

PR #7770 完整报告

verl-project/verl

[training_utils] feat: Add automatic CUDA/NPU OOM snapshots and multi-step memory history

合并时间: 2026-09-08 14:30

原文链接: <http://prhub.com.cn/verl-project/verl/pull/7770>

执行摘要

- 一句话: 为 torch_memory 分析器增加 CUDA/NPU OOM 自动快照和多步内存历史窗口。
- 推荐动作: 该 PR 是一个高质量的增强功能, 解决了一个常见的痛点。代码结构清晰, 异常处理完善, 测试覆盖全面。建议相关开发者 (尤其是进行大规模训练调优的团队) 精读其设计, 特别是 OOM 回调的实现和多步窗口的配置使用。

功能与动机

解决在训练过程中, OOM 发生时无法获取事发现场内存状态的痛点。PR body 中提到: 'helps diagnose OOMs that occur before the next scheduled snapshot'。同时, 引入 `memory_snapshot_num_steps` 配置项, 以支持分析跨步的内存增长模式, 满足更精细的性能调优需求。

实现拆解

1. OOM 观察者注册与回调: 在 `verl/utils/profiler/torch_memory_profile.py` 中, 为 `TorchMemoryProfiler` 新增类属性 `_oom_observer_attached` 和实例方法 `_attach_oom_observer`。在 `__init__` 中, 若当前 rank 被选中, 则自动尝试注册一个进程本地的 OOM 观察者回调 `_on_out_of_memory`。该回调负责记录 OOM 详情 (分配大小、设备总内存、空闲内存)、打印 Python 调用栈, 并在不同步 CUDA 的情况下转储快照。
2. 多步内存历史窗口: 在 `TorchMemoryToolConfig` 中新增 `memory_snapshot_num_steps` 配置项 (默认值 1, 保持向后兼容)。在 `TorchMemoryProfiler` 中引入 `_steps_in_window`、`_window_start_step`、`_window_end_step` 三个状态变量, 以及辅助方法 `_window_sub_dir`。start 和 stop 方法被修改以跟踪窗口进度, 仅在累计步数达到配置值后才执行快照转储和历史重置。
3. 底层快照转储适配: 在 `verl/utils/memory_utils.py` 的 `dump_memory_snapshot` 方法中, 新增 `synchronize` 参数以支持 OOM 回调中的非同步转储。同时, 针对 NPU 设备, 改用 `device.memory._dump_snapshot` 而非序列化原始快照, 以避免启动 / 停止分析器带来的额外开销。
4. 配置与文档同步更新: 在多个 PPO 训练器配置文件 (`ppo_trainer.yaml`, `_generated_*.yaml`) 和角色配置文件 (`actor.yaml`, `critic.yaml` 等) 中, 为 `torch_memory` 工具添加 `memory_snapshot_num_steps` 字段。同步更新了 `docs/perf/verl_profiler_system.md` 文档。

5. 全面测试覆盖：在 tests/utils/test_torch_memory_profile.py 中新增

TestTorchMemoryProfiler 和 TestNpuTorchMemoryProfiler 测试类，覆盖 OOM 观察者自动注册、跨 rank 逻辑、进程单例注册、设备可用性检查、API 缺失处理、多步窗口行为以及诊断失败容错等关键场景。

关键文件：

- verl/utils/profiler/torch_memory_profile.py（模块 内存分析器；类别 source；类型 core-logic；符号 _attach_oom_observer, _on_out_of_memory, _window_sub_dir）：核心实现文件，包含 OOM 观察者注册、回调逻辑和多步内存窗口管理。
- verl/utils/memory_utils.py（模块 内存工具；类别 source；类型 core-logic；符号 dump_memory_snapshot）：底层快照转储逻辑的适配层，为 OOM 回调提供非同步转储和 NPU 兼容性支持。
- tests/utils/test_torch_memory_profile.py（模块 内存分析器测试；类别 test；类型 test-coverage；符号 TestTorchMemoryProfiler, test_oom_observer_is_automatic_and_dumps_without_sync, test_snapshot_window_keeps_history_until_the_configured_step_count）：全面的单元测试文件，确保 OOM 观察者和多步窗口功能的正确性和健壮性。

关键符号：_attach_oom_observer, _on_out_of_memory, _window_sub_dir, dump_memory_snapshot

关键源码片段

verl/utils/profiler/torch_memory_profile.py

核心实现文件，包含 OOM 观察者注册、回调逻辑和多步内存窗口管理。

```
def _attach_oom_observer(self) -> None:
    """Register one process-local callback that writes a snapshot at allocator OOM time."""
    if TorchMemoryProfiler._oom_observer_attached:
        return

    # 限制仅在 CUDA/NPU 设备上注册 OOM 观察者
    if self._device_name not in ("cuda", "npu"):
        logger.warning("[torch_memory] automatic OOM snapshots are only available on CUDA/
        NPU devices")
        return

    try:
        if not get_torch_device().is_available():
            logger.warning("[torch_memory] automatic OOM snapshots require an available
            accelerator")
            return

        # 根据设备类型选择对应的后端模块（CUDA 或 NPU）
        if self._device_name == "npu":
            import torch_npu
            backend = torch_npu._C
```

```

else:
    backend = torch._C
    # 动态查找 attach_observer 函数, 例如 torch._C._cuda_attach_out_of_memory_observer
    attach_observer = getattr(backend, f"_{self._device_name}_attach_out_of_memory_
observer", None)
    if attach_observer is None:
        logger.warning("[torch_memory] this build does not support %s OOM observers", self._
device_name.upper())
        return

    # 注册回调, 确保整个进程只注册一次
    attach_observer(self._on_out_of_memory)
    TorchMemoryProfiler._oom_observer_attached = True
    logger.info("[torch_memory] %s OOM snapshot observer attached", self._device_name.
upper())
except Exception as exc:
    logger.warning(
        "[torch_memory] failed to attach %s OOM snapshot observer: %s", self._device_name.
upper(), exc
    )

def _on_out_of_memory(self, device: int, alloc: int, device_total: int, device_free: int) -> None:
    """Best-effort snapshot callback invoked by the CUDA/NPU allocator."""
    out_dir = self.config.save_path or "outputs/profile"
    sub_dir = f"oom_{datetime.now().strftime('%Y%m%d-%H%M%S-%f')}"
    # NPU 可能报告的是进程内存限制, 而不仅仅是物理设备总量。
    total_label = "total_or_limit" if self._device_name == "npu" else "total"
    logger.error(
        "[torch_memory] %s OOM on device %s: requested=%s %s=%s free=%s; dumping allocator
snapshot",
        self._device_name.upper(), device, alloc, total_label, device_total, device_free,
    )
    logger.error("[torch_memory] Python stack at OOM:\n%s", "".join(traceback.format_stack()))
    try:
        # 在 OOM 回调中收集额外的内存分配器信息
        logger.error("[torch_memory] allocator memory at OOM: %s", get_memory_info())
    except Exception as exc:
        logger.warning(f"[torch_memory] failed to collect allocator memory at OOM: {exc}")
    try:
        # 关键: OOM 时可能无法同步设备, 因此设置 synchronize=False
        self.sampler.dump_memory_snapshot(
            out_dir=out_dir, tag="torch_memory_oom", sub_dir=sub_dir, synchronize=False
        )
    except Exception as exc:
        logger.warning("[torch_memory] failed to dump %s OOM snapshot: %s", self._device_name.
upper(), exc)

```

verl/utils/memory_utils.py

底层快照转储逻辑的适配层，为 OOM 回调提供非同步转储和 NPU 兼容性支持。

```
def dump_memory_snapshot(
    self, out_dir: str = "./mem_snapshots", tag: str = "snapshot", sub_dir: str = None,
    synchronize: bool = True
) -> Path | None:
    """
    Generates a memory snapshot and saves it as a pickle file.

    Args:
        out_dir: Directory for snapshot files.
        tag: Filename prefix.
        sub_dir: Subdirectory within out_dir.
        synchronize: Whether to sync the device before dumping. Disable for OOM observer.
    """
    # ... ( 路径计算逻辑不变 ) ...

    device = get_torch_device()
    if not device.is_available():
        logger.warning("[memory_visualize] is only available on accelerator devices.")
        return None
    try:
        if synchronize:
            device.synchronize()
        # NPU 需要特殊处理：使用 pickle 序列化，避免调用可能触发分析器的 _dump_snapshot
        if get_device_name() == "npu":
            snapshot = device.memory._snapshot()
            with path.open("wb") as f:
                pickle.dump(snapshot, f)
        else:
            device.memory._dump_snapshot(str(path))
        logger.info(f"[memory_visualize] dumped: {path}")
        return path
    except Exception as e:
        logger.warning(f"[memory_visualize] dump failed: {e}")
        return None
```

tests/utils/test_torch_memory_profile.py

全面的单元测试文件，确保 OOM 观察者和多步窗口功能的正确性和健壮性。

```
class TestTorchMemoryProfiler(unittest.TestCase):
    # ... (setUp 方法) ...

    def test_oom_observer_is_automatic_and_dumps_without_sync(self):
        """验证 OOM 观察者自动注册且转储时不同步设备。"""
        profiler = TorchMemoryProfiler(rank=0, config=self._config(), tool_config=
            TorchMemoryToolConfig())

        self.attach_observer.assert_called_once() # 确认观察者已注册
        observer = self.attach_observer.call_args.args[0] # 获取注册的回调函数
```

```

with (
    patch("verl.utils.profiler.torch_memory_profile.traceback.format_stack", return_value=
        ["stack"]),
    patch("verl.utils.profiler.torch_memory_profile.get_memory_info", return_value=
        {"allocated": 1234}),
    patch.object(profiler.sampler, "dump_memory_snapshot") as dump_snapshot,
):
    # 模拟 OOM 回调被调用
    with self.assertLogs("verl.utils.profiler.torch_memory_profile", level="ERROR"):
        observer(0, 4096, 1234, 5678)

    dump_snapshot.assert_called_once()
    kwargs = dump_snapshot.call_args.kwargs
    self.assertEqual(kwargs["tag"], "torch_memory_oom") # 确认快照标签
    self.assertFalse(kwargs["synchronize"]) # 关键断言：转储时不应同步设备

def test_snapshot_window_keeps_history_until_the_configured_step_count(self):
    """验证多步内存历史窗口在达到配置步数前不转储。"""
    tool_config = TorchMemoryToolConfig(memory_snapshot_num_steps=2) # 配置为 2
    步一个窗口
    with patch("verl.utils.profiler.torch_memory_profile.clear_memory_history"):
        profiler = TorchMemoryProfiler(rank=0, config=self._config(), tool_config=tool_config)
        with patch.object(profiler.sampler, "dump_memory_snapshot") as dump_snapshot:
            # 第一步：不应触发转储
            profiler.start(profile_step=4)
            profiler.stop()
            dump_snapshot.assert_not_called()

            # 第二步：达到窗口大小，应触发转储
            profiler.start(profile_step=5)
            profiler.stop()
            dump_snapshot.assert_called_once_with(
                out_dir="/tmp/profiles", tag="torch_memory", sub_dir="steps4-5" #
                窗口目录命名正确
            )

```

评论区精华

PR 仅有一条 APPROVED 审核记录，无公开讨论。设计决策（如 OOM 回调中禁用同步、NPU 使用 pickle 序列化）均通过代码和 PR body 中的 'Limitations' 部分进行说明。

- 暂无高价值评论线程

风险与影响

- 风险：

- 1.OOM回调稳定性：OOM发生时设备流可能处于错误状态，转储操作本身存在失败风险。代码通过多层 try...except 并记录警告日志来降低风险，但 synchronize=False 可能导致写入不完整的快照文件。

2. 性能开销：注册 OOM 观察者本身开销极低。多步窗口通过延迟转储减少了 I/O 频率，性能影响可忽略。
3. 兼容性：memory_snapshot_num_steps 默认值为 1，完全向后兼容。新功能仅在选择 torch_memory 工具时激活，对其他训练流程无影响。
4. 测试覆盖：测试全面覆盖了主要功能路径和异常路径，降低了回归风险。

- 影响：

1. 用户体验：显著降低了调试 CUDA/NPU OOM 问题的难度，提供了前所未有的事后分析能力。
2. 系统健壮性：增强了系统在内存压力下的可观测性，有助于提前发现和修复潜在的内存问题。
3. 开发者工作流：为性能工程师和训练工程师提供了更强大的内存分析工具链，促进了针对大规模训练任务的优化。影响范围主要限于 verl/utils/profiler 和 verl/utils/memory_utils 模块，以及使用 torch_memory 工具的配置。 - 风险标记：OOM 回调容错依赖，设备流错误状态处理

关联脉络

- PR #7722 [perf, tool] fix: Avoid overlapping rollout and training profilers in Trainer V1: 同属性能分析工具链（profiler）的改进，关注点在避免采集冲突，而本 PR 关注 OOM 诊断和历史窗口，共同提升了分析器的可靠性。
- PR #7724 [veomni] feat: add async_activation_offload config: 同样关注内存优化，但侧重点不同（激活卸载 vs OOM 诊断），共同服务于降低训练内存占用的长期目标。