PR #21509 完整报告

sgl-project/sglang

[MLX] Support radix cache

合并时间：2026-04-18 07:00

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/21509

执行摘要

- 一句话：为 MLX 后端添加基数缓存，提升共享前缀工作负载的预填充吞吐量。
- 推荐动作：该 PR 值得精读，特别是关注基数缓存与调度器的集成设计、MLX 原生内存管理策略（如自动池大小计算）以及批处理解码的实现权衡。建议工程师在类似后端扩展时参考其模块化设计（如分离 KV 池、缓存类和注意力包装器），但需注意测试覆盖和架构兼容性的不足。

功能与动机

根据 PR body 的描述，动机是“为 MLX 后端添加基数缓存，使其更接近基于 PyTorch 的实现的功能对等，并提高具有共享前缀的工作负载的预填充吞吐量”。作者在 issue 评论中进一步说明，此变更旨在减少重复计算，提升 MLX 后端在处理共享提示前缀时的效率。

实现拆解

1. 创建 KV 缓存子包：在 python/sglang/srt/hardware_backend/mlx/kv_cache/ 下新增多个文件，定义核心缓存组件。 - kv_pool.py 中的 MlxKVPool 类提供扁平 KV 存储池，根据可用 Metal 内存自动调整大小。 - contiguous_cache.py 中的 ContiguousKVCache 和 PoolBackedCache 分别用于解码和预填充的缓存管理，OffsetCache 作为无数据占位符满足 mlx-lm 缓存协议。 - attention_wrapper.py 中的 MLXAttentionWrapper 支持 BS>1 的解码批处理，通过线程局部上下文实现向量化 RoPE。 - model_patching.py 提供模型自省和注意力层修补功能。
2. 修改模型运行器：更新 python/sglang/srt/hardware_backend/mlx/model_runner.py 中的 MlxModelRunner 类，集成基数缓存逻辑。 - 新增 _acquire_cache、_release_cache 等方法管理缓存重用。 - prefill 和 decode_batch 方法现在接收调度器提供的 prefix_slot_ids 和 new_slot_ids，从 MlxKVPool 读取缓存 KV 并写入新结果。 - 添加 _compute_pool_size 根据可用内存自动计算池大小。
3. 更新工作器和存根：修改 tp_worker.py 以提取 ModelWorkerBatch 中的前缀信息并传递给 MLX 运行器；调整 model_runner_stub.py 以支持池大小配置。
4. 基准测试配套：更新 bench_one_batch.py，添加 disable_radix_cache、mem_fraction_static 和 pool_size 参数，便于性能评估。
5. 导入和依赖调整：新增 kv_cache/__init__.py 统一导出符号，确保模块间正确引用。

关键文件：

- python/sglang/srt/hardware_backend/mlx/model_runner.py（模块 MLX 模型运行器；类别 source；类型 core-logic；符号 MlxModelRunner, _acquire_cache, _release_cache, _eval_with_cache）：作为 MLX 后端的主模型运行器，负责集成基数缓存逻辑，处理预填充和解码的 KV 缓存读写，是整个变更的核心入口。
- python/sglang/srt/hardware_backend/mlx/kv_cache/contiguous_cache.py（模块 MLX 缓存类；类别 source；类型 core-logic；符号 ContiguousKVCache, PoolBackedCache, OffsetCache, update_and_fetch）：定义了核心缓存类 ContiguousKVCache 和 PoolBackedCache，分别用于解码时的连续存储和预填充时的池后端缓存，是基数缓存的数据结构基础。
- python/sglang/srt/hardware_backend/mlx/kv_cache/attention_wrapper.py（模块 注意力包装器；类别 source；类型 core-logic；符号 MLXAttentionWrapper, BatchedDecodeContext, _batched_decode, set_context）：实现批处理解码注意力包装器，通过线程局部上下文支持 BS>1 的向量化 RoPE 和缓存写入，是提升解码吞吐量的关键。
- python/sglang/srt/hardware_backend/mlx/kv_cache/kv_pool.py（模块 KV 存储池；类别 source；类型 core-logic；符号 MlxKVPool, set_kv, get_kv, get_kv_all_layers）：提供扁平的 KV 存储池 MlxKVPool，作为所有缓存 KV 的中心化存储，支持基于槽位 ID 的快速散射和聚集操作。
- python/sglang/srt/hardware_backend/mlx/tp_worker.py（模块 工作器集成；类别 source；类型 dependency-wiring；符号 _ensure_mlx_pool_initialized, forward_batch_generation）：修改 MLX 特定的 TP 工作器以初始化基数缓存池并传递前缀信息，确保调度器与 MLX 运行器之间的数据流连接。

关键符号：MlxModelRunner.prefill, MlxModelRunner.decode_batch, ContiguousKVCache.update_and_fetch, PoolBackedCache.update_and_fetch, MLXAttentionWrapper._batched_decode, MlxKVPool.set_kv, MlxKVPool.get_kv, _ensure_mlx_pool_initialized

关键源码片段

python/sglang/srt/hardware_backend/mlx/model_runner.py

作为 MLX 后端的主模型运行器，负责集成基数缓存逻辑，处理预填充和解码的 KV 缓存读写，是整个变更的核心入口。

```
class MlxModelRunner:
    """MLX 模型运行器，支持基数缓存前缀共享。"""

    def _acquire_cache(self) -> list[ContiguousKVCache]:
        """从池中获取可重用的缓存列表，或创建新列表。

        这是缓存复用的关键优化，避免每次请求都分配新内存，减少开销。
        """
        if self._cache_pool:
            cache = self._cache_pool.pop()
            for c in cache:
                c.offset = 0 # 重置偏移量，重用缓冲区
```

```
            return cache
        return [
            ContiguousKVCache(max_seq_len=self._max_seq_len)
            for _ in range(self._num_layers)
        ]

    def _release_cache(self, cache: list[ContiguousKVCache]) -> None:
        """将缓存列表返回到池中以供重用。"""
        self._cache_pool.append(cache)

    def prefill(self, req_id: str, token_ids: list[int],
                prefix_slot_ids: list[int] | None = None,
                new_slot_ids: list[int] | None = None) -> int:
        """运行预填充，支持基数缓存前缀重用。

        Args:
            prefix_slot_ids: 调度器提供的已缓存前缀槽位 ID，用于从 KV 池读取。
            new_slot_ids: 新分配的槽位 ID，用于将计算后的 KV 写入池中。
        关键逻辑：如果禁用基数缓存，则回退到从头计算；否则使用 PoolBackedCache 聚集缓存 KV。
        """
        if self.disable_radix_cache:
            # 回退路径：无缓存重用
            cache = [ContiguousKVCache() for _ in range(self._num_layers)]
            input_ids = mx.array([token_ids], dtype=mx.int32)
        else:
            # 基数缓存路径：使用 PoolBackedCache 从 MlxKVPool 读取前缀
            cache = [
                PoolBackedCache(self._kv_pool, i, prefix_slot_ids)
                for i in range(self._num_layers)
            ]
            input_ids = mx.array([token_ids[-len(new_slot_ids):]], dtype=mx.int32)
        # 运行模型并写入新 KV 到池
        model_output = self.model(input_ids, cache=cache)
        if new_slot_ids is not None:
            self._kv_pool.set_kv_all_layers(mx.array(new_slot_ids), new_k, new_v)
        return mx.argmax(logits[:, -1, :], axis=-1).item()
```

python/sglang/srt/hardware_backend/mlx/kv_cache/contiguous_cache.py

定义了核心缓存类 ContiguousKVCache 和 PoolBackedCache，分别用于解码时的连续存储和预填充时的池后端缓存，是基数缓存的数据结构基础。

```
class ContiguousKVCache:
    """预分配的 KV 缓冲区，用于单个请求的单个层。

    形状为 (1, n_kv_heads, max_seq_len, head_dim)，通过切片赋值而非 mx.concatenate
    来提升性能。
    延迟分配：首次写入时才分配内存，节省初始开销。
    """
```

```
    def __init__(self, n_kv_heads: int | None = None, head_dim: int | None = None,
                 max_seq_len: int = 4096, dtype: mx.Dtype | None = None):
        if n_kv_heads is not None and head_dim is not None and dtype is not None:
            # 提前分配缓冲区
            self.keys = mx.zeros((1, n_kv_heads, max_seq_len, head_dim), dtype=dtype)
            self.values = mx.zeros((1, n_kv_heads, max_seq_len, head_dim), dtype=dtype)
        else:
            self.keys = None
            self.values = None
        self.offset = 0 # 当前有效令牌数
        self.max_seq_len = max_seq_len

    def update_and_fetch(self, keys: mx.array, values: mx.array) -> tuple[mx.array, mx.array]:
        """追加 K/V 并返回截至当前偏移量的所有有效 K/V。

        这是 mlx-lm 缓存协议的核心方法，在每次前向传播中被调用。
        若缓冲区未分配，则根据 keys 的形状动态分配。
        """
        if self.keys is None:
            self._allocate(keys) # 延迟分配
        S = keys.shape[2]
        end = self.offset + S
        if end > self.max_seq_len:
            self._grow(end) # 缓冲区不足时倍增容量
        # 切片赋值，避免拼接开销
        self.keys[:, :, self.offset:end, :] = keys
        self.values[:, :, self.offset:end, :] = values
        self.offset = end
        return self.keys[:, :, :end, :], self.values[:, :, :end, :]
```

评论区精华

- 测试覆盖：Jonahcb 要求为 _batched_decode 函数和基数缓存添加单元测试，作者回应将在后续 PR 中补充，因 Mac CI 基础设施尚未就绪。
  - OffsetCache的必要性：Jonahcb 质疑其作用，作者解释它作为无数据占位符，满足 mlx-lm 模型对缓存协议的期望（如提供 offset 和 make_mask），而实际 KV 管理由 MLXAttentionWrapper 处理。
- 兼容性风险：alexnails 和 changminbark 讨论此实现可能破坏链式解码功能，changminbark 表示需要后续合并和修复。
- 实现细节：
  - alexnails 询问注意力配置是否适用于 DeepSeek 等非标准架构，作者承认当前回退逻辑有限，需后续增强。
  - alexnails 确认批处理预填充仍支持，本 PR 仅新增批处理解码路径。
  - 关于采样策略，作者说明当前 MLX 后端仅支持贪婪采样，非贪婪采样为后续任务。
- 代码质量：alexnails 建议使用 mx.finfo 替代硬编码掩码值，作者已修复。

- 单元测试覆盖 (testing): 作者同意在后续 PR 中补充测试，因 Mac CI 基础设施尚未就绪，但已本地准备了一些测试文件。
- OffsetCache 的必要性 (design): 作者澄清了设计意图，OffsetCache 被保留以确保模型正确运行，避免 RoPE 位置和掩码错误。
- 与链式解码的兼容性 (correctness): changminbark 表示可能冲突，需要后续合并和修复工作；作者承认需协调。
- 注意力配置推断的局限性 (correctness): 作者承认当前实现有限，建议在后续 PR 中增强以支持更广的模型架构。

风险与影响

- 风险：- 回归风险：对 model_runner.py 的核心路径进行重大重构，可能影响现有 MLX 推理流程，需充分测试确保向后兼容。
- 兼容性问题：与 changminbark 正在开发的链式解码功能可能存在冲突，需要后续协调合并，否则可能引入行为不一致。
- 测试覆盖不足：review 中多次指出缺少单元测试，尤其是对新缓存组件和批处理解码逻辑，可能隐藏边界条件 bug。
- 性能风险：基数缓存的延迟同步策略（如解码 KV 延迟同步）在并发场景下可能导致数据不一致，如 issue 评论中讨论的预填充读取陈旧 KV 问题，作者已部分修复但仍需验证。
- 架构局限性：注意力配置推断逻辑（_get_attn_config）可能不适用于所有模型架构（如 DeepSeek 风格），需扩展以提升鲁棒性。
- 影响：- 用户影响：MLX 后端用户在处理共享前缀的提示时将获得显著的预填充吞吐量提升（日志显示从 0.01 token/s 提高到 102.45 token/s），但当前仅支持贪婪采样，可能限制使用场景。
- 系统影响：引入新的缓存子系统和数据流，增加了 MLX 后端的复杂性，需要额外维护；基数缓存与现有调度器（RadixCache 和 TokenToKVPoolAllocator）紧密集成，确保跨后端一致性。
- 团队影响：工程师需要熟悉新增的缓存组件和 MLX 特定优化模式，后续开发需考虑与 PyTorch 后端的功能对等。
- 风险标记：核心路径变更，缺少测试覆盖，兼容性风险，架构局限性

关联脉络

- PR #22673 [Perf] Precompute gemma_weight to avoid redundant add on every forward: 同属性能优化类 PR，涉及缓存和计算优化，可对比学习 MLX 后端与通用 SGLang 的性能改进策略。
- PR #22891 [HiCache] fix: HiCacheFile component key suffixing: 涉及缓存系统的修复，虽针对不同后端（HiCache），但展示了 SGLang 项目中缓存组件的常见问题和维护模式。
- PR #23006 [Pipeline Parallelism][Bug] Fix scheduler hang in pipeline parallelism setup: 涉及调度器与缓存交互的 bug 修复，与本 PR 中基数缓存与调度器集成相关，可参考其调试方法。