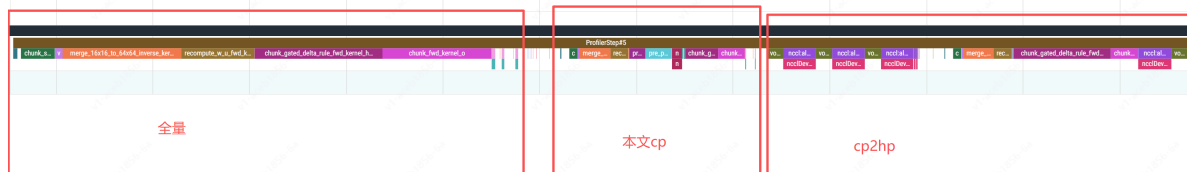


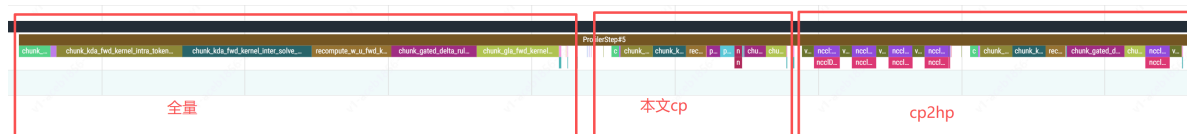
# GDN和KDA的高效CP方法

全新的CP方法。基于前人的伟大博客[DeltaNet如何做序列并行](#)加上自己的理解进行实现，大大加速GDN (KDA) 在长文本下的训练效率，并且CP无限开，轻松训练到1M长度。目前已给flash-linear-attention提了pr，欢迎大家尝试或者进行review。

GDN

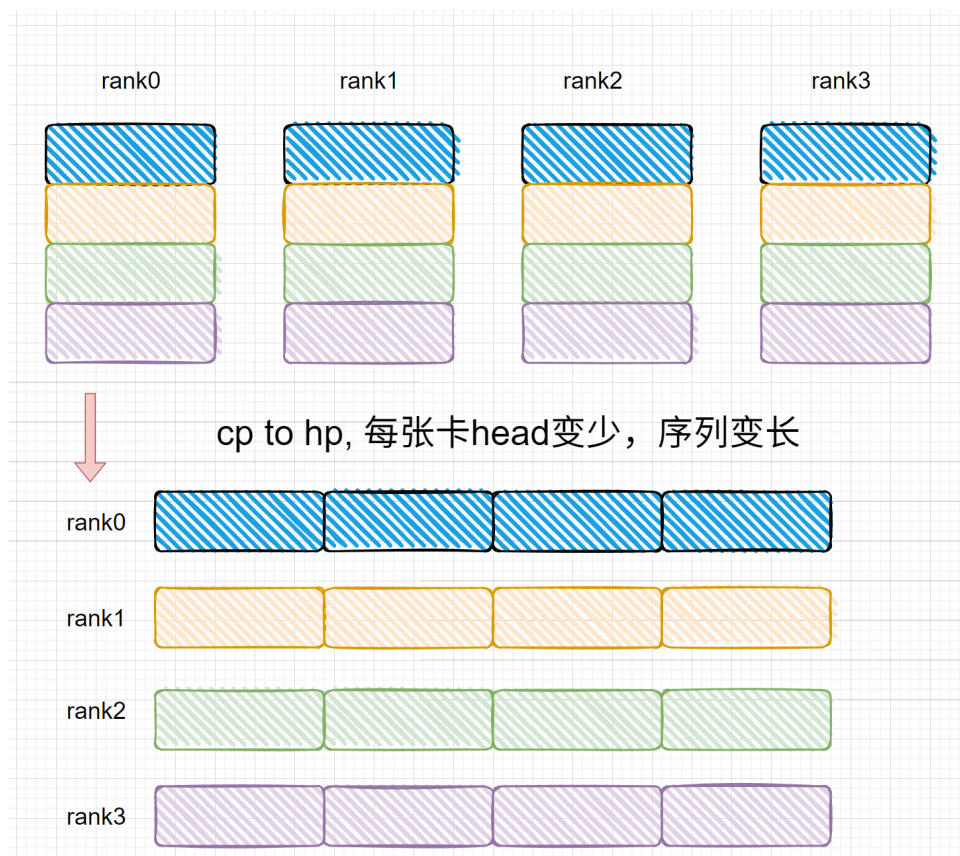


KDA



## 为什么需要更高效的CP

- GDN或KDA等linear attn是为解决长序列而存在的，面对长序列训练有两个基本要求：
  - 要能在足够长的序列上进行训练，即有可行的CP方案
  - 在长序列上训练性能不会严重裂化
- 现存CP方案
  - 目前GDN的CP方案开源出来的，我看到的很少，比如有[Megatron-LM dev](#)。丰富程度远不如MHA/GQA等标准attention
  - Megatron-LM的实现、我自己之前的实现以及和其他人的讨论得知的方案，大家基本都是使用的cp2hp (**deepspeed-ulysses**) 的并行方式，因为算法本身就是在head维度进行并行，对head不感知，因此算子无需任何修改即可
  - ring attention (p2p) 或者 all-gather attention (llama3 cp)对GDN都不太兼容，因此无法使用



- cp2hp的不足

- **最大cp数量受限head个数。** `max_cp_size = num_heads`。Kimi-linear报告中显示支持1M的context, 该model只有32个head, 如果是原生训练1M长度的话, 把cp拉满, 每张卡需要放64k长度的数据, 这显然非常大了。因此我猜测可能没训到1M或者采用了其他的CP方式 (不是单纯的cp2hp的现有方案)

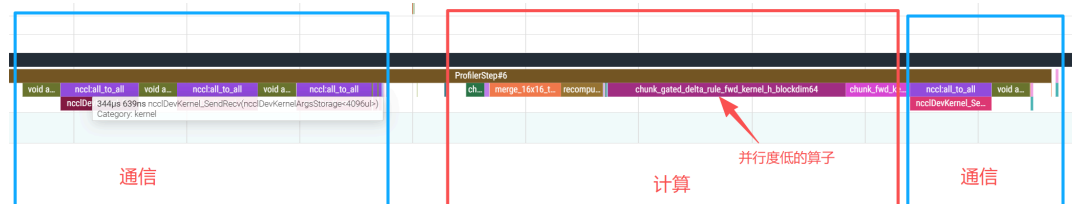
- **算子并行度低。** GDN(KDA)的核心算子之一

`chunk_gated_delta_rule_fwd_kernel_h_blockdim64` 目前只能在batch, head和chunk V之间进行并行

(N, H, V//BV), 如果batch=1, cp2hp可能导致H也等于1, 那么该算子就是灾难级别的, 可能只会用到2-4个SM, GPU利用率大大降低。

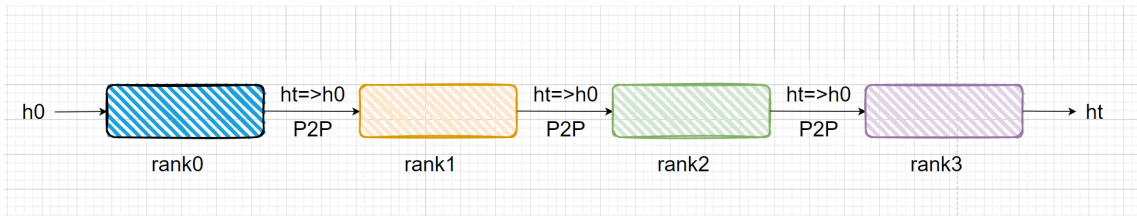
```
494     def grid(meta): return (triton.cdiv(V, meta['BV']), N*H)
495     chunk_gated_delta_rule_fwd_kernel_h_blockdim64[grid](<
```

- **通信时间长。** linear attn的耗时短, cp2hp的通信占比会被放大。



## Scan并行算法

- 最简单的方式: GDN(KDA)只有 `chunk_gated_delta_rule_fwd_kernel_h_blockdim64` 这一个算子依赖序列间的信息传递, 其他算子都和序列无关。因此一个办法就是逐rank去算, 比如rank0先算, 得到final\_state传给rank1作为initital\_state, rank1再算final\_state传给rank2.....该方法也很直观, 但是gpu之间无法并行计算, rank i需要等待rank i-1算完才能算, 会有严重的bubble, 类似pp并行的warm up阶段。这显然也不是一个很好的办法。



- 如何解决上述问题？RNN类模型在过去为提高计算效率，将状态( $S_t$ )之间的变化从非线性变化转成线性变化，通过Scan算法可以大幅度提高计算并行度。GDN也可以做类似的事情，只是复杂一些。

---

**Algorithm 1** Parallel linear recurrence on  $p$  processors
 

---

```

1: Let  $y = [(\Lambda_1, x_1), (\Lambda_2, x_2), \dots, (\Lambda_T, x_T)]$ 
2: Let binary operator  $\bullet$  act as  $(\Lambda, x) \bullet h = \Lambda h + x$ 
3: Let  $S_0 = 1, S_i < E_i, E_i + 1 = S_{i+1}, E_{p-1} = T$  for  $i$  in  $0, p - 1$ 
4:
5: parfor  $i \leftarrow 0, p - 1$  do
6:    $P_i = \text{REDUCE}(\odot, \Lambda_{S_i:E_i}, I)$ 
7:    $R_i = \text{REDUCE}(\bullet, y_{S_i:E_i}, 0)$ 
8: end parfor
9:
10: Let  $z = [(P_0, R_0), (P_1, R_1), \dots, (P_p, R_p)]$ .
11:  $C = \text{SCAN}(\bullet, z, h_0)$   $\triangleright$  compute  $C_i = P_i C_{i-1} + R_i$  with  $C_{-1} = h_0$ 
12:
13: parfor  $i \leftarrow 0, p - 1$  do
14:    $h_{S_i:E_i} = \text{SCAN}(\bullet, y_{S_i:E_i}, C_{i-1})$ 
15: end parfor
16: return  $h$ 
  
```

---

## GDN下的Scan算法

### 前言

本人是阅读博客[DeltaNet如何做序列并行](#)之后才有的想法（之前根本想不到QAQ，建议大家也阅读下原文），并在该基础上进行了扩展，推导了GDN和KDA下的并行，以及加上自己的一些理解，对GDN和KDA实现了真正的序列并行。

PS：我数学一般，幸好我只能看懂GDN最后的两个公式，前面的推导看不懂一点，而前面的公式不涉及序列信息的传递，因此可以直接跳过！

$$\begin{aligned}
 \mathbf{S}_{[t+1]} &= \overrightarrow{\mathbf{S}}_{[t]} + \left( \widetilde{\mathbf{U}}_{[t]} - \overleftarrow{\mathbf{W}}_{[t]} \mathbf{S}_{[t]}^\top \right)^\top \overrightarrow{\mathbf{K}}_{[t]} && \in \mathbb{R}^{d_v \times d_k} \\
 \mathbf{O}_{[t]} &= \overleftarrow{\mathbf{Q}}_{[t]} \mathbf{S}_{[t]}^\top + (\mathbf{Q}_{[t]} \mathbf{K}_{[t]}^\top \odot \mathbf{M}) \left( \widetilde{\mathbf{U}}_{[t]} - \overleftarrow{\mathbf{W}}_{[t]} \mathbf{S}_{[t]}^\top \right) && \in \mathbb{R}^{C \times d_v}
 \end{aligned}$$

where  $\overleftarrow{\mathbf{q}}_{[t]}^r = \gamma_{[t]}^r \mathbf{q}_{[t]}^r$ ,  $\overleftarrow{\mathbf{w}}_{[t]}^r = \gamma_{[t]}^r \mathbf{w}_{[t]}^r$ ,  $\overrightarrow{\mathbf{k}}_{[t]}^r = \frac{\gamma_{[t]}^C}{\gamma_{[t]}^r} \mathbf{k}_{[t]}^r$ , and  $\overrightarrow{\mathbf{S}}_{[t]} = \gamma_{[t]}^C \mathbf{S}_{[t]}$  like we defined in Eq. 2.

### 从代码进行切入

假设CP=4，一个完整序列被平均切分为4份。在唯一和序列相关的函数

`chunk_gated_delta_rule_fwd_h`中，每个rank的 `initial_state` 默认都为0，这对于rank0是对的，对rank1, 2, 3都是错误的，如果有一个额外的函数，能够算出成真实的 `initial_state`，那么下面的函数输出的结果就是正确的了。因此我的设计出发点是：写一个额外的函数插入到该函数之前。

```

flash-linear-attention > fla > ops > gated_delta_rule > chunk.py > chunk_gated_delta_rule_fwd
24 def chunk_gated_delta_rule_fwd(
68
69     h, v_new, final_state = chunk_gated_delta_rule_fwd_h(
70         k=k,
71         w=w,
72         u=u,
73         g=g,
74         initial_state=initial_state,
75         output_final_state=output_final_state,
76         cu_seqlens=cu_seqlens,
77     )
78

```

反向类似，rank3的dht为0，rank0，1，2的dht需要真实的dht，需要一个额外函数算出真实的dht。

```

flash-linear-attention > fla > ops > gated_delta_rule > chunk.py > chunk_gated_delta_rule_bwd
93 def chunk_gated_delta_rule_bwd(
150     dh, dh0, dv = chunk_gated_delta_rule_bwd_dhu(
151         q=q,
152         k=k,
153         w=w,
154         g=g,
155         h0=initial_state,
156         dht=dht,
157         do=do,
158         dv=dv,
159         scale=scale,
160         cu_seqlens=cu_seqlens,
161     )

```

## 前向公式推导

我为了方便写代码，推导过程中的公式的shape和代码中都是一致的，对S\_t都进行了转置进行表式。

$$\begin{aligned}
 S_{[t+1]}^T &= \overrightarrow{S}_{[t]}^T + \overrightarrow{K}_{[t]}^T (U_{[t]}^T - \overleftarrow{W}_{[t]} S_t^T) \\
 &= \overrightarrow{S}_{[t]}^T + \overrightarrow{K}_{[t]}^T (U_{[t]}^T - \overleftarrow{W}_{[t]} S_t^T) \\
 &= \gamma_{[t]}^C S_{[t]}^T + K_{[t]}^T \text{diag}(\gamma_{[t]}) (U_{[t]}^T - \overleftarrow{W}_{[t]} S_t^T) \\
 &= (I\gamma_{[t]}^C - K_{[t]}^T \text{diag}(\gamma_{[t]}) \overleftarrow{W}_{[t]}) S_t^T + K_{[t]}^T \text{diag}(\gamma_{[t]}) U_{[t]}^T
 \end{aligned} \tag{1}$$

$$O_{[t]} = \overleftarrow{Q}_{[t]} S_{[t]}^T + (Q_{[t]} K_t^T \odot M) (U_{[t]}^T - \overleftarrow{W}_{[t]} S_t^T) \tag{2}$$

公式1和2是论文中的两个公式，其中的decay的Q和W我没有展开，是因为这两个与核心算子都无关。在公式1的第二行，K的decay放到了括号那项，这是原始代码中的做法。

$$S_{[t+1]}^{*T} = (I\gamma_{[t]}^C - K_{[t]}^T \text{diag}(\gamma_{[t]}) \overleftarrow{W}_{[t]}) S_{[t]}^{*T} + K_{[t]}^T \text{diag}(\gamma_{[t]}) U_{[t]}^T \tag{3}$$

假设每张rank都初始化S\_0为0，得到S\*\_t，即公式3，这时rank1，2，3的结果都是不正确的。

$$S_{[t+1]}^T - S_{[t+1]}^{*T} = (I\gamma_{[t]}^C - K_{[t]}^T \text{diag}(\gamma_{[t]}) \overleftarrow{W}_{[t]}) (S_{[t]}^T - S_{[t]}^{*T}) \tag{4}$$

$$S_{[i+k]}^T - S_{[i+k]}^{*T} = \left( \prod_{j=0}^{k-1} (I\gamma_{[i+j]}^C - K_{[i+j]}^T \text{diag}(\gamma_{[i+j]}) \overleftarrow{W}_{[i+j]}) \right) (S_{[t]}^T - S_{[t]}^{*T}) \tag{5}$$

然后用公式1减公式3，得到公式4，对公式4的最后一项继续进行展开，就能得到chunk之间的递推公式5。

$$S_{[i+k]}^T = \left( \prod_{j=0}^{k-1} (I\gamma_{[i+j]}^C - K_{[i+j]}^T \text{diag}(\gamma_{[i+j]}) W_{[i+j]}) \right) (S_{[t]}^T - S_{[t]}^{*T}) + S_{[i+k]}^{*T} \quad (6)$$

$$\text{另 } M_{[i]} = I\gamma_{[i]}^C - K_{[i]}^T \text{diag}(\gamma_{[i]}) W_{[i]}$$

$$\text{则 } M_{[i+1]} = (I\gamma_{[i+1]}^C - K_{[i+1]}^T \text{diag}(\gamma_{[i+1]}) W_{[i+1]}) M_{[i]} \quad (7)$$

对公式5进行移项后得到公式6。

## 前向算法设计

根据上面的公式，即可设计如下算法。

假设有16个chunk，每张卡4个chunk。

rank0: chunk 1, 2, 3, 4, 初始化S\*\_1=0, M\_0=I

rank1: chunk 5, 6, 7, 8, 初始化S\*\_5=0, M\_4=I

rank2: chunk 9, 10, 11, 12, 初始化S\*\_9=0, M\_8=I

rank3: chunk 13, 14, 15, 16, 初始化S\*\_13=0, M\_12=I

使用公式1（第2个等号）计算S\*，并行算。只需存储一个S，大小为H \* K \* V

rank0: S\*\_5

rank1: S\*\_9

rank2: S\*\_13

rank3: S\*\_17

使用公式7计算M，并行算。只需存储一个M，大小为H \* K \* K

rank0: M\_4

rank1: M\_8

rank2: M\_12

rank3: M\_16

all\_gather。通信量大小为CP \* H \* K \* (K+V)

S\* = [S\*\_5, S\*\_9, S\*\_13, S\*\_17]

M = [M\_4, M\_8, M\_12, M\_16]

使用公式6进行merge，并行算。唯一缺点，该算子，每张卡计算不均衡，但是占比很小

rank0: S\_1 = 0

rank1: S\_5 = S\*\_5

rank2: S\_5 = S\*\_5, S\_9 = M\_8 @ S\_5 + S\*\_9

rank3: S\_5 = S\*\_5, S\_9 = M\_8 @ S\_5 + S\*\_9, S\_13 = M\_12 @ S\_9 + S\*\_13

现在，每个rank都得到有真实的initial\_state，继续走GDN的剩余步骤

## 反向公式推导

反向和前向类似，也可以写成前向的形式，只不过方向是反过来来的。算法设计我就不是写了。

$$\text{另公式1中的: } (U_{[t]}^T - \overleftarrow{W_{[t]}} S_{[t]}^T) = V_1, \text{ 则 } dV_1 = \text{diag}(\gamma_{[t]}) K_{[t]} dS_{[t+1]}^T$$

$$\text{另公式2中的: } (U_{[t]}^T - \overleftarrow{W_{[t]}} S_t^T) = V_2, dV_2 \text{ 可以通过其他算子直接求到}$$

对公式1和2中的S\_t进行求导。

$$\begin{cases} dS_{[t]}^{T(1)} = \overleftarrow{Q}_{[t]}^T dO_{[t]} \\ dS_{[t]}^{T(2)} = -\overleftarrow{W}_{[t]}^T dV_1 = \overleftarrow{W}_{[t]}^T \text{diag}(\gamma_{[t]}) K_{[t]} dS_{[t+1]}^T \\ dS_{[t]}^{T(3)} = -\overleftarrow{W}_{[t]}^T dV_2 \\ dS_{[t]}^{T(4)} = \gamma_{[t]}^C dS_{[t+1]}^T \end{cases}$$

$$dS_{[t]}^T = (\gamma_{[t]}^C I - \overleftarrow{W}_{[t]}^T \text{diag}(\gamma_{[t]}) K_{[t]}) dS_{[t+1]}^T + \overleftarrow{Q}_{[t]}^T dO_{[t]} - \overleftarrow{W}_{[t]}^T dV_2$$

$$dS_{[t]}^{*T} = (\gamma_{[t]}^C I - \overleftarrow{W}_{[t]}^T \text{diag}(\gamma_{[t]}) K_{[t]}) dS_{[t+1]}^{*T} + \overleftarrow{Q}_{[t]}^T dO_{[t]} - \overleftarrow{W}_{[t]}^T dV_2$$

$$dS_{[t]}^T - dS_{[t]}^{*T} = (\gamma_{[t]}^C I - \overleftarrow{W}_{[t]}^T \text{diag}(\gamma_{[t]}) K_{[t]}) (dS_{[t+1]}^T - dS_{[t+1]}^{*T})$$

$$dS_{[t]}^T - dS_{[t]}^{*T} = \left( \prod_{j=0}^{k-1} (\gamma_{[t+j]}^C I - \overleftarrow{W}_{[t+j]}^T \text{diag}(\gamma_{[t+j]}) K_{[t+j]}) \right) (dS_{[t+k]}^T - dS_{[t+k]}^{*T})$$

$$dS_{[t]}^T = \left( \prod_{j=0}^{k-1} (\gamma_{[t+j]}^C I - \overleftarrow{W}_{[t+j]}^T \text{diag}(\gamma_{[t+j]}) K_{[t+j]}) \right) (dS_{[t+k]}^T - dS_{[t+k]}^{*T}) + dS_{[t]}^{*T}$$

## 代码设计

### context初始化（最核心）

实际场景中是varlen的，不同rank是可以跳过一些计算的。下面的初始化是最核心的，后面的算子都依赖这些信息，请看下面代码中的注释进行理解。

```
S = cu_seqlens_list[-1]
part_len = (S // world_size)
start, end = part_len * rank, part_len * (rank + 1) # 每个rank的切分范围
def contain(left, right): # 判断该rank的范围与该样本的范围是否重合。
    return (min(right, end) - max(left, start)) > 0
cu_seqlens = []
is_last_rank_list, pre_num_ranks_list, is_first_rank_list, post_num_ranks_list =
[], [], [], []
for idx in range(len(cu_seqlens_list) - 1):
    left, right = cu_seqlens_list[idx], cu_seqlens_list[idx + 1]
    if left >= end: # 该样本以及之后的都不会在这个rank范围内
        break
    if contain(left, right):
        cu_seqlens.append(max(left - start, 0))
        pre_num_conv_tokens_list.append(max(0, start - left))
        # cross rank
        if left < start or right > end:
            first_rank = left // part_len
            last_rank = (right - 1) // part_len
            is_last_rank_list.append(rank == last_rank) # forward
            pre_num_ranks_list.append(rank - first_rank) # forward
            is_first_rank_list.append(rank == first_rank) # backward
            post_num_ranks_list.append(last_rank - rank) # backward
```

```

else:
    is_last_rank_list.append(True)
    pre_num_ranks_list.append(0)
    is_first_rank_list.append(True)
    post_num_ranks_list.append(0)
if cu_seqlens[-1] != part_len:
    cu_seqlens.append(part_len)
cu_seqlens = torch.tensor(cu_seqlens, dtype=torch.int32,
device=torch.cuda.current_device())
# 取最后一个, 如果是False, 那么该rank需要计算state传给下一个
is_last_rank = is_last_rank_list[-1]
# 取第一个, 如果>0, 需要接收前面rank的state
pre_num_ranks = pre_num_ranks_list[0]
# 取第一个, 如果>0, 需要接收前面rank的conv_tokens
pre_num_conv_tokens = pre_num_conv_tokens_list[0]
# 取第一个, 如果是False, 那么该rank需要计算dstate传给上一个
is_first_rank = is_first_rank_list[0]
# 取最后一个, 如果>0, 需要接收后面rank的dstate
post_num_ranks = post_num_ranks_list[-1]

```

## 算子

首先 `pre_process_fwd_kernel_stage1` 直接copy fla的算子

`chunk_gated_delta_rule_fwd_kernel_h_blockdim64`, 对其进行删删减减即可, 非常easy。注意grid是没有batch这个维度的, 如果是varlen, 该rank的batch数量比较大, 那么该算子的耗时非常短。

然后 `pre_process_fwd_bwd_kernel_stage2` 是对公式6进行实现。该grid是没有batch这个维度的。这个算子有个缺点, 公式中6中有个 $K\_T@W$ , 并且还需要连乘, 因为K和W的head\_dim维度没法拆开, 因此head\_dim目前最大只可以是128, 如果是192和256算子都跑不通。为了提高并行度, 可以在每个sm初始化 $M\_0$ 的时候, 只初始化 $[K, K//BK2]$ 的大小, 这样 $K\_T@W$ 的计算都是重复的, 但是 $M_{i+1} = (*) @ M_i$ 的计算量可以变小。

接着进行 `all_gather` 通信, 通信量和CP的大小成正比

最后 `merge_fwd_bwd_kernel` 会更新该rank的第一个样本的initial\_state

反向和前向一样, 只是 `pre_process_fwd_kernel_stage1` 换成了 `pre_process_bwd_kernel_stage1`, 是对 `chunk_gated_delta_rule_bwd_kernel_dhu_blockdim64` 进行删删减减

```

def chunk_gated_delta_rule_fwd_h_pre_process(*):
    hm = k.new_zeros(H, K, (V + K), dtype=DTYPE)
    initial_state = k.new_zeros(N, H, K, V, dtype=DTYPE)

    if not context.is_last_rank:
        def grid(meta): return (triton.cdiv(V, meta['BV']), H)
        pre_process_fwd_kernel_stage1[grid](hm, cu_seqlens=cu_seqlens[-2:], *)

        def grid(meta): return (triton.cdiv(K, meta['BK2']), H)
        pre_process_fwd_bwd_kernel_stage2[grid](hm, cu_seqlens=cu_seqlens[-2:],
*)

    ag_hm, _ = all_gather(hm, group=context.group)

    if context.pre_num_ranks > 0:
        def grid(meta): return (triton.cdiv(V, meta['BV']), H)

```

```
merge_fwd_bwd_kernel[grid](initial_state[0], rank, pre_num_ranks, *)
```

```
return initial_state
```

## 接入带GDN或者KDA中

首先初始化context

```
flash-linear-attention > fla > layers > gated_deltanet.py > ...
322 class GatedDeltaNetWithCP(GatedDeltaNet):
362     def forward(
387
388         set_gdn_cp_context(cu_seqlens, self.group, kernel_size=self.conv_size if self.use_short_conv else None)
389         context = get_gdn_cp_context()
390
391         kwargs['cu_seqlens'] = context.cu_seqlens
392         out = super().forward(
393             hidden_states,
394             attention_mask,
395             past_key_values,
396             use_cache,
397             output_attentions,
398             **kwargs,
399         )
400         kwargs['cu_seqlens'] = cu_seqlens
401
402     return out
403
```

forward中进行插入

```
flash-linear-attention > fla > ops > gated_delta_rule > chunk.py > chunk_gated_delta_rule_fwd
24 def chunk_gated_delta_rule_fwd(
58     context = get_gdn_cp_context()
59     initial_state = chunk_gated_delta_rule_fwd_h_pre_process(
60         k=k,
61         w=w,
62         u=u,
63         g=g,
64         cu_seqlens=cu_seqlens,
65         initial_state=initial_state,
66         context=context,
67     )
68
69     h, v_new, final_state = chunk_gated_delta_rule_fwd_h(
70         k=k,
71         w=w,
72         u=u,
73         g=g,
74         initial_state=initial_state,
75         output_final_state=output_final_state,
76         cu_seqlens=cu_seqlens,
77     )

```

backward中进行插入



```
flash-linear-attention > fla > ops > gated_delta_rule > chunk.py > chunk_gated_delta_rule_bwd
93 def chunk_gated_delta_rule_bwd(
135
136     dht, initial_state = chunk_gated_delta_rule_bwd_dhu_pre_process(
137         q=q,
138         k=k,
139         w=w,
140         do=do,
141         dv=dv,
142         g=g,
143         scale=scale,
144         cu_seqlens=cu_seqlens,
145         dht=dht,
146         initial_state=initial_state,
147         context=context,
148     )
149
150 dh, dh0, dv = chunk_gated_delta_rule_bwd_dhu(
151     q=q,
152     k=k,
153     w=w,
154     g=g,
155     h0=initial_state,
156     dht=dht,
157     do=do,
158     dv=dv,
159     scale=scale,
160     cu_seqlens=cu_seqlens,
161 )
```

注意的是，如果用到的conv1d，还需要拿到上一个rank的尾块token，需要进行额外的通信。可恶的conv1d，什么时候去死！

```
flash-linear-attention > fla > modules > convolution.py > ShortConvolution
806 class ShortConvolution(nn.Conv1d):
901 def forward(
966     self.backend = 'triton'
967     x, cu_seqlens = pre_process_for_conv1d(x, cu_seqlens)
968     y, cache = causal_conv1d(
969         x=x,
970         weight=rearrange(self.weight, "d 1 w -> d w"),
971         bias=self.bias,
972         residual=residual,
973         initial_state=cache,
974         output_final_state=output_final_state,
975         activation=self.activation,
976         backend=self.backend,
977         cu_seqlens=cu_seqlens,
978         chunk_indices=chunk_indices,
979         **kwargs,
980     )
981     y = post_process_for_conv1d(y, T)
982     return y, cache
983
```

## 性能表现

32k, cp=4, num\_head=64, head\_dim=128. torch=2.8, cuda=12.8, triton=3.4, gpu=H800.

3种模式，全量，a2a模式(包含通信，但是每张卡head变少)，以及本文所讲的CP方式。只对比了核心chunk函数，不包含linear和conv1d等操作。

据一位朋友告知：256k下，训练类似kimi-linear的模型，端到端可加速20%，money大大的省！

cu\_seqlen: [0, 32768]

GDN forward

custom cp, non-cp time: 5.432 ms, cp time: 2.160 ms, rate: 62.86 %

all to all, non-cp time: 5.432 ms, cp time: 4.068 ms, rate: 33.38 %

GDN forward + backward

custom cp, non-cp time: 16.516 ms, cp time: 5.878 ms, rate: 70.24 %

all to all, non-cp time: 16.516 ms, cp time: 10.592 ms, rate: 38.98 %

KDA forward

custom cp, non-cp time: 7.753 ms, cp time: 2.622 ms, rate: 73.93 %

all to all, non-cp time: 7.753 ms, cp time: 5.047 ms, rate: 38.41 %

KDA forward + backward

custom cp, non-cp time: 37.592 ms, cp time: 10.801 ms, rate: 87.01 %

all to all, non-cp time: 37.592 ms, cp time: 15.877 ms, rate: 59.19 %

长文本训练种也包含一些短文本

cu\_seqlen: [0, 2960, 5212, 9513, 13567, 17443, 20634, 23521, 26281, 31785, 32768]

GDN forward

custom cp, non-cp time: 4.816 ms, cp time: 1.665 ms, rate: 72.31 %

all to all, non-cp time: 4.816 ms, cp time: 3.460 ms, rate: 34.80 %

GDN forward + backward

custom cp, non-cp time: 15.815 ms, cp time: 4.897 ms, rate: 80.74 %

all to all, non-cp time: 15.815 ms, cp time: 8.608 ms, rate: 45.93 %

KDA forward

custom cp, non-cp time: 8.066 ms, cp time: 2.351 ms, rate: 85.76 %

all to all, non-cp time: 8.066 ms, cp time: 4.750 ms, rate: 42.45 %

KDA forward + backward

custom cp, non-cp time: 37.730 ms, cp time: 10.489 ms, rate: 89.93 %

all to all, non-cp time: 37.730 ms, cp time: 15.162 ms, rate: 62.21 %

## profile分析

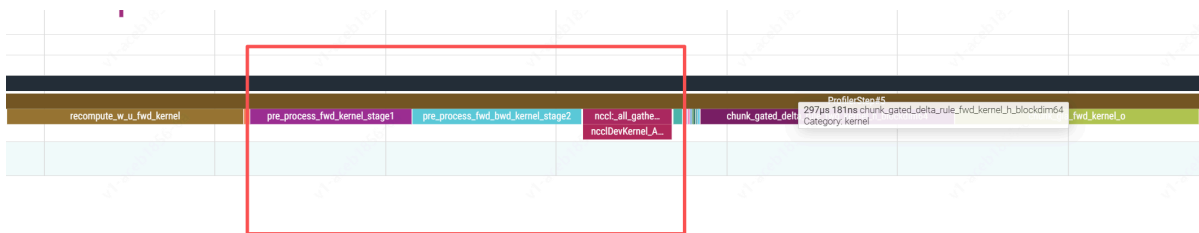
以KDA前向为例，拿的rank2的timeline信息

可以看到本文介绍的cp方式比cp2hp的方式要友好很多，主要原因是all2all的时间占比很大

下面是cu\_seqlen=[0, 32768]



本文cp多了3个算子+一次通信，然后 chunk\_gated\_delta\_rule\_fwd\_kernel\_h\_blockdim64 大概耗时 300us左右

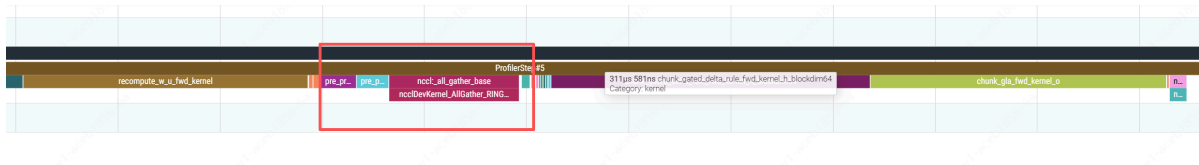


cp2hp中， chunk\_gated\_delta\_rule\_fwd\_kernel\_h\_blockdim64 大概耗时800us左右，因为head变少，并行度降低，效率大大折扣



下面是cu\_seqlens=[0, 2960, 5212, 9513, 13567, 17443, 20634, 23521, 26281, 31785, 32768]

本文cp，可以看到因为单个rank的样本数增加，我们只需要算一个样本即可，因此前2个算子时间大大减少，通信时间不变



cp2hp，样本数增加，并行度增加，因此时间也会减少一些。

