- Run Test
- 背景
  - Linear Attention
  - Chunkwise Parallel
- Gated Linear Attention Layer
  - 模型设计
  - 并行形式
- Chunked Fuse 的实现
  - fwd_decay_cumsum
  - prepare_qg_kg
  - fused_chunk_gla_fwd_kernel
  - fwd_inner_chunk
  - combine inner and intra

Run Test

```
python3 main.py
```

背景

Linear Attention

下面的公式是attention的全并行形式。

$$
\begin{gathered}
Q, K, V = XW_Q, XW_K, XW_V \\
O = \mathrm{softmax}(QK^T)V
\end{gathered}
$$

下面的公式是attention的单步计算形式（预测时）：

$$
o_t = \frac{\sum_{i=1}^{t} \exp(q_t k_i^T) v_i}{\sum_{i=1}^{t} \exp(q_t k_i^T)}
$$

linear attention的核心是用一个kernel function将query 和key序列中的token映射到正值，然后用内积（内积的数据并行可以用矩阵乘高效实现）作为两者之间的相似性度量，$< \phi(q_i), \phi(k_i)^T >$，用来替代softmax中的$\exp(q_i, k_i^T)$。于是输出$o_t$的计算可以被简化为：

$$
\begin{aligned}
o_t &= \frac{\sum_{i=1}^{t} \phi(q_t)\phi(k_i)^T v_i}{\sum_{i=1}^{t} \phi(q_t)\phi(k_i)^T} \\
&= \frac{\phi(q_t)\sum_{i=1}^{t} \phi(k_i)^T v_i}{\phi(q_t)\sum_{i=1}^{t} \phi(k_i)^T} \\
o_t &\triangleq \frac{\phi(q_t)S_t}{\phi(q_t)z_t} \\
S_t &= S_{t-1} + \phi(k_t)^T v_t \\
z_t &= z_{t-1} + \phi(k_t)
\end{aligned}
$$

(1)，(2)，(3)进一步简化，令$\phi = I$（indentity function），不归一化于是有：

$$
\begin{aligned}
o_t &= q_t S_t \\
S_t &= S_{t-1} + k_t^T v_t
\end{aligned}
$$

式(4)和(5)就是linear attention的recurrent形式。从式(4)和(5)可以看出：

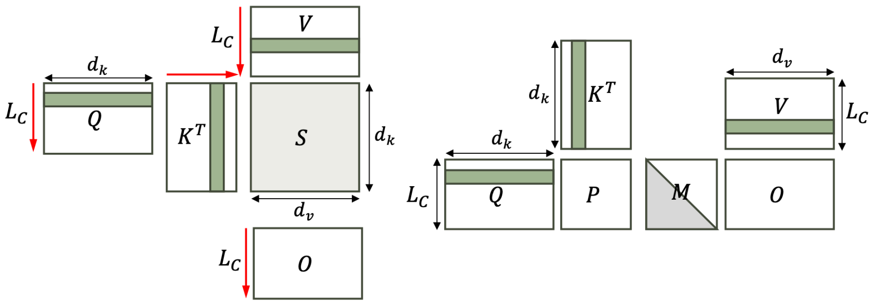
1. linear attention是一个hidden state是matrix的RNN；
2. 状态更新是以$+$为二元操作符，不断累加k_i和v_i的外积；$+$具有结合性，这样的一个累加过程，是可并行的，也就是parallel scan算法；
3. 如果以recurrent方式完成计算，linear attention只在序列长度上循环一次，不论是时间还是空间复杂度都不是序列长度的二次方。过程如下图所示如果在一个chunk之内以recurrent方式计算，这时chunk之内的计算是外积，向量乘以矩阵和加法。无法利用tensor core。

Chunkwise Parallel

Fig 4是chunk form linear attention的示意图，Q，K，V是含有12个token的序列，每一个长方形都是一个维度为d的词向量，Q，K，V被分为了3个chunks，每个chunk含有4个token。Fig 4中部的scores矩阵是单个query token和key token的相似度分析矩阵，每一个小方块是一个scalar。

不仅仅是linear attention，即使标准的MHA也可以以chunkwise方式进行计算（flash attention），linear attention和MHA相比优点有两个：

1. 预测的时候节约k-v cache；
2. 序列长度方向是parallel scan，当序列特别长的时候，在序列长度方向还有并行性；



(a) 以recurrent方式完成chunk内计算　　(b) 以矩阵方式完成chunk内计算

Fig 1. attention的chunk内计算

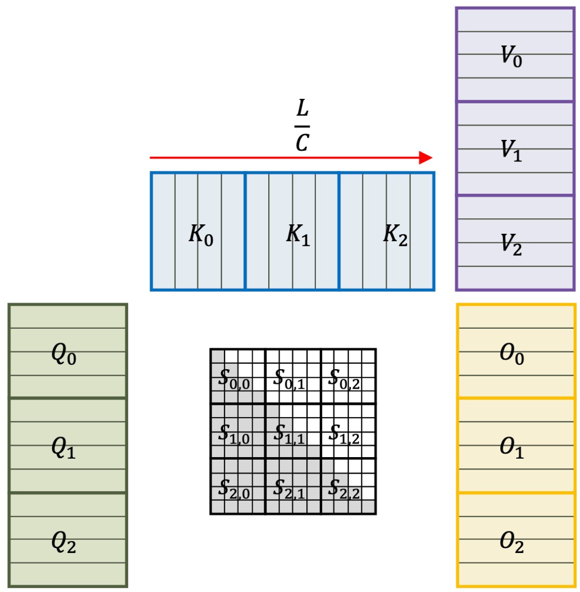


Fig 2. chunk-form linear attention.

chunk之间沿着序列长度方向对状态S_t进行累积，在序列长度方向的parallel scan（下面公式的红色部分），chunk之内（蓝色）可以以全并行或者recurrent 方式进行计算。

$$O_{i+1} = Q_{i+1}S_i + \left(\left(Q_{i+1}K_{i+1}^T\right) \odot M\right) V_{i+1}$$

到这里就会出现两种实现选择：

1. 以更大的并行性将所有的chunk独立进行计算，每个分块计算出状态S_i；这时候状态会占一个较大的空间，存储回DRAM；然后再用一个scan kernel以序列级别的并行性对状态进行累加，计算出最终的O；
2. kernel内部在序列长度方向循环，在SRAM上计算出状态，对状态进行累加，这时候不会产生DRAM和SRAM之间的I/O，但是并行性较小；

Gated Linear Attention Layer

模型设计

gated linear attention在linear attention的基础上加上input-dependent的对状态的衰减，也就是下图红色的G矩阵。

$$
\begin{aligned}
o_t &= q_t S_t & [1, d_v] &= [1, d_k] \otimes [d_k, d_v] \\
S_t &= {\color{red}G_t} \odot S_{t-1} + k_t^T v_t & [d_k, d_v] &= [d_k, d_v] \odot [d_k, d_v] + [d_k, 1] \otimes [1, d_v]
\end{aligned}
$$

注意：按照这个公式的设计，${\color{red}G_t}$是一个$d_k \times d_v$大小的矩阵。如果要求这个gating factor是依赖于输入的，如果通过full rank的方法得到G_t，那么需要$d \times d_k \times d_v$大小的映射矩阵，参数可能会非常多；为了使用一种更加parameter-efficient的方式得到G，GLA的模型设计中：

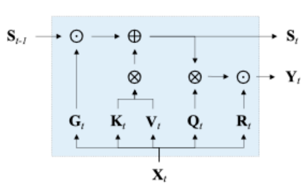
1. 首先通过一个低秩方法（先映射到一个***维度非常小的向量空间，再映射到最终的向量空间***）得到整个序列长度级别的gating factors矩阵：

$$G = \text{sigmoid}\left((x_{[1,d]} W^1_{[d,d_1]}) W^2_{[d_1,d_k]}\right)$$

2. 然后每个时间步${\color{red}G_t} = \left(\alpha_t^T \mathbf{1}\right)$；

GLA通过增加了：$d \times d_1 + d_1 \times d_k = (d + d_1) \times d_k$ 可学习参数得到了gating factor，其中$d_1 \lll d$，比full rank 方式要高效许多。

下面四行公式是gated linear attention layer在算法设计上得到的最终模型，公式（8）和（9）是linear attention部分，(10)和（11）是normalization和类似于输出门。



$$\mathbf{S}_t = \mathbf{G}_t \odot \mathbf{S}_{t-1} + \mathbf{K}_t^{\top} \mathbf{V}_t \in \mathbb{R}^{d_k \times d_v}, \tag{8}$$

$$\mathbf{O}_t = \mathbf{Q}_t^{\top} \mathbf{S}_t \in \mathbb{R}^{1 \times d_v}, \tag{9}$$

$$\mathbf{R}_t = \text{Swish}(\mathbf{X}_t \boldsymbol{W}_r + \boldsymbol{b}_r) \in \mathbb{R}^{1 \times d_v}, \tag{10}$$

$$\mathbf{Y}_t = (\mathbf{R}_t \odot \text{LN}(\mathbf{O}_t)) \boldsymbol{W}_O \in \mathbb{R}^{1 \times d}. \tag{11}$$

Fig 2. GLA layer以recurrent模式单步计算的线性代数公式

下图是GLA layer的实现转换为tensor operator之间的数据流依赖。**图中GLA圈出来的部分实现了论文中的公式（8）和（9），但分成了多个kernel。** 和RNN一样公式（8）带有时序依赖。

从DNN模型设计的角度看GLA的数据流动，上图中蓝色虚线这一枝信息的流动起到了和传统RNN中input gate类似的功能，也是GLA这个模型中"gated"所指的部分。绿色虚线这一枝信息的流动起到了和传统RNN中output gate类似的功能。

> GLA中的gating factor是一个和输入K大小相同的tensor，***沿着时间维度对2D state的每个hidden维度以一个$[0,1)$进行不同强度的衰减***。

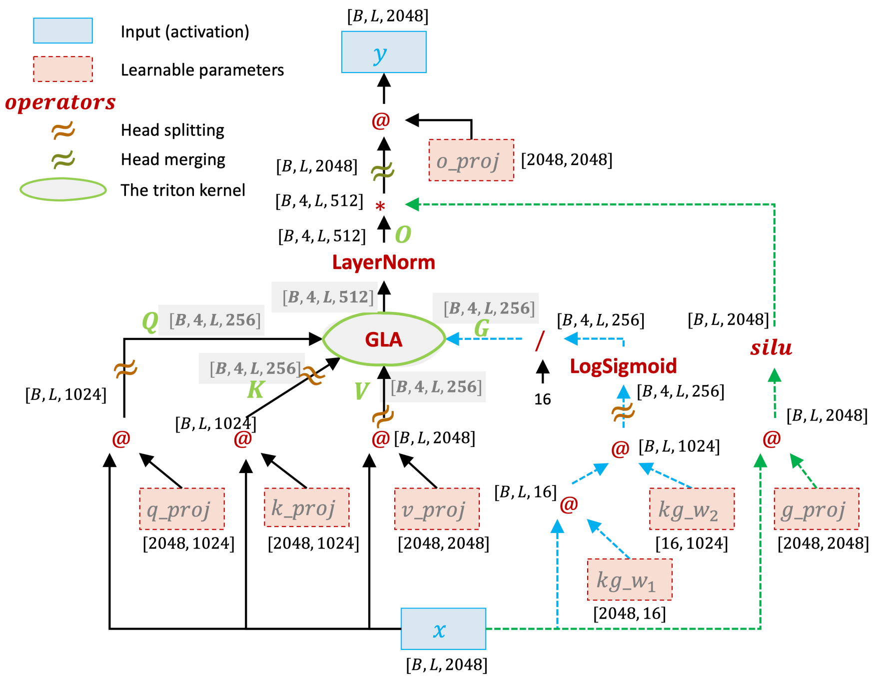


Fig 3. GLA layer的数据流依赖

一些注释：

1. $\mathrm{LogSigmoid}(x)=\log\frac{1}{1+\exp(-x)}$。将sigmoid函数的输出映射到log空间，将连乘转换为对数空间的连加。

   > 作者的代码实现看，PyTorch里LogSigmoid计算是以2位为底的对数，通过换底公式，将LogSigmoid激活的输出乘以 $\frac{1}{\ln 2}$ （$\log_2(x)=\frac{\ln x}{\ln 2}$ ），转换为以 e 为底的对数。然后通过2^x还原回sigmoid的输出，也就是gating的输出（gate是[0, 1)之间的一个浮点数）。

2. $\mathrm{silu}(x)=x*\mathrm{sigmoid}(x)$

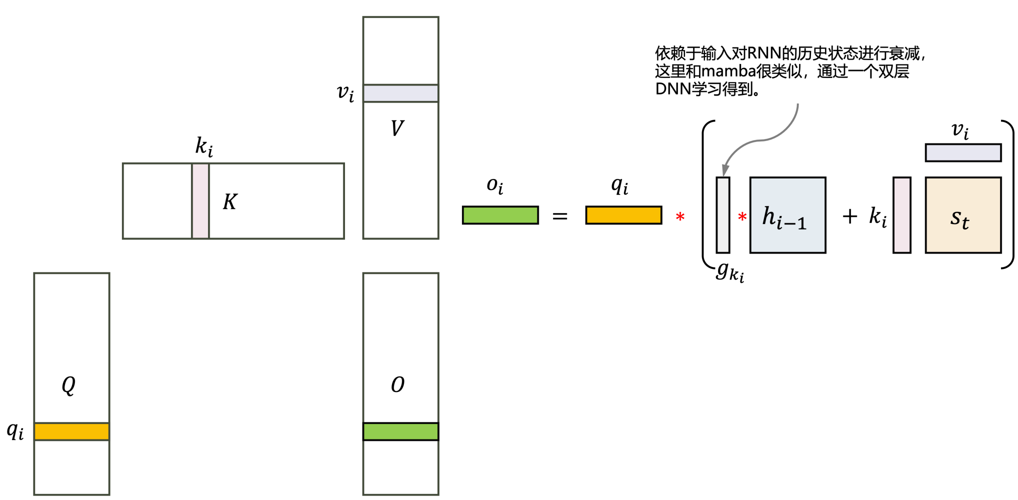


Fig 4. GLA layer以recurrent模式单步计算单时间步访问的数据

我们把GLA看做是一个RNN layer（causal形式，不attend到未来时刻，i时刻只去attend i时刻之前）由以下计算得到：

> for $i \in [0, L-1)$
>
> $h_i = h_{i-1} * \exp(g_{k_i}) + k_i \otimes v_i$ // 对状态进行衰减，与当前时刻状态相加
>
> $o_i = \mathrm{sum}\left(q_i * \gamma * h_i, \dim = -2\right)$ // 进行reduce压缩到1D

和MHA相比，linear attention独特之处在于一个for循环，而不是嵌套的for循环，就可以算出来最终的输出，这个是算法层面地对计算量和存储需求的巨大改善。

并行形式

引入G_t对状态S_t进行衰减之后，我们很容易发现上面的chunk-wise并行方式需要重新进行推导。将$S_t = G_t \odot S_{t-1} + k_t^T v_t$进行unroll展开，我们可以得到：

$$
\begin{aligned}
S_1 &= G_1 \odot S_0 + k_1^T v_1 \\
S_2 &= G_2 \odot S_1 + k_2^T v_2 \\
&= G_2 \odot \left(G_1 \odot S_0 + k_1^T v_1\right) + k_2^T v_2 \\
&= G_2 \odot G_1 \odot S_0 + G_2 \odot k_1^T v_1 + k_2^T v_2 \\
S_3 &= G_3 \odot S_2 + k_3^T v_3 \\
&= G_3 \odot \left(G_2 \odot G_1 \odot S_0 + G_2 \odot k_1^T v_1 + k_2^T v_2\right) + k_3^T v_3 \\
&= \color{red}{G_3 \odot G_2 \odot G_1} \odot S_0 + \color{red}{G_3 \odot G_2} \odot k_1^T v_1 + \color{red}{G_3} \odot k_2^T v_2 + k_3^T v_3
\end{aligned}
$$

注意观察上面的公式，假设当前时刻为t，从1到t时刻的gating factor全部都会作用于S_0，对S_0进行衰减，也就是说距离当前时刻越远的状态对t时刻的影响越小。于是当我们不以recurrent递归式的方式进行计算，假设每个时间步都从0时刻开始运算，会得到以下展开的S_t计算公式。在GLA中，为了以parameter efficient的方式得到G_t（大小$d_k \times d_v$），实际上G_t来自于一个向量α_t（大小$1 \times d_k$），即：$G_t = \left(\alpha_t^T \mathbf{1}\right)$（这里$\mathbf{1}$的大小$1 \times d_v$），于是有：

$$
\begin{aligned}
S_t &= \sum_{i=1}^{t} \left(\left(\prod_{j=i+1}^{t} G_j\right) \odot \left(k_i^T v_i\right)\right) \\
&= \sum_{i=1}^{t} \left(\left(\prod_{j=i+1}^{t} \alpha_j^T \mathbf{1}\right) \odot \left(k_i^T v_i\right)\right)
\end{aligned}
$$

令$b_t \triangleq \prod_{j=1}^{t} \alpha_j$，于是$S_t$可以写成：

$$S_t = \sum_{i=1}^{t} \left(\left(\frac{b_t}{b_i} \right)^T \mathbf{1} \right) \odot k_i^T v_i$$

$o_t = q_t S_t$，于是有：

$$\begin{aligned} o_t &= q_t \sum_{i=1}^{t} \left(\left(\frac{b_t}{b_i} \right)^T \mathbf{1} \right) \odot k_i^T v_i \\ &= \sum_{i=1}^{t} (q_t \odot b_t) \left(\frac{k_i}{b_i} \right)^T v_i \end{aligned}$$

经过这个公式变形地转化，gating factor同时作用于q_t和k_i。令$B \in (0,1)^{L \times d}$，是将b_i^T（$i = 0, \ldots, L$）进行stack，将上面的公式写成矩阵化的形式：

$$\mathbf{O} = \left((\mathbf{Q} \odot \mathbf{B}) \left(\frac{\mathbf{K}}{\mathbf{B}} \right)^T \odot \mathbf{M} \right) \mathbf{V}$$

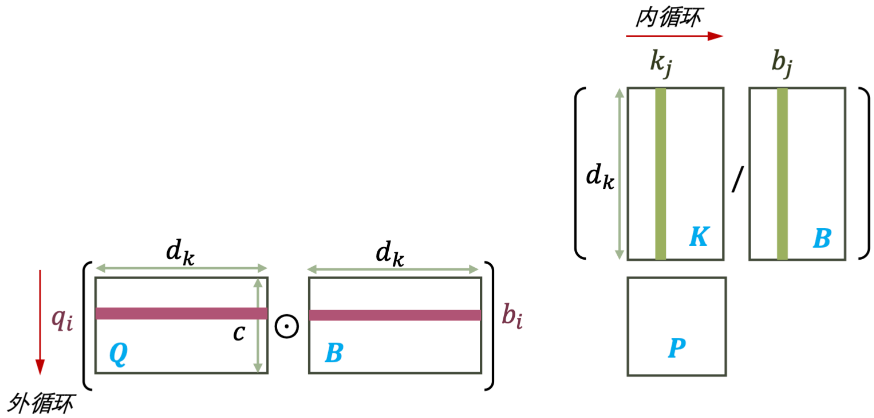
公式中每一个操作数的形状为：$[C, d_v] = [C, d_k] \odot [C, d_k] \left(\frac{[C, d_k]}{[C, d_k]} \right)^T [C, d_v]$

上式就是GLA的分块并行公式，如果我们能够提前对$\mathbf{Q}$和$\mathbf{K}$进行衰减得到$\widetilde{\mathbf{Q}} = \mathbf{Q} \odot \mathbf{B}$及$\widetilde{\mathbf{K}} = \frac{\mathbf{K}}{\mathbf{B}}$，那么$\widetilde{\mathbf{Q}}\tilde{\mathbf{K}}^T$依然可以利用矩阵乘完成。

但是$\mathbf{B}$是一个对$(0,1)$之间浮点数沿着序列长度方向上的连乘，随着序列长度增加数值会无限趋近于0，导致计算$\frac{\mathbf{K}}{\mathbf{B}}$是数值不稳定的。为了让数值计算文档，$\mathbf{P} \triangleq (\mathbf{Q} \odot \mathbf{B}) \left(\frac{\mathbf{K}}{\mathbf{B}} \right)^T$，可以将计算转换到对数空间，连乘变成对数空间的连加，于是有：

$$\mathbf{P}_{ij} = \sum_{k=1}^{d_k} \mathbf{Q}_{i,k} \mathbf{K}_{j,k} \exp \left(\log \left(\mathbf{B}_{i,k} \right) - \log \left(\mathbf{B}_{j,k} \right) \right)$$

上面这个公式的计算pattern和矩阵乘一样，如下图所示。连加就是矩阵乘的K维度reduce，但是可以注意由于A，B操作数（沿用矩阵乘的namning convention）需要首先和gating factor进行element-wise运算。



Chunked Fuse 的实现

分成了5步，前4步是4个triton kernel，最后一步是一个简单的相加，用了PyTorch的operator。下面的符号都尽量沿用了代码中对应的变量名，以便和代码对应。

Notation	Explanation	取值量级	
B	batch size	32	
L	sequence length	2048	
H	head number	4	
D_{qk}	query和key的hidden dimension	1024或者2048这样的量级	
D_v	value的hidden dimension	1024或者2048这样的量级	
BT	序列长度维度上的分块大小	*固定取16*	太小了
BK	D_{qk}维度上的分块大小	D_{qk}和64中的较小值	
NK	$NK = \frac{D_{qk}}{BK}$	D_{qk}维度上的分块数目	
BV	D_v维度上的分块大小	D_v和64中的较小值	
NV	$NV = \frac{D_v}{BV}$	D_v维度上的分块数目	

输入tensor	形状
Q	$[B, H, L, D_{qk}]$
K	$[B, H, L, D_{qk}]$
V	$[B, H, L, D_v]$

输入tensor	形状
gK	$[B, H, L, D_{qk}]$

下面表格第3列的”数据划分“，就对应了CUDA device kernel launch config中blocks的三个维度，也就是并发blocks数目。

Kernel	数据划分	theads per CTA
$g_o = \text{fwd\_decay\_cumsum}(g)$	$NK, \frac{L}{BT}, B * H$	32
$q_g, k_g = \text{prepare\_qg\_kg}(q, k, g_o)$	$NK, \frac{L}{BT}, B * H$	32
$o = \text{fused\_chunk\_gla\_fwd\_kernel}(q_g, k_g, v, g_o, o)$	$NK, NV, B * H$	64
$o_2 = \text{fwd\_inner\_chunk}(q, k, g_o)$	$NK, \frac{L}{BT}, B * H$	128
combine inner and intra chunks	/	由PyTorch operator完成，见后面的说明

fwd_decay_cumsum

> 这个kernel是为了计算论文中的$\mathbf{B}$矩阵，在序列长度维度上进行分块，实现的时候分块大小固定为16。但是由于这个kernel在序列长度上并行分布blocks，因此gating factor的累乘只在长度为16的窗口内进行。

第1个kernel是一个很小的element-wise kernel，用来计算沿着序列长度L维度的gated factor的累乘。输入的是通过低秩方法得到的gating factor $GK_{[B,H,L,D_{qk}]}$。由于多样本和多head之间完全独立，我们总是可以忽略B，H这两个全并行维度，他们最终会映射到CUDA的blocks之上并发进行处理。于是我们始终只关注如何处理一个序列。

Fig 4是fwd_decay_cumsum这个kernel处理数据的示意图，这个kernel在G的hidden维度和序列长度维度并行。

1. 每一个CTA相互独立地处理$D_{qk} \times BT$大小的数据。

2. 这个kernel的pattern是一个2D kernel，在序列长度维度上进行scan，在G的hidden dimension上没有数据依赖，可以全并发处理。

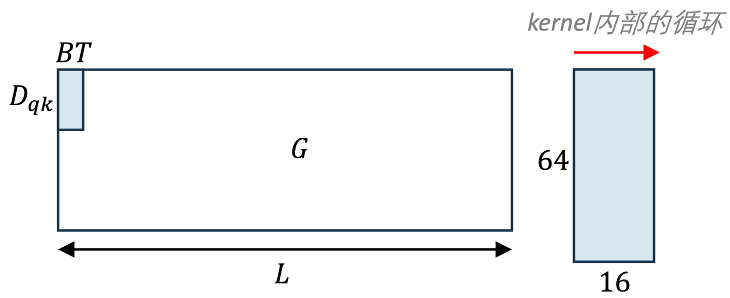


Fig 5. fwd_decay_cumsum的并行方式

这个kernel内部带有一个pattern为scan的for循环： $Y = \text{scan}\left((\vec{s}, \vec{x}) \to f, I = \vec{1}, \text{rows}(X)\right)$, $f(\vec{s}, \vec{x})$是scan携带的二元算符，公式如下：

$$f(\vec{s}, \vec{x}) = \vec{s} + \vec{x}/\ln 2$$

这个kernel**可能是为了提高并行性**，在蓝色小块内部进行了累积和运算，但是小块之间是独立的，也就是说这个kernel计算完毕后，**gating factor在每一个长度为16的local窗口内进行了累积，但是并没有在全序列长度范围内进行累积。**

prepare_qg_kg

以Q，K，V和G为输入，以和Q，K等大的Q_g和K_g为输出，给Q和K都乘以gating factor G，进行衰减。这个kernel完成的数学计算如下（实现中需要先对g计算2的幂运算，将映射到对数空间的gating factor进行还原得到sigmoid的输出，下面的公式省略了这个2的幂运算)：

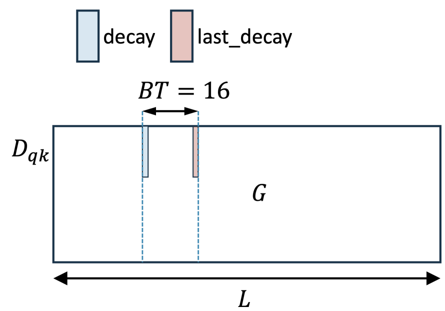
```
load(last_decay)

for i in range(BT):
  load(_q)  # [BK]
  load(_k)  # [BK]
  load(_g)  # [BK]

  _q *= _g * scale
  _k *= (last_decay - _g)

  store(_k)
  store(_g)
```

$$q_i = q_i * g_i * \gamma$$
$$k_i = k_i * (\text{last\_decay} - g_i)$$

这个kernel和 `fwd_decay_cumsum` 的分数据方式，kernel内部循环方式完全一样，也就是每个kernel依然独立地去处理Fig 4中蓝色部分的数据块。当前kernel需要读取gating factor矩阵G中D_{qk}行，BT列大小的一块数据（Fig 4蓝色方块大小）的一块数据，来看 `last_decay` 的指针偏移计算， `last_decay` 取到这块数据的最后一列。

```
last_decay = tl.load(g + i_bh * s_qk_h + (i_c * BT + BT - 1) * DK + i_k * BK + tl.arang
```

取到的数据如下图所示：



fused_chunk_gla_fwd_kernel

这个kernel的分数据方案如下，kernel 内部有**一个for**在序列长度维度进行循环。这和MHA的二重循环相比，已经极大的减少了计算的复杂度。作者在实现的时候，序列长度之上的分块固定为16，这个分块越大，重复计算的部分就会越多。

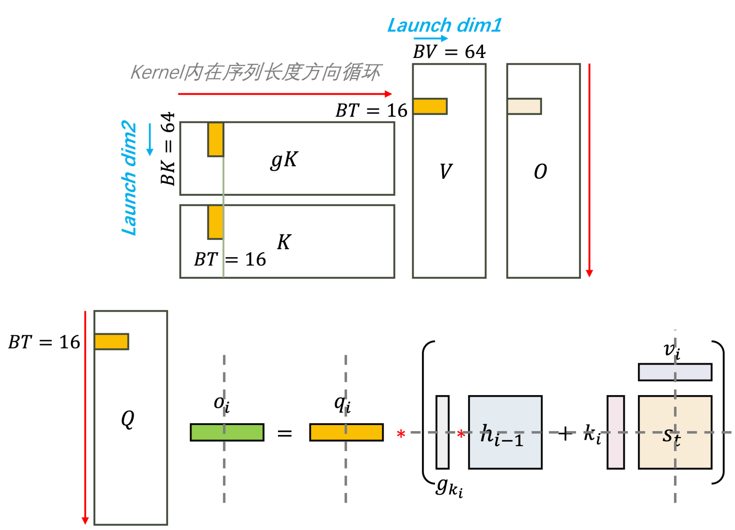


Fig 6. fused_chunk_gla_fwd_kernel的并行方式

```
d_b = g + i_bh * s_qk_h + (BT - 1) * s_qk_t + i_k * BK + tl.arange(0, BK)
load(d_b)
```

`d_b` 取到了一个序列第一个分块（跨16列）的最后一列。随着kernel内的循环在序列长度分块上移动，g_{db}总是指向当前分块的最后一列。类似于 `prepare_qg_kg` 中 `last_deacy` 的作用。

```
b_h = zeros(BK, BV)  # hidden初始化为0
for i in range(L / BT):
    load(b_k)  # b_k 形状: [BK, BT]
    load(b_v)  # b_v 形状: [BT, BV]
    load(b_q)  # b_q 形状: [BT, BK]
    load(d_b)  # d_b 形状: [BK]

    b_o = zeros(BT, BV, dtype=float32)

    b_o = b_q @ b_h  # [BT, BV] = [BT, BK] @ [BK, BV]
    b_h = b_h * d_b + b_k @ b_v  # [BK, BV] = [BK, BV] * [BK, 1] + [BK, BT] @ [BT, BV]

    store(b_o)
```

这个kernel内部的for循环在整个序列长度方向上进行循环，每个循环步处理一个分块。

$$
\begin{aligned}
o_t &= q_t \otimes h_{t-1} \\
h_t &= h_{t-1} * g_{db} + k_t \otimes v_t
\end{aligned}
$$

第2个公式中的g_{db}对应了代码中的 `d_b` 。

fwd_inner_chunk

这个kernel分数据的方式和kernel 1，2 完全相同。每个小分块之间完全的独立。以Q，K和G为输入，输出tensor的大小$[NK, B, H, \frac{L}{BT}, BT, BT]$。

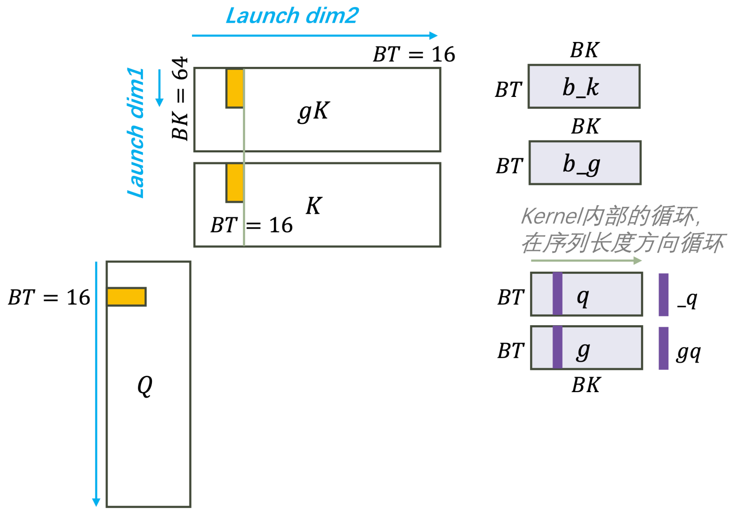


Fig 7. fwd_inner_chunk的并行方式

kernel内部的for循环在序列长度方向循环。Fig 6的右半部分是 `fwd_inner_chunk` kernel内部access数据的示意图，下面的代码是这个kernel完成的计算（下面的变量名和上图以及作者代码中的实现完全相同，方便对应回作者的代码）：

```
for i in range(BT):
  s = (_q * b_k) * (gq - b_g)  # [BT, BK]，注意这里的形状，第一个“*”和最后一个“-”是broadcast
  score = sum(s, axis=1)  # [BT, 1]，reduce运算
  score = tl.where(o_i <= i, score, 0)  # [BT, BT]， o_i = range(0, BT)，causal mask
```

combine inner and intra

这一步是用PyTorch的tensor operator完成的。

```
# A has a shape of [NK, B, H, L/BT, BT, BT]
v2 = rearrange(v, 'b h (n c) d -> b h n c d', n=num_chunk)
A = A.sum(0)  # A是`fwd_inner_chunk的输出`，[B, H, L/BT, BT, BT]
o2 = A @ v2  # [B, H, L/BT, BT, Dv]
o2 = rearrange(o2, 'b h n c d -> b h (n c) d')  # [B, H, L, Dv]
o.add_(o2)
```