PR #23106 完整报告

sgl-project/sglang

[Perf] Make EAGLE bigram key an O(1) view on `RadixKey`

合并时间：2026-04-21 03:01

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/23106

执行摘要

- 一句话：将 EAGLE bigram key 从 O(N) 元组物化改为 O(1) 视图，显著提升缓存插入和匹配性能。
- 推荐动作：值得精读，特别是 RadixKey 类的设计展示了如何通过视图模式避免物化开销，是性能优化和数据结构设计的典型案例。建议关注 __getitem__ 切片逻辑和 maybe_to_bigram_view 的 O(1) 实现。

功能与动机

根据 PR body，EAGLE radix keys 传统上存储为 List[Tuple[int, int]]，通过 convert_to_bigram_key 生成，每次插入和匹配都需分配 N-1 个 Python 元组，导致在 1M 上下文时占用约 138ms（其中 insert 约 84ms、match_prefix 约 54ms），成为 cache_unfinished_req 等热点路径的性能瓶颈。

实现拆解

1. 重构 RadixKey 类（python/sglang/srt/mem_cache/radix_cache.py）：
   - 添加 __slots__ 优化内存，引入 is_bigram 标志区分视图模式。
   - 修改 __len__：bigram 模式下长度计算为 max(0, len(token_ids) - 1)。
   - 重写 __iter__：bigram 模式下迭代返回 (t[i], t[i+1]) 元组，普通模式直接迭代 token_ids。
   - 调整 __getitem__：支持 bigram 切片时共享边界 token，逻辑为 token_ids[start:stop+1]。
   - 新增 maybe_to_bigram_view 方法：O(1) 切换 is_bigram 标志，替代旧的 maybe_bigram_convert。
2. 更新页面对齐函数（同文件）：
   - page_align_keys 新增 is_bigram 参数，bigram 模式时保留额外边界 token 以确保对齐逻辑长度。
3. 调整调用站点（python/sglang/srt/mem_cache/unified_radix_cache.py 等）：
   - 移除 maybe_bigram_convert 导入和调用，改用 key.maybe_to_bigram_view。
   - 在 cache_finished_req 和 cache_unfinished_req 中，直接传递原始 token_ids 给 page_align_keys，通过 is_bigram 标志控制语义。
4. 测试配套（test/registered/unit/mem_cache/test_swa_unittest.py）：

- 更新测试以验证 bigram 视图行为，例如检查 list(last_node.key) 返回元组列表而非直接访问 token_ids。

5. 哈希兼容性维护：
   - compute_node_hash_values 等函数直接处理原始 token_ids 的字节对，确保向后兼容的哈希流。

关键文件：

- python/sglang/srt/mem_cache/radix_cache.py（模块 缓存核心；类别 source；类型 core-logic；符号 RadixKey.init, RadixKey.len, RadixKey.iter, RadixKey.getitem）：核心数据结构变更，重新设计 RadixKey 以支持 bigram 视图，是性能优化的基础。
- python/sglang/srt/mem_cache/unified_radix_cache.py（模块 统一缓存；类别 source；类型 core-logic；符号 match_prefix, insert, cache_finished_req, cache_unfinished_req）：统一缓存调用站点调整，移除旧转换函数，使用新视图方法。
- python/sglang/srt/mem_cache/swa_radix_cache.py（模块 SWA 缓存；类别 source；类型 core-logic；符号 insert, cache_finished_req, cache_unfinished_req）：SWA 缓存调用站点优化，跳过元组物化路径，直接使用 bigram 视图。
- python/sglang/srt/mem_cache/hiradix_cache.py（模块 高基数缓存；类别 source；类型 core-logic；符号 match_prefix, insert）：高基数缓存调用站点微调，保持外部哈希键兼容性。
- test/registered/unit/mem_cache/test_swa_unittest.py（模块 单元测试；类别 test；类型 test-coverage；符号 test_swa_radix_cache_eagle）：测试配套更新，验证 bigram 视图行为正确性。

关键符号：RadixKey.init, RadixKey.len, RadixKey.iter, RadixKey.getitem, RadixKey.maybe_to_bigram_view, page_align_keys

关键源码片段

python/sglang/srt/mem_cache/radix_cache.py

核心数据结构变更，重新设计 RadixKey 以支持 bigram 视图，是性能优化的基础。

```
class RadixKey:
    """is_bigram=True: token_ids 持有原始令牌（N个bigrams对应N+
    1个tokens）；切片共享一个边界令牌。"""

    __slots__ = ("token_ids", "extra_key", "is_bigram") # 优化内存使用

    def __init__(self, token_ids: List[int], extra_key: Optional[str] = None, is_bigram: bool = False)
    :
        self.token_ids = token_ids # 原始令牌序列，两种模式下都是整型列表
        self.extra_key = extra_key # 额外键（如 lora_id、cache_salt）
        self.is_bigram = is_bigram # bigram 视图标志：True 时长度 = max(0, len(token_ids) - 1)

    def __len__(self) -> int:
        if self.is_bigram:
            n = len(self.token_ids)
```

```
            return n - 1 if n > 0 else 0 # bigram 模式下逻辑长度为原始令牌数减一
        return len(self.token_ids)

    def __iter__(self) -> Iterator:
        if self.is_bigram:
            t = self.token_ids
            for i in range(len(t) - 1):
                yield (t[i], t[i + 1]) # 迭代返回 bigram 元组，避免物化整个列表
        else:
            yield from self.token_ids # 普通模式直接迭代令牌

    def __getitem__(self, idx: Union[int, slice]) -> "RadixKey":
        # 将整数索引规范化为切片，简化后续处理
        if isinstance(idx, int):
            if idx < 0:
                idx += len(self)
            if idx < 0 or idx >= len(self):
                raise IndexError(f"RadixKey索引越界: {idx}")
            idx = slice(idx, idx + 1)
        start, stop, step = idx.indices(len(self))
        if step != 1:
            raise ValueError("RadixKey切片步长必须为1")

        if self.is_bigram:
            # bigrams [start, stop) 对应原始令牌 [start, stop + 1)；空切片返回空列表
            raw = self.token_ids[start : stop + 1] if stop > start else []
            return RadixKey(raw, self.extra_key, is_bigram=True) # 新视图共享边界令牌
        return RadixKey(self.token_ids[start:stop], self.extra_key)

    def maybe_to_bigram_view(self, is_eagle: bool, value: Optional[torch.Tensor] = None):
        """O(1)转换：翻转bigram标志而非物化元组列表。"""
        if is_eagle and not self.is_bigram:
            self.is_bigram = True # 直接切换视图模式
            if value is not None:
                value = value[: len(self)] # 截断张量以匹配 bigram 长度
        return self, value
```

评论区精华

review 中未出现实质性争议，PR 被 ispobock 直接批准，表明设计得到团队认可。提交历史显示多次合并和修复（如修复索引越界、迭代器分支丢失），侧面反映了开发过程中的细致调整。

- 批准与无争议 (other)：设计被接受，无修改意见。

风险与影响

- 风险：边界条件处理风险：RadixKey 切片逻辑在 bigram 模式下需精确处理空切片和边界 token 共享，若实现有误可能导致数据不一致或崩溃。兼容性风险：虽然哈希计算保持相同，但外部依赖 hiradix_cache.py 仍使用 convert_to_bigram_key 作为存储键，需确保交互

无影响。回归风险：核心路径变更可能影响 EAGLE 模式下的缓存匹配和插入正确性，依赖测试覆盖。

- 影响：性能影响：在 1M 令牌基准测试中，convert 步骤从 ~47ms 降至 ~0ms（>100,000 倍优化），匹配时间从 113ms 降至 16ms（约 7 倍提升），整体 cache_unfinished_req 周期从 70ms 降至 23ms（约 3 倍加速）。系统影响：降低长上下文 EAGLE 推理的延迟，提升 KV 缓存子系统吞吐量。团队影响：引入更高效的视图模式，减少 Python 对象分配，但增加了 RadixKey 的复杂度，需确保团队成员理解新语义。
- 风险标记：核心路径变更，边界条件风险，测试覆盖调整

关联脉络

- PR #23275 fix: add back priorty as radix cache policy: 同属 KV 缓存子系统，涉及基数树缓存策略调整。
- PR #23202 [core] Always-on StreamingSession in UnifiedRadixCache: 涉及统一基数缓存的核心重构，与本 PR 的性能优化互补。