PR #24755 完整报告

sgl-project/sglang

Optimize large add_constant tensors

合并时间：2026-05-30 22:25

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/24755

执行摘要

- 一句话：向量化 add_constant 大张量，H200 加速 35%
- 推荐动作：值得精读。向量化 kernel 的设计（架构感知向量宽度、对齐检查、阈值判断、fallback 路径）是 CUDA kernel 优化的典型模式。benchmark 的实现也值得参考，可以复用到其他 kernel。

功能与动机

优化大张量 add_constant 性能瓶颈，在 H200 上 4M 元素耗时从 15.6us 降至 10.1us (35% 加速)，B200 上 16M 元素从 37us 降至 27.6us。

实现拆解

1. 分析原标量 kernel 在大张量下的性能瓶颈，确定向量化优化方向。
2. 在 python/sglang/jit_kernel/csrc/add_constant.cuh 中添加向量化 kernel add_constant_vectorized_kernel，使用 device::AlignedVector 加载存储，基于架构选择向量宽度（device::kMaxVecBytes 决定 16 或 32 字节）。
3. 在入口函数 add_constant 中添加对齐检查和大小阈值判断（kVectorizedMinElements = 1M），符合条件的调用向量化 kernel，否则回退标量 kernel。
4. 新增 python/sglang/jit_kernel/benchmark/bench_add_constant.py 使用 Triton Benchmark 框架，在多种尺寸上对比 JIT module / JIT wrapper / PyTorch 三大实现。
5. 在 python/sglang/jit_kernel/tests/test_add_constant.py 中补充大张量对齐 / 未对齐组合的测试用例，确保向量化路径的正确性。

关键文件：

- python/sglang/jit_kernel/csrc/add_constant.cuh（模块 JIT 内核；类别 source；类型 core-logic；符号 add_constant_vectorized_kernel, is_aligned_for_vector, add_constant）：核心向量化 kernel 实现，引入架构感知向量宽度和对齐检查，实现大张量加速路径与标量 fallback。
- python/sglang/jit_kernel/benchmark/bench_add_constant.py（模块 JIT 内核；类别 source；类型 benchmark；符号 benchmark, fn）：新增 benchmark 脚本，用于比较 JIT module、JIT wrapper 和 PyTorch 原生加法性能，覆盖各种尺寸。

- python/sglang/jit_kernel/tests/test_add_constant.py（模块 常数加法；类别 test；类型 test-coverage；符号 test_add_constant_unaligned_input, test_add_constant_large_aligned_input, test_add_constant_large_unaligned_input）：增加了大张量对齐 / 未对齐输入测试，确保向量化路径正确性。

关键符号：add_constant_vectorized_kernel, is_aligned_for_vector, add_constant, benchmark, test_add_constant_unaligned_input, test_add_constant_large_aligned_input, test_add_constant_large_unaligned_input

关键源码片段

python/sglang/jit_kernel/csrc/add_constant.cuh

核心向量化 kernel 实现，引入架构感知向量宽度和对齐检查，实现大张量加速路径与标量 fallback。

```
// 根据编译时架构选择向量宽度：Blackwell 为 32 字节，其余为 16 字节
constexpr size_t kVectorBytes = device::kMaxVecBytes;
static_assert(kVectorBytes % sizeof(int32_t) == 0,
              "Vector byte width must contain whole int32_t elements");
constexpr size_t kElementsPerVector = kVectorBytes / sizeof(int32_t);

// 检查指针是否满足对齐要求
template <typename Vector>
bool is_aligned_for_vector(const int32_t* ptr) {
  return reinterpret_cast<uintptr_t>(ptr) % alignof(Vector) == 0;
}

// 向量化 kernel：一次性处理 kElementsPerVector 个元素
template <int32_t kConstant, size_t kElementsPerVector>
__global__ void add_constant_vectorized_kernel(int32_t* dst,
                                                const int32_t* src,
                                                size_t length) {
  using Vector = device::AlignedVector<int32_t, kElementsPerVector>;
  const size_t work_idx = blockIdx.x * blockDim.x + threadIdx.x;
  const size_t vector_count = length / kElementsPerVector;
  const size_t tail_start = vector_count * kElementsPerVector;

  // 处理完整向量部分
  if (work_idx < vector_count) {
    auto values = device::load_as<Vector>(src, work_idx);
    #pragma unroll
    for (size_t i = 0; i < kElementsPerVector; ++i) {
      values[i] += kConstant;
    }
    device::store_as<Vector>(dst, values, work_idx);
  } else {
    // 处理尾部剩余元素（标量）
    const size_t tail_idx = tail_start + work_idx - vector_count;
    if (tail_idx < length) {
```

```
      dst[tail_idx] = src[tail_idx] + kConstant;
    }
  }
}

// 主入口：根据大小和对齐决定 kernel 选择
void add_constant(tvm::ffi::TensorView dst, tvm::ffi::TensorView src) {
  // ... 参数解析 ...
  const auto* src_ptr = static_cast<const int32_t*>(src.data_ptr());
  auto* dst_ptr = static_cast<int32_t*>(dst.data_ptr());
  using Vector = device::AlignedVector<int32_t, kElementsPerVector>;
  const bool is_vector_aligned = is_aligned_for_vector<Vector>(src_ptr) &&
                                 is_aligned_for_vector<Vector>(dst_ptr);
  // 只有大张量且对齐时才使用向量化路径
  if (num_elements >= kVectorizedMinElements && is_vector_aligned) {
    const size_t grid_size = div_ceil(num_elements / kElementsPerVector +
                                      num_elements % kElementsPerVector,
                                      kBlockSize);
    LaunchKernel(grid_size, kBlockSize, device,
                 add_constant_vectorized_kernel<kConstant, kElementsPerVector>,
                 dst_ptr, src_ptr, num_elements);
  } else {
    const size_t grid_size = div_ceil(num_elements, kBlockSize);
    LaunchKernel(grid_size, kBlockSize, device,
                 add_constant_kernel<kConstant>,
                 dst_ptr, src_ptr, num_elements);
  }
}
```

python/sglang/jit_kernel/benchmark/bench_add_constant.py

新增 benchmark 脚本，用于比较 JIT module、JIT wrapper 和 PyTorch 原生加法性能，覆盖各种尺寸。

```
# bench_add_constant.py
from sglang.jit_kernel.add_constant import _jit_add_constant_module, add_constant
from sglang.jit_kernel.benchmark.utils import (
    DEFAULT_DEVICE, get_benchmark_range, run_benchmark_no_cudagraph,
)

# 注册 CI 任务，大 suite 用于基准测试
register_cuda_ci(est_time=15, suite="base-b-kernel-benchmark-1-gpu-large")

CONSTANT = 7 # 测试用常数值

# 定义测试尺寸范围：全量和 CI 精简版
SIZE_LIST = get_benchmark_range(
    full_range=[128, 1024, 1025, 4096, 4097, 65536, 2**20, 2**22, 2**24],
    ci_range=[4096, 2**20],
)
```

```
@triton.testing.perf_report(
    triton.testing.Benchmark(
        x_names=["size"],
        x_vals=SIZE_LIST,
        line_arg="provider",
        line_vals=["jit_module", "jit_wrapper", "torch"],
        line_names=["JIT module", "JIT wrapper", "PyTorch"],
        styles=[("blue", "-"), ("orange", "-"), ("green", "--")],
        ylabel="us",
        plot_name="add-constant-performance",
    )
)
def benchmark(size: int, provider: str):
    src = torch.arange(size, dtype=torch.int32, device=DEFAULT_DEVICE)
    if provider == "jit_module":
        # 直接调用编译模块的接口
        dst = torch.empty_like(src)
        module = _jit_add_constant_module(CONSTANT)
        def fn():
            module.add_constant(dst, src)
    elif provider == "jit_wrapper":
        # 通过包装函数 add_constant
        def fn():
            add_constant(src, CONSTANT)
    else:
        # PyTorch 原生加法作为 baseline
        def fn():
            src + CONSTANT
    return run_benchmark_no_cudagraph(fn)

if __name__ == "__main__":
    benchmark.run(print_data=True)
```

python/sglang/jit_kernel/tests/test_add_constant.py

增加了大张量对齐 / 未对齐输入测试，确保向量化路径正确性。

```
# test_add_constant.py
# ... 原有测试 ...

def test_add_constant_unaligned_input() -> None:
    """验证未对齐输入的向量化路径正确性。"""
    src = torch.arange(0, 4098, dtype=torch.int32, device="cuda")[1:] # 起始地址不对齐
    dst = add_constant(src, 7)
    assert torch.all(dst == src + 7)

@pytest.mark.parametrize("size", [2**20, 2**20 + 3])
def test_add_constant_large_aligned_input(size: int) -> None:
    """验证大张量对齐输入的正确性。"""
    src = torch.arange(0, size, dtype=torch.int32, device="cuda")
```

```
    dst = add_constant(src, -3)
    assert torch.all(dst == src - 3)

def test_add_constant_large_unaligned_input() -> None:
    """验证大张量未对齐输入的正确性。"""
    src = torch.arange(0, 2**20 + 4, dtype=torch.int32, device="cuda")[1:]
    dst = add_constant(src, 7)
    assert torch.all(dst == src + 7)
```

评论区精华

PR body 提供了详尽的 H200/B200 benchmark 数据，明确显示大张量 30-35% 加速，小张量性能中性（变化在 ±2-3% 内）。Review 由 yuan-luo 批准，无额外讨论。

- Large tensor performance improvement (performance): 确认大张量显著加速，小张量性能中性，优化有效。

风险与影响

- 风险：向量化路径要求输入输出指针满足对齐（16 或 32 字节），未对齐时会自动回退标量路径，功能正确性由新增的未对齐测试覆盖。无性能回归风险（小张量保持标量），无安全和兼容性风险。
- 影响：直接影响使用 add_constant 的所有场景，例如 attention 中的 bias 加法、layer normalization 中的常数加法等。由于是底层 kernel 优化，用户无需修改代码即可受益。预估对大 batch size 推理任务有积极影响。
- 风险标记：对齐要求，小张量性能中性，回退标量路径

关联脉络

- 暂无明显关联 PR