

PR #37591 完整报告

sgl-project/sglang

[ROCm] Make DSA indexer top-k exact with cooperative selection

合并时间: 2026-09-06 14:55

原文链接: <http://prhub.com.cn/sgl-project/sglang/pull/37591>

执行摘要

本 PR 为 ROCm (AMD GPU) 平台实现了一个精确且高性能的 DSA 索引器 top-k 内核，解决了由自动转换 (hipify) 生成的旧内核在高集中度分数分布下召回率低至 0.53 的严重精度问题。新内核采用 fp32 粗直方图、精确基数 tie 细化和溢出重扫描策略，确保 100% 精确度，同时在多种基准测试中实现 1.5 至 3 倍的性能提升。变更涵盖了核心 HIP 实现、API 包装器、全面测试套件和构建配置。

功能与动机

动机源于一个影响模型正确性的关键 bug：先前的 ROCm 路径使用由 `topk.cu` 自动转换生成的 `topk.hip`，其 fp16 粗直方图键在处理来自 GLM-5.2 解码的高集中度日志时，会将大量元素折叠到同一桶中，导致候选缓冲区溢出并静默丢弃有效元素，测量召回率低至 0.53。为修复此问题并利用 ROCm 硬件特性优化性能，需要实现一个原生的、精确的 top-k 内核。

实现拆解

1. 核心算法 (`dsa_topk_coop.cuh`)：实现了协作式选择内核。算法分为四步：一次粗直方图 pass（使用 fp32 键的高 16 位）、阈值桶搜索、scatter pass（划分赢家 and ties）、以及对 ties 进行基数排序以精确解决。关键设计在于使用 fp32 粗键保留区分度，并在缓冲区溢出时重新扫描整行，保证精确性。实现了单块和行分割两种执行路径，后者在批处理小、行长度大时通过分割行提升 GPU 利用率。
2. 包装器与集成 (`topk.hip`)：提供了与 `topk.cu` 相同的三个 API 入口点。包装器负责设备检查、根据行长度和批处理大小执行行分割决策 (`row_split` 函数)，并调用相应的内核。关键参数如 `kTopK=2048` 被固化。
3. 测试验证 (`test_topk.py`)：新增了严格的精确性测试。`_make_scores` 生成模拟高集中度分布 (banded, narrow, subnormal)。 `assert_exact` 和 `assert_exact_rows` 函数通过精确值多集比较，而非仅索引比较，来确保检测到静默精度损失。测试覆盖了两种执行路径、可变窗口和所有三个 API。
4. 构建系统 (`setup_rocm.py`, `rocm_hipify.py`)：将编译源从 `topk.cu` 更改为 `topk.hip`，并从 HIPify 源列表中移除 `topk.cu`，确保 ROCm 构建使用新的原生实现，且不会在 wheel 构建时被覆盖。

`python/sglang/kernels/aot/include/hip/dsa_topk_coop.cuh`

核心协作式 top-k 内核的完整 HIP 实现，包含精确算法 (fp32 粗直方图、tie 细化、溢出重扫描) 和两种执行路径 (单块、行分割)。是理解本 PR 技术方案的关键。

// 摘自 dsa_topk_coop.cuh, 展示核心算法思想与关键实现单元。

// 有序 32 位键: 翻转符号位使无符号比较等价于浮点比较。

```
__device__ __forceinline__ uint32_t order_key32(float x) {  
    uint32_t bits = __float_as_uint(x);  
    return (bits & 0x80000000u) ? ~bits : (bits | 0x80000000u);  
}
```

// 粗直方图键: 取有序键的高 16 位 (符号位 +8 位指数 +7 位尾数)。

// 截断单调键仍保持单调性, 这是分区正确的基础。

```
__device__ __forceinline__ uint32_t order_key_hi16(float x) {  
    return order_key32(x) >> 16;  
}
```

// 单块内核的启动参数结构

```
struct CoopParams {  
    const float* input; // [B, L] 分数矩阵  
    int32_t* out_idx; // [B, k] 输出索引  
    const int32_t* row_starts; // [B] 行起始偏移 (用于 ragged API)  
    const int32_t* lengths; // [B] 每行的有效长度  
    OutMap map; // 输出映射 (raw/paged/ragged)  
    int64_t stride; // 行步长  
};
```

// 单块内核入口点。遍历整个行, 执行两轮粗直方图 (round0: 12 位, round1: 4 位)

// 和一次 scatter/refine。

```
template <uint32_t kTopK, uint32_t kHistBits, uint32_t kTieCap, uint32_t BlockSize>  
__global__ void coop_topk_kernel(CoopParams p) {  
    // ... 内核实现细节 ...  
    // 核心逻辑:  
    // 1. 线程协作构建全局直方图。  
    // 2. 二分搜索找到阈值桶。  
    // 3. Scatter pass: 将高于阈值的桶直接输出, 阈值桶的成员存入 tie 缓冲区。  
    // 4. 如果 tie 缓冲区溢出 (kTieCap 不够), 则调用 fallback 重扫描整行。  
    // 5. 否则, 对 tie 缓冲区使用完整 fp32 键进行基数排序以精确解决 ties。  
    // 这保证了即使在最坏分布下 (如 88% 的元素落入一个桶), 结果也是精确的。  
}
```

python/sglang/kernels/aot/csrc/elementwise/topk.hip

新 top-k 内核的包装器, 暴露与原有 `topk.cu` 相同的 API, 并实现了关键的行分割决策逻辑和设备检查。是理解内核如何集成到系统中的桥梁。

// 摘自 topk.hip, 展示行分割决策和启动逻辑。

// 行分割策略: 决定是否将一个行分割给多个线程块处理。

// 基于批处理大小和行长度 (使用行步长作为提示) 的启发式。

// 分割可以提升 GPU 利用率, 但会引入额外的全局同步开销。

```
int row_split(int batch, int64_t row_len_hint) {  
    // 允许通过环境变量强制禁用分割 (值设为 0) 以便基准测试。
```

```

if (const char* e = std::getenv("SGL_DSA_TOPK_ROW_SPLIT"); e != nullptr && *e != '\0') {
    const int v = std::atoi(e);
    if (v >= 0) { return v; }
}
// 查询并缓存当前设备的计算单元数量。
// ... 省略设备属性查询 ...
constexpr int64_t kMinRowLen = 65536; // 低于此长度不分割，因为收益小且开销相对大。
constexpr int kMinSplit = 4; // 最少分割块数，低于此则不分割。
constexpr int kMaxSplit = 32; // 最大分割块数，受缓存的计算单元数限制。
if (row_len_hint < kMinRowLen) { return 0; }
const int g = std::min(cached_cu / std::max(batch, 1), kMaxSplit);
return g < kMinSplit ? 0 : g;
}

// 启动函数：根据 row_split 结果选择单块或行分割多块内核。
void launch(...) {
    const int g = row_split(batch, stride);
    if (g == 0) {
        // 单块路径：启动 coop_topk_kernel，每个行块一个线程块。
        coop_topk_kernel<<<batch, kBlock, 0, stream>>>(...);
    } else {
        // 行分割路径：分配工作区，启动多轮内核（hist0, hist1, scatter, refine）。
        // 使用 PyTorch 分配器分配工作区，避免 HIP 图捕获时的地址固化问题。
        at::Tensor ws = at::empty(...);
        coop_mb_hist0<<<grid, kBlock>>>(p, g);
        coop_mb_hist1<<<grid, kBlock>>>(p, g);
        coop_mb_scatter<<<grid, kBlock>>>(p, g);
        coop_mb_refine<<<batch, kBlock>>>(p); // refine 是行级的，不分割。
    }
}

```

评论区精华

讨论揭示了一个关键的设计与集成问题：

1am9trash: “Hi @jiejingzhangamd, I noticed that #36851 turned top-k v2 on by default for GLM on ROCm. Does that mean we never reach this kernel?”

此评论指出，PR #36851 已在 ROCm 上为 GLM 模型默认启用了 **topk_v2**。当环境变量 **SGLANG_OPT_USE_TOPK_V2=1** 被设置时，代码将分派到 **topk_v2** 内核，而非本次 PR 实现的原生内核。这导致本 PR 的内核在默认配置下可能未被使用。作者后续回复提供了更详细的性能对比数据，表明 **topk_v2** 在集群分布下不精确（召回率 0.022），而本内核保持精确（召回率 1.0）。此问题未在讨论中形成最终决策。

风险与影响

1. 正确性 / 性能权衡风险：最大的风险来自 topk_v2 与本内核的 并存。如评论所述，在默认配置下（SGLANG_OPT_USE_TOPK_V2=1），本精确内核可能不被使用，用户可能仍在使用一个已知在某些分布下不精确的内核。这需要产品层面明确配置策略。

2. 启发式决策风险：行分割决策 `row_split` 基于启发式规则（阈值来自 MI355X 测量）。在不同硬件或未来数据分布变化时，该启发式可能非最优，但不会导致不正确结果。
3. 平台特定风险：变更 仅影响 ROCm 构建，CUDA 构建不受影响。这降低了全局风险，但也意味着精度问题的修复仅限于 AMD GPU 用户。

关联脉络

本 PR 与 PR #36851 存在直接关联。PR #36851 在 ROCm 上为 GLM 默认启用了 `topk_v2` 内核，而本 PR 为同一功能点（DSA 索引器 top-k）提供了另一个更精确的内核实现。两者共同构成了该功能在 ROCm 平台上的演进，揭示了正确性、性能和默认配置之间的复杂权衡。未来的整合可能需要考虑如何路由到更合适的内核实现。