

PR #45457 完整报告

vllm-project/vllm

[Perf] Reuse topk SparseMatrix routing metadata in GPT-OSS MoE forward

合并时间: 2026-08-30 00:14

原文链接: <http://prhub.com.cn/vllm-project/vllm/pull/45457>

执行摘要

- 一句话: 复用 topk 元数据, GPT-OSS MoE 路由提速约 43%
- 推荐动作: 值得精读。这是一个高信噪比的小改动: 核心设计决策是识别并复用库返回值中已计算好的元数据 (`SparseMatrix.mask_metadata`), 而非重复计算; 同时通过清晰的边界划分 (不碰 router/gate matmul、不碰 EP 路径、legacy 路径不动) 把风险控制到最小。值得关注的设计细节: `gate_scal` dtype 继承策略与 `n_expts_act` 的取值来源, 以及用递归比较器做完整数据结构 parity 验证的测试方法论。

功能与动机

PR 标题与 body 明确这是一次针对 GPT-OSS MoE 前向的性能优化, 服务于两条主线: 一是 issue #28986 (GPT-OSS Router 融合内核特性请求, 专家选择占单层约 3.5% 开销), 本 PR 虽未新写融合内核, 但消除了 v3.6 拆分解耦后引入的重复计算; 二是 issue #9969 (`triton_kernels` 3.6.0 相对 3.5.1 的性能回归), v3.5.1 的融合 `routing()` 本无此重复, 本 PR 等价于追平 `bundled` 版本的路径开销。PR body 特别说明设计边界: 遵循 @mgoin 在 #28986 中的指引不修改 router/gate matmul, @ijpq 的 data-hazard 担忧因此不适用。

实现拆解

1. 定位冗余计算源头: `triton_kernel_moe_forward` 的 `no-expert-map` 分支调用 `topk_fn` 后只取 `vals` 与 `indx`, 丢弃了 `SparseMatrix` 构造时已完成的 bitmatrix 打包和 `mask_metadata` (行 / 列排序索引、列和直方图); 随后 `make_routing_data` 用 `pack_bitmatrix_triton` 重新打包 bitmatrix 并构造第二个 `SparseMatrix`, 同一份元数据被计算两遍, 且发生在每个 MoE 层、每个 forward step。
2. 新增 `routing_data_from_sparse_topk`: 在 `gpt_oss_triton_kernels_moe.py` 中新增该函数, 从 `sparse_topk.mask_metadata` 直接读取 `row_sorted_indx`、`col_sorted_indx`、`col_sum`, 构建 `ragged_batch_metadata`、`RoutingData`、`GatherIndx`、`ScatterIndx`。两处关键语义与原 `make_routing_data` 不同: `n_expts_act` 取自 `sparse_topk.indx.shape[1]` 而非显式 `topk` 参数; `gate_scal` 不再强制 `bfloat16` cast, 继承 gating 输出 dtype (注释说明该路径所有调用者均为 `bfloat16`)。
3. 改造 `triton_kernel_moe_forward` 控制流: 提取 `is_sparse_topk = not isinstance(topk_result, tuple)` 布尔, 统一用于解包与路由路径选择。`no-expert-map` 分支内按 `is_sparse_topk` 分流: `SparseMatrix` 走 `routing_data_from_sparse_topk`; `tuple` 形式 (旧 API) 保持 `make_routing_data`。`expert-map` 分支保持不变 (`id` 重映射后必须重新

打包 bitmatrix) 。

4. 收敛实现: `make_routing_data` 的非 legacy 尾部从内联构造改为直接 `return routing_data_from_sparse_topk(sparse_logits, num_local_experts)`, 避免两份并行实现后续漂移, 也让新路径与老路径的语义收敛到同一函数。
5. 测试配套: 新增 `test_routing_data_from_sparse_topk_parity`, 参数化 $n_tokens \in \{1, 33, 512\} \times (n_experts, topk) \in \{(32,4), (128,4)\}$ 共 6 组配置, 用递归比较器 `assert_equivalent` 逐叶子比对 `RoutingData` 全部字段及 gather/scatter 的 src/dst 索引; legacy v3.5.1 环境自动 skip。
6. 验证配套: 作者补充微基准 (路由构造耗时 -43%、kernel 20 $\rightarrow$ 9 次)、GSM8K 精度对比 (与 main 完全一致) 和 `vllm bench serve` 端到端数据 (concurrency 1 下 TPOT -10-12%), 覆盖 H800 (SM90) 与 RTX 5090 (SM120) 两套硬件。

关键文件:

- `vllm/model_executor/layers/fused_moe/experts/gpt_oss_triton_kernels_moe.py` (模块 MOE 路由; 类别 source; 类型 data-contract; 符号 `routing_data_from_sparse_topk`, `make_routing_data`, `triton_kernel_moe_forward`): 优化的唯一承载点: 新增 `routing_data_from_sparse_topk` 复用 `SparseMatrix` 预计算元数据, 改造 `triton_kernel_moe_forward` 分流逻辑, 并将 `make_routing_data` 尾部收敛为同一函数。
- `tests/kernels/moe/test_gpt_oss_triton_kernels.py` (模块 内核测试; 类别 test; 类型 test-coverage; 符号 `test_routing_data_from_sparse_topk_parity`, `assert_equivalent`): 新增参数化 parity 测试, 用递归比较器逐叶子验证复用路径与 `make_routing_data` 在所有数据结构上严格等价, 是守护该优化的关键回归防线。

关键符号: `routing_data_from_sparse_topk`, `make_routing_data`,
`triton_kernel_moe_forward`, `test_routing_data_from_sparse_topk_parity`,
`assert_equivalent`

关键源码片段

`vllm/model_executor/layers/fused_moe/experts/gpt_oss_triton_kernels_moe.py`

优化的唯一承载点: 新增 `routing_data_from_sparse_topk` 复用 `SparseMatrix` 预计算元数据, 改造 `triton_kernel_moe_forward` 分流逻辑, 并将 `make_routing_data` 尾部收敛为同一函数。

```
# ---- 调用点: triton_kernel_moe_forward 的 no-expert-map 分支 (v3.6+) ----
# v3.6+ 的 topk 返回 SparseMatrix, 其构造时已算好 bitmatrix 与路由元数据
# (排序后的 dispatch/combine 索引、per-expert 直方图)。直接复用即可, 无需
# 像 make_routing_data 那样重新打包 bitmatrix、重建第二个 SparseMatrix。
# 该复用路径仅由 monolithic 前向 (OAITritonMx4ExpertsMonolithic) 到达;
# expert-parallel 的 modular 路径 (OAITritonExperts) 直接调用 make_routing_data,
# 不受本分支影响。
if is_sparse_topk:
    routing_data, gather_idx, scatter_idx = routing_data_from_sparse_topk(
        topk_result, gating_output.shape[-1]
    )
```

```

else:
    topk_ids = topk_ids_raw.to(torch.long)
    routing_data, gather_idx, scatter_idx = make_routing_data(
        topk_ids, topk_weights, gating_output.shape[-1]
    )

# ---- 新函数: 从 SparseMatrix 直接构建路由结构 ----
def routing_data_from_sparse_topk(
    sparse_topk, # SparseMatrix
    n_expts: int,
) -> tuple['RoutingData', 'GatherIndx', 'ScatterIndx']:
    """Build routing structures from the ``SparseMatrix`` returned by
    ``triton_kernels.topk``, reusing its precomputed bitmatrix metadata.
    """

    # mask_metadata 提供按行 / 按列排序后的 dispatch/combine 索引, 以及
    # col_sum 直方图——正是 make_routing_data 里重复计算的同一批数据
    dispatch_idx = sparse_topk.mask_metadata.row_sorted_idx
    combine_idx = sparse_topk.mask_metadata.col_sorted_idx
    ragged_batch_metadata = make_ragged_tensor_metadata(
        sparse_topk.mask_metadata.col_sum,
        dispatch_idx.shape[0],
    )

    # 激活专家数取自 indx 实际列数, 而非显式 topk 参数; 常规场景两者相等
    n_expts_act = sparse_topk.indx.shape[1]
    # 与 make_routing_data 不同: 不做 bfloat16 强制转换, gate_scal 直接
    # 继承 gating 输出 dtype (该路径所有调用者均为 bfloat16)
    gate_scal = sparse_topk.vals.flatten()[combine_idx]
    routing_data = RoutingData(
        gate_scal,
        ragged_batch_metadata.block_sizes,
        n_expts,
        n_expts_act,
        ragged_batch_metadata,
    )
    gather_idx = GatherIndx(combine_idx, dispatch_idx)
    scatter_idx = ScatterIndx(dispatch_idx, combine_idx)
    return routing_data, gather_idx, scatter_idx

```

tests/kernels/moe/test_gpt_oss_triton_kernels.py

新增参数化 parity 测试, 用递归比较器逐叶子验证复用路径与 make_routing_data 在所有数据结构上严格等价, 是守护该优化的关键回归防线。

```

def assert_equivalent(new, ref, path='RoutingData'):
    """递归比较路由结构中所有 tensor 值叶子, 任何差异都会带字段路径报出。"""
    if isinstance(ref, torch.Tensor):
        torch.testing.assert_close(new, ref, msg=lambda m: f'{path}: {m}')
    elif isinstance(ref, (int, float, bool, str)) or ref is None:
        assert new == ref, f'{path}: {new!r} != {ref!r}'

```

```

elif is_dataclass(ref):
    # 逐个字段下沉比较，路径形如 RoutingData.gate_scal，便于定位差异
    for f in fields(ref):
        assert_equivalent(
            getattr(new, f.name), getattr(ref, f.name), f'{path}.{f.name}'
        )
elif callable(ref):
    assert callable(new), f'{path}: callable vs {type(new)}'
elif hasattr(ref, '__dict__'):
    # 非 dataclass 的普通对象，遍历 __dict__ 继续下沉比较
    for k, v in vars(ref).items():
        assert_equivalent(getattr(new, k), v, f'{path}.{k}')
else:
    assert type(new) is type(ref), f'{path}: {type(new)} vs {type(ref)}'

@pytest.mark.parametrize('n_tokens', [1, 33, 512])
@pytest.mark.parametrize('n_experts,topk', [(32, 4), (128, 4)])
def test_routing_data_from_sparse_topk_parity(n_tokens, n_experts, topk):
    """routing_data_from_sparse_topk must produce routing structures
    identical to make_routing_data for the same topk result."""
    # legacy v3.5.1 (vLLM 默认捆绑版本) 没有 SparseMatrix 路径，直接跳过；
    # 该用例专门守护 v3.6+ 的复用路径与老路径在数据结构上严格等价
    if use_legacy_triton_kernels:
        pytest.skip('SparseMatrix path requires triton_kernels v3.6.0+')

    from triton_kernels.topk import topk as topk_fn

    torch.manual_seed(0)
    logits = torch.randn(n_tokens, n_experts, dtype=torch.bfloat16, device='cuda')
    sparse_topk = topk_fn(logits, topk, apply_softmax=True)
    assert not isinstance(sparse_topk, tuple)

    # 同一份 topk 结果分别喂给两条路径，递归比对所有 tensor 叶子
    rd_new, gather_new, scatter_new = routing_data_from_sparse_topk(
        sparse_topk, n_experts
    )
    rd_ref, gather_ref, scatter_ref = make_routing_data(
        sparse_topk.indx.to(torch.long), sparse_topk.vals, n_experts
    )
    assert_equivalent(rd_new, rd_ref)
    torch.testing.assert_close(gather_new.src_indx, gather_ref.src_indx)
    torch.testing.assert_close(gather_new.dst_indx, gather_ref.dst_indx)
    torch.testing.assert_close(scatter_new.src_indx, scatter_ref.src_indx)
    torch.testing.assert_close(scatter_new.dst_indx, scatter_ref.dst_indx)

```

评论区精华

yewentao256（合并者）在首轮 review 要求补充精度与端到端收益数据：要求加上 `lm_eval accuracy` 和 `vllm bench serve` 与 main 的对比。作者随后在 H800 上补齐 GSM8K（200 samples, flexible-extract/strict-match 均为 0.865/0.035, 与 main 完全一致）与 `vllm bench serve` 数据（concurrency 1 下 TPOT 降低 10-12%），并用临时 sentinel 验证 serving 路径确实进入本 PR 新增的 `elif` 分支——这一点很关键，因为 main 默认捆绑 `triton_kernels v3.5.1`，常规 CI 根本不执行新分支。

varun-sundar-rabindranath（GPT-OSS 维护者）提出 EP（expert-parallel）场景 parity 疑问：建议用 `gpt-oss-120b` 跑 `--enable-expert-parallel` 验证。作者通过代码追踪确认 `routing_data_from_sparse_topk` 仅被 `monolithic` 路径（`OAITritonMx4ExpertsMonolithic` → `triton_kernel_moe_forward`）到达，EP 下的 `modular OAITritonExperts` 直接调用 `make_routing_data`；并在 2xH800 上实测确认两种路径都会 JIT 编译 `pack_bitmatrix`，证明 EP 行为与 main 一致。作者结论：under EP, this PR behaves the same as main; the optimization is scoped to the non-EP monolithic path。reviewer 验证后给出 The PR LGTM! Thanks 并 APPROVED。

另外有若干代码风格 nit（提取 `is_sparse_topk` 布尔、参数 `n_expts_tot` 改名 `n_expts`、合并测试 imports），作者全部采纳并标注为行为不变的 refactor。

- 要求补充 `lm_eval` 精度与 `vllm bench serve` 端到端收益数据 (testing): 作者在 H800 上补充 GSM8K（200 samples）结果：flexible-extract/strict-match 均为 0.865/0.035，与 main 完全一致；端到端 `vllm bench serve` 在 concurrency 1 下 TPOT 降低 10-12%。
- EP（expert-parallel）路径是否受复用逻辑影响 (correctness): EP 行为与 main 完全一致，优化范围限定在非 EP `monolithic` 路径；reviewer 验证后给出 The PR LGTM 并 APPROVED。
- 新分支的测试覆盖盲区（CI 默认捆绑 `triton_kernels v3.5.1`）(testing): 新环境手工验证通过；parity 测试在 legacy 环境自动 skip，不阻塞 CI。SM120 上 `test_equiv` 的失败被确认为 pre-existing（MXFP 检测不识别 Blackwell），与 main 行为一致。
- 代码风格 nit: `is_sparse_topk` 布尔、参数命名与 imports 合并 (style): 作者全部采纳，提交信息标注为行为不变的 refactor，测试 11/11 通过。

风险与影响

- 风险：
 1. dtype 契约变化: `routing_data_from_sparse_topk` 中 `gate_scal` 不再强制 `bfloat16 cast`，改为继承 `gating` 输出 dtype。当前所有调用者 `gating` 输出均为 `bfloat16`，但未来若出现 `fp32 gating` 输出，`RoutingData.gate_scal` 的 dtype 会静默变化，可能影响下游 kernel 行为。
 1. 参数语义变化: `RoutingData` 构造中原本传入显式 `topk` 参数的位置，现改为 `sparse_topk.indx.shape[1]` (`n_expts_act`)。常规场景两者相等，但若未来出现 padding 或无效槽位场景（如 -1 标记），两者可能不等价，parity 测试未覆盖该类情形。
 2. 测试覆盖盲区: 新分支只在 `triton_kernels v3.6+` 环境执行，vLLM 默认捆绑 `v3.5.1`，常规 CI 不覆盖该路径；回归防护依赖新 parity 测试在升级后的环境运行。作者通过 sentinel 与 H800 手工验证弥补了这一盲区。

3. 库内部契约耦合加深：新函数直接读取 `SparseMatrix.mask_metadata` 的字段（`row_sorted_indx`、`col_sorted_indx`、`col_sum`），若 `triton_kernels` 后续调整该内部结构，会同时影响 `make_routing_data` 与新路径。
4. 性能收益边界：Eager 模式下约 43% 的路由构造耗时下降包含 launch 开销摊薄因素；CUDA graph 捕获的 decode 场景下 launch 收益减少，但重复的 kernel work（bitmatrix 打包、排序、直方图）仍每 step 执行，收益不会完全消失。- 影响：对用户：gpt-oss 系列模型在 `triton_kernels` v3.6+ 下 MoE 路由开销显著下降，decode 场景 TPOT 提升 10-12%（concurrency 1 实测），且精度无损失；EP 与 legacy 路径行为完全不变。对系统：每 MoE 层每 step 减少约 11 次 CUDA kernel launch 与一次 bitmatrix 重打包，gpt-oss-20b（24 层）整体收益可观；对依赖 `mask_metadata` 的库版本升级有正向意义。对团队：为 #47917（`triton_kernels` 3.7 兼容）铺平道路——该 PR 已包含同类复用逻辑，本 PR 落地后 #47917 可只保留 3.7 专属 API 变更；同时为 `triton_kernels` 3.6 性能回归（issue #9969）提供了 vLLM 侧的针对性修复样本。影响范围集中在单一模型路径，风险受控。- 风险标记：核心路径变更，测试覆盖盲区，依赖库内部结构，dtype 契约变化

关联脉络

- PR #35619（早期探索）GPT-OSS routing 元数据复用：PR body 明确说明本 PR 取代 #35619（作者 @banparth 的早期探索），因路由代码重构后基于当前版本重新实现，是同一想法的延续。
- PR #47917 ROCm 升级 `triton_kernels` 3.7 兼容：Rohan138 在评论中询问本 PR 是否仍活跃（ROCm 已用 `triton_kernels` 3.6 并可能迁移 3.7）；作者指出 #47917 已包含同样的 `SparseMatrix` 复用路径，本 PR 先落地可让 #47917 只保留 3.7 专属 API 变更。
- PR #39236 `triton_kernels` 3.6.0 性能回归报告：issue #9969 引用了 #39236 作为回归复现入口；本 PR 消除了 v3.6 路径相对 v3.5.1 融合 `routing()` 引入的重复计算，是 vLLM 侧对该回归的针对性修复。