_
    2168 bytes stack frame, 3104 bytes spill stores, 3896 bytes spill loads
ptxas info    : Used 255 registers, used 1 barriers, 2168 bytes cumulative stack size, 2560 bytes smem, 464 bytes cmem[0], 8 bytes cmem[2]
ptxas info    : Compiling entry function '_Z14kernel_forwardIdEviiPKT_S2_S2_S2_PS0_S3_' for 'sm_89'
ptxas info    : Function properties for _Z14kernel_forwardIdEviiPKT_S2_S2_S2_PS0_S3_
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 164 registers, used 1 barriers, 1536 bytes smem, 408 bytes cmem[0]
[3/3] c++ main.o cuda.cuda.o -shared -L/home/xrete/.local/lib/python3.8/site-packages/torch/lib -lc10 -lc10_cuda -ltorch_cpu -ltorch_cuda -
Loading extension module gated_att_1...
```

图 9. kernel 编译结果

```
::: check diff in fp64
============================
abs(a-b)......(mean, max)
----------------------------
y     (1.9948834341510188e-15, 2.842170943040401e-14)
y/d q (1.5153377187669795e-13, 1.8189894035458565e-12)
y/d k (1.6498559472097907e-13, 3.637978807091713e-12)
y/d g (1.051720288549468e-12, 1.9940671336371452e-10)
y/d v (6.696581381682569e-14, 1.3642420526593924e-12)
y/d s (2.0695755367335558e-15, 5.684341886080802e-14)
----------------------------
s     (2.5442249227338614e-17, 8.881784197001252e-16)
s/d q (0.0, 0.0)
s/d k (2.386565608586129e-15, 1.1368683772161603e-13)
s/d g (4.8232719279151 34e-15, 6.998845947236987e-13)
s/d v (2.716105206835636e-15, 1.1368683772161603e-13)
s/d s (1.336697134370197e-19, 5.551115123125783e-17)
----------------------------
============================
::: check grad in fp64
============================
Comparing analytical grads and numerical grads:
check -> True
============================
```

图 10. 梯度验证结构

4.3 GLA Transformer 实现

基于 GLA kernel，就可以构建时间混合（TimeMix）计算层，进而组建整个 GLA Transformer。图 11， 12分别是 GLA 时间混合层的定义和前向计算代码。

```python
class GatedLinearAttention(AutoModule):
    def __init__(self, dim_feature, num_head, normalizer=None, emb_position=None, **kwargs):
        super().__init__(locals())

        assert dim_feature % num_head == 0
        self.siz_head = dim_feature // num_head

        self.kernel = GLACuda(dim_feature, num_head)

        self.linear_q = nn.Linear(dim_feature, dim_feature, bias=False)
        self.linear_k = nn.Linear(dim_feature, dim_feature, bias=False)
        self.linear_v = nn.Linear(dim_feature, dim_feature, bias=False)
        self.linear_g = nn.Sequential(nn.Linear(dim_feature, 16, bias=False), nn.Linear(16, dim_feature, bias=True))
        self.linear_r = nn.Linear(dim_feature, dim_feature, bias=True)
        self.linear_o = nn.Linear(dim_feature, dim_feature, bias=False)
        self.normalizer = normalizer and self.instantiate(normalizer) or GroupNorm(num_head, dim_feature)
        self.emb_position = emb_position and self.instantiate(emb_position, dim_embed=self.siz_head) or AutoIdentity()

        self.state = self.declare_auto_state('hidden_state')
        # hidden_state (siz_batch, num_head, siz_head, siz_head)
```

图 11. 时间混合层定义

```python
    def forward(self, x, q, k, v, g):

        siz_batch, num_token, _ = x.shape

        q, k, v, g = self.linear_q(q), self.linear_k(k), self.linear_v(v), self.linear_g(g)
        # q k v g (siz_batch, num_token, dim_feature)
        g, k = torch.sigmoid(g), self.emb_position(k)

        s = self.state.if_not(
            init=lambda: torch.zeros(siz_batch, self.num_head, self.siz_head, self.siz_head).to(q)
        ).detach()
        mix, s = self.kernel(q, k, g, v, s)
        self.state.update(s)
        # mix (siz_batch, num_token, dim_feature) # s (siz_batch, num_head, siz_head, siz_head)

        x = self.linear_o(self.normalizer(mix) * x * torch.sigmoid(x))
        # x (siz_batch, num_token, dim_feature)

        return x
```

图 12. 时间混合层计算

图 13便是一个完整的 GLA 层，其构建的结构如图 14所示

```python
def block_default_gla__(dim_feature, num_head):
    """Default GLA block config"""
    return p(TimeChannelMixingBlock,
             dim_feature=dim_feature,
             num_head=num_head,
             normalizer=p(nn.LayerNorm, normalized_shape=dim_feature),
             time_mixing=p(AutoSequential, do='x->x',
                           module_list=[
                               p(AutoSum, do='x -> q, k, v, g'),
                               p(GatedLinearAttention)
                           ]),
             channel_mixing=p(AutoSequential, do='x->x',
                              module_list=[
                                  p(SwiGLUChannelMixing)
                              ]),
             )()
```

图 13. GLA Block 代码

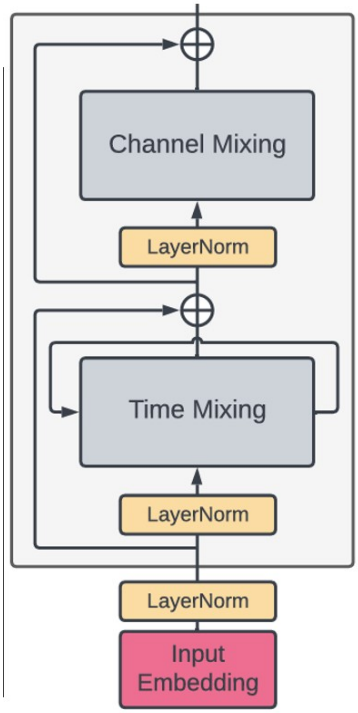

图 14. GLA Block 示意图

4.4 创新点

本次复现还对 GLA 进行了微创新，为 GLA 添加了 TokenShift 机制。TokenShift 类似一种一维的 CNN，可以增加模型的“本地性”，即短期信息处理能力，同时也是一种对于文字信息来说非常好的先验，其结构如 15图所示。同时还对 GLA 的 FNN 部分进行了更改，其最终代码如图 16所示。

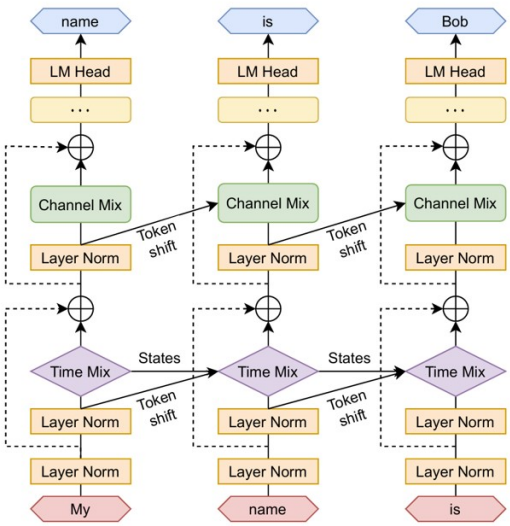

图 15. TokenShift 机制

```
def block_gla_with_token_shift__(dim_feature, num_head):
    """A better GLA block config, but slower"""
    return p(TimeChannelMixingBlock,
            dim_feature=dim_feature,
            num_head=num_head,
            normalizer=p(GroupNorm, num_head=1),
            time_mixing=p(AutoSequential, do='x->x',
                          module_list=[
                              p(TokenShift, do='x -> shift'),
                              p(AutoSum, do='shift -> q, k, v'),
                              p(AutoSum, do='x -> g'),
                              p(GatedLinearAttention,
                                normalizer=p(nn.LayerNorm, normalized_shape=dim_feature),
                                emb_position=None,
                                )
                          ]),
            channel_mixing=p(AutoSequential, do='x->x',
                             module_list=[
                                 p(TokenShift, do='x -> r, k'),
                                 p(RWKV5ChannelMixing)
                             ]),
            )()
```

图 16. 改进版 GLA

5 实验结果

本次复现在 Associated Recall 测试和小说文本上测试了所复现的架构。其中，Associated Recall 测试是一种测试架构长期记忆能力的测试。如图 17所示，其通过架构在前文中保存和回忆信息的多少来量化架构的长期记忆能力。

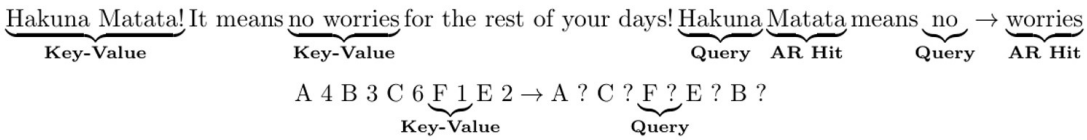

图 17. Associated Recall

图 18是原文中的测试结果，图 19是本次测试的结果。在本次测试中，对架构进行了一些调整才能得出正常结果：1，去掉了架构中的 swish gate，swish gate 是一种在文本这种部分信息更加重要的环境中表现得比较好的结构，但是测试中所有信息都很重要。2，手动调整了 g 的初始化值，使其更加倾向于记忆而非遗忘。3，加上了 token shift。

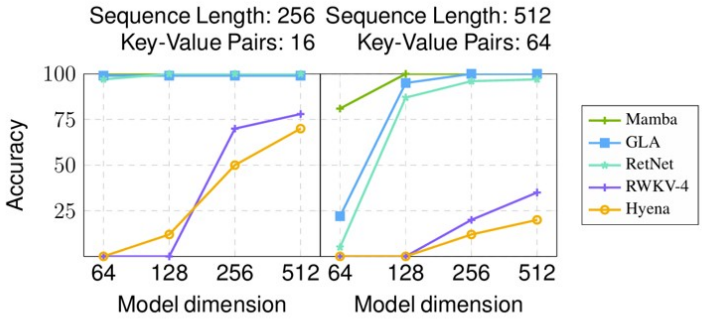

Figure 4: Accuracy (%) on the synthetic MQAR task.

图 18. 原文 AR 测试结果

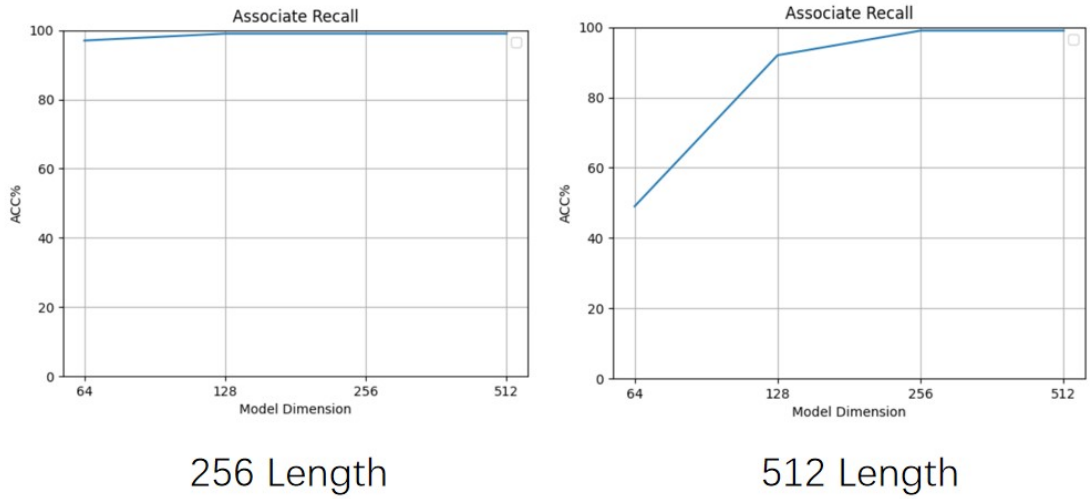

图 19. 复现 AR 测试结果

本次复现还在小说文本上进行了训练，并对比了原版 GLA 和本次复现的改进版 GLA 的 PPL。如图 20所示，可以发现改进的 GLA 的有较小的优势。

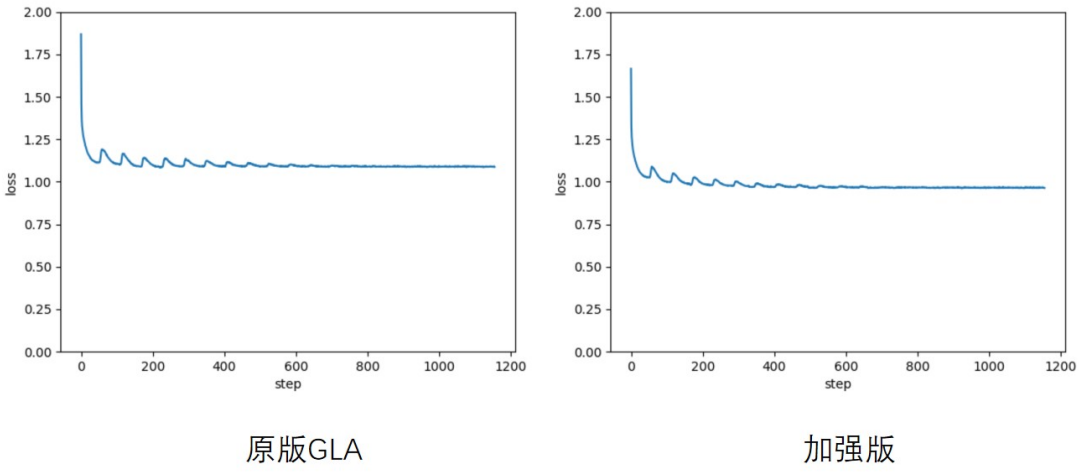

图 20. 小说文本的 PPL

6 总结与展望

本次复现以 parallel scan 的方式实现了 GLA 的 CUDA 内核，在保证计算效率的同时，显著提升了内核代码的可读性。通过引入 chunk-wise 的形式，将 GLA Transformer 的并行和串行计算统一为一致的块处理模式。此外，还设计了高效的隐状态管理机制，简化了隐藏状态的重置、转移等操作。这些优化使得研究者能够更加便捷直观地基于 GLA 开展新的架构探索。

在 ARtest 和小说数据集上对实现的 GLA 模型进行了训练和测试。通过对原始 GLA 模型进行调整，在 ARtest 数据集上取得了与原论文略优的结果，而在小说数据集上则获得了更低的 PPL，充分验证了复现的代码的有效性。

在本次复现的基础上，可以进一步尝试各种不同架构的组合，探索基于 GLA 的创新架构。

参考文献

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

[2] A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

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