PR #25954 完整报告

sgl-project/sglang

add LRU eviction for mooncacke embedding cache

合并时间：2026-06-12 12:38

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/25954

执行摘要

- 一句话：为 Mooncake Embedding Cache 添加 LRU 淘汰机制
- 推荐动作：该 PR 值得精读，尤其是引用计数设计、锁策略以及与 RDMA 异步操作的交互。展示了如何在缓存系统中安全地添加逐出而不破坏在途 I/O，对类似系统的缓存实现有参考价值。

功能与动机

当前 embedding_cache_controller 没有本地缓冲驱逐，当本地缓冲区满时会导致问题。PR 旨在实现 LRU 驱逐来管理内存，并修复全局缓存中的内存泄漏。

实现拆解

1. 优化 ContiguousMemoryAllocator：在 *embedding_cache_controller.py* 中，为 ContiguousMemoryAllocator 添加 *allocated_size* 运行计数器，*allocate* 和 *free* 时更新，使 *get_allocated_size()* 成为 O(1) 操作；同时新增 *get_free_size()* 方法。
2. 扩展 EmbeddingCacheController 结构：在 __init__ 中新增参数 *enable_eviction* 和 max_eviction_batch。增加 *access_order*（字典）用于 LRU 追踪，通过删除再插入键将访问过的哈希移到末尾。增加 ref_counts 字典用于 RDMA 引用计数，防止正在使用的内存被驱逐。增加 stats 记录总分配、总驱逐、分配失败等。
3. 实现 LRU 驱逐核心逻辑：新增 *_select_eviction_candidates* 遍历 *access_order* 自然顺序，跳过引用计数大于零的条目，收集候选直到满足所需字节或达到 max_eviction_batch。新增 *_evict_hashes* 执行驱逐：从 hash_to_metadata 删除、释放分配器内存、从 *access_order* 删除。新增 *_allocate_with_eviction*：先尝试正常分配，失败后调用驱逐再重试。
4. 接入预取 / 插入流程：在 prefetch、insert_batch、*get_embeddings* 等方法中调用 *_update_access_time* 更新访问时间，在 RDMA 操作前后增减引用计数，保证驱逐安全。
5. 调整 *encode_server.py* 处理部分预取失败：将布尔型 prefetch_status 改为按元素的 fallback_mask。预取完成后检查每个命中哈希是否真正获取到数据（可能因池满而失败），失败则标记为需 ViT 回退。所有 rank 对标记项执行 ViT 编码。同时释放缓存引用。
6. 扩展单元测试：新增 test_embedding_cache_controller.py，覆盖 ContiguousMemoryAllocator 基本操作、大小跟踪、LRU 驱逐策略、引用计数场景、统计数据。

关键文件：

- python/sglang/srt/mem_cache/storage/mooncake_store/embedding_cache_controller.py（模块 缓存控制器；类别 source；类型 core-logic；符号 get_allocated_size, get_free_size, _update_access_time, _protect_hash）：核心修改文件，新增 LRU 驱逐、引用计数、统计功能，重构分配器大小追踪。
- test/registered/unit/mem_cache/test_embedding_cache_controller.py（模块 测试；类别 test；类型 test-coverage；符号 TestContiguousMemoryAllocator, test_basic_alloc_free, test_alloc_fails_when_full, test_free_merges_adjacent）：新增 716 行单元测试，覆盖分配器、LRU 驱逐、引用计数、统计等场景。
- python/sglang/srt/disaggregation/encode_server.py（模块 编码服务器；类别 source；类型 core-logic）：修改预取流程，使用 fallback_mask 替换布尔状态，支持部分加载失败时的 ViT 降级。

关键符号：get_allocated_size, get_free_size, _update_access_time, _protect_hash, _release_hash, _select_eviction_candidates, _evict_hashes, _allocate_with_eviction

关键源码片段

python/sglang/srt/mem_cache/storage/mooncake_store/embedding_cache_controller.py

核心修改文件，新增 LRU 驱逐、引用计数、统计功能，重构分配器大小追踪。

```
# python/sglang/srt/mem_cache/storage/mooncake_store/embedding_cache_controller.py
# ContiguousMemoryAllocator 中 allocate 和 free 的改进
# 使用运行计数器获得 O(1) 的 allocated_size

class ContiguousMemoryAllocator:
    def __init__(self, total_size_bytes: int):
        self.total_size = total_size_bytes
        self.free_blocks = [(0, total_size_bytes)]
        self.allocated_map = {} # {offset: size_bytes}
        self.allocated_size = 0 # 新增运行计数器
        self.lock = threading.Lock()

    def allocate(self, size_bytes: int) -> Optional[int]:
        with self.lock:
            # 简单 First-Fit 分配
            for i, (offset, block_size) in enumerate(self.free_blocks):
                if block_size >= size_bytes:
                    remaining_size = block_size - size_bytes
                    if remaining_size > 0:
                        self.free_blocks[i] = (offset + size_bytes, remaining_size)
                    else:
                        self.free_blocks.pop(i)
                    self.allocated_map[offset] = size_bytes
                    self.allocated_size += size_bytes # 更新计数器
                    return offset
```

```
            return None

    def free(self, offset: int, size_bytes: int):
        with self.lock:
            # 从分配记录中移除并更新计数器
            if offset in self.allocated_map:
                self.allocated_size -= self.allocated_map[offset]
                del self.allocated_map[offset]
            # 归还空闲块并合并相邻块
            self.free_blocks.append((offset, size_bytes))
            self.free_blocks.sort()
            merged = []
            if not self.free_blocks:
                return
            curr_offset, curr_size = self.free_blocks[0]
            for next_offset, next_size in self.free_blocks[1:]:
                if curr_offset + curr_size == next_offset:
                    curr_size += next_size
                else:
                    merged.append((curr_offset, curr_size))
                    curr_offset, curr_size = next_offset, next_size
            merged.append((curr_offset, curr_size))
            self.free_blocks = merged

    def get_allocated_size(self) -> int:
        """O(1) 返回已分配的字节数"""
        with self.lock:
            return self.allocated_size

    def get_free_size(self) -> int:
        """返回空闲字节数 (O(N) 遍历) """
        with self.lock:
            return sum(block_size for _, block_size in self.free_blocks)
```

python/sglang/srt/disaggregation/encode_server.py

修改预取流程，使用 fallback_mask 替换布尔状态，支持部分加载失败时的 ViT 降级。

```
# python/sglang/srt/disaggregation/encode_server.py
# 改进预取：使用 fallback_mask 处理部分加载失败

# Step 3: Rank 0 预取命中的嵌入并构建 fallback_mask
fallback_mask = torch.zeros(num_items, dtype=torch.int32)
cached_slices = []

if self.rank == 0:
    if hit_indices:
        hit_hashes = [str_mm_hashes[i] for i in hit_indices]
        hit_tokens = [
            self.get_num_tokens(grid_thw[i], modality) for i in hit_indices
```

```
        ]
        self.mm_global_cache.prefetch(req_id, hit_hashes, hit_tokens, modality)
        try:
            async def _wait_prefetch():
                while not self.mm_global_cache.check_prefetch_progress(req_id):
                    await asyncio.sleep(0.005)
            await asyncio.wait_for(_wait_prefetch(), timeout=60.0)
            # 检查哪些项实际加载了
            cached_slices = self.mm_global_cache.get_embeddings(hit_hashes)
            for i, idx in enumerate(hit_indices):
                if cached_slices[i] is None:
                    fallback_mask[idx] = 1 # 标记为需要 ViT 回退
        except (asyncio.TimeoutError, Exception) as e:
            for idx in hit_indices:
                fallback_mask[idx] = 1

    # Step 4: 广播 fallback_mask 到所有 rank
    if self.server_args.tp_size > 1:
        torch.distributed.broadcast(fallback_mask, src=0,
            group=self.mm_global_cache.prefetch_tp_group)

    # Step 5: 所有 rank 对需要回退的项执行 ViT
    fallback_indices = [i for i in range(num_items) if fallback_mask[i].item() == 1]
    fallback_slices = None
    if fallback_indices:
        fallback_slices = self._encode_missing(
            mm_feature, mm_inputs, fallback_indices, modality, get_feature_fn
        )
```

评论区精华

- gemini-code-assist[bot]指出 _select_eviction_candidates 中排序 access_order 是 O(N log N)，建议利用 Python dict 的插入顺序直接迭代。作者采纳，改用 dict 删除再插入策略。
- liusy58质疑驱逐安全性："What happens if an embedding gets evicted mid-transfer — can that actually happen?" 作者回复会添加引用检查，后续提交增加了 ref_counts。
- liusy58在 encode_server.py 中注意到 _encode_missing 只在 rank0 调用可能导致 TP>1 死锁，作者确认并移除该问题。
- gemini-code-assist[bot]建议使用运行计数器代替 sum(allocated_map.values()) 实现 O(1) get_allocated_size，作者采纳。
- liusy58建议未来考虑重叠 ViT 计算和 RDMA 预取以隐藏延迟。
- LRU 驱逐选择排序优化 (performance): 作者接受建议，后续提交改用 dict 删除再插入策略，遍历时直接使用 keys() 顺序。
- 驱逐安全性与引用计数 (correctness): 作者添加 ref_counts 字典，在 RDMA 操作前后增减引用计数，驱逐时跳过引用计数 > 0 的条目。
- TP > 1 下 encode_missing 只被 rank0 调用导致挂起 (correctness): 移除只在 rank0 调用的错误逻辑，改为所有 rank 执行 ViT 回退。

- 运行计数器替代 sum() 实现 O(1) get_allocated_size (performance): 作者采纳，添加 allocated_size 计数器并在 allocate/free 中更新。
- 建议重叠 ViT 和 RDMA 预取作为后续工作 (design): 同意作为后续优化，不阻塞此 PR。

风险与影响

- 风险:
  - 并发安全性：多个锁（access_lock、allocator.lock）嵌套调用可能引入死锁，需确保锁顺序一致。
  - 引用计数泄漏：ref_counts 的 increment/decrement 必须严格配对，否则条目永不被驱逐。
  - 驱逐粒度：max_eviction_batch 固定值为 100，可能无法满足超大分配需求需全部驱逐的场景。
  - 性能开销：_select_eviction_candidates 仍以 O(N) 遍历所有条目，高并发下可能增加延迟。
  - 兼容性：新增参数默认启用，对现有配置无破坏性。
- 影响:
  - 用户影响：使用 Mooncake Embedding Cache 的用户自动获得 LRU 驱逐，避免本地池满报错；预取部分失败时自动降级到 ViT，提高稳定性。参数 max_eviction_batch 可调。
  - 系统影响：内存管理更高效，但引入引用计数和锁开销，主要影响多模态推理中使用了全局嵌入缓存的场景。
  - 团队影响：代码结构更清晰，但并发复杂性增加，需仔细审计引用计数逻辑。
  - 风险标记：核心缓存路径变更，并发安全需验证，引用计数可能泄漏，驱逐批次可能不足

关联脉络

- 暂无明显关联 PR