PR #25874 完整报告

sgl-project/sglang

[CPU] add faster KV-cache writes

合并时间：2026-05-25 10:28

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/25874

执行摘要

- 一句话：CPU KV-cache 写入使用 OpenMP + AVX512 专用 kernel 加速
- 推荐动作：值得精读，尤其是 CPU 加速的通用模式：将 ATen 原生接口与 OpenMP 结合，并遵循库注册流程。可作为类似优化的参考。

功能与动机

CPU 上写 KV-cache 时使用 index_put_ 散射效率低下，参照 CUDA store_cache 思路，需要为 CPU 实现一个专用逐行复制内核，以减少内存访问延迟并利用多核和向量化指令。

实现拆解

1. C++ 内核（kvcache.cpp）：定义 store_cache_cpu，接受三维张量 [B,H,D] 并检查最后两维连续；通过 at::parallel_for 并行遍历 batch，内部 copy_stub 优先使用 AVX512 向量化加载 / 存储。
2. 注册与声明（torch_extension_cpu.cpp）：在 sgl_kernel 库中添加 store_cache_cpu 的定义和实现，将 k_cache/v_cache 标记为 Tensor(a!) 以支持图模式。
3. SRT 内存池集成（memory_pool.py）：在 _set_kv_buffer_impl 中添加 if _is_cpu and _cpu_has_amx_support 分支，直接调用 torch.ops.sgl_kernel.store_cache_cpu 跳过 fallback。
4. 公共头增强（common.h）：新增 AT_DISPATCH_REDUCED_FLOATING_TYPES_AND 宏以支持 uint8（FP8 KV）类型分派。
5. 单元测试（test_store_cache.py）：参数化 dtype、head_dim、num_heads、batch_size，对比 kernel 输出与参考赋值，并测试 int32 索引路径。
6. Docker 镜像调整（xeon.Dockerfile）：安装 pytest 以便在容器中运行测试。

关键文件：

- sgl-kernel/csrc/cpu/kvcache.cpp（模块 CPU 内核；类别 source；类型 core-logic；符号 store_cache_cpu, copy_stub, store_cache_kernel_impl）：核心实现：添加 OpenMP 并行 + AVX512 向量化的 KV-cache 写入内核。
- test/registered/cpu/test_store_cache.py（模块 测试；类别 test；类型 test-coverage；符号 _store_cache_cpu, _random_tensor, test_store_cache, test_store_cache_int32_indices）：全面的参数化测试，覆盖各种 dtype/ 形状 / 索引类型。

- sgl-kernel/csrc/cpu/torch_extension_cpu.cpp（模块 内核注册；类别 source；类型 core-logic；符号 store_cache_cpu）：注册新内核到 sgl_kernel 库。
- python/sglang/srt/mem_cache/memory_pool.py（模块 缓存层；类别 source；类型 core-logic；符号 _set_kv_buffer_impl）：在热路径中调用新内核，是实际生效点。
- sgl-kernel/csrc/cpu/common.h（模块 公共头；类别 source；类型 dependency-wiring；符号 AT_DISPATCH_REDUCED_FLOATING_TYPES_AND）：添加宏以支持 uint8 类型分派，使 kernel 可处理 FP8 KV。
- docker/xeon.Dockerfile（模块 部署；类别 infra；类型 infrastructure）：安装 pytest 以便在容器中运行 CPU 测试。

关键符号：store_cache_cpu, copy_stub, store_cache_kernel_impl, _set_kv_buffer_impl, test_store_cache, test_store_cache_int32_indices

关键源码片段

test/registered/cpu/test_store_cache.py

全面的参数化测试，覆盖各种 dtype/ 形状 / 索引类型。

```
import pytest
import torch

# 注册到 CI 框架
from sglang.test.ci.ci_register import register_cpu_ci
register_cpu_ci(est_time=25, suite="base-b-test-cpu")

torch.manual_seed(42)

DEVICE = "cpu"
CACHE_SIZE = 4096

# FP8 KV 以 uint8 存储，例如 float8_e4m3fn/f8_e5m2
DTYPES = [torch.float16, torch.bfloat16, torch.uint8]
DTYPE_IDS = ["float16", "bfloat16", "uint8"]


def _store_cache_cpu(k, v, k_cache, v_cache, indices):
    # 计算 row_dim = H * D，然后调用底层算子
    row_dim = k.size(1) * k.size(2)
    torch.ops.sgl_kernel.store_cache_cpu(k, v, k_cache, v_cache, indices, row_dim)


def _random_tensor(shape, dtype):
    """对于 uint8，使用 randint 代替 randn（后者不支持 Byte）"""
    if dtype == torch.uint8:
        return torch.randint(0, 256, shape, dtype=torch.uint8, device=DEVICE)
    return torch.randn(shape, dtype=dtype, device=DEVICE)

@pytest.mark.parametrize("dtype", DTYPES, ids=DTYPE_IDS)
```

```
@pytest.mark.parametrize("head_dim", [64, 128])
@pytest.mark.parametrize("num_heads", [1, 8, 16, 32])
@pytest.mark.parametrize("batch_size", [1, 7, 133])
def test_store_cache(batch_size, num_heads, head_dim, dtype):
    shape = (batch_size, num_heads, head_dim)
    cache_shape = (CACHE_SIZE, num_heads, head_dim)
    k = _random_tensor(shape, dtype)
    v = _random_tensor(shape, dtype)
    k_cache = _random_tensor(cache_shape, dtype)
    v_cache = _random_tensor(cache_shape, dtype)
    # 随机选择索引（不重复）
    indices = torch.randperm(CACHE_SIZE, device=DEVICE, dtype=torch.int64)[:batch_size]

    # 建立参考结果（使用标准 scatter 赋值）
    k_cache_ref = k_cache.clone()
    v_cache_ref = v_cache.clone()
    k_cache_ref[indices] = k
    v_cache_ref[indices] = v

    # 调用新内核
    _store_cache_cpu(k, v, k_cache, v_cache, indices)

    # 验证整个 cache 与参考完全一致（包括未修改部分）
    assert torch.equal(k_cache, k_cache_ref)
    assert torch.equal(v_cache, v_cache_ref)


@pytest.mark.parametrize("dtype", DTYPES, ids=DTYPE_IDS)
@pytest.mark.parametrize("head_dim", [64, 128])
@pytest.mark.parametrize("num_heads", [1, 8])
@pytest.mark.parametrize("batch_size", [11])
def test_store_cache_int32_indices(batch_size, num_heads, head_dim, dtype):
    # 与 test_store_cache 类似，但索引使用 int32
    shape = (batch_size, num_heads, head_dim)
    cache_shape = (CACHE_SIZE, num_heads, head_dim)
    k = _random_tensor(shape, dtype)
    v = _random_tensor(shape, dtype)
    k_cache = _random_tensor(cache_shape, dtype)
    v_cache = _random_tensor(cache_shape, dtype)
    indices = torch.randperm(CACHE_SIZE, device=DEVICE, dtype=torch.int64)[
        :batch_size
    ].to(torch.int32)

    k_cache_ref = k_cache.clone()
    v_cache_ref = v_cache.clone()
    k_cache_ref[indices.long()] = k
    v_cache_ref[indices.long()] = v

    _store_cache_cpu(k, v, k_cache, v_cache, indices)
```

```
    assert torch.equal(k_cache, k_cache_ref)
    assert torch.equal(v_cache, v_cache_ref)
```

sgl-kernel/csrc/cpu/torch_extension_cpu.cpp

注册新内核到 sgl_kernel 库。

```
// 在文件前部添加声明
// kvcache
void store_cache_cpu(
    const at::Tensor& k,
    const at::Tensor& v,
    const at::Tensor& k_cache,
    const at::Tensor& v_cache,
    const at::Tensor& indices,
    std::optional<int64_t> row_dim);

// 在 TORCH_LIBRARY_FRAGMENT(sgl_kernel, m) 中添加 def/impl
  // ... 其他算子注册 ...

  // kvcache: 标记 k_cache/v_cache 为 Tensor(a!) 以便图模式正确追踪 in-place 修改
  m.def(
      "store_cache_cpu(Tensor k, Tensor v, Tensor(a!) k_cache, Tensor(a!) v_cache, Tensor
      indices, int? row_dim) -> "
      "()");
  m.impl("store_cache_cpu", torch::kCPU, &store_cache_cpu);
```

python/sglang/srt/mem_cache/memory_pool.py

在热路径中调用新内核，是实际生效点。

```
def _set_kv_buffer_impl(
    k: torch.Tensor,
    v: torch.Tensor,
    k_cache: torch.Tensor,
    v_cache: torch.Tensor,
    indices: torch.Tensor,
    row_dim: int, # head_num * head_dim
    store_dtype: torch.dtype,
    device_module: Any,
    alt_stream: Optional[torch.cuda.Stream] = None,
    same_kv_dim: bool = True,
) -> None:
    row_bytes = row_dim * store_dtype.itemsize

    # CUDA / HIP 优化路径（已有）
    if (_is_cuda or _is_hip) and same_kv_dim and can_use_store_cache(row_bytes):
        return store_cache(
            k.view(-1, row_dim), v.view(-1, row_dim),
            k_cache.view(-1, row_dim), v_cache.view(-1, row_dim),
            indices, row_bytes=row_bytes,
```

```
        )

    # 新增 CPU 加速路径：仅在 CPU 且有 AMX 支持时启用
    # 注释：内核使用 OpenMP + AVX512，未来可放宽条件
    if _is_cpu and _cpu_has_amx_support:
        return torch.ops.sgl_kernel.store_cache_cpu(
            k, v, k_cache, v_cache, indices, row_dim,
        )

    # fallback: 使用标准 scatter 赋值（CUDA graph capture 处理异步流）
    from sglang.srt.model_executor.cuda_graph_runner import get_is_capture_mode
    if get_is_capture_mode() and alt_stream is not None:
        current_stream = device_module.current_stream()
        alt_stream.wait_stream(current_stream)
        k_cache[indices] = k
        with device_module.stream(alt_stream):
            v_cache[indices] = v
        current_stream.wait_stream(alt_stream)
    else:
        k_cache[indices] = k
        v_cache[indices] = v
```

评论区精华

审阅中主要讨论点：

- 高：在 torch_extension_cpu.cpp 中应标记 k_cache/v_cache 为 (a!) 以保证图模式正确性（已采纳）。
- 中：_cpu_has_amx_support 条件过于保守，建议对所有 CPU 启用（未采纳，仍保持 AMX 限制）。
- 中：kvcache.cpp 中 row_dim 逻辑可简化（已采纳，仅保留 CHECK_EQ）。
- 中：test_store_cache_int32_indices 应像主测试一样验证整个 cache 未改动的部分（已采纳，已使用 clone + 全量 equal 断言）。
  - 标记 k_cache/v_cache 为 (a!) 以支持图模式 (correctness): 已采纳，最终代码中 m.def 使用了 Tensor(a!) k_cache 和 Tensor(a!) v_cache。
  - 移除 _cpu_has_amx_support 限制 (performance): 未采纳，最终代码仍保留 _cpu_has_amx_support 条件。
  - 简化 kvcache.cpp 中 row_dim 处理逻辑 (style): 已采纳，最终代码仅保留 if (row_dim.has_value()) { CHECK_EQ(...); }。
  - 测试断言健壮性：验证整个 cache 未被修改 (testing): 已采纳，最终测试代码中 int32 测试也使用了 clone + 全量 equal。

风险与影响

- 风险：

1. 兼容性风险：当前仅在有 AMX 支持的 CPU 上启用，非 AMX CPU 仍回退到慢速路径，未充分利用优化。
2. 测试覆盖风险：测试覆盖了常见 dtype/ 形状，但未测试非连续张量（kernel 会抛异常），行为符合预期。
3. 核心路径变更：memory_pool.py 在每次 KV 写入新增分支判断，但开销极小。
4. 文档缺失：PR checklist 未勾选 documentation，新内核缺少用户可见的说明。 - 影响：影响范围限于 CPU 后端推理请求的 KV-cache 写入操作，预期显著降低写入延迟（尤其 batch_size 较大时）。用户无感知，自动生效。团队需维护新增的 C++ 内核及其注册。 - 风险标记：核心路径变更，AMX 限制可能过严，缺少文档更新

关联脉络

- 暂无明显关联 PR