1.1 大模型加速

1.1.1 访存优化

$$
\begin{aligned}
O &= softmax(\frac{QK^T}{\sqrt{d}})V = \\
&= softmax\left(\begin{bmatrix} q_1 \\ q_2 \\ \vdots \\ q_n \end{bmatrix} \begin{bmatrix} k_1^T & k_2^T & \dots & k_m \end{bmatrix} \frac{1}{\sqrt{d}}\right) \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_m \end{bmatrix} \\
&= softmax\left(\begin{bmatrix} q_1k_1^T/\sqrt{d} & q_1k_2^T/\sqrt{d} & \dots & q_1k_m^T/\sqrt{d} \\ q_2k_1^T/\sqrt{d} & q_2k_2^T/\sqrt{d} & \dots & q_2k_m^T/\sqrt{d} \\ \vdots & \vdots & \ddots & \vdots \\ q_nk_1^T/\sqrt{d} & q_nk_2^T/\sqrt{d} & \dots & q_nk_m^T/\sqrt{d} \end{bmatrix}\right) \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_m \end{bmatrix} \\
&= \left[\frac{\exp(q_ik_j^T/\sqrt{d})}{\sum_j \exp(q_ik_j^T/\sqrt{d})}\right]_{ij} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_m \end{bmatrix} = \left[\frac{\exp(q_ik_j^T)}{\sum_j \exp(q_ik_j^T)}\right]_{ij} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_m \end{bmatrix}
\end{aligned}
$$

上面的运算的实现过程如 tu1.1. 可以看到需要多次在不同显存 (HBM 和 SRAM) 之间 IO 数据。FlashAttention [1] 采用分块的方法实现上面的运算，每个分块运算都是一次性写入 GPU

Algorithm 0 Standard Attention Implementation

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM.
1: Load $\mathbf{Q}, \mathbf{K}$ by blocks from HBM, compute $\mathbf{S} = \mathbf{QK}^\top$, write $\mathbf{S}$ to HBM.
2: Read $\mathbf{S}$ from HBM, compute $\mathbf{P} = \mathrm{softmax}(\mathbf{S})$, write $\mathbf{P}$ to HBM.
3: Load $\mathbf{P}$ and $\mathbf{V}$ by blocks from HBM, compute $\mathbf{O} = \mathbf{PV}$, write $\mathbf{O}$ to HBM.
4: Return $\mathbf{O}$.

图 1.1: self attention 计算过程

的 SRAM 内存中完成并完成运算，减少内存 IO。矩阵乘法的分块运算好说，但是 softmax 涉及到要计算全部的元素是个难点。我们现在来看看 softmax 如何分块运算，对于向量 $\mathbf{x} \in R^{2n}$，softmax 计算如下：

$$
softmax(\mathbf{x})_i = \frac{\exp(\mathbf{x}_i)}{\sum_{j=1}^{2n} \exp(\mathbf{x}_j)} = \frac{\exp(\mathbf{x}_i)\exp(-\max(\mathbf{x}))}{\sum_{j=1}^{2n} \exp(\mathbf{x}_j)\exp(-\max(\mathbf{x}))} = \frac{\exp(\mathbf{x}_i - \max(\mathbf{x}))}{\sum_{j=1}^{2n} \exp(\mathbf{x}_j - \max(\mathbf{x}))}
$$

现在我们把 $\mathbf{x}$ 分为两块 $\mathbf{x} = [\mathbf{x}^1\ \mathbf{x}^2], \mathbf{x}^1 \in R^n, \mathbf{x}^2 \in R^n$，先分别计算两块的 softmax，所以有：

$$
softmax(\mathbf{x}^1)_i = \frac{\exp(\mathbf{x}_i^1 - \max(\mathbf{x}^1))}{\sum_{j=1}^{n} \exp(\mathbf{x}_j^1 - \max(\mathbf{x}^1))} \qquad softmax(\mathbf{x}^2)_i = \frac{\exp(\mathbf{x}_i^2 - \max(\mathbf{x}^2))}{\sum_{j=n+1}^{2n} \exp(\mathbf{x}_j^2 - \max(\mathbf{x}^2))}
$$

可以看出，$softmax(\mathbf{x}^1)_i$ 和 $softmax(\mathbf{x}^2)_i$ 都不是最终的正确值，我们称之为局部的 softmax。我们先计算 $softmax(\mathbf{x}^1)_i$，并且在计算完毕后更新到下面两个标量：

$$\begin{aligned}
&\mathbf{xm}_{new} = \max(\mathbf{xm}_{old}, \max(\mathbf{x}^1)) \\
&\mathbf{xs} = \mathbf{xs} + \textstyle\sum_{j=1}^{n} \exp(\mathbf{x}_j^1 - \max(\mathbf{x}^1))
\end{aligned}$$

其中 $\mathbf{xm}$ 和 $\mathbf{xs}$ 初始为 0，接着类似地计算 $softmax(\mathbf{x}^2)_i$，然后更新

$$\begin{aligned}
&\mathbf{xm}_{new} = \max(\mathbf{xm}_{old}, \max(\mathbf{x}^2)) \\
&\mathbf{xs} = \exp(\mathbf{xm}_{new} - \mathbf{xm}_{old}) \star \mathbf{xs} + \exp(\mathbf{xm}_{new} - \max(\mathbf{x}^2)) \star \textstyle\sum_{j=n+1}^{2n} \exp(\mathbf{x}_j^2 - \max(\mathbf{x}^2))
\end{aligned}$$

此时再做如下运算：

$$\begin{aligned}
softmax(\mathbf{x}^1)_i &= softmax(\mathbf{x}^1)_i \star \exp(-\mathbf{xm}_{new} + \mathbf{xm}_{old})/\mathbf{xs} \\
softmax(\mathbf{x}^2)_i &= softmax(\mathbf{x}^2)_i \star \exp(-\mathbf{xm}_{new} + \max(\mathbf{x}^2))/\mathbf{xs}
\end{aligned}$$

此时 softmax 表示最终的正确值。这就是 FlashAttention 中对 softmax 动态更新的本质。FlashAttention 的计算步骤如下：

- step0. 在 HBM 上初始化 O，对 Q, K, V 三个矩阵根据 SRAM 的大小按行分块。
- step1. 在 SRAM 上执行分块之间的矩阵乘法 $Q_i K_j^T$。
- step2. 按照上述的原理动态计算 softmax。
- step3. 将动态计算的 softmax 与块 V_j 相乘。
- step4. 动态更新 HBM 内存上的 O 矩阵

FlashAttention 加速效果见图1.2，可以看到取得了 3 倍的加速。

Model implementations	OpenWebText (ppl)	Training time (speedup)
GPT-2 small - Huggingface [87]	18.2	9.5 days (1.0×)
GPT-2 small - Megatron-LM [77]	18.2	4.7 days (2.0×)
GPT-2 small - FlashAttention	18.2	**2.7 days (3.5×)**
GPT-2 medium - Huggingface [87]	14.2	21.0 days (1.0×)
GPT-2 medium - Megatron-LM [77]	14.3	11.5 days (1.8×)
GPT-2 medium - FlashAttention	14.3	**6.9 days (3.0×)**

图 1.2: FlashAttention 加速效果

1.1.2 KV Cache

我们再回顾下 decoder 中 attention 的计算过程，设输入序列是长度为 T 的序列 $\{x_t\}, t = 1, \dots, T$，我们有 H 个 Head：

$$
\begin{aligned}
& q_t^h = W_q^h x_t^T,\ k_t^h = W_k^h x_t^T,\ v_t^h = W_v^h x_t^T, && \text{QKV 变换} \\
& o_t^h = \sum_{j<t} \frac{\exp(q_t^h k_j^T)}{\sum_{j<t} \exp(q_t^h k_j^T)} v_j^h && \text{Causal Attention} \\
& O^h = \begin{bmatrix} o_1^h \\ o_2^h \\ \vdots \\ o_m^h \end{bmatrix} \qquad \hat{X} = W_o \begin{bmatrix} O_1^T \\ \vdots \\ O_H^T \end{bmatrix}
\end{aligned}
\tag{1.1}
$$

其中 $t = 1, \dots, T, h = 1, \dots, H$。Decoder 在预测时是一个一个 token 地预测，每次预测时是根据前面已经预测得到的 token 一起输入得到下一个 token，这就存在重复计算的问题。比如输入" 中国的首都"，我们期望得到的输出为" 是北京"，于是：

- step1. 将" 中国的首都" 输入，得到输出" 是"。
- step2. 将" 中国的首都是" 输入，得到输出" 北"。
- step3. 将" 中国的首都是北" 输入，得到输出" 京"。

每个 step 输入都要经过复杂的 attention 运算，可以看到" 中国的首都" 被重复输入了三次，但是从式1.1中可以看到 X 矩阵对应" 中国的首都" 这个 token 的行 embedding 向量没有变化，那么经过线性变换得到的 Q_h, K_h, V_h 对应的行其实也不会变化。新预测出来的第 $t+1$ 个 token，并不会影响到已经算好的 $k_{<t}^h, v_{<t}^h$，因此这部分结果我们可以缓存下来供后续生成调用，避免不必要的重复计算，这就是所谓的 KV Cache 。KV Cache 是 Transformer 推理性能优化的一项重要工程化技术，各大推理框架都已实现并将其进行了封装。

LLM 推理时，随着模型越来越大，KV Cache 也越来越大。KV Cache 不仅依赖于模型的体量，还依赖于模型的输入长度，也就是在推理过程中是动态增长的，当 Context 长度足够长时，它的大小就会占主导地位，可能超出一张卡甚至一台机 (8 张卡) 的总显存量。MQA(Multi-Query Attention) [2] 的思路很简单，直接让所有 Head 共享同一个 W_k 和 W_v 矩阵，每个头只单独保留了一个单独的 W_q^h 矩阵，如此依赖这部分显存的占用只有原来的 $1/H$。有人担心 MQA 对 KV Cache 的压缩太严重，以至于会影响模型的学效果。GQA(Grouped-Query Attention) [3] 的思想也很朴素，将所有 Head 分为多个组，每组共享同一个 W_k, W_v。

参考文献

[1] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.

[2] Noam Shazeer. Fast transformer decoding: One write-head is all you need, 2019.

[3] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints, 2023.