PR #51862 完整报告

vllm-project/vllm

[ROCm][Perf] Kimi-K3 Remove prefill pipeline stall in chunk KDA

合并时间：2026-08-13 17:11

原文链接：http://prhub.com.cn/vllm-project/vllm/pull/51862

执行摘要

- 一句话：ROCm Kimi-K3 预填充去掉 chunk KDA 流水线停顿
- 推荐动作：值得精读：重点看 vllm/models/kimi_k3/amd/kda_metadata.py 的设备端 metadata 构建思路，以及 gdn_attn.py 中 _build_chunk_metadata 的可覆写设计。合入前应确认新增 Triton kernel 与 CPU 路径输出一致，建议后续补充边界条件测试（空序列、混合 prefill/decode batch、num_seqs 超过 _MIN_BLOCK_N 等）。同时留意与 #51540 的关系：本 PR 是 ROCm Kimi-K3 的定向优化，#51540 是通用 GDN 优化，合并策略由维护者 tjtanaa 拍板为先合本 PR。

功能与动机

PR body 指出：在 ROCm 的 Kimi-K3 路径，每个 prefill/mixed step 在 prepare_chunk_indices 上出现 stall，原因是 .tolist() 的 D2H 拷贝让 host 等待设备队列排空，且 host 端操作在下一个 H2D 拷贝前执行时设备处于空闲。作者进一步说明，host 等待的数据其实已由 GDNAttentionMetadataBuilder 从 query_start_loc_cpu 构建并保存在 attention metadata 中，ROCm Kimi-K3 路径却没有使用，属重复劳动。

实现拆解

1. 抽取可覆写构建钩子：在 vllm/v1/attention/backends/gdn_attn.py 的 GDNAttentionMetadataBuilder 中新增 _build_chunk_metadata(prefill_query_start_loc, prefill_query_start_loc_cpu, device)，把 build() 内联的 cutedsl 分支与“CPU 预计算 + async_tensor_h2d”分支收拢为可覆写方法；build() 统一调用该方法。默认逻辑与原来等价，NVIDIA 及 Kimi-Linear 等其它 GDN 模型路径不受影响。
2. 新增 ROCm Kimi-K3 专属 backend：新建 vllm/models/kimi_k3/amd/kda_metadata.py，定义 Triton kernel _chunk_metadata_kernel 和 prepare_chunk_metadata_device，在 GPU 上从 cu_seqlens 直接生成 chunk_indices（每个元素为 (seq_idx, chunk_idx) 对）与累加偏移 chunk_offsets，全程无 D2H/H2D 同步；KimiK3ROCmKDAMetadataBuilder 覆写 _build_chunk_metadata，KimiK3ROCmKDABackend 注册名字 KIMI_K3_KDA_ROCM。
3. KDA 内核签名扩展与数据透传：在 vllm/models/kimi_k3/amd/ops/third_party/kda/chunk.py 的 chunk_kda_with_fused_gate、chunk_kda_with_fused_gate_fwd、_chunk_kda_fwd_with_cumulative_g 中新增可选参数 chunk_indices / chunk_offsets；只有调用方未提供 chunk_indices 时才回退到 prepare_chunk_indices(cu_seqlens, chunk_size)，避免设备端重建。KimiK3DeltaAttention（

vllm/models/kimi_k3/amd/kda.py）新增 get_attn_backend() 返回新 backend，并把 m.chunk_indices、m.chunk_offsets 传入 chunk_kda_with_fused_gate。

4. 平台分支导入重构：vllm/model_executor/layers/mamba/gdn/kimi_gdn_linear_attn.py 把 AMD/NVIDIA KDA 内核的运行时导入改为 TYPE_CHECKING 加平台分支，运行时按 current_platform.is_rocm() 选择实现，既满足类型检查又保持平台隔离。
5. 验证与测试配套：本 PR 未新增自动化测试文件；作者通过 MI355X 上的 vllm bench serve 与 lm_eval gsm8k 做端到端验证，并报告并发 8 到 64 的吞吐、TTFT、TPOT 对比表（低并发提升约 3%，高并发基本持平）；期间因合并冲突 rebase 后重新触发 Buildkite CI。

关键文件：

- vllm/models/kimi_k3/amd/kda_metadata.py（模块 模型层；类别 source；类型 data-contract；符号 _chunk_metadata_kernel, prepare_chunk_metadata_device, KimiK3ROCmKDAMetadataBuilder, _build_chunk_metadata）：新增文件，包含设备端 Triton chunk metadata 内核、ROCm Kimi-K3 专属 metadata builder 与 attention backend，是消除 prefill stall 的核心。
- vllm/v1/attention/backends/gdn_attn.py（模块 注意力后端；类别 source；类型 core-logic；符号 _build_chunk_metadata）：共享 GDN metadata builder 的构建逻辑被抽取为可覆写方法 _build_chunk_metadata，是通用路径的关键改造点。
- vllm/models/kimi_k3/amd/kda.py（模块 模型层；类别 source；类型 data-contract；符号 get_attn_backend）：KimiK3DeltaAttention 注册新的 attention backend，并把 metadata 中的 chunk_indices/chunk_offsets 透传给 KDA 内核，是端到端消除 stall 的接线层。
- vllm/model_executor/layers/mamba/gdn/kimi_gdn_linear_attn.py（模块 模型层；类别 source；类型 refactor）：AMD/NVIDIA KDA 内核的平台分支导入改用 TYPE_CHECKING，保证不同厂商实现签名差异不会破坏类型检查。
- vllm/models/kimi_k3/amd/ops/third_party/kda/chunk.py（模块 内核层；类别 source；类型 data-contract；符号 chunk_kda_with_fused_gate, chunk_kda_with_fused_gate_fwd, _chunk_kda_fwd_with_cumulative_g）：KDA 内核签名扩展，接受调用方现成的 chunk_indices/chunk_offsets，避免在 kernel 内部重复构建造成的同步。

关键符号：_chunk_metadata_kernel, prepare_chunk_metadata_device, KimiK3ROCmKDAMetadataBuilder._build_chunk_metadata, KimiK3ROCmKDABackend.get_name, KimiK3ROCmKDABackend.get_builder_cls, GDNAttentionMetadataBuilder._build_chunk_metadata, GDNAttentionMetadataBuilder.build, KimiK3DeltaAttention.get_attn_backend, chunk_kda_with_fused_gate, chunk_kda_with_fused_gate_fwd, _chunk_kda_fwd_with_cumulative_g

关键源码片段

vllm/models/kimi_k3/amd/kda_metadata.py

新增文件，包含设备端 Triton chunk metadata 内核、ROCm Kimi-K3 专属 metadata builder 与 attention backend，是消除 prefill stall 的核心。

```
# vllm/models/kimi_k3/amd/kda_metadata.py
# 设备端 Triton kernel：一次生成 chunk_indices 与 chunk_offsets，
# 避免 host 侧 Python 循环、D2H `.tolist()` 与 H2D 拷贝带来的同步点。
@triton.jit(do_not_specialize=["N"])
def _chunk_metadata_kernel(
    cu_seqlens,
    chunk_indices,
    chunk_offsets,
    N,
    BT: tl.constexpr,
    BLOCK_N: tl.constexpr,
    BLOCK_T: tl.constexpr,
):
    # 每个 program 负责一个序列，先取 bos/eos，再算该序列的 chunk 数。
    i_n = tl.program_id(0)
    offs_n = tl.arange(0, BLOCK_N)
    is_seq = offs_n < N
    bos = tl.load(cu_seqlens + offs_n, mask=is_seq, other=0).to(tl.int32)
    eos = tl.load(cu_seqlens + offs_n + 1, mask=is_seq, other=0).to(tl.int32)
    nt = tl.where(is_seq, tl.cdiv(eos - bos, BT), 0)

    # 用 tl.sum 归约出当前序列的起始 chunk 偏移 base，以及自身 chunk 数。
    base = tl.sum(tl.where(offs_n < i_n, nt, 0))
    num_chunks = tl.sum(tl.where(offs_n == i_n, nt, 0))

    tl.store(chunk_offsets + i_n, base)
    if i_n == 0:
        tl.store(chunk_offsets + N, tl.sum(nt))

    # 把 (seq_idx, chunk_idx) 展平写入 chunk_indices，每对占两列。
    for t0 in range(0, num_chunks, BLOCK_T):
        offs_t = t0 + tl.arange(0, BLOCK_T)
        mask_t = offs_t < num_chunks
        row = (base + offs_t) * 2
        tl.store(chunk_indices + row, tl.full([BLOCK_T], i_n, tl.int32), mask=mask_t)
        tl.store(chunk_indices + row + 1, offs_t.to(tl.int32), mask=mask_t)


def prepare_chunk_metadata_device(
    cu_seqlens: torch.Tensor,
    cu_seqlens_cpu: torch.Tensor,
    chunk_size: int,
) -> tuple[torch.Tensor, torch.Tensor]:
    # host 侧只用 cu_seqlens_cpu 计算总 chunk 数以分配显存，
    # 不等待 device 队列排空；kernel 本身完全在 device 上计算。
    num_seqs = cu_seqlens_cpu.numel() - 1
```

```
    seq_lens = cu_seqlens_cpu[1:] - cu_seqlens_cpu[:-1]
    num_chunks = int(((seq_lens + chunk_size - 1) // chunk_size).sum())

    chunk_indices = torch.empty(
        num_chunks, 2, dtype=cu_seqlens.dtype, device=cu_seqlens.device
    )
    chunk_offsets = torch.empty(
        num_seqs + 1, dtype=torch.int64, device=cu_seqlens.device
    )
    _chunk_metadata_kernel[(num_seqs,)](
        cu_seqlens,
        chunk_indices,
        chunk_offsets,
        num_seqs,
        BT=chunk_size,
        BLOCK_N=max(_MIN_BLOCK_N, next_power_of_2(num_seqs + 1)),
        BLOCK_T=_BLOCK_T,
        num_warps=4,
    )
    return chunk_indices, chunk_offsets
```

vllm/v1/attention/backends/gdn_attn.py

共享 GDN metadata builder 的构建逻辑被抽取为可覆写方法 _build_chunk_metadata，是通用路径的关键改造点。

```
# vllm/v1/attention/backends/gdn_attn.py
# 新增可覆写的 metadata 构建钩子：默认实现仍用 CPU 预计算 + 异步 H2D，
# ROCm Kimi-K3 子类可以覆写为纯 device 端构建，从而消除流水线停顿。
def _build_chunk_metadata(
    self,
    prefill_query_start_loc: torch.Tensor,
    prefill_query_start_loc_cpu: torch.Tensor,
    device: torch.device,
) -> tuple[torch.Tensor, torch.Tensor]:
    from vllm.third_party.flash_linear_attention.ops.utils import FLA_CHUNK_SIZE

    # cutedsl 后端走原有设备端路径：直接在 GPU 上准备 metadata。
    if self.gdn_prefill_backend == "cutedsl":
        from vllm.model_executor.layers.mamba.ops.gdn_chunk_cutedsl import (
            prepare_metadata_cutedsl,
        )

        assert prefill_query_start_loc is not None
        assert prefill_query_start_loc_cpu is not None
        total_tokens = int(prefill_query_start_loc_cpu[-1].item())
        return prepare_metadata_cutedsl(
            prefill_query_start_loc,
            total_tokens,
            FLA_CHUNK_SIZE,
```

```
        )

    # 默认 Triton 路径：CPU 预计算 chunk 索引后异步拷贝到 GPU，
    # 避免 GPU→CPU 的 .tolist() 同步；ROCm Kimi-K3 覆写后不再走这里。
    from vllm.third_party.flash_linear_attention.ops.index import (
        prepare_chunk_indices,
        prepare_chunk_offsets,
    )

    assert prefill_query_start_loc_cpu is not None
    return (
        async_tensor_h2d(
            prepare_chunk_indices(prefill_query_start_loc_cpu, FLA_CHUNK_SIZE),
            device=device,
        ),
        async_tensor_h2d(
            prepare_chunk_offsets(prefill_query_start_loc_cpu, FLA_CHUNK_SIZE),
            device=device,
        ),
    )
```

评论区精华

- 与 #51540 的关系（njhill）：最初认为这是 #51540 的 duplicate，随后修正为“not completely duplicate but overlapping”，即并非完全重复但存在重叠。
- 作者的澄清（kliuae）：本 PR 是扩展而非重复——#51540 在 ROCm KDA 只处理 Kimi-Linear 路径，Kimi-K3 仍会调用 prepare_chunk_indices 产生同步；本 PR 额外把 chunk_offsets 接入调用点，并在 metadata builder 中直接于 device 端构建 chunk_indices / chunk_offsets。
- 合并决策（tjtanaa）：批准本 PR，并建议先合本 PR 再合 #51540——#51540 更通用、影响 NVIDIA 与 ROCm 两条路径，但未覆盖 ROCm 上的 Kimi-K3，两者正交。
- 工程细节：mergify 提示过合并冲突，作者 rebase 后重新触发 CI（Buildkite #83526），最终合并。
  - 与 #51540 的重复性判断 (design): 两者正交：本 PR 覆盖 ROCm Kimi-K3，#51540 覆盖通用 GDN 路径（NVIDIA/ROCm 但未含 ROCm K3）。
  - 合并顺序与发布策略 (other): 先合本 PR，后合 #51540。
  - 合并冲突与 CI (other): 通过 rebase 解决冲突并完成 CI。

风险与影响

- 风险：
  - 新增 Triton kernel 的正确性：_chunk_metadata_kernel 用 tl.sum 归约计算每个序列的 chunk 偏移，若与原有 CPU 路径在边界条件（空序列、序列数非 2 的幂、超大 chunk 数）上不一致，会导致索引错乱；目前没有对应的单元测试。

- 数据契约变化：chunk_kda_with_fused_gate 等函数签名新增参数，AMD/NVIDIA 两条 kimi_k3 路径共用同一份调用约定；kimi_gdn_linear_attn.py 用 TYPE_CHECKING + 平台分支导入，若某平台实现签名未来漂移，会在运行时暴露而非类型检查期。
- 共享后端重构风险：gdn_attn.py 的 GDNAttentionMetadataBuilder.build() 是 GDN 家族共用路径，抽取 _build_chunk_metadata 必须保持默认行为与原来完全等价，Kimi-Linear 等其他 GDN 模型需要回归验证。
- 平台覆盖：本 PR 只影响 AMD 路径；若后续共享代码被 NVIDIA 路径复用，需确认 chunk_offsets 的语义一致。

- 影响：
  - 用户与系统：ROCm 上 Kimi-K3 服务的 prefill/mixed step 不再因 chunk metadata 构建而停顿，低并发（8/16）下总吞吐提升约 2.9%-3.3%，TTFT 缩短 1.4%-2.6%，TPOT 缩短约 3.2%-3.4%；高并发（32/64）下基本持平或略有改善。NVIDIA 路径完全不受影响。
  - 代码结构：GDN metadata 构建从 build() 内联逻辑中解耦为可覆写钩子，为后续通用方案（如 #51540）和厂商专属 backend 提供扩展点；新增的 KimiK3ROCmKDABackend 也是 vendor 专属 attention backend 的注册样板。
  - 团队协作：本 PR 与 #51540 正交，先合本 PR 让 ROCm Kimi-K3 立即受益，后续 #51540 再统一 NVIDIA/ROCm 通用路径。
  - 风险标记：核心路径变更，新增设备端内核，缺少自动化测试，ROCm 专属

关联脉络

- PR #51540 （上下文未提供标题，讨论中仅以链接引用）：同一功能线：移除 GDN chunk KDA 的 prefill pipeline stall；维护者确认其更通用但未覆盖 ROCm Kimi-K3，与本 PR 正交。
- PR #51772 [Attention][MLA] Fuse Kimi-K3 chunked-context K/V packing: 同属 Kimi-K3 预填充性能优化线，但作用于 NVIDIA/MLA 路径；本 PR 是 ROCm/KDA 路径的对应优化。