PR #24160 完整报告

sgl-project/sglang

[lora] Share MoE LoRA Info

合并时间：2026-05-29 10:01

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/24160

执行摘要

- 一句话：共享 MoE LoRA batch 信息减少重复计算
- 推荐动作：建议对 weight_indices 可能为 -1 的情况进行防御性处理（如掩码后 scatter），并增加相应测试。在非 CUDA 平台上验证 kernel 兼容性。整体设计良好，值得精读。

功能与动机

当前 MoE LoRA 设计在每一层都重复计算仅依赖 batch 的信息，造成了不必要的开销。PR body 指出 'a significant amount of information is computed every layer despite information depending solely on just the batch info.'

实现拆解

1. 新增 MoELoRABatchInfo 数据结构（python/sglang/srt/lora/utils.py）：定义集中存放 MoE LoRA 所需 per-batch 元数据的 dataclass，包括 seg_indptr、req_to_lora、adapter_enabled、token_lora_mapping。
2. 集中计算 MoE LoRA 信息（python/sglang/srt/lora/backend/base_backend.py）：新增 _add_moe_lora_info 方法，由各后端在 prepare_lora_batch 末尾调用；该方法内部调用 _compute_moe_lora_info 函数（通过 @torch.compile 或 Triton kernel）一次性生成所有元数据，并写入 batch_info.moe_lora_info。
3. Triton kernel 实现（base_backend.py）：新增 _compute_moe_lora_info_kernel，高效计算 token_lora_mapping（通过 repeat_interleave 方式）和 adapter_enabled（基于 lora_ranks 权重），并断言 kernel 覆盖全部 token。
4. 消费侧简化（python/sglang/srt/lora/layers.py 和 lora_moe_runners.py）：FusedMoEWithLoRA._get_lora_info 直接使用 batch_info.moe_lora_info 填充 LoRAInfo 结构，移除原有的逐层 adapter_enabled 计算和 _compute_token_lora_mapping 函数。
5. 后端集成：在 ascend_backend、chunked_backend、torch_backend、triton_backend 的 prepare_lora_batch 末尾添加 _add_moe_lora_info 调用，并在 FusedMoEWithLoRA 构造时设置 lora_backend.is_moe_lora = True 以启用该路径。
6. 单元测试（test/registered/lora/test_moe_lora_info.py）：测试 _compute_moe_lora_info 在不同 segment 长度、预分配缓冲区 vs 新分配、以及 kernel 覆盖不足错误抛出等场景下的正确性。

关键文件：

- python/sglang/srt/lora/backend/base_backend.py（模块 后端核心；类别 source；类型 core-logic；符号 is_moe_lora, _add_moe_lora_info, _compute_moe_lora_info_kernel, _compute_moe_lora_info）：核心文件：新增 MoE LoRA 信息计算的 Triton kernel 和统一的批量信息准备接口，是整个共享方案的核心。
- python/sglang/srt/lora/utils.py（模块 数据定义；类别 source；类型 core-logic；符号 MoELoRABatchInfo）：新增 MoELoRABatchInfo 数据类，并在 LoRABatchInfo 中增加 moe_lora_info 字段，是共享信息的容器。
- python/sglang/srt/lora/layers.py（模块 层包装；类别 source；类型 core-logic）：FusedMoEWithLoRA 中使用预计算的 MoELoRABatchInfo，简化 _get_lora_info 逻辑，移除逐层计算 adapter_enabled 的代码。
- python/sglang/srt/lora/lora_moe_runners.py（模块 MoE 运行；类别 source；类型 core-logic；符号 _compute_token_lora_mapping）：移除 _compute_token_lora_mapping 函数，直接使用预计算的 token_lora_mapping，简化 LoraHooks 逻辑。
- test/registered/lora/test_moe_lora_info.py（模块 测试；类别 test；类型 test-coverage；符号 _expected_adapter_enabled, test_compute_moe_lora_info_expands_segments, test_compute_moe_lora_info_rejects_undercovered_launch）：新增测试文件，覆盖新 kernel 在不同 segment 长度、预分配缓冲区、不足覆盖等场景下的正确性。

关键符号：is_moe_lora, _add_moe_lora_info, _compute_moe_lora_info_kernel, _compute_moe_lora_info, MoELoRABatchInfo, _expected_adapter_enabled, test_compute_moe_lora_info_expands_segments, test_compute_moe_lora_info_rejects_undercovered_launch, _compute_token_lora_mapping, _get_lora_info

关键源码片段

python/sglang/srt/lora/utils.py

新增 MoELoRABatchInfo 数据类，并在 LoRABatchInfo 中增加 moe_lora_info 字段，是共享信息的容器。

```
@dataclass
class MoELoRABatchInfo:
    # Per-request segment indptrs used by MoE LoRA routing, shape (bs + 1,).
    seg_indptr: torch.Tensor

    # Per-request adapter index used by MoE LoRA routing, shape (bs,).
    req_to_lora: torch.Tensor

    # A mask indicating if lora adapter is enabled. Shape (num_loras,)
    adapter_enabled: torch.Tensor

    # A mapping of which lora adapter is used for each token. Shape (num_tokens,)
    # If a token has no lora adapter, the value is -1.
    token_lora_mapping: torch.Tensor
```

```
@dataclass
class LoRABatchInfo:
    # ... existing fields ...
    # MoE LoRA batch info
    moe_lora_info: Optional[MoELoRABatchInfo] = None
```

test/registered/lora/test_moe_lora_info.py

新增测试文件，覆盖新 kernel 在不同 segment 长度、预分配缓冲区、不足覆盖等场景下的正确性。

```
import sys

import pytest
import torch

from sglang.srt.lora.backend.base_backend import _compute_moe_lora_info
from sglang.test.ci.ci_register import register_cuda_ci

register_cuda_ci(est_time=5, stage="base-b", runner_config="1-gpu-small")


def _expected_adapter_enabled(
    lora_ranks: torch.Tensor,
    weight_indices: torch.Tensor,
) -> torch.Tensor:
    # 使用 scatter 构建预期的 adapter_enabled 掩码
    expected = torch.zeros_like(lora_ranks)
    expected.scatter_(
        0,
        weight_indices.long(),
        (lora_ranks[weight_indices.long()] > 0).to(torch.int32),
    )
    return expected


@pytest.mark.parametrize("use_preallocated_buffers", [False, True])
def test_compute_moe_lora_info_expands_segments(use_preallocated_buffers: bool):
    # 测试 kernel 在不同 segment 长度下能否正确展开 token_lora_mapping 和 adapter_enabled
    device = "cuda"
    seg_lens = torch.tensor([5, 1, 7, 3, 9, 2], dtype=torch.int32, device=device)
    seg_indptr = torch.zeros((seg_lens.numel() + 1,), dtype=torch.int32, device=device)
    seg_indptr[1:] = torch.cumsum(seg_lens, dim=0)

    weight_indices = torch.tensor([2, 0, 5, 2, 3, 7], dtype=torch.int32, device=device)
    lora_ranks = torch.tensor(
        [0, 12, 16, 32, 24, 8, 0, 4], dtype=torch.int32, device=device
    )
    num_tokens = int(seg_indptr[-1].item())
```

```
    if use_preallocated_buffers:
        adapter_enabled = torch.full_like(lora_ranks, 123)
        token_lora_mapping = torch.full(
            (num_tokens + 11,), 456, dtype=torch.int32, device=device
        )
    else:
        adapter_enabled = None
        token_lora_mapping = None

    actual_enabled, actual_mapping = _compute_moe_lora_info(
        num_tokens,
        seg_indptr,
        lora_ranks,
        weight_indices,
        adapter_enabled,
        token_lora_mapping,
        max_len=int(seg_lens.max().item()),
    )
    torch.cuda.synchronize()

    expected_mapping = torch.repeat_interleave(weight_indices, seg_lens)
    expected_enabled = _expected_adapter_enabled(lora_ranks, weight_indices)

    torch.testing.assert_close(actual_mapping, expected_mapping)
    torch.testing.assert_close(actual_enabled, expected_enabled)

    if use_preallocated_buffers:
        # 验证使用了预分配缓冲区而非新分配
        assert actual_mapping.data_ptr() == token_lora_mapping.data_ptr()


def test_compute_moe_lora_info_rejects_undercovered_launch():
    # 测试 kernel 无法覆盖所有 token 时是否抛出 AssertionError
    device = "cuda"
    seg_indptr = torch.tensor([0, 300], dtype=torch.int32, device=device)
    weight_indices = torch.tensor([0], dtype=torch.int32, device=device)
    lora_ranks = torch.tensor([16], dtype=torch.int32, device=device)

    with pytest.raises(AssertionError, match="under-covers tokens"):
        _compute_moe_lora_info(
            300,
            seg_indptr,
            lora_ranks,
            weight_indices,
            None,
            None,
            max_len=1,
        )
```

```
if __name__ == "__main__":
    sys.exit(pytest.main([__file__]))
```

评论区精华

只有一个来自 gemini-code-assist[bot] 的 review 评论，指出 _compute_moe_lora_info 实现中如果 weight_indices 包含 -1（表示无 LoRA 的 token），则 scatter_ 会因负索引崩溃，lora_ranks[weight_indices.long()] 也会错误索引。该问题被标记为 high priority，但最终合并的代码中未显式处理 -1，可能需要调用方保证 weight_indices 不含 -1 或后续修复。

- weight_indices 为 -1 时的处理 (correctness): 未在最终代码中显式修复，但可能依赖调用方保证 weight_indices 不含 -1。

风险与影响

- 风险：
  1. 索引越界风险：如果 weight_indices 中出现 -1，_compute_moe_lora_info_kernel 中的 scatter_ 和 lora_ranks 索引会导致崩溃或错误结果。当前测试和主要调用路径中 weight_indices 均为非负值，但缺少对副作用的保护。
  2. 新 kernel 平台兼容性：Triton kernel _compute_moe_lora_info_kernel 依赖于 CUDA，在非 CUDA 平台（如 NPU）上可能无法运行。但 ascend_backend 使用了不同的实现路径（调用 _add_moe_lora_info 但实际执行的是 _compute_moe_lora_info 的纯 PyTorch 版本？从代码看 ascend_backend 也调用了 _add_moe_lora_info，而后者当前 CUDA-only，仍需验证）。
  3. 断言失败风险：kernel 内部断言要求覆盖所有 token，若 max_len 过小或 segment 长度异常可能导致推理中断。
  4. 内存开销增加：预计算的 token_lora_mapping 和 adapter_enabled 需要额外显存，与 batch size 成正比。 - 影响：影响范围仅限于使用 MoE LoRA 的模型（如 Kimi K2.5、DeepSeek 等）。对于非 MoE LoRA 的场景无影响。正面性能收益显著：单层 MoE 减少 20% 耗时，端到端 decode 提升约 12%。对开发人员的影响：需要理解 MoELoRABatchInfo 数据结构，并确保在添加新 LoRA 后端时调用 _add_moe_lora_info。测试覆盖良好。 - 风险标记：索引越界风险 (weight_indices -1)，新 kernel 平台兼容性

关联脉络

- 暂无明显关联 PR