PR #28658 完整报告

sgl-project/sglang

[AMD] Fuse shared-expert sigmoid + bf16->fp32 cast into the MoE append kernel (3 kernels -> 1)

合并时间：2026-07-08 17:34

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/28658

执行摘要

- 一句话：融合 sigmoid 和类型转换到 append 内核
- 推荐动作：值得阅读。该 PR 展示了如何通过内核融合消除小内核启动开销，设计思路清晰，且通过条件编译确保 CUDA 路径不受影响。对于理解 AMD 共享专家融合路径的优化方向有参考价值。

功能与动机

在 AITER 共享专家融合路径上，每次 decode 步骤每个 MoE 层需要启动三个内核：sigmoid_kernel_cuda（~5.5 us）、bfloat16tofloat32_copy_kernel_cuda（~4.5 us）和 append 内核（~4.2 us）。这些内核都是带宽 / 启动受限的小型逐元素操作，融合后可消除两次全局往返。

实现拆解

1. 修改 _fused_append_shared_experts_with_weights_kernel：增加 scale 运行时参数和 APPLY_SIGMOID 编译时常量。当 APPLY_SIGMOID=True 时，在加载共享权重后执行 tl.sigmoid(shared_ws.to(tl.float32)) * scale，直接以 fp32 格式输出。
2. 修改 fused_append_shared_experts_with_weights 函数：增加 apply_sigmoid=False, scale=1.0 参数。当 apply_sigmoid=True 时，跳过 shared_weights.to(topk_weights.dtype) 的显式类型转换，保持原始 logits 的 bf16 格式传入内核。
3. 修改 _get_shared_expert_weights 方法：返回原始 gate logits 和 1/ep_size 缩放因子组成的元组，而不是预激活的 sigmoid 输出。对于 CUDA 路径（非 AITER），仍然返回预激活权重以保持行为不变。
4. 修改 _append_shared_to_topk_output 方法：根据 _use_aiter 标志传递 apply_sigmoid=True 和缩放因子。

关键文件：

- python/sglang/srt/models/qwen2_moe.py（模块 模型路由；类别 source；类型 data-contract；符号 _get_shared_expert_weights, _append_shared_to_topk_output）：修改了 _get_shared_expert_weights 和 _append_shared_to_topk_output 方法，返回原始 logits 和缩放因子，并根据 _use_aiter 条件判断是否在调用 append 内核时启用

sigmoid 融合。

- python/sglang/srt/layers/moe/moe_runner/triton_utils/fused_moe_triton_kernels.py（模块 MoE 内核；类别 source；类型 core-logic；符号 _fused_append_shared_experts_with_weights_kernel, fused_append_shared_experts_with_weights）：修改了 Triton 内核 _fused_append_shared_experts_with_weights_kernel 和 Python 包装函数 fused_append_shared_experits_with_weights，实现了 sigmoid 和类型转换的内核内融合。

关键符号：_get_shared_expert_weights, _append_shared_to_topk_output, _fused_append_shared_experts_with_weights_kernel, fused_append_shared_experts_with_weights

关键源码片段

python/sglang/srt/models/qwen2_moe.py

修改了 _get_shared_expert_weights 和 _append_shared_to_topk_output 方法，返回原始 logits 和缩放因子，并根据 _use_aiter 条件判断是否在调用 append 内核时启用 sigmoid 融合。

```
def _get_shared_expert_weights(
    self, hidden_states: torch.Tensor
) -> Optional[Tuple[torch.Tensor, float]]:
    # 返回 (raw_logits, scale) 元组，而非预激活的 sigmoid 权重
    # 在 AITER 路径上，sigmoid 和缩放将在内核中完成
    if not self.enable_shared_expert_fusion or self.shared_expert_gate is None:
        return None
    shared_out = self.shared_expert_gate(hidden_states)
    shared_logits = shared_out[0] if isinstance(shared_out, tuple) else shared_out
    scale = 1.0
    moe_ep_size = get_parallel().moe_ep_size
    if moe_ep_size > 1 and not is_deepep_class_backend():
        scale = 1.0 / float(moe_ep_size)
    # 仅在 AITER 路径上返回原始 logits，CUDA 路径保持 eager 计算
    if not _use_aiter:
        return F.sigmoid(shared_logits) * scale, 1.0
    return shared_logits, scale

def _append_shared_to_topk_output(
    self,
    topk_output: StandardTopKOutput,
    hidden_states: torch.Tensor,
) -> StandardTopKOutput:
    if not self.enable_shared_expert_fusion:
        return topk_output
    shared = self._get_shared_expert_weights(hidden_states)
    if shared is None:
        return topk_output
    shared_weights, shared_scale = shared
    # AITER 路径传入原始 logits 并启用 sigmoid 融合
```

```
    # CUDA 路径传入已激活的权重，禁用融合
    fused_topk_ids, fused_topk_weights = fused_append_shared_experts_with_weights(
        topk_output.topk_ids,
        topk_output.topk_weights,
        shared_weights,
        self.num_fused_shared_experts,
        N=self.num_experts,
        apply_sigmoid=_use_aiter,
        scale=shared_scale,
    )
    return StandardTopKOutput(
        topk_weights=fused_topk_weights,
        topk_ids=fused_topk_ids,
        router_logits=topk_output.router_logits,
    )
```

python/sglang/srt/layers/moe/moe_runner/triton_utils/fused_moe_triton_kernels.py

修改了 Triton 内核 _fused_append_shared_experts_with_weights_kernel 和 Python 包装函数fused_append_shared_experits_with_weights，实现了sigmoid和类型转换的内核内融合。

```
@triton.jit
def _fused_append_shared_experts_with_weights_kernel(
    topk_ids_ptr,
    topk_weights_ptr,
    shared_weights_ptr,
    out_ids_ptr,
    out_weights_ptr,
    N_BASE,
    scale, # 运行时缩放因子，用于 1/ep_size 补偿
    K: tl.constexpr,
    S: tl.constexpr,
    BLOCK_K: tl.constexpr,
    BLOCK_S: tl.constexpr,
    APPLY_SIGMOID: tl.constexpr, # 编译时常量，决定是否融合 sigmoid
):
    pid = tl.program_id(0)
    ids_row_ptr = pid * K
    out_row_ptr = pid * (K + S)
    # 加载并存储 top-k 部分（不变）
    offs_k = tl.arange(0, BLOCK_K)
    mask_k = offs_k < K
    ids = tl.load(topk_ids_ptr + ids_row_ptr + offs_k, mask=mask_k)
    ws = tl.load(topk_weights_ptr + ids_row_ptr + offs_k, mask=mask_k)
    tl.store(out_ids_ptr + out_row_ptr + offs_k, ids, mask=mask_k)
    tl.store(out_weights_ptr + out_row_ptr + offs_k, ws, mask=mask_k)
    # 处理共享 expert 部分
    offs_s = tl.arange(0, BLOCK_S)
    mask_s = offs_s < S
```

```
    shared_ids = tl.cast(N_BASE + offs_s, ids.dtype)
    shared_ws = tl.load(shared_weights_ptr + pid * S + offs_s, mask=mask_s)
    if APPLY_SIGMOID:
        # 在寄存器中融合 sigmoid + 类型提升（bf16 -> fp32）+ 缩放
        # 消除单独的 sigmoid 和 bf16->fp32 复制内核
        shared_ws = tl.sigmoid(shared_ws.to(tl.float32)) * scale
    tl.store(out_ids_ptr + out_row_ptr + K + offs_s, shared_ids, mask=mask_s)
    tl.store(out_weights_ptr + out_row_ptr + K + offs_s, shared_ws, mask=mask_s)

def fused_append_shared_experts_with_weights(
    topk_ids, topk_weights, shared_weights, num_fused_shared_experts,
    N=None, apply_sigmoid=False, scale=1.0,
):
    # ... 省略形状检查和空值处理 ...
    # 当融合 sigmoid 时，保持原始 logits 的 dtype（内核直接输出 fp32）
    # 否则向后兼容，转换为 topk_weights 的 dtype
    shared_weights_2d = (
        shared_weights if apply_sigmoid else shared_weights.to(topk_weights.dtype)
    )
    # ... 维度调整和 contiguity 处理 ...
    _fused_append_shared_experts_with_weights_kernel[(m,)](
        # ... 其他参数 ...
        N_BASE=N,
        scale=scale,
        APPLY_SIGMOID=apply_sigmoid,
    )
    return out_ids, out_weights
```

评论区精华

讨论主要集中在 CI 覆盖不足上：amd-bot 评论指出 PR 的实际变更代码路径未被任何 PR CI 测试覆盖，仅由夜间测试覆盖。审阅者 HaiShaw 要求提交者提供 Qwen3.5 夜间测试结果，提交者随后附上了单元测试通过的日志。

- CI 覆盖缺失 (testing): 审核者 HaiShaw 要求提交者提供 Qwen3.5 夜间测试结果，提交者提供了单元测试通过的日志，证明变更正确。最终 PR 被合并。

风险与影响

- 风险：低风险。该变更仅在 AMD AITER 后端启用，CUDA 路径保持完全不变。内核中 sigmoid 计算使用 fp32 精度，比原来在 bf16 上的 sigmoid 更精确，不会引入数值回归。单元测试验证了融合路径与 *eager* 路径的等价性。
- 影响：对 AMD AITER 用户具有正向性能影响，每个 *decode* 步骤每个 MoE 层节省约 10.5 us（2 个内核启动开销）。由于 *decode* 步骤中这部分操作占比很小（<0.1% ITL），端到端吞吐量提升在基准测试噪声范围内。
- 风险标记：缺失测试覆盖 , 平台特定优化

关联脉络

- PR #30443 [NVIDIA] Allow modelopt_mixed quantization with flashinfer_cutedsl MoE runner: 同样涉及 MoE runner 层修改，但方向不同（量化支持 vs 内核融合）
- PR #30348 [refactor] ctx.resources: named slots, stream leases, and workspace buffer leases: 与 MoE 相关的重构，影响资源管理
- PR #30347 [refactor] Collect MoE and DP-attention runtime state into typed flag groups: MoE 状态管理重构，间接影响 MoE 内核的上下文