PR #24397 完整报告

sgl-project/sglang

[diffusion] chore: clean CUDA cache only at explicit release points

合并时间：2026-05-05 22:30

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/24397

执行摘要

- 一句话：明确 CUDA cache 清理时机，移除 stage 边界隐式清理
- 推荐动作：推荐阅读，尤其是以下设计决策：
  - 将隐式副作用（debug 日志中的 empty_cache）显式化，分离观测和清理职责，是良好的工程实践。
  - 只在 memory_intensive 组件释放且满足特定条件（CUDA 存储释放或 LayerwiseOffloadStrategy）时清理，避免频繁 empty_cache。
  - 性能基线的同步更新策略以及为有波动的 stage 设置动态容差的方法值得借鉴。

功能与动机

旧 debug 内存日志在每次 stage 边界隐式调用 empty_cache()，导致 debug 日志改变请求延迟和 stage 计时；完全移除后又可能在请求失败或大组件释放后留下内存压力。本 PR 将 cache 清理移动到明确的点：请求失败 /OOM 处理，以及组件释放（_empty_cache_after_large_release）时。

实现拆解

1. 组件管理器 (component_manager.py)：新增 _module_on_cuda 和 _empty_cache_after_large_release 方法，在 _finish_use 和 finish_request 中调用。仅对 memory_intensive 组件且在释放 CUDA 存储或使用 LayerwiseOffloadStrategy 时执行 empty_cache()，替代原先每个 stage 边界隐式清理。
2. 性能日志 (perf_logger.py)：修改 StageProfiler.__enter__ 中的 debug 内存日志，调用 current_platform.get_available_gpu_memory(empty_cache=False)，保持观察性而不触发清理。
3. GPU 工作器 (gpu_worker.py)：在异常捕获片段中，若为 OOM 异常，添加 torch.cuda.empty_cache()，确保失败后即时释放。
4. 性能基线 (perf_baselines.json)：收紧 joyai_image_edit_ti2i 和 qwen_image_edit_2511_ti2i 等模型的多个 stage 期望值（如 ImageEncodingStage 从 948ms 降至 740ms，ImageVAEEncodingStage 从 96ms 降至 84ms 等）。
5. 测试工具 (test_server_utils.py)：在 stage 验证逻辑中，对 DecodingStage 增加 90% 相对容差和 250ms 最小绝对容差，以容忍因异步 work 归属导致的波动；其他 stage 保持原 120ms 绝对容差。

关键文件：

- python/sglang/multimodal_gen/runtime/managers/component_manager.py（模块 组件管理器；类别 source；类型 core-logic；符号 _module_on_cuda, _empty_cache_after_large_release）：核心变更，新增 _module_on_cuda 和 _empty_cache_after_large_release 方法，控制 cache 清理时机。
- python/sglang/multimodal_gen/runtime/utils/perf_logger.py（模块 性能日志；类别 source；类型 core-logic）：修改 debug 内存日志，不再隐式调用 empty_cache，保持观测性。
- python/sglang/multimodal_gen/runtime/managers/gpu_worker.py（模块 GPU 工作器；类别 source；类型 core-logic）：在 OOM 异常处理后添加 empty_cache，确保失败后内存释放。
- python/sglang/multimodal_gen/test/server/perf_baselines.json（模块 性能基线；类别 test；类型 test-coverage）：更新性能基线以匹配新的 stage 计时。
- python/sglang/multimodal_gen/test/server/test_server_utils.py（模块 测试工具；类别 test；类型 test-coverage）：调整 DecodingStage 的测试容差，避免因异步 work 归属导致 flaky。

关键符号：_module_on_cuda, _empty_cache_after_large_release, StageProfiler.enter, _execute_forward_common (OOM 处理)

关键源码片段

python/sglang/multimodal_gen/runtime/managers/component_manager.py

核心变更，新增 _module_on_cuda 和 _empty_cache_after_large_release 方法，控制 cache 清理时机。

```
# 在 ComponentResidencyManager 类中

def _module_on_cuda(self, module: nn.Module | None) -> bool:
    """返回模块是否在 CUDA 上"""
    return self._module_device(module) == "cuda"

def _empty_cache_after_large_release(
    self,
    use: ComponentUse,
    strategy: ComponentResidencyStrategy,
    module: nn.Module,
    was_on_cuda: bool,
) -> None:
    """在可能释放大组件后显式清理 CUDA 缓存"""
    if not use.memory_intensive: # 仅针对标记为 memory intensive 的组件
        return
    # 检查是否从 CUDA 移出（释放了 CUDA 存储）
    released_cuda_storage = was_on_cuda and not self._module_on_cuda(module)
    # 或者是 LayerwiseOffloadStrategy（逐层卸载）
    released_layerwise_storage = isinstance(strategy, LayerwiseOffloadStrategy)
```

```
        if not (released_cuda_storage or released_layerwise_storage):
            return
        if not torch.get_device_module().is_available():
            return
        torch.get_device_module().empty_cache()
        self._trace("empty_cache", use, strategy, module, detail="after_release")

    def _finish_use(self, use, *, module=None, keep_on_warmup=False):
        module = module or self.get_module(use.component_name)
        if module is None:
            self._trace("skip_missing", use)
            return
        should_keep = (keep_on_warmup and self.state.batch_is_warmup) \
                    or self._should_keep_after_use(use)
        if should_keep:
            self._trace("keep", use, self.strategy_for(use.component_name, module), module)
            return
        strategy = self.strategy_for(use.component_name, module)
        self._trace("finish", use, strategy, module)
        # 新增：记录当前模块是否在 CUDA 上
        was_on_cuda = self._module_on_cuda(module)
        strategy.finish_use(module, use, self.state)
        # 新增：在可能释放大组件后清理 cache
        self._empty_cache_after_large_release(use, strategy, module, was_on_cuda)
```

python/sglang/multimodal_gen/runtime/utils/perf_logger.py

修改 debug 内存日志，不再隐式调用 empty_cache，保持观测性。

```
    def __enter__(self):
        if self.log_stage_start_end:
            msg = f"[{self.stage_name}] started..."
            if self.logger.isEnabledFor(logging.DEBUG):
                # 保持观测性：获取可用内存但不隐式调用 empty_cache
                available_memory = current_platform.get_available_gpu_memory(
                    empty_cache=False
                )
                msg += f" ({round(available_memory, 2)} GB left)"
            self.logger.info(msg)
        # ... 其余部分
```

评论区精华

本 PR 无 review 评论，由作者直接合并。

- 暂无高价值评论线程

风险与影响

- 风险：

  - 若 memory_intensive 标记不准确，某些大组件释放后可能不会触发 cache 清理，但 OOM 路径始终会清理，降低了风险。
  - 放宽 DecodingStage 容差（90% + 250ms）可能掩盖该 stage 的真实性能回归，需关注 CI 长期趋势。
  - 收紧的性能基线在硬件或驱动变化时可能误报，需定期维护。
  - 整体风险较低，变更经过 CI 验证（NVIDIA CI job 25357009336 和 74363515606 均通过）。
- 影响：
  - 用户：成功请求不再支付 per-stage empty_cache 开销，延迟降低（如 qwen_image_edit_2511_ti2i E2E 从 23525ms 降至 23405ms，joyai_image_edit_ti2i ImageEncodingStage 从 948ms 降至 740ms）。
  - 系统：GPU 内存清理时机从“每个 stage”变为“失败或大组件释放后”，减少不必要的同步开销。
  - 团队：需要同步本地测试基线；CI 中解码阶段容差放宽降低 flaky，但需注意不掩盖回归。
  - 风险标记：内存回收时机变更，测试容差放宽，性能基线收紧，核心路径变更

关联脉络

- 暂无明显关联 PR