MoE 优化

Mixtral-8x7B 模型优化

该模型有8个专家，每个专家7B规模，模型总大小约为 96G, 因此用 2 卡推理较为合适。

从组网看，该模型只是在 llama 的基础上，将 mlp部分 变成了 moe部分，moe 组网也较为简单，核心代码如下：

```
class FeedForward(nn.Module):
    def __init__(
        self,
        config
    ):
        """
        Initialize the FeedForward module.

        Args:
            dim (int): Input dimension.
            hidden_dim (int): Hidden dimension of the feedforward layer.
            multiple_of (int): Value to ensure hidden dimension is a multiple of this value.
            ffn_dim_multiplier (float, optional): Custom multiplier for hidden dimension. Defaults to None.

        Attributes:
            w1 (ColumnParallelLinear): Linear transformation for the first layer.
            w2 (RowParallelLinear): Linear transformation for the second layer.
            w3 (ColumnParallelLinear): Linear transformation for the third layer.

        """
        super().__init__()

        self.w1 = nn.Linear(
            config.hidden_size, config.intermediate_size, bias=False
        )
        self.w2 = nn.Linear(
            config.intermediate_size, config.hidden_size, bias=False
        )
        self.w3 = nn.Linear(
```

```python
            config.hidden_size, config.intermediate_size, bias=False
        )

    def forward(self, x):
        return self.w2(F.silu(self.w1(x)) * self.w3(x))

class MoE(nn.Module):
    def __init__(
        self,
        config,
    ):
        super().__init__()
        self.config = config
        num_experts = config.num_experts
        self.experts = nn.ModuleList([FeedForward(config) for i in range(num_experts)])
        self.gate = nn.Linear(config.hidden_size, num_experts, bias=False)
        self.num_experts_per_token = config.num_experts_per_token

    def forward(self, x):
        orig_shape = x.shape #(batch, seq_len, d_model)
        x = x.view(-1, x.shape[-1]) #(num_tokens, d_model)

        scores = self.gate(x) #(num_tokens, num_experts)

        expert_weights, expert_indices = torch.topk(scores, self.num_experts_per_token, dim=-1) #(num_tokens, k)

        expert_weights = expert_weights.softmax(dim=-1) #(num_tokens, num_experts_per_token)
        flat_expert_indices = expert_indices.view(-1) #(num_tokens * num_experts_per_token)

        x = x.repeat_interleave(self.num_experts_per_token, dim=0) #(num_tokens * num_experts_per_token, d_model) -->

        y = torch.empty_like(x)
        for i, expert in enumerate(self.experts):
            print(f"flat_expert_indices == i =: {flat_expert_indices == i}")
            y[flat_expert_indices == i] = expert(x[flat_expert_indices == i])
        y = (y.view(*expert_weights.shape, -1) * expert_weights.unsqueeze(-1)).sum(dim=1) #(num_tokens * num_experts_per_token, d_model) --> (num_tokens, num_experts_per_token, d_model)

              # --> (num_tokens, 1, d_model)
        return y.view(*orig_shape)
```

注意这里对 gating 后的分数 scores 的处理方式：

top_k + softmax 归一化

```
expert_weights, expert_indices = torch.topk(scores, self.num_experts_per_token, 
expert_weights = expert_weights.softmax(dim=-1) #(num_tokens, num_experts_per_to
```

Vllm 推理

Vllm 较新的代码已经支持了该模型的部署，但是使用的是 EP 并行，forward 部分，采用 for 循环，性能可能较差，下面是具体的测试数据：

A100，双卡

elasped	Prompt = 1	Prompt = 1000	Prompt = 2000	Prompt = 4000
Batch = 1	39ms	42ms	44ms	48ms

注意 vllm 的实现，对 gating 后的分数（router_logits）处理方式不同

Softmax + topk + 除以sum归一化

```
routing_weights = F.softmax(router_logits, dim=1, dtype=torch.float)
routing_weights, selected_experts = torch.topk(routing_weights,
                                               self.top_k,
                                               dim=-1)
routing_weights /= routing_weights.sum(dim=-1, keepdim=True)
```

https://github.com/vllm-project/vllm/blob/main/vllm/model_executor/models/mixtral.py#L143

乍一看，代码实现上确实不同，但是简单地做下数学推导，就会发现，是一致的！

使用 group gemm 优化

该 moe 没有 EP 均衡措施，所以无法使用 BMM 替代 for 循环进行加速计算，因此可以使用 group gemm

参考 trt-llm 的 moe 实现，将 trt-llm 中的 moe kernel 抽离出来然后修改，编译为单独的算子库，方便即插即用！

未来可以继续拓展、维护，变成支持更多算子的算子库。

https://git.singularity-ai.com/shengying.wei/MoE

运行里面的单测，可以检测算子的正确性。

使用vllm 框架的最新代码，重新组网 mixtral ，利用我们的 MoE 算子库加速

如上所说，vllm 中 mixtral 使用的 ep 并行，重新组网使用 TP 并行即可:

```python
class MixtralMoE(nn.Module):
    def __init__(
        self,
        config: MixtralConfig,
        linear_method: Optional[LinearMethodBase] = None,
    ) -> None:
        super().__init__()
        self.tp_size = get_tensor_model_parallel_world_size()
        self.hidden_size = config.hidden_size
        self.inter_size = config.intermediate_size // self.tp_size
        self.top_k = config.num_experts_per_tok
        self.num_experts = config.num_local_experts
        params_dtype = torch.get_default_dtype()
        self.gate = ReplicatedLinear(config.hidden_size,
                                     self.num_experts,
                                     bias=False,
                                     linear_method=None)

        from torch.nn.parameter import Parameter
        self.moe_up_proj_weight = Parameter(torch.empty((self.num_experts, self.hidden_size, 2 * self.inter_size),
                                                        device=torch.cuda.current_device(),
                                                        dtype=params_dtype))

        self.moe_down_proj_weight = Parameter(torch.empty((self.num_experts, self.inter_size, self.hidden_size),
                                                        device=torch.cuda.current_device(),
                                                        dtype=params_dtype))

        import moe_ops
        self.run_moe_fc = moe_ops.run_moe_fc

    def forward(self, hidden_states):
        batch_size, sequence_length, hidden_dim = hidden_states.shape
        hidden_states = hidden_states.view(-1, hidden_dim) # (num_tokens, hidden_dim)
        num_rows = hidden_states.size(0)

        gating_output, _ = self.gate(hidden_states) # (num_tokens, n_experts)

        moe_output = self.run_moe_fc(hidden_states, gating_output, self.moe_up_proj_weight,
                                     "Swiglu", self.moe_down_proj_weight, num_rows, self.top_k )
        if self.tp_size == 1:
            return moe_output
        torch.distributed.all_reduce(moe_output, group=get_tensor_model_parallel_group())
        return moe_output.view(batch_size, sequence_length, hidden_dim)
```

如图，我们直接把 8 个专家合并在一起（self.moe_up_proj_weight, self.moe_down_proj_weight），然后进行切分， forward 函数中，使用我们 moe 算子库中封装好的支持torch tensor 的api，组网变得非常简洁！

最后，因为组网的改变，把 8 个专家合并在一起，所以模型切分逻辑稍微复杂一些，切分专家的主要代码如下：

```python
if ("block_sparse_moe.experts." in name): # like "model.layers.31.block_sparse_moe.experts.7.w2.weight"
    substrings = name.split('.')
    layer_num = int(substrings[2])
    expert_num = int(substrings[5])
    if("w1" in name or "w3" in name): # 列切
        loaded_weight = loaded_weight.T # (hidden_size, inter_size)
        shard_size = loaded_weight.shape[-1] // tp_size
```

```
        loaded_weight_slice = loaded_weight[:, tp_rank * shard_size : (tp_rank + 1) * shard_size] # (hidden_size, inter_size // tp_size)
        coresponed_param_name = f"model.layers.{layer_num}.block_sparse_moe.moe_up_proj_weight"
        param = params_dict[coresponed_param_name] #(num_expert, hidden_size, 2 * inter_size // tp_size)
        if ("w1" in name):
            param_slice = param[expert_num, : , shard_size : ]
            assert loaded_weight_slice.shape == param_slice.shape
            param_slice.data.copy_(loaded_weight_slice)
            continue
        else: # w3
            param_slice = param[expert_num, : ,  : shard_size]
            assert loaded_weight_slice.shape == param_slice.shape
            param_slice.data.copy_(loaded_weight_slice)
            continue
    elif ("w2" in name): # 行切
        loaded_weight = loaded_weight.T # (inter_size, hidden_size)
        shard_size = loaded_weight.shape[0] // tp_size
        loaded_weight_slice = loaded_weight[tp_rank * shard_size : (tp_rank + 1) * shard_size, :] # (inter_size // tp_size, hidden_size)
        coresponed_param_name = f"model.layers.{layer_num}.block_sparse_moe.moe_down_proj_weight"
        param = params_dict[coresponed_param_name] #(num_expert, inter_size // tp_size, hidden_size, )
        param_slice = param[expert_num]
        assert loaded_weight_slice.shape == param_slice.shape
        param_slice.data.copy_(loaded_weight_slice)
        continue
    else:
        raise ValueError(f"invalid params: {name}")
    continue
```

完整组网和切分代码见附录。

性能测试

Mixtral-8x7B，平均每个token 耗时，包含首token

Vllm 原生推理（双卡), EP 并行

elasped	Prompt = 1	Prompt = 1000	Prompt = 2000	Prompt = 4000
Batch = 1	39 ms	42 ms	44 ms	48 ms

使用 group gemm 优化后， TP 并行

elasped	Prompt = 1	Prompt = 1000	Prompt = 2000	Prompt = 4000
Batch = 1	18.64 ms	19.82 ms	21.70 ms	24.00 ms

Vllm 最新的代码支持 cuda-gprah， 开启后：

首token / 生成耗时	Prompt = 1	Prompt = 1000	Prompt = 2000	Prompt = 4000
Batch = 1	26 ms 16.64 ms	129 ms 18.23 ms	240 ms 19.3 ms	464 ms 21.79 ms

cuda-graph 约有 5% 到 10% 的性能收益。

手动配置 cutlass gemm config，避免内部自己遍历寻优

首token / 生成耗时	Prompt = 1	Prompt = 1000	Prompt = 2000	Prompt = 4000
Batch = 1	25.6 ms / 16.00 ms	106 ms 17.40 ms	192 ms 18.4 ms	372 ms 20.35 ms

生成性能提升不明显：约 4%

首token 性能有较大提升：在 prompt 长度1000 以上，有 20%+ 的提升

四卡推理：

elasped	Prompt = 1	Prompt = 1000	Prompt = 2000	Prompt = 4000
Batch = 1	11.61 ms	12.61 ms	13.45 ms	14.75 ms

精度已对齐。

```
prompt = "The capital of America is"
```

```

vllm 原生推理输出:
' a city that is full of history and culture. It is also a city that is full of things to do. If you are looking for a place to stay in Washington DC, then you should consider staying in a hotel. There are many hotels in Washington DC, and each one has its own unique features.\n\n## The Best Hotels in Washington DC\n\nThere are many hotels in Washington DC, and each one has its own unique features. Some of the best hotels in Washington DC include'

优化后输出:
' a city that is full of history and culture. It is also a city that is full of things to do. If you are looking for a place to stay in Washington DC, then you should consider staying in a hotel. There are many hotels in Washington DC, and each one has its own unique features.\n\n## The Best Hotels in Washington DC\n\nThere are many hotels in Washington DC, and each one has its own unique features. Some of the best hotels in Washington DC include'
```

部署细节

vllm: https://git.singularity-ai.com/shengying.wei/vllm-moe moe_develop_0117 分支

Moe 算子库:https://git.singularity-ai.com/shengying.wei/MoE master 分支

Docker: gpu-image-cn-shanghai.cr.volces.com/gpu-train/skyllm2:v0.1

代码细节

浅谈 moe 中 kernel 的设计思想

了解 kernel 的算法设计，才能吃透代码，感受 cuda 的艺术。

topkGatingSoftmax

该 kernel 完成对 gating 之后的分数的处理，对应的 python 逻辑如下:

```python
expert_scales = F.softmax(input_dict["gating_output"], dim=-1)
expert_scales, experts_for_row = torch.topk(expert_scales, k, dim=-1)
expert_scales /= expert_scales.sum(dim=-1, keepdim=True) #除和归一化
```

该 kernel 处理的任务规模不大，输入的 shape 为 (num_tokens, num_experts)， 在 mixtral 中，专家个数为 8。

1. **向量化读取（提升内存读取效率）：**

a. 该kernel约定，每个线程每次加载的字节个数为： BYTES_PER_LDG，那么每次可以加载的元素个数为: ELTS_PER_LDG = BYTES_PER_LDG / sizeof(T)， 因此，把 ELTS_PER_LDG 个元素看成一个向量，线程只需要一个内存加载操作即可完成该向量（ELTS_PER_LDG 个元素）的加载， ELTS_PER_LDG 也即向量的长度。

2. **算法设计（thread block的协同）**

a. 每一行都有 num_experts 个元素（专家），因此至少用一个线程束 warp 来负责一行，那么每个线程负责的元素个数为：EXPERTS / WARP_SIZE， 即 EXPERTS / (ELTS_PER_LDG * WARP_SIZE) 个向量。如果 num_experts 较少，则**每个线程只负责一个向量**，一个线程束可以处理多行。代码如下：

```
static constexpr int VECs_PER_THREAD = std::max(1, EXPERTS / (ELTS_PER_LDG * WARP_SIZE));
static constexpr int VPT = VECs_PER_THREAD * ELTS_PER_LDG; //This gives the total number of elements (or vectors, depending on how you interpret it) processed by a single thread.
```

b. 进一步便可以求出，一行需要多少个线程来处理，以及一个线程束可以处理多少行

```
static constexpr int THREADS_PER_ROW = EXPERTS / VPT; // 每一行需要 THREADS_PER_ROW 个 thread 来处理
static constexpr int ROWS_PER_WARP = WARP_SIZE / THREADS_PER_ROW; // 一个线程束 wrap 可以处理 ROWS_PER_WARP 行
```

3. **线程组织方式（grid_dim and block_dim）**

a. 上述计算出，一个线程束 warp 可以处理 ROWS_PER_WARP 行，那么对于输入的数据规模，则可以计算出一共需要多少个线程束，每次 thread block 最多有 WARPS_PER_TB 个线程束，因此一共需要的 block 数目也可随之计算出来：

```
const int num_warps = (num_rows + ROWS_PER_WARP - 1) / ROWS_PER_WARP; //一行需要 ROWS_PER_WARP 个 warp 来处理，所以一共需要 num_warps 个 wrap
const int num_blocks = (num_warps + WARPS_PER_TB - 1) / WARPS_PER_TB; //一个 thread block 中最多有 WARPS_PER_TB 个 warp， 所以需要 num_blocks 个 thread block
dim3 block_dim(WARP_SIZE, WARPS_PER_TB); //一个 thread block 中有 WARPS_PER_TB 个 warp
```

4. **线程束 warp 内部的合作方式**

a. 一个 warp 可能会负责多行，所以一个 warp 中的32个线程，也会被分割为多个sub-group，每个子部分 sub-group 处理一行。所以需要计算出，每个线程需要处理哪一行。当然，专家的个数可能是 32 的倍数，那么一个 warp 可能只处理一行，这时候，每个线程需要负责多个向量。

```
// The threads in a warp are split into sub-groups that will work on a row.
// We compute row offset for each thread sub-group
const int thread_row_in_warp = threadIdx.x / THREADS_PER_ROW;
const int thread_row = warp_base_row + thread_row_in_warp; // 当前线程需要处理的对应行
```

b. 合并内存访问

正如上面所说，假如 warp 内的每个线程需要处理 2 个向量，那么处理方式如下：

[0][1][2][3]......[31][0][1][2]......[31]

而不是：

[0][0][1][1]......[31][31]

对应代码部分：LDG_PER_THREAD 即每个线程需要处理 LDG_PER_THREAD 个向量，注意代码中的跳跃式加载。

```
#pragma unroll
for (int ii = 0; ii < LDG_PER_THREAD; ++ii) // 将全局内存中的 input 数据，加载到 row_chunk_input 来
{
    row_chunk_vec_ptr[ii] = vec_thread_read_ptr[ii * THREADS_PER_ROW];
}
```

5. **Shared memory**

使用 cutlass 中提供的静态数组，将 global memory 中的input数据，加载到 shared memory中

```
cutlass::Array<T, VPT> row_chunk_input; // 每个线程需要处理 VPT 个元素
cutlass::Array<ComputeType, VPT> row_chunk = compute_type_converter(row_chunk_input); // 把加载进来的输入数据，转换为 float
```

6. **计算精度**

a. 由于有 softmax ，需要求和以及指数运算，因此计算精度最好是 float，这需要把输入的 half 或者 bfloat16 转换为 float。

```
using ComputeType = float;
using Converter = cutlass::NumericArrayConverter<ComputeType, T, VPT>;
Converter compute_type_converter;
cutlass::Array<ComputeType, VPT> row_chunk =
compute_type_converter(row_chunk_input); // 把加载进来的输入数据，转换为 float
```

7. **其他cuda kernel常用变成技巧**
   a. 线程束内通信：__shfl_xor_sync、butterfly reduce 等。
   b. 强制精度类型转换

更多详细细节可以查看代码和相关注释。

单测代码

```python
from vllm import LLM, SamplingParams
import time
prompt = [
            "The capital of America is"
            ]
max_tokens = 100
sampling_params = SamplingParams(temperature=0.8, top_p=0.95, top_k = 1, max_tokens = max_tokens, ignore_eos=True)

llm = LLM(model="/mnt/infra/weishengying/model/Mixtral-8x7B-v0.1", max_num_batched_tokens=32768, tensor_parallel_size=2, trust_remote_code=True, enforce_eager=False)

# while True:
    # prompt = input("User:\t")
    # if prompt == 'quit':
    #     break
warm_time = 2
repeat_time = 5

for _ in range(warm_time):
    llm.generate(prompt, sampling_params)

start = time.time()
for _ in range(repeat_time):
    outputs = llm.generate(prompt, sampling_params)
end = time.time()
print(f"total elasped time: {(end-start)*1000/repeat_time}ms")
# Print the outputs.
for output in outputs:
```

```
    prompt = output.prompt
    generated_text = output.outputs[0].text
    print("generate len: ", len(output.outputs[0].token_ids))
    print(f"Generated text: {generated_text!r}")
```

Mixtral 组网和切分代码：

```
# coding=utf-8
# Adapted from
# https://github.com/huggingface/transformers/blob/v4.28.0/src/transformers/models/llama/modeling_llama.py
# Copyright 2023 The vLLM team.
# Copyright 2022 EleutherAI and the HuggingFace Inc. team. All rights reserved.
#
# This code is based on EleutherAI's GPT-NeoX library and the GPT-NeoX
# and OPT implementations in this library. It has been modified from its
# original forms to accommodate minor architectural differences compared
# to GPT-NeoX and OPT used by the Meta AI team that trained the model.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
"""Inference-only Mixtral model."""
from typing import List, Optional, Tuple

import numpy as np

import torch
import torch.nn.functional as F

from torch import nn
from transformers import MixtralConfig

from vllm.model_executor.input_metadata import InputMetadata
from vllm.model_executor.layers.attention import PagedAttention
```

```
from vllm.model_executor.layers.layernorm import RMSNorm
from vllm.model_executor.layers.linear import (LinearMethodBase,
                                               ReplicatedLinear,
                                               QKVParallelLinear,
                                               RowParallelLinear)
from vllm.model_executor.layers.rotary_embedding import get_rope
from vllm.model_executor.layers.sampler import Sampler
from vllm.model_executor.layers.vocab_parallel_embedding import (
    VocabParallelEmbedding, ParallelLMHead)
from vllm.model_executor.parallel_utils.communication_op import (
    tensor_model_parallel_all_reduce)
from vllm.model_executor.parallel_utils.parallel_state import (
    get_tensor_model_parallel_rank, get_tensor_model_parallel_world_size)
from vllm.model_executor.sampling_metadata import SamplingMetadata
from vllm.model_executor.weight_utils import (default_weight_loader,
                                              hf_model_weights_iterator)
from vllm.sequence import SamplerOutput

KVCache = Tuple[torch.Tensor, torch.Tensor]

from vllm.model_executor.parallel_utils.parallel_state import (
    get_tensor_model_parallel_world_size,
    get_tensor_model_parallel_group,
)
from vllm.utils import NcclAllReduce

class MixtralMoE(nn.Module):
    def __init__(
        self,
        config: MixtralConfig,
        linear_method: Optional[LinearMethodBase] = None,
    ) -> None:
        super().__init__()
        self.tp_size = get_tensor_model_parallel_world_size()
        self.hidden_size = config.hidden_size
        self.inter_size = config.intermediate_size
        self.top_k = config.num_experts_per_tok
        self.num_experts = config.num_local_experts
        params_dtype = torch.get_default_dtype()
        self.gate = ReplicatedLinear(config.hidden_size,
                                     self.num_experts,
                                     bias=False,
                                     linear_method=None)

        from torch.nn.parameter import Parameter
        self.moe_up_proj_weight = Parameter(torch.empty((self.num_experts,
self.hidden_size, 2 * self.inter_size // self.tp_size),
```

```
                                                    device=torch.cuda.current_device(),
                                                    dtype=params_dtype))

        self.moe_down_proj_weight = Parameter(torch.empty((self.num_experts, self.inter_size // self.tp_size, self.hidden_size),
                                                    device=torch.cuda.current_device(),
                                                    dtype=params_dtype))
        import moe_ops
        self.run_moe_fc = moe_ops.run_moe_fc

    def forward(self, hidden_states):
        batch_size, sequence_length, hidden_dim = hidden_states.shape
        hidden_states = hidden_states.view(-1, hidden_dim) # (num_tokens, hidden_dim)
        num_rows = hidden_states.size(0)

        gating_output, _ = self.gate(hidden_states) # (num_tokens, n_experts)

        moe_output = self.run_moe_fc(hidden_states, gating_output, self.moe_up_proj_weight,
                                     "Swiglu", self.moe_down_proj_weight, num_rows, self.top_k )
        if self.tp_size == 1:
            return moe_output
        # torch.distributed.all_reduce(moe_output, group=get_tensor_model_parallel_group())
        moe_output = NcclAllReduce.all_reduce(moe_output)
        return moe_output.view(batch_size, sequence_length, hidden_dim)

class MixtralAttention(nn.Module):

    def __init__(self,
                 hidden_size: int,
                 num_heads: int,
                 num_kv_heads: int,
                 max_position: int = 4096 * 32,
                 rope_theta: float = 10000,
                 linear_method: Optional[LinearMethodBase] = None,
                 sliding_window: Optional[int] = None) -> None:
        super().__init__()
        self.hidden_size = hidden_size
        tp_size = get_tensor_model_parallel_world_size()
        self.total_num_heads = num_heads
```

```python
        assert self.total_num_heads % tp_size == 0
        self.num_heads = self.total_num_heads // tp_size
        self.total_num_kv_heads = num_kv_heads
        if self.total_num_kv_heads >= tp_size:
            # Number of KV heads is greater than TP size, so we partition
            # the KV heads across multiple tensor parallel GPUs.
            assert self.total_num_kv_heads % tp_size == 0
        else:
            # Number of KV heads is less than TP size, so we replicate
            # the KV heads across multiple tensor parallel GPUs.
            assert tp_size % self.total_num_kv_heads == 0
        self.num_kv_heads = max(1, self.total_num_kv_heads // tp_size)
        self.head_dim = hidden_size // self.total_num_heads
        self.q_size = self.num_heads * self.head_dim
        self.kv_size = self.num_kv_heads * self.head_dim
        self.scaling = self.head_dim**-0.5
        self.rope_theta = rope_theta
        self.sliding_window = sliding_window

        self.qkv_proj = QKVParallelLinear(
            hidden_size,
            self.head_dim,
            self.total_num_heads,
            self.total_num_kv_heads,
            bias=False,
            linear_method=linear_method,
        )
        self.o_proj = RowParallelLinear(
            self.total_num_heads * self.head_dim,
            hidden_size,
            bias=False,
            linear_method=linear_method,
        )
        self.rotary_emb = get_rope(
            self.head_dim,
            rotary_dim=self.head_dim,
            max_position=max_position,
            base=int(self.rope_theta),
            is_neox_style=True,
        )
        self.attn = PagedAttention(
            self.num_heads,
            self.head_dim,
            self.scaling,
            num_kv_heads=self.num_kv_heads,
            sliding_window=self.sliding_window,
        )
```

```python

    def forward(
        self,
        positions: torch.Tensor,
        hidden_states: torch.Tensor,
        kv_cache: KVCache,
        input_metadata: InputMetadata,
    ) -> torch.Tensor:
        qkv, _ = self.qkv_proj(hidden_states)
        q, k, v = qkv.split([self.q_size, self.kv_size, self.kv_size], dim=-1)
        q, k = self.rotary_emb(positions, q, k)
        k_cache, v_cache = kv_cache
        attn_output = self.attn(q, k, v, k_cache, v_cache, input_metadata)
        output, _ = self.o_proj(attn_output)
        return output

class MixtralDecoderLayer(nn.Module):

    def __init__(
        self,
        config: MixtralConfig,
        linear_method: Optional[LinearMethodBase] = None,
    ) -> None:
        super().__init__()
        self.hidden_size = config.hidden_size
        # Requires transformers > 4.32.0
        rope_theta = getattr(config, "rope_theta", 10000)
        self.self_attn = MixtralAttention(
            hidden_size=self.hidden_size,
            num_heads=config.num_attention_heads,
            max_position=config.max_position_embeddings,
            num_kv_heads=config.num_key_value_heads,
            rope_theta=rope_theta,
            sliding_window=config.sliding_window,
            linear_method=linear_method)
        self.block_sparse_moe = MixtralMoE(config=config,
                                           linear_method=linear_method)
        self.input_layernorm = RMSNorm(config.hidden_size,
                                       eps=config.rms_norm_eps)
        self.post_attention_layernorm = RMSNorm(config.hidden_size,
                                                eps=config.rms_norm_eps)

    def forward(
        self,
        positions: torch.Tensor,
        hidden_states: torch.Tensor,
        kv_cache: KVCache,
```

```
        input_metadata: InputMetadata,
        residual: Optional[torch.Tensor],
    ) -> torch.Tensor:
        # Self Attention
        if residual is None:
            residual = hidden_states
            hidden_states = self.input_layernorm(hidden_states)
        else:
            hidden_states, residual = self.input_layernorm(
                hidden_states, residual)
        hidden_states = self.self_attn(
            positions=positions,
            hidden_states=hidden_states,
            kv_cache=kv_cache,
            input_metadata=input_metadata,
        )

        # Fully Connected
        hidden_states, residual = self.post_attention_layernorm(
            hidden_states, residual)
        hidden_states = self.block_sparse_moe(hidden_states)
        return hidden_states, residual

class MixtralModel(nn.Module):

    def __init__(
        self,
        config: MixtralConfig,
        linear_method: Optional[LinearMethodBase] = None,
    ) -> None:
        super().__init__()
        self.padding_idx = config.pad_token_id
        self.vocab_size = config.vocab_size

        self.embed_tokens = VocabParallelEmbedding(
            config.vocab_size,
            config.hidden_size,
        )
        self.layers = nn.ModuleList([
            MixtralDecoderLayer(config, linear_method=linear_method)
            for _ in range(config.num_hidden_layers)
        ])
        self.norm = RMSNorm(config.hidden_size, eps=config.rms_norm_eps)

    def forward(
        self,
        input_ids: torch.Tensor,
```

```
        positions: torch.Tensor,
        kv_caches: List[KVCache],
        input_metadata: InputMetadata,
    ) -> torch.Tensor:
        hidden_states = self.embed_tokens(input_ids)
        residual = None
        for i in range(len(self.layers)):
            layer = self.layers[i]
            hidden_states, residual = layer(positions, hidden_states,
                                            kv_caches[i], input_metadata,
                                            residual)
        hidden_states, _ = self.norm(hidden_states, residual)
        return hidden_states

class MixtralForCausalLM(nn.Module):

    def __init__(
        self,
        config: MixtralConfig,
        linear_method: Optional[LinearMethodBase] = None,
    ) -> None:
        super().__init__()
        self.config = config
        self.linear_method = linear_method
        self.model = MixtralModel(config, linear_method)
        self.lm_head = ParallelLMHead(config.vocab_size, config.hidden_size)
        self.sampler = Sampler(config.vocab_size)
        NcclAllReduce.init_comm()

    def forward(
        self,
        input_ids: torch.Tensor,
        positions: torch.Tensor,
        kv_caches: List[KVCache],
        input_metadata: InputMetadata,
    ) -> torch.Tensor:
        hidden_states = self.model(input_ids, positions, kv_caches,
                                   input_metadata)
        return hidden_states

    def sample(
        self,
        hidden_states: Optional[torch.Tensor],
        sampling_metadata: SamplingMetadata,
    ) -> Optional[SamplerOutput]:
        next_tokens = self.sampler(self.lm_head.weight, hidden_states,
                                   sampling_metadata)
```

```python
        return next_tokens

    def load_weights(self,
                     model_name_or_path: str,
                     cache_dir: Optional[str] = None,
                     load_format: str = "auto",
                     revision: Optional[str] = None):
        stacked_params_mapping = [
            # (param_name, shard_name, shard_id)
            ("qkv_proj", "q_proj", "q"),
            ("qkv_proj", "k_proj", "k"),
            ("qkv_proj", "v_proj", "v"),
        ]
        params_dict = dict(self.named_parameters())
        tp_size = get_tensor_model_parallel_world_size()
        tp_rank = get_tensor_model_parallel_rank()
        assert tp_size == 2
        for name, loaded_weight in hf_model_weights_iterator(
                model_name_or_path,
                cache_dir,
                load_format,
                revision,
                fall_back_to_pt=False):
            if "rotary_emb.inv_freq" in name:
                continue
            for (param_name, weight_name, shard_id) in stacked_params_mapping:
                if weight_name not in name:
                    continue
                name = name.replace(weight_name, param_name)
                # Skip loading extra bias for GPTQ models.
                if name.endswith(".bias") and name not in params_dict:
                    continue
                param = params_dict[name]
                weight_loader = param.weight_loader
                weight_loader(param, loaded_weight, shard_id)
                break
            else:
                # Skip loading extra bias for GPTQ models.
                if name.endswith(".bias") and name not in params_dict:
                    continue
                # Skip experts that are not assigned to this worker.
                if ("block_sparse_moe.experts." in name): # like "model.layers.31.block_sparse_moe.experts.7.w2.weight"
                    substrings = name.split('.')
                    layer_num = int(substrings[2])
                    expert_num = int(substrings[5])
                    if("w1" in name or "w3" in name): # 列切
```

```
                        loaded_weight = loaded_weight.T # (hidden_size, inter_size)
                        shard_size = loaded_weight.shape[-1] // tp_size
                        loaded_weight_slice = loaded_weight[:, tp_rank * shard_size : (tp_rank + 1) * shard_size] # (hidden_size, inter_size // tp_size)
                        coresponed_param_name = f"model.layers.{layer_num}.block_sparse_moe.moe_up_proj_weight"
                        param = params_dict[coresponed_param_name] # (num_expert, hidden_size, 2 * inter_size // tp_size)
                        if ("w1" in name):
                            param_slice = param[expert_num, : , shard_size : ]
                            assert loaded_weight_slice.shape == param_slice.shape
                            param_slice.data.copy_(loaded_weight_slice)
                            continue
                        else: # w3
                            param_slice = param[expert_num, : ,  : shard_size]
                            assert loaded_weight_slice.shape == param_slice.shape
                            param_slice.data.copy_(loaded_weight_slice)
                            continue
                    elif ("w2" in name): # 行切
                        loaded_weight = loaded_weight.T # (inter_size, hidden_size)
                        shard_size = loaded_weight.shape[0] // tp_size
                        loaded_weight_slice = loaded_weight[tp_rank * shard_size : (tp_rank + 1) * shard_size, :] # (inter_size // tp_size, hidden_size)
                        coresponed_param_name = f"model.layers.{layer_num}.block_sparse_moe.moe_down_proj_weight"
                        param = params_dict[coresponed_param_name]  # (num_expert, inter_size // tp_size, hidden_size, )
                        param_slice = param[expert_num]
                        assert loaded_weight_slice.shape == param_slice.shape
                        param_slice.data.copy_(loaded_weight_slice)
                        continue
                    else:
                        raise ValueError(f"invalid params: {name}")
                    continue
                else:
                    param = params_dict[name]
                    weight_loader = getattr(param, "weight_loader",
                                            default_weight_loader)
                    weight_loader(param, loaded_weight)
```