清华大学
高级机器学习
2024 年秋季学期

小作业-LLM 代码复现

杨天行 2024310659 2024 年 12 月 14 日

1 实验任务

- 根据课上所讲内容以及相关的阅读材料了解当前大语言模型的基础结构，包括但不仅限于 MultiGroup Attention、RMSNorm、SwiGLU、RoPE 等。
- 基于给定的代码模板进行补全，复现 GLM4 的基础代码，支持推理用的 KV-Cache 等参数。Attention、RMSNorm 等主要的模块需要手动实现，其余如 Softmax，Linear 等基础模块可以直接调用 torch 自带的函数。
- 加载 GLM4-9B 官方 hugging face 权重代码，并通过改名的方式将其权重转换到自己的模型中。保证模型在使用 hugging face 的 generation 接口后可以进行正常对话。(提示：作业用国内的 hugging face 镜像)
- 尝试将 attention 部分改为 flash attention，并测试模型在不同长度下的空间、时间加速比。

2 Checkpoint 转换

2.1 GLM-4-9b 模型结构分析

从 Hugging Face 上下载 GLM-4-9b 的权重，直接使用 AutoModel 类加载，输出模型 state dict 中的 keys 以及 values 的 shape 如下（其中 x 为 0 ∼ 39)：

```
transformer.embedding.word_embeddings.weight torch.Size([151552, 4096])
transformer.rotary_pos_emb.inv_freq torch.Size([32])

transformer.encoder.layers.x.input_layernorm.weight torch.Size([4096])
transformer.encoder.layers.x.self_attention.query_key_value.weight torch.Size([4608,
    4096])
transformer.encoder.layers.x.self_attention.query_key_value.bias torch.Size([4608])
transformer.encoder.layers.x.self_attention.dense.weight torch.Size([4096, 4096])
transformer.encoder.layers.x.post_attention_layernorm.weight torch.Size([4096])
transformer.encoder.layers.x.mlp.dense_h_to_4h.weight torch.Size([27392, 4096])
transformer.encoder.layers.x.mlp.dense_4h_to_h.weight torch.Size([4096, 13696])

transformer.encoder.final_layernorm.weight torch.Size([4096])
transformer.output_layer.weight torch.Size([151552, 4096])
```

在 GLM-4 的技术报告[1]中，对 GLM-4 模型的结构描述如下：

The recent GLM-4 model adopts the following architecture design choices.

[1]https://arxiv.org/abs/2406.12793

- **No Bias Except QKV**: To increase training speed, we removed all bias terms with the exception of the biases in Query, Key, and Value (QKV) matrices of the attention layers. In doing so, we observed a slight improvement in length extrapolation.
- **RMSNorm and SwiGLU**: We adopted RMSNorm and SwiGLU to replace LayerNorm and ReLU, respectively. These two strategies brought better model performance.
- **Rotary positional embeddings (RoPE)**: We extended the RoPE to a two-dimensional form to accommodate the 2D positional encoding in GLM.
- **Group Query Attention (GQA)**: We replaced Multi-Head Attention (MHA) with Group Query Attention (GQA) to cut down on the KV cache size during inference. Given GQA uses fewer parameters than MHA, we increased the FFN parameter count to maintain the same model size, i.e., setting d_{ffn} to 10/3 of the hidden size.

结合上面输出的模型权重，可以总结出如下信息：

- GLM-4-9b 的词表大小为 151552，将每个 token 映射到 4096 维的向量，对应权重中出现的两个 [151552, 4096] 的矩阵，分别用于编码与解码
- 权重中大小为 [4096] 的 tensor 是 RMSNorm Layer 的 scaling factor
- 在 Attention 机制中使用 QKV 共享权重矩阵，大小为 [4608, 4096]；对应的 bias 矩阵为 [4608]

2.2 State Dict 转换

写好 run_glm4.py 中各个组件的代码后，按同样方法输出模型 state dict 中的 keys 以及 values 的 shape 如下（其中 x 为 0 ~ 39）：

```
transformer.word_embedding.weight torch.Size([151552, 4096])
transformer.model.layers.x.self_attention.query_key_value.weight torch.Size([4608, 4096])
transformer.model.layers.x.self_attention.query_key_value.bias torch.Size([4608])
transformer.model.layers.x.self_attention.dense.weight torch.Size([4096, 4096])
transformer.model.layers.x.mlp.fc1.weight torch.Size([27392, 4096])
transformer.model.layers.x.mlp.fc2.weight torch.Size([4096, 13696])
transformer.model.layers.x.input_layernorm.gamma torch.Size([4096])
transformer.model.layers.x.post_attention_layernorm.gamma torch.Size([4096])
transformer.model.final_layernorm.gamma torch.Size([4096])
transformer.output_layer.weight torch.Size([151552, 4096])
transformer.rotary_pos_emb.inv_freq torch.Size([32])
```

分析这里 state dict 与 hugging face 上原始模型的 state dict 之间的差异，可以写出二者之间的转换脚本（通过识别矩阵名称进行改名）：

```python
def convert_ckpt():
    huggingface_model = AutoModelForCausalLM.from_pretrained(
        "THUDM/glm-4-9b-chat",
        torch_dtype=torch.bfloat16,
        low_cpu_mem_usage=True,
        trust_remote_code=True,
        device_map="auto"
    ).eval()

    model_dict = huggingface_model.state_dict()
    new_model_dict = {}
    for k, v in model_dict.items():
        if k == "transformer.embedding.word_embeddings.weight":
            new_model_dict["transformer.word_embedding.weight"] = v
        elif k == "transformer.rotary_pos_emb.inv_freq":
            new_model_dict["transformer.rotary_pos_emb.inv_freq"] = v
        elif "transformer.encoder.layers" in k:
            new_k = k.replace("transformer.encoder.layers", "transformer.model.layers")
            new_k = new_k.replace("mlp.dense_h_to_4h", "mlp.fc1")
            new_k = new_k.replace("mlp.dense_4h_to_h", "mlp.fc2")
            new_k = new_k.replace("input_layernorm.weight", "input_layernorm.gamma")
            new_k = new_k.replace("post_attention_layernorm.weight",
                "post_attention_layernorm.gamma")
            new_model_dict[new_k] = v
        elif k == "transformer.encoder.final_layernorm.weight":
            new_model_dict["transformer.model.final_layernorm.gamma"] = v
        else:
            new_model_dict[k] = v

    torch.save(new_model_dict, "glm4.pt")
```

3 基础模块实现

3.1 MLP

按照代码中的要求：

> MLP will take the input with h hidden state, project it to 4*h hidden dimension, perform nonlinear transformation, and project the state back into h hidden dimension.

按照 GLM-4-9b 技术报告的说法使用 SwiGLU，除了 qkv 之外都不带 bias。用两层的 MLP 实现，中间层的宽度按照 config.hidden_size 进行配置，第一个全连接层投影至 2*config.hidden_size 的长度。

```
class MLP(nn.Module):
    def __init__(self, config, device=None, dtype=torch.bfloat16):
        super(MLP, self).__init__()
        self.hidden_size = config.hidden_size
        self.intermediate_size = config.ffn_hidden_size

        self.fc1 = nn.Linear(self.hidden_size, 2 * self.intermediate_size,
           device=device, dtype=dtype, bias=False)
        self.fc2 = nn.Linear(self.intermediate_size, self.hidden_size, device=device,
           dtype=dtype, bias=False)
```

```python

    def forward(self, hidden_states):
        x = self.fc1(hidden_states)
        x_l, x_g = x.chunk(2, dim=-1)
        x = F.silu(x_l) * x_g
        output = self.fc2(x)
        return output
```

3.2 RMSNorm

RMSNorm 是计算输入向量中元素的均方根：$RMS(x) = \sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2}$ 将每个元素除以计算得到的 RMS 值，再与缩放因子 γ 相乘，其中 γ 是可学习的参数。

```python
class RMSNorm(nn.Module):
    def __init__(self, normalized_shape, eps=1e-5, device=None, dtype=None, **kwargs):
        super(RMSNorm, self).__init__()
        self.normalized_shape = normalized_shape
        self.eps = eps
        self.gamma = nn.Parameter(torch.ones(normalized_shape, device=device,
            dtype=dtype))

    def forward(self, hidden_states: torch.Tensor):
        input_dtype = hidden_states.dtype
        rms = hidden_states.to(torch.float32).pow(2).mean(-1, keepdim=True)
        hidden_states = hidden_states * torch.rsqrt(rms + self.eps)
        return (self.gamma * hidden_states).to(input_dtype)
```

3.3 Attention

实现 Attention 算子。首先计算投影尺寸 =kv 通道数 $\times$ 注意力头数，以及注意力得分计算 $\frac{QK^T}{\sqrt{d_T}}$。使用 mask_fill 将不可见位置的注意力得分设置为 ∞，使用 softmax 进行归一化，context layer 为注意力分数与 value 的乘积。

```python
class Attention(torch.nn.Module):
    def __init__(self, config, layer_number):
        super(Attention, self).__init__()
        self.config = config
        self.layer_number = layer_number
        self.is_causal = True
        # 计算投影尺寸
        projection_size = config.kv_channels * config.num_attention_heads

        # 每层的维度计算
        self.hidden_size_per_partition = projection_size
        self.hidden_size_per_attention_head = projection_size //
            config.num_attention_heads
        self.num_attention_heads_per_partition = config.num_attention_heads

        self.norm_factor = math.sqrt(self.hidden_size_per_attention_head)
        coeff = self.layer_number
        self.norm_factor *= coeff
        self.coeff = coeff

        self.attention_dropout = torch.nn.Dropout(config.attention_dropout)

    def forward(self, query_layer, key_layer, value_layer, attention_mask):
        # [b, np, sq, sk]
        output_size = (query_layer.size(0), query_layer.size(1), query_layer.size(2), key_layer.size(2))
        query_layer = query_layer.view(output_size[0] * output_size[1], output_size[2], -1)
        key_layer = key_layer.view(output_size[0] * output_size[1], output_size[3], -1)

        # 计算注意力得分并进行缩放
        attention_scores = torch.matmul(query_layer, key_layer.transpose(-1, -2))
        attention_scores /= self.norm_factor

        attention_scores = attention_scores.view(*output_size)

        attention_scores = attention_scores.float() * self.coeff
        if attention_mask is None and attention_scores.shape[2] == attention_scores.shape[3]:
            attention_mask = torch.ones(output_size[0], 1, output_size[2], output_size[3],
                                        device=attention_scores.device, dtype=torch.bool)
            attention_mask.tril_()
            attention_mask = ~attention_mask
        if attention_mask is not None:
            attention_scores = attention_scores.masked_fill(attention_mask, float("-inf"))
        attention_probs = F.softmax(attention_scores, dim=-1)
        attention_probs = attention_probs.type_as(value_layer)

        attention_probs = self.attention_dropout(attention_probs)

        output_size = (value_layer.size(0), value_layer.size(1), query_layer.size(1), value_layer.size(3))
        value_layer = value_layer.view(output_size[0] * output_size[1], value_layer.size(2), -1)
        attention_probs = attention_probs.view(output_size[0] * output_size[1], output_size[2], -1)
        context_layer = torch.bmm(attention_probs, value_layer)
        context_layer = context_layer.view(*output_size)
        context_layer = context_layer.transpose(1, 2).contiguous()
        new_context_layer_shape = context_layer.size()[:-2] +
            (self.hidden_size_per_partition,)
        context_layer = context_layer.reshape(*new_context_layer_shape)

        return context_layer
```

3.4 Attention Block

按照 GLM-4 使用 Group Query Attention 的架构，qkv_hidden_size 的大小为 query 的大小加上 2 倍 num_query_groups 的 kv hidden dim 大小。在 forward 过程中计算得到 maxed_x_layer 后划分为 qkv layers，query layer 与 key layer 进行位置编码后使用 attention 算子计算（key_layer 与 value_layer 需要对齐 query_layer 的维度）。最后合并多头输出并通过最后的线性层。

```
class AttentionBlock(torch.nn.Module):

    def __init__(self, config, layer_number, device=None, dtype=torch.bfloat16):
        super(AttentionBlock, self).__init__()
        self.projection_size = config.kv_channels * config.num_attention_heads
        self.layer_number = layer_number
        # Per attention head and per partition values.
        self.hidden_size_per_attention_head = self.projection_size //
            config.num_attention_heads # 每个头的键值维度
        self.num_attention_heads_per_partition = config.num_attention_heads

        self.multi_query_attention = config.multi_query_attention
        self.qkv_hidden_size = 3 * self.projection_size
        # 计算QKV隐藏层大小
        if self.multi_query_attention:
            self.num_multi_query_groups_per_partition = config.multi_query_group_num
            self.qkv_hidden_size = (
                    self.projection_size + 2 * self.hidden_size_per_attention_head *
                        config.multi_query_group_num
            )
        # 定义qkv共用参数
        self.query_key_value = nn.Linear(config.hidden_size, self.qkv_hidden_size,
                                         bias=config.add_bias_linear or
                                             config.add_qkv_bias,
                                         device=device, dtype=dtype
                                         )


        # Attention算子
        self.core_attention = Attention(config, self.layer_number)
        self.dense = nn.Linear(self.projection_size, config.hidden_size,
            bias=config.add_bias_linear,
                               device=device, dtype=dtype
                               )

    def forward(
            self, hidden_states, attention_mask, rotary_pos_emb, kv_cache=None,
                use_cache=True
    ):
        # 计算qkv投影，并分割为query, key, value
        mixed_x_layer = self.query_key_value(hidden_states)

        if self.multi_query_attention:
            (query_layer, key_layer, value_layer) = mixed_x_layer.split(
                [
                    self.num_attention_heads_per_partition *
                        self.hidden_size_per_attention_head,
                    self.num_multi_query_groups_per_partition *
                        self.hidden_size_per_attention_head,
                    self.num_multi_query_groups_per_partition *
                        self.hidden_size_per_attention_head,
                ],
                dim=-1,
            )
            query_layer = query_layer.view(
                query_layer.size()[:-1] + (self.num_attention_heads_per_partition,
                    self.hidden_size_per_attention_head)
            )
            key_layer = key_layer.view(
                key_layer.size()[:-1] + (self.num_multi_query_groups_per_partition,
                    self.hidden_size_per_attention_head)
            )
            value_layer = value_layer.view(
                value_layer.size()[:-1]
                + (self.num_multi_query_groups_per_partition,
                    self.hidden_size_per_attention_head)
            )
        else:
            new_tensor_shape = mixed_x_layer.size()[:-1] + \
                               (self.num_attention_heads_per_partition,
                                3 * self.hidden_size_per_attention_head)
            mixed_x_layer = mixed_x_layer.view(*new_tensor_shape)

            # [b, sq, np, 3 * hn] --> 3 [b, sq, np, hn]
            (query_layer, key_layer, value_layer) =
                split_tensor_along_last_dim(mixed_x_layer, 3)

        query_layer, key_layer, value_layer = [k.transpose(1, 2) for k in [query_layer,
            key_layer, value_layer]]

        # 位置编码
        if rotary_pos_emb is not None:
            query_layer = apply_rotary_pos_emb(query_layer, rotary_pos_emb)
            key_layer = apply_rotary_pos_emb(key_layer, rotary_pos_emb)

        # adjust key and value for inference
        if kv_cache is not None:
            cache_k, cache_v = kv_cache
            key_layer = torch.cat((cache_k, key_layer), dim=2)
            value_layer = torch.cat((cache_v, value_layer), dim=2)
        if use_cache:
            if kv_cache is None:
                kv_cache = torch.cat((key_layer.unsqueeze(0).unsqueeze(0),
                    value_layer.unsqueeze(0).unsqueeze(0)),
                                     dim=1)
            else:
                kv_cache = (key_layer, value_layer)
        else:
            kv_cache = None

        if self.multi_query_attention:
            key_layer = key_layer.unsqueeze(2)
            key_layer = key_layer.expand(
                -1, -1, self.num_attention_heads_per_partition //
                    self.num_multi_query_groups_per_partition, -1, -1
            )
            key_layer = key_layer.contiguous().view(
                key_layer.size()[:1] + (self.num_attention_heads_per_partition,) +
                    key_layer.size()[3:]
            )
            value_layer = value_layer.unsqueeze(2)
            value_layer = value_layer.expand(
                -1, -1, self.num_attention_heads_per_partition //
                    self.num_multi_query_groups_per_partition, -1, -1
            )
            value_layer = value_layer.contiguous().view(
                value_layer.size()[:1] + (self.num_attention_heads_per_partition,) +
                    value_layer.size()[3:]
            )
        # 合并多头输出，并通过最终线性层
        context_layer = self.core_attention(query_layer, key_layer, value_layer,
            attention_mask)
        output = self.dense(context_layer)

        return output, kv_cache
```

3.5 Layer

Layer 类实现了一个完整的 Transformer 基本层，包括 self-attention、feed-forward network、residual connections 与 layernorm。输入的 hidden_states 进行层归一化后输入 self_attention 层，进行残差连接

后再次层归一化，最后通过前馈网络与第二个参差连接。
残差连接的配置在 GLM-4 技术报告里没有具体说明，可以
在`https://huggingface.co/THUDM/glm-4-9b-chat/blob/main/config.json` 里找到模型推理的详细配置，按照完整的配置写对应的残差连接。

```python
class Layer(torch.nn.Module):
    def __init__(self, config, device):
        super(Layer, self).__init__()
        self.dtype = torch.bfloat16 if config.torch_dtype == "bfloat16" else
            torch.float32
        self.self_attention = AttentionBlock(config, device=device, dtype=self.dtype)
        self.mlp = MLP(config, device=device, dtype=self.dtype)

        self.input_layernorm = RMSNorm(
            config.hidden_size,
            eps=config.layernorm_epsilon,
            device=device,
            dtype=self.dtype)
        self.post_attention_layernorm = RMSNorm(
            config.hidden_size,
            eps=config.layernorm_epsilon,
            device=device,
            dtype=self.dtype)

    def forward(
            self, hidden_states, attention_mask, rotary_pos_emb, kv_cache=None,
                use_cache=True,
    ):
        normed_hidden_states = self.input_layernorm(hidden_states)
        attention_output, new_kv_cache = self.self_attention(
            normed_hidden_states, attention_mask, rotary_pos_emb, kv_cache, use_cache
        )

        residual_connection = hidden_states + attention_output
        normed_residual_connection = self.post_attention_layernorm(residual_connection)

        ffn_output = self.mlp(normed_residual_connection)
        output = residual_connection + ffn_output

        return output, new_kv_cache
```

3.6 Transformer

需要构建的参数包括每一层的 Layer 以及最后的 post_layer_norm（若启用）。在 forward 方法中依次推理 hidden_states，每一层输出作为下一层输入，在 decoding 环境下更新 kv cache。

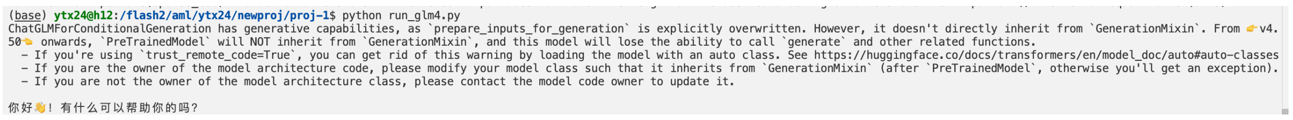
```python
class Transformer(torch.nn.Module):
    def __init__(self, config, device):
        super(Transformer, self).__init__()
        self.num_layers = config.num_layers
        self.post_layer_norm = config.post_layer_norm
        self.dtype = torch.bfloat16 if config.torch_dtype == "bfloat16" else
            torch.float32
        self.layers = nn.ModuleList([Layer(config, device=device) for _ in
            range(self.num_layers)])

        if self.post_layer_norm:
            self.final_layernorm = RMSNorm(
                config.hidden_size,
                eps=config.layernorm_epsilon,
                device=device,
                dtype=self.dtype
            )

    def forward(
            self, hidden_states, attention_mask, rotary_pos_emb, kv_caches=None,
            use_cache: Optional[bool] = True,
            output_hidden_states: Optional[bool] = False,
    ):
        # 存储每一层隐藏状态
        all_hidden_states = [] if output_hidden_states else None

        # 初始化kv cache
        new_kv_caches = [] if kv_caches is not None else None

        for layer_idx, layer in enumerate(self.layers):
            if output_hidden_states:
                all_hidden_states.append(hidden_states)

            # 如果已有kv cache
            layer_kv_cache = kv_caches[layer_idx] if kv_caches is not None else None
            # 前向传播
            hidden_states, new_layer_kv_cache = layer(
                hidden_states, attention_mask, rotary_pos_emb,
                kv_cache=layer_kv_cache, use_cache=use_cache
            )
            # 保存新的kv cache
            if new_kv_caches is not None:
                new_kv_caches.append(new_layer_kv_cache)

        if self.post_layer_norm:
            hidden_states = self.final_layernorm(hidden_states)
        if output_hidden_states:
            all_hidden_states.append(hidden_states)

```

```python
        if new_kv_caches is not None:
            return hidden_states, tuple(new_kv_caches)
        else:
            return hidden_states, None
```

4 运行模型 & 对话示例

首先手动设置`HF_HOME`环境变量，下载 GLM-4-9b 模型至指定位置，再设置`TRANSFORMERS_OFFLINE=1`使代码离线运行。直接离线加载 AutoTokenizer 会报错 ValueError: Unrecognized configuration class <class 'transformers_modules.THUDM.glm-4-9b-chat.(omitted).configuration_chatglm.ChatGLMConfig'> to build an AutoTokenizer.，搜索相关 issue 后发现 hugging face 自动下载缺少数个文件，手动下载放到对应目录即可。

成功运行的截图如下：



```
(base) ytx24@h12:/flash2/aml/ytx24/newproj/proj-1$ python run_glm4.py
ChatGLMForConditionalGeneration has generative capabilities, as `prepare_inputs_for_generation` is explicitly overwritten. However, it doesn't directly inherit from `GenerationMixin`. From 👉v4.
50👈 onwards, `PreTrainedModel` will NOT inherit from `GenerationMixin`, and this model will lose the ability to call `generate` and other related functions.
  - If you're using `trust_remote_code=True`, you can get rid of this warning by loading the model with an auto class. See https://huggingface.co/docs/transformers/en/model_doc/auto#auto-classes
  - If you are the owner of the model architecture code, please modify your model class such that it inherits from `GenerationMixin` (after `PreTrainedModel`, otherwise you'll get an exception).
  - If you are not the owner of the model architecture class, please contact the model code owner to update it.

你好👋！有什么可以帮助你的吗？
```

图 1: 运行对话截图

5 Flash Attention 实现与测试

5.1 Flash Attention 2 实现

这里实现`FlashAttention2`类，继承自之前的`Attention`，调用了`flash_attn`库进行计算。首先对输入的 qkv layer 进行 pad 操作，对齐一个 batch 中的输入长度，并对应地对 attention mask 进行修改，使用 flash attention 2 库提供的 attention 算子进行计算，最后进行 unpad 操作。

```python
class FlashAttention2(Attention):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self._flash_attn_uses_top_left_mask = False

    def forward(self, query_states, key_states, value_states, attention_mask):
        query_states = query_states.transpose(1, 2)
        key_states = key_states.transpose(1, 2)
        value_states = value_states.transpose(1, 2)
        batch_size, query_length = query_states.shape[:2]
        if not self._flash_attn_uses_top_left_mask:
            causal = self.is_causal
        else:
            causal = self.is_causal and query_length != 1
        dropout = self.config.attention_dropout if self.training else 0.0
        # Contains at least one padding token in the sequence
        if attention_mask is not None:
            query_states, key_states, value_states, indices_q, cu_seq_lens, max_seq_lens = self._upad_input(
                query_states, key_states, value_states, attention_mask, query_length
            )

            cu_seqlens_q, cu_seqlens_k = cu_seq_lens
            max_seqlen_in_batch_q, max_seqlen_in_batch_k = max_seq_lens

            attn_output_unpad = flash_attn_varlen_func(
                query_states,
                key_states,
                value_states,
                cu_seqlens_q=cu_seqlens_q,
                cu_seqlens_k=cu_seqlens_k,
                max_seqlen_q=max_seqlen_in_batch_q,
                max_seqlen_k=max_seqlen_in_batch_k,
                dropout_p=dropout,
                softmax_scale=None,
                causal=causal,
            )

            attn_output = pad_input(attn_output_unpad, indices_q, batch_size, query_length)
        else:
            attn_output = flash_attn_func(
                query_states, key_states, value_states, dropout, softmax_scale=None, causal=causal
            )
        attn_output = attn_output.reshape(batch_size, query_length, self.hidden_size_per_partition).contiguous()
        return attn_output

    def _upad_input(self, query_layer, key_layer, value_layer, attention_mask, query_length):
        indices_k, cu_seqlens_k, max_seqlen_in_batch_k = _get_unpad_data(attention_mask)
        batch_size, kv_seq_len, num_key_value_heads, head_dim = key_layer.shape

        key_layer = index_first_axis(
            key_layer.reshape(batch_size * kv_seq_len, num_key_value_heads, head_dim), indices_k
        )
        value_layer = index_first_axis(
            value_layer.reshape(batch_size * kv_seq_len, num_key_value_heads, head_dim), indices_k
        )
        if query_length == kv_seq_len:
            query_layer = index_first_axis(
                query_layer.reshape(batch_size * kv_seq_len,
                    self.num_attention_heads_per_partition, head_dim),
                indices_k
            )
            cu_seqlens_q = cu_seqlens_k
            max_seqlen_in_batch_q = max_seqlen_in_batch_k
            indices_q = indices_k
        elif query_length == 1:
            max_seqlen_in_batch_q = 1
            cu_seqlens_q = torch.arange(
                batch_size + 1, dtype=torch.int32, device=query_layer.device
            )  # There is a memcpy here, that is very bad.
            indices_q = cu_seqlens_q[:-1]
            query_layer = query_layer.squeeze(1)
        else:
            # The -q_len: slice assumes left padding.
            attention_mask = attention_mask[:, -query_length:]
            query_layer, indices_q, cu_seqlens_q, max_seqlen_in_batch_q = unpad_input(query_layer, attention_mask)

        return (
            query_layer,
            key_layer,
            value_layer,
            indices_q,
            (cu_seqlens_q, cu_seqlens_k),
            (max_seqlen_in_batch_q, max_seqlen_in_batch_k),
        )
```

5.2 flash 加速测试

首先构造一系列使模型输出长度不同的问题进行准备：

```python
queries = [
    "你喜欢什么颜色？",
    "太阳从哪里升起？",
    "狗是如何表达情感的？",
    "你认为人工智能会对未来的工作市场产生什么影响？",
    "简述一下中国的历史朝代以及它们的特点。",
    "请介绍一下量子计算的基本原理。",
    "请详细解释一下人工智能的发展历史，包括重要的里程碑事件。",
    "描述一下地球生态系统的构成，涉及生物圈、气候变化和人类活动的影响。",
    "你如何看待全球气候变化问题？可以从政治、经济、社会和科技角度分析。",
    "请写一篇详细的文章，讨论可持续发展的多个方面，并给出具体案例。",
    "从科学、哲学、伦理和法律的角度分析人类基因编辑技术的潜力和挑战。",
    "以未来20年为时间框架，探讨人工智能如何影响全球经济、社会结构和文化变迁。"
]
```

在测试的过程中，首先对模型进行 warm-up，即在正式测试开始之前使模型推理其它问题 5 次。在测试过程中使用`torch.cuda.memory_allocated()`与`torch.cuda.max_memory_allocated()`来记录显存使用量，最后整理模型输出回答的 token 数以及 elasped time。由于问题输入的长度相比于输出长度比例较小，我们在考虑长度的影响时只考虑输出长度。测试环境为单卡 Nvidia H100。

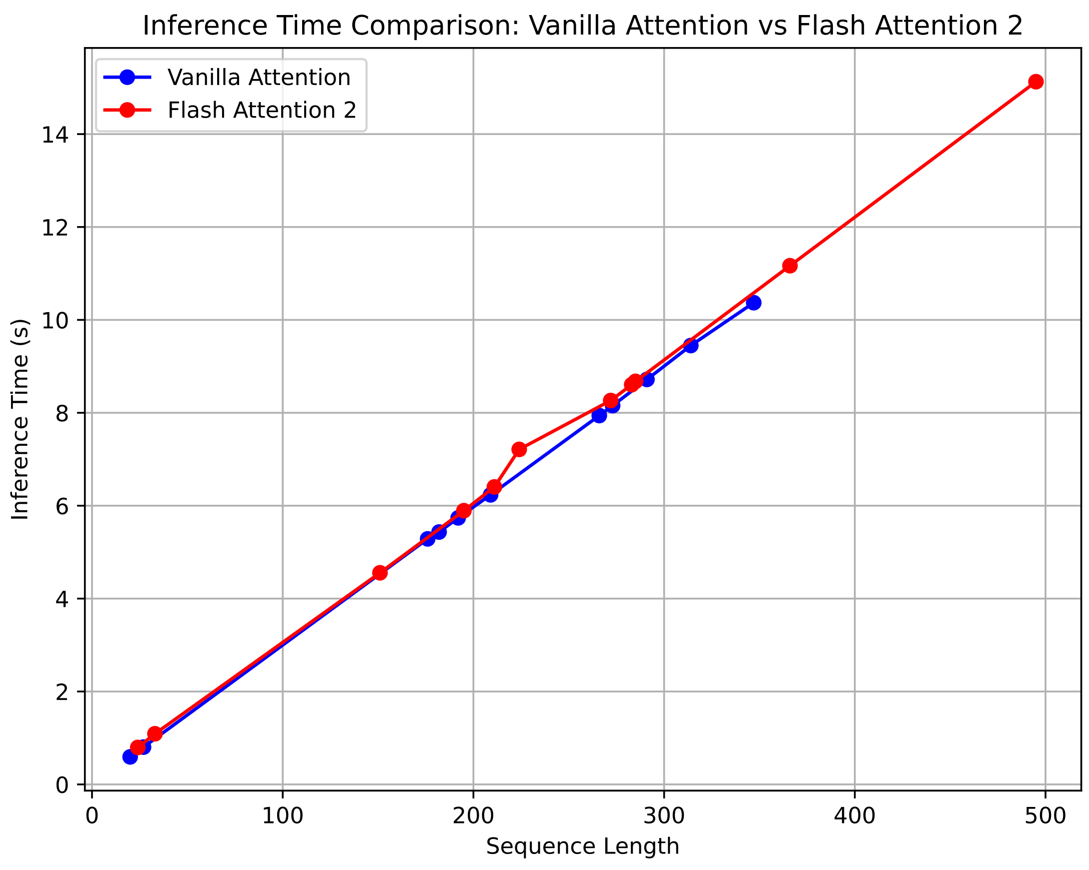


图 2: 不同 Attention 实现的推理时间

从上图可以看到根据实现的代码，Vanilla Attention 与 Flash Attention 2 在生成相同长度序列上用时接近一致。二者的显存使用均为 35938MB。