

PR #35604 完整报告

sgl-project/sglang

[CPU] Add native CPU kernel for MurmurHash32

合并时间: 2026-09-07 09:10

原文链接: <http://prhub.com.cn/sgl-project/sglang/pull/35604>

执行摘要

- 一句话: 为 MurmurHash32 添加原生 CPU 内核实现与完整测试。
- 推荐动作: 此 PR 是一个有价值的功能补充, 实现了标准算法的原生 CPU 支持。核心变更 (C++ 内核) 逻辑清晰, 测试覆盖了关键场景 (正确性、形状), 并且与项目架构对齐。建议值得精读, 尤其是 `sampling.cpp` 中的内核实现和 `murmur_hash.py` 中的调度逻辑, 它们展示了为现有函数增加平台后端支持的完整模式。主要关注点是确保实现与已有 (如有) GPU 版本或其他实现的行为一致。

功能与动机

实现拆解

1. 核心 C++ 内核实现: 新增 `python/sglang/kernels/aot/csrc/cpu/sampling.cpp` 文件。在其中实现了 `murmur_hash32_cpu` 函数及其内部的并行化内核 `murmur_hash32_kernel_impl`。该内核遵循标准 MurmurHash3 算法, 对输入的 `seed`、`positions` 和 `col_indices` 进行混合哈希计算。函数使用 `at::parallel_for` 实现了数据并行, 以提升在多核 CPU 上的性能。变更还包括添加了必要的输入类型检查 (`CHECK_INPUT`, `CHECK_DIM`) 和数据维度校验。
2. PyTorch 算子注册: 修改 `python/sglang/kernels/aot/csrc/cpu/torch_extension_cpu.cpp`。在 `TORCH_LIBRARY_FRAGMENT` 中添加了对 `murmur_hash32_cpu` 的函数声明和实现注册 (`m.def` 和 `m.impl`), 使其成为可通过 `torch.ops.sgl_kernel` 调用的 CPU 算子。
3. Python 调度入口更新: 修改 `python/sglang/kernels/ops/sampling/murmur_hash.py`。在 `murmur_hash32` 函数入口处增加了设备判断逻辑: 当输入张量在 CPU 上时, 导入 `sgl_kernel` 并直接调用注册的 C++ 算子 `torch.ops.sgl_kernel.murmur_hash32_cpu.default`, 从而绕过可能存在的 Python 路径, 确保使用高性能的原生内核。同时移除了原先为 CPU 准备的纯 Python 实现 (`_murmur_hash32_torch` 等函数)。
4. 单元测试补充: 新增 `test/registered/cpu/test_sampling.py`。包含两个参数化测试用例: `test_murmur_hash32_cpu_known_values` 验证函数在已知输入下输出固定且正确的哈希值; `test_murmur_hash32_cpu_shape_and_dtype` 验证输出张量的形状、设备类型和数据类型正确无误。测试文件还通过 `register_cpu_ci` 注册到 Intel CPU 的 CI 测试流程中。

关键文件:

- python/sglang/kernels/aot/csrc/cpu/sampling.cpp (模块 CPU 内核; 类别 source; 类型 core-logic; 符号 murmur_hash32_cpu, murmur_hash32_kernel_impl, murmur3_mix, fmix32) : 核心变更文件, 包含了 MurmurHash32 算法的完整 C++ CPU 内核实现, 是本次 PR 的价值所在。
- python/sglang/kernels/ops/sampling/murmur_hash.py (模块 调度入口; 类别 infra; 类型 infrastructure; 符号 murmur_hash32) : Python 层的调度入口, 根据设备类型决定调用原生 C++ 内核还是其他路径, 是连接用户接口与底层实现的关键桥梁。
- test/registered/cpu/test_sampling.py (模块 CPU 测试; 类别 test; 类型 test-coverage; 符号 test_murmur_hash32_cpu_known_values, test_murmur_hash32_cpu_shape_and_dtype) : 为新增的 CPU 内核提供测试覆盖, 验证算法正确性和基本接口行为, 是确保质量的关键。
- python/sglang/kernels/aot/csrc/cpu/torch_extension_cpu.cpp (模块 算子注册; 类别 source; 类型 dependency-wiring) : 将新的 C++ 函数注册为 PyTorch 算子, 使其能在 Python 层被 torch.ops 调用, 是完成集成的必要步骤。

关键符号: murmur_hash32, murmur_hash32_cpu, murmur_hash32_kernel_impl, murmur3_mix, fmix32, rotl32

关键源码片段

python/sglang/kernels/aot/csrc/cpu/sampling.cpp

核心变更文件, 包含了 MurmurHash32 算法的完整 C++ CPU 内核实现, 是本次 PR 的价值所在。

```
// C++ 内核: 实现了 MurmurHash32 算法, 并使用 at::parallel_for 进行数据并行处理。
// 输入: 种子、位置和列索引的张量指针, 以及输出矩阵的维度 (n, m)。
// 核心逻辑: 对每个 (行, 列) 元素, 混合其对应的种子、位置和列索引值, 生成一个 32 位哈希值。
template <typename pos_t>
void murmur_hash32_kernel_impl(
    const uint64_t* seed_ptr,
    const pos_t* positions_ptr,
    const int64_t* col_indices_ptr,
    uint32_t* output_ptr,
    int64_t n,
    int64_t m) {
    const int64_t total = n * m;
    // 使用 ATen 的并行 for 循环, 自动根据系统 CPU 核心数分块执行。
    at::parallel_for(0, total, 0, [&](int64_t begin, int64_t end) {
        for (int64_t idx = begin; idx < end; ++idx) {
            // 通过线性索引计算矩阵的行列坐标。
            const int64_t row = idx / m;
            const int64_t col_idx = idx % m;
            const uint64_t seed = seed_ptr[row];
            // 将位置转换为 32 位整数。
            const uint32_t pos = static_cast<uint32_t>(positions_ptr[row]);
            const uint32_t col = static_cast<uint32_t>(col_indices_ptr[col_idx]);
            // 将 64 位种子拆分为两个 32 位块进行处理。
```

```

const uint32_t seed_low = static_cast<uint32_t>(seed);
const uint32_t seed_high = static_cast<uint32_t>(seed >> 32);
uint32_t h = 0;
// 依次混合种子低部、种子高部、位置和列索引。
h = murmur3_mix(h, seed_low);
h = murmur3_mix(h, seed_high);
h = murmur3_mix(h, pos);
h = murmur3_mix(h, col);
h ^= 16u; // 最终长度混合
h = fmix32(h); // 雪崩混合
output_ptr[idx] = h;
}
});
}

```

python/sglang/kernels/ops/sampling/murmur_hash.py

Python 层的调度入口，根据设备类型决定调用原生 C++ 内核还是其他路径，是连接用户接口与底层实现的关键桥梁。

```

def murmur_hash32(seed, positions, col_indices):
    """MurmurHash32 implementation for token hashing."""
    assert len(seed.shape) == 1 and len(col_indices.shape) == 1, (
        f"Inputs must be 1D tensors {seed.shape=} {col_indices.shape=}"
    )
    # 新增：设备类型检查，为 CPU 路径提供原生内核。
    device = seed.device
    if device.type == "cpu":
        import sgl_kernel # noqa: F401
        # 直接调用在 C++ 中注册并实现的算子。
        return torch.ops.sgl_kernel.murmur_hash32_cpu.default(
            seed, positions, col_indices
        )
    n = seed.shape[0]
    m = col_indices.shape[0]
    # ... 后续 GPU 或其他设备的原有逻辑 ...

```

test/registered/cpu/test_sampling.py

为新增的 CPU 内核提供测试覆盖，验证算法正确性和基本接口行为，是确保质量的关键。

```

import torch
from sglang.kernels.ops.sampling.murmur_hash import murmur_hash32

# 测试 1：使用已知的输入和预期的哈希输出值进行精确验证。
@pytest.mark.parametrize("positions_dtype", [torch.int64, torch.uint64])
def test_murmur_hash32_cpu_known_values(positions_dtype):
    seed = torch.tensor([0, 1, 42, 0x123456789ABCDEF0], dtype=torch.uint64)
    positions = torch.tensor([0, 7, 123, 456], dtype=positions_dtype)
    col_indices = torch.tensor([0, 1, 2, 17], dtype=torch.int64)
    actual = murmur_hash32(seed, positions, col_indices)

```

```
# 预期值基于标准 MurmurHash3 算法计算得出。
expected = torch.tensor(
    [
        2167721464, 10027521, 2423355346, 2203067026],
        [3755322398, 4196701286, 1002451629, 183234019],
        [772287619, 548237471, 2740678348, 3656549299],
        [3746406971, 2891010872, 104055988, 3372550890],
    ],
    dtype=torch.uint32,
)
torch.testing.assert_close(actual, expected)
```

```
# 测试 2: 验证输出张量的设备类型、数据类型和形状。
@pytest.mark.parametrize("positions_dtype", [torch.int64, torch.uint64])
def test_murmur_hash32_cpu_shape_and_dtype(positions_dtype):
    seed = torch.tensor([1, 2, 3], dtype=torch.uint64)
    positions = torch.tensor([10, 20, 30], dtype=positions_dtype)
    col_indices = torch.arange(128, dtype=torch.int64)
    actual = murmur_hash32(seed, positions, col_indices)
    assert actual.device.type == "cpu"
    assert actual.dtype == torch.uint32
    assert actual.shape == (3, 128)
```

评论区精华

核心讨论集中在实现方案的演进上。作者最初提交了一个纯 Python 的 PyTorch 回退实现。维护者 mingfeima 明确建议: “prefer to move these into sgl-kernel and provide a C++ interface.” 这推动了方案从便捷的 Python 实现转向性能更优、与项目现有内核架构一致的 C++ 原生实现。后续讨论中, mingfeima 还指出了输入检查的重要性 (“the implementation expects that inputs are contiguous in memory, add `CHECK_INPUT...`”), 以及后续因 CI 命名变更 (PR#37395) 导致的测试注册冲突, 需要作者进行 rebase 和更新。

- 从 Python 回退方案到 C++ 原生内核的方案变更 (design): 作者采纳了建议, 将实现从 Python 迁移到 C++ 并集成了原生内核。
- C++ 内核的输入检查 (correctness): 作者在 `murmur_hash32_cpu` 函数中添加了 `CHECK_INPUT` 和 `CHECK_DIM` 宏调用, 并对数据类型和维度进行了 `TORCH_CHECK`。
- CI 测试注册命名冲突 (other): 维护者 mingfeima 指导作者在 PR#37395 合并后 rebase 并更新测试注册名。

风险与影响

- 风险: 主要风险在于 算法正确性。新增的 C++ 内核实现必须与标准的 MurmurHash3 算法 bit-for-bit 一致。PR 中通过 `test_murmur_hash32_cpu_known_values` 测试用例, 使用硬编码的预期值进行验证, 这有效降低了风险。另一个风险是 输入假设。内核实现假设输入张量是内存连续的 (由 `CHECK_INPUT` 宏保证), 如果传入非连续张量可能引发未定义行为或性能下降, 但当前的输入检查已覆盖此情况。此外, `at::parallel_for` 的使用可能在极端小数据量时引入微小开销, 但鉴于这是针对多核 CPU 设计的, 影响可忽略。

- 影响：此变更对 用户几乎透明，仅影响在 CPU 设备上调用 murmur_hash32 函数的路径。对于依赖该哈希函数的 CPU 采样逻辑，理论上能获得更低的延迟和更高的吞吐。对 系统而言，它扩展了 sgl-kernel 的 CPU 算子库，增强了跨平台一致性。对 团队来说，这引入了一个符合项目模式的 CPU 内核开发范例，但同时增加了内核模块的维护点。
- 风险标记：算法正确性依赖测试覆盖

关联脉络

- PR #37395 [CI] Rename Xeon CI suite to align with naming convention: 该 PR 重命名了 Intel CPU (Xeon) 的 CI 测试套件名称，直接影响了本 PR 中测试文件的 register_cpu_ci 调用，导致本 PR 需要 rebase 和更新以解决命名冲突。