PR #32890 完整报告

sgl-project/sglang

feat(kernels): port standalone Kimi K3 kernels

合并时间：2026-08-01 13:26

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/32890

执行摘要

- 一句话：移植 Kimi K3 独立内核与四组硬件测试，为 Blackwell 集成铺路
- 推荐动作：值得精读，尤其适合内核与 JIT 基础设施维护者。建议重点看：(1) PR body 的移植边界切割与 #32624 的 kernel/runtime 分离策略；(2) chunk_fwd.py 的缓存设计（用 data_ptr/shape/stride 做 key 规避 id 复用陷阱、消除 GPU->CPU sync 的 50-200 us stall）；(3) review 中 AOT vs JIT、PTX helper 组织、拷贝策略 benchmark 的讨论；(4) “恢复损失性导出”的逐文件审计方法，对任何跨分支移植都有借鉴价值。一般服务端工程师只需阅读执行摘要与风险章节。

功能与动机

Kimi K3 是面向 Blackwell（SM100/SM103）的高性能 Delta Attention 模型，其 Day0 工作维护在独立 kimi-k3 分支上。PR body 的 Motivation 明确："Port the standalone kernels from the Kimi K3 Day0 work to main first. Keeping the kernels and their direct tests separate makes the reusable pieces reviewable before the more invasive model, scheduler, and serving integration changes." 即先让内核与测试以独立、可审查的方式进入 main，把侵入式集成推迟到后续 PR，从而降低单次合入的审查与回归风险。

实现拆解

1. 界定移植边界：先通过 docs 系列提交（design/plan/inventory）盘点 kimi-k3 分支的内核清单，只抽取可独立审查的 kernel 层与直接测试，明确排除模型（如 srt/models/kimi_k3.py）、调度器、投机解码运行时、多模态处理器与服务端集成；同时移除 tuner/benchmark 脚手架和 Day0 runtime dispatch 代码。
2. 共享 JIT/TMA 基础：新增 sglang/kernels/jit 下的通用 JIT 支持（make_cpp_args、cache_once、PDL 架构探测等），把 K3 collectives 需要的系统级 PTX helper 收敛到 csrc/kimi_k3/comm/ptx_sys.cuh（保持 device::distributed 命名空间），并在 warp.cuh 中提供 warp::copy_bytes 设备拷贝原语（对应 review 中的 g2s_copy 改名与基准验证）。
3. KDA 内核家族移植：kda_nvidia_prefill/ 下 vendor NVIDIA KDA_prefill 的 K1+K2+K3 融合 persistent kernel（fuse_kernel123_persistent.py）、K4-only persistent varlen kernel（fuse_k4_only_persistent.py）、Akk 下三角块求逆（Akk_inverse_lower_triangle_bf16.py）与编排层 chunk_fwd.py；decode 侧新增 kda_fused_decode.py、kda_packed_decode.py、kimi_k3/kda_decode_mtp.py；投机解码侧新增 attention/fla/kda_replayssm_spec_decode.py，用 exact-fold 取代全量 SSM 快

照。attention/fla/kda.py 是本次唯一修改的存量文件，为 K3 增加 recompute_w_u_fwd_kernel 配置分支与预编译入口。

4. 计算与量化内核：移植 SiTU 激活（kimi_k3/activation.py）、attn_res 融合 TMA 聚合（kimi_k3/attn_res.py）、MLA KV 融合写路径（set_mla_kv_concat_q.py）、视觉预处理（mm/process/image.py 的 normalize_and_patchify）、tiny_gemm.py 与 vision_rope.py 等通用前置内核。
5. 多卡 collectives：新增 kimi_k3/all_reduce.py（push/pull 两套算法 × res/norm 两种 epilogue）、gemm_ar.py/gemm_ag.py（GEMM 与 all-reduce/all-gather 融合）、sp_collective.py（SP-MoE reduce-scatter/all-gather，JSON 调优表驱动 dispatch）。所有入口通过 register_comm 绑定 CustomAllReduceV2 存储面，并断言同一 world_size 只允许一个 communicator。
6. 测试与 CI：保留四个硬件分组注册测试入口（H100 prerequisite、B200 compute、B200 collective、Blackwell prefill correctness）；H100 侧 9 个保留测试全部通过（12.91 s），13 个 Blackwell-only 测试方法靠架构守卫在 H100 上跳过，SM100/SM103 专属测试交由 B200/GB300 CI 执行。
7. 审查修复与清理：根据 review 回退 communicator.cuh 至 pre-PR 状态、删除 align_single_token（并记录 kimi-k3 接缝）、合并重复 device helper、去除 vendor 文件内嵌的 main/bench harness、按 kimi-k3 逐文件审计恢复所有 lossy exports（PDL、radix-select 快速路径、v2p_ptr/PAGE_MULT 等），并移植 #32624 的 TP16/TP32 fused KDA decode kernel 半部分。

关键文件：

- python/sglang/kernels/ops/attention/linear/kda_nvidia_prefill/fuse_kernel123_persistent.py（模块 预填内核；类别 source；类型 core-logic；符号 k1_internal_barrier, pack_bf16x2_f32, mma_bf16_m16n8k16, fast_rcp）：KDA prefill 的核心实现：K1+K2+K3 融合 persistent kernel，1024 线程、148 个驻留 block，warp-specialized mbarrier 流水，并首次引入防止 LICM hoisting 的 opaque_zero_from_work_id 技巧，是 Blackwell prefill 正确性与性能的关键证据。
- python/sglang/kernels/ops/attention/linear/kda_nvidia_prefill/fuse_k4_only_persistent.py（模块 预填内核；类别 source；类型 core-logic；符号 _DummyExperimentalDSL, transform_partitioned_tensor_layout, _patched_get_pipeline, k4_persistent_kernel）：K4-only persistent varlen kernel：基于 GDNTileScheduler 的 3D tile 调度与 TensorMapManager 每 tile TMA 描述符更新，每个 chunk 执行 6 个 MMA；还包含对 CuTeDSL pipeline 注入 ptx-options='--uumn' 的运行时补丁。
- python/sglang/kernels/ops/attention/linear/kda_nvidia_prefill/chunk_fwd.py（模块 预填内核；类别 source；类型 entrypoint；符号 _ct, _get_eqlen_dummies, _cute_int_type, _tkey）：KDA chunk forward 的编排入口，把 K123、K4 与 Akk 求逆串成 FLA 兼容接口；集中体现了服务安全设计：用 (data_ptr, shape, stride, dtype) 做 tensor 缓存 key，避免 id 复用陷阱，并把 varlen K4 预处理全部放到 GPU 侧以消除 50-200 us 的 GPU->CPU stall。
- python/sglang/kernels/ops/kimi_k3/kda_decode_mtp.py（模块 解码内核；类别 source；类型 core-logic；符号 _stream_state, _p2_lanes, _qk_smem, _state_tile_v）：KDA

conv-MTP decode 内核：CuTe DSL 编写，按一次 wave 内 / 外动态选择 256/512 block 线程，phase 2 在“状态驻留寄存器”与“逐 tile 流式”两种策略间切换，刻画了多 tokenverify 与单 token decode 的差异权衡。

- python/sglang/kernels/ops/attention/fla/kda_replayssm_spec_decode.py（模块 投机解码；类别 source；类型 core-logic；符号 kda_replayssm_exact_fold_kernel, commit_kda_replayssm_spec, commit_kda_replayssm_spec_all_layers, commit_kda_replayssm_after_verify）：ReplaySSM 投机解码状态提交的替代方案：用 per-request 小输入窗口 + 提交时 exact-fold 取代全量 SSM 快照缓存，把 dspark 并发的内存大户缩小为常数量级；kernel 与 recurrent baseline 逐位一致，附有明确的“不要重排 / 不要改 num_warps”契约。
- python/sglang/kernels/ops/kimi_k3/all_reduce.py（模块 集体通信；类别 source；类型 entrypoint；符号 _jit_module, _CommEntry, register_comm, PullTuning）：K3 MNNVL fused all-reduce 的 Python JIT 封装：push（1shot 多播）与 pull（2shot NVLS）两种算法族 × res/norm 两种 epilogue，复用 CustomAllReduceV2 存储面，register_comm 对同一 world_size 的重复注册做断言防呆。
- python/sglang/kernels/ops/kimi_k3/sp_collective.py（模块 集体通信；类别 source；类型 core-logic；符号 Tuning, Dispatch, FusionDispatch, _device_name）：SP-MoE 的 reduce-scatter / all-gather 内核封装：JSON 调优表按 (world_size, hidden_size, device_name) 精确命中，按 num_tokens 分桶选择策略，策略为 nccl/separate 时优雅回退，是 K3 多卡 MoE 路径的通信基石。
- python/sglang/kernels/ops/attention/set_mla_kv_concat_q.py（模块 MLA 优化；类别 source；类型 core-logic；符号 set_mla_kv_concat_q_module, can_use_set_mla_kv_concat_q, _row_aligned, covered）：融合 MLA decode prepare tail：把 paged-KV scatter 与 absorb-q concat 合并为一次 launch，SM90+ 走 TMA bulk store；covered()/covered_fp8() 提供逐调用门控，不满足对齐或宽度约束时回退两内核路径，是运行时安全的典型范式。
- python/sglang/kernels/ops/attention/fla/kda.py（模块 预填内核；类别 source；类型 core-logic；符号 recompute_w_u_fwd_kernel, _recompute_w_u_fwd_kernel, precompile_k3_recompute_w_u_kernel, _get_k3_recompute_w_u_config）：本 PR 唯一修改的存量 kernel 文件：为 Kimi K3 的 recompute_w_u 前向增加专用配置与预编译入口，是后续集成直接接驳的适配点。

关键符号：fused_kernel123, make_host_function, opaque_zero_from_work_id, k4_persistent_kernel, chunk_kda_fwd, kda_decode_mtp_kernel, kda_replayssm_exact_fold_kernel, commit_kda_replayssm_spec, all_reduce_push_res, all_reduce_pull_norm, reduce_scatter_res, all_gather_direct, set_mla_kv_concat_q, covered, can_use_set_mla_kv_concat_q, attn_res_fused_tma, attn_res_fused_direct_ag, situ_and_mul, normalize_and_patchify, tiny_n_gemm_bf16

关键源码片段

python/sglang/kernels/ops/attention/linear/kda_nvidia_prefill/fuse_kernel123_persistent.py

KDA prefill 的核心实现：K1+K2+K3 融合 persistent kernel，1024 线程、148 个驻留 block，warp-specialized mbarrier 流水，并首次引入防止 LICM hoisting 的 opaque_zero_from_work_id 技巧，是 Blackwell prefill 正确性与性能的关键证据。

```
# fuse_kernel123_persistent.py —— KDA prefill 的 K1+K2+K3 融合 persistent kernel
# 从 NVIDIA KDA_prefill 包 vendor，供 Kimi-K3 chunked prefill 使用。
# 布局：grid = (NUM_SMS, 1, 1)，每 SM 一个驻留 block；work units 按
# i, i+NUM_SMS, i+2*NUM_SMS ... 轮转；1024 线程分为 K1+TMA（warp 0-15）、
# K2 MMA（warp 16-27）、Store/Inversion（warp 28-31）三组，组内各自计算
# 不变式，避免跨组寄存器压力。

# 关键几何常量：BT=64 子块、BC=16、K_DIM=128、K_STRIDE=136（pad 防 bank conflict）。
BT = 64
K_DIM = 128
K_PAD = 8
K_STRIDE = K_DIM + K_PAD # 136
NUM_SMS = 148 # persistent：每 SM 一个驻留 block
NUM_K1_TMA_WARPS = 16
NUM_MMA_WARPS = 11 # warp 26 是专用 TMA producer
NUM_STORE_WARPS = 4
NUM_WARPS = 32 # 1024 线程

# 具名 barrier：K1+TMA warp 组（0-15，512 线程）专用 barrier_id=2。
@dsl_user_op
def k1_internal_barrier(*, loc=None, ip=None):
    # membar.cta + bar.sync 2, 512；输出 0 只是满足内联汇编输出约束。
    llvm.inline_asm(
        T.i32(), [],
        "membar.cta; bar.sync 2, 512; mov.u32 $0, 0;",
        "=r", has_side_effects=True, is_align_stack=False,
        asm_dialect=llvm.AsmDialect.AD_ATT, loc=loc, ip=ip,
    )

# 把两个 fp32 打包成 bf16x2（高 16 位放 hi，低 16 位放 lo），
# 供 mma.sync.aligned.m16n8k16 的 A/B 操作数使用。
@dsl_user_op
def pack_bf16x2_f32(hi_f32, lo_f32, *, loc=None, ip=None):
    # cvt.rn.bf16x2.f32 -> d[31:16]=bf16(a), d[15:0]=bf16(b)
    result = llvm.inline_asm(
        T.i32(),
        [hi_f32.ir_value(loc=loc, ip=ip), lo_f32.ir_value(loc=loc, ip=ip)],
        "cvt.rn.bf16x2.f32 $0, $1, $2;",
        "=r,f,f", has_side_effects=False, is_align_stack=False,
        asm_dialect=llvm.AsmDialect.AD_ATT, loc=loc, ip=ip,
    )
    return cutlass.Int32(result)

# 关键优化：返回 0 的“不透明”操作。
# has_side_effects=True 让 MLIR LICM 认为它有内存副作用，不会把它
```

```
# 提升出 for_generate 循环；派生值因此对 LICM 呈现 loop-variant，
# 阻止 get_slice() 等标量布局不变计算被提升到 kernel prologue。
# 效果：prologue 寄存器压力 < 64，消除 440 字节栈帧及其每迭代
# 约 300 cycle 的 L2 LDL 惩罚。汇编体按语义精简。
@dsl_user_op
def opaque_zero_from_work_id(*, loc=None, ip=None):
    result = llvm.inline_asm(
        T.i32(), [], "mov.u32 $0, 0;", "=r",
        has_side_effects=True, is_align_stack=False,
        asm_dialect=llvm.AsmDialect.AD_ATT, loc=loc, ip=ip,
    )
    return cutlass.Int32(result)
```

评论区精华

审查交锋集中在五个问题上。一是 situ_and_mul.cuh 与 dsv4 silu_and_mul 的去重边界：DarkSharpness 认为两者 99% 相似，BBuf 论证 SiTU 对 gate 与 up 两个操作数都做变换、double softcap 本身就是 FP8 bound，参数化共享会让 dsv4 背负无谓的 beta 管道，最终保持独立。二是 align_single_token 的删除：DarkSharpness 先质疑其是否还有调用者，BBuf 一度恢复（kimi-k3 的 fused_marlin_moe.py 有内嵌 import），最终按作者要求删除并记录 handoff。三是 communicator.cuh 的 PTX helper 归属，最终回退文件并新建 ptx_sys.cuh。四是 warp.cuh 拷贝原语的命名与策略，BBuf 改名 copy_bytes 并给出 B300 实测基准。五是 kda_prefill.cu 的编译路线，确认需要 AOT 迁移。

- situ_and_mul 与 dsv4 silu_and_mul 的去重边界 (design): 保持独立文件，不做模板化合并。
- align_single_token bs=1 MoE 快速路径的去留 (design): kernel、Python wrapper 与测试全部删除；main 无消费者，kimi-k3 分支需配套跟进。
- communicator.cuh 保持原状，PTX helper 移到使用处 (design): communicator.cuh diff 归零，PTX helper 收敛到 ptx_sys.cuh。
- warp 协作拷贝的命名与性能策略 (performance): 保留通用向量拷贝策略并改名，benchmark 表写入头文件。
- gemm_ar.cuh 的错误检查宏 (correctness): 统一使用 host::CHECK_CUDA，错误路径行为一致。
- set_mla_kv_concat_q 的 TMA 发射线程选举 (performance): 使用文件内 elect.sync helper，不与 cute 耦合。
- kda_prefill.cu 的 JIT 编译与 AOT 迁移路线 (design): 同意 AOT，但本次合入前不迁移；需要在 follow-up PR 处理并发安全与 B200 镜像。
- flash_attention_v4.py 无关改动与 lossy exports 审计 (correctness): 恢复全部丢失导出，kernel 树与 kimi-k3 保证一致。

风险与影响

- 风险：
  1. Blackwell 验证缺口：SM100/SM103 专属测试只注册在 B200/GB300 CI，作者本地无 Blackwell 机器，H100 结果不能代表 Blackwell 正确性；kda_prefill.cu 经

torch.utils.cpp_extension.load 编译的产物只含 sm_103a，B200/GB200（cookbook 列出的目标）拿不到 kernel 镜像。

2. JIT 冷启动与并发：kda_prefill.cu 冷编译约 109.2 s、热编译 5.0 s，且 torch.utils.cpp_extension.load 的并发安全性未确认，多进程首启可能互相踩踏 ~/.cache/torch_extensions。
3. vendored 代码回归风险：大量 raw PTX/cutlass DSL 代码，提交记录已记载一次"编译通过但静默移动 expert id"的 radix 提取事故，任何改写都可能无声改变数值结果，Blackwell 回归必须靠 CI 兜底。
4. 分支同步风险：align_single_token 删除后，kimi-k3 分支 fused_marlin_moe.py:244 的内嵌 import 会在 Marlin + 单 token decode 时抛 ImportError（服务能启动、负载下才暴露），需要 kimi-k3 侧配套删除。
5. 缓存契约风险：chunk_fwd.py 的多级 tensor 缓存（_tkey/_ct_cached/_get_varlen_k4_inputs）依赖调用方复用同一 tensor 对象，虽然已用 data_ptr/shape/stride 加固，仍是后续运行时的隐性契约。 - 影响：影响面集中在 kernels 层与 JIT 基础设施：main 上现有模型 / 调度 / 服务路径没有新调用者（唯一修改的存量文件 fla/kda.py 只是为 K3 增加配置分支），线上行为不变。对团队而言这是一次约 2.6 万行内核代码的基础设施注入，后续 Kimi K3 集成 PR 将直接消费这批符号，kimi-k3 分支可逐步删除自己的内核副本；同时需要持续投入 B200/GB300 CI 资源维持硬件分组测试矩阵。对用户暂无直接可感知变化，Kimi K3 的服务化价值要等 runtime integration 落地后才体现。 - 风险标记：Blackwell 专属测试依赖 CI，JIT 冷启动 109s 且仅 sm_103a，kimi-k3 分支同步风险，vendored 内核回归风险，AOT 迁移待办

关联脉络

- PR #32624 k3-fused-kda-tp32 (kimi-k3 branch): 提交 b87597b 明确本 PR 移植了该 PR 的 kernel 半部分（TP16/TP32 fused KDA decode），srt/models/kimi_k3.py 等运行时部分留待后续集成 PR。
- PR #33143 Replace Kimi K3 DeepGEMM patch with 0.1.5.post1: 同属 kimi-k3 生态向 main 收敛的工作，涉及 Kimi K3 的依赖与镜像基础设施，与本 PR 的内核移植互为配套。
- PR #32828 [Kimi] Support DCP + DSpark (ported from kimi-k3 branch): 同一"kimi-k3 移植到 main"脉络的投机解码 /kv-cache 侧工作，与本 PR 的 KDA/MTP 内核形成后续集成时的依赖关系。