PR #31320 完整报告

sgl-project/sglang

[NPU] [Diffusion] support distributed inference pipeline for GLM-Image

合并时间：2026-08-28 20:39

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/31320

执行摘要

- 一句话：新增 GLM-Image 分布式推理拓扑：AR 与 DiT 分角色 fan-out
- 推荐动作：值得精读。该 PR 展示了如何在既有 N:M:K 解耦编排器中并行走第二条 fan-out 调度模式：两段队列 + ThreadPoolExecutor 重叠 AR/denoiser、ZMQ monitor 驱动 worker 生命周期、n>1 拆分 - 顺序执行 - 合并的数据契约，都是可借鉴的设计。建议重点关注超时 / 断连恢复与共享 batch 属性访问的健壮性是否在后续 PR 中补齐，并为外部 AR 服务的 batching 语义变化补充回归基线。

功能与动机

GLM-Image 的生成链路由自回归 prior 生成与 DiT 扩散去噪两阶段组成，单张 NPU 卡上 DiT 权重常驻显存、只能以 batch-1 运行，而 AR 阶段天然适合批量。PR body 明确说明设计目标："adds a GLM-Image distributed serving topology that separates autoregressive (AR) prior-token generation from diffusion execution"，head 批量请求外部 AR 服务后 "fans each AR-complete request out to an independent batch-1 denoiser"。这样 AR 侧获得 batching 吞吐、denoiser 侧获得独立水平扩展能力，同时避免在角色间搬运大体积 latent/embedding 张量（GLM 分布式模式只走 ZMQ 元数据 + prior token，绕过 Mooncake/RDMA）。

实现拆解

实现按 5 步展开：

1. 模式识别与启动装配（runtime/launch_server.py）：parse_url_string 改为容忍 None；kill_process_tree 下沉到 runtime/utils/common.py 供编排器与测试复用；launch_disagg_server 新增 glm_distributed_mode_enabled 判定（GlmImagePipelineConfig + --srt-encoder-url + 未配置 --encoder-urls/--decoder-urls），该模式下必需参数只校验 --denoiser-urls 并提前 set_global_server_args。
2. 编排器两段队列调度（runtime/disaggregation/orchestrator.py）：新增 _GlmDistributedRequest 与 _GlmDistributedModeState（pending_ar_requests/pending_denoiser_requests 两段 deque、denoiser_worker_available 位图、active_ar_batch Future）；_dispatch_glm_ar_batch_if_ready 按分辨率与 batching_max_size 成组提交到 ThreadPoolExecutor，使阻塞式外部 AR 调用与 denoiser 工作重叠；_process_glm_ar_batch_result_if_ready 将 prior token 行与 usage 写回各请求；

_dispatch_glm_denoiser_requests_if_ready 逐个派发；
_handle_glm_denoiser_monitor_event 用 ZMQ monitor 事件维护连通性，断连 worker 停止接单、在途请求 requeue 并重置 tracker 状态。

3. 请求数据契约（runtime/disaggregation/scheduler_mixin.py）：
_is_glm_distributed_mode 决定跳过预分配槽位等常规传输初始化；
_execute_glm_distributed_denoiser_request 只中转请求元数据与 prior-token 张量；
_expand_glm_distributed_outputs 处理 n>1（校验 token 行数、
expand_request_outputs 拆分、逐行赋值与 usage_by_output 透传）；
_advertised_pool_work_endpoint 把 0.0.0.0 替换为 --disagg-p2p-hostname/host；另增 _BASE_SAMPLING_PARAM_FIELDS 修正默认值判断。

4. 流水线阶段角色绑定（runtime/pipelines/glm_image.py、
pipelines_core/stages/model_specific_stages/glm_image.py）：新增
GlmImageDenoiserPreparationStage/GlmImageDenoiserDecodingStage，role_affinity 固定为 DENOISER，把原 head 侧的 prompt/glyph 准备与原 decoder 侧的 VAE 解码都合并到 denoiser；run_grouped_requests 拆出 generate_and_assign_prior_tokens，device 参数化并改用 current_platform.get_local_torch_device()。

5. 测试与文档配套：新增 test/server/ascend/test_glm_image_distributed.py（AR + denoiser + head 三进程集群冒烟测试，覆盖 n=1、n=2 输出合并与生命周期清理）；
testcase_configs_npu.py 增加权重路径；docs/docs/sglang-diffusion/disaggregation.mdx 增加 AR-to-DiT fan-out 部署章节（14 个 denoiser 脚本、AR server 参数、限制与 benchmark）。

关键文件：

- python/sglang/multimodal_gen/runtime/disaggregation/scheduler_mixin.py（模块 调度器；类别 source；类型 core-logic；符号 _advertised_pool_work_endpoint, _expand_glm_distributed_outputs, _is_glm_distributed_mode, _execute_glm_distributed_denoiser_request）：分布式调度 Mixin 的核心扩展：GLM 分布式模式识别、n>1 输出拆分、0.0.0.0 地址修正与仅元数据 /prior token 的角色间传输，是本次拓扑的数据契约关键载体。
- python/sglang/multimodal_gen/runtime/disaggregation/orchestrator.py（模块 编排器；类别 source；类型 core-logic；符号 _deserialize_request_metrics, _GlmDistributedRequest, _GlmDistributedModeState, _handle_decoder_result_frames）：编排器是本 PR 的核心：新增 _GlmDistributedModeState 两段队列状态机、AR 批处理派发 / 结果回写、denoiser 空闲位图与 ZMQ monitor 连通性管理，覆盖 405 行新增逻辑。
- python/sglang/multimodal_gen/runtime/pipelines/glm_image.py（模块 流水线阶段；类别 source；类型 core-logic；符号 GlmImageDenoiserDecodingStage, role_affinity, GlmImageDenoiserPreparationStage）：通过 GlmImageDenoiserPreparationStage/DecodingStage 的 role_affinity 把准备与解码阶段绑定到 denoiser，是拓扑角色合并的关键装配点。
- python/sglang/multimodal_gen/runtime/pipelines_core/stages/model_specific_stages/glm_image.py（模块 自回归阶段；类别 source；类型 data-contract；符号 run_grouped_requests, generate_and_assign_prior_tokens）：run_grouped_requests

拆出 generate_and_assign_prior_tokens 并把设备获取改为 current_platform，支撑外部 AR 批量与 head 侧 token 回写。

- python/sglang/multimodal_gen/runtime/launch_server.py（模块 启动装配；类别 source；类型 entrypoint；符号 kill_process_tree, parse_url_string）：定义了 GLM 分布式模式的判定与必需参数校验，并把 kill_process_tree/parse_url_string 通用化，是模式入口。
- python/sglang/multimodal_gen/test/server/ascend/test_glm_image_distributed.py（模块 冒烟测试；类别 test；类型 test-coverage；符号 _kill_process_tree, _tail_log, _wait_for_log, _GlmDistributedCluster）：新增的 Ascend 冒烟测试，用 _GlmDistributedCluster 拉起 AR/denoiser/head 三进程验证端到端拓扑与 n>1 合并。

关键符号：_advertised_pool_work_endpoint, _expand_glm_distributed_outputs, _is_glm_distributed_mode, _execute_glm_distributed_denoiser_request, _dispatch_glm_ar_batch_if_ready, _process_glm_ar_batch_result_if_ready, _dispatch_glm_denoiser_requests_if_ready, _handle_glm_denoiser_monitor_event, _deserialize_request_metrics, generate_and_assign_prior_tokens, GlmImageDenoiserPreparationStage.role_affinity, GlmImageDenoiserDecodingStage.role_affinity, parse_url_string

关键源码片段

python/sglang/multimodal_gen/runtime/disaggregation/scheduler_mixin.py

分布式调度 Mixin 的核心扩展：GLM 分布式模式识别、n>1 输出拆分、0.0.0.0 地址修正与仅元数据 /prior token 的角色间传输，是本次拓扑的数据契约关键载体。

```
def _advertised_pool_work_endpoint(server_args) -> str:
    # GLM 分布式模式下 denoiser 会把自身的 work endpoint 上报给 head，
    # 如果地址是 0.0.0.0，对端将无法建立连接，因此用 P2P 主机名或实际
    # host 替换掉通配地址。
    host = server_args.disagg_p2p_hostname or server_args.host or "127.0.0.1"
    if host == "0.0.0.0":
        host = server_args.disagg_p2p_hostname or "127.0.0.1"
    return server_args.pool_work_endpoint.replace("0.0.0.0", host)


def _expand_glm_distributed_outputs(req: Req) -> list[Req]:
    """把外部 AR 服务返回的 prior token 行拆成多个单输出请求。

    batch-1 的 denoiser 一次只能处理一个输出，因此当
    num_outputs_per_prompt > 1 时，先把 AR 批量结果按输出拆开、
    逐条去噪，最后在 head 侧按原始输出顺序合并回响应。
    """
    output_count = max(1, int(req.num_outputs_per_prompt or 1))
    if output_count == 1:
        return [req] # 单输出直接复用原请求，避免不必要的克隆开销

    prior_token_ids = req.prior_token_id
    if not isinstance(prior_token_ids, torch.Tensor) or (
```

```
            prior_token_ids.shape[0] != output_count
        ):
            # 行数与输出数不一致时尽早失败，避免把形状错误的 token 发给 denoiser
            actual_count = (
                prior_token_ids.shape[0]
                if isinstance(prior_token_ids, torch.Tensor)
                else type(prior_token_ids).__name__
            )
            raise RuntimeError(
                "Cannot split GLM-Image AR output for distributed inference: "
                f"expected {output_count} token rows, got {actual_count}."
            )

        usage_by_output = req.extra.get("usage_by_output")
        output_reqs = expand_request_outputs(req)
        for output_index, output_req in enumerate(output_reqs):
            # 每个输出只保留属于自己的那行 prior token，并透传对应的 usage 统计
            output_req.prior_token_id = prior_token_ids[output_index : output_index + 1]
            output_req.extra.pop("usage_by_output", None)
            if usage_by_output is not None and output_index < len(usage_by_output):
                output_req.usage = usage_by_output[output_index]
        return output_reqs
```

python/sglang/multimodal_gen/runtime/disaggregation/orchestrator.py

编排器是本 PR 的核心：新增 _GlmDistributedModeState 两段队列状态机、AR 批处理派发 / 结果回写、denoiser 空闲位图与 ZMQ monitor 连通性管理，覆盖 405 行新增逻辑。

```
@dataclass
class _GlmDistributedRequest:
    """追踪一个客户端请求从 AR 阶段到 denoiser 阶段的完整流转。"""

    client_request_id: str
    req: Req
    enqueue_time: float
    worker_idx: int | None = None # 最近一次被分派的 denoiser 下标


@dataclass
class _GlmDistributedModeState:
    """仅 GLM 外部-AR 分布式拓扑使用的调度状态。

    该拓扑下 head 内联持有 AR stage：先按分辨率把请求成组发给外部
    SRT AR 服务，再把每个 AR 完成的请求单独派发给空闲的 batch-1
    denoiser，因此同时需要两段队列和一个 denoiser 空闲位图。
    """

    server_args: "ServerArgs"
    ar_stage: "GlmImageAR"
    executor: ThreadPoolExecutor # 阻塞式外部 AR 调用跑在独立线程，可与 denoiser 工作重叠
```

```
    denoiser_worker_available: list[bool] # 每个 denoiser 是否空闲可接单
    pending_ar_requests: deque[_GlmDistributedRequest] = field(default_factory=deque)
    pending_denoiser_requests: deque[_GlmDistributedRequest] = field(
        default_factory=deque
    )
    denoiser_requests: dict[str, _GlmDistributedRequest] = field(default_factory=dict)
    active_ar_batch: tuple[Future, list[_GlmDistributedRequest]] | None = None
```

python/sglang/multimodal_gen/runtime/pipelines/glm_image.py

通过 GlmImageDenoiserPreparationStage/DecodingStage 的 role_affinity 把准备与解码阶段绑定到 denoiser，是拓扑角色合并的关键装配点。

```
class GlmImageDenoiserDecodingStage(GlmImageDecodingStage):
    """GLM 分布式拓扑没有 decoder worker，VAE 解码改在 denoiser 上执行。"""

    @property
    def role_affinity(self) -> RoleType:
        return RoleType.DENOISER


class GlmImageDenoiserPreparationStage(GlmImageBeforeDenoisingStage):
    """AR 阶段运行在 head 侧，DiT 的 prompt/glyph 准备改在 denoiser 上执行。"""

    @property
    def role_affinity(self) -> RoleType:
        return RoleType.DENOISER


# GlmImagePipeline.create_pipeline_stages 的关键分支：
def create_pipeline_stages(self, server_args: ServerArgs):
    # 判定 GLM 分布式模式：denoiser 角色 + 配置了外部 AR 服务地址
    is_glm_distributed_mode = (
        self._disagg_role == RoleType.DENOISER
        and server_args.srt_encoder_url is not None
    )
    # 分布式模式下准备阶段与解码阶段都绑定到 denoiser，
    # 否则维持原 encoder -> denoiser -> decoder 三段式装配。
    before_denoising_stage_cls = (
        GlmImageDenoiserPreparationStage
        if is_glm_distributed_mode
        else GlmImageBeforeDenoisingStage
    )
    ... # 其余 stage 装配略
    if is_glm_distributed_mode:
        self.add_stage(
            GlmImageDenoiserDecodingStage(vae=self.get_module("vae"), pipeline=self),
            "decoding_stage",
        )
    else:
```

```
        self.add_stage_factory(
            RoleType.DECODER,
            lambda: GlmImageDecodingStage(vae=self.get_module("vae"), pipeline=self),
            "decoding_stage",
        )
```

评论区精华

review 中最有价值的交锋集中在三点：

1) gemini-code-assist[bot] 指出 orchestrator 超时处理会把 worker 永久置为不可用且泄漏 shard 状态（“can sideline workers permanently … will cause memory leaks”），建议由 ZMQ monitor 驱动真实断连清理； 2) pipeline_configs/glm_image.py 直接访问 batch.prompt_embeds_mask/prompt_seq_lens 存在 AttributeError/IndexError 风险，建议 getattr 默认值； 3) layernorm.py 直接导入 sgl_kernel_npu.fused_scale_shift，内核缺失时 forward 崩溃，建议 try-except 回退。ping1jing2 则要求核对部署文档命令（model-path 统一为 zai-org/GLM-Image）并优化函数命名；OrangeRedeng 逐一回应：更新文档、删除 dynamic_batching.py 旧代码。最终 ping1jing2 approve。三处 bot 高优先级建议是否完全落实未见明确回复。

- 超时处理永久搁置 worker 与内存泄漏风险 (correctness): 最终版本将状态整合为 _GlmDistributedModeState，并在提交 7d48715 中实现断连 requeue + tracker 状态重置（DENOISING_RUNNING→DENOISING_WAITING），超时路径是否完全释放槽位仍需回归验证。
- batch 属性直接访问易抛异常 (correctness): 未见作者回复，需确认是否已加固。
- sgl_kernel_npu 导入缺少兜底 (correctness): 未见明确回复；该文件不在高信号文件列表中，是否已按建议修改需查证。
- 部署文档命令核对 (documentation): 作者已更新文档，后续 approve。
- 函数命名与旧代码清理 (style): 已解决（删除旧代码）。

风险与影响

- 风险：
  - 超时 / 断连恢复：早期版本超时会把 worker 永久搁置并泄漏 shard 状态；最终版整合为 _GlmDistributedModeState 并支持断连 requeue，但超时路径是否完全释放槽位仍需回归验证。
  - 共享 batch 属性访问：pipeline_configs/glm_image.py 直接读取 batch.prompt_embeds_mask 等字段，其他 pipeline 复用或字段缺失时有 AttributeError/IndexError 风险，review 建议未见作者明确回复。
  - NPU 内核导入无兜底：runtime/layers/layernorm.py 直接导入 sgl_kernel_npu.fused_scale_shift，自定义内核缺失时推理直接崩溃。
  - n>1 延迟：多输出在 batch-1 denoiser 上串行执行，makespan 随输出数线性增长（测试预期 6 输出约 172s），高 n 场景延迟敏感。

    - 契约变化：batching_max_size 按输出槽位计（n>1 占多槽），外部 AR 服务行为变化可能影响既有 GLM-Image 用户；ZMQ-only 传输拒绝 latent/embedding，限制未来张量级扩展。
- 影响：
    - 用户：GLM-Image 在 NPU 上获得可水平扩展的 denoiser 池 + AR 侧 batching，吞吐与资源利用率提升；但部署拓扑更复杂（外部 AR server + 多 denoiser + head），文档已给出脚本与限制说明。
    - 系统：DiffusionServer 编排器新增与 encoder→denoiser→decoder 并行的第二条调度模式，共享 PoolDispatcher 与 request tracker 框架；默认路径由参数组合隐式开关，保持向后兼容。
    - 团队：新增 Ascend standalone CI 覆盖，多 worker 生命周期管理与日志排查成为新的运维成本；后续 diffusion kernel 优化可在此拓扑的 denoiser 侧继续叠加。
    - 风险标记：核心调度路径变更，超时处理可能永久搁置 worker，NPU 内核导入无兜底，共享 batch 属性访问风险，n>1 串行延迟线性增长

关联脉络

- PR #30683 GLM-Image dynamic batching（前序 PR，提交历史多次引用）：提交历史中多次出现 Merge PR #30683 into glm-ar-dit-fanout / Merge latest PR #30683 head，并有 Remove GLM-Image DiT dynamic batching 的收敛 commit；本 PR 的 AR 批量能力继承自 #30683，并裁剪了 DiT 侧 batching 以适配 batch-1 fan-out 拓扑。
- PR #35613 [diffusion] refactor: scope model-specific API parameters: 同属 sglang/multimodal_gen 模块的模型级配置演进，与 GLM-Image 分布式模式的 pipeline config 与 API 参数收敛处于同一功能线，后续可相互支撑。