# PR #49919 完整报告

vllm-project/vllm

[Core] Explicitly manage torch CPU threads in workers

合并时间：2026-08-05 01:06

原文链接：http://prhub.com.cn/vllm-project/vllm/pull/49919

## 执行摘要

- 一句话：显式管理 worker 的 torch 线程数，修复结构化输出解码 2-6x 吞吐回归
- 推荐动作：值得精读。这是一个从用户报障（#49013）到 bisect 定位、根因分析、系统修复、review 竞态再修复的完整范例，对任何做推理引擎或高性能服务的人都有参考价值。重点关注四个设计决策：① 启动期 / 服务期两阶段线程策略；② cgroup v1/v2 配额感知的可用 CPU 计数；③ 用环境变量所有权标记（VLLM_OMP_NUM_THREADS_SET_BY_VLLM）区分「系统设置」与「用户设置」；④ 将线程池创建前移到父进程以规避 dlopen 竞态与 fork 死锁。若在容器内跑 vLLM 或维护多 worker 部署，建议结合 #49013 的 bisect 日志一起阅读，理解 spin-wait 与 CFS 配额如何酿成 2 倍级性能落差。

## 功能与动机

PR body 明确指出：torch 默认 intra-op 线程池按宿主核心数创建，忽略进程亲和性、cgroup CPU 配额和共置 worker 进程；任何超过 parallel_for grain size（32k 元素）的 torch CPU 操作都会扩散到整个线程池，OpenMP worker 在每个并行 region 后自旋等待，偷走引擎循环串行代码的 CPU 周期，在容器中还会烧掉 CFS 配额导致整个进程被限流。这正是 #49013 中 bisect 定位到的回归根因——apply_grammar_bitmask 每步对约 12.6 MB grammar bitmask 的 CPU staging（torch.full + numpy view 写入）触发多线程扇出，且所有复现都是 TP=1，而此前只有 multiproc executor 通过导出 OMP_NUM_THREADS=1 缓解，uniproc、Ray、外部启动器路径均未覆盖。

## 实现拆解

整个方案按「启动期有界并行、服务期单线程、用户设置优先」三原则展开，分四步实现：

1. 工具层新增 CPU 配额感知的线程计算（vllm/utils/torch_utils.py）：新增 _cgroup_cpu_limit() 解析 /proc/self/cgroup，兼容 v1（cpu.cfs_quota_us / cpu.cfs_period_us）与 v2（cpu.max），沿层级向上取最紧配额；available_cpu_count() 用调度亲和性（os.sched_getaffinity）∩ cgroup 配额计算真实可用 CPU 数；startup_omp_num_threads(num_local_procs) 将可用 CPU 数平分给本节点 worker；set_torch_threads_for_runtime() 在服务阶段将线程收束为 1，但对外部设置的 OMP_NUM_THREADS 保留最高优先级，大于 1 时仅记录 warning。
2. 执行器在 fork/spawn 前预设置线程数（vllm/v1/executor/multiproc_executor.py）：set_multiprocessing_worker_envs(local_world_size) 由父进程在创建 worker 前计算每 worker 线程数并写入 OMP_NUM_THREADS，同时写入私有标记

VLLM_OMP_NUM_THREADS_SET_BY_VLLM 以区分「vLLM 设置的」与「用户设置的」；父进程同步调用 torch.set_num_threads() 以便 fork 的 worker 继承。父进程在 wait_for_ready 后也调用 set_torch_threads_for_runtime() 收束为 1——调度进程不需要 intra-op 并行。这是修复第一版 dl-tls 竞态的关键重设计：线程池创建必须发生在 dlopen 共享对象与 fork 之前。

3. Worker 在 warmup 后收束线程（vllm/v1/worker/gpu_worker.py）：在 compile_or_warm_up_model() 末尾（enable_gpu_sync_check() 之后）调用 set_torch_threads_for_runtime()，确保权重加载、kernel warmup、编译等启动期工作享受多线程并行，进入稳态服务后立即收为单线程，避免热路径 CPU 操作触发 spin-wait 与配额烧蚀。
4. 测试配套（tests/utils_/test_torch_utils.py）：新增 restore_torch_threads fixture 与 4 个用例，覆盖启动线程数按本节点 worker 平分、运行时收束为 1、尊重用户 OMP_NUM_THREADS、覆盖 vLLM 自设的 OMP_NUM_THREADS 四条策略分支。

关键文件：

- vllm/utils/torch_utils.py（模块 线程工具；类别 source；类型 core-logic；符号 _cgroup_cpu_limit, visit, read_v2, read_v1）：新增 cgroup 配额解析、可用 CPU 计数、启动期线程平分与服务期收束的全部核心逻辑，是本次变更的策略中枢。
- vllm/v1/executor/multiproc_executor.py（模块 执行器；类别 source；类型 core-logic；符号 set_multiprocessing_worker_envs）：线程数必须在 fork/spawn worker 前由父进程定好并写入环境变量，这里是执行器侧的关键编排逻辑，也是修复 dl-tls 竞态的重设计落点。
- tests/utils_/test_torch_utils.py（模块 单元测试；类别 test；类型 test-coverage；符号 restore_torch_threads, test_startup_omp_num_threads_divides_between_local_workers, test_set_torch_threads_for_runtime, test_runtime_threads_respect_user_omp_num_threads）：新增 4 个测试覆盖线程策略的全部关键分支：启动平分、运行时收束、尊重用户设置、覆盖 vLLM 自设值，是保障后续演进不回归的重要配套。
- vllm/v1/worker/gpu_worker.py（模块 工作进程；类别 source；类型 dependency-wiring）：服务阶段切换的触发点：warmup 完成后调用 set_torch_threads_for_runtime，是「启动期并行、服务期单线程」策略在 worker 侧的落点。

关键符号：_cgroup_cpu_limit, available_cpu_count, startup_omp_num_threads, set_torch_threads_for_runtime, set_multiprocessing_worker_envs

## 关键源码片段

### vllm/utils/torch_utils.py

新增 cgroup 配额解析、可用 CPU 计数、启动期线程平分与服务期收束的全部核心逻辑，是本次变更的策略中枢。

```
# vllm/utils/torch_utils.py 新增的 CPU 配额感知工具
# 核心思想：torch 默认线程数 = 宿主核心数，但容器场景真正可用的是
# 调度亲和性 ∩ cgroup 配额，必须自行解析 /proc/self/cgroup。

def _cgroup_cpu_limit() -> float | None:
    """返回当前进程 cgroup 的有效 CPU 配额，无限制时返回 None。
```

```
同时兼容 cgroup v1（cpu.cfs_quota_us / cpu.cfs_period_us）与 v2
（cpu.max 的 quota/period），并沿 cgroup 层级向上取最紧的限制。
"""
limit: float | None = None
try:
    with open("/proc/self/cgroup") as f:
        entries = [line.strip().split(":", 2) for line in f]

    def visit(base: str, rel_path: str, read_quota) -> None:
        # 从叶子路径逐级向上（去掉最后一段）读取配额，取最小值
        nonlocal limit
        path = rel_path
        while path:
            quota = read_quota(os.path.join(base, path.lstrip("/")))
            if quota is not None:
                limit = quota if limit is None else min(limit, quota)
            path = path.rsplit("/", 1)[0]

    def read_v2(cg_dir: str) -> float | None:
        # v2: cpu.max 形如 "quota period"，quota 为 "max" 表示不限
        try:
            with open(os.path.join(cg_dir, "cpu.max")) as f:
                quota, period = f.read().split()
            return None if quota == "max" else float(quota) / float(period)
        except (OSError, ValueError):
            return None

    def read_v1(cg_dir: str) -> float | None:
        # v1: cfs_quota_us <= 0 表示不限
        try:
            with open(os.path.join(cg_dir, "cpu.cfs_quota_us")) as f:
                quota = int(f.read())
            if quota <= 0:
                return None
            with open(os.path.join(cg_dir, "cpu.cfs_period_us")) as f:
                return quota / int(f.read())
        except (OSError, ValueError):
            return None

    for entry in entries:
        if len(entry) != 3:
            continue
        _, controllers, rel_path = entry
        if controllers == "": # cgroup v2 路径
            visit("/sys/fs/cgroup", rel_path, read_v2)
        elif "cpu" in controllers.split(","): # cgroup v1 的 cpu 控制器
            visit("/sys/fs/cgroup/cpu", rel_path, read_v1)
except OSError:
    pass # 解析失败时按无限制处理，退回保守默认行为
```

```
    return limit


def available_cpu_count() -> int:
    """进程实际可用的 CPU 数：调度亲和性 ∩ cgroup 配额。

    os.cpu_count() 对配额无感知，容器里会高估，导致把线程数摊到
    根本不存在的 CPU 上，放大 OpenMP 自旋等待的浪费。
    """
    if sys.platform != "linux":
        return os.cpu_count() or 1
    count = len(os.sched_getaffinity(0))
    limit = _cgroup_cpu_limit()
    if limit is not None:
        count = min(count, int(limit))
    return max(1, count) # 至少保留 1 个线程，避免除零 / 空集
```

## vllm/v1/executor/multiproc_executor.py

线程数必须在 fork/spawn worker 前由父进程定好并写入环境变量，这里是执行器侧的关键编排逻辑，也是修复 dl-tls 竞态的重设计落点。

```
# vllm/v1/executor/multiproc_executor.py 重写后的 worker 环境准备函数
# 关键教训（第一版被 cjackal 报告的 bug）：
# 1. torch.set_num_threads() 会在调用点立即把线程池建出来（即使是在降低线程数）
# 2. worker 启动过程中（dlopen 共享对象）建线程池会触发 glibc TLS 竞态
#  （_dl_allocate_tls_init 断言失败），fork 的 worker 里还会因
# libgomp 非 fork-safe 而死锁
# 因此必须在父进程 fork / spawn worker 之前把线程数定好并写进环境变量。

def set_multiprocessing_worker_envs(local_world_size: int = 1):
    """在创建 worker 进程前设置它们将继承的线程环境变量"""

    _maybe_force_spawn()

    # CPU 后端走 OMPProcessManager 单独配置亲和性；外部显式设置的
    # OMP_NUM_THREADS 必须尊重，两种情况都直接返回
    if current_platform.is_cpu() or "OMP_NUM_THREADS" in os.environ:
        return

    # 线程数 = 本节点可用 CPU 数 / 本节点 worker 数：
    # 让每个 worker 在加载权重时能并行，但不会各自抢占全部核心
    num_threads = startup_omp_num_threads(local_world_size)
    os.environ["OMP_NUM_THREADS"] = str(num_threads)
    # 用私有标记区分「vLLM 设置的」与「用户设置的」：
    # 前者可在服务阶段被安全降为 1，后者必须保留
    os.environ[OMP_NUM_THREADS_SET_BY_VLLM] = "1"

    # spawn 的 worker 在 import torch 时从环境变量读取；
    # fork 的 worker 则继承本进程状态，所以这里也要 set 一次。
```

```
    # 安全前提：父进程此时还未真正运行任何并行 region
    torch.set_num_threads(num_threads)
    logger.debug(
        "Set OMP_NUM_THREADS=%d for %d worker process(es).",
        num_threads,
        local_world_size,
    )
```

## 评论区精华

Review 中最重要的交锋来自 cjackal 对第一版实现报告的竞态问题：

> cjackal：TP=8、48 vcore（k8s 环境）下，worker 中执行 torch.set_num_threads(6) 时反复出现 _dl_allocate_tls_init: Assertion 'listp != NULL' failed（重复 8 次）。

根因是 torch.set_num_threads() 会在调用点 急切创建线程池（即使是在降低线程数），worker 启动期间正逢 dlopen 共享对象，并发建线程池与 glibc TLS 分配竞争。njhill 随后的 commit 将线程数设置全部前移到父进程（fork/spawn 前写环境变量 + 父进程同步 set），worker 不再自行建池。cjackal 验证新 commit 后服务器可正常启动，但转述同事意见：此类错误常为间歇性，需多环境反复验证——该疑虑未被完全关闭。

此外 body 中给出了完整的性能证据链：Qwen3-235B-A22B-Thinking-2507-FP8（221 GiB，TP=4，144 核）warm cache 下权重加载 37.52 s → 31.60 s（−15.8%），总初始化 81.79 s → 76.06 s（−7.0%）；cold cache 下权重加载 −15.3%、总初始化 −10.4%。并引用 #49168 的验证确认解决了 TP=1 结构化输出运行时性能回归。

- worker 进程内创建 OpenMP 线程池触发 glibc TLS 竞态 (correctness): njhill 重写为在父进程 set_multiprocessing_worker_envs 里预先设置 OMP_NUM_THREADS 并同步 torch.set_num_threads()，worker 不再自行建池；spawn 的 worker 从环境变量读取，fork 的 worker 继承父进程状态。
- dl-tls 竞态是否已被完全排除 (other): 未完全定论；作者表示无法精确复现原错误，修复基于机制分析（急切建池 + dlopen 竞态），跨环境覆盖面有限，值得后续观察。

## 风险与影响

- 风险：具体风险如下：

1. glibc TLS 竞态未被完全排除（vllm/v1/executor/multiproc_executor.py）：重设计规避了 worker 内建池，但 cjackal 明确提示该错误有间歇性，跨环境验证仍有限；若未来有代码在 set_multiprocessing_worker_envs 之前触发并行 torch op，fork 场景下 libgomp 非 fork-safe 的死锁风险会重现。
2. 服务期强制单线程的隐性假设（vllm/utils/torch_utils.py 的 set_torch_threads_for_runtime）：假设稳态服务期没有值得并行的 CPU 算子，若未来引入大张量 CPU 运算（如新采样器、新 grammar 后端），单线程会直接成为瓶颈。外部 OMP_NUM_THREADS>1 时仅警告不强制的策略，意味着用户覆写后性能问题仍可能出现。
3. cgroup 解析的降级路径（_cgroup_cpu_limit）：非 Linux 平台回退 os.cpu_count()；Linux 下 cgroup 文件缺失或格式异常时静默返回 None，线程数可能仍高估，配额烧蚀风险未根除；int(limit) 截断配额小数可能低估可用 CPU。

4. fork 安全依赖隐含约定：set_multiprocessing_worker_envs 中 torch.set_num_threads(num_threads) 的安全性建立在「父进程此前未运行并行 region」之上，这一约束没有显式机制保证，属于易被未来改动破坏的脆弱点。
5. 覆盖范围不完全：CPU 后端（current_platform.is_cpu()）与用户已设 OMP_NUM_THREADS 的路径直接 return，这些场景的线程治理仍维持原状。 - 影响：影响范围：

- 用户侧：所有 vLLM 部署受益。TP=1 单进程、Ray、外部启动器路径首次获得防护，容器化部署不再因 OpenMP spin-wait 烧毁 CFS 配额；结构化输出（xgrammar guided-JSON）decode 吞吐恢复至回归前水平（#49013 实测从 ~121 req/min 回到 ~233 req/min 基线）。
- 启动性能：multiproc worker 权重加载从单线程变为有界并行，Qwen3-235B 测例权重加载 -15.8%、总初始化 -7.0%～-10.4%，大模型冷启动收益显著。
- 系统行为变更：服务期 torch 线程统一为 1，外部 OMP_NUM_THREADS 的语义从「被 vLLM 覆写」变为「用户优先、vLLM 警告」；新增环境变量 VLLM_OMP_NUM_THREADS_SET_BY_VLLM 成为内部契约。
- 团队侧：确立了「启动期有界并行 + 服务期单线程 + 用户设置优先」的线程治理模式，后续新增 worker 或新部署路径时有了可复用的工具函数与警告机制。
- 风险标记：OpenMP 线程池与 dlopen 竞态（验证有限），服务期强制单线程的隐性假设，fork 安全依赖父进程不提前使用线程池，cgroup 解析失败静默降级，外部 OMP_NUM_THREADS>1 仅警告不干预

## 关联脉络

- PR #45424 [Core] Ensure memory is pinned prior to async h2d copy: issue #49013 中 bisect 定位的回归引入 PR，本 PR 修复了其 apply_grammar_bitmask staging 重写触发的性能回退。
- PR #49013 [Perf] ~2x decode throughput regression for structured outputs since #45424: 本 PR body 明确 Fixes #49013，是其根因修复的直接关联 issue；复现场景全部为 TP=1，正是此前 uniproc 路径未被防护的空白地带。
- PR #49168 Confirmed fix of TP=1 structured output runtime regression: PR body 引用该 PR 的 comment 确认修复生效，是验证本 PR 效果的关键证据链。
- PR #50879 [Misc] Avoid importing nixl_ep on every vllm serve config: 同方向的启动路径治理（惰性导入减少启动开销与副作用），与线程管理共同改善进程启动与运行时行为。