PR #35734 完整报告

sgl-project/sglang

[diffusion] park a layerwise component's non-layer weights between uses

合并时间：2026-08-23 12:05

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/35734

执行摘要

- 一句话：组件闲置期暂存非层权重至主机，缓解显存压力
- 推荐动作：值得精读。这是 layerwise offload 体系的关键补全，有四个值得借鉴的设计决策：① 用“真实可用显存 = driver 空闲 + allocator 未使用预留”衡量 headroom，避免 warm 进程下被缓存分配器误导；② 用 PARK_SIGNIFICANCE 阈值把“主观偏好”转化为“实测门控”，避免大显存卡白付传输成本；③ 占位符按 (device, dtype) 共享而非逐权重分配；④ 刻意不用 pinned memory，换取内核可回收的主机内存。注意 PR 中的 denoise 性能数据因关联 bug 已失效，合并后应在真实 GPU 上重新测量传输开销。

功能与动机

MiniMax-H3 在 864x480 / 124 frames / 20 steps 配置下，VAE decode 占总耗时近一半：decode_temporal 每个 temporal chunk 都会重跑整个 decoder，在 layerwise offload 下 36 个 block 每 chunk 都要重新流式传输，decode 高达 140.60 s。让 VAE 的 36 个块驻留后 decode 降到 23 s，却需要 9 GB 显存；在 12 GiB 硬上限下 decode 仍因差 20 MiB 而 OOM，而此刻 DiT 占着 2.09 GB、文本编码器占着 1.40 GB，全是 decode 阶段碰不到的权重。PR 的解法是把这些非层权重在组件闲置期 park 到主机内存，把显存让给紧随其后的 stage。

实现拆解

实现分三层拆解：

1. 生命周期钩子接入（入口）：在 component_residency_strategies.py 的 LayerwiseOffloadStrategy 中，finish_use 在 manager.release_all() 释放所有 layer 之后追加 module.park_non_layer_weights()；prepare_for_use 在 module.prepare_for_next_req() 之前追加 module.restore_non_layer_weights()。这样组件恰好在自己不被使用的窗口期内让出显存，MPS 分支保持原有逻辑不动。
2. 核心机制（layerwise_offload.py）：LayerwiseOffloadableModuleMixin 新增 park_non_layer_weights() / restore_non_layer_weights() 两个方法。park 时通过 _managed_layer_parameter_names() 从各个 manager 的 _weight_metadata 收集已被流式化的层参数名并排除在外，其余非层参数以 detach().to("cpu", copy=True) 生成一份 pageable 的主机拷贝，原参数则替换为 _park_placeholder() 产生的 (1,) 占位张量；占位符按 (device, dtype) 共享，避免每个权重各自分配。restore 时把主机拷贝搬回设备。buffers 刻意不 park，因为共享的 RoPE cache 等 buffer 被多层引用，驻留是有意为之。

3. 门控与防御：park 行为由 configure_layerwise_offload 中的 server_args.performance_mode == "memory" 决定开关；_device_headroom_bytes() 计算真实可用显存——driver 报告的空闲内存加上缓存分配器已预留但未分配的部分，若非层权重总字节数不超过该余量的 10%（PARK_SIGNIFICANCE = 0.1），则不仅不 park，还会清掉已持有的主机拷贝，防止机器有富余时付出无谓的传输开销。_parked_non_layer_weights 与 _park_placeholders 用懒初始化的 __dict__ 存储属性实现，避免在 Mixin 上引入跨实例共享的类属性。
4. 测试配套：test_layerwise_offload.py 新增 7 个用例，覆盖 memory 模式开关、流式层权重不被 park、非 memory 模式 no-op、余量充足时跳过、余量紧张时执行、余量变大后释放主机拷贝、占位符按 (device, dtype) 共享。测试通过 _headroom() helper 用 monkeypatch 伪造显存测量，不依赖真实 GPU。

关键文件：

- python/sglang/multimodal_gen/runtime/managers/memory_managers/layerwise_offload.py（模块 逐层卸载；类别 source；类型 core-logic；符号 PARK_SIGNIFICANCE, park_non_layer_weights_between_uses, _managed_layer_parameter_names, park_non_layer_weights）：核心实现文件。新增 park_non_layer_weights / restore_non_layer_weights 及配套的 headroom 测量、placeholder 共享与懒初始化存储，是整套机制的主体。
- python/sglang/multimodal_gen/runtime/managers/memory_managers/component_residency_strategies.py（模块 驻留策略；类别 source；类型 core-logic；符号 LayerwiseOffloadStrategy.prepare_for_use, LayerwiseOffloadStrategy.finish_use）：生命周期集成点。LayerwiseOffloadStrategy.finish_use 与 prepare_for_use 分别调用 park 与 restore，是机制生效的入口。
- python/sglang/multimodal_gen/test/unit/test_layerwise_offload.py（模块 卸载测试；类别 test；类型 test-coverage；符号 _ParkableResidentComponent, test_non_layer_parking_follows_memory_performance_mode, test_parking_leaves_streamed_layer_weights_alone, test_parking_is_a_no_op_outside_memory_mode）：测试配套。7 个新用例覆盖模式门控、层权重保护、headroom 阈值、主机拷贝归还与占位符共享，全部基于 monkeypatch 的 fake device。

关键符号：park_non_layer_weights, restore_non_layer_weights, _managed_layer_parameter_names, _device_headroom_bytes, _park_placeholder, LayerwiseOffloadStrategy.prepare_for_use, LayerwiseOffloadStrategy.finish_use

关键源码片段

python/sglang/multimodal_gen/runtime/managers/memory_managers/layerwise_offload.py

核心实现文件。新增 park_non_layer_weights / restore_non_layer_weights 及配套的 headroom 测量、placeholder 共享与懒初始化存储，是整套机制的主体。

```
# Parking 一个组件的非层权重能释放显存，代价是每次使用多两次传输
```

```
# 和一份与 page cache 竞争的主机拷贝。只有当释放量占真实可用显存的
# 比例足够大时才值得做；显存有富余时 park 是纯损失。
PARK_SIGNIFICANCE = 0.1

class LayerwiseOffloadableModuleMixin:
    # 由 configure_layerwise_offload 依据 --performance-mode memory 设置
    park_non_layer_weights_between_uses: bool = False

    def _managed_layer_parameter_names(self) -> set:
        # 已被 layerwise manager 流式化的层参数名，parking 绝不能碰它们
        return {
            name
            for manager in self.layerwise_offload_managers
            for names in manager._weight_metadata.values()
            for name in names
        }

    def park_non_layer_weights(self) -> None:
        # MPS 有自己按 subphase 裁剪的机制，这里直接让路
        if not self.park_non_layer_weights_between_uses:
            return
        if current_platform.is_mps():
            return
        managed = self._managed_layer_parameter_names()
        resident = [
            (name, parameter)
            for name, parameter in self.named_parameters()
            if name not in managed and parameter.device.type != "cpu"
        ]
        holds = sum(p.numel() * p.element_size() for _, p in resident)
        # 关键防御：权重占比不超过余量 10% 时不 park，
        # 并归还已有的主机拷贝，避免大显存卡白付传输成本
        if holds <= self._device_headroom_bytes() * PARK_SIGNIFICANCE:
            self._parked_non_layer_weights.clear()
            return
        parked = self._parked_non_layer_weights
        with torch.inference_mode(False), torch.no_grad():
            for name, parameter in resident:
                if name not in parked:
                    parked[name] = parameter.detach().to("cpu", copy=True)
                # 设备侧只留一个 (1,) 占位符，真实权重由主机拷贝持有
                parameter.data = self._park_placeholder(parameter)

    def _device_headroom_bytes(self) -> int:
        # driver 报告的空闲内存不含缓存分配器已预留未分配的部分，
        # 在 warm 进程上会严重低估余量，因此要加回 unused reserve
        free = int(
            current_platform.get_available_gpu_memory(empty_cache=False) * (1 << 30)
        )
```

```
        device_module = torch.get_device_module()
        unused_reserve = (
            device_module.memory_reserved() - device_module.memory_allocated()
        )
        return free + max(0, unused_reserve)

    def _park_placeholder(self, parameter: torch.Tensor) -> torch.Tensor:
        # 每个 (device, dtype) 只分配一个共享占位符，避免逐权重分配
        key = (parameter.device, parameter.dtype)
        placeholder = self._park_placeholders.get(key)
        if placeholder is None:
            placeholder = torch.empty(
                (1,), dtype=parameter.dtype, device=parameter.device
            )
            self._park_placeholders[key] = placeholder
        return placeholder

    def restore_non_layer_weights(self) -> None:
        # 组件再次被使用前把权重搬回设备。主机拷贝是 pageable 的，
        # 传输会经 driver 的 pinned staging buffer 且必然同步——
        # 刻意不用 pinned memory，避免占住内核无法回收的主机内存
        parked = self._parked_non_layer_weights
        if not parked:
            return
        device = current_platform.get_local_torch_device()
        parameters = dict(self.named_parameters())
        with torch.inference_mode(False), torch.no_grad():
            for name, host_tensor in parked.items():
                parameter = parameters.get(name)
                if parameter is None:
                    continue
                parameter.data = host_tensor.to(device)
```

python/sglang/multimodal_gen/runtime/managers/memory_managers/component_residency_strategies.py

生命周期集成点。LayerwiseOffloadStrategy.finish_use 与 prepare_for_use 分别调用 park 与 restore，是机制生效的入口。

```
class LayerwiseOffloadStrategy(ComponentResidencyStrategy):
    # 组件生命周期钩子：finish_use 先释放所有层，再把非层权重 park 到
    # 主机；prepare_for_use 先把权重 restore 回来，再进入下一轮使用
    def prepare_for_use(self, module, use, state) -> None:
        if isinstance(module, LayerwiseOffloadableModuleMixin):
            # MPS 组件保留 checkpoint-backed CPU 权重、逐层同步物化，
            # 整模块搬移会破坏有界驻留契约，保持原逻辑
            if current_platform.is_mps():
                if module.mps_stream_non_layer_weights:
                    return
                _module_to_local_device(module, dtype=use.target_dtype)
```

```
                return
            module.restore_non_layer_weights()
            module.prepare_for_next_req()

    def finish_use(self, module, use, state) -> None:
        if not isinstance(module, LayerwiseOffloadableModuleMixin):
            return
        for manager in module.layerwise_offload_managers:
            manager.release_all()
        # 层已经释放，组件剩余部分在下次被使用前都是设备上的死重，
        # 而紧随其后的 stage 可能正是需要显存的那一个
        module.park_non_layer_weights()
        if current_platform.is_mps():
            torch.mps.synchronize()
            module.restore_mps_cpu_non_layer_weights()
            torch.mps.empty_cache()
```

评论区精华

PR 没有留下独立的 review 评论，但提交历史揭示了评审过程中的一次关键追问：

- “在显存有富余的机器上引入额外传输是否有害？”第二个提交 9d88e801 的提交信息直接记录了这个问题：“Review question: does this hurt a machine that has room? It did.”——最初的实现只以 --performance-mode memory 作为开关，而这是用户表达偏好而非实测结果，导致大显存卡在 memory 模式下每个请求都白付两次传输并长期占一份主机拷贝。最终方案引入 PARK_SIGNIFICANCE = 0.1 阈值，用 _device_headroom_bytes()（driver 空闲内存 + 缓存分配器未使用预留）作为实测门控。
- benchmark 数据有效性争议：mickqian 在 Issue 评论中主动更正——#35701 存在 store-aliasing bug（_mapped_cpu_weights 持有参数本身，(1,) 占位符覆盖了它），导致 mapped 层权重被广播重建，denoise 对比数字（141.20 s vs 140.60 s）和 0.35 s 传输估计均无效；而 12 GiB OOM 观察与 decode 数据（150 s vs 23 s）不受影响，因为 video VAE 在该配置下是 pageable 而非 mapped。变更本身与层权重存放位置无关，但 Speed Tests 需要重测。
  - 显存富余的机器上 park 是否造成无谓开销 (performance)：引入 PARK_SIGNIFICANCE = 0.1 阈值：只有当非层权重总字节数超过真实可用显存的 10% 时才 park；否则不仅跳过，还清掉已持有的主机拷贝。真实余量用 driver 空闲内存加缓存分配器未使用预留共同计算。
  - benchmark 数据有效性：store-aliasing bug 的影响范围 (correctness)：变更本身不受 bug 影响（park 的非层权重与层权重存放位置无关），但 Speed Tests 部分需要在 #35813 合入后重新测量。

风险与影响

- 风险：
  1. 主机内存压力：park 需要一份 pageable 主机拷贝，H3 场景约 3.5 GB，与 page cache 竞争主机内存；代码注释也明确承认这一点，32 GiB 上限下实测主机内存不变（

24.5 GiB），但更小的主机内存机器可能受影响。

2. 同步传输阻塞：pageable 拷贝经 driver 自身 pinned staging buffer，传输是同步的（注释中明确说明），每次组件切换都会引入一段不可隐藏的阻塞延迟，0.35 s 的估计因 store-aliasing bug 而失效，需要重测。
3. 显存余量测量依赖平台 API：_device_headroom_bytes() 依赖 get_available_gpu_memory 与 memory_reserved / memory_allocated，不同后端（ROCm、NPU、MPS）语义可能不同；MPS 已显式排除，但其他平台的行为未被测试覆盖。
4. 与 checkpoint-mapping 路径交互：PR body 说明测试是在 checkpoint-mapping 分支（#35813 修复后）上验证的，若该分支未合入或行为变化，park 可能与 mapped 权重路径产生耦合。
5. 测试覆盖局限：7 个单元测试全部基于 monkeypatch 的 fake device 与伪造显存数值，没有真实 GPU 上的集成测试验证 _device_headroom_bytes 在 warm 进程下的准确性。- 影响：用户侧：显存受限（约 8–12 GiB）的显卡上可以跑通 MiniMax-H3 全流程；memory 模式下 VAE 驻留配合 park 机制，decode 从 23 s 进一步降至 13 s，端到端 192 s → 177 s。系统侧：layerwise offload 的组件生命周期增加一次 D2H + H2D 往返，仅在 --performance-mode memory 且余量紧张时发生，其他模式零影响。团队侧：该机制与 courier（#35882）正交，是 layerwise offload 体系从“层级流式”到“非层权重空闲期回收”的补全，为后续 8 GB 卡支持和更高分辨率 decode 铺路。

- 风险标记：内存模式行为变更，主机内存占用增加，显存余量测量依赖平台驱动，测试仅覆盖模拟设备，性能数据部分失效需重测

关联脉络

- PR #35688 Layerwise resident layers support（PR body 引用）：引入 --layerwise-resident-layers video_vae=36，让 VAE 36 个块驻留以跳过重复流式传输；本 PR 的 OOM 场景正是建立在该机制之上。
- PR #35967 fp16 decoder memory shrink（PR body 引用）：fp16-held decoder 将 video_vae=36 从 9.7 GiB 降到 4.9 GiB，使 12 GiB OOM 在 main 上不再复现，本 PR 据此 re-scope 为面向 8 GB 卡与更高分辨率 decode 的机制。
- PR #35882 courier（PR body 引用）：PR body 明确说明本机制与 courier 正交：courier 负责权重搬运通道，park 负责非层权重闲置期回收。
- PR #35813 Mapped CPU weights store-aliasing fix（Issue 评论引用，标题为推断）：修复 #35701 的 store-aliasing bug，是本 PR benchmark 数据有效性的前提；testing 需在该分支上验证。
- PR #35701 Store-aliasing bug source（Issue 评论引用，标题为推断）：存在 store-aliasing bug 的关联 PR，导致本 PR 的 denoise 数据与传输估计失效，需以 #35813 修复后重测。
- PR #36034 [diffusion] UX: clean up startup and offload logs: 同一文件 layerwise_offload.py 的后续改动，迁移 all_gather_single 并清理日志，与本 PR 属于同一内存管理模块的持续演进。