

PR #34330 完整报告

sgl-project/sglang

[AMD] Fix weight checking for AITER-shuffled block FP8 weights

合并时间: 2026-09-12 06:49

原文链接: <http://prhub.com.cn/sgl-project/sglang/pull/34330>

执行摘要

- 一句话: 修复 AITER 洗牌 FP8 权重检查误报, 先恢复逻辑布局再反量化
- 推荐动作: 值得精读。本 PR 展示了如何让一个平台无关的权重校验工具适配后端特定的张量布局: 通过加载期元数据标注 + 比较期布局还原, 把 " 布局知识 " 集中到量化工具层而非散落在检查器里。review 中关于 " 检查锚点应布局无关 " 的讨论与 round-trip 测试策略尤其值得借鉴; 缺点是手写 unshuffle 仍未完全摆脱对 AITER 内部约定的耦合, 可作为后续改进点 (理想方案是 AITER 提供 canonicalization API) 。

功能与动机

AITER 在 ROCm 上为了优化推理内核会对 block-FP8 权重做预洗牌 (pre-shuffle), 而权重检查器在反量化时把洗牌后的物理布局当成原始逻辑布局, 导致即使推理正确也会报出 weight-update mismatch 的假阳性。PR body 明确指出: "The weight checker was treating the shuffled physical layout as the original logical layout during dequantization, causing false weight-update mismatches even though inference was correct." 此外 AITER 派生的 rotary `cos_cache` / `sin_cache` 缓冲区不在检查器既有的派生缓冲区跳过模式里, checker 会随机化它们而权重更新又不会恢复, 同样造成假不匹配。

实现拆解

1. 标记洗牌元数据 (python/sglang/srt/layers/quantization/fp8.py): 在 `Fp8LinearMethod.process_weights_after_loading_block_quant` 和 `Fp8MoEMethod.process_weights_after_loading_block_quant` 的三处 `shuffle_weight(..., (16, 16))` 调用之后, 为权重参数追加 `is_shuffled = True` 动态属性。选择参数属性而非模块属性, 是因为洗牌是原地 copy, shape、dtype、strides 全部不变, 只有显式元数据才能让下游检查器感知布局变化;
2. 新增 unshuffle 工具与网关: `fp8_utils.py` 新增 `unshuffle_aiter_fp8_weight`, 用 `reshape(-1, n//16, k//32, 2, 16, 16).permute(0, 1, 4, 2, 3, 5).contiguous().reshape(shape)` 还原 AITER (16,16) 布局, 并校验元素字节宽与 N、K 维度约束; `fp8.py` 新增 `unshuffle_fp8_weight` 网关, 非 AITER 环境直接抛 `RuntimeError`, 防止 NVIDIA 等未洗牌路径误用。这是响应 HaiShaw review 意见、把布局逻辑从检查器下沉到量化工具层的修订结果;
3. 检查器元数据传播 (python/sglang/srt/utils/weight_checker.py): `QuantizedWeight` 增加 `is_shuffled: bool = False` 字段; `_build_quantized_set` 从 `set` 改为

dict(named_parameters()) 以通过 `getattr(parameter, "is_shuffled", False)` 读取标记；
`_build_check_entries` 把该字段透传给 `Fp8BlockComparable`；

4. 比较器恢复逻辑布局 (`python/sglang/srt/utils/weight_checker_comparator.py`) :
`Fp8BlockComparable` 增加 `is_shuffled` 状态, 在 `iter_chunks()` 与 `dequantize()` 中先调用 `unshuffle_fp8_weight` 再进入 `block_quant_dequant`, 确保校验和、ULP 比较基于逻辑布局; 同时 `_NON_PERSISTENT_BUFFER_PATTERNS` 扩展 `cos_cache`、`sin_cache` 两个模式, 避免检查器随机化 AITER 派生的 rotary 缓存;
5. 测试与 CI 配套: `test_weight_checker.py` 新增 `TestShuffledFp8Comparable` (mock 验证 `unshuffle` 先于反量化)、`test_fp8_quant_pair_preserves_shuffled_flag` (元数据传播) 与 `test_skips_cos_cache / test_skips_sin_cache` 等测试, 并注册 `register_amd_ci`; 新增 `test_fp8_utils_aiter.py`, 用 AITER 官方 `shuffle_weight` 做 round-trip 对拍替代手写预期布局测试。

关键文件:

- `python/sglang/srt/utils/weight_checker_comparator.py` (模块 权重检查; 类别 source; 类型 core-logic; 符号 `Fp8BlockComparable`, `Fp8BlockComparable.init`, `Fp8BlockComparable.iter_chunks`, `Fp8BlockComparable.dequantize`) : 核心修复点: `Fp8BlockComparable` 增加 `is_shuffled` 状态, 并在分块迭代与整权重量化前调用 `unshuffle_fp8_weight` 恢复逻辑布局, 是消除假不匹配的关键。
- `python/sglang/srt/layers/quantization/fp8_utils.py` (模块 FP8 量化; 类别 source; 类型 core-logic; 符号 `unshuffle_aiter_fp8_weight`) : 新增 `unshuffle_aiter_fp8_weight` 工具函数, 下沉布局还原逻辑到量化工具层 (响应 review 意见), 是 `unshuffle` 能力的具体实现。
- `python/sglang/srt/utils/weight_checker.py` (模块 权重检查; 类别 source; 类型 core-logic; 符号 `QuantizedWeight`, `_build_quantized_set`, `_build_check_entries`, `_NON_PERSISTENT_BUFFER_PATTERNS`) : 元数据传播链路: `QuantizedWeight` 增加 `is_shuffled` 字段, `_build_quantized_set` 从参数读取标记并透传, 同时扩展派生 `buffer` 跳过模式覆盖 AITER rotary 缓存。
- `python/sglang/srt/layers/quantization/fp8.py` (模块 FP8 量化; 类别 source; 类型 core-logic; 符号 `unshuffle_fp8_weight`, `process_weights_after_loading_block_quant`) : 元数据源头: 在 `dense` 与 `MoE` 的 AITER 洗牌路径上设置 `is_shuffled` 标记, 并提供 `unshuffle_fp8_weight` 网关 (非 AITER 抛错)。
- `test/registered/unit/utils/test_weight_checker.py` (模块 权重检查; 类别 test; 类型 test-coverage; 符号 `TestShuffledFp8Comparable`, `test_iter_chunks_unshuffles_before_dequantization`, `test_dequantize_unshuffles_before_checksum`, `test_fp8_quant_pair_preserves_shuffled_flag`) : 新增 `TestShuffledFp8Comparable` 与 `is_shuffled` 传播、`cos_cache/sin_cache` 跳过等测试, 并注册 AMD CI stage, 是本次行为变更的回归护栏。
- `test/registered/kernel/quantization/test_fp8_utils_aiter.py` (模块 FP8 量化; 类别 test; 类型 test-coverage; 符号 `TestAiterFp8Utils`, `test_unshuffle_weight_round_trip`) : 用 AITER 官方 `shuffle_weight` 做 round-trip 对拍验证 `unshuffle` 正确性, 替代手写预期布局

测试，是对 review 意见的直接响应。

关键符号：unshuffle_aiter_fp8_weight, unshuffle_fp8_weight, Fp8BlockComparable.init, Fp8BlockComparable.iter_chunks, Fp8BlockComparable.dequantize, _build_quantized_set, _build_check_entries

关键源码片段

python/sglang/srt/utils/weight_checker_comparator.py

核心修复点：Fp8BlockComparable 增加 is_shuffled 状态，并在分块迭代与整权重复化前调用 unshuffle_fp8_weight 恢复逻辑布局，是消除假不匹配的关键。

```
class Fp8BlockComparable(ComparableWeight):
```

```
    """Deepseek 风格 FP8 块量化权重的比较器。
```

```
    AITER 在 ROCm 上会把权重按 (16, 16) 布局预洗牌以适配优化后的 GEMM,
    此时 `w_q` 的物理字节序不再等于逻辑布局。`is_shuffled` 标记告诉比较器
    在反量化、校验与 ULP 比较之前先恢复逻辑布局。
    """
```

```
    def __init__(
        self,
        w_q: torch.Tensor,
        w_s: torch.Tensor,
        is_shuffled: bool = False,
    ):
        self.w_q = w_q
        self.w_s = w_s
        self.is_shuffled = is_shuffled
```

```
    def __repr__(self) -> str:
        return (
            f"fp8_block(shape={tuple(self.w_q.shape)} dtype={self.w_q.dtype} "
            f"is_shuffled={self.is_shuffled})"
        )
```

```
    def iter_chunks(self):
        s, block_size = self._scale_and_block_size()
        for q, s_chunk in self._iter_quant_chunks(self.w_q, s, block_size[0]):
            q, s_chunk = q.cuda(), s_chunk.cuda()
            if self.is_shuffled:
                # 注意：AITER 洗牌只作用于权重，scale 保持未洗牌，
                # 所以这里只需恢复权重布局，再交给 block_quant_dequant。
                q = unshuffle_fp8_weight(q)
            yield (
                block_quant_dequant(q, s_chunk, block_size, dtype=torch.bfloat16),
                block_quant_dequant(
                    self._quant_ulp(q), s_chunk, block_size, dtype=torch.float32
                ),
            ),
```

)

```
def dequantize(self, dtype: torch.dtype = torch.bfloat16) -> torch.Tensor:
    s, block_size = self._scale_and_block_size()
    w_q = self.w_q
    if self.is_shuffled:
        w_q = unshuffle_fp8_weight(w_q)
    return block_quant_dequant(w_q, s, block_size, dtype=dtype)
```

python/sglang/srt/layers/quantization/fp8_utils.py

新增 `unshuffle_aiter_fp8_weight` 工具函数，下沉布局还原逻辑到量化工具层（响应 review 意见），是 `unshuffle` 能力的具体实现。

```
def unshuffle_aiter_fp8_weight(weight: torch.Tensor) -> torch.Tensor:
    """撤销 AITER `shuffle_weight(..., layout=(16, 16))` 对 FP8 权重的洗牌。

    AITER 的 (16, 16) 布局约定：把最后两维按 N // 16、K // 32 切块后，
    每个 16x16 块内部字节以 2 个 16 元素行的交错顺序存储。这里通过
    reshape + permute + contiguous 把它还原为逻辑行主序，返回值是
    新张量，不会原地修改原权重。
    """
    if weight.element_size() != 1:
        raise ValueError("AITER FP8 unshuffle requires a one-byte element type")

    shape = weight.shape
    n, k = shape[-2:]
    if n % 16 != 0 or k % 32 != 0:
        raise ValueError(
            "AITER (16, 16) FP8 layout requires N % 16 == 0 and K % 32 == 0, "
            f"got shape {tuple(shape)}"
        )

    return (
        weight.reshape(-1, n // 16, k // 32, 2, 16, 16)
        .permute(0, 1, 4, 2, 3, 5)
        .contiguous()
        .reshape(shape)
    )
```

python/sglang/srt/layers/quantization/fp8.py

元数据源头：在 `dense` 与 `MoE` 的 AITER 洗牌路径上设置 `is_shuffled` 标记，并提供 `unshuffle_fp8_weight` 网关（非 AITER 抛错）。

```
def unshuffle_fp8_weight(weight: torch.Tensor) -> torch.Tensor:
    """恢复后端洗牌后 FP8 权重的逻辑布局。

    仅当启用 AITER 时可用；NVIDIA 等其他后端从不产生洗牌权重，
    直接调用会抛出 RuntimeError 而不是静默返回错误结果。
    """
```

```

if not _use_aiter:
    raise RuntimeError("FP8 weight unshuffle requires AITER")
return unshuffle_aiter_fp8_weight(weight)

# 洗牌是原地操作: shape、dtype、strides 均不变, 下游无从察觉,
# 因此显式记录元数据, 供权重检查器在反量化前恢复逻辑布局。
if (
    _use_aiter_bpreshuffle_gfx95
    and self.w8a8_block_fp8_linear is aiter_w8a8_block_fp8_linear
    and not getattr(layer, "skip_aiter_bpreshuffle", False)
):
    n, k = layer.weight.shape
    if not use_aiter_triton_gemm_w8a8_tuned_gfx950(n, k):
        t = shuffle_weight(layer.weight, (16, 16))
        layer.weight.copy_(t)
        del t
        layer.aiter_bpreshuffled = True
        layer.weight.is_shuffled = True

```

评论区精华

- 检查时机质疑: XinyuJiangCMU 问 " 权重到达时仍是逻辑布局, 能否在 AITER 洗牌前检查 ? " Xinyu-Kang 回应: checker 验证的是完整 weight-update 路径之后的最终模型状态, 包括后处理; 洗牌前检查会漏掉洗牌本身引入的错误。
- 布局无关锚点的设计张力: HaiShaw 提出 "The weight check anchor point should be low level aiter shuffle agnostic. Also the self constructed method of unshuffle is very aiter scheme specific, easy to break." Xinyu-Kang 修订为把 unshuffle 下沉到 fp8_utils.py 并用 round-trip 测试对拍 AITER shuffle_weight(), 最终 HaiShaw APPROVED, 但 " 检查器仍依赖手写布局还原 " 这一设计张力并未完全消除。
- cos_cache / sin_cache 必要性: yuankaichen-amd 问为什么本 PR 要加这两个模式, Xinyu-Kang 解释 Miles CI 的 [test_deepseek_v4_flash_4layer_ci.py](#) 需要: checker 会随机化派生的 rotary 缓存, 而权重更新不会恢复它们, 导致假不匹配。
- 权重检查时机: 为何在 AITER 洗牌之后而非之前检查 (question): 确认保持洗牌后检查, 因为目标是验证最终模型状态; 同时通过把 unshuffle 下沉到 fp8_utils 并由 round-trip 测试兜底来缓解布局逻辑重复问题。
- unshuffle 实现过于 AITER 特定, 检查锚点应布局无关 (design): HaiShaw APPROVED, 方案以 "unshuffle 下沉到量化工具层 + round-trip 对拍验证 " 收尾; 但手写还原对 AITER 布局约定的耦合仍在, 属于遗留设计张力。
- 为何在本 PR 引入 cos_cache / sin_cache 跳过 (question): 确认为派生缓冲区, 跳过合理; _TinyModel 同步补了对应 buffer 与测试。

风险与影响

- 风险:

- `unshuffle_aiter_fp8_weight` 是手写 `reshape + permute + contiguous`，强依赖 AITER (16,16) 布局约定；一旦 AITER 变更洗牌 `scheme`，round-trip 测试只能证明当前版本正确，未来可能静默产生错误的比较结果；
- `is_shuffled` 是挂在 `Parameter` 上的动态属性，若后续代码路径重新绑定 `Parameter`（例如 `normalize_e4m3fn_to_e4m3fnuz` 后新建 `Parameter` 的场景已有手动补标，但其他路径仍可能漏标），元数据会静默丢失导致误报回归；
- `cos_cache / sin_cache` 被硬编码进全局跳过模式，若未来某模型存在同名的真实权重参数会被误跳过；
- 对 NVIDIA 无行为影响（`is_shuffled` 默认 `False`，`unshuffle` 网关在非 AITER 下抛错），但 `_build_quantized_set` 的遍历方式从 `set` 改为 `dict`，是全平台共享代码，需留意性能可忽略的字典构造开销；
- CI 侧新增 AMD 测试注册，增加 stage-b AMD 任务的执行时间。
- 影响：
 - 用户影响：AMD ROCm + AITER 后端的权重检查器用户（尤其是 Miles DeepSeek-V4 四层 E2E CI）不再被 FP8 假不匹配干扰，检查结果 `num_exceed=0` 可信；NVIDIA 用户无行为变化；
 - 系统影响：权重加载路径新增元数据标注，权重检查路径新增 `unshuffle` 计算，但后者仅在显式开启 `checker` 时执行，正常推理性能不受影响；
 - 团队影响：AMD CI 新增 `weight checker` 与 `fp8 utils` 两处测试覆盖，为后续 AITER 后端改动提供了回归护栏。
 - 风险标记：手写 `unshuffle` 依赖 AITER 布局约定，动态属性元数据易随参数重建丢失，硬编码跳过模式可能误伤同名 `buffer`，AMD 特定路径，CUDA 回归风险低

关联脉络

- PR #34016 [Miles] Fix weight checking for AITER-shuffled block FP8 weights: 本 PR 的直接来源：PR body 明确说明 "This ports #34016 from sglang-miles to main following maintainer feedback"，布局恢复、`cos_cache/sin_cache` 跳过与测试均源自该 PR 的维护者反馈迭代。
- PR #38756 [AMD] aiter: resolve SWA KV pool for draft workers + guard paged decode: 同为 AMD AITER 后端的 bugfix，配套单测节奏一致，属于 aiter 后端在 sglang main 上持续完善的同一功能线。
- PR #34432 [AMD][DCP 1/N] add dcp support for aiter backend: 同样是面向 AITER 后端的能力扩展，说明 AMD AITER 集成是持续演进方向；本 PR 补齐该方向上的权重检查能力。