PR #35626 完整报告

sgl-project/sglang

[diffusion] fix: keep large vocab tables in host memory under layerwise offload

合并时间：2026-08-20 15:20

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/35626

执行摘要

- 一句话：大 vocab 表驻留 host，layerwise offload 显存降 48%
- 推荐动作：值得精读。重点看三点：为何由 shape 自动发现改为显式 opt-in（forward hook 覆盖边界是核心教训）；detach/restore 如何包住 .to(device) 迁移窗口；以及 7 个测试如何 pin 契约（declared/undeclared/sharded/round trip/ 等价性）。对做显存卸载或 CPU 驻留优化的同学是很好的模板。

功能与动机

PR body 指出 layerwise offload 假设剩余参数很小（norm、patch embed、time embed），但对文本编码器不成立：umT5-XXL 的 token 表 256384 x 4096 fp32 达 3.91 GiB，在 12 GiB 卡上占三分之一，而每个请求只读取 8 MiB，利用率 0.2%。streaming 逐层搬移反而更差（搬 3.91 GiB 只读 8 MiB），因此表应留在 host，查找在 host 执行、结果过 PCIe。这直接导致 --performance-mode auto 在 12 GiB 卡上不可用：warmup 无余量，每个 probe size 都失败。

实现拆解

1. 入口与选择机制：在 python/sglang/multimodal_gen/runtime/managers/memory_managers/layerwise_offload.py 新增模块级阈值 HOST_RESIDENT_TABLE_MIN_BYTES = 256 MiB，以及 _resolve_submodule、_host_resident_tables 两个辅助函数。_host_resident_tables 遍历 model.modules()，读取各模块声明的 host_resident_table_names 属性，解析点分路径得到表模块，再过滤条件：权重存在且为 2 维、tp_size == 1（分片表输出要喂 all-reduce，必须留在设备）、字节数不低于阈值。
2. detach/restore 核心流程：detach_host_resident_tables 在 post-streaming 的 .to(device) 之前把命中表的 weight.data 替换为 torch.empty(0)，使整模型迁移时跳过这几 GiB 数据；restore_host_resident_tables 在迁移后恢复原权重，并调用 _install_host_gather_hooks 安装 forward hooks。这组函数同时服务 LayerwiseOffloadManager._initialize 与 configure_layerwise_offload 的 multi-group 路径，保证所有卸载入口行为一致。
3. hook 实现：_install_host_gather_hooks 定义 _inputs_to_host（forward pre-hook，把 token id 输入搬到 CPU）和 _output_to_device（forward hook，把 gather 结果以 non_blocking=True 拷回原始 device）。这样每次请求只发生一次 host gather，只有 8 MiB 结果穿越 PCIe，而 3.91 GiB 表保持驻留 host。

4. 模型契约（第二个提交的关键修复）：t5.py 的 T5EncoderModel/UMT5EncoderModel 声明 host_resident_table_names = ["shared"]，qwen3vl.py 的 Qwen3VLTextModel 声明 host_resident_table_names = ["embed_tokens"]。选择 opt-in 而非最初的按 shape 自动发现，是因为 forward hook 只覆盖表自身的 __call__；CI 在 sana_wm 上抓到 Expected all tensors to be on the same device 的真实 bug，证明按 shape 发现不安全，只有表只经 forward 访问的模型才能声明。
5. 测试与验证配套：新增 test_host_resident_vocab_table.py 共 7 个测试，覆盖选择规则（declared 被选、undeclared/ 小表 / 分片表不动、dotted path 解析、缺失路径跳过）和 detach/restore round trip 及 hook gather 与普通 lookup 等价性；已有 layerwise/offload/residency 套件共 136 个测试通过，未破坏既有行为。精度测试在 RTX 3060 12 GiB 上三次运行 SHA-256 完全一致。

关键文件：

- python/sglang/multimodal_gen/runtime/managers/memory_managers/layerwise_offload.py（模块 显存卸载；类别 source；类型 core-logic；符号 _resolve_submodule, _host_resident_tables, detach_host_resident_tables, restore_host_resident_tables）：核心实现文件：新增 host-resident 大表机制（阈值、选择、detach/restore、forward hooks），并在两个调用点集成，是本 PR 的主要逻辑所在。
- python/sglang/multimodal_gen/test/unit/test_host_resident_vocab_table.py（模块 单元测试；类别 test；类型 test-coverage；符号 _Declared, _Undeclared, _Nested, TestSelection）：新增 7 个单测，覆盖选择规则、detach/restore round trip 与 hook gather 等价性，是 host-resident 契约的回归护栏。
- python/sglang/multimodal_gen/runtime/models/encoders/t5.py（模块 T5 编码器；类别 source；类型 data-contract；符号 host_resident_table_names）：T5EncoderModel / UMT5EncoderModel 声明 shared 表为 host-resident，是该机制的首批落地模型；encoder-only、无 tied lm_head 保证表只经 gather 访问。
- python/sglang/multimodal_gen/runtime/models/encoders/qwen3vl.py（模块 VL 编码器；类别 source；类型 data-contract；符号 host_resident_table_names）：Qwen3VLTextModel 声明 embed_tokens；文本模型仅以 self.embed_tokens(input_ids) 使用，无 tied output head，满足契约要求。

关键符号：_resolve_submodule, _host_resident_tables, detach_host_resident_tables, restore_host_resident_tables, _install_host_gather_hooks, _inputs_to_host, _output_to_device

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host-resident大表机制（阈值、选择、detach/restore、forwardhooks），并在两个调用点集成，是本 PR 的主要逻辑所在。

```
# Adapted from skywork AI Infra diffusion optimize
# 低于该阈值的表不值得每次请求做一轮 host 往返；高于它，
```

```
# 表大小相对实际读取行数的比例让驻留设备明显是浪费。
HOST_RESIDENT_TABLE_MIN_BYTES = 256 * 1024**2


def _resolve_submodule(root: torch.nn.Module, path: str) -> torch.nn.Module | None:
    current: Any = root
    for part in path.split("."):
        current = getattr(current, part, None)
        if current is None:
            return None
    return current if isinstance(current, torch.nn.Module) else None


def _host_resident_tables(model: torch.nn.Module) -> List[torch.nn.Module]:
    """收集模型声明过、且大到驻留设备纯属浪费的 vocab 表。

    表是按 gather 而非 GEMM 读取的：每个 token 一行，所以 512-token
    的 prompt 只碰 umT5-XXL 3.91 GiB 表中的 8 MiB。按层流式传输更糟
    ——为了读 8 MiB 要搬 3.91 GiB——因此表应留在 host，查找在 host
    执行。这里按模型显式 opt-in 而不是按 shape 自动发现：bridge 是
    forward hook，只覆盖表自身的 __call__；若模型在 forward 之外还
    直接读 weight（如 tied lm_head、第三方 backbone 里的函数式
    gather），会在图中间看到 host tensor。只有表只经 forward 到达的
    模型才可以声明。
    """
    tables = []
    for module in model.modules():
        for path in getattr(module, "host_resident_table_names", ()) or ():
            table = _resolve_submodule(module, path)
            weight = getattr(table, "weight", None)
            if weight is None or not hasattr(weight, "dim") or weight.dim() != 2:
                continue
            # 分片表已被世界大小均分，且其输出喂给期望设备 tensor 的 all-reduce
            if getattr(table, "tp_size", 1) != 1:
                continue
            if weight.numel() * weight.element_size() < HOST_RESIDENT_TABLE_MIN_BYTES:
                continue
            if table not in tables:
                tables.append(table)
    return tables


def detach_host_resident_tables(model: torch.nn.Module) -> List[Tuple[torch.nn.Module, torch.
Tensor]]:
    """把大 vocab 表换成占位符，让后续 .to(device) 跳过它们。"""
    detached = []
    for module in _host_resident_tables(model):
        weight = module.weight
        detached.append((module, weight.data))
```

```
        weight.data = torch.empty(0, dtype=weight.dtype, device=weight.device)
    return detached


def restore_host_resident_tables(detached: List[Tuple[torch.nn.Module, torch.Tensor]], device) -
> None:
    for module, data in detached:
        module.weight.data = data
        _install_host_gather_hooks(module, device)
        logger.info(
            "Keeping %s (%.2f GiB) in host memory: a gather reads one row per "
            "token, so residency buys almost nothing.",
            type(module).__name__,
            data.numel() * data.element_size() / (1024**3),
        )


def _install_host_gather_hooks(module: torch.nn.Module, device) -> None:
    """让该模块的 gather 在 host 上执行，只搬结果回 device。"""

    def _inputs_to_host(_module, args, kwargs):
        # 只移动张量输入（token id 索引），其余参数保持原样
        if not args or not torch.is_tensor(args[0]):
            return None
        return (args[0].to("cpu"),) + args[1:], kwargs

    def _output_to_device(_module, _args, output):
        if not torch.is_tensor(output):
            return output
        return output.to(device, non_blocking=True)

    # pre-hook 负责把输入索引搬到 host，forward hook 把结果非阻塞搬回设备
    module.register_forward_pre_hook(_inputs_to_host)
    module.register_forward_hook(_output_to_device)
```

python/sglang/multimodal_gen/test/unit/test_host_resident_vocab_table.py

新增 7 个单测，覆盖选择规则、detach/restore round trip 与 hook gather 等价性，是 host-resident 契约的回归护栏。

```
class TestSelection:
    def test_a_declared_table_is_selected(self):
        # 声明过的表且超过阈值时被选中
        model = _Declared()
        with patch(THRESHOLD_PATH, 1024):
            assert _host_resident_tables(model) == [model.embed]

    def test_an_undeclared_table_is_left_alone(self):
        # 回归护栏：第三方 backbone 可能在 forward 之外读 weight，
        # forward hook 覆盖不到，所以未声明的一律不动
```

```
        with patch(THRESHOLD_PATH, 1024):
            assert _host_resident_tables(_Undeclared()) == []

    def test_a_sharded_table_is_left_alone(self):
        # 分片表输出要喂 all-reduce，必须留在设备
        model = _Declared()
        model.embed.tp_size = 2
        with patch(THRESHOLD_PATH, 1024):
            assert _host_resident_tables(model) == []


class TestDetachAndRestore:
    def test_the_weight_survives_a_move_that_skips_it(self):
        model = _Declared()
        original = model.embed.weight.data.clone()
        with patch(THRESHOLD_PATH, 1024):
            detached = detach_host_resident_tables(model)
            assert model.embed.weight.numel() == 0
            model.to("cpu")
            restore_host_resident_tables(detached, "cpu")
        assert torch.equal(model.embed.weight.data, original)

    def test_the_gather_matches_a_plain_lookup(self):
        model = _Declared()
        ids = torch.tensor([[1, 2, 3], [4, 5, 6]])
        expected = torch.nn.functional.embedding(ids, model.embed.weight.data.clone())
        with patch(THRESHOLD_PATH, 1024):
            restore_host_resident_tables(detach_host_resident_tables(model), "cpu")
        assert torch.equal(model.embed(ids), expected)
```

评论区精华

PR 本身没有 reviewer 评论，唯一实质讨论来自作者在 CI 失败后的说明。multimodal-gen-test-1-gpu (3) 抓到一个真实 bug：最初按 shape 自动发现表是错的，forward hook 只覆盖表自身的 __call__，而 sana_wm 经 Diffusers-backed HF Gemma3ForConditionalGeneration 编码（sana_wm/refiner.py:512），权重在 forward 之外被直接读取，于是出现 Expected all tensors to be on the same device, but found at least two devices, cuda:0 and cpu。作者确认该 shard 在 #35335 和 #35538 是绿的、问题由本 PR 引入，并在 dce49f7 改为每模型 opt-in 声明。

- 按 shape 发现 host-resident 表是否安全 (design): 改为每个模型显式声明 host_resident_table_names，opt-in 而非 shape 自动发现；只有表仅经 forward 访问的模型才能声明。

风险与影响

- 风险：

1. 契约假设风险：opt-in 声明要求表只经 forward 访问；若未来某模型声明后又在 forward 之外直接读 weight（如 tied lm_head、第三方 backbone 内的函数式 gather），会再次出现设备不一致。新模型接入时必须审计访问路径。
2. CPU 性能风险：host gather + 8 MiB H2D 拷贝成为每次请求的 CPU 同步点，编码阶段 +0.28%；batch 增大或请求频率上升时可能放大。
3. 编译路径风险：_initialize 目前有 @torch.compiler.disable 保护，forward hooks 在 torch.compile 图模式下的行为未验证。
4. 资源转移：每实例宿主内存增加约 3.91 GiB，PCIe 流量按请求增加 8 MiB，多副本部署需评估 CPU 内存预算。
5. 阈值启发式：256 MiB 是固定模块级常量，对表大小分布不同的模型可能错过优化或产生误判。 - 影响：对用户：12 GiB 小显存卡从 --performance-mode auto 不可用变为可用，峰值显存 -4.0 GiB（-48%），且显存波动更平稳（4420-4500 vs 8000-9180），总请求耗时基本不变甚至略降。对系统：以每请求 8 MiB 的 H2D 拷贝换取 3.91 GiB 显存驻留，是合理的资源交换。对团队：确立了 host-resident 声明式表 + forward hook 的模式，后续 diffusion 模型接入需遵循同一契约；新增 7 个单测并保持既有 136 个相关测试通过，回归护栏清晰。 - 风险标记：opt-in 契约依赖 forward 单一路径，新增 CPU 同步点，torch.compile 下 hook 兼容性未验证，宿主内存增加约 3.91 GiB

关联脉络

- PR #35618 [diffusion] UX: report where a component's weights are: 同一文件 layerwise_offload.py，且同属 diffusion 权重驻留与可观测性演进线。
- PR #35612 [diffusion] fix: keep Cosmos3 T=1 fusion on Blackwell: 同为 diffusion 运行时修复与性能回归治理，体现该模块近期的修复节奏。
- PR #35615 [diffusion] ci: use canonical residency selector: diffusion 测试基建中的 selection 逻辑，与本次选择器语义相关。