

PR #37667 完整报告

sgl-project/sglang

[Speculative Decoding] Add native UNO serving support

合并时间: 2026-09-03 20:08

原文链接: <http://prhub.com.cn/sgl-project/sglang/pull/37667>

执行摘要

- 一句话: 为 UNO 投机解码提供原生 SGLang 服务支持, 实现无损推理加速。
- 推荐动作: 这是一个高价值、高风险的 PR, 值得技术负责人和相关工程师精读。它引入了一种新颖的、有潜力的推理加速算法, 并进行了深度的系统集成。精读重点应放在:
 1. 架构设计: UnoWorkerV2 如何与现有调度器和投机解码框架交互, 特别是重叠调度和 CUDA 图管理。
 2. 关键实现: UnoCublasLoRABackend 的 LoRA-A 重叠计算机制, 以及 uno_tree.py 中的 GPU 原生树构建算法。
 3. 采样正确性: 理解 uno_utils.py 中的采样路径 (稀疏 / 稠密拒绝) 及其与 EAGLE 路径的复用关系, 注意与标准采样的分布差异。
 4. 内存与调度契约: UnoDraftInput.prepare_for_decode 如何管理 UNO 特定的页对齐内存分配, 以及 GenerationBatchResult 中 output_stride 和 non_draft_token_count 的作用。

对于大多数工程师, 关注核心模块和测试即可把握要点; 对于深耕推理优化的工程师, 深入内核和调度集成将极具启发性。当前的 CI 测试覆盖存在一个关键缺口 (适配器未公开), 这应在后续迭代中解决以确保生产稳定性。

功能与动机

原生集成 UNO 到 SGLang 运行时, 以支持这种新颖的、无损的投机解码方法。UNO 通过扩散路径起草令牌, 然后由基础模型验证, 从而在不引入额外草稿模型的情况下提高吞吐量。PR body 明确了需求: "Add native SGLang serving support for UNO, a diffusion-augmented LLM." 并提供了详细的性能基准 (TPF 从 1.0 提升至 2.5+, 吞吐量提升 40-170%), 证明了其价值。

实现拆解

1. 核心调度与验证逻辑: 引入 UnoWorkerV2 作为 UNO 的原生工作器 (python/sglang/srt/speculative/uno_worker_v2.py), 继承自 BaseSpecWorker。它封装了目标工作器, 协调 UNO 的双通路 (LoRA 草稿和基础验证) 调度, 管理专属的草稿注意力后端和 CUDA 图。同时扩展了 speculative_hook.py 添加 _handle_uno 以注册算法。
2. UNO 信息与内存管理: 新增 UnoDraftInput 和 UnoForwardInput 数据类 (python/sglang/srt/speculative/uno_info.py) 来携带跨迭代的状态和元数据, 包括

bonus_tokens、forward_width 和未来索引。集成到内存分配系统 (allocation_sizing.py) 中，为 UNO 的双通路解码预留页对齐的额外 KV 空间，并修改调度器的结果处理 (batch_result_processor.py) 以正确计数 UNO 的生成令牌。

3. 优化 LoRA 执行后端：实现专用的 UnoCublasLoRABackend (python/sglang/srt/lora/backend/uno_cublas_backend.py)，通过 CUDA 流实现 LoRA-A 计算与基础 GEMM 的重叠。它支持根据令牌行路由跳过种子令牌，并管理批处理状态以适配 CUDA 图捕获。
4. 线性与树采样算法：在 uno_utils.py 中实现了 UNO 的线性模式（稀疏和稠密拒绝采样）和树模式草稿分布构建。在 uno_tree.py 中实现了 GPU 原生的提案树构建，复用 EAGLE 的验证路径。这包括专用的 Triton 内核用于候选概率归一化和树搜索。
5. CUDA 图与验证：新增 UnoDecodeCudaGraphRunner (python/sglang/srt/speculative/uno_cuda_graph_runner.py)，为线性 UNO 的双变体 (LoRA/ 基础) 和树 UNO 的独立草稿 / 验证路径分别捕获和重放 CUDA 图。同时，为树验证添加了融合的后缀注意力合并内核，并将其集成到共享的 FlashAttention 后端。
6. 请求验证与基准测试：添加请求验证逻辑 (uno_validation.py)，确保 UNO 的配置约束。创建了完整的基准测试框架 (benchmark/uno/)，包括数学评估器 (run_math_eval.py)、数据集准备 (math_data.py) 和评分器 (math_grader.py)，用于复现和验证 UNO 的性能与数学准确性。
7. 测试套件：添加了大量单元测试和集成测试，覆盖配置验证 (test_uno_tree_config.py)、不活跃 LoRA 批处理 (test_uno_inactive_lora_batch.py)、稀疏采样 (test_uno_tree_sparse_sampling.py)、后缀注意力合并 (test_suffix_attention_merge.py) 以及端到端 CUDA 图正确性 (test_uno.py)。

关键文件：

- python/sglang/srt/speculative/uno_worker_v2.py (模块 UNO 工作器；类别 source；类型 core-logic；符号 UnoWorkerV2, init, _build_uno_draft_attn_backend, init_attention_backends)：核心 UNO 工作器，封装调度逻辑、双通路执行、注意力后端管理和 CUDA 图集成。
- python/sglang/srt/lora/backend/uno_cublas_backend.py (模块 LoRA 后端；类别 source；类型 dependency-wiring；符号 UnoCublasLoRABackend, prepare_lora_token_segments, _compute_lora_a, _accumulate_lora_b)：为 UNO 优化的 LoRA 执行后端，实现 LoRA-A 与基础 GEMM 的 CUDA 流重叠，是性能关键组件。
- python/sglang/srt/speculative/uno_tree.py (模块 UNO 树构建；类别 source；类型 core-logic；符号 UnoTreeProposal, _candidate_lsepartials_kernel, _build_tree_kernel)：实现 UNO 树模式的 GPU 原生提案树构建，是支持高吞吐量树采样的核心算法。

关键符号：UnoWorkerV2.init, UnoWorkerV2._build_uno_draft_attn_backend, UnoWorkerV2.init_cuda_graphs, UnoCublasLoRABackend.prepare_lora_token_segments, UnoCublasLoRABackend._compute_lora_a, UnoDraftInput.prepare_for_decode, build_uno_tree_proposal, _sparse_rejection_from_support, _build_tree_kernel, _candidate_lse_finalize_kernel, init_uno_lora_manager, UnoDecodeCudaGraphRunner.capture_one_shape

关键源码片段

python/sglang/srt/speculative/uno_worker_v2.py

核心 UNO 工作器，封装调度逻辑、双通路执行、注意力后端管理和 CUDA 图集成。

```
# python/sglang/srt/speculative/uno_worker_v2.py
```

```
# 核心: UnoWorkerV2 调度双通路执行
```

```
class UnoWorkerV2(BaseSpecWorker):
```

```
    """单模型 UNO 工作器，支持线性和原生 EAGLE 树解码。"""
```

```
    def __init__(self, server_args, gpu_id, ps, nccl_port, target_worker):
```

```
        # ... 初始化 ...
```

```
        self._target_worker = target_worker
```

```
        self.model_runner = target_worker.model_runner
```

```
        self.lora_manager = self.model_runner.lora_manager
```

```
        self.uno_lora_id = self.model_runner.uno_lora_id
```

```
        # 根据配置确定是否为树模式
```

```
        configured_topk = int(get_spec().speculative_eagle_topk or 1)
```

```
        self.tree_mode = configured_topk > 1
```

```
        # 设置前向宽度、验证宽度等关键参数
```

```
        self.forward_width = (
```

```
            int(get_spec().speculative_num_steps) + 1
```

```
            if self.tree_mode
```

```
            else int(get_spec().speculative_num_draft_tokens)
```

```
        )
```

```
        self.verify_width = int(get_spec().speculative_num_draft_tokens)
```

```
    def _build_uno_draft_attn_backend(self):
```

```
        """为树模式构建专属的 F/1 草稿注意力后端。"""
```

```
        model_runner = self.model_runner
```

```
        original_workspace_flag = model_runner.init_new_workspace
```

```
        try:
```

```
            # 临时覆盖配置以获取正确的草稿后端
```

```
            with get_spec().override(
```

```
                speculative_num_steps=1,
```

```
                speculative_eagle_topk=1,
```

```
                speculative_num_draft_tokens=self.forward_width,
```

```
            ):
```

```
                return model_runner._get_attention_backend(init_new_workspace=True)
```

```
        finally:
```

```
            model_runner.init_new_workspace = original_workspace_flag
```

```
    def init_cuda_graphs(self):
```

```
        """为树模式捕获 F-wide 的草稿图。"""
```

```
        # 仅树模式需要独立草稿图
```

```
        if self.tree_mode and self.model_runner.decode_cuda_graph_runner is not None:
```

```
            with self._bind_uno_draft_runtime():
```

```
                self._uno_draft_cuda_graph_runner = UnoDecodeCudaGraphRunner(
```

```

        self.model_runner,
        tree_draft_attn_backend=self._uno_draft_attn_backend,
        tree_draft_width=self.forward_width,
    )

```

python/sglang/srt/lora/backend/uno_cublas_backend.py

为 UNO 优化的 LoRA 执行后端，实现 LoRA-A 与基础 GEMM 的 CUDA 流重叠，是性能关键组件。

```

# python/sglang/srt/lora/backend/uno_cublas_backend.py
# 核心：通过 CUDA 流重叠 LoRA-A 计算

```

```

class UnoCublasLoRABackend(TritonLoRABackend):
    """UNO 草稿前向传播的快速 LoRA 后端，使用 cuBLAS 和 CUDA 流。"""

    def __init__(self, max_loras_per_batch, device, **kwargs):
        super().__init__(max_loras_per_batch, device, **kwargs)
        # 为每个 CUDA 图运行器的主流分配独立的 LoRA-A 侧流，避免并发干扰
        self._lora_a_streams: dict[torch.cuda.Stream, torch.cuda.Stream] = {}
        self._pending_lora_a: Optional[_PendingLoRAA] = None
        self._use_cublas_lora_b = False

    def prepare_lora_token_segments(self, *, segment_lens, weight_indices, lora_ranks, scalings):
        """准备 UNO 特有的令牌段路由。"""
        super().prepare_lora_token_segments(...)
        route = None
        # 识别 UNO 的批量草稿结构：每个请求 1 个种子 + (F-1) 个草稿令牌
        if (len(segment_lens) >= 2 and len(segment_lens) % 2 == 0
            and segment_lens[0] == 1 and segment_lens[1] > 0):
            batch_size = len(segment_lens) // 2
            forward_width = segment_lens[1] + 1
            # 检查是否匹配 UNO 单适配器模式
            if (...): # 验证所有段匹配 base/adapters 模式
                route = _UnoSingleAdapterRoute(
                    weight_index=adapter_index, rank=adapter_rank,
                    scaling=float(scalings[adapter_index]),
                    batch_size=batch_size, forward_width=forward_width,
                    total_rows=sum(segment_lens),
                    active_rows=batch_size * (forward_width - 1),
                )
            # 将路由信息附加到当前批处理信息中，用于后续 GEMM 执行
            self.batch_info.uno_single_adapter_route = route

    def _compute_lora_a(self, lora_input, active_a, route, *, output=None):
        """执行 LoRA-A 计算，并为批量请求处理种子令牌。"""
        hidden = torch.mm(lora_input, active_a.t(), out=output)
        if route.batch_size > 1:
            # 对于批量请求，将种子令牌行的隐藏状态置零，避免其影响
            hidden.view(route.batch_size, route.forward_width, -1)[:1, 0].zero_()

```

```
return hidden
```

python/sglang/srt/speculative/uno_tree.py

实现 UNO 树模式的 GPU 原生提案树构建，是支持高吞吐量树采样的核心算法。

```
# python/sglang/srt/speculative/uno_tree.py
```

```
# 核心：GPU 原生的提案树构建
```

```
@dataclass(frozen=True)
```

```
class UnoTreeProposal:
```

```
    """EAGLE 原生表示的 UNO 树批次。
```

```
    它输出 `top_scores_index` 和 `parent_list`，可直接传递给 EAGLE 的 `build_tree_kernel_efficient`。
```

```
    """
```

```
    root_tokens: Tensor
```

```
    draft_tokens: Tensor
```

```
    parent_list: Tensor
```

```
    top_scores_index: Tensor
```

```
    candidate_top_k: int
```

```
    max_depth: int
```

```
@property
```

```
def num_verify_tokens(self) -> int:
```

```
    # 包含根令牌的总验证令牌数
```

```
    return int(self.draft_tokens.shape[1]) + 1
```

```
@triton.jit
```

```
def _build_tree_kernel(
```

```
    top_token_ids, top_log_probs, draft_tokens,
```

```
    selected_edges, parent_list,
```

```
    search_depths, search_log_masses,
```

```
    # ... 内存步长常量 ...
```

```
    NUM_DEPTHS: tl.constexpr, K: tl.constexpr, Q: tl.constexpr,
```

```
    BLOCK_CANDIDATES: tl.constexpr,
```

```
):
```

```
    """在 GPU 上执行固定预算的最佳优先树构建。
```

```
    每个程序实例处理一个批次，通过搜索状态（深度、对数质量）扩展最佳前驱节点。
```

```
    """
```

```
    batch = tl.program_id(0)
```

```
    # ... 初始化根节点的搜索状态 ...
```

```
    for node_index in range(1, Q):
```

```
        # 对每个候选节点，检查其父节点是否有效且已被使用
```

```
        # 更新当前节点的深度和累积对数概率质量
```

```
        # 选择尚未使用的最佳候选进行扩展
```

```
        # 将选定的边写入 `selected_edges`，并将父节点映射写入 `parent_list`
```

```
        pass
```

```
    # ... 最终写入生成的草稿令牌和父节点列表 ...
```

评论区精华

Review 讨论集中在几个关键的设计和技术细节上，已通过维护者和作者的对话得到解决：

- 内存分配尺寸：BBuf 指出 UNO 的行大小 (`get_req_to_token_extra_context_len`) 在页面大小为 1 时可能不足 ($2*(V+1)$ vs $V+4$)，可能导致 `prepare_for_decode` 出错。作者 drproduck 确认了该问题并提及这是已知问题 (#33579)，计划在通用修复 (#33581) 中解决。
- 采样分布一致性：BBuf 质疑 UNO 的 top-k-first 采样与标准采样器的 joint 过滤不同，可能影响分布的无损性。drproduck 回应称，为与现有投机解码 (EAGLE、DFLASH) 保持计算成本的一致性以便公平比较，他有意遵循了该约定。这是一个有意的设计权衡。
- 生成令牌计数：BBuf 发现 UNO 的生成令牌计数逻辑 (`num_correct_drafts = accept_len - 2`) 与下游计数器不匹配，导致每个请求少计一个生成令牌。作者已修复此问题，将非草稿令牌计数 (`num_non_draft_tokens_per_req`) 集中化并一致地应用于所有相关指标。
- 测试覆盖与正确性：BBuf 建议在 PR 离开草稿状态前启用并增强测试，特别是添加固定的贪婪 UNO 与 AR 的前缀检查以确保无损正确性。作者已添加了线性和树模式的 AR 无差异测试，验证前 32 个生成令牌完全匹配。
 - UNO 行内存大小计算在页面大小为 1 时可能不足 (correctness): 问题已确认，解决方案在更通用的 PR#33581 中规划。当前 PR 提供了临时修复思路。
 - UNO 采样分布与标准采样器不一致，影响无损性宣称 (design): 这是一个有意的设计权衡，遵循现有投机解码约定以保证比较公平性。作者建议可能在未来统一，但保留当前行为。
 - UNO 的生成令牌计数少计一个 (correctness): 计数错误已修复，修复方案更具通用性，所有投机算法现在使用统一的 `num_non_draft_tokens_per_req` 字段。
 - 端到端 CUDA 测试应启用以验证无损正确性 (testing): 测试覆盖已增强，添加了关键的正确性验证。测试启用依赖于 UNO 适配器的公开。

风险与影响

- 风险：1. 架构与集成风险：本次变更是核心架构层面的添加 (`importance_score: 8`)。UNO 深度集成了调度器、内存分配器、注意力后端、LoRA 系统和 CUDA 图捕获。任何一处集成点的疏忽都可能导致运行时崩溃或正确性错误。具体风险点包括：
 - `UnoDraftInput.prepare_for_decode` 中的页对齐内存分配逻辑（已在 review 中识别并修复）。
 - UNO 对调度器重叠调度 (`overlap_schedule`) 的依赖，若未正确处理可能影响异步执行。
 - 融合后缀注意力内核与回退路径的兼容性。
- 性能与正确性风险：2. 无损性保证：虽然有 AR 对比测试，但 PR 说明中提及的树采样与标准采样的差异 (`top-k-first`) 是一个潜在的微妙偏差，可能在非常特定条件下影响输出分布。
- CUDA 图捕获 / 重放：为 UNO 设计了复杂的多变体 (LoRA/ 基础) 图捕获，任何状态管理错误（如 LoRA 元数据）都可能导致图重放时产生错误结果。
- 新内核：引入的 Triton 树构建内核和融合注意力内核未经广泛生产环境验证。3. 兼容性风险：新增了服务器参数 `--uno-lora-path` 和 `--speculative-algorithm UNO`。如果默认行为或与现有参数的交互处理不当，可能破坏现有用例。
- 测试覆盖风险：尽管测试广泛，但关

键的 CUDA 端到端测试 (`test_uno.py`) 目前因适配器未公开而在 CI 中禁用 (`disabled="The UNO draft LoRA checkpoint is not published yet."`)，这降低了自动化回归检测的能力。

- 影响：用户影响：
 - 功能：为 SGLang 用户提供了一种新的、高性能的推理模式。通过 `--speculative-algorithm UNO` 和 `--uno-lora-path` 参数启用，可显著提升吞吐量（基准测试显示 40-170% 的 tok/s 提升）。
 - 性能：在兼容的模型和适配器（如 Qwen3-8B）上，可实现无损加速，验证了 TPF（每轮生成令牌数）从 1.0 提升至 2.5+。
 - 易用性：集成了现有的多 LoRA 管理器和 CUDA 图，用户启动方式类似其他投机解码算法。提供了详尽的基准测试脚本和文档。系统影响：
- 资源：UNO 需要加载和驻留一个额外的 LoRA 适配器 (`__uno_draft__`)，增加了显存占用。CUDA 图捕获也会消耗额外显存和时间。
- 兼容性：对共享基础设施的修改（如 `batch_result_processor.py` 中的令牌计数）是向后兼容的，仅为 UNO 添加了新的分支。
- 依赖：依赖 `math_verify` 包进行数学评估，依赖特定的 UNO 适配器（如 `s-sahoo/uno-qwen3-8B`）。团队影响：
- 开发：引入了复杂的投机解码新变体，为后续类似功能（如其他单模型投机解码）提供了实现模式和组件（如重叠 LoRA 后端）。
- 维护：新增了大量代码（+6300 行）和文件，需要团队熟悉 UNO 的机制以进行维护。测试和基准测试框架可支持未来的研究和优化。
- 风险标记：核心架构变更，新算法引入，依赖未公开组件，内存分配契约变更

关联脉络

- PR #37872 Fix UNO test adapter subdirectory resolution: 同属 UNO 功能线，修复了 UNO 测试中适配器子目录解析的问题，是本 PR 测试基础设施的一部分。
- PR #36223 [CP V1 Deprecation 2/5] Make strategy prefill CP canonical: 近期重构了并行计算（CP）的策略预填充路径。UNO 的双通路调度设计可能受益于此类调度层面的规范化重构。
- PR #37567 Fix buffer-mode idle tracking and VLM memory sizing: 修复了缓冲区空闲检测问题，UNO 的内存分配 (`prepare_for_decode`) 依赖于正确的内存池状态和调度器不变量。