PR #28534 完整报告

sgl-project/sglang

[AMD] Enable JIT staged HiCache write-back and fix CPU-index crash

合并时间：2026-07-09 16:22

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/28534

执行摘要

- 一句话：启用 ROCm HiCache JIT 分阶段写回并修复索引崩溃
- 推荐动作：该 PR 值得精读，尤其是 JIT kernel 平台适配和向量化技巧。对于维护多后端的团队，此 PR 展示了如何优雅处理 PTX 平台差异。建议关注后续是否回退 #28473，以及是否需要在更多后端（如 NPU）上推行类似模式。

功能与动机

ROCm 上 page_first + kernel HiCache 写回在第一次预填充时崩溃：RuntimeError: Destination indices must be a CUDA tensor。根因是 HiCacheController.start_writing() 将 host_indices 保留在 CPU，但非 JIT kernel 需要设备索引。之前的 workaround (#28473) 强制使用 layer_first 布局，但关闭了 JIT 分阶段写回，导致 ROCm 性能降级并偏离 CUDA 路径。此 PR 从根源修复，使 ROCm 使用与 CUDA 相同的 JIT 写回 kernel，保留带宽和向量化非临时 load/store，同时消除维护两套路径的成本。

实现拆解

1. 扩展 can_use_jit 到 HIP：在 memory_pool_host.py 和 pool_host/mha.py 中，将 can_use_jit 和 can_use_write_back_jit 的条件从 _is_cuda 扩展为 _is_cuda or _is_hip，并添加详细注释，使得 ROCm 也能构建和执行 JIT HiCache kernel。
2. ROCm 非临时 load/store 适配：在 hicache.cuh 中，通过 #ifdef USE_ROCM 保护 PTX 指令，为 load_nc/store_nc 实现基于 __builtin_nontemporal_load/store 的 ROCm 版本。使用 Clang ext_vector_type 和 __builtin_bit_cast 确保向量化指令（global_{load,store}_dwordx{2,4}），避免编译器退化为标量操作导致带宽下降。
3. 设备匹配器统一：在 hicache.cuh 和 staged_write_back.cuh 中，将 TensorMatcher 的 .with_device 从硬编码的 kDLCUDA/kDLCUDAHost 替换为通过条件宏定义的 kDLGPU/kDLGPUHost，兼容 CUDA 和 ROCm。同样在 utils.cuh 中添加宏定义。
4. cache_controller 条件修复：在 start_writing() 中，除了检查 layout == "page_first" 和 io_backend == "kernel" 外，额外检查 can_use_write_back_jit 属性（通过 getattr 安全访问）。仅在 JIT kernel 可用时保留 host indices 在 CPU，否则移到设备，避免非 JIT 路径崩溃。
5. 新增单元测试：创建 test_hicache_page_first_write_back.py，测试 MHA 和 MLA 场景的 page_first 写回路径，同时在 CUDA 和 AMD CI 中注册，验证 JIT kernel 在 ROCm 上的

构建和执行。

6. 回退 #28473：在 PR 描述中明确要求合并后回退 #28473，移除对 ROCm 强制 layer_first 的 fallback。

关键文件：

- python/sglang/srt/mem_cache/memory_pool_host.py（模块 缓存层；类别 source；类型 core-logic；符号 MLATokenToKVPoolHost.init, MLATokenToKVPoolHost._init_write_back_staging_buffers）：核心开关：扩展 can_use_jit 到 HIP，使 ROCm 使用 JIT HiCache 路径
- python/sglang/srt/mem_cache/pool_host/mha.py（模块 缓存层；类别 source；类型 core-logic；符号 MHATokenToKVPoolHost.init, MHATokenToKVPoolHost._init_write_back_staging_buffers）：MHA 池对应启用 JIT：与 MLA 池相同的 HIP 扩展
- python/sglang/srt/managers/cache_controller.py（模块 控制器；类别 source；类型 entrypoint；符号 HiCacheController.start_writing）：修复崩溃的关键：在 start_writing 中检查 can_use_write_back_jit，避免在 JIT 不可用时保留 CPU 索引
- python/sglang/jit_kernel/csrc/kvcacheio/hicache.cuh（模块 JIT 内核；类别 other；类型 core-logic；符号 load_nc, store_nc, HiCacheKernel）：平台适配核心：添加 ROCm 非 temporal load/store，使用 Clang 向量类型确保向量化，TensorMatcher 兼容 ROCm
- test/registered/jit/test_hicache_page_first_write_back.py（模块 测试套件；类别 test；类型 test-coverage；符号 _token_indices_for_pages, _pinned_host_pool, _fill_with_offset, _assert_pages_equal）：新增单元测试覆盖 page_first JIT 写回路径，同时在 CUDA 和 AMD CI 中注册

关键符号：MLATokenToKVPoolHost.init, MLATokenToKVPoolHost._init_write_back_staging_buffers, MHATokenToKVPoolHost.init, MHATokenToKVPoolHost._init_write_back_staging_buffers, HiCacheController.start_writing, load_nc (uint1/uint2/uint4), store_nc (uint1/uint2/uint4), HiCacheStagedWriteBackKernel

关键源码片段

python/sglang/srt/mem_cache/memory_pool_host.py

核心开关：扩展 can_use_jit 到 HIP，使 ROCm 使用 JIT HiCache 路径

```
# sglang/srt/mem_cache/memory_pool_host.py (partial)

class MLATokenToKVPoolHost(HiSparseHostPoolMixin, HostKVCache):
    def __init__(self, ...):
        # ... (super init omitted)

        # 关键变更：将 _is_cuda 扩展为 _is_cuda or _is_hip
        # 因为 JIT HiCache 内核也通过 hipcc 编译，并且 hicache.cuh 中
        # 的 PTX 辅助函数已被 USE_ROCM 保护，ROCm 使用非 temporal 内建函数作为替代
        self.can_use_jit = (_is_cuda or _is_hip) and can_use_hicache_jit_kernel(
            element_size=self.kv_cache_dim * self.dtype.itemsize
```

```
        )
        # ... (data_refs, data_ptrs init omitted)
        self._init_write_back_staging_buffers()

    def _init_write_back_staging_buffers(self):
        self.staging_page_capacity = 0
        self.staging_token_capacity = 0
        self.staging_buffer = None
        self.can_use_write_back_jit = False
        if self.layout != "page_first" or (_is_npu or _is_xpu or _is_mps):
            return

        # 同样，ROCm 上也启用 staged write-back JIT 内核
        self.can_use_write_back_jit = (
            _is_cuda or _is_hip
        ) and can_use_write_back_jit_kernel(
            element_size=self.kv_cache_dim * self.dtype.itemsize,
        )
        if not self.can_use_write_back_jit:
            return

        # 分配 staging buffer（device memory）
        self.staging_page_capacity = min(self.page_num, _WRITE_BACK_STAGING_PAGE_CHUNK)
        self.staging_token_capacity = self.staging_page_capacity * self.page_size
        self.staging_buffer = torch.empty(
            (self.staging_token_capacity, self.layer_num, 1, self.kv_cache_dim),
            dtype=self.dtype,
            device=self.device_pool.device,
        )
```

python/sglang/srt/managers/cache_controller.py

修复崩溃的关键：在 start_writing 中检查 can_use_write_back_jit，避免在 JIT 不可用时保留 CPU 索引

```
# sglang/srt/managers/cache_controller.py (partial)

def start_writing(self) -> None:
    if len(self.write_queue) == 0:
        return

    op = CacheOperation.merge_ops(self.write_queue)

    # Kernel write-back 仅在 page_first 布局且分阶段 JIT 写回内核可用时
    # 才将 host indices 保留在 CPU（JIT 内核会将这些索引通过设备内存传递，
    # 并且接受 CPU 目标索引）。否则，普通的传输内核要求目标索引在设备上，
    # 因此必须将索引移动到设备。如果没有 can_use_write_back_jit 检查，
    # 在后端 JIT 内核不可用的情况下会崩溃，报错 "Destination indices must be a CUDA tensor"。
    if (
        self.io_backend == "kernel"
```

```
            and self.mem_pool_host.layout == "page_first"
            and getattr(self.mem_pool_host, "can_use_write_back_jit", False)
        ):
            host_indices, device_indices = op.host_indices, op.device_indices
        else:
            host_indices, device_indices = self.move_indices(
                op.host_indices, op.device_indices
            )
        self.write_queue.clear()
        # ... (stream 和实际写回调用省略)
```

python/sglang/jit_kernel/csrc/kvcacheio/hicache.cuh

平台适配核心：添加 ROCm 非 temporal load/store，使用 Clang 向量类型确保向量化，TensorMatcher 兼容 ROCm

```
// python/sglang/jit_kernel/csrc/kvcacheio/hicache.cuh (partial)

// NVIDIA 通过 PTX 指令 ld.global.L1::no_allocate 提供显式的一级缓存旁路。
// ROCm 没有等效 PTX，但非 temporal（流式）load/store 表达了相同的意图，
// 适用于 HiCache 写回这类不应污染缓存的一次性流量。
// 将 PTX 保护在 USE_ROCM 之后，使 JIT 模块也能通过 hipcc 编译。

#ifdef USE_ROCM
// 使用 Clang 原生向量类型，使得单个 __builtin_nontemporal_{load,store}
// 映射到一条向量化的 global_{load,store}_dwordx{2,4} 指令。
// 如果发出 N 个独立的 32 位非 temporal 操作，编译器可能不会合并，
// 甚至可能丢弃 non-temporal 提示，从而降低 HiCache 带宽。
// uint2/uint4 带有 8B/16B 对齐，因此指针 reinterpret_cast 保持正确对齐。
typedef uint32_t native_uint2 __attribute__((ext_vector_type(2)));
typedef uint32_t native_uint4 __attribute__((ext_vector_type(4)));
#endif

SGL_DEVICE uint1 load_nc(const uint1* __restrict__ src) {
#ifndef USE_ROCM
    uint32_t tmp;
    asm volatile("ld.global.L1::no_allocate.b32 %0,[%1];" : "=r"(tmp) : "l"(src));
    return uint1{tmp};
#else
    return uint1{__builtin_nontemporal_load(&src->x)};
#endif
}

SGL_DEVICE uint2 load_nc(const uint2* __restrict__ src) {
#ifndef USE_ROCM
    uint32_t tmp0, tmp1;
    asm volatile("ld.global.L1::no_allocate.v2.b32 {%0,%1},[%2];" : "=r"(tmp0), "=r"(tmp1) : "l"(src))
    ;
    return uint2{tmp0, tmp1};
#else
```

```
    native_uint2 tmp = __builtin_nontemporal_load(
        reinterpret_cast<const native_uint2*>(src));
    return __builtin_bit_cast(uint2, tmp);
#endif
}

SGL_DEVICE uint4 load_nc(const uint4* __restrict__ src) {
#ifndef USE_ROCM
    uint32_t tmp0, tmp1, tmp2, tmp3;
    asm volatile("ld.global.L1::no_allocate.v4.b32 {%0,%1,%2,%3},[%4];" : "=r"(tmp0), "=r"(tmp1),
    "=r"(tmp2), "=r"(tmp3) : "l"(src));
    return uint4{tmp0, tmp1, tmp2, tmp3};
#else
    native_uint4 tmp = __builtin_nontemporal_load(
        reinterpret_cast<const native_uint4*>(src));
    return __builtin_bit_cast(uint4, tmp);
#endif
}

SGL_DEVICE void store_nc(uint1* __restrict__ dst, const uint1& value) {
#ifndef USE_ROCM
    uint32_t tmp = value.x;
    asm volatile("st.global.L1::no_allocate.b32 [%0],%1;" ::"l"(dst), "r"(tmp));
#else
    __builtin_nontemporal_store(value.x, &dst->x);
#endif
}

SGL_DEVICE void store_nc(uint2* __restrict__ dst, const uint2& value) {
#ifndef USE_ROCM
    uint32_t tmp0 = value.x; uint32_t tmp1 = value.y;
    asm volatile("st.global.L1::no_allocate.v2.b32 [%0],{%1,%2};" ::"l"(dst), "r"(tmp0), "r"(tmp1));
#else
    __builtin_nontemporal_store(
        __builtin_bit_cast(native_uint2, value),
        reinterpret_cast<native_uint2*>(dst));
#endif
}

SGL_DEVICE void store_nc(uint4* __restrict__ dst, const uint4& value) {
#ifndef USE_ROCM
    uint32_t tmp0 = value.x; uint32_t tmp1 = value.y; uint32_t tmp2 = value.z; uint32_t tmp3 =
    value.w;
    asm volatile("st.global.L1::no_allocate.v4.b32 [%0],{%1,%2,%3,%4};" ::"l"(dst), "r"(tmp0),
    "r"(tmp1), "r"(tmp2), "r"(tmp3));
#else
    __builtin_nontemporal_store(
        __builtin_bit_cast(native_uint4, value),
        reinterpret_cast<native_uint4*>(dst));
```

```
#endif
}
```

评论区精华

主要 review 讨论集中在三点：

- 向量化非临时 load/store：*gemini-code-assist* 建议使用 Clang *ext_vector_type* 和 __builtin_bit_cast 来确保单个向量化指令，避免多个 32 位操作导致编译器不合并。作者采纳并修改。
- HostPoolGroup 兼容：*gemini* 提到当 *mem_pool_host* 是 HostPoolGroup 时可能没有 *can_use_jit* 属性。作者最终使用了 *getattr(self.mem_pool_host, "can_use_write_back_jit", False)* 来安全降级，但此问题未完全解决，仍存在静默降级风险。
- 使用 kDLGPU 宏：HaiShaw 建议用统一的 kDLGPU 替代显式枚举 kDLCUDA, kDLROCM，作者实施并修改了所有相关匹配器，减少了平台重复。
  - ROCm non-temporal load/store 向量化 (performance): 作者采纳，在 hicache.cuh 中实现向量化版本
  - HostPoolGroup 属性兼容 (design): 作者使用 *getattr* + 默认 False 部分解决，但 HostPoolGroup 场景仍未完全覆盖
  - 使用 kDLGPU 宏统一设备类型 (design): 作者实施并在所有 TensorMatcher 中替换

风险与影响

- 风险：
  1. ROCm JIT kernel 性能风险：非临时 load/store 的向量化质量依赖 Clang 编译器，可能仍有退化，需通过 benchmark 验证。
  2. HostPoolGroup 覆盖不足：如果 mem_pool_host 是分组池，*can_use_write_back_jit* 属性可能不存在，*getattr* 返回 False，导致 JIT 路径被静默关闭，用户可能未察觉。需要后续检查或统一添加属性。
  3. #28473 回退依赖：若未按计划回退，ROCm 仍会被强制 layer_first，PR 效果被抵消。需手动跟踪。
  4. 集成测试缺失：AMD CI 仅运行单元测试，集成场景（PD 分离、大模型）仅在 nightly 中覆盖，可能遗漏边界条件。 - 影响：对用户：ROCm（AMD）用户现在可以使用 page_first 布局 + kernel 写回，获得与 CUDA 相似的性能和稳定性。之前受影响的用户（被迫使用 layer_first）可恢复正常配置。对系统：ROCm 和 CUDA 在 HiCache 写回上保持一致，降低维护成本。对团队：需确保 #28473 被回退，否则布局强制 fallback 仍会生效。新测试在 AMD CI 中运行，增强信心。 - 风险标记：ROCm JIT 路径无集成测试，HostPoolGroup 可能静默降级，#28473 需手动回退

关联脉络

- PR #28473 [AMD] Fall back to layer_first layout for kernel write-back on ROCm: 先前的 workaround 强制 ROCm 使用 layer_first，此 PR 修复根本原因后需要回退该 PR

- PR #21631 [HiCache & JIT Kernel] Refactoring HiCache Write-Back Kernel: 该 PR 引入了 page_first + JIT 写回路径，使 ROCm 默认 page_first 后暴露问题