PR #45269 完整报告

vllm-project/vllm

[CPU][RISC-V] Add RVV path for W4A8 INT4 GEMM

合并时间：2026-06-25 16:18

原文链接：http://prhub.com.cn/vllm-project/vllm/pull/45269

PR 分析报告：为 RISC-V CPU 添加 W4A8 INT4 GEMM RVV 路径

执行摘要

本 PR 为 vLLM 的 CPU W4A8 INT4 GEMM 算子新增 RISC-V Vector（RVV）后端，通过向量化实现 5 倍性能提升。改动集中在 C++ 内核层（gemm_int4.cpp、vec.h、cpu_types_riscv_defs.hpp），并辅以 Python 调度和 cmake 构建的少量适配。使用条件编译和标量回退保证了兼容性，无 breaking change。

功能与动机

此前 CPU W4A8 算子仅拥有针对 x86 AVX512 的优化路径，RISC-V 平台只能退回到逐元素标量循环，性能低下。PR 的目标是利用 RVV intrinsics 为 RISC-V 提供等效的向量化加速，同时复用现有的 VNNI4 打包权重布局和 32 列块形状，以最小改动获得最大收益。

实现拆解

1. 基础类型与编译宏：在 cpu_types_riscv_defs.hpp 中增加 LMUL_64 宏和 fixed_u8x8_t / fixed_i8x8_t 等固定向量类型，并新增对 VLEN=128/256 的 LMUL 映射。在 vec.h 中通过检测 __riscv_v_min_vlen 定义 CPU_CAPABILITY_RVV，并包含上述头文件。
2. RVV 微内核函数：在 gemm_int4.cpp 的 #if defined(CPU_CAPABILITY_RVV) 分支内实现五个函数：load_uint4_as_int8_rvv（带跨步加载的解包）、gemm_accum_uint8_int8_rvv（int8×uint8 乘加）、gemm_accum_uint4_rvv（解包→减 zero-point→乘加）、_dequant_and_store_rvv（补偿→转浮点→缩放→累加）以及主循环 _dequant_gemm_accum_rvv。
3. 集成到现有调度：在 _dequant_gemm_accum 函数中添加 RVV 分支：当 CPU_CAPABILITY_RVV 定义且 N=32 时调用 _dequant_gemm_accum_rvv，否则重用原标量实现。
4. Python 层使能：在 cpu.py 的 process_weights_after_loading 中增加 supports_riscv 检测，使 use_w4a8 在满足 AMX 或 RISC-V 条件时开启，从而进入 RVV 或 AVX512 路径。
5. 构建系统调整：修正 cmake 中 VLLM_RVV_VLEN 的合法性检查（允许 0 表示强制标量），并完善注释描述零值含义。

csrc/cpu/sgl-kernels/gemm_int4.cpp

核心修改，新增 RVV 实现和调度分支

```
// RVV 核心实现：从 VNNI4 布局加载 8 个 int4 权重并转换为 int8
// 使用 strided load（vlse8）高效解包，然后通过移位和掩码提取低 4 位
// group 参数选择 8 列组（0-3），对应 32 列块中的子列位置
// 返回 int8x8 向量，其中每个元素是解包后的权重值

template <int64_t N, int64_t ldb, int group>
inline fixed_i8x8_t load_uint4_as_int8_rvv(const uint8_t* __restrict__ B, int64_t k) {
  constexpr int64_t n_group_size = 8; // 每组 8 列
  constexpr int64_t vnni_size = 4; // VNNI 维度
  static_assert(N == 32); // 当前仅支持 32 列块
  static_assert(ldb == N / 2); // 每行步长等于一半列数
  static_assert(group >= 0 && group < N / n_group_size); // 4 个组

  const int64_t ki = k % vnni_size;
  const int64_t k_base = k - ki;
  constexpr int64_t packed_group = group / 2;
  const uint8_t* packed_ptr = B + k_base * ldb + packed_group * n_group_size * vnni_size + ki;

  // 跨步加载：每隔 vnni_size 个字节取一个 vnni_size 字节元素，共 n_group_size 个
  fixed_u8x8_t packed = RVVI(__riscv_vlse8_v_u8, LMUL_64)(packed_ptr, vnni_size, n_group_
  size);
  // 高 4 位组（group 奇数）则需要右移 4 位
  if constexpr (group % 2 == 1) {
    packed = RVVI(__riscv_vsrl_vx_u8, LMUL_64)(packed, 4, n_group_size);
  }
  fixed_u8x8_t nibbles = RVVI(__riscv_vand_vx_u8, LMUL_64)(packed, 0x0f, n_group_size);
  return RVVI4(__riscv_vreinterpret_v_u8, LMUL_64, _i8, LMUL_64)(nibbles);
}

// int8 与 uint8 的乘加：先将 int8 扩展到 int16，再用 vwmacc 与 uint16 乘加
inline fixed_i32x8_t gemm_accum_uint8_int8_rvv(fixed_i32x8_t acc, uint8_t a, fixed_i8x8_t b) {
  constexpr int64_t vl = 8;
  fixed_i16x8_t b_i16 = RVVI(__riscv_vsext_vf2_i16, LMUL_128)(b, vl);
  return RVVI(__riscv_vwmacc_vx_i32, LMUL_256)(acc, static_cast<int16_t>(a), b_i16, vl);
}

// 解包 + 减 zero-point + 乘加（完整的 int4 权重计算）
template <int64_t N, int64_t ldb, int group>
inline fixed_i32x8_t gemm_accum_uint4_rvv(
    fixed_i32x8_t acc,
    const uint8_t* __restrict__ B,
    const int8_t* __restrict__ qzeros_b,
    uint8_t a,
    int64_t k) {
  constexpr int64_t n_group_size = 8;
  fixed_i8x8_t b = load_uint4_as_int8_rvv<N, ldb, group>(B, k);
  // 加载对应组的 zero-point（每组 8 个 int8）
```

```
  fixed_i8x8_t qzeros =
      RVVI(__riscv_vle8_v_i8, LMUL_64)(qzeros_b + group * n_group_size, n_group_size);
  // 权重值减去 zero-point
  b = RVVI(__riscv_vsub_vv_i8, LMUL_64)(b, qzeros, n_group_size);
  return gemm_accum_uint8_int8_rvv(acc, a, b);
}

// 对 32 列块中每组 8 列进行反量化（补偿、缩放）并累加到浮点输出
// 接受 int32 累加器，消除激活 zero-point 贡献，转换为 float，乘以 scales，再与旧值相加
template <int group>
inline void _dequant_and_store_rvv(
    float* __restrict__ C,
    fixed_i32x8_t acc,
    const float* __restrict__ scales_a,
    const int32_t* __restrict__ qzeros_a,
    const float* __restrict__ scales_b,
    const int32_t* __restrict__ compensation,
    int64_t m,
    int64_t ldc) {
  constexpr int64_t n_group_size = 8;
  constexpr int64_t n = group * n_group_size;
  constexpr int64_t vl = n_group_size;

  // 加载 compensation（预先计算的激活 zero-point * 权重的逐列和）
  fixed_i32x8_t comp = RVVI(__riscv_vle32_v_i32, LMUL_256)(compensation + n, vl);
  // 激活 zero-point 补偿
  fixed_i32x8_t zp_comp = RVVI(__riscv_vmul_vx_i32, LMUL_256)(comp, qzeros_a[m], vl);
  acc = RVVI(__riscv_vsub_vv_i32, LMUL_256)(acc, zp_comp, vl);

  // 转 float 并乘以激活 scale 和权重 scale
  fixed_fp32x8_t acc_f = RVVI(__riscv_vfcvt_f_x_v_f32, LMUL_256)(acc, vl);
  acc_f = RVVI(__riscv_vfmul_vf_f32, LMUL_256)(acc_f, scales_a[m], vl);
  fixed_fp32x8_t scale_b = RVVI(__riscv_vle32_v_f32, LMUL_256)(scales_b + n, vl);
  acc_f = RVVI(__riscv_vfmul_vv_f32, LMUL_256)(acc_f, scale_b, vl);

  // 累加到已有的浮点缓冲区（可能已包含 bias）
  float* c_ptr = C + m * ldc + n;
  fixed_fp32x8_t c_old = RVVI(__riscv_vle32_v_f32, LMUL_256)(c_ptr, vl);
  fixed_fp32x8_t c_new = RVVI(__riscv_vfadd_vv_f32, LMUL_256)(c_old, acc_f, vl);
  RVVI(__riscv_vse32_v_f32, LMUL_256)(c_ptr, c_new, vl);
}
```

csrc/cpu/cpu_types_riscv_defs.hpp

新增 LMUL_64 及 int8/uint8/int16 固定向量类型，为 RVV 内核提供基础类型

```
// VLEN 到 LMUL 的映射：LMUL_64 用于 8 元素 int8/uint8 向量
// 在 VLEN=128 时对应 mf2，VLEN=256 时对应 mf4
#if __riscv_v_min_vlen == 128
  #define LMUL_64 mf2
  #define LMUL_128 m1
```

```
  #define LMUL_256 m2
  #define LMUL_512 m4
  #define LMUL_1024 m8
  #define BOOL_256 b16
  #define BOOL_512 b8
#elif __riscv_v_min_vlen == 256
  #define LMUL_64 mf4
  #define LMUL_128 mf2
  #define LMUL_256 m1
  #define LMUL_512 m2
  #define LMUL_1024 m4
  #define BOOL_256 b32
  #define BOOL_512 b16
#else
  #error "cpu_types_riscv_defs.hpp: unsupported __riscv_v_min_vlen"
#endif

// 关键固定长度向量类型定义
// uint8 / int8 （8 元素）
typedef RVVTYPE(vuint8, LMUL_64, _t) fixed_u8x8_t
    __attribute__((riscv_rvv_vector_bits(64)));
typedef RVVTYPE(vint8, LMUL_64, _t) fixed_i8x8_t
    __attribute__((riscv_rvv_vector_bits(64)));

// int16 （8 元素，LMUL_128）
typedef RVVTYPE(vint16, LMUL_128, _t) fixed_i16x8_t
    __attribute__((riscv_rvv_vector_bits(128)));
```

评论区精华

无讨论。

风险与影响

- 风险：缺少针对 RVV 内核的单元测试，正确性依赖集成测试；RVV 代码静态断言 N=32，未来扩展块形状时需要修改。
- 影响：RISC-V 用户性能提升约 5 倍；其他平台无影响；构建系统新增 VLLM_RVV_VLEN 选项；团队未来需维护 RVV 内核。

关联脉络

本 PR 是 vLLM 在 RISC-V 平台上的第一个专用内核优化，与此前 CPU 的 AVX512 路径（及相关的 cmake 基础设施）构成互补。暂无其他直接关联的 PR。