PR #50090 完整报告

vllm-project/vllm

[Kimi-K3] Add AttnRes kernels

合并时间：2026-07-28 15:22

原文链接：http://prhub.com.cn/vllm-project/vllm/pull/50090

执行摘要

- 一句话：为 Kimi-K3 添加 AttnRes 核函数（CUDA + Triton）
- 推荐动作：该 PR 值得精读，尤其关注 CUDA 内核的 warp-specialized 设计（producer-consumer 并行模式）以及 Triton 内核的 online softmax 实现。对于平台适配（NVIDIA + AMD）的架构设计也有借鉴意义。建议在合入后尽快添加 stride 检查并考虑更多测试场景。

功能与动机

PR 目的关联 Issue #50000，需要为 Kimi-K3 模型添加 AttnRes 操作的高性能实现。AttnRes 是模型中的关键算子，包含 residual 连接、RMSNorm 和 softmax 注意力计算。为支持 NVIDIA SM100（Blackwell）提供极致性能，并为其他 NVIDIA GPU（通过 Triton fallback）和 AMD GPU（通过 Triton）提供可用的实现。同时需要通过 Buildkite CI 确保测试持续运行。

实现拆解

1. 编写定制 CUDA C++ 内核（csrc/libtorch_stable/kimi_k3/attn_res_kernel.cu）：专为 NVIDIA SM100（Blackwell）设计，采用 warp-specialized 架构，包含 1 个 producer warp 执行 cp.async.bulk 加载和 8 个 consumer warps 计算 online softmax + residual + RMSNorm。还将 Q=res_weight*rms_weight 融合计算持久化在寄存器中，V 行在首次 pass 时转换为 FP32 并缓存在 TMEM 中。
2. 注册 Python 绑定（vllm/_custom_ops.py、csrc/libtorch_stable/ops.h、csrc/libtorch_stable/torch_bindings.cpp）：在 vllm/_custom_ops.py 中新增 kimi_k3_attn_res 函数，分配输出张量并调用 C++ 扩展；在头文件和绑定文件中声明和注册内核函数，受 VLLM_ENABLE_KIMI_K3_ATTN_RES 宏控制。
3. 实现 Triton fallback 和 AMD 内核（vllm/models/kimi_k3/nvidia/ops/attn_res.py、vllm/models/kimi_k3/amd/ops/attn_res.py）：NVIDIA 路径中提供 Triton 实现的 _attn_res_kernel，作为非 SM100 的 fallback；还包含 get_attn_res_triton_warmup_profiles 函数供内核预热使用。AMD 路径提供了独立的 Triton 内核，使用 online softmax 实现 attention 计算。
4. 添加包结构和导入（vllm/models/kimi_k3/nvidia/__init__.py、vllm/models/kimi_k3/nvidia/ops/__init__.py、vllm/models/kimi_k3/amd/__init__.py）：创建必要的包初始化文件，使模块可导入。

5. 添加测试和 CI（tests/models/kimi_k3/test_attn_res.py、tests/models/kimi_k3/test_amd_attn_res.py、.buildkite/test_areas/models_basic.yaml）：NVIDIA 测试使用参考实现（_reference，基于纯 PyTorch）对比 Triton 和 CUDA 内核的输出，覆盖空块、写入块、full attention 等场景；AMD 测试验证 Triton 实现正确性。Buildkite CI 配置新增模型基本测试任务以包含 K3 测试。

关键文件：

- tests/models/kimi_k3/test_attn_res.py（模块 注意力残差；类别 test；类型 test-coverage；符号 _randn_with_row_padding, _reference, test_attn_res, test_attn_res_block_counts）：主测试文件，包含参考实现和覆盖 Triton/CUDA 多种场景的参数化测试，是验证正确性的基石。
- vllm/models/kimi_k3/nvidia/ops/attn_res.py（模块 模型操作；类别 infra；类型 infrastructure；符号 get_attn_res_triton_warmup_profiles, _attn_res_kernel, attn_res）：NVIDIA 平台的 AttnRes Triton 实现入口，包含 Triton kernel 和调度逻辑，是 fallback 和预热的核心。
- tests/models/kimi_k3/test_amd_attn_res.py（模块 AMD 测试；类别 test；类型 test-coverage；符号 _randn_with_row_padding, _reference, test_amd_attn_res_matches_reference）：AMD 平台测试文件，验证 Triton 实现正确性，覆盖多组参数。
- vllm/models/kimi_k3/amd/ops/attn_res.py（模块 AMD 操作；类别 infra；类型 infrastructure；符号 _attn_res_kernel, attn_res）：AMD 平台的 AttnRes Triton 实现，提供 AMD GPU 上的算子支持。
- vllm/_custom_ops.py（模块 算子绑定；类别 source；类型 core-logic；符号 kimi_k3_attn_res）：Python 绑定入口，新增 kimi_k3_attn_res 函数连接 C++ 扩展。
- csrc/libtorch_stable/kimi_k3/attn_res_kernel.cu（模块 CUDA 内核；类别 other；类型 dependency-wiring）：NVIDIA SM100 专属 AttnRes 高性能 CUDA 内核，warp-specialized 设计。
- csrc/libtorch_stable/ops.h（模块 头文件；类别 source；类型 core-logic）：头文件声明 kimi_k3_attn_res 函数原型，受 VLLM_ENABLE_KIMI_K3_ATTN_RES 宏控制。
- csrc/libtorch_stable/torch_bindings.cpp（模块 绑定注册；类别 source；类型 core-logic）：C++ torch 绑定，注册 kimi_k3_attn_res 为 torch custom op。
- vllm/models/kimi_k3/amd/__init__.py（模块 包结构；类别 source；类型 data-contract）：AMD 子包初始化，使 vllm.models.kimi_k3.amd 可导入。
- vllm/models/kimi_k3/nvidia/__init__.py（模块 包结构；类别 source；类型 data-contract）：NVIDIA 子包初始化，使 vllm.models.kimi_k3.nvidia 可导入。
- vllm/models/kimi_k3/nvidia/ops/__init__.py（模块 包结构；类别 infra；类型 infrastructure）：ops 子包初始化，导出 attn_res 函数。
- .buildkite/test_areas/models_basic.yaml（模块 CI 配置；类别 config；类型 configuration）：Buildkite CI 配置，新增 Kimi-K3 测试任务。

关键符号：kimi_k3_attn_res, attn_res, _attn_res_kernel, get_attn_res_triton_warmup_profiles

关键源码片段

tests/models/kimi_k3/test_attn_res.py

主测试文件，包含参考实现和覆盖Triton/CUDA多种场景的参数化测试，是验证正确性的基石。

```
# SPDX-License-Identifier: Apache-2.0
# SPDX-FileCopyrightText: Copyright contributors to the vLLM project

import pytest
import torch
import torch.nn.functional as F

from vllm.models.kimi_k3.nvidia.ops import attn_res
from vllm.platforms import current_platform

HIDDEN_SIZE = 7168
MAX_BLOCKS = 8
EPS = 1e-5


def _randn_with_row_padding(*shape, padding=0):
    """生成带有 optional 行填充的随机张量，以模拟非连续存储场景。"""
    storage = torch.randn(*shape[:-1], shape[-1] + padding,
                          device="cuda", dtype=torch.bfloat16)
    return storage[..., :shape[-1]]


def _reference(prefix, delta, blocks, norm_weight, qk_weight,
               output_norm_weight, num_blocks):
    """纯 PyTorch 参考实现：如果 delta 存在则 prefix += delta，
    拼接 blocks 前 num_blocks 列和 prefix，做 RMSNorm，
    然后 QK 点积 -> softmax -> 加权求和，可选输出 RMSNorm。"""
    if delta is not None:
        prefix = prefix + delta
    values = torch.cat((blocks[:, :num_blocks], prefix.unsqueeze(1)), dim=1)
    keys = F.rms_norm(values, (HIDDEN_SIZE,), norm_weight, EPS)
    probs = (keys @ qk_weight).softmax(dim=-1)
    output = torch.matmul(probs.unsqueeze(1), values).squeeze(1)
    if output_norm_weight is not None:
        output = F.rms_norm(output, (HIDDEN_SIZE,), output_norm_weight, EPS)
    return output, prefix


@pytest.mark.parametrize(
    ("num_tokens", "num_blocks", "row_padding",
     "write_block", "has_delta", "backend"),
    [
        pytest.param(1, 0, 0, True, False, "triton", id="triton-empty"),
        pytest.param(1, 0, 0, True, True, "triton", id="triton-empty-add"),
```

```
        pytest.param(17, 5, 7, True, False, "triton", id="triton-write"),
        pytest.param(17, 5, 7, True, True, "triton", id="triton-write-add"),
        pytest.param(3, 8, 0, False, False, "triton", id="triton-full"),
        pytest.param(3, 8, 0, False, True, "triton", id="triton-full-add"),
        pytest.param(320, 1, 0, False, True, "nvidia", id="nvidia-1"),
        pytest.param(320, 4, 0, False, True, "nvidia", id="nvidia-4"),
        pytest.param(320, 8, 0, False, True, "nvidia", id="nvidia-8"),
    ],
)
def test_attn_res(num_tokens, num_blocks, row_padding,
                  write_block, has_delta, backend):
    if backend == "nvidia" and not current_platform.is_device_capability_family(100):
        pytest.skip("NVIDIA AttnRes requires the SM100 family")

    prefix = _randn_with_row_padding(num_tokens, HIDDEN_SIZE, padding=row_padding)
    delta = (_randn_with_row_padding(num_tokens, HIDDEN_SIZE, padding=row_padding)
             if has_delta else None)
    blocks = _randn_with_row_padding(num_tokens, MAX_BLOCKS, HIDDEN_SIZE, padding=row_
    padding)
    norm_weight = 1 + 0.1 * torch.randn(HIDDEN_SIZE, device="cuda", dtype=torch.bfloat16)
    qk_weight = torch.randn(HIDDEN_SIZE, device="cuda", dtype=torch.bfloat16) / HIDDEN_
    SIZE**0.5
    output_norm_weight = 1 + 0.1 * torch.randn(HIDDEN_SIZE, device="cuda", dtype=torch.
    bfloat16)
    original_blocks = blocks.clone()

    expected, expected_prefix = _reference(
        prefix.clone(), delta, blocks, norm_weight, qk_weight,
        output_norm_weight, num_blocks,
    )
    block_write_idx = num_blocks if write_block else -1

    actual = attn_res(
        prefix, delta, blocks, norm_weight, qk_weight, output_norm_weight,
        num_blocks, block_write_idx, EPS, EPS,
    )

    torch.testing.assert_close(actual, expected, atol=8e-2, rtol=3e-2)
    torch.testing.assert_close(prefix, expected_prefix, atol=0, rtol=0)
    if write_block:
        original_blocks[:, block_write_idx].copy_(expected_prefix)
    torch.testing.assert_close(blocks, original_blocks, atol=0, rtol=0)
    assert actual.is_contiguous()
```

vllm/models/kimi_k3/nvidia/ops/attn_res.py

NVIDIA 平台的 AttnRes Triton 实现入口，包含 Triton kernel 和调度逻辑，是 fallback 和预热的核心。

```
# SPDX-License-Identifier: Apache-2.0
```

```
# SPDX-FileCopyrightText: Copyright contributors to the vLLM project
# SPDX-FileCopyrightText: Songlin Yang, Yu Zhang, Zhiyuan Li
#
# This file contains code adapted from the flash-linear-attention project.

import torch
from vllm import _custom_ops as ops
from vllm.platforms import current_platform
from vllm.triton_utils import tl, triton

def get_attn_res_triton_warmup_profiles(max_blocks):
    """生成用于 kernel_warmup 的小批量 profiles,
    包含全 softmax 和 block-write 模式。"""
    profiles = [(num_blocks, False, -1, True) for num_blocks in range(2, max_blocks + 1)]
    profiles.extend((bw, True, bw, True) for bw in range(2, max_blocks))
    profiles.append((max_blocks, True, -1, False))
    return tuple(profiles)

@triton.jit
def _attn_res_kernel(
    prefix_ptr, delta_ptr, blocks_ptr, norm_weight_ptr, qk_weight_ptr,
    output_norm_weight_ptr, output_ptr,
    stride_prefix_m, stride_delta_m, stride_block_m, stride_block_r, stride_output_m,
    num_blocks: tl.constexpr, hidden_size: tl.constexpr,
    block_write_idx: tl.constexpr, eps: tl.constexpr, output_norm_eps: tl.constexpr,
    HAS_DELTA: tl.constexpr, WRITE_BLOCK: tl.constexpr,
    APPLY_OUTPUT_NORM: tl.constexpr,
    BLOCK_L: tl.constexpr, BLOCK_D: tl.constexpr, launch_pdl: tl.constexpr,
):
    row_idx = tl.program_id(0).to(tl.int64)
    d_offsets = tl.max_contiguous(tl.arange(0, BLOCK_D), BLOCK_D)
    d_mask = d_offsets < hidden_size

    # 加载 prefix，若 HAS_DELTA 则叠加 delta 并写回 prefix
    updated_prefix = tl.load(prefix_ptr + row_idx * stride_prefix_m + d_offsets,
                             mask=d_mask, other=0.0).to(tl.float32)
    if HAS_DELTA:
        delta = tl.load(delta_ptr + row_idx * stride_delta_m + d_offsets,
                        mask=d_mask, other=0.0).to(tl.float32)
        updated_prefix += delta
        updated_prefix = updated_prefix.to(tl.bfloat16).to(tl.float32)
        tl.store(prefix_ptr + row_idx * stride_prefix_m + d_offsets,
                 updated_prefix, mask=d_mask)
    if WRITE_BLOCK:
        # 将更新后的 prefix 写入指定的 block 位置
        tl.store(blocks_ptr + row_idx * stride_block_m +
                 block_write_idx * stride_block_r + d_offsets,
                 updated_prefix, mask=d_mask)
    # 当 num_blocks==0 时，softmax 退化为恒等，直接输出 prefix
```

```
    if num_blocks == 0:
        mixed = updated_prefix
    else:
        # ... online softmax 循环（省略）
        pass
    # 可选输出 RMSNorm
    if APPLY_OUTPUT_NORM:
        # ...
        pass
```

csrc/libtorch_stable/kimi_k3/attn_res_kernel.cu

NVIDIA SM100 专属 AttnRes 高性能 CUDA 内核，warp-specialized 设计。

```
/*
 * Production AttnRes forward for Blackwell (SM100).
 * Warp-specialized online softmax + residual + RMSNorm:
 * - 1 producer warp issues cp.async.bulk row loads into shared memory.
 * - 8 consumer warps compute reductions and output.
 * - Q = res_weight * rms_weight remains in registers across persistent tokens.
 * - V rows are converted once and cached as FP32 in TMEM between passes.
 * Integration contract: Kimi K3 H=7168, 1<=num_blocks<=8, token-major block residual
 storage.
 */

#include "../torch_utils.h"
#include <cfloat>
#include <cstdint>
#include <cuda_runtime.h>
#include <type_traits>

using bf16_t = __nv_bfloat16;

namespace sm100 {
namespace fwd_prod_v2 {

constexpr int K_TILE = 1024;
constexpr int N_CHUNK_DEFAULT = 4;
constexpr int CHUNK_DEPTH = 2;
constexpr int BLK = 288; // 1 producer + 8 consumer warps
constexpr int CONSUMER_THREADS = BLK - 32; // 256
constexpr int CONSUMER_WARPS = CONSUMER_THREADS / 32;
constexpr int CONSUMER_GROUPS = 2;
constexpr int CONSUMER_THREADS_PER_GROUP = CONSUMER_THREADS / CONSUMER_
GROUPS;
constexpr int FIRST_USER_NAMED_BARRIER = 8;

__device__ __forceinline__ const bf16_t* residual_addr(
    const bf16_t* block_res, const bf16_t* layer_res, int source, int N,
    int token, int block_stride_m, int block_stride_r, int H) {
```

```
    if (source < N - 1) {
        return block_res + static_cast<long long>(token) * block_stride_m +
               source * block_stride_r;
    }
    return layer_res + static_cast<long long>(token) * H;
}
// ... mbarrier helper functions omitted for brevity
} // namespace fwd_prod_v2
} // namespace sm100
```

评论区精华

核心讨论来自 Issue 评论（namgyu-youn 提问，gau-nernst 回应），涉及 SM100 内核硬编码行步幅假设的潜在问题。评论指出内核使用 H 硬编码作为 layer_res/delta/output 的行步幅，但只检查了最后一个维度的 stride 为 1，未检查 stride(0) == hidden_size，可能导致 padded input 时结果错误。作者承认这个假设，并同意添加 STD_TORCH_CHECK 加强验证，但当前优先合并主 PR，后续由其他贡献者修复。此外，reviewer zyongye 批准了该 PR。

- SM100 内核硬编码行步幅假设的潜在风险 (correctness): 作者同意需要检查，但在当前 PR 中 deferred，计划在 #50000 合并后单独修复。

风险与影响

- 风险：
  - 行步幅假设风险（csrc/libtorch_stable/kimi_k3/attn_res_kernel.cu）：CUDA 内核假定输入行步幅等于隐藏层大小，即输入张量必须是连续非 padding 的。如果 K3 模型传递了 padded 的行，结果会不正确。目前未添加显式检查，可能导致静默错误。后续应增加 STD_TORCH_CHECK。
  - 平台依赖风险：NVIDIA SM100 内核仅适用于 Blackwell 架构，在不支持的设备上需依靠 Triton fallback。Triton fallback 的性能可能不如定制内核，需确认 fallback 是否按预期启用。
  - 测试覆盖：测试覆盖了多种场景，但未包含极端边界情况（如极小 hidden_size、非 7168 隐藏层）。AMD 测试参数化较好，但缺少 block_write 场景。
  - CI 新增：Buildkite CI 配置新增可能导致 CI 时间增加，但这是预期内的。
- 影响：
  - 用户影响：Kimi-K3 模型用户将获得专用的 AttnRes 高性能内核，提升推理速度。SM100 用户获得最大优化，其他 NVIDIA GPU 和 AMD GPU 用户通过 Triton 获得功能支持。
  - 系统影响：新增约 1.7k 代码，包括 CUDA 和 Triton 内核。CI 配置变更增加测试覆盖，确保模型稳定性。
  - 团队影响：该 PR 为 Kimi-K3 模型支持的一部分（关联 #50000），后续将集成到模型正向传播中。需要持续维护和修复潜在 bug（如 stride 假设）。
  - 风险标记：行步幅假设，缺乏 stride 检查，仅 Blackwell 专属，Triton 性能回退

关联脉络

- PR #50131 [Bugfix] Add missing vllm/models/kimi_k3/__init__.py: 同属 Kimi-K3 模型支持系列，本 PR 添加了内核和测试，而 #50131 补充了缺失的包初始化文件。