PR #29156 完整报告

sgl-project/sglang

Fix bounded checkpoint prefetching and buffered drop-cache handling

合并时间：2026-06-30 05:49

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/29156

执行摘要

- 一句话：修复 checkpoint prefetch 并发控制和 drop-cache 逻辑
- 推荐动作：本 PR 展示了如何将异步并发逻辑改造成有界线程池并编写可测试的代码，适合精读。尤其是 _InlineThread/_InlineExecutor 的 mock 模式值得借鉴。设计决策方面，选择 concurrent.futures 而非 asyncio 以简化依赖，并确保与 weight_utils 其余部分线程模型一致。

功能与动机

PR 清理了 checkpoint prefetching，通过使用有界线程池和拒绝无效线程数来防止任务挂死；同时修复了缓冲 safetensors 加载器中 drop_cache_after_load 在每个分片后实际运行的问题，并添加了测试覆盖 prefetch 和 drop-cache 路径。

实现拆解

1. 替换异步 prefetch 为有界线程池：在 _prefetch_all_checkpoints 中删除了 asyncio 和 async/await，改用 concurrent.futures.ThreadPoolExecutor。通过 itertools.islice 初始提交 num_threads 个任务，然后每次等待任意任务完成时补充一个新任务，确保同时运行的任务数不超过 num_threads。
2. 添加线程数校验：在函数开头检查 num_threads < 1 并抛出 ValueError，防止无效配置导致挂起。
3. 修复 drop_cache_after_load 执行：在 buffered_multi_thread_safetensors_weights_iterator 中确保 drop_cache_after_load 在每个 safetensors 分片加载后立即调用，而不是只在所有分片加载后调用。
4. 测试覆盖：新增测试文件 test_prefetch_checkpoints.py，包含 4 个测试用例：权重一致性、无效线程数拒绝、有界窗口验证、失败日志记录。使用 _InlineThread 和 _InlineExecutor 辅助类模拟并发，避免真实线程开销。

关键文件：

- python/sglang/srt/model_loader/weight_utils.py（模块 模型加载；类别 source；类型 core-logic；符号 _prefetch_all, record_complete, _prefetch_all_checkpoints）：核心源码变更：将异步 prefetch 改为有界线程池，添加线程数验证，修复 drop_cache_after_load 执行

- test/registered/unit/model_loader/test_prefetch_checkpoints.py（模块 模型加载；类别 test；类型 test-coverage；符号 TestPrefetchCheckpoints, test_weights_match_with_and_without_prefetch, test_prefetch_rejects_invalid_thread_count, test_prefetch_keeps_bounded_pending_window）：新增测试覆盖 prefetch 的边界情况：拒绝无效线程数、保持有界窗口、记录失败

关键符号：_prefetch_all_checkpoints, _prefetch_all, record_complete, test_weights_match_with_and_without_prefetch, test_prefetch_rejects_invalid_thread_count, test_prefetch_keeps_bounded_pending_window, test_prefetch_logs_failed_futures, _InlineThread.init, _InlineThread.start, _InlineExecutor.init, _InlineExecutor.enter, _InlineExecutor.exit, _InlineExecutor.submit

关键源码片段

python/sglang/srt/model_loader/weight_utils.py

核心源码变更：将异步 prefetch 改为有界线程池，添加线程数验证，修复 drop_cache_after_load 执行

以下展示 _prefetch_all_checkpoints 中替换后的核心逻辑：

```
def _prefetch_all_checkpoints(
    sorted_files: List[str],
    num_threads: int = 4,
) -> None:
    """..."""
    # 拒绝无效线程数，避免后续 `max_workers` 参数错误导致任务挂死
    if num_threads < 1:
        raise ValueError("weight loader prefetch num_threads must be >= 1")

    # 节点本地 rank 分割文件列表，避免跨节点重复 prefetch
    # page cache 仅在同一节点内共享
    ...

    def _prefetch_all() -> None:
        completed = 0
        next_log_pct = 10

        def record_complete() -> None:
            nonlocal completed, next_log_pct
            completed += 1
            if total_for_rank > 0 and next_log_pct <= 100:
                pct = 100 * completed / total_for_rank
                # 逐段打印日志，避免跳过多级百分比
                while pct >= next_log_pct and next_log_pct <= 100:
                    logger.info(
                        "Rank %d: prefetching checkpoint files: %d%% (%d/%d)",
```

```
                    local_rank, next_log_pct, completed, total_for_rank,
                )
                next_log_pct += 10

        with concurrent.futures.ThreadPoolExecutor(
            max_workers=num_threads
        ) as executor:
            file_iter = iter(my_files)
            # `pending` 字典用于追踪尚未完成的任务及其对应的文件路径
            pending: Dict[concurrent.futures.Future, str] = {}
            # 初始填充：向线程池提交 `num_threads` 个任务，填满有界窗口
            for path in itertools.islice(file_iter, num_threads):
                pending[executor.submit(_prefetch_checkpoint_file, path)] = path

            while pending:
                # 等待任意一个任务完成
                done, _ = concurrent.futures.wait(
                    pending,
                    return_when=concurrent.futures.FIRST_COMPLETED,
                )
                for future in done:
                    path = pending.pop(future)
                    try:
                        future.result() # 重新抛出 prefetch 中的异常
                    except Exception:
                        logger.warning(
                            "Failed to prefetch checkpoint file %r.",
                            path, exc_info=True,
                        )
                    finally:
                        record_complete()
                # 窗口腾出空间后立即提交下一个未处理文件
                next_path = next(file_iter, None)
                if next_path is not None:
                    pending[executor.submit(_prefetch_checkpoint_file, next_path)] = next_path

    def _run_prefetch() -> None:
        start = time.perf_counter()
        _prefetch_all()
        elapsed = time.perf_counter() - start
        logger.info("Rank %d: prefetching checkpoint files into page cache took %.3f seconds",
                    local_rank, elapsed)

    prefetch_thread = threading.Thread(target=_run_prefetch, daemon=True)
    prefetch_thread.start()
    return prefetch_thread
```

test/registered/unit/model_loader/test_prefetch_checkpoints.py

新增测试覆盖 prefetch 的边界情况：拒绝无效线程数、保持有界窗口、记录失败

以下展示 _InlineThread 和 _InlineExecutor 辅助类以及一个关键测试用例的简化实现：

```
# 辅助类：将 `threading.Thread` 替换为在当前线程内同步执行的模仿实现
class _InlineThread:
    def __init__(self, target, daemon=None):
        self.target = target
        self.daemon = daemon

    def start(self):
        # 立即在当前线程调用 target，避免真实并发
        self.target()


# 辅助类：将 `ThreadPoolExecutor` 替换为同步执行的伪执行器
class _InlineExecutor:
    def __init__(self, max_workers):
        self.max_workers = max_workers

    def __enter__(self):
        return self

    def __exit__(self, exc_type, exc, tb):
        return False

    def submit(self, fn, *args, **kwargs):
        # 直接执行函数并返回已完成 Future
        future = Future()
        try:
            future.set_result(fn(*args, **kwargs))
        except Exception as exc:
            future.set_exception(exc)
        return future


class TestPrefetchCheckpoints(unittest.TestCase):

    @patch("torch.distributed.is_initialized", return_value=False)
    def test_prefetch_keeps_bounded_pending_window(self, _):
        """验证 concurrent.futures.wait 调用时 pending 集合大小不超过 num_threads"""
        paths = [f"model-{i:05d}.safetensors" for i in range(20)]
        pending_sizes = []
        submitted_paths = []

        class RecordingExecutor(_InlineExecutor):
            def submit(self, fn, path):
                submitted_paths.append(path)
                return super().submit(fn, path)
```

```
    def record_pending_size(fs, return_when):
        pending_sizes.append(len(fs))
        return set(fs), set() # 返回 done 和 not_done 空集

    with (
        patch("threading.Thread", _InlineThread),
        patch("concurrent.futures.ThreadPoolExecutor", RecordingExecutor),
        patch("concurrent.futures.wait", side_effect=record_pending_size),
        patch("sglang.srt.model_loader.weight_utils._prefetch_checkpoint_file"),
    ):
        _prefetch_all_checkpoints(paths, num_threads=4)

    self.assertEqual(submitted_paths, paths)
    # 所有 wait 调用中 pending 集合尺寸均不超过 4
    self.assertLessEqual(max(pending_sizes), 4)
```

评论区精华

PR 作者 mmangkad 请求 review，b8zhong 触发 /rerun-test 命令运行测试，结果全部通过。b8zhong 随后批准了 PR。Gemini Code Assist 的自动代码审查总结了变更但未提出具体问题。

- 测试重跑以确认 prefetch 单元测试通过 (testing): 测试通过，无需额外修改。

风险与影响

- 风险：
  1. 并发模型变更风险：从 asyncio 切换到 ThreadPoolExecutor 改变了 prefetch 的并发语义，可能影响分布式环境下的 page cache 预热效率，但测试验证了权重一致性和有界窗口。
  2. drop_cache_after_load 影响：修复后每个 shard 加载后立即释放 page cache，可能增加后续 shard 读取的 I/O 压力，但整体内存开销更可控。
  3. 线程数与资源：新增的线程数校验避免了负值或零值导致的死锁，但用户配置错误时会立即报错而非降级。
  4. 测试覆盖：单元测试使用了 mock 避免真实 I/O，但未覆盖分布式多 rank 的真实交互场景。 - 影响：直接影响模型加载阶段的 page cache 使用和 I/O 性能：prefetch 并发更可控，drop-cache 按 shard 释放内存，降低大模型加载时的内存峰值。对用户透明，无需修改配置。影响范围为所有使用 checkpoint prefetch 的模型加载场景（主要影响大规模模型如 DeepSeek 等）。 - 风险标记：核心路径变更，并发模型变更，测试覆盖良好

关联脉络

- 暂无明显关联 PR