PR #31834 完整报告

sgl-project/sglang

Support a same-size mixed q dtype in the fused RoPE kernels

合并时间：2026-07-25 07:19

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/31834

执行摘要

- 一句话：fused RoPE 核支持混合 q 数据类型
- 推荐动作：值得精读，特别是 CUDA 核的模板重构和 radix_attention 输出 dtype 调整，可以作为支持不同数据类型混用的范式。

功能与动机

某些 fused QK-norm 实现产生与 k 不同 dtype 的 q（如 fp16 q 与 bf16 k），但 fused RoPE 核之前假设单一 dtype，导致混合 dtype 下注意力后端输出 dtype 错误。本 PR 解决此问题，使 RoPE 核支持混合 dtype。

实现拆解

1. CUDA 核层：将旋转逻辑抽取为 rope_rotate_head 模板函数，引入 T 模板参数，使得 q 和 k 分支可以使用不同的 T（大小相同），并通过 static_assert 约束。k 分支中缓存写入路径通过 cache_out 参数谓词化，避免代码重复。
2. JIT 模块层：_jit_fused_rope_module 函数增加 q_dtype 参数，加入缓存键和 make_cpp_args 调用，确保编译正确的特化版本。
3. 运行时层：RadixAttention.forward 中 out_dtype 改为使用 v.dtype（若 v 非空），并保留 fp8→bf16 回退逻辑。这样混合 dtype 下输出缓冲区大小匹配模型 dtype，不会因 q.dtype 不同而改变后端输出。
4. 测试层：新增 test_rope_mixed_q_dtype（in-place）和 test_rope_store_mixed_q_dtype（fused KV store）函数，通过逐位比较混合 dtype 运行结果与对应单 dtype 运行结果，验证正确性。

关键文件：

- test/registered/kernels/ops/attention/test_rope.py（模块 RoPE 测试；类别 test；类型 test-coverage；符号 test_rope_mixed_q_dtype, test_rope_store_mixed_q_dtype, run）：新增测试覆盖混合 dtype 的 RoPE 逐位等价性，确保核修改的正确性
- python/sglang/srt/layers/radix_attention.py（模块 分页注意力；类别 source；类型 core-logic）：核心运行时调整：输出缓冲区 dtype 改为沿用 v.dtype 而非 q.dtype，避免混合 dtype 下注意力后端输出错误
- python/sglang/kernels/jit/csrc/elementwise/rope.cuh（模块 CUDA 核；类别 other；类型 core-logic）：核心 CUDA 核重构：将旋转逻辑抽取为 rope_rotate_head 模板函数，引

入 T 模板参数区分 q/k 数据类型

- python/sglang/kernels/ops/attention/rope.py（模块 JIT 管理；类别 infra；类型 infrastructure；符号 _jit_fused_rope_module）：JIT 模块管理调整：缓存键和调用参数增加 q_dtype，以支持不同 q/k dtype

关键符号：rope_rotate_head, apply_rope_inplace, apply_rope_inplace_with_kvcache, test_rope_mixed_q_dtype, test_rope_store_mixed_q_dtype, RadixAttention.forward, _jit_fused_rope_module

关键源码片段

test/registered/kernels/ops/attention/test_rope.py

新增测试覆盖混合 dtype 的 RoPE 逐位等价性，确保核修改的正确性

```
# test_rope_mixed_q_dtype 和 test_rope_store_mixed_q_dtype
# 验证混合 dtype 下 RoPE 核输出与纯 dtype 核逐位一致

@pytest.mark.parametrize("is_neox", IS_NEOX_LIST)
@pytest.mark.parametrize("rope_dim", get_ci_test_range([64, 128], [64]))
def test_rope_mixed_q_dtype(is_neox: bool, rope_dim: int) -> None:
    """fp16 q + bf16 k (a fused QK-norm may emit q in a different dtype):
    q must match the all-fp16 kernel bitwise, k the all-bf16 kernel bitwise."""
    batch_size, num_kv_heads, gqa_ratio = 129, 4, 8
    num_qo_heads = num_kv_heads * gqa_ratio
    # 创建不同 dtype 的 q/k：q fp16, k bf16
    q16 = torch.randn(batch_size, num_qo_heads, rope_dim, device=DEVICE, dtype=torch.
    float16)
    kbf = torch.randn(batch_size, num_kv_heads, rope_dim, device=DEVICE, dtype=torch.
    bfloat16)
    positions = torch.randint(0, MAX_SEQ_LEN, (batch_size,), device=DEVICE, dtype=torch.int64)
    cos_sin_cache = create_cos_sin_cache(rope_dim)

    # 混合 dtype 的 RoPE 前向
    q_mixed, k_mixed = q16.clone(), kbf.clone()
    sglang_jit_rope(q_mixed, k_mixed, cos_sin_cache, positions, is_neox)

    # 参考：全 fp16 核处理 q 和 k_f16
    q_ref, k_f16 = q16.clone(), kbf.to(torch.float16)
    sglang_jit_rope(q_ref, k_f16, cos_sin_cache, positions, is_neox)
    # 参考：全 bf16 核处理 q_bf16 和 k
    q_bf16, k_ref = q16.to(torch.bfloat16), kbf.clone()
    sglang_jit_rope(q_bf16, k_ref, cos_sin_cache, positions, is_neox)

    # 逐位比较：混合 q 应与全 fp16 结果一致，混合 k 应与全 bf16 结果一致
    assert torch.equal(q_mixed, q_ref)
    assert torch.equal(k_mixed, k_ref)

@pytest.mark.parametrize("is_neox", IS_NEOX_LIST)
def test_rope_store_mixed_q_dtype(is_neox: bool) -> None:
```

```
    """Fused RoPE + KV store with fp16 q + bf16 k: q bitwise vs the all-fp16
    kernel; k, k_cache, v_cache bitwise vs the all-bf16 kernel."""
    from sglang.kernels.ops.attention.rope import apply_rope_inplace_with_kvcache

    batch_size, num_kv_heads, gqa_ratio, rope_dim = 129, 4, 8, 64
    num_qo_heads = num_kv_heads * gqa_ratio
    row_size = num_kv_heads * rope_dim
    q16 = torch.randn(batch_size, num_qo_heads, rope_dim, device=DEVICE, dtype=torch.
    float16)
    kbf = torch.randn(batch_size, num_kv_heads, rope_dim, device=DEVICE, dtype=torch.
    bfloat16)
    vbf = torch.randn(batch_size, num_kv_heads, rope_dim, device=DEVICE, dtype=torch.
    bfloat16)
    positions = torch.randint(0, MAX_SEQ_LEN, (batch_size,), device=DEVICE, dtype=torch.int64)
    out_loc = torch.randperm(CACHE_SIZE, device=DEVICE, dtype=torch.int64)[:batch_size]
    cos_sin_cache = create_cos_sin_cache(rope_dim)

    def run(q, k, v):
        # 创建与 k 同 dtype 的 KV 缓存
        k_cache = torch.zeros(CACHE_SIZE, row_size, device=DEVICE, dtype=k.dtype)
        v_cache = torch.zeros(CACHE_SIZE, row_size, device=DEVICE, dtype=k.dtype)
        apply_rope_inplace_with_kvcache(q, k, v, k_cache, v_cache, cos_sin_cache, positions, out_
        loc, is_neox=is_neox)
        return k_cache, v_cache

    # 混合 dtype 路径
    q_mixed = q16.clone()
    k_mixed, v_mixed = kbf.clone(), vbf.clone()
    kc_mixed, vc_mixed = run(q_mixed, k_mixed, v_mixed)

    # 参考：全 fp16 路径（q 和 cache）
    q_ref = q16.clone()
    kc16, _ = run(q_ref, kbf.to(torch.float16), vbf.to(torch.float16))
    # 参考：全 bf16 路径（q 和 cache）
    q_bf16 = q16.to(torch.bfloat16)
    kc_ref, vc_ref = run(q_bf16, kbf.clone(), vbf.clone())

    # 逐位比较
    assert torch.equal(q_mixed, q_ref) # q 应与 fp16 一致
    assert torch.equal(kc_mixed, kc_ref) # k_cache 应与 bf16 一致
    assert torch.equal(vc_mixed, vc_ref) # v_cache 应与 bf16 一致
```

python/sglang/srt/layers/radix_attention.py

核心运行时调整：输出缓冲区 dtype 改为沿用 v.dtype 而非 q.dtype，避免混合 dtype 下注意力后端输出错误

```
# radix_attention.py 中 out_dtype 计算逻辑（修改后）
# 输出 dtype 沿用 v.dtype（模型 dtype），当 v 非空时；否则回退到 q.dtype
# 同时保留 fp8 → bf16 回退逻辑，避免 fp8 输出缓冲区被 cast-copy 截断
```

```
out_dtype = v.dtype if v is not None else q.dtype
if out_dtype in (torch.float8_e4m3fn, torch.float8_e5m2):
    out_dtype = torch.bfloat16

if self.qk_head_dim != self.v_head_dim:
    output = q.new_empty(
        (q.shape[0], self.tp_q_head_num * self.v_head_dim),
        dtype=out_dtype,
    )
else:
    output = torch.empty_like(q, dtype=out_dtype)
```

python/sglang/kernels/jit/csrc/elementwise/rope.cuh

核心 CUDA 核重构：将旋转逻辑抽取为 rope_rotate_head 模板函数，引入 T 模板参数区分 q/k 数据类型

```
// rope_rotate_head 模板函数：旋转单头行，支持不同 T 类型
// kIsNeox: 旋转风格；kRopeDim: 旋转维度；kVecSize: 向量化宽度
// 当 cache_out 非空时，同时写入缓存（用于 fused KV store 的 k 路径）
template <typename T, bool kIsNeox, int64_t kRopeDim, int64_t kVecSize>
__device__ __forceinline__ void
rope_rotate_head(void* input, const void* cos_ptr, const void* sin_ptr,
                 uint32_t lane_id, void* cache_out = nullptr) {
  using namespace device;
  using T2 = packed_t<T>;
  using Storage = AlignedVector<T2, kVecSize>;
  if constexpr (kIsNeox) {
    // Neox 风格：交替半拆分，分别加载输入向量的前半和后半
    using CacheStorage = AlignedVector<fp32x2_t, kVecSize>;
    const auto input_x = input;
    const auto input_y = pointer::offset(input, (kRopeDim / 2) * sizeof(T));
    auto input_vec_x = load_as<Storage>(input_x, lane_id);
    auto input_vec_y = load_as<Storage>(input_y, lane_id);
    const auto cos_pair = load_as<CacheStorage>(cos_ptr, lane_id);
    const auto sin_pair = load_as<CacheStorage>(sin_ptr, lane_id);
#pragma unroll
    for (int64_t j = 0; j < kVecSize; ++j) {
      // 使用 cos/sin 执行旋转
      const auto [x0, x1] = cast<fp32x2_t>(input_vec_x[j]);
      const auto [y0, y1] = cast<fp32x2_t>(input_vec_y[j]);
      const auto [cos_0, cos_1] = cos_pair[j];
      const auto [sin_0, sin_1] = sin_pair[j];
      const auto out_x0 = x0 * cos_0 - y0 * sin_0;
      const auto out_y0 = x0 * sin_0 + y0 * cos_0;
      const auto out_x1 = x1 * cos_1 - y1 * sin_1;
      const auto out_y1 = x1 * sin_1 + y1 * cos_1;
      input_vec_x[j] = cast<T2, fp32x2_t>({out_x0, out_x1});
      input_vec_y[j] = cast<T2, fp32x2_t>({out_y0, out_y1});
    }
```

```
      store_as<Storage>(input_x, input_vec_x, lane_id);
      store_as<Storage>(input_y, input_vec_y, lane_id);
      if (cache_out != nullptr) {
        store_as<Storage>(cache_out, input_vec_x, lane_id);
        const auto cache_out_y = pointer::offset(cache_out, (kRopeDim / 2) * sizeof(T));
        store_as<Storage>(cache_out_y, input_vec_y, lane_id);
      }
    } else {
      // Default 风格：直接对每个元素应用旋转
      using CacheStorage = AlignedVector<float, kVecSize>;
      auto input_vec = load_as<Storage>(input, lane_id);
      const auto cos_vec = load_as<CacheStorage>(cos_ptr, lane_id);
      const auto sin_vec = load_as<CacheStorage>(sin_ptr, lane_id);
#pragma unroll
      for (int64_t j = 0; j < kVecSize; ++j) {
        const auto [x, y] = cast<fp32x2_t>(input_vec[j]);
        const auto cos = cos_vec[j];
        const auto sin = sin_vec[j];
        const auto out_x = x * cos - y * sin;
        const auto out_y = x * sin + y * cos;
        input_vec[j] = cast<T2, fp32x2_t>({out_x, out_y});
      }
      store_as<Storage>(input, input_vec, lane_id);
      if (cache_out != nullptr) {
        store_as<Storage>(cache_out, input_vec, lane_id);
      }
    }
  }
```

评论区精华

review 讨论较少，主要在 CI 评论中作者进行 triage，确认所有相关测试通过，其他 CI 失败为无关扩散模型问题。

- 暂无高价值评论线程

风险与影响

- 风险：核层面通过 static_assert 限制相同大小类型，避免不兼容 dtype。radix_attention 输出缓冲区根据 v.dtype 分配，若 v 为 None 则回退到 q.dtype，安全性好。测试覆盖了混合 dtype 的 in-place 和 KV 存储路径，确保核修改正确性。
- 影响：对用户：支持 fused RoPE 与不同 dtype q/k 的兼容性，无 breaking change。对系统：降低 RoPE 后处理的数据类型约束，使得 QK-norm 实现选择更灵活。对团队：维护了与不同 dtype RoPE 的兼容性。
- 风险标记：相同大小类型限制，v.dtype 依赖，核重构，测试覆盖

关联脉络

- 暂无明显关联 PR