深入解析 Group Query Attention（GQA）在 GemmaAttention 中的计算流程

深入解析 Group Query Attention（GQA）在 `GemmaAttention` 中的计算流程

在 **GemmaAttention** 的实现中，**Group Query Attention (GQA)** 是关键优化点之一。相较于标准的 **Multi-Head Attention (MHA)**，GQA 通过**减少 K 计算量**，提高推理效率，同时保持 注意力机制 的表现力。

本文将深入解析：

1. **GQA 的数学原理**
2. **计算 Q, K, V 的详细过程**
3. **如何共享 K, V 以减少计算**
4. **如何在 `GemmaAttention` 中执行计算**
5. **最终如何合并输出**

---

1. Group Query Attention (GQA) 的数学原理

在 **标准的多头注意力（MHA, Multi-Head Attention）** 中，每个 Query 头都有对应的 Key 头和 Value 头，即：

- **查询向量（Query, Q）** 维度：(B, L, H_q, D)
- **键向量（Key, K）** 维度：(B, L_k, H_k, D)
- **值向量（Value, V）** 维度：(B, L_k, H_k, D)

计算公式：

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{d_k}\right)V$$

其中：

- $H_q = H_k = H_v$（所有头的数量一致）
- 计算复杂度：$O(H \cdot L \cdot L_k \cdot D)$

1.1 GQA 的优化

GQA（Group Query Attention） 通过 **减少 Key-Value 头的数量**，让多个 Query 共享相同的 Key、Value：

- **查询头（Query Heads）** 仍然是 H_q 个。
- **但 Key-Value 头（KV Heads）数量减少为 H_k（通常 $H_k \ll H_q$）**。
- 每个 KV 头服务多个 Query 头，即：

$$H_q = H_k \times R$$

其中，$R = \frac{H_q}{H_k}$ 表示**每个 Key-Value 头负责多少 Query 头**。

计算复杂度对比：

- **MHA**：（$O(H_q \cdot L \cdot L_k \cdot D)$）
- **GQA**：（$O(H_k \cdot L \cdot L_k \cdot D)$）
  - 因为（$H_k \ll H_q$），所以计算量减少。

---

2. `GemmaAttention` 中 Q, K, V 的计算

在 `GemmaAttention` 的 `forward()` 方法中：

python AI写代码 登录复制

```python
qkv = self.qkv_proj(hidden_states)  # 计算 QKV
xq, xk, xv = qkv.split([self.q_size, self.kv_size, self.kv_size], dim=-1)
```

- `qkv_proj` **一次性计算 Query、Key 和 Value**，然后拆分：
  - `xq` 形状：（(B, L, H_q, D)）（Query）
  - `xk` 形状：（(B, L, H_k, D)）（Key）
  - `xv` 形状：（(B, L, H_k, D)）（Value）

然后：

python AI写代码 登录复制

```python
xq = xq.view(batch_size, -1, self.num_heads, self.head_dim)
xk = xk.view(batch_size, -1, self.num_kv_heads, self.head_dim)
xv = xv.view(batch_size, -1, self.num_kv_heads, self.head_dim)
```

- **Query 头 `H_q` 仍然是完整的多头数量。**
- **Key、Value 头 `H_k` 变少了**（减少计算量）。

---

3. GQA 共享 K, V 计算

由于 `H_k` 变少了，我们需要让多个 Query 共享相同的 Key-Value：

python AI写代码 登录复制

```python
if self.num_kv_heads != self.num_heads:
    key = torch.repeat_interleave(k_cache, self.num_queries_per_kv, dim=2)
    value = torch.repeat_interleave(v_cache, self.num_queries_per_kv, dim=2)
```

- `repeat_interleave()` **沿着注意力头维度（`dim=2`）复制 K, V**，确保每个 Query 头都能访问对应的 Key-Value 头。
- **新的 `key` 和 `value` 形状变为 ((B, L_k, H_q, D))**，匹配 Query。

---

4. 计算注意力权重

python AI写代码 登录复制

```python
q = xq.transpose(1, 2)  # (B, H_q, L, D)
k = key.transpose(1, 2)  # (B, H_q, L_k, D)
v = value.transpose(1, 2)  # (B, H_q, L_k, D)
```

- **交换 Batch 维度和 Head 维度**，让计算符合 **Batched Matrix Multiplication** 形式。

计算 QK^T：

python 复制

```python
scores = torch.matmul(q, k.transpose(2, 3))  # (B, H_q, L, L_k)
scores.mul_(self.scaling)  # 归一化
```

- 计算（QK^T）并进行（$\frac{1}{D}$）归一化。

5. Sliding Window Attention

python AI写代码 登录复制

```python
if (
    self.attn_type == gemma_config.AttentionType.LOCAL_SLIDING
    and self.sliding_window_size is not None
):
    all_ones = torch.ones_like(mask)
    sliding_mask = torch.triu(all_ones, -1 * self.sliding_window_size + 1) * \
                   torch.tril(all_ones, self.sliding_window_size - 1)
    mask = torch.where(sliding_mask == 1, mask, -2.3819763e38)
```

- **使用 `triu` 和 `tril` 构造滑动窗口注意力**：
  - 仅允许 **窗口内** 的 Query 访问 Key。
  - 其他位置填充极小值，防止 `softmax()` 赋予非零权重。

---

6. 计算最终的注意力输出

python AI写代码 登录复制

```python
scores = scores + mask
scores = F.softmax(scores.float(), dim=-1).type_as(q)

output = torch.matmul(scores, v)  # (B, H_q, L, D)
output = (output.transpose(1, 2).contiguous().view(
    batch_size, input_len, -1))
output = self.o_proj(output)
```

1. **添加 Mask 进行 `softmax()`**，计算最终注意力分布。
2. **计算 `V` 的加权求和**。
3. **转换回 `(B, L, hidden_dim)` 形式** 并通过 `o_proj` 线性变换输出。

---

7. 总结

✅ GQA 的计算流程

1. **Q, K, V 计算**：
   - `H_q` 个 Query 头
   - **`H_k` 个 KV 头（`H_k < H_q`）**
2. **让多个 Query 共享 KV**：
   - 通过 `repeat_interleave()` 复制 KV 头
   - **减少计算量**
3. **注意力计算**：
   - 计算 `QK^T`
   - 归一化 `softmax()`
4. **滑动窗口注意力（SWA）**
   - **限制 Query 访问的 Key 范围**
   - **减少计算量，适用于长序列**
5. **最终输出**
   - 计算 `V` 的加权和
   - 通过 `o_proj` 变换回 hidden_size

🚀 **GemmaAttention 通过 GQA 和 Sliding Window Attention 大幅优化计算，提高推理效率**

补充滑动窗口注意力的代码实现部分

我们来详细解析 `sliding_mask` 的构造方式以及 `to`

1. 目标：实现 Sliding Window Attention

Sliding Window Attention (SWA) 主要的目的是：

- **限制 Query 只能看到固定窗口大小范围内的 Key**
- **防止 Query 关注窗口外的 Key**
- **减少计算复杂度，适用于长序列推理**

通常，**注意力掩码（Attention Mask）** 的作用是：

- **有效的注意力权重保持原值**
- **无效的注意力分数被赋值一个极小的负数（例如 `-inf`），使得 `softmax` 之后的概率为 `0`**

---

2. 代码解析

python AI写代码 登录复制

```python
if (
    self.attn_type == gemma_config.AttentionType.LOCAL_SLIDING
    and self.sliding_window_size is not None
):
    all_ones = torch.ones_like(mask)
    sliding_mask = torch.triu(all_ones, -1 * self.sliding_window_size + 1) * \
                   torch.tril(all_ones, self.sliding_window_size - 1)
    mask = torch.where(sliding_mask == 1, mask, -2.3819763e38)
```

这里的 `mask` 变量**通常是注意力掩码（attention mask）**，它用于调整 `QK^T` 的注意力分数。

2.1 生成滑动窗口掩码

python AI写代码 登录复制

```python
all_ones = torch.ones_like(mask)
sliding_mask = torch.triu(all_ones, -1 * self.sliding_window_size + 1) * \
               torch.tril(all_ones, self.sliding_window_size - 1)
```

- `torch.triu(all_ones, -1 * self.sliding_window_size + 1)`
  - **下三角矩阵**，保留对角线以下 `sliding_window_size - 1` 个元素
- `torch.tril(all_ones, self.sliding_window_size - 1)`
  - **上三角矩阵**，保留对角线以上 `sliding_window_size - 1` 个元素
- 结果 `sliding_mask` 是 **一个包含 1 和 0 的滑动窗口矩阵**：

  python AI写代码 登录复制

  ```python
  sliding_mask =
  [[1, 1, 1, 0, 0],
   [1, 1, 1, 1, 0],
   [1, 1, 1, 1, 1],
   [0, 1, 1, 1, 1],
   [0, 0, 1, 1, 1]]
  ```

这个 `sliding_mask` 的作用是**标记 Query 可以看到的 Key 的位置**。

2.2 `torch.where()` 如何应用 `sliding_mask`

python AI写代码 登录复制

```python
mask = torch.where(sliding_mask == 1, mask, -2.3819763e38)
```

这行代码的作用是：

- **对于 `sliding_mask == 1` 的位置，保留 `mask`**

- 对于 `sliding_mask == 0` 的位置，赋值 `-2.3819763e38`（极小值，等价于 `-inf`）

换句话说：

- 窗口内的注意力权重保持原值
- 窗口外的注意力分数被设为 `-inf`，导致 `softmax` 计算时这些位置的概率变为 `0`

---

3. 为什么这样做？

在 `softmax` 计算时，所有元素都会被指数运算：

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum e^{x_j}}$$

- 如果某个 `x_i = -inf`，则 `e^{-inf} = 0`，该位置的注意力权重变为 0
- 这样 Query 只能关注滑动窗口内部的 Key，完全忽略窗口外的 Key

---

4. 总结

- `sliding_mask` 生成了一个 允许注意力关注的滑动窗口。
- `torch.where(sliding_mask == 1, mask, -inf)` 保留窗口内的权重，并屏蔽窗口外的权重。
- `softmax` 之后，窗口外的 Key 权重变为 `0`，从而实现 **Sliding Window Attention**。

✅ 关键点

1. 不是把滑动窗口内部的元素变成 `-inf`，而是外部的 Key 设为 `-inf`，确保 `softmax` 之后 Query 只能关注窗口内部。
2. 正确使用 `torch.triu()` 和 `torch.tril()` 生成滑动窗口掩码，保证 Query 只能看到最近的 `sliding_window_size` 个 Key。
3. 屏蔽窗口外的 Key 位置，确保 `softmax` 之后的权重为 `0`，达到局部注意力机制的目的。

🚀 这样就成功实现了 Sliding Window Attention！ 🎯

附录

官方gemma　attention的源代码

c　　AI写代码　　登录复制

```c
class GemmaAttention(nn.Module):

    def __init__(
        self,
        hidden_size: int,
        num_heads: int,
        num_kv_heads: int,
        attn_logit_softcapping: Optional[float],
        query_pre_attn_scalar: Optional[int],
```

展开

后记

2025年2月24日16点16分于上海，在GPT4o大模型　辅助下完成。