PR #23009 完整报告

sgl-project/sglang

Remove deprecated double sparsity feature

合并时间：2026-04-18 04:33

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/23009

执行摘要

- 一句话：移除已弃用的双稀疏注意力优化特性，清理代码库和相关文档。
- 推荐动作：建议开发者浏览此 PR 以学习如何安全移除大型特性，重点关注跨文件引用清理和文档更新模式；对于涉及类似弃用代码清理的项目，可参考此次实现步骤。

功能与动机

PR body 中明确指出“Remove the deprecated double sparsity attention optimization feature entirely”，目的是清理不再维护的代码，降低维护复杂性并消除潜在的技术债务。

实现拆解

1. 删除核心实现文件：移除 python/sglang/srt/layers/attention/double_sparsity_backend.py 中的 DoubleSparseAttnBackend 类及其方法（如 init_forward_metadata、forward_extend），删除 python/sglang/srt/layers/attention/triton_ops/double_sparsity_attention.py 中的 Triton 内核函数（如 flash_decode_attention_fwd），原因是这些代码已弃用且不再使用。
2. 清理内存池类：从 python/sglang/srt/mem_cache/memory_pool.py 中移除 DoubleSparseTokenToKVPool 类及其方法（如 get_key_buffer、set_kv_buffer），因为该内存池专用于双稀疏特性，移除后简化内存管理逻辑。
3. 移除服务器参数和配置逻辑：在 python/sglang/srt/server_args.py 中删除双稀疏相关命令行参数（如 --enable-double-sparsity），在 python/sglang/srt/model_executor/model_runner.py 中移除 init_double_sparsity_channel_config 方法和相关初始化代码，确保模型启动时不再依赖这些配置。
4. 更新文档和测试：删除 test/manual/test_double_sparsity.py 测试文件，移除 docs/advanced_features/server_arguments.md 和 docs/platforms/ascend/ascend_npu_support_features.md 中的双稀疏相关描述，保持文档与代码同步。

关键文件：

- python/sglang/srt/layers/attention/double_sparsity_backend.py（模块 注意力后端；类别 source；类型 deletion；符号 DoubleSparseAttnBackend, init, init_forward_metadata, forward_extend）：核心后端实现文件，移除 DoubleSparseAttnBackend 类，这是双稀疏特性的主要逻辑入口。

- python/sglang/srt/mem_cache/memory_pool.py（模块 内存管理；类别 source；类型 core-logic；符号 DoubleSparseTokenToKVPool, init, get_key_buffer, get_value_buffer）：关键内存池文件，移除 DoubleSparseTokenToKVPool 类，影响 KV 缓存管理。
- python/sglang/srt/model_executor/model_runner.py（模块 模型执行器；类别 source；类型 data-contract；符号 init_double_sparsity_channel_config）：模型运行器文件，移除双稀疏初始化逻辑和配置方法，影响模型启动流程。
- test/manual/test_double_sparsity.py（模块 测试套件；类别 test；类型 deletion；符号 TestDoubleSparsity, setUpClass, tearDownClass, test_mmlu）：专用测试文件，移除双稀疏功能的 MMLU 测试用例，确保测试套件不再包含已弃用特性。

关键符号：DoubleSparseAttnBackend.init, DoubleSparseAttnBackend.init_forward_metadata, DoubleSparseAttnBackend.forward_extend, DoubleSparseAttnBackend.forward_decode, DoubleSparseTokenToKVPool.init, DoubleSparseTokenToKVPool.get_key_buffer, DoubleSparseTokenToKVPool.set_kv_buffer, init_double_sparsity_channel_config

关键源码片段

python/sglang/srt/mem_cache/memory_pool.py

关键内存池文件，移除 DoubleSparseTokenToKVPool 类，影响 KV 缓存管理。

```
from typing import List, Optional
import torch
from torch import nn
from sglang.srt.layers.radix_attention import RadixAttention

class DoubleSparseTokenToKVPool(KVCache):
    def __init__(
        self,
        size: int,
        page_size: int,
        dtype: torch.dtype,
        head_num: int,
        head_dim: int,
        layer_num: int,
        device: str,
        heavy_channel_num: int,
        enable_memory_saver: bool,
        start_layer: Optional[int] = None,
        end_layer: Optional[int] = None,
    ):
        super().__init__(size, page_size, dtype, layer_num, device, enable_memory_saver, start_
        layer, end_layer)
        with self.memory_saver_adapter.region(GPU_MEMORY_TYPE_KV_CACHE):
            # 在自定义内存池或默认上下文中分配缓冲区
            with (
                torch.cuda.use_mem_pool(self.custom_mem_pool)
```

```
            if self.enable_custom_mem_pool
            else nullcontext()
        ):
            # 为每层分配 key 缓冲区，形状为 [大小+页大小, 头数, 头维度]
            self.k_buffer = [
                torch.zeros((size + page_size, head_num, head_dim), dtype=dtype, device=device)
                for _ in range(layer_num)
            ]
            # 为每层分配 value 缓冲区
            self.v_buffer = [
                torch.zeros((size + page_size, head_num, head_dim), dtype=dtype, device=device)
                for _ in range(layer_num)
            ]
            # 为每层分配标签缓冲区，用于存储重通道信息，形状为 [大小+1, 头数, 重通道数]
            self.label_buffer = [
                torch.zeros((size + 1, head_num, heavy_channel_num), dtype=dtype, device=
                device)
                for _ in range(layer_num)
            ]

    def get_key_buffer(self, layer_id: int):
        """根据层 ID 返回对应的 key 缓冲区张量。"""
        return self.k_buffer[layer_id - self.start_layer]

    def get_value_buffer(self, layer_id: int):
        """根据层 ID 返回对应的 value 缓冲区张量。"""
        return self.v_buffer[layer_id - self.start_layer]

    def get_label_buffer(self, layer_id: int):
        """根据层 ID 返回对应的标签缓冲区张量。"""
        return self.label_buffer[layer_id - self.start_layer]

    def get_kv_buffer(self, layer_id: int):
        """返回指定层的 key 和 value 缓冲区元组。"""
        return (self.k_buffer[layer_id - self.start_layer], self.v_buffer[layer_id - self.start_layer])

    def set_kv_buffer(self, layer: RadixAttention, loc: torch.Tensor, cache_k: torch.Tensor, cache_
    v: torch.Tensor, cache_label: torch.Tensor):
        """将 KV 缓存和标签数据设置到缓冲区的指定位置。"""
        layer_id = layer.layer_id
        self.k_buffer[layer_id - self.start_layer][loc] = cache_k
        self.v_buffer[layer_id - self.start_layer][loc] = cache_v
        self.label_buffer[layer_id - self.start_layer][loc] = cache_label
```

评论区精华

PR 中没有 review 评论或讨论，变更由作者直接提交并合并，表明移除操作已达成共识或无争议。

- 暂无高价值评论线程

风险与影响

- 风险：回归风险：低，因为特性已标记为弃用，其他代码应不再依赖；但需确保所有引用已清理，避免导入错误。兼容性风险：用户若仍在使用相关服务器参数（如 *--enable-double-sparsity*）将遇到错误，需更新配置。性能风险：无，移除未使用代码不会影响系统性能。
- 影响：对用户：使用双稀疏特性的用户需移除相关配置，否则服务启动失败。对系统：代码库更简洁，减少维护开销和潜在 bug。对团队：简化代码结构，便于新开发者理解，但需确保 CI 测试覆盖移除后的场景。
- 风险标记：已弃用代码移除，兼容性影响，缺少测试覆盖

关联脉络

- 暂无明显关联 PR