# PR #31626 完整报告

sgl-project/sglang

[Feature] Beam search support

合并时间：2026-08-27 07:56

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/31626

## 执行摘要

- 一句话：请求级 beam search：列式 member-row 架构支持超宽 beam
- 推荐动作：值得精读。重点看三处设计：coordinator.py 的 select/commit 双半边拆分（overlap 下如何保持 tensor 侧无 D2H）、fork.py 的 share-on-fork KV 与 group 级去重释放、batch_tail.py 的列式行布局与剥离。评审对 admission 错误处理「fail-fast 而非 skip」的结论也值得借鉴。阅读时建议搭配 test/registered/unit/beam_search 与 test/manual/beam_search 负载测试，理解资源账目与 retract 边界。

## 功能与动机

PR body 将 beam search 定位为请求级特性：sampling_params.beam_width = k 运行 k 宽 beam 并返回 top-n 序列，n 默认 1，与 HF 及 OpenAI API 语义一致；明确「No server flag; beam and ordinary requests share the same batches and memory pools」，目标是让超宽 beam（实测 5000）在共享调度路径上可用，同时保持普通请求零回归。Issue 评论中 kjuuii 进一步提出：Qwen3.5 等 hybrid linear-attention 小模型（0.6B/2B）低成本高效率、非常适合 beam search，是这一功能的代表场景——目前该类模型仍在 Not supported 列表（SWA/mamba hybrid caches），评论未获回复。

## 实现拆解

1. 搜索核心与选择算法：新增 beam_search/beam_group.py，定义 BeamGroup 状态机（BeamGroupState、CompletedBeam、BeamResult）与 frontier 累积 logprob、leaves 父节点、_pending_steps 暂存队列；配套 beam_search/history.py 维护 BeamNode DAG，beam_search/joint_select.py 提供纯 tensor 的 joint_select、select_final_topk，固定输出形状且无 D2H 同步，保证 overlap 与 CUDA graph 兼容。
2. 列式 member-row 架构：beam_search/fork.py 提供 alias_members_prompt_kv（member 共享 leader 的 prompt KV 映射，只读）、remap_kv_mapping（reparent 仅重映射 req_to_token、不拷贝 KV）、free_member_rows（共享槽位由 group 级去重释放）、collect_orphan_slots（回收无人继承的槽位）；beam_search/batch_tail.py 的 append_beam_tail 在每个 decode forward 前把 member 行拼到 row 张量尾部，strip_beam_tail 在每个批操作（filter/merge/prepare）入口切回 1:1 布局；beam_search/logits_capture.py 的 BeamLogitsCapture.capture_pre_sample_logits 捕获预采样 logits 供选择使用。

3. 调度器接线与 overlap 分半：beam_search/coordinator.py 的 BeamCoordinator 承载全部钩子：准入 validate_and_init、relay 点 maybe_select_and_relay / select_leader_prefill（member 生成、选择与 KV 重映射）、group 完成 finalize；每个 tick 拆成 select_（launch 半边，纯 tensor、无 D2H）与 commit_（deferred 半边，DAG 构建与 finish/abort），overlap 下 commit 滞后一个 forward 并丢弃溢出步；scheduler.py 新增 init_beam_coordinator，get_num_allocatable_reqs 为 beam member 行预留 req_to_token 槽。
4. 输出与协议链路：beam_search/output.py 沿 scheduler → detokenizer → tokenizer manager 传递 BeamSearchOutput：pack_beam_search_output 打包 top-n 序列、decode_beam_search_output 填充文本、build_beam_search_out 写入 meta_info.beam_results；模块刻意不在顶层 import 调度器专属符号，保证 detokenizer、tokenizer 进程可独立导入；OpenAI 协议侧删除旧 mixin，sequence_score 收口到 sgl_ext。
5. 准入限制与测试配套：validate_and_init 显式拒绝 speculative decoding、PD 分离、dp attention、pipeline parallel、page_size > 1、层级缓存、SWA/mamba 混合缓存、LoRA、sessions、constrained decoding 等组合，并排除 mixed-chunk prefill 批；beam 组不可 retract（member 与 leader 共享 prompt KV），KV 压力下先驱逐普通请求、仍不足则整组 abort 返回 HTTP 500。测试配套包括 registered/unit/beam_search 的 core/fork/output 单测与 manual/beam_search 的 HF 对齐、负载、性能扫描测试（2062 单测通过，few_shot_gsm8k 200q 0.470 回归不变）。

关键文件：

- python/sglang/srt/beam_search/coordinator.py（模块 调度接线；类别 source；类型 core-logic；符号 _rows_topk_logprobs, BeamCoordinator, request_beam_width, validate_and_init）：beam search 的调度器接线中枢：准入、relay 点选择、生命周期与 overlap 双半边时序全部在此协调，是理解整个特性的入口。
- python/sglang/srt/beam_search/beam_group.py（模块 搜索状态；类别 source；类型 core-logic；符号 BeamGroupState, CompletedBeam, BeamResult, BeamGroup）：每请求的 beam 搜索状态机：frontier、已完成候选池、overlap 下的 num_generated/num_committed 双计数与 pending_steps 暂存，是搜索语义的核心载体。
- python/sglang/srt/beam_search/output.py（模块 输出链路；类别 source；类型 data-contract；符号 pack_beam_search_output, beam_completion_tokens, is_beam_search_batch, decode_beam_search_output）：beam 输出载体：定义跨 scheduler → detokenizer → tokenizer manager 的打包、解码与 meta_info.beam_results 构建，且刻意隔离调度器依赖，是多进程契约的关键。
- python/sglang/srt/beam_search/batch_tail.py（模块 解码批尾；类别 source；类型 core-logic；符号 BeamTailEntry, BeamTail, append_beam_tail, strip_beam_tail）：列式 member-row 架构的批级载体：decode 批尾部追加 / 剥离逻辑汇总于此，ScheduleBatch 只保留 beam_tail 字段与一行钩子，是混批安全的关键。
- python/sglang/srt/beam_search/fork.py（模块 分叉原语；类别 source；类型 core-logic；符号 StagedOrphans, neutral_member_sampling_params, alias_members_prompt_kv, free_member_rows）：member 行的分叉原语：alias 共享

leader prompt KV、reparent 仅重映射、orphan 槽位回收与 group 级去重释放，是 KV内存账目的核心。

- python/sglang/srt/beam_search/joint_select.py（模块 选择算法；类别 source；类型 core-logic；符号 SelectResult, FinalSelect, _scatter_fixed, _ranked_candidates）：纯 tensor 的联合选择算法：固定输出形状、无 D2H、无数据相关 host 分支，是 overlap/CUDA graph 兼容性的技术基础。
- python/sglang/srt/managers/scheduler.py（模块 调度器；类别 source；类型 core-logic；符号 init_beam_coordinator, get_num_allocatable_reqs）：核心调度器接线点：初始化 BeamCoordinator 并在可分配请求数计算中预留 beam member 行，普通请求的调度路径唯一改动处。
- python/sglang/srt/beam_search/logits_capture.py（模块 采样捕获；类别 source；类型 core-logic；符号 BeamLogitsCapture, capture_pre_sample_logits）：采样前 logits 捕获器：把 beam 需要的预采样 logits 集中成单一结构，避免散落字段污染 logits_processor。
- test/registered/unit/beam_search/test_beam_search_core.py（模块 核心单测；类别 test；类型 test-coverage；符号 run_select, reference_select, TestJointSelectGolden, test_no_stop_fast_path）：joint_select 的金标准单测：run_select 与 reference_select 对照，覆盖无 stop 快速路径与 stop 路由分支，是选择算法正确性的主要保障。

关键符号：_rows_topk_logprobs, request_beam_width, validate_and_init, maybe_select_and_relay, select_leader_prefill, append_beam_tail, strip_beam_tail, num_beam_member_rows, beam_retraction_order, alias_members_prompt_kv, remap_kv_mapping, free_member_rows, collect_orphan_slots, joint_select, select_final_topk, pack_beam_search_output, decode_beam_search_output, beam_completion_tokens, capture_pre_sample_logits, init_beam_coordinator

## 关键源码片段

### python/sglang/srt/beam_search/coordinator.py

beam search 的调度器接线中枢：准入、relay 点选择、生命周期与 overlap 双半边时序全部在此协调，是理解整个特性的入口。

```
# python/sglang/srt/beam_search/coordinator.py（节选）
# 一个 beam_width = k 的请求以「1 个 leader Req + k - 1 个裸 member row」运行：
# member row 只是 req_to_token 里的物理行，背后没有 Req 对象。
# decode 前由 batch_tail.append_beam_tail 追加到行张量尾部，
# 采样前由 tp_worker 切掉，所以 reqs 对齐的世界永远看不到它们。

def _rows_topk_logprobs(pieces: Sequence[torch.Tensor], num_candidates: int):
    # 每个 piece 单独计算 logsumexp：CUDA 上 logsumexp 不是 batch 不变的，
    # 折叠 piece 会让 leader 的 lse 差一个 ulp，从而翻转分数接近的 beam。
    vals, toks = [], []
    for piece in pieces:
        if piece.shape[0] == 0:
            continue
        x = piece.float()
        # 用 lse 代替完整 [rows, vocab] log_softmax：单调平移不改变
```

```
        # 原始 logits 上的 topk 顺序，省掉一次 softmax 的规约开销。
        lse = torch.logsumexp(x, dim=-1, keepdim=True)
        v, t = torch.topk(x, num_candidates, dim=-1)
        vals.append(v - lse)
        toks.append(t)
    return torch.cat(vals), torch.cat(toks)


class BeamCoordinator(msgspec.Struct, kw_only=True):
    model_config: ModelConfig
    spec_algorithm: SpeculativeAlgorithm
    dllm_enabled: bool
    max_req_len: int
    req_to_token_pool: ReqToTokenPool
    token_to_kv_pool_allocator: BaseTokenToKVPoolAllocator
    tree_cache: BasePrefixCache
    future_map: FutureMap

    # 存活的（未退役）group 数；这是每个 forward relay 钩子的 O(1) 门，
    # 没有 beam 请求时几乎零开销。
    _num_live_groups: int = 0

    @staticmethod
    def request_beam_width(recv_req) -> int:
        # 请求的 beam_width（1 表示非 beam 请求），来自 sampling_params。
        return getattr(recv_req.sampling_params, "beam_width", None) or 1
```

## python/sglang/srt/beam_search/beam_group.py

每请求的 beam 搜索状态机：frontier、已完成候选池、overlap 下的 num_generated/num_committed 双计数与 pending_steps 暂存，是搜索语义的核心载体。

```
# python/sglang/srt/beam_search/beam_group.py（节选）
# 每个 beam 请求一个 BeamGroup：管理 frontier（前沿 beam）、已完成候选池与生命周期。
# member 不是请求：group 把它们作为 req_to_token 行（member_rows）列式跟踪，
# 与 leader 行锁步 decode。

class BeamGroup:
    def __init__(
        self,
        *,
        beam_width: int,
        length_penalty: float = 1.0,
        stop_token_ids: Sequence[int] = (),
        max_new_tokens: int,
        num_return: Optional[int] = None,
        device: torch.device | str = "cpu",
    ):
        self.beam_width = beam_width
        self.num_candidates = 2 * beam_width # 每步扩展 2k 个候选
```

```
        self.length_penalty = length_penalty
        self.max_new_tokens = max_new_tokens
        # 返回条数 n：未指定时回退 beam_width，后续由 coordinator 按请求
        # 参数解析，PR body 中 n 默认 1（与 HF、OpenAI API 对齐）。
        self.num_return = num_return if num_return is not None else beam_width
        self.stop_token_ids = torch.tensor(
            sorted(stop_token_ids), dtype=torch.int64, device=device
        )

        # frontier 初始为一条伪行（prompt 本身，累积 logprob 0）
        self.frontier_cum_logprobs = torch.zeros(1, dtype=torch.float32, device=device)
        self.leaves: List[Optional[BeamNode]] = [None] # 下一个 token 的父节点
        # num_generated 是 launch 半边的计数，num_committed 是 deferred 半边
        # 的计数（真实长度）；overlap 下 generated 可能领先一步。
        self.num_generated = 0
        self.num_committed = 0
        self.completed: List[CompletedBeam] = []
        self.state = BeamGroupState.DECODING
        # launch 半边按 (forward tick, sel) 暂存选择结果，commit 按 tick 顺序消费
        self._pending_steps: List[tuple] = []
        # coordinator 在 group 离开活跃集合（finish / abort / leader 死亡）时置位，
        # 防止重复记账。
        self.retired = False

        # 调度器接线由搜索核心外部填充：member 没有 Req，
        # 其 seq len 与 KV 长度都由 leader 隐含决定。
        self.leader = None
        # GPU [k - 1] 个 member 行下标，及对应 host 副本（用于释放行槽）；
        # 在 post-prefill 生成后、free 后为 None。
        self.member_rows: Optional[torch.Tensor] = None
        self.member_rows_cpu: Optional[torch.Tensor] = None
        # GPU [k]：leader 行在前，随后是 member_rows（frontier 行序）
        self.all_rows: Optional[torch.Tensor] = None
        # launch 半边暂存、deferred 半边释放的孤儿槽，按 tick 门控
        self.pending_orphans: List[StagedOrphans] = []
        # GC 已归还槽数累计：持有 KV 是 host 端算术（allocated - freed），
        # 避免读 tensor。
        self.slots_freed = 0

    @property
    def num_member_rows(self) -> int:
        # member 行数；未生成（prefill 前 / 已释放）时为 0。
        return 0 if self.member_rows is None else self.member_rows.shape[0]
```

## python/sglang/srt/beam_search/batch_tail.py

列式 member-row 架构的批级载体：decode 批尾部追加 / 剥离逻辑汇总于此，ScheduleBatch 只保留 beam_tail 字段与一行钩子，是混批安全的关键。

```
# python/sglang/srt/beam_search/batch_tail.py（节选）
```

```
# beam member 行搭 decode 批的布局：追加 / 剥离 / retract。
# 所有逻辑都读写 ScheduleBatch，但放在批类之外，让 ScheduleBatch
# 只保留 beam_tail 字段与一行钩子调用。

def append_beam_tail(batch: ScheduleBatch) -> None:
    # 把每个存活 group 的 member 行追加到 reqs 对齐行之后，使 decode forward
    # （分配、relay 解析、attention）覆盖到它们。reqs 大小的 host 元数据
    # （sampling_info、top_logprobs_nums、rids 等）刻意不扩展：
    # worker 在采样前会把尾部切掉。
    assert batch.beam_tail is None
    entries = []
    tails = []
    tails_cpu = []
    leader_idx = []
    widths = []
    t = 0
    for i, req in enumerate(batch.reqs):
        group = req.beam_group
        if group is None or group.member_rows is None or group.retired:
            continue
        m = group.num_member_rows
        entries.append(BeamTailEntry(group, i, t, t + m))
        tails.append(group.member_rows)
        tails_cpu.append(group.member_rows_cpu)
        leader_idx.append(i)
        widths.append(m)
        t += m
    if not entries:
        return

    leader_idx_cpu = torch.tensor(leader_idx, dtype=torch.int64)
    widths_cpu = torch.tensor(widths, dtype=torch.int64)
    leader_idx_dev = leader_idx_cpu.to(batch.device, non_blocking=True)
    widths_dev = widths_cpu.to(batch.device, non_blocking=True)

    batch.req_pool_indices = torch.cat([batch.req_pool_indices, *tails])
    batch.req_pool_indices_cpu = torch.cat([batch.req_pool_indices_cpu, *tails_cpu])
    # member 的 seq len 沿用 leader 的，用 repeat_interleave 按宽度展开
    batch.seq_lens = torch.cat(
        [
            batch.seq_lens,
            torch.repeat_interleave(batch.seq_lens[leader_idx_dev], widths_dev),
        ]
    )
    if batch.seq_lens_cpu is not None:
        batch.seq_lens_cpu = torch.cat(
            [
                batch.seq_lens_cpu,
                torch.repeat_interleave(batch.seq_lens_cpu[leader_idx_cpu], widths_cpu),
```

```
            ]
        )
    batch.orig_seq_lens = torch.cat(
        [
            batch.orig_seq_lens,
            torch.repeat_interleave(batch.orig_seq_lens[leader_idx_dev], widths_dev),
        ]
    )
    batch.seq_lens_sum = None
    batch.beam_tail = BeamTail(num_base_rows=len(batch.reqs), entries=entries)


def strip_beam_tail(batch: ScheduleBatch) -> None:
    # 恢复 1:1 的 reqs <-> rows 布局。在每个批操作（filter / merge / prepare）
    # 入口调用，因此 tail 只覆盖一个 forward。
    tail = batch.beam_tail
    if tail is None:
        return
    n = tail.num_base_rows
    assert n == len(batch.reqs), "reqs changed while a beam tail was attached"
    batch.beam_tail = None
    batch.req_pool_indices = batch.req_pool_indices[:n]
    batch.req_pool_indices_cpu = batch.req_pool_indices_cpu[:n]
    batch.seq_lens = batch.seq_lens[:n]
    if batch.seq_lens_cpu is not None:
        batch.seq_lens_cpu = batch.seq_lens_cpu[:n]
    batch.orig_seq_lens = batch.orig_seq_lens[:n]
    if batch.input_ids is not None:
        batch.input_ids = batch.input_ids[:n]
    batch.out_cache_loc = None
    batch.seq_lens_sum = None
```

## 评论区精华

该 PR 无 inline review 评论，ch-wan 直接 APPROVED（空 body），技术讨论主要沉淀在提交信息与 PR body 中：

- 提交 1c4c4c5「Addresses review feedback on BeamSearchAdmissionError handling」记录了 admission 错误处理的关键评审结论：原实现 catch-and-skip 跳过 decode tick，但跳过不会释放任何 req_to_token 槽（没有请求完成），池欠账会死循环，因此改为 fail-fast 并拒绝无法满足的 beam_width。
- 提交 166e196 记录 page_size > 1 下多 beam 分支可能共享同一个未填满的页块、导致 decode 期间 KV 槽冲突，最终在 beam 路径强制 page_size = 1。
- PR body 的 Retraction 节明确了不可回滚的设计取舍：member 行 alias 了 leader 的 prompt KV，部分回滚会破坏整组；KV 压力下先驱逐普通请求，仍不足则整组 HTTP 500，而非 requeue。
- kjuuii 的 issue 评论询问 Qwen3.5 等 hybrid 模型支持计划，目前未回复、仍未解决。

- admission 错误处理：fail-fast vs skip (correctness): 改为 fail-fast：池耗尽直接报错，并在准入时拒绝无法满足的 beam_width。
- page_size > 1 下的 KV 槽冲突 (correctness): beam 路径强制 page_size = 1，每个 decode 分配独立槽位。
- beam 组不可 retract 的取舍 (design): 接受不可回滚语义，用驱逐优先级 + 整组 abort 兜底。
- Qwen3.5 等 hybrid 模型支持计划 (question): 未回复，仍未解决。
- 返回条数 n 的默认语义 (design): n 默认 1，且 n <= k 在准入时校验。

# 风险与影响

- 风险：
  - 核心 decode 批路径变更：append_beam_tail / strip_beam_tail 挂在每个批操作入口，断言「reqs changed while a beam tail was attached」保护 1:1 布局，混批场景下任何钩子遗漏都会波及普通请求；回归验证依赖 2062 单测与 few_shot_gsm8k 0.470 不变。
  - 内存压力下的整组 abort：beam 组不可 retract，准入虽为整组预留 req_to_token 行，但 decode KV 只按每步 1 token 预算，大 beam_width 在负载下可能触发整组 HTTP 500 路径。
  - 功能组合限制：spec decoding、PD 分离、dp attention、pipeline parallel、page_size > 1、层级缓存、SWA/mamba、LoRA、sessions 等均在准入时拒绝，beam 请求也不进入 mixed-chunk prefill 批，后续需逐一解除。
  - overlap 双半边时序：select_与 commit_ 分属 launch 与 deferred 半边，commit 滞后一个 forward 并丢弃溢出步，依赖 tick 门控，出错排查成本高；作者以 SGLANG_ENABLE_STRICT_MEM_CHECK_DURING_BUSY 全程验证。
  - 平台覆盖不足：单测注册为 CUDA suite（移除 AMD），Apple Silicon 行为未验证。
- 影响：
  - 用户侧：每个请求可独立启用 beam search，n > 1 返回多条候选，与 HF/OpenAI 语义对齐；超宽 beam（数千）可用但需注意内存与准入预算。
  - 系统侧：beam 与普通请求共享批与内存池，无 beam 请求时 _num_live_groups 是 O(1) 门，近乎零开销；KV 通过 share-on-fork 与 group 级去重释放缓解内存放大。
  - 团队侧：新增 beam_search/ 子模块，扩展了调度器、detokenizer、tokenizer manager 之间的 IPC 契约（BeamSearchOutput）；随 release-highlight 发布，后续需跟进 hybrid 模型、LoRA 等限制解除与 Apple Silicon 验证。
  - 风险标记：核心 decode 批路径变更，beam 组不可 retract（内存压力下 HTTP 500），功能组合限制较多，overlap 双半边时序复杂，测试仅覆盖 CUDA 路径

# 关联脉络

- PR #15645 Beam search support（原始实现）：本 PR 的前身：vedantjh2 的提交明确说明「Brings PR #15645 (beam search) up to date with current upstream/main and

aligns it with the new architecture, on top of the original commits」；hnyls2002 随后重写为列式 member-row 架构并删除旧实现。

- PR #21640 Remove BatchMultimodalOutput: 上游移除了 BatchMultimodalOutput，beam 代码相应清理 tokenizer manager mixin 中的 Union 类型引用（提交 22a1603）。
- PR #36622 config: the record is not an object that gets passed around: config 重构系列（36618/36620/36621/36725 等）将配置解析收口到 runtime_context bags，beam 代码改为从 bags 读取已解析配置（提交「read resolved config from the bags, not the server_args seed」），与并行配置直读趋势保持一致。
- PR #36288 [1/N][Mix] Mixed Chunk Prefill Base: Mixed Chunk Prefill 引入异构 step 调度，beam 请求当前被排除在 mixed-chunk prefill 批之外，两条调度演进线未来可能整合。