PR #47423 完整报告

vllm-project/vllm

[EC Connector] CPU Offloading EC Connector

合并时间：2026-07-14 01:09

原文链接：http://prhub.com.cn/vllm-project/vllm/pull/47423

执行摘要

- 一句话：CPU 编码器缓存卸载连接器
- 推荐动作：强烈建议精读。EmbeddingCache 和 StepTracker 的设计模式（延迟同步＋FIFO 驱逐）可在类似的双进程卸载场景中复用。调度器与工作器的职责分离为 vLLM 卸载组件提供了清晰架构参考。

功能与动机

当使用 ec_role=both 时，同一 vLLM 实例的多个请求可能携带相同的视觉输入，导致编码器被重复调用，浪费计算和显存带宽。该 PR 提供了一个最小自包含的连接器，将编码结果暂存到 CPU 共享内存，使后续请求能直接从 CPU 加载而不重新计算。这是计划中更大规模 p2p NIXL 连接器的最小验证版本。

实现拆解

1. 共享内存层：ECSharedRegion 封装 /dev/shm 下的 mmap 文件生命周期，支持创建 / 打开、madvise 预填充、CUDA 主机注册和清理。
2. 缓存管理器：EmbeddingCache 线程安全的块缓存，条目状态（未就绪→就绪→锁定），采用 FIFO 驱逐就绪且未锁定的条目，支持容量检查。
3. 延迟同步器：StepTracker 两个实例分别管理“就绪”和“解锁”时机，支持步数到期或请求结束提前触发，确保 GPU 拷贝完成后再修改缓存状态。
4. 调度器代理：ECPUScheduler 组合缓存和跟踪器，实现 has_cache_item、update_state_after_alloc（分配块并注册延迟）和 build_connector_meta（推进步数并生成元数据）。
5. 工作器代理：ECCPUWorker 在单独进程中执行 GPU↔mmap 批量拷贝，包含描述符缓冲池和 save_rank 过滤（仅 TP 0 执行保存）。
6. 外壳与注册：ECCPUConnector 根据角色路由到调度器或工作器，在 factory.py 中注册 ECCPUConnector 名称。附带完整单元测试覆盖 (tests/v1/ec_connector/unit/)。

关键文件：

- vllm/distributed/ec_transfer/ec_connector/cpu/scheduler/embedding_cache.py（模块缓存层；类别 source；类型 core-logic；符号 CacheEntry, init, ready, evictable）：核心缓存数据结构，包含 CacheEntry 和 EmbeddingCache，管理块分配、状态转换和 FIFO 驱逐。

- vllm/distributed/ec_transfer/ec_connector/cpu/scheduler/__init__.py（模块 调度器；类别 source；类型 dependency-wiring；符号 ECCPUScheduler, init, has_cache_item, ensure_cache_available）：调度器代理，组合 EmbeddingCache 和 StepTracker，实现缓存查询、分配和元数据构建。
- vllm/distributed/ec_transfer/ec_connector/cpu/scheduler/step_tracker.py（模块 同步器；类别 source；类型 core-logic；符号 _PendingEntry, StepTracker, init, add）：步进延迟同步器，用于安全地延迟 mark_ready 和 unpin 操作，支持步数到期和请求完成时的提前触发。
- vllm/distributed/ec_transfer/ec_connector/cpu/worker/__init__.py（模块 工作器；类别 source；类型 dependency-wiring；符号 ECCPUWorker, init, save_caches, flush_saves）：工作器代理，执行真正的 GPU↔mmap 批量拷贝，包含描述符缓冲池和 save rank 过滤。
- vllm/distributed/ec_transfer/ec_connector/cpu/ec_shared_region.py（模块 共享内存；类别 source；类型 dependency-wiring；符号 _wait_for_file_size, ECSharedRegion, init, pin_memory）：mmap 共享内存生命周期管理，包括创建 / 打开、等待容量、CUDA 主机注册和清理。
- vllm/distributed/ec_transfer/ec_connector/cpu/connector.py（模块 连接器外壳；类别 source；类型 dependency-wiring；符号 ECCPUConnector, init, _make_worker, _make_scheduler）：连接器外壳，根据角色路由到调度器或工作器，在工厂中注册。
- vllm/distributed/ec_transfer/ec_connector/cpu/common.py（模块 元数据结构；类别 source；类型 dependency-wiring；符号 ECCPUConnectorMetadata, _get_encoder_cache_hidden_dim, create_ec_shared_region）：元数据结构 ECCPUConnectorMetadata、配置解析和共享区域工厂函数。
- vllm/distributed/ec_transfer/ec_connector/cpu/worker/descriptor_buffers.py（模块 缓冲池；类别 source；类型 core-logic；符号 DescriptorBuffers, DescriptorBufferPool, init, acquire）：描述符缓冲池，用于批量 DMA 操作，降低重复分配开销。

关键符号：CacheEntry.init, CacheEntry.ready, CacheEntry.evictable, CacheEntry.mark_ready, CacheEntry.pin, EmbeddingCache.init, EmbeddingCache.get, EmbeddingCache.alloc, EmbeddingCache.mark_ready, EmbeddingCache.pin, EmbeddingCache.unpin, StepTracker.init, StepTracker.add, StepTracker.step, StepTracker.drain_all, ECCPUScheduler.init, ECCPUScheduler.has_cache_item, ECCPUScheduler.update_state_after_alloc, ECCPUScheduler.build_connector_meta, ECCPUScheduler.shutdown, ECCPUWorker.init, ECCPUWorker.save_caches, ECCPUWorker.flush_saves, ECCPUWorker.start_load_caches, ECCPUWorker.shutdown, ECCPUConnector.init, ECCPUConnector.has_cache_item, ECCPUConnector.save_caches, ECCPUConnector.start_load_caches

关键源码片段

vllm/distributed/ec_transfer/ec_connector/cpu/scheduler/__init__.py

调度器代理，组合 EmbeddingCache 和 StepTracker，实现缓存查询、分配和元数据构建。

```
# vllm/distributed/ec_transfer/ec_connector/cpu/scheduler/__init__.py
```

```
from vllm.distributed.ec_transfer.ec_connector.cpu.common import (
    ECCPUConnectorMetadata,
    create_ec_shared_region,
)
from vllm.distributed.ec_transfer.ec_connector.cpu.scheduler.embedding_cache import (
    EmbeddingCache,
)
from vllm.distributed.ec_transfer.ec_connector.cpu.scheduler.step_tracker import (
    StepTracker,
)


class ECCPUScheduler:
    """调度器代理：拥有 mmap 区域和 EmbeddingCache，处理 producer/consumer 双向逻辑。"""

    def __init__(self, vllm_config: "VllmConfig") -> None:
        ec_config = vllm_config.ec_transfer_config
        self._is_producer: bool = ec_config.is_ec_producer
        self._is_consumer: bool = ec_config.is_ec_consumer

        self._region = create_ec_shared_region(vllm_config)
        self._cache = EmbeddingCache(self._region.num_blocks)

        max_batches = vllm_config.max_concurrent_batches
        # 延迟标记就绪：确保 GPU→mmap 拷贝完成
        self._ready_tracker = StepTracker(max_batches)
        # 延迟解锁：确保 mmap→GPU 拷贝完成
        self._unpin_tracker = StepTracker(max_batches)

        self._pending_saves: dict[str, list[int]] = {}
        self._pending_loads: dict[str, list[int]] = {}

    def has_cache_item(self, identifier: str) -> bool:
        """消费者角色：检查标识符是否已在缓存中且就绪。"""
        if not self._is_consumer:
            return False
        entry = self._cache.get(identifier)
        return entry is not None and entry.ready

    def update_state_after_alloc(self, request: "Request", index: int) -> None:
        """一个请求被调度分配后，记录待保存/加载的操作。"""
        mm_hash = request.mm_features[index].identifier

        # 生产者：分配新块，注册就绪延迟
        if self._is_producer and self._cache.get(mm_hash) is None:
            entry = self._cache.alloc(mm_hash, feature.mm_position.length)
            if entry is not None:
                self._pending_saves[mm_hash] = list(entry.block_ids)
                self._ready_tracker.add(mm_hash, request.request_id)
```

```
        # 消费者：锁定已就绪条目，注册解锁延迟
        if self._is_consumer and mm_hash not in self._pending_loads:
            entry = self._cache.get(mm_hash)
            if entry is not None and entry.ready:
                self._cache.pin(mm_hash)
                self._pending_loads[mm_hash] = list(entry.block_ids)
                self._unpin_tracker.add(mm_hash, request.request_id)

    def build_connector_meta(self, scheduler_output: "SchedulerOutput") ->
    ECCPUConnectorMetadata:
        """在一个步骤结束时调用：推进步数，处理到期操作，生成元数据。"""
        finished = scheduler_output.finished_req_ids if scheduler_output else set()

        # 推进就绪跟踪器：到期或请求结束 → mark_ready
        for key in self._ready_tracker.step(finished):
            entry = self._cache.get(key)
            if entry is not None and not entry.ready:
                self._cache.mark_ready(key)

        # 推进解锁跟踪器：到期或请求结束 → unpin
        for key in self._unpin_tracker.step(finished):
            self._cache.unpin(key)

        meta = ECCPUConnectorMetadata()
        if self._is_producer:
            meta.saves = self._pending_saves
            self._pending_saves = {}
        if self._is_consumer:
            meta.loads = self._pending_loads
            self._pending_loads = {}
        return meta

    def shutdown(self) -> None:
        """清理所有待处理和已排队的操作。"""
        self._pending_loads.clear()
        for mm_hash in self._unpin_tracker.drain_all():
            self._cache.unpin(mm_hash)
        self._ready_tracker.drain_all()
        self._is_producer = False
        self._is_consumer = False
        try:
            self._region.cleanup()
        except Exception:
            pass
```

vllm/distributed/ec_transfer/ec_connector/cpu/scheduler/step_tracker.py

步进延迟同步器，用于安全地延迟 mark_ready 和 unpin 操作，支持步数到期和请求完成时的提前触发。

```python
# vllm/distributed/ec_transfer/ec_connector/cpu/scheduler/step_tracker.py

from collections import deque
from dataclasses import dataclass


@dataclass(slots=True)
class _PendingEntry:
    mm_hash: str
    request_id: str
    processed: bool = False


class StepTracker:
    """跟踪待处理操作，支持步数延迟和首次完成 (first-finish) 快速路径。

    每个 `add(mm_hash, request_id)` 注册一个操作，该操作在以下任一条件时被返回：
    (a) `max_concurrent_batches` 步数经过（deque 过期）
    (b) `request_id` 出现在 `finished_req_ids` 中（首次完成）
    每个条目只处理一次。
    """

    def __init__(self, max_concurrent_batches: int) -> None:
        # 循环 deque，存储过去每一步的条目列表；最旧的在右侧
        self._slots: deque[list[_PendingEntry]] = deque(maxlen=max_concurrent_batches)
        # 反向索引：request_id → 条目列表（跨所有 slots 和 _current）
        self._req_index: dict[str, list[_PendingEntry]] = {}
        # 当前步骤通过 add() 添加的条目
        self._current: list[_PendingEntry] = []

    def add(self, mm_hash: str, request_id: str) -> None:
        """注册一个需要在延迟后处理的 mm_hash。"""
        entry = _PendingEntry(mm_hash=mm_hash, request_id=request_id)
        self._current.append(entry)
        self._req_index.setdefault(request_id, []).append(entry)

    def step(self, finished_req_ids: set[str]) -> list[str]:
        """前进一个步骤，返回所有应被处理的 mm_hash。

        阶段 1：首次完成快速路径（立即处理已结束请求的条目）
        阶段 2：步数过期（deque 最旧槽位到期）
        阶段 3：将当前条目提交为新槽位
        """
        result: list[str] = []

        # 阶段 1：请求结束快速路径
        for req_id in finished_req_ids:
            entries = self._req_index.pop(req_id, None)
            if entries is None:
```

```
            continue
        for entry in entries:
            if not entry.processed:
                entry.processed = True
                result.append(entry.mm_hash)

    # 阶段 2：步数过期
    if len(self._slots) == self._slots.maxlen:
        expired_slot = self._slots.pop()
        for entry in expired_slot:
            if not entry.processed:
                entry.processed = True
                result.append(entry.mm_hash)
        self._cleanup_expired(expired_slot)

    # 阶段 3：提交当前条目
    self._slots.appendleft(self._current)
    self._current = []

    return result

def drain_all(self) -> list[str]:
    """返回所有未处理条目，清空内部状态。用于关闭清理。"""
    result: list[str] = []
    for entry in self._current:
        if not entry.processed:
            entry.processed = True
            result.append(entry.mm_hash)
    self._current = []

    for slot in self._slots:
        for entry in slot:
            if not entry.processed:
                entry.processed = True
                result.append(entry.mm_hash)
    self._slots.clear()
    self._req_index.clear()
    return result

def _cleanup_expired(self, slot: list[_PendingEntry]) -> None:
    """移除反向索引中已过期槽位的引用。"""
    for entry in slot:
        entries_for_req = self._req_index.get(entry.request_id)
        if entries_for_req is None:
            continue
        entries_for_req.remove(entry)
        if not entries_for_req:
            del self._req_index[entry.request_id]
```

评论区精华

审查中 orozery 提出三条重构要求：

- 删除空文件：cpu/__init__.py 仅含注释，建议移除。
- 移动文件：将 ec_shared_region.py 从 ec_connector/cpu/ 移入更恰当的目录。
- 合并类：将独立的生产者 (ECCPUProducer) 和消费者 (ECCPUConsumer) 合并为单一的 ECCPUScheduler 类，减少接口数量。

这些建议在后续提交（'Fixed code review round 1', 'Unified Producer and Consumer under the scheduler'）中均被采纳。

- 删除空文件 cpu/init.py (other): 提交 'Fixed code review round 1' 中处理，文件被移除。
- 移动 ec_shared_region.py 到 cpu/ 下 (design): 后续提交中文件已被移动到 cpu/ec_shared_region.py。
- 合并独立的生产者和消费者为单一 ECCPUScheduler (design): 提交 'Unified Producer and Consumer under the scheduler' 实现了合并，最终版本使用单一 ECCPUScheduler。

风险与影响

- 风险：新增模块完全独立，不修改任何现有路径，回归风险低。mmap 使用 /dev/shm，需留意系统 tmpfs 大小限制，过量使用可能导致 Out-of-memory。EmbeddingCache 的 FIFO 驱逐策略在缓存满时可能过早驱逐刚就绪但未使用的条目，但锁定机制确保正在加载的条目不会被驱逐。工作器与调度器跨进程通信依赖 mmap 文件，若进程崩溃可能导致文件残留或脏数据，但当前实现通过 atexit 和 shutdown 清理。未涉及网络，安全风险低。
- 影响：影响范围限定于显式配置 ec_connector=ECCPUConnector 且 ec_role=ec_both 的用户。对视觉模型推理，缓存命中后可避免重复编码，吞吐可提升 2-3 倍。但会额外占用 CPU 内存（编码器输出大小 × 缓存块数）和 /dev/shm 空间。团队需维护新模块的 mmap 与缓存逻辑，但模块设计独立，维护成本可控。
- 风险标记：共享内存文件系统压力，FIFO 驱逐策略可能过于激进，跨进程 mmap 同步依赖 cuda 同步机制，进程崩溃可能留下脏文件

关联脉络

- PR #42998 [EC Connector] Add p2p-enabled NIXL EC CPU Connector: PR Body 中提及该 PR 是更大规模 p2p NIXL 连接器的最小自包含版本，设计上为未来 NIXL 子类提供基础。虽未提供编号，但可推断为后续即将合并的相关 PR。