PR #46135 完整报告

vllm-project/vllm

[HARDWARE][POWER] Enable fp16 support for PowerPC

合并时间：2026-06-23 21:24

原文链接：http://prhub.com.cn/vllm-project/vllm/pull/46135

执行摘要

本 PR 为 PowerPC (ppc64le) 架构启用 fp16 (Half) 数据类型推理支持。通过 VSX 向量指令实现 IEEE 754 兼容的软件转换函数，并在旋转位置编码、注意力 kernel 和 MLA 解码等核心路径中添加特化代码。影响范围仅限于 PowerPC 平台，风险较低，但缺少显式测试覆盖。

功能与动机

根据 PR 描述，目的是为 PowerPC 系统启用 FP16（半精度）推理支持。此前 PowerPC 平台仅支持 bfloat16 和 float32，无法利用半精度模型的内存和带宽优势。

实现拆解

实现分为以下 5 个步骤：

1. 向量化转换函数（core layer）— 在 csrc/cpu/cpu_types_vsx.hpp 中新增 fp16_to_fp32_bits 和 fp32_to_fp16_bits 两个内联函数，纯位操作实现 IEEE 754 转换。同时定义了 FP16Vec8 向量类型，提供加载 / 保存封装。
2. 宏扩展 (dispatch)— 将 VLLM_DISPATCH_CASE_FLOATING_TYPES 宏扩展为同时包含 Float、BFloat16 和 Half，使得所有使用该宏的 kernel 自动获得 fp16 分派能力。
3. 旋转位置编码特化— 在 csrc/cpu/pos_encoding.cpp 中对 c10::Half 显式特化 rotary_embedding_impl：向量部分使用 FP16Vec8 加载，转 FP32Vec8 计算后存回；尾部残差用标量处理。
4. 注意力 kernel 适配— 在 csrc/cpu/cpu_attn_vsx.hpp 中添加 load_row8_B_as_f32<c10::Half> 特化，使注意力计算支持 fp16 类型的 KV cache。同时，在 csrc/cpu/mla_decode.cpp 中删除之前为 PowerPC 单独定义的预处理分支，统一使用与 x86 相同的向量类型策略；在 csrc/cpu/cpu_attn_impl.hpp 中移除 !defined(__powerpc__) 保护条件。
5. 平台配置— 在 vllm/platforms/cpu.py 中，为 PowerPC 架构的 supported_dtypes 加入 torch.float16，使得服务端能识别并允许 fp16 模型加载。

csrc/cpu/cpu_types_vsx.hpp

核心转换函数和向量类型定义，是 fp16 支持的基石

```
// 文件：csrc/cpu/cpu_types_vsx.hpp (head 版本)
// 已扩展类型分发宏，使所有使用它的 kernel 支持 fp16
#define VLLM_DISPATCH_CASE_FLOATING_TYPES(...) \
  AT_DISPATCH_CASE(at::ScalarType::Float, __VA_ARGS__) \
```

```
  AT_DISPATCH_CASE(at::ScalarType::BFloat16, __VA_ARGS__) \
  AT_DISPATCH_CASE(at::ScalarType::Half, __VA_ARGS__)

namespace {

/* 将 4 个 FP16 值（每个 16 位）转换为 4 个 FP32 值（每个 32 位）
 * 输入：x 是 4 个 16 位 FP16 打包在 __vector unsigned int 的低 16 位中 */
FORCE_INLINE __vector float fp16_to_fp32_bits(__vector unsigned int x) {
  const __vector unsigned int mask_sign = {0x8000, 0x8000, 0x8000, 0x8000};
  const __vector unsigned int mask_exp = {0x7C00, 0x7C00, 0x7C00, 0x7C00};
  const __vector unsigned int mask_mant = {0x03FF, 0x03FF, 0x03FF, 0x03FF};
  const __vector unsigned int bias_adj = {112, 112, 112, 112}; // 127 - 15 = 112
  const __vector unsigned int exp_max_fp16 = {0x1F, 0x1F, 0x1F, 0x1F};
  const __vector unsigned int exp_max_fp32 = {0xFF, 0xFF, 0xFF, 0xFF};

  __vector unsigned int s = (x & mask_sign) << 16;
  __vector unsigned int e = (x & mask_exp) >> 10;
  __vector unsigned int m = (x & mask_mant) << 13;

  __vector __bool int is_nan_inf = vec_cmpeq(e, exp_max_fp16);
  __vector unsigned int e_normal = e + bias_adj;
  e = vec_sel(e_normal, exp_max_fp32, is_nan_inf);

  return (__vector float)(s | (e << 23) | m);
}

/* 将 4 个 FP32 值转换为 4 个 FP16 值，返回打包在 __vector unsigned int 的低 16 位中
 * 包含 rounding 和溢出处理，IEEE 754 兼容 */
FORCE_INLINE __vector unsigned int fp32_to_fp16_bits(__vector float f_in) {
  // ... 完整实现请参考 PR head 版本
  // 关键步骤：解析符号、指数、尾数，偏差调整，rounding to nearest even
}

} // anonymous namespace
```

csrc/cpu/pos_encoding.cpp

为旋转位置编码添加 fp16 特化，使 RoPE 支持 Half 类型

```
// 文件：csrc/cpu/pos_encoding.cpp (head 版本)
// FP16 特化的旋转位置编码实现，使用 FP16Vec8 加载、FP32Vec8 计算、再转回 Half 存储
template <>
void rotary_embedding_impl<c10::Half>(
    const int64_t* __restrict__ positions,
    c10::Half* __restrict__ query,
    c10::Half* __restrict__ key,
    const c10::Half* __restrict__ cos_sin_cache,
    const int rot_dim, const int64_t query_stride, const int64_t key_stride,
    const int num_heads, const int num_kv_heads, const int head_size,
    const int num_tokens) {
```

```
using scalar_vec_t = vec_op::FP16Vec8;
constexpr int VEC_ELEM_NUM = scalar_vec_t::get_elem_num();
const int embed_dim = rot_dim / 2;
bool flag = (embed_dim % VEC_ELEM_NUM == 0);
const int loop_upper = flag ? embed_dim : embed_dim - VEC_ELEM_NUM;

auto compute_loop = [&](const int64_t token_head, const c10::Half* cache_ptr,
                        c10::Half* qk) {
  int j = 0;
  for (; j < loop_upper; j += VEC_ELEM_NUM) {
    const int rot_offset = j;
    const int x_index = rot_offset;
    const int y_index = embed_dim + rot_offset;
    const int64_t out_x = token_head + x_index;
    const int64_t out_y = token_head + y_index;

    // 以 FP16Vec8 加载 cache 和 qk 数据
    const vec_op::FP16Vec8 cos_fp16(cache_ptr + x_index);
    const vec_op::FP16Vec8 sin_fp16(cache_ptr + y_index);
    const vec_op::FP16Vec8 q_x_fp16(qk + out_x);
    const vec_op::FP16Vec8 q_y_fp16(qk + out_y);

    // 转成 FP32Vec8 进行计算
    const vec_op::FP32Vec8 fp32_cos(cos_fp16);
    const vec_op::FP32Vec8 fp32_sin(sin_fp16);
    const vec_op::FP32Vec8 fp32_q_x(q_x_fp16);
    const vec_op::FP32Vec8 fp32_q_y(q_y_fp16);

    auto out1 = fp32_q_x * fp32_cos - fp32_q_y * fp32_sin;
    auto out2 = fp32_q_y * fp32_cos + fp32_q_x * fp32_sin;

    // 转回 FP16 并保存
    vec_op::FP16Vec8(out1).save(qk + out_x);
    vec_op::FP16Vec8(out2).save(qk + out_y);
  }
  // 尾部残差处理（标量）
  if (!flag) {
    for (; j < embed_dim; ++j) {
      const int x_index = j;
      const int y_index = embed_dim + j;
      const int64_t out_x = token_head + x_index;
      const int64_t out_y = token_head + y_index;
      const float fp32_cos = static_cast<float>(cache_ptr[x_index]);
      const float fp32_sin = static_cast<float>(cache_ptr[y_index]);
      const float fp32_q_x = static_cast<float>(qk[out_x]);
      const float fp32_q_y = static_cast<float>(qk[out_y]);
      qk[out_x] = static_cast<c10::Half>(fp32_q_x * fp32_cos - fp32_q_y * fp32_sin);
      qk[out_y] = static_cast<c10::Half>(fp32_q_y * fp32_cos + fp32_q_x * fp32_sin);
    }
```

```
    }
  };

  // 外层循环处理所有 tokens 和 heads
#pragma omp parallel for
  for (int token_idx = 0; token_idx < num_tokens; ++token_idx) {
    int64_t pos = positions[token_idx];
    const c10::Half* cache_ptr = cos_sin_cache + pos * rot_dim;
    for (int i = 0; i < num_heads; ++i) {
      const int head_idx = i;
      const int64_t token_head = token_idx * query_stride + head_idx * head_size;
      compute_loop(token_head, cache_ptr, query);
    }
    if (key != nullptr) {
      for (int i = 0; i < num_kv_heads; ++i) {
        const int head_idx = i;
        const int64_t token_head = token_idx * key_stride + head_idx * head_size;
        compute_loop(token_head, cache_ptr, key);
      }
    }
  }
}
```

评论区精华

无 Review 评论。PR 由 Maintainer bigPYJ1151 直接批准合并。

风险与影响

- 风险：
  - 缺少针对 PowerPC + fp16 的端到端测试，可能隐藏 kernel 特化遗漏或转换函数的边界条件问题。
  - 代码中使用 VSX 位操作，其正确性依赖编译器对 Altivec 内建函数的实现，但在 GCC/LLVM 上较为成熟。
  - 所有修改均条件编译或特化，不影响其他架构。
- 影响：
  - PowerPC 用户受益，可运行 fp16 模型，降低内存带宽与占用。
  - 其他架构无行为变化。
  - 团队需要维护 PowerPC 特有的向量类型代码，但已有良好隔离。

关联脉络

未在历史 PR 中识别到直接关联的变更。此 PR 是 PowerPC 架构功能补齐的一步，未来可能需要跟进 fp16 GEMM 或更多算子的显式优化。