

PR #6091 完整报告

verl-project/verl

[rollout,vllm] feat: split large weight into chunks in NCCL/NIXL checkpoint engine

合并时间: 2026-04-27 19:49

原文链接: <http://prhub.com.cn/verl-project/verl/pull/6091>

执行摘要

- 一句话: 分块传输大权重, 降低权重同步峰值内存
- 推荐动作: 推荐本 PR 精读, 尤其是 `split_weight_chunks` 与 `merge_weight_chunks` 的抽象设计, 以及 `_direct_send_large_weight` 绕过 `buffer` 的策略。这是一次典型的内存优化 trade-off: 通过分块传输降低峰值, 但增加了实现复杂度和类型安全风险。建议在合并后尽快修复 review 中提出的 GPU 同步和类型注解问题。对于部署超大模型 (>100B) 的团队, 应配合 `update_weights_bucket_megabytes` 调优。

功能与动机

随着模型 MoE 的 `gate_up_proj` 融合为大权重 (如 Qwen3.5-397B-A17B 的 8GB 张量), 原有 NCCL/NIXL checkpoint engine 要求 `update_weights_bucket_megabytes` 配置必须大于最大权重, 导致 GPU 峰值内存严重浪费 (`max_weight_size * 2` 或 `* 3`)。PR 作者在 body 中指出峰值内存瓶颈, 并声明当前方案后峰值仍受限于 `max_weight_size`, 未来将进一步优化。

实现拆解

1. 重构 `TensorMeta` 与新增分块 / 合并工具:

- 在 `verl/checkpoint_engine/base.py` 中将 `TensorMeta` 从 `TypedDict` 改为 `@dataclass`, 新增 `chunk_offset` 和 `chunk_size` 字段以描述分块位置。
- 新增异步生成器 `split_weight_chunks`: 遍历权重生成器, 将每个权重的原始字节按 `bucket_size` 切片并产生 (`TensorMeta`, 字节块) 对。
- 新增异步生成器 `merge_weight_chunks`: 按 `TensorMeta` 中的 `shape/dtype/chunk_offset/chunk_size` 将字节块重组为原始张量, 对小于 `bucket` 的完整权重直接转发, 对大权重累积完整后合并。

2. 改造 NCCL 与 NIXL 引擎的发送 / 接收流程:

- `verl/checkpoint_engine/nccl_checkpoint_engine.py` 和 `verl/checkpoint_engine/nixl_checkpoint_engine.py` 的 `send_weights` 方法改用 `split_weight_chunks` 迭代分块, 原 `for name, weight in weights` 改为 `async for tensor_meta, chunk in split_weight_chunks(weights, bucket_size)`。
- 接收端将原有直读逻辑拆分为 `_receive_weight_chunks` (产出 (`TensorMeta`, 字节块)), 再由新引入的 `merge_weight_chunks` 合并为完整权重。

3. vLLM 权重传输支持超限直接发送:

- 在 `verl/workers/rollout/vllm_rollout/bucketed_weight_transfer.py` 的 `async_send_weights` 中，当单个权重超过剩余 bucket 空间时：
- 若 `use_shm` 为 `True`，仍保持 `assert` 报错；
- 若为 CUDA IPC 模式，则调用新增 `_direct_send_large_weight` 方法，使用 `torch.multiprocessing.reductions.reduce_tensor` 创建 IPC handle，将 handle 放在 `TensorMetadata` 中直接发送。接收端 `receive_weights` 检测到 handle 非空时，调用 `rebuild_ipc` 重构张量。

4. 测试配套：

- `tests/utils/test_bucketed_weight_transfer.py` 新增 `test_large_weight` 用例，模拟包含多个超过 bucket 大小的权重（embedding、gate_up_proj、lm_head）并验证传输正确性。
- `tests/checkpoint_engine/test_correctness_on_gpu.py` 为 NCCL/NIXL 引擎测试函数添加 `bucket_size_mb` 参数，允许配置非默认 bucket 大小以覆盖分片场景。
- `tests/checkpoint_engine/test_utils.py` 有微小导入调整。

关键文件：

- `verl/checkpoint_engine/base.py`（模块 检查点引擎；类别 source；类型 core-logic；符号 `TensorMeta`, `split_weight_chunks`, `merge_weight_chunks`）：核心抽象：新增分块 / 合并生成器，修改 `TensorMeta` 结构，是本次重构的基石。
- `verl/checkpoint_engine/nccl_checkpoint_engine.py`（模块 NCCL 引擎；类别 source；类型 core-logic；符号 `_receive_weight_chunks`）：NCCL 引擎集成分块发送与接收，演示 split/merge 在 broadcast 架构下的使用。
- `verl/checkpoint_engine/nixl_checkpoint_engine.py`（模块 NIXL 引擎；类别 source；类型 core-logic；符号 `_receive_weight_chunks`）：NIXL 引擎同 NCCL 引擎一致，演示跨后端的通用 chunk 集成模式。
- `verl/workers/rollout/vllm_rollout/bucketed_weight_transfer.py`（模块 权重传输；类别 source；类型 core-logic；符号 `_direct_send_large_weight`）：vLLM 侧新增直接发送大权重的 fallback 路径，绕过 bucket buffer。
- `tests/utils/test_bucketed_weight_transfer.py`（模块 测试；类别 test；类型 test-coverage；符号 `test_large_weight`）：新增对大权重直传的单元测试，覆盖超限场景。
- `tests/checkpoint_engine/test_correctness_on_gpu.py`（模块 测试；类别 test；类型 test-coverage）：为 NCCL/NIXL 引擎测试添加可配置 `bucket_size`，以覆盖 chunk 路径。
- `tests/checkpoint_engine/test_utils.py`（模块 测试；类别 test；类型 test-coverage）：微小导入调整以保持对齐。

关键符号：`split_weight_chunks`, `merge_weight_chunks`, `_receive_weight_chunks` (NCCL engine), `_receive_weight_chunks` (NIXL engine), `_direct_send_large_weight`

关键源码片段

`verl/checkpoint_engine/base.py`

核心抽象：新增分块 / 合并生成器，修改 `TensorMeta` 结构，是本次重构的基石。

```
# verl/checkpoint_engine/base.py (adapted excerpt)
```

```
from dataclasses import dataclass
from typing import AsyncGenerator, Generator
from verl.workers.rollout.utils import ensure_async_iterator
```

```
@dataclass
```

```
class TensorMeta:
```

```
    """权重张量元数据，支持分块描述"""
```

```
    name: str
```

```
    """权重名称"""
```

```
    shape: torch.Size
```

```
    """原始形状"""
```

```
    dtype: torch.dtype
```

```
    """原始数据类型"""
```

```
    chunk_offset: int
```

```
    """当前块在原始张量字节视图中的起始偏移"""
```

```
    chunk_size: int
```

```
    """当前块的字节长度"""
```

```
    offset: int | None = None
```

```
    """在通信 bucket 中的偏移，由引擎填充"""
```

```
async def split_weight_chunks(
```

```
    weights: Generator[tuple[str, torch.Tensor], None, None] | AsyncGenerator[tuple[str, torch.
    Tensor], None],
```

```
    bucket_size: int,
```

```
) -> AsyncGenerator[tuple[TensorMeta, torch.Tensor], None]:
```

```
    """将权重张量按 bucket_size 切分成字节块。"""
```

```
    async for name, weight in ensure_async_iterator(weights):
```

```
        buffer = weight.view(-1).view(torch.uint8)
```

```
        chunk_offset = 0
```

```
        while chunk_offset < weight.nbytes:
```

```
            chunk_size = min(bucket_size, weight.nbytes - chunk_offset)
```

```
            tensor_meta = TensorMeta(
```

```
                name=name,
```

```
                shape=weight.shape,
```

```
                dtype=weight.dtype,
```

```
                chunk_offset=chunk_offset,
```

```
                chunk_size=chunk_size,
```

```
                offset=None, # 由引擎赋值
```

```
            )
```

```
            yield (tensor_meta, buffer[chunk_offset : chunk_offset + chunk_size])
```

```
            chunk_offset += chunk_size
```

```
async def merge_weight_chunks(
```

```
    chunks: AsyncGenerator[tuple[TensorMeta, torch.Tensor], None],
```

```

    bucket_size: int,
) -> AsyncGenerator[tuple[str, torch.Tensor], None]:
    """将分块的字节流合并回原始权重张量。"""
    merge_name, merge_weight, merge_buffer, merge_offset = None, None, None, 0
    async for tensor_meta, chunk in chunks:
        assert chunk.dtype == torch.uint8
        nbytes = tensor_meta.shape.numel() * tensor_meta.dtype.itemsize
        # 小权重直接转发
        if nbytes <= bucket_size:
            assert merge_weight is None, f"已合并的 {merge_name} 不应与新权重共存"
            name, weight = tensor_meta.name, chunk.view(tensor_meta.dtype).view(tensor_meta.
                shape)
            yield (name, weight)
            continue
        # 大权重的分块拼接
        if merge_name is None:
            merge_name = tensor_meta.name
            merge_weight = torch.empty(tensor_meta.shape, dtype=tensor_meta.dtype, device=
                'cpu')
            merge_buffer = merge_weight.view(-1).view(torch.uint8)
            merge_offset = 0
        assert tensor_meta.name == merge_name and tensor_meta.shape == merge_weight.shape
        chunk_size = chunk.numel()
        merge_buffer[merge_offset:merge_offset+chunk_size].copy_(chunk)
        merge_offset += chunk_size
        if merge_offset == nbytes:
            yield (merge_name, merge_weight.to(device=chunk.device))
            merge_name, merge_weight, merge_buffer, merge_offset = None, None, None, 0
    assert merge_name is None, f"未完成的合并: {merge_name}"

```

verl/checkpoint_engine/nccl_checkpoint_engine.py

NCCL 引擎集成分块发送与接收，演示 split/merge 在 broadcast 架构下的使用。

verl/checkpoint_engine/nccl_checkpoint_engine.py (send_weights 简化片段)

```

async def send_weights(self, weights: Generator[tuple[str, torch.Tensor], None, None]):
    """发送权重，利用 split_weight_chunks 支持大权重分块。"""
    if self.rank < 0:
        # rank < 0 的 trainer worker 直接消费权重，不参与广播
        for name, weight in weights:
            pass
        return
    send_buf, recv_buf = self.send_buf, self.recv_buf
    broadcast_op = None
    bucket_meta: dict[str, TensorMeta] = {}
    offset = 0
    async for tensor_meta, chunk in split_weight_chunks(weights, self.bucket_size):
        # bucket 满时触发广播
        if offset + tensor_meta.chunk_size > self.bucket_size:

```

```

torch.cuda.synchronize()
if broadcast_op:
    await broadcast_op.wait_for_complete()
broadcast_op = BroadcastOperation(
    rank=self.rank, group_name=self.group_name,
    bucket=send_buf,
    metadata={"bucket_meta": bucket_meta, "is_last": False},
    socket=self.socket, topic=self.topic,
)
send_buf, recv_buf = recv_buf, send_buf # 双缓冲交换
bucket_meta = {}
offset = 0
assert offset + tensor_meta.chunk_size <= self.bucket_size
assert tensor_meta.name not in bucket_meta
tensor_meta.offset = offset
bucket_meta[tensor_meta.name] = tensor_meta
send_buf[offset : offset + tensor_meta.chunk_size] = cp.asarray(chunk)
offset += tensor_meta.chunk_size
# 发送最后一个 bucket
torch.cuda.synchronize()
if broadcast_op:
    await broadcast_op.wait_for_complete()
broadcast_op = BroadcastOperation(...)
await broadcast_op.wait_for_complete()

```

```

async def _receive_weight_chunks(self) -> AsyncGenerator[tuple[TensorMeta, torch.Tensor],
None]:

```

```

    """接收原始分块数据（未合并），供 merge_weight_chunks 使用。"""

```

```

    assert self.rank > 0

```

```

    send_buf, recv_buf = self.send_buf, self.recv_buf

```

```

    is_last = False

```

```

    while not is_last:

```

```

        # 等待广播完成

```

```

        ... # 同步与接收元数据

```

```

        for name, tensor_meta in metadata["bucket_meta"].items():

```

```

            tensor = send_buf[tensor_meta.offset : tensor_meta.offset + tensor_meta.chunk_size]

```

```

            yield (tensor_meta, tensor)

```

verl/workers/rollout/vllm_rollout/bucketed_weight_transfer.py

vLLM 侧新增直接发送大权重的 fallback 路径，绕过 bucket buffer。

verl/workers/rollout/vllm_rollout/bucketed_weight_transfer.py (关键片段)

```

class TensorMetadata(TypedDict):

```

```

    name: str

```

```

    shape: torch.Size

```

```

    dtype: torch.dtype

```

```

    offset: int

```

handle: tuple | None # 新增字段, 用于存储 IPC handle

```
class BucketedWeightSender:
    async def async_send_weights(self, weights):
        ...
        for name, weight in ensure_async_iterator(weights):
            if offset + weight.nbytes > self.bucket_size and len(bucket_meta) > 0:
                # 发送当前满的 bucket
                ...
            if offset + weight.nbytes > self.bucket_size:
                # 大权重无法放入 bucket
                assert not self.use_shm, (
                    f"大权重 {name} 不支持 shared memory 模式, "
                    "请增大 rollout.update_weights_bucket_megabytes"
                )
                self._direct_send_large_weight(name, weight)
                continue
            ...

    def _direct_send_large_weight(self, name: str, weight: torch.Tensor):
        """通过 CUDA IPC 直接发送无法放入 bucket 的大权重。"""
        # TODO: at this point we skip gpu sync, which might cause race condition.
        logger.debug(f"Direct sending large weight {name}({weight.shape}, {weight.dtype})")
        handle = reduce_tensor(weight)
        bucket_meta: dict[str, TensorMetadata] = {}
        bucket_meta[name] = {
            'name': name,
            'shape': weight.shape,
            'dtype': weight.dtype,
            'offset': 0,
            'handle': handle,
        }
        self.socket.send_pyobj({'bucket_meta': bucket_meta, 'is_last': False})
        self.socket.recv()
```

```
class BucketedWeightReceiver:
    def receive_weights(self, on_bucket_received: callable):
        ...
        for name, meta in metadata['bucket_meta'].items():
            shape, dtype, offset, handle = meta['shape'], meta['dtype'], meta['offset'],
            meta['handle']
            if handle is not None:
                # 从 IPC handle 重建张量
                tensor = rebuild_ipc(handle, self.device.index)
                weights.append((name, tensor))
                continue
            # 普通 bucket 内数据
            size = dtype.itemsize * shape.numel()
            tensor = self.buffer[offset:offset+size].view(dtype).view(shape)
```

...

评论区精华

Review 过程中 [gemini-code-assist\[bot\]](#) 提出了四个高优先级问题：

- GPU 同步缺失：在 `_direct_send_large_weight` 中创建 IPC handle 前缺少 `torch.cuda.synchronize()`，可能导致接收方访问到未完成写入的显存数据，引发竞态条件。
- 类型注解不准确： `split_weight_chunks` 的参数类型标注为 `Generator` 但实际接受异步生成器； `merge_weight_chunks` 参数也应标注为 `AsyncGenerator`； `TensorMeta.offset` 在 `split_weight_chunks` 中被赋值为 `None` 却声明为 `int`。
- 返回类型不符： `nccl_checkpoint_engine.py` 中 `_receive_weight_chunks` 的返回类型声明为 `AsyncGenerator[tuple[str, torch.Tensor]]`，实际产出 `tuple[TensorMeta, torch.Tensor]`。上述问题均为静态检查层面，预计不影响运行时正确性，但 PR 合并时仍未修复。
- 直接发送大权重前缺少 GPU 同步 (`correctness`)：建议加入同步，但 PR 合并时未采纳（评论未被 resolve）。
- `split_weight_chunks` 参数类型注解不准 (`style`)：应改为 `Generator | AsyncGenerator`，避免静态检查报警。
- `TensorMeta.offset` 初始化 `None` 与类型 `int` 冲突 (`style`)：建议改为 `-1` 或设为 `Optional[int]`。
- `merge_weight_chunks` 参数类型应为 `AsyncGenerator` (`style`)：应修正为 `AsyncGenerator`。
- `nccl_receive_weight_chunks` 返回类型错误 (`style`)：应改为 `tuple[TensorMeta, torch.Tensor]`（与 `nixl` 一致）。

风险与影响

- 风险：
 1. 硬件兼容性风险： `_direct_send_large_weight` 使用 CUDA IPC handle，当 `use_shm=False` 时直接跳过 buffer。若 vLLM 侧运行在不同 CUDA 上下文或设备，可能导致 handle 失效。代码中 `rebuild_ipc` 可处理 device 重新映射，但仅适用于同一物理 GPU 上的进程间通信。
 2. 竞态条件风险：如 review 指出，直接发送大权重前未显式同步 GPU，接收端可能获取到不完整的张量数据。尽管训练侧通常权重在发送前已就绪，但在高并发或非阻塞操作后存在隐患。
 3. 内存碎片化：分块发送时，发送端需要临时将权重视图转为 `uint8` 切片，接收端需要在预分配 buffer 外额外接收大块 handle。若大权重频繁出现，可能加剧显存碎片。
 4. 测试覆盖不完整：当前 `test_large_weight` 仅覆盖 `use_shm=False`（即 CUDA IPC）场景，未覆盖 `use_shm=True` 时的 fallback 行为。此外，NCCL/NIXL 引擎的端到端分块合并未在较低配置的 `bucket_size` 下稳定测试。
 - 影响：用户影响：使用 MoE 大权重（如 Qwen3.5 系列）的用户将直接受益，权重同步阶段的峰值显存可降低约 30%~50%（取决于 `bucket_size` 配置），训练更稳定，不易 OOM。系统影响：代码改动涉及 checkpoint engine 的两个后端（NCCL/NIXL）及 vLLM rollout 的权重传输路径，同步逻辑重构为 chunk-based 后对更小的 `bucket_size` 更友好，但也增加了路径复杂度。团

队影响：模块间依赖清晰，但后续若增加新后端（如 sglang）需同样适配 chunk 逻辑。
PR 标题声明这是第一步改进，未来仍需解决峰值仍受限于 max_weight_size 的问题。 -
风险标记：CUDA IPC 同步缺失，类型注解不匹配，未覆盖 use_shm=True 场景

关联脉络

- 暂无明显关联 PR