PR #37206 完整报告

vllm-project/vllm

[KV Offload] Unified memory layout for offloading workers

合并时间：2026-04-15 02:33

原文链接：http://prhub.com.cn/vllm-project/vllm/pull/37206

执行摘要

- 一句话：引入跨 TP 工作者共享的 mmap 内存区域，实现 KV 卸载的统一布局，提升跨实例兼容性。
- 推荐动作：该 PR 值得精读，特别是关注 SharedOffloadRegion 的设计如何协调多工作者内存映射，以及 compute_sub_block_ptrs 的向量化优化如何支持非连续布局。建议团队学习其错误处理和性能权衡的讨论，以应用于类似共享资源场景。

功能与动机

根据 PR body 描述，之前每个 TP 工作者使用 torch.zeros(..., pin_memory=True) 独立分配 CPU tensors，导致内存布局完全依赖于 TP 配置，无法在不同并行配置的 vLLM 实例间共享或迁移 KV 块。新目标是通过一个统一的 mmap 区域（路径为 /dev/shm/vllm_offload_{instance_id}.mmap）实现 TP 无关的交错块布局，从而支持跨实例的 KV 块操作，提升系统灵活性和资源利用率。

实现拆解

1. 新增 SharedOffloadRegion 类（文件：vllm/v1/kv_offload/cpu/shared_offload_region.py）：
   - 负责创建和管理共享 mmap 文件，通过 __init__ 方法使用 O_EXCL 标志协调多个工作者的文件创建（第一个工作者创建并 truncate 文件，其他工作者等待文件大小）。
   - 实现交错布局：内存按块行组织，每行包含所有工作者对一个块的贡献，通过 _row_stride 和 _worker_offset 计算偏移。
   - 使用 MADV_POPULATE_WRITE 预填充页面以减少延迟，并提供 create_next_view 方法分配工作者的内存视图。
   - 清理逻辑在 cleanup 方法中处理，包括关闭 mmap、文件描述符和可选的内存 unpin。
2. 修改 CpuGpuOffloadingHandlers（文件：vllm/v1/kv_offload/worker/cpu_gpu.py）：
   - 引入 mmap_region 参数，在初始化时如果提供 SharedOffloadRegion，则调用 mmap_region.create_next_view 获取 CPU tensor，替代原有的 torch.zeros 分配。
   - 添加 pin_mmap_region 函数，使用 torch.cuda.cudart().cudaHostRegister 固定整个 mmap 区域以加速 DMA 传输。
3. 重构块指针计算（文件：vllm/v1/kv_offload/worker/cpu_gpu.py）：

- 将原有的 expand_block_ids 函数重构为 compute_sub_block_ptrs，支持非连续内存布局（如交错块），通过向量化计算子块指针，提高效率。

4. 测试配套更新：
   - 新增 tests/v1/kv_offload/test_shared_offload_region.py，包含多工作者竞争创建、内存访问和清理的单元测试。
   - 修改 tests/v1/kv_offload/test_cpu_gpu.py，增加 use_shared_memory 参数，测试共享内存和传统路径的传输正确性。

关键文件：

- vllm/v1/kv_offload/cpu/shared_offload_region.py（模块 共享内存区域；类别 source；类型 core-logic；符号 _wait_for_file_size, SharedOffloadRegion, init, create_next_view）：核心实现文件，定义了 SharedOffloadRegion 类，负责共享 mmap 区域的创建、布局管理和清理。
- vllm/v1/kv_offload/worker/cpu_gpu.py（模块 卸载处理器；类别 source；类型 core-logic；符号 expand_block_ids, compute_sub_block_ptrs, pin_mmap_region）：主要逻辑修改文件，引入共享内存支持，重构块指针计算并添加内存固定功能。
- tests/v1/kv_offload/test_shared_offload_region.py（模块 单元测试；类别 test；类型 test-coverage；符号 _set_spawn_method, _make_region, _cleanup_file, _region）：新增单元测试文件，全面验证 SharedOffloadRegion 的多工作者协调、内存访问和清理行为。
- tests/v1/kv_offload/test_cpu_gpu.py（模块 集成测试；类别 test；类型 test-coverage）：修改现有测试文件，增加 use_shared_memory 参数以覆盖共享内存和传统路径的传输测试。

关键符号：SharedOffloadRegion.init, SharedOffloadRegion.create_next_view, SharedOffloadRegion.cleanup, compute_sub_block_ptrs, pin_mmap_region

关键源码片段

vllm/v1/kv_offload/cpu/shared_offload_region.py

核心实现文件，定义了 SharedOffloadRegion 类，负责共享 mmap 区域的创建、布局管理和清理。

```
class SharedOffloadRegion:
    """
    单mmap支持的内存区域，在vLLM实例的所有工作者间共享。
    工作者通过文件系统协调：第一个工作者用O_EXCL打开文件并调用ftruncate；
    其余工作者打开现有文件并等待其达到预期大小。每个工作者然后mmap()整个文件。
    文件路径：/dev/shm/vllm_offload_{instance_id}.mmap
    """

    def __init__(
        self,
        instance_id: str,
        total_size_bytes: int,
        num_blocks: int,
        rank: int | None,
        num_workers: int,
```

```
    cpu_page_size: int,
) -> None:
    self.page_size = mmap.PAGESIZE
    self.total_size_bytes = total_size_bytes
    self.mmap_path = f"/dev/shm/vllm_offload_{instance_id}.mmap"
    self._creator = False # 仅当此工作者创建文件时为True
    self.num_blocks = num_blocks
    self.rank = rank
    # 交错布局步幅：一行 = 所有工作者对一个块的数据
    self._row_stride = cpu_page_size * num_workers
    if rank is not None:
        # 此工作者在每个块行内的起始字节偏移
        self._worker_offset = rank * cpu_page_size
    try:
        # 排他创建 —— 只有一个工作者成功
        self.fd: int | None = os.open(
            self.mmap_path, os.O_CREAT | os.O_EXCL | os.O_RDWR, 0o600
        )
        os.ftruncate(self.fd, self.total_size_bytes)
        self._creator = True
        logger.info(
            "Created mmap file %s (%.2f GB)",
            self.mmap_path,
            self.total_size_bytes / 1e9,
        )
    except FileExistsError:
        self.fd = os.open(self.mmap_path, os.O_RDWR)
        _wait_for_file_size(self.fd, self.total_size_bytes) # 等待文件大小达到预期
        logger.info("Opened existing mmap file %s", self.mmap_path)

    self.mmap_obj: mmap.mmap | None = mmap.mmap(
        self.fd,
        self.total_size_bytes,
        flags=mmap.MAP_SHARED,
        prot=mmap.PROT_READ | mmap.PROT_WRITE,
    )
    # 预填充页面以加速访问
    _MADV_POPULATE_WRITE = getattr(mmap, "MADV_POPULATE_WRITE", 23)
    if rank is not None:
        # 仅预填充此工作者的页面
        worker_offset = rank * cpu_page_size
        for block in range(num_blocks):
            raw_offset = block * self._row_stride + worker_offset
            aligned_offset = (raw_offset // self.page_size) * self.page_size
            end = raw_offset + cpu_page_size
            aligned_length = end - aligned_offset
            self.mmap_obj.madvise(
                _MADV_POPULATE_WRITE, aligned_offset, aligned_length
            )
```

```
        else:
            # 无rank时预填充整个区域
            self.mmap_obj.madvise(_MADV_POPULATE_WRITE, 0, self.total_size_bytes)

        self._base = torch.frombuffer(memoryview(self.mmap_obj), dtype=torch.int8)
        self._views: list[torch.Tensor] = []
        self.is_pinned: bool = False
```

vllm/v1/kv_offload/worker/cpu_gpu.py

主要逻辑修改文件，引入共享内存支持，重构块指针计算并添加内存固定功能。

```
def compute_sub_block_ptrs(
    block_ids: np.ndarray,
    block_size_factor: int,
    output: np.ndarray,
    tensor: torch.Tensor,
    skip_count: int = 0,
):
    """
    计算给定块ID的子块字节指针。
    每个块包含block_size_factor个子块。子块j的指针为：
        base_ptr + b * row_stride + j * sub_block_size
    其中sub_block_size = tensor.shape[1] // block_size_factor（GPU页面大小）。
    这处理了行步幅不等于block_size_factor * sub_block_size的张量（如非连续CPU张量）。
    """
    assert skip_count < block_size_factor
    num_sub_blocks = len(output)
    base_ptr = tensor.data_ptr()
    row_stride = tensor.stride(0)

    if block_size_factor == 1:
        # 快速路径：1:1映射，无需子块扩展
        output[:] = base_ptr + block_ids[:num_sub_blocks] * row_stride
        return

    # 向量化扩展，适用于block_size_factor > 1
    assert tensor.shape[1] % block_size_factor == 0
    sub_block_size = tensor.shape[1] // block_size_factor
    sub_offsets = np.arange(block_size_factor, dtype=np.int64) * sub_block_size
    # (num_blocks, 1) + (1, block_size_factor) -> (num_blocks, block_size_factor)
    all_ptrs = (
        base_ptr + block_ids.astype(np.int64)[:, np.newaxis] * row_stride
    ) + sub_offsets[np.newaxis, :]
    # 展平并应用skip_count/截断
    flat = all_ptrs.ravel()
    output[:] = flat[skip_count : skip_count + num_sub_blocks]
```

评论区精华

- 布局设计讨论：orozery 建议使用交错布局 (num_blocks, num_workers, ...) 以实现 TP 无关性，并指出每个工作者需要固定整个内存区域。讨论中确定了最终方案：块按工作者交错存储，避免工作者间碰撞。
- 性能优化讨论：orozery 询问操作时间（如 ftruncate、cudaHostRegister），omerpaz95 提供了具体数据：mmap MAP_POPULATE 42.95 GB 耗时 50.671 秒，cudaHostRegister 耗时 2.967 秒，而传统 torch.zeros pinned tensor 10.74 GB 耗时 6.488 秒。讨论了 MADV_POPULATE_WRITE 与 cudaHostRegister 的交互，决定保留预填充以减少开销。
- 错误处理与清理：gemini-code-assist[bot] 指出 assert 可能导致崩溃，建议改为异常处理；orozery 建议改进清理逻辑，如添加警告日志和外部触发清理。最终代码调整了错误处理和日志级别。
- 测试覆盖：orozery 要求添加针对 SharedOffloadRegion 的独立单元测试，并移除冗余测试代码，导致新增了完整的测试文件。
- 内存布局设计 (design): 采纳交错布局方案，通过 _row_stride 和 _worker_offset 计算偏移，并在代码中实现。
- 性能测试与优化 (performance): 保留 MADV_POPULATE_WRITE 预填充以减少 cudaHostRegister 开销，并在代码中添加时间日志。
- 错误处理与清理 (correctness): 调整错误处理，添加更健壮的清理代码和警告日志，但未完全移除 assert。

风险与影响

- 风险：- 内存分配竞争风险：SharedOffloadRegion.__init__ 中使用 O_EXCL 创建文件，存在多个工作者同时创建时的竞争条件，尽管有等待机制，但在高并发或网络文件系统下可能超时或失败。
- 性能开销风险：mmap 和 cudaHostRegister 操作可能引入显著延迟（如 PR 中报告的 50 秒以上），尤其在大型内存分配时，可能影响系统启动时间。
- 清理不彻底风险：清理逻辑依赖工作者正确调用 cleanup，如果进程异常退出可能导致 mmap 文件残留，占用 /dev/shm 空间。
- 兼容性风险：依赖于 Linux 特定特性（如 MADV_POPULATE_WRITE），在旧内核或非 Linux 系统上可能降级或失败。
- 影响：- 对系统的影响：统一内存布局使得 KV 卸载块可以跨不同 TP 配置的 vLLM 实例共享，提升了系统在分布式场景下的灵活性和资源复用能力。可能减少总体内存占用，但引入共享协调开销。
- 对用户的影响：用户无需关注 TP 配置即可迁移或共享 KV 块，简化了多实例部署。但启动时间可能增加，需要监控内存使用。
- 对团队的影响：为 kv-offload 模块提供了标准化基础，便于后续扩展（如异步预填充），但增加了代码复杂度，需要维护新的共享内存逻辑。
- 风险标记：内存分配竞争，性能开销，清理不彻底

关联脉络

- PR #37460 [Core][Metrics][BugFix] Replace num_cached_tokens/num_external_computed_tokens with PrefillStats: 该 PR 同样涉及 kv-connector 标签，修改了调度器 metrics，可能与 KV 卸载统计相关，但无直接代码重叠。
- PR #39107 [MoE Refactor] Remove MoE DP chunking: 同属 v1 标签下的核心重构，展示了架构标准化趋势，但无直接关联。