PR #26065 完整报告

sgl-project/sglang

[XPU] fix correctness issue of GDN triton kernel for XPU

合并时间：2026-05-25 13:18

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/26065

执行摘要

- 一句话：修复 XPU 上 GDN kernel 长序列的正确性
- 推荐动作：值得精读，尤其是 chunk_delta_h.py 中的循环重构策略——将时间步设为外层循环有利于维护跨时间步的状态一致性，是 Triton 中复杂 kernel 的典型优化模式。review 中关于 A dtype 的讨论也值得关注，可作为后续精度增强的切入点。

功能与动机

PR 标题和 body 明确指出：修复 XPU 上 GDN Triton kernel 在长序列长度下的正确性问题。原始代码中 K 循环在外层，导致 h 状态在时间步之间的传递被错误地按 K 维度分段更新，长序列时累积误差变大。

实现拆解

1. 重构 chunk_delta_h kernel 循环结构(python/sglang/srt/hardware_backend/xpu/kernels/fla/chunk_delta_h.py): 将原来的 K- 外循环 + 时间步 - 内循环改为时间步 - 外循环，在每个时间步内分两阶段处理所有 K 块：Phase 1 将 pre-update h 写出到输出并累积 v 修正项，Phase 2 读取前一步的 h 并应用 gate 和 k^T @ v 更新，写回 scratch。这保证了时间步之间 h 的一致性。
2. 清理 chunk_fwd kernel scratch 残留(python/sglang/srt/hardware_backend/xpu/kernels/fla/chunk_fwd.py): 在 chunk_gated_delta_rule_fwd_kkt_solve_kernel_low_reg 的 epilogue 中添加了一个 tl.static_range 循环，将上三角区（Pass 2 写入的临时 A_ij 块）清零，避免后续 recompute_w_u_fwd 读到脏数据。
3. 新增长提示回归测试(test/registered/attention/test_chunk_gated_delta_rule.py): 添加 test_long_prompt 方法，使用 B=1,2 和 T_per_seq=1024/1536/2048 的组合，覆盖跨多个 chunk 的场景，确保 cross-chunk 边界正确。测试通过 _check_shape 与参考实现对比精度。
4. 精度验证(PR body 中提供): 在 Intel B60 上使用 Qwen3.5-4B、9B、35B-A3B 在 GSM8K 上验证，准确率与 SOTA 相当。

关键文件：

- python/sglang/srt/hardware_backend/xpu/kernels/fla/chunk_delta_h.py（模块 Attention；类别 source；类型 core-logic；符号 chunk_gated_delta_rule_fwd_kernel_h_blockdim64_k_loop, chunk_gated_delta_rule_fwd_h）：核心 kernel 的循环重构，修复长

序列正确性问题的主要改动所在

- python/sglang/srt/hardware_backend/xpu/kernels/fla/chunk_fwd.py（模块 Attention；类别 source；类型 core-logic；符号 chunk_gated_delta_rule_fwd_kkt_solve_kernel_low_reg, chunk_gated_delta_rule_fwd_intra）：修复 KKT solve kernel 的 scratch 残留，避免长序列下读取未初始化数据
- test/registered/attention/test_chunk_gated_delta_rule.py（模块 测试；类别 test；类型 test-coverage；符号 test_long_prompt）：新增长提示回归测试，覆盖多 chunk 场景，验证跨 chunk 正确性

关键符号：chunk_gated_delta_rule_fwd_kernel_h_blockdim64_k_loop, chunk_gated_delta_rule_fwd_h, chunk_gated_delta_rule_fwd_kkt_solve_kernel_low_reg, chunk_gated_delta_rule_fwd_intra, test_long_prompt

关键源码片段

python/sglang/srt/hardware_backend/xpu/kernels/fla/chunk_delta_h.py

核心 kernel 的循环重构，修复长序列正确性问题的主要改动所在

```
# 此 kernel 在 for 循环中处理 K 块以减少寄存器溢出。
# 时间步为外层循环；每个时间步内分两阶段处理 K 块：
# 阶段 1：将 h 写出到输出，累积 v_correction = sum_k(w_k @ h_k^T)
# 阶段 2：更新 h = gate * h + k^T @ v_gated，保存到 scratch (initial_state)
@triton.jit
def chunk_gated_delta_rule_fwd_kernel_h_blockdim64_k_loop(...,):
    # ... 前处理：初始化指针，加载索引
    index = tl.load(initial_state_indices + i_n).to(tl.int32)
    h0 = initial_state + index * stride_h
    ht = initial_state + index * stride_h
    if USE_INITIAL_STATE:
        h0 = h0 + i_h * V * K
    if INPLACE_UPDATE:
        ht = ht + i_h * V * K

    # 主要循环：时间步为外层循环
    for i_t in range(NT):

        ################################################################################
        #
        # Phase 1: store h to output, compute v_new = u - sum_k(w_k @ h_k^T)

        ################################################################################
        #
        b_v_corr = tl.zeros([BT, BV], dtype=tl.float32)
        for k_blk in range(0, K, 64):
            # 加载 h：从 initial_state (i_t==0) 或 scratch (i_t>0)
            if i_t == 0:
                if USE_INITIAL_STATE:
                    p_hs = tl.make_block_ptr(
```

```
                h0, (V, K), (K, 1), (i_v * BV, k_blk), (BV, 64), (1, 0)
            )
            b_h = tl.load(p_hs, boundary_check=(0, 1)).to(tl.float32)
        else:
            b_h = tl.zeros([BV, 64], dtype=tl.float32)
    else:
        p_hs = tl.make_block_ptr(
            ht, (V, K), (K, 1), (i_v * BV, k_blk), (BV, 64), (1, 0)
        )
        b_h = tl.load(p_hs, boundary_check=(0, 1)).to(tl.float32)

    # 将 pre-update h 写出到输出
    p_ho = tl.make_block_ptr(
        h + i_t * stride_h, (V, K), (K, 1), (i_v * BV, k_blk), (BV, 64), (1, 0)
    )
    tl.store(p_ho, b_h.to(p_ho.dtype.element_ty), boundary_check=(0, 1))

    # 累积修正项：w_k @ h_k^T
    b_w = w_desc.load([i_t * BT, k_blk])
    b_v_corr += tl.dot(b_w, tl.trans(b_h).to(b_w.dtype))

# v_new = u - 修正项
b_v = v_desc.load([i_t * BT, i_v * BV]) - b_v_corr

if SAVE_NEW_VALUE:
    v_new_desc.store([i_t * BT, i_v * BV], b_v.to(v_new.dtype.element_ty))

# 对 v 应用门控
last_idx = min((i_t + 1) * BT, T) - 1
if USE_G:
    b_g_last = tl.load(g + bos * H + last_idx * H + i_h)
    p_g = tl.make_block_ptr(
        g + bos * H + i_h, (T,), (H,), (i_t * BT,), (BT,), (0,)
    )
    b_g = tl.load(p_g, boundary_check=(0,))
    b_v = b_v * tl.expand_dims(safe_exp(b_g_last - b_g), 1)
    b_g_last = exp(b_g_last)

b_v = b_v.to(k.dtype.element_ty)


# ###########################################################################
# Phase 2: reload h, apply gate, update h += k^T @ v, save to scratch
# ###########################################################################
for k_blk in range(0, K, 64):
    # 加载 pre-update h（从 h0 或 ht 中，同 Phase 1）
```

```
    b_h = tl.load(...) # 同 Phase 1 的加载逻辑
    # 应用门控
    if USE_G:
        b_h = b_h * b_g_last
    if USE_GK:
        # 应用 key 门控
        # ...
    # 更新：b_h += k^T @ v
    b_k = tl.trans(k_desc.load([i_t * BT, k_blk]))
    b_h += tl.trans(tl.dot(b_k, b_v))
    # 将更新后的 b_h 写回 scratch（ht）
    p_hs_new = tl.make_block_ptr(
        ht, (V, K), (K, 1), (i_v * BV, k_blk), (BV, 64), (1, 0)
    )
    tl.store(p_hs_new, b_h.to(p_hs_new.dtype.element_ty), boundary_check=(0, 1))
```

python/sglang/srt/hardware_backend/xpu/kernels/fla/chunk_fwd.py

修复 KKT solve kernel 的 scratch 残留，避免长序列下读取未初始化数据

```
# 在 chunk_gated_delta_rule_fwd_kkt_solve_kernel_low_reg 函数末尾添加：
    # 清理 scratch 插槽：Pass 2 将原始 A_ij 块存储在第 i_tc0 行的上三角部分（列 BC..3*BC）。
    # 必须将这些区域清零，因为 recompute_w_u_fwd 会读取整个 BTxBT 块。
    b_zero = tl.zeros([BC, BC], dtype=tl.float32)
    for sc in tl.static_range(1, BT // BC):
        p_scratch = tl.make_block_ptr(
            A, (T, BT), (H * BT, 1), (i_tc0, sc * BC), (BC, BC), (1, 0)
        )
        tl.store(p_scratch, b_zero.to(A.dtype.element_ty), boundary_check=(0, 1))
```

评论区精华

- reviewer [gemini-code-assist] 关于 A 的 dtype：建议将 chunk_fwd_intra 中的中间张量 A 初始化为 float32 以避免低精度 dtype 下的精度损失，并取消注释 A.to(k.dtype) 以保证 tl.dot 匹配。作者回复“Fixed”，但最终代码并未修改 A 的 dtype（仍使用 k.dtype），相关注释被保留但该转换代码仍被注释。这是一个未解决的潜在精度隐患。
- reviewer 关于 assert False 的用法：指出 Triton kernel 内不应使用 assert False，应改用 tl.static_assert 或删除不可达分支。作者回复“Fixed”，最终代码中该 assert 被移除。
- reviewer [mingfeima] 关于测试参数化：建议使用 pytest.mark.parametrize 替代手动 for 循环。作者选择用 for 循环简化，但未采用参数化。该讨论已解决，测试结构保持简单。
  - chunk_fwd_intra 中 A 的 dtype 应使用 float32 避免精度损失 (correctness): 作者回复 Fixed，但最终代码中 A 的 dtype 仍为 k.dtype，转换代码保持注释。未实际修复。
  - Triton kernel 中不应使用 assert False (correctness): 作者回复 Fixed，最终代码中 assert 被移除。
  - 测试用例应使用 pytest.mark.parametrize 简化 (testing): 作者采用 for 循环 + subTest 实现，未使用 parametrize，但功能等价。讨论结束。

风险与影响

- 风险：
  - 精度风险（残留）：chunk_fwd.py 中 A 的 dtype 仍沿用 k.dtype（如 bfloat16），reviewer 指出的精度损失在长序列下可能复发。虽当前无实测异常，但对低精度 dtype 是潜在风险。
  - 回归风险：循环重构改变了 kernel 内读写模式，可能影响其他硬件后端（如 CUDA）。但此 kernel 仅用于 XPU，且测试覆盖了多种形状，回归概率低。
  - 性能影响：时间步 - 外循环可能略微增加寄存器压力，但原 PR 未提供性能数据。由于修复正确性是首要目标，性能可后续优化。
  - 测试覆盖：新增的 test_long_prompt 仅覆盖 B=1,2 和 T 为 1024~2048，未覆盖更大的 batch 或与 gate/initial_state 组合的复杂场景。
  - 影响：用户：Intel XPU 上使用 GDN attention 的模型（如 Qwen3.5 系列）在长序列推理时不再产生错误结果，精度显著提升。系统：仅修改 XPU 后端 kernel，对 CUDA 等其他后端无影响。团队：为 XPU 平台上的 Triton kernel 开发提供了时序循环结构的参考模式，有助于后续维护。
- 风险标记：精度 dtype 残留风险，仅覆盖有限长序列测试，无性能数据对比

关联脉络

- 暂无明显关联 PR