PR #34490 完整报告

sgl-project/sglang

[AMD] Add Radix-4 MoE top-k router kernel for Kimi-K3 routing

合并时间：2026-08-23 14:57

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/34490

执行摘要

- 一句话：Kimi-K3 新增 ROCm Radix-4 MoE top-k 路由内核
- 推荐动作：值得精读，重点关注三点： 1) “并列契约”设计——两个实现按批大小分工时如何保证行为一致（复刻 + 对拍钉死），是自定义内核替换第三方库时的通用范式； 2) 门控分层——服务侧 available() 吞编译失败保可用性，测试侧 supported_hardware() + build() 让编译失败变红色，同一内核两种门各有取舍； 3) 内核侧 radix-4 直方图 + DPP 前缀和 + ballot 的组合，以及“轮数随 key 位宽而非 topk”的复杂度论证。若团队在 AMD 上跑 Kimi-K3，建议开启 SGLANG_K3_RADIX4_TOPK，并在升级 aiter 时留意对拍测试是否变红。

功能与动机

Kimi-K3 每层从 896 个专家中无分组选出 16 个，在 ROCm 上落入 aiter 的通用 biased_grouped_topk 路径——该路径每选一个专家就花一轮扫描，耗时随 topk 线性增长，MI355X 上每层约 10.4us；而 aiter 更快的 pivot 路径被 DeepSeek 的精确形状（256 专家、8 组、top-8）门控，K3 永远够不到。PR 同时提出正确性层面的动机：prefill 大批量仍走 aiter、decode 小批量走自研内核，若两者在分数并列时选出不同专家，同一 token 的路由会随出现阶段变化——PR 正文明确写到“If the two implementations pick different experts on a tie, the same token would get routed to different experts purely because it showed up during prefill versus decode——prefill and decode routing would disagree”。因此新内核必须逐列复刻 aiter 的行为。

实现拆解

实现按 5 步拆解：

1. HIP 内核本体（python/sglang/kernels/jit/csrc/moe/route_radix4_hip.cuh，+527 行）。一个 block 路由一个 token（256 线程），每个线程把 896 个分数切片放进寄存器；每轮用 16-bin 直方图（nibble 打包进一个 64 位寄存器）解析 4 个关键位，从高 bin 向下累加定位第 k 个 key 所在 bin，总轮数只与 key 位宽相关。跨 lane 聚合用 4 步 DPP 前缀和加一次 ballot（wave_sum_dpp）替代 16 步 bin 遍历；命中专家的合并写入存在竞争，但收尾阶段会按排名重排输出，竞争不影响结果。契约细节：NaN 排名 key 固定为 0（垫底永不入选）、-0.0 与 +0.0 映射同值、sigmoid 使用 aiter 的 exp2f(-log2e * x) 近似保证 ULP 级一致、并列按 kAiterTieLaneRank 表分离、输出权重不含 bias、renormalize 对非正和做保护。文件顶部 #ifndef USE_ROCM #error 强制 CDNA 专属。

2. JIT 封装与形状门控（python/sglang/kernels/ops/moe/moe_route_radix4.py，+137 行）。supported_hardware() 只认 gfx942/gfx950；build() 用 load_jit 编译并以 @cache_once 缓存，显式不开启 fast-math 以保证与 aiter 可比；available() 在服务侧吞掉编译失败（宁可回退 aiter 也不拒绝启动）；covered() 是窄形状门：[M, 896]、行连续、top-16、无分组、M <= 1024、bf16/fp32。这里刻意区分“服务门”与“测试门”：测试用 supported_hardware() + build()，让内核编译失败变成红色失败而不是跳过。
3. 服务端分发接入（python/sglang/srt/layers/moe/topk.py +18/-2，python/sglang/srt/environ.py +2）。在 biased_grouped_topk_gpu 的 aiter 分支里，先把 correction_bias 预转成 gating dtype、routed_scaling_factor 归一为 1.0；当 SGLANG_K3_RADIX4_TOPK 开启且 available() && covered() 时直接返回 route_radix4(...)，否则原样走 aiter_biased_grouped_topk。environ.py 在 Kimi-K3 环境变量组注册 SGLANG_K3_RADIX4_TOPK = EnvBool(False)，默认关闭，零默认行为变化。
4. 双参考测试与 CI 注册（test/registered/kernels/ops/moe/test_moe_route_radix4.py，+307 行）。测试文件自行解释了“两种参考”：纯 torch fp32 oracle 定义契约（bias 只参与排名、NaN 永不赢、并列顺序、renormalize 守卫、胜者按排名降序输出），覆盖形状与边界（全平局行、NaN、饱和 sigmoid、全 0 行、padding 非连续 stride、图回放、重复执行逐位可复现、covered() 门控正反例）；真正的 aiter 则用于逐列对拍（test_route_radix4_matches_aiter、test_route_radix4_tie_order_is_aiters、test_route_radix4_tie_straddling_the_cutoff_matches_aiter），并刻意在测试里保留 _TIE_LANE_RANK 第二副本直接钉住 aiter 侧行为。测试通过 register_amd_ci 注册进 MI35X 的 stage-b 套件（约 30 秒）。
5. 验证数据。gsm8k 1319 题准确率 0.952；MI355X 图捕获内核耗时：M=1 时 10.24us -> 5.99us（1.71x）、M=1024 时 11.09us -> 9.44us（1.17x）、M=1536 时 12.08us -> 11.95us（1.01x，接近拐点）；端到端 TTT 提升 1-3%。

关键文件：

- python/sglang/kernels/jit/csrc/moe/route_radix4_hip.cuh（模块 路由内核；类别 other；类型 entrypoint；符号 RouteRadix4Kernel, tie_priority, tie_rank, sortable）：527 行 HIP 内核本体，radix-4 选择算法、kAiterTieLaneRank 并列顺序、NaN/ 符号位排序 key、DPP 前缀和聚合全部在此，是本 PR 的核心实现单元。
- python/sglang/kernels/ops/moe/moe_route_radix4.py（模块 内核封装；类别 infra；类型 infrastructure；符号 supported_hardware, build, available, covered）：JIT 封装与门控层，supported_hardware/build/available/covered 是服务端分发的安全边界，也是“服务侧吞失败、测试侧显失败”双门控哲学的载体。
- python/sglang/srt/layers/moe/topk.py（模块 路由分发；类别 source；类型 dependency-wiring；符号 biased_grouped_topk_gpu）：服务端 MoE 路由入口 biased_grouped_topk_gpu 的分发接入点，决定何时走 radix4、何时回退 aiter，是优化真正落到线上的最后一段管线。
- test/registered/kernels/ops/moe/test_moe_route_radix4.py（模块 契约测试；类别 test；类型 test-coverage；符号 _tie_priority, TIE_PRIORITY, _oracle, _assert_matches_oracle）：307 行双参考测试：纯 torch fp32 oracle 定义契约，真 aiter 对拍钉死并列顺序，并把 _TIE_LANE_RANK 第二副本直接暴露在测试里，是本 PR

正确性保障的核心。

- python/sglang/srt/environ.py（模块 环境开关；类别 source；类型 configuration；符号 SGLANG_K3_RADIX4_TOPK）：新增默认关闭的 SGLANG_K3_RADIX4_TOPK 开关，保证默认零行为变化，是“可选性能路径”与“默认安全”之间的边界。

关键符号：route_radix4, covered, available, build, supported_hardware, tie_priority, rank_key, sortable, wave_sum_dpp, stage_wave_sum, _tie_priority, _oracle, _assert_matches_oracle, biased_grouped_topk_gpu

关键源码片段

python/sglang/kernels/jit/csrc/moe/route_radix4_hip.cuh

527 行 HIP 内核本体，radix-4 选择算法、kAiterTieLaneRank 并列顺序、NaN/ 符号位排序 key、DPP 前缀和聚合全部在此，是本 PR 的核心实现单元。

```
// 内核契约（摘自 route_radix4_hip.cuh 顶部注释）：
// - 一个 block 路由一个 token，每个线程把 896 个分数切片放进寄存器；
// - 每轮固定解析 4 个关键位：16-bin 直方图以 nibble 打包进一个 64 位寄存器，
// 从高 bin 向下累加定位第 k 个 key 所在的 bin，总轮数随 key 位宽而非 topk 增长；
// - 跨 lane 聚合用 4 步 DPP 前缀和加一次 ballot，替代 16 步 bin 遍历。
namespace sglang {

inline constexpr uint32_t kRadix4NumExperts = 896;
inline constexpr uint32_t kRadix4TopK = 16;
inline constexpr uint32_t kRadix4Block = 256; // 每 token 一个 block
inline constexpr uint32_t kRadix4Wave = 64; // wave64 专用

// aiter 用 exp2f(-log2e * x) 近似 sigmoid 而非 expf(-x)；
// 这里逐位对齐 topk_softmax_kernels_group.cu 的 C_LOG2E，保证与 aiter 的 ULP 级一致。
inline constexpr float kAiterSigmoidLog2E = 1.44269504088896340736f;

// aiter 路由器的 wave64 遍历顺序：两个排名值完全相同的专家按该顺序分离。
// 该表通过与 aiter 逐项比对读出，并由 test_moe_route_radix4 回钉，
// aiter 侧一旦变化会以测试失败暴露，而不是静默漂移。
static __device__ __constant__ uint8_t kAiterTieLaneRank[64] = {
    56, 57, 58, 59, 63, 62, 61, 60, 52, 53, 54, 55, 51, 50, 49, 48,
    40, 41, 42, 43, 47, 46, 45, 44, 36, 37, 38, 39, 35, 34, 33, 32,
    24, 25, 26, 27, 31, 30, 29, 28, 20, 21, 22, 23, 19, 18, 17, 16,
    8, 9, 10, 11, 15, 14, 13, 12, 4, 5, 6, 7, 3, 2, 1, 0,
};

// 专家在 aiter 遍历顺序中的位置：到 [0, 896) 的双射。
// 专家每 4 个一组落在一个 lane 上，224 组循环分配：lane 0-31 各 4 组（bank 0-3）、
// lane 32-63 各 3 组（bank 0-2），因此 rank < 32 时乘 3、否则乘 4。
SGL_DEVICE uint32_t tie_priority(int expert) {
    const int group = expert >> 2;
    const int lane = group & 63;
    const int bank = group >> 6;
```

```
    const int rank = static_cast<int>(kAiterTieLaneRank[lane]);
    assert((rank < 32) == (lane >= 32));
    const int group_rank = (rank < 32) ? (rank * 3 + bank) : (96 + (rank - 32) * 4 + bank);
    return static_cast<uint32_t>((group_rank << 2) + (expert & 3));
}

// 单调 float -> uint32 映射，用无符号比较给浮点排序。
// 映射永不返回 0：负数得到 ~u（仅全 1 的 NaN 为 0），正数带符号位；
// key 0 因此可以专用于表示"NaN，排在一切数值（含 -inf）之下"。
SGL_DEVICE uint32_t sortable(float f) {
    uint32_t u = __float_as_uint(f);
    if (u == 0x80000000u) u = 0u; // 把 -0.0 与 +0.0 映射到同一值
    return (u & 0x80000000u) ? ~u : (u | 0x80000000u);
}

// 排名 key：NaN 为 0，保证它永远不能挤掉排名值为数值的专家。
SGL_DEVICE uint32_t rank_key(float x) {
    return (x == x) ? sortable(x) : 0u;
}

// 波内 inclusive 前缀和：每级把相邻 lane 的 N 个 uint32 相加，
// 6 级 row_shr/row_bcast 之后 lane L 得到 0..L 的和，lane 63 得到波内总和。
// row_shr:4 / row_shr:8 上的窄 bank 掩码恰好关掉源 lane 落在 row 外的级，
// 那些级本应加 0，掩不掩码结果相同。
template <int N>
SGL_DEVICE void wave_sum_dpp(uint32_t (&x)[N]) {
    dpp_add_stage<0x111, 0xf, 0xf>(x); // row_shr:1
    dpp_add_stage<0x112, 0xf, 0xf>(x); // row_shr:2
    dpp_add_stage<0x114, 0xf, 0xe>(x); // row_shr:4
    dpp_add_stage<0x118, 0xf, 0xc>(x); // row_shr:8
    dpp_add_stage<0x142, 0xa, 0xf>(x); // row_bcast:15
    dpp_add_stage<0x143, 0xc, 0xf>(x); // row_bcast:31
}

} // namespace sglang
```

python/sglang/kernels/ops/moe/moe_route_radix4.py

JIT 封装与门控层，supported_hardware/build/available/covered 是服务端分发的安全边界，也是"服务侧吞失败、测试侧显失败"双门控哲学的载体。

```
# K3 路由形状常量：896 专家、top-16。
_NUM_EXPERTS = 896
_TOPK = 16
# 每 token 一个 block，token 数超过约 1.5k 后 grid 会占满整机，
# 把 1 个 token 摊到 4 个 wave 上反而更慢；实测拐点约 1.5k，
# 1k 以下仍领先 1.2 倍以上，prefill 规模的批则远超上限留在 aiter。
_MAX_TOKENS = 1024

def supported_hardware() -> bool:
```

```
    """内核只面向 gfx942/gfx950：全程 wave64 与 GFX9 DPP。"""
    if not is_hip_runtime() or not torch.cuda.is_available():
        return False
    gcn_arch = torch.cuda.get_device_properties(0).gcnArchName
    return any(arch in gcn_arch for arch in ("gfx942", "gfx950"))


@cache_once
def build() -> Module:
    """编译并加载内核；工具链失败时直接抛错（测试需要失败可见）。"""
    return load_jit(
        "moe_route_radix4",
        cuda_files=["moe/route_radix4_hip.cuh"],
        cuda_wrappers=[("run", "RouteRadix4Kernel::run")],
        # 显式不开 fast-math：专家 id 的选择在并列与 NaN 下必须与 aiter 可比。
        extra_cuda_cflags=["-O3"],
    )


@cache_once
def available() -> bool:
    """服务侧门控：编译失败被吞掉，宁可回退 aiter 也不拒绝启动。
    这与测试侧故意相反——测试用 supported_hardware() + build()，
    让编译失败成为红色失败而不是跳过。"""
    if not supported_hardware():
        return False
    try:
        build()
        return True
    except Exception as e:
        logger.warning(f"Failed to load the JIT ROCm radix router: {e}")
        return False


def covered(scores, bias, topk, num_expert_group, topk_group) -> bool:
    """只覆盖 K3 路由形状：[M, 896] 行连续 scores、top-16、无分组、M <= 1024。
    分组路由直接排除而不是模拟：内核同时给全部 896 个专家排名，
    没有先把整组屏蔽掉的概念。"""
    return (
        scores.dim() == 2
        and scores.size(0) <= _MAX_TOKENS
        and scores.size(1) == _NUM_EXPERTS
        and int(topk) == _TOPK
        and scores.dtype in (torch.bfloat16, torch.float32)
        and bias.dtype == scores.dtype
        and scores.stride(1) == 1
        and bias.is_contiguous()
        and (num_expert_group or 1) == 1
        and (topk_group or 1) == 1
```

```
    )


def route_radix4(scores, bias, topk, renormalize, routed_scaling_factor):
    """返回 (weights [M, topk] fp32, ids [M, topk] int32)；调用方必须已检查 covered()。
    专家按 sigmoid(score) + bias 排名，但输出的权重是不含 bias 的 sigmoid；
    并列时按 aiter 的遍历顺序（kAiterTieLaneRank）区分，保证与 aiter 列对列一致，
    decode 与 prefill 的路由不会因批大小不同而分叉。"""
    M = scores.shape[0]
    out_w = torch.empty((M, topk), dtype=torch.float32, device=scores.device)
    out_i = torch.empty((M, topk), dtype=torch.int32, device=scores.device)
    build().run(
        scores,
        bias,
        out_w,
        out_i,
        topk,
        float(routed_scaling_factor),
        bool(renormalize),
    )
    return out_w, out_i
```

test/registered/kernels/ops/moe/test_moe_route_radix4.py

307 行双参考测试：纯 torch fp32 oracle 定义契约，真 aiter 对拍钉死并列顺序，并把 _TIE_LANE_RANK 第二副本直接暴露在测试里，是本 PR 正确性保障的核心。

```
NUM_EXPERTS = 896
TOPK = 16
# 内核把 NaN 的 key 打成 0，排在一切数值（含 -inf）之下；
# 这里用 -inf 代替 NaN 参与 oracle 排名。
NAN_RANK = float("-inf")

# kAiterTieLaneRank 刻意在测试里再保留一份：
# test_route_radix4_tie_order_is_aiters 直接拿它与 aiter 对拍，
# aiter 侧一旦变化，这里失败而不是在内核里静默漂移。
_TIE_LANE_RANK = [
    56, 57, 58, 59, 63, 62, 61, 60, 52, 53, 54, 55, 51, 50, 49, 48,
    40, 41, 42, 43, 47, 46, 45, 44, 36, 37, 38, 39, 35, 34, 33, 32,
    24, 25, 26, 27, 31, 30, 29, 28, 20, 21, 22, 23, 19, 18, 17, 16,
    8, 9, 10, 11, 15, 14, 13, 12, 4, 5, 6, 7, 3, 2, 1, 0,
] # fmt: skip


def _tie_priority(expert):
    """aiter 的 wave64 遍历到达某专家的位置：到 [0, 896) 的双射。"""
    group = expert >> 2
    lane, bank = group & 63, group >> 6
    rank = _TIE_LANE_RANK[lane]
    group_rank = rank * 3 + bank if rank < 32 else 96 + (rank - 32) * 4 + bank
```

```
    return (group_rank << 2) + (expert & 3)


TIE_PRIORITY = [_tie_priority(e) for e in range(NUM_EXPERTS)]
assert len(set(TIE_PRIORITY)) == NUM_EXPERTS, "tie priority is not a permutation"

# 按并列顺序排好专家列，稳定降序排序后相同值自然按该顺序落位。
_BY_PRIORITY = torch.tensor(
    sorted(range(NUM_EXPERTS), key=TIE_PRIORITY.__getitem__), device="cuda"
)


def _oracle(scores, bias, renormalize, scaling):
    """契约（见 route_radix4_hip.cuh 注释）：bias 只参与排名、输出权重不含 bias；
    NaN 排名垫底永远不赢；并列走 aiter 遍历顺序；
    renormalize 先对胜者权重求和，和为非正时受保护为 1 再做缩放。
    胜者按排名值从高到低输出，同值按同一并列顺序。"""
    s = torch.sigmoid(scores.float())
    biased = s + bias.float()
    biased = torch.where(torch.isnan(biased), torch.full_like(biased, NAN_RANK), biased)
    by_priority = biased[:, _BY_PRIORITY]
    picked = torch.argsort(by_priority, dim=-1, descending=True, stable=True)[:, :TOPK]
    ranked = _BY_PRIORITY[picked]
    w = s.gather(1, ranked)
    if renormalize:
        total = w.sum(-1, keepdim=True)
        w = w / torch.where(total > 0, total, torch.ones_like(total))
    return w * scaling, ranked.to(torch.int32)


def _assert_matches_oracle(scores, bias, renormalize=True, scaling=2.5):
    ref_w, ref_ids = _oracle(scores, bias, renormalize, scaling)
    w, ids = moe_route_radix4.route_radix4(scores, bias, TOPK, renormalize, scaling)
    # 逐列比对：胜者落位也是契约的一部分。
    assert torch.equal(ids, ref_ids)
    # 内核 sigmoid 是硬件近似序列（与 aiter 对齐），末位与 torch 精确 sigmoid 不同。
    torch.testing.assert_close(w, ref_w, rtol=1e-5, atol=1e-6)
```

评论区精华

本 PR 没有产生实质 review 评论：HaiShaw 以空正文 APPROVED，唯一介入是 issue 里的 /tag-and-rerun-ci 触发重跑。最有价值的“讨论”沉淀在 PR 正文与 commit 演进中：

- 并列契约是核心设计决策。正文指出两套实现按批大小分工，若并列打破规则不同就会导致 prefill/decode 路由分叉；aiter 的并列顺序来自其内部 wave64 max 归约的硬件比较次序，与 expert id 无直接关系，只能读出无法推导。因此 kAiterTieLaneRank 是“read off directly by comparison against aiter”，并用 test_route_radix4_tie_order_is_aiters 对拍，保证“a change on aiter's side surfaces as a test failure rather than a silent

divergence”。

- M <= 1024 上限被论证为性能门而非正确性门。每 token 一个 block，超过约 1.5k token 后 grid 占满整机，把单个 token 摊到 4 个 wave 的成本大于收益；decode 批远低于上限，prefill 块远高于上限并留在 aiter。
- commit 演进能看到方向校准：a6cdd1a 先按 route_radix.cuh 的契约把 NaN 垫底、并列给最低 expert id；随后 155a747（Match aiter's tie order）与 dff46e4（match aiter's approximate sigmoid）两次提交把策略从“自己定义并列规则”修正为“逐位对齐 aiter”，这是本 PR 最重要的设计转向。
- 并列打破顺序：decode 与 prefill 路由必须一致 (design): 内核通过 kAiterTieLaneRank 表复刻 aiter 遍历顺序，并在测试中保留第二副本直接与真 aiter 对拍（test_route_radix4_tie_order_is_aiters等），aiter变化会以测试失败暴露而不是静默漂移。
- M <= 1024 的 token 上限是性能门而非正确性门 (performance): 接受上限并纳入 covered() 门控，超出即静默回退 aiter；正确性不受影响，性能拐点在 M=1536 处（1.01x）得到实测印证。
- CI 重跑与 AMD 并行 Run 状态 (other): 以重跑加审批完成合入；AMD ROCm 7.2 路的失败未阻塞合入，提示该环境存在不稳定因素，相关风险由注册测试承担可见性。

风险与影响

- 风险： 1) 对 aiter 内部行为的强依赖：并列遍历顺序与近似 sigmoid 都是从 aiter 逐项读出而非推导，aiter 升级若改变这些细节，会以测试失败暴露，但依赖 AMD CI 对 gfx942/gfx950 的持续覆盖。 2) 数值近似差异：内核 sigmoid 与 torch 精确实现存在 ULP 级差异，测试以 rtol=1e-5、atol=1e-6 容忍；该容差伴随路由权重传播，极小概率下可能影响选路结果。 3) 门控回退：covered() 要求严格（M <= 1024、行连续、bf16/fp32、无分组），条件不满足静默回退 aiter，行为正确但性能预期可能落空（例如 decode 批超过 1024 时）。 4) 编译失败路径：服务侧 available() 吞掉 JIT 编译异常并回退 aiter，若内核在目标硬件上编译失败，用户只看到 warning，性能优化静默失效。 5) CI 观察：PR 关闭时 PR Test (Base) 为绿，但 PR Test (Extra) 与 AMD ROCm 7.2 两个并行 Run 显示失败，经 /tag-and-rerun-ci 后由 HaiShaw 合入，提示 AMD CI 环境存在不稳定因素。 6) 默认关闭：该路径默认在常规 CI 覆盖之外，回归保护完全依赖注册测试兜底。
  - 影响：用户侧：仅 AMD CDNA（gfx942/gfx950）+ Kimi-K3 且显式设置 SGLANG_K3_RADIX4_TOPK=1 的用户受影响，decode 批路由内核提速 1.2-1.9 倍，端到端 TTT 提升 1-3%，gsm8k 准确率保持 0.952；其余平台与形状零变化。系统侧：MoE 路由入口（python/sglang/srt/layers/moe/topk.py 的 aiter 分支）新增一条 env 门控捷径，默认关闭、回退路径完整保留；JIT 编译缓存与软失败机制避免启动期故障。团队侧：新增约 30 秒的 AMD MI35X 注册测试（stage-b-test-1-gpu-small-amd-mi35x），测试对 aiter 行为强依赖，aiter 升级需同步关注对拍用例；同时为后续 AMD 定制内核确立了“契约 oracle + 真实实现对拍”的测试范式与“available()/build() 双门控”的运行范式。
  - 风险标记：默认关闭需显式开启，对齐 aiter 内部行为存在漂移风险，仅 gfx942/gfx950 可用，AMD CI 并行 Run 有失败记录，批量超 1024 时性能回退 aiter

关联脉络

- PR #36004 [AMD][DSV4] perf: use full 1024-thread block for indexer top-k on ROCm: 同为 AMD/ROCm 侧 MoE top-k 路由相关内核的性能优化（sgl-kernel JIT + 基准测试 / 测试保护），与本 PR 共享"ROCm 专属 top-k 内核"技术路线。
- PR #35508 [NPU] [DOC] Add Ascend NPU (A3) recipe to the Kimi-K3 cookbook: 同一 Kimi-K3 支持线的配套 PR（NPU 部署文档），显示 K3 在 SGLang 多硬件平台的持续演进；本 PR 是 AMD 侧的性能拼图。