PR #35641 完整报告

sgl-project/sglang

[diffusion] feat: plan pinned host memory against the cgroup cap not the machine

合并时间：2026-08-20 19:32

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/35641

执行摘要

- 一句话：按 cgroup 上限规划 pinned 主机内存预算
- 推荐动作：值得精读。该 PR 展示了 " 从容器真实限制出发做资源预算 " 的设计思路，HostPinBudget 的热度排序（字节 × 步数）比简单的 DiT 优先更符合传输量实际，且把回退代价量化进 docstring 和日志，是资源治理的好范例。关注 host_memory_budget.py 的 cgroup 解析与 layerwise_offload.py 的排序注入点。

功能与动机

PR body 指出：offloaded weights 放在 pinned host memory 中，内核既不能 swap 也不能回收，而 pin_cpu_memory 只是一个全局布尔值，运行时在提交前从未查看剩余主机内存。psutil.virtual_memory() 读 /proc/meminfo 只反映宿主机而看不到容器限制，实测一台租用 4x RTX 4090 机器 psutil 报告 2015.7 GiB 而 cgroup 上限为 1117.2 GiB，900 GiB 的虚报正好会把容器推向 OOM-kill。因此必须直接读取 cgroup 的 memory cap 来规划 pin 预算。

实现拆解

1. 新增 python/sglang/multimodal_gen/runtime/managers/memory_managers/host_memory_budget.py：实现 _read_int（安全读取文件并处理 max 关键字）、cgroup_memory_limit_bytes（按 v2 优先、v1 回退读取 (cap, usage)，用 _UNLIMITED_ABOVE 哨兵识别 v1 的近乎 2^63 无限上限）、host_memory_available_bytes（取 psutil.virtual_memory().available 与 max(0, limit - usage) 的较小值）、module_weight_bytes（统计模块参数与 buffer 字节数）与 pin_benefit_bytes（权重字节 × 每请求使用次数）。
2. 核心类 HostPinBudget：构造时计算 available_bytes 与 reserve_bytes = max(5%, 2 GiB)，spendable_bytes 为扣除储备和已承诺后的余量；request(component_name, weight_bytes) 在余量内承诺并返回 True，否则记日志并返回 False。日志与 docstring 明确量化了 pageable 回退的代价（text encoding 阶段 3.04 s -> 6.26 s，约 2x；H2D 带宽 pinned 13.09 GB/s vs pageable 8.84 GB/s，且 copy_(non_blocking=True) 对 pageable 源会静默变同步）。
3. 修改 layerwise_offload.py：configure_layerwise_offload 签名增加 pin_budget 与 component_name 参数，逐 layer 用 pin_budget.request(...) 决定该 manager 是否 pin；新增 _default_num_inference_steps 从 pipeline 配置类读取默认采样步数，新增 _h2d_bytes_a_pin_would_save 计算每个组件的 pin 收益（DiT 按步数多次传输）；

configure_layerwise_offload_modules 每个配置轮次构建一个 budget，并按 pin_benefit_bytes 降序 offer 给各组件，拒绝的组件回退为 pageable。

4. 测试配套：新增 test_host_memory_budget.py（15 个单测，覆盖 v2/v1 路径、无 cgroup、两种 unlimited 写法、cap 与 free 取小、储备及下限、热组件优先耗尽、体积估算去重）；test_layerwise_offload.py 为 _TestServerArgs 补充 pipeline_class_name=None 字段，避免 6 个既有测试因新读取逻辑而失败。

关键文件：

- python/sglang/multimodal_gen/runtime/managers/memory_managers/host_memory_budget.py（模块 内存预算；类别 source；类型 dependency-wiring；符号 _read_int, cgroup_memory_limit_bytes, host_memory_available_bytes, HostPinBudget）：新增的核心模块：从 cgroup v1/v2 读取容器内存上限，结合 psutil 可用内存计算可提交预算，并提供 HostPinBudget 按请求分配 pin 额度。是本次变更的入口和策略载体。
- python/sglang/multimodal_gen/runtime/managers/memory_managers/layerwise_offload.py（模块 层间卸载；类别 source；类型 core-logic；符号 configure_layerwise_offload, _default_num_inference_steps, _h2d_bytes_a_pin_would_save）：修改核心调度逻辑：在 configure_layerwise_offload 中按 pin_budget 逐层决策是否 pin，并在 configure_layerwise_offload_modules 中按 pin_benefit_bytes 排序分配预算。
- python/sglang/multimodal_gen/test/unit/test_host_memory_budget.py（模块 单元测试；类别 test；类型 test-coverage；符号 _point_at, TestCgroupLimit, test_v2_cap_is_read, test_v1_cap_is_read_when_v2_is_absent）：新增 15 个单元测试，覆盖 cgroup v1/v2 解析、无 cgroup、unlimited 哨兵、cap 与 free 取小、储备、预算耗尽与存储去重等关键路径，是验证预算逻辑的主要手段。
- python/sglang/multimodal_gen/test/unit/test_layerwise_offload.py（模块 单元测试；类别 test；类型 test-coverage）：为 _TestServerArgs 补充 pipeline_class_name 字段，修复因新增默认步数读取而破坏的 6 个既有测试。

关键符号：configure_layerwise_offload, _default_num_inference_steps, _h2d_bytes_a_pin_would_save, cgroup_memory_limit_bytes, host_memory_available_bytes, HostPinBudget.request, HostPinBudget.spendable_bytes, module_weight_bytes, pin_benefit_bytes

关键源码片段

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

新增的核心模块：从 cgroup v1/v2 读取容器内存上限，结合 psutil 可用内存计算可提交预算，并提供 HostPinBudget 按请求分配 pin 额度。是本次变更的入口和策略载体。

```
# host_memory_budget.py
# 关键点：pin 过的页内核既不能 swap 也不能回收，
# 所以提交前必须按容器 cgroup 上限而非宿主机总内存规划。

import psutil
```

```
from sglang.multimodal_gen.runtime.utils.logging_utils import init_logger

logger = init_logger(__name__)

GIB_BYTES = 1024**3

# cgroup v2 与 v1 的 limit/usage 文件路径，v2 优先
_CGROUP_V2 = ("/sys/fs/cgroup/memory.max", "/sys/fs/cgroup/memory.current")
_CGROUP_V1 = (
    "/sys/fs/cgroup/memory/memory.limit_in_bytes",
    "/sys/fs/cgroup/memory/memory.usage_in_bytes",
)

# v1 的 " 无限 " 上限用接近 2**63 的哨兵值表示，超过该值视为无 cap
_UNLIMITED_ABOVE = 1 << 62

# 保留 5%（至少 2 GiB）不参与 pin，留给激活、暂存缓冲区等中间分配
HOST_RESERVE_FRACTION = 0.05
MIN_HOST_RESERVE_BYTES = 2 * GIB_BYTES


def _read_int(path: str) -> int | None:
    """读取文件中的整数，文件缺失、"max" 或非法内容都返回 None。"""
    try:
        with open(path) as handle:
            text = handle.read().strip()
    except OSError:
        return None
    if text == "max": # cgroup v2 的 "max" 表示无限制
        return None
    try:
        return int(text)
    except ValueError:
        return None


def cgroup_memory_limit_bytes() -> tuple[int, int] | None:
    """返回 (cap, usage)，无 cgroup 或无限时返回 None。"""
    for limit_path, usage_path in (_CGROUP_V2, _CGROUP_V1):
        limit = _read_int(limit_path)
        # 无限制或哨兵过大时跳过，尝试下一个版本
        if limit is None or limit >= _UNLIMITED_ABOVE:
            continue
        usage = _read_int(usage_path) or 0
        return limit, usage
    return None


def host_memory_available_bytes() -> int:
```

```
    """进程还能安全提交的字节数：宿主机可用与 cgroup 余量的较小值。"""
    available = int(psutil.virtual_memory().available)
    capped = cgroup_memory_limit_bytes()
    if capped is None:
        return available
    limit, usage = capped
    return min(available, max(0, limit - usage))


class HostPinBudget:
    """按热度分配 pin 额度：热组件先问，冷组件在预算耗尽后回退 pageable。"""

    def __init__(self, available_bytes: int | None = None) -> None:
        if available_bytes is None:
            available_bytes = host_memory_available_bytes()
        self.available_bytes = available_bytes
        # 储备取 5% 与 2 GiB 中的较大者，防止运行时后续分配无内存可用
        self.reserve_bytes = max(
            int(available_bytes * HOST_RESERVE_FRACTION), MIN_HOST_RESERVE_BYTES
        )
        self.committed_bytes = 0

    @property
    def spendable_bytes(self) -> int:
        return max(0, self.available_bytes - self.reserve_bytes - self.committed_bytes)

    def request(self, *, component_name: str, weight_bytes: int) -> bool:
        """若 weight_bytes 可被 pin 则承诺并返回 True，否则记日志返回 False。

        对热组件也是硬 cap：超额授予不会减小内存占用，只会把失败推迟到
        pin 分配本身或 swap 阶段。优先级靠"先问"体现，而不是靠超发。
        """
        if weight_bytes <= 0:
            return True
        if weight_bytes <= self.spendable_bytes:
            self.committed_bytes += weight_bytes
            return True
        logger.info(
            "Host pin budget: refusing %s (%d bytes), only %d bytes left",
            component_name,
            weight_bytes,
            self.spendable_bytes,
        )
        return False
```

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

修改核心调度逻辑：在 configure_layerwise_offload 中按 pin_budget 逐层决策是否 pin，并在 configure_layerwise_offload_modules 中按 pin_benefit_bytes 排序分配预算。

```
# layerwise_offload.py 中与 pin 预算相关的改造片段

    def configure_layerwise_offload(
        self,
        server_args: ServerArgs,
        *,
        pin_budget: HostPinBudget | None = None,
        component_name: str | None = None,
    ):
        self.layerwise_offload_managers = []
        named_modules = dict(self.named_modules())
        configured_layer_names = []
        # 既有 DiT 调优旋钮（prefetch/resident）仍只作用于 DiT 组
        dit_tuning_enabled = self.layerwise_offload_dit_group_enabled
        for layer_name in self.layer_names:
            module_list = named_modules.get(layer_name)
            if not isinstance(module_list, (torch.nn.ModuleList, torch.nn.Sequential)):
                continue
            if len(module_list) == 0:
                continue

            num_layers = len(module_list)
            # ... 原有 prefetch_size / resident_layers 计算 ...

            # pin 这些权重是拷贝流超前于计算的前提，但 pinned 页不可回收，
            # 所以只有当预算允许时才 pin；预算会被 request 逐层扣除。
            pin_cpu_memory = server_args.pin_cpu_memory
            if pin_cpu_memory and pin_budget is not None:
                pin_cpu_memory = pin_budget.request(
                    component_name=f"{component_name or type(self).__name__}.{layer_name}",
                    weight_bytes=module_weight_bytes(module_list),
                )
            manager = LayerwiseOffloadManager(
                model=self,
                layers_attr_str=layer_name,
                num_layers=num_layers,
                enabled=True,
                pin_cpu_memory=pin_cpu_memory,
                prefetch_size=prefetch_size,
                resident_layers=resident_layers,
                initialize=False,
            )
            # ... 后续注册 manager ...

    @staticmethod
    def _default_num_inference_steps(server_args) -> int:
```

```
        """读取 pipeline 配置类里的默认推理步数，用于估算每请求传输次数。"""
        from sglang.multimodal_gen.registry import get_pipeline_config_classes

        pipeline_class_name = server_args.pipeline_class_name
        if not pipeline_class_name:
            return 1
        config_classes = get_pipeline_config_classes(pipeline_class_name)
        if config_classes is None:
            return 1
        return max(1, int(config_classes[1]().num_inference_steps))

    def _h2d_bytes_a_pin_would_save(self, name: str) -> int:
        """pin 某组件能省下的传输字节：权重字节 × 每请求使用次数。

        用乘积而非"是否为 DiT"排序，是因为 few-step 模型下 1 GB DiT 步进
        4 次移动 4 GB/请求，而 20 GB 文本编码器一次移动 20 GB，
        只看 DiT 布尔值会把预算给到更需要的一方。
        """
        module = self.modules[name]
        if not isinstance(module, LayerwiseOffloadableModuleMixin):
            return 0
        return pin_benefit_bytes(
            weight_bytes=module_weight_bytes(module),
            uses_per_request=(
                default_steps if module.layerwise_offload_dit_group_enabled else 1
            ),
        )
```

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

新增 15 个单元测试，覆盖 cgroup v1/v2 解析、无 cgroup、unlimited 哨兵、cap 与 free 取小、储备、预算耗尽与存储去重等关键路径，是验证预算逻辑的主要手段。

```
# test_host_memory_budget.py
# 用 monkeypatch 把 cgroup 路径重定向到 tmp_path，避免触碰真实系统文件。

def _point_at(monkeypatch, tmp_path, *, v2=None, v1=None):
    """把 _CGROUP_V2 / _CGROUP_V1 指向临时文件，构造 v2/v1 场景。"""

    def write(name, value):
        path = tmp_path / name
        path.write_text(str(value))
        return str(path)

    missing = str(tmp_path / "absent")
    v2_paths = (
        (write("memory.max", v2[0]), write("memory.current", v2[1]))
        if v2
        else (missing, missing)
    )
```

```
    v1_paths = (
        (write("limit_in_bytes", v1[0]), write("usage_in_bytes", v1[1]))
        if v1
        else (missing, missing)
    )
    monkeypatch.setattr(host_memory_budget, "_CGROUP_V2", v2_paths)
    monkeypatch.setattr(host_memory_budget, "_CGROUP_V1", v1_paths)


class TestCgroupLimit:
    def test_v2_cap_is_read(self, monkeypatch, tmp_path):
        _point_at(monkeypatch, tmp_path, v2=(32 * GIB_BYTES, 4 * GIB_BYTES))
        assert cgroup_memory_limit_bytes() == (32 * GIB_BYTES, 4 * GIB_BYTES)

    def test_v1_cap_is_read_when_v2_is_absent(self, monkeypatch, tmp_path):
        _point_at(monkeypatch, tmp_path, v1=(64 * GIB_BYTES, 8 * GIB_BYTES))
        assert cgroup_memory_limit_bytes() == (64 * GIB_BYTES, 8 * GIB_BYTES)

    def test_no_cgroup_reports_uncapped(self, monkeypatch, tmp_path):
        _point_at(monkeypatch, tmp_path)
        assert cgroup_memory_limit_bytes() is None

    def test_v2_max_keyword_is_uncapped(self, monkeypatch, tmp_path):
        _point_at(monkeypatch, tmp_path, v2=("max", 4 * GIB_BYTES))
        assert cgroup_memory_limit_bytes() is None

    def test_v1_sentinel_is_uncapped(self, monkeypatch, tmp_path):
        # 无限制的 v1 cgroup 报告接近 2**63 的数字而非 "max"
        _point_at(monkeypatch, tmp_path, v1=(2**63 - 4096, 8 * GIB_BYTES))
        assert cgroup_memory_limit_bytes() is None

    def test_the_cap_wins_over_what_the_kernel_reports_free(
        self, monkeypatch, tmp_path
    ):
        # 租用机上实测场景：psutil 看到整台机器，cgroup 是真实上限
        _point_at(monkeypatch, tmp_path, v2=(32 * GIB_BYTES, 8 * GIB_BYTES))
        monkeypatch.setattr(
            host_memory_budget.psutil,
            "virtual_memory",
            lambda: type("VM", (), {"available": 900 * GIB_BYTES})(),
        )
        assert host_memory_available_bytes() == 24 * GIB_BYTES
```

评论区精华

该 PR 无 review 评论线程，提交信息体现了两次设计迭代：第一次提交按 " 是否是 DiT" 分配预算，第二次提交改为按 " 字节 × 步数 "（pin_benefit_bytes）排序，理由是 few-step 模型下小 DiT 步进多次移动 4 GB/ 请求，而 20 GB 文本编码器一次移动 20 GB，布尔判断会反向

分配。第三次提交修复测试桩缺失 pipeline_class_name 字段导致 6 个既有测试失败的问题。

- 暂无高价值评论线程

风险与影响

- 风险：主要风险集中在回退路径：当组件因预算不足改为 pageable 后，copy_(non_blocking=True) 会退化为同步拷贝，实测 text encoding 阶段耗时近 2 倍（3.04 s -> 6.26 s），端到端约慢 23%，对延迟敏感的多租户场景可能放大排队效应。cgroup 解析逻辑对 v1/v2 的路径和哨兵处理依赖内核行为，在系统监管缺失（无 cgroup 挂载或权限受限）时回退为宿主机 free 值，无法真正防御容器超卖；HostPinBudget 为每个配置轮次新建，若配置轮次间共享进程状态则可能重复承诺，当前实现按模块内单次构建规避了该问题。
- 影响：影响 layerwise offload 在容器内运行的 diffusion 服务：之前可能因 pinned 内存无界提交触发 OOM-kill，现在预算在 cgroup cap 内硬性约束；热组件优先 pin，冷组件（如大文本编码器）可能被降级为 pageable，带来 stage 级 2x 延迟但避免进程死亡。对非容器宿主机或内存充裕场景行为不变，属于防御性增强。团队后续可基于该机制暴露 pin 预算与回退日志，进一步做容量规划与告警。
- 风险标记：容器内存上限误判风险，pageable 回退导致同步拷贝变慢，cgroup 解析依赖内核行为，主机内存预算为硬 cap 可能误伤热组件，测试桩依赖新增字段

关联脉络

- PR #35668 placeholders: 同作者在同一 memory_managers 目录下的后续配套（placeholder，占位说明）
- PR #35618 [diffusion] UX: report where a component's weights are: 同为 layerwise offload 组件驻留的可见性改进，本 PR 的日志报告与预算拒绝日志互补
- PR #35626 [diffusion] fix: keep large vocab tables in host memory under layerwise offload: 处理 layerwise offload 下大表驻留内存的策略，与本 PR 的主机内存规划同属内存驻留治理